Research Article

Anup Bhat Brahmavar, Harish Sheeranalli Venkatarama*, and Geetha Maiya

PUC: parallel mining of high-utility itemsets with load balancing on spark

https://doi.org/10.1515/jisys-2022-0044
received June 21, 2021; accepted March 06, 2022

Abstract: Distributed programming paradigms such as MapReduce and Spark have alleviated sequential bottleneck while mining of massive transaction databases. Of significant importance is mining High Utility Itemset (HUI) that incorporates the revenue of the items purchased in a transaction. Although a few algorithms to mine HUIs in the distributed environment exist, workload skew and data transfer overhead due to shuffling operations remain major issues. In the current study, Parallel Utility Computation (PUC) algorithm has been proposed with novel grouping and load balancing strategies for an efficient mining of HUIs in a distributed environment. To group the items, Transaction Weighted Utility (TWU) values as a degree of transaction similarity is employed. Subsequently, these groups are assigned to the nodes across the cluster by taking into account the mining load due to the items in the group. Experimental evaluation on real and synthetic datasets demonstrate that PUC with TWU grouping in conjunction with load balancing converges mining faster. Due to reduced data transfer, and load balancing-based assignment strategy, PUC outperforms different grouping strategies and random assignment of groups across the cluster. Also, PUC is shown to be faster than PHUI-Growth algorithm with a promising speedup.

Keywords: high utility itemset mining, apache spark, big data analytics, MapReduce, load balancing

MSC 2020: 68T09, 68T35

1 Introduction

The era of Big Data has been ushered in not so much due to the availability of heterogeneous data from ubiquitous devices, but due to the technological advances that have led to better network availability, fault-tolerant storage devices, and efficient computing solutions to process it. Big Data Analytics (BDA) processing engine is largely fuelled by techniques that are at the intersection of machine learning and data mining, often computations being carried out in distributed environments. One such data mining technique, Association Rule Mining (ARM), aims at discovering hidden, non-trivial inter-dependencies between customer purchases from the voluminous customer transaction database of a retail store. It has seen a plethora of applications and has evolved to be a formal, predictive, and exploratory BDA task. According to a prediction by Gartner, "By 2020, more than 40% of all data analytics projects will relate to an aspect of customer experience" [1]. Hence, retail industries that adopt predictive analytics using pattern mining techniques are shifting towards distributed solutions for processing the herculean data and extract patterns for providing an improved customer experience.

* **Corresponding author: Harish Sheeranalli Venkatarama,** Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, Karnataka, India, e-mail: harish.sv@manipal.edu
Anup Bhat Brahmavar: Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, Karnataka, India, e-mail: bhatanupb@gmail.com
Geetha Maiya: Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, Karnataka, India, e-mail: geetha.maiya@manipal.edu

At the core of ARM lies the phase of mining frequent itemsets. Essentially, any association rule $X \longrightarrow Y$ is deemed to be interesting if the probability of buying an itemset Y, provided another itemset X has been already purchased, is atleast c. This measure is called the confidence of the rule. Also, not all combinations of items are significant. Only those that are deemed to have a frequency/support count of atleast s are considered for the formation of such rules. Frequent Itemset Mining (FIM) has been applied in various domains such as information retrieval [2,3], bioinformatics [4,5], pharmacovigilance [6,7], and so on to discover rules of statistical significance. However, a major drawback of the FIM model is that it does not take into account the quantity or the unit profit of the items purchased in a transaction. On the contrary, as per the support measure, it solely relies on whether the item is present in a transaction. To discover profitable itemsets, mining HUIs has emerged.

Although several algorithms to mine HUIs exist, they are sequential [8–12]. Recently, a single-phase algorithm, namely, SPUC (Single-Phase Utility Computation), using a compressed tree structure in conjunction with a prefix tree was proposed [13]. While the prefix tree called Utility Count Tree (UCT) guided the mining process, the transaction-compressed String Utility Tree (SUT) facilitated the utility computation without an additional database scan. However, the sequential algorithms scale poorly as the volume of the transaction database increases. In the past, a few multi-threaded solutions have been proposed [14,15]. However, the centralised shared memory architecture again poses a bottleneck. Recently, parallel computing frameworks that leverage data processing using a cluster of commodity hardware such as the likes of MapReduce (MR) and Spark have resulted in a paradigm shift for mining itemsets of interest from transaction databases. Further, enterprises leverage these frameworks through PaaS via cloud to exploit the elasticity and horizontal scalability that it has to offer.

Designing and developing algorithms in Big Data environment through computing paradigms such as MapReduce [16] or in-memory computing using resilient distributed datasets [17] is not straight forward. While the frameworks such as Apache Hadoop [18] and Apache Spark [19] (their respective open-source implementations) offer a rich set of APIs in popular programming languages, a meticulous design that translates the sequential algorithms to a cluster environment must consider such factors as load balancing, and communication cost. The data transfer incurs network overhead, especially when shuffling operations are used. This can have a significant impact on the overall mining performance. To alleviate this, several studies incorporate clustering of items based on transaction similarity before assigning their respective search-space across cluster nodes. This reduces the vertical splits of the projected database due to which the shuffling bottleneck is overcome. However, not many distributed HUIM algorithms group the promising items based on transaction similarity prior to search-space assignment. Another factor that contributes to the mining performance is the strategy employed for assigning the items/groups to different nodes for mining. Random assignments often ignore the mining load resulting in an uneven distribution of workload across the cluster, thus affecting the mining performance.

In the context of mining HUIs, clustering the transactions merely based on the presence or absence of items is not sufficient. This is due to the fact that each item is now associated with a purchase quantity in a transaction. Furthermore, clustering overhead should be kept minimal as the end goal is mining HUIs. In this regard, the current study explores to group the items by considering the Transaction Weighted Utility (TWU) values as a degree of transaction similarity. While this minimizes the transaction splits during partition, strategic placement of such groups across cluster nodes is also essential. To address this issue, a prefix length-based workload estimation is designed. Given this, contributions of the study are as follows:

- A parallel adaptation of the sequential Single-Phase Utility Computation (SPUC) algorithm [13] called Parallel Utility Computation (PUC) for mining HUIs is proposed. PUC develops a search-space split-based parallel workflow for utility computation using SPUC on Spark cluster.
- Jenks Natural Breaks Optimization (JNBO) is employed for grouping the sequence of TWU value of items. TWU is a measure that accounts for the weighted presence of an item in different transactions. It sums up the transaction utilities across the supporting transactions. Hence, items that co-occur fall into similar groups. Experimental evaluations have demonstrated that TWU-JNB grouping strategy effectively reduces the execution time of PUC in comparison to the grouping performed by either arranging the items in ascending or descending order of TWU.

– For load balancing, computation load across nodes is proportional to the mining load. This is estimated as a function of length of the prefix path of items. Specifically, the number of subsets enumerated in SPUC is determined. This load is utilized for a balanced assignment of the groups across the available nodes of the cluster prior to mining. The proposed algorithm called parallel utility computation with vertical partitioning using JNB of TWU and prefix-length-based load balancing is shown to be efficient in terms of execution time as against the round-robin assignment of groups to nodes. Furthermore, PUC displays superior performance in comparison to PHUI-Growth algorithm [20].

The rest of this article is organised as follows: Section 2 introduces the preliminary concepts of high utility itemset mining. A summary of the studies relevant to mining frequent itemsets and HUIs using distributed frameworks is provided in Section 3. The overall flow of PUC algorithm is provided in Section 4 with JNB-based grouping and load balancing strategies in Sections 4.2.1 and 4.2.2, respectively. Experimental evaluation that compares the effectiveness of grouping and load balancing strategies along with relative speed up is demonstrated in Section 5. This article concludes with Section 6.

2 Background

Given a transaction database D with n distinct items, $I = \{i_1, i_2, \ldots, i_n\}$, each transaction T_d in D is identified by *TID*, the transaction identifier records a collection of items purchased along with its quantity or internal utility as shown in Table 1(b). An ordered pair (i, q) in each transaction indicates that item i was purchased in q quantities in that transaction. Each item is also associated with unit profit or external utility as shown in Table 1(a).

Table 1: Sample database

(**a**) **Profit Table**

Item	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Profit	5	1	2	1	3	1	10	5	2	4

(**b**) **Transaction table**

TID	**Transaction**
T_1	{(1,7)(2,5)(3,1)(8,9)(10,1)}
T_2	{(3,1)(4,5)(5,5)(7,2)}
T_3	{(4,3)(7,1)(8,10)(9,2)(10,1)}
T_4	{(1,6)(2,3)(3,5)(6,7)(7,1)}
T_5	{(5,3)(6,1)(8,1)(10,1)}

Definition 2.1. (Utility of an item) Utility of an item i in transaction T_d, denoted by $u(i, T_d)$, is measured as the product of quantity q and unit profit $p(i)$.

Definition 2.2. (Utility of an itemset) Utility of an itemset X in transaction T_d, denoted by $u(X, T_d)$, is defined as $\sum_{i \in X \wedge X \subseteq T_d} u(i, T_d)$.

Definition 2.3. (Utility of an itemset in database) Utility of an itemset X in D denoted by $u(X)$ is defined as

$$u(X) = \sum_{X \subseteq T_d \wedge T_d \in D} u(X, T_d) \tag{1}$$

e.g.,

$u(\{4\}, T_2) = 5 \times 1 = 5$

$u(\{4, 7\}, T_2) = u(\{4\}, T_2) + u(\{7\}, T_2) = 5 + 20 = 25$

$u(\{4, 7\}) = u(\{4, 7\}, T_2) + u(\{4, 7\}, T_3) = 25 + 13 = 38$.

