Association Rule Mining Algorithms for Big Data using RDD-ECLAT Algorithms

[1]Sheshikala M ,[2]Ramdas Vankdothu, [3]Mohd Abdul Hameed,4 Rekha Gangula

[1]Associate Professor, Department of Computer Science & Engineering, SR University Warangal, India
[2]Research Scholar, Department of Computer Science & Engineering, Osmania University Hyderabad, India
[3]Assistant Professor, Department of Computer Science & Engineering, Osmania University Hyderabad, India
[4]Assistant Professor, Department of Computer Science & Engineering, KITS Warangal, India

Abstract: The revolution in technology for storing and processing big data leads to data intensive computing as a new paradigm. To find the valuable and precise big data knowledge, efficient and scalable data mining techniques are required. In data mining, different techniques are applied depending on the kind of knowledge to be mined. Association rules are generated from the frequent itemsets computed by frequent itemset mining (FIM) algorithms. The problem of designing scalable and efficient frequent itemset mining algorithms on the Spark RDD framework. The research done in this thesis aims to improve the performance (in terms of execution time) of the existing Spark-based frequent itemset mining algorithms and efficiently re-design other frequent itemset mining algorithms on Spark. The particular problem of interest is re-designing the Eclat algorithm in the distributed computing environment of the Spark. The paper proposes and implements a parallel Eclat algorithm using the Spark RDD architecture, dubbed RDD-Eclat. EclatV1 is the earliest version, followed by EclatV2, EclatV3, EclatV4, and EclatV5. Each version is the consequence of a different technique and heuristic being applied to the preceding variant. Following EclatV1, the filtered transaction technique is used, followed by heuristics for equivalence class partitioning in EclatV4 and EclatV5. EclatV2 and EclatV3 are slightly different algorithmically, as are EclatV4 and EclatV5. Experiments on synthetic and real-world datasets.

Keywords: Association Rule, Data Mining, Big Data, Spark, RDD framework.

1. INTRODUCTION

It is either machine itself or human using machine; both are generated data continuously. Such a petabyte scale of data may be produced from business transactions, scientific simulation or experimentation, social networking sites, web contents, sensor devices etc. Data is the new oil used as the fuel for creating big business value [towardsdatascience.com]. But, without using certain appropriate algorithms, the big business values or insight cannot be harvested from the data

The development in technologies for storing and processing large amounts of data has ushered in a new paradigm of data intensive computing. To extract meaningful and precise knowledge from large amounts of data, efficient and scalable data mining approaches are required. Data mining is the process of identifying and analysing hidden and fascinating patterns within a massive amount of data. Various techniques are used in data mining, depending on the type of knowledge to be mined. The data mining technique association rule mining (ARM) is used to uncover relevant relationships between database data objects. Association rules are derived from frequently occurring itemsets computed using frequent itemset mining (FIM) techniques. FP-Growth is one of the three fundamental algorithms for frequent mining itemsets. Numerous versions and expansions of these algorithms have been developed to facilitate the mining

of frequently occurring itemsets. These are sequential algorithms that run on a single machine with limited processing and memory resources. Thus, these algorithms have been parallelized and disseminated to execute parallel and distributed computing systems. However, these parallel and distributed algorithms are incapable of efficiently handling and processing large amounts of data, as typical distributed systems focus on data exchange, which requires a high level of communication and network capacity. Additionally, it lacks a fault-tolerant programming language and a high-level parallel programming language. Hadoop [http://hadoop.apache.org] is a distributed system that relocates computation to the location of the data rather than relocating the data. Hadoop is a fault-tolerant distributed batch processing system for massive amounts of data. HDFS (Hadoop Distributed File System) [http://hadoop.apache.org] and MapReduce are its two primary components. MapReduce is a scalable and parallel programming model for processing data stored in HDFS in parallel. HDFS holds large amounts of data in the form of blocks across the Hadoop cluster's nodes, and a MapReduce job is conducted as a series of independent tasks on the various data splits throughout the Hadoop cluster's nodes.

While MapReduce is scalable and fault-tolerant, it is inefficient for iterative data mining techniques due to the substantial I/O and network overhead associated with writing/reading intermediate results to/from HDFS. Additionally, invoking a new MapReduce job for each iteration increases the overall execution time of algorithms. Spark [http://spark.apache.org], a more efficient and faster big data processing platform, overcomes the limitations of Hadoop MapReduce by incorporating a number of powerful features such as resilient distributed datasets (RDDs), a rich set of operations including map and reduce in-memory computation, and batch, interactive, iterative, and streaming data processing. RDD is a set of immutable data items that are partitioned across Spark's nodes. Spark caches intermediate results in memory, minimizing read/write cycles to the storage system. It maintains RDDs and operations using a DAG (Directed Acyclic Graph), which makes it significantly faster on disc than Hadoop MapReduce [http://spark.apache.org/] .org].

The following are the section's key contributions:

1. The parallel Eclat method is proposed as RDD-Eclat using the Spark RDD framework.
2. RDD-Eclat is implemented as five variants named as EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5. These variants are based on the different strategies applied in speculation that better performance will be achieved.
3. Transaction filtering is adopted on some algorithm variants to observe the effect on space and time complexity. The heuristics for partitioning equivalence classes are applied on some algorithms to achieve partitions with a balanced workload.
4. Extensive experiments are carried to compare the performance of all proposed algorithms with Spark-based Apriori. Additionally, the proposed methods are compared in terms of performance and scalability.

2. RELATED WORK

The formal name is KDD (Knowledge Discovery in Database) [Han *et al.* (2006)]. KDD is comprised of a set of three basic steps, respectively known as pre-processing, post-processing. The data mining step is the core of KDD that has several algorithms depending on the type of knowledge to be extracted. The pre-processing step consists of data cleaning, data integration, data selection, and data transformation. The post-processing step deals with patterns evaluation and knowledge representation [Han *et al.* (2006)], [Jen *et al.* (2016)]. In some literature, Data Mining and KDD are used interchangeably. However, data mining is the essential step that discovers or extracts hidden and interesting patterns from a huge volume of data [Han *et al.* (2006)]. The data mining methods are classified into two types: predictive and descriptive [Jen *et al.* (2016)]. The methods like classification, regression, and outlier detection are predictive, whereas the methods like association rule mining, cluster analysis are descriptive [Han *et al.* (2006)], [Tan *et al.* (2005)], [Jen *et al.* (2016)].

The Association Rule Mining (ARM) technique [Agrawal *et al.* (1993)] of data mining. Association Rules Mining reveals the correlations or associations among the database's set of attributes (or items). The very first application of ARM was the market basket analysis. The market basket analysis identifies customers' buying patterns based on the frequent items purchased by them [Jen *et al.* (2016)]. This descriptive method of data mining is now used in several areas other than business. Some of them are financial data analysis, customer relationship management, protein sequences, health care informatics, text data analysis, and social network analysis, etc. [Altaf *et al.* (2017)], [Rajak *et al.* (2008)], [Han *et al.* (2006)], [Jen *et al.*(2016)].

ARM is based on the frequent pattern (or itemset) analysis. First, frequent itemsets are generated, and then association rules are generated from frequent itemsets. An itemset that frequently occurred in the database is termed frequent itemset [Jen *et al.* (2016)]. The frequent occurrence is determined against a user-defined threshold value called minimum support. A similar threshold called minimum confidence is used during association rule generation [Jen *et al.* (2016)], [Kovacs *et al.* (2013)]. Only strong rules satisfying minimum confidence are produced. Although ARM is comprised of two processes, frequent itemset mining (FIM) and rule generation, FIM has higher computational complexity than rule

generation [Han *et al.* (2006)], [Kovacs *et al.* (2013)]. So, most of the research has been done to design and develop efficient frequent itemset mining algorithms. Agrawal and Srikant introduced the first efficient algorithms for frequent itemset mining and association rule creation [Agrawal et al. (1994)]. The approach is called the Apriori algorithm since it is based on the Apriori property, which produces frequently occurring itemsets [Kovacs et al. (2013)]. Later on, several authors suggested various methods, each claiming to be more efficient and scalable than the previous one. Zaki et al. [Zaki et al. (1997b)] presented the Eclat and Clique algorithms and their variants based on the equivalence class and clique notions. Han et al. [Han et al. (2000)] introduced the FP-Growth algorithm, which is based on the frequent pattern tree (FP-tree) structure for compressing frequent itemsets. Various authors have developed numerous more FIM algorithms, which are extensions or enhanced versions of the three previously mentioned methods Apriori, Eclat, and FP-Growth. Hadoop is a distributed computing system that overcomes the

