分 类 号 _____ 学号 M201172389

学校代码 10487 密级 _____

华中科技大学

硕士学位论文

分布式文件系统中数据压缩策略研究

学位申请人: 付 宁

学 科 专 业: 计算机系统结构

指 导 教 师: 冯 丹 教授

答 辩 日 期: 2014 年 1 月 21 日

A Thesis Submitted in Partial Fulfillment of the Requirements
For the Degree of Master of Engineering

Research on
Data Compression in Distributed File System

Candidate : Fu Ning

Major : Computer Architecture

Supervisor : Prof. Feng Dan

Huazhong University of Science and Technology

Wuhan, Hubei 430074, P. R. China

Jan., 2014

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在_____年解密后适用本授权书。

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

随着互联网数据信息的爆炸式增长，以及现今云计算、云存储环境下大规模数据密集型应用的蓬勃发展，分布式文件系统以其高可靠性，高吞吐率以及海量的存储能力等优点，受到了学术界和工业界越来越多的重视，同时也面临着越来越多的挑战。数据压缩通过对数据进行重新编码和进一步组织，可起到缩减数据量的效果。将其运用到分布式文件系统中，可以通过增加部分计算量，显著减少存储空间、传输带宽消耗，提高数据传输、处理效率。

然而目前分布式文件系统中压缩策略的应用仍然存在许多问题，没有根据分布式文件系统的应用特性进行有效地优化定制。通过将各类数据压缩算法集成到分布式文件系统中，可以分析不同的非结构化大数据集的压缩率，以及各类压缩算法对分布式文件系统 IO 吞吐率的影响，并由此发现了以下四个问题：非结构化数据多样性需要系统进行压缩率预测；压缩和解压缩速率成为了整个文件读写流程的瓶颈；各压缩算法压缩与解压缩的不对称性会对整体吞吐率造成不同影响；全文件压缩极大地限制了压缩算法的应用场景和范围。

在对上述问题进行充分分析研究的基础上，根据分布式文件系统按块存储的应用特点，分块检测、分块压缩的方案被提出并实现。方案首先将待写入文件预先分块，分别对各分块进行压缩率预测及压缩，然后再写入到数据服务器上。通过分块检测，可以更好地应对各种待压缩数据，有效地发现不适合压缩的数据类型及数据块，避免不必要的压缩增加读写文件所花费时间，浪费 CPU 以及内存资源。；通过分块压缩，可以消除传统压缩策略带来的全文件压缩弊端，并且将数据压缩过程与传输过程部分重叠，充分利用客户端的计算资源，在微量影响压缩率的情况下，减轻数据压缩带来的时间开销，从而减轻压缩策略对分布式文件系统吞吐率的影响。

测试结果表明，分块检测策略可以很好地预测待压缩数据的压缩率，误差范围在10%以内；分块压缩策略将文件读写速率相较于传统的边压缩边传输方案提升了两倍以上。

关键词：数据压缩，分布式文件系统，分块检测，分块压缩

Abstract

With the explosive growth of Internet data, as well as the development of data-intensive large-scale systems applied in cloud computing environment, distributed file system has attracted more and more attention of experts for its high reliability, high throughput and massive storage capacity, while also facing more and more challenges. Data compression algorithm can achieve the effect of reducing the data capacity by recoding data, which can be applied to the distributed file system by increasing some calculation，in order to reduce storage , bandwidth consumption and improve data transmission or processing efficiency.

Through the application that data compression algorithms integrated into distributed file system, this thesis analyzes the compression rate of different unstructured data sets and the influence of distributed file system IO throughput caused by kinds of algorithms, thus find that compression and decompression rate become an bottleneck during file read and write process. Each compresses and decompresses asymmetry of algorithms is similar to the actual load of distributed file system, which can cause different effects on overall throughput.

On this basis, I proposed block detection and block compression scheme, according to the characteristics of block storage in distributed file system. First, the scheme pre-split the file block, and then does compression and compression rate prediction for each sub-block. Though the block detection, can better respond to data to be compressed and effectively found unsuitable type of data for compression to reduce the unnecessary compression. Though compressing block, can eliminate drawbacks of full file compression caused by traditional compression strategy, the data compression can be partially overlapped with the transfer process, and we can take full use of computing resources of the client, it can reduce the time caused by the data compression in the case of micro-impact on compression ratio, thereby reducing the impact of compression policies on system throughput.

Test result shows that sub-block detection strategy can be a good predictor to the compression rate of data to be compressed, and the error range is within 10%. The introduction of the scheme improves the read and writes rates more than twice compared to the conventional compressed and transmission scheme.

Key words: Data Compression, Distributed File System, Block Detection, Block Compression

目 录

1 绪论

1.1 课题背景

随着信息化时代的来临和当今科学技术的高速发展，人类社会所产生的数据信息呈爆炸式增长。特别是在天文研究、气象预报、社交网络、商业计算、数据挖掘等领域，数据信息的产生无论在量上还是在增长速率上都是非常惊人的，数据中心存储能力以及需求都迅速向 PB 级乃至 EB 级递增。统计数据表明，随着存储容量的不断增长，存储管理所花费的成本也随之逐日递增，甚至高出硬件成本的若干倍[1, 2]。在大数据时代，如何存储和管理海量的数据信息，成为国内外科研人员的重要关注点。分布式文件系统以其高可靠性，高吞吐率以及海量的存储能力等优点，受到了学术界和工业界越来越多的重视，并成为大数据时代的发展基石。然而分布式文件系统在如何科学有效地管理海量数据，满足上层应用高吞吐率需求等方面面临越来越多的问题，这些问题是否能解决关系到分布式文件系统是否能健康成熟地发展，关系到分布式文件系统是否能胜任大数据时代发展基石的角色。

1.1.1 数据激增问题

根据 IDC 公司（International Data Corporation）的统计报告[3]，2011 年全球创建及复制的数据量达到 1.8ZB（10 的 21 次方）。这些数据中高达 85%的数据为半结构化数据或非结构化数据，通常需要采用分布式文件系统或者非关系型数据库进行存储管理。以国内电商巨头淘宝公司为例，淘宝目前的数据总量已超过 40PB，目前仍以每月 3PB 的速度增长。而这些数据，大部分都是存储在分布式文件系统上的，图 1 为阿里巴巴内部 Hadoop 集群规模增长趋势。该集群中运用了大约 36000 块磁盘，存储的数据总量达到 60PB[4]。而在国外，截止到 2012 年 8 月，国外社交网络巨头 Facebook 每天数据处理量已超过 500TB，在其最大的分布式文件系统集群中，硬盘空间超过了 100PB。此外基因组学、天体物理学等以数据为中心的科学研究领域产生的数据量也越来越庞大，以脑科学为例，对人脑中的突触网络采用电子显微镜进行重建，每 $1mm^3$ 人脑内部图像生成的信息数据量超过 1000TB。

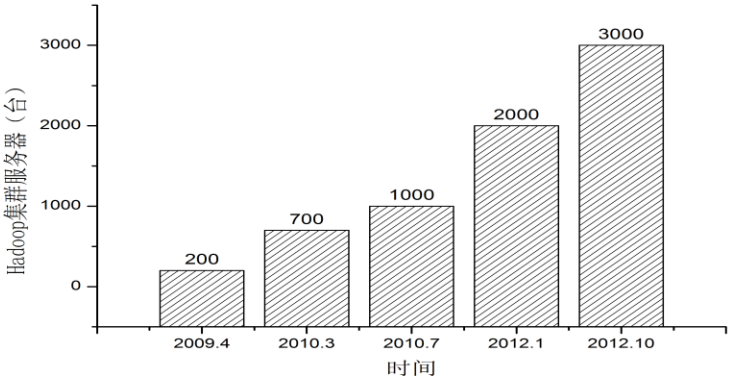


图 1.1 阿里巴巴 Hadoop 集群规模

由这些数据可以看到，当今大型互联网公司内部的分布式文件系统集群磁盘容量在不断扩大，数据规模更是急剧扩张，集群规模也在不断增长。然而尽管磁盘以及网络带宽价格逐年下降，但其下降速度远远不能抵消数据增长所带来的影响[5]，此外统计数据表明，至 2007 年左右开始，全世界所供应的存储介质总量和信息数据总量两者的增长曲线开始出现明显分离，目前，信息数据总量已经大大超过了存储介质总量。

就当前的发展情况来看，网络带宽的增长的速度也是极其迅速的，然而其增长速度仍然远远落后与信息数据的增长速度。目前在大型数据中心内部，千兆以太网、万兆以太网的应用越来越广泛，但即使是这样的发展速度仍然不能满足数据的传输需求。以万兆以太网为例，传输 10TB 数据理想情况下也需要超过 8000 秒，这远远不能满足上层高吞吐需求类型应用的需求。并且在数据密集型应用中，往往需要持续地对大规模数据进行操作，这进一步加剧了数据中心内部网络带宽的占用。

1.1.2 数据密集型应用激增问题

在云计算环境下，以 web 索引和搜索，商业数据信息提取，科学数据处理等为代表的数据密集型应用也越来越广泛，并成为现今研究的热点[6, 7]，同时也为海量数据的传输和处理提出了更高的要求。2007 年，图灵奖得主 Jim Gray 在其报告中指出，当今时代已经是数据密集型科学研究时代，数据密集型科学研究是“第四范式”[8]。存储系统的高性能、高吞吐率需求在天体物理学、医学影像等科学研究领域日益增加。例如美国国安局的信息检索系统，要求一万亿个文件在 10 分钟内检索完成，每秒生成 32,000 个文件，对后台存储传输带来巨大压力。为适应上述需求，分布式文

件系统需要提供与数据规模相适应的 I/O 处理速率和用户响应时间[9]。现有的分布式文件系统通过与 MapReduce 框架结合，非常适合非结构化数据处理和并行处理，已成为大数据分析的主流技术[9]。然而现今的发展情况下，高性能计算并行计算、以及 CPU 计算能力的增长速度远远超过了磁盘读写以及网络传输速度，现在的系统集群内部计算能力与 I/O 速度已经越来越不匹配。

数据激增带来的存储空间消耗、网络带宽占用问题，数据密集型应用激增带来的高 IO 处理速率需求、计算能力与读写传输能力的不匹配问题[10]，使得数据压缩技术在分布式文件系统中的必要性日益显现。数据压缩技术是一种通用的信息处理技术，它通过对数据进行重新编码组织，可以达到缩减数据量的效果[11, 12]。将其运用到分布式文件系统中，可以通过增加部分计算开销来显著减少存储空间、传输带宽消耗，提高数据传输、处理效率，从而极大地缓解上述几个问题。然而不同的数据压缩算法其压缩率往往会受到不同数据集的影响，并且在进行压缩和解压缩的时候，通常需要消耗一定的时间和 CPU 资源，因此需要对数据集进行预检测，并在压缩率以及压缩时间之间找到平衡点。此外，压缩算法的压缩速率与解压缩速率往往不对称，压缩速率往往远远小于解压缩速率，将压缩技术应用到分布式文件系统中后，其对分布式文件系统读写带来的影响需要进一步的研究分析。此外，分布式文件系统上层的某些典型的数据密集型应用，包括各类涉及到 Map-Reduce 的应用，常常需要并发地访问存储在分布式文件系统中的块信息，因此并不是所有的数据压缩算法都适用于这些应用。本文将在此基础上分析研究数据压缩策略在现今分布式文件系统中的应用情况，并提出优化地将压缩策略应用于分布式文件系统的分块检测、分块压缩方案。

1.2 国内外研究现状

1.2.1 分布式文件系统研究概况

随着数据信息的不断膨胀，以及非结构化数据在各大领域中的大范围应用，分布式文件系统应运而生。分布式文件系统，往往架设在成百上千台机器集群之上，通过集中管理，可以利用海量的廉价存储资源来存储数据，从而满足海量数据存储管理需求。分布式文件系统最重要的特性包括高可靠性、高可扩展性以及高吞吐率。高可扩展性是指随着机器数的增多，分布式文件系统能线性地增加其存储管理空间；

高吞吐率是指分布式文件系统应该能为上层业务提高较高的数据读写速度，应对较高的并发压力。目前大部分分布式文件系统往往分割处理元数据与数据，分离控制流与数据流，借此提高分布式文件系统的 IO 并发性。根据该种方案，分布式文件系统通常由三方架构组成，包括元数据服务器（MDS）、数据存储服务器和客户端三个部分。在目前分布式文件系统典型的三方架构中，通常采用单个元数据服务器，如著名的 GFS（Google File System）[13]，HDFS（Hadoop Distributed File System）[14]，Luster[15]等。以 HDFS 为例，HDFS 由一个元数据服务器 Namenode 和大量的数据存储服务器 Datanode 构成，允许海量的客户端（Client）访问，其系统架构图如图 1.2 所示。

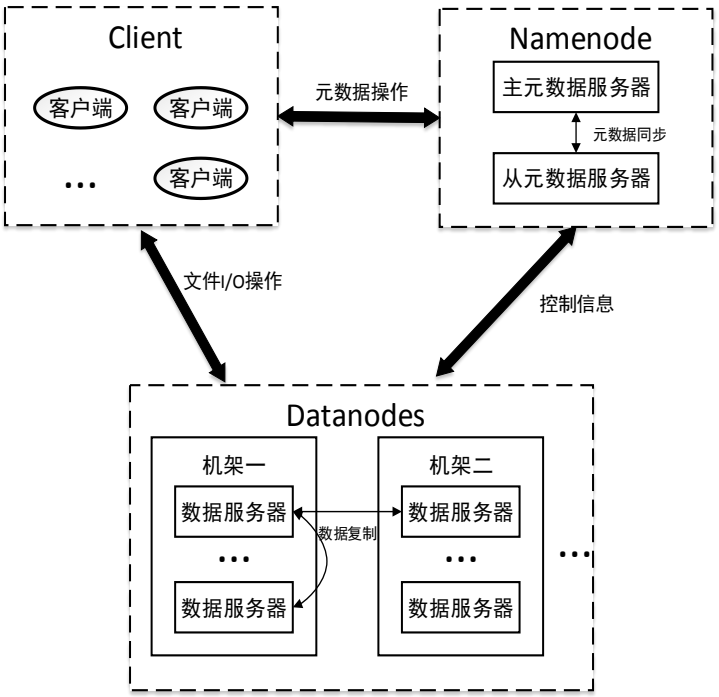


图 1.2 HDFS 系统架构图

在 GFS、HDFS 等典型的分布式文件中，通常存储的均是大文件，文件大小超过若干 GB。系统中文件以数据块的形式组织起来，每个数据块 64MB（可配置）。为了保证数据的可靠性，数据块被保存为三个备份（可配置），存放在成百上千的 Datanode 数据服务器节点中。采用数据块存储数据的好处在于简化管理，同时也使得文件更加便于读取和备份，若需要读取文件中某部分数据，只需要读取相应数据块即可；若需要备份文件，只需要对每个块单独进行复制即可。元数据服务器 Namenode 保证了系统中数据块的副本数保持一定，当某 Datanode 上的数据块副本

