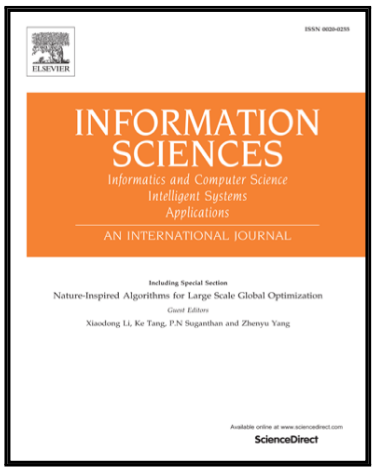


Mining Maximal Frequent Patterns in Transactional Databases and Dynamic Data Streams: a Spark-based Approach

Md. Rezaul Karim, Michael Cochez, Oya Deniz Beyan,
Chowdhury Farhan Ahmed, Stefan Decker

PII: S0020-0255(17)31126-X
DOI: 10.1016/j.ins.2017.11.064
Reference: INS 13292

To appear in: *Information Sciences*

Received date: 13 February 2016
Revised date: 27 November 2017
Accepted date: 30 November 2017

Please cite this article as: Md. Rezaul Karim, Michael Cochez, Oya Deniz Beyan, Chowdhury Farhan Ahmed, Stefan Decker, Mining Maximal Frequent Patterns in Transactional Databases and Dynamic Data Streams: a Spark-based Approach, *Information Sciences* (2017), doi: 10.1016/j.ins.2017.11.064

Mining Maximal Frequent Patterns in Transactional Databases and Dynamic Data Streams: a Spark-based Approach

Md. Rezaul Karim[a,b], Michael Cochez[a,b,d], Oya Deniz Beyan[b,a], Chowdhury Farhan Ahmed[c], Stefan Decker[a,b]

[a] *Fraunhofer FIT, Schloss Birlinghoven, D-53754 Sankt Augustin, Germany*
[b] *Chair of Computer Science 5 - Information Systems, RWTH Aachen University, Germany*
[c] *Department of Computer Science & Engineering, University of Dhaka, Bangladesh*
[d] *Faculty of Information Technology, University of Jyvaskyla, Finland*

Abstract

Mining maximal frequent patterns (*MFPs*) in transactional databases (*TDBs*) and dynamic data streams (*DDSs*) is substantially important for business intelligence. MFPs, as the smallest set of patterns, help to reveal customers' purchase rules and market basket analysis (*MBA*). Although, numerous studies have been carried out in this area, most of them extend the main-memory based Apriori or FP-growth algorithms. Therefore, these approaches are not only unscalable but also lack parallelism. Consequently, ever increasing big data sources requirements cannot be met. In addition, mining performance in some existing approaches degrade drastically due to the presence of null transactions. We, therefore, proposed an efficient way to mining *MFPs* with *Apache Spark* to overcome these issues. For the faster computation and efficient utilization of memory, we utilized a prime number based data transformation technique, in which values of individual transaction have been preserved. After removing null transactions and infrequent items, the resulting transformed dataset becomes denser compared to the original distributions. We tested our proposed algorithms in both real static *TDBs* and *DDSs*. Experimental results and performance analysis show that our approach is efficient and scalable to large dataset sizes.

Keywords: Big data, transactional databases, dynamic data streams, null transactions, prime number theory, data mining, Apache Spark, maximal frequent patterns.

1. Introduction

The term *data mining* refers to the non-trivial extraction of valid, implicit, potentially useful and ultimately understandable information in large databases with help of the emerging computing technologies. Among them, finding frequent patterns plays a significant role in every data mining task such as association analysis, market basket analysis (*MBA*), clustering, and classification [31, 42, 2, 15, 12]. Consequently, the problem of mining frequent patterns has been discussed widely in data mining research.

Apriori [2] is one of the first algorithms for mining frequent itemsets and association rules from transactional databases. It starts by identifying frequent 1-itemsets in the database and extending them to larger itemsets as long as those itemsets are frequent enough in the database. The frequent itemsets determined by Apriori can be used to determine association rules which highlight general trends in the database for market basket analysis. Apriori uses a breadth-first search strategy to count the support of itemsets and uses a candidate generation function which exploits the *downward closure property* of support. This property states that every subset of a frequent itemset is also frequent.

*Corresponding author: rezaul.karim@fit.fraunhofer.de

FP-growth [15] is an improvement of Apriori designed to eliminate some of the heavy bottlenecks in Apriori. The biggest advantage using FP-growth is the fact that the algorithm only needs to read the file twice, as opossed to Apriori which reads it once for every iteration. Secondly, FP-growth removes the need to calculate the pairs to be counted, which is very processing heavy, because it uses the *FP-tree*, which makes FP-growth much faster than Apriori.

Mining only for *Maximal* Frequent Patetterns (MFPs, i.e., patterns for which no 'larger' frequent pattern exists, see also definition 3) limits the number of frequent patterns to improve efficiency of business intelligence. Since the set of MFPs is much smaller than that of frequent patterns, extraction of MFPs will improve efficiency as MFPs also reveals customer's purchase rules [20] and help in MBA. Thus, developing an efficient maximal frequent pattern mining technique has been an important research direction in the area of data mining and knowledge discovery from data (*KDD*). We further discuss work in this area in the related work section below.

Today with the development of ubiquitous computing devices and the rapid computerization of business, the amount of data available for analysis has grown exponentially in many areas such as the retail industry, financial forecasting, decision support and intrusion detection. The ever increasing volume of the data leads to the era of big data [1] that described by the process of extracting actionable intelligence from disparate, and often times non-traditional data sources. These data sources may include structured data such as database, sensor, click-stream, and location data, as well as unstructured data like e-mail, web pages, social data, and images [1]. The extracted actionable data may be represented visually using graphs, but it is often distilled down to a structured format, which is later stored in databases for further manipulation[1]. In these big data sources, size is not the only concern but complexity arises in multiple dimensions.

Currently, existing data mining solutions (e.g., mining frequent pattern, mining maximal frequent patterns, MBA, association rule mining, etc.) are either desktop-based or traditional distributed system which can not meet these challenges. Fortunately, next generation big data processing engines such as *Apache Spark*[2] provides powerful libraries for big data processing and light-weight analytics mechanism to overcome data parallelism and scalability limitations [29, 23] in data mining from big datasets. Thanks to the caching mechanism that holds the previous computation result in memory, Spark outperforms the *Hadoop MapReduce*[3] framework [10] significantly. This is because Spark does not need to persist all the data into intermediate storage like disks for each round of parallel processing [1, 16, 11].

In this paper, we propose a novel and efficient approach for mining MFPs based on numerical methods with Spark libraries. To the best of our knowledge this is the first method that combines Spark framework with numerical methods. We have shown how to lift an existing single-node pipeline to a multi-node cluster pipeline within the Spark framework. As a result, our proposed tree structure and mining algorithms can be applied in parallel by benefiting from this solution. Particularly, our method extends and improves methods described in some early research papers [31, 21] to identify the complete set of MFPs. The key contributions in this paper can be summarized as follows:

- In order to overcome the limitations and drawbacks of existing approaches for finding MFPs, we proposed a simple but effective Spark based method to handle large transactional databases (TDBs) and dynamic

[1]See also: `http://www.scaledb.com/big-data-php.php`
[2]Apache Spark: `https://spark.apache.org/`
[3]Hadoop: `http://hadoop.apache.org/`

data streams (DDSs) to achieve massive parallelism and scalability.

- We designed a prime number based data transformation technique thereby making the computation in a faster way. In contrast to previous approaches (e.g. [31, 21]), we have not only used the properties of prime numbers but also provided formal proofs for them.
- Two new algorithms namely *TV* and *iTV* (algorithm 1 and 5) are developed for fetching big TDBs and DDSs respectively and create transaction value databases by utilizing prime number based data transformation technique in a static and an incremental setting.
- We provide a memory efficient and faster way to construct an efficient tree structure called *ASP-Tree* (algorithm 2). The advantage of the proposed algorithm compared to the FP-growth [15] like trees or conventional lattice structures is the small space requirement.
- We also developed two effective pruning techniques (algorithm 3 and definition 9) that maintain partial downward closure property [3] to reduce the search space and the number of candidate frequent itemsets.
- We proposed a new algorithm called **M**aximal **F**requent **P**attern with **A**pache **S**park (*MFPAS*, algorithm 4), which needs less effort and time for finding natural numbers that represent the MFPs from the search tree. The same algorithm is then capable of converting these numbers back into the original MFPs.
- We tested our proposed algorithms on real and synthetic datasets, by considering both highly dense as well as sparse datasets. These experiments and performance studies indicate the effectiveness of our approach.

In this paper, we use several terms interchangeably: itemsets and patterns; database and dataset; and support and frequency. We also used the following acronyms: Transaction Value (TV), minimum support threshold (*min_sup*), **M**aximal **F**requent **T**ransaction **V**alue (*MFTV*) and **A**pache **S**park-based **P**rime Tree (*ASP-Tree*).

We used the following structure in this paper: first, in section 2 we formaly introduce and define the problem of maximal frequent pattern mining, followed by an overview of earlier approaches in section 3. Then, before the introduction of our new scalable approach to frequent pattern mining in section 5, we introduce the Spark framework which is used as a runtime environment, in section 4. In section 6 we extend the approach to mining maximal frequent patterns on data streams. Section 7 contains the results of our evalution of both the static and dynamic mining approaches. In the last section, we summarize the paper, and provide an outlook on future research work opportunities.

2. Problem Statement

In this section, we will define the problem of mining maximal frequent patterns (MFPs) from a static dataset and dynamic data streams. We will also present necessary background knowledge about patterns, frequent patterns, maximal frequent patterns, and null transactions. We also introduce several notations and definition from our earlier work [21].

Definition 1. Given the items $i_1, i_2, ..., i_n$, which can occur, the set $I = \{i_1, i_2, ..., i_n\}$ is called the *domain* of the problem, where n is the number of distinct items.

For example, in a market basket analysis scenario, the domain is the set of all products which can occur in the baskets.

Definition 2. A *transactional database* T is a collection of *transactions* $t_1, t_2, \ldots, t_N$. Each transaction is a subset of the domain I. N is the number of transactions in the database.

Definition 3. Pattern, frequent pattern and maximal frequent pattern: any set $X \subseteq I$ is a pattern[4]. Given a transaction t (from a transactional database T), if $X \subseteq t$, then it is said that X occurs in t or inversely that t contains X. Support(X) denotes the number of transactions in T that contain X. Given a natural number $min\_sup \in [0, +\infty)$ then, if Support$(X) \geq min_sup$, we say that X is a *frequent pattern* in T. If X is a frequent pattern and no superset of X is frequent, then X is a *maximal frequent pattern* in T.

Table 1: An example of a static transactional database

Transaction ID	Transaction
1	A, B, C, D, F
2	A, B, C, E
3	B, C, D, E, F
4	A, C, D, E
5	C, D, F
6	G, H
7	D, E, F
8	D, E
9	C, D, F
10	C,F
11	A
12	A, C, D, E
13	C, E
14	C
15	F

Let us look at an example. Table 1 contains an example of a transactional database with 15 transactions. Each of these transactions has an ID and a set of items which are part of the transaction. For the sake of simplicity, we have not shown information like transaction time and customer information that are also usually stored in real life transactional databases.

Now, the number of occurrences of the patterns CD, DE and CDF are 6, 5 and 4 respectively. If we now assume min_sup to be 4, all of them are also *frequent* patterns. However, CD is not a maximal frequent pattern, since its super-pattern CDF is also a frequent pattern. If, however, min_sup would be 5, then CD becomes a MFP. In summary, we can state the following definition for mining maximal frequent patterns.

Definition 4. Given a domain I, a (non-empty) transactional database T, and a minimum support threshold $min\_sup \in [0, +\infty)$, the set of maximal frequent patterns MFP contains all patterns which are maximal and have a support of at least min_sup.

For the streaming case, we are interested in frequent patterns in the data from the first transaction which arrived until the last transaction which has been received. In order to keep this set of patterns up to date, we have to always maintain the set of maximal frequent patterns and update it upon arrival of new data. In practice, we will not receive these transactions one at a time, but rather in batches of new transactions (i.e., a collection of transactions). Hence, we can define the maximal frequent pattern set as follows:

Definition 5. Given a domain I, a collection of batches of transactions $\hat{T}$, and a minimum support threshold $min\_sup \in [0, +\infty)$, the set of maximal frequent patterns MFP for this collection of batches is the same as for T, where T is the collection of transactions obtained by taking each transaction out of each batch in $\hat{T}$.

[4]Note that a transaction is a pattern occurring in a transactional database.

