The Journal of Supercomputing (2020) 76:7619–7634
https://doi.org/10.1007/s11227-020-03190-5

SWEclat: a frequent itemset mining algorithm over streaming data using Spark Streaming

Wen Xiao[1,2] · **Juan Hu**[3]

Published online: 4 February 2020

Abstract
Finding frequent itemsets in a continuous streaming data is an important data mining task which is widely used in network monitoring, Internet of Things data analysis and so on. In the era of big data, it is necessary to develop a distributed frequent itemset mining algorithm to meet the needs of massive streaming data processing. Apache Spark is a unified analytic engine for massive data processing which has been successfully used in many data mining fields. In this paper, we propose a distributed algorithm for mining frequent itemsets over massive streaming data named SWEclat. The algorithm uses sliding window to process streaming data and uses vertical data structure to store the dataset in the sliding window. This algorithm is implemented by Apache Spark and uses Spark RDD to store streaming data and dataset in vertical data format, so as to divide these RDDs into partitions for distributed processing. Experimental results show that SWEclat algorithm has good acceleration, parallel scalability and load balancing.

Keywords Frequent itemset mining · Streaming data · Sliding window · Distributed · Spark Streaming

1 Introduction

Frequent itemset mining (FIM) is one of the most basic and important data mining tasks. Since it was proposed [1], it has attracted more and more attention. Classical FIM algorithms for mining static data include: Apriori [2] based on Generation-Test and its series of improved algorithms [3–5]; Frequent Pattern Growth (FP-Growth)

✉ Wen Xiao
cyees@163.com

[1] College of Computer Science and Information, HOHAI University, Nanjing, China

[2] Key Laboratory of Unmanned Aerial Vehicle Development and Data Application of Anhui Higher Education Institutes, Wanjiang University of Technology, Maanshan, China

[3] Ma'anshan Engineering Technology Research Center for Wireless Sensor Network and IntelliSense, Wanjiang University of Technology, Maanshan, China

[6] algorithm and other algorithms based on Pattern Growth; and Equivalence Class Transformation (Eclat) [7], Diffset Eclat (dEclat) [8] and other algorithms based on vertical data format. Mining frequent itemsets from data streams is one of the most important issues in FIM. It is widely used in retail chain data analysis, network traffic analysis, click-stream mining, IOT data analysis and other fields. Data stream is a continuous, unbounded, timely ordered sequence of data elements with high speed; these characteristics lead to some special limitations in Mining frequent itemsets: All elements in the data stream can only be visited once; data streams grow continuously, but memory during processing is limited and only part of the elements in data streams can be processed; the high speed of data stream also requires the high speed of mining. Therefore, the aim to design FIM algorithm for streaming data is generally required to complete the mining only by scanning the data stream once, and the complexity of time and space is relatively low to ensure that it can be completed in limited memory. Some elements accessed are generally the most recently arrived elements [9].

Using time-sensitive window to represent the most recently arrived part of the data stream is the main technique to process the data stream. There are three commonly used window models, including landmark model, the damped model and sliding window model. In the landmark model, only the data between the landmark time point and the current time point in the data stream are considered. Typical algorithms are Lossy Counting [10], Frequent Data stream Pattern Mining (FDPM) [11] and Data Stream Mining for Frequent Itemsets (DSM-FI) [12]. In damped model, different weights are given according to the order of arrival of elements in data stream. The typical algorithms are estDec [13] and so on. In sliding window model, we need to specify a fixed length time window to represent the latest arrival data. Typical algorithms are Moment [14] and Compact Pattern Stream Tree (CPS-Tree) [15]. Because the sliding window model not only extracts part of the data stream for processing, but also fully considers the value of the latest data, the sliding window model is the most commonly used in streaming data process. In this model, with the window sliding, the old elements in the data stream are deleted from the window; the newly arrived elements are inserted into the window for processing.

The core of designing FIM algorithm over data stream based on sliding window model is to select or design a data structure to store data in the window. This data structure must meet the following requirements: Because the elements in the data stream can only be visited once, this data structure can only be created by scanning the data stream once; window sliding will lead to many operations of deleting and inserting in data structure, which requires that the data structure has high efficiency in deletion and insertion; because window sliding brings the problem of "concept change," in order to mine the data in the window accurately, this data structure must store the information of all items (including frequent and infrequent items); in order to reduce the storage requirement in the mining process, it is necessary to have a high compression ratio of this data structure and a high efficiency of FIM in this data structure, that is, a better time–space efficiency. The commonly used data structures include Data Stream Tree (DS-Tree) [16], CPS-Tree [15], Parallel Stream Data (PSD-Tree) [9] based on prefix tree, and tidset [7] and diffset [8] based on vertical data format.

