Evaluation of Erasure Coding and other features of Hadoop 3

August 2019

AUTHOR:
NAZERKE SEIDAN

SUPERVISORS:
Emil Kleszcz
Zbigniew Baranowski

PROJECT SPECIFICATION

Erasure coding, a new feature in HDFS, can reduce storage overhead by approximately 50% compared to replication while maintaining the same durability guarantees. This would allow to save a lot of disk capacity in needed by project hosted in CERN IT Hadoop service. The goal of the project is to evaluate the new features of Hadoop 3 and make an assessment of its readiness for production systems (this includes installation and configuration of a test hadoop3 cluster, copying production data to it, conducting multiple performance test on the data).

References:

https://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop/

ABSTRACT

Hadoop ecosystem is distributed computing platform for Big Data solutions by comprising autonomous components such as HDFS, Spark, YARN etc. HDFS is a Hadoop Distributed File System for data storage. Current HDFS supports 3x replication for data fault-tolerance. When a client writes a file to HDFS, the file will be replicated twice and distributed over the DataNodes of the cluster. It has 200% storage overhead. However, there is a big improvement in reducing the storage cost which is called Erasure Coding (EC). Erasure Coding (EC) decreases the storage overhead by approximately 50%. The overall target of this project is to do some performance testing for HDFS High Availability (HA) features: Erasure Coding (EC), Triple NameNode High Availability and HDFS Router-based Federation. We mainly focus on measurement of raw storage and analytics performances of Erasure Coding in comparison with 3x replication. We evaluate the performance of write/read operations to/from HDFS on small datasets. We test TPC-DS benchmark on Spark SQL queries of 10GB, 100GB and 1TB datasets in Parquet and JSON file formats.

TABLE OF CONTENTS

1. INTRODUCTION

Apache Hadoop is an open-source computing framework for storing sheer amount of data and processing them in a distributed way. It is made up of three domains: data computation, data storage and resource management. Data computation can be any data processing engine like Spark, Hive while HDFS is considered as the main data storage system. HDFS is a distributed file system which is totally different from non-distributed file systems in a way that data is replicated and stored in multiple machines for fault tolerance. In the next sections 3x replication and Erasure Coding (EC) policies will be described in details.

2. 3X REPLICATION

Current HDFS supports 3x replication policy for data availability and redundancy by default. Suppose a client wants to write a file to HDFS. At first, the client connects to NameNode (NN) to ask for available DataNode (DN) location which is nearby. Then the client is linked to the nearest DN to write the file. Actually, the file is written to HDFS as a set blocks in hard drives. The blocks will be replicated twice and stored in different DNs of the cluster. Here the replication factor is 3 but it is configurable which means that the system can tolerate up to 2 DataNode failures. The visualization for this policy is as follows (see Figure 1):

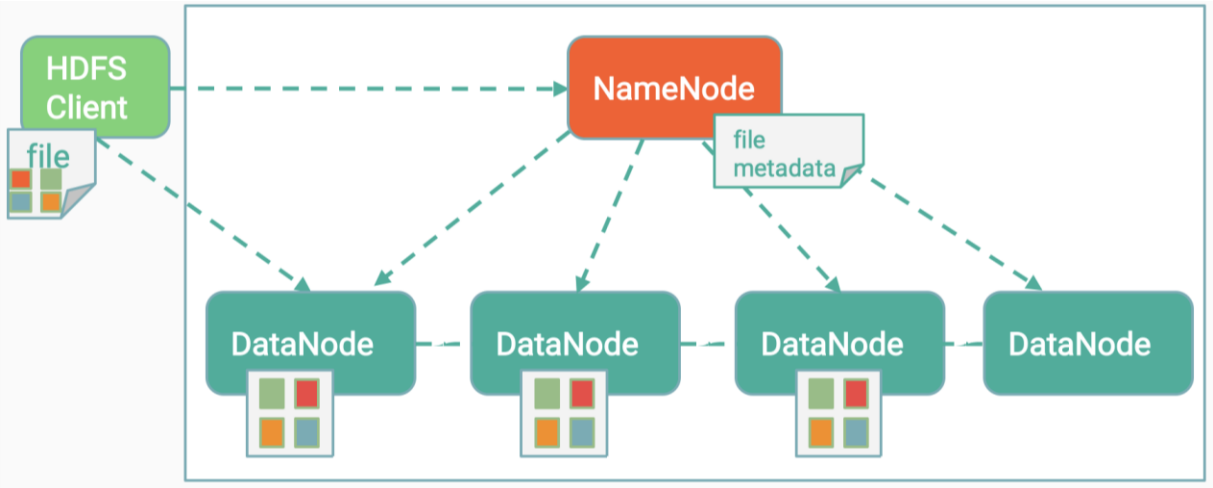

Figure 1 HDFS 3x replication

If you are wondering why we need to duplicate the file, consider the case when one of the DNs in the cluster fails for some reason which does not function anymore. Then the file stored in the failed DN is lost. Therefore, HDFS provides the data redundancy feature by replicating the file. The additional storage cost for this 3x replication accounts for 200%. Actually, there is a big improvement in reducing the storage overhead by introducing a new policy, Erasure Coding (EC).

