




An Efficient Spark-Based Hybrid Frequent Itemset Mining Algorithm for Big Data

Mohamed Reda Al-Bana ^{1,*}, Marwa Salah Farhan ^{1,2,*} and Nermin Abdelhakim Othman ^{1,2}

¹ Department of Information Systems, Faculty of Computers and Artificial Intelligence, Helwan University, Cairo 11795, Egypt; drnermin@fci.helwan.edu.eg or Nermin.Othman@bue.edu.eg

² Faculty of Informatics and Computer Science, British University in Egypt, Cairo 11837, Egypt

* Correspondence: Mohammed.Bana@fci.helwan.edu.eg (M.R.A.-B.); Marwa.salah@fci.helwan.edu.eg (M.S.F.)

Abstract: Frequent itemset mining (FIM) is a common approach for discovering hidden frequent patterns from transactional databases used in prediction, association rules, classification, etc. Apriori is an FIM elementary algorithm with iterative nature used to find the frequent itemsets. Apriori is used to scan the dataset multiple times to generate big frequent itemsets with different cardinalities. Apriori performance descends when data gets bigger due to the multiple dataset scan to extract the frequent itemsets. Eclat is a scalable version of the Apriori algorithm that utilizes a vertical layout. The vertical layout has many advantages; it helps to solve the problem of multiple datasets scanning and has information that helps to find each itemset support. In a vertical layout, itemset support can be achieved by intersecting transaction ids (tidset/tids) and pruning irrelevant itemsets. However, when tids become too big for memory, it affects algorithms efficiency. In this paper, we introduce SHFIM (spark-based hybrid frequent itemset mining), which is a three-phase algorithm that utilizes both horizontal and vertical layout diffset instead of tidset to keep track of the differences between transaction ids rather than the intersections. Moreover, some improvements are developed to decrease the number of candidate itemsets. SHFIM is implemented and tested over the Spark framework, which utilizes the RDD (resilient distributed datasets) concept and in-memory processing that tackles MapReduce framework problem. We compared the SHFIM performance with Spark-based Eclat and dEclat algorithms for the four benchmark datasets. Experimental results proved that SHFIM outperforms Eclat and dEclat Spark-based algorithms in both dense and sparse datasets in terms of execution time.

Keywords: big data; frequent pattern mining; horizontal layout; vertical layout; diffset; Spark



Citation: Al-Bana, M.R.; Farhan, M.S.; Othman, N.A. An Efficient Spark-Based Hybrid Frequent Itemset Mining Algorithm for Big Data. *Data* **2022**, *7*, 11. <https://doi.org/10.3390/data7010011>

Academic Editor: Giuseppe Ciaburro

Received: 28 November 2021

Accepted: 7 January 2022

Published: 14 January 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

We are currently living in the big data age. Data appear everywhere in a variety of formats and types ranging from structured to unstructured data and are produced by a huge number of sources across a wide range of disciplines and types, including transactional systems, user interactions, social networks, the Internet of Things, the World Wide Web, and many others. Companies and individuals gather and store all these generated data to analyze them for insight, knowledge, and decision making. Therefore, we have been swamped with big data, not just because we already have large amounts of data that need to be processed but also because the amount of data is rapidly growing every moment. The concept of big data has some properties that are collectively known as the “3Vs” model. Volume is defined as the amount of data; enormous amounts of data are generated and gathered. Velocity refers to the high rate at which data are created, gathered, and processed (streams, batch, near-real-time, and real-time). Variety indicates the different types of data: audio, images/video, and text; conventional structured data; and mixed data. In addition, there are two more features added to the “3Vs” model and known as the “5Vs” model. Veracity refers to how much the data is accurate and trusted when it comes from various

sources with different reliability. Value refers to how important the data are and the benefit that data should bring [1–3].

Today, distributed storage systems, which are designed to carefully manage access and administration in a fault-tolerant way, are frequently used to tackle the problem of big data gathering. Parallelization of algorithms is a solution for processing big data. This is usually done in one of two methods: data parallelism, in which the data are divided into partitions and each partition is calculated at the same time, or task parallelism, in which the algorithm is divided into stages that may be executed concurrently [4]. The rising availability of massive volumes of data has increased the importance of big data analytic systems and the interest in data mining, a critical collection of techniques for extracting useful knowledge from data. Big data analytics problems present significant challenges for researchers. Indeed, applying standard data mining methods and techniques to large amounts of data is difficult, and several of the most current data mining techniques had to be redeveloped to cope with the new environment. The key research field on which big data analytics depend is data mining with machine learning. It covers (i) clustering methods for detecting hidden structures in unlabeled data [5], (ii) FIM (frequent itemset mining) methods for detecting correlations between data [6], and (iii) supervised learning methods to predict what will happen in the future [2].

Frequent itemsets mining (FIM) is a basic data mining model that refers to attempting to extract or mine knowledge from large amounts of data [1]. Many other problems, such as association rules, correlations, and classifications, can use FIM as a primary calculation phase. In general, FIM counts the frequencies of co-occurring items, known as itemsets, in records of unique items from a transaction-oriented dataset. Moreover, FIM identifies all frequent itemsets with frequencies greater than or equal to a given support threshold. In the real world, FIM has been used in various real-world applications where many physical objects or activities appear together in scenarios and events. For example, FIM is used in health care for analyzing and forecasting the risk level of heart disease based on selected symptoms to assist medical professionals in determining the disease's risk at an early stage [7]. Market basket analysis utilizes FIM to find linkages between entities and things that frequently occur together, such as the products in a shopper's cart [8]. Additionally, text mining uses FIM to detect common bigrams, trigrams, or phrases in texts, as well as word co-occurrences in terms of adjacency [9]. More on this, FIM has many other usages in other domains including network traffic data [10], biological data [11], energy data [12], and images [13]. As a result, FIM is important; it plays a vital role in data mining and has a wide variety of applications. There are many FIM algorithms used to discover frequent patterns such as Apriori, Eclat, dEclat, and more [9].

Apriori [14,15] is an elementary iterative algorithm used to mine frequent itemset in datasets. It follows the horizontal layout (a data format in which each transaction contains a list of items) datasets and the repetitive dataset scan strategy to discover the frequent pattern. Each dataset scan (iteration) is responsible for two main processes. The first process is the candidate generation in which Apriori generates a set of candidate itemsets that are recommended to be frequent, while the second process computes the support (frequency) of an itemset, then prunes itemsets whose support is greater than or equal to the minimum support threshold given by the user to determine which candidate itemset is frequent.

Eclat [16,17] is a depth-first search algorithm that stands for Equivalence Class Clustering and bottom-up Lattice Traversal. Eclat is a more efficient and scalable version of Apriori algorithm that uses datasets in vertical layout. Each item has a list of transaction ids where the item appears. In vertical data format, every item is associated with its tidset (the set of transactions' identifiers that have the item). Eclat computes itemsets' support via tidset intersections. Furthermore, dEclat [18] is a more scalable version of Eclat that is better suited for dense datasets. Instead of intersections, it computes the differences between tidsets/diffsets. Diffset is the set difference between two related itemsets' tidsets/diffsets. When compared to tidset, diffset consumes less memory because it shrinks with each iteration. Because execution time and memory consumption are both important

elements in addressing FIM problems, diffset will enhance execution time and memory consumption because smaller sets will be mined in less time and consume less space. FIM algorithms are efficient when practicing on small datasets, but when the data gets bigger, they start to suffer. Hence, big data parallel processing frameworks including Hadoop, Spark, and Flink [19] are employed to accelerate the execution time of FIM algorithms and, consequently, the frequent itemset patterns are discovered in a reasonable period of time.

Many people mistakenly believe Hadoop and MapReduce [20,21] are synonymous; however, this is not the case. Hadoop was first released in 2007 as an open-source implementation of the MapReduce processing engine coupled with a distributed file system [4], but it has since grown into a vast web of projects covering every aspect of a big data workflow, including data collection, storage, processing, and more. MapReduce [22] is one of the most common parallel processing frameworks used to write the intermediate files generated to local disk to use these files in reduce phase; this process consumes lots of time. Unlike MapReduce, Spark stores the intermediate results in cache memory to speed up data processing. Spark also provides some other libraries such as Spark SQL, Spark streaming, Spark MLIB in one framework, and supports different languages such as Scala, Python, and Java. Some work is being done to support R as well.

Spark [23,24] is one of the most effective open-source big data tools used in running large-scale data analytics applications across clusters of nodes. Spark is a parallel processing framework that can handle both batch and real-time analytics and data processing workloads. It was designed to fill the shortcomings of Hadoop [21], especially in the data processing. Fault tolerance in Spark [25,26] aims to recover any failure or other loss in data that happened by rebuilding the RDDs by applying all the transformations found in the DAG (directed acyclic graph). Furthermore, Spark moves the processing and computations close to the data instead of moving the data itself to the computations; this is called data locality. Data locality is the process of moving the processing to data location instead of moving the data through the networks, and thus reducing the network communication and increasing the throughput of the system.

Bloom Filter (BF) [27] is a compact data structure that is used in the probabilistic representation of a set of variables to make sure whether an element in the set exist certainly or isn't existing in the set. BF is a very quick way of checking if something is in a set of data that gives one of two answers either: "x is not in this data set" or "x might be in this data set." BF provides a particularly useful optimization which means most of the queries can return "not found" without having to start looking through your indexes.

Recently, big data have impacted many aspects of life, including economics, politics, and culture, as well as science and technology. Big data have a lot of advantages in terms of production, trading, services, management, discovery, and so on; however, it also has a lot of challenges in terms of computer science and information technology, such as management, storage, and processing and analysis, security, and visualization. Data Mining generally and Frequent Itemsets Mining (FIM) especially play a critical role in that impact. Parallel FIM algorithms are still inefficient to cope with large amounts of big data, despite various techniques and methodologies addressing execution efficiency and memory optimization. FIM must be equipped with methodologies and solutions to operate effectively and reliably on such large datasets without out-of-memory and quickly enough to satisfy users. These requirements are significantly challenging for FIM. As a result, it is important to develop a new efficient algorithm for discovering the frequent itemset mining pattern as quickly as possible to meet customers' expectations with the wise use of memory.

In this paper, we introduce SHFIM, a new hybrid Spark-based method that attempts to improve FIM efficiency while detecting common patterns. SHFIM is divided into three phases, the first of which tries to extract the first frequent itemset. Next, the second phase concentrates on extracting the second most frequent itemset without candidate generation and converting the horizontal dataset layout to a vertical layout in a single step using BF. Because the candidate generation and vertical layout transformation processes are time expensive in conventional FIM algorithms, we combine them into a single step

using BF. Finally, in the third phase, SHFIM extracts the remaining frequent itemsets in parallel using diffset, which is more memory and time efficient. Because FIM is an iterative problem that requires many dataset scans, we chose Spark because it is 100 times quicker than Hadoop in data processing due to in-memory processing and lazy evaluation, which causes Spark to delay the execution of transformations until action is encountered. Furthermore, Spark can considerably improve the efficiency of repetitive algorithms that visit the same dataset regularly. As a result, the proposed algorithm is developed and tested using the Spark framework and written in such a way that it improves time by leveraging Spark best practices such as using broadcasted variables to speed up the lookup process and reduceByKey instead of groupByKey to avoid the shuffling steps. At last, we compared SHFIM performance with Spark-based algorithms (Eclat and dEclat) built on the same cluster, using four benchmark datasets (two sparse and two dense) to evaluate execution time.

