Size-based Data Placement Strategy in SAN

Yihua Lan,Yong Zhang, *Haozheng Ren
School of Computer Engineering, Huaihai Institute of Technology, Lianyungang, China
Email: lanhua_2000@sina.com,zhyhglyg@126.com,renhaozheng666@163.com

Chao Yin
School of Computer Science, Huazhong University of Science and Technology, Wuhan, Hubei 430074 P.R.China
Email: 675332223@qq.com

***Abstract*—This paper introduced an even data distribution strategy in SAN. Even data distribution is always the target of distributed storage design. Many articles dedicate the purpose. We use size-based data placement here to guarantee the data distributed evenly at furthest. Size-based data distribution separates the SAN into zones, and each zone has some storage devices (SD), different sized data will fill into different zones. Two kind of allocating algorithms are used to keep the data distribute evenly: orderly and best fit first. Our design will achieve 1) the data will even distributed on each storage devices at furthest; 2) read and write requests will even access the SDs, and 3) bandwidth can be improved at our system. Experimental results showed our design can optimize data distribution.**

***Index Terms*—Data placement, Size-based, SAN, Even distribution**

I. INTRODUCTION

Data placement is a fascinated field in data storage [1-2]. Common consideration in data placement is effectivity, even distribution, and less energy consumption. Effectivity is a complex issue, for many conditions can affect the affectivity, such as CPUs' ability of computation, disks' speed, throughout of network or I/O circuit and so on. Then, energy saving is another hot point of current storage research, and it is not a single problem of data placement. In this article, we dedicate in the even distribution of data.

Even distribution of data also has many solutions and different angles to the problem. Consistent hashing [3] proposed an even distribution in distributed storages. Ref. [4] is a fast data replacement focused on replicated data. Andre Brinkmann et.al [5] proposed a compact, adaptive data placement for non-uniform disks. Andre Brinkmann et.al [6] also found an efficient distributed data strategy for Storage Area Networks. Ricardo Vilaca et.al [7] researched the relationships between data, and proposed a correlation-aware data placement strategy for key-value stores. In peer-to-peer system, Ramazan S. Aygun et.al proposed a conceptual model for data management and distribution method [8].

Our consideration here is for even distribution, and not only the data requests, but also focuses on the size of requests. Our distributed environment is a Storage Area Networks (SAN), and in the SAN, there are some storage managers(SM) and some storage devices (SD). Storage managers distribute the data to the storage devices, distribution strategy based on the free capacity of each disks or devices.

Manuscript received August 8, 2012; revised September 1, 2012; accepted September 1, 2012. This work was part supported by the Natural Science Foundation of Huaihai Institute of Technology under Grant Z2011036, *corresponding author.

II. GENERAL DESIGN OF SYSTEM

The system we designed here as "Fig. 1" shows.

In the SAN, there is m SMs to manage the outer I/O request. The main job of SM are: get outer I/O requests and dispatch them to the SDs, including read requests and write requests; manage the free space of each SD, allocate the write requests to SDs to even distribute the data based on requests' size; manage the metadata and data allocation table. The n SDs are store data. A single SD can be a single disk, or a RAID.

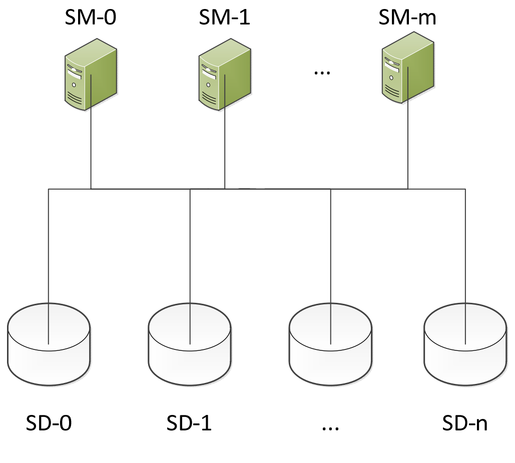

Figure 1. General design of our SAN.

A. Data Reallocating Strategy in SM

SM will reallocate the data. To reallocate a request, two ways can be used. The first way is calculate the original address to new address, that is:

$$new\ address = \Phi(original\ address) \quad (1)$$

doi:10.4304/jsw.8.2.426-434

All the original address can map to a new address, and the function can be hashing function, such as consistent hashing, or simple arithmetic function. This way is easy for SM which can only store the function Φ.

Another way is using mapping table. Mapping table stores a couple of addresses such as: (original address, new address). Mapping table can help SM reallocate data in flexible manner, but SM should store a huge mapping table in memory.

The compromise is to use hierarchic storage management. Our strategy with data reallocating is storing different size of data in different SDs. The first level of storage system is classifying level. This level composed by one or more SMs. These SMs dispatch the outer I/O requests to other SMs, dispatching based on the requests' size. For example, the request can fall into three zone according the size of data: (0, 8kB], (8kB, 16kB],(16kB,+∞). Each size zone can be managed by some SMs, these SMs are belong to second level. As "Fig. 2" shows, each size zone has 1 SM, and first level only has 1 SM. The third level is storage level, it is composed

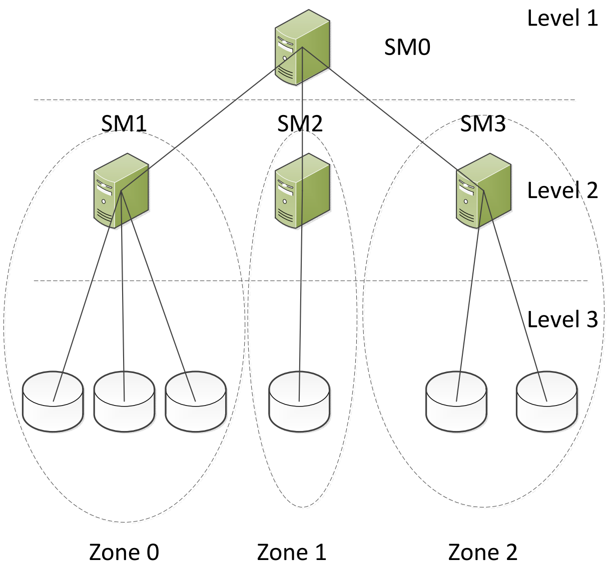

Figure 2. Three zones in storage system.

by all SDs. The third level stores all the data. Every SD belongs to a size zone, and only store the data which size is the same as corresponding size zone.

