Hindawi
Mathematical Problems in Engineering
Volume 2019, Article ID 2617630, 10 pages
https://doi.org/10.1155/2019/2617630

Research Article

Data Placement Algorithm for Improving I/O Load Balance without Using Popularity Information

Xiangyu Luo,[1,2] **Gang Xin**,[3] **and Xiaolin Gui**[2]

[1]*College of Compute Science and Technology, Xi'an University of Science and Technology, Xi'an, Shaanxi Province 710054, China*
[2]*School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, Shaanxi Province 710049, China*
[3]*AVIC Computing Technique Research Institute, China*

Correspondence should be addressed to Xiangyu Luo; lxyseu@gmail.com

Received 27 September 2018; Accepted 24 December 2018; Published 17 January 2019

Academic Editor: Wlodzimierz Ogryczak

Data placement considerably affects the I/O performance of distributed storage systems such as HDFS. An ideal placement algorithm should keep the I/O load evenly distributed among different storage nodes. Most of the existing placement algorithms with I/O load balance guarantee depend on the information of data popularity to make the placement decisions. However, the popularity information is typically not available in the data placement phase. Furthermore, it usually varies during the data lifecycle. In this paper, we propose a new placement algorithm called Balanced Distribution for Each Age Group (BEAG), which makes data placement decisions in the absence of the popularity information. This algorithm maintains multiple counters for each storage node, with each counter representing the amount of data belonging to a certain age group. It ensures that the data in each age group are equally scattered among the different storage nodes. As the popularity variance of the data belonging to the same age group is considerably smaller than that of the entire data, BEAG significantly improves the I/O load balance. Experimental results show that compared to other popularity independent algorithms, BEAG decreases the I/O load standard deviation by 11.6% to 30.4%.

1. Introduction

In the big data era, distributed storage systems have attracted considerable attention [1–3]. In a distributed storage system, the data placement algorithm considerably affects the overall I/O performance. An ideal data placement algorithm should not only generate a balanced usage of disk space but also a balanced distribution of the I/O load among the different storage nodes.

The balanced usage of disk space is easy to achieve, but to guarantee a balanced distribution of the I/O load is extremely difficult. The I/O load caused by one data file is equal to the product of its size and popularity. Data popularity typically follows a skewed distribution such as Zipf. Hot files are frequently accessed, while cold ones are rarely accessed. Therefore, storage nodes assigned with more hot files are prone to be overloaded, while the other nodes may be idle, causing the entire system to be underutilized [4, 5]. Even worse, the popularity of each data file, measured by the average number of requests accessing the file per unit time, is typically not available in the data placement phase and dynamically changes during the entire data lifecycle.

To achieve the I/O load balance, many data placement algorithms for distributed storage systems have been proposed. According to whether the popularity information is taken as the necessary prerequisite for making placement decisions, the existing data placement algorithms can be divided into two categories: popularity-dependent algorithms and popularity-independent algorithms.

Popularity-dependent algorithms take data popularity information as the necessary prerequisite for making placement decisions. They mainly include two steps. First, the I/O load caused by each data file is calculated as the product of the data file size and its popularity. Second, an optimization algorithm is used to minimize the variance of the I/O load assigned to each storage node.

A popularity-independent algorithm makes data placement decisions without using any popularity information. A typical popularity-independent algorithm is the pseudo-random algorithm based on hash computations. It takes the

file identifier as the input of a hash function and takes the output of the function as the destination storage node. Such a placement algorithm is extremely easy to implement in engineering applications. A finely designed hash function ensures the balanced usage of disk space. However, its drawback is the poor level of I/O load balance as the distribution of data popularity is highly skewed. Moreover, even if the placement result was accidently desirable within a certain time period, it may be unsatisfactory in another time period as the data popularity changes dynamically.

The contributions of this paper include the following. Firstly, it presents the idea of making data placement decisions depending on the information of the data creation time instead of the data popularity. It takes advantage of the discipline existing in many applications in which the popularity variance of the data belonging to the same age group is considerably smaller than that of the entire data. Secondly, it proposes a practical algorithm realizing the abovementioned idea. The algorithm maintains multiple counters for each storage node, with each counter representing the amount of data created within a certain time period. It ensures that the data created within each time period are equally scattered among the different storage nodes. Thirdly, extensive experiments have been conducted showing that the proposed algorithm achieves 11.6% to 30.4% reduction of the standard deviation of the I/O load.

