

Computing Mutual Information of Big Categorical Data and Its Application to Feature Grouping

Junli Li

School of Computer Science and Technology
Taiyuan University of Science and Technology
Taiyuan, China
1468375302@qq.com

Jifu Zhang*

School of Computer Science and Technology
Taiyuan University of Science and Technology
Taiyuan, China
jifuzh@sina.com

Chaowei Zhang

Department of Computer Science and Software Engineering
Auburn University
Auburn, USA
czz0032@tigermail.auburn.edu

Xiao Qin

Department of Computer Science and Software Engineering
Auburn University
Auburn, USA
xqin@auburn.edu

Abstract—This paper develops a parallel computing system - *MiCS* - for mutual information of big categorical data on the Spark computing platform. The *MiCS* algorithm is conducive to processing a large amount and strong repeatability of mutual-information calculation among feature pairs by applying a column-wise transformation scheme. And to improve the efficiency of the *MiCS* and the utilization rate of Spark cluster resources, we adopt a virtual partitioning scheme to achieve balanced load while mitigating the data skewness problem in the Spark Shuffle process.

Index Terms—Parallel Mutual-information Computation, Feature Grouping, Data Skewness, Big Categorical Data, Spark.

I. INTRODUCTION

Mutual-information computation requires expensive calculations, which become a serious performance bottleneck in processing big data. The overall objective of *MiCS* is to alleviate the challenging performance problem in mutual-information computation of big categorical data.

A. Observations

The inception of *MiCS* and its application is made possible by the following three observations.

- Parallel mutual-information computation of categorical data is critical in optimizing the performance of big-data applications.
- The Spark platform is an ideal mode to deal with the rapid development in the big-data arena.
- Uneven data distribution in mutual information computation can cause data skew, which leads to imbalanced work load in Spark clusters.

Our proposed *MiCS* algorithm applies mutual information as a measurement metric to quantify the similarities among categorical features in feature grouping [1]. In order to improve the performance of mutual-information computation, an growing number of parallel-computing strategies are proposed [2] [3] [4]. Importantly, these parallel algorithms fall

into the camps of GPU computing and multi-core computing. Our *MiCS* fully utilizes the Spark computing platform to compute mutual information, and a column-wise transformation method is utilized to reduce the large amount and strong repeatability of mutual-information calculation between feature pairs. For tackle data skewness in Spark, novel algorithms and models [5] [6] [7] have been proposed in recent years. Our *MiCS* algorithm explores virtual partitioning, which ensures that large partitions are eliminated.

II. PARALLEL MUTUAL-INFORMATION COMPUTING

A. Mutual Information

Let DS be a big categorical dataset, and there are n data objects and m features in DS . Mutual information $MI(y_i : y_j)$ of two features y_i and y_j is defined as

$$\begin{aligned} MI(y_i; y_j) &= H(y_i) + H(y_j) - H(y_i, y_j) \\ &= \sum_{k=1}^{d_i} \sum_{l=1}^{d_j} P_{ij}(y_i = v_{ik} \wedge y_j = v_{jl}) \\ &\quad \times \log \frac{P_{ij}(y_i = v_{ik} \wedge y_j = v_{jl})}{P_i(y_i = v_{ik}) P_j(y_j = v_{jl})}. \end{aligned} \quad (1)$$

where $P_{ij}(y_i = v_{ik} \wedge y_j = v_{jl})$ denotes the probability that features y_i and y_j equal to v_{ik} and v_{jl} (i.e., $y_i = v_{ik}$ and $y_j = v_{jl}$), respectively. d_i and d_j are the number of values that feature y_i and y_j contain respectively. $D(y_i)$ and $D(y_j)$ are the range of feature y_i and y_j , and $D(y_i) = \{v_{i1}, \dots, v_{idi}\}$, $D(y_j) = \{v_{j1}, \dots, v_{jdj}\}$. Mutual-information computation of features $MI(y_i; y_j)$ has to calculate the entropy and joint entropy of feature y_i and y_j (see Line 1 in (1)).