失效时，Namenode 会自动从其他 Datanode 复制该副本到该 Datanode 上。

分布式文件系统的研究始终聚焦在其性能、扩展性和可靠性等要点。近年来分布式文件系统为应对大数据时代所面临的问题越来越多，研究热点包括存储效率优化、元数据管理、吞吐率提升等方面[16-18]。其中纠删码技术[19]主要用于解决存储系统需要重点关注的可靠性问题。其原理是将 K 个磁盘的数据存储到 N 个磁盘中（N>K），然后通过采用某种纠删码对一个条带的数据进行编码，生成相应的校验信息。整个操作完成后系统能够容忍最多 N-K 个磁盘发生节点故障。这种技术的一个典型代表就是采用了 1 个冗余磁盘的 RAID 5[20]。然而纠删码技术在应用上依然存在不少问题，包括如何解决负载均衡，如何解决降级读问题[21, 22]。文献[23] 讲述了通过采用纠删码来降低数据丢失时的最小修复带宽。纠删码技术通过对文件编码生成校验来达到容错目的，能够极大地提升分布式文件系统的存储效率，近年来其在分布式文件系统中的应用一直是研究的热点。

1.2.2 数据压缩算法研究概况

数据压缩技术是在通信技术、计算机技术等的大规模发展应用的基础上成立并壮大起来的[24]，迄今已有六十多年历史。1948 年贝尔实验室的香农博士和 1949 年麻省理工学院的范诺博士提出的对符号进行编码的香农-范诺编码被认为是第一个有效的数据压缩编码。而在此三年后哈夫曼发表的关于《A Method for the Construction of Minimum Redundancy Codes》[25]的论文成为了数据压缩算法发展过程中的重要里程碑，数据压缩技术自此开始在各学科中大范围采用。数据压缩技术的主要原理是通过过消耗少量的计算时间对数据进行重新编码组织，采用最少的数码来表示信号，从而减少数据中间存在的冗余，达到缩减数据量的效果[26]。

冗余度即数据的重复度，信息数据能被压缩算法压缩是因为其原始信源的数据存在着很大的冗余。冗余的类型包括统计冗余、结构冗余等。数据压缩的实质就是减少信息中本来存在的冗余度。香农博士通过借鉴热力学中“熵”的概念，用“信息熵”概念称呼信息中消除了冗余后的平均信息量[27]。统计数据表明，目前各种信息系统中所存储的数据存在约 60%的重复数据，邮件网页 Web 等也存在大量的重复数据[28, 29]。通过数据压缩技术，可以大量地削减这些冗余数据，从而节省存储空间消耗。数据压缩技术如今可划分为两大类：一种是无损数据压缩技术[30]，另一种是有损数据压缩技术[31]。其分类如图 1.3 所示。

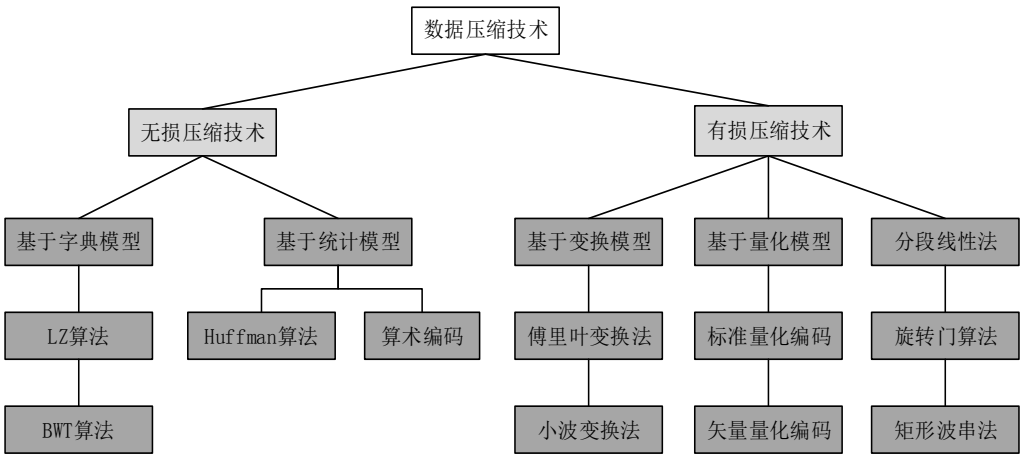


图 1.3 数据压缩技术分类图

其中无损数据压缩技术主要是用于消除统计冗余，其理论基础是采用尽量少的数据位数来表示原始数据。无损数据压缩技术主要可分为基于概率统计模型和基于字典模型。在上述两种模型中，概率统计模型内部又可进一步地分为静态统计模型和自适应模型。静态统计模型通过预扫描文件，以统计字符出现概率，这部分包括著名的 HUFFMAN 编码[25]；自适应模型假定所有字符出现概率相等，随着字符不断输入和编码，统计并记录字符出现的概率，并将此概率应用于后续字符，算术编码通常采用自适应模型实现。基于字典模型的编码是由 Jacab Ziv 和 Abraham Lempel 这两位杰出的以色列科学家在 1977 年提出的[32]。字典模型并对字符出现的概率进行直接考察统计，而是将已经编码过的数据作为标识保存，然后利用查字典的方式对字符串进行编码，这种模型下的编码包括著名的 LZ77[32]，LZ78[33]，LZW[34]等。

有损压缩技术往往采用某些较高的有限失真压缩算法，对冗余信息进行大幅压缩，从而获得远高于无损压缩的压缩率。经有损压缩后产生的数据总是不能恢复成原始数据。经过多年来的发展，有损数据压缩领域已经取得了非常引人注目的成就，目前所见到的绝大部分多媒体数据，均采用了有损压缩技术，通过有损数据压缩，视频文件能够在损失部分质量的情况下，实现超过 300:1 的压缩比。相较于无损压缩，有损压缩主要存在信号丢失、受信号源影响、转换不方便等不足，目前有损压缩算法通常应用于影音、图像等多媒体数据的压缩[35,36]。

广义上来说，重复数据删除技术也算是数据压缩技术的一种[37]。它在保证数据精确性和完整性的基础上，删除全局数据集中重复的数据，只对其中一个备份进行保留，以此消除冗余数据。重复数据删除的检测方法可分为整文件检测与分块检测。

其中整文件检测技术以整个文件为粒度，处理速度快，但消除粒度粗；分块检测技术通过将文件切割为若干个互不交叠的块，通过哈希计算来判断块内容是否重复，从而消除重复的数据块。重复数据删除技术目前主要在数据备份系统和归档系统中大量应用。

数据压缩技术最早主要是用于信号处理领域。目前数据压缩技术在通讯领域的应用以及新算法的研究依然是研究热点[38-40]。近年来随着信息数据的爆发性增长，数据压缩技术在数据密集型应用，以及存储领域特别是云存储领域的应用和研究变得越来越广泛[41, 42]，现在的研究热点还包括数据压缩技术在数据库，特别是实时数据库中的应用[43]。

1.2.3 数据压缩在分布式文件系统中的研究概况

近年来随着信息时代的发展，数据信息的迅猛增长已经愈发超出了人类的控制。数据压缩技术在存储领域特别是云存储领域的应用和研究变得越来越广泛。在分布式文件系统这一越来越重要的领域，数据压缩技术的应用和研究也越来越广泛。数据压缩技术为分布式文件系统带来的好处主要有三个方面：即节省存储空间消耗、减少系统 IO 及网络 IO 消耗。

数据压缩带来的这三方面的好处，极大地缓解了当前分布式文件系统应对大数据需求的压力。在工程领域，越来越多的分布式文件系统开始在系统内部实现各种无损数据压缩算法，其中最突出的是 HDFS 分布式文件系统。在 HDFS 的最新版本中，已经允许通过配置实现 zlib、bzip2 以及 Snappy 等常用无损数据压缩算法，并且允许在 MapReduce 的各个阶段选择压缩策略。例如对于永久保存的文件可以采用 bzip2 算法进行压缩，因为其压缩率相对较高；而对于 Map 的结果可以通过 LZO 或 Snappy 算法等压缩后再进行写和传输，从而带来性能的提升。然而包括 HDFS 在内的大部分分布式文件系统均并没有对压缩算法在系统中的应用进行分析探讨，也没有对压缩算法使用场景和策略进行优化，仅仅是实现了多种压缩算法供使用者选择使用。

由上述调查分析结果可以得出，目前分布式文件系统中所存储数据多样，对不同压缩算法的压缩率影响较大；压缩解压缩性能对文件系统读写速率存在影响；并行计算任务与压缩算法之间还存在诸如数据是否可分拆压缩等问题。目前分布式文件系统中数据压缩的应用和研究主要集中在如何集成有效的适合应用场景的算法，以及如何通过数据压缩策略减轻上层 MapReduce 应用高负载压力上[44, 45]。FAST2013

的文献[46]介绍了如何通过两层过滤，高效精准地判断数据在压缩时将带来多大的空间节省从而节省 CPU 开销。

数据压缩技术中的有损压缩技术并不适用于分布式文件系统，原因在于分布式文件系统中所存储的往往是各种非结构化数据，主要包括科学数据、生产日志、交易记录、URL 信息等，这些数据对于上层应用来说是比较重要的，需要保证数据安全，也不允许其质量下降。而分布式文件系统中存储的数据的音频视频文件也几乎都是压缩过的，不需要二次压缩。

重复数据删除技术在主存储系统中的应用尚在探索发展中，基于分块检测的重复数据删除技术在分布式文件系统应用还并不够广泛，其原因主要在于重复数据删除技术对主存储系统性能以及数据一致性的影响。

1.3 主要工作和论文组织结构

本文充分研究了现有分布式文件系统中数据压缩的相关技术，将现阶段国内外研究现状作为背景，以实验室自主研发的 RaccoonFS 分布式文件系统作为实验平台，详细分析测试了各类压缩算法对分布式文件系统 IO 吞吐率的影响，以及各压缩算法的适用场景。并在此基础上，针对当前分布式文件系统中应用数据压缩策略所存在的一些问题，提出并设计实现了分块检测、分块压缩的方案。通过分块检测，可以自适应地发现不适合压缩的数据类型，减少不必要的压缩开销。通过分块压缩，可以将数据压缩过程与传输过程部分重叠，在不影响压缩率的情况下，减轻数据压缩带来的时间开销。本文共有五章：

第一章介绍了分布式文件系统研究概况，数据压缩算法研究概况，以及分布式文件系统中数据压缩算法应用的国内外研究现状，分析了现有分布式文件系统中研究热点以及存在的问题，由此得出研究分布式文件系统中数据压缩策略的意义。

第二章介绍了数据压缩在分布式文件系统中应用存在的问题，这些问题包括数据多样性问题、压缩算法性能影响问题、压缩解压缩不对称问题、全文件压缩问题等。接着分析了重复数据删除技术的原理及其在主存储系统中的应用情况，然后对 zlib、bzip2、Snappy 三种无损压缩算法的原理和性能进行了简要分析，最后提出分块检测、分块压缩的优化方案。

第三章介绍了在 RaccoonFS 分布式文件系统上应用数据压缩算法，采用分块检测、分块压缩的设计方案和具体实现，具体讨论了方案架构，模块设计和实现等。

第四章介绍了对 RaccoonFS 分布式文件系统中是否进行数据压缩的对比测试，以及本文方案的性能测试，包括测试目的、测试环境、测试过程，对比了不同方案下系统的性能，并对测试结果进行了详细分析。

第五章对全文进行总结，分析并提出了可以进一步完善的地方。

2 分布式文件系统中数据压缩存在的问题分析

随着数据信息的爆炸式增长，以及云计算、云存储环境下大规模数据密集型应用的蓬勃发展，分布式文件系统在管理海量数据，满足上层应用高吞吐率需求等方面面临越来越多的问题。数据压缩技术通过增加部分计算开销来显著减少存储空间、传输带宽消耗，提高数据传输、处理效率，从而极大地缓解上层应用带来的压力。为了保证数据的质量和安全，有损数据压缩技术在分布式文件系统中几乎没有得到采用，目前分布式文件系统中主要采用的是无损数据压缩技术。然而目前将无损数据压缩应用到分布式文件系统中的策略尚存在许多问题，这些问题包括数据多样性问题、压缩算法性能问题、压缩算法压缩解压缩不对称问题、全文件压缩问题等。本章将首先对这些问题进行分别研究，并讨论现有的解决方案，然后讨论数据压缩在分布式文件系统中应用存在的问题及解决方法，提出基于现有方案的优化方法。

2.1 无损数据压缩存在的问题

在将分块检测、分块压缩方案应用到分布式文件系统之前，首先需要分析当前典型分布式文件系统的管理及应用特性，以及当前分布式文件系统中数据压缩技术应用所存在的问题和影响。将无损数据压缩技术应用到分布式文件系统能够带来很多好处，但是现有的包括 HDFS 在内的工程实现都没有考虑无损数据压缩技术与分布式文件系统结合所存在的问题，没有对在分布式文件系统中应用无损数据压缩策略进行优化定制。下面将对上述问题一一进行分析。

2.1.1 数据多样

分布式文件系统所存储的数据往往是异构的，半结构化或者非结构化的。非结构化数据是相对于结构化数据而言，即非结构化数据往往不方便采用数据库中的二维逻辑表来表示。非结构化数据的格式包括文本、办公文档、图像视频信息等。统计结果表明，现今人类所产生的数据信息中超过 85%的数据为非结构化数据[2,3]。分布式文件系统中所存储的数据类型包括天文研究、气象预报、社交网络、商业计算、数据挖掘等诸多领域，且多为数 GB 的大文件。相比于结构化数据，海量的非结构化数据必然增加了数据管理的复杂性。将数据压缩算法应用于分布式文件系统中需要综合考虑到分布式文件系统所存储的数据类型的多样性，以及数据压缩算法本身

对于数据类型的敏感性。非结构化数据由于其数据类型不同，往往存在差别较大的压缩率，不能统一对待。对于文本、网页、日志等数据类型，采用数据压缩能够获得较高的压缩率，减少存储空间和带宽消耗；而对于图片、视频等数据类型，压缩算法并不能对这种文件进行进一步压缩，反而会增加读写文件所花费的时间，浪费 CPU 以及内存资源。而大文件内部的数据不均匀性也使得对待压缩数据进行压缩率预测成为必要。

2.1.2 压缩算法性能

不同的压缩算法其压缩率以及压缩和解压缩速度往往是不同的，甚至可能因为设计目标的不同而相去甚远，即使在同一压缩算法中，部分压缩算法也支持通过参数配置而在最高压缩率以及最快压缩速率间进行调整[48]。在应用压缩算法对分布式文件系统进行压缩存储时，需要考虑各压缩算法所需要的压缩时间以及解压缩时间，从而使得选取的压缩算法满足当前的性能要求。目前在典型的分布式文件系统中，如 HDFS 分布式文件系统，所采用的压缩处理流程通常如图所示。客户端首先将源文件按顺序以数据流形式读入客户端内存，然后对读入部分进行压缩，然后将压缩后数据写入分布式系统中的数据服务器端。若写入的数据长度满足一个数据块 block 所需长度（一般为 64MB）后，系统自动将数据写入下一个数据块 block。如图 2.1 所示。