3. Related Work

Since, a complete set of MFPs can derive all the frequent patterns efficiently, many algorithms for mining MFPs have been proposed [12, 20, 8, 32, 13, 38, 18]. Most of these algorithms are designed in a bottom-up breadth-first fashion similar to Apriori, which was presented in the introduction.

The MaxMiner algorithm [5] was one of the first attempts for mining MFPs. MaxMiner [5] extends the Apriori algorithm by inheriting the downward closure property. It builds a concept frame called Rymon's set enumeration tree [34], and uses a breadth-first traversal to traverse on the search space. While the fundamental limitations of Apriori algorithm with respect to pattern length remains, the super-set frequency pruning technique of MaxMiner reduces the search time drastically. However, many passes trough the original database are still needed to find all the MFPs.

Pincer-Search [28] combines both bottom-up and top-down searches to find the MFPs. The *two-way search approach* can speed up the overall MFPs mining process. todo[inline]MC in which cases? In general, if some MFPs are long and distributed in a disorderly manner, then the problem of discovering MFPs can be tough computationally.

MAFIA [8] was proposed for mining MFPs and uses the lexicographic subset tree which is based on a vertical bitmap representation for support counting. As a result the pruning mechanisms can be applied effectively for searching the itemset in the lattice structure. This idea of mining MFPs is based on earlier work [2, 5]. On the downside, it generates many insignificant patterns including candidate and closed frequent itemsets. In addition, the MAFIA algorithm also assumes that the entire database fits into main memory, but practically that might not be possible.

Similar to MaxMiner, GenMax [13], also uses Rymon's set enumeration technique [34]. It introduces *progressive focusing technique* and *diffset propagation* to perform fast support counting to maintain a set of local MFPs to reduce the cost of subset tests. Despite its ability to reduce candidate itemsets, it spends half of the time in maximality checking, which leads to a significant increase in runtime.

Flex [39] is a lexicographic tree designed in a vertical layout format to store pattern and a list of transaction ids (TIDs). Although, Flex is efficient in finding long and representative MFPs, it needs a considerable amount of memory to store the list of TIDs in memory with an enormous number of generated reoccurring patterns.

Recently, a canonical order tree or *CanTree* was proposed by Leung *et al.* [27, 26]. It is an efficient extension of the FP-tree [15] to capture the contents of TDBs with just one time database scan in a canonical order. Hence, there is no need for a search, it finds merge-able paths and swap tree nodes by frequency ordering. Therefore, once a CanTree is constructed, frequent patterns can be mined from the tree in a way similar to FP-growth. Although, the simplicity and lower cost of CanTree construction solved the weaknesses of the FP-growth based trees, it holds both the frequent and non-frequent items, causing significant space and memory overhead. Moreover, when TDBs are sparse, size of the tree becomes wide. Consequently, time for the mining process increases.

Another approach [18] proposes mining MFPs in two consecutive steps: i) compressing TDBs into a condensed data structure through dividing the attributes into an information matrix to avoid repeated, costly database scans, ii) than MFPs are generated utilizing the information matrix so that fewer candidate itemsets are generated. This approach can reduce both the I/O and CPU time. However, if the number of items in each group is n, it needs to allocate a matrix with size of 2^n x 2^n. This results in an exponential increase of memory.

Moreover, any matrix-based representation is inefficient with an $O(n^2)$ algorithmic complexity.

In our earlier work[21], we proposed a tree structure called PS_Tree and a mining algorithm called $MFPM$ that retrieves the complete set of MFPs from the PS_Tree. The algorithm uses a prime number-based data transformation technique that reduces the size of the original transactional databases. First, it removes infrequent items and transforms every transaction into a compact value called TV value (i.e. transform vector). Then it constructs the PS_Tree tree structure using those TV values. After that, a complete set of maximal TV values is generated from the tree. Finally, a complete set of MFPs is retrieved from the maximal TV values. Although this approach has better performance and more compaction compared to [31], it has some drawbacks:

- Our previous approach does not consider the overhead of null transactions. Therefore, the compaction rate is not satisfactory. On the other hand, our recent performance study shows that efficient handling of null transactions can achieve 10% to 20% more compaction rate towards 5% to 10% better memory usages and mining time.
- Following the previous literature [30, 37] it also uses the vertical layout representation technique for finding infrequent items, in which both TIDs and items need to be stored in main memory. This limitation results in elevated memory consumption.
- Similar to the PC_Tree [31] structure, it has to maintain two support counts (for the TV values in the tree), *local_count* and the *global_count* which results in a non-trivial space overhead. Interestingly, our current empirical study has shown that for the purpose of MFPs mining only the *global_count* is enough to construct and maintain the TV values in the tree structure.
- It assumes that the entire dataset can be loaded into main memory and therefore it is not scalable to increasing data sizes.
- It only considers the static transactional dataset for mining and will therefore not meet the requirements of real-time response for DDSs.

Until now we have summarized related literature regarding to mining MFPs from static TDBs. The rest of the section will focus on mining MFPs from dynamic data streams. In recent years various mining algorithms such as [14, 17, 24, 25] have been proposed to discover useful patterns from dynamic data streams (DDS). However, mining frequent patterns from DDSs is more challenging and complicated compared to traditional static mining [9]. The reason is that a currently observed infrequent pattern may become frequent in later batches. Therefore, special care and consideration needs to be taken before pruning any infrequent patterns in early stages. Otherwise, we may not be able to get the complete information such as frequencies of some useful patterns at later stages.

In addition, a dynamic data stream cannot be scanned multiple times because once the streams has been processed, it is lost completely [24, 25, 9]. Thus, we need to capture all the useful information before mining MFPs. Nevertheless, incorporation of big data in these approaches is not possible -i.e. not scalable. However, it should be an essential competency to handle massive dataset so that both the transactional databases and heterogeneous data streams can be used to achieve outstanding outcomes.

4. Spark for Big Data Processing

Three tier architectures have been widely used for relational database management system (RDBMS) over the decades. In this architecture the data processing is performed by the application server (client node). The data is then typically stored on the database server, and the processing cycle roughly goes as follows

- The application server sends a query or request to the database server to retrieve necessary data
- The data is sent to the application server which processes the received data
- the application server saves the resulting data to the database server after processing has completed.

The above processing cycle is a traditional data processing paradigm, whereby date is moved to the code. With very large datasets it becomes impossible to store all the data on a single database server (DB server). Besides, also more application servers are needed to deal with the increased processing power required to process the data. When the number of application servers and DB servers for storing and processing the data is increases, more data needs to be transferred back and forth across the network. Therefore, during the processing cycle, the network becomes a major bottleneck.

The Hadoop MapReduce framework has been used widely for the last few years to solve some of these problems. At its core, is the idea that data should not be moved around, but instead the code (the map and reduce functions) are moved to the noes where the data resides. However, Hadoop has some problems with I/O, algorithmic complexity, low-latency streaming jobs and its fully disk based operation [22]. Countering these problems, *Spark's* in-memory cluster computing framework allows user programs to load data into a clusters memory and query it repeatedly. This also makes it well-suited for machine learning algorithms. Spark caches the intermediate data into memory and provides the abstraction of *Resilient Distributed Datasets* (RDDs) [41], which can be used to overcome the above mentioned issues, achieving broad uptake in the recent past. For example, Spark has been used for handling Big Data for drug discovery [16], Big Data analytics [23], sequence alignment [43], network anomaly detection [36], predicitive machine learning on historical data [44] and sentiment analysis [6].

The Spark framework will run the code as follows:

- The program execution starts at a driver, which orchestrates the actual execution across many worker servers within the cluster.
- Data is no longer transferred to the driver program and the driver program works with data references rather than the data itself. These data references are identifiers to locate the corresponding data residing on the worker servers.
- The code is then moved from the driver program to the workers. There, the execution happens and data is modified at the worker servers whenever possible without leaving the machine.
- Finally, driver program requests the modified data which resides on worker nodes and only the results will be transferred to the driver program.

In line with the above vision, Spark has two significant parts: i) a programming model that creates a dependency graph, and ii) a runtime system which uses that graph to schedule work units on the cluster. The runtime system also distributes the code to the computing nodes and collects the final results. At the core of the Spark programming model are *RDDs* [40], which are abstractions of a dataset which is distributed over the

worker nodes. Operations in spark will transform these RDDs as if they are local variables, while the actual operations will be distributed over the workers.

Spark uses a lazy evaluation approach, meaning that no computation is happening until an outcome is requested (in Spark terms it is also said that an action is performed). This means that Spark can optimize the dependency graph to reduce the amount of data movement across nodes. This is crucial because there are two types of operations. Those which do not need any communication between nodes, like `map()` which transforms each element in the RDD in isolation or `filter()` which makes a selection of the elements based on a predicate. Others require nodes to communicate like, for example, `groupByKey()` which will group elements by a chosen key or `joining` two RDDs, which requires at least one of the RDDs to be cached or persisted. Choosing the order and actual implementation of these operations wisely leads to a reduction of the computation cost.

5. Proposed Approach for Static Mining Maximal Frequent Patterns

In this section, the pipeline of the proposed model will be described first. Then the MFPAS algorithm will be described with necessary mathematical details. Finally, a step-by-step example will be elaborated to make our approach clearer.

5.1. Processing Pipeline

We assume that the transaction databases are stored in a distributed way on the Hadoop distributed file system (*HDFS*) in a cluster of worker nodes servers, also called a DB server. Such server is a computing node with large storage and main memory which can be flexibly assigned tasks. The driver PC is also a computing node that mainly controls the overall computation. It needs to have a main memory large enough for processing and holding Spark code fragments to be sent across the computing nodes in the cluster. In our case, the Spark code consists of DB server id, the minimum support threshold and algorithms for encoding, tree construction, mining, and decoding. Each worker takes the input codes sent from the driver machine and starts the local computation. Figure 1 shows a high-level schematic diagram of the pipeline of the proposed approach for mining MFPs from the large-scale static TDBs.

From an environment variable (i.e. *SparkSession*), we create some initial data reference or RDD objects. Then we transform the initial RDDs and create more RDD objects on the DB server. At first, the dataset is read and infrequent items and null transactions are filtered using narrow and wide transformations (i.e. flatMap, mapToPair, reduceByKey, sortbyKey etc.). This way, the filter join RDD operation provides a data segment without null transactions and infrequent items. The RDD objects are then materialized through an action, which dumps the RDD into the DB server. Spark's inter RDD join operation allows us to combine the contents of multiple RDDs into a single computing node [40]. In summary, we follow below steps before getting the filtered dataset:

- Enable distributed processing model and the cluster manager (i.e. *Apache Mesos*)
- Create *Spark Session* and instantiate all the necessary parameters
- Create `JavaRDD` and read the transactions and perform necessary partitioning
- Perform `flatMap` over the String to split the transactions into single items
- Perform `mapToPair` to find the key/value pairs of the items

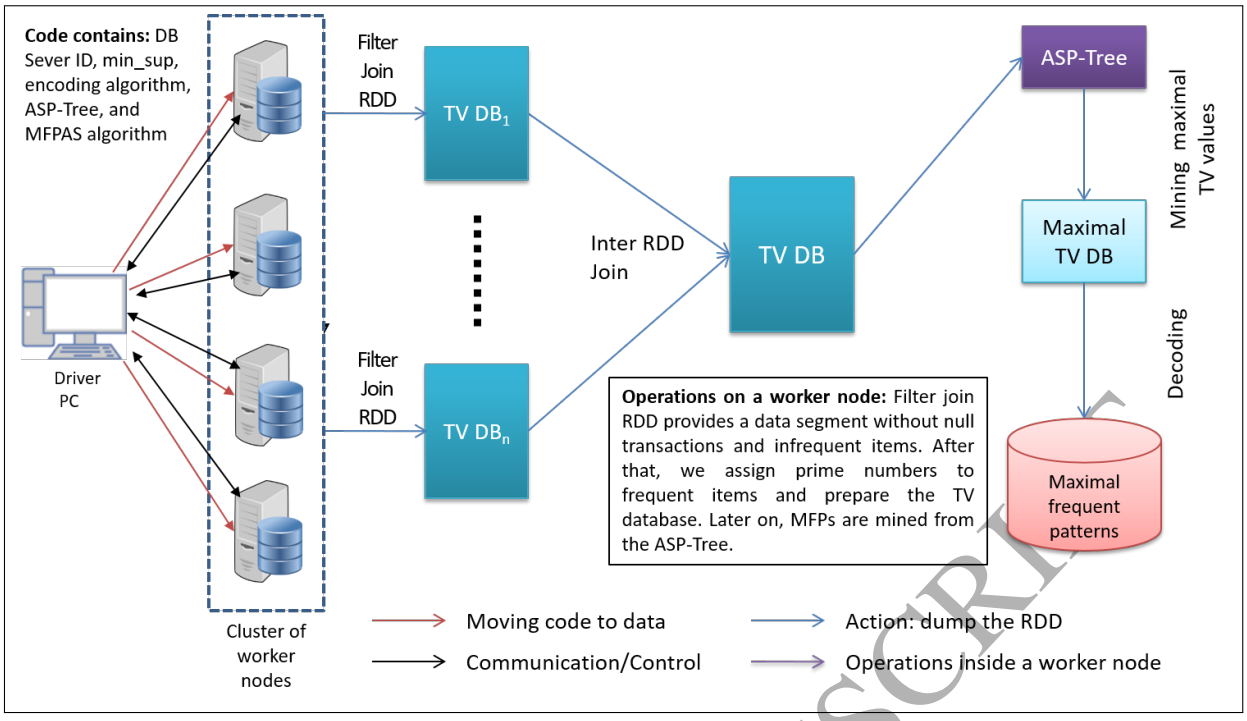


Figure 1: Proposed pipeline for mining maximal frequent patterns from static transactional databases

- Perform `reduceByKey` to find the support count of the items
- Perform `filter` to remove the null transactions (see section 5.2) and infrequent 1-itemsets
- Perform `sortByKey` to assign the primes against individual frequent 1-itemsets
- Materialize an inter RDD join action to save the filtered dataset on the DB servers. We call these transactions *filtered* transactions

When we have the filtered transactions, we apply our encoding algorithm to create the TV database DB^* through prime number assignment. As a first step, we need to map each of the unique items to a different prime number. As we will see later, it is useful to map items with more support to smaller prime numbers. Then, the TV of each filtered transaction will be computed as the multiplication of the prime numbers assigned to its items.