B. Column-wise Transformation

MiCS first splits original dataset DS into several feature subsets to compute mutual information in parallel. Next, to address the problem of repeatedly computing mutual information in the iteration, we use two variable-length arrays to

buffer the results of single feature value frequency and feature pair value frequency needed to compute mutual information.

Let n and m be the size of categorical dataset DS and the number of features in data objects. $y_1 \sim y_m$ are features in DS , which is split into different feature subsets according to computing tasks. Each feature subset forms an RDD object that is independent of other feature subset. The process of the column-wise transformation is illustrated in Fig. 1.

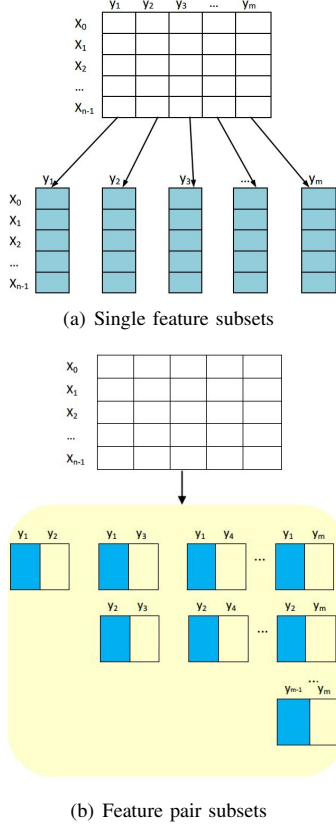


Fig. 1: The process of the column-wise transformation.

In MiCS, parallel mutual-information computing mainly performs the two important operations.

Map Operations convert RDD to the $\langle K, V \rangle$ format (e.g., $\langle (x(k), 1) \rangle$), in which $x(k)$ is a single feature value of feature k ; 1 signifies that the feature value appears once (see Line 8 in Algorithm 1). To calculate mutual information of feature y_m and y_n (i.e., feature pair (m, n)), we have to track the overall occurrences of feature m (i.e., $x(m)$) and feature n (i.e., $x(n)$).

CountByKey or reduceByKey Operations calculate the occurrences of feature-pair values and single-feature values (e.g., $x(k)$). Given a single feature k , each node in a Spark cluster independently counts $x(k)$'s local occurrences; thus, multiple nodes are measuring $x(k)$'s occurrences in parallel. After we statistically computed the array of local $x(k)$ occurrences, local occurrences of $x(k)$ are aggregated into a global count of $x(k)$.

TABLE I: Example of the *singleCol* of MiCS

Indexes	1	2	3	4	5	...
value	(a,10)	(c,35)	(f,23)	(d,65)	(e,76)	...

TABLE II: Example of the *doubleCol* of MiCS

Indexes	1	2	3	4	5	...
1		((a,c),8)	((a,b),13)	((c,f),65)	((d,e),76)	...
2			((e,f),12)	((m,d),34)	((n,e),24)	...
3				((a,b),87)	((z,n),19)	...
4					((b,c),92)	...
5						...
...						...

To mitigate the problem that computing load linearly increases with the number of iterations in the feature-grouping process, the frequencies of *single-feature* value and *feature-pair* value are calculated and buffered to two variable-length arrays named *singleCol* and *doubleCol*, respectively, which are broadcast to all slave nodes. Subsequently, the tasks load the corresponding data from the *singleCol* and *doubleCol* arrays. Hence, single-feature and feature-pair results are effectively reused, thereby cutting back computing load.

Table I and Table II shows an example of variable-length arrays *singleCol* and *doubleCol* in MiCS.

C. Data Skewness

1) *An Overview*: In the shuffle phase, if the amount of data corresponding to a key is particularly large, skewed data will emerge.

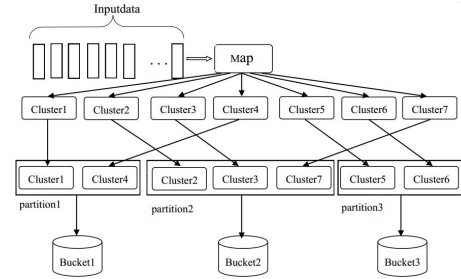


Fig. 2: Data distribution in the Spark Shuffle.

