|  |  |
| --- | --- |
| **学校代码** | **10699** |
| **分 类 号** | **TP311** |
| **密 级** | **公开** |
| **学 号** | **2011201749** |

题目 Hadoop环境下负载均衡机制的设计与实现

|  |  |
| --- | --- |
| **作者** | **徐剑** |

|  |  |
| --- | --- |
| **学科、专业** | **计算机软件与理论** |
| **指 导 教 师** | **陈群** |
| **申请学位日期** | **2014年2月** |

# 

西北工业大学

硕士学位论文

(学位研究生)

题目：Hadoop环境下负载均衡机制

的设计与实现

作 者： 徐剑

学科专业：计算机软件与理论

指导教师： 陈群

二零一四年二月

Northwestern Polytechnical University

A Dissertation Submitted for the

Master Degree

Title:A design and implementation of load balance mechanism in Hadoop environment

Author XUJian

Specialty Computer Software and Theory

Supervisor CHENQun

February 2014

# 摘要

随着大数据时代的到来，大量、多样、快速的数据给IT界提出了巨大挑战。 Google提出的MapReduce框架为大规模数据的分析、处理提供了最初的模型。Hadoop是MapReduce的一种开源实现，凭借着高可靠性、高扩展性、高效性和高容错性，它迅速成为分布式计算的热点平台。Hadoop环境下集群的负载均衡也成为分布式计算领域的一个研究热点，本文主要研究的是Hadoop环境下集群的负载均衡机制，及其设计和实现。

Hadoop处理任务时，集群负载，特别是Reduce端负载，常出现不均衡的现象，瓶颈任务大大拖长了任务周期，降低了系统效率。针对这个问题，本文设计了一套基于Hadoop的负载均衡系统。该系统通过数据采样，探知数据在集群中的分布信息；样本数据经过分类，去重，归并等处理之后被传输给系统的决策中心；决策器对其进行分析，判断集群中是否存在数据倾斜以及倾斜的严重程度，并据此对分区进行重组，通过均衡重组后逻辑分区的负载，达到均衡Reduce任务负载的目的。

Hadoop负载均衡系统是在Hadoop的基础上，通过修改Hadoop源码实现，并命名为“Hadoop\_Load\_Balance”。该系统主要包含四个模块，即数据采样模块，信息整合模块，决策模块和消息传递模块。

（一）采样模块，负责对系统的输入数据和Map任务的输出数据进行采样，采集“键-值”对的元组个数及其占用的磁盘空间字节数。采样数据是决策模块的输入。（二）信息整合模块，负责对采样信息进行分类，去重和归并处理，并将结果传递给决策模块。（三）决策模块，负责计算各个分区的负载量，依据数据在集群中的分布情况，数据倾斜的严重程度等信息，运用启发式贪心算法搜索最佳的分区重组方案，并将其提交给系统。（四）消息传递模块，负责模块间的数据传输和消息传递，将各个模块连接成为一个有机的整体。

Hadoop\_Load\_Balance系统在处理多表连接问题时，大规模的中间结果加剧了集群负载不均衡的状况，从而严重降低了系统效率。为解决该问题，笔者提出一种“替换-查询”方法，该方法通过对连接表建立索引，将预输出的元组集替换为索引信息输出到中间结果，以索引的形式参与多表连接，以此减少中间结果规模，；并运用缓冲池、二次排序和多线程技术对索引信息进行优化管理，加快索引的查询速度，提升系统的整体性能。

由于数据不在本地，Reduce任务Shuffle阶段常出现大规模的数据远程拷贝现象，拥塞的网络导致瓶颈（Reduce）任务长时间处于数据拷贝阶段，系统性能下降明显。针对该问题，笔者将本地化思想引入系统，运用贪心算法来控制Reduce的任务推送过程，使得各个节点优先接收、处理本地化程度高的任务，以此降低网络开销，进而完成对Hadoop\_Load\_Balance系统的数据本地化的优化设计。

经验证，Hadoop\_Load\_Balance能够有效解决原Hadoop系统中Reduce端负载不均衡的问题，经过优化之后，系统的时间性能和空间性能较原系统都有大幅度的提升。

**关键词：**负载均衡，Hadoop，数据倾斜，多表连接，本地化

# Abstract

The era of big data has come. A large,diverse data is generated so quickly that it presents a tremendous challenge to IT industry. MapReduce model which is proposed by Google becomes the initial model for the processing of large-scale data. Apache's Hadoop is an open source implementation of MapReduce model.Because of its high reliability, high scalability, high efficiency and high fault tolerance ,Hadoop becomes a hot platform rapidly. Hadoop cluster load balance has also become a hot topic in the field of distributed computing. This paper studies methods of Hadoop cluster load balance management in data skew environment.

In a data skew environment, when handling a user’s job,the problem of load imbalance often occurs in Hadoop system cluster. Bottleneck tasks prolong the cycle of the task greatly. To address this problem, this paper proposes a load balancing intelligent decision-making system.In order to get distribution information of input data,it provides a sampler. Sample data is classified. Duplicate information of sample data is removed. At last, sample data is merged together and is transmitted to the Decision block of the system. In order to get the information of whether there is data skew in input data and the severity of data skew, The decision-making unit analyses sample data. The partitions are regrouped to balance logical partition load. In the end,the load of every reduce task has a similar size.

On the basis of Hadoop, it achieves load balancing intelligent decision-making system by modifying the source code of Hadoop. The new system is named “Hadoop\_Load\_Balance”. It consists of four modules,data sampling module,information integration Module, decision-making module and messaging module.

(I): Sampling module of the system samples input data and output data of Map tasks. It gets a collection which contains the number of tuples and the bytes of disk space these record tuples occupied.These information is used as the input data of the Decision-Making module. (II): Information Integration module is mainly responsible for information classification, merging.The processed information is transmitted to Decision-Making module. (III): Decision-making module is responsible for calculating the load capacity of each partition .Combined with the data distribution information in cluster as well as the severity of the data skew , heuristic greedy algorithm provides us a better partition restructuring scheme. The scheme is submitted to Hadoop system to execute this program. (IV): Messaging module is responsible for the data transfer between the modules. It keeps several modules connected.

When dealing with the issue of multi-table join on Hadoop\_Load\_Balance, a large number of intermediate results are written to local disks. As a result, efficiency of the system becomes very low. In order to solve this problem, it proposes a "Replace-Query" method. This method builds indexes on tables which are involved in the problem. Intermediate results are replaced by indexes whose size is much smaller. The I/O cost of the intermediate results becomes quite low. In order to improve system performance,it makes fully use of buffer pool,secondary sort, and multi-thread technique to optimize the management of indexes. These indexes participate in the whole multi-table join process and records can be fully and rapidly recovered by querying.

Since the data is not local, remote copy of large-scale data often appears durning Shuffle stage of Reduce task. Network bandwidth is very congested. It takes the bottleneck (Reduce) task a long time to copy data from remote machines. The load of system will be more uneven. To solve this problem, it introduces the idea of localization to the system. Greedy algorithm is used to control the push of Reduce tasks to different nodes. Each node receives and processes a higher degree of localization tasks in order to reduce network overhead and complete the localized optimization of the system.

The experiments prove that compared with the original Hadoop system, the optimized load balancing decision-making system can effectively balance the load of the system, and the performance of both time and space has been greatly improved.

**Keywords:**data skew,load balance,Hadoop,multi-table join,data localization.

# 目录

[摘要 I](#_Toc13735)

[Abstract III](#_Toc11250)

[目录 1](#_Toc6823)

[第一章 绪论 1](#_Toc27447)

[1.1 研究背景与意义 2](#_Toc12075)

[1.2 国内外研究现状 3](#_Toc24409)

[1.3 论文的内容及组织 4](#_Toc17380)

[1.3.1 研究内容 4](#_Toc15214)

[1.3.2 组织结构 4](#_Toc6138)

[第二章 Hadoop及负载均衡相关技术综述 7](#_Toc24779)

[2.1 Hadoop技术综述 7](#_Toc13098)

[2.1.1 分布式文件系统（HDFS） 7](#_Toc7429)

[2.1.2 MapReduce模型 9](#_Toc17984)

[2.2 负载均衡理论概述 12](#_Toc14618)

[2.2.1 负载均衡的基本概念 12](#_Toc27671)

[2.2.2负载均衡算法 13](#_Toc27373)

[2.3 本章总结 14](#_Toc14262)

[第三章 Hadoop负载均衡系统的设计和实现 15](#_Toc875)

[3.1 系统总体设计 15](#_Toc8268)

[3.2 系统详细设计之数据采样模块 17](#_Toc18201)

[3.2.1 采样的必要性 17](#_Toc8727)

[3.2.2 采样器的设计 17](#_Toc3037)

[3.3 系统详细设计之信息整合模块 22](#_Toc25467)

[3.3.1 信息分类 22](#_Toc20029)

[3.3.2 信息去重 22](#_Toc26580)

[3.3.3 信息归并 23](#_Toc16748)

[3.3.4 整合模块的设计 23](#_Toc26049)

[3.4 系统详细设计之决策模块 24](#_Toc14550)

[3.4.1 数据倾斜 25](#_Toc17672)

[3.4.2 代价模型 28](#_Toc32532)

[3.5 系统详细设计之消息传输模块 37](#_Toc10295)

[3.5.1 心跳信息的调制 37](#_Toc3434)

[3.5.2 加载配置文件 39](#_Toc18606)

[3.6 本章总结 40](#_Toc28729)

[第四章 Hadoop负载均衡系统优化 41](#_Toc24516)

[4.1 一种基于Hadoop的多表连接优化策略 41](#_Toc18449)

[4.1.1中间结果集的膨胀问题 41](#_Toc28178)

[4.1.2 优化方案的整体设计 42](#_Toc5190)

[4.1.3 NIHDFS索引设计 42](#_Toc31285)

[4.1.4 基于NIHDFS的“替换-查询”方法 44](#_Toc6964)

[4.1.5 “替换-查询”方法的系统实现 44](#_Toc27269)

[4.1.6 若干细节优化 45](#_Toc9621)

[4.2 一种基于Hadoop的数据本地优化策略 47](#_Toc31119)

[4.2.1 数据本地化的必要性 47](#_Toc11669)

[4.2.2 数据本地化的设计和实现 47](#_Toc853)

[4.3 本章总结 50](#_Toc26511)

[第五章 实验部分 51](#_Toc6563)

[5.1 实验条件及环境搭建 51](#_Toc32764)

[5.1.1 实验条件 51](#_Toc8212)

[5.1.2 搭建Hadoop集群 51](#_Toc29668)

[5.2实验内容以及结果分析 54](#_Toc12704)

[5.2.1 多表连接策略优化前后系统的性能对比................................................. 54](#_Toc3224)

[5.2.2 数据本地优化前后系统的性能对比........................................................ 56](#_Toc16413)

[5.2.3 Hadoop负载均衡系统的参数调优及与原Hadoop的性能对比 56](#_Toc5494)

[5.3实验结论 62](#_Toc20004)

[5.4 本章总结 62](#_Toc8347)

[第六章 总结与展望 63](#_Toc32671)

[6.1 本文研究总结 63](#_Toc15239)

[6.2 课题研究展望 63](#_Toc10339)

[参考文献 65](#_Toc15812)

[致谢 69](#_Toc28851)

[知识产权说明书 7](#_Toc28851)1

第一章 绪论

随着移动无线带宽的快速提升，社会交际网络的快速普及，物联网相关应用的更加丰富，云计算相关技术的快速发展，更多的设备被添加到网络中，数据的产生速度飞速增长。快速化、大量化和多样化是大数据的显著特征，快速、大量、多样的数据给IT业提出了巨大挑战，对大规模数据的处理和分析也成为这个领域的研究热点。

这是一个数据爆炸式增长的时代，天文学，医疗记录，RFID（射频识别），社会网络，摄影视频，基因组学，社会数据（由于数据革命的社会），复杂科研，大气科学，传感器集合，搜索索引，网络日志，电话呼叫记录，生物化学，以及军事侦察，电子商务等，所有的这些都在时刻不停的产生着新数据。美国互联网中心发布的信息，全球（互联网）数据的年增长率大约为50%，每两年数据将增长一倍。这些数据不是单纯指的是互联网数据信息，全球的汽车、电表等工业设备上有着数不清的传感器设备，他们时时刻刻都在产生新数据，大数据时代已经来临。

大数据有四层面的特点：第一，数据类型多种多样。如语音，视频，文本等，数据种类非常多样。第二，数据量巨大。从TB级升到了PB级别。第三，处理速度快。业界将其归纳为四个“V”，即Volume（大量）、Velocity（高速）、Variety（多样）、Veracity（精确）。

为了能够快速有效的处理大数据，各大互联网公司，如谷歌（Google），（国际商业机器）IBM，微软和亚马逊，阿里巴巴，百度等，竞相发展并逐步形成了各具特色的云计算平台。

谷歌的硬件条件优势，大型及超大型的数据中心，搜索相关技术的支撑，促进谷歌云计算迅速发展，使其成为云计算领域的佼佼者，谷歌的云计算平台主要有三部分组成，它们分别是Map-Reduce模型框架，GFS，分布式数据存储系统（BigTable）。谷歌开发了一系列的相关组件，其中包含一个描述语言组建和一个分布式（Distributed）锁机制。谷歌还开发了一种建立在Map-Reduce模型基础上的Sawzall语言，主要是用在大规模及超大规模的的数据信息处理上。当集群中有机器宕机时，使用Chubby服务来进行处理，而这个服务运用的是Paxos机制。

IBM蓝云是一个结合了软，硬件系统，并且使用了许多领先的数据分析技术和服务应用技术，对开放标准提供支持技术的开源的软件平台。蓝云平台包含了监控软件、数据中心、以及Tivoli相关的系统管理部署软件、DB2技术数据库、安装有WebSphere软件平台的IBM服务器以及开源的信息处理和虚拟化软件。智慧云计算是IBM智慧战略中的非常重要组成部分，其云智慧正在不断的向云计算领域延伸。

亚马逊设计了自己的云系统，该系统最大的特点是支持弹性计算，用户管理自己任务的方法是通过网络UI进行远程控制，用户使用弹性资源的状况决定了费用，当用户提交自己的任务并运用系统资源进行计算时，系统就开始计费，当任务执行完毕之后停止计费，该平台建立在亚马逊公司内部的大量机器上。该平台中包含了若干个正在虚拟机中运行的实例。用户可以非常方便的访问自己租用的机器（虚拟机），并且对该机器拥有所有的控制权限。计算被当做服务出租给用户，并依据用户的实际使用状况收费。

Abiquo公司有三种能够架构云平台的产品，即AbiCloud、 AbiNtense和AbiData，他们各有自己的特点，AbiCloud 属于开源云软件，它具有强大的可扩展性，令它能够非常容易的，扩展自己的集群规模。它提供了一个比较完备的Web管理平台，通过该平台可以方便的对集群进行管理，操作少，门槛低。

Eucalyptus(Elastic Utility Computing Architecture for LinkingYour Programs To Useful Systems)是加利福尼亚大学(SantaBarbara)为进行云计算研究而开发的一个云计算平台，它能兼容商业服务接口。

10gen 是一个开源的云平台，它不仅更新快，而且通过它用户可以方便快捷的创建和管理云平台。10gen 平台上开发者可以灵活的选择自己比较擅长的语言，因为该平台兼容Python、JavaScript以及Ruby语言。

