分 类 号 _____ 学号 200915055

学校代码 10487 密级 _____

華中科技大學

毕业设计[论文]

题目：基于 mapreduce 的单路连接算法的比较和分析

院　　系：计算机科学与技术

专　　业：计算机科学与技术

姓　　名：_____

指导教师：_____

2013 年 6 月 9 日

摘 要

在数据分析的工作中，最常见的从两个数据集中组合和整理数据的操作是连接（Join）操作。尽管目前针对不同环境中的连接，已经有许多连接算法，但是，面对海量数据集，对连接处理提出了前所未有的挑战。随着 MapReduce 架构的应用，需要这个环境下支持复杂的连接操作。

对两个数据集的单路任意连接算法进行了深入的研究。首先，对现有的连接算法：重新划分(Repartition)和广播(Broadcast)连接算法进行了比较和分析。分析了其中所存在的问题，针对分布式计算的 I/O 优化，设计了针对分布式数据划分的新算法，即通过两数据集比例和 Reducer 数量的关系计算出比传统算法 I/O 开销优化的数据划分方案。其次，改写了 Hadoop 源代码 MapReduce 架构中 shuffle 阶段 Partition 模块的数据分配机制，将原来的一对一数据分配机制改写为支持一对多的数据分配机制，从而支持在 Hadoop 中实现新的数据划分算法；改写了 Hadoop 中 Reduce 阶段 Context 的文件读取方式，通过对输入流 NIO(java New IO 提供多路非阻塞式的高伸缩性网络 I/O)的实现，将原来的输入流顺序 I/O 一次性读取，扩展成可支持定位多次读取的模式，以此解决 Reduce 阶段的小表无法缓存问题；实现了新的 MapReduce 单路任意连接的算法，这种算法能更好地支持单路任意连接在分布式环境下的计算，通过计算数据的划分优化，对基于 MapReduce 的分布式连接计算解决方案进行优化和改进。

最后通过实验对改进前后的连接算法进行了比较和分析，通过实验证明所提连接算法的有效性。

关键词：Hadoop，MapReduce，分布式计算，数据仓库，单路连接，数据划分

Abstract

In the data analysis work, the most common combinations and the data processing operation from the two data sets is join operation. Despite the current connections for different environments, there have been many connections algorithm, but, faced with massive data sets, put forward unprecedented challenges for join process. With the application of MapReduce framework that require this environment to support complex connection operations.

On two data sets single any join algorithm conducted in-depth research. First, the existing connection algorithms: Repartition and Broadcast join algorithm are compared and analyzed. Wherein the analysis of the problems of distributed computing I/O optimization, and designed for distributed data partitioning a new algorithm, two data sets by relationship between the number of Reducer and proportional algorithms to calculate the I/O cost optimization data partition plan which is better than conventional. Secondly, rewriting the source code of Hadoop MapReduce architecture shuffle phase Partition module data distribution mechanism, the original one to one data distribution mechanisms rewritten to support one to many data distribution mechanisms to support the implementation of new data partitioning algorithm in Hadoop . rewritten in Hadoop Reduce phase Context file reads way through the realization of the input stream NIO, the original input stream sequential I/O read one time, can be extended to support positioned read many times model, in order solve Reduce phase small table can not cache problem. achieve a new single join MapReduce algorithm, which is support single join calculation better in a distributed environment. through calculation optimal partitioning of data optimized and improved distributed joins computing solutions which base on MapReduce .

The experiments compared and analyzed the algorithms on the join before and after improvement through experiments prove the effectiveness of the proposed join algorithm.

