SSRG International Journal of Computer Science and Engineering
ISSN: 2348–8387 / https://doi.org/10.14445/23488387/IJCSE-V12I9P103

Volume 12 Issue 9, 12-23, September 2025
© 2025 Seventh Sense Research Group®

Original Article

An Efficient Spark-Based Parallel FP-Growth For Big Data Mining With Key-Value Pair Model

Baokui Liao[1,2*], Mohd Nurul Hafiz Ibrahim[3], Mustafa Muwafak Alobaedy[4], S. B. Goyal[5]

[1]*City Graduate School, City University Malaysia, Petaling Jaya, 46100, Kuala Lumpur, Malaysia.*
[2]*Guizhou Light Industry Polytechnic University, Guizhou, China.*
[3,4]*Faculty of Information Technology, City University Malaysia, Kuala Lumpur, Malaysia.*
[5]*Chitkara University Institute of Engineering & Technology, Chitkara University, Punjab, India.*

**Corresponding Author : liaobaokui1988@gmail.com*

Received: 21 July 2025 Revised: 25 August 2025 Accepted: 08 September 2025 Published: 30 September 2025

Abstract - *In the era of big data, the exponential growth of information has made extracting valuable insights from massive datasets an urgent challenge. Association rule mining, particularly the FP-Growth, plays a crucial role in discovering high-frequency patterns. Traditional FP-Growth faces significant challenges when processing large-scale data, including memory overflow and computational inefficiency. Existing improvements to FP-Growth have achieved some success in parallelization, with the PFP being a notable example. This paper proposes an algorithmic parallelization scheme based on Spark, enhancing mining efficiency by splitting FP trees using key-value pairs and optimizing database scanning processes. Unlike traditional methods relying on global FP trees, this algorithm leverages Spark's distributed in-memory computing model to eliminate time-consuming FP tree traversal operations and reduce inter-node communication. Tests demonstrate that the KVBFP exhibits high stability, achieving approximately 55% lower communication overhead compared to PFP. It also reduces the mean variance of cluster CPU and memory utilization by 87.2% and 92.4%, respectively, while boosting overall mining efficiency by 44.7%.*

Keywords - *FP-Growth, Spark, Parallel Mining, Big Data, Data Mining.*

1. Introduction

Data mining is a key technique for analyzing big data and aims to extract meaningful patterns, associations, and trends from large-scale datasets. It uses algorithms in the fields of machine learning, statistics, and artificial intelligence to discover hidden relationships in data that may not be detected by traditional analytics. The main goal of data mining is to transform raw data into useful knowledge to aid in decision-making and prediction [1]. FP-Growth, as an important algorithm in the association rule mining branch of data mining, stores Frequent Item (FI) sets by constructing a global FP-tree, and then finds frequent item sets by traversing the tree structure [2].

The rise of Big Data stems from the proliferation of digital technologies such as the Internet of Things (IoT), social media platforms, and mobile devices that are generating unprecedented data [3]. Millions of social media posts, financial transactions, and sensor data from smart devices are recorded every minute. Extracting valuable information from this massive amount of data can give organizations a competitive advantage, speed up the decision-making process, and lead to innovations in a variety of areas such as healthcare, finance, and urban planning [4]. Among the many big data processing frameworks, Apache Spark is one of the most widely used. It provides a distributed computing engine featuring in-memory processing and Resilient Distributed Datasets (RDDs), which significantly speed up computation and reduce I/O overhead [5]. Its streamlined application program interface and dynamic task scheduling make it easier for developers to build scalable data pipelines without having to deal with the complexities of underlying distributed programming [6]. Big data and data mining technologies are two hot research topics at present, and there have been many research results in their intersection. This study focuses on the parallelization of FP-Growth in data mining technology on the big data platform Spark, and proposes a more efficient parallelization strategy based on existing research results.

FP-Growth was designed to run on a single machine and was not intended for distributed or parallel execution [7]. As the size of the dataset increases, the global FP tree expands accordingly, which can easily lead to memory overflow when its size exceeds the available system memory [8]. In addition, traversing the global FP-tree requires a huge amount of computation, which consumes a lot of CPU resources, and when the FP-tree becomes very large, the traversal operation will lead to CPU overload [9]. The strong reliance on the global FP-Tree creates a bottleneck in parallel frameworks such as Spark [10], as this structure cannot be fully replicated across all executors, and traversing the global FP-Tree in a distributed environment places stringent requirements on memory synchronization and coordination.

To address the limitations of the FP-Growth, existing research has introduced various parallelization strategies. These approaches address the load balancing problem to some extent but have inherent flaws, such as load balancing, I/O bottlenecks, excessive dependence on hardware, and cumbersome algorithm complexity. This study proposes an enhanced version of the FP-Growth, which replaces the global FP-Tree structure with a key-value pair for data mining. This study also proposes a database scanning strategy. This approach preprocesses and reorganizes the source database through two Spark tasks to efficiently filter and sort high-frequency items. Through designed experiments, the algorithm demonstrates high stability, reduces hardware consumption compared to existing parallel FP-Growth implementations, and enhances cluster communication efficiency, load balancing, and mining efficiency.

2. Related Work

A systematic review of FP-Growth parallelization research over the past five years reveals current research gaps. This paper introduces six of the most highly cited FP-Growth parallelization approaches and provides a detailed comparison of their respective shortcomings. Senthilkumar et al. proposed an efficient MapReduce-based FP-Growth in 2020 to address memory consumption and communication bottlenecks in traditional FP-Growth under big data scenarios [11]. This approach employs four key optimizations: dictionary encoding for item set compression, improved hash partitioning to reduce network load, compression strategies to enhance data transfer efficiency, and a combiner to optimize the reduction phase load. Experiments demonstrate that this algorithm exhibits excellent execution efficiency and scalability in Hadoop cluster environments, making it suitable for distributed frequent item set mining tasks. In 2021, Zakria Mahrousa et al. proposed an improved FP-Growth that combines MapReduce with directed graph structures to enhance frequent item set mining efficiency on large-scale datasets [12].

This algorithm distributes raw data across multiple nodes, compresses transaction databases using graph structures, and constructs FP-Trees in parallel, thereby effectively reducing memory consumption and computation time. Research results show that PGFP-Growth outperforms standard MapReduce implementations when processing high-dimensional, high-density transaction databases and offers significant advantages in memory management. Amr Essam et al. proposed an enhanced balanced parallel frequent pattern mining algorithm in 2021 [13] to optimize load balancing and efficiency for Parallel FP-Growth on Spark during big data processing. This method introduces a load-balanced grouping strategy to achieve even task distribution across cluster nodes. Simultaneously, it refines the conditional pattern base to eliminate low-frequency items, thereby reducing memory consumption and the overhead of constructing local FP-Trees. Test results demonstrate that this algorithm achieves a 21.56% to 39.72% improvement in runtime compared to FP-Growth, significantly enhancing execution efficiency and scalability. Priyanka Gupta et al. proposed parallel Apriori and FP-Growth in 2021 [14], deployed on the Apache Spark platform to boost frequent item set mining efficiency in big data environments. This approach leverages Spark's in-memory computing capabilities, utilizing RDD-supported data partitioning and parallel processing to run both Apriori and FP-Growth concurrently across multiple datasets, comparing and enhancing their performance. Experimental results demonstrate faster execution speeds and good scalability when adjusting support thresholds. Youssef Fakir et al. proposed a parallel FP-Growth algorithm based on Apache Spark in 2024 for large-scale medical data mining in diabetes prediction tasks [15].

