Journal of Artificial Intelligence and Data Mining (JAIDM), Vol. 11, No. 4, 2023, 639-648.

Shahrood University of Technology

Journal of Artificial Intelligence and Data Mining (JAIDM)
Journal homepage: http://jad.shahroodut.ac.ir

Research paper

Parallel Incremental Mining of Regular-Frequent Patterns from WSNs Big Data

Sadegh Rahmaniboldaji[1*], Mehdi Bateni[2] and Mahmood Mortazavi Dehkordi[3]

1. Computer Engineering, Sheikh Bahaei University, Isfahan, Iran.
2. Computer Science and Computer Engineering, University of Isfahan, Khansar Campus, Isfahan, Iran
3.MSE, University Canada West, Vancouver, Canada.

Article Info

Article History:
Received 05 May 2023
Revised 08 August 2023
Accepted 15 December 2023

DOI:10.22044/jadm.2023.13034.2448

Keywords:
Regular-frequent Pattern, Big Streaming Data, Parallel Algorithm, Incremental Mining

**Corresponding author: sadeghrahmani@shbu.ac.ir (S. Rahmaniboldaji).*

Abstract

Efficient regular-frequent pattern mining from sensors-produced data has become a challenge. The large volume of data leads to prolonged runtime, thus delaying vital predictions and decision-makings, which need an immediate response. Thus, using big data platforms and parallel algorithms is an appropriate solution. Additionally, an incremental technique is more suitable to mine patterns from big data streams than static methods. This study presents an incremental parallel approach and compact tree structure for extracting regular-frequent patterns from the data of wireless sensor networks. Furthermore, fewer database scans have been performed in an effort to reduce the mining runtime. This study was performed on Intel 5-day and 10-day datasets with 6, 4, and 2 nodes clusters. The findings show that the runtime was improved in all 3 cluster modes by 14, 18, and 34% for the 5-day dataset and by 22, 55, and 85% for the 10-day dataset, respectively.

1. Introduction

Pattern mining of big data using traditional techniques is not efficient. Some traditional methods use static mining and unparalleled platforms for mining process, in which streams of data or updates to data are not considered. Thus, in presence of data stream and dire situations such as decision-making in the case of critical patient's conditions, applying incremental pattern mining as well as speeding up the pattern mining process are necessary. Mining the patterns is performed through data mining techniques [1] like the growth-based methods [2]. For this aim, various growth-based methods use compact tree structures to efficiently extract all the information from databases for fast mining of regular or frequent patterns. In [3] and this work, a table is built alongside the tree to store the information from each sensor. Despite using the compact tree and table structures, the process is still time-consuming. This study attempts to reduce the runtime and consider the incremental essence of data streaming through three approaches: 1) the frequency reduction of the database scans to one, 2) implementation of incremental mining, 3) parallelization of process using Apache Storm [4]. For the aim of reducing database scan times and incremental mining, a compact tree structure named SDRF-tree (Sensor Data Regularity and Frequency-tree) and SSDRF-growth (Storm Sensor Data Regularity and Frequency-growth) algorithm are introduced. The outcomes are compared with the results in Rashid's work, which used FP-growth [5] algorithm and MapReduce [6] in order to mine regular- frequent patterns [7]. The rest of the paper is organized in follows. Section 2 introduces the problem of finding regular-frequent patterns on incremental sensor networks. Section 3 discusses related works. Section 4 introduces big data stream mining and our proposed SDRF-tree and SSDRF-growth structure. Section 5 discusses evaluation results, and Section 6 concludes.

Table 1. Sensor Database (SD) sample.

Epoch		Epoch		Epoch	
t_n	Occurrence	t_n	Occurrence	t_n	Occurrence
1	S1, S2, S6	**4**	S2, S3, S4, S5	**7**	S1, S2, S6
2	S3, S4, S6	**5**	S4, S5, S6	**8**	S3, S4, S5
3	S1, S2, S4, S5, S6	**6**	S1, S2, S6	**9**	S1, S3, S4

2. Problem Formulation

Same as the problem formulation in [3], the basic definitions for frequent-regular pattern mining are given, with only new concepts mentioned and similar definitions exemplified using table 1 data.

Based on Table 1, it comprises 6 sensors and 9 epochs. In this example, epoch 1 is $t_1 = (1, \{s_1, s_2, s_6\})$, and the pattern $Z = \{s_1, s_2\}$ has occurred at timeslots 1, 3, 6, and 7. Therefore, $T^z = \{t_1^z, t_3^z, t_6^z, t_7^z\}$.

Definition 1 (a period of pattern X). For simplicity, the first epoch, t_{first}, is assumed to be zero, with no sensor data existence, and the last epoch is n, based on Table 1, $t_{last} = 9$. Thus, according to the period definition in [3] and Table 1, the pattern S1,S2,S6 has occurred in t1, t3, t6, and t7. Thus, the period of the this pattern, with $t_{first}=0$ and $t_{last}=9$, is, $(1 - t_{first}), (3 - 1), (6 – 3), (7 – 6), (t_{last} - 7) =(1, 2, 3, 1, 2)$.

Definition 2 (regularity of pattern *X*). Based on regularity definition in [3], the regularity of the pattern *Z* is 3 as $Max(1, 2, 3, 1, 2) = 3$. Thus, if the user defines 2 non-occurrence timeslots as the threshold, *Z* is not identified as a regular pattern.

Definition 3 (frequency of pattern X). If a pattern occurrences number reaches a specific number defined by the user that is called Minimum Support, it is considered a frequent pattern.

Definition 4 (regularity and frequency of pattern X). A pattern is regular-frequent if it satisfies both definitions 2 and 3.

Definition 5 (stream period). It refers to the number of windows of stream data that is user-specified.

