中图分类号：TP319

论文编号：10006ZY1606519

北京航空航天大學

硕士学位论文

大数据应用程序性能分析技术的研究与实现

作者姓名　　周红刚

学科专业　　计算机系统结构

指导教师　　李云春　教授

培养院系　　计算机学院

Research and Implementation of Performance Analysis Technique for Big Data Applications

A Dissertation Submitted for the Degree of Master

Candidate：Zhou Honggang

Supervisor：Prof. Li Yunchun

School of Computer Science & Engineering

Beihang University, Beijing, China

中图分类号：TP319

论文编号：10006ZY1606519

硕 士 学 位 论 文

大数据应用程序性能分析技术的研究与实现

作者姓名　　周红刚　　申请学位级别　学术硕士

指导教师姓名　李云春　　职　　称　　教授

学科专业　　计算机系统结构　　研究方向　　并行计算

学习时间自　2016 年　9 月　1 日　起　至　2019 年　1 月　15 日　止

论文提交日期　2018 年　11 月　30 日　　论文答辩日期　2018 年　12 月　10 日

学位授予单位　北京航空航天大学　　学位授予日期　2018 年　1 月　15 日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是由本人在指导教师的指导下独立进行研究所取得的成果，论文中的有关资料和数据是真实的。尽我所知，除文章中加标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对本论文做的任何贡献均已在文中的致谢部分做出了明确的说明。

若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名：_____ 日期： 年 月 日

学位论文使用授权书

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。

保密学位论文在解密后的使用授权同上。

学位论文作者签名： 日期： 年 月 日

指导教师签名： 日期： 年 月 日

指导教师签名： 日期： 年 月 日

摘 要

伴随着互联网和人工智能的兴起，大数据在工业生产和学术研究中扮演着越来越重要的角色，然而至今没有一套成熟的性能分析工具，大数据应用程序的性能分析和调优目前还依赖于大数据框架提供的简单性能诊断工具，这些工具无法系统地反映大数据应用程序在复杂调用关系中关键的性能瓶颈，并且依赖于应用程序开发人员的经验来定位低效行为以及执行相应的优化。本文主要面向 Spark 应用程序，在应用层面框架层面和分布式文件系统层面提出了一套系统性的插桩工具和低效行为分析方法，将大数据应用程序在不同层面的性能问题以及导致这些性能问题的原因自动诊断出来，方便应用程序开发人员以及框架开发人员提升大数据应用程序的性能。

本文以 Spark 为例阐述了大数据应用程序运行过程中的分层关系以及不同层面可能出现的性能瓶颈。在应用层面，本文主要提出了基于统计分析的慢任务原因检测算法，通过搜集集群不同节点资源占用信息、应用运行时 JVM 信息、任务处理数据量信息、数据混洗信息、任务序列化和反序列化信息等，通过统计方法识别导致慢任务产生的原因，针对不同的特征，本文提出了诸多约束条件来确保根原因分析（root-cause analysis）的精确率。在框架层面，本文提出了基于 Byteman 的 Spark 插桩方法以及低效算子评分算法，可以高效识别低效算子，并通过动态获取不同算子的 JVM 特征并基于 Kmeans 分析低效算子产生的原因。在分布式文件系统层面，本文提出了基于 HTrace 的插桩方法，通过追踪 I/O 请求在不同节点之间的延迟以及 I/O 处理的数据量大小，可以精确地定位性能瓶颈，此外，本文提出了针对分布式文件系统的调用树压缩算法，可以高效地压缩性能轨迹文件的大小，便于可视化。

本文自上而下地构建了分层的大数据应用程序的性能分析系统，完整地刻画了低效行为及其产生的原因，从资源占用到框架设计，从参数配置到源代码，从 JVM 到组件耦合等。在工程和方法上都有一定的创新性，提出了资源监控、动态插桩、静态插桩相结合的工程设计，提出了针对大数据的慢任务根原因分析方法，Spark 算子低效评分方法和基于 Kmeans 的算子低效特征挖掘算法等。

关键词：大数据, 性能分析, 插桩，根原因分析

Abstract

With the prevalence of Internet and Artificial Intelligence, big data has been playing a more and more important role. So far, there is no mature performance analysis tool. The performance analysis and tuning of big data applications is still very dependent on the performance diagnostic tools provided by the big data framework. These tools cannot systematically reflect the key performance bottlenecks of big data applications under complex call relationships. What's more, they rely heavily on the experience of application developers to locate inefficient behavior and perform appropriate optimizations. This paper focuses on Spark applications, and proposes a set of instrumentation tools to diagnose inefficient behavior at the application level, framework level and distributed file system level. The performance problems of big data applications at different levels and the reasons behind them are automatically diagnosed, making it easy for application developers and framework developers to improve the performance of big data applications.

This paper takes Spark as an example to illustrate the hierarchical relationship in the running process of big data applications and the possible performance bottlenecks at different levels. At the application level, this paper mainly proposes a straggler root-cause analysis algorithm based on statistical methods, by collecting cluster resource utilization information, runtime JVM information, data skew information, data shuffle information, task serialization and deserialization information, etc. and applying statistical methods to identify the root-cause feature. For different features, this paper proposes different constraints to ensure the accuracy of root cause analysis. At the framework level, this paper proposes a Byteman-based Spark instrumentation method and Inefficiency Score (IS) to obtain inefficient operations, and dynamically obtain the JVM features of different operations based on Kmeans to analyze the root-causes of inefficient operations. At the distributed file system level, this paper proposes an HTrace-based instrumentation method. By tracking the delay of I/O requests and the amount of data processed by I/O, the bottleneck can be automatically located. In addition, this paper proposes a call tree compression algorithm for the distributed file system can efficiently compress the size of traces.

This paper builds a hierarchical performance analysis system for big data applications from top to bottom, completely portraying inefficient behaviors, including resource utilization, framework design, parameter configuration, source code, JVM features, components coupling, etc. There are certain innovations in engineering and methods. The engineering design of resource monitoring, dynamic instrumentation and static instrumentation is proposed. The straggler root-cause analysis method for big data, Spark operation inefficiency scoring method and Kmeans-based operation inefficient feature mining algorithm are proposed.

Key words: Big Data, Performance Analysis, Instrumentation, Root-Cause Analysis

目 录

第一章　绪论

大数据是现代信息社会发展到一定程度的产物，数据量随时间成指数增长，目前基本上各行各业都出现了大数据的需求，包括搜索、电子商务、社交、金融、新闻、医疗等，可以说有信息的地方就有大数据。大数据不同于传统的数据库应用，数据库一般记录的是静态的信息，只需要支持快速检索就足够了，但是大数据则全然不同，大数据一般需要大量的数据之上进行计算来发掘潜在的价值，而且往往比传统服务器端或者客户端程序更加耗时，更重要的是这些程序往往会重复使用，所以分析大数据程序的性能瓶颈是十分迫切的需求。

本章主要阐述课题的研究背景和研究意义，分析国内外相关研究的现状，提出本课题的研究目标，最后介绍论文的组织结构并进行小结。

1.1 选题背景及意义

近二十年间随着计算机与通讯技术的飞速发展，社会的数字化变革，互联网的蓬勃发展和数码产品的普及，数据的产生和传播速度也越来越快，人类积累的各类数据规模急剧膨胀，呈现爆炸式增长趋势。无论是在生物信息、能源勘探、航空航天、武器装备等科学研究和工程领域，还是在电子商务、教育、媒体、金融、医疗、社交、娱乐等信息服务领域，各类应用产生和使用的数据量巨大。例如，Google 搜索引擎，每天提供超过 2 亿次的网页查询服务，搜索网页超过 81 亿张，图片超过 10 亿张，需要的存储容量超过 5PB；欧洲原子能研究机构的大型粒子对撞机（LHC）每秒钟产生 100MB 原始数据，每年的数据量为 15PB。据国际数据公司 IDC 的统计，2008 年底全球数据内容总量为 487EB，2011 年这一数据增长到 1.8ZB，预计到 2020 年，全世界的数据总量将达到惊人的 40ZB。对这些庞杂的海量数据进行有效的规约、整合、存储和利用，不但关系到经济和社会科学发展的持续性、高效性和稳定性，而且与社会公共服务质量和效率、进而与最末端的社会生活密切相关，对于国计民生具有十分深远的影响。

数据增长的规模已经远超人们的想象，大数据时代悄然影响着人类社会的各个方面。大数据承载了复杂的应用，蕴含着丰富的价值。数据量越大，性能问题越应该得到关注，

常规的大数据程序往往运行时间比较长，而且很多程序会周期性运行，比如搜索引擎会周期性地更新网页库以抓取新的内容，大数据监控平台会周期性地从日志里面挖掘有用的信息，对这些经常运行的大数据程序进行性能分析可以有效地指导性能优化。

大数据应用程序一般运行在分层的框架之上，应用程序需要提交给大数据计算框架如 Hadoop、Spark[2] 、Dryad[3] 等，而应用程序的 I/O 则要通过访问分布式文件系统。和传统的单机应用程序的性能分析不同，大数据应用程序的性能分析需要同时考虑应用程序本身、集群资源的影响、框架的影响和分布式文件系统的影响。除此之外，大数据应用程序低效行为的表现方式和传统的应用程序有所不同，传统的应用程序更偏重于指令层面和冗余计算的识别和优化，而大数据应用程序的性能瓶颈主要集中集群计算能力的不均衡或者计算量分配的不均衡，所以提出一套新的性能分析方法。

准确来说，大数据应用程序的性能分析主要是识别应用程序的低效行为及其背后的原因，在不同层面，低效行为识别和根原因分析有着不同的方法，因此本文偏重有所不同。比如在应用层面，本文更偏重低效行为根原因分析，这是因为应用层面低效行为已经有了明确的定义，而在根原因分析方面却没有成熟的方法和研究；在框架层次和分布式文件系统层面，本文更偏重低效行为的定位，因为这两个层次的低效行为背后的原因往往是框架本身的设计或者特定应用程序相关的。

在不同层面的性能分析和改进对大数据应用至关重要的作用，有效的性能分析是优化大数据应用程序必要的和十分关键的步骤，而改进和优化则要根据性能分析的结果具体考虑。本文主要从三个层面进行大数据应用程序的性能分析，任务层面、框架层面和分布式系统层面。在应用层面，我们考虑集群的资源特征、任务序列化和反序列化的特征、JVM 垃圾搜集特征、任务处理数据量特征、任务混洗的特征等。通过对资源特征的分析，我们可以识别不同应用程序对不同类型资源占用的需求，进而优化调度系统或者应用配置提升性能；通过对任务序列化和反序列化特征、JVM 垃圾搜集特征的低效行为定位，我们可以优化应用程序的压缩方式和 JVM 堆内存的配置；通过对任务处理的数据量和数据混洗低效行为的识别，我们可以优化数据分配来改进应用程序的性能。

针对框架层面，由于 Spark 对 Map-Reducr 大数据编程范型支持较好，并且支持图计算模型，具备较大的灵活性，便于研究框架内部性能，所以本文我们主要针对 Spark 平台。通过基于 Byteman 的动态插桩，我们可以获取算子粒度的性能数据，通过从中发

掘低效行为，我们可以找出低效算子，反馈给框架开发人员，开发更加高效的框架，或者有针对性地优化应用程序源代码（算子可以定位到具体的源代码）。

针对分布式文件系统，由于 HDFS[1] 是最常用的分布式文件系统并且和 Hadoop、Spark 兼容性较好，本文针对 HDFS 研究分布式文件系统的瓶颈。通过基于 HTrace 的插桩，我们可以跟踪分布式文件系统请求的调用在各个节点上的延迟，得到不同类型的访问请求下 HDFS 的低效函数，进而可以有针对性地采取优化措施。

通过更加完整地刻画大数据应用程序在不同层面的低效行为，本文揭示了应用程序在资源优化、配置优化、框架优化、分布式文件系统优化等各个层面的潜力，可以有效地指导优化的方向。本文提出的性能分析系统相比于以往的工作有以下几个优点：适用性广泛，性能问题的覆盖面更广。

1.2 国内外研究现状

国内对大数据应用程序性能分析工具的研究大多只是停留在应用配置和资源监控层次，没有深入分析框架的内部信息和低效行为背后原因。李萌[4] 等人提出了一套基于 Hadoop 的自动化性能分析工具，该系统主要由 Java 和 Python 实现，分为 Profiler 模块，Optimizer 模块、What-If Engine 模块，其中，Profiler 模块基于 BTrace 获取配置和应用程序性能的关系，What-If Engine 模块负载对配置参数和应用性能进行建模，预测应用程序的性能，而 Optimizer 模块利用递归随机搜索法对最佳配置参数进行优化。作者认为 Hadoop 的应用程序性能主要由四个因素决定：Hadoop 执行的应用程序、应用程序的输入、Hadoop 集群的资源、Hadoop 的配置参数。其中应用程序无法进行自动优化，数据输入也无法优化，Hadoop 的集群资源无法优化，只有 Hadoop 配置是可以优化的，所以作者只考虑了 Hadoop 配置的影响，所以作者提出的性能分析系统无法满足对大数据应用程序性能分析的需求。

Weichen Qi[5] 基于决策树对应用程序的低效行为的根原因进行了分析，并对 Spark 的低效算子进行了研究。该文认为通过决策树训练一个从大数据应用特征到慢任务的分类器，经过一定的剪枝和泛化，可以根据右枝分析得到哪些特征是导致慢任务的原因，但是这种方法有一定的缺陷，就是决策树的有一定的不稳定性，如果数据稍有变化，得到的决策树可能完全不同，但是该文第一次提出根据信息熵判定根原因，具有一定的创

新性。其次，该文利用动态插桩得到了 Spark 算子粒度的性能信息，但是只是分析了哪些算子执行时间较长，并未进行深入地分析。

除了在计算层次的分析，也有文献针对分布式文件系统进行了性能分析。杨斌[6] 为了解决生产环境下 I/O 性能检测以及 I/O 冲突干扰分析困难，该文在神威太湖之光上开发了 Beacon，Beacon 可以采集太湖之光上所有机器的 I/O 数据，包括前端节点、代理节点、存储节点和元数据服务器，对数据进行了清洗和压缩，按照作业类别给出报告，最后该文也针对性地提出了优化方法，比如修改源代码、改变 I/O 模式、改变请求优先级、代理节点资源的再分配、存储节点的异常检测和移除等。Beacon 在多层次 I/O 提取的时候选择了文件系统插桩，在前端进行 I/O 性能数据进行压缩，将连续的顺序读写进行压缩，然后在后端对 I/O 数据进行采样，为了还原完整的 I/O 路径，论文还对应用提交日志进行记录和分析，然后将应用和 I/O 行为联系起来。Beacon 的可视化支持系统在任意时间段内 IOPS 的查询、带宽查询、应用查询以及用户查询。Beacon 在建立 I/O 请求和应用的对应关系的时候没有考虑到并发的因素，而且 Beacon 没有对分布式 I/O 请求进行追踪，只是测量单个节点的 I/O 性能。

国外的有很多研究集中在大数据应用程序的推断执行上，如果一个任务执行时间过长，那么就在另外一个空的计算节点上启动同样的任务，这样在某个任务执行失败或者特别慢的情况下，任务可以在别的节点快速执行完成。MapReduce 框架一开始就把推断执行设计成默认的机制，并验证该技术可以提升 44%的应用程序响应时间[7] 。后来的研究对这种简单的推断执行方法提出了改进。M. Zaharia 等人针对异构集群提出了 LATE 模型来改进推断执行的效率[8] 。现有的云计算环境一般都是动态分配的虚拟主机，多个虚拟主机肯能会共用一台物理主机，会相互竞争资源，所以不同虚拟主机计算资源有着较大的差别，如果不考虑不同主机计算能力的差别，推断执行就会变得低效，计算能力较差的节点一般会更慢地完成任务，而其他任务必须等待这些慢任务的完成。Hadoop 推断执行没有考虑到集群的异构性，如果在低效节点执行推断任务，不仅不能加快应用程序执行时间，还会大量占用集群资源，干扰正常节点的执行。LATE 会动态确定不同主机的计算能力，计算每个任务的执行速度，预测每个任务完成时间，将推断执行任务发送到最快完成任务的节点上运行。然而，这种推断执行面临着低效的风险，如果一个任务执行慢是由于数据倾斜导致的（任务处理的数据量比别的任务大），推断执行并不能

加快应用程序的执行。Mantri **错误!未找到引用源。**首先提出了用慢任务原因定位的方法改进推断执行的效率。Mantri 认为造成任务执行时间过长的原因有一下几种：处理器、内存等资源竞争，硬盘损坏，网络拥塞，负载不均衡。Mantri 据此提出了原因感知的推断执行模型，根据需要重启任务，保存任务输出，根据网络情况部署任务。Mantri 提出了任务完成时间估算模型和重启任务完成时间估算模型，如果重启任务会缩短任务完成时间，就在新的节点重启任务。Mantri 还通过测量上行和下行的网络流量来决定任务如何放置，确保集群网络拥塞最小化。在多种优化的共同作用下，Mantri 部署在微软 Bing 集群后应用程序运行速度提高了 32%。Dolly [10] 提出了一种新的方法改进小程序的执行时间。Dolly 完全克隆小程序并采用延迟分配的方式解决克隆带来的数据冲突。Dolly 对小程序加速了 34%到 46%，只消耗了 5%的额外资源。小程序一般交互性比较强，只包含几个小任务，对响应时间要求比较严格，但是这些小程序更容易产生慢任务。这些小程序在商业集群上占的比例也特别高，据估计，在 Facebook 的 Hadoop 集群和 Microsoft Bing 的 Dryad 集群上，分别有超过 80%和 60%的程序包含的任务小于十个。但是程序克隆会带来严重的数据冲突，先完成的任务会同时发送数据给多个其他任务会占用大量的带宽，Dolly 综合了两种策略来解决这个问题，一种策略是只从一个克隆组发送数据到另一个克隆组，一种是不考虑数据冲突，所有克隆组都从最先完成的任务中拿数据。Dolly 让所有任务都等待一定的时间，如果在等待时间内任务能够独占数据，就从上一个任务获取数据，如果超时还不能独占数据，就采用竞争的策略。

这种推断执行的策略都有同样的缺点，无效的推断执行会造成资源竞争，E. Bortnikov 等人[11] 估计在生产环境的集群下 90%以上的推断执行是浪费的。他们提出了用梯度提升树的方法来预测任务的执行时间，采用的特征包括任务特征和硬件特征。任务特征包括应用粒度特征和数据倾斜特征，应用粒度特征包括 mapper 和 reducer 的数量、输入格式、数据压缩方式等，数据倾斜特征包括任务处理的数据量，任务数据的数据量。硬件特征又包括主机硬件特征，网络特征，Map-reduce 特征，主机硬件特征包括 CPU 占用率、内存占用率、磁盘占用率、I/O 带宽、网络带宽、TCP 连接的数量、JVM 线程数、本地和远程数据请求数等，网络特征包括机架内流量、机架间流量、累积流量等，Map-reduce 特征包括占用的任务槽数量等。

后来出现了一些离线分析的方法研究造成慢任务的原因。[12] 采用了新的指标来定

位慢任务，即 Degree of Straggler（DOS），该指标即反映了任务处理数据的速度，排除了因数据倾斜的情况。他们定位慢任务之后计算慢任务和不同特征之间的相关性来定位慢任务的原因，他们最后得到的结论是 59%的慢任务是因为 CPU 占用率过高，42%的慢任务是因为磁盘负载过大，34.3%的慢任务是由于数据请求处理缓慢。

还有研究综合了在线分析和离线分析进一步提升性能瓶颈的定位准确度。[13] 采用离线日志分析来计算集群的任务特征，确定在线分析慢任务的判定的阈值，在线分析确定慢任务并采用推断执行来改进效率。其中离线分析部分包括应用采样，慢任务定位，阈值确定三个部分，应用采样负责对应用和任务的模式进行建模，一般任务的执行进度是不均衡的，可能出现一部分运行较快，一部分运行较慢的情况，最后对任务的不同阶段进行线性拟合；慢任务定位负责统计慢任务的类型和影响；阈值确定负责确定正常任务的执行速度，如果某些任务执行速度小于该阈值，就被定义为慢任务，采用推断执行等方法加速这些任务的执行。

传统的性能分析工具主要是针对 MPI 的。[14] 提出了基于 MPI 的插桩工具 TAU，成为并行计算性能分析领域常用的分析工具。TAU 包含三层：插桩层，测量层和分析层，其中每一层都为用户提供了一些可以自定义的模块。插桩层支持不同层次的插桩，即源代码层，编译层和运行层。TAU 的分析和可视化层同样支持不同组件的组合，并且不同的进程分开进行数据呈现。Intel Pin 是一个常用的动态插桩工具，[15] 提出了基于 Pin 但是可以对应到源代码的动态插桩工具 CCTLib。Pin 不能记录插桩函数的调用信息，提供函数的调用信息便于进行更细粒度的诊断信息，CCTLib 支持指令粒度的调用信息获取，而且可以把这些指令对应到源代码，而且 CCTLib 造成的额外负载很小，就算对于长时间运行的程序，也能进行指令粒度的插桩。

大数据负载是进行性能分析的必要条件。**错误!未找到引用源。**提出了弹性测试集的模型来拟合真实的生产环境下的 I/O 特征并在阿里云上进行了测试。常见的云平台文件系统包括 GFS[23] ，HDFS 和 Ceph，这些文件系统的优化面临异构集群和异构应用的挑战，论文分析了 2500 个节点的阿里云为期两周的 I/O 日志，得到了不同应用的负载和数据分布，请求到达时间，请求的数据大小，数据分布情况等的规律，提出了 Porcupine 测试集，Porcupine 具备可扩展性和高效性，拟合了真实生产环境下的应用 I/O 特点。文件大小对文件系统提出了不同的要求，对于大文件，找到连续可用区域是至关重要的考

验，而对于小文件，查找性能才是最关键的，因为小文件一般数量较大，元数据的查找成为性能瓶颈。阿里云存放着超过 2 亿文件，单个文件的大小从 0B 到 2TB 之间分布，超过 50%的文件小于 256KB，超过 90%的文件小于 64MB（hdfs 文件块的大小）。数据本地性是影响文件系统最关键的因素之一，文件本地性越好，网络负载就越小并且访问延迟越低，Hadoop 集群超过 80%的数据都在本地[18]，但是阿里云的数据本地性更低，对于读请求，数据在本地的比例只有 46%，对于写操作，数据在本地的比例有 80%，以前的测试集没有考虑到这种数据本地性，所以无法准确测试系统的 I/O 性能。

1.3 课题来源与研究内容

本课题来源于国家科技部的国家重点研发计划课题，“大数据存储性能评测的新理论与新方法”。本课题的研究目标之一，是针对大规模大数据应用程序，设计一套能够实现细粒度性能分析与性能瓶颈挖掘的性能分析工具，以指导大数据存储系统上的应用性能优化。

本文选用 Spark 和 HDFS 作为目标的大数据平台，主要研究在大数据应用程序在不同层面的低效行为。本文的层面划分是以 Spark 应用程序运行的系统结构为基础的，主要分为应用层面、框架层面和分布式文件系统层面。不同层面在进行性能分析的时候都有不同的方法和评价指标，其主要内容如下：

（1） 应用层面的慢任务根原因分析

应用层面主要是指 Spark 将用户应用程序划分成不同的任务，应用层面的性能分析就是分析任务的低效行为（慢任务）产生的原因，该层面的分析不涉及插桩或者低效行为识别。Spark 将图计算划分成若干个算子，将连续的窄依赖算子合并成一个阶段，然后将一个阶段分成若干个任务在不同的 Executor 上并行执行，同一个阶段的不同任务会到达一个 barrier，执行较快的任务会等待执行较慢的任务，产生严重的性能瓶颈，本文抽取系统资源特征、任务序列化和反序列化的特征、JVM 垃圾搜集特征、数据处理量特征、数据混洗特征等，定位慢任务产生的根原因。应用层面的低效行为分析不需要插桩，Spark 会自动记录每个任务运行情况并写入日志，但是系统资源情况需要手动启动脚本进行资源监控和日志处理。应用层面的低效行为主要分析配置低效、资源低效、数据低效、JVM 低效等。本文在这个层次最主要的工作内容包括针对不同的特征提出不同的根

原因分析方法，提升精确率；在受控的实验环境下验证了算法的有效性；针对特殊的根原因提出了相应的优化算法。

（2） 框架层面低效算子识别与根原因分析

应用层面的性能分析不涉及应用程序和框架本身的低效性，框架层面的性能分析就是为了弥补这个不足，该层面的性能分析结果既可以精确定位用户源代码的低效之处，又能识别 Spark 本身可能的低效设计。任务内部被划分成许多算子，但是本文发现慢任务中并不是所有的算子都运行更慢，运行较快的任务也会包含运行较慢的任务，所以定位低效算子较为困难。同时，算子更多地和用户源代码关联，所以我们采用 JVM 特征作为低效算子的潜在根原因。Spark 本身不记录算子层次的性能数据，所以我们需要插桩获取。最后我们得到的低效算子相关的 JVM 根原因也可以指导 JVM 的配置优化。本文在这个层次最主要的工作内容是提出基于 Byteman 动态插桩方法和动态 JVM 性能数据抽取方法；提出低效算子的评分用于识别低效算子；提出基于 Kmeans 的根原因分析方法；