3. ERASURE CODING

Erasure Coding (EC) is an error-correction method, it uses RAID 5/6 concept to protect data. It can give the same level of fault tolerance as 3x replication but with much less storage overhead. Assume we would like to write a file to a HDFS directory where EC is applied. The given file is striped into 1MB so then it can be distributed over N data blocks. In order to provide data redundancy and availability, EC computes K parity chunks using Reed-Solomon(N,K) algorithm. Reed-Solomon(N,K) codes are a collection of error-correcting codes. It takes two parameters that are number of data and parity blocks. For illustration N=6, K=3 and the stripe cell size = 1024k, block size = 256MB(see Figure 2).

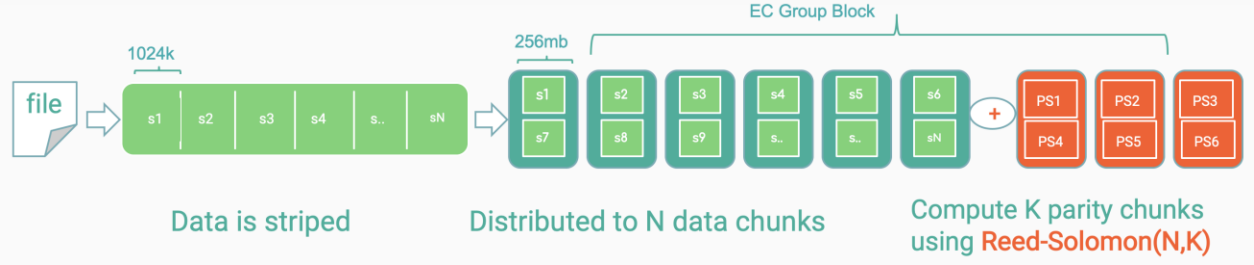

Figure 2 Write a file to an erasure-coded directory in HDFS

EC can recover blocks up to K failed DNs through decoding the parity chunks. For example, when a user wants to read a file which has missing blocks. Then EC computes the missing blocks using parity information (Figure 3):

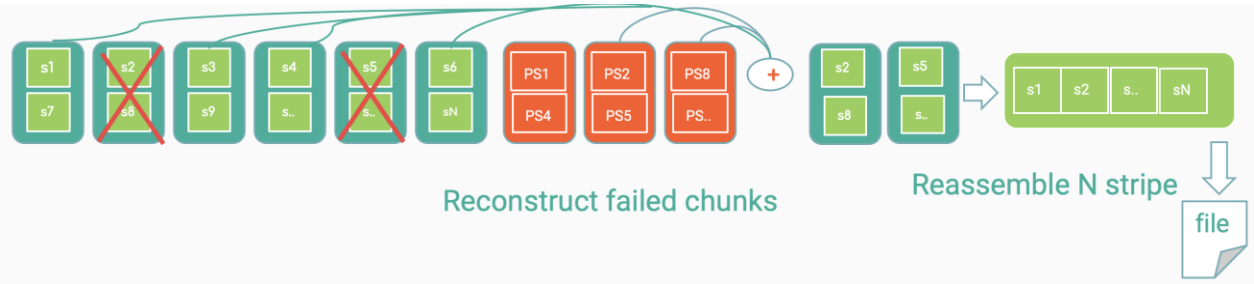

Figure 3 Block Recovery with Erasure Coding

Currently there are three supported EC policies based on the parameters: RS(3,2), RS(6,3) and RS(10,4). The following table shows the difference between the EC policies:

Replication Policy	Additional Storage Cost	Fault Tolerance
RS(3,2)	66%	2
RS(6,3)	50%	3
RS(10,4)	40%	4
3x replication	200%	2

Figure 4 Different types of EC policies

From the table (Figure 4) it can be seen that with Erasure Coding we can save about 40-60% disk space without compromising the fault tolerance. It is worth to mention that it is recommended to have at least N+K DNs in the cluster to achieve corresponding data durability as shown in the table.

Let us take a look how Erasure Coding can be put into practice. EC policy is applied at a directory level. We create a directory, then we apply EC RS(6,3) policy on it. These steps are shown using a couple of commands:

$ **hdfs dfs -mkdir hdfs://hadoop3/user/nseidan/ec**

$ **hdfs ec -setPolicy -path hdfs://hadoop3/user/nseidan/ec -policy RS-6-3-1024k**

Once we have an erasure-coded directory we can evaluate the EC policy on it. In the next section evaluation methods will be described.

4. EVALUATION METHODS

We have focused on two test performances: raw storage performance (write/read operations) and analytics performance. In terms of raw storage, we evaluated the performance of write/read operations to/from HDFS on different datasets such as 100MB, 1GB, 10GB, 100GB and etc. For analytics performance, we used TPC-DS benchmarking tool[1] to measure the performance of Spark SQL queries on 100GB, 1TB and 10TB datasets in Parquet and JSON file formats. TPC-DS is a decision support benchmark to measure the performance of data processing engines such as Spark 2.4.1. For this measurement, we configured a cluster which has 16 machines. The configuration for this cluster details as follows:

[1]github.com/tr0k/Miscellaneous/blob/master/Spark_Notes/Spark_Misc_Info.md

- 16 machines: 14 DataNodes, 2 NameNodes
- 512GB RAM/server
- 16 physical CPU cores
- 48 drives/DataNode
- 5.5TB/drive

The configuration for TPC-DS benchmark test:

- 84 executors
- 48GB executor memory
- 4 executor cores

5. EVALUATION RESULTS

In this section evaluation results mostly comparisons between Erasure Coding and 3x replication will be presented. First of all, let us take a look at the next bar chart (Figure 5) to have some insight how EC reduces the storage space.

