


Article

An Efficient and Scalable Algorithm to Mine Functional Dependencies from Distributed Big Data

Wanqing Wu^{1,2} and Wenyu Mao^{1,2,*} ¹ College of Cyber Security and Computer, Hebei University, Baoding 071000, China; wuwanqing8888@126.com² Key Laboratory of High Trusted Information System in Hebei Province (Hebei University), Baoding 071000, China

* Correspondence: mwy070017@163.com

Abstract: A crucial step in improving data quality is to discover semantic relationships between data. Functional dependencies are rules that describe semantic relationships between data in relational databases and have been applied to improve data quality recently. However, traditional functional discovery algorithms applied to distributed data may lead to errors and the inability to scale to large-scale data. To solve the above problems, we propose a novel distributed functional dependency discovery algorithm based on Apache Spark, which can effectively discover functional dependencies in large-scale data. The basic idea is to use data redistribution to discover functional dependencies in parallel on multiple nodes. In this algorithm, we take a sampling approach to quickly remove invalid functional dependencies and propose a greedy-based task assignment strategy to balance the load. In addition, the prefix tree is used to store intermediate computation results during the validation process to avoid repeated computation of equivalence classes. Experimental results on real and synthetic datasets show that the proposed algorithm in this paper is more efficient than existing methods while ensuring accuracy.

Keywords: data mining; functional dependency; distributed computing; big data



Citation: Wu, W.; Mao, W. An Efficient and Scalable Algorithm to Mine Functional Dependencies from Distributed Big Data. *Sensors* **2022**, *22*, 3856. <https://doi.org/10.3390/s22103856>

Academic Editor: Danda B. Rawat

Received: 24 April 2022

Accepted: 17 May 2022

Published: 19 May 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

In the information age, data has become the most important asset of a company, and data-driven decisions can bring good results to every organization and company [1]. However, with the explosive growth of data volume and the variety of data sources, low quality data inevitably appears. Specifically, the collected data may contain missing, redundant, and semantic contradictions. For example, in the process of interaction with the environment, the sensor network is easily damaged under the influence of the natural environment such as sunlight and rain, resulting in equipment failure and an inability to return data or leading to the return of incorrect data [2]. Business decisions made with low-quality data can lead to huge financial losses and irreversible consequences [3,4].

Therefore, data cleaning has grown up to be a necessary prerequisite for designing and completing system engineering and has received extensive attention from many scientific researchers and related practitioners. According to statistics, in applications such as machine learning and data mining, researchers spend more than 60% of their time and energy on data preprocessing [5]. It can be seen that the theory and method of improving data quality have significant research significance and value.

Taking measures at the data source to avoid the generation of low-quality data is usually not achievable, so the main method to improve data quality is to perform error detection and repair on the dataset [6]. Many scholars have studied the process of automatic data detection and repair, including outlier detection [7–9], dependency conflict detection [10–12], and duplicate value detection [13,14]. Dependency-based methods [10–12] detect errors and repair data through semantic relations between data, which are represented

by various integrity constraints [15], such as functional dependencies [16], conditional functional dependencies [17], and denial constraints [18]. Detecting and repairing data through dependency-based methods [19,20] still requires end-user input of integrity constraints, and the system utilizes these inputs to detect units that conflict with dependencies. However, manually writing integrity constraints are inefficient and requires sufficient domain knowledge, so it is usually necessary to mine the dependencies on the dataset with the help of automatic discovery algorithms.

Functional dependency [15] is one of the most basic and important integrity constraints. In the process of using functional dependencies to improve data quality, the primary problem is how to efficiently and automatically discover functional dependencies from table data. The study in [21] shows that the existing seven important functional dependency discovery algorithms are only suitable for small-scale centralized data sets, and cannot be extended to table data with hundreds of columns or millions of rows, and in the case of data distributed storage, these algorithms can lead to erroneous results. Therefore, with the advent of the era of big data, the amount of data has increased dramatically, and the wide application of distributed databases has brought new problems and challenges to functional dependency discovery.

Example 1. Given a relation R , as shown in Figure 1a, R is horizontally divided into two parts R_1 and R_2 and distributed on different nodes, as shown in Figure 1b. According to Figure 1b, for any two tuples t_i and t_j on R_1 or R_2 , $t_i[A] = t_j[A]$, then $t_i[B] = t_j[B]$, the functional dependency $A \rightarrow B$ can be obtained holds on either R_1 or R_2 .

ID	A	B	C
1	1	1	2
2	2	2	3
3	1	2	1
4	1	2	2
5	2	1	3

R

(a)

ID	A	B	C
1	1	1	2
2	2	2	3

R_1

ID	A	B	C
3	1	2	1
4	1	2	2
5	2	1	3

R_2

(b)

Figure 1. (a) Relation R ; (b) The horizontal segmentation R_1 and R_2 of R .

However, functional dependencies that pass local validation do not necessarily hold on global data. According to Figure 1a, it can be concluded that the functional dependence $A \rightarrow B$ does not hold on R . Therefore, the existing centralized functional dependency discovery algorithms cannot be directly applied to the distributed environment.

In distributed scenarios, functional dependency discovery for large-scale data has gradually become a research hotspot. In the distributed functional dependency discovery algorithm, FDcent_discover [22] presents a distributed database functional dependency discovery framework. Firstly, functional dependency discovery is performed at each node, and then the data of each node is sent to the master node, where the centralized discovery algorithm is used to discover. The HFDD [23] and FDPPar_Discover [24] algorithms adopt the data redistribution scheme to group candidate functional dependencies using the left-hand features of the functional dependencies, and send the tuples with the same common attribute values to the same node, the functional dependency discovery algorithm is performed in parallel at each node. However, there are still the following problems: First, the characteristics of the data set itself are not considered, which leads to the verification of many invalid function dependencies and increases the computational cost. Second, in dis-

tributed scenarios, when the distribution of attribute values is uneven, the load unbalance is expected to result in a waste of computing resources. Third, the repeated computation of equivalence classes in the process of verifying candidate functional dependencies leads to inefficiency.

The contributions of this paper are as follows:

1. A spark-based distributed functional dependency discovery algorithm is proposed.
2. Aiming at the unbalanced load caused by the uneven distribution of attribute values, the greedy-based task assignment strategy is proposed to balance the computing tasks of each node and avoid the unbalanced load causing too long computing time.
3. A dynamic memory management strategy is proposed to store calculated equivalence classes in memory and periodically clear equivalence classes that have not been accessed for a long time to maximize the use of memory space.
4. Verify the distributed functional dependency discovery algorithm proposed in this paper on real and artificial data sets through experiments.

This paper is organized as follows. In Section 2, the definitions and related work are introduced. Section 3 presents the algorithm structure and implementation process of the algorithm DisTFD. Section 4 presents the experimental results and the comparison of existing methods. Section 5 is the conclusion and outlook for future work.

2. Preliminaries

2.1. Definition

This section introduces definitions related to FD discovery. Let R be a relational schema and r be an instance on R . $t[X]$ represents the projection of a tuple t in R onto the subset $X \subseteq R$.

Definition 1. *Functional dependency.* A functional dependency $X \rightarrow A$ specifies that the value of X functionally determines the value of A , where $X \subseteq R$ and $A \in R$. If all tuple pairs $t_1, t_2 \in r$ in R satisfy $t_1[X] = t_2[X]$, then $t_1[A] = t_2[A]$, then the functional dependency $X \rightarrow A$ on the instance r of R is established. Let X be the left part (LHS) of the FD and A be the right part (RHS) of the FD.

Definition 2. *Non-trivial functional dependency.* If a functional dependency $X \rightarrow A$ holds and $A \notin X$, then $X \rightarrow A$ is said to be a non-trivial functional dependency.

Definition 3. *Minimum functional dependency.* If a functional dependency $X \rightarrow A$ holds and any proper subset \bar{X} of X cannot determine the value of attribute A , that is, for any $\bar{X} \in X$, $\bar{X} \in X \in A$ does not hold, then we call $X \rightarrow A$ the minimum functional dependency.

