计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
2025,34(2):111−121 [doi: 10.15888/j.cnki.csa.009761] [CSTR: 32024.14.csa.009761]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

面向纠删码的高性能冗余转换机制①

柏志伟, 吕　敏

(中国科学技术大学 计算机科学与技术学院, 合肥 230027)
通信作者: 吕　敏, E-mail: lvmin05@ustc.edu.cn

摘　要: 分布式存储系统采用纠删码来实现高可靠和低开销的数据存储. 为了提供不同的可靠性和多样的访问性能, 存储系统需要对纠删码数据进行冗余转换, 即改变其编码参数. 条带合并机制为存储系统的冗余转换提供了一种思路. 然而, 基于传统纠删码的条带合并会在过程中引发大量的数据块重分布和校验块重计算 I/O 开销, 且在多次合并中会进一步加剧 I/O. 针对此问题, 本文提出了一种新的树型里德-所罗门 (*TRS*) 码, 通过分散数据块以消除数据块重分布 I/O, 并通过设计编码矩阵以节约校验块重计算 I/O. 树型里德-所罗门码进一步设计了存储单元, 将参与合并的条带组织成一棵树, 使得多次合并依据树结构自底向上高效完成. 本文设计实现了分布式存储原型系统. 实验表明, 树型里德-所罗门码相较于传统纠删码, 可以大大减少条带合并的完成时间.

关键词: 存储系统; 纠删码; 冗余转换; 条带合并; I/O 开销

引用格式: 柏志伟,吕敏.面向纠删码的高性能冗余转换机制.计算机系统应用,2025,34(2):111–121. http://www.c-s-a.org.cn/1003-3254/9761.html

High Performance Redundancy Transitioning Scheme for Erasure Coding

BAI Zhi-Wei, LYU Min

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract: Distributed storage systems achieve high-reliability and low-overhead data storage by erasure code. To provide different reliability and access performance, storage systems need to perform redundancy transitions on erasure code data by changing coding parameters. The stripe merging mechanism provides a way for redundancy transitioning in storage systems. However, the stripe merging process based on traditional erasure code can result in a large amount of data block redistribution and checksum block re-computation I/O overhead. Worst still, the I/O will be amplified in multiple merging operations. In response to these issues, this study proposes new Tree Reed-Solomon (*TRS*) codes that eliminate data block redistribution I/O by decentralizing data blocks, and save checksum block re-computation I/O by designing coding matrices. *TRS* codes further design storage units to organize the stripes taking part in merging into a tree, enabling multiple merging operations to be efficiently completed from bottom to top based on tree structure. To test the performance of *TRS* codes, this study designs and implements a distributed storage prototype. Experiments have shown that compared to other erasure codes, *TRS* codes can greatly reduce stripe merging operation time.

Key words: storage system; erasure coding; redundancy transitioning; stripe merging; I/O overhead

目前, 企业越来越多地采用分布式存储系统[1−3]以应对快速增长的数据. 其中一个关键问题是, 随着规模增大, 系统变得更易发生错误[4]. 因此, 大规模分布式存储系统需要具备容错能力, 以有效应对频繁发生的硬件故障、软件错误、网络故障[5]. 为了容错, 系统引入冗余数据. 传统的冗余构建方式是多副本, 它实现简单,

① 基金项目: 国家自然科学基金面上项目 (62172382)
收稿时间: 2024-06-28; 修改时间: 2024-07-25, 2024-08-20; 采用时间: 2024-08-27; csa 在线出版时间: 2024-11-28
CNKI 网络首发时间: 2024-11-29

© 中国科学院软件研究所 http://www.c-s-a.org.cn

性能较好, 然而存储开销过大, 难以适用于 ZB 级别的海量数据[6–8]. 纠删码是另外一类冗余构建方式, 通过编码原始数据块产生额外的校验块来容错. 相较于多副本, 纠删码在实现相同容错能力的条件下, 可以大大降低存储开销[8–11], 因此被广泛部署于分布式存储系统[12–15]. 在众多纠删码中, 里德-所罗门 (Reed-Solomon, *RS*) 码[16,17]是最为知名的一种, 它将 k 个数据块编码产生 m 个校验块, 这 $k+m$ 个块称为一个条带, 并通过访问条带内的任意 k 个块即可重构出原始数据块.

分布式存储系统的工作负载表现出高度倾斜的访问模式, 其中一小部分热数据被频繁访问, 而其余大部分冷数据则很少被访问[18–20]. 此外, 不同用户、不同数据类型、生命周期内不同阶段的数据对存储开销、访问性能、可靠性的需求动态多变[4,21,22]. 因此, 存储系统需要具备弹性能力: 对于热数据, 需要部署某种纠删码以实现好的访问性能; 而对于冷数据, 则需要部署其他纠删码以节约存储空间. 为了实现弹性, 存储系统需要对纠删码数据进行冗余转换, 即改变纠删码的编码参数 k 和 m. 传统的冗余转换策略[14](从 (k, m) 转换成任意的 (k', m)) 会引发大量的 I/O 开销. 新型的条带合并[4,23–26]在 I/O 性能上更有优势. 此外, 存储系统的数据热度是随着时间降低的[27–29]. 一个好的方式是采用多次条带合并: 先采用小参数条带用于热数据, 然后合并成若干中间条带用于温数据, 最后合并成少量大条带用于冷数据.

然而, 条带合并尤其是多次合并会产生大量的数据块重分布和校验块重计算 I/O 开销. 首先, 由于不同小条带的数据块和校验块可能聚集于相同节点上, 为了保证合并后条带的容错能力, 存储系统需要进行大量的数据块重分布 I/O 操作. 此外, 由于合并之后纠删码的编码矩阵发生改变, 系统需要重计算校验块, 而校验块重计算需要跨节点访问老的校验块加上大量数据块才能完成. 而多次合并则进一步加剧了系统的 I/O 开销. 已有的里德-所罗门码[16]在每一次的条带合并过程中均会产生大量的 I/O 开销. 弹性里德-所罗门码[24]虽然针对第 1 次条带合并做了优化, 仍会在后续合并中引起大量的 I/O 开销.

针对此问题, 本文设计了一种新型的编码方案, 称为树型里德-所罗门码, 具有以下特点: (1) 树型里德-所罗门码首先设计参与合并小条带的数据布局, 通过分散放置小条带的数据块, 使得任何一次合并都不引起数据块重分布的 I/O 开销. (2) 树型里德-所罗门码其次设计小条带的编码矩阵, 通过分别利用一个大编码矩阵不同位置的子矩阵, 使得任何一次合并仅需读取老的校验块即可完成校验块重计算操作, 而校验块重计算可以仅访问存储校验块的节点即可完成. (3) 为了有效组织和管理参与合并的小条带, 树型里德-所罗门码设计了存储单元的逻辑结构, 将小条带组织为一棵树, 使得条带合并从叶子节点开始到根结点结束, 按照从子节点到父节点的方式, 依次进行.

本文进一步设计了一个分布式存储原型系统, 实现了本文的树型里德-所罗门码、传统的里德-所罗门码[16]和近期提出的弹性里德-所罗门码[24]. 实验表明, 相较于里德-所罗门码, 树型里德-所罗门码可以将总的合并时间最多减少 86%, 并将单次合并时间最多减少 88%. 相较于弹性里德-所罗门码, 树型里德-所罗门码可以将总的合并时间最多减少 80%, 并将单次合并时间最多减少 88%.