3. Related Works

Apriori-based and FP-growth-based are two types of regular-frequent patterns mining methods. Apriori-based methods generate a large number of candidate patterns or increase the frequency of database scans [8]. FP-growth-based methods use a compact tree structure to mine patterns with fewer database scans [3]. Tanbeer *et al.* proposed the RP-tree method, an FP-growth-based approach, to extract patterns from static databases with two scans [9]. Tanbeer in [10], introduced IncRT, a tree structure for mining regular patterns from incremental databases. Both [9] and [10] require a two-time scan of database which increases the runtime. Moreover, Tanbeer *et al.* developed the CP method for mining frequent patterns from incremental transactional databases using a prefix tree [11]. Also, Tanbeer *et al.* developed the SDR-growth method for mining regular patterns from incremental transactional databases using an SDR-tree with a single scan [3]. Goyal *et al.* developed the AnyF1 algorithm for mining frequent patterns from transactional databases using a BFI-Forest to handle stream data at varying speed ratios [12]. In 2021, Xun *et al.* created FPMSIM, an FP-growth-based algorithm for mining frequent patterns from incremental transactional databases [13]. None of these methods can mine regular-frequent patterns. Rashid *et al.* introduced the RF-Tree method, which mines these patterns with two database scans [14]. Rashid's subsequent work, RFSP-Tree, mines the regular-frequent patterns with a single database scan [7]. Neither of RF-Tree or RFSP-Tree, addressed incremental transactional databases. Also, RP-Tree [9], RF-Tree [14], and RFSP-Tree [7] approaches run on one processor that cause prolonged runtime especially in big data domain.

MapReduce, a distributed computing framework, has been successful in analyzing big data [6]. Lin *et al.* [15] proposed three Apriori-based algorithms called SPC (single-pass count), FPC (fixed-passes combined-counting), and DPC (dynamic-passes combined-counting) that use MapReduce to mine frequent patterns from transactional data. Riondato *et al.* [16] developed a parallel randomized algorithm called PARMA for mining approximations to the top-k frequent item-sets and association rules from transactional data using MapReduce. Aridhi *et al.* [17] developed a MapReduce-based approach for distributed frequent subgraph mining. M. Bhuiyan and M. Al Hasan proposed an iterative MapReduce-based frequent subgraph mining algorithm [18]. C.K.-S. Leung, Y. Hayduk proposed a MapReduced-growth (MR-growth) algorithm to mine accurate frequent itemsets from uncertain data using MapReduce framework [19]. In another work [20] a new technique, the multi-objective k-means algorithm, is used to aggregates medical data. This is complemented by a parallel pattern mining approach using GPU and MapReduce architectures for pattern creation. Also, the work [21] explores parallel big data processing techniques for finding frequent sequences in large datasets and proposes SPARSS, a scalable algorithm for distributed systems handling large sequential data. These works used MapReduce as a foundation for data mining from stored databases.

Rashid *et al.* introduced RFSP-H, a parallel method using MapReduce on sensor databases to mine regular-frequent patterns in a multiprocessor environment [7]. The method embodies a Mapping and two Reduce phases. The Mapping phase discovers a set of candidate patterns using the Balanced FP-growth method [5]. The first Reduce phase mines frequent patterns from these candidates, while the second mines regular patterns from the frequent ones, resulting in regular-frequent patterns. Balanced FP-growth [5] requires two database scans and lacks support for incremental databases, crucial for streaming data. Furthermore, in RFSP-H [7], middle results between Map and Reduce phases are inefficiently stored on external storage.

4. Proposed SSDRF-growth Structure and Mining Process

We proposed SSDRF-growth, an FP-growth-based method [2], to mine regular-frequent patterns. Firstly, we will describe the SDRF-tree structure that is used in mining process.

4.1. Proposed SDRF-tree Structure and Construction

Here, we describe the single-pass construction process of proposed SDRF-tree and its sensor data table based on Table 1. For simplicity, before inserting into a tree, epochs sort alphabetically rather than arbitrary orders. Figure 1 shows the tree and table state after the insertion of epoch 1.

Sensor data table				
D	LES	ESS	R	P
1	1	{1}	8	
2	1	{1}	8	
6	1	{1}	8	

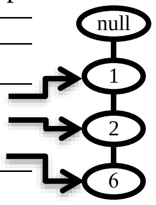


Figure 1. Insertion of first epoch, according to the Table 1.

The sensor data table contains five columns D, LES, ESS, R, and P, representing the sensor's data value, the last observed epoch for the sensor, all observed epochs for it, the regularity measure, and the pointer to the corresponding node in the tree, respectively. To examine the insertion process according to Table 1, the first element of the first epoch, S1, is inserted into the tree. As it is the first sensor, it is inserted as null's child and the data table for this node is set. Since no S1 has been seen previously, a row with a data value of 1 is created in the table. As no epoch has been seen before, epoch one is currently the last epoch in which S1 is seen, so LES becomes 1. For the ESS column, since this is the first epoch that S1 has been observed in, only {1} is placed in the set of epochs. The regularity value is now calculated using ESS. Since 1 is the only epoch, it is clear that the first occurrence of the sensors is epoch 1 and generally, the last occurrence of the sensors in a window is 9. So, for the simplicity of regularity calculation, by considering a null epoch at the beginning of the epochs, the largest non-occurrence interval indicates the sensor's regularity, which is equal to maximum of 1-0 = 1 and 9-1 = 8. Therefore, the regularity of the sensor is 8. Then, a pointer is created in the sensor's data value in the table, which points to its first location in the tree. The pointer makes it easier to access the node in the tree during pattern mining. The table values for S1 are completed. S2 and S6 will be inserted in the tree and table similar to S1. The second epoch enters after the first is fully inserted into the tree and data table. The tree and table will then be in the state shown in Figure 2.

Sensor data table				
D	LES	ESS	R	P
1	1	{1}	8	
2	1	{1}	8	
3	2	{2}	7	
4	2	{2}	7	
6	2	{1,2}	7	

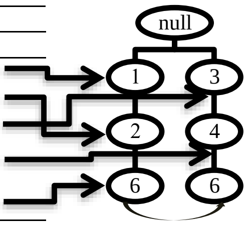


Figure 2. Insertion of 2nd epoch, according to the Table 1.