This algorithm employs a horizontal data partitioning strategy combined with Spark's distributed computing framework, significantly accelerating frequent item set mining while maintaining accuracy. Experimental validation on medical datasets demonstrates superiority over traditional methods in both prediction accuracy and operational efficiency. Shubhangi Chaturvedi et al. proposed a vector-distance-based FP-Growth in 2023 [16]. This approach constructs local FP-Trees based on the prefix distance from each frequent item to the root node as the load-balancing grouping criterion. These local FP-Trees are then distributed to computational nodes for parallel mining. Test results demonstrate that the Parallel FP-growth (PFP) achieves significant improvements in load balancing, maintains stable performance when handling massive datasets, and exhibits highly efficient mining capabilities. While all six parallelization approaches can accomplish parallel mining tasks, each possesses inherent limitations. Table 1 summarizes the shortcomings of each algorithm.

Given the limitations of the algorithms listed in Table 1, the current parallel FP-Growth domain still lacks an efficient parallelization scheme that simultaneously achieves load balancing, low complexity, and minimal cluster communication overhead. Therefore, this paper proposes an improved parallel algorithm based on key-value pairs, termed Key-based Vertical Based FP-Growth (KVBFP). By replacing key-value pairs across all stages of FP-Growth, this approach maximizes compatibility with the Spark platform to address existing algorithmic limitations. Among the reviewed algorithms, PFP introduced the concept of vector distance for the first time, significantly enhancing load balancing. This study selects PFP as the reference point for further investigation.

Table 1. Related research limitations summary

Author	Limitation
Priyanka Gupta et al.（2021）	Parallel implementation of classical methods fails to address FP-tree expansion, Apriori candidate set explosion issues, and has poor pruning capability.
Amr Essam et al.（2021）	Focused on load balancing optimization, but local FP-Tree redundancy remains severe, lacking tree pruning and compression strategies.
Youssef Fakir et al.（2024）	The algorithm is based on Spark but does not incorporate path optimization or compressed structures; frequent item set generation is relatively broad, limiting its applicability.
Senthilkumar et al.（2020）	Optimized networking and storage, but slow due to disk I/O; lacks FP-Tree hierarchical optimization.
Zakria Mahrousa et al.（2021）	Path compression using directed graph structures lacks support for adaptability and pruning/merging mechanisms; redundant paths may still be generated for high-dimensional data.
Shubhangi Chaturvedi et al.（2023）	Vector distance calculations are computationally complex and can easily lead to excessive communication frequency between clusters.

3. Algorithm Proposal

3.1. Algorithm Process

The KVBFP is implemented in two main stages. The first phase transforms the raw transaction dataset into a vertical data structure (Swap the rows and columns of the source dataset) that compresses and filters transactions based on item frequency. The second phase performs frequent item set mining by applying a series of key-value transformations using Spark's RDD operations. The entire process flow chart of KVBFP is shown in Figure 1.

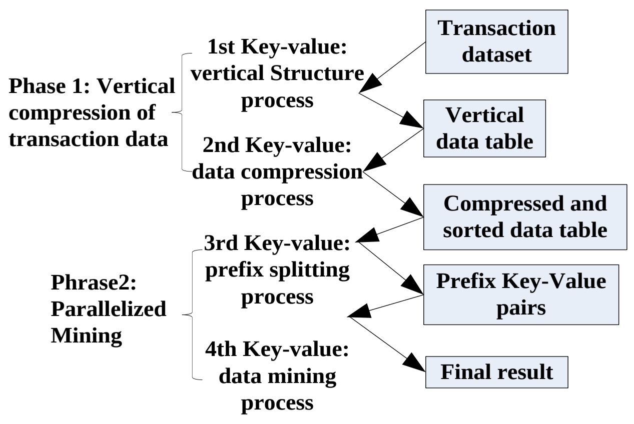

Phase 1: Vertical compression of transaction data
1st Key-value: vertical Structure process
2nd Key-value: data compression process
Transaction dataset
Vertical data table
Compressed and sorted data table
Phrase2: Parallelized Mining
3rd Key-value: prefix splitting process
4th Key-value: data mining process
Prefix Key-Value pairs
Final result

Fig. 1 Entire procedure of KVBFP

3.1.1. Phase 1: Vertical Compression of Transaction Data

If the original transaction database consists of a set of records as shown in Table 2, where each transaction contains a set of item sets, here the support threshold is preset to 2 to mine the database for frequent item sets. Spark's Distributed Data Processing framework converts and compresses the dataset into a vertical data format in two steps. In these Tables and Figures, 'Sup' is the support for each element, 'Num' is the transaction number, 'E' represents the items in each transaction, and 'N' represents the specific label value corresponding to each transaction.

Table 2. Original transaction database

Num	Trans
N1	E1, E2, E5
N2	E2, E4, E6
N3	E2, E3
N4	E1, E2, E4
N5	E1, E3, E7
N6	E2, E3
N7	E1, E3, E8
N8	E1, E2, E3, E5
N9	E1, E2, E3

Step 1: Load all transactions into the RDD and map each item to the list of transaction IDs that appear in it. Calculate the support count for each item using flatMap() and reduceByKey(). Items that do not meet the support threshold are filtered out. The remaining frequent items are sorted in descending order of support to form a vertical item sets table. The results of Step 1 are shown in Table 3.

Table 3. Vertical data table

Element	E2	E1	E3	E4	E5
Sup	7	6	6	2	2
Num	N1, N2, N3, N4, N6, N8, N9	N1, N4, N5, N7, N8, N9	N3, N5, N6, N7, N8, N9	N2, N4	N1, N8

The data flow of Job1 from Table 2 to Table 3 is shown in Figure 2.

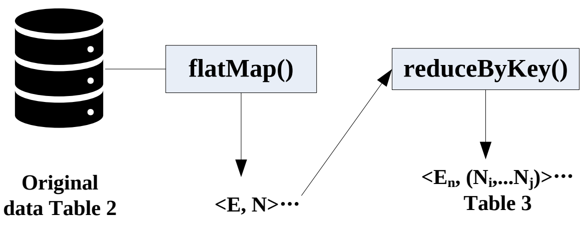


Fig. 2 Data flow of vertical compression process

Step 2: Each transaction is then reconstructed by removing the infrequent items using the vertical table and sorting the remaining items in descending order according to the support threshold.This results in a compressed database containing only frequent items, which facilitates the subsequent mining phase. The results of Step 2 are shown in Table 4.

Table 4. Data table after compression and sorting

Num	Trans
N1	E2, E1, E5
N2	E2, E4
N3	E2, E3
N4	E2, E1, E4
N5	E1, E3
N6	E2, E3
N7	E1, E3
N8	E2, E1, E3, E5
N9	E2, E1, E3

The data flow of Job2 from Table 3 to Table 4 is shown in Figure 3.