With the rapid development of Internet of Things, information transmission and storage technology, the amount of data generated and needed to be mined has explosive growth; we have entered the era of big data. The most basic and prominent feature of big data is the huge amount of data. FIM is a task with high computing load and storage requirements, If there are n items in the dataset, the size of search space is 2^n-1, and the computing load and storage requirements is extremely heavy. When the amount of dataset is large, distributed algorithms are needed to meet the above challenges. Popular big data distributed computing platform such as Hadoop and Spark are based on the idea of data localization, which can easily realize efficient, automatic balancing and automatic fault-tolerant distributed mining. Several FIM algorithms based on sliding window using popular big data platforms have been proposed: Vanteru et al. [17] propose an algorithm for mining frequent itemsets in data stream using sliding windows. It uses Canonical-Order Tree (CanTree) [18] to store data in an window, generates GTree to mine frequent itemsets by projecting CanTree further and implements distributed mining by Hadoop. Hadoop is mainly used for data batch processing with input, and intermediate results are stored on Hadoop Distributed File System (HDFS), which is inefficient for streaming data processing. The algorithm proposed in [19] is similar to [17]. It also uses Hadoop to realize distributed mining. The difference is that Tail Point Table Tree (TPT-Tree) is used to store data in windows. Carlos et al. [20] use Spark to parallelize Frequent Itemset Mining over Time-sensitive Streams (FIMoTS) algorithm. FIMoTS [21] uses prefix tree to save the data in the window and classifies all itemsets in the window according to the support degree. It decides whether the itemsets in the window data are retained or not by the upper and lower limits defined by the user called Type Transforming Upper/Lower Bound. Obviously, the itemsets below the lower limits will be discarded and it is an approximate mining algorithm.

In this paper, we propose a distributed FIM algorithm over streaming data named Sliding Window Eclat (SWEclat) which is based on Spark Streaming. Firstly, the algorithm uses the sliding window which suits for big data to process streaming data; second, the algorithm uses the vertical data format to store the data in the current window; third, the algorithm uses spark RDD to distributed store data in current window and uses the functions provided by spark to perform distributed parallel processing. The algorithm includes four main phases: initializing vertical data format, partitioning equivalent class conditional database, distributed mining conditional database and updating vertical database.

The rest of this paper is organized as follows: Sect. 2 introduces the preliminaries of SWEclat algorithm and discusses the related work. Section 3 proposes the SWEclat algorithm and implements the algorithm in detail using Spark Streaming. Section 4 analyzes the experimental results. Finally, conclusion is made in Sect. 5.

2 Preliminaries and related works

2.1 Problem definition

Let $I=\{i_1, i_2, \ldots, i_n\}$ be a set of *items*. Let $X=\{i_1, i_2, \ldots, i_k\} \subseteq I$ be an itemset; size k is called a *k-itemset*. Let T be a transaction, $T=\{tid, X'\}$ which *tid* is an identifier

and X' is an itemset. Given an itemset $X \subseteq I$, an transaction T contains X if and only if $X \subseteq X'$.

Let $DB = \{T_1, T_2, \ldots, T_n\}$ be a database of transactions, $|DB|$ is the number of transactions in DB. The vertical format of DB denoted DB' consists of a set of items and a list of tids containing it.

$$DB' = \{(i_j, C_{ij} = \{tid | i_j \in X, (tid, X) \in DB\})\};$$

C_{ij} is called the tidset of item i_j.

Let $DS = \{T_1, T_2, \ldots, T_n\}$ be a data stream of transactions, where T_i, $i \in [1,m]$ is the ith arrived transaction. A window W_{ij} can be referred to as a set of all transactions between the ith and jth timestamp where $j > i$ arrival of transactions, $W_{ij} = \{T_i, T_{i+1}, \ldots, T_{j-1}, T_j\}$, and the size of W_{ij} denoted as $|W_{ij}|$ is the number of transactions contained in the window.

The support of an itemset X in a Window W, denoted as $\mathrm{Sup}_W(X)$, is the number of transactions in W that contain X. An itemset X is called frequent in W only if its support is no less than an user-defined threshold *minsup* s, with $\mathrm{Sup}_W(X) \geq |W| \times s$. Given DS, W and s, the problem of FIM over streaming data is to find all the frequent itemsets which denoted as F_W in the window W.