complexity and limitations of traditional distributed systems. For example, the traditional distribution system is not easy to manage, highly dependent on the network. It does not support high-level language like Java, Python, etc., and fault tolerance in node failure. Further, in the case of well known parallel programming model MPI (Message Passing Interface), the programmer has to write the complex code in the languages like C and FORTRAN, take care of data partitioning, synchronization, scheduling, and re-execution of the job in case of node failure [Lin *et al.* (2012)], [Moens *et al.* (2013)]. Hadoop [http://hadoop.apache.org] is a large-scale distributed batch processing infrastructure developed for parallel big data processing on an extremely scalable cluster of commodity computers [Yahoo! Hadoop Tutorial]. Big Data is that type of data, which cannot be managed and handled by a single machine. Gartner and IBM have defined Big Data initially with four characteristics that are known as four V's concept: Volume, Velocity, Variety and Veracity [Douglas (2001)], [Beyer*et al.* (2012)], [IBM (four-vs-big-data)]. However, currently, the concept of four V's stretched to 10 V's (Volume, Velocity, Variety, Veracity, Variability, Validity, Vulnerability, Volatility, Value, and Visualization) [Walker (2014)]. Big data does not have high volume, but other complexities also like streaming, unstructured, and uncertainty nature. Hadoop is capable of storing, managing, and processing big data based on a completely different parallel and distributed computing model. The traditional distributed system transports the data to nodes of the cluster for computation that very inefficient for big data. Hadoop does not transport the data but moves the computation to the node at which data is resided [Jones (2010)], [Kerzner *et al.* (2013)]. The two components of Hadoop, HDFS, and MapReduce make it possible to compute the data locally at each node [Kerzner *et al.* (2013)]. Data movement requires a lot of network bandwidth, computing power, and time, which degrades the performance. Further, Hadoop handles fault tolerance, data distribution, parallelization, and load balancing internally and automatically [http://hadoop.apache.org].

The sequential data mining algorithms, as well as their parallel and distributed versions of traditional distributed systems are no longer efficient to mine big data at present. That requires re-designing of the data mining algorithms on big data processing platforms like Hadoop MapReduce. The data mining researchers have developed and proposed a number of MapReduce based data mining algorithms on the Hadoop cluster. While MapReduce is scalable and fault-tolerant, it is inefficient for iterative data mining methods due to the high I/O and network overhead associated with writing/reading intermediate results to HDFS. Additionally, invoking a new MapReduce job for each iteration increases the overall execution time of algorithms. Spark [Zaharia et al. (2010)], [http://spark.apache.org], a more efficient and faster big data processing platform, overcomes the limitations of Hadoop MapReduce and adds a number of strong features, such as Resilient Distributed Datasets (RDDs) [Zaharia et al. (2012)], [Kerzner et al. (2013)]. Numerous operations are available, including map and reduce in-memory computation, and batch, interactive, iterative, and streaming data processing. RDD is a set of immutable data items that are partitioned across Spark's nodes. Spark caches intermediate results in memory, minimising read/write cycles to the storage system. It is 100 times quicker than Hadoop MapReduce in memory and ten times faster on disc [http://spark.apache.org]. Due to the introduction of Spark, the research focus has switched away from Hadoop MapReduce-based algorithms and toward Spark-based algorithms. Several authors have presented Spark-based FIM algorithms in recent years [Qiu et al. (2014)], [Rathee et al. (2015)], and [Sethi et al. (2017)]. These algorithms are

re-designs of Apriori and FP-Growth algorithms on the Spark RDD architecture. This thesis aims to develop efficient and scalable FIM algorithms on the Spark RDD architecture, which are critical for extracting insights from large amounts of data.

Yang *et al.* [Yang *et al.* (2015)] improved the classical Apriori and parallelize it on Spark. The improved Apriori optimizes the pruning step, and avoids the repeated scans of the database by mapping it to memory through the array vectors data structure. The Spark version of this algorithm consists of two phases: local frequent itemset generation and global frequent itemset generation. In the first phase, improved Apriori runs locally on each horizontal partition of database distributed to different workers. In contrast, the second phase merges all local itemsets and generates global frequent itemsets.

3. ASSOCIATION RULE MINING AND RDD-ECLAT ALGORITHMS

Association Rule Mining (ARM) is a descriptive-based data mining technique that finds interesting correlations among a set of attributes of a large database. The other techniques of data mining are cluster analysis, classification, and outlier analysis, etc. One or more technique is applied, depending on the kind of pattern or knowledge to extract. In general, data mining is the non-trivial method of recognizing valid, hidden, potentially useful, and eventually understandable patterns in data [Pujari (2017)]. The mining of association rules is exclusively dependent on the Frequent Pattern Analysis or Frequent Pattern Mining. A ***pattern*** may be the set of items, subsequences, substructures etc. Frequent pattern mining is called frequent itemset mining (FIM) when patterns are itemsets in a database. The majority of ARM algorithms are focused on determining frequent itemsets efficiently [Han *et al.* (2006)], [Pujari (2017)].

Problem Statement of Frequent Itemsets and Association Rule Mining

Let $I = \{i1, i2, ...,im\}$ be a set of m distinct ***items*** or attributes. An ***itemset*** is a set of items from set I. A ***k-itemset*** is an itemset consisting of k items. Let $D = \{T1, T2, ...,Tn\}$ be the database of n transactions, such that each transaction T_i comprise of an unique transaction identifier TID_i and an itemset, i.e. in the form of $<TID_i, i1, i2, ..., ik>$. So, the transaction T_i is also a subset of I. The number of items that constitute a transaction is called the ***transaction width*** of that transaction. A transaction T_i is said to support an itemset X if $X \subseteq T_i$. The number of transactions in D, supporting the itemset X is called ***support counts*** of X, denoted as $\sigma(X)$. An itemset X is said to be a ***frequent itemset*** if $\sigma(X) \geq min\_sup$, where min_sup is a user-specified minimum support threshold [Tan s*et al.* (2005)], [Pujari (2017)]. The ***frequent itemset mining*** **(FIM)** is the computation of all frequent itemsets in a given database [Agrawal *et al.* (1994)]. Association rule mining [Agrawal *et al.* (1994)] task is decomposed into two sub-problems. The first sub-problem is to generate all frequent itemsets and is computationally intensive. The second sub-problem is to produce all desired rules simply. An association rule is a conditional implication of the form $X => Y$, where $X, Y \subseteq I$, and $X \cap Y = \phi$. The ***confidence of the rule*** $X => Y$ is measured as $\sigma(X \cup Y) / \sigma(X)$, in terms of support count of itemsets. A rule holds only if its confidence is more than min_conf, a user-specified minimum confidence threshold [Tan *et al.* (2005)], [Pujari (2017)]. The generation of all frequent itemsets is a computationally and memory and disk I/O intensive task compared to rule generation [Zaki *et al.* (1997c)]. Various efficient algorithms have been developed to overcome these issues.

Association Rule Generation

In order to generate association rules, computing and checking support and confidence of every potential rule is not an efficient approach. The confidence of a rule is defined in terms of support of the itemsets that form the rule [Jen *et al.* (, 2016)]. So, the association rule generation is divided into two steps. The first step produces frequent itemsets along with their support, and in the second step, strong rules are produced from these frequent itemsets [Han *et al.* (2006)]. The process of frequent itemset generation or FIM is a more computation as well as memory, and disk I/O intensive task [Zaki *et al.* (1997c)], [Jen *et al.* (2016)].

RDD-ECLAT ALGORITHMS

discussed in the Chapter 3. After the production of frequent itemsets, association rules are computed. For each frequent itemset *l*, all nonempty subsets of *l* are generated. For every nonempty subset *s* of *l*, the rule *"s □ (l-s)"* is generated if $\sigma(l) / \sigma(s) \geq min\_conf$ [Han *et al.* (2006)], [Jen *et al.* (2016)].