The SM in first level uses a single function to dispatch the requests. For the example above, the corresponding function should be:

$$Send\ to = \begin{cases} SM1\ data\,size < 8KB \\ SM2\ 8KB \le data\,size < 16KB \\ SM3\ data\,size \ge 16KB \end{cases} \quad (2)$$

The SMs in second level can use different way to dispatch requests to their corresponding SDs. The first way is simply sending the data into SDs. This method requires the SM record the mapping table of all the data. For large data size requests, this method is working well for little requests can be large data size, so less records in mapping table. The mapping table should be a triple: (original address, new address in SD, data size). The new address in SD composed by two components: SD number and offset address in the corresponding SD.

The second way to send data to SDs is using linear hashing function. This is very useful when requests are small data size. Mapping table should be large if data size is small, and linear hashing function is the better way to reallocate the small data requests.

B. Operation Commands of SM

For completing the SM's work, some commands should be defined. These commands mainly include write and read operations. SMs in first level have some different command with level 2 SMs, and the commands show as follow:

Transfer requests: This command will dispatch requests to different level 2 SMs based on data size. The command requires outer data, especially its data address, data size and read/write flag. Data requests should dispatched to level 2 SMs according to the data size. Output of this command is level 2 SM number as (2) illustrated. For read requests, this command will wait until next command "Return reading result" returns, while write requests, this command will return true or false when lower devices success write data or fail.

Return reading result: This command receives the data from level 2 SMs and returns it to the upper file system.

The second level SMs have different commands with level 1 SMs, and they are showed below:

Dispatch requests: This command dispatches the requests from level 1 SMs to the lower SDs. It requires requests' information including data address, data size and read/write flag. According the reallocate strategy described in section 2.A, the SMs in level 2 will fill the mapping table or calculate the corresponding arranged address according to a linear hashing function such as (1).

Return reading result: This command will return reading requests' data from SDs which belongs to the same zone of level 2 SM.

C. Lookup the Requests' Address

Generally, the requests will find their address according the algorithm and commands we talked above, but sometimes, different requests will access the same address with different data size. For example, zone divided as (2) defined, and two requests come with following information:

a) Write 32KB in address of 3412.
b) Read 4KB in address of 3412.

The scenario will occur when write a file to the disk, then read a piece of file in a certain demand. According the algorithm, and (2), request a) will write the data in a storage device in zone 2, the address transformation will complete in SM3, but if request b) comes, SM0 found its data is only 4KB, so the SM0 will dispatch request b) to SM1. SM1 have no correct data what request b) needed, so it certainly fail.

To solve the problem, some SMs will record the original data address. Some SMs in level 2 already record the mapping table, and the original addresses of requests are recorded. So we only to add an original address table

(OAT) to the SMs which uses hashing function to dispatch requests.

OAT only record which requests' addresses are in the zone. The size of OAT will be large if small data requests continuing come. The novel way to store these address is to merge adjacent data. To merge adjacent data, data size of each request will record. Luckily, the data request is a multiple of some certain base size, i.e. 1KB, so we can split the OAT into several groups. For example above, we can split OAT in SM1 as 8 groups, that is: group 0(1KB), group 1(2KB), group 2(3KB) and so on. All the data with the same size will in the same group. If some data are adjacent, they can be record as a single item. With grouping, the OAT item will be:

(Group number, original address, data block account)

Most adjacent data will be not the same size, and this is more complex for OAT item. We should record all the data size in the item. The final item form should be:

(Original address, data size, data block account, data size, data block account, …)

All the items in OAT are sorted by original address.

"Fig. 3" illustrated the scenario of adjacent data with same size, and the item in OAT would be (0x10000, 1, 4). This is the final form, and the second number means the single data block is 1 times of 1KB.

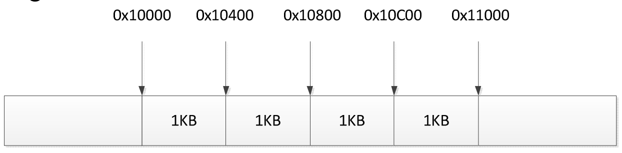

Figure 3. Adjacent data with same size.

"Fig. 4" illustrated the scenario of adjacent hybrid data blocks with different size. The corresponding item in OAT is: (0x10000,1,2,2,2,1,1).

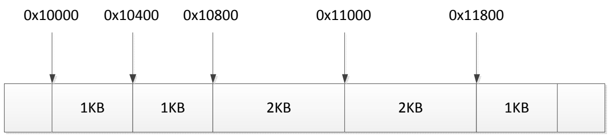

Figure 4. Adjacent data with same size.

OAT will fill with coming requests, and each request will fill one item of OAT.