Enomalism是开源的计算平台，该平台提供了对集群弹性的计算能力的管理能力，该系统基于Linux，提供类似于亚马逊的弹性计算能力。

1.1 研究背景与意义

大数据催生了众多的计算机技术，其中最有影响力是Google提出的MapReduce框架模型，该模型为大数据的处理提供了最初的模型。它由两部分构成：编程模型和运行时环境。

MapReduce框架模型的优势十分显著。主节点通过周期性的接收子节点发来的工作和状态报告获取信息，迅速高效的完成对任务和集群的管理，管理过程中操作的原子性确保了系统的稳定性。因此MapReduce编程框架被众多的公司开发利用，其中比较有影响力的是Apache Software Foundation的Hadoop。

Hadoop由 Apache Software Foundation 公司于 2005 年秋天作为[Lucene](http://baike.baidu.com/view/371811.htm" \t "_blank)的子hadoop logo项目[Nutch](http://baike.baidu.com/view/46642.htm" \t "_blank)的一部分正式引入。它受到最先由GoogleLab开发的 MapReduce和Google File System([GFS](http://baike.baidu.com/view/805525.htm" \t "_blank)) 的启发。2006 年 3 月份，MapReduce和Nutch Distributed File System (NDFS) 分别被纳入称为Hadoop的项目中。它的高可靠性，高扩展性，高效性，高容错性让Hadoop迅速的成为分布式计算的热点平台。 Hadoop采用[并行执行](http://baike.baidu.com/view/160033.htm" \t "_blank)机制，将该任务切分成众多的小任务并且在不同的机器上并行执行，因此能大大提高效率。对于可并行化的计算密集型任务或者数据密集型任务，Hadoop都能够快速有效的进行解决。

在实际的应用中，利用Hadoop处理问题的时候或多或少的存在集群负载失衡的问题，尤其在数据倾斜环境下，集群的负载失衡常常会大大的降低系统的处理效率。随着数据量的增加和数据应用范围的扩大，数据采集、存储、加工、使用等多方面原因导致的数据倾斜问题日益严重。由于初始数据存在倾斜，在没有考虑数据的倾斜情况下进行的随机的任务分配，很大程度上造成个别子节点处理的数据量远远超过其他子节点，系统长时间的处于等待该节点执行任务的状态中，从而降低了系统的处理效率。

任务分配策略的不合理也是造成集群负载失衡的重要原因。Hadoop中Reduce任务分配采用的是随机策略，即某一时刻，哪个任务被分配是随机的，该任务被分配到哪个节点上也是随机的。如果某一个时刻，某个重负载任务被分配到现有的重负载节点上的时候，那么该节点就成为了集群的瓶颈节点，随机策略对该节点分配的过重的负载导致这个节点要花费相当长的时间来完成任务，因此任务的分配策略也是集群负载的重要影响因素之一。

1.2 国内外研究现状

对于Hadoop的负载均衡问题，国内的众多学者对其进行了研究。集群的负载均衡对HDFS同样有着非常显著的影响，HDFS中本身自带的负载均衡策略不够灵活，常常导致HDFS的负载均衡效率非常低下，针对这个问题，文献[1]描述了一种动态的负载均衡策略,该策略通过对网络带宽进行动态的分配以达到均衡各个节点数据负载的目的。这个策略的主要贡献在于给集群添加一个新机器时，在不影响HDFS访存性能的前提之下,有效的提高了均衡效率。

文献[2] 通过负载超重机架被优先处理，来达到均衡集群中各个节点的数据负载的目的，方法思路如下，首先通过对机架引用进行排序，找到集群中各个节点的负载量的大小，以机架作为基本单位进行处理，令负载比较大的机架能够被优先处理。Hadoop本身带有均衡器，该均衡器中存在着运行所需的多个阈值，对这些阈值赋值完成之后，均衡器才能正常运行，针对于这个缺点文献[3]提出了一种动态的负载均衡策略，该策略通过动态的对均衡器输入的各个阈值的估计，来均衡系统负载的目的。

国际上对于集群的负载均衡也有许多的研究，文献[4]首先研究了任务执行过程中集群中各个节点负载的变化情况，从中找出相关规律的集合；其次，针对这些规律，提出了一套集群负载的动态分配算法，通过运用变换之后的负载权重值来衡量各个节点目前的负载情况，然后采用随机策略对任务进行动态分配。

文献[5] 提出由于集群中各机器节点的异构性，机器的实际处理能力常常是不相同的，另外数据在网络中传输的过程中，不同节点也将导致不同的数据传输延迟，据此集群的最小代价节点被挑选出来，当分配一个新任务时，系统优先将该任务分配给这个节点，达到提升系统性能的目的。

1.3 论文的内容及组织

1.3.1 研究内容

(1)对MapReduce模型进行进一步研究，将理论和实验相结合，确定Hadoop存在的各种类型的负载不均衡以及产生的机理。

(2)对Hadoop环境下集群的负载均衡问题进行具体研究，分析影响负载均衡的各个因素，并建立负载均衡的代价模型。

(3)在该模型基础上，针对模型中的各种问题，设计解决方法并实现相应的算法。如负载均衡的衡量算法，任务调度算法，采样器的采样算法，消息传递算法等，并在此基础上搭建Hadoop负载均衡系统。

(4)引入数据本地化的思想，对系统进行优化，从而减少机器节点之间过多数据传输造成的网络代价。在系统处理多表连接问题时，中间结果的快速增长导致系统性能下降明显，针对这个问题，笔者设计了“替换-查询”的Hadoop多表连接策略，通过对连接表建立索引对系统进行优化，以减少中间结果爆炸对系统性能的影响。

(5)设计实验，与一些相关工作进行对比，调整系统的参数，说明各个参数的设定对算法的影响，总结了本文所设计的系统的适用场合、优势与不足及后续需要完善补充的地方。

1.3.2 组织结构

第二章，主要介绍了Hadoop系统以及负载均衡的相关技术理论。

由于本文选用的系统平台是开源的Hadoop系统，本章节分别对Hadoop的分布式文件系统（HDFS）和MapReduce编程模型进行了介绍。本章节还对集群负载均衡的基础概念，基本原理和相关算法进行了介绍。

第三章对Hadoop负载均衡系统中的数据采样，决策，消息传递以及信息整合四部分的关键技术，设计细节进行详细的描述，主要介绍设计方法，功能，以及相关的算法等。

第四章对Hadoop负载均衡系统进行优化。优化主要解决两个问题，(1)系统在处理多表连接问题时，由于中间结果急剧膨胀，导致系统I/O代价过高的问题。(2)由于数据迁移而导致的节点间大量的数据传输，致使网络代价过高，降低系统性能的问题。

第五章主要介绍了实验条件，Hadoop系统搭建，并且设计实验，验证了系Hadoop负载均衡系统的有效性。

第六章是对本文工作的总结以及未来工作的展望，说明本文方法的局限性与不足之处，今后可以改进的地方及提升的空间。

第二章 Hadoop及负载均衡相关技术综述

2.1 Hadoop技术综述

Hadoop是一个开源项目，该项目由Apache倡导并开发的。HDFS是Hadoop平台的文件系统，MapReduce模型是Hadoop的计算模型。为了提高文件的容错性，系统将数据的多个备份存放在不同的机器节点上，一个节点上存储的数据出现错误是，系统能够自动将恢复数据并补足数据备份的数目。由于Hadoop是开源平台，开发者或者用户能够方便的下载其源码。

Hadoop在数据密集和计算密集环境下展现出良好的性能优势，再加上它本身的开源性，开发者可以依据需要来对其进行个性化的定制，从而深受广大用户的喜爱。

Hadoop平台是一个强大工具，利用它可以方便有效的对大型数据集进行操作。Hadoop平台不仅架构起来非常轻松，而且使用起来也十分方便。Hadoop应用实例的开发起来相当简单，开发周期也相对较短，利用Hadoop用户可以方便快捷的处理海量数据。Hadoop的优点主要有以下几个：

(1)高可靠性；一方面，Hadoop是以位作为基本单位进行存储的；另一方面，Hadoop集群整合了集群的处理和运算能力，性能十分强劲。

(2)高扩展性；Hadoop对集群进行管理的时候，忽略了机器的异构性，使得系统可以非常方便的进行扩展。

(3)高效性。Hadoop利用MapReduce模型，将大任务切分成一系列的子任务，各个节点将子任务的运算结果汇总起来得到最终的结果，任务执行的高并行度，保证了Hadoop的高效性。

(4)高容错性。Hadoop分布式文件系统采用的是多副本机制，当某些副本出错时，系统能够自动探知并重新拷贝一个，来保证正确副本的数量。

(5)低成本。

2.1.1 分布式文件系统（HDFS）

HDFS，全称Hadoop Distributed File system，是Hadoop生态圈的分布式文件系统。

**(1)HDFS的架构**

如图2-1所示，HDFS本身采用的Master/Slave架构，Master是集群的主控节点，一个集群中存在一个主控节点，主控节点又叫NameNode节点。一个HDFS集群中可以存在多个子节点，子节点又叫做DataNode。整个系统中Master节点是服务器，它的主要功能是对文件进行命名，以及文件的维护。文件的访问过程控制也有NameNode来控制，如文件的读出、写入等。当用户提交一个工作给Hadoop集群时，这个任务将会被切割，切割好的子任务将被分配到集群的各个节点上运行，每个任务执行节点对Hadoop系统的数据（本地的和非本地的），都可以进行访问。

此外，HDFS本身具有高容错性，当文件被切分成数据块并被存储到HDFS之后，对于某一个数据块，HDFS存储着多个该数据块的副本。HDFS的主控节点可以控制用户程序并行的对这些数据进行访问控制，因而Hadoop具备很高的数据吞吐率。

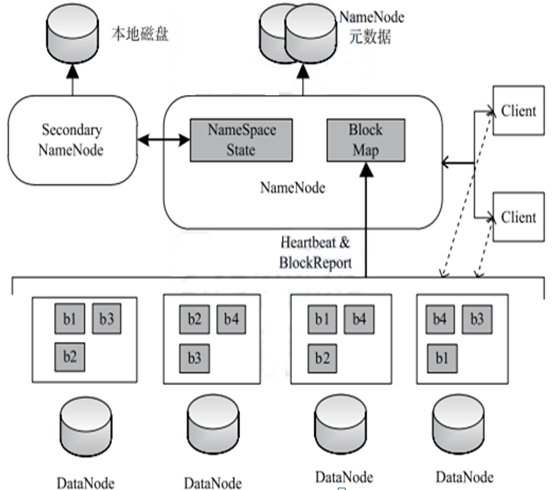


图2-1 HDFS架构图

**(2)HDFS相关概念**

1)数据块

数据块是HDFS最基本的单元，默认情况下，每一个数据块的大小是64M。数据块比磁盘块大得多，降低了寻址开销。另外，HDFS上还存放着一些个独立的分块，即（chunk）。HDFS的一个特点是数据块不会占用整个的数据块空间，节省了大量的磁盘资源。

2)Client

Client（用户）通过与NameNode 和DataNode交互访问HDFS 中的文件。 3)NameNode

NameNode，即名字节点，该节点时HDFS的主控节点，负责对文件和数据节点的管理。

2.1.2 MapReduce模型

**(1)MapReduce架构**

同HDFS一样，Hadoop MapReduce也采用了Master/Slave（M/S）架构，具体如图2-2所示。它主要由以下几个组件组成：Client（User） 、JobTracker（Master） 、 TaskTracker（worker）。

Hadoop系统会将HDFS中的数据块切分成若干大小相同的Split块，Split是逻辑块，一个Split可能对应着多个Block块，这些Split是Map任务的输入，默认情况下，一个Map Task处理且仅处理一个Split。当某一个Map任务完成之后，Reduce函数的执行节点，会从Map任务的执行节点读取Map的运行结果，在Reduce函数运行之前，需要对来自不同节点的数据进行排序，归并。Reduce函数运行完毕之后，所有的结果会被存储在系统的HDFS上，具体流程如图2-4所示。

|  |
| --- |
| 图2-2 MapReduce流程示意图 |

**(2)MapReduce相关概念**

1)Client

Hadoop中所有的MapReduce程序是通过Client提交到JobTracker的；Client提供丰富的计数器，用户可以方便获取系统的状态和任务的执行信息

2）JobTracker

JobTracker运行在集群的Master节点上，是Hadoop系统运维和任务的调度分配的控制中心，主要负责监控和管理Hadoop集群和用户任务。

3）TaskTracker

TaskTracker是任务执行节点，主要负责map任务和reduce任务的执行。TaskTracker通过心跳信息向主节点（JobTracker）发送任务的申请信息，当TaskTracker满足任务分配的条件时，主节点向TaskTracker会给该节点分配一个任务（Map任务或者Reduce任务），TaskTracker接受到回复信息之后，从其中解析中任务执行的信息，拉取数据，处理数据，并定时将任务的状态信息发送给主节点，直到该任务执行完毕。

4）Task

Hadoop中Task 分为Map Task和Reduce Task两种，均由TaskTracker 启动并完成。

**(3)MapReduce执行机制**

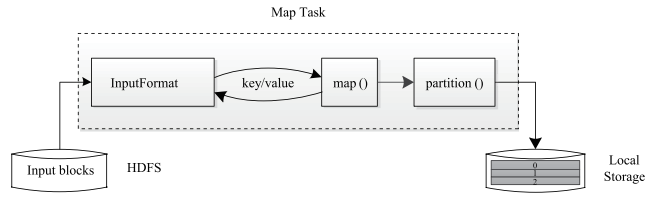
MapReduce由两个阶段组成：Map和Reduce。Hadoop将HDFS中数据块切分成Split之后，被解析成一系列的键值对，每一个键值对作为一条记录，被map函数处理。Map函数将结果输出到磁盘上。Hadoop对map函数的输出结果进行分区操作，将其分成一系列的分区，这些分区被传输到Reduce函数的执行节点，经过排序，归并之后，最终结果作为Reduce函数的输入，Reduce函数执行完毕之后，系统会将结果存储在HDFS上。

2）Map和Reduce机制

映射函数，即Map函数，对数据集中的所有元素都必须执行的一系列的操作，对各个元素处理的结果互不影响，因此在不影响结果的前提下，众多的Map任务可以同时执行。

Map Task执行过程如图2-3所示。Hadoop先将输入的Block块解析成一系列的键值对，并作为Map函数的输入进行处理，Map函数的执行结果经过分区处理之后，也以键值对的形式存放在本地磁盘上。

