基于ARM的硬件压缩算法在Spark中的性能研究

朱常鹏[1)] 汤景仁[2)] 梁 昀[2)] 张小川[1)] 韩 博[3)] 赵银亮[4)]

[1)](重庆理工大学数据科学与大数据系 重庆 401135)

[2)](华为科技有限公司 广东 深圳 518100)

[3)](西安交通大学网络空间安全学院 西安 710049)

[4)](西安交通大学计算机科学与技术学院 西安 710049)

摘 要 鲲鹏920 CPU是2021年面世、全球第一款基于7纳米制造工艺的ARM 64位CPU,该CPU内置一个名为KAEzip的硬件加速引擎,其核心是一个硬件压缩算法,能通过硬件提升压缩与解压缩性能.相关研究表明,压缩算法的硬化与传统软件压缩算法相比具备明显性能优势.但大数据领域中的基础性系统软件都无法识别和使用这类算法.因此研究评估硬件压缩算法在大数据环境下的性能,发现揭示制约这类算法性能的关键因素以及可能存在的缺陷具有重要意义.为此,本文首先提出一种基于"生产-消费"模型的Spark任务性能模型,形式化地表示多维资源、压缩算法和Spark任务性能之间的内在关系,从理论上分析揭示出Spark下影响压缩算法性能的关键因素.然后提出一种三层架构支持Spark识别使用硬件压缩算法.这种分层架构为进一步调优硬件压缩算法在Spark中的性能提供了灵活性,也能复用到其他大数据系统软件.在此基础上本文以KAEzip为实验对象,使用经典Spark基准测试程序全面评估它在Spark中的性能,结合性能模型分析挖掘制约KAEzip性能的关键因素与根源.对KAEzip的测试表明:(1)硬件压缩算法可有效提升Spark性能。比如,KAEzip比snappy有最多13.8%的压缩性能优势、最多7%的解压优势和最多5.7%的实际应用场景下的性能优势;(2)磁盘的数据传输率与硬件压缩算法性能之间的不匹配是制约硬件压缩算法性能的重要因素;(3)压缩算法在Spark中的运行机制更易导致CPU的数据处理能力与硬件压缩算法性能不匹配,也制约着硬件压缩算法的性能.测试结果也表明KAEzip在压缩小数据时会导致数据膨胀问题.为此,本文扩展三层架构分析揭示出导致该问题的根源,并结合压缩算法在Spark中的运行机制提出一种优化方法.硬件压缩算法作为压缩算法领域的新研究方向,本文的研究工作不仅可广泛用于优化内置于CPU中的硬件压缩算法在Spark下的性能,也有助于持续演化完善KAEzip和鲲鹏920 CPU.

关键词 鲲鹏920 CPU;KAEzip;大数据;Spark;硬件压缩算法;根源分析

中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2023.02626

A Performance Study of ARM-Based Hardware-Based Compression Algorithms in Spark

ZHU Chang-Peng[1)] TANG Jing-Ren[2)] LIANG Yun[2)] ZHANG Xiao-Chuan[1)] Han Bo[3)] Zhao Yin-Liang[4)]

[1)](*Department of Data Science and Big Data, Chongqing University of Technology, Chongqing* 401135)

[2)](*Huawei Technologies Co., Ltd, Shenzhen, Guangdong* 518100)

[3)](*School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an* 710049)

[4)](*School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an* 710049)

收稿日期:2022-11-16;在线发布日期:2023-07-17.本课题得到鲲鹏众智计划中的Spark使能KAE压缩项目(OAA21091100464724D)、国家留学基金委员会(201708505099)、国家自然科学基金(61702063)资助.**朱常鹏**(通信作者),博士,讲师,中国计算机学会(CCF)会员,主要研究领域为大数据处理和虚拟机.E-mail:is99zcp@cqut.edu.cn.**汤景仁**,硕士,主要研究领域为硬件加速引擎和高性能计算.**梁 昀**,学士,主要研究领域为硬件加速引擎和高性能计算.**张小川**,硕士,教授,主要研究领域为计算机博弈和软件工程.**韩 博**,博士,高级工程师,主要研究领域为信息处理和软件工程.**赵银亮**,博士,教授,主要研究领域高性能计算、并行计算和编程语言设计.https://gitee.com/kunpeng_compute/sparkforkae提供了本文算法的实现代码,实验环境搭建与测试的脚本和文档说明.

Abstract As the first 7-nm ARM-based 64-bit multi-core CPU, Kunpeng 920 stands out as one of the notable landmarks. The CPU is equipped with a hardware accelerator for compression, i. e. KAEzip. Recent research efforts indicate that hardware-based compression algorithms have significant performance advantages over traditional ones. However, the most of the crucial foundational software in big data fields, for example, Hadoop and Spark, cannot recognize and leverage them. Therefore, how such algorithms perform in big data environment remains to be an open issue. To address this problem, this paper first proposes a "producer-customer"-based model for Spark tasks to formally describe the relationships among compression, multi-dimension hardware resources, and execution times of Spark tasks, and then extract what crucial factors have influence on compression in Spark. Afterwards, a three-layer architecture on top of Spark is proposed to enable Spark to use hardware-based compression algorithms. The layered architecture is quite easy to be extended for optimizing their performance in Spark and is conveniently reused in other big data software.

Based on the model and a series of experiments, we evaluate the performance of KAEzip in Spark and then reveal root causes, which have a vital influence on KAEzip. Our experiments indicate that hardware-based compression algorithms can bring significant performance upgrades, for example, KAEzip outperforms snappy by up to 13.8%, 7% and 5.7% in compression, decompression and LDA, respectively, and that data transmission speed of disks and data process capability of CPU have a decisive influence on the performance of compression algorithms in Spark, to some extent. These experiments furthermore demonstrate that KAEzip may cause data inflation problem. Finally, this paper presents an optimization on top of the architecture to reveal the root cause of the problem and then resolve it by exploiting the working mechanism of compression algorithms in Spark. This research work not only promotes the optimization and evolution of KAEzip and Kunpeng CPU but also benefits hardware-based compression algorithms in other CPUs in the future, to achieve better performance and lower overhead.

Keywords Kunpeng 920; KAEzip, big data; Spark; hardware-based compression algorithms; root cause analysis

1 引 言

近年作为全球第一款基于7-nm制程的ARM 64位多核SoC CPU，华为鲲鹏920 CPU[1]内置一个加速引擎(Kunpeng Accelerator Engine，KAE)，提供一种基于硬件的压缩算法，简称为KAEzip. 与传统的zlib压缩算法相比，该算法在压缩与解压缩方面都具有明显性能优势. 相关研究[2-3]也表明压缩算法的硬化较传统软件压缩具有明显性能优势. 但是硬件压缩算法作为一种新的压缩方式，目前大数据领域中的主流基础性系统软件均不支持，比如Spark[4]. 这导致这类算法无法有效地用于大数据的存储、计算与分析.

Spark大数据处理框架已被广泛地应用于大数据领域，成为该领域中的重要基础性系统软件之一. 它支持主流压缩算法对不同场景下的数据进行压缩与解压缩，以提升数据的处理性能. 比如，对Shuffle的输出数据进行压缩和解压缩. 由于缺少对硬件压缩算法的支持，Spark应用程序无法有效地利用这类算法提升执行性能，阻碍了评估这类算法在Spark环境下的性能优势以及揭示制约它们性能的关键因素与根源，也妨碍对KAEzip与鲲鹏920 CPU在Spark环境下进行有效评估与改进.

硬件压缩算法[2-3,5]的共性是能够极大提升压缩与解压缩性能，但在大数据环境下的实际性能仍鲜有研究，其主要原因是它们还未融入到主流的CPU中，比如Intel/AMD x86 CPU或ARM CPU. 这使得无法构建相应大数据实验环境对其进行性能评估. KAEzip作为第一种内置于ARM CPU中可实用的硬件压缩算法，本文以它为实验对象研究硬件压缩算法在Spark下的实际性能.

首先，根据压缩算法在Spark环境下的运行机制，基于“生产-消费”模型提出一种Spark任务的性能模型，形式化描述任务的执行性能、多维硬件资源利用率和压缩算法之间的内在联系，分析抽取出Spark环境下影响压缩算法性能的关键因素．该模型独立于硬件环境，具有较好普遍性，可用于分析评估任何压缩算法在Spark环境下的性能，比如软件压缩算法snappy[6]和硬件压缩算法KAEzip.

其次，由于目前软件硬化技术普遍采用软硬件分离设计：软件的常用功能硬化在CPU中，额外的驱动程序对外提供统一访问接口．针对这种设计，本文提出一种三层架构支持Spark识别使用硬件压缩算法．该架构的第一层实现对Spark输入输出流的读写操作，第二层负责数据流的预处理与优化，第三层则主要以Java本地接口(Java native interface，JNI)方式与硬化算法的驱动进行交互，实现数据的压缩或解压缩．通过该三层架构，Spark能正确识别和利用内置于CPU中的硬件压缩算法．得益于软硬件分离设计，本文提出的三层架构不仅可以支持鲲鹏920 CPU中的KAEzip，也能适用于其他CPU中的硬件压缩算法．

然后，融入三层架构到Spark，再结合Spark任务性能模型和经典Spark基准测试程序，采用广泛的实验测试评估硬件压缩算法对Spark应用程序性能的影响，分析揭示出Spark环境下制约这类算法性能的根源及其可能存在的缺陷．基于KAEzip的实验测试表明：硬件压缩算法能有效提升Spark应用程序性能但仍受多种硬件资源瓶颈约束，其根源是磁盘数据传输率或CPU的数据处理速度与硬件压缩算法的压缩或解压缩速度不匹配．这为进一步优化这类算法指明了方向．

此外，实验也指出KAEzip在压缩小数据时会引发数据膨胀问题．为此，本文扩展三层架构剖析压缩算法的执行过程，采集分析若干反映压缩性能的关键指标，进而揭示出导致数据膨胀问题的根源．在此基础上，根据压缩算法在Spark下的运行机制提出一种优化方法．它通过引入缓存机制将被压缩的数据拷贝至缓冲区，并在缓冲区被填充满时才对数据进行压缩．实验表明这种延迟压缩策略不仅能有效地解决KAEzip的数据膨胀问题，也能提升软压缩算法在Spark下的性能，比如zlib压缩算法．

结合目前的参考文献，本文是第一篇理论结合实验分析研究Spark环境下硬件压缩算法性能的论文，其研究工作不仅可用于优化硬件压缩算法在Spark下的性能，也有助于持续演化完善KAEzip和鲲鹏920 CPU，其贡献可以总结为：

(1)首先，提出一种Spark任务的性能模型，形式化地表示压缩算法、多维硬件资源利用率和Spark任务性能之间的内在联系，分析不同资源利用率和压缩算法对任务性能的影响，理论上揭示出影响Spark环境下压缩算法性能的根源与关键因素．该性能模型可广泛用于分析压缩算法在Spark环境下的性能．

(2)其次，针对软件硬化技术广泛采用软硬件分离设计的特点，提出一种扩展Spark支持硬件压缩算法的三层架构，为下一步研究评估该类算法在Spark环境下的性能奠定基础．这种基于分层的架构不仅为调优硬件压缩算法在Spark中的性能提供了灵活性，也能有效复用到其他大数据基础软件，比如Hadoop.

(3)然后，以KAEzip为实验对象，通过广泛的测试评估它在Spark环境下的性能，再结合本文提出的性能模型，首次揭示出在Spark环境下制约硬件压缩算法性能的关键因素与根源，并给出相应的优化策略，为进一步优化该类算法奠定基础．

(4)最后，结合三层架构提出一种基于Spark的优化方法．它不仅能提升软压缩算法的性能，而且可有效解决KAEzip在压缩时引发的数据膨胀问题，为KAEzip和鲲鹏920 CPU的持续迭代完善奠定了基础．受益于软硬分离设计，这种基于三层架构的优化方法也能广泛用于其他硬件压缩算法．

本文第2章展示背景与动机；第3章介绍相关工作；第4章展示一种面向压缩算法的Spark任务的性能模型；第5章首先提出一种扩展Spark支持硬件压缩算法的三层架构，然后描述KAEzip面临的数据膨胀问题，在此基础上提出一种基于三层架构的压缩与解压缩优化方法，最后讨论该架构的复用性；第6章是实验与评估，第7章总结全文．

2 背景和动机

压缩软件的硬化可有效提升压缩与解压缩的性能，但为了提升该类硬化算法的应用范围，其普遍采用软硬件分离设计，即核心的压缩与解压缩功能硬化在CPU内，再辅以驱动程序提供统一的对外接口．比如，KAEzip是一种内置于鲲鹏920 CPU的硬件压缩引擎，其软硬件分离结构如图1所示：它的驱动程序*libkaezip*动态库位于操作系统层，应用程序

通过访问该库可以触发KAEzip中的硬件压缩与解压缩功能.

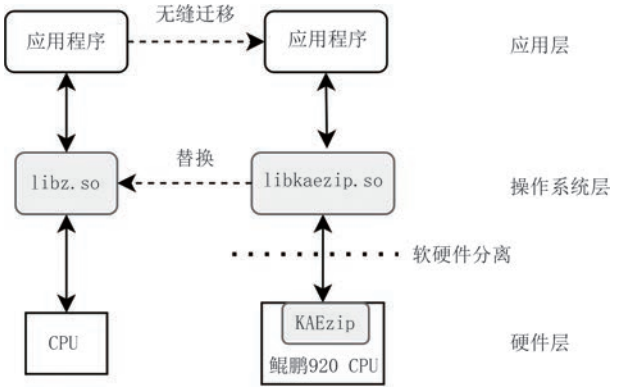

图1　KAEzip的软硬件分离设计

zlib[7]是一款免费、通用且法律上不受阻碍(即未被任何专利覆盖)的无损压缩解压缩软件库,被广泛地应用于数据的压缩与解压缩.为了进一步地提升KAEzip的应用范围,*libkaezip*完全兼容zlib的接口.因此通过替换*libz*动态库①就能使基于zlib的应用程序无缝运行在鲲鹏920 CPU之上,更加便于构建良好的应用生态.

lzbench[8]是一款基于内存的基准测试程序,广泛用于压缩算法的性能测试.如图2所示,基于lzbench的测试结果[9]表明:与zlib相比,KAEzip可提升压缩性能约43倍、解压缩性能约7倍.另一方面,压缩比也是衡量压缩性能的另一个指标,可定义为:

$$compression\ ratio = \frac{compressed\ data\ size}{data\ size}$$

其中压缩比越低越好.图2表明KAEzip的压缩比比zlib的压缩比高大约9%.

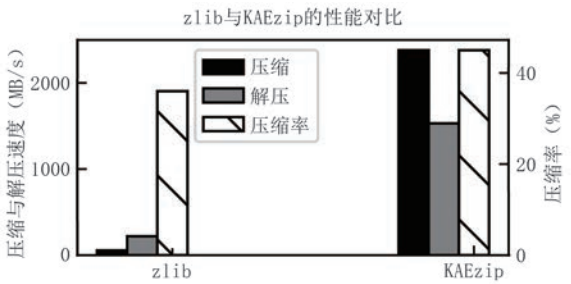

图2　zlib与KAEzip的性能比较

基于KAEzip的实验表明压缩算法的硬化在压缩与解压缩方面能取得巨大性能优势.如何使大数据领域中的重要基础性系统软件识别并利用硬化压缩算法,评估该类算法在大数据环境下的性能优势与缺陷,揭示出制约其性能和导致其缺陷的关键因素与根源,具有重要现实意义.

