申请上海交通大学工程硕士学位论文

基于计算特性的Spark内存自适应管理策略研究

|  |  |
| --- | --- |
| 学校代码： | 10248 |
| 作者姓名： | 赵耀 |
| 学 号： | 1130379091 |
| 第一导师： | 胡飞 |
| 第二导师： | 陈昊鹏 |
| 学科专业： | 软件工程 |
| 答辩日期： | 年 月 日 |

上海交通大学软件学院

年 月

A Dissertation Submitted to Shanghai Jiao Tong University

for Master Degree of Engineering

Based on the adaptive computing characteristics of Spark memory management strategy research

|  |  |
| --- | --- |
| University Code： | 10248 |
| Author： | Yao Zhao |
| Student ID: | 1130379091 |
| Mentor 1： | Fei Hu |
| Mentor 2: | Haopeng Chen |
| Field： | Software Engineering |
| Date of Oral Defense： |  |

School of Software

Shanghai Jiaotong University

Jun, 2008

**上海交通大学**

**学位论文原创性声明**

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 年 月 日

**上海交通大学**

**学位论文版权使用授权书**

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

**保密**□，在 年解密后适用本授权书。

本学位论文属于

**不保密**□。

（请在以上方框内打“**√**”）

学位论文作者签名： 指导教师签名：

日期： 年 月 日 日期： 年 月 日

**基于计算特性的Spark内存自适应管理策略研究**

摘 要

伴随着信息时代到来的是海量的数据资源，在被称为“大数据”时代的21世纪，数据的特点是：数据规模越来越大，数据种类繁多，数据价值密度低，以及数据处理的时效性低。目前处理大数据最为高效和经济的技术是分布式计算机技术，通过建立分布式集群，能让分布式系统获得堪比超级计算机的运算能力，同时还能获得巨大的存储能力。然而，要想设计好一个分布式计算机系统并不是那么的容易。Hadoop分布式框架的出现，使得对大数据的处理发生了质的转变，成为了分布式计算机技术发展史上的一个里程碑。随后，Spark分布式框架的出现，使得分布式系统的性能又上了一个台阶，不仅有着高容错、高可靠性和良好的鲁棒性等性能，而且在处理迭代式机器学习算法和交互式查询操作时，获得的性能比Hadoop框架要高出40倍左右。这都得益于Spark巧妙的设计和内存使用特性。

Spark可以将任务产生的中间结果存放到内存中，从而节省了从外部存储系统中存取数据的开销。但是，相对于海量的数据而言，内存是有限的，当内存中缓存的数据量减少时，Spark系统性能也随之降低。因此，是否能够高效利用内存资源，决定了能否提升Spark的系统性能。传统的人工手动调优方式，对开发人员的要求较高，而且取得的效果也不尽如人意。本文提出了一种基于Spark内存计算特性的自适应调优策略，并通过在Spark内部实现一个子系统来实现这一策略。Spark自适应调优主要分为三个方面，首先是数据序列化的自适应调优，其次是数据压缩的自适应调优，最后是垃圾回收的自适应调优。前两种调优方式，是根据Spark中集成的数据序列化和压缩算法针对不同工作集时表现出来的不同性能，来对当前系统所采用的序列化和压缩算法进行调整。为了达到自适应调优的目的，在任务运行时，系统会动态收集当前系统的实时信息，根据对这些实时信息的分析来对Spark系统所采用的策略进行调优。

关键词 大数据，分布式计算系统，Spark， 内存计算，自适应

**Based on the adaptive computing characteristics of Spark memory management strategy research**

ABSTRACT

With the advent of the information age is the massive data resources. In the 21st century which has been called the "big data" era, big data can be explained in four Vs: volume, increasing amount of data, from terabytes (TB) level to petabytes (PB) level and even to ten trillion gigabytes (ZB) level; variety, a wide range of data, including both traditional structured data also includes such as text, video, pictures and audio and other unstructured data, and the proportion of the rapid increase in unstructured data; value, low density value of the data, it is difficult to predict analysis, operational intelligence, decision support computing; velocity, large data processing issues become more prominent, timeliness is not guaranteed. Distributed computer provides a good platform with powerful computing capability, excellent scalability and fault-tolerant disaster recovery capabilities to handle large data. A well-designed distributed cluster’s computation power is far beyond any supercomputer in the world, and also can get more powerful computing capabilities by constantly adding new nodes for; in the case of multiple node failure, the system is still capable of normal operation; for failure node, if the node is able to recover, you can aid disaster recovery system before you get the result of the operation, if not recover, then the intermediate result of the operation will be transferred to another node, rather than be recalculated.

Spark is a leader in distributed computing framework, its use of in-memory computation features, in dealing with common iterative machine learning algorithms and interactive query operations, performance was far more than other similar frameworks. However, the memory capacity is the bottleneck Spark framework, when there is insufficient memory, its performance will be greatly affected, but so far no internal Spark optimization strategies for memory usage. Aiming at this problem, a dynamic collection workloads in the process of tasks running in the operation data, and then dynamically adjusts the operation of an adaptive strategy Spark memory adjustment strategy based on these data collected.

**Keywords** big data, distributed computer system, Spark, in-memory computation, adaptive

目录

[基于计算特性的Spark内存自适应管理策略研究 1](#_Toc438849190)

[Based on the adaptive computing characteristics of Spark memory management strategy research 2](#_Toc438849191)

[**ABSTRACT** 2](#_Toc438849192)

[第一章 绪 论 1](#_Toc438849193)

[1.1 研究背景与意义 1](#_Toc438849194)

[1.2 研究目的和内容 2](#_Toc438849195)

[1.3 研究意义 3](#_Toc438849196)

[1.4 论文结构 4](#_Toc438849197)

[第二章 Spark内存优化技术相关研究与分析 5](#_Toc438849198)

[2.1 分布式计算技术概述 5](#_Toc438849199)

[2.1.1 MapReduce计算框架 5](#_Toc438849200)

[2.1.2 Tez计算框架 5](#_Toc438849201)

[2.1.3 Spark计算框架 6](#_Toc438849202)

[2.2 Spark框架研究 6](#_Toc438849203)

[2.2.1 RDD 6](#_Toc438849204)

[2.2.2 Spark的Job生成 6](#_Toc438849205)

[2.2.3 Spark逻辑执行流程 7](#_Toc438849206)

[2.2.4 Spark物理执行流程 8](#_Toc438849207)

[2.3 分布式框架优化技术研究 9](#_Toc438849208)

[2.3.1 参数空间优化法 9](#_Toc438849209)

[2.3.2 历史数据优化法 9](#_Toc438849210)

[2.3.3 融入机器学习的混合优化法 10](#_Toc438849211)

[2.3.4 Spark优化策略 10](#_Toc438849212)

[2.4 本章小结 11](#_Toc438849213)

[第三章 系统设计 12](#_Toc438849214)

[3.1 SATS架构设计 12](#_Toc438849215)

[3.1.1 SATS架构 12](#_Toc438849216)

[3.1.2 SATS主要模块 12](#_Toc438849217)

[3.2 运行时数据收集模块设计 13](#_Toc438849218)

[3.2.1 Spark任务运行分析 13](#_Toc438849219)

[3.2.2 SATS运行时数据收集模块设计 15](#_Toc438849220)

[3.3 自适应决策模块设计 15](#_Toc438849221)

[3.3.1 Spark中内存的使用 16](#_Toc438849222)

[3.3.2 序列化自适应策略设计 16](#_Toc438849223)

[3.3.3 压缩算法自适应策略设计 18](#_Toc438849224)

[3.3.4 垃圾回收自适应策略设计 20](#_Toc438849225)

[3.4 参数优化模块设计 22](#_Toc438849226)

[3.4.1 参数值的初始设定 23](#_Toc438849227)

[3.4.2 模拟退火算法 23](#_Toc438849228)

[3.4.3 参数优化模块设计 25](#_Toc438849229)

[3.5 本章小结 26](#_Toc438849230)

[第四章 系统实现 27](#_Toc438849231)

[4.1 SATS系统总体架构说明 27](#_Toc438849232)

[4.1.1 Spark存储体系 27](#_Toc438849233)

[4.1.2 SATS在Spark框架中的作用 28](#_Toc438849234)

[4.2 运行时数据收集模块实现 29](#_Toc438849235)

[4.2.1 运行时数据及其特点 29](#_Toc438849236)

[4.2.2 运行时数据收集模块的实现 30](#_Toc438849237)

[4.3 自适应决策模块的实现 31](#_Toc438849238)

[4.3.1 序列化自适应策略实现 31](#_Toc438849239)

[4.3.2 压缩算法自适应策略实现 35](#_Toc438849240)

[4.3.3 垃圾回收自适应策略实现 36](#_Toc438849241)

[4.4 参数优化模块的实现 37](#_Toc438849242)

[4.4.1 优化参数说明 38](#_Toc438849243)

[4.4.2 参数优化模块实现 40](#_Toc438849244)

[4.5 本章小结 43](#_Toc438849245)

[第五章 系统测试与结果分析 44](#_Toc438849246)

[5.1 实验方法和目的 44](#_Toc438849247)

[5.2 实验环境搭建 44](#_Toc438849248)

[5.2.1 实验环境 44](#_Toc438849249)

[5.2.2 集群配置 44](#_Toc438849250)

[5.2.3 Hadoop安装 45](#_Toc438849251)

[5.2.4 Spark安装 45](#_Toc438849252)

[5.3 自适应决策模块测试 46](#_Toc438849253)

[5.3.1 序列化自适应算法测试与分析 46](#_Toc438849254)

[5.3.2 压缩自适应算法测试与分析 48](#_Toc438849255)

[5.3.3 垃圾回收自适应算法测试与分析 49](#_Toc438849256)

[5.3.4 模块集成测试 51](#_Toc438849257)

[5.4 参数优化模块测试 52](#_Toc438849258)

[5.5 系统性能整体测试 53](#_Toc438849259)

[5.6 本章小结 54](#_Toc438849260)

[第六章 总结和展望 55](#_Toc438849261)

[6.1 全文总结 55](#_Toc438849262)

[6.2 未来展望 55](#_Toc438849263)

# 绪 论

随着互联网和计算机科学的快速发展，数据信息逐渐呈现爆炸增长的姿态，这些数据信息产生自网络、企业、科研机构和其他一些与互联网相关的行业和组织，“大数据”是对这个信息爆炸时代的最好诠释。大数据中蕴含着极大的商业和科研价值，然而要想将这些隐藏在数据中的价值挖掘出来并不是一件容易的事。分布式计算机技术是现下处理大数据最常用的一种技术，一个良好的分布式集群的处理能力相当于一台超级计算机。作为分布式平台中的佼佼者，Spark分布式计算平台自Hadoop框架发展而来，充分利用了内存运算的优点在处理迭代式机器学习算法和交互式查询上，获得了比Hadoop高出40倍左右的运算性能，并具有高容错和快恢复等特性，开辟了大数据处理的新局面。

## 研究背景与意义

互联网和计算机技术的飞速发展，使得随之产生的数据信息爆炸式增长，21世纪也被称为“大数据”时代。大数据的特点有：数据量大（Volume），新浪微博、facebook等大型交互式网站每天要处理千万甚至上亿条的评论，而企业和科研机构产生在数据在日积月累下也逐渐达到了TB、PB甚至是ZB级别；种类繁多（Variety），既包括了传统的结构化数据，又包括了文本、音频、视频等非结构化数据，而且非结构化的数据比例还在不断增加；价值密度低（Value），往往在大量的数据上才能提取出一点有用的价值，而且预测分析、运营智能和决策支持上难以有效进行；处理速度慢（Velocity），数据处理的时效性难以保证，而实时的处理对商业决策等实时性要求高的数据服务至关重要。

要处理规模庞大的大数据，单靠少数几台计算机的运算能力远远不够，即便是使用超级计算机也不一定能够满足要求，而且由于价格高昂，也不是一种能够普遍使用的方法。分布式计算机技术将处于计算机网络中的计算资源整合成为一个计算机集群来使用，相当于获取了一台超级计算机，而且集群的计算和存储能力会随着新节点的加入而不断增加。因而，分布式技术成为了当前处理大数据性价比最高的技术并被广泛使用。

传统的分布式计算平台，如Hadoop，拥有高性能读写、负载均衡、高容错性等特点，用户在使用时，不必关心底层的实现就可以获得这些良好的系统特性。然而，使用传统的分布式框架处理来处理迭代式运算和交互式查询操作时，运算性能会大大降低。究其原因，这两种任务在运行中会产生很强的数据局部依赖性，即下一次的运算对上一次结果有很强的依赖性，而Hadoop的MapReduce模型会产生大量的中间结果，这些中间结果会被转存到磁盘或是其他外部存储系统中，当下一次的迭代或是查询进行的时候，系统又会将这些中间结果从外部存储系统中取出。由于磁盘的I/O操作代价远远大于内存的存取时间，导致了整个任务的绝大部分运行时间都消耗在了无意义的I/O操作上。针对这个问题，Spark分布式框架应运而生，它能够将任务中产生的中间结果缓存在内存中，这样一来下一次的运算所需的数据就不用到外部存储系统中获取，节省了大量的时间。使用了内存计算特性的Spark平台，在处理迭代式机器学习算法和交互式查询的任务时，运行速度比Hadoop要高出40倍左右。