图 2.1 引入压缩策略后分布式文件系统写入文件示意图

虽然数据压缩算法减少了网络以及存储空间消耗，但对于文件的写入流程，整个写入时间的瓶颈受到了压缩速率的限制。然而对于通常的追求高压缩率的算法来说（包括 zlib、bzip2 等），其压缩速率往往比较慢，通常在几兆到几十兆之间，显然应用这类压缩算法，将大大影响客户端的写入速度。此外对于文件的读过程，整个读取时间也受到了解压缩速率的限制。但相对来说，读文件过程受到的影响较小，原因在于，绝大多数压缩算法的解压缩速率都是比较快的。将压缩算法应用到分布式文件系统中，应考虑到如何在以压缩算法理论性能特征为固有指标情况下，如何采取合适策略运用实践这些算法，以获得性能改善。文献[49]对比了各压缩算法的压缩

速率，可以看出，同一压缩算法对不同的数据格式，压缩速率相差并不大，而不同的压缩算法之间压缩速率相差是比较大的，这中间存在的普遍规律是压缩速率往往需要牺牲压缩速率来获得。

2.1.3 压缩解压缩不对称

对于绝大多数无损压缩算法而言，其压缩速率与解压缩速率是不对称的，解压缩速率通常在压缩速率的三倍以上，甚至达到数十倍的差距[49]。这样的算法很多，包括常见的 zlib、bzip2、lz4、lzma、Snappy 算法等等，实际上你很难找到哪个压缩算法是解压缩速率慢于压缩速率的。这种不对称性恰好与目前主流的分布式文件系统的设计目标是相符合的，而这种不对称性也与实际应用负载情况是相匹配的，即分布式文件系统总是读多写少，读写负载比例不对称，负载比例在 8:2 左右[50]。因此，将数据压缩技术应用到分布式文件系统中，需分析测试上述情况，避免不恰当的应用影响了系统的性能。表 2 所示为 gzip、bzip2、lzma 三种压缩算法的解压缩速率与压缩速率的比值。

表 2.1 解压缩速率与压缩速率的比值表

压缩文件类别	gzip 算法	bzip2 算法	lzma 算法
日志文件	7.5	4.4	122
二进制文件	7.6	4.1	100
URL 文件	7.2	4.6	165

2.1.4 全文件压缩

在以 GFS 为代表的典型分布式文件系统中，通常将大文件分割为 64M 大小的数据块 block 进行存储和管理。数据块大小设定为 64MB 带来的主要好处包括减少寻址开销、提高并行处理度等。数据块大小设置过小，则会增加元数据服务器的内存开销，增加磁盘寻道时间；数据块大小设置过大，则导致了问题分解过大，也不利于 MapReduce 并发处理任务。在典型的分布式文件系统中，传统方案是对于整个文件进行压缩存储，将压缩后的数据进行分块。这种方案还需要考虑压缩算法的 split 特性，所谓 split 特性是指一个压缩算法是否允许对已压缩文件中某部分数据单独进行解压缩。目前绝大部分压缩算法是不支持 split 特性的。比如 DEFLATE 算法[51]，它将数据存储为连续的压缩块，不允许从数据流中的任意一点知道下一个压缩块的起始位置，压缩块的开始位置并没有任何特征。因此全文件压缩存在以下两个问题：

（1） 破坏了数据块 64MB 大小的设定，使得数据块所存储数据远大于 64MB，

不利于对数据块中的部分进行单独处理，也不利于 MapReduce 操作的并发进行。

（2） 大部分的数据压缩算法往往不支持 split 特性。这导致对于存储在分布式文件系统中的压缩文件，并不能通过 MapReduce 任务对其中的某个数据块 block 单独进行操作，用户必须读出整个文件进行解压缩，这显然大大影响了压缩策略的适用范围。

下表 2.2 显示了各压缩算法是否支持 split 特性。

表 2.2 各压缩算法与 split 关系表

压缩算法	是否支持 split
zlib	否
gzip	否
bzip2	是
LZO	否（可通过标记后支持）
Snappy	否

2.2 重复数据删除技术

广义上来说，重复数据删除技术也是数据压缩技术的一种，属于全局压缩。其原理是在文件系统全局范围内查找并用指示符替代不同文件中不同位置的重复的数据块。重复数据删除技术删除全局数据集中重复的数据，只对其中一个备份进行保留，以此消除冗余数据，如图 2.2 所示。

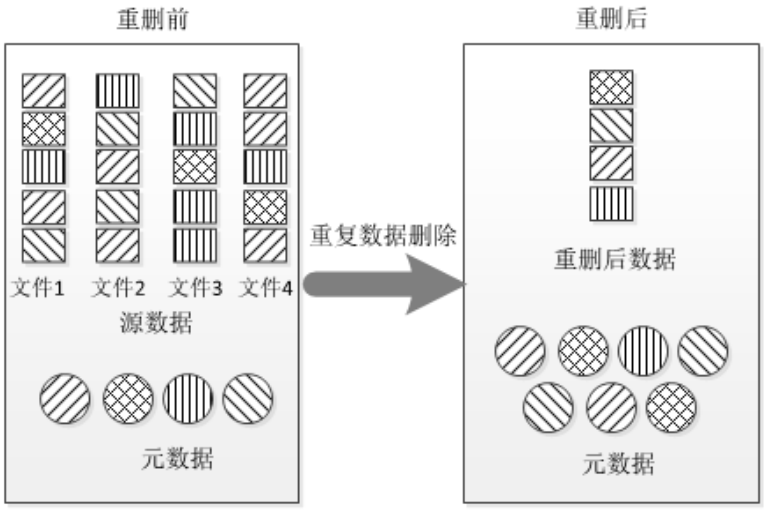


图 2.2 重复数据删除原理图

重复数据删除技术大多均使用在备份归档系统中，原因在于统计表明归档系统的不同版本间以及周期性的全备份系统中，总是存在大量重复数据。目前重复数据删

除技术在主存储系统中的应用尚不够广泛，主要存在以下原因。

（1） 重复数据删除技术需要对数据块进行切分、指纹计算以及检索，这些步骤会消耗可观的资源，对主存储系统性能产生影响；

（2） 主存储系统中主要数据需要每天更新和快速访问，因此需要避免因重复数据删除而造成的性能影响，确保只在合适的数据上应用重复数据删除技术。

（3） 从减少数据的方面来说，在主存储系统中的一般文件和图像数据往往都是唯一的，并且需要随机访问，因此重复数据删除仅能带来有限的数据减少。

目前在分布式文件系统中实现的主要还是文件级别的重复数据删除技术。文献[52]提出了一种通过使用 MD5 和 SHA-1 哈希函数计算文件的哈希值，然后将值传递给 Hbase 去重复表的文件级重复数据删除技术。下表 2.3 为无损数据压缩技术与重复数据删除技术的对比分析。

表 2.3 无损数据压缩技术与重复数据删除技术

类别	无损数据压缩	重复数据删除
作用范围	单个文件内	全局存储空间
冗余发现方法	字符串匹配	数据指纹
空间占用	降低到 1/10 与 1/2 之间	特殊情况下可降低很多
带宽占用	降低到 1/10 与 1/2 之间	部分情况下有降低
吞吐率影响	与具体压缩算法有关	不变
性能影响	较低	较高

2.3 无损数据压缩技术

无损压缩编码是基于信息熵原理，无损数据压缩编码属于熵编码，要求保存信息熵，在编码过程中不丢失信息量。与重复数据删除技术不同，无损数据压缩技术通常都作用于单个文件，并不考虑被压缩信息性质，将所有的数据当做比特序列，根据消息出现概率的分布特性而进行。对采用无损压缩后的数据进行解压缩，解压缩产生的数据应该与原始数据完全相同。图 2.3 为无损数据压缩处理流程的示意图。本节分析了几种不同类型的当前应用在分布式存储系统中的无损数据压缩技术，主要有 zlib 数据压缩库、bzip2 数据压缩库、Snappy 数据压缩库，然后对各压缩库的性能进行了横向对比，对比指标包括压缩率、压缩速率、解压缩速率等。

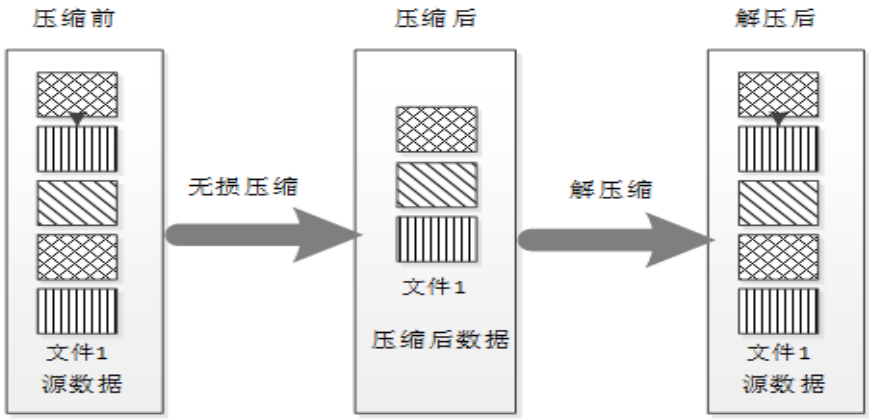


图 2.3 无损数据压缩原理图

2.3.1 zlib 数据压缩库

zlib 使用抽象化的 DEFLATE 算法，而 DEFLATE 算法又是 LZ77 与哈夫曼编码的组合，下面先对哈夫曼编码以及 LZ77 算法的原理进行简单介绍。

哈夫曼编码中定义符号为文件中一定位长的值，对文件中符号进行重新编码的是这些符号在文件中出现的频率。在编码的原理中，按照以下原则：对于出现次数很多的数值，采用略少的数字位来代替和表示，对于出现次数很少的数值，采用较多的数字位来代替和表示，整个编码过程就是根据这种替换来获得数据量的减少。进行哈夫曼编码的过程中，第一步是读取整个文件，在读的过程中统计并记录每个符号出现的次数。哈夫曼树是根据符号的出现次数建立的，通过哈夫曼树可以得到每个符号的新编码。

DEFLATE 算法综合采用了哈夫曼编码以及 LZ77 算法，其压缩流程如下图 2.4 所示。哈夫曼编码在 DEFLATE 中的实际实现包括了静态和动态哈夫曼编码两种，在对经过 LZ77 压缩后的数据进行哈夫曼编码时，会同时使用两种哈夫曼编码，然后对压缩结果进行比较，从中选取生成内容较短的一个座位某块数据最终的编码。此外 DEFLATE 还对 LZ77 编码的部分内容进行了调整，以使得在最差情况下，LZ77 编码所得到的最长长度也不会超过原始内容的长度[51]。LZ77 算法在解压缩的过程中，第一步需要从文件的开始到结束每一次都需要先读 1 个标志位，然后通过这个标志位来判断后面的数据内容是 1 个（两数据块间距离，数据块匹配长度）信息对，还是 1 个没有被压缩过程改动的字节，然后分别进行相应的处理，这部分工作主要是对数据进行不断的替换。从上述过程可以发现，LZ77 算法压缩时得要进行许多的字符串

匹配任务，但解压缩需要做的任务相对要少很多，这表明解压缩相对于压缩将会快很多。上述原理对于许多一次压缩，多次解压缩的应用场景来说是非常合适的，而这恰好也与分布式文件系统读写需求相匹配。

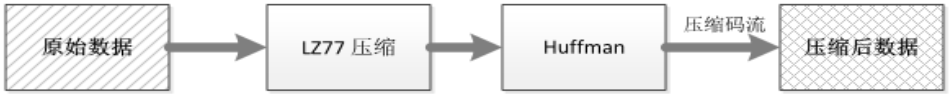


图 2.4　DEFLATE 算法的压缩流程示意图

2.3.2　bzip2 数据压缩库

bzip2 是由 Julian Seward 开发并开源的数据压缩库。普遍认为，在通常情况下 bzip2 的压缩率要略高于 zlib 等基于 LZ77/LZ78 的算法。bzip2 算法使用 BWT（Burrows-Wheeler Transform）排列文本压缩算法和哈夫曼编码来压缩文件[53]。在 bzip2 压缩算法中所有的数据将被切割，并被看作是大小一致的纯文本数据块。BWT 算法通过对输入的块数据执行一个可逆的变换来使得该块数据更容易被下一步的压缩算法所压缩，BWT 算法本身并不会对输入数据进行任何压缩处理。在此过程中，BWT 算法并不会流式地处理数据，而是将数据看作单个的单元，经过 BWT 处理过的数据块中的相同字符通常被聚集到了一起。在 bzip2 算法中允许输入块的大小从 10K 到 90K 共 9 个等级，块越大，压缩性能越好，但所消耗的内存和时间资源也更多，下表 2.4 为 bzip2 在 9 个等级配置下压缩解压缩 Linux kernel 2.6.11.0 源码包的性能记录。

表 2.4 bzip2 各级配置性能

等级	压缩率	压缩速率 MB/s	解压缩速率 MB/s
1	21.1%	2.9	16
2	19.7%	2.4	10
3	19.1%	2.1	8.3
4	18.7%	1.9	7.5
5	18.4%	1.7	7.0
6	18.2%	1.6	6.7
7	18.1%	1.5	6.5
8	17.9%	1.4	6.3
9	17.8%	1.4	6.2

bzip2 算法的主要流程如图 2.5 所示。

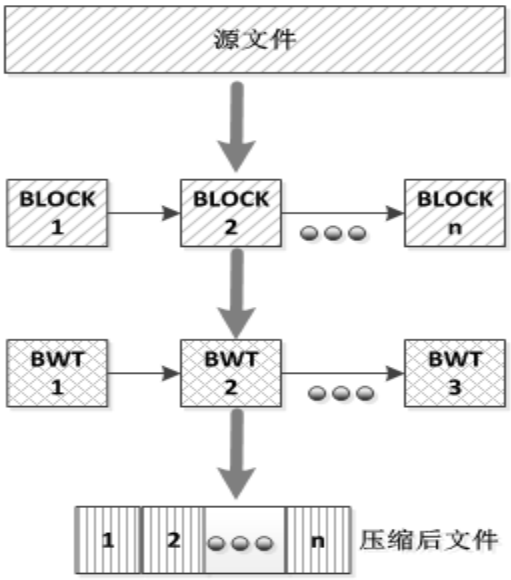


图 2.5 bzip2 算法的主要流程示意图

2.3.3 Snappy 数据压缩库

Snappy 是谷歌公司开源的一个 C++数据压缩库。Snappy 的设计目标并不是追求最高的压缩率，而是在较快的压缩速度之上仍然保证相对合理的压缩率。在某些情况下，Snappy 算法与默认模式的 zlib 压缩库相比，压缩速率能快接近一个数量级，当然其压缩率也相应的比 zlib 等压缩算法低 20%到 100%。总的来说，Snappy 是一个快速、稳定而且开源的数据压缩库。目前 Snappy 算法在谷歌内部的分布式文件系统、BigTable、MapReduce 中使用非常广泛。