3　相关工作

鲲鹏920 CPU是全球第一款基于7纳米制造工艺的ARM 64位CPU,近来对它的性能研究已经成为一个新的研究热点.

文献[1]详细地介绍了鲲鹏920 CPU的架构设计以及它的高带宽与低延迟优势.基于HQL-SQL的评估表明这种架构设计使得鲲鹏920 CPU较Intel 6148/6248有大约20%的性能优势.文献[10]使用TeaLeaf、SNAP、CloverLeaf和GTC-P四种经典的CPU基准测试程序深入评估了鲲鹏916/920 CPU的算术浮点和内存子系统的性能.实验结果表明鲲鹏920 CPU在低精确度的科学计算方面较Intel 6148具有明显的性能优势,这主要得益于混合进程技术和高带宽超低延迟的内存性能,但在双精度科学计算能力仍然不及Intel 6148. 本文与文献[1,10]类似之处是:均在单节点环境下测试评估鲲鹏920 CPU的性能,以减少网络对测试结果的影响.但最大的不同点在于:本文主要测试评估鲲鹏920 CPU中的KAEzip在Spark环境下的压缩性能,而它们主要是测试该CPU的计算性能.

已有的研究表明压缩算法对大数据环境下的应用程序性能具有重要影响.

文献[11]测试评估了压缩算法对Hive和Spark SQL在查询两种列式文件——ORC和Parquet——的性能影响.其测试结果表明块大小、页面大小和压缩算法对性能都具有重要影响,并且指出在相同配置下zlib能提升最多60%的ORC文件的查询性能,snappy则可以提升大约7%的Parquet文件的查询性能.文献[12]的研究表明压缩算法snappy较bzip2对MapReduce和Tez应用程序的性能提升更大.它们与本文的研究内容存在共同之处,即使用经典基准测试程序与数据集评估压缩算法在大数据环境下的性能.但最大的不同是:它们的研究对象是基于软件的压缩算法,而本文的研究对象是硬化压缩算法,并且理论结合实验地分析抽取出Spark环境下影响这类算法的关键因素:磁盘的数据传输率、CPU单位数据处理能力和压缩比.这为进一步迭代优化KAEzip和其他硬化压缩算法指明方向.

① libz动态库是zlib的具体实现

传统压缩算法主要专注如何优化重复数据的发现，以提升压缩或解压缩性能，比如lz4[13]和snappy。但是硬件压缩算法的重心是如何充分利用硬件特性优化压缩与解压缩的执行过程，已经成为压缩领域中的一个新研究方向.

文献[2]展示了一种基于FPGA芯片的自适应数据压缩系统用于列式文件的压缩，其主芯片是Intel i7 CPU. 实验结果表明引入硬件压缩可以最高提升压缩性能大约6倍. 文献[5]展示了一种lz4压缩算法的硬件实现. 这种基于FPGA的实现揭示出lz4的硬软实现之间存在很多差异，其中数据供给率是影响硬压缩性能的瓶颈之一，这为优化硬件设计指明了方向. 文献[3]提出了一种基于RISC-V的snappy压缩算法实现，并且指出这种软件硬化方式可以最多提升100倍的性能. KAEzip与上述硬化算法的区别在于KAEzip基于ARM CPU. x86、RISC-V和ARM这些芯片架构上的差异性使得无法构建统一实验平台比较KAEzip与它们在Spark下的性能差异. 此外，本文的核心是研究评估KAEzip在Spark下的性能，并且揭示出硬件压缩算法对Spark应用程序性能的提升幅度远低于预期值的根源. 文献[14]提出一种基于GPU的压缩算法，通过充分利用GPU的并行计算能力与高带宽，增加压缩与解压缩的并行度以提升性能. 文献[15]综述了基于硬件增强的信息安全.

对Spark应用程序的性能评估以及影响它们性能的根源分析已经成为一个研究热点.

文献[16-18]评估了经典Spark应用程序在不同运行环境下的性能. 比如，文献[16]指出与传统的虚拟机相比，容器能明显地提升Spark应用程序的性能，其根源是容器具有更小的CPU开销和内存开销. 文献[17]指出AUFS文件系统会导致容器在运行时产生额外IO操作，进而影响Spark应用程序的性能. 文献[18]指出Kubernetes中的网络名字空间会恶化Spark应用程序的数据局部性，从而降低其性能. 文献[19]指出Spark默认调度策略并不能产生一个全局最优的任务调度，进而将调度问题等价转换为一个图匹配问题，并使用Kuhn-Munkres算法求取出最优调度. 本文使用类似的Spark基准测试程序评估压缩算法在Spark中的性能，但最大的不同是本文专注研究KAEzip这种硬件压缩算法对Spark应用程序的性能，而忽略资源分配使用、IO操作和任务调度等其他因素.

综合分析多维资源利用率和应用程序内部执行逻辑之间的内在联系是揭示出影响应用程序性能根源的有效策略.

文献[20-21]提出了一个名为ELBA的根源检测系统，可有效定位检测出导致n层web系统发生微瓶颈的根源. 其主要原理是在结合n层web系统的内部执行流程基础上，通过分析CPU、内存和磁盘等多维资源利用率的突变来挖掘出引发微瓶颈的根源. 文献[22]利用ELBA发现部署在Kubernetes上的n层web系统存在的性能问题，并确定导致该问题的根源为工作负载不均衡，进而提出一种基于pods的动态伸缩机制用于解决该问题. 文献[23]提出一种方法用于确定导致Spark异常任务的根源. 该方法的基本思想也是在结合Spark任务内部执行流程的基础上，通过分析CPU、GC、磁盘和网络四种资源利用率，确定导致异常任务发生的根源. 与这些文献类似，本文也采用类似方法分析确定影响KAEzip在Spark环境下的压缩性能的关键因素.

文献[24]提出了一种性能预测模型用于预测Spark应用程序的执行时间. 该模型假设阶段是顺序执行. 文献[18]指出Spark应用程序中的阶段也可并发执行，并在此基础上提出一种性能模型用于指导分析多维资源对Spark性能的影响. 与它们不同的是：本文提出的Spark性能模型是基于“生产-消费”模型，主要用于形式化地描述多维资源，压缩算法和Spark任务性能之间的内在关系，进而理论上分析确定影响Spark环境下压缩算法性能的关键因素，为进一步迭代优化KAEzip与硬件压缩算法奠定基础.

4 压缩算法在Spark环境下的运行机制与性能模型

本节主要描述压缩算法在Spark环境下的运行机制，然后形式化描述压缩与Spark应用程序性能之间的关系，进而揭示出在Spark环境下影响压缩性能的关键因素，为进一步支持Spark识别、使用和优化新的软/硬压缩算法奠定基础.

4.1 压缩算法在Spark环境下的运行机制与缺陷

在Spark应用程序的执行中主要有三个过程涉及压缩与解压缩：变量的广播(broadcast)、RDD的检测点(checkpoint)和Shuffle的读写. 前两个过程与传统压缩和解压缩过程类似，即直接对数据进行压缩与解压缩，本文不做相关描述. 本文重点描述Shuffle读写中的压缩与解压缩机制以及可能存在的缺陷.

(1)Shuffle写中的数据压缩与缺陷

Spark采用典型的流模式处理数据，在Map任务将Shuffle数据最终写入到文件之前，源数据一般需要经过四类处理操作，具体如图3(左)所示．首先数据处理流使用Spark应用程序中的业务逻辑处理数据，然后使用文件流存储到指定的Shuffle文件．当处理后的数据属于多个分区时，为了提高Shuffle写的并发性能，不同分区的数据会对应不同的文件流．Spark通常使用“键值对”存储处理结果，因此在处理完一个对之后就会调用文件流中的*写操作*存储数据．此外，处理后的数据一般为对象类型，因此文件流增加一个*序列化操作*以便对流中数据依次序列化．如果Spark应用程序中设置了压缩算法，则文件流会再增加一个*压缩操作*，压缩已序列化的数据并存入到指定的Shuffle文件．

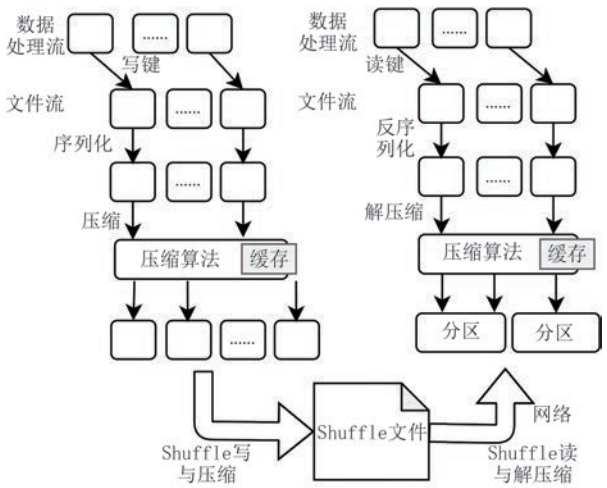

图3　Shuffle中的数据压缩与解压缩机制

Spark只有在处理完一个键值对之后才会产生两次压缩调用，这与传统压缩过程存在较大差异．因此除了磁盘数据传输率和压缩算法的压缩性能两个关键因素之外，还存在其他因素影响压缩算法在Spark环境下的性能．首先，当处理数据的速度低于压缩速度时，压缩过程会被中断以等待下一批数据抵达．这种情况下，处理数据的速度往往决定了最终的压缩性能．其次，当输入数据量较大时，Spark的流模式处理机制会导致调用过于频繁，这会产生额外开销．因此，数据处理速度和压缩调用次数是另外两个影响Spark环境下压缩性能的关键因素．

图4(a)展示了Spark sort的Map任务在执行中的压缩调用统计，主要包括调用的起止时间和执行时间．可以看出一个Map任务的输出数据分别属于四个分区，因此有四个文件流并发执行数据的压缩

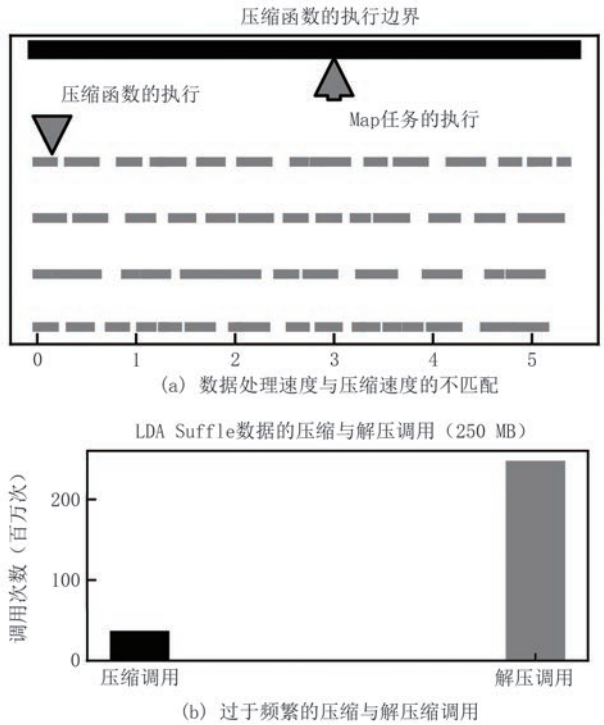

图4　Spark环境下的压缩与解压缩调用统计

操作，并且压缩过程存在较为频繁的中断现象．其根源是数据处理速度小于压缩速度，压缩过程必须等待下一批压缩数据的抵达．

图4(b)展示了Spark LDA在执行过程中的压缩调用统计．LDA在执行过程中总共产生了大约250MB中间数据，但将它们写入到Shuffle文件总共触发了大约36 931 317次压缩调用，平均一次压缩大约7.2个字节．这些统计结果表明Spark环境下的压缩过程与传统的压缩过程存在巨大差异．Spark特有的执行机制中蕴含着影响压缩性能的关键因素，其中最主要的就是数据处理速度与频繁的压缩调用．

(2)Shuffle读中的数据解压缩与缺陷

Shuffle读的过程如图3(右)所示．当Reduce任务开始执行时，它的第一个操作就是通过网络从其它主机上并发地获取Shuffle文件，然后将同一分区的数据以文件流的方式传递给数据处理流．由于Spark通常使用“键值对”存储处理结果，数据处理流依次使用读键和读值两个操作从文件流中读取数据，并使用Spark应用程序中的业务逻辑处理它们．Shuffle文件的数据是通过网络获取，因此在读取键和值之后必须对其反序列化以获取真实数据．若Shuffle文件在写时被压缩了，在反序列化之前还需要对数据先进行解压缩，这导致一次读取就会触发一次解压调用．

Shuffle读中的解压缩运行机制表明尽管数据处理速度不再是制约压缩性能的一个关键因素，但是当

Shuffle文件较大时，频繁解压调用仍然会影响到压缩算法的解压性能. 比如图4(b)展示了Spark LDA在执行过程中的解压缩调用统计：解压缩250 MB的Shuffle数据共触发了247 851 186次解压调用.

4.2 Spark环境下的压缩性能模型

任务是Spark中的最小执行单位，其执行流程如图5所示. 任务从数据源读取输入数据然后进行数据处理并最终将输出数据写入到指定文件. 任务的数据处理主要包含三个部分：数据解压缩、数据的业务逻辑处理和数据压缩. 缓冲区被广泛用于数据的读取和写入以提升IO性能，因此整个任务的执行过程可以简化为两个相互关联的“生产- 消费”过程，其中任务的执行既是第一个过程中的消费者也是第二个过程中的生产者.

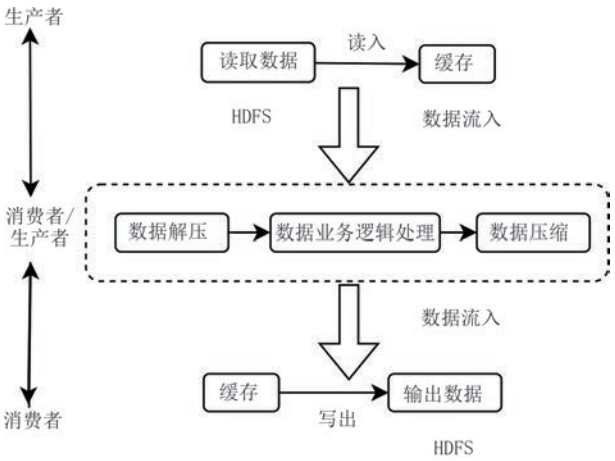

图5 Spark应用程序的任务执行模型

根据任务的执行流程，本节提出一种基于“生产-消费”模式的Spark任务的性能模型. 不同于其他Spark性能模型[18,24]，该模型并不是用于预测Spark应用程序的执行时间，而是揭示压缩算法与该性能之间的内在联系，以分析确定导致它们性能差异的根源.

为了便于描述，假设一个Spark应用程序包含n个阶段，则它可形式化地表示为：

$$App=\{S_1,\cdots,S_n\}$$

假设阶段S_i包含m个任务并同时运行在k个执行器上，则任务的平均执行时间$\overline{T}_{S_i}^{k}$可以表示为：

$$\overline{T}_{S_i}^{k}=\frac{\sum_{i=1}^{m}T_{t_i}}{m} \tag{1}$$

其中T_{t_i}表示任务t_i的执行时间.

该程序的所有任务的平均执行时间可以表示为：

$$\overline{T}_{App}=\frac{\sum_{i=1}^{n}\overline{T}_{S_i}^{k}}{n}$$

由于不同阶段的执行时间往往相差较大，执行时间较小的阶段可以忽略. 假设n个阶段中的p个阶段具有相对较长的执行时间，则上述公式可简化为：

$$\overline{T}_{App}=\frac{\sum_{i=1}^{p}\overline{T}_{S_i}^{k}}{p} \tag{2}$$

不失一般性，假设任务t包含读数据、数据处理和写数据三个部分，则它的执行时间T_t则可以表示为：

$$T_t=T_r+T_p+T_w$$

其中T_r、T_p和T_w分别表示数据的读取、处理和写入时间.

