

密级：_____



中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

分布式数据并行应用内存溢出错误分析与诊断

作者姓名： 许利杰

指导教师： 魏峻 研究员

中国科学院软件研究所

学位类别： 工学博士

学科专业： 计算机软件与理论

培养单位： 中国科学院软件研究所

2015 年 10 月

Understanding and Diagnosing Causes of Out of Memory Errors
in Distributed Data-Parallel Applications

By

Lijie Xu

**A Dissertation Submitted to
University of Chinese Academy of Sciences
In partial fulfillment of the requirement
For the degree of
Doctor of Philosophy**

Institute of Software

Chinese Academy of Sciences

October 2015

学位论文独创性声明

本人郑重声明：我所呈交的学位论文是本人在导师指导下进行的研究工作及所取得的研究成果。尽我所知，除了文中已经标注引用的内容外，本论文中不含含其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中作了明确的说明或致谢。本人知道本声明的法律结果由自己承担。

学位论文作者签名: _____ 日期: _____

关于学位论文使用授权的说明

本人完全了解中国科学院软件研究所有关保留、使用学位论文的规定，即：中国科学院软件研究所有权保留送交论文的复印件，允许论文被查阅和借阅；中国科学院软件研究所可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。

(涉密的学位论文在解密后，按此规定执行。)

学生签名: _____ 导师签名: _____ 日期: _____

分布式数据并行应用内存溢出错误分析与诊断

摘要

近年来，大数据以数据量大、数据类型多样、产生与处理速度要求快、价值高的 4V 特性成为工业界和学术界的研究热点。由于传统软件系统难以在可接受的时间范围内处理大数据，工业界和学术界设计了具有高扩展性的分布式数据并行处理框架，如 MapReduce，Apache Hadoop，Apache Spark 等。目前，这些框架被广泛应用于数据密集型应用的开发和执行，如 Web 索引的建立、日志挖掘、机器学习、大规模图分析等。

分布式数据并行框架为用户提供了简单的编程模型，并能自动完成数据的并行与分布处理。这种设计方式可以让用户更关注于数据处理逻辑本身，而无需了解应用的物理执行过程。然而，如果应用在运行时出现性能问题或者运行时错误，用户难以诊断出问题原因，更难以修复错误。由于应用需要在内存中处理大量数据，内存溢出错误成为分布式数据并行应用中一种常见的运行时错误，而且该错误不能被框架的错误容忍机制修复。除了内存溢出问题，用户在正常使用框架时也面临不了解应用内存消耗，不知道该为应用配置多大内存空间的问题。

本文旨在分析和解决分布式数据并行应用中的两个内存使用问题：(1) 如何分析并估算应用的内存使用量？(2) 如何理解、诊断与修复应用内存溢出错误？针对这两个研究问题，本文从以下三个方面开展了研究：

(1) 内存溢出错误的实证分析：研究目的是分析应用内存溢出错误的常见原因，常见修复方法，并探索可以提高框架错误容忍能力的方案。我们研究了 123 个真实 Hadoop 和 Spark 应用的内存溢出错误，发现了内存溢出错误的三大原因：框架暂存的数据量过大，数据流异常以及内存使用密集的用户代码。我们也从 42 个包含修复信息的错误中总结出了常用的修复方法。另外，我们也提出了可以提升框架错误容忍能力和错误诊断能力的三种方法。

(2) 内存用量模型构建与用量估算：研究目的是构建应用的内存用量模型，解释、量化应用的静态因素（数据、配置、用户代码）与动态内存用量的关系，并估算出新应用的内存用量。我们以数据流为中心构建了应用内存用量模型，模型包含数据流模型、框架内存模型和用户代码模型。为了解决如何在用户代码未知的情况下构建用户代码内存用量与其输入数据之间关系的问题，我们在用户代码模型中设计了生命周期敏感的内存监控方法。在内存用量模型的基础上，我们通过在小数据上运行应用来估算该应用在大数据集上的内存用量。

(3) 内存溢出错误的诊断方法：研究目的是设计出内存溢出错误诊断方法及工具。

本文基于内存用量模型设计了一个内存分析器 Mprof。Mprof 可以自动建立应用静态因素与动态内存用量的关系，方法是重建应用数据流，重建用户代码内存使用历史信息，并对两者进行关联分析。Mprof 也包含定量诊断规则，这些规则根据应用静态因素与动态内存的关联关系来定位内存溢出错误相关的代码，错误相关的数据，以及不恰当的配置参数。

本文的研究成果对于分布式数据并行框架用户和框架设计者均有实际参考意义，也对后续研究工作（比如研究其他框架或应用的内存溢出问题）有借鉴意义。

关键词：分布式数据并行应用，MapReduce，内存溢出，内存用量模型，错误分析与诊断

Understanding and Diagnosing Causes of Out of Memory Errors in Distributed Data-parallel Applications

ABSTRACT

In recent years, Big Data has become a hot research topic in academia and industry for its four features: large volume, high variety, high velocity, and high value. Since traditional software systems cannot handle big data effectively and efficiently, academic and industrial researchers have designed highly scalable distributed data-parallel frameworks, such as MapReduce, Apache Hadoop and Apache Spark. Nowadays, these distributed frameworks are widely used in academia and industry to develop data-intensive applications, such as Web indexing, click-log mining, machine learning, and graph analysis.

Distributed data-parallel frameworks provide users with simple programming models and hide the details of parallel/distributed execution. This design helps users focus on the data processing logic, but burdens them when the applications running atop these frameworks have performance problems or runtime errors. Since data-parallel applications usually process large data in memory, out of memory (OOM) errors become the common and serious runtime errors in these applications. OOM errors can directly lead to the application failure, and cannot be tolerated by frameworks' fault-tolerant mechanisms. Apart from OOM errors, users are also confused about how to analyze and estimate an application's memory usage, since they need to configure the right memory limit of the application.

In this dissertation, we focus on two critical memory problems in distributed data-parallel applications: (1) How to analyze and estimate an application's memory usage? (2) How to understand, diagnose, and fix the OOM errors in distributed data-parallel applications? To solve these two problems, we have done three pieces of work as follows.

(1) Empirical study on OOM errors: This work aims to analyze the common cause patterns and fix patterns of OOM errors in distributed data-parallel applications. We performed the first characteristic study on 123 real-world OOM errors in Hadoop and Spark applications. We found that the OOM root causes are memory-consuming user code, abnormal dataflow, and large buffered/cached data. We also summarized the common fix patterns for most OOM root causes. Our findings inspire us to propose three potential solutions to improve frameworks' fault tolerance and facilitate the OOM error diagnosis.

(2) Memory usage modeling and estimation: This work aims to build a memory usage

model for distributed data-parallel applications and estimate the applications' memory usage. We propose a dataflow-centric memory usage model that can figure out the correlation between an application's static information (data, configurations, and user code) and its runtime memory usage. The model contains a dataflow model, a framework object model, and a user code model. The main challenge is to quantify the relationship between memory usage of black-box user code and its input data. We solve this problem through designing and performing a lifecycle-aware memory monitoring strategy. For memory estimation, we first run the newcomer application on sample data, and then use the memory usage model to estimate the application's memory usage on big data.

3) OOM error diagnosis: This work aims to design a tool for diagnosing the OOM errors in distributed data-parallel applications. We designed a memory-profiling tool named *Mprof*. *Mprof* can automatically figure out the correlation between an application's runtime memory usage and its static information (configurations and user code). *Mprof* achieves this through reconstructing statistical dataflow, memory usage history of user code, and performing correlation analysis on them. *Mprof* can further trace OOM errors back to the problematic user code, data, and configurations, through applying quantitative rules. *Mprof* only relies on enhanced logs, dataflow counters, and heap dumps, without any modifications to the user code.

We believe our results can help both users and the framework designers to handle OOM errors properly. The results are also useful for further research work (e.g., research on OOM errors in other systems or applications).

KEY WORDS: Data-parallel applications, MapReduce, Out of memory, Memory usage modeling, Error analysis and diagnosis

目录

第一章 绪论	1
1.1 研究背景	1
1.1.1 大数据与大数据处理	1
1.1.2 分布式数据并行处理框架	1
1.1.3 分布式数据并行应用	2
1.1.4 应用执行的三层结构	2
1.1.5 应用内存溢出错误	3
1.2 论文工作	4
1.2.1 研究动机	4
1.2.2 研究问题	5
1.2.3 研究思路	6
1.2.4 研究内容	6
1.2.5 创新点	7
1.3 论文组织	8
第二章 分布式数据并行应用及相关研究工作介绍	11
2.1 概述	11
2.2 分布式数据并行框架架构	11
2.3 应用编程模型	12
2.4 应用开发的三层结构	13
2.4.1 用户层	14
2.4.2 分布式数据流层	17
2.4.3 物理内存层	19
2.5 应用性能与内存使用问题	20
2.6 框架错误容忍机制	20
2.7 本章小结	21
第三章 数据并行应用内存溢出错误实证分析	23
3.1 概述	23
3.2 研究方法	23
3.2.1 研究对象	23
3.2.2 错误原因与修复方法的收集	25

3.2.3 错误重现方法	25
3.2.4 研究方法的局限性	25
3.3 RQ1（常见错误原因）的研究成果	26
3.3.1 错误类别 1：框架暂存了大量的中间数据	27
3.3.2 错误类别 2：数据流异常	28
3.3.3 错误类别 3：内存使用密集的用户代码	30
3.4 RQ2（常见修复方法）的研究结果	33
3.4.1 数据存储相关的修复方法	34
3.4.2 数据流相关的修复方法	34
3.4.3 用户代码相关的修复方法	36
3.4.4 Driver 程序相关的修复方法	37
3.5 RQ3（框架改进方法）的研究结果	38
3.5.1 辅助错误诊断的方法	38
3.5.2 提高框架的错误容忍能力方法	39
3.6 讨论	40
3.7 相关工作比较	40
3.8 本章小结	41

第四章 应用内存用量模型及用量估算方法.....43

4.1 概述	43
4.2 问题定义	44
4.3 挑战	44
4.4 问题分析与模型建立思路	45
4.5 内存用量模型	47
4.5.1 数据流模型	47
4.5.2 框架内存对象模型	51
4.5.3 用户代码对象模型	53
4.5.4 模型总结	59
4.6 实验评价	59
4.6.1 实验环境与实验数据	59
4.6.2 实验步骤	61
4.6.3 实验结果	61
4.7 进一步讨论	62
4.8 相关工作比较	63

4.9 本章小结	63
第五章 应用内存溢出错误诊断方法及工具	65
5.1 概述	65
5.2 问题分析	65
5.3 内存溢出诊断工具 Mprof 的设计与实现	66
5.3.1 数据流分析器	67
5.3.2 用户代码内存分析器	68
5.4 Mprof 用于内存溢出错误诊断	70
5.5 实验评价	73
5.5.1 实验环境及实验步骤	73
5.5.2 诊断结果	73
5.5.3 案例研究	75
5.6 讨论	78
5.7 相关工作比较	79
5.8 本章小结	80
第六章 结束语	81
6.1 论文工作总结	81
6.2 进一步的工作	82
6.2.1 内存溢出错误实证分析方面	82
6.2.2 应用内存用量估算方面	82
6.2.3 内存溢出错误诊断工具设计方面	82
参考文献	83
作者简历及攻读学位期间发表的学术论文与科研成果	95
参与课题	97
致谢	99

图目录

图 1 分布式数据并行应用开发执行的三层结构与内存溢出错误例子	3
图 2 论文整体研究思路	6
图 3 分布式数据并行框架部署图	12
图 4 应用开发执行的三层结构图	13
图 5 用户手写的 Hadoop MapReduce 代码	15
图 6 用户手写的 Spark 代码	15
图 7 由高层语言 Pig Latin 撰写的代码	15
图 8 Driver 程序	16
图 9 Spark job 的逻辑执行图	18
图 10 Spark 数据流	18
图 11 用户描述的内存溢出错误例子	24
图 12 专家对内存溢出错误的诊断与回答	24
图 13 不合适的数据划分和热点 key 例子	29
图 14 MapReduce job 的内存使用情况	45
图 15 应用内存用量模型的构成	47
图 16 数据流模型	48
图 17 框架内存使用情况	51
图 18 Mapper 的用户代码模型	54
图 19 Reducer 的用户代码模型	55
图 20 生命周期敏感的用户代码内存用量监控策略	57
图 21 Map/reduce task 的内存用量估算相对误差（抽样数据为 1GB）	62
图 22 Map/reduce task 的内存用量估算相对误差（抽样数据为 5GB）	62
图 23 生命周期敏感的用户代码内存用量监控策略（使用 heap dump）	69
图 24 内存溢出错误诊断过程（如果错误发生在 reduce() 中）	71
图 25 案例中的用户代码内存用量趋势图	76

表目录

表 1 一个 Hadoop reduce task 抛出的内存溢出错误栈	4
表 2 出现内存溢出错误的应用分布表	24
表 3 内存溢出错误的常见错误原因	26
表 4 内存溢出错误常见修复方法	33
表 5 数据流模型	48
表 6 提升过的 task 的日志	49
表 7 要获取或计算的参数	49
表 8 需要估计的数据流参数	50
表 9 框架对象模型（框架缓冲区）	52
表 10 框架对象模型（内存中暂存的中间数据）	52
表 11 用户代码对象模型	56
表 12 用户代码空间复杂度	58
表 13 内存用量模型总结	59
表 14 实验评价所用的 5 个典型 MapReduce 应用	60
表 15 数据流异常模型	67
表 16 用户代码内存使用模式识别方法	70
表 17 用户代码规则（用于诊断用户代码相关的错误原因和错误相关的数据）	72
表 18 数据流规则（用于诊断数据流异常和不恰当的配置参数）	73
表 19 内存溢出错误诊断结果	74
表 20 本文引用的现实世界的内存溢出错误用例	92

第一章 绪论

本章简要介绍了论文的研究背景，并分析了论文研究的三个主要问题，接下来给出了论文的研究思路和研究内容。在本章的结尾，列出了论文的组织方式。

1.1 研究背景

1.1.1 大数据与大数据处理

数据一直是工业界和学术界的重要研究对象。从数据中获益，从数据中获取知识成为数据处理的主要目的。例如，搜索引擎每天都在收集、处理、分析海量的网页及多媒体数据，并对外提供数据查询服务。社交网站每天记录大量的用户数据，组织形成虚拟的人际网络。商业智能公司依靠分析企业生产和销售的数据，来为企业提供商务决策支持。学术研究机构也在天文、物理、化学、地理、生命科学等方面不断积累大量的实验数据，从数据中分析挖掘各种科学知识。

互联网、云计算、移动计算、物联网等技术的发展使得数据的产生速度越来越快、数据规模越来越大、数据类型越来越多。诸如社交网站 Facebook 每天要处理 25 亿条消息，500+TB 的新数据，用户每天上传 3 亿张照片，每半个小时扫描的数据大约为 105TB [FBD]。早在 2008 年，Google 每天就要处理约 20,000 TB (20PB) 的数据 [GoogleD]。YouTube 网站每分钟用户会上传约 48 小时的视频 [YouTubeD]。早在 2012 年，Twitter 每天大约产生 1.75 亿条新微博 [TwitterD]。

为了描述数据的新特性，“大数据”的概念被提出。大数据具有数据量大、数据类型多样、产生与处理速度要求快、价值高的 4V 特性 [BDvalues]。这些特性也使得传统软件系统难以在可接受的时间范围内对大数据进行处理。例如，出现于 70 年代的关系数据库很好地解决了关系型数据的存储与 OLTP（在线事务处理）。之后出现的数据仓库将数据按主题组织，解决了数据建模及 OLAP（在线数据分析）。然而在大数据环境下，无论是传统的数据库还是数据仓库，都面临着可扩展性的问题，该问题导致这些传统软件难以处理大数据或者处理大数据效率低下。为了解决这个问题，工业界和学术界经过十多年的探索和实践，设计出了多种可扩展的分布式数据并行处理框架。

1.1.2 分布式数据并行处理框架

为了解决大数据高效处理问题，工业界和学术界提出了很多分布式并行处理框架。2004 年 Google 在 OSDI（操作系统设计与实现）会议上提出了基于分治、归并、函数

式编程思想的 MapReduce 分布式计算框架 [Dean2004], 获得巨大成功也受到广泛关注。随后, Apache 社区将 Google File System [Ghemawat2003] 和 MapReduce 开源实现为 Hadoop [Hadoop], 经过多年发展, Hadoop 已经形成一个完整的生态系统, 并成为业界大数据存储和处理的事实标准。2007 年微软公司提出了 Dryad 分布式计算框架 [Isard2007]。Dryad 的思路源于 MapReduce, 但其更加灵活。不同于 MapReduce 固定的数据处理流程定, Dryad 允许用户将任务处理阶段组织成 DAG (有向无环图) 来达到更强的数据处理表达能力。2012 年 UC Berkeley 的 AMPLab 提出了基于内存, 适合迭代计算的 Spark 分布式数据并行处理框架 [Zaharia2012]。该框架允许用户将公共或要重用的数据缓存到内存, 极大缩短了数据处理的时间。这些框架的共同特点是: 拥有共同的编程模型, 即 MapReduce 或者 MapReduce-like, 采用“分治一聚合”策略来对数据进行分布并行处理。

1.1.3 分布式数据并行应用

分布式数据并行应用指的是运行在分布式数据并行框架之上, 对大数据进行分布处理的作业, 比如 Hadoop MapReduce jobs, Spark jobs。分布式数据并行应用在工业界和学术界广泛存在, 比如用在网页索引的构建、日志挖掘、大数据 SQL 查询、机器学习、社交网络图分析等。

在这些分布式数据并行应用中, 用户不仅可以手写代码实现数据处理逻辑, 也可以借助构建于框架之上的高层语言或者高层库开发应用。比如, 在 Hadoop 之上, Yahoo 开发了 SQL-like 的 Pig Latin 语言 [ApachePig], 可以将 SQL-like 脚本转换成 Hadoop MapReduce jobs。Facebook 开发的分布式数据仓库 Hive [ApacheHive], 构建在 Hadoop 之上, 也可以将类 SQL 查询分析语言转变成 MapReduce jobs。Apache Mahout [ApacheMahout] 提供了基于 Hadoop MapReduce 的机器学习库。在 Spark 之上, GraphX [Gonzalez2014] 提供了面向大规模图处理的库, MLlib [SparkMLlib] 提供了面向大规模机器学习的库, Spark SQL [SparkSQL] 提供了基于 Spark 的 SQL 查询框架及语言。

1.1.4 应用执行的三层结构

按照从上到下的层次, 分布式数据并行应用的整个开发和执行过程可以被分为三层 (如图 1): 用户层、分布式数据流层和物理内存层。

用户层负责开发应用, 应用 (job) 的组成元素是 <数据, 配置参数, 用户代码>。数据指的是要处理的大数据, 数据一般提前存放在分布式文件系统 (如 Hadoop File System [HDFS] 上)。配置参数用于指定框架运行时需要的信息, 如 partition 个数、缓冲区大小。用户代码包括 map(), reduce() 以及可选实现的 combine()。

分布式数据流层负责 job 的执行, 一个 job 可以包含多个 map/reduce tasks。框架负

责启动每个 task，分布执行 task 的数据处理步骤。如在 Hadoop 中，整个数据处理阶段可以分为 map, shuffle, reduce 三个阶段。数据流指的是各个处理阶段中的输入、输出、中间数据，以及数据之间关系。每个 task 是一个进程或线程，比如 Hadoop MapReduce 中每个 task 以 JVM（Java 虚拟机）的方式来运行。

物理内存层反映应用的真实内存消耗，即 task 的内存用量。应用的内存消耗主要包括两部分：框架暂存于内存中的中间数据，用户代码在处理数据时产生的中间计算结果。

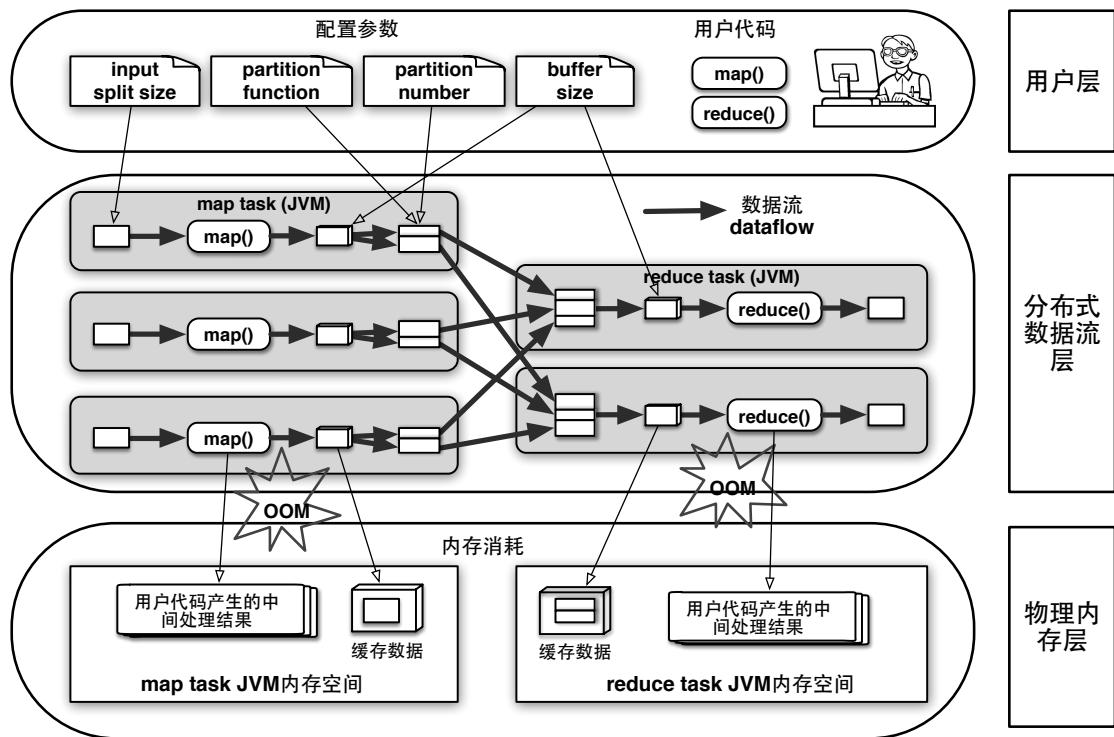


图 1 分布式数据并行应用开发执行的三层结构与内存溢出错误例子

1.1.5 应用内存溢出错误

在运行应用时，框架为了提升性能会暂存一部分中间数据到内存中，用户代码也会在处理数据时在内存中产生中间处理结果。当两者的内存消耗要超过 task 的内存界限时，会发生内存溢出错误。图 1 展示了两个分别发生在 map task 和 reduce task 中发生的内存溢出错误。

在错误发生时，task 会抛出内存溢出错误栈（如表 1），用户往往只能从错误栈中观察到正在运行的函数或方法，很难直接确定内存溢出错误原因。

表 1 一个 Hadoop reduce task 抛出的内存溢出错误栈

```

FATAL org.apache.hadoop.mapred.Child:
Error running child : java.lang.OutOfMemoryError: Java heap space
at java.util.Arrays.copyOf((Arrays.java:2882)
at java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:100)
at it.unimi.dsi.fastutil.objects.AbstractObject2IntMap.toString(AbstractObject2IntMap.java:298)
...
at org.apache.hadoop.mapred.ReduceTask$NewTrackingRecordWriter.write(ReduceTask.java:665)
at cloud9.ComputeCooccurrenceMatrixStripes$MyReducer.reduce(ComputeCooccurrenceMatrixStripes.java:136)
at org.apache.hadoop.mapreduce.Reducer.run(Reducer.java:216)
at org.apache.hadoop.mapred.Child.main(Child.java:404)

```

1.2 论文工作

1.2.1 研究动机

分布式并行处理框架为用户提供了简单的编程模型和编程接口，用户不需要具备分布式系统知识也可以开发出可扩展的数据并行应用。这种设计方式使得用户可以只关注于数据处理逻辑本身，无需考虑应用是如何分布执行及中间处理过程。但对用户来说，如果应用出现运行时错误，那么难以诊断出错误原因，更难以修复错误。由于应用会在内存中处理大数据，内存溢出错误成为一种常见而且严重的运行时错误。内存溢出错误可以导致应用失败，而且目前还不能被框架的错误容忍机制修复。除了内存溢出问题，用户在正常使用框架时也面临着不了解应用内存消耗，不知道该为应用分配多少内存的问题。我们在公开论坛，如 StackOverflow.com 上发现很多这两方面的内存问题，具体的问题如下：

问题 1：内存溢出错误常见，但用户不知道错误原因及修复方法。

1. “我很诧异为什么我的 job 会在 map 阶段出现内存溢出错误，内存溢出的原因是什么？[OOMP1]”
2. “如何处理并修复内存溢出错误，我调了一些配置参数（如 reducer 数目），但不起作用？[OOMP2]”

问题 2：用户不明白应用的内存用量，也不知该为应用分配多少内存。

1. “为什么我的 map task 会消耗如此多的内存 [MemUsageP1]？”
2. “我能把 task 的内存设为小于 2GB 吗 [MemUsageP2]？”
3. “我想准确预测出 job 的内存用量，因为内存设的不合理，我的 job 在 40 个节点的集群上跑了 5 天后失败了 [MemUsageP3]？”

问题 3：用户对框架运行机制不理解，手工诊断内存溢出错误困难。

1. “我的 job 总是内存溢出错误，这是 Hadoop 自己的 bug 吗？我不懂处理流程中哪些步骤会消耗过多内存，希望能够得到建议 [OOMDiagnosisP1]”。
2. “我不是很理解 MapReduce 数据流的工作原理。我的一个很紧要的 job 在 reduce 阶段出现了内存溢出错误，希望大家帮我看一下为什么会出现这种错误 [MemUsageP3]”。

另外，我们也从公开论坛（StackOverflow.com, Hadoop/Spark mailing list, 开发者博客等）收集到 276 个实际 Hadoop/Spark 应用的内存溢出错误。我们发现各类应用（不管是用户手写代码产生的应用还是通过高层语言撰写产生的应用）均存在内存溢出错误，而且错误可以发生各个运行阶段（如 map, shuffle, reduce）。

1.2.2 研究问题

针对分布式数据并行应用实际存在的内存问题，我们首先分析问题出现的原因，进而针对性地确定研究问题。

第一个问题（内存溢出错误常见，但用户不知道错误原因及修复方法）出现的原因是影响应用内存用量的因素太多，有静态配置，用户代码，也有动态数据流，用户很难弄清楚它们是怎么导致内存溢出错误的，更不知道如何调整这些因素降低内存用量。所以，我们的第一个研究问题是：

研究问题 1：能否总结出内存溢出错误的常见错误原因及修复方法？

第二个问题（用户不明白应用的内存用量，也不知道该为应用分配多少内存）出现的原因是应用静态因素（如数据，配置，用户代码）不直接影响其动态内存用量，用户难以建立两者之间的关系，更不知道如何对新应用进行内存用量估算。所以第二个研究问题是：

研究问题 2：能否建立应用的内存用量模型，进而估算新应用的内存用量？

第三个问题（用户对框架运行不理解，手工诊断内存溢出错误困难）出现的原因是内存溢出错误诊断需要用户对框架、应用内存使用及应用分布式调试等有丰富经验，而且即使是有经验的框架开发者也很难把调试经验总结成工具或方法。因此，第三个研究问题是：

研究问题 3：能否设计一套应用内存用量问题的诊断方法和工具？

1.2.3 研究思路

这三个研究问题是相互关联的，如图 2 所示，我们的研究思路是：首先通过分析现实世界的应用的内存溢出错误来总结常见的错误原因及修复方法，得出影响内存用量的关键影响因素。之后，分析各个内存影响因素之间的关系，构建应用内存用量模型，建立静态因素（数据、配置、用户代码）与动态内存用量的关系，然后利用内存模型对新应用的内存用量进行估算。最后，基于总结出的内存溢出错误原因和构建出的内存用量模型，设计内存溢出错误诊断方法及工具。

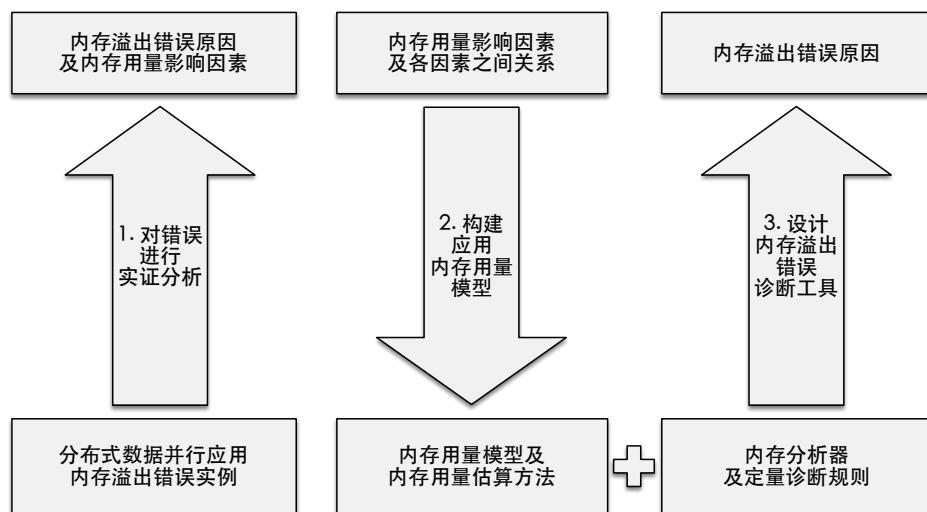


图 2 论文整体研究思路

1.2.4 研究内容

基于上述研究问题和思路，本文从以下三方面开展研究：

(1) 内存溢出错误的实证分析：具体研究内容是分析真实分布式数据并行应用内存溢出错误的常见原因，常见修复方法，并探索可以提高框架错误容忍能力的方案。本文从公开论坛等渠道实际收集和研究了 123 个真实 Hadoop 和 Spark 应用的内存溢出错误。我们发现应用内存溢出错误的三大原因是：框架暂存的数据量过大，数据流异常及内存使用密集的用户代码。我们也从 42 个包含修复信息的错误中总结出了常用的修复方法，包括如何降低框架暂存的数据量，如何减小运行时的中间数据量，及如何降低用户代码的内存消耗的方法。最后，我们也提出了可以提升框架错误容忍能力和错误诊断能力的三种方法。

(2) 内存用量模型构建与用量估算：具体研究内容是构建应用的内存用量模型，解释并量化应用的静态因素（数据、配置、用户代码）与动态内存用量的关系，并设计

算法来估算新应用的内存用量。本文以数据流为中心构建了分布式数据并行应用的内存用量模型。该模型包含三个子模型：数据流模型、框架内存模型和用户代码模型。数据流模型可以量化“静态配置—数据流”的关系，框架内存模型可以量化“数据流—框架内存用量”之间的关系，用户代码模型可以量化“数据流—用户代码内存用量”之间的关系。为了解决如何在代码未知的情况下量化用户代码内存用量与其输入数据的关系，我们在用户代码模型中设计了生命周期敏感的内存监控方法。在内存用量模型构建好后，我们设计了如何通过在小数据上运行应用来估算该应用在大数据集上的内存用量的参数估计方法。我们在五个有代表性的 Hadoop MapReduce 应用做了内存用量估算实验，实验结果表明我们的估算方法的相对误差率在 20% 以内。

(3) 内存溢出错误的诊断方法及工具：具体研究内容是设计出内存溢出错误诊断方法及工具。当应用出现内存溢出错误时，可以定位出错误相关的代码，错误相关的数据，以及不恰当的配置参数。本文基于内存用量模型设计了一个内存分析器 Mprof。Mprof 可以自动建立应用静态因素与动态内存用量的关系，方法是重建应用数据流，重建用户代码内存使用历史信息，并对两者进行关联分析。Mprof 也包含定量诊断规则，这些规则根据应用静态因素与动态内存的关联关系来定位内存溢出错误相关的代码，错误相关的数据，以及不恰当的配置参数。Mprof 仅仅依赖于我们提升过的 task 的运行日志、动态数据流监控器以及 heap dumps，不需要用户对代码做任何改动。我们在 20 个真实的 Hadoop MapReduce 内存溢出错误上对 Mprof 做了实验评估，发现 Mprof 可以正确诊断出 15 个内存溢出错误，部分正确诊断出 5 个内存溢出错误（因为这 5 个错误的代码由高层语言撰写，Mprof 还不能诊断出这 5 个错误的相关代码段）。