The contribution work is summarized as follows (1) Developing and designing a new hybrid framework and algorithm on Spark. (2) Utilizing both the horizontal and vertical layout (3) Developing some improvements to eliminate the candidate itemset generation (4) SHFIM uses diffset to minimize the memory consumption (5) SHFIM uses shared variables to share datasets among cluster nodes (6) Conducting an Experiment on a multi-node cluster using four datasets. (7) Evaluate and discuss the proposed algorithm (SHFIM) against other FIM Algorithms.

The rest of the paper is structured as follows: Section 2 discusses FIM algorithms and their disadvantages. Section 3 demonstrates the fundamental concepts of FIM. SHFIM's proposed framework is depicted in Section 4. Section 5 addresses the Spark cluster requirements, dataset characteristics and SHFIM experiments. The evaluation and discussion are covered in Section 6. Section 7 depicts the suggested algorithm's complexity. Section 7 is dedicated to the conclusion and future work.

2. Related Work

This section provides an overview of the most common FIM algorithms used in large data mining. Section 2.1 addresses horizontal layout-based algorithms, Section 2.2 discusses vertical layout-based algorithms, and Section 2.3 discusses tree-based FIM techniques.

2.1. Horizontal Layout-Based Algorithms

EAFIM [28] is a parallel version of Apriori that has been implemented over Spark that enhances the Apriori by generating the candidate itemsets “on-the-fly” and applying a database reduction after each iteration. EAFIM outperforms YAFIM in almost all iterations and performs better than R-Apriori in all iterations except the second one due to the BF.

Adaptive-Miner [29] is a parallel version of Apriori that has been implemented over Spark. It makes execution plans before every iteration to decide which data structure should be used either BF or hash-Trees based on the cost of execution plans. The approach of taking decisions before every iteration makes Adaptive-Miner scalable and more efficient than YAFIM. Adaptive-Miner is used to generate the candidate itemsets in a separate step, then iterates over the dataset to fetch transaction by transaction to calculate the support of each candidate like YAFIM.

HFIM [30] is a hybrid version of Apriori that has been developed over Spark. It uses the vertical layout rather than the horizontal layout to mitigate the iterative dataset scanning of Apriori. The vertical layout takes less amount of storage and no need to iterate over the dataset many times. A dataset in the horizontal layout is partitioned and distributed over the cluster to generate the candidate itemsets in the workers rather than the driver and make a dataset reduction to keep only the items that are frequent only. Datasets are shared among cluster nodes using broadcasting variables. HFIM outperformed the YAFIM algorithm because of the usage of the vertical layout.

DFIMA [31] is a parallel version of the Apriori algorithm that has been developed over the Spark framework. It is a Matrix-based pruning technique that decreases the number of

generated candidates' itemsets. DFIMA outperformed the PFP [32] concerning speed and scalability. Broadcasting variables has been used to enhance the efficiency of the algorithm. DFIMA and PFP seem to run faster when `min_sup` is low.

R-Apriori [33] is a parallel version of the Apriori algorithm that has been implemented over Spark which enhances the execution time of the YAFIM algorithm by adding a BF in the second iteration. As a result, it reduces the runtime due to eliminating the time taken by candidate generation at the second iteration. R-Apriori outperformed YAFIM in the second iteration and performs the same as YAFIM in all other iterations.

YAFIM [34] is a parallel version of the Apriori algorithm that has been developed on the Spark framework. Spark is specially designed for in-memory parallel computing to support iterative and interactive data mining. All Apriori-based algorithms that have been implemented over MapReduce, YAFIM achieved 18x speedup on average for various benchmarks. YAFIM generates the candidate itemsets in a separate step through applying a Cartesian product between (k-1 frequent itemset) and (k-1 frequent itemset), then iterates over the dataset (transaction by transaction) to calculate the support of each candidate.

2.2. Vertical Layout-Based Algorithms

DFP [35] is a parallel version of Eclat that was introduced as a distributed approach to increase the mining efficiency for association rules. The transmission cost will be increased if the communication between clients (nodes) increased. The suggested DFP method effectively avoided this difficulty by moving the mining result from the executing client to the server. The experiment findings also revealed that the suggested technique had much greater computational efficiency than DistEclat and BigFIM. The suggested technique decreases computation efficiency by estimating the required memory and clients. By calculating the maximum number of TIDs for each client, the DFP algorithm overcomes memory constraints and repetitive scans, and the extra TIDs are handled by an extended mechanism known as MP.

RDD-Eclat [36] is a parallel Eclat algorithm entitled RDD-Eclat and the implementation of its five variations on the Spark RDD framework. EclatV1 is the first version, while the others are EclatV2, EclatV3, EclatV4, and EclatV5. Each version is the consequence of a new technique and heuristic being applied to the preceding variant. Each variant is the consequence of a new technique and heuristic being applied to the preceding variant. After EclatV1, the filtered transaction approach is used, and in EclatV4 and EclatV5, heuristics for equivalence class partitioning are used. It has been shown that using the filtered transaction approach to reduce the size of the original dataset enhances the performance. Furthermore, the equivalence class partitioning techniques considerably reduced the execution time.

SVT [37] is a hybrid algorithm that has been implemented over the Spark framework. Unlike the traditional vertical Layout, it follows the way of switching between the Eclat (tidset) and dEclat (diffset) relying on the dataset density. If the dataset is sparse, it uses Eclat. Otherwise, it uses dEclat. The experimental results proved that the SVT outperforms the YAFIM in execution time performance.

PEclat [38] is a hybrid algorithm that has been implemented over the Hadoop MapReduce framework. PEclat works on the vertical layout datasets to increase the speed of the algorithm. Unlike the traditional algorithms that use just one vertical format either tidset or diffset, it uses the two formats simultaneously on the item level rather than the dataset level. It works without putting into consideration if the dataset is dense or even sparse. PEclat follows the strategy of mixset in which it decides to continue with diffset or tidset based on the tidset or diffset size. If the size of tidset is greater than or equal to the size of diffset, then continue using tidset, and the opposite is right.

BigFIM [39], introduces two algorithms in the area of big data and frequent pattern mining over Hadoop (Map Reduce). The first algorithm is the DistEclat (Distributed Eclat) which is a new parallel version of Eclat that has been developed on Hadoop. The second algorithm is the BigFIM, a scalable hybrid algorithm between Apriori and Eclat to mine

big datasets. The experimental results proved that DistEclat is the fastest among all the algorithms in dense datasets, however, it may fail because it requires lots of memory.

2.3. Tree-Based Algorithms

Map-Optimize-Reduce [40] is a parallel version of FP-Growth that has been implemented over Hadoop MapReduce. It is based on the Map-Optimize-Reduce framework which decreases the search time and enhances the performance of the MapReduce framework. It builds a CAN tree that makes the FP-Growth algorithm performs better than using FP-Tree and enhances the database efficiency through making some data preprocessing and normalization.

DFPS [41] is a parallel version of FP-Growth that has been implemented over the Spark framework. It works in the same way as MapReduce; it distributes the workload and computation cost over the cluster. It allows each node to build the conditional FP-tree and mine the tree to extract the frequent pattern. DFPS showed good performance in terms of time and scalability against Apriori.

PFP [32] is a parallel version of FP-Growth [42] that has been developed over the Hadoop MapReduce framework. PFP generates frequent itemset without candidate generation step through constructing an FP-Tree into the workers' memory. It performs well when the memory is big enough in data centers and may fail if the memory cannot fit the tree.

Table 1 shows a comparative analysis of FIM algorithms. Apriori has a problem with multiple database scanning, which is time-consuming. Eclat uses the vertical layout, which is faster than the horizontal layout, however, it has a problem with the dense datasets. It takes lots of time due to the tidset intersections. If the memory is not sufficient, Eclat may fail. dEclat is a scalable version of Eclat using diffset. Although dEclat is memory-wise and can mine data in less amount of time, it shows deficient performance with sparse datasets. Researchers do not recommend starting with Eclat or dEclat and start with a memory-wise (Breadth-First Search) algorithm like Apriori instead, then switch to Depth First Search. Thus, this research aims to propose SHFIM hybrid spark-based algorithm that enhances the execution time of extracting the frequent pattern from big datasets.

FIM (Frequent Itemset Mining) offers three approaches: horizontal, vertical, and tree based. Horizontal like Apriori, vertical like Eclat, and tree-based, like PFP. Each algorithm has advantages and disadvantages; for example, Apriori is the best in terms of memory, but the worst in terms of execution time. Eclat, on the other hand, is the fastest in terms of execution time, but it requires extensive processing due to the intersections. PFP examines the dataset once to generate a tree data structure. Although PFP is effective in some circumstances, it has several drawbacks. When at least a set of projected transactions does not fit into the machine's memory, it fails. When the projected transactions significantly overlap, the communication overhead can be very high, because in that situation overlap of data is sent to the network several time [2,39]. This variance and difference in algorithms encourage researchers to develop a new type of algorithm that can utilize the advantages of current algorithms known as Hybrid. Therefore, we designed our hybrid algorithm to start as Apriori to take advantage of memory early on, then move data to a BF to avoid memory and time overhead in producing candidates and layout transformation, and finally, leverage diffset to take advantage of memory and time together. Diffset enhances execution time and memory consumption because it shrinks itemsets into smaller sets that will be mined in less time and consume less space [18]. Furthermore, we designed and developed the algorithm using Spark best practices to reduce network traffic and shuffling between worker nodes.

Table 1. FIM algorithms comparative analysis.

