Adaptive Multi-copy Layout Algorithm Based on Mass Storage System

Guosong Jiang*, Qing Zhang

School of Computer Science, Huanggang Normal University, Huanggang 438000, China

(hustjgs@126.com)

Abstract

Large-scale storage systems face significant challenges in reliability and adaptability, thus it needs reliable, adaptive and effective data layout algorithms. Existing studies only partially meet these goals. This paper first puts forward a reliable copy data layout algorithm (RCDL) and an effective adaptive data layout algorithm (ADL), and on this basis, by combining the two algorithms, this paper proposes a multi-copy adaptive data layout algorithm MCADL, which can achieve better reliability, adaptability and effectiveness. The RCDL distributes the same copies to different storage devices to avoid the same replica on adjacent storage devices, thus obtaining higher redundancy and fault tolerance. The ADL algorithm combines the clustering algorithm with the consistent hash method, and introduces a small amount of virtual storage devices, greatly reducing the consumption of storage space. Data are distributed fairly according to the weights of the storage devices, so it is adaptive to system expansion and reduction. In order to utilize the respective advantages of RCDL and ADL, MCADL divides data into hot and cold data according to the data access frequency. RCDL layout is used for hot data and ADL layout is used for cold data. Theoretical and experimental results show that MCADL can obtain higher redundancy and fault tolerance and can fairly distribute data and add and remove adaptive storage devices according to the weights of storage devices, migrate optimal data amount when the scale of the storage system changes, and can quickly locate data, consuming less storage space.

Key words

Large-scale network storage, Data layout.

1. Introduction

A data layout algorithm is mainly used to solve the problem of how to choose the storage devices to store data and establish the mapping relationship between data collection and storage devices collection using an effective mechanism and needs to meet certain goals at the same time, such as fair distribution of data on the storage device, adaptive storage scale changes, same copies on different storage devices and location of data within a very short period of time and space. How to effectively lay out data is a major challenge faced by large-scale network storage systems. At present, most of the data layout algorithms consider only a single copy. However, if only a single copy is stored, it would be difficult to restore it once damaged, and thus reliability and availability are affected. A feasible method is to store multiple copies of the data block and distribute the multiple copies between the storage devices. The placement policy for multiple copies of data involves the redundancy problem - different copies of each data block should be stored on different storage devices; in other words, the same copies of the data block cannot be stored on the same storage device. Some working groups are committed to research on redundant data layout algorithms. This paper divides such algorithms into two categories: continuous copy layout and random copy layout.

A continuous copy layout algorithm chooses a storage device to store the master copy of the data using certain a single-copy layout algorithm, and then stores the rest of the copies on subsequent storage devices. The continuous copy layout is well applied in P2P systems (such as CFS [1] and PAST [2]). The same copies are assigned to different storage devices to ensure redundancy and improve reliability. However, when a storage device fails, the system fault-tolerance will deteriorate and its reliability will be reduced because the workload of the failed storage device will be be gathered at an adjacent storage device. Moreover, the continuous copy layout algorithm cannot adapt to the changes of storage devices. When the collection of storage devices is changed, the migrated data are concentrated in the peripheral storage devices. A random copy layout algorithm randomly assigns copies to storage devices [3-6]. When a storage device fails, the load is randomly dispersed in the system; therefore, the random copy layout has better fault-tolerance than the continuous copy layout. However, it cannot guarantee the same copies are placed on different storage devices, reducing the redundancy and leading to low reliability. Another disadvantage of the random copy layout is the poor adaptivity, because changes in the storage devices will cause data to reorganize. Some random algorithms can solve the problem of low redundancy [7-10]. Algorithm recursively places k copies and ensures that the same copies are on different storage devices. However, when the collection of storage devices is changed, the amount of data re-organization is k2 times the most optimal strategy. The self-adaptability of RUSH and

CRUSH algorithm in terms of addition on a single storage device is both poor. These random algorithms all try to avoid the redundancy problem, but still have a poor adaptability.

2. Reliable Copy Layout Algorithm

In order to eliminate the defects in the fault tolerance of the continuous copy layout and avoid the low redundancy problem of the random copy layout, this paper proposes a reliable copy layout algorithm (RCDL) which assigns copies of data to different storage devices. Its self-adaptivity is discussed in Section 2.2. First, this section defines three kinds of relevance and redundancy regarding storage devices and data collection, abstracts the layout problem as RCDL and proves it to be the NP- hard problem. Then, it weakens the conditions of the problem, abstracts it into a semi-definite programming problem and uses a polynomial time algorithm to solve the problem. Given the size of the problem, this paper analyzes the cost of semi-definite programming.

2.1 RCDL Problem

First this paper introduces the definitions related to the RCDL problem. The heterogeneous mass storage system for large-scale data can be abstracted as a mathematical model. The data and the storage devices are regarded as a set respectively. The problem is abstracted to how to establish the mapping between these two sets.

Suppose the data set in storage system is: $X_0=\{x_1,...,x_m\}$, where m represents the total number of data, and $x_i \in x_0$ represents a data element.

Suppose the storage device space D is $\{d_1,…,d_N\}$, where N represents the total number of storage devices, and the d_i represents a storage device. The weight can be expressed as capacity, bandwidth or a combination of both.

Suppose the current storage device set is $D_0=\{d_1,…,d_N\}$, where $n \leq N$, and $D_0 \subseteq D$. Suppose the capacity of a storage device $d_i \in D_0$ is C_i, it means the amount of data that can be stored on the device. For example, if the capacity of a storage device is 5000, then the storage device can store 5000 data. To put it simple, data x_i is expressed as i, and the storage device d_k as k.

DEF.1 (Correlation between data and storage devices)

$\forall i \in X_0$. If i is assigned to storage device $k \in D_0$, then $l(i,k)=1$; else $l(i,k)=0$.

DEF.2 (Copy correlation among data elements).

$\forall i,j \in X_0$, where $i \neq j$. If i and j are the same copies, then $rc(i,j)=1$; else $rc(i,j)=0$.

DEF.3 (Location correlation between data elements)

$\forall i, j \in X_0$, where $i \neq j$. If i and j are laid out on different storage devices, then $lc(i,j)=1$; else $lc(i,j)=0$.

DEF.4 (Redundancy)

$\forall i, j \in X_0$, where $i \neq j$ and $rc(i,j)=1$. Suppose when $lc(i,j)=1$, the redundancy is $\lambda \cdot degree(i,j)$, where λ is the adjustment factor; otherwise, the redundancy is 0. λ is related to the importance of data, which gives more attention to important data.

The definition of the problem RCDL is as follow:

$$IP:\ maximize(\textstyle\sum_{i,j\in X_0 \wedge i\neq j} \lambda \cdot degree(i,j) \cdot rc(i,j) \cdot lc(i,j)) \tag{1}$$

Constraints:

$$\forall\, i \in X_0:\ \textstyle\sum_{k\in D_0} l(i,k) = 1 \tag{2}$$

$$\forall\, k \in D_0:\ \textstyle\sum_{i\in X_0} l(i,k) \leq C_k \tag{3}$$

The problem of the reliable copy layout of RCDL is defined above. In this definition, the goal of Formula (1) is to maximize redundancy. Constraint (2) ensures that data are only placed on a single storage device and Constraint (3) ensures that the amount of data on each storage device does not exceed the capacity of the storage device.