But the requests will come at any time, and two operations will affect the OAT. The first is writing a data block that exactly adjacent with some data blocks, the other operation is erasing a data block that this data blocks is a part of an adjacent data blocks. The first operation will not create new item in OAT, just modify the corresponding item, if the next item's start address is at the end of current item's end address, the two items will merge into one item. The second operation need break an item into two items if the erased data block is at the center of chain, or only modify the start address, if the erased data block is at head of the chain, or subtract 1 of last number in the item, if the erased data block is at tail of the chain. The algorithm as "Algorithm 1" and "Algorithm 2" described.

Searching for a data is simple according to the OAT items. Next paragraph, we will use some examples to explain various operation about OAT.

Example 1. Request reads 1KB with the address 0x10400 in "Fig. 3".

Address should be lookup at first. For OAT is sorted by original address, the first address larger than 0x10400 can be found, we go back to find the previous address, that is 0x10000, and the required data probable in the item with the start address is 0x10000. Then, the end address can be calculated according to the item, which is 0x11000. 0x10400 less than 0x11000, so the SM can send read request to the SDs to read 1KB data.

Algorithm 1. Writing data operation in OAT

```
Require: data address(DA), data size(DS)
1.  Find the item in OAT whose original address is the
    largest address which less than DA, mark this item as
    item0, its next item as item1;
2.  According item0, calculate the last address LA of the
    item;
3.  if(LA<DA and original address of item1 > DA)
4.  {
5.          Create a new item(DA,DS,1) in OAT;
6.          Return true;
7.  }
8.  if(LA==DA)
9.  {
10.         Add (DS,1) to item0's last;
11.         if(DA+DS==original address of item1)
12.         {
13.                 Add item1 to item0;
14.         }
15.         Return true;
16. }
17. if(LA>DA)
18. {
19.         Modify item0 according DA and DS;
20.         if(DA+DS==original address of item1)
21.         {
22.                 Add item1 to item0;
23.         }
24.         Return true;
25. }
26. if(LA<DA)
27. {
28.         Modify item1 according DA and DS;
29.         Return true;
30. }
31. Return false;
```

Algorithm 2. Remove data operation in OAT

```
Require: data address(DA), data size(DS)
1.  Find the item in OAT whose original address is the
    largest address which less than DA, mark this item as
    item0, its next item as item1;
2.  Calculate the original address and last address of
    item0 and item1,marked with SA0,LA0,SA1,LA1;
3.  if(SA0≤ DA≤LA0)
4.  {
5.          Split item0 into 2 sub-items;
6.         Sub-item0's original address is SA0, and its
      last address is DA, sub-item1's original address is
      DA+DS, last address is LA0;
7.         Return true;
8.    }
9.    if(SA1≤ DA≤LA1)
10.   {
11.        Split item1 into 2 sub-items;
12.        Sub-item0's original address is SA1, and its
      last address is DA, sub-item1's original address is
      DA+DS, last address is LA1;
13.        Return true;
14.   }
15.   Return false;
```

Example 2. The request reads 1KB at 0x11000 in "Fig. 4".

First find the largest address which less than 0x11000, and that is 0x10000 in "Fig. 4". Then calculate the last address of the item. The item shows this data block chain has 2 data blocks which is 1KB, and 2 data blocks which is 2KB, at last, 1 data block which 1KB. Total data size is 7KB, so the last address of this chain is 0x11C00, it is larger than 0x11000, so the data stores in 0x11000 is correct data, the SM can send request to lower SDs.

Example 3. Add 1KB data at 0x11000 in "Fig. 3".

According to the "Algorithm 1", DA = 0x11000, DS=1KB, the largest address which less than DA is 0x10000, LA=0x11000. Because DA == LA, (1KB,1) should added to the item. The item is all 1KB data blocks, so only modify the number of data block to 5, that is (0x10000,1,5).

Example 4. Remove 1KB data at 0x10C00 in "Fig. 3".

According to the "Algorithm 2", DA = 0x10C00, SA0 = 0x10000, LA0 = 0x11000, SA0≤ DA≤LA0, so we split the item into 2 sub-items. First sub-item is(0x10000,1,3), the second sub-item really does not exist because the original address of sub-item1 is DA+DS = 0x11000, it is the same as LA0, so there are no data in sub-item1.

Example 5. Remove 1 KB data at 0x10800 in "Fig. 4".

According the algorithm, the item found first. SA0 = 0x10000, LA0 = 0x11C00, DA = 0x10800, DS = 1KB, and SA0≤ DA≤LA0. Two sub-items will be split. Sub-item0 is (0x10000, 1,2). Sub-item1's original address is DA+DS, that is 0x10C00, the data block at 0x10800 is a 2KB-size data block, 1KB removed and 1KB remained. So the sub-item1 is (0x10C00, 1,1,2,1,1,1).

For storing OAT in different level 2 SMs, the problem we described at the beginning of this section has not solved yet. 32KB data stores in zone2, while the next request read 4KB data, it still be dispatched to zone0. So the OAT should store in level 1 SMs, when requests come, level 1 SMs should not dispatch it to the zones simply according its data size, but first lookup the OAT to find the probable address. If the address is not found, new item will add to OAT (if the request is write request) or return false to file system (if the request is read request).

OAT will be large in level 1 SMs. Larger the OAT is, more search time cost. To accelerate the searching, more SMs can be added to level 1.

OAT in level 1 also has a problem. See the "Fig. 5", two requests are described as follow:

a) Write 1KB data block at 0x10000.

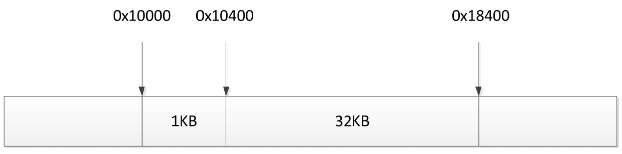

Figure 5. Two requests with different data size.

b) Write 32KB data block at 0x10400.