2.3.4 性能对比

为了更好地理解和综合评价各无损压缩算法，下面对各无损压缩算法的三个指标即压缩率、压缩速率、解压缩速率进行测试分析，其中 zlib 选取的是 Z_DEFAULT_COMPRESSION 参数，即在最高压缩率与最快压缩速率之间取中间值；bzip2 采用了-9 参数，即最大压缩率。压缩率测试结果将在第四章测试与结果分析中进行详细的说明和分析，压缩速率测试结果如图 2.6 所示，解压缩速率测试结果如图 2.7 所示。

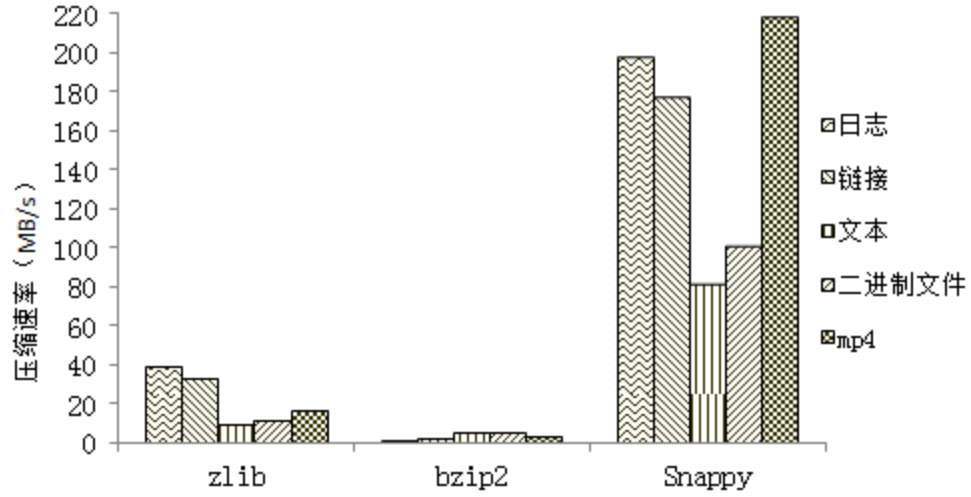


图 2.6 各压缩算法压缩速率

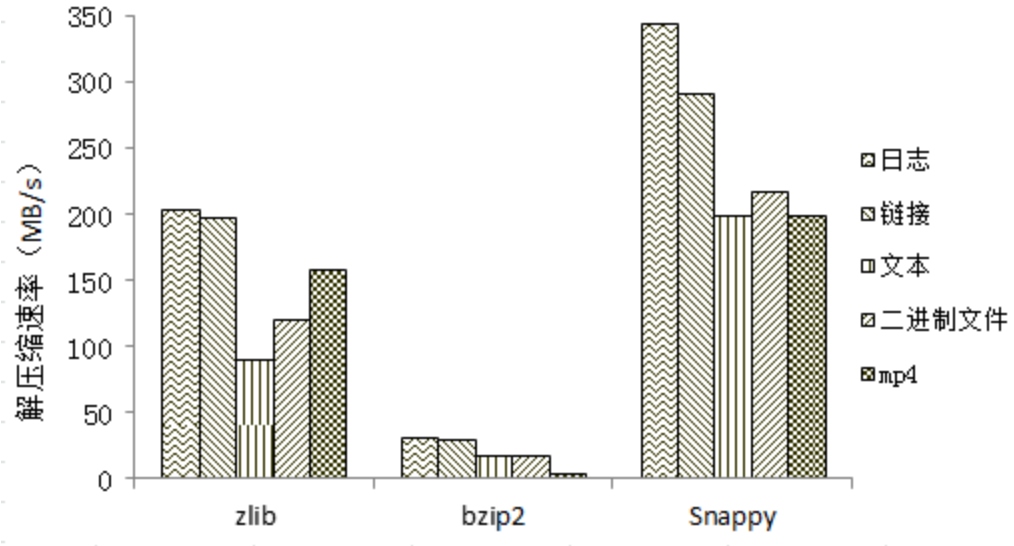


图 2.7 各压缩算法解压缩速率

由测试结果可以看出，对于不同的数据集，各压缩算法的压缩速率与解压缩速率往往都是不同的。bzip2 算法的压缩率往往最高，但对比来看其压缩速率也是最慢的，Snappy 算法的压缩速率和解压缩速率都最高，但是压缩率最低，zlib 算法相对来说处于一个中间位置，在压缩率与压缩速率之间控制较好。此外还可以看出，各压缩算法的解压缩速率均大于其压缩速率，这与 2.3.1 节中的分析是一致的。

2.4 分块检测、分块压缩方案分析

针对上述问题，本文提出了分块检测、分块压缩的解决方案，在压缩算法指标固有不变的情况下，通过应用策略的不同，改进压缩算法在分布式文件中的性能。分块检测、分块压缩方案首先按照分布式文件系统所存储的数据块大小（一般为 64MB）对文件进行分块，然后对各分块数据进行压缩处理。

虽然将整个文件进行预先分块后，使得损失了块之间的冗余，然而这个影响是非常小的，并且在分块较大的情况下，影响会更小。其原因在于，现今的无损数据压缩算法通常作用于数据流，消除冗余范围受到滑动窗口的限制，为了性能需要，通常采用较小的窗口，只能对局部数据产生作用。类似的压缩原理示例见上述的 2.3.1 节和 2.3.2 节。采用分块压缩与源文件压缩对比测试结果如表 2.5 所示，源文件大小为 640MB，分别切分为 10、20、64 个块，测试结果为各分块下压缩后大小比整块压缩后大小多出的百分比。由测试结果可以看出，虽然各个压缩算法间各有差异，但总的说来分块压缩带来的块间冗余依然是非常非常小的。

表 2.5 分块压缩与源文件压缩对比测试结果

分块个数	zlib 算法	bzip2 算法	Snappy 算法
10	0.002%	0.1%	0%
20	0.005%	0.2%	0%
64	0.006%	0.5%	0%

通过分块检测策略，可以针对分布式文件系统存储数据异构多样的特性，对待输入分布式文件系统的不同源文件进行分块的压缩率预测。对于 LZ 类型的压缩算法而言，对压缩率较低的数据进行压缩往往会消耗更高的资源和时间开销。比如采用 zlib 数据压缩算法库压缩相同大小的两种数据，压缩率为 0.9 的数据类型比压缩率为 0.1 的数据类型要多花 8 倍的时间。表 2.6 展示了 zlib 算法库对于不同压缩率文件所花费的压缩时间。

表 2.6 zlib 算法不同压缩率所用压缩时间

压缩率	压缩时间（微秒）
0.1	200
0.2	400
0.3	600
0.4	800
0.5	800
0.6	1000
0.7	1200
0.8	1300
0.9	1500

通过分块压缩策略，对待输入源文件预先进行分块，然后再采用多线程并发地对已分好块的数据进行压缩。分块压缩策略利用分布式文件系统分块存储数据的特点，将大文件预先分为 64MB 的数据块，将分块数据进行压缩后再写入分布式文件系统。通过这种方法，可以规避传统压缩策略全文件压缩的弊端，使文件数据仍然保持实际 64MB 大小的约束，也有利于上层 MapReduce 任务对各个数据块进行直接读取。分块压缩策略直接与分块检测策略结合，分块检测的结果可以直接应用于分块压缩，两种策略相辅相成。

分块检测、分块压缩的整个方案流程如图 2.8 所示。通过应用该数据压缩方案解决了上述问题：

（1） 通过采用分块检测，对待写入的源文件进行压缩率预测，在进行实际压缩之前预先判断源文件的压缩率是否符合需求，从而避免不必要的压缩浪费 CPU 以及内存资源，增加读写文件所花费时间。此外还可以将控制压缩在块级别上进行，仅对压缩率较高的块进行压缩，进一步优化压缩算法使用。

（2） 通过采用多线程并发压缩，从而利用多核优势，加快压缩时间，减少完成整个文件写入所需要的时间，充分利用 CPU 和带宽资源；

（3） 通过对数据先进行分块后压缩，保证了每个块中所存储实际数据仍然为 64MB 大小，有利于上层 MapReduce 应用对各个数据块并发进行处理，同时也允许了上层应用对单个数据块 block 进行操作，而无需读取整个文件。

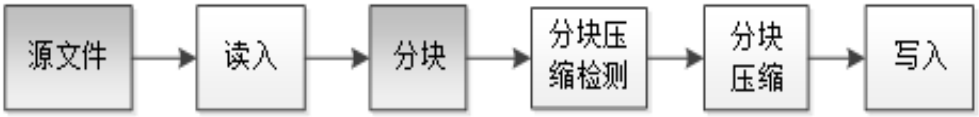


图 2.8 引入分块检测、分块压缩方案后写入文件处理流程示意图

2.5 本章小结

本章首先详细讨论了分布式文件系统目前面临的几个关键问题，包括存储成本高以及在数据密集型应用发展下日益增长的高吞吐率需求等。同时研究了业界普遍采用的一些现有解决方法，比如通过采用纠删码技术在保证数据可靠性的基础上减少副本数目，从而降低存储空间占用；重复数据删除技术着重于对备份系统中的冗余数据进行处理并替换，可以大幅度降低存储空间占用，并间接降低拷贝等操作带来的带宽占用，但重复数据删除技术在分布式文件系统中使用并不广泛。数据压缩技术通过对文件中的冗余数据进行压缩，通过增加部分计算量，可以显著减少存储空

间、传输带宽消耗，提高数据传输、处理效率。通过对现进分布式文件系统中所使用的压缩策略进行分析，提出了更适合分布式文件系统的分块检测、分块压缩存储方案。

3 分块检测、分块压缩方案设计与实现

信息存储与应用实验室自主研发的分布式文件系统 RaccoonFS 设计的主要目的是为解决海量大文件存储，为上层服务提供高吞吐量的数据访问，非常适合数据密集型应用等需要并发处理数据的场景。RacoonFS 与 HDFS 一样，对大文件进行分块存储，每个数据块 64MB（可配置），并将数据块保存三个副本。RaccoonFS 的整体架构也与 GFS 和 HDFS 等成熟文件系统类似，为典型的三方架构：NameServer、DataServer 和 Client。RaccoonFS 架构如图 3.1 所示。其中 Client 为读写请求发起端，NameServer 为元数据存储端，而 DataServer 服务器为实际数据存储端，各个数据块按 64MB 大小进行存储。

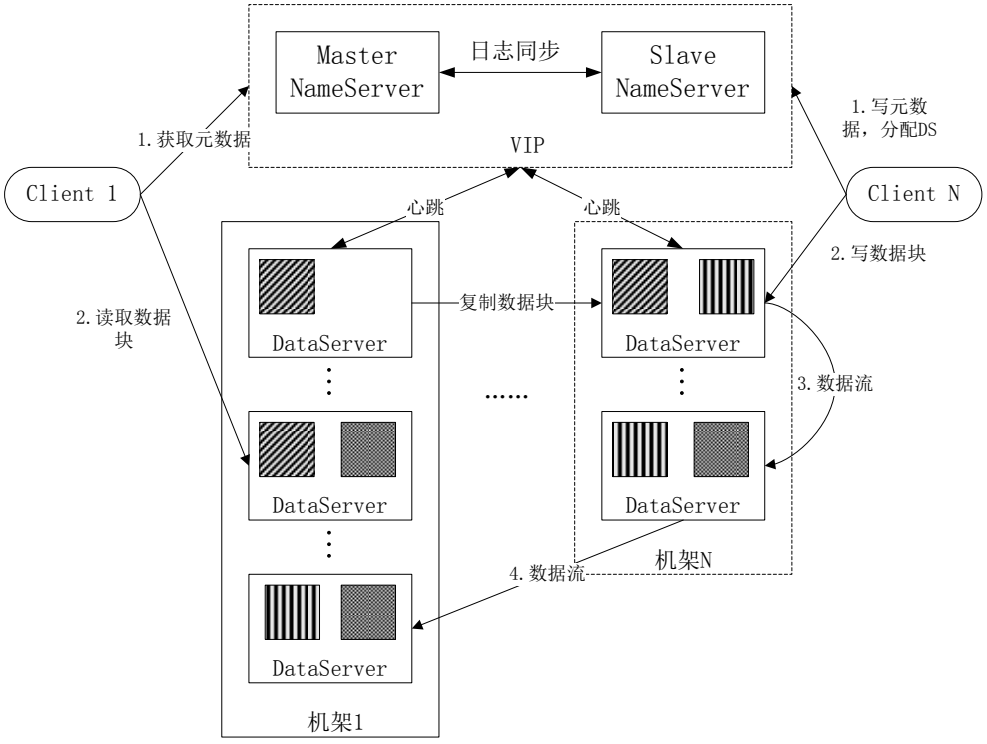


图 3.1 RaccoonFS 整体架构

本文在 RaccoonFS 分布式文件系统的基础上，实现了数据压缩的各种算法，并以此为基础，在客户端的基本框架上，实现了分块检测、分块压缩方案。本章将详细阐述在 RacoonFS 分布式文件系统中实现的分块检测、分块压缩方案，内容包括相应的压缩算法管理模块、分块检测模块、分块压缩模块的具体设计与实现以及引入分块检测、分块压缩方案后 RaccoonFS 分布式文件系统的读文件、写文件流程。

3.1 架构设计

RaccoonFS 分布式文件系统中为了优化当前存储系统的压缩策略，解决当前存储系统存在的问题，引入了分块检测、分块压缩的方案，方案总体结构如图 3.2 所示。

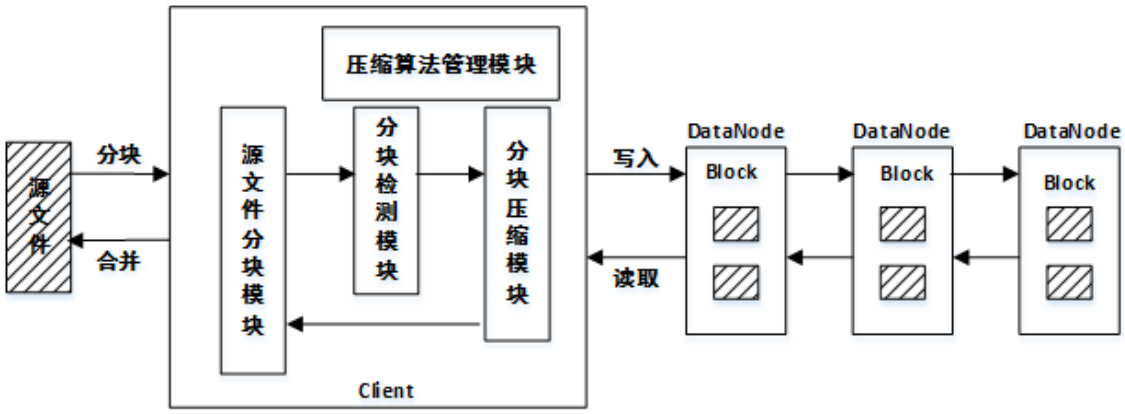


图 3.2 分块检测、分块压缩方案架构图

分块检测、分块压缩方案的主要功能模块如下。

源文件分块模块：负责对源文件进行判断，首先判断文件后缀名是否需要不可压缩集合，然后判断该文件大小，并按照 64MB 每块预先分块；在解压缩时，对读到客户端的各个分块进行合并，形成原始文件。