如图5所示，Spark内部采用基于缓存区的数据传输处理，因此上述三个部分实际上是并发执行. 比如，假设一个任务直接从HDFS上读取原始数据，然后将处理的结果写入到HDFS中. 由于缓冲区的存在，数据总是先被写入到缓冲区然后被传输给压缩算法进行处理，最终结果优先被缓存，待缓冲区填满时再一并写入到磁盘. 这种执行机制完全符合“生产-消费”模型，并且任务的整个执行过程可以用两个“生产-消费”模型表示：读取数据和数据处理构成第一个模型，数据处理和数据写入构成第二个模型. 根据“生产-消费"的特点，整体性能等于生产者性能和消费者性能中的最低性能. 因此根据这两个模型，分别有：

$$T_1=max\{\frac{data\_size}{P_{disk}^{r}},\frac{data\_size}{DPR}\}$$

$$T_2=max\{\frac{data\_size}{DPR},\frac{data\_size'}{P_{disk}^{w}}\}$$

其中data_size表示任务的输入数据大小，其缺省值为128MB，data_size′表示任务执行完之后的输出数据大小，P_{disk}^{r}和P_{disk}^{w}分别表示磁盘的读写性能.

本文使用数据处理率(Data Process Rate，DPR)[25]表示系统处理数据的性能. 由于数据处理既是第一个模型的消费者又是第二模型的生产者，因此任务的执行时间可以表示为：

$$\begin{aligned}T_t=&max\{T_1,T_2\},=\\&max\{max\{\frac{data\_size}{P_{disk}^{r}},\frac{data\_size}{DPR}\},\\&max\{\frac{data\_size}{DPR},\frac{data\_size'}{P_{disk}^{w}}\}\}=\\&max\{\frac{data\_size}{P_{disk}^{r}},\frac{data\_size}{DPR},\frac{data\_size'}{P_{disk}^{w}}\}\end{aligned} \tag{3}$$

如图3所示，压缩算法一般自带一个缓冲区用于存储待压缩或解压缩的数据，因此压缩或解压缩与它们的数据供给端也可用“生产-消费”模型表示，其压缩时间和解压缩时间可分别表示为：

$$T_{comp}=max\left\{\frac{data\_size}{data\_trans\_speed},\frac{data\_size}{P_{comp}}\right\} \quad (4)$$

$$T_{decomp}=max\left\{\frac{data\_size}{data\_trans\_speed},\frac{data\_size}{P_{decomp}}\right\} \quad (5)$$

其中data_trans_speed表示数据的传输速度，P_{comp}和P_{decomp}分别表示算法的压缩速度和解压缩速度.

下面基于公式(3)～(5)分析揭示影响Spark下数据压缩性能的关键因素.

(1)任务的执行逻辑只包含数据压缩

在这种情况下，磁盘的读取数据即为压缩的数据供给端，因此公式(3)可以表示为：

$$T_t=max\left\{\frac{data\_size}{P^r_{disk}},\frac{data\_size}{P_{comp}},\frac{data\_size'}{P^w_{disk}}\right\} \quad (6)$$

其中data_size′表示压缩后的数据大小，它等于原始数据大小乘以压缩算法的压缩比，即

$$data\_size'=data\_size*compression\_ratio$$

(2)任务的执行逻辑只包含数据解压缩

这种情况下公式(3)可以表示为：

$$T_t=max\left\{\frac{data\_size}{P^r_{disk}},\frac{data\_size}{P_{uncomp}},\frac{data\_size'}{P^w_{disk}}\right\} \quad (7)$$

其中$data_size'$表示解压缩后的数据大小，其值为data_size/compression_ratio.

公式(6)和(7)首先表明在压缩数据或解压数据时影响任务执行时间的重要因素是：磁盘的读写性能，压缩算法的压缩性能、解压性能或压缩比. 它们同时也揭示出当其中一个因素成为性能瓶颈时，其他因素对任务的执行时间将不会产生任何影响. 比如在压缩数据时，当磁盘读成为性能瓶颈，任务的执行时间只由读取数据的时间决定，与数据写入时间和压缩时间无关.

(3)Map任务和Reduce任务

如4.1节所述，Shuffle读写也涉及到数据的压缩与解压缩，其中Map任务负责Shuffle写而Reduce任务负责Shuffle读. 因此前者主要负责数据的读取、处理、压缩和写入而后者主要负责数据的解压缩、处理与写入. 对于Map任务，压缩算法的数据即为任务处理完之后的数据，因此有：

$$\frac{data\_size'}{data\_trans\_speed}=\frac{data\_size}{DPR} \quad (8)$$

data_size′表示输入数据被处理之后的大小. 结合公式(3)和(4)，Map任务执行时间可以表示为：

$$T_{map}=max\left\{\frac{data\_size}{P^r_{disk}},\frac{data\_size}{DPR},\frac{data\_size'}{P_{comp}},\frac{data\_size''}{P^w_{disk}}\right\} \quad (9)$$

其中data_size″则表示压缩数据的大小. 同理，结合公式(3)和(5)，Reduce的任务执行时间可表示为：

$$T_{red}=max\left\{\frac{data\_size}{P^r_{disk}},\frac{data\_size}{P_{uncomp}},\frac{data\_size'}{DPR},\frac{data\_size''}{P^w_{disk}}\right\} \quad (10)$$

其中data_size′表示解压后的数据大小，data_size″则表示数据被处理之后的大小.

公式(8)表明DPR成为影响压缩性能的一个重要因素，其中CPU性能和任务的执行逻辑共同决定了DPR.

(4)压缩算法在Spark环境下的性能比较策略

图3所示，任务的执行是基于数据流且压缩与解压缩只是执行的一部分，因此无法从任务的执行时间中有效分离出数据的压缩时间和解压时间，以致无法直接比较不同压缩算法在Spark环境下的性能. 但是同一阶段的所有任务在不同压缩算法下的输入数据与执行逻辑均是相同的，因此公式(3)～(10)中的data_size、P^w_{disk}、$\frac{data_size}{DPR}$和P^r_{disk}均独立于压缩算法. 于是可使用T_t来比较不同压缩算法在任务执行中的性能差异，并且T_t可以从Spark日志系统中直接获取. 因此可使用公式(2)比较不同压缩算法在Spark环境下的性能差异.

公式(4)和(5)表明当数据供给速度低于压缩算法的压缩速度或解压速度时，压缩算法的实际性能由数据供给速度决定. 因此为了比较压缩算法在Spark环境下的性能，在使用公式(2)比较压缩算性能时必须满足如下条件：

$$\frac{data\_size}{P^r_{disk}}>P_{decomp}$$

$$\frac{data\_size}{P^w_{disk}}>P_{comp}$$

$$\frac{data\_size}{DPR}>P_{comp}$$

即只有在数据供给速度应该大于压缩算法的压缩或解压缩速度时，才能有效地评估压缩算法在Spark环境下的执行性能. 一般而言，data_size可以通过

Spark系统日志获取，P^{r}_{disk}和P^{w}_{disk}可以通过系统检测工具周期性采集磁盘的数据传输率来估算，$\frac{data_size}{DPR}$可以通过日志中的T_t估算，P_{comp}和P_{decomp}可以通过压缩算法的基准测试程序估算.

5 一种支持Spark使能硬件压缩算法的三层架构

内置于CPU中的硬件压缩算法可有效提升压缩算法的性能，但无法被操作系统和应用程序有效地识别和使用. 为此，硬件压缩算法一般采用软硬件分离设计：压缩算法的核心功能硬化在CPU中，额外的驱动程序对外提供统一接口供上层应用程序使用.

KAEzip是一种采用软硬件分离设计并内置于鲲鹏920 CPU中的硬件压缩算法. 本节以KAEzip为基础提出一种基于Spark的三层架构，支持硬化压缩算法，为评估改进这类压缩算法在Spark环境下的性能奠定基础. 得益于软硬件分离设计，该三层架构也能广泛应用于其他硬件压缩算法.

5.1 Spark的压缩模块

Spark Core是Spark的一个核心组件，设计实现了基于RDD的存储与执行机制，它的压缩模块核心是一个Scala特征，如图6所示. 它抽象出所有压缩算法必须实现的方法：*CompressedOutputStream*()和*CompressedInputStream*()，实现以插件方式扩展该模块支持各种压缩算法. 以谷歌的snappy压缩算法为例，为了使Spark Core支持该算法，仅需要定义一个实现这两个方法的类即可. 这个两个方法最终以Java本地接口的方式调用操作系统中的snappy动态库，实现对数据的压缩与解压.

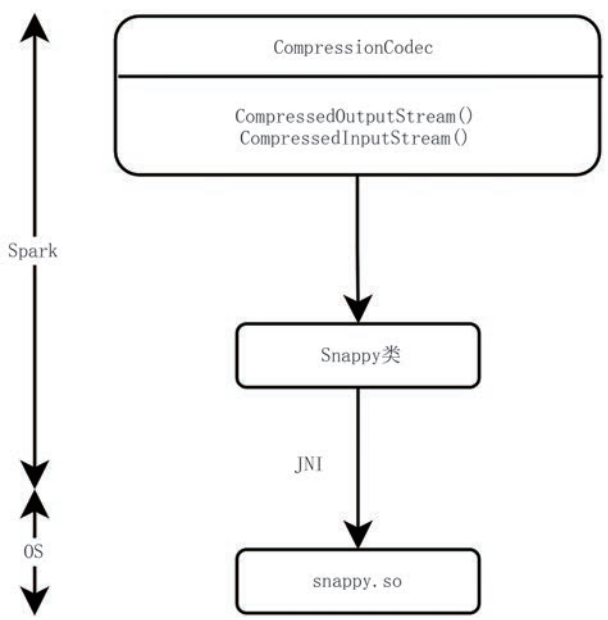

图6 Spark Core的压缩模块与功能实现

5.2 一种Spark使能硬化压缩算法的三层架构

本节首先提出一种基于Spark Core的三层架构，支持Spark使能内置于芯片中的硬件压缩算法. 在此基础上，展示一种如何扩展该架构揭示硬化压缩算法缺陷的方法，最后提出一种基于该架构的优化方法.

5.2.1 三层架构的组成

图7展示了本文提出的三层架构，其中*HWCompression*表示一种硬件压缩算法的名字. 首先特征CompressionCodec被扩展用于识别关键字*HWCompression*：在遇到该关键字时，将其映射成一个HWCompressionCodec类. 该类负责对关键字进行语义计算，即支持采用*HWCompression*对数据流进行压缩与解压缩.

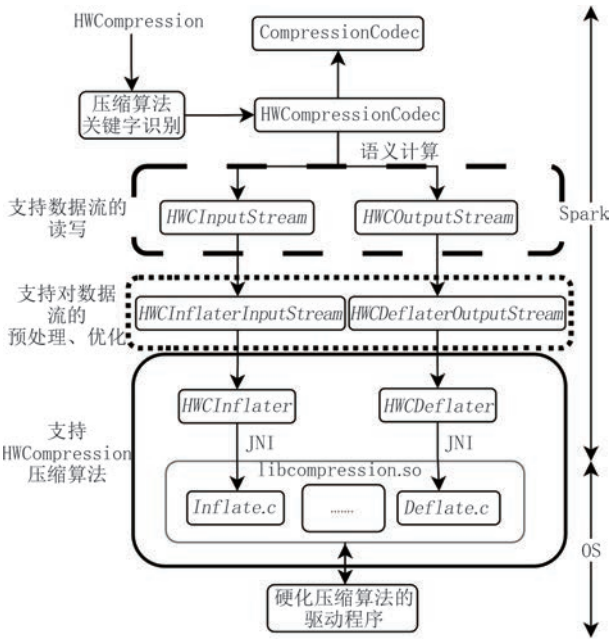

图7 Spark Core支持硬化压缩算法

具体而言，语义计算可依次划分为三层：1）支持Spark Core的输入输出数据流，2）支持对数据流的预处理和优化，和3）支持采用*HWCompression*压缩算法对数据进行压缩与解压缩. 它们之间的关系如图7所示. 第一层中的HWCIntputStream类和HWCOutputStream类分别继承JDK中的抽象类InputStream和OutputStream，并各自实现这两个类的所有抽象方法，其中包括最重要的*read*()方法和

write()方法，支持对Spark Core的流数据进行读写操作. 类HWCIntputStream和HWCOutputStream均内含缓冲区以提升读写性能. 第二层中的类HWCInflaterInputStream和HWCDeflaterOutputStream分别继承JDK中的抽象类FilterInputStream和FilterOutputStream，支持对流数据的压缩与解压缩. 为了进一步支持使用HWCompression压缩算法，它们都内建缓冲区存储待处理的数据，然后分别创建第三层中的类HWCInflater和HWCDeflater的对象实现对数据进行解压缩与压缩操作. 此外，作为一个中间层，第二层还负责数据的预处理与优化.

第三层主要负责使用*HWCompression*压缩算法对数据进行压缩或解压缩. 由于该层中的类HWCInflater和HWCDeflater的工作机制大致相同，本文以HWCDeflater为例介绍如何实现支持*HWCompression*压缩算法. 该类并未通过Java语言实现具体的压缩算法，而是采用JNI方式调用到动态库*libcompression. so*中的相应压缩算法. 由于C应用程序的执行性能远高于Java应用程序，这种基于动态库的实现方式使得HWCDeflater具备较高压缩性能. 具体而言，HWCDeflater中的*deflate*()函数主要以字节为单位负责从缓冲区读取数据，然后通过该类中声明的本地接口函数*deflateBytes*()将数据传递给Deflate. c中的同名函数，实现对数据的压缩. 数据压缩完成之后会被写入到输出流并实时地更新缓冲区的偏移量以便继续压缩下一组数据. Deflate. c是一个基于*HWCompression*硬件压缩算法驱动程序的C应用程序，它使用驱动程序提供的对外接口实现HWCDeflater中声明的所有本地接口方法，其中包括*deflateBytes*(). 当本地方法被调用时，*HWCompression*就会被驱动程序激活进而对数据采用硬件压缩.

类HWCompressionCodec使用上述三层架构中的类实现使用*HWCompression*对数据流进行压缩或解压缩. 但为了使Spark Core支持*HWCompression*，该类还要实现特征CompressionCodec中规定的两个方法，即将Spark应用程序运行时产生的数据流转为HWCInputStream对象或HWCOutputStream对象.

5.2.2　三层架构的扩展与KAEzip的数据膨胀问题

除了支持Spark使能新的压缩算法外，三层架构也能被扩展用于剖析Spark环境下压缩的执行特点，助力解决硬件压缩算法可能存在的缺陷. 比如，融入日志系统到三层架构中第三层，可以轻量级地统计Spark任务执行中的压缩次数、压缩的输入数据大小和输出数据大小等关键性能指标. 这些指标均与Spark运行时所选择的压缩算法无关，可有效分析研究压缩算法在Spark环境下的执行特征.

