DOI 10.18699/sblai2023-42

# *MapReduce vs Non-MapReduce - Efficiency and Scalability in Big Data Computing*

# *MapReduce и не-MapReduce – эффективность и масштабируемость в области вычислений больших данных*

***Xudong Sun***,
Shenzhen University
ShenZhen, China,
孙旭东
深圳大学
中国，深圳
*sunxudong2016@email.szu.edu.cn*

***Dingming Wu***,
Shenzhen University
ShenZhen, China,
吴定明
深圳大学
中国，深圳
*dingming@szu.edu.cn*

***Yongda Cai***,
Shenzhen University
ShenZhen, China,
蔡湧达
深圳大学
中国，深圳
*caiyongda2021@email.szu.edu.cn*

***Lingxiang Zhao***,
Shenzhen University ShenZhen, China,
赵凌翔
深圳大学
中国，深圳
*2200271016@email.szu.edu.cn*

***Changda Xiao***
Shenzhen University
ShenZhen, China
萧畅达
深圳大学
中国，深圳
*2200271024@email.szu.edu.cn*

***Joshua Zhexue Huang***,
Shenzhen University
ShenZhen, China
黄哲学*
深圳大学
中国，深圳
*zx.huang@szu.edu.cn*

***Abstract***. MapReduce is a popular distributed computing paradigm for processing big data in a massively parallel fashion. However, when it is used to implement and run highly iterative algorithms for analyzing distributedly stored big data, the MapReduce paradigm loses its computing efficiency and data scalability due to the communication costs occurring in iterations of the algorithm over the entire dataset. Non-MapReduce is an alternative computing paradigm that removes the communication costs when executing iterative algorithms on big data that is stored using the random sample partition data model. In the Non-MapReduce paradigm, a set of random sample data blocks are selected and loaded into the memory of computing nodes. An iterative algorithm is dispatched to each computing node and executed on local data set independently and in parallel without communications among the nodes. Afterwards, the local results are transferred to the master node for computing the final result. In this paper, we propose the LOGO computing framework, a core technology for Non-MapReduce paradigm, and demonstrate its computing performance with widely used supervised learning, unsupervised learning, and pattern mining algorithms. The experiment results show that LOGO outperforms the state-of-the-art Spark by orders of magnitude in terms of the running time. LOGO is scalable to terabyte-scale data sets with high-quality results.

***Аннотация.*** MapReduce — это популярная парадигма распределенных вычислений для обработки больших данных в массово-параллельном режиме. Однако когда он используется для реализации и запуска высокоитеративных алгоритмов анализа распределенно хранящихся больших данных, парадигма MapReduce теряет свою вычислительную эффективность и масштабируемость данных из-за затрат на связь, возникающих при итерациях алгоритма по всему набору данных. Non-MapReduce — это альтернативная парадигма вычислений, которая устраняет затраты на связь при выполнении итерационных

алгоритмов на больших данных, хранящихся с использованием модели данных раздела случайной выборки. В парадигме Non-MapReduce набор блоков данных случайной выборки выбирается и загружается в память вычислительных узлов. Итерационный алгоритм отправляется на каждый вычислительный узел и выполняется на локальном наборе данных независимо и параллельно без связи между узлами. После этого локальные результаты передаются на главный узел для вычисления окончательного результата. В этой статье мы предлагаем вычислительную среду LOGO, базовую технологию для парадигмы Non-MapReduce, и демонстрируем ее вычислительную производительность с помощью широко используемых алгоритмов контролируемого обучения, неконтролируемого обучения и анализа шаблонов. Результаты эксперимента показывают, что LOGO превосходит современный Spark по времени работы на несколько порядков. LOGO масштабируется до наборов данных размером в терабайты с получением высококачественных результатов.

摘要—MapReduce 计算范式是当今流行的一种以大规 模并行方式处理大数据的分布式计算范式。然而，当它被用 于实现和运行高度迭代的算法来分析分布式存储的大数据 时，由于算法在整个数据集上迭代计算产生的通信开销，使 MapReduce 计算范式降低了计算效率并失去了数据可扩展 性。Non-MapReduce 计算范式是一种替代的计算范式，它采 用了随机样本划分数据模型存储大数据，在执行迭代算法时消除 了数据通信开销。在 Non-MapReduce 计算范式中，首先选 择一组随机样本数据块被选择并加载到计算节点的内存中；然后 将迭代算法分派到每个计算节点，并在节点上独立并行地处理 本地数据集，无需节点间数据通信；最后，节点计算的本地结果 被传输到主节点，计算最终结果。本文提出 Non-MapReduce 计算范式的核心技术 LOGO 计算框架，并通过常见的有监督 学习、无监督学习和模式挖掘算法展示其计算性能。实验结果表 明，LOGO 计算框架在运行时间上要优于目前最先进的 Spark 计算框架几个数量级。除此之外，LOGO 计算框架的数据可扩 展性达到 TB 级别，并可以获得令人满意的结果。

## *1. Introduction*

Big data analytics enables data-guided decision-making which has found successful applications in smart cities [1], fraud detection [2–4], disease diagnosis and prediction [5, 6], industrial production [7, 8], weather forecasting [9], etc. Nowadays, we are seeing a massive growth in demand for processing and anaylzing big data. According to the Statista report (*https://www.statista.com*), the overall amount of data generated in the world is around 97 zettabytes in 2022, and it is expected to grow to 181 zettabytes by 2025. It is shown by Precedence Research (*https://www.precedenceresearch.com/data-analytics-market*) that the global data analytics market size was exhibited at USD 41.39 billion in 2022 and is projected to surpass USD 346.33 billion by 2030.

Currently, distributed computing platforms and systems [10] are widely adopted to handle big data, since it is prohibitively expensive or physically impossible to process and analyze big data using a single machine. The popular general purpose distributed platforms such as Hadoop (*https://hadoop.apache.org*) and Spark (*https://hadoop.apache.org*) adopt the MapReduce architecture, shown in Fig.1. The

big data files are divided into smaller blocks and stored on distributed file system. Analysis tasks are performed by conducting iterations of mapper and reducer operations on these blocks. An iteration consists of a Mapper and a Reducer phases. The shuffling operation in between the Mapper and the Reducer phases incurs an N-to-N data transferring. The communication overhead occurs in each iteration and is proportional to the number of iterations in the algorithm [11]

Native distributed machine learning systems such as TensorFlow [12], MXNeT [13], DIANNE [14], and PyTorch [15] follow the parameter server architecture, shown in Figure 2. Each worker node computes local parameters using local data. The CPU is responsible for initializing, storing, and updating the parameters, while the GPU transforms input data into tensors and trains the model. The global model parameters are then aggregated and updated on the server nodes. This process is iterative, where in each iteration, the worker nodes need to communicate with the server node.

Commonly used analytical algorithms such as classification [16], clustering [17], and pattern mining [18] are iterative, which need to scan and process the entire dataset many times. Hence, the MapReduce and the parameter server architectures have

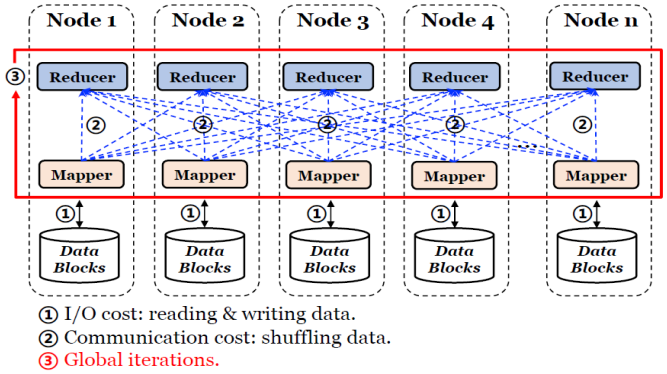



*Fig.1: The architecture of MapReduce.*

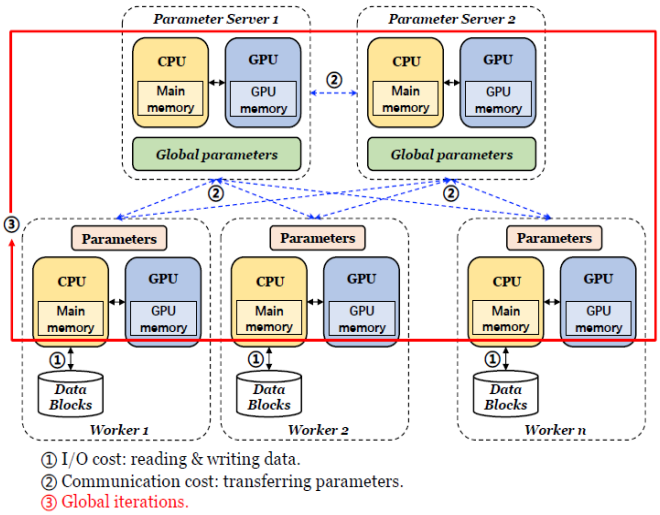



*Fig. 2: The architecture of parameter server.*

the following limitations when executing iterative algorithms to process big data.

• High communication cost. In the MapReduce and the parameter server, the communication cost occurs between two consecutive iterations when executing iterative algorithms. All the machines need to send and receive data and parameters before entering the next iteration. Thus, the communication cost grows substantially as the number of iterations and the volume of data increase.

• High RAM requirement. When the data volume exceeds the main memory (RAM) capacity in the system, it will cause frequent memory-swapping that moving data between main memory and secondary storage, which is time-consuming. Spark reduces the data input and output cost by using a distributed in-memory data abstraction called Resilient Distributed Datasets (RDDs). However, to handle big data, it requires a large amount of RAM. This is an impractical solution for rapid growth of data volume.

• Inconvenient parallel programming model. It is tedious and complex to convert analytical algorithms into the corresponding parallel versions in the current distributed systems. Programmers have to rewrite analytical algorithms using the available programming interfaces. Moreover, debugging and optimizing parallel programs are often difficult and timeconsuming.

The above limitations can be eliminated by partitioning a big dataset as data blocks which are made as random samples of the big dataset, e.g., the random sample partition (RSP) data model [19]. This data representation allows each data block being processed independently in a computing node by an iterative algorithm, and many data blocks being processed by the same algorithm in a massive parallel fashion. The data block results are then transferred to the master node for computing the final result. This Non-MapReduce approach, which we will discuss in this article, has the following merits in big data analytics.

(i) Scalability. When handling a growing amount of data, e.g., terabyte-scale data, iterative algorithms are able to respond in a reasonable time.

(ii) I/O efficiency. When the data partitions and the intermediate results cannot fit in memory, the memory-swapping is not frequent.

(iii) Communication reduction. When performing iterations, the communication costs among the computing nodes keep low.

(iv) Simple parallel programming model for iterative algorithms. Complex iterative algorithms can be easily packaged as operators to execute in a parallel fashion.

The remainder of this article is organized as follows. First, we review the core technologies of Non-MapReduce Computing - RSP data model and LOGO framework, and its implementation in Spark. Then, we use three types of machine learning algorithms for supervised learning, unsupervised learning, and pattern mining to illustrate the major steps of Non-MapReduce Computing in executing these

iterative algorithms over large data sets. Finally, we provide simulated results to show the performance of efficiency and data scalability of Non-MapReduce computing in running machine learning algorithms over big data sets up to 10 TB. We conclude this article by a summary of the advantages of Non-MapReduce computing in big data analytics.

***2. Non-mapreduce computing***

Non-MapReduce computing is powered by two core technologies – The RSP data model [19] which enables the data blocks of a distributed data file being used as random samples of the big data file, and the LOGO framework which we propose in this article to enable Non-MapReduce computing. The two technologies are detailed in the following two sections which are followed by an implementation in Spark.

***2.1 RSP Data Model***

A distributed data file is saved as a set of data blocks which are distributed on the nodes of a cluster and managed by a distributed file system, e.g., HDFS [20]. The random sample partition or RSP data model represents a big data set as a partition of RSP data blocks [19] which are also saved in a distributed fashion and managed by HDFS. However, each RSP data block is made by an random sample partitioning algorithm [21] a random sample of the big data set so it can be used to compute the statistical estimates of the big data set.

Formally, let $D=\{r_1, r_2, ..., r_N\}$ be a multivariate data set of $N$ records. Each data record $r_i=\{x_1, x_2, ..., x_M\}$ has $M$ features, where each feature $x_i$ is a random variable. Let $T$ be a random sample partitioning algorithm that divides $D$ into $K$ nonoverlapping subsets $D_1, D_2, ..., D_K$ which satisfy the following three conditions:

$(i)\, D=U_{i=1}^{K} D_i, (ii)\, D_i \cap D_j=\emptyset, 1\leqslant i, j\leqslant K, i\neq j,$

$(iii)|F(D_i)-F(D)|<\delta, 1\leqslant i\leqslant K$, where $F()$ is the cumulative distribution function (CDF) of a data set, and $\delta$ is a small positive constant. This condition states that the CDFs of all RSP data blocks must be similar to the CDF of the data set.

The set of $D_1, D_2, ..., D_K$ is called an RSP data model of D. The algorithm T in [21] works as follows: Initially, data blocks are stored across $n$ machines (a.k.a. nodes). Suppose that each node stores $K$ data blocks. On each node, all the records in each data block are randomized, and then each data block is sliced into $K$ mini-blocks. This procedure is executed in parallel on all nodes. After that, each node receives one mini-block from each of all the other node, and then merges them with its local one mini-block. In the end, the original data set is reorganized as RSP blocks and saved in HDFS. Figure 3 shows the program workflow

***2.2. LOGO Framework***

Given a data set D as an RSP data model, in Non-MapReduce computing, an

analytical task of D is carried out under the Local Operations with Global Operations (LOGO) computing framework on commodity clusters. The architecture of LOGO consists of two layers: the data layer and the computation layer, as shown in Fig. 4.

At the data layer, data files take the form of RSP blocks that are distributedly stored across $n$ nodes. Usually, an RSP block corresponds to a random sample of the data file to be analyzed. At the computation layer, the execution process of an analytical task consists of a local phase and a global phase. One node is responsible for the global phase, called GO master. All the nodes can join the local phase, called LO workers.