2.2 Spark Streaming

Apache Spark [24] is a unified analytic engine for large-scale data processing based on a cluster, originally developed by AMPLab of UC Berkeley's. Spark uses Resilient Distributed Datasets (RDDs) as an abstract model of dataset and can distribute RDD among worker nodes of cluster to implement distributed processing. To speed up the process, Spark stores the intermediate results in memory rather than in HDFS like Hadoop. Spark also extends the popular MapReduce computing model to achieve more complex computing tasks such as iteration and streaming. A typical Spark cluster consists of a master node and several worker nodes. The master node manages the resources of the node and assigns tasks to the worker nodes. The working model of Spark Framework is shown in Fig. 1.

Spark Streaming is a component specially designed by Spark for streaming data. It allows users to write streaming applications using a set of APIs very close to batch processing, which is very convenient to use. Spark Streaming uses data stream RDD to represent data streams, and dataset in each sliding window is still represented by an RDD, so data stream can be regarded as a sequence of RDDs. Data stream can be created from multiple input sources, such as file system, Flume, Kafka and HDFS.

2.3 Related works

Data structures which are used to store data in windows must ensure that they can be created only by scanning data once, and can be inserted, deleted and mined efficiently. The data structures used in existing algorithms can be divided into two types: prefix tree-based and vertical data format-based.

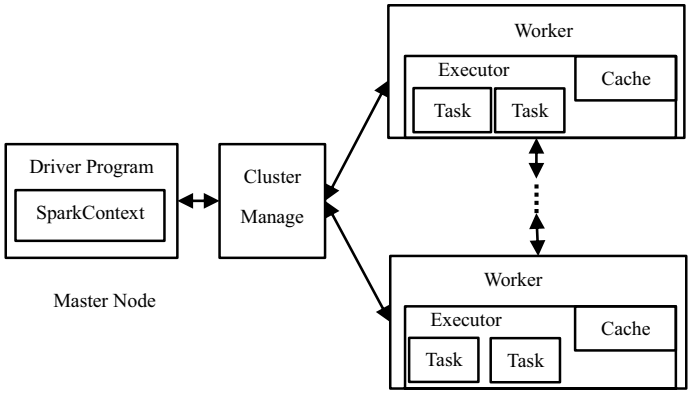


Fig. 1 Working model of Spark

The most commonly used prefix tree in FIM is the FP-Tree used in FP-Growth [6], but construct a FP-Tree needs to scan dataset twice, which is not suitable for streaming data. Most prefix tree for streaming data is extensions or modifications to FP-Tree. DS-Tree [16] can be constructed by scanning dataset only once, dividing all transactions in the window into several panes. Items in the transaction are inserted into or deleted from the prefix tree according to canonical order. After the prefix tree is constructed or adjusted, the traditional FP-Growth algorithm is used to mine the dataset. Unlike nodes in DS-Tree in canonical order, nodes in CPS-Tree [15] are arranged in descending order of support, and the descending order of support of nodes in the tree is maintained by monitored reconstruction. This reconstruction is time-consuming, especially in the case of concept change frequently, but the items in the prefix tree in descending order of support can effectively reduce the need for storage space and accelerate the speed of mining. Experiments in [15] show that CPS-Tree has better performance than DS-Tree. Similar prefix tree structures include CanTree [18], PSD-Tree [9] and so on.

There are three main vertical data formats for transaction: Eclat [7] uses the classical set of {item–tidset} key–value pairs to represent the dataset. Tidset consists of all tids of transactions which contain this item. The length of tidset is the support of item or itemset. For dense data, the size of tidset may be very large. In order to reduce storage space, dEclat [8] proposed the concept of diffset to store tids without item. Mining frequent itemsets in vertical data format is to get tidset of new candidate itemsets by running intersection operation of tidset of two itemsets. In order to further speed up the intersection operation of tidset, VIPER [22] used bit vector to represent tidset of itemsets. Every bit in a bit vector is used to indicate whether an itemset exists in a corresponding transaction. Existence is represented by number 1, while nonexistence is represented by number 0. The number of 1 in a bit vector is the support degree of item or itemset. This representation converts the intersection operation of a set into bit-by-bit AND operation of a bit vector. It has high efficiency when the data are dense. But when the dataset is sparse, it will waste a lot of space to store a large number of number 0.