压缩算法管理模块：针对现在业界存在各种不同的压缩算法，且各种压缩算法的压缩率以及压缩速率往往差异较大的问题，允许用户可配置的采用各种压缩算法，以便满足更高压缩率或者更短压缩时间的各类需求。

分块检测模块：针对分布式文件系统存储文件格式多样的特性，对待输入源文件进行分块压缩率预测，预测文件可能的压缩比，避免对不能压缩的文件进行不必要压缩。

分块压缩模块：缓解读写速率受到数据压缩算法限制的弊端，满足分布式文件系统高吞吐率需求。采用多线程对大文件的各个数据块并发进行压缩。系统架构如图所示。

3.2 基本数据结构设计

本节首先对系统中所涉及到的基本数据结构进行了简单说明。通过这些数据结构，可以进一步认识 RaccoonFS 分布式文件系统中文件的存储方式，以及文件和数

据块之间的组织关系。

RaccoonFS 与现有主流分布式文件系统一样，采用了三方架构的方式来存储文件，整个系统分为元数据服务器端 NameServer、数据服务器端 DataServer，和客户端 Client。将文件的元数据和数据分离，元数据存储在 NameServer 上，数据存储在多个 DataServer 组成的集群上，通过 Client 端发起读写删除等常规文件操作。NameServer 中采用 B+树数据结构存储元数据信息，如表 3.1 所示为 RaccoonFS 中所存储文件的主要元数据信息。

表 3.1 RaccoonFS 中文件的元数据结构

file ID	文件标识符
file type	文件类型
replica number	文件的副本数目
mtime	文件的修改时间
crtime	文件的创建时间
block count	文件包含的数据分块个数
file size	文件的大小
block ID	组成该文件的数据块的 ID

其中，与 Linux 中的系统文件类似，file ID 为文件在系统中的唯一标识符，用一个 64 位整型数表示，ID 号不断递增；file type 表示文件类型，包括普通文件、目录两种类型；replica number 表示文件副本数，默认值为 3，即同一文件存三个备份；mtime 和 crtime 分别表示该文件的修改时间和创建时间；block count 表示该文件所占有的 block 个数，除最后一个 block 外，每个 block 均为 64MB；file size 表示该文件的大小。file ID 与 block ID 信息的映射关系存储在 B+树数据结构中，可以通过 B+树提供的 getalloc()接口获得。

与 file ID 类似，block ID 是数据块在系统中的唯一标识，用一个 64 位整型数表示。num bytes 为数据块长度；block version 为当前数据块的版本号，用来区分当前块是否属于过期版本；file ID 为当前数据块所属的文件号。系统中还使用了一个 Map 数据结构来保存 block ID 到 DataServer ID 的映射，DataServer ID 为各个数据服务器在文件系统中的唯一标识。通过 DataServer ID，客户端即可知道向哪个数据服务器发起读写请求了。系统中涉及到 block ID 的 Map 均选用了 hash 方式的 Map 实现，其原因在于，block ID 的数目往往比较巨大，相较于 STL 中的红黑树实现，hash 方式能提供更快的查找读取时间。数据块元数据结构如表 3.2 所示，其中最重要的数据类型为各个 ID 号。

表 3.2 RaccoonFS 中数据块的元数据结构

block ID	数据块标识符
num bytes	数据块长度
block version	数据块版本号
file ID	数据块所属文件号
DataServer ID	数据服务器标识符

3.3 源文件分块模块

由于分布式文件系统所存储的源文件具有各种形式和大小，因此需要在将源文件写入分布式文件系统前，对源文件首先进行基本的判断，判断该文件是否能够压缩，是否能够被分块。模块中采用文件后缀名识别的方法判断该文件是否属于不可压缩集合，然后通过指针定位获取文件大小，判断该文件是否能够进行下一步的分块处理。在分布式文件系统读文件阶段，经过对各分块解压缩后，该模块还需要负责对各个模块进行组合，从而可以根据需求重新生成源文件。

在文件后缀名识别部分，模块首先对待读入源文件的文件名进行分析，查看其在文件名末尾是否具有“.”分隔符。若没有该分隔符，则认为该文件需要进一步分块检测；若有该分割符，则通过对该分隔符后的文件类型名进行哈希判断，查看该文件类型名是否属于不可压缩集合，若不属于则继续进行大小检测。不可压缩集合中的文件类型名是可配置的，所包括的类型包括各种多媒体文件如 MP3、AVI、JPG 等以及被其他压缩算法（如 LZO、7z、bzip2 等）所默认使用的后缀类型，包括 bz、7z、rpm、gz 等。

在文件大小判断分割部分，模块首先调用 C 函数 fseek()，将文件定位到文件末尾，然后调用 ftell()函数计算文件长度。如果文件长度大于 64MB，则对文件进行定位分块，此处同样是调用 fseek()函数。在分割过程中，以及分割完成后，均不会对源文件数据造成影响。

在读文件阶段，该模块对解压缩后的内容，调用相应的 fseek()函数，采用追加写方式，将各个数据块 block 中的内容写入新的文件中，从而完成对各数据块的组合工作。其他相关函数定义及说明如下：

(1) int CompressPreSplit(char * sourcefile, char* destfile, int type);

功能描述：从本地读取文件并判断当前文件是否可以被进一步进行分块，type 为选取的压缩算法类型。

3.4 压缩算法管理模块

随着压缩算法研究的不断发展，目前业界所流行的压缩算法越来越多，各种压缩算法往往具有不同的偏重点。bzip2 等压缩算法追求压缩率，LZO、Snappy 等压缩算法追求压缩速率，还有其他一些压缩算法追求解压缩速率，追求 CPU 资源和内存资源的最小占用。因此有必要加入压缩算法管理模块对各压缩算法进行管理，允许动态地对各压缩算法进行配置，满足业务的各种需求。在压缩算法管理模块中采用了工厂方法以及策略模式两种设计模式来使得动态引入压缩算法对系统的影响最小。设计了基类 CompressAll 作为所有压缩算法类的父类，通过多态实现各自的分块检测算法、分块压缩算法与解压缩算法。目前在 RaccoonFS 文件系统中引入的无损压缩算法包括 zlib、bzip2 和 Snappy 算法。下面以 zlib 数据压缩库为例，对压缩算法在文件系统中的应用进行阐述。

为了将 zlib 算法库集成到客户端 Client，满足两方的接口调用，同时尽量提高性能，减少压缩流程对正常读写流程的影响，需要调用各算法库的底层接口，使用底层的压缩流结构。RaccoonFS 中 zlib 压缩算法的压缩过程具体实现如图所示，代码中省略了部分非核心部分，时间复杂度为 O(n)。n 为源文件长度，CHUNK 为每次读入的数据大小，source 为源文件名称，in 和 out 分别为输出缓冲区和输入缓冲区。

算法 3.1

RaccoonFS 中 zlib 压缩算法实现

```
1.  输入：源文件
2.  输出：压缩后写到服务器端的文件
3.  cons tint CHUNK  = 16384
4.  do {
5.          strm.avail_in = fread(in, 1, CHUNK,  source);/*从 source 中读入数据*/
6.          flush = feof(source) ? Z_FINISH : Z_NO_FLUSH;  /*若到文件尾，设置标记为结束*/
7.          strm.next_in = in; /*指向待输入缓冲区*/
8.          /* 执行 deflate()压缩  直到输出缓冲区满*/
9.          do {
10.             strm.avail_out = CHUNK;
11.             strm.next_out = out; /*  设置输出缓冲区  */
12.             ret = deflate(&strm, flush);       /*  根据参数压缩  */
13.             have = CHUNK  - strm.avail_out; /*  压缩后数据长度  */
14.             writelen = client->Write(fid,  out, have); /*  将数据写入 DataServer*/
15.             writeall += writelen;
16.         } while (strm.avail_out == 0); /*  若可输出长度为 0 */
17.         assert(strm.avail_in == 0);        /*  所有输入缓冲区数据均被压缩  */
```

```
18.             readall += readlen;
19.   }while (flush != Z_FINISH);
20.  (void)deflateEnd(&strm); /*  关闭 strm 压缩流 */
```

RaccoonFS 中 zlib 压缩算法的解压缩过程具体实现如图所示，时间复杂度为 O(n)，n 为源文件长度，CHUNK 为每次读入的数据大小，source 为源文件名称，in 和 out 分别为输出缓冲区和输入缓冲区。

算法 3.2

RaccoonFS 中 zlib 解压缩算法实现

```
21.  输入：DataServer 中的压缩文件
22.  输出：解压缩到本地的源文件
23.  cons tint CHUNK  = 16384
24.  do {
25.             strm.avail_in = client->Read(fid, (void *)in, CHUNK);/*从 RaccoonFS 中读入数据*/
26.             strm.next_in = in; /*指向待输入缓冲区*/
27.             /*  调用 inflate()解压缩  直到输出缓冲区满*/
28.             do {
29.                  strm.avail_out = CHUNK;
30.                  strm.next_out = out; /*  设置输出缓冲区  */
31.                  ret = inflate(&strm, Z_NO_FLUSH);        /*  根据参数解压缩  */
32.                  have = CHUNK  - strm.avail_out; /*  解压缩后数据长度  */
33.                  writelen = fwrite(fid, out, have); /*  将数据写入到本地*/
34.                  writeall += writelen;
35.             } while (strm.avail_out == 0); /*  若可输出长度为 0 */
36.             assert(strm.avail_in == 0);         /*  所有输入缓冲区数据均被解压缩  */
37.             readall += readlen;
38.   }while (flush != Z_FINISH);
39.  (void)deflateEnd(&strm); /*  关闭 strm 解压缩流 */
```

为了保证输出缓冲区能够完全存储解压缩后的数据，有些压缩算法可以通过调用来相关函数来确定最小输出缓冲区的长度。比如 Snappy 算法就可以通过 GetUncompressedLength()函数调用来实现，从而最小化内存占用。

3.5 分块检测模块

鉴于分布式文件系统存储文件格式多样的特性，需要对待输入源文件进行压缩率预测，预测文件可能的压缩比，从而避免对不可压缩文件进行压缩，减少不必要的性能损失。在 RaccoonFS 分布式文件系统中主要采用了两种方式对源文件进行压缩率预测，一种是前述的后缀名检测，判断后缀名是否属于不可压缩类型，如果后缀

名不属于不可压缩类型，需要对源文件进一步进行分块检测，通过分块检测来判断文件可能的压缩率。本节主要对分块检测模块的具体设计和实现进行了详细阐述。

所谓分块检测是通过对分块后的各个数据块进行并发压缩率预测，判断该数据块可能的压缩率，以及整个文件可能的压缩率。由于各压缩算法底层实现原理不同，需要针对不同压缩算法，实现不同的压缩率预测方法。比如 zlib 数据压缩库，就需要根据待压缩文件类型的不同，多次选取底层的数据进行压缩判断，直到得出一个合理值。但总体而言，分块检测的流程如下：对于各分块数据块，随机选取一部分数据进行压缩，然后对该部分数据的压缩率进行计算，最后汇总压缩率结果，对压缩率取平均值后作为整个文件的压缩率。整个算法流程如算法所示。

算法 3.3

RaccoonFS 中分块检测算法伪码实现

```
输入：分块数据，期望压缩值
输出：压缩比值，bool：是否低于期望压缩值
cons tint CHUNK  = 16384
do {
        readlen = fread(in, 1, CHUNK,  source);/*从 source 中读入待检测数据，
                                                读取长度为 readlen*/
        flush = feof(source) ? Z_FINISH : Z_NO_FLUSH;  /*若到文件尾，设置标记为结束*/
        ret = deflate(&strm, flush);       /*  根据参数压缩  */
        have = CHUNK  - strm.avail_out; /*  压缩后数据长度  */
        writeall += have;
        readall += readlen;
        ret = ComputeRatio();/*  计算压缩值  */
  }while (ret == true); /*  压缩值是否合理  */
(void)deflateEnd(&strm); /*  关闭 strm 压缩流  */
```

采用分块检测的好处在于使得压缩率的预测更加准确，此外，鉴于分布式文件系统采用分块存储的特性，采用分块压缩，上层业务可以根据需要来决定在数据块级别还是文件级别进行压缩，这对于内部数据不均匀的文件来说显得更有意义。

3.6 分块压缩模块

为了最大限度的解决读写速率受到数据压缩算法限制的弊端，满足分布式文件系统高吞吐率需求，在 RaccoonFS 分布式文件系统中引入了分块压缩模块，采用多线程对大文件的各个数据块并发进行压缩。分块压缩模块的结构图如图 3.3 所示，主要

由压缩消息队列以及压缩线程池组成。其中压缩请求消息队列接受来自分块检测后的 block 的分块信息，和所属的文件信息；压缩线程池主要负责从压缩请求消息队列中获得的任务并将这些任务并发执行，然后将其写入数据服务器端 DataServer。本节将对该模块进行详细地阐述。

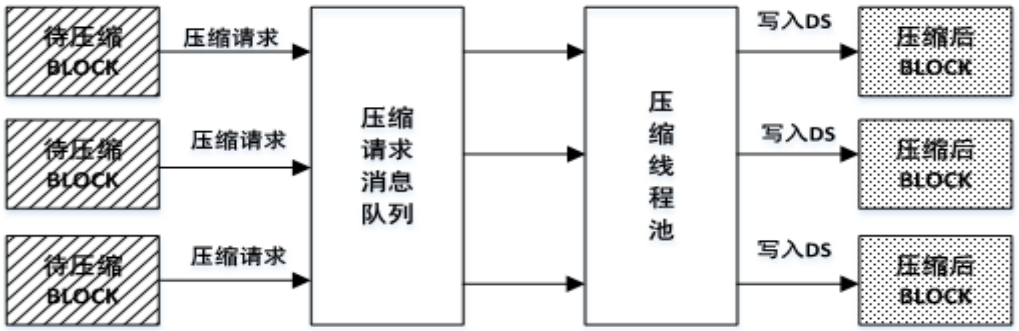


图 3.3 分块压缩模块的结构图

3.6.1 压缩请求消息队列

压缩请求消息队列主要接受来自客户端中其他模块发送的请求消息，包括压缩请求和解压缩请求。消息包的基类为 BladePacket，提供的主要接口函数定义说明如下：

（1）virtual int pack();

功能描述：将消息包中的成员变量等进行打包填充，成为数据流格式。成员标量包括 file ID，block ID 等。

（2）virtual int unpacket();

功能描述：将数据流格式中的消息解释出来，重新生成消息包。

（3）int reply(BladePacket * resp_packet)

功能描述：发送消息响应包。

为了使单个线程能够直接对待压缩的块进行压缩，以及并发地将压缩后的内容直接写入 DataServer 端，CompressPacket 中需要包含的数据结构如下表 3.3：

表 3.3 RaccoonFS 中数据块的元数据结构

sourcefile	源文件名称
destfile ID	目标文件 ID
source offset	源文件偏移
type	压缩算法类型
block ID	数据块 ID
dataserver ID	待写入数据服务器 ID
length	数据块长度