Frequent Itemsets Cohort

Suppose a database consists of *k* items, *{i1, i2, ..., ik}* then it will generate up to $2^k - 1$ potential itemset excluding null set using the brute force approach [Tan *et al.* (2005)]. If the database contains *n* transactions and *w* is the maximum transaction width, then the complexity of support counting will be $O(n*w*(2^k-1))$ [Jen *et al.* (2016)]. This brute force approach is very expensive as the value of *v, w,* and *k* are very large in real-life scenarios [Tan *et al.* (2005)]. For example, Figure.1 illustrates the itemset lattice of 15 itemsets generated from the set of five items *{a, b, c, d, e}* excluding the null set.

Several frequent itemset mining algorithms have been proposed that efficiently generate the frequent itemsets as well as require less memory. Frequent itemset mining algorithms are discussed in the following section.

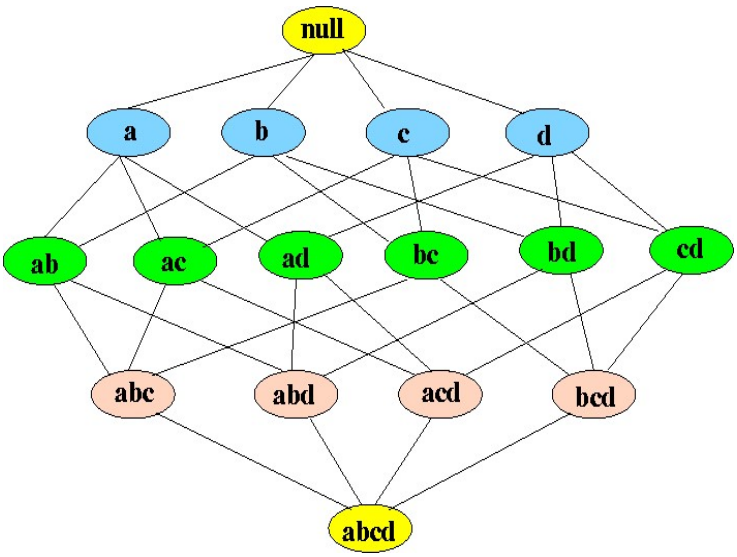


Figure 1**:** Itemset lattice formed by four items (the figure is reproduced from the slides taken from with the author's permission)

FREQUENT ITEMSET MINING ALGORITHMS

The three elementary algorithms of frequent itemset mining are Apriori [Agrawal *et al.* (1994)], Eclat and Clique [Zaki *et al.* (1997b)], and FP-Growth [Han *et al.* (2000)]. These algorithms are based on three fundamental distinct properties: Apriori, equivalence class and clique, and FP-tree. There is a long list of FIM algorithms, and most of them are either extension or improved version of these three algorithms. Apriori is the seminal algorithm for mining frequent itemsets efficiently, based on candidate generation using the Apriori property. A candidate or candidate itemset is a potentially frequent itemset not checked against minimum support. The algorithms Eclat and Clique exploit equivalence class, clique, and lattice properties to generate frequent itemsets. The FP-Growth algorithm mine frequent itemsets in a faster way without generating candidate itemsets. This thesis explores the first two frequent itemset mining algorithms, Apriori and Eclat, in Spark's distributed environment.

SPARK

While Hadoop MapReduce is scalable and robust, it is inefficient for iterative data mining techniques due to the significant I/O and network overhead associated with writing/reading intermediate results to/from HDFS. Additionally, invoking a new MapReduce job for each iteration increases the overall execution time of algorithms. Spark [http://spark.apache.org], a more efficient and faster big data processing platform, [Zaharia et al. (2010)], overcomes the limitations of Hadoop MapReduce by incorporating a number of powerful features such as Resilient Distributed Datasets (RDDs) [Zaharia et al. (2012)], a rich set of operations including map and reduce, in-memory computation, and support for batch, interactive, iterative, and parallel processing RDDs (Resilient Distributed Datasets) are a collection of immutable data objects that are partitioned among the Spark cluster's nodes. Spark caches intermediate results in memory, minimising read/write cycles to the storage system. It is 100 times quicker than Hadoop MapReduce in memory and ten times faster on disc [http://spark.apache.org].

A Spark application has a ***driver program*** running the main() function that implements the control flow of application and launches different operations on the Spark cluster in parallel. A Spark cluster consists of a master node and a number of worker nodes; all can be on a single machine. The architectural details of Spark cluster is shown in Figure 2 which is reproduced from [Cluster Overview (2018)].

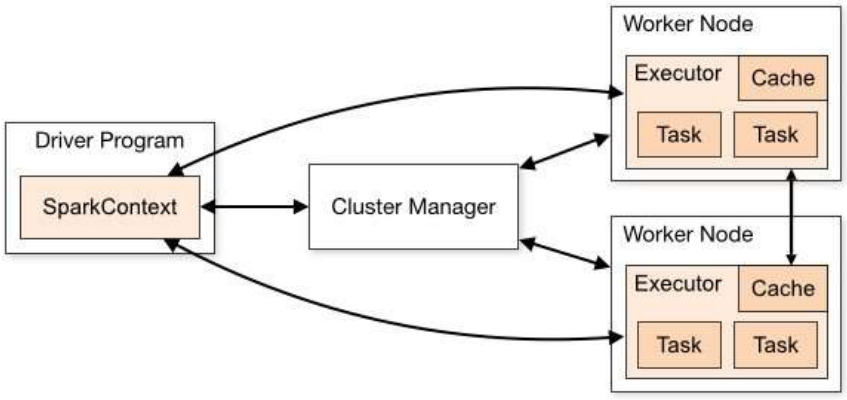


Figure 2**:** Spark Cluster Architecture (the figure is reproduced from Apache Spark Website[https://spark.apache.org/docs/latest/cluster-overview.html])

RDDs can be formed in four ways: by parallelizing an existing collection in the driver programme, by importing

them from an external storage system such as HDFS, by transforming existing RDDs, or by altering the persistence of existing RDDs [Zaharia et al. (2010)]. There are two sorts of operations available in RDDs: transformations and actions. The transformation operations (e.g. map, flatMap, filter, reduceByKey, and so on) generate a new RDD from an existing one, whereas the active operations produce a new RDD from an existing one (e.g. collect, count, saveAsTextFile etc.) return the result to driver program or writes to external storage, after applying the relevant computation on RDD. Spark uses ***lazy evaluation*** technique, i.e. the transformations will not execute until an action is triggered [RDD Programming Guide (2018)]. Spark also provides two restricted types of ***shared variables***: broadcast variables and accumulators. The ***broadcast variable*** is a mechanism to efficiently distribute a copy of a large read-only data to every worker to be cached on. The ***accumulators*** are variables that workers can only "add" to through an associative and commutative operation, and only the driver can read it [Zaharia *et al.* (2010)].

Another, one of the most excellent features of Spark is RDD Persistence i.e. persisting or caching the data in memory throughout the operations. It is key features for iterative algorithms where results of some computed data can be persisted as an RDD that can be reused again and again without re-computation that make latter operation much faster. There are two methods to persist an RDD, persist() and cache(). These methods used on the action and after that result is kept in memory across the nodes of cluster. The persist() method takes a parameter as the storage level for persisting the RDD at that storage level. The different storage levels are memory only, disk only, memory and disk only. The cache() method cache the RDD at default storage level, memory only [RDD Programming Guide (2018)].

RDD-ECLAT ALGORITHMS

We implemented the Eclat algorithm in parallel using the Spark RDD framework and dubbed it RDD-Eclat. By employing increasingly varied techniques and heuristics, we propose five somewhat different variations of RDD-Eclat. EclatV1 is the initial implementation, while its descendants EclatV2, EclatV3, EclatV4, and EclatV5 are the result of modifications to the preceding method. Each of the proposed algorithms is segmented into three to four phases. Phase does not refer to a MapReduce phase in this case, but to a logical step in the process.

EclatV1

The paired RDD is an RDD that contains the pairs of (key, value). The groupByKey() transformation concatenates all pairs that share a common key. The support count of an item is equal to the size of the tidset that contains it. The filter() transformation eliminates items with a support count less than min sup and creates a paired RDD, freqItemTids, that contains only frequent items and their associated tidset. The paired RDD freqItemCounts holds pairs of (item, count), where count is the item's support count. (itemTid. 1, itemTid. 2) is a pair of (key, value) values from a Tuple2 [RDD Programming Guide (2018)] type object, itemTid. Finally, the action collect() provides the complete contents of RDD, freqItemTids to the driver software, where it is sorted and placed in a list in ascending order of item support. The lineage graph for RDDs in Phase-1 of EclatV1 is depicted in Figure 3.