Theorem 1: RCDL is an NP-hard problem

Proof: consider an instance of RCDL problem. Suppose the storage device set is $D_0=\{d_1,\ldots,d_N\}$, the data set is $X_0=\{x_1,\ldots,x_m\}$, and the capacity of each storage device is C. Suppose that none of the data sizes is greater than C and data can be freely placed on any storage device, thus ensuring the data amount on each storage device will not exceed its capacity. We construct an undirected entitled graph $G(V,E)$, with the data in X_0 as a node and the correlation between data as sides. The weight of each side is $\lambda \cdot degree(i,j) \cdot rc(i,j) \cdot lc(i,j)$, Then, the instance is converted to seeking the maximum n-separation problem of the graph G. Any instance of the maximum n-separation problem of the graph G can be used as an instance of RCDL problem. Given the maximum n-separation problem is NP-hard, the RCDL problem is NP-hard accordingly.

2.2 RCDL Semi-definite Programming Relaxation

The RCDL problem is NP-hard. We can relax the RCDL as a semi-definite programming

problem (SDP), which makes the SDP solved in polynomial time. First, we re-define the RCDL problem. Suppose $\{a_1, a_2, \ldots, a_n\}$ is the n-dimension vector set, where $a_1=(1,0,\ldots0)$, $a_2=(0,1,\ldots0)$ and $a_n=(0,0,\ldots1)$. We modify DEF.1 to DEF.5.

DEF.5 (Correlation between data and storage devices)

$\forall\ i \in X_0$. If i is assigned to storage device $k \in D_0$, then $l(i,k)=1$; else $l_i=a_k$. $lc(i,k)$ can be re-written as $(1 - l_i \cdot l_j)$, where $l_i=\{a_1, a_2, \ldots, a_n\}$. Then we re-write $\lambda \cdot degree(i,j) \cdot rc(i,j)$ to $w(i,j)$. The definition of the RCDL problem is equivalent to the following re-definition of the problem RCDL.

$$IP': maximize \sum_{i,j \in X_0 \wedge i \neq j} w(i,j)(1 - l_i \cdot l_j) \tag{4}$$

Constraints:

$$\forall\ i \in X_0: l_i \in \{a_1, a_2, \cdots, a_n\} \tag{5}$$

$$\forall\ k \in D_0: \sum_{i \in X_0} a_k \cdot l_i < C_k \tag{6}$$

Suppose $S_{n-1} = \{x \in R_n : |x| = 1\}$ is the unit sphere of n-dimension vectors. We replace l_i with v_i, and then v_i is a vector in S_{n-1}.

$$SDP: maximize \sum_{i,j \in X_0 \wedge i \neq j} w(i,j)(1 - v_i \cdot v_j) \tag{7}$$

Constraints:

$$\forall\ i \in X_0: v_i \in S_{n-1} \tag{8}$$

$$\forall\ k \in D_0: \sum_{i \in X_0} a_k \cdot v_i < C_k \tag{9}$$

$v_i \cdot v_j$ is expressed as X_{ij}, and then $X=[X_{ij}]$ is positive semi-definite. The target of Formula (7) can be expressed as $maximize \sum_{i,j \in X_0 \wedge i \neq j} w(i,j) \cdot (1 - X_{ij})$. This formula is equivalent to $minimize \sum_{i,j \in X_0 \wedge i \neq j} w(i,j) \cdot X_{ij}$. Constraint (8) can be expressed as $\forall\ i \in X_0: X_{ii} = 1$. We add a condition $X \succ 0$ to show that X is positive semi-definite. Then the SDP problem can take four steps to solve:

1. First of all, we solve the semi-definite programming problem as follows, and then get the matrix X.

$$SDP': maximize \sum_{i,j \in X_0 \wedge i \neq j} w(i,j)(1 - X_{ij}) \iff minimize \sum_{i,j \in X_0 \wedge i \neq j} W \cdot X \tag{10}$$

Constraints:

$$\forall i \in X_0: X_{ii} = 1 \tag{11}$$

$$X \succ 0 \tag{12}$$

2. Suppose $A=(v_1, v_2,\ldots,v_m)^T$. Because $X=[X_{ij}]=[v_i v_j]$, we have $X=A \cdot A^T$. We solve $X=A \cdot A^T$. to get the matrix A, and thus obtain the vector $v_1, v_2,\ldots,v_m \in S_n$.

3. Each element in vector $v_i \in \{v_1, v_2,\ldots,v_m\}$ represents the relationship between the data i and storage device m. We round v_i using a rounding algorithm [11], making data only assigned to a single storage device. First of all, we take n random vectors $r_1, r_2,\ldots,r_n$ from the n-dimensional space and set $r_i = (r_1^i, r_2^i, \cdots, r_n^i)$ and $1 \leq i \leq n$. r_j^i is a sample obtained from the standard distribution $(0,1)$, where $1 \leq j \leq n$. If the distance from a vector in n random vectors $r_1, r_2,\ldots,r_n$ to v_i is the shortest, then the data i is assigned to this vector.

4. To ensure data are fairly laid out on storage devices, we sort $r_1, r_2,\ldots,r_n$ by the amount of data obtained by each vector, and meanwhile sort the storage devices by their capacities, and then distribute the data obtained by vector r_i to the corresponding storage device i.

5. We repeat this random rounding process to make the amount of data on each storage device not exceed its capacity, so condition (6) is met.

2.3 Computing Cost

We measure the computing cost of RCDL by analyzing the five steps to solve SDP. The data amount and the number of storage devices are m and n, respectively.

In the first step, the variable related to the matrix X is $\frac{(m-1) \times m}{2}$, and the number of equations involved in Constraint (11) is m. In the second step, there are $m \times n$ variables and $\frac{(m-1) \times m}{2}$ equations. Because the data amount is much greater than the number of storage devices, the total number of variables and equations is $\frac{m \times (m-1)}{2} + m + m \times n + \frac{m \times (m+1)}{2} = m^2 + m + m \times n = O(m^2)$.

The rounding number in the rounding algorithm is $m \times n$, and the complexity of the rounding algorithm is $O(m \times n)$. In the fourth step, the sorting operation of vectors and storage devices can be done within $O(n \times logn)$ time; therefore, the complexity of rounding and sorting is $O(m \times n)$. The rounding algorithm needs to be executed several times to get good fairness.

3. Adaptive Data Layout Algorithm (ADL)

In order to achieve adaptability and efficiency of the layout algorithm, we put forward the ADL algorithm, which integrates the clustering algorithm and uniform hash method and introduces a small amount of virtual storage devices, greatly reducing the required storage space. The ADL algorithm can fairly distribute data and adaptively add/delete storage devices in accordance with the weights of the storage devices and migrate the least data amount when the storage system scale is changed, and moreover, locate data quickly and consume less memory space.

3.1 Problem Definition

Suppose the function f_0: $X_0 \rightarrow D_0$ maps the data set X_0 to the storage device set D_0. The relative weight of the storage device $i \in D_0$ is w_i, and then the weights of the n storage devices are $\{w_1, w_2, \ldots, w_n\}$, respectively, and $\sum_{i=1}^{n} w_i = 1$. The data layout algorithm A can be represented as a binary function $A(x,f)$, $x \in X_0$, and the parameter f represents a matching function, and $A(x,f) = A(x)$. The data x assigned to the storage device is identified by the value of $A(x,f)$.