At the beginning of the second epoch insertion, it is checked whether the null root has any children of value 3 or not. If not, S3 is inserted into the tree same as for the first epoch's sensors. To insert S3 into the table, its occurrence in earlier epochs will be checked. Since it has not occurred and the table has no data for S3, in the table, the value of S3 and its associated columns will be inserted. S4 will be inserted in the same way and then S6 as well. However, since S6 has occurred earlier, it does not re-enter it in the table. Instead, its associated columns are updated. The LES and ESS columns for row 6 will be 2 and {1,2}, respectively. The regularity criterion equals to *Max*(1-0 = 1, 2-1 = 1 and 9-2 = 7), which is 7. As this sensor has occurred in earlier epochs, it takes a different position in the SDRF-tree, and a pointer connects the previous S6 observation to the current node. After inserting all epochs into the tree, the sensor data table and the SDRF-tree will look like Figure 3. (To simplify the node traversal pointers are not shown.)

Sensor data table				
D	LES	ESS	R	P
1	9	{1,3,6,7,9}	3	•
2	7	{1,3,4,6,7}	2	•
3	9	{2,4,8,9}	4	•
4	9	{2,3,4,5,8,9}	3	•
5	8	{3,4,5,8}	3	•
6	7	{1,2,3,5,6,7}	2	•

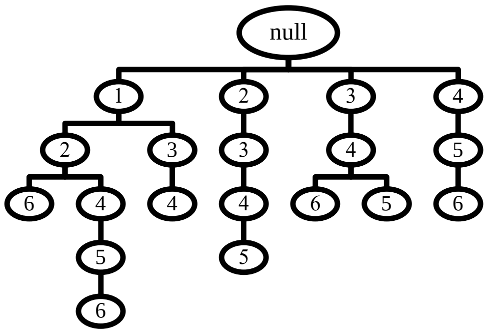


Figure 3. SDRF-tree structure and sensor data table after inserting all epochs according to Table 1.

The leaf nodes have a list called TidList, which stores the number of epochs in which that branch from root to leaf has occurred. In algorithm 1, the SDRF-tree pseudo-code is shown.

table is created simultaneously with the tree. For this purpose, initially, the emptiness of the table is checked. There are two possible scenarios. First, the table is empty when the first element of the first epoch is checked at the start of the process. Therefore, the current node and its associated values such as *LES* sensor's regularity, and *ESS* are inserted into the empty table. Secondly, if the table is not null, it is first checked to see if the current node and its value are in the table (same node, with the same position in the tree). If yes, the *LES*, *R*, and *ESS* will be updated. If no, the same node's pointers in the table are checked as well. If the current node is among the pointers, the sensor's values will be updated. If it is not among the pointers, then the current node will be added as a new pointer to the list of pointer nodes associated

```
Input: SD: The sensor database
Output: A compact tree and it's information table as SDRF-tree
1.      create the root, R of an SDRF-tree, T, and label it as "null" and also create empty SD_table;
2.      for each epoch $t_i$ in SD do
3.              if $t_i \neq$ NULL then
4.                  sort $t_i$ according to given order and call SDRF-tree ($t_i$, R);
5.function SDRF-tree ($t_i$, R)
6.      Let the sorted sensor list in $t_i$ to be [y|Y], where y is the first sensor and Y is the remaining list;
7.      If (R has a child C such that C.sensor = y.sensor)
8.          select C as the current node;
9.      Else
10.         create a new node C as child of R     ;
11.     If (SD_table is empty)
12.         insert current node in SD_table and set LES, Reg, ESS columns in SD_table for current node;
13.     Else
14.         if (current node equal to existing sensor in SD_table)
15.             update LES, Reg, ESS columns in SD_table for current node;
16.         Else
17.             compare current node with pointer node of existing sensor in SD_table;
18.         If (current node equal to one of pointer node)
19.             update LES, Reg, ESS columns in SD_table for current node;
20.         Else
21.            add current node to pointer node list of existing sensors in SD_table;
22.         update LES, Reg, ESS columns in SD_table for current node;
23.     If (SD_table ≠ empty and current node not inserted to pointer node list of existing nodes in SD_table)
24.          Insert current node in SD_table and set LES, Reg, ESS columns in SD_table for current node;
25.     If (end of $t_i$ reached)
26.          add the tid of $t_i$ in current node's tid-list;
27.     Else
28.          call SDRF-tree (Y, current node);
```

Algorithm 1. SDRF-tree construction algorithm.

According to Algorithm 1, the function receives epoch and root as inputs. The first value of the input epoch and the remaining elements of that epoch are considered as *y* and *Y*, respectively. Then, the root of the input, *R*, is examined to see whether it has a child similar to *y* or not. If so, *y* is set as the child of *R*, *C*, which is the current node; otherwise, a new node, *C*, is created using *y* as the child of *R*. Here, unlike many other methods that build the table after the creation of the tree such as [3] the

with the equivalent node in the table. (Curve s6 pointer in Figure 2); otherwise, if the table is not empty and no similar nodes are found, it will be added to the table as a new node. The values will be inserted for the current node, similar to other sensors in the *SD_table*. After completing the insertion operation in the *SD_table*, it is checked whether the current node is the end node of the epoch or not. If yes, the current epoch number will be inserted into *tid-list*, which is the list of epoch

numbers that the sensors in the current epoch (or tree branch) have occurred in them. If it is not the epoch's last sensor, the same process will be recursively done for the next value in *Y*. When the number of epochs inserted in the tree reaches the window size, the constructed *SD_table*, along with the current window number and window size, is sent to the next step to mine the pattern from *SD_table*.
As the proposed method is based on Apache Storm [4], firstly, its basics will be discussed.

4.2. Stream processing with Apache Storm

Apache Storm has 3 main parts: spout, bolt, and topology.