图2-3 Map Task流程示意图



化简操作（Reduce）主要负责将隶属于一个列表的元素进行合并，它的并行度没有map那么高，但是化简的最终结果常常非常简单，所以Reduce能够适应高度并行的环境。Reduce Task执行过程如图2-4所示。该过程分为三个阶段：

a.将Map任务的处理结果从Map任务的执行节点拷贝到Reduce任务的执行节点，这一阶段也称为copy阶段；

b.对拷贝过来的数据进行归并和排序；

c.经过归并和排序的数据被Reduce函数处理，系统将最终的处理结果存储在HDFS上。

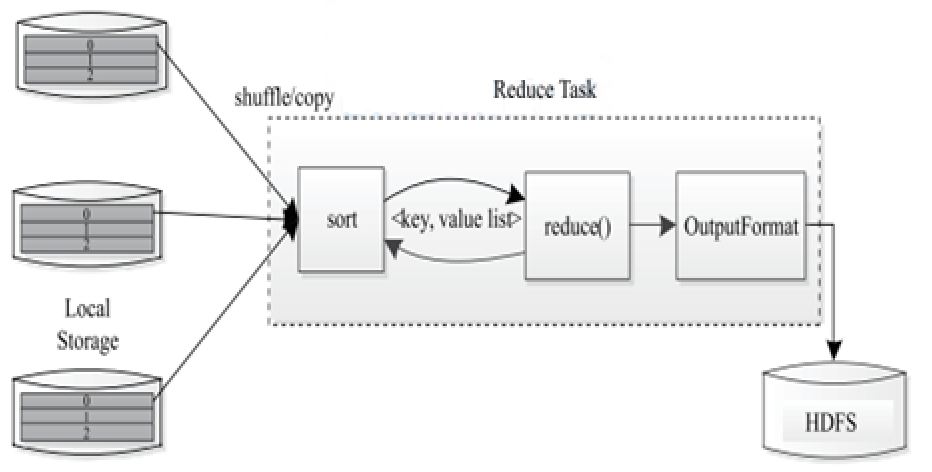


图2-4 Reduce Task执行流程示意图

2.2 负载均衡理论概述

2.2.1 负载均衡的基本概念

**（1）负载均衡的含义**

负载均衡（Load Balance），其意思是将负载进行平衡、分配给多个（硬件）操作单元上进行执行。

负载均衡的含义主要有两个方面。

一方面，单个大负载的任务被分配到多个节点上并行运行，各个[节点](http://baike.baidu.com/view/47398.htm" \t "_blank)将处理结束之后，对各个节点的结果进行汇总，并将最终结果返回给用户，这就是集群（clustering）技术。

其次，在大并发控制的环境中，受机器性能的限制，单台机器的处理时间太长，将任务切分并派发到多台机器上执行，通过任务的并行执行来降低相应时间，提升用户体验。

**（2）负载均衡分类**

1）静态/动态负载均衡

从负载均衡策略上可以分为两种，即静态负载均衡和动态负载均衡。集群负载存在着所要执行的任务集，在不考虑当前负载的情况下，将任务集中的任务分配到集群的各个机器上执行，使得任务的执行时间最短，这就是静态负载均衡。

对任务集中任务进行分配的时候，将集群的负载情况考虑在内，依据当前的集群负载状况，决策出不同的任务分配方案，使得系统的整体执行时间时间最短，这就是动态负载均衡。

2）集中/分布式负载均衡

存在中心控制器的负载均衡就是集中式的负载均衡，这种负载均衡是Master/Slave结构，系统中有且仅有一个中心控制器，即Master，负责统筹集群负载信息，管理任务分配，集群中其他节点是Slave，Salve要向Master定期的汇报负载情况，并且接受Master的调度，该方法容易实现，具有较好的稳定性。

分布式均衡策略，即各个节点上都能对集群的负载进行控制，系统中不存在中心控制器，各个节点通过向别的节点发送信息来控制负载的控制，均衡集群中各个节点的负载。该方法算法复杂，会消耗客观的网络带宽资源。

3）软/硬件负载均衡

软件负载均衡是通过软件的方式来控制集群负载均衡的方式，这种方式成本低廉，使用简单而且具有很强的灵活性。

硬件负载均衡主要是通过负载均衡设备来均衡各个节点的负载，负载均衡设备独立于集群的其他设备之外，而且专门用来均衡集群负载，因此这种方式非常高效，但是成本也高。

2.2.2负载均衡算法

对于负载均衡系统来说，平衡算法好坏与否非常关键。一个好的负载均衡算法能够大大缩短任务的执行时间，提升系统效率，而一个不好的平衡算法，均衡效果不明显，甚至有可能加剧集群的负载失衡状况。

一个算法的好坏，跟它使用的场合有很大的关系，同样的一个算法，在不同场景中，平衡效果有可能相差巨大，因此，选用平衡算法时要考虑算法是否符合应用场景。

平衡算法主要分为两类，即持续性算法（Non-Persistent）和非持续性算法（Persistent）。处理用户请求时，一个用户的不同请求有可能被分配到不同的机器上执行，这一类算法是非持续性算法。该类算法主要有最少连接、响应速度、加权轮叫、轮询、加权最少链接等。

一个特定的用户所发出的请求都被分配到同一台机器上处理，这类算法叫做持续性负载均衡算法。经典算法主要有三个，即基于“报头/请求”算法，基于IP算法，以及基于Cookie的算法。

**（1）轮转法**

轮转法是一种随机选择算法，选项集合中各选项被选中的概率都相同。当选项集中有M个选项的时候，每一个选项被选中的概率都是1/M。该方法实现起来比较简单，适用于集群各节点处理性能相同的场景中，它经常跟其他方法结合使用。

**（2）散列法**

散列法又叫哈希法，利用哈希函数的映射关联规则，将负载发往集群中的各个节点。该方法实现简单，效果的好坏取决于集群当前的负载情况和哈希函数。

**（3）最少连接法**

平衡器统计集群中各节点的活跃连接数量，当一个新的请求到来时，系统将系该请求发送给当前连接数最少的那个节点。

**（4）最低缺失法**

平衡器（长期）记录各节点的请求处理数量并对其进行更新，当一个新的请求到来是，将新请求分配给历史上处理请求次数最少的节点。

2.3 Hadoop技术与负载均衡的关系

由于Hadoop的易扩展性，一个Hadoop集群包含成百上千台机器。当该集群处理任务时，数据倾斜、中间结果集膨胀、数据不在本地等问题导致瓶颈任务的负载过大，大大拖长了任务的执行周期，降低了系统的处理效率。在Hadoop集群中，这种负载不均衡的现象非常普遍，已经严重影响了系统性能。

经典的负载均衡问题是通过子任务分配达到均衡各个任务任务量的目的，从而缩小其他任务与瓶颈任务的执行时间差，以提高系统性能。

本文是将经典的负载均衡思想运用到Hadoop集群环境中，通过对Hadoop集群中负载均衡机制的研究、设计和实现，达到平衡集群负载的目的，从而提升系统效率。

2.4 本章总结

本章主要介绍了相关的背景技术，主要有两方面的内容，一方面讲述了Hadoop的总体结构，对Hadoop中的分布式文件系统，即HDFS，以及MapReduce相关理论、架构进行了描述。另一方面介绍了负载均衡的基本概念，理论以及相关的算法。

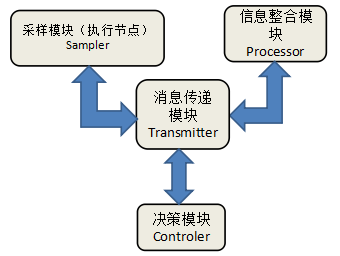
第三章 Hadoop负载均衡系统的设计和实现

本章重点介绍Hadoop负载均衡系统的设计和实现，该系统包含了采样，信息整合，决策和消息传递描四个模块。本章从系统的总体设计和详细设计两方面入手，对系统的设计思路、实现方法以及相关算法做了详细的描述。

3.1 系统总体设计

Hadoop负载均衡系统，即Hadoop\_Load\_Balance，主要包含采样，信息整合，决策控制和消息传递四个模块，他们之间的结构关系如图3-1所示。

图3-1 负载均衡决策框架图



当某个Hadoop任务提交给系统之后，在任务的Map阶段，机器节点TT1（TaskTracker1）和机器节点TT2（TaskTracker2）上同时运行着三个Map Task，每一个Map Task的采样器会对其所处理的Split数据和Map输出的结果进行采样。 机器节点TT1和TT2的信息整合模块接收样本数据，对样本数据进行消息分类，去重之后，将结果输出给消息传递模块。

消息传递模块将各个TaskTracker经过整合的采样信息经网络传递给Hadoop的主控节点（Master/JobTracker）。主控节点的信息整合模块对收到的信息进行进一步的过滤、分类、去重以及归并处理，对样本信息进行更新并作为主控节点决策模块的输入。决策模块依照代价模型对样本数据进行分析，依据集群代价模型，

|  |
| --- |
|  |

图3-2 Hadoop\_Load\_Balance设计框架示意图

计算集群中各节点和各个待分配任务的负载，将集群负载的方差最小作为目标，对解空间进行搜索，得出最佳的任务分配方案。主控节点将最优的任务分配方案通过消息传递模块发送到各个TaskTracker节点，接收到任务分配方案的子节点，对分配方案进行解析，找到自己所要完成的任务，并即刻调集数据开始执行。整个过程的详细流程图如图3-2所示。

3.2 系统详细设计之数据采样模块

采样模块主要是对系统的输入数据以及Map任务输出的数据进行采样。该部分在Map阶段进行。采样结果作为决策模块的输入。

具体要对那些数据进行采样，用户可以自定义，本文设计的Hadoop负载均衡系统主要是采集元组记录个数和所占用的字节数。

3.2.1 采样的必要性

造成Hadoop负载不均衡的主要原因在于数据倾斜，在用户提交任务之后，系统对任务的输入数据的具体组成并不知情，因此当输入的初始数据存在数据倾斜时，系统依旧按照正常的情况进行任务分配，很有可能给某个别节点的任务量太大，从而造成Hadoop集群负载的不均衡情况。

采样模块主要是在系统决策之前，对输入数据和Map的输出数据进行采样，取得输入数据和Map Task的输出数据信息。比如知道key1，key2，key3，key4……，对应的记录的个数分别是n1，n2，n3，n4……,对应的记录元组的大小分别是s1，s2，s3，s4……。根据各个key所对应的记录个数和占用磁盘字节数的大小就可以得知该任务数据是否存在倾斜以及倾斜的严重程度。

综上所述，采样模块的设计能够使的系统在分配Reduce任务之前，对于输入数据和Map Task的输出结果有个比较清晰的了解，更好的掌握数据的倾斜状况。

3.2.2 采样器的设计

**(1)采样器**

设计采样器的目标是采集数据，并利用采集的数据准确的估算出系统所处理的数据的整体情况。采样器的设计有两种，一种是利用数据预处理对系统所处理的数据进行采样，该方法需要写一个Hadoop的采样实例，在运行用户程序之前，先执行一遍该预处理程序。另一种是侵入式的采样器，在用户任务执行的同时，并行地进行数据采样，该方式需要通过修改Hadoop系统源码来实现。

1)数据预处理采样器

运用数据预处理的方式对系统所处理的数据提前进行处理，通过统计或者采样的手段得出数据的特征，从而判断该数据中是否存在倾斜。

2)侵入式采样器

侵入式采样器是通过修改Hadoop源码完成的，对于用户来说是透明的，用户在运行某个任务的时候无须关心采样器的执行时机，采样算法等细节，系统内置的采样器可以完成对数据的采样工作。

由于该方式需要修改Hadoop源码，开发者需要有一定的Hadoop源码基础，门槛相对较高。

3)两种采样器的对比

a.开发的难易程度

数据预处理采样器，它本质上就是一个Hadoop的MapReduce应用实例程序，需要在运行用户程序之前运行该实例程序。它属于Hadoop应用实例级的开发。侵入式采样器是通过将采样器内置到Hadoop中，需要对Hadoop源码进行深层次的修改，属于Hadoop开发级别的开发。就开发的难易程度上来说，侵入式采样器相对于预处理采样器来说要容易的多。

b.执行性能的高低

首先，预处理采样的方法需要对数据至少进行一次额外的读写操做，大量的磁盘I/O代价会严重阻碍系统效率的提升。其次，预处理本身也是一个任务，这种方式相当于增加了一个新任务，系统将会启动多个Map Task和Reduce Task来完成该任务，这些Task的启动和销毁也将耗费可观的系统资源。

c.用户使用的便利程度

预处理采样器是Hadoop应用实例，用户在执行自己的程序之前，需要先运行一遍该程序，并且将预处理的结果转换成变量，写入到自己的应用程序中去。侵入式采样器是Hadoop内置采样器，对于用户是透明的，用户无需关心采样器的内部细节就可以方便有效的使用，因此侵入式采样器用户使用起来会更方便。

本文Hadoop负载均衡系统所使用的采样器是侵入式的采样器。

**(2)采样时机**

由于采样器需要对任务的输入数据和Map Task执行的结果进行采样，所以将采样器设计到原型系统的Map阶段。

Map阶段的细节如下如图3-3所示。

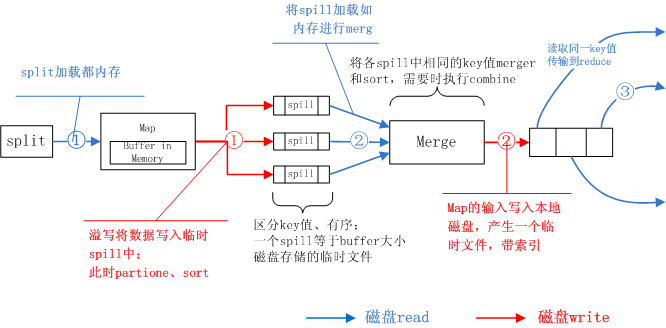


图3-3 Map执行过程示意图

1)Map阶段的执行流程

由于Map函数对文件集的操作是并行执行的，所以Map阶段首先对文件进行切割，得到一系列的小切片，即Split。一个文件存入HDFS时，当它的大小超过了用户设定的Block大小（默认值64M）时，该文件将被切分成多个数据块，而且切分的时候不考虑文件的内部逻辑结构，所以一条记录有可能被切分到两个不同的Block中，默认情况下，HDFS的一个Block块对应于一个Split，导致被切分开的记录存储在不同的Split中，由于一个Map函数处理一个Split，因此Map函数无法得到完整的记录。为了解决这个问题，Hadoop系统允许用户自定义Split的切分策略，从而保证Split中每一条记录都是完整的。

当一个节点成功申请到一个Map任务时，Hadoop会为该任务开启书写器线程，该线程能够依照用户自定义的输入格式来读Split中的信息，并将其中的信息输出成“键-值”对的形式，遇到边界记录时，进行边界处理，保证Split中各个记录都是完整的，从而保证运行结果的正确性。

用户自定义输入的格式不一定是有意义的，Hadoop系统中的TextInputFormat将该行的文本内容作为值，将文本的的偏移量作为键，从而组成“键-值”对，在实际的应用中，文本偏移量的使用的机会很少。

Map函数对输入的键值对进行处理之后，需要将结果也以键值对的形式输出出来，键的类型和值的类型由用户自定义，但这两个类型要跟Reduce函数的输入键值对的类型相一致。系统将Map函数的处理结果（键值对）存入环形内存缓冲区中。

环形内存缓冲区默认的容量是100M，阈值系数是0.8。当环形缓冲区已占用空间达到总容量的80%时，缓冲区将被锁定，缓冲区中的内容将被溢写到本地磁盘上。溢写磁盘之前，系统的会对这些数据执行（系统默认或者用户自定义）分区函数，并按照分区值将结果溢写到不同的文件中。最终相同分区的溢写文件被归并成一个大文件，作为Reduce阶段Shuffle阶段拷贝的对象。

2)相关的概念

a.溢写（Spill）

Map输出结果被存储内存的环形缓冲区中，但是系统的内存资源是有限的，缓冲区的大小也有大小的限制，系统默认情况下是一百兆，当环形缓冲区的大小超过阈值时，就将缓冲区中的内容集中输出到本地磁盘上，内存中的数据经过分区、排序和合并之后最终被输出到磁盘上。

b.合并（Combine）

Hadoop系统为了减少在Map阶段溢写的数据量，溢写之前会将会将相同的键值对进行合并处理，对合并之后的结果做局部的Combine，将Combine之后的结果写入磁盘中。但是combine使用范围比较有限，它只适合于能够简单切分的任务。

c.归并（Merge）

每一次溢写都会产生一个溢写文件，当Map任务需要输出的数据量比较大时，将会产生许多次的溢写操作和多个溢写文件，Hadoop系统会将所有的溢写文件归并成一个文件，这个过程就是归并的过程。在归并的同时，还讲对数据做排序操作，减少Reduce端排序的压力。