Different from static dataset, the processing of streaming data needs to use window to process part of the dataset in the data stream. With the rapid sliding of window, a large number of insertion and deletion operations are needed for the data structure that stores the dataset. Let data stream DS = $\{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8\}$, and windows $W_1 = \{T_1, T_2, T_3, T_4\}$ and $W_2 = \{T_3, T_4, T_5, T_6\}$, $W_3 = \{T_5, T_6, T_7, T_8\}$. In the process of window sliding from W_1 to W_3, two transactions are inserted and deleted in each sliding. When the data stream is massive, window sliding will bring huge updating overhead. For data structures based on prefix trees, inserting or deleting a transaction requires access to the prefix tree with time complexity of $O(\log n)$, while for vertical data structures, inserting and deleting transactions only need to modify the tidset of the corresponding item, with time complexity of $O(1)$. So, the latter one has more advantages than prefix tree. The experimental results in [23] also show that the vertical data structure algorithm has greater advantages in execution time and storage requirement than DS-Tree and CPS-Tree, and is more suitable for FIM over streaming data with sliding window.

3 SWEclat algorithm

The proposed SWEclat algorithm uses the vertical data structure *VDB* in the form of <item, tidset> to represent transaction dataset *DB* in sliding window. After initialization, it uses Eclat [7] algorithm to mine *VDB* distributedly. *VDB* is modified with window sliding, and distributed mining is performed. It uses equivalence class (frequent items) to divide *VDB* into several conditional databases and distribute them to multiple worker nodes for parallel processing. Based on Spark Streaming, the algorithm SWEclat uses DStream to represent data stream and uses functions provided by Spark Streaming to operate DStream and RDD which represents dataset in a window. The algorithm consists of four phases, and the main framework is:

1. Initializing the construction of vertical data format *VDB*
    (a) Scan the horizontal format of streaming data and store it in the corresponding RDD;
    (b) Convert horizontal format to vertical format RDD; for example, the horizontal format of $\{t_1, (i_1, i_2)\}\{t_2, (i_2, i_3)\}$ is transformed into the vertical format of $\{i_1, (t_1)\}\{i_2, (t_1, t_2)\}\{i_3,(t_2)\}$;
    (c) Sort all items in the vertical format *VDB* in canonical order;
2. Partitioning conditional databases and Distribution
    (a) Delete infrequent items in *VDB*.
    (b) Divide *VDB* into several equivalent class conditional databases VDB_x by frequent items.
    (c) Distribute conditional databases VDB_x to worker nodes.

3. Distributed parallel mining of conditional databases on worker nodes

   (a) Mine VDB_x using Eclat algorithm on worker node.
   (b) Collect the mining result of each worker node to get frequent itemsets.

4. Updating *VDB* with window sliding

   (a) Update *VDB* using deleted transactions.
   (b) Update *VDB* using inserted transactions.

5. Repeating phase 2–4 until the end of the data stream.

3.1 Phase 1: initialize *VDB*

The goal of this phase is to convert transactions *lines* in horizontal format into vertical data in the form of <item, tidset> and to initialize the construction of *VDB*. The detailed procedure is shown in Algorithm 1.

Algorithm 1.InitVDB	
Input: *lines*	
Output: *VDB*	
1:	*RDD<tid,String> lines* ← input
2:	*RDD<item,List<tid>> VDB* ←
	lines.flatMapToPair(f: {tid,String} →{item, tid})
3:	*.groupByKey();*
4:	*.partitionBy(HashPartitioner(n))*
5:	*VDB.sortByKey()*

In this algorithm, firstly, the horizontal format of streaming data is arrived and stored in the RDD named *lines* (line 1). *VDB* is represented by a key–value pair of RDD. All items of transactions in the window are keys, and tidsets of items are values (line 2). The function *f* is passed to the *flatMapToPair* function of *VDB* to convert each item in T_i to the output in the form of {item: tid} (line 2). All tids of the identical item are merged through the *groupByKey* function, and the tidset of an item is stored in a collection List < tid > (line 3). At this point, the transaction in the window has been converted from horizontal format to vertical format and stored in *VDB*. In order to implement distributed and parallel processing, the default partitioner named *HashPartitioner* is used to divide *VDB* into *n* partitions (line 4). Finally, the elements in *VDB* are sorted according to item in canonical order (line 5).

3.2 Phase 2: divide and distribute conditional database

The goal of this phase is to divide *VDB* into several independent conditional databases according to the method of equivalence class partition [7] and distribute these conditional databases to worker nodes to distributed and parallel mining. The detailed procedure is shown in Algorithm 2.