KAEzip在压缩数据时可能会发生数据膨胀. 比如，为了量化地表示一个文档与它的所有单词之间的关系，LDA在训练模型之前需要通过笛卡尔乘积运算构建一个带权重的边集. 在Spark中这样的一个运算会产生一个Shuffle. 图8(a)展示了该阶段的Shuffle写的数据量，其中none表示未设置任何压缩算法，在此配置下的Shuffle写的数据量即为待压缩的数据量. 可以看出lz4、snappy和zlib都能有效地对数据进行压缩，但是KAEzip却导致了“数据膨胀”. 比如，当测试数据为50 MB时，在该阶段Shuffle写的数据量大约为250 MB，这四种压缩算法分别将其压缩为36. 3 MB、32. 5 MB、16. 9 MB和489. 2 MB. 图8(b)进一步展示了sort、wordcount、KMeans和LDA四个基准测试程序的Shuffle文件在不同压缩算法下的压缩比. 可以看出KAEzip仅在LDA下的压缩比大于1(发生了数据膨胀).

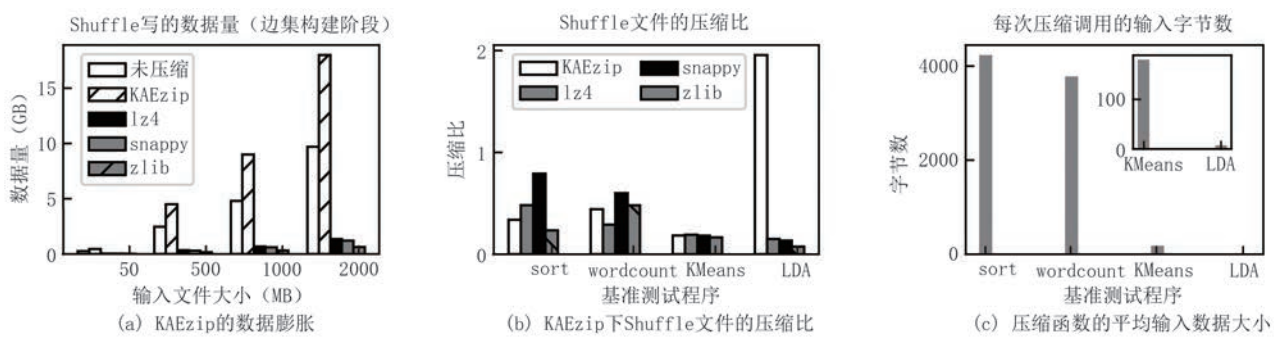

图8　数据膨胀与根源分析

为了揭示KAEzip导致数据膨胀的根源，三层架构中的第三层被扩展用于采集与压缩相关的性能指标，其中图8(c)展示了四个测试程序运行时压缩函数的平均输入数据(按字节计算)，可以看出sort和wordcount一次压缩的数据平均值分别为4234和3770个字节，但是KMeans和LDA的平均值则分别为180和7.2个字节. 通过对比可知压缩函数过低的输入数据是导致KAEzip压缩数据膨胀的诱因.

引发数据膨胀的根源与普遍性 KAEzip是基于字典的无损压缩算法，即通过先前输入数据构建字典，然后基于字典匹配后续输入数据的最长子串，并以三元组(D，L，C)的形式表示，其中D表示匹配的起始点，L表示匹配的长度，C表示下一个待匹配字符. 如果不匹配则D和L分别为0. 因此若L小于3则会引发压缩时的数据膨胀.

LDA中，Shuffle写的数据内容是一个对(pair)集合，其中对的第一个元素是一个Integer类的对象，用来记录分区号；第二个元素是一个Tuple3类的对象，用来记录一条边的信息，即文档ID、单词ID以及它们之间的权重. 根据压缩算法在Spark中的运行机制，在Shuffle写将数据写入到文件时，一个对的写入操作会触发两次压缩函数调用，使得LDA下所有压缩算法的压缩函数的输入数据量都非常小. 比如，当压缩对的第一个元素时，压缩函数的输入数据量是4个字节. 在这种环境下KAEzip无法构建出有效字典，导致后续输入数据在压缩时无法匹配字典中的数据，使得一个字节在压缩之后不得不使用三个字节表示，最终促使KAEzip发生压缩时的数据膨胀.

社交网络领域中的机器学习算法往往也包含大量求取两个集合的笛卡尔积，用于获取人际关系网络，这与LDA中边集构建类似：(1)集合中的元素包含的数据量都很小，往往是一个整型或Long型对象，(2)都通过笛卡尔乘积操作求取. 因此这类算法在KAEzip下也会产生类似于LDA中的数据膨胀问题.

5.2.3 基于三层架构的压缩与解压缩优化

如4.1节所述，在Spark应用程序的执行中，一次对象的写入或读取就会触发一次压缩或解压缩调用. 大量函数调用不仅减少压缩函数的平均输入数据量，诱发KAEzip导致数据膨胀等问题，而且会增加执行开销影响程序的执行性能. 为此，本文提出一种基于Spark的优化方法. 该方法的基本思想是：将待压缩对象存入到缓存区，当缓存被充满时才调用相应的压缩算法对缓冲区中的数据进行压缩，具体过程如图9所示.

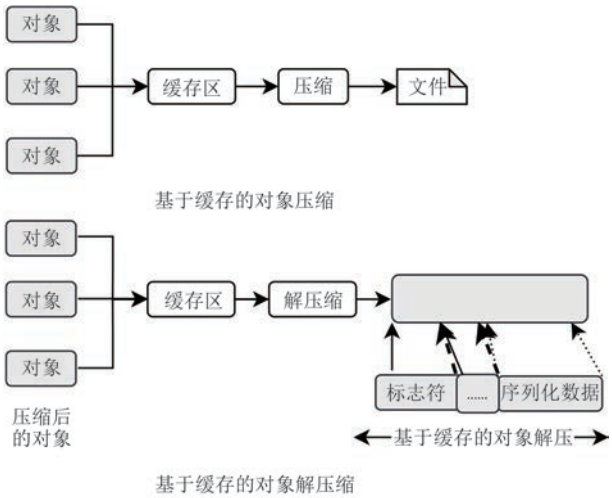

图9 基于缓存的压缩与解压缩

缓存的引入将对单个对象的压缩转变成对批量对象的压缩，从而有效地减少了函数的调用次数. 该优化方法可融入到上述第二层中的类HWCDeflaterOutputStream，算法1描述了该优化方法的细节. 它将字节数组*b_delay*作为缓存区融入到HWCDeflaterOutputStream中的*write*()函数. 在序列化一个对象时，JDK中的*writeObject*()方法会调用到*write*()对数据进行压缩. *write*()在压缩字节数组*b*中的字节时，先判断缓存区*b_delay*是否有足够空间容纳待压缩的数据(第3行). 如果有，则将待压缩数据直接拷贝到*b_delay*. 否则将*b_delay*设置为压缩算法的输入数据，调用压缩算法进行数据压缩，压缩完之后清空*b_delay*然后把数组*b*中的数据拷贝到*b_delay*中，为下一次压缩缓存数据.

算法1. 基于缓存的数据压缩.

```
输入：byte[] b, offset, length
输出：the number of compressed data in byte
write(b, off, len)：
    IF ! def.finished()
        IF(b_delay.length - b_delay_len) > len
            copy(b, off, len);
        ELSE
            def.setInput(b_delay, 0, b_delay_len);
            WHILE ! def.needsInput()
                deflate();
            ENDWHILE
            clearBDelay();
            copy(b,off, len);
        ENDIF
    ENDIF
```

缓存也能引入到数据流的解压过程，具体如

图9展示. 当需要解压时，*HWCompression*会预先解压指定大小的数据并存入到缓存中，接下来的解压操作只需直接从缓存中读取数据即可，进而通过减少频繁地解压缩调用提升*HWCompression*的解压缩性能. 为了正确地获取一个解压后的对象，该优化方法还需要精确地将原解压缩的数据流映射到缓存中的相应位置，以便能正确地读取一个对象的序列化数据，否则可能导致反序列化失败. 这些优化方法均可通过融入到图7中的第二层实现对KAEzip的优化.

该优化方法也可融入到上述第二层中的类HWCDeflaterInputStream，优化*HWCompression*在Spark中的解压缩性能，算法2描述了该方法的细节. 它使用字节数组*buffer*(默认大小为8KB)作为缓存区融入到HWCDeflaterInputStream中*read*()函数. 在反序列化一个对象时，JDK中*readObject*()函数会调用到*read*()函数对数据进行解压缩. *read*()在解压缩指定数量的字节到字节数组*b*时，会优先从字节数组*buffer*中直接拷贝数据到*b*(第1行). 同时更新数据的起始位置*off*与待压缩的字节数*len*(第2行)，以确保下一次数据解压缩的正确性. 如果*buffer*中的数据充足，则返回解压出的字节数(第4行)，否则预解压一定量的数据到*buffer*中，然后继续从*buffer*中拷贝剩余字节数到*b*(第19行).

算法2.　基于缓存的数据解压缩.

```
输入:byte[] b, offset, length
输出:the number of uncompressed data in byte
read(b, off, len):
    int n = copy(b, off, len);
    off += n; len -= n; read_num += n;
    IF len == 0
        RETURN n;
    ENDIF
    WHILE len > 0
        k =pre_read();
        IF k == -1
            IF read_num == 0
                RETURN -1;
            ELSE
                RETURN read_num;
            ENDIF
        ENDIF
        IF k == 0
            RETURN read_num;
        ENDIF
        n = copy(b,off,len);
        off += n; len -= n; read_num += n;
    ENDWHILE
```

算法3描述了预解压函数*pre_read*(). 如果*buffer*为空，它调用压缩算法中的解压方法解压一定量的数据到*buffer*中，并返回已经解压出的字节数(第3行). 如果数据流中的数据已经读完，则返回-1(第7行).

算法3.　数据的预解压缩.

```
输出:the number of uncompressed data in byte
int pre_read():
    ensureOpen();
    IF remaining_elements == 0
     WHILE (n = inf. inflate (buffer, 0, buffer.
     length)) == 0
         IF n == 0
           IF inf.finished()
                 reachEOF = true;
                 buffer_elements = 0;
                 buffer_off = 0;
                 RETURN -1;
             ENDIF
         ENDIF
     ENDWHILE
     RETURN n;
    ENDIF
```

本节提出的基于缓存的压缩与解压缩方法不仅能够有效地解决KAEzip的数据膨胀问题，而且能够有效提升Spark应用程序的性能，更多的实验验证信息请见实验6.4节和6.5节.

5.3　三层架构在Hadoop中的复用与可扩展

本节提出的三层架构方法具有较好的复用性，可以相对容易复用到大数据领域中的其他基础性软件中. 比如，与Spark类似，Hadoop也包含一个压缩组件*compress*支持各种压缩算法，包括lz4、BZip2、snappy和zstd. 当前Hadoop中的HDFS、MapReduce、Hive和Hbase都应用该组件实现对数据的压缩与解压缩. 因此实现对该组件的扩展就能够使这些大数据领域中的重要基础性系统软件支持新的压缩算法，比如KAEzip.

图10(左)展示了*compress*组件的架构图，其中*CompressionCodec*向上提供统一接口，以便HDFS和MapReduce等基础性系统软件无需做任何修改就能使用*compress*组件提供的压缩与解压缩功能. 以压缩算法BZip2为例，该组件也是采用典型的三层架构支持各种压缩算法：第一层主要使*compress*

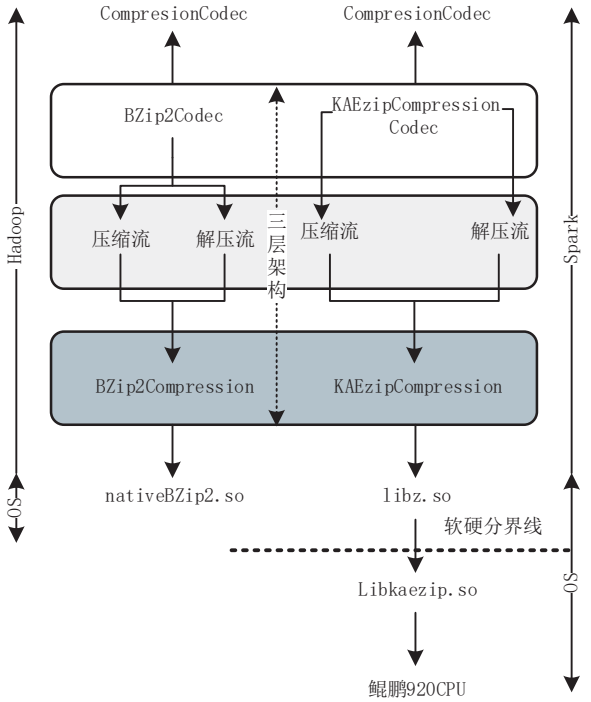

图10 三层架构在Hadoop中的复用

组件能够识别利用该压缩算法;第二层负责数据流的压缩与解压缩;第三层则是实现数据的压缩与解压缩操作. 出于执行性能的考虑,第三层并未包含用Java实现的压缩与解压缩算法,而是直接通过调用相应的动态库实现数据的压缩与解压缩,即动态库用C语言实现相应的压缩与解压缩算法并对外提供接口函数,第三层直接调用这些接口实现对数据的压缩与解压缩.

对比图10中的左右两个架构可知:三层架构实际上同构于Hadoop中*compress*组件的架构. 这使得三层架构可以很容易地复用融入到Hadoop中,实现Hadoop支持新的硬件压缩算法,从而使它包含的系统软件能识别使用该算法.

此外,基于我们的前期工作经验[30],分层可以有效提高架构的可扩展性,因此本文提出的三层架构遵守基于功能性的分层设计思想,即一层只针对一个(组)功能而设计,并且不同层之间的功能不重叠. 这使得架构可以根据具体的应用场景和数据特点对不同层做相应的扩展和改进,而无需担心层之间的相互影响. 比如,5.2.3节针对Spark中的数据流特点进行压缩或解压缩优化,则只需重点专注如何对第二层进行改进而无需考虑改进对其它层的影响. 因此三层架构具备较好的可扩展性.

6 实验与评估

KAEzip作为一款可实用的硬件压缩算法,本章以它为实验对象,通过广泛实验测试评估它在Spark环境下的性能,然后再结合4.2节中的性能模型分析揭示出Spark环境下制约硬件压缩算法性能的关键因素,为进一步演化迭代这类算法奠定基础. 在此基础上,通过设置新的实验环境测试评估三层架构的性能优势与最坏情况下的性能开销.

实验环境的具体配置如表1所示,其中鲲鹏920的实验评估环境主要包含一台华为泰山200服务器①. 它配置了一颗128核的鲲鹏920 CPU、128 GB内存和2块2 TB磁盘. 操作系统为华为自研的openEuler 20.03. 由于鲲鹏920 CPU是基于ARM 64位架构,因此实验中的Java虚拟机使用ARM 64位版本. Spark为通过扩展融合三层架构的Spark3.0.3. 数据产生器为BigDataGeneratorSuite[26,27]. 本文使用文献[16-18]中的典型基准测试程序评估KAEzip在Spark环境下的压缩与解压缩性能. 表1底部展示了所有的基准测试程序以及对应测试数据.

鲲鹏920实验平台用于评估验证三层架构在ARM架构下的有效性,x86实验平台用于评估三层架构的开销与性能优势. 本节所有实验均运行五次然后取平均值以减少实验误差. 为了便于描述不同压缩算法的性能差异,本节以zlib压缩算法在Spark下的性能作为基线性能.

6.1 KAEzip在Spark中的压缩性能评估

本节主要评估KAEzip在Spark中的数据压缩性能. 测试数据由BigDataGeneratorSuite产生并以文本格式存储在HDFS上,其大小从512 MB逐渐增至25 GB. 测试程序的主要执行逻辑是:从HDFS上读取并压缩数据,然后将结果保存到HDFS中.

