Reducing insertion time in LRC-based cloud storage systems

Zhenyuan Liu, Zhen Huang and Dongsheng Li
School of Computer Science
National University of Defense Technology
Changsha, China
liuzhenyuan@outlook.com

***Abstract*—In traditional erasure codes, all the redundant data are created and uploaded to the different storage nodes by a unique source node. However, such a source node may have limited communication and computation capabilities, which constrain the storage process throughput. In-network redundancy generation can improve data insertion throughput through distributing the data insertion load among the source and storage nodes. But it's hard to schedule the generation process. Many works have proposed some heuristic scheduling algorithms to improve data insertion throughput. We propose a new method combined global and local optimization to schedule the process. Experimental results show that our method reduce insertion time up to 18% compared with the best heuristic scheduling algorithm.**

Keywords—Insertion time; in-network redundancy generation; LRC

I. INTRODUCTION

Cloud storage systems have gained prominence in recent years [1]. In cloud storage systems, data redundancy is essential due to storage node failures, or user attrition [2]. This redundancy can be achieved using either replication, or (erasure) coding techniques. Compared to replication, erasure codes are more space efficient but its data insertion is restricted to node who owns the original data (called source node) [3,9]. Source node has limited communication and computation resources which leads to a lower insertion throughput. [4,5] introduce in-network redundancy generation which can encode data during the insertion process. How to schedule the in-network redundancy generation has been proved to be a complicated problem [5].

To schedule the in-network redundancy generation, L. Pamies-Juarez et al. [4,5] proposes several heuristic scheduling algorithms which aim at maximizing the utilization of the spare resources of the storage nodes, thus improving the backup throughput. But these heuristic scheduling algorithms all belong to local optimization that lack global control. We propose a method combined global and local optimization to improve the performance of in-network redundancy generation.

After in-network redundancy generation process is completed, nodes used in the process will form a transport graph. The transport graph is non-trivial. Different transport graphs lead to different results. We design an algorithm to get a transport graph before in-network redundancy generation process and use the graph as a guide on how to schedule the process. We design a system combined the transport graph with a kind of heuristic scheduling algorithm. Experiment shows that our system can reduce insertion time and improve insertion throughput compared with [4,5].

II. BACKGROUND

Locally Repairable Codes.

Erasure code is drawing more and more attention in cloud storage system since it can improve system reliability and data availability, more importantly, it reduces storage cost compared to replication. Reed-Solomon codes are the standard design choice but their high repair cost is often considered unacceptable. There has recently been intense interest to explore alternatives, such as the Regenerating Code (RGC) [7,8], Hierarchical Code(HC) [9] and Homomorphic Self-repairing Code(HSRC) [6], where the repair cost for lost data can be reduced. These kinds of codes have local repairable property that an erased encoded block can be reconstructed from a small amount of data.

1) Locally Repairable Codes: We use a kind of Locally Repairable Codes (LRC) introduced in [4] which has the property that two codeword symbol can be xored for a third one just like HSRC [6]. A $\langle n,k \rangle$ code takes an object $o=(o_1,o_2,..,o_k), o_i \in F_{2^q}$, and generates $c=(c_1,..,c_n)$ that contains the k original symbols. We can represent the locally repairable property in terms of repair groups $R=\{r_i,..,r_r\}$, and each codeword symbol $c_i \in r_j$ can be repaired/generated by summing the other symbols in r_j, $c_i=\sum \psi_k c_k, c_k \in r_j (k \neq i)$, for predetermined ψ_k values, $\psi_k \in F_{2^q}$.

In-Network Redundancy Generation.

Besides high repair cost, another main drawbacks of using classical erasure codes for storage is that redundant fragments can only be generated by applying coding operations on the original data. The generation of new redundancy is then restricted to nodes that possess the original object (or a copy), namely: the source node which then also bears the load of inserting the encoded fragments at other storage nodes. The amount of data the source node uploads is then considerably

© 2015. The authors - Published by Atlantis Press

larger, including the data object and its corresponding redundancy, resulting in lower data insertion throughput.

