RDIM: A Self-adaptive and Balanced Distribution for Replicated Data in Scalable Storage Clusters*

Zhong Liu, Nong Xiao, and Xing-Ming Zhou

Institute of Computer, National University of Defense Technology,
Changsha, China, 410073
Liuzhong@zhmail.com

Abstract. As storage systems scale from a few storage nodes to hundreds or thousands, data distribution and load balancing become increasingly important. We present a novel decentralized algorithm, RDIM (Replication Under Dynamic Interval Mapping), which maps replicated objects to a scalable collection of storage nodes. RDIM distributes objects to nodes evenly, redistributing as few objects as possible when new nodes are added or existing nodes are removed to preserve this balanced distribution. It supports weighted allocation and guarantees that replicas of a particular object are not placed on the same node. Its time complexity and storage requirements compare favorably with known methods.

1 Introduction

As the use of large distributed systems and large-scale clusters of commodity computers has increased, significant research has been devoted toward designing scalable distributed storage systems. Its applications now span numerous disciplines, such as: higher large-scale mail system, online numeric periodical, digital libraries, large online electric commerce system, energy research and simulation, high energy physics research, seismic data analysis, large scale signal and image processing applications, data grid application and peer-to-peer storage application, etc. Usually, it will no longer be possible to do overall upgrades of high performance storage systems. Instead, systems must grow gracefully over time, adding new capacity and replacing failed units seamlessly—an individual storage device may only last five years, but the system and the data on it must survive for decades. Since the capacities of storage nodes usually are non-uniform and storage nodes are dynamically changed in large-scale distributed storage systems, systems must distribute data objects among the storage nodes according to their capabilities and afford to immediately rebalance data objects distribution according to weight of storage nodes when storage nodes are changed. So we study the problem of designing flexible, adaptive strategies for the distribution of objects among a heterogeneous set of servers. Ideally, such a strategy should be able to adapt with a minimum amount of replacements of objects to changes in the capabilities of the servers so that objects are always distributed among

* Supported by the National Basic Research Program 973 of China (No.2003CB317008).

Y. Pan et al. (Eds.): ISPA 2005, LNCS 3758, pp. 21 – 32, 2005.

the servers according to their capabilities. Finally, Xin, et al.[1] reports that the mean time to failure (of a single disk) in a petabyte-scale (10^{15} bytes) storage system will be approximately one day. In order to prevent data loss, we must allow for data replication. Furthermore, the data replication scheme should guarantee that replicas of the same object get placed on different servers, or the effect of replication will be nullified.

Previous techniques are able to handle these requirements only in part. For example, a typical method to map data object to storage nodes in an optimally balanced way is a simple Round-Robin (RR) assignment. The storage node number assigned to a given data object can be easily calculated using modular arithmetic: h(id)=id mod n, where id is object ID and n is the number of storage nodes in system. If storage nodes have the uniform capabilities, it can be used to distribute data objects evenly among n servers. However, they usually do not adapt well to a change in the capabilities. Moreover, If a new server is added, approximately the fraction n/(n+1) of the data objects must be moved from one storage node to another before the data can be accessed using the new mapping. For a large storage system, this leads to a long period of unavailability of data, which is not acceptable to many applications. In contrast, the minimum fraction that must be relocated to obtain a balanced mapping is approximately 1/(n+1). A different approach is to maintain object-to-node mapping in a stored directory (SD). In this case, a directory of B entries is maintained in which the ith entry contains the node number assigned to object i, where B is the total number of objects and is usually a fairly large integer. Thus, each object can be individually assigned or reassigned to any storage node. When new storage nodes are added, individual objects are selected for relocation to the new nodes so that only the minimum amount of object is moved. However, this approach suffers from severe performance bottleneck problems and consumes a significant amount of memory. Litwin, et al. [2] has developed many variations on Linear Hashing (LH*), the LH* variants are limited in two ways: they must split buckets, and they have no provision for buckets with different weights. LH* splits buckets in half, so that on average, half of the objects on a split bucket will be moved to a new empty bucket, resulting in suboptimal bucket utilization and a “hot spot” of bucket and network activity between the splitting node and the recipient and the distribution is unbalanced after replacement. Moreover, the LH* variants do not support weighted allocation and data replication. Other data structures such as DDH [3] suffer from similar splitting issues. Choy, et al. [4] describes algorithms for perfect distribution of data to disks that move an optimally low number of objects when disks are added. However, these algorithms do not support weighting of disks, removal of disks and data replication. Brinkmann, et al. [5, 6] proposes a method for pseudo-random distribution of data to multiple disks using partitioning of the unit range. This method accommodates growth of the collection of disks by repartitioning the range and relocating data to rebalance the load. However, this method does not move an optimally number of objects of replacement, and does not allow for the placement of replicas. Honicky, et al. [7,8] presents algorithms for balanced distribution of data to disks that move an optimally low number of objects when disks are added, which supports weighting of disks and replication, but do not support removal of disks [7], however, the methods relies upon iterating for producing the same sequence of numbers regardless of the number actually required, and the large-scale iterations increase the lookup time. We present