Utility measure is neither anti-monotone nor monotone. Table 2 compares the utility for few subsets of {4, 7, 8}. Although the support is strictly increasing across the subsets, the utility is neither increasing nor decreasing. Hence, while the support of an itemset is downward closed, the utility measure is not.

Table 2: Support vs utility

Itemset	Support	Utility
{4, 7, 8}	1	63
{4, 7}	2	38
{7}	3	40

Definition 2.4. (Transaction Utility [TU]) Transaction utility of a transaction T_d, denoted by $TU(T_d)$ is defined as the sum of the utilities of all the items in that transaction i.e., $TU(T_d) = \sum_{i \subseteq T_d} u(i, T_d)$. Table 3 records TU for transactions in Table 1(b).

Table 3: TU for transactions in Table 1

TID	TU
T_1	91
T_2	42
T_3	71
T_4	60
T_5	19

Definition 2.5. (High Utility Itemset) An itemset X is called a high-utility itemset (HUI) if $u(X) \geq min\_util$, where *min_util* is the minimum utility provided by the user. For example, if the threshold is set to 30%, then $min\_util = 0.30 \times 283 = 84.9$, where 283 is the sum of TU of all the transactions in the sample database. Then, HUIs for this threshold are {8}, {8, 10}, {1, 2, 3}, {1, 2, 8}, {1, 2, 3, 8}, {1, 2, 8, 10}, {1, 2, 8, 10}, and {1, 2, 3, 8, 10} with utilities of 100, 112, 85, 85, 87, 89, and 91, respectively.

Table 4: TWU for items in Table 1

Item	TWU
1	151
2	151
3	193
4	113
5	61
6	79
7	173
8	181
9	71
10	181

Definition 2.6. (Transaction Weighted Utility [TWU]) Transaction weighted utility of an itemset X, denoted by $TWU(X)$, is defined as the sum of the transaction utility of all the transactions in D that contain X, i.e., $TWU(X) = \sum_{X \subseteq T_d \wedge T_d \in D} TU(T_d)$. The TWU for items using TU in Table 3 is calculated and provided in Table 4.

Definition 2.7. (High-Transaction Weighted Utility Itemset) An itemset X is a high-transaction weighted utility itemset (HTWUI), if $TWU(X) \geq min\_util$. Also, if an itemset X is not a HTWUI, then it cannot be a HUI.

Definition 2.8. (Property 1. [TWU Downward Closure Property]) If an itemset X is a HTWUI, then all its subsets are HTWUIs or if an itemset X is not a HTWUI, then none of its supersets can be HTWUIs. For example, $TWU(\{5, 6\}) = 19 < min\_util$. Hence, any higher order itemset need not be enumerated from $\{5, 6\}$ as this property ensures they shall be neither HTWUI nor HUI.

In the current study, the goal is to discover all the HUIs from a given transaction database D that satisfy a user-defined threshold or minimum utility min_util. As the database volume grows, distributed storage and computing frameworks render mining HUIs feasible. By considering TWU as a degree of transaction similarity, the proposed algorithm groups the co-occurring items for mining at a single node. Also, a prefix length-based load estimation is determined for each group to facilitate their suitable assignment across the cluster nodes.

3 Literature survey

Parallel mining of itemsets has been implemented in both shared memory and distributed computing architectures. However, scalability remains a major issue with shared memory architecture. In this regard, distributed computing solutions, namely, Apache Hadoop and Spark, not only address the data storage issue through a fault-tolerant Hadoop Distributed File System (HDFS) but also provides for utilities that process the data in a real-time manner via MapReduce engine. In this section, a few algorithms for mining frequent itemsets along with high utility itemsets in the aforementioned frameworks are discussed.

Apriori, FP-Growth, and ECLAT are the major centralized algorithms that have been widely adapted for FIM using distributed computing frameworks. In [21], three Apriori adaptations was proposed using MapReduce framework. Single Passes Counting (SPC) algorithm involves multiple phases of MapReduce to count the support and outputs frequent k-itemsets at the end of the kth phase. To reduce the phases that involve disk I/O, Fixed Passes Combined-counting (FPC) that generates candidates of higher orders in a single phase was proposed. However, this burdened certain worker nodes with numerous candidates to process. Hence, the third approach in this study called the Dynamic Passes Combined-counting (DPC) dynamically adjusted the extent to which the higher order candidates are to be generated in a given phase based on the execution time of the previous phase. PApriori (parallel Apriori) [22] was proposed in lines similar to SPC. YAFIM (Yet Another Frequent Itemset Mining) proposed by Qiu et al. [23] on Spark has been reported to be 25 times faster than PApriori.

Parallel FP-Growth algorithms have been developed using both Apache Hadoop and Spark frameworks. PFP (Parallel FP-Growth) [24] on Hadoop and DFPS (Distributed FP-Growth on Spark) [25] on Spark are well-known implementations of FP-Growth. In the first MR phase, the frequent 1-items are identified and stored in a F-list. In the second, the items in the F-list are grouped and the local projections of items are generated for each item taking into account the grouping information. At the end of this phase, a projected database that contains the CPB of all the items belonging to the group is available at a single node as aggregated by the reduce operation. In the third phase, independent FP-trees are constructed and mined as per the FP-Growth algorithm [26].

Since the implementations of sequential algorithms in distributed environments have been proposed, several challenges have required attention for enhancing the mining performance [27,28]. One of the major challenges is reducing the shuffling cost at the end of the second MR phase. For example, The *reduceByKey* () incurs a significant data movement across cluster nodes when local projections of items in the group are generated. This intermediate data incur network overhead, adding to the overall execution time. Another major challenge is balancing the workload across the cluster nodes. Recent studies such as FiDoop-DP [29],

HBPFP-DC [30], and BIGMiner [31] have sought to address these issues. FiDoop-DP proposes to consider transaction similarity to reduce the shuffling cost via LSH (Locality Sensitive Hashing) of items in *F*-list to groups such that bottleneck due to shuffling can be alleviated. In addition to this, HBPFP also considers prefix length-based approximation of mining load of items in the group for addressing the workload skewness. A few studies that evaluated several popular distributed FIM algorithms focussed on the need to develop suitable assignment strategies of independent subproblems, i.e., CPBs for better load balancing and thus an enhanced mining performance.

Popular choices for parallelizing HUIM algorithms include the two-phase [8], HUI-Miner [10], and EFIM [12]. Their counterparts are PHUI-Growth [20], PHUI-Miner [32], and EFIM-Par [33], respectively. Similar to the distributed FIM algorithms, these algorithms work in three phases. In the first phase, mappers and reducers are employed to determine $\langle item, TWU \rangle$ pairs or *TWU-list* for every item in the transaction database. Transactions are then re-organised to reflect the ascending order of the items as per their TWUs. With another MR phase, node data are generated in the case of PHUI-Miner and EFIM-Par. The node data includes part or a complete transaction that contains all the itemsets that can be mined from the item(s) assigned to the node. The items that form the part of the item's search-space are as determined from search space set enumeration tree. Eventually, with an additional MR phase, each node mines HUIs it is responsible for. However, PHUI-Growth iteratively mines the itemsets in a level-order manner such that after every kth iteration, all the k-HUIs are output. Recently, using the pruning strategies proposed in EFIM and the concepts of fuzzy set theory, fuzzy HUIs were mined using Hadoop framework [14]. The authors employed Apache HBase for distributed storage and Apache Avro for mining in three MapReduce phases.

Assigning the search-space corresponding to every item in *TWU-list* contributes to the overall mining performance. In PHUI-Miner, a circular assignment of items to nodes was proposed. While pFHM+ [34] (that parallelizes FHM+), and EFIM-Par also employ this strategy, a few studies have been proposed to modify the assignment [35,36] for an enhanced mining performance. While, ref. [36] proposes to cluster the transactions, in [35], the sub-space size corresponding to each item and a utility upper-bound constraint was used for a better workload distribution across the cluster. Although studies such as [29,36] propose to cluster the transactions, the clustering overhead can be large. Further, not many studies apart from EFIM-Par and PHUI-Growth mine the complete set of HUIs in a distributed manner.

3.1 Differences from previous works

At the very outset, the current study develops a parallel workflow to adapt SPUC [13] for mining HUIs in a distributed environment. Although SPUC mines HUIs in a single phase, scalability remains a major issue. With the integration of distributed storage and in-memory computation provided by HDFS and Apache Spark, PUC scales better than SPUC. Further, PUC mines a complete set of HUIs unlike PHUI-Miner. In contrast to PHUI-Miner that parallelises HUI-Miner, which is a list-based algorithm, PUC parallelises trees-based SPUC algorithm. Apart from distributed FIM algorithms, not many distributed HUIM algorithms employ transaction clustering and load balancing prior to the mining task. PHUI-Miner and EFIM-Par adopts a round-robin assignment strategy. However, PUC determines the workload due to every item for a balanced assignment. While HBPFP-DC considers a prefix path length-based workload approximation, PUC estimates it as proportional to the number of subsets in the prefix path of an item. Also, the items in the *TWU-list* are clustered using Jenks Natural Breaks algorithm [37] to obtain groups of items such that the vertical splits produced during the second phase are significantly reduced.