1.2.5 创新点

本文针对分布式数据并行应用的内存用量及内存溢出错误开展研究，研究问题可以归类为应用的性能与可靠性问题。目前分布式数据并行应用内存方面的研究工作还比较少，与已有工作相比，我们的创新点主要体现在以下几个方面：

(1) 对应用内存溢出错误进行了实证分析：之前对于大数据应用错误的分析主要是针对通用性的错误（只要是可以导致 job 失败的错误）。比如，Li 等人 [Li2013] 研究了 250 个运行在微软 Dryad 框架之上 SCOPE job [SCOPE] 的故障错误，发现错误主要是未定义的列，错误的数据模式，不正确的行格式等等。Kavulya 等人 [Kavulya2010] 分析了 4100 个执行失败的 Hadoop jobs，这些 jobs 运行在 Yahoo! 管理的 M45 集群。他们发现 36% 的错误是数组访问越界错误，还有 23% 的错误是 IO 异常。我们发现内存溢出错误很常见而且不能被框架的错误容忍机制处理，有必要对其单独展开研究。我们分析了常见的错误原因，并解释了错误是怎么发生的。对于大多数错误原因，

我们也总结了相应的错误修复方法，这些方法可以帮助用户实际修复内存溢出错误。我们也提供了潜在的可以提升框架错误容忍能力的方法。

(2) 构建了应用内存用量模型并设计了内存用量估算方法：之前研究人员关注的应用性能问题主要是 MapReduce 执行时间估算与最优资源分配(如何使 job 执行更快)。比如，Morton 等人 [Morton2010] 借鉴了数据库中的 SQL 查询进度估计器提出了一个在线启发式方法来预测 MapReduce jobs 的 pipeline 执行时间。Verma 等人 [ARIA2011] 提出了一个具有理论界限的时间模型来分析估算 MapReduce 的执行时间。他们 [Verma2011] 也讨论了如何为 MapReduce job 分配最佳的资源 (map/reduce slot) 来保证 job 能够在给定时间界限内完成。StarFish [Herodotou2011] 提供了一个基于代价模型的优化器来寻找使得 job 执行时间最短的配置参数。StarFish 包含一个 what-if 引擎来预测 job 在不同配置参数下的性能 (主要是执行时间)。与这些工作不同的是，我们主要关注内存资源消耗，我们提出了一个以数据流为中心的应用内存模型，模型刻画了应用静态因素 (数据、配置、用户代码) 与 task 的动态内存用量之间的关系。我们还提出了参数估计算法，该算法可以通过获取应用在小数据集上的运行信息来预测应用在大数据上运行时的内存消耗。

(3) 设计了应用内存溢出错误诊断工具：目前诊断应用内存溢出错误完全靠用户经验，框架只提供了一些关于内存使用的指导和调优建议 [HadoopTuning, Tips4Hadoop, SparkTuning]，还没有自动化的诊断工具。也没有研究工作关注分布式框架的内存溢出错误诊断与修复。当前的内存分析工具如 Eclipse MAT [MAT]，和内存泄漏检测方法如 [Cherem2007, Xie2005, Jump2007, Xu2008]，不能直接用于内存溢出错误诊断。因为这些工具只能分析出内存中有哪些对象或者哪些对象本应该释放却没有释放，不能分析出这些对象的来源及对象占用内存空间大的原因。我们基于内存用量模型设计了一个内存分析器 Mprof。Mprof 可以自动建立应用静态因素与动态内存用量的关系。Mprof 也包含定量诊断规则，这些规则根据应用静态因素与动态内存的关联关系来定位内存溢出错误相关的代码，错误相关的数据，以及不恰当的配置参数。

本文的研究成果对于分布式数据并行框架用户和框架设计者均有实际参考意义，也对后续研究工作（比如研究其它框架或应用的内存溢出问题）有借鉴意义。

1.3 论文组织

论文总共分为六章，其中，后续章节的组织结构和内容如下：

第二章介绍论文所需的背景知识，包括分布式数据并行应用基本特征，以及应用开发和执行的基本过程。

第三章以实证分析的角度来研究内存溢出错误的原因、修复方法，并探索提升框架

内存溢出错误容忍能力的方法。

第四章讨论如何构建应用内存用量模型，也就是如何建立应用静态因素与 task 动态内存用量之间的关系。在此基础上，讨论如何在给定新应用后，预测应用的内存用量。

第五章介绍如何设计针对 MapReduce 应用的内存溢出错误诊断方法及工具。

第六章介绍了总结论文工作，并对进一步的研究工作进行展望。

第二章 分布式数据并行应用及相关研究工作介绍

本章介绍论文所需的背景知识及相关的研究工作。背景知识包括分布式数据并行应用的基本特征、应用运行过程及内存使用情况。相关研究主要介绍在框架和应用功能、性能和可靠性方面的工作。

2.1 概述

一个分布式数据并行应用可以表示成 <输入数据, 配置, 用户代码>。应用的输入数据一般以分块(比如以 64MB 为一块)形式存储于分布式文件系统(比如 Hadoop)之上。用户在向分布式数据并行框架提交应用之前, 需要准备输入数据(指定数据存储位置), 撰写用户代码(也就是数据处理逻辑), 并设定配置参数。之后, 用户可以将应用提交给分布式数据并行框架运行。应用的整个开发和执行过程分为三个层次: 用户层负责应用的开发, 分布式数据流层负责应用的分布执行, 物理内存层反映应用实际执行的内存消耗。

当前的研究工作包括分布式数据并行框架设计、应用高层编程语言及高层库的设计、应用性能优化、应用可靠性保证等。

以下小节会具体介绍框架和应用的基本特性及相关的研究工作。

2.2 分布式数据并行框架架构

分布式数据并行框架一般是主从(Master-Slave)结构, 主节点(master 节点)负责接收用户提交的应用(job), 管理 job 运行的整个生命周期。从节点(slave 节点)负责执行具体的数据处理任务(task), 并在运行过程中向主节点汇报任务的执行状态。以 Hadoop 为例(图 3), 在主节点运行的 JobTracker 进程首先接收用户提交的 job, 然后根据 job 的配置、输入数据等信息将 job 分解为具体的数据处理任务(map/reduce tasks), 最后将 tasks 交给任务调度器调度运行。任务调度器根据各个从节点的资源总量与资源使用情况将 map/reduce tasks 分发到合适的从节点的 TaskTracker。TaskTracker 进程会为每个 map/reduce task 启动一个进程(在 Hadoop 中是 JVM 进程)来运行 task 的各个处理步骤。每个从节点可以同时运行的 tasks 数目由集群管理员根据该节点的 CPU 核数等资源状况配置。

集群同时可以运行多个 job, 也就是多个 map/reduce tasks。比如在图 3 中, Slave 1 节点上运行了 3 个 map task 及 1 个 reduce task, 正在运行的 map task 和 Slave 2 节点上

正在运行的 map task 可以分属于不同的 job。在 Spark 中，用户还要撰写 job 运行管理程序（driver program），该程序可以向 tasks 广播数据，也可以收集 tasks 的执行结果。

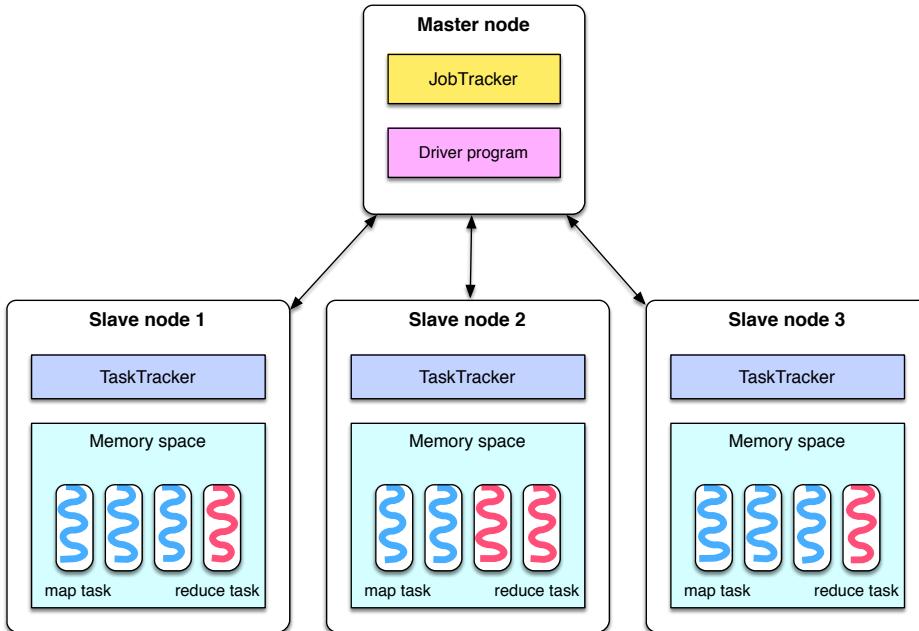


图 3 分布式数据并行框架部署图

当前，分布式数据并行框架资源管理方面的研究工作主要强调将资源管理模块与框架运行模块分离，将资源管理和任务调度模块构造为一个统一的集群资源管理系统，称为“集群操作系统” [Zaharia2011]。该系统可以集中调度多个不同框架中的 jobs。如，第二代 Hadoop 的资源管理与调度框架 YARN [YARN] 能够同时为集群中运行的多种框架（如 Hadoop MapReduce，Spark）的 jobs 分配资源。用户在提交 job 到 YARN 上前需要预先指定 job 的资源需求，如 CPU 核数及内存空间大小。UC Berkeley 提出的 Mesos [Mesos] 与 YARN 类似，负责管理集群上各种 job 的资源分配与调度，如 MapReduce job、Spark job、MPI [MPI] job 等。尽管 YARN 和 Mesos 提供了相当成熟的资源管理策略，可以统一分配、管理、回收不同节点上的计算资源。然而，它们的资源分配策略依赖用户提供的资源需求与当前集群资源监控信息，对应用语义并不敏感。

2.3 应用编程模型

分布式数据并行框架为用户提供了简单且具有扩展性的编程模型。没有任何并行或分布式应用开发经验的用户也可以通过简单的编程模型来开发数据密集型应用。目前通用的分布式数据并行框架，比如 Hadoop，Dryad 和 Spark，都是以 MapReduce 编程模型为基础，MapReduce 编程模型可以简单地表示成：

Map 阶段: $map(k_1, v_1) \Rightarrow list(k_2, v_2)$

Reduce 阶段: $reduce(k_2, list(v_2)) \Rightarrow list(k_3, v_3)$

在 *map* 阶段, $map(k, v)$ 从输入数据的一个分块 (input split) 中按顺序一个个读入 $\langle k_1, v_1 \rangle$ record, 处理该 record, 并输出新的 $\langle k_2, v_2 \rangle$ record。在 *reduce* 阶段, 框架会将 *map* 阶段输出的 $\langle k_2, v_2 \rangle$ records 按照 key 聚合成 $\langle k_2, list(v_2) \rangle$ groups, 然后调用 $reduce(k, list(v))$ 来处理每一个 $\langle k_2, list(v_2) \rangle$ group, 最后输出 $\langle k_3, v_3 \rangle$ records。Hadoop 原生支持这个编程模型, 而 Dryad 和 Spark 提供了更一般的, 对用户友好的操作符 (operator) 来替代原生的 *map()* 和 *reduce()*, 比如 *map()*, *flatMap()*, *groupByKey()*, *reduceByKey()*, *coGroup()*, *join()* 等等。这些操作符 (operator) 构建在原生的 *map()* 和 *reduce()* 函数之上。

另外的关于应用编程模型的研究工作 Map-Reduce-Merge [Yang2007] 在 MapReduce 的编程范型基础上增加了 merge 阶段。

Map 阶段: $map(k_1, v_1) \Rightarrow list(k_2, v_2)$

Reduce 阶段: $reduce(k_2, list(v_2)) \Rightarrow (k_2, list(v_3))$

Merge 阶段: $merge((k_2, list(v_3)), (k_2, list(v_3))) \Rightarrow (k_4, v_5)$

Merge 阶段可以将两个 reducer 产生的结果进行归并, 这样使得框架更容易实现需要多个 MapReduce job 才能完成的操作, 如两个表的 *join()*。

2.4 应用开发的三层结构

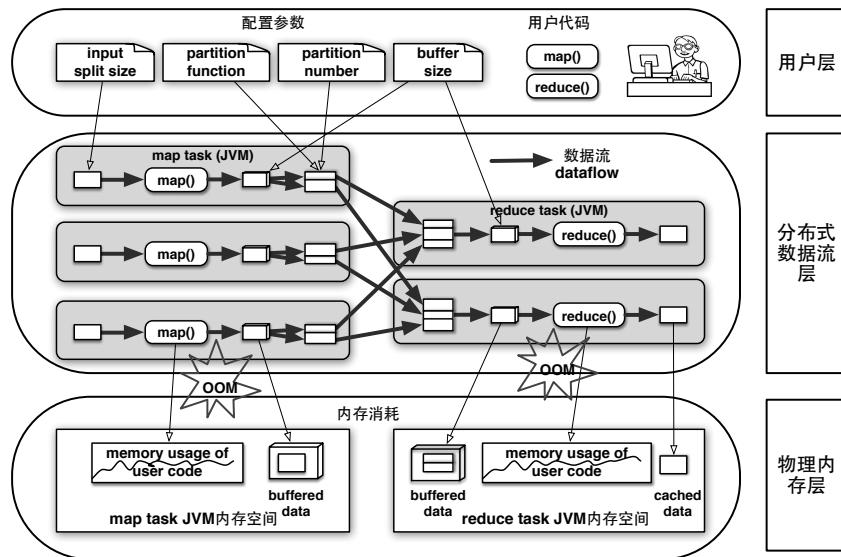


图 4 应用执行的三层结构图

如图 4 所示，分布式数据并行应用的整个开发和执行过程，可以从上到下分为三个层次：用户层负责开发应用，分布式数据流层负责应用的分布执行，物理内存层反映应用实际运行时的内存消耗。

2.4.1 用户层

用户（也就是应用开发者）在开发应用时需要准备输入数据，配置各项参数，撰写用户代码。

A. 输入数据

由于目前分布式处理框架主要应用于大数据的批处理，因此用户在提交 job 之前，需要提前准备好输入数据。输入数据可以预先存放在分布式文件系统（如 Hadoop 的分布式文件系统 HDFS）上，分布式 key/value 数据库（如 HBase [HBase]）上，也可以存放到关系数据库中。输入数据在 job 提交时会由框架进行自动分片（split），每个分片一般对应一个具体 task。

数据存储部分的研究工作主要集中在如何降低磁盘 I/O 来提升 job 的执行性能。PACMan [PACM] 会根据一定策略提前将 task 所需的部分数据 cache 到内存以提高 task 的执行性能。Tachyon [Li2014] 构造了一个基于内存的分布式数据存储系统，该系统主要用于存放不同 jobs 的输入、输出数据，这样可以使得不同类型的 jobs 可以在内存中直接共享数据。

B. 用户代码

用户代码可以是用户手写的 MapReduce 代码，比如用户自定义的 map()，reduce()，见图 5。用户也可以使用 Spark 提供的通用数据处理操作符 flatMap()，join() 等来撰写 Spark job 的代码，见图 6。用户代码也可以由高层语言或者高层库产生。比如在图 7 中，SQL-like 的 Pig 脚本可以自动生成二进制的 map() 和 reduce()，但其源代码不可见。一些高层库提供了更简单方式来产生应用，如 Spark 之上的机器学习库 MLlib [MLlib] 在用户选择算法及算法参数后，可以自动生成可执行的 Spark jobs。除了 map() 和 reduce() 函数，为了优化应用性能，用户也可以定义一个“迷你”的 reduce()，叫做 combine()。Combine() 可以在 reduce() 执行之前对中间数据进行聚合，这样可以减少 reduce() 的输入数据量，因此 combine() 和 reduce() 一般具有相同的代码，可以将 combine() 视作小型 reduce()。

除了 map() 和 reduce() 以外，用户还需要撰写一个 driver 程序用来提交应用。Driver 程序一般运行在客户端。在 Hadoop 中，driver 负责设定输入输出数据类型，并向 Hadoop 框架提交 job。在 Spark 中，如图 8 所示，driver 程序不仅可以产生数据、广播数据给各

个 task，也可以收集 task 的运行结果，最后在 driver 内存中进行计算最终结果。

```

1. public class Mapper {
2.   StanfordLemmatizer slem = new StanfordLemmatizer();
3.   public void map(Long key, Text value) {
4.     String line = value.toString();
5.     for(String word: slem.lemmatize(line))
6.       emit(word, 1);
7.   }
8. }
```

```

1. public class Reducer
2.   public void reduce(Text key, Iterable<OHMap> values) {
3.     Iterator<OHMap> iter = values.iterator();
4.     OHMap wordMap = new OHMap();
5.     while (iter.hasNext()) {
6.       wordMap.plus(iter.next()); => wordMap (287MB)
7.     }
8.     emit(key, wordMap);
9.   }
10. }
```

图 5 用户手写的 Hadoop MapReduce 代码

```

1. val textFile = spark.textFile("hdfs://...")
2. val counts = textFile.flatMap(line => line.split(" "))
3.           .map(word => (word, 1))
4.           .reduceByKey(_ + _)
5. counts.saveAsTextFile("hdfs://...")
```

图 6 用户手写的 Spark 代码

```

1. pTable = LOAD "tableA" as (pagerank,pageurl,aveduration);
2. rankTable = GROUP pTable BY pagerank;
3. urlTable = FOREACH rankTable {
4.   urls = DISTINCT urlTable.pageurl;
5.   GENERATE group, COUNT(urls), SUM(pTable.aveduration);
6. };
7. STORE urlTable into "/output/newTable";
```

图 7 由高层语言 Pig Latin 撰写的代码

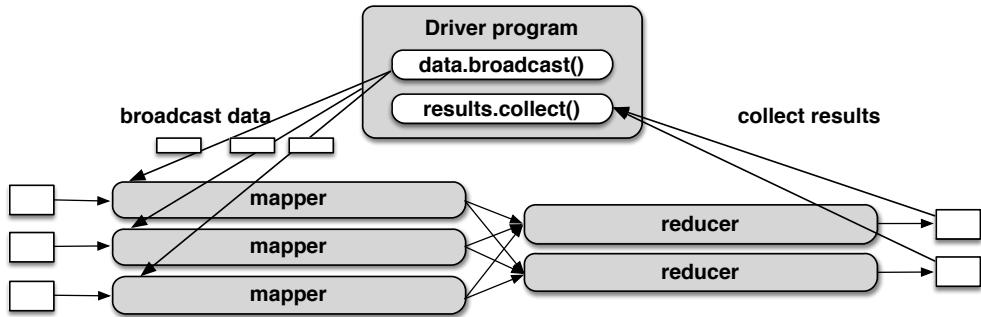


图 8 Driver 程序

在用户代码方面的研究工作主要集中于如何设计更简单、更通用的高层语言或高层库，使得用户不用去手写较为繁琐的 `map()`, `reduce()` 代码。Google 提出了 FlumeJava [Chambers2010]，将多个 MapReduce job 以 pipeline 的形式串联起来，并提供了一些基本的数据操作符，如 `group()`, `join()`，使得一些常见的编程任务变得简单。Cascading [Cascading] 是一个用 Java 编写的，构建在 Hadoop 之上的一套数据操作函数库。与 FlumeJava 类似，Cascading 同样为用户提供了一些基本数据操作符，可以方便用户构建出较为复杂的数据流程。Google 也设计了 Sawzall [Pike2005]，Sawzall 是一种数据查询的脚本语言，偏向统计分析，Sawzall 脚本可以转化为 MapReduce job 执行，使得分析人员不用直接写 MapReduce 程序就可以进行大数据分析。Google 还设计了 Tenzing [Chattpadhyay2011]，该模块构建在 MapReduce 框架之上，支持 SQL 查询语言，并提供低延迟的高效数据查询服务。微软研究院也设计了自己的用户层语言 DryadLINQ [DryadLINQ] 和 SCOPE [SCOPE]。DryadLINQ 将针对数据项操作的 LINQ 语言转化成 Dryad 任务，利用 Dryad 处理框架来并行分布处理数据。SCOPE 与 Sawzall 在一个层次，将 SQL 逻辑转化成 Dryad DAG 任务，也是利用 Dryad 处理框架来并行分布处理数据。SCOPE 和 Dryad 是 C#/C++ 实现的。

在用户代码的优化方面：PeriSCOPE [PeriSCOPE] 可以自动优化运行在 SCOPE 上的 job 性能，该系统可以根据 job pipeline 的拓扑结构来对用户代码采用类似编译优化的优化措施。

C. 配置

一个分布式数据并行应用的配置参数虽然有很多（比如 Hadoop 包含 200 多个），但主要有两类：(1) 内存相关的配置参数，比如 buffer size，直接影响 map/reduce task 的内存用量。Buffer size 负责定义框架缓冲区的大小。在 Hadoop 中，map/reduce task 实际启动一个 JVM 来运行，因此用户还要设置 JVM 的大小，也就是 heap size。(2) 数据流相关的配置参数，比如 partition number/function，通过影响数据流来间接影响

map/reduce task 的内存用量。比如 partition function 定义如何划分 map() 的输出数据, partition number 定义会有多少个数据分块产生, 也就是有多少个 reducer 会被运行。Input split size 定义 mapper 的输入数据 (input split) 大小。

由于 Hadoop 本身没有提供自动优化配置的功能, 当前关于配置的研究工作主要关注如何寻找最优参数来对 job 进行性能调优。StarFish [Herodotou2011] 通过对 job 运行的历史信息进行分析, 并结合参数空间搜索的方法来得到 MapReduce job 最优的参数配置。StarFish 的核心是一个 Just-In-Time 的优化器, 该优化器可以对 Hadoop job 进行分析, 并根据分析结果来预测 job 在不同配置参数下的性能 (主要是执行时间), 调整框架中的各种参数和资源分配。Verma 等人 [Verma2011M] 讨论了如何为 MapReduce job 分配最佳的资源 (map/reduce slot) 来保证 MapReduce job 能够按时完成。

2.4.2 分布式数据流层

一个分布式数据并行应用包含一个或多个 MapReduce jobs。如图 4 所示, 一个 job 包含 map 和 reduce 两个阶段。Dryad 和 Spark 特殊的地方是一个 job 可以包含多个 map 和 reduce 阶段, 这些阶段可以构成一个有向无环图 (DAG 图, 如图 10 所示)。Map 阶段包含多个可以并行执行的 map tasks (mappers), reduce 阶段包含多个可以并行执行的 reduce tasks (reducers)。Map task 负责将输入的分块数据进行 map() 处理, 将其输出结果写入缓冲区, 然后对缓冲区中的数据进行分区 (partition)、排序、combine、归并等操作, 最后将数据输出到磁盘上的不同分区 (partition) 中。Reduce task 负责将 map task 输出的对应分区数据通过 HTTP 拷贝到本地内存中, 内存空间不够存放时, 会将内存数据块归并到磁盘, 然后经过磁盘排序、归并等阶段产生 reduce() 的输入数据。Reduce() 处理完输入数据后, 将输出数据写入分布式文件系统。数据流指的是流动在 map tasks 中, reduce tasks 中, 以及它们之间的数据。如图 4 中的方块和箭头所示, 数据流不仅包含用户代码的输入和输出数据, 也包含由框架负责管理的中间数据。

Dryad 和 Spark 对输入、输出、中间数据进行了抽象, 将这些数据表示成一个统一的数据结构。比如在 Spark 中, 输入、输出、中间数据被表示成 RDD (分布式弹性数据集)。数据处理操作符, 如 map(), cogroup(), join() 都在 RDD 上执行。RDD 内部将数据划分为多个分区 (partition, 如图 9 所示), 每个 partition 一般有一个 task 进行处理 (如图 10 所示)。RDD 可以存在于内存或磁盘上。Spark job 的数据流指的就是在 RDD 以及 RDDs 之间流动的数据。Spark 中的 task 也分为两种: map task 负责将 RDD 对每个数据分区 (partition) 执行数据操作, reduce task 负责将多个 RDD 中的数据进行聚合, 执行数据操作并输出结果。

MapReduce 与 Dryad/Spark 最主要的区别是后者支持数据流水线 (pipeline)。如图

10 所示，在数据流水线中，map/reduce tasks 可以连续执行多个用户代码，比如连续运行两个 map()，而不需要将第一个 map() 的运行结果先存放到磁盘上。另外，在 Spark 中，用户可以显式地告诉框架（利用 cache() 接口）去缓存一些可以被重用的中间数据，比如 reduce() 的输出可能会被下一个 job 使用到，那么可以使用 cache() 去缓存 reduce() 的输出结果。

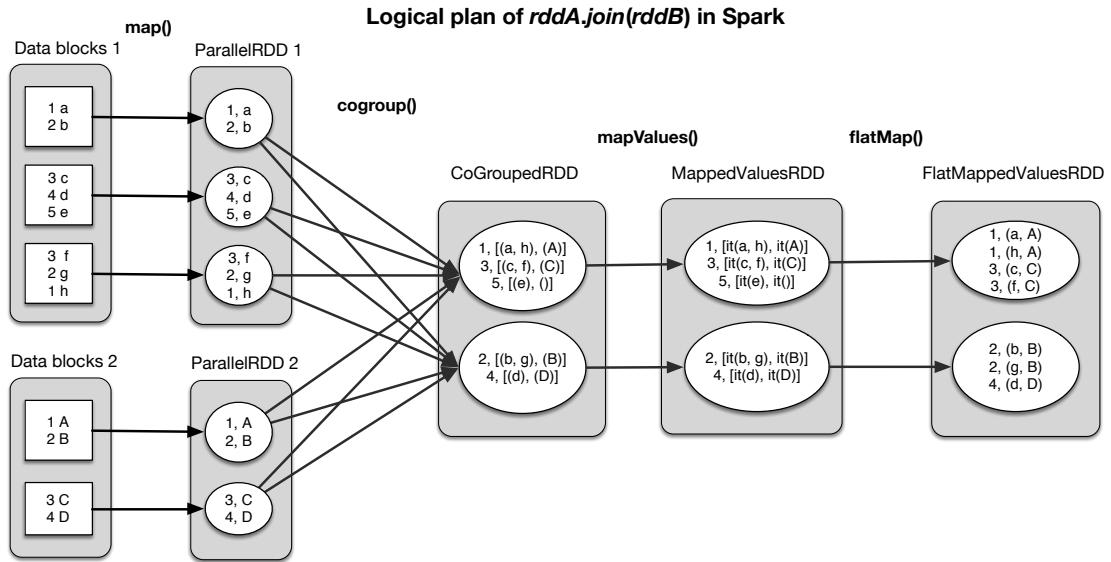


图 9 Spark job 的逻辑执行图

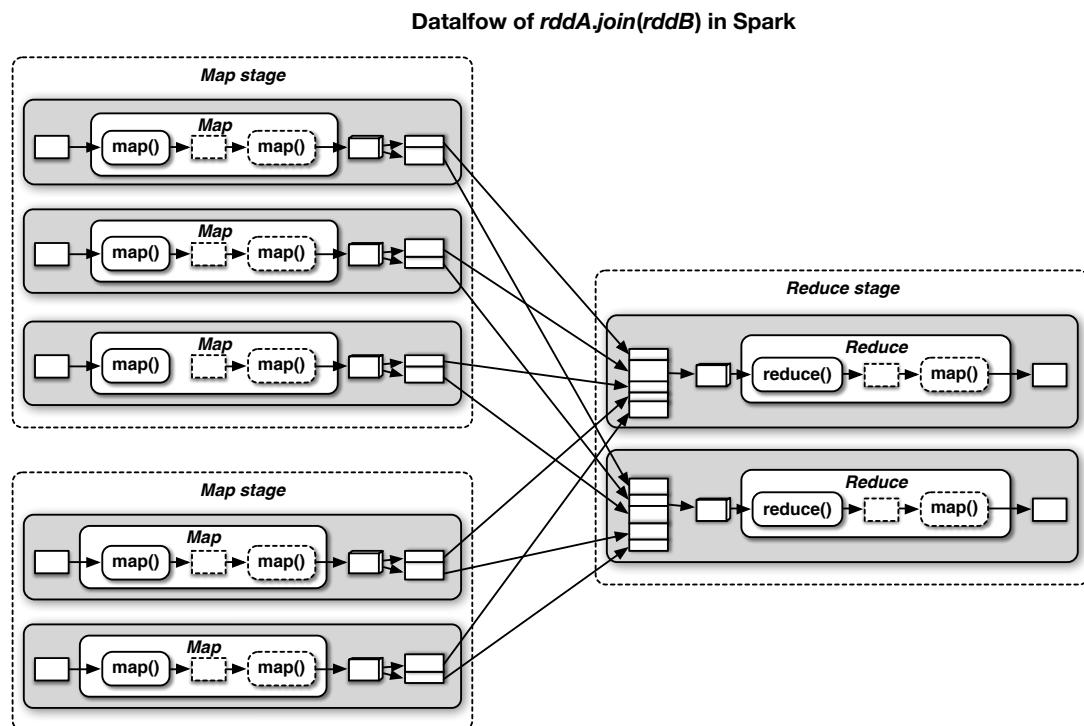


图 10 Spark 数据流

除了上面介绍的分布式处理框架 MapReduce、Dryad 与 Spark 以外，Yahoo 改进并提出了 Map-reduce-merge [Yang2007] 框架，通过在 reduce 阶段后面加入 Merge 阶段，提高了 MapReduce 对二维表的关系代数处理能力。UC Berkeley 提出了 MapReduce Online [Condie2010]，改进了 map 阶段到 reduce 阶段的数据流动方式，更加流水线化，提高 MapReduce 对数据的在线处理能力。UCI 的 Bu 等人提出了 HaLoop [Bu2010, Bu2012]，提高了 MapReduce 迭代型任务的执行性能。NYU 提出了 Piccolo [Power2010]，可以使用分区表创建快速，并行内存计算的分布式程序。另外，Google 针对交互式数据查询设计了 Dremel [Melnik2010]，该系统能够对只读型的嵌套型海量数据进行快速分析，使用了多版本的执行树及列数据模型等技术。

2.4.3 物理内存层

应用的 map/reduce task 在实际执行时会启动一个进程或线程来执行具体数据处理任务。在 Hadoop 中，每个 map/reduce task 实际启动一个 JVM 来完成数据处理任务。在 Spark 中，每个 map/reduce task 的数据处理任务由 JVM 中的一个线程完成。应用在未运行前，我们无法预知 task 的内存消耗，也就是无法预知 JVM 中的 heap usage。

但从应用特点来分析，我们可以知道 task 的内存消耗主要包括三个部分：(1) 框架缓冲区的中间数据 (buffered data)。比如，map()输出到缓冲区的数据和 reducer shuffle 到缓冲区的数据。(2) 框架缓存数据 (cached data)。比如，在 Spark 中，用户故意 cache 到内存的要被重用的数据。(3) 用户代码产生的中间计算结果。用户代码 map(), reduce(), combine() 在处理输入数据时会在内存中产生中间计算结果。

框架本身在设计时就考虑了内存的使用问题：Spark 框架是基于内存计算，将大量的输入或者中间数据缓存到内存，提高交互型和迭代型 job 的执行效率。UCSD 提出了 ThemisMR [ThemisMR2012]，重新设计了 MapReduce 的数据流过程，设计了新的内存管理方案，有效地将中间数据磁盘读写次数降低为两次，提高了 job 的执行性能。

