Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

DDUC: an erasure-coded system with decoupled data updating and coding*

Yaofeng TU[1,2], Rong XIAO[2], Yinjun HAN‡[1,2], Zhenghua CHEN[2], Hao JIN[2], Xuecheng QI[2], Xinyuan SUN[2]

[1]*State Key Laboratory of Mobile Network and Mobile Multimedia Technology, Shenzhen 518000, China*

[2]*ZTE Corporation, Nanjing 210000, China*

E-mail: tu.yaofeng@zte.com.cn; xiao.rong1@zte.com.cn; han.yinjun@zte.com.cn; chen.zhenghua@zte.com.cn; jin.hao1@zte.com.cn; qi.xuecheng@zte.com.cn; sun.xinyuan@zte.com.cn

Received Oct. 15, 2022; Revision accepted Feb. 12, 2023; Crosschecked Mar. 31, 2023

Abstract: In distributed storage systems, replication and erasure code (EC) are common methods for data redundancy. Compared with replication, EC has better storage efficiency, but suffers higher overhead in update. Moreover, consistency and reliability problems caused by concurrent updates bring new challenges to applications of EC. Many works focus on optimizing the EC solution, including algorithm optimization, novel data update method, and so on, but lack the solutions for consistency and reliability problems. In this paper, we introduce a storage system that decouples data updating and EC encoding, namely, decoupled data updating and coding (DDUC), and propose a data placement policy that combines replication and parity blocks. For the (N, M) EC system, the data are placed as N groups of $M+1$ replicas, and redundant data blocks of the same stripe are placed in the parity nodes, so that the parity nodes can autonomously perform local EC encoding. Based on the above policy, a two-phase data update method is implemented in which data are updated in replica mode in phase 1, and the EC encoding is done independently by parity nodes in phase 2. This solves the problem of data reliability degradation caused by concurrent updates while ensuring high concurrency performance. It also uses persistent memory (PMem) hardware features of the byte addressing and eight-byte atomic write to implement a lightweight logging mechanism that improves performance while ensuring data consistency. Experimental results show that the concurrent access performance of the proposed storage system is 1.70–3.73 times that of the state-of-the-art storage system Ceph, and the latency is only 3.4%–5.9% that of Ceph.

Key words: Concurrent update; High reliability; Erasure code; Consistency; Distributed storage system
https://doi.org/10.1631/FITEE.2200466 **CLC number:** TP333

1 Introduction

In distributed storage systems, data losses usually occur due to node crashes. To maintain availability and reliability of the system, common methods, including replication (Ousterhout et al., 2010) and erasure code (EC) (Rizzo, 1997), use redundancy. Compared with replication, EC has better storage efficiency (Weatherspoon and Kubiatowicz, 2002).

EC is a fault-tolerant method, usually recorded as (N, M), where N denotes the number of data blocks and M denotes the number of parity blocks. EC encodes N original data blocks to generate M new parity blocks. The N data blocks and M parity blocks form a stripe. Using the maximum distance separable code (Pless, 1998), M missing blocks can be recovered by decoding any N known blocks.

‡ Corresponding author

* Project supported by the National Key Research and Development Program of China (No. 2021YFB3101100)

ORCID: Yaofeng TU, https://orcid.org/0000-0002-2616-2273; Yinjun HAN, https://orcid.org/0000-0001-5578-2351

EC mode can normally tolerate at most M blocks lost in the stripe. However, when updating data, the maximum number of failed blocks tolerated will be smaller than M, which leads to worse system reliability (Aguilera et al., 2005a). There are two common solutions (Peter and Reinefeld, 2012). The first one is locking the main node or each node separately to convert the concurrent updates into a sequence, which degrades the update performance. The second one is saving the updates in additional logs or caches, which ensures the reliability and update performance; however, it reduces the read performance on account of merging the data block with the updates in logs or caches. Therefore, the problem of balancing the read/write performance and reliability in a highly concurrent distributed EC system is an urgent problem to be solved.

In this paper, we propose a storage system called decoupled data updating and coding (DDUC), which decouples data updating and EC encoding. DDUC uses a hybrid redundancy mode of replication and EC to achieve high concurrency and reliability in decoupling data updating and EC encoding using an innovative placement policy and an update method. It also takes features of EC's high storage efficiency and replication's good performance in read/write into account. The main contributions include the following:

1. A placement policy that combines replicas and parity blocks is proposed to realize the decoupling of data updating and EC encoding. For the (N, M) EC system, the data are placed as N groups of $M+1$ replicas, and the redundant data blocks of the same stripe are all placed in the parity nodes, which enables the parity nodes to perform local EC encoding autonomously, without keeping the stripe states identical to each other.

2. A two-phase data update method is proposed. For the (N, M) EC system, data update is performed in phase 1 according to the $M+1$ replica mode, and EC encoding can be done independently by the parity nodes in phase 2, which solves the problem of data reliability degradation caused by concurrent updating. It also improves write performance and avoids possible read–modify–write when overwriting stripes.

3. A lightweight log mechanism based on persistent memory (PMem) hardware is proposed. Combining with the replica mode of data update, the metadata log requires only eight bytes. Therefore, it realizes high-performance metadata read/write while maintaining strong consistency using PMem hardware's features of byte addressing and eight-byte atomic writing.

2 Background and motivation

2.1 Background

We choose the Reed–Solomon (RS) code (Weatherspoon and Kubiatowicz, 2002) as a widely used EC scheme for use in this study. The nodes storing original data blocks are called data nodes, and those storing parity blocks are called parity nodes.

2.1.1 EC encoding

EC encoding modes are divided into the following two types:

1. Full encoding

Given N data blocks $(d_1, d_2, \ldots, d_N)$ and a positive integer M, the RS code generates M parity blocks $(p_1, p_2, \ldots, p_M)$. A linear (N, M) EC can be expressed as follows:

$$(p_1, p_2, \ldots, p_M) = \boldsymbol{G}_{M\times N} \cdot (d_1, d_2, \ldots, d_N), \quad (1)$$

where $\boldsymbol{G}_{M\times N}$ is the generation matrix of linear ECs.

2. Incremental encoding

Suppose that $d_i^{(1)}, d_i^{(2)} \ldots, d_i^{(r)}$ indicate a series of r write operations for the same data block d_i $(i = 1, 2, \ldots, N)$, and the parity block is p_j $(j = 1, 2, \ldots, M)$. Here, $d_i^{(0)}$ and $p_j^{(0)}$ indicate the original values of d_i and p_j, respectively. According to the incremental coding equation $\Delta p_j = G_{ij} \cdot \Delta d_i$, we derive the following:

$$\begin{aligned} p_j^{(r)} &= p_j^{(0)} + \sum_{x=1}^{r} \Delta p_j^{(x)} \\ &= p_j^{(0)} + \sum_{x=1}^{r} G_{ij} \left(d_i^{(x)} - d_i^{(x-1)} \right) \\ &= p_j^{(0)} + G_{ij} \cdot \left(d_i^{(r)} - d_i^{(0)} \right). \end{aligned} \quad (2)$$

According to this equation, the final parity block $p_j^{(r)}$ $(j = 1, 2 \ldots, M)$ depends only on $p_j^{(0)}$, $d_i^{(0)}$, and $d_i^{(r)}$ (Li et al., 2017).

2.1.2 EC update

EC update modes include the following three categories: (1) in-place update overwrites the

original data block with a new data block directly (Aguilera et al., 2005b) (Fig. 1a); (2) append update saves the update data in extra space, such as logs or caches (Ghemawat et al., 2003; Huang C et al., 2012) (Fig. 1b); (3) hybrid update uses in-place and append update schemes on data blocks and parity blocks, respectively (Fig. 1c).