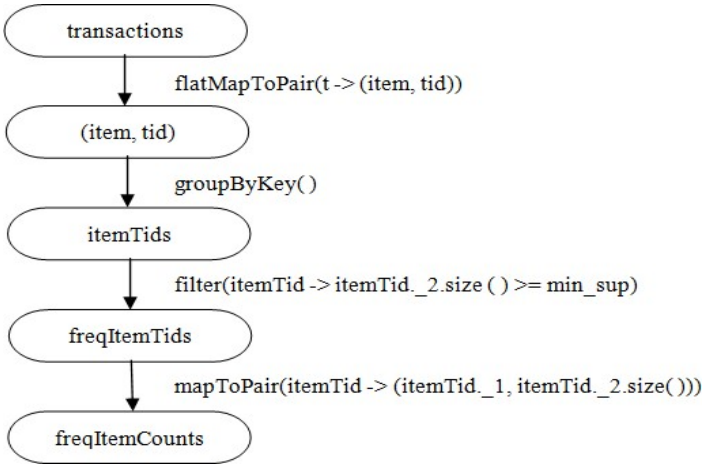


Figure 3**:** Lineage graph for RDDs in Phase-1 of EclatV1

EclatV2

EclatV2 applies the filtered transaction technique adopted from [Borgelt (2003)] to reduce the size of horizontal database. The filtered transactions technique removes the infrequent items from transactions. Consequently, it reduces the cost of operations further applied over the horizontal database, and memory requirement. This optimization is achieved at the cost of additional scanning of database that may amortize if the database is reduced significantly after the filtering. The Phase-1 starts with the partitioning of transactions of the database and creating RDD. The number of partitions is the default that is equal to the number of cores in all machines of the cluster. The transformation, *flatMap()* splits each transaction into items constituting the transaction, then *mapToPair()* converts each item into *(key, value)* pairs, where *key* is the item and *value* is 1. The *reduceByKey()* transformation sums up all values having the same key, then *filter()* transformation checks the sum of values against *min_sup* and returns only frequent items and their support count. Finally, frequent items are returned by the action, *collect()* to the driver, where it is sorted in alphanumeric order, and stored as a list. Here, the level of parallelism is same in all transformations as the number of partitions of parent RDD is preserved throughout the lineage. Figure 4 shows the lineage graph for RDDs in Phase-1 of EclatV2

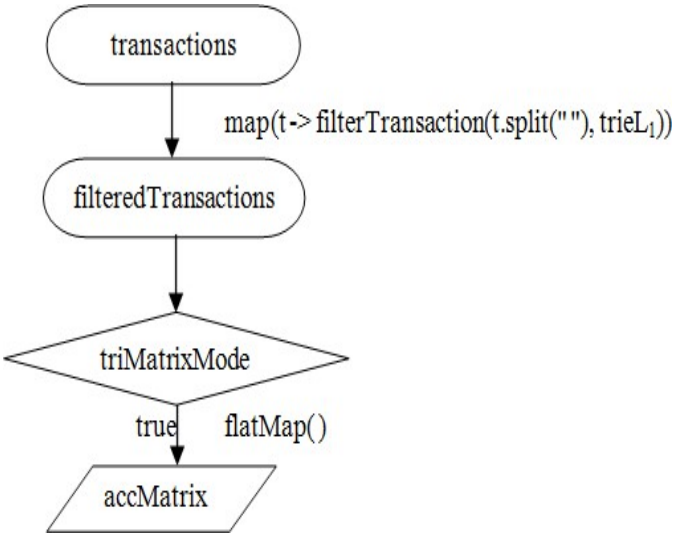


Figure 4**:** Lineage graph for RDDs in Phase-2 of EclatV2

EclatV3

The hashmap, *freqItemTidsMap* is used to sort the list of frequent items of Phase-1, by total order of increasing support count. Algorithm 1 describes the pseudo code of Phase-3.

Algorithm 1: Phase-3 of EclatV3

```
1:   filteredTransactions = filteredTransactions.coalesce(1);
2:   create a hashmap, freqItemTidsMap
3:   pass freqItemTidsMap as accumulator variable, accMap
4:   filteredTransactions.flatMapToPair(t -> {
5:      tid = 1;
6:      accMap.update((t, tid));
7:      tid++;
8:   });
9:   freqItemTidsMap = accMap.value(s);
10:  freqItemList = sort(freqItemList, freqItemTidsMap);
```

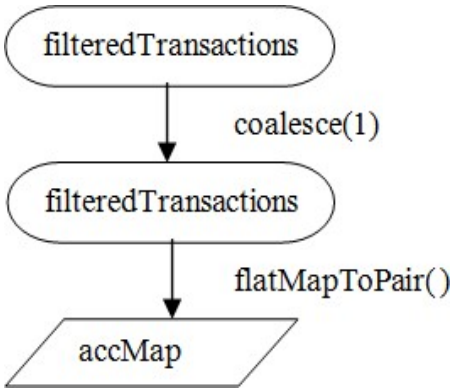


Figure 5 shows the lineage graph for RDDs

EclatV4 and EclatV5

Algorithms EclatV4 and EclatV5 split the equivalence classes into p partitions using EclatV3's heuristics, where p is a user-supplied value. Heuristics are employed to balance the partitions of equivalence classes. These two algorithms differ in phase four from EclatV3, while the first three phases are identical.

Equivalence Class Partitioners

In general, an efficient partitioning of RDDs reduces data shuffling across the network in the subsequent transformations. This section describes the partitioning techniques used in the proposed algorithms to partition the RDDs of equivalence classes, *ECs*, for the parallel and independent computation of frequent itemsets. Partitioning is done on the basis of prefixes of equivalence classes. Spark facilitates to implement custom partitioner; our custom partitioners are based on the HashPartitioner, a default partitioner of Spark. A custom partitioner extends the Partitioner class of Spark, and implements its *getPartition()* method. The heuristic of partitioning is defined in this method. Algorithm 4.9 describes the pseudo codes of *getPartition(v)* method of three custom partitioners, where *v* is the unique value assigned to the 1-length prefix of equivalence classes. A hash map is used to map each frequent item to a unique integer between *0* and *n-1*, where *n* is number of frequent items. EclatV1, EclatV2, and EclatV3 create partitions individually for each equivalence class, i.e. *(n-1)*

partitions. It is termed as default partitioning. EclatV4 and EclatV5 impose a restriction on the number of partitions and construct p partitions, with the user specifying the value of p.The numbers of partitions determine the number of parallel tasks.

4.RESULT AND DISCUSSION

The experimental environment, datasets used, and method performance in terms of execution time. Three categories of performance analysis are discussed.

Execution Time on Varying Value of Minimum Support

Execution times of the proposed RDD-Eclat based algorithms are compared with each other as well as with the Apriori algorithm. Figures 6.8– 6.14 show the execution time of various algorithms on the datasets for the varying value of minimum support.The figures 6.8(a)–6.14(a) compare the execution times of the suggested algorithms to those of the Apriori algorithm, whereas the figures 6.8(b)–6.14(b) compare the execution times of the proposed algorithms EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5. RDD-Eclat outperforms RDD-Apriori on all datasets (Figures 6.1(a)–6.14(a)), and the execution time difference between them is larger when the minimal support value decreases (Figures 6.1(a)–6.14(a)).