The item in OAT could be (0x10000,1,1,32,1). If a new request read 1KB at 0x10400, the SM still does not know dispatch the new request to which zone. The solution is identified the items in OAT with zone number. Each zone has an OAT, and searching OATs is synchronous. The items in different OAT should be (0x10000, 1,1) and (0x10400, 32,1).

Another problem is write many small sized data blocks and read a large sized data block which covers all the small data blocks.

If the small data are all in a single zone, the read request success. While if the data blocks are in different zones, more steps should be follow to search the correct data.

For all data is the multiple of a base size, i.e. 1KB, so we can split the read request into more tiny requests that each tiny request has 1KB data read request. The big read request success till all the tiny requests success.

D. Data Address Hashing in Level 2 SMs

We talked about data address mapping above. All we talked about are based on mapping table, including OAT. This section we will talk about hashing algorithm.

Hashing algorithm also mapping data address to storage devices. Consistent hashing algorithm is a good option, if multiple SDs in a zone and the SDs is distributed environment. If a zone has only 1 SD or all the SDs are composed a RAID, whatever which level the RAID is, simple linear hashing function can used.

In our design, no distributed environment is present, so we give the linear hashing function we used.

$$NA = OA \bmod C \quad (3)$$

In (3), NA is new address in SDs, and OA is original address that request's data, and C is the capacity of SDs. If the zone has only 1 storage device, C is the real capacity of the device, while if all the SDs are the devices in a RAID, the capacity of C decided by the RAID level. If the RAID has parity, the C is less than arithmetic capacity of sum of every device's capacity.

III. EVEN DISTRIBUTION OF REQUESTS

Data requests in different application will be not the same. Commonly, we do not know the requests' size and its address before it comes. But we can collect a day trace, and then analyze the trace to confirm the trace structure and data distribution according to the size.

In (2), three zones are divided and three size periods are: (0, 8kB], (8kB, 16kB],(16kB,+∞). This is only an example.

Generally, system has Z zones and zone Z_i has S_i SDs. Each SD in a zone numbered with D_0, D1,…, DSi. Totally, there are M SDs in the system.

$$M = \sum_{i=0}^{Z-1} S_i \quad (4)$$

Requests' size period splitting is to find an optimized way to fill different sized storages with different sized data, and it is a NP-hard problem. NP-hard problem has no optimized solution in global, or it is waste of time to get the solution.

To solve the problem, we assume two scenarios: the first one is fixed storage devices and the second one is requests' data size periods are fixed. The first scenario means the devices in each zone is fixed.

We will analyze the two methods to distribute data evenly. There are three hypothesizes that: a) all the storage devices are the same capacity, b) no parity exist in the RAID and c) the bandwidth of each storage devices are the same, that means if a single device's bandwidth is B, than a RAID's bandwidth with 3 devices will achieve 3B.

A. Fixed Storage Devices

Even distribution means each SD has the same size of data. Suppose the trace has N_i requests with data size is DS_i, total data size T is the sum-up of all the requests, without rewrite operation:

$$T = \sum \left(N_i \times DS_i\right) \quad (5)$$

The write new data requests will distribute in M SDs, it will be prorated to the each zone according to the SD's number. The proportion P will be:

$$P = S_0 : S_1 : S_2 : ... : S_{z-1} \quad (6)$$

Each zone will get corresponding portion of data according to (6), suppose DZ_i is the data size stored in Z_i.

$$DZ_i = \frac{S_i T}{M} \quad (7)$$

For allocating new data write requests, we should group the requests according to the data size. G_0 contains all the requests that data size less than 1KB, G_1 contains all the requests that data size less than 2KB, and so on. Totally, there may be thousands of groups. The algorithm here shows how to allocate data size periods according to DZ.

Algorithm 3. Generate data size periods orderly

Require: Total new data size(T), SD number in each zone (S_i), total SDs number(M), data size group account(GN)
Return: portion list P.

```
1.  Construct a list L that has Z elements, each element
    is the max capacity of each zone, which is $DZ_i$.
2.  Construct P that has Z elements, each element is the
    begin size of the periods, P[0] is 0KB.
3.  j=-1;
4.  For i =0 to Z-2;
5.  {
6.      CurrentCount = 0;
7.      While(j++<=GN)
8.      {
9.          if(CurrentCount+ total data size in $G_j$<L[i])
10.         {
11.             CurrentCount+=total data size in $G_j$;
12.             P[i+1]=single data size of $G_j$;
13.         }
14.         Else break;
15.     }
16. }
```

The algorithm returns P and it contains all the start data size of the periods. So the size periods can be:[P[0],P[1]),[P[1],P[2]), … ,[P[Z-1], +∞).

This algorithm fill the data orderly to the zones, each zones must contains data less than it max capacity. A better algorithm is use best fit first algorithm. Best fit means find the best capacity of zone that fit the sum of data size of several groups. The total data size of several groups may less than capacity of the zone, or large than capacity of the zone.

Algorithm 4. Generate data size periods at best fit first

Require: Total new data size(T), SD number in each zone (S_i), total SDs number(M), data size group account(GN)
Return: portion list P.

```
1.  Construct a list L that has Z elements, each element
    is the max capacity of each zone, which is $DZ_i$.
2.  Construct P that has Z elements, each element is the
    begin size of the periods, P[0] is 0KB.
3.  Construct a list K that has Z elements, each element
    are false, it mark the zone filled or not.
4.  j=-1;
5.  while(K has 2 elements are true at least)
6.  {
7.      CurrentCount = 0;
8.      while(j++<GN)
9.      {
10.         CurrentCount+=total data size in $G_j$;
11.         if(CurrentCount best fit L[i] and K[i] ==
    false)
12.         {
13.             K[i] = true;
14.             P[i] = single data size of $G_j$;
15.             if(only 1 element in K is true)
16.             {
17.                 Find the element K[k];
18.                 P[k] = +∞ ;
19.             }
20.             break;
21.         }
22.     }
23. }
24. Sort P ascend;
```

Algorithm 4 also returns P, but the element in P is the max data size of each period, that means the periods are: [0,P[0]) ,[P[0],P[1]), … ,[P[Z-2], P[Z-1]). Algorithm 4 set P[k] is +∞ in line 18, and +∞ can be a comparative large number such as 10000, only guarantee no single request write such a big data.