由于应用在处理大数据时会消耗大量的内存，一些研究者提出了一些减少应用内存消耗的办法。Tachyon [Li2014]提供了一个基于内存的分布式数据存储系统，主要用于存放不同应用 (jobs) 产生的重用数据。用户可以将原本直接缓存到框架的数据转移存放到 Tachyon 上，以降低框架的内存消耗。FAÇADE [Nguyen2015] 提供了用于降低用户代码内存消耗的用户代码编译和执行环境。FAÇADE 设计目的是将数据存储和数据操作分开，方法是将数据存放到 JVM 的堆外内存，将对堆内对象的数据操作转换为对 FAÇADE 的函数调用。对于 Java 对象本身产生的 overhead (也就是 Java 对象自身所需的 header 和 reference)，Bu 等人 [Bu2013] 提出了两种减少 overhead 的方法，比如将大量数据对象 (record object) 合并少量的大对象。然而，他们没有研究内存溢出错误的诊断问题。Interruptile Tasks [Fang2015] 改进了现有的 task，使得 task 具备一定的错误

容忍能力。当 task 在运行时遇到内存用量过量或者内存溢出问题时，Interruptile task 会暂停当前 task 的运行，回收 task 中部分运行数据及中间结果，并将不能回收的结果 spill 到磁盘，然后执行用户定义的 interrupt 逻辑，等到内存用量下降到一定程度后，再让 task 继续运行。

2.5 应用性能与内存使用问题

虽然 MapReduce 等数据并行处理框架具有编程范型简单、任务自动调度和集群自动管理的优点，但是由于应用开发和执行的三层结构，用户代码可以被任意撰写、参数配置繁多、要处理的数据量大等因素，应用很容易产生性能和运行时错误。具体如下：

(1) 用户只负责编写用户代码，实现数据处理逻辑，对 job 的执行性能，包括执行时间、数据流、资源消耗等，并不了解。用户更不知道如何去分析、预测 job 的资源消耗，尤其是内存资源消耗。很多研究人员关注 Job 执行时间的预测，job 执行时间除了能够方面用户了解 job 执行进度和决策 job 启停时间以外，也为任务调度器提供了决策依据。华盛顿大学研究人员提出了 KAMD [Morton2010]，ParaTimer [ParaTimer]，根据 job 执行历史信息并结合正在运行的 job 处理的数据量，使用启发式方法来估算 pipeline 和 DAG 型的 job 剩余执行时间。UIUC 的研究人员提出了 ARIA [Verma2011]，细粒度地分析了单个 MapReduce job 的执行阶段，提出了基于上下界的时间估算公式，通过 job 历史信息或调试信息来估算执行时间。华盛顿大学的研究人员又提出了 PerfXplain [Khoussainova2012]，通过对比两个包含同样处理逻辑的 job 的性能指标，解释执行效率不同的原因。

(2) 由于框架和用户代码要在内存中处理大数据，内存消耗又受到配置参数、用户代码等多种因素影响，job 很容易出现内存用量过高、内存溢出等运行时错误。一旦出现错误后，除了很难定位原因外，出错层次也能很定位，可能是用户层代码的问题，可能是框架层数据流的问题，也可能是配置参数的问题。内存消耗影响因素多且难以预知。

2.6 框架错误容忍机制

由于不能避免系统 bug、节点宕机、网络不通、磁盘损坏等软硬件可靠性问题，分布式系统一般在设计时会有错误容忍机制，实现时也会针对各种失效情况采取相应措施。分布式数据并行框架也不例外，框架设计了各种针对 master 节点失效，task 执行失败等问题错误容忍机制。但对于 task 的执行失败问题，框架的错误容忍机制很简单：只是选择合适节点重新运行该 task。然而，对于内存溢出问题来说，简单地重新运行 task 并不能解决问题，内存溢出错误仍然会出现。一般用户在出错时很难找到真正出错原因，即使是十分熟悉框架运行细节的用户，在缺乏分析诊断工具的情况下，也难以快速找到出错原因。

目前关于内存问题的研究主要集中在内存泄露检测与容忍方面。微软研究院提出了 SWAT [Hauswirth2004] 使用抽样程序执行和适应统计学习方法来检测内存泄露。德克萨斯奥斯丁分校提出了 Cork [Jump2007] 可以动态归结堆内存消耗情况，进而检测内存泄露，并且有较低的时间和空间开销。以色列 Tel-Aviv University 的研究人员针对 GC 不能处理的“对象可以获取到但不会再访问到”的情况，提出了数组内存出现泄露时的自动化去除方法 [Shaham2000]。俄亥俄州立大学的研究人员提出了 LeakSurvivor [Tang2008]，使得 Java 程序在出现内存泄漏时，通过将内存对象放到磁盘上来容忍错误。该校又提出了 LeakChaser [Xu2011]，帮助程序员准确定位容易导致内存泄露的无用的对象引用，并使用高层语义解释无效引用导致内存泄露的原因。

2.7 本章小结

本章介绍了分布式数据并行处理框架的基本架构及应用的基本特征。重点介绍了应用开发和运行过程中的三层结构：用户层如何开发应用、分布式数据流层如何执行数据处理流程，物理内存层怎么反映内存用量。我们还介绍了应用开发和运行方面的相关研究工作，框架的错误容忍机制以及其不足之处。这些背景知识是探讨下面章节中的研究问题与研究方法的基础。

第三章 分布式数据并行应用内存溢出错误实证分析

本章通过实证分析来研究内存溢出错误的原因、修复方法，并探索提升框架内存溢出错误容忍能力的方法。

3.1 概述

在绪论中，我们介绍过在分布式数据并行应用中，内存溢出错误很常见而且当前框架的错误容忍机制不能解决内存溢出错误问题。为了帮助用户理解、分析、修复内存溢出错误，我们从实证分析角度来研究下面三个问题：

RQ1：应用内存溢出的错误原因是什么？有没有一些常见的错误类型（cause patterns）？

RQ2：用户是怎么修复内存溢出错误的，有没有一些常见的修复策略？

RQ3：有没有可以提高框架错误容忍能力或者辅助错误诊断的方法？

3.2 研究方法

3.2.1 研究对象

我们选择运行在 Apache Hadoop 和 Apache Spark 上的分布式数据并行应用作为我们的研究对象。因为目前没有专门针对内存溢出错误的缺陷库（bug repository），用户在遇到内存溢出错误时会选择公开论坛或者 Hadoop/Spark 的邮件列表等公开途径来询问错误原因及修复方法。因此，我们以这些公开应用为研究对象，获得应用的方法是：我们先通过 Google 搜索内存溢出错误的关键字，比如“Hadoop out of memory”，“Hadoop outofmemory”，“Spark OOM” 等。

通过关键词搜索，我们一共发现 1151 个用例（issue），这些用例来自于 StackOverflow.com，Hadoop 邮件列表 [HadoopMailList]，Spark users/dev 邮件列表 [SparkMailList]，开发者的博客，及两本关于 MapReduce 的畅销书籍 [MRDesignPattern, TextMR]。

我们手工检测每个用例，然后筛选出满足下列两个条件的用例：

(1) 是 Hadoop/Spark 的内存溢出错误。

其中有 786 例子不是内存溢出错误，比如只包含部分的关键字“Hadoop Memory”，或只讲 Hadoop Memory 的配置方法。

(2) 内存溢出错误发生在 Hadoop/Spark 应用里。

有些内存溢出错误发生在 Hadoop/Spark 系统本身，比如发生在 Hadoop 调度器或 Hadoop 的资源管理器（TaskTracker）里。

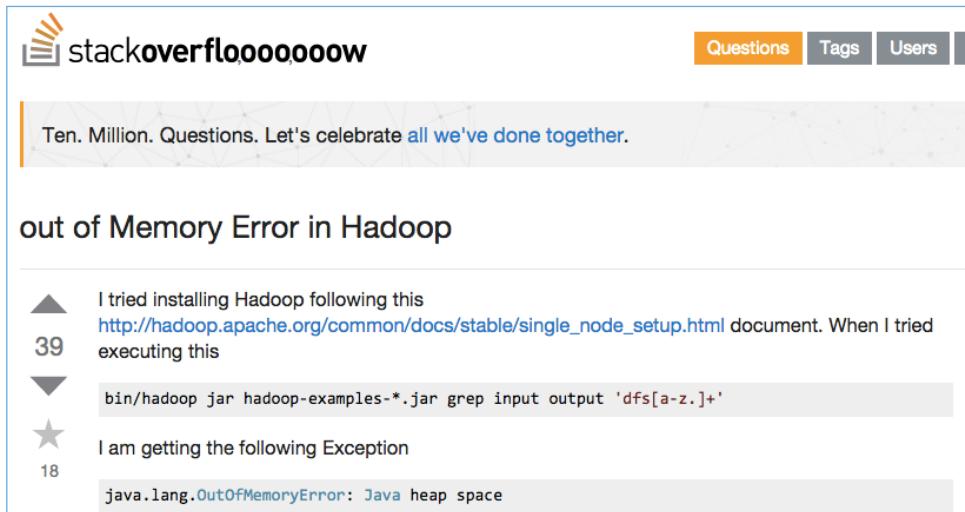


图 11 用户描述的内存溢出错误例子

1 ▲ In your first job you are keeping all the values corresponding to a specific key in a list. As you have 5cr rows and each row have 9 attributes the size of all the values corresponding to a specific key will be too large for a normal List in java to keep in heap memory. That is the reason for `java.lang.OutOfMemoryError: Java heap space` exception. You have to avoid keeping all the values corresponding to a key in an object in java heap. – donut Apr 13 '14 at 14:10 ↗

图 12 专家对内存溢出错误的诊断与回答

经过筛选，我们留下 276 个错误用例 [OOMCases]，我们发现其中 123 个用例的错误原因已经被诊断出来（通过 3.2.2 节中的方法），所以我们的研究对象就是这 123 个内存溢出错误用例。这些用例既有用户手工撰写代码生成的应用，也有依赖高层语言（如 Apache Pig）和高层库（如 MLlib，Cloud9 [Cloud9]）生成的应用，应用的详细分布如下表：

表 2 出现内存溢出错误的应用分布表

框架	应用来源	手写	Pig	Hive	Mahout	Cloud9	GraphX	MLlib	总数	重现数
Hadoop	StackOverflow.com	20	4	2	4	0	0	0	30	16
	Hadoop mailing list	5	5	1	0	1	0	0	12	6
	Developers' blogs	2	1	0	0	0	0	0	3	2
	MapReduce books	8	3	0	0	0	0	0	11	2
Spark	Spark mailing list	16	0	0	0	0	1	2	19	3
	StackOverflow.com	42	0	0	0	0	1	5	48	14
总数		93	13	3	4	1	2	7	123	43

3.2.2 错误原因与修复方法的收集

对于每一个错误用例，我们手工分析用户的错误描述信息以及专家的诊断信息。专家包含 Hadoop/Spark 的框架设计开发者，有经验的应用开发者或者 MapReduce 书籍作者，这些专家有来自 Hadoop/Spark 框架开发者，比如 cloudera.com 和 databricks.com，也有来自框架用户比如 ebay.com, huawei.com。我们发现 276 个错误用例中有 123 个已经被诊断出了错误原因（满足了下面三个条件之一）。

- (1) 专家诊断出了错误原因，用户也接受了专家的回答，一共 66 个。
- (2) 用户自己诊断出了错误原因。用户详细描述了错误原因，包括数据异常信息，不恰当的配置，异常的用户代码逻辑等，一共 45 个。
- (3) 我们自己重现了错误（重现方法见 3.2.3 节），手工分析诊断出了错误原因，一共 12 个。

除了获取错误原因，我们也从 42 个有专家修复建议或用户自己使用的修复方法的用例中获取修复方法。我们首先抽取相应的修复方法，然后将类似的修复方法归并在一起，最终得到 11 个修复方法类别（pattern）。

3.2.3 错误重现方法

为了深入研究内存溢出错误原因，我们重现了 43 个（35%）内存溢出错误用例。这 43 个错误用例中包含了可以重现错误的详细信息（比如数据类型，数据大小，可重现的用户代码和内存溢出错误栈）。因为我们无法得到用户自己使用的数据集，我们使用公开数据（如英文维基百科）或者合成数据（随机产生的文本和一个公开的 benchmark [BrownBench]）。重现实验在 10 个节点上的集群进行，框架版本是 Hadoop-1.2 和 Spark-1.2。每个节点包含 16GB 的内存，map 和 reduce task 的内存大小设为 1GB。我们确认了重现出来的错误栈信息和用户报告的信息的一致。

3.2.4 研究方法的局限性

应用的代表性： 我们只选取了运行在开源框架（也就是 Apache Hadoop 和 Apache Spark）上的应用。尽管很多公司（比如 Facebook 和 Yahoo!）都在使用这两个框架 [HadoopPB, SparkPB]，一些大公司建立了自己的分布式数据并行框架（比如 Microsoft 的 Dryad），我们还没有研究运行在这些私有框架上的应用。

错误类型的完整性： 我们只研究了 276 个错误用例中的 123 个，还有 153 个的错误原因是未知的。这 153 个错误中可能还存在新的错误原因或者修复方法。另外，当一个应用的内存溢出错误存在多个错误原因时，用户的错误描述和专家的诊断信息可能只覆盖了主要的错误原因。比如，专家可能只诊断出来错误原因是用户代码缺陷，但另

外一个原因可能是框架在内存中暂存了大量的数据。

分类的主观性：尽管我们已经尽力去理解、判断和归类错误原因，可能仍然存在不准确的分类。比如，对于一个应用的内存溢出错误，专家可能诊断出来多个错误原因，但我们只选取被用户接受的错误原因。

3.3 RQ1（常见错误原因）的研究成果

尽管这 123 个内存溢出错误的错误原因多种多样，我们可以根据其与数据存储、数据流和用户代码的关系将错误原因分成 3 大类（共 10 个小类），详见下表：

表 3 内存溢出错误的常见错误原因

错误类别	错误原因	原因描述	Hadoop	Spark	总数	比例
框架暂存了大量数据	框架缓冲(buffer) 了大量数据	大量的中间数据被存放到了框架的缓冲区里	6	2	8	6%
	框架缓存(cache) 了大量数据	用户显式地将大量数据缓存到内存用(用于数据重用)	0	7	7	6%
	子类别总数		6	9	15	12%
数据流异常	数据划分不恰当	一些数据块(partition) 变的太大	3	13	16	13%
	热点 key	运行时一些 $\langle k, list(v) \rangle$ 变得太大	15	8	23	18%
	单个 record 太大	运行时单个 $\langle k, v \rangle$ 太大	6	1	7	6%
	子类别总数		24	22	46	37%
内存使用密集的用户代码	用户代码加载了大量外部数据	用户代码在未处理 record 之前，加载了大量的外部数据	8	0	8	6%
	单个中间计算结果过大	用户代码在处理单个 record 的时候产生了大量的中间计算结果	4(3)*	2	6(3)	5%
	累积中间计算结果过大	用户代码在内存中累积了大量的中间计算结果	30[13]*	10[1]	40[14]	33%
	Driver 生成了大量的数据	Driver 程序本身产生了大量的数据	0	9	9	7%
	Driver 收集的计算结果太大	Task 输出的计算结果太大，被 driver 全部收集	0	16	16	13%
	子类别总数		42	37	79	64%
	总数		72	68	123+17	113%

*注：4(3)表示 4 个内存溢出错误中有 3 个的错误原因既包含“单个中间计算结果太大”也包含“单个 key/value record 太大”。30[13]表示 30 个内存溢出错误中有 13 个的错误原因既“中间计算结果累积量太大”，也包含“热点 key”。113%的意思是 30 个内存溢出错误中有 13 个有两个错误原因类型。

上表说明最主要（占 64%）的错误原因是用户代码消耗了过多的内存。第二大（占 37%）的错误原因是数据流异常，13%的错误既是由于用户代码消耗了大量内存也是由

于数据流异常导致的。

3.3.1 错误类别 1：框架暂存了大量的中间数据

A. 框架缓冲（buffer）了大量数据

为了降低磁盘的读写 I/O，框架一般会在内存中先对中间数据（即 map() 的输出和 reduce() 的输入数据）进行缓冲（buffer）。缓冲区有两种：固定缓冲区和虚拟缓冲区。固定缓冲区直接占内存的大块空间，比如 Hadoop 的 map buffer 是一个大的 byte[] 数组。虚拟缓冲区只是一个内存界限，用来限定内存中最多有多少空间可以用来存储中间数据。Hadoop 和 Spark 都有一个虚拟缓冲区叫做 shuffle buffer，用来暂存从 map 任务 shuffle 过来的的数据。

有 8 个（6%）内存溢出错误可以被划分到这个类别中。4 个错误是因为 Hadoop 的 map buffer 分配的过大（通过 *io.sort.mb* 设置）。比如，一个错误用例配置了 300MB 的 map buffer，但给 mapper 分配的内存只有 200MB [e01]。另外 4 个错误是由于 shuffle buffer 开的过大。比如，一个用户将 shuffle buffer 设为 0.7（也就是 shuffle buffer 可以占用 70% 的 reducer 内存空间）。由于这个值太大（设置为 0.3 可以成功执行），应用出现了内存溢出错误。

B. 框架缓存（cache）了大量重用数据

除了框架可以在 buffer 中暂存数据以外，用户也可以显式地在框架中缓存（cache）数据（一般是用于重用的数据）。在一些应用，特别是迭代型的机器学习应用或者图计算应用（比如 PageRank 和 Kmeans 聚类）中，输入数据（比如输入的原始图或者训练数据）或者一些中间计算结果（比如权重参数）可以在不同的 jobs 中进行重用。Hadoop 的重用方法是将当前 job 的输出数据写入分布式文件系统（HDFS），然后下一个 job 可以从 HDFS 中读取。Spark 提供了内存缓存机制，用户可以显式地指定哪些中间数据要缓存到内存中。如果在 Spark 中缓存的数据量太大，会导致内存溢出的错误。

这个子类别包含 7 个内存溢出错误（占 6%）。这些错误都是由于 Spark job 在内存中缓存了大量的重用数据（RDD）。比如，一个应用在试图缓存 job 最后输出的大的 RDD 时产生了内存溢出错误 [e03]。有 3 个用例的错误原因是 job 不断在内存中缓存数据。比如，一个机器学习应用 *SVDPlusPlus* 在每次迭代中均在内存中缓存图数据，最后因为数据量太大导致内存溢出错误 [e04]。最后一个错误因为用户显式地缓存了 driver 程序广播过来的大的数据 [e05]。

通过这两个错误类别的实证分析，我们得到研究发现：

Finding 1: 尽管用户可以通过配置参数来限制框架的内存使用量，但还是有很多（12%）的错误是由于框架暂存大量的中间数据导致的。

Implication: 用户很难设置正确的配置参数来调配框架和用户代码的内存用量。

3.3.2 错误类别 2: 数据流异常

因为数据在框架中是被分布处理的，而且在每个处理步骤中的数据量大小（比如 key/value record 大小和 key 分布）是运行时动态决定的，所以框架不能避免产生数据流异常。这里，数据流异常包括：产生大的数据分块（large data partition），大的 $\langle k, list(v) \rangle$ group，大的单一 key/value record 对。数据流异常可以直接导致内存溢出错误，也可以影响用户代码的内存用量。该类别包含三种错误类型。

A. 不合适的数据划分

为了实现“分治一聚合”的并行处理模式，框架将 mapper 的输出数据进行划分（partition），然后不同的数据分块（partition）会被汇集（shuffle）到不同的 reducer 进行处理。在 Hadoop/Spark 中，map()会根据输出的 key/value records 的 *partition id*（比如 $id = hash(key) \% partitionNumber$ ）将 records 输出到不同的数据块。比如在下图中， $K1$ 和 $K4$ 有相同的 partition id。在相同 partition 中的 records 会被相同的用户代码（reduce() 或 combine()）处理。不合适的数据划分包含两种子类别：(1) 当 partition number 比较小的时候，所有的数据分块都可能很大。(2)不均衡的 partition 函数可以导致某些数据分块远远大于其他数据分块。常用的 partition 函数（比如 hash partition 和 range partition）都不能避免产生不平衡的数据分块。不合适的数据划分可以导致暂存于内存中的中间数据量过大。比如在下图中， $P1$ 包含的 key/value records 个数远远大于 $P2$ 和 $P3$ ，那么在 shuffle $P1$ 的时候 reducer 需要在内存缓冲更多的数据。更严重的是，如果把 Spark 的 spill 参数设置为 false，那么 Spark 会将整个 $P1$ 都缓冲到内存中，可以直接导致内存溢出错误。不合适的数据划分同样可以导致用户代码的输入数据量过大，由于用户代码的内存消耗量通常与处理的数据量有关，那么输入数据量过大可以导致用户代码产生内存溢出错误。比如在下图中，如果 $P1$ 包含的 key/value records 个数远远大于 $P2$ 和 $P3$ ，那么 reduce() 可能在处理 $P2$ 和 $P3$ 的时候运行正常，但在处理 $P1$ 时产生内存溢出错误。

这个类别包含 16 个内存溢出错误。其中 7 个错误是因为 partition number 设置的过小。比如，将 partition number 增加到 1000 后，用户报告说每个数据分块变的很小，也没有内存溢出错误了 [e06]。有 4 个错误是因为 partition 函数不平衡导致的。比如，一个用户报告说大多数中间数据都聚集到了 2 个 reducer 那里，其中的 1 个 reducer 的输入

数据超过了总数据量的一半 [e07]。在剩下的 5 个错误中，用户只报告说数据分块变的很大，但没有详细描述 partition number 和 partition 函数的情况。

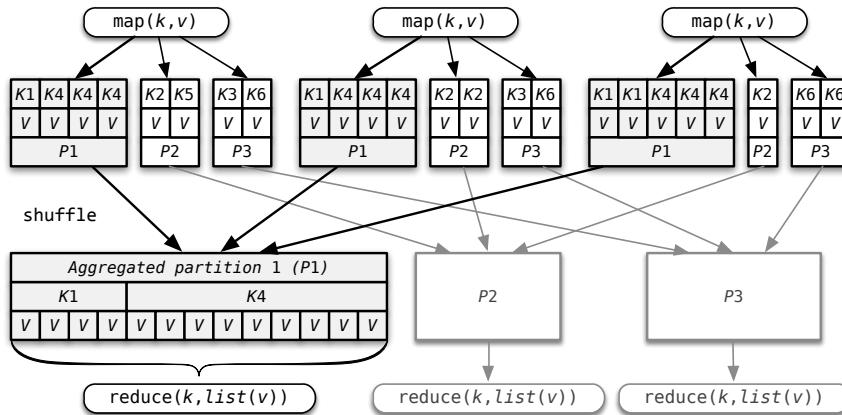


图 13 不合适的数据划分和热点 key 例子

B. 热点 key

一个数据分块可以看作是一个粗粒度的 key/value records 集合，集合里面的 records 仍然可以有不同的 key。尽管在一个 partition 里面的 records 要被同样的用户代码处理，这些 records 要先根据 key 被聚集成不同的 $\langle k, list(v) \rangle$ groups。然后用户代码（reduce() 或者 combine()）处理 group 里面的每一个 record。热点 key 的意思是某些 $\langle k, list(v) \rangle$ groups 包含的 $\langle k, v \rangle$ records 远远多于其他 groups。Partition 个数可以影响每个 partition 的大小，但不能影响每个 $\langle k, list(v) \rangle$ 的大小，因为 group 的大小是运行时动态决定的（也就是多少个 records 具有相同的 key）。比如在图 9 中，如果 $K4$ 对应的 $\langle K4, list(v) \rangle$ group 包含的 records 个数远远大于 $K1$ 对应的 $\langle K1, list(v) \rangle$ ，那么框架在聚合 $\langle K4, list(v) \rangle$ 时可能会出现内存溢出错误，处理 $\langle K4, list(v) \rangle$ 的用户代码也可能产生内存溢出错误。

这个类别包含 23 个内存溢出错误（占 18%），这些错误均是由于某个 key 对应的 values 个数太多。比如，一个用户报告说错误原因是一些 key 只有 1 个或 2 个 values，但其他一些 key 对应 10 万个 values [e08]。另外一个用户报告说某个 key $\langle custid, domain, level, device \rangle$ 严重倾斜，大约 42% 的 records 都包含这个 key [e09]。另外 6 个错误是由于 Spark 的 `groupByKey()`, `reduceByKey()` 和 `coGroup()` 等聚合函数出现了热点 key。比如，一个用户报告说某个 key 对应太多的 values，这些 values 不能被存放到内存中 [e10]。专家解释说当前的 `groupByKey()` 要求一个 key 对应的所有 values 都能够存放到内存中 [e11]。

C. 单个 key/value record 过大

因为用户代码需要将整个 record 读入内存进行处理，所以如果一个 key/value record 过大，那么用户代码会直接出现内存溢出错误，或者在处理这个 record 的时候产生大量的中间计算结果，导致内存溢出错误。因为 record 的大小是动态决定的，所以数据流相关的配置也不能控制 record 的大小。

这个错误类别包含 7 个内存溢出错误（占 6%），都是因为单个 record 过大。比如，一个用户报告说内存只有 200MB 的空间，但应用产生了一个 350MB 的 record（这个 record 只包含一行字符都是 *a* 的文本）[e12]。另外一个用户报告说一些 record 只有 1MB，但另外一些有 100MB [e13]。更严重的是，一个用户报告说因为数据格式变化但分割符没有变化，导致全部的 100GB 数据变成了一个 record [e14]。

通过这三个错误类别的实证分析，我们得到研究发现：

Finding 2: 数据流异常很常见（占 37% 的错误），异常数据流可以导致运行时数据量过大，比如 large data partition, hotspot key 和 large single key/value record。
 Implication: 框架在进行数据划分时没有考虑运行时数据的特征（比如 key 分布），导致中间数据量和用户代码的输入数据量不可控。

3.3.3 错误类别 3：内存使用密集的用户代码

不同于传统的单机程序，分布式数据并行应用的用户代码有一个重要特征就是 streaming-style。Key/value records 被用户代码一个个按顺序读取，处理，最后输出。所以，当一个 record 被处理完以后，该 record 及其对应的中间处理结果会被清空。除非这个 record 或其对应的中间结果以后会被用到（被用户代码显式缓存）。基于这个特征，我们总结出三种错误类型：单个中间计算结果太大，累积中间计算结果过大，用户代码加载了大量的外部数据。另外，driver 程序也能触发内存溢出错误，相应的错误类型有两种：一是 driver 程序本身产生了大量的数据，二是 driver 收集了过大的 task 的处理结果。

A. 用户代码加载了大量的外部数据

不同于被框架暂存（buffer/cache）的数据，外部数据指的是被用户代码直接加载的数据。一些应用在处理 $\langle k, v \rangle$ records 之前，需要从本地文件系统、分布式文件系统、数据库或者其他地方加载一些数据用来辅助处理 records。比如，为了查询 input records 的 key 是否在一部字典里，用户代码会加载一部字典到内存中（比如 HashMap 里面）。如果加载的外部数据太大，那么会直接导致内存溢出错误。

这个错误类别包含 8 个内存溢出错误（占 6%）。有 3 个错误出现在 Mahout 应用中，

这些应用中的 mapper 从分布式文件系统上加载了大量的训练模型数据用来做分类 [e15] 和聚类 [e16]。另外一个错误发生在 Hive 应用里，这个应用加载了一个大的外部表 [e17]。还有一个错误出现在 Pig 脚本的 UDF（用户定义函数）里面，这个 UDF 加载了一个很大的外部文件 [e18]。

B. 单个中间计算结果过大

中间计算结果指的是用户代码在处理单个 $\langle k, v \rangle$ record 过程中产生的计算结果。这个错误类型包含两种情况：(1) 输入的单个 $\langle k, v \rangle$ record 很大，导致代码产生的中间结果也很大。比如，如果一个 record 是一个 64MB 的句子，那么这个句子被分词后，也会占用 64MB 左右的空间。(2) 一个很小的 input record 仍然可以产生较大的中间处理结果。比如，一个 record 包含两个集合，那么这两个集合的笛卡尔集要比这个 record 大的多。

这个错误类别包含 6 个内存溢出错误（占 5%）。其中 3 个用例的错误原因是 input record 过大。另外一个错误是由于 reduce()生成了一个非常长的 record [e19]。在另外一个错误用例中，一个机器学习应用在内存中建立了一个非常大的训练模型用于矩阵分解 [e20]。最后一个错误发生在文本处理应用 [e21] 中，这个文本处理程序使用了第三方的库 *StanfordLemmatizer* 来处理文本，代码见下表。在处理文本中的每一行 (*line*) 的时候，*lematize()*会分配一个大的数据结构用来进行动态规划，该库的作者解释说这个数据结构是输入 *line* 的 3 倍。当一个 *line* 过大时，*lematize()*会产生内存溢出错误。

```
public class Mapper {
    StanfordLemmatizer slem = new StanfordLemmatizer();
    public void map(Long key, Text value) {
        String line = value.toString();
        for(String word: slem.lemmatize(line))
            emit(word, 1);
    }
}
```

C. 累积的中间计算结果过大

如果当前 record 对应的中间计算结果被显式地缓存到内存中，那么这些计算结果会成为累积计算结果。这样，records 被处理的越多，内存中就会累积越多的中间计算结果。比如，为了对 input records 进行去重，map()会分配一个 Set 用来持有不同的 records。如果不同的 records 个数很多，那么该 Set 也会变得很大。reduce()在处理一个大的 $\langle k, list(v) \rangle$ 的时候也会产生大量的累积中间结果，这个大的累积中间结果可能是由于热点 key 导致的。

这个错误类别包含 40 个内存溢出错误（占 33%）。在 11 个错误用例中，用户代码分配了内存数据结构用来持有 input records。比如，一个用户使用 *ArrayList* 来持有一个 key 对应的所有的 values，而 value 的个数可能达到 1000 万个 [e22]。在其他的错误中，用户试图去累积中间结果，比如累积 word 的次数 [e23] 和训练参数 [e24]。用户代码也会累积中间计算结果，比如去累积不重复的元组 [e25]，累积元素进行基于内存的排序，计算中值，或者做笛卡尔集 [e26]。下面代码产生的内存溢出错误发生在 *reduce()*，*reduce()* 分配了一个类似 *HashMap* 的数据结构 *HMapStIW* 用来持有每个词（word）的邻接词。因为文档中的 word 过多，而且每个 word 有很多邻接词，具有累积逻辑的 *plus()* 将太多的词存放到了 *map* 数据结构中，最终导致了内存溢出错误。

```

public class Reducer {
    void reduce(Text key, Iterable<HMapStIW> values) {
        Iterator<HMapStIW> iter = values.iterator();
        HMapStIW map = new HMapStIW();
        while (iter.hasNext()) {
            map.plus(iter.next());
        }
        emit(key, map);
    }
}

```

D. Driver 产生或收集了大量的数据

尽管 driver 程序不直接处理 $\langle k, v \rangle$ records，driver 程序仍然会出现内存溢出错误，而且有两个错误类别：(1) driver 程序本身在内存中生成了大量数据。(2) driver 收集了大量的 task 处理结果。

第一个错误类别包含 9 个内存溢出错误（占 7%）。Driver 程序自己生成数据有两个目的：一是为了广播到其他节点，一是为了在本地计算某些参数。用于数据广播的用例有：一个 driver 程序产生了 1GB 的数组 [e28]，另外一个产生了 0.15GB 的变量 [e29]。用于本地计算的用例：一个 driver 程序产生了 8000×8000 的矩阵，大约 256MB [e30]。另外一个 driver 为了计算条件概率，产生了 4 亿个 double 变量。

第二个错误类别包含 16 个内存溢出错误（占 13%）。例如，一个 driver 程序使用 *graph.edges.collect()* 将 4.5GB 的图的所有边都收集并放到内存 [e32]。另外一个 driver 将 *reduceByKey()* 的计算结果（包含 2 亿个 word）收集起来放到了内存 [e33]。另外，在一个迭代型应用里面，driver 把每一轮迭代 task 产生的结果都收集起来放到内存，这样结果在内存慢慢积累，最后导致内存溢出。

通过上面四个错误类别的实证分析，我们得到研究发现：

Finding 3: 内存使用密集的用户代码是最主要的错误原因（占 64% 的错误），错误原因是由于用户代码不小心处理了大量数据或者产生了大量的计算结果。

Implication: 用户很难在不知道运行时输入数据量的情况下设计出内存使用效率高的代码。

3.4 RQ2（常见修复方法）的研究结果

从 RQ1 的研究结果可以看到内存溢出错误与数据存储、运行时数据流和用户代码都有关系，所以修复内存溢出错误不是一件容易的事。用户一般采取的方法是去提高内存界限，但这种方法没有深入考虑数据处理的机制，即使分配更多的内存使用空间，也会被迅速耗尽。

