Simulation-Based Performance Analysis of HDFS-RAID and Locally Repairable Codes(LRC)

Ankit Sharma
DA-IICT, Gandhinagar
200901099@daiict.ac.in
Supervisor
Prof. Manish K. Gupta

***Abstract*—Hadoop Distributed File System(HDFS) is a highly fault tolerant system that runs on commodity hardware and data is reliably stored in replicated manner. But the problem with this architecture is the storage overhead it creates while replicating the data. To overcome this overhead, erasure coding techniques are used. In these techniques, instead of using replication, either data parities are created, keeping the original data as it is or encoded data is stored. HDFS-RAID uses Reed-Solomon codes to generate the parity of the original data. Further, to decrease the repair bandwidth of single block failure in HDFS-RAID, Locally Repairable Codes(LRC) are used. This paper briefs about HDFS-RAID and LRC and their performance comparison with basic HDFS architecture using the built simulator.**

***Index Terms*—Big Data, Hadoop, HDFS, HDFS-RAID, Locally Repairable Codes(LRC), Parity**

I. INTRODUCTION

A data becomes ”big data” when it surpasses the processing capability of a traditional database systems and cannot be analyzed with conventional computing techniques. Organizations working in logistics, financial, healthcare services, etc. are capturing more and more data. And the huge social media is producing vast quantity of digital data. In the current world scenario, there are billions of webpages, billion Facebook users [5] and many more Facebook pages, hundreds of millions of twitter accounts [6], millions of tweets per day [6], billions of Google queries per day [7], millions of servers [8] and terabytes of data powering all this. And around 80% of this data is unstructured [9]. The quantity of data generated daily in the world is growing exponentially. But, not every enterprise handles this huge amount to data. Many large enterprises just have few thousands of servers. And their architecture pretty much looks like this:

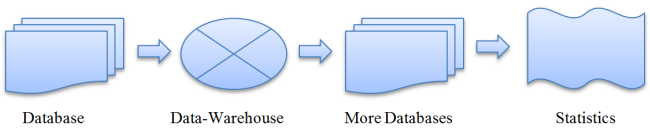

Fig. 1. Data handling architecture for large enterprises with less data processing workload

Here we have some databases where data gets collected. Then that data gets transferred into Data Warehouses. And then taken out into further databases on which some statistics or reporting is performed. But for the large enterprises which handles massive amount of data, this approach simply did not work. Companies like Facebook, Google, Yahoo!, Amazon where processing of large volumes of big data is done, did not use traditional databases. In fact they could not.
In Dec 2004, Google released white-papers describing Google infrastructure. There were three white papers: GFS(Google File System), Google MapReduce and Google BigTable. But Google never released their source code. Then Doug Cutting and Michael J. Cafarella researched every nook and cranny of GFS and redefined it into Hadoop Distributed File System (HDFS) [3]. Later some companies like Yahoo!, Apache, Cloudera came up with Hadoop (HDFS) which is totally open source. Thereafter, GFS became Apache HDFS, Google MapReduce became Apache MapReduce and Google BigTable became Apache HBase. Around 95% of the architecture described in the Google white-papers are implemented in these java based Apache projects.

Characteristics of Hadoop:

1) It is optimized to handle massive quantities of data which could be structured, unstructured or semi-structured.
2) Its massive parallel processing is done with great performance.
3) It replicates its data across different computers so that if one goes down the date can be obtained from one of the replicated computers.
4) High fault tolerance, high scalable, fast regeneration of data and robust coherency.

This paper is further organized as follows: We initially present the theory behind HDFS-RAID and LRC in section II and III respectively. Section IV presents the description of the simulator with its technical specifications. Section V discusses the experimental observations of the simulation. We finally conclude in Section VI.

II. HDFS-RAID

Due to these interesting characteristics, the HDFS architecture is becoming more and more famous in the fields like Data Management, Data Processing and Data Analytics. Most of the HDFS systems use the replication factor of 3, i.e. the same data is stored at 3 different locations. The

main problem with this architecture is that it creates the storage overhead by 200%. Due to this reason, to overcome this overhead, many companies are shifting their "cold date"(data which is rarely accessed) and easily archival data, into encoded form using erasure coding techniques. Here we describe an open source erasure coding module called HDFS-RAID which is based on Reed-Solomon codes.