The rest of this paper is organized as follows. Section 2 summarizes the related work. In Section 3, we elaborate the BEAG algorithm. Section 4 presents the experimental results, and Section 5 concludes this paper.

2. Related Work

I/O load balance is an important method to improve the I/O performance of distributed storage systems [6]. The level of I/O load balance is considerably affected by the data placement algorithm. Designing a data placement algorithm with the ideal I/O load balance guarantee is very challenging, as data popularity follows an extremely skewed distribution.

Many data placement algorithms have been proposed to achieve the I/O load balance. According to whether the data popularity information is taken as the necessary prerequisite for making placement decisions, the existing data placement algorithms can be divided into two categories: popularity-dependent algorithms and popularity-independent algorithms.

The popularity-dependent algorithms can be further divided into two subclasses: static popularity-dependent algorithms and dynamic popularity-dependent algorithms. Static popularity-dependent algorithms assume that the popularity of each data file is known in advance and will never change in the future. Dynamic popularity-dependent algorithms assume that there is a popularity monitor that periodically collects the popularity information of each data file. The algorithms dynamically make data migration decisions according to the information provided by the popularity monitor, thereby improving the level of I/O load balance. However, their main drawback is the high overhead. On one hand, to figure out the optimal data migration plan requires complex computations. On the other hand, to execute the data migration plan causes additional I/O load competing for the scarce I/O bandwidth resources.

SP [7] is a static popularity-dependent algorithm. First, the average I/O load assigned for each storage node is calculated according to the size and the popularity of each data file. Second, the data files are sorted in the descending order of their sizes. Third, the data files are assigned to the storage nodes in the abovementioned order. The data files will not be assigned to a new storage node until the current storage node is assigned with its I/O load reaching the average value calculated in advance.

CDRM [8] is a dynamic popularity-dependent algorithm. It continuously monitors the data popularity variance and updates the blocking probability on each storage node. On the basis of the blocking probability, it dynamically replicates data on the idle storage nodes to improve the level of I/O load balance.

Kari et al. [9] also proposed a dynamic popularity-dependent algorithm that migrates data from one node to another for load balance or in the event of a system failure or upgrades. The main contribution is the maximization of the number of simultaneous transfers to minimize the data migration time. Further, Sha et al. [10] proposed a dynamic popularity-dependent algorithm for mapreduce applications.

DRS [11] is a dynamic popularity-dependent algorithm based on Markov model. It constructs a transition probability matrix based on the file accessing times in a period and calculates the stationary probability distribution of the system. It utilizes the results to distinguish different data types and then increases extra replicas for hot data, cleaning up these extra replicas when the data cool down.

Kinesis [12] is a popularity-independent algorithm. It adopts a hash-based replica placement strategy. It devises k independent hash functions and generates k candidate storage nodes for each data file. Among the k storage nodes, only r ($r < k$) nodes will be finally selected to store a replica of the file.

Xie and Chen [13] also proposed a hash-based popularity-independent algorithm. It offers an elastic distributed storage system with power proportionality. When the system load decreases, some of the storage nodes will be powered off to reduce energy consumption.

Wang et al. [14] proposed a placement strategy dealing with the data placement problem among multiple data centers.

3. BEAG Placement Algorithm

3.1. Main Idea. Data popularity directly reflects the pressure on the I/O load. However, its value is typically not available in the data placement phase. Therefore, we need another variable whose value is easily obtainable and can reflect data popularity, perhaps in an indirect way. The age of data is a good alternative. It represents how long the data has been created. On one hand, the age of the data is extremely easy to obtain. To calculate the age, the only required information is

the creation time of the data, which has already been recorded in the existing storage systems. On the other hand, the age of the data is correlated with the data popularity in statistics. The popularity variance of the data belonging to the same age group is considerably smaller than that of the entire dataset.

The main idea of the BEAG algorithm can be stated as follows. The algorithm calculates the age of each data file according to its creation time. All of the data are then classified into multiple groups according to their age. The algorithm ensures not only that all of the data are equally scattered among the different storage nodes but also that the data in each age group are equally scattered, preventing that some nodes store more new data while other nodes store more old data. As the popularity variance of each age group is considerably smaller than that of the entire dataset, the algorithm can obtain a better I/O load balance than the pseudorandom algorithms.

3.2. Elaboration of the Algorithm. To equally distribute the data in each age group among the different storage nodes, the algorithm should maintain an array of counters for each storage node. Each counter represents the amount of data allocated to the storage node within a certain age group. The algorithm keeps the counter arrays for different storage nodes approximately the same all the time.