To do this on Spark an outer or inner join operation is performed. This way, we get a combined TV dataset with all the TV values (see more in section 4). In summary, we listed below tasks for creating the TV values:

- Generate a unique prime number for each unique frequent 1-itemset. If we have n frequent 1-itemsets, we will generate the first n prime numbers.
- Read the filtered transactions
- Perform a map operation, mapping each frequent 1-itemset to its associated prime number
  - assign the smallest prime to the item having the highest support count
- Perform multiplication operations to get the TV values for each transaction (see section 5.3)
- Materialize an action inter RDD join operation to save the TV values on a DB server or split across nodes if it does not have enough storage

5.2. Screenings of Null Transactions

A transaction that does not contain any itemsets being examined is a null transaction [20]. In our case, that means transactions which only contain one item. From a data mining perspective, null transactions do not give any information for association rule mining that can be further used in market basket analysis [20, 33, 19]. The reason is that these single itemset will not give any information about association or correlation between a pair of items, for the simple reason that they are not associated with any itemset in that particular transaction.

Hence, we use the following definition:

Definition 6. A *null transaction* is a transaction which does not contain any frequent item.

For the example in table 1, assuming the minimum support is 2, our proposed approach considers transactions with ID 11, 14, and 15 as null transactions, since they contain just a one itemset which is not frequent. This is as opposed to other tree based approaches like FP and CanTree which consider all the transactions.

Typically, the total number of null-transactions outweighs the number of individual purchases in a real retail setting. For example, suppose an electronic shop has 100 transactions including 10 insignificant null transactions. Then, the *FP-tree*, *PC_Tree* or *CanTree* algorithms will scan all the 100 transactions while, our proposed approach first reduces the amount to 90 by considering only the significant transactions. This way, we can save precious computation time for performing mining operations and also obtain a better compaction rate.

5.3. Database Transformation and Decoding Techniques

In a section above we mentioned that each unique item is mapped to a unique prime number. To do this, we need to make a small modification to our definition of transaction, as follows. We first assume that a total order $<$ has been defined on the domain I, i.e., that for each two distinct elements i_k and i_l of I either $i_k < i_l$ or $i_l < i_k$. Then, a transaction can be transformed into an ordered list of items without duplicates. We will henceforth call this ordered list the transaction. Now, using these transactions, we can continue to define our procedure.

Definition 7 ($\mathcal{P}$)**.** $\mathcal{P}$ is a function which maps unique items to unique prime numbers. This function is a bijection and the inverse mapping will be written as $\mathcal{P}^{-1}$.

In prime number theory, it is well known that any positive integer N can be expressed as a *unique* product of prime numbers. This is known as the *Unique Factorization Theorem (UFT)*. Therefore, $N = P_1^{m1} * P_2^{m2} * .. * P_r^{mr}$, where P_i are prime numbers and $P_1 < P_2 < \cdots < P_r$ and m_i are the positive integers [35, 4]. Now, we will use the earlier defined bijection and the preserve the features of this theorem to convert each transaction into a transaction value (TV) as follows:

Definition 8 (transaction value TV)**.** Given a (filtered) transaction $t = i_1, i_2, \ldots i_n$, (where i_j items of the transaction) the transaction value (TV) is computed as $\mathcal{P}(i_1) * \mathcal{P}(i_2) * \cdots * \mathcal{P}(i_k)$.

This definition means that TV is like a prime factorization without duplicate items, (i.e., the exponents m_i will always be 1). This is because in a transaction there are no duplicate items ($i_j = i_k \iff j = k$). Table 2 shows an example of this transformation and algorithm 1 shows the database transformation technique. In table 2, we eliminate the infrequent item -i.e. B for reducing search space similar to FP-tree [15]. For

Table 2: Database transformation using prime number

Items	Mapping to Prime	Transformation	TV
A, C, D, F	2, 3, 5, 11	2 x 3 x 5 x 11	330
A, C, E	2, 3, 7	2 x 3 x 7	42
C, D, E, F	3, 5, 7, 11	3 x 5 x 7 x 11	1155
A, C, D, E	2, 3, 5, 7	2 x 3 x 5 x 7	210
C, D, F	3, 5, 11	3 x 5 x 11	165
D, E, F	5, 7, 11	5 x 7 x 11	385
D, E	5, 7	5 x 7	35
C, D, F	3, 5, 11	3 x 5 x 11	165
C, F	3, 11	3 x 11	33
A, C, D, E	2, 3, 5, 7	2 x 3 x 5 x 7	210
C, E	3, 7	3 x 7	21

Algorithm 1: TV transformation algorithm

Input : DB server ID, *min_sup*; newline indicates new transactions
Output : TV database in which transactions are transformed into transaction values.
Data : *ApacheSparkAnalytics* reads TDBs on HDFS and creates RDDs, then it filters infrequent items and null transactions. Finally, it joins RDDs using push down predicate approach.

```
1 Procedure ApacheSparkAnalytics(DB_i, min_sup)
2     String[ ] items ← line.split(" ")
3     TV ← 1
4     SparkSession with master URL, driver host IP, executor memory, driver memory
5     for (String item : items) do
6         TV ← TV * P(item)
7     end
8     TV.saveAsTextFile(filepointer.getAbsolutePath())
9 EndProcedure
```

each transaction, the TV is obtained by multiplying the prime numbers produced by $\mathcal{P}$ For example, the transformation of transaction 1 is $\{2, 3, 5, 11\}$ and its TV becomes $2 * 3 * 5 * 11 = 330$.

Now we describe some interesting and useful properties of prime numbers.

Property 1. The largest prime representable as a 32-bit integer is 214,748,3647 and there are total 105,097,565 primes between 1 and 214,748,3647. Hence, as long as we do not have more as that number of items, the function $\mathcal{P}$ can be implemented using just 32 bit integers. The function TV, however, is needs to multiply (potentially many) of these numbers. This will lead to overflows (even when using 64 bit numbers) and hence, in our implementation, we used `java.math.BigInteger` objects to represent these numbers.

As we mentioned above, we improved algorithm such that the more frequent the 1-itemset, the smaller the prime which will be assigned. The heuristic is that having low numbers for more frequent items will, on average, reduce the size (i.e., memory footprint) of the computed products.

Property 2. If two transactions are different (i.e., they have different items), then their TVs are also different. This is the same as saying that the TV computation is also a bijection.

This can be shown easily by contradiction. Assume there are be two transactions with different items and the same TV value. Then, as the TV is computed from a set of prime numbers determined by the items, there must be two different sets of prime numbers resulting in the same product, which contradicts with the UPF theorem.

For example, in table 1, both transaction 5 and 9 have items (C, D, F). Now, if we assign C=2, D=3, F=5, then the TV will be the same (i.e., 30). We can now be certain that if we get any TV value of 30, also that

transactions will have the same items.

Property 3. Let m be the greatest common divisor (GCD) of the TVs of two transactions, t_1 and t_2. Now, if $m > 1$, then t_1 and t_2 have common items, namely those which are mapped to the factors of the prime factorization of m.

Since $m > 1$, it can be factorized into primes, i.e., $m = P_1 * P_2 * \cdots * P_k$ where $k \geq 1$. Because m is a common divisor of both TVs, $t1 = P_1 * \cdots * P_k * Q_i * \cdots * Q_j$ where $0 \leq i \leq j$ and $t2 = P_1 * \cdots * P_k * R_m * \cdots * R_p$ where $0 \leq m \leq p$. Hence, using the same contradiction strategy used above, we find that both transactions have the items corresponding to the prime numbers $P_1, P_2, \ldots, P_k$. Then, because m is the *greatest* common divisor, we know that there are no common factors (i.e., items) in the remaining parts of the TVs ($Q_i * \cdots * Q_j$ and $R_m * \cdots * R_n$). If there would be common factors, with product l, then $l * m$ would have been the GCD.

Note that the previous property is a special case of this one, where $m = TV_1 = TV_2$

For example, suppose the TV value of transaction t_1 is 30 and the TV value of transaction t_2 is 42. Then the prime factorization would be (2 * 3 * 5) and (2 * 3 * 7) respectively. Now the GCD of 30 and 42 is 6 (factorized into 2 * 3).

Property 4. Let TV_1 and TV_2 be the TVs of transactions t_1 and t_2, respectively. If $\exists k \in \mathbb{N}_{>1}$ with $TV_1 = k * TV_2$, then t_1 is a superpattern of t_2.

This can be shown using the previous property. If $TV_1 = k * TV_2$, then the GCD m of the TVs is TV_2. Now, this means that the the number m and p from the proof of the previous property must be zero. Hence, all factors of TV_2 are also factors of TV_1, meaning that all items in t_2 are also in t_1, and hence t_1 is a superpattern of t_2. Note that if we allow $k = 0$, then the transactions could also be equal. Since this is not allowed, there are some factors in TV_1 which are not in TV_2 and hence t_1 is a strict superpattern of t_2.

For example, consider two transactions (C, D, F) and (C, F), with TV values 165 and 33 (for prime mapping $C = 3, D = 5, F = 11$). From property 3, it can be found that the common items were mapped to 3 and 11, mapping backwards to C and F. Besides, 165 can be divided by 33 and consequently (C, D, F) is a superpattern of (C, F) (alternatively, (C, F) is a sub-pattern of (C, D, F)). If we would consider the set theory then, the sub-pattern and super-pattern relationships also proved.

As depicted in fig. 1, we transform our original transactional database in table 1 into DB^* (4th column of table 2), which has transformed transaction values, since this DB^* has same information with original TDB.

5.4. The ASP-Tree Construction Algorithm

After mapping the transactions to TVs, we insert all the TVs into a data structure, which we call ASP-Tree This structure includes a root and nodes, representing the TVs. Despite its name this structure is not a real tree as it allows nodes to have multiple parents. However, because it otherwise looks like a tree (having a root, and representing a partial order), we will still continue calling it a tree.

The nodes in the tree contain two values. The first one, which we call *value* is the TV of the pattern (frequent itemset) this node represents. This value never changes. The second one is a variable called *global_count* and is used to register the support of the pattern. The root of the tree is (which is only implicitly represented in the implementation) is initially included in the otherwise empty structure. The value of the root is the TV of the frequent itemset containing all possible items and an imaginary extra item which never occurs, i.e., if there are n unique items $i_1, i_2, \ldots, i_n$, it will be the value $\Pi_{j=1}^{j=n+1} \mathcal{P}(i_j)$. The *global_count* for the root is set to zero. Inside the data structure the following invariants are maintained:

- Each node N in the tree is a direct child of all nodes which a value that is a multiple of the value of N.
- The global_count of each node N is the total support for the value of the node represents.

Algorithm 2 is the pseudo-code for the ASP-Tree construction algorithm. The tree construction operation mainly consists of the insertion procedure and re-ordering of the TV values. Insertion of a value TV into the structure is based on the following rules:

- If there is another node with the same TV, increase the global_count of that node and any of its possible child nodes with 1.
- Otherwise,
  - For all nodes of which TV is a divisor, add a link from that node to a newly created node with value set to TV. In practise, a link from the root to each node is not needed, so that link is only explicitly added in case there are no other links to the new node. The value of global_count is set to the sum of the global_count of all these parent nodes plus one. This ensures that the node global_count will include all existing support and the support of the newly added TV.
  - And, if there is a node with value TV_{other} and TV_{other} is a divisor of TV, then these nodes will be reordered, i.e., the new node becomes the parent node of the node with value TV_{other}. The global_count of the node with value TV_{other} is incremented with 1.