然而，由于单个节点的内存资源是有限的，如果任务的规模过于庞大，产生的中间结果大部分都无法缓存在内存中，Spark的运算性能就会受到严重影响。目前Spark针对这个问题并没有对应的优化方案，只提供了一些参数，让使用者针对具体的任务进行调整，这无疑增加了Spark的使用难度。那么，让Spark针对不同的任务自动进行调整以获取最佳性能，推动内存计算技术的发展，是一个很有意义的尝试。

因此，在这一研究背景下，通过对常见任务的统计分类，针对不同类型的任务，让Spark任务在运行时通过动态的收集和分析系统信息，根据分析结果分别采取不同的优化策略以获取最佳的运行性能，这是一个具有广泛的实际应用和商业及科研价值的研究。

## 研究目的和内容

在目前Hadoop、Spark等大数据平台的研究中，主要是针对如何提高系统的计算性能和将各种机器学习算法用这些平台提供的API进行重写。目前基于Spark资源管理的研究较少，主要集中在各个节点的数据负载均衡，而对Spark内存优化的研究则更少。内存作为系统中的稀缺资源理应得到合理的应用，特别是对于Spark等基于内存运算的分布式平台，内存的合理应用尤为重要。目前所采用的优化方案是通过手工进行调整，这样的做法不仅过程复杂耗时，而且还需要有经验的专业人员来进行操作，最后得到的结果也不尽如人意。这一点给使用Spark的带来了不便。本文的研究目的是要找出一种比较合理的方法，将内存优化这件事交给计算机来做，而不需要人为参与，并且保证面对不同的工作集的时候，系统能够选择适当的调优方案。

在任务的运行过程中，系统的资源是在不断发生变化的，不同时刻消耗的系统资源，如内存资源、CPU资源等，是不同的。因此，一种事先选定的优化方案往往不会发挥太大作用，有时候甚至在经过初始时的性能提升之后反而还会造成性能的下降。由此本文的思路是，需要按照不同时刻系统资源的情况对初始时所采用的优化策略进行修改。这就是本文要进行研究的Spark自适应调优策略（SATS，Spark Adaptive Tuning Strategy），一种能够按照系统资源的分配情况来进行自动调优的方法。

要设计一个能够适时调整系统优化方法的算法，首先要进行数据的实时收集工作，并且对这些收集到的信息进行实时地分析，将分析的结果反馈到系统中。这样的做法有两个难点：第一是实时收集的数据也会占用一定的空间给系统的性能造成一些损耗；第二是从收集数据到对数据分析完成需要消耗一定的时间，这段时间造成反馈信息的非实时性，即当系统得到分析结果时，当前系统的状态又发生了转变。因此，要保证自适应算法所带来的空间消耗和反馈的滞后性。其次就是要分析具体的优化方式，Spark提供的优化方案主要有三个方面：第一是进行数据的序列化，序列化不仅能够节省内存，而且还能够加快系统垃圾回收的速度；第二是数据的压缩存储，压缩后的数据往往滞后原来数据的几分之一，甚至十几分之一，这将大大节省内存，但却会消耗更多的CPU资源；第三是优化系统的垃圾回收机制，因为Spark使用scala语言进行编写，而scala语言又运行在JVM上，这个为题就转换为了在Spark环境下针对JVM的垃圾回收优化的问题。

## 研究意义

本文所提出的自适应方法的研究意义主要体现在两个方面。首先是将自适应调优的思路应用到Spark的内存优化上，改变了传统的单纯依赖手工进行调优的方式；其次是提升Spark系统性能。一直以来，对大型系统的优化是业内难以解决且避无可避的问题。分布式系统的设计要考虑到性能、负载均衡、容错性等等诸多问题，是一个复杂的大型系统，而且会随着集群的扩大而变得越来越复杂。越复杂的系统优化起来越困难，有时候优化反而会造成系统性能的下降！Spark在诸多分布式框架中算是一个轻量级框架——核心代码只有5万行左右，而且其最主要的数据结构RDD的设计遵循了简单即美的原则。这些特点减少了Spark内存优化的复杂度，使得对其的优化变得相对简单了一些。Spark的内存计算特性使得其性能和内存的利用率有很大联系，系统的性能会随着缓存在内存中的数据的减少而下降。如何高效利用内存空间是提升Spark性能一个关问题。与Hadoop类似，Spark中的shuffle操作会产生很多的临时对象，这些临时对象占用了大量的内存空间。数据序列化和压缩算法能够为Spark节省大量内存空间，但是如何恰当地使用它们，需要进行精心地设计。本文提出了基于Spark计算特性的自适应算法，算法会在任务运行时收集数据，根据对这些数据进行分析来优化系统性能。

## 论文结构

本文的结构如下：第一章介绍研究背景、研究目的和内容和研究意义；第二章主要介绍一些常见的大数据平台、针对这些平台的优化技术和Spark内存优化技术的国内外研究现状；第三章阐述了Spark的自适应内存优化算法的架构设计，主要是针对数据序列化、数据压缩和垃圾回收优化三个方面的自适应算法设计；第四章将介绍介绍系统的具体实现，包括将Spark的序列化单位从一个RDD细化到一个分区；第五章将描述系统试验部分并对实验的结果进行分析；最后，第六章对全文做一个总结并对未来要进行的工作做一个展望。

# Spark内存优化技术相关研究与分析

本章主要研究Spark内存优化技术，旨在解决内存不足的情况下，Spark能够根据运行时的数据信息和内存使用情况，能够自动地调整相关策略，提高内存的使用效率。要想分析上述问题，首先要了解一些常用的内存优化策略以及它们的性能和适用情况。本章主要阐述了Spark常用的内存优化技术。

## 分布式计算技术概述

### MapReduce计算框架

MapReduce的概念源自于函数式编程语言，它将任务的处理过程分成两个部分：Map（映射）和Reduce（规约）。首先，Map操作会将一个指定的操作运用到待处理数据集中的每一个数据上。在此过程中，原始的数据集不会被改变，Map操作会产生一个新的数据集来存放处理后的结果，因而Map操作是会产生很多的小文件。多个Map操作是可以并行进行的，也就是说可以将一个数据集很大的任务分解成为多个数据量较小的任务来进行处理。在多核环境中，可以让这些任务并行执行，从而获得比原来更少的处理时间。其次，执行Reduce任务的线程在获取Map操作的结果后，会对这些结果执行“规约”操作，即按照指定的操作将接收到的结果进行归并。同样的，Reduce操作也是可以并行执行的，即可以让多个Reduce任务接收多个Map任务的结果。

MapReduce计算框架充分利用了系统多核资源，大大加快了任务的执行速度。根据MapReduce思想实现的分布式系统Hadoop提供了高性能，高并发，高可靠性，良好的负载均衡机制和容错机制。目前，MapReduce计算框架被广泛应用到大数据处理领域中，并取得了巨大的成功，成为当下最流行的分布式计算技术之一。但MapReduce在处理框架在处理迭代式运算和交互式查询时，性能稍显不足。同时，MapReduce的编程接口过于严苛死板，缺乏灵活性。

### Tez计算框架

Tez是针对Hadoop数据处理应用程序的分布式执行框架。与MapReduce不同，Tez采用DAG作业调度的方式，将多个有依赖的作业转换成为一个作业从而提升DAG作业的性能。在一些不适合应用MapReduce的场景，例如机器学习算法，Hadoop可以借助Tez来完成。Tez的目的是高度订制化，即让人们不必借助外部方式就能够顺利完成自己的工作。为了达到这个目的，Tez提供了更具表现力的数据流API，让用户能够形象的描述自己所要运行的DAG作业图。Tez灵活的输入—处理器—输出的运行模式对比与MapReduce的映射—规约，显得更加灵活。此外，Tez的数据类型无关、动态图重新配置、部署的简单性等特性，大大拓展了其应用领域。

### Spark计算框架

在大数据处理领域，根据处理时效性的不同，可将大数据框架划分了两种不同的类型：脱机型和实时型。Hadoop的MapReduce和Tez都属于前者，它们处理大数据的时效性都不高，无法为使用者提供实时的服务。在信息爆炸的21世纪，信息量急剧膨胀，谁能及时处理好这些数据信息，谁就能够抢占发展的先机。因而，对大数据的实时处理变得越来越重要。Hadoop虽然高效但依然存在一些问题，在处理实时性任务的性能上远远不如Spark。为了获得高性能，Spark简化了内存数据结构，通过将任务处理的中间结果缓存在内存中的方式，在处理迭代式运算和交互式查询的任务上，获得了超过Hadoop框架40倍以上的速度。此外，巧妙的设计还让Spark在高可靠性和高容错性上也不逊色于其他的分布式计算框架。

## Spark框架研究

### RDD

RDD（Resilient Distributed Dataset）是Spark内存计算框架的核心，它是一种抽象的数据结构，每一个RDD都是只读的，且都是产生自另一个RDD或者输入数据上的转换（transformation）操作。RDD中包含了用于存放数据的数据分区（partition）、元数据信息和对父RDD或是原始输入数据的一组依赖（Lineage）。Spark应用程序开发主要是针对RDD的操作来进行的。在Spark的官方文档中，RDD的操作可以分为两类：transformation和action。每个RDD或是输入的原始数据都可以通过一个transformation操作生成新的RDD，但是这个过程并没有触发任务的执行，也就是说通过转换操作得到的RDD中只包含了一些元数据信息和依赖关系，缺少数据分区。而actions操作则会触发任务的执行。

### Spark的Job生成

Spark应用程序开发主要是针对RDD的操作来进行的。在Spark的官方文档中，RDD的操作可以分为两类：transformation和action。每个RDD或是输入的原始数据都可以通过一个transformation操作生成新的RDD，但是这个过程并没有触发任务的执行，也就是说通过transformation操作得到的RDD中只包含了一些元数据信息和依赖关系，缺少数据分区。而action操作则会触发任务的执行，得到执行的结果数据信息。transformation操作建立起了每一个RDD之间的依赖关系，这些相关的RDD组合在一起构成了一张任务执行DAG（Directed Acyclic Graph）图，该图描述了这个任务的执行流程，这个过程称为Spark的逻辑执行流程。action操作触发了真正的Spark任务执行，计算得到了结果数据信息，这个过程称为Spark的物理执行流程。

### Spark逻辑执行流程

Spark的逻辑执行流程实际上就是如何建立RDD之间联系的过程，这个问题包含以下三个方面：

1）RDD自身的依赖关系。不妨将经过transformation操作之后的RDD称为child RDD，转换操作的原数据RDD称为parent RDD。RDD的依赖关系就是RDD是依赖一个parent RDD还是多个parent RDDs？

2）RDD x会有多少个partition？

3）RDD x和它的parent RDDs之间是什么依赖关系？是依赖parent RDD中的一个还是多个partition？

对于第一个问题，可以直接从transformation操作中获知，例如c = a.Union(b)，就可知RDD c依赖于RDD a和RDD b。第二个问题，RDD x中partition的个数一般由用户进行制定，若没有指定的话取max(numberPartitions[parent RDD1],…,numberPartitions[parent RDDn])。第三个问题较为复杂，要视具体的transformation而定，但总的来说partition之间的依赖关系大体可以分为三类：

1）1:1，即一个child RDD中的一个partition依赖parent RDD中的一个partition，这种关系也称为OneToOneDependency。

2）N:1，即一个child RDD中的一个partition依赖parent RDD中的多个partitions，并且每一个parent RDD中partition只能依赖child RDD中的唯一的一个partition。这种关系称为NarrowDependency。

