A NEW METHOD FOR AUTOMATED POOL REBALANCING

Krasimir Miloshev
Solutions Architect
EMC Corp. - TSG NE/Canada

EMC² PROVEN PROFESSIONAL

EMC²

Table of Contents

Disclaimer: The views, processes, or methodologies published in this article are those of the author. They do not necessarily reflect EMC Corporation's views, processes, or methodologies.

Introduction to Automated Pool Rebalancing

Virtual Provisioning (VP) reduces allocated but unused storage and avoids over-allocation of physical storage to applications. This mechanism uses thin storage pools, which reduces the cost of storage, energy consumption, and footprint.

Automated Pool Rebalancing is a feature that balances the used capacity of data devices (when it comes to Symmetrix®) within a thin pool. The operation runs as a background process against an entire thin pool, scans the used capacity of the data devices within a pool, and moves thin extents from the most utilized devices to the least.

- Thin Devices —seen by OS as "normal" device. Physical storage, which is taken from a pool of Data devices, need not be completely allocated at device creation
- Data Devices — internal, non-addressable device, that provides the physical storage that is used to supply disk space for a thin device. It has multiple RAID protection types- RAID5, RAID6, RAID1 for Symmetrix.(RAID1/0 for CLARiiON/VNX).
- Extends — part of the thin devices; Thin device extents for Symmetrix contain 768 KB. Each extent contains 1536 x 512 byte blocks.
- Storage pools — consist of data devices for Symmetrix or simply disks for CLARiiON®/VNX®.

This Knowledge Sharing article explores ways to reduce Automated Pool Rebalancing process time which currently takes many hours to complete.

Using load balancing for relocating extends

In computer science, load balancing is a technique used to spread work evenly between two or more computers, network components, CPUs, hard drives, or, in general, between two or more resources, in order to achieve optimal resource utilization, maximize throughput, and minimize response time.

Consider this example of load balancing; A pool contains a certain number of disks **n** and a certain number of newly added disks **ms.** Each of the current disks has a number of extends, part of which are called extra extends. Assume L_k is the number of extra extends residing on each of those current pool disks, where k=1..n. 'Extra" means those extends that can be moved to "empty" disks (with no extends on them). Each L_k is the amount of extra extends residing on each of the disks 1, 2,….n. Then, $\{L_1, L_2, L_{3...}, L_n\}$ is a set containing all the extra extends. Our aim is to partition this set to ms subsets $P_{1,}$ $P_{2,...}$ P_{ms} thus that we have minimal imbalance between the partitions $L_{p1,}$ $L_{p2,...}$ $L_{ms.}$ The goal is to optimally

distribute the existing extra extends among the newly added disks, thereby minimizing overall pool rebalancing execution time and increasing efficiency.

We will call the number of extends “load”, to make things comparable with the well-known load balancing problem. There is an exact algorithm for solving the load balancing problem, but it is impractical since its time complexity is exponential; it searches the optimal balancing among all possible partitions of the loads (extra extends). We suggest a new heuristic method with an algorithm for extends re-distribution among the newly added disks to the pool. Heuristic methods are used to speed up the process of finding a satisfactory solution, based on an intuitive judgment or common sense. It may not be the optimal solution, but could be the best one from practical points of view. Our particular algorithm is faster than the existing round-robin algorithm and can be applied for a large numbers of disks. The basic idea of that algorithm is to run as many steps (passing through procedures) as ms (ms is the number of newly added disks to the pool) and for each step to distribute an equal number of extends such that the pool experiences minimum load misbalance between the disks.

Consider a storage pool consisting of a certain number of disks **n**. This storage pool can be extended with newly added disks if additional capacity becomes necessary. After extending the storage pool, Automated Pool Rebalancing is run in order to redistribute the load over the existing disks and the newly added disks, thus minimizing imbalance. Extra extends mean the difference between the number of extends on each current disk (member of the pool before adding new disks) and the average number **aver** of extends that we try to achieve. The average number of extra extends among the current disks will be used in our algorithm and can be determined by dividing the sum of all extra extends with the number of newly added disks **ms**. The average number of extends that we try to achieve means that in the ideal situation, after adding new disks (**ms**) to the pool, we have to have an equal number of extends (**aver)** on each of the **n+ms** disks of the extended pool.