If we consider Eclat1 to compare with Apriori, then it can be seen that EclatV1 is at least nine times faster than Apriori on the datasets BMS2 and T40I10D100K (Figures 6.12(a) and 6.14(a)), seven times on the dataset chess (Figure 6.9(a)), six times on the dataset BMS1 (Figure 6.11(a)), four times on the dataset c20d10k, and two times on the datasets mushroom and T10I4D100K (Figures 6.10(a) and 6.13(a)), at the lowest value of minimum support in each case. In short, RDD- Eclat outperforms Apriori by at least two times and up to nine times on some datasets

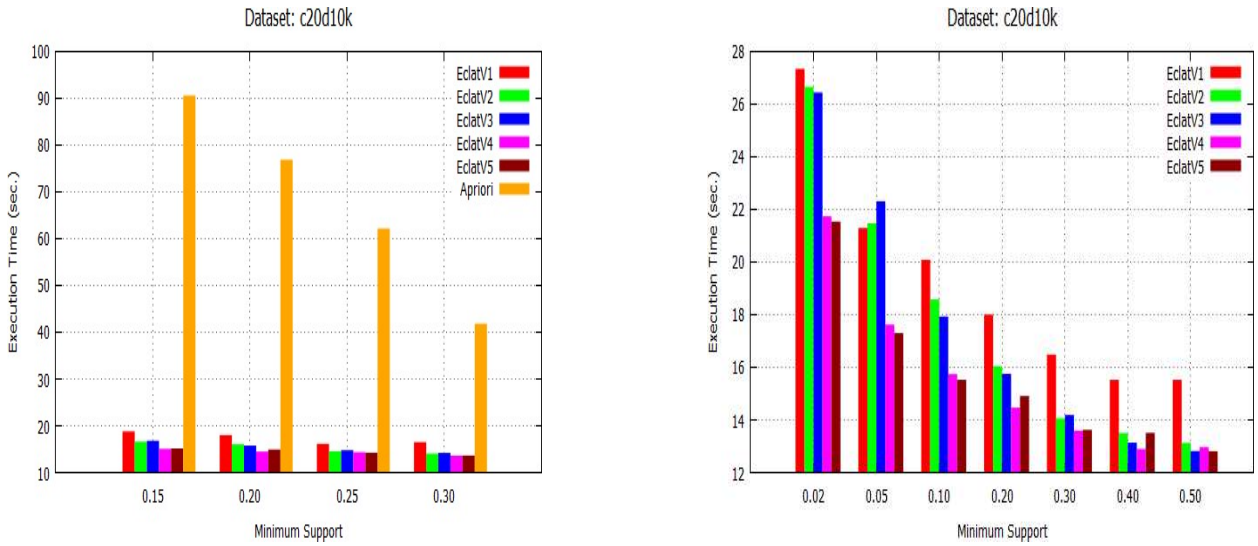


Figure 6.8: Execution time of algorithms (a) EclatV1, EclatV2, EclatV3, EclatV4, EclatV5, and Apriori (b) EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5 for varying minimum support on dataset c20d10k

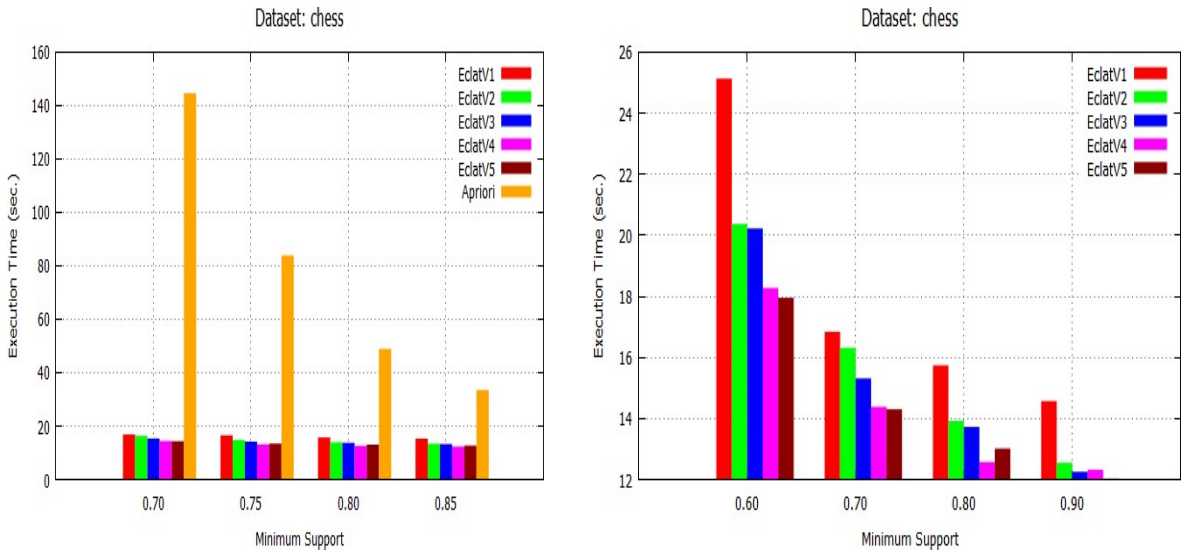


Figure 6.9: Execution time of algorithms (a) EclatV1, EclatV2, EclatV3, EclatV4, EclatV5, and Apriori (b) EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5 for varying minimum support on dataset chess.

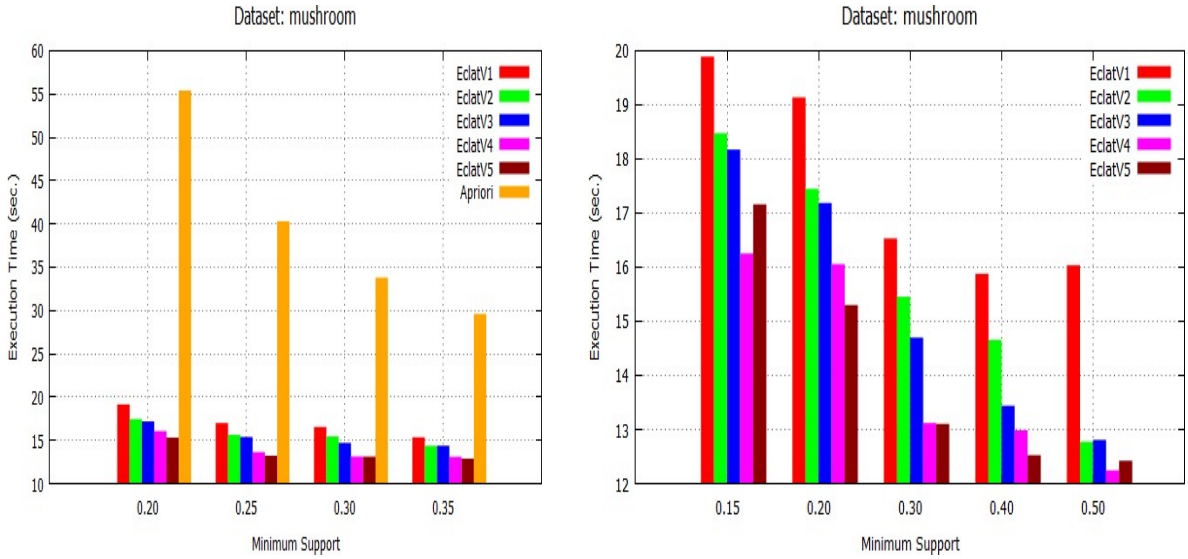


Figure 6.10: Execution time of algorithms (a) EclatV1, EclatV2, EclatV3, EclatV4, EclatV5, and Apriori (b) EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5 for varying minimum support on dataset mushroom.

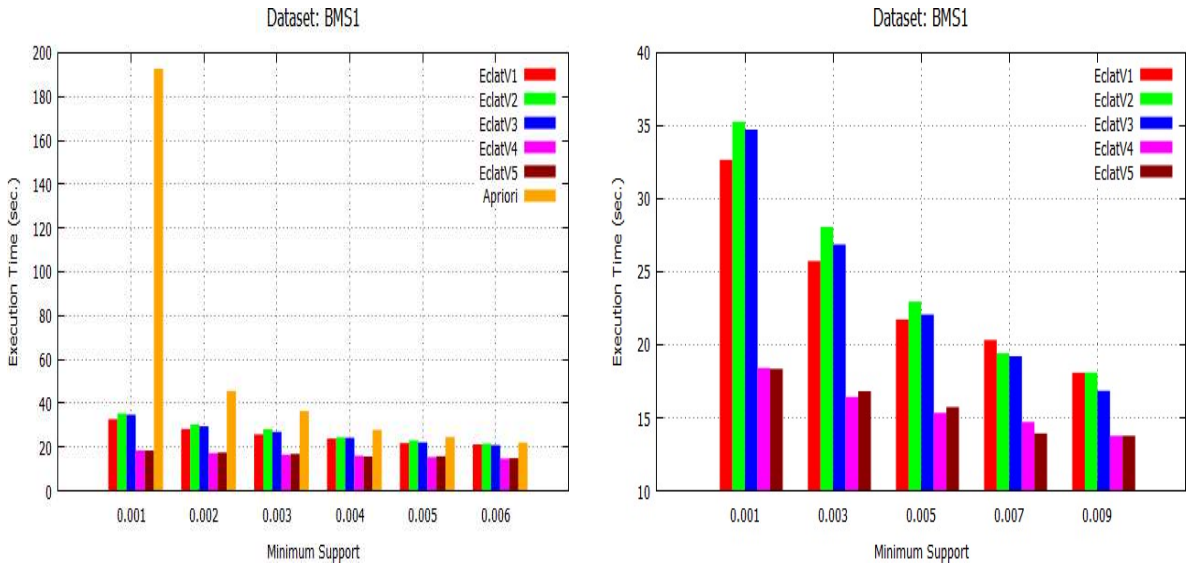


Figure6.11: Execution time of algorithms (a) EclatV1, EclatV2, EclatV3, EclatV4, EclatV5, and Apriori (b) EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5 for varying minimum support on dataset BMS_WebView_1.