1 背景与挑战

1.1 里德-所罗门码

里德-所罗门 (*RS*) 码是纠删编码中最常用的编码之一, 它的本质是将原始数据块编码产生若干额外的校验块, 并利用校验块来进行容错. 一个参数为 (k, m) 的 *RS* 码将一个对象拆分为 k 个数据块, 并计算产生 m 个校验块, 且 $k+m$ 个块中的任意 k 个可以重构出原始数据块; 这 $k+m$ 个块一起称为一个条带. 实际存储时, 系统将一个条带分散放置于 $k+m$ 个节点, 从而使得系统能够容任意 m 个节点出错. 实际系统通常包含许多条带, 每个条带都是独立进行编解码的. 图 1 表示了一个参数为 (k=3, m=2) 的 *RS* 码.

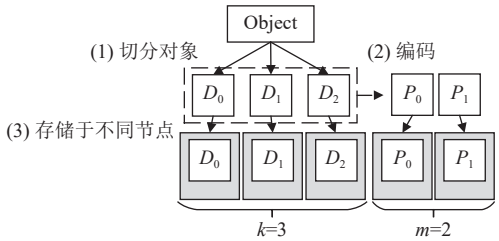


图 1 *RS* 编码过程

RS 码的编码过程可以用矩阵 $A_\{m\times k\}$来表示. 它将 A 乘以数据块向量 D, 产生校验块向量 P. 例如式 (1) 是参数为 *RS*(9, 3) 的编码公式.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

$$\begin{bmatrix}P_0\\P_1\\P_2\end{bmatrix}=\begin{bmatrix}1&1&1&1&1&1&1&1&1\\1&2&3&4&5&6&7&8&9\\1&2^2&3^2&4^2&5^2&6^2&7^2&8^2&9^2\end{bmatrix}\times\begin{bmatrix}D_0\\D_1\\D_2\\D_3\\D_4\\D_5\\D_6\\D_7\\D_8\end{bmatrix} \tag{1}$$

1.2 冗余转换与条带合并

为提供不同的可靠性和多样的访问性能, 存储系统需要对纠删码数据进行冗余转换, 即改变纠删码的编码参数 k 和 m. 传统冗余转换策略[14](从 (k, m) 转换成任意的 (k', m)) 会引发大量的数据块重分布和校验块重计算 I/O 开销. 近期的研究采用条带合并方法[4,23–26]进行冗余转换. 条带合并方法可大大节省数据块重分布的 I/O 开销, 且可以同时改变多个条带的冗余度, 相较于传统方法更有优势. 因此, 本文采用条带合并来进行冗余转换.

条带合并指将 x 个参数为 (k, m) 的小条带, 合并成一个参数为 (xk, m) 的大条带. 由于实际系统的数据热度是随着时间逐步变冷的, 因此一个好的方式是多次合并: 首先采用多个小条带用于热数据, 其次采用较少的中间条带用于温数据, 最后采用更少的大条带用于冷数据. 图 2 展示了 3 个 $RS(2, 1)$ 的小条带, 经过 2 次合并, 最终合并为一个 $RS(6, 1)$ 的大条带.

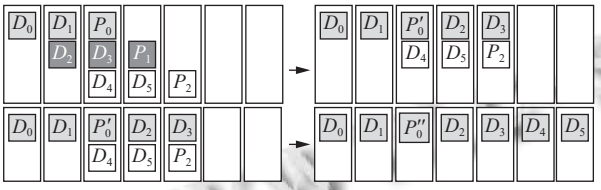


图 2 条带合并过程举例

1.3 弹性里德-所罗门码

为了有效应对冗余转换, 文献[24]提出了弹性里德-所罗门码 (*ERS*). *ERS* 码的参数为 (k, m, k'), 它改变了传统 *RS* 码中数据块和节点一一对应的存储方式, 将 k 个数据块以扩展的方式存放在 $k'>k$ 个节点上, 以此消除冗余转换时的数据块重分布操作. 除此之外, *ERS* 码通过对转换前后编码矩阵的设计, 使得转换后的新校验块可以由转换前的老校验块加上少量额外的数据块更新计算得出, 这样就不需要读取所有数据块, 降低了转换的 I/O 开销.

图 3 展示了参数为 (2, 1, 3) 的 *ERS* 码, S_0–S_2 表示条带, X_0–X_3 表示节点. 冗余转换前的 k 为 2, 转换后的 k'为 3, m 为 1. 方案首先计算 k 和 k'的最小公倍数为 6, 然后将原始对象分为 6 个数据块, 将数据块按行优先顺序存入 3 个节点中. 接下来根据参数 $RS(2, 1)$ 计算出 3 个校验块, 并存入 1 个节点中. 由于 $RS(2, 1)$ 的数据块扩展地放置在 3 个节点上, $RS(2, 1)$ 和 $RS(3, 1)$ 共享了相同的数据布局, 所以当冗余转换发生时, 无需重分布数据块. 此外, $P'_0=P_0+D_2$, $P'_1=P_2+D_3=P_2+P_1+D_2$, 因此校验块重计算仅需读取老的校验块加上 1 个数据块 D_2, 即可完成.

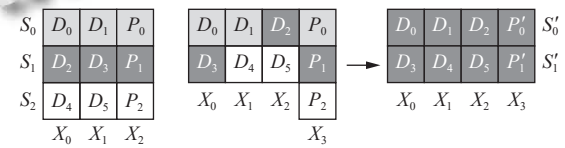


图 3 *ERS*(2, 1, 3)

1.4 已有方案缺点分析

1.4.1 里德-所罗门码条带合并的 I/O 开销巨大

算法的复杂度分析: *RS* 码在条带合并时, 因不同小条带的数据块可能聚集于相同节点上, 需先遍历记录了块所在节点 *id* 的向量, 得出哪些节点有聚集, 此操作时间复杂度为 O(k), 然后还需要将该节点上的其中一个条带的数据块迁移至其他节点, 导致条带合并需要进行大量的数据块重分布 I/O 操作. 此外, 由于编码矩阵发生改变 (从 $A_\{m\times k\}$变为 $A\_\{m\times xk\}$), 校验块需要进行重计算, 这需要读取所有小条带的数据块并计算产生新的校验块, 重新读取所有数据块产生大量 I/O, 计算校验块为时间复杂度 O($k\times m\times d\_length$) 的矩阵乘法过程 (d_length 是数据块大小), 十分消耗系统资源.

举例说明, 图 4 为 2 个 $RS(2, 1)$ 的小条带合并成一个 $RS(4, 1)$ 的大条带的过程. 由于第 1 个小条带的 D_1、P_0 和第 2 个小条带的 D_2、D_3 聚集在相同节点上, 为了保证大条带的容错能力, 需要将 D_2、D_3 迁移到另外两个节点上. 此外, 由于编码矩阵改变, 需要将 P_0 和 P_1 更新为P'_0, 一个直接的方式是读取 4 个原始数据块重计算出P'_0.