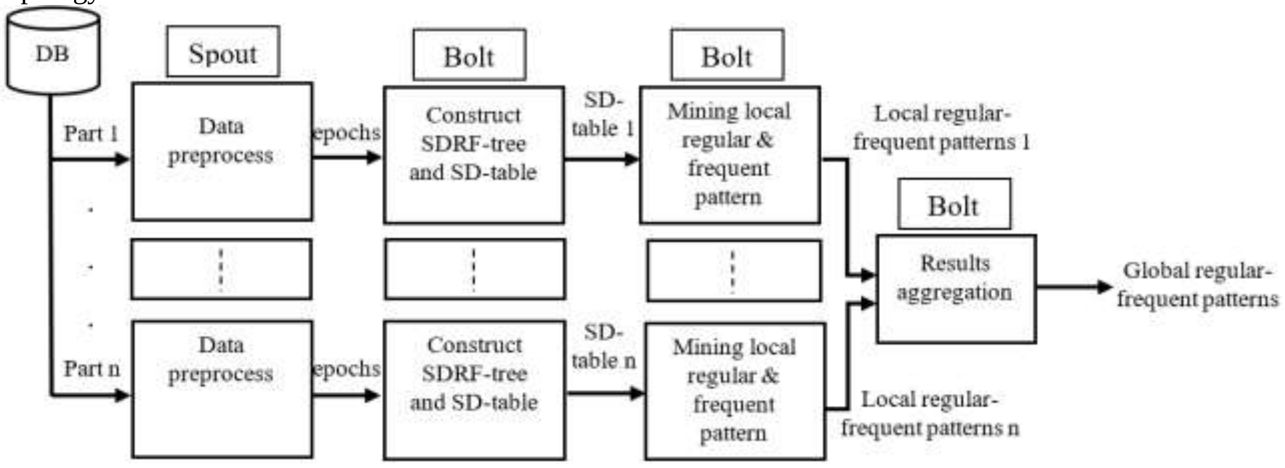

DB
Spout
Bolt
Bolt
Bolt
Part 1
Part n
Data preprocess
epochs
Construct SDRF-tree and SD-table
SD-table 1
SD-table n
Mining local regular & frequent pattern
Local regular-frequent patterns 1
Local regular-frequent patterns n
Results aggregation
Global regular-frequent patterns

Figure 4. Block diagram of the proposed method.

The spout receives and pre-processes input data. The bolt processes input data and generates output. The relationships between spouts and bolts are determined in topology. In fact, there is a directed graph composed of spouts and bolts. As the connection between spout and bolt is provided by RAM, there is no need for any external memory for their middle results. This has a significant effect on reducing the runtime of the operation [4].
The connection between the spout and the bolt, or the two bolts is created via the tuple, which is an ordered list of data elements. In the Storm framework, it is possible to select and implement the type of stream data between the nodes, which is called stream grouping [4]. In this work, direct grouping method is used, in which the tuple producer determines the receiver of the tuples.

4.3 Architecture of Proposed Method

The proposed method architecture consists of four general sections (Figure 4). The first part receives the input data and passes it in the form of epochs to the next part. In the second part, SDRF-tree and SD_table are constructed based on the received window of epochs. In the third part, SD_table will be mined by constructing the first conditional pattern-base (PB) tree and table for each sensor in the SD_table. According to the PB-table, the conditional tree (CT) in form of table structure will be constructed. Then in the pattern mining section, the regular-frequent patterns will be mined from the local conditional table. The locality of the pattern means each part of the dataset is processed separately in parallel, and the local results are obtained from each one. Finally, the local results are transmitted to the final section.
for aggregation to extract global regular-frequent patterns. In the following, the details of these components will be described.

4.3.1 Receiving and Managing Input Data in Parallel

In this section, stream data reception is performed through the spout of Apache storm [4]. The input dataset file is read line by line and each line represents an epoch of sensors occurrences. By performing pre-processing, the input string converts to a list of numbers. This list is then sorted in accordance with the proposed order, and eventually is sent to the next section for the SDRF-tree construction. It is noteworthy that in the spout, the data is manually divided by defining the window size as a parameter, then each section of the data is sent by the spout to the destination bolt. The destination bolt is also selected with the help of direct grouping method [4], so that the data is distributed purposefully. This method is necessary when a node with specific hardware or software is required to process the specific data. Here, because of the uniformity of the data, as well as the equalization of the hardware and software capabilities of the bolts, each spout sends its output to one of the bolts without considering any orders. In addition, in order to simulate the stream of data, each epoch is part of a window and when the number of epochs reaches the size of the window, the number of epochs is reset to zero.

4.3.2 Parallel Construction and Algorithm of SDRF-tree and SD-table

The construction of an SDRF-tree is performed in parallel through several node instances, regarding the division of the data. Each instance is responsible for creating the SDRF-tree and the SD-table for a portion of the data. For this, the SDRF-tree function is called by each receiving epoch, until the window size limit is reached, then it is restarted. The process of constructing the tree and the table is mentioned in Section 4.1.

for storing the generated patterns. After mining the patterns, when the number of windows reaches the stream period threshold, the *RFP* and window size are sent to the next section.

According to the Figure 3, the SSDRF-growth algorithm starts with the last sensor, S6, as the leaf node in the SD_table and the last sensor is considered as α, temporarily. Also, last node of the branch holds the *TidList*, which stores all the epochs that branch occurred in. To construct the conditional pattern-base tree, all paths leading to α are checked to see if the sensors in that path meet the regularity threshold. Sensors that meet this threshold are copied to the conditional pattern-base

```
Input: SD_table, current window number as win_ num, RFP, window Size as win_size
Output: local RFP
1. Function SSDRF-growth (SD_table, win_ num, RFP, win_size)
2.    while SD_table not empty do
3.        call BuildPB (sensor $\alpha$);
4.        call Mine (PB_table($\alpha$), sensor $\alpha$) and then remove sensor $\alpha$;
5. Function BuildPB (sensor $\alpha$)
6.    for each node of $\alpha$ do
7.        project path that ends to node from its parent to root in new tree & table called PB_tree($\alpha$) and PB_table($\alpha$);
8.    return PB_table($\alpha$);
9. Function Mine (PB_table($\alpha$), sensor $\alpha$)
10.    call BuildCT(PB_table($\alpha$));
11.    if CT_table($\alpha$) $\neq$ null then
12.        for each sensor $\beta$ in CT_table($\alpha$) from bottom do
13.            generate pattern $\beta = \beta \cup \alpha$;
14.            if ($\beta$ although is regular)
15.                if ($\beta$ exist in RFP)
16.                    update $\beta$'s last occurrence and $\beta$'s number of occurrences in RFP;
17.                    if (number occurrence of $\beta$ >= Minimum Support)
18.                        mark $\beta$ as frequent Pattern;
19.                else
20.                    add $\beta$ in RFP with its last occurrence and number of occurrences;
21.                    if (number occurrence of $\beta$ >= Minimum Support)
22.                        mark $\beta$ as frequent Pattern;
23.            returnedPB = BuildPB($\beta$);
24.            call Mine (returnedPB, $\beta$) and remove sensor from bottom of CT_table($\alpha$);
25. Function BuildCT (PB_table($\alpha$))
26.    for each sensor $\beta$ in PB_table($\alpha$) do
27.        if regularity of $\beta > \lambda$ then
28.            for each node N$\beta$ in PB_table($\alpha$) do
29.                project the path ended with N$\beta$ from its parent up to root in a new tree & table called CT_tree($\alpha$) and CT_table($\alpha$)
30.    return CT_table($\alpha$)
```