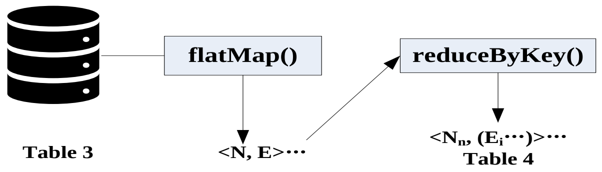


Fig. 3 Data flow of sorting process

As can be seen from Table 4, relative to Table 2, E6, E7, and E8, which do not satisfy the support thresholds, have been deleted, and the items in each transaction are listed in order of decreasing support.

```
Phase 1 Pseudocode
Input:
   D ← original table: mapping Num → [$E_1$, $E_2$, ..., $E_n$]
   σ ← minimum support threshold
Phase 1: Build Item-to-Column Map and Count Frequencies
1. For each (Num, [$E_1$, ..., $E_n$]) in D:
       For each item $E_i$:
           Emit ($E_i$, Num)
2. Group by $E_i$ to form inverted index: $E_i$ → [$Num_1$, $Num_2$, ..., $Num_k$]
3. For each $E_i$:
       Count support = length of [Num list]
4. Filter: retain only $E_i$ where support $\geq$ σ
5. Store frequency table: freq($E_i$) = support count
Phase 2: Reconstruct Column-to-Items Mapping
6. For each $E_i$ in the frequency table:
       For each Num in $E_i$'s list:
           Emit (Num, $E_i$)
7. Group by Num to get: Num → [$E_1$, ..., $E_n$]
8. For each Num:
       Remove items not in the frequency table
       Sort items in descending order by freq($E_i$)
9. Return sorted lists per Num
```

3.1.2. Phase 2: Mining Frequent Item Sets through Key-Value Transformations

Using the preprocessed data in Table 4, each transaction is decomposed into multiple key-value pairs, with the key being the item in the transaction and the value being the item preceding that item. This decomposition is accomplished using Spark's flatMap function. Each transaction is converted into multiple key-value pairs of the form <E, prefix>. For example, transaction [E2, E1, E5] is decomposed into <E2, null>, <E1, E2>, and <E5, (E2 E1)>, and the splitting results are shown in Table 5.

```
Phase 2 Pseudocode of the splitting process
Input:
   T ← filtered and sorted transaction table: Num → [$E_1$, $E_2$, ..., $E_n$]
flatMap:
   1.For each (Num, [$E_1$, $E_2$, ..., $E_n$]) in T:
       For i from 0 to length(items) - 1:
           key ← $E_i$
           value ← items[0:i]
           Emit (key, value)
reduceByKey:
2. Group by key $E_i$:
       Collect all value lists as a conditional pattern base of $E_i$
       Emit ($E_i$, [$prefix_1$, $prefix_2$, ..., $prefix_k$])
```

Table 5. Results of Table 4 after splitting by flatMap

Num	Key-value
N1	<E2, NULL>, <E1, E2>, <E5, (E2, E1)>
N2	<E2, NULL>, <E4, E2>
N3	<E2, NULL>, <E3, E2>
N4	<E2, NULL>, <E1, E2>, <E4, (E2, E1)>
N5	<E1, NULL>, <E3, E1>
N6	<E2, NULL>, <E3, E2>
N7	<E1, NULL>, <E3, E1>
N8	<E2, NULL>, <E1, E2>, <E3, (E2, E1)>, <E5, (E2, E1, E3)>
N9	<E2, NULL>, <E1, E2>, <E3, (E2, E1)>

The data flow of Job3 from Table 4 to Table 5 is shown in Figure 4.

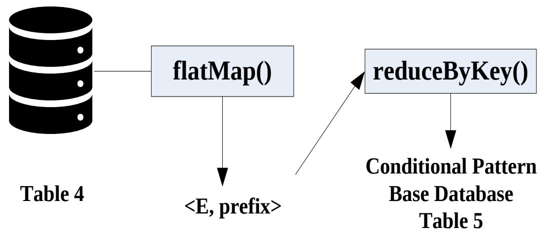


Fig. 4 Data flow splitting process

Then all key-value pairs are grouped using the reduceByKey operation. For each key, a conditional FP-Tree is constructed locally with all its corresponding values, and at the same time, the conditional FP-Tree is mined for frequent item sets in the reduceByKey function to obtain the final result. The key-value pairs after grouping by reduceByKey and the constructed conditional FP-Tree are shown in Table 6.

```
Phase 2 Pseudocode of mining process
Input:
   T ← filtered and sorted transaction table: Num → [$E_1$, $E_2$, ..., $E_n$]
flatMap:
1.For each (Num, [$E_1$, $E_2$, ..., $E_n$]) in T:
      For i from 0 to length(items) - 1:
         key ← $E_i$
         value ← items[0:i]
         Emit (key, value)
reduceByKey:
2. Group by key $E_i$:
      Collect all value lists as a conditional pattern base of $E_i$
      Emit ($E_i$, [$prefix_1$, $prefix_2$, ..., $prefix_k$])
```

Table 6. Mining results from Table 4

Key	E1	E2	E3	E4	E5
Value	(E2), (E2), (E2), (E2)	-	(E2), (E1), (E2), (E1), (E2, E1), (E2, E1)	(E2), (E2, E1)	(E2, E1), (E2, E1, E3)
FI	{E2, E1:4}	-	{E2, E1, E3:2}, {E2, E3:4}, {E1, E3:4}	{E2, E4:2}	{E2, E5:2}, {E1, E5:2}, {E2, E1, E5:2}

The data flow of Job4 from Table 5 to the final results is shown in Figure 5.

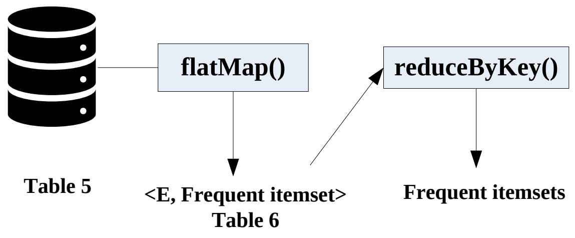


Fig. 5 Data flow of mining process

3.2. Algorithm Complexity Analysis

Complexity analysis requires quantifying each stage of the KVBFP's mining process. From the algorithmic procedure, two key metrics influencing computational complexity emerge: computational complexity and communication complexity. Let the dataset contain D transactions, with an average transaction length of T and an average length of T' after infrequent item filtering. The algorithm's execution flow on Spark is divided into two phases by two Shuffle operations.

3.2.1. Phrase 1 - Step 1: Item Frequency Counting

This step is accomplished using flatMap and reduceByKey. It performs a single linear scan of the dataset with a computational complexity of O(D·T). The reduceByKey operation triggers the first shuffle, whose communication complexity is O(D·T).

3.2.2. Phrase 1 - Step 2: Transaction Preprocessing

This step uses flatMap to filter and sort each transaction. By distributing the item frequency table via broadcast variables, this stage avoids shuffle operations. It involves sorting operations with a computational complexity of O(D·(T + T'logT')).

3.2.3. Phrase 2 : Parallelized Mining

This Phrase first generates prefix paths using flatMap, then groups paths by item using groupByKey. Its computational cost is O(D·T'). The groupByKey operator triggers a second Shuffle, with communication complexity of O(D·T').