Keywords: MapReduce，Hadoop，Distributed Computing，Data Warehouse，Two Way Join，Data Partitioning

目 录

1 绪论

1.1 问题的提出

随着大数据和 Hadoop 的普及[9]，使用 MapReduce 对大数据进行数据挖掘已经成为了一种趋势，MapReduce 正在被用于不同的数据挖掘应用。单路连接是各种数据分析中的一种常见操作。但是，MapReduce 通常使用简单但是效率低下的算法来执行连接。

在对 MapReduce 架构进行研究的过程中发现了单路连接操作所存在的一些问题：传统的 Map 到 Reduce 的数据发送是一对一关系，所有数据在 Reduce 都是唯一的，因为单路非等值连接中两表的元组往往是多对多关系，这导致一般情况下两个表之间无法实现多个 Reduce 的单路非等值连接计算。将所有数据集中在一个 Reduce 中进行单路非等值连接计算不但要缓存数据而且由于完全没有发挥 MapReduce 的分布式计算优势，执行效率比较慢，因此有研究提出了拆分大表通过 Map 发送到多个 Reduce 中，每个 Reduce 直接下载缓存小表的方式来解决[12]，但这种方法在大小表数据比例梯度较小时会引起大量 I/O 开销导致分布式计算效率降低，在小表不可缓存的情况下任务无法进行，在因此本文希望对过去的算法其进行改进，达到在大小表数据梯度较小的环境下提高任务执行效率和在小表无法被缓存的环境下实现任务计算的目的。

1.2 国内外概况

标准的重新分配连接是可以在一个 MapReduce 任务中实现的，在这个连接策略的 shuffle 阶段中，L(左表大表)和 R(右表小表)以连接键进行了 merge(聚类)，并且在 Reduce 阶段中以 merge 划分中对应的数据集合进行分类并连接。但是它有一个潜在的问题就是，来自 L 和 R 的给定连接键的所有的记录都会被缓存。当键的基数很小，或者数据是高度倾斜的，给定键值的所有记录就不适合于放在内存中。

在大多数应用中，表 R 远远小于表 L，即|R|<<|L|。不是像基于重新分配连接那样，将 R 和 L 都通过网络移动，可以在 Map 阶段只划分大表数据，在 Reduce 阶段简单地广播(发送小表到每个 Reducer)较小的表 R，因为它避免了 shuffle 阶段在两个表上的排序，这种方式称为广播连接。但是它有一定的局限性，连接中的小表要能被内存缓存。

这几年国内也有一些研究致力于改进 MapReduce 架构中的连接操作。华东师范大学在文献[10]中对 MapReduce 框架下的数据分析算法进行了优化，优化的

目标集中在数据仓库中典型连接操作算法。他们提出的优化方案是通过 HdBmp 索引的高效构建以及合理布局，提升了连接的效率和可扩展度。

另外还有研究提出一种基于自适应分片的优化算法 AFR-AS[11]。借助自适应分片可以解耦 Map 任务数与数据集复制代价之间的高度耦合关系。利用自适应分片的动态构造算法，Map 任务可以在任务生命周期内处理多个普通分片，从而有效降低任务启动开销以及非对称分片复制连接中的数据广播开销，同时保证了基于普通分片的细粒度负载平衡和容错能力。

不论是国内还是国外，可能是因为 MapReduce 的原始架构对数据唯一性的局限，导致对 MapReduce 下的单路任意连接都止步于广播连接，这也就无法避免对小表的完全缓存问题和小表完全下载到每个 Reducer 的 I/O 开销问题。

1.3 主要研究工作

现有 MapReduce 的连接算法[2, 3]只考虑等值连接查询。没有一个能够很好地支持非等值连接查询。在认真学习文献的基础上，深入理解了单路连接算法的难点，比较分析了多种单路连接算法。在 MapReduce 的单路等值连接算法的基础上，本文提出了单路非等值连接的新算法。

本文的组织结构如下：

第 2 章的相关技术部分介绍了 MapReduce、Hadoop、连接算法等技术，分析了现有的单路连接方法，比较了其优缺点及其适用范围，给后续章节的展开提供了必要的理论基础。

