Measuring and Improvig Geo Distributed Storage

CS838 Fall 2016 Project Report

Karan Bavishi
UW Madison

Hasnain Ali Pirzada
UW Madison

1. PROBLEM STATEMENT

HDFS and other distributed file systems are designed to work in a single data center and do not scale well to a geo distributed environment. The primary reason for this is the heterogeneity of the the inter and intra datacenter links and the fact that the file system itself is unaware of the thin links that connect multiple geo locations. One major way to overcome these problems is to use Erasure Coding instead of replication since erasure coding produces much less amount of data compared to replication and as a result the network and storage overheads are reduced. To solve these problems our goal in this project was two fold. Our first target was to quantify and analyse the performance degradation of HDFS (both replication and erasure coding based) running geo distributedly. Depending upon our findings of this, our second major goal was to reduce the impact of thin WAN links on the performance of geo distributed HDFS. We achieved these goals by first benchmarking both HDFS and HDFS Erasure Coding (HDFS-EC) running geo distributedly and compared their performance with the same system running in a single datacenter. In the second phase of the project we modified HDFS-EC and used greedy heuristics to make it WAN aware while performing both reads and writes.

Keywords

HDFS; Erasure Coding; Geo Distributed Storage

2. SYSTEM SET UP

The first part of our project was to quantify the degradation in HDFS and HDFS-EC performance in Geo Distributed (GD) settings. To do this, we set up a cluster of machines placed in different geo locations , with HDFS running on top of them. We needed to chose the number of machines and the replication factor in such a way that at least one replica is guaranteed to be on a datanode in another DC so that the WAN links are brought into play.

To be able to measure the extent of degradation in GD settings we also needed to run the same set of experiments in a non GD environment. Furthermore, to be able to compare the extent of degradation in replication based scheme vs the erasure coding scheme we needed to run the same set of experiments on HDFS with replication as well as HDFS-EC in both GD and non GD environment. So in total we had four different settings which are described and explained as follows:

1. **HDFS-Replication based** HDFS running with replication factor 3 with three datanodes in Wisc CloudLab. This is the base case for measuring and compare the GD degradation in replication based HDFS. There is no remote node since there is just one datacenter.

2. **HDFS-Replica-GD** HDFS running with replication factor 3 with two datanodes in Wisc CloudLab and one datanode in Clemson. A replication factor of 3 here with 2 datanodes in Wisconsin and 1 in Clemson ensure that one replica of each block is located remotely.

3. **HDFS-EC** HDFS running with erasure coding scheme Reed-Solomon (3,2) with five datanodes in Wisc CloudLab. HDFS-EC fork that we used required at least n + r datanodes in the cluster while using RS(n,r) erasure coding scheme. This is done to ensure that stripes cells are well distributed across the cluster since the number of failures that can be tolerated here is equal to r i.e., 2 in this case. A good distribution of the stripe cells ensures better fault tolerance. Hence, for this configuration we require $3 + 2 = 5$ datanodes. All of them are in Wiscnosin.

4. **HDFS-EC-GD** HDFS running with erasure coding scheme Reed-Solomon (3,2) with four datanodes in Wisc CloudLab and one datanode in Clemson. The reason for choosing 5 nodes in total is same as the above one. However, now four of the five datanodes are in Wisconsin and one is in Clemson forcing every write to involve the WAN.

The WAN speed for all the above configurations is measure to be around 1 Gbps while the intra DC links are all 10 Gbps. This high difference in the two link speeds allows us to more realistically observe the effect of WAN link degradation. We rigorously evaluated all the HDFS variants described using the TestDFSIO workloads. The details are in the experiments and results section.

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

3. SYSTEM DESIGN & IMPLEMENTATION

Drawing insights from the results (which we describe and explain in the results section) of our performance benchmarking of HDFS in GD settings (both replicationa and EC based) we implemented WAN awareness in HDFS-EC and tuned both the read and write pipelines to take WAN into account while deciding on the datanodes for read/write for the HDFS client.

3.1 WAN awareness for HDFS-EC Reads

Let us say we have a file stored in HDFS-EC running atop a cluster which has nodes in two datacenters A and B. The file is stored as a stripe of cells (a picture here maybe?) with some cells representing actual data and others representing parity. Further assume that the file has 3 cells of data which would result in a total of 5 cells written in HDFS 3 being the data cells and 2 being the parity cells. Assume that datacenter A has 2 data cell and one parity cell, while datacenter B has 1 data cell and one parity cell. Now, there HDFS-EC client in datacenter A who wants to read a file. The default policy is to read all the data cells. The parity cells are only read when the data cell is unavailable because of the high computation cost of reconstructing the data from parity.

In a single data center circumstances, this makes sense because the computation cost is very likely to exceed the network cost that is paid to read a data cells from a remote node. However, in a GD cluster, this may not be true. So in this case, depending upon the current WAN cost it might be faster to actually read 2 data cells and 1 parity cell locally and reconstruct all the three datacells in datacenter A.