（3） 分布式文件系统层面的低效函数分析

Spark 应用程序运行离不开分布式文件系统，框架层次的性能分析只能黑盒地获取分布式系统文件系统的性能信息，不能获取分布式文件系统内部低效行为，也就很难进行相应的优化。本文选择 HDFS 作为分布式文件系统的代表进行低效行为识别，HDFS 分为 NameNode，DataNode 和客户端节点，一个 I/O 请求会涉及到 NameNode 访问、DataNode 访问、DataNode 和客户端节点之间的数据交互、DataNode 和 NameNode 之间的交互等。I/O 请求的分布式追踪是识别 HDFS 低效行为非常关键的部分，本文采用基于 HTrace 的插桩进行分布式追踪，提出了高效的调用树压缩算法，在不同负载下识别 HDFS 的低效行为，可以用于指导 HDFS 的优化。本文在这个层次最主要的工作内容是改进了 HDFS 的插桩，增加了很多插桩函数，解耦了网络延迟和本地 I/O 延迟，在日志文件中加入了关键的参数，便于分析数据处理速率和发包异常；提出了调用树的压缩方法，可以极大地节省存储空间，便于可视化和低效行为定位；研究了不同大数据负载下 HDFS 瓶颈的变化。

1.4 论文组织结构

根据本文的研究目标和研究内容，本文的其余部分将依次介绍大数据程序性能分析相关技术，应用层面的低效行为根原因分析应用层面，框架层面性能分析，分布式文件系统性能分析。具体的组织结构如下：

第二章，相关技术分析。该部分介绍了常见的大数据系统的计算架构，大数据系统的存储架构，大数据应用程序的低效行为的表现方式，大数据应用程序分析的一般方法，本文用到的插桩工具介绍。

第三章，应用层面的性能分析。该部分介绍了应用层面特征抽取的方法和实现，针对不同特征提出的根原因分析方法，在受控实验环境下对根原因分析方法的验证，针对不同的大数据负载进行瓶颈分析，针对特殊的性能瓶颈提出优化方案。

第四章，框架层的性能分析。该部分介绍了框架层次的插桩方法、低效算子评分方法、基于 Kmeans 的根原因分析方法、典型低效算子模式分析、不同大数据负载下低效算子根原因分析。

第五章，分布式文件系统的性能分析。该部分介绍了 HTrace 的基本用法、本文对 HTrace 的扩展、HTrace 插桩点的介绍、HDFS 轨迹压缩算法介绍、HDFS 在不同负载下低效行为分析。

第六章，系统实现。该部分介绍了整个性能分析系统各个模块的详细实现。

总结与展望：概括全文，总结论文完成的工作及研究成果。

1.5 小结

本章介绍了本文的背景意义以及国内外相关技术的研究现状，在此基础上提出了本文的研究目标和研究内容，最后对论文的组织结构和各章节的研究内容进行了概述。

第二章 相关技术分析

本章介绍了常见的大数据系统的计算架构，大数据系统的存储架构，大数据应用程序的低效行为的表现方式，大数据应用程序分析的一般方法，本文用到的插桩工具介绍。

2.1 典型大数据框架概述

传统的单节点应用程序无法适应大规模数据处理，单节点数据处理即受限于存储空间也受限于计算能力。而传统的高性能计算框架如 Message Passing Interface（MPI）也不适用于大数据，一方面是由于 MPI 编程过于繁琐，也没有较好的分布式文件系统支持，所有的数据部署和计算都要手动编程指定，另一方面是由于 MPI 容错机制不够好，当集群中有一个节点出故障时，整个应用程序都无法继续运行。正由于传统的编程泛型的不足，Google 在 2008 年提出了 Map-Reduce 编程泛型，极大地简化了大数据应用程序的编程工作，后面 Hadoop 成为 Apache 的一个开源框架变得渐渐流行起来，Hadoop 底层自带分布式文件系统 HDFS，在生产环境和实验室都有较为广泛的应用。

2.1.1 MapReduce 编程范型

MapReduce 最早是 Google 研究提出的一种面向大规模数据并行处理的方法，MapReduce 程序由 map 函数组成，它执行过滤和排序和 reduce 函数，它执行规约操作。MapReduce 通过在集群上并行地运行各种任务，管理系统各部分之间的所有通信和数据传输以及提供冗余来协调处理和容错。

该模型是用于数据分析的拆分-转化-组合策略的专业化。它的灵感来自于函数式编程中常用的 map 和 reduce 函数，尽管它们在 MapReduce 框架中的用途与它们的原始形式不同。MapReduce 框架的关键贡献不是实际的 map 和 reduce 函数，但是可扩展性和容错性通过优化执行引擎实现了各种应用程序。因此，MapReduce 的单线程实现通常不会比传统（非 MapReduce）实现更快;任何收益通常只能在多线程实现中看到。只有当优化的分布式混洗操作（降低网络通信成本）和 MapReduce 框架的容错功能发挥作用时，才能使用此模型。优化通信成本对于良好的 MapReduce 算法至关重要。MapReduce 库已经用许多编程语言编写，具有不同的优化级别，支持分布式 shuffle 的流行开源实现是 Hadoop 以及后来支持内存计算的 Spark 的一部分。MapReduce 这个名称最初是指专有

的 Google 技术，但后来被通用化了。

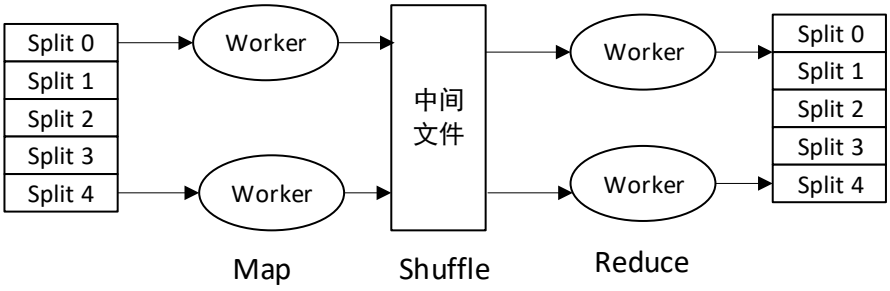


图 1 MapReduce 示意图

MapReduce 框架通常由三个步骤组成：

(1) Map：每个计算节点将映射函数应用于本地数据，并将输出写入临时存储。主节点确保仅处理冗余输入数据的一个副本。

(2) Shuffle：计算节点根据输出键（由 Map 函数生成）重新分配数据，以便属于一个键的所有数据都位于同一个计算节点上。

(3) Reduce：工作节点现在并行处理每个键的每组输出数据。

MapReduce 允许分布式处理 Map 和 Reduce 操作。只要每个 Map 操作独立于其他 Map 操作，映射可以并行执行；实际上，这受到独立数据源的数量和每个源附近的 CPU 数量的限制。类似地，只要将同一个 Key 分配给同一个 Reducer，也可以将 Reduce 函数并行化，这样可以极大地并行化处理数据。并行性还提供了在函数执行期间从服务器或硬盘的部分故障中恢复的一些可能性：如果一个 Mapper 或 Reducer 发生故障，则可以重新执行计算。

2.1.2 Hadoop 框架

Hadoop 是一组开源软件实用程序，使用 MapReduce 编程范型为大数据的分布式存储和处理提供了一个软件框架。最初设计用于从廉价硬件构建的集群，但现在还被用于高端硬件集群。Hadoop 中的所有模块都设计有一个基本假设，即硬件故障是常见现象，应该由框架自动处理。

Hadoop 的核心包括一个存储部分，称为 Hadoop 分布式文件系统（HDFS），以及一个数据处理部分，它支持 MapReduce 编程模型。 Hadoop 将文件拆分为数据块，并将它们分布在集群中的节点上。然后，它将打包的代码传输到节点中以并行处理数据。这种

方法利用了数据局部性，其中每个节点处理他们被分配的数据，这使得数据集的处理速度更快，效率更高。基本的 Apache Hadoop 框架由以下模块组成：

(1) Hadoop Common：包含其他 Hadoop 模块所需的库和程序；

(2) Hadoop 分布式文件系统：一种分布式文件系统，可在商用机器上存储数据，在整个群集中提供非常高的聚合带宽；

(3) Hadoop YARN：于 2012 年推出，是一个资源管理平台，负责管理安排用户的应用程序;

(4) Hadoop MapReduce：用于大规模数据处理的 MapReduce 编程范型的实现。

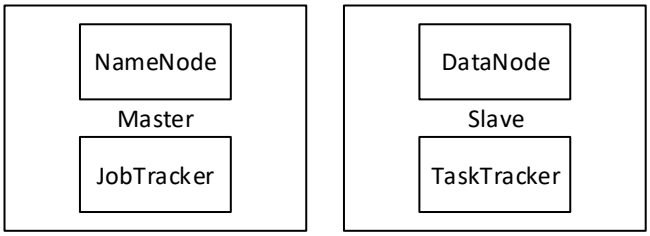


图 2 Hadoop 计算系统架构示意图

Hadoop 框架本身主要使用 Java 编程语言编写，其中一些本机代码用 C 编写，命令行实用程序编写为 Shell 脚本。虽然 MapReduce 经常采用 Java 代码，但任何编程语言都可以与“Hadoop Streaming”一起使用来实现应用的“map”和“reduce”部分。

2.1.3 Spark

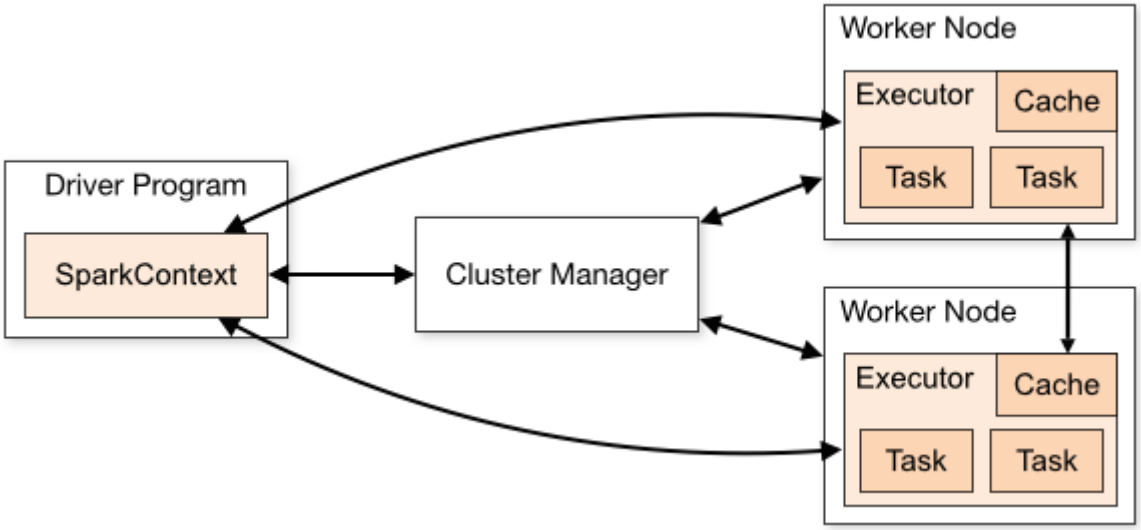


图 3 Spark 架构示意图[注1]

Apache Spark 是一个开源的分布式通用集群计算框架。最初由加州大学伯克利分校的 AMPLab 开发，Spark 代码库后来被捐赠给 Apache 软件基金会，该基金会从那时起就一直在维护它。Apache Spark 的架构基础是弹性分布式数据集（RDD），它是分布在集群上的只读数据集，以容错方式维护。在 Spark 1.x 中，RDD 是主要的应用程序编程接口（API），但是从 Spark 2.x 开始，即使 RDD API 没有被弃用，也鼓励使用数据集 API，RDD 技术仍然是数据集 API 的基础。

Spark 及其 RDD 是在 2012 年开发的，以响应 MapReduce 集群计算范例的局限性，该范例强制分布式程序上的特定线性数据流结构：MapReduce 程序从磁盘读取输入数据，在数据中映射函数，减少结果映射，并将减少结果存储在磁盘上。Spark 的 RDD 作为分布式程序的工作集，提供受限的分布式共享内存。Spark 有助于实现迭代算法，在循环中多次访问其数据集，以及交互式数据分析，即重复的数据库式数据查询。与 MapReduce 实现相比，此类应用程序的延迟可能会减少几个数量级，迭代算法类中的机器学习系统的训练算法是开发 Spark 的最初动力。

Apache Spark 需要集群管理器和分布式存储系统。对于集群管理，Spark 支持独立（本机 Spark 集群），Hadoop YARN 或 Apache Mesos。对于分布式存储，Spark 可以与

注 1：http://spark.apache.org/docs/latest/cluster-overview.html

各种各样的接口，包括 Hadoop 分布式文件系统（HDFS），MapR 文件系统（MapR-FS），Cassandra，OpenStack Swift，Amazon S3，Kudu 或可以实现自定义解决方案。Spark 还支持伪分布式本地模式，通常仅用于开发或测试目的，不需要分布式存储，可以使用本地文件系统；在这种情况下，Spark 在一台机器上运行，每个 CPU 核心有一个执行程序。

与 Hadoop 和 Storm 等其他大数据和 MapReduce 技术相比，Spark 有如下优势：

(1) Spark 提供了一个全面、统一的框架用于管理各种有着不同性质（文本数据、图表数据等）的数据集和数据源（批量数据或实时的流数据）的大数据处理的需求

(2) Spark 通过在内存中缓存数据，可以将 Hadoop 集群中的应用运行速度提升 100 倍，由于 Spark 自身的优化，甚至应用程序在磁盘上的运行速度也能提升 10 倍

(3) Spark 支持更加灵活的编程模式，Spark 除了支持 MapReduce 编程范型，还支持几十个算子操作。

Spark 的 RDD 是保存于内存或者磁盘中的一种高效数据结构，RDD 一般会划分成很多分区，方便并行地进行多种类的映射操作，同时 Spark 将 RDD 组织成有向无环的依赖图，RDD 可以缓存在内存中，也可以动态计算。 DAG 计算模型在开始执行程序前会对数据流进行依赖分析，RDD 的依赖分为宽依赖和窄依赖。宽依赖是指 RDD 依赖于其他多个分区，计算前需要进行比较耗时的 shuffle 操作，而窄依赖则不需要其他分区，所以其父节点 RDD 不需要进行同步。

Spark 会根据宽依赖将一个应用程序划分成若干个阶段（Stage），虽然通过最大化内存操作，减少了不必要 I/O，但是阶段之间会进行混洗操作，这是 Spark 的一个比较耗时计算。同时，Spark 仍然存在因为不同任务负载不均衡导致的低效行为。任务负载不均等原因导致的额外等待开销。

由于 Spark 是一种比较全面的框架，甚至可以说是 Hadoop 的一个超集，包含了独立的调度系统、兼容多种分布式文件系统、实现了复杂的计算范式、提供了多种高效的函数库、支持更多的大数据应用，因此本文以 Spark 为例进行性能分析。

2.1.4 HDFS

HDFS 是一个用 Java 编写的用于 Hadoop 框架的分布式、可扩展且可移植的文件系统。Hadoop 集群名义上包括一个 NameNode 和若干个 DataNode，但由于 NameNode 的重要性，一般会设置冗余节点，例如 SecondaryNameNode。每个 DataNode 使用特定于

HDFS 的块协议在网络上提供数据块，文件系统使用 TCP / IP 套接字进行通信，客户端使用远程过程调用（RPC）相互通信。

HDFS 跨多台计算机存储大文件，它通过跨多个主机复制数据来实现可靠性，因此理论上不需要主机上的独立磁盘冗余阵列（RAID）存储，数据节点可以相互通信以重新平衡数据，移动副本以及保持高数据复制。 HDFS 不完全符合 POSIX，因为 POSIX 文件系统的要求与 Hadoop 应用程序的目标不同。另一个原因是为了提高数据吞吐量，并支持非 POSIX 操作，例如 Append。

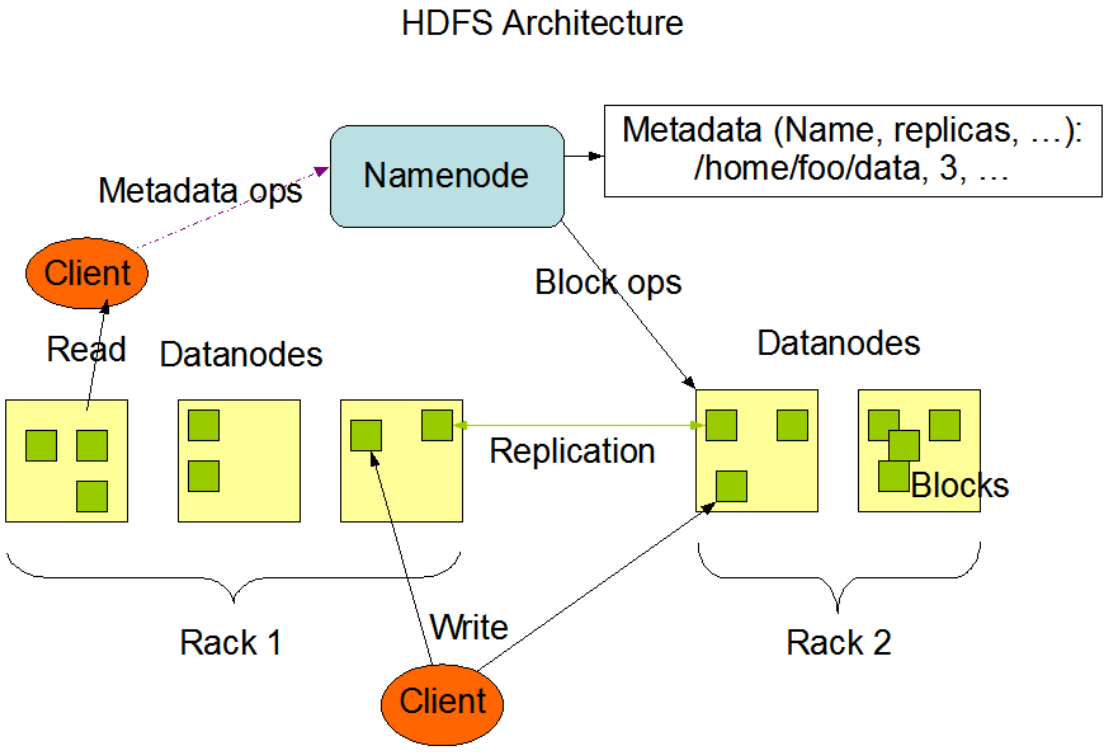


图 4 HDFS 的架构图[注1]

2.2 Spark 大数据程序性能影响因素

Spark 应用程序的运行环境较为复杂，影响其性能的主要因素包括以下 5 个类别：

（1） 应用配置：Spark 支持很多配置参数，Spark 启动时也可以指定特定的 JVM 参数，这些参数如果配置不合理会造成应用程序非常低效，比如，如果采用复杂度较高的任务压缩算法，会导致任务序列化和反序列化时间过长；如果

注 1：http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html

给 JVM 分配了较小的堆内存，会导致 JVM 垃圾收集时间过长，影响 Spark 应用程序的性能。这种性能瓶颈通常可以在应用层面的性能分析中被检测出来。

（2） 资源占用：商业集群上经常会同时运行很多计算任务，很容易出现集群资源分配不均衡的情况，这通常会导致 Spark 有些任务执行过慢，这些资源占用包括 CPU 占用、I/O 时间、网络传输的数据量。在分析这些资源占用的时候，我们需要排除应用自身运行导致的资源占用过高的情况，这种性能瓶颈通常可以在应用层面的性能分析中被检测出来。

（3） 应用程序源代码：通常来说一个 Spark 应用程序要重复运行很多次，所以从源代码层次寻找低效行为是十分有必要的。Spark 的设计使得算子和应用程序源代码对应起来，当我们能够定位低效算子的时候，也直接定位了低效的源代码，这种性能瓶颈通常可以在框架层次的性能分析中被检测出来。

（4） 数据倾斜：几乎所有的大数据应用程序在某个阶段都要执行数据流并行的计算，如果此时不同节点分配的数据量大小不同，则会导致执行时间的差异，因而产生性能瓶颈，这种性能瓶颈通常可以在应用层面的性能分析中被检测出来。

（5） 框架设计：大数据框架和分布式文件系统的设计也会存在一些不足导致应用程序较为低效，通过识别框架本身在不同负载下的性能瓶颈，可以有效指导框架的优化和改进，这种性能瓶颈通常可以在框架层次和分布式文件系统层次的性能分析中被检测出来。

2.3 插桩工具

为了获取大数据应用程序的性能信息，需要对框架进行插桩（即在框架的特定位置插入相应的代码获取时间信息、运行时信息、资源信息等）。本节主要介绍本文用到的插桩工具以及选择这些工具的原因。

2.3.1 Byteman

本文采用基于 Byteman 的插桩方法获取 Spark 算子的性能信息，Byteman 是一个灵活的动态插桩工具，用于 Java 或基于 Java 的语言，如 Scala。它使用 ECA（条件时间行为）规则脚本语言来构建插入 Java 的代码片段和控制它的执行。Byteman 代理可以在应用程序启动时加载或加载到正在运行的应用程序中，并且可以在应用程序继续运行时加

载、卸载或重新加载规则。Byteman 的限制是它只能在应用程序的某个位置进行插桩，并且不能保存调用堆栈信息。Byteman 的最大优点是所有的测试代码都可以在不修改框架或应用程序源代码的情况下动态加载。

本文选用 Byteman 的原因是对 Spark 算子进行性能分析不需要知道算子在执行时的调用堆栈，同时，由于 Byteman 动态部署的特点，极大方便了插桩代码的调试，也保证了我们的性能分析工具具备可扩展性。本文在对算子进行插桩的时候还会采集 JVM 的信息，Byteman 支持自定义插桩辅助函数，所以本文通过修改插桩辅助函数，使得算子在执行和结束执行的时候能够捕获 JVM 的性能信息。

2.3.2 HTrace

常见的分布式文件系统的性能轨迹信息获取工具包括 Dapper[24] ，Magpie[27] ，Stardust[25] ，Xtrace[26], HTrace 等。HTrace 是 Apache 基金会下的一个分布式追踪框架，支持多种语言和分布式框架，不同于上述提到的其他几个分布式框架，HTrace 已经被写入 Hadoop 的标准实现，不需要额外的插桩就能获取基本的插桩数据，而且 Hadoop 在进行远程过程调用的时候已经将 HTrace 的信息写入包头，可以还原跨节点的函数调用关系，基于 HTrace 扩充性能分析的功能便于兼容现有的 Hadoop 版本，基于以上考虑，本文选取 HTrace 作为性能分析工具的基础。

HTrace 是一个用于分布式跟踪的开源框架。它可以与独立应用程序和库一起使用。HTrace 设计用于大型分布式系统，例如 Hadoop 分布式文件系统和 HBase 存储引擎。但是，没有关于 HTrace 是一个通用的库，也可以用于其他的分布式系统。HTrace 中包含几个重要的分布式插桩和追踪的概念，包括 Span，TraceScope，Tracer。Span 是指被插桩函数的时间长度，包含以毫秒为单位的开始时间，结束时间，描述以及许多其他字段。Span 有父子关系，类似堆栈。每个 Span 都有一个唯一的 128 位 ID。 由于 128 位数字的空间足够大，HTrace 可以使用随机生成来避免冲突。

TraceScope 对象管理 Span 对象的生命周期。创建 TraceScope 时，通常会附带一个关联的 Span 对象。关闭 TraceScope 时，将所有的 Span 发送到 SpanReceiver 进行处理。HTrace 通过使用线程本地性数据跟踪当前线程中的跟踪作用域是否处于活动状态。通过避免传递上下文对象的需要，这种方法可以更轻松地将 HTrace 添加到现有代码中。

Tracer 是用于创建跟踪 TraceScope 对象的 API。Tracer 还负责对 TraceScope 的采样，

单个进程或库可以包含许多 Tracer 对象。每个 Tracer 对象都有自己的配置。思考 Tracer 对象的一种方法是它们与 log4j 中的 Log 对象类似。与 TraceScope 和 Span 不同，Tracer 对象是线程安全的。在同一个 Tracer 对象上同时调用 Tracer＃newScope 的多个线程是完全可以接受的。

然而，Hadoop 自带的 HTrace 插桩无法满足现有的分布式系统性能分析的需要，主要有以下几点原因。首先，Hadoop 自带的插桩获取的信息比较有限，无法获取更加细节的信息，比如，NameNode 和客户端节点之间通信，DataNode 和 NameNode 之间的通讯，客户端节点和 DataNode 之间的通信，客户端节点和 NameNode 之间的通讯，DataNode 之间的通讯，而 Hadoop 自带的插桩信息只能获取客户端节点和 DataNode 获取性能数据，无法获取其他的性能轨迹信息。其次，默认的插桩信息无法获取更加详细的参数信息，对于分布式文件系统来说，不仅需要每一个函数调用的时间序列，还需要知道每个函数处理的文件数量的大小，便于分析各个组件文件的处理速度。最后，Hadoop 自带的插桩信息无法进行有效的规约、可视化，不便于分析，短短几分钟的运行就会产生数百兆的轨迹文件，难以诊断性能问题。