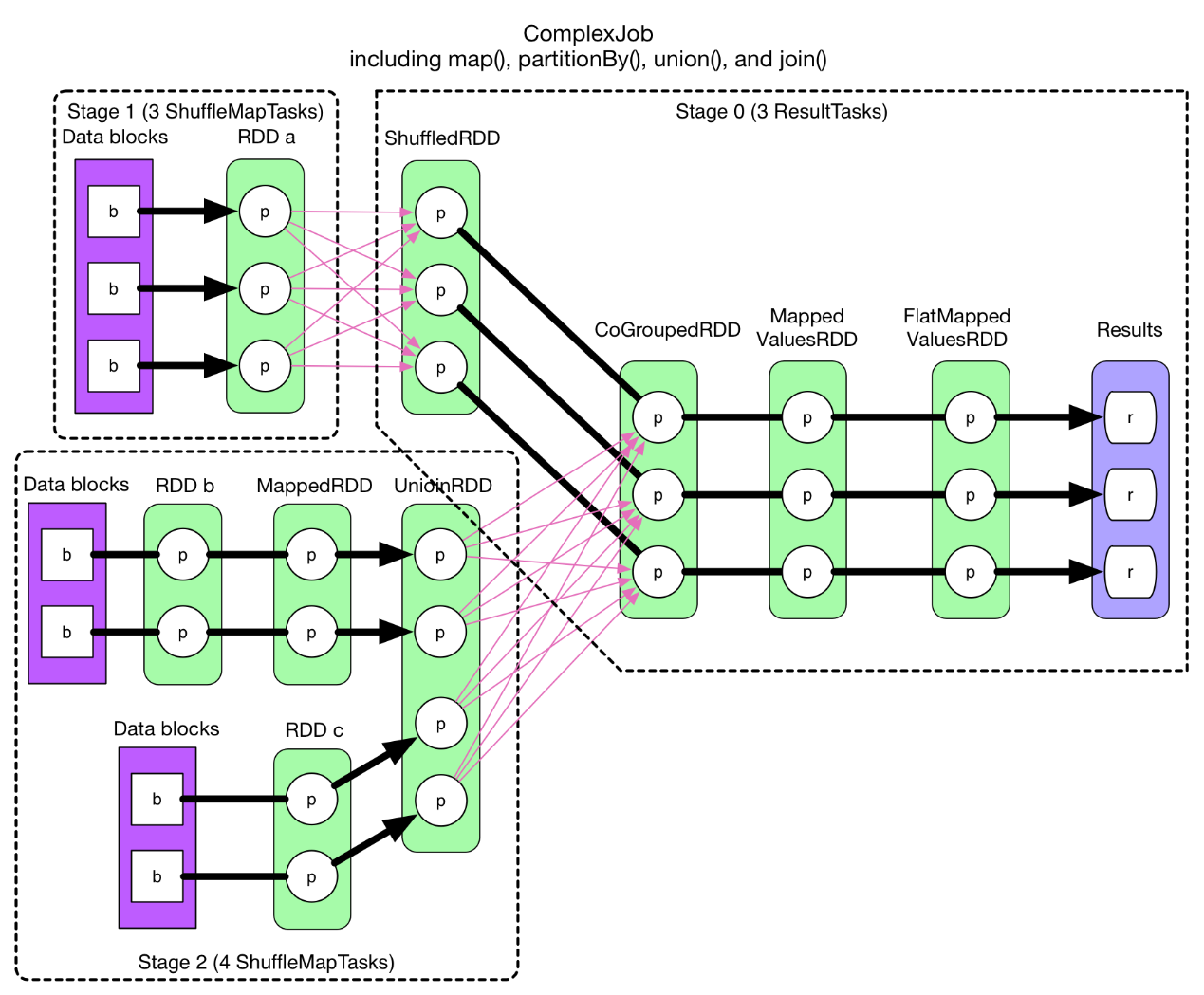
3）N:N，也是一个child RDD中的一个partition依赖多个parent RDD中的partition，但每一个parent RDD中的partition可以依赖多个child RDD中的多个partition。这也是一种NarrowDependency。

4）ShuffleDependency，child RDD中的每个partition依赖parent RDD中partition的一部分。

解决了这三个问题，Spark就可以顺利地生成任务执行流程图——DAG图了。

### Spark物理执行流程

Spark的物理执行流程是如何执行由transformation生成的任务执行图的过程。物理执行流程会先把逻辑执行流程中生成的DAG图，切分成多个小的字图，这些小的字图被称为stage。那么如何来划分stage呢？Spark采用了从后往前进行划分的方法，即从最后一个RDD开始向前寻找它所依赖的parent RDD。如果它们之间的依赖关系是NarrowDependency，那么继续往前寻找；如果是ShuffleDependency依赖就断开，断开的部分组成一个stage。按照同样的方法继续寻找，知道所有的RDD都加入了一个stage为止。图例是一个DAG图划分stage的例子。



Spark在进行任务提交的时候，也是从后往前依次提交stage。stage中最后一个RDD称为Final RDD，在计算Final RDD时，任务会向前找它所依赖的parent RDD获取数据，如果它的parent RDD中的数据也没有准备好，该parent RDD同样会向前寻找依赖的RDD获取数据，如此类推。当前一个RDD的数据准备好了时候就可以通过运算得到它的child RDD中的数据了。每个stage所产生的任务数量和其Final RDD的数量一致。当程序执行到action操作的时候，就会触发这些任务的执行。

## 分布式框架优化技术研究

每当系统的规模扩大10倍的时候，原来的系统就已经不能正常运行，需要进行重新设计。这意味着，随着系统的不断扩大，需要对平台进行优化，才能保证系统能够发挥最大的性能。分布式系统的优化方式有很多种，但基本上都是通过对参数的调整来实现，本节主要讨论当前一些分布式平台的调优方案。

### 参数空间优化法

比较常见的调优方法是通过待调优的大数据平台所提供的参数来实现，这样的调优方法称为参数空间优化法。具体来说，为了能够使系统在针对具有不同特性的工作集的时候，能够发挥其最佳性能，主流的大数据平台都提供了大量的参数供开发者进行选择。这些参数的所有可能组合组成的空间称之为参数空间（configuration space），举个例子来说，假设系统提供的可调整参数有N个，每个参数可能的值有M种，那么这个平台的参数空间中有个元素。得到这个参数空间之后，就可以针对某个具体的工作集来得到最优的参数配置方案：将参数空间中的每一种配置分别测试一遍，性能最优的结果所采用的配置就是最优配置。当处理相同或类似的工作集的时候就可以使用之前找到的参数配置，从而达到调优的目的。使用这种方法能够针对不同类型的工作集得到最佳的参数配置，但却耗时巨大。试想当一个新的工作集到达的时候，要用参数空间的每一种配置进行一一测试，这种做法的时效性无疑是非常低下的。因而这种方法通常会结合其他方法一起使用。

### 历史数据优化法

在任务的运行中会产生大量的任务日志，除此之外开发者也可以根据自己的需求收集一些特定的信息。利用这些历史信息来进行系统的优化是另一种比较常见的大数据优化方法。举例来说，在Quan Chen等提出的HAT系统中，会给每一个任务分配一个权值，再通过收集到的历史信息对任务的权值进行调整，最后根据权值的大小来确定哪些任务是会最终影响系统运行总时间的，并通过加速这些任务的执行速度来提升总体的运行速度。Spark生态圈中的Shark系统也提出了一种利用历史数据来加速系统运行的技术——PDE（Partial DAG Execution），该技术利用在先前执行的任务中收集到的相关信息，例如数据输入大小、输出大小、计算时间等，来优化当前的查询任务。历史数据优化法能够根据历史信息来预测任务的执行特征，根据这些特征来进行调优。其缺点就是根据历史数据来分析当前执行的任务特征有时候会存在很大的偏差，因为即便是相同的任务运行多次之后的结果也不尽相同。这是因为不同时刻系统资源的分配可能会有些差异，这会影响到计算框架中使用的算法对任务的划分和执行。

### 融入机器学习的混合优化法

在前面提到的参数空间优化法中，往往某一个分布式计算框架提供的参数是非常多的，直接导致了参数空间过于巨大。如果使用一一代入测试的方法，会消耗很多的时间，而且每当一个新的任务到达的时候都要使用这样的代入法进行测试，这对于一些实时性要求高的任务和资源稀缺的集群来说会造成很大的负担。要解决这个问题可以从参数空间入手。仔细分析就会发现，对于某一个具体的任务，参数空间中的很多配置方案明显都是不可行的。如何排除这些明显不符合条件的配置方案成为了简化参数空间的关键。

通常情况下有两种方法可以用于解决这个问题。其一是使用人工筛选的方法，针对常见的任务特征制定筛选条件。这样的做法对筛选条件的制定有很高的要求，如果筛选条件太少，那么能够甄别出来的任务特征差异过少，会导致筛选后的参数空间依然很大；如果筛选条件过多，那么还要考虑到这些筛选条件之间会不会产生冲突，以及由于筛选的条件过多过于苛刻而导致最优配置方案可能被剔除掉，造成过度过滤的后果。而且随着任务的运行，会产生越来越多的任务特征，还要针对这些新产生的特征指定新的筛选条件，这无疑会增加系统的维护成本。另外一种方式就是使用机器学习算法来解决这个问题，即选择一种合适的机器学习算法来快速找到参数空间中的最优配置。在Min Li等人提出的MRONLINE系统中，就通过采用一种智能爬山算法来快速找到MapReduce任务最优的配置方案。

### Spark优化策略

Spark是新型的基于内存计算的分布式框架，与传统分布式平台的优化方式不同的是，针对Spark的优化更加注重对内存使用的优化，能否高效地利用内存资源是决定Spark系统性能优劣的关键。Spark的优化可以从两个方面来考虑，一是减少运行时所需数据块的缺失所造成的“抖动”，一是避免频繁的GC操作影响系统性能。常用的优化手段有采用高效的序列化算法序列化数据，采用高效的压缩算法压缩数据，以及结合Spark系统自身的状态对JVM过程进行优化。序列化方法在减少数据占用空间的同时，还保证了数据在网络上的稳定传输，以及减少了GC操作次数。压缩数据使得要存储的数据降低到了原来的几分之一，甚至十几分之一，是节省内存空间的最有效的手段。

## 本章小结

本章首先介绍了Spark内存运算的过程和特点，其次详细讨论了序列化、压缩和内存垃圾回收三种主要的优化方法。最后分析了人工手动调优存在的弊端，提出了自适应调优的方法。下一章将针对本章提到的序列化、压缩和垃圾回收三种优化方法，设计出相对应的自动调优方法。

# 系统设计

自适应调优摒弃了手动调优的传统调优方式，在Spark运行过程中动态收集系统信息，根据对这些信息的分析结果对系统中的序列化方法、压缩算法和垃圾回收策略进行自动调整。本章将从这三个方面，设计相应的自适应算法。

## SATS架构设计

本节从两方面介绍自适应优化系统机构。3.1.1将介绍整个自适应系统的架构设计，3.1.2将描述架构的主要模块。

### SATS架构

本小节将介绍Spark自适应优化系统框架设计。Spark自适应优化系统是一个分层的结构的系统，在Spark应用程序运行的时候开始运行。当Spark任务运行完成时，SATS系统不会停止运行，而是会根据在Spark任务运行过程中收集到的数据对SATS系统的自身参数进行优化。当下一个Spark应用开始运行时，SATS会停止当前的参数优化活动，并保存当前的计算结果，然后开始处理新的Spark任务。

为了增加系统的灵活性以及提升系统存储空间的利用率，SATS系统对Spark原来的数据序列化和数据压缩基本单位进行了修改：将数据操作的基本单元由一个RDD改为了一个Partition。采取这样的做法使得当一个RDD中包含多个Partition的时候，SATS系统可以使用不同的数据序列化算法或者数据压缩算法对这个RDD中不同Partition进行处理，从而提高了SATS系统对存储空间的支配能力。

### SATS主要模块

SATS系统分为三个模块：运行时数据收集模块，自适应决策模块和系统参数优化模块。以下分别对这个模块进行相关说明：

运行时数据收集模块主要负责在Spark任务运行时收集系统和任务相关信息，例如系统CPU消耗、内存消耗、RDD的大小和计算耗时等等。这些收集到的数据必然会占用一定的存储空间，这对基于内存计算的Spark框架来说是一个不小的开销。如果将这些收集到的数据存放到外部存储设备当中，那么存取这些数据会造成额外的时间开销，而且这些数据的时间局部性相关性非常大（收集到的数据会在之后的数据分析模块中被用到）。因而SATS系统会将这些数据存储在内存当中，但为了节省内存的开销，系统会使用循环存储结构来进行存放。

自适应决策模块是SATS系统的核心模块，该模块负责分析运行时数据收集模块收集到的数据，并通过一套自适应决策算法根据收集到的数据对当前系统的存储策略进行调整。自适应模块中又包含了三个子模块：序列化算法自适应模块、压缩算法自适应模块和垃圾回收自适应模块，这三个子模块会在任务运行时，动态调整系统的数据序列化算法、压缩算法和垃圾回收策略。

参数优化模块会对运行时收集到的数据进一步进行分析，通过分析的结果对自适应决策算法中的相关参数进行调整。因为参数优化模块中涉及到一些比较耗时的机器学习算法，若该模块在Spark任务运行时执行，会消耗大量的系统资源从而有很大的可能使得Spark系统性能降低。因而，参数优化模块是在Spark应用运行完成后才执行，并且当有新的Spark应用任务到达的时候，该模块会停止当前的参数优化过程并保存相关分析结果，释放所占用的资源供Spark任务运行使用。