为了找到可靠的修复方法，我们研究了 42 个包含修复信息的内存溢出错误用例，从修复信息（来自专家修复建议或者用户自己的修复方法）中总结得到 11 种修复方法。其中 8 种是比较传统的方法，3 种是有些 tricky 的方法，比如重新设计 key，跳过异常大的数据，更改应用相关的参数。表 4 总结了每种错误类型对应的修复方法（有 3 种错误类型缺少相应的修复方法）。标签 C 表示该修复方法需要修改配置参数，标签 U 表示该修复方法需要更改用户代码。*Errors(n)* 表示该修复方法包含相应的错误个数和被修复的错误个数（n）。

表 4 内存溢出错误常见修复方法

修复方法类别	错误原因	修复方法	C	U	Errors(n)
数据存储相关的修复方法	框架缓冲（buffer）了大量数据	降低缓冲区大小	✓		6 (6)
	框架缓存（cache）了大量数据	降低缓存界限，或者使用基于磁盘的缓存	✓		2 (2)
数据流相关的修复方法	数据划分不恰当	增加 partition 个数，或者更改 partition 函数	✓		12 (6)
	热点 key	重新设计 key（比如使用组合 key）		✓	3 (0)
	单个 key/value record 太大	把大的 record 拆分成小的 records		✓	4 (1)
用户代码相关的修复方法	累积中间计算结果过大	将累积操作变成流式操作		✓	2 (2)
		把累积操作拆分成多个轻量级操作		✓	3 (1)
		把部分累积计算结果 spill 到磁盘上		✓	3 (1)
		跳过异常大的数据		✓	2 (2)
	Driver 收集的计算结果太大	使用树形聚集来替代直接收集		✓	3 (2)
		调整应用本身的参数	✓		2 (2)
总数					42 (25)

3.4.1 数据存储相关的修复方法

该类别包含两种可以减少框架暂存数据的修复方法。这两种修复方法都只需更改内存相关的配置参数。

A. 降低缓冲区大小

降低缓冲区大小的意思是减小 map buffer 或者减小 shuffle buffer。两者都可以直接减小框架暂存的数据量，但代价是会有更多的磁盘 I/O。

这个修复方法已经修复了 6 个内存溢出错误。其中 3 个是通过减小 Hadoop 的 map buffer(比如把 *io.sort.mb* 从 128MB 调小到 64MB[e35])修复的。有 2 个是通过降低 Hadoop 的 shuffle buffer 修复的。比如，一个错误修复方法是把 *shuffle.input.buffer.percent* 从 70% 调小到 30%，但会导致更长（2 个小时）的执行时间 [e02]。最后一个错误采用一种极端的修复方法，把 Spark 的 shuffle buffer 设置为 0。这样做虽然会消除内存溢出错误，但导致执行时间变长了 20% [e36]。

B. 降低框架的缓存界限

缓存界限是一个配置参数，用来限制可以用于数据缓存的内存空间大小。降低缓存界限可以直接减少缓存的数据量，代价就是大的数据块不能被缓存。为了解决这个问题，Spark 提供了多种内存+磁盘的 cache 机制，用户可以选择多种存储模式，memory-only，memory+disk，或者 disk-only。

这个修复方法已经修复了 2 个内存溢出错误。其中一个是将缓存界限 (*spark.storage.memoryFraction*) 从 0.66 调小到 0.1 [e37]。另外一个是将存储模式从 memory-only 改为 disk-only，代价就是执行速度会变慢。

3.4.2 数据流相关的修复方法

由于数据 partition 过大，热点 key，单个 key/value record 过大都可以导致内存溢出错误，相应的修复方法就是要对运行时的数据进行进一步划分，具体包含 3 种修复方法。

A. 改变 partition 个数或 partition 函数

数据块 (partition) 的大小与 partition 个数和 partition 函数直接相关。一种直接减小 partition 的大小的方法就是增加 partition 个数，尽管这不是 100% 可靠的方法。另外一种修复方法是尝试其他 partition 函数来避免不平衡的 partition，比如使用 *range partition*, *round-robin partition*, 或者自定义的 partition 函数。

有 12 个内存溢出错误用例用到了这种修复方法。其中 6 个错误可以通过增加

partition 个数来修复。在 Hadoop 中，partition 个数的配置参数就是 *reduce number*。在 Spark 种，partition 个数的配置参数是 *spark.default.parallelism*。比如，当把 partition 个数提高到 1000 后，用户报告说每个 partition 的数据量减少很多，内存溢出错误也消失了 [e06]。一个用户报告说“我唯一的修复方法是去增加 partition number，但这好像不是一种绝对可靠的方法”。专家解释说当 key 分布均匀时，增加 partition 个数才是一种可靠的修复方法 [e39]。在另一个错误中，用户尝试对一个 range partition 函数调整其 range 来避免不平衡的划分。在剩余的 4 个错误中，用户只是被建议增加 partition 个数。

B. 重新设计 key

重新设计 key 的目的是让 key 分布更均匀，避免热点 key 的出现。。一种简单的方法是使用组合 key 来替代单一 key。比如 map()可以输出一个 key 组(k_1, k_2)来替代单一 k ，如下：

$$\begin{aligned} \text{map}(k, v) &\Rightarrow \text{list}((k_1, k_2), v) \\ \text{reduce}((k_1, k_2), \text{list}(v)) &\Rightarrow \text{list}(k, v) \end{aligned}$$

这样，原来很大的 $\langle k, \text{list}(v) \rangle$ group 就被划分为多个小的 $\langle (k_1, k_2), \text{list}(v) \rangle$ groups。

有 3 个内存溢出错误用到了这个修复方法。在第一个错误用例中，某个 key (*vehicle*) 对应太多的 values。专家建议用户使用 (*vehicle, day*) 作为 key，这样可以减少一个 key 对应的 values [e40]。在另外一个错误中，专家建议用户在原始 key 后面加上一个哈希值（比如 1-10）来将大的 $\langle k, \text{list}(v) \rangle$ group 划分为多个小的 groups [e11]。在最后一个错误中，用户建议在 key 后面加上 1 或 2 [e26]。

C. 对单个 $\langle k, v \rangle$ record 进行划分

这个修复方法是要将大的单一 record 划分为多个小的 records。

有 4 个内存溢出错误使用了这个修复方法。在一个错误用例中，专家建议用户在 map()中将 100MB+的 record 划分为多个小的 records，这样接下来的 reduce()处理的 record 会小[e41]。在另一个错误用例中，专家建议用户不要输出一个巨大的字符串，而要将字符串划分为多个小的字符串输出 [e42]。在第三个错误用例中，专家建议用户使用 *pos maxlen* 配置参数来将过长的句子划分为多个短的句子 [e21]。在最后一个错误用例中，修复方法是将大的 record(用于倒排索引的一个长链表)划分为多个小的 record，然后让每一个 key 输出多个 records [e43]。

3.4.3 用户代码相关的修复方法

下面列举的修复方法只针对用户代码中的 map() 和 reduce()。

A. 将累积操作转换为流式操作

因为累积操作是造成内存溢出的重要原因，所以专家建议用户将累积操作转换为流式操作。

有两个内存溢出错误用到了这个修复方法，这两个错误都发生在基于内存进行排序的用户代码。为了对 $\langle k, list(v) \rangle$ 中的 $list(v)$ 进行排序，用户代码在内存分配一个 $List$ 来持有每一个 v ，这个累积操作导致了内存溢出错误。专家建议用户利用框架的排序机制来将累积操作转换为流式操作。具体方法是在 key 中添加 v ，原来的 $\langle k, v \rangle$ record 变成 $\langle (k, v), v \rangle$ record。这样当框架在排序 key 的时候，同时会对 v 排序。最后结果就是，用户代码不用分配任何数据结构，只需进行流式操作（即将读入的 $list(v)$ 中的 v 一个个输出）[e44]。

B. 把累积操作拆分成多个轻量级操作

这个修复方法是将内存消耗大的累积操作拆分成多个轻量级操作。轻量级操作是指该操作只处理部分数据或者具有较小的内存消耗。

有 3 个错误用到这个修复方法。在其中一个错误中，专家建议用户在 map() 之后和 reduce() 之前加上 combine()（也就是一个迷你 reduce()）来提前进行部分聚合。这样，reduce() 就会收到较少的数据（更少的不重复的 values）[e45]。在另一个错误中，专家建议用户使用使用 `aggregateByKey()` 来替代 `groupByKey()`，因为前者可以进行本地聚合，这样可以使得 `shuffle` 和 `reduce()` 的输入数据量变小 [e46]。最后一个错误 [e47] 的修复方法是将一个聚合操作拆分为两个轻量级操作。输入数据是一张包含多列的大表，用户想要统计 `column A` 中每个不同的元素在 `column B` 中对应的不同元素数。用户先进行 `groupby(A)` 操作，然后进行 `count(distinct B)` 的聚合操作。由于列 `A` 中的某些元素对应列 `B` 中的不同元素过多，导致 `count(distinct B)` 操作出现内存溢出错误。专家建议用户将累积操作拆分为两个轻量级操作如下：

原始版本：`groupby(A) ⇒ count(distinct B)`

优化版本：`groupby(A, B) ⇒ output(A, B) ⇒ (1)`

`groupby(A) ⇒ output(B) (2)`

第一步先将 `A` 和 `B` 联合成一个 key 进行聚合，也就是使用 `groupby(A, B)`，得到的

结果是不同的(A, B)。第二步进行 $groupby(A)$, 然后直接输出 $groupby$ 的结果（也就是不同的 B ）即可。

C. 将部分累积计算结果 spill 到磁盘上

这个修复方法不是去优化用户代码, 而是在累积的中间结果变得过大之前将这些中间结果 spill 到磁盘上。

有 3 个内存溢出错误用到了这个修复方法。在其中一个错误用例中, $map()$ 每处理 n 个 records, 就把中间计算结果 spill 到磁盘上 [e23]。在一个 Hive 错误中, 专家建议用户将 *hive.map.aggr.hash.percentmemory* 从 0.5 调小到 0.25, 这意味着在 $map()$ 中进行结果累积的 *HashMap* 占到内存总量 25% 的时候会将存放的数据 spill 到磁盘上 [e48]。同样, 在一个 Pig 错误用例中, 专家建议用户将 *pig.cachedbag.memusage* 设为 0.1 或者更低。这样, 当用户代码在内存中存放的内部数据 (bag) 达到这个界限时会 spill 到磁盘 [e25]。

D. 跳过异常大的数据

这个修复方法的意思是当遇到特别大的 record 或者 $\langle k, list(v) \rangle$ group 时, 直接跳过这个数据而不处理。这样 tricky 的修复方法产生的原因是用户有时难以去优化用户代码, 尤其是当用户代码使用了第三方包的时候。这个修复方法同样适用于那些某些不需要精确结果的应用, 比如对大规模数据进行简单统计的应用。

有 2 个内存溢出错误用到了这个修复方法。第一个错误发生在一个视频推荐应用里面。出错原因是这个应用的用户代码处理了异常数据 (有一个用户包含了 10 万个观影信息)。修复方法是跳过这些异常用户的数据, 因为这些用户可能是机器人, 而且这个应用最多只需要一个用户的 100 部观影信息来进行特征建模 [e49]。在另一个错误用例中, 热点 key 对应的 values 被直接跳过, 因为它们对最终结果并没有产生影响 [e38]。

3.4.4 Driver 程序相关的修复方法

针对 driver 收集大量 task 计算结果而导致内存溢出的情况, 有两种相应的修复方法。

A. 使用树形聚合 (Tree aggregation)

这个修复方法以多层树的方式来收集和聚合从 task 那里获得的输出结果。在树的第一层, 该方法对 task 的输出数据进行两两聚合, 聚合后 task 输出结果只剩一半。接着, 在第二层到第 n 层, 仍然执行相同的两两聚合操作, 最后的两两聚合由 driver 程序来完成, 但此时存在于 driver 中的数据将会很少。

有 2 个 **MLlib** 应用出现内存溢出错误时用到了这个修复方法。用户使用树形聚合方法来避免直接去收集过多的 task 结果（比如向量 [e50] 和梯度 [e51]）。

B. 调整应用本身的参数

为了修复内存溢出错误，专家和用户甚至尝试去调整应用本身的参数。在一些机器学习应用，比如 *SVD* 和 *ALS* 中，task 的输出数据大小与应用本身的参数（比如 *SVD* 中的 *k* 和 *ALS* 中的 *rank*）存在线性或多项式相关性。降低这些参数的值，会使得 task 的输出数据变小。

有 2 个 **MLlib** 应用出现内存溢出错误时用到了这个修复方法。第一个是去调小 *SVD* 的参数 *k*，第二个是把 *ALS* 的 *rank* 调小到 40 以下。

Finding 4: 常见的修复方法包含：调整内存或者数据流相关的配置，对运行时数据进行动态划分，优化用户代码逻辑。我们也很惊奇地发现一些 tricky 的修复方法，比如使用组合 key，跳过异常大的 records 等。

Implication: 没有一种统一的方法去修复内存溢出错误。一般的修复方法是去限制框架暂存的数据大小、降低运行时数据大小、减少用户代码的内存消耗。

3.5 RQ3（框架改进方法）的研究结果

因为内存溢出错误不能被现有的错误容忍机制解决，我们需要新的错误容忍机制和错误诊断方法。

我们从总结出的错误原因和错误诊断经验中发现两种可以提高框架错误容忍能力的方法以及一种辅助错误诊断的方法。

3.5.1 辅助错误诊断的方法

A. 提供详细的数据流统计信息

数据流异常是内存溢出错误的常见原因，但是当前的框架只提供了很少的数据流信息。比如，Hadoop 只统计当前已经处理过的 records 个数，而 Spark 还需要用户自己写程序去统计当前已经处理的 records 个数。

如果框架能够提供详细的数据流统计信息，那么内存溢出错误诊断将会变得简单。数据统计信息包括已经处理过的 records 统计信息，比如最大、最小、平均 record 大小，每个 $\langle k, list(v) \rangle$ 组中包含的 records 最大、最小、平均个数，每个数据块中包含的 records 最大、最小、平均个数。我们在 Hadoop 中实现了一个数据流统计器，该统计器监控得到的数据流统计信息帮助我们诊断出 8 个数据流异常错误原因，包括 1 个不合适的数据

划分，6个热点 key，和1个单一 key/value record 过大。

3.5.2 提高框架的错误容忍能力方法

A. 设计动态内存管理方法

分析错误原因后可以发现，有12%的内存溢出错误的原因是框架暂存了大量的数据。因此，我们可以得知用户很难设置正确的配置参数来调配框架和用户代码的内存占用比。

动态内存管理目的是去动态调配框架和用户代码的内存用量。直接实时计算用户代码的内存消耗量是困难的，但我们可以实时监控总的内存消耗量 (*total*) 和框架暂存的数据量 (*du*)。可以将两者之差 (*total-du*) 简单地当作用户代码的消耗量。当内存总消耗量 *total* 达到一个界限时，框架可以暂停执行用户代码，将暂存的数据序列化或持久化到磁盘，然后恢复执行用户代码。同样，当用户代码的内存消耗降低到一个界限时，可以将序列化或持久化到磁盘的缓存数据读回到内存。

B. 提供内存磁盘混合数据结构

用户代码经常使用内存数据结构来对 key/value records 或者中间计算结果进行聚合。但这种操作很容易导致内存溢出错误，比如我们发现有18个(15%)内存溢出错误在内存数据结构中出现，这些数据结构包括 *List/ArrayList*(9个错误)，*Map/HashMap*(5个错误)，*Set/HashSet*(3个错误)和 *PriorityQueue*(1个错误)。

一个理想解决方案是为用户提供常用的内存磁盘混合数据结构。新的数据结构应该为用户提供与原始数据结构类似的 API (比如 Java 的 Collections 和 C++的 STL)。与通用程序语言中的数据结构不同的是，新的数据结构应该能够根据当前的内存用量自动在内存和磁盘上进行 swap。

Spark 已经实现一个类似 *HashMap* 的内存磁盘数据结构叫做 *ExternalAppendOnlyMap*。*AppendOnly* 的意思是可以向这个数据结构添加或更改 key，但不可以删除 key。然而这个数据结构不能像通用语言里面的数据结构那样支持增删改查，而且不能被用户代码使用(当前只在框架提供的操作，比如 *reduceByKey()* 中使用)。其他的开源项目，比如 STXXL [STXXL] 和 TPIE [TPIE] 已经实现了全磁盘的数据结构和相应的算法，适用于数据量太大无法放入内存中的情况。如果能将这些数据结构进行改写，支持内存和磁盘混合存储，而且针对 key/value records 聚合进行优化，那么可以很好地降低内存溢出错误发生的概率。

Finding 5: 当前框架为内存溢出错误的诊断和错误容忍提供了非常有限的支持。

Implication: 框架可以提供多种机制来辅助错误诊断，进行错误容忍：提供详细的数据流统计信息，提供动态内存管理策略，提供方便进行 key aggregation 且可以在内存和磁

盘进行 swap 的数据结构。

3.6 讨论

A. 研究结果的通用性

我们的研究对象是 Apache Hadoop 和 Apache Spark 的应用。大部分研究结果可以适用于其他类似 MapReduce 编程范型的框架，比如微软的 Dryad [Isard2007] 和 Naiad [Murray2013]，Yahoo! 早期的 Map-Reduce-Merge [Yang2007]，和最近正在开发中 Apache Flink [Flink]。这些框架与 MapReduce 有类似的编程模型和分布式数据流。

B. 应用的覆盖情况

当前我们主要研究的是通用处理框架（也就是 MapReduce 及 MapReduce-like）的应用。尽管这些应用已经覆盖了文本处理，SQL 处理，机器学习，图处理等，实际还存在很多为专门应用设计的框架。比如 Pregel [Malewicz2010] 和 Powergraph [Gonzalez2012] 是专门用来进行大规模图处理和分析的框架，Apache Storm [Storm] 是专门用于流处理的框架。我们把研究这些非 MapReduce 框架应用中的内存溢出错误作为我们未来的研究工作。

当前我们只研究了公开的内存溢出错误，但实际上我们也研究了工业界（包括阿里巴巴淘宝和腾讯）里面的 Hadoop/Spark 应用发生的内存溢出错误。只是考虑到商业机密原因，我们无法将应用本身及其错误原因进行详细分析。然而，我们发现工业界的内存溢出错误（比如 [TencentOOM]）与我们研究的公开错误（比如[e47]）有相同的错误原因和修复方法。

3.7 相关工作比较

A. 大数据应用错误的实证分析

很多研究人员已经实证分析了大数据应用或大数据系统的故障错误。Li [Li2013] 等人研究了 250 个 SCOPE job（运行在微软的 Dryad 框架之上）的故障错误，发现错误主要是未定义的列，错误的数据模式，不正确的行格式等等。他们也发现 3 个内存溢出错误，错误原因是在内存累积了大量的数据（比如一个大表的所有行被存放到内存中）。Kavulya 等人 [Kavulya2010] 分析了 4100 个执行失败的 Hadoop jobs，这些 jobs 运行在 Yahoo! 管理的 M45 集群。他们发现 36% 的故障是数组访问越界错误，还有 23% 的故障是 IO 异常。Xiao 等人 [Xiao2014] 研究了 MapReduce 程序（同样是 SCOPE 程序）执行结果的正确性。他们研究发现 5 种非交换的 reduce() 如果运行多次会输出不一致的运行结果。

Zhou 等人 [Zhou2015] 研究了微软大数据处理平台的质量问题。他们发现 36% 的

故障是系统缺陷导致的，有 2 个故障（1%）与内存有关。Gunawi 等人 [Gunawi2014] 研究了 3655 个云计算系统（比如 Hadoop, HBase）中出现的开发和部署故障。他们发现一个 HBase 中的内存溢出错误，错误原因是用户提交的查询请求需要在大规模的数据上进行。他们也发现一个 HDFS 中出现的内存溢出错误，原因是用户并行地创建了上千个小文件。

B. 减少大数据应用的内存消耗

一些研究者提出了一些减少大数据应用内存消耗的办法。Tachyon [Li2014] 提供了一个基于内存的分布式数据存储系统，主要用于存放不同应用 (jobs) 产生的重用数据。用户可以将原本直接缓存到框架的数据转移存放到 Tachyon 上，以降低框架的内存消耗。FAÇADE [Nguyen2015] 提供了用于降低用户代码内存消耗的用户代码编译和执行环境。FAÇADE 设计目的是将数据存储和数据操作分开，方法是将数据存放到 JVM 的堆外内存，将对堆内对象的数据操作转换为对 FAÇADE 的函数调用。对于 Java 对象本身产生的 overhead（也就是 Java 对象自身所需的 header 和 reference），Bu 等人 [Bu2013] 提出了两种减少 overhead 的方法，比如将大量数据对象 (record object) 合并少量的大对象。然而，他们没有研究内存溢出错误的诊断问题。

C. 内存泄漏错误分析

内存泄漏可以直接导致内存溢出错误，所以之前的研究者提出了很多可以检测内存溢出错误的方法和工具。方法主要有静态检测方法 [Cherem2007, Xie2005] 和动态检测方法 [Jump2007, Xu2008]。内存泄漏的含义是用户忘记释放本应该释放的内存对象。在我们研究的应用中，我们还没有发现内存泄漏错误。一方面原因是用户代码是由具有垃圾回收机制的语言（Java 或 Scala）写的。另一方面的原因是我们很难去判定用户代码持有的大数据和中间计算结果是被该释放还是不该释放。

3.8 本章小结

本章详细研究分析了 123 个真实的 Hadoop 和 Spark 的应用的内存溢出的错误原因和修复方法。我们发现内存溢出的三大错误原因是：内存使用密集的用户代码，数据流异常和框架暂存的数据量过大。我们也从 42 个包含修复信息的错误中总结出了常用的修复方法。另外，我们提供了潜在的可以提升框架错误容忍能力和辅助错误诊断的三种方法。本章的研究结果不仅可以帮助用户和框架设计者更好地处理内存溢出问题，而且可以帮助以后的研究者发现新的问题。

第四章 应用内存用量模型及用量估算方法

本章主要讨论如何构建应用内存用量模型，也就是如何建立应用静态因素与 task 动态内存用量之间的关系。在此基础上，讨论如何在给定新应用后，预测应用的内存用量。

4.1 概述

分布式数据并行框架的最大优势是：即使没有任何并行或分布式系统开发经验的用户也能开发出可扩展的数据密集型应用。然而，框架的缺点也很明显，即不提供性能模型及性能估算工具。用户在提交应用到框架之前需要自己去配置应用的资源需求量，特别是内存需求量。

问题是用户不知道要配置多大的内存空间，因为内存用量是运行时决定的，而且用户通常对框架的分布式执行原理及流程并不清楚。因此，用户不知道怎么去分析内存用量，更不知道如何去估算和配置应用的内存需求量。

基于以上原因，用户一般根据自己的经验或“想像”去配置内存大小，但这会带来两方面的问题：(1) 如果内存配置过大，会导致内存资源浪费，进而降低集群的并行度（同时能够运行的 task 数目）。比如任务资源管理与调度框架 YARN [Vavilapalli2013] 和 Mesos [Hindman2011] 都需要用户事先指定内存用量需求，这样 YARN 和 Mesos 才能为 task 寻找具有相应资源的节点，将 task 调度到该节点运行。(2) 如果内存设置的过小，可能会导致 task 出现内存溢出错误。

为了帮助用户分析、估算应用的内存用量，我们以 MapReduce 框架为基础，研究了 MapReduce 应用的内存用量模型构建方法，并提供从抽样数据集上运行应用来估算应用在大数据上的内存用量的方法。

为了解决内存用量估计问题，我们主要解决三个子问题：**内存用量建模**，**用户代码空间复杂度分析**和**模型参数估计**。

我们观察发现静态配置参数影响分布式数据流，而分布式数据流加上用户代码影响内存使用量。基于此我们提出了一种以数据流为中心的内存用量模型，该模型可以量化“静态配置—数据流—内存用量”的映射关系。构建该模型最主要的挑战点是在用户代码未知的情况下，如何去量化用户代码内存用量和输入数据量之间的关系。在分析了用户代码模版后，我们发现用户对象有固定的生命周期，而不同生命周期的用户对象受输入数据中不同部分的影响。据此，我们提出了一个生命周期敏感的内存用量监控方法以及一个分段累积函数来量化用户对象大小和输入数据量之间的关系。因为我们的模型里面有一些参数（数据流参数和用户代码空间复杂度参数），我们通过在小数据集上预先

运行应用来得到参数，然后利用统计方法（比如线性回归）来得到用户代码的空间复杂度，最后估算应用在大数据集上运行时的参数。整个方法归属于白盒方法。我们的假设是在小数据集（比如 1GB）上运行时的中间数据量与在大数据集（比如 10GB）上运行产生的中间数据量之间存在线性关系。

在五个有代表性的 MapReduce 应用上实验评测发现，在给定一个 MapReduce 应用<数据，配置，用户代码>后，我们的方法可以估算出应用在大数据集上运行时的内存用量。评价结果显示我们的估算方法的平均误差率在 20%以内。

4.2 问题定义

本章主要针对 MapReduce 应用，也就是 job (data, configurations, user code)，研究其内存用量模型及内存用量估算问题：

- (1) 如何建立 task 的内存用量与 job 静态因素 (data, configurations, user code) 之间的关系，也就是 $\text{MemoryUsage}(\text{task}) = f(\text{data}, \text{configurations}, \text{user code})$ ？
- (2) 给定具体的 job (data, configurations, user code)，如何估算其 task 的内存用量，也就是 $\text{MemoryUsage}(\text{task})$ ？

其中，task 分为 map task (mapper) 和 reduce task (reducer)，我们需要分别估算 mapper 和 reducer 的内存用量。因为 mapper 和 reducer 有多个，且要配置的内存需求量与 task 的最大内存用量相关，因此我们要估算的是 mapper 和 reducer 的最大内存用量，也就是下面的 U_m 和 U_r 。

$$\begin{aligned} U_m &= \max(U(\text{mapper}_i)), 1 \leq i \leq m \\ U_r &= \max(U(\text{reducer}_i)), 1 \leq i \leq r \end{aligned}$$

m 是 mapper 的个数， r 是 reducer 的个数（也就是 reducer number）。

4.3 挑战

应用内存用量模型的建立与用量估算有两个研究挑战：(1) 内存用量受多个静态配置参数的影响，而且不是直接影响。比如 *input split size* 会影响每个 map task 的输入数据大小，*partition function* 定义 *map()* 如何对输出的 $\langle k, v \rangle$ records 进行划分。*Partition number* 定义会有多少个数据分块 (partition) 产生，也就是会有多少个 reducer 运行。*Buffer size* 定义框架的缓冲区大小。(2) 用户代码可以被任意书写，也可以由高层语言或高层库（比如 SQL-like 的 Pig 和 Hive 脚本）自动产生。这个特性使得我们需要将用户代码当作黑盒来看待。

当前 MapReduce 应用性能方面的研究工作集中于 MapReduce 应用的执行时间估算。

然而，执行时间的估算方法很难被用于内存用量的估算。对于执行时间估算来说，基本假设是用户代码的执行时间与其处理的数据量存在线性关系，而对于内存用量估算来说内存用量与处理的数据量不一定存在线性关系。

4.4 问题分析与模型建立思路

在 MapReduce 框架中，每一个 task 是一个独立进程（比如在 Hadoop 中，每个 map/reduce task 是一个独立的 JVM 进程）。如图 10 所示，map()先把数据输出到缓冲区，然后当缓冲区要满的时候将数据 spill 到磁盘。用户代码产生的中间计算结果的大小会随着时间变化。在应用（job）运行之前，我们不知道多少数据要被缓存到内存中，也不知道用户代码会产生多少中间计算结果。

上一章内存溢出错误的研究结果表明，task 的内存用量与静态配置、数据流和用户代码都相关。更具体地，task 的内存消耗主要包含两种内存对象：

- (1) 框架暂存到内存中的中间数据（称之为数据对象，buffered data）
- (2) 用户代码产生的中间计算结果（称之为用户对象，user objects）

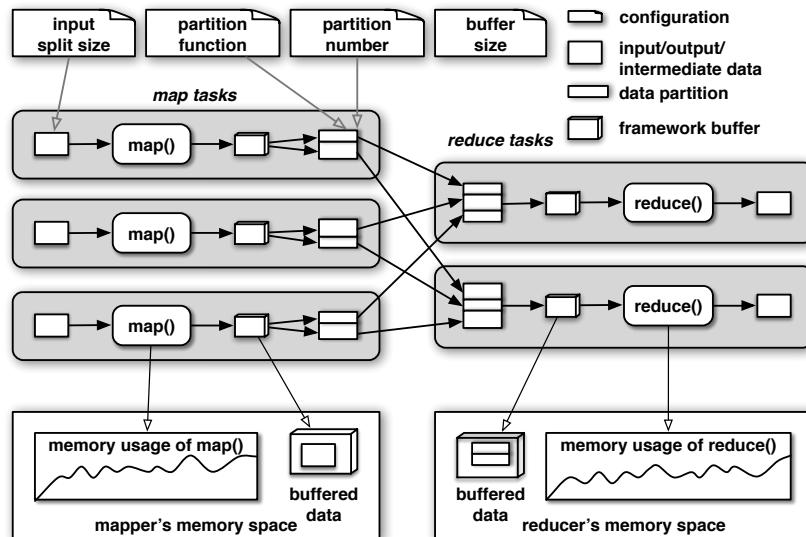


图 14 MapReduce job 的内存使用情况

A. 分析对象的来源与对象大小影响因素

框架暂存的中间数据量与运行时的中间数据量有关。运行时的中间数据包括 map()的输出数据与 reduce()的输入数据（通过 shuffle 得到）。map()的输出数据量又与其输入数据量有关，中间数据量又与 *partition number*, *partition function* 等配置参数有关。最后，在内存中暂存的中间数据量又与框架缓冲区大小有关。因此，框架暂存的中间数据的内存用量 $U(\text{buffered data})$ 可以简单地用下面的公式表示：

$$U(\text{buffered data}) = f(\text{runtime data, configurations})$$

用户代码产生的中间计算结果与运行时用户代码的输入数据量和代码空间复杂度有关。因此，用户对象的内存用量 $U(\text{user objects})$ 可以简单地用下面的公式表示：

$$U(\text{user objects}) = f(\text{runtime input data, space complexity of user code})$$

B. 内存用量模型建立思路

通过上述分析可以发现，buffered data 和 user objects 这两种对象都与运行时的中间数据（也就是数据流）有关。Buffered data 又与静态配置相关，而 user objects 又与用户代码有关。所以，我们建立模型的思路是以数据流为中心建立“静态配置—数据流—内存用量”的定量关系，具体要解决两个子问题：

- (1) 如何建立“静态配置—数据流”的关系？

解决思路是通过量化 MapReduce 基本处理步骤中的基本数据依赖关系来建立数据流模型。MapReduce 数据流主要包含三个基本的数据处理步骤（map, groupByKey 和 reduce）及两种基本的依赖关系（partition 和 aggregation）。我们可以通过量化这三个基本数据处理步骤来建立数据流模型，相关的配置参数如 *input split size*, *partition number* 等可以直接被包含到模型中。

- (2) 如何建立“数据流—内存用量”的关系？

解决思路是分别建立框架内存对象模型和用户代码对象模型。框架内存对象模型负责量化“中间数据与框架暂存的中间数据对象”之间的关系，而用户代码对象模型负责量化“代码输入数据与用户对象”之间的关系。主要挑战是如何在用户代码是黑盒的情况下，量化用户对象与输入数据之间的关系。我们观察发现：定义在用户代码不同位置的对象具有固定但不同的生命周期，而不同生命周期的用户对象受输入数据中的不同部分影响。基于这个特征，我们提出一个生命周期敏感的内存监控策略（lifecycle-aware memory monitoring strategy）来还原具有不同生命周期的用户对象的大小历史变化信息，进而通过建立分段累积函数来量化对象大小和输入数据量之间的关系。