The total computational complexity is the sum of all stage computations. Its expression refers to Equation 1.

$$C_{comp} = O(D \cdot T + D \cdot (T + T'\log T') + D \cdot T') \quad (1)$$

The overall algorithmic complexity exhibits a linear positive correlation with the number of transactions D, the average transaction length T, and the number of sorting operations. Meanwhile, the sorting operations show a logarithmic positive correlation with the filtered average transaction length.

The total communication complexity C_comm is primarily determined by the shuffle data volume in Spark, representing the sum of two shuffles. Its expression is given by Equation 2.

$$C_{comm} = O(D \cdot T + D \cdot T') \quad (2)$$

The overall communication complexity of the algorithm exhibits a linear positive correlation with the number of transactions D, the average transaction length T, and the filtered average transaction length T'.

3.2.4. Algorithm Correctness Discussion

The algorithmic process demonstrates that the KVBFP is a key-value pair description of FP-Growth, preserving its original logic. Consequently, the mining results remain consistent with FP-Growth. Table 7 maps each step of KVBFP to its corresponding FP-Growth step and conducts an equivalence analysis for each.

Table 7. Comparison of KVBFP and FP-growth

Core Phase	FP-Growth	KVBFP	Equivalence Analysis
Frequency Statistics	First scan: Traverse the entire dataset and compute the support count for each item	Job1: Parallel computation of support counts for each item using flatMap and reduceByKey	Equivalent results: Both methods counted the frequency of each item's occurrence in the global dataset
Data Preprocess	Second Scan: 1. Remove all infrequent items. 2. Sort frequent items within each transaction by descending global frequency	Job2: Utilize the frequency table from the previous step to filter and sort each transaction in parallel using the flatMap operation	Data equivalence: Upon completion of this phase, both algorithms have produced datasets that are identical in both content and sequence
Construct Data Structures	Constructing the global FP-Tree: Compress the entire preprocessed dataset into an FP-Tree	Job3-flatMap: Use flatMap to split each transaction into multiple <E, prefix> pairs	Logical decomposition: KVBFP skips the step of constructing a global FP-Tree, directly decomposing it into a set of prefix paths indexed by each frequent item
Mining Process	Recursive Mining: 1. Select a frequent item E from the FP-Tree. 2. Extract its conditional pattern base. 3. Construct a conditional FP-Tree based on this pattern base. 4. Perform recursive mining on this conditional FP-Tree	Job3-groupByKey: 1. Aggregate all prefix paths of E into a single groupByKey; 2. Construct a conditional FP-tree and perform mining within this task	Process Equivalence: KVBFP's groupByKey operation simulates the process of extracting conditional pattern bases in FP-Growth. Each groupByKey task is logically equivalent to a single recursive call to FP-Growth.
Results Merge	Stepwise Accumulation: During the recursive process, the final result is obtained by combining the item sets extracted from each conditional FP tree with their corresponding suffixes	Job4: All groupByKey tasks directly output the frequent item sets they mine, then aggregate and output the final results	Complete and non-redundant results: Each frequent item is processed as a suffix in only one groupByKey task, ensuring results are identical to those from FP-Growth

4. Performance And Evaluation

To comprehensively evaluate the KVBFP, this study designed tests for algorithm stability, cluster interaction frequency, load balancing, and operational efficiency comparisons. By simulating real-world scenarios, we measured various algorithm metrics and compared the mining efficiency gains achieved by KVBFP relative to PFP and FP-Growth.

4.1. Dataset

The experimental data originates from a paper cited on the Kaggle website [17]. The data was generated by the paper's authors using scripts and made freely available to researchers. The fundamental characteristics of the data are shown in Table 8.

Table 8. Experimental data characteristics table

Characteristics	Value
Transaction	5,000,000
Number of items	50,000
Max transaction	5 – 100
Frequent set density	0.1 – 0.8
Data file size	854.71M

4.2. Platform Configuration

The experimental platform consists of six PCs and a network switch, connected in a star topology. Table 9 summarizes the hardware and software specifications used in the experiment.

Table 9. Platform details of software and hardware

Hardware		Software	
CPU	Intel i7-12700K, 12-Core, 3.6GHz	**OS**	CentOS 7.9
RAM	32GB DDR4	**Java Runtime**	OpenJDK 11
Storage	2TB HDD, 7200 RPM	**Distributed Engine**	Apache Spark 3.3.2
Motherboard	Gigabyte B660M DS3H AX	**Simulation Tool**	MATLAB R2021a
Network Switch	Cisco Nexus 3172PQ 10GbE	**Monitoring Stack**	Prometheus 2.43
		Development IDE	Eclipse IDE for Enterprise Java

4.3. Algorithm Stability Testing

To validate the stability of the KVBFP in real distributed environments, three types of typical fault simulation experiments were designed on a Spark cluster. These scenarios include: worker node crashes, HDFS data block loss, and network communication failures. The support threshold was set to 0.01 for all three test sets.

4.3.1. Work Node Failure Test

Launch the cluster, upload the dataset to HDFS, and run KVBFP to mine the dataset continuously. At the 110th second of task execution, manually shut down one Worker node to simulate a failure scenario. Observe three metrics-CPU utilization, memory utilization, and task completion progress-after the Spark cluster detects the node failure. Test results are shown in Figure 6.

CPU Data Acquisition Script： sar -u 30

The calculation method for CPU Usage refers to Equation 3.

$$CPU(t) = \frac{\sum_{n=0}^{N}(usr_n(t)+sys_n(t))\cdot cores_n}{\sum_{n=0}^{N} 100\cdot cores_n} \times 100\% \qquad (3)$$

Where usr$_n$ and sys$_n$ represent the CPU usage percentage for node n, and cores$_n$ denotes the number of CPU cores corresponding to node n. The weighted average is employed primarily to account for varying CPU configurations across cluster computers. By calculating CPU usage based on these weights, the method ensures that CPU utilization accurately reflects the actual computational load across the entire cluster. Memory data collection script: sar -r 30. The memory usage calculation method refers to Equation 4.

$$Mem(t) = \frac{\sum_{n=0}^{N}\left(used_{mem_n(t)}\right)}{\sum_{n=0}^{N} total_{mem_n}} \times 100\% \qquad (4)$$

Where used_mem$_n$ represents the memory usage of the nth node, and total_mem$_n$ denotes the memory size of the nth node.

Task completion progress data collection command:
curl http://<driver>:4040/api/v1/applications/<appId>/stages
Memory usage calculation method:

Assume at time t, the number of completed tasks is C(t), the number of tasks currently running is R(t), the number of tasks waiting to be executed is P(t), and the total number of tasks is T = C + R + P. Task progress definition refers to Equation 5.

$$Progress(t) = \frac{C(t)}{T} \times 100\% \qquad (5)$$

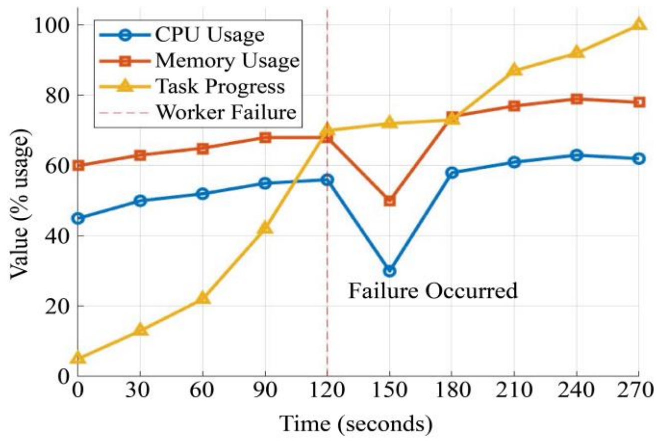


Fig. 6 Cluster indicators under worker node failure

As shown in Figure 6, when the mining task reached 110 seconds, one PC was shut down, causing a brief, sharp drop in both CPU and memory utilization. This occurred because the system halted data mining for approximately 60 seconds to reallocate system resources. By approximately 180 seconds, resource reallocation concluded, and mining resumed. CPU and memory utilization returned to pre-failure levels, though slightly elevated compared to before. This increase resulted from the cluster compensating for the lost computational resources of one PC by raising overall CPU and memory usage. Task progress virtually halted between 120 and 180 seconds, confirming no data mining occurred during this minute-long interval. Overall, despite the node failure, the KVBFP successfully completed the data mining task.