Irving Reed and Gustave Solomon invented Reed-Solomon Codes while working in MIT Lincoln Labs in 1960. Nowadays, RS codes are found in Wireless Communications, Storage devices, Digital Televisions, etc. RS codes are the systematic and linear blocks codes and come under the subclass of non-binary BCH codes. The basic idea of RS encoding is, some extra information is added to the original data which is called "Parity". The parity is generated in such a way that when some part of the original data gets corrupted, the decoder can correct it back using that parity. The decoder can correct only up to a certain number of errors in the code. And that error correcting limit depends on the size of parity taken.

Reed Solomon code is represented by (n,k)RS code[2], where:

n: total number of block symbols.
k: number of original message symbols
n-k: number of parity symbols (which is called 2t)

A (n,k)RS code can correct up to ((n-k) / 2) or t symbols.

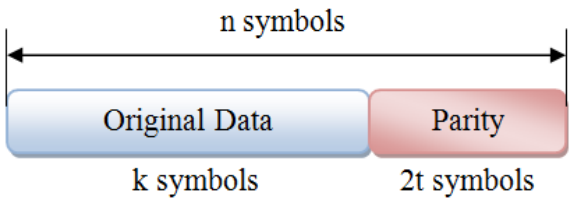

Fig. 2. Structure of a General Reed-Soloman Code

In our implementation we are using RS(26,16) code. Therefore, up to 5 errors can be corrected.

During RS Encoding, The original message of k message symbols is represented by polynomial s(x). Encoder divides s(x) by the generator polynomial g(x) to generate required parity r(x).

Decoder is much more complex task than encoder. The parity generation is based on the idea of creating a large polynomial of n coefficients which can be divided by generator polynomial. Therefore, the decoder divides the received message by g(x). If it creates zero as a remainder, the message is said to be error-free. Otherwise the data is corrupted. In our implementation we use Euclidean algorithm [4] for error correction at decoder end.

Now, instead of having storage overhead of 200% in the HDFS architecture, it has shifted down to 62.5% (parity overhead). This is a huge difference when it comes to today's Big Data. Therefore, RS codes are significantly storage efficient compare to HDFS replication.

III. LOCALLY REPAIRABLE CODES(LRC)

During RS Decoding, transferring of all 16 blocks is required to regenerate failed blocks, even if only one block is failed (or corrupted). This consumes a huge decode bandwidth and temporary disk I/O.

Therefore, to overcome this major problem, a new family of erasure codes called Locally Repairable Codes(LRCs) [1] are introduced. These codes are efficient both in terms decode bandwidth and disk I/O.

Implementation of LRC:

LRC is implemented over RAID. That is, encoding and decoding is almost the same as done in RS codes. Things to keep in mind for LRC are:

1) During Encode: Apart from RS parity, add two local parities for each set of 16 original data symbols. One parity for first 8 blocks and the other of the remaining 8. And add one local parity for 10 RS parities.
2) During Decoding: When more than one block is failed/corrupted, use the original RS decoding. But when only one block is failed, use the local parity to repair the failed block.

Now, here we describe the (26,16,8) LRC code that we have implemented in our simulation. This LRC code is implemented over (26,16) RS code. Therefore we have the stripe with blocks B1, B2, B3,..,B16. And the generated parities P1, P2,.., P10. Now, the local parities are generated as shown in Fig. 3.

Local parities S1, S2 and S3 are added. Here, S1 is generated by XOR operation between first eight blocks:

$$S1 = B1 \oplus B2 \oplus B3 \oplus B4 \oplus B5 \oplus B6 \oplus B7 \oplus B8 \quad (1)$$

And similarly S2 is generated by next eight blocks. S3 is generated by using XOR operation between RS parities.

$$S3 = P1 \oplus P2 \oplus P3 \oplus P4 \oplus P5 \oplus P6 \oplus P7 \oplus P8 \quad (2)$$

Now, if a single block failure occurs, it can be repaired by generated local parity. For example, if B3 fails, it can be regenerated using XOR operation between remaining 7 blocks and generated local parity, as follows:

$$B3 = S1 \oplus B1 \oplus B2 \oplus B4 \oplus B5 \oplus B6 \oplus B7 \oplus B8 \quad (3)$$

If local parity is failed it is again computed in the same way it was computed before.