Algorithm 2. SSDRF-growth Algorithm.

4.3.3 Incremental Mining of Regular-Frequent Patterns in Parallel

Similar to the construction of the SDRF-tree, in case of having multiple mining instances, each one mines the regular-frequent patterns of a local *SD-table*. In the following section, the process of mining regular-frequent patterns in both local and global forms will be discussed.

After receiving the constructed tree and table, in the mining bolt, the *SSDRF-growth* function is called with the *SD-table*, the current window number, and the *RFP* as inputs. *RFP* is a variable

tree of α (*PB_tree(α)*). The table with this tree is called *PB_table(α)*. Figure 5 shows the S6's *PB_tree* and *PB_table*, which are constructed using a regularity measure equal to 3. Note that non-regular nodes such as S3, and self-mining node, S6, are not copied to the desired branch.

PB_table(S6)

D	LES	ESS	R	**P**
1	7	{1,3,6,7}	3	•
2	7	{1,3,6,7}	3	•
4	5	{2,3,5}	4	•
5	5	{3,5}	4	•

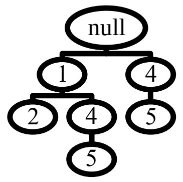


Figure 5. PB_table(S6) and PB_tree(S6) of S6.

After constructing the *PB_tree(α)* and *PB_table(α)*, they will be mined. As some paths were not transferred to the mentioned tree and table, sensor values, such as their regularity in the *PB_table(α)* have been changed. To re-check the regularity, the conditional tree is constructed by removing irregular sensors from the *PB_table(α)* to generate another table called *CT_table(α)*. The *CT_table(α)* can be seen in Figure 6.

CT_table$_{(S6)}$				
D	LES	ESS	R	**P**
1	7	{1,3,6,7}	3	•
2	7	{1,3,6,7}	3	•

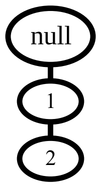


Figure 6. CT_table and CT_tree of S6 (CT_tree(α) is not constructed but is shown just for better understanding.)

For mining, *CT_table(α)* is checked to determine whether it is null or not. If not, sensors are chosen from the bottom of the *CT_table(α)* and become a union with *α* as a two-element pattern known as *β*. *β* is considered as a local regular pattern, and then is stored with its last occurrence and the number of its occurrences. If the number of pattern occurrences is equal to or greater than the Minimum Support, this pattern also is known as a local frequent pattern. For mining three- or more element patterns, if existed, the algorithm tries to treat *β* as a new sensor and build *PB_table(β), PB_tree(β), and CT_table(β)* recursively. For example, if the Minimum Support measure is also considered as 3, according to the CT_table(S6) and based on S6 occurrences, 3 patterns {S1, S6}, {S2, S6} and {S1, S2, S6} are regular-frequent pattern. All of the patterns in *RFP*, obtained from the evaluation of all sensors in SD-table, belong to the current window. When these operations are done on the subsequent windows, their last occurrence and the number of occurrences is updated if the generated patterns from those windows are in the *RFP*; otherwise, the pattern will be inserted in the *RFP* with its last occurrence and the number of occurrences. In contrast to the RFSP-H method [7], which uses a static mining approach to mine regular-frequent patterns only in one window, the proposed method considers incremental mining. Thus, the regularity and frequency of the pattern can be tracked in the sequence of windows. To calculate the regularity of a pattern in sequence of windows, the difference between pattern's occurrence in the previous window (if exists) and the current window is firstly calculated. If the pattern's regularity measure does not exceed *λ*, the pattern is still regular locally. The pattern is locally frequent if its occurrence's number reaches the local Minimum Support threshold. If it wasn't frequent in the current window, the pattern is kept until the next windows are received. Afterward, if the pattern is seen and its total occurrences in both current and previous windows reached the local Minimum Support, it is marked as a local frequent pattern; otherwise, this process will continue until the end of the current stream period. After mining the patterns of a window, constructed *SDRF-tree* and *SD_table* will be eliminated. But their generated result in the *RFP* will still be available. Because of this, the regular-frequent patterns can be detected across multiple windows in a stream period. After the stream period is completed, the *RFP* is sent to the results aggregation bolt.