3)采样时机确定

通过以上对Map执行流程，溢写（Spill），Combine以及Merge的详细分析，在Map Task将所处理的Split加载到内存并被Map函数处理之后写入到环形内存缓冲区之间进行采样处理最为合理，如图3-4所示。有两个比较合适的采样时机，即加载时采样和溢写时采样。

一方面，这两个时机下可以得到完整的数据对象，采样之前Mapper函数已经执行完毕，而采样的对象是系统所的输入数据和Map阶段的输出数据，这两个时机下都可轻松获取完备的数据对象。另一方面，这两个时机下，数据对象都是存在于内存中的，此时进行采样不会造成过多的I/O代价。本文hadoop负载均衡系统采用的是加载时采样。

|  |
| --- |
| 图3-4 Map阶段的细节流程图 |

1. **定时采样算法**

启动一个定时器线程，并设定一个阈值，当定时器的值不小于该阈值的时候就将该条记录元组采集到样本中，并给定时器值为0，知道整个程序完成为止。算法伪码如下。

|  |
| --- |
| **Sampler** |
| **Input:**  *M*:Map Task’s Output  *N*:Job’s split data  *Threshold*:timer’s Threshold  **Output:**  *Scale[M/N]*:*M/N*’s scale（The number of tuples and the size of the space occupied）  Start Timer’s thread  **for** *i* from *1* to *N* **do**  Map();  **for** *j* from *1* to *M* **do**  **if** Timer >= *Threshold* **do**  *Threshold=0；*  Pick *M[j]* update to *Scale[M/N]*;  Pick *N[i]*;  **If** *Scale[M/N] don’t contain N[i]***do**  将*N[i]* update to *Scale[M/N]；*  **end if**  **end if**  **end for**  **end for**  **return** *Scale[M/N]*; |

3.3 系统详细设计之信息整合模块

信息整合模块主要负责采样信息的分类，去重和归并操作。

3.3.1 信息分类

对于样本数据，系统要对其进行进一步处理，其中数据分类就是其中最重要的一项操作。

**(1)按照类型分**

由于任务输入数据的种类繁多，可能是网页数据、电子书，也可能是某一段音频数据、视频数据等。可以根据输入数据的种类，对样本进行分类。

分类方法：不同类型的数据的文件的后缀名是不同的，比如txt，doc，ppt等表示文本文件，MP3表示音频文件，MP4，rmvb，flv，avi表示视频格式。信息整合模块通过获取各个Map Task所处理的Split块所属的文件名，可以得到文件的类型信息，依此为基础，根据类型对采样数据进行分类。

**(2)按照来源分**

由于采样模块采集的数据有两个来源，一是任务的原始输入数据，另外一个便是Map Task的输出数据。

分类方法：原始输入数据是Map Task的Map()函数的输入，因此当Map()函数执行前，采样数据来源于原始输入数据。Map Task的输出数据是在Map()函数运行完毕之后才生成的，在Map()运行完毕之后进行采样，此时获取的数据属于Map Task的输出数据。

**(3)按照语义分**

如果系统处理的数据是某个班级学生的相关信息。当预期的目标信息是求取学生数学平均成绩时，那么采样数据就可以只按照是否是数学成绩来分，而不必关心语文成绩或者其他与数学成绩无关的信息。

分类方法：依据预期的目标，根据用户关心的信息对数据进行分类。

3.3.2 信息去重

采样信息中常存在着重复信息，比如采集某个学生的数学成绩，采集得到的信息通常会包含很多其他的信息比如性别信息，当需要再次采集该同学的英语成绩时，那么他的性别信息就再次被采集。当用户对该学生的性别信息比较关心的时候，那么该样本中存在着多个相同的信息。

信息去重，就是删除样本中重复出现的信息，这样可以避免信息冗余，节省数据的存储资源，同时也便于对采样信息进行管理和维护。

3.3.3 信息归并

由于样本数据众多，种类也各不一样，对于某个Partition值为N的分区，若用户关心分区N中包含的记录元组个数以及所消耗的存储资源，那么就需要将所有分区值为N的样本数据归并在一起，所有分区值为N的采样数据都被统计到N分区中，而且N只包含有分区值为N的样本数据。

对于某个分区N，将该分区中所有的样本数据归并起来，删除样本数据，只保存归并结构，这就是数据整合。经过信息整合，样本数据将被准确分类和大幅的压缩，减少存储开销和读写磁盘的I/O代价。

3.3.4 整合模块的设计

**(1)结构设计**

整合模块的主要内容有分类器（Classifier），信息去重器（Duplicate removal Processor），信息整合器（Merge）三部分构成。其中分类器的主要负责对采样信息的进行分类，并将结果传递给信息去重器，信息去重器接收到分类器发来的消息之后，对数据进行去重操作，最后信息整合器将去重之后的这些信息归并到一起，自此信息整合模块的任务才算是真正的完成。信息整合模块的整体框架如图3-5所示。

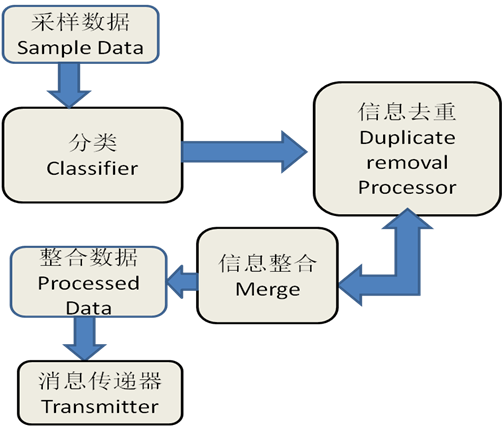


图3-5 信息这个和模块设计图

1. **整合算法**

将整合模块的算法如下所示。

|  |
| --- |
| **Processor** |
| **Input:**  *S*:Sample data get from Sampler  *Class*: data’s class  **Output:**  *Processed[X][S]*: Sample data which has been processed by Processor  **Related data:**  *M*:Map Task’s Output  *N*:Job’s split data  Start Processor’s thread  **for** *i* from *1* to *S* **do**  *Class*=Classifier ();  **if** *Class* Belongs to *M* **do**  Pick *S[i]*;  **if** *Processed[M][S] don’t contain S[i]***do**  Update *S[i]* to *Processed[M][S]*;  **end if**  **end if**  **else if** *Class* Belongs to *N* **do**  Pick *S[i]*;  **If** *Processed[N][S] don’t contain S[i]***do**  Update *S[i]* to *Processed[N][S]*;  **end if**  **end else if**  **else** do nothing **end else**  **end for**  **return** *Processed[X][S]*; |

3.4 系统详细设计之决策模块

子节点的采样信息，经过信息整合模块处理之后，经由消息传递模块发送到决策模块。决策模块的主要工作是对子节点发送过来的采样信息进行进一步的分析，分析集群中的数据分布情况，判断是否存在数据倾斜以及倾斜的程度，最后依据数据倾斜的状况，决策器对分区进行重组，使得重组后的各逻辑分区的负载量比较均匀，从而达到平衡Hadoop集群负载的目的。

3.4.1 数据倾斜

**(1)瓶颈任务**

1）瓶颈任务的概念

并行计算中，各个任务的执行时间是不同的，由于机器异构性，机器承担任务量的大小以及数据规模等因素，使得某个任务的执行速度缓慢，系统长期处于等待该任务完成的状态中，导致任务周期延长，该任务就被称为瓶颈任务。

2）瓶颈任务的成因

首先，机器的异构性。不同配置的机器拥有者不同的性能。对于一个特定的任务，不同配置的机器来执行，执行时间可能都会有比较大的差异。针对于此，本文负载的代价模型假设集群中各个机器的配置是相同的。

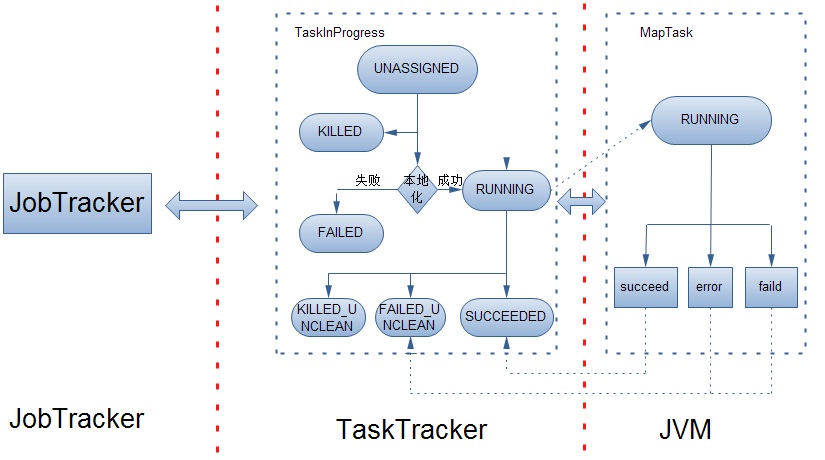
其次，应用的类型不同。如page rank 或者shortest path等，由于算法不同，每条记录消耗的成本不太一样，相对于数据密集型的问题来说，计算密集型的问题消耗单条记录将会消耗更多的计算成本。针对于此，本文选用了两个wordcount和multiple tables join作为实例对系统进行对比，其中word count是数据密集型计算，word count的算法时间复杂度和空间复杂度都远不及multiple table join高，这部分实验会在第五章中着重介绍。

最后，系统分配的任务量不同。对于Hadoop系统来说，一个Reduce任务处理一个Partition分区， 某个Partition分区来说，它常常会包含多个key。而Partition分区中会包含哪些种类key的决定因素，除了未知的输入数据之外就是系统的Partition()函数。Hadoop集群中采用的分区策略大多是哈希函数法。由于哈希算法本身的局限性，按键值进行哈希或多或少的造成数据倾斜，使得某个分区的数据过多或者过少。当系统处理的数据是均衡分布的，即每个key对应的记录的个数相同，那么包含key个数越多的Partition分区就会包含越多的记录元组。在不考虑集群机器配置差异时，针对同一个应用实例，那么包含key个数最多的那个分区的负载量是最大的，运行时间也最长，因此这个任务就成了集群的瓶颈任务。

本文研究的重点是最后这种情况，即由于不同的Reduce Task处理的Partition分区的所包含的记录元组个数相差太大，导致的系统负载不均衡的情况。那个包含记录最多的Partition分区必定成为系统的瓶颈任务。

**(2)Map端数据倾斜**

图3-6 Map Task的执行流程图



由以上图3-6所示的Map Task的执行流程可知，Map的输入是Split，是一个逻辑的切片。Hadoop中默认的该逻辑切片的大小是64M，一个Split对应一个Map Task，因此每一个Map Task的输入的大小都不会超过64M（文件最后的那个Split的大小可能会小于64M）。因此一个Job的各个Map Task之间处理的数据的量大小是差不多，数据倾斜情况并不明显。而JobTracker分配Map Task的时候主要是根据申请任务的TaskTracker的Map Slot的个数，当机器节点的Map Slot个数比较多时，那么它便可以同时运行多个Map Task任务，即当Map Task个数足够时，TaskTracker能够同时执行的Map Task个数是该节点的Map Slot的个数。因此由于Map Slot在集群中分布不合理时，大量的Map Task被分配到集群中个别的机器上运行，从而引发Map阶段的数据倾斜。

整体上说，Hadoop系统Map阶段的数据倾斜不明显，而且也不会随着任务输入数据的种类和规模而有明显的变化。

**(3)Reduce端数据倾斜**

1)ReduceTask的执行

JobTracker启动任务时通过每一个TaskInProgress来launchTask，这时会把Task对象（即MapTask和ReduceTask）序列化写入相应的TaskTracker服务中，TaskTracker收到后会创建对应的TaskInProgress（此TaskInProgress实现非JobTracker中使用的TaskInProgress，作用类似）用于监控和调度该Task。ReduceTask.run方法开始和MapTask类似，包括initialize()初始化，runJobCleanupTask()，runJobSetupTask()，runTaskCleanupTask()。之后的工作主要有三个，即：Copy、Sort、Reduce。

a.Copy

Copy就是将Map的处理结果拷贝到本地。该任务具体流程如下。

①索取任务。由GetMapEventsThread线程完成。该线程的run方法不停的调用getMapCompletionEvents方法，该方法又使用RPC调用TaskUmbilicalProtocol协议的getMapCompletionEvents，该方法使用所属的jobID向其父TaskTracker询问此作业个Map任务的完成状况。返回一个数组TaskCompletionEvent events。TaskCompletionEvent包含taskid和ip地址之类的信息。

②得到Map Task的执行节点信息后，系统开启一个专门的线程，负责数据的拷贝工作，将Map Task的相应的结果拷贝到Reduce的执行节点上。MapOutputCopier的run循环调用copyOutput，copyOutput又调用getMapOutput，运用HTTP协议进行远程拷贝，直到所有的数据拷贝完毕。

③最后内存Merger线程InMemFSMergeThread和文件Merger线程LocalFSMerger同步工作，对下载过来的文件执行归并排序处理，从而降低输入文件的数量，为后续的排序工作减负。

b.Merg/Sort

系统将所map阶段的结果全部拷贝到Reduce任务的执行节点完毕时，对这些文件做归并处理。使用工具类Merger归并所有的文件，在归并的同时对数据进行排序处理。

c.Reduce

Reduce是Reduce任务的最后一个阶段。系统会准备keyClass，valueClass和Comparator，最后调用runOldReducer方法执行reduce函数。Reduce输出的结果都被存储在分布式文件系统中，而非本地磁盘。输入方面，ReduceTask会用KeyClass、ValueClass、KeyComparator等之类的自定义类，构造出Reducer所需的键类型和值的迭代类型Iterator。

2)Reduce端数据倾斜分析

Reduce 端在执行多线程的归并排序（multi-pass merge sort）的情况，运行较慢reduce task与I/O相关的度量会比其他reduce大很多。将Reduce端数据倾斜状况总结如表3-1所示。

表3-1 Reduce端典型的数据倾斜状况

|  |  |  |
| --- | --- | --- |
| 操作 | 系统输入 | 导致结果 |
| Join | 其中一个表较小，但是key集中。 | 某些Reduce处理的数据量远高于平均值。 |
| 大表与大表，但是连接属性的0值或者空值太多。 | 0值或者空值由一个reduce处理，任务量大，处理时间长。 |
| Group by | group by 维度过小，某值的数量过多。 | 处理某值的reduce灰常耗时。 |
| Count Distinct | 某特殊值过多。 | 处理此特殊值的reduce耗时。 |

a.造成Reduce端数据倾斜的原因如下。

①key分布不均匀

②业务数据本身的特性。

③设计表格时考虑不周。

④某些具体应用引发数据倾斜。

b.Reduce端数据倾斜的表现形式

其他任务都已执行完毕的情况下，瓶颈任务的运行时间过长。单一reduce处理的记录数与平均记录数差异过大。

3.4.2 代价模型

当JobTracker接收到来自集群中众多TaskTracker发来的采样信息之后，首先，JobTracker的信息整合模块会对这些信息进行整合，并将结果发送给JobTracker中的决策模块。其次，决策模块对这些样本信息进行分析，掌握各个Key的元组记录个数、占用的磁盘空间以及在集群中分布情况，依据建立的代价模型计算出各个分区的负载量。最后，运用启发式贪心算法对分区进行重组，使得重组之后的各逻辑分区的负载量大小差不多，并以集群负载的方差最小为目标，并最终形成Partition的重组方案。