an algorithm for balanced distribution of data to nodes that move probabilistically an optimally number of objects when nodes are added or removed, which supports weighting of nodes, but do not support replication [9].

In the algorithm, data objects are always distributed among the storage nodes according to their weights. When new nodes are added or existing nodes are removed, it distributes objects to nodes evenly, and redistributing as few objects as possible and preserves this balanced distribution. Moreover, our algorithm almost always moves a statistically optimal number of objects from every storage node in the system to each new storage node, rather than from one storage node to one storage node. It supports data replication and guarantees that replicas of a particular object are not placed on the same node. The algorithm is very fast, and scales with the number of storage nodes groups added to the system. Its time complexity and storage requirements compare favorably with known methods. The rest of the paper is organized as follows. Section 2 contains definitions, including descriptions of the measures of "goodness" of a mapping method that are of interest to us. Section 3 presents a self-adaptive data objects placement algorithm supporting weighted allocation and replication. Section 4 gives performance analysis and simulation results. Section 5 summarizes the paper.

2 The Model and Definitions

Given a positive integer B, the number of data objects, and a positive integer N, the number of storage nodes, and a positive integer R, the maximum degree of replication for an object, the problem is to construct a mapping f from the set of object id's (0, 1,2, . . . , B-1) and the replica number r ($0 \le r < R$) of the object in question to the set of node id's (0, 1,2, . . . , N-1). Typically, B is much larger than N. When an expansion occurs, the number of storage nodes increases from N to some N', we have to construct a new mapping f' to reassign the node number in N' for data access. We can view a mapping method as a function M (x, r, p) that takes a data object id x, the replica number r and a representation p of a particular mapping, and returns a storage node id. That is, f (x, r) = M (x, r, p) where p is the representation of f. For example (no replication), for the RR method mentioned in the Introduction, the representation p is simply n, and M (x, p) = x mod n; for the SD method, p is a list (y_0, y_1, . . . y_{B-1}) of integers, and M (x, p) = y_x.

Let the size of storage node i under the mapping f is l_i, which is the number of data objects that f maps to i. Let the weight of storage node i is w_i. Measures of the goodness of a solution include the following:

(1) Balance. A mapping f from B objects onto N nodes is said to be balanced if for every pair of nodes in the system i and j, the expected ratio between the size of i and j is equal to the ratio of the weights assigned to i and j (i.e. $\frac{l_i}{l_j} = \frac{w_i}{w_j}$).

(2) Mapping Complexity. This is the number of operations needed to compute f (x), given an object id x.

(3) Mapping Storage. This is the amount of storage needed to store a representation of the mapping. In placing upper bounds on the mapping storage of a particular

mapping method M (x, p), we bound only the storage needed for the representation p (which can, in general, depend on N, B, and the number of expansions), and we ignore the (constant) storage needed to hold an algorithm for computing M.