第 3 章详细叙述了数据划分的新算法，对 Hadoop 源码的改写，新连接算法的提出，包括算法的流程，对 Hadoop 源代码的修改介绍等。

第 4 章对重分配和广播连接算法的性能进行了比较和分析，并且对新提出的算法实现进行了实验和对比。

第 5 章对全文的研究工作进行总结，并对一步的研究做出展望。

2 相关技术

2.1 概述

本章将回顾 MapReduce 分布式计算架构和 Hadoop 开源系统的架构，这是本文深入研究单路任意连接的基础。详细介绍基于 MapReduce 的重分配连接和广播连接的相关内容，具体分析他们在单路连接作用和局限，提出这两种算法在面对单路任意连接时会遇到的挑战。

2.2 MapReduce

MapReduce[4]是 Google 提出的一个软件架构，用于大规模数据集的并行运算。概念“Map”和“Reduce”，他们的主要思想都是从函数式编程语言借来的，还有从矢量编程语言借来的特性。

MapReduce 提供了一个新的用并行方法来处理数据的平台，尤其是大数据。许多 MapReduce 的方案[1, 2, 3, 6]的开发是为了处理数据仓库中大数据的连接查询。

当前的软件实现是指定一个 Map 函数，用来把一组键值对映射成一组新的键值对，指定并发的 Reduce 函数，用来保证所有映射的键值对中的每一个共享相同的键组。

简单说来，一个 Map 函数就是对一些独立元素组成的概念上的列表的每一个元素进行指定的操作。事实上，每个元素都是被独立操作的，而原始列表没有被更改，因为这里创建了一个新的列表来保存新的结果。这就是说，Map 操作是可以高度并行的，这对高性能要求的应用以及并行计算领域的需求非常有用。

而 Reduce 操作指的是对一个列表的元素进行适当的合并。虽然它不如 Map 函数那么并行，但是因为 Reduce 总是有一个简单的答案，大规模的运算相对独立，所以 Reduce 函数在高度并行环境下也很有用。

MapReduce 通过把对数据集的大规模操作分发给网络上的每个节点实现可靠性，每个节点会周期性地把完成的工作和状态的更新报告回来。如果一个节点保持沉默超过一个预设的时间间隔，主节点记录下这个节点状态为死亡，并把分配给这个节点的数据发到别的节点。每个操作使用命名文件的不可分割操作以确保不会发生并行线程间的冲突；当文件被改名的时候，系统可能会把他们复制到任务名以外的另一个名字上去。

Reduce 操作工作方式很类似，但是由于 Reduce 操作在并行方面能力较差，主节点会尽量把 Reduce 操作调度在一个节点上，或者离需要操作的数据尽可能近的节点上了。