DEF.6 (Fairness)

Let the storage device set of storage system be $D_0 = \{d_1, d_2, \ldots, d_n\}$, the relative weight of the storage device $d_i \in D_0$ be w_i, and the data set be $X_0 = \{x_1, x_2, \ldots, x_m\}$, and suppose the function f_0: $X_0 \rightarrow D_0$ map the data set X_0 to the storage device set D_0, $\forall x \in X_0$, and that the probability that the layout algorithm A assigns the data X to the device $d_i \in D_0$ is p. $\forall \varepsilon > 0$, and if $|p(A(x, f_0) = d_i) - w_i| < \varepsilon$, then the algorithm A is fair [12].

The fairness of the algorithm makes the probability of the data $x \in X_0$ assigned to the storage device $d_i \in D_0$ infinitely close to the relative weight of the device d_i, thus ensuring the data are fairly distributed to each storage device.

DEF.7 (Adaptive)

Let the storage device set of the storage system be $D_0 = \{d_1, d_2, \ldots, d_n\}$, the relative weight of the storage device $d_i \in D_0$ be w_i and data set be $X_0 = \{x_1, x_2, \ldots, x_m\}$, and suppose the function f_0: $X_0 \rightarrow D_0$ maps the data set X_0 to the storage device set D_0. If the layout algorithm A meets the following two requirements:

(1) The current storage device set D_0 becomes $D^+ = \{d_1, d_2, \ldots, d_{n+1}\}$. Suppose the function f_+: $X_0 \rightarrow D^+$ maps the data set X_0 to the storage device set D^+. $\forall\, x \in X_0$, and if $A(x, f_+) = f_+(x) \in D_0$, then $A(x, f_+) = A(x, f_0)$;

(2) The current storage device set D_0 becomes $D^- = \{d_1, d_2, \cdots, d_{i-1}, d_{i+1}, \cdots, d_n\}$. Suppose the function $f_-: X_0 \longrightarrow D^-$ maps the data set X_0 to the storage device set D^-. $\forall\, x \in X_0$, and if $A(x, f_0) = f_0(x) \in D^-$, then $A(x, f_0) = A(x, f_-)$.

Then the layout algorithm A is adaptive.

When the storage system scale is changed, due to its adaptability, the layout algorithm makes the migrated data only move from the non-changed storage devices to the newly added storage devices or move from the deleted storage devices to the non-changed storage devices, and there is no data migration among the non-changed storage devices, thus ensuring the least amount of data migrated

When the storage device d_{n+1} is added, the storage device set D_0 becomes $D^+ = \{d_1, d_2, \cdots, d_{n+1}\}$,, and data migrate only from a storage device in D_0 to the storage device d_{n+1}, and there is no data migration among storage devices in D_0. When the storage device d_i is deleted, the storage device set D_0 becomes $D^- = \{d_1, d_2, \cdots, d_{i-1}, d_{i+1}, \cdots, d_n\}$, and data migrate only between the storage device d_i and the storage devices in D^-, and there is no data migration among storage devices in D^-.

This method makes the amount of data migration equal to the data amount on the added or deleted storage device, and the system can still compute the data location using the algorithm A after data migration without increasing any new rules, thus the algorithm A can adapt to the changes of the storage system scale, which are transparent to users. A user only needs to modify the system's configuration file about system scale, and then correctly position data according to the algorithm A. In the isomorphic-consistent hash mechanism, in order to solve the heterogeneous problem of data layout, we need to introduce the corresponding virtual storage devices according to the weights of storage devices. If the weight difference between storage devices is very big, then we need to introduce a large number of virtual storage devices.

In the worst case, if the weight of a storage device in system is very small and there are large differences between its weight and those of other storage devices, then the number of virtual storage devices introduced to system cannot be tolerated by system memory space. In this paper, the ADL algorithm consists of three steps:

(1) First, classify the storage device set usoffing the clustering algorithms and make the weight differences among storage devices in each class within a preset range.

(2) After completion of the clustering, the layout mechanism between classes in accordance with the weights of the class divides the interval [0,1] into a plurality of sub-intervals, allocates a sub-interval to each class and assigns the data to a sub-interval to the corresponding class.

(3) The inner layout mechanism of each class uses the consistent hash method to re-distribute data to specific storage devices.

The following respectively introduces the clustering algorithm, distribution mechanism among the classes and layout mechanism within the class.

3.2 Effective Adaptive Data Layout Algorithm (ADL)

DEF.8 (Distance between the storage devices)

The weights of d_i and d_j in the storage device set D_0 are w_i and w_j respectively, and then the distance between d_i and d_j is $w_{ij} = |w_i - w_j|$.

DEF.9 (Distance from storage device to class)

Let the storage device class be S and the clustering center be d_i, where $d_i \in S$, and then the distance from the storage device d_j to class S is equal to the distance from the storage device d_j to d_i.

The goal of clustering algorithm is to make the distance from the storage device in each class to its cluster center less than the preset value. We can calculate the distance from a storage device to the cluster center according to the distance formula in Definition 8.

Suppose the intra-class distance threshold value is T. Compare the distance from the storage device to the cluster center with T and determine which class the storage device falls within or take it the center of a new class. First, take any one storage device d_i as the cluster center of the first class S_1. For example, take d_1 as the center of the class S_1, and compute the distance from the next storage device d_2 to ^{D_1}. If the value is less than or equal to T, d_2 will be normalized to the class S_1; otherwise, d_2 will be the center of the new class S_2. Suppose we have k cluster centers. Compute the minimum distance from the unsorted storage device d_i to k cluster centers. If this value is greater than T, make the storage device d_i as the center of the new class S_{k+1}; otherwise assign the storage device to the class nearest to the device in class k. Repeat the process until all storage devices are classified into each class. The clustering algorithm divides the storage device set $D_0 = \{d_1, d_2, \ldots, d_n\}$ into a g class set $C = \{D_1, D_2, \ldots, D_g\}$. Suppose the class D_j is $\{D_i^j; i = 1, 2, \cdots, n_j\}$, where $1 \leq j \leq g$, then $\sum_{j=1}^{g} n_j = n$. Suppose the weight of the class D_j is w_j, then $w_j = \sum_{i=1}^{n_j} w_i$, and the total weight of g classes is $\sum_{i=1}^{g} w_j = 1$. The problem is transformed into distributing data in the heterogeneous class set C. In order to fairly distribute data in the class set $C = \{D_1, D_2, \cdots, D_g\}$, the interval [0,1] is re-

divided into multiple sub-intervals according to the weight of each class in C, and the sub-interval of the class D_j is $I_j = (\sum_{i=1}^{j-1} w_i \cdot \sum_{i=1}^{j} w_i)$. The inter-class allocation algorithm DPBC (Data Placement between Classes) maps the data $x \in X_0$ to the interval $[0,1]$ using the hash function $h_2: X \rightarrow [0,1]$. If $h_2(x) \in I_j$, then assign X to the sub-interval I_j. By the probability theory, it is known that the data amount falling into each interval is proportional to the interval size. Thus data can be evenly distributed to the class set $C = \{D_1, D_2, \cdots, D_g\}$.

Suppose the function $f_c: X_0 \rightarrow C$ represents mapping the data set X_0 to the class set C, then the DPBC can be expressed as the function $DPBC(x, f_0)$, where $DPBC(x, f_c) = f_c(x)$.