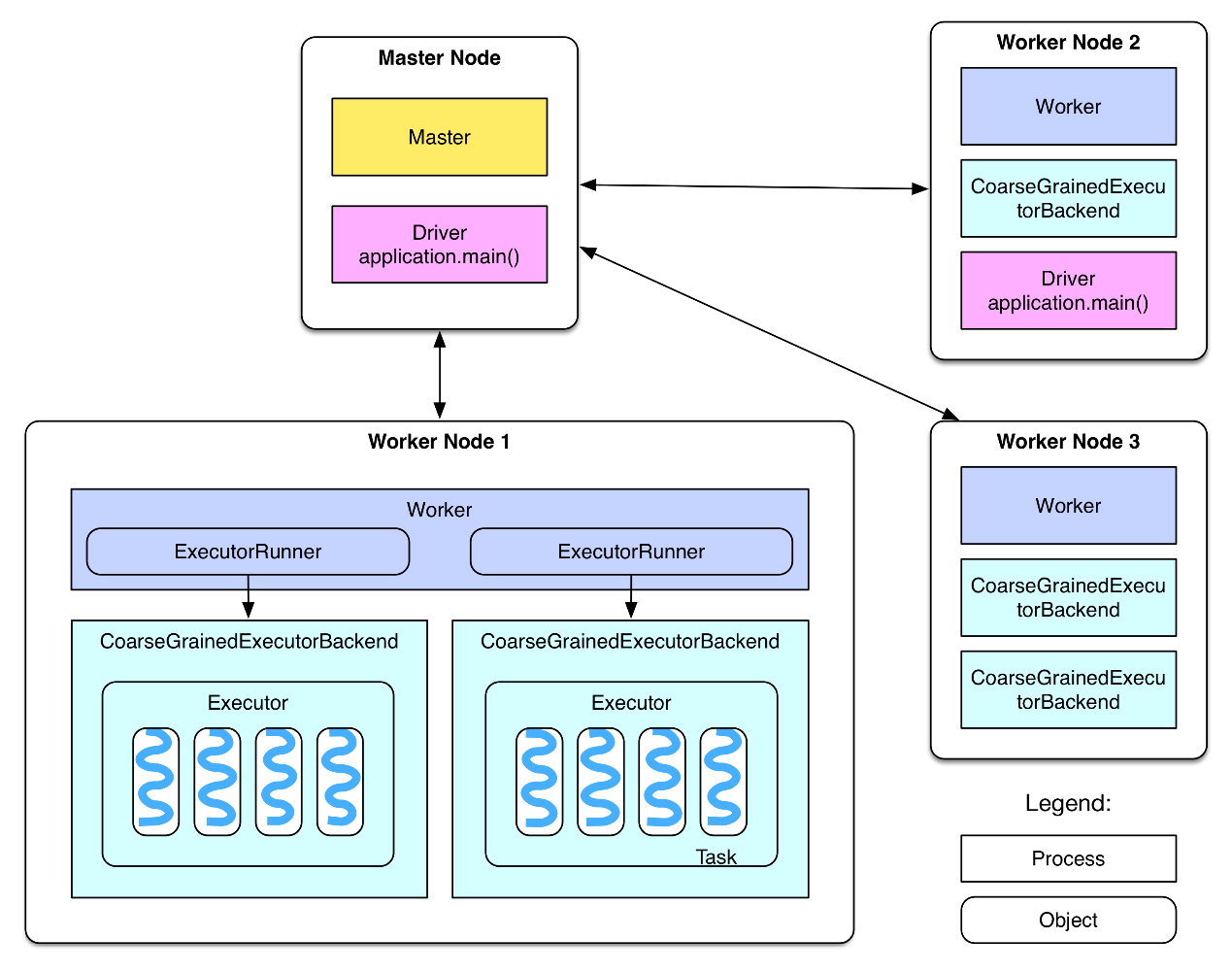
这三个模块在Spark任务运行过程中相互协作，共同完成对Spark系统自适应优化操作。

## 运行时数据收集模块设计

本节将介绍SATS系统运行时数据收集模块的设计，运行时数据收集模块主要负责在Spark任务运行过程中收集相关数据，这些相关数据的收集和存储涉及到之后的自适应决策模块是否能够良好地运行。

### Spark任务运行分析

Spark框架模型在第二章中已做详细描述，在此不再赘述。当Spark程序运行到action算子的时候，Spark任务就会开始执行。这里需要说明的是，在Spark中一共有两种算子，一种是transformation算子，一种是action算法。常见的transformation算子有map、reduceByKey、sort等，每个transformation算子都会生成新的RDD，这类算子不会触发Spark任务的执行。当程序执行到这类算子的时候，Spark会根据各个RDD之间的依赖关系构建DAG图。常见的action算子由collect、take等，当Spark程序执行到action算子的时候，就会触发任务的执行。关于这两类算子的用法说明在Spark的官方文档中有详细说明，在此就不做详细说明了。

Spark的任务部署图如图xxx所示，Spark采用的是经典的Master-Slaver架构。在集群中选择一个节点作为Master，其他节点作为Worker（Slaver），driver应用程序运行在Master上。Master将运行的Spark应用程序分成多个tasks，再将这些tasks按照一定的算法分配到各个Worker上。每一个Worker可以包含多个CoarseGrainedExecutorBackend，但是每个CoarseGrainedExecutorBackend属于不同的Spark应用程序。每一个CoarseGrainedExecutorBackend进程会保持一个Executor对象，这个Executor对象持有一个线程池。当有任务到达的时候Executor会从线程池中取出一个空闲的线程来执行这个任务。如果没有空闲的线程，那么这个任务会被加入到WaitingTasks队列当中等待。数据的收集工作发生在任务运行时，即当有任务在Executor对象中被执行时。

### SATS运行时数据收集模块设计



从3.2.1节中的分析中可以看出，Spark任务执行过程是在Executor类中完成的。数据收集模块流程图如图xxx所示，SATS系统收集收据的开始时间就是Executor开始执行第一个任务的时间，直到最后一个任务结束为止，收集数据的过程才算结束。为了设计上的简易性，SATS将收集数据操作看做是一个特殊的任务，当Executor开始第一个任务时，这个特殊任务会占用一个线程而启动；当最后一个任务结束时，该任务释放掉持有的线程，结束数据收集。这样的做法使得不仅让系统的资源消耗到达最低（只占用了一个线程），而且还大大地简化了数据收集模块的设计，即不用开辟单独的接口专门来进行数据收集，而使用Executor中的接口就可以了。

## 自适应决策模块设计

本节将介绍SATS核心模块——自适应决策模块的设计，该模块涉及到了三个方面的优化设计：数据序列化、数据压缩和垃圾回收。前两个方面在Spark系统中同属于Storage层次的内容，因而在自适应算法的设计上有一定的联系性。

### Spark中内存的使用

在Spark中，开发人员可以通过参数spark.driver.memory和spark.executor.memory来配置driver节点和executor（slave）节点的内存大小。但是，在进行数据缓存的时候，系统所能用到的内存空间，远远没有这么大。在2.3节时提到，Spark是运行在JVM虚拟机上的。在介绍Spark的缓存原理之前，首先来介绍一下JVM对内存的使用，以便于更好地了解Spark的缓存机制。JVM动态运行时区域如图xxx所示，在JVM中用来缓存数据的区域只有堆区，而且还不是堆区的全部空间。JVM堆原本是用来存放运行时产生的对象，Spark对JVM堆的机制进行了修改，使其能够用来存放RDD数据。开发人员通过参数spark.storage.memoryFraction来设置缓存区域的大小，其默认值为0.6，即使用堆区60%的空间来缓存RDD数据，剩余的40%用来存放运行时产生的临时对象。当这两个区域空间被用完时，系统会进行空间回收。用来存放临时变量的JVM堆，会通过JVM虚拟机的垃圾回收机制进行空间回收。Spark用来缓存数据的区域则使用LRU（Least Recently Used）策略进行回收。

### 序列化自适应策略设计

(1)Spark序列化技术分析

序列化主要有两个方面的作用。首先，java对象在网络上的传输需要通过序列化来实现。序列化能够将对象转换成为字节流在网络上进行传输，接收端接收到传输的字节流之后通过反序列化对传输的对象进行重构。其次，序列化对于节省内存空间和提高垃圾回收效率有重要的意义。多个同一类型的java对象经过序列化之后可以转换成为一个字节数组，这种方式不仅节省了内存空间，而且减少了垃圾回收时所需要遍历的java对象个数。因此，正确地应用序列化技术可以对Spark性能带来很大的提升。

Spark内置了两种序列化算法：Java和Kryo，Java序列化算法是java语言自带的，使用起来比较灵活，只需让需要被序列化的类继承Serializable 接口即可。Java序列化算法虽然使用起来比较灵活，但其缺点就是序列化的速度过慢，现在主流的序列化方法在序列化速度和压缩率上普遍都比Java序列化方法要好。Kryo序列化算法要比Java序列化算法高效很多，但是在使用的时候，需要对要被序列化的对象所属的类进行注册，如果没有注册的话，有可能会发生异常。目前，Spark已经自动将本身类在Kryo上进行了注册。但是，并不支持对用户自定义类的注册，在使用时，用户需要自己进行注册。Kryo存在的问题就是比较消耗CPU资源。Spark在默认情况下是使用Java序列化算法来对数据进行序列化，这里的序列化包括了两个方面：首先是闭包序列化，目前Spark只提供了Java序列化方法对闭包进行序列化；在经过闭包序列化之后，Spark会对序列化后的数据闭包再进行一次序列化，在这个过程中有两个序列化方法可以选择，即Java序列化方法和Kryo序列化方法。由此可见，目前所能做的优化集中在了第二次序列化过程中，也就是在什么情况下对两种序列化方法进行切换。

进行序列化调优的目的是为了加快任务的运行速度，Kryo序列化方法虽然比较节省空间，但是相对的，却增大了对CPU的消耗，而CPU的消耗会降低任务的运行速度；另一方面，Java序列化方法虽然对CPU的消耗很小，但其序列化后的数据大小却比Kryo序列化后的数据大得多，导致能够被缓存在内存中的数据量下降，增大了系统从外部存储系统存取数据的概率，也会影响到任务的执行速度。因而，对于如何选择合适的序列化方法，就必须综合考虑CPU消耗和内存消耗两方面的情况。

(2)序列化自适应算法设计



那么应该采取什么样的策略进行调优呢？首先来考虑序列化算法切换的条件问题。通过刚刚的讨论可以看出，Java序列化算法的优势在于其对CPU消耗比较少，而Kryo算法的优势在于序列化后的数据所占用的空间少。在Spark官方文档给出的调优方案中提到：节省内存空间能够更多地缓存数据，而其根本目的是为了加快任务的运行速度。显然节省空间只是一种手段，而加快任务的执行速度才是调优的目的。CPU的高消耗会直接性地导致任务的运行速度变慢。相较而言，缓存数据变少给Spark任务运行带来的影响并不是直接性的。具体来说，Spark的缓存策略使用的是LRU（Least Recently Used）算法：当Spark任务运行所需的数据不在内存中的时候，LRU算法会将缓存在内存中最近未被使用的数据块清理掉，替换成为从外部存储系统中刚刚读取到的（或是重新计算得到）任务所需数据块。当系统使用Kryo序列化算法时，内存中能够缓存的数据块要比Java序列化算法多得多。这样就大大降低了数据块的Miss率，减少了内存的抖动，从而加快了任务的执行效率。由此分析可知，缓存数据的减少只是导致了数据Miss率的增多，任务所消耗的时间有很大的概率会增加，但却不会直接性地导致任务的运行速度变慢。由此可以得出的结论是：在进行序列化调优时，首先要考虑到CPU的消耗情况，其次再考虑能够缓存数据的多少。

其次要考虑的是资源阀值设置问题。算法初次运行时，将系统的CPU阀值初始设定为90%，当CPU的使用率超过这个值的时候，首先判断当前系统所使用的序列化算法是不是Java序列化算法，如果不是，则将当前的序列化算法切换为Java序列化算法。缓存阀值的设定会稍微复杂一些，Spark将JVM堆空间分成两个部分来使用，一个部分用于缓存RDD数据，另一部分用来存储系统运行时产生的对象。这两个部分的比例通过参数spark.storage.memoryFraction来进行设置，因此，在此设置了两个内存阀值，一个表示整个JVM堆使用阀值，一个表示JVM堆中用来缓存RDD数据内存容量的阀值。为了算法的简洁性，将这两个阀值上限都设置为90%。同时将阀值的下限设置为40%，当内存使用低于这个值得时候将当前的序列化算法设置为Java序列化算法。算法的流程图如图xxx所示。

### 压缩算法自适应策略设计

压缩算法能够将数据压缩到原来的几分之一甚至十几分之一，正确地应用压缩算法能够很大地节省空间，使得能够存放到内存中的数据量大大增加，从而提高任务的执行效率。本节讨论如何在Spark任务运行中采用自适应策略选择适当的压缩算法来节省内存空间，从而提高任务的执行效率。

1. Spark压缩算法分析

|  |  |  |  |
| --- | --- | --- | --- |
| 算法名称 | 压缩速度 | 解压速度 | 特点 |
| Snappy | 250MB/s | 500MB/s | 速度快，压缩率高 |
| LZF | 150MB/s | 200MB/s | 速度最慢，压缩率最低 |
| LZ4 | 400MB/s | 1GB/s | 速度最快，压缩率最高 |