图11(a)中的实验结果表明:随着测试数据的增加,KAEzip提升测试程序性能的幅度逐渐减小直至与其他压缩算法大致相当. 具体而言,当测试数据为512 MB时,KAEzip较zlib提升了测试程序性能将近62%,较snappy相比也有大约13.8%的性能提升. 测试数据增加到2 GB时,KAEzip领先zlib的性能优势减少到大约37%,与snappy的性能差异小于1%. 测试数据增加到10 GB和25 GB时,测试程序在四种压缩算法下的性能差异均小于3%,可以

① 该服务器由华为技术有限公司提供.

表1　实验环境

鲲鹏920实验平台的硬件与软件配置	
Kunpeng920 CPU	128核
内存	128 GB
网络	1000 Mbps
磁盘	2*2 TB 7200 RPM 6 Gbps SATA
操作系统	openEuler 20.03 (4.19.90-2012.4.0.0053 oe1.aarch64)
Java版本	Java 1.8.0 aarch64
Spark版本	3.0.3
Hadoop版本	3.1.1 aarch64
Parquet版本	2.4.0
ORC版本	1.16
executor-cores	1
exeuctor-memory	2 GB
数据产生器	BigDataGeneratorSuite
压缩算法	
zlib版本	1.2.11
KAEzip版本	1.13.1
lz4版本	1.9.0
snappy版本	1.1.3
x86实验平台的硬件与软件配置	
Intel i5-8400	6核
内存	24 GB
网络	1000 Mbps
磁盘	2 TB 7200 RPM 6 Gbps SATA
操作系统	Centos7.6
Java版本	1.8.0
Spark版本	3.0.3
Hadoop版本	3.1.1
基准测试程序	
测试程序	测试数据
Spark SQL write(压缩)	512 MB～25 GB
Spark SQL read(解压缩)	512 MB～25 GB
sort	512 MB～50 GB
wordcount	512 MB～50 GB
KMeans	1 GB～30 GB
LDA	1 GB～4 GB
SQL Join	大表:2 GB～55 GB 小表:70 MB～2 GB

认为它们对测试程序的性能影响基本相当. 对比图11(a)和图11(b)可以发现任务的性能提升与测试程序的性能提升存在显著的正向强相关. 比如,与zlib和snappy相比,在KAEzip下的测试程序性能分别提升了62%和13.8%时,在KAEzip下的任务平均执行时间也相应的缩短将近62%和16%,当测试程序在所有压缩算法下的性能差异小于3%时,任务平均执行时间也在大约5%以内波动.

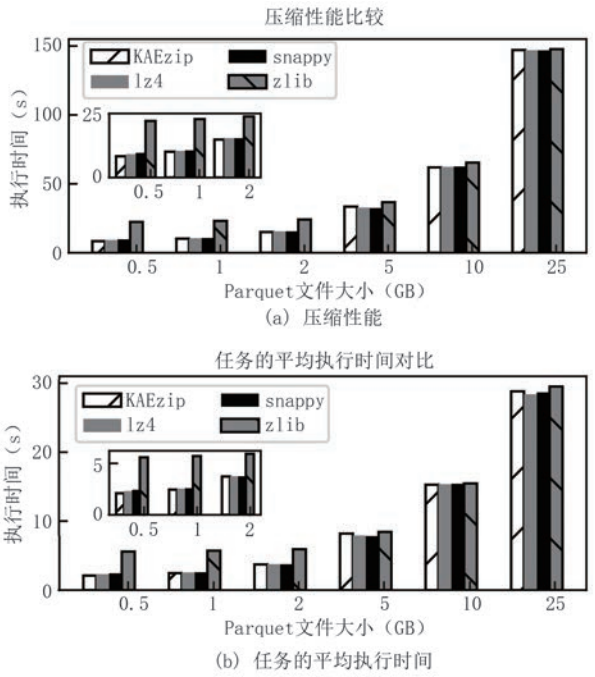

图11　Parquet文件格式压缩性能比较

根源分析如下. 图12(a)～(d)展示了在测试程序执行时磁盘写的数据传输率. 该程序主要是压缩读取到的数据然后写入到HDFS中,但是这四个子图表明程序执行结束前并未发生大量的写操作. 这是由于HDFS采用延迟写策略:程序提交到HDFS的写数据往往在它执行完之后才开始写入. 因此写并不会影响到它的执行性能. 针对这些实验结果,本文在分析压缩算法对测试程序性能的影响时也忽略写操作,因此公式(6)可以简化为:

$$T_{task} = max\left\{\frac{data\_size}{P^{r}_{disk}}, \frac{data\_size}{P_{comp}}\right\} \qquad (6')$$

可以看出磁盘的读性能与压缩算法的压缩性能决定了任务的执行时间.

图12(e)～(f)展示了测试程序执行时磁盘读的数据传输率. 首先可以看出,随着输入数据的增加,传输率也随着增加. 当数据增为2 GB时,KAEzip和snappy下的传输率率先接近磁盘读性能的峰值,即200 000 KB/s. lz4和zlib下的数据传输率在5 GB时也接近峰值. 在10 GB时,四种压缩算法下的数据传输率均到达了峰值,并随着输入数据增加仍维持在峰值附近轻微波动. 另一方面,随着KAEzip和snappy下的磁盘数据传输率逐渐上升至峰值,它们领先zlib的性能分别从62%和56%逐渐降至为2.8%和7%. 当它们的数据传输率维持在峰值附近波动时,它们与zlib的性能差异也在3%附近波动. 这表明随着输入数据的增加,磁盘的读性能会成为

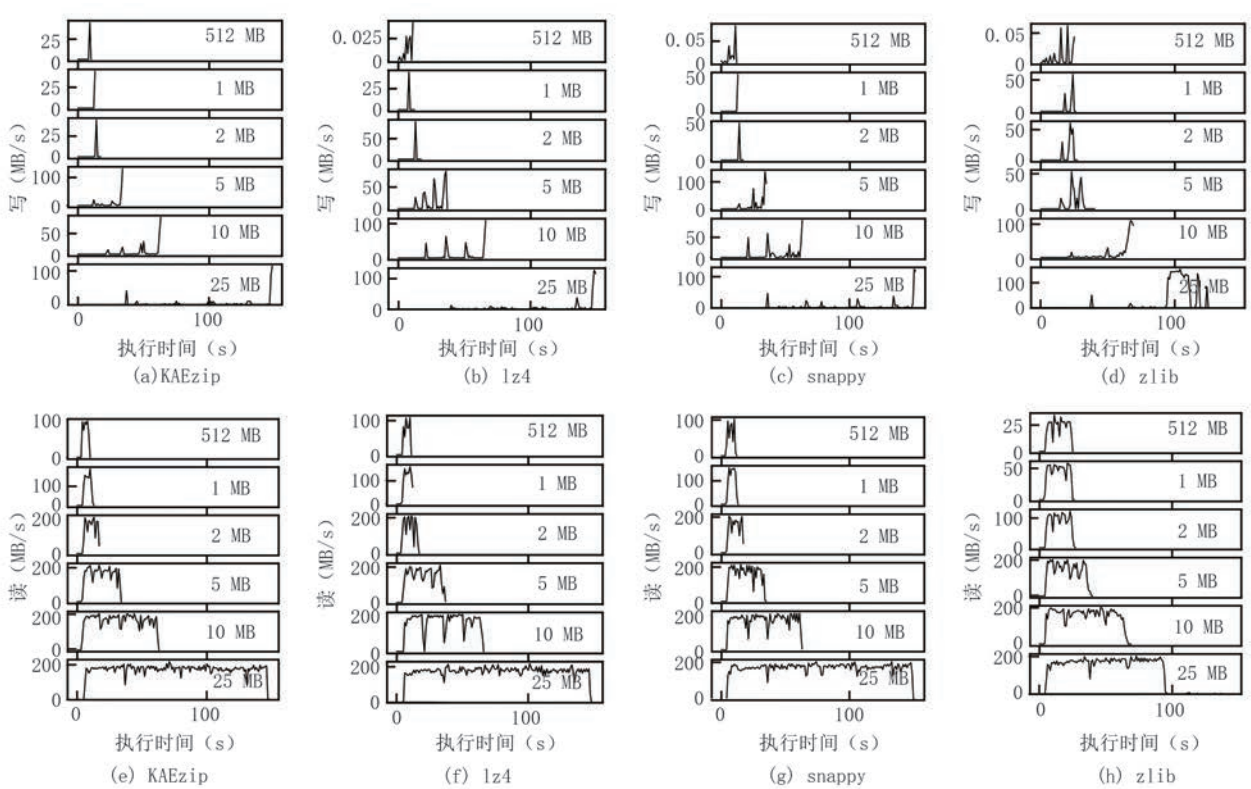

图12 不同压缩算法压缩数据时的磁盘读写性能(Parquet文件格式)

任务执行性能的瓶颈,最终决定任务的执行时间.

假设测试程序的执行器数目为k,则它的并发度也为k,一个任务的数据传输率可以表示为:

$$P_{disk}^{r}=\frac{200\,000\,\text{KB}}{k} \tag{11}$$

在测试中执行器的数目设置为①:

$$k=executor\_num=\min\left\{\frac{input\_data\_size}{512\,\text{MB}},40\right\} \tag{12}$$

当输入数据为512 MB时执行器的数量为1,对比图12(e)~(f)可知,KAEzip、lz4、snappy和zlib下的数据传输率分别大约为100 000 KB/s、90 000 KB/s、800 000 KB/s和25 000 KB/s. 由于这些数据传输率远未到达磁盘数据传输的极值,因此根据公式(6′)知:数据传输率反映出压缩算法在Spark环境下的压缩性能. 这也表明KAEzip具有最好的压缩性能.

随着输入数据的增加,执行器的数量也线性增加,但磁盘的数据传输率并未呈现出相应幅度的增加. 比如当输入数据为512 MB时,KAEzip下的磁盘数据传输率为100 000 KB/s;当数据增至1 GB时,执行器的数量增加了一倍,但是数据传输率只增至大约180 000 KB/s. 根据公式(11)可知,此时的P_{disk}^{r}大约为90 000 KB/s,已经小于KAEzip的最大压缩速度;当数据增至2 GB时,执行器的数量再次增加一倍但是数据传输率仅仅增加了大约10%,并开始接近磁盘数据传输率的峰值,此时的P_{disk}^{r}进一步降为大约50 000 KB/s. 当输入数据为5 GB时,P_{disk}^{r}为20 000 KB/s,已经小于所有压缩算法的最大压缩速度. 根据公式(6′)可知任务的执行性能已经由磁盘读的传输率决定,而不是压缩算法的压缩性能决定. 这是导致所有压缩算法下的测试程序性能大致相当的根本原因.

在四种压缩算法下,测试程序将压缩结果保存为文本文件时,它们的性能表现与保存为Parquet格式文件相似,本文不再做相应描述.

数据供给率和压缩性能共同决定了压缩算法在Spark下的压缩性能. 在数据供给充足条件下,KAEzip压缩算法较其他压缩算法在Spark压缩中有明显的性能优势. 但当并发度随着输入数据的增加而增加时,单个任务的数据供给率呈线性下降的趋势. 当数据供给率小于压缩算法的最大压缩性能时,压缩算法的压缩性能就呈现出一致性,即四种压缩算法在Spark下的压缩性能大致相当.

① 由于主机的内存为128 GB,最多只能支持64个执行器并发执行. 考虑到HDFS和操作系统的内存使用,故将执行器数目上限设置为40以保证实验能正常执行.

6.2　KAEzip在Spark中的解压性能评估

图13展示了解压测试程序的执行流程：生成测试数据和测量测试程序的执行时间. 为了确保测试公平，先利用四种压缩算法对相同数据进行压缩产生四种压缩文件并存储到HDFS中，作为测试程序的数据集. 测试程序会根据压缩文件的类型选择相应压缩算法对其进行解压，但解压结果不会再保存到HDFS中. 测试程序的执行时间可以反映出四种压缩算法在Spark环境下的解压性能.

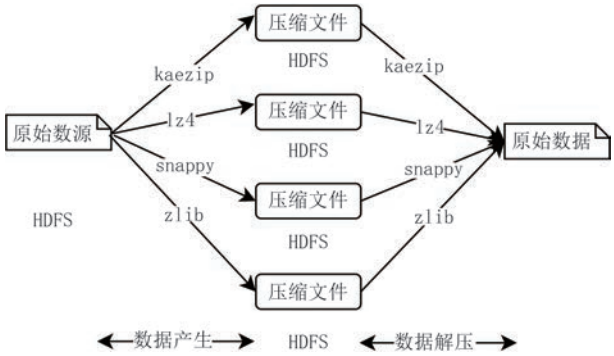

图13　Spark解压测试程序的执行流程

测试程序在解压文本文件时会产生两个阶段，但在解压Parquet文件时则产生三个阶段. 图14(a)展示了原始数据为2 GB时测试程序的执行时间①. 实验结果表明在解压文本文件时第一阶段的执行时间决定了整个程序的执行时间，而在解压Parquet文件时，程序的执行时间由前两个阶段共同决定. 鉴于此，本节在分析压缩算法在Spark环境下的解压性能时会忽略文本文件下的第二阶段和Parquet文件下的第三阶段.

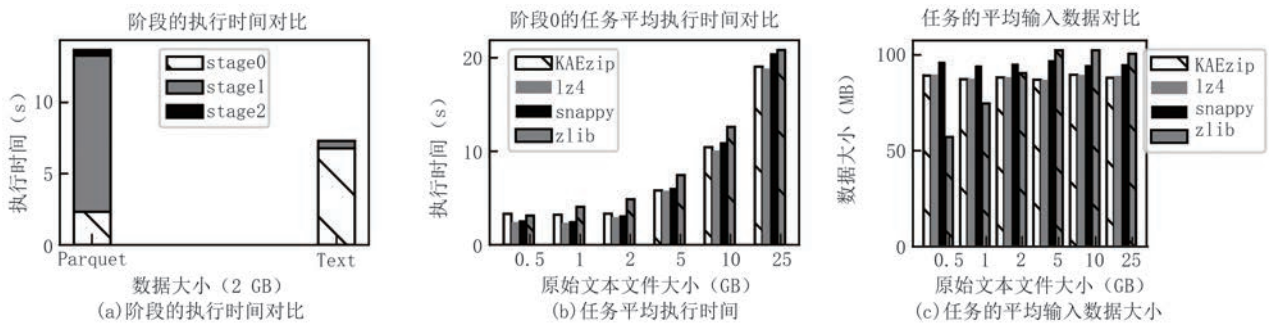

图14　文本文件的解压性能比较

图14(b)展示了测试程序在解压文本文件时第一阶段的任务平均执行时间. 实验结果表明：当原始数据为512 MB时，与zlib相比，KAEzip下的任务平均执行时间长大约6%，与snappy相比也长大约26%. 当数据增加至1 GB时，KAEzip下的任务平均执行时间反而较zlib缩短大约20%，但仍较snappy长大约26%. 当输入数据从2 GB增加到25 GB时，与zlib相比，在KAEzip下的任务平均执行时间从短大约37%逐渐降至短8.5%；而与snappy相比，任务平均执行时间则从长5.4%逐渐变为短7%.

不同压缩算法具有不同的压缩比，因此同一文件在被它们压缩之后的大小也不尽相同. 在解压时，这些不同大小的压缩文件会产生不同的任务数，进而影响任务的平均输入数据大小. 图14(c)展示了不同压缩算法的任务的平均输入数据大小. 比如，当原始文件大小为512 MB时，zlib下任务的平均输入数据量比KAEzip少大约35.9%，比snappy少大约40.7%.