B. Fixed Data Size Periods

Fixed data size periods need rearrange the zones. It also has two strategies: orderly and best fit first.

For rearrange the zones should remove and add SDs inter zone, we do not talk about it here.

IV. Experimental Results

We use real traces to evaluate our design. The traces used are:

a) TPC-C: is a database trace [9]. It is a whole day trace collected by the Performance Evaluation laboratory at Brigham Young University.

b) MSR-Cambridge: this is a serial of traces [10]. They are collected according the volume of disks. The volumes include hardware monitoring server (HM), Media server (MDS), Firewall/web proxy server (Prxy) and others. But we only choose these three to evaluate our design.

Table I illustrates the trace characteristics. For we only focus the new write requests, the table list the write traffic and working set. They are all sectors, means 512B.

TABLE II.
Trace Characteristics

Trace	R/W Ratio	Write Traffic (Sectors)	Working Set (Sectors)
TPC-C	223.03	187,024	4,405,685
HM	0.55	42,943,920	5,235,621
Prxy	0.03	112,821,145	2,040,080
MDS	0.13	15,446,039	6,443,362

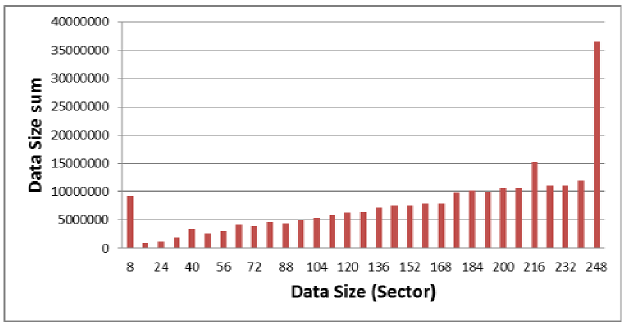

Figure 6. Data size count of TPC-C

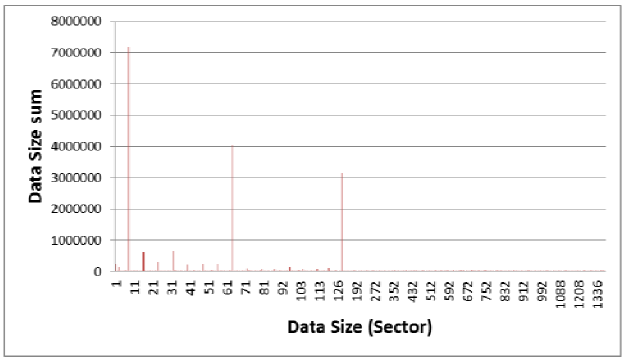

Figure 7. Data size count of HM

“Fig. 6” through “Fig. 9” are data size sum, it is the multiple of write requests count and single request’s data size.

The environment we present here is a variable configured SMs and SDs. That is to say, for different trace, we use different configurations to compare which one will fit well.

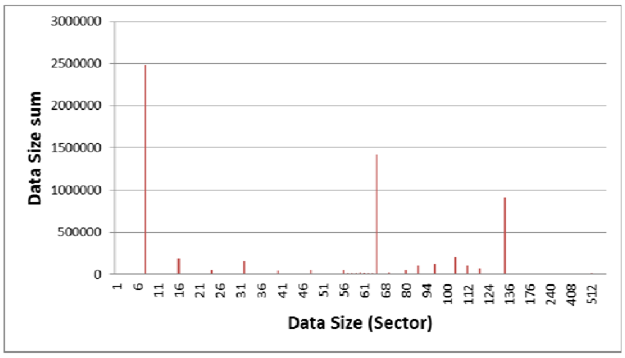

Figure 8. Data size count of Prxy

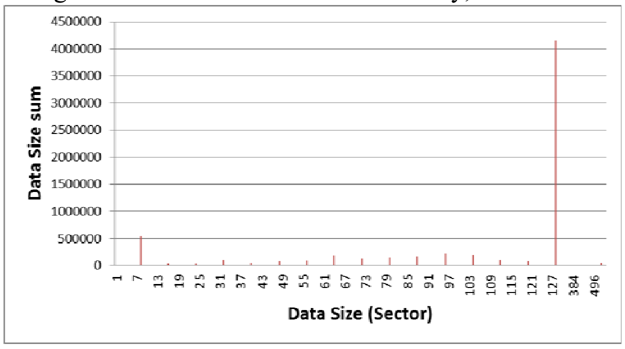

Figure 9. Data size count of MDS

A. Same Amount of SDs in Each Zone

To evaluate simply, we first configure 3 zones and each zone has the same amount of SDs.

Table II is the data size periods splitting of the traces with two algorithms.

The evaluation results show TPC-C will distribute more evenly than other traces. Some traces like MDS, different zones have notable uneven distribution. The reason of this situation is some data size is too concentrated. For MDS, request whose data size is 128 sectors almost a thousand times than others, this cause whichever the data is, the zone will store very large data. We can adjust the SDs’ amount in every zone to achieve more even distribution.