In replication based storage system, redundancy data can be transferred in a pipelined way which allows quick data insertion and replication. Unlike replication, where in-network redundancy generation is achieved trivially, traditional erasure codes are not easily amenable. L. Pamies-Juarez et al. [4,5] have paid much attention on achieving in-network redundancy generation through LRC and HSRC's locally repairable property. They use some heuristic scheduling algorithms to get efficient in-network redundancy generation, but these heuristic scheduling algorithms belong to local optimization which may leads to unfavorable results. We propose a new method combined global and local optimization to improve the performance of in-network redundancy generation.

III. Scheduling the In-network Redundancy Generation

In this section, we present a new method to schedule the in-network redundancy generation process.

Problem Statement

Take a $\langle n,k \rangle$ LRC for example, it takes an object $o=(o_1,o_2,o_3)$, and generates a codeword $c=(c_1,..,c_7)$ where $c_1=o_1$, $c_2=o_2$, $c_3=o_3$. It has 7 different repair groups $R=\{r_1,..,r_7\}$, where: $r_1=\{c_1,c_2,c_4\}$, $r_2=\{c_1,c_3,c_5\}$, $r_3=\{c_1,c_6,c_7\}$, $r_4=\{c_2,c_3,c_6\}$, $r_5=\{c_2,c_5,c_7\}$, $r_6=\{c_3,c_4,c_7\}$, $r_7=\{c_4,c_5,c_6\}$.

We use the triplet notation $(c_i,c_j)\Rightarrow c_k$ to represent the possibility to generate fragment c_k by xoring c_i and c_j, $c_k=c_i+c_j$. r_1 has three possible generation relationships: $(c_1,c_2)\Rightarrow c_4$, $(c_1,c_4)\Rightarrow c_2$ and $(c_2,c_4)\Rightarrow c_1$. We call a repair group is available when one of its generation relationships exists.

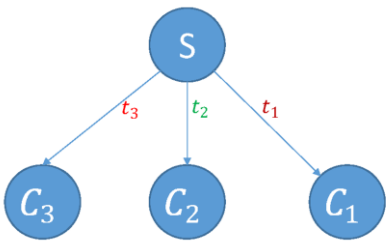

Fig. 1. Traffic from source node

The in-network redundancy generation process starts from source node that possesses the original object $o=(o_1,o_2,o_3)$. Source node is responsible for transferring o_1, o_2 and o_3 to three different nodes sequentially which generates $c_1=o_1,c_2=o_2,c_3=o_3$. We call a node u_i when it stores fragment c_i. And to simplify the discussion, we assume that completing transferring any one fragment would costs an equal time-step. From Fig. 1, we could see the source node needs 3 time-steps to complete its transfer. t_i in Fig. 1 means the transfer order.

After the source node's work, it's time to decide how to proceed the generation process. One possible solution is that we randomly choose an available repair group to generate the next fragment. We are able to complete generation at last but we may get an unfavourable results. Fig. 2 shows a random process. In Fig. 2, node u_1 and u_2 transfer fragment c_1 and c_2 respectively to node u_4. Fragment $c_4=c_1+c_2$ is then generated in node u_4. In the same way, c_5, c_6 and c_7 are generated in u_5, u_6 and u_7 respectively. It costs 7 time-steps altogether to complete this redundancy generation process. Fig. 3 shows another kind of process and it only needs 5 time-steps which means transport graph in Fig. 3 would leads to a better insertion throughput. If we see Fig. 2 and Fig. 3 as two directed acyclic graph, we could find that the max distance from source node to the last generated node is shorter in Fig. 3. Making the max distance as shorter as possible is crucial to construct a transport graph like Fig. 3. We show how we construct a transport graph in next subsection.

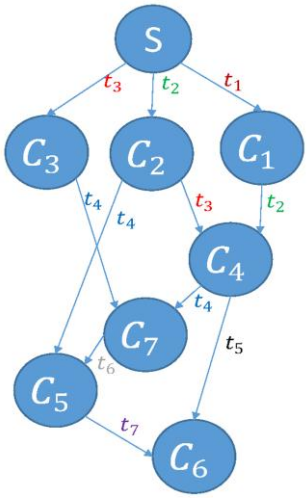

Fig. 2. A random process

Modeling the redundancy generation process.