To achieve this, we associate costs with both the computation of the data cell and reading it over the WAN and the client greedily chooses the whichever one has the lower cost. We start with simple static costs for the WAN links and computation which we hard code at the beginning. However, both of these can be made a function of current cpu load of the client and the current WAN load.

3.2 WAN awareness for HDFS-EC Writes Using Greedy Heuristic

The write policy in HDFS-EC is similar. The namenode selects (r + s) nodes for writing r data cells and the corresponding s parity cells. It then writes these cells in round robin way to all these datanodes. This is done to avoid the loss of multiple data cells in case of a datanode failure. However, while choosing the datanodes to store data and parity cells, the namenode can select a datanode in the other datacenter despite having a datanode locally. This would involve the the write to go through a WAN.

To prevent this situation, we again implement a simple cost model in the HDFS-EC write pipeline. We associate a parameter same-rack-penalty with every write. Moreover, we have link costs available between every pair of racks in the topology. Whenever an HDFS-EC client wants to write a file, the namenode sorts the available datanodes according to their distance from the writer. Now, for each block to be written, it picks the datanode with the minimum cost and remembers the rack from which this datanode was picked. When choosing a datanode for the next block to be written it picks the datanode with the minimum cost with the added constraint that if another datanode has already been chosen from the same rack the total cost of this node becomes y and if this total cost is still the minimum of the cost of all the available data nodes this datanode is chosen. Now, for any subsequent iterations, the cost of this datanode would be (linkcost + n * samerackpenalty) where n is the number of times a datanode from this rack has been chosen for this client.

How does this scheme help?

This method of selecting the datanodes for writes has a direct affect on the WAN usage. Since the cost of any datanode in datacenter B is always much higher, the namenode will select the write locations which are closer to the writer reducing the WAN usage. A side effect of this would be that the data would be aggregated near the producers. For large enough files however, the datanodes in the remote datacenter would eventually be selected since penalty for a given rack increases linearly with the number of blocks that have already been written in the same rack so at some point selecting a rack in the same datacenter would have more cost than writing to the remote one.

4. EXPERIMENTS & RESULTS

Our experiments and results for this project are divided into two parts. In the first part we describe and explain the the benchmarking experiments we performed to measure the degradation of performance of HDFS and HDFS-EC in GD settings. In the second part we discuss the results of our WAN aware HDFS-EC.

5. DISCUSSION

5.1 Erasure Coding Properties and their effect

There are various kinds of encoding families that can be used to encode data. These include RS codes, LRC codes and the product codes. Since HDFS-EC used RS encoding we will restrict our discussion to tradeoffs of using different values k and r in RS encoding. According to the coding theory a (k, r) RS code entails the minimum storage overhead among all (k, r) erasure codes that tolerate any r failures [cite hitchhiker]. Moreover, RS codes can be constructed with arbitrary values of k and r depending upon the system at hand [cite hitchiker]. These two properties make RS codes very attractive for use in large scale storage systems.

The choice of the parameters k and r for any particular system determines the storage overhead and recovery cost for the system. In general achieving lower storage overhead increases the recovery cost of the system and vice versa. For Example, an RS(10, 4) system has a 1.4x overhead compared to 1.5x for RS (6, 3). However, in case of an unavailable datablock RS (10, 4) has to read 10 blocks while RS (6, 3) only needs to read 6.This trend can be generalized for any possible values or k and r.

5.2 Significance of Storage Unit to Encode

The storage unit to be encoded is determined by the layout of the file system. This layout can be contiguous or striped. In contiguous layout a logical block of the file system is mapped directly to a datanode. Reading that block merely involves contacting that particular datanode and doing one I/O. In striped block layout however, a single logical block in the file system is broken down into small cells and these cells are then mapped to the available datanodes in a round robin manner. Therefore, in the striped layout case,

reading a single logical datablock of the file system involves doing I/O with all the datanodes that store the cells of that block.

In the replication based storage, a contiguous layout seems to be an obvious choice as there is no point in paying an extra cost of many remote reads for every read event. However, in Erasure Coded storage, things become a little more interesting. This is because the number of parity blocks computed in RS encoding scheme is always equal to r irrespective of the total number of data blocks in the file [cite blog]. For example, while using RS (10, 4) if a file consists of just one data block, it will still end up computing and writing 4 parity blocks resulting in a total storage overhead of 400% which is worse than 3x replication! However, in a striped layout and assuming a cell size of 1 MB and the HDFS data block size of 128MB the same 1 block file would still have a 1.4 x overhead since it will produce about 51 MBs of parity (since the parity is computed for 1 MB stripe cells) compared to 512 MB of parity produced by the contiguous layout for the same 128 MB file.

Therefore, the type of file system layout directly affects the storage overhead for the erasure coding scheme at hand. This choice of layout in turn is determined the file size distribution on the cluster. Hence, a cluster with many small files is not suitable for erasure coding in the contiguous layout because the storage overhead would be worse than replication. The HDFS-EC alpha release that we used and extended uses a striped layout with 1 MB cell size.