FIM Algorithm	Advantages	Disadvantages	Framework	Algorithm
BigFIM [39]	<ul style="list-style-type: none"> A scalable hybrid FIM algorithm can mine truly big data. 	<ul style="list-style-type: none"> Cannot deal with lower frequencies and sometimes out of memory happens. 	MapReduce	Apriori/Eclat
Dist-Eclat [39]	<ul style="list-style-type: none"> The fastest algorithm among others regardless of the candidate generation time. Accelerate the speed using a strategy of simple load balancing. 	<ul style="list-style-type: none"> Cannot deal with a large amount of data due to the memory and processing needed. 	MapReduce	Eclat
YAFIM [34]	<ul style="list-style-type: none"> More scalable and rapid than MapReduce 	<ul style="list-style-type: none"> Shows bad performance in low minimum supports. Requires multiple dataset scan. 	Spark	Apriori
R-Apriori [33]	<ul style="list-style-type: none"> Eliminate the time taken to generate the candidates of the second iteration by adding Bloom Filter. Outperform YAFIM in the second iteration. 	<ul style="list-style-type: none"> Generate many infrequent candidates. Requires multiple dataset scan. 	Spark	Apriori
DFIMA [31]	<ul style="list-style-type: none"> Vector data representation Matrix-based pruning technique. Scalable for Big Data. 	<ul style="list-style-type: none"> Shows bad performance when the minimum support is low. 	Spark	Apriori
PECLAT [38]	<ul style="list-style-type: none"> Can deal with sparse and dense datasets. Saves memory and time through applying one of the approaches (ECLAT, DECLAT) on itemset level not on dataset level. Workload balancing through itemset ordering. 	<ul style="list-style-type: none"> The computation cost of the mixset can be time-consuming. The communication cost between nodes is high. 	MapReduce	Eclat/dEclat
HFIM [30]	<ul style="list-style-type: none"> Utilizes the vertical layout and horizontal layout. Uses the broadcast variables. Workload balancing. Less memory complexity. Avoid iterative dataset scans. 		Spark	Apriori
DFPS [41]	<ul style="list-style-type: none"> No communication loads. The skewness of Workload. 	<ul style="list-style-type: none"> Cannot deal with low minimum supports and sometimes it fails. 	Spark	FP-Growth
Adaptive-Miner [29]	<ul style="list-style-type: none"> Switch between a different approach based before each iteration to improve the execution time. 	<ul style="list-style-type: none"> Consume some time to identify which approach should apply. Requires multiple dataset scan. 	Spark	Apriori
SVT [37]	<ul style="list-style-type: none"> Reduces the cost of communication across nodes in the cluster. Distributes workload among the clusters' nodes. 	<ul style="list-style-type: none"> Computation cost and memory usage are high. Needs more memory in low minimum supports. 	Spark	Eclat/dEclat
EAFIM [28]	<ul style="list-style-type: none"> On-the-fly candidate generation. Dataset reduction after each iteration. Shows good performance in higher minimum supports. 	<ul style="list-style-type: none"> Requires multiple dataset scan. Shows bad performance in lower minimum supports. 	Spark	Apriori
RDD-Eclat [36]	<ul style="list-style-type: none"> Shows better performance in sparse datasets. Increases ECLAT performance by applying partitioning and some heuristics. 	<ul style="list-style-type: none"> Shows bad performance in dense datasets. May fail if the memory is not sufficient. 	Spark	Eclat
Map-Optimize-Reduce [40]	<ul style="list-style-type: none"> Increases dataset performance by applying data preprocessing. CAN tree reduces the scanning time of the dataset. Balancing the workload across the cluster's nodes. EPC based optimizer is used to enhance the MapReduce framework 	<ul style="list-style-type: none"> The communication cost between nodes is high. Needs more memory in low minimum supports. 	MapReduce	FP-Growth

3. Preliminaries

The problem of the frequent pattern mining is to find correlations between elements in a database. The problem is summarized as follows: Find all patterns P that appears in at least a fraction of the transactions in a database D containing transactions $T_1 \dots T_N$. The list of notations used in this paper is shown in Table 2.

Table 2. List of notations.

Notation	Description
P	a Pattern in D
D	a dataset of transactions
T	a transaction in D
n	Number of items in D
i	an item
R	a set of records in D
m	Number of records in D
r	a record in D
I	an itemset in D
k	Number of items in I and it is also the iteration number
σ	Support
min_sup	Minimum support
t	Tidset of an itemset
d	Diffset of an itemset
f_k	K Frequent itemsets
N	The number of itemsets in k frequent itemsets
C	K Candidate itemset
O	Big O notation
k_n	The maximum number of iterations

3.1. Definition (Pattern)

A pattern is described as a collection of objects, events, or items that occur frequently in a database. Formally, a pattern P in a database D is defined as a subset of elements $P \subseteq \{i_1 \dots i_n\} \in D$ that represent valuable data properties [43].

3.2. Definition (Frequency of a Pattern)

The frequency of a pattern P is defined as the number or the percent of data records that contain the subset of items indicated by P . In other words, a set of items $I = \{i_1 \dots i_n\}$, in a database D consisting of $R = \{r_1 \dots r_m\} \in D$ data records, with each record containing a set of items $\forall r \in R: r \subseteq I$, the frequency of P is defined as $|\{\forall r \in R: P \subseteq r, r \subseteq I \in D\}|$. The FIM process is extremely difficult due to the vast number of candidate patterns (itemsets) that must be processed [43].

Given an n -item dataset, the number of patterns (or itemsets) of size k is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ for any $k \leq n$. As a result, with such a dataset, the total number of possible patterns is $2^n - 1$, and the complexity of locating patterns of interest is in exponential order. When the frequency of each pattern is determined, the complexity increases to $O((2^n - 1) \times m \times n)$ for a dataset of m records and n items. To provide some light on the complexity, a dataset of only 33 items (which is a small number of items) yields more than 8.5×10^9 possible patterns, which is more than the world population in 2018. (Almost 7.7 billion people) [43].

3.3. Definition (Itemset)

Assume that I is a set of items. An itemset is a collection of items from I . A k -itemset is an itemset created from k items. An itemset S is considered a subset of an itemset I if all of the items in S are also in I , denoted as $S \subseteq I$ [18].

3.4. Definition (Frequent Itemset)

Consider D is a dataset of transactions, each transaction T contains a set of items. I is an itemset and a subset of transaction T . I is frequent if its support (the number of transactions contains I) defined as, $\sigma(I) = |\{T \mid I \subseteq T \wedge T \in D\}|$ is greater than or equal to minimum support (min_sup) given by the user, denoted as $FI = \{I \mid \sigma(I) \geq \text{min_sup}\}$ [14].

As a running example, consider the given dataset D in Table 3. D contains 4 transactions $\{T_1 \dots T_4\}$; each transaction contains a set of items. Our goal is to discover the frequent itemsets whose support is greater than or equal to min_sup . Let $\text{min_sup} = 3$, so that the first frequent itemsets = $\{A\}, \{C\}, \{D\}, \{F\}$, second frequent itemsets = $\{A, C\}, \{A, D\}$, and third frequent itemsets = $\{A, C, D\}$.

Table 3. Dataset D items.

TID	Items
1	A, B, C, D, F
2	A, C, D, F
3	B, C, D, F
4	A, C, D

3.5. Definition (Tidset)

Consider D to be a dataset containing a set of items i . The tidset of an itemset I is the set of transactions identifiers that have the item, represented by $\text{tidset}(I) = \{t.\text{tid} \mid t \in D, I \subseteq t\}$. Thus, $\sigma(I) = |\text{tidset}(I)|$, denoted as Equation (1). Additionally, k -itemset's tidset is formed by intersecting the tidsets of two $(k - 1)$ -itemsets that share the first $k - 2$ items, represented by Equation (2) [16].

$$\sigma(ABC) = |t(ABC)| \tag{1}$$

$$t(ABC) = t(AB) \cap t(AC) \tag{2}$$

3.6. Definition (Diffset)

Consider $t(A)$ is the tidset of A . The set of transaction ids that are present in $t(A)$ but not in $t(AB)$ is known as the diffset $d(AB)$, $d(AB) = t(A) - t(AB) = t(A) - t(B)$, denoted as Equation (3). Let $t(AB)$ and $t(AC)$ are present in the dataset, AB and AC are sharing the same prefix A . The target is to compute the $\sigma(ABC)$. By definition, $\sigma(ABC) = \sigma(AB) - |d(ABC)|$, denoted as Equation (4). As a result, only $d(ABC)$ is required to determine $\sigma(ABC)$. It has been demonstrated that $d(ABC) = d(AC) - d(AB) = t(AB) - t(AC)$, denoted as Equation (5) [18] so that the diffset and support of a produced itemset can be obtained from the tidsets or diffsets and supports of its produced itemsets.

$$d(ABC) = t(AB) - t(AC) \tag{3}$$

$$\sigma(ABC) = \sigma(AB) - |d(ABC)| \tag{4}$$

$$d(ABC) = d(AC) - d(AB) \tag{5}$$

Figure 1 presents a comparison of tidset and diffset, as well as how diffset works in vertical data representation. We can utilize diffset from the very beginning, as shown in the figure, or from any other level. Assume we start at tidset level 2 where $k = 2$ to compute the 2-itemset (an itemset contains two elements), we determine that $d(AC) = t(A) - t(C) = \{1,3,4,5\} - \{1,2,3,4,5,6\} = \{\}$ by Equation (3), and $\sigma(AC) = \sigma(A) - |d(AC)| = 4 - 0 = 4$ by Equation (4). AC tidset is achieved by $t(AC) = t(A) \cap t(C) = \{1,3,4,5\} \cap \{1,2,3,4,5,6\} = \{1,3,4,5\}$ by Equation (2), and $\sigma(AC) = |t(AC)| = 4$ by Equation (1). As a result, $d(AC)$ has an empty set, whereas $t(AC)$ has a set of four tids, however, they have the same support. In Figure 1, 1-itemset diffset has 7 entries, whereas tidset has 22. Diffset has 12 entries for a 2-itemset, whereas tidset has 34 elements. As a result, diffset is at

least two or three times lower in size than tidset, making it more memory efficient and faster to compute.

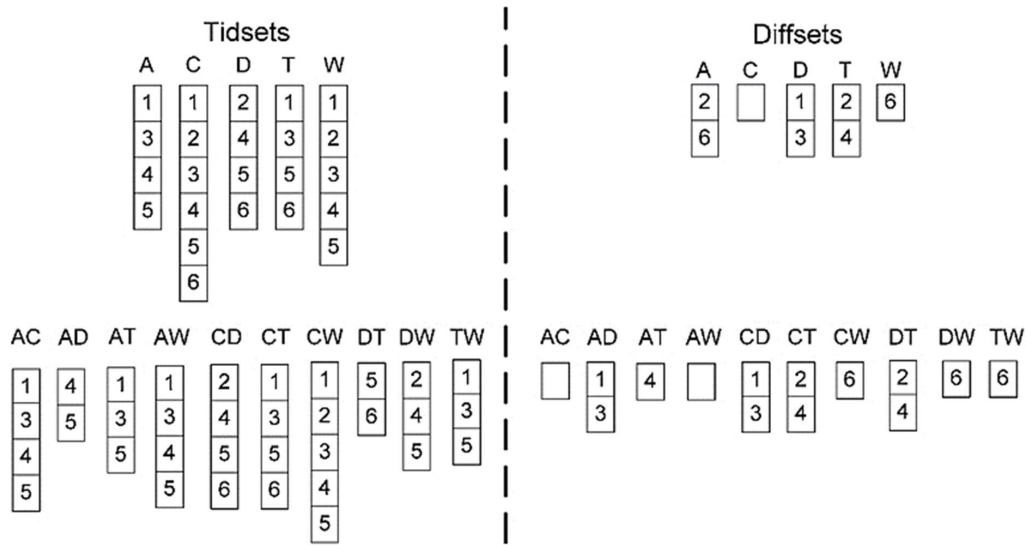


Figure 1. Tidset and diffset itemsets [18].