The in-place update scheme has high reliability and access efficiency, but low update efficiency because of read amplification. The append update scheme has better update efficiency, but worse access efficiency, because we need to merge the update with the original data first. The hybrid update scheme has high efficiency on both data update and access, but it has a complex recovery process. Chan et al. (2014) proposed a scheme of saving the parity logs in space near the parity block to improve the recovery performance. Li et al. (2017) proposed an optimized hybrid update scheme PARIX, which improves data update performance by saving the original data in parity nodes and merging multiple update operations.

The state-of-the-art distributed system Ceph (https://docs.ceph.com/en/latest/rados/operations/erasure-code/) uses in-place update, which requires the entire stripe to be read–updated–recoded–written when overwriting. Although Ceph introduces write-caching to improve the overwrite performance, the effect is not so satisfactory. In addition, Hadoop distributed file system (HDFS, https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html) does not support hflush, hsync, concat, truncate, or append for EC files due to technical difficulties.

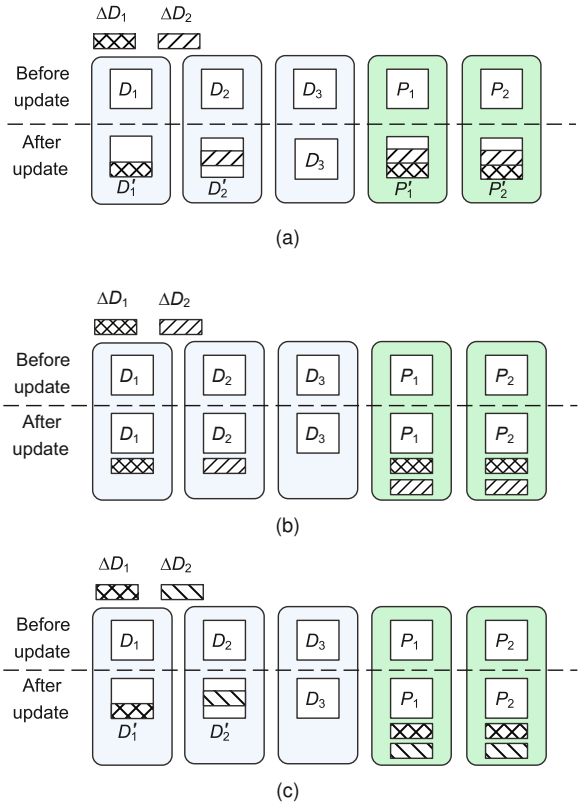

Fig. 1 Data update modes in erasure code: (a) in-place update; (b) append update; (c) hybrid update (*D*: data block; *P*: parity block)

2.1.3 Consistency

Consistency in EC is essential and necessary. In distributed storage systems, when data are updated concurrently, consistency and reliability will be harmed by node crashes or network delay (Aguilera et al., 2005a; Peter and Reinefeld, 2012).

As shown in Fig. 2, when clients update D_1 and D_2 concurrently, the data nodes send ΔD_1 and ΔD_2 (the increments of D_1 and D_2, respectively) to the parity nodes. If the messages do not reach the parity nodes in the same order, at time T_1, the values of ΔD on the two parity nodes are inconsistent, which means that the consistency of the EC stripe is reduced. If any two nodes crash at this time, the lost blocks cannot be recovered according to the blocks on the remaining nodes.

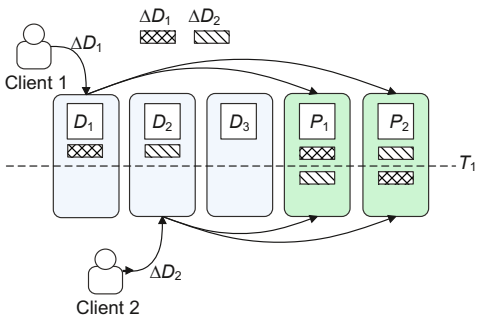

Fig. 2 Concurrent update causing consistency degradation (*D*: data block; *P*: parity block)

To maintain the consistency and reliability of the EC stripe, the data blocks and parity blocks must be updated consistently. Peter and Reinefeld (2012) listed three approaches: (1) the pessimistic sequential writing (PSW) protocol uses a master node to hold a lock of an EC stripe while updating the blocks; (2) the distributed pessimistic sequential

writing (DistPSW) protocol distributes the locking function of the master node to each data node; (3) the optimistic concurrent writing (OCW) protocol uses logs to save the updated data on both data and parity nodes and a version identity document (ID) to maintain the consistency of the stripe blocks.

In general, the PSW protocol reduces the update efficiency, but guarantees the consistency of the EC stripe. DistPSW allows more parallel write operations, but reduces data reliability and efficiency in read operations. Furthermore, OCW has better concurrent update performance than DistPSW but worse access performance than PSW.

2.2 Motivation

In conclusion, existing distributed EC systems are not yet well implemented to achieve the coexistence of high reliability and high performance in high-concurrency scenarios. The trade-off of EC systems is that to support strong consistency among EC stripes, the performance of concurrent reads and writes needs to be sacrificed, resulting in EC systems rarely being used in high-throughput scenarios. This study is dedicated to solving the problem of how to support high-concurrency reads and writes while ensuring data consistency and reliability.

In recent years, the PMem technology has been developed rapidly and is being used widely. PMem, also known as storage-class memory (SCM), has both the byte-addressable and low-latency characteristics of the ordinary dynamic random access memory (DRAM) and the non-volatile and high-capacity characteristics of persistent storage. Therefore, the PMem technology redefines the boundary between volatile and non-volatile in computer architecture. The way to give full play to the dual advantages of PMem is one of the important directions to explore in the field of high-performance storage, which also provides new opportunities for designing high-performance distributed storage systems.

3 Design

3.1 Data placement policy

There is a natural conflict between high-consistency and high-concurrency accesses. How to ensure high consistency while improving the performance of concurrent accesses is a necessary issue to be considered for distributed EC systems. In this study, we propose a placement policy allowing the existence of both replication and parity blocks to resolve the conflict by decoupling data block updating and EC encoding.

DDUC implements a hybrid scheme, in which hot data (i.e., more-frequently accessed data) use replication and cold data (i.e., less-frequently accessed data) use EC. In the data layout, the data blocks in the (N, M) EC system are managed as N groups with $M+1$ replication blocks, and the data consistency between nodes is ensured through logs. Meanwhile, the redundant blocks in the same stripe are all placed in the parity nodes, so that the parity nodes can implement EC encoding locally and switch between replica and EC modes autonomously. Hence, this method can decouple data block updating and parity block encoding in the stripe.

As shown in Fig. 3a, in a (3, 2) EC system, an EC stripe consists of three data nodes and two parity nodes. When a client writes new data, the data block is written to one data node and two parity nodes simultaneously. This is similar to the three-replica mode where the data node is the master node and the parity nodes are the slave nodes in the replica mode. Moreover, write and update operations are performed in the replica mode. The data node can take turns to be the master node in different stripes to balance the overhead. Then, the client can send write requests to different data nodes according to the placement policy.

When the data blocks on the parity node have not been updated for a while or the free space in the parity node reaches the threshold, the parity node can start EC encoding by itself. If all N data blocks of a stripe already exist in the parity node, full encoding can be performed, which means that M parity blocks can be generated according to the N data blocks. In the scheme of this study, the parity node needs only to store its own parity block, which is determined by the position of the parity node in the EC stripe. However, when conditions for full encoding are not satisfied and full encoding has been performed before, Eq. (2) can be used to calculate a new parity block, based on the old parity block, the old data block, and the new data block, which refers to the incremental encoding mode.