关于 MapReduce 的直观流程见下图 2.1。

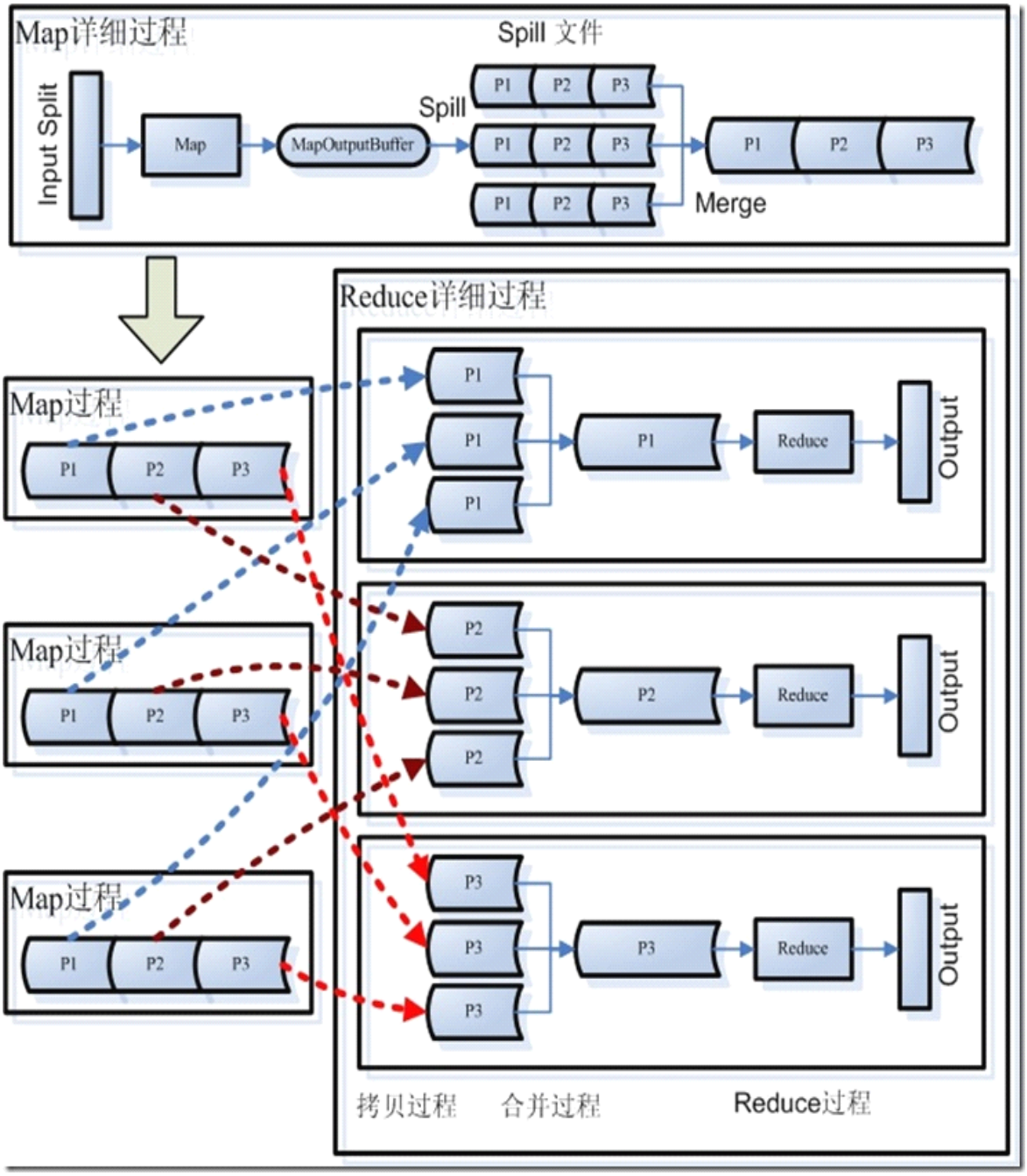

图 2.1 MapReduce 流程图

最初输入的数据按照 split(一种对 Map 输入文件按大小默认 64M 分割的划分)规定大小划分到各个 Map 中，发送到 Map 中的数据根据 spill(split 中划分成更小的数据碎片)的规定大小划分后按批次处理，每个 spill 中的数据在处理后内部划分到对应的 Partition(对应上图的 P1，P2，P3)中，所有 spill 都内部划分完毕后，

对应相同 Partition 的数据相互合并，合并后的数据就是 Map 输出数据。Reduce 下载所有 Map 端对应编号的 Partition 数据，在本地合并，开始执行 Reduce 任务，最后输出 Reduce 结果。

2.3 Hadoop

一个分布式系统基础架构，由 Apache 基金会开发。用户可以在不了解分布式底层细节的情况下，开发分布式程序。充分利用集群的高速运算和存储。Hadoop[7,8,17-20]以 google 的 GFS[5]为原型实现了一个分布式文件系统（Hadoop Distributed File System），简称 HDFS。HDFS 有着高容错性的特点，设计用来部署在低廉的硬件上。而且它提供高传输率来访问应用程序的数据，适合那些有着超大数据集的应用程序。HDFS 放宽了 POSIX 的要求，这样可以流的形式访问文件系统中的数据。