Definition 4. *Equivalence class.* The equivalence class of a tuple $t \in r$ is expressed as $[t]_X = \{u \in r | \forall A \in X, t[A] = u[A]\}$. Taking the relation R in Example 1 as an example, an equivalence class of the tuple t_1 on the attribute C is $\{1, 4\}$.

Definition 5. *Partition.* Divide all tuples in r into multiple equivalence classes based on the attribute set $X \in R$. The partition $\Pi_X = \{[t]_X | t \in r\}$ of relation r on attribute set X is the set of all equivalence classes, and $|\Pi_X|$ represents the number of equivalence classes in Π_X . In Example 1, the relation R can be divided into multiple equivalence classes on the attribute set $\{C\}$: $\Pi_C = \{\{1, 4\}, \{2, 5\}, \{3\}\}$, $|\Pi_C| = 3$.

Definition 6. *Stripped partition.* The stripped partition $\hat{\Pi}_X$ of relation r on attribute set X refers to the partition obtained by removing all equivalence classes with 1 element on the basis of Π_X . In Example 1, the relation R is divided into $\Pi_C = \{\{1, 4\}, \{2, 5\}, \{3\}\}$ based on the attribute set $\{C\}$, then its stripped partition $\hat{\Pi}_C = \{\{1, 4\}, \{2, 5\}\}$.

2.2. Related Work

Functional dependency discovery. Existing functional dependency discovery algorithms are mainly used in centralized environments and can be divided into three categories: lattice search algorithms, difference and consensus set algorithms, and hybrid algorithms.

Lattice search algorithm: the typical representatives are TANE [25], FUN [26], and FD_Mine [27] algorithms. The search space is modeled as the lattice of attribute combinations to represent all candidate functional dependencies, and a bottom-up search strategy is adopted to verify the candidate functional dependencies at each layer. The time complexity of the lattice search algorithm mainly depends on the size of the lattice, and the size of the lattice depends on the number of attributes of the dataset. Therefore, the lattice search algorithm has better row scalability and is suitable for large-scale datasets with fewer columns.

Difference set and consistent set algorithm: the typical representatives are Dep-Miner [28] and FastFDs [29] algorithms. Based on the comparison between tuples, the consistent set and the difference set are obtained, and finally the candidate functional dependency is verified according to the difference set. The time complexity of difference and consistent set algorithms depends on the number of tuples. Therefore, the difference set and consistent set algorithms have better column scalability and are suitable for small-scale datasets with many columns.

Hybrid Algorithm: HyFD [30] uses a hybrid discovery strategy to combine the advantages of the lattice search algorithm and the difference set and consistent set algorithms, and has better scalability in rows and columns. HyFD first generates a consensus set from the sampled data, identifies candidate functional dependencies from the consensus set, and uses FDTree to represent the corresponding attribute set. Then, HyFD is transformed into the lattice search algorithm, and candidate functional dependencies are verified by traversing the FDTree.

Approximate functional dependency discovery. In 1992, Kivinen and Mannila [31] first proposed an error metric for approximate functional dependencies. Subsequently, CORDS [32] automatically discovered unary approximate functional dependencies from relational data. To further speed up the discovery of approximate functional dependencies, the authors of [33] used heuristics to prune the candidate space of approximate functional dependencies. Mandros and Boley [34] represented the approximation of functional dependencies more precisely by scores.

The authors of [35] use a machine learning approach to infer approximate functional dependencies by comparing tuples with each other. The method finds all conflicting functional dependencies by tuple pair comparison, applies an error threshold to remove infrequent conflicting tuple pairs, and finally, inferring approximate functional dependencies from the remaining conflicting tuple pairs.

In recent work, Caruccio and Deufemia [36] proposed a new candidate approximate functional dependency verification method to discover multiple types of approximate functional dependencies by constructing a difference matrix of attributes. AFDDPar [37] proposed a parallel approach in a distributed environment for discovering approximate functional dependencies in a distributed environment, balancing the load of individual nodes before data redistribution, and pruning candidate approximate functional dependencies quickly after data redistribution.

3. The Distributed Algorithm for Mining Functional Dependency

In this chapter, a description of the distributed functional dependency discovery problem and a general overview of the algorithm DisTFD are given. In this paper, functional dependency discovery is carried out in a distributed big data environment, a distributed processing method is designed, and intermediate results are reasonably stored. On the premise of ensuring the correct rate, the load of each computing node is balanced as much as possible to reduce the time consumption of the algorithm.

3.1. Algorithm Architecture Overview

The algorithm DisTFD consists of multiple components, which are divided into different logical modules. The framework of the algorithm DisTFD is shown in Figure 2.

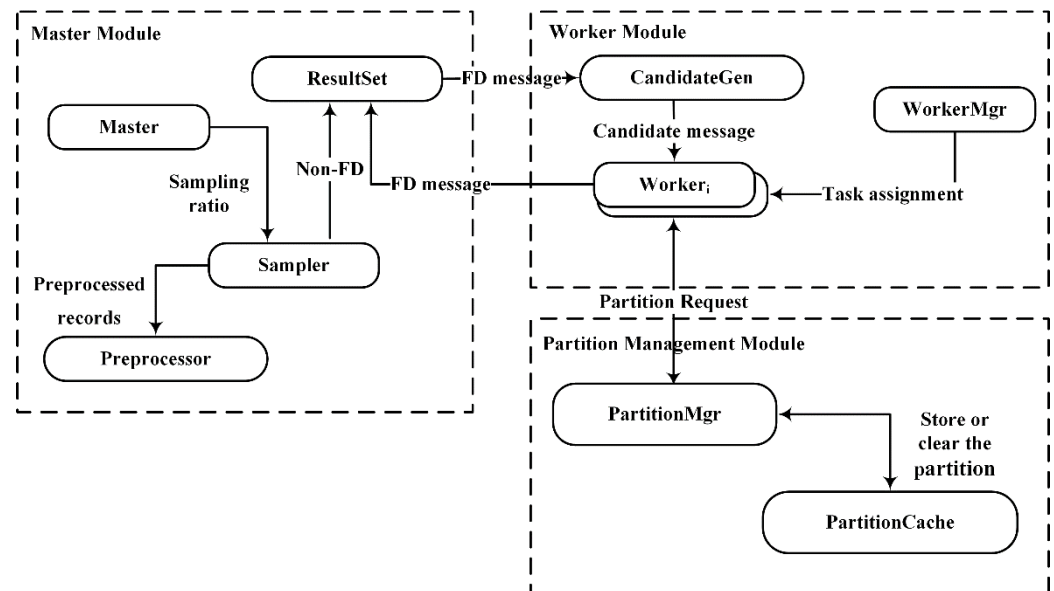


Figure 2. DisTFD logical structure diagram.

The components of DisTFD can be divided into three modules: Master Module, WorkerModule, and Partition Management Module.

The Master Module mainly performs data input, output, control sampling ratio, and data preprocessing. The master module can only be located on the master node. The worker module has several work nodes, which are mainly responsible for data storage and generation and verification of candidate function dependencies, and send the verification results to the master node. The partition management module merges the results calculated by multiple worker nodes, and stores the partition used for verifying candidate functional dependencies.

The specific functions of the components in the three modules are as follows:

ResultSet. The ResultSet component stores the invalid functional dependencies and the valid functional dependencies as two sets, respectively.

CandidateGen. The CandidateGen component generates candidate functional dependencies in the form of the lattice and sends the candidate functional dependencies to each worker node. After each validation, CandidateGen prunes candidate function dependencies according to the validation results in the ResultSet.

Sampler. The Sampler component samples the data according to the ratio set in the master node and is responsible for verifying the received candidate FD on the sampling data set D' . If the verification result is true, the candidate function will be sent to the work nodes for further verification. If the verification result is false, the candidate functional dependency will be sent to the ResultSet.