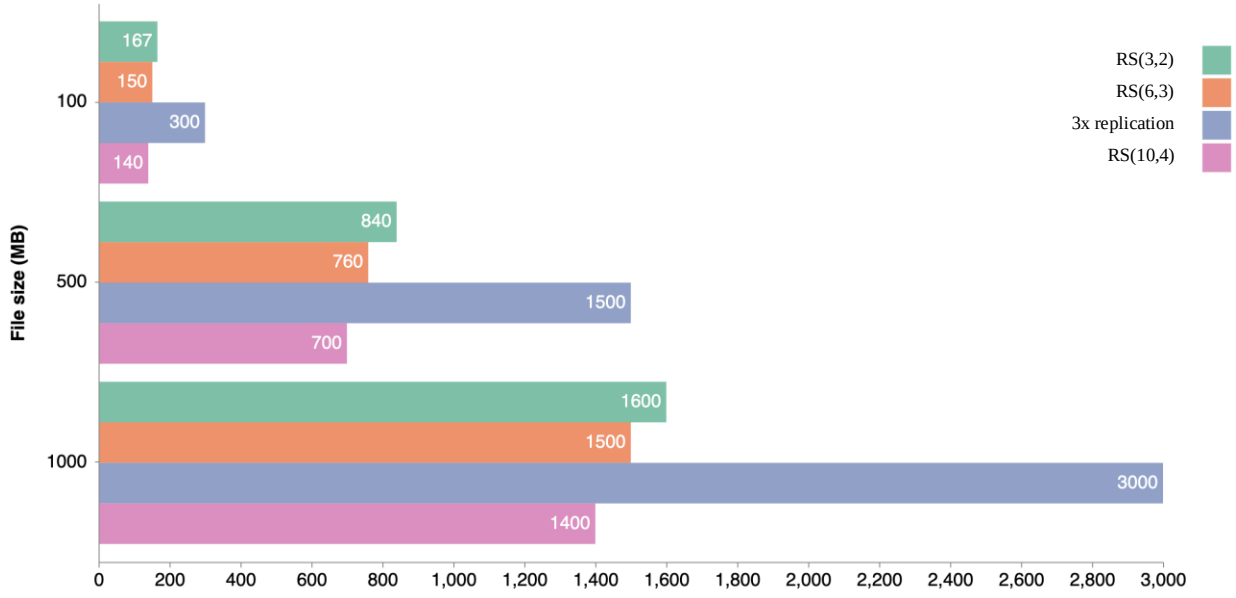

Figure 5 Storage Cost of EC policies & 3x replication

It can be seen from Figure 5 on the y-axis different file sizes are given, while on the x-axis actual storage costs are shown. For instance, if we write 1000MB data in HDFS, then it takes 3000MB disk space in DataNode, which is expensive from the point of storage cost. Nevertheless, applying any of EC policy, indeed we can save immense amount of disk space.

Although Hadoop is not for storing small data files, there might be a use case that client writes small-sized files to HDFS. Unfortunately, if the file size is smaller than the stripe cell size, erasure-coded data occupies more space than 3x replication. The graph for this use case is as follows:

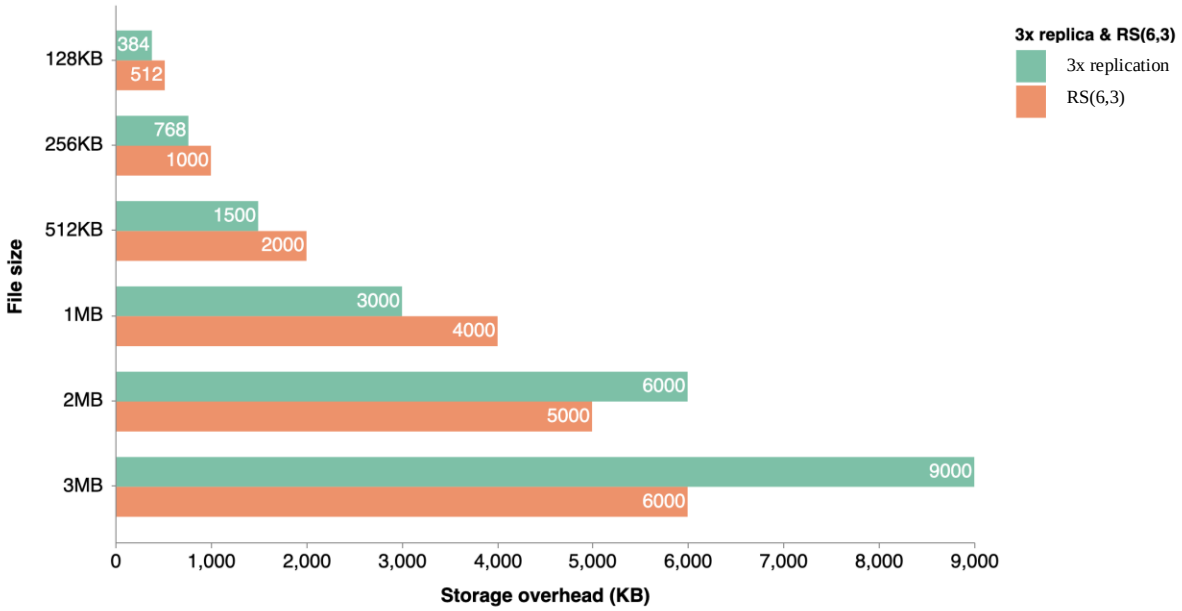

Figure 6 Storage Cost on small datasets

Another measurement that we measured the performance of writing files from local file system to HDFS. The main point from the following bar chart is that write operation is slower with EC than 3x replication due to parity blocks computation using Reed-Solomon algorithm which is very CPU intensive (see Figure 7).

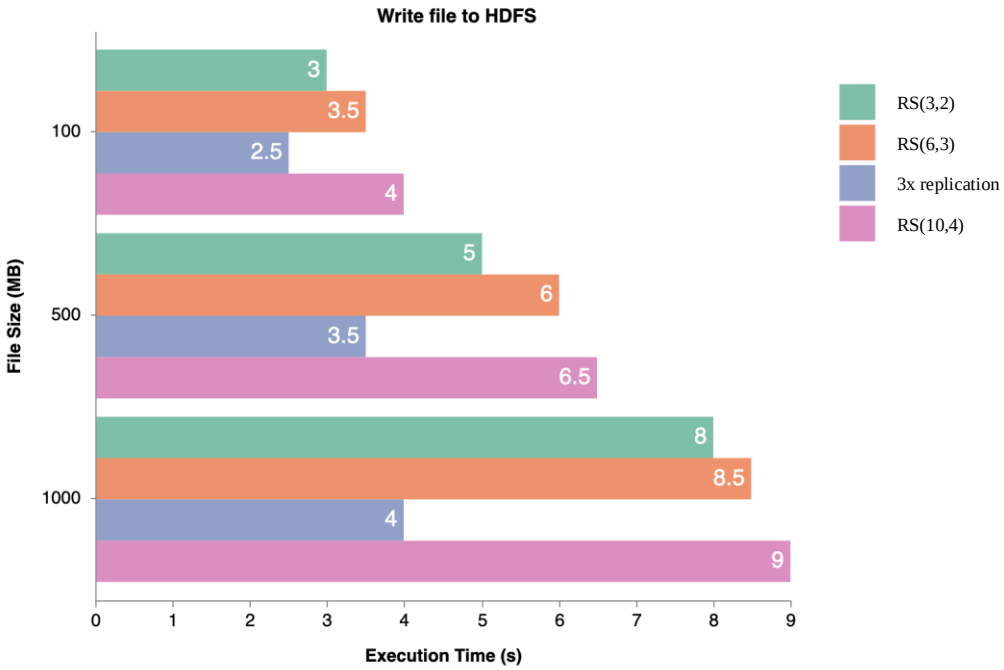

Figure 7 Write to HDFS

We tried to improve the performance of write operation on Erasure Coding using Intel's ISA-L[2] library. ISA-L is a set of optimized low-level methods aiming storage applications. Thanks to ISA-L library we got around 30% performance improvement (see Figure 8).

[2] github.com/intel/isa-l

Figure 8 Write 100GB data file to HDFS

Along with the evaluation of write operation, we have measured the performance of read operation from HDFS to local file system. Apparently reading from a directory where EC is enabled is almost twice faster than 3x replication because of parallelism. EC reads from N hard drives at a time where N is the parameter of RS(N,K) algorithm (see Figure 9).

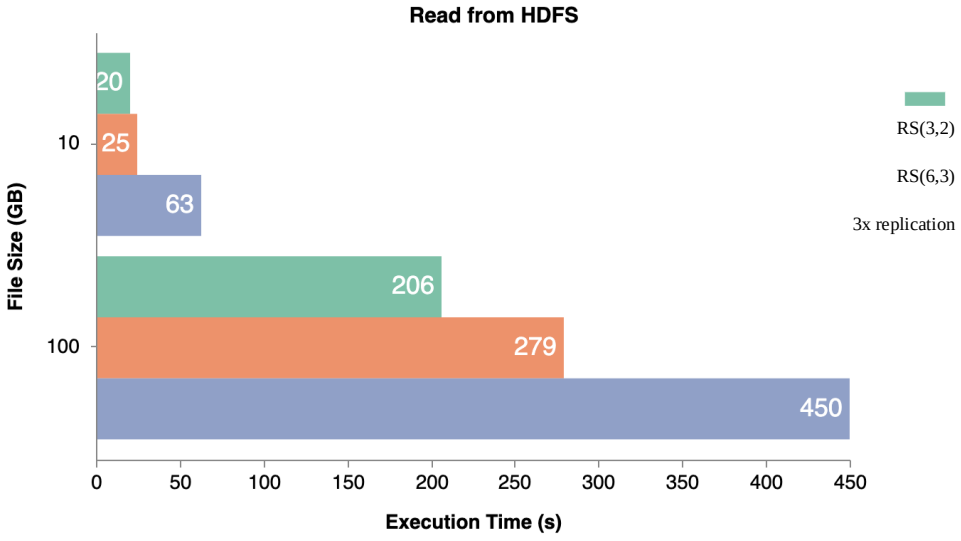

Figure 8 Read from HDFS

After completing the raw storage performance tests, we have evaluated TPC-DS benchmark test as analytics performance plays a crucial role for data processing engines like Spark, Hive and etc. Initially, we generated 100GB, 1TB and 10TB datasets in Parquet, JSON formats. Afterwards, we run the benchmark on the generated datasets using Spark shell. Once we finished running benchmarks, we extracted the results by taking the average of all queries' execution time. There were in total 99 SQL-based queries.