We define distance between two adjacent nodes as 1. For a general $\langle n,k \rangle$ LRC, when source node complete its transfer, the max distance from source node to last generated node is 1. At this moment, the max number of available repair groups is C_k^2 in theory so we could generate up to C_k^2 fragments in distance 2. Then there will be at least $n-k-C_k^2$ fragments waiting to be generated. At every distance, more fragments are

generated, less fragments will be left. Less fragments means we could complete generation process earlier which leads to a shorter distance. If we monitor available repair groups in real time and generate fragments as much as possible in every distance, we could make the max distance shortest.

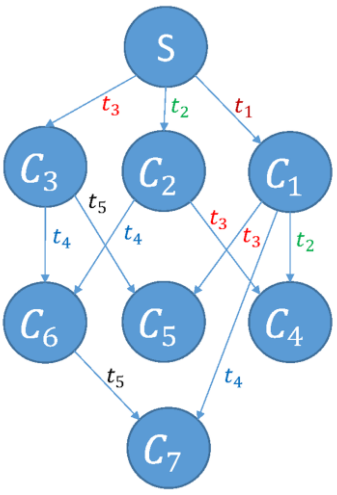

Fig. 3. Another one

Now we explain how it works to monitor available repair groups in real time. For every node u_i, we use Q_i to represent its job queue. We add a job to a node's job queue when the repair group its fragment belongs to becomes available. A node should keep working until its job queue is empty. Take Fig. 3 as example, at first all $Q_i, i \le 7$ are empty. After t_2 time-step, c_2 is generated which leads to $(c_1, c_2) \Rightarrow c_4$ available. Then we add c_4 to Q_1 and Q_2. When c_3 is generated, $(c_1, c_3) \Rightarrow c_5$ and $(c_2, c_3) \Rightarrow c_6$ become available at the same time. Then Q_1, Q_2 and Q_3 are updated accordingly. At last, we could get a transport graph G depicted in Fig. 3.

We design a greedy algorithm in Algorithm 1 to get a transport graph. We define a graph $G = (V, E)$ where V represents a set of vertexes and E represents a set of edges. We store G with an adjacency list Adj.

Algorithm 1 Greedy algorithm to get transport graph G with a $\langle n, k \rangle$ LRC

Require: a $\langle n, k \rangle$ LRC
Ensure: G
add source node s to V
for $i = 1; i < k; i{+}{+}$ do
    add u_i to V
    add u_i to $G.Adj[s]$
end for
$count = n - k$
while $count > 0$ do
    if a new available repair group $(u_i, u_j) \Rightarrow u_k$ emerges then
        add u_k to V
        add u_k to $G.Adj[u_i]$, add u_k to $G.Adj[u_j]$
        $count = count - 1$
    end if
end while
return G

In practical application, we could start transferring data before its repair group is available. For example, in Fig. 3, u_1 can transfer fragment c_1 to u_4, u_5 and u_7 once c_1 is generated. Besides, if a node needs to transfer data to several nodes, we prefer to transfer data to nodes with bigger out-degree. Fig. 4 shows the effect of out-degree. In Fig. 4, node u_3 needs to transfer data to u_5 and u_6. Out-degree of u_6 is bigger than u_5. We think node u_6 has a higher priority for the reason that it has more data to transfer.