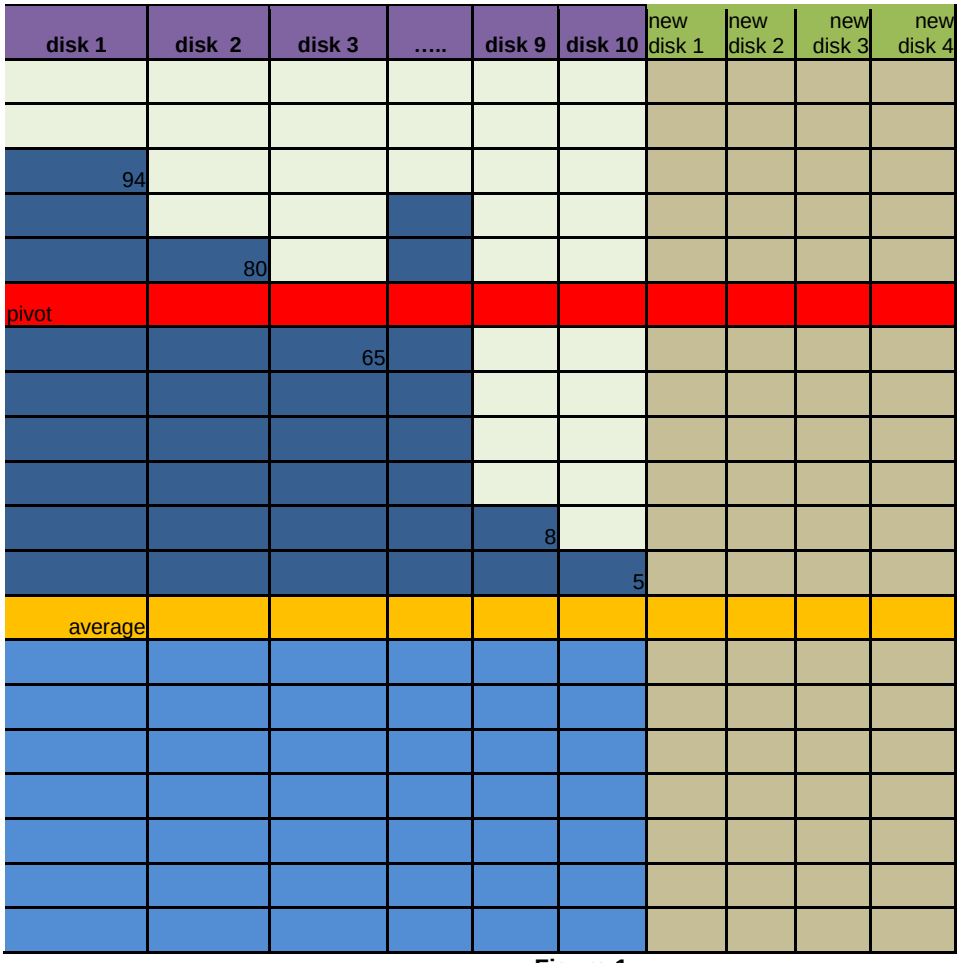

Figure 1

Figure 1 presents both existing disks in a pool and the newly added as well as the number of extends each current disk in the pool has accommodated.

In Figure 1 all the extra extends within the current disks (part of the VP storage pool) are shown in dark blue; extends that will not be moved are shown in light blue; new disks that are to be added to the VP pool are shown in light brown.

$S = \sum_{i=1}^{n} A[i]$ **; piv=S/ms, piv** is the average number of extra extends among the current disks;

$S = \sum_{i=1}^{n} A[i]$; **piv=S/(n+ms), aver** is the average number of extends that we try to achieve after adding some additional new disks (ms) to the pool;

Redistributing the load means redistributing the current extra extends. That is supposed to assure better performance on LUNs level.

If, for example, we have 194 extends on disk 1, 180 extends on disk 2 …, 105 extends on disk 10 , and the average number **aver** after adding new disks to the pool is 100, the

number of extra extends on disk 1 will be 94, on disk 2 will be 80,…., and on disk 10 will be 5. We will try to redistribute those extends among the newly added disks. The extra extends are presented in Table 1.

A fast heuristic algorithm for redistributing extends within thin storage pools

Let us name the new algorithm APR. Input data for APR are the current pool disks **n** with their extra extends and the number ms of new disks, which will be added to the existing storage pool. The number of current disks **n** and their number of extra extends are presented via the **A** array. **Mserv_load** is an array where we will keep the total number of extends for each of the disks after applying the method. **Mserv_index** is an array where we will keep track of the redistributed extends.

1. Input data for the APR algorithm:

    n - number of current disks

    ms - number of newly added disks

    A [n] - an array containing the number of extra extends for each of the current disks

2. Output data for the APR algorithm:

    Mserv_load [ms] – an array of ms elements, where each element contains a certain number of extends that have been allocated to the newly added disks;

    Mserv_index [ms] – an array of ms elements, where each element contains the indexes of the extra extends that have been allocated to the newly added disks.