Let us see the execution time for each query based on 100GB dataset in Parquet and JSON formats (Figure 10 and 11 respectively):

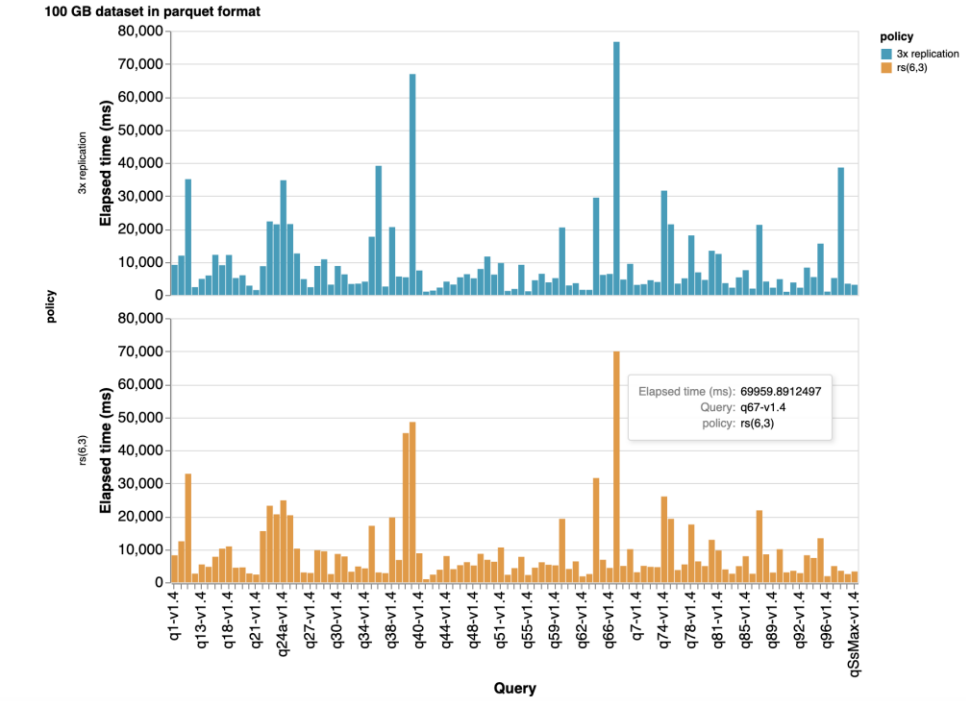

Figure 9 TPC-DS benchmark on 100GB dataset in Parquet

We run all queries with 3 iterations in order to obtain quite accurate result. For both policies EC and 3x replication, overall average execution time is approximately 16 minutes on the same dataset. Whereas the average execution time in JSON format with 3x replication accounted for about 40 minutes. Surprisingly, JSON format with EC RS(6,3) policy took 80 minutes on average to execute all queries.