压缩请求消息队列 TaskQueue 是在 C++ STL 提供的 std::queue 上封装实现的，加入了信号量操作、多线程互斥锁等并发控制机制，使得该队列能正常支持多线程环境下的 push()和 pop()操作。TaskQueue 提供的主要接口函数定义如下：

（1）bool TaskQueue<T>::WaitTillPush(const T& node)

功能描述：等待直到允许将压缩消息包 push 到 queue 中，主要是通过调用 sem_wait()以及 sem_post()实现，这两个函数均是 C 语言中控制信号量的原子操作。

（2）bool TaskQueue<T>::WaitTillPop(const T& node)

功能描述：等待直到允许将压缩消息包从 queue 中 pop 出来，主要是通过调用 sem_wait()以及 sem_post()实现。

3.6.2 压缩处理线程

压缩处理线程主要是负责并发地对从 TaskQueue 中提取出的消息包进行解压缩，然后按照消息包内容进行相应的压缩/解压缩以及读写操作。为了更简洁有效地进行任务处理，对压缩线程进行了封装，设计实现了继承 Runnable 方法的 CompressMultiThread 类来进行管理。

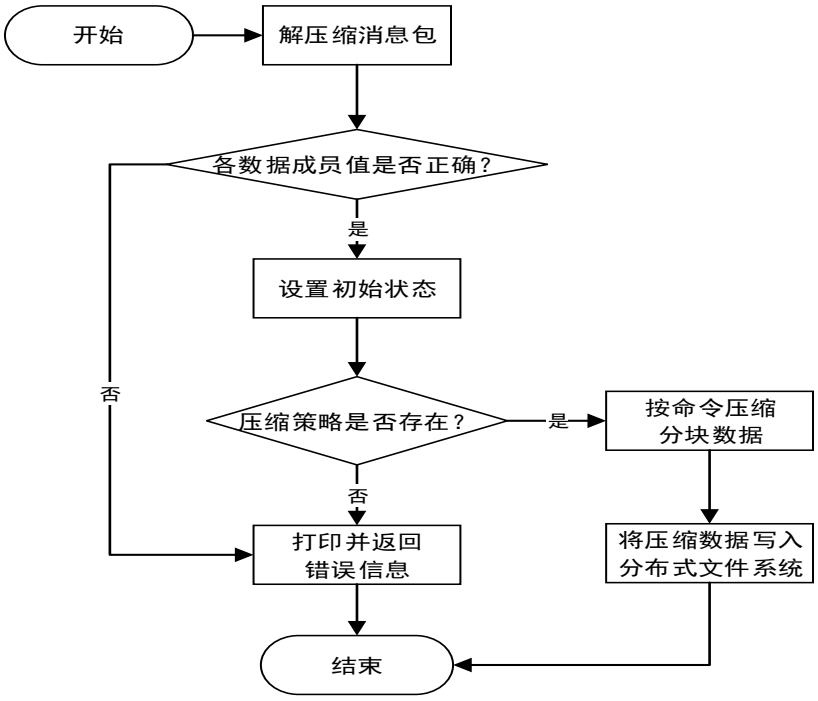


图 3.4 压缩处理流程图

线程获取到 CompressPacket 包后，首先解压缩提取表 3.3 所示的数据结构，通过

block ID、destfile ID 等设置初始状态，然后通过 sourcefile 与 source offset，读取源文件相应位置中数据，然后根据 type 所示的压缩算法类型，对数据进行压缩，接着向 dataserver ID 所表示的的数据服务器写入块数据，在写入数据前，先向其确认好 destfile ID，与 block ID 等必要的信息。整个流程如图 3.4 所示。

3.7 文件读写流程

为了更好地阐述整个分块检测、分块压缩方案在 RaccoonFS 分布式文件系统中的应用，更好地理解整个压缩、读写流程，本节将对加入了分块检测模块，分块压缩模块的客户端完整读写流程进行详细阐述。

下面首先介绍 RaccoonFS 实现分块检测、分块压缩方案后的写流程，客户端对文件系统写入数据的流程如下：

（1） 客户端首先初始化相应的系统参数，然后根据用户调用的 Create()接口向 NameServer 发起创建文件的请求，NameServer 检查文件是否合法，然后在名字空间的 B+树中增加一个文件节点，并返回一个文件 ID。

（2） 客户端调用源文件分块模块，判断源文件是否需要被分块、是否能够被压缩。如果均能够，则将源文件进行分块，将分块信息传递给分块检测模块。

（3） 客户端的分块检测模块，对分块信息进行检测，判断其是否满足业务需需要的压缩率，若满足，则调用 Addblock()接口向 NameServer 请求该文件的 block ID 信息、DataServer ID 信息，以及块长度信息，并将这些信息压缩到 CompressPacket 消息包中传递给分块压缩模块。

（4）客户端的分块压缩模块通过获得的 CompressPacket 消息包，对相应的 block 进行并发的压缩操作，将压缩后的数据调用 WriteToDataServer()接口写入 DataServer 中。

（5） 当客户端开始向 DataServer 写入 block 的时候，采用三副本备份策略（默认），向 DataServer 申请建立流水线，如图 3.5 所示。如果其中某个 DataServer 出现异常，则放弃该 block，向 NameServer 返回错误信息。

（6） 客户端建立流水线后，开始以流水线的形式将数据块（包含校验信息）写入三个 DataServer 中，只在流水线中最后一个 DataServer 上做数据校验。

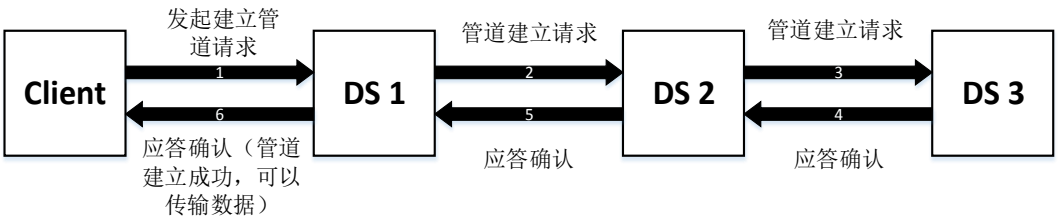


图 3.5 流水线的建立

（7） 当 block 传输完毕，客户端会移除租约，重新返回步骤（4）接受新的 CompressPacket，直到整个文件都写完。整个操作的流程图如图 3.6 所示。

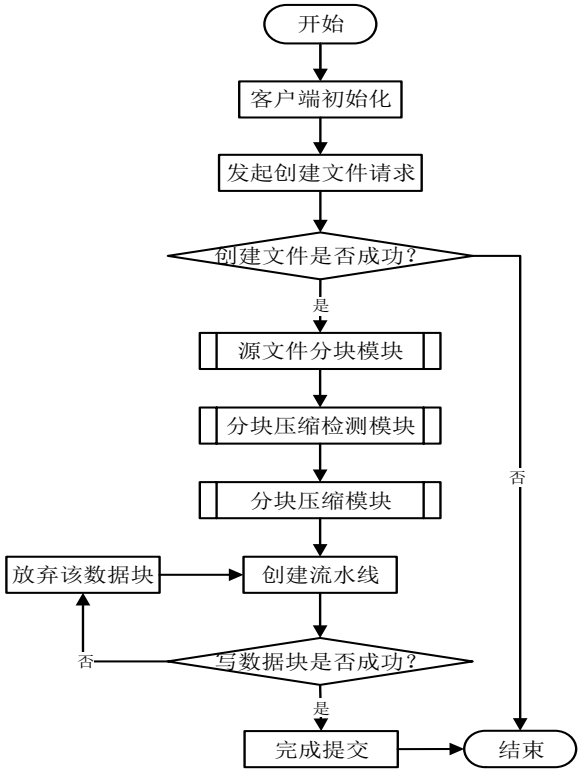


图 3.6 写文件流程图

下面介绍 RaccoonFS 实现分块检测、分块压缩方案后的读流程，客户端从系统中读取文件的流程如下：

（1）客户端首先初始化相应的系统参数，然后根据用户调用的 Open()接口查询本地缓存的元数据，如果本地缓存中有要读取文件的元数据信息，则直接根据元数据信息到 DataServer 上读取文件，跳到步骤（3），否则跳到步骤（2）。

（2）客户端向 NameServer 发起读取文件的请求，NameServer 检查文件名是否存在，若存在则返回相应的文件 ID。

（3）客户端根据参数向 NameServer 发起读取文件请求。NameServer 根据请求找到指定文件的文件 ID 和 block 信息并返回，此处 NameServer 会将三个 DataServer 根据负载均衡策略进行排序，保证优先读取负载轻的 DataServer。

（4）客户端选取负载轻的 DataServer 建立链接，并并发地分块读取 block 数据，若读取 block 数据时出错，则从排序列表中选取下一个 DataServer 读取同一个 block 数据。

（5）选取列表中的第一个 DataServer 读取 block 数据，如果读取操作成功，则继续读取下一个 block。若读取数据时出现读出错，则从列表中下一个 DataServer 上读数据；若所有 DataServer 都无法读取，则向客户端返回错误。

（6）当读完所有 block 后，文件的读取还没有结束，客户端会继续向 NameServer 获取下一批的 block 列表，否则读流程结束。整个操作的详细流程如图 3.7 所示。

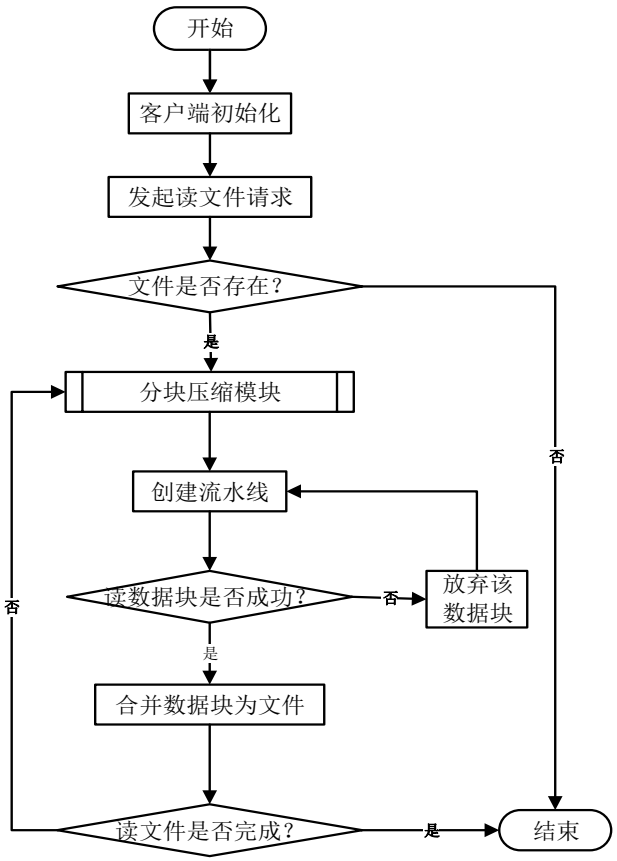


图 3.7 读文件流程图

3.8 本章小结

本章详细论述了在 RaccoonFS 分布式文件系统中实现的分块检测、分块压缩方案。

首先，对 RaccoonFS 分布式文件系统的整体架构进行了说明，并在此基础上引出了本文分块检测、分块压缩的压缩策略应用方案。接着对在本文方案中需要用到的基本数据结构进行了介绍。

其次，详细介绍了分块检测、分块压缩方案中的源文件分块、压缩算法管理、分块检测、分块压缩这几大模块，说明了其用到的一些函数和功能，并对重要的步骤给出了算法执行步骤，阐述了各模块之间的调用关系。

最后，对各模块的实际实现过程做出了总结，并详细阐述了在 RaccoonFS 分布式文件系统中实现了分块检测、分块压缩方案之后的完整的读写流程，以及流程图说明。

4 测试与结果分析

本章首先分析阐述了将压缩算法应用到 RaccoonFS 分布式文件系统中后对于不同数据集的读写速率比较。接着分析阐述了将分块检测、分块压缩方案应用到 RaccoonFS 中的测试结果，然后对分块检测模块的压缩预测准确率进行了测试分析，最后对分布式文件系统引入分块检测、分块压缩方案、未引入压缩、引入传统压缩策略三种方式进行了对比测试分析。

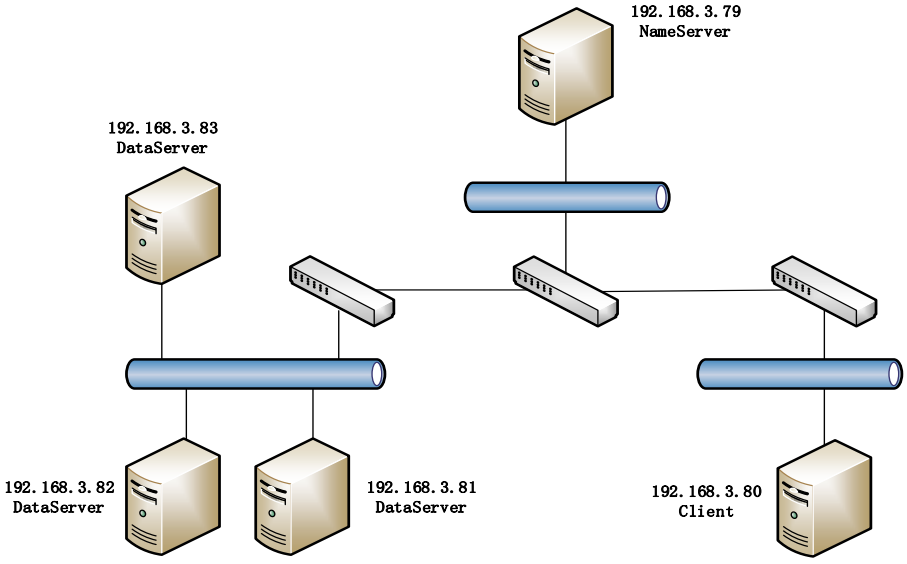
4.1 测试环境

测试所采用的服务器配置均相同，其配置如表 4.1 所示。单独使用 1 台服务器作为 NameServer，使用 3 台服务器作为 DataServer，1 台服务器作为客户端，各个服务器之间用千兆局域网连接。

表 4.1 测试配置

设备	配置
CPU	Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
内存	DDR3 16GB
硬盘	SATA 300GB @ 7200R/M
操作系统	Red Hat Enterprise Linux Server release 5.4

系统网络拓扑图如图 4.1 所示。



4.1 系统网络拓扑图

由于分布式文件系统主要是存储非结构化数据类型，因此测试所用的数据集主要也是从非结构化数据类型范围内选取的，包括维基百科的部分操作日志（log），维基百科的部分 URL 链接（link），维基百科的部分文章（article），以及用于无损压缩算法测试的二进制文件（exe）和一个已进行过有损数据压缩的多媒体文件（mp4）。数据压缩库选择了较通用和传统的 zlib、bzip2 和 Snappy，其中 zlib 采用了默认的方式 Z_DEFAULT_COMPRESSION 参数，即在最高压缩率与最快压缩速率之间取中间值；bzip2 采用了-9 参数，即最大压缩率。