4 Parallel utility computation

In this section, PUC algorithm that parallelises sequential SPUC algorithm using Apache Spark is proposed. Before providing the details of this, a brief overview of SPUC is provided first.

4.1 Overview of SPUC

SPUC algorithm leverages two compact tree structures viz., Utility Count Tree (UCT) and String Utility Tree (SUT), for efficiently mining high-utility patterns in a single phase. Both the tree structures are constructed with a single database scan without discarding any items. After arranging the items of a transaction in ascending order, UCT stores in its node the item, the count, and the utility along the path in the tree. SUT is also a prefix tree structure. However, it captures entire transaction information in a node by concatenating the item and utility values as string separated by a delimiter. For more details about the construction procedure, the reader can refer [13]. Figures 1 and 2, respectively, denote UCT and SUT for the sample database 1.

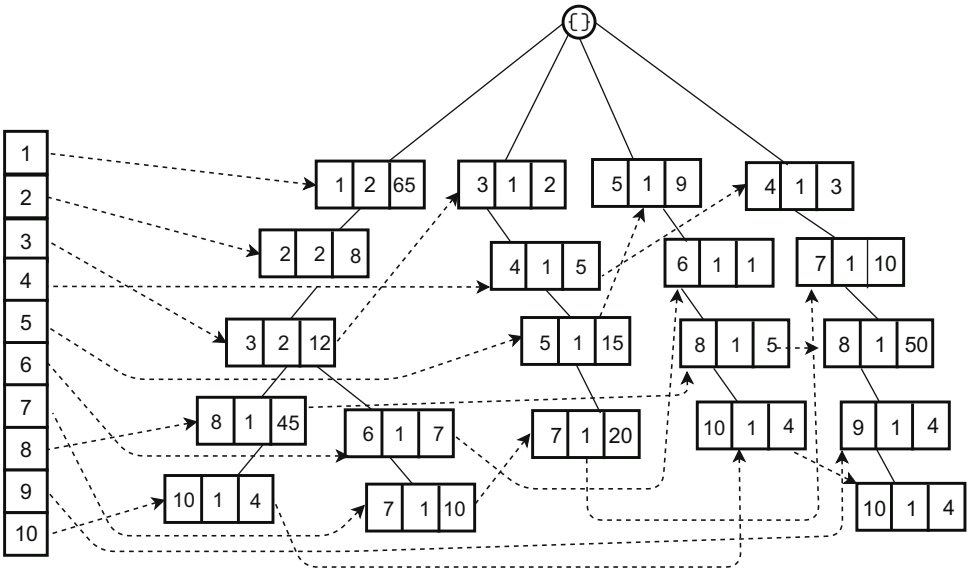

Figure 1: Utility Count Tree for database Table 1.

Algorithm 1 Single-phase utility computation algorithm – mining using UCT and SUT

Input: *UCT*, *SUT*, *minUtil*

HashMap hash_item_utilities(*List_Integer itemset, Integer utility*)
for each item i from the bottom of H **do**
  $CPB(i) \leftarrow$ Get all the prefix paths of i and calculate path utility
  if sum of the path utilities >= *minUtil* **then**
    foreach prefix path, $p \in CPB(i)$ **do**
      itemset_list $\leftarrow$ generate subsets from the items in p that include i
      itemset_list $\leftarrow$ *itemset_list*$\setminus$*subset_List*
      subset_List $\leftarrow$ *subset_List* $\bigcup$ *itemset_list*
      foreach itemset X in *itemset_list* **do**
        if $OU(X) <$ *minUtil* **then**
          itemset_list $\leftarrow$ *itemset_list*$\setminus X$
        end if
      end for
      Call MINE(*itemset_list*, *SUT*. *root*)
    end for
  end if
end for

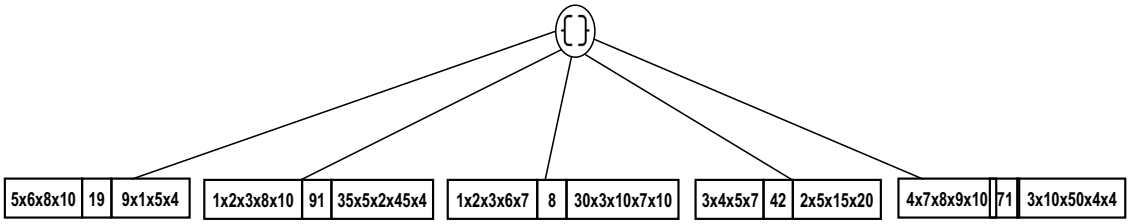

Figure 2: String Utility Tree for database 1, the items and utilities are delimited by "x."

The SPUC mining process begins by traversing the header list of the UCT from the bottom and collecting the prefix paths of each item (*line* 2 & 3) as shown in Algorithm 1. This forms the CPB of the item. *Path Utility* based pruning strategy is employed here to discard the CPB if the sum of the path utilities is not at least *min_util* (*line* 4). If the item passes this test, then its CPB is examined for enumerating subsets of items from every path (*line* 6). As the utility of the item stored in the node of the UCT is an overestimated value, *Overestimated Utility* based pruning is applied to further discard the enumerated itemsets (*line* 10). The utilities of the remaining itemsets are then obtained by traversing the SUT (*line* 14). The details pertaining to the pruning strategies and accessing SUT for utility computation can be found in ref. [13]. Since SUT stores the utility information in a compact manner, no additional database scan is required for utility computation. CPB of an item denotes the transaction database conditioned on the item under consideration. Upon enumerating the subsets that contain this item from different transactions or paths in the CPB, all the valid itemsets that participate with this item can be obtained, and thus, it is sufficient to calculate their utilities. Intuitively, CPB examination of different items can be performed in parallel provided the different paths that form the CPB are made available at a particular node. With this rationale, MR phases of Spark are employed to mine across a cluster of nodes by partitioning the database and shuffling the projections at required nodes. As mining at a node depends on the CPB to be examined, the proposed algorithm takes into account the mining complexity for estimating the workload. Apart from this, to mitigate the network overhead during shuffling, grouping of TWU is also incorporated in the parallel version.

4.2 PUC

The proposed algorithm works in three phases to mine HUIs. The parallel workflow is displayed in Figure 3. The first phase determines TWU of the items, the second phase splits the transaction database vertically based on the grouping strategy, and in the third phase, mining is carried out on the projected database. The steps performed in these phases are as follows:

- During the first phase, the transaction database stored in different shards of HDFS is read and the TWU values of the items are determined. This requires a single Map-Reduce (MR) phase. The *map*() operation reads each transaction T, and outputs <*item*, *TU*> pairs for every *item* in T. The *reduce*() operation sums the *TU* value for every item. As the *item* − *TWU* list is small enough to fit in the memory of a single node, the *collectAsMap*() Spark operation collects this list as a global map/dictionary where *item* is the key and its corresponding TWU is the value.
- Now, the TWU values are grouped using Jenks Natural Breaks algorithm. The details of this procedure is provided in Section 4.2.1. The items corresponding to the TWU values in every group is then determined and stored in a global dictionary, *item_group*. These disjoint groups of items are then assigned to different nodes for mining.
- Mining requires Conditional Pattern Base (CPB) of every item to be present at the node where the group is assigned to. In this regard, the database is scanned once again, and each transaction is projected based on the group the item is present in. The projected database represents the collection of prefix paths of all the items present in that group. Algorithm 2 denotes the steps in generating the projections for T using the grouping information stored in *item_group* dictionary. As T is scanned in the reverse order (*line 2*),