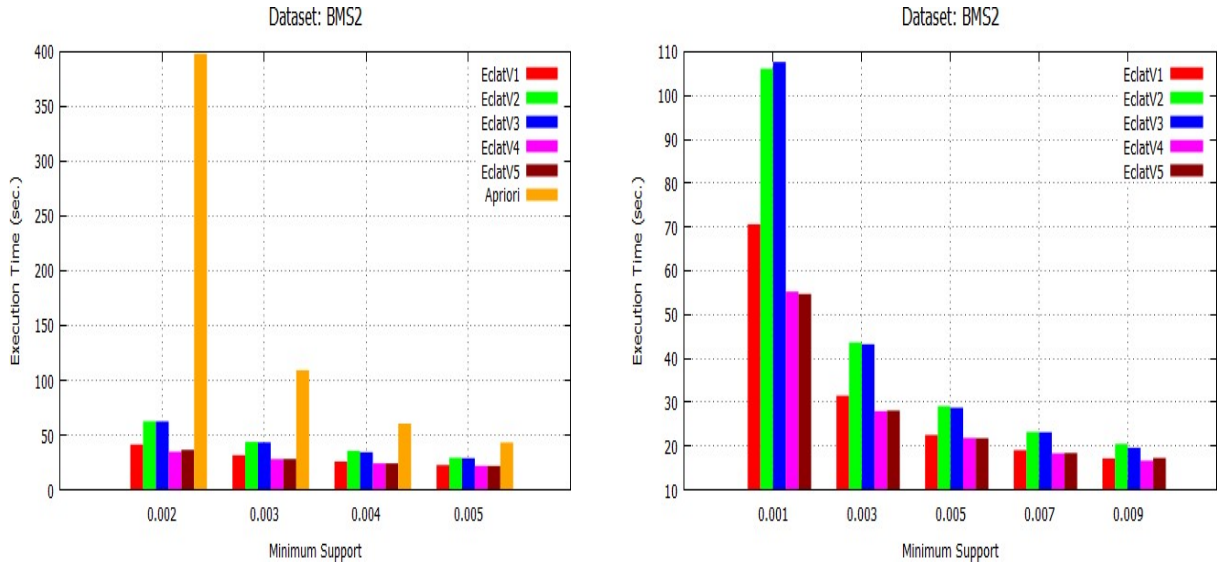


Figure6.12: Execution time of algorithms (a) EclatV1, EclatV2, EclatV3, EclatV4, EclatV5, and Apriori (b) EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5 for varying minimum support on dataset BMS_WebView_2.

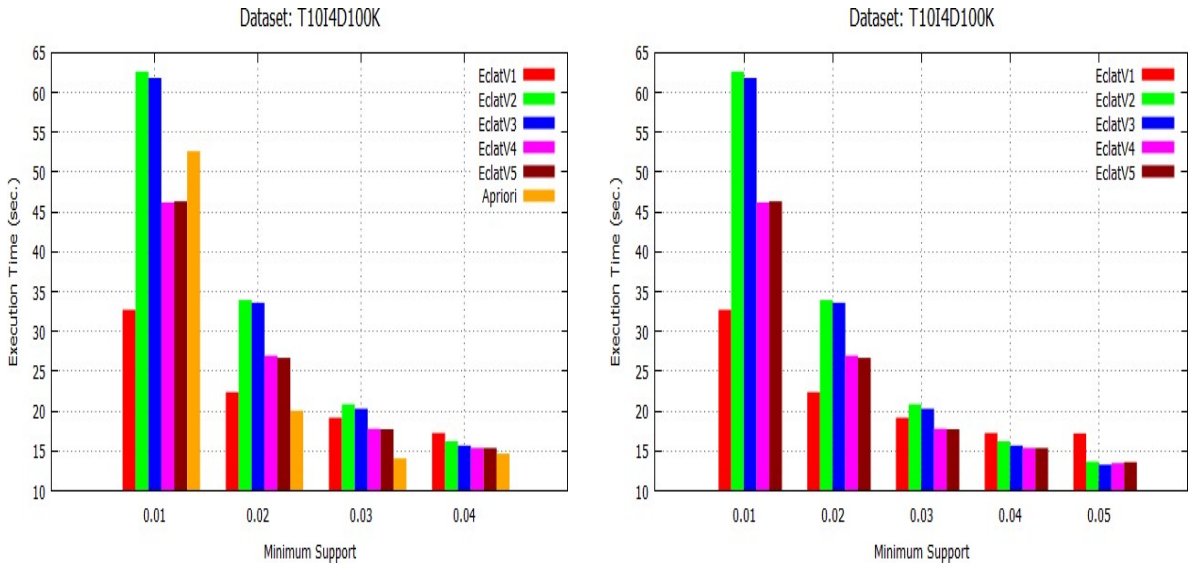


Figure6.13: Execution time of algorithms (a) EclatV1, EclatV2, EclatV3, EclatV4, EclatV5, and Apriori (b) EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5 for varying minimum support on dataset T10I4D100K.

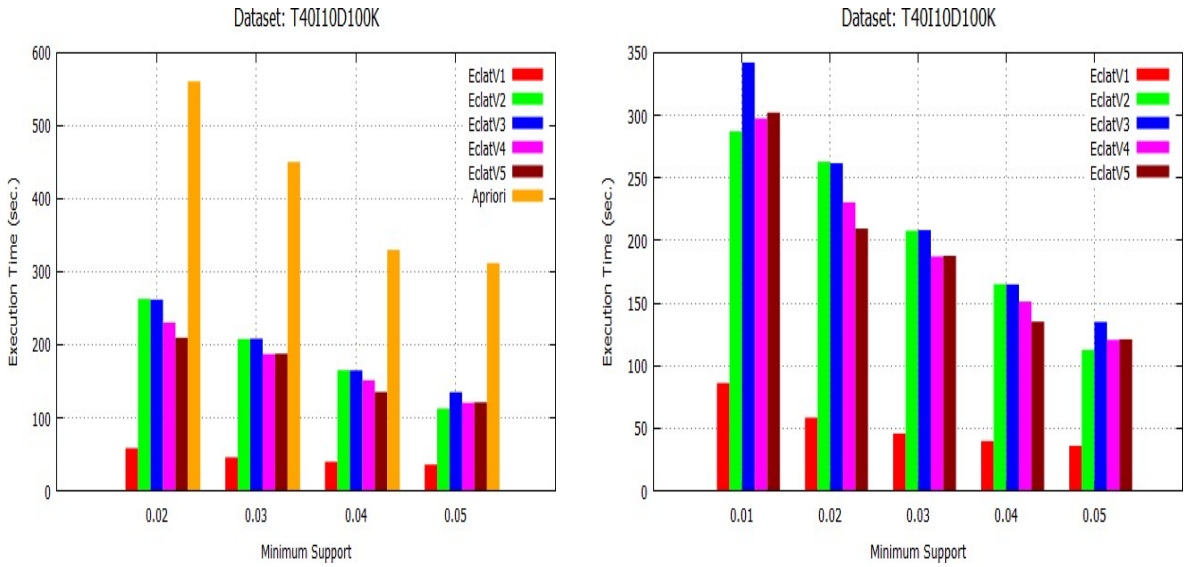
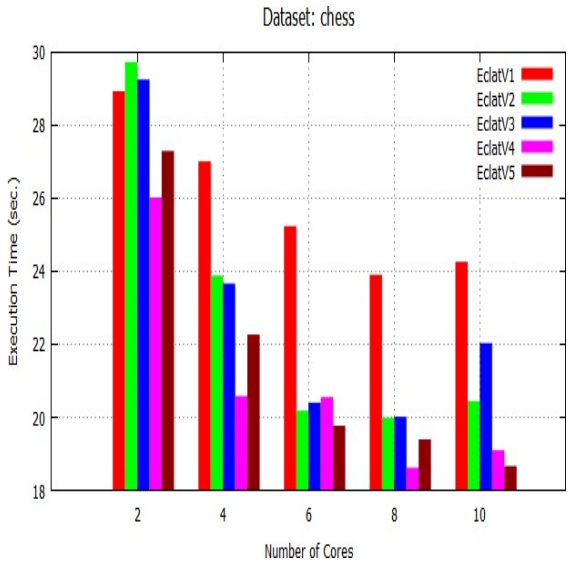