所以本文针对 HDFS 的特定基于 HTrace 进行扩展，解决上述问题。通过重写插桩区域并在插桩的时候加入关键的参数，获取更多关键的性能信息，计算分布式文件系统各个组件以及组件之间的交互的数据处理速率，找出瓶颈。除此以外，本文在名称节点、数据节点加入测量名称节点的性能、数据节点与其他节点之间交互的性能，将 I/O 层次和网络层次分离开，获得层次化、弱耦合的性能轨迹数据。最后，本文还提出了一种针对分布式文件系统的数据规约方法，包括还原函数调用的树结构、在树内进行压缩、在树间进行压缩，对压缩后的典型树结构进行可视化等，方便开发人员快速定位性能瓶颈。

2.4 性能分析方法

插桩是性能分析的第一步，获取到性能信息之后还要经过进一步的处理才能得到低效行为以及根原因。常见的性能分析方法分为两类，一种是基于统计模型的方法，另一种是基于机器学习的方法。

基于统计模型的方法一般是计算特定的统计指标，然后设定阈值来判定低效行为及其根原因，常见的指标有相关系数，中位数、分位点、平均值。在慢任务识别方法，常用的方法是计算所有任务执行时间的中位数或者平均值[8]，将任务执行时间大于同一个阶段中所有任务中位数或者平均值的任务作为慢任务。对于慢任务的根原因分析，[12]

采用了相关性分析，将和慢任务相关性较强的任务作为慢任务产生的根原因。[9] 采用层次性的根原因分析方法，首先，该方法首先检测任务处理的数据量或者从网络获取的数据量是否太大（与中位数相比），如果是的话，就将此作为慢任务的根原因；否则，检测跨机架的网络流量，如果依然不能解释慢任务，则把原因归咎于机器资源占用率过高或者硬件故障。

基于机器学习的方法一般采用聚类、降维或者分类的算法，如 Kmeans，PCA，决策树等。[5] 采用了决策树进行慢任务根原因诊断，采用的方法是抽取任务执行过程中的特征，然后采用决策树利用这些特征预测一个任务是否是慢任务，根据决策树的特性（C4.5）算法，最后有一个特征将任务分为慢任务和正常任务，将这种特征作为慢任务的根原因。[41] 将 PCA 用于分析 Hadoop 配置和性能的关系，PCA 可以根据方差将高维的数据变成低维的数据，同时尽可能地保留原始数据的信息，通过定位不同的主成分，可以得到哪些特征和应用程序的性能相关。然后，[41] 应用聚类算法对不同的应用程序进行分类（仅仅采用已经分析得到的几个主成分作为特征），方便优化配置。[42]采用聚类的方法分析 Hadoop 产生的日志，传统的日志处理方法一般是让有经验的运维工程师进行关键词搜索，这种方法非常低效，尤其是日志数量众多的时候，通过聚类，可以自动找出出现错误的日志。

2.5 本章小结

本章简要介绍了常见的大数据计算框架和存储框架，介绍了常见的大数据编程范型，为大数据应用程序可能出现的低效行为奠定了基础。其次，本章还介绍了大数据应用程序影响性能的因素，从分层的角度分析了这些影响因素在不同层次的分布。最后，本章介绍了本文用到的插桩工具及其选择这些插桩工具的原因。

第三章 应用层面的慢任务根原因分析

本章讨论如何以 Spark 为例在应用层进行性能分析，一方面应用层面可以说是以 Spark 的任务为粒度进行性能分析，任务是 Spark 进行资源分配和调度的基本单位，是 Spark 根据应用程序的算法自动划分的，所以称之为应用层面；另一方面，从 Spark 系统的设计角度看，Spark 的性能日志基本上都是围绕任务构建，这个层次进行性能分析不需要插桩工作，在工程上具备独立性。本章主要从任务出发分析不同特征对任务运行时间的影响。

3.1 应用层面低效行为描述

Spark 将一个阶段划分为多个任务，如果某些任务慢于同一阶段中的其他任务，整个应用程序的执行会因这些任务（也称为慢任务）而变慢。这些慢任务会显著影响整个应用程序的执行速度，但是传统的性能分析方法没能有效挖掘慢任务背后的原因并针对性地改进。

慢任务产生的原因比较复杂，首先，不同的任务处理不同的数据，数据量越大，任务越慢（也称为数据倾斜）。其次，任务可能用到远程节点中的数据，如果远程节点出现了网络拥塞，就会导致严重的延迟。而且，不同的节点可能有不同的硬件配置，这是导致任务执行时间差异的另一个因素，在阶段结束时发生的混洗操作时也会导致数据倾斜。Spark 的配置也会导致慢任务的产生，比如由特定压缩算法导致的不同任务序列化和反序列化的执行时间差异较大。JVM 的垃圾搜集也会影响任务的执行，JVM 在进行垃圾搜集的时候会暂停所有进程的执行，由于不同机器内存占用不同，当机器内存不足的时，JVM 就会进行垃圾搜集，造成慢任务的产生。最后，不同机器的 CPU、I/O 占用率也会极大地影响，很多文献将 CPU、I/O 作为主要瓶颈并相应地执行推断任务[10] 。

如果可以分析慢任务产生的根原因，则可以通过多种方式来使得任务执行时间更加均衡。例如，如果数据倾斜是根本原因，我们可以调整 RDD 的划分键或将数据分割为更多分区；如果资源竞争是根本原因，我们可以在负载较小的节点上启动推断任务，如果 JVM 垃圾搜集是根本原因，我们可以。以往不同的研究对慢任务有着不同的定义。Mantri **错误!未找到引用源。**定义了低效任务为任务完成时间比同一个阶段中所有任务执行时间的中位数的 1.5 倍。本文采用与 Mantri 相同的定义。

有关慢任务的产生的原因和解决方案，以前的研究可以大概分为两类，一类是在线

预测，一种是离线预测。前者**错误!未找到引用源。**在任务执行的过程中动态动态检测是否有低效行为的发生，当检测到有的任务执行时间比别的任务慢时，则重新在另一个节点重新执行任务（推断执行），在检测的时候，一般会同时分析节点任务执行较慢的原因，如果是任务处理的数据量较大导致的，则不执行推断执行，如果该任务伴随着 CPU 占用率过高，则在 CPU 占用率较低的节点推断执行。这种方法有以下几种缺陷，首先，提前检测慢任务准确率较低，如前所述，影响任务执行时间的因素较多，只有当数据倾斜是导致慢任务的主要原因时，这种检测才比较有效，所以该方法有可能出现以下两种情况，任务在推断执行后提前完成，造成计算资源的浪费；任务在执行后期阶段检测出了慢任务，但是重启任务的时候已经无法加快任务执行。其次，推断执行会造成计算资源的浪费，尤其是在生产集群中，90%以上的推断执行任务都是无效的。后者比较常用方法是利用特征和慢任务和相关性判定慢任务的根原因，这些特征一般包括资源占用、网络堵塞、数据倾斜，这种方法有诸多的缺点，首先，慢任务和特征的相关性较强并不能说明该特征是根原因，对于常见的 CPU 占用率来说，有可能是任务自身的执行造成的；其次，这种相关性分析方法只能宏观地分析哪些特征会导致慢任务，无法进行任务粒度的根原因分析，无法针对具体应用进行分析；其次，相关性分析要求大量的数据，无法有效迁移到新的集群或者应用；最后，相关性分析有较低的召回率，由于慢任务产生的因素比较复杂，很多特征和慢任务之间没有明显的相关性，因而造成该方法准确率较低。

3.2 多维度特征提取

我们选择被广泛采用的系统特征，包括 CPU、I/O、网络流量等资源占用特征以及数据局部性、随机读写字节、读取字节、混洗读写字节 JVM 垃圾收集时间、任务序列化时间和任务反序列化时间等特征。

我们使用 Linux 采样工具收集系统信息，包括 iostat，mpstat 和 sar。用式（3.1）来分别计算系统资源占用特征。其中，$user\_time_t$ 是指用户占用的 CPU 时间，$total\_time_t$ 是指总的时间，IO_time_t 是 I/O 占用的时间，$bytes\_send_t$ 是 t 时间内网卡发送的数据量，$bytes\_received_t$ 是 t 时间内网卡接受的数据量。

$$F_{cpu} = \frac{1}{t_1 - t_0}\sum_{t=t_0}^{t_1}\frac{\text{user\_time}_t}{\text{total\_time}_t}$$

$$F_{disk} = \frac{1}{t_1 - t_0}\sum_{t=t_0}^{t_1}\frac{IO\_time_t}{total\_time_t} \tag{3.1}$$

$$F_{network} = \frac{1}{t_1 - t_0}\sum_{t=t_0}^{t_1}\text{bytes\_send}_t + Bytes\_received_t$$

表 1　数据本地性可能出现的不同情况

本地性	含义
Process Local	数据在本进程的地址空间中
Node Local	数据在本节点其他进程的地址空间中
Rack Local	数据在同一个机架的机器中
Any	数据在其他机架的机器中
Nopref	数据在任何地方都一样

表 2　从 Spark 日志中提取的特征

特征名	定义	解释
F_{read_bytes}	B/B_{avg}	读取数据特征，其中 B 为从磁盘中读取的数据量，B_{avg} 为阶段平均读取的任务量
$F_{shuffle\_read\_bytes}$	B/B_{avg}	混洗读数据量的的大小
$F_{shuffle\_write\_bytes}$	B/B_{avg}	混洗写数据量的大小
$F_{momery\_bytes\_spilled}$	B/B_{avg}	内存溢出数据量的大小
$F_{disk\_bytes\_spilled}$	B/B_{avg}	磁盘溢出数据量的大小
$F_{JVM\_GC\_time}$	T/T_{task}	JVM 垃圾收集占用的时间，其中 T 为垃圾收集占用的时间，T_{task} 为整个任务持续的时间
$F_{serialize\_time}$	T/T_{task}	任务序列化占用的时间
$F_{deserialize\_time}$	T/T_{task}	任务反序列化占用的世界

我们在启动 Spark 的时候，会同时在集群启动性能采样进程，记录采样开始时间戳，每秒钟搜集一次系统资源占用信息，写入日志，当 Spark 应用程序结束运行的时候，调度器在集群节点上杀死采样进程，然后从集群搜集采样日志，通过和 Spark 任务执行时间进行对比，就可以得到任务运行时的资源占用情况。

从 Spark 日志文件中提取的特征反映了慢任务产生的内部原因，如数据倾斜，数据局部性、JVM 垃圾搜集、任务序列化和反序列化、数据本地性等。数据本地性是本文的方法中包含的一个特殊特征，它在 Spark 中有几个状态，如表 1 所示，Spark 可以通过

spark.locality.wait.* 记录这些本地性状态中相应的等待时间。例如，如果当前进程没有可用的槽来处理其本地数据，则它将进入 spark.loaclity.wait.process 状态以等待，直到超时。之后，Spark 将开始进入降级本地性，例如节点级别。我们使用数值来表示任务局部性特征，如公式 3.2 所示。从 Spark 日志中提取的其他特征如表 2 所示。

$$F_{locality} \begin{cases} = 0, & Process_{local} \\ = 1, & Node_{local} \\ = 2, & 其他 \end{cases} \quad (3.2)$$

3.3 根原因分析方法

我们使用的特征可以分为四类，包括离散特征，数值特征，资源特征和时间特征。资源特征和时间特征属于数值特征，但是具备一定的特殊性，需要特殊的约束条件。对于数值特征，当满足方程(3.3)中的条件时，我们认为它是根源特征，其中 $\text{global\_quantile}_{\lambda q}$ 是所有任务中特征值的 λ_q 分位数，λ_q是超参数，用于调节低效行为的阈值，F_{peer} 是同一阶段内的其他任务。第一个条件旨在限制根源特征的绝对值。这是因为即使低效特征的值大于其对等任务，它仍然可以在正常方差内，因此对任务持续时间影响不大。对于时间特征，我们应用附加规则$F > T\lambda_T$。也就是说，该特征的值必须比任务持续时间的 λ_T 倍大。如果时间特征远小于任务执行时间，则该特征不太可能成为根原因。

对于资源特征，我们提出了一种名为边缘检测（Edge Detection）的方法来过滤由任务本身导致的高资源占用率的情况。这个算法的基本思想是在任务开始之前和任务结束之后的一段时间内监视系统资源占用率。如果系统资源占用率在任务开始后上升并在任务结束后下降，我们会将高资源占用率归因于任务本身，因此资源利用率不应被视为根原因。数学上地，如果任务满足公式(3.4)中的条件，那么我们就认为高资源占用率是由任务本身引起的，其中 Mean_t^{head} 和 Mean_t^{tail} 是任务开始之前和任务完成之后 t 时间内的平均资源利用率，λ_e是超参数。任务本地性是本文唯一考虑的离散特征。如果局部性值为 2 并且满足公式**错误!未找到引用源。**中的条件，其中 num（normal_task）是正常任务的数量，我们将局部性作为根原因。整体流程如图 5 所示。

$$F > global\_quantile_{\lambda_q} \quad (3.3)$$
$$F > mean(F_{peer})\lambda_q$$

$$Mean_t^{head}(F_{resource}) > \lambda_e F_{resource} \quad (3.4)$$
$$Mean_t^{tail}(F_{resource}) > \lambda_e F_{resource}$$

$$sum\left(F_{locality}^{normal_{task}}\right) < \frac{num(normal_{task})}{2} \quad (3.5)$$

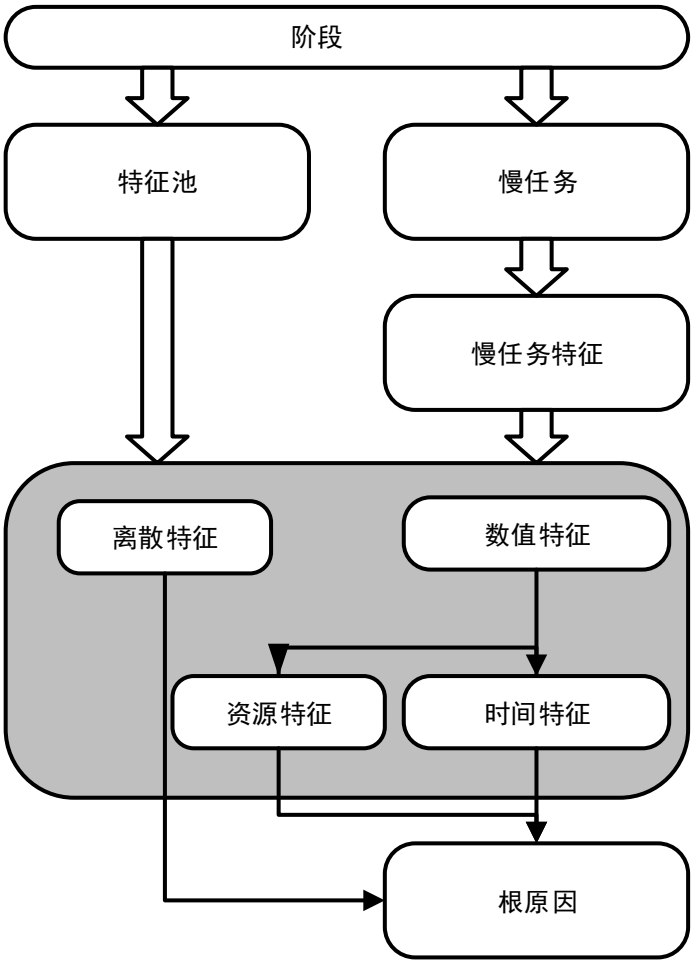


图 5　根原因分析流程

3.4 低效行为根源因验证

3.4.1　实验配置

以前的研究进行性能分析的一个重要缺陷是没有进行相应的验证，本文在受控的环境下验证了算法的有效性。本文的实验主要集中在两方面：1）将本文的算法应用于由高资源利用率产生的低效任务，并利用这些低效任务验证根源分析的准确性。2）分析 Hibench[28] 中具有代表性的大数据负载的慢任务的根原因。我们的实验是在 6 台服务进行的。每台服务器都配备了 Intel Xeon E5-5620，其中包含 16 个核心，32KB 一级缓存，

256KB 二级缓存，12MB 三级缓存，16GB 内存，1Gbps 网络，我们使用 Spark v2.2.0 和 HDFS v2.2.0，一台服务器作为主服务器，其他五台服务器作为从服务器，操作系统是 CentOS v6.5。我们在 Spark 上系统架构如图 6 所示。

调度器负责调度 Spark 作业并触发异常生成器（AG）以生成高 CPU，I/O 和网络利用率。根据调度程序的决定，AG 负责在从节点中启动资源占用生成程序。调度程序等待应用运行完毕，从从节点搜集 Spark 日志、AG 日志、资源监控日志，并将其发送到分析器。分析器从日志中提取性能信息，将它们组织成任务独立的数据结构，然后执行根原因分析算法。下面将详细说明每种系统资源的异常发生器的设计。

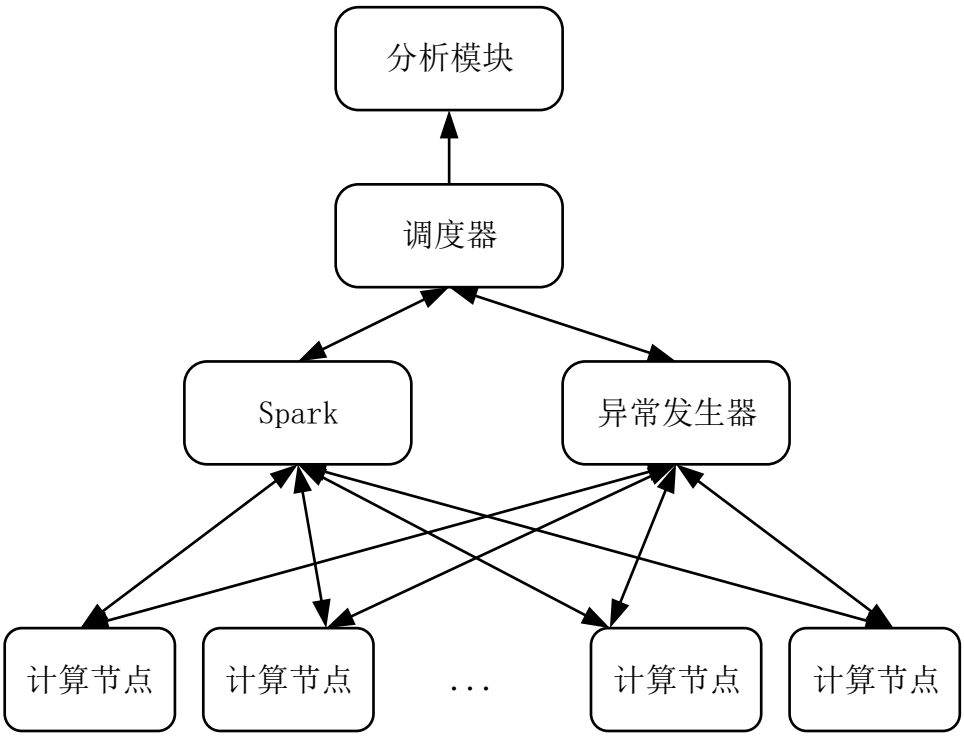


图 6 验证根原因分析方法的系统结构图

1）CPU 异常发生器

CPU AG 生成 1 百万个随机整数，然后对每个数据进行循环操作以模拟计算密集型工作负载。随机将这些数据中的一个元素转储到磁盘，以避免编译器优化。考虑到 CPU 是多核的，我们采用八个进程来同时运行 CPU AG。

2）I/O 异常发生器

对于 I/O AG，我们不断将10^8个字符写入磁盘以模拟 I/O 密集型工作负载。与 CPU AG 类似，我们同时采用八个进程进行工作。

3）网络异常发生器

对于网络 AG，我们不断向远程 TCP 服务器发送 512 个字符，并接收来自服务器的回复以模拟与远程主机交换数据的工作负载。服务器与客户端位于同一局域网中，以支持较大的网络流量，同时采用八个进程进行工作。

为了进行比较，我们实现了皮尔逊相关系数法（PCC），该法已被现有的工作广泛应用于根源分析[29] 。公式(3.6)显示了特征 F 如何被识别为 PCC 中的根源特征，其中 ρ 是当前特征和任务持续时间的 Pearson 系数，λca 和 λq 是超参数。另外，我们还选择了 Mantri 作为比较。

$$|\rho| > \lambda_{ca} \qquad (3.6)$$
$$F > quantile_{\lambda_q}$$

3.4.2 异常注入分析

本文使用 huge 规模的输入（100 万页和 100 个类）的 NaiveBayes 工作负载，并应用不同类型的 AG 来验证根原因检测算法。我们在一个计算节点（Slave Node）周期性地启动 AG 以模拟真实的集群环境（资源利用率波动）。如果一个任务的持续时间与 AG 注入周期重叠，我们认为这个任务受到 AG 的影响。如果一个 AG 的影响导致一个慢任务，BigRoots 应该能够将我们注入的异常识别为这个低效任务的根源特征。图 7，图 8，图 9，图 10 分别显示了 NaiveBayes 分类器在不启动 AG 以及分别启动 CPU、I/O、网络 AG 的时候资源利用率、慢任务和本文提出的算法的检测结果。在图中 X 轴是任务执行的时间线，左侧的 Y 轴表示资源利用率（百分比），右边的 Y 轴表示通过将慢任务的持续时间除以阶段持续时间中位数计算出的慢任务因子。图中水平的黑色部分表示标注了慢任务的时间跨度、慢任务因子及其根原因。图中同时也显示了未确定原因的慢任务作为比较。图像上方的黑色虚线表示注入不同类型异常的开始时间和持续时间。