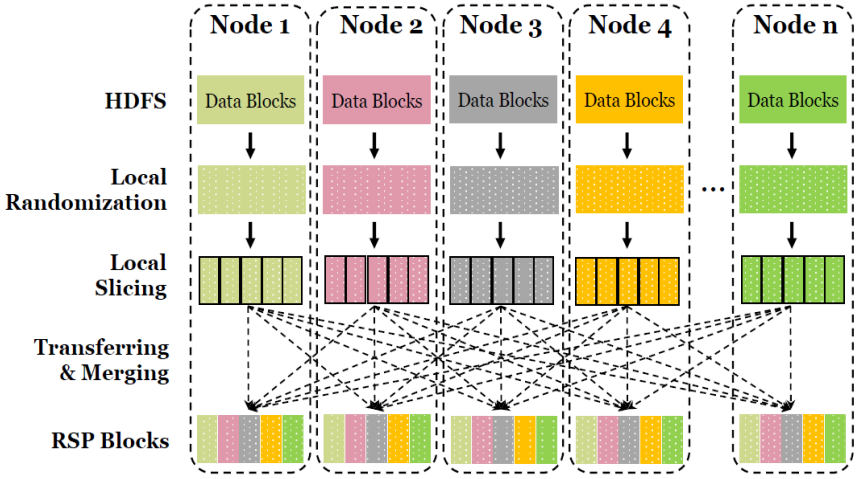



*Fig. 3: Workflow of algorithm T.*

Local Phase. Each LO worker performs a local operation on its own RSP data block without communicating with other workers. The local operation can be a simple algorithm or a highly iterative one. At the end of the local phase, a local model based on one RSP data block is produced. Global Phase.

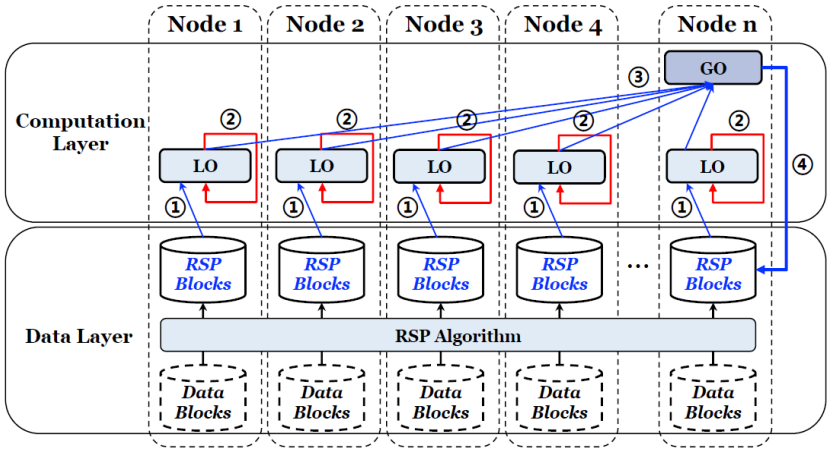



*Fig. 4.*
*①I/O cost: reading RSP blocks.②Local iterations. ③Communication cost: transferring*

*local results. ④I/O cost: writing final results.*

When all LO workers complete their local tasks, the GO master is notified and uses remote procedure calls to read the local models from all LO workers. It performs a global operation that merges all the local models and saves the final model in the storage. Different from MapReduce which executes an iterative algorithm through iterating many pairs of map and reduce operations resulting from heave data communication costs, under the LOGO framework, the iterative algorithm completes its executions in the LO workers in parallel without the need of data communications. Therefore, it is very efficient in executing iterative algorithms.

### *2.3 Implementation in Spark*

Non-MapReduce computing with the RSP data model and the LOGO computing framework is implemented on HDFS (*https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html*) and Spark. The following three main components are detailed below. (i) RspRDD Abstraction. Spark uses RDDs to share data among computations. We follow this idea and create a data-sharing abstraction called RspRDD by extending the RDD in Spark. The RSP blocks in HDFS are represented in Spark by an RspRDD object that can be manipulated in parallel, e.g., class RspRDD[T: ClassTag](prev: RDD[T]) extends RDD[T](prev){...} (ii) RSP Conversion.We provide function RandomizeAndSlice() for randomizing the records in a data block and slicing the data block into $K$ mini-blocks. Users can use it in the following code snippet below (Spark API in Scala) to covert the data blocks in HDFS into RSP data blocks. Line 1 specifies the data file to be converted and the number $K$ of partitions in an RDD. Line 2 passes function RandomizeAndSlice() to Spark API function mapPartitions(), so that local randomization and slicing are executed on each data block across all nodes. Then, each mini-block is assigned an integer key in the range of [1,$K$]. Line 3 calls Spark API function reduceByKey(), so that mini-blocks are transferred and merged, resulting in RSP blocks on each node. Finally, line 4 saves these RSP blocks in an HDFS file.

```
1: val RDD1 = sparkContext.textFile(filepath, K)
2: val RDD2 = RDD1.mapPartitions(RandomizeAndSlice())
3: val rspRDD = RDD2.reduceByKey()
4: rspRDD.saveAsTextFile(rspfilepath)
```

(iii) LOGO APIs. The local operation and the global operation in the LOGO framework are expressed by functions LO() and GO(), respectively. The input RSP blocks as partitions of an RDD are first processed by function LO(), which outputs a local model on each RSP partition. Function GO() merges all local models produced by function LO() and outputs the final model. LOGO APIs support multiple programming languages such as Java, Scala, and Python. The details of the LOGO API functions are described below.

• rspTextFile(rspfilepath): It loads an RSP data file in HDFS and returns an RspRDD object that represents the RSP data blocks as RSP partitions in RDD.

• getSubPartitions(m): It randomly selects *m*% of the RSP partitions.

• LO(L_Algorithm()): It accepts a function L_Algorithm(), which is user-defined and can be any analytical algorithm. It dispatches function L_Algorithm() to all LO workers. Each LO worker applies function L_Algorithm() to its own RSP partition, and then produces a local model from each RSP partition. Similarly, LO(preProcess()) is also valid, which specifies the function for preparing data on each RSP partition. For instance, running LO(preProcess()) before LO(L_Algorithm()) indicates the data preprocessing before modeling.

• GO(G_Algorithm()): It accepts a user-defined function G_Algorithm() that merges all local models and produces the final model. Client Program. The code snippet in Scala below shows a user application example using LOGO. Line 1 prepares an RspRDD object for the specified RSP data file. Line 2 takes □□RSP partitions. Line 3 preprocesses each data partition. Line 4 executes function LO() on each RSP partition and returns *m* local models. Line 5 executes function GO() that produces the final result by merging the *m* local models. Line 6 saves the final model in HDFS at filepath.

```
1: val rspRDD = sparkContext.rspTextFile(rspfilepath)
2: val dataRDD = rspRDD.getSubPartitions(m)
3: val trainRDD = dataRDD.LO(preProcess())
4: val modelRDD = trainRDD.LO(L_Algorithm())
5: val resultRDD = modelRDD.GO(G_Algorithm())
6: resultRDD.save(filepath)
```

We can see from this example that using LOGO, it becomes easier to write a user application for an analytical task.

### *3. Machine learning algorithms under LOGO*

Machine learning algorithms are widely used in big data analytics, and most of them are iterative over data sets to search for an optimal solution. Non-MapReduce computing has advantages in executing iterative machine learning algorithms under LOGO. Below, we use three types of machine learning algorithms for supervised learning, unsupervised learning, and pattern mining to demonstrate how these algorithms work under LOGO.

3.1 Supervised Learning Algorithms In the LOGO framework, a supervised learning algorithm trains a model on each computing node using local RSP blocks in the local phase and aggregates all local models into the final model in the global phase. Local Phase. It follows the routine below.

(i) The local function is executed on each LO worker. It divides the local input RSP partition into a training set and a testing set.

(ii) It calls the corresponding supervised learning algorithm and generates a

local model on the training set.

(iii) It obtains the accuracy of the local model based on the testing set.

(iv) It sends the local model to the GO worker. Global Phase. Given the local models produced by LO workers, we proceed to introduce how to design the global function that merges the local models. There exist different methods to combine them, e.g., weighted averaging[22–24], voting[25–27], relearning[26, 28, 29], etc.We adopt the idea of the voting method and propose global function G_SupervisedLearning. It takes a set of local models, each of which is associated with an accuracy. The set of local models is sorted in ascending order of their accuracy. Based on the central limit theorem[30], the first and last 5% of the ranked local modes are removed, which helps increasing the confidence of the prediction results. Let $M$ be the set of remaining local models. For each data record in the input dataset $D$, it produces the predicted label by taking the majority voting [31] of the local models in $M$.

### *3.2 Unsupervised Learning Algorithms*

In the LOGO framework, a unsupervised learning algorithm performs clustering on each computing node using local RSP blocks in the local phase. The clustering results on all computing nodes are aggregated in the global phase. Local Phase. It follows the routine below.

(i) The local function is executed on each LO worker. It calls the corresponding unsupervised learning algorithm in SMILE (https://pypi.org/project/smile) and generates a local model on the local RSP partition.

(ii) Each local model consists of a set of cluster centers, which is sent to the GO worker.

***Global Phase.*** Given the sets of cluster centers generated by LO workers, the global function executes a clustering process on all local cluster centers and returns new cluster centers □□Then, each data point is assigned to its nearest cluster centers in □□

### *3.3 Frequent Itemset Mining Algorithms*

In the LOGO framework, we design and implement two algorithms for the frequent itemset mining, i.e., FIM_I and FIM_II. Both algorithms compute approximate frequent itemsets using a set of $m$ RSP partitions, given the support threshold $\delta$. Algorithm FIM_I contains three times of communication. Firstly, each LO worker computes the local frequent itemsets $SL$ by executing an existing frequent itemset mining algorithm, e.g., FPGrowth algorithm [32], on the local RSP partition. Then, the LO workers send their local results (itemset-count pairs) to the GO worker. After the GO worker unions the received itemsets and aggregates the count by itemset, it broadcasts the unioned set $SG$ to the LO workers. Next, each LO worker computes the counts of the itemsets in $SL \backslash SG$ and send the results to the GO worker. In the end, the GO worker updates the counts of the itemsets in $SG$ and finalizes the

frequent itemsets. The support of each itemset is calculated by count $/(m\times T)$, where $m$ is the number of RSP partitions and $T$ is the number of transactions in each RSP partition. Fig. 5 illustrates the workflow of algorithm FIM_I on the left side. In step○1 , three LO workers compute their local frequent itemsets $SL$ and send the local results to the GO worker. In step○2 , the GO worker broadcasts the unioned set $SG$={{$a$, $b$}, {$b$,$c$}, {$c$,$d$}, {$d$,$e$}} to LO workers. In step ○3 , each LO worker computes the counts of the itemsets in $SG\backslash SL$. Then, to the GO worker, LO Worker I sends ({$c$,$d$}, 4), LO Worker II sends ({$d$,$e$}, 4), and LO Worker III sends ({$d$,$e$}, 4). In the end, the GO worker finalizes the counts of the itemsets. Given $m$= 3, $T$= 10, and the support threshold $\delta$= 0.5, the frequent itemsets are {$a$, $b$}, {$b$,$c$}, and {$c$,$d$}.

Algorithm FIM_II only communicates once. Firstly, similar to algorithm FIM_I, all LO workers compute frequent itemsets on their own RSP partitions and send the local results to the GO worker. Besides unioning and aggregating the received local results, the GO worker additionally collects the votes of each itemset from LO workers. If an itemset appears in the local results of a LO worker, it receives one vote [33]. Then, at the GO worker side, for the itemsets whose voting ratio votes/$m$ is larger than $\lambda$, its count is updated as count = count/votes. Next, the support of each itemset is calculated by count/$T$. In the end, the frequent itemsets are finalized according to the given support threshold $\delta$. Fig. 5 illustrates the workflow of algorithm FIM_II on the right side, where $\lambda$ = 50%, $m$ = 3, $T$= 10, and $\delta$= 0.5. In step ○1 , three LO workers compute their local frequent itemsets and send the local results to the GO worker. Then, the GO worker aggregates the counts by itemsets and collects the votes. In step○2 , since the voting ratio of itemset {$d$,$e$} is 1/3 which is less than λ= 50%, it is discarded. Next, the counts of the other three itemsets are updated. In the end, the frequent itemsets are {$a$, $b$}, {$b$,$c$}, and {$c$,$d$}.

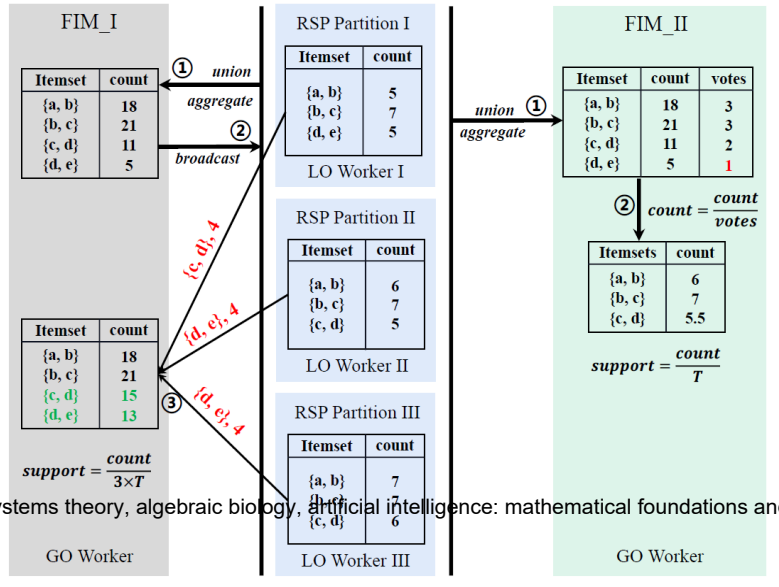
## *4. Performance*

To demonstrate the efficiency and the effectiveness of the LOGO computing framework, we compare it with three configurations of Spark:

• Spark: Running algorithms on the Spark by taking the entire dataset.

• Online-Sample-Spark: First calling the built-in sampling operator in the Spark to get a sample data (5% of the original dataset) and then running algorithms on the sample data.

• Sample-Spar




the time cost of the sampling ope

All the evaluat

Among them, 24 nodes each has t

*Fig. 5: Frequent itemset mining algorithms in LOGO.*

16 cores at 2.6GHz and 128GB main memory, and 8 nodes each has two Intel Xeon E5-2630 CPUs with 12 cores at 2.6GHz and 128GB main memory. Spark 2.4.0 and YARN 3.0.0 are running on the cluster. For each algorithm, we use YARN to allocate at most 100 executors and 1.6TB main memory.