4.3.2. Data Block Loss Test

In this test, HDFS's default Data Block size of 128MB and default redundancy level of 3 were used. The 854.71MB test data file could be divided into 7 blocks, totaling 21 blocks, including redundancy copies. Therefore, manually deleting 2 random blocks could ensure the system could recover the original data. After starting the cluster and uploading the dataset to HDFS, KVBFP was run to perform continuous mining on the dataset. At 50 seconds into the algorithm's execution, two random blocks were manually deleted from HDFS. Four metrics were monitored: CPU usage, memory usage, block count, and task progress within the Spark cluster. Test results are shown in Figure 7.

Command for collecting data block counts:
hdfs fsck /path/to/dataset -files -blocks -locations
The data block calculation method refers to Equation 6.

$$ReplicaCount(t) = \sum_{\mathrm{b\in B(t)}} replication(b,t) \qquad (6)$$

Here, B(t) denotes the set of logical blocks for the file. In this example, with a block size of 128 MB, the 854 MB file is divided into 7 blocks. Given a replication factor of 3, B(t) equals 21, where b represents the number of blocks per node.

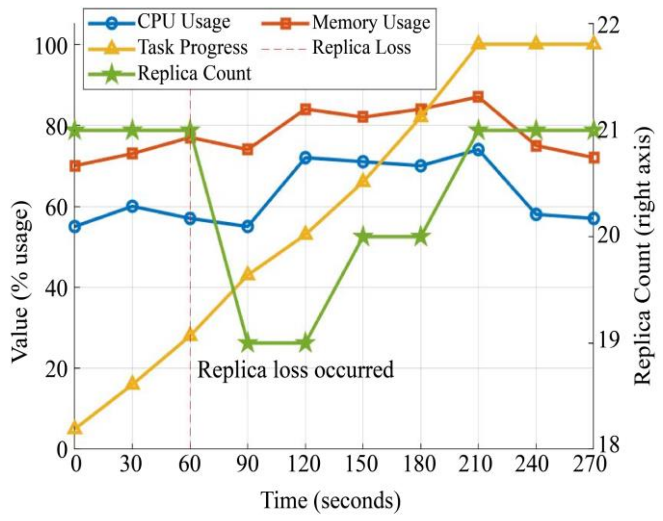


Fig. 7 Cluster indicators under block loss

As shown in Figure 7, when the mining task reached 60 seconds, two blocks were randomly deleted manually. Both CPU and memory usage experienced a slight increase. This occurred because the system invoked HDFS's recovery mechanism to replenish the deleted blocks, thereby increasing CPU and memory consumption. Once the replenishment task was completed, both usage metrics returned to their previous levels. After the Replica Count was deleted, the system detected a reduction from 21 to 19 replicas. Following approximately 30 seconds of adjustment, the first block was recovered. After roughly 30 seconds, the second block was recovered, restoring the previous state. The task execution progress remained unaffected, and all mining tasks were successfully completed around 210 seconds. Overall data indicates that even when data block loss occurs in the system, the KVBFP can still complete the mining task.

4.3.3. Network Communication Error

The Shuffle phase is the most network-intensive stage in the Spark system, generating massive data transfers. Network load during Shuffle typically accounts for over 70% of the entire task's I/O. Shuffle Fetch retrieves data from shuffle files on other nodes and delivers it to reduceByKey. When network failures occur, Shuffle Fetch interrupts and continuously retries. Therefore, we inject network failures during the Shuffle phase and characterize network status by the number of Shuffle Fetch failures. After cluster startup, the KVBFP runs to perform data mining on the dataset. At 80 seconds into execution, delay and packet loss are injected into two arbitrary PCs (Worker1 and Worker2 in this example). After 60 seconds, the faults are removed. Four metrics are observed: CPU, memory, Shuffle Fetch Failures, and task execution progress. Test results are shown in Figure 8.

Fault injection command: sudo tc qdisc add dev eth0 root netem delay 400ms loss 10%
Fault removal command: sudo tc qdisc del dev eth0 root

Shuffle Fetch Failures collection command:
curl -s http://<driver_host>:4040/api/v1/applications/<appId>/stages (Execute command every 30 seconds)

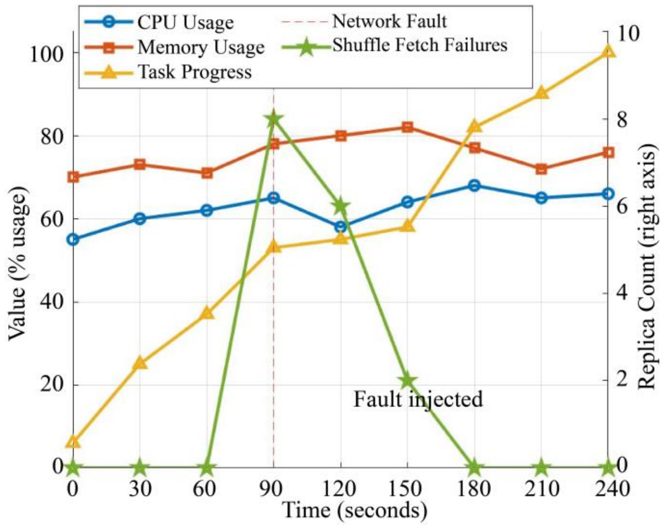


Fig. 8 Cluster indicators under network fault

As shown in Figure 8, CPU utilization remained nearly stable before fault injection. After injecting network failures, some compute threads entered a blocked state due to failed data fetching, causing a brief dip in CPU utilization. Upon removing the faults, utilization gradually recovered to its previous state.

Memory usage slightly increased after the network fault injection. This occurred because workers needed to cache more unfinished shuffle blocks and intermediate data, increasing memory pressure. After network recovery, the caches were gradually released, and memory usage returned to normal levels. Shuffle Fetch Failures surged sharply during the fault injection.

This was due to frequent data fetch failures in workers. However, Spark's retry mechanism promptly handled these failed requests, allowing the system to stabilize again after the fault ended. Task progress noticeably slowed during the 90-150s fault window, indicating the algorithm entered a waiting state due to the failure. As network connectivity recovered, the progress curve accelerated upward again, ultimately reaching 100% at 240s. Compared to fault-free conditions, task completion time was delayed, but the task was still completed successfully overall. This demonstrates that KVBFP maintains good fault tolerance under network failures.