当进行多次合并时, 由于以上原因, 冗余转换的 I/O 开销就更加巨大. 图 5 展示了 $RS(4, 2)$、$RS(3, 2)$

© 中国科学院软件研究所 http://www.c-s-a.org.cn

和 *RS*(6, 2) 合并为一个 *RS*(13, 2) 的过程, 一共迁移了 9 个数据块, 读取了 13 个数据块用以计算校验块.

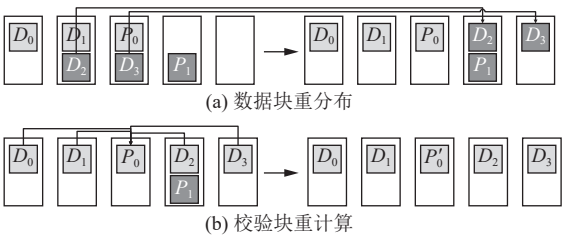


图 4 2*RS*(2, 1)→*RS*(4, 1)

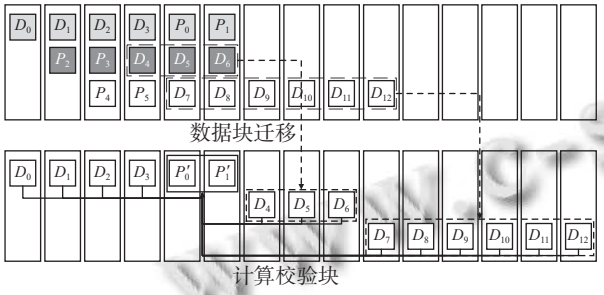


图 5 *RS*(4, 2)+*RS*(3, 2)+*RS*(6, 2) 经过 2 次合并生成 *RS*(13, 2)

1.4.2 弹性里德-所罗门码难以应对多次合并

ERS 码的确降低了条带合并中数据块重分布和校验块重计算的 I/O 开销, 但是它固定了转换前后的参数, 仅适用于单次合并, 在多次合并的后续合并中, 情况会恶化为传统 *RS* 码合并, 仍会引起巨大的 I/O 开销, 分析同第 2.1 节.

下面举例 3 个 *RS*(4, 2), *RS*(3, 2) 和 *RS*(6, 2) 的小条带, 其中 *RS*(4, 2) 和 *RS*(3, 2) 条带符合 *ERS* 的数据布局. 在第 1 次合并中 (即 *RS*(4, 2) 和 *RS*(3, 2) 合并成 *RS*(7, 2)), 如图 6 所示, 由于 *RS*(4, 2) 和 *RS*(3, 2) 的数据块分散放置, 因此无需迁移任何数据块, 且可以充分利用老的校验块来更新计算新的校验块.

然而在第 2 次合并中 (即 *RS*(7, 2) 和 *RS*(6, 2) 合并成 *RS*(13, 2)), 如图 7 所示, 此时两个条带发生了数据块聚集, 需要迁移 5 个数据块. 且为了更新计算新的校验块, 需要读取所有 13 个数据块, 产生大量 I/O.

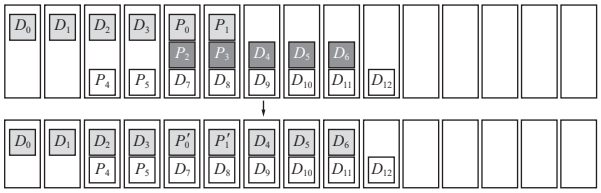


图 6 *ERS* 的第 1 次合并, *RS*(4, 2)+*RS*(3, 2)→*RS*(7, 2)

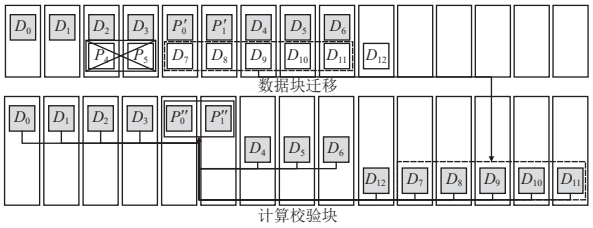


图 7 *ERS* 的第 2 次合并, *RS*(7, 2)+*RS*(6, 2)→*RS*(13, 2)

1.4.3 优化思路

分析上述合并过程可知, 数据块迁移开销主要由合并前条带的数据块 (或校验块) 重叠导致, 校验块更新开销主要由读取所有条带的数据块导致. 为了降低数据块迁移开销, 本文考虑提前设计合并前条带的放置方式, 使条带合并前后数据块位置一致. 为了降低校验块更新开销, 本文考虑对校验块的计算过程以及公式进行设计, 使之读取更少的块. 可以注意到条带在合并前已有校验块, 这些旧校验块是使用合并前条带的数据块计算得到, 如果计算新校验块时能够利用这些旧校验块, 则不必读出所有数据块. 此外还需要合理组织管理参与合并的 x 个小条带, 使得多次合并都能有效进行. 基于上述思路, 本文设计出新的编码方案, 于第 2 节介绍.

2 树型里德-所罗门码

2.1 树型里德-所罗门码概述

为了有效应对多个不同参数条带的多次合并, 本文提出了树型里德所罗门码 (*TRS*). 一个参数为 (x, y, k_1, k_2,⋯, k_x, m) 的 *TRS* 码将 x 个参数为 (k_1, m)、(k_2, m)、⋯、(k_x, m) 的小条带, 通过 $1 \leqslant y \leqslant x-1$ 次合并, 形成参数为 ($k_1+k_2+\cdots+k_x$, m) 的大条带. 例如, *TRS*(4, 2, 2, 2, 2, 2, 1) 将 4 个参数为 (2, 1) 的小条带, 通过 2 次合并, 依次形成 2 个 (4, 1) 的中间条带和 1 个 (8, 1) 的大条带.

TRS 码首先设计 x 个小条带的数据布局, 通过分散放置 x 个小条带的数据块, 使得任何一次合并都不需要数据块重分布操作.

其次, 设计 x 个小条带的编码矩阵, 通过利用 1 个大编码矩阵不同位置的子矩阵, 使任何一次合并仅需读取已有校验块即可重计算出新校验块.

为了有效组织和管理这 x 个小条带, 本文进一步设计了存储单元的逻辑结构. 存储单元将这 x 个小条带组织为一棵树, 条带合并从叶子节点开始到根结点结束, 按照从子节点到父节点的方式, 依次进行合并.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

2.2 数据布局设计

对于参与合并的 x 个小条带, 其布局规则如下.

规则 1. 将 $k_1+k_2+\cdots+k_m$ 个数据块分散放置于 $k_1+k_2+\cdots+k_m$ 个不同的节点.

规则 2. 将 $x\times m$ 个校验块, 聚集放置于 m 个不同的节点, 注意存放校验块的节点与存放数据块的节点不同.

例如, 存储参数为 $TRS(2, 1, 4, 3, 2)$, 数据块和校验块放置如图 8 所示. 当两条带合并为一个参数为 $RS(7, 2)$ 的条带时, 无需迁移任何数据块, 且使用后文将介绍的编码矩阵设计, 校验块更新可只访问存储校验块的节点本身.