PartitionMgr. The PartitionMgr component accepts the request for partition by the work node, and if there is a partition of the request in the PartitionCache, it will be sent to the corresponding work node. If the requested partition does not exist in PartitionCache, Worker will calculate the partition and merge calculation results by PartitionMgr. Then, PartitionMgr stores calculated partitions in the PartitionCache, and periodically clears the partitions that have not been accessed for a long time.

Worker. The Worker component verifies the candidate functional dependencies, and sends the result to the ResultSet and requests a new verification job from CandidateGen.

WorkerMgr. The WorkerMgr component records the load of each node after data redistribution. When the node load is unbalanced, the task assignment algorithm is called to assign the task to achieve load balance.

This paper proposes a distributed functional dependency discovery algorithm DisTFD based on attribute space traversal as shown in Algorithm 1:

Algorithm 1. Distributed Functional Dependency Discovery Algorithm DisTFD

Input: dataset $D = (D_1, \dots, D_{n-1}, D_n)$, attribute set X

Output: Minimum non-trivial functional dependency set Σ

/* Set the sampling ratio n data preprocessing output sorted attribute set */

```

1.   $D' = \text{Sampler}(D, n)$ 
   /* Data preprocessing output sorted attribute set */
2.  SortedAttribute = Pre_processing( $D', X$ )
   /* Generate candidate function dependencies */
3.  Candidate_FD = CandidateGen( $X$ )
4.  SamplingValidate( $\varphi, D'$ )
5.  for each  $A_i \in \text{SortedAttribute}$  do
6.      ReDistributedDataSet( $D, A_i$ )
7.      if ( $A_i \in \text{SkewAttribute}$ ) {
8.          Assignment ( $D$ )
9.      }
   /* task assignment to balances the load of worker nodes */
   /* Verify that each function in the candidate space depends on  $\varphi \in \text{Candidate\_FD}$  */
10.  if (GlobalValidate( $\varphi, D$ ) == true) {
11.      Pruning(Candidate_FD,  $\varphi$ )
12.       $\Sigma = \Sigma \cup \varphi$ 
13.  }
14.  end for
15.  Return  $\Sigma$ 

```

3.2. Data Preprocessing

The preprocessor preprocesses the data, including statistical attribute cardinality and attribute value frequency. In the case of a large amount of data and distributed storage, it is necessary to summarize the results for all data statistics multiple times, which make the cost extremely high. Therefore, this article counts attribute-related information on the sampling data set and will introduce the sampling method in Section 3.3.

The number of types of attribute values is called the cardinality of the attribute, and the number of tuples corresponding to each attribute value is called the frequency of the attribute value. Based on the statistics of the cardinality and frequency information, the skewness of each attribute is then calculated. Given an attribute A , let c be the cardinality of attribute A , V be the set of all values of attribute A , $\text{frequency}(V_k)$ represents the frequency of the k -th value of attribute A , then the skewness of attribute A is expressed as:

$$\text{Inc}(A) = \max_{1 \leq k \leq c} \frac{\text{frequency}(V_k)}{n} \quad (1)$$

where, $k \in [1, c]$, n is the total number of tuples in the dataset. The data preprocessing process is shown in Algorithm 2.

Algorithm 2. Pre_processing**Input:** sample data set D' , attribute set X **Output:** Sorted attribute set SortedAttribute

```

1.  Set the Skew threshold  $t$ 
2.  For  $A_i \in X$  do
3.     $Inc(A_i) = \max_{1 \leq k \leq c} \frac{frequency(V_k)}{n}$ 
4.    If  $(Inc(A_i) > t)$  {  $SkewAttribute \leftarrow A_i$  }
5.    else {  $NonSkewAttribute \leftarrow A_i$  }
6.  end for
7.   $SortByCardinality(NonSkewAttribute)$ 
8.   $SortedAttribute \leftarrow NonSkewAttribute \cup SkewAttribute$ 
9.  Return  $SortedAttribute$ 

```

After calculating the skewness of each attribute, the attributes are divided into Skew attribute and non-Skew attribute according to the given threshold. Then, sort all the attributes, and specify that the Skew attribute is ranked after the non-Skew attribute.

3.3. Sampling Validation Framework

Sampling refers to taking a part of the population of the research objects for investigation or statistics according to a certain procedure, so as to make inferences about the population of the research objects. In this paper, the statistical attribute information of the sampling data set reflects the situation of the attribute in the overall data set.

Sampler uses systematic sampling [38] to sample population data. According to the preset sample size n , determine an integer k closest to N/n , randomly select an integer r in the range of $[1, k]$ as the starting unit of the sample, and then select a unit every k as a sample unit until n samples are drawn.

The size of the sampled data set D' is much smaller than the overall data set D and is only stored on the master node. Therefore, the cost of functional dependency discovery on the sampled data set D' is small. The functional dependencies found in D and D' have the following two properties:

1. Completeness: A functional dependency φ that holds on D also holds on D' .
2. Minimality: The minimum functional dependence φ that holds on D' , if the functional dependence holds on D , then the functional dependence φ is also the smallest functional dependence on D .

According to the above two properties, the invalid or non-minimum functional dependencies can be quickly verified in the sampled data set, saving the time of distributed verification and improving the efficiency of the algorithm.

3.4. Search and Prune

The row-efficient functional dependence discovery algorithm is appropriate for large-scale data sets with many tuples. Therefore, this paper uses the lattice of TANE, FUN and other algorithms to generate candidate functional dependence search space. Given a relational schema $R = \{A, B, C, D\}$, all of its candidate functional dependencies are shown in Figure 3.

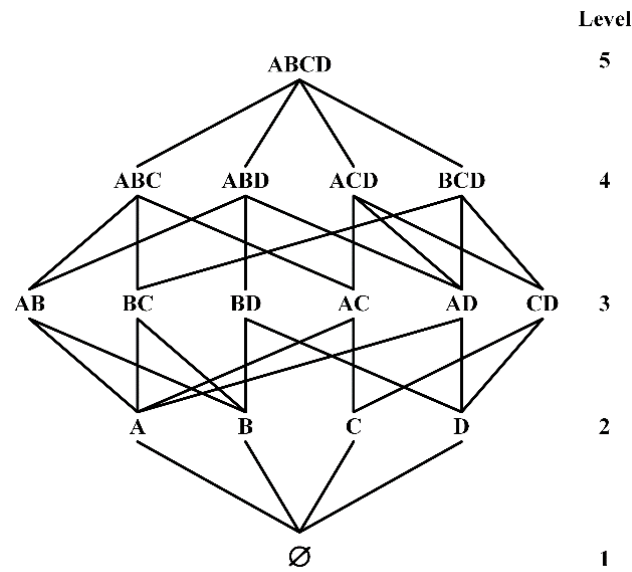


Figure 3. Candidate FDs composed of attribute sets A, B, C, D.

The LHS of the candidate FDs is all possible attribute combinations in R , the connection between the first node of Level-5 and the first node of Level-4 represents the candidate function dependency $ABC \rightarrow D$, the connection between the first node at level 3 and the first node at level 2 represents the candidate function dependency $A \rightarrow B$, and so on.

Lemma 1. Given the attribute set $\{A_1, \dots, A_n\}$ defined on the relational schema R , then the number of all non-trivial minimum functional dependencies is $n * 2^{n-1} - n$.