(4) Object Relocation. When a mapping f is replaced with another mapping f ' as the result of an expansion, the object relocation of the expansion is the number of objects that are assigned to different nodes by f and f', i.e., the number of object id's x such that $f(x) \neq f'(x)$ and $0 \leq x < B$.

3 Replication Under Dynamic Interval Mapping

3.1 Representation of the Mapping

We assume that system storage nodes are partitioned into sub-clusters; sub-clusters consist of identical storage nodes that are added, removed, and reweighed as a group. The entire storage system consists of multiple server sub-clusters, accreted over time. In most systems, sub-clusters of storage nodes have different properties—newer storage nodes are faster and have more capacity. We must therefore add weighting to the algorithm to allow some storage nodes to contain a higher proportion of objects than others. We assign weight factor w_j to a single storage node in sub-cluster j. This factor will likely be a number that describes the power (such as capacity, throughput, or some combination of the two) of the storage node. Suppose that we are in a situation where m expansions have occurred. Part of the representation of the mapping is the sequence $N_0, N_1, N_2, \ldots, N_m$, where $N_0>0$ is the number of storage nodes initially, and N_j is the total number of storage nodes after the jth expansion. It is convenient to define $N_{-1}=0$. Let $d_j=N_j-N_{j-1}$ for $0 \leq j \leq m$. Thus, at the jth expansion, d_j storage nodes are added to the existing N_{j-1} storage nodes to create a new total of N_j storage nodes. Note that $d_j>0$ for $0 \leq j \leq m$, since $N_{j-1}<N_j$. In what follows, we assume that the numbers d_j is also stored, although an alternative is to recompute a particular d_j whenever it is needed. Define the jth sub-cluster, for $0 \leq j \leq m$, to be storage nodes with id's in the interval $[N_{j-1}, N_j)$. (For integers z_1 and z_2 with $z_1<z_2$, the interval $[z_1, z_2)$ contains all integers z with $z_1 \leq z < z_2$.) Thus, d_j is the number of storage node in the jth sub-cluster.

Suppose that we have a random function $H: \{0, 1, \ldots, M\} \rightarrow [0,1)$, the function H maps the data object's id uniformly at random to real numbers in the interval [0,1). The basic idea of the mapping is to map the space [0, B) of data object id's into intervals in [0,1) and divide the interval [0, 1) into different length intervals according to weight of sub-clusters; All objects mapped to the same interval are mapped to storage nodes that belong to the same sub-cluster. A storage node can contain objects from several different intervals. When sub-clusters are changed, current intervals are divided into more small intervals rather than the interval [0,1) is redefined and different intervals are reassigned into new sub-clusters, resulting in data objects replacement.

In addition to m, the N_j's, and the d_j's, the rest of the representation of the mapping consists of the following:

(1) An integer $k \geq 1$, the number of intervals.

(2) Real numbers a_i for $0 \leq i \leq k$ where

$0=a_0<a_1<a_2<\ldots<a_k=1$

The ith interval is $[a_{i-1}, a_i)$, for $1 \leq i \leq k$. We imagine that the intervals are ordered from left to right, and we say that the ith interval is to the left of the jth interval (and that the jth is to the right of the ith) if $i < j$.

(3) Nonnegative integers b_i, for $1 \leq i \leq k$. For the ith interval $[a_{i-1}, a_i)$, the number b_i is the sub-cluster number associated with this interval. Thus, $0 \leq b_i < m$. All data objects H(x) in $[a_{i-1}, a_i)$ are mapped to storage nodes in sub-cluster b_i, Define sub-cluster(x) = b_i, for all H(x) in $[a_{i-1}, a_i)$.

In general, several intervals can be mapped to the same sub-cluster; that is, we can have $b_i = b_j$, for different i and j.