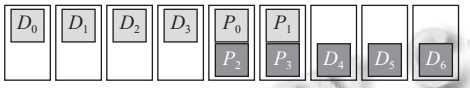


图 8 $TRS(2, 1, 4, 3, 2)$

该设计效果如下: ① 对于任何一次合并, 由于参与合并的小条带的数据块分散放置, 不会产生数据块碰撞, 无需数据块重分布. 例如 (k_1, m) 与 (k_2, m) 合并成 (k_1+k_2, m) 的过程中, k_1+k_2 个数据块是分散的, 因此无需数据块重分布的 I/O. ② 其次由于校验块聚集放置, 校验块重计算仅需访问存放校验块的节点即可. 例如 (k_1, m) 与 (k_2, m) 合并成 (k_1+k_2, m) 的过程中, $2m$ 个校验块聚集于 m 个节点, 校验块重计算仅需访问这 m 个存放校验块的节点即可.

2.3 编码设计

对于 $TRS(x, y, k_1, k_2, \cdots, k_x, m)$, 参与合并的 x 个参数分别为 (k_1, m)、(k_2, m) $\cdots$ (k_x, m) 的小条带, 设 A 为编码矩阵, 其编码规则如下.

规则 3. 所有小条带编码都采用矩阵运算式 $P=A\times D$, P 为校验块向量, A 为编码矩阵, D 为数据块向量.

规则 4. 编码矩阵 A 为 $m\times(k_1+k_2+\cdots+k_m)$ 的大矩阵, 参与合并的小条带均采用 A 作为编码矩阵.

规则 5. 所有小条带都使用有相同元素个数的数据块向量 D, 每个小条带的 D 中, 第 $k_0+k_1+\cdots+k_{i-1}$(设 $k_0=0$) 到第 $k_0+k_1+\cdots+k_i-1$ 个元素是该小条带的数据块, 其余为 0.

例如, 存储参数为 $TRS(2, 1, 4, 3, 2)$, 两个小条带的编码式如下所示:

$$\begin{bmatrix} P_0 \\ P_1 \end{bmatrix} = A_{2\times7}\times\begin{bmatrix} D_0 & D_1 & D_2 & D_3 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}} \quad (2)$$

$$\begin{bmatrix} P_2 \\ P_3 \end{bmatrix} = A_{2\times7}\times\begin{bmatrix} 0 & 0 & 0 & 0 & D_4 & D_5 & D_6 \end{bmatrix}^{\mathrm{T}} \quad (3)$$

而合并后条带的编码式如下:

$$\begin{bmatrix} P'_0 \\ P'_1 \end{bmatrix} = A_{2\times7}\times\begin{bmatrix} D_0 & D_1 & D_2 & D_3 & D_4 & D_5 & D_6 \end{bmatrix}^{\mathrm{T}} \quad (4)$$

因此, $P'_0=P_0+P_2$, $P'_1=P_1+P_3$. 结合之前的校验块放置设计, 校验块聚集存储在相同节点上, 因此校验块更新可在相应节点上完成, 如图 9 所示.

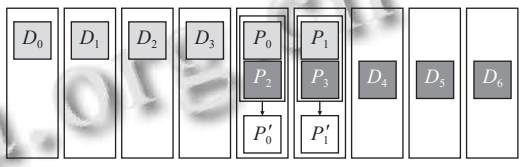


图 9 同一节点上的校验块更新

该设计的效果如下: 对于任意一次合并, 由于参与合并条带的大编码矩阵相同, 所以合并后条带的校验块可由对应节点的旧校验块相加得出.

2.4 存储单元

上文所述的数据布局和编码矩阵的联合设计, 可以大幅降低条带合并的 I/O 开销. 本节进一步设计存储单元, 以有效组织和管理参与合并的 x 个小条带.

2.4.1 存储单元概述

存储单元是多个条带的逻辑结构, 包含了元数据管理功能以及条带存取、条带编码、条带合并的算法. 参与合并的 x 个小条带记为一个存储单元, 它有固定的节点用于存储数据块和校验块. 每个存储单元的编码矩阵是相同的.

存储单元将参与合并的 x 个小条带组织成一棵树. 条带合并从叶子节点开始到根结点结束, 按照从子节点到父节点的方式, 依次进行合并操作.

存储单元能放置的数据块个数为系统的节点数减去该存储单元用于存放校验块的节点数, 存储单元可能存满也可能存不满. 注意此处的放置和存满都是逻辑意义上的, 属于元数据管理, 并不是条带在系统中的物理存储.

系统中可能有多个存储单元, 为了提高存储单元的利用率, 系统将存储单元按距离放满所需的数据块个数分为若干组, 每次存储新条带都按组分配存储单元, 每次存储结束都动态调整分组.

2.4.2 存入条带的块放置

每个存储单元包含了若干个条带, 为了降低其中

© 中国科学院软件研究所 http://www.c-s-a.org.cn

任意条带、任意次数合并的开销, 本文设计算法控制每个新存入的条带在存储单元内的放置, 使存储单元内任意两条带间符合前述的最优布局.

思路分析如下: 为了使两条带间符合最优布局, 存储单元内所有条带的数据块需存储于不同节点, 校验块需按顺序对应存储于相同节点, 因此存储单元需记录上一个存入条带最后存放数据块的位置, 作为下一个条带存入时数据块放置的起始位置, 此外固定放置校验块的起始位置也需要记录. 对于节点数确定的存储系统, 假设唯一的标识每个节点的参数是节点 id (id 从 0 开始小于总节点数), 则存储单元内需记录两个起始 id, 既数据块起始 id 和校验块起始 id, 数据块起始 id 每次存入新条带都要更新, 校验块起始 id 不变 (不同存储单元可变).

对于一个有 n 个节点的存储系统, 节点 id 从 0 到 n–1, 假设现在存入编码参数为 $RS(k, m)$ 的条带, 条带数据块向量为 $D[k]$, 校验块向量为 $P[m]$存入的存储单元为 Q, Q 的数据块起始节点 id 为 d_st, 校验块起始位置 id 为 p_st, 其过程如算法 1 所示.

算法 1. 存储单元块放置算法

1) 取出存储单元 Q 及其数据块起始节点 d_st, 校验块起始节点 p_st
2) for i←0 to k–1 do //遍历所有数据块
3) if $d\_st+i<p\_st$ //数据块位置是否与校验块位置重叠
4) 将 $D[i]$存入 id 为 d_st+i 的节点
5) else 将 $D[i]$存入 id 为 d_st+m+i 的节点
6) end if
7) end for
8) for i←0 to m–1 do //遍历所有校验块
9) 将 $P[i]$存入 id 为 p_st+i 的节点
10) end for
11) $d\_st+=k$ //更新数据块起始位置
12) if $d\_st+k\geq p\_st$
13) $d\_st+=m$
14) end if

举例如图 10 所示的一个有 15 个节点的存储系统, 其中存储了 4 个编码参数为 $RS(3, 2)$、$RS(4, 2)$、$RS(6, 2)$ 和 $RS(9, 2)$ 的条带, 一共组织为 2 个存储单元, 其中一个存储了 3 个条带已经存满, 另一个存储了 1 个条带并未存满, 剩余了 4 个逻辑意义上的节点 (并非物理上空余) 可放置数据块.

单元内的合并以其中第 1 个已经存满的存储单元为例, 如图 11.