4.4. Communication Frequency Test

This test group compared the communication frequency between clusters during mining tasks between the KVBFP and PFP. After launching the clusters, both KVBFP and PFP were run on the Spark platform with a support threshold of 0.01. The communication volume per PC during execution for both algorithms is shown in Figure 9.

Data Collection Method: A cluster monitoring system built using Prometheus + Node Exporter was employed to design a data collection mechanism for communication behavior. The Node Exporter service was deployed on each Worker node. Through Prometheus configuration files, nodes were identified as independent job instances, enabling the separate collection of inbound and outbound network metrics for each node. The sampling frequency was set to 10 seconds, with the sampling interval spanning from 10 seconds after program startup to 10 seconds before completion. PromQL queries are used to extract the network interaction frequency per unit time between each pair of Worker nodes until the algorithm completes.

Collection Commands (using Worker1 as an example):
Worker1 send packet count:
```
rate(node_network_transmit_packets_total{instance="worke
r1-ip:9100", device=~"eth0|ens.*"}[1m])
```
Worker1 receive packet count:
```
rate(node_network_receive_packets_total{instance="worker
1-ip:9100", device=~"eth0|ens.*"}[1m])
```
To illustrate communication traffic trends during algorithm execution, collect PC1-PC6 communication data (send + receive) every 10 seconds on each node.

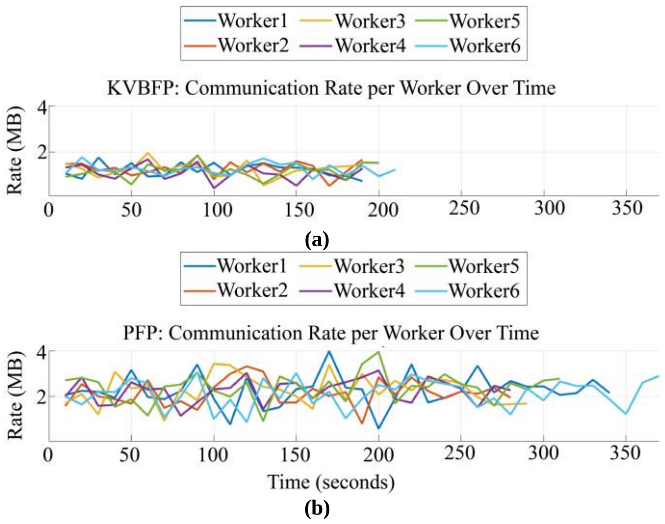


Fig. 9 Communication rate of KVBFP and PFP

To observe the overall send and receive data volumes of KVBFP and PFP at each node, the communication volume throughout the entire execution process can be accumulated. The calculation formula refers to Equation 7, where T is the program runtime and Δt is the sampling interval. The accumulated results are shown in Figure 10.

$$Data\,Volume_{send/recv} = \sum_{\mathrm{t}=0}^{T} Network\ Rate_{send/recv}(t) \times \Delta t \tag{7}$$

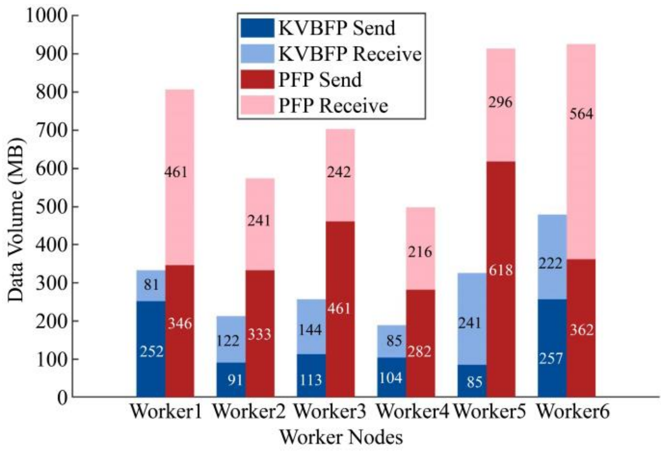
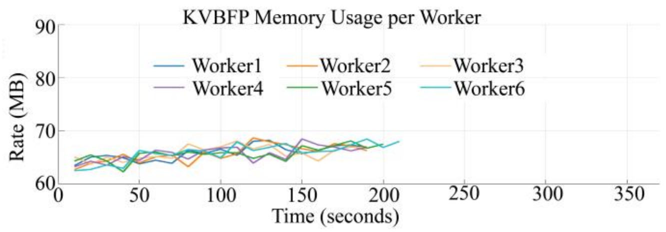


Fig. 10 Stacked communication volume of KVBFP and PFP

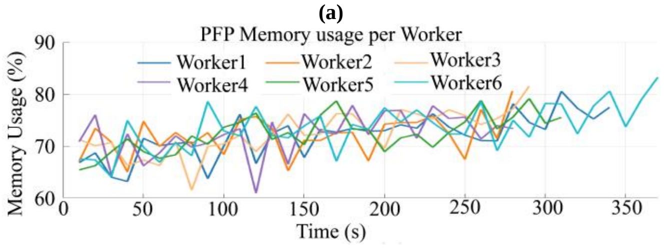
Figure 9(a) shows that the KVBFP maintains a relatively stable communication rate of 1~2 MB/s throughout its 210-second runtime, with minimal overall fluctuation. In Figure 9(b), the PFP generally maintains a communication rate of 2~4 MB/s throughout its 370-second runtime cycle. Its average rate is nearly double that of KVBFP, with multiple instances of noticeable peak fluctuations. This indicates that PFP relies more heavily on network transmission during execution and involves more frequent interactions between nodes. This difference stems from the algorithms' underlying principles: KVBFP performs two rounds of key-value pair operations to count frequent items and reconstruct the transaction table. It then employs parallel mining using local sub-FP-Trees, avoiding the centralized construction of a global FP-Tree. This approach localizes communication and maintains low, stable communication demands. In contrast, PFP must first establish a global FP-Tree during execution, group items based on prefix distance, and redistribute them to nodes. This process involves multiple rounds of global communication and data distribution, resulting in a heavier and more volatile communication load. As shown in Figure 10, KVBFP reduces total communication volume by approximately 55% compared to PFP. Communication across KVBFP nodes remains relatively balanced, ranging between 180 and 500 MB. In contrast, PFP's communication volume generally exceeds 500 MB, with some nodes approaching 1000 MB—nearly double that of KVBFP. Notably, Worker5 and Worker6 exhibit significantly higher communication volumes than other nodes, reflecting an uneven distribution. This imbalance stems directly from PFP's prefix-distance-based grouping strategy, which cannot guarantee complete equilibrium under large-scale data and data skew conditions, consequently overloading certain nodes with excessive communication pressure.