(4) Nonnegative real numbers c_i, for $1 \leq i \leq k$. For each i, the number c_i, is the total length of intervals of objects x's H(x) in intervals to the left of the ith interval (i.e.,$H(x) < a_{i-1}$) such that x is mapped to a sub-cluster b_i, (i.e., sub-cluster(x) = b_i). The c_i's are helpful in computing the mapping. Note that c_i is the total length of intervals of objects x in intervals to the left of the ith such that x is mapped to sub-cluster b_i. We call c_i the offset adjustment of the ith interval.

3.2 Computation of the Mapping

The algorithm becomes slightly more complicated when we add replication because we must guarantee that no two replicas of an object are placed on the same server, while still allowing the optimal placement and migration of objects to new sub-clusters. Given a data object id x and its replica number r, the way to compute the mapping is first to determine the number of replicas which belong in each sub-cluster according to its weight, and find the interval $[a_{i-1}, a_i)$ to which x belongs, and then to compute the mapping using b_i, N_{j-1}, and d_j ($j=b_i$). Once it has determined that a particular sub-cluster should contain u replicas of an object, it selects u storage nodes randomly from that sub-cluster. Pseudo-code for the mapping computation is given by Algorithm 1 in Figure 1, where $0=u_0<u_1<u_2<\ldots<u_m=1$, the interval length of $[u_{j-1}, u_j)$ is the weight rate of the jth sub-cluster.

```
Algorithm 1: Mapping Computation
Input: A object id x and its replica number r
Find i such that H(x) is in [a_{i-1}, a_i)
j = b_i
if (r = 0)
        Return N_{j-1} + x mod d_j
else
        Find j such that H(r) is in [u_{j-1}, u_j)
        choose a random prime number p> d_j
        Return N_{j-1} + (x +r*p) mod d_j
end if
```

Fig. 1. Algorithm for mapping computation

3.3 The Initial Representation

Initially, when there are no expansions have occurred, the representation is given by m=0, k=1, a_0=0, a_1=1, b_1=c_1=0, and d_0=N_0, Thus, the mapping is exactly given by y=x+r*p mod N_0.where p is a random prime number (p> N_0)

When the number of storage nodes is changed, the representation of the mapping must be modified Assume that we are in a situation where m expansions have occurred previously (for some m≥0) and that we have a representation of the mapping, from B data objects to N_m storage nodes, as described above; call this mapping the old mapping. There are two cases.

3.4 Adding Sub-cluster

Suppose that the (m+1)th sub-cluster is added, which consists of storage nodes in [N_m, N_{m+1}). The basic idea is, for each sub-cluster j with 0≤j≤m, to move the proper number of objects from sub-cluster j to the (m+1)th sub-cluster so as to produce a new balanced mapping from B objects to N_{m+1} nodes. Among the objects in sub-cluster j, the ones with a larger random number H(x) are moved. This has the effect that if an object stays in the same sub-cluster, then it remains mapped to the same node. So for each sub-cluster j with 0≤j≤m, there will be a splitting point s_j such that, for each object x mapped to sub-cluster j in the old mapping, if H(x)<s_j, then object x remains in sub-cluster j in the new mapping, and if H(x)≥s_j, then object x is moved to the new

Algorithm 2: Computation of Adding Cluster Actions
Input: A new number N_{m+1} of Nodes

total = $\sum_{j=0}^{m+1} d_j * w_j$

```
for j =0 to m
    t_j = d_j*w_j/total
end for
w = 0
for i = 1 to k
    j = b_i
    if t_j≥(a_i-a_i-1+c_i) then
        A_i = Null
    else if t_j≤c_i then
        A_i = Move(w)
        w = w + a_i-a_i-1
    else
        s =a_i-1 + t_j - c_i,
        A_i = Split(s,w)
        w = w +a_i - s
end for
```

Fig. 2. Algorithm for computing adding cluster actions

(m+1)th sub-cluster in the new mapping. If $a_{i-1}<s_j<a_i$, for some interval $[a_{i-1}, a_i)$ with $b_i=j$ in the representation of the old mapping, then this interval will be split into two intervals, $[a_{i-1}, s_j)$ that remains mapped to sub-cluster j, and $[s_j, a_i)$ that is mapped to the (m+1)th sub-cluster. To make the following description of mapping expansion independent of implementation, the result is given as a set of actions to be performed. There is an action A associated with each interval $[a_{i-1},a_i)$ in the representation of the old mapping. There are three types of actions:

1. If A_i = Null, then objects in the interval $[a_{i-1}, a_i)$ do not move. The sub-cluster number and the offset adjustment of the interval do not change.
2. If A_i = Move(c), then all objects in the interval $[a_{i-1}, a_i)$ are moved to the (m+1)th sub-cluster. The sub-cluster number of the interval is changed to m, and c becomes the new offset adjustment of the interval.
3. If A_i = Split(s, c) , then the interval $[a_{i-1}, a_i)$ is split into two intervals, $[a_{i-1}, s)$ and $[s, a_i)$. Objects with H(x) in $[s, a_i)$ are moved to the (m+1)th sub-cluster, and c is the offset adjustment of the interval $[s, a_i)$. Objects in $[a_{i-1},s)$ do not move; the sub-cluster number and offset adjustment of $[a_{i-1},s)$ are identical to those of $[a_{i-1}, a_i)$ in the old mapping.

Pseudocode for computing the appropriate actions is given by Algorithm 2 in Figure 2.

Algorithm 3: Computation of Removing Cluster Actions
Input: A removed rth cluster

total = $\sum_{j=0, j\neq r}^{m} d_j * w_j$

```
for j =0, j≠r to m
      t_j = d_j*w_j/total
end for
j = 0
for each interval [a_{i-1},a_i) with b_i=r
      if j≠r then
            t = total interval length of the cluster j
            if (t_j – t) ≥ (a_i-a_{i-1}) then
                  A_i = Move (t)
                  t = t + a_i-a_{i-1}
            else
                  s = t_j – t + a_{i-1}
                  A_i = Split(s, t)
                  j = j + 1
      end if
end for
```

Fig. 3. Algorithm for computing removing cluster actions

3.5 Removing Sub-cluster

Suppose that the rth sub-cluster is removed, which consists of storage nodes in $[N_{r-1}, N_r)$. The basic idea is to move the proper number of objects from sub-cluster r to other sub-cluster j with $0 \le j \le m$ and $j \ne r$, so as to produce a new balanced mapping from B objects to $N_m - d_r$ nodes. So for each interval $[a_{i-1}, a_i)$ of the rth sub-cluster, either the all the interval $[a_{i-1}, a_i)$ is moved to some sub-cluster j with $0 \le j \le m$ and $j \ne r$, or there will be a splitting point s such that, $[a_{i-1}, s)$ is moved to some sub-cluster j, $[s, a_i)$ is remained to next movement To make the following description of mapping expansion independent of implementation, the result is given as a set of actions to be performed. There is an action A associated with each interval $[a_{i-1}, a_i)$ of the rth sub-cluster. There are two types of actions:

1. If A_i = Move(c), then all objects in the interval $[a_{i-1}, a_i)$ are moved to sub-cluster j. The sub-cluster number of the interval is changed to j, and c becomes the new offset adjustment of the interval.
2. If A_i = Split(s, c) , then the interval $[a_{i-1}, a_i)$ is split into two intervals, $[a_{i-1}, s)$ and $[s, a_i)$. Objects with H(x) in $[a_{i-1}, s)$ are moved to sub-cluster j, and c is the offset adjustment of the interval $[a_{i-1}, s)$. Replace the interval $[a_{i-1}, a_i)$ of the rth sub-cluster with $[s, a_i]$ and continue.

Pseudocode for computing the appropriate actions is given by Algorithm 3 in Figure 3.

The RDIM method has the following property:

- The number of objects placed in a sub-cluster is proportional to the total length of intervals mapped to the corresponding sub-cluster.
- The number of objects placed in any sub-cluster is proportional to its weights.
- When storage nodes are changed, the number of objects migrated is the minimum.