4. SHFIM Proposed Framework

In this section, we introduce SHFIM hybrid algorithm to implement a frequent pattern mining framework for analyzing big data in a parallel and distributed way over Spark. The main goal of SHFIM is to improve the execution time performance and reduce the search space to extract the frequent Itemset from big data. SHFIM algorithm is a Scala-based algorithm implemented over Spark cluster with Spark best practices to reduce the execution time required to extract the frequent itemsets.

Unlike Eclat and dEclat algorithms, SHFIM can handle thousands and even millions of variable-length transactions without leaking or spilling memory. The SHFIM framework is presented in Figure 2 and is divided into three stages: (1) singletons extraction phase, (2) second frequent itemset extraction phase in vertical arrangement, and (3) K frequent itemset extraction phase. The next subsections go through each of these three phases in depth.

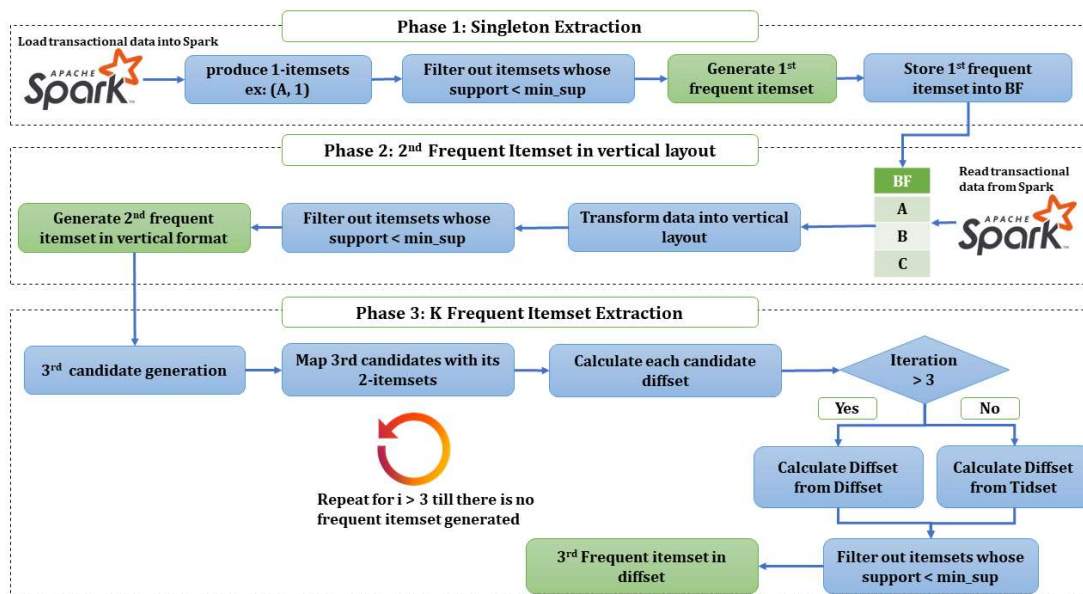


Figure 2. SHFIM Framework.

4.1. Singletons Extraction Phase

The goal of this phase is to extract the first frequent itemset. It is a single iteration that uses a single map function and a single reduce function to extract the first frequent itemset. This phase's inputs are the dataset in horizontal format and the user-defined `min_sup`. Each line in the dataset represents a transaction, which has a set of items separated by spaces, the first of which is the transaction id. For example, transaction #10 should look like {10,22,24,35,39,71,91}, where 10 is the tid (Transaction id) and the rest are the items.

The SHFIM algorithm pseudo-code is shown in Algorithm 1, in which SHFIM begins reading the dataset and assigns each partition of data to a worker node in the Spark cluster. To flatten the itemsets, each worker node uses a `flatMap` function on its data. When the `getFirstFrequentItemset` method is used, it returns an RDD containing the first frequent itemsets in a (key, value) pair, where the key is an itemset and value is its support. In other words, (A, 5) is a 1-itemset pair containing an itemset (A) with support of five.

Algorithm 1. Discover frequent pattern using SHFIM

Input: *D*: Dataset of transactions, *min_sup*: minimum support threshold.

Output: frequent itemsets: list of frequent itemsets.

dataRDD ← Read data from HDFS

singletons ← `getFirstFrequentItemset` (*dataRDD*, *min_sup*)

If *singletons* = \emptyset **then**

 | `system_exit` ()

end if

singletonsList ← `broadcast` (*singletons*)

secondFrequentTidItemsetRDD ← `findPairsBloomFilter` (*dataRDD*, *singletonsList*, *min_sup*)

hasCoverage ← `false`

k ← 2

kFrequentItemsetRDD ← `secondFrequentTidItemsetRDD`

While *hasCoverage* = `false` **do**

 | *candidateItemsets* ← `generatekCandidates` (*kFrequentItemsetRDD*)

If *candidateItemsets* = \emptyset **then**

 | *hasCoverage* ← `true`

else

 | *k* ← *k* + 1

 | *candidateItemsetsBC* ← `broadcast` (*candidateItemsets*)

 | *assignCandidateToItemsetRDD* ← `assignCandidateToItemset` (*candidateItemsetsBC*, *kFrequentItemsetRDD*)

 | **If** *k* = 3 **then**

 | *kFrequentItemsetRDD* ← `getDiffsetFromTid` (*assignedCandidatesToItemsetRDD*, *min_sup*)

 | **else**

 | *kFrequentItemsetRDD* ← `getDiffsetFromDiff` (*assignedCandidatesToItemsetRDD*, *min_sup*)

 | **end if**

end if

end while

Figure 3 shows the visual workflow of this phase in SHFIM, which reads the data in horizontal layout format from HDFS and loads it into Spark RDD, where a map function is applied to construct a key-value pair. Following that, it uses a reduce function to aggregate the values based on their keys. At the end of this phase, SHFIM filters out itemsets/keys whose support is less than `min_sup`, and the first frequent itemset is retrieved and stored in the BF.

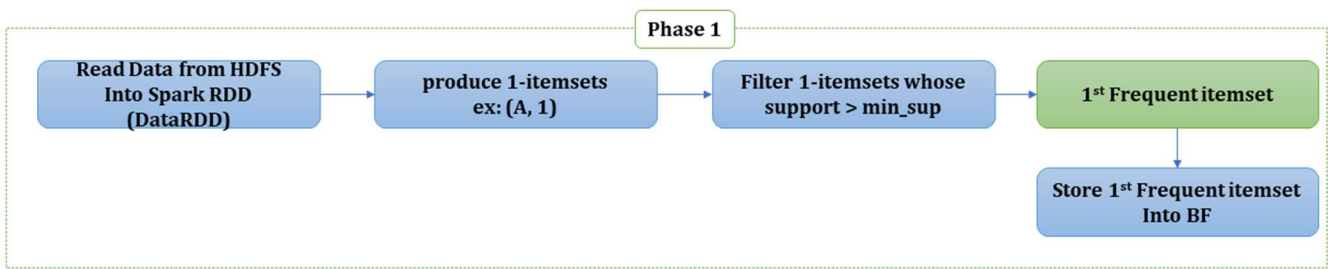


Figure 3. SHFIM Phase one workflow.

Figure 4 shows an example of how the SHFIM algorithm works in the first phase and how data is processed in parallel. SHFIM begins by reading transactional data from HDFS (Hadoop Distributed File System) in horizontal format and loading it into a Spark RDD. Each mapper accepts a collection of transactions (e.g., Mapper-1 takes transactions T1 and T2). To separate each transaction into discrete items, mappers use a flatMap function on each record in the RDD. The map function is used to create corresponding (key, value) pairs, where the key is an itemset and value is 1. As running example of a dataset D in Table 3, T1 = {A, B, C, D, F} is mapped into (A,1), (B,1), (C,1), (D,1), and (F,1). Subsequently, the reducers use the reduceByKey method to aggregate the data based on the key and filter away irrelevant itemsets, leaving only those with support greater than or equal to min sup. If min sup = 3, SHFIM stores in a BF only (A, C, D, F) because their support is more than or equal to 3. Finally, the BF is shared among the cluster’s machines so that it can be accessed locally by each worker node.

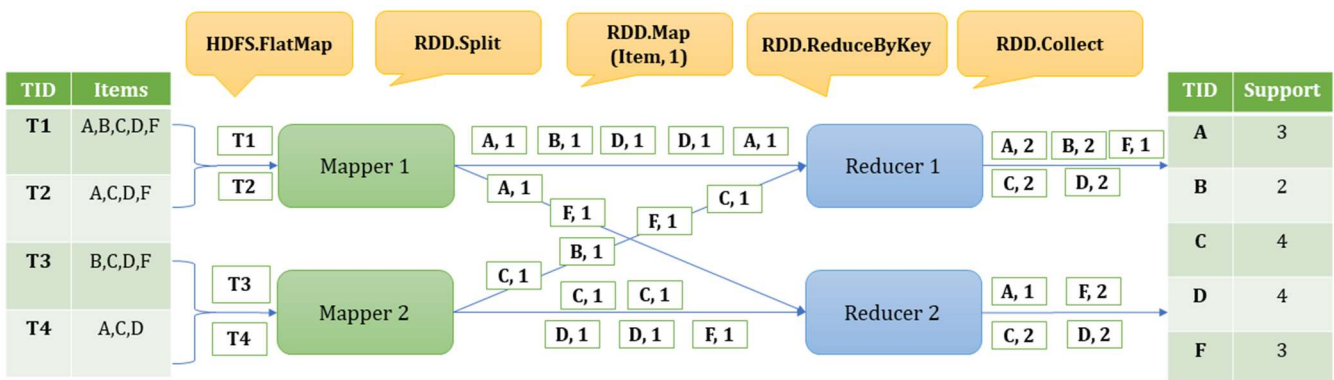


Figure 4. Spark MapReduce architecture for Phase I.

4.2. 2nd Frequent Itemsets in a Vertical Layout Phase

This phase aims to extract the second frequent itemset in a vertical layout. The input of this phase is the first frequent itemset (Singletons). Mappers take each transaction and prune it according to the singletons stored in the BF, producing all feasible candidates together with their tid in pairs. This step eliminates the process of candidate generation, which is time-consuming in most FIM algorithms.