Proof of Lemma 1: Consider only the nontrivial minimal functional dependencies for which RHS has one property. For all candidate functional dependencies on the relation R , the number of attributes of the LHS takes the value $[1, n-1]$. The number of candidate functional dependencies of LHS with only one attribute is $C_n^1 * C_{n-1}^1$, the number of candidate functional dependencies of LHS with two attributes is $C_n^2 * C_{n-2}^1$, and the number of candidate functional dependencies of LHS with three attributes is $C_n^3 * C_{n-3}^1, \dots$, and the number of candidate functional dependencies of LHS with $n-1$ attributes is $C_n^{n-1} * C_1^1$. Therefore, the total number of non-trivial minimum functional dependencies for which RHS has a property is:

$$\begin{aligned}
 & C_n^1 * C_{n-1}^1 + C_n^2 * C_{n-2}^1 + C_n^3 * C_{n-3}^1 + \dots + C_n^{n-1} * C_1^1 \\
 &= 1 * C_n^1 + 2 * C_n^2 + 3 * C_n^3 + \dots + (n-1) * C_n^{n-1} \\
 &= 0 * C_n^0 + 1 * C_n^1 + 2 * C_n^2 + \dots + (n-1) * C_n^{n-1} \\
 &+ n * C_n^n - n * C_n^n = \sum_{j=0}^n j * C_n^j - n * C_n^n \\
 &= n * 2^{n-1} - n
 \end{aligned}$$

□

When verifying candidate functional dependencies, most existing lattice searches verify candidate functional dependencies one by one in a bottom-up or top-down order and the set of candidate functional dependencies is pruned using the following lemma:

Lemma 2. Let X, Y, Z be the three attribute sets of the relation R . If $Y \subset X$ and $X \twoheadrightarrow Z$, then $Y \twoheadrightarrow Z$.

Lemma 3. Let X, Y, Z be the three attribute sets of the relation R . If $Y \subset X$ and $Y \rightarrow Z$ hold, then $X \rightarrow Z$ holds.

According to Lemma 2, the top-down search strategy can be used to prune functional dependencies that do not hold in lower levels. For example, it has been verified that functional dependencies $ABC \rightarrow D$ do not hold, then $AB \rightarrow D$ and $AC \rightarrow D$ do not hold. Therefore, if most of the functional dependencies at the upper level are valid and those at the lower level are not, then the top-down strategy will verify more useless functional dependencies and reduce the verification efficiency.

According to Lemma 3, the bottom-up search strategy can be used to prune the functional dependencies at higher levels. For example, it has been verified that the functional dependencies $AB \rightarrow D$ holds, then $ABC \rightarrow D$ must hold, and bottom-up search strategy can avoid the verification of non-minimal functional dependencies. However, when there are many lower levels functional dependencies that do not hold, the search space cannot be effectively pruned.

In this paper, we adopt the validation method in [39] and use a two-way alternating search validation strategy in the sampling validation process. The validation is alternated from both ends of the search space. It is assumed that there are n levels of candidate functional dependencies. DisTFD verify the Level- i ($i \leq n/2$) firstly, if the verification result is true, Lemma 2 is used to prune the functional dependencies greater than the Level- i . Then, verify the Level- j ($j = n + 1 - i$), Lemma 3 is used to prune the functional dependencies smaller than the Level- j if the verification result is false, and then verify the Level- $(i+1)$, and so on until all candidate functional dependencies are verified. For example, in the 4-attribute search space shown in Figure 3, the verification order is Level-2: $\emptyset \rightarrow A$; Level-4: $ABC \rightarrow D$; Level-2: $\emptyset \rightarrow B$; ...; Level-3: $CD \rightarrow B$.

3.5. Global Validation

Candidate function dependencies verified by sampling are further verified using data redistribution.

3.5.1. Partition Caching

Calculating the number of equivalence classes in a partition to verify candidate functional dependencies. For example, verifying $X \rightarrow Y$ requires comparing $|\Pi_X| = |\Pi_{XY}|$ for equality.

Theorem 1. A functional dependency $X \rightarrow Y$ hold if and only if $|\Pi_X| = |\Pi_{XY}|$.

Proof of Theorem 1: Since $|\Pi_X| = |\Pi_{XY}|$ by definition 4 and definition 5, the number of equivalence classes in X is equal to the number of equivalence class in XY so the total number of tuples contained in the X and XY equivalence classes is equal. That is, for any tuple t_i , if t_i is in an equivalence class of X , then t_i is also in the same equivalence class of XY , and $t_i[X] = t_j[X]$ is satisfied for two tuples if t_i and t_j in the same equivalence class $|\Pi_X|$, then $t_i[Y] = t_j[Y]$, in line with the definition of functional dependency, it can be concluded that $X \rightarrow Y$ is hold. \square

The partition Π_{XY} can be derived from $\Pi_X \cap \Pi_Y$, a process called computing the intersection of partition. As shown in Figure 1, $\Pi_A = \{\{1, 3, 4\}, \{2, 5\}\}$, $\Pi_C = \{\{1, 4\}, \{2, 5\}\}$, the process of calculating Π_{AC} is as follows: First, Π_C is converted into the attribute vector $v_c = (1, 2, 0, 1, 2)$, the value that appears only once is coded as 0, and the other values are coded as 1, 2, ..., n in sequence. Then, group the equivalence classes in Π_A according to the value other than 0 in v_c , $\Pi_A = \{\{1, 3, 4\}, \{2, 5\}\}$ can be divided into $1 \rightarrow \{1, 4\}$ and $2 \rightarrow \{2, 5\}$. Finally, among all the obtained groups, groups with size greater than 1 form a new partition, $\Pi_{AC} = \{\{1, 4\}, \{2, 5\}\}$. The computational complexity of this process is high and a large amount of intermediate data will be generated during the calculation process,

resulting in a long calculation time. Therefore, this paper stores the intermediate results in the partition cache to avoid repeated calculations in the verification process.

DisTFD stores the calculated partition in the prefix tree [40] shown in Figure 4 for easy query. Each node stores the partition corresponding to the path, and the number on the node indicates the size of the partition. In the above example, to calculate Π_{AC} , the attribute set $\{A, C\}$ is converted into an attribute list (A, C) according to the attribute order in the relational schema R , and then (A, C) is used as a keyword to query in the prefix tree.

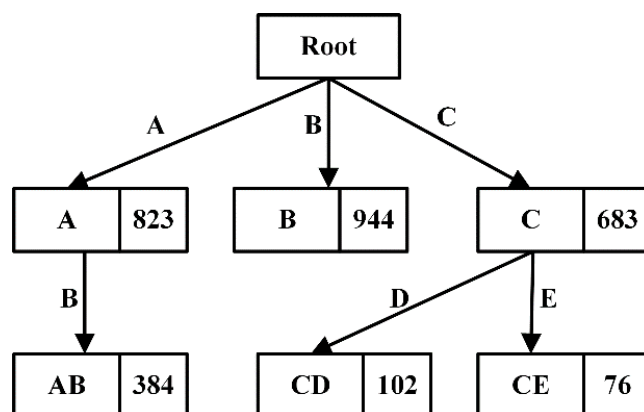


Figure 4. Prefix tree cache partition.

When using the partition in the cache, the following two rules should be followed:

1. When calculating the results, the number of partition intersections should be as few as possible.
2. In each calculation of $\Pi_X \cap \Pi_Y$, the $|\Pi_X|$ and $|\Pi_Y|$ should be minimized as much as possible.

Algorithm 3 gives the execution process of using partition to cache the calculation results under the two rules above.

Algorithm 3. RetrievePartition

Input: Partition cache Cache, attribute set X

Output: Partition Π_C

1. $\Phi \leftarrow$ Query the partition of all X subsets in the Cache
 2. $\Pi_Y \leftarrow$ Find the smallest stripped partitioning in Φ
 3. $L \leftarrow$ new list $C \leftarrow Y$
 4. **For** $C \subset X$ **do**
 5. $\Pi_Z \leftarrow$ Find the partition in Φ that satisfies $|Z| = \max\{|X/C|\}$
 6. $L = L.append(\Pi_Z)$ $C \leftarrow C \cup Z$
 7. **End for**
 8. Sort the partition in L in ascending order
 9. $C \leftarrow Y$
 10. **For** $\Pi_Z \in L$ **do**
 11. $\Pi_{C \cup Z} = \Pi_C \cap \Pi_Z$ $C \leftarrow C \cup Z$
 12. **Return** Π_C
-

In Algorithm 3, the partitions of all subsets of attribute set X are first queried in the cache and stored in the query result Φ , the smallest number of partitions is found as the starting unit of the partition intersection calculation. Next, according to rule 1, select the partition with the most newly added attributes in Φ to calculate the intersection, until all attributes in X appear at least once in the selected partition. Finally, the order of partition intersection calculation is determined according to Rule 2, the partition with a small number of equivalence classes should perform intersection calculation as soon as possible.