In the DDUC system, a stripe consists of N data nodes and M parity nodes. Parity nodes are

represented by different physical nodes in different stripes. As shown in Fig. 3b, this is a (3, 2) EC stripe. The first three nodes in stripe 1 are data nodes (DNs), followed by two parity nodes (PNs). To balance the system load, the next $N + M$ nodes in the stripe are set to be the sliding window.

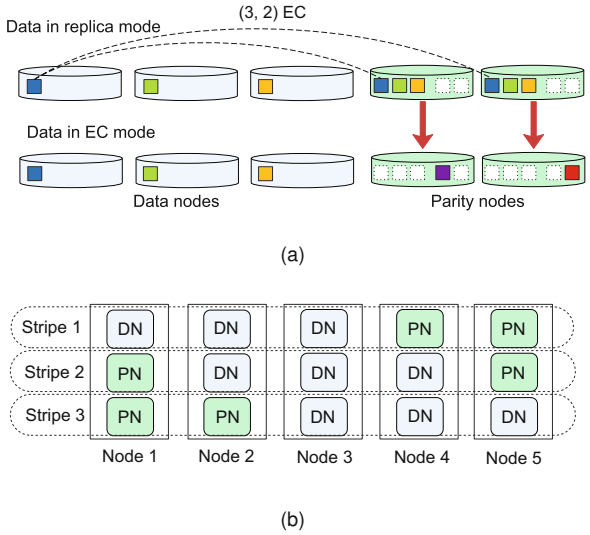

Fig. 3 Data placement: (a) data placement policy; (b) data stripe (DN: data node; PN: parity node)

3.2 Data update scheme and consistency

Based on the above strategy and placement policy, we propose a high-performance and high-reliability concurrent data update scheme to solve the problems of degradation in performance and reliability caused by concurrent data update. The data update process is shown in Fig. 4 and Algorithm 1.

For the data nodes: (1) A client updates data block D to D' and sends a write request to the master data node of the stripe according to the routing policy. (2) The master data node uses the log to record the block ID to ensure consistency when the node crashes (line 2); then, it sends the write request to all parity nodes in this stripe (line 3) and waits. (3) After receiving the pull request from the parity node, the master data node reads the original data block D and sends it to the parity node (lines 5–6). (4) After receiving the successful ACK message sent by all the parity nodes of this stripe, the master data node writes D' into the local disks (lines 8–9), replies to the client (line 10), deletes the log (line 11), and notifies the parity nodes to delete the log (line 12).

For the parity nodes: (1) After receiving the

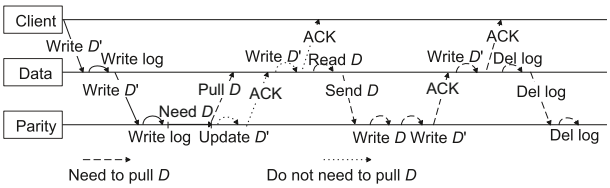

Fig. 4 Data update procedure

write request from the master data node, the parity node first records the block ID in the log (line 14) and then determines whether the data block D exists locally. If so, it directly replaces data block D with D' and replies a successful ACK message to the master data node (lines 16–17); if not, it needs to request data block D from the master data node and wait (lines 18–19). (2) After receiving the data block D, the parity node writes both the data blocks D and D' into the local disks and returns a successful ACK message to the master data node (lines 22–25). (3) After receiving the "delete log" request from the master data node, the parity node deletes the log (line 27).

Using this data update scheme, in the worst case, EC generates more network latency from an extra reading step of data block D from data nodes compared with replica mode. On the other hand, in the case of frequent updates, there is no need to read data block D from data nodes every time, so the read overhead is smaller. In addition, compared with PSW and DistPSW, which need to wait for the parity nodes to finish EC encoding and then return success to the client, our scheme considers update as completed when the data block is written to the parity node. In this case, it has better update performance. When performing the read operation, it can read the latest data block D' directly from the data nodes. Therefore, the scheme has better read and repair performance than OCW (which needs to merge logs and data before reading).

PMem can easily record and read structure-persistent data, avoid data collation and read/write amplification caused by changes in data structure, and reduce the difficulty of software implementation and improve system performance at the same time. However, compared with traditional block-based storage devices, PMem has smaller capacity and higher unit storage cost than solid-state drives (SSDs), and there is asymmetry in the read/write process of current commercial PMem. For example,

Algorithm 1 Update workflow

```
/* Update procedure within data nodes */
 1: RecvUpdateFromClient [msg]
 2:   writeMetadatalog(msg) // write metadata
 3:   SendUpdatetoParityNode(msg)
    // send msg to parity node
 4: RecvPullReqFromParityNode [msg]
 5:   Buffer origindata ← readLocal(msg)
    // read original data block
 6:   SendDatatoParityNode(msg, origindata)
    // send original data block to parity node
 7: RecvAckFromParityNode [msg]
 8:   Buffer newdata ← getData(msg)
    // obtain data from msg
 9:   writeDatablock(newdata) // write data block
10:   SendAcktoClient(msg) // return ACK to client
11:   delMetadatalog(msg) // delete metadata
12:   notifyParityDelLog(msg)
    // notify parity node to delete metadata
/* Update procedure within parity nodes */
13: RecvUpdateFromDataNode [msg]
14:   writeMetadatalog(msg) // write metadata
15:   Buffer newdata ← getData(msg)
    // obtain data from msg
16: If newdata exist, then
17:   updateData(newdata) // update data
18: Else
19:   pullOriginData(msg)
    // send pull original data msg to data node
20: End if
21: RecvOriginDataFromDataNode [msg, origindata]
22:   writeDatablock(origindata) // write original data
23:   Buffer newdata ← getData(msg)
    // obtain data from msg
24:   writeDatablock(newdata) // write data block
25:   SendAcktoDataNode(msg) // send ACK to data node
26: RecvDelLogFromDataNode [msg]
27:   delMetadatalog(msg) // delete metadata
```

the read bandwidth of Intel Optane DC PMem is 6 GB/s, while its write bandwidth is only 2 GB/s. Therefore, PMem is more suitable for storing small data, such as logs and metadata.

DDUC uses logs to ensure consistency among nodes in the replica mode. Logs are stored in the PMem hardware. First, PMem creates a log file locally to record the list of data blocks being updated in the node and appends the log file when updating. The format of the log is a 64-bit block ID, indicating that the current data block is being updated. This log needs to be deleted after the data block is updated successfully. The PMem hardware supports eight-byte atomic writes, so each update or deletion of the log requires only one atomic operation.

3.3 Encoding and reliability

When updating, the data node forwards the update message to all parity nodes in the stripe; then, the data node and all parity nodes record their logs before performing other operations. Only after all nodes in the stripe complete the data block update, can the data node notify all nodes to delete the local log records. Therefore, the existence of log indicates possible inconsistencies in the state of the data block.

Data consistency check may be triggered by events such as node power-on or network disconnection, initiated by the data node and carried out between the data node and parity nodes. The specific process is as follows: (1) The data node collects logs of itself and all parity nodes in the stripe to obtain a list of data blocks in need of consistency check. (2) According to the data block list, the data node reads the data block locally and sends it to all the parity nodes to complete data synchronization. If the data block does not exist in the data node, the parity nodes should perform the "delete" operation. (3) The parity nodes synchronize the logs to a consistent state. New log records are generated as the data node begins to accept new read and write requests.

Because the logs are written synchronously, when a data node fails, any parity node can replace the data node to initiate consistency check until the synchronization between the nodes is completed. For the parity node, the existence of logs indicates that the data block is being updated or there is inconsistency. On these occasions, EC encoding cannot be performed.