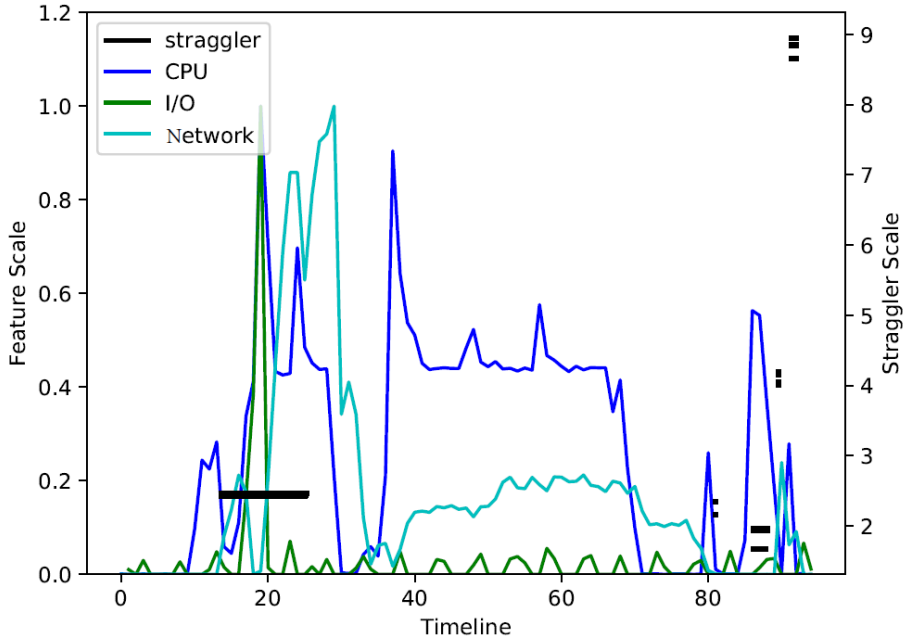


图 7 无异常注入时资源利用率变化和慢任务的分布

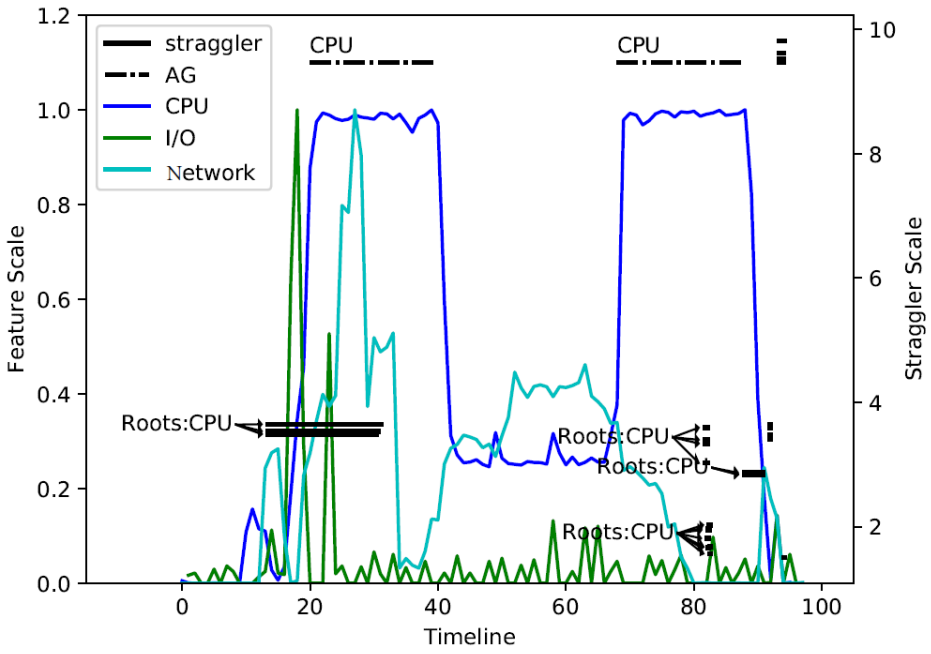


图 8 CPU 异常注入时资源利用率变化和慢任务的分布以及本文的算法检测到的根原因

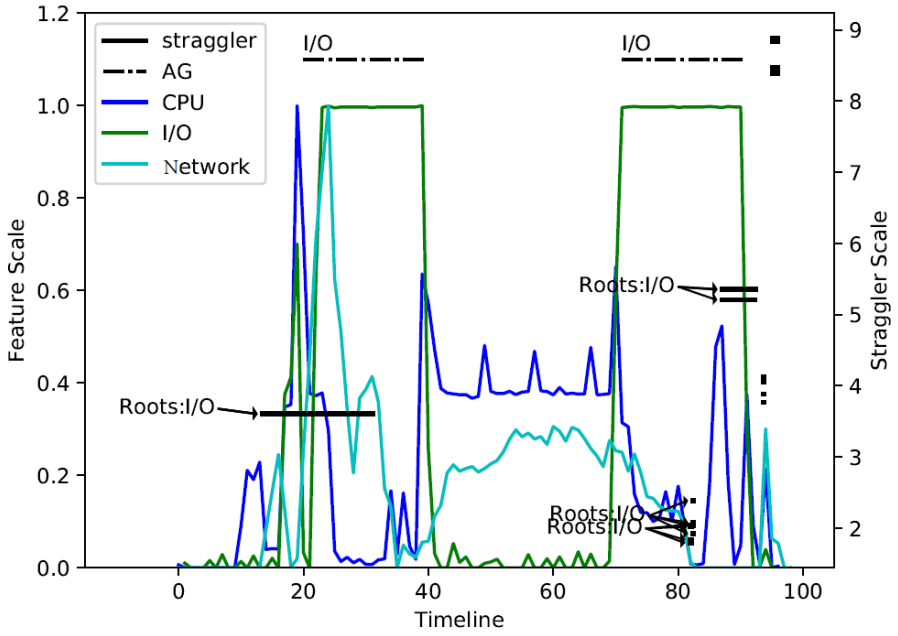


图 9　I/O 异常注入时资源利用率变化和慢任务的分布以及本文的算法检测到的根原因

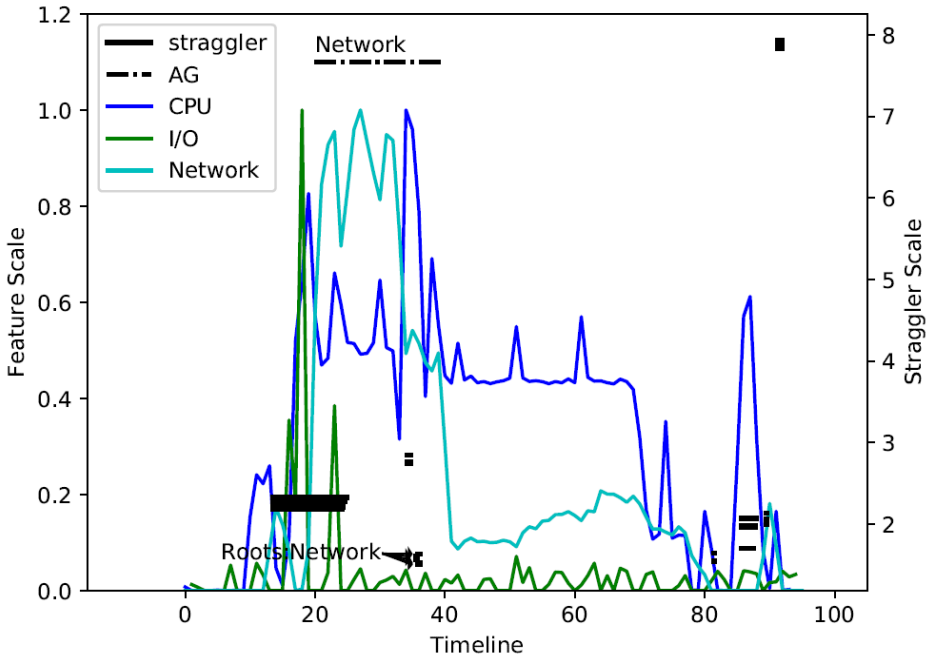


图 10　网络异常注入时资源利用率变化和慢任务的分布以及本文的算法检测到的根原因

从图 8 可以看出，当 CPU 异常注入的时候，本文的算法能够正确地将慢任务归因于高 CPU 利用率。当我们注入 CPU 异常时，在 13s 至 26s 的时间内，慢任务因子从 2.43 增加到 3.55，增加了 46%。较高的 CPU 利用率也会导致更多的图 7 中不存在的慢任务（从 82s 到 85s），图 9 显示了 I/O 异常注入时算法的找到的根原因，低效模式与 CPU 异常注入类似。但是我们可以看到 I/O 资源换竞争比 CPU 资源竞争对慢任务造成了更严

重的影响。与图 7 相比，在 90s 时间段内，慢任务因子增加了 2.6 倍，我们的算法仍然准确地确定了高 I/O 利用率是产生慢任务的根原因。网络异常注入较为不同，如图 10 所示，网络资源竞争对慢任务的影响很小。这是由于在局域网（LAN）中网速较快，甚至超过本地 I/O 的速度，因此传输数据的性能瓶颈不在于网络拥塞。因此，图 10 **错误!未找到引用源。**中只有三个慢任务的低效原因被标注为高网络占用率。

本文的算法和其他算法的比较如表 3 所示。对于 PCC 和 BigRoots，我们通过参数搜索选择最佳参数设置。通过列举给定范围内给定步长的可调参数，我们可以获得最佳参数设置，实现最高 TPR 和最低 FPR。由于正样本的数量远小于负样本的数量，因此我们使用 TPR+1-FPR 作为参数搜索的度量。参数的范围和步骤如表 4 所示。Mantri 和 Xue 使用固定参数设置，没有使用参数搜索。

表 3 我们的算法和其他算法的对比

异常类型		CPU	I/O	网络
本文的算法	TPR(%)	100.00	100.00	38.46
	FPR(%)	1.15	0.00	0.00
PCC	TPR(%)	100.00	56.25	38.46
	FPR(%)	28.65	27.85	30.24
Mantri	TPR(%)	5.00	12.50	0.00
	FPR(%)	1.72	3.22	1.33
Xue	TPR(%)	100.00	100.00	0.00
	FPR(%)	30.37	33.42	29.44

表 4 本文的算法和 PCC 超参数搜索范围

模型	超参数	下界	上界	步长
本文的算法	λ_p	0.0	4.0	0.2
	λ_q	0.1	1.0	0.1
PCC	λ_{ca}	0.0	1.0	0.05
	λ_{cq}	0.0	1.0	0.1

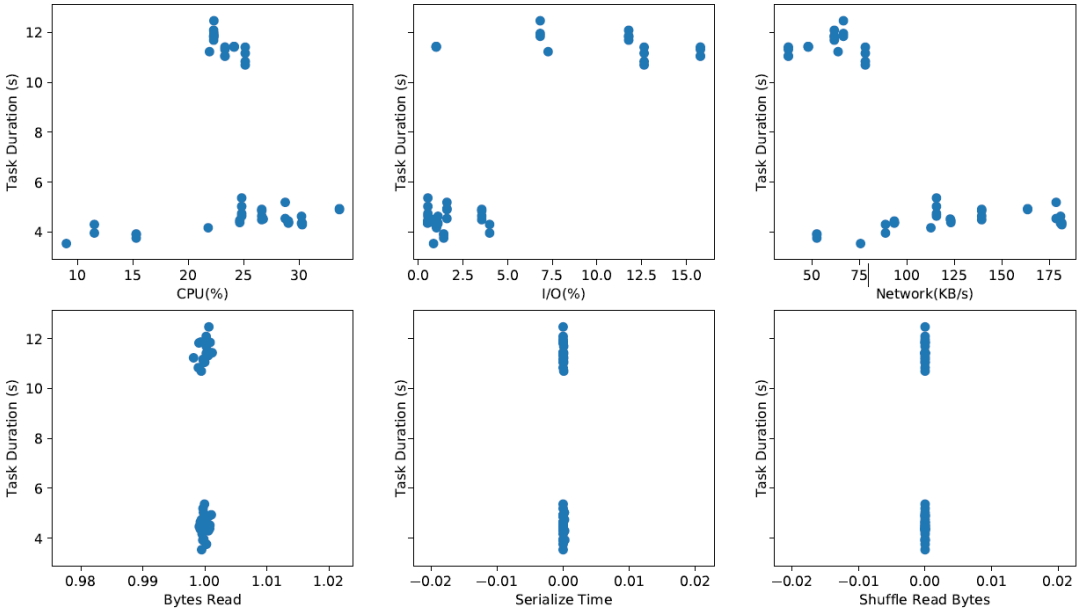


图 11　不同的特征和任务持续时间示意图

本文的算法在所有方法中达到了最高的准确率。PCC 和 Xue 具有较高的 TPR，但也具有较高的 FPR。Mantri 在我们的实验中也表现不佳。在我们的实验中，现有的其他方法无法准确识别根原因的原因是存在许多影响任务持续时间的特征，这些特征相互之间也可能会相互影响，因此，直接对任务持续时间与单个特征建模是不够的。如图 11 所示，任务持续时间和单个特征之间没有简单的相关性，这解释了 PCC 和 Mantri 的不在。通过整合资源特征和其他应用层面的特征，本文的算法克服了 PCC 和 Mantri 的缺点。Xue 方法的一个局限性是只有少数特征被采用，它不能分析网络拥塞的影响，这就解释了当注入网络异常时 TPR 为 0 的原因。

为了定量地计算不同异常注入对应用程序性能的影响，本文还计算了不同异常注入下应用程序持续时间。对于混合异常注入，本文随机启动三种异常之一，我们重复实验 10 次，结果如图 12 所示。可以看到注入 I/O 异常对应用持续时间的影响最大，网络异常注入对工作时间的影响最小。CPU，I / O，网络和混合异常注入下的应用程序持续时间延迟分别为 4.22%，5.86%，3.53%和 4.02%。但是总的来说，这些异常注入对应用程序执行时间影响有限。

3.4.3　敏感性分析

为了理解阈值设置对我们根源分析准确性的影响，我们在不同的阈值配置下画出了不同模型的 ROC 曲线。Mantri 和 Xue 不包括在内，因为它们使用固定的参数设置。

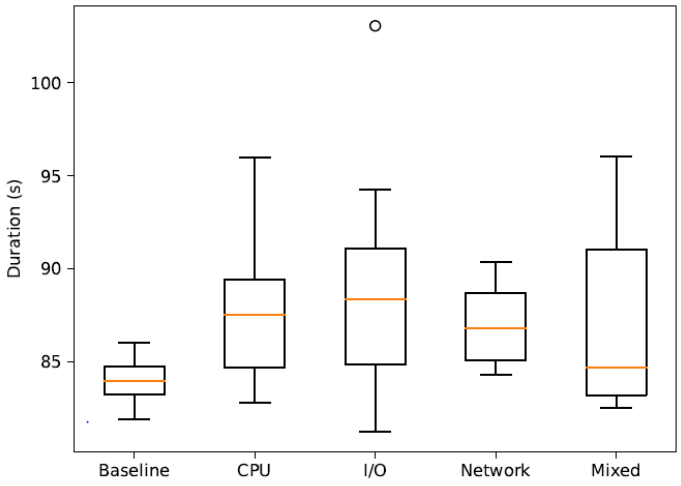


图 12 不同异常注入时对应用持续时间的影响

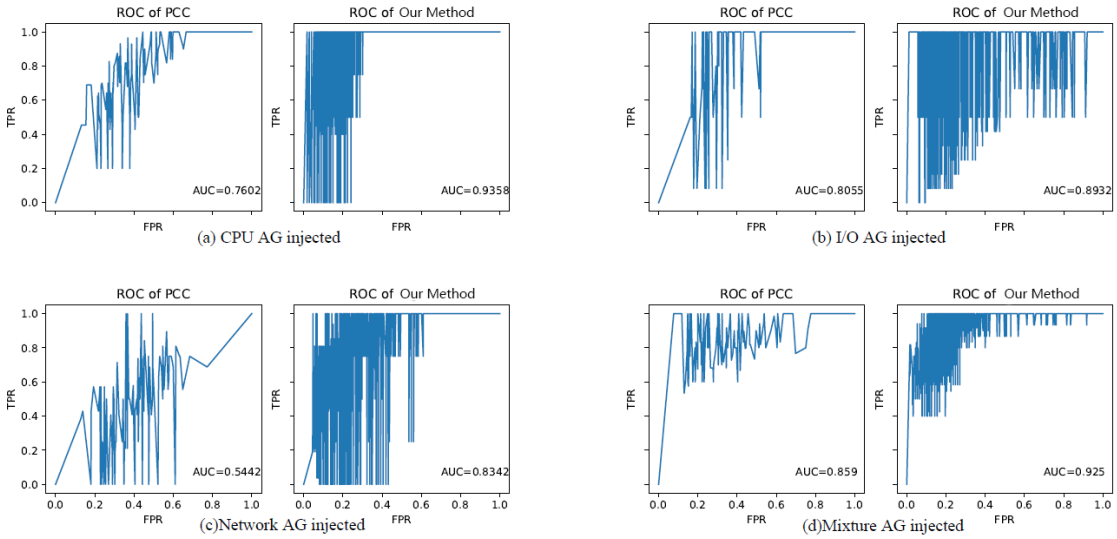


图 13 ROC 分析

本文的算法在根原因分析中使用两个阈值，λ_q 控制同一阶段中当前值与最大值之间的差异，而 λ_p 控制当前值大于特征中值的程度。PCC 也使用两个阈值，λca 控制最小皮尔逊相关系数，λcq 控制同一阶段内特征值接近最大值的程度。我们重复实验 10 次以消除系统噪声的影响。ROC 的波动是由两个阈值的联合影响引起的。考虑图 13 中所有实验的曲线下面积（AUC），本文的算法优于 PCC。

如图 13（a）-（c）所示，当单独注入 CPU、I/O、网络异常时，本文的算法的 AUC 分别比 PCC 大 23.10%，10.90%和 53.29%。本文的算法的 ROC 曲线有更多的左上角点，说明本文的算法可以找到更多的优化配置参数。PCC 的 ROC 曲线略高于对角线，

这意味着 PCC 只比随机猜测略好。在图 13（d）中，当混合注入异常时，本文的算法的 AUC 比 PCC 大 7.6%。混合资源竞争下我们的方法与 PCC 相比准确度只是稍高的原因是不同异常的联合注入会导致更大的相关系数，这对于 PCC 来说更容易检测到。

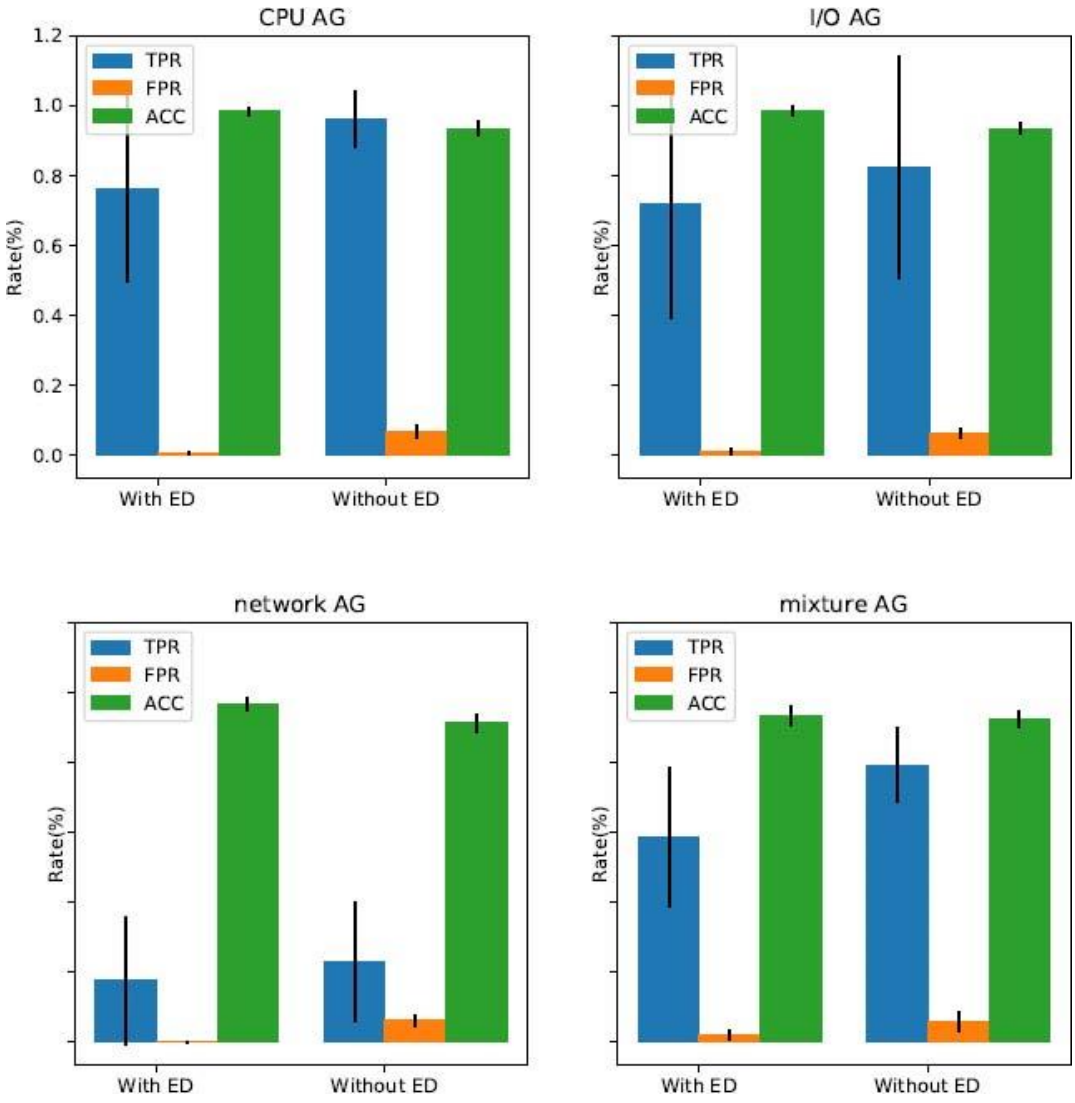


图 14　边缘检测的对比图

以前的文献中没有考虑应用程序本身对资源占用率的影响。我们通过比较带边缘检测和不带边缘检测的算法来证明边缘检测的有效性。结果如图 14 所示，分别注入 CPU，I/O，网络，混合 AG 时，带边缘检测的算法将 FPR 分别降低了 85.71%，78.12%，100.00%，62.23%，ACC 分别增加了 0.88%，4.87%，6.53%，1.24%。通过边缘检测，我们的方法实现了更低的 FPR 和更高的 ACC，边缘检测能够有效地识别资源竞争情况下的根原因。

为了评估集群规模的 BigRoots，我们随机启动不同的 AG，这些 AG 持续跨越不同节点的随机周期。 表 5 显示了不同 AG 注入节点的时间和持续时间.BigRoots 和其他方

法的结果如表 6 所示。我们可以看到 BigRoots 的 FPR（0.35％）远小于 PCC 和薛（16.25％）），这表明 BigRoots 通过将不相关的功能作为根本原因来减少错误。 Mantri 在此实验中无法运行，因为它无法处理跨节点的多个异常。 虽然 BigRoots 的 TPR（召回）（60.56％）小于 PCC（66.19％），但 BigRoots 仍然达到了最高的 ACC。

为了评估跨节点的异常注入的影响，我们随机启动不同类型的异常注入，这些异常跨越不同节点并持续随机的时间长度。 表 5 显示了不同异常注入节点的时间和持续时间，本文的算法和其他方法的结果如表 6 所示。我们可以看到本文的算法的 FPR（0.35％）远小于 PCC 和 Xue（16.25％）。虽然本文的算法的 TPR（60.56％）小于 PCC（66.19％），但 ACC 依然较高。

表 5 跨节点异常注入的影响

节点	开始时间/持续时间（s）	异常类型
Slave1	0/10	CPU
	100/110	I/O
Slave2	30/40	CPU
	63/73	CPU
	83/93	CPU
Slave3	99/109	I/O
Slave4	27/37	网络
	87/97	I/O
	112/122	网络
Slave5	33/43	I/O
	53/63	CPU
	69/79	I/O
	100/110	CPU

表 6 跨节点异常注入时不同算法的对比

算法	TP	TN	FP	FN	FPR(%)	TPR(%)	ACC(%)
本文的算法	43	282	1	28	0.35	60.56	91.81
PCC	47	237	46	24	16.25	66.19	80.22
Mantri	0	283	0	71	0.00	0.00	79.94
Xue	55	179	104	16	77.46	36.75	66.10

3.5 闭环优化分析

表 7 针对 Hibench 不同工作负载分析根原因得到的结果

机器学习	Kmeans	Shuffle_write_bytes (2), CPU (1), I/O (14), Network (1)	40
	Naive Bayes	Shuffle_read_bytes (3), CPU (6), I/O (16), Network (4)	78
	Logistic Regression	Bytes_read(158), CPU (67), I/O (98), Network (26)	1033
	PCA	CPU (84), I/O (240), Network (138)	2592
	SVM	CPU (74), I/O (138), Network (91)	1501
Micro	Sort	-	8
	Terasort	CPU (9)	22
	Wordcount	-	12
图计算	NWeight	Bytes_read (12), CPU (2), I/O (6), Network (2)	64
SQL	Aggregation	I/O (3), Network (3)	18
搜索引擎	Pagerank	CPU (9), I/O (4), Network (8)	24

本节我们分析 Hibench 中的不同工作负载，以找出低效的根源。结果如表 7 所示，资源竞争是工作负载中最常见的根原因特征，这一结论与以前的文献一致。I/O 竞争比 CPU 和网络竞争更有可能成为低效的根原因，因此 I/O 有很大的改进空间。此外，包括 Kmeans、NaiveBayes，Logistic Regression 和 NWeight 在内的机器学习负载和图计算工作负载更容易产生数据倾斜。机器学习工作负载通常会迭代多次以进行收敛，因此数据倾斜会严重影响性能通常，内因特征（如数据倾斜）可以用程序优化来改进，而外因特征（如资源竞争）需要更好的资源管理来优化性能，一般是跟具体的集群相关，优化不具备通用性，所以本节的优化主要考虑内因。

```
41  // Load training data in LIBSVM format.
42  val data: RDD[LabeledPoint] = sc.objectFile(
        inputPath)
43  // Split data into training (60\%) and test
        (40\%).
44  val splits = data.randomSplit(Array(0.6, 0.4),
        seed = 11L)
45  val training = splits(0).cache()
```

图 15 Logistic 回归中数据倾斜瓶颈对应的源代码

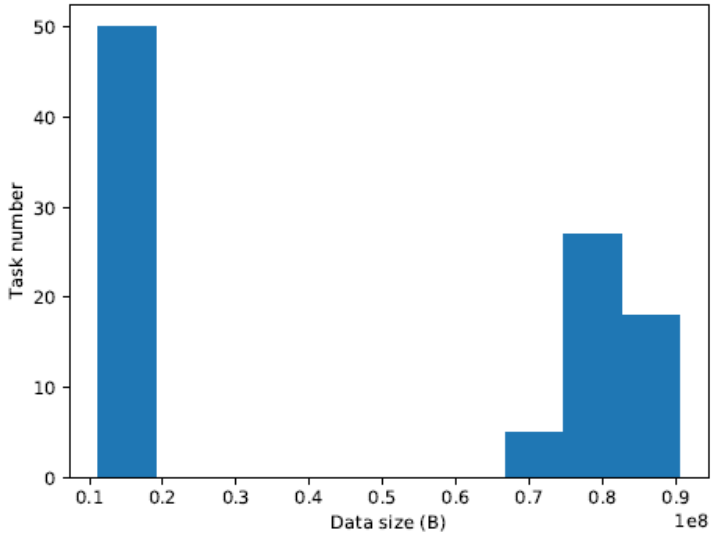


图 16 优化前 Logistic 回归的数据分布

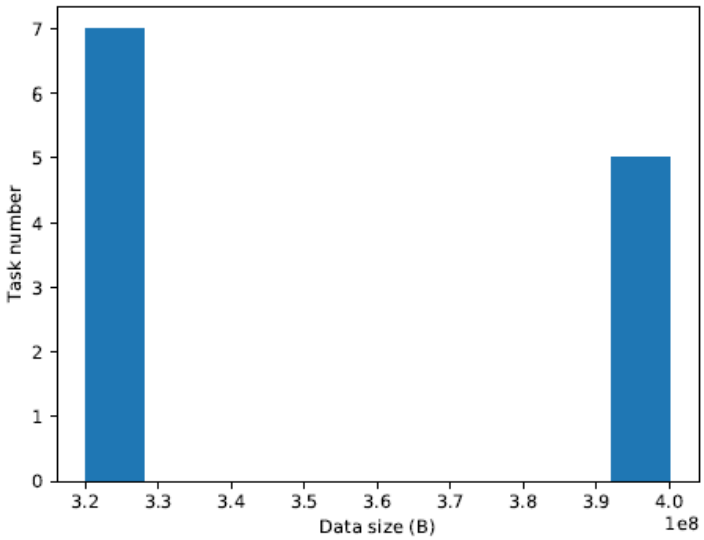


图 17 优化后 Logistic 回归的数据分布

在 Logistic 回归工作负载中，本文的算法识别出由 Bytes_read 引起的 158 个慢任务。我们发现这些慢任务数据倾斜的原因可归于源代码，如图 15 所示。从 HDFS 加载的训练数据在训练过程中的 72 个阶段中被重用。如果数据倾斜，应用程序迭代的性能会受到严重影响。从 HDFS 读取的训练数据的分布，如图 16 所示。我们可以看到数据分布是高度倾斜的，我们的优化策略是在训练前重新分配数据，通过这种优化，数据分布的标准偏差从 0.6985 降至 0.1116，应用持续时间从 601.31s 降至 376.75s。图 17 显示了优化后的分割分布。

```
74  val edges = sc.textFile(input, numPartitions).
        flatMap {
75      ...
76  }.partitionBy(part).map(_._2)
```