5.5. The MFPAS Algorithm

The MFPAS (algorithm 4) traverses the ASP-Tree created according to the previous subsection to find the complete set of maximal TVs to in turn generate maximal frequent itemsets after decoding.

We perform first a bottom-up traversal in which we prune the tree and create an initial candidate list. Then, we reduce that candidate list to obtain a first part of the maximum TV values. Finally, we perform a bottom-up traversal in the pruned tree to extract final maximum TV values. After having obtained all the maximum TV values, we decode them to obtain the maximal frequent patterns. During the process, there is no need to scan the original database because all the information about items is stored in the tree.

The first bottom-up traversal starts from each leaf node in turns (if during the pruning new leafs appear, they are also considered, until all leafs have been considered at least once). If the leaf has a global_count smaller as *min_sup*, it is ignored and stays in the tree. If the leaf node has a global_count larger or equal to *min_sup*, then it might represent a maximal frequent pattern. However, if any of its parents also has sufficient support, then the pattern of the leaf was certainly not maximal and the leaf is pruned from the tree without any further action (the parent(s) will be considered at a later iteration). If none of the parents has sufficient support, then the TV is saved to the initial candidate list L_0. Ignoring leafs of which a parent has sufficient support is valid because of the partial downward closure property [3] and reduces the search space and amount of candidate itemsets (refer algorithm 3 for details on the pruning technique). Then, to reduce the list L_0 we do the following:

Definition 9 (*MFTV* list pruning)**.** If a TV value (i.e. Frequent TV value) of a node n_r is included in the candidate list L_0 and if it is a divisor of any other TV value in L_0 then n_r will be excluded from the list L_0. This results a new list of Maximal Frequent TV values, which we call L_1 (refer section 5.6 for an example).

Algorithm 2: ASP-Tree construction algorithm

Input : TV database and *min_sup*
Output : All the tree nodes containing TV values and their global_count
Data : The nodes containing global count and links between related nodes.

```
1  Procedure ApacheSparkTree(TV_i)
2      Root ← NULL
3      min ← 1000000
4      cntRDD ← countedRDD.collect()
5      nodeList ← newArrayList()
6      for (Tuple2 tuple : cntRDD) do
7          ASP-Tree node ← newASP − Tree(Integer)
8          ASP-Tree treeNode ← makeTreeNode(node, nodeList)
9          treeNode.setGlobal(treeNode.getGlobal() + treeNode.getCount())
10         nodeList.add(treeNode)
11     end
12     for (ASP-Tree node : nodeList) do
13         value ← node.getValue()
14         Root ← TV_1
15         i ← i + 1
16         if ((TV_i ≥ Root) && (value % treeNodeValue == 0) ) then
17             Root ← TV_i
18             treeNode.addChild(treeNode)
19             node.addChild(treeNode)
20             i ← i + 1
21             global_count ← global_count + 1
22         else if (min ≥ node.getValue()) then
23             min ← node.getValue()
24             minNode ← node
25         else if (value / treeNodeValue! = 0) then
26             newNode ←newASP − Tree(newNode)
27             newNode.addChild(value)
28             node.addChild(newNode)
29             i ← i + 1
30             global_count ← global_count + 1
31         else if (value.exists()) then
32             reorder()
33             value.addParent(value)
34             i ← i + 1
35             global_count ← global_count + 1
36         else
37             treeNode.setMainParent(minNode)
38         end
39     end
40     return treeNode
41 EndProcedure
```

Now, to mine the rest of the candidate maximal TV values from the pruned ASP-Tree, a second traversing is required and carried out from the root to each branch as top-down fashion. To do that, we take each branch as a new entry to an array list and finds the set of TV values for a given node. To be more specific, taking the first node of the first branch and the first, second, third (and so on) node of the second branch as the array list, our algorithm finds the *GCD* (property 2) of these nodes. Then, these greatest common divisor values are included in the list L_0 (while maintaining the list limited by applying definition 9 after each insertion).

Algorithm 3: Pruning algorithm

```
1  Procedure TreePruning(nodeList, min_sup)
2      tn, len, n ← 0
3      nodeList ← newArrayList()
4      tampNode ← newArrayList()
5      L ← newHashSet()
6      for (index = nodeList.size() − 1; index >= 0; index − −) do
7          len ← tampNode.size()
8          if (tn.getParentList().size()!= 0 && !tn.isTaken()) then
9              traverse(tn)
10             tn ← nodeList.get(index)
11         else if ((len! = 0) && (n.getValue()! = 1)) then
12             n ← tampNode.get(len − 1)
13             L.add(n)
14             tempNode.get(n).setParentList(null)
15             n.setTaken(true)
16             tempNode.clear()
17         else
18             TN ← tempNode
19         end
20         for (node : nodeList) do
21             if (len! = 0) && (n.getValue()! = 1) then
22                 for (i = 0; i ≤ node.getGlobal(); i++) do
23                     finalList.add(node);
24                 end
25             else
26                 finalList.add(TN)
27             end
28         end
29     end
30     return finalList
31 EndProcedure
```

Finally, the decoder factorizes these resulting maximal TV values into the product of primes, and the original maximal frequent patterns are obtained.

5.6. An example of Static Mining

Suppose, *min_sup* is set to be 4 for the database shown in table 1. Items B, G and H items are infrequent, so we removed them from the database based on our algorithm and support threshold. Moreover, transactions 11, 14 and 15 are null transactions and consequently get removed from the database *DB* too. The filtered database is shown in table 2 without infrequent items and null transactions. Now, before getting the TV database, we perform several transformations and actions on the original datasets with Spark and prime number theory as follows:

- The `flatMap()` method gives the list of frequent and infrequent items in the datasets over the DB server -e.g. [A, B, C, D, E, F]
- The `mapToPair()` gives all the items along with their i-itemsets count - e.g. [(A,1), (B, 1), (C, 1), (D, 1), (E, 1), (F, 1), (C, 1), (D, 1), (E, 1), (E,1)]

Algorithm 4: MFPAS algorithm

```
Input  : Pruned ASP-Tree and min_sup
Output : Maximal TV values and MFPs with support count
1  Procedure StaticMiningMTV(NodeList, min_sup)
2      for (ASP-Tree node : L) do
3          for (Iterator it.hasNext()) do
4              if (it.next() == node.getValue() then
5                  PrimeList ←primeFactors(node.getValue()
6              else
7                  PrimeList ←primeFactors(it.next()
8              end
9          end
10     end
11     return PrimeList
12 EndProcedure
13 Procedure StaticMiningMFP(PrimeList)
14     reverseList ← null
15     Call − − > StaticMiningMTV(NodeList, min_sup)
16     for (ASP-Tree node : L) do
17         for (Iterator it.hasNext()) do
18             if (it.next() == node.getValue() then
19                 for (Integer v: primeFactors(node.getValue())) do
20                     reverseList.append(reverseUniqueKey.get(v))
21                 end
22             else
23                 reverseList.append(reverseUniqueKey.get(it.next()))
24             end
25         end
26     end
27     return reverseList
28 EndProcedure
```

- The `reduceByKey()` gives the items along with their aggregated support i.e. [(A, 2), (B, 3), (C, 2), (D, 4)] etc.
- The `sortByKey()` gives the support of individual support counts of frequent 1-itemsets
- Then we remove infrequent itmes from the datasets based on the support threshold
- Applying further `map()` operation provides the filtered datasets
- We generate primes and assign unique primes to each frequent 1-itemset
- We then perform another map and generate the `TV database`.

Table 2 shows the filtered database consisting of individual frequent 1-itemsets. Suppose, following prime number assignment is done: A=2, C=3, D=5, E=7, and F=11. Table 2 shows the prime mapping and construction of the corresponding vector table DB^* in the 4th column. Since, we have the TV database DB^*, we now describe the creation and insertion procedure into the proposed ASP-Tree structure based on above mentioned rules. First, we take TV value 330 and 42, where, 330 is not divisible by 42 so these two values will be inserted as two new nodes, where, 1155 is also inserted as a unique node. TV 210 is the dividend of 42 and $42 < 210$, so we re-order them -i.e. 210 becomes the parent node and 42 would be the descendant node. Then we also update the links accordingly.

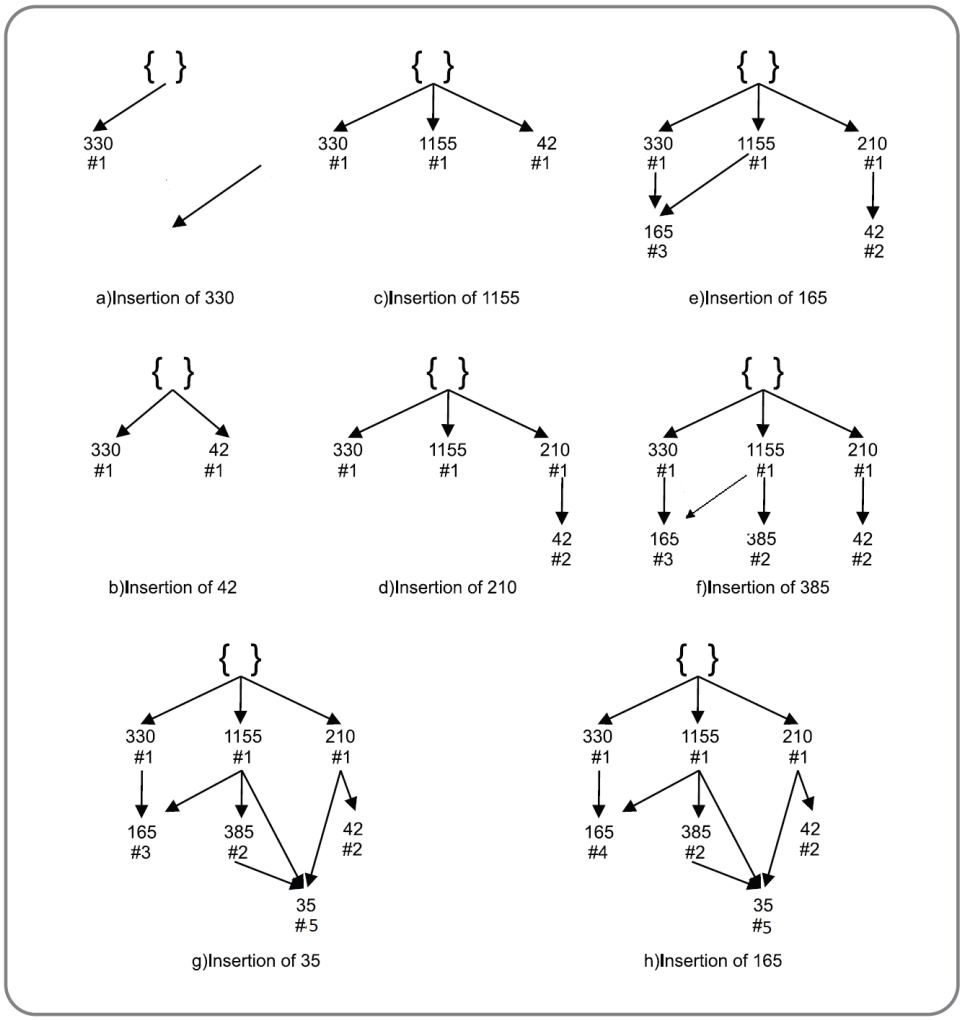


Figure 2: Step by Step ASP-Tree construction procedure: TV value (a) - (f)

TV 165 is a divisor of both 330 and 1155 but $330 < 1155$, so 165 is inserted as the descendant node of 330 and the *global_count* of 165 will be 3 (i.e. for 165, 330 and 1155) and we update the link accordingly. 385 is the divisor of 1155 so it will be the descendant node of 1155 and *global_count* will be 2 as well. 35 is the divisor of 210, 1155 and 385 at the same time but among them, 210 is the smallest, so it is inserted as the descendant node of 210 and the link and global count will be updated accordingly. TV 165 is already inserted in the tree, so we just increase the *global_count* of node 165. At the same time, we update the *global_count* of all the divisors of 165 as well. TV 33 is inserted as the descendant of 165 although it has another two dividends -i.e. 330 and 1155. The same rule is applied to next TV 210.

Finally, TV 21 is inserted as a descendant node of 42 (being the smallest one among three dividends -i.e. 1155, 42 and 210). Refer fig. 2 and 3 for a step-by-step TV insertion procedure. Figure 4 on the other hand, shows the complete pruned tree based on pruning technique (see algorithm 3).