The workflow of this phase is illustrated visually in Figure 5, where the BF reads every item in each transaction from the horizontal layout dataset. Afterward, BF prunes items (that are not present in BF) and then produces key-value (key is the itemset and value is the TID) pairs candidates from items that are remained after pruning. Next, reducers aggregate the values based on their keys to convert the data into a vertical layout such as (Itemset, {list of tids}). At the end of this phase, SHFIM filters out the itemsets whose support is less than min_sup, and then the second frequent itemset is extracted. Line 7 in Algorithm 1 has a function called findPairsBloomFilter, which accepts three parameters (data RDD, singletons, min_sup) and returns the second frequent itemset in vertical layout tid format.

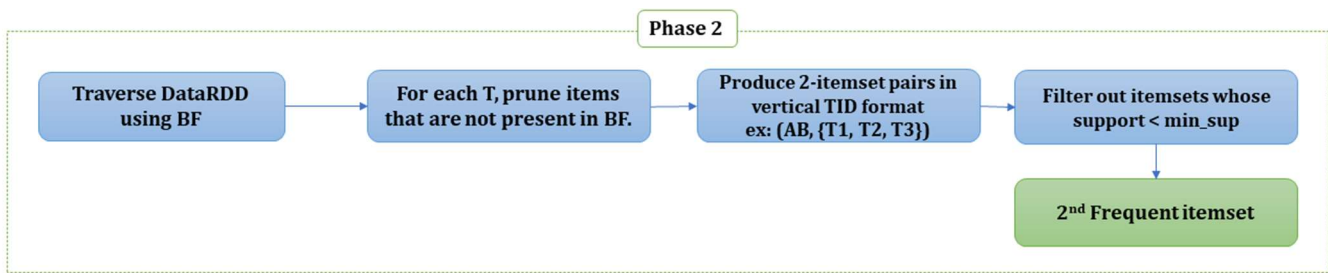


Figure 5. SHFIM phase two workflow.

Figure 6 illustrates how SHFIM extracts the second frequent itemsets in a vertical format. A Spark RDD is used to load the transactional dataset from HDFS. Mappers take each transaction and prune it so that only the items listed in the BF are retained, and produce all feasible (key, value) pairs from the trimmed transaction, where the key represents the newly generated itemset and the value represents its tid (Transaction id). Subsequently, Reducers use the reduceByKey transformation to combine all pairs based on their key to build a new RDD containing a collection of (key, value) pairs where the key is the itemset and value is its tidset and filter out itemsets whose support is less than min_sup.

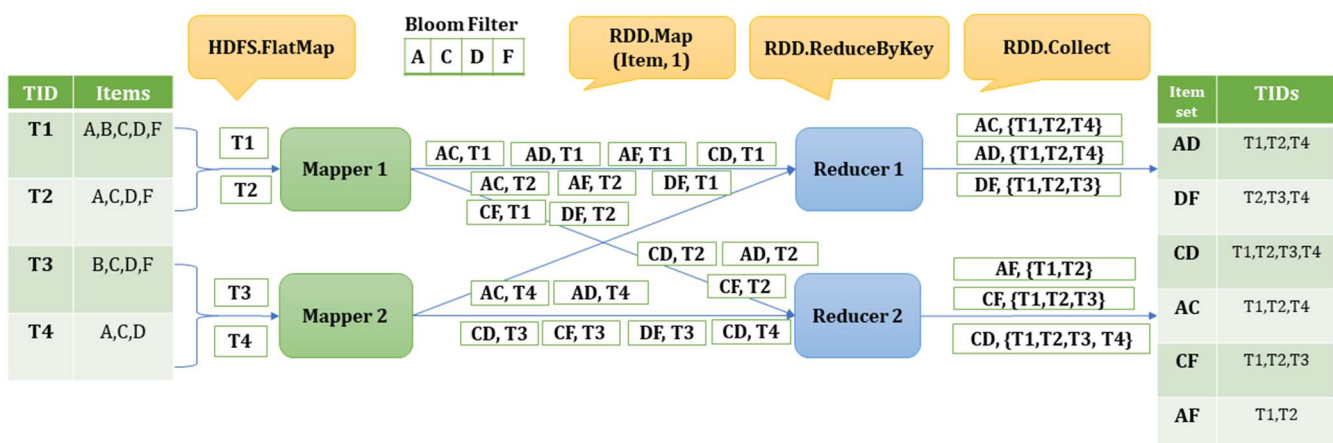


Figure 6. Spark MapReduce architecture for Phase II.

As a running example, Mapper-1 takes T1: (A, B, C, D, F) and T2: (A, C, D, F), then it prunes item B because it does not exist in the BF (A C D, F). Mapper-1 yields all pairs for T1 and T2 as (key, value) pair, where the key is itemset and value is tid, such as (AC, T1), (AD, T1), (AF, T1), (CD, T1), (CF, T1), (DF, T1), (AC, T2), (AD, T2), (AF, T2), (CD, T2), (CF, T2), (DF, T2). Reducers combine all the generated pairs based on their key and generate a new RDD of (key, value) pairs that contains (AC, {T1, T2, T4}), (AD, {T1, T2, T4}), and so on, and filter out itemsets whose support less than min_sup. Finally, the second frequent itemsets in vertical format are stored in Spark RDD and ready for the next phase.

4.3. K Frequent Itemset Extraction Phase

The goal of this phase is to use diffset to extract the K frequent itemsets, where $K \geq 3$. The input of this phase is the $(k - 1)$ frequent itemset in the form of key value pair RDD, where the key is the itemset and value is the tidset.

Figure 7 presents the workflow of this phase which begins with producing the candidate k-itemsets from $(k - 1)$ frequent itemsets and broadcasts the generated candidate list among cluster nodes. Next, mappers traverse each itemset in $(k - 1)$ frequent itemset to map with their corresponding candidate k-itemset. As a result, a key value pair RDD is generated, where key is the candidate k-itemset and value is its mapped $(k - 1)$ itemset.

Afterward, reducers integrate the values in the paired RDD based on their key to produce another key-value pair, but the key, in this case, is the candidate k -itemset and the value represents a list of all corresponding $(k - 1)$ frequent itemsets. Finally, the diffset is computed for each candidate k -itemset based on Equations (3) and (5) in the preliminaries section. At the end of this phase, SHFIM filters out the itemsets/keys whose support is less than min_sup , and the k -frequent itemset is extracted. Because the FIM is an iterative approach, SHFIM repeats the third phase for each $k > 3$ to extract the remaining frequent itemsets, and this process lasts until no more frequent itemsets can be obtained.

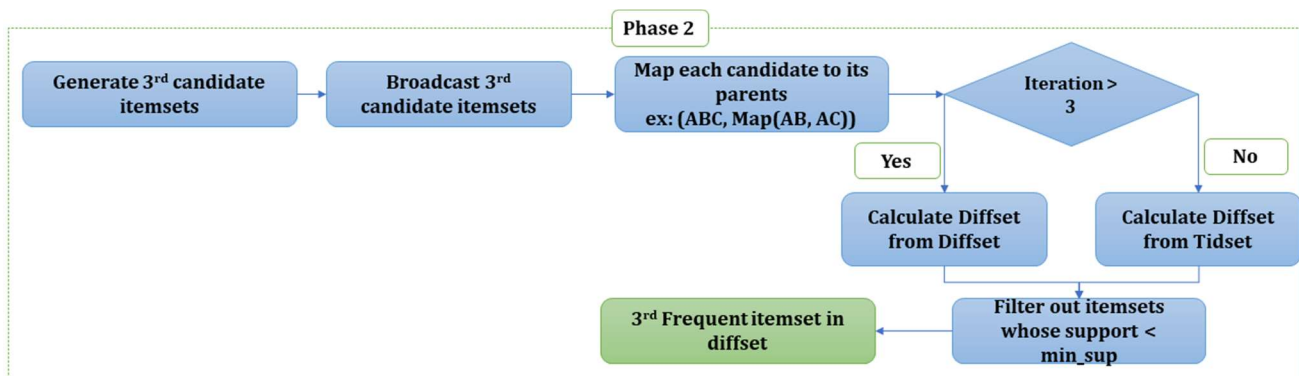


Figure 7. SHFIM phase three workflow.

SHFIM uses the function `generatesKCandidates` in Algorithm 1 line 12 to generate candidate k -itemsets. The candidate generation rule states that every two $(k - 1)$ -itemsets must share the same prefix and be different in the last. In other words, let ABC be a 3-itemset candidate derived from the 2-itemsets AB , AC , which share the same prefix (A) but differ in the last element B and C . In Algorithm 1 line 18, SHFIM calls the function `assignCandidateToItemset`, which returns a $(\text{key}, \text{value})$ pair RDD. This function uses a `flatMap` to assign or map every candidate k -itemset to each frequent $(k - 1)$ -itemset. SHFIM invokes a function `getDiffsetFromTID` where $K = 3$ and `getDiffsetFromDiff` where $K > 3$ on lines 19–23. Finally, a filter function is invoked to filter out itemsets whose support is greater than or equal to the min_sup .

Figure 8 shows an example for the third phase and how SHFIM extracts the frequent k -itemsets. Mapper-1 takes AD , AC , and DF as vertical itemsets and then applies a filter on the candidate RDD to keep only candidates whose itemsets are already in the second frequent itemset, such as ABC , which is a candidate 3-itemset that is derived from AB and AC , both of which are present in the frequent 2-itemset. After applying the filter function, mapper-1 generates a $(\text{key}, \text{value})$ pair RDD in which key is the candidate list and value is an itemset such as $(\{ACD, ACF\}, AC)$ is derived from AC , whereas $(\{ACD, ADF\}, AD)$ is derived from AD . On contrary, DF has an empty list of candidates and will be ignored because there are no other 2-itemsets that share the same parent or start with (D) . Consequently, a `flatMap` function is applied on the RDD to split the candidates into separated key-value pairs. As a running example, $(\{ACD, ACF\}, AC)$ is transformed using `flatMap` into two pairs (ACD, AC) , (ACF, AC) . Reducer-1 combines (ACD, AC) and (ACD, AD) to generate $(ACD, \text{Map}(AC \geq \{T1, T2, T4\}, AD \geq \{T1, T2, T4\}))$. Next, mapper-1 calculates the diffset of each candidate itemset. Therefore, ACD is a candidate itemset has both AC and AD , $d(ACD) = t(AC) - t(AD) = \{T1, T2, T4\} - \{T1, T2, T4\} = \{\}$, using Equation (4), whereas $\sigma(ACD) = s(AC) - |d(ACD)| = 3 - 0 = 3$, using Equation (5). Finally, repeat the third phase in each subsequent $k > 3$ until no frequent itemsets are existed.