图 18 Nweight 图计算中数据倾斜瓶颈对应的源代码

Nweight 工作负载遇到与 Logistic 回归相同的问题，但是导致数据倾斜的 Bytes_read 只存在于 NWeight 中的第一个计算阶段。低效行为对应的源代码如图 18 所示。与 Logistic 回归相比，倾斜数据不会被重用。因此性能下降不那么严重。另一个区别是 NWeight 的数据倾斜是由 HDFS 调度策略引起的，HDFS 调度策略将所有块存储在单个节点中。我们的优化是将块重新分配给不同的节点。通过这种优化，数据分布的标准偏差从 0.6025 降至 0.4007，应用持续时间从 153.38 秒降至 139.78 秒。

在应用上述优化之后，我们重新对这两个工作负载进行根源分析，结果如表 8 所示。经过优化后，数据倾斜不再是两种工作负载的根原因。对于 Logistic 回归，资源竞争引起的根原因也减少了，CPU 竞争完全被消除。

表 8 优化前后根原因分析对比

工作负载	优化前	优化后
Logistic 回归	Bytes_read(158),CPU(67), I/O(98), Network(26)	I/O (9), Network (2)
Nweight	Bytes_read (12), CPU (14), I/O (3)	CPU(16), I/O(3), Network(6)

3.6 本章小结

本章详细介绍了 Spark 应用层面的性能问题，可能造成性能瓶颈的特征，低效行为的分析方法，验证了低效行为分析方法的正确性，最后对分析得到的低效行为进行了相应的优化。相比于以前的研究，本文提出的算法具备广泛的适应性，针对不同的应用和不同的特征都适用；更加细粒度，本文提出的算法可以分析得到任务粒度的低效行为根原因分析，而其他的离线根原因分析算法仅仅能得到所有低效任务和特征的相关性；包含了更加全面的特征，包含了资源占用特征和 Spark 日志中抽取的特征；在 TPR、FPR、ACC 等多种指标下本文提出的算法效果都更好。

第四章 框架层面低效算子识别与根原因分析

本章讨论如何以 Spark 为例在 Spark 框架内部进行性能分析，Spark 内部将任务划分成多个算子（Operation）执行，算子是 Spark 框架内部执行计算的最小粒度，和具体的应用分离，所以本文称之为框架层次，第三章的应用层面的性能分析无法获得算子的低效行为，Spark 本身也没有提供算子的性能日志信息，所以本文通过插桩获取这些性能信息。框架层次的性能分析的必要性提现在以下几个方面，首先，通过识别低效算子，我们可以改进 Spark 框架的具体实现，或者在编程的时候尽量避免使用这些算子；其次，算子和应用程序源代码有着一一对应关系，低效算子也意味着低效代码，可以帮助应用开发人员改进应用程序源代码；最后，可以得到常见的算子低效模式，帮助应用开发人员和框架开发人员优化应用程序设计。

4.1 框架层次低效行为描述

在高层，每个 Spark 应用程序都提交到 Driver（主节点），然后 Driver 将计算并行地分发到集群中。 为了有效地并行执行，Spark 将数据抽象成弹性分布式数据集（RDD），它是跨集群节点不同 partition 的元素集合，执行并行计算。 RDD 可以通过读取 HDFS 中的文件得到，也可以由内存中的数据转换得到，RDD 既可以保留在内存中以供后续计算使用，也可以保留在磁盘中防止内存溢出，这就是 Spark 内存计算的基本思想。Spark 不仅仅计算具有鲁棒性（失败的任务会自动在别的节点重启），数据也有鲁棒性（RDD 被损坏时可以自动计算恢复）。

RDD 支持两种类型的操作：Transformations（将现有的 RDD 转变成另外一种 RDD）和 Actions（将 RDD 结果返回给 Driver）。例如，map 是一个转换，它通过一个函数传递每个数据集元素，并返回一个表示结果的新 RDD。而 reduce 是一个使用某个函数聚合 RDD 的所有元素的操作，并将最终结果返回给驱动程序。Spark 中的所有 Transformations 都是惰性的，只有当出现 Actions 操作的时候才会执行计算。RDD 之间会存在相互依赖关系，组成复杂的有向无环图，而 Spark 执行计算的时候只会考虑 Actions 算子所依赖的所有算子，不会做额外的计算。

所有大数据框架都会面临 shuffle 的问题，即不同节点之间交换数据，如果没有 shuffle，大数据框架将会大大简化、大数据应用程序性能也会有极大的提高。一般来说，

shuffle操作非常耗时，以reduceByKey算子为例，reduceByKey算子生成一个新的RDD，其中单个键的所有值都组合成一个元组 - 键和对与该键关联的所有值执行 reduce 函数的结果。但是单个 Key 包含的数据有可能在别的 RDD 分区上甚至别的节点上，所以必须从别的节点将属于同一个 Key 的数据拷贝到同一个节点。Shuffle 涉及到磁盘 I/O，数据序列化和网络 I/O，Spark 通常自动生成一些 map 和 reduce 操作，在底层，map 操作将根据不同的分区将数据排好序写入一个文件，然后 reduce 再把相关的文件读入。shuffle 会消耗大量的堆内存，在内存数据溢出的时候，Spark 将额外的数据写入磁盘，会造成更大的 I/O 开销和频繁的垃圾搜集。

所以在 Spark 的图依赖关系中，如果一个 RDD 的 partition 依赖于父 RDD 的多个 partition，Spark 则需要 shuffle 数据需要等待父 RDD 所有 partition 计算完成，Spark 根据这种计算将应用程序划分成多个阶段。但是如果只依赖父 RDD 的一个 partition，就无须等待别的 partition 计算完成，而本章主要分析这种情况下算子的低效行为。一个简单的阶段划分如图 19 所示。

这种算子粒度的低效行为比低效任务的检测更为复杂，对于任务来说，由于任务之间要执行同步，所以执行时间较长的慢任务就是性能瓶颈，但是低效算子却没那么简单。低效算子不仅要考虑算子相对于其他算子执行时间的长短，还要考虑算子自身的执行时间，算子所在的任务的执行时间。同时，对低效算子对由于执行时间较短，而且同一个任务的不同的算子不存在数据量大小不一致的问题，因此需要从更细粒度的特征中挖掘导致低效行为的原因，而不是系统资源、数据清洗等。

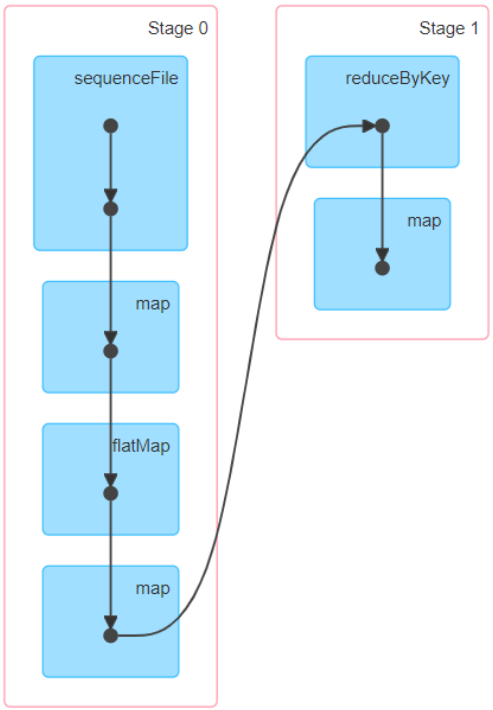


图 19 WordCount 应用程序对应的算子操作

4.2 插桩方法

由于 Spark 本身不提供算子粒度的信息，所以必须通过插桩获取算子的性能数据。考虑到 Spark 算子粒度的插桩不需要获取应用程序上下文的信息，所以本文选取了 Byteman 作为插桩工具。Byteman 作为一个支持动态插桩的工具，有着部署方便、配置简单的优点。由于本文需要在获取算子性能信息的同时对 JVM 性能数据进行采样，所以本文选择了重写 Byteman 的插桩辅助函数。

Spark 所有的 RDD 函数都继承自 org.apache.spark.rdd.RDD 类，而 Byteman 支持同时插桩某个基类和该基类的子类，只需要在类名前面加上符号^。所以本文的主要插桩脚本如附录 1 所示。

本文对所有算子在开始执行和执行结束时分别进行插桩，调用重写的插桩辅助函数 MainHelper，同时记录下实际类名、开始结束的标记、时间戳、阶段 ID、线程号、partition ID 等信息。便于在日志解析的时候能够准确还原每个算子的运行轨迹。对于插桩辅助函数，我们采用了 Java 的管理接口 ManagementFactory，在算子记录插桩信息的时候采样 JVM 性能信息，最后将插桩时发送的消息和 JVM 性能信息一起写入日志文件。本文采样的 JVM 特征如表 9 所示。

表 9　本文采集的 JVM 特征

名称	描述
LoadedClassCount (LSC)	JVM 加载的类数量
ObjectPendingFinalizationCount(OPFC)	等待结束的对象的数量
HeapUsed(HU)	分配的堆内存的大小
HeapCommited(HC)	申请的堆内存的大小
HeapOccupationRate(HOR)	分配的堆内存的大小/申请的堆内存的大小
NonHeapUsed(NHU)	非堆内存的使用量
NonHeapCommited(NHC)	分配的非堆内存大小
NonHeapOccupationRate(NHOR)	非堆内存的使用量/分配的非堆内存大小
ThreadNum(TN)	线程数量
DeadLockedNum(DLN)	死锁内存的数量
CurrentThreadCPUTime(CTCT)	当前线程占用的 CPU 时间
CurrentThreadUserTime(CTUT)	当前线程的用户 CPU 时间
UserCPURate(UCR)	用户 CPU 时间/线程 CPU 时间
DaemonThreadCount(DTC)	守护线程的数量
CompilationTime(CT)	编译占用的时间
GCCount(GCC)	累积的垃圾搜集次数
GCTime(GCT)	累积的垃圾搜集时间
BufferPoolCount(BPC)	缓冲池的数量
BufferMemory(BM)	缓冲池占用的内存大小

在进行日志解析的时候，采用以 PartitionId+RDD+线程 ID+阶段 ID 为唯一 key 记录每个算子的性能信息，由于算子开始执行的性能日志和结束执行的性能日志是分离的，所以本文需要匹配同一个算子开始和结束日志。同时，本文还面临另外一个问题，算子是嵌套调用的，所有原始日志的格式如附录 2 所示。

所以在进行日志解析的时候需要排除嵌套子算子的影响，对于持续时间和 JVM 垃圾搜集这些累积变量，本文将算子的特征减去父算子的相应特征得到真实的特征。所以在解析的时候不仅要记录一个字典数据结构，还要记录同一个线程执行的堆栈，便于还原父子关系。

4.3 性能分析方法

4.3.1　低效算子评分

为了定位低效算子，本文为每个算子计算一个低效算子评分，然后将低效评分较高

的算子作为低效算子，由于本文提出的低效算子评分不具备归一性，所以无法设定一个有效的阈值判定，只能取每个阶段中评分较高的算子作为低效算子。在具体探究这个问题之前，本文首先澄清几个概念。

- 算子：指 Spark 划分的一个计算单位，是 DAG 调度的一个节点，算子是一个静态概念
- 算子实例：在任务中执行的算子，是一个动态的概念
- 同阶算子：属于同一个算子的算子实例
- 算子流水线：同一个任务包含不同的算子实例组成一个没有等待延迟的算子流水线，算子流水线的执行时间等于任务的执行时间
- 关键流水线：执行时间最长的算子流水线，决定了真个阶段的执行时间

本文主要考虑一下几个评分原则：

- 算子实例的执行时间差异很大，执行时间较长的算子实例应该分配较高的分数；
- 在关键流水线优化后，其他算子流水线有可能会成为新的关键流水线，因此，我们应该为执行时间较长的算子流水线中的算子实例分配较高的分数；
- 算子实例的优化潜力也应该被考虑，一般来说，相比于同阶算子执行时间更长的算子实例具有更多优化潜力，应该被分配较高的分数。

为了满足第一个原则，我们使 IS 与算子实例的执行时间成正比。为了满足第二个和第三个原则，我们提出流水线因子（PF）为每个算子流水线分配一个合理的评分，如图 20 所示。根据 N 个算子流水线的持续时间，将关键流水线的持续时间分为 N 部分。最左边的片段意味着其他流水线执行时间都大于它，因此我们分配给它最小权重。从左到右其他片段的权重逐渐增加。PF 可以用公式 4.1 计算，其中 i 代表第 i 条算子流水线，j 代表第 j 个片段，n_i代表第 i 条算子流水线内的片段数量。整个算子流水线相比于其他算子流水线执行时间越长，PF 值越大。

$$\mathrm{PF_i} = \sum_{j}^{n_i} \frac{S_j}{N-j} \quad (4.1)$$

为了满足第三个原则，我们使用 Straggler Scale（SS）作为 IS 的一个因子。SS 是当前算子实例的执行时间除以同阶算子实例执行时间的中位数，这个因子代表了算子的优化潜力。把上述因素放在一起后，我们使用公式 4.2 推导出 IS。

$$IS_{ij} = PF_i \times span_{ij} \times SS_{ij} \quad (4.2)$$
$$= \frac{\sum_j^{n_i} S_j * span_{ij}^2}{(N-j) * Median_j}$$

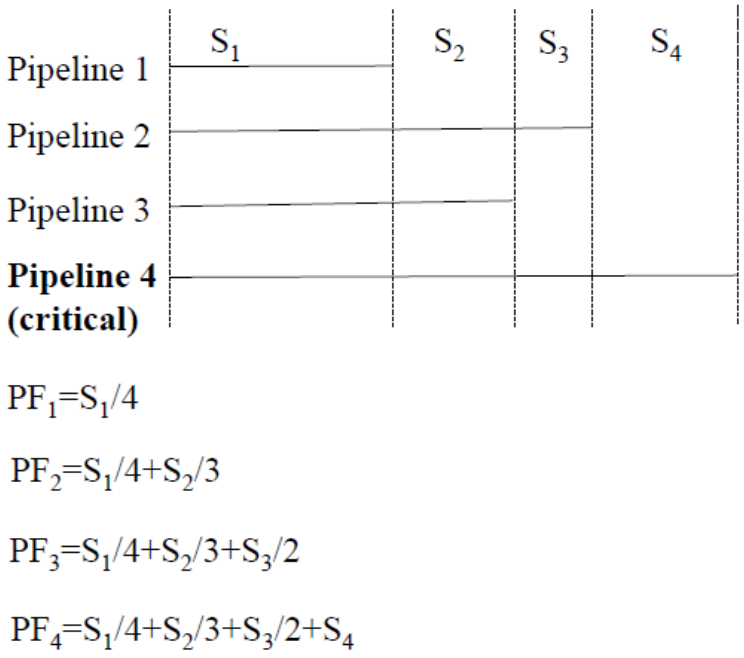


图 20　低效算子评分示意图

4.3.2　基于聚类的算子低效行为分析

通过采样每个算子执行期间 JVM 的底层性能信息，我们可以计算这些特征和低效算子的相关性。第三章的根原因分析方法不适用于算子粒度的性能分析，首先，影响算子的特征和第三章提出的特征有着根本的不同，第三章提出的任务序列化与反序列对算子来说不存在这样的操作，对于数据倾斜，同一个任务的不同算子实例处理的数据量是一样的，不能用来解释算子的低效行为，对于资源占用特征，同一个任务的不同算子大致处于同样的环境下，所以资源占用也不能用于解释低效算子；其次，不同特征相互比较的方法也不适用于低效算子的根原因分析，一个重要的原因是低效算子不仅要考虑到同阶算子，更要考虑到同一个算子流水线里的其他算子。所以本文先用 IS 定位低效算子，再采用聚类的方法对低效算子的 JVM 特征进行聚类，通过分析聚类中心的特征大小，就可以判定哪些特征和低效算子相关性较强。

本文的聚类算法采用 Kmeans，给定多个特征向量 X_i，我们首先初始化几个聚类中心μ，然后计算属于同一个聚类中心的特征向量，再用公式 4.3 重新计算聚类中心。其中 S 为所有特征向量的集合，k 为聚类中心的数量，C_i为属于第 i 个聚类中心的特征向量的

集合。

$$\arg\min_{S} \sum_{i=0}^{k} \sum_{x \in C_i} ||x - \mu_i||^2 \quad (4.3)$$

4.4 不同负载下的低效算子

我们的实验是在五台服务器组成的集群上进行的，一台服务器作为主节点，另外四台作为计算节点。表 10 列出了每台服务器的配置。我们在实验中使用 Hibench 测试集，工作负载包括机器学习算法（NaiveBayes 分类，K-Means 聚类等），微基准（Sort，WordCount，TeraSort），网页搜索（Nutch Indexing，PageRank）。我们选择 Hibench 是因为它具有一定的代表性和多样性。微型基准测试集代表了很多类型的大数据作业。排序基准代表输出与输入大小相似的程序，而 WordCount 代表大输入小输出的程序。网页搜索测试集代表搜索引擎集群中使用的程序。机器学习测试集代表在 Spark 上迭代执行数据分析作业的场景。

表 10 集群中每个节点的配置

处理器	2*Intel Xeon E5-5620	
	核数	16
	线程数	2
	L1 Cache	32KB
	L2 Cache	256KB
	L3 Cache	12MB
内存	16GB DDR3	
操作系统	CentOS 6.5	
网络带宽	1Gbps	
Spark 版本	V2.2.0	
HDFS 版本	V3.0.0	
JVM 版本	V1.8.0_111	

在本节中，我们将定性和定量分析不同工作负载的低效算子分布情况。由于工作负载的算子出现的次数非常大，因此我们只是采取一些有代表性的阶段进行分析并总结典型的低效算子模式。表 11 列出了这些阶段的基本统计指标。

•Wordcount：Wordcount 有两个阶段。第一阶段包括五个算子：一个 HadoopRDD 算子和四个 MapPartitionRDD 算子。HadoopRDD 负责从 HDFS 读取数据，而

MapPartitionRDD 将 map 函数应用于每个分区。如图 21 所示，HadoopRDD 算子造成的延迟是其他 MapPartitionRDD 算子的 100 倍，这种模式称为模式 A，其中一个算子的执行时间几乎等同于整个任务的执行时间，对这种模式进行算子的低效行为分析不会获得比任务低效行为分析更多的信息。如图 22 所示 Wordcount 的第二个阶段包含一个 ShuffledRDD 算子和一个 MapPartitionRDD 算子。ShuffledRDD 算子从不同的计算节点中获取数据，执行时间的波动较大。但此阶段的 MapPartitionRDD 算子的执行时间与数据大小不成正比，这意味着时间波动不是由数据倾斜造成的。该算子在源代码中对应于将分区数据转换为字符串并存储到 HDFS 的源文件。由此我们得到结论，网络和磁盘 I/O 不稳定性是造成这种现象的原因，这种模式被称为模式 B，其特征在于虽然某一个算子执行时间占了算子流水线的大部分时间，但是其他算子的异常波动也对算子流水线的执行时间有所影响。

表 11 不同低效模式的统计指标。同一个阶段包含的多个算子的统计指标按照先后顺序排列，对于算子分布，我们给出了低效算子的类别和算子实例出现的数量

模式	标准差	均值	IS 最高的 10%的算子分布
A	111.0, 0.5, 0.0, 0.5, 0.51	1498.0, 7.5, 2.0, 2.5, 7.5	HadoopRDD:1
B	102.41, 15.15	79.76, 9.74	ShuffledRDD:9, MapPartitionsRDD:1
C	703.67, 6.64, 9.18, 12.05, 3251.3	1535.54, 15.96, 6.88, 13.75, 6815.88	MapPartitionsRDD:12
D	7.57, 4.77	18.0, 6.33	MapPartitionsRDD:1, NewHadoopRDD:3
E	1427.06, 106.25	3750.64, 22.56	MapPartitionsRDD:1, ShuffledRDD:9
F	27.75, 22.58, 192.45, 340.42, 649.64	8.14, 10.5, 380.54, 864.5, 112.45	MapPartitionsRDD:25, ShuffledRDD:5, ZippedPartitionsRDD2:45
G	1.53, 6.1, 9.46, 0.96	1.54, 3.22, 8.18, 1.8	MapPartitionsRDD:17, PartitionwiseSampledRDD:3

•Terasort：TeraSort 与 Wordcount 工作负载不同。第一阶段包括一个 NewHadoopRDD 算子，三个 MapPartitionRDD 算子和一个 PartitionPruningRDD 算子，如图 23 所示。NewHadoopRDD 算子执行时间非常平衡，并且算子流水线的波动主要是由最后的 MapPartitionRDD 算子引起的。鉴于数据均匀分布在不同的任务中，因此波动的原因是

用户定义的函数，我们称这种模式为模式 C。第二阶段包含一个 NewHadoopRDD 算子和一个 MapPartitionRDD 算子，如图 24 所示。这些算子对任务的持续时间具有相同的影响（类似共振），这种模式称为模式 D。第三阶段包含一个 ShuffledRDD 算子和一个 MapPartitionRDD 算子，如图 25。与模式 A 类似，从其他节点读取数据的算子占主导地位，但是在同阶算子可以被忽略的情况下，个别 MapPartitionRDD 算子实例导致了整个流水线低效，我们称这种模式模式为模式 E。

•NWeight：NWeight 工作负载的第五阶段显示出了另一种模式。它由五个算子组成，包括两个 ShuffledRDD 算子，两个 MapPartitionsRDD 算子和一个 ZippedPartitionsRDD 算子。ZippedPartitionsRDD 算子的个别算子实例导致了整个算子流水线的低效，尽管其他算子执行时间也存在波动，我们将这种模式命名为 F。应用程序开发人员应该考虑如何对数据进行重新分区，以使那些需要大量计算的数据更均匀地分布。

•SVM：SVM 工作负载的第十八阶段显示出与上述不同的模式。该阶段由四个算子组成：三个 MapPartionsRDD 算子和一个 PartitionwiseSampledRDD 算子。这四个算子基本上沿着流水线保持相同的规模，这意味着不同流水线之间的算子执行时间的差异主要是由数据倾斜引起的（我们的集群是同构的，所以执行时间的差异不是由硬件性能引起的），我们把这种模式称为模式 G。

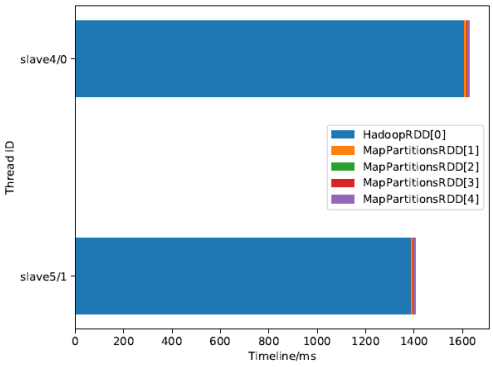


图 21 低效模式 A

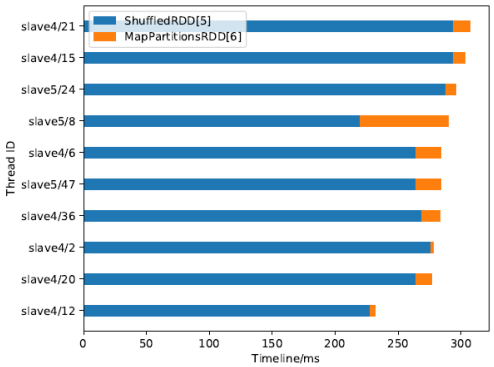


图 22 低效模式 B

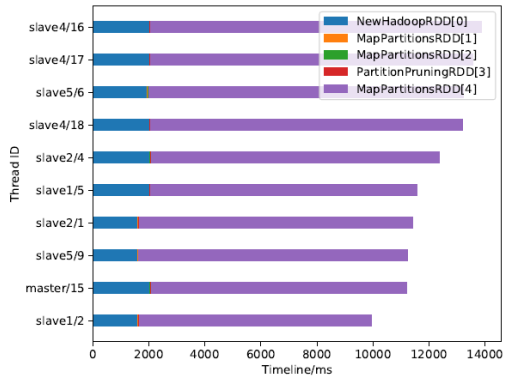


图 23 低效模式 C

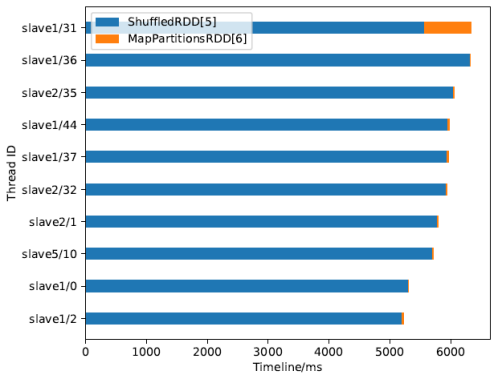


图 25 低效模式 E

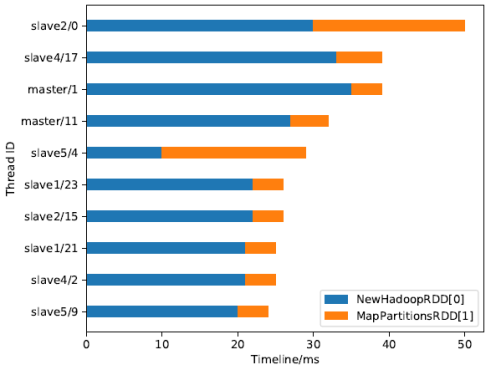


图 24 低效模式 D

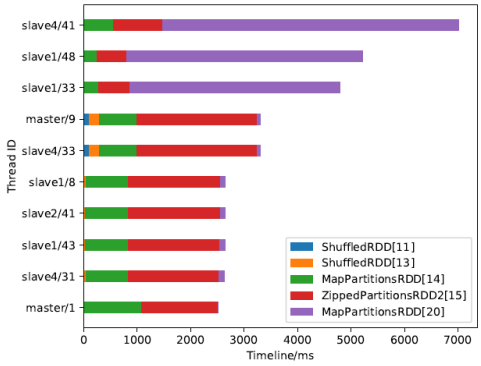


图 26 低效模式 F

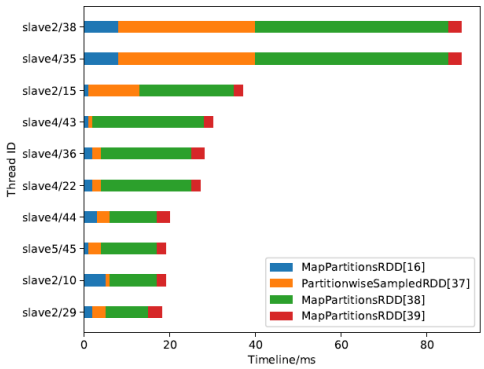


图 27 低效模式 G

表 12 不同模式的低效算子三个聚类中心不同特征的分布

JVM 特征	算子模式					
	B	C	D	E	F	G
BM	0.98	0.82	1	0.7	0.07	0.74
	0.76	0.83	0.83	1	0	0.66
	0.76	1	1	1	0.01	0.94
BPC	0.6	0.83	0.97	0.85	0.61	0.89