注：，压缩率越低节省空间越多

Spark一共内置了三种压缩算法：Snappy、LZF和LZ4。有关这三种压缩算法的对比如表xxx所示。这三种压缩算法各有优缺点。Snappy压缩算法着重于加快压缩速度，而不考虑压缩率（压缩后的文件体积和原文件体积的比例）和兼容性。其单核压缩速度达到了250MB/s，解压缩速度为500MB/s。LZF算法着重于减小压缩后数据的体积，因而具有很高的压缩效率，相比于Snappy能够节省更多的空间。但是，LZF的压缩速度却不是很快，单核压缩速度为150MB/s，解压速度为200MB/s。LZ4在这三种压缩算法中，是压缩品质最好且压缩速度最快的。其单核压缩速度为400MB/s，解压速度更是达到了GB/s的级别。为了保证压缩的高品质，LZ4的压缩效率并是不很高。在Spark实际运行中，需要根据任务数据的不同来选择最优的压缩算法，以使得系统的性能得到最大的提升。压缩算法对不同的数据集所表现出来的性能大不相同，那么就要根据数据类型的不同采取不同的压缩算法，以达到最优效果。

1. 压缩算法自适应策略设计



在内存比较充足的情况下，首先考虑采用LZ4压缩算法，不仅能够获取较高的压缩速度，还能够保证数据的正确率。随着Spark任务的运行，用于缓存RDD的内存空间逐渐减少。这时候就可以使用Snappy算法，在保证一定压缩速度的同时，减少压缩数据所占用的空间。当内存资源紧缺，频繁发生GC操作，甚至是内存溢出的时候，可以将压缩算法切换为LZF，进一步减小压缩后的数据体积。算法的工作流程如图xxx所示。

### 垃圾回收自适应策略设计

相较于Hadoop等传统大数据处理框架，Spark的最大的优势在于能将计算过程中产生的中间结果存储在内存中，以便于对数据进行重复使用，减少从磁盘取数据的I/O开销。但当内存不足以存放所有的中间结果的时候，JVM会对内存中的对象进行回收。JVM的垃圾回收策略将近期不使用的java对象释放掉，这意味着JVM要将所有的对象都要遍历一遍。由此可见，过于频繁的垃圾回收会导致系统性能下降。本节将讨论Spark的内存回收策略。

Spark使用scala语言编写，scala语言运行在JVM上，那么针对Spark的垃圾回收优化与JVM的垃圾回收优化有很大的关系。本节主要讨论如何使用自适应的方法针对Spark垃圾回收进行优化。

1. JVM垃圾回收策略分析

Spark为开发者提供了不同的存储级别（Storage level），在进行具体应用的开发时，可以根据具体情况进行调整。例如，存储级别MEMORY\_ONLY表示任务运行过程中产生的中间结果只能存储在内存中，如果内存中存放不下则将多余的数据直接丢弃！当这些被丢弃的数据在接下来的计算中要被用到时，Spark会根据相关RDD中的依赖关系重新计算这部分数据。除此之外，还可以使用MEMORY\_DISK\_ONLY存储级别，在这种级别下，系统会将内存中无法缓存的数据存放到磁盘中，当任务需要这些数据的时候就可以从磁盘中获取，而不必重新计算。

Spark使用JVM堆中的部分空间进行数据的缓存，在默认情况下，Spark将使用60%的堆空间用于数据缓存，另外40%的空间用于存放JVM运行时产生的新对象。开发人员可以根据自己所开发的Spark应用进行相应的调整，比如，如果JVM垃圾回收过于频繁或者发生内存溢出，说明用于存放运行时产生对象的空间过小，那么可以适当的将其比例调大；反之，可以适当的将缓存数据所占的空间比例调大。

1. 垃圾回收自适应策略模块设计



在java运行时状态区中，JVM堆被分成两个部分：新生代与老生代。新生代中存放存活时间比较短的java对象，老生代中存放存活时间比较长的对象。而新生代区进一步又可以分为Eden，Survivor1，Survivor2三个更小的区域。JVM运行过程中产生的新生对象会被存放到Eden与Survivor1中，当这两个区域的空间不足以存放所有的新生对象时，会发生一次Minor GC操作。Minor GC会将新生代前两个区域中的存活对象拷贝到Survivor2中，然后交换Survivor1和Survivor2中的内容。在此过程中，如果Survivor2的空间不能存放前两者的对象，那么这些对象会被存储到老生代区中，如果老生代中剩余的空间也不能存放这些对象，那么将发生一次Major GC。相对于Minor GC，Major GC是相当耗时的，因为在进行Major GC时，JVM虚拟机会将整个内存中的对象都遍历一遍，以找到不活跃的对象进行回收。

Spark中垃圾回收调优的目的是要将长期存活的对象存放到老生代中，而且要尽量保证新生代区能足够大以存放新生的对象，这样可以避免任务运行时由于回收临时变量导致的Major GC操作。为了达到这个目的，较为常用的调优手段有：

1）通过运行时收集GC信息，判断垃圾回收是否频繁发生。如果再一次任务开始之前发生过了过多的Major GC，说明JVM堆中用来存放临时变量的空间过少，那么可以将这部分空间的比例调大。

2）在收集到的GC信息中，如果老生代区域剩余空间不多，那么可以降低缓存数据所占空间的比例。因为获取较高的执行效率才是调优的目的，而缓存数据只是实现这个目的的一种手段。

3）如果发生过多的Minor GC，而鲜有Major GC发生，则意味着新生代中Eden区域所占空间过小。这种情况下，可以将Eden区域所占的空间比例增大，以减缓Minor GC的发生。Eden区域到底调整为多大比较合适，可以根据运行时的任务数量进行估算，例如，当从HDFS上读取数据时，如果需要执行4个任务，每个任务占用一个数据块，由于从HDFS上读取到的数据经过解序列化和解压缩后的大小为原来的2~3倍，每一个数据块的大小为64MB，那么执行这四个任务所需要的空间大小约为：4×3×64MB。

针对垃圾回收的自适应算法流程图，如图xxx所示。

## 参数优化模块设计

本节主要介绍参数优化模块的设计，该模块与SATS系统性能的优劣有着密切的关系。本文采用混合优化方式对Spark系统中的参数进行优化，将使用模拟退火算法来对参数空间进行寻优，找到近似最优的参数配置。该算法会在3.4.2中进行介绍。

### 参数值的初始设定

在自适应决策模块中，SATS使用三个优化算法来对存储策略进行调整，这些算法中会涉及到一些常量参数，这些参数的取值会很大地影响算法的性能。参数优化模块主要负责的是利用运行时收集到的数据对自适应决策模块中所用到的一些算法的参数进行优化。在SATS系统第一次启动的时候，因为还没有可利用的历史数据来对调优算法中的参数进行优化，因此这个时候系统所采用的参数是根据Spark应用程序员开发过程中积累下来的经验得到的参数值。显而易见，这些通过经验得到的参数值在面对不同的工作集的时候不一定能够提升系统性能，有可能还会让系统的性能降低。因此，参数的优化工作显得尤为重要，根据不同的工作集可以得到不同的优化参数。SATS系统会收集这些参数的值，当相同或是类似的任务达到的时候，选择对应的优化参数就能够到达优化的效果。

### 模拟退火算法

在进行参数配置选择的时候，有经验的开发者可以根据自己的经验来进行恰当的配置。但是，相对于没有大数据框架开发经验的开发人员来说，想要找到一组最优配置，往往是比较困难的。一般的做法，是经过反复的测试找到合适的配置。如果想要获得最优配置，往往只有使用穷举法，即将参数空间中的所有参数配置组合都实验一遍之后，得到最优配置。这种做法的时效性是相当低下的。模拟退火算法是一种随机寻优算法，与贪心算法和爬山算法不同，它是一种启发式的搜索算法，有一定的概率能够找到全局最优解。模拟退火算法为全局择优提供了更好的时效性，如果参数设置得当，其效率是远远超过穷举法的。

模拟退火算法模拟了物理退火的加温、等温和冷却过程，其中加温过程增强了物体内部的粒子运动，消除了系统原先的不均匀状态，算法在这个过程获取候选解空间；等温过程模拟系统的状态总是自发地向能量较少的方向变化，当能量达到最小的时候，系统达到平衡的状态，这个最优状态就是算法的最优解；冷却过程是物体内部粒子能量逐渐减少的过程，总是会出现某些粒子的能量首先降低，达到“稳定”的状态，这就相当与寻找最优解的过程，最优值取最稳定的粒子的能量值。算法的数学模型如下所述：

在温度T时，分子停留在状态r满足Boltzmann概率分布：

其中表示分子能量的一个随机变量，E(r)表示状态r的能量，>0位Boltzmann常数。Z(T)为概率分布的标准化因子：

在同一个温度T下，选定两个能量，有

当温度相同时，分子停留在“稳定”状态（能量较低）的概率比停留在“不稳定”状态（能量较高）的概率要大。

模拟退火算法之所有能够寻找到最优解，是因为当算法找到局部最优解时，算法有一定概率p接受下一个状态。这个规则称为Metropolis准则，可用公式表示为：

整个算法的流程图如图xxx所示：



### 参数优化模块设计

参数优化模块主要有两个功能，一个功能是在任务运行前找到与当前任务相匹配的历史数据，从这些历史数据中找到最优的配置。另一个功能是在任务运行之后优化自适应决策模块中自定义的阈值参数。下面进行详细说明：

在2.2节中已经分析了Spark的任务运行过程，当程序中出现action算子的时候，Spark才会执行任务。在此之前，Spark会根据各个RDD的依赖关系建立DAG任务执行图。在一个Spark程序中可以有多个action操作，每一个action操作都会生成一张DAG图。系统通过对比两个DAG图的相似性，找到与当前即将运行的DAG图相似的DAG图，将找到的DAG图的配置作为其配置。

在自适应决策模块中使用了一些预先设定好的阈值参数，这些阈值参数用来指定数据序列化算法、压缩算法和垃圾回收策略的临界转换条件。对于这些阈值的设定，刚开始只能根据开发人员的经验值来进行设置，这样的做法会使得系统遭遇一个“冷启动”过程，即由于初始参数设置的不合情理导致了系统性能不高。为了能够更好的设定这些初始阈值，自适应决策模块会在Spark系统的不断运行中分析收集到的执行过了的应用程序的历史信息，通过分析结果来进行阈值设置。

采用模拟退火算法在Spark的参数配置空间中找到近似最优解，是一种将机器学习算法和参数优化结合起来的调优方法。

因为参数优化模块运行在全部的Spark任务运行完成之后，参数优化模块的运行不会影响Spark应用程序的运行性能。当一个Spark应用程序运行完成之后，系统就得到了这个Spark应用程序的数据集，为了得到针对这个应用程序的最优化参数，参数优化模块会调整优化算法中所使用到的参数来试探运行这个Spark应用程序。然后从所得到的多组参数配置中选择性能最优的配置，作为这个Spark应用程序的最优配置。在同样或是类似的任务到达的时候，SATS就会选择这组参数配置，这样较之于第一次运行这个Spark应用程序，系统会获得更高的性能。

## 本章小结

本章主要介绍了SATS系统的设计思想和各个模块的思路，通过运行时数据收集模块收集到的数据，利用这些收集到的数据，自适应决策模块根据序列化优化算法、数据压缩优化算法和垃圾回收优化算法，对系统的存储策略进行调整。自适应决策模块的三个算法所用到的参数，可以在Spark程序运行完成之后，通过参数优化模块进行优化，已得到最优的算法参数配置。下一次，当相同或类似的Spark应用程序到达的时候，SATS系统选择这个最优化参数就可以获得更高的运行性能。

# 系统实现

本章将介绍SATS系统的具体实现细节和原理。SATS系统分为三个子模块：运行时数据收集模块，自适应决策模块和参数优化模块。其中数据收集模块和自适应决策模块是在Spark任务运行时开始运行，而参数优化模块是在所有Spark任务运行结束后运行。自适应优化模块是SATS系统的核心模块，在4.3中将会对其进行重点阐述。

## SATS系统总体架构说明

本节主要针对SATS系统进行总体架构上的说明，具体描述了SATS系统在Spark框架中的作用。SATS系统主要是在数据的存储上进行一系列的优化操作，为了能够更好地说明SATS系统在Spark框架中的作用，本节首先会对Spark的存储架构进行介绍。之后，本节将会描述SATS系统在Spark框架中的作用。

### Spark存储体系

Spark的存储模块Storage的类图关系如图xxx所示，Spark任务运行的结果通过BlockManager管理的DiskStore和MemoryStore存放到磁盘和内存中，其中存放到磁盘中的操作称为checkpoint，存放到内存中的操作称为cache。