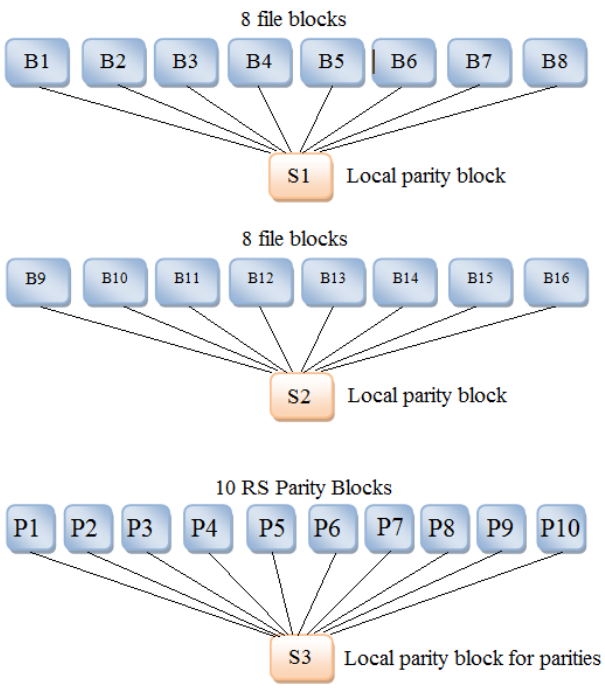

Fig. 3. Locally repairable codes(LRC) implemented in the simulator. The Ten parity blocks P1, , P2,, P10 are constructed with (26,16)RS code and the local parities S1, S2 and S3 are constructed with XOR operations. Local parities provide efficient repair in the case of single block failures.

The main disadvantage with LRC is the extra storage. Instead of 26 blocks in RS encoding, we are now using 3 extra blocks. But the robustness it provides when it comes to repairing of single block failure is phenomenon. The observation of simulation done on RAID and LRC is provided in section V.

It can further be optimized by not storing S3 and selecting S1 and S2 in such a way that

$$S1 + S2 + S3 = 0 \tag{4}$$

In this case, just using XOR operations between the blocks is not enough. Each block has to introduce a field element coefficient c such that local parity is generated as:

$$S1 = c1B1 + c2B2 + c3B3 + c4B4 + c5B5 \tag{5}$$

For now, in our simulation, we are using only XOR approach.

IV. IMPLEMENTATION

We have implemented a java based simulator for simulating HDFS, HDFS-RAID and LRC encoding and decoding algorithms to compare the performance efficiency between them. We have constructed some graphs to represent the detailed analysis of our work.

Salient features of the simulator:

- *File Upload:* This feature allows the user to upload a file so that it can be encoded with different encoding techniques.
- *Project Directory:* This feature allows the user to choose a directory to store the project works. Files related to HDFS, HDFS-RAID and LRC are stored under the subdirectories "\HDFS\", "\RAID\" and "\RAID\Nodes\LRC\" respectively.
- *Encoder:* The Encoder provides different encoding options to the user. Once the user has selected the encoding format(s), the Encoder will make different directories under Project Directory and will store the related encoded data in those respective directories.
- *Node Failure Options:* This feature provides the options to the user, that whether only one block is to be failed or more than that. If one block failure option is selected, then only LRC can be used to reconstruct the failed block. And if more than one block failure option is selected then LRC failure recovery cannot be achieved.
- *Decoder:* Once the number of block failures are selected, the user can decode(or recover) the failed blocks by decoder. Decoder will provide different decoding options to the user. But only those options in which the encoding of the file was done can be selected for decoding.
- *File Regeneration:* This feature will provide the user to regenerate the original file from the encoded files. The user can choose HDFS and HDFS-RAID as the file regenerating options.
- *Generate Report:* This feature will generate the analysis of decoding and file regeneration in different options selected.

A. Implemenation of HDFS-RAID

When HDFS-RAID encoding mode is selected, a new subdirectory under the main project directory will be created with the name "\RAID\". Then according to the file size, different nodes will be created. One node can contain maximum 16 blocks. And 1 block has maximum size of 1MB. Therefore, one Node can contain maximum 16MB of original data. For example, if the file has the size of 50 MB, then four Nodes will be created. In the first 3 Nodes, 16 blocks of 1MB each will be filled and in the last Node, remaining 2 blocks will filled.