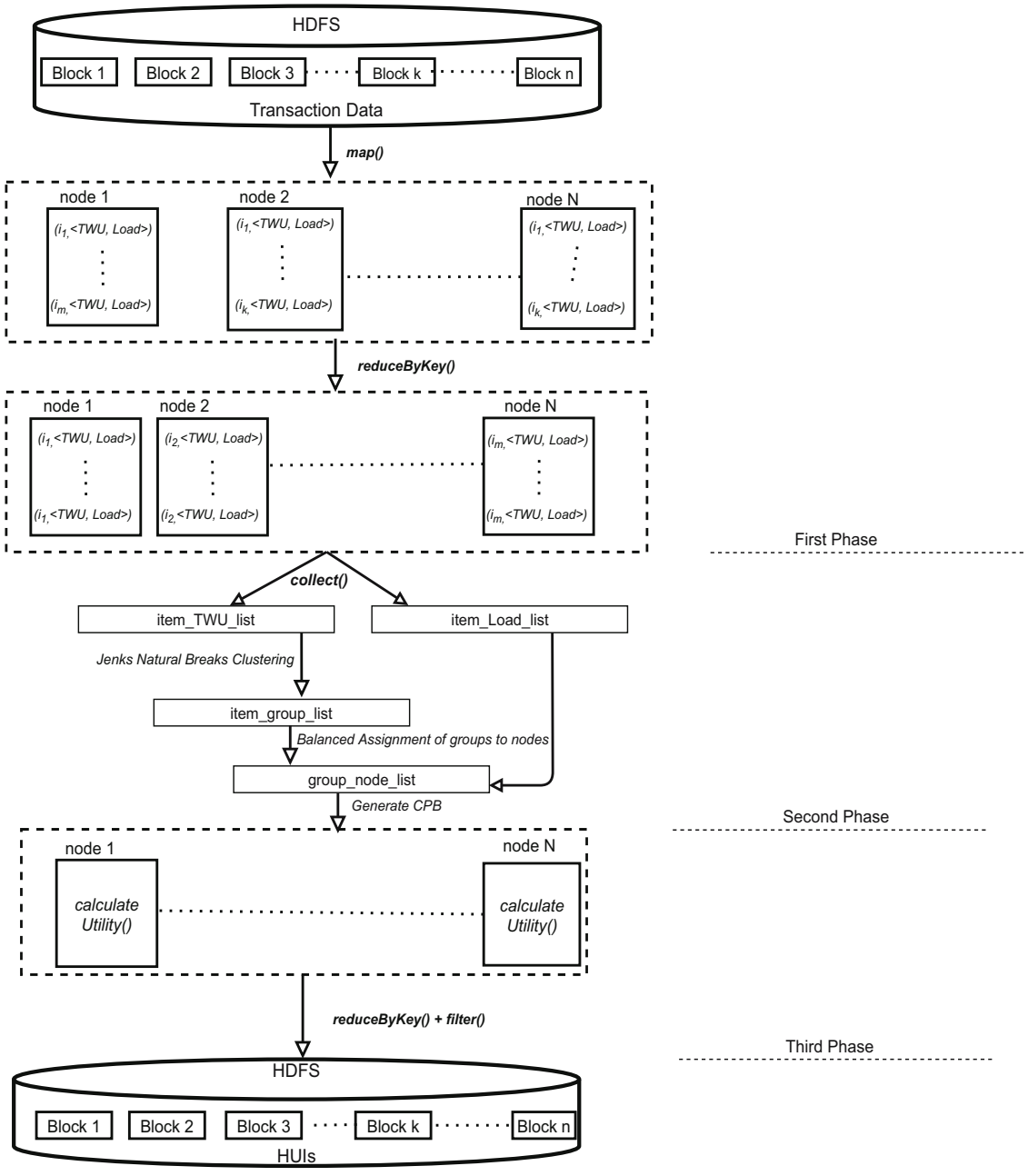

Figure 3: Flow of parallel utility computation algorithm.

the group corresponding to each item is obtained from *item_group* (*line 3*). The prefix path *Pathi* for an item *i* includes *i* and all the items preceding it and output along with the *groupId* (*line 6*). To keep track of the items in *T* that map to the same group, a *group_list* stores the resolved *groupId*s (*line 4*). Hence, for any item $j \in Path_i$ no prefix paths shall be output provided *i* and *j* belong to the same group. While *map*() operation employs this algorithm to generate the prefix paths, the *reduce*() operation accumulates the paths corresponding to a group into a single node. Using this projected database, the CPB for every item in the group can be determined aiding the subsequent utility computation.

– In the third phase, Algorithm 3 provides the steps for the utility computation. Essentially, a *map*() operation is applied to the <*groupId*, *DB′*> obtained after the second scan of the database that splits the database vertically so that each group gets the projected database containing the prefix paths of all the items corresponding to the group. The *suffix _items_list* identifies all the items that belong to the

group with id *groupId*. These are the items whose CPBs are to be determined for enumerating itemsets for utility computation (*line 3*). Initially, the path utility corresponding to these suffix items are determined (*line 4 to line 12*). The utility of all the items preceding the suffix item in a path contributes to the path utility (*line 8*). The accumulated value is updated in a *suffix_pu* dictionary. The suffix items whose path utility is not atleast *min_util* are discarded. The details of the path utility based pruning is provided in ref. [13]. Following this, the subsets of the items from every path are enumerated. Specifically, those subsets that contain the suffix items are required for utility computation (*line 15*). As the item and utility are present in the path and do not get modified during any of the previous MR phases, the utility can be directly obtained for the itemsets enumerated (*lines 16 to 18*).

Algorithm 2 Algorithm for the generation of local CPB by transaction partitioning

Input: Transaction T, HashMap *item_group*
Output: (*groupId, prefix_path*)

1: $group\_list = \phi$
2: **for** $i \leftarrow T.\ length - 1, 0$ **do**
3: **if** $item\_group.\ Get(T[i]) \notin group\_list$ **then**
4: $group\_list.\ Add(item\_group.\ Get(T[i]))$
5: $Path_i \leftarrow T[i], T[i-1], \ldots, T[0]$
6: $out(groupId, Path_i)$
7: **end if**
8: **end for**

Algorithm 3 Utility computation algorithm

Input: Group Id *groupId*, Projected Database DB', Minimum Utility *minUtil*
Output: (*groupId, itemset_list*)

1: *itemset _utility_list* $= \phi$
2: *HashMapsuffix_pu*(*Integer item, Integer path_utility*)
3: *suffix _items_list* $\leftarrow \{i \ni item\_group.\ Get(i) == groupId\}$
4: **for each** path $p \in DB'$ **do**
5: **for each** *item_util* in p **do**
6: **if** $item \in suffix\_items\_list$ **then**
7: $k \leftarrow p.\ indexOf(item)$
8: $pu \leftarrow pu[k].\ util + pu[k-1].\ util + \ldots\ldots + pu[0].\ util$
9: $suffix\_pu.\ Put(item, pu)$
10: **end if**
11: **end for**
12: **end for**
13: *suffix _items_list* $\leftarrow \{i \ni suffix\_pu.\ Get(i) >= minUtil\}$
14: **for each** path, $p \in DB'$ **do**
15: *itemset _list*$\leftarrow$ generate subsets from the items in p that contain $i \in$ *suffix _items_list*
16: **for each** itemset X in *itemset_list* **do**
17: $X.\ util \leftarrow p[i].\ util, \forall i \in X$
18: **end for**
19: **end for**

4.2.1 Vertical partitioning using Jenks natural breaks

The items in the *item-TWU* list are collected by the driver program towards the end of the first phase. The TWU measure for items that co-occur will be the same. For example, consider the items 1 and 2 from the sample database 1. Both of these items co-occur in transactions T_1 and T_4. Thus, they have same TWU of 151. Since TWU measure is contributed by transaction utility of all the transaction the item participates, grouping TWU of items intuitively accounts for transaction similarity. To this end, Jenks Natural Breaks has been adopted to find "natural" breaks in the sequence of TWU values. JNB is an iterative approach to find classes with similar ranges such that the in-class variance is minimized and between-class variance is maximized. The following measures are adopted:

- SDAM (Sum of squared Deviation from Array Mean): Initially, the sum of squared deviations from the array mean is determined.
- SDBC (Sum of squared Deviation Between Classes): The array values are grouped into m arbitrary classes. For each class C_h, $h \in [1, m]$ SDAM is calculated. SDBC is then calculated as $\sum_{h=1}^{m} SDAM_h$.

In each iteration, the measure of SDBC is calculated after moving data points between different classes so as to maximise the goodness of variance fit (GVF), $gvf = (SDAM - SDBC)/SDAM$. Iteratively, the classes are rearranged to attain the input *gvf* (between 0 and 1) threshold for given initial number of classes.

Employing JNB, the TWU values of the items are grouped into different classes. For the running example, for initial number of classes as 2 and a *gvf* of 0.9, the TWU grouping obtained is shown in Table 5. Thus, the snapshot of *item_group* dictionary is as shown in Table 6. Using Spark's *broadcast*() operation, *item_group* is made available to all the *map*() operations that vertically split the transactions of the database as per Algorithm 2. For example, consider a *map*() operation processing transaction T_3. The transaction is processed from the tail and item 10 is encountered first. Hence, <*groupId*, $Path_{10}$> output will be <4, {4, 7, 8, 9, 10}>. The next item is 9 and the <*groupId*, $Path_9$> output is <1, {4, 7, 8, 9}>. For the next two items, i.e., 7 and 8, no output is produced as these items occur in the prefix path of previously output item 10 having the same group id of 4.

Table 5: TWU-grouping using JNB for items in database 1

Group Id	TWU
G_1	61, 71
G_2	79
G_3	113
G_4	151, 151, 173, 181, 181, 193

Table 6: Item grouping for vertical partitioning of database 1

Item	Group Id
1	4
2	4
3	4
4	3
5	1
6	2
7	4
8	4
9	1
10	4

JNB provides for a grouping of TWU values that maximises the *gvf*. Grouping of items has a significant affect on the number of times a transaction has to be split. For example, if the TWU values were arranged in a descending order and divided into four groups (Table 7), T_1 would produce two splits, <4, {1, 2, 3, 8, 10}> and <3, {1, 2}>, for such a grouping. However, with grouping due to JNB, as all the items of T_1 fall in the same group, G_4, only a single split, <4, {1, 2, 3, 8, 10}>, is output. Thus, JNB-based grouping of item reduces the number of vertical splits of the transaction database.

Table 7: TWU-grouping based on descending order for items in database 1

Group Id	TWU
G_1	61
G_2	113, 79, 71
G_3	173, 151, 151
G_4	193, 181, 181

4.2.2 Load balancing

The utility computation algorithm filters out unpromising items based on path utility. For the remaining items, the CPB is determined from the projected database, DB'. For each prefix path in the CPB of item i, the subsets of the itemsets are then generated. The number of such itemsets containing i is $2^{(l-1)}$, where $l = |\text{Path}_i|$ is the length of the prefix path being considered. This is computationally intensive, especially for large values of l and is a major component that contributes to the mining load. Hence, for efficient mining, the groups generated based on JNB should be assigned to the nodes such that the load across the computing nodes is balanced. In this regard, the load of every group L_G is determined as follows:

$$L_G = \sum_{i \in G} L_i.$$

Hence, the mining load L_i due to any item i is estimated as follows:

$$L_i = \sum_{i \subseteq T_d \wedge T_d \in D} 2^{\left(|Path_{i,T_d}|-1\right)}.$$

During the first scan of the database it is possible to determine the length of the prefix path, and hence, L_i is computed in the first MR-phase. This value is stored in another dictionary similar to *item – TWU*, *item_load*. For example, consider transaction T_1. The load of item 10 is calculated as $2^{(\text{Path}_{10,T_1})-1} = 2^{5-1} = 16$. Similarly, loads due to the other items of T_1 from the tail end shall be 8, 4, 2, and 1, respectively. At the end of the first MR-phase when TWU is determined, a separate *collectAsMap*() operation collects the loads of the items in *item_load* dictionary. The snapshot of this dictionary for sample database 1 is as shown in Table 8.