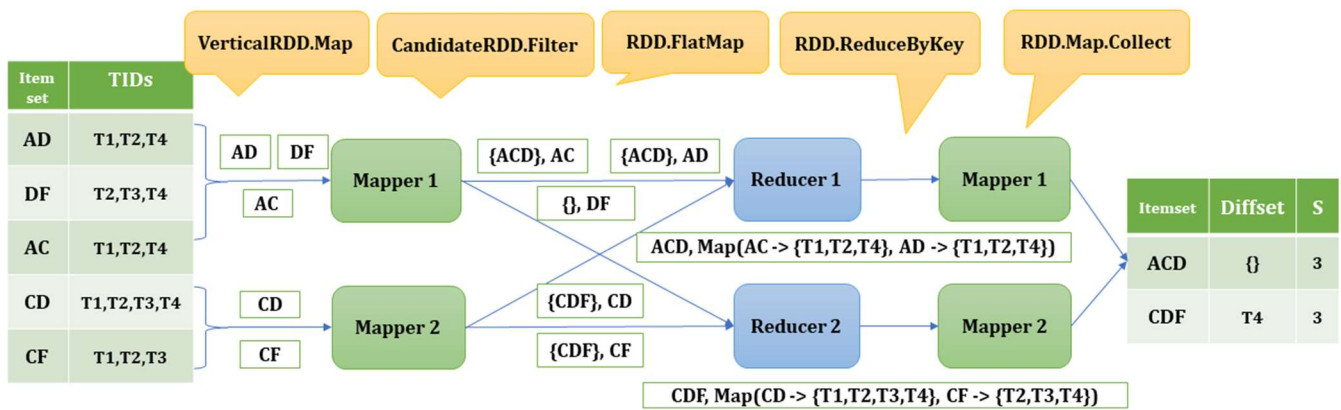


Figure 8. Spark MapReduce architecture for Phase III.

5. Performance Evaluation

This section evaluates the performance of the SHFIM algorithm by comparing its execution time performance against Eclat and dEclat, where Eclat is a Spark-based vertical layout FIM algorithm depending on intersections between itemset tidset. On the other hand, dEclat is a Spark-based vertical layout FIM algorithm depending on calculating differences between itemsets. Moreover, we implemented Eclat and dEclat on the same environment of the SHFIM algorithm to make sure that the comparison between them all is done on the same cluster using same specifications.

To analyze the SHFIM algorithm, we set up a cluster of 2 nodes. They have Intel(R) Core (TM) i3-4500U CPU @ 1.80GHz, 2401 MHz, 2 Core(s), 4 Logical Processor(s). All nodes have 6 GB RAM, and a 500GB hard disk. The worker nodes are installed on Ubuntu 16.04, Hadoop 2.6.0, Spark 2.2.0, and Scala 2.11.8.

5.1. Dataset

In this section, we present the characteristics of datasets that have been used in the evaluation. The proposed solution (SHFIM) has been tested using four datasets [44] which are the most common datasets and the benchmark in all frequent pattern mining algorithms in big data.

Table 4 shows the characteristics of the datasets used in performance evaluation. The first dataset is a sparse T1014D100k dataset generated using IBM’s generators, whereas, the second dataset is a Retail dataset which is a sparse dataset containing transactions that happened from the half of Dec 1999 to the end of Nov 2000 for a UK-based online non-store system. While the third is a chess dataset, which is a dense dataset containing the chess king vs. king and rook end positions. Finally, the fourth dataset is the dense mushroom dataset for 23 gilled mushroom species. The dataset selection criteria we use are to apply the algorithm to both sparse and dense datasets to evaluate algorithm efficiency and analyze the influence of each dataset on our algorithm.

Table 4. Properties of the datasets used in performance evaluation.

Dataset	No. of Transactions	No. Of Different Items	Density	Type
Mushroom: (http://fimi.uantwerpen.be/data/mushroom.dat) (accessed on 1 January 2022)	8124	119	Dense	Real-life
Chess: (http://fimi.uantwerpen.be/data/chess.dat) (accessed on 1 January 2022)	3196	75	Dense	synthetic
T1014D100k: (http://fimi.uantwerpen.be/data/T1014D100K.dat) (accessed on 1 January 2022)	100,000	870	Sparse	Real-life
Retail: (http://fimi.uantwerpen.be/data/retail.dat) (accessed on 1 January 2022)	87,988	16,470	Sparse	Real-life

5.2. Experiment and Result

In this section, we will show the experimental results of SHFIM algorithm against Eclat and dEclat in terms of execution time.

Figure 9a compares SHFIM, dEclat, and Eclat in different min_sup starting from 7% and ending at 0.6%. At min_sup (7–2%), FIM algorithms generate the frequent itemset in a single iteration. Similarly, Eclat and dEclat do the same operation (convert the dataset from horizontal to a vertical layout, then compute the support), and they nearly reach the same execution time. Unlike Eclat and dEclat, SHFIM only computes each itemset support, then filters out itemsets with support less than min_sup and generates singletons without transforming the dataset into a vertical layout. From min_sup (1%), the iterations start to increase and require Eclat and dEclat to generate the second candidate itemsets by applying intersections (Eclat) or differences (dEclat) between itemsets to extract the second frequent itemset. SHFIM, on the other hand, eliminates the time required to produce the candidate itemsets for the second frequent itemset using the BF. As a result, it saves a lot of memory because it avoids loading the memory with second itemset candidates.

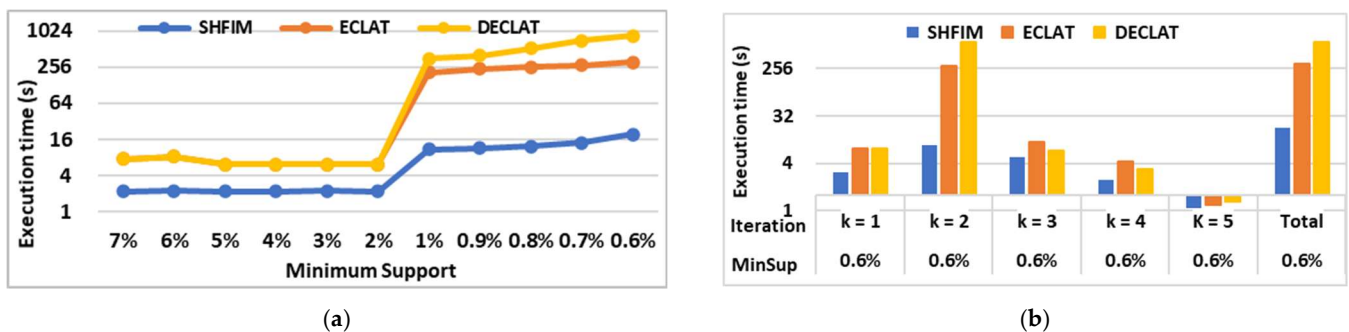


Figure 9. Execution time of SHFIM, Eclat, dEclat in T1014D100k dataset. (a) SHFIM, dEclat, and Eclat at different min_sup (7–0.6%), (b) SHFIM, dEclat, and Eclat by iteration at min_sup = 0.6%.

5.2.1. T1014D100k Dataset

In Figure 9b, a comparison between SHFIM, Eclat, and dEclat on iteration level at min_sup (0.6%). It is worth noting that for any min_sup, each algorithm should complete several iterations. SHFIM outperforms Eclat and dEclat in all iterations at min_sup (0.6%), particularly at the second iteration. Eclat and dEclat finish in about 287 s, and 821 s, respectively, whereas SHFIM takes only 19 s.

5.2.2. Retail Dataset

Figure 10a shows a comparison between the vertical layout algorithms for different min_sup (30–0.2%). We noticed that dEclat and Eclat have recorded the same results at higher supports (30% to 3%) since it is one iteration, and they accomplish the same task “extract the first frequent itemset in vertical layout”. On contrary, SHFIM executes almost twice faster than dEclat and Eclat in higher supports (30% to 3%) that have only one iteration because SHFIM uses only a flatMap in the first iteration, which requires timeless than Eclat and dEclat time required to transform into the vertical layout. Notably, Eclat outperforms dEclat in most iterations due to the sparsity of the Retail dataset and Eclat ability to compete in the sparse datasets.

A comparison between SHFIM, Eclat, and dEclat on iteration level at min_sup (1%) is shown in Figure 10b. SHFIM outperforms Eclat and dEclat in terms of execution time in all iterations except the last iteration. At the last iteration, Eclat outperforms SHFIM and dEclat due to the sparsity of the Retail dataset. On the other hand, dEclat spent a long time in the second and third iterations. As a result, Eclat and dEclat consume about 26 and 318 s, respectively, whereas SHFIM takes only 12 s.

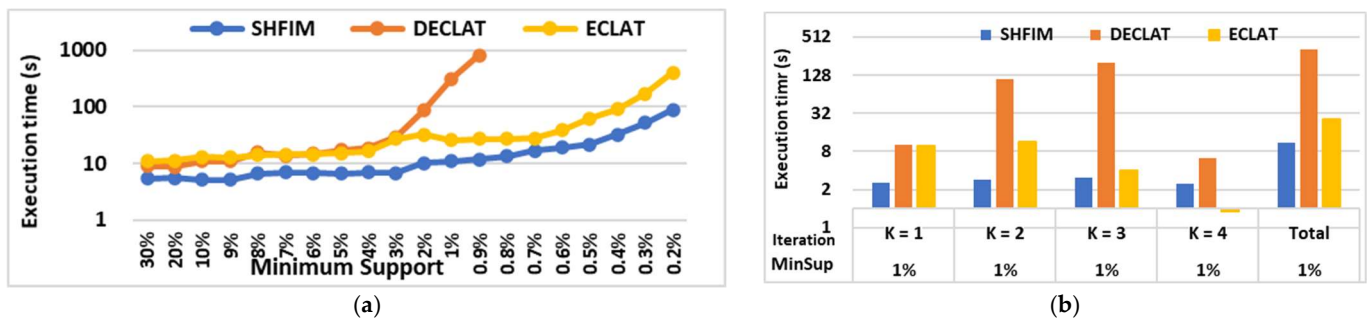


Figure 10. Execution time of SHFIM, Eclat, dEclat in Retail Dataset. (a) SHFIM, dEclat, and Eclat at different min_sup (30–0.2%), (b) SHFIM, dEclat, and Eclat by iteration at min_sup = 1%.

5.2.3. Mushroom Dataset

Figure 11a compares FIM vertical layout algorithms with SHFIM for various min_sup (90–30%). Due to the density of the dataset, Eclat has the slowest execution time in all min_sup. At min_sup (90%), SHFIM executes in 4 s, while dEclat and Eclat execute in 5 s, 6 s, respectively. At min_sup (90–60%), the three algorithms almost show the same performance. However, at min_sup (50–30%), Eclat starts to suffer and takes much time to execute, while SHFIM and dEclat are still competing.