### *4.1 Performance of the supervised learning algorithms*

We use 100-dimensional datasets to evaluate the performance of four supervised learning algorithms: decision tree, random forest, logistic regression, and SVM. Figures 6 and 7 show the running time and the accuracy of four supervised learning algorithms on LOGO and the three configurations of Spark, when varying the dataset size from 50GB to 10TB. The running time of Spark increases exponentially as the dataset size increases. Its computational cost is high when the dataset size is larger than 1.5TB. It is because that Spark executes the supervised learning algorithms on the entire dataset and the iterations in the algorithms cause high communication cost. Both Online-Sample-Spark and Sample-Spark execute the supervised learning algorithms on a sample of the entire dataset. Thus, the running time of them increase linearly as the dataset size increases. However, when the dataset size is larger than 2TB, the running time of Online-Sample-Spark rises rapidly. It is because the online sampling is slow on large datasets. Excluding the cost of the online sampling, it is observed that Sample-Spark and LOGO are close in terms of the running time. Nevertheless, LOGO is faster than Sample-Spark. The reason is that on the same amount of data, Sample-Spark has communication cost at the end of each iteration, while LOGO performs iterations locally and only has communication cost when all iterations terminate. Excluding the online sampling operation, Online-Sample-Spark and Sample-Spark are the same and have the same results and accuracy. Spark learns models on the entire dataset, while Online-Sample-Spark and Sample-Spark have the same learning process but using a sample of the dataset. Hence, the accuracy of Spark is higher than Online-Sample-Spark and Sample-Spark. For the decision tree and the random forest algorithms, the accuracy of LOGO is better than that of Spark. It is because that the learning process in LOGO is similar to the ensemble learning which

has been shown as an effective strategy for the decision tree and the random forest algorithms [34, 35]. Thus, although LOGO learns models on a sample of the dataset, it achieves better performance than Spark. Whereas, for the logistic regression and the SVM algorithms, the accuracy of LOGO is close to that of Spark. The reason might be that the accuracy of Spark is high, i.e., above 90%, so that the room of improvement is limited.

### *4.2 Performance of the unsupervised learning algorithms*

We use 100-dimensional datasets to evaluate the performance of two unsupervised learning algorithms: Kmeans and Bisecting Kmeans. Figures 8 and 9 show the running time, the purity, and the C-dist of two unsupervised learning algorithms on LOGO and the three configurations of Spark, when varying the dataset size from 50GB to 10TB.

Similar to the evaluation results of the supervised learning algorithms, the running time of Spark increases rapidly and it is impractical for Spark to deal with dataset larger than 1TB. The online sampling operation is expensive when the dataset size is larger than 2TB. Excluding the cost of online sampling, the running time of Sample-Spark and LOGO are similar for the Kmeans algorithm and LOGO is faster than Sample-Spark regarding the bisecting Kmeans algorithm.

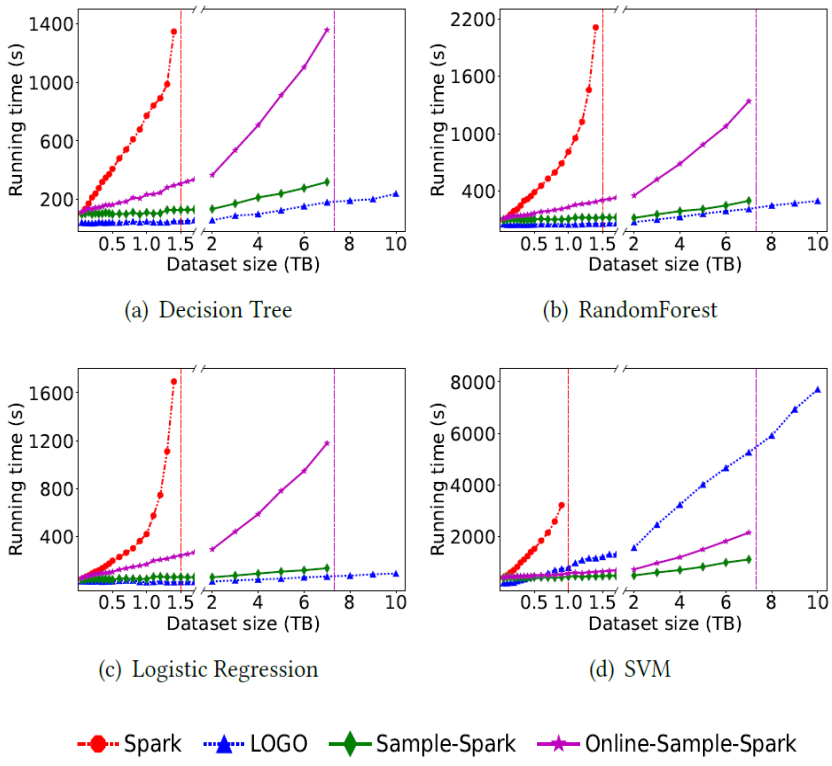



*Fig. 6: Running time of the supervised learning.*

The reason is that on the same amount of data, the communication cost of LOGO is lower than that of Sample-Spark. We evaluate the quality of the discovered clusters in terms of the Purity (the higher the better) and the C-dist (the lower the better). As shown in Figure 9, having the same clustering process, since Spark uses the entire dataset while Sample-Spark and Online-Sample-Spark use a sample data, Spark achieves better performance. The quality of the clusters found by LOGO is similar to that of Spark.

### *4.3 Performance of frequent itemset mining algorithms*

Fig. 10(a) shows the running times of frequent itemset mining algorithms on Spark and LOGO when varying the number of transactions. The support threhold is set to 0.15. For the datasets containing less than 2.75 millions transactions, we organize 5000 transactions as one data block. For the datasets with more than 2.75 millions transactions, one data block contains 1 million transactions. The running time of Spark is extremely high when handling 2.75 millions of transactions. The reason is that the FP-Growth algorithm in the Spark library [36] builds an distributed FP-tree on all partitions and recursively extracts frequent itemsets. It generates a large number of intermediate variables, which requires large memory size and is time-consuming. Both Online-Sample-Spark and Sample-Spark also adopt the FP-Growth algorithm in the Spark library.

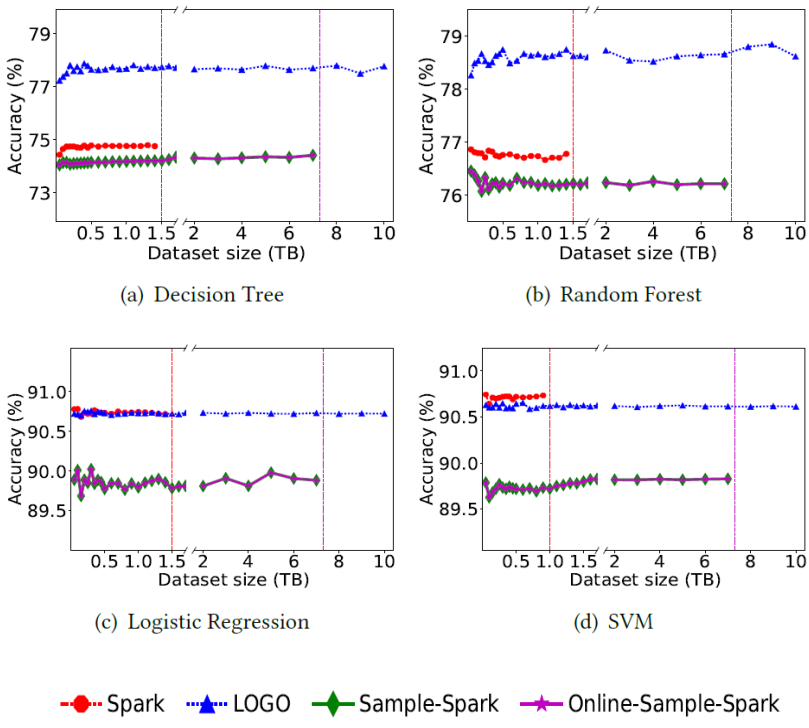


*Fig. 7: Accuracy of the supervised learning.*

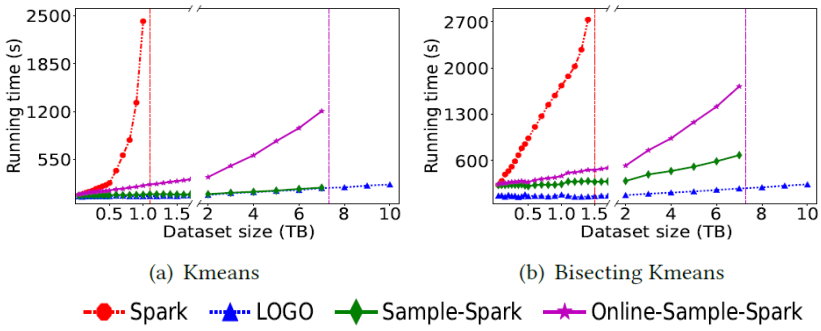
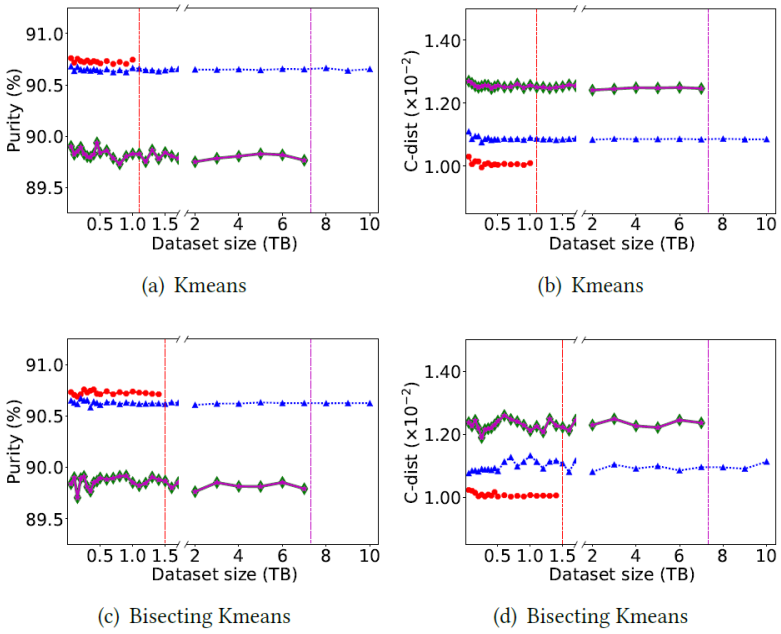



*Fig. 8: Running time of the unsupervised learning.*

Due to the same reason of Spark, they cannot respond in a reasonable time when the number of transactions exceeds 60 millions. Algorithms FIM_I and FIM_II on LOGO are able to handle 1 billion transactions. Their running times increase sub-linearly as the number of transactions increases. The computational cost of algorithm FIM_I is slightly higher than that of algorithm FIM_II, because FIM_I contains one more round of communications. Figures 10(b) and 10(c) show the precision and the recall of algorithms FIM_I and FIM_II on LOGO, by taking the frequent itemsets found by Spark as the ground truth. The precision and the recall of FIM_I fluctuates around 98% when the number of transactions varies from 105 to 107.

The precision of FIM_II is below 90% when the number of transactions varies from 105 to 107. The reason is that both algorithms use the frequent itemsets in samples to approximate the real results. It happens that some infrequent itemsets turn to be frequent in the samples. When the dataset is small, this randomness affects the precision of the results.



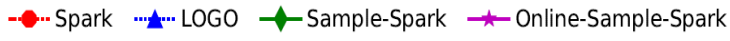
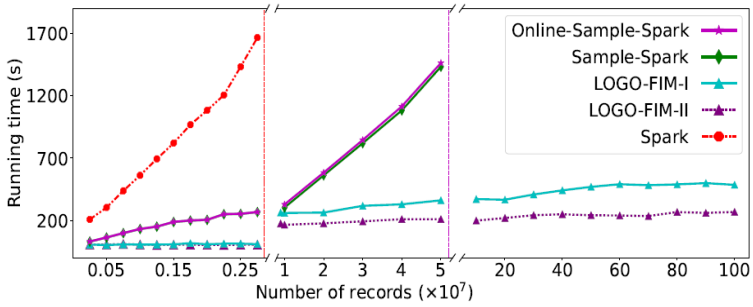
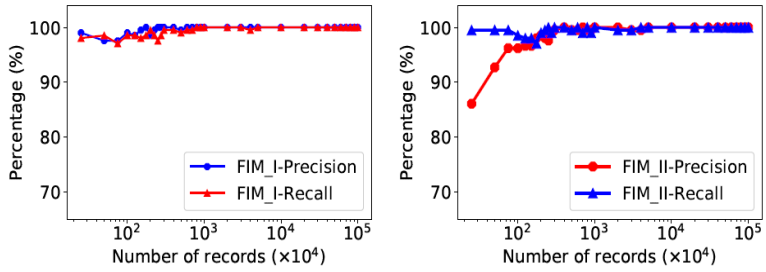



*Fig. 9: Purity and C-dist of the unsupervised learning.*




(a) Running time.




(b) FIM-I. (c) FIM-II.

*Fig. 10: Frequent itemset mining.*

However, when the dataset is large, this effect can be ignored. As shown in the figures, the precision and the recall of FIM_I and FIM_II are close to 100% when the number of transactions exceeds 108. Overall, FIM_I is better than FIM_II in terms of the accuracy of the results. Because FIM_I has one more round of communication to share information among the computing nodes. Whereas, FIM_II is better than FIM_I in terms of the running time. Hence, for the frequent itemset mining, LOGO is superior to Spark, Online-Sample-Spark and Sample-Spark in terms of running time and the scalability. Moreover, the approximate results provided by algorithms FIM_I and FIM_II are close to the ground truth.

## *5. Conclusions*

In this paper, we propose a Non-MapReduce paradigm for distributed big data analytics. It takes random sample partitions instead of the entire dataset as the input. By adopting this approach, iterative algorithms perform iterations on data partitions parallelly and independently, so that the I/O and the communication costs are significantly reduced. The output is the aggregation of the results obtained from data partitions. We implement the Non-MapReduce paradigm by extending Spark and introduce the LOGO computing framework. It is I/O and communication efficient, capable of performing iterative analytical algorithms on terabytescale datasets, and

has simple parallel programming model. We also apply the LOGO framework to commonly used supervised learning, unsupervised learning, and frequent itemset mining algorithms. The experimental results demonstrate the efficiency and effectiveness of LOGO.

### *1. Введение*

Аналитика больших данных позволяет принимать решения на основе данных, что нашло успешное применение в «умных» городах [1], обнаружении мошенничества [2–4], диагностике и прогнозировании заболеваний [5, 6], промышленном производстве [7, 8], прогнозировании погоды. [9] и т. д. В настоящее время мы наблюдаем массовый рост спроса на обработку и анализ больших данных. Согласно отчету Statista (https://www.statista.com), общий объем данных, генерируемых в мире, составит около 97 зеттабайт в 2022 году, и ожидается, что к 2025 году он вырастет до 181 зеттабайт. Об этом свидетельствует Precedence Согласно исследованию (https://www.precedenceresearch.com/data-analytics-market), размер мирового рынка анализа данных в 2022 году составил 41,39 миллиарда долларов США, а к 2030 году, по прогнозам, он превысит 346,33 миллиарда долларов США.

В настоящее время для обработки больших данных широко применяются распределенные вычислительные платформы и системы [10], поскольку обрабатывать и анализировать большие данные с помощью одной машины непомерно дорого или физически невозможно. Популярные распределенные платформы общего назначения, такие как Hadoop (https://hadoop.apache.org) и Spark (https://hadoop.apache.org), используют архитектуру MapReduce, показанную на рис. 1. Большие файлы данных делятся на более мелкие блоки и хранятся в распределенной файловой системе. Задачи анализа выполняются путем проведения итераций операций отображения и преобразования над этими блоками. Итерация состоит из фаз Mapper и Reducer. Операция перетасовки между фазами Mapper и Reducer предполагает передачу данных N-N. Накладные расходы на связь возникают на каждой итерации и пропорциональны количеству итераций в алгоритме [11].