Hadoop 是一个能够对大量数据进行分布式处理的软件框架。但是 Hadoop 是以一种可靠、高效、可伸缩的方式进行处理的。Hadoop 是可靠的，因为它假设计算元素和存储会失败，因此它维护多个工作数据副本，确保能够针对失败的节点重新分布处理。Hadoop 是高效的，因为它以并行的方式工作，通过并行处理加快处理速度。Hadoop 还是可伸缩的，能够处理 PB 级数据。此外，Hadoop 依赖于社区服务器，因此它的成本比较低，任何人都可以使用。

Hadoop 具体任务流程见下图 2.2。

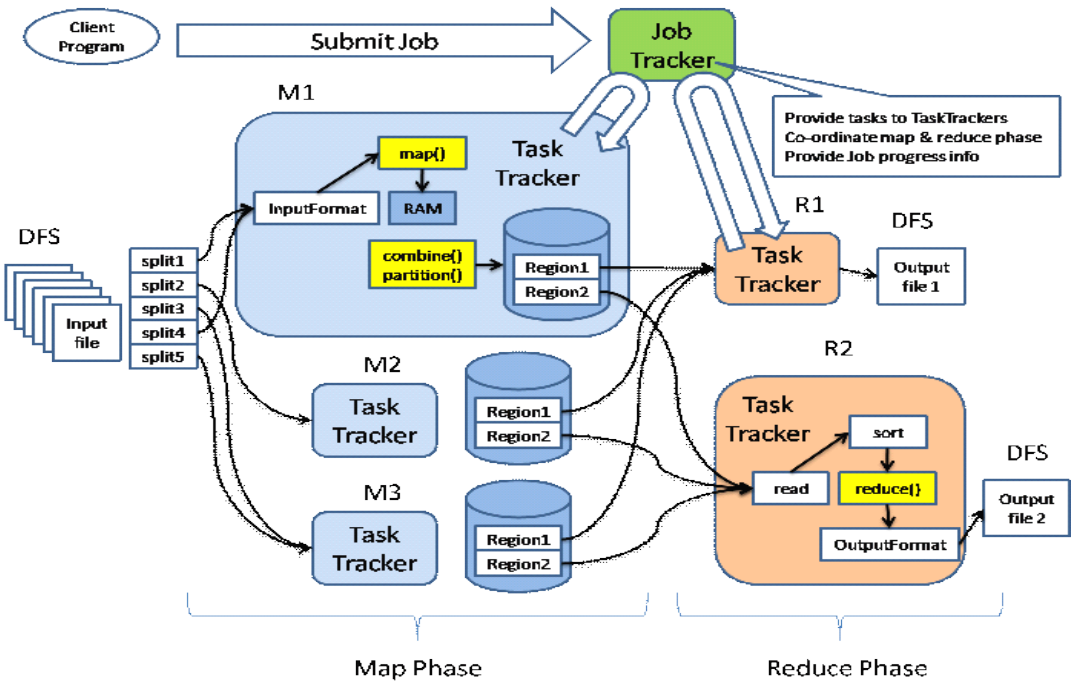

图 2.2 Hadoop 流程图

本地任务配置完成后提交给 job tracker，job tracker 规划 Map 和 Reduce 任务等待 task tracker 通过心跳通讯来领取合适任务进行计算，其中 Map 任务会访问 DFS 读取输入数据，进行 Map 数据处理，生成 Map 输出数据等待相应 Reduce 下载数据。Reduce 从 Map 下载全部数据后开始 Reduce 任务计算，最后输出计算结果到 DFS。