TABLE I.
Data Size Periods Splitting in Same SDs

Trace	Orderly(sector)	Best fit first(sector)
TPC-C	0,144,208,+∞	0,152,216,+∞
HM	0,9,80,+ ∞	0,9,80,+ ∞
Prxy	0,9,66,+ ∞	0,9,66,+ ∞
MDS	0,104,128,+ ∞	0,105,152,+ ∞

B. Arbitary SDs in Every Zone

This configuration is designed for uneven distributed traces like MDS and Prxy.

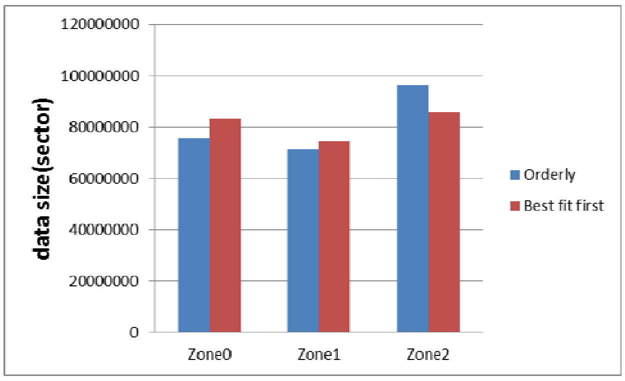

Figure 10. TPC-C's Data distribution in three zones

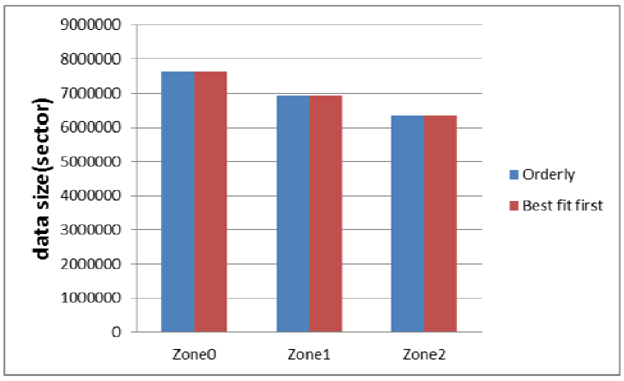

Figure 11. HM's Data distribution in three zones

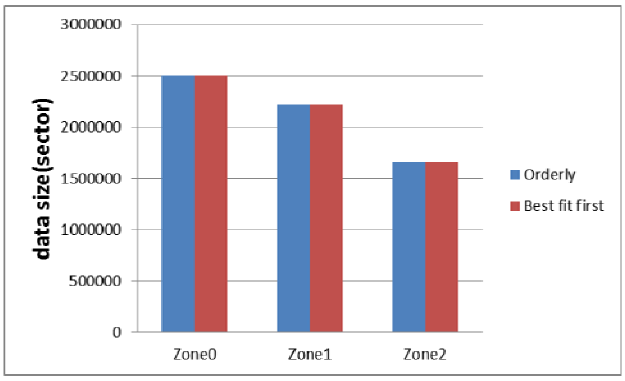

Figure 12. Prxy's Data distribution in three zones

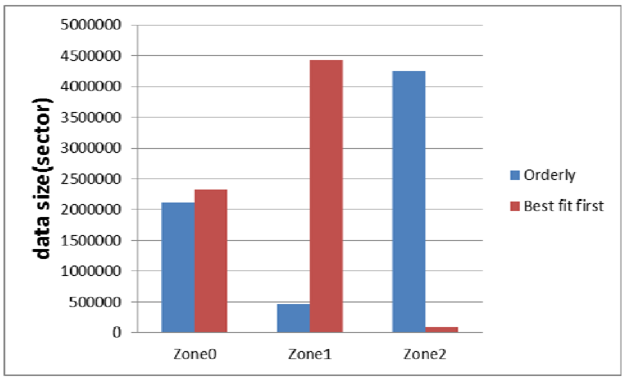

Figure 13. MDS's Data distribution in three zones

Uneven distributed traces need more SDs to store the extra-large data. We set zone0 has 3 SDs, zone1 has 10 SDs, and zone2 has 20 SDs.