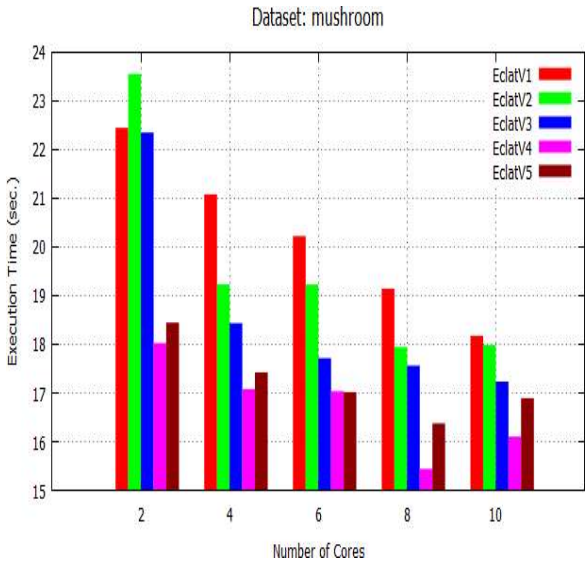
Figure6.14: Execution time of algorithms (a) EclatV1, EclatV2, EclatV3, EclatV4, EclatV5, and Apriori (b) EclatV1, EclatV2, EclatV3, EclatV4, and EclatV5 for varying minimum support on dataset T40I10D100K.

Execution Time on Increasing Number of Executor Cores

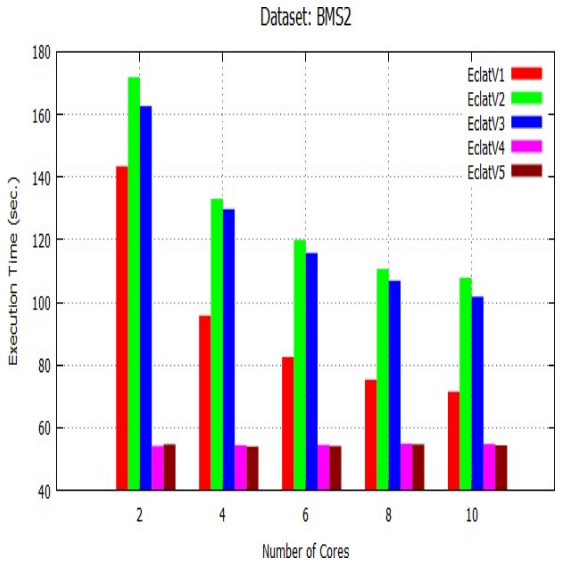
The behavior of the proposed algorithms is investigated on the different datasets at the respective value of lower minimum support, for the increasing number of executor cores, as shown in Figure 4.15(a – e).



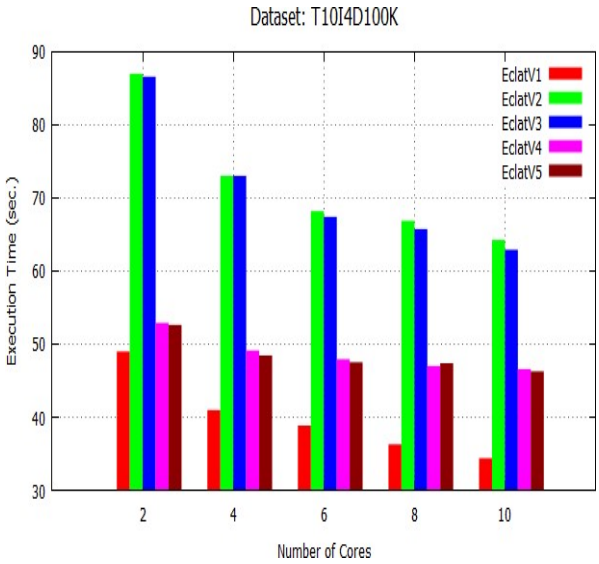
(a) Dataset chess at min_sup = 0.60



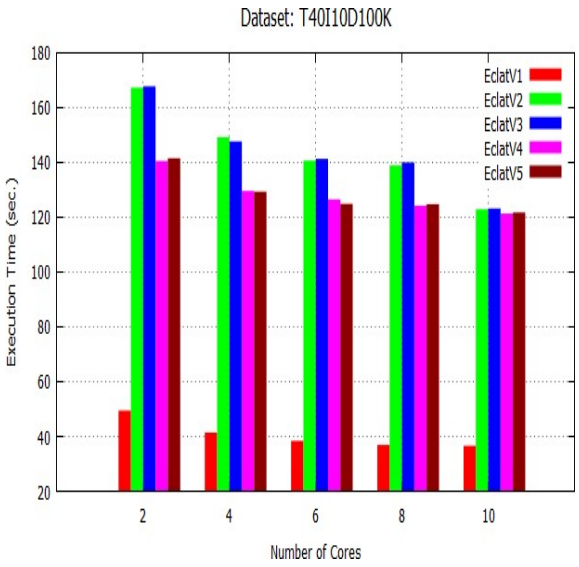
(b) Dataset mushroom at min_sup = 0.15



(c) Dataset BMS_WebView_2 at min_sup = 0.001



(d) Dataset T10I4D100K at min_sup = 0.01



(e) Dataset T40I10D100K at min_sup = 0.01

Figure 6.15 (a – e): Execution time on varying number of executor cores for five datasets.
Execution times were determined for the five datasets utilising 2, 4, 6, 8, and 10 executor cores. The execution time of algorithms lowers as the number of cores increases. The reduction is more pronounced for algorithms that take a longer time than for those that take a shorter time (Figure 4.15(a – e)). This shows that execution time can be decreased or maintained by assigning additional cores or nodes.

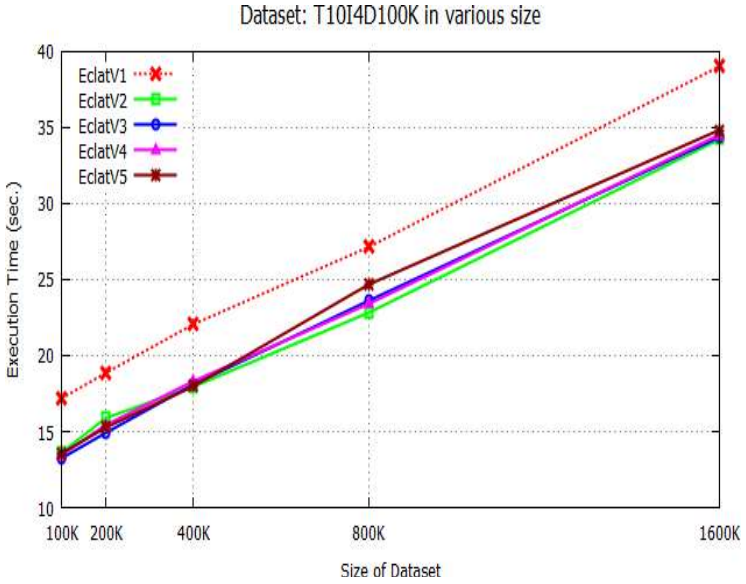


Figure6.16: Execution time on increasing size of dataset T10I4D100K at min_sup = 0.05

5.CONCLUSION

The purpose of this article is to discuss the challenge of developing scalable and efficient frequent itemet mining algorithms for use in big data analytics. The specific and as-yet-unexplored problem at hand is re-designing the Eclat algorithm for use in the Spark distributed computing environment. We devised and implemented five variations of the Eclat algorithm using the Spark RDD framework, dubbed RDD-Eclat. EclatV1 is the earliest version, followed by EclatV2, EclatV3, EclatV4, and EclatV5. Each version is the consequence of a different technique and heuristic being applied to the preceding variant. Following EclatV1, the filtered transaction technique is used, followed by heuristics for equivalence class partitioning in EclatV4 and EclatV5. EclatV2 and EclatV3 are slightly different algorithmically, as are EclatV4 and EclatV5. Experiments on synthetic and real-world datasets.