The inter-class allocation mechanism fairly distributes data to each class, and at the same time adds a new class, migrating the optimum data amount to allow data distribution to achieve fairness again. We also need a layout mechanism within class to fairly assign data distributed to each class to each storage device in the class, and migrate the least data amount when the storage scale is changed.

After clustering the storage device set in the storage system, the storage device capacity of each class is little different, and thus there will not be too many virtual storage devices introduced while the consistent hash mechanism within class is used. The intra-class layout algorithm DPIC (Data Placement in Class) adopts the consistent hash mechanism standard [13]. Suppose the interval of the class D_j is I_j, and the data set falling into the class D_j is X_j according to the DPBC algorithm. The hash values of the data elements in X_j are gathered in the interval I_j in $[0,1]$, which, under the state of local aggregation, are not in uniform distribution in interval $[0,1]$. In order to ensure fairness, recalculate the hash values of the data elements in X_j using $h_3: X_j \rightarrow [0,1]$, make the data elements in X_j fairly distributed in the interval $[0,1]$. $\forall d_i^j \in D_j$, and map d_i^j to one point in $[0,1]$ using $h_1: D_j \rightarrow [0,1]$. Because the element in D_j is unevenly distributed in the interval $[0,1]$, the corresponding virtual storage device for each storage device needs to be introduced. According to the clustering algorithm, it is known that the weight difference between any two storage devices in D_j is less than $2T$. We ignore the weight difference between storage devices, and then the number of introduced virtual storage devices for each storage device is the same. According to Literature [13], in order to ensure fairness, the number of introduced virtual storage devices for each storage device is $klog|N|$. Copy each storage device $klog|N|$ times, and then use h_1 to map each virtual storage device to the interval $[0,1]$. $\forall x \in X_j$. Suppose the distance from X to a storage device d_i^j is $\left|h_3(x) - h_1(d_i^j)\right|$. Calculate the distance from X to each storage device (including virtual storage devices), and allocate the data to the nearest storage device.

Suppose the function $f_j: X_j \to D_j$ represents mapping the data set X_j to the class set D_j, then the DPIC can be expressed as the function $DPIC(x, f_j)$, where $DPIC(x, f_j) = f_j(x)$.

3.3 Data Re-organization

The storage system can batch increase storage devices at one time during extension. Usually the capacities and performance of these storage devices are very similar, which can be categorized as a new class; therefore the inter-class distribution mechanism needs to adapt to the storage scale. Storage devices in class are not allocated according to the physical locations; therefore deleting the whole storage devices in class does not have generality, which is not considered here. The situation of adding and deleting a single storage device will be described in data migration within class. The following discusses the data transport process of batch-adding storage devices. In the initial case, the clustering algorithm divides the storage device set in the system into g classes, and the interval [0,1] is divided into g sub-intervals. The storage devices are batch-added as a new class D_{g+1}. Since the class D_{g+1} is introduced, the weights of all classes in the system are changed, and thus the interval needs to be re-distributed in accordance with the new class weight. Suppose the weight of the class D_j is w_j, where $1 \leq j \leq g$. In order to achieve the fairness of data distribution, the class D_j needs to migrate $(w_j - w_j')m$ data to the class D_{g+1}, where m represents the total data amount. The interval I_j of Class D_j is divided into $[\sum_{i=1}^{j-1} w_i, \sum_{i=1}^{j-1} w_i + w_j')$ and $[\sum_{i=1}^{j-1} w_i + w_j', \sum_{i=1}^{j} w_i)$, where the interval $[\sum_{i=1}^{j-1} w_i + w_j', \sum_{i=1}^{j} w_i)$ is assigned to the class D_{g+1}, and then the sub-interval I_{g+1} corresponding to class D_{g+1} is $\cup_{j=1}^{g} [\sum_{i=1}^{j-1} w_i + w_j', \sum_{i=1}^{j} w_i)$, and the data falling within the interval are migrated to the class D_{g+1}. In this way, the data are fairly re-distributed in $g+1$ classes. It can be seen from the above migration process that there is only migration of the old data to the new class, and no migration between the old classes, so the data migration amount is equal to the data amount of the new class. Thus, the inter-class allocation algorithm can effectively ensure the fairness and has a good self-adaptability. Add a new class, and divide the interval of the existing classes according to the weights, assign the redundant interval of each class to the new class, and at the same time migrate the data falling into the interval to the new class. When the storage system scale is changed, like adding a new storage device $d_{n_j+1}^{j}$, use $h_1: D_j \to [0,1]$ to map $d_{n_j+1}^{j}$ to a certain point in the interval [0,1]. For data X on the left neighbor and right neighbor of this point, if X is closer to the point, it is migrated to the new storage device. Thus the migration result is the same as the calculation result by the layout algorithm. Similarly, when the old storage device d_i^{j} is deleted, for data on the storage device, calculate the distance to the left neighbor and right neighbor

and migrate the data to the nearest neighbor. Repeat the same process for virtual storage devices which add new storage devices or delete storage devices. The processing for virtual storage devices can make data distribution after migration achieve fairness again. From the above migration process, it can be seen that the data migrate only between added or deleted storage devices and unchanged storage devices, and does not introduce additional data migration between unchanged storage devices.

3.4 Theoretical Analysis

1. Adaptivity

In the clustering algorithm, storage devices are gathered to each class according to its weight. Storage devices within class are not allocated according to the physical location; therefore, deleting the whole storage devices in class does not have generality, which is not considered here. The storage system can batch increase storage devices at one time during extension. Usually the capacity and performance of these storage devices are very similar, which can be categorized as a new class. For the situation where a single storage device is added, first determine which class the storage device falls within, and then migrate the data within class. While deleting the single storage device, make adjustments within the class. Although migrating data only within class will lose the whole fairness, there are much more storage devices within class, and adding storage devices will barely change the weight of this class. Thus, whether a storage device in a class is added or deleted, the weight of each class hardly changes, and the data are still fairly distributed in all classes. The following discusses the self-adaptivity of the ADL algorithm with two cases.

(1) When storage devices are batch-added, their capacity and performance are similar. Suppose the class set $C = \{D_1, D_2, \cdots, D_g\}$ becomes $C_1 = \{D_1, D_2, \cdots, D_g, D_{g+1}\}$, and the current storage device set D_0 becomes D_0'. For any class $D_j \in C$, recalculate the weight of the D_j according to the weight of D_{g+1}, which will become smaller. Split the interval of the class D_j according to the weight difference and distribute the redundant interval to D_{g+1}, and migrate the data falling into the redundant interval to D_{g+1}. After the migration, suppose the function $f_0': X_0 \rightarrow D_0'$ represents mapping the data set X_0 to the new storage device set D_0'. As can be seen from the migration process, D_j just moves the data out to D_{g+1}, and there is no data moving into D_j and no data moving into the storage device d_i^j in D_j. If a data is stored on the device d_i^j in D_j after the migration, then the data is also stored on the device d_i^j in D_j before migration; in other words, if $ADL(x, f_0') = d_i^j$, and $d_i^j \in D_j$, then $ADL(x, f_0) = d_i^j$. Because $d_i^j \in D_j$, and $D_j \in D_0$, we have $d_i^j \in D_0$. If $ADL(x, f_0') = d_i^j$, and $d_i^j \in D_0$, then $ADL(x, f_0) = d_i^j = ADL(x, f_0')$. Batch-deleting storage devices is not a

general case, so we do not take it into consideration. By Definition 7, it can be seen that the ADL algorithm is adaptive.