该算法效果如下: ① 同一存储单元内任意条带、任意次数条带合并都无数据块迁移. 例如图 11 中 3 个条带最终合并为 1 个编码参数为 $RS(13, 2)$ 的条带, 过程无需任何数据块迁移. ② 由于校验块聚集放置, 任意条带、任意次数条带合并仅需访问校验块所在节点. 如图 11 中计算 3 个条带合并后的校验块, 仅需对应节点的校验块取出相加即可.

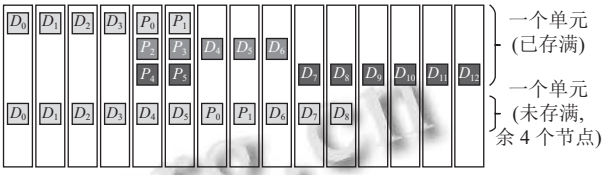


图 10 4 个条带组织为 2 个存储单元
(一个已存满, 一个未存满)

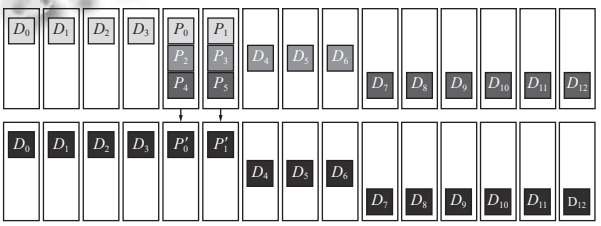


图 11 已存满存储单元合并 (3 个小条带进行两次合并, 最终合并为 1 个大条带)

2.4.3 存入条带的编码

存储单元内所有条带的大编码矩阵相同, 数据块按所处节点 (除去存放校验块的节点) 顺序填入数据块向量. 为计算存入条带的校验块, 本文设计了算法 2.

假设存储系统中有 n 个节点, 节点 id 从 0 到 n–1, 当前编码条带的参数为 $RS(k, m)$, 条带的第 i 个数据块 d_i 存储在 id 为 N_i 的节点上, 其中校验块所在节点的起始 id 特别的记为 N_m, 矩阵 $A_\{m\times(n-m)\}$为 $RS(n-m, m)$ 的编码矩阵, 简记为 A.

算法 2. 存储单元编码算法

1) 新建一个含 n–m 个块的数据块向量 D //用于保存计算中需要的含有数据块和 0 块的数据块向量
2) 初始化变量 X, X=0 //用于指向当前数据块应该填放在向量 D 中的位置
3) for i←0 to k–1 do //循环遍历每个数据块
4) if $N_i<N_m$ //如果该数据块的位置小于校验块起始位置
5) $X=F_i$ //该数据块在向量 D 中的位置与其所在节点 id 一致
6) else $X=F_i-m$ //如果数据块位置大于校验块起始位置, 则数据块在向量 D 中的位置为其所在节点 id 减校验块数量 m
7) end if
8) 将数据块 d_i 填入数据块向量 D 中的第 X 个块
9) end for //数据块遍历结束
10) for i←X+1 to $X+n-m-k$ do //从最后一次填入数据块的位置开始, 遍历向量 D 中没有填入数据块的位置

© 中国科学院软件研究所 http://www.c-s-a.org.cn

11) if $i>n-m$ //如果当前位置超过向量的维度
12) $i=i-(n-m)$ //循环至位置 0 开始
13) end if
14) 向数据块向量的第 i 个块中填入空白块
15) end for
16) 设定 A 为编码矩阵
17) 根据 $P=A\times D$ 计算得出校验块向量 P
18) 输出 P

该算法效果如下: 存储单元内任意条带、任意次数条带合并时都无需取出所有数据块重新编码, 只需取出新校验块对应节点上的旧校验块相加即可.

如图 12, 以编码参数为 $RS(6, 2)$ 的条带的编码过程为例, 因为系统中 3 个条带的 k 参数总和为 13, 所以采用编码参数为 $RS(13, 2)$ 的编码矩阵, 数据块向量中, 由于数据块存储于节点 id 为 9–14 的节点上, 所以将条带的数据块按顺序填入数据块向量的第 7–12 个块中, 最终计算得出校验块向量.

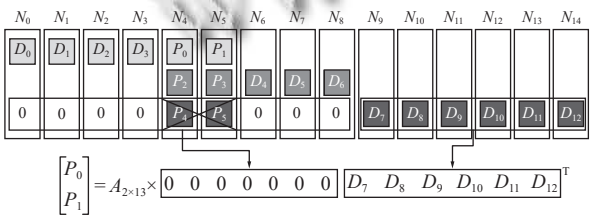


图 12 存储单元编码举例

该存储单元中的其余两个条带的校验块也都按此算法计算得出.

2.4.4 存储单元选择

当向系统写入一个条带时, 需要决定往哪个存储单元写入该条带. 具体思路如下.

首先基于存储单元差几个数据块 (记为 r) 才能放满将存储单元进行划分为若干组. 每写入一个条带, 都会根据相应存储单元的差值数量 r 调整其分组. 当新来一个条带 (k_i, m) 时, 选择 $k_i\leqslant r$ 且 r 值最小的存储单元, 写入该条带. 若一个存储单元已满, 则将其进行封装, 而后将其作为条带合并的候选对象.

具体算法如算法 3. 假设系统有 n 个节点, 存储单元一共 u 组, 用 $L[i]$表示第 i 组, 当前存入条带编码参数为 $RS(k, m)$, 唯一标识条带的是条带的 key.

算法 3. 存储单元选择算法

1) for $i\leftarrow k$ to u
2) if $L[i]$组存在未放满的存储单元
3) 选择该组中未放满的存储单元 Q
4) break
5) end if
6) end for
7) if 没有找到存在未放满存储单元的组
8) 新建存储单元 Q //新建时需指定数据块和校验块起始位置
9) end if
10) 调用算法 1 和算法 2 将条带存入存储单元 Q
11) 调整 Q 的分组编号为当前组减 k, 当前可合并条带数量加 1, 将该条带的 key 加入合并列表

注意每次存入参数为 $RS(k, m)$ 的新条带, 都优先存放于距离放满差 k 个数据块的存储单元, 所以差值小的存储单元总是能优先获得小参数的条带, 使得存储单元的内部碎片更小.

举例来说, 假设存储节点数 n=9, 其中校验块个数 m=1, 即每个单元内数据块个数为 8, 现在按先后顺序存储如下条带: (2, 1)、(2, 1)、(4, 1)、(3, 1)、(3, 1)、(5, 1)、(2, 1). 首先存储 (2, 1)、(2, 1)、(4, 1) 这 3 个条带在同一个单元内, 如图 13 所示.

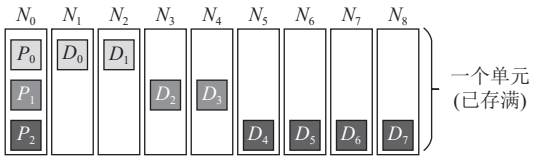


图 13 (2, 1)、(2, 1)、(4, 1) 条带存满一个单元

接下来存储参数为 (3, 1)、(3, 1)、(5, 1)、(2, 1) 的条带, 存储 (2, 1) 时, 因为 2 个 (3, 1) 条带所处单元在距离放满还差 2 个数据块的组中, 而 (5, 1) 在 3 个的组中, 所以优先放入前者, 如图 14.