Fig. 2 depicts that *Partition 2* is a whole lot larger than *Partition 1* and *3*. Since the input data is unevenly distributed, the use of the system's default Hash partition may lead to large discrepancies in the sizes of each partition in child RDDs, resulting in skewed data.

2) *The Data-skewness Model*: We build a data-skewness model to quantify the issue induced by skewed data among partitions created by Spark. Fig. 2 depicts the default hashing distribution mechanism in Spark clusters, which conduct the following three steps. First, Map tasks retrieve input data. Then, these data are processed by the Map tasks, which produce intermediate results organized in the format of key-value pairs. Last, the intermediate results are grouped into

partitions using the keys. A road block in the last step is that these partitions have uneven sizes due to skewed data.

Assuming there are p unique keys when data are aggregated according to their keys, we let K represent the keys, $K = \{k_1, \dots, k_p\}$. And we denote V as the values of all the keys in set K . Thus, we present value set V as

$$V = \{ \underbrace{v_{11}, \dots, v_{1l_1}}_{\text{dom}(k_1)}, \dots, \underbrace{v_{i1}, \dots, v_{il_i}}_{\text{dom}(k_i)}, \dots, \underbrace{v_{p1}, \dots, v_{pl_p}}_{\text{dom}(k_p)} \}, \quad (2)$$

where there are the number p of partitions; values in each partition share a single key (e.g., a group of values $v_{i1}, \dots, v_{il_i}$ share key k_i). It is noteworthy that all the partitions may have distinct size. For example, the sizes of the i th and j th partitions are l_i and l_j , respectively. These two partitions may differ from one another in size (i.e., $l_i \neq l_j$).

$\text{dom}(k_i)$ in (2) denotes the domain of key k_i ; thus, we have $\text{dom}(k_i) = \{v_{i1}, \dots, v_{il_i}\}$, $i \in 1, \dots, p$. Now we measure the size of the i th partition for key k_i using domain $\text{dom}(k_i)$'s size, which is formally expressed as $|\text{dom}(k_i)|$. The average partition size is represented by $|\text{dom}(K)|_{\text{avg}}$, which is gauged by the average domain size as follows:

$$|\text{dom}(K)|_{\text{avg}} = \frac{\sum_{i=1}^p |\text{dom}(k_i)|}{p} = \frac{\sum_{i=1}^p \sum_{j=1}^{l_i} v_{ij}}{p} \quad (3)$$

The degree of data skewness among the number p of partitions is defined as the deviation of the partition sizes (a.k.a., domain sizes or $|\text{dom}(k_i)|$). Let $s(k_i)$ be the skewness degree of the i th partition or domain $\text{dom}(k_i)$. Formally, skewness degree $s(k_i)$ with respect to domain $\text{dom}(k_i)$ is given as

$$s(k_i) = \frac{|\text{dom}(k_i)|}{|\text{dom}(K)|_{\text{avg}}} = \frac{p \times \sum_{j=1}^{l_i} v_{ij}}{\sum_{i=1}^p (\sum_{j=1}^{l_i} v_{ij})}, \quad (4)$$

where the denominator on the right-hand size of (4) is the average domain size $|\text{dom}(K)|_{\text{avg}}$ derived from (3).

D. Handling Skewed Data in MiCS

To mitigate data skewness in MiCS, we conduct virtual partitioning while calculating single-feature values. Fig. 3 depicts the process of virtual partitioning.

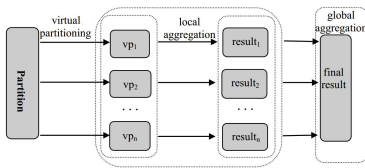


Fig. 3: The process of virtual partitioning.

Here we add a random prefix to each key in RDD, followed by the *reduceByKey* aggregation. By adding a random prefix to the same key and changing it into several different keys, the data originally processed by one task are dispersed to multiple tasks for local aggregation.

