Dynamic Copysets Merging

Kaige Zhang, Tao Luo, Zhicheng Wu

Abstract

Cloud storage systems use replica to prevent data loss. Different replication algorithms affect the possibility of losing data. Copyset[1] and Tiered Replication[2] propose novel methods to significantly reduces the frequency of data loss events. However, they are not performed well in dynamic cluster due to the side effect of increasing copyset number. We present a merging algorithm which can be run periodically and helps to reduce the number of copysets. This algorithm deals with dynamic change and introduces trivial overhead to performance.

1 Introduction

Currently, we have several choices when performing replication. The most common and widely used algorithm is Random Replication. However, it is almost guaranteed to lose data due to cluster-wide power outages. Random copyset replication[1] use a static algorithm which is not good for practical use. When the cluster changes, the whole generation changes. It means all of the data have to migrate. Tiered Replication[2] use greedy algorithm to generate copyset groups.

In real case, the changing number of nodes in the data center will mess up with the generated groups. The increasing replication groups cause the increasing of the possibility of data loss. This side effect is hard to be avoid during the generation. That's why we have to do an additional merging to optimize the generated result.

2 Problem formulation

Data availability is the most important consideration of cloud computing. Comparing to existing replication generation algorithm, it turns out that greedy algorithm is suitable to practical cases. However, we found that with long time running of this algorithm in a dynamic cluster, the number of copysets grows quickly than direct generation.

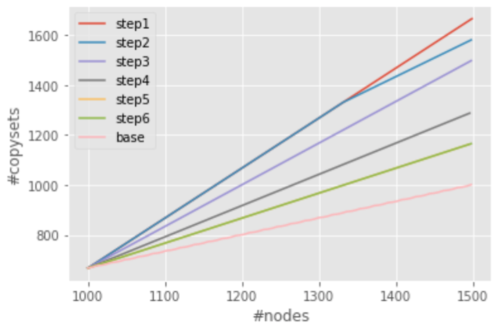

Figure 1: Scale up with different steps

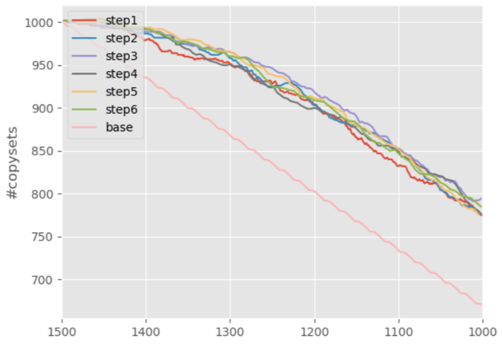

Figure 2: Scale down with different steps

We test tiered replication with different changing steps shown in Figure 1 and Figure 2. Each step means we add or get rid of a certain number of nodes to simulate the scaling up and down cluster. Comparing to direct generation as base, the number of copyset grows much more faster especially in a small step. Generally, more replication groups means higher possibility of data loss.

There is a trade off between the number of replication group, recovery time and data loss probability. To achieve optimal performance, all of the replication variables needs to be tuned with the practical case.

3 Design

There are two general approaches to conquer this ramping Copysets problem. The first approach needs to carefully deal with additional copysets structure for each step of membership change. In this way, one can optimize for the copysets amount after each step while preserving previous copysets structure. Tiered replication takes this approach. Nevertheless, after several steps of membership change, aggregated copysets generated by those per-step optimal copysets are not optimal any more, especially compared to static generated copyset. The second approach requires proactively move replicas around to reduce the coypsets of a cluster. Given an arbitrary copyset structure, we need a scheme to gradually merge those copysets such that the resulting copyset structure has less copysets than before. It works like a garbage collector in the background. We take second approach, and our simulation shows the resulting copysets are near offline optimal.

3.1 Intuition

In order to convert a copysets structure into an offline optimum, our starting point is two helpful observations from original offline copysets:

1. Each node belong to exactly P copysets.
2. Any two copysets has at most one overlapping nodes.

The general idea of our algorithm is to identify nodes and copysets that breaks above two patterns, and reallocate relevant replicas in a way that reduces copysets amount.

3.2 Copysets merging algorithm

The algorithm we propose consist of two steps:

1. we conceptually kick over-scattered nodes out of its copysets until each node only belongs P copysets.
2. we reduce the copyset amount by merging half-full copysets into full-sized copysets, where the full-sized copyset is determined by replication factor.

The rational of Step 1 comes from Observation 1 - each node only belongs to exactly P copysets to be optimal. Note that after this step, copysets amount remains the same. For copysets that nodes left from, the replication factor requirement is broken, and it depends on next step to fix this issue.

In Step 2, matching those half-full copysets is the essential problem of merging process. The simple case is two half-full copysets can fortunately complement each other and form a full-sized copyset. Otherwise, we need to convert to the simple case matching by splitting the half-full copyset into two half-full copysets. After splitting, one of splitted copyset can complement with another un-splitted half-full copyset and form a full-sized one. We then leave the remaining splitted copyset to match with other half-full copyset later.