Algorithm 4 Balanced assignment of groups to nodes

Input Number of Groups N_G, Number of nodes N, *HashMap item_group, HashMap item_load*
Output *HashMap group_node*

```
HashMap group_node(Integer groupId, Integer nodeId)
HashMap node_load(Integer nodeId, Integer load)
group_count ← N_G
for i = 1 to N do
    group_node.Put(group_count, i)
    node_load.Put(i, COMPUTEGROUPLOAD(group_count))
    group_count ← group_count − 1
```

```
8:        end for
9:        for i = group_count to 0 do
10:         min_load_node ← node_load. Key(min(node_load. Values()))
11:         group_node.Put(i, min_load_node)
12:         node_load.Put(min_load_node, node_load.Get(nodeId) + COMPUTEGROUPLOAD(i))
13:       end for
14:       procedure COMPUTEGROUPLOAD (groupId)
15:         group_items ← {i∋item_group. Get(i) = =groupId}
16:         group_load ← 0
17:         for each item, i in group_items do
18:           group _load ← group_load + item_load. Get(i)
19:         end for
20:       end procedure
```

Table 8: *item_load* dictionary for database 1

Item	Load
1	2
2	4
3	9
4	3
5	5
6	8
7	26
8	16
9	8
10	40

Once the groups are determined using JNB, the groups are assigned to the available computing nodes using Algorithm 4. The task is to assign N_G groups among N nodes of the cluster ($N_G > N$) such that the load is balanced. Initially, N out of N_G groups with *groupId*s from N_G to $N_G - N + 1$ are assigned to the N nodes with *nodeId*s from 1 to N (*line 4 to 8*). A *group_node* dictionary maintains the mapping of *groupId* to *nodeId*. As the groups are assigned, the load of each group is computed as the sum of the load due to individual items of the group using the COMPUTEGROUPLOAD procedure (*lines 14 to 20*). This group load also becomes the load of the node to which the group is assigned (*line 6*) and is stored in *node_load* dictionary. The remaining $N_G - N$ groups are assigned to the N nodes by determining the node with minimum load (*lines 10 and 11*) for each assignment. Once a group is assigned to the node with the minimum load min_*load_node*, its load is updated with this new assignment (*line 12*).

Consider the assignment of $N_G = 4$ groups to $N = 3$ nodes given the *item_group* (Table 6) and *item_load* (Table 8) dictionaries for sample database Table 1. Initially, groups with Ids 4, 3, and 2 get assigned to the nodes with Ids 1, 2, and 3, respectively. The node load due to this assignment will be the load due to the items in the groups. If the load of assigning group 4 to node 1 is considered, then it is $L_4 = load(1) + load(2) + load(3) + load(7) + load(8) + load(10)$. These loads can be obtained from *item_load* in Table 8 as 2 + 4 + 9 + 26 + 16 + 40 = 97. Similarly, loads for other assignments are determined (Table 9). The remaining group with Id 1 is now assigned to the node, which has minimum load. Here, the load on node 2 is the least. Hence, group 1 is assigned to node, 2 and the load of node 2 gets updated with load of items in group 2 to reflect this new assignment. Thus, the final assignment along with updated load will be as shown in the columns *Group_final* and *Load_final* of Table 9. In contrast to this, if the groups were assigned to the nodes in a way as provided in ref. [32], the loads on nodes would be 97, 3, and 21 resulting in an unbalanced assignment of groups to the nodes.

Table 9: *node_group* and *node_load* dictionary for database 1

Node Id	*Group_initial*	*Load_initial*	*Group_final*	*Load_final*
1	4	97	4	97
2	3	3	{3, 1}	16
3	2	8	2	8

4.3 Complexity analysis

PUC mines HUIs in three phases of MapReduce. As displayed in Figure 3, JNB and load balancing (Refer Algorithm 4) algorithms are executed as part of the driver program. Thus, the most crucial phase is the second, where the CPBs are generated based on the grouping of items obtained in conjunction with the predetermined loads. In this section, the *Key Complexity* of the different phases of PUC is discussed. Although only a few studies provide the theoretical framework for determining computational complexity in a MapReduce framework, the running time and size of <*key*, *value*> pairs output by a mapper have been assessed for the proposed adaptation as per Goel and Munagala [38].

In the first phase, each mapper outputs the *TWU* and *load* for each item from the split of the database it is working on. If $f_1, f_2, \ldots f_{|I|}$ denote the frequency of items in I, then the key complexity shall be $O(f)$, where $f = max_i\ f_i$. The second phase shuffles the transaction splits as per Algorithm 2, where each mapper outputs all the unique *groupID*s. Hence, a reduce record for a group N_g with *groupId*, the size shall be upper-bounded by the frequency of item j, where $f_j = \max_i f_i(\forall j \in N_g)$. Hence, key complexity can be given as $O(f_j)$.

In the third phase, the local mining using the principles of SPUC is executed as per Algorithm 3. The details of computational complexity can be found in ref. [13]. In the driver program, the proposed assignment algorithm, Algorithm 4 runs a total of N_G number of times ($N < N_G < |I|$). Let $N_{\max}$ denote the group with maximum number of items. In a case where two or more groups have $|N_{\max}|$ number of items, then $N_{\max} = N_{C'_k}$, $k' \in [1, k]$, where max $N_{C'_k} = \max_k N_{C_i}$. The assumption here is that $k(k \in [1, N_G])$ candidate groups, i.e., $N_{C_1}, N_{C_2}, N_{C_3} \ldots N_{C_k}$, generated by JNB contain the maximum number of items. During the assignment, the complexity of computeGroupLoad needs to be accounted for. As this depends on the number of items in each group, the run time complexity shall be $O(|N_{\max}| * N_G)$.

5 Experimental evaluation

In this section, the performance of PUC algorithm in terms of the effectiveness of JNB-based TWU grouping and load balancing has been evaluated. The algorithm has been developed in Python using the Spark utilities via *PySpark* API. JNB implementation available in *jenkspy* package is adopted for the grouping of TWU values [39]. A cluster of upto 12 nodes was provisioned using the Elastic Compute Cloud (EC2) of Amazon Web Services [40]. Specifically, *r3.xlarge* type of instances that has four virtual CPUs, with 2 cores capable of running 2 threads each was selected as the type of EC2 instance. The datasets available from SPMF were adopted for comparative purposes [41]. Their characteristics is summarised in Table 10. The synthetic dataset, *s*1 has been generated using the transaction database generator provided in the SPMF toolbox. As per the literature, the quantities of items for this dataset has been generated in refs [1,10] using a uniform distribution with the unit profit values following a Gaussian distribution.

In Section 5.1, various TWU-grouping strategies have been compared. This is followed with evaluation of load balancing in PUC against different assignment strategies 5.2. Also, PUC has been compared against PHUI-Growth for a few datasets here. The performance of PUC as cluster scales horizontally, and the relative speedup is reported in Section 5.3.

Table 10: Characteristics of datasets

Dataset	$\|D\|$	$\|I\|$	T	Density (%)
Foodmart	4,141	1,559	4.4	0.28
OnlineRetail	540,455	2,603	4.37	0.17
Liquor	52,131	4026	7.87	0.19
PowerC	1,040,000	125	7.0	5.6
*s*1	300,000	9,956	5.5	0.054

5.1 Comparison of TWU-based grouping strategies

In this section, the effectiveness of the proposed *TWU-JNB* grouping is evaluated on a cluster of 12 nodes. The initial number of classes and GVF are set to 2 and 0.9 for JNB grouping algorithm, respectively. The obtained groups of items are then randomly assigned by the Spark scheduler across the nodes in the cluster, i.e., no load balancing or assignment strategies have been employed during this evaluation. To compare the performance of PUC due to JNB-based grouping of TWU, two more variants of grouping item TWUs are considered, viz., *TWU-ASC* and *TWU-DSC*. In these, the items are grouped after arranging the TWU values in ascending and descending orders, respectively. For the sake of uniform comparison, the number of groups for these variants is also set to the number of groups determined by the JNB grouping algorithm.

The experiment has been conducted to evaluate the execution time of PUC on *OnlineRetail*, *Liquor*, and *PowerC* datasets as shown in Figure 4. Two scenarios with N_g set to 3 and *sc.default.parallelism* have been considered (except *PowerC*, where N_g was set to 50 instead of *sc.default.parallelism*). PUC with *TWU-JNB* outperforms the *TWU-ASC* and *TWU-DSC*. This can be attributed to the JNB grouping that finds natural breaks in the sequence of TWU values for grouping. Such grouping brings together items that co-occur in different transactions. Consequently, with limited vertical splits, the bottleneck on the shuffling that occurs at the *reduceByKey*() operation for collecting the group-based CPBs significantly reduces. Hence, the overall execution time reduces. Further, with the increased number of groups, all the grouping strategies take significantly lesser time. This is due to the inverse relationship between the number of groups and the group size as per $group\_size = \lceil |I|/N_g \rceil$. As each group gets lesser items with the increase of N_g, the mining operation converges faster as it has to consider lesser number of suffix items for mining. Nevertheless, PUC with *TWU-JNB* remains effective due to the prudent grouping of items.