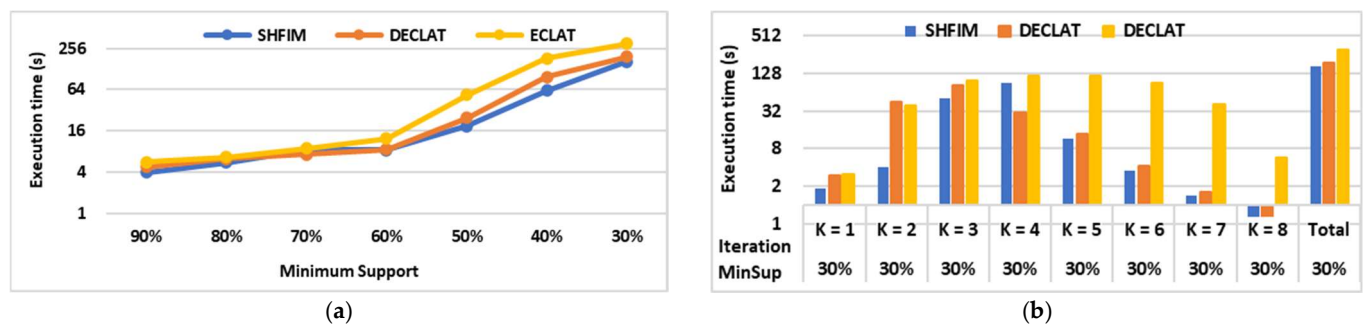


Figure 11. Execution time of SHFIM, Eclat, dEclat in Mushroom Dataset. (a) SHFIM, dEclat, and Eclat at different min_sup (90–30%), (b) SHFIM, dEclat, and Eclat by iteration at min_sup = 30%.

In Figure 11b, a comparison between SHFIM, Eclat, and dEclat on iteration level at min_sup (30%). In all iterations, SHFIM outperformed dEclat and Eclat in terms of execution time, except for the 4th iteration. At the 4th iteration, SHFIM took longer than dEclat. Because of the density of the Mushroom dataset, the driver must create candidates in a long time. Finally, SHFIM has a total runtime of 164 s, while Eclat and dEclat have total runtimes of 303 and 194 s, respectively.

5.2.4. Chess Dataset

Figure 12a shows a comparison between SHFIM, Eclat, and dEclat for different min_sup (90%–85%). SHFIM executes faster than Eclat and slower than dEclat in different min_sup. At min_sup (90%), SHFIM has recorded 41 s while Eclat and dEclat have recorded 98 and 31 s, respectively. Notably, due to the high dataset density of chess dataset, Eclat has recorded the slowest performance.

In Figure 12b, a comparison between SHFIM, Eclat, and dEclat on iteration level at min_sup (85%). SHFIM outperformed Eclat in all iterations, whereas took nearly as long as dEclat in all iterations except the third and fourth, because of the time required by the driver to produce candidates and save the results to HDFS causes SHFIM to take longer than usual. SHFIM takes only 61 s to complete, but Eclat and dEclat take roughly 375 and 55 s, respectively.

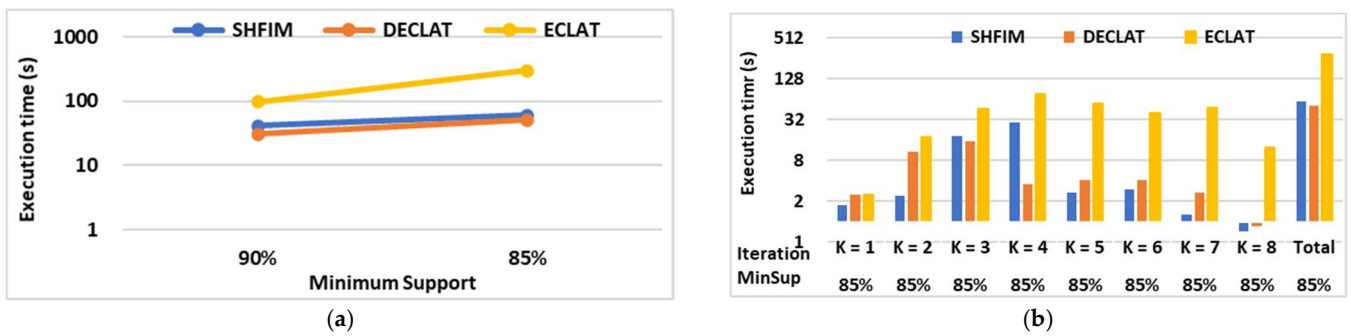


Figure 12. Execution time of SHFIM, Eclat, and dEclat in chess dataset. (a) SHFIM, dEclat, and Eclat at different min_sup (90–85%), (b) SHFIM, dEclat, and Eclat by iteration at min_sup = 85%.

6. Discussion

Table 5 shows the average time (in seconds) of the FIM algorithm on the four datasets. The results conclude that the SHFIM algorithm has recorded 7, 5, and 39 s in T1014D100k, Retail, and Mushroom datasets respectively, which is the best performance among the others. While in chess dataset, SHFIM has recorded 51 s, which is an intermediate performance between Eclat and dEclat. We discovered that the SHFIM is better suited to datasets containing thousands of variable-length transactions because, at high and low min sup, the SHFIM can adapt to datasets containing thousands, if not millions, of variable-length transactions without a memory leak or spilling due to the enhancements made. Furthermore, the SHFIM performs admirably in dense and sparse datasets with low and high min_sup, and it is a scalable approach that can mine large amounts of data. SHFIM demonstrated considerably higher execution time performance with datasets containing a high proportion of variable-length transactions, which is the big data nature.

Table 5. Average time (in seconds) of the FIM algorithm on the four datasets.

Dataset/Algorithm	Min_Sup	SHFIM	ECLAT	DECLAT
Mushroom	90–30%	39	81	49
Chess	90–85%	51	236	43
T1014D100k	7–0.6%	7	121	260
Retail	30–0.9%	5	18	105

Table 6 summarizes the frequent itemsets for various min_sup in both dense and sparse datasets. The summary includes the number of transactions associated with each dataset sample. Min_sup is calculated by multiplying the number of transactions by the predefined min_sup% to determine if an itemset is frequent or not. In addition, it contains the number of frequent itemsets for each dataset based on min_sup. The tidset size per itemset is the number of transactions’ ids in each itemset resulting from intersections between itemsets. Finally, diffset size per itemset is the number of transactions’ ids that are located in each itemset due to itemset differences. For example, mushroom is an 8124-transaction dense dataset that contains 2735 frequent itemsets whose supports are larger than min_sup (2437). Each frequent itemset must have a tidset count ranging from 2437 to 8124. On the other hand, diffset size is at least two or three times less than tidset size since the difference reduces data significantly. As a result, diffset is memory-conscious, compact, and quick to process.

Table 6. Datasets summary.

Dataset	T1014D100k		Retail		Mushroom		Chess	
Sparsity	Sparse		Sparse		Dense		Dense	
No. of Transactions	100,000		87,988		8124		3196	
Min_sup (%)	0.6%	0.7%	0.4%	0.5%	30%	40%	85%	90%
Min_sup	600	700	352	440	2437	3250	2717	2876
No. of Frequent itemsets	772	603	831	580	2735	565	2669	622
Tidset size per frequent itemset	600– 100,000	700– 100,000	352– 87,988	440– 87,988	2437– 8124	3250– 8124	2717– 3196	2876– 3196
Diffset size per frequent itemset	At least 2 or 3 times less than tidset size							

7. Time Complexity

In this section, we are going to present the time complexity of the SHFIM. The complexity of SHFIM can be influenced by factors such as min_sup threshold, the average number of items per transaction, the number of transactions, and the total number of dataset items. Suppose we have a Dataset D contains n number of transactions, k number of items, and m as the number of items in the biggest transaction. In phase 1, we need to traverse each item in every transaction to discover the first frequent itemset. This process takes in the worst-case $O(n \times m)$. In Phase 2, BF starts to access each transaction item and prune the items that are not present in the BF and creates pairs from the items that are present after pruning, which takes in the worst-case $O(n \times m)$. Furthermore, BF keeps tracking of each transaction ID for each pair created which costs $O(1)$ then finally reduce the results to create the vertical layout which costs $O(1)$. The time complexity of phase 2 can be measured by $O(n \times m + 1 + 1)$ equal to $O(n \times m)$. Assume F_k is the vertical frequent dataset from phase 2 that contains N number of itemsets. Let the number of candidates' itemsets of iteration k is C_k . Phase 3 in SHFIM is an iterative process; hence, we find the time complexity for kth iteration/pass. Phase 3 begins with $k = 3$ till the end of iterations in which SHFIM starts to generate C_k candidates from F_{k-1} , which takes $O\left(\binom{N}{k}\right)$ time. Afterward, all the candidates should be mapped with their originated itemsets which are present F_{k-1} which takes $O(N)$ time. We further visit each candidate after mapping to calculate the difference between each mapped itemsets which requires $O(C_k)$, and it requires $O(1)$ to calculate the differences between its itemsets. We assume that the shared or broadcasted candidate list data requires $O(C_k)$ time. Therefore, phase 3 takes $O\left(\binom{N}{k}\right) \times N \times C_k \times C_k$ time to be completed. Since phase 3 is an iterative process based on k, the complexity will be $O\left(\sum_{k=3}^{kn} \left(\binom{N}{k}\right) \times N \times C_k^2\right)$, where $k \geq 3$. Finally, the total complexity of SHFIM proposed algorithm is the summation of the complexity of phase 1, phase 2, and phase 3.

8. Conclusions

FIM is the most common technique used in discovering frequent patterns from transactional datasets. Frequent patterns have a wide effect in many applications including classifications, market basket analysis, mobile computing, web mining, etc. Apriori is computing intensive algorithm; therefore, lots of resources (Memory and processing) are required. Moreover, Apriori uses horizontal data representation and has some challenges such as multiple dataset scans and candidate generating in each iteration, which makes Apriori suffer from big data. Vertical data representation is smaller than horizontal representation in size and carries information through tidsets about each itemset support. Eclat uses vertical data representation (tidset) and achieved observed performance in sparse

datasets, but in dense datasets, it suffers when intermediate results of tidsets become too large for memory.

In this paper, we proposed SHFIM (Spark-based Hybrid Frequent Itemset Mining) a novel algorithm that utilizes both the horizontal and vertical layouts to solve the drawbacks in both Apriori and Eclat. SHFIM is a three phases algorithm, which works perfectly in a distributed environment. Phases one and two use the horizontal layout to extract the first and second frequent itemset. Phase three is an iterative process to extract k frequent itemset in k iterations. This phase uses mainly diffset to enhance execution time and memory consumption because it shrinks itemsets into smaller sets that will be mined in less time and consume less space. The support of an itemset is calculated by exploiting the vertical layout in every worker node. As the vertical layout size is smaller than the horizontal layout, therefore it requires less memory and less execution time. We developed SHFIM on Spark framework due to its ability to deal with the iterative problem better than Hadoop MapReduce. Spark is 100 times quicker than Hadoop in data processing and has lots of features such as in-memory processing, RDD data structure, broadcasting variables, partitioning of data, and applied Spark best practices to reduce data shuffling between nodes. These features make the Spark the best choice for us that help SHFIM to deal with big data efficiently and increase its execution time performance. Extensive experiments have been conducted between SHFIM, Eclat, and dEclat over Spark clusters for dense and sparse datasets. The Experimental results proved that SHFIM can compete well in both dense and sparse datasets and shows noticeably better performance in either lower or higher min_sup in terms of execution time than others in datasets that have lots of variable-length transactions which is the nature of big data. In the future work, we are planning to enhance the SHFIM be more efficient. The results proved that the use of tidset, diffset, and Bloom Filter accelerate the speed of FIM in big datasets. We plan to choose between tidset and diffset on the itemset itself rather than the whole dataset instead of applying the diffset and continue using diffset from the third iteration in the whole dataset.