For example, assuming that Π_{ABCDE} is currently calculated using the partition caching shown in Figure 4, $\Phi = \{\Pi_A, \Pi_B, \Pi_C, \Pi_D, \Pi_E, \Pi_{AB}, \Pi_{AD}, \Pi_{CE}\}$ is searched in the prefix tree, and the smallest Π_{CE} is selected as the starting unit. Then, select Π_{AB} with the most newly added attributes, and then select Π_D . After the partition selection of the intersection calculation is completed, the order of intersection calculation is determined from small to large, and finally $\Pi_{ABCDE} = \Pi_{CE} \cap \Pi_{AB} \cap \Pi_D$ can be obtained.

When caching partitions, memory resources are usually limited. If all partitions are cached, excessive memory space may be occupied. Most partitions are only used for a period of time, DisTFD save the memory space by clearing partitions that are no longer used [41]. Each time the partition cache is returned, PartitionMgr records the access time of each partition and periodically clears the recently unused partitions.

3.5.2. Task Assignment and Validation

Select the sorted attributes in turn as public attributes for data redistribution. In the process of data redistribution, the tuples with the same value on the common attribute are sent to the same node by calculating the hash value of the common attribute value. When the non-Skew attribute is used as the public attribute, it is directly verified after data redistribution, and when the skewed attribute is selected as the public attribute, DisTFD assignment the task based on the greedy strategy to achieve load balancing [42].

Each attribute value of the public attribute is represented by $key_i (1 \leq i \leq m)$, and the process of the task assignment shown in Figure 5 is as follows:

1. Sort keys from small to large according to the frequency of each attribute value counted in data preprocessing.
2. Add up all key frequencies to calculate mean Avg relative to the number of nodes.
3. Traverse the key, if the key frequency is greater than Avg, split it and assign it to a node with a load of 0, record the corresponding relationship between the key and the node allocation, and subtract the Avg from the frequency of the key. Repeat this step until the frequency of the key is less than Avg. If the frequency of the key is not 0, the key is re-inserted into the queue.
4. Repeat step 3 until all keys with a frequency greater than avg are processed.
5. Select the remaining nodes that are not involved in step 3, traverse the key queue and find the sum of the node load and key frequency, if Sum is less than Avg, assign the key to the current node, and Sum is used as the load of the current node, then delete the information of the key in the queue. Repeat the above steps until all keys in the queue are processed.
6. Repeat step 5 to balance the load of the remaining nodes, and record the correspondence between keys and node assignments.

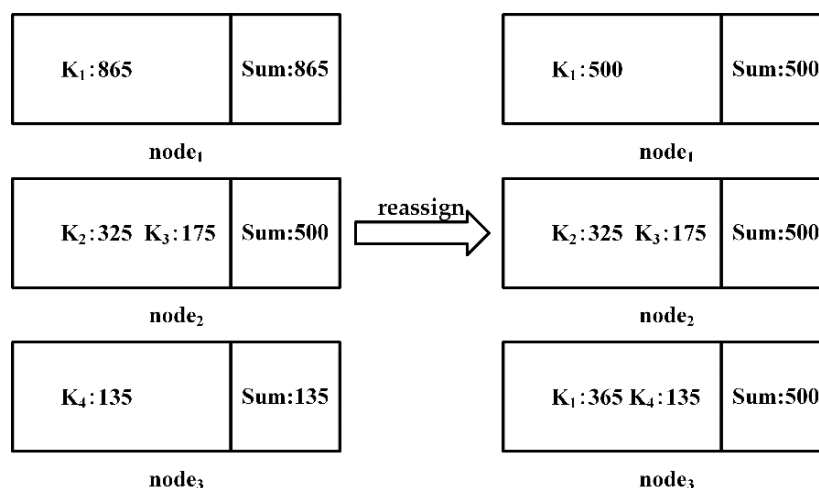


Figure 5. Task assignment to achieve load balancing.

Algorithm 4 describes the process of Task assignment. Lines 1–3 calculate the sum of the key frequencies and calculate the average load Avg on m nodes. Lines 4–11 split the partitions with a load greater than the average, and record the assignment relationship between keys and nodes. Lines 12–20 traverse the Key queue, merge the partitions with a load less than the average, and record the relationship between keys and node assignment.

Algorithm 4. Assignment

Input: dataset $D = (D_1, \dots, D_{n-1}, D_n)$

Output: true

```

1.  Read the frequency of Key in preprocessing and record it to Kfreq
2.  Klist = SortByFreq (Kfreq)
3.  Avg = Sum/m
   /* Split Keys with greater than average frequency */
4.  for each Key  $\in$  Klist
5.    if (Key.size > Avg)
6.      For Key.size > Avg do
7.        /*Key is assigned to the node with a load of 0*/
8.        Node.add(Key, i) Key.size -= Avg
9.      end for
   /* Re-insert the split Key into the queue */
10.   Klist.Sort (Key)
11.   Else break
12. end for
13. for each node do
14.   for each Key  $\in$  Klist
15.     If (Avg  $\geq$  node.size + Key.size)
16.       node.add (Key, i) node.size+ = Key.size
17.       Klist.remove (Key)
18.     end for
19.   end for
20. Return true

```

After the load balance is achieved, the local equivalence classes are obtained by computing the partitions in parallel at each node, and the local equivalence classes with the same value are merged. Finally, the partition of the candidate function dependent on LHS and LHSURHS is obtained. The process of merging local equivalence classes is shown in Figure 6.

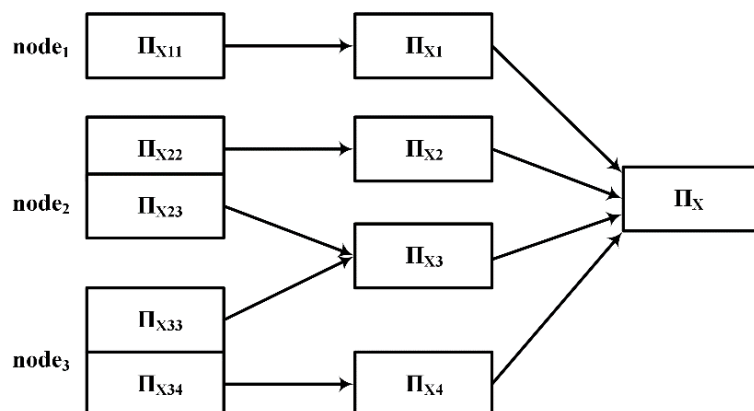


Figure 6. The process of merging the local equivalence classes of each node.

Let A be a set of common attributes of candidate function dependencies, $d = (d_1, \dots, d_{m-1}, d_m)$ is the data after redistributing the attribute value of A , then

$$|\Pi_X| = \sum_{j=1}^m |\Pi_{X_j}| \quad (2)$$

where, m is the number of keys, $j \in [1, m]$.

Algorithm 5 shows the process of parallel verification of selected function dependencies at each node.

Algorithm 5. GlobalValidate

Input: dataset $D = (D_1, \dots, D_n)$, candidate function dependency $\varphi : X \rightarrow Y$

Output: true or false

```

1.  $\Pi_Z = \text{RetrievePartition}(X, \text{Cache})$ 
2. /* Each node computes  $\Pi_{X/Z}$  in parallel */
3.  $\text{Compute}(d, X/Z)$ 
4. for each  $\Pi_{X/Zij}$  in  $\text{node}_i$ 
5.   If  $(|\Pi_{X/Zij}| \neq |\Pi_{X/Z \cup Yij}|)$ 
6.     Return false
7.   for  $k \in [1, n]$ 
8.     If  $(k \neq i)$ 
9.        $\Pi_{X/Zj} \leftarrow \Pi_{X/Zij} \cup \Pi_{X/Zkj}$ 
10.    end for
11.  end for
12.  $|\Pi_{X/Z}| = \sum_{j=1}^m |\Pi_{X/Zj}|$ 
13. If  $(|\Pi_{X/Z}| \neq |\Pi_{X/Z \cup Y}|)$ 
14.   Return false
15.  $\Pi_X \leftarrow \Pi_{X/Z} \cap \Pi_Z$ 
16.  $\text{Cache} \leftarrow \Pi_X$ 
17. If  $(|\Pi_X| = |\Pi_{X \cup Y}|)$ 
18.   Return true
19. }else {
20.   Return false
21. }
```

The input of Algorithm 5 is the redistributed data set D , the candidate function dependency $\varphi : X \rightarrow Y$, and the output is the verification result. The algorithm first sends a request to the partitioned cache to obtain partial results, then computes the partition of the remaining attributes in X . Then, verify the candidate functional dependency on a single node, if the functional dependency is true on each node, merge the results with the same Key. Before storing the partition in the cache, the merged result is used to verify again to avoid storing invalid partition, and finally output the verification result of candidate function dependency.