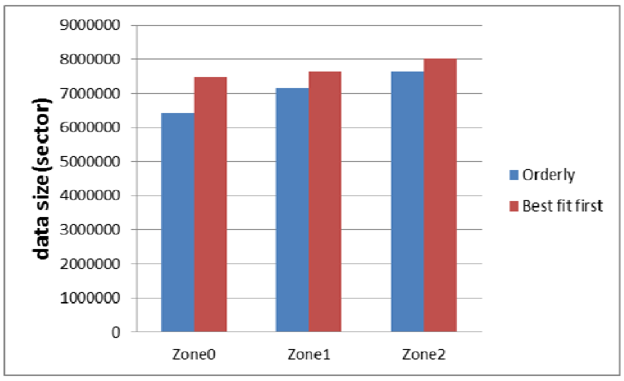

Figure 14. TPC-C's Data distribution in a SD

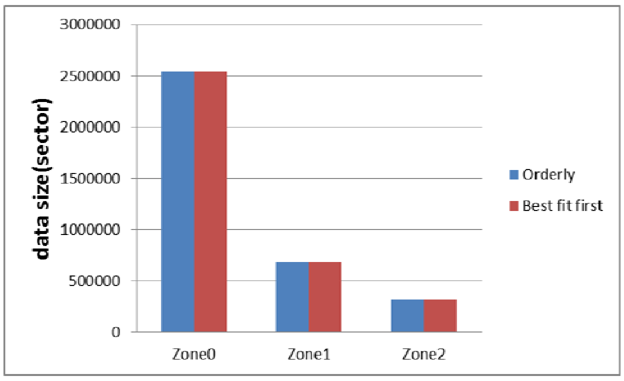

Figure 15. HM's Data distribution in a SD

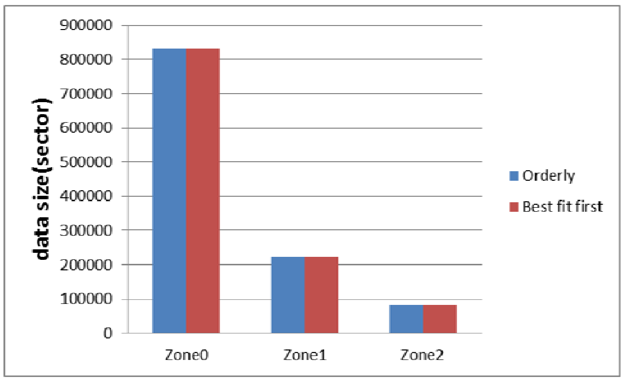

Figure 16. Prxy's Data distribution in a SD

"Fig. 14" through "Fig. 17" illustrated the results of

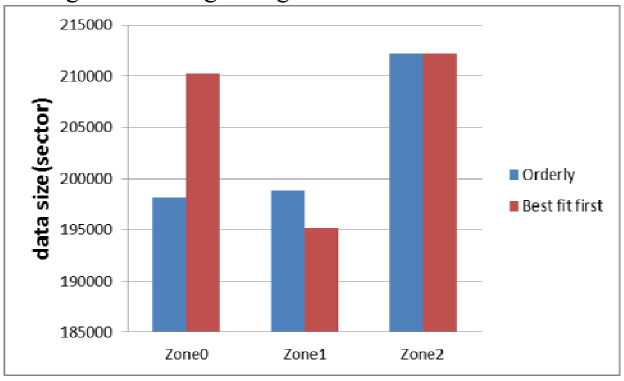

Figure 17. MDS's Data distribution in a SD

this configuration; Table III is the data size periods.

TABLE III.
DATA SIZE PERIODS SPLITTING IN ARBITRARY SDS

Trace	Orderly(sector)	Best fit first(sector)
TPC-C	0,56,160,+∞	0,64,168,+∞
HM	0,9,65,+ ∞	0,9,65,+ ∞
Prxy	0,9,66,+ ∞	0,9,66,+ ∞
MDS	0,16,128,+ ∞	0,17,128,+ ∞

Evaluation results showed that even distributed traces like TPC-C can work well in arbitrary SDs in zone. If the SDs' amount varies according to the trace trend, like MDS, it works better than the same SDs amount in each zone. But if the SDs' amount varies against the trace trend, like HM and Prxy, the distribution can be worse.

C. Contrast of Different Zone Amount

Zone amount can also affect the distribution of the data. More SDs can average the data. We use the TPC-C to evaluate the data distribution. In this configuration, we compare the data distribution of 2 zones, 3 zones, 4 zones and 10 zones, each zone has 3 SDs.

Table IV is the data size periods splitting about the configuration.

"Fig.18" through "Fig. 21" showed more zones the SAN has, more evenly the data distribute.

TABLE IV.
DATA SIZE PERIODS SPLITTING IN DIFFERENT ZONE AMOUNT WITH TPC-C

Zone amount	Orderly(sector)	Best fit first(sector)
2	0,192,+∞	0,192,+∞
3	0,144,208,+ ∞	0,152,216,+ ∞
4	0,120,184,224,+ ∞	0,128,192,232,+ ∞
10	0,64,104,136,160,176,192, 208,216,224,+ ∞	0,64,104,136,160,184,200,216, 232,248,+ ∞

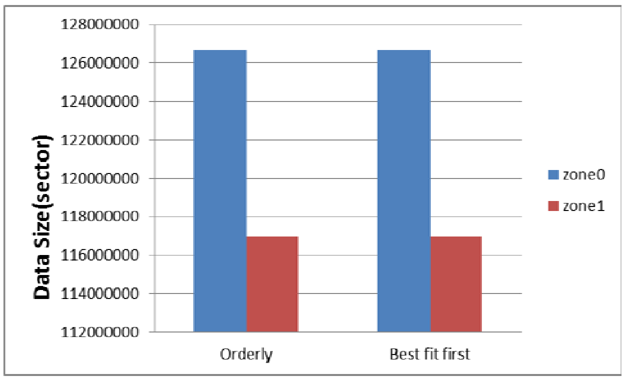

Figure 18. 2 zones in SAN

V. CONCLUSIONS

The experimental results showed that our design offers a new data distribution method. This method can distribute data evenly in every storage devices.