Algorithm APR:

Step 1 $S = \sum_{i=1}^{n} A[i]$; piv=S/ms, piv is the average load among the extra extends for the current disks in the pool.

Step 2 Sorting all the elements of A in decreasing order using a sorting algorithm. Of course, the general purpose sorting algorithm—such as quick-sort—can be used in any case.

Step 3 In the beginning, we mark each element of *A* as unused by setting 0 for each element of the array named used. Initiating Mserv_index[k] and Mserv_load[k]. We assign A[1] to the first available resource, A[2] to the second, and A[ms] to the ms-one.

Thus for k=4 we have:

Mserv_load[1]=A[1] and Mserv_index[1]={1},

Mserv_load[2]=A[2] and Mserv_index[2]={2},

Mserv_load[3]=A[3] and Mserv_index[3]={3},

Mserv_load[4]=A[4] and Mserv_index[4]={4}.

Step 4 Building up Mserv_load[k] and Mserv_index[k] by passing through all elements of these arrays. This is a loop with control variable k=1, 2, 3,ms, whereas ms is the number of so called extra extends. For each element Mserv_load[k], we go through all elements of the A array (this is an inner loop with control variable j=ms+1, ms+2, ms+3, . . . , n) to update the current Mserv_load[k] and Mserv_index[k]. For each unused element of A[j], we determine temp[j] (the current load Mserv_load[k]) and delta[j]=temp[j]-piv, where delta[j] is the difference between the current load and the average load, where the load is actually related to the number of extends.

If delta[j] > 0, this unused element of A is not picked up and it is not included in Mserv_index[k]; if delta <= 0, this unused element gets included in Mserv_index[k] and Mserv_load[k].

During the pass through all the elements of the array A, minimal absolute positive and absolute negative differences among all the delta[j] values are determined. We pick up the smaller one and based on that value, we determine the selected elements which are included in Mserv_index[k]. Essentially, we can say that our selecting criterion would be the minimum among all absolute values of delta [j].

Step 5 Print out all the elements of Mserv_load and Mserv_index to get the load amount and what extra extends were assigned for each of the newly added disks. Actually, each member of the Mserv_index array represents a set of elements, and each element of that set shows a specific newly added disk.

Example of how the algorithm works:

Let us have the ms=4 newly added disks; n=10 existing current disks and the following values (number of so called extra extends) for each of the disks in the pool: 80,25,12,5,84,65,43,17,32,8.

i	1	2	3	4	5	6	7	8	9	10
A[i]	94	80	65	43	32	25	17	12	8	5

Table 1

First, we can determine S=380 (number of extends) and pivot=380/4=95. Then after sorting the A array elements in decreasing order, we get an updated array A, as shown in Table 1.

A. For the first pass of the algorithm (k=1) and Mserv_load [1]=A[1]=94, we get the following values for temp[j] and for delta[j], whereas delta[j]= temp[j] – pivot, shown in Table 2.

temp [j]	delta [j]
94	-1
126=94+32	31
119=94+25	24
111=94+17	16
106=94+12	11
102=94+8	7
100=94+6	5

Table 2

From Table 2 we can see that min |delta[j]| = 1, therefore;
Mserv_load [1]=94 and Mserv_index [1]={1}

B. For the second pass of the algorithm (k=2) and Mserv_load [2]=A[2]=80, we get the following results as shown in Table 3:

temp [j]	delta [j]
80	-15
112=80+32	17
105=80+25	10
98=80+17	2
92=80+12	-3
100=80+12+8	5
98=80+12+6	3

Table 3

C. For the third pass of the algorithm (k=3) and Mserv_load[3]=A[3]=65, we get the following results as shown in Table 4:

temp [j]	delta [j]
65	-30
97=65+32	2
90=65+25	-5
102=65+25+12	7
98=65+25+8	3
95=65+25+5	0

Table 4

Here we have min Idelta [j]I=2, thus we will have Mserv_load [2]=80+17=97 and Mserv_index [2]={2,7}. The used element 17 should not be included in our next passes. We

have min |delta[j]| = 0 for the last sum, thus we will have Mserv_load [3]=95 and Mserv_index [3]= {3,6,10}. The used elements are 65,25,5.

D. For the last pass (k=4) and Mserv_load [4]=A[4]=43, the non-used elements are left, thus we will have:

Mserv_load [4]=43+32+12+8=95 and Mserv_index [4]={4,5,8,9}

We have distributed extends between all four newly added disks as follows:

1. For our first disk, we have assigned extra extends from the current disk 1 with total load=94 (number of extra extends).