Since objects are distributed evenly to storage node in any sub-cluster by the algorithm for mapping computation. So we draw the conclusion that the dynamic interval mapping is balanced algorithm and the number of objects relocated is the minimum.

4 Performance and Simulation Results Analysis

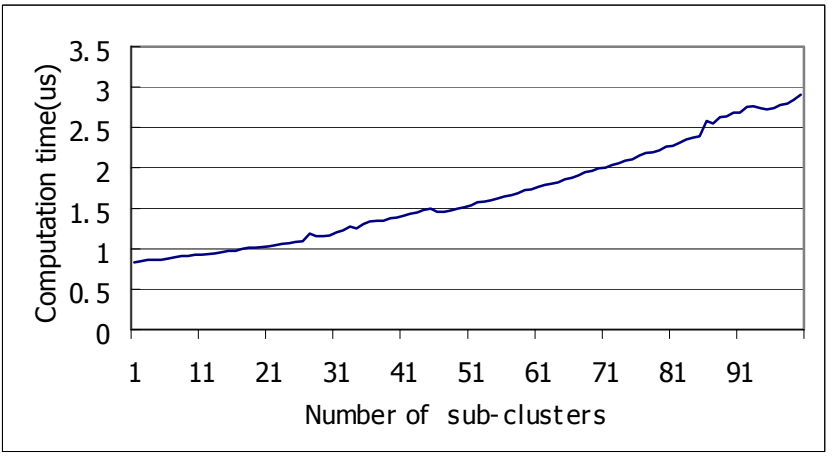
4.1 Performance

Since both mapping complexity and mapping storage depend on the number k of intervals, it is useful to have an upper bound on k as a function of m. The following gives such a bound.

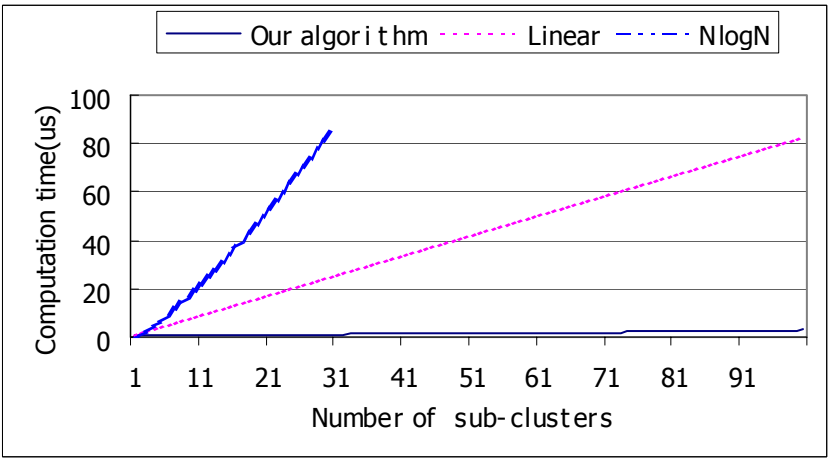
THEOREM 1. If k intervals are produced as the result of m expansions to the number of storage nodes, then

$$k \le \frac{1}{2} m*(m+1) + 1$$

Proof. The proof is by induction on m. Initially (when m=0) there is one interval. Assuming that the bound holds for m expansions, we prove it for m+1 expansion. Just before the (m+1)st expansion, there are m+1 sub-clusters, 0 through m. For each of these sub-clusters, there will be at most one interval that is mapped to the sub-cluster and that is split during the (m+1)st expansion. So the (m+1)st expansion causes at most m+1 intervals to be split, thus creating at most m+1 new intervals. Therefore, using the induction hypothesis, the total number of intervals after m+1 expansions is at most $\frac{1}{2}$ m*(m+1)+1+(m+1)=$\frac{1}{2}$ (m+1)*(m+2)+1.