Algorithm 2.DivideAndDistributeVDB	
Input: *VDB*, *s*	
Output: VDB_x	
1:	*VDB* ←*VDB.filter(f:sup(item$_x$)* ⩾*s)*
2:	*RDD<ec,<item,List<tid>>>conVDB*←
	VDB.flatMapToPair(
3:	*f: {item,List<tid>}*→*{ec, (tid,List<tid>)})*
4:	*.groupByKey()*
5:	*.partitionBy(HashPartitioner(n))*

The algorithm uses the vertical data format *VDB* obtained in the previous phase and minimum support *s* as input and outputs the conditional database with equivalent classes as key. According to the Apriori property [2], frequent itemsets must consist of frequent items. If an itemset contain non-frequent items, then this itemset cannot be frequent. Therefore, items with support less than minimum support (non-frequent items) in *VDB* can be filtered out (line 1). A key–value pair RDD named *conVDB* represents the conditional database, where the key is the equivalent class (actually frequent items) and the value is the Set of <item, tidset> (line 2) in the conditional database. Using the *flatMapToPair* function, a function *f* is passed to generate a conditional database (line 3) according to the equivalent class. The final conditional databases of equivalence classes are obtained by merging the results according to equivalence classes (line 4). Finally, the conditional database is distributed to each worker node (line 5) according to the equivalent class, which is prepared for the next phase of distributed parallel mining. *N* is the number of partitions.

3.3 Phase 3: distributed and parallel mining of conditional databases on work nodes

The task of this phase is relatively simple. The conditional database generated in the previous phase is mined using Eclat algorithm, and the final result can be obtained by collecting frequent itemsets generated by all worker nodes. The detailed procedure is shown in Algorithm 3.

Algorithm 3. DistributedMining	
Input: *conVDB,s*	
Output:*FrequentItemsets*	
1:	*RDD<String,Integer>res ←VDBec.reduceByKey(f:Eclat(s))*
2:	*FrequentItemsets ←res.collect()*

Because the conditional database *conVDB* takes the equivalent class as the key, *reduceByKey* can be used to mine each conditional database using Eclat algorithm (line 1), and *collect* function can be used to collect the results distributed on every worker node (line 2) and output the final results (line 2).

3.4 Phase 4: update VDB as the window slides

Because of the continuous and unbounded of streaming data, the basic method of processing streaming data is to use sliding window to process sub dataset in streaming data. With the sliding of the window, the old transaction leaves the window, and the new transaction enters into window. *VDB* needs to be updated for the next distributed mining. Figure 2 illustrates the above process intuitively as an example.

There is a data stream and two windows as shown in Fig. 2. When the current window is W_1, $VDB = \{a{:}1, 2, 3, 4; b{:}1, 2, 4; c{:}2, 3, 4; d{:}1, 2, 3\}$. When the window slides from W_1 to W_2, T_1 and T_2 are removed from the window, while T_5 and T_6 are inserted into the current window, and *VDB* needs to be updated. After removing T_1 and T_2, $VDB = \{a{:}3, 4; b{:}4; c{:}3, 4; d{:}3\}$, and after inserting T_5 and T_6, $VDB = \{a{:}3, 4; b{:}4; c{:}3, 4, 5, 6; d{:}3, 5; e{:}5, 6\}$. Update *VDB* is about to remove the leaving transactions and insert the newly arrived transactions from *VDB*. The detailed procedure is shown in Algorithm 4.

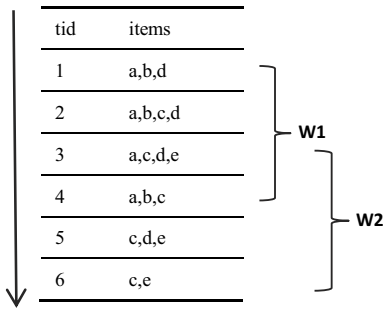


Fig. 2 An example of a sliding window

Algorithm 4.UpdateVDB

Input: DB_{del} DB_{insert}

Output: *VDB*

```
1:  DeleteList<item,tidset> ← VDB_del
2:  InsertList<item,tidset> ← VDB_insert
3:  VDB.mapValues(f:x→x.tidset.del(DeleteList(x)))
4:  VDB.mapValues(f:(x,tidset)→
5:      if VDB.keys().contain(x) tidset.add(InsertList(x)))
6:      else VDB.Add(x,InsertList(x))
7:      end if
8:  )
9:  VDB.sortByKey()
```

The algorithm uses the removed and inserted dataset as input and outputs the updated *VDB*. According to algorithm 1 (line 3–9), VDB_{del} which contain removed transactions and VDB_{insert} which contain inserted transactions are transformed into vertical data format in the form of <item, tidset> and stored in *DeleteList* and *InsertList*, respectively, to speed up subsequent operations (line 1–2). For removed transactions, it is relatively simple and just need to modify the tidset of the corresponding item in *VDB*. For inserted transactions, there may be items that are not included in *VDB*. In this case, new element is added (line 6). For existing items, just modify their tidset directly (line 5).