E. Algorithm Design

Algorithm 1 combines two functions, namely, column-wise transformation and virtual partitioning. The pseudocode of this algorithm is depicted as follow.

Algorithm 1 Column-wise Transformation and Virtual Partitioning

```

1: Input: Data set DS ( $n$  objects  $\times$   $m$  features), RDD datapre built from DS
2: Output: two arrays of single feature value and feature pair value
3: val doubleCol
4:   =new Array[ArrayBuffer[Map[(String,String),Long]]](dimension)
5: for ( $m \leftarrow 0$  until dimension)
6:   for ( $n \leftarrow 0$  until dimension)
7:     doubleCol( $m$ )( $n$ ) += datapre.map( $x \Rightarrow ((x(m), x(n)), 1)$ )
8:     .countByKey().toMap
9:   end for
10: end for
11: val singleCol = ArrayBuffer[Map[String,Long]]()
12: for ( $k \leftarrow 0$  until dimension)
13:   singleCol += datapre.map(
14:     val random:Random = new Random()
15:     val prefix:Int = random.nextInt(10)
16:     prefix + "-" + x(k), 1))
17:   .reduceByKey(_+_ )
18:   .map( $\text{line} \Rightarrow (\text{line}._1.\text{split}("-") (1), \text{line}._2)$ )
19:   .reduceByKey(_+_ )
20:   .collectAsMap().toMap
21: end for

```

Step 1. Keyword *val* defines two variable-length arrays, (i.e., *singleCol* and *doubleCol*). We adopt the *map* transformation to convert the *dataPre* into the format of pair $((x(m), x(n)), 1)$. One thing to be aware of is that $(x(m), x(n))$ is the value of feature-pair (m, n) ; 1 signifies that the value of feature-pair appears once.

Step 2. To calculate mutual information of feature y_m and y_n (i.e., feature pair (m, n)), we have to track the overall occurrences of feature m (i.e., $x(m)$), feature n (i.e., $x(n)$) and feature-value pair (e.g., $(x(m), x(n))$).

Step 3. The array *singleCol* (see Line 12) is similar to *doubleCol*, but *singleCol* tracks the single-feature value which occurrences on one dimension.

Step 4. Since data skewness is likely to occur in the calculation of single-feature value, virtual partitioning (see Line 15 - 20) is performed to alleviate the data skewness problem.

III. EXPERIMENTAL RESULTS

A. Experimental Setup

The MiCS system is implemented and evaluated on a 24-node Spark Cluster, each computing node of which has an Intel processor (i.e., E5-1620 v2 series 3.7 GHz) with four cores and 16GB RAM. *Master's* hard drive is 500 GB; each *Worker* has a 2 TB disk capacity. We evaluate the performance of MiCS using both real-world and synthetic categorical datasets.

Real-world datasets. UCI datasets, including Connect-4, Mushroom, Chess, Breast-cancer, and Splice.

Synthetic datasets. Synthetic categorical datasets are generated in two steps. First, a relatively small dataset is originated using the random data generator. Next, to enlarge the dataset

size, we copy the dataset generated in the previous step; the data sizes are configured to be 8GB, 16GB, 24GB and 32GB, respectively. There are 100 features in these datasets.

B. Impacts of Column-wise Transformation on MiCS

Fig. 4 reveals the runtime results of MiCS with and without column-wise transformation running on a single node.

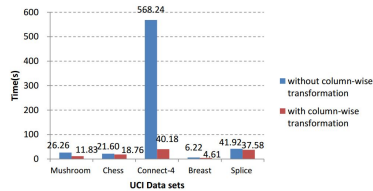


Fig. 4: Column-wise transformation influences.

Fig 4 unveils that regardless of datasets, the column-wise transformation significantly optimize the performance of MiCS in all the five cases. And we observe from Fig 4 that the performance improvement made by the column-wise transformation becomes more pronounced when the data size is expanded.

C. Impacts of Virtual partitioning

1) *Impacts of Data-skewness Degree*: DEFH [8] is the most widely-used hashing algorithm, which is a default mechanism in Spark. We vary the data-skewness degree of normal datasets. The impacts of data skewness are evaluated from the perspective of processing time.