2. For our second newly added disk, we have assigned extra extends from the current disk 2 and extra extends from the current disk 7 with total load 80+17=97 (number of extra extends).

3. For the third newly added disk, we have assigned extra extends from the current disk 3, extra extends from the current disk 6, and extra extends from the current disk 10 with total load of 65+25+5=95 (number of extra extends).

4. For the fourth newly added disk, we have assigned extra extends from the current disk 4, extra extends from the current disk 5, extra extends from the current disk 8, and extra extends from the current disk 9 with total load 43+32+12+8=95 (number of extra extends).

We have gotten 97-94=3 as a difference between the maximal load and the minimal load, which would be the load imbalance. Clearly, 3 is an extremely small imbalance number, making this algorithm close to optimal. Also we have equal load of 95 evenly distributed extra extends among the two other newly added disks, which makes this procedure quite precise. Obviously the APR algorithm time complexity without the time for sorting is T=O(ms n), where ms is the number of newly added disks and n - number of existing pool disks, because we have ms iterations of the outer loop and on each iteration we have n processed elements (inner loop). If we add the time for counting, then T= O(ms.n)+O(b+n)=O(ms.n) (b is an integer).

Conclusion

The provided APR algorithm solves the Automated Pool Rebalancing task by using fast extra extends redistribution. The method is based on distributing an even number of extends among the disks in the “old” pool and creating a maximally balanced distribution of extends over the newly added disks. So, in practice, we have created ideal load distribution on the bigger existing pool and close to ideal load distribution on the small “additional” pool, such

that, after the extension, the two pools combined create one unified almost perfectly balanced pool. This approach is typical for the heuristic methods.

APR has been implemented as a program and tested with different n (number of current disks) and ms (number of newly added disks). As ms grows, the imbalance becomes lower and lower with small variations. Also the higher numbers of n we have, the better extra extends redistributions we will get, making this algorithm perfect for large spools containing hundreds of disks. Thus, our heuristic algorithm works optimally for large numbers of pool disks and newly added disks.

Table 5 shows some practical results after running a program implementing this algorithm. Our example is for the run time (T *1000) and imbalance for a cases with n=100 to 1200, ms=4 and 6, and randomly generated array A containing extra extends. The run time is received by 1000 times execution of the program for each n since T cannot be measured (is too small – mostly 0 on one execution). Figure 2 show the graphics of the run time from Table 5; they confirm the theoretical time complexity O (ms. n) – linear at fixed ms.

N	100	200	300	400	500	600	700	800	900	1000	1200
T*1000 sec(k=4)	0.16	0.33	0.55	0.71	0.88	1.10	1.26	1.43	1.65	1.81	2.25
Imbalance (ms=4)	32	5	19	4	1	4	6	0	1	5	1
T*1000 sec(k=6)	0.27	0.49	0.77	1.04	1.32	1.59	1.87	2.14	2.42	2.69	3.24
Imbalance (ms=6)	34	21	7	3	5	5	3	2	1	5	4

Table 5

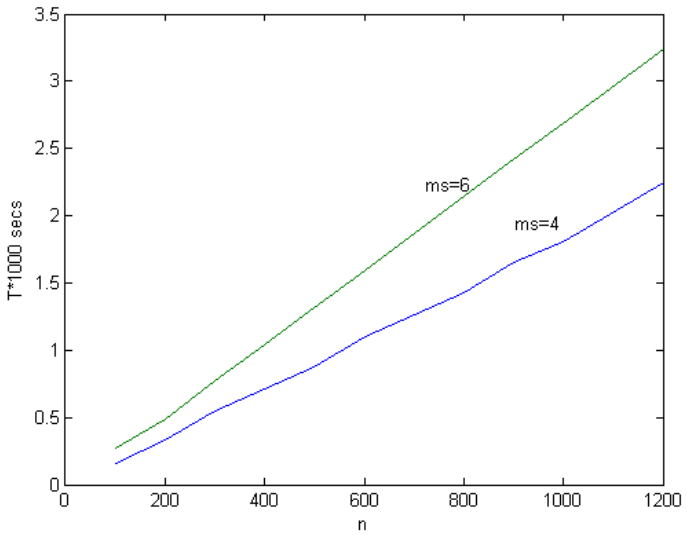

Figure 2

References

[1] Barnes M. Efficient generation of Graphical partitions, Disc. Appl. Math. 78: pp.17-26, 2003
[2] Bourke T. Server Load Balancing, O'Reily Media Inc., 2002
[3] Garey, Michael R. and Johnson, David S. "A 71/60 theorem for bin packing", Journal of Complexity, Vol. 1, pp. 65–106, 1985
[4] Gyori, Ervin More Sets, Graphs and Numbers, Springer, 2000
[5] Mertens S. Number partitioning, URL http://arxiv.org/ftp/cond-mat/papers/0310/0310317.pdf, 2003