4. Experiment

In this chapter, experiments are performed on real and synthetic datasets, and compared with other existing algorithms to verify the efficiency, scalability, and accuracy of the proposed algorithm.

4.1. Experimental Setup

In this experiment, a cluster consisting of 8 servers connected through a local area network is used. The configuration of each server is as follows: the CPU is Intel Xeon2 processor, 32GB memory, and the operating system is Ubuntu 10.4. The algorithm is written in Java and runs on Apache Spark and the HDFS distributed file system.

Three different types of datasets are used in the experiments: (1) A dataset with 0.5 million tuples generated by ONTS [43], the US Department of Transportation's flight statistics. (2) Airline, a dataset with large number of columns, with 109 attributes and 0.5 million tuples [44]. (3) Synthetic dataset, a synthetic dataset Stud with 2 million tuples and 25 attributes. (4) Abalone, a small-scale dataset to evaluate the accuracy of the algorithm.

A summary of the experimental dataset is shown in Table 1.

Table 1. Summary of experimental dataset.

DataSet	#Tuples	#Attributes
ONTS	0.5	64
Airline	0.5	109
Stud	2	25
Abalone	0.004177	9

4.2. Scalability

In this section, the scalability of DisTFD (Node scalability and Data scale scalability) is evaluated and compared with other algorithms.

Node Scalability. By changing the number of nodes $|V|$, $3 \leq |V| \leq 8$, the dataset scale is fixed, evaluate the scalability of this algorithm to the number of nodes. Figure 7a,b show the response times of algorithms Cet, HFDD and DisTFD under different numbers of nodes.

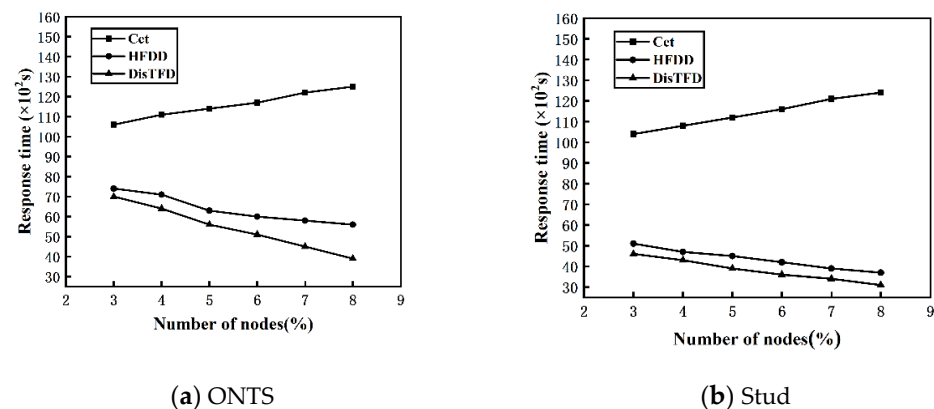


Figure 7. (a) Response time with different number of nodes (ONTS) (b) Response time with different number of nodes (Stud).

As shown in Figure 7a,b, as the number of nodes increases, the response time of the algorithm HFDD and the algorithm DisTFD decreases significantly, and the response time of the algorithm Cet increases slowly. The algorithm Cet verifies the candidate functional dependencies by concentrating the data into master node. When the number of nodes increases, the data of each node migrates to the master node, the amount of data migration becomes larger and the load is unbalanced, which leads to an increase in response time. Algorithm HFDD and algorithm DisTFD verify candidate function dependencies in parallel, so as the number of nodes increases, the response time will be significantly reduced, but when the number of nodes is same, algorithm DisTFD is more efficient than algorithm HFDD. The results show that the algorithm DisTFD has better node scalability.

Data scale scalability. By changing the scale of the data set $|D|$, the scalability of the algorithm for the data scale is evaluated. The fixed number of nodes $|V| = 4$, and the value range of the data scale is 20–100%. Figure 8a,b show the response times of algorithms Cet, HFDD and DisTFD under different data scales, respectively.

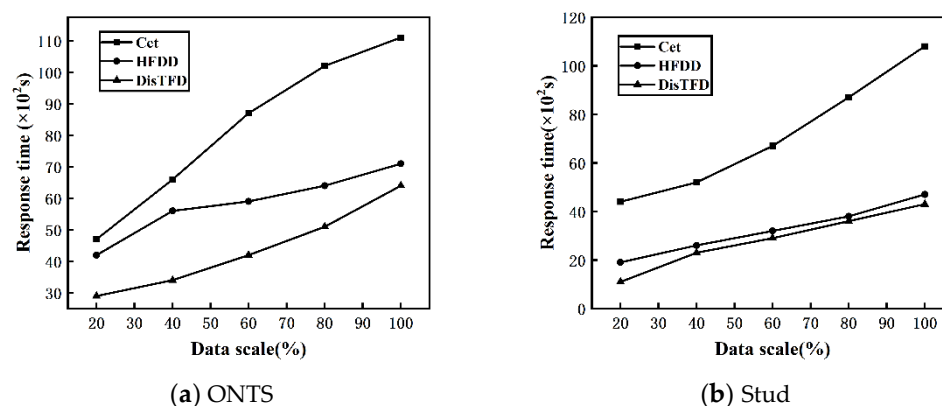


Figure 8. (a) Response time at different data scale (ONTS) (b) Response time at different data scale (Stud).

From Figure 8a,b, it can be concluded that with the expansion of the data scale, the response times of algorithms Cet, HFDD, and DisTFD show an increasing trend. Under the same conditions, the distributed discovery algorithms HFDD and DisTFD have less response time than the centralized discovery algorithm Cet. Compared with the algorithm HFDD, the algorithm DisTFD has a significant improvement in execution efficiency. From the above, it can be concluded that the algorithm DisTFD proposed in this paper has better scalability in terms of data scale.

4.3. Evaluation of Accuracy

In this section, we evaluate the accuracy of the algorithm by comparing the results of algorithms Cet, HFDD, and DisTFD with those of the TANE [25] algorithm, respectively, using the method in the literature [45].

We consider Precision, Recall, and F1measure as the metric of algorithm accuracy. The confusion matrix for classification results is shown in Table 2. Precision, Recall, and F1measure can be calculated as:

$$Precision = TP / (TP + FP) \quad (3)$$

$$Recall = TP / (TP + FN) \quad (4)$$

$$F1measure = 2 * Precision * Recall / (Precision + Recall) \quad (5)$$

Table 2. Confusion matrix for classification results.

Truth	Predicted	
	Positive	Negative
Positive	TP	FN
Negative	FP	TN

As shown in Table 2, the results can be divided into four categories: true positive (TP), false positive (FP), true negative (TN), and false negative (FN). We take the results of algorithm TANE as Truth, and the results of algorithms Cet, HFDD, and DisTFD as prediction values, and calculate Precision, Recall, and F1measure of the three algorithms and compare them, respectively. The comparison results are shown in Table 3.

Table 3. Comparison of Cet, HFDD, and DisTFD accuracy on Abalone.