Spark将数据的存储级别分成了以下五种：MEMORY\_ONLY，DISK\_ONLY，MEMORY\_AND\_DISK，MEMORY\_SER，MEMORY\_AND\_DISK\_SER，其中MEMORY\_ONLY表示将任务结果数据存放到内存中。当内存不足以存放所有数据的时候，多余的数据会被直接丢弃，当这些被丢弃的数据在计算中要被用到时会重新进行计算。如果开发者选择MEMORY\_AND\_DISK选项，Spark任务运行的结果会先往内存进行存储，对于内存存放不下的数据，会被存放到磁盘中，当这些存放在磁盘的数据在计算中要被用到时，系统会从磁盘中进行存取。添加了“\_SER”后缀的存储选项表示，在对数据进行存储之前会使用序列化方法对数据进行序列化，这样做的好处是使得要存放的数据体积变小了，但是数据的序列化和解序列化却会消耗一定的时间。

### SATS在Spark框架中的作用

如图xxx所示，展示了SATS系统在Spark框架中与其他子系统之间的关系。从图中可以看出，Executor子系统执行任务所需的数据会直接从Storage子系统中获取。但是在数据的存储上，SATS系统会在任务开始运行后，实时地对系统所采用的策略进行调整。

## 运行时数据收集模块实现

运行时数据模块主要负责在SATS系统运行期间收集优化相关的数据，并对这些数据进行存储。4.2.1中首先对要收集的数据进行说明，之后的4.2.2将详细说明模块的实现。

### 运行时数据及其特点

在单个节点中，系统的存储能力和计算能力是主要的系统资源。对于Spark这类基于内存计算的分布式框架来说，内存的容量是限制系统性能的重要瓶颈，外部存储系统的大小虽然也对系统性能有所影响，但远远不如内存的影响大。这是因为CPU访问内存的速度比访问外部存储设备的速度通常要快100倍以上。在SATS系统中主要考虑CPU和内存资源对Spark系统性能的影响，因此，CPU和内存的实时消耗情况就成为了主要收集的数据信息。

1. 内存数据收集

Spark运行在JVM上，因此它所使用的内存只是JVM所管理的内存中的一部分。具体来说JVM运行时数据区分为五个部分，如图xxx所示，Spark用来缓存数据的区域是JVM堆区，参数spark.storage.memoryFraction用来指定用来缓存数据的JVM堆空间所占的比例。该参数的默认值为0.6，表示使用60%的JVM堆空间来缓存数据，另外的40%用来存放JVM运行时产生的java对象。在此，SATS系统将内存的使用情况分为两类：一类是用来存放java运行时产生变量的JVM堆空间的使用情况，另一类是JVM堆空间中用来缓存数据的空间的使用情况。这两类空间都会对Spark的性能造成影响，当用于存放运行时java对象的JVM堆的空间使用率过高的时候，Minor GC操作就会频繁发生（GC操作只会发生在用来存放java对象的JVM堆空间上），Full GC操作发生的概率也会增大，从而导致系统运行速度变慢；当用来缓存数据的JVM堆空间使用比例过高时，发生Spark任务运行数据缺失的概率就会增大，Spark会采用一定的策略（默认是LRU）对JVM堆中的数据进行替换，这就造成了内存数据的“抖动”，同样也会影响系统的运行性能。因而，数据收集模块会收集这两类内存的使用情况。

****

1. CPU数据收集

之所以要收集CPU资源的使用情况情况，主要是为了在之后的自适应决模块中为序列化自适应算法和压缩策略自适应算法提供数据。在这两个自适应算法中，CPU使用情况是用来进行算法选择决策的一种重要参数。

1. JVM信息收集

Spark运行在JVM上，JVM的性能调优对Spark系统性能的提升影响非常大。对JVM的调优主要包括两个方面，首先是通过JVM的运行信息来调整用于缓存和用于运行时创建对象的JVM堆空间的比例（通过参数spark.storage.memoryFraction来实现）；其次是对JVM中的参数进行调整。在Spark中可以通过JVM相关配置参数-verbose:gc –XX:+PrintGCDetails –XX:+PrintGCTimeStamps将有关的GC信息打印出来。

### 运行时数据收集模块的实现

SATS系统各模块与Spark的关系如图xxx所示，运行时数据收集模块（Runtime Data Collect Module）会从Spark中获取所需数据。

数据收集模块的实现，是通过在Spark的任务处理相关的类中插入一些监听代码，并通过这些代码来获取任务运行时的一些数据。在原生态的Spark中，数据序列化和压缩是以RDD为基本单位的，SATS系统将这个基本单位从RDD改为了partition。这样的做法增加了序列化和压缩算法在使用上的灵活性，即可以使用不同的序列化和压缩算法来处理同一个RDD中不同的partition。为了区别出具体某一个partition是用什么序列化和压缩算法来处理的，需要能识别每一个partition的唯一ID，这个ID在Spark中是用blockId来表示的。blockId由rddId和partitionId组成，若是shuffle操作的话还涉及到shuffleId。经过这样的改造后，在Spark的RDD模块中维护两个Hash表，一个用来存放blockId到所使用序列化算法的映射，一个用来存放blockId到所使用压缩算法的映射。为了节省内存空间，在设计上使用序列化算法对象的hashCode来代替序列化算法本身。

为了说明数据收集的过程，接下来将介绍Spark数据序列化和压缩的过程。Spark采用数据驱动模式，当一个任务所需的数据收集完成的时候，该任务就会开始执行。任务运行完成后得到的结果会通过Actor消息传送机制传送到driver进程。首先来说明数据序列化的过程，Spark的任务由Executor对象来执行，一个Executor对象持有一个线程池，当有任务到来时Executor对象会从线程池中取出一个空闲线程来执行该任务。这首先需要对任务所需的任务进行解序列化。Spark对数据进行了两次序列化：闭包序列化和常规序列化，闭包序列化所使用的序列化方法只Java序列化算法，而常规序列化的序列化算法有两种：Java序列化算法和Kryo序列化算法。任务执行完成后得到的结果也要经过这两次序列化，先进行闭包序列化，在进行常规序列化。系统会在这个时刻进行数据序列化信息的收集。数据压缩的过程发生在I/O操作时候，Spark会根据配置文件选择将不压缩或是选择何种压缩算法对数据进行操作。

需要说明的是，Executor类中并没有直接提供rdd的相关信息，有关的信息被包装到了一个ExecutorBackend引用当中。为了避免对原生Spark接口进行改动而带来的不必要的工作量，在开发过程中放弃了采用改动原有接口进行传参的方式，采取了反射技术来提取该引用中rdd的相关信息。这种方式是比较低效的，但却节省了不少的工作量。

## 自适应决策模块的实现

本节将介绍自适应决策模块（ADM）的实现，该模块主要进行数据序列化、数据压缩和垃圾回收三个方面自适应算法的实现。4.3.1将介绍序列化自适应算法的实现，4.3.2会描述数据压缩自适应算法的实现过程，之后的4.3.3将阐述垃圾回收自适应算法的实现。

### 序列化自适应策略实现

1. 两种序列化方法的性能对比

从图xxx中可以看出，Kryo序列化算法比Java序列化算法更能节省时间，图xxx显示了Kryo比Java序列化算法更能节省存储空间。

1. 数据序列化具体过程

Spark数据序列化主要发生在两个阶段，一是任务执行完成时对生成数据的序列化，序列化过程分为闭包序列化和常规序列化，序列化后的数据经由Actor系统发送至driver。一是在进行数据读取的时候会发生解序列化过程，在这个过程中，当前执行的任务会读取所需的数据。读数据的过程比较复杂，以下进行详细描述：

从2.2对Spark框架的分析中可知，发生数据读取的方式，主要有两种，一种是在stage内的数据读取，这种读取方式比较简单，通常是对读取的数据进行简单处理之后生成新的RDD数据。一种是在两个相邻stage之间的数据读取，即进行shuffle操作时的数据读取，基于Hadoop框架衍生而来的框架都会有shuffle操作带来极大性能开销的问题，Spark也不例外。在shuffle操作过程中，reducer会从ShuffleMapTask中生成结果中读取所需数据，那么reducer的每一个partition对会对这些数据产生依赖，相应的依赖类型在2.2.3中进行了详细介绍，在此不再赘述。将这些依赖看做是一种映射关系的话，shuffle过程有两种映射方式：Hash Based Shuffle和Sort Based Shuffle，Hash Based Shuffle是数据根据Hash的结果，把各个partition的数据写到单独的文件中。这种方式的缺点是，当生成的partition文件过多的时候，会产生很多的磁盘文件。如要对这么多的临时文件进行读写操作，那么就要分配大量的内存，给GC操作带来压力。而Sort Based Shuffle会根据操作类型的不同对数据进行排序，排序带来的好处就是能够让多个ShuffleMapTask的结果写到同一个文件中，减少了临时文件的数量，缺点是排序过程非常耗时。

在Spark1.2之后，系统默认的shuffle操作类型从Hash方式变为了Sort方式。在进行序列化自适应的时候，数据是以数据块为单位进行读取。那么，在读取每一个数据块的时候，通过之前建立的数据块—序列化算法的哈希表来获取每个数据块对应的序列化方法进行解压缩操作。

1. 算法设计

在收集完数据后ADM模块会根据收集到的数据采用如图xxx所示的算法，对Spark所使用的序列化方法进行调整。

|  |
| --- |
| **Algorithm: Adaptive Serialization** |
| 1. if cpuUsg > ***cpuThr*** do 2. ***cpuOTIMES*** += 1 3. if ***cpuOTIMES*** == 3 do 4. curSer = JavaSer 5. ***cpuOTIMES*** = 0 6. endif 7. else 8. ***cpuOTIMES*** = 0 9. if memUsg > ***memRUpperThr*** || memCacheSize >***memCacheUpperThr*** do 10. curSer = KryoSer 11. elif memUsg<***memRLowerThr*** || memCacheSize<***memCacheLowerThr*** do 12. curSer = JavaSer 13. endif 14. endif |

参数说明：

cpuThr——CPU的使用阈值

cpuOTIMS——记录CPU连续超过阈值的次数

memRUpperThr——非缓存RDD内存使用上限

memRLowerThr——非缓存RDD内存使用下限

memCacheUpperThr——缓存RDD的内存使用上限

memCacheLowerThr——缓存RDD的内存使用下限

cpuUsg——当前系统CPU的利用率

memUsg——已使用的非缓存RDD的内存大小

memCacheSize——已使用的缓存RDD的内存大小

curSer——当前使用的序列化方法

JavaSer——Java序列化算法

KryoSer——Kryo序列化算法

算法中考虑了两个主要影响系统性能的因素：CPU使用率和内存使用率，在序列化自适应算法的设计中已经阐述了算法的工作原理，在此就不在赘述。算法每隔1s，会根据当前的系统资源状况对序列化算法进行调整。值得注意的是，由于环境和系统因素，在某一时刻，CPU的使用率会急剧升高，这种情况并不是CPU使用率过高，因而不必对算法进行切换。为了避免这种状况，算法默认在侦测到CPU使用率连续3次超过阈值的时候（在算法中使用参数cpuOTIMES来记录），才会对序列化算法进行切换（参数cpuOTIMES用来记录CPU利用率连续超过阈值的次数）。

### 压缩算法自适应策略实现

|  |
| --- |
| **Algorithm: Adaptive Compression** |
| 1. if memUsg < ***memLowerThr*** do 2. curCodec = “” 3. elif *memUsg* >***memLowerThr*** && *memUsg* <***memMiddleThr*** do 4. curCodec = “lz4” 5. elif memUsg > ***memMiddleThr*** && memUsg < ***memUpperThr*** do 6. curCodec = “snappy” 7. else 8. curCodec = “lzf” 9. endif |

参数说明：

memUsg——已使用内存大小

memLowerThr——内存使用低，

memMiddleThr——内存使用高，

memUpperThr——内存使用上限，表示系统对内存的消耗达到瓶颈

curCodec——当前系统所使用的压缩算法

压缩算法自适应策略如图xxx所示，算法中设置了三个阈值：memLowerThr、memMiddleThr和memUpperThr，用来判断内存的使用级别。内存使用量没有超过memLowerThr，表示内存使用量低，这时候没有必要使用压缩算法对缓存数据进行压缩存储。超过了memLowerThr，但低于memMiddleThr表示缓存的数据开始增多，这时候使用LZ4算法，该算法能在节省能存的同时保证较快的压缩速度。当内存使用值超过memMiddleThr，低于memUpperThr时，表示内存的使用已经达到一个较高的程度，这时候需要进一步节省内存空间，在这个阶段采用压缩率更高的Snappy算法。当内存使用量超过memUpperThr时，表示内存使用已经达到了瓶颈，在这个阶段内存嫉妒消耗，GC操作频繁发生，还有产生内存溢出的可能，这时候使用压缩率最低的LZF算法。

### 垃圾回收自适应策略实现

|  |
| --- |
| **Algorithm: Adapive Garbage Collection** |
| 1. if totalGCVec > ***gcUpperThr*** do 2. reduceCacheRatio() 3. elif totalGCVec < ***gcLowerThr*** do 4. increaseCacheRatio() 5. endif 6. if fullGCVec > ***fullGCUpperThr*** do 7. increaseNewRatio() 8. elif fullGCVec < ***fullGCLowerThr*** do 9. reduceNewRatio() 10. endif 11. if minorGCVec > ***minorGCUpperThr*** do 12. increaseSurvivorRatio() 13. elif minorGCVec < ***minorGCUpperThr*** do 14. reduceSurvivorRatio() 15. endif |