4.2 集成压缩算法性能测试

在 RaccoonFS 分布式文件系统中集成 zlib、bzip2、Snappy 无损压缩算法后，对上述的非结构化数据数据集，包括日志、URL 链接等进行测试，首先测试了各种无损压缩算法对非结构化大文件数据集的压缩率，然后测试了各压缩算法在读写负载比例为 8:2 情况下对分布式文件系统吞吐率的影响。

不同压缩算法对于各非结构化大文件数据集的压缩率测试结果如图 4.2 所示。由图可以看出，对于不同的数据集，采用相同的压缩算法，其压缩率是不一样的；并且对于同一测试数据，不同压缩算法的压缩率也是不一样的。就普遍情况来看，zlib 的压缩率在 0.3 左右，bzip 压缩率最低，而 Snappy 的压缩率在 0.5 左右，三种压缩算法均不能对已进行有损压缩后的 mp4 文件进行压缩。这表明压缩算法对于非结构化数据集的类型是非常敏感的。

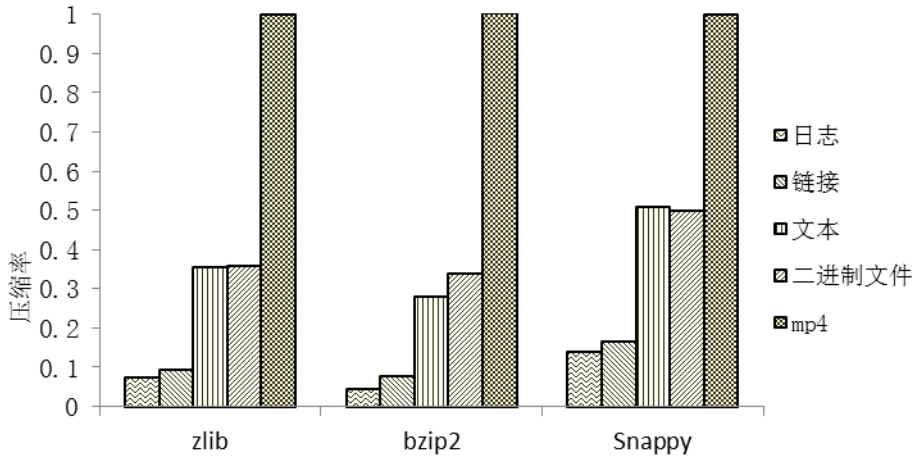


图 4.2 压缩率对比

引入压缩算法后，在模拟的读写比例为 8:2 情境下，分布式文件系统吞吐率测试结果如图 4.3 所示，其中测试数据集选取了压缩率相差较大的文本和链接两种类型。文本类型数据集压缩率较低约在 0.5 左右，压缩速率较慢；而链接型数据集压缩率较高约在 0.1 左右，压缩速率较高。压缩算法的整体压缩速率，以及待压缩数据集的压缩速率对文件系统的吞吐率有很大影响。对于 bzip2 这种无论对任何数据类型，压缩速率和解压缩速率都非常慢的算法（参数为-9），整个系统的吞吐率大概只有 20MB/s 左右。而对于 zlib 这种平均压缩速率约在 40MB/s，解压缩速率约在 100MB/s 的算法，其对分布式文件系统吞吐率的影响主要来自待压缩数据集。若待压缩数据集压缩率较高，压缩速率较快，则系统吞吐率能提高到近 300MB/s，若待压缩数据集压缩率较低，仅为其最差压缩率，则系统吞吐率基本与未采用压缩算法情况下相当。对于 Snappy 这种无论对任何数据集压缩解压缩速率都非常快的算法，其对吞吐率的影响总是促进的，能够极大地提高系统的吞吐率。因此 Snappy 算法非常适合用于处理并行计算的中间结果。可以得出，客户端的读写速率完全受到了压缩算法的影响，如果压缩算法的速率较慢，那么整个读写流程的速率将严重受到压缩算法速率的影响，降低系统性能。相比于写流程，读流程受到压缩算法影响的程度更低，原因在于绝大多数压缩算法的解压缩速率相对都比较高。

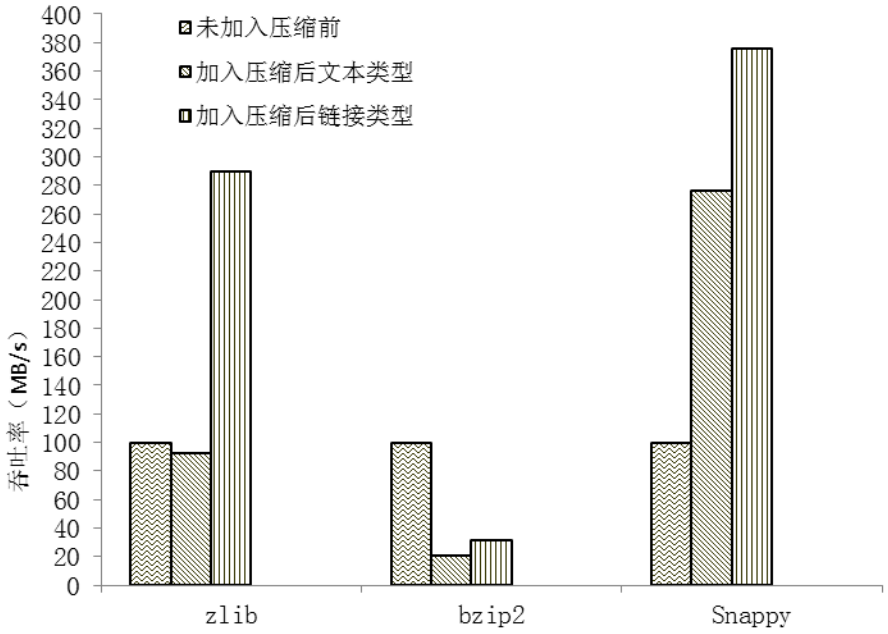


图 4.3 读写比例 8:2 下吞吐率对比

由测试结果总结得出：为了在分布式文件系统中更高效的使用压缩算法，需要在应用该压缩算法前对它的压缩率、压缩速率和解压缩速率等方面有一个充分的了解。如果业务需要更高效的压缩率，需要采用 zlib、bzip2 这样的压缩算法，而如果业务需要更快的传输处理时间，需要采用 Snappy 这样的压缩算法。为了避免不必要的资源浪费，在进行实际压缩前对待压缩文件进行压缩率预测是非常有必要的，提高分布式文件系统中引入各种压缩算法后的读写性能也是非常有必要的。

4.3 分块检测准确率测试

表 4.2 所示为 RaccoonFS 分布式文件系统中，分块检测模块测得的压缩率与实际压缩率测试结果对比。使用 zlib、bzip2、Snappy 算法分别对上述的非结构化数据数据集，包括日志、文本等进行压缩率测试。可以看出各算法对不同的结构化数据均有较好的压缩率，而对于多媒体 mp4 则不能进行压缩。采用分块检测得出的压缩率与实际压缩率得出的数据非常接近，处于一个可接受的误差范围内，其中对于日志型数据的误差基本在 10%以内，而对于不可压缩的 mp4 文件，几乎不存在误差。由此可以得出，对于常规的非结构化数据集，分块检测能够很好地预测整个文件的实际压缩率，通过分块检测能够大大减少分布式文件系统不必要开销，避免对不可压缩文件进行压缩，造成资源浪费。同时分块测试模块结果还可以对待压缩数据提供压缩率参考，使得业务在未进行实际压缩前就对待压缩数据的可压缩程度有一个较准确的判断。

表 4.2 分块检测预测压缩率与实际压缩率对比

文件格式 / 压缩算法	实际压缩率			分块检测压缩率		
	日志	文本	mp4	日志	文本	mp4
zlib	0.07	0.35	1	0.06	0.33	1
bzip2	0.04	0.28	1	0.04	0.29	1
Snappy	0.13	0.56	1	0.13	0.60	1

4.4 分块压缩性能对比测试

分块压缩性能对比测试主要测试在 RaccoonFS 中引入分块压缩策略后，系统的读写速率性能，并将测试结果与未采用压缩策略方案、采用传统压缩策略方案两种方式进行了对比。未采用压缩策略方案即文件系统按照正常情况进行读写，其中未引入任何压缩策略。写文件时，数据按照正常的流水线方式写入三个 DataServer 中，

三个副本均写入成功才算写文件完成，除最后一块数据块外，数据按照每块 64MB 大小存放；在读文件时，客户端自动选择负载最轻的 DataServer 读取文件。采用传统压缩策略方案即在文件系统中引入了压缩模块，在进行文件读写之前，先对数据进行正常的压缩，写完成后除最后一块数据块外，每个 block 存放压缩后的 64MB 数据。

写文件阶段，三种方案测试对比结果如图 4.4 所示。其中 RaccoonFS 分布式文件系统在未采用压缩策略前的写速率大概是 60MB/s，采用传统压缩策略后，虽然系统中传输和存储的数据量减少了，但是整个系统的吞吐率受到了压缩速率的影响，如果压缩速率慢于网络传输速率和写磁盘速率，系统的吞吐率将会降低。由图 4.4 可以看出，bzip2 这种压缩速率极慢的算法成为了整个文件的瓶颈，极大地影响了写速率，可以说整个系统几乎处于不可用状态。而对于 zlib 以及 Snappy 这两种压缩速率较高的算法，其对系统吞吐率的影响与待压缩数据类型有关，若待压缩数据压缩率较低，压缩速率也会减慢，导致文件写速率严重降低。引入分块压缩传输后，客户端能够将待写入文件分割成若干块，并并发地对数据块进行压缩，进而提高了客户端的 CPU 利用率，将部分压缩数据时间与传输数据时间重叠起来，极大地减弱了压缩算法对写速率的负面影响。与传统的边压缩边传输的方案对比，分块压缩策略使得各算法下文件系统的写文件速率都提升了 2 到 5 倍。

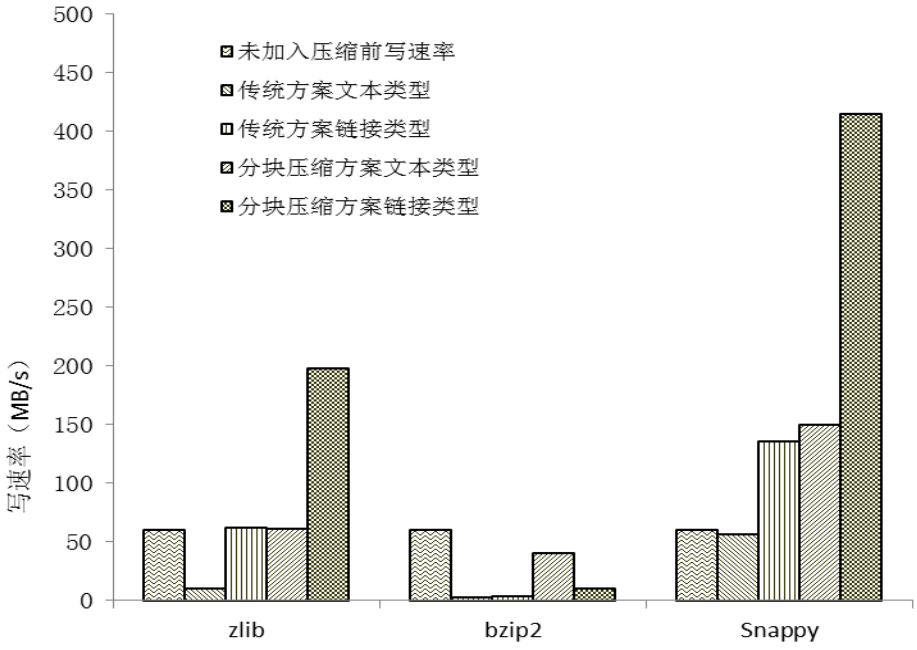


图 4.4 三种方案写速率对比

读文件阶段，三种方案测试对比结果如图 4.5 所示。其中 RaccoonFS 分布式文件系统在未采用压缩策略前的读速率大概是 100MB/s，加入压缩算法后，与写文件阶段类似，整个系统的读速率主要受到了解压缩速率的影响。然而相对于写速率，读速率受到的影响较小，其原因在于普遍各压缩算法的解压缩速率都快于压缩速率。另外传统压缩策略存在的另一个重要问题是，由于整个文件被压缩到了一起，因此不能并发地对文件进行读取，也不能仅仅读取文件中的一个数据块，这较大地降低了文件的读取效率和使用范围。由测试结果可以看出，通过对各分块数据进行并发的解压缩，可以充分克服传统压缩策略存在的上述问题，大大提高读文件的速率。与传统的边解压缩边传输的方案对比，分块解压缩策略使得各算法下文件系统的读文件速率都提升了 2 到 10 倍。其中 bzip2 这样的解压缩速率较慢的算法受到的影响最大。

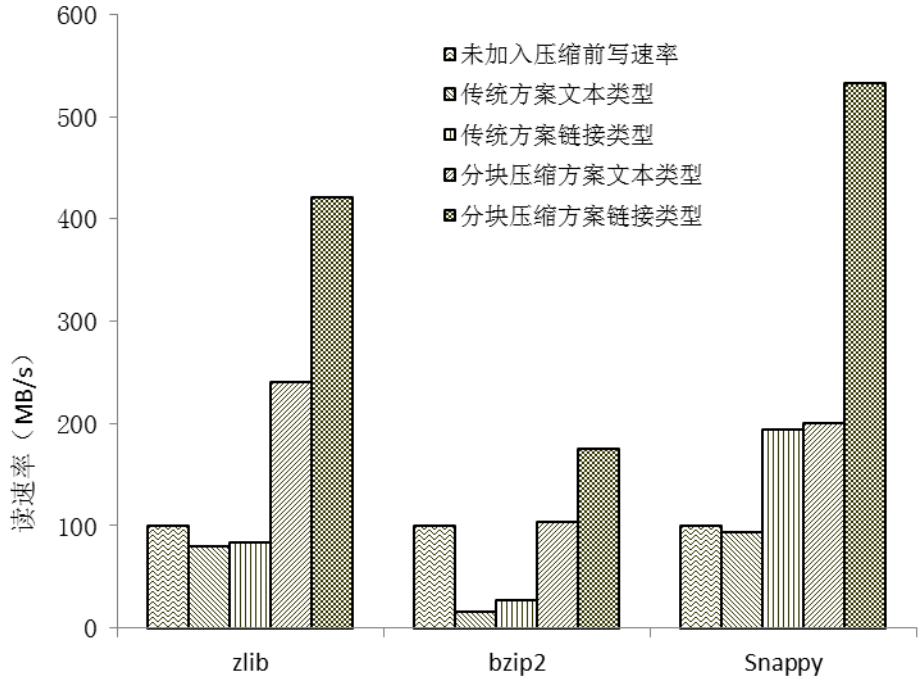


图 4.5 三种方案读速率对比

4.5 本章小结

本章主要选用了非结构化数据作为测试数据集，将压缩算法应用到 RaccoonFS 分布式文件系统中后对于不同数据集的读写速率进行了比较，测试结果表明，部分压缩算法的压缩解压缩速率，成为了读写速率的瓶颈，影响到了系统的吞吐率，此外待压缩数据类型也对系统吞吐率有较大影响。然后对分块检测模块的压缩预测准