Собственные распределенные системы машинного обучения, такие как TensorFlow [12], MXNeT [13], DIANNE [14] и PyTorch [15], следуют архитектуре сервера параметров, показанной на рисунке 2. Каждый рабочий узел вычисляет локальные параметры, используя локальные данные. ЦП отвечает за инициализацию, хранение и обновление параметров, а графический процессор преобразует входные данные в тензоры и обучает модель. Затем параметры глобальной модели агрегируются и обновляются на узлах сервера. Этот процесс является итеративным, где на каждой итерации рабочие узлы должны взаимодействовать с серверным узлом.

Обычно используемые аналитические алгоритмы, такие как классификация [16], кластеризация [17] и анализ шаблонов [18], являются итеративными, которые требуют многократного сканирования и обработки всего набора данных. Следовательно, архитектуры MapReduce и сервера параметров имеют

*Рис.1: Архитектура MapReduce.*

*Рис. 2: Архитектура сервера параметров.*

следующие ограничения при выполнении итерационных алгоритмов обработки больших данных.

• Высокая стоимость связи. В MapReduce и сервере параметров стоимость связи возникает между двумя последовательными итерациями при выполнении итерационных алгоритмов. Всем машинам необходимо отправлять и получать данные и параметры перед переходом на следующую итерацию. Таким образом, стоимость связи существенно возрастает по мере увеличения количества итераций и объема данных.

• Высокие требования к оперативной памяти. Когда объем данных превышает емкость основной памяти (ОЗУ) в системе, это приводит к частой подкачке памяти, при которой данные перемещаются между основной памятью и дополнительным хранилищем, что занимает много времени. Spark снижает затраты на ввод и вывод данных за счет использования абстракции распределенных данных в памяти, называемой устойчивыми распределенными наборами данных (RDD). Однако для обработки больших данных требуется большой объем оперативной памяти. Это непрактичное решение при быстром росте объема данных.

• Неудобная модель параллельного программирования. Преобразование аналитических алгоритмов в соответствующие параллельные версии в существующих распределенных системах утомительно и сложно. Программистам приходится переписывать аналитические алгоритмы, используя доступные программные интерфейсы. Более того, отладка и оптимизация параллельных программ зачастую сложна и требует много времени.

Вышеупомянутые ограничения можно устранить путем разделения большого набора данных на блоки данных, которые представляют собой случайные выборки большого набора данных, например, модель данных разделения случайной выборки (RSP) [19]. Такое представление данных позволяет обрабатывать каждый блок данных независимо в вычислительном узле с помощью итеративного алгоритма, а также обрабатывать множество блоков данных с помощью одного и того же алгоритма в массовом параллельном режиме. Результаты блока данных затем передаются на главный узел для вычисления окончательного результата. Этот подход без MapReduce, который мы обсудим в этой статье, имеет следующие преимущества в анализе

больших данных.

(i) Масштабируемость. При обработке растущего объема данных, например данных терабайтного масштаба, итеративные алгоритмы способны реагировать в разумные сроки.

(ii) Эффективность ввода-вывода. Если разделы данных и промежуточные результаты не помещаются в памяти, подкачка памяти выполняется нечасто.

(iii) Сокращение общения. При выполнении итераций затраты на связь между вычислительными узлами остаются низкими.

(iv) Простая модель параллельного программирования для итерационных алгоритмов. Сложные итеративные алгоритмы можно легко упаковать в виде операторов для параллельного выполнения.

Оставшаяся часть этой статьи организована следующим образом. Сначала мы рассмотрим основные технологии вычислений без MapReduce — модель данных RSP и структуру LOGO, а также их реализацию в Spark. Затем мы используем три типа алгоритмов машинного обучения: обучение с учителем, обучение без учителя и анализ шаблонов, чтобы проиллюстрировать основные этапы вычислений без использования MapReduce при выполнении этих итерационных алгоритмов над большими наборами данных. Наконец, мы предоставляем смоделированные результаты, чтобы продемонстрировать производительность, эффективность и масштабируемость данных вычислений без MapReduce при запуске алгоритмов машинного обучения над большими наборами данных до 10 ТБ. Мы завершаем эту статью кратким обзором преимуществ вычислений без MapReduce в анализе больших данных.

## *2. Вычисления без использования Mapreduce*

Вычисления без использования MapReduce основаны на двух основных технологиях: модели данных RSP [19], которая позволяет использовать блоки данных распределенного файла данных в качестве случайных выборок файла больших данных, и платформе LOGO, которую мы предлагаем в этой статье для включить вычисления без MapReduce. Эти две технологии подробно описаны в следующих двух разделах, за которыми следует реализация в Spark.

### *2.1 Модель данных RSP*

Распределенный файл данных сохраняется как набор блоков данных, которые распределены по узлам кластера и управляются распределенной файловой системой, например HDFS [20]. Раздел случайной выборки или модель данных RSP представляет собой большой набор данных в виде раздела блоков данных RSP [19], которые также сохраняются распределенным образом и управляются HDFS. Однако каждый блок данных RSP создается с помощью алгоритма разделения случайной выборки [21] случайной выборки большого набора данных, поэтому его можно использовать для вычисления

статистических оценок большого набора данных.

Формально пусть $D=\{r_1, r_2, \ldots, r_N\}$ это многомерный набор данных из N записей. Каждая запись данных $r_i=\{x_1, x_2, \ldots, x_M\}$ имеет M признаков, где каждый признак xi является случайной величиной. Пусть T — алгоритм разделения случайной выборки, который делит D на K непересекающихся подмножеств $D_1, D_2, \ldots, D_K$ , которые удовлетворяют следующим трем условиям:

Formally, let $D=\{r_1, r_2, \ldots, r_N\}$ be a multivariate data set of *N* records. Each data record $r_i=\{x_1, x_2, \ldots, x_M\}$ has *M* features, where each feature $x_i$ is a random variable. Let *T* be a random sample partitioning algorithm that divides *D* into *K* nonoverlapping subsets $D_1, D_2, \ldots, D_K$ which satisfy the following three conditions:

$$(i)\, D=U_{i=1}^{K} D_i, (ii)\, D_i \cap D_j=\varnothing, 1\leqslant i, j\leqslant K, i\neq j,$$

$$(iii)\, |F(D_i)-F(D)|<\delta, 1\leqslant i\leqslant K \quad ,$$

где F() — кумулятивная функция распределения (CDF) набора данных, а δ — небольшая положительная константа. Это условие гласит, что CDF всех блоков данных RSP должны быть аналогичны CDF набора данных.

Набор $D_1, D_2, \ldots, D_K$ называется моделью данных RSP D. Алгоритм T в [21] работает следующим образом: первоначально блоки данных хранятся на n машинах (также известных как узлы). Предположим, что каждый узел хранит K блоков данных. На каждом узле все записи в каждом блоке данных рандомизируются, а затем каждый блок данных разбивается на K мини-блоков. Эта процедура выполняется параллельно на всех узлах. После этого каждый узел получает по одному мини-блоку от каждого из всех остальных узлов, а затем объединяет их со своим локальным мини-блоком. В конце исходный набор данных реорганизуется в блоки RSP и сохраняется в HDFS. На рисунке 3 показан рабочий процесс программы.

### *2.2 Структура логотипа*

Учитывая набор данных D в качестве модели данных RSP, в вычислениях без MapReduce аналитическая задача D выполняется в рамках вычислительной среды локальных операций с глобальными операциями (LOGO) на товарных кластерах. Архитектура LOGO состоит из двух уровней: уровня данных и уровня вычислений, как показано на рис. 4.

На уровне данных файлы данных принимают форму блоков RSP, которые распределены по n узлам. Обычно блок RSP соответствует случайной выборке файла данных, подлежащей анализу. На вычислительном уровне процесс выполнения аналитической задачи состоит из локальной фазы и глобальной фазы. Один узел отвечает за глобальную фазу, называемую мастером GO. Все узлы могут присоединиться к локальной фазе, называемой рабочими LO.

*Рис. 3: Рабочий процесс алгоритма T.*

Локальная фаза. Каждый исполнитель LO выполняет локальную операцию над своим собственным блоком данных RSP, не обмениваясь данными с другими исполнителями. Локальная операция может быть простым алгоритмом или очень итеративным. В конце локального этапа создается локальная модель на основе одного блока данных RSP. Глобальная фаза.

*Рис. 4.*
*① Стоимость ввода-вывода: чтение блоков RSP. ② Локальные итерации. ③ Стоимость связи: передача местных результатов. ④ Стоимость ввода-вывода: запись окончательных результатов.*

Когда все рабочие LO завершают свои локальные задачи, мастер GO получает уведомление и использует удаленные вызовы процедур для чтения локальных моделей от всех рабочих LO. Он выполняет глобальную операцию, объединяющую все локальные модели и сохраняющую окончательную модель в хранилище. В отличие от MapReduce, который выполняет итерационный алгоритм путем итерации множества пар операций отображения и сокращения, возникающих из-за больших затрат на передачу данных, в рамках LOGO итерационный алгоритм завершает выполнение в рабочих процессах LO параллельно без необходимости передачи данных. Следовательно, он очень эффективен при выполнении итерационных алгоритмов.

### *2.3 Реализация в Spark*

Вычисления без использования MapReduce с моделью данных RSP и вычислительной платформой LOGO реализованы в HDFS (https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html) и Spark. Следующие три основных компонента подробно описаны ниже. (i) Абстракция RspRDD. Spark использует RDD для обмена данными между вычислениями. Мы следуем этой идее и создаем абстракцию совместного использования данных под названием RspRDD, расширяя RDD в Spark. Блоки RSP в HDFS представлены в Spark объектом RspRDD, которым можно манипулировать параллельно, например, класс RspRDD[T: ClassTag](prev: RDD[T]) расширяет RDD[T](prev){...} (ii) Преобразование RSP. Мы предоставляем функцию RandomizeAndSlice() для рандомизации записей в блоке данных и разрезания блока данных на K мини-блоков. Пользователи могут использовать его в следующем фрагменте кода ниже (Spark API в Scala), чтобы преобразовать блоки данных в HDFS в блоки данных RSP. В строке 1 указывается файл данных, который необходимо преобразовать, и количество K разделов в RDD. В строке 2 функция RandomizeAndSlice() передается функции Spark API MapPartitions(), так что

локальная рандомизация и нарезка выполняются для каждого блока данных на всех узлах. Затем каждому мини-блоку назначается целочисленный ключ в диапазоне [1,K]. Строка 3 вызывает функцию Spark API уменьшитьByKey(), так что мини-блоки передаются и объединяются, в результате чего на каждом узле создаются блоки RSP. Наконец, строка 4 сохраняет эти блоки RSP в файле HDFS.

1: val RDD1 = sparkContext.textFile(путь к файлу, K)
2: val RDD2 = RDD1.mapPartitions(RandomizeAndSlice())
3: val rspRDD = RDD2.reduceByKey()
4: rspRDD.saveAsTextFile(rspfilepath)

(iii) API-интерфейсы логотипа. Локальная операция и глобальная операция в структуре LOGO выражаются функциями LO() и GO() соответственно. Входные блоки RSP как разделы RDD сначала обрабатываются функцией LO(), которая выводит локальную модель для каждого раздела RSP. Функция GO() объединяет все локальные модели, созданные функцией LO(), и выводит окончательную модель. API-интерфейсы LOGO поддерживают несколько языков программирования, таких как Java, Scala и Python. Подробности функций LOGO API описаны ниже.

• rspTextFile(rspfilepath): загружает файл данных RSP в HDFS и возвращает объект RspRDD, который представляет блоки данных RSP как разделы RSP в RDD.

• getSubPartitions(m): случайным образом выбирает m% разделов RSP.

• LO(L_Algorithm()): принимает функцию L_Algorithm(), которая определяется пользователем и может быть любым аналитическим алгоритмом. Он отправляет функцию L_Algorithm() всем работникам LO. Каждый исполнитель LO применяет функцию L_Algorithm() к своему разделу RSP, а затем создает локальную модель из каждого раздела RSP. Аналогично, допустимо также использование LO(preProcess()), которое определяет функцию подготовки данных для каждого раздела RSP. Например, запуск LO(preProcess()) перед LO(L_Algorithm()) указывает на предварительную обработку данных перед моделированием.

• GO(G_Algorithm()): принимает определяемую пользователем функцию G_Algorithm(), которая объединяет все локальные модели и создает окончательную модель. Клиентская программа. В приведенном ниже фрагменте кода в Scala показан пример пользовательского приложения с использованием LOGO. Строка 1 подготавливает объект RspRDD для указанного файла данных RSP. Строка 2 принимает разделы RSP. Строка 3 предварительно обрабатывает каждый раздел данных. Строка 4 выполняет функцию LO() для каждого раздела RSP и возвращает m локальных моделей. В строке 5 выполняется функция GO(), которая дает конечный результат путем слияния m локальных моделей. Строка 6 сохраняет окончательную модель в HDFS по пути к файлу.

```
1: val rspRDD = sparkContext.rspTextFile(rspfilepath)
2: val dataRDD = rspRDD.getSubPartitions(m)
3: val trainRDD = dataRDD.LO(preProcess())
4: val modelRDD = trainRDD.LO(L_Algorithm())
5: val resultRDD = modelRDD.GO(G_Algorithm())
6: resultRDD.save(путь к файлу)
```

Из этого примера видно, что с помощью LOGO становится проще написать пользовательское приложение для аналитической задачи.

### *3. Алгоритмы машинного обучения под LOGO*

Алгоритмы машинного обучения широко используются в анализе больших данных, и большинство из них итеративно обрабатывают наборы данных для поиска оптимального решения. Вычисления, отличные от MapReduce, имеют преимущества при выполнении итеративных алгоритмов машинного обучения под LOGO. Ниже мы используем три типа алгоритмов машинного обучения: обучение с учителем, обучение без учителя и анализ шаблонов, чтобы продемонстрировать, как эти алгоритмы работают под LOGO.

3.1 Алгоритмы обучения с учителем В структуре LOGO алгоритм обучения с учителем обучает модель на каждом вычислительном узле с использованием локальных блоков RSP на локальной фазе и объединяет все локальные модели в окончательную модель на глобальной фазе. Локальная фаза. Это следует процедуре, приведенной ниже.

(i) Локальная функция выполняется на каждом работнике LO. Он делит локальный входной раздел RSP на обучающий набор и тестовый набор.

(ii) Он вызывает соответствующий алгоритм обучения с учителем и генерирует локальную модель на обучающем наборе.