5.2 Performance evaluation of PUC with load balancing

In this section, the strategies of assigning the groups to nodes have been evaluated. PUC with *TWU_JNB* grouping incorporating load balancing as proposed in Algorithm 4 is denoted as *PUC*(*TWU_JNB* + *LB*). This is compared with two variants of PUC -*PUC*(*TWU_JNB* + *CA*) and *PUC*(*CA*). Both these variants adopt the assignment strategy as proposed in PHUI-Miner [32]. However, while the former assigns the groups, the latter directly assigns the items without grouping them. A 12 node cluster was provisioned to compare the execution times of these.

Figure 5 displays the comparison of execution time of PUC variants. Overall, PUC with proposed grouping and load balancing strategies performs better than the variants. Although at higher thresholds the difference in execution time was significantly lesser, at lower thresholds, *PUC*(*TWU_JNB* + *LB*) converged faster than its variants. Although *PUC*(*TWU_JNB* + *CA*) also groups the items based on JNB, the assignment does not take into account the mining load. The former determines the load on each cluster as a function of the subsequent utility computation algorithm, without any extra MR phase. LB calculates the loads during the first phases as the length of the CPB. Thus, the consideration of mining load provides for a balanced assignment alleviating workload skewness. Hence, mining converges faster with *PUC*(*TWU_JNB* + *LB*). For the rest of the experiments, PUC shall refer to PUC with *PUC*(*TWU_JNB* + *LB*), unless otherwise specified.

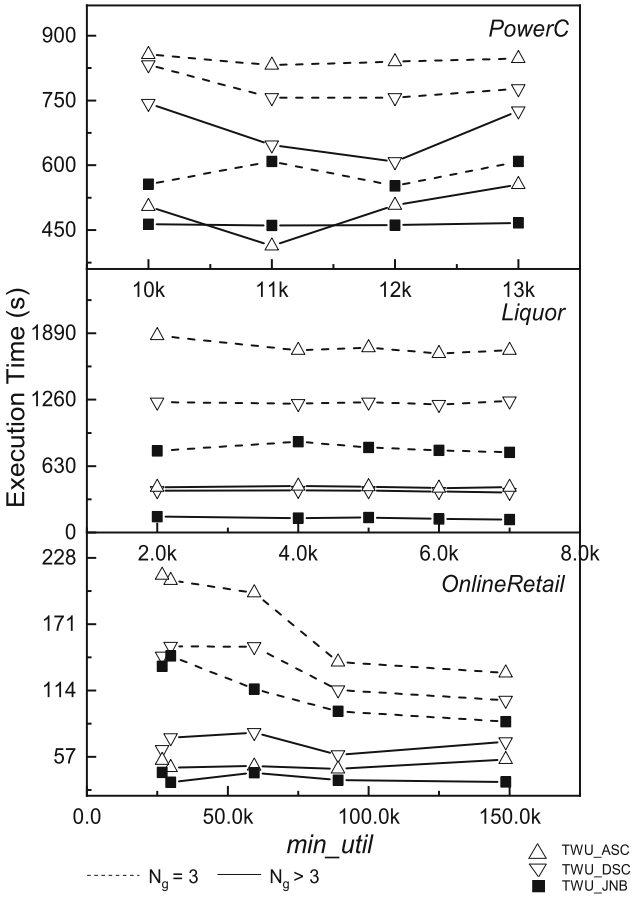

Figure 4: Comparison of TWU-grouping strategies.

Furthermore, execution time of PUC was compared against the PHUI-Growth algorithm on 8 (for *Foodmart* and *s*1) and 12 node clusters (for *Liquor*) as shown in Figure 6. Although, PHUI-Growth was proposed on Hadoop platform, the authors have customised its implementation on Spark for the sake of uniformity. Across the datasets PUC performs better in terms of execution time than the level-wise PHUI-Growth algorithm. Although *PHUI−Growth* enumerates candidates similar to Apriori approach, it employs *DLR* pruning strategy to discard unpromising items from the conditional transactions prior to enumerating itemsets of length k. However, due to the absence of grouping, the shuffling cost affects the mining performance. Also, PUC employs path utility based pruning prior to enumerating subsets of the items that are grouped in a node meticulously through TWU-JNB – facilitating enhanced performance.

Using *s*1 dataset, a scalability test was performed on a 12-node cluster. At each step, $|D|$ was increased by 50k transactions. As displayed in Figure 7, at *min_util* of 2,000, the PUC variants that employed grouping performed better. On an average, the execution time increased by 110.74 seconds when *PUC*(*CA*) was employed. However, the increase was only 85.49 and 86.99 seconds in the case of *PUC*(*TWU_JNB* + *LB*) and *PUC*(*TWU_JNB* + *CA*), respectively. Thus, grouping ensures faster completion of the second phase and early start of mining phase. In conjunction with strategic assignment by load considerations, *PUC* (*TWU_JNB* + *LB*) has a better performance over others as the database size grows.

On a 12 node cluster, the memory consumed by the algorithms for *Liquor* and *s*1 is compared and displayed in Figure 8. The memory usage of the cluster was obtained using the metrics provided by Ganglia cluster monitoring tool [42]. Across both the datasets, the PUC variants take up lesser memory during execution than PHUI-Growth. Especially at lower thresholds, while PHUI-Growth becomes more space demanding, space requirements for PUC does not show a significant rise.

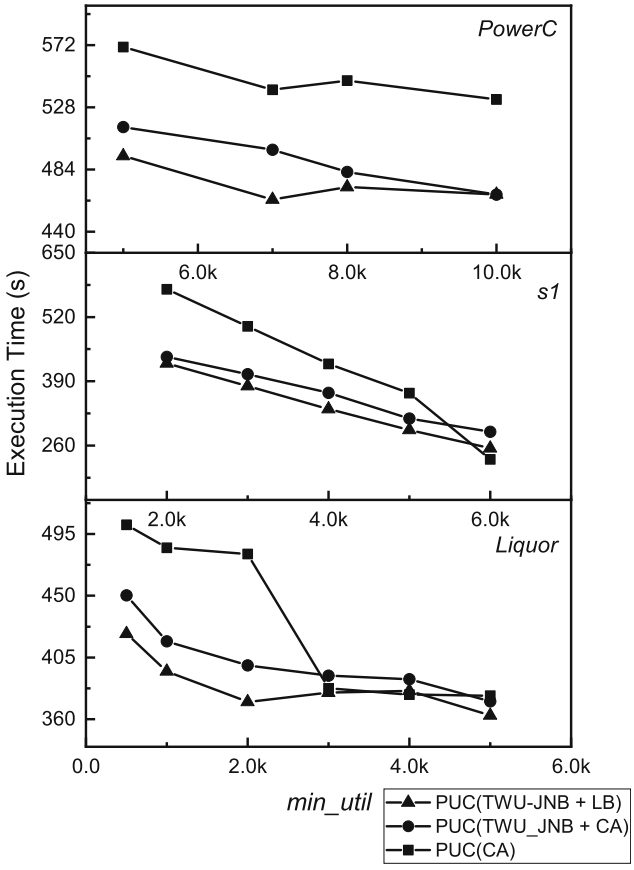

Figure 5: Comparison of different assignment strategies for PUC.

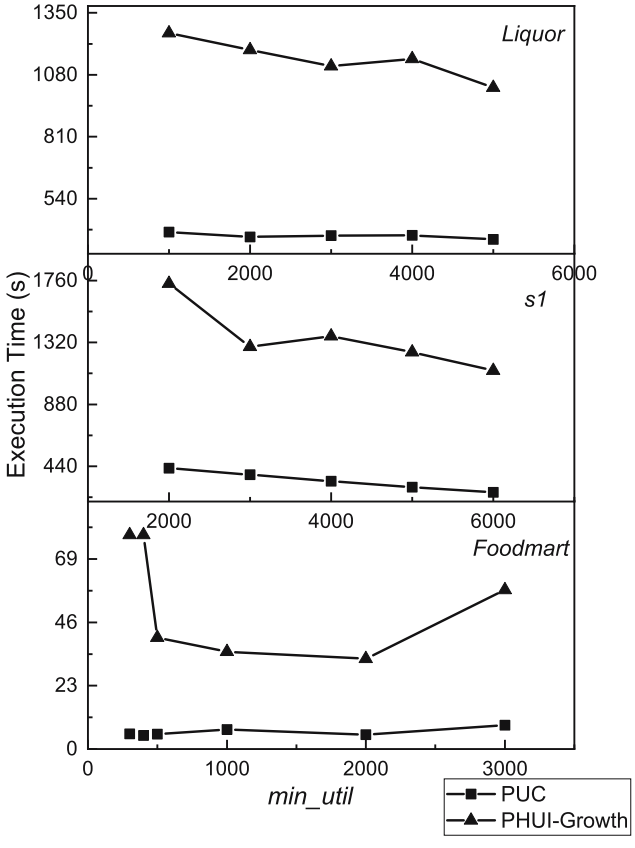

Figure 6: Comparison of PUC variants with PHUI-Growth.