确率进行了测试分析，测试结果表明，分块检测可以较准确的预测待压缩文件的可压缩率，与实际压缩率误差基本在 10%以内。最后重点对分块压缩策略的文件读写速率进行了测试分析，测试结果表明，分块压缩策略的吞吐率相较于传统压缩策略有了 2 到 5 倍以上的提升，在减轻带宽占用和存储空间占用的基础上，提高了系统性能。总体而言，本章内容验证了分块检测、分块压缩方案在分布式文件系统中使用的有效性。

5 全文总结

随着互联网及大数据时代下，数据信息特别是非结构化数据信息的爆炸式增长，分布式文件系统在发展中的地位越来越重要，同时分布式文件系统为管理海量数据，亟待解决的难题也越来越多。数据压缩算法通过对数据进行重新编码组织，可以达到缩减数据量的效果。将其运用到分布式文件系统中，可以通过增加部分计算量，显著减少存储空间，降低带宽消耗，提高数据传输、处理效率。然而现有的包括 Hadoop 在内的工程实现都没有考虑数据压缩算法与分布式文件系统结合所存在的问题，以及应用数据压缩技术的优化方案。本文通过调研分析现有数据压缩技术与分布式文件系统本文结合所存在的问题，提出了分块检测、分块压缩的应用方案，本文所做的主要工作具体如下：

（1）分析了当前分布式文件系统所面临的亟待解决的问题，即存储、传输成本问题以及高吞吐率需求问题，进而得出在分布式文件系统中应用数据压缩策略的重要性。接着分析了当前的分布式文件系统中应用数据压缩策略所存在的四个问题，即分布式文件系统存储的非结构化数据多样性问题；压缩算法对分布式文件系统读写性能影响问题；压缩算法压缩解压缩不对称问题；全文件压缩导致的块实际数据过大，不可分割问题。通过对上述问题的分析，提出了分块检测、分块压缩的方案，对待存储数据预先进行分块并对分块进行检测和并发压缩传输。

（3）设计并实现了 RaccoonFS 分布式文件系统中的分块检测、分块压缩方案，对方案进行了架构设计，实现了源文件分块模块、压缩算法管理模块、分块检测模块、分块压缩模块，介绍了引入分块检测、分块压缩方案后的读文件和写文件流程。

（4）采用了不同的非结构化数据集对压缩算法进行了测试，主要是测试了压缩比以及引入压缩算法后对分布式文件系统读写速度的影响。然后测试了分块检测的准确率，最后测试了分块压缩的性能。测试结果证明，分块压缩策略的吞吐率相较于传统压缩策略有了 2 到 5 倍以上的提升，在减轻带宽占用和存储空间占用的基础上，提高了系统性能。

下一步工作：

（1）进一步提高分块检测对于数据不均匀文件的压缩率预测准确率，使得分块检测方案能够更好地节省系统资源。

（2） 进一步分析研究在分布式文件系统中引入重复数据删除技术的可行性和

必要性。

（3）进一步分析测试分块压缩在高并发的实际生产环境中的性能，并提升该模块应对非常规状况的可靠性。

致 谢

转眼间，两年多的研究生生涯就快要结束。回望曾经的日子，一切仿佛都还发生在昨天，而我仍然是那个在浩瀚的知识面前懵懂无知的男孩。很庆幸能够在光电国家实验室信息存储部度过我的研究生时光，这里给我提供了很好的平台让我能够认真搞科研。更可贵的是，我的师兄师姐，我的各个老师都是非常优秀的人，他们带领我一步步走进科研的圣地，让我认识到自己的不足，也让我慢慢成长。我的心中满是感恩，正式因为有了这样的环境，这样的人文关怀，我才能够不断进步，不断地追求卓越。

首先要感谢的是实验室的领头人，也是我的导师冯丹教授，是她对于整个实验室的准确把握和掌舵，让实验室这艘拥有百多号硕博士的大船能够正常航行，劈风斩浪。在硕士期间，冯丹教授对于学生和其他老师总是严格要求，也让我们受益良多。在此向导师致以最崇高的感谢和敬意。相信冯老师必然能够带领实验室勇攀学术高峰，完成国家任务，取得更多优秀的成果。

其次需要感谢我们 F309 房间的指导老师施展老师。忘不了第一次遇见施老师的情景，那时的我还刚大三，没有做过什么项目，没有什么重点研究方向，对于真正的研究学习一无所知。施老师指点了我，让我第一次认识到什么才是真正的科研，什么才是确实在用的技术。施老师非常尊重学生的意见，启发同学们认真思考，做有意义的事情。施老师总是和睦近人，他欢乐爽朗的笑声让我们感受到了来自实验室老师的温暖。不管以后在哪里工作，施老师都是我的榜样，我会对施老师一直怀着感激和敬佩的心。

感谢实验室的李洁琼老师、王芳老师、谭支鹏老师、刘景宁老师、曾令仿老师、华宇老师、童薇老师、熊双莲老师等，是大家的辛勤付出为我们营造了一个良好的学术氛围。

感谢广域网存储组的各位学长学姐，忘不了第一次进实验室时候热心的贺燕和郑颖师姐，忘不了有个性的大牛赵恒师兄，忘不了可爱的陈云云、姚莹莹、叶为民、王敬轩。此外我还得到了刘小军、王霁欣以及柳青、李勇、李宁、焦田丰、尹祎的指导，还有同级的桂莅、王艳萍、赵千、方波以及各学弟学妹李白、黄力、韩江、郭鹏飞、张成文、李剑等的帮助，因为大家的存在，我们才是一个温暖又热闹的集体。

最后，要感谢我的家人，感谢父母无私的爱，我终于能够自力更生回报，你们回报社会了。爸爸妈妈，你们辛苦了！虽然孩子很少跟你们说我爱你，可孩子永远不会让你们失望，孩子永远是你们的骄傲！最后感谢我的女朋友小熊，三年多的时光，我们一起经历了不少风风雨雨，有了你的支持和鼓励，我才能不断向前，不断取得进步。

路漫漫其修远兮，吾将上下而求索。吾生也有涯，而知也无涯。

参考文献

[1] D. Howe, M. Costanzo, P. Fey, et al. Big data: The future of biocuration. Nature, 2008, 455(7209): 47~50

[2] Veitch A C, Riedel E, Towers S J, et al. Towards Global Storage Management and Data Placement. in: Proceedings of the 8th Workshop on Hot Topics in Operating Systems. Oberbayern, Germany, 2001. 184

[3] Gantz J, Reinsel D. Extracting value from chaos. IDC iView, 2011(11): 1~12

[4] 张建勋, 古志民, 郑超. 云计算研究进展综述. 计算机应用研究, 2010, 27(2): 429~433

[5] Gantz J F, Chute C. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC, 2008(5): 33~35

[6] Brill E, Lin J J, Banko M, et al. Data-Intensive Question Answering. in: Proceedings of the Tenth Text REtrieval Conference. Gaithersburg, Maryland, USA: TREC. 2001. 26~31

[7] Ekanayake J, Pallickara S, Fox G. MapReduce for data intensive scientific analyses. in: Proceedings of the eScience 2008. IEEE Fourth International Conference. Indianapolis, Indiana, USA. 2008. 277~284

[8] Hey A J, Tansley S, Tolle K M. The fourth paradigm: data-intensive scientific discovery. 2009. 14~16

[9] Patel S M, Mikesell P A, Schack D P. Systems and methods for providing a distributed file system utilizing metadata to track information about data stored throughout the system. U.S. Patent 7,743,033. 2010. 10~13

[10] Raicu I, Foster I T, Zhao Y, et al. The quest for scalable support of data-intensive workloads in distributed systems. in: Proceedings of the 18th ACM international symposium on High performance distributed computing.Garching near Munich, Germany:ACM, 2009. 207~216

[11] Salomon D. Data Compression: The Complete Reference. Springer, 2004. 5(4): 54~57

[12] Sayood K. Introduction to data compression. Access Online via Elsevier, 2012. 63~67

[13] Ghemawat S, Gobioff H, Leung S. The Google file system. in: Proceedings of the nineteenth ACM symposium on Operating systems principles. Bolton Landing, NY, USA: ACM, 2003. 29~43

[14] Shvachko K, Kuang H, Radia S, et al. The Hadoop Distributed File System. in: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies. Lake Tahoe, Nevada, USA: IEEE Computer Society, 2010. 1~10

[15] Schwan P. Lustre: Building a file system for 1000-node clusters. in: Proceedings of the 2003 Linux Symposium. Ottawa, Ontario, Canada. 2003. 111~113

[16] Kulkarni K, Ren K, Patil S, et al. Giga+ TableFS on PanFS: Scaling Metadata Performance on Cluster File Systems. Carnegie Mellon University Parallel Data Lab Technical Report. 2013, 13(101): 54~55

[17] Zhao D, Burlingame K, Debains C, et al. Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms. in: Preceedings of the IEEE CLUSTER 2013. Indianapolis, IN. 2013. 13~18

[18] Thusoo A, Shao Z, Anthony S, et al. Data warehousing and analytics infrastructure at facebook. in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. New York, USA:ACM, 2010. 1013~1020

[19] Aguilera M K, Janakiraman R, Xu L. Using erasure codes efficiently for storage in a distributed system. in Proceedings of Dependable Systems and Networks. International Conference on. IEEE. Yokohama, Japan. 2005. 336~345

[20] Chen P M, Lee E K, Gibson G A, et al. RAID: High-performance, reliable secondary storage. ACM Computing Surveys, 1994, 26(2): 145~185

[21] Weatherspoon H, Kubiatowicz J D, Erasure coding vs. replication: A quantitative comparison. In Peer-to-Peer Systems, Springer, 2002. 328~337

[22] Shum K W. Cooperative regenerating codes for distributed storagde systems.in: Proceedings of the 3rd 2011 IEEE International Conference on Comunications. IEEE.Kyoto, Japan. 2011. 1~5

[23] Khan O, Burns R, Plank J, et al. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. in: Proceedings of the 10th USENIX Conference on File and Storage Technologies. San Jose, CA. 2012. 43~44

[24] Sayood K. Introduction to data compression. Access Online via Elsevier, 2012. 43-46

[25] Huffman D A. A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 1952, 40(9): 1098~1101

[26] Hankerson D R, Harris G A, Johnson P P D. Introduction to information theory and data compression. CRC press, 2003, 10(13): 54~56

[27] Shannon C E. A mathematical theory of communication. ACM SIGMOBILE Mobile Computing and Communications Review, 2001, 5(1): 31~55

[28] McKnight J, Asaro T, Babineau B. Digital Archiving: End-User Survey and Market Forecast 2006–2010. The Enterprise Strategy Group, 2006. 67~68

[29] Tolia N, Kaminsky M, Andersen D G, et al. An Architecture for Internet Data Transfer. NSDI. 2006.76~77

[30] Arnold R, Bell T. A corpus for the evaluation of lossless compression algorithms. in: Proceedings of the 7th Data Compression Conference, Snowbird, Utah. 1997. 201~210

[31] R. E. Bryant, R. H. Katz, E. D. Lazowska. Big-data computing: Creating revolutionary breakthroughs in commerce, science, and society. In Computing Research Initiatives for the 21st Century. Computing Research Association, 2008. 351~369

[32] Ziv J, Lempel A. A universal algorithm for sequential data compression. Information Theory, IEEE Transactions on, 1977, 23(3): 337~343

[33] Ziv J, Lempel A. Compression of individual sequences via variable-rate coding. Information Theory, IEEE Transactions on, 1978, 24(5): 530~536

[34] Nelson M R. LZW data compression. Dr. Dobb's Journal, 1989, 14(10): 29~36

[35] Salomon D. Data Compression.: The Complete Reference. Springer, 2004. 143-146

[36] Al-Bastaki Y A L. GIS image compression and restoration: A neural network approach. Information Technology Journal, 2006, 5(1): 88~93

[37] Biggar H. Experiencing data de-duplication: Improving efficiency and reducing capacity requirements. The Enterprise Strategy Group, 2007. 110~118

[38] P. Buneman, S. Davidson, G. Hillebrand, et al. A query language and optimization techniques for unstructured data. in: Proceedings of ACM International Conference on Management of Data. 1996. 505~516

[39] Larrauri J I. A new algorithm for lossless compression applied to two-dimensional static images. in: Proceedings of the 6th international conference on Communications and Information Technology, and Proceedings of the 3rd World conference on Education and Educational Technologies. Corfu Island , Greece. 2012. 56~60

[40] Shahi G, Singh C. To Secure and Compress the Message on Local Area Network. International Journal of Computer Applications, 2013. 69(17): 23~30

[41] Chandra A, Chakrabarty K. System-on-a-chip test-data compression and decompression architectures based on Golomb codes. Computer-Aided Design of Integrated Circuits and Systems, 2001, 20(3): 355~368

[42] Govinda K, Kumar Y. Storage Optimization in Cloud Environment using Compression Algorithm. International Journal, 2012, 1(1): 333~336

[43] Muthukumar M, Ravichandran T. Analyzing compression performance for real time database systems. International Conference on Advanced Computer Engineering and Applications. 2012. 221~223

[44] Welton B, Kimpe D, Cope J, et al. Improving i/o forwarding throughput with data compression. Cluster Computing, 2011. 438~445

[45] Ruan G, Zhang H, Plale B. Exploiting MapReduce and data compression for data-intensive applications. in: Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery. San Diego, CA, USA:ACM, 2013. 38

[46] Harnik D, Kat R, Margalit O, et al. To Zip or not to Zip: Effective Resource Usage for Real-Time Compression. in: Proceedings of the 11th USENIX Conference on File and Storage Technologies.San Jose, CA. 2013.

[47] Chen Y, Ganapathi A, Katz R H. To compress or not to compress-compute vs. IO tradeoffs for mapreduce energy efficiency. in: Proceedings of the first ACM SIGCOMM workshop on Green networking. New Delhi, India: ACM, 2010. 23~28

[48] Pankratius V, Jannesari A, Tichy W F. Parallelizing bzip2: A case study in multicore software engineering. Software, IEEE, 2009, 26(6): 70~77

[49] Held G, Marshall T. Data Compression.Techniques and Applications. Lifetime Learning Publications, 1991, 50(1): 248~258

[50] Raicu I, Foster I T, Zhao Y, et al. The quest for scalable support of data-intensive workloads in distributed systems. in: Proceedings of the 18th ACM international symposium on High performance distributed computing. Garching, Germany: ACM, 2009. 207~216

[51] Deutsch L P. DEFLATE compressed data format specification version 1.3. 1996. 232~237

[52] Sun Z, Shen J, Yong J. A novel approach to data deduplication over the engineering-oriented cloud systems. Integrated Computer-Aided Engineering, 2013, 20(1): 45~57

[53] Nelson M. Data compression with the Burrows-Wheeler transform. Dr. Dobb’s Journal, 1996. 5(23): 39~43