Ethical Standards statements

I. Ethical Approval:
This paper is not Ethical Approval**.**

II. Funding Details:
There are no funding details available.

III. Conflict of Interest:
The authors declare that they have no known competing for financial interests or personal relationships that could have influenced the work reported in this paper.

- ✓ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:
- ✓ The Research work has been carried out utilizing R& D Lab setup with the Department of Computer Science and Engineering, Osmania University, Hyderabad, Telangana, India.

IV. Informed Consent
There is no Informed Consent.

6.REFERENCES

[1] Fournier-Viger P., Lin C. W., Gomariz A., Gueniche T., Soltani A., Deng Z., Lam H. T., The SPMF Open-Source Data Mining Library Version 2, Proc. 19th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2016) Part III, Springer LNCS 9853, pp. 36-40, 2016.

[2] Ghemawat S., Gobioff H., and Leung S. T., The Google filesystem, in ACM SIGOPS operating systems review, vol. 37, no. 5, pp. 29-43, October 2003.

[3] Han E. H., Karypis G., and Kumar V., Scalable Parallel Data Mining for Association Rules, in Proceedings of ACM Conf. on Management of Data, ACM Press, New York, pp. 277–288, 1997.

[4] Han J., Pei J., and Yin Y., Mining Frequent Patterns without Candidate Generation, in ACM SIGMOD International Conference on Management of Data, vol. 29, no. 2, pp. 1-12, 2000.

[5] Han J., and Kamber M., Chapter 1 and Chapter 5, Data Mining: Concepts and Techniques, Morgan Kaufmann Publishers, Elsevier, 2006.

[6] Hu H., Wen Y., Chua T.S., and Li X., Toward scalable systems for big data analytics: A technology tutorial, IEEE access, 2, pp.652-687, 2014.

[7] IBM Big Data & Analytics Hub, The Four V's of Big Data, http://www.ibmbigdatahub.com/infographic/four-vs-big-data, accessed on Sept. 2015.

[8] Jen T. Y., Marinica C., and Ghariani A., Mining Frequent Itemsets with Vertical Data Layout in MapReduce, in: International Workshop on Information Search, Integration, and Personalization, Springer International Publishing, pp. 66- 82, 2016.

[9] Jones M. T., Distributed data processing with Hadoop, IBM developer Works, 2010, http://www.ibm.com/developerworks/library/l-hadoop-1/, accessed on Sept. 2015.

[10] Kerzner M., and Maniyam S., Hadoop Illuminated, 2013, available online at, https://github.com/hadoop-illuminated/hadoop-book, accessed on Sept. 2015.

[11] Kovacs F., and Illes J., Frequent Itemset Mining on Hadoop, in Proceedings of IEEE 9th International Conference on Computational Cybernetics (ICCC), pp. 241-245, 2013..

[12] Lee K. H., Lee Y. J., Choi H., Chung Y. D., and Moon B., Parallel data processing with MapReduce: a survey, ACM SIGMOD Record, vol. 40, no. 4, pp. 11-20, 2011.

[13] Li H., Wang Y., Zhang D., Zhang M., and Chang E.Y., PFP: Parallel FP- Growth for Query Recommendation, in Proceedings of the ACM Conference on Recommender System, pp. 107-114, 2008.

[14] Li N., Zeng L., He Q., and Shi Z., Parallel Implementation of Apriori Algorithm based on MapReduce, in Proceedings of 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), IEEE, pp. 236-241, 2012b.

[15] Lin M. Y., Lee P. Y., and Hsueh S. C., Apriori-based frequent itemset mining algorithms on MapReduce, in Proceedings of 6th International Conference on Ubiquitous Information Management and Communication (ICUIMC '12), ACM, Article 76, 2012.

[16] Liu J., Wu Y., Zhou Q., Fung B.C.M., Chen F., Yu B., Parallel Eclat for Opportunistic Mining of Frequent Itemsets, In: Database and Expert Systems Applications, Lecture Notes in Computer Science, Springer, Vol. 9261, pp. 401-415, 2015.

[17] Moens S., Aksehirli E., and Goethals B., Frequent Itemset Mining for Big Data, in Proceedings of IEEE International Conference on Big Data, pp. 111- 118, 2013.

[18] Mueller A., Fast Sequential and Parallel Algorithms for Association Rule Mining: A Comparison, Tech. Report CS-TR-3515, Univ. of Maryland, College Park, Md., 1995.

[19] Padhy R. P., Big Data Processing with Hadoop-MapReduce in Cloud Systems, International Journal of Cloud Computing and Services Science (IJ- CLOSER), vol. 2, no. 1, pp. 16-27, 2013.

[20] Park J. S., Chen M., and Yu P. S., Efficient Parallel Data Mining for Association Rules, in Proceedings of ACM Int'l Conf. on Information and Knowledge Management, ACM Press, New York, pp. 31–36, 1995a.

[21] Park J. S., Chen M. S., and Yu P. S., An Effective Hash-Based Algorithm for Mining Association Rules, SIGMOD ACM, vol. 24, no. 2, pp. 175-186, 1995b.

[22] Pei J., Han J., Lu H., Nishio S., Tang S., and Yang D., H-mine: Hyper- structure mining of frequent patterns in large databases, In Proceedings of IEEE International Conference on Data Mining (ICDM 2001), pp. 441-448, 2001.

[23] Pérez M. S., Sánchez A., Herrero P., Robles V., and Peña J. M., Adapting the Weka Data Mining Toolkit to a Grid Based Environment, in AWIC, Springer-Verlag Berlin Heidelberg LNAI 3528, pp. 492–497, 2005.

[24] Pramudiono I., and Kitsuregawa M., Parallel FP-growth on PC cluster, in Advances in Knowledge Discovery and Data Mining, Springer Berlin Heidelberg, pp. 467-473, 2003.

[25] Pujari A. K., Data Mining Techniques, Fourth Edition, Universities Press, 2017.

[26] Qiu H., Gu R., Yuan C., Huang Y., YAFIM: A parallel frequent itemset mining algorithm with Spark, in Proceedings of the IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW), pp. 1664- 1671, 2014.
[27] Rathee S., Kaul M., and Kashyap A., R-Apriori: an efficient apriori based algorithm on spark, in Proceedings of the 8th Ph. D. Workshop in Information and Knowledge Management, ACM, pp. 27-34, 2015
[28] Rajak A., and Gupta M. K., Association Rule Mining: Applications in Various Areas, in International Conference on Data Management Ghaziabad, pp. 3- 7, 2008.
[29] Rathee S., and Kashyap A., Adaptive-Miner: an efficient distributed association rule mining algorithm on Spark, Journal of Big Data , vol. 5, no. 1, p. 6, 2018.RDD Programming Guide, https://spark.apache.org/docs/latest/rdd-programming-guide.html, online accessed on December 2018.
[30] Riondato M., DeBrabant J. A., Fonseca R., and Upfal E., PARMA: A Parallel Randomized Algorithm for Approximate Association Rules Mining in MapReduce, in Proceedings of the 21st ACM international conference on Information and knowledge management, pp. 85-94, 2012.
[31] Sethi K. K., Ramesh D., HFIM: a Spark-based hybrid frequent itemset mining algorithm for big data processing, The Journal of Supercomputing, 73(8), pp. 3652-3668, 2017.
[32] Shi X., Chen S., and Yang H., DFPS: Distributed FP-growth algorithm based on Spark, 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Chongqing, pp. 1725- 1731, 2017.