(iii) Он получает точность локальной модели на основе тестового набора.

(iv) Он отправляет локальную модель работнику GO. Глобальная фаза. Учитывая локальные модели, созданные работниками LO, мы переходим к описанию разработки глобальной функции, объединяющей локальные модели. Существуют различные методы их объединения, например, взвешенное усреднение[22–24], голосование[25–27], повторное обучение[26, 28, 29] и т. д. Мы принимаем идею метода голосования и предлагаем глобальную функцию G_SupervisedLearning. Требуется набор локальных моделей, каждая из которых связана с определенной точностью. Набор локальных моделей отсортирован в порядке возрастания их точности. На основании центральной предельной теоремы[30] удаляются первые и последние 5% ранжированных локальных мод, что помогает повысить достоверность результатов прогнозирования. Пусть M — множество оставшихся локальных моделей. Для каждой записи данных во входном наборе данных D он создает прогнозируемую метку, приняв большинство голосов [31] локальных моделей в M.

### *3.2 Алгоритмы обучения без учителя*

В структуре LOGO алгоритм обучения без учителя выполняет кластеризацию на каждом вычислительном узле с использованием локальных блоков RSP на локальной фазе. Результаты кластеризации на всех вычислительных узлах агрегируются на глобальной фазе. Локальная фаза. Это следует процедуре, приведенной ниже.

(i) Локальная функция выполняется на каждом работнике LO. Он вызывает соответствующий алгоритм неконтролируемого обучения в SMILE (https://pypi.org/project/smile) и генерирует локальную модель в локальном разделе RSP.

(ii) Каждая локальная модель состоит из набора кластерных центров, который отправляется работнику GO.

Глобальная фаза. Учитывая наборы центров кластеров, сгенерированные рабочими LO, глобальная функция выполняет процесс кластеризации во всех локальных центрах кластеров и возвращает новые центры кластеров □□Затем каждая точка данных присваивается ближайшему центру ее кластера в □□

### *3.3 Часто встречающиеся алгоритмы интеллектуального анализа наборов элементов*

В структуре LOGO мы разрабатываем и реализуем два алгоритма для частого анализа наборов элементов, то есть FIM_I и FIM_II. Оба алгоритма вычисляют приблизительные часто встречающиеся наборы элементов, используя набор из m разделов RSP, учитывая порог поддержки δ. Алгоритм FIM_I содержит три раза связи. Во-первых, каждый исполнитель LO вычисляет локальные часто встречающиеся наборы элементов SL, выполняя существующий алгоритм анализа часто встречающихся наборов элементов, например, алгоритм FPGrowth [32], в локальном разделе RSP. Затем рабочие LO отправляют свои локальные результаты (пары «набор элементов-количество») рабочему GO. После того как рабочий GO объединяет полученные наборы элементов и суммирует счетчик по набору элементов, он передает объединенный набор SG рабочим LO. Затем каждый исполнитель LO вычисляет количество наборов элементов в SL\SG и отправляет результаты работнику GO. В конце концов, исполнитель GO обновляет количество наборов элементов в SG и завершает частые наборы элементов. Поддержка каждого набора элементов рассчитывается по счетчику, где m — количество разделов RSP, а T — количество транзакций в каждом разделе RSP. На рис. 5 показан рабочий процесс алгоритма FIM_I слева. На шаге○1 три исполнителя LO вычисляют свои локальные часто встречающиеся наборы элементов SL и отправляют локальные результаты работнику GO. На шаге ○2 исполнитель GO передает объединенный набор SG={{a, b}, {b,c}, {c,d}, {d,e}} работникам LO. На шаге

○3 каждый исполнитель LO вычисляет количество наборов элементов в SG\SL. Затем работнику GO работник LO I отправляет ({c,d}, 4), работник LO II отправляет ({d,e}, 4), а работник LO III отправляет ({d,e}, 4). В конце концов, работник GO завершает подсчет наборов элементов. Учитывая m = 3, T = 10 и порог поддержки δ = 0,5, частыми наборами элементов являются {a, b}, {b,c} и {c,d}.

Алгоритм FIM_II обменивается данными только один раз. Во-первых, аналогично алгоритму FIM_I, все исполнители LO вычисляют часто встречающиеся наборы элементов в своих собственных разделах RSP и отправляют локальные результаты работнику GO. Помимо объединения и агрегирования полученных локальных результатов, работник GO дополнительно собирает голоса каждого набора элементов от работников LO. Если набор элементов появляется в локальных результатах работника LO, он получает один голос [33]. Затем на стороне рабочего GO для наборов элементов, у которых коэффициент голосования голосов/m больше λ, их счетчик обновляется как count = count/votes. Затем поддержка каждого набора элементов рассчитывается по числу count/T. В конце концов, наборы частых элементов дорабатываются в соответствии с заданным порогом поддержки δ. На рис. 5 справа показан рабочий процесс алгоритма FIM_II, где λ = 50%, m = 3, T = 10 и δ = 0,5. На шаге ○1 три исполнителя LO вычисляют свои локальные часто встречающиеся наборы элементов и отправляют локальные результаты работнику GO. Затем исполнитель GO суммирует подсчеты по наборам элементов и собирает голоса. На шаге ○2, поскольку коэффициент голосования набора элементов {d,e} равен 1/3, что меньше λ = 50%, он отбрасывается. Затем обновляются счетчики трех других наборов элементов. В конце концов, наиболее часто встречающимися наборами элементов являются {a, b}, {b,c} и {c,d}.

### *4. Производительность*

Чтобы продемонстрировать эффективность и результативность вычислительной среды LOGO, мы сравниваем ее с тремя конфигурациями Spark:

• Spark: запуск алгоритмов в Spark с использованием всего набора данных.

• Online-Sample-Spark: сначала вызывается встроенный оператор выборки в Spark, чтобы получить выборочные данные (5 % исходного набора данных), а затем выполняются алгоритмы на выборочных данных.

• Sample-Spark: то же самое, что Online-Sample-Spark, но без учета временных затрат на операцию выборки.

Все оценки выполняются в кластере, состоящем из 32 вычислительных узлов. Среди них 24 узла имеют по два процессора Intel Xeon E5-2650 с

*Рис. 5: Алгоритмы анализа часто встречающихся наборов элементов в LOGO.*

16 ядер с частотой 2,6 ГГц и 128 ГБ основной памяти, а 8 узлов имеют по два процессора Intel Xeon E5-2630 с 12 ядрами с частотой 2,6 ГГц и 128 ГБ основной памяти. В кластере работают Spark 2.4.0 и YARN 3.0.0. Для каждого алгоритма мы используем YARN, чтобы выделить не более 100 исполнителей и 1,6 ТБ оперативной памяти.

### *4.1. Производительность алгоритмов обучения с учителем*

Мы используем 100-мерные наборы данных для оценки производительности четырех алгоритмов обучения с учителем: дерево решений, случайный лес, логистическая регрессия и SVM. На рисунках 6 и 7 показано время работы и точность четырех контролируемых алгоритмов обучения в LOGO и трех конфигурациях Spark при изменении размера набора данных от 50 ГБ до 10 ТБ. Время работы Spark увеличивается экспоненциально по мере увеличения размера набора данных. Его вычислительные затраты высоки, если размер набора данных превышает 1,5 ТБ. Это связано с тем, что Spark выполняет контролируемые алгоритмы обучения для всего набора данных, а итерации в алгоритмах вызывают высокие затраты на связь. И Online-Sample-Spark, и Sample-Spark выполняют алгоритмы контролируемого обучения на выборке всего набора данных. Таким образом, время их работы увеличивается линейно по мере увеличения размера набора данных. Однако если размер набора данных превышает 2 ТБ, время работы Online-Sample-Spark быстро возрастает. Это связано с тем, что онлайн-выборка больших наборов данных выполняется медленно. Если исключить стоимость онлайн-семплирования, можно заметить, что Sample-Spark и LOGO близки по времени работы. Тем не менее, LOGO работает быстрее, чем Sample-Spark. Причина в том, что для одного и того же объема данных Sample-Spark имеет стоимость связи в конце каждой итерации, тогда как LOGO выполняет итерации локально и имеет стоимость связи только после завершения всех итераций. За исключением операции онлайн-выборки, Online-Sample-Spark и Sample-Spark одинаковы и имеют одинаковые результаты и точность. Spark изучает модели на всем наборе данных, в то время как Online-Sample-Spark и Sample-Spark имеют тот же процесс обучения, но используют образец набора данных. Следовательно, точность Spark выше, чем Online-Sample-Spark и Sample-Spark. Для алгоритмов дерева решений и случайного леса точность LOGO выше, чем у Spark. Это связано с тем, что процесс обучения в LOGO аналогичен ансамблевому обучению, которое было показано как эффективная стратегия для алгоритмов дерева решений и случайного леса [34, 35]. Таким образом, хотя LOGO изучает модели на выборке набора данных, он обеспечивает более высокую производительность, чем Spark. Тогда как для логистической

регрессии и алгоритмов SVM точность LOGO близка к точности Spark. Причина может заключаться в том, что точность Spark высока, т. е. превышает 90%, поэтому возможности для улучшения ограничены.

### *4.2 Производительность алгоритмов неконтролируемого обучения*

Мы используем 100-мерные наборы данных для оценки производительности двух алгоритмов обучения без учителя: Kmeans и Bisecting Kmeans. На рисунках 8 и 9 показано время работы, чистота и C-dist двух алгоритмов неконтролируемого обучения в LOGO и трех конфигурациях Spark при изменении размера набора данных от 50 ГБ до 10 ТБ.

Как и в случае с результатами оценки алгоритмов контролируемого обучения, время работы Spark быстро увеличивается, и для Spark становится непрактично работать с наборами данных размером более 1 ТБ. Операция онлайн-выборки является дорогостоящей, если размер набора данных превышает 2 ТБ. За исключением стоимости онлайн-выборки, время работы Sample-Spark и LOGO аналогично для алгоритма Kmeans, а LOGO быстрее, чем Sample-Spark, в отношении алгоритма деления Kmeans пополам.

*Рис. 6: Время контролируемого обучения.*

Причина в том, что при одинаковом объеме данных стоимость связи LOGO ниже, чем у Sample-Spark. Качество обнаруженных кластеров мы оцениваем по показателям Чистота (чем выше, тем лучше) и C-dist (чем ниже, тем лучше). Как показано на рисунке 9, при использовании одного и того же процесса кластеризации, поскольку Spark использует весь набор данных, а Sample-Spark и Online-Sample-Spark используют образец данных, Spark достигает более высокой производительности. Качество кластеров, найденных LOGO, аналогично качеству Spark.

### *4.3 Производительность алгоритмов интеллектуального анализа часто встречающихся наборов элементов*

На рис. 10(a) показано время работы алгоритмов интеллектуального анализа часто встречающихся наборов элементов в Spark и LOGO при изменении количества транзакций. Порог поддержки установлен на уровне 0,15. Для наборов данных, содержащих менее 2,75 миллионов транзакций, мы организуем 5000 транзакций в один блок данных. Для наборов данных с более чем 2,75 миллионами транзакций один блок данных содержит 1 миллион транзакций. Время работы Spark чрезвычайно велико при обработке 2,75 миллиона транзакций. Причина в том, что алгоритм FP-Growth в библиотеке Spark [36] строит распределенное FP-дерево по всем разделам и рекурсивно извлекает часто встречающиеся наборы элементов. Он генерирует большое

количество промежуточных переменных, что требует большого объема памяти и отнимает много времени. И Online-Sample-Spark, и Sample-Spark также используют алгоритм FP-Growth в библиотеке Spark.

*Рис. 7: Точность контролируемого обучения.*

*Рис. 8: Время выполнения обучения без учителя.*

По той же причине Spark не может ответить в разумные сроки, когда количество транзакций превышает 60 миллионов. Алгоритмы FIM_I и FIM_II на LOGO способны обрабатывать 1 миллиард транзакций. Время их работы сублинейно увеличивается по мере увеличения количества транзакций. Вычислительные затраты алгоритма FIM_I немного выше, чем у алгоритма FIM_II, поскольку FIM_I содержит еще один раунд связи. На рисунках 10(b) и 10(c) показана точность и отзыв алгоритмов FIM_I и FIM_II в LOGO, принимая за основу часто встречающиеся наборы элементов, найденные Spark. Точность и полнота FIM_I колеблются в районе 98%, когда количество транзакций варьируется от 105 до 107.

Точность FIM_II ниже 90%, когда количество транзакций варьируется от 105 до 107. Причина в том, что оба алгоритма используют частые наборы элементов в выборках для аппроксимации реальных результатов. Бывает, что некоторые нечастые наборы элементов оказываются в выборках частыми. Когда набор данных небольшой, эта случайность влияет на точность результатов.

*Рис. 9: Чистота и C-dist обучения без учителя.*

*Рис. 10: Частый анализ набора элементов.*

Однако если набор данных большой, этот эффект можно игнорировать. Как показано на рисунках, точность и полнота FIM_I и FIM_II близки к 100%, когда количество транзакций превышает 108. В целом FIM_I лучше, чем FIM_II, с точки зрения точности результатов. Потому что у FIM_I есть еще один раунд связи для обмена информацией между вычислительными узлами. Принимая во внимание, что FIM_II лучше, чем FIM_I, с точки зрения времени работы. Следовательно, для частого анализа наборов элементов LOGO превосходит Spark, Online-Sample-Spark и Sample-Spark с точки зрения времени работы и масштабируемости. Более того, приблизительные результаты, полученные алгоритмами FIM_I и FIM_II, близки к истине.

## *5. Выводы*

В этой статье мы предлагаем парадигму Non-MapReduce для распределенного анализа больших данных. В качестве входных данных он принимает фрагменты случайной выборки, а не весь набор данных. Принимая этот подход, итерационные алгоритмы выполняют итерации разделов данных параллельно и независимо, что значительно снижает затраты на ввод-вывод и связь. Результатом является агрегирование результатов, полученных из разделов данных. Мы реализуем парадигму Non-MapReduce, расширяя Spark и представляя вычислительную среду LOGO. Он эффективен при вводе-выводе и обмене данными, способен выполнять итеративные аналитические алгоритмы с наборами данных терабайтного масштаба и имеет простую модель параллельного программирования. Мы также применяем структуру LOGO к часто используемым алгоритмам контролируемого обучения, неконтролируемого обучения и частым алгоритмам интеллектуального анализа наборов элементов. Результаты экспериментов демонстрируют эффективность и результативность LOGO.