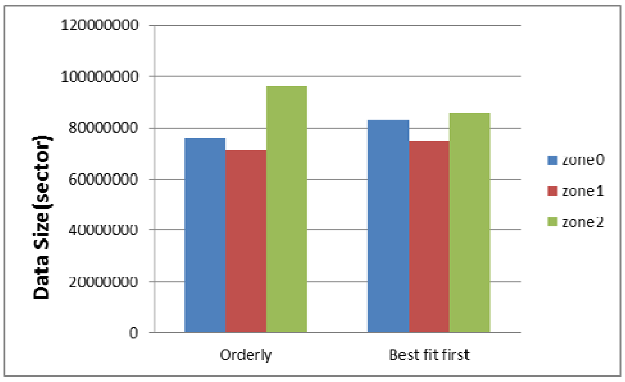

Figure 19. 3 zones in SAN

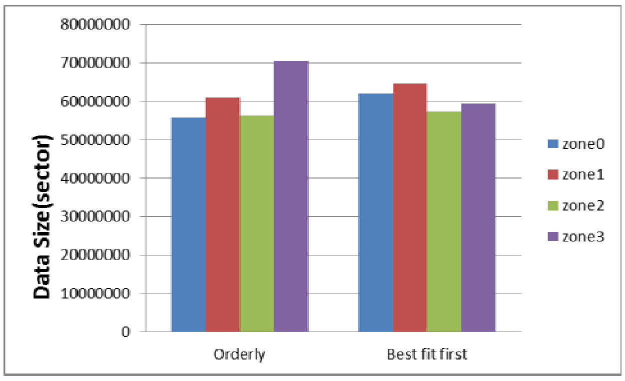

Figure 20. 4 zones in SAN

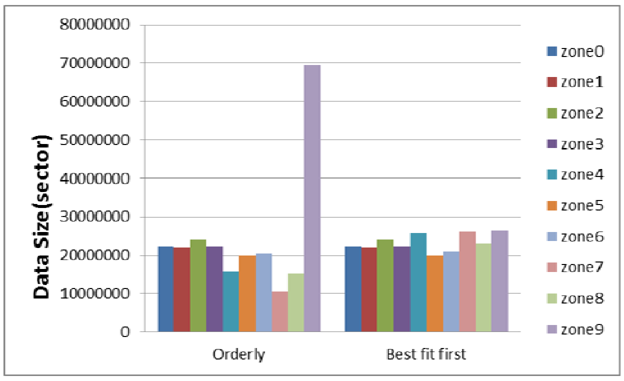

Figure 21. 10 zones in SAN

There are some conclusions from the results:

1) Even distributed trace can work well in arbitrary zones and arbitrary SDs in each zone.
2) If the arrangement of zones and SDs tie in the trace's trend, data can evenly distribute.
3) More zones are better than fewer zones.

REFERENCES

[1] Y. Zhou, Q. Zhu and Y. Zhang, "Spatial Data Dynamic Balancing Distribution Method Based on the Minimum Spatial Proximity for Parallel Spatial Database," Journal of Software, volume. 6, pp. 1337-1344, 2011.

[2] C. Jin, N. Liu and L. Qi,"Research and Application of Data Archiving based on Oracle Dual Database Structure," Journal of Software, vol. 7, pp. 844-848, 2012.

[3] Xianhui Li, Cuihua Ren, Menglong Yue, “A Distributed Real-time Database Index Algorithm Based on B+ Tree and Consistent Hashing”, In International Conference on Advances in Engineering 2011 , Volume 24, pp 171-176, 2011

[4] Zhenbin Yan, Wenzheng Li , “Research of a scheduling and load balancing scheme based on large-scale distributed systems”, in Software Engineering and Service Science (ICSESS), 2012 IEEE 3rd International Conference, pp 22-24, June 2012

[5] Bong Jun Ko, Vasileios Pappas, Ramya Raghavendra, Yang Song, Raheleh B. Dilmaghani, Kang-won Lee, Dinesh Verma, “An information-centric architecture for data center networks”, in ICN '12 Proceedings of the second edition of the ICN workshop on Information-centric networking, pp 79-84, 2012.

[6] Jui-Chi Liang, Jyh-Cheng Chen, Tao Zhang, “An adaptive low-overhead resource discovery protocol for mobile ad-hoc networks”, Wireless Networks, Volume 17, pp 437-452, 2011.

[7] Ricardo Vilaca, Rui Oliveira and Jose Pereira, “A correlation-aware data placement strategy for key-value stores”, in proceedings of the 11th IFIP WG 6.1 international conference on distributed applications and interoperable systems, pp 214-227, 2011

[8] Ramanzan S. Aygun, Yi Ma, Kemal Akkaya, Glenn Cox, Ali Bicak, “A conceptual model for data management and distribution in peer-to-peer systems”, in Peer-to-Peer Networking and applications, Springer New York, volume 3, issue 4, pp 294-322, 2010

[9] “TPC-C blocks i/o trace.” Performance Evaluation Laboratory, Brigham Young University, 2010. http://tds.cs.byu.edu/tds/.

[10] D. Narayanan, A. Donnelly, and A. Rowstron, “Write off-loading: Practical Power Management for Enterprise Storage,” ACM Transactions on Storage, vol. 4, pp. 1–23, Nov. 2008.

Yihua Lan received his Ph.D. degree in Computer Science from the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan(HUST) in 2011, now he held teaching and research positions at School of Computer Engineering, Huaihai Institute of Technology (HHIT), Jiangsu, China. His research areas are image processing and analysis. His research interests include PDE methods for image processing, iterative methods, Krylov subspace methods, optimization algorithms, artificial intelligence, and high performance computer.

Haozheng Ren received the M.S.degree in computer engineering, China, in 2006, from the Lanzhou University of Technology. She is currently a teacher of the School of Computer Engineering, Huaihai Institute of Technology, where she has been an Instructor since 2008. Her research interests include PDE methods for image processing, iterative methods, Krylov subspace methods, and parallel algorithms.

Yong Zhang received the M.S.degree in in School of computer science, SuZhou University in 2007, China. His M.S. subject is digital image processing. He held teaching and researching positions at the School of Computer Engineering, Huaihai Institute of Technology, where he has been an instructor. His research interests include image processing and machine vision.

Chao Yin has received his bachelor degree in chemistry in Huazhong University of Science and Technology in 2001 has received his master degree in software engineer in Huazhong University of Science and Technology in 2005. Now he is a Ph.D. student of grade two in Computer Science in Huazhong University of Science and Technology. His research areas are storage and theory analysis. His research interests include storage, high performance computer, and reliability research and energy consumption.