The details of Step 2 are as follows. To systematically address this matching problem for all half-full copysets. we first sort all half-full copyset by their sizes, and merge from two ends - starting from the current biggest copyset and merge with smallest one(s). If matched copysets can only form a copyset bigger than full-size (e.g. 3), we merge them into one full-size copyset and leave another splitted half-full copyset for later matching. For example, given replication factor equals 3, we can match size-2 copyset with size-2 and merge into a size-3 (full-sized) copyset and leave a size-1 copyset for later matching.

After Step 2, we reduce the copyset amount by merging previous half-full copysets into a full-size new copyset. However, previous half-full copyset is under-replicated in the newly formed copyset. This isssue is addressed by re-replication. Specifically, within a new copyset, for each node that a replica doesn't present, we re-replicate that replica. In this way we can satisfy the requirement of replication factor that Step 1 breaks.

4 Evaluaiton

We run several simulations and experiments to testify the correctness of the algorithm and discuss the cost of algorithm in actual system.

4.1 Simulation

After figuring out how the merge algorithm should work, we decided to run the simulation at first, in order to find out more features about the algorithm and how it will perform on different cluster setups.

At first, we followed the greedy approach mentioned in Tiered Replication[2] for statically generating the minimal number of copysets that can meet the requirement. During reviewing this simple but effcient method, we find that there is still some room to improve the performance of algorithm, which means generating less number of copysets. The greedy approach sort all nodes based on their current scatter width and choose first R nodes to form a new copysets, by repeating these steps until the first node has scatter width equal or larger than the required one we can get the copysets setup. We improved this algorithm by adding a ”in-group penalty” to the others nodes which are in same copysets as the chosen node. This will reduce the possibility of choosing highly overlapped copysets and thus a better result as shown in Figure 3.

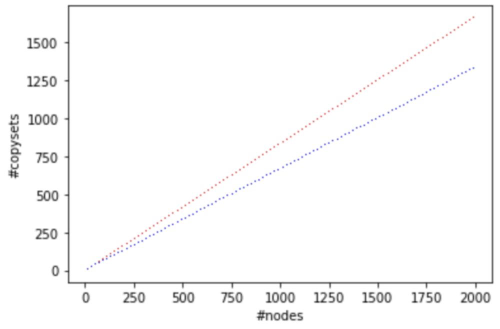

Figure 3: Improved greedy approach, blue line is the one with in-group penalty.

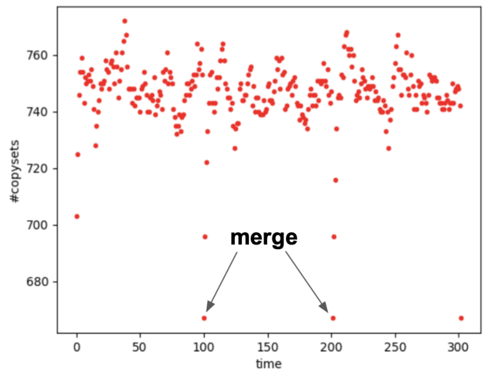

Figure 4: Merge invoked periodically

We use the improved greedy approach to statically generate copysets as the optimal setup. Then we simulate the cluster changes and execute merge copysets algorithm along with it. At first, we tried the cluster setup with 1000 nodes and join then leave one percent of the nodes after each step. The merge is called periodically and the simulation result shows that merging can actually restore the number of copysets very close to the optimal one(statically generated) as shown in Figure 4.

4.2 Observations

From the result of simulation, we can see that the cluster takes more than 5 steps to reach the worst upper bound. As the number of merged copysets grows, merging will need to migrate more data to finish the process. Therefore, we argue that if we execute merge as early as possible when cluster nodes change, actual data migration cost will be limited to minimal as well. Furthermore, as a result, by following this aggressive execution policy, we can keep the number of copysets always optimal in dynamic situation and only brings trivial influence on system performance.

Assuming data is uniformly distributed across all nodes in a cluster, the size of data needs to be migrated/dispatched for each node can be computed as:

$$size = \frac{N_{migrated\_replica} * S_{replica}}{N_{nodes}} \tag{1}$$

The number of replicas that will be migrated can be derived from the number of copysets that are eliminated by merging and the size of each replica can be roughly calculated under uniformly distributed data assumption.

$$N_{migrated\_replica} = N_{reduced\_copysets} * R \tag{2}$$

$$S_{replica} = \frac{S_{all}}{N_{copysets} * R} \tag{3}$$

Finally we can present the estimated cost as the proportion of data that each node needs to send to the target node.

$$data\_prop = \frac{N_{reduced\_copysets}}{N_{copysets}} \tag{4}$$

As we can see from the estimated cost, earlier we execute the algorithm, less data need to be transferred and thus less resource need to be committed to merge algorithm in order to achieve our goal.