To mining the maximal TV values from the tree in fig. 3 (k), we traverse from root to each branch of the PS_Tree. We maintain a list of frequent TV values called L_0 and traversing the PS_Tree we get the following list of TVs that satisfy minimum support threshold, $L_0 = \{165, 35, 33, 21\}$. Although 33 satisfies the minimum support threshold but FTV value 33 is the divisor of 165; so 33 to be excluded from the list L_0 according to definition 5. Therefore, the new list of Maximal Frequent TV values $L_1 = \{165, 35, 21\}$. Fig. 4 shows the

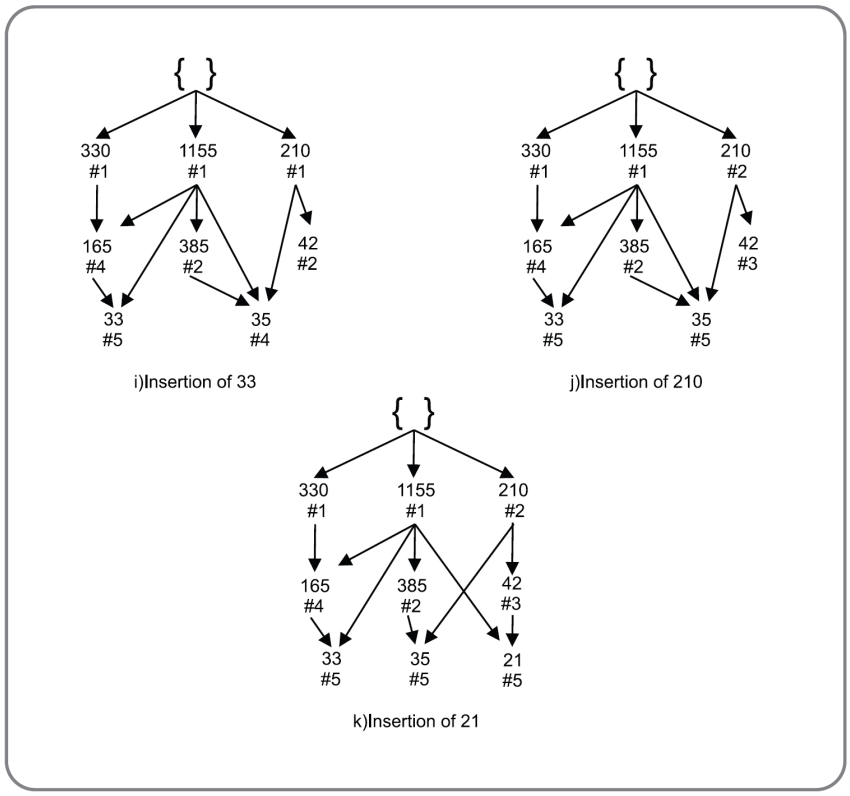


Figure 3: Step by Step ASP-Tree construction procedure TV value (i) - (k)

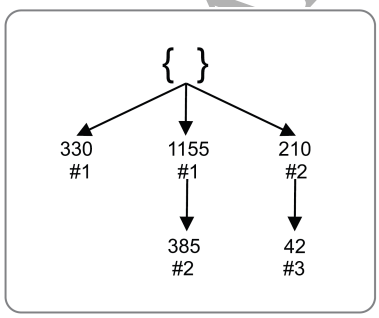


Figure 4: Pruned *PS_Tree* with *global_count* excluding items of the list

Table 3: Candidate *GCD* list L of Maximal TV values

Paths	**TV List**	**GCD Value**
P_1	1155, 210, 210, 42	21
P_2	385, 385, 1155, 210	35
P_3	385, 1155, 210, 210	35
P_4	330, 1155, 385, 385	55
P_5	330, 210, 210, 42	6
P_6	210, 210, 42, 42	6
P_7	42, 42, 42, 210	6
P_8	385, 385, 1155, 210	35

complete pruned tree without the items that are already in the list L_1 with the *global_count*. Taking each branch as a new entry to an array list we find the set of TVs calculated previously for a given node. Now we find the products of the greatest common divisors of the candidate paths in table 3, where, table 4 shows the resulting value calculations. The *GCD* value 21 and 35 are already included in the list L_1, so these values will not be included in the MFTV list. The *GCD* value 6 is unique so has been included in the list L_f. Therefore,

the final list will be $L_f = \{165, 35, 21, 6\}$.

Once we have the final list of maximal TV values, our hashing based decoding algorithm converts them back into corresponding primes factors. Table 4 shows the frequent patterns retrieval technique and resulting values. After the final calculation, we get four MFPs as follows: $\{C, E\}$, $\{D, E\}$, $\{C, D, F\}$ and $\{A, C\}$ based on table 3 and 4.

Table 4: Retrieving MFPs from the maximal TV values

Maximal TV	Prime Factorization	MFP	Support
165	3*5*11	C, D, F	4
35	5*7	D, E	5
21	3*7	C, E	5
6	2*3	A, C	4

6. Incremental Mining Maximal Frequent Patterns from Dynamic Data Streams

In the previous section, we showed the mining process of MFPs from static TDBs. The ASP-Tree and $MFPAS$ algorithm work well when handling with static databases. However, in some situations there is a need to deal with dynamic streaming data. Table 5 shows a transactional database, where live streaming transactions are coming as batches. In this database, although BC and BE are frequent patterns BCE, AC and CD are MFPs by considering $batch_1$ and $batch_2$ if the minimum support count is 2.

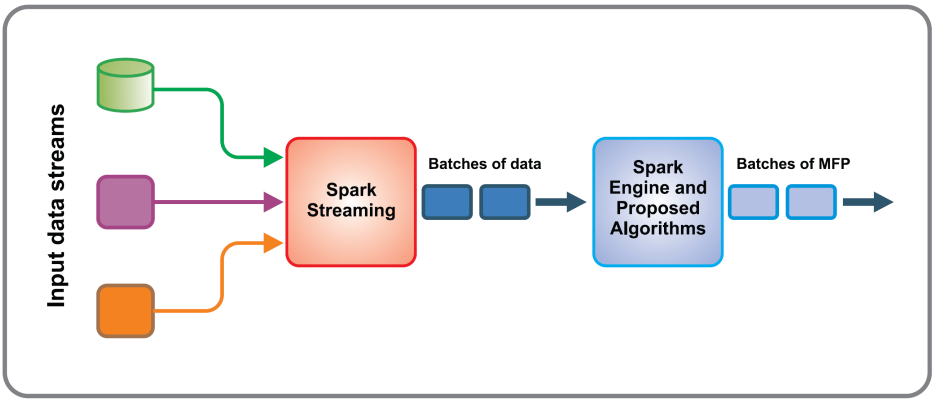


Figure 5: Extraction of transactional data from dynamic data streams with Apache Spark

The Spark Streaming library receives live input data streams and divides the data into batches, which are then processed by the Spark engine and our proposed algorithms to generate the final stream of results in batches. Spark Streaming, which provides a high-level abstraction called a discretized stream or DStream, deals with continuous data streams.

In this section, we propose an adapted algorithm for handling batches of data in an incremental manner using the Spark Streaming library. The corresponding algorithm is named iTV, which helps to materialize the DB^* database to be stored on the HDFS in a DB server from the streaming data. Then we call ASP-Tree and $MFPAS$ algorithm for mining MFPs. When using the stream mining for processing data streams, transactions in each batch (regardless of whether they are historical or recent data) are treated equally. As such, all batches (irrespective of whether they are old or recent) are treated equal weights. Fig. 5 shows the high-level view of extraction of transactional data from DDSs with Spark and related technologies.

Algorithm 5: iTV transformation algorithm

Input : Initial input: data streams, second input: DB server ID and the *min_sup*
Output : TV database with transactions are transformed into transaction values
Data : *iDBConstruction* converts incoming streams into batches, *iApacheSparkAnalytics* materializes RDDs using push down predicate approach.

Procedure *iDBConstruction()*
  JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(time))
  jssc.checkPoint ← `hdfs://transaction`
  streamingContext.start()
  JavaDStream[Status] transactions = createStream(jssc)
  JavaRDD[Status] outputRDD = transactions.repartition(number_of_partition)
  B_i ← outputRDD.saveAsTextFile(transactions)
  B ← $B_1 + B_2 + ... + B_i$
  streamingContext.awaitTermination()
  DB ← materialized version of B
  DB.saveAsTextFile(filepointer.getAbsolutePath())
  return DB
EndProcedure

Algorithm 6: Overal algorithm

Procedure *ApacheSparkAnalytics()*
  `DB` ← iDBConstruction()
  `TV_database` ← ApacheSparkAnalytics(DB, min_sup)
  return `TV_database`
EndProcedure

Table 5: A data stream (i.e., batches of transactions)

Transaction with batches		
Batch ID	Transaction ID	Transaction
B1	1	A, C, D
	2	B, C, E
	3	A, C
B2	4	C, D
	5	B, E
	6	B, C, E

Internally, a DStream, which is represented by a continuous series of RDDs, is Sparks abstraction of an immutable, distributed dataset [41, 3] and each RDD in a DStream contains data from a certain interval. Any materialization or operation applied on a DStream translates to operations on the underlying RDDs. The Spark engine computes these radical RDD transformations. The DStream operations cover most of these details and provide the developer with higher-level API for convenience[5]. In a nutshell, the whole procedure can be summarized as follows:

- Create SparkSession by specifying the AppName, master with URL, local IP of the computing nodes, driver host IP, executor memory, and driver memory etc.
- Create a JavaStreamingContext from the existing SparkSession object
- Define the input sources by creating input DStreams

[5]Spark Structured Streaming `https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html`

- Define the streaming computations by applying transformation and output operations to DStreams
- Start receiving data as batch and processing it using streamingContext.start()
- Wait for processing to be stopped: streamingContext.awaitTermination() OR streamingContext.stop()
- Materzlize the input data on the DB server for further operation
- After getting the data, convert it into Spark RDDs for computation
- Call *iTV* algorithm to create the TV database

Refer algorithm 5 for the TV database creation from dynamic data streams as incremental manner. Once the TV database is constructed, we call the ASP-Tree algorithm to create the tree structure. After that, we call the pruning algorithm to prune unnecessary nodes from the tree structure. Then, the *MFPAS* algorithm along with the pruning techniques are applied to traverse the pruned tree for mining MFPs.

7. Experiment Results

7.1. Programming Environment and Datasets

The proposed model is implemented on Spark, hence making our algorithms scalable and computationally faster. All programs were written in Java and Spark library. The operating system of each computing node (DB server) was Ubuntu 16.04 64-bit stable version, and the version of Spark was 2.2.0. During the reporting period, we used 30 computing nodes (with 29 as DB server and one work as Driver program). Communication between nodes was carried out using standard Gigabit network connection. The *SOMA* platform[6] is used for the management of our distributed analytic jobs which also helps in big data processing environment required for job executions.

We used `BigInteger`(whenever necessary -e.g. number of items in a transaction is high) of Java programming language to manipulate the Map, HashSet, and List for handling large prime numbers as well as large multiplication numbers to handle up to 1000 transactional items.

Table 6: Characteristics of the datasets; *TL= transaction length

Dataset	No. of transactions	No. of items	maxTL	avgTL	Density
T10I4D5000K	5000,000	10,000	29	10.10	Sparse
T20I4D1000K	1000,000	10,000	42	19.81	Dense
Mushroom	8,124	119	23	23.00	Dense
Retail	88,163	41,270	18	13	Sparse

We have conducted a set of experiments to validate our hypothesis. In these experiments we used synthetic sparse datasets *T10I4D5000K* and *T20I4D1000K* generated by the (*IBM Quest Synthetic Data Generator*[7]); a real dense datasets generated by *Mushroom*[8]; a real sparse retail dataset donated by *Tom Brijs*[7]; and also the (anonymous) retail market basket data from an anonymous Belgian retail store. The Belgian retail store data contains total 5,133 customers who have purchased at least one product in the supermarket during the data collection period. In this dataset, the average number of distinct items (i.e. different products) purchased per shopping visit equals 13, and most customers buy between 7 and 11 items. The number of transactions,

[6] `https://www.insight-centre.org/content/soma-linked-data-platform`
[7] `http://sourceforge.net/projects/ibmquestdatagen/`
[8] `https://archive.ics.uci.edu/ml/datasets/Mushroom`

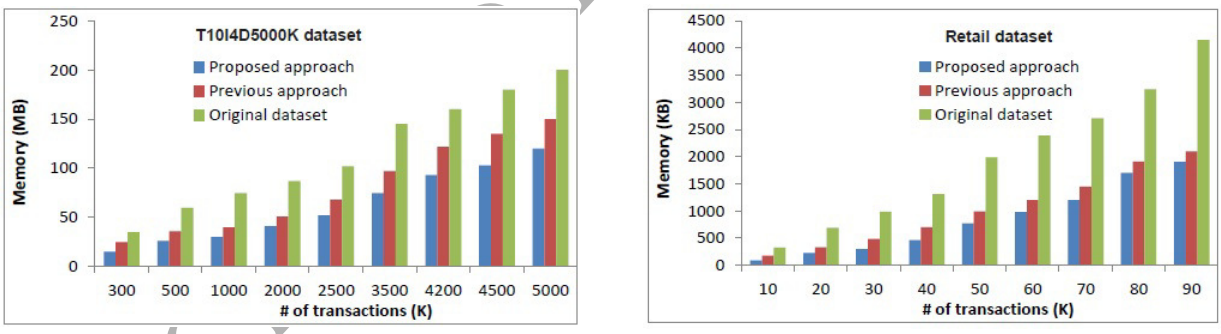
the average transaction length and the average pattern length of T10I4D5000K are set to 5000K, 10 and 4 respectively. Where, the number of transactions, the average transaction length and the average pattern length of T20I4D1000K is 1000K, 10 and 4 respectively. The *Mushroom* dataset record contains data on characteristics of various Mushroom species. The number of records, the number of items and the average record length are be 8124, 119 and 23 respectively.