(2) Next we consider adding and removing a single storage device. When a single storage device is added, the clustering algorithm first determines which class that storage device falls within and a new storage device will be added to the class. Then $D_j = \{d_i^j; i = 1,2,\cdots,n_j\}$ becomes $D_j' = \{d_i^j; i = 1,2,\cdots,n_j, n_j + 1\}$, and the whole storage device set D_0 becomes D_0''. Map the new storage device $d_{n_j+1}^j$ to a point on the ring $[0,1]$, and migrate part of the data in the left or right neighbor to this point. For several virtual nodes falling into this node, the same data migration method is also used. After the migration, suppose the function $f_0'': X_0 \to D_0''$ represents mapping the data set X_0 to the new storage device set D_0''. As can be seen from the migration process, for any storage device $d_i^j \in D_j$, the data only migrate to the new storage device $d_{n_j+1}^j$, and there is no data migrating into the mapping point and the virtual storage device of d_i^j. If a data is stored on d_i^j in class D_j after the migration, the data is also stored on d_i^j before the migration. And for any other class $D_k \subset D_0$ and $D_k \neq D_j$, the data position in D_k has not changed. if a data is stored on d_i^k in class D_k after the migration, then the data is also stored on d_i^k before migration. To sum up the above, if a data is stored on d_i in class D_0 after the migration, then the data is also stored on d_i before the migration. Therefore, for $1 \leq t \leq g$, if $ADL(x, f_0'') = d_i^t$, and $d_i^t \in D_t$, then $ADL(x, f_0) = d_i^t$. Because $d_i^k \in D_k$, and $D_k \subset D_0$, we have $d_i^k \in D_0$. Therefore, if $ADL(x, f_0'') = d_i^t$, and $d_i^t \in D_0$, then $ADL(x, f_0) = d_i^t$. By Definition 2, it can be seen that the ADL algorithm is adaptive. When a storage device is removed from the class D_j, the whole storage device set D_0 becomes D_0'''. After a similar approach is used to migrate data, suppose the function $f_0''': X_0 \to D_0'''$ represents mapping the data set X_0 to the new storage device set D_0''', the following can also be proved: for $1 \leq t \leq g$, if $ADL(x, f_0) = d_i^t$, and $d_i^t \in D_0$, then $ADL(x, f_0'') = d_i^t = ADL(x, f_0)$. By Definition 7, it can be seen that the ADL algorithm is adaptive.

2. Performance analysis

The ADL algorithm can preprocess the clustering process. One clustering result can be reused for data layout, and thus we do not consider the clustering time in analyzing the time complexity of ADL. The clustering algorithm divides the storage device set into several classes based on the distance threshold within the class. It assigns the interval $[0,1]$ to each class in accordance with the class weight. When arranging the data X among classes, we first traverse all intervals to decide which interval the data X should fall into according to the $h(x)$ value. Suppose the total number of intervals is I, in the initial classification, each class has an interval, so $I = g$. With the addition of

classes, the value of *I* is constantly increasing. The tree data structure is adopted to save intervals and traversing of *I* values can be done within $O(\log I)$ time. When the classes are increased for the first time, $I_1 = g + g = 2g$. When they are increased for the second time, $I_2 \le 2g + (g + 1)$. When they are increased for the s-th time, $I_s \le I_{s-1} + (g + s - 1)$. After the clustering algorithm is adopted, compared with the number of storage devices, the class number is greatly reduced. When the PB scale system is running, the number is limited in the bulk purchase of hundreds of storage devices, thus the value of I does not become greater. Using the consistent hash positioning mechanism within class from the literature, we can calculate the data location within $O(1)$ time. Thus, using the ADL algorithm to locate a data requires $O(\log I)$ time, where *I* represents the interval number in the current system. Next, we discuss the storage requirements of the ADL algorithm. According to the clustering algorithm, we can find that the weight differences between storage devices in class are within a preset range. During intra-class layout, ignore the weight differences between storage devices, and the number of virtual storage devices for each storage device is the same. Compared with the improved simple consistent hash method, the number of virtual storage devices is greatly reduced. In the ADL algorithm, the total number of intervals assigned to all classes is *I*, and each interval needs to be expressed with $\log I$ bits. For any class $D_j\{d_i^j; i = 1, 2, \cdots, n_j\} \in C\{D_1, D_2, \cdots, D_g\}$, suppose the weight of the storage device d_i^j is w_j^i, and the storage device with the smallest weight is w_j^{min}, then the class D_j needs $\sum_{i=1}^{n_j} \frac{w_j^i}{w_j^{min}} k \log|N|$ virtual storage devices. Because the weight differences between storage devices in class are not big, $\sum_{i=1}^{n_j} \frac{w_j^i}{w_j^{min}} k \log|N|$ is approximately equal to $n_j k \log|N|$. The largest number of storage devices in class is $n_{max} = \max_j n_j$, where $1 \le j \le g$. The class with the largest number of storage devices requires $\log(n_{max} k \log|N|)$ bits to represent a virtual storage device. In the whole set of storage devices, representing a virtual storage device requires $(\log I + \log(n_{max} k \log|N|))$ bits. In the improved simple consistent hash method, distinguishing all the virtual storage devices needs $\log\left(\sum_{i=1}^{n} \frac{w_i}{w_{min}} k \log|N|\right)$ bits. When there are relatlvely large differences between the performance and capacities of the storage devices in the storage system, for the 1≤i≤n, the majority values of the $\frac{w_i}{w_{min}}$ are great. Since the number of storage devices in the mass storage system is hundreds or thousands, and per the above discussion, the value of I will not become very large, $(\log I + \log(n_{max} k \log|N|))$ is far less than $\log\left(\sum_{i=1}^{n} \frac{w_i}{w_{min}} k \log|N|\right)$. Therefore, the ADL algorithm reduces a lot of storage space.

3. Fairness

LEMMA.1 (the inter-class algorithm DPBC is fair)

Proof: Suppose the function $f_c: X_0 \to C$ represents mapping the data set X_0 to the class set $C\{D_1, D_2, \cdots, D_g\}$ within the inter-class mechanism, then the inter-class algorithm DPBC can be expressed as a function of $DPBC(x, f_c)$, where $DPBC(x, f_c) = f_c(x)$. The interval distributed to the class $D_j \in C\{D_1, D_2, \cdots, D_g\}$ is I_j. Suppose the length of the interval I_j is $|I_j|$, and the relative weight of the class D_j is w_j. DPBC assigns the interval to classes according to the relative weights of classes, and thus $|I_j|=w_j$. Per the probability theory, it can be found that $\forall \, x \in X_0$, and that the probability of X falling into the interval I_j is $|I_j|$, namely, the probability of X assigned to the class D_j is $|I_j|$. Therefore, $p\big(DPBC(x, f_c) = f_c(x) = D_j\big) = |I_j| = w_j$, that is, $\big|p\big(DPBC(x, f_c) = D_j\big) - w_j\big| = 0$. So, $\forall \, \varepsilon > 0$, $\big|p\big(DPBC(x, f_c) = D_j\big) - w_j\big| < \varepsilon$. According to Definition 6, the algorithm DPBC is fair, and X_0 is fairly distributed in the class set C.