数据的离中趋势，它指的是数据具有偏离中心位置的趋势，反映了一组数据本身的离散程度和差异性程度。方差能综合反映一组数据的离散程度或个别差异程度，因此本文选取方差作为集群负载均衡程度的衡量标准。

**(1)问题描述**

1)问题的描述

Map阶段唤醒缓冲区溢写磁盘时，会执行用户程序的Partitioner()函数，将环形缓冲区中的信息映射到不同的Partition分区，并写入磁盘的溢写文件中，Map Task最终会将所有的溢写文件归并到一起并对其进行排序，最终Map任务输出到同一个分区的数据也被合并到一起。

假设系统中有N个Partition分区和M个Reduce任务，并且一定有N>=M， 那么如何将Map阶段产生的N个Partition分区指派到M个Reduce上，才能使得这M个Reduce的任务负载的方差最小呢？如图3-7所示。

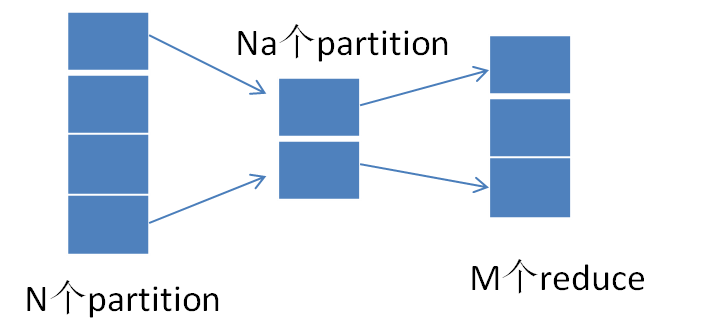


图3-7 分区重组问题示意图

2)假设条件

a.假设集群中各个机器的硬件配置相同，处理性能也相同。

不同配置机器拥有者不同的性能，给它们相同的一个任务，执行时间可能都会有相对比较大的差异。针对于此，本文的负载均衡模型假设集群中各个机器的配置是一样的，本文系统认为集群中各节点的数据处理能力是相同的。

b.假设集群中各个节点之间的网络传输速度是一个恒定值。

Hadoop集群在不运行任务时，Master节点和Slave节点通过心跳信息进行通信，当Hadoop执行任务时，除了心跳信息之外，还常常牵涉到数据迁移，消息通讯等额外的网络通信。网络在不同的负载时，它的通讯能力是不同的，本文的原型系统假设网络通信的速度是一个恒定值，不随着网络通信量的大小而改变。

**(2)影响因子**

1)元组记录的条数

元组记录的条数代表着模型所要处理数据的规模，它的多少直接影响了任务的负载量的大小。元组记录个数和负载量的成正相关，即元组记录的条数越多，那么这些元组数据对应的负载量也更大。

2)元组数据量

元组的数据量，即元组记录所占用的物理空间，它决定了Reduce的执行节点要耗费的内存空间以及网络传输时间的消耗情况。当某个Reduce处理的Partition中包含的元组数据量非常大时，那么数据迁移的网络代价，以及文件读写的I/O代价都将将不可忽略，因此元组的数据量是影响负载大小的一个重要因素，并且它们之间成正相关。

**(3) 数学建模**

1)变量的定义

为了对Hadoop系统建立负载均衡的代价模型，现将变量定义如下：

a. Partition分区总个数：，



b. Reduce任务个数：M，

c. Reduce函数算法复杂度：O，

d. 第i个Partition分区:，

e. Partition分区i的元组个数：，

f. Partition分区i的元组占用的磁盘空间：，

g. 第i个Reduce任务:，

h. Partition分区i被分配到第j个Reduce任务上：，

i. Partition分区i没有被分配到第j个Reduce任务上：，

j. Partition分区i被分配到第j个Reduce任务上时的负载代价，

k. Hadoop集群的负载总代价：，

l. Hadoop集群的负载方差：，

m. Hadoop集群的Reduce负载均值：，

n. Hadoop集群的Reduce负载中位数：，

o. Reduce的负载量：,

p. Partition分区和Reduce任务的对应矩阵:

2)约束条件



a. Partition分区的个数不小于Reduce任务的个数，即：

b. Partition分区是否被分配到Reduce任务上，只有被分配了和没有被分配两种状态，即0和1。

c. Partition分区和Reduce任务的对应矩阵的任意一行的和都必须为1，即：对于某一行t，有成立。

3)目标函数

Partition分区i被分配到第j个Reduce任务上时的负载代价的计算公式如公式（3-1）所示。其中α，β分别对应于元组个数和元组占用空间的权重值，由经验可得α=0.01，β=0.3，,

 公式（3-1）

Hadoop集群的Reduce端总负载的计算公式如公式（3-2）所示。

 公式（3-2）

Hadoop集群的Reduce端负载的均值计算公式如公式（3-3）所示。

 公式（3-3）

Hadoop集群的Reduce 的负载计算公式如公式（3-4）所示。

 公式（3-4）

Hadoop集群的Reduce任务的负载方差值计算公式如公式（3-5）所示。

 公式（3-5）

4)模型的有效性分析

该模型通过假设条件，忽略了集群中机器的异构性和网络的不稳定性，为代价模型营造了一个相对稳定的外部环境。元组个数，即元组规模，直接影响着负载的大小。元组规模和负载量成正相关，对于同一个算法，元组规模越大，那么负载量越大，所消耗的时间也将越长。磁盘I/O代价的高低取决于元组占用的空间的大小，当元组占用的空间越大，磁盘操作的负载量也就越大，两者之间成正相关。元组个数信息和占用的空间信息分别从时间和空间两个角度影响负载，因此，将其作为负载的影响因素既合理又有效。

方差是衡量一组数据波动大小的量，是测算数值型数据离散程度最重要的方法。方差越大，表明数据的稳定性越差，波动也越大。目标函数选用方差，方差的大小能够直观的反映出系统的负载均衡程度，方差越小表明集群的负载越均衡。

**(4)模型实现**

1)决策模块在原型系统中的位置

子节点将采样结果经过信息整合模块的进行进一步处理，被发送到集群的JobTracker节点上，决策模块收到上述信息之后，根据原型系统的负载均衡模型，计算各个Partition分区的负载代价，并产Partition分区的重组方案，之后的Shuffle过程，就按照该方案进行。

决策模块整体是位于Hadoop的JobTracker（Master）中，如图3-8所示。



图3-8 决策模块位置示意图

2)决策模块详细设计

决策模块的主要包含以下几个部分，信息维护器，决策器，分区重构器和任务推送器这四部分，如图3-9所示。

a.信息维护器，负责维护决策的输入数据。在该系统中，主要是维护各个Partition分区的采样数据。

b.决策器，实现了决策算法，主要负责计算各个分区的负载代价，对解空间进行搜索，找到最优的Partition任务分配方案。

c.分区重组器，跟据决策器产生的Partition分区任务分配方案，方案中会存在多个Partition分区对应于一个Reduce任务的情况，分区重组器的主要任务就是将这些Partition分区拟合成一个逻辑分区。

d.任务推送器，负责将任务分配方案，推送到Reduce的执行节点。



图3-9 决策模块的详细设计图

**(5)相关算法**

1）信息维护算法。

决策模块的信息维护器主要维护决策器的输入数据，其中最重要的是对表3-2和表3-3的维护。

表3-2主要描述了Hadoop集群中各个Partition分区的元组记录条数在各个节点上数据分布情况，其中表3-2中表示Node\_i上存储的分区值为Partition\_j的记录元组数。

表3-3则描述了Hadoop集群中各个Partition分区在各个节点上占用的磁盘空间情况，其中表3-3中表示分区值Partition\_j记录元组在Node\_i上占用的磁盘空间大小。

表3-2 分区元组个数信息分布表

|  |  |  |  |
| --- | --- | --- | --- |
|  | Partition\_0 | Partition\_1 | Partition\_… |
| Node\_1 | -- | -- | -- |
| Node\_2 | -- | -- | … |
| Node\_... | -- | -- |  |

表3-3 分区元组占用的空间信息分布表

|  |  |  |  |
| --- | --- | --- | --- |
|  | Partition 0 | Partition 1 | Partition … |
| Node\_1 | -- | -- | -- |
| Node\_2 | -- | -- | … |
| Node\_... | -- | -- |  |

经常有其他的进程或者线程对这些表格进行读写操作，因此在系统设计的时候特别关注了信息的互斥性，即当某个进程或者线程需要更新数据时，那么就等待一段时间，直到其他程序释放改数据的控制权限时，再进行信息维护。并且一旦进入信息的维护流程，在信息维护完成之前，都不会释放该表的更新权限。具体算法如图3-10所示。

2）决策器

实现了决策算法，主要负责计算各个分区的负载代价，对解空间进行搜索，找到最优或者较优的Partition分区分配方案，即该方案能够能够使得集群各个Reduce端负载的方差最小。

Partition分区分配方案，本质上是Partition分区和Reduce任务之间的对应关系，由于Partition分区的个数不会比Reduce任务数少。Partition分区和Reduce任务之间的对应关系满足满射的前提，当两者的数量一样时，那么他们之间的对应关系肯定一一映射。当Partition分区的个数比较多的时候，那么他们之间的对应关系就变成了多对一，此时也满足满射的条件，任何一个Reduce任务至少对应一个Partition分区。

根据上一章节介绍的决策模型，本文的原型系统采用贪心启发式搜索对决策模型的解空间进行搜索。

启发式搜索是一种将启发信息作为引导的搜索策略，事物的条件，状况，动态变化规律等都可以作为启发信息。一个好的启发函数常常使得问题的解决变的容易。

图3-10 信息维护算法流程图



贪心算法是通过利用局部最优解来逼近最优解的一种算法，这种算法得到的结果往往不是最优解。算法的核心是优化测度的选择，它根据当前的状况和用户选取的优化测度，选出当前情况下最最优的下一步，而不关心这一步的选择的整体最优性，它就这样一步步的迭代到最后一步，产生整个的运行方案。

贪心算法是一种简单的算法，由于它不对解空间的进行全搜索，所以它的算法复杂度很低，执行效率很高。

本文选取的启发式贪心算法就是将启发式算法和贪婪算法融合到一起的一种算法。启发式算法的核心是选择一个比较好的启发式函数，针对于多数的问题贪心算法也是非常有效的，将贪心算法的和启发式算法结合到一起，算法能够保证找到比较优秀的结果，又因启发式算法的剪枝非常高效，所以启发式贪心算法的执行效率也会比较高。

启发式贪心算法的几个优先原则：

a.负载量大的Partition分区优先被分配；

b.负载量小的Reduce分区优先作为分配对象；

启发式贪心算法思路如下。对Partition分区任务按照负载量从大到小进行排序，从中挑选出负载量最大并且尚未被分配（大负载分区优先的原则）的那个Partition分区，把该分区分配给此时负载最小的那个Reduce任务上，将该Partition分区从Partition分区中标记为已经分配，并将该Partition分区的负载量累加到该Reduce任务上。依次遍历完Partition分区所有没有分配的分区，整体的分配方案就产生了。该算法伪代码如下所示。

|  |
| --- |
| Heuristic Greedy Search algorithm |
| **Input:**  *M*:Reduce number  *N*: Partition number  *R[M]*:Load of every Reducer  *Q[N]*:Load of every Partition  *sign*  *set*  **Output:**  *Scheme\_Map[M,set[Integer]]*:*Task allocation scheme*  Sort(*Q[N]*);//*Q[N]* is sorted from big to small  **for** *i* from *1* to *N* **do**  **if** Q[i] has not been scheduled **do**  *sign***=**1;  **for** *j* from *1* to *M* **do**  **if** *R[j]*<*R[sign]***do**  swap(*j,sign*);  **end if**  *set.clear();*  *set.add(i);*  *Scheme\_Map.put[sign,set];*  Sign Q[N] had been scheduled*;*  **end for**  **end if**  **end for**  **return** *Scheme\_Map[M,set[Integer]]*; |

3）分区重组器

Hadoop原系统中，由Partition函数决定的Partition分区和系统的Reduce任务是一一对应的，它们之间映射是满射。原Hadoop系统给某个Reduce任务分配并且只分配一个Partition分区。

本文系统的Reduce端的任务的指派粒度是Partition分区。实际上，最好的粒度是元组记录的key，但是绝大多数Hadoop任务的数据规模都相对比较大，数据来源和数据类型也是多种多样，因此一个Hadoop工作中key的种类非常多，对key的维护代价和网络传输代价也将变得相当的巨大。因此跟Hadoop系统一样，原型系统依旧选择Partition分区作为Reduce任务分配的粒度，只是该分区的粒度相比于原Hadoop分区粒度小许多。

在原型系统分区函数的基础上，增加了一个膨胀系数N，经过该分区函数得到的分区的个数将是Reduce任务个数的N（N为整数且N>=1且N<=任务的key的种类数/Reduce个数））倍。当N=1时，此时产生的分区的数量跟（原Hadoop系统是一样的。N越大的时候，产生的Partition分区的个数就越多，单个Partition分区所包含的key的种类也将变少，粒度就更加精细。当N的数量等于任务的key的种类数/Reduce个数的值时，那么一个Partition分区中只包含有一个key，一个Reduce任务就只处理一个key。系统对Partition分区的维护代价将随着N的增加而增大，因此选择一个适当的N值将会对系统的性能产生很大的影响，具体的取值见第五章。

决策器产生的Partition分区任务分配方案中会存在多个Partition分区对应于一个Reduce任务的情况，分区重组器的主功能是将这些Partition拟合成一个逻辑分区。本文设计的原型系统就是将逻辑分区作为基本单位进行Reduce任务分配的。

4）任务推送器，负责将最终的任务分配方案，推送到Reduce的任务执行节点。

3.5 系统详细设计之消息传输模块

消息传输模块是联络数据采样，信息整合和决策模块的桥梁，它主要负责各个模块之间的联络和消息传递，它是原型系统不可或缺的组成部分。

该部分的是通过调制心跳信息和加载配置文件实现的。下面的小节中会对两部分做详细的介绍。

3.5.1 心跳信息的调制

**（1）心跳机制原理**

Hadoop集群是Master/Slave模式，Master包括NameNode和JobTracker，Slave包括DataNode和TaskTracker。Hadoop的主节点（Master）启动的时候，会开一个ipc的服务器进程来等待子节点（Slave）的心跳信息。Slave启动时会连接Master，并每隔3秒钟主动向Master发送一个“心跳”（通过“heartbeat.recheck.interval”属性来设置），向主节点汇报自己的状态，主节点也通过心跳信息向子节点发送指令，来完成对整个集群的控制。

NameNode与DataNode之间的通信，JobTracker与TaskTracker之间的通信，都是通过“心跳”完成的。心跳信息是Hadoop集群的重要机制，Hadoop集群的Master节点就是通过心跳信息来获取集群的状态信息的。

心跳信息的调制机制，就主要是利用Hadoop的集群的心跳机制，来完成各个节点之间的消息传递功能的。

**（2）调制心跳信息**

如图3-11所示，其中原Hadoop心跳信息中包含了几种不同类型的信息，如机器硬件状态信息，Hadoop相关参数的状态信息，任务状态信息和任务分配信息。笔者在心跳信息中添加了采样信息，并对任务分配信息进行了扩展。