After dividing the files into chunks and storing it into different Nodes, parity files will be generated. For each Node with 16 blocks, 10 parity files of 1MB each will be generated. Therefore, one node contains total parity of maximum 10MB. This encoding is done using RS encoding. And the parity files for each Node is stored under the sub-directory named "\RS_Parity\".

Our approach can tolerate up to maximum 5 block failures. If a block is failed, the parity files will be called and the blocks will be repaired using RS decoding.

B. Implementation of LRC

LRC is implemented over the HDFS-RAID. Therefore, to implement LRC, first the HDFS-RAID must be implemented. Then, under each Node, a subdirectory will be created which

will contain LRC parity with the name "\LRC_Parity\". Now for every 8 blocks, one local parity will be stored under the subfolder. Therefore, one Node can have maximum 2 local parities. The parity will be generated with the XOR operation between the corresponding 8 blocks. Local parity for the 10 RS parity files will also be stored under "\LRC_Parity\".

Now, if one block is failed, it can be recovered by using the local LRC parity and other remaining 7 blocks.

V. OBSERVATIONS

A. Time taken by HDFS Encoder and HDFS-RAID Encoder

In HDFS architecture, a file is first divided into chunks of equal lengths and then each chunk is replicated according to the default replication factor and stored at random locations. In HDFS-RAID, after dividing the file into chunks, party files are generated using Reed-Solomon Encoding. Time taken in replicating and storing a 20MB file in HDFS is approximately 2 seconds. Whereas, time taken by RS encoder in HDFS-RAID to generate parities of a 2MB file is approximately 10 seconds. This is because RS parities are generated bit-wise. Therefore, encoding time in HDFS-RAID is very high as compared to that in normal HDFS. The encoding time is highly dependent on the quality of the processor used.

B. Decoding time in normal HDFS and HDFS-RAID

The maximum number of node failures that HDFS can reliably handle is equal to the replication factor. And the number of maximum nodes that HDFS-RAID can repair depends upon the construction of RS codes. The comparison analysis of node repair time between HDFS and HDFS-RAID is shown in fig 4.

In HDFS, when a node is failed, the NameNode will check its log files to obtain the locations at which the failed blocks were replicated. Then it will copy those replicated blocks from the obtained locations and finally store them into a new node. In HDFS-RAID, when a node is failed, NameNode will check its log files to obtain its remaining data blocks and parity blocks. Then, it will repair the failed nodes using RS decoding. Here, HDFS-RAID decoding takes much larger time than normal HDFS decoding because of the bit-wise decoding performed in RS decoding.