由于测试程序并未将解压结果存入到HDFS，因此公式(7)可以简化为：

$$T_{task}=max\left\{\frac{data\_size}{P^{r}_{disk}},\ \frac{data\_size}{P_{uncomp}}\right\} \qquad (7')$$

图15(a)～(d)展示了测试程序执行时的磁盘读的数据传输率. 可以看出当原始数据为512 MB时，lz4与snappy下的数据传输率的峰值大约为75 000 KB/s，KAEzip的峰值接近50 000 KB/s，zlib的峰值最慢——大约为32 000 KB/s. 这些数据远未达到磁盘读的数据传输峰值，因此根据公式(7′)知这些数据传输率可以反映出压缩算法在Spark环境下的实际解压速度，即lz4和snappy具有最快解压速度，KAEzip次之，zlib的解压速度最差. 由于zlib下任务的平均输入数据量更小，这使得zlib下的任务平均执行时间反而比KAEzip缩短了大约6%，但仍落后snappy大约16%. 这表明当数据量较小时，zlib在Spark环境下较KAEzip具有更好的数据解压性能.

① 其他环境下的测试结果均与之类似，本文不再展示.

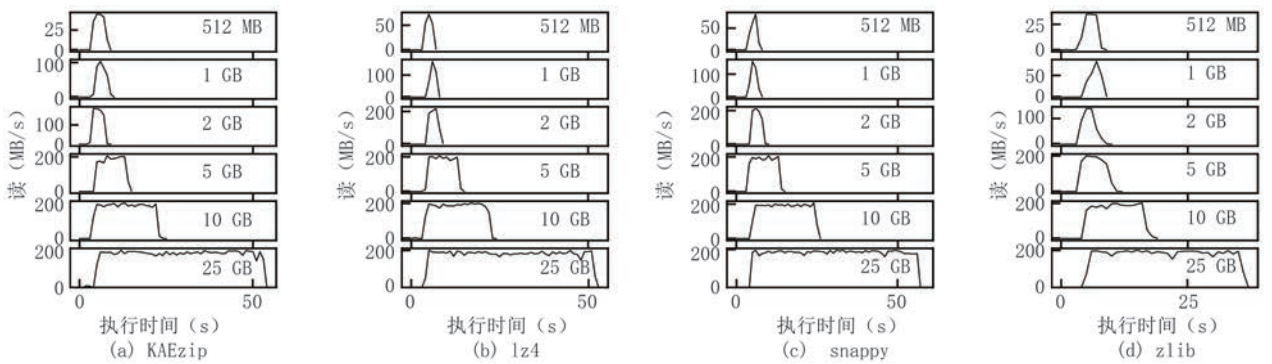

图15 不同压缩算法解压数据时的磁盘读性能(文本文件格式)

在原始文件大小为1024 MB时,与zlib相比,KAEzip下的任务平均输入数据量多大约14.6%,但更快解压速度使得KAEzip下的任务平均执行时间减少了大约21%. 由于KAEzip下的任务平均输入数据量仍只比snappy少大约7.6%,因此KAEzip下的任务平均执行时间仍然落后snappy大约26%. 当原始文件大小为5120 MB时,在所有压缩算法下磁盘读的数据传输率达到了它的峰值. 由于解压测试程序的执行器数量仍根据公式(12)决定,因此根据公式(7′)可知:在四种压缩算法下任务输入数据的数据传输率均大约为20 000 KB/s,已经小于所有压缩算法的解压速度. 在此情况下,任务的平均输入数据量将成为影响其执行时间的关键因素. 由于KAEzip下的任务平均输入数据量小于snappy大约11%,因此KAEzip下的任务平均执行时间开始领先snappy大约4.8%. 随着输入数据的增加,这种领先优势始终由这两种压缩算法下的任务的平均输入数据量决定.

数据供给率、数据解压速度与压缩比共同决定压缩算法在Spark环境下的解压性能. 在数据供给充足条件下,snappy较KAEzip具有明显的解压性能优势. 更高压缩比导致snappy压缩文件比KAEzip更大,从而导致snappy下的任务平均输入数据量也大. 当并发度随着输入数据的增加而增加时,单个任务的数据供给率呈线性下降的趋势. 在数据供给率小于压缩算法的解压速度时,任务的平均输入数据量成为影响任务解压性能的关键因素,这会使得KAEzip比snappy在Spark环境下具有更高解压性能. 与之类似,在数据输入最小时,更小压缩比使得zlib下的任务平均输入数据量更小,从而它比KAEzip在Spark环境下具有更高的解压性能.

6.3 KAEzip在应用场景下的性能评估

应用场景测试程序需要先进行数据处理然后才能进行压缩操作,或者解压数据之后再进行数据处理. 因此影响压缩算法在应用场景下的性能除了磁盘外还包含CPU或内存等因素.

6.3.1 sort

sort在执行时被划分为四个阶段:前两个阶段主要负责数据读取与采样,后两个阶段分别是Shuffle Map阶段(阶段2)和Shuffle Reduce阶段(阶段3),它们主要负责数据的分区、排序与合并. 图16(a)展示了KAEzip下四个阶段执行时间的对比图,其他压缩算法下的阶段执行时间对比与之类似. 可以看出阶段2和阶段3的执行时间分别大约占据sort总执行时间的30%和50%. 压缩算法主要影响阶段2中的Shuffle写操作性能和阶段3中Shuffle读操作性能.

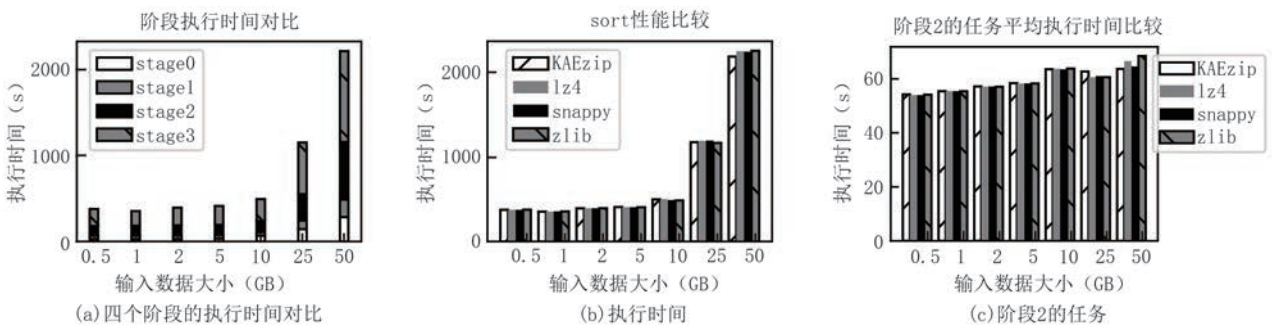

图16 sort性能比较

以zlib下的sort执行时间为基准，图16(b)展示其他压缩算法下的sort执行时间与该基准的差距在0.1%～3%附近波动. 图16(c)展示了阶段2的任务平均执行时间①. 可以看出四种压缩算法下两个阶段的任务平均执行时间的差异均小于3%. 这最终使得四种压缩算法下sort的执行时间基本相当.

图17(a)～(d)展示四种压缩算法下sort执行时磁盘读的数据传输率. 可以看出：当输入数据小于50 GB时磁盘读操作主要发生在阶段0，在其他阶段基本上未见磁盘读操作发生. 读操作的数据传输率随着输入数据增至5 GB时到达了它的峰值20 000 KB/s，这也是磁盘读的数据传输峰值. 当输入数据为50 GB时，由于输入数据过大导致阶段1甚至阶段2都有较为密集的读操作，但是它们的数据传输率远未到达峰值20 000 KB/s.

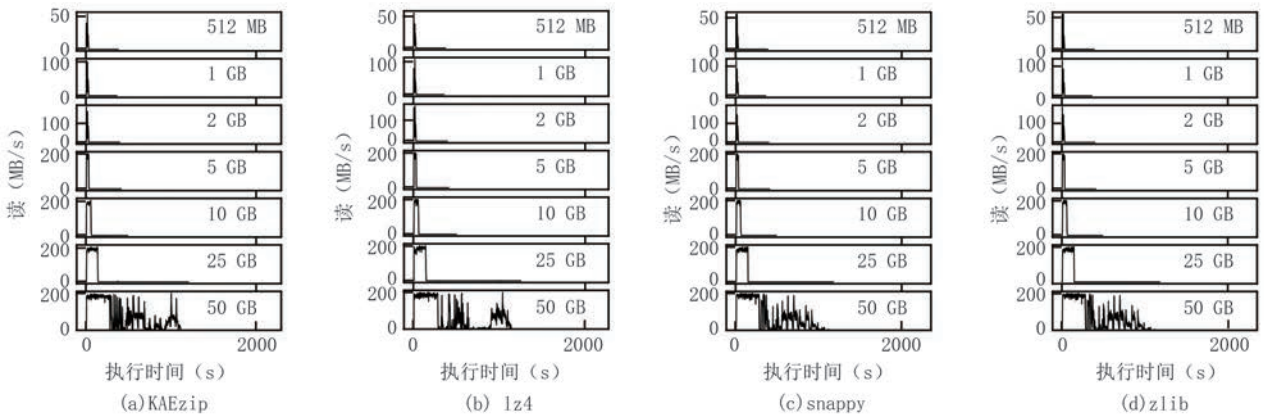

图17　不同压缩算法下sort的磁盘读性能

在Shuffle写将任务的输出数据压缩之前必须等待数据处理完成，这包括对输入数据的分区以及分区内的排序等操作. 这些操作均是CPU密集型操作. 因此公式(9)可以表示为

$$T_{task}=max\left\{\frac{data\_size}{DPR},\frac{data\_size'}{P_{comp}}\right\}\qquad(9')$$

其中如图16(c)所示，实验结果表明阶段2的任务平均执行时间均大于50秒，而任务的输入数据量data_size的平均大小约为128 MB. 6.1节的实验数据表明四种压缩算法的压缩速度$P_{comp}\in\{25\,000\,\mathrm{KB}/s,100\,000\,\mathrm{KB}/s\}$. 因此有：

$$\frac{128\,MB}{50}=2.56\,\mathrm{MB}/s\ll P_{comp}$$

可以看出影响阶段2的任务执行时间的关键因素数是CPU的数据处理能力而不是压缩算法的压缩性能.

四种压缩算法对sort执行时间的影响基本相当. 根本原因是决定了压缩算法的数据供给率，但它又远远低于压缩算法的压缩速度与解压速度，最终导致压缩算法对sort执行时间的影响基本可以忽略.

6.3.2　wordcount

wordcount主要包含两个阶段：Map阶段和Reduce阶段. 前者主要对数据进行读取、分区和合并操作，最终将结果写入到临时文件；后者则对临时文件的数据进行读取解压和再合并. 不同于sort，wordcount的Map阶段的输出数据在合并之后会急剧减少.

图18(a)展示了wordcount的两个阶段的执行时间，可以看出Map阶段的执行时间占总执行时间的比重超过了98%. 图18(b)展示了wordcount在不同压缩算法下的执行时间. 以zlib下wordcount的执行时间为基准，其他压缩算法下的wordcount执行时间与该基准的差距均小于0.1%～3.5%. 因此压缩算法对wordcount的性能影响并不显著.

Map阶段的任务会对输出结果进行合并，然后执行压缩与写入操作. 数据的合并会极大的减少待压缩的数据量. 比如，在输入数据为50 GB时，KAEzip下的一个任务Shuffle写的平均数据量大约为14 KB，远远小于任务的平均输入数据量——128 MB. 因此压缩和解压任务的输出数据的时间实际远小于任务对数据的处理时间，进而可忽略. 这是压缩算法不会影响wordcount执行时间的根本原因. 图18(c)展示了Map阶段的任务平均执行时

① 阶段3的平均任务执行时间与之类似，本文不再展示.

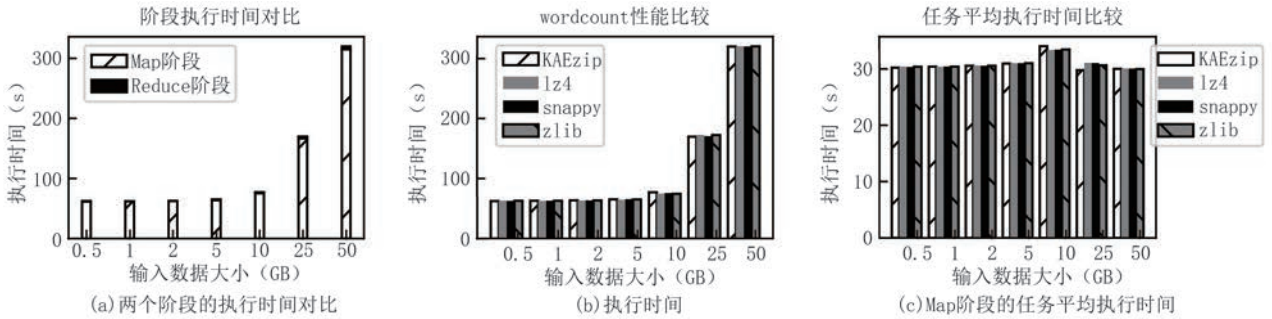

图18 wordcount性能比较

间.以zlib压缩算法下的任务平均执行时间为基准,可以看出其他压缩算法下的任务平均执行时间与该基准的差距均在2.8%以内波动,这表明压缩算法对Map阶段的任务执行无显著影响.

6.3.3 KMeans

在KMeans测试中,聚类数和循环次数分别设置为5和3.KMeans在执行时共产生了八个阶段:前两阶段负责数据的采样与读取,后六个阶段负责数据计算.这六个阶段分别由3次循环产生,其中一次循环都会产生一个Map阶段和一个Reduce阶段.压缩算法主要影响Map阶段的Shuffle写和Reduce阶段的Shuffle读.

图19(a)展示了KAEzip下八个阶段的执行时间.可见阶段0、2、4和6占据了KMeans的大部分执行时间,其中阶段2、4和6是Map阶段,阶段3、5和7分别是它们对应的Reduce阶段.这八个阶段在其他压缩算法下的执行时间与图19(a)类似,本文不再展示.图19(b)展示了KMeans在不同压缩算法下的执行时间.以zlib下KMeans的执行时间为基准,其他压缩算法下的KMeans执行时间与该基准的差距均小于1%.因此压缩算法对KMeans的性能影响并不显著.

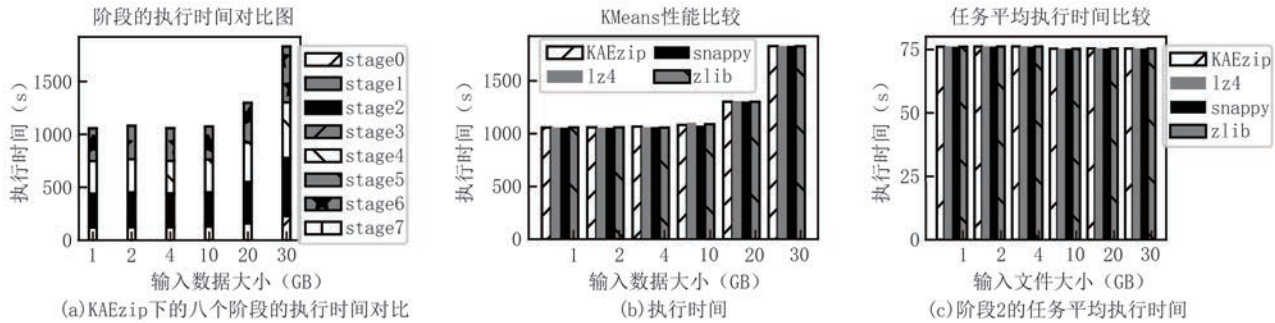

图19 KMeans性能比较

与wordcount一样,KMeans的Map阶段的任务在执行Shuffle写之前会对处理完的数据进行合并,从而极大减少待写的数据量.与Map阶段的任务执行时间相比,压缩Map任务的输出数据的时间可以忽略不计.因此压缩算法基本不影响Map阶段的任务执行.图19(c)展示了阶段2的任务平均执行时间.仍以zlib下任务平均执行时间为基准,其他压缩算法下的任务平均执行时间与该基准的差距均小于1%.阶段4和阶段6的任务平均执行时间与图19(c)类似,文本不再展示.