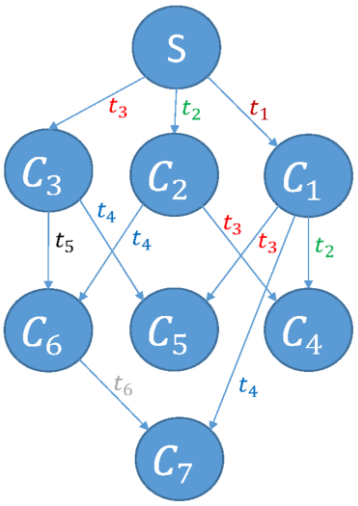

Fig. 4. Effect of out-degree

For a general $\langle n, k \rangle$ LRC, we could get a transport graph G and take it as a guide on network topology, but how to select the specific physical nodes still be an important issue. [4,5] introduced some heuristic scheduling algorithms about how to select nodes. L. Pamies-Juarez et al. [4,5] observe that *RndFlw* policy achieves significantly better results in comparison with other policies. *RndFlw* policy combines Random and Maximum Flow. Random means it select random

nodes when scheduling traffic from the source node. In Maximum Flow, nodes are sorted in descending order according to the amount of redundant data these nodes can help generate and nodes are selected according to their priority. After source node complete its transfer, Maximum Flow tries to maximize the use of those nodes that can potentially generate more redundancy. We build a two-layer system to combine transport graph and *RndFlw* which is presented in next subsection.

System Overview.

We will provide a brief overview of our system in this subsection. As depicted in Fig. 5, we build a two-layer system to combine the transport graph and *RndFlw*.

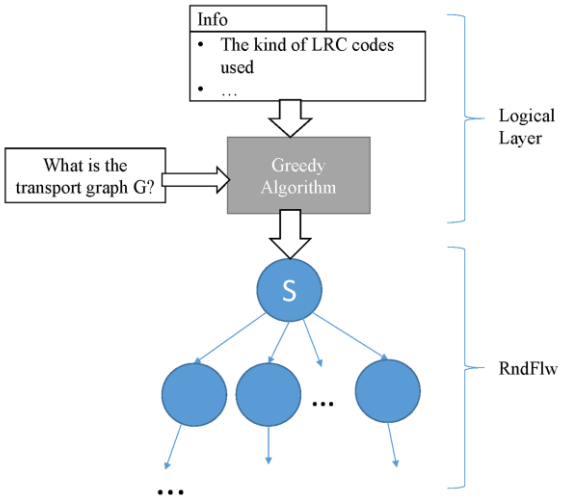

Fig. 5. System architectural

We will provide a brief overview of our system in this subsection. As depicted in Fig. 5, we build a two-layer system to combine the transport graph G and *RndFlw*.

In logical layer, we compute the transport graph G according to which kind of LRC are used. In physical layer, we select specific nodes using *RndFlw*. To explain the two layer's relationship and how they work together, we still use $\langle 7,3 \rangle$ LRC as an example. Firstly, we get $\langle 7,3 \rangle$ LRC's transport graph in logical layer which is depicted in Fig. 3. From Fig. 3, we see that source node needs to transfer data to 3 different nodes. Using *RndFlw*, we select randomly 3 node for source node. After source node's work, we know from Fig. 3 that we need 3 nodes for $(c_1, c_2) \Rightarrow c_4$, $(c_1, c_3) \Rightarrow c_5$ and $(c_2, c_3) \Rightarrow c_6$ respectively. Using *RndFlw*, we select 3 nodes with higher priority. At last, we need a node for $(c_1, c_5) \Rightarrow c_7$ and we still use *RndFlw* to select a node. Note that no matter how we select specific nodes, the transport graph G is fixed in logical layer.

IV. Evaluation

This section we evaluate the insertion performance of our system. We experiment on different size of data blocks when we insert only once and when we need to do insert operation many times continuously. We compare our system with *RndFlw*. For all experiments, the results are reported by averaging six runs.