参数说明：

totalGCVec——总GC操作速度（次/秒）

fullGCVec——full GC操作速度（次/秒）

minorGCVec——minor GC操作速度（次/秒）

gcUpperThr——GC操作上限，超过表示GC操作过于频繁

gcLowerThr——GC操作下限

fullGCUpperThr——full GC操作上限

minorGCUpperThr——minor GC操作下限

函数说明：

increaseCacheRatio——增大参数spark.storage.memoryFraction的值

reduceCacheRatio——降低参数spark.storage.memoryFraction的值

increaseNewRatio——增大JVM参数-XX:NewRatio（增大老生代所占比例）

reduceNewRatio——降低-XX:NewRatio（增大新生代所占比例）

inceaseSurvivorRatio——增大新生代中Eden区所占比例

reduceSurvivorRatio——降低新生代中Eden区所占比例

垃圾回收自适应策略如图xxx所示。在之前的设计分析中得出了两个关于调优的结论：CPU过高的消耗会直接导致任务运行速度变慢，而Spark缓存在内存的数据量变少则是增大了数据块的miss率，会间接地导致任务执行速度变慢。因而，垃圾回收自适应策略首先要考虑的就是GC操作的信息，如果GC操作过于频繁，说明JVM堆中用于存放运行时临时变量的空间过少，应将其比例增大，在算法中通过函数reduceCacheRatio降低spark.storage.memoryFraction的值来实现；其次才考虑用于缓存数据的JVM堆空间大小，如果GC操作不是那么的频繁，则算法尝试增大用于缓存数据的堆空间大小，在算法中通过函数increaseCacheRatio增大spark.storage.memoryFraction的值来实现。

因为Spark是运行在JVM上的，因此可以通过优化JVM的相关参数来提升系统整体性能。在3.3.3中提到，JVM堆分为新生代区和老生代区，在进行调优的时候要尽量保证存活时间长的对象被存放到老生代区中。在任务运行时，如果full GC发生得过于频繁表示老生代区过小，可通过increaseNewRatio来增大其比例；反之，通过reduceNewRatio来降低其比例。而新生代区又可以分为Eden、survivor1和survivor2三个区域，在进行GC操作的时候，Eden和survivor1中到达一定生存时间的对象会被转存到survivor2中（如果survivor2中存放不下则转存到老生代区中）。之后交换survivor区的名字，即原来的survivor1变为了survivor2，原来的survivor2变成了survivor1。在这里需要说明的是，如果Eden区过大而survivor区过小（两个survivor区的大小是相等的），那么在进行GC操作时，如果Eden区中幸存的对象由于体积过大不能存放到survivor区中，则会被转存到老生代中。这会导致老生代区很快被填满，进而发生非常耗时的full GC操作，导致系统性能下降，这显然不是开发者想要见到的。因而，如果minor GC频繁发生，可以将Eden区占整个新生代区的比例增大，在算法中通过increaseSurvivorRatio函数来实现；反之，降低Eden区所占比例，通过算法中的reduceSurvivorRatio函数来实现。

## 参数优化模块的实现

本节介绍参数优化模块的具体实现，参数优化模块主要是用来对数据序列化和压缩算法自适应策略中的阈值参数进行调整，以及对Spark中的配置参数组合进行寻优。主要应用机器学习算法从参数空间中，选取最优的配置方案。

### 优化参数说明

|  |  |
| --- | --- |
| **配置参数** | **默认值** |
| **内存配置参数** |  |
| spark.driver.maxResultSize | 1g |
| spark.reducer.maxSizeInFlight | 48m |
| spark.shuffle.file.buffer | 32k |
| spark.shuffle.memoryFraction | 0.2 |
| spark.io.compression.lz4.blockSize | 32k |
| spark.io.compression.snappy.blockSize | 32k |
| spark.kryoserializer.buffer.max | 64m |
| spark.kryoserializer.buffer | 64k |
| spark.serializer.objectStreamReset | 100 |
| spark.broadcast.blockSize | 4m |
| spark.storage.memoryFraction | 0.6 |
| spark.storage.memoryMapThreshold | 2m |
| spark.storage.unrollFraction | 0.2 |
| **CPU配置参数** |  |
| spark.driver.cores | 1 |
| spark.executor.cores | 1 |
| spark.cores.max | NULL |
| spark.task.cpus | 1 |

以上是Spark中有关内存和CPU的可调控参数，这些参数对Spark性能的影响很大，之后会对这些参数进行说明。

1. 内存参数

首先来说明与shuffle过程相关的参数，主要有两个参数：spark.shuffle.file.buffer和spark.shuffle.memoryFraction。前者用来指用来存放shuffle操作产生的临时文件的内存大小，如果该参数设置得过小，则系统会将存放不下的数据转存到磁盘中，如果设置得过大，则浪费了系统的内存空间。一般来说，把这个参数设置为比产生的临时文件中占用空间最大的文件要大。spark.shuffle.memoryFraction参数用来指定用于缓存shuffle临时文件的内存空间占整个JVM堆空间的比例。其默认值是0.2，即使用20%的内存空间来存放shuffle临时文件，多出来的数据将会被转存到磁盘中。该参数的设置还关联到spark.shuffle.spill参数，如果后者被设置为true，表示当临时文件无法存放在内存中时，会将它们存放到磁盘中；如果设置为false，系统不会使用磁盘来存储临时文件，而会对内存空间进行回收，这往往会造成内存溢出错误。因此，为了安全起见，会把spark.shuffle.spill设置为true（其默认值是true）。在设置spark.shuffle.memoryFraction参数时，要考虑到spill操作是否频繁，如果spill操作过于频繁，可以适当地将该参数值调高，否则，可以降低该参数值。但是，对JVM堆空间的分配会影响到用于缓存RDD的内存空间，因此在调整该参数的时候，还需要考虑到spark.storage.memoryFraction参数的设置。

其次，说明与数据序列化和压缩有关的参数。主要参数有：

1）spark.io.compression.lz4.blockSize

2）spark.io.compression.snappy.blockSize

3）spark.kryoserializer.buffer.max

4）spark.kryoserializer.buffer

5）spark.serializer.objectStreamReset

1）和2）指定lz4和snappy算法输出数据块的大小，如果设置得过大则会浪费空间；过小，很可能造成原来只需要一个数据块来存放的压缩数据，需要使用跟多的数据快来存放。当shuffle操作产生的临时文件过多时，可以将以上两个参数调小，用来减少临时文件占用的空间。3）和4）分别用来指定存放经过Kryo序列化后的数据所使用缓存空间的最大值和初始值。在Spark中每个CPU会分配一个spark.kryoserializer.buffer。如果运行中发生“buffer limit exceeded”异常时，可以将3）的值调大。5）是和Java序列化相关的参数。采用Java序列化方式时，系统会将序列化后的对象进行缓存，作为灾备恢复数据。这些被缓存的序列化对象对数量，会在达到参数5）设定的数值时进行回收。该参数如果设置得过大，会占用过多内存空间，影响JVM的垃圾回收，有可能造成内存溢出错误；设置得过小，会导致回收对象的操作过于频繁，使得系统性能下降。

最后，来说明与存储相关的参数。主要有三个：

1. spark.storage.memoryFraction
2. spark.storage.memoryMapThreshold
3. spark.storage.unrollFraction

1）在前中文也多次提到，表示用来缓存RDD的内存空间占整个JVM堆空间的比例。任务运行的过程中，如果GC操作过于频繁，可以把这个参数值调小；否则，可以将其调大。2）表示进行内存映射时，读取磁盘数据块的大小。如果这个参数设置得接近或者小于一个物理页的大小，那么会带来很高的系统开销。3）表示用来物化RDD数据的内存空间占用缓存RDD数据的JVM堆空间的比例。在Spark中，为了节省内存空间，当对一个RDD进行缓存的时候，系统只是将这个RDD的相关信息（大小，位置，元数据）等信息写入内存当中。当这个RDD真正被用到的时候，才会从磁盘中将其数据块抓取到内存中，这个过程称为“物化”。参数3）设置了使用多少缓存空间来物化RDD，如果物化所用空间不够，系统会动态回收内存空间。

1. CPU参数

Spark并没有提供类似于Hadoop用来精密控制内核虚拟化的参数，而只是粗略地提供了如表xxx所示的四个参数。因为CPU的调整优化不是本文主要的研究内容，所以在此不对CPU参数进行参数的调整，使用系统提供的默认值。

### 参数优化模块实现

|  |
| --- |
| **Algorithm: Simulated Annealing** |
| 1. *local\_search* = 1 2. *local\_search\_Thres* 3. config[m] = LHS(m) 4. t = 0 5. = best(config[m]) 6. while T > && cost() < e do 7. while *local\_search* < *local\_search\_Thres* do 8. = adjust\_neighbor() 10. if < 0 do 11. = 12. else 13. p = exp() 14. c = random(0,1) 15. if c < p do 16. = 17. endif 18. endif 19. endwhile 21. t ++ 22. *local\_search* = 1 23. endwhile |

参数说明：

local\_search——本地搜索次数

local\_search\_Thres——本地搜索次数阈值

config[m]——包含了m个配置组合的集合

——最优配置方案

——候选配置方案

p——Metropolis值

T——初始温度

函数说明：

LHS——拉丁超立方抽样方法

best——找到最优配置

cost——用来评估一个配置方案的消耗

adjust\_neighbor()——邻域函数，根据现有的配置产生新的配置

random(a,b)——产生a到b之间的随机数，其中a < b

算法xxx是模拟退火算法的伪代码，算法主要有三个优点：1）保证一定概率能够获得全局最优解，2）有一定的抗噪声干扰能力，3）采用了比较高效的LHS抽样算法。下面对算法进行详细说明：

算法分成两层循环，外层循环用来控制降温过程，为了增强系统的时效性，这里使用的是快速降温方式，内存循环用来找最优解。寻找最优解的过程分为两步。第一步采用超立方拉丁抽样方法从参数配置样本集中抽取出m个样本，再从这m个配置方案中选取能让任务运行最快的配置，设为 ，此时得到的这个配置是局部最优解。进入内层循环后，使用邻域函数根据当前的最优配置得到新的候选配置方案，设为配置候选集。计算与 两个配置方案的性能差，若表示是更优的配置方案，将的值赋给。否则，根据Metropolis准则产生参数p（取值范围[0,1]），并利用随机函数生成一个值在[0,1]范围的随机数r，对比r与p的大小。若r < p，则接受作为最优解；否则，放弃配置。当内循环达到限定值的时候（例如，5次），内循环退出。在外循环中，对T进行降温，若T达到了设定的温度下限，外循环退出，整个算法结束。

按照算法的使用时刻的不同，可将算法对参数的优化分为两类：

1. 动态调优：这种调方式是在任务运行的时候使用算法进行调优，首先从候选配置参数空间中选取m个参数配置组合，再将这m个参数配置应用到m个任务中。收集这些任务的运行时间，其中运行速度最快的配置作为当前最优配置。之后，通过邻域函数产生多个基于生成的候选配置集合（集合中的元素个数由迭代的次数决定，即算法中的local\_search\_Thres参数），假设集合中有n种配置方案，那么将这些配置方案应用到接下来的n个任务中。这些任务运行完成之后，比较他们的运行时间，选取消耗时间最少的配置作为候选配置。对比使用和配置的任务执行消耗时间，将消耗时间少的配置作为最优配置。
2. 静态调优：静态调优的过程和动态调优相似，区别是静态调优是在一个Spark应用程序的所有任务执行完成后进行的，而不像动态调优在运行任务的同时收集数据，再将收集到的数据反馈给算法。

动态调优和静态调优除了在运行的时刻不同之外，进行调优的参数也不尽相同，表xxx描述了两者的区别：

|  |  |  |
| --- | --- | --- |
| **调优类型** | **调优参数** | **特点** |
| 动态调优 | spark.shuffle.file.buffer  spark.shuffle.memoryFraction  spark.io.compression.lz4.blockSize  spark.io.compression.snappy.blockSize  spark.kryoserializer.buffer.max  spark.kryoserializer.buffer  spark.storage.memoryMapThreshold  spark.storage.unrollFraction | 在任务运行时调整参数，会对任务的执行速度有一定的影响，优点是能够及时适应系统变化，主要用来调整Spark系统参数 |
| 静态调优 | cpuThr  memRestUpperThr  memRestLowerThr  memCacheUpperThr  memCacheLowerThr  memCodecLowerThr  memCodecMiddleThr  memCodecUpperThr  gcUpperThr  gcLowerThr | 在所有任务运行结束后运行，对系统变化的反应较慢，主要用来调整自适应决策模块中的自定义参数 |