The BEAG placement algorithm mainly contains three subalgorithms. The first one is called the initialization subalgorithm. It is responsible for the initialization of the counter arrays and the placement of the initial set of data files. The second one is called the in-progress subalgorithm. It keeps running all the time after the completion of the initialization subalgorithm. It is responsible for handling all the possible events that can change the data placement. Such events include the creation of a new file, deletion of an existing file, joining of a new node, and the exit of an existing node. The last one is called the self-refreshing subalgorithm. It also keeps running all the time after the completion of the initialization subalgorithm, but it will not change the data placement results while changing only the values of the counter arrays. With the passage of time, the age group that a data file belongs to changes. Therefore, the counter arrays should be updated in time.

We focus on distributed storage systems that use the architecture adopted by HDFS [15], that is, the master-slave architecture. There are two types of nodes in the system: a name node and multiple data nodes. The name node, also called the metadata node, is responsible for making the data placement decisions and maintaining the mapping of data files and data nodes. A data node, also called a storage node, is responsible for storing data and handling access requests. The proposed placement algorithm, BEAG, runs on the name node.

3.2.1. Initialization Subalgorithm. The initialization subalgorithm works in two stages. In the first stage, it initializes the counter arrays. In the second stage, it determines the placement of the initial set of data files.

To initialize the counter array for each storage node, the initialization subalgorithm needs to determine the number of counters contained in each counter array as well as the age group that each counter corresponds to. Let N denote the number of counters contained in each counter array; then, the data files are divided into N age groups. To describe the N age groups, we just require $(N-1)$ positive numbers as the division points. Let $A = \{a_i \mid 1 \le i \le N-1\}$ denote the set of division points. The first age group contains the data whose age is not greater than a_1. The last age group contains data older than a_{N-1}. The jth $(1 < j < N)$ age group contains the data whose age is between a_{j-1} and a_j. We call the value $(a_j - a_{j-1})$ the age span of the jth age group. Note that the age span of the first age group is equal to a_1 and that of the last age group is positive infinity. Moreover, we take "day" rather than "year" as the measurement unit for the age of the data. As the data age greater than a_{N-1} is not differentiated from the point of view of the age group, we call a_{N-1} the maximum differentiable age.

To determine the $(N-1)$ division points, we choose an exponential function; that is, $a_i = 2^i$. The first age group only contains the data files created no more than two days back, the second age group contains the data files created more than two days but not more than four days back, the third age group contains the data files created more than four days but not more than eight days back, and so on.

The above method for age group division has two advantages. First, only a small N is required to obtain the ideal maximum differentiable age. For example, $N = 20$ means that the maximum differentiable age is 2^{19} days, that is, more than 1,436 years. For realistic storage systems, the data lifecycle is only tens or hundreds of years long. Therefore, irrespective of the type of storage systems, we can always divide the data into 20 age groups. Second, the popularity variance is very small for each age group. Although the age span increases exponentially with an increase in the number of groups, the popularity variance does not increase. As the popularities of old data usually fall in the range between zero and a small positive number, the popularity variance is not very large.

After determining the number of counters in each counter array and the age group that each counter corresponds to, the initialization subalgorithm assigns each counter with zero for all the storage nodes. The next step is to solve the placement of the initial set of data files. For each data file, the algorithm calculates its age, that is, the difference between the current time and the file's creation time. Note that the initial files can be created at different times, as they may be stored earlier in other devices or systems. According to the age of the file, the algorithm determines which age group it belongs to. The files are then grouped into N subsets on the base of age. We independently distribute each subset. For each subset, each storage node maintains a counter to record the amount of data assigned to it. Files in the subset are distributed one by one. Every time a file in the subset is distributed to the storage node with the minimum counter. Eventually, files in each subset (or age group) are approximately equally distributed to different storage nodes.

```
Data: Data file set F; storage node set S.
Result: Mapping of the data file set F to the storage node set S; counter arrays C.
Initialize each element C[i][j] with 0, each element A[k] with 2^k, and each element F_k' with ∅
while there is an unhandled file f in F do
    Query the file's creation time t_c
    Calculate the file's age t_a
    k = 1
    while k is between 1 and N − 1 do
        if t_a < A[k] then
            break;
        end
        k++
    end
    F_k = F_k ∪ {f}
End
while k' is between 1 and N do
    while there is an unhandled file f in F_k' do
        min := C[1][k']
        sn = 1
        i = 2
        while i is between 2 and M do
            if C[i][k'] < min then
                sn := i
                min = C[sn][k']
            end
            i++
        end
        Place the file f on the storage node sn
        C[sn][k'] := C[sn][k'] + Size(f)
    end
End
```

Algorithm 1: Initialization subalgorithm.