	0.5	0.85	0.94	1	0.29	0.85
	0.5	0.97	0.97	1	0.73	0.97
CT	0.59	0.46	0.13	0.88	0.16	0.03
	0	0.94	0.86	0.13	0.33	0.78
	0.85	0.72	0.45	0	0.13	0.54
DTC	0.97	0.94	1	0.92	0.93	0.99
	0.93	0.94	0.96	1	0.77	0.98
	0.93	0.99	1	1	0.92	1
DLN	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	0	0	0	0
GCC	0	0.48	0	0.68	0.55	0
	0	0.88	0	1	0.38	0
	0	0.72	0	0	0.13	0
GCT	0	0.53	0	0.73	0.44	0
	0	0.69	0	1	0.13	0
	0	0.28	0	0	0.07	0
HOR	0.98	0.76	1	0.8	0.18	0.55
	0.57	0.91	0.63	0.71	0.77	0.45
	0.57	0.49	1	0.71	0.14	0.87
LCC	0.98	1	1	0.98	0.92	1
	0.96	1	0.99	1	0.73	1
	0.96	1	1	1	0.98	1
NHOR	0.99	0.99	0.99	0.99	0.7	1
	1	0.99	1	0.93	0.98	1
	1	1	0.99	0.93	0.63	0.98
OPFC	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	0	0	0	0
TN	0.97	0.94	1	0.92	0.93	0.99
	0.93	0.94	0.97	1	0.78	0.98
	0.93	0.99	1	1	0.92	1
UCR	0.91	0.92	0.5	0.99	0.96	0
	0	0.79	0.37	0.98	0.93	0.29
	0.93	0.87	1	0	0.92	0.04

本文对 IS 最大的 10%的算子实例的 JVM 特征进行聚类，为了便于分析，根据经验

选择 3 个聚类中心。结果如表 12 所示。以下是我们根据聚类分析所得出的理解：

•死锁和执行析构函数的对象不是性能问题之一，在所有聚类中心，它们都是零；

•除模式 F 外，缓冲内存是所有模式的重要性能问题；

•守护线程，线程，加载的类和非堆内存使用量代表 JVM 的资源占用量，并在所有聚类中心都有较高的值，充分说明这些特征是影响低效算子的一个重要的因素；

•缓冲池，编译时间，用户时间率和堆使用情况有较大的波动，他们对算子性能的影响较小；

•垃圾收集与算子性能无关，在大多数聚类中心中，GC 时间和 GC 次数为零或非常低。

表 13　不同负载下低效算子的分布

负载	IS 最高的 10%算子的分布
WordCount	ShuffledRDD:9, HadoopRDD:1, MapPartitionsRDD:1
Sort	ShuffledRDD:8, MapPartitionsRDD:7
TeraSort	MapPartitionsRDD:14, ShuffledRDD:9, NewHadoopRDD:3
NWeight	MapPartitionsRDD:266, ZippedPartitionsRDD2:143, ShuffledRDD:75, HadoopRDD:32, VertexRDDImpl:13
PageRank	ShuffledRDD:72, CoGroupedRDD:47, HadoopRDD:8, Map-PartitionsRDD:5
Bayes	MapPartitionsRDD:42, ShuffledRDD:34, HadoopRDD:10
K-Means	ShuffledRDD:30, MapPartitionsRDD:15, ZippedPartitionsRDD2:2, HadoopRDD:1
LR	MapPartitionsRDD:796, HadoopRDD:100, ShuffledRDD:72
PCA	MapPartitionsRDD:4171, PartitionwiseSampledRDD:404,ShuffledRDD:184, HadoopRDD:22
总和	MapPartitionsRDD:10085, HadoopRDD:2075, ShuffledRDD:716, PartitionwiseSampledRDD:614, ZippedPartitionsRDD2:145, CoGroupedRDD:47, VertexRDDImpl:13, UnionRDD:12, SlidingRDD:12, NewHadoopRDD:3

我们同时研究在不同工作负载下具有高 IS 的算子，以便分析哪些算子更容易出现低效行为。结果如表 13 所示。最常见的 3 种低效算子分别是 MapPartitionRDD 算子，HadoopRDD 算子和 ShuffledRDD 算子，这些算子分别代表了 Spark 计算中的三个主要问题：不平衡的用户函数执行，从分布式文件系统读取数据以及 shuffle。

4.5 本章小结

本章介绍了算子粒度的低效行为的表现，进行框架层次性能分析的必要性，解释了

为什么要用动态插桩，以及基于 Byteman 插桩算子的方法。然后，本章又提出了如何定位低效算子以及如何识别对低效算子造成影响的特征。最后，我们以 Hibench 为例，对多种负载进行了低效算子分析，总结了 7 中典型的低效模式，并采用聚类的方法识别了这些算子低效的影响因素，得到了几种最为常见的低效算子。

第五章 分布式文件系统层面的低效函数分析

本章以 HDFS 为例介绍分布式文件系统内部的低效行为，这些性能信息在应用层面和框架层次的性能分析中都无法获取，但是分布式文件系统对所有的大数据系统都是至关重要的，这是大数据应用程序区别于高性能程序的一个重要特点。分布式文件系统的性能和应用层面、框架层次的性能息息相关，应用层面的数据倾斜和框架层次的 HadoopRDD 算子都和分布式文件系统性能紧密相关。本章通过分布式插桩跟踪 HDFS 的 I/O 请求在各个组件的延迟，分析在不同请求下的性能瓶颈。

5.1 分布式文件系统低效行为描述

诊断分布式文件系统的性能问题是一件具有挑战性的事情，分布式文件系统的性能瓶颈可能会来自系统的各个组件，甚至是不同组件之间的交互。HDFS 的几个主要组成部分是名称节点（NameNode），数据节点（DataNode）和客户端节点（Client Node）。HDFS 把一个文件分成若干个数据块，每个数据块的大小一般是 64MB，数据块是 HDFS 的最小存储单位，是 HDFS 设计数据一致性的基础，每个块都要独立的校验码，防止数据块中的数据被破坏，同时，HDFS 默认还会为数据块保留 3 个备份，在数据发生丢失的时候自动恢复数据块。就像单机常见的文件系统一样，HDFS 也有文件索引系统，只不过文件索引信息都在名称节点上，同时，副名称节点（SecondaryNameNode）也会保留一份备份，在名称节点不可用的时候临时承担起名称节点的作用。而客户端节点则是请求 HDFS 文件的节点，一般来说，客户端节点可以是名称节点和数据节点，也可以是其他的远程节点。客户端节点在请求文件之前需要创建 HDFS 客户端对象，然后通过这个对象和名称节点、数据节点交互。HDFS 的性能瓶颈可能存在于数据请求的任何一个环节上。下面将阐述 HDFS 基本读写操作的过程。

如图 28 所示，客户端节点在读文件的时候首先调用 FileSystem 对象的 open 方法，其实获取的是一个 DistributedFileSystem 的对象。DistributedFileSystem 通过 RPC(远程过程调用)获得文件的第一批 block 的 locations，同一 block 按照重复数会返回多个 locations，这些 locations 按照拓扑结构排序，距离客户端近的排在前面。

前两步会返回一个 FSDataInputStream 对象，该对象会被封装成 DFSInputStream 对象，DFSInputStream 可以方便的管理数据节点和名称节点数据流。客户端调用 read 方法，DFSInputStream 就会找出离客户端最近的数据节点并连接数据节点。数据从数据节

点源源不断的流向客户端。如果第一个数据块的数据读完了，就会关闭指向第一个数据块的数据节点连接，接着读取下一个数据块。这些操作对客户端来说是透明的，从客户端的角度来看只是读一个持续不断的流。如果第一批数据块都读完了，DFSInputStream 就会去名称节点拿下一批 blocks 的 location，然后继续读，如果所有的 block 块都读完，这时就会关闭掉所有的流。

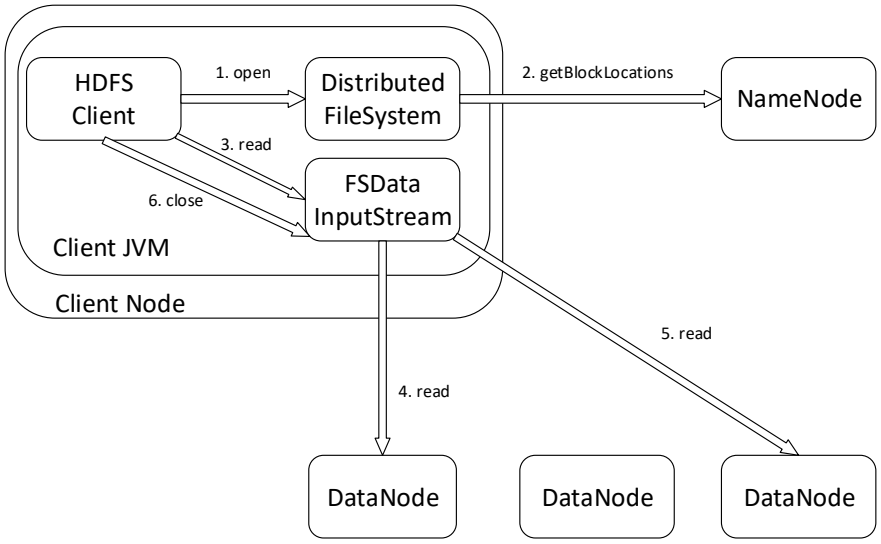


图 28 HDFS 读文件的过程

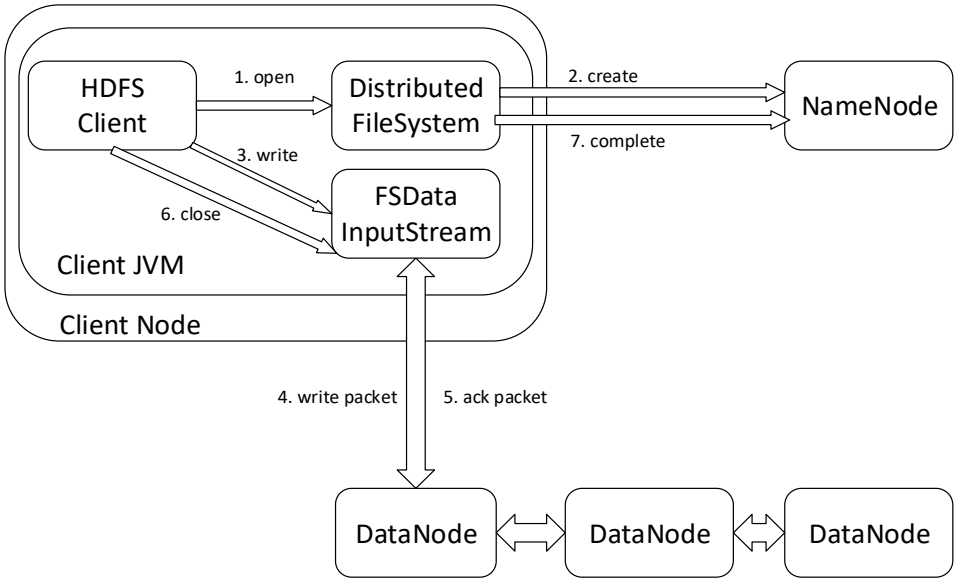


图 29 HDFS 写文件的过程

如图 29 所示，HDFS 的文件写入原理，主要包括以下几个步骤。客户端节点通过调用 DistributedFileSystem 的 create 方法，创建一个新的文件。DistributedFileSystem 通过 RPC（远程过程调用）调用 名称节点，去创建一个没有数据块关联的新文件。创建前，

名称节点会做各种校验，比如文件是否存在，客户端有无权限去创建等。如果校验通过，名称节点就会记录下新文件，否则就会抛出 IO 异常。前两步结束后会返回 FSDataOutputStream 的对象，和读文件的时候相似，FSDataOutputStream 被封装成 DFSOutputStream，DFSOutputStream 可以协调名称节点和数据节点。客户端开始写数据到 DFSOutputStream，DFSOutputStream 会把数据切成一个个小 packet，然后排队通过 TCP 协议发送。DataStreamer 会去处理接受数据队列，它先问询名称节点这个新的数据块最适合存储的在哪几个数据节点里，比如重复数是 3，那么就找到 3 个最适合的数据节点，把它们排成一个流水线。DataStreamer 把 packet 按队列输出到管道的第一个数据节点中，第一个数据节点又把 packet 输出到第二个数据节点中，以此类推。

DFSOutputStream 还有一个队列叫 ack 队列，也是由 packet 组成，等待数据节点的收到响应，当流水线中的所有数据节点都表示已经收到的时候，这时 ack 队列才会把对应的 packet 包移除掉。客户端完成写数据后，调用 close 方法关闭写入流。DataStreamer 把剩余的包都发送到流水线中，然后等待 ack 信息，收到最后一个 ack 后，通知数据节点将文件刷新缓存到磁盘，然后把文件标示为已完成。

HDFS 在这些过程中性能瓶颈可能存在于名称节点查找文件元信息、数据节点发送数据包，数据节点从本地磁盘读取数据，数据节点从接受数据包，客户端节点创建文件系统对象，客户端节点发出请求的延迟等。

5.2 性能分析方法

5.2.1 分布式插桩

由于对 HDFS 进行性能分析必须还原每一个 I/O 请求的调用树，否则便无法分析 HDFS 消耗在各个函数的延迟，定位造成性能瓶颈的高延迟函数，由于第四章中基于 Byteman 的插桩无法获取分布式的调用序列，所以对分布式文件系统的插桩不能采用动态插桩，只能采用静态插桩的方式（直接修改源代码并进行编译）。通过对多种分布式插桩框架进行调研（第二章），本文选择 HTrace 进行插桩，HTrace 已经被集成进 Hadoop 的官方实现，但是官方的插桩仅仅局限于客户端节点和数据节点的少量的函数，日志记录的信息也十分有限，为了和 Hadoop 的兼容性，本文选择采用 HTrace 对 HDFS 插桩进行扩展。

本文的扩展主要从以下几个角度出发：

- 在对可以获取函数处理数据量的函数进行插桩的时候在日志里同时记录这些数据量的信息，通过这些信息我们可以计算函数的数据处理速率，然后可以判定数据处理速率是否是导致 I/O 延迟的主要原因
- 获取名称节点处理文件元信息的延迟，官方的 Hadoop 没有对名称节点进行插桩，无法获取文件元信息的处理性能，但是在生产环境下文件元信息的处理至关重要，在 PB 乃至 EB 即的大数据集群中，会产生非常多的数据块，对这些数据块进行检索、增删都会面临很大的挑战，所以对名称节点的性能分析也至关重要
- 获取数据节点本地 I/O 和网络 I/O 的延迟，官方的 Hadoop 没有考虑到数据节点在发送数据中从本地 I/O 读取数据到缓存和从缓冲通过 socket 发送到其他节点的延迟，但是解耦本地 I/O 和网络 I/O 的性能数据是十分重要的，可以帮助用户发现 HDFS 集群的优化主要在磁盘还是局域网
- 获取数据节点之间的数据交互的延迟、获取名称节点和数据节点之间数据交互的延迟。尤其是在写文件的时候，这些数据交互变得更加重要

本文主要的插桩的函数及其含义如表 14 所示。

表 14 本文插桩的主要函数以及获取的主要参数

类	节点	函数	参数	解释
DFSInputStream	客户端节点	构造函数	src	构造该类的对象
		openInfo	filename	获取一个文件的读写信息
		fetchBlockAt	-	从名称节点获取一个数据块的信息
		readWithStrategy	buffer_size	将数据块读取到本地缓存
		actualGetFromOneDataNode	length	从一个文件中读取一部分数据
BlockReaderLocalLegacy/ BlockReaderRemote2/	客户端节点/	构造函数	-	构造对象
		readFully	length	读取一定大

BlockReaderLocal/ BlockReaderRemote				小的数据到缓存
		readAll	length	读取一定大小的数据到缓存
BlockSender	数据节点	sendPackage (transferTo)	-	采用 transfer 的方法传输数据包
		BlockSender#sendPacket (readLocal)	length	从本地存储中读取一定长度的数据
		BlockSender\# sendPacket (writeToSocket)	length	将本地缓存写入 socket
		doSendBlock	-	发送数据块的封装
BlockReceiver	数据节点	flushOrSync	-	将数据块元信息和内存数据写入磁盘
		receivePacket	-	接收 packet
		receiveBlock	-	接收数据块
FSNameSystem	名称节点	getBlockLocations	-	获取持有某个数据块的节点列表
		renameTo	path	重命名一个文件
		delete	path	删除一个文件
		getFileInfo	path	获取一个文件的元信息
DFSClient/ DistributedFileSystem	客户端节点	getBlockLocations	src	获取数据块位置信息
		rename	src	重命名
		delete	src	删除文件
		getFileBlockLocations	src	获取一个文件所有数据块的信息
		create	src	创建一个新

				文件

HTrace 支持多种日志记录方式，包括 HTracedSpanReceiver，LocalFileSpanReceiver，StandardOutSpanReceiver，ZipkinSpanReceiver 等，也可以自定义 SpanReceiver，本文为了便于后续日志分析，采用了 LocalFileSpanReceiver。HTrace 的插桩可以动态配置，只需要在 core-site.xml 中配置 hadoop.htrace.span.receiver.classes 为 LocalFileSpanReceiver，在 hadoop.htrace.local.file.span.receiver.path 中配置日志文件的路径，在 hadoop.htrace.sampler.classes 中配置采样器（HTrace 支持对调用树进行采样，按照一定的概率保留一棵完整的调用树），hadoop.htrace.sampler.fraction 中配置采样的概率。

除了在配置文件中配置插桩，HTrace 也支持命令行动态配置，使用 hadoop trace –remove 命令可以动态地移除插桩日志接收器（当日志接收器为空的时候，相当于关闭了 HTrace 的采样），hadoop trace –add 可以动态增加日志接收器，hadoop trace –list 可以列出当前配置的日志接收器。

5.2.2 调用树压缩

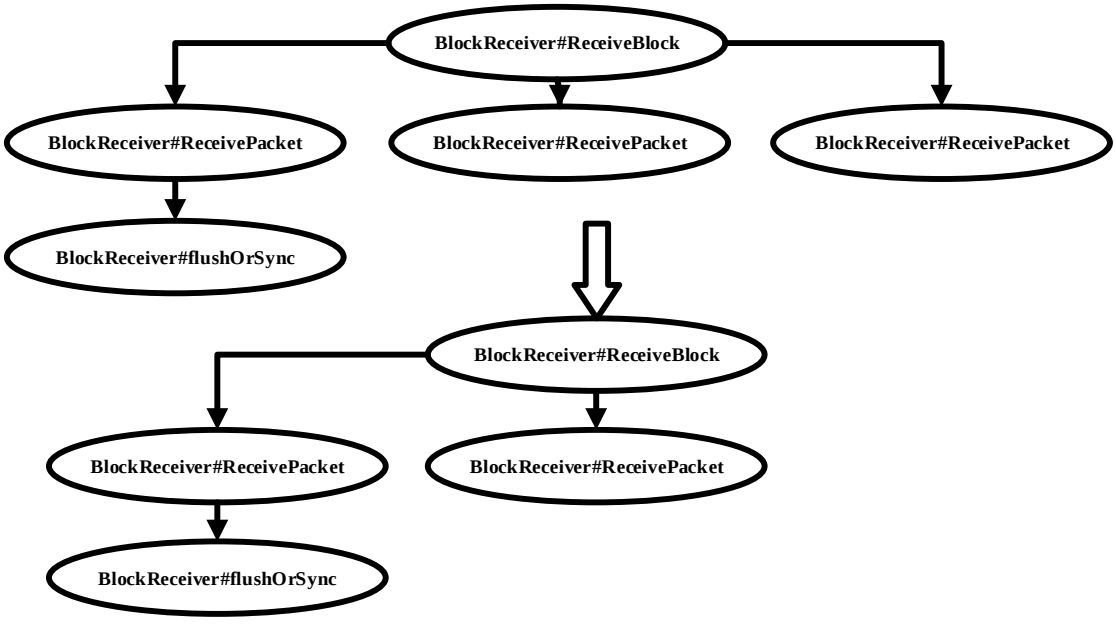


图 30 HTrace 日志调用树压缩示意图

在获取性能日志文件之后，我们首先将 HTrace 的时间持续时间利用父子关系还原成多个调用树结构，这些树结构规模十分庞大，不便于分析，因此我们采用一种同构树压缩的算法进行大规模压缩。如图 30 所示，在进行压缩前，同一个函数可能会发生多次调用，（尤其是对于数据节点发送 packet 来说，一个数据块可能要发送成千上万的 packet，每次发送都会写入日志，造成大量的信息冗余，同理，调用发送数据块的函数

也会包含大量的重复和冗余），但是这些重复的调用会使得树结构过于庞大，不利于可视化和规约分析，因此我们会宽度优先遍历树结构，递归查找子树结构，一旦发现同构的子树，就将其压缩成一棵子树，将性能信息封装到合并后的节点。在树内进行压缩后再对所有的树进行压缩，这种方法能将整个森林的规模压缩数百倍，为后面进行统计分析作下铺垫。详细的压缩算法如图 31 所示。

在进行压缩之后，还可以对压缩节点的数据进行规约，比如只保留函数执行时间的均值、方差、极值、分位点等统计信息。经过压缩后不仅数据规模大幅下降，同时也方便提取典型 I/O 模式以及每种 I/O 模式下的性能瓶颈。