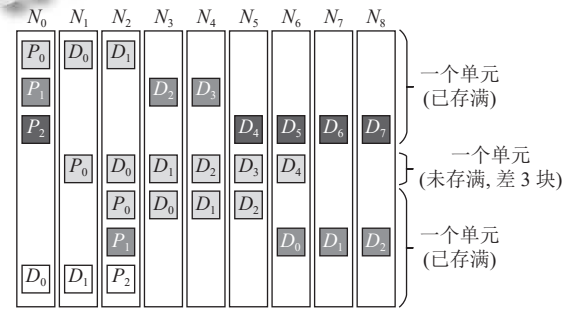


图 14 存入 (3, 1)、(3, 1)、(5, 1)、(2, 1) 条带

2.4.5 树型条带合并算法

每个存储单元内组织了若干条带, 每次合并其中两个条带, 直至所有条带合并为一个大条带, 这一过程为树状结构, 每个叶子节点为一个小条带, 中间节点为若干次合并后条带, 根节点为最终的大条带. 为了实现

© 中国科学院软件研究所 http://www.c-s-a.org.cn

灵活的条带合并, 每次合并如算法 4 所示.

算法 4. 树型条带合并算法

1) 新建校验块变量 P_{new}, 并初始化为 0
2) for i←0 to m–1 do //遍历所有存储校验块的节点
3) 从 id 为 p_st+i 的节点上读取条带 A 和 B 的校验块 P_A 和 P_B
4) $P_{new}=P_A+P_B$ //此处为有限域运算
5) 将 P_{new} 存储于 p_st+i 节点
6) end for
7) 存储单元 Q 的可合并条带数量减 1, 从合并列表中去除 B //保留 A 作为合并后条带的新 key

假设两条带 key 为 A 和 B, 参数为 (k_A, m) 和 (k_B, m), 所在存储单元为 Q, 校验块起始节点 id 为 p_st.

例如前例中, 存储单元中有编码参数为 *TRS*(3, 2, 3, 4, 6, 2) 的 3 个条带, 即合并列表中有 *RS*(3, 2)、*RS*(4, 2)、*RS*(6, 2)3 个条带的 key, 为了方便表示, 记 A, B, C 为 3 个条带的 key. 如图 15 所示, 第 1 次将 *RS*(3, 2)、*RS*(4, 2) 合并为 *RS*(7, 2), 合并后条带 key 为 A, 合并列表中删除 B, 剩余 A 和 C, 可合并条带数为 2; 第 2 次合并 *RS*(7, 2) 和 *RS*(6, 2) 为 *RS*(13, 2), 合并后条带 key 为 A, 合并列表中删除 C, 剩余 A, 可合并条带数为 1 (因为一次条带合并需要两个条带, 所以此时存储单元不可进行合并).

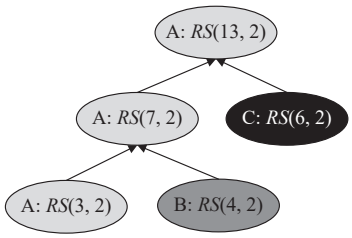


图 15 编码参数为 *TRS*(3, 2, 3, 4, 6, 2) 的 A、B、C 这 3 个条带, 合并过程被组织为树型

3 实验分析

本文通过测试展现 *TRS* 在条带合并中的性能优于 baseline/*RS* 和 *ERS*, 其中 baseline/*RS* 是将所有存入集群的条带随机放置, *ERS* 是将每两个条带作为一个单元, 单元内两条带所有数据块在不同节点校验块在相同节点, 单元间条带随机放置.

3.1 配置

本实验在本地集群上部署 memcached[30]分布式存储系统, 集群有 19 个节点, 每个节点使用 CentOS 版本 7.9.2009, 配备 2 个 12 核 2.20 GHz Intel(R) 至强 E5-2650 V4 处理器, 64 GB 内存以及一个 Seagate ST1000NM0023 7200RPM 1 TB 的 SATA 硬盘, 每个节点都有 10 Gb/s 的网络带宽.

3.2 方法

实验设置如下参数配置, 采用从 4 KB–4 MB 的数据大小, 编码参数从 (2, 1) 到 (6, 2), 合并的条带数 2–6 个. 实验使用不同的参数组合, 控制其中部分参数一致以测量每一次条带合并的时间 (该组的条带个数有几个就进行几次合并, 每次并入 1 个条带), 每组参数测量 10 次并将测量结果取平均值.

3.3 实验 1: 不同条带大小的条带合并用时分析

实验首先测试不同大小条带的冗余转换性能, 验证条带大小不同时, *TRS* 相较于 baseline/*RS* 和 *ERS* 是否有优化. 实验测试条带的数据大小从 4 KB 到 4 MB 的 3 个条带的合并用时, 其中 3 个条带的编码参数为 3 个 (2, 1) 和 3 个 (5, 2). 实验测量了 3 个条带的两次合并用时和总时长.

实验结果如图 16、图 17, 可以看出转换时间随着数据大小增大而增大, *TRS* 的性能明显比 baseline/*RS* 和 *ERS* 都更好. 编码参数为 (2, 1) 时, *TRS* 的总转换时间相比 baseline/*RS* 降低了 48%–74%, 其中第 1 次转换时间降低 40%–66%, 第 2 次降低 54%–79%; 相比较 *ERS* 总时间降低了 44%–69%, 第 1 次合并时间差不多 (都采用优化的放置方式), 第 2 次合并时间降低了 64%–83%. 编码参数为 (5, 2) 时, *TRS* 的总转换时间比 baseline/*RS* 降低了 55%–86%, 第 1 次降低了 50%–83%, 第 2 次降低了 49%–88%; 相比较 *ERS* 总时间降低了 53%–80%, 第 1 次仍然是十分接近, 第 2 次降低了 70%–88%. 可看出方案对于条带合并的优化比例较为可观, 且对数据较大的条带合并的优化比例明显大于数据较小的, 对编码参数更大的条带合并的优化比例明显大于编码参数更小的.

综上所述, 在条带大小不同时, *TRS* 相较于 baseline/*RS* 和 *ERS* 都有优化.

3.4 实验 2: 不同编码参数的条带合并用时分析

实验接下来测试不同编码参数条带的冗余转换性能, 验证条带编码参数不同时, *TRS* 相较于 baseline/*RS* 和 *ERS* 是否有优化. 实验测试条带的编码参数从 (2, 1) 到 (6, 2) 的 3 个条带的合并用时 (3 个条带编码参数不一定一致). 为了简便起见, 实验结果的柱状图中用 *TRS* 的编码参数表示 3 个条带的参数 (例如 *TRS*(3, 2, 2, 3, 4, 1) 代表参与合并的 3 个小条带参数为 (2, 1), (3, 1),

© 中国科学院软件研究所 http://www.c-s-a.org.cn

(4, 1)). 其中条带数据块大小为 64 KB 和 256 KB. 实验测量了 3 个条带合并的两次合并用时和总时长.