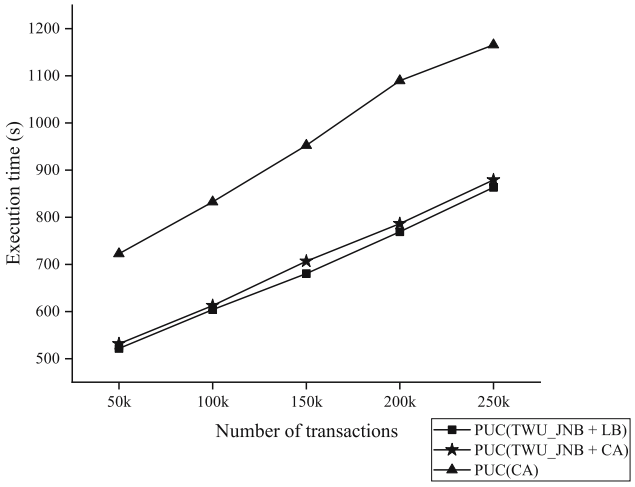

Figure 7: Comparison of PUC variants with PHUI-Growth.

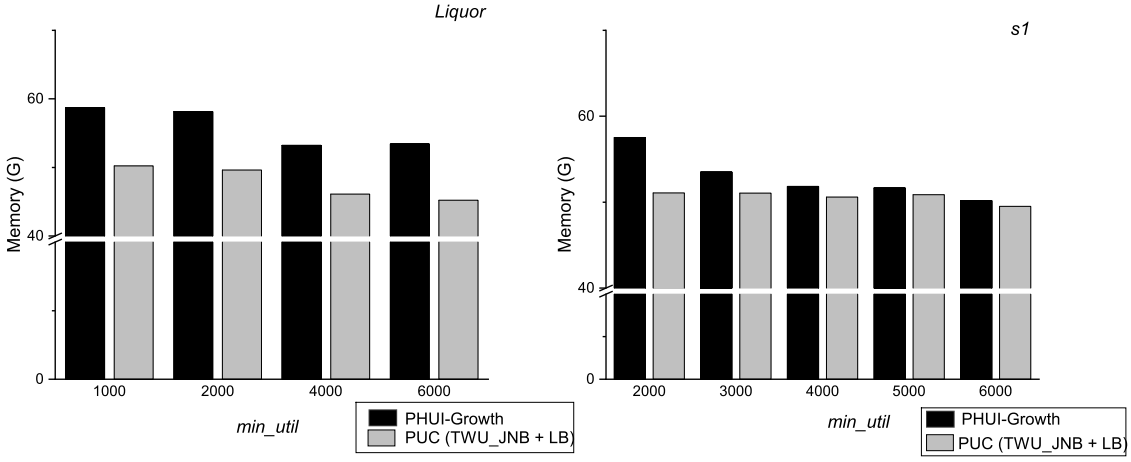

Figure 8: Comparison of cluster memory used.

5.3 Speedup

In this section, the behaviour of PUC upon scaling the cluster is assessed. For this purpose, execution times on a cluster of 4, 8, and 12 nodes were compared. Figure 9 shows that PUC executes faster with an increase in the number of nodes. As the degree of parallelism increases, the number of tasks scheduled per Spark operation increases. Furthermore, the number of groups into which the items are categorised also increases. As a result of this, each group or *map* operation has lesser number of suffix items to process. Thus, PUC performs promisingly as the cluster size increases.

For the datasets *s*1 and *PowerC*, the relative speed up is plotted in Figure 10 at *min_util* of 4 k, and 10 k, respectively. Relative speed up of PUC on a cluster of i nodes is measured with respect to execution time on a 3 node cluster as follows

$$\text{Speedup} = \frac{ET\_N_3}{ET\_N_i},$$

i.e., it compares the execution time of PUC on N_i, $i \in \{4, 8, 12\}$ number of nodes to the execution time on a 3-node cluster at a chosen threshold. Across both the datasets, a linear speed up was obtained for both *PUC*

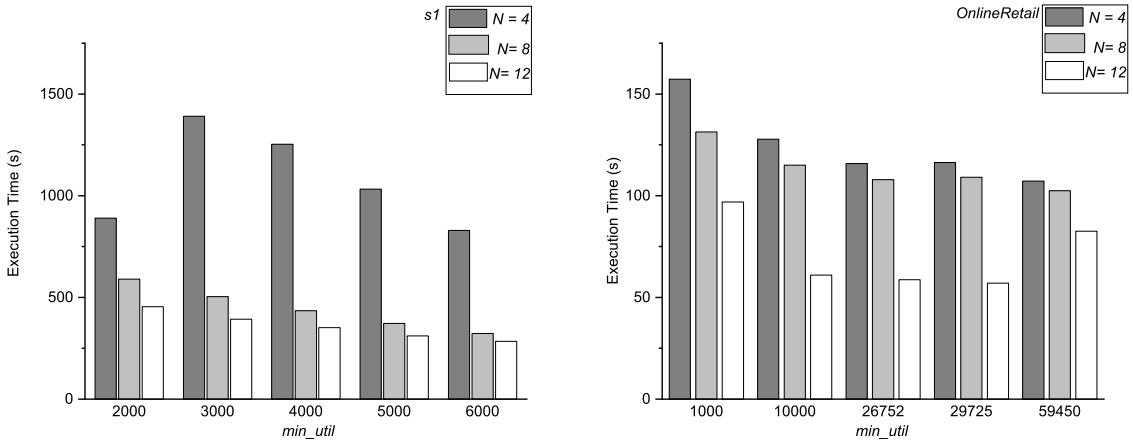

Figure 9: Comparison of execution time by varying number of cluster nodes.

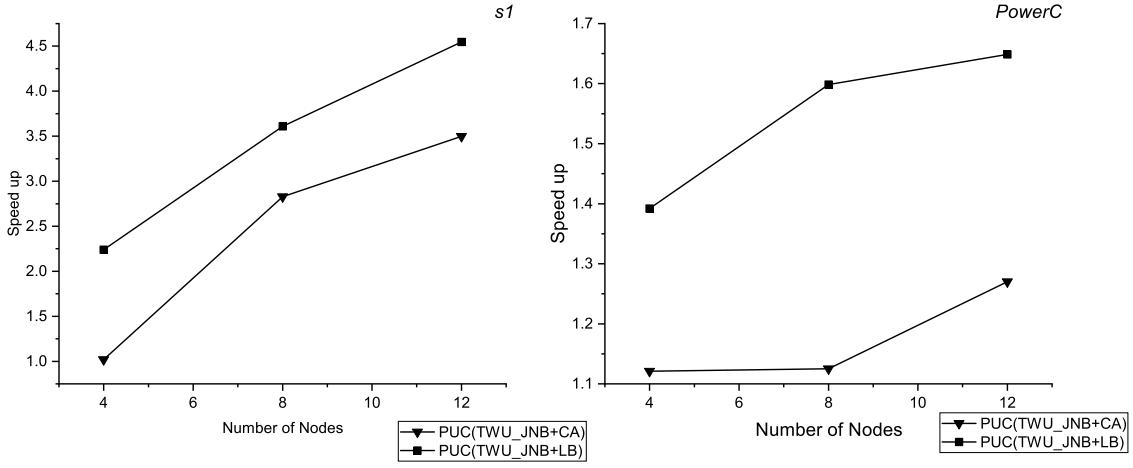

Figure 10: Comparison of speedup of PUC.

(*TWU_JNB* + *LB*) and *PUC*(*TWU_JNB* + *CA*) as shown in Figure 10. However, the former scales better due to the intelligent assignment strategy that considers the load across the cluster with a given degree of parallelism.

6 Conclusion

In this study, a Spark-based PUC (Parallel Utility Computation) algorithm for mining HUIs has been proposed. The algorithm is a parallel adaptation of the SPUC algorithm. PUC considers TWU values of items as a measure of transaction similarity. To this end, by grouping the items using Jenks Natural Breaks algorithm, mining performance improved as the number of local projections reduced, a major factor that contributes to the shuffling cost. Experimental evaluation also demonstrated that grouping items based on JNB clustering of TWU values outperformed in comparison to the same number of groups generated after sorting TWU in either ascending order or descending order. Furthermore, the groups were assigned to the nodes by approximating the mining load based on the subsets of the item to be examined from the local projected database. This load was estimated based on the prefix length of an item, which was determined without additional MapReduce phase. This load balancing-based assignment strategy decreased the

workload skewness across the cluster nodes resulting in faster convergence of the utility computation procedure. Specifically PUC with TWU based grouping of items and load balancing performed significantly better than the round robin assignment strategy and outperformed PHUI-growth algorithm. This version also displayed almost linear speed up when cluster was scaled horizontally.

In future, with the promising results obtained, the algorithm shall be extended to mine HUIs by considering on-shelf time. Apart from this, the authors would also customise the implementation on Apache Flink, another promising distributed framework for processing workloads in cluster environment.

Acknowledgments: This work was supported by Manipal Academy of Higher Education Dr. T.M.A Pai Research Scholarship under Research Registration No. 170900117.

Author Contributions: Anup Bhat Brahmavar - Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data Curation, Writing (Original Draft), Writing (Review & Editing), Visualization. Geetha Maiya - Conceptualization, Methodology, Resources, Data Curation, Writing (Review & Editing), Visualization, Supervision, Project administration, Funding acquisition. Harish Sheeranalli Venkatarama - Conceptualization, Resources, Data Curation, Writing (Review & Editing), Visualization, Supervision, Project administration, Funding acquisition.