LEMMA.2 (the intra-class algorithm DPIC is fair)

Proof: for any $D_j \in C$, where $1 \le j \le g$, suppose the function $f_j: X_j \to D_j$ represents mapping the data set X_j to the storage device set D_j within the intra-class mechanism, then the intra-class algorithm DPIC can be expressed as a function of $DPIC(x, f_i)$, where $DPIC(x, f_i) = f_i(x)$. Per Literature, $p\big(f_j(x) = d_i^j\big) = \frac{1}{n_j}$, where $d_i^j \in D_j$. Because the relative weight of d_i^j in the D_j is w_i^j, which is equal to $\frac{1}{n_j}$, and $DPIC(x, f_j) = f_j(x)$, we have $p\big(DPIC(x, f_j) = f_j(x) = d_i^j\big) = \frac{1}{n_j} = w_i^j$. i.e. $\big|p\big(DPIC(x, f_j) = d_i^j\big) - w_i^j\big| = 0$. So, $\forall \, \varepsilon > 0$, $\big|p\big(DPIC(x, f_j) = d_i^j\big) - w_i^j\big| < \varepsilon$. According to Definition 6, the DPIC algorithm is fair, and Xj is fairly distributed in the storage device set D_j.

Theorem 2: the ADL algorithm is fair

Proof: suppose the function $f_0: X_0 \to D_0$ represents mapping the data set X_0 to the storage device set D_0 within the layout mechanism proposed by this paper, then the ADL algorithm can be expressed as a function of $ADL(x, f_0)$, where $ADL(x, f_0) = f_0(x)$. The ADL algorithm is composed by the DPBC algorithm and the DPIC algorithm. Therefore, we have $\forall \, x \in X_0$. Suppose $ADL(x, f_0) = f_0(x) = d_i$, then there exists $D_j \in C$, making $DPBC(x, f_c) = D_j$, $DPIC(x, f_i) = d_i^j$, and $d_i^j = d_i$. Thus,

$$p(ADL(x, f_0) = d_i) = p(ADL(x, f_c) = D_j) \cdot p(ADL(x, f_j) = d_i) \tag{13}$$

According to Lemma 1, $\forall \, \varepsilon_1 > 0$, and there is

$$\left|p\big(DPBC(x, f_c) = D_j\big) - w_j\right| < \varepsilon_1 \tag{14}$$

Per Lemma 2, $\forall\ \varepsilon_2 > 0$, and there is

$$\left|p\big(DPIC(x, f_j) = d_i^j\big) - w_i\right| < \varepsilon_2 \tag{15}$$

where w_j denotes the weight of D_j relative to D_0, and w_i represents the proportion accounted for by d_i in D_j. Then, the weight of d_i in D_0 is $w_j \cdot w_i$, so,

$$\left|p(ADL(x, f_0) = d_i) - w_j \cdot w_i\right|$$

$$= \left|p\big(DPBC(x, f_c) = D_j\big) \cdot p\big(DPIC(x, f_j) = d_i\big) - w_j \cdot w_i\right|$$

$$= \left| p\big(DPIC(x, f_j) = d_i\big) \cdot \big(p\big(DPBC(x, f_c) = D_j\big) - w_j\big) + w_j \cdot \big(p\big(DPIC(x, f_j) = d_i\big) - w_i\big)\right|$$

$$\leq \left|p\big(DPIC(x, f_j) = d_i\big) \cdot \big(p\big(DPBC(x, f_c) = D_j\big) - w_j\big)\right| + \left|w_j \cdot \big(p\big(DPIC(x, f_j) = d_i\big) - w_i\big)\right|$$

$$< p\big(DPIC(x, f_j) = d_i\big) \cdot \varepsilon_1 + w_j \cdot \varepsilon_2$$

Because $p\big(DPIC(x, f_j) = d_i\big) \in [0,1]$, and $w_j \in [0,1]$, according the arbitrariness of $\varepsilon_1 > 0$ and $\varepsilon_2 > 0$ and Definition 6, it can be seen that the ADL algorithm is fair.

4. Experimental and Results Analysis

The RCDL algorithm has considered the copy correlation between data during data layout and stores as many copies of the same data as possible on different storage devices to obtain higher redundancy. At the same time, RCDL fairly distributes data in accordance with the capacity of the storage device. But the computational overhead of the algorithm is high – it saves data through tables and matches storage device information and position data, which requires querying the match table, consuming large amount of time and memory space. In order to solve the heterogeneous storage environment, ADL adopts the clustering algorithm to classify storage devices to ensure that the capacities of the storage devices in same class are within the preset range. Then it uses the interval hash to distribute data between classes, and uses the consistent hash method within class. ADL has very good fairness and adaptability and by introducing a small number of virtual storage devices, it greatly reduces the storage space. ADL can use a predefined function to calculate data location, thus it does not need to look up table. But ADL does not consider the copy problem. If the same copies are in the same storage device, it will tend to reduce the redundancy. In order to

make up the shortcomings of RCDL and ADL, we combine RCDL and ADL and call it MCADL. The RCDL computational cost, query cost and storage cost are related to the data amount, so limiting the data amount can reduce these costs. At present, in a large number of data-intensive applications, the popularity of data shows high skewness. For example, the popularity of web objects presents the ZIPF distribution. Thus we divide the data into hot data and cold data according to the data access frequency, and use RCDL only for a small amount of hot data and use the ADL for cold data, in this way to solve the overhead problems of RCDL and also make up for the defect that ADL ignores data copies. In order to evaluate the performance of MCADL, we evaluate it in a simulated environment. The initial storage device set is configured as 20, and the capacities of four storage devices are 1000, those of three are 2000, those of three are 4000, those of four are 6000, those of three are 8000 and those of three are 9000. We send 400000 data, and replicate 80000 data 5 times. It is assumed that the data access obeys the ZIPF distribution. The data are divided into hot data and cold data. The data with the highest access frequencies are hot data, and the rest are cold data. The value of can be adjusted. Suppose $K = 2000$, then the 2000 more frequently accessed hot data are laid out by RCDL, and the rest are laid out by ADL. Based on the configuration, we set a series of experiments to evaluate various characteristics of MCADL, such as redundancy, fault tolerance and adaptability. All of the experiments are run on the 2.4 GHz intel dual-core machine.
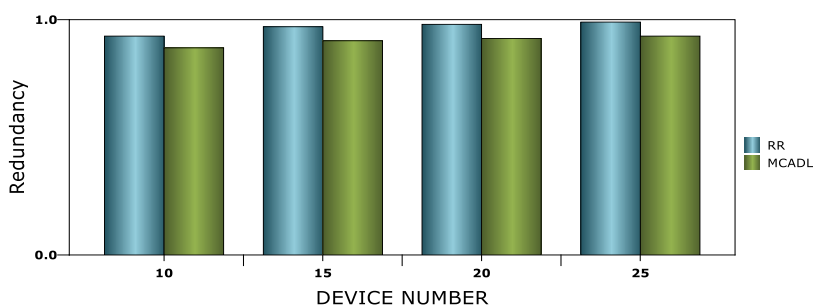

Fig.1. Redundancy of RR and MCADL.