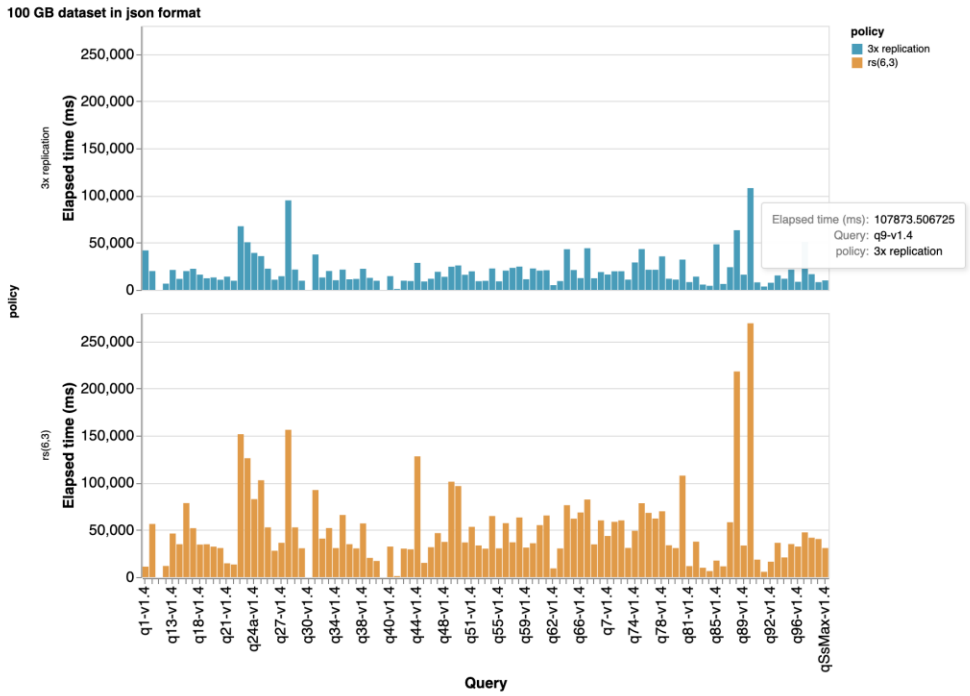

Figure 10 TPC-DS benchmark on 100GB dataset in JSON

Furthermore, we increased the data size by 1TB to see how EC performs in different file formats, especially with JSON format. If you notice on the following graph (Figure 12), you can see that EC with JSON was taking much amount of time compared to 3x replication. We have observed very high network utilisation between cluster nodes in this case of EC with JSON. This could be explained by lack of a data locality in processing big erasure coded JSON files. Therefore, from this experiment, we can conclude that data file formats without optimized organization for analytic-like access (e.g. text-based formats – JSON, CSV) should be avoided for analytic processing with EC policies at all. Otherwise remote full scanning of terabytes of data may lead to saturation of entire cluster network.

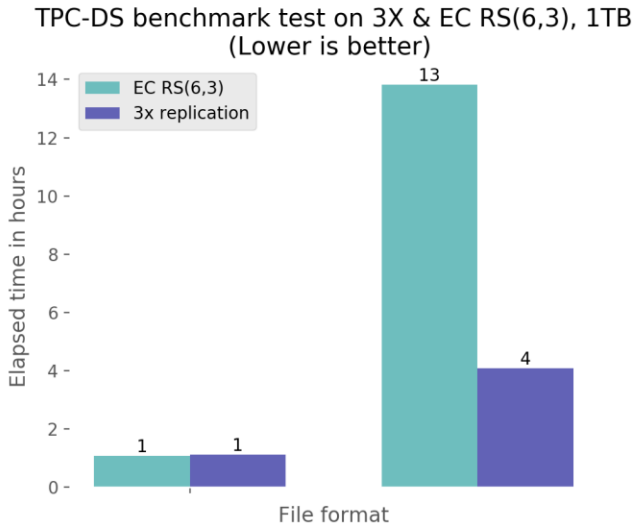

Figure 11 TPC-DS benchmark on 1TB dataset

After evaluation of Erasure Coding feature of Hadoop 3, we continued to measure the performance of other feature of Hadoop 3. Namely, we focused on two features: Triple NameNode High Availability and HDFS Router-based Federation.

6. TRIPLE NAMENODE HIGH AVAILABILITY

Hadoop 3 overcomes a single point of failure by providing support for several NameNodes (NNs) in HDFS cluster. Current NameNode High Availability architecture (see Figure 13) has an option to run two NNs in the same cluster as one is in the active state, while the other one is in the standby state.

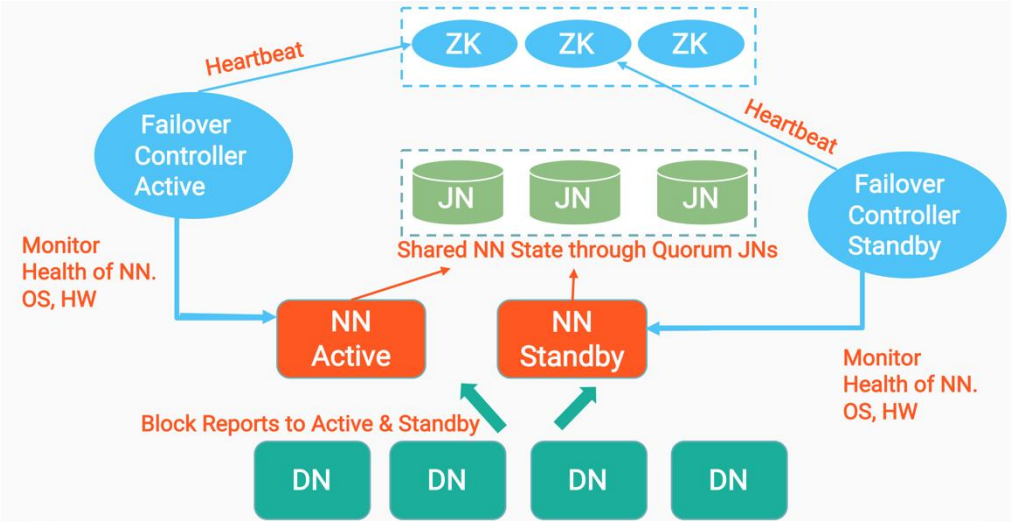

Figure 12 NameNode High Availability

Active NameNode handles all operations in the cluster, while Standby NameNode acts as a slave for providing a fast failover. When clients and DataNodes modify the namespace, the Active NN modifies the edit logs in the Quorum Journal Nodes to reflect the changes on the namespace, whereas Standby NN pulls the latest modifications from Quorum Journal Nodes to synchronize with the Active NN. Note that there must be at least three Quorum Journal Nodes to continue functioning after failure. Failover Controllers monitor health of NNs and report their heartbeats to ZooKeeper. If ZooKeeper does not hear any response from Failover Controller about NN's heartbeat, he elects Standby NN as Active. But what if both NNs Active and Standby NNs fail, then the whole system is down. Therefore, we introduce Triple NameNode High Availability.