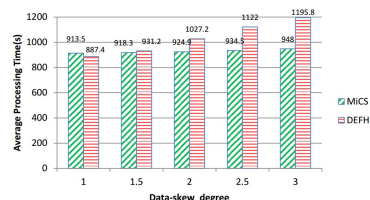


Fig. 5: The impacts of data-skewness degree.

Fig. 5 shows the processing times of MiCS and DEFH as functions of the data-skewness degree, which varies from 1 to 3 with an increment of 0.5. We observe that the processing time of MiCS is less sensitive to data-skewness degree than DEFH. The results confirm that MiCS makes use of virtual partitioning to effectively mitigates the performance problem induced by skewed data. The performance improvement of MiCS over DEFH becomes more pronounced under a higher data-skewness degree.

2) *Shuffling-Cost Analysis*: Let us vary the number of computing nodes to compare the shuffling cost of MiCS and DEFH. The *shuffle-write size* of slave nodes is monitored as the shuffling cost of the algorithms. Taking the 8G data as an example, Fig. 6 plots the shuffling cost comparison.

As an apparent observation, the *shuffle-write sizes* of MiCS in all the test cases are significantly smaller than those of

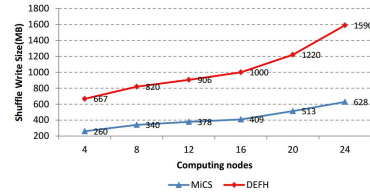


Fig. 6: The shuffling cost comparative analysis.

DEFH. More importantly, the shuffle-write-size gap between the two solutions is widened with an increasing number of computing nodes. DEFH is reliance on the Spark's default hash partition leading to tasks frequently accessing data across multiple nodes. Different from DEFH, our MiCS leverages the virtual partitioning techniques to slash the amount of data transmission in the Spark environment.

IV. CONCLUSIONS

In this study, we investigated the column-wise transformation and virtual partitioning scheme in MiCS. MiCS is implemented on a 24-node Spark driven by both real and synthetic datasets with categorical features. It is evident that MiCS is conducive to speeding up the mutual-information calculation by the virtue of column-wise transformation. Further, MiCS is adroit at suppressing the amount of skewed data to optimize the network performance in Spark processing big categorical datasets.

ACKNOWLEDGMENT

This work is partially supported by the National Natural Science Foundation of P. R. China 61572343.

* corresponding author

REFERENCES

- [1] J. Li, J. Zhang, N. Pang, and X. Qin, "Weighted outlier detection of high-dimensional categorical data using feature grouping," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. PP, no. 99, pp. 1–14, 2018.
- [2] S. Ramtin, S. Parastoo, K. Rodney, and H. Richard, "Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images," *Computer Methods and Programs in Biomedicine*, vol. 99, no. 2, pp. 133–146, 2010.
- [3] S. Saxena, S. Sharma, and N. Sharma, "Parallel computation of mutual information in multicore environment and its applications in medical image registration," in *International Conference on Medical Imaging*, 2014.
- [4] A. Adinetz, J. Kraus, M. Axer, M. Huysegoms, S. Khnen, and D. Pleiter, "Computation of mutual information metric for image registration on multiple gpus," in *IEEE Aerospace Conference*, 2013.
- [5] Z. Tang, W. Lv, K. Li, and K. Li, "An intermediate data partition algorithm for skew mitigation in spark computing environment," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2018.
- [6] G. Liu, X. Zhu, W. Ji, D. Guo, W. Bao, and G. Hui, "Sp-partitioner: A novel partition method to handle intermediate data skew in spark streaming," *Future Generation Computer Systems*, vol. 86, pp. 1054 – 1063, 2018.
- [7] M. Phinney, S. Lander, M. Spencer, and C. R. Shyu, "Cartesian operations on distributed datasets using virtual partitioning," in *IEEE Second International Conference on Big Data Computing Service and Applications*, 2016, pp. 1–9.
- [8] P. W. Holden Karau, Andy Konwinski and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Incorporated, 2015.