Appendix

APR Algorithm Code (C++):

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
typedef int t_element;
const int nmax = 10000, key_min = 0, key_max = 2001,kmax=4,maxint=100000;
void bucket_sort( t_element a[], t_element pom[], int n, int kmin, int
kmax1 )
{       int kliuch, broi, i;
        for( kliuch = kmin; kliuch <= kmax1; kliuch++ ){
                pom[kliuch] = 0;
        }
        for( i = 1; i <= n; i++){
                pom[a[i]] = pom[a[i]] + 1;
        }
        i = 1;
        for( kliuch=kmax1; kliuch>= kmin; kliuch-- ){
                for( broi=1; broi<=pom[kliuch]; broi++ ){
                        a[i] = kliuch;
                        i++;
                }
        }
}

void main( void )
{int n=10000 , index,i,k,jpmin,jomin,j,jj;
double temp,piv,delta,dpmin,domin,s,snova,mmax,mmin,
 server[kmax+1];
        t_element  a[nmax], pom[key_max],used[nmax];
        while( n > 9999 ){
                cout << " Insert the number of elements of the array
(<1000): ";
                cin >> n;
        }

/*      for( index = 1; index <= n; index ++ ){
                while( a[index]<1 || a[index]>100 ){
                        cout << " enter element (>0&<101) : " << index <<
" : ";
                        cin >> a[index];
                }
        }*/
        int r1,r=2000;
        srand(r);
        r1=rand();
        for (i=0;i<=n;i++)a[i]=rand()%2000+1;

        bucket_sort( a, pom, n, key_min, key_max );
        cout << "The sorted array is : \n";
   /* for( index = 1; index <= n; index ++ ){
                cout << " " << a[index] << " :";};
                cout<<endl;*/

 s=0;
 for (i=1; i<= n;i++) { s=s+a[i];used[i]=0; };
 piv=s/kmax;
 for (k=1; k<=kmax-1;k++)
```

```
 {
   delta=a[k]-piv;
   server[k]=a[k];
   if (a[k] >=piv) goto e10;
   dpmin=maxint;
   domin=-maxint;
   if (delta>=0) { dpmin=delta;
    jpmin=k;}
   else { domin=delta;jomin=k;used[k]=-k; };
   temp=a[k];
   j=kmax+1;
   while (j<=n)
   {
     if (used[j]==0 )
         {temp=temp+a[j];delta=temp-piv;
       if (delta>=0)
        //then
           { if (delta==0){ server[k]=piv;used[j]=k;goto e10; };
           if (delta<dpmin){ dpmin=delta;
            jpmin=j; };
           temp=temp-a[j];
           }
        else
                { if (abs(delta)<abs(domin)) { domin=delta;jomin=j ;};
          used[j]=-k;
                }
        //{end of if delta}
   }//{end of the first if};
     j=j+1;
          };//{end of the  second loop - on j}
   if (abs(dpmin)<=abs(domin))
   { server[k]=piv+dpmin;
        used[jpmin]=k;
        for (jj=jomin; jj>= jpmin+1;jj--)
               if (used[jj]=-k)used[jj]=0;
                       else server[k]=piv+domin;}
e10: ;
 };//end of the  first loop - on k
server[kmax]=a[kmax];
for (j=kmax+1; j<=n;j++) if( used[j]==0) server[kmax]=server[kmax]+a[j];
//{writeln; writeln('s=',s,' pivot=',piv);}
snova=0;
mmin=server[1];mmax=mmin;
for (k=1;k<=kmax;k++ )
{
  //{write(' k=',k, ' ',server[k]);}
snova=snova+server[k];
  if (server[k]>mmax) mmax=server[k]; else if (server[k]<mmin)
    mmin=server[k];
 //{writeln;writeln('snova=',snova);}if snova<>s then writeln('error');
 //{writeln('misbalans=',mmax-mmin)}
};
// finish(t);
// report('t=',t);
 cout<<"s="<<s<<" pivot="<<piv<<endl;
 cout<<"snova="<<snova<<endl;
 cout<<"misbalans="<<(mmax-mmin)<<endl;
}
```