4.3 Experiment

As we have mentioned in the observations, there exits an upper bound for the increasing number of copysets. This is intuitive because adding nodes in greedy approach will generate redundant copysets while removing nodes from cluster could eliminate those redundant copysets associated to the removed nodes. In our simulation, the number of reduced copysets is at most 15% of the static number.

We leveraged Upgrade Domain feature and Balancer implementation in Hadoop 2.10.0 to understand the influence of moving data around multiple nodes on overall throughput. After modifying Upgrade Domain, it allows Hadoop HDFS to have an imbalanced data distribution by enforcing some nodes in special domain to reject receiving data from client and any other nodes. The current Balancer implementation in Hadoop has configuration `maxBalancerBandwidth` to limit Balancer's impact on the whole system. We set up a small cluster on CloudLab and hardware for each node is listed as followed: Eight-core Intel Xeon D-1548 at 2.0 GHz, 4x 16 GB DDR4-2133 SO-DIMMs, 256 GB NVMe flash storage. A Dual-port Mellanox ConnectX-3 10 Gbps NIC groups all nodes. Hadoop HDFS is set with replication factor as 2.

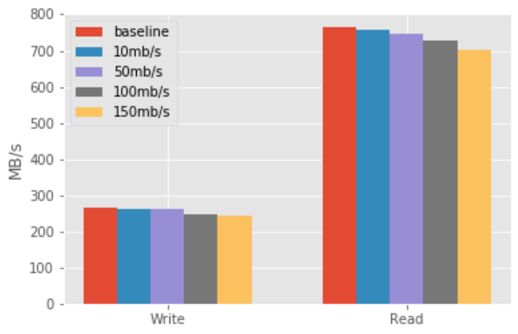

Figure 5: Write/Read throughput with Balancer

We ran TestDFSIO with different setup of Balancer: 1)No Balancer, 2)Balancer with bandwidth 10MB/s(default), 3)50MB/S, 4)100MB/s, 5)150MB/s and the result is shown in Figure 5. The result shows with 150MB/s bandwidth for moving data, which is already close to the actual write throughput of HDFS, Write and Read throughput has only degraded 8% compared to the baseline.

Following up the previous cost estimation, a full copysets merge execution on a cluster with each node's capacity 500GB and 50% capacity utilization, it takes at most 3.1 hours to finish one step of merge even with 10MB/s bandwidth limit. Compared to the frequency of nodes number change, doing merge algorithm aggressively incurs only marginal overhead to the system and meanwhile the number of copysets is kept roughly as same as the static one.

5 Future work

5.1 Implementation

An apparent future work of this project could be implementing our algorithm in existing distributed storage system. We found HDFS fits this work very well. Because HDFS has a well-organized source code, we only need to 1) implement a new replica placement policy inherited from HDFS's default placement policy and put greedy placement logic inside; 2) implement a new rebalance class that can move replicas following our merging algorithm.

5.2 Algorithm improvement

5.2.1 Load balance

Our algorithm doesn't consider the load balance of cluster. Specifically, problems of workload balance and capacity balance remain open. After merging copyset, it is likely several separate hot replicas now get moved into same copyset, creating overloaded node. The capacity concern is raised since new copyset need to accommodate several half-full copysets. If half-full copysets have large data size, they are not likely to fit in a new copyset.

Fortunately, this copysets merging algorithm still has design space to explore, both in Step 1 and Step 2. For example, We can deliberately choose light loaded copysets to leave in step 1, we can also consider the resulting new copyset's load while matching half-full copysets in Step 2.

Even if those two approaches fail to balance load, nevertheless, we can do copyset-level rebalance by moving some replicas outside of new copyset.

5.2.2 bandwidth scheduling

It is desirable to has minimal bandwidth interference with foreground IO requests while moving replicas from original copysets into merged one. Although our experiment on small-scale cluster shows this algorithm has negligible IO performance overhead on foreground tasks, for extreme large-scale cluster with dynamic and imbalanced foreground traffic, scheduling all replica migration tasks in a given time period remains challenging. We can borrow the idea from Dayu[3]. The general idea is to schedule a subset of all replica migration tasks in a given time slot and dynamically adapt each time-slot's migration task to current foreground traffic.

6 Conclusion

This report introduces a merge algorithm to address to problem of Copyset Replication in dynamic cluster environment. Previous works show Copyset Replication has capability of preventing correlated failure but is not flexible enough to support dynamic nodes change, thus difficult to implement it in most of the distributed storage system. By merging copysets, we show that Copyset Replication also can deal with dynamic change and introduces trivial overhead to performance. We believe after more thorough considerations with the algorithm, Copyset Replication will become a more practical policy for distributed storage community.

7 Acknowledgements

We would like to thank professor Asaf Cidon for valuable suggestions and generous help while taking his seminar course.

References

[1] Cidon et al. Copysets: Reducing the Frequency of Data Loss in Cloud Storage

[2] Cidon et al. Tiered Replication: A Cost-effective Alternative to Full Cluster Geo-replication

[3] Zhufan Wang et al. Dayu: Fast and Low-interference Data Recovery in Very-large Storage Systems