4.3.4 Mining global regular-frequent patterns

The results aggregation node consists of one bolt instance in which all the patterns that were generated in different bolts will be integrated. In some conditions, there may be sensor networks in different areas that are processed locally or it becomes necessary to aggregate the data of different areas. For aggregation, all of the patterns in the received *RPF* are checked with the *final_RFP* (stored data of previous window RFPs). In the case of pattern existence in the *final_RFP*, its regularity is calculated by subtraction of the sensor's first occurrence in the current *RFP* from its last occurrence in the *Final_RFP*. If its regularity is lower than *λ*, the pattern is still regular.

```
Input: bolts RFPs, window size as win_size
Output: final_RFP
1. Function execute()
2.   receive RFP and win_size from pervious bolt
3.   for each item in received RFP
4.     compare with items in final_RFP
5.     if item exists in final_RFP and item is frequent in bolt RFP
6.         calculate global frequency of item
7.         if item global frequency > global frequency threshold as glob_freq_threshold
8.           let p_last(item) be the timeslot id of the last epoch containing item in pervious RFP
9.           let p_first(item) be the timeslot id of the first epoch containing item in current RFP
10.          if (p_first(item) - p_last(item)) < λ
11.            item still is regular and update item in final_RFP with new frequency and occurrence information
```

Algorithm 3. Mining global regular-frequent patterns.

Table 2. Characters of tested datasets.

Dataset type	Gap between epochs	Days number	Epochs number	Sensors number	Datasets
Real	31 sec	5-days	14400	54	**Intel data (5-day)** [22]
Real	31 sec	10-days	28800	54	**Intel data (10-day)** [22]
Artificial	-	-	100000	870	**T10I4D100K** [23]
Artificial	-	-	3196	75	**Chess** [23]
Artificial	-	-	990002	41270	**Kosarak** [23]

If it is regular, then its frequency is set by summing its occurrences' number in the current *RFP* and the *final_RFP*. Also, the new occurrence information of pattern will be updated. Therefore, its frequency and regularity can be tracked during the next *RFP*s. The mentioned process is defined in Algorithm 3.

5. Experimental Result

This section presents the datasets [22] [23], the test environment, and evaluation criteria. Also, the performance of the proposed SDRF-tree and mining method using SSDRF-growth is discussed. A seven-node Linux cluster is used for evaluation, with each node having two 2.00 GHz processor cores and 2 GB of memory. One storm node is Master for Nimbus Ghost and the other six are Workers for Supervisor Ghost implementation.

5.1. Evaluation metrics

This study considers the runtime of mining regular-frequent patterns from WSN big data as an evaluation criterion. To reduce it by considering the incremental mining process, the SDRF-tree and SSDRF-growth algorithms are used on the Apache Storm platform [4] parallelly. The runtime of SDRF-tree has been compared with one of the successful tree structures called SDR-tree [3] and also the whole mining runtime has been compared to the results of the Rashid *et al.* study [7], which used the Hadoop platform [24] with MapReduce [6] and the two-pass mining Algorithm [5].

5.1.1 SDRF-tree runtime

The SDRF-tree algorithm's runtime was evaluated on different datasets *[23]*, the results of which can be seen in Figure 7.

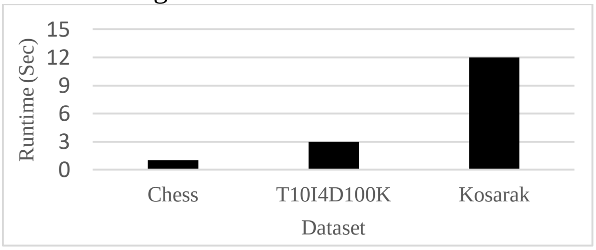


Figure 7. SDRF-tree runtime on datasets in [23]

In Figure 7, the runtime of the SDRF-tree based on the size of the dataset, shows the proposed method performs very well in dealing with large volume data and increasing the size of dataset does not impair that. The Kosarak dataset is 9.9 times bigger than the T10I4D100K dataset but the runtime just increased 10 seconds that is four times bigger than T10I4D100K runtime. This indicates the SDRF-tree optimum runtime given the big volume of data. Figure 8 shows the single node runtime of the SDRF-tree compared with the SDR-tree [3] with two nodes on the T10I4D100K dataset with 100000 records that indicates the better performance of SDRF-tree even with the fewer number of processors.

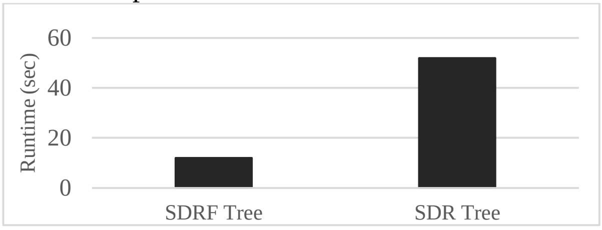
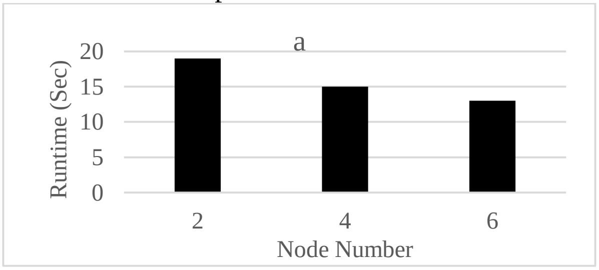


Figure 8. Comparison of SDRF-tree runtime with SDR-tree on T10I4D100K dataset.

5.1.2 Proposed method runtime

The proposed method's runtime (the total time of reading data and processing it), is evaluated by changing the number of cluster nodes. The applied conditions were: the local Minimum Support of 0.7%, the global Minimum Support was set based on the number of data divisions, the local and global λ were equal to 20%, and the number of cluster's Worker nodes was considered as 2, 4, and 6 and in all experiments, the Master node was considered as a separate node.



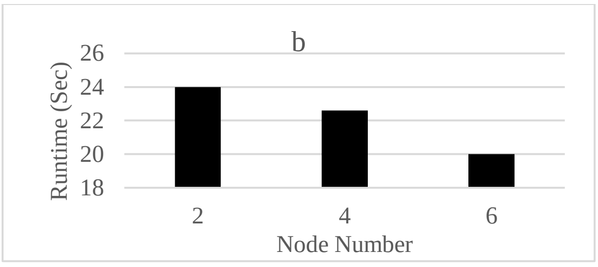


Figure 9. Runtime of the proposed method in a) intel 5-day data, b) intel 10-day data [22].

As it is shown in figures 9-a and 9-b, the runtime on these data sets is very low and it declines with increasing the number of nodes in the cluster for the two introduced datasets.