6.3.4 LDA

在LDA测试中,三层架构中的压缩与解压缩优化被关闭、聚类数和循环次数分别设置为5和2.它的执行共产生了39个阶段.图20(a)展示了在不同压缩算法下LDA的执行时间.以zlib下的LDA执行时间为基准,在50 MB的输入数据下,KAEzip下的LDA执行时间比基准慢大约52%,但是lz4和snappy下的LDA执行时间则比基准快大约50%;当输入数据增至2 GB时,KAEzip下的LDA执行时间仍落后基准大约30%,但是lz4和snappy下的LDA执行时间则仍分别领先基准大约11%和13.9%.

图20(b)展示了该阶段在四种压缩算法下的任务平均执行时间,实验结果表明KAEzip下的任务平均执行时间最长,远超过其他压缩算法下的任务平均执行时间.任务的整个执行过程可以划分为三个部分:产生边集,压缩数据和写数据到临时文件.

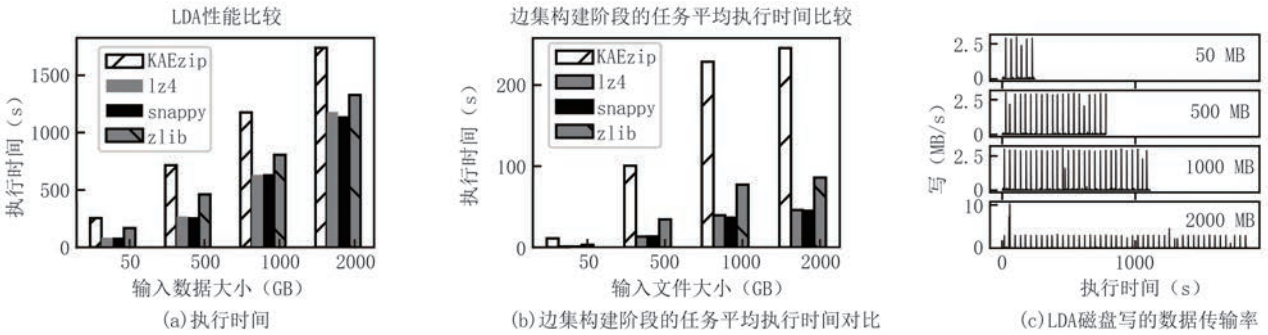

图20　LDA性能比较与分析

根据公式(9)，一个任务的执行时间表示：

$$T = max\{\frac{data\_size}{DPR}, \frac{data\_size'}{P_{comp}}, \frac{data\_size'*ratio}{P^{w}_{disk}}\}$$

其中data_size是输入数据的大小，data_size′是产生的边集大小.

由于具有相同输入数据，四种压缩算法下的第一部分的执行时间是一样的，最大的差异在第二部分和第三部分. 为了便于比较压缩算法对任务执行时间的影响，T可以进一步简化为：

$$T = max\{\frac{data\_size'}{P_{comp}}, \frac{data\_size'*ratio}{P^{w}_{disk}}\}$$

图20(c)展示了在KAEzip下LDA执行期间磁盘写的数据传输率，其中一个峰值表示一次Shuffle写的数据量. 可以看出磁盘写的平均速度远低于磁盘的写峰值200 000 KB/s. 根据上述公式可知，影响任务平均执行时间的主要因素是压缩算法的压缩性能，而不是磁盘的写性能. 因此5.2.2节描述的KAEzip的数据膨胀问题正是导致其压缩性能落后的根源.

关于导致KAEzip发生数据膨胀的根源请详见5.2.2节.

6.3.5　SQL Join

SQL Join测试程序主要链接两个表然后返回符合条件的记录，它主要包含四个阶段：阶段0和1分别从HDFS中读取并重新划分小表和大表的数据，然后将结果写到文件中，其中大表的数据集从2 GB逐渐增加至55 GB，对应小表是其大小的三十分之一；阶段2是读取前两个阶段的输出数据然后进行链接操作；阶段3则是输出结果. 前三个阶段均是CPU密集型.

图21(a)展示了KAEzip下四个阶段的执行时间，其中阶段1和2占据了SQL Join的大部分执行时间. 这两个阶段在其他压缩算法下的执行时间与该图类似，本文不再展示. 图21(b)～(c)展示了阶段1和2的任务在不同压缩算法下的平均执行时间. 以zlib下任务的平均执行时间为基准，其他压缩算法下的任务平均执行时间与该基准的差距均小于5%. 因此压缩算法对它们的性能影响并不显著.

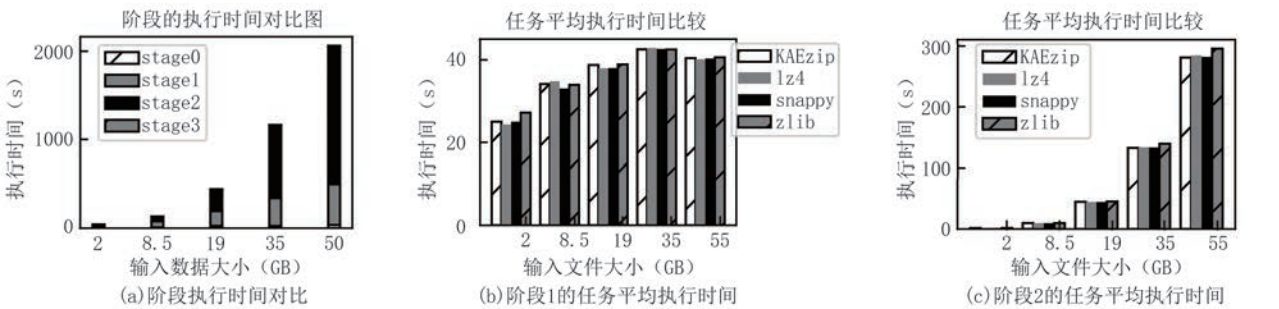

图21　SQL Join性能比较

阶段1的任务执行时间可由公式(9)表示. 由于在阶段1的执行过程中并未发生大量的磁盘写操作，因此公式(9)可以简化为：

$$T_{task} = max\left\{\frac{data\_size}{P^{r}_{disk}}, \frac{data\_size}{DPR}, \frac{data\_size'}{P_{comp}}\right\} \quad (9'')$$

图22展示了SQL Join在KAEzip下的磁盘读的数据传输率，其他压缩算法下的数据传输率与之类似. 首先从该图可以看出数据传输峰值主要出现在阶段0和1的执行过程中. 由于该峰值远低于磁盘数据传输率的极值200 000 KB/s，因此磁盘不是影

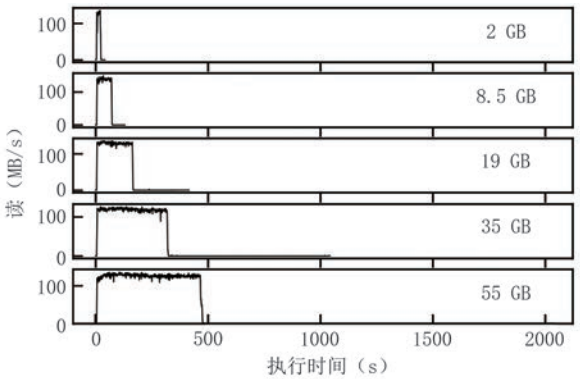

图22 SQL Join在KAEzip下的磁盘读性能

响任务执行时间的关键因素. 阶段1主要是对数据进行重新划分,这并不会导致数据增加,因此划分后的任务输入数据大小一般仍然小于默认值——128 MB. 图21(b)指出任务的平均执行时间均大于20 s,这远大于任一压缩算法压缩128 MB数据的时间开销. 因此CPU是影响阶段1的任务执行时间的关键因素,这也是四种压缩算法对任务性能影响不显著的根源. 同理,CPU对阶段2的性能有类似的影响.

6.3.6 KAEzip和硬件压缩算法的优化

上述实验表明在大数据环境下,磁盘的数据传输率和CPU的处理能力是制约Spark下KAEzip性能的关键因素,其实质均是数据供给率低于KAEzip的压缩或解压缩速度. 在此环境下,压缩算法的压缩性能和解压性能由该供给率决定,而与压缩算法自身性能无关. 因此仅仅提升KAEzip的压缩性能并无法有效提升Spark应用程序的性能. 为此,可以从三个方面进行优化:

(1)优化压缩算法在Spark中的工作机制,设计一种多层缓存机制,增加待压缩或解压缩的数据量,进而充分利用KAEzip的压缩与解压缩性能,提升Spark应用程序的性能;

(2)融入异步机制,使得数据的处理与压缩或解压缩异步执行,从而可最大程度地利用KAEzip的性能优势;

(3)4.1节中的图4(a)表明Spark中数据的压缩往往具备间断性特征,因此从微观尺度(100毫秒以内)上讲,这类压缩具有显著I/O操作特性. 因此可以借鉴相关的I/O调度优化方法[28,29]优化鲲鹏920 CPU对KAEzip的调度,通过减少对KAEzip的无效调度,降低它在压缩数据时的CPU开销和能耗. 这些优化策略不仅是我们将来的研究内容之一,也同样适用于其他硬件压缩算法.

6.4 三层架构的实验评估

如5.2所示,本文提出的三层架构具有较好的普遍性,可以广泛用于支持采用软硬件分离设计的硬件压缩算法在Spark环境下的应用,而并不仅仅限于KAEzip. 为了避免硬件压缩算法对三层架构的影响. 本节首先在x86平台下以zlib压缩软件测试它的开销,具体软硬件实验环境如表1所示. 然后在x86和鲲鹏920平台下,分别评估测试程序性能的提升幅度,并通过对比揭示三层架构对KAEzip的影响. 最后测试三层架构对解决KAEzip数据膨胀问题的有效性.

6.4.1 三层架构的开销评估

在三层架构中,第一层主要是创建数据流,第二层主要以拷贝方式缓存压缩和解压缩数据,而第三层主要是调用zlib动态库中的压缩与解压函数. 该架构的主要开销来自于第二层---数据拷贝. 因此对它的开销评估包括两个部分:(1)评估第二层的开销,(2)将三层架构与zlib压缩库进行比较,以评估三层架构的开销所产生的收益,即提升压缩与解压缩性能的幅度.

Spark sort在数据处理中不会导致数据减小,其所有的输入数据都会被压缩. 因此该程序可以较好测试三层架构的性能开销. 320 MB输入数据下sort的实验结果如下图23所示,展示了三层架构的开销与收益:压缩中的拷贝时间占据整个压缩时间的0.6%,解压缩中的拷贝时间则占据解压时间的1.9%,但是三层架构分别减少了3%和9.6%的压缩和解压时间.

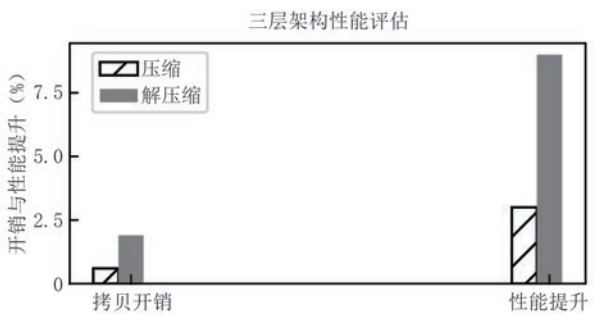

图23 三层架构开销与优势

6.4.2 三层架构的性能评估

(1)基于x86 64位平台的性能评估

表2展示了测试基准程序①和数据集大小,仍以

① 由于压缩算法在SQL Join与sort中的执行流程和数据量的变化都类似,本节的实验并未包含SQL Join测试程序.

表2　测试数据大小

测试程序	small	large	huge
sort	320 MB	3.2 GB	32 GB
wordcount	320 MB	3.2 GB	32 GB
KMeans	1 GB	10 GB	30 GB
LDA	100 MB	250 MB	660 MB

zlib库下的执行时间为基线，它们在三层架构下的执行时间与基线的比值如图24所示. 可以看出，LDA在三层架构下的性能领先zlib大约25%，这是因为LDA在执行中触发了大量的压缩与解压缩调用. 如4.1节所述，250 MB的原数据大概触发了247 851 186次压缩调用，最终产生了不可忽略的开销. 但是融入缓存机制的三层架构能避免这类开销，从而有效提升LDA的执行性能. wordcount和KMeans在三层架构下的性能与zlib下的性能基本相当，这是因为它们的阶段主要由reduceByKey产生，从而导致Shuffle写的数据量很小. 比如，6.3.2节指出wordcount的Shuffle写的数据量大约为14 KB. 这缩小了三层架构提升性能优势的幅度. 最后，该架构对sort性能提升可忽略不计，其根源是sort中待压缩的输入数据较大，其平均值大约为4 KB，使得三层架构无法大幅提升压缩性能，如图23所示，最终导致它无法有效提升sort性能。

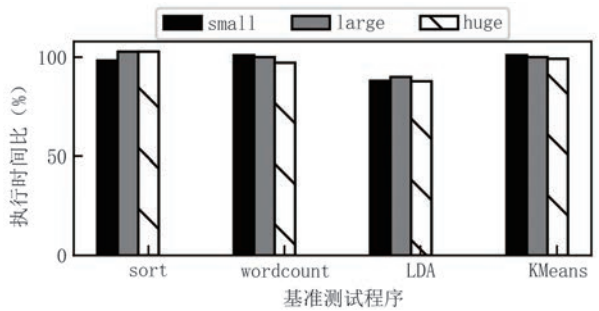

图24　三层架构与zlib的性能比较

(2)基于鲲鹏920平台的性能评估

6.4.1节的实验表明三层架构中基于缓存的压缩机制可以有效提升压缩与解压缩性能. 本节在开启和关闭该机制这两种环境下，重新运行表1中的测试程序，通过对比实验结果评估该机制对KAEzip的影响. 实验结果表明除了LDA，其他工作负载的实验结果基本相同，本文不再展示.

LDA的实验结果如图25所示，其中图25(a)展示了LDA的边集构建阶段在不同压缩算法下产生的Shuffle写数据量. 实验结果表明：在输入数据为50 MB时，优化方法使得在KAEzip下的Shuffle写

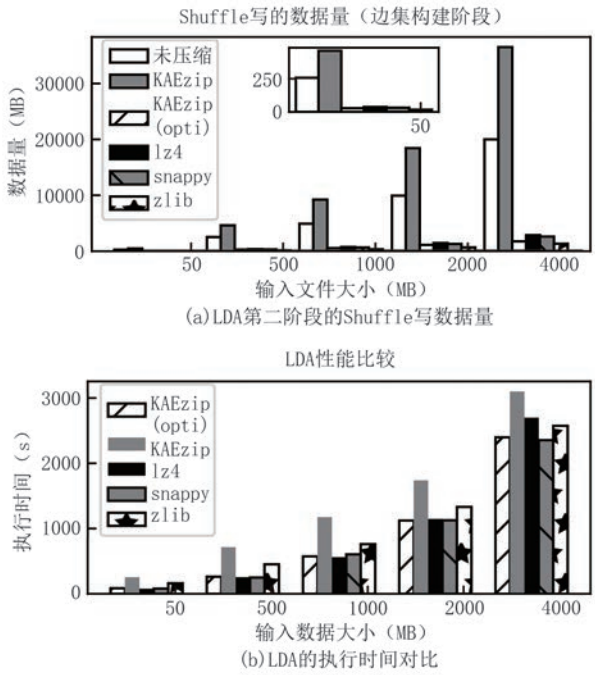

图25　LDA在三层架构下的性能测试

的数据量从原先的489.2 MB减少至28.2 MB，有效地消除KAEzip导致的数据膨胀问题，并且压缩比低于lz4和snappy但仍高于zlib. 当输入数据逐步增至4 GB时，该方法均表现出相同性能.