The initialization subalgorithm can be described as Algorithm 1. The symbol M represents the number of storage nodes in the system. N represents the number of counters in each counter array. As discussed above, 20 is an acceptable value for N. C is a $M \times N$ matrix. The element $C[i][j]$ represents the value of the jth counter for the ith storage node. A represents the set of division points. F represents the initial set of data files, S represents the set of storage nodes, and M represents the number of storage nodes in S.

3.2.2. In-Progress Subalgorithm. After the initialization, the storage system may exist for a long time such as tens of years. During its lifetime, new data placement decisions are required to be made under the following four circumstances. Firstly, a new file is created into the system. Secondly, an existing data file is required to be deleted from the system. Thirdly, a new node joins into the system. Finally, an existing node leaves from the system. The in-progress subalgorithm mainly handles the abovementioned four types of events. It contains four reactors with each one handling a different type of event. Whenever an event takes place, the corresponding reactor is activated.

The file-creation reactor (Algorithm 2) is used to handle the event of the creation of a new file. The age of a new file is equal to zero, which falls in the range of the first age group. Firstly, the reactor checks the first counter in each counter array to find out the minimum one. Secondly, the reactor places the file into the storage node with the minimum first counter. Finally, the counter is increased by the size of the file.

```
Data: Newly created file f
Result: Destination storage node for f
Initialize min with C[1][1], sn with 1 and i with 2
while i is between 2 and M do
    if C[i][1] < min then
        sn := i
        min = C[sn][1]
    end
end
Place the file f on the storage node sn
C[sn][1] := C[sn][1] + Size(f)
```

Algorithm 2: File-creation reactor.

The file-deletion reactor (Algorithm 3) is used to handle the event of the deletion of a file. Firstly, the reactor queries the creation time of the file and calculates its age. Secondly, the reactor checks which age group the file belongs to and

```
Data: File f to delete
Result: Renewal of the counter array
Initialize k with 1
Query the file's creation time t_c
Calculate the file's age t_a
while k is between 1 and N − 1 do
    if t_a < A[k] then
        break;
    end
    k++
end
Query the holding storage node sn
Delete f from the storage node sn
C[sn][k] := C[sn][k] − Size(f)
```

Algorithm 3: File-deletion reactor.

which storage node the file is placed on. Thirdly, the reactor deletes the file from the storage node and decreases the corresponding counter for the storage node by the size of the file.

The node-joining reactor (Algorithm 4) is used to handle the event of a new node joining into the storage system. The system has to assign an equal I/O load to the new node as that assigned to the other nodes. For each age group, the reactor first calculates the total amount of the corresponding counters, the average amount before the new node joining, and the average amount after the new node joining. The difference between the average before and the average after is the amount of data required to migrate from each existing node to the new node. Next, the reactor migrates the required amount of data belonging to the age group from each existing node to the new node. The corresponding counter for each existing node is decreased, while that of the new node is increased by the amount of the migrated data.

The node-leaving reactor (Algorithm 5) is used to handle the exit of a node. Without any loss of generality, suppose that the node numbered M leaves from the system. For each age group, the reactor first calculates the average amount of data required to migrate to each left nodes and divides the files in the age group into approximately equal $(M - 1)$ subsets. Then, the reactor migrates each subset to each left node and increases the corresponding counter by the amount of the migrated data.

3.2.3. Self-Refreshing Subalgorithm. With the passage of time, the age of each file increases and the age group that it belongs to changes. Therefore, the counter arrays need to be updated in time. Otherwise, the in-progress algorithm will make incorrect decisions.

As we take "day" as the unit for the data age, the data age of each file has to be updated every day, and thus, the self-refreshing algorithm is triggered every day. Suppose that the files are sorted in the descending order of the creation time, thus the ascending order of the data age. It is noted that the files are required to be ordered only once. With the passage of time, all the files will become older and their orders will remain the same. Suppose that k_j $(1 \le j \le N - 1)$ represents the largest order number of the file belonging to the jth age group. We know that the age of the k_jth file is not greater than a_j and that of the (k_j+1)th file is greater than a_j. Once the self-refreshing algorithm is triggered, the most important task is to update the values of each k_j. Once the value of each k_j is determined, the value of each counter $C[i][j]$ can be deduced, that is, the total number of files stored in the node i with the global order no bigger than k_j.