采样信息中主要存放的是各个子节点采集到的各个partition或者key的元组个数，以及消耗的空间字节数信息，另外还包含HostName，IP等在内的辅助信息。

在任务分配信息中添加了两部分信息，一部分是map任务所要处理的Block信息，包含该Block块的ID、存储节点等信息。另外一部分是重组的逻辑分区信息，由于Hadoop负载均衡系统中所处理的分区是由多个实际分区组合起来的逻辑分区，一个Reduce任务往往要处理多个实际分区，该部分信息包含该逻辑分区所对应的多个实际分区信息（分区值）。



图3-11 心跳信息的调制图

3.5.2 加载配置文件

**(1) 数据加载**

1)配置文件设计

原型系统将配置文件设计成xml格式，通过对xml文件的解析可以方便的获取配置文件中的内容，配置文件的命名为UserConf.xml，将它放到/Hadoop/conf/目录下。配置文件的设计如下所示：

|  |
| --- |
| *<?xml version="1.0"?>*  *<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>*  *<!-- Put site-specific property overrides in this file. -->*  *<configuration>*  *<property>*  *<name>my.configuration</name>*  *<value>myconfiguration</value>*  *</property>*  *<property>*  *<name>NodeNum</name>*  *<value>5</value>*  *</property>*  *</configuration>* |

2)配置文件的加载时机

原型系统需要在配置文件中查找所需要的信息时，创建一个配置文件信息的查询请求，并创建一个线程到本地的/Hadoop/conf/目录下找到配置文件，根据需要查找并返回所需的配置信息。

**(2)序列化和反序列化**

查找到配置文件信息以后，系统需要将该信息传回到主进程（NameNode，JobTracker，DataNode，TaskTracker）。Hadoop系统为了提升消息传递效率，用户可以选择对数据进行序列化和反序列化处理。序列化的目标是对信息进行排序和压缩，反序列化是将目标数据解析成原始数据，是序列化的逆过程。

3.6 本章总结

本章主要介绍了Hadoop负载均衡系统，即Hadoop\_Load\_Balance系统的总体设计和详细设计，对系统中的数据采样模块，信息整合模块，决策模块和消息传输模块的功能，原理，架构和涉及到的相关算法都做了详细的介绍。

1. Hadoop负载均衡系统优化

4.1 一种基于Hadoop的多表连接优化策略

Hadoop\_Load\_Balance系统在处理多表连接问题时，大规模的中间结果加剧了集群负载不均衡的状况，从而严重降低了系统效率。为解决该问题，笔者提出一种“替换-查询”方法，该方法通过对连接表建立索引，将预输出的元组集替换为索引信息输出到中间结果，以索引的形式参与多表连接，以此减少中间结果规模，；并运用缓冲池、二次排序和多线程技术对索引信息进行优化管理，加快索引的查询速度，提升系统的整体性能。

4.1.1中间结果集的膨胀问题

Hadoop系统处理多表连接问题时，常常出现中间结果快速增加的问题，主要有三方面的原因。首先，为了防止连接属性出现缺失，本次连接要尽可能的将下一步甚至下几步连接相关的属性信息输出到中间结果；其次，每一步连接之后，中间结果属性列就会增加，伴随着连接次数的增加，输出到中间结果的属性列就越来越多；最后，子连接中间结果要被多次复制，溢写磁盘或者网络传输到其他节点。这些因素使得中间结果快速膨胀，从而导致大量的内存占用，过多的磁盘溢写和不必要的网络传输代价，严重降低了系统性能。

为了解决多表连接中间结果快速增加的问题，本文提出“替换-查询”方法，该方法具体如下，每一次连接结果集输出到中间结果之前，系统对结果集中的记录元组建立非侵入式的Hadoop分布式文件索引（Non-invasive Index of Hadoop Distributed File System，NIHDFS），在保证结果集信息完整性的前提下，用索引替代记录元组信息并输出到中间结果，记录元组以索引的形式参与之后的连接过程。当需要记录元组信息时，该方法通过查询操作快速的实现元组信息恢复。在索引的平均长度比记录元组的平均长度短或者连接表的数量较多的情况下，该方法能够有效的减少中间结果的数据规模并降低其增长速度。

在Hadoop\_Load\_Balance上，修改系统源码并实现“替换-查询”多步连接的处理方法。在与原Hadoop\_Load\_Balance系统的对比实验中，“替换-查询”的多表连接处理方法，能够有效的提升系统处理多表连接的性能。

4.1.2 优化方案的整体设计

图4-1 “替换-查询”方法的系统设计示意图

原Hadoop在处理多表连接问题时，Map阶段对输入的记录元组进行键-值解析，解析出真实的记录元组之后便直接进行Map端的连接操作。优化系统则不然，在执行Map端连接操作之前，系统要对输入的信息依照“替换-查询”方法进行处理。该处理包含三个部分，分别是对连接表建立NIHDFS索引，替换连接表元组信息以及元组信息的快速查询。他们分别由索引化模块，替换模块和查询模块来完成，如图4-1所示。

T3是原始连接表，当该连接表进入系统后，先后进行索引化和替换操作，并用记录元组的索引信息和本次连接的属性信息替换原始记录元组，然后再进行Map端连接操作。Output\_1是子连接的连接结果，IndexT1\_1表示连接表T1中某个记录元组的索引，IndexT1\_1：IndexT2\_1表示连接表T1和T2的一条连接结果，其中只存储着两条记录对应的索引信息。当Output\_1作为该系统的输入时，该系统需要对其中的索引信息进行查询，恢复出原始记录元组并进行解析，然后再进行Map端的连接操作。Map端和Reduce端连接操作的操作内容由具体的多表连接算法来决定。

4.1.3 NIHDFS索引设计

HDFS（Hadoop Distributed File System）是被设计成适合运行在通用硬件上的分布式文件系统。本文以HDFS为基础，建立非侵入式的Hadoop分布式文件系统索引，即NIHDFS。它是根据连接表各记录元组在HDFS存储的物理位置信息建立起来的预连接索引。

NIHDFS主要包含文件的统一资源定位符（File URL），数据块间偏移量（Block Offset），块内记录偏移量（Record Offset），以及记录元组长度（Record Length）四个部分，结构如图4-2所示。统一资源定位符(File URL)是文件在HDFS中存储的逻辑路径，该路径可以选择系统默认值，也可以由用户定义。数据块偏移量（Block Offset）是索引对应记录所在的数据块在文件中的起始偏移量。针对特定数据块中的不同记录，它们的长度和存储的物理位置各不相同，本文用块内记录偏移量(Record Offset)来存储记录在数据块中的位置信息。记录元组长度（Record Length）指记录元组的大小。

如在HDFS中存储着一个维表，该表在HDFS中的逻辑路径是URL；它被HDFS切分成3个数据块，分别是Block\_A，Block\_B，Block\_C，起始的偏移量分别是Offset\_A，Offset\_B，Offset\_C；某条记录存储在Block\_B中，块内偏移量是Offset\_rc，该记录所对应的索引是“URL + Offset\_B + Offset\_rc + Record\_tmp.Length”。



图4-2 NIHDFS结构示意图

由于图4-2所示的索引中只包含一个数据块信息，当维表中的记录元组被切分到两个数据块上时，根据图4-2建立的索引只能够恢复记录元组的部分信息，这显然是不正确的。这种情况下，本文特别设计了如图4-3所示的索引，该索引中包含有两个数据块偏移量，块内记录元组偏移量和记录元组的长度信息，因此它能够完整的存储记录的位置信息。由于记录元组的后半部分数据存储在第二个数据块的起始位置上，所以记录在第二个数据块的块内记录偏移量的值是零。

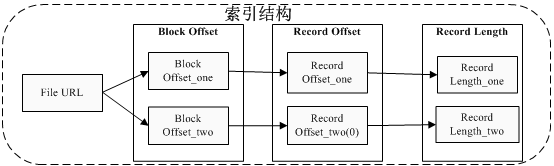


图4-3被切分记录的NIHDFS示意图

4.1.4 基于NIHDFS的“替换-查询”方法

“替换-查询”的多表连接处理是通过索引化，替换和查询三个模块来实现的，如图4-4所示。

**(1) 索引化**

在处理多表连接问题时，Map-Reduce任务的输入有两种类型，一种是前面某个Map-Reduce任务产生的连接结果，一种是原始连接表。由于连接结果中存储的是记录元组的索引信息，所以无需进行索引化操作。当输入是原始连接表时，要根据NIHDFS索引结构对表中的记录元组建立索引。当被索引化的记录元组被切分到两个数据块时，建立图4-3中所示的索引。

**(2)替换**

根据索引化过程中对记录元组建立的NIHDFS索引，用索引信息和本次连接的连接属性组成的键-值对替换掉原始的记录元组，并保证替换操作的正确性和索引信息的完整性。在之后的连接操作中，如其他经典的多表连接算法一样，连接属性主导着整个连接过程，索引信息作为附属信息或被输出到中间结果，或被发送到其他的节点。

**(3)查询**

当Map-Reduce任务的输入是连接结果时，因为连接结果中存储的是记录元组的一系列索引信息，并不包含本次连接的连接属性信息。此时要对NIHDFS索引执行查询操作，到达获取连接属性信息的目的，系统将启动专门线程来完成该查询操作。

4.1.5 “替换-查询”方法的系统实现

查询过程如图4-4所示，TaskTracker是多表连接操作的任务执行节点，该节点中运行着多个Map或Reduce任务，这些任务提出查询请求并将该请求和索引信息打包发送到系统的文件管理中心，即NameNode。

NameNode对索引信息进行解析，获取文件名；查询HDFS中隶属于该文件的所有数据块及副本信息，并从中随机选择一个数据块来响应查询请求；将该数据块的ID以及NIHDFS索引信息发送给数据块所在的机器节点，即查询服务节点。

查询服务节点收到查询请求信息之后，首先根据请求信息中的数据块ID和路径信息在本地定位到该数据块；其次，对NIHDFS索引信息进行解析，依据块内偏移量，准确定位记录元组的物理位置；最后，根据记录元组的起始位置和记录长度，获取完整的元组信息，并将其回送到请求节点。查询请求节点对接受到的记录元组进行解析，从中获取本次连接所需的连接属性信息，从而完成整个查询流程。

当遇到如图4-3所示的索引时，需进行两次查询操作，将两次查询结果按照先后顺序整合起来，得到完整的记录元祖信息。

 图4-4 查询过程示意图

4.1.6 若干细节优化

**(1)NIHDFS优化**

在对NIHDFS索引执行查询操作是，File URL是不可或缺的。隶属于同一个文件的所有记录元组都对应着一个相同的File URL，这导致File URL被多次重复存储，由于File URL相对较长，多次重复存储所占用的磁盘空间是不可忽略的。为了解决该问题，本文用哈希方法把文件标示符集合S与File URL集合做成一一映射，用文件标示符S替代File URL，从而减少File URL长度，降低磁盘消耗。

采用预处理添加分隔符的方法对索引中的块内记录元组长度信息（Record Length）进行优化，具体方法如下，在上传HDFS之前，对连接表进行预处理。一方面，将一条完整的记录元组信息存储为一行，行与行之间的回车换行符作为元组与元组之间的分隔符，用分隔符替代块内记录长度。当元组被载入内存进行索引化操作或者查询服务节点响应查询请求获取记录元组时，都以分隔符作为元组记录结束的标志，而不必关心各记录元组的长度。将记录长度信息从索引中去掉，达到简化索引结构的目的。另一方面，在不同属性之间添加分隔符，降低Map-Reduce任务解析属性信息的难度。

**(2)查询过程优化**

1)缓冲池法

连接过程中，部分记录元组在短时间内会被多次重复查询，每次都要执行完整的查询过程，这是不必要的。本文采用缓冲池法对其进行优化，将经常被查询的记录元组存放于内存的缓存池中，当查询请求到来时，先检查缓存池中是否存在该记录元组。若存在，则直接从缓存池中提取，否则才执行完整的查询操作，获取元组信息。本文采用LRU作为缓存池中记录元组信息的更新算法。

2)二次排序法

在数据块中定位记录元组时，文件指针的移动操作是很频繁的。由于文件指针的单向向下的移动特性，当访问的内容不是按照偏移量由小到大有序时，指针无效移动次数会相当高。本文采用二次排序的方法对系统进行优化。在原Hadoop系统上添加二次排序功能模块，用以对中间结果中索引信息进行排序。排序策略如下：依照数据块间偏移量的大小，对数据块从小到大进行排序，即一次排序；同一个数据块内的记录，依照块内记录元组偏移量的大小，按从小到大顺序进行排序，即二次排序。有序的中间结果能够大幅减少查询的过程中文件指针无效移动的次数，从而提高定位效率。由于Map-Reduce过程中会对记录元组进行排序，二次排序在此基础上进行，开销不大。

3)并行执行

连接过程中，查询记录所在的物理位置集中于数据块的某些区域，并且在一定的时间范围内该区域被查询访问的频率很高，本文把这样的区域称为查询热区。采用并行执行的思想，把单个数据块查询热区的数量上限设置为*M*，对于含有*N*个查询热区的数据块来说，如果*N*<=*M*,那么就开启*N*个线程，每一个线程中设置一个指向对应热区的文件指针，该指针只响应本热区的记录元组查询请求。当*N*>*M*时，采用随机的方法选取两个相邻的热区并将其合并成一个热区，同时对*N*执行减一操作，直到*N*<=*M*，然后采取与*N*<=*M*时相同的操作。

查询热区的并行执行方法，一方面使得数据块拥有同时响应多个查询请求的能力，提升查询操作的并行度，在查询请求集中爆发时依然能够保证查询效率；另一方面，它将指针的定位操作集中到数据块的查询热区上，从而避免了热区之间大量的无效指针移位，提升定位效率。

4.2 一种基于Hadoop的数据本地优化策略

由于数据不在本地，Reduce任务Shuffle阶段常出现大规模的数据远程拷贝现象，网络带宽拥塞，导致瓶颈（Reduce）任务长时间处于数据拷贝阶段，使系统负载更加不均衡。针对该问题，笔者将本地化思想引入系统，运用贪心算法来控制Reduce的任务推送过程，使得各个节点优先接收、处理本地化程度高的任务，以此降低网络开销。

4.2.1 数据本地化的必要性

由于任务处理的数据不在本地，导致了大量数据在集群各个节点之间频繁传输的问题，严重的网络拥塞以及频繁的I/O操作，消耗掉集群大量的网络带宽，磁盘，内存等资源，从而延长了计算时间，降低了系统效率。针对该问题，采取的解决方法如下。MapReduce模型框架在任务（Map任务和Reduce任务）分配过程中，要充分考虑到数据本地化，即本地的数据要尽可能的在本地处理，而不传输到的其他节点上进行处理，从而节省网络带宽等集群资源，提高系统效率。

4.2.2 数据本地化的设计和实现

1. **Reduce阶段本地化**

Reduce主要负责将隶属于一个列表的元素进行合并，它的并行度没有map那么高，但是化简的最终结果常常非常简单，所以Reduce能够适应高度并行的环境。Reduce Task执行过程如图2-7所示。该过程分为三个阶段：

a.将Map任务的处理结果从Map任务的执行节点拷贝到Reduce任务的执行节点，这一阶段也称为copy阶段；

b.对拷贝过来的数据进行归并和排序；

c.经过归并和排序的数据被Reduce函数处理，系统将最终的处理结果存储在HDFS上。

Copy过程中的数据来源主要有两个，一个是Reduce任务所在节点的Map Task输出，就是集群中其他节点的Map Task，即图中从Other Maps拷贝的数据。当后者的数据量非常大的时候，从远端拷贝大量的数据到Reduce任务的执行节点将产生大量的磁盘读写和网络传输代价。