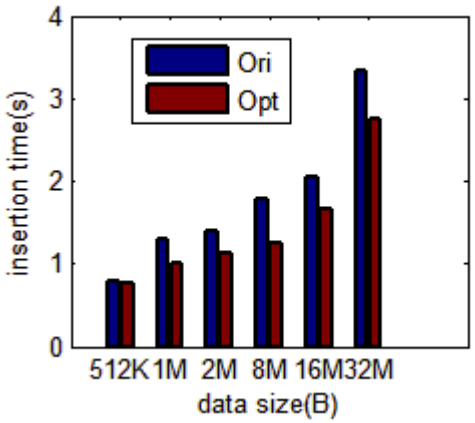

Fig. 6. Insert only once

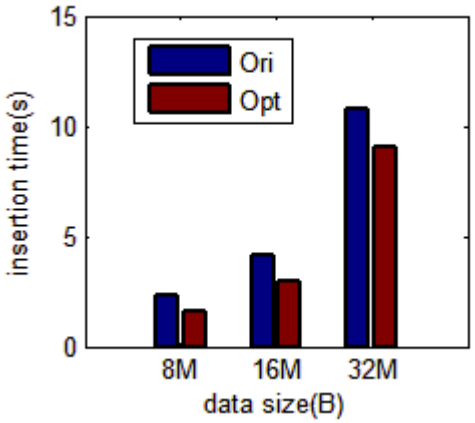

Fig. 7. Insert Five Consecutive Times

Experimental Environment.

The experimental study is conducted on an in-house cluster, consisting of 10 nodes hosted on one rack. The nodes are connected by a 1 Gbps switch. Each cluster node is equipped with a quad-core Intel Xeon 2.4GHz CPU, 8GB memory and two 500 GB SCSI disks.

Insert Only Once

We evaluate our system when we insert data just only once. Data blocks used are 512KB, 1MB, 2MB, 8MB, 16MB and 32MB. From Fig. 6, we could see our system outperforms *RndFlw* in all cases. When data size is small, our system takes almost equal time with *RndFlw* as a result of additional overhead of the logical layer.

Insert Five Consecutive Times.

In practical application, it is more likely to insert data in a streamlined way. We test insertion performance when we insert data five times consecutively. From Fig. 7, we could see our system reduce insertion time by 15%, 18% and 17% when data size is 8MB, 16MB and 32MB respectively compared with *RndFlw*.

V. CONCLUSION

In this paper, we have proposed a novel method for the scheduling of in-network redundancy generation process which allows us to improve data insertion throughput. Our method combines the transport graph and *RndFlw*. The transport graph are obtained by a greedy algorithm. Experimental results show that our method improve insertion throughput compared with *RndFlw*. As future work, we intend to analyze nodes availability's effect on our method in practical systems.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (Grant No. 61402490) and Excellent Ph.D. Dissertation Foundation of Hunan, all support is gratefully acknowledged.

REFERENCES

[1] Information on http://en.wikipedia.org/wiki/Cloud_storage

[2] Ghemawat S, Gobioff H, Leung S T. The Google file system[C]//ACM SIGOPS operating systems review. ACM, 2003, 37(5): 29-43.

[3] Huang Z, Biersack E, Peng Y. Reducing repair traffic in p2p backup systems: exact regenerating codes on hierarchical codes[J]. ACM Transactions on Storage (TOS), 2011, 7(3): 10.

[4] Pamies-Juarez L, Datta A, Oggier F. In-network redundancy generation for opportunistic speedup of data backup[J]. Future Generation Computer Systems, 2013, 29(6): 1353-1362.

[5] Pamies-Juarez L, Oggier F, Datta A. Data insertion and archiving in erasure-coding based large-scale storage systems[M]//Distributed Computing and Internet Technology. Springer Berlin Heidelberg, 2013: 47-68.

[6] Oggier F, Datta A. Self-repairing homomorphic codes for distributed storage systems[C]//INFOCOM, 2011 Proceedings IEEE. IEEE, 2011: 1215-1223..

[7] Dimakis A G, Godfrey P B, Wainwright M J, et al. The benefits of network coding for peer-to-peer storage systems[C]//Third Workshop on Network Coding, Theory, and Applications. 2007.

[8] Dimakis A G, Godfrey P B, Wu Y, et al. Network coding for distributed storage systems[J]. Information Theory, IEEE Transactions on, 2010, 56(9):4539-4551.

[9] Duminuco A, Biersack E. Hierarchical codes: How to make erasure codes attractive for peer-to-peer storage systems[C]//Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on. IEEE, 2008:89-98