4.5. Load Balancing Test

This test group compares the load balancing across cluster nodes during runtime between the KVBFP and the PFP, with a support threshold of 0.01. For both algorithms, CPU and memory usage data from all PCs in the cluster are collected every 10 seconds. The sampling interval spans from 10 seconds after program startup to 10 seconds before shutdown, enabling continuous tracking of the entire load balancing process. Test results are shown in Figures 11 and 12.
CPU usage collection command: sar -u 10
Memory usage collection command: sar -r 10

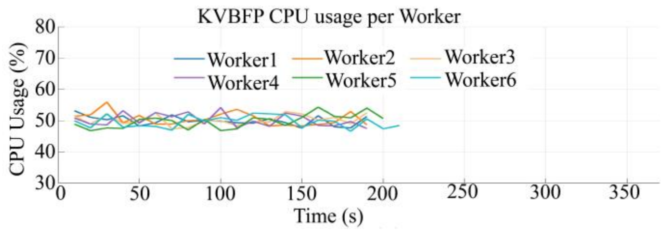


(a)

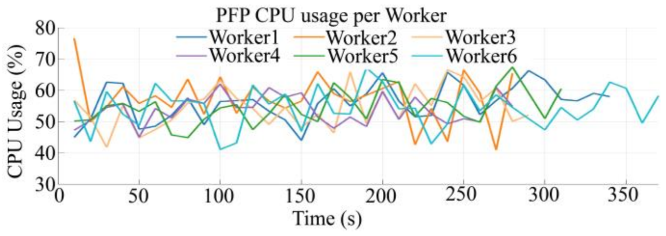


(b)

Fig. 11 Memory usage of KVBFP and PFP



(a)



(b)

Fig. 12 CPU usage of KVBFP and PFP

At each time sampling point, the degree of dispersion in CPU and memory usage across workers represents the load balance of the cluster at that moment. The variance corresponding to each time sampling point can be calculated by computing the average variance. This average variance serves as a measure of load balance, with the calculation formula referencing Equation 8.

$$\overline{Var} = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{n}\sum_{j=1}^{n}\left(\chi_{ij} - \frac{1}{n}\sum_{k=1}^{n}\chi_{ik}\right)^2\right) \quad (8)$$

m: Number of Workers
n: Number of sampling points per Worker
x_{ij}: CPU/memory utilization of the i-th Worker at the j-th sampling point

By calculation, Average variance bar charts of memory and CPU usage for PFP and KVBFP are plotted at each sampling point. The calculation interval spans from 10 seconds after program startup until at least two computers remain operational. For the KVBFP, this ranges from 10 to 200 seconds; for the PFP, it ranges from 10 to 340 seconds. The average CPM/memory variance bar chart between KVBFP and PFP is shown in Figure 13.

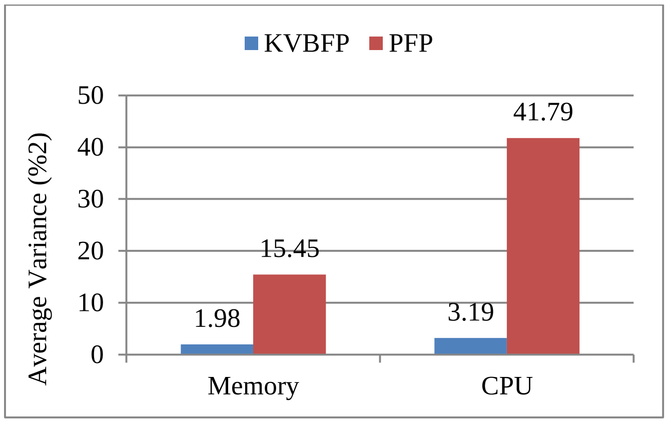


Fig. 13 Comparison of average CPU/memory variance between KVBFP and PFP

Figure 11(a) shows that the KVBFP exhibits uniform memory curves with minimal fluctuations across all worker nodes, generally maintaining usage between 64% and 68%. This indicates KVBFP achieves more balanced task partitioning, preventing resource bottlenecks on specific nodes. In Figure 11(b), the memory curve for the PFP is noticeably more dispersed, with some nodes experiencing utilization rates exceeding 80% during operation. This reflects the frequent task redistribution and data replication during its execution, leading to uneven memory usage. A similar trend is evident in the CPU utilization graph (Figure 12). KVBFP maintains CPU usage across all worker nodes between 45% and 57%, with a smooth line indicating that KVBFP's approach of splitting the Global FP-Tree using key-value pairs effectively reduces redundant scans and intermediate result calculations, thereby lowering computational resource consumption. In contrast, the PFP's CPU usage exhibits significantly more pronounced fluctuations, with some nodes even experiencing instantaneous spikes. This is closely related to its computation involving full table scans of global frequent item sets and the recursive construction of conditional FP-trees. Figure 11 and Figure 12 also reveal trailing phenomena in some nodes during the later stages of algorithm execution. Worker6 was the last node to complete mining tasks, but KVBFP exhibited significantly less trailing than PFP. Figure 13 quantifies the dispersion of CPU and memory usage across nodes using variance metrics. KVBFP exhibits an 87.2% lower CPU usage variance and a 92.4% lower memory usage variance compared to PFP. This stems from KVBFP's implementation of key-value pairs throughout the entire process (from database scanning to mining), achieving true compatibility with Spark. Spark leverages its superior task scheduling capabilities to maximize load balancing across all nodes in the cluster.

4.6. Algorithm Efficiency Comparison Test

This test group compares the time consumed by FP-Growth, PFP, and KVBFP when mining the dataset, with FP-Growth testing conducted on a single computer. Under fixed software and hardware conditions, the mining time for all three algorithms depends solely on the support threshold and transaction count. Therefore, this test group comprises two experiments: the first fixes the support threshold while varying the transaction count; the second fixes the transaction count while varying the support threshold.

Node runtime collection Java code:

```
long start = System.currentTimeMillis();
... // process
long end= System.currentTimeMillis();
long duration = end - start;
```

4.6.1. Fixed Support Threshold

The minimum support threshold is fixed at 0.01, with the transaction count ranging from 500,000 to 5,000,000, increasing by 500,000 transactions each time. For each algorithm, the total mining time is recorded. The test results are shown in Figure 14.

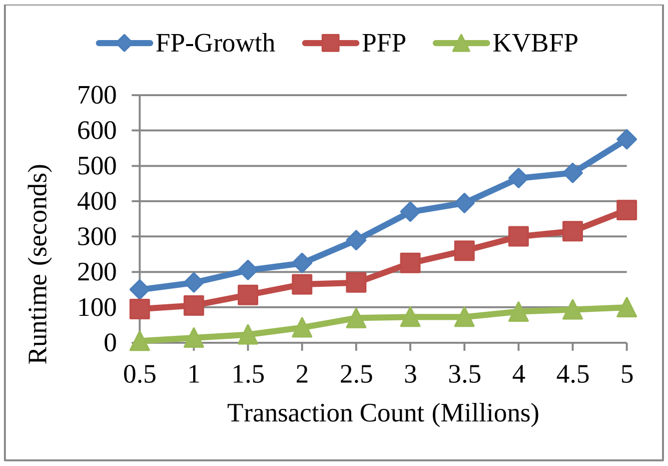


Fig. 14 Efficiency of FP-growth, PFP, and KVBFP (fixed support)

As shown in Figure 14, the KVBFP consistently demonstrates the lowest runtime, completing processing of 5 million transactions in just 214 seconds-significantly outperforming PFP's 371 seconds and FP-Growth's 572 seconds. Compared to FP-Growth, KVBFP achieves a 62.6% improvement, and compared to PFP, it achieves a 42.3% improvement. The KVBFP curve exhibits minimal fluctuation and a gentler growth slope, indicating higher mining efficiency and stability, making it particularly suitable for large-scale data mining scenarios.