Based on the data placement policy and update method of DDUC, for an EC stripe, the data on the data node have only two states (the data block D before the update, or the data block D' after the update), and the replacement of D to D' is atomic. There are three types of data blocks on the parity node: the data block before the update (denoted by D), the data block after the update (denoted by D'), and the parity block (denoted by P). According to the different states, there may exist the following datasets on the parity node:

1. Data block set before updating, $\{d_1, d_2, \ldots, d_N\}$, which may include any block in the stripe.

2. Data block set after updating, $\{d'_1, d'_2, \ldots, d'_N\}$, corresponding to $\{d_1, d_2, \ldots, d_N\}$.

3. The parity block attributable to this node,

denoted as p_j $(j = 1, 2, \ldots, M)$.

For a data node or parity node, the possible states of its data blocks are shown in Table 1. Therefore, the EC calculation of DDUC can be completed by the parity node independently. When writing new data to the system, since the parity node saves the data blocks $d_1, d_2, \ldots, d_N$ of the entire stripe, according to Eq. (1), each parity node can calculate the parity block, and each parity node saves only its own p_j $(j = 1, 2, \ldots, M)$. In the concurrent update, the original data block d_i, the updated data block d'_i, and the parity block p_j $(j = 1, 2, \ldots, M)$ saved in the parity node, according to Eq. (2), can be used to calculate the updated parity block p'_j $(j = 1, 2, \ldots, M)$.

The blocks in the EC stripe must be consistent to ensure data reliability. The DDUC system ensures data reliability by saving both old and new versions of the data block in the parity node during concurrent updates.

Let us take (3, 2) EC as an example. Fig. 5a illustrates a schematic of a stripe that writes the new data for the first time, with each data block distributed over one data node and two parity nodes. It satisfies the fault tolerance of two, which is equivalent to the three-replica mode. Fig. 5b illustrates the data after EC encoding. Three data blocks and two parity blocks are stored in a stripe, with fault tolerance of two. If any two blocks in the stripe are lost, the lost blocks can be recovered by EC decoding. Fig. 5c illustrates the state when client 1 and client 2 concurrently update D'_1 and D'_2: the parity nodes receive D'_1 and D'_2 updates in an inconsistent order. At this moment, the old and new versions of D'_1 and D'_2 are stored in parity node 1 and parity node 2. Fig. 5d illustrates the inconsistent states of P_1 being updated to P'_1 from D'_1 and P_2 being updated to P'_2 from D'_2. At this moment, D_1 and D'_1 have been deleted from parity node 1, and D_2 and D'_2 have been deleted from parity node 2. Fig. 5e illustrates the case in which P_1 is updated to P''_1 according to D'_1 and D'_2 and P_2 is not updated yet. At this moment, the old and new versions of D'_1 and D'_2 are already deleted from the two parity nodes. Fig. 5f illustrates the state that EC encoding is performed on parity node 1 and parity node 2 again. This state equals the state of Fig. 5b.

Table 2 shows the redundancy analysis of data blocks when two nodes in the system fail in the state shown in Fig. 5d. The first row of the table shows that when data node 3 and parity node 2 fail, D'_1 has a copy in data node 1, D'_2 has a copy in data node 2 and party node 1, and D_3 can be decoded from P'_1 (by D'_1, D_2, and D_3 encoding), D_2, and D'_1 (in data node 1). It can be seen that when any two nodes fail, the data block still has a copy in the stripe or it can be recovered through EC decoding, so data reliability is guaranteed. Furthermore, the states of Figs. 5c and 5e are similar to that of Fig. 5d, and their data reliability does not degrade when any two nodes fail.

Using the placement policy of this scheme, the parity node saves both data block and parity block before and after the update to maintain the high reliability of data. Although this policy inevitably brings storage overhead of the parity node, the scheme stores only hot data in the replica mode. When the data become cold or the parity node's space threshold is reached, EC encoding is performed

Table 1 Data states in nodes

Node type	Original data block	Updated data block	Parity block	Description
Data node	d_i $(i = 1, 2, \ldots, N)$			Before update
		d'_i $(i = 1, 2, \ldots, N)$		After update
Parity node	$d_1, d_2, \ldots, d_N$			Replica mode: data are newly written
	$d_1, d_2, \ldots, d_N$	$d'_1, d'_2, \ldots, d'_N$		Replica mode: data are updated
			p_j $(j = 1, 2, \ldots, M)$	Converted to EC
	$d_1, d_2, \ldots, d_N$	$d'_1, d'_2, \ldots, d'_N$	p_j $(j = 1, 2, \ldots, M)$	The hybrid state: $d_1, d_2, \ldots, d_N$ and p_j form the EC stripe; $\{d'_1, d'_2, \ldots, d'_N\}$ is the redundant set of $\{d'_1, d'_2, \ldots, d'_N\}$ on the data node

EC: erasure code

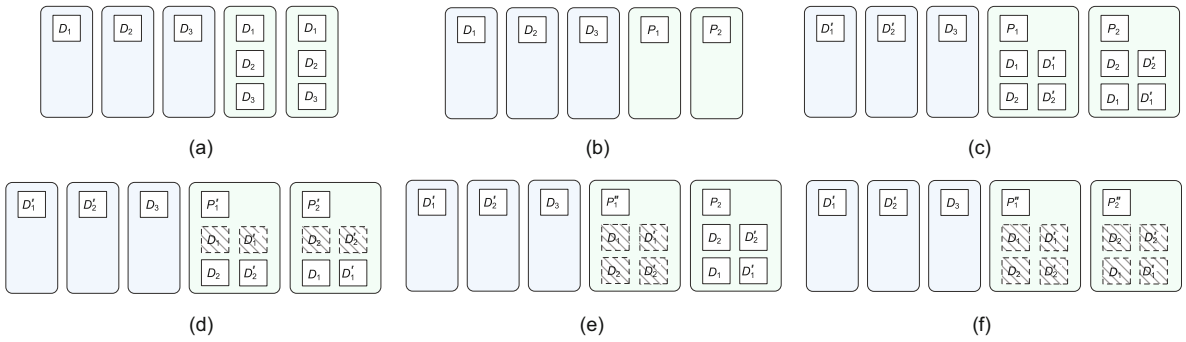

Fig. 5 Data reliability: (a) data blocks in the replica mode; (b) data blocks and parity blocks in the EC mode; (c) clients update D_1 and D_2 into D_1' and D_2', respectively; (d) parity node 1 and parity node 2 perform EC encoding inconsistently; (e) parity node 1 performs EC encoding but parity node 2 does not; (f) parity nodes go back in the EC mode again (D: data block; P: parity block; EC: erasure code)

Table 2 Reliability

DN_1	DN_2	DN_3	PN_1	PN_2	D_1'	D_2'	D_3
√	√	–	√	–	One copy in DN_1	One copy in DN_2, one copy in PN_1	Decoding from P_1' (D_1', D_2, D_3), D_2, and D_1' (in DN_1)
√	–	√	√	–	One copy in DN_1, one copy in PN_1	One copy in PN_1	One copy in DN_3
–	√	√	√	–	Decoding from P_1' (D_1', D_2, D_3), D_2, and D_3 (in DN_3)	One copy in DN_2, one copy in PN_1	One copy in DN_3
√	√	–	–	√	One copy in DN_1, one copy in PN_2	One copy in DN_2	Decoding from P_2' (D_1, D_2', D_3), D_1, and D_2' (in DN_2)
√	–	√	–	√	One copy in DN_1, one copy in PN_2	Decoding from P_2' (D_1, D_2', D_3), D_2', and D_3 (in DN_3)	One copy in DN_3
–	√	√	–	√	One copy in PN_2	One copy in DN_2	One copy in DN_3
√	–	–	√	√	One copy in DN_1, one copy in PN_2	One copy in PN_1	Decoding from P_1' (D_1', D_2, D_3), D_2, and D_1' (in DN_1). Decoding from P_2' (D_1, D_2', D_3), D_1, and D_2' (in PN_1)
–	√	–	√	√	One copy in PN_2	One copy in DN_2, one copy in PN_1	Decoding from P_1' (D_1', D_2, D_3), D_2, and D_1' (in PN_2). Decoding from P_2' (D_1, D_2', D_3), D_1, and D_2' (in DN_2)
–	–	√	√	√	One copy in PN_2	One copy in PN_1	One copy in DN_3

If any two nodes fail, the data can still be recovered, as in Fig. 5d. D: data block; DN: data node; P: parity block; PN: parity node; D_1', D_2', and D_3 are the data blocks. "√" indicates that the node is normal; "–" indicates that the node has failed

and replica mode is converted to EC mode. As for the entire system, the proportion of hot data is relatively small; hence, the extra space overhead produced by this scheme is controllable.