5.3 不同负载下 I/O 瓶颈

5.3.1 负载生成

我们选用两种负载模式用于我们的实验，一种是 Hibench 产生的大数据应用的工作负载，一种是给基于阿里云 HDFS 请求分析得到统计学模型，经过合理缩放得到的负载**错误!未找到引用源。**。第一种负载是为了测试经典的大数据应用程序的 I/O 请求，也是本文在第三章和第四章所采用的方法，但是这种负载和工业生产环境下的实际的 I/O 是有所不同的，生产环境下的 HDFS 集群往往同时运行着许多的应用程序，而且应用程序的和访问文件的类型也不是 Hibench 所能模拟的，所以我们采用另外一种思路，直接进行 I/O 负载生成。阿里云是亚洲最大的云服务提供商，其 HDFS 集群的负载包含 web 访问和大数据计算，[30] 对被访问的文件大小，访问间隔，访问频率，访问会话进行建模，用特定的概率分布准确拟合了这些特征，如图 32-35 所示。由于会话大小是每天随着时间而变化的，而在某一个具体的时刻，会话大小都是固定的，所以本文将会话大小设定为固定值。我们在我们的集群上也实现了这种概率分布特征直接生成 I/O 请求，并对访问频率进行了归一化，由于阿里云有几千个节点，而我们的集群只有 5 个节点，所以我们对采样的 IOPS 进行了缩放，这样既能控制访问频率，也能有效地适应我们的集群规模。具体来说，本文采用公式（5.1）所示的 LogNormal 分布采样 IOPS，本文设定$\mu = 6.5$，$\sigma = 0.8$；本文采用公式（5.2）所示得到 Pareto 分布采样请求间隔，本文设定$x_m = 1$，$\alpha = 5$；文件大小的采样也是基于 LogNormal 分布，但是$\mu = 8.05$，$\sigma = 2.58$。

$$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(lnx-\mu)^2}{2\sigma^2}} \quad (5.1)$$

$$\mathrm{p(x)} = \frac{\alpha x_m^{\alpha}}{x^{\alpha+1}} \quad (5.2)$$

为了能够产生高并发，我们采用 Intel Xeon Phi 作为负载请求节点，该节点拥有 250GB 内存，64 核 128 线程，能够同时启动多个 HDFS 客户端进行访问。[30] 采用 pydoop 作为生成负载的框架，但是本文的实验发现 pydoop 对多进程访问支持不好，所以我们采用了原生的 Hadoop 进行 HDFS 文件操作，由于整个访问请求的调度系统是用 Python3 实现的，对于 Java 和 Python 的通信，我们选择了 socket 作为通信工具，为了有效保证高并发，我们在 Java 服务器端采用了线程池的模型，每有一个新的请求，便启动一个线程初始化 HDFS 客户端访问 HDFS 集群。虽然本文使用的负载生成节点 Intel Xeon Phi 具备足够多的内存和 CPU 核，但是由于 Linux 系统对线程数的限制，而 Hadoop 客户端进程又要启动很多子线程，所以依然无法实现很高的并发访问请求，所以本文修改了 Linux 下的/etc/security/limits.conf 和/etc/security/limits.d/20-nproc.conf，将 nproc 调成 65535。本文的系统的总体结构如图 36 所示。负载生成器用于生成 HDFS 访问请求，整个 HDFS 集群替换成经过插桩的 HDFS，访问完毕搜集集群的日志到数据库，然后根据用户配置解析调用树，最后执行轨迹压缩和可视化。本系统的一个典型配置如附录 3 所示。

Algorithm The call tree compress algorithm.

```
Procedure isSame: {Return whether the two trees have the same structure.}
Input: Node1, Node2
if Node1.name!=Node2.name or Node1.children.length!=Node2.children.length
then
  return false
end if
length=Node1.children.length
for i in range(length) do
  if not isSame(Node1.children[i],Node2.children[i]) then
    return false
  end if
end for
return true

Procedure merge: {Merge two trees}
Input: Node1, Node2
Node1.attr.extend(Node2.attr)
length=Node1.children.length
for i in range(length) do
  merge(Node1.children[i],Node2.children[i])
end for

Procedure compress: {Compress a tree.}
Input: Root
for Child in Root.children do
  compress(child)
end for
newChildren=[]
for i in range(Root.children.length-1 do
  if isSame(Root.children[i],Root.children[i+1]) then
    merge(Root.children[i],Root.children[j])
    Root.children[i]=None
  end if
end for
Remove None in Root.children
```

图 31 调用树压缩算法

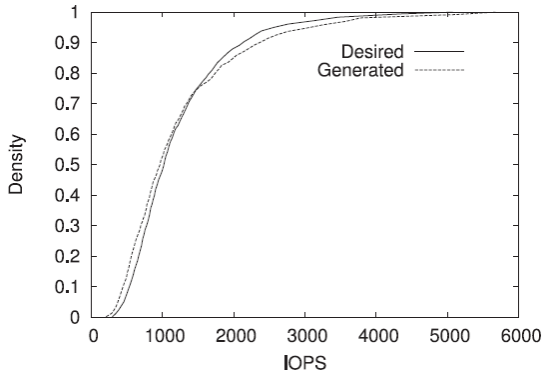
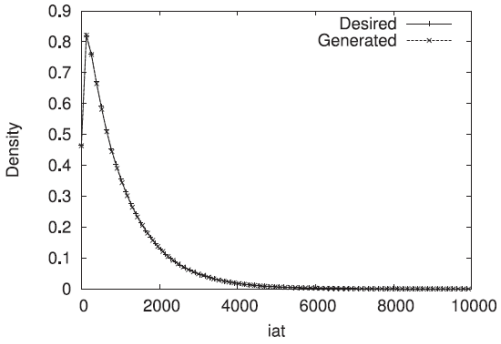
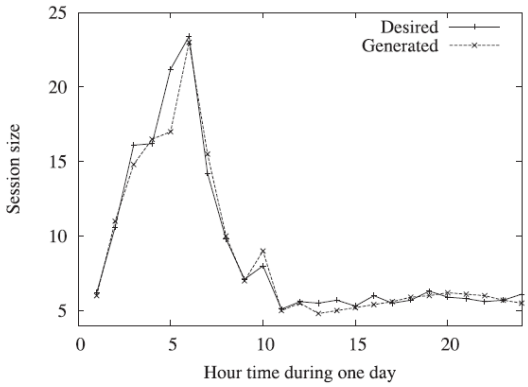


图 32 阿里云 IOPS 的概率分布图



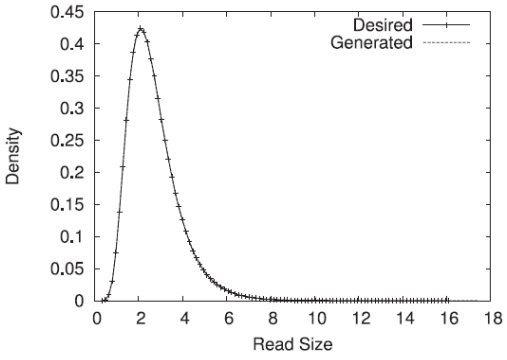
(b) Inter-arrival time (us).

图 33 阿里云访问间隔的概率分布



(c) Session size over hour time.

图 34 阿里云会话大小的概率分布



(d) Read Request Size Distribution (10KB).

图 35 阿里云被访问的文件大小的概率分布

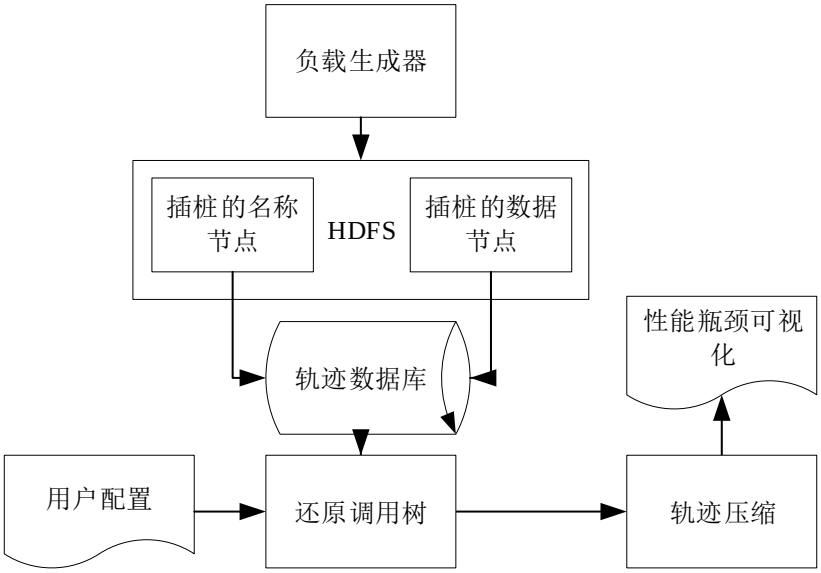


图 36 HDFS 性能分析系统各个模块示意图

5.3.2 不同负载下瓶颈分析

本文的实验主要从以下几个角度探究不同负载下 HDFS 的性能瓶颈，研究不同大数据应用程序 HDFS 低效行为的不同，研究负载大小对 HDFS 低效行为的影响，研究不同访问频率对 HDFS 低效行为的影响。

1）本文选择 Hibench 小规模（tiny）的输入的不同负载分析 HDFS 的性能瓶颈的不同。对于机器学习负载，数据一般会迭代很多次，从而会产生较大的日志文件，不便于分析，因此本文采用随机采样的方法（概率为 0.05）减少日志大小。对于 Wordcount 负载，最大的时延发生在创建文件系统的函数过程（FileSystem#createFileSystem），而数据传输占用的时间非常少，在数据写入 HDFS 的过程中（DFSOutputStream#close，如图 37 所示）实际用于 I/O 和网络传输的时间只占用了不到 14%。这些充分说明对于小数据集的微基准负载，I/O 和网络不是性能瓶颈，而文件系统的创建、等待是主要的瓶颈。对于 Sort 负载、Nweight 负载、Terasort 负载也有类似的结论。但是，FileSystem＃createFileSystem 的平均延迟更多比 Wordcount 负载大 2 倍。

Pagerank 负载与上述负载不同。Pagerank 是一个用于网页检索的算法，需要加载大量的边和节点，但是写入 HDFS 的数据量较小，所以性能瓶颈主要在 FileSystem#createFileSystem 和 newDFSInputStream，后者主要负责从名称节点获取数据块的信息，但是在名称节点感知到的延迟远远小于客户端节点感知到的延迟（分别为 0.02s 和 1.66s），所以性能瓶颈主要在客户端节点。

对于 Kmeans 负载，最大的延迟依然是 FileSystem#createFileSystem，造成了 7.34s 的延迟，DFSInputStream#byteArrayRead 是第二大的延迟，这个函数直接从本地 BlockReader 读取数据，所以没有子函数。对于 LogisticRegression 负载，最大的延迟依然是 FileSystem#createFileSystem，造成了 16.83s 的延迟，但是第二个的延迟是 getFileInfo，负责从名称节点读取文件元信息。Bayes 负载最大的延迟由 BlockSender#sendBlock 造成，从远程节点和本地节点读取数据分别造成 3.10s 和 0.49s 的延迟。

2）我们使用 Wordcount 负载来研究文件大小对 HDFS 性能的影响。我们使用 tiny, small, large, huge 大小的数据量，分别包含 32000,320000000,3200000000,320000000 个单词，分别使用 1, 0.01, 0.001, 0.0001 的采样率。随着数据量的增加，FileSystem#createFileSystem 对性能的影响越来越小。在 tiny 大小的负载中，与应用程序时间 28s 相比，该函数导致总延迟 91.92 秒（我们将来自数据节点的延迟加起来）。在

small 大小的负载中，与应用程序时间 32s 相比，该函数的延迟为 1.55 秒。在 large 大小的负载中，该函数未被采样到。

文件大小确实会对 HDFS 的性能瓶颈，Spark 的文件访问主要调用 DFSInputStream#byteArrayRead。如果数据在本地节点中，此函数将直接调用 DFSInputStream#readWithStrategy 并从本地文件系统读取数据。如果不是，则此操作将触发更多的调用树。本地文件访问在大数据应用程序中非常重要。但是本地文件访问性能几乎与文件大小无关。在 tiny, small, large, huge 大小的负载中，本地访问的延迟分别为 $3.23*10^{-4}$，$3.12*10^{-5}$，$2.57*10^{-5}$, $3.50*10^{-5}$。在数据量较大的负载会发送更少的 packet，意味着数据量较大的负载应该着重优化本地 I/O 性能。

3）在[30] 中，作者直接对 AliCloud 上的实际请求模式建模 IOPS，到达间隔时间，会话大小和读取请求大小。但是，Alicloud 是一个包含数万个节点的非常大的集群。对于我们的小型集群，我们将 IOPS 乘以不同的因子α，但保留模型的分布。随着请求频率的增加，我们可以研究 HDFS 的性能瓶颈的变化。

该模型的请求的延迟主要是由 sendBlock 函数引起的。但是，此函数的平均延迟会随着请求频率的增加而减少，如表 15 所示。尽管 FileSystem #createFileSystem 在请求延迟中起着重要作用，但其持续时间与请求频率几乎没有关系。可以发现 sendBlock 的延迟随请求频率先增加再减小。在小频率下，HDFS 处于冷启动状态，因此延迟相对较大。当请求频率非常大时，资源争用更加严重。BlockSender#sendPacket，FSNamesystem#getBlockLocations 函数具有相同的结论。频繁请求下的性能瓶颈既不是名称节点也不是数据节点，而是客户端节点。因此，优化客户端节点中的并发请求更为重要。

表 15 不同函数的延迟随 α 的变化

函数	$\alpha = 0.001$	$\alpha = 0.005$	$\alpha = 0.01$	$\alpha = 0.05$	$\alpha = 0.1$	$\alpha = 0.5$
FileSystem# createFileSystem	0.1567	0.1842	0.1827	0.2060	0.1887	0.1992
sendBlock	0.0054	0.0026	0.0020	0.0012	0.0009	0.0015
BlockSender# sendPacket	0.0005	0.0004	0.0002	0.0001	0.0001	0.0002
FSNamesystem# getBlockLocations	0.0088	0.0006	0.0005	0.0003	0.0007	0.0014
全部	0.0423	0.0439	0.0426	0.0648	0.0801	0.0674

平均数据处理速率随着访问频率的增加而降低。在不同的α下，数据处理速率分别为6.66×10^5, 5.43×10^5, 4.37×10^5, 3.28×10^5, 2.27×10^5, 3.65×10^5。由于平均请求文件大小相同（遵循相同的分布），我们可以得出结论，较不频繁的请求将生成更多空包或小包，导致了平均数据处理速率的下降。

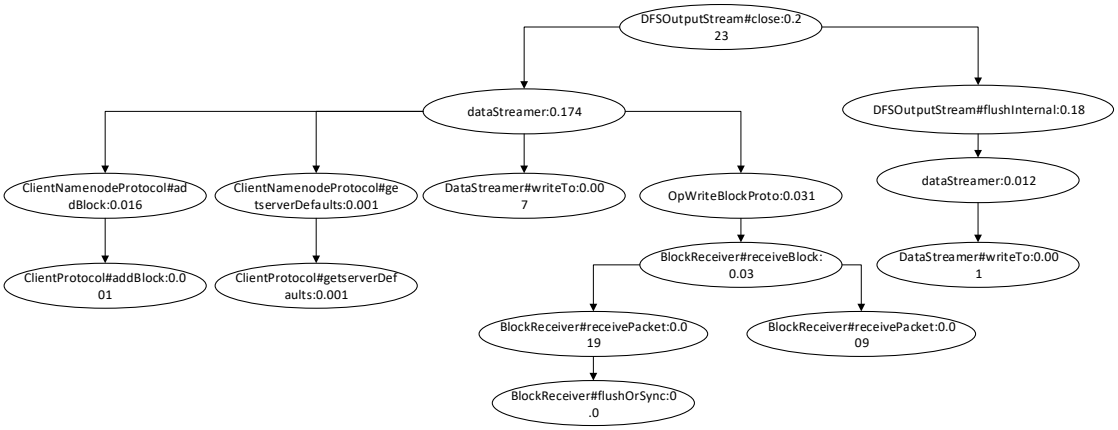


图 37　HDFS 在运行 Wordcount 时 DFSOutputStream#close 的函数调用关系

表 16　运行 HiBench 排序算法（tiny 数据量）获得的部分函数性能数据（时间持续时间）

函数名称	均值	标准差	最小值	最大值	0.25 分位点	0.75 分位点	调用次数	总时长（s）
FileSystem#createFileSystem	0.3712	0.2888	0.0	0.874	0.001	0.616	249	92.444
BlockReceiver#receiveBlock	0.6553	3.3579	0.005	18.738	0.02	0.041	30	19.66
BlockReceiver#receivePacket	0.0715	0.3250	0.0	4.897	0.002	0.025	272	19.46
DFSOutputStream#close	0.2491	0.1275	0.004	0.782	0.188	0.292	52	12.955
…								
sendBlock	0.0027	0.0034	0.001	0.023	0.001	0.002	51	0.141
…								

表 17　运行 HiBench 排序算法（tiny 数据量）获得的部分典型函数调用树性能数据（时间持续

时间）

树结构（先序遍历）	函数名称	平均值	标准差	最小值	最大值	0.25分位点	0.75分位点	调用次数	总时长（s）
FileSystem#createFileSystem	FileSystem#createFileSystem	0.3712	0.2888	0.0	0.874	0.001	0.616	249	92.444
newStreamForCreate ClientNamenodeProtocol#create FSNamesystem#removeBlocks	newStreamForCreate	0.1177	0.069	0.004	0.272	0.004	0.011	52	6.121
	ClientNamenodeProtocol#create	0.0222	0.0138	0.003	0.078	0.003	0.009	52	1.159
	FSNamesystem#removeBlocks	0	0	0	0	0	0	2	0
…									

5.4 本章小结

本章以 HDFS 为例介绍了分布式文件系统的性能分析方法，首先本文介绍了分布式文件系统可能存在的问题，然后介绍了分布式插桩的方法和具体的插桩工作，最后我们在不同负载下分析了 HDFS 性能瓶颈的不同。

第六章 原型系统设计与实现

本章根据第三章、第四章和第五章提出的性能分析方法，设计并实现了原型系统。本章将详细介绍系统架构和各个组件。

6.1 系统功能和架构设计

本原型系统主要以 Spark 为例分析大数据应用在应用层面、框架层次和分布式文件系统层次的低效行为。在不同的层次，有着不同的插桩方法和性能分析方法，三个层次工程上有一定的独立性，但是又包含相同的数据流程，整个系统架构（数据流和功能模块）如图 38 所示。整个系统的运行流程如图 39 所示。

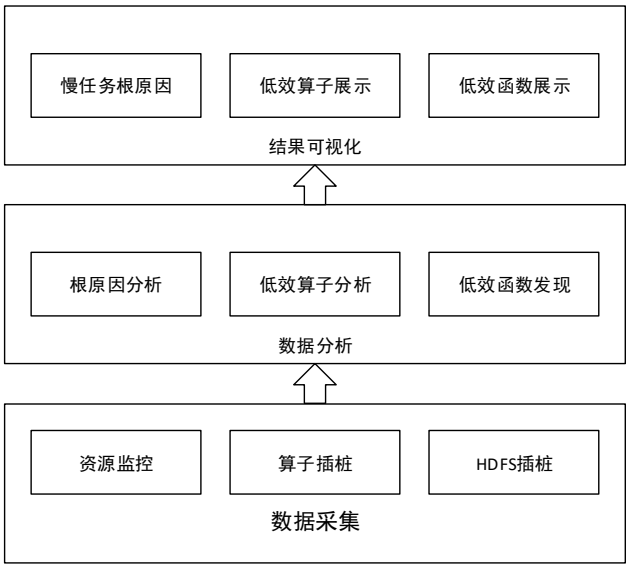


图 38 系统架构图

6.2 性能采集模块设计

性能采集模块主要分为三个部分，第一个部分是应用层数据采集，在应用层面主要有两个性能信息来源，一个是 Spark 自身的日志，Spark 在运行的时候会自动记录 JVM 垃圾搜集的时间、任务处理的数据量的大小、任务混洗的数据量大小等，所以我们只需要搜集集群的资源性能就可以了。采用的方法是在集群主节点和计算节点都启动 Linux 系统自带的 mpstat（搜集 CPU 占用率信息），iostat（搜集 I/O 信息），sar（搜集网卡信息），这些工具每隔一秒记录一次系统资源状态并写入日志。第二个部分是框架层数据

采集，我们采用 Byteman 在所有算子开始执行和结束执行的时候采集算子的性能信息。Spark 支持在 Executor（Spark 的计算进程）和 Driver（Spark 主进程，负责调度计算任务）启动时增加 JVM 选项，本文在 spark-defaults.conf 配置如附录 5 所示。

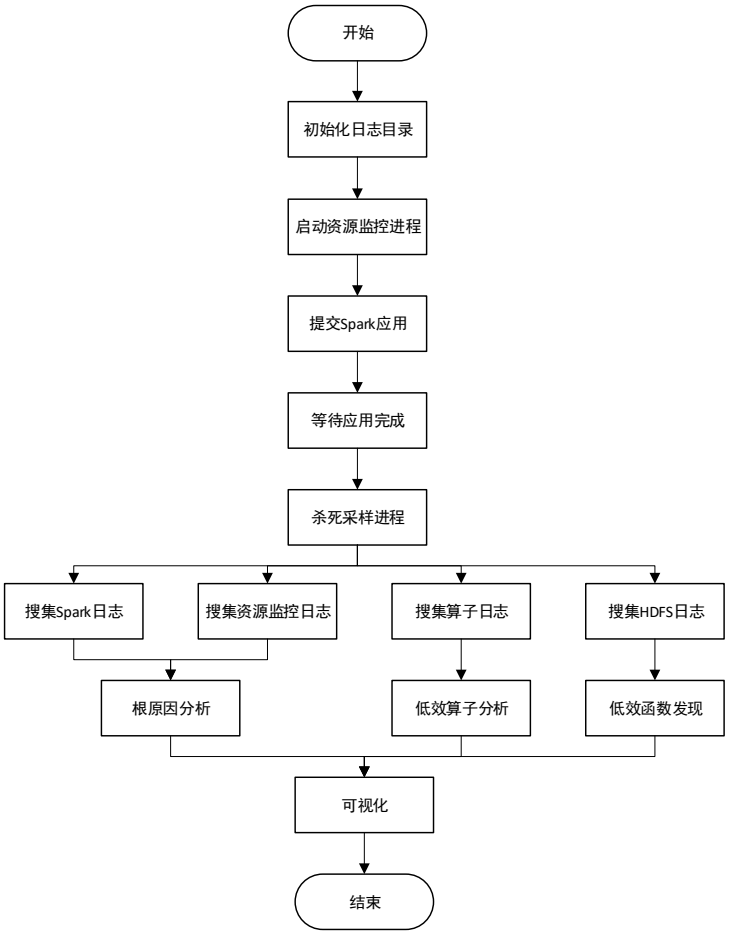


图 39 性能分析工作流程图

但是这样 Spark 无法识别我们的自定义插桩辅助函数，所以还需要在 spark-env.sh 中配置：

export SPARK_DIST_CLASSPATH=$(hadoop classpath):/home/zhg/bigdata/framework/scripts

第三个部分是分布式文件系统层次的性能搜集，我们在本地采用 HTrace 插桩 HDFS 源代码，编译之后部署在集群中，然后在 Hadoop 中配置 HTrace 将日志写入本地文件。

6.3 数据分析模块设计

数据分析模块同样分为三个部分。第一个部分是应用层面的数据分析，包括部分日志解析和根原因分析两个过程。Spark 的原始日志都是 JSON 格式的，数据都是未归一化的，而且没有包含阶段的特征，所以需要对 Spark 原始日志进行解析，计算任务特征、

识别慢任务并对任务和阶段建立索引。同时，还需要解析资源监控日志文件，mpstat 的原始日志如附录 4 所示。

需要对从这些日志里面抽取资源占用特征，根据采样开始时间和应用程序开始时间校正时间戳，然后将这些特征合并到任务特征和阶段特征里面。第二个过程是利用第三章的算法进行慢任务根原因分析。第二个部分是框架层次的数据分析，同样包含两个过程，第一个过程是将原始日志解析成 Json 格式，主要是处理算子开始执行的日志、算子结束的日志匹配的问题和算子嵌套的处理。第二个过程是计算每个算子的 IS，定位低效算子。第三个部分是分布式文件系统的数据分析，包含三个过程。第一个过程是解析 HTrace 日志，将原始日志解析成调用树组成的森林。第二个过程是运行调用树压缩算法，第三个过程是对压缩后的调用树进行统计分析。

6.4 结果可视化模块

本系统的可视化模块主要通过 Web 的形式展现，Web 框架采用 Tornado。为了方便管理，本系统还提供了 Webshell 作为向后台提交任务的接口。Webshell 采用 websocket 实现，WebSocket 协议支持浏览器和服务器之间进行持续的交互，不必再采用代价较高的轮询方式，具有较低的开销，一般用于客户端和服务器之间有频繁的数据交互的场景。通过这种方式，可以在客户端和服务器之间进行双向持续对话。整个前端的设计如图 40 所示，用户可以通过 webshell 直接在后台执行命令，方便了提交任务，在后台执行任务的时候本系统会捕捉标准输出，然后一行一行地将消息反馈给前端。在应用层面，我们展示集群每个节点的资源变化、捕捉到的慢任务和分析得到的慢任务产生的根原因，如图 41 所示。在框架层次，我们展示了每个算子的基本统计信息（均值、方差、极值），IS 最高的 10%的算子分布，对低效算子进行聚类分析的结果，如图 42 所示。在分布式文件系统层次，我们展示了函数粒度和调用树粒度的低效行为以及压缩后的调用树森林，如图 43 所示。对于函数粒度的低效行为，我们统计了每个函数的调用次数、每次调用的时间、造成的总延迟，还有函数处理数据大小、处理数据的速率的分布（如果有的话）。对于调用树粒度的低效行为，我们展示了压缩后的每棵树（先序遍历进行序列化）的每个节点的延迟的分布、处理数据大小和速率的分布。如下所示：

DFSInputStream#byteArrayRead DFSInputStream#readWithStrategy
Mean | Std | Min | Max | Q_25 | Q_75 | Num | Sum
7.262759170653908e-05 | 0.0008801417359420873 | 0.0 | 0.082 | 0.0 | 0.001 | 50160 | 3.6430000000000002

Mean | Std | Min | Max | Q_25 | Q_75 | Num | Sum
2.4720893141945774e-06 | 5.0058470780784786e-05 | 0.0 | 0.002 | 0.0 | 0.0 | 50160 | 0.12400000000000001