C. 内存用量估算思路

内存用量模型只是建立了“静态配置—数据流—内存用量”的定量关系，模型中存在诸多参数（如数据流参数和用户代码空间复杂度参数）。在未运行应用之前我们无法得到这些参数，也就无法估算内存用量。我们的基本思路是可以通过在小数据集上预先运行应用来得到相关参数，然后估算应用在大数据集上执行时的参数。对于数据流参数，我们的基本假设是应用在小数据集上运行时产生的中间数据量与在大数据集上运行产生的中间数据量之间存在线性关系。对于用户代码空间复杂度参数，我们的基本假设是

用户代码运行在小数据和大数据集上时具有相同的空间复杂度。在获得这些参数后，就可以利用内存用量模型来估算 task 的内存用量。

D. 实验评价思路

给定未运行的应用 job (data, configurations, user code) 后，首先对 data 进行随机抽样（比如从 100GB 的 data 中随机抽取出 1GB 的 sample data），然后在 sample data 上运行 job，获取数据流与用户代码内存使用信息，从这些信息中抽取并估算出内存用量模型中的参数，然后将参数带入模型，得出 job 在原始 data 上的内存用量。最后在原始 data 上实际运行 job，对比估算的内存用量与实际内存用量之间的误差。

4.5 内存用量模型

根据上一节的建模思路描述，内存用量模型主要包含三部分：

- (1) 数据流模型定量描述运行时的输入/输出/中间数据关系。
- (2) 框架内存对象模型定量描述“中间数据与框架暂存的数据对象”的关系。
- (3) 用户代码对象模型定量描述“代码输入数据与用户对象”的关系。

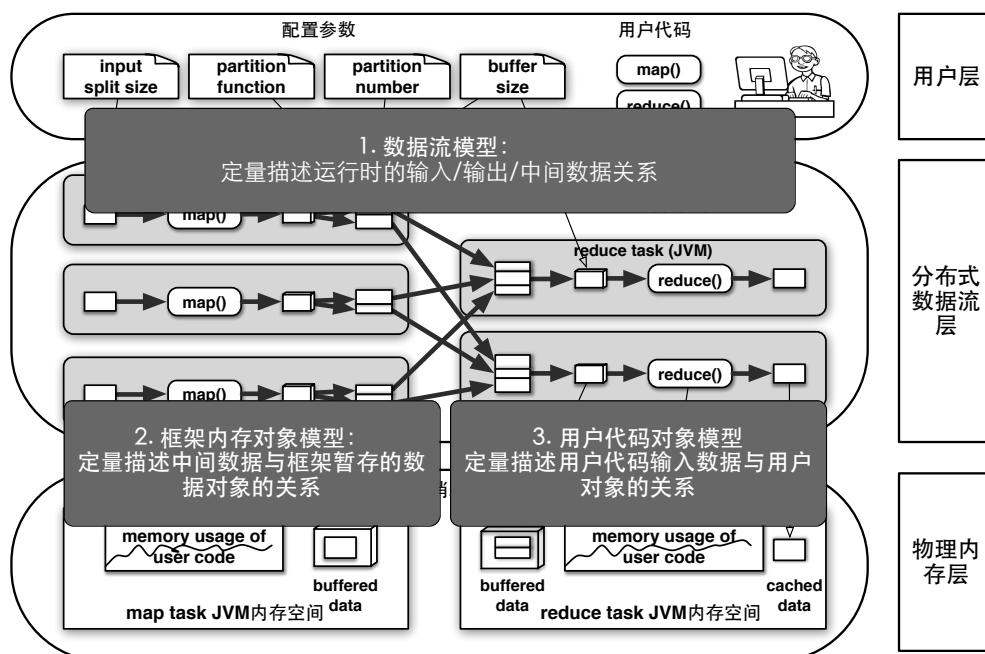


图 15 应用内存用量模型的构成

4.5.1 数据流模型

A. 具体模型

数据流模型根据 MapReduce 固定的数据依赖关系构造。虽然 MapReduce 数据流包

含多个处理步骤和数据依赖关系，MapReduce 数据流可以分解为三个基本处理步骤（`map()`, `groupByKey()`, 和 `reduce()`）及两个基本的数据依赖关系（partition 和 aggregation），如图 12 所示

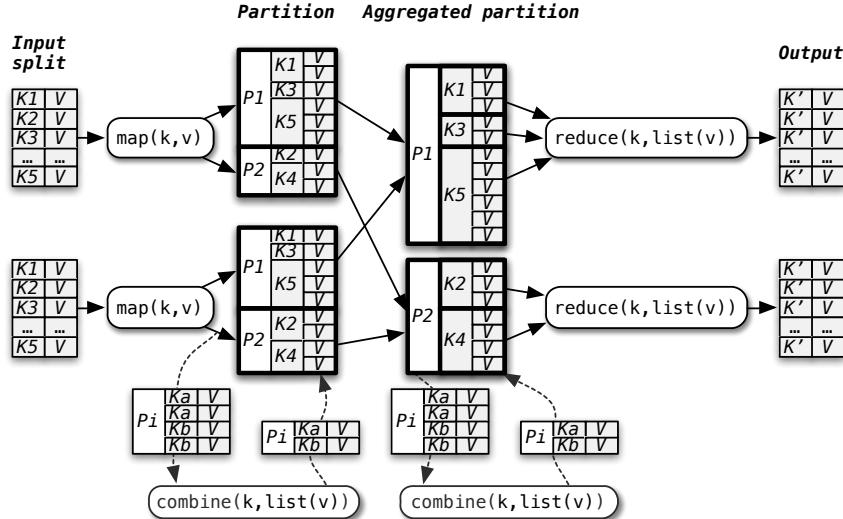


图 16 数据流模型

根据图 12 中的数据依赖关系，我们可以量化这三个基本步骤，如下表 5 所示。其中， $InputSplit_i(k, v)$ 表示第 i 个 input split 中的 $\langle k, v \rangle$ records 数目，第 i 个 input split 由第 i 个 mapper 进行处理。 $P_{ip}(k, v)$ 表示 mapper i 输出的第 p 个 partition 中的 $\langle k, v \rangle$ records 数目。根据 partition-aggregation 关系，在 aggregated partition P_i 中的 $\langle k, v \rangle$ records 数目可以由 $\sum_{m=1}^{N_m} P_{mi}(k, v)$ 表示，这里 N_m 代表 mapper 的个数。经过 `groupByKey()` 后，在 aggregated partition P_i 中的 records 个数可以由 $\sum_{g=1}^{N_k} G_{ig}(k, list(v))$ 表示，这里 $G_{ig}(k, list(v))$ 代表 P_i 中第 g 个 $\langle k, list(v) \rangle$ group 中的 records 个数， N_k 表示该 partition 中的 group 个数（也就是不同的 key 的个数）。最后的 $Record_i(k, v)$ 表示 reducer i 输出的 records 个数。

表 5 数据流模型

基本的数据处理步骤	输入数据	输出数据
$map_i()$	$InputSplit_i(k, v)$	$\sum_{p=1}^{N_p} P_{ip}(k, v)$
$groupByKey_i()$	$\sum_{m=1}^{N_m} P_{mi}(k, v)$	$\sum_{g=1}^{N_k} G_{ig}(k, list(v))$
$reduce_i()$	$\sum_{g=1}^{N_k} G_{ig}(k, list(v))$	$Record_i(k, v)$

数据流相关的配置参数被直接包含在数据流模型中。比如 $InputSplit_i(k, v)$ 的总大小与配置参数 `input split size` 相同。 N_p 等于 `partition number`。Partition function 没有被直

接包含在模型中，但我们可以通过计算各个 partition 中 records 的数目比值来模拟 *partition function*。

B. 参数估计

步骤 1：首先去获取应用在小数据集上运行时的数据流参数。

获取数据流参数的方法是：从我们提升过的 task 的 log 日志和数据流计数器 (dataflow counters) 中获取。我们提升了 Hadoop MapReduce 的日志功能，使其 task 在执行每个处理步骤时将详细的输入/输出数据信息打印出来，如表 6 所示。我们可以从 log 中获取到 map(), combine(), reduce() 执行完成时的输入/输出数据量，map() 输出的每个 partition 中包含的 records 个数，以及 reducer 从每个 mapper 那里 shuffle 得到的 records 个数。另外，Hadoop MapReduce 自身提供了数据流计数器，能够实时记录用户代码当前已经输入/输出的 record/group 数目。

表 6 提升过的 task 的日志

Mapper 1	[map() starts]
	[map () ends] InputRecords = 17,535, OutputRecords = 17,525
	[map output][Partition 1] Records = 8,653, Keys = 3,943
	[map output][Partition 2] Records = 8,559, Keys = 3,715
Reducer 1	[shuffling] Partition 1 (Records = 8,653, Keys = 3,943) from mapper 1
	[shuffling] Partition 1 (Records = 8,882, Keys = 4,036) from mapper 2
	[shuffling] Partition 1 (Records = 9,016, Keys = 4,087) from mapper 3
	[combine() starts] to merge the buffered partitions from mapper (1, 2)
	[combine() ends] OutputRecords = 7,979, Keys = 7,979
	[reduce() starts] InputRecords = 16,995, Keys = 12,066

在进行参数估计前，我们需要获取或计算表 7 中的参数。

表 7 要获取或计算的参数

要获取或计算的参数	计算公式	Average	Max
$map_i()$ 输出输入比 r_i	$\frac{map\ output\ records_i}{map\ input\ records_i}$	$\frac{1}{N_m} \sum_{i=1}^{N_m} r_i$	$\max_{1 \leq i \leq N_m} (r_i)$
Aggregated partition P_i 中的 records 数 $AP_i(k, v)$	$\sum_{m=1}^{N_m} P_{mi}(k, v)$	$\frac{1}{N_p} \sum_{i=1}^{N_p} AP_i(k, v)$	$\max_{1 \leq i \leq N_p} (AP_i(k, v))$
Group i 中的 records 数目 $G_i(k, v)$	$G_i(k, v)$	$\frac{1}{N_g} \sum_{g=1}^{N_g} G_i(k, v)$	$\max_{1 \leq i \leq N_g} (G_i(k, v))$

* $map\ input\ records_i$ 和 $map\ output\ records_i$ 可以通过增强后的 log 获得， $P_{mi}(k, v)$ 和 $G_i(k, v)$ 可以通过 dataflow counters 获得。

步骤 2：估算应用在大数据集上运行时的数据流参数。

获取到小数据集上的数据流参数后，下一步是去估算应用在大数据上运行时的数据流参数，也就是表 8 中的参数：

表 8 需要估计的数据流参数

基本步骤	EstimatedInput	EstimatedOutput
$map_i()$	$NewInputSplit_i(k, v)$	$\sum_{p=1}^{newN_p} NewP_{ip}(k, v)$
$groupByKey_i()$	$\sum_{m=1}^{newN_m} NewP_{mi}(k, v)$	$\sum_{g=1}^{newN_k} NewG_{ig}(k, list(v))$
$reduce_i()$	$\sum_{g=1}^{newN_k} NewG_{ig}(k, list(v))$	$NewRecord_i(k, v)$

由于应用在大数据集上运行时其配置参数（尤其是 *partition number*）可能改变，也可能会出现数据倾斜（data skew）。我们根据应用在小数据集上运行时表现出的数据流特征（如 partition 中 records 的平均和最大个数）来推断其在大数据集上的数据流参数。具体方法是采用平均值 + 最大值的方法来估算在大数据集上的数据流参数，估算公式如下：

估算 Parameter 1: map() 的输入数据（平均值与最大值）：

$$avg(NewInputSplit_i(k, v)) = \frac{New input split size}{input split size} * avg_{1 \leq i \leq N_m}(map input records_i)$$

$$\max(NewInputSplit_i(k, v)) = \frac{New input split size}{input split size} * \max_{1 \leq i \leq N_m}(map input records_i)$$

估算 Parameter 2: map() 的输出的 partition 大小（平均值与最大值）：

$$avg(NewP_{ip}(k, v)) = \frac{NewInputSplit(k, v) * avg_{1 \leq i \leq N_m}(r_i)}{newN_p}$$

$$\max(NewP_{ip}(k, v)) = \frac{NewInputSplit(k, v) * \max_{1 \leq i \leq N_m}(r_i)}{newN_p}$$

估算 Parameter 3: mapper 个数：

$$newN_m = \frac{Size(data)}{New input split size}$$

估算 Parameter 4: reduce() 读入的 $< k, list(v) >$ group 大小（平均值与最大值）：

$$avg(NewG_{ig}(k, list(v))) = \frac{Size(data)}{sampleData} * avg(G_i(k, v))$$

$$\max(NewG_{ig}(k, list(v))) = \frac{\text{Size}(data)}{\text{sampleData}} * \max(G_i(k, v))$$

估算 Parameter 5: Aggregated partition P_i 中包含的 $\langle k, list(v) \rangle$ group 个数 (平均值与最大值):

$$\text{avg}(newN_k) = \frac{\sum_{m=1}^{newN_m} \text{avg}(NewP_{mi}(k, v))}{\text{avg}(NewG_{ig}(k, list(v)))}$$

$$\max(newN_k) = \frac{\sum_{m=1}^{newN_m} \max(NewP_{mi}(k, v))}{\text{avg}(NewG_{ig}(k, list(v)))}$$

得到这些估算的参数，我们可以构造应用在大数据集上运行时的数据流。

4.5.2 框架内存对象模型

框架内存对象包含两部分：框架缓冲区(ramework buffer)，缓存的中间数据(buffered data)。下面我们就以 Hadoop MapReduce 为例对这两种框架对象进行分析与建模。

A. 框架缓冲区 (framework buffer)

对象定义：框架缓冲区包含固定 buffer 和虚拟 buffer。固定 buffer 直接占据固定大小的内存空间。虚拟 buffer 只是一个阈值，用来限定框架最多可以暂存多少中间数据到内存中。

在 Hadoop 中的情况：在 Hadoop 中，每个 mapper 分配一个固定 buffer (称为 map buffer) 来缓存 map()输出的 key/value records。等到 buffer 快满的时候，mapper 会将缓存数据排序，然后将排序后的数据 spill 到磁盘上的不同的数据分块 (partition) 中去。每个 reducer 在 shuffle 阶段会分配一个虚拟 buffer (shuffle buffer) 来缓存从 mapper 那里 shuffle 过来的数据，当虚拟 buffer 被填满的时候，buffer 中的数据会被排序、聚合、输出到磁盘上。用户也可以在 reduce 阶段分配一个虚拟 buffer (称为 reduce buffer) 来暂存 shuffle 过来但没有被 spill 到磁盘的中间数据。

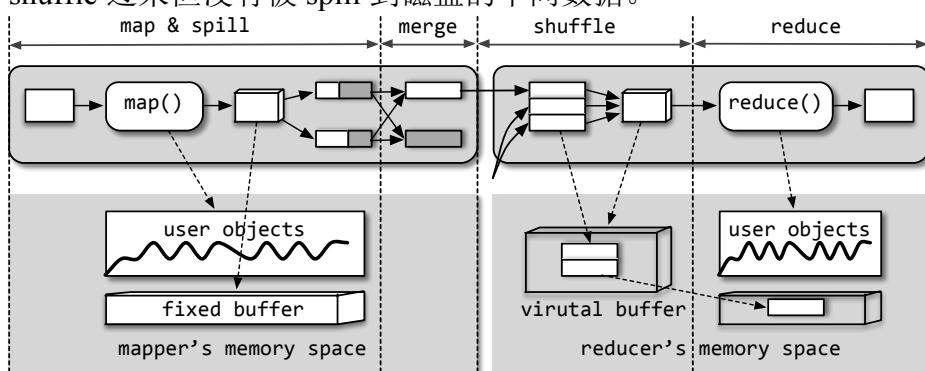


图 17 框架内存使用情况

对象生命周期：缓冲区并不是一直都存在。以 Hadoop 为例，固定缓冲区 map buffer 只在 map&spill 阶段存在，在 merge 阶段被回收。虚拟缓冲区 shuffle buffer 在 shuffle 阶段存在，reduce buffer 在 reduce 阶段存在。

大小影响因素：缓冲区大小由用户直接定义，比如配置参数 $io.sort.mb=300$ 即定义 map buffer 为 300MB， $mapred.job.shuffle.input.buffer.percent=0.7$ 定义 shuffle buffer 占 reducer 中 70% 的内存空间。虚拟 buffer 实际会消耗多少内存还要看 buffer 缓冲了多少中间数据。

具体模型： $U(buffer) = f(configurations)$ ，具体函数见表 9。

表 9 框架对象模型（框架缓冲区）

阶段	Buffer 名称	类型	$U(buffer) = f(configurations)$
Map	map buffer	固定 buffer	$io.sort.mb$
Shuffle	shuffle buffer	虚拟 buffer	$HeapSize * mapred.job.shuffle.input.buffer.percent$
Reduce	reduce buffer	虚拟 buffer	$HeapSize * mapred.job.reduce.input.buffer.percent$

B. 暂存到内存中的中间数据（buffered data）

对象定义：暂存到内存中的中间数据指的是暂存到虚拟 buffer 中的数据。

在 Hadoop 中的情况：在 shuffle 阶段，当虚拟 buffer 被填满的时候，buffer 中的数据会被排序、聚合、输出到磁盘上。如果配置了 reduce buffer，那么在 shuffle 阶段未被 merge 到磁盘的中间数据会被存放到 reduce buffer 中。

生命周期：尽管存放在 buffer 中的数据不定时会被 spill 到磁盘，但从内存连续占用情况来看，存放到内存中的中间数据与虚拟 buffer 的生命周期相同。

大小影响因素：暂存在虚拟 buffer 的数据与用户配置的虚拟 buffer 大小和 shuffle 得到的数据量有关。存放在 shuffle buffer 中的数据量直接由 shuffle buffer size 和 shuffled data（也就是从所有 mapper 那里 shuffle 过来的 partitions）的数据量决定，是两者中的较小者。经过 shuffle 阶段后，没有被 merge 到磁盘上的数据量为 $Size(shuffled data) \% shuffle buffer size$ ，而存放到 reduce buffer 中的数据量还受到 reduce buffer 限制，因此最后在 reduce buffer 的数据量大小是两者的较小值。

具体模型：具体函数见表 10。

$$U(in-memory intermediate data) = f(configurations, intermediate data)$$

表 10 框架对象模型（内存中暂存的中间数据）

阶段	内存中的中间数据	$U(in-memory data) = f(configurations, intermediate data)$
Shuffle	Data in shuffle buffer	$\min(shuffle buffer size, shuffled data)^*$
Reduce	Data in reduce buffer	$\min(reduce buffer size, shuffled data \% shuffle buffer size)$

* $shuffled data = aggregated partition P_i = \sum_{m=1}^{newN_m} NewP_{mi}(k, v)$

4.5.3 用户代码对象模型

由于用户代码可以被任意书写也可以由高层语言或高层库自动生成，用户代码对我们来说相当于一个黑盒，我们很难用静态分析方法去确定其空间复杂度。

然而，我们观察发现 MapReduce 用户代码有一个重要特点，可以帮助我们使用动态模拟+统计的方法去确定用户代码的空间复杂度。这个特点就是 streaming-style，也就是用户代码会一个个读入、处理、并输出 key/value records。这个特点使得在用户代码不同位置定义的用户对象拥有不同的生命周期，而拥有不同生命周期的对象大小由用户代码输入数据的不同部分决定。根据这个特点，我们设计了一个生命周期敏感的内存监控策略（lifecycle-aware memory monitoring strategy）用来还原具有不同生命周期的用户对象大小历史变化信息，并在此信息基础上建立分段累积函数来量化对象大小和输入数据量之间的关系。

A. 用户对象的生命周期

MapReduce 编程范型的最大特点是用户以 streaming-style 来处理输入的 key/value records。这样，当下一个 input record 被用户代码读入时，当前 record 及其相关的中间处理结果会从内存中清除，除非它们被用户故意缓存到内存中用于后面的 records 的处理。这个特性使得用户对象（user objects）具有固定的生命周期，下面通过分析用户代码模版来确定用户对象的固定生命周期及其相关的输入数据范围。

用户代码模版：

用户代码模版来自现实世界的 Google/Hadoop MapReduce 例子，以及经典书籍 MapReduce Design Pattern [MRDesignPattern] 中的例子。

map()产生的用户对象生命周期：下面的代码模版显示了 map() 的代码结构及其内部的用户对象。map()的编程范型很简单，map(K, V) 每读入一个 $\langle k, v \rangle$ record 就处理一个，所以每新来一个 record，map(K, V)就会被调用一次。这样，map()中的用户对象生命周期分两种：(1) map-level (用①标示)。定义在 map(K,V)之外的的对象，如 mapLevelBuffer，会一直在内存中，直到所有的 records 被处理完才会被回收。(2) record-level (用②标示)。定义在 map(K,V)里面或在 process()中产生的用户对象（比如 iResults）只在当前 record 被处理时存在，当下一个 record 被用户代码读入时，当前的 iResults 会被清除，除非 iResults 被用户 cache 到 map-level buffer 里面。

```

public class Mapper {
    private Object mapLevelBuffer; // ①
    public void map(K key, V value) {
        // record-level 中间计算结果，可以被存放在 mapLevelBuffer
        Object iResults = process(key, value); // ②
        emit(newKey, newValue); // using iResults
    }
}

```

因此，如图 14 所示，record-level 用户对象与当前被处理的单个 record 有关，而 map-level 的用户对象与 map()读入的全部 records 有关。

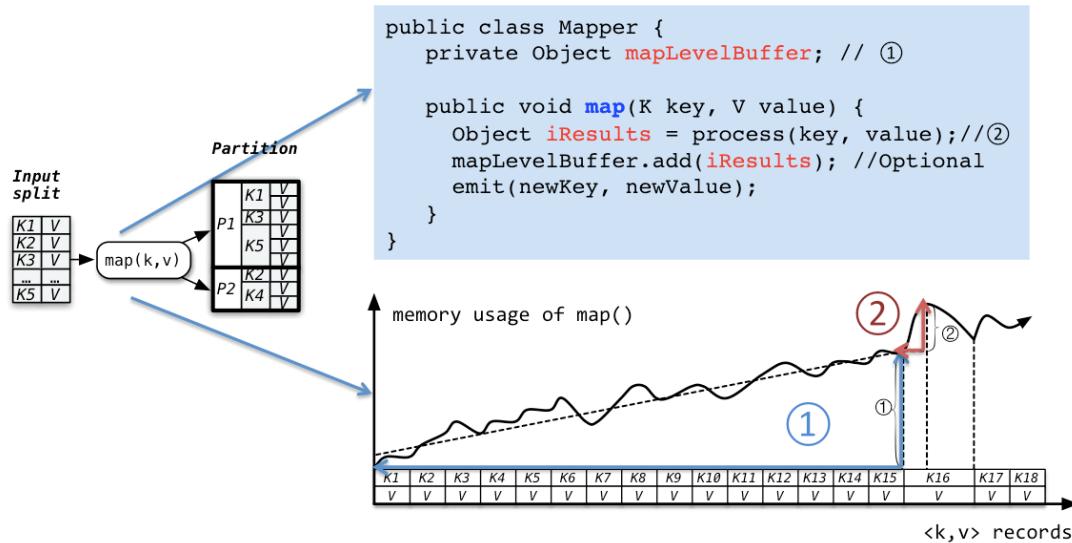


图 18 Mapper 的用户代码模型

reduce()产生的用户对象生命周期：下面代码显示了 reduce() 的代码结构及其内部的用户对象。reduce()的编程范型略复杂。reduce(K,list(V))负责处理 $\langle k, list(v) \rangle$ groups，每新来一个 $\langle k, list(v) \rangle$ group，reduce(K, list(V))就被调用一次， $\langle k, list(v) \rangle$ 中的 values 被一个个读入并处理。reduce(K, list(V))中的用户对象生命周期有三种：(1) reduce-level (用③标示)。定义在 reduce(K, list(V))之外的对象，如 reduceLevelBuffer，会在内存中一直存在，直到所有的 $\langle k, list(v) \rangle$ groups 被处理完。所以 reduce-level 的对象大小与所有的 $\langle k, list(v) \rangle$ groups 有关。(2) group-level (用①标示)。定义在 reduce(K, list(V))里面，但在 while()外面的对象会在当前 $\langle k, list(v) \rangle$ group 被处理的时候一直存在，直到当前的 $\langle k, list(v) \rangle$ group 被处理完后回收。所以 group-level 的用户对象与当前 $\langle k, list(v) \rangle$ group 中的所有 records 相关。(3) record-level (用②标示)。定义在 while()内部的用户对象与当前被处理的 record 有关，当下一个 record (也就是下一个 value) 被读入前会被回收，除

非被 cache 到高层(group-level 或 reduce-level)的 buffer 里面。Group-level 和 record-level 用户对象的关系与 map()里面 map-level 与 record-level 用户对象关系类似。因为 combine() 和 reduce() 的编程范型相同，所以 combine() 中用户对象的生命周期与 reduce() 的情况相同。

```
public class Reducer {
    private Object reduceLevelBuffer; // ③

    public void reduce(K key, Iterable<V> values) {
        Object groupLevelBuffer; // ①
        while (values.hasNext()) { // 处理<k, list(v)>
            V value = values.next();
            // 中间计算结果，可以被 cache 到 group/reduce-level 的 buffer 里
            Object iResults = process(key, value); // ②
            emit(newKey, newValue); // may be here
        }
        emit(newKey, newValue); // may be here too
    }
}
```

因此，如图 15 所示，reduce-level 用户对象与 reduce() 的全部输入数据（也就是整个 aggregated partition）有关，group-level 用户对象与当前被处理的 $\langle k, list(v) \rangle$ group 有关，record-level 用户对象与当前被处理的 $\langle k, v \rangle$ record 有关。

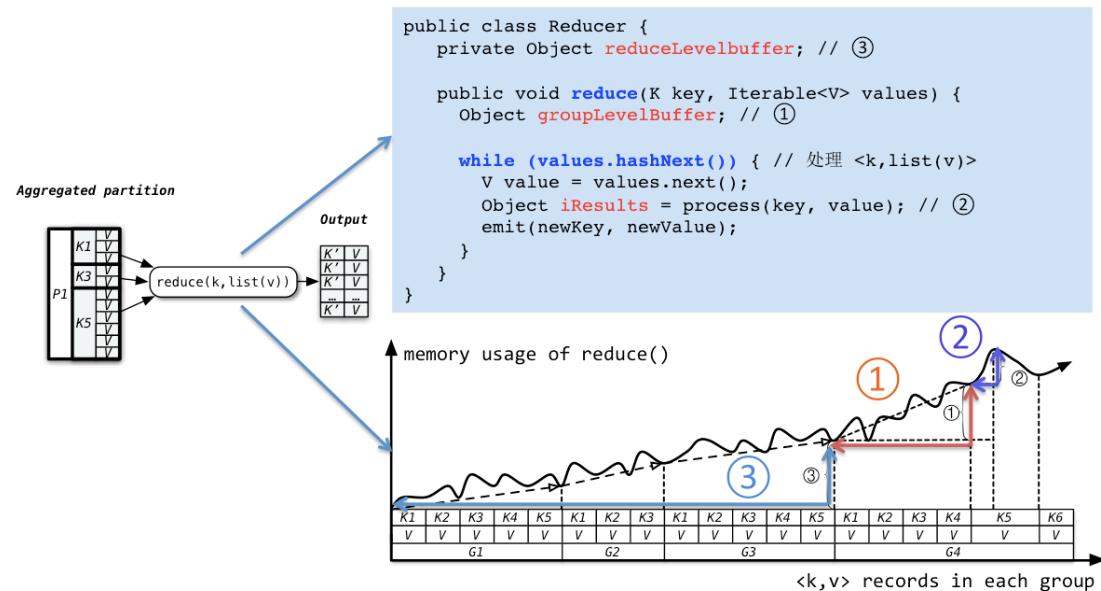


图 19 Reducer 的用户代码模型

B. 用户对象模型

针对 mapper 和 reducer 中不同生命周期的用户对象，我们依次建立对象与其相关数据的函数关系（见表 11）。对于 mapper，假设在 t 时刻， $\text{map}()$ 已经处理过的 records 个数为 map input records ，当前正在处理的是第 i 个 record。对于 reducer，假设在 t 时刻， $\text{reduce}()$ 已经处理过的 $\langle k, \text{list}(v) \rangle$ group 个数为 $\text{reduce input groups}$ ，正在处理第 g 个 $\langle k, \text{list}(v) \rangle$ group 中的第 i 个 record。

表 11 用户代码对象模型

阶段	对象类型	$U(\text{user objects}) = f(\text{input data})$
Mapper	map-level objects	$U(\text{map-level objects}) = f(\text{map input records})$
	record-level objects	$U(\text{record-level objects}) = h(\text{record } i)$
Reducer	reduce-level objects	$U(\text{reduce-level objects}) = f(\text{reduce input groups})$
	group-level objects	$U(\text{group-level objects}) = g(\text{records in group } g)$
	record-level objects	$U(\text{record-level objects}) = h(\text{record } i)$

我们使用线性 / 非线性回归（见下一节的参数估计）来估算具体的 f 。最后将各个生命周期的对象相加得到最终模型

$$U_m = U(\text{map-level objects}) + U(\text{record-level objects})$$

$$U_r = U(\text{reduce-level objects}) + U(\text{group-level objects}) + U(\text{record-level objects})$$

C. 生命周期敏感的内存用量监控策略

为了量化不同生命周期的用户对象与其对应的输入数据之间的关系（也就是计算 f, g, h ），我们需要在运行时获取用户对象的大小变化信息。

在 t 时刻，用户对象由 map/group/reduce-level user objects 和 record-level user objects 构成。根据上一节的生命周期分析，map/group/reduce-level user objects 是累积计算结果，与特定范围内的输入数据相关，因此需要在这些特定范围内监控对象的大小变化。而 record-level user objects 是中间计算结果，需要在当前 record 被处理时监控其变化。

在 t 时刻，获取 task 中的用户对象大小有两种方法：(1) 精确方法。对 task 进行 heap dump，分析 heap 中哪些对象被用户代码引用，然后计算这些用户对象的总大小，这种方法虽然计算精确但耗时较长（分析 heap 需要几秒到 1 分钟之间）。(2) 粗略方法。首先将缓冲区大小设为 0，这样 task 在执行时基本只有用户代码消耗内存。在 t 时刻，我们让 task 的 heap 进行一次 GC（垃圾回收，去除在内存中但已经不再被引用的对象），之后直接读取 heap usage 作为用户对象大小，这种方式耗时短（微秒级），但不适合监控在大数据集上运行的应用（需要将缓冲区设为 0）。由于我们只需监控应用运行在小

数据上的内存使用量，我们选择第二种方法来获取 t 时刻的用户对象大小。对于 map/group/reduce-level user objects 和 record-level user objects，我们分别设计了如下两种监控策略。

(1) 监控 map/group/reduce-level user objects 的大小变化

这些用户对象属于累积计算结果，我们的监控策略首先会在这些用户对象对应的输入数据范围内选取多个特殊的时间点，然后在这些时间点（这些时间点里 heap 中不包含 record-level objects）进行监控得到用户对象的大小变化。对于 map-level objects，可选的时间点是当用户代码刚刚处理完当前的 record (R_i)，正准备去处理下一个 record (R_{i+1})。因为在这些时间点 record-level user objects 已经被清除，所以当前 heap 中的用户对象大小即是 map-level user objects 大小。对于 reduce-level 用户对象来说，可选的时间点是用户刚刚处理完当前的 $\langle k, list(v) \rangle$ group (G_i)，正准备去处理下一个 $\langle k, list(v) \rangle$ group (G_{i+1})。为了提升效率，我们并不是在每个 R_i 和 G_i 处都进行监控（GC 会有微妙级的 overhead），而是隔一些 R_i 和 G_i 进行监控（如图 16 中的竖线所示）。对于 map-level user objects，我们选择每隔 n 个 records 进行监控（我们提供了配置参数来设置，默认是 10）。对于 reduce-level user objects，我们选择每隔 n 个 $\langle k, list(v) \rangle$ group 进行监控。对于 group-level user objects，其内存监控策略与 map-level user objects 的策略是一样的（即每隔 n 个 records 进行监控）。唯一区别是我们不知道 reduce() 会在处理哪个 group 时产生最大的 group-level user objects，因此监控策略会在多个 groups 里面监控 group-level user objects 的变化，最后取变化斜率的平均值和最大值。在得到所有监控时间点上的 heap usage 后，就可以作出 map/group/reduce-level user objects 的大小趋势线。