4.1 Redundancy

Redundancy can reflect the reliability of a storage system. We compare the redundancy of three kinds of algorithms: (1) MCADL; (2) continuous layout CD; (3) random copy layout RR. Because the CD lays out copies on continuous storage devices, it does not have the same copies on the same storage device, and thus, it can obtain the highest redundancy. With CD as the reference standard, we define the redundancy factor of MCADL:

$$Redundancy_{MCADL} = \frac{Redundancy\ Degree\ of\ MCADL}{Redundancy\ Degree\ of\ CD}$$

The redundancy factor of RR is:

$$Redundancy_{RR} = \frac{Redundancy\ Degree\ of\ RR}{Redundancy\ Degree\ of\ CD}$$

Figure 1 shows the comparison of MCADL with RR. MCADL can obtain higher redundancy. At the same time, we find that the redundancy of MCADL is very close to the optimal value, and that with the number of storage devices increasing, its redundancy also increases. A large-scale network storage system includes hundreds of thousands of storage devices, so MCADL can be effectively used in a large-scale network storage system.

4.2 Fault Tolerance

We use the fault tolerance to describe the distribution of the copies from the failed storage device onto the other storage devices. When a storage device fails, and other storage devices store a copy of this storage device, the load on the failed storage device will be positioned to other storage devices. So fault-tolerance can reflect the storage load balancing during device failures.
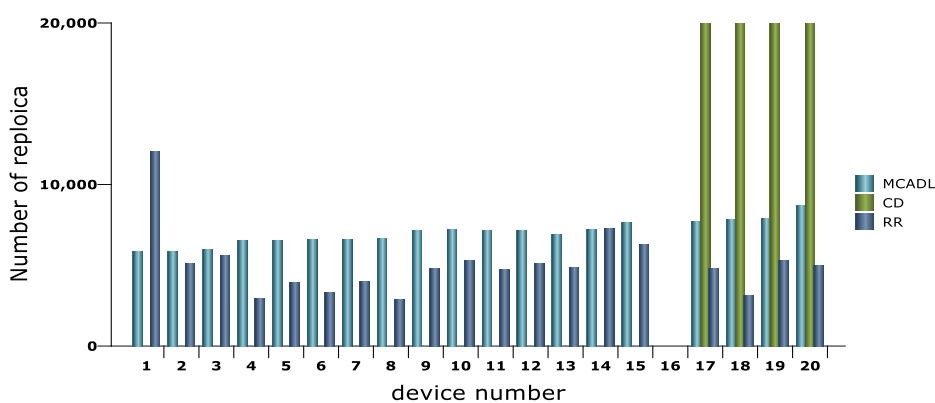

Fig.2. Copy Distribution during a Device Failure.

Figure 2 shows the copy distribution on other storage devices when the storage device 16 fails. From Figure 2, it can be seen that, MCADL can fairly distribute load from the storage device 16 to other storage devices. This figure also shows CD allocates the load on the storage device 16 to the

adjacent four storage devices. The fault-tolerance of RR is also displayed. The high column indicates that the corresponding storage device has more load from the failed storage device and the low column indicates the corresponding storage device has less load from the failed storage device.

From Figure 2, it can be seen that the fault tolerance of MCADL is far better than that of CD, and also better than that of RR.

4.3 Fairness

We send 100000, 200000, 300000, 400000 and 600000 data respectively to the storage devices to test the data distribution within class and between classes.

Figure 3 (a) is an inter-class data distribution map, where the x-axis indicates the different data amounts sent, and the y-axis represents the occupancy rate of each class, which is the ratio of the data amount assigned to the class to its weight. As can be seen from Figure 3 (a), when 100,000 data are sent, the occupancy rate of each class is about 10%, and data are fairly distributed among various classes. When 200,000, 300,000, 400,000, and 600,000 data are sent respectively, the occupancy rate of each class is almost the same. Due to the limitation of the data amount, the occupancy rate of each class are not completely identical, but very close to each other, Figure 3 (a) shows that the differences are very small, and in terms of statistical significance, the occupancy rate of each class is equal.

The next step is to test the fairness of data distribution within class. The experiment uses the second class as an example. When different data amounts are sent, the data amount that the second class acquires is respectively 6144, 12217, 18614, 24665 and 37081. The data are assigned to each storage device in lass 2 according to the intra-class layout mechanism.

Figure 3 (b) shows the occupancy rate of each storage device in lass 2. The occupancy rate of a storage device is equal to the ratio of the data amount assigned to the storage device to its weight. As can be seen from Fig. 3 (b), when 100,000 data are sent, the occupancy rate of each storage device in Class 2 is about 10%, but the deviations from 10% are larger than in the inter-class case, which is because the data amount is smaller.

When other different data amounts are sent, the occupancy rate of each storage device in Class 2 also fluctuates around the mean value. In terms of statistical significance, the occupancy rate of each storage device is equal. Figure 3 shows that the distribution of data on the storage devices is fair, and therefore the ADL algorithm is fair.
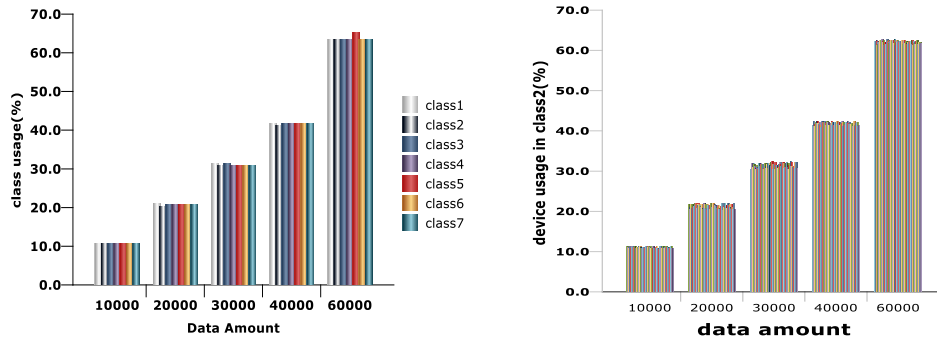
(a) Inter-class   (b) Intra-class.

Fig.3. Data Distribution.

Figure 4 shows the re-distribution of data when the storage system scale changes. Based on the above experiments, we still send 400,000 data as the precondition and adding three storage devices for consecutively four times. The inter-class data re-distribution is shown in Figure 4 (a), where the x-axis indicates the number of classes in the system, and the y-axis represents the occupancy rate of each class. With the increase of classes, the occupancy rate of each class becomes smaller.

From Figure 4 (a), it can be seen that,when a new class is added, the occupancy rate of each class is almost the same. After storage devices whose weight is 2000 are added or deleted successively, the re-distribution of data is shown in Figure 4 (b). The x-axis represents the number of storage devices in the class and the y-axis represents the occupancy rate of each storage device.
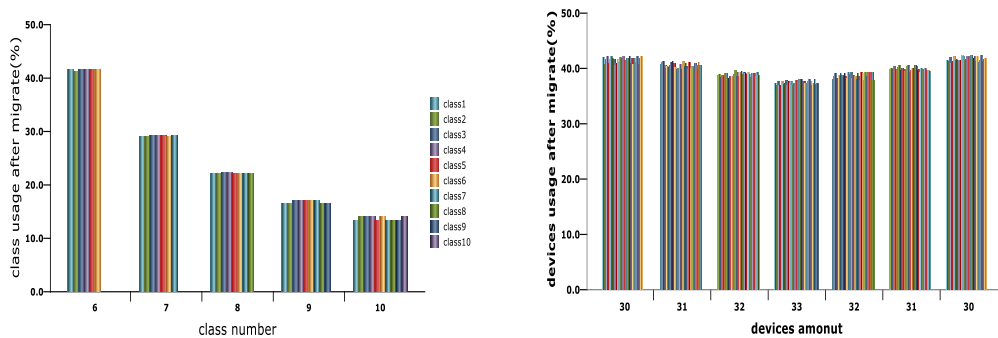
(a) After the addition of a class   (b) After the addition of delection of a device

Fig.4. Data Re-distribution

When storage devices are added successively the occupancy rate gradually becomes smaller; and when storage devices are deleted continuously, the occupancy rate gradually becomes larger. It can be seen from Figure 4 (b) that, when a single storage device is added or deleted, the

482

occupancy rate of each storage device is almost the same. Figure 4 shows that, after the storage scale changes, the ADL algorithm can satisfy the fairness of data distribution again.