(a) Time per lookup per replica as the number of sub-clusters increases



(b)Time per lookup compared to linear and nlogn functions

Fig. 4. Time for looking up an object versus the number of sub-clusters in the system

In the implementation, the numbers a_i, b_i, c_i, d_j, and N_j, are stored in random-access tables or tree, The Find operation in Algorithm 1 is done by binary search in the table or the tree. Obviously, mapping complexity is O(log k) and mapping storage is O(k). By Theorem 1, mapping complexity is O(log m) and mapping storage is O(m2).

In our algorithm, we need a random function H, which maps the objects uniformly at random to real numbers in the interval [0,1). We select the Mersenne Twister[10] as the random function H in the implementation of our algorithm.

In order to quantify the real world performance of our algorithm, we tested the average time per lookup under many different configurations for a system with 1000000 objects and 4 replicas per object. First, we ran a test starting with 10 storage nodes in a single sub-cluster and computed the average time for these 4000000 lookups, and then added sub-clusters, 10 storage nodes at a time, and timed the same 4000000 lookups over the new server organization. Figure 4(a) shows the per-object per-replica lookup time with slightly growth rates for the capacity of the most recently added sub-clusters, even with 100 sub-clusters in the system, the amortized lookup time is less than 3 μs on the 1.4GHz Pentium Ⅳ on which we ran these experiments; In Figure 4(b), we can see that the line for lookups grows far slower than linear and NlogN.

4.2 Data Distribution

We evaluate the balanced distribution of data objects supporting weighted allocation and replication. The simulation system includes 3 sub-clusters; the first sub-cluster includes three storage nodes with weight 1, the second sub-cluster includes two storage nodes with weight 3, the third sub-cluster includes four storage nodes with weight 5, the maximum degree of replication for each object is 3. The 100000, 200000, 400000, 800000 data objects from four clients are sent respectively to storage nodes. Figure 5 show that data objects sent from four clients and the total sums are always distributed among the storage nodes according to their weights.