图25(b)展示了LDA在不同压缩算法下的执行时间. 首先当输入数据从50 MB增至4 GB时，与未优化相比，优化方法使得LDA在KAEzip下的执行时间分别缩短了200%至29.2%. 其次，在输入数据为50 MB时，Shuffle写的数据量大约为250 MB，优化后的方法使得LDA在KAEzip下的执行时间落后snappy大约6.1%但领先zlib大约47.5%. 当输入数据增至500 MB时，Shuffle写的数据量增至大约2.5 GB，KAEzip与snappy之间的性能差距缩小至大约5.5%，但仍领先zlib大约42%. 这是因为相比较其他压缩算法，snappy通过牺牲压缩比提高压缩与解压缩速度，在对小量数据进行压缩和解压缩时具备较好的整体性能优势. 当输入数据增至1000 MB时，Shuffle写的数据量增至大约5 GB，在此情况下压缩比开始成为影响解压缩性能的一个重要因素，这能给KAEzip带来性能提升. 因此LDA在KAEzip下的执行时间开始领先snappy大约5.1%. 随着输入数据增至2000 MB和4000 MB，影响LDA执行时间的重要因素变为负责执行大量数据计算的CPU资源而不是压缩算法，因此LDA在KAEzip和snappy的执行时间大致相当，并且领先zlib的性能优势逐渐缩小至大约9%.

(3)三层架构在x86与鲲鹏920上的性能对比

reduceByKey操作能极大地减少数据量,所以压缩算法对包含这类操作的Spark应用程序的性能并无显著影响,比如wordcount和KMeans. 因此三层架构并无法有效提升运行于x86和鲲鹏920平台上的该类应用程序的性能. sort中压缩函数的平均输入数据较大,削弱了三层架构对压缩与解压缩的性能提升幅度,导致该架构并无法有效提升sort在x86和鲲鹏920平台上的性能. 但是三层架构能最多提升x86平台上的LDA应用程序大约25%的执行性能,但该架构能最多提升鲲鹏920平台下的LDA大约47.5%的执行性能,可见消除数据膨胀的KAEzip能有效提升LDA的执行性能. 此外,在6.1节与6.2节对数据进行直接压缩与解压测试中,三层架构并未对这类环境下的压缩或解压缩采用任何优化策略,所以KAEzip的性能优势完全来自于其本身.

(4)三层架构解决数据膨胀问题的有效性评价

如5.2.3节所示,三层架构中基于缓存的压缩机制可有效地解决KAEzip在LDA下引发的数据膨胀问题. 本节通过实验测试分析三层架构下Shuffle文件的压缩比,以全面评估该机制对数据膨胀的有效性. 实验结果如图26所示,图26(a)表明基于缓存的压缩并未改变sort、wordcount和KMeans的Shuffle文件压缩比,但是显著地降低了LDA的Shuffle文件压缩比,有效解决了KAEzip压缩时的数据膨胀问题. 其根源如图26(b)所示:缓存的引入将LDA下KAEzip的压缩函数的平均输入大小从7.2个字节提升至大约7700字节.

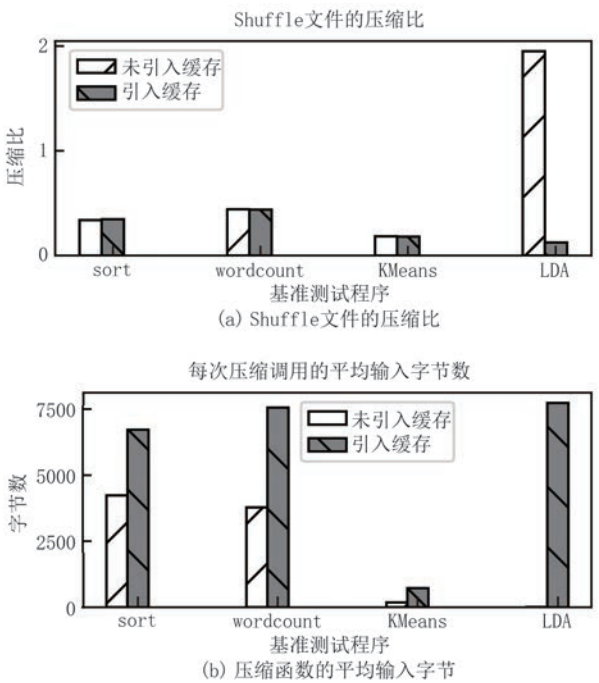

图26 三层架构对数据膨胀的测试

7 总结与展望

已有的研究表明压缩算法的硬化可以有效提升压缩与解压缩性能,但是目前主流的大数据软件均无法有效识别支持这类压缩算法,比如Spark. 这阻碍了对Spark环境下硬件压缩算法的性能进行研究。为此,本文首先提出一种基于"生产-消费"模型的Spark任务的性能模型,形式化地描述压缩算法与Spark任务性能之间的内在联系,分析确定Spark下影响压缩算法性能的关键因素. 其次,本文提出一种三层架构支持Spark识别利用硬件压缩算法. 在此基础上以KAEzip为实验对象、基于任务性能模型和广泛的实验,研究评估KAEzip在Spark下的压缩性能,并分析揭示出影响制约硬件压缩算法性能的根源. 这些研究不仅利于KAEzip和鲲鹏920 CPU的持续改进与完善,也能用于其他硬件压缩算法的优化.

另一方面,广泛的实验测试表明KAEzip在压缩极小量数据流时会导致数据膨胀,为此,本文基于三层架构提出一种基于缓存的压缩与解压缩方法. 实验表明该方法可以有效地解决KAEzip引发的数据膨胀问题,并最多可提高KAEzip大约200%的压缩性能.

我们下一步仍计划以KAEzip为实验对象,研究硬件压缩算法在大数据存储领域中的压缩/解压缩性能与可能的缺陷,其中包括这类压缩算法与重要存储格式——ORC和Parquet——之间的性能关系. 希望这些研究能为CPU中的硬件压缩算法在大数据领域中的全面应用奠定基础.

致 谢 感谢戚广鸿和尹博文对部分实验的验证与实验结果的分析.

参 考 文 献

[1] Xia Jing, Cheng Chuanning, Zhou Xiping, Hu Yuxing, PeterChun. Kunpeng 920: the first 7-nm chiplet-based 64-Core ARM SoC for cloud services. IEEE Micro, 2021, 41(5): 67-75

[2] LiY, et al. A self-aware data compression system on FPGA in Hadoop. //Proceedings of the International Conference on Field Programmable Technology (FPT). Queenstown, New Zealand,

2015：196-199

[3] KovacKyle. A Hardware Implementation of the Snappy Compression Algorithm. Technical Report at UC Berkely, USA. Technical Report No：UCB/EECS-2019-85，2019

[4] ZahariaMatei，ChowdhuryMosharaf，FranklinMichael J.，ShenkerScott，StoicaIon. Spark：cluster Computing with working sets//Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud′10. Boston，USA，2010：1-7

[5] BartíkM.，UbikS.，P. Kubalik. LZ4 compression algorithm on FPGA//Proceedings of the IEEE International Conference on Electronics，Circuits and Systems (ICECS). Cairo，Egypt，2015：179-182

[6] snappy，https://google.github.io/snappy/ 2019,4,18

[7] DeutschP.，GaillyJ. -L. RFC1950：ZLIB compressed data format specification version 3.3. USA，RFC Editor，1996

[8] lzbench，https://morotti.github.io/lzbench-web/ 2022

[9] Compression and decompression performance testing of KAEzip，https://compare-intel-kunpeng. readthedocs. io/zh_CN/latest/accelerator.html (in chinese)
(KAEzip 加解压测试，https://compare-intel-kunpeng.readthedocs.io/zh_CN/latest/accelerator.html)

[10] Wang Yi-Chao，Chen Jin-Kun，Li Bin-Ruiet al. An empirical study of HPC workloads on huawei kunpeng 916 Processor//Proceedings of the 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS). Tianjin，China，2019：360-367

[11] Ivanov，T，Pergolesi，M. The impact of columnar fifile formats on SQL-on-Hadoop engine performance：a study on ORC and Parquet. Concurrency Computat Pract Exper. 2020，32(5)：e5523 1-31.

[12] Rattanaopas，K. A performance comparison of Apache Tez and MapReduce with data compression on Hadoop cluster//Proceedings of the 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)，Nakhon Si Thammarat，Thailand，2017：1-5.

[13] lz4.https://github.com/lz4/lz4. Accessed：2019-4-18.

[14] LuL.，HuaB.. G-Match：A fast GPU-friendly data compression algorithm. //Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications，Zhangjiajie，China，2019：788-795.

[15] Qian W.，et al. Research progress on privacy protection techniques in big data computing environments. Chinese Journal of Computers. 2022，45(4)：669-701.
(钱文君等. 大数据计算环境下的隐私保护技术研究进展. 计算机学报,2022,45(4)：669-701.)

[16] ZhangQ.，LiuL.，PuC.，DouQ.，WuL. and ZhouW.. A comparative study of containers and virtual machines in big data environment. //Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). San Francisco，USA，2018：178-185.

[17] Ruan B.，Huang H.，Wu S.，Jin，H. A performance study of containers in cloud environment. //Proceedings of the Asia-Pacifific Services Computing Conference. Zhangjiajie，China，2016：343-356.

[18] Zhu，and HanChangpeng，Bo and Zhao，Yinliang. A comparative performance study of spark on kubernetes. Journal of Supercomputing. 2022，78(11)：13298-13322.

[19] Fu，TangZ.，YangL. and LiuC.. An optimal locality-aware task scheduling algorithm based on bipartite graph modelling for Spark applications. IEEE Transactions on Parallel and Distributed Systems. 2020，31(10)：2406-2420.

[20] C. Puet al. The millibottleneck theory of performance bugs，and its experimental verifification. //Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). Atlanta，USA，2017：1919-1926.

[21] LaiC. -A.，KimballJ.，ZhuT.，WangQ. and PuC.. MilliScope：a fine-grained monitoring framework for performance debugging of n-tier web services//Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS).Atlanta,USA,2017:92-102

[22] Zhu，Changpeng，Bo Han，ZhaoYinliang. A bi-metric autoscaling approach for n-tier web applications on kubernetes. Frontiers of Computer Science,2022，16(3)：1-12

[23] LuSiyang，Xiang Wei，RaoBingbing，TakByungchul，Long Wang，and WangLiqiang. LADRA：log-based abnormal task detection and root-cause analysis in big data processing with Spark. Future Generation Computer Systems.2019,95:392-403

[24] WangK.，KhanM. M. H.. Performance prediction for Apache Spark platform//Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications. NewYork，USA，2015:166-173

[25] HeX.，ShenoyP.. Firebird：network-aware task scheduling for spark using sdns//Proceedings of the 2016 25th International Conference on Computer Communication and Networks (ICCCN). Waikoloa，USA，2016:1-10

[26] Zhan Jianfeng,Gao Wanling,Wang Lei，et al.BigDataBench：an open source big data system evaluation benchmark. Chinese Journal of Computers,2016,(1):196-211. (in Chinese)
(詹剑锋,高婉铃,王磊等.BigDataBench:开源的大数据系统评测基准.计算机学报,2016,(1):196-211.)

[27] L. Wanget al. BigDataBench：a big data benchmark suite from internet services//Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). Orlando，USA，2014：488-499

[28] Huang et al. Enhancing proportional IO sharing on containerized big data file systems. IEEE Transactions on Computers,2021,70(12):2083-2097.

[29] M. Kachmaret al. A smart background scheduler for storage systems// Proceedings of the 2020 28th International Symposium on Modeling，Analysis，and Simulation of Computer and Telecommunication Systems (MASCOTS)，Nice，France，2020:1-8

[30] Zhao Yinliang，Zhu Changpeng，Han Bo，etc. Calculus using fitness testing for method redirection. Journal of Software. 2013,27(4):1495-1511 (in Chinese)
(赵银亮、朱常鹏、韩博等.一种利用适合性测试支持方法重定向的演算。软件学报. 2013,27(4):1495-1511)

ZHU Chang-Peng, Ph. D., lecturer. His research interests focus on bigdata processing and virtual machine.

TANG Jing-Ren, M. S. His research interests focus on hardware-based acceleration engine and high performance computing.

LIANG Yun, B. S. His research interests focus on hardware-based acceleration engine and high performance computing.

ZHANG Xiao-Chuan, M. S., professor. His research interests focus on computer games and software engineering.

HAN Bo, Ph. D., senior engineer. His research interests focus on information processing and software engineering.

ZHAO Yin-Liang, Ph. D., professor. His research interests focus on high performance computing, parallel computing and programming language desgin.

Background

Over the last few decades, Spark has emerged as a widely used platform for large-scale data processing and analysis. However, processing a massive amount of immediate data in Spark often leads to excessive I/O overhead. To mitigate this issue, Spark has incorporated serval compression algorithms to reduce the size of data for better performance. Currently, mainly compression algorithms, for example, LZ4, LZF, Snappy, and ZSTD, each with its own tradeoffs between compression ratio, compression speed, and decompression speed. This is because maximizing speed and minimizing ratio are often conflicting goals, leading to different pros and cons for each algorithm, also signifying that the performance benefits of compression algorithms for Spark applications remain an open issue.

Traditional compression algorithm researches mainly focus on the optimization and improvement of algorithms themselves, ignoring the support from hardware, for example CPU and GPU. KAEzip is a hardware accelerator for compression embedded in the first 7-nm ARM-based 64-bit multi-core CPU, Kunpeng 920. Recent research efforts indicate that hardware-based compression algorithms have significant performance advantages over traditional ones.

However, most of the crucial foundational software in big data fields cannot recognize and leverage them.

This paper presents a formal description of the relationships among compression, multi-dimension hardware resources, and execution times of Spark tasks. And then it extracts what crucial factors have influence on compression in Spark. Afterwards, a three-layer architecture on top of Spark is proposed to enable Spark use hardware-based compression algorithms. The layered architecture is quite easy to be extended for optimizing their performance in Spark and is conveniently reused in other big data software.

With extensive experiments, our evaluation illustrates that hardware-based compression algorithms can bring significant performance upgrades; however, data transmission speed of disks and data process capability of CPU have a decisive influence on the performance of compression algorithms in Spark, not matter if they are software-based compression algorithms or hardware-based ones. What's more, our evaluation also indicates that hardware-based compression algorithms may cause data inflation problem, and then a software-based solution can be represented to solve it.

This research work not only promotes the optimization and evolution of KAEzip and Kunpeng CPU but also benefits hardware-based compression algorithms in other CPUs in the future, to achieve better performance and lower overhead.

This work is supported by Huawei Non-recurring Engineering Project (No. OAA21091100464724D), China Scholarship Council (No. 201708505099) and National Natural Science Foundation of China under Grant(No. 61702063)