Hadoop 是一个能够让用户轻松架构和使用的分布式计算平台。用户可以轻松地在 Hadoop 上开发和运行处理海量数据的应用程序。它主要有以下几个优点：

(1) 高可靠性。Hadoop 按位存储和处理数据的能力值得人们信赖。

(2) 高扩展性。Hadoop 是在可用的计算机集簇间分配数据并完成计算任务的，这些集簇可以方便地扩展到数以千计的节点中。

(3) 高效性。Hadoop 能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。

(4) 高容错性。Hadoop 能够自动保存数据的多个副本，并且能够自动将失败的任务重新分配。

Hadoop 带有用 Java 语言编写的框架，因此运行在 Linux 生产平台上是非常理想的。Hadoop 上的应用程序也可以通过 stream 使用其他语言编写，比如 C++。

2.4 连接算法

2.4.1 重新分配连接算法

重新分配连接(Repartition Join)是 MapReduce 架构中最常用的连接策略。在这个连接策略中，L 和 R 在连接键上进行了动态划分，并且以划分后对应的元组进行连接。

这个连接策略类似于并行关系数据库管理系统(RDBMS)中的分区 sort-merge 连接。它也是 Hadoop 属性连接包中提供的连接算法。标准的重新分配连接可以在一个 MapReduce 任务中实现。在 Map 阶段，每个 Map 任务在 R 或 L 的一个 split 上工作。为了确认一个输入记录是来自于哪个表，每个 Map 任务标记记录的来源表，并且输出提取的连接键以及标记记录作为一个键值对。然后对输出进行划分，排序以及合并。每个连接键的所有记录被分组在一起，并且最终供应给 Reducer。对于每一个连接键，Reduce 函数首先根据表标签将输入记录分离并且缓存到两个集合，然后在这些集合之间执行一个笛卡尔积。标准的重新分配连接中有一个潜在的问题就是，来自 L 和 R 的给定连接键的所有的记录都会被缓存。当键的基数很小，或者数据是高度倾斜时，给定键值的所有记录就不适合于放在内存中。

为了解决标准重新分配连接中的 buffer(缓存给定连接键的所有记录)问题，

有研究[12]做了一些关键的改变，引入了改进的重新分配连接。首先，在 Map 函数中，输出键改成了由连接键和表标签构成。表标签要保证在给定连接键上 R 的记录会排序在 L 记录的前面。其次，定制划分函数使得哈希码只根据组合键的连接键部分来计算。这样，有相同连接键的记录仍然会被分配到相同的 Reduce 任务。Reducer 中的分组函数也进行定制，使得记录只在连接键上进行分组。最后，由于对于给定的连接键，来自较小表 R 的记录被确保在 L 的前面，所以只有 R 的记录被缓存，而 L 记录进行流动产生连接输出。改进的重新分配连接解决了标准版本中的 buffer 问题。但是这两个版本包含了两个主要的会损失性能的开销源。特别是，L 和 R 都需要排序，并且在 MapReduce 的 shuffle 阶段需要在网络上传输。

如果在连接操作之前，L 和 R 都已经在连接键上被划分好了，那么在重新分配连接中的 shuffle 开销就可以减少。这可以通过 log 记录产生时在连接键上预划分（pre-Partitioning）L 以及 R 载入 DFS 时在连接键上预划分 R 来实现。然后，在查询时间上，来自 L 和 R 的匹配划分就可以直接被连接了。

注意，Hadoop 中连接包中提供的 Map 端的连接类似于直接连接算法。但是，Map 端的连接要求输入表的每个划分中的所有记录按照连接键严格地进行排序。另外，这个算法将所有输入表中具有相同连接键的所有记录都缓存到内存中。至于倾斜的表 L，Map 端的连接会很容易地用完内存。相比较而言，直接连接算法只将较小表 R 中的划分进行缓存，因此可以处理 L 中的数据倾斜。