In order to evaluate the performance of the proposed data transformation technique, we experiment various *min_sup* between 1% to 100%, and observed the increase and decrease in the number of frequent patterns, compact size of the original dataset, memory usage and runtime estimation. table 6 shows characteristics of the real and synthetic datasets used in experiments. Before running our algorithms and observing their performance, we processed our datasets as follows: we partitioned T10I4D5000K into 50 batches with each batch contains 100K transactions. T10I4D1000K was partitioned into 10 batches of 100K transactions in each. The objective of the splitting is to provide the dynamic data streams and distributed environments.

7.2. Performance Analysis of the TV algorithms

7.2.1. Compactness of the datasets

Our approach achieves higher reduction rates compared to existing approaches. The compaction size is almost 60% that is 10% to 20% better compared to [31, 21]. The reason behind these rates is the 1-itemset filtering and the removal of null transactions. Additionally we avoid the vertical layout representation used in previous methods to find frequent items, in which both TIDs and items need to be stored in the main memory resulting in an extensive memory usage. Figure 6 shows comparative performance of our approach in two datasets. We applied our approach to real life anonymous retail store and synthetic T10I4D5000K datasets. Retail store dataset contains many null transactions.



(a) T10I4D5000K dataset

(b) Retail dastaset

Figure 6: Compactness of the datasets on synthetic and real dataset

The compactness rate for the real *Retail* dataset can be higher than synthetic dataset, since a real dataset might be uniformly distributed or sparse. Another reason is the size of the transaction value used for a transaction is almost independent of kinds of datasets, but the average length of items in real datasets is usually higher than synthetic datasets [20]. Table 6 presents results reflecting these dataset characteristics.

7.2.2. TV database building time, efficiency and consistency

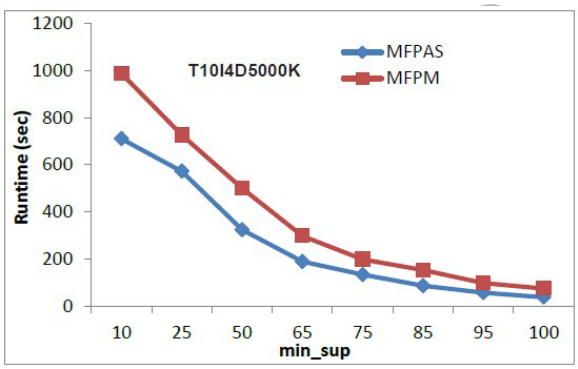
In the sub-section 7.2.1, we have demonstrated the improvement achieved in compaction rate by using our approach. This reduction provides a a big advantage for reducing the TV database size as well. Typically, a TV database builds time consists of filtered database reading time, prime generation, prime assignment, multiplication and Spark's materializing action time. Where, the database reading is much faster operation

compared to single computer machine as we used Spark APIs to read the database. In parallel, we used a faster prime generation algorithm where the prime assignment time is almost linear. Moreover, the multiplication operation is a faster operation done by the math co-processor(incorporated inside the main processor) hence, it also takes almost linear time. After the multiplication is done, we use Spark's API to store and materialize the TVs as TV database, which is eventually used for building the tree structure. According to sub-section 5.3, decoding and reverse prime assignment is also consistent and can correctly retrieve the correct pattern demonstrated into sub-section 7.4.1.

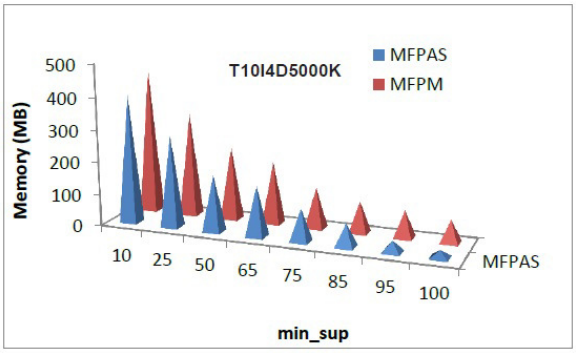
7.3. Performance Analysis of the ASP-Tree structure

7.3.1. Memory Usage and Tree Construction Time

In this experiment, we compared the tree construction time and memory usage between our proposed tree structure (ASP-Tree) and the *PS_Tree* structure for both dense (T20I4D1000K) and sparse (T10I4D5000K) dataset. As we already stated that both *PC_Tree* [31] and *PS_Tree* [21] structure maintains two support counts (*local_count* and *global_count*) for the TV values in the tree, resulting more space overhead. While we have been able to construct the tree structure using only the *global_count*. Therefore, we don't need to keep track of the additional support count of the TV values, and this saves lots of precious memories and, of course, processing time as well since, it does need some extra time to increase the support count for two separate counters. Moreover, unlike previous approaches [31, 21], our approach does not need to sort the TV values before inserting into ASP-Tree structure. We consider *min_sup* as the support threshold of this dataset, where, *min_sup* was increased from 1% to 100% to evaluate the memory usage and runtime of the ASP-Tree structure. From fig. 7 and 8, we can observe that ASP-Tree shows better efficiency compared to *PS_Tree* in terms of both runtime and memory usage.



(a) Runtime of ASP-Tree structure

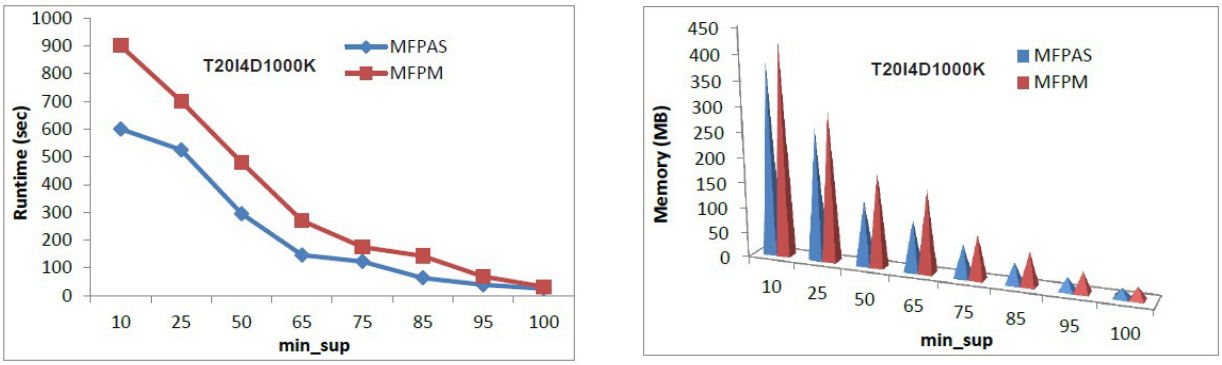


(b) Memory usage of ASP-Tree structure

Figure 7: Memory usage and tree construction time of ASP-Tree structure on T10I4D5000K

7.3.2. Scalability of the ASP-Tree Structure

To measure the scalability of ASP-Tree structure, we compared the runtime and memory usage of proposed approach the previous approach [21] (see figures 7 and 8). In these two figures, the T10I4D5000K dataset is sparser compared to T20I4D1000K. Figure 7 and 8 present the result of the scalability tests on the variation of *min_sup* and required number of tree nodes for the dataset. Clearly, as the *min_sup* decreases, the overall tree construction and mining time, and required memory increases. ASP-Tree structure shows a stable performance with a linear increase in runtime and memory consumption as the *min_sup* decreased for the datasets. Moreover, the results demonstrate that the ASP-Tree can hold all the TV values and candidate frequent patterns on these



(a) Runtime of ASP-Tree structure

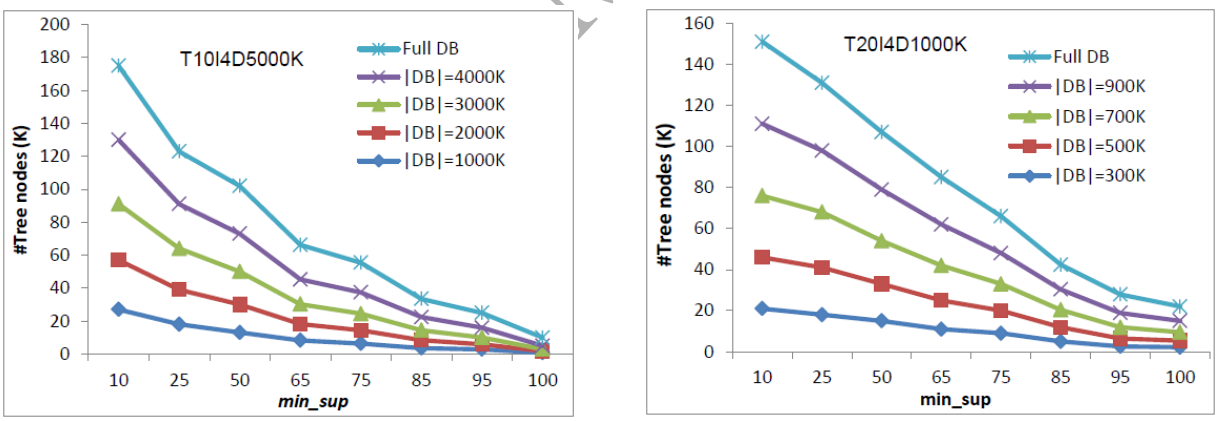
(b) Memory usage of ASP-Tree structure

Figure 8: Memory usage and tree construction time of ASP-Tree structure on T20I4D1000K

datasets for a reasonably small value of support threshold with a considerable amount of execution time and memory.

7.3.3. Compactness of the ASP-Tree Structure

In this section we examine the compactness of the ASP-Tree structure regarding the number of tree nodes. Note that, the Mushroom dataset has a fixed transaction length. Therefore, maximum transaction length for every possible pattern in the dataset is always the same. Consequently, every item in the dataset passes the lazy pruning phase and contributes to the tree. Hence, for a particular portion of the Mushroom dataset, the tree size (i.e., the number of nodes) is the same with the variation of *min_sup*. However, the number of nodes varied from 32,131 (when $|TDB| = 1000K$) to 135,154 (when $|TDB| = 4000K$) for T10I4D5000K dataset and for the full dataset, it is around 175K.



(a) T10I4D5000K

(b) T20I4D1000K

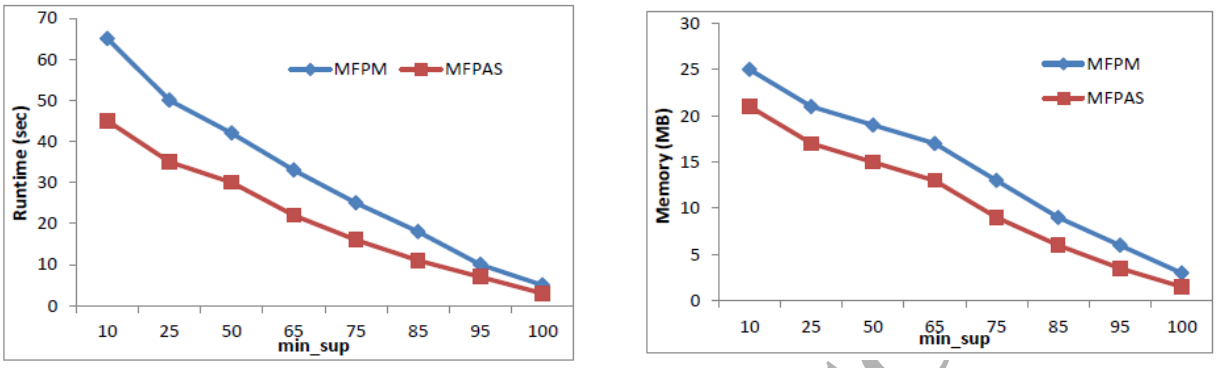
Figure 9: Compactness of ASP-Tree structure

On the other hand, the number of nodes varied from 22,877 (when $|TDB| = 300K$) to 118,368 (when $|TDB| = 900K$) for T20I4D1000K dataset and for the full dataset, it is around 148K. The compactness of ASP-Tree on the different portion of T20I4D1000K and T10I4D5000K are presented in fig. 9. The size of the tree structure is gradually reduced in the both datasets with the increase of *min_sup*. As expected, in both datasets, the number of nodes increased with the rise in the size of the database. However, as far as the total number of nodes is concerned, one can observe that, irrespective of fixed or variable transaction length, a ASP-Tree structure is compact enough to fit into a reasonable amount of memory.