### I. 绪论

大数据分析技术支持数据驱动的决策制定，已成功应用于智慧城市 [1]、欺诈检测 [2]–[4]、疾病诊断与预测 [5], [6]、工业生产 [7], [8] 和天气预报 [9] 等领域。目前，处理和分析大数据的需求正在大幅增长，根据 Statista 的报告[1]，2022 年全球产生的数据总量约为 97 ZB，预计到 2025 年将增至 181 ZB；Precedence Research[2]展示，2022 年全球数据分析市场规模为 413.9 亿美元，预计到 2030 年将超过 3463.3 亿美元。

现如今，分布式计算平台和系统 [10] 被广泛用于处理大数据。使用单台机器处理和分析大数据的成本过高，很多情况下已经变得物理上不可能。当下流行的通用分布式平台如：Hadoop[3] 和 Spark[4]，都采用了 MapReduce 计算范式。如图 1所示，大数据文件被分成较小的块存储在分布式文件系统中。分析任务通过对这些块进行 Mapper 和 Reducer 迭代操作来完成。一次完整的迭代过程包括 Mapper 和 Reducer 两个阶段，在 Mapper 和 Reducer 阶段之间还存在 shuffle 操作。该操作会导致 N-to-N 的数据传输，并产生与算法中的迭代次数成正比的通信开销 [11]。

本地分布式机器学习系统如 TensorFlow [12]、MXNeT [13]、DIANNE [14] 和 PyTorch [15] 均采用参数服务器架构。如图 2所示，每个工作节点使用本地数据计算本地参数。CPU 负责初始化、存储和更新参数，而 GPU 则将输入数据转换为张量并训练模型，然后在服务器节点上汇总和更新全局模型参数。该过程需

*为通讯作者

[1]https://www.statista.com

[2]https://www.precedenceresearch.com/data-analytics-market

[3]https://hadoop.apache.org

[4]https://spark.apache.org

要迭代进行，且每次迭代都会在工作节点与服务器节点产生通信开销。

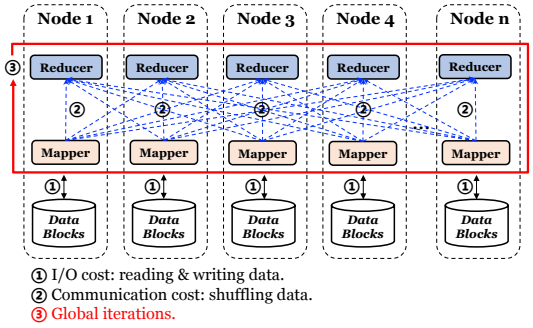



图 1: MapReduce 计算范式

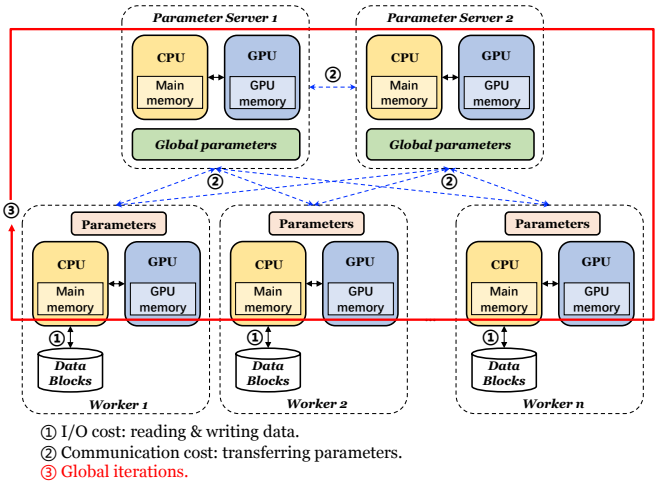



图 2: 参数服务器架构

分类 [16]、聚类 [17] 和模式挖掘 [18] 等常用的分析算法都是迭代算法，需要对整个数据集进行多次扫描和处理。因此，在执行迭代算法处理大数据时，MapReduce 计算范式和参数服务器架构存在以下限制。

- 通信开销高。在 MapReduce 计算范式和参数服务器中，执行迭代算法时的通信成本发生在连续两次迭代之间。在进入下一次迭代之前，所有机器都需要发送和接收数据和参数。因此，随着迭代次数和数据量的增加，通信成本也会大幅增长。
- 内存需求高。数据量超过系统的内存（RAM）容量会导致频繁的内存交换，即在内存和二级存储之间移动数据，该操作需要很高的时间成本。Spark 通过使用一种名为弹性分布式数据集（RDD）的分布式抽象内存数据结构，来降低数据输入和输出成本。当处理大数据时，Spark 需要大量的 RAM 进行内存计算。但是在数据量快速增长的条件下，靠增加大量的内存提高处理能力的 Spark 解决方案是不切实际的。
- 并行编程困难。在当前的分布式系统中，将分析算法转换为相应的并行算法是一项繁琐且复杂的工作。程序员必须使用现有的编程接口重写分析算法。此外，对并行程序进行调试和优化往往困难又耗时。

将大数据集划分为数据块作为大数据集的随机样本，例如随机样本划分（RSP）数据模型 [19]，便可以消除上述限制。这种数据表示方式允许每个数据块在一个计算节点中采用迭代算法进行独立处理，并允许多个数据块以大规模并行方式用同一算法进行处理。然后，各个数据块结果被传输到主节点，以计算最终结果。本文将讨论的这种 Non-MapReduce 计算范式，在大数据分析中具有以下优点。

(i) 可扩展性。在处理不断增长的数据量（如 TB 级数据）时，迭代算法能够在合理的时间内做出响应。

(ii) *I/O* 效率高。当内存不足以存储数据分区和中间结果时，内存交换并不频繁。

(iii) 通信开销低。在进行迭代时，计算节点之间的通信成本低。

(iv) 迭代算法的简单并行编程模型。复杂的迭代算法可轻松封装成算子，并可以并行执行。

本文的内容组织如下。首先，回顾了 Non-MapReduce 计算范式的核心技术——RSP 数据模型和 LOGO 计算框架，以及它在 Spark 中的实现。然后，使用监督学习、无监督学习和模式挖掘三种机器学习算法来说明 Non-MapReduce 计算范式在大型数据集上执行这些迭代算法的主要步骤。其次，提供模拟结果来展示在 10TB 级别的大数据集上运行机器学习算法时，Non-MapReduce 计算范式的高效性和数据可扩展性。最后，总结 Non-MapReduce 计算范式在大数据分析中的优势。

## II. Non-MapReduce 计算

Non-MapReduce 计算范式由两项核心技术提供支持–RSP 数据模型 [19] 和 LOGO 计算框架，前者可将分布式数据文件的数据块用作大数据文件的随机样本，后者则是在本文中为实现 Non-MapReduce 计算范式而

提出的新的计算框架。下面两节详细介绍这两项技术，后续介绍在 Spark 中的实现。

### A. RSP 数据模型

分布式数据文件是以一组数据块的形式保存并分布在集群的各个节点上，由分布式文件系统（如 HDFS [20]）管理。RSP 数据模型将大数据集表示为 RSP 数据块 [19] 的集合，这些数据块也是分布式保存在计算集群上并由 HDFS 管理。每个 RSP 数据块都是由随机样本划分算法 [21] 生成，是大数据集的随机样本，因此，可用于计算大数据集的统计估计值。

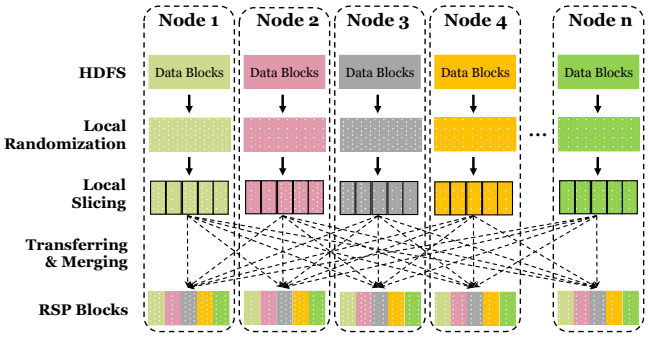



图 3: 算法 $\mathbb{T}$ 的工作流程

假设 $\mathbb{D} = \{r_1, r_2, \cdots, r_N\}$ 为一个包含 $N$ 条记录的多变量数据集，每一条数据 $r_i = (x_1, x_2, \cdots, x_M)$ 有 $M$ 个特征，每一个特征 $x_i$ 是一个随机变量。记 $\mathbb{T}$ 为一个随机样本划分算法，将 $\mathbb{D}$ 切分成 $K$ 不相交的子集 $D_1, D_2, \cdots, D_K$。这些生成的子集满足以下三个特性：

(i) $\mathbb{D} = \cup_{i=1}^{K} D_i$。

(ii) $D_i \cap D_j = \emptyset$, $1 \le i, j \le K$, $i \ne j$。

(iii) $|F(D_i) - F(\mathbb{D})| < \delta$ 对于 $1 \le i \le K$ 成立，其中 $F()$ 是数据集的累积分布函数（CDF），$\delta$ 是一个小的正数。该特性说明所有 RSP 数据块的 CDF 必然与数据集的 CDF 相似。

$D_1, D_2, \cdots, D_K$ 的集合称为 $\mathbb{D}$ 的一个 RSP 数据模型。算法 $\mathbb{T}$ [21] 的工作流程（如图 3）如下：假设数据存储在 $n$ 台机器（又称节点）上，每个节点存储有 $K$ 个数据块。首先，节点对每一个数据块中的所有记录进行随机化处理，然后切成 $K$ 个小块，这一过程在所有节点上并行执行。然后，每个节点从所有其他节点接收一个小块，将接受的小块与本地的一个小块合并重组为 RSP 块。最后，将获得的 RSP 块保存在 HDFS 中。图 3显示了算法 $\mathbb{T}$ 的工作流程。

### B. LOGO 计算框架

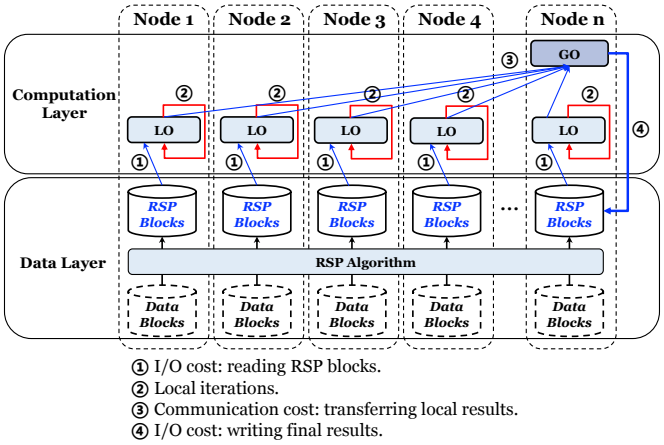



图 4: LOGO 计算框架

给定一个数据集 $\mathbb{D}$ 作为 RSP 数据模型，在 Non-MapReduce 计算范式下，$\mathbb{D}$ 的分析任务是在集群上的局部操作与全局操作（LOGO）计算框架下执行的。LOGO 计算框架由数据层和计算层组成，如图 4所示。

在数据层，数据文件以 RSP 块的形式分布式地存储在 $n$ 个节点上。通常，一个 RSP 块对应一个待分析数据文件的随机样本。在计算层，分析任务的执行过程由局部阶段和全局阶段组成，所有节点（`LO` workers）都可以参与局部阶段计算操作，而全局阶段只需要一个节点（`GO` master）负责计算操作即可。

**局部阶段。** 每个`LO` worker 在不与其他`LO` worker 通信的情况下，对节点本地的 RSP 数据块执行本地局部操作。局部操作可以是简单的算法，也可以是高度迭代的算法。在局部阶段结束时，每个`LO` worker 会生成一个基于节点本地单 RSP 数据块的局部模型。

**全局阶段。** 当所有`LO` worker 完成本地局部任务后，`GO` master 将收到通知并通过远程过程调用的方式从所有`LO` workers 上读取局部模型，然后将执行一个全局操作，将所有局部模型合并成最终的全局模型，并保存到存储器中。

不同于迭代多对 map 和 reduce 操作来执行迭代算法而存在巨大数据通信开销的 MapReduce 计算范式，LOGO 计算框架可以在每个`LO` worker 中并行地完成执

行迭代算法而无需数据通信，因此，它在执行迭代算法时效率非常高。

### C. Spark 实现

采用 RSP 数据模型和 LOGO 计算框架的 Non-MapReduce 计算范式是在 HDFS[5]和 Spark 上实现的。接下来详细介绍三个组件。

**(i) RspRDD**。Spark 使用 RDD 在计算之间共享数据。遵循这一理念，通过扩展 Spark 中的 RDD，创建了一个名为`RspRDD`的抽象对象来进行数据共享。HDFS 中的 RSP 块在 Spark 中由`RspRDD`表示，它可以并行操作，具体定义如下所示：

```
class RspRDD[T: ClassTag](prev: RDD[T])
                          extends RDD[T](prev){...}
```

**(ii) RSP 转换**。提供了将数据块中的数据随机化并切片为 $K$ 个小块的方法`RandomizeAndSlice()`。用户可以在下面的代码片段 (Scala 版本的 Spark API) 中使用它，从而将 HDFS 中的数据块转换为 RSP 数据块。第 1 行指定要转换的数据文件和 RDD 中分区的数量 $K$。第 2 行将函数`RandomizeAndSlice()`传递给 Spark API 函数`mapPartitions()`，以便在所有节点的每个数据块上执行局部随机化和切片。然后，为每个小数据块分配一个范围区间为 $[1, K]$ 的整数键。第 3 行调用 Spark API 函数`reduceByKey()`传输和合并小数据块，从而在每个节点上生成 RSP 块。最后将这些 RSP 块保存到 HDFS 中。

```
1: val RDD1 = sparkContext.textFile(filepath, K)
2: val RDD2 = RDD1.mapPartitions(RandomizeAndSlice())
3: val rspRDD = RDD2.reduceByKey()
4: rspRDD.saveAsTextFile(rspfilepath)
```

**(iii) LOGO APIs**。LOGO 计算框架中的局部操作和全局操作分别用`LO()`和`GO()`表示。函数`LO()`首先处理作为 RDD 分区的输入 RSP 块，并在每个 RSP 分区上输出一个局部模型。函数`GO()`会合并函数`LO()`生成的所有局部模型，并输出最终模型。LOGO API 支持 Java、Scala、Python 等多种编程语言，其函数的详细说明如下。

- `rspTextFile(rspfilepath)`：在 HDFS 中加载 RSP 数据文件，并返回一个`RspRDD`对象，该对象将 RSP 数据块表示为 RDD 中的一个 RSP 分区。
- `getSubPartitions(m)`：随机选择 $m$ 的 RSP 分区。
- `LO(L_Algorithm())`：接受一个用户定义的函数`L_Algorithm()`，该函数可以是任何分析算法。它将函数`L_Algorithm()` 分派给所有`LO` workers。每个`LO` worker 将函数`L_Algorithm()` 应用到自己本地的 RSP 分区，然后从每个 RSP 分区生成一个局部模型。类似地，函数`LO(preProcess())` 则指定了在每个 RSP 分区上进行数据预处理的函数。例如，在执行函数`LO(L_Algorithm())` 之前运行函数`LO(preProcess())`表示在建模之前进行数据预处理操作。
- `GO(G_Algorithm())`：它接受一个用户自定义的函数`G_Algorithm()`，该函数合并所有局部模型并生成最终模型。