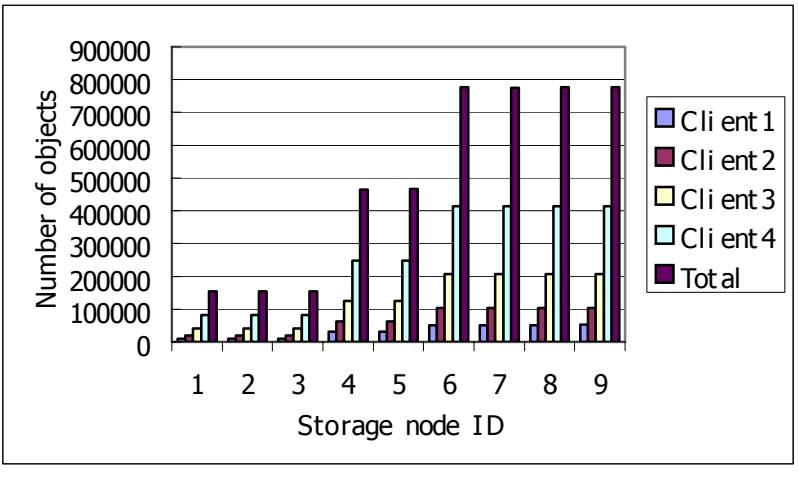


Fig. 5. The distribution of data objects according to nodes weight

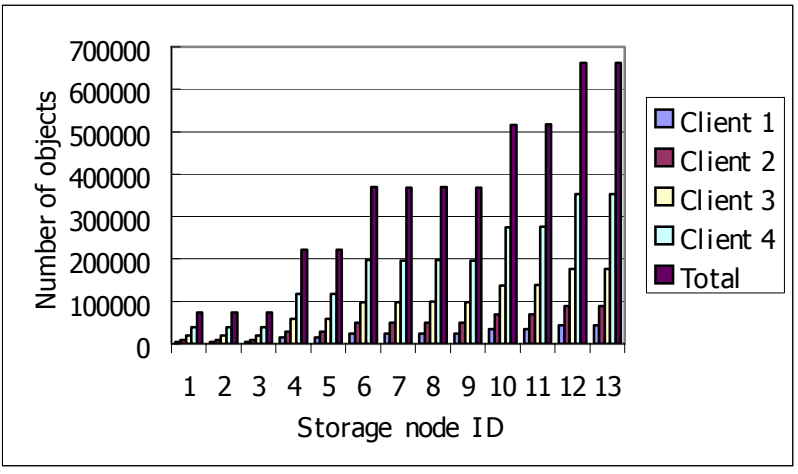


Fig. 6. The redistribution of data objects after adding two clusters

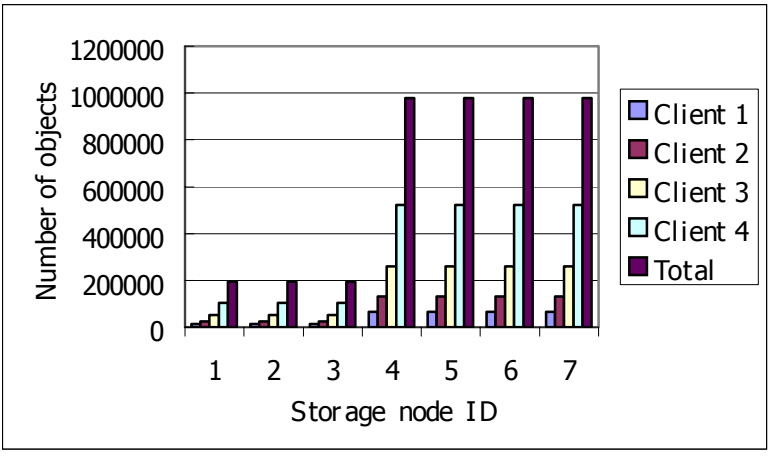


Fig. 7. The redistribution of data objects after removing one cluster

Then, we evaluate the balanced redistribution of data objects supporting weighted allocation by adding two sub-clusters and removing one sub-cluster respectively. (1) Add two sub-clusters, the first sub-cluster includes two storage nodes with weight 7; the second sub-cluster includes two storage nodes with weight 9. (2) Remove the second sub-cluster, which includes two storage nodes with weight 3. Figure 6 and Figure 7 show that data objects sent from four clients and the total sums are always redistributed among the storage nodes according to their weights after adding or removing sub-cluster.

5 Conclusions

In this paper, we propose a self-adaptive and balanced distribution algorithm for replicated data objects in scalable storage clusters, which distributes objects to nodes evenly, redistributing as few objects as possible when new nodes are added or existing nodes are removed to preserve this balanced distribution. It supports weighted allocation and guarantees that replicas of a particular object are not placed on the same node. Its time complexity and storage requirements compare favorably with known methods.

References

[1] Q. Xin, E. L. Miller, D. D. E. Long, S. A. Brandt, T. Schwarz, and W. Litwin. Reliability mechanisms for very large storage systems. In Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies, pages 146–156, Apr. 2003.

[2] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH*—a scalable, distributed data structure. ACM Transactions on Database Systems, 1996, 21(4): 480-525.

[3] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms, pages 101–114, 1993.

[4] D. M. Choy, R. Fagin, and L. Stockmeyer. Efficiently extendible mappings for balanced data distribution. Algorithmica, 1996, 16:215-232.

[5] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA), ACM Press. Extended Abstract. 2000, 119-128.

[6] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform capacities. In Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA), Winnipeg, Manitoba, Canada , Aug. 2002. 53-62.

[7] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In Proceedings of the 17th International Parallel & Distributed Processing Symposium, Nice, France, Apr. 2003.

[8] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004), Santa Fe, NM, Apr. 2004. IEEE.

[9] Zhong Liu, Xing-Ming Zhou. An Adaptive Data Objects Placement Algorithm For Non-Uniform Capacities, In Proceedings of the 3rd International Conference on Grid and Cooperative Computing, WuHan, Oct. 2004.

[10] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 1998.