After the *VDB* is updated, the new *VDB* is used to re-partition conditional databases for distributed and parallel mining until the end of the data stream.

4 Experimental results and analysis

4.1 Experiments settings

In order to evaluate the performance of SWEclat algorithm, experiments are divided into three groups. The first group of experiments verifies the acceleration of the algorithm, that is, the improvement in runtime. Since we did not find a similar algorithm based on Spark Streaming and using sliding window technology to mine all frequent itemsets in the window, we compared the distributed version with the non-distributed version of SWEclat algorithm named NoDisSWEclat. The second group of experiments checked the distributed scalability of the algorithm and the runtime of the algorithm under different distributed degrees. The third group of experiments examined the load balancing after using the default HashPartitioner to partition the related RDDs.

In the experiments, mushroom and retail, the standard datasets from FIMI Repository [25], are selected, and their attributes are shown in Table 1. Mushroom is dense, while retail is a sparse dataset. To simulate streaming data, the famous NCAT which is a general-purpose command-line tool for reading, writing, redirecting and encrypting data across a network in Linux is used to send dataset to a specific port. The batch time of streaming data is set to 500 ms; the window size is two times batch size; and the sliding step is 1 batch.

We have conducted all the experiments on a 64-bit architecture with 16 cores on Intel Xeon E5-2620 and with 64 GB of RAM functioning over an operative system with CentOS 7.5. The algorithm is implemented in Java with Spark version 2.4.3 and Java version 1.8.0.

4.2 Acceleration

This group of experiments compared the running time of SWEclat and NoDisSWEclat under different minimum supports. The experimental results are shown in Fig. 3.

Figure 3 shows that SWEclat algorithm has better performance than its non-distributed version, and its acceleration performance is better with the decrease in minimum support, which leads to the increase in computation overhead in each window. Because of the characteristics of RDD used by Spark Streaming, the operation of RDD can be automatically distributed to worker nodes and run in parallel. For dense data, its acceleration is more obvious than that for sparse data.

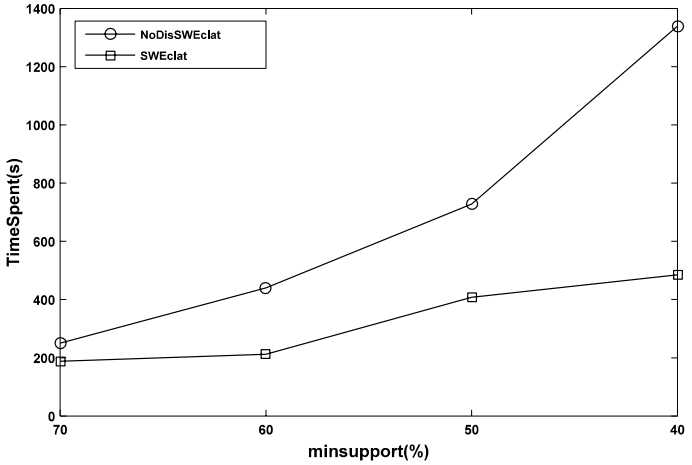
4.3 Scalability

This set of experiments examines the scalability of the algorithm and the runtime of the algorithm at different degrees of parallelism which realized by adjusting the number of partitions of *VDB* and *conVDB*. The minimum support is set at 40% on mushroom dataset and 0.5% on retail dataset. The experimental results are shown in Fig. 4.

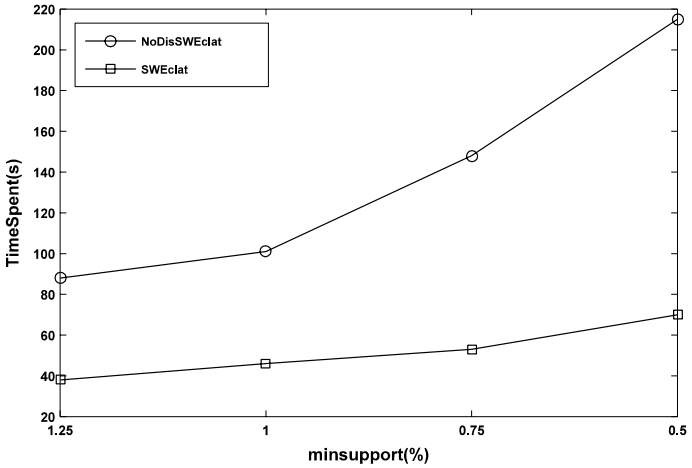
Figure 4 shows that the runtime on both datasets decreases with the increase in the number of RDD partitions. The main steps of SWEclat algorithm can be executed in parallel by distributing the corresponding RDD into multiple partitions. As the number of partitions increases, the processing overhead of each partition will decrease correspondingly, and the overall performance of the algorithm will be improved.