4.4 Adaptivity

Next, we test the adaptability of the MCADL algorithm in three cases - adding a single class, adding and deleting a single storage device. Three storage devices are added for consecutively 4 times and in each batch the weight increase of storage devices are respectively 10000, 12000, 14000 and 16000. According to the clustering algorithm, each batch increases storage device points in a class. The experiment sends 400000 data as the precondition. Figure 5 (a) shows the data migration amount and theoretical migration amount after new classes are added. It can be seen that, with the new classes added, the migration data amount is very close to the theoretical value. Figure 5 (b) shows data migration amount and theoretical migration amount when storage devices (with a weight of 2000) are added and deleted (with a weight of 2000). It can be seen from figure 5 that the MCADL algorithm has good adaptability.
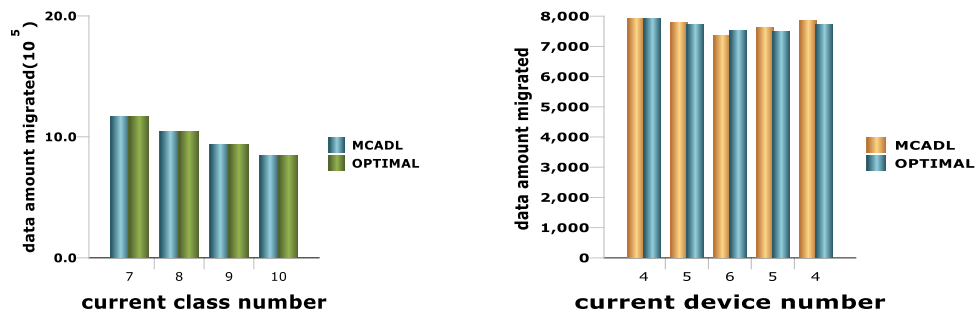
(a) Batch addition of devices   (b) Addition or delection of devices.

Fig.5. Adaptivity of the Algorithm When Devices Are Changed.

5. Summary

Currently, most of the layout algorithms only consider the distribution of a single copy, and the redundancy or fault tolerance of multi-copy layout algorithms is poor. In order to make up for the defects in multi-copy layout, this paper puts forward a multi-copy adaptive data layout algorithm (MCADL), combining the reliable copy layout algorithm (RCDL) and the effective adaptive data layout algorithm (ADL). RCDL aims at placing the same copies on different storage devices and at the same time, it satisfies fairness, increases redundancy and improves fault tolerance. It not only makes up the fault tolerance problem of the continuous copy layout, but also avoids the low redundancy of the random copy layout.

This paper formally defines the RCDL problem, and proves that any instance of the maximum n-separation problem can be converted to the RCDL problem, and thus the RCDL problem is NP-hard. Then this paper uses the semi-definite programming relaxation method to weaken the conditions of the RCDL problem, and by solving the semi-definite programming problem, it gets the approximate solution of the RCDL problem. In order to reduce the calculation cost, and time cost of RCDL, the proposed algorithm uses RCDL to lay out the hottest K data and uses ADL to lay out other data. ADL combines the clustering algorithm and the consistent hash algorithm. First, it clusters the storage device set by weight, which makes the weight differences between storage devices in the same class less than the pre-set value. Then according to the weights of classes, it assigns the sub-intervals in the interval [0, 1], and distributes the data mapped into the sub-intervals to the corresponding classes to ensure that the data are fairly assigned to classes, and that only the number of sub-intervals is related to the number of classes. The proposed algorithm uses the consistent hash method within each class to distribute data to storage devices within the class. As the weight differences between storage devices within the class are very small, t we do not need to introduce a large number of virtual storage devices, thus solving the problem of space waste brought by the simple improved consistent hash method. The amount of the migrated data when the storage scale changes is equal to the optimal data amount. The time of data positioning is only related to the number of classes. The theoretical and experimental analysis shows that, MCADL has higher redundancy and better fault tolerance, fairness and adaptability.

Acknowledgments

References

1. F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, Wide–area cooperative storage with CFS, 2001, In Proceedings of the 18th ACM Symposium on Operating Systems Principles, pp. 202–215.
2. A. Rowstron, P. Druschel, Storage management and caching in PAST, a large–scale, persistent peer–to–peer storage utility, 2001, In Proceedings of the 18th ACM Symposium on Operating Systems Principles.

3. S. Ghemawat, H. Gobioff, S.T. Leung, The google file system, 2003, In Proceedings of the 19th ACM Symposium on Operating Systems Principles, New York: ACM Press, pp. 19–22.

4. J. Kubiatowicz, D. Bindel, Y. Chen, OceanStore: An architecture for global–scale persistent storage, 2000, ASPLOS.

5. R. Renesse, Efficient reliable Internet storage, 2004, In Proceedings of the dependable distributed data management workshop.

6. R.J. Honicky, E.L. Miller, A fast algorithm for online placement and reorganization of replicated data, 2003, In Proceedings of the 17th International Parallel & Distributed Processing Symposium, pp. 57-66.

7. R.J. Honicky, E.L. Miller, Replication under scalable hash: a family of algorithms for scalable decentralized data distribution, 2004, In Proceedings of the 18th International Parallel and Distributed Processing Symposium. pp. 96-105.

8. S.A. Weil, S.A. Brandt, E.L. Miller, C. Maltzahn, CRUSH: controlled, scalable and decentralized placement of replicated data, 2006, In Proceedings of Super Computing.

9. A. Brinkmann, S. Effert, F.M. Heide, C. Scheideler, Dynamic and redundant data placement, 2007, In Proceedings of the 27th International Conference on Distributed Computing Systems. pp. 29-38.

10. S. Weil, A. Leung, S.A. Brandt, C. Maltzahn, RADOS: A fast, scalable, and reliable storage service for petabyte–scale storage clusters, 2007, In Proceedings of the ACM petascale data storage workshop.

11. A. Frieze, M. Jerrum, Improved approximation algorithms for MAX k–CUT and MAX BISECTION, 1997, Algorithmica.

12. A. Brinkmann, K. Salzwedel, C. Scheideler, Compact, adaptive placement schemes for non–uniform distribution requirements, 2002, In Proceedings. of the 14th ACM Symposium on Parallel Algorithms and Architectures, Winnipeg, Manitoba, Canada, pp. 53-62.

13. D. Karger, E. Lehman, T. Leighton, M. Levine, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, 1997, In Proceedings of the 29th ACM Symposium on Theory of Computing, pp. 654-663.