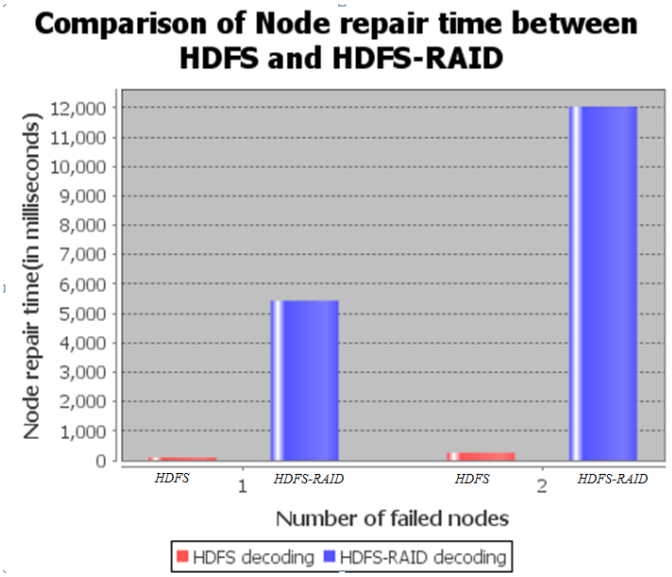

Fig. 4. Comparison of node repair time between normal HDFS and HDFS-RAID

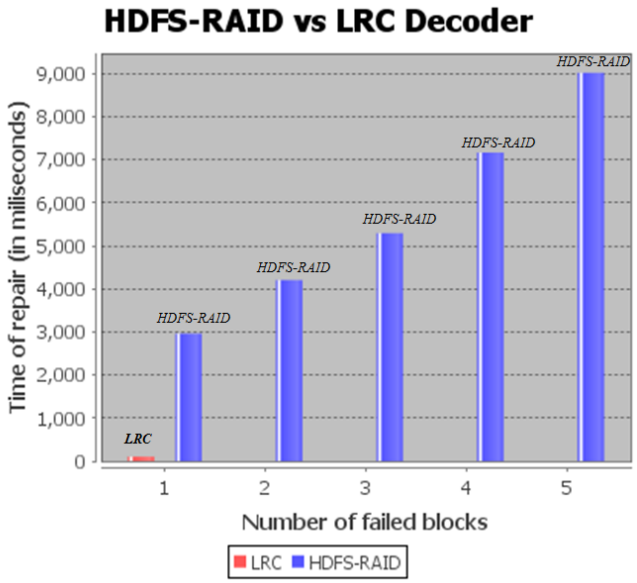

Fig. 5. Comparison of block repair time between HDFS-RAID and LRC

C. Decoding time in HDFS-RAID and LRC

Comparison between HDFS-RAID decoding and LRC decoding can only be done over single block failure. The observation obtained by repairing single block failure in LRC vs. single and multiple block failure(s) in HDFS-RAID is shown in Fig 5.

Fig 5 shows that time taken by LRC decoder to repair a single block failure is approximately 200 milliseconds. Whereas, time taken by RS decoder is approximately 1.8 seconds. Therefore, time taken to repair single block failure in HDFS-RAID is 800% more than that in LRC. RS decoder can repair more than one block and the block repair time increases linearly as the number of failed blocks increases.

D. Storage in HDFS, HDFS-RAID and LRC

In HDFS, memory storage depends on the replication factor. And in HDFS-RAID, it depends on the type of RS code chosen. In our simulation we are taking HDFS replication factor as 3, the code type as (26,16) RS code and the block size of each data-chunk as 1MB. Now, by taking a file of 100MB, the following observation is obtained:

TABLE I
COMPARISON OF STORAGE REQUIREMENTS

Architecture	Storage Requirement(in MB)
HDFS	300
HDFS-RAID	162.25
LRC	182.25

VI. Conclusions

This paper mainly focuses on the comparison of performance analysis between HDFS, HDFS-RAID and LRC. We have thoroughly understood the algorithms used in these architectures and implemented them into a simulator. We understood that normal HDFS creates storage overhead because of its replication property but the time taken to repair any failed node is very less as compared to that in HDFS-RAID. On the other hand, HDFS-RAID only takes approximately half of the storage space as compared to normal HDFS. Further, it is been observed that LRC technique is more optimal in case of single block failure. Normal HDFS is preferred because of its simple and easy to operate architecture. But to overcome the hassle of Big Date we cant just go on making it more ”Big”. Instead, Erasure coding techniques must use for Big Data. More research should be done in this field to make a better and more robust distributed systems.

The source code of the simulator is available at http://www.guptalab.org/mankg/public_html/WWW/students.html

Acknowledgment

We would like to express our sincere gratitude to Prof. Manish Kumar Gupta for helping us out with his continuous guidance, patience and invaluable support throughout the course of our project.

References

[1] M. Sathiamoorthy, M. Asteris, D. S. Papailiopoulos, A.G. Dimakis, R.Vadali, S. Chen, D. Borthakur, ”XORing Elephants: Novel Erasure Codes for Big Data”, VLDB 2013.
[2] ”Reed Solomon Codes”, Elektrobit, Joel Sylvester, Jan. 2011, http://www.csupomona.edu/.about.jskang/files/rs1.pd.
[3] HDFS Architecture Guide by Apache: http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html
[4] ReedSolomon error correction: http://en.wikipedia.org/wiki/Reed-Solomon_error_correction
[5] Facebook Statistics: http://newsroom.fb.com/Key-Facts
[6] Twitter Statistics: http://www.statisticbrain.com/twitter-statistics/
[7] Google Statistics: http://www.statisticbrain.com/google-searches/
[8] Google wiki: http://en.wikipedia.org/wiki/Google
[9] Unstructured data: http://en.wikipedia.org/wiki/Unstructured_data