Conflict of interest: The authors state no conflict of interest.

References

[1] Gartner SW. 3 steps to get the most from customer data. 2017. https://www.gartner.com/smarterwithgartner/3-steps-to-get-the-most-from-customer-data/. Accessed: 2021-03-26.
[2] Tran T, Vo B, Le TTN, Nguyen NT. Text clustering using frequent weighted utility itemsets. Cybern. Syst. 2017;48(3):193–209. doi: 10.1080/01969722.2016.1276774.
[3] Djenouri Y, Belhadi A, Fournier-Viger P, Lin JC. Fast and effective cluster-based information retrieval using frequent closed itemsets. Inf Sci 2018;453:154–67, doi: 10.1016/j.ins.2018.04.008.
[4] Naulaerts S, Meysman P, Bittremieux W, Vu TN, Berghe W, Goethals B, et al. A primer to frequent itemset mining for bioinformatics. Brief Bioinform. 2015;16(2):216–31. doi: 10.1093/bib/bbt074.
[5] Henriques R, Ferreira FL, Madeira SC. Bicpams: software for biological data analysis with pattern-based biclustering. BMC Bioinform. 2017;18(1):1–6. doi: 10.1186/s12859-017-1493-3.
[6] Borah A, Nath B. Identifying risk factors for adverse diseases using dynamic rare association rule mining. Expert Syst Appl. 2018;113:233–63. doi: 10.1016/j.eswa.2018.07.010.
[7] Cai R, Liu M, Hu Y, Melton B, Matheny ME, Xu H, et al. Identification of adverse drug-drug interactions through causal association rule discovery from spontaneous adverse event reports. Artif Intell Med. 2017;76:7–15. doi: 10.1016/j.artmed.2017.01.004.
[8] Liu Y, Liao W-k, Choudhary A. A two-phase algorithm for fast discovery of high utility itemsets. In: Proceeding PAKDD'05 Proceedings of the 9th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining; 2005. p. 689–95. doi: 10.1007/11430919_79.
[9] Tseng VS, Wu CW, Fournier-Viger P, Yu PS. Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans Knowledge Data Eng. 2016;28(1):54–67. doi: 10.1109/TKDE.2012.59.
[10] Liu M, Qu J. Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management; 2012. p. 55–64. doi: 10.1145/2396761.2396773.
[11] Ryang H, Yun U. Indexed list-based high utility pattern mining with utility upper-bound reduction and pattern combination techniques. Knowl Inf Syst. 2017;51(2):627–59. doi: 10.1007/s10115-016-0989-x.
[12] Zida S, Fournier-Viger P, Lin JC-W, Wu C-W, Tseng VS. Efim: a fast and memory efficient algorithm for high-utility itemset mining. Knowledge Inform Syst. 2017;51(2):595–625. doi: 10.1007/s10115-016-0986-0.
[13] Bhat BA, Harish SV, Geetha M. A single-phase algorithm for mining high utility itemsets using compressed tree structures. ETRI J. 2021;43(6):1024–37. doi: 10.4218/etrij.2020-0300.
[14] Wu JM, Srivastava G, Wei M, Yun U, Lin JC. Fuzzy high-utility pattern mining in parallel and distributed hadoop framework. Inf Sci. 2021;553:31–48. doi: 10.1016/j.ins.2020.12.004.
[15] Nguyen TDD, Nguyen LTT, Vo B. A parallel algorithm for mining high utility itemsets. In: Świątek J, Borzemski L, Wilimowska Z. editors. Information Systems Architecture and Technology: Proceedings of 39th International Conference

on Information Systems Architecture and Technology – ISAT 2018. Cham: Springer International Publishing; 2019. p. 286–95.
[16] Dean J, Ghemawat S. Mapreduce: a flexible data processing tool. Commun ACM. 2010;53(1):72–7. doi: 10.1145/1629175.1629198.
[17] Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, et al. Apache spark: A unified engine for big data processing. Commun ACM. 2016;59(11):56–65.
[18] Foundation, T. A. S. Hadoop. 2020. http://hadoop.apache.org/. Accessed: 2020-05-01.
[19] Foundation, A. S. Apache spark: Lightning-fast unified analytics engine. 2021. https://www.open-mpi.org/. Accessed: 2021-02-01.
[20] Lin YC, Wu C-W, Tseng VS. Mining high utility itemsets in big data. In: Cao T, Lim E-P, Zhou Z-H, Ho T-B, Cheung D, Motoda H, editors, Advances in Knowledge Discovery and Data Mining. Cham: Springer International Publishing; 2015. p. 649–61. doi: http://doi.acm.org/10.1145/2934664.
[21] Lin M-Y, Lee P-Y, Hsueh S-C. Apriori-based frequent itemset mining algorithms on mapreduce. In: Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, ICUIMC '12, New York, NY, USA: Association for Computing Machinery; 2012. doi: 10.1145/2184751.2184842.
[22] Li N, Zeng L, He Q, Shi Z. Parallel implementation of apriori algorithm based on mapreduce. In: 2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing; 2012. p. 236–41. doi: 10.1109/SNPD.2012.31.
[23] Qiu H, Gu R, Yuan C, Huang Y. Yafim: A parallel frequent itemset mining algorithm with spark. In: 2014 IEEE International Parallel Distributed Processing Symposium Workshops; 2014. p. 1664–71. doi: 10.1109/IPDPSW.2014.185.
[24] Li H, Wang Y, Zhang D, Zhang M, Chang EY. Pfp: Parallel fp-growth for query recommendation. In Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys '08. New York, NY, USA: Association for Computing Machinery; 2008. p. 107–44. doi: 10.1145/1454008.1454027.
[25] Shi X, Chen S, Yang H. Dfps: Distributed fp-growth algorithm based on spark. In: 2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC); 2017. p. 1725–31. doi: 10.1109/IAEAC.2017.8054308.
[26] Han J, Cheng H, Xin D, Yan X. Frequent pattern mining: current status and future directions. Data Mining and Knowledge Discovery. 2007;15(1):55–86. doi: 10.1007/s10618-006-0059-1.
[27] Kumar S, Mohbey KK. A review on big data based parallel and distributed approaches of pattern mining. J King Saud Univ Comput Inform Sci. 2019. doi: 10.1016/j.jksuci.2019.09.006. https://www.sciencedirect.com/science/article/pii/S131915781930905X.
[28] Apiletti D, Baralis E, Cerquitelli T, Garza P, Pulvirenti F, Venturini L. Frequent itemsets mining for big data: a comparative analysis. Big Data Res. 2017;9:67–83. doi: 10.1016/j.bdr.2017.06.006.
[29] Xun Y, Zhang J, Qin X, Zhao X. Fidoop-dp: Data partitioning in frequent itemset mining on hadoop clusters. IEEE Trans Parallel Distributed Syst. 2017;28(1):101–14. doi: 10.1109/TPDS.2016.2560176.
[30] Xun Y, Zhang J, Yang H, Qin X. Hbpfp-dc: A parallel frequent itemset mining using spark. Parallel Comput. 2021;101:102738. doi: 10.1016/j.parco.2020.102738.
[31] Chon K-W, Kim M-S. Bigminer: a fast and scalable distributed frequent pattern miner for big data. Cluster Comput. 2018;21(3):1507–20. doi: 10.1007/s10586-018-1812-0.
[32] Chen Y, An A. Approximate parallel high utility itemset mining. Big Data Res. 2016;6:26–42. doi: 10.1016/j.bdr.2016.07.001.
[33] Tamrakar A. High utility itemsets identification in big data. Master's thesis, University of Nevada, UNLV Theses, Dissertations, Professional Papers, and Capstones. 2017. https://digitalscholarship.unlv.edu/thesesdissertations/3044/.
[34] Sethi KK, Ramesh D, Edla DR. P-fhm.: Parallel high utility itemset mining algorithm for big data processing. Proc Comput Sci. 2018;132:918–27, International Conference on Computational Intelligence and Data Science.
[35] Sethi KK, Ramesh D, Sreenu M. Parallel high average-utility itemset mining using better search space division approach. In: Fahrnberger G, Gopinathan S, Parida L, editors. Distributed Computing and Internet Technology. Cham: Springer International Publishing; 2019. p. 108–24.
[36] Belhadi A, Djenouri Y, Lin C-W, Cano A. A general-purpose distributed pattern mining system. Appl Intell. 2020;50:2647–62. doi: 10.1007/s10489-020-01664-w.
[37] Jenks GF. The data model concept in statistical mapping. Int Yearbook Cartograph. 1967;7:186–90.
[38] Goel A, Munagala K. Complexity measures for map-reduce, and comparison to parallel computing. CoRR. 2012. abs/1211.6526.
[39] Viry M. Compute natural breaks (jenks algorythm). 2021. https://pypi.org/project/jenkspy/. Accessed: 2021-02-01.
[40] Amazon Web Services, Inc. R3: Announcing the next generation of Amazon EC2 Memory-optimized instances. 2021. https://aws.amazon.com/about-aws/whats-new/2014/04/10/r3-announcing-the-next-generation-of-amazon-ec2-memory-optimized-instances/ Accessed: 2020-12-04.
[41] Philippe Fournier-Viger SPMF An Open-Source Data Mining Library, Datasets. 2020. https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php, Accessed: 2020-12-15.
[42] Amazon Web Services (2021). Ganglia. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-ganglia.html. Accessed: 2020-03-04.