Instead of having only one Standby in Hadoop 3, we can have several of them for failover. In our case, we added one more NN to the current HDFS NN architecture (see Figure 14). It is worth to mention there must be only one NN active at a time, otherwise it will lead to Split-Brain Scenario which would divide the cluster into smaller sub-clusters. The protection against such state is implemented in a Quorum Journal Node – it allows only one NN to write a log in a given incarnation.

Pertaining to our evaluation, we manually shut down two NNs to see how the system handles the case. Fortunately, the system was able to do a fast failover by setting the third Standby NN as an active NN. Besides that, we tried to flip the states (active and standby) between NNs for the performance purpose. The command that we used for this flipping scenario:

doAs hdfs hdfs haadmin -ns <cluster namespace> -failover <current active> <target active>

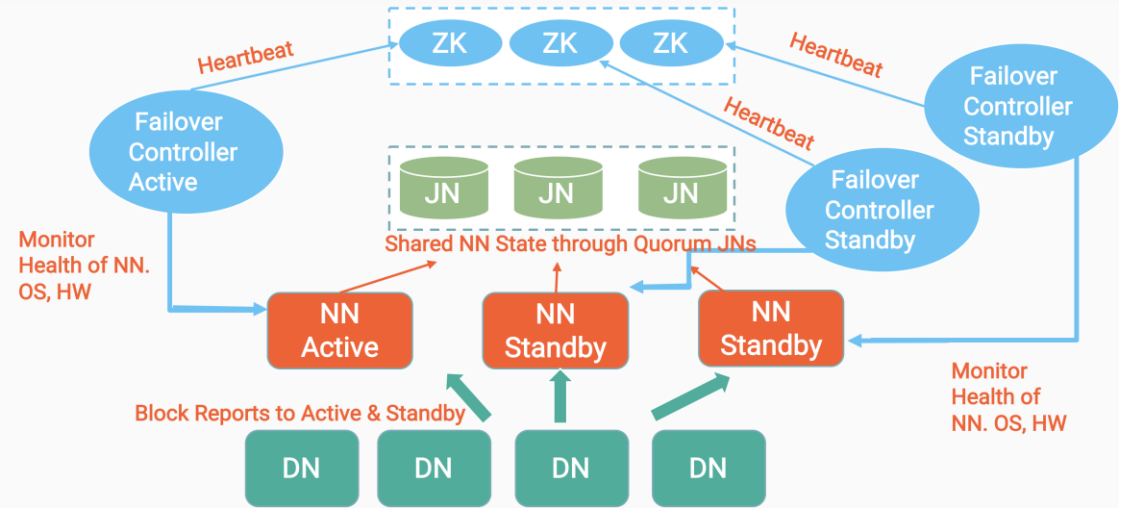

Figure 13 Triple NameNode High Availability Architecture

7. HDFS ROUTER-BASED FEDERATION

NameNodes have scalability issues because of the metadata overhead from file blocks, DN heartbeats and etc. To overcome this problem, Router-based federation has been recently introduced in Hadoop 3. The main idea of this feature is that users can access any sub-cluster of the federation through a router. The router is added in the top layer which handles client requests. For illustration, let's have a look at the following diagram to understand this feature better (Figure 15):