	Precision	Recall	F1measure
Cet	0.9852	0.9708	0.9780
HFDD	1	0.9781	0.9890
DisTFD	1	0.9854	0.9926

As shown in Table 3, the algorithms Cet, HFDD, and DisTFD have little difference in Precision, Recall, and F1measure, and the F1measure of DisTFD is slightly improved, indicating that all the above algorithms have higher accuracy, but the algorithm DisTFD is more efficient with similar accuracy.

4.4. Evaluation of Performance

In this section, the effectiveness of the proposed method is evaluated by two sets of experiments, respectively.

Evaluation of partition cache. By changing the number of columns in the ONTS and Airline datasets, we evaluate the effect of turning off and on the partition cache on the response time of the algorithm. The fixed number of nodes $|V| = 4$, and the range of the number of data columns is 10–60%. Figure 9a,b shows the change of the response time of the DisTFD algorithm with the partition cache turning on or off as the number of columns increases.

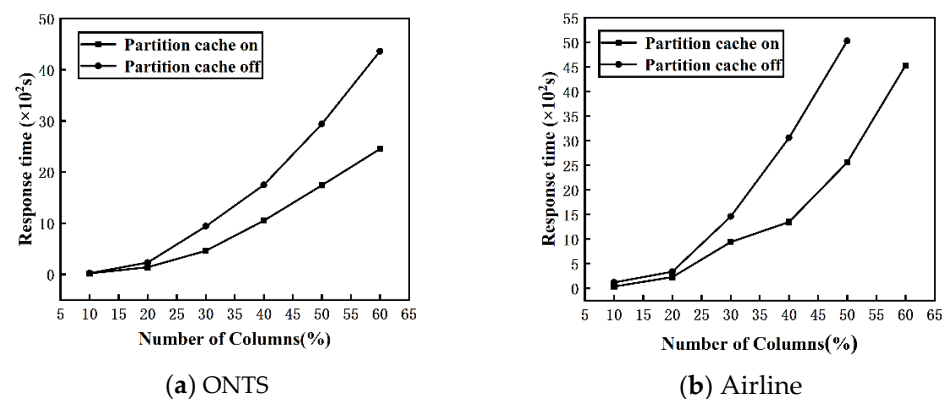


Figure 9. (a) Evaluation of effectiveness of partition cache (ONTS) (b) Evaluation of effectiveness of partition cache (Airline).

As shown in Figure 9, when the number of columns is large, partition caching can significantly reduce the response time of the algorithm. When the number of columns is small, the response time of turning on or off the partition cache does not change significantly. As the number of columns increases, the partition cache significantly improves the execution efficiency of the algorithm.

Evaluation of load balancing. By injecting attribute values with different skewness into the synthetic dataset Stud, the performance of the algorithm under different uniformity of attribute values is evaluated. The fixed number of nodes $|V| = 4$, and according to the ratio of the number of tuples corresponding to the attribute value with the largest attribute value at the left end of the functional dependence to the total number of tuples in the data set from the lowest 10% to the highest 60%, the experiment is carried out. Figure 10 shows the response time of algorithms Cet, HFDD and DisTFD under the different skewness of attribute values.

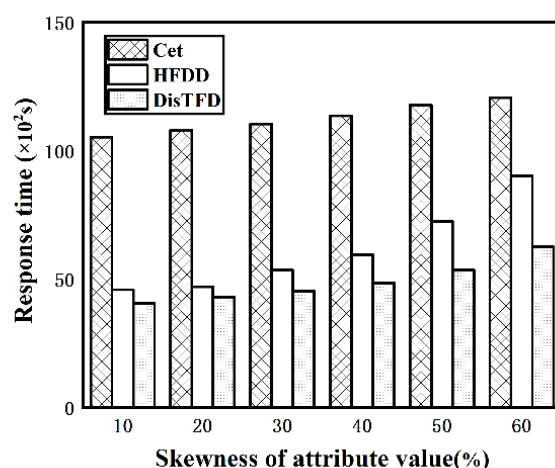


Figure 10. Evaluation of effectiveness of load balancing.

As shown in Figure 10, the response time of algorithm Cet increases slightly with the increase in skewness, and the response time of algorithm HFDD increases significantly in the case of larger skewness. However, the algorithm DisTFD has no significant change in response time as the skewness increases. Therefore, the algorithm DisTFD has better performance in the case of uneven distribution of attribute values.

5. Conclusions and Future Work

Aiming at the problems existing in the process of centralized functional dependency discovery, this paper proposes an algorithm to discover functional dependencies from distributed data. This paper proposes a functional dependency discovery algorithm suitable for distributed data, focusing on reducing the response time of distributed functional dependency discovery. In order to improve the efficiency of functional dependency discovery in a distributed environment, the intermediate results in the calculation process are stored in the cache to reduce the repeated calculation of equivalence classes. Balance the load during the verification process to avoid inefficiencies caused by the unbalanced load. The proposed algorithm is validated on real and synthetic datasets. The results show that the algorithm has good scalability in terms of node and data scale, and significantly improves the execution efficiency compared with existing methods. In future work, we will consider discover approximately functional dependencies and discover functional dependencies in the case of incomplete data. In addition, how to improve the column scalability of the algorithm is also a problem that needs to be considered.

Author Contributions: Conceptualization, W.W. and W.M.; methodology, W.W. and W.M.; software, W.M.; validation, W.W.; formal analysis, W.W. and W.M.; writing—original draft preparation, W.M.; writing—review and editing, W.W.; supervision, W.W.; project administration, W.W.; funding acquisition, W.W. All authors have read and agreed to the published version of the manuscript.

Funding: The authors are supported by the Science and Technology Research Project of Higher Education of Hebei Province Nos. ZD2021011.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Provost, F.; Fawcett, T. Data science and its relationship to big data and data-driven decision making. *Big Data* **2013**, *1*, 51–59. [[CrossRef](#)] [[PubMed](#)]
2. Rubin, D.B. Inference and missing data. *Biometrika* **1976**, *63*, 581–592. [[CrossRef](#)]