**客户端程序**。下面的 Scala 代码片段展示了一个使用 LOGO 计算框架的应用示例。第 1 行为指定的 RSP 数据文件准备一个`RspRDD`对象。第 2 行获取 $m$ 个 RSP 分区。第 3 行对每个数据分区进行预处理。第 4 行在每个 RSP 分区上执行函数`LO()`，并返回 $m$ 个本地局部的模型。第 5 行执行函数`GO()`，该函数通过合并 $m$ 个局部模型产生最终的结果。第 6 行将最终模型保存在 HDFS 的指定`filepath`位置。

```
1: val rspRDD = sparkContext.rspTextFile(rspfilepath)
2: val dataRDD = rspRDD.getSubPartitions(m)
3: val trainRDD = dataRDD.LO(preProcess())
4: val modelRDD = trainRDD.LO(L_Algorithm())
5: val resultRDD = modelRDD.GO(G_Algorithm())
6: resultRDD.save(filepath)
```

从这个例子中可以看到，使用 LOGO 计算框架来编写分析任务的应用程序变得更加容易。

## III. LOGO 计算框架下的机器学习算法

机器学习算法被广泛应用于大数据分析，其中，大多数算法都是在数据集上迭代搜索最优解。Non-mapreduce 计算范式在 LOGO 计算框架支持下执行迭代机器学习算法具有优势。下面使用监督学习、无监督学习和模式挖掘三种机器学习算法来演示如何在 LOGO 计算框架下进行数据分析。

### A. 有监督学习算法

在 LOGO 计算框架中，有监督学习算法在局部阶段使用局部 RSP 块在每个计算节点上训练出一个模型，并在全局阶段将所有局部模型汇总为最终模型。

**局部阶段**。步骤如下：

[5]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

(i) 局部函数在每个LO worker 上执行，RSP 分区块分为训练集和测试集。

(ii) 调用相应的有监督学习算法，并在i部模型。

(iii) 根据测试集获得本地模型的准确度。

(iv) LO workers 将本地模型发送给GO ma

**全局阶段**。有了LO workers 生成的局部模计全局函数来合并局部模型。合并局部模多样，如加权平均法 [22]–[24], 投票法 [2法 [26], [28], [29] 等。本文采用投票法的全局函数G _ SupervisedLearning。它型，且每个模型都与准确度相关联。将确度升序排序，基于中心极限定理 [30] 5%和后 5%的局部模型，从而提高预测设 $M$ 为剩余的本地模型集，对于输入数据集 $D$ 中的每个数据记录，它通过对 $M$ 中的局部模型进行多数表决投票 [31] 的方式来生成预测标签。

### *B. 无监督学习算法*

在 LOGO 计算框架中，无监督学习算法在局部阶段时，每个计算节点使用本地的 RSP 块进行聚类。全局阶段则由GO master 对所有计算节点上的局部聚类结果进行聚合。

**局部阶段**。步骤如下：

(i) 本地函数在每个LO worker 上执行。它调用 SMILE [6]库中相应的无监督学习算法，并在本地 RSP 分区上生成一个本地模型。

(ii) 每个局部模型由一组聚类中心组成，都会发送给GO master。

**全局阶段**。给定由LO workers 生成的聚类中心集，全局函数对所有局部聚类中心进行聚类，生成新的聚类中心 $\theta$，然后将每个数据点分配到与其最近的聚类中心。

### *C. 频繁项集挖掘算法*

在 LOGO 计算框架中设计并实现了两种频繁项集挖掘算法：FIM_I和FIM_II。给定支持度阈值 $\delta$，这两种算法都使用一组 $m$ 个 RSP 分区来计算近似频繁项集。

算法FIM _ I包含三次通信。首先，每个LO worker 通过在本地 RSP 分区上执行现有的频繁项集挖掘算法

[6]https://pypi.org/project/smile

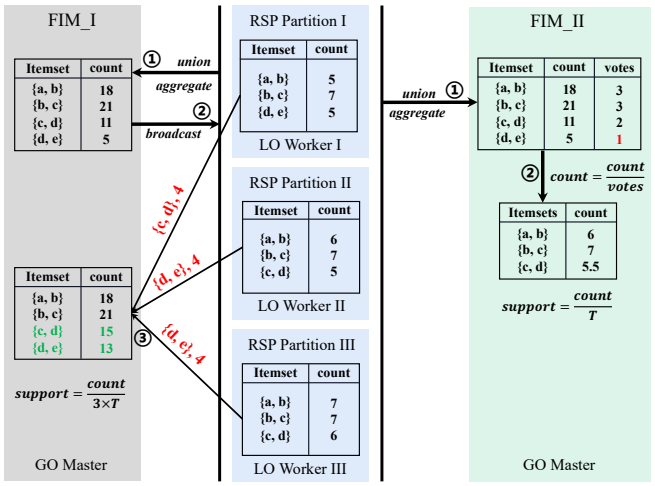



图 5: LOGO 中频繁项集挖掘算法

(如 FP-Growth 算法 [32]) 来计算本地频繁项集 $S_L$；然后，LO workers 将它们的本地结果 (项集-数量对) 发送给GO master，GO master 将收到的项集联合起来并按项集汇总计数后，向LO worker 广播联合集 $S_G$。接下来，每个LO worker 计算 $S_G \setminus S_L$ 中各项集的计数，并将结果发送回GO master。最后，GO master 更新 $S_G$ 中项集的计数，并最终确定频繁项集。每个项集的支持度按照 $count/(m \times T)$ 来计算，其中 $m$ 是 RSP 分区的数量，$T$ 是每个 RSP 分区中的交易数量。图 5左侧展示了算法FIM _ I的工作流程。在步骤 ① 中，三个LO worker 计算各自的本地频繁项集 $S_L$，再将本地结果发送给GO master。在步骤 ② 中，GO master 将联合集 $S_G = \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}\}$ 广播给LO workers。在步骤 ③ 中，每个LO worker 计算 $S_G \setminus S_L$ 中项集的计数，然后LO worker I 向GO master 发送 $(\{c, d\}, 4)$，LO worker II 发送 $(\{d, e\}, 4)$，LO worker III 发送 $(\{d, e\}, 4)$。最后，GO master 最终确定项目集的计数。给定 $m = 3$、$T = 10$ 和支持阈值 $\delta = 0.5$ 的条件下，频繁项集为 $\{a, b\}$、$\{b, c\}$ 和 $\{c, d\}$。

算法FIM _ II只通信一次。首先，与FIM _ I算法类似，所有LO worker 在自己的 RSP 分区上计算频繁项集，并将本地结果发送给GO master。除了汇总收到的本地结果外，GO master 还从LO worker 收集每个项集的投票。如果一个项集出现在一个LO worker 的本地结果中，它将会获得一票 [33]。然后在GO master 端，对于投票率 $votes/m$ 大于 $\lambda$ 的项集，其计数更新为

$count = count/votes$。接下来，再通过 $count/T$ 计算每个项目集的支持度。最后根据给定的支持阈值 $\delta$ 确定频繁项集。图 5右侧展示了算法FIM_II的工作流程，其中 $\lambda = 50\%$、$m = 3$、$T = 10$、$\delta = 0.5$。在步骤 ① 中，三个LO worker 计算各自的本地频繁项集，再将本地结果发送给GO master。然后，GO master 按项集汇总计数并收集投票。在步骤 $textcircles2$ 中，由于项集 $\{d, e\}$ 的投票比率为 1/3，小于 $\lambda = 50\%$，故将其丢弃。接下来再更新其他三个项集的计数，得到频繁项集为 $\{a, b\}$，$\{b, c\}$, and $\{c, d\}$.

## IV. 实验评估

为了展示 LOGO 计算框架的有效性和高效性，本文将其与 Spark 的三种配置进行了比较：

- `Spark`：在 Spark 上使用整个数据集运行算法。
- `Online-Sample-Spark`：首先调用 Spark 中内置的采样算子来获取样本数据（原始数据集的 5%），然后在样本数据上运行算法。
- `Sample-Spark`：与 Online-Sample-Spark 相同，但不包括采样操作的时间成本。

在包含 32 个计算节点的集群上进行实验评估。其中，24 个节点各配置 2 个 16 核，频率为 2.6GHz 的 Inter Xeon E5-2650 CPU 和 128GB 的内存；8 个节点各配置 2 个 12 核，频率为 2.6GHz 的 Inter Xeon E5-2650 CPU 和 128GB 的内存。集群上运行的是 Spark 2.4.0 和 YARN 3.0.0。本实验使用 YARN 为每个算法分配最多 100 个虚拟机和 1.6TB 内存。

### A. 有监督学习算法的性能

本文使用 100 维的数据集来评估四种有监督学习算法的性能：决策树、随机森林、逻辑回归和支持向量机。图 6 和 7 给出了这四种有监督学习算法在 LOGO 计算框架和三种 Spark 内置方案下，当数据集大小从 50GB 增加到 10TB 时的运行时间和准确率。

Spark的运行时间随着数据集大小的增加呈指数增长。当数据集大于 1.5TB 时，其计算成本很高，这是因为Spark在整个数据集上执行有监督学习算法，算法需要迭代，所以通信开销很大。而Online-Sample-Spark和Sample-Spark都是在整个数据集的样本上执行有监督学习算法，因此，它们的运行时间随着数据集大小的增加而线性增加。然

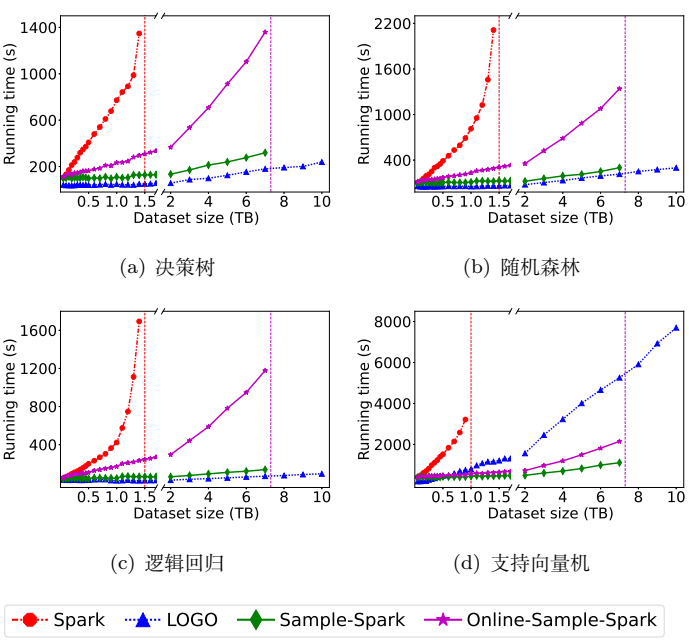



图 6: 有监督学习算法的运行时间

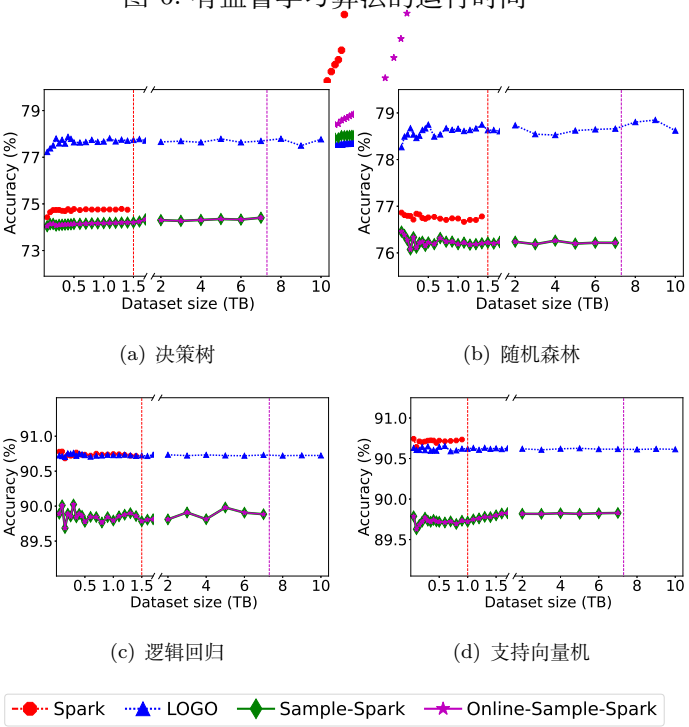



图 7: 有监督学习算法的准确度

而，当数据集大于 2TB 时，Online-Sample-Spark的运行时间迅速上升，这是因为在大数据集上进行在线采样速度很慢。除去在线采样的时间成本，可以观察到Sample-Spark和LOGO的运行时间相近。LOGO比Sample-Spark运行更快，原因是在相同的数据

量下，Sample-Spark在每次迭代结束时都有通信成本，而LOGO在局部阶段进行迭代计算，只有在局部阶段结束时才有通信成本。

除在线采样操作外，Online-Sample-Spark与Sample-Spark是一样的，得到的结果和精度相同。Spark在整个数据集上学习模型，而Online-Sample-Spark和Sample-Spark使用数据集的样本进行学习。因为他们的学习过程是一样的，所以Spark的准确率高于Online-Sample-Spark和Sample-Spark。对于决策树和随机森林算法，LOGO的准确率优于Spark，这是因为LOGO中的学习过程类似于集成学习，而集成学习已被证明是决策树和随机森林算法 [34], [35] 的有效策略。所以，即便LOGO在数据集的一个样本上学习模型，但它得到的性能比Spark更好。然而对于逻辑回归和支持向量机算法，LOGO的准确率与Spark相接近。原因可能是Spark的精度较高，在 90% 以上，因此，LOGO 改进的空间有限。

### *B. 无监督学习算法的性能*

本文使用 100 维的数据集来评估两种无监督学习算法的性能：Kmeans 和 Bisacting Kmeans。图 8和9给出了这两种无监督学习算法在 LOGO 计算框架和三种 Spark 内置方案下，当数据集大小从 50GB 增加到 10TB 时的运行时间和 Purity 以及 C-dist 度量结果。

与有监督学习算法的实验结果类似，Spark的运行时间随着数据量的增加而迅速增加。结果展示，使用Spark处理大于 1TB 的数据集是不切实际的。当数据集大于 2TB 时，在线采样操作开销较大。除去在线采样的成本，对于 Kmeans 算法，Sample-Spark和LOGO的运行时间相近；对于 Bisecting Kmeans 算法，LOGO比Sample-Spark更快，这是因为在相同数据量下LOGO的通信成本要低于Sample-Spark。