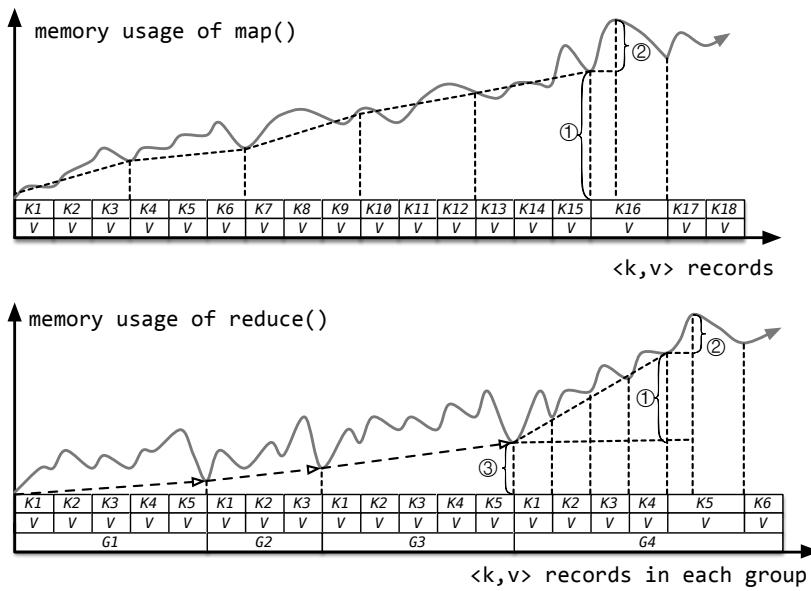


图 20 生命周期敏感的用户代码内存用量监控策略

(2) 监控 record-level user objects 的大小

因为 record-level user objects 与当前被处理的 record 有关，所以我们选择的监控时间点是：当 R_i 要被处理，和 R_i 正在被处理（即将要调用 $emit(key,value)$ 输出新的 $\langle k, v \rangle$ record）。这两个时间点的 heap usage 之差（比如上图中的②）可以被当作 record-level user objects 的大小。由于我们不确定用户代码会在处理哪个 record 时产生最大的 record-level user objects，我们选择在多个 R_i 处监控 heap usage 的变化，然后计算平均值和最大值。对于 map()，假设已选择在 R_i 处监控 map-level user objects，那么同时也会选择在 R_i 被处理时监控 record-level user objects。如图 16 中的第一个图所示，在 K_{16} 要被处理时监控 map-level user object 大小，同时也会在 K_{16} 被处理时监控 record-level user objects 大小。对于 reduce()，假设已选择在 G_i 中的 R_j 处监控 group-level user objects 的大小，那么同时也会在 R_j 被处理时监控 record-level user objects 的大小，最后对多点监控得到的 heap usage 差变化取平均值和最大值。

D. 参数估计

得到用户代码内存使用量趋势后，我们使用线性/非线性回归来得到用户对象与输入数据的具体函数关系 f 。比如，如果 map-level user objects 的内存使用曲线是线性的，我们可以用线性回归来计算出具体的函数关系 f ，而且可以推断出 map() 的空间复杂度是 $O(n)$ 。如果 map() 的内存使用曲线是二次的，那么我们可以使用多项式回归去计算出二次函数关系 f ，并推断出 map() 的空间复杂度是 $O(n^2)$ 。由于我们事先不知道趋势线是线性还是非线性的，我们只能尝试各种回归函数，然后选取最佳的回归函数（ ε 最小），表 12 显示了我们使用的回归函数与对应的空间复杂度 $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ 。

表 12 用户代码空间复杂度

回归函数 f	空间复杂度
$U = \beta_0 + \beta_1 \log(x) + \varepsilon$	$O(\log n)$
$U = \beta_0 + \beta_1 x + \varepsilon$	$O(n)$
$U = \beta_0 + \beta_1 x \log(x) + \varepsilon$	$O(n \log n)$
$U = \beta_0 + \beta_1 x + \beta_2 x^2 + \varepsilon$	$O(n^2)$
$U = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \varepsilon$	$O(n^3)$

其中， U 是用户对象大小， x 是被处理的 records 或 groups 数目（取决于用户对象是 map/group-level 还是 reduce-level）， β 是回归参数， ε 是回归误差。由于我们会对多个 mapper 的 map() 和 reducer 的 reduce() 进行监控，因此可以获取多组 map/group/reduce-level

用户对象的监控结果(每组包含多个散列点),也可以获得多个 record-level 用户对象的大小。对于前者,将同类型的用户对象的监控结果(散列点)放在一起进行回归,最终计算得到 f 和 g 。对于后者,我们取 record-level 用户对象大小的中位数作为其大小 $\text{avg}(\text{record-level objects})$ 。

最后,估计应用在大数据上运行时用户代码 map()的对象大小 U_m 及 reduce() 的对象大小 U_r ,如下:

$$\begin{aligned} U_m &= U(\text{map-level objects}) + U(\text{record-level objects}) \\ &= f(\text{input data of map}()) + \text{avg}(\text{record-level objects}) \\ &= f(\text{avg}(\text{NewInputSplit}_i(k, v))) + \text{avg}(\text{record-level objects}) \end{aligned}$$

$$\begin{aligned} U_r &= U(\text{reduce-level objects}) + U(\text{group-level objects}) + U(\text{record-level objects}) \\ &= f(\text{input data of reduce}()) + g(\text{input records in a group}) \\ &= f\left(\text{avg}\left(\sum_{g=1}^{\text{newN}_k} \text{NewG}_{ig}(k, \text{list}(v))\right)\right) + g\left(\text{avg}(\text{NewG}_{ig}(k, \text{list}(v)))\right) \\ &\quad + \text{avg}(\text{record-level objects}) \end{aligned}$$

4.5.4 模型总结

根据以上几节的描述,整个应用内存用量模型可以用表 13 表示:

表 13 内存用量模型总结

阶段	框架对象	用户对象
Map	$\text{map buffer} = f(\text{conf})$	$U_{\text{map}} = U(\text{map-level objects}) + U(\text{record-level objects})$
Shuffle	$\text{data in shuffle buffer} = f(\text{conf}, \text{shuffled data})$	$U_{\text{combine}} = U(\text{reduce-level objects}) + U(\text{group-level objects}) + U(\text{record-level objects})$
Reduce	$\text{data in reduce buffer} = f(\text{conf}, \text{shuffled data})$	$U_{\text{reduce}} = U(\text{reduce-level objects}) + U(\text{group-level objects}) + U(\text{record-level objects})$

4.6 实验评价

4.6.1 实验环境与实验数据

我们搭建了 10 个节点的 Hadoop 集群(1 个 master 节点,9 个 slave 节点),Hadoop 版本为 1.2。每个节点拥有 4 核 CPU,16GB 的内存,2TB 的硬盘。操作系统是 Ubuntu-11.04 x86_64, JDK 版本为 Hotspot 64-bit Server VM (build 1.6.0_27-b07)。每个 slave 节点配置 4 个 map slot(可以同时运行 4 个 mapper)和 2 个 reduce slot(可以同时运行 2 个 reducer)。

我们选取了 5 个常见的 MapReduce 应用来评测我们的内存用量模型和内存用量估算方法。这 5 个应用（见表 14）来自于 MapReduce Benchmark [BrownBench] 和论文 [ARIA2011, Verma2011] 中的应用，具体如下：

1. **WikiWordCount**: 该应用逻辑与 Google MapReduce 论文中的 WordCount 应用逻辑一样，都是统计文本中词出现的次数。源代码来源于 Hadoop 的 examples 包，数据来源于英文版 Wikipedia (enwiki-20110405-pages.articles.xml)，大小为 9.4GB。

2. **BuildInvertedIndex**: 这个应用逻辑是建立 Wikipedia 的倒排索引。对网页建立倒排索引是搜索引擎的日常任务。该应用源代码来源于马里兰大学开发的 Hadoop 套件 Cloud9 [Cloud9]，数据也是英文版的 Wikipedia，大小为 9.4GB。

3. **Pig-UserVisitAggr**: 这个应用由 Pig 脚本生成，目的是分析网站的用户访问日志，脚本来源于一个常用的 benchmark [BrownBench]。这个 pig 脚本主要包含一个 Group by 操作，数据集也是由该 benchmark 中的脚本生成，数据大小为 75GB。

4. **TwitterBiEdgeCount**: 这个应用逻辑是统计 Twitter 图中的双向边个数。Twitter 图的数据来源于 [Kwak2010]，一共包含 4 千万个节点和 15 亿条边。

5. **TeraSort**: 该应用逻辑是对大规模字符串进行排序，使用标准的 TeraSort 程序，源代码来源于 Hadoop 的 examples 包，数据也是有该包中的数据生成程序产生，总大小为 36GB。需要注意的是这个应用的用户代码并不包含任何有效的数据处理逻辑，map() 和 reduce() 只是将读入的 $\langle k, v \rangle$ records 直接输出。

给定 job (data, configurations, user code) 后，首先对 data 进行随机抽样（比如从 100GB 的 data 中随机抽取出 1GB 的 sample data），然后在 sample data 上运行 job，获取数据流与用户代码内存使用信息，从这些信息中抽取并估算出内存用量模型中的参数，然后将参数带入模型，得出 job 在原始 data 上的内存用量。最后在原始 data 上实际运行 job，对比估算的内存用量与实际内存用量之间的误差。

表 14 实验评价所用的 5 个典型 MapReduce 应用

MapReduce 应用	数据集大小	配置参数		
		Input split size (MB)	Partition (reduce) number	Buffer size (MB)
WikiWordCount	9.4 GB	64, 128, 256	9, 18, 36	200, 400
BuildInvertedIndex	9.4 GB	64, 128, 256	9, 18, 36	200, 400
Pig-UserVisitAggr	75 GB	64, 128, 256	9, 18, 36	200, 400
TwitterBiEdgeCount	24.4 GB	64, 128, 256	9, 18, 36	200, 400
TeraSort	36 GB	64, 128, 256	9, 18, 36	200, 400

4.6.2 实验步骤

对于每一个 MapReduce 应用，也就是 job <data, configurations, user code>，我们先在抽样数据（1GB 和 5GB）上运行应用，获取必要的信息，然后通过模型估算应用运行在大数据上的内存用量，具体步骤如下：

1. 对每个大数据集进行随机抽样（使用一个 MapReduce job），每个数据集分别抽出 1GB 和 5GB 作为抽样数据。
2. 选取一组配置参数（实验中使用 $input\ split\ size = 128MB$, $partition\ number = 9$, $buffer\ size = 200MB$ ）后，在小数据集上运行应用，获得应用运行时的数据流信息和用户代码内存用量信息。
3. 建立应用在大数据集上的内存用量模型：数据流模型，框架内存对象模型和用户代码对象模型。
4. 对于应用的每一组配置参数（每个应用有 $3*3*2=18$ 组不同的配置参数），根据模型提供的参数估计方法和小数据集上的运行信息估算模型参数，得到应用在该配置下运行在大数据集上时的内存用量。
5. 对于应用的每一组配置参数，我们在大数据集上实际运行该应用，获取真实的 map/reduce task 的内存用量。
6. 计算估算相对误差，误差计算方法如下：
7. 估算相对误差 = $\left| \frac{\text{估算得到的内存用量} - \text{实际内存用量}}{\text{实际内存用量}} \right| \times 100\%$

4.6.3 实验结果

每个应用在每个抽样数据上有 18 组估算结果（18 个相对误差），为了显示方便，我们计算了这 18 个相对误差的平均值与标准差。图 17 显示了使用 1GB 抽样数据的估算结果，图 18 显示了使用 5GB 抽样数据的估算结果。使用 1GB 抽样数据时，估算结果的平均相对误差在 20% 以内，标准差在 4% 以内。误差来源主要是抽样比例，当把抽样数据提高到 5GB 时，估算结果的平均相对误差在 13% 以内，标准差在 3% 以内。误差第二个来源是数据倾斜，比如 WikiWordCount 和 BuildInvertedIndex 应用处理的数据是 Wikipedia，而 Wikipedia 中某些常用词 word（如 *and, of* 等），出现的频率要远高于其他词，因此会导致最终估算结果有偏差。TwitterBiEdgeCount 也是如此，该应用处理的是 Twitter 中的用户关系图，某些节点连接的边个数要远远大于其他节点，因此平均相对误差略高于其他两个应用 Pig-UserVisitAggr 和 TeraSort（这两个应用处理的数据分布相对均匀）。

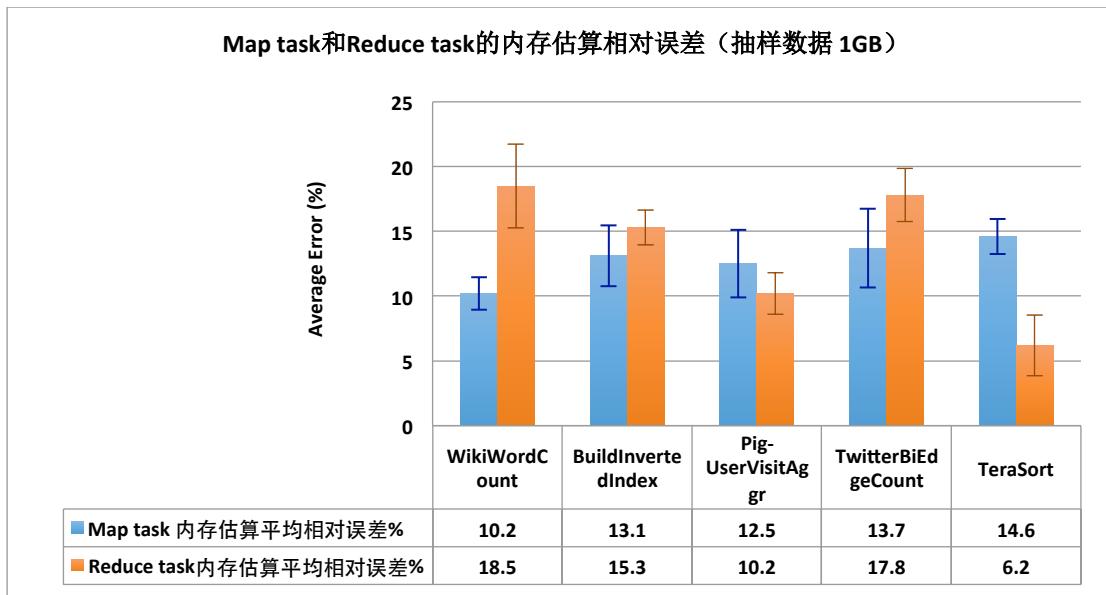


图 21 Map/reduce task 的内存用量估算相对误差（抽样数据为 1GB）

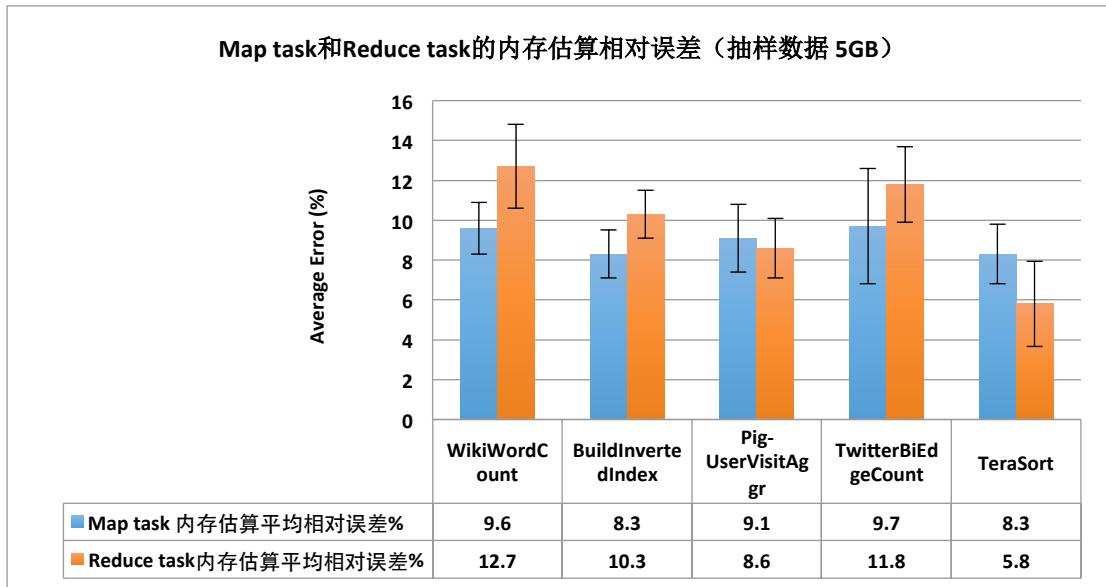


图 22 Map/reduce task 的内存用量估算相对误差（抽样数据为 5GB）

4.7 进一步讨论

在线预测问题：由于内存用量模型中的一些参数（如用户代码的复杂度）需要在运行后才能确定，因此我们采用在小数据集上运行应用来获得相关信息，从而能够计算这些参数。在线预测没有历史数据可以参考，因此目前还难以实现。

Garbage Collection 对内存用量的影响问题：由于 JVM 在 GC 之前不会主动清除已经不再被引用的对象，因此直接监控得到的内存用量可能会大于实际的内存用量。为

了避免这种情况的发生，我们在 t 时刻监控内存时，会先让 JVM 进行一次 GC。

多个应用同时运行时对内存用量的影响问题：集群中同时运行的应用个数和类型对某个应用的 task 内存用量来说没有影响。因为在 Hadoop MapReduce 中，每个应用的 task 是一个独立进程（JVM），进程和进程之间的内存空间隔离，不存在影响。只有当集群内存资源非常有限，而且过多的 task 在同时运行的话，每个 task 可用的内存空间会变小，但不影响 task 本身的内存用量。

4.8 相关工作比较

由于 MapReduce 框架不提供性能模型和性能预测工具，很多研究人员研究了 MapReduce 应用性能模型及应用性能优化方法。

MapReduce 执行时间估算与最优资源分配：Morton 等人 [Morton2010] 提出了一个在线的启发式方法来预测 MapReduce job 的执行时间，该方法借鉴了数据库中的 SQL 查询进度估计器。Verma 等人 [ARIA2011] 提出了一个具有理论上下界的时间模型来分析 MapReduce 的执行时间。他们 [Verma2011] 也讨论如何为 MapReduce job 分配最佳的资源（map/reduce slot）来保证 MapReduce job 能够按时完成。Ganapathi 等人 [Ganapathi 2010] 使用机器学习方法（Kernal Canonical Correlation Analysis）来构建 Hive query 特征和 query 执行时间之间的关系。Starfish [Herodotou2011] 提供了一个 cost-based 的优化器来寻找 job 运行时的最佳配置参数。Starfish 包含一个 what-if 引擎可以预测 job 在不同配置参数下的性能（主要是执行时间）。Starfish 也包含一个数据流模型，但没有涉及内存用量的估算。PerfXplain [Khoussainova2012] 提出了一个查询语言来解释为什么 job 运行性能与用户观察或者期望的性能有区别。PerfXplain 通过成对 jobs 的特征对比来回答用户提出的 job 性能差异问题。

很少有工作关注 MapReduce 的内存使用情况，Singer 等人 [Singer2011] 为单机多核环境设计了一个 fork-join 风格的 MapReduce 框架，在该框架中，他们使用机器学习方法来寻找最合适的垃圾回收算法（GC policy）。但方法不涉及内存用量的分析和估算，也不是针对分布式的 MapReduce 框架。

4.9 本章小结

为了帮助用户分析、估算应用的内存用量，我们以 MapReduce 框架为基础，研究了 MapReduce 应用的内存用量模型构建方法，该方法以数据流为中心构建应用内存用量模型，目的是量化“静态配置—数据流—内存用量”之间的映射关系。更进一步，我们基于从抽样数据上获得的应用运行信息来估算应用在大数据上的内存用量。我们在常见的 MapReduce 应用上做了评估，实验发现当抽样数据为 1GB 时，平均误差小于 20%，当抽样数据为 5GB 时，平均误差小于 13%。

第五章 应用内存溢出错误诊断方法及工具

本章讨论如何设计面向分布式数据并行应用的内存溢出错误诊断方法及工具。

5.1 概述

虽然内存溢出错误的原因及修复方法已经在第三章中分析过，但当某个应用出现内存用量过高或者内存溢出错误时，用户仍然难以定位出错误原因。为此，我们希望设计一个可以诊断分布式数据并行应用中内存溢出错误的方法及工具。

当前的内存分析工具和内存泄漏检测方法不能直接用于内存溢出错误诊断，因为它们只能分析出内存中有哪些对象及哪些对象本应该释放却没有释放，不能分析出这些对象的来源及对象占用空间大的原因。

在第三章中，我们研究发现不恰当的配置参数、数据倾斜、内存使用密集的用户代码都可以导致应用的内存消耗量过大。在本章中，我们基于上一章构建的内存用量模型，设计了一个内存分析与诊断工具，叫做 Mprof。Mprof 可以自动建立 MapReduce 应用内存用量与应用静态因素之间的关系，也就是“静态配置—数据流—内存用量”之间的关系。Mprof 采用的方法是重建数据流，重建用户代码内存使用历史信息，并对两者进行关联分析。Mprof 也包含一些定量诊断规则，这些规则根据“静态配置—数据流—内存用量”定量关系来定位内存溢出错误相关的代码，错误相关的数据，及不恰当的配置参数。Mprof 仅仅依赖于 task 的运行日志，数据流计数器和 heap dumps，不需要对用户代码进行修改。

5.2 问题分析

对用户来说，当 map/reduce task 出现内存溢出错误时，只能从错误栈中查看到正在运行的函数（方法），无法直接定位出错误原因。对我们来说，设计内存溢出错误诊断方法和工具有两个挑战：(1) 静态配置不直接影响分布运行的 map/reduce task 的内存用量。(2) 用户代码可以被任意书写，也可以由高层语言（比如 SQL-like 的 Pig 脚本）或高层库（比如 Mahout）产生。这样，对错误诊断来说，用户代码相当于黑盒。

当前的内存分析工具，比如 Eclipse MAT [MAT]，JProfiler [JProfiler]，只能分析当前内存（heap）里面有哪些对象及对象的特征（如大小，引用关系）。静态和动态的内存泄漏诊断方法，如 [Cherem2007, Xie2005, Jump2007, Xu2008] 可以诊断出哪些对象（可能）没有被释放。但不管是内存分析工具还是内存泄漏诊断工具都不能分析出内存对象的来源及对象占用空间大的原因。而且，内存溢出错误的主要原因是内存消耗过大，

而非内存泄漏。

不同于单机程序，MapReduce 应用有三大特点，这些特点可以帮助我们诊断内存溢出错误。这三个特点是：(1) 用户需要配置大小合适的框架缓冲区来平衡框架和用户代码的内存消耗。(2) MapReduce 框架分布处理 $\langle k, v \rangle$ records，并在运行时对 $\langle k, v \rangle$ records 进行划分(partition)和聚合(aggregation)。(3) 用户代码以 streaming-style 的方式读入、处理、输出 $\langle k, v \rangle$ records。

回顾第三章的研究发现，我们总结出的三个内存溢出的原因是：(1) 不合适的配置参数可以导致框架暂存过多的中间数据。(2) 数据流异常（数据倾斜）可以导致运行时产生中间数据过大。(3) 内存使用密集的用户代码可以导致用户代码在处理数据时内存消耗过大。因为内存溢出错误与配置、数据流和用户代码都相关，所以我们诊断内存溢出错误的思路是先建立应用内存用量与静态信息（配置和用户代码）之间的关系，然后设计定量规则来将内存溢出错误归因到错误相关的代码、数据和配置参数。

基于以上思路，我们可以设计一个内存分析工具。该工具首先自动化地构建 task 的内存用量与 job 静态信息之间的关系。构建方法是基于上一章的内存用量模型来重建 job 的数据流，重建用户代码的内存历史使用信息，然后对两者进行关联分析量化“静态配置—数据流—内存用量”之间的关系。与上一章估算内存用量不同的是，本章的问题是出了错误以后诊断，不需要进行内存估算，但需要准确量化“静态配置—数据流—内存用量”之间的关系。得到量化关系后，我们可以设计一些定量的诊断规则，利用规则将内存溢出错误归因到错误相关的用户代码、数据及静态配置参数。整个工具可以只依赖增强过的 task 的日志、数据流计数器和 heap dumps，不需要对用户代码进行修改。

5.3 内存溢出诊断工具 Mprof 的设计与实现

基于上节的问题分析，我们设计并实现了内存分析与内存溢出诊断工具 Mprof。Mprof 主要包括三个部分：(1) 数据流分析器 (dataflow profiler)。该分析器目的是重建数据流，也就是量化运行过的 tasks 中的输入/输出/中间数据。该分析器首先构造上一章提到的数据流模型，然后通过采集 task 的日志和数据流计数器中的数据流信息，重建并量化 MapReduce 数据处理过程中的基本数据依赖关系。(2) 用户代码内存分析器 (user code memory profiler)。该分析器目的是构建运行过的 tasks (特别是出现内存溢出错误的 tasks) 的用户代码内存历史使用信息。与构建内存用量模型一样，这里最主要的任务是去量化用户代码产生的中间结果 (用户对象) 与用户代码输入数据之间的关系。在上一章中，我们发现用户对象有不同但是固定的生命周期，而且也设计了一个生命周期敏感的内存监控策略 (lifecycle-aware memory monitoring strategy)，但使用的是粗略的监控方法。在 Mprof 中，我们仍然采取这个监控策略，但使用的是基于 heap dump 的精

确监控方法。通过使用这个策略，Mprof 可以直接量化用户代码产生的不同生命周期对象与其对应数据之间的关系，还能识别出内存使用模式。(3) 两种定量的诊断规则(rules)。这两种定量规则用来定位错误相关的代码、数据和配置参数：(a) 面向用户代码的规则(rules for user code)，可以定位出用户代码中的错误类型和错误相关的输入数据。(b) 面向数据流的规则(rules for dataflow)，该类规则基于数据流模型（包含对数据倾斜的度量），可以定位出具体出现异常的数据及不恰当的配置参数。

5.3.1 数据流分析器

数据流分析器的目的是重建数据流，具体包含下面三个步骤：(1) 构建数据流模型，模型与 4.5.1 小节的数据流模型一致。(2) 获取模型参数。上一章中获取参数的方法是先在小数据集上运行 job，然后从 task 的增强日志，数据流计数器中获取数据流信息，最后估算 job 在大数据集上运行时的数据流参数。不同的是这里不需要进行参数估计，直接将获取到的数据流参数填入模型中即可。(3) 对数据流进行异常检测，检测目的是衡量数据流是否异常，比如数据划分是否均衡(partition unbalance)，是否有异常大的 $\langle k, list(v) \rangle$ group (outlier group in aggregated partition P_i) 和异常大的单一 $\langle k, v \rangle$ record (outlier record in G_i)，检测指标如表 15 所示：

表 15 数据流异常模型

数据流异常检测指标	衡量标准
Partition unbalance	$Gini(P_1, P_2, \dots, P_r) > 0.4, P_i = \sum_{m=1}^M P_{mi}(k, v)$
Outlier group in P_i	$x_i \geq UpperInnerFence(G_{i1}, G_{i2}, \dots, G_{in})$
Outlier record in G_i	$x_i \geq UpperInnerFence(R_{i1}, R_{i2}, \dots, R_{in})$

* $UpperInnerFence(x_1, x_2, \dots, x_n) = Q3[x_1, x_2, \dots, x_n] + 1.5 * IQ[x_1, x_2, \dots, x_n]$. 如果 $x_i \geq UpperInnerFence$, 那么 x_i 被认为是 outlier. Q3 是三分位数, 而 IQ 是四分位距.