4.6.2. Fixed Transaction Count

With the transaction count fixed at 5 million, the support threshold was varied from 0.1 to 0.8, and the total mining time was recorded for each setting. The test results are shown in Figure 15.

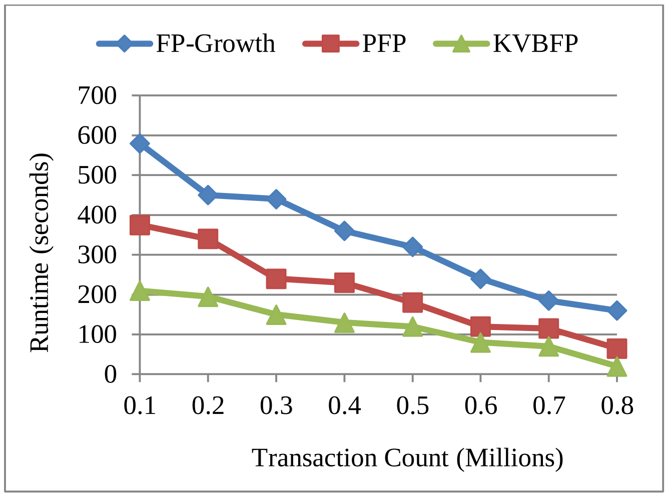


Fig. 15 Efficiency of FP-growth, PFP, and KVBFP (fixed transactions)

With a fixed transaction count of 5 million, the runtime of all three algorithms decreases as the minimum support threshold increases from 0.1 to 0.8. KVBFP demonstrated lower runtime and a more stable decline trend across all support levels. Overall, its mining efficiency improved by approximately 44.7% on average compared to PFP and by as much as 67.7% compared to FP-Growth. FP-Growth and PFP incurred significant overhead at low support levels, being heavily influenced by the number of frequent itemsets, resulting in lower efficiency than KVBFP.

5. Conclusion

In the era of big data, traditional association rule mining algorithms like FP-Growth can no longer meet practical demands when handling massive datasets. Therefore, parallelization modifications are required to make FP-Growth compatible with big data platforms. This study proposes a key-value pair-based FP-Growth, KVBFP. This algorithm simulates the entire FP-Growth process using key-value pairs, enabling compatibility with the Spark platform. It achieves significant improvements over existing parallelization schemes in mining efficiency, load balancing, and communication overhead. The proposed algorithm provides a theoretical reference for parallelizing other data mining algorithms. In practical applications, particularly in industries demanding high mining efficiency, this approach offers a novel solution. However, this study has limitations: experiments conducted on only six computers cannot comprehensively evaluate all algorithmic metrics, and the tested dataset is specialized for the experimental environment, lacking real-world data validation. Addressing these limitations represents key directions for future research.

Conflict of Interest

The authors declare that there are no financial or non-financial conflicts of interest influencing this work.

Funding

This work was not supported by any funding agency, grant, or sponsorship

References

[1] Raghavendra Kumar Chunduri, and Aswani Kumar Cherukuri, "Scalable Algorithm for Generation of Attribute Implication Base using FP-Growth and Spark," *Soft Computing*, vol. 25, pp. 9219-92401, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[2] Jeffri Prayitno Bangkit Saputra, Silvia Anggun Rahayu, and Taqwa Hariguna, "Market Basket Analysis Using FP-Growth Algorithm to Design Marketing Strategy by Determining Consumer Purchasing Patterns," *Journal of Applied Data Sciences*, vol. 4, no. 1, pp. 1-12, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[3] Ali Hassani et al., "Escaping the Big Data Paradigm with Compact Transformers," *arXiv Preprint*, pp. 1-18, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[4] Trevor Hastie et al., "The Elements of Statistical Learning: Data Mining, Inference, and Prediction," *Journal of the Royal Statistical Society Series A: Statistics in Society*, vol. 173, no. 3, pp. 693-694, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[5] Chaganti Sri Karthikeya Sahith, Satish Muppidi, and Suneetha Merugula, "Apache Spark Big data Analysis, Performance Tuning, and Spark Application Optimization," *2023 International Conference on Evolutionary Algorithms and Soft Computing Techniques (EASCT)*, pp. 1-8, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[6] Farhan Ullah et al., "NIDS-VSB: Network Intrusion Detection System for VANET using Spark-Based Big Data Optimization and Transfer Learning," *IEEE Transactions on Consumer Electronics*, vol. 70, no. 1, pp. 1798-1809, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[7] Dewi Anisa Istiqomah, Yuli Astuti, and Siti Nurjanah, "Implementation of FP-Growth and Apriori Algorithms for Product Inventory," *Polinema Informatics Journal*, vol. 8, no. 2, pp. 1-6, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[8] Yang Yang et al., "A Parallel FP-Growth Mining Algorithm with Load Balancing Constraints for Traffic Crash Data," *International Journal of Computers Communications & Control*, vol. 17, no. 4, pp. 1-16, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[9] Mingzheng Li et al., "TCM Constitution Analysis Method Based on Parallel FP-Growth Algorithm in Hadoop Framework," *Journal of Healthcare Engineering*, vol. 2022, no. 1, pp. 1-14, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[10] Xinyan Wang, and Guie Jiao, "Research on Association Rules of Course Grades based on Parallel FP-Growth Algorithm," *Journal of Computational Methods in Sciences and Engineering*, vol. 20, no. 3, pp. 759-769, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[11] A. Senthilkumar, and D. Hariprasad, "A Spark Based Frequent Itemset Mining Using Resource Management for Implementation of FP-Growth Algorithm in Cloud Environment," *Annals of the Romanian Society for Cell Biology*, vol. 25, no. 4, pp. 6050-6059, 2021. [Google Scholar] [Publisher Link]

[12] Zakria Mahrousa, Dima Mufti Alchawafa, and Hasan Kazzaz, "Frequent Itemset Mining Based on Development of FP-Growth Algorithm and Use MapReduce Technique," *Association of Arab Universities Journal of Engineering Sciences*, vol. 28, no. 1, pp. 1-16, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[13] Amr Essam, Manal A. Abdel-Fattah, and Laila Abdelhamid, "Towards Enhancing the Performance of Parallel FP-Growth on Spark," *IEEE Access*, vol. 10, pp. 286-296, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[14] Priyanka Gupta, and Vinaya Sawant, "A Parallel Apriori Algorithm and FP- Growth Based on SPARK," *International Conference on Automation, Computing and Communication*, vol. 40, pp. 1-5, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[15] Youssef Fakir, Salim Khalil, and Mohamed Fakir, "Extraction of Association Rules in a Diabetic Dataset using Parallel FP-growth Algorithm under Apache Spark," *International Journal of Informatics and Communication Technology*, vol. 13, no. 3, pp. 445-452, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[16] Shubhangi Chaturvedi, Sri Khetwat Saritha, and Animesh Chaturvedi, "Spark based Parallel Frequent Pattern Rules for Social Media Data Analytics," *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, Bangalore, India, pp. 168-175, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[17] Jeff Heaton, "Comparing Dataset Characteristics that Favor the Apriori, Eclat or FP-Growth Frequent Itemset Mining Algorithms," *SoutheastCon 2016*, Norfolk, VA, USA, pp. 1-7, 2016. [CrossRef] [Google Scholar] [Publisher Link]