DDUC uses the least recently used (LRU) algorithm to generate the list of data blocks to be encoded; the most infrequently rewritten data blocks are encoded first. After encoding, only the latest parity block P_j' is kept, and D, D', and P_j are deleted to free up the parity node disk space. For the same stripe, the parity node needs to make sure that all data blocks within the stripe are not being updated before it can start encoding.

In summary, the parity node of the DDUC system can perform EC encoding autonomously, and the encoding process does not require the data node to transmit data or to maintain consistency with other parity nodes. This reduces network traffic and read amplification, in addition to solving the problem of data recoverability degradation caused by concurrent updates of data blocks. Finally, the data blocks are sorted by the LRU algorithm, and the parity node can switch data between the replica and EC modes by itself, which effectively balances performance and space utilization.

4 Implementation

In recent years, hardware products in fast iterations based on non-volatile memory express (NVMe) and PMem have been able to provide high throughput and low-latency access. Meanwhile,

application protocol stacks of high-performance software are gradually maturing, and there are increasingly higher demands on storage performance in various business application scenarios. Therefore, DDUC is a high-performance storage system based on the above new implementations of both hardware and software. The system architecture is shown in Fig. 6, which consists of several modules, such as metadata service (MDS), chunk storage daemon (CSD), and client.

1. MDS: metadata service, used mainly for cluster configuration and metadata management.

2. CSD: chunk storage daemon, responsible mainly for actual data storage.

3. Client: the client providing access interfaces for the block storage system.

MDS is responsible for the configuration management of DDUC and the coordination of cluster management operations to ensure the consistency of the distributed system. MDS is also responsible for CSD creation and status maintenance, disk management, volume routing configuration management, and so on.

CSD is responsible for the data storage of DDUC. One CSD corresponds to one storage service process, which manages several storage hardware devices on the node, including SSD, NVMe SSD, PMem, and serial advanced technology attachment (SATA). One storage server host can run multiple CSD service processes. In this study, CSD uses the NVMe equipment as its main storage device and the PMem equipment for storing metadata and logs.

Client is responsible for providing external access interfaces. Client messages are hashed to all stripes according to the MessageID, and we need one more hash to obtain the CSD to which the message belongs inside the stripe. The node where this CSD is located is called the data node of this block. All data nodes and parity nodes on this stripe can be sensed according to the MDS configuration, and DDUC places the data in the data node and parity nodes on this stripe according to the placement policy.

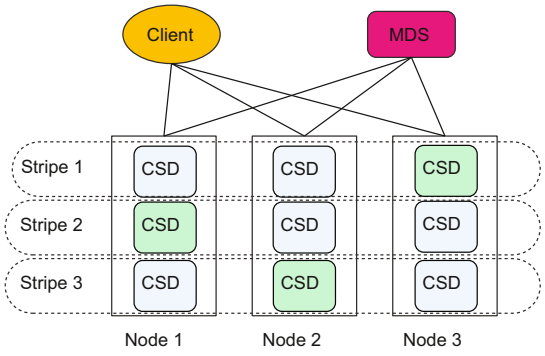

Fig. 6 System architecture (CSD: chunk storage daemon; MDS: metadata service)

5 Evaluation

The DDUC system supports both replica and EC modes. The replica mode has better performance but lower space utilization. In the case of the two-replica mode, for example, only 50% of the disk capacity can be used to store user data. In contrast, (2,1) EC provides the same fault tolerance but improves space utilization up to 67%.

Ceph is one of the most widely used distributed storage systems and supports three types of data storage: file, object, and block. Since DDUC is implemented as a block storage system, the block storage mode of Ceph is used for testing in this study. We simultaneously test the DDUC replica, DDUC EC, Ceph replica, and Ceph EC modes, and compare their read/write latency, input/output operations per second (IOPS), and space utilization horizontally to evaluate the performance of DDUC. From the user's perspective, the most likely evaluation strategy is to compare the performance and space utilization of replica and EC modes with the same fault tolerance, for example, two-replica vs. $(N,1)$ EC, or three-replica vs. $(N,2)$ EC. The larger the value of N is, the higher the space utilization will be, but usually the lower the performance is.

As for a block device, 4 KB random reads/writes and 64 KB sequential reads/writes are usually selected to evaluate latency, IOPS, and bandwidth.

5.1 Test environment

The tests use six R5300G4 servers, one as client and the others as servers. The environment configuration is shown in Table 3.

The version of Ceph is Octopus 15.2.16. The EC volume of Ceph uses the replicated pool to store meta information and the EC pool to store data information. Therefore, the performance tests of Ceph in this study use PMem to build the replica pool and NVMe disks to build the EC pool. For disabling the cache, we use FIO's RBD to run the read and write

Table 3 Test environment configuration

Configuration item	Value
CPU	Intel® Xeon® Gold 6230N CPU @2.30 GHz (×2)
DRAM	64 GB (×4)
Persistent memory	Intel AEP 256 GB (×2)
NVMe SSD	Intel P4610 1.6 TB (×4)
Network	Mellanox ConnectX-5 100 GB RoCE

tests.

DDUC uses PMem and NVMe disks to provide data storage capabilities, where PMem is used mainly for storing metadata and logs, and NVMe disks are used mainly for storing actual data. Similar to Ceph, a custom IOEngine of FIO is used to access the DDUC.

5.2 Performance

First, we test the latency (average and P99) and IOPS of 4 KB random reads/writes and 64 KB sequential reads/writes in the EC mode for DDUC. The test uses the single-thread multidepth mode to simulate concurrent scenarios; that is, the system load is increased by adjusting the I/O depth of FIO, and the I/O depth is increased by a multiple of 2 (taking the values 1, 2, 4, 8, 16, ...). This stops when the IOPS values of two adjacent tests tend to stabilize. At this point, usually accompanied by a significant increase in latency, we take the test with the lower latency as the final result and record its IOPS, average latency, and I/O depth.

Fig. 7 shows the DDUC's IOPS and latency performances in the (3, 2) EC mode at different I/O depth values. It is shown that as the I/O depth increases, the IOPS of DDUC increases gradually. The IOPS of 4 KB random reads and writes peaks when the I/O depth equals 128 and decreases slightly afterward, while the latencies increase significantly. The IOPS of 64 KB sequential writes peaks when the I/O depth equals 128, and its sequential read slows down after the I/O depth reaches 128, while the latencies of both write and read increase significantly; therefore, we select I/O depth to be 128 for DDUC's subsequent tests. We test Ceph in the same way and select I/O depth to be 128 for subsequent tests as well.