The self-refreshing algorithm can be described as Algorithm 6: The symbol F represents the set of files stored in the storage system.

3.3. Complexity Analysis. First, the time complexity of the initialization subalgorithm is $O(|F|(|S| + N))$. $|F|$ represents the number of the files and $|S|$ represents the number of the storage nodes. Since N, that is, the number of the age groups, is typically not greater than 20, the time complexity of the initialization subalgorithm can be written as $O(|F| \cdot |S|)$. Second, the time complexity of the file creation process is $O(|S|)$ and that of the file deletion is $O(1)$. Third, both the time complexity of the node joining process and that of the node leaving process are $O(|S|)$. Finally, the time complexity of the self-refreshing subalgorithm is $O(|F|)$. Therefore, BEAG is a light-weight solution to the data placement problem.

4. Evaluation

We implemented the BEAG data placement algorithm and compared it with both the pseudorandom algorithm and Kinesis [12]. We chose the standard deviation of the I/O load assigned to each storage node as the criterion for the performance evaluation. Let M denote the number of storage nodes and L_i $(1 \le i \le M)$ denote the I/O load assigned to the ith storage node. The standard deviation of the I/O load assigned to each storage node is denoted by $\delta(L)$ and calculated as follows: $\delta(L) = \sqrt{\sum(L_i - \overline{L})^2/M}$. In this expression, $\overline{L} = \sum L_i/M$. The I/O load assigned to the ith storage node L_i is equal to the sum of the I/O load generated by each data file placed onto the storage node, and the I/O load generated by each data file is equal to the product of its size and popularity. Note that all of the three algorithms, that is, BEAG, Kinesis, and the pseudorandom algorithm, make placement decisions without using any popularity information, and the popularity is only used to evaluate the performance afterwards. Here a smaller $\delta(L)$ means a higher level of I/O load balance.

The method of performance evaluation described above requires convincing I/O workload generators. We used two types of I/O workload generators. One is called Reproducer, which reproduces the file creation and access process on the basis of the metadata collected from a real-world application (i.e., the blog system of ScienceNet.cn). The other is called Medisyn [16] designed by Hewlett Packard Labs, which relies on a simulation model to describe the process of file creation and access in video-on-demand applications.

The evaluation mainly includes four steps. First, we use the I/O workload generator to get each file's age, size,

```
Data: New storage node
Result: Data migration; renewal of the counter array
Initialize j with 1
M := M + 1
while jis between 1 andN do
    C[M][j] := 0
    sum := 0
    i = 1
    while iis between 1 andM − 1 do
        sum := sum + C[i][j]
        i + +
    end
    avg_old = sum/(M − 1)
    avg_new = sum/M
    delta = avg_old − avg_new
    i := 1
    while iis between 1 andM − 1 do
        select a subset of files subset_ij from the ith
        storage node belonging to the jth age group with
        the total size of the files in subset_ij
        approximately equal to delta;
        migrate the files in subset_ij from the ith storage
        node to the new storage node;
        C[i][j] := C[i][j] − Size(subset_ij)
        C[M][j] := C[M][j] + Size(subset_ij)
        i + +
    end
    j + +
end
```

Algorithm 4: Node-joining reactor.

```
Data: Leaving storage node
Result: Data migration; renewal of the counter array
Initialize i with 1
while iis between 1 andN do
    avg := C[M][i]/(M − 1)
    Divide the files belonging to the ith age group on the
    Mth node into (M − 1) subsets, with the total size of
    the files in each subset approximately equal to avg;
    j := 1
    while jis between 1 andM − 1 do
        migrate the jth subset from the Mth node to the
        jth node;
        C[i][j] := C[i][j] + Size(subset_j)
        j + +
    end
    i + +
end
M := M − 1
```

Algorithm 5: Node-leaving reactor.

and popularity. Second, we, respectively, employ the three algorithms to map the files into the storage nodes. Third, we calculate the I/O load assigned to each storage node by summing up the I/O load generated by each file placed on it. Finally, we evaluate the performance through the standard deviation of the I/O load assigned to each storage node.