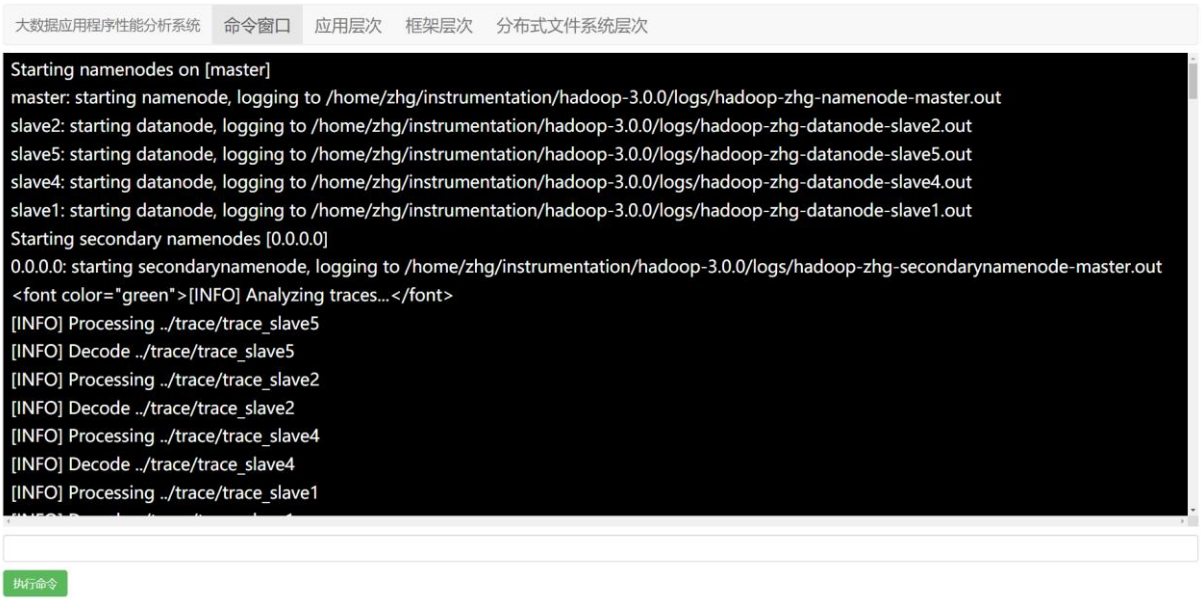


图 40 可视化交互系统

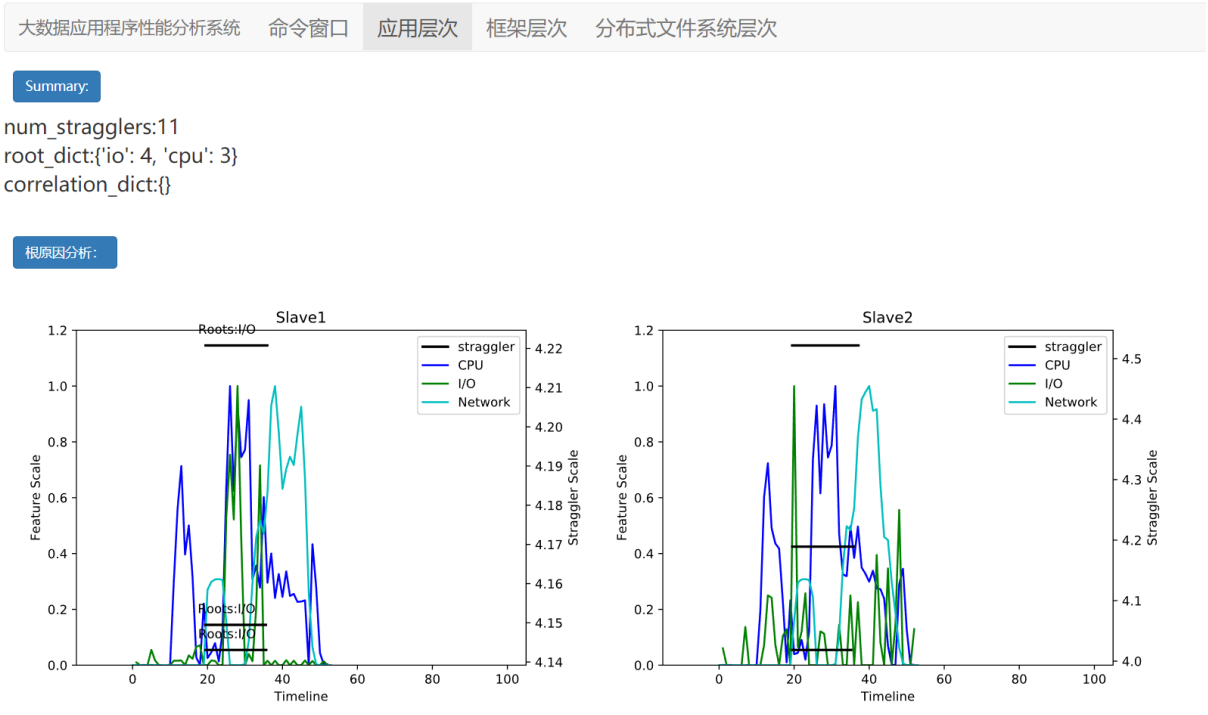


图 41 应用层面可视化

[INFO] IS statistics
ShuffledRDD[5]->54892.01
[INFO] IS distribution
ShuffledRDD->2
[INFO] Kmeans centers
BufferMemory, BufferPoolCount, CompilationTime, DaemonThreadCount, DeadLockedNum, GCCount, GCTime, HeapOccupationRate, LoadedClassCount, NoneHeapOccupationRate, ObjectPendingFinalizationCount, ThreadNum, User/CPU

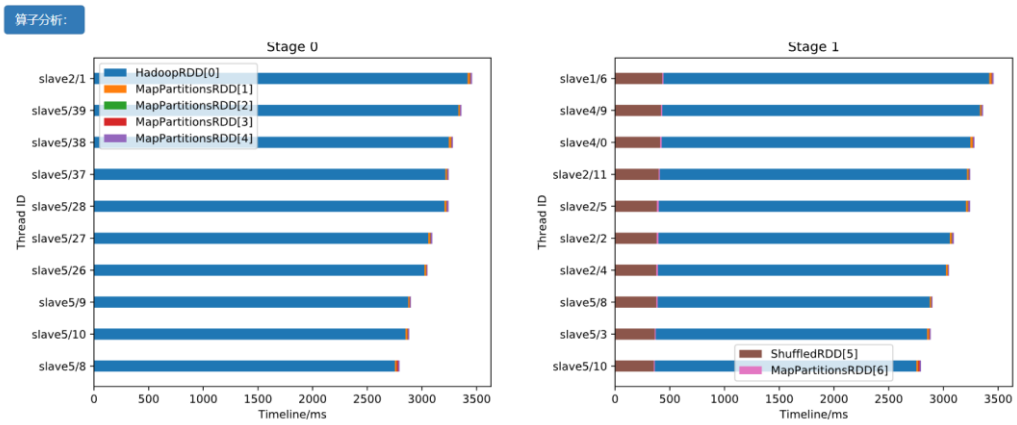


图 42　框架层次可视化

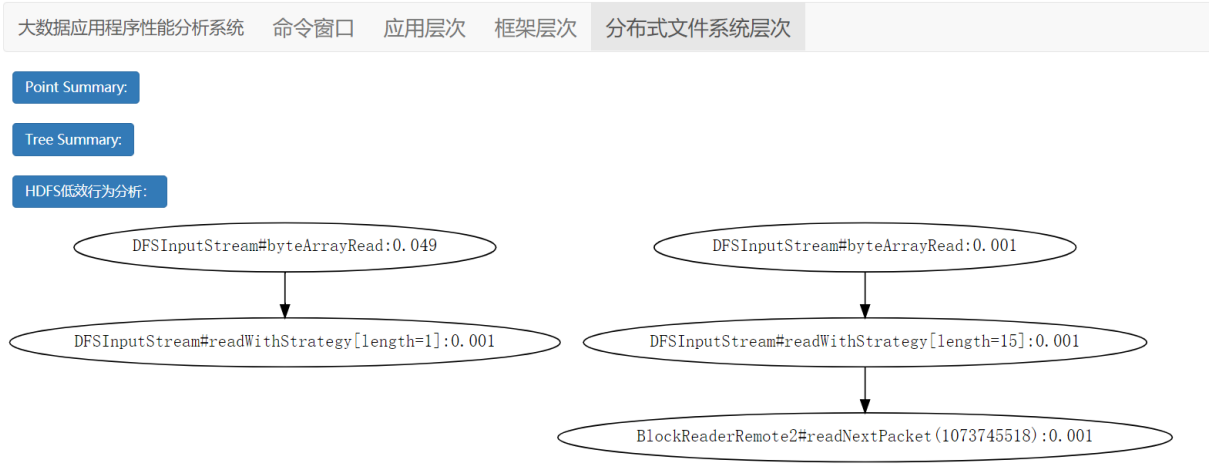


图 43　分布式文件系统层次可视化

6.5 本章小结

本章主要介绍了基于第三章，第四章和第五章提出的分层性能分析原型系统设计与实现。分别就原型系统总体架构，以及个模块的具体实现思路和方法进行了介绍。

总结与展望

论文工作总结

面向大数据应用程序的性能分析是当前研究的热点，在大数据应用十分广泛的今天，具备十分重要的现实意义。大数据应用程序的性能分析完全不同于传统的单机程序的性能分析或者高性能程序的性能分析，大数据应用程序的特点在于数据量较大、完全基于框架、层次较多、编程范型较为固定，这是传统的应用程序所不具备的。本文针对这些特点提出了分层的大数据应用程序性能分析框架，更详细地剖析了大数据应用程序在各个层次低效行为的特点和性能分析方法。

本文将大数据应用程序性能问题划分为三个层次：应用层面、框架层次和分布式文件系统层次，对于这三个层次分别在 IEEE Access 期刊，HPCC2018 国际会议，NPC2018 国际会议上发表了论文，其中应用层面的性能分析还申请了国家专利，工作得到了国际学术界的认可。

下一步工作展望

本文的工作很多问题还没有想到更加完美的解决方案，在 HDFS 层次进行插桩时，发现很多性能日志捕捉不到信息，调用树会出现一定的确实，需要进行更多的插桩或者优化 HTrace 框架本身。而且 HDFS 默认的采样器是有问题的，有的调用树较为频繁，但是有的调用树不那么频繁却对性能有很大的影响，而默认的采样器往往忽略了这些调用树的重要性，所以这方面也有优化空间。

参考文献

[1] Borthakur D. The hadoop distributed file system : Architecture and design[J]. Hadoop Project Website, 2007, 11(11):1 - 10.

[2] Zaharia, Matei, Chowdhury, et al. Spark: cluster computing with working sets[J]. 2010:10-10.

[3] Isard M, Budiu M, Yu Y, et al. Dryad:distributed data-parallel programs from sequential building blocks[C]// ACM, 2007:59-72.

[4] 李萌. 大数据系统性能分析工具的设计与实现[D]. 北京交通大学, 2014.

[5] 祁未晨. 面向大数据程序的性能分析技术研究与实现[D]. 北京航空航天大学, 2018.

[6] 杨斌. 面向神威太湖之光的 I/O 性能监测和分析诊断系统[D]. 山东大学, 2018.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters[J]. Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[8] Zaharia M, Konwinski A, Joseph A D, et al. Improving MapReduce performance in heterogeneous environments[C]// Usenix Conference on Operating Systems Design and Implementation. USENIX Association, 2008:29-42.

[9] Ananthanarayanan G, Kandula S, Greenberg A, et al. Reining in the outliers in map-reduce clusters using Mantri[C]// Usenix Conference on Operating Systems Design and Implementation. USENIX Association, 2010:265-278.

[10] Ananthanarayanan G, Ghodsi A, Shenker S, et al. Effective straggler mitigation: attack of the clones[J]. Proc Nsdi, 2013, 21(10):185-198.

[11] Frank A, Frank A, Hillel E, et al. Predicting execution bottlenecks in map-reduce clusters[C]// Usenix Conference on Hot Topics in Cloud Ccomputing. USENIX Association, 2012:18-18.

[12] X. Ouyang, P. Garraghan, R. Yang, et al. Reducing late-timing failure at scale: straggler rootcause analysis in cloud datacenters[C]// Fast Abstracts in the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, 2016.

[13] Garraghan P, Ouyang X, Yang R, et al. Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters[J]. IEEE Transactions on Services Computing, 1939, PP(99):1-1.

[14] Shende, Sameer S, and A. D. Malony. The Tau Parallel Performance System[J] International Journal of High Performance Computing Applications 20.2(2006):287-311.

[15] Chabbi M, Liu X, Mellor-Crummey J. Call Paths for Pin Tools[C]// Ieee/acm International Symposium on Code Generation and Optimization. ACM, 2014:76-86.

[16] 杨伟光, 李文. 使用 MPI 的并行 I/O 实现及性能分析[J]. 计算机工程与应用, 2006, 42(17):96-98.

[17] Ren Z, Shi W, Wan J, et al. Realistic and Scalable Benchmarking Cloud File Systems: Practices and Lessons from AliCloud[J]. IEEE Transactions on Parallel & Distributed Systems, 2017, PP(99):1-1.

[18] Y. Chen, S. Alspaugh, and R. H. Katz. Design insights for Map-Reduce from di-verse production workloads[J]. EECS Dept., Univ.California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2012–17, Jan. 2012.

[19] Ren Z, Wan J, Shi W, et al. Workload Analysis, Implications, and Optimization on a Production Hadoop Cluster: A Case Study on Taobao[J]. IEEE Transactions on Services Computing, 2014, 7(2):307-321.

[20] Zaharia M, Borthakur D, Sarma J S, et al. Delay scheduling:a simple technique for achieving locality and fairness in cluster scheduling[C]// European Conference on Computer Systems. ACM, 2010:265-278.

[21] 王志翔. 面向高并发数据访问的并行 I/O 性能优化机制研究[D].华中科技大学,2015.

[22] 郭梦影,蒋德钧,陈静,熊劲. 基于虚拟化平台的 Hadoop 应用 I/O 性能分析[J]. 计算机研究与发展,2015,52(S2):155-162.

[23] Ghemawat S, Gobioff H, Leung S T. The Google file system[J]. Acm Sigops Operating Systems Review, 2003, 37(5):29-43.

[24] Sigelman B H, Barroso L A, Burrows M, et al. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure[J]. 2011.

[25] Thereska E, Salmon B, Strunk J, et al. Stardust:tracking activity in a distributed storage system[C]// Joint International Conference on Measurement & Modeling of Computer Systems. ACM, 2006:3-14.

[26] Fonseca R, Porter G, Katz R H, et al. X-trace: a pervasive network tracing framework[C]//

NSDI. 2007:20-20.

[27] Barham P, Donnelly A, Isaacs R, et al. Using magpie for request extraction and workload modelling[C]// Conference on Symposium on Opearting Systems Design & Implementation. USENIX Association, 2004:18-18.

[28] Huang S, Huang J, Dai J, et al. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis[C]// IEEE, International Conference on Data Engineering Workshops. IEEE, 2010:41-51.

[29] Magalhaes J P, Silva L M. Detection of Performance Anomalies in Web-Based Applications[C]// IEEE International Symposium on Network Computing and Applications. IEEE, 2010:60-67.

[30] Ren Z, Shi W, Wan J, et al. Realistic and Scalable Benchmarking Cloud File Systems: Practices and Lessons from AliCloud[J]. IEEE Transactions on Parallel & Distributed Systems, 2017, PP(99):1-1.

[31] Lai C A, Kimball J, Zhu T, et al. milliScope: A Fine-Grained Monitoring Framework for Performance Debugging of n-Tier Web Services[C]// IEEE, International Conference on Distributed Computing Systems. IEEE, 2017:92-102.

[32] Arefin A, Singh V K, Jiang G, et al. Diagnosing Data Center Behavior Flow by Flow[C]// IEEE, International Conference on Distributed Computing Systems. IEEE, 2013:11-20.

[33] Al-Fares M, Radhakrishnan S, Raghavan B, et al. Hedera: dynamic flow scheduling for data center networks[C]// Usenix Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, Ca, Usa. DBLP, 2010:281-296.

[34] Li H, Ghodsi A, Zaharia M, et al. Tachyon:Reliable, Memory Speed Storage for Cluster Computing Frameworks[C]// 2014:1-15.

[35] Reiss C, Tumanov A, Ganger G R, et al. Heterogeneity and dynamicity of clouds at scale:Google trace analysis[C]// ACM Symposium on Cloud Computing. ACM, 2012:1-13.

[36] Shi J, Qiu Y, Minhas U F, et al. Clash of the titans: MapReduce vs. Spark for large scale data analytics[J]. Proceedings of the Vldb Endowment, 2015, 8(13):2110-2121.

[37] Kwon Y C, Balazinska M, Howe B, et al. SkewTune in action: mitigating skew in MapReduce applications[J]. Proceedings of the Vldb Endowment, 2012, 5(12):1934-1937.

[38] Chen Q, Zhang D, Guo M, et al. SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment[C]// IEEE International Conference on Computer and Information Technology. IEEE Computer Society, 2010:2736-2743.

[39] Lin C, Guo W, Lin C. Self-Learning MapReduce Scheduler in Multi-job Environment[C]// International Conference on Cloud Computing and Big Data. IEEE, 2014:610-612.

[40] Macqueen J. Some Methods for Classification and Analysis of MultiVariate Observations[C]// Proc. of, Berkeley Symposium on Mathematical Statistics and Probability. 1965:281-297.

[41] Yang H, Luan Z, Li W, et al. Statistics-based Workload Modeling for MapReduce[C]// Parallel and Distributed Processing Symposium Workshops & Phd Forum. IEEE, 2012:2043-2051.

[42] Lin Q, Zhang H, Lou J G, et al. Log Clustering Based Problem Identification for Online Service Systems[C]// Ieee/acm International Conference on Software Engineering Companion. IEEE, 2016:102-111.

附录

Byteman 插桩脚本

附录1：

```
RULE trace compute start
CLASS ^org.apache.spark.rdd.RDD
METHOD compute
HELPER MainHelper
AT ENTRY
IF true
DO
traceln("tag=[RDD]"+$CLASS+";flag=start;timestamp="+System.currentTimeMillis()+";stage="+$context.stageId()+";thread="+Thread.currentThread().getId()+";this="+$0+";partitionId="+$context.partitionId());
ENDRULE

RULE trace compute end
CLASS ^org.apache.spark.rdd.RDD
METHOD compute
HELPER MainHelper
AT EXIT
IF true
DO
traceln("tag=[RDD]"+$CLASS+";flag=end;timestamp="+System.currentTimeMillis()+";stage="+$context.stageId()+";thread="+Thread.currentThread().getId()+";this="+$0+";partitionId="+$context.partitionId());
ENDRULE
```

算子插桩日志

附录2：

```
tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=start;timestamp=1537215237018;stage=0;thread=59;this=MapPartitionsRDD[4] at ; partitionId=6…
```

tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=start;timestamp=15 37215237031;stage=0;thread=59;this=MapPartitionsRDD[3] at ; partitionId=6…

tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=start;timestamp=15 37215237032;stage=0;thread=59;this=MapPartitionsRDD[2] at ; partitionId=6…

tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=start;timestamp=15 37215237033;stage=0;thread=59;this=MapPartitionsRDD[1] at ; partitionId=6…

tag=[RDD]org.apache.spark.rdd.HadoopRDD;flag=start;timestamp=1537215 237040;stage=0;thread=59;this=HadoopRDD[0] at;partitionId=6…

tag=[RDD]org.apache.spark.rdd.HadoopRDD;flag=end;timestamp=15372152 40323;stage=0;thread=59;this=HadoopRDD[0] at;partitionId=6…

tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=end;timestamp=153 7215240330;stage=0;thread=59;this=MapPartitionsRDD[1] at ; partitionId=6…

tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=end;timestamp=153 7215240331;stage=0;thread=59;this=MapPartitionsRDD[2] at ; partitionId=6…

tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=end;timestamp=153 7215240331;stage=0;thread=59;this=MapPartitionsRDD[2] at ; partitionId=6…

tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=end;timestamp=153 7215240332;stage=0;thread=59;this=MapPartitionsRDD[3] at ; partitionId=6…

tag=[RDD]org.apache.spark.rdd.MapPartitionsRDD;flag=end;timestamp=1537215240 333;stage=0;thread=59;this=MapPartitionsRDD[4] at ; partitionId=6…

附录3：

HDFS 日志解析配置文件示例

[default]

 # 插桩函数作用域

domain=namenode datanode client local

domain_color=green blue white yellow

 # 是否压缩调用树

compress=true

showMyInstrumentOnly=false

 # 是否运行统计分析

observer=true

```
# 是否打印进度条
enable_progress=true
# 是否排除一些插桩点
enable_exclude=false
[client]
# 客户端节点包含的函数
include=DFSInputStream#openInfo
    DFSInputStream#fetchBlockAt
    DFSInputStream#readWithStrategy
    DFSInputStream#actualGetFromOneDataNode
[datanode]
# 数据节点包含的函数
include=BlockSender#doSendBlock
    BlockSender#sendPacket(writeToSocket)
    BlockReceiver#flushOrSync
    BlockReceiver#receivePacket
    BlockReceiver#receiveBlock
[namenode]
# 名称节点包含的函数
include=FSNamesystem#getBlockLocations
    FSNamesystem#renameTo
    FSNamesystem#delete
    FSNamesystem#removeBlocks
    FSNamesystem#getFileInfo
    FSNamesystem#mkdirs
[compress]
# 压缩方式
compress.level=forest|tree
[observer]
# 统计方法
```

include=pointobserver treeobserver

[exclude]

 # 需要排除的插桩点（由于老旧、重复）

include=OpWriteBlockProto

dataStreamer

waitForAckedSeqno

CPU 采样原始日志

Linux 3.10.107-1.el6.elrepo.x86_64 (slave1) 2018 年 09 月 18 日 _x86_64_ (16 CPU)

附录4：

04 时 13 分 30 秒

CPU	*%usr*	*%nice*	*%sys*	*%iowait*	*%irq*	*%soft*	*%steal*	*%guest*	*%idle*
04 时 13 分 31 秒	*all*	*0.00*	*0.00*	*0.12*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
99.88									
04 时 13 分 31 秒	*0*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
100.00									
04 时 13 分 31 秒	*1*	*0.00*	*0.00*	*0.99*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
99.01									

附录5：

Spark 配置

spark.driver.extraJavaOptions -javaagent:/home/zhg/instrumentation/site-packages/byteman-download-4.0.0/lib/byteman.jar=script:/home/zhg/bigdata/framework/scripts/main.btm

spark.executor.extraJavaOptions -javaagent:/home/zhg/instrumentation/site-packages/byteman-download-4.0.0/lib/byteman.jar=script:/home/zhg/bigdata/framework/scripts/main.btm

攻读硕士学位期间取得的学术成果

攻读硕士期间，发表学术论文 4 篇，申请发明专利 1 项。其中 SCI Q1 期刊论文 1 篇，CCF C 类会议论文 3 篇。学术成果列表如下：

（1） Zhou H, Li Y, Yang H, et al. BigRoots: An Effective Approach for Root-cause Analysis of Stragglers in Big Data System[J]. IEEE Acess, 2018.

（2） Zhou H, Li Y, Jia J, et al. SparkOT: Diagnosing Operation Level Inefficiency in Spark[C]// IEEE, International Conference on High PERFORMANCE Computing and Communications; IEEE, International Conference on Smart City; IEEE, International Conference on Data Science and Systems. IEEE, 2018

（3） Liu Y, Li Y, Zhou H, et al. A Fine-grained Performance Bottleneck Analysis Method for HDFS[C]// IFIP International Conference on Network and Parallel Computing. Springer, Cham, 2018.

（4） Qi W, Li Y, Zhou H, et al. Data Mining Based Root-Cause Analysis of Performance Bottleneck for Big Data Workload[C]// IEEE, International Conference on High PERFORMANCE Computing and Communications; IEEE, International Conference on Smart City; IEEE, International Conference on Data Science and Systems. IEEE, 2017.

（5） 杨海龙, 周红刚, 李云春. 一种针对大数据平台的慢任务原因检测方法: 中国, 201711436008.2[P]

致　　谢

时光荏苒，两年半的研究生生涯将要结束，此刻满怀对北航的留恋。在计算机学院的这两年半的时间，我不仅学到了很多受益终身的知识，也认识了很多可爱可敬的人。从一个只会玩游戏的 19 系毕业生，到一个合格的工程师和研究员，我的人生改变了太多。在这个过程中，我有太多的人要感谢。

首先要感谢我的女朋友，是她给了我努力的动力，让我认清我的追求，重拾工作和生活的信心。感谢李云春老师，给我指点了正确科研的方向，给了我大量无私的帮助，让我从刚入学时计算机零基础走到了现在。同时，我还要感谢李巍老师，是李巍老师耐心的指导让我不断认识工作的不足。我还要感谢杨海龙老师，杨老师卓越的科研能力和敏锐的学术直觉让我受益匪浅，在我的科研生涯中给予了无微不至的帮助。还要感谢栾钟治老师在节日提供的关怀以及为整个实验室营造了良好的环境。另外要感谢祁未晨学长、刘一学弟，他们在科研上给了我很多的帮助。最后，我要感谢我的父母，是他们一直在背后默默为我付出。

再次对以上所有人，以及未能提及的在这两年半研究生生涯中所有帮助过我的人，表示我由衷的感谢，谢谢！