For comparison, we test the average latency and IOPS of 4 KB random reads and 64 KB sequential reads in Ceph three-replica, Ceph (3, 2) EC, and

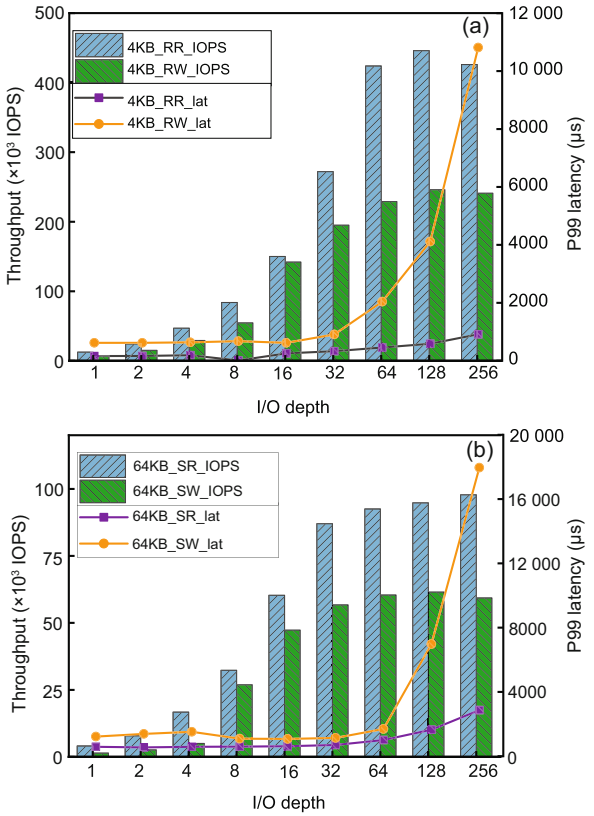

Fig. 7 Concurrency of DDUC: (a) 4 KB random reads/writes of (3, 2) EC; (b) 64 KB sequential reads/writes of (3, 2) EC (DDUC: decoupled data updating and coding; EC: erasure code)

DDUC three-replica modes. The values and test procedures of I/O depth are the same as in the DDUC (3, 2) EC mode.

Fig. 8 shows the IOPS comparison of 4 KB random reads/writes and 64 KB sequential reads/writes in different modes: three-replica and (3, 2) EC modes of DDUC and Ceph.

Evaluation results show that the IOPS of the write operations of the DDUC system is significantly higher than that of Ceph. As shown in Figs. 8a and 8b, in the (3, 2) EC mode, the IOPS values of 4 KB random writes and 64 KB sequential writes of DDUC are 3.73 and 3.24 times those of Ceph, respectively. Ceph's EC write process uses the primary object storage device (OSD) to perform data slicing and encoding, and the overwrite process requires the entire stripe to be read–modified–encoded–written, which has a significant impact on the performance. The write process of DDUC is similar to that of replica, but the difference is that it requires at most one trip to the data node to pull the original data block, so the overall performance is significantly higher than that

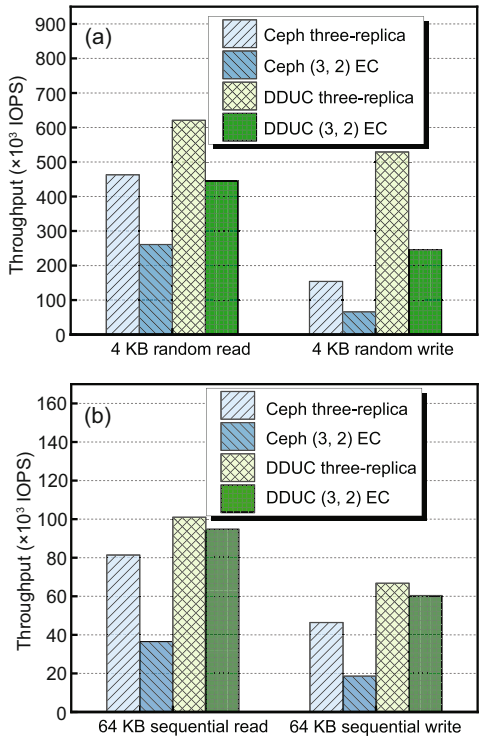

Fig. 8 IOPS comparison: (a) 4 KB random reads/writes of three-replica vs. (3, 2) EC; (b) 64 KB sequential reads/writes of three-replica vs. (3, 2) EC (IOPS: input/output operations per second; EC: erasure code)

for Ceph. The 4 KB random write and 64 KB sequential write IOPS values of DDUC's (3, 2) EC mode decrease by 53.4% and 9.6% respectively compared to its replica mode. The main reason for this decline is the overhead incurred by pulling the original data block from the data nodes. The random write performance drops more than that of the sequential write because the random write is not concentrated in terms of heat. The parity node deletes the original data block after EC encoding, and the subsequent writing of this block requires pulling the original data block again, where concentrated sequential write is less likely to occur. Compared to the replica mode, the IOPS values of Ceph decrease by 57.2% and 59.9%, respectively, for the random and sequential writes. The main reason for this is the complexity of the writing process and the dependence on the primary agent. In addition, Ceph's metadata write is in the replica pool and data write is in the EC pool, while DDUC's metadata and data writes are combined in one message interaction; moreover, the network and stack overhead is smaller than that in Ceph.

In the meantime, the IOPS of the read operations of the DDUC system is significantly higher than that of Ceph. As shown in Figs. 8a and 8b, in the (3, 2) EC mode, the IOPS values of 4 KB random reads and 64 KB sequential reads of DDUC are 1.70 and 2.59 times those of Ceph, respectively. Compared with the replica mode, the 4 KB random read and 64 KB sequential read IOPS values of DDUC's (3, 2) EC mode decrease by only about 28.2% and 6.1%, respectively. This is because DDUC's strategy of hot data in the replica and cold data in the EC enables the data node directly respond to the client, which can fully retain the high-performance characteristics of the replica read operation. The small drop in IOPS relative to replica is due to the stack overhead generated by the more complex routing strategy for finding the data blocks in the EC mode, while 64 KB reads have a smaller drop than 4 KB reads because the metadata query overhead is diluted. The read IOPS values of Ceph decrease by about 43.6% and 55.1% respectively compared to its replica mode, for the reason that Ceph's placement policy makes the read process in need of aggregating each slice before responding to the client, which has a large impact on the performance.

Fig. 9 shows the latency performance comparison of 4 KB random reads/writes and 64 KB sequential reads/writes at peak IOPS in the three-replica and (3, 2) EC modes for DDUC and Ceph.

As shown in Fig. 9, the advantage of DDUC's write latency is obvious. In the (3, 2) EC mode, the latencies of DDUC's 4 KB random writes and 64 KB sequential writes are 3.4% and 4.0% those of Ceph, respectively. Ceph's write operations use the primary OSD for slicing and EC encoding in real time and the read–write–modify (RWM) for the overwrite process, which involves huge disk I/O and network overhead. Specifically, the data update operations on each node are also performed serially to write logs (with old data) and write data to avoid abnormal power outages that cause data blocks to be lost, which further increases latency. The DDUC placement strategy, on the other hand, makes the update method approach the replica mode. In addition, the lightweight PMem-based logging strategy eliminates double-writing of metadata and data, so the overall latency is much lower than that for Ceph.

The latencies of Ceph's 4 KB random writes and 64 KB sequential writes both increase by more

than a factor of 1, while the latencies of DDUC increase by 112.5% and 13.6% respectively compared to its replica mode. Compared to replica, there is no significant increase in the DDUC's 64 KB latency. The 4 KB latency increases because the initial write requires pulling the original data block in the data node, while 4 KB random writes have a greater chance of requiring pulling of the original data block due to the lack of heat concentration.

Meanwhile, the read operation latency of DDUC is significantly lower than that of Ceph. In the (3, 2) EC mode, the latencies of DDUC's 4 KB random reads and 64 KB sequential reads are 5.9% and 4.8% those of Ceph's, respectively. Ceph's EC read process requires primarily to calculate the read range for each slice, then to read from each OSD separately, and finally to aggregate and answer to the client. This causes Ceph's read latency to increase by more than a factor of 1 compared to its own replica mode. The read process of DDUC answers to the client directly from the data node; so, the overall latency is much lower than that of Ceph.