4.1. Evaluation on Real-World Datasets. The I/O workload generator Reproducer generates a number of tuples in the

Data: Counter arrays and each file's age
Result: Renewal of the counter arrays
Increase the age of each file by one day
Set each element $C[i][j]$ with 0
$i := 1$
$j := 1$
while j*is between* 1 *and* $N - 1$ **do**
  while $i < |F|$ **do**
    if *the age of the ith file is larger than* a_j **then**
      break
    end
    i++
  end
  $k_j = i - 1$
  if $j == 1$ **then**
    $k_{j-1} = 0$
  end
  $k = k_{j-1} + 1$
  while i*is between* $k_{j-1} + 1$ *to* k_j **do**
    Query the number sn of the node that holds the ith file f_i
    $C[sn][j] = C[sn][j] + Size(f_i)$
    $i++$
  end
End

Algorithm 6: Self-refreshing subalgorithm.

form of $\langle t_{creation}, p_{current}, s\rangle$. In the tuple, $t_{creation}$ represents the file's creation time, $p_{current}$ denotes the file's current popularity, and s represents the size of the file. The file's creation time is recorded in the system and can be directly obtained. The file's current popularity is expressed with the number of the access requests on the observed day. Once again, note that $p_{current}$ is only used for the performance evaluation, and all of the three algorithms make placement decisions without using any popularity information. In the experiment, we developed a web crawler to collect the publication time and download times of 4,301 articles from the blog system of ScienceNet.cn. By analyzing their publication time, we found that the articles' ages fell in the range between one day and 2,865 days. Therefore, we divided the files into 12 age groups, with the ith group containing the files ranging from 2^{i-1} to 2^i days in age.

Firstly, we analyzed the data popularity distribution. The aim was to compare the popularity variance of the data in the same age group and that of the entire dataset.

Secondly, we compared the proposed algorithm with both Kinesis and the pseudorandom algorithm. We adopted $\delta(L)$, that is, the standard deviation of the I/O load assigned to each storage node, as the metric for the I/O load balance.

4.1.1. Popularity Distribution Analysis. We used the standard variance for describing the variance of data popularity. For the data files as a whole, the standard variance of data popularity was calculated as $\delta(p_{current}) = \sqrt{\sum(p_{current}(f_j) - \overline{p_{current}})^2/|S|}$. Here, $p_{current}(f_j)$ represents the current popularity of the file f_j, $\overline{p_{current}}$ represents the average current popularity of the files in F, and $|F|$ represents the number of files contained in the file set. For the data files divided into multiple age groups, we first computed the standard variance of data popularity in each age group. Then, we obtained the expectation of the standard variance of the data popularity by using the following formula: $\overline{\delta(p_{current})} = \sum(\delta_i(p_{current})|F_i|)/|F|$. Here, $\delta_i(p_{current})$ represents the standard variance of the data popularity in the ith age group and $|F_i|$ represents the number of files contained in the age group.

Through the analysis of the data generated by the I/O workload generator Reproducer, we obtained $\delta(p_{current})$, that is, the standard deviation of the popularity for the files as a whole, was equal to 60.6, and $\overline{\delta(p_{current})}$, that is, the expectation of the standard deviation of the popularity for the files divided into multiple age groups, was equal to 10.3. The above results validated our assumption that the popularity variance of the data in the same age group was considerably smaller than that in the entire data fileset.

For a further comparison of the popularity diversity in the entire file set and that in each age group, we computed the ratio of the maximal popularity to the minimal popularity. A larger ratio means a higher level of diversity. The results are listed in Table 1. In the table, F_i represents the file subset composed of files belonging to the ith age group and F represents the entire file set. This shows that the diversity of the data popularity in each age group is considerably smaller than that in the entire file set.

4.1.2. Performance Comparison. In the experiment, we assumed a storage system composed of 10 storage nodes.

Table 1: Comparison of popularity diversity in each file set.

File set	$\mathbf{Max}(p_{current})/\mathbf{Min}(p_{current})$
F_1	92.7
F_2	91.8
F_3	82.0
F_4	47.0
F_5	46.0
F_6	35.5
F_7	21.0
F_8	28.5
F_9	96.5
F_{10}	30.5
F_{11}	16.0
F_{12}	10.5
F	1437.5

Table 2: Comparison of three algorithms (Ten Storage Nodes).

Algorithm	$Load_{max}/Load_{min}$	$\delta(L)$
Pseudorandom	15230.7/4253.6	3993.9
Kinesis	16533.0/4467.0	3592.8
BEAG	13343.0/4541.4	2778.7

We, respectively, used the pseudorandom algorithm, Kinesis, and BEAG to place the files onto the storage nodes. The comparison results of the three algorithms are shown in Table 2. $Load_{max}$ represents the amount of the I/O load on the most loaded storage node, while $Load_{min}$ represents the amount of the I/O load on the least loaded storage node. The unit of I/O load is KB/s.