本文根据 Purity（越高越好）和 C-dist（越低越好）来评估聚类结果。如图 9所示，在相同的聚类算法下，由于Spark使用整个数据集进行学习，而Sample-Spark和Online-Sample-Spark使用数据集的一个样本进行学习，因此Spark的性能相对于他们更好，而LOGO与Spark的聚类结果相似。

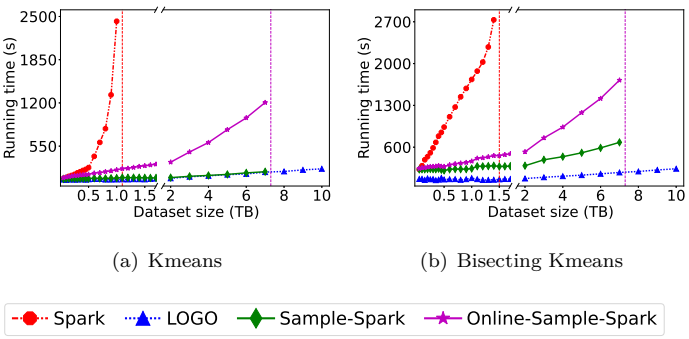



图 8: 无监督学习算法的运行时间

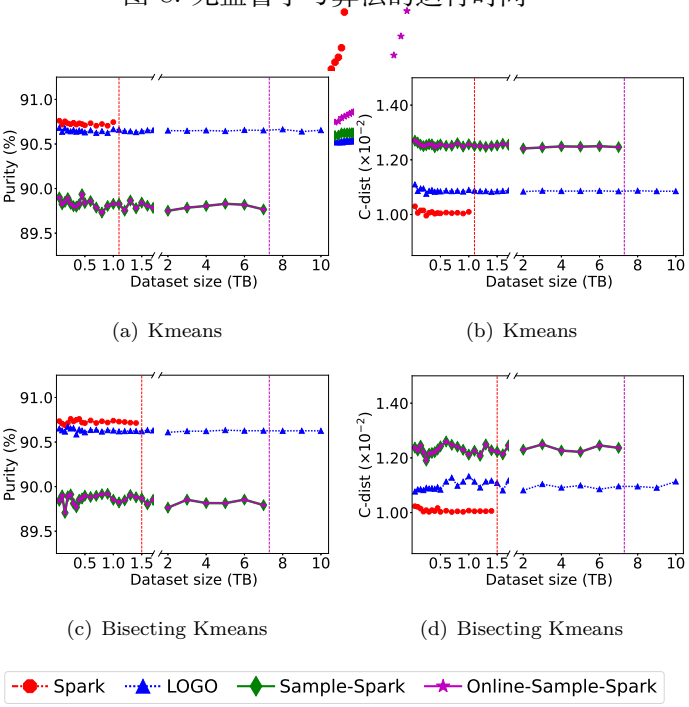



图 9: 无监督学习算法的 Purity 和 C-dist

### *C. 频繁项集挖掘算法的性能*

图 10（a）展示了频繁项集挖掘算法在 Spark 和 LOGO 计算框架上改变交易数量时的运行时间。支持度阈值设置为 0.15。对于数据条数少于 275 万的数据集，本实验将 5000 条数据组成一个数据块。对于数据条数超过 275 万的数据集，则让一个数据块包含 100 万条数据。

当处理数据条数为 275 万条数据集时，Spark的运行时间非常高，这是因为 Spark 库 [36] 中的 FP-Growth 算法在所有分区上构建一个分布式频繁模式树并递归地提取频繁项集。这会产生大量的中间变量，需要很大的内存而且时间开销巨

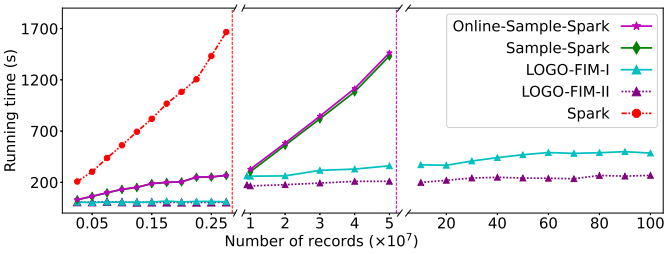
大。Online-Sample-Spark和Sample-Spark都采用了 Spark 库中的 FP-Growth 算法。和Spark的原因相类似，当处理的数据条数超过 6000 万时，他们将无法在合理的时间内做出响应。而LOGO上的算法FIM_I和FIM_II可以处理数据数量高达 10 亿条的数据集，它们的运行时间随着处理的数据数量的增加呈次线性增长。由于算法FIM_I多包含一轮通信，算法FIM_I的计算成本略高于算法FIM_II。

图 10（b）和 10（c）以Spark发现的频繁项集作为基准真值，展示了算法FIM_I和算法FIM_II在 LOGO 计算框架下的准确率和召回率。当处理的数据数量从 $10^5$ 增加到 $10^7$ 时，FIM_I的准确率和召回率在 98% 左右波动，而FIM_II的准确率低于 90%。原因是两种算法都使用样本中的频繁项集来近似真实结果。在样本中，一些不常见的项集变成了频繁项集。当数据集较小时，这种随机性会影响结果的精度。但是，当数据集较大时，这种影响可以忽略不计。如图所示，当交易数超过 $10^8$ 时，FIM_I和FIM_II的准确率和召回率接近 100%。总的来说，就结果的准确性而言，FIM_I优于FIM_II。因为FIM_I有多一轮通信来在计算节点之间共享信息。然而，在运行时间方面，FIM_II优于FIM_I。

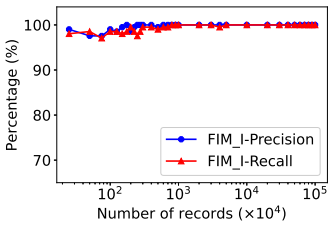
因此，对于频繁项集挖掘，LOGO在运行时间和可扩展性方面优于Spark、Online-Sample-Spark和Sample-Spark。此外，算法FIM_I和FIM_II提供的近似结果更接近于基准真值。

## V. 总结

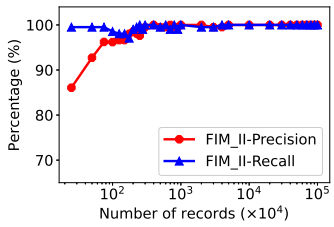
本文提出了一种用于分布式大数据分析的 Non-MapReduce 计算范式。它使用随机样本分区作为输入，而不是使用整个数据集。采用这种方法，迭代算法可以并行地、独立地对数据分区进行迭代，从而大大降低 I/O 和通信成本。输出是从数据分区获得的结果的聚合。通过扩展 Spark 实现了 Non-MapReduce 计算范式，并引入 LOGO 计算框架。它具有 I/O 和通信效率高的优势，能够在 TB 规模的数据集上执行迭代分析算法，并且具有简单的并行编程模型。将 LOGO 计算框架用于常用的有监督学习、无监督学习和频繁项集挖掘算法计算，实验结果展示了 LOGO 计算框架的有效性和高效性。




(a) 运行时间




(b) FIM-I.




(c) FIM-II.

图 10: 频繁项集挖掘

## VI. 致谢

感谢国家自然科学基金（61972261）和深圳市基础研究专项基金（JCYJ20220818100205012）对该研究的大力支持。

## Литература References 参考文献

[1] Vaia Moustaka, Athena Vakali, and Leonidas G. Anthopoulos. A systematic review for smart city data analytics. *ACM Comput. Surv.*, 51(5):103:1–103:41, 2019.

[2] Can Liu, Yuncong Gao, Li Sun, Jinghua Feng, Hao Yang, and Xiang Ao. User behavior pre-training for online fraud detection. In Aidong Zhang and Huzefa Rangwala, editors, *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, pages 3357–3365. ACM, 2022.

[3] Noussair Fikri, Mohamed Rida, Noureddine Abghour, Khalid Moussaid, and Amina El Omri. An adaptive and real-time based architecture for financial data integration. *J. Big Data*, 6:97, 2019.

[4] Shaosheng Cao, Xinxing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. Titant: Online real-time transaction fraud detection in ant financial. *Proc. VLDB Endow.*, 12(12):2082–2093, 2019.

[5] Pijush Kanti Dutta Pramanik, Saurabh Pal, and Moutan Mukhopadhyay. Healthcare big data: A comprehensive overview. *Research anthology on big data analytics, architectures, and applications*, pages 119–147, 2022.

[6] Kee Yuan Ngiam and Wei Khor. Big data and machine learning algorithms for health-care delivery. *The Lancet Oncology*, 20(5):e262–e273, 2019.

[7] Paul Suganthan G. C., Chong Sun, Krishna Gayatri K., Haojun Zhang, Frank Yang, Narasimhan Rampalli, Shishir Prasad, Esteban Arcaute, Ganesh Krishnan, Rohit Deep, Vijay Raghavendra, and AnHai Doan. Why big data industrial systems need rules and what we can do about it. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 265–276. ACM, 2015.

[8] Mingming Zhang, Tianyu Wo, Xuelian Lin, Tao Xie, and Yaxiao Liu. Carstream: An industrial system of big data processing for internet-of-vehicles. *Proc. VLDB Endow.*, 10(12):1766–1777, 2017.

[9] Jianyi You, Auwal Sagir Muhammad, Xin He, Tianqi Xie, Zhiyuan Wang, Xiaoliang Fan, Zhiyong Yu, Longbiao Chen, and Cheng Wang. PANDA: predicting road risks after natural disasters leveraging heterogeneous urban data. *CCF Trans. Pervasive Comput. Interact.*, 4(4):393–407, 2022.

[10] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *ACM Comput. Surv.*, 53(2):30:1–30:33, 2021.

[11] Min Yoon, Hyeong-Il Kim, Dong Hoon Choi, Heeseung Jo, and Jae-Woo Chang. Performance analysis of mapreduce-based distributed systems for iterative data processing applications. In James J. Park, Hojjat Adeli, Namje Park, and Isaac Woungang, editors, *Mobile, Ubiquitous, and Intelligent Computing - MUSIC 2013, FTRA 4th International Conference on Mobile, Ubiquitous, and Intelligent Computing, September 4-6, 2013, Gwangju, Korea*, volume 274 of *Lecture Notes in Electrical Engineering*, pages 293–299. Springer, 2013.

[12] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[14] Elias De Coninck, Steven Bohez, Sam Leroux, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. DIANNE: a modular framework for designing, training and deploying deep neural networks on heterogeneous distributed infrastructure. *J. Syst. Softw.*, 141:52–65, 2018.

[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[16] Raj Kumar and Rajesh Verma. Classification algorithms for data mining: A survey. *International Journal of Innovations in Engineering and Technology (IJIET)*, 1(2):7–14, 2012.

[17] Absalom E. Ezugwu, Abiodun M. Ikotun, Olaide Nathaniel Oyelade, Laith Mohammad Abualigah, Jeffrey O. Agushaka, Christopher I. Eke, and Andronicus Ayobami Akinyelu. A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects. *Eng. Appl. Artif. Intell.*, 110:104743, 2022.

[18] Chin-Hoong Chee, Jafreezal Jaafar, Izzatdin Abdul Aziz, Mohd Hilmi Hasan, and William Yeoh. Algorithms for frequent itemset mining: a literature review. *Artif. Intell. Rev.*, 52(4):2603–2621, 2019.

[19] Salman Salloum, Joshua Zhexue Huang, and Yulin He. Random sample partition: A distributed data model for big data analysis. *IEEE Trans. Ind. Informatics*, 15(11):5846–5854, 2019.

[20] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[21] Chenghao Wei, Salman Salloum, Tamer Z. Emara, Xiaoliang Zhang, Joshua Zhexue Huang, and Yu-Lin He. A two-stage data processing algorithm to generate random sample partitions for big data analysis. In *CLOUD*, volume 10967, pages 347–364, 2018.

[22] Robi Polikar. Ensemble learning. In *Ensemble machine learning*, pages 1–34. Springer, 2012.

[23] Zhice Fang, Yi Wang, Ling Peng, and Haoyuan Hong. A comparative study of heterogeneous ensemble-learning techniques for landslide susceptibility mapping. *Int. J. Geogr. Inf. Sci.*, 35(2):321–347, 2021.

[24] Antonio Galicia, Ricardo L. Talavera-Llames, Alicia Troncoso Lora, Irena Koprinska, and Francisco Martínez-Álvarez. Multi-step forecasting for big data time series based on ensemble learning. *Knowl. Based Syst.*, 163:830–841, 2019.

[25] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Mach. Learn.*, 36(1-2):105–139, 1999.

[26] Yuyan Wang, Dujuan Wang, Na Geng, Yanzhang Wang, Yunqiang Yin, and Yaochu Jin. Stacking-based ensemble learning of decision trees for interpretable prostate cancer detection. *Appl. Soft Comput.*, 77:188–204, 2019.

[27] Shadi Khalifa, Patrick Martin, and Rebecca Young. Label-aware distributed ensemble learning: A simplified distributed classifier training model for big data. *Big Data Res.*, 15:1–11, 2019.

[28] Deepak Gupta and Rinkle Rani. Improving malware detection using big data and ensemble learning. *Comput. Electr. Eng.*, 86:106729, 2020.

[29] Yue-Shan Chang, Satheesh Abimannan, Hsin-Ta Chiao, Chi-Yeh Lin, and Yo-Ping Huang. An ensemble learning based hybrid model and framework for air pollution forecasting. *Environmental Science and Pollution Research*, 27(30):38155–38168, 2020.

[30] Sang Gyu Kwak and Jong Hae Kim. Central limit theorem: the cornerstone of modern statistics. *Korean journal of anesthesiology*, 70(2):144–156, 2017.

[31] Karima Sid and Mohamed Batouche. Ensemble learning for large scale virtual screening on apache spark. In Abdelmalek Amine, Malek Mouhoub, Otmane Aït Mohamed, and Bachir Djebbar, editors, *Computational Intelligence and Its Applications - 6th IFIP TC 5 International Conference, CIIA 2018, Oran, Algeria, May 8-10, 2018, Proceedings*, volume 522 of *IFIP Advances in Information and Communication Technology*, pages 244–256. Springer, 2018.

[32] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1):53–87, 2004.

[33] Timur Valiullin, Zhexue Huang, Chenghao Wei, Jianfei Yin, Dingming Wu, and Iuliia Egorova. A new approximate method for mining frequent itemsets from big data. *Comput. Sci. Inf. Syst.*, 18(3):641–656, 2021.

[34] Bahzad Charbuty and Adnan Abdulazeez. Classification based on decision tree algorithm for machine learning. *Journal of Applied Science and Technology Trends*, 2(01):20–28, 2021.

[35] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[36] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In Pearl Pu, Derek G. Bridge, Bamshad Mobasher, and Francesco Ricci, editors, *Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys 2008, Lausanne, Switzerland, October 23-25, 2008*, pages 107–114. ACM, 2008.