5.1.3 Effect of Regularity Measure on Runtime

Increasing the regularity measure leads to increasing the patterns because the patterns with less occurrence can also be considered as regular patterns. The effect of this measure is shown in Figures 10-a and 10-b.

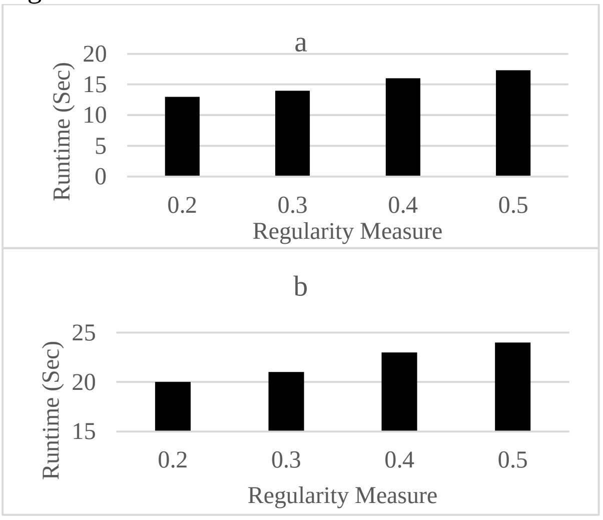
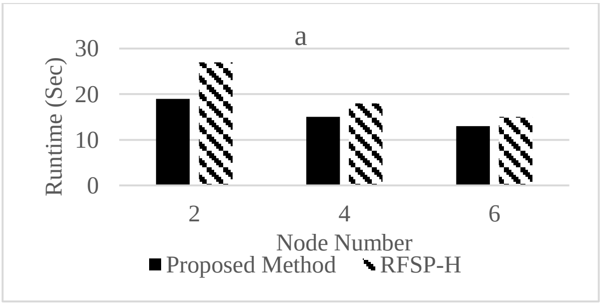


Figure 10. Regularity measure effect on the proposed method runtime in a) intel 5-day, b) intel 10-day data [22].

5.1.4 Comparison of the proposed method with RFSP-H method

RFSP-H method [7] mined the regular-frequent patterns of WSN data using Hadoop platform [24] and the two-pass algorithm called Balanced FP-growth [5]. The figures 11-a and 11-b shows the performance comparison of RFSP-H and proposed method.



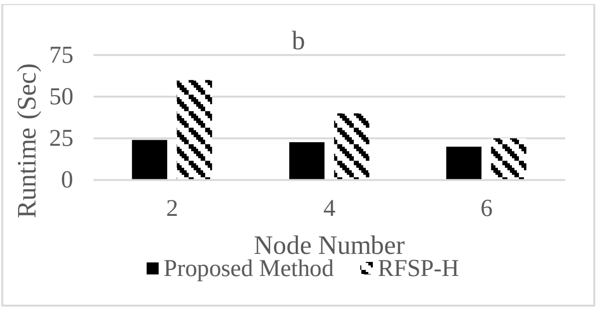


Figure 11. Proposed method's and RFSP-H's [7] runtime comparison in a) intel 5-day data, b) intel 10-day data [22]

As shown in figure 11, comparing with the RFSP-H [7], our method reduced runtime by 14%, 18%, and 34% on the Intel 5-day dataset and by 22%, 55%, and 85% on the Intel 10-day dataset in 6, 4, and 2 node clusters respectively.

5.1.5 Effect of Dataset Size

Experiments show that our method outperforms the RFSP-H [7] more significantly with larger datasets. On average, our method's runtime on the 5-day dataset is 24% (4.3 seconds) lower across all 2, 4, and 6 node clusters. This difference increases dramatically to 61% (19.5 seconds) on the 10-day dataset.

6 Conclusion

This study aimed to reduce the runtime of mining regular-frequent patterns while considering data flow. This was achieved through three measures: 1) reducing database scans to one, 2) implementing incremental pattern mining, and 3) parallelizing the mining process using SDRF-tree and SSDRF-Growth parallel algorithms on the Apache Storm platform [4]. This study's results were obtained using 6, 4, and 2 Linux clusters and two Intel lab datasets (5-day and 10-day) [22]. With respect to the order of the mentioned clusters, the proposed method was compared to the RFSP-H method [7] and exhibited an improved runtime of 14, 18 and 34 percent in the 5-day dataset, and 22, 55 and 85 percent in the 10-day dataset. It was observed that the proposed method has superior performance than the RFSP-H method [7], especially in a larger dataset. In future research, we will try to detect the noise of the data and neutralize their destructive effects, before the pattern mining process. This will be carried out with the help of non-deterministic methods such as machine learning algorithms which cause better accuracy in addition to the fast-mining runtime.

References

[1] W. Gan, J. C.-W. Lin, P. Fournier-Vige, H.-C. Chao, and P. S. Yu, "A survey of parallel sequential pattern

mining," *ACM Transactions on Knowledge Discovery from Data (TKDD),* Vol. 13, No. 3, pp. 1-34, 2019.

[2] J. Han, M. Kamber, and J. Pei, Data mining: concepts and techniques, 3 ed., Morgan Kaufmann, 2012.

[3] S. K. Tanbeer, M. M. Hassan, A. Almogren, M. Zuair, and B.-S. Jeong, "Scalable regular pattern mining in evolving body sensor data," *Future Generation Computer Systems,* vol. 75, pp. 172-186, 2017.

[4] "Apache Storm," [Online]. Available: http://storm.apache.org/. [Accessed 26 1 2023].

[5] K.-M. Yu, J. Zhou, and W. C. Hsiao, "Load balancing approach parallel algorithm for frequent pattern mining," in *International Conference on Parallel Computing Technologies*, Berlin, Heidelberg, 2007.

[6] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM,* vol. 55, no. 1, p. 2008, 107-113.

[7] M. Rashid, I. Gondal,, and J. Kamruzzaman, "Dependable large scale behavioral patterns mining from sensor data using Hadoop platform," *Information Sciences,* vol. 379, pp. 128-145, 2017.