The BEAG algorithm yielded the best I/O load balance, with the standard deviation of I/O load assigned to each storage node $\delta(L)$ decreasing by 30.4% and 22.7%, respectively, compared with the pseudorandom algorithm and Kinesis.

4.2. Evaluation on Synthetic Datasets. The I/O workload generator Medisyn can also generate a number of tuples in the form of $\langle t_{creation}, p_{current}, s\rangle$. It does not recur in any realistic systems, while depending on a stochastic model to simulate the process of file creation and access. It was demonstrated that the model could correctly describe the I/O workload characteristics for video-on-demand applications. In this model, the file creation process obeys the Poisson distribution, the popularity among different files obeys the Zipf distribution, the popularity evolution process for each file obeys the log-normal distribution, and the size of the file also obeys the Zipf distribution. The Poisson distribution is described with the parameter λ, the Zipf distribution is described with the parameter α, and the log-normal distribution is described with two parameters μ and δ. Moreover, we need two parameters $|F|$ and L. $|F|$ represents the total number of files, and L represents the total I/O load.

4.2.1. Popularity Distribution Analysis. With λ=0.5, α=0.8, $|F|$=45,000, T=180,000,000, μ=3, and δ=3, we generated 45,000 tuples with the form $\langle t_{creation}, p_{current}, s\rangle$. By analyzing the data age, we found that the ages fell in the range of 0 to 2499 days. Therefore, we divided the files into 12 age groups. For the entire dataset, the standard variance of the data popularity was calculated as $\delta(p_{current}) = \sqrt{\sum(p_{current}(f_j) - \overline{p_{current}})^2/|S|}$. For the data files divided into multiple age groups, we first computed the standard variance of the data popularity in each age group. Then, we obtained the expectation of the standard variance of the data popularity through the formulation $\overline{\delta(p_{current})} = \sum(\delta_i(p_{current})|F_i|)/|F|$.

Through the analysis of the synthetic data, we obtained $\delta(p_{current})$, that is, the standard deviation of the popularity for the files as a whole, was equal to 27.0, and $\overline{\delta(p_{current})}$, that is, the expectation of the standard deviation of the popularity for the files divided into multiple age groups, was equal to 4.6. The above results also validated our assumption that the popularity variance of the data in the same age group was considerably smaller than that in the entire data file set.

Once again, for a further comparison of the popularity diversity in the entire file set and that in each age group, we computed the ratio of the maximal popularity to the minimal popularity; the results are listed in Table 3. In the table, F_i represents the file subset composed of the files belonging to the ith age group, and F represents the entire file set. The table shows that the diversity of the data popularity in each age group was also considerably smaller than that in the entire file set.

Table 3: Comparison of popularity diversity in each file set.

File set	$\mathbf{Max}(p_{current})/\mathbf{Min}(p_{current})$
F_1	49.7
F_2	13.1
F_3	26.2
F_4	56.8
F_5	57.7
F_6	601.2
F_7	106.4
F_8	185.1
F_9	310.7
F_{10}	462.3
F_{11}	169.7
F_{12}	332.2
F	113585.8

4.2.2. Performance Comparison. In the experiment, we assumed a storage system composed of 100 storage nodes. We, respectively, used the pseudorandom algorithm, Kinesis, and BEAG to place the 45,000 files onto the storage nodes. The comparison results of the three algorithms are shown in Table 4. The unit of I/O load is KB/s.

The BEAG algorithm yielded the highest level of I/O load balance, with the standard deviation of the I/O load assigned

Table 4: Comparison of three algorithms (100 Storage Nodes).

Algorithm	$Load_{max}/Load_{min}$	$\delta(L)$
Pseudorandom	16266.8/498.0	1610.1
Kinesis	10572.3/647.6	1594.9
BEAG	10174.7/1013.7	1410.0

to each storage node $\delta(L)$ decreasing by 12.4% and 11.6%, respectively, compared with the pseudorandom algorithm and Kinesis.