数据流异常检测指标将会被应用于 5.4 节的诊断规则。*Gini* 系数 [GiniIndex] 在经济学中广泛被应用于度量收入是否均衡。这里，我们使用 *Gini* 系数来度量数据划分 partition 是否均衡，一般 *Gini* 系数大于 0.4 就被视为不均衡。另外，为了度量是否存在异常大的 $\langle k, list(v) \rangle$ group 和异常大的单一 record，我们采用统计学中常用的 outlier 算法 [OutlierAlgo]。该算法在给定一组数字 (x_1, x_2, \dots, x_n) 后，首先计算该组数字的 upper inner fence，具体计算公式见上表，Q3 是三分位数，IQ 是四分位距。然后，如果 x_i 大于 upper inner fence，那么 x_i 被视为 outlier。这里，如果 aggregated partition i 中的第 j 个 $\langle k, list(v) \rangle$ group (也就是 G_{ij}) 中包含的 records 个数大于或等于 $UpperInnerFence(G_{i1}, G_{i2}, \dots, G_{in})$ ，那么 G_{ij} 被视为是异常大的 $\langle k, list(v) \rangle$ group。

5.3.2 用户代码内存分析器

用户代码内存分析器目的是去构建和量化用户代码内存消耗与其输入数据之间的关系，并识别出内存使用模式。在上一章中，我们发现用户对象有不同的生命周期，map/group/reduce-level 用户对象属于累积计算结果 (accumulated results)，而 record-level 用户对象属于暂时性的中间计算结果 (intermediate results)。基于此特性，我们也设计了一个生命周期敏感的内存监控策略 (lifecycle-aware memory monitoring strategy)。通过这个策略，我们可以重建用户代码内存使用历史信息，识别出内存使用模式。

生命周期敏感的内存用量监控策略（使用 heap dump 精确监控）

在内存溢出错误发生的时候，task 中的用户对象由累积计算结果 (map-level, group-level 或 reduce-level user objects) 和暂时性的中间计算结果 (record-level user objects) 组成。这些计算结果有固定的生命周期且与输入数据中的不同部分相关，因此我们可以将大问题（还原并量化用户对象与输入数据之间的关系）化成小问题（分别量化累积计算结果和中间计算结果与其相应的输入数据之间的关系），具体方法与上一章中的监控策略基本一致，不同的是这里使用基于 heap dump 的精确监控方法。这样，监控结果不仅精确，而且可以辅助定位出错误相关的代码段。

A. 还原并量化累积计算结果与其输入数据之间的关系

为了还原出累积计算结果的累积趋势，即与其输入数据之间的关系，我们选择在一些特殊的时间点进行 heap dump (将当前时间点的 JVM heap 进行快照，并将快照存放至磁盘)。对于 map-level 的累积计算结果，可选的时间点是当用户代码刚刚完成当前的 record 处理 (R_i)，并将去处理下一个 record (R_{i+1})。在这些时间点，record-level 的中间结果已经被清理，所以当前 heap 中的用户对象大小即是当前累积结果的大小。对于 reduce-level 的累积结果来说，可选的时间点是用户刚刚处理当前的 $\langle k, list(v) \rangle$ group (G_i)，正准备去处理下一个 $\langle k, list(v) \rangle$ group (G_{i+1})。为了提高效率，我们并不是在每个 R_i 和 G_i 处都进行 heap dump，而是每隔若干个 R_i 和 G_i 进行 heap dump (如图 19 中的竖线所示)。对于 map-level 的累积结果，我们选择每隔 $(n-1)/m$ 个 records 进行 heap dump， n 是内存溢出发生时已经被处理过的 records 个数，而 m 是想要生成的 heap dump 个数 (默认是 10，也可以由用户设置)。对于 reduce-level 的累积结果来说， n 是内存溢出错误发生时已经被处理过的 $\langle k, list(v) \rangle$ group 个数。对于 group-level 累积结果来说 (我们只关心在最后一个 $\langle k, list(v) \rangle$ group 里面的 group-level 累积结果)，内存用量监控策略与 map() 中的策略是一样的。最后我们抽取并计算每个 heap 中的用户对象大小，然后将这些对象大小连接起来作出累积结果的内存使用趋势线。图中 M_i 和 S_i 分别表示在第 i

个 record/group 间隔的积累率 (accumulation rate)。另外，在 R_1 和 G_1 被处理前的用户对象大小表示被用户加载到用户代码的外部数据大小。

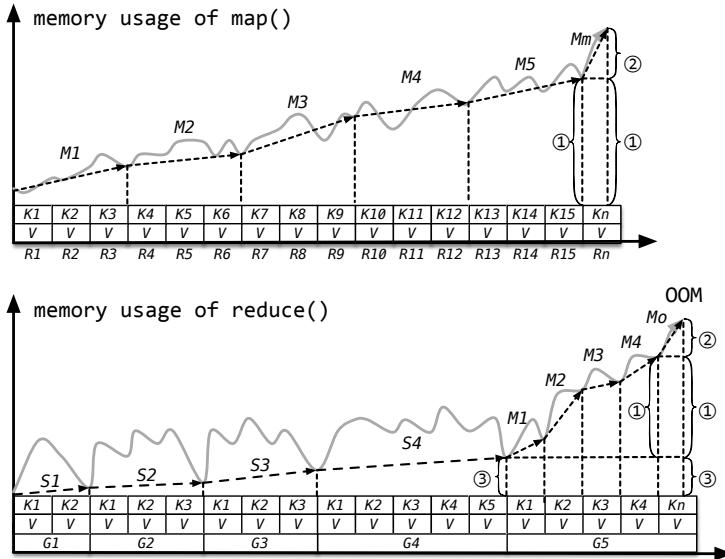


图 23 生命周期敏感的用户代码内存用量监控策略（使用 heap dump）

B. 还原并量化当前中间计算结果与当前输入 record 的关系

我们只关心在当前（也就是内存溢出错误发生时正在处理的 record） R_n 对应的中间计算结果。所以，要选择的 heap dump 时间点是： R_n 即将被处理及处理 R_n 发生内存溢出错误时。这两个 heap 之间的用户对象大小差（如上图中的 M_o ）可以被视作当前中间结果大小。然而，我们仍然需要去检测这个 record-level 的中间计算结果是否只与 R_n 相关还是与之前的 records 也相关。具体的检测方法如下：我们重新运行用户代码，并让它只处理 R_n （跳过其他的 records，Hadoop 框架支持 skip records 机制），然后重新计算 R_n 即将被处理和被处理时（要输出 new record 的时候）的对象大小差，如果这个差与 M_o 相同，那么我们可以断定当前 record-level 的中间计算结果只与 R_n 相关。

C. 监控策略的实现与实施

我们在 Hadoop MapReduce 框架中实现了上述监控策略。具体方法是在框架提供的 record 接口，比如 `RecordReader.nextKeyValue()` 和 `ValueItertor.next()` 中，插桩数据流监控和 heap dump 的代码。当 `map()`/`reduce()`/`combine()` 调用这些接口处理下一个 R_i/G_i 时，代码会检查是否要在当前的 R_i/G_i 处进行 heap dump，如果需要就先进行一次 GC，清除掉不再被引用的对象，然后进行 heap dump。

当内存溢出错误发生的时候，我们重新运行 job，同时将在 job 上应用上述监控策略。因为重新运行的 job 的输入数据、配置参数和用户代码均未改变，内存溢出错误还会像上次一样发生。唯一非确定性（non-deterministic）的地方是如果 reducer 运行两次

的话，reducer 可能 shuffle 得到不同次序的数据分块（data partition）[Xiao2014]。为了避免这个问题，我们在第一次 job 失效时会记录 shuffle 到的数据分块顺序，然后在重新运行的 job 上重放（replay）这个顺序。

D. 识别用户代码内存使用模式

在得到用户代码的历史使用信息后，用户代码内存分析器会对内存使用趋势和输入数据进行关联分析，进而发现内存使用模式。发现内存模式主要是去判定是否有持续增长（continuous growth）和陡峭增长（sharp growth）趋势。我们的分析器具有扩展性，更多的监控策略可以随时添加到分析器中。与上一章分析用户代码的空间度一样，我们的分析器主要判定在输入数据的特定区间内是否存在内存持续增长或陡峭增长的趋势。持续增长趋势包括线性增长和非线性增长，比如上一章分析过的线性、多项式、指数增长。陡峭增长主要是识别用户代码在处理某个 $\langle k, list(v) \rangle$ group 或 $\langle k, v \rangle$ record 时是否存在内存突然增长趋势。如果存在，说明这个 group 或者 record 与错误强相关。具体的模式判别法则如表 16 所示：

表 16 用户代码内存使用模式识别方法

内存使用模式	相关数据	模式识别方法
Linear growth	$[G_1, G_m], [R_1, R_n]$	$PearsonCorr([x_1, x_n], [U_{x_1}, U_{x_n}])^*$
Polynomial growth	$[G_1, G_m], [R_1, R_n]$	$U = \beta_0 + \beta_1 x + \beta_2 x^2 + \dots + \beta_n x^n$
Sharp growth at current record R_n	R_n	$S_n \geq UpperInnerFence(M_1, M_2, \dots, M_n)$
Sharp growth at current G_m	G_m	$S_m \geq UpperInnerFence(S_1, S_2, \dots, S_m)$

*在统计学中，pearson 系数 [PearsonCorr] 被广泛应用于检测两组变量的线性相关性。如果 PearsonCorr(X, Y) > 0.9，X 和 Y 被视为强线性相关。在这里，如果 PearsonCorr(X, U) > 0.9，那么 $[G_1, G_m]$ 或 $[R_1, R_n]$ 上的内存用两区是被视为线性增长。Linear growth 和 polynomial growth 都被视为持续增长（continuous growth）。如果当前的 record (R_n) 或者 group (G_m) 出现增长斜率如果大于等于其他 record/group 中的增长斜率的 UpperInnerFence，那么在 R_n 或 G_m 上的增长被视为陡峭增长（sharp growth）。

5.4 Mprof 用于内存溢出错误诊断

内存溢出错误诊断的目的是：(1) 诊断出内存消耗量大的用户代码，包含用户代码的错误类型及内存消耗高的代码段。(2) 诊断出内存溢出错误相关数据。(3) 诊断出不恰当的配置参数。

图 20 展示了内存溢出错误的诊断过程。诊断过程先定位内存消耗较大的代码段，然后将内存消耗量大的用户对象归因到错误相关的数据和配置参数。具体包含四个步骤。

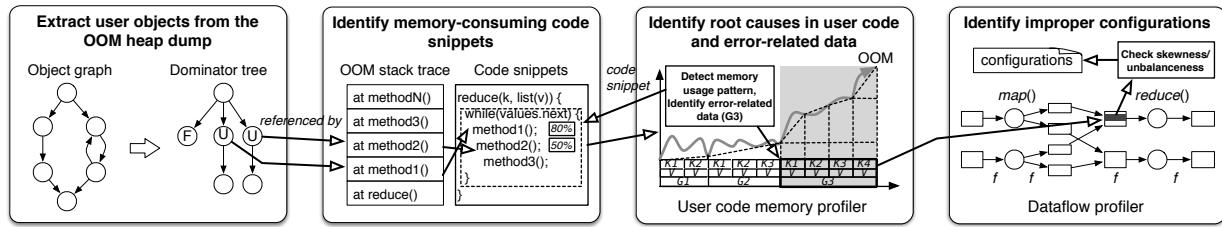


图 24 内存溢出错误诊断过程（如果错误发生在 reduce() 中）

A. 识别内存消耗量大的代码段

这一步首先从内存溢出时的 heap dump 中抽取占用空间大的用户对象，然后识别出引用这些对象的代码段（方法）。一个 heap dump 就是一个对象图，图中每个节点代表一个对象，每条边代表对象的引用关系。为了抽取出占用空间大的对象，首先将对象图转换为 dominator tree [DominatorTree]。然后通过识别对象被引用的方法来抽取用户对象，之后再筛选占用空间大的用户对象（比如 3MB+ 的对象）。比如在图 20 中，如果 *ArrayList* 被 *method2()* 引用，也就是被 *reduce()* 间接引用，那么这个对象被认为是 *reduce()* 产生的用户对象。通过计算每个方法引用的用户对象大小，可以定位内存消耗量大的代码段。比如在图 20 中，80%（比如 80MB）的用户对象被 *method1()* 引用，50% 的用户对象被 *method2()* 引用，那么内存消耗量大的代码段是 *method1()* 和 *method2()*。下面的代码段展示了一个 411MB 的用户对象 *wcMap:HashMap* 被 *InMemWordCount\$Mapper* 类的 *map()* 引用，具体内存消耗量大的代码段是 *Mapper.map(InMemWordCount.java:49)*。

```

at java.util.HashMap.put(HashMap.java:372)
at mapper.InMemWordCount$Mapper.map(InMemWordCount.java:49)
=> wcMap java.util.HashMap @ 0xdbc5e0c0 (430,875,440 B)
at org.apache.hadoop.mapreduce.Mapper.run(Mapper.java:203)
...
at org.apache.hadoop.mapred.Child.main(Child.java:404)

```

对于框架对象（也就是框架缓冲区和暂存在缓冲区中的中间数据），Mprof 通过识别对象名可以直接将框架对象从 heap dump 抽取出来。比如在 Hadoop MapReduce 中，*org.apache.hadoop.mapred.Merger.Segment* 表示的就是暂存的中间数据（shuffled data），*kvbuffer* 表示的是 map buffer。

B. 识别用户代码中的错误类型和错误相关的数据

基于还原出的用户代码内存消耗趋势及识别出的内存使用模式，我们设计了五个用户代码规则（rules for user code）来识别用户代码中的错误类型以及错误相关的数据。

如果某个规则 r 对应的错误现象出现，那么该规则被使用。比如用户对象在 $[R_1, R_n]$ 上如果出现持续增长，那么我们可以判定 map() 出现了 map-level 的累积计算结果，其对应的数据是 $[R_1, R_n]$ ，也就是 map() 所有的输入 records。当前的 rules 虽然不完整，但是规则具有扩展性。比如，对于复杂的增长趋势需要使用复杂的函数来拟合。为了简单，我们当前只考虑线性和多项式增长情况。对于其他的不能匹配到情况，我们的工具会绘制内存使用趋势线，而且会输出最陡峭的 N 个内存增长区间及其相关数据。用户可以根据趋势线去断定哪些区域是错误最相关的数据。

Rule 1, 4 和 5 被用于 map()，而 rule 2, 3 和 4 被用于 reduce()。在 rule 1 中，如果用户对象在 $[R_1, R_n]$ 区间出现持续增长趋势，那么工具会判定错误原因是 map-level 的累积结果，错误相关数据是 $[R_1, R_n]$ 。下一个执行步骤（这里是检查 *input split size*）会识别出 *input split size* 配置参数过大。其他 rules（2 到 5）的工作模式也一样。如果 rule 2 被匹配到，那么错误相关的数据是 reduce() 的全部输入，也就是 aggregated partition，下一个动作就是依据数据流模型，调用 rule 6-1 和 6-2 去检测 data partition 是否异常。Rule 3 的下一步是去调用 rule 7 去检测热点 key 是否存在。Rule 4 的下一步是调用 rule 8 去检测当前的 R_n 是否是一个异常大的 record。

表 17 用户代码规则（用于诊断用户代码相关的错误原因和错误相关的数据）

ID	现象	错误原因	错误相关数据	下一步操作
1	$[R_1, R_n]$ 上内存用量持续增长	Map-level 累积计算结果太大	Map()所有的输入数据	检测 <i>input split size</i> 是否过大
2	$[G_1, G_n]$ 上内存用量持续增长	Reduce-level 累积计算结果太大	Reduce() 的所有输入数据	检测数据划分是否异常（采用 Rule 6）
3	G_n 内部内存用量持续增长	Group-level 累积计算结果太大	G_n 里的所有 records	检测是否存在热点 key (Rule 7)
4	R_n 处内存用量陡峭增长	Record-level 中间计算结果太大	当前输入的 record	检测单一 record 是否过大
5	R_1 或 G_1 处的内存用量是外部数据量的 m 倍	用户加载了大量外部数据		输出 m

C. 识别数据倾斜和不合适的配置参数

基于识别出的错误相关数据和还原的数据流，我们设计了三个统计规则 rule 6, 7 和 8 来识别数据倾斜和不恰当的配置参数。比如在 rule 7 中，如果在当前 $\langle k, list(v) \rangle$ 组 G_n 中出现了陡峭增长（相比其他 G_i 中的内存增长斜率， G_n 中的内存增长斜率是 outlier），而且 G_n 比其他 groups 包含更多的 records，那么识别出的错误原因是热点 key。最后，rule 9-1 和 9-2 用于判定错误是否是因为框架暂存了大量数据。

表 18 数据流规则（用于诊断数据流异常和不恰当的配置参数）

ID	现象	错误原因	不合适的配置参数
6-1	Rule 2 生效且 partition 是平衡的	不合适的数据划分	reduce number 太小
6-2	Rule 2 生效但 partition unbalance	不合适的数据划分	Partition 函数不均衡
7	G_n 中内存用量出现陡峭增长，且 G_n 具有比其他组包含更多的 records	热点 key	Partition 函数选择了热点 key
8	Rule 4 生效且 R_n 比其他 R_i 都大	单一 record 过大	无
9-1	固定 buffer 存在且 map() 将要完成	框架缓冲区太大	固定 buffer 太大
9-2	虚拟 buffer 被填满，且 combine() 或 reduce() 将要完成（比如执行了 80%）	框架缓冲区太大	虚拟 buffer 太大

5.5 实验评价

实验目的是验证我们的工具诊断出来的溢出错误原因是真正的错误原因。

5.5.1 实验环境及实验步骤

我们重现了 20 个现实世界的内存溢出错误，这些错误发生在不同类型的 MapReduce 应用里面。这些应用代码包括用户手写的 MapReduce 代码，也包含由高层语言或高层库（Apache Pig, Apache Hive, Apache Mahout, Cloud9 等）产生的代码。这些错误来自公开论坛，包括 StackOverflow.com, Hadoop mailing list, 和 MapReduce 开发者的博客。这些错误具有详细的输入数据信息，可重现的用户代码，内存溢出错误栈等等。因为我们没有用户原始的输入数据，我们使用公开数据集（英文 Wikipedia）和合成数据集（随机文本和一个知名的用于对比 MapReduce 和并行数据库性能的 benchmark [BrownBench]）作为输入数据。我们确认重现的内存溢出的错误栈与用户报告的相同。所有的应用运行在实验室 10 个节点的 Hadoop 集群上，使用的 Hadoop 版本是我们提升过的 Hadoop-1.2 [EHadoop]，这个版本可以支持自动在 R_i/G_i 上面进行 heap dump。集群中每个节点配备 16GB 的内存，每个 map/reduce task 的 heap 大小设置为 1GB。默认情况下，我们配置的 *input split size* 是 512MB, *buffer size* 是 500MB, *reduce number* 是 9, *partition* 函数是 HashPartition。

错误原因诊断流程：对于每一个 job，我们首先将其按照一个正常 job 运行，然后记录在内存溢出发生时已经处理的 record 数目。之后，我们重新运行这个 job，job 自动使用内存用量监控策略产生 heap dumps。我们提升过的 Eclipse MAT [EMAT] 可以自动从每个 heap dump 中抽取用户对象和框架对象，并计算它们的大小。最后，我们使用诊断工具 Mprof 来执行整个诊断流程，定位出错原因。

5.5.2 诊断结果

表 19 内存溢出错误诊断结果

阶段	Job 名称	规则	错误原因	诊断正确	验证(修复方法)
Map	NLPLemmaizer	4,8	Record-level 中间计算结果过大, 单个 record 过大	✓	
	InMemWordCount	1	Map-level 累积结果过大	✓	
	MapSideAggregation	1	Map-level 累积结果过大	☒	
	PigDistinctCount	3	Group-level 累积结果, 热点 key	☒	
	CDHJob	9-1	固定缓冲区过大	✓	
	MahoutBayes	5	加载大量外部数据	✓	
	MahoutConvertText	5	加载大量外部数据	✓ (+)	减小外部数据(训练数据)
	HashJoin	5	加载大量外部数据	✓ (+)	减小外部数据(第一个表)
Shuffle	ShuffleInMemory	6-1	不恰当的数据划分	✓ (+)	增加 partition number
	WordCount-like	9-2	虚拟缓冲区过大	✓	
	PigJoin	9-2	虚拟缓冲区过大	✓	
	NestedDISTINCT	3	Group-level 累积结果过大, 热点 key	☒(+)	更换 key, 将累积操作分步执行
	PigOrderLimit	3,7	Group-level 累积结果过大, 热点 key	☒	
Reduce	GraphPartitioner	3	Group-level 累积结果过大	✓	
	FindFrequentValues	3	Group-level 累积结果过大	✓	
	ReduceJoin	3,7	Group-level 累积结果过大, 热点 key	✓	
	PositionalIndexer	3,7	Group-level 累积结果过大, 热点 key	✓	
	BuildInvertedIndex	3,7	Group-level 累积结果过大, 热点 key	✓	
	CooccurMatrix	3,7	Group-level 累积结果过大, 热点 key	✓	
	JoinLargeGroups	3,7	Group-level 累积结果过大, 热点 key	☒(+)	更换 key

表 19 展示了 Mprof 的诊断结果, 结果中包含 job 名称(里面链接指向真实的应用), 匹配到的规则, 以及诊断出的错误原因是不是真正的错误原因。其中 15 个错误原因已经被专家(Hadoop committer 或有经验的开发者)诊断出来, 剩下的 5 个错误(标记为+)通过间接验证的方法出来(比如尝试一些修复方法来看内存溢出错误是否会消失)。对号表示我们的方法诊断出来的错误原因与真实的错误原因一样。半对号表示我们诊断

出来的错误原因与真实的错误原因部分相同(只有内存消耗量大的代码段不能诊断出来,因为这 5 个应用的用户代码由高层语言或高层库产生)。除了诊断结果,我们也将通过案例研究来展示了 Mprof 如何利用量化后的“内存用量—数据流—配置参数”和量化的规则来诊断出来错误原因。

5.5.3 案例研究

案例 1: InMemWordCount

这个应用的目的是去统计维基百科中每个单词出现的次数。这个例子与 MapReduce 论文 [MapReduce] 中给出的 WordCount 的例子的功能一样。唯一的差别是这里的 map() 分配了一个 *HashMap* 来聚合每个 word 产生的中间结果`<word, count>`, 而非像原始的 WordCount 那样直接输出`<word, count>`。这里, 每个 record 的 value 是一个 *line*, 而 key 是这个 *line* 在文本中的位置。从源码看来, 我们只能猜测错误原因是 *wcMap* 缓存了太多的`<word, count>`中间结果。然而, 原因也可能是 StringTokenizer *st* 在处理一个异常大的 *line* 的时候发生内存溢出错误。在不知道代码语义的情况下, 我们的方法先去还原 map() 用户代码的内存用量趋势(见图 21a)。然后, 我们的诊断工具发现 $[R_1, R_n)$ 上的内存用量具有线性增长趋势(这里 $n=9,651$), 而且 $R_{9,651}$ 处的内存增长只有($M_o=1.2\text{MB}$)。所以, Rule 1 被匹配, 错误原因是 map-level 的累积中间结果过大(也就是产生了大量的`<word, count>`中间结果)。更确切地, 诊断工具发现累积的计算结果被存放在一个 411MB 的 *wcMap:HashMap* 里面, 这个 *wcMap* 被 *wcMap.put(word, 1)* 所引用(用`=>`标示)。我们诊断出的错误原因与有经验的应用开发者诊断出的错误原因一致: *wcMap* 存放了太多的`<word, count>`中间结果。Rule 1 的下一步诊断出不合适的配置参数是 *input split size*, 这里是 512MB。为了修复这个错误, 用户可以降低 *input split size* 来减少相应的累积结果, 或者仍然使用最初版本的 WordCount(没有累积操作)。

```
public class Mapper {
    Map wcMap = new HashMap<String, Integer>();
    public void map(Long key, Text value) {
        st = new StringTokenizer(value.toString());
        while (st.hasMoreTokens()) {
            String word = st.nextToken();
            if (wcMap.containsKey(word))
                wcMap.put(word, wcMap.get(word) + 1);
            else
                wcMap.put(word, 1); => wcMap:HashMap (411MB)
        }
    }
}
```

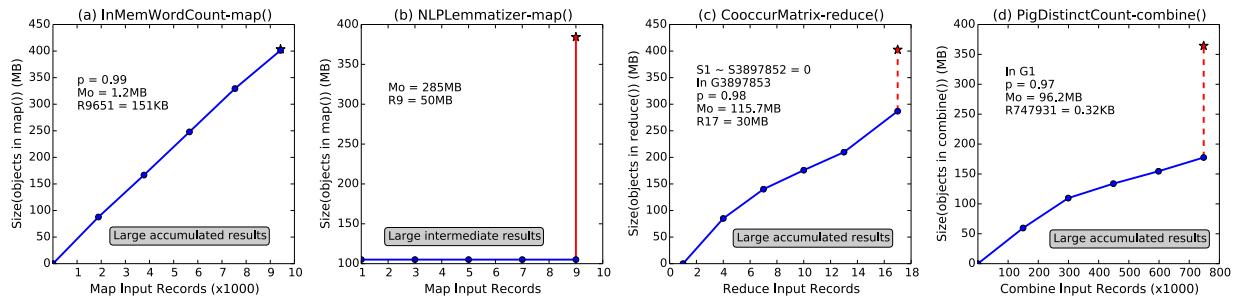


图 25 案例中的用户代码内存用量趋势图

案例 2: NLPLemmatizer

这个应用通过使用一个第三方库 *StanfordLemmatizer* 来对维基百科中的 words 进行屈折变换 (lemmatize)。每个 record 的 value 是文本中的一行 (*line*), key 是该行 (*line*) 在文本中的位置。从代码中我们不能直接确定错误原因, 原因可能是 *slem* 累积了太多的中间结果, 当前 record 产生的中间结果太大, 或者两者都有。在还原出 map() 的内存使用趋势 (见图 21b) 后, 我们的诊断工具识别出在 $[R_1, R_n]$ (这里 $n=9$) 上并没有一个持续增长趋势, 但 R_9 处有一个陡峭增长, 而且这个陡峭增长只与 R_9 有关, 所以 Rule 4 被匹配, 得到错误原因是 record-level 中间计算结果过大。更确切地, 这个 record-level 计算结果包含一个 232MB 的 *ArrayList* 和一个 50MB 的 *String*, 这两者都被 *slem.lemmatize()* 引用。然后, Rule 4 的下一步是去调用 Rule 8 去计算 R_9 的大小, 计算得到 R_9 的大小是 50MB (属于特别大的 *line*), 远远大于其他的 records (12MB)。所以, Rule 8 被匹配到, 错误原因是单一 record 过大。我们诊断出的错误原因与专家 (库作者) 诊断出的相同: *lemmatize()* 在对每个 *line* 进行打标签 (tag) 的时候会分配一个大的临时的数据结构, 这个数据结构比要处理的 *line* 大好多倍。为了修复这个错误, 用户可以将 R_9 分割成多个小的 records (*line*)。

```
public class Mapper {
    StanfordLemmatizer slem = new StanfordLemmatizer();
    public void map(Long key, Text value) {
        String line = value.toString();
        for (String word: slem.lemmatize(line)) => 282MB
            emit(word, 1);
    }
}
```

案例 3: CooccurMatrix

这个应用目的是去计算维基百科中词共现矩阵 (word co-occurrence matrix)。Reduce()分配了一个自定义的数据结构 (叫做 *OHMap*) 来聚合每个 $\langle k, list(v) \rangle$ group 里面的 records。每一个 key 是一个 word, value 是一个 *OHMap*, 用来持有该 word 的邻居 words。只分析源代码的话, 我们不能确定 *plus()* 的语义, 更不能确定错误原因。我们的诊断工具首先识别出 $[G_1, G_n] (n=3,897,853)$ 上没有线性增长关系, 因为 $[G_1, G_n]$ 上的增长斜率 ($S_1 \sim S_{3,897,852}$) 是 0。然而, 在 $G_n (n=3,897,853)$ 上的内存用量增长是线性增长。所以, Rule 3 被匹配到, 错误原因是 group-level 的中间累积结果 (也就是 *wordMap* 持有了太多 word 的临近词)。这些累积结果被存放在一个 372MB 的 *OHMap* 里面, 被引用的代码段是 *wordMap.plus(iter.next())*。然后, Rule 3 的下一步调用 Rule 7 来检测产生这么多 word 的原因是不是热点 key。我们的诊断工具发现 $G_{3,897,853}$ 比其他的 $\langle k, list(v) \rangle$ group 要多 6 倍的 records ($G_{3,897,853}$ 对应的 k 是 “of”)。另外, 在当前被处理的 R_{17} 处, 有一个内存陡增, 然而这个陡增 (用红色虚线标示) 与 $G_{3,897,853}$ 里面的 records 都相关, 因为当 reduce()单独处理 R_{17} 的时候, 内存的增量只有 30MB。错误栈中的关键词 “*Arrays.copyOf()*”告诉我们这个陡增的原因是数据结构膨胀。一些容器型的数据结构, 比如 *ArrayList* 和 *HashMap*, 可以在快要被填满的时候扩张 1.5 或者 2 倍。为了修复这个错误, 用户可以尝试去将累积的中间结果周期性地 spill 到磁盘上。

```
public class Reducer {
    public void reduce(Text key, Iterable<OHMap> values) {
        Iterator<OHMap> iter = values.iterator();
        OHMap wordMap = new OHMap();
        while (iter.hasNext()) { // for (V value : values)
            wordMap.plus(iter.next()); => wordMap (287MB)
        }
        emit(key, wordMap);
    }
}
```

案例 4: PigDistinctCount

内存溢出错误栈显示这个应用错误发生在 *combine()* 中。然而, 很难去确定错误原因, 因为这个应用是上层语言 Pig 产生的, 并没有显式的用户代码。该应用的整个处理过程如下: 该脚本首先对 *tableA* 中的 *pagerank* 列进行 *groupBy* 操作, 然后在每一个 *pagerank* 的 group 里面进行 *count(distinct pageurl)* 操作, 也就是统计每个 *pagerank* 值对应的不同的 *pageurl* 的个数。

```

a = LOAD "tableA" as (pagerank:int, pageurl, aveduration:int);
b = GROUP a BY pagerank;
c = FOREACH b {
    urls = DISTINCT a.pageurl;
    GENERATE group, COUNT(urls), SUM(a.aveduration);
};
STORE c into "/output/newTable";

```

我们的方法检测出在 G_1 里面内存用量有一个线性增长趋势, G_1 同时包含 747,931 个 records(见图 21d)。所以, Rule 3 被匹配, 错误原因是 `combine()` 在处理 G_1 中的 records 时, 产生了过多的 group-level 的中间累积结果(也就是包含相同 `pagerank` 的不同 `pageurl` 过多)。与案例 3 相同, R_{747931} 上的陡峭增长原因也是数据结构扩张。更确切地, Mprof 诊断出累积的中间结果包含一个 77MB 的 `pig.data.BinSedesTuple` 和 281MB 的 `pig.data.InternalDistinctBag`, 这两个大对象都被 `PigCombine.processOnePackageOutput()` 引用。查看 Pig 的 API 会发现, 对象 `InternalDistinctBag` 会将产生的中间数据存放到一个 `ArrayList` 里面进行排序。然后, 如果想要定位产生这个 group-level 累积结果的具体操作, 我们可以进一步去查看 Pig 守则 [PigManual], 然后会发现 `DISTINCT` 操作是错误的来源。这个操作会生成一个数据结构(也就是 `InternalDistinctBag`)来持有所有的输入 records, 然后将这些 records 进行排序和去重。为了修复这个错误, 用户可以选择不同的 key, 或者尝试将 `DISTINCT` 操作分割成多个流式的操作(利用框架的排序机制)。

5.6 讨论

当前方法的最大问题是需要一些用户手工操作: (1) 当一些用户代码用量出现一些特殊的趋势(比如含有大幅度跳跃的情况), 我们的规则不能被匹配到, 用户需要判定哪个区域的输入 records 是错误相关的(我们的工具可以辅助判定 top N 内存增幅相关的输入数据)。(2) 因为自动理解用户代码的语义是困难的, 用户需要将诊断出来的错误类型(比如 group-level 的累积结果)与代码语义结合起来, 特别是当用户代码是由高层语言产生的时候。

Mprof 没有增加应用运行时的 overhead, 仅仅依靠重新运行时的数据流信息和 heap dumps。唯一的 overhead 是需要用户重新运行一下 job 产生所需的 heap dumps。

尽管我们的方法是针对 MapReduce 框架设计的, 也可以用于其他的分布式处理框架如 Dryad 和 Spark。虽然这些 MapReduce-like 的框架有灵活的基于 DAG 的数据流, 但它们的基本数据依赖关系与 MapReduce 数据流相同。虽然这些框架支持 pipeline, 但这些框架的用户代码仍然是一个个处理 record(也就是 streaming-style), 所以用户代码

对象仍然有固定的生命周期。另外，job 的配置参数如 *buffer size*, *partition number* 和 *partiton* 函数同样在这些框架有效。

5.7 相关工作比较

通用编程语言里的内存相关问题已经被深入研究过，但是内存溢出错误诊断的工作很少，尤其是诊断大数据应用中的内存溢出问题。

内存泄漏检测：内存泄漏是常见错误而且可以导致内存溢出，所以研究人员提出了很多内存泄漏的检测方法。比如，flow-sensitive 分析 [Cherem2007, Xie2005] 将内存泄漏诊断问题变换为可达性问题。Cork [Jump2007] 利用对象的类型增长信息来检测包含无用对象的数据结构。Container profiling [Xu2008] 会跟踪 container 上的操作来定位没有被使用到的数据元素。然而，这些检测方法不能被直接用于数据并行应用的内存溢出错误诊断，因为这些方法不能量化“静态配置—数据流—内存用量”之间的关系。

减少应用内存用量方面：由于应用在处理大数据时会消耗大量的内存，一些研究者提出了一些减少应用内存消耗的办法。FAÇADE [Nguyen2015] 提供了用于降低用户代码内存消耗的用户代码编译和执行环境。FAÇADE 设计目的是将数据存储和数据操作分开，方法是将数据存放到 JVM 的堆外内存，将对堆内对象的数据操作转换为对 FAÇADE 的函数调用。对于 Java 对象本身产生的 overhead（也就是 Java 对象自身所需的 header 和 reference），Bu 等人 [Bu2013] 提出了两种减少 overhead 的方法，比如将大量数据对象（record object）合并少量的大对象。然而，他们没有研究内存溢出错误的诊断问题。Interruptile Tasks [Fang2015] 改进了现有的 task，使得 tas 具备一定的错误容忍能力。当 task 在运行时遇到内存用量过量或者内存溢出问题时，Interruptile task 会暂停当前 task 的运行，回收 task 中部分运行数据及中间结果，并将不能回收的结果 spill 到磁盘，然后执行用户定义的 interrupt 逻辑，等到内存用量下降到一定程度后，再让 task 继续运行。

配置参数问题检测：研究人员提出了很多黑盒和白盒方法来检测配置参数错误。比如，PeerPressure [Wang2004] 是一种黑盒方法，它使用统计方法来识别正常和异常节点的参数状态的变化。ConfAid [Attariyan2010] 是一种白盒方法，可以识别出配置参数中有问题的标记（token），方法是去分析应用二进制代码的控制流和数据流。这里，数据流指的是代码中的变量依赖关系（如 $val a = b + c$ 中的数据流是 b 和 c 流动到 a ）。上面的检测方法不能被直接用于诊断内存溢出相关的配置参数，因为这些方法不能建立动态内存用量和静态配置参数之间的关系。我们的方法也是一种白盒方法，即根据 MapReduce 数据流特点建立数据流模型，根据用户代码编程范型建立用户对象模型。

数据倾斜问题：在并行数据库领域，数据倾斜问题是一个广泛关注的研究问题。

比如 Pig 包含一个 SkewedJoin 机制，该机制的思想来自于 [DeWitt1992]。在 MapReduce 应用中，数据倾斜也被研究过，比如 SkewTune [Kwon2012] 可以根据用户自定义的代价函数来优化数据划分算法。SkewTune 可以同时优化 map 和 reduce 阶段的数据倾斜问题，而且可以保持数据输入的顺序。这些工具可以被用于修复数据流异常类型的内存溢出错误。