[8] V. M. Nofong, "Discovering productive periodic frequent patterns in transactional databases," *Annals of Data Science,* pp. 235-249, 2016.

[9] S. K. Tanbeer, C. Farhan Ahmed, B.-S. Jeong, and Y.-K. Lee, "Rp-tree: A tree structure to discover regular patterns in transactional database," in *Conference on Intelligent Data Engineering and Automated Learning*, Berlin, Heidelberg, 2008.

[10] S. K. Tanbeer, C. Farhan Ahmed, and B.-S. Jeong, "Mining regular patterns in incremental transactional databases," in *Web Conference (APWEB), 2010 12th International Asia-Pacific,* Busan, Korea (South), 2010.

[11] S. K. Tanbeer, C. Farhan Ahmed, B.-S. Jeong and Y.-K. Lee, "Efficient single-pass frequent pattern mining using a prefix-tree," *Information Sciences,* vol. 179, no. 5, pp. 559-583, 2009.

[12] P. Goyal, J. S. Challa, S. Shrivastava, and N. Goyal, "Anytime Frequent Itemset Mining of Transactional Data Streams," *Big Data Research,* vol. 21, 2020.

[13] Y. Xun, X. Cui, J. Zhang, and Q. Yin, "Incremental frequent itemsets mining based on frequent pattern tree and multi-scale," *Expert Systems With Applications,* vol. 163, 2021.

[14] M. Rashid, R. Karim, B.-S. Jeong, and H.-J. Choi, "Efficient mining regularly frequent patterns in transactional databases," in *International Conference on Database Systems for Advanced Applications,* 2012.

[15] 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,* 2012.

[16] M. Riondato, J. A. DeBrabant, R. L. C. Fonseca, and E. Upfal, "PARMA: a parallel randomized algorithm for approximate association rules mining in MapReduce," in *Proceedings of the 21st ACM international conference on Information and knowledge management,* 2012.

[17] S. Aridhi, L. d'Orazio, M. Maddouri, and E. Mephu, "A novel MapReduce-based approach for distributed frequent subgraph mining," in *Reconnaissance de Formes et Intelligence Artificielle (RFIA)* , 2014.

[18] M. A. Bhuiyan and M. Al Hasan, "An iterative MapReduce based frequent subgraph mining algorithm," *IEEE Transactions on Knowledge and Data Engineering,* vol. 27, no. 3, p. 2014, 608-620.

[19] C. K.-S. Leung and Y. Hayduk , "Mining frequent patterns from uncertain data with MapReduce for big data analytics," in *International Conference on Database Systems for Advanced Applications*, 2013.

[20] Y. Djenouri, A. Belhadi, G. Srivastava, and J. Chun-Wei Lin, "A Secure Parallel Pattern Mining System for Medical Internet of Things," *IEEE/ACM Transactions on Computational Biology and Bioinformatics,* pp. 1-12, 2023.

[21] A. B. Can, M. Zaval, M. Uzun-Per, and M. Aktas S., "On the big data processing algorithms for finding frequent sequences," *Concurrency and Computation: Practice and Experience,* 2023.

[22] "Intel Lab Data," [Online]. Available: http://db.csail.mit.edu/labdata/labdata.html. [Accessed 26 1 2023].

[23] "Frequent Itemset Mining Dataset Repository.," [Online]. Available: http://fimi.ua.ac.be/data/. [Accessed 26 1 2023].

[24] "Apache Hadoop," [Online]. Available: https://hadoop.apache.org/. [Accessed 26 1 2023].

کاوش افزایشی الگوهای باقاعده-مکرر از کلان داده‌های سنسوری به صورت موازی

صادق رحمانی بلداجی[۱،*]، مهدی باطنی[۲] و محمود مرتضوی دهکردی[۳]

[۱] گروه مهندسی کامپیوتر، دانشگاه شیخ بهایی، بهارستان، اصفهان، ایران .

[۲] گروه علوم کامپیوتر، دانشکده ریاضی و کامپیوتر خوانسار، دانشگاه اصفهان، اصفهان، ایران .

[۳] گروه MSE، دانشگاه کانادا وِست (UCW)، ونکوور، بریتیش کلمبیا، کانادا .

ارسال ۲۰۲۳/۰۵/۰۵؛ بازنگری ۲۰۲۳/۰۸/۰۸؛ پذیرش ۲۰۲۳/۱۲/۱۵

چکیده:

استخراج الگوی باقاعده-مکرر از داده‌های تولید شده توسط حسگرها به صورت کارآمد به یک چالش تبدیل شده است. حجم زیاد داده‌ها منجر به طولانی شدن زمان اجرا می شود، بنابراین پیش‌بینی‌ها و تصمیم‌گیری‌های حیاتی را که نیاز به پاسخ فوری دارند به تاخیر می‌اندازد. بنابراین استفاده از پلتفرم‌های کلان داده و الگوریتم‌های موازی راه‌حل مناسبی است. علاوه بر این، تکنیک‌های افزایشی برای استخراج الگوها از جریان‌های کلان داده مناسب تر از روش‌های ایستا است. این مطالعه یک رویکرد موازی افزایشی و ساختار درختی فشرده را برای استخراج الگوهای باقاعده-مکرر از داده‌های شبکه‌های حسگر بی‌سیم ارائه می‌کند. همچنین، اسکن پایگاه داده کمتری به منظور کاوش الگوها برای کاهش زمان اجرا انجام شده است. این مطالعه بر روی مجموعه داده‌های ۵ و ۱۰ روزه اینتل با خوشه‌های ۶، ۴ و ۲ گره انجام شد. یافته‌ها نشان می‌دهد که زمان اجرا در هر ۳ حالت خوشه‌ای به ترتیب ۱۴، ۱۸ و ۳۴ درصد برای مجموعه داده ۵ روزه و ۲۲، ۵۵ و ۸۵ درصد برای مجموعه داده ۱۰ روزه بهبود یافته است.

کلمات کلیدی: الگوی باقاعده-مکرر، جریان کلان داده، الگوریتم موازی، کاوش افزایشی.