3. DeSimone, J.A.; Harms, P.D. Dirty data: The effects of screening respondents who provide low-quality data in survey research. *J. Bus. Psychol.* **2018**, *33*, 559–577. [[CrossRef](#)]
4. Yetman, M.H.; Yetman, R.J. Do donors discount low-quality accounting information? *Account. Rev.* **2013**, *88*, 1041–1067. [[CrossRef](#)]
5. Jordan, M.I.; Mitchell, T.M. Machine learning: Trends, perspectives, and prospects. *Science* **2015**, *349*, 255–260. [[CrossRef](#)]
6. Prokoshyna, N.; Szlichta, J.; Chiang, F.; Miller, R.J.; Srivastava, D. Combining quantitative and logical data cleaning. In Proceedings of the 41st International Conference on VLDB Endowment, Waikoloa Village, HI, USA, 31 August–4 September 2015.
7. Reddy, A.; Ordway-West, M.; Lee, M.; Dugan, M.; Whitney, J.; Kahana, R.; Ford, B.; Muedsam, J.; Henslee, A.; Rao, M. Using gaussian mixture models to detect outliers in seasonal univariate network traffic. In Proceedings of the 2017 IEEE Security and Privacy Workshops (SPW), San Jose, CA, USA, 25 May 2017; pp. 229–234.
8. Mariet, Z.; Harding, R.; Madden, S. *Outlier Detection in Heterogeneous Datasets Using Automatic Tuple Expansion*; MIT Computer Science & Artificial Intelligence Laboratory: Cambridge, MA, USA, 2016.
9. Liu, Y.; Li, Z.; Zhou, C.; Jiang, Y.; Sun, J.; Wang, M.; He, X. Generative adversarial active learning for unsupervised outlier detection. *IEEE Trans. Knowl. Data Eng.* **2019**, *32*, 1517–1528. [[CrossRef](#)]
10. Schelter, S.; Lange, D.; Schmidt, P.; Schelter, S.; Lange, D.; Schmidt, P.; Celikel, M.; Biessmann, F. Automating large-scale data quality verification. In Proceedings of the 44th International Conference on VLDB, Rio de Janeiro, Brazil, 27–31 August 2018; pp. 1781–1794.
11. Dallachiesa, M.; Ebaid, A.; Eldawy, A.; Elmagarmid, A.; Ilyas, I.F.; Ouzzani, M.; Tang, N. NADEEF: A commodity data cleaning system. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; pp. 541–552.
12. Rammelaere, J.; Geerts, F. Cleaning data with forbidden itemsets. *IEEE Trans. Knowl. Data Eng.* **2019**, *32*, 1489–1501. [[CrossRef](#)]
13. Koumarelas, I.; Papenbrock, T.; Naumann, F. MDedup: Duplicate detection with matching dependencies. In Proceedings of the 46th International Conference on VLDB, Tokyo, Japan, 31 August–4 September 2020; pp. 712–725.
14. Chu, X.; Ilyas, I.F.; Koutris, P. Distributed data deduplication. In Proceedings of the 42nd International Conference on VLDB, New Delhi, India, 5–9 September 2016; pp. 864–875.
15. Pena, E.H.M.; de Almeida, E.C.; Naumann, F. Discovery of approximate (and exact) denial constraints. In Proceedings of the 45th International Conference on VLDB, Los Angeles, CA, USA, 26–30 August 2019; pp. 266–278.
16. Yao, H.; Hamilton, H.J. Mining functional dependencies from data. *Data Min. Knowl. Discov.* **2008**, *16*, 197–219. [[CrossRef](#)]
17. Li, J.; Liu, J.; Toivonen, H.; Yong, J. Effective pruning for the discovery of conditional functional dependencies. *Comput. J.* **2013**, *56*, 378–392. [[CrossRef](#)]
18. Chu, X.; Ilyas, I.F.; Papotti, P. Discovering denial constraints. In Proceedings of the 7th International Conference on VLDB, Riva del Garda, Italy, 30 August 2013; pp. 1498–1509.
19. Rekatsinas, T.; Chu, X.; Ilyas, I.F.; Ré, C. Holoclean: Holistic data repairs with probabilistic inference. *arXiv* **2017**, arXiv:1702.00820. [[CrossRef](#)]
20. Chiang, F.; Gairola, D. Infoclean: Protecting sensitive information in data cleaning. *J. Data Inf. Qual. (JDIQ)* **2018**, *9*, 1–26. [[CrossRef](#)]
21. Papenbrock, T.; Ehrlich, J.; Marten, J.; Neubert, T.; Rudolph, J.-P.; Schönberg, M.; Zwiener, J.; Naumann, F. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proc. VLDB Endow.* **2015**, *8*, 1082–1093. [[CrossRef](#)]
22. Gu, C.; Cao, J. Functional Dependency Discovery on Distributed Database: Sampling Verification Framework. In Proceedings of the International Conference on Data Service, Istanbul, Turkey, 11–12 October 2019; Springer: Singapore, 2019; pp. 463–476.
23. Tu, S.; Huang, M. Scalable functional dependencies discovery from big data. In Proceedings of the 2016 IEEE Second International Conference on Multimedia Big Data (BigMM), Taipei, Taiwan, 20–22 April 2016; pp. 426–431.
24. Li, W.; Li, Z.; Chen, Q.; Jiang, T.; Liu, H. Discovering functional dependencies in vertically distributed big data. In Proceedings of the International Conference on Web Information Systems Engineering, Miami, FL, USA, 1–3 November 2015; Springer: Cham, Germany, 2015; pp. 199–207.
25. Huhtala, Y.; Kärkkäinen, J.; Porkka, P.; Toivonen, H. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.* **1999**, *42*, 100–111. [[CrossRef](#)]
26. Novelli, N.; Cicchetti, R. Fun: An efficient algorithm for mining functional and embedded dependencies. In Proceedings of the International Conference on Database Theory, London, UK, 4–6 January 2001; Springer: Berlin/Heidelberg, Germany, 2001; pp. 189–203.
27. Yao, H.; Hamilton, H.J.; Butz, C.J. FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences. In Proceedings of the ICDM, Maebashi City, Japan, 9–12 December 2002; pp. 729–732.
28. Lopes, S.; Petit, J.M.; Lakhal, L. Efficient discovery of functional dependencies and armstrong relations. In Proceedings of the International Conference on Extending Database Technology, Konstanz, Germany, 27–31 March 2000; Springer: Berlin/Heidelberg, Germany, 2000; pp. 350–364.
29. Wyss, C.; Giannella, C.; Robertson, E. Fastfids: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In Proceedings of the International Conference on Data Warehousing and Knowledge Discovery, Munich, Germany, 5–7 September 2001.
30. Papenbrock, T.; Naumann, F. A hybrid approach to functional dependency discovery. In Proceedings of the 2016 International Conference on Management of Data, San Francisco, CA, USA, 26 June–1 July 2016; pp. 821–833.

31. Kivinen, J.; Mannila, H. Approximate dependency inference from relations. In Proceedings of the International Conference on Database Theory, Berlin, Germany, 14–16 October 1992.
32. Ilyas, I.F.; Markl, V.; Haas, P.; Brown, P.; Aboulnaga, A. CORDS: Automatic discovery of correlations and soft functional dependencies. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France, 13–18 June 2004; pp. 647–658.
33. Sánchez, D.; Serrano, J.M.; Blanco, I.; Martin-Bautista, M.J. Using association rules to mine for strong approximate dependencies. *Data Min. Knowl. Discov.* **2008**, *16*, 313–348. [\[CrossRef\]](#)
34. Mandros, P.; Boley, M.; Vreeken, J. Discovering reliable approximate functional dependencies. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, 13–17 August 2017; pp. 355–363.
35. Flach, P.A.; Savnik, I. Database dependency discovery: A machine learning approach. *AI Commun.* **1999**, *12*, 139–160.
36. Caruccio, L.; Deufemia, V.; Polese, G. Mining relaxed functional dependencies from data. *Data Min. Knowl. Discov.* **2020**, *34*, 443–477. [\[CrossRef\]](#)
37. Li, W.; Li, Z.; Chen, Q.; Jiang, T. Discovering Approximate Functional Dependencies from Distributed Big Data. In Proceedings of the Asia-Pacific Web Conference, Suzhou, China, 23–25 September 2016.
38. Mostafa, S.A.; Ahmad, I.A. Recent developments in systematic sampling: A review. *J. Stat. Theory Pract.* **2018**, *12*, 290–310. [\[CrossRef\]](#)
39. Cheng, F.; Yang, Z. New Pruning Methods for Mining Minimal Functional Dependencies from Large-Scale Distributed Data. In Proceedings of the 2018 Sixth International Conference on Advanced Cloud and Big Data (CBD), Lanzhou, China, 12–15 August 2018; pp. 269–274.
40. Qu, J.F.; Hang, B.; Wu, Z.; Wu, Z.; Gu, Q.; Tang, B. Efficient mining of frequent itemsets using only one dynamic prefix tree. *IEEE Access* **2020**, *8*, 183722–183735. [\[CrossRef\]](#)
41. Kendrick, P.; Baker, T.; Maamar, Z.; Hussain, A.; Buyya, R.; Al-Jumeily, D. An efficient multi-cloud service composition using a distributed multiagent-based, memory-driven approach. *IEEE Trans. Sustain. Comput.* **2018**, *6*, 358–369. [\[CrossRef\]](#)
42. Ghomi, E.J.; Rahmani, A.M.; Qader, N.N. Load-balancing algorithms in cloud computing: A survey. *J. Netw. Comput. Appl.* **2017**, *88*, 50–71. [\[CrossRef\]](#)
43. Available online: <https://www.transtats.bts.gov/> (accessed on 13 November 2021).
44. Available online: <https://www.bts.gov/topics/airlines-and-airports-0> (accessed on 26 October 2021).
45. Babić, I.; Miljković, A.; Čabarkapa, M.; Nikolić, V.; Đorđević, A.; Randelović, M.; Randelović, D. Triple Modular Redundancy Optimization for Threshold Determination in Intrusion Detection Systems. *Symmetry* **2021**, *13*, 557. [\[CrossRef\]](#)