5.8 本章小结

内存溢出错误是分布式数据并行应用中常见且严重的运行时错误。当前的内存分析工具和内存泄漏诊断工具均不能诊断出应用内存溢出错误的原因。本章中，我们提出了一个面向分布式数据并行应用的内存溢出错误诊断工具 Mprof。Mprof 可以自动建立 应用内存用量与应用静态因素之间的关系。Mprof 也包含的定量诊断规则，这些规则可以根据“静态配置—数据流—内存用量”定量关系来定位内存溢出错误相关的代码，错误相关的数据，及不恰当的配置参数。我们在 20 个真实的内存溢出错误上评测了我们的方法，结果显示 Mprof 可以有效地诊断出 MapReduce 应用中的内存溢出错误。

第六章 结束语

本章对全文工作进行总结，列举了论文工作的主要贡献，分析了需要进一步研究的相关问题。

6.1 论文工作总结

本文深入研究了分布式数据并行应用的内存使用问题：(1) 如何分析与估算应用的内存使用量。(2) 如何分析、诊断与修复应用内存溢出错误。针对这两个实际研究问题，我们在三个方面开展了研究，取得的研究成果总结如下：

(1) 内存溢出错误的实证分析：本文从公开论坛收集研究了 123 个真实 Hadoop 和 Spark 应用的内存溢出错误。我们发现应用内存溢出错误的三大原因是：框架暂存的数据量过大，数据流异常以及内存使用密集的用户代码。我们也从 42 个包含修复信息的错误中总结出了常用的修复方法，包括如何降低框架暂存的数据量，如何减小运行时的中间数据量，及如何降低用户代码的内存消耗的方法。另外，我们也提出可以提升框架错误容忍能力和错误诊断能力的三种方法。

(2) 内存用量模型构建与用量估算：为了解释并量化应用的静态因素（数据、配置、用户代码）与动态内存用量的关系，我们以数据流为中心构建了分布式数据并行应用的内存用量模型。该模型包含三个子模型：数据流模型、框架内存模型和用户代码模型。数据流模型可以量化“静态配置—数据流”的关系，框架内存模型可以量化“数据流—框架内存用量”之间的关系，用户代码模型可以量化“数据流—用户代码内存用量”之间的关系。为了解决如何在代码未知的情况下量化用户代码内存用量与其输入数据的关系，我们在用户代码模型中设计了生命周期敏感的内存监控方法。在内存用量模型的基础上，我们设计了如何通过在小数据上运行应用来估算该应用在大数据集上的内存用量的参数估计方法。我们针对五个有代表性的 Hadoop MapReduce 应用做了内存用量估算实验，实验结果表明我们的估算方法的相对误差率在 20% 以内。

(3) 内存溢出错误的诊断方法及工具：为了帮助用户诊断内存错误，我们基于内存用量模型设计了一个内存分析器 Mprof。Mprof 可以自动准确建立应用静态因素与动态内存用量的关系，方法是重建应用数据流，重建用户代码内存使用历史信息，并对两者进行关联分析。Mprof 也包含定量诊断规则，这些规则根据应用静态因素与动态内存的关联关系来定位内存溢出错误相关的代码，错误相关的数据，以及不恰当的配置参数。Mprof 仅仅依赖于我们提升过的 task 的运行日志、动态数据流监控器以及 heap dumps，

不需要用户对代码做任何改动。我们在 20 个真实的 Hadoop MapReduce 内存溢出错误上对 Mprof 做了实验评估，结果显示 Mprof 可以有效地诊断出 MapReduce 应用中的内存溢出错误。

本文的研究结果不仅可以帮助用户和框架设计者更好地处理内存溢出问题，也对后续研究工作（比如研究其他框架的内存溢出问题）有借鉴意义。

6.2 进一步的工作

6.2.1 内存溢出错误实证分析方面

分析更多系统的内存溢出错误：当前我们主要研究的是通用处理框架（MapReduce 或 MapReduce-like）的应用。尽管我们研究的应用已经覆盖了文本处理，SQL 处理，机器学习，图处理等多个领域，还有很多运行在专门处理某一类数据的框架上的应用。比如 Pregel [Malewicz2010] 和 Powergraph [Gonzalez2012] 是专门用来进行大规模图处理和分析的框架，Apache Storm [Storm] 是专门用于流处理的框架。我们把研究这些非 MapReduce 框架上应用中的内存溢出错误作为我们未来的研究工作。

6.2.2 应用内存用量估算方面

内存用量在线预测：目前我们采用在小数据集上运行应用来预测应用在大数据上的内存消耗，我们希望能够在后续工作中能将当前估算方法扩展成在线预测方法，也就是在应用运行过程中预测 task 的内存用量。

缩小估算误差：目前的内存用量估算误差在 20% 左右，我们希望能建立更准确的模型来缩小估算误差，比如设计更准确的参数估计方法来减小数据倾斜等问题的影响。

在更多的框架上实现：尽管我们只在 Hadoop MapReduce 应用上测试我们的内存模型和内存用量估算方法，我们在后续工作中会考虑如何将模型扩展到其他框架，比如 Spark。

6.2.3 内存溢出错误诊断工具设计方面

诊断出高层语言里内存消耗高的操作：目前我们的诊断工具 Mprof 还不能将内存溢出错误定位到高层语言里面的操作，这是因为高层语言产生的应用的用户代码经过了高层库的翻译，而这个翻译过程各个库都不同，很难去建立统一的诊断模型。

适用于更多的框架：虽然其他 MapReduce-like 的框架，比如 Spark，数据流中的基本数据依赖关系与 MapReduce 数据流相同。但这些框架支持 pipeline，而且处理阶段可以将处理阶段组织成更灵活的 DAG 图。因此，下一步工作是将现有的诊断工具扩展到其他框架。

参考文献

- [FBD] Facebook 大数据：每天处理逾 25 亿条内容和 500TB 数据，
<http://www.leiphone.com/news/201406/0823-danice-facesbooks-data.html>
- [GoogleD] Google Processing 20,000 Terabytes A Day, And Growing.
<http://techcrunch.com/2008/01/09/google-processing-20000-terabytes-a-day-and-growing/>
- [YoutubeD] A Comprehensive List of Big Data Statistics.
<http://wikibon.org/blog/big-data-statistics/>
- [TwitterD] A Conversation On The Role Of Big Data In Marketing And Customer Service.
<http://www.mediapost.com/publications/article/173109/a-conversation-on-the-role-of-big-data-in-marketing.html>
- [BDValues]为决策支持带来价值大数据的 4V 理论。
<http://server.51cto.com/News-281578.htm>
- [Dean2004] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *6th Symposium on Operating System Design and Implementation (OSDI)*, 2004, pp. 137–150.
- [Ghemawat2003] Ghemawat S, Gobioff H, Leung S T. “The Google file system”, ACM SIGOPS operating systems review. ACM, 2003, 37(5): 29-43.
- [Hadoop] Apache Hadoop. <http://hadoop.apache.org/>
- [Isard2007] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in Proceedings of the 2007 EuroSys Conference (EuroSys), 2007, pp. 59–72
- [Zaharia2012] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012, pp. 15–28.
- [ApachePig] Apache Pig. <http://pig.apache.org>
- [ApacheHive] Apache Hive. <http://hive.apache.org/>
- [ApacheMahout] Apache Mahout. <http://mahout.apache.org/>
- [Gonzalez2014] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “GraphX: Graph processing in a distributed dataflow framework,” in *11th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014, pp. 599–613.

[SparkMLlib] MLlib: Apache Spark's scalable machine learning library.

<http://spark.apache.org/ml/>

[SparkSQL] Spark SQL: Spark's module for working with structured data.

<http://spark.apache.org/sql/>

[HDFS] Hadoop File System.

<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>

[OOMP1] Why the identity mapper can get out of memory?

<http://stackoverflow.com/questions/12302708/why-the-identity-mapper-can-get-out-of-memory>

[OOMP2] Will reducer out of java heap space.

<http://stackoverflow.com/questions/16116022/will-reducer-out-of-java-heap-space>

[MemUsageP1] hadoop mapper over consumption of memory(heap).

<http://stackoverflow.com/questions/15316539/hadoop-mapper-over-consumption-of-memoryheap>

[MemUsageP2] Hadoop - Can I set task memory limit lower than 2GB.

<http://stackoverflow.com/questions/13238098/hadoop-can-i-set-task-memory-limit-lower-than-2gb>

[MemUsageP3] Detailed dataflow in hadoop's mapreduce?

<http://stackoverflow.com/questions/19490723/detailed-dataflow-in-hadoops-mapreduce>

[OOMDiagnosisP1] Hadoop Streaming Memory Usage.

<http://stackoverflow.com/questions/17975335/hadoop-streaming-memory-usage>

[Li2013] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie, “A characteristic study on failures of production distributed data-parallel programs,” in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 963–972.

[Kavulya2010] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, “Analysis of traces from a production mapreduce cluster,” in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2010, pp. 94–103.

- [Morton2010] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, “Estimating the progress of MapReduce pipelines”, in *the 26th International Conference on Data Engineering (ICDE)*, 2010.
- [ARIA2011] A. Verma, L. Cherkasova, and R. H. Campbell, “ARIA: Automatic resource inference and allocation for MapReduce Environments”, In *the 8th International Conference on Autonomic Computing (ICAC)*, 2011.
- [Verma2011] A. Verma, L. Cherkasova, and R. H. Campbell, “Resource provisioning framework for MapReduce jobs with performance goals”, In *12th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2011.
- [Herodotou2011] H. Herodotou and S. Babu, “Profiling, what-if analysis, and cost-based optimization of MapReduce programs”, *PVLDB*, 4(11):1111–1122, 2011.
- [MAT] “Eclipse Memory Analyzer.” [Online]. Available: <http://www.eclipse.org/mat>
- [Cherem2007] S. Cherem, L. Princehouse, and R. Rugina, “Practical memory leak detection using guarded value-flow analysis,” in Proceedings of *the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 480–491.
- [Xie2005] Y. Xie and A. Aiken, “Context- and path-sensitive memory leak detection,” in Proceedings of *the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2005, pp. 115–125.
- [Jump2007] M. Jump and K. S. McKinley, “Cork: dynamic memory leak detection for garbage-collected languages,” in Proceedings of *the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007, pp. 31–38.
- [Xu2008] G. H. Xu and A. Rountev, “Precise memory leak detection for java software using container profiling,” in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 151–160.
- [HBase] <http://hbase.apache.org/>
- [HadoopTuning] Hadoop Performance Tuning.
<http://wiki.apache.org/hadoop/PerformanceTuning>
- [Tips4Hadoop] Cloudera: 7 Tips for Improving MapReduce Performance.
<http://blog.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/>

- [SparkTuning] Tuning Spark. <http://spark.apache.org/docs/latest/tuning.html>
- [Zaharia2011] Zaharia M, Hindman B, Konwinski A, et al. The datacenter needs an operating system[C]//Proceedings of the 3rd USENIX conference on Hot topics in cloud computing. USENIX Association, 2011: 17-17.
- [PACM] G. Ananthanarayanan, A. Ghodsi, Andrew Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. “PACMan: Coordinated memory caching for parallel jobs”. In NSDI, Apr. 2012.
- [Cascading] Cascading. <http://www.cascading.org>.
- [Chambers2010] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, Nathan Weizenbaum: “FlumeJava: easy, efficient data-parallel pipelines”. In PLDI 2010:363-375
- [Pike2005] Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan: “Interpreting the data: Parallel analysis with Sawzall”. Scientific Programming (SP) 13(4):277-298 (2005)
- [Chattpadhyay2011] Biswajesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghoo Kwon, Michael Wong: “Tenzing A SQL Implementation On The MapReduce Framework”. PVLDB 4(12):1318-1327 (2011)
- [DryadLINQ] Yu Y, Isard M, Fetterly D, et al. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In OSDI. 2008, 8: 1-14.
- [SCOPE] Chaiken R, Jenkins B, Larson P Å, et al. SCOPE: easy and efficient parallel processing of massive data sets. Proceedings of the VLDB Endowment, 2008, 1(2): 1265-1276.
- [PeriSCOPE] Guo, Z., Fan, X., Chen, R., Zhang, J., Zhou, H., McDirmid, S., ... & Zhou, L. (2012, October). Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE. In OSDI (pp. 121-133).
- [Condie2010] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, Russell Sears: MapReduce Online. In NSDI 2010:313-328
- [Bu2010] Yingyi Bu, Bill Howe, Magdalena Balazinska, Michael D. Ernst: HaLoop: Efficient Iterative Data Processing on Large Clusters. PVLDB 3(1):285-296 (2010)
- [Bu2012] Yingyi Bu, Bill Howe, Magdalena Balazinska, Michael D. Ernst: The HaLoop

- approach to large-scale iterative data analysis. VLDB J. (VLDB) 21(2):169-190 (2012)
- [Power2010] Russell Power, Jinyang Li: “Piccolo: Building Fast, Distributed Programs with Partitioned Tables”. In OSDI 2010:293-306
- [Melnik2010] Melnik S, Gubarev A, Long J J, et al. Dremel: interactive analysis of web-scale datasets[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 330-339.
- [ThemisMR2012] A Rasmussen, M Conley, R Kapoor, VT Lam, G Porter, A Vahdat: ThemisMR: An I/O-Efficient MapReduce. SOCC 2012
- [Hauswirth2004] Matthias Hauswirth, Trishul M. Chilimbi: Low-overhead memory leak detection using adaptive statistical profiling. ASPLOS 2004:156-164
- [Shaham2000] Ran Shaham, Elliot K. Kolodner, Shmuel Sagiv: Automatic Removal of Array Memory Leaks in Java. CC 2000:50-66
- [Tang2008] Yan Tang, Qi Gao, Feng Qin: LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages. USENIX Annual Technical Conference 2008:307-320
- [Xu2011] Guoqing (Harry) Xu, Michael D. Bond, Feng Qin, Atanas Rountev: LeakChaser: helping programmers narrow down causes of memory leaks. PLDI 2011:270-282
- [Morton2010] K. Morton and A. Friesen. KAMD: A Progress Estimator for MapReduce Pipelines. In ICDE, 2010.
- [ParaTimer] Kristi Morton, Magdalena Balazinska, Dan Grossman: ParaTimer: a progress indicator for MapReduce DAGs. SIGMOD 2010:507-518
- [Verma2011] Abhishek Verma, Ludmila Cherkasova, Roy H. Campbell: ARIA: automatic resource inference and allocation for mapreduce environments. ICAC 2011:235-244
- [Verma2011M] Abhishek Verma, Ludmila Cherkasova, Roy H. Campbell: Resource Provisioning Framework for MapReduce Jobs with Performance Goals. Middleware 2011: 165-186
- [Khoussainova2012] Nodira Khoussainova, Magdalena Balazinska, Dan Suciu: PerfXplain: Debugging MapReduce Job Performance. PVLDB 5(7):598-609 (2012)
- [Fang2015] L. Fang, K. Nguyen, G. H. Xu, B. Demsky, and S. Lu, “Interruptible tasks: Treating memory pressure as interrupts for highly scalable data- parallel programs,” in ACM SIGOPS 25th Symposium on Operating Systems Principles (SOSP), 2015.

[HadoopMailList] Hadoop mailing list.

<http://hadoop-common.472056.n3.nabble.com/Users-f17301.html>

[SparkMailList] Spark mailing list. <http://apache-spark-user-list.1001560.n3.nabble.com/>

[MRDesignPattern] D. Minerand, A. Shook, Map Reduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems. O'Reilly Media, Inc., 2012.

[TextMR] J. Lin and C. Dyer, Data-Intensive Text Processing with MapReduce, ser. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010. <https://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf>

[OOMCases] Real-world OOM Errors in Distributed Data-parallel Applications, <https://github.com/JerryLead/MyPaper/blob/master/OOM-Study.pdf>

[Cloud9] “Cloud9: A Hadoop toolkit for working with big data.” <http://lintool.github.io/Cloud9/>

[BrownBench] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, “A comparison of approaches to large-scale data analysis,” in Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2009, pp. 165–178.

[HadoopPB] <https://wiki.apache.org/hadoop/PoweredBy>

[SparkPB] <https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark>

[STXXL] STXXL: Standard Template Library for Extra Large Data Sets.”<http://stxxl.sourceforge.net/>

[TPIE] TPIE - The Templated Portable I/O Environment. <http://madalgo.au.dk/tpie/>

[Murray2013] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: a timely dataflow system,” in ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP), 2013, pp. 439–455.

[Yang2007] H. Yang, A. Dasdan, R. Hsiao, and D. S. P. Jr., “Map-reduce-merge: simplified relational data processing on large clusters,” in Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2007, pp. 1029–1040.

[Flink] Apache Flink. <https://flink.apache.org/>

[Malewicz2010] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in

- Proceedings of the *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010, pp. 135–146.
- [Gonzalez2012] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 17–30.
- [Storm] Apache Storm. <https://storm.apache.org/>
- [TencentOOM] “The OOM error in Count(distinct) in Tencent.” <http://download.csdn.net/detail/happytofly/8637461>
- [Xiao2014] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou, “Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs,” in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 44–53.
- [Zhou2015] H. Zhou, J. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin, “An empirical study on quality issues of production big data platform,” in *37th International Conference on Software Engineering (ICSE)*, 2015.
- [Gunawi2014] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Elazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, “What bugs live in the cloud?: A study of 3000+ issues in cloud systems,” in Proceedings of the *ACM Symposium on Cloud Computing*, Seattle (SoCC), 2014, pp. 7:1–7:14.
- [Li2014] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, “Tachyon: Reliable, memory speed storage for cluster computing frameworks,” in Proceedings of the *ACM Symposium on Cloud Computing (SoCC)*, 2014, pp. 6:1–6:15.
- [Nguyen2015] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, “FACADE: A compiler and runtime for (almost) object-bounded big data applications,” in Proceedings of the *Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 675–690.
- [Bu2013] Y. Bu, V. R. Borkar, G. H. Xu, and M. J. Carey, “A bloat-aware design for big data applications,” in *International Symposium on Memory Management (ISMM)*, 2013, pp. 119–130.
- [Vavilapalli2013] Vavilapalli, Vinod Kumar, et al. "Apache hadoop yarn: Yet another

- resource negotiator." Proceedings of *the 4th annual Symposium on Cloud Computing (SOCC)*, 2013.
- [Hindman2011] Hindman, Benjamin, et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." *NSDI*. Vol. 11. 2011.
- [Kwak2010] H. Kwak et al. What is twitter, a social network or a news media? In *WWW*, 2010.
- [Ganapathi 2010] A. Ganapathi, Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson, "Statistics-driven workload modeling for the cloud", In Workshops Proceedings of the 26th International Conference on Data Engineering (ICDE Workshops), 2010.
- [Khoussainova2012] N. Khoussainova, M. Balazinska, D. Suciu, "PerfXplain: Debugging MapReduce Job Performance", *PVLDB*, 598—609, 2012.
- [Singer2011] J. Singer et al, "Garbage collection auto-tuning for java MapReduce on multi-cores", In Proceedings of *the 10th International Symposium on Memory Management (ISMM)*, 2011.
- [JProfiler] Java Profiler.
<https://www.ej-technologies.com/products/jprofiler/overview.html>
- [GiniIndex] Gini Index. http://en.wikipedia.org/wiki/Gini_coefficient
- [OutlierAlgo] "What are outliers in the data?"
<http://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm>
- [PearsonCorr] Pearson product-moment correlation coefficient.
http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient
- [DominatorTree] Dominator tree. http://help.eclipse.org/mars/topic/org.eclipse.mat.ui.help/concepts/dominatortree.html?cp=44_2_2
- [Wang2004] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y. Wang, "Automatic misconfiguration troubleshooting with peerpressure," in *6th USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004, pp. 245–258.
- [Attariyan2010] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *9th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, 2010, pp. 237–250.
- [DeWitt1992] D. DeWitt, J. Naughton, D. Schneider, and S. S. Seshadri, "Practical skew handling in parallel joins", In the Proceeding of *the 18th Very Large Date Bases*

(VLDB), 1992.

[Kwon2012] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, “Skewtune: mitigating skew in mapreduce applications”, In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD), 2012.

[EHadoop] Enhanced Hadoop-1.2. <https://github.com/JerryLead/hadoop-1.2.0-enhanced>

[EMAT] Enhanced Eclipse MAT. <https://github.com/JerryLead/enhanced-Eclipse-MAT>

[PigManual] DISTINCT operator in Pig Latin.

<http://pig.apache.org/docs/r0.13.0/basic.html#distinct>

表 20 本文引用的现实世界的内存溢出错误用例

ID	现实世界的内存溢出错误	版本	URL
e01	Out of heap space errors on TTs	Hadoop 0.20.2	http://tinyurl.com/p9prayt
e02	pig join gets OutOfMemoryError in reducer when mapred.job.shuffle.input.buffer.percent=0.70	Pig	http://tinyurl.com/kb4zmot
e03	Kyro serialization slow and runs OOM	Spark	http://tinyurl.com/qbchkfp
e04	[Graphx] some problem about using SVDPlusPlus	GraphX	http://tinyurl.com/pb6857w
e05	Problems with broadcast large datastructure	Spark	http://tinyurl.com/osvc2dg
e06	Why does Spark RDD partition has 2GB limit for HDFS	Spark	http://tinyurl.com/qcugugs
e07	RDD Blocks skewing to just few executors	Spark	http://tinyurl.com/pzp4l3u
e08	Building Inverted Index exceed the Java Heap Size	Hadoop	http://tinyurl.com/ojf9npb
e09	memoryjava.lang.OutOfMemoryError related with number of reducer?	Hadoop	http://tinyurl.com/q6jomja
e10	OutOfMemory in “cogroup”	Spark	http://tinyurl.com/qaofbsc
e11	Understanding RDD.GroupBy OutOfMemory Exceptions	Spark 1.1	http://tinyurl.com/os6hbgo
e12	Hadoop Streaming Memory Usage	Hadoop	http://tinyurl.com/orfv3n3
e13	Hadoop Pipes: how to pass large data records to map/reduce tasks	Hadoop	http://tinyurl.com/phwdob4
e14	Hadoop Error: Java heap space	Hadoop 2.2	http://tinyurl.com/qy6wyj9
e15	OutOfMemory Error when running the wikipedia bayes example on mahout	Mahout	http://tinyurl.com/p3cj4ve
e16	Mahout on Elastic MapReduce: Java Heap Space	Mahout 0.6	http://tinyurl.com/na5wodj
e17	Hive: Whenever it fires a map reduce it gives me this error	Hive 0.10	http://tinyurl.com/p32aqfd
e18	OutOfMemoryError of PIG job (UDF loads big file)	Pig	http://tinyurl.com/ne6o6z3
e19	Writing a Hadoop Reducer which writes to a Stream	Hadoop	http://tinyurl.com/p46zupz
e20	MLLib ALS question	Spark 1.1	http://tinyurl.com/oa5eotk
e21	java.lang.OutOfMemoryError on running Hadoop job	Hadoop 0.18.0	http://tinyurl.com/odydwfx
e22	Why does the last reducer stop with java heap error during merge step	Hadoop	http://tinyurl.com/crb6q8
e23	MapReduce Algorithm - in Map Combining	Hadoop	http://tinyurl.com/ohcue2r
e24	how to solve reducer memory problem?	Hadoop	http://tinyurl.com/okq74kp
e25	java.lang.OutOfMemoryError while running Pig Job	Pig	http://tinyurl.com/ovpo8th
e26	A join operation using Hadoop MapReduce	Hadoop	http://tinyurl.com/b5m72hv
e27	Set number Reducer per machines	Cloud9	http://tinyurl.com/m4fo6wr
e28	trouble with broadcast variables on pyspark	Spark	http://tinyurl.com/nktoyp4

e29	driver memory	Spark	http://tinyurl.com/oa6bn5f
e30	RowMatrix PCA out of heap space error	MLlib 1.1.0	http://tinyurl.com/p382x4k
e31	Running out of memory Naive Bayes	MLlib 1.0	http://tinyurl.com/nwzq4sr
e32	GraphX does not work with relatively big graphs	GraphX	http://tinyurl.com/qj4fst5
e33	something about rdd.collect	Spark	http://tinyurl.com/ok2zkjn
e34	How to efficiently join this two complicated rdds	Spark 0.9	http://tinyurl.com/ppa9suv
e35	CDH 4.1: Error running child: java.lang.OutOfMemoryError: Java heap space	Cloudera 4.1	http://tinyurl.com/oggngsg3
e36	org.apache.spark.shuffle.MetadataFetchFailedException: Missing an output location for shuffle 0	Spark	http://tinyurl.com/pu3nfzc
e37	[0.9.0] MEMORY AND DISK SER not falling back to disk	Spark 0.9.0	http://tinyurl.com/nbajgc3
e38	Reducer's Heap out of memory	Pig 0.8.1	http://tinyurl.com/mftlvvv
e39	OOM writing out sorted RDD	Spark	http://tinyurl.com/oe9dj78
e40	Lag function equivalent in an RDD	Spark	http://tinyurl.com/ng23nl6
e41	Hadoop Pipes: how to pass large data records to map/reduce tasks	Hadoop	http://tinyurl.com/phwdob4
e42	OOM with groupBy + saveAsTextFile	Spark 1.1.0	http://tinyurl.com/pa7ells
e43	Efficient Sharded Positional Indexer	Hadoop	http://tinyurl.com/pdbcp7y
e44	OutOfMemory during Plain Java MapReduce	Hadoop	http://tinyurl.com/otq9ucs
e45	Hadoop UniqValueCount Map and Aggregate Reducer for Large Dataset (1 billion records)	Hadoop	http://tinyurl.com/q6vglsu
e46	spark aggregatebykey with collection as zerovalue	Spark	http://tinyurl.com/qh6nacx
e47	ORDER ... LIMIT failing on large data	Pig	http://tinyurl.com/q3ef6ak
e48	Out of memory due to hash maps used in map-side aggregation	Hive	http://tinyurl.com/qyeg9zc
e49	Reducers fail with OutOfMemoryError while copying Map outputs	MapR 3.0.1	http://tinyurl.com/ozavd4q
e50	news20-binary classification with LogisticRegressionWithSGD	MLlib 1.0.0	http://tinyurl.com/p8kw3pd
e51	fail to run LBFS in 5G KDD data in spark 1.0.1?	MLlib 1.0.1	http://tinyurl.com/newu7d7
e52	java.lang.OutOfMemoryError while running SVD MLlib example	MLlib 1.1.0	http://tinyurl.com/pb8lg2b
e53	MLlib/ALS: java.lang.OutOfMemoryError: Java heap space	MLlib	http://tinyurl.com/oxmjugf

作者简历及攻读学位期间发表的学术论文与科研成果

教育背景 博士生（计算机软件与理论） 2009 年 9 月——现在
中国科学院软件研究所
导师：魏峻 研究员

学士（计算机科学与技术） 2005 年 9 月—2009 年 7 月
武汉大学计算机学院

学术论文

- [1] **Lijie Xu**, Wensheng Dou, Feng Zhu, Chushu Gao, Jie Liu, Hua Zhong, Jun Wei. A Characteristic Study on Out of Memory Errors in Distributed Data-Parallel Applications. *In the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE 2015)*, Washington DC, USA, 2015.
- [2] **Lijie Xu**, Jie Liu and Jun Wei. FMEM: A Fine-grained Memory Estimator for MapReduce Jobs. *In the 10th International Conference on Autonomic Computing (ICAC 2013)*, 65-68, San Jose, CA, USA, 2013.
- [3] **Lijie Xu**, Wensheng Dou, Feng Zhu, Chushu Gao, Jie Liu, Jun Wei. Mprof: A Memory Profiler for Diagnosing Memory Problems in MapReduce Applications. *Submitted to 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2016 under review)*, 2016.
- [4] **Lijie Xu**, Jie Liu and Jun Wei. MapReduce Framework Optimization via Performance Modeling. *In the 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPS PhD Forum 2012)*, 2506-2509, Shanghai, China, 2012.
- [5] Feng Zhu, Jie Liu, **Lijie Xu**, Dan Ye, Jun Wei, Tao Huang. A Lightweight Evaluation Framework for Table Layouts in MapReduce Based Query Systems. *In the 17th Asia-Pacific Web Conference (APWeb 2015)*, Guangzhou, China, 2015.
- [6] Feng Zhu, Jie Liu and **Lijie Xu**. A Fast and High Throughput SQL Query System for Big Data. *Web Information Systems Engineering (WISE 2012)*, 783-788, Springer Berlin Heidelberg, 2012.

参与课题

- [1] 国家自然科学基金, "云平台并行数据流程序的中间数据管理优化技术" (批准号: 61202065)

致谢

六年多的研究生生涯是一个痛并快乐着的过程。痛的是前路漫漫、险阻不断，快乐的是能够探索未知、磨练心志。在这期间，有很多人给我提供了帮助，这些帮助既包括学术上的指导、工作上的支持、也包括生活上的关怀。在论文即将完成之际，我诚挚地、也很荣幸地向他们表达由衷的感激之情！

首先我要感谢我的导师魏峻研究员。魏老师给我创造了宽松的学习和研究环境，支持我自由地进行学术探索，同时也容忍了我的很多任性。从论文开题到最后完稿，魏老师不仅在大的方面（如研究方向、论文选题、论文逻辑组织方面）给予我充分的指导，也在每次博士生讨论班上提出细节上的修改意见，使得我能够走在正确的研究路上。从魏老师身上我不仅学到了研究方法，还感受到一个优秀的科研工作者所具备的责任感和献身精神。另外，魏老师也给予了我很多支持：比如支持参会和支持对外交流。

其次要感谢窦文生师兄，在我论文连续被拒，心灰意冷之时，窦师兄帮助我重新整理研究思路，重新定位论文的发出点和落脚点，使得文章最终能够顺利成稿。在投稿前夕，窦师兄在百忙之中抽出时间帮我字斟句酌，非常感谢，希望以后能继续合作。

感谢高楚舒师兄在论文修改、论文 rebuttal 以及出国开会事宜上提供的帮助。感谢刘杰师兄组织的博士生小组讨论，以及在研究思路和论文写作上提供的意见。感谢叶丹研究员在我学习阶段各个流程环节上提供的支持和帮助。

我还要感谢软件工程技术中心的其他师长。非常感谢黄涛、钟华、金蓓弘老师对我的关心和指导，他们对科学的热忱和敬业的态度始终感染着我。感谢吴国全、王伟、秦秀磊、陈伟、吴恒、王焘等师兄们，他们给我的工作和论文提出了很多有益的建议和帮助。感谢在读研期间一起工作和学习过的兄弟姐妹们：伍晓泉、严慧、伍海江、孙耀、徐继伟、王杰、赵薇、柯叶青、陈茜、汪静甜、杨鑫晟、王彦士、高强、陈昊、吴雨龙、亢良伊、崔光范等。特别感谢和我一起硕博连读的同伴王卅以及我的师弟兼室友朱锋，我们一起经历了前进过程中的种种艰险，始终是不离不弃的朋友。

感谢微软亚洲研究院系统组郭振宇、周虎成和钱正平研究员在我论文选题、研究方法和论文写作上提供的指导意见。与你们一起工作的六个月让我收获很多，不仅仅是学术上和工作上的。

感谢阿里巴巴淘宝数据挖掘与计算小组的各位同学：明风、雷飙、玄畅、一帝、木艮、佳米、岩岫等。虽然与你们经历了三个月非常忙碌的日子（一边写论文，一边忙着解决问题），但一起学习讨论、一起表演节目、一起泡温泉的日子已成为我的珍贵回忆。

感谢人生海海小组的小伙伴们：赵占平、许畅、赵鑫、张苇如、程瑶、高蕾、朱鑫，我们的友谊长存。感谢黄俊杰等一些好友在生活上提供的帮助。感谢远在海外的姜军、

谭笑、陈昂，对我的关心和支持。

感谢李晋飞老师教我弹吉他，使我在压力大、孤单寂寞冷的时候能够得到慰藉。

感谢匿名审稿人提出的各类修改意见，使我不断反思如何让论文组织的更好。

“岁月是一场有去无回的旅行，好的坏的都是风景”。虽然自己对整个研究工作和研究成果并不满意，但希望学到的研究方法能够帮助我今后做出更好的成绩。

最后，特别感谢我的父母和家人，你们 20 多年来的支持是我不断前进的动力，我爱你们！