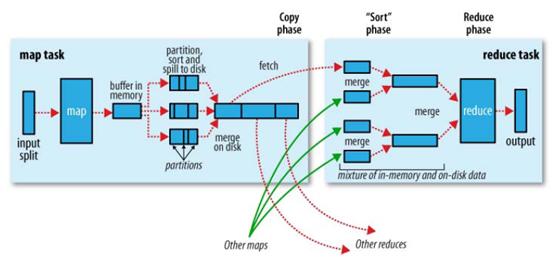


图4-5 Reduce端Shuffle示意图

Reduce端本地化设计图如4-6所示。其中注释1,2是原Hadoop系统的Reduce任务申请过程，首先TaskTracker以心跳信息的方式向JobTracker发送一个Reduce任务的申请请求；然后JobTracker对该系统信息进行分析，当满足以下四个条件是给该节点分配Reduce任务。

图4-6 Reduce端本地化设计图



条件1：达到了Reduce任务的执行时机。Hadoop系统中Reduce任务的开始执行的时机是系统中有一个map任务完成。

条件2：系统没有map任务或者该节点不满足map任务分配的条件。

条件3：该节点上存在空的Reduce slot。

条件4：系统中有剩余的Reduce任务未被执行。

当满足了以上的四个条件之后，在原系统中，JobTracker随机的从没有分配的Reduce任务中随机选择一个，并以心跳信息的方式传送给TaskTracker节点。

原系统Reduce任务到机器节点过程中的随机策略，使得Shuffle阶段从远端读取数据的问题尤为突出，为了解决该问题，笔者设计了Reduce端的本地化策略，具体流程如图4-6注释（1），（2），（3）所示。当TaskTracker满足Reduce任务分配的四个条件之后，触发JobTracker中决策模块的Reduce本地化策略执行，从多个分区中，挑选出一个TaskTracker本地化成都最高的那个任务，并将其以心跳信息的形式推送给TaskTracker节点，进而完成shuffle，merge/sort过程。

1. 介绍的负载均衡的代价模型中，Hadoop负载均衡系统在Map阶段采集的数据非常的丰富，其中包含Map Task输出的元组记录个数和元组记录大小，并存储在系统的主节点上的表3-2和表3-3中。从这两个表中，可以得到各个Partition分区在集群中的数据分布信息。对于某个特定的逻辑分区，跟据以上两个表格中的信息，可以方便的得到该分区在各个节点上占的比重，比重越大的节点存储就意味着存储更多的该逻辑分区元组记录。

算法如图4-7所示。当某个TaskTracke\_t向主节点（JobTracker）申请Reduce任务的时候，决策控制器采用贪心算法进行控制，该方法先遍历未被分配的逻辑Partition分区在集群中分布信息，找出在该节点上所占比重最大的那些逻辑分区（若分区x的元组记录在TaskTracke\_t上占有的比重最大，那么就将x挑选出来），并将它们添加到分区集合Set中。根据Set中元素个数的多少，分以下的三种情况处理。

1)Set为空，将各个Partition分区在该节点的比重值存入到数组TP中，从TP中找出比重值最大的那个Partition分区。若比重值不小于阈值，那么将该分区分配这个节点；否则将阈值缩小为原值的95%。

2)Set不为空，但是Set仅包含一个元素。直接将Set中的这个Partition分区分配给该节点。

3)Set不为空，而且Set中元素的个数不小于2。遍历Set中Partition分区，将这些分区在该节点的本地数据量的大小存入数组TP中，从中选出最大的那个Partition分区，并将该分区分配给这个节点。

图4-7 Reduce端数据本地化算法示意图



4.3 本章总结

本章对Hadoop\_Load\_Balance系统做了两方面的优化。首先，针对系统处理多表连接问题时，中间结果极速膨胀导致降低系统性能下的问题，本章详细介绍了“替换-查询”的优化方法。其次，数据不在本地所引发了集群内大规模的数据迁移，大量的网络带宽被占用导致系统能明显下降，针对该问题，本章介绍了数据本地化的优化方法，并对Reduce阶段本地化的优化方法，算法，设计和实现都做了详细的介绍。

第五章 实验部分

5.1 实验条件及环境搭建

5.1.1 实验条件

软件环境： Ubuntu10.10，JDK1.6，Hadoop-0.20.205.0以及Hadoop负载均衡系统（Hadoop\_Load\_Balance）。

开发环境：Eclipse3.6，JDK1.6，Shell脚本语言，TPC-H，Hadoop集群环境。

硬件环境：5台1.6GHz的八核CPU，16G内存，150G硬盘的曙光系列服务器。

5.1.2 搭建Hadoop集群

利用试验环境中介绍的各种软件和硬件资源，搭建一个拥有五个节点的Hadoop集群环境，该集群中包含一个Master节点，四个Slave节点。Hadoop版本为Hadoop-0.20.205.0。这5个节点的IP地址是172.19.0.118~122，对应的节点的主机名是test118~122，test118作为Master节点，其他的四台机器作为Slave节点,搭建Hadoop集群环境，具体的搭建方法如下：

(1)新建用户dic并将该用户添加到sudoers组中。

1. $:su - //进入root用户
2. $:useradd -m dic //-m的意思是强制创建用户的目录
3. $:passwd dic
4. $:vi /etc/sudoers //添加“dic ALL=(ALL) ALL”到文件中。

(2)安装SSH

ssh免密码登陆原理：ssh证书登录方式:假设想要在ubuntuA访问另外一台ubuntuB, 于是我们需要在A上生成公钥密钥对. 然后把公钥拷贝到B上, 在B上把公钥加入当前用户的.ssh/authorized\_keys里。然后在A上就可以免密码登录B了。如果A需要免密码登陆C,D,E,F...等机器, 只要把在A上生成的公钥文件id\_rsa.pub拷贝到C,D,E,F...上即可。

1. $ ssh-keygen -t rsa -P '' -f ~/.ssh/id\_rsa
2. 将产生的id\_rsa.pub拷贝到其他机器上的/home/dic/.ssh目录下（注意修改文件名称，防止本地的id\_rsa.pub被覆盖,如test118.pub）
3. 在/home/dic/.ssh/创建authorized\_keys文件，然后将复制过来的test118.pub内容追加到authorized\_keys中（cat test118.pub >> authorized\_keys）。
4. 将“172.19.0.118 test118”写入到/etc/hosts中。
5. 注意务必将~/.ssh文件夹的权限设置成700，将authorized\_keys的权限设置成600，不然认证信息将无法加载。

(3)安装Java

1. JDK的版本：JDK1.6。
2. 安装路径：/home/jvm/jdk1.6.0\_26。
3. 在/etc/profile中配置JAVA\_HOME等环境变量如下所示。

|  |
| --- |
| *export JAVA\_HOME=/home/jvm/jdk1.6.0\_26*  *export JRE\_HOME=/home/jvm/jdk1.6.0\_26/jre*  *export CLASSPATH=.:$JAVA\_HOME/lib:$JRE\_HOME/lib:$CLASSPATH export PATH=$JAVA\_HOME/bin:$JRE\_HOME/bin:$PATH:$ANT\_HOME/bin* |

(4)配置文件

1. 配置core-site.xml如下所示。

|  |
| --- |
| *<?xml version="1.0"?>*  *<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>*  *<!-- Put site-specific property overrides in this file. -->*  *<configuration>*  *<property>*  *<name>hadoop.tmp.dir</name>*  *<value>/home/dic/Hadoop-0.20.205.0/hadooptmp</value>*  *<description>*  *A base for other temporary directories.</description>*  *</property>*  *<property>*  *<name>fs.default.name</name>*  *<value>hdfs://172.19.0.118:9000</value>*  *</property>*  *</configuration>* |

1. 配置hdfs-site.xml如下所示。

|  |
| --- |
| *<?xml version="1.0"?>*  *<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>*  *<!-- Put site-specific property overrides in this file. -->*  *<configuration>*  *<property>*  *<name>dfs.replication</name>*  *<value>3</value>*  *</property>*  *<property>*  *<name>dfs.http.address</name>*  *<value>172.19.0.118:50070</value>*  *</property>*  *<property>*  *<name>dfs.web.ugi</name>*  *<value>dic,supergroup</value>*  *</property>*  *</configuration>* |

1. 配置mapred-site.xml如下所示。

|  |
| --- |
| *<?xml version="1.0"?>*  *<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>*  *<!-- Put site-specific property overrides in this file. -->*  *<configuration>*  *<property>*  *<name>mapred.job.tracker</name>*  *<value>172.19.0.118:9001</value>*  *</property>*  *<property>*  *<name>mapred.tasktracker.map.tasks.maximum</name>*  *<value>4</value>*  *</property>*  *<property>*  *<name>mapred.tasktracker.reduce.tasks.maximum</name>*  *<value>2</value>*  *</property>*  *</configuration>* |

1. 配置填写master如下所示。

|  |
| --- |
| *172.19.0.118 test118* |

1. 配置填写slave如下所示。

|  |
| --- |
| *172.19.0.119 test119*  *172.19.0.120 test120*  *172.19.0.121 test121*  *172.19.0.122 test122* |

(5) 启动Hadoop

1. 格式化一个新的分布式文件系统：

$ bin/hadoop namenode -format

1. 在分配的NameNode上，运行下面的命令启动Hadoop集群：

$ bin/start-all.sh

(6)停止Hadoop

1. 关闭Hadoop系统：

$ bin/stop-all.sh

5.2实验内容以及结果分析

[5.2.1 多表连接策略优化前后系统的性能对比](#_Toc380793448)

（1）实验目的：验证Hadoop负载均衡系统“替换-查询”的多表连接策略的有效性。

（2）实验数据：TPC-H的基准数据集，选用nation, region，customer，orders四张表作为输入数据，数据规模为397.6M。

（3）实验用例：多表连接。

（4）Hadoop系统：基于Hadoop的负载均衡系统。

（5）实验内容及及结果分析。

优化前后的两套系统分别执行以上四张表的连接操作，连接的具体过程如图5-1所示。连接算法是经典的Reduce Join算法，两个系统分别执行了五遍该算法，将各个阶段的统计信息整合求取平均值，并将其填入表5-1中，由于优化后系统的最终结果是一系列NIHDFS索引信息，需要将其恢复为记录元组，该阶段消耗的时间是t4，产生的中间结果是size4，原系统在该阶段不需要做任何操作。

表5-1 实验结果汇总表一

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | t1 | t2 | t3 | t4 | 时间合计 | size1 | size2 | size3 | size4 | 空间合计 |
| 优化前 | 11s | 336s | 520s | 0s | 876s | 4K | 469.1M | 630.2M | 0M | 1099M |
| 优化后 | 12s | 237s | 271s | 244s | 762s | 265b | 35.5M | 38.5M | 630.2M | 704M |

优化后的系统产生的中间结果size1，size2，size3远比优化之前要小，连接操作所消耗的时间也比优化前的系统要小很多，空间和时间消耗对比如图5-3和图5-4所示。与优化之前相比，空间性能提升35.5%,时间性能提升12.9%，时间和空间整体对比如图5-2所示。本实验表明，“替换-查询”的表连接处理方法能够有效的减少中间结果的数据规模和增长速度，提升系统效率。

|  |  |
| --- | --- |
| 图5-1 连接过程示意图 | 图5-2 空间和时间对比图 |

|  |  |
| --- | --- |
| 图5-3 各阶段空间消耗对比图 | 图5-4 各阶段时间消耗对比图 |

[5.2.2 数据本地化前后系统的性能对比](#_Toc380793449)

（1）实验目的：验证Hadoop的负载均衡系统Reduce端数据本地化优化的有效性。

（2）实验数据：网页数据2308M。

（3）实验用例：word count。

（4）Hadoop系统：Hadoop负载均衡系统。

（5）实验内容及结果

优化前后的两套系统分别对应实验数据执行word count，统计任务执行Reduce远程拷贝的数据量信息，以及reduce总体执行时间。每个实例各运行五次，并将任务ID，Reduce端远程拷贝的数据量以及Reduce的执行时间等信息的统计结果填入表5-2。

表5-2 实验结果汇总表二

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 实验任务ID | Reduce远端拷贝的数据量（字节） | Reduce运行时间（s） | Reduce运行时间均值（s） |
| 优化前 | job\_201211280048\_0001 | 895,005,332 | 118 | 101 |
| job\_201211280048\_0002 | 665 | 98 |
| job\_201211280048\_0003 | 665 | 87 |
| job\_201211280048\_0004 | 665 | 104 |
| job\_201211280048\_0005 | 941,104,313 | 96 |
| 优化后 | job\_201211282004\_0001 | 36 | 85 | 82 |
| job\_201211282004\_0002 | 36 | 78 |
| job\_201211282026\_0001 | 36 | 85 |
| job\_201211282026\_0002 | 36 | 81 |
| job\_201211282026\_0003 | 36 | 81 |

采用数据本地化策略，所能提升性能的重点是减少Shuffle阶段从远程计算机上拷贝的数据量，将其转换为从本地读取，从而避免了大量的节点间数据传输，用以提升性能。

从该试验的结果中可以看出，优化之前系统的从远端拷贝数据的数据有大有小，不是一个固定的值。这是由于优化前的决策模块并没有考虑负载均衡的因素，某个Reduce任务随机的被推送到集群中任何一个字节点，因此各个任务中Reduce从远端拷贝的数据量都有所不同。优化后系统的决策模块充分考虑了reduce端的数据本地化，因此优化后系统各个reduce远程拷贝的数据量都相对下了很多，节省了大量的网络带宽资源，优化后系统的时间性能提升了20%。

[5.2.3 Hadoop负载均衡系统参数调优及与原Hadoop的性能对比](#_Toc380793450)

1. **Hadoop负载均衡系统参数调优实验**

实验目的：Hadoop负载均衡系统决策时机的参数调优，即map运行到什么进度时执行决策，系统的总体性能最好。

（1）实验数据：5106M的TPC-H的数据。

（2）实验参量。

1. Partition分区膨胀系数：3。
2. 决策执行时机：map执行的范围为从0%到100%，增量为5%。
3. map task和 reduce task都一轮完成。

（3）实验用例：Reduce Join。

（4）实验结果。

每组实验跑3次取平均值，详细结果如表5-3所示。决策的执行时机依据map的执行进度。由于执行时间与系统效率成负相关，因此Hadoop负载均衡系统效率标准化选用的公式是S=1000/t，S越大时说明系统执行时间越短，效率也越高。

表5-3 实验结果汇总表三

|  |  |  |
| --- | --- | --- |
| 执行时机（map的执行进度n\*10%） | Hadoop负载均衡系统执行时间t(s) | Hadoop负载均衡系统执行效率标准化（S=1000/t） |
| 0 | 372 | 2.688172043 |
| 0.5 | 369 | 2.7100271 |
| 1 | 363 | 2.754820937 |
| 1.5 | 357 | 2.801120448 |
| 2 | 354 | 2.824858757 |
| 2.5 | 340 | 2.941176471 |
| 3 | 325 | 3.076923077 |
| 3.5 | 304 | 3.289473684 |
| 4 | 286 | 3.496503497 |
| 4.5 | 278 | 3.597122302 |
| 5 | 266 | 3.759398496 |
| 5.5 | 261 | 3.831417625 |
| 6 | 252 | 3.968253968 |
| 6.5 | 253 | 3.95256917 |
| 7 | 254 | 3.937007874 |
| 7.5 | 266 | 3.759398496 |
| 8 | 278 | 3.597122302 |
| 8.5 | 300 | 3.333333333 |
| 9 | 313 | 3.194888179 |
| 9.5 | 324 | 3.086419753 |
| 10 | 341 | 2.93255132 |

|  |
| --- |
| 图5-5 系统效率随执行时机的变化图 |

1. 实验分析。

由图5-5可知，决策执行时机对Hadoop负载均衡系统效率的影响非常显著。由于采样模块是在map阶段执行的，map进度不同时，采样器采集的数据量也不相同，以采样信息作为输入的决策模块也将制定出不同的分区重组方案。