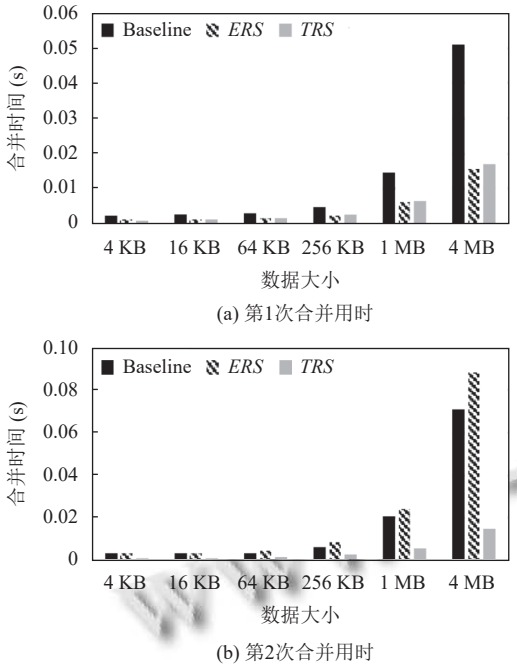


图 16 参数为 (2, 1) 的条带合并用时

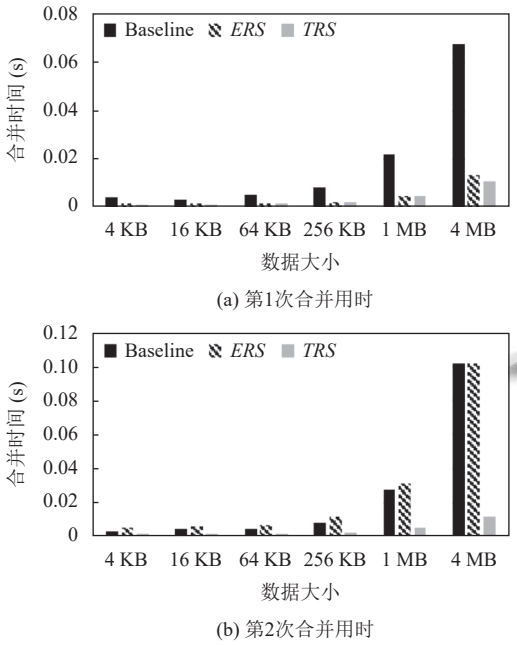


图 17 参数为 (5, 2) 的条带合并用时

实验结果如图 18、图 19, 可以看出对于所有编码参数, *TRS* 相较于 baseline 和 *ERS* 都有可观的性能提升. 数据块大小为 64 KB 时, *TRS* 的总转换时间相比 baseline 降低了 37%–75%, 其中第 1 次转换时间降低 29%–69%, 第 2 次降低 43%–80%; 相比较 *ERS* 总时间降低了 36%–60%, 第 1 次合并时间差不多 (都采用优化的放置方式), 第 2 次合并时间降低了 52%–75%. 数据块大小为 256 KB 时, *TRS* 的总转换时间比 baseline 降低了 56%–81%, 第 1 次降低了 47%–75%, 第 2 次降低了 62%–84%; 相比较 *ERS* 总时间降低了 53%–71%, 第 1 次仍然是十分接近, 第 2 次降低了 67%–83%.

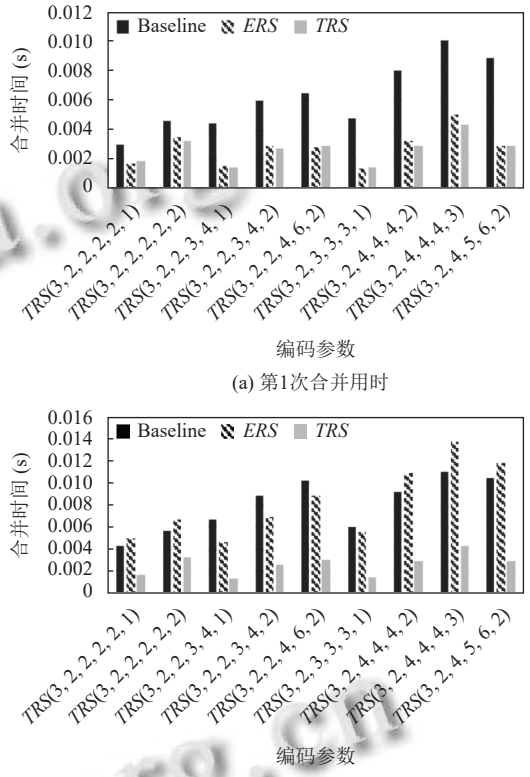


图 18 64 KB 数据块大小合并用时

综上所述, 在条带编码参数不同时, *TRS* 相较于 baseline/*RS* 和 *ERS* 都有优化.

3.5 实验 3: 不同编码参数不同条带个数合并用时分析

实验最后测试不同编码参数不同条带个数的冗余转换性能. 实验测试条带的编码参数从 (2, 2) 到 (5, 2) 的若干个不同条带的合并用时 (每组条带个数不同, 每组内每个条带的编码参数不一定相同), 其中条带的数据块大小为 128 KB. 实验测量了若干个条带的每次合并用时和总时长, 不同参数的合并次数不同.

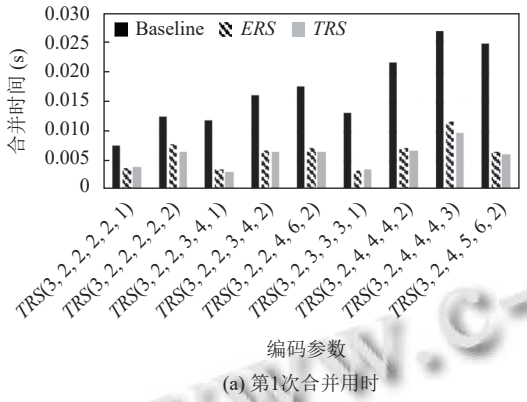
实验结果如表 1, 可以看出对于所有编码参数, *TRS* 相较于 baseline 和 *ERS* 都有可观的性能提升.

4 结论与展望

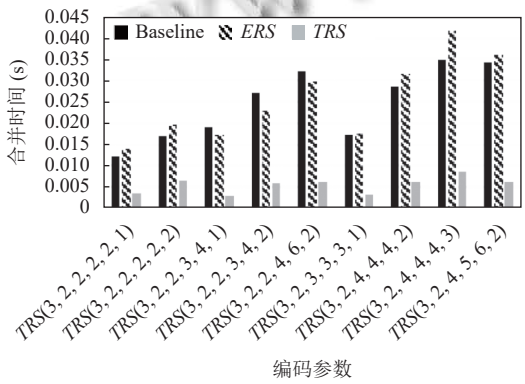
本文针对条带合并过程中引发大量数据块重分布

© 中国科学院软件研究所 http://www.c-s-a.org.cn

和校验块重计算 I/O 开销的问题, 设计了一种新的编码方案, 称为树型里德-所罗门码. 该编码方案联合设计了数据块放置方式和编码矩阵, 并设计了用于管理条带的结构, 即存储单元. 实验结果表明, 对于不同数据大小、不同编码参数、不同数量的条带, *TRS* 都对条带合并过程具有优化意义.



(a) 第1次合并用时



(b) 第2次合并用时

图 19 256 KB 数据块大小合并用时

表 1 各参数下 *TRS* 对 baseline/*RS* 与 *ERS* 优化比例 (%)

对照组	*TRS*(2, 1, 2, 3, 2)	*TRS*(3, 2, 2, 3, 4, 2)	*TRS*(4, 3, 2, 3, 4, 5, 2)	*TRS*(5, 4, 2, 2, 2, 3, 3, 2)	*TRS*(6, 5, 2, 2, 3, 3, 2, 4, 1)
baseline/*RS*	52	67	71	61	82
ERS	—	46	60	52	70

虽然新方案在性能上做到了优化, 但是方案本身要求条带在存入系统时要按设计的方式进行, 优化只对按该方案存放的数据有效, 而原本就存于系统中的数据的冗余转换仍然有性能问题, 未来将围绕这一问题展开研究.