7.4. Performance Analysis of the MFPAS Algorithm

7.4.1. Mining Performance in Terms of Time and Memory Usage

In this experiment we compared the performance between *MFPM* [21] and proposed *MFPAS* algorithm. To demonstrate the comparison, we performed the analysis as Spark's standalone mode to validate, since, *MFPM* was also carried out on single machine mode. Fig. 10 indicates that proposed *MFPAS* algorithm outperforms *MFPM*.



(a) Runtime

(b) Memory usage

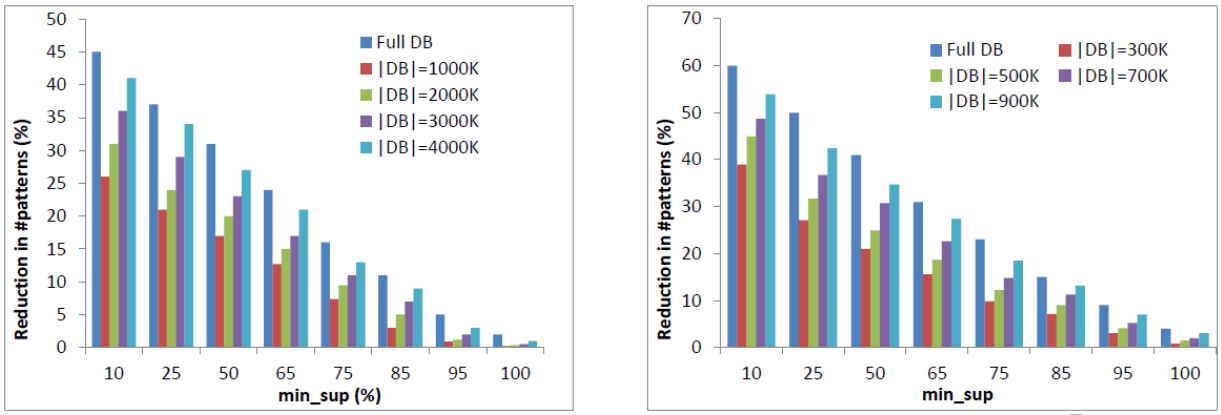
Figure 10: Runtime and memory usage of *MFPAS* algorithm with Mushroom dataset

It is obvious because, *MFPM* has: (i) relatively high candidate generation problem because of not considering the null transactions, ii) which results in less compression rate and iii) larger search space. Also, we have been able to reduce the tree size by applying effective pruning technique that provides a partial downward closure property. We have compared the performance study of MFPAS algorithm with *MFPM* [21] with the Mushroom dataset to show the effectiveness of our approach. Therefore, we just show the mining performance concerning time and memory usage in fig. 10. However, to keep it simple, we did not compare the performance study with other MFP mining algorithms, since this is the first algorithm proposed for mining MFPs in big data environment with Spark.

7.4.2. Reduction of MFPs with the Changes of min_sup

Similar to the previous experiment, we also examined the number of patterns generated by our *MFPAS* algorithm, when we varied the dataset size and *min_sup* threshold. Fig. 11 shows the reduction in the number of patterns in percentage when increasing the *min_sup* values in both the Mushroom and T20I4D5000K datasets with different dataset size. Each data point in the x-axes reports the change of *min_sup* from a low to a high value while the y-axes indicate the percentage change in the number of patterns generated from a low to a high *min_sup* value. Note that, depending on dataset characteristics, the reduction rate also varied. For example, for the Mushroom dataset, the reduction rate dropped sharply when *min_sup* was changed.

In contrast, T10I4D5000K showed a consistent reduction rate when lowering the *min_sup* value. For example, for the mushroom dataset, the reduction rate was around 25% when increasing the threshold from 50% to 65%. For T20I4D1000K, the reduction rate was around 18% when raising the threshold from 50% to 65% for example. Interestingly, the pattern count reduction rate was very similar irrespective of the database size. We observed that the pattern generation characteristics of the MFP mining algorithm were consistent with the variation of *min_sup* and database size.



(a) T10I4D5000K (b) Mushroom

Figure 11: Patterns reduction using *MFPAS* algorithm with respect to *min_sup*

7.5. Performance Analysis of the Approach

The following sub-sections demonstrate the performance of the proposed approach with runtime, scalability and speedup process.

7.5.1. Runtime of the Overall Mining Process

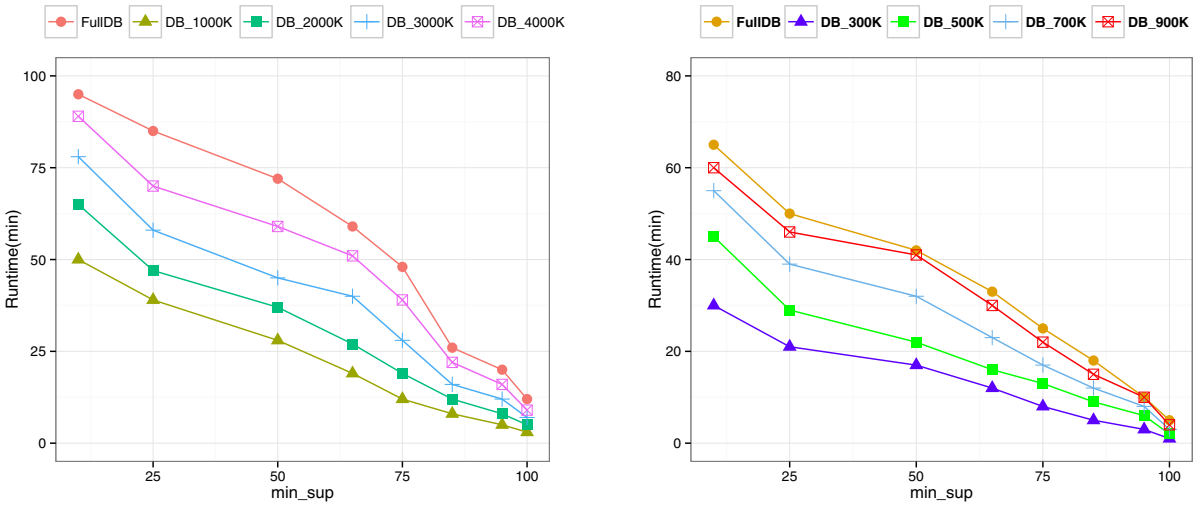
In this section, we analyze the execution time required for mining MFPs over different datasets. We tested our proposed approach in terms of density and changes in *min_sup*. The execution time here includes all the processing, TV database conversion, the ASP-Tree construction and the corresponding mining time. The results on one sparse dataset (e.g., T10I4D5000K) and one dense dataset (e.g., T20I4D1000K) are presented in fig. 12 We have not included the mining time of real dataset intentionally (i.e. Mushroom since the number of transactions in the Mushroom is subtle to consider in a distributed environment, however, we have shown the effect of Mushroom dataset for the speedup of this approach in the section 7.5.2.

To observe the impact of mining on the variation of the example datasets, we performed MFP mining while increasing the size of both of the datasets as batches: (i) From 1 to full(50) for the T10I4D5000K dataset and (ii) from 1 to full(10) for T20I4D1000K.

Thus, the series for *FDB* represent the results for the full size of datasets. Both datasets required more execution time when mining larger datasets. As the database size increased and *min_sup* decreased, the tree structure size and number of MFPs increased. Hence, a comparatively longer time was required to generate large numbers of MFPs from large trees. Although the T20I4D1000K dataset is smaller in size, the transaction lengths of all transactions are the same (i.e. 23). Hence, the TV database building, tree construction and mining took a longer time when compared to a dataset with a variable length such as T10I4D5000K. The experimental results show that mining the corresponding ASP-Tree for MFPs is time efficient for both sparse and dense datasets.

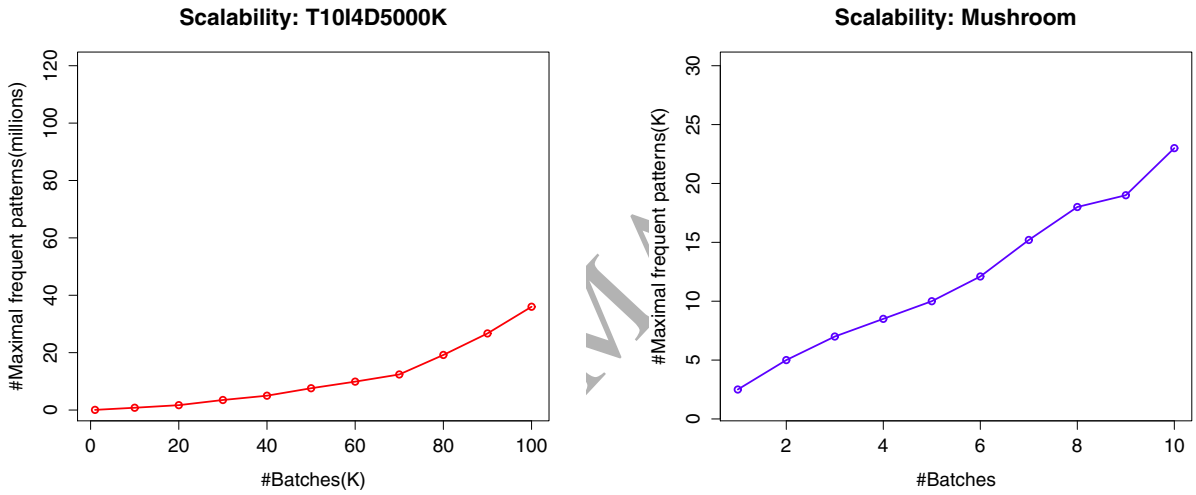
7.5.2. Scalability and Speedup Results of the Overall Approach

Finally, we studied the scalability and speedup of the overall approach. In particular, fig. 13 shows that, when the number of batches is increased, the number of patterns to be stored in the ASP-Tree structure also gradually increased. Consequently, it increases the total number of tree nodes. These results, an increase in the number of mined MFPs, which is proportional to the number of batches for the streaming data. Moreover, memory usage and network latency time is also increased significantly (not shown the corresponding values in



(a) T10I4D5000K (b) T20I4D1000K

Figure 12: Runtime of the overall mining process of the approach



(a) T10I4D5000K (b) Mushroom

Figure 13: Scalability of the approach when varying number of batches

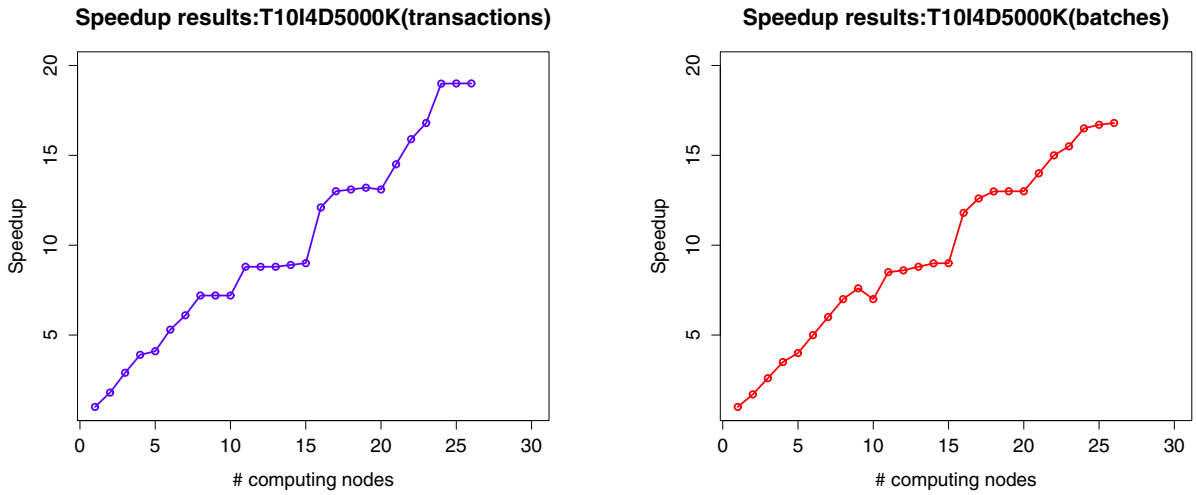
this case). From fig. 14, we can also observe a significant speedup of our approach. During the experiments, we observed almost linear speedup on up to 27 computing nodes on the SOMA infrastructure. However, it becomes saturates when we engage up to 22 and 27 nodes with 19x and 16x speedup for TDBs and DDSs respectively.