当系统在map开始，即map执行进度为0%，执行决策，此时采样器尚未采集到数据，决策器的输入为空，此时系统的分区重组策略是随机的选用3个分区组合成一个逻辑分区，由于不知道系统中数据的分布情况，因此无法保证重组后各逻辑分区的负载是均衡的，由图5-5中信息可知，此时系统的效率是最低的。

当map从0%逐渐增加到60%时，由于采样信息的增加，决策模块能够得到更多数据的分布信息，重组方案也更加合理，因此在这个阶段，系统效率提升很快。

当map从60%逐渐增加到100%时，丰富的采样信息使得决策模块能够产出更好的分区重组方案，但reduce阶段Shuffle的开始时间太晚，导致shuffle和map的并行度降低，使得系统性能逐渐下降。

1. 实验结论。

Hadoop负载均衡系统的效率随着决策执行时机的变化会产生明显变化，系统性能提升最快的决策时机是从25%到40%，map运行到60%时执行决策，系统效率最高。

1. **Hadoop负载均衡系统与原Hadoop系统的性能对比实验**

实验目的：将Hadoop负载均衡系统与原Hadoop系统的负载均衡能力进行对比。

实验一：

（1）实验数据：3816M的TPC-H的数据。

（2）实验参量。

1. Partition分区膨胀系数：3。
2. 决策执行时机：map执行至60%。
3. map task和 reduce task都一轮完成。

（3）实验用例：word count。

（4）实验结果。

每组实验跑3次取平均值，详细结果如表5-4所示。其中Reduce阶段的时候包括了shuffle时间（copy,sort,merge三个过程的时间）和reduce()运行时间。

（5）实验分析及结论

由图5-6，图5-7可知，Hadoop负载均衡系统的map阶段运行时间比原Hadoop系统长2秒。这是由于在系统在map阶段添加了采样模块，该模块对系统的输入数据和Map任务的输出数据进行采样消耗了一定的时间。

表5-4 实验结果汇总表四

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 系统 | 原Hadoop系统 | | | | 负载均衡系统 | | | |
| 分区 | P0 | P1 | P2 | P3 | P0 | P1 | P2 | P3 |
| 各分区数据量（M） | 318 | 318 | 318 | 2862 | 961 | 956 | 947 | 952 |
| map阶段时间（s） | 14 | | | | 16 | | | |
| reduce阶段时间（s） | 57 | 57 | 57 | 73 | 63 | 61 | 62 | 63 |
| 任务运行总时间（s） | 91 | | | | 85 | | | |

|  |  |
| --- | --- |
| 图5-6 Reduce任务数据量对比图 | 图5-7 Reduce运行时间对比图 |

Hadoop负载均衡系统的各个Reduce任务的处理数据量基本上相差不多，与原Hadoop系统相比，新系统能够有效的平衡Reduce任务的负载，缩短Reduce端的整体运行时间，从而提升系统的整体性能。原Hadoop系统的运行总时间是91（s），Hadoop负载均衡系统的运行时间85（s），新系统的时间性能提升了6.59%。

实验二：

（1）实验数据：3816M的TPC-H数据。

（2）实验参量。

1. Partition分区膨胀系数：3。
2. 决策执行时机：map执行至60%。
3. map task和 reduce task都一轮完成；

（3）实验用例：Reduce join。

（4）实验结果

每组实验跑三遍，将平均值填入表5-5。

表5-5 实验结果汇总表五

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 原Hadoop系统 | | | | 负载均衡的系统 | | | |
|  | R0 | R1 | R2 | R3 | R0 | R1 | R2 | R3 |
| 各Reduce处理的数据量（M） | 53 | 52 | 53 | 1571 | 481 | 481 | 486 | 482 |
| Shuffle阶段时间（s） | 100 | 102 | 99 | 103 | 104 | 104 | 104 | 100 |
| Sort阶段时间（s） | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 3 |
| Reduce阶段时间（s） | 104 | 104 | 103 | 248 | 150 | 149 | 149 | 148 |
| Reduce阶段时间（s） | 248 | | | | 150 | | | |
| 任务运行总时间（s） | 271 | | | | 176 | | | |

（5）实验分析及结论

由以上的图5-8,图5-9以及表5-5的信息可知，由于原Hadoop系统的Reduce任务R3处理的数据量相差太大，导致R3的运行时间拖慢了系统145(s)时间，R3是这个工作中的瓶颈任务。相比于原Hadoop系统，Hadoop负载均衡系统的各个Reduce任务的处理数据量基本上相差不多，Hadoop负载均衡系统能够有效的平衡Reduce任务负载，缩短Reduce端的整体运行时间。原Hadoop系统的运行总时间是271（s），Hadoop负载均衡系统的运行时间176（s），新系统的时间性能提升了35.1%。

性能提升的空间取决于原Hadoop系统reduce负载失衡的严重程度和Reduce端算法的复杂程度。Reduce任务失衡程越严重，Reduce端算法复杂程度越高，系统的性能提升效果越好。

|  |  |
| --- | --- |
| 图5-8 Reduce任务数据量对比图 | 图5-9 Reduce运行时间对比图 |

5.3实验结论

针对Hadoop\_Load\_Balance，提出并实现了“替换-查询”的多表连接策略，解决了系统处理多表连接问题时，因中间结果爆炸，导致的负载不均衡问题。经过实验验证，系统的空间性能提升了35.5%。针对Hadoop\_Load\_Balance，实现了Reduce端的数据本地化技术，减少了系统Shuffle阶段远程拷贝的数据量，提升了20%的Reduce阶段系统时间性能。

Hadoop\_Load\_Balance的效率随着决策执行时机的变化会产生明显变化，系统性能提升最快的决策时机是从25%到40%，map运行到60%时执行决策，系统效率最高。

实验表明，Hadoop\_Load\_Balance能够有效解决系统的负载不均衡问题，数据倾斜情况下，能明显改善系统性能。在Reduce算法复杂度高的场合中，该系统性能优异，Reduce函数算法复杂度越高，性能提升越明显。良好的系统性能，对代价模型的有效性也进行了验证。

5.4 本章总结

本章主要介绍了实验条件，以及Hadoop集群环境的搭建步骤，并设计实验验证了Hadoop\_Load\_Balance系统的有效性。经验证，该Hadoop\_Load\_Balance能够有效的平衡Hadoop集群任务负载，系统的性能得到了明显的提升。

第六章 总结与展望

6.1 本文研究总结

Hadoop系统处理用户提交的工作时，常出现集群负载不均衡的问题，瓶颈任务大大拖长了整个任务的执行周期。针对这个问题本文提出了一套基于Hadoop的负载均衡系统，该系统通过数据采样来探知系统中的数据分布信息，样本数据经过信息分类，去重，归并之后被传输到的系统的决策模块中，该模块对其进行分析，得知系统中是否存在数据倾斜以及数据倾斜的严重程度，并据此对分区进行重构，通过降低重构后的逻辑分区负载的方差值，来达到均衡各个Reduce任务负载的目的。

针对Hadoop\_Load\_Balance系统处理连接问题时，因中间结果爆炸，导致的负载不均衡问题，通过对中间结果建立索引，用小巧的索引替代冗长的记录元组，从而大大降低了系统I/O代价。在系统上实现了Reduce端的数据本地化技术，减少了系统Shuffle阶段远程拷贝的数据量，提升了系统性能。

6.2 课题研究展望

本文提出的负载代价模型，忽略了集群中各个机器节点的异构性，也没有关注机器CPU，Memory等资源的占用率与系统性能的变化关系。在以后的研究中可以将异构性考虑在内，在为系统分配任务之前，根据机器节点的状态确定是否为其分配新任务，从而对集群负载做到更加精细的控制。另外，本文中只考虑了Map和Reduce任务一轮就执行完毕的情况，针对Map和Reduce任务需要多轮才能执行完毕的情况，将作为下一步要研究的点。

# 参考文献

[1]林伟伟 , 刘波 . 基于动态带宽分配的 Hadoop 数据负载均衡方法[J] . 华南理工大学学报: 自然科学版, 2012, 40(9): 42-47.

[2]吕良干. 云计算环境下资源负载均衡调度算法研究 [D][D]. 鸟鲁木齐: 新疆大学, 硕士学位论文,2010.

[3]程斌. 负载均衡调度系统的设计与实现[D]. 华中科技大学,硕士学位论文, 2011.

[4]Liu Y, Li M, Alham N K, et al. Load balancing in MapReduce environments for data intensive applications[C]//Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on. IEEE, 2011, 4: 2675-2678.

[5]Eltabakh M Y, Tian Y, Özcan F, et al. CoHadoop: flexible data placement and its exploitation in Hadoop[J]. Proceedings of the VLDB Endowment, 2011, 4(9): 575-585

[6] Ranger C, Raghuraman R, Penmetsa A, et al. Evaluating mapreduce for multi-core and multiprocessor systems[C]//High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on. IEEE, 2007: 13-24.

[7] Zaharia M, Konwinski A, Joseph A D, et al. Improving MapReduce Performance in Heterogeneous Environments[C]//OSDI. 2008, 8(4): 7.

[8] Ekanayake J, Li H, Zhang B, et al. Twister: a runtime for iterative mapreduce[C]//Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. ACM, 2010: 810-818.

[9]JimGray.Sortbenchmarkhomeage.http://research.microsoft.com/barc/SortBenchmark/.

[10] L. Huston, R. Sukthankar, R.Wickremesinghe, M. Satyanarayanan,G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond:A storage architecture for early discard in interactivesearch. In Proceedings of the 2004 USENIX Fileand Storage Technologies FAST Conference, April 2004.

[11] Jeffrey Dean, Sanjay Ghemawat, “Simplified Data Processing onLarge Cluster,” Proceedings of the 6th conference on Symposium onOperating Systems Design & Implementation, 2004.

[12] Abouzeid A, Bajda-Pawlikowski K, Abadi D, et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads[J]. Proceedings of the VLDB Endowment, 2009, 2(1): 922-933.

[13] He B, Fang W, Luo Q, et al. Mars: a MapReduce framework on graphics processors[C]//Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008: 260-269.

[14] JUNG 2.0, [http://jung.sourceforge.net](http://jung.sourceforge.net/).

[15] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[16] McKenna A, Hanna M, Banks E, et al. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data[J]. Genome research, 2010, 20(9): 1297-1303.

[17] X. Song, B. Li, and H. Yang. Improved ant colony algorithm and itsapplications in tsp. In Proceedings of the Sixth International Conferenceon Intelligent Systems Design and Applications - Volume 02, ISDA ’06,pages 1145–1148, Washington, DC, USA, 2006. IEEE ComputerSociety.

[18] Chu C T, Kim S K, Lin Y A, et al. Map-reduce for machine learning on multicore[C]//NIPS. 2006, 6: 281-288.

[19] Dean J, Ghemawat S. MapReduce: a flexible data processing tool[J]. Communications of the ACM, 2010, 53(1): 72-77.

[20] <http://www.google.com.hk/webhp?client=aff-360daohang>.

[21]张建勋,古志民,郑超. [云计算研究进展综述](http://d.wanfangdata.com.cn/Periodical_jsjyyyj201002007.aspx)[J].计算机应用研究,2010,(02):429-433.doi:10.3969/j.issn.1001-3695.2010.02.007.

[22]陈康,郑纬民. [云计算:系统实例与研究现状](http://d.wanfangdata.com.cn/Periodical_rjxb200905025.aspx)[J].软件学报,2009,(05):1337-1348.doi:10.3724/SP.J.1001.2009.03493.

[23]朱珠. 基于Hadoop 的海量数据处理模型研究和应用 [D][J]. 北京邮电大学, 2008, 1: 7-20.

[24]王珊, 王会举, 覃雄派, 等. 架构大数据: 挑战, 现状与展望[J]. 计算机学报, 2011, 34(10): 1741-1752.

[25]陈志刚, 李登. 分布式系统中一种动态负载均衡策略, 相关模型及算法研究[J]. 小型微型计算机系统, 2002, 23(12): 1434-1437.

[26]余楚礼, 肖迎元, 尹波. 一种基于 Hadoop 的并行关联规则算法[J]. 天津理工大学学报, 2011, 27(1): 25-28.

[27]林伟伟. 一种改进的 Hadoop 数据放置策略[J]. 华南理工大学学报: 自然科学版, 2012, 36(1): 152-158.

[28] Kolb L, Thor A, Rahm E. Load balancing for mapreduce-based entity resolution[C]//Data Engineering (ICDE), 2012 IEEE 28th International Conference on. IEEE, 2012: 618-629.

[29]Schatz M C. CloudBurst: highly sensitive read mapping with MapReduce[J]. Bioinformatics, 2009, 25(11): 1363-1369.

[30] JDean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[31]Eltabakh M Y, Tian Y, Özcan F, et al. CoHadoop: flexible data placement and its exploitation in Hadoop[J]. Proceedings of the VLDB Endowment, 2011, 4(9): 575-585.

[32]徐剑, 陈群等. 一种基于Hadoop的多表连接策略[J]. 现代电子技术, 2014年第8期.

# 在校期间发表论文

论文题目：《一种基于Hadoop的多表连接策略》

作者：徐剑，陈群，王卓，李战怀，潘巍

发表时间：2014.1.15

发表状态：已录用(2014年8期)

期刊名：现代电子技术

ISSN：1004-373X

CN：61-1224/TN

# 致谢

在本文完成后，我即将结束为期三年的研究生生涯，我衷心地感谢在我硕士期间，所有对我的学习、工作及生活给予帮助的人。

首先，感谢我的硕士生导师陈群教授，感谢他在我研究生期间对我的关心和帮助。生活中，陈老师像家长一样，常常与我进行沟通与交流，并给予我鼓励与安慰，使我倍感温暖；工作中，陈老师一丝不苟，从严治学，他严谨的工作风格给我留下了深刻的印象，是我以后科研及工作的榜样。我毕业论文的顺利完成，与陈老师的悉心指导是分不开的。再一次向陈老师致以真诚的敬意。

我还要感谢我的另一位导师，潘巍老师。在进行项目研究的过程中，潘老师丰富的经验以及扎实的理论功底，使我受益匪浅，收获颇丰。无论是研究点的设立、算法的设计，还是科研文章的撰写、做科研的态度，潘老师都会一一耐心地教导我，使我逐步走进科研的殿堂，同时，潘老师活跃的思想，谦逊的性格以及追求真理的态度是我以后科研道路上的榜样。

同时，我要感谢教研室的博士师兄王卓、索勃、樊峰峰、硕士师弟陈肇强、尤立、白松、师妹勾志营、以及与我共同科研、共同成长的硕士同学徐东海、杜晶、胡嘉琪、徐曜和王浩，是他们在我遇到技术难关时，及时给予帮助，在我困惑于一个科研问题时，牺牲时间与我讨论，再一次感谢他们。

最后，我要特别感谢我的父母，我的家人，他们是我学习、科研的坚强后盾，一直在为我默默地付出，无私地奉献。无论我遇到什么困难，什么挫折，他们总会站在我身边，给我最强有力的支持，他们的支持是我顺利完成学业的保证。谨以此文献给他们。

# 知识产权说明书