5. Conclusion

Most traditional data placement algorithms for distributed storage systems depend on the information of data popularity for making placement decisions to realize the I/O load balance. However, data popularity is usually unknown in the data placement phase and changes dynamically during the data lifecycle. We proposed a new data placement algorithm without using any popularity information. The algorithm makes use of the correlation between a file's creation time and its popularity. It ensures that the data created in each time period are evenly scattered among different storage nodes. Compared to other popularity independent algorithms, the proposed algorithm guarantees a higher level of load balance, with the I/O load standard deviation decreasing by 11.6% to 30.4%. However, the algorithm only considers homogenous storage systems. In the future, we plan to extend it to heterogeneous environments.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (Grant No. 61702408), Natural Science Basic Research Plan in Shaanxi Province of China (Program No. 2017JQ6053), the National Natural Science Foundation of China (Grant No. 61472316), and the Innovation Group for Interdisciplinary Computing Technologies in Xi'an University of Science and Technology.

References

[1] C. Liu, J. Chen, L. T. Yang et al., "Authorized public auditing of dynamic big data storage on cloud with efficient verifiable fine-grained updates," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2234–2244, 2014.

[2] H. Cai, B. Xu, L. Jiang, and A. V. Vasilakos, "IoT-Based Big Data Storage Systems in Cloud Computing: Perspectives and Challenges," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 75–87, 2017.

[3] M. W.-U. Rahman, N. S. Islam, X. Lu, and D. K. D. Panda, "A Comprehensive Study of MapReduce over Lustre for Intermediate Data Placement and Shuffle Strategies on HPC Clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 633–646, 2017.

[4] G. Ananthanarayanan, S. Agarwal, S. Kandula et al., "Scarlett: Coping with skewed content popularity in MapReduce clusters," in *Proceedings of the 6th ACM EuroSys Conference on Computer Systems, EuroSys 2011*, pp. 287–300, Austria, April 2011.

[5] C. Hsu, V. W. Freeh, and F. Villanustre, "Trilogy: Data placement to improve performance and robustness of cloud computing," in *Proceedings of the 2017 IEEE International Conference on Big Data (Big Data)*, pp. 2442–2451, Boston, MA, USA, December 2017.

[6] W. Dai, I. Ibrahim, and M. Bassiouni, "An Improved Replica Placement Policy for Hadoop Distributed File System Running on Cloud Platforms," in *Proceedings of the IEEE International Conference on Cyber Security and Cloud Computing*, pp. 270–275, June 2017.

[7] L.-W. Lee, P. Scheuermann, and R. Vingralek, "File assignment in parallel I/O systems with minimal variance of service time," *IEEE Transactions on Computers*, vol. 49, no. 2, pp. 127–140, 2000.

[8] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng, "CDRM: A cost-effective dynamic replication management scheme for cloud storage cluster," in *Proceedings of the IEEE International Conference on Cluster Computing*, pp. 188–196, 2010.

[9] C. Kari, Y. Kim, and A. Russell, "Data Migration in Heterogeneous Storage Systems," in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS)*, pp. 143–150, Minneapolis, MN, USA, June 2011.

[10] E. H.-M. Sha, Y. Liang, W. Jiang, X. Chen, and Q. Zhuge, "Optimizing data placement of mapreduce on ceph-based framework under load-balancing constraint," in *Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems, ICPADS 2016*, pp. 585–592, China, December 2016.

[11] K. Qu, L. Meng, and Y. Yang, "A dynamic replica strategy based on Markov model for hadoop distributed file system (HDFS)," in *Proceedings of the 4th IEEE International Conference on Cloud Computing and Intelligence Systems, CCIS 2016*, pp. 337–342, August 2016.

[12] J. MacCormick, N. Murphy, V. Ramasubramanian, U. Wieder, J. Yang, and L. Zhou, "Kinesis: A new approach to replica placement in distributed storage systems," *ACM Transactions on Storage (TOS)*, vol. 4, article 11, no. 4, 2009.

[13] W. Xie and Y. Chen, "Elastic Consistent Hashing for Distributed Storage Systems," in *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017*, pp. 876–885, June 2017.

[14] T. Wang, S. Yao, Z. Xu, and S. Jia, "DCCP: an effective data placement strategy for data-intensive computations in distributed cloud computing systems," *The Journal of Supercomputing*, vol. 72, no. 7, pp. 2537–2564, 2016.

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST '10)*, pp. 1–10, Piscataway, NJ, USA, May 2010.

[16] W. Tang, Y. Fu, L. Cherkasova, and A. Vahdat, "MediSyn: A synthetic streaming media service workload generator," in *Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp. 12–21, USA, June 2003.