Author Contributions: Conceptualization, M.S.F.; Data curation, M.S.F.; Formal analysis, M.R.A.-B.; Investigation, M.R.A.-B.; Methodology, M.R.A.-B.; Project administration, M.S.F.; Software, M.R.A.-B.; Supervision, M.S.F. and N.A.O.; Validation, N.A.O.; Visualization, M.R.A.-B.; Writing—original draft, M.R.A.-B.; Writing—review & editing, M.S.F. and N.A.O. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Jiawei, H.; Kamber, M. Data Mining Concepts and Techniques, 550. Available online: https://www.researchgate.net/publication/235902451_Data_Mining_Concept_and_Techniques (accessed on 13 December 2021).
2. Apiletti, D.; Baralis, E.; Cerquitelli, T.; Garza, P.; Pulvirenti, F.; Venturini, L. Frequent Itemsets Mining for Big Data: A Comparative Analysis. *Big Data Res.* **2017**, *9*, 67–83. [CrossRef]
3. Big Data Tutorial | All You Need to Know about Big Data | Edureka. Available online: <https://www.edureka.co/blog/big-data-tutorial> (accessed on 4 January 2022).
4. Landset, S.; Khoshgoftaar, T.M.; Richter, A.N.; Hasanin, T. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. *J. Big Data* **2015**, *2*, 24. [CrossRef]
5. Tai, D.D.; Huynh, V.N. K-PbC: An Improved Cluster Center Initialization for Categorical Data Clustering. *Applied Intelligence* **2020**, *50*, 2610–2632. [CrossRef]
6. Naulaerts, S.; Meysman, P.; Bittremieux, W.; Vu, T.N.; Berghe, W.V.; Goethals, B.; Laukens, K. A primer to frequent itemset mining for bioinformatics. *Brief. Bioinform.* **2015**, *16*, 216–231. [CrossRef]
7. Ilayaraja, M.; Meyyappan, T. Efficient Data Mining Method to Predict the Risk of Heart Diseases through Frequent Itemsets. *Procedia Comput. Sci.* **2015**, *70*, 586–592. [CrossRef]

8. Loshin, D. Knowledge Discovery and Data Mining for Predictive Analytics. *Bus. Intell.* **2013**, 271–286. [CrossRef]
9. Luna, J.M.; Fournier-Viger, P.; Ventura, S. Frequent itemset mining: A 25 years review. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **2019**, 9, e1329. [CrossRef]
10. Apiletti, D.; Baralis, E.; Cerquitelli, T.; Chiusano, S.; Grimaudo, L. SeaRum: A Cloud-Based Service for Association Rule Mining. In Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, Washington, DC, USA, 16–18 July 2013; pp. 1283–1290.
11. Gao, C.; Tung, A.K.H.; Xu, X.; Pan, F.; Yang, J. FARMER: Finding interesting rule groups in microarray datasets. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France, 13–18 June 2004; p. 143. [CrossRef]
12. Tania, C.; Di Corso, E. Characterizing Thermal Energy Consumption through Exploratory Data Mining Algorithms. 2016. Available online: <https://iris.polito.it/handle/11583/2639284> (accessed on 9 January 2022).
13. Antonie, M.; Zaiane, O.R.; Coman, A. Application of Data Mining Techniques for Medical Image Classification. In Proceedings of the Second International Conference on Multimedia Data Mining, San Francisco, CA, USA, 26 August 2001.
14. Rakesh, A.; Srikant, R. Fast Algorithms for Mining Association Rules. Available online: <https://dl.acm.org/doi/10.5555/645920.672836> (accessed on 9 January 2022).
15. Apriori Algorithm—GeeksforGeeks. Available online: <https://www.geeksforgeeks.org/apriori-algorithm/> (accessed on 4 January 2022).
16. Zaki, M. Scalable algorithms for association mining. *IEEE Trans. Knowl. Data Eng.* **2000**, 12, 372–390. [CrossRef]
17. ML | ECLAT Algorithm—GeeksforGeeks. Available online: <https://www.geeksforgeeks.org/ml-eclat-algorithm/> (accessed on 4 January 2022).
18. Zaki, M.J.; Gouda, K. Fast vertical mining using diffsets. In Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining—KDD '03, Washington, DC, USA, 24–27 August 2003; pp. 326–335.
19. Rao, T.R.; Mitra, P.; Bhatt, R.; Goswami, A. The big data system, components, tools, and technologies: A survey. *Knowl. Inf. Syst.* **2019**, 60, 1165–1245. [CrossRef]
20. Big Data Analysis Using Apache Hadoop. Available online: https://www.researchgate.net/publication/261309523_Big_data_analysis_using_Apache_Hadoop (accessed on 29 November 2020).
21. Apache Hadoop. Available online: <http://hadoop.apache.org/> (accessed on 28 November 2020).
22. Weets, J.-F.; Kakhani, M.K.; Kumar, A. Limitations and challenges of HDFS and MapReduce. In Proceedings of the 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), NW Washington, DC, USA, 8–15 October 2015; IEEE: Manhattan, NY, USA, 2015; pp. 545–549.
23. Frequent Pattern Mining—RDD-Based API—Spark 2.2.0 Documentation. Available online: <https://spark.apache.org/docs/2.2.0/ml-lib-frequent-pattern-mining.html> (accessed on 23 December 2020).
24. Salloum, S.; Dautov, R.; Chen, X.; Peng, P.X.; Huang, J.Z. Big data analytics on Apache Spark. *Int. J. Data Sci. Anal.* **2016**, 1, 145–164. [CrossRef]
25. Frequent Pattern Mining—Spark 3.0.1 Documentation. Available online: <https://spark.apache.org/docs/latest/ml-frequent-pattern-mining.html> (accessed on 22 December 2020).
26. Cai, B.Z.; Zhu, X.; Zheng, Y.; Liu, D.; Xu, L. *A Caching-Based Parallel FP-Growth in Apache Spark*; Springer International Publishing: Cham, Switzerland, 2018. [CrossRef]
27. BloomFilter (Spark 2.1.0 JavaDoc). Available online: <https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/util/sketch/BloomFilter.html> (accessed on 27 May 2021).
28. Raj, S.; Ramesh, D.; Sreenu, M.; Sethi, K.K. EAFIM: Efficient apriori-based frequent itemset mining algorithm on Spark for big transactional data. *Knowl. Inf. Syst.* **2020**, 62, 3565–3583. [CrossRef]
29. Rathee, S.; Kashyap, A. Adaptive-Miner: An efficient distributed association rule mining algorithm on Spark. *J. Big Data* **2018**, 5, 6. [CrossRef]
30. Sethi, K.K.; Ramesh, D. HFIM: A Spark-based hybrid frequent itemset mining algorithm for big data processing. *J. Supercomput.* **2017**, 73, 3652–3668. [CrossRef]
31. Zhang, F.; Liu, M.; Gui, F.; Shen, W.; Shami, A.; Ma, Y. A distributed frequent itemset mining algorithm using Spark for Big Data analytics. *Clust. Comput.* **2015**, 18, 1493–1501. [CrossRef]
32. Li, H.; Wang, Y.; Zhang, D.; Zhang, M.; Chang, E.Y. RecSys '08. In Proceedings of the 2008 ACM Conference on Recommender Systems, Lausanne, Switzerland, 23–25 October 2008; pp. 107–114.
33. Rathee, S.; Kaul, M.; Kashyap, A. R-Apriori. In *Proceedings of the 8th Workshop on Ph.D. Workshop in Information and Knowledge Management*; ACM Press: New York, NY, USA, 2015; pp. 27–34.
34. Qiu, H.; Gu, R.; Yuan, C.; Huang, Y. YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark. In Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, 19–23 May 2014; pp. 1664–1671.
35. Huang, P.-Y.; Cheng, W.-S.; Chen, J.-C.; Chung, W.-Y.; Chen, Y.-L.; Lin, K.W. A Distributed Method for Fast Mining Frequent Patterns From Big Data. *IEEE Access* **2021**, 9, 135144–135159. [CrossRef]
36. Singh, P.; Singh, S.; Mishra, P.K.; Garg, R. RDD-Eclat: Approaches to Parallelize Eclat Algorithm on Spark RDD Framework. In *Lecture Notes on Data Engineering and Communications Technologies*; Springer: Berlin/Heidelberg, Germany, 2020; Volume 44, pp. 755–768. [CrossRef]

37. Leung, C.K.; Zhang, H.; Souza, J.; Lee, W. *Scalable Vertical Mining for Big Data Analytics of Frequent Itemsets. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer International Publishing: Cham, Switzerland, 2018; Volume 11029.
38. Liu, J.; Wu, Y.; Zhou, Q.; Fung, B.C.M.; Chen, F.; Yu, B. Parallel Eclat for Opportunistic Mining of Frequent Itemsets. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; International Conference on Database and Expert Systems Applications: Valencia, Spain, 2015; Volume 9261, pp. 401–415.
39. Moens, S.; Aksehirli, E.; Goethals, B. Frequent Itemset Mining for Big Data. *2013 IEEE Int. Conf. Big Data* **2013**, 111–118. [[CrossRef](#)]
40. Ragaventhiran, J.; Kavithadevi, M. Map-optimize-reduce: CAN tree assisted FP-growth algorithm for clusters based FP mining on Hadoop. *Futur. Gener. Comput. Syst.* **2020**, *103*, 111–122. [[CrossRef](#)]
41. Shi, X.; Chen, S.; Yang, H. DFPS: Distributed FP-growth algorithm based on Spark. In Proceedings of the 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Chongqing, China, 25–26 March 2017; IEEE: Manhattan, NY, USA, 2017; pp. 1725–1731.
42. Han, J.; Pei, J.; Yin, Y. Mining frequent patterns without candidate generation. *ACM SIGMOD Rec.* **2000**, *29*, 1–12. [[CrossRef](#)]
43. Frequent Pattern Mining. *Freq. Pattern Min.* **2014**, *9783319078212*, 1–471. [[CrossRef](#)]
44. Frequent Itemset Mining Dataset Repository. Available online: <http://fimi.uantwerpen.be/data/> (accessed on 12 December 2020).