Table 1 Characteristics of datasets used in experiments

Dataset	Number of transactions	Number of items	Average length of transactions	Size (KB)
Mushroom	8124	119	29	558
Retail	88,162	16,470	10	4070



(a) mushroom dataset



(b) retail dataset

Fig. 3 Running time of two algorithms under different minimum supports

4.4 Load balancing

As discussed above, algorithms based on Spark Streaming can use Spark RDD to store data and divide the RDD into multiple partitions so that operations on RDD can be executed in parallel to improve performance. Specifically to SWEclat algorithm, it mainly divides *VDB* and *conVDB*, which are used to store vertical transaction data, and realizes parallel operations such as conditional database mining and VDB updating. In order to achieve load balancing, each partition size after partitioning is almost equal. In order to verify the validity of using default HashPartitioner, different partitioning numbers are tested for two datasets. The

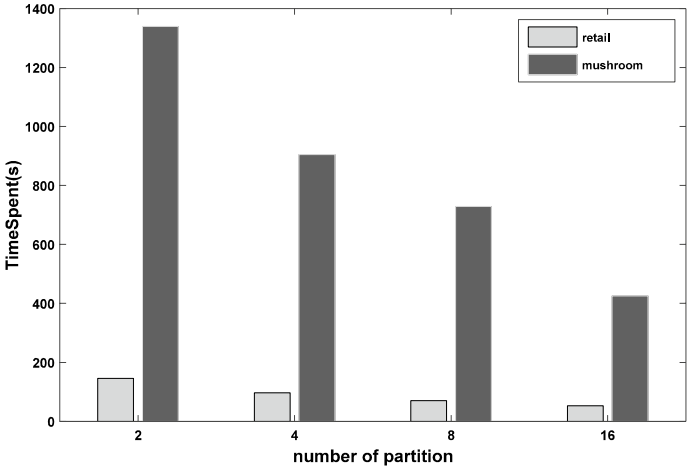


Fig. 4 Runtime of SWEclat under different numbers of partitions

mushroom dataset uses 40% minimum support, and the retail dataset uses 0.5% minimum support. The experimental results are shown in Table 2.

In Table 2, the first column is the name of the dataset, the second is the No. of each partition, the third to fifth columns are the size of each partition when *VDB* is divided into 4, 8 and 16 partitions, and the sixth to eighth columns are the size of each partition when *conVDB* is divided into 4, 8 and 16 partitions. From the table, we can see that using default HashPartitioner to partition RDD can achieve better load balancing. VDB and conVDB partitions are basically the same size. When the amount of data is larger, the effect is better, which creates a better condition for parallel execution of related operations in RDD.

5 Conclusions

In this paper, we study the problem of distributed frequent itemset mining over streaming data and propose a distributed algorithm named SWEclat based on Spark Streaming which is a popular streaming data processing platform. SWEclat algorithm uses sliding window technology to access partial data in the streaming data. Because the vertical data structure used by Eclat algorithm has better insertion and deletion performance, and can be divided into independent search space and conditional database, it is used for storing data in current window. The proposed algorithm use spark RDD to save and partition the data in the current window, and use functions provided by spark to realize distributed and parallel mining. Experiments show that the proposed SWEclat algorithm has good acceleration and scalability, and can be used to solve many problems based on frequent itemset mining of streaming data.

Table 2 Partition sizes of VDB and conVDB under different numbers of partitions

Dataset	Partition no.	Size of VDB partitions			Size of conVDB partitions		
Mushroom	P0	29	14	11	19	7	1
	P1	29	15	9	22	9	0
	P2	31	14	6	19	10	1
	P3	30	17	8	19	9	0
	P4	–	14	10	–	10	0
	P5	–	14	6	–	12	1
	P6	–	16	5	–	12	1
	P7	–	15	7	–	10	1
	P8	–	–	5	–	–	2
	P9	–	–	6	–	–	2
	P10	–	–	6	–	–	1
	P11	–	–	6	–	–	2
	P12	–	–	8	–	–	3
	P13	–	–	7	–	–	2
	P14	–	–	9	–	–	1
	P15	–	–	10	–	–	3
Retail	P0	4116	2062	1127	51	27	14
	P1	4119	2054	1133	51	27	13
	P2	4119	2060	1077	66	26	15
	P3	4116	2061	1123	53	24	10
	P4	–	2057	965	–	24	23
	P5	–	2059	1000	–	26	17
	P6	–	2055	939	–	40	14
	P7	–	2062	1040	–	27	13
	P8	–	–	1107	–	–	13
	P9	–	–	980	–	–	12
	P10	–	–	926	–	–	14
	P11	–	–	928	–	–	14
	P12	–	–	947	–	–	17
	P13	–	–	1097	–	–	7
	P14	–	–	1061	–	–	11
	P15	–	–	1020	–	–	14

Acknowledgements This work was supported by the Natural Science Foundation of the universities in Anhui province (No. KJ2019A1274).