8. Conclusion and Outlook

In this section, we summary and discuss the advantages and disadvantages our propososed approach. We also discuss some potential limitations that could be improved via further research. Finally, we have reported some outlook as future works.

8.1. Summary of the Work

In this paper, we proposed an efficient numerical method for mining MFPs from the TDBs and DDSs. This paper also establishes a foundation for combining *Spark* with the data mining techniques. Moreover, we showed promosing results obtained by experiments based on a large real and synthetic data set. Our first results are



(a) Static data

(b) Data streams

Figure 14: Speedup results of the approach for same dataset T10I4D5000K (as static data and data streams)

encouraging; naturally, there are many interesting open issues to be investigated further. Our method also showed an efficient data transformation technique, a novel encoded and compressed ASP-Tree and some other efficient algorithms which reduce both search space, mining and searching time.

The experimental results showed that our proposed algorithms outperform existing MFPs mining algorithms like MAFIA, GenMax, *PC_Miner* or *PS_Tree* and tree structure like FP-growth, CanTree or the *PC_Tree* algorithms respectively. Our performance study with speedup has a shortcoming that is we could not use massive datasets to evaluate the performance in a distributed way. There are two reasons behind this limitation. Firstly, it's difficult to generate gigantic synthetic dataset. Secondly, real transactional datasets are rarely found freely open or with data license agreement. In next section, we discuss the encoding, mining and decoding overheads of our proposed approach.

8.2. Encoding, Mining and Decoding Overheads

We assess the size of transformed transactions in terms of time and memory for TV transformation. Although our proposed approach reduces the size of the original database, the overhead is O(n) + O(m), where n is the number of items and m is the number of prime assignment. We study the overhead at the driver PC of the SOMA infrastructure with a pattern mining task. We measured the performance in run time as well as memory usages. Besides, we also measured the increase in the number of frequent patterns obtained from mining TV database, considering different support thresholds. The overhead is mainly for the TV transformation, I/O, and memory usage. The decoding of the Maximal TV values into MFPs does not take much time according to our observation. In next section, we discuss some possible future works and improvement plans.

8.3. Outlook

Experimental results of our ASP-Tree, *MFPAS* and other associated algorithms are encouraging, and open a way for the defining a general Spark approach for outsourcing of customer purchase rules, frequent pattern mining or association rule mining. Since, many interesting problems area are still uninvestigated, in future we intend to extend this approach by considering more factors and parameters for not only the transactional datasets but also e-commerce datasets, web-sequencing mining, and social network data mining task in details.

A possible domain might be the game industry, using our approach for processing and discovering patterns from the potential firehose of real-time in-game events. In this sector having the capability of immediate response could yield a lucrative business. In the e-commerce industry, real-time transaction information could be passed to a streaming clustering algorithm like k-means or collaborative filtering. Besides, Spark stack could be applied to a fraud or intrusion detection system or risk-based authentication in finance or security industry.

Acknowledgment

The authors would like to thank the anonymous reviewers for their useful comments which helped us to further extend and improve the draft. Part of the work was performed at the Fraunhofer Institute for Applied Information Technology FIT, Germany. Authors also thank the SOMA infrastructure team at Insight Centre for Data Analytics, NUI Galway, Ireland.

References

[1] D. Agrawal, P. Bernstein, et al., Challenges and Opportunities with Big Data, URL `cra.org/ccc/docs/init/bigdatawhitepaper.pdf`, [Online; Last accessed 29-August-2017], 2017.

[2] R. Agrawal, R. Srikant, et al., Fast algorithms for mining association rules, in: Proc. 20th Int. Conf. Very Large Data Bases, VLDB, vol. 1215, 487–499, 1994.

[3] R. Agrawal, R. Srikant, et al., Fast algorithms for mining association rules, in: Proc. 20th int. conf. very large data bases, VLDB, vol. 1215, 487–499, 1994.

[4] T. Apostol, Introduction to analytic number theory, vol. 1, Springer, 1976.

[5] R. J. Bayardo Jr, Efficiently mining long patterns from databases, ACM Sigmod Record 27 (2) (1998) 85–93.

[6] M. S. Bhuvan, V. D. Rao, S. Jain, T. Ashwin, R. M. R. Guddeti, Semantic sentiment analysis using context specific grammar, in: Computing, Communication & Automation (ICCCA), 2015 International Conference on, IEEE, 28–35, 2015.

[7] T. Brijs, G. Swinnen, K. Vanhoof, G. Wets, Using association rules for product assortment decisions: A case study, in: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 254–260, 1999.

[8] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, T. Yiu, MAFIA: A maximal frequent itemset algorithm, Knowledge and Data Engineering, IEEE Transactions on 17 (11) (2005) 1490–1504.

[9] A. Cuzzocrea, F. Jiang, C. K. Leung, D. Liu, A. Peddle, S. K. Tanbeer, Mining Popular Patterns: A Novel Mining Problem and Its Application to Static Transactional Databases and Dynamic Data Streams, in: Transactions on Large-Scale Data-and Knowledge-Centered Systems XXI, Springer, 115–139, 2015.

[10] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[11] Ericsson, Big Data Analytics, in: Ericsson White paper, Ericsson, 1–12, uen 288 23-3211 Rev B, 2015.

[12] K. Gouda, M. J. Zaki, Efficiently mining maximal frequent itemsets, in: Data Mining(ICDM,2001), Proceedings IEEE Int. Conf. on, IEEE, 163–170, 2001.

[13] K. Gouda, M. J. Zaki, Genmax: An efficient algorithm for mining maximal frequent itemsets, Data Mining and Knowledge Discovery 11 (3) (2005) 223–242.

[14] A. Gupta, V. Bhatnagar, N. Kumar, Mining closed itemsets in data stream using formal concept analysis, in: Data Warehousing and Knowledge Discovery, Springer, 285–296, 2010.

[15] J. Han, J. Pei, Y. Yin, R. Mao, Mining frequent patterns without candidate generation: A frequent-pattern tree approach, Data mining and knowledge discovery 8 (1) (2004) 53–87.

[16] D. Harnie, M. Saey, A. E. Vapirev, J. K. Wegner, A. Gedich, M. Steijaert, H. Ceulemans, R. Wuyts, W. D. Meuter, Scaling machine learning for target prediction in drug discovery using Apache Spark, Future Generation Computer Systems 67 (Supplement C) (2017) 409 – 417, ISSN 0167-739X, URL http://www.sciencedirect.com/science/article/pii/S0167739X1630111X.

[17] N. Jiang, L. Gruenwald, Research issues in data stream association rule mining, ACM Sigmod Record 35 (1) (2006) 14–19.

[18] Y. Kai, M. Yuan, A fast algorithm for discovering maximum frequent itemsets, in: Communication Software and Networks (ICCSN), 3rd Int. Conf. on, IEEE, 434–438, 2011.

[19] M. R. Karim, S. Halder, B.-S. Jeong, H.-J. Choi, Efficient mining frequently correlated, associated-correlated and independent patterns synchronously by removing null transactions, Human Centric Technology and Service in Smart Space (2012) 93–103.

[20] M. R. Karim, B. Jeong, H. Choi, Mining E-shopper's Purchase Behavior Based on Maximal Frequent Patterns: An E-commerce Perspective, in: Information Science and Applications (ICISA), 2nd Int. Conf. on, IEEE, 234–238, 2012.

[21] M. R. Karim, M. M. Rashid, B. Jeong, H. Choi, Privacy Preserving Mining Maximal Frequent Patterns in Transactional Databases, in: Database Systems for Advanced Applications(DASFAA), Springer, 303–319, 2012.

[22] M. R. Karim, R. Sahay, D. Rebholz-Schuhmann, A Scalable, Secure and Realtime Healthcare Analytics Framework with Apache Spark, in: Proceedings of the 2nd INSIGHT student conference on Data Analytics, The Insight Centre for Data Analytics, 83–83, 2015.

[23] M. R. Karim, A. Sridhar, Scala and Spark for Big Data Analytics, Packt Publishing Limited, 2017.

[24] C. K.-S. Leung, B. Hao, Mining of frequent itemsets from streams of uncertain data, in: Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on, IEEE, 1663–1670, 2009.

[25] C. K.-S. Leung, F. Jiang, Frequent pattern mining from time-fading streams of uncertain data, in: Data Warehousing and Knowledge Discovery, Springer, 252–264, 2011.

[26] C. K.-S. Leung, Q. Khan, T. Hoque, CanTree: a tree structure for efficient incremental mining of frequent patterns, in: Data Mining(ICDM), Fifth IEEE Int. Conf. on, IEEE, 8–pp, 2005.

[27] C. K.-S. Leung, Q. Khan, Z. Li, T. Hoque, CanTree: a canonical-order tree for incremental frequent-pattern mining, Knowledge and Information Systems 11 (3) (2007) 287–311.

[28] D. Lin, Z. Kedem, Pincer-search: an efficient algorithm for discovering the maximum frequent set, Knowledge and Data Engineering, IEEE Transactions on 14 (3) (2002) 553–566.

[29] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, D. K. Panda, Accelerating spark with RDMA for big data processing: Early experiences, in: High-Performance Interconnects (HOTI), 2014 IEEE 22nd Annual Symposium on, IEEE, 9–16, 2014.

[30] A. Meenakshi, K. Alagarsamy, Efficient Storage Reduction of Frequency of Items in Vertical Data Layout, International Journal 3.

[31] N. Mustapha, M. Nadimi-Shahraki, A. Mamat, M. Sulaiman, et al., A Numerical Method For Frequent Patterns Mining, Journal of Theoritical and Applied Information Technology (2005) 92–98.

[32] M. Nadimi-Shahraki, N. Mustapha, M. N. B. Sulaiman, A. B. Mamat, A new method for mining maximal frequent itemsets, in: Information Technology, 2008. ITSim 2008. Int. Symposium on, vol. 2, IEEE, 1–4, 2008.

[33] B. Nair, A. K. Tripathy, Accelerating Closed Frequent Itemset Mining by Elimination of Null Transactions, Journal of Emerging Trends in Computing and Information Sciences 2 (7) (2011) 317–324.

[34] R. Rymon, Search through systematic set enumeration, Technical Reports (CIS) (1992) 297.

[35] A. Selberg, An elementary proof of the prime-number theorem, The Annals of Mathematics 50 (2) (1949) 305–313.

[36] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, Statistical technique for online anomaly detection using Spark over heterogeneous data from multi-source VMware performance data, in: Big Data (Big Data), 2014 IEEE International Conference on, IEEE, 1086–1094, 2014.

[37] J. Vaidya, C. Clifton, Privacy preserving association rule mining in vertically partitioned data, in: Proceedings of the 8th ACM SIGKDD int. conf. on Knowledge discovery and data mining, ACM, 639–644, 2002.

[38] H. Wang, C. Hu, Mining Maximal Patterns Based on Improved FP-tree and Array Technique, in: Intelligent Information Technology and Security Informatics (IITSI), Third Int. Symposium on, IEEE, 567–571, 2010.

[39] J. Wang, C. Xu, Y. Pan, An incremental algorithm for mining privacy-preserving frequent itemsets, in: Proceedings of Fifth Int. Conference on Machine Learning and Cybernetics, Citeseer, 13–16, 2006.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2–2, 2012.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with Working Sets, in: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, USENIX Association, Berkeley, CA, USA, 10–10, URL http://dl.acm.org/citation.cfm?id=1863103.1863113, 2010.

[42] M. Zaki, Scalable algorithms for association mining, Knowledge and Data Engineering, IEEE Transactions on 12 (3) (2000) 372–390.

[43] G. Zhao, C. Ling, D. Sun, SparkSW: scalable distributed computing system for large-scale biological sequence alignment, in: Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, IEEE, 845–852, 2015.

[44] J. Zheng, A. Dagnino, An initial study of predictive machine learning analytics on large volumes of historical data for power system applications, in: Big Data (Big Data), 2014 IEEE International Conference on, IEEE, 952–959, 2014.