参考文献

1 Huang C, Simitci H, Xu YK, *et al*. Erasure coding in Windows Azure storage. Proceedings of the 2012 USENIX Conference on Annual Technical Conference. Boston: USENIX Association, 2012. 15–26.

2 Ghemawat S, Gobioff H, Leung ST. The Google file system. Proceedings of the 19th ACM Symposium on Operating Systems Principles. Bolton: ACM, 2003. 29–43.

3 Rashmi KV, Shah NB, Gu DK, *et al*. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems. San Jose: USENIX Association, 2013. 8.

4 Yao QR, Hu YC, Cheng LF, *et al*. StripeMerge: Efficient wide-stripe generation for large-scale erasure-coded storage. Proceedings of the 41st IEEE International Conference on Distributed Computing Systems. Washington: IEEE, 2021. 483–493.

5 Ford D, Labelle F, Popovici FI, *et al*. Availability in globally distributed storage systems. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation Vancouver: USENIX Association, 2010. 61–74.

6 Chen HB, Zhang H, Dong MK, *et al*. Efficient and available in-memory KV-store with hybrid erasure coding and replication. Proceedings of the 14th USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2016. 167–180.

7 Yiu MMT, Chan HHW, Lee PPC. Erasure coding for small objects in in-memory KV storage. Proceedings of the 10th ACM International Systems and Storage Conference. Haifa: ACM, 2017. 14.

8 Weatherspoon H, Kubiatowicz JD. Erasure coding vs. replication: A quantitative comparison. Proceedings of the 1st International Workshop on Peer-to-peer Systems. Cambridge: Springer, 2002. 328–337. [doi: 10.1007/3-540-45748-8_31]

9 Chiniah A, Mungur A. On the adoption of erasure code for cloud storage by major distributed storage systems. EAI Endorsed Transactions on Cloud Systems, 2022, 7(21): e1. [doi: 10.4108/eai.14-9-2021.170955]

10 Qiao Y, Zhang MH, Zhou Y, *et al*. NetEC: Accelerating erasure coding reconstruction with in-network aggregation. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(10): 2571–2583. [doi: 10.1109/TPDS.2022.3145836]

11 Nachiappan R, Calheiros RN, Matawie KM, *et al*. Optimized proactive recovery in erasure-coded cloud storage systems. IEEE Access, 2023, 11: 38226–38239. [doi: 10.1109/ACCESS.2023.3267106]

© 中国科学院软件研究所 http://www.c-s-a.org.cn

12 Cheng LF, Hu YC, Lee PPC. Coupling decentralized key-value stores with erasure coding. Proceedings of the 2019 ACM Symposium on Cloud Computing. Santa Cruz: ACM, 2019. 377–389. [doi: 10.1145/3357223.3362713]

13 Ren YJ, Ren YM, Li XL, *et al*. ELECT: Enabling erasure coding tiering for LSM-tree-based storage. Proceedings of the 22nd USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2024. 18.

14 Hu YC, Zhang XY, Lee PPC, *et al*. NCScale: Toward optimal storage scaling via network coding. IEEE/ACM Transactions on Networking, 2022, 30(1): 271–284. [doi: 10.1109/TNET.2021.3106394]

15 Hu YC, Cheng LF, Yao QR, *et al*. Exploiting combined locality for wide-stripe erasure coding in distributed storage. Proceedings of the 19th USENIX Conference on File and Storage Technologies. USENIX Association, 2021. 233–248.

16 Plank JS. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software: Practice and Experience, 1997, 27(9): 995–1012. [doi: 10.1002/(SICI)1097-024X(199709)27:9<995::AID-SPE111>3.0.CO;2-6]

17 Plank JS, Ding Y. Note: Correction to the 1997 tutorial on Reed-Solomon coding. Software: Practice and Experience, 2005, 35(2): 189–194. [doi: 10.1002/spe.631]

18 Xia MY, Saxena M, Blaum M, *et al*. A tale of two erasure codes in HDFS. Proceedings of the 13th USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2015. 213–226.

19 Li J, Li BC. Demand-aware erasure coding for distributed storage systems. IEEE Transactions on Cloud Computing, 2021, 9(2): 532–545. [doi: 10.1109/TCC.2018.2885306]

20 Yang JC, Yue Y, Rashmi KV. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. ACM Transactions on Storage, 2021, 17(3): 17. [doi: 10.1145/3468521]

21 Taranov K, Alonso G, Hoefler T. Fast and strongly-consistent per-item resilience in key-value stores. Proceedings of the 13th EuroSys Conference. Porto: ACM, 2018. 39. [doi: 10.1145/3190508.3190536]

22 Chandrashekhara S, Kumar MR, Venkataramaiah M, *et al*. Cider: A case for block level variable redundancy on a distributed flash array. Proceedings of the 26th International Conference on Computer Communication and Networks. Vancouver: IEEE, 2017. 1–9. [doi: 10.1109/ICCCN.2017.8038466]

23 Wu S, Du QP, Lee PPC, *et al*. Optimal data placement for stripe merging in locally repairable codes. Proceedings of the 2022 IEEE Conference on Computer Communications. London: IEEE, 2022. 1669–1678. [doi: 10.1109/INFOCOM48880.2022.9796704]

24 Wu S, Shen ZR, Lee PPC, *et al*. Elastic reed-solomon codes for efficient redundancy transitioning in distributed key-value stores. IEEE/ACM Transactions on Networking, 2024, 32(1): 670–685. [doi: 10.1109/TNET.2023.3303865]

25 Xue HX, Wu CT, Li J, *et al*. Cauchy-merge: An efficient cauchy matrix based stripe merging method for Reed-Solomon codes. Proceedings of the 38th International Conference on Massive Storage Systems and Technology. Santa Clara, 2024.

26 Wu S, Lin GT, Lee PPC, *et al*. Optimal wide stripe generation in locally repairable codes via staged stripe merging. Proceedings of the 44th International Conference on Distributed Computing Systems. Jersey City: IEEE, 2024. 450–460. [doi: 10.1109/ICDCS60910.2024.00049]

27 Chen YP, Alspaugh S, Katz R. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. Proceedings of the 2012 VLDB Endowment, 2012, 5(12): 1802–1813. [doi: 10.14778/2367502.2367519]

28 Abad CL, Roberts N, Lu Y, *et al*. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. Proceedings of the 2012 IEEE International Symposium on Workload Characterization. La Jolla: IEEE, 2012. 100–109. [doi: 10.1109/IISWC.2012.6402909]

29 Berg B, Berger DS, McAllister S, *et al*. The CacheLib caching engine: Design and experiences at scale. Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation. USENIX Association, 2020. 753–768.

30 Memcached. https://memcached.org/blog. (2024-03-27) [2024-07-28].

(校对责编: 王欣欣)

© 中国科学院软件研究所 http://www.c-s-a.org.cn