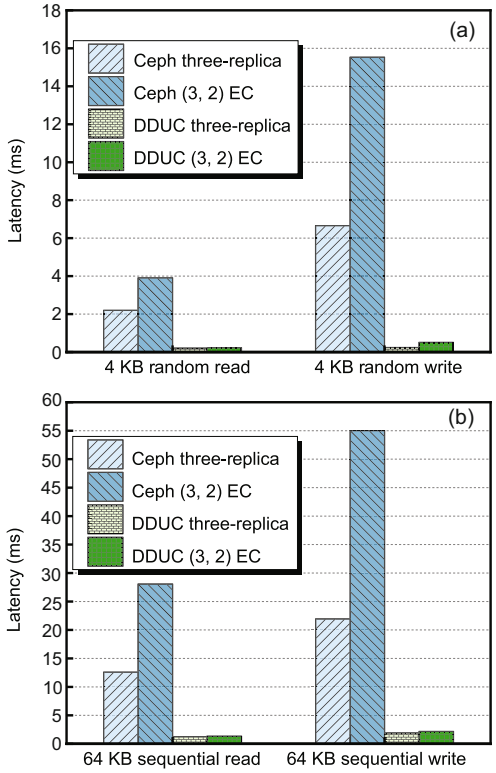

Fig. 9 Latency comparison: (a) 4 KB random reads/writes of three-replica vs. (3, 2) EC; (b) 64 KB sequential reads/writes of three-replica vs. (3, 2) EC (EC: erasure code)

5.3 Disk space recycling

The DDUC system adopts a hybrid mode of replica and EC, and when to perform EC encoding is affected by the popularity of data, disk space, and other factors.

This test uses one disk for each of the three nodes configured in the (2, 1) EC mode and writes 20 GB of data randomly in 4 KB size. Each node is configured to use only one 50-GB partition instead of its full capacity. In addition to the LRU scenarios (i.e., EC-LRU), we test the EC-encoding scenarios with fixed concurrency (the number of concurrent EC encoding processes is fixed and the LRU algorithm is not used, i.e., EC-fixed) and scenarios with EC encoding disabled (simulating all hot data, i.e., EC-disabled) as the comparison groups. Therefore, the balance between the performance and space efficiency of the disk space recycle mechanism can be observed.

As shown in Fig. 10, we keep observing for 300 s to confirm the changes of IOPS and disk space. The client stops after completing the write of 20-GB data, and the end time of the write operation varies among the three scenarios due to the difference in IOPS.

It can be observed that the IOPS continues to increase in all three scenarios. This is because there is a space allocation operation for the first write of an empty volume. Since DDUC uses a preallocation mechanism triggered by requests, this overhead will decrease rapidly with the write operation. As a result, the subsequent IOPS continues to increase. For the EC-disabled process, since it is not affected by encoding, IOPS increases rapidly in a near-linear manner. After 15 s, affected by the background EC encoding, the IOPS values of the three scenarios begin to show differences. For the EC-LRU process, the IOPS improvement is slightly

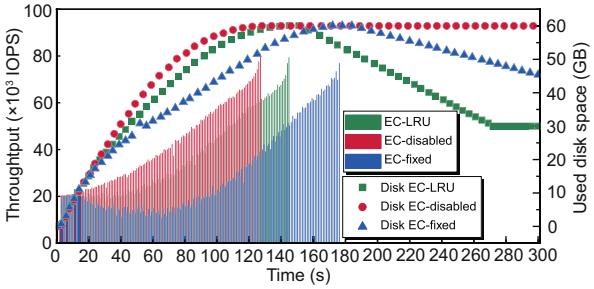

Fig. 10 Disk space efficiency (References to color refer to the online version of this figure)

later than for EC-disabled, and the overall curve is smoother and shows an upward trend, which does not cause a significant negative impact on the IOPS. The IOPS values of EC-fixed first decrease and then increase. Backend EC encoding and user write operations generate resource contention, resulting in user-perceivable performance degradation.

In these three scenarios, as the data are being written to the disk, the used space increases rapidly and finally reaches the peak of 60 GB, exceeding the theoretical value of 40 GB for the two-replica mode. This is because the parity node additionally stores old and new versions of the data block. In the rising phase, due to the differences in IOPS, the disk space of EC-disabled increases the fastest, followed by that of EC-LRU and EC-fixed in sequence. In the falling phase, since EC encoding is not performed, the disk space of EC-disabled remains unchanged, while the disk usages of EC-LRU and EC-fixed gradually decrease after EC encoding starts. In EC-LRU, disk space drops faster and finally reaches 30 GB, while disk space efficiency reaches 67.7% that of (2, 1) EC. This is because after the system disk space reaches the threshold, EC-LRU dynamically adjusts the parameters to recycle disk space faster. However, EC-fixed is limited by the degree of concurrency; so, the encoding speed is low. The disk space drops to 45 GB and the encoding operation of all data is not completed during the observation period.

5.4 Workload testing

To compare the performances of DDUC and Ceph under business load scenarios, this experiment uses the TPC-C benchmark test with DDUC and Ceph as the back-end storage of PostgreSQL database and uses benchmarksql-5.0 to test the transaction performance of the database. This experiment configures both DDUC and Ceph as (3, 2) EC (with the TPC-C parameter WareHouse=100), imports 10 GB of business data to PostgreSQL, and simulates the load pressure on the database by modifying the number of concurrent threads to test the throughput and latency of transactions under different load scenarios.

The throughput and P90 latency in the TPC-C test are shown in Fig. 11, where throughput tpmC is the number of transactions executed per minute (transactions per minute). As is shown, the performance of DDUC-based database transactions is significantly better than that of Ceph-based database transactions. When the concurrency reaches 32, the tpmC performance of DDUC is 3.67× that of Ceph, and the P90 latency of DDUC is only 30% of Ceph. As the number of concurrent threads increases, the tpmC performances of both grow linearly, with DDUC's growth slope being greater. The higher the concurrency, the more obvious the performance advantage. The main reason is that the hybrid model of DDUC's replica+EC mode retains high concurrency and low latency of replica to the greatest extent, which supports the PostgreSQL database transaction's parallel processing capability in a better way.

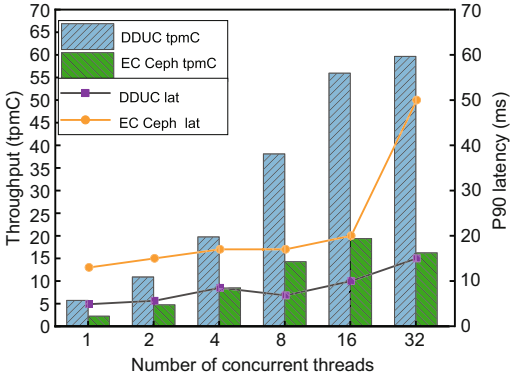

Fig. 11 TPC-C test (tpmC: transactions per minute, TPC-C; lat: latency)

Test results show that the scheme of decoupling data updating and EC encoding adopted by DDUC can retain the advantages of high performance of the replica mode. Compared with the current state-of-the-art distributed system Ceph, the performance of DDUC in the EC mode has great advantages. Although saving replicas of hot data may lead to waste in disk usage space during a short period, DDUC fixes this issue well.

6 Related works

In recent years, EC-based optimization works have focused mainly on several aspects:

1. Data update schemes

Comparable to our work, Li et al. (2017) saved original data in the parity nodes, but their work differs in that its consistency was guaranteed mainly by locks. Huang JZ et al. (2019) proposed a method by grouping and merging update operations to reduce the read amplification. Wang YJ et al. (2018)

constructed an adaptively updated high-efficiency tree structure to pass update data from the top to the bottom.