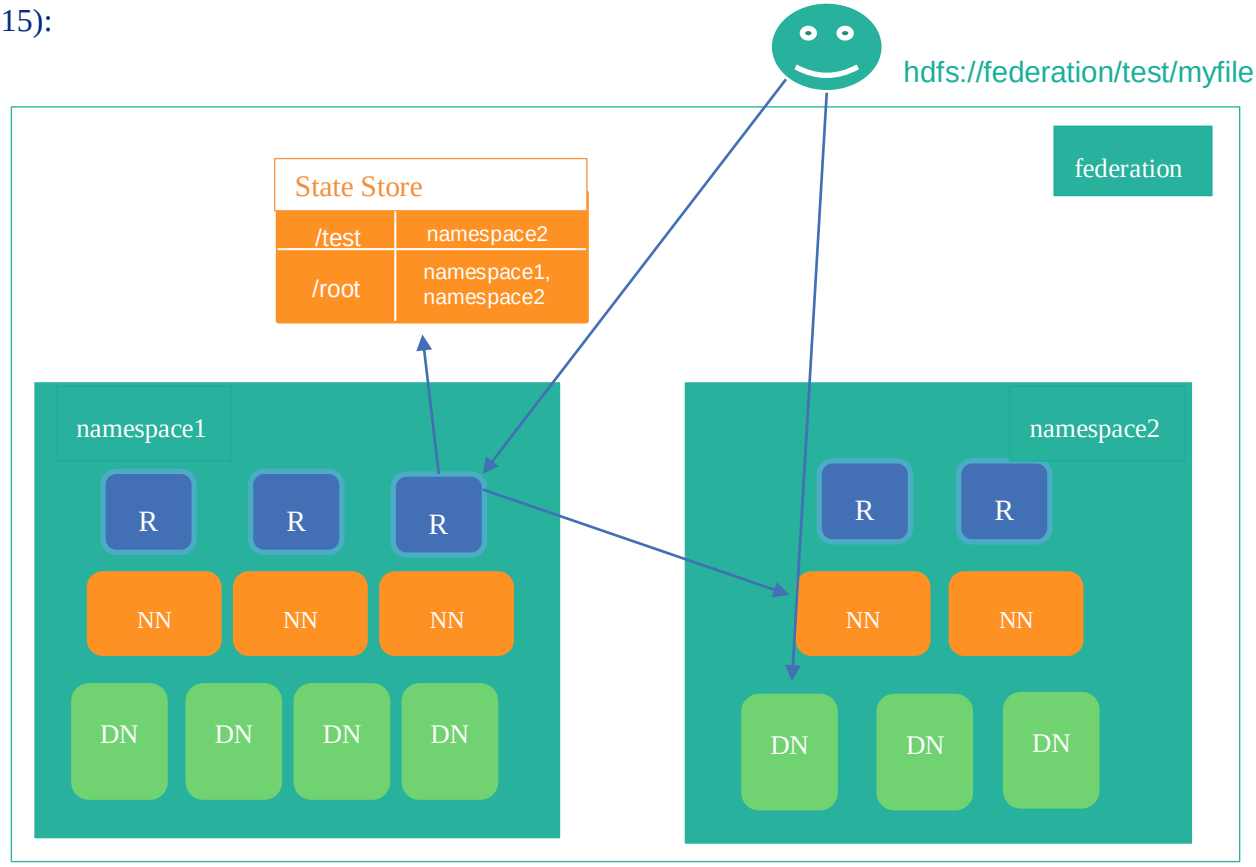

Figure 14 HDFS Router-based Federation

When users ask for a file from the federation, the router mimics (proxy) the behaviour of a NameNode. It first looks up State Store to see which namespace is responsible for storing the requested path. Then the router on behalf of the user queries the relevant NameNode and return back the results to the user with the information about location of the file blocks. In the same way router handles write path, however in this case depending on the configuration it can decide which namespace to use to store directories and file. To register a new path in the State Store:

hdfs dfsrouteradmin -add /test namespace2 /test

The above command registers a path, called test, to the State Store. The State Store is a simply mapping table from a path to an appropriate namespace. To set a quota of the path:

hdfs dfsrouteradmin -setQuota /test -ssQuota 10000m

It sets the quota of 10GB so that users can write to this path up to 10GB dataset. Let us assume namespace1 and namespace2 are clustered under the federation logical namespace. When a user writes a data file to the test path, then namespace1 cluster must have the path called test. So that the user is able to write the data file:

hdfs dfs -put myfile hdfs://federation/test

The myfile will be written to namespace1 as the State Store maps to that namespace1. Furthermore, it is possible to register a path under both namespaces:

hdfs dfsrouteradmin -add /test namespace1, namespace2 /test

In this case to decide which namespace to write, it uses order parameter, there are the following methods: HASH, LOCAL, RANDOM, HASH_ALL, SPACE. If the order parameter is set to SPACE, then it looks at the available space in sub-clusters of the federation. Whereas HASH puts all the files/folders under one sub-cluster, while HASH_ALL spreads the given files throughout the federation. Once we register a path in the State Store, we can modify the settings by the following command:

hdfs dfsrouteradmin -update /test -order SPACE

In addition to the order parameter we can set the path as readonly so that users cannot write any files/folders to that path:

hdfs dfsrouteradmin -update /test -order SPACE -readonly true

This HDFS Router-based architecture is used on our production-like system. The main challenge while configuring this feature was about secure authentication and authorization. Routers will not proxy to clusters where the security is on. This security feature is not released yet in Hadoop 3 (v3.2.0). Therefore, we had to get the implemented security patch from the upstream and merge it with our Hadoop version. All in all, it is properly functioning.

8. CONCLUSION

In this report we have evaluated the performance of Erasure Coding, Triple NameNode High Availability and HDFS Router-based Federation features of Hadoop 3.

From our evaluation tests, we concluded that Erasure Coding policy can be applied to directories where cold data is stored and writing speed is not on a critical path, since writing data to erasure-coded directories are not efficient as it needs some time for parity computing. Also, massive reading from text files (like JSON) is not effective with EC policy. Nevertheless, EC policy provides cheaper option to store data by reducing the storage space than 3x replication.

We have covered Triple NameNode for High Availability purpose. In case of two NN failures, HDFS should be available and serving the service to users.

Last but not least feature that we evaluated was Router-based federation. The main point of this feature was to overcome NameNode scalability limits by providing extra layer to transparently proxy user requests to multiple namespaces.

9. REFERENCES

[1] Visualization tool: https://vega.github.io/vega/

[2] Erasure Coding Concept: https://www.usenix.org/system/files/login/articles/10_plank-online.pdf

[3] HDFS Erasure Coding in Production: https://blog.cloudera.com/hdfs-erasure-coding-in-production/

[4] HDFS High Availability: https://hadoop.apache.org/docs/r3.1.2/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html

[5] HDFS Router-based federation: https://hadoop.apache.org/docs/r3.1.2/hadoop-project-dist/hadoop-hdfs-rbf/HDFSRouterFederation.html