References

1. Agrawal R, Imieliński T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp 207–216
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, vol 1215, pp 487–499
3. Park JS, Chen MS, Yu PS (1997) Using a hash-based method with transaction trimming for mining association rules. IEEE Trans Knowl Data Eng 9(5):813–825
4. Ozel SA, Guvenir HA (2001) An algorithm for mining association rules using perfect hashing and database pruning. In: 10th Turkish Symposium on Artificial Intelligence and Neural Networks. Springer, Berlin, pp 257–264
5. Brin S, Motwani R, Ullman JD, Tsur S (1997) Dynamic itemset counting and implication rules for market basket data. In: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, pp 255–264
6. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. ACM Sigmod Rec 29(2):1–12
7. Zaki MJ (2000) Scalable algorithms for association mining. IEEE Trans Knowl Data Eng 12(3):372–390
8. Zaki MJ, Gouda K (2003) Fast vertical mining using diffsets. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 326–335
9. He Y, Yue M (2014) Parallel frequent itemset mining on streaming data. In: 2014 10th International Conference on Natural Computation (ICNC). IEEE, pp 725–730
10. Manku GS, Motwani R (2002) Approximate frequency counts over data streams. In: VLDB'02: Proceedings of the 28th International Conference on Very Large Databases. Morgan Kaufmann, pp 346–357
11. Yu JX, Chong Z, Lu H, Zhou A (2004) False positive or false negative: mining frequent itemsets from high speed transactional data streams. In: VLDB, vol 4, pp 204–215
12. Li HF, Shan MK, Lee SY (2008) DSM-FI: an efficient algorithm for mining frequent itemsets in data streams. Knowl Inf Syst 17(1):79–97
13. Chang JH, Lee WS (2003) Finding recent frequent itemsets adaptively over online data streams. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 487–492
14. Chi Y, Wang H, Yu PS, Muntz RR (2004) Moment: maintaining closed frequent itemsets over a stream sliding window. In: Fourth IEEE International Conference on Data Mining (ICDM'04). IEEE, pp 59–66
15. Tanbeer SK, Ahmed CF, Jeong BS, Lee YK (2009) Sliding window-based frequent pattern mining over data streams. Inf Sci 179(22):3843–3865
16. Leung CKS, Khan QI (2006) DSTree: a tree structure for the mining of frequent sets from data streams. In: Sixth International Conference on Data Mining (ICDM'06). IEEE, pp 928–932
17. Kusumakumari V, Sherigar D, Chandran R, Patil N (2017) Frequent pattern mining on stream data using Hadoop CanTree-GTree. Procedia Comput Sci 115:266–273
18. Leung CKS, Khan QI, Li Z, Hoque T (2007) CanTree: a canonical-order tree for incremental frequent-pattern mining. Knowl Inf Syst 11(3):287–311
19. Bo C, Yong DC, Xiue G (2016) A frequent pattern parallel mining algorithm based on distributed sliding window. Comput Syst Sci Eng 31(2):101–107
20. Fernandez-Basso C, Francisco-Agra AJ, Martin-Bautista MJ, Ruiz MD (2019) Finding tendencies in streaming data using big data frequent itemset mining. Knowl Based Syst 163:666–674
21. Li H, Zhang N, Zhu J, Cao H, Wang Y (2014) Efficient frequent itemset mining methods over time-sensitive streams. Knowl Based Syst 56:281–298
22. Shenoy P, Haritsa JR, Sudarshan S, Bhalotia G, Bawa M, Shah D (2000) Turbo-charging vertical mining of large databases. ACM Sigmod Rec 29(2):22–33

23. Deypir M, Sadreddini MH (2011) EclatDS: an efficient sliding window based frequent pattern mining method for data streams. Intell Data Anal 15(4):571–587
24. Apache Spark. http://spark.apache.org/. Accessed 27 Jan 2020
25. Frequent Itemset Mining Dataset Repository. http://fimi.uantwerpen.be/data/. Accessed 27 Jan 2020