2. Placement policies

Partially similar to our idea, Konwar et al. (2017) proposed a tiered distributed system: Tier-1 uses the replica mode to store frequently updated data, while Tier-2 uses the EC mode to store unchanging data. Xiong et al. (2021) improved space utilization by continuously hashing the path for data block placement. Jiang et al. (2021) proposed a new redundant array of independent disks (RAID) architecture as a shared storage pool, where requests were spread to all SSDs to achieve low latency on commodity SSD arrays.

3. Data transfer reduction

Shen and Lee (2018) proposed a cross-rack-aware mechanism to reduce data transfer and solved the reliability degradation problem during concurrent updates by storing temporary replicas. Gong et al. (2021) proposed a new rack-coordinated mechanism to suppress cross-rack update traffic. Pu et al. (2020) proposed a multi-data-node mechanism based on the ant colony optimization algorithm to relieve traffic when updating. Peter and Reinefeld (2012) proposed a software-defined cooperatively controlled mechanism, which can balance network load using an optimized link selection algorithm. Wang F et al. (2019) used programmable network devices to perform "exclusive OR" (XOR) operations in multiple storage nodes to reduce network traffic and eliminate network bottlenecks effectively.

4. Algorithm improvement

Liu et al. (2021) dynamically selected a write scheme with fewer XORs for each parity block to be updated to improve the update performance. Meng et al. (2019) proposed DLRC (dynamic local reconstruction code) based on the idea of grouping codes. By adjusting the parameter values, a dynamic balance among storage overhead, fault tolerance, and reconstruction overhead was achieved.

7 Conclusions

In this paper, we have addressed the consistency problem in concurrent updating of EC in existing distributed storage systems, analyzed the principles for their generation and basic solutions, and summarized the latest research progress and the advantages and disadvantages. In this way, we have proposed a storage system called DDUC, which decouples data updating and EC encoding.

It realizes decoupling in data block updating and parity block encoding through the application of an innovative data placement policy, an update method, and the lightweight log mechanism based on PMem, so that parity nodes can transform the state of data blocks between replica and EC modes when the data are hot or cold. It also performs EC encoding independently without locking the whole stripe or sorting the updates. This not only supports highly concurrent data update but also ensures high data reliability at the same time. The system also achieves balance between performance and space efficiency using a space recycle algorithm. Experimental results showed that the performance optimization is obvious.

Contributors

Yaofeng TU designed the research. Yinjun HAN and Zhenghua CHEN completed the detailed design. Rong XIAO drafted the paper. Hao JIN implemented the scheme. Xuecheng QI processed the data. Xinyuan SUN revised and finalized the paper.

Compliance with ethics guidelines

Yaofeng TU, Rong XIAO, Yinjun HAN, Zhenghua CHEN, Hao JIN, Xuecheng QI, and Xinyuan SUN declare that they have no conflict of interest.

Data availability

Due to the nature of this research, participants of this study did not agree for their data to be shared publicly, so supporting data are not available.

References

Aguilera MK, Janakiraman R, Xu LH, 2005a. On the erasure recoverability of MDS codes under concurrent updates. Proc IEEE Int Symp on Information Theory, p.1358-1362. https://doi.org/10.1109/ISIT.2005.1523564

Aguilera MK, Janakiraman R, Xu LH, 2005b. Using erasure codes efficiently for storage in a distributed system. Int Conf on Dependable Systems and Networks, p.336-345. https://doi.org/10.1109/DSN.2005.96

Chan JCW, Ding Q, Lee PPC, et al., 2014. Parity logging with reserved space: towards efficient updates and recovery in erasure-coded clustered storage. Proc 12th USENIX Conf on File and Storage Technologies, p.163-176.

Ghemawat S, Gobioff H, Leung ST, 2003. The Google file system. Proc 19th ACM Symp on Operating Systems

Principles, p.29-43. https://doi.org/10.1145/945445.945450

Gong GW, Shen ZR, Wu SZ, et al., 2021. Optimal rack-coordinated updates in erasure-coded data centers. 40th IEEE Conf on Computer Communications, p.1-10. https://doi.org/10.1109/INFOCOM42981.2021.9488813

Huang C, Simitci H, Xu YK, et al., 2012. Erasure coding in Windows Azure Storage. Proc USENIX Annual Technical Conf, p.2.

Huang JZ, Xia J, Qin X, et al., 2019. Optimization of small updates for erasure-coded in-memory stores. *Comput J*, 62(6):869-883. https://doi.org/10.1093/comjnl/bxz003

Jiang TY, Zhang GY, Huang ZC, et al., 2021. FusionRAID: achieving consistent low latency for commodity SSD arrays. 19th USENIX Conf on File and Storage Technologies, p.355-370.

Konwar KM, Prakash N, Lynch N, et al., 2017. A layered architecture for erasure-coded consistent distributed storage. Proc ACM Symp on Principles of Distributed Computing, p.63-72. https://doi.org/10.1145/3087801.3087832

Li HB, Zhang YM, Zhang ZM, et al., 2017. PARIX: speculative partial writes in erasure-coded systems. Proc USENIX Annual Technical Conf, p.581-587.

Liu YJ, Wei B, Wu JG, et al., 2021. Erasure-coded multi-block updates based on hybrid writes and common XORs first. 39th Int Conf on Computer Design, p.472-479.

Meng YL, Zhang LL, Xu D, et al., 2019. A dynamic erasure code based on block code. Proc Int Conf on Embedded Wireless Systems and Networks, p.379-383.

Ousterhout J, Agrawal P, Erickson D, et al., 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Oper Syst Rev*, 43(4):92-105. https://doi.org/10.1145/1713254.1713276

Peter K, Reinefeld A, 2012. Consistency and fault tolerance for erasure-coded distributed storage systems. Proc 5th Int Workshop on Data-Intensive Distributed Computing, p.23-32. https://doi.org/10.1145/2286996.2287002

Pless V, 1998. Introduction to the Theory of Error-Correcting Codes (3rd Ed.). John Wiley & Sons, Hoboken, USA. https://doi.org/10.1002/9781118032749

Pu WJ, Chen NJ, Zhong QW, 2020. SDCUP: software-defined-control based erasure-coded collaborative data update mechanism. *IEEE Access*, 8:180646-180660. https://doi.org/10.1109/ACCESS.2020.3028381

Rizzo L, 1997. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Comput Commun Rev*, 27(2):24-36. https://doi.org/10.1145/263876.263881

Shen ZR, Lee PPC, 2018. Cross-rack-aware updates in erasure-coded data centers. Proc 47th Int Conf on Parallel Processing, Article 80. https://doi.org/10.1145/3225058.3225065

Wang F, Tang YJ, Xie YW, et al., 2019. XORInc: optimizing data repair and update for erasure-coded systems with XOR-based in-network computation. Proc 35th Symp on Mass Storage Systems and Technologies, p.244-256. https://doi.org/10.1109/MSST.2019.00005

Wang YJ, Pei XQ, Ma XK, et al., 2018. TA-update: an adaptive update scheme with tree-structured transmission in erasure-coded storage systems. *IEEE Trans Parall Distrib Syst*, 29(8):1893-1906. https://doi.org/10.1109/TPDS.2017.2717981

Weatherspoon H, Kubiatowicz JD, 2002. Erasure coding vs. replication: a quantitative comparison. 1st Int Workshop on Peer-to-Peer Systems, p.328-337. https://doi.org/10.1007/3-540-45748-8_31

Xiong YL, Zhou J, Su L, et al., 2021. ECCH: erasure coded consistent hashing for distributed storage systems. IEEE Int Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, p.177-184. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00036