## 本章小结

本章主要介绍SATS系统的实现和原理，首先在4.1节中对SATS系统的整体架构和其在Spark中的作用进行了描述。接下来的三节分别详细描述了运行时数据收集模块，自适应决策模块和参数优化模块。自适应决策模块是系统的核心模块，模块实现了对序列化算法、压缩算法和垃圾回收策略的自适应调整策略。4.4节介绍了参数优化模块，该模块主要针对系统中的配置参数进行调优。该采用了模拟退火算法，分为动态调优和静态调优两种调优类型。

# 系统测试与结果分析

## 实验方法和目的

本实验通过部署安装原生态的Spark和实现了SATS系统的Spark系统，对两者的性能进行对比。通过实验结果来验证基于Spark内存计算特性的自适应调优策略在进行Spark系统优化中的作用。采用控制变量法，进行单元测试、集成测试和整体测试。

基于内存计算特性的优化策略，其重点在于如何高效地利用有限的内存资源。本实验的目在于摒弃了传统的手动调优的方式，通过收集分析任务运行时的特征，让系统自动根据系统资源状态的变化来调整相关参数，保证通过自适应调优后的系统性能要高于简单的手工配置，并能够接近通过具有相当开发经验的开发人员手工调优后的系统性能。

## 实验环境搭建

### 实验环境

|  |
| --- |
| **参数类型 参数值** |
| CPU信息 Intel(R) i7-4770S 3.10GHz  CPU核数 8核  内存 16GB  硬盘 1TB  网络 1000M  操作系统 ubuntu14.10-desktop |

### 集群配置

本实验采用虚拟机群来进行搭建，在主机上使用VMware Workstation 10建立四个虚拟机。虚拟机的配置如图xxx所示。

|  |
| --- |
| **节点名称 参数类型 参数值** |
| driver CPU信息 Intel(R) i7-4770S 3.10GHz  CPU核数 1核  内存 1GB  硬盘 128GB  网络 1000M  操作系统 ubuntu14.10-server-amd64  slave(包含slave1, slave2, slave3三个节点，配置均相同)：  CPU信息 Intel(R) i7-4770S 3.10GHz  CPU核数 2核  内存 2GB  硬盘 128GB  网络 1000M  操作系统 ubuntu14.10-server-amd64 |

实验将原生态的Spark和实现了SATS系统的Spark都安装在driver节点中。之后，通过NFS网络系统将以上两个系统的安装目录挂载到三个slave节点中。这样搭建的好处是，当要对系统进行修改的时候，只需要修改driver上的系统即可，省去了将修改后的系统再次在各个slave中安装、编译的过程，大大节省了开发时间。Driver节点的spark.driver.memory配置为1GB，三个slave节点的spark.executor.memory配置为1GB。

### Hadoop安装

本实验将HDFS作为外部存储系统进行数据的存放。HDFS是Hadoop中的一个子系统，因此，在此需要安装Hadoop。本实验选择的Hadoop版本为Hadoop-2.5.2。在HDFS系统中个，将driver节点作为NameNode，slave1 、slave2 和slave3作为DataNode。

### Spark安装

本实验采用的Spark版本为1.5.1，git地址：<https://github.com/apache/spark>。

## 自适应决策模块测试

本节将对自适应决策欧快进行测试，分为三个单元测试：序列化、压缩和垃圾回收自适应算法的测试与分析；之后，之后会对整个模块进行集成测试。

### 序列化自适应算法测试与分析

|  |
| --- |
| **Algorithm: WordCount** |
| 1. ***Require****: FilePath* 2. ***Ensure****: top 10 most popular IP* 3. val output = sc.textFile(FilePath) 4. .map(line => line.split(“\t”)).map(line => (line(2),1)) 5. .cache() 6. .reduceByKey( + ,2).filter(x => x. 2 > 100) 7. .collect() 8. .sortWith( . 2 > . 2) 9. .take(10) 10. .map(x => (x. 1,x. 2)) |

WordCount用来找出访问最频繁的10个IP地址，数据格式为：服务器IP客户端IP 访问日期。实验中原生态的Spark使用了默认的序列化方法——Java序列化方法。算法如图xxx所示。

实验结果如图xxx所示，从图中可以看出随着实验数据集的增大，采用了序列化自适应算法的系统性能与原生态Spark性能差距越来越明显。当实验数据超过2GB的时候，使用了SATS的系统性能达到了原生态Spark的2倍！

PageRank是常用的推介算法之一，是典型的图计算算法。算法通过图中的点边关系计算出最受欢迎的网站地址。在计算过程中，图中的每一个点根据相邻点的pagerank值来计算自身的pagerank。当所有的点都完成了自身数据的更新后，将整个过程称为一次迭代。再进过多次迭代后，每个点的pagerank值变化会越来越小，当这个变化值达到指定的收敛值时，算法运行结束。从以上分析可以看，PageRank算法是典型的迭代式计算算法，即每一次计算得到的值，将会被下一次计算用到。

图xxx展示了实验结果，同实验一，原生态的Spark采用的序列化方法为默认的Java序列化方法。从结果中可以看出实现SATS的系统性能相比于原生态的Spark并没有多大的提升。这是因为PageRank算法在Spark中会产生大量shuffle操作，这些shuffle操作会产生大量的体积很小临时文件。对这么多的临时文件采用不同序列化方法进行序列化和解序列化，会消耗掉较多的系统资源，导致任务运行速度变慢。同时，JVM堆中用于存放shuffle数据的空间有限（默认仅使用RDD缓存空间的20%），多余的数据要被存放到磁盘中，增加了系统从磁盘读取数据的开销。

### 压缩自适应算法测试与分析

从图xxx中可以看出，在单纯消耗内存使用的情况下，自适应的压缩策略能够取得更高的运行效率。当数据集的大小超过内存大小的时候（数据集为2GB时），采用了压缩自适应算法的Spark的执行速度达到了原生态的Spark的2倍左右。在运行过程中，压缩自适应算法会根据内存的使用情况，自动调整所使用的压缩算法，这相比于默认情况下使用snappy压缩算法的原生态Spark，要灵活的多。

图xxx展示了pagerank的运行结果，从图中可以看出，与之前实现了序列化自适应算法的Spark在pagerank算法上的表现类似。实现了压缩自适应算法的Spark在pagerank上的性能并没有执行的更快，其原因与序列化自适应算法类似，都是因为pagerank在有过多的shuffle操作，产生了大量的体积较小的临时文件。对这些临时文件的压缩处理，在一定程度上是节省了内存空间，但是多次的压缩与解压缩操作也占用了大量的系统时间。因而，压缩自适应算法在pagerank算法上的应用并不如在WordCount上取得的效果好。

从这两个实验中可以看出，实现了压缩自适应算法的Spark比原生态的Spark执行速度要快，在WordCount实验中获得了超过原生态Spark一倍多的运行效率。但是在pagerank实验中，其性能和原生态的Spark相差较小，这主要是shuffle操作的问题。针对shuffle操作的优化已经超出了本文的研究范畴，在此不做详细讨论。

### 垃圾回收自适应算法测试与分析

对于垃圾回收的调优操作，主要是建立在对JVM堆的垃圾回收优化上。图xxx显示了实现垃圾回收优化策略后的系统在WordCount上与原生态系统的对比。从图中可以看出，垃圾回收自适应算法表现得并不是很理想，甚至在内存充足的时候（数据集大小为1GB时，没有超过内存的使用量1GB），其效率是低于原生态Spark的。分析其原因，是在进行JVM堆中各个区域所占空间进行动态调整的时候，会发生数据的迁移，虽然是内存—内存的数据迁移，速度很快，但是当数据量较大的时候也会带来不小的开销。因而，会考虑是否应该把对JVM堆中垃圾回收相关部分的调整，放置到Spark应用程序结束后进行，即取消动态调整，转而进行静态调整。

图xxx是pagerank用例的运行结果，结果与数据序列化和压缩自适应算法的结果相差不大。原因也是由于过得shuffle操作导致产生了过多的临时文件的问题，在此不再赘述。

### 模块集成测试

本实验将SATS系统的核心模块——自适应决策模块进行集成测试，测试结果如图xxx和图xxx所示。由测试结果看出，WordCount测试用例的所获得的性能与单独采用序列化和压缩自适应算法相近，而pagerank测试用例的性能获得了显著的提高。

## 参数优化模块测试

从图xxx和图xxx中可以看出，参数优化模块在初始时，因为没有历史数据的缘故，所获得的系统性能。

## 系统性能整体测试

图xxx和图xxx展示系统整体测试的结果，从结果可以看，当数据集的大小超出系统可用内存时，实现了SATS系统的WordCount的性能是原生态Spark的2倍以上；但在PageRank测试用例上，SATS-Spark的性能在内存不足时稍微比Spark要高一些，根本原因在之前的实验中已经分析过，在此不再赘述。

## 本章小结

本章通过实验验证了实现了SATS的Spark不仅解决了自动配置的问题，而且还在一定程度上提升了系统的性能。但是，对于shuffle过程的调优是基于Hadoop发展而来的分布式平台的一大难题，本文的重点在与如何自适应地根据系统历史信息和运行时系统状态，进行系统的调优工作。使得Spark系统能够获得比在naïve的配置方案下，更加良好的性能。从实验结果可以看出，本文提出的SATS-Spark基本实现了这个目的。

# 总结和展望

## 全文总结

本文通过对Spark采用自适应的优化策略，改变了传统的通过手工进行系统优化的方式，增加了Spark的易用性。同时，通过自适应的优化方案，在一定程度上提升了系统的性能。

Spark自适应优化系统（SATS），着力于如何合理的使用系统资源，如CPU、内存等。现有的针对大数据分布式平台的优化方式主要有三种，第一种通过找到参数空间最优化配置方案的方式，来获取系统的最优配置。这种方法虽然效果明显，但是因为要对参数空间中的所有组合进行一一测试，需要消耗大量的时间。第二种方法通过收集运行完成的任务的历史信息，来对及将运行的任务进行优化。方法二的关键在于要对要运行的任务进行一个分类操作，即找到它的特征，然后在运行过的任务中找到与其相似的任务，然后使用这个与其类似的任务的最优化配置方案作为其配置方案。这种方法的关键点在于如何定义两个任务的相似性。第三种方法是将前两种方法混合起来使用，并加入了一些机器学习算法和规则来加速最优化配置的筛选过程。本文采取混合式的优化方法，通过在运行时收集系统数据来对系统的内存、CPU等资源进行优化调整，并配合使用数据序列化、压缩和针对JVM的垃圾回收优化来进行系统的整体优化。

本文的第三、四章主要介绍了SATS系统的设计与实现。SATS系统主要包括三个模块，运行时数据收集模块（RDDM）、自适应决策略模块（ADM）和参数优化模块（PTM）。其中自适应优化模块是系统的核心模块，主要负责数据序列化、压缩和垃圾回收等三个自适应算法的实现。在这三个算法中，垃圾回收自适应算法主要是针对JVM进行优化。在此需要说明的是，针对JVM的优化随着java语言的广泛流行而被普遍研究。JVM以其良好的跨平台运行、垃圾回收等特性，在各类分布式系统中被广泛使用，例如Spark、Hadoop等。本文借鉴了一些常用的JVM优化方法，并结合Spark自身的特征设计了自适应调优算法。而数据序列化和压缩自适应算法，则是专门针对Spark特有的内存计算特性进行设计的，目的在于如何减少Spark的内存消耗。第五章对文中提出的理论进行了实验验证，针对各个模块分别进行了单元测试和整体系统测试。

## 未来展望

Spark分布式框架发展到现在还是不很完善，还有很多方面有待提高。例如，目前的Spark只整合了两种数据序列化算法：Java和Kryo，其中Java序列化算法各方面的性能在众多的序列化算法中并不是很突出，甚至可以说很差。但因为其是Java语言本身自带的序列化算法，而被沿用到了Spark中。这样造成在序列化自适应算法中能够进行选择的序列化算法很少，没有完全开发出算法的优势。此外Kryo算法使得使用还需要经过注册，使用Kryo对未经注册的类的对象进行序列化，效果并不是太好。现在Spark并不支持Kryo对用户自定义类的注册功能。未来Spark会不断的发展完善，相信到时候系统可以支持多种功能更加丰富的序列化算法，未来的工作将会结合系统集成的这些序列化算法进行进一步的调优工作。

对于基于MapReduce编程模型发展而来的分布式计算框架，普遍存在的问题就是shuffle操作带来的高消耗。shuffle操作不仅消耗空间——会产生很多的临时文件，虽然这些临时文件很小，但是由于数量过多，用于存放文件基本信息的空间也必然增多；而且还很消耗时间——任务运行时，要打开大量的文件，必然会消耗大量的时间。针对shuffle操作进行系统优化，一直是这类系统所面临的一个难题。本文虽然稍微对涉及到shuffle操作的参数进行了调整，并未做深入的研究和分析，在下一步的工作中，将会重点研究shuffle操作的自适应优化问题。此外，还会对系统进行一些细节的改进和完善。