基于等值连接的重新分配连接算法的主要思想是在 shuffle 阶段对数据按连接键进行排序聚类，在 Reduce 阶段将具有相同连接键的 value 集合按表号分类，然后遍历做笛卡尔积筛选，由于相同连接键的集合容量一般比较小，所以此方法在处理等值连接操作时具有很快的速度，Hadoop 中自带的实现脚本为 datajoin。

基于非等值连接的重新分配连接算法由于存在连接键内容无关性，是不能按照连接键进行聚类的。原始的 Hadoop 架构也不支持 Map 输出数据多个 Reduce 接收功能，所以只能在单节点对两表所有数据进行笛卡尔积遍历筛选。显然，这么做是存在很多问题的，首先为了结果的完整性，数据无法进行分布式计算，单点性能低下，其执行速度必然会很慢，其次如果两表数据较大，而此方案对非等值连接又要缓存两表，就有可能产生内存不足(OOM)问题。

在 MapReduce 连接算法中，重新分配连接算法是基于 Reduce 的连接中研究最多的内容之一。本节主要围绕重新分配连接算法进行归纳总结。具体的基于单路等值连接的重分配连接算法描述见算法 1。

算法 1：标准重分配等值连接

Map(key:null,value:来自表 L 或表 R 一个 split 数据块中的一个元组){

从 value 中提取连接键

```
        给 value 打上表 L 或表 R 的表号的标签
        Emit<连接键,打上标签的 value>//向连接键哈希后对应的 Reduce 发送
    }
    Reduce(key:连接键，valuelist:来自表 L 或表 R 的连接键为 key 的所有元组){
        用缓存创建表 L 和表 R 的临时表
        For(value in valuelist){
            将 value 根据标签加入相应的临时表
        }
        For(value1 in  表 L 临时表){
            For(value2 in  表 R 临时表){
                若 value1 和 value2 满足连接条件 Emit<null,连接后的元组(value1,value2)>
            }
        }
    }
```

2.4.2 广播连接算法

在大多数环境中，表 R 远远小于表 L，即|R|<<|L|。不是像基于重新分配连接那样，将 R 和 L 都通过网络移动，可以简单地广播较小的表 R，因为它避免了在两个表上的排序，更重要的是避免了移动较大表 L 所产生的网络开销，这种连接方式被称为广播连接(Broadcast Join)。

广播连接作为一个 Map-only(只有 Map 阶段)任务来运行。在每个节点，所有的 R 都从 DFS 中取出来避免在网络上发送 L。每个 Map 任务使用一个主存哈希表将 L 的一个 split 和 R 连接。

在每个 Map 任务的预处理函数中，广播连接检查 R 是否已经存储在本地文件系统中。如果没有，它就会从 DFS 中取出 R，根据连接键划分 R，并且存储这些划分到本地文件系统。

广播连接动态地检查是否在 L 或 R 上建立哈希表。在 R 和 L 的 split 中选择更小的来建立哈希表，假设该哈希表总是能够适合放在内存中。这是一个安全的假设，因为一个典型的 split 会小于 100MB。如果 R 比 L 的 split 更小，初始化函数用于载入所有的 R 划分到内存中来建立哈希表。然后，Map 函数从 L 中每条记录上抽取连接键，并且使用它来探测哈希表，并且产生连接输出。另一方面，如果 L 的 split 比 R 更小，连接就不在 Map 函数中执行了，而是，Map 函数按照和划分 R 一样的方式来划分 L。然后，在 Map 后的结束阶段，R 和 L 对应的划分被连接。如果 L 对应的划分没有记录，就可以避免载入 R 中的那些划分。当连接键的域非常大的时候，这种优化是非常有用的。

注意，在 Map 任务中，R 的划分可以被多次重载，因为每个 Map 任务作为一个独立的进程运行。尽管没有方法来精确控制 DFS 中副本的物理位置，但是通过为 R 增加副本因子，可以保证簇中大多数的节点都有一个 R 的本地副本。这样可以保证广播连接避免从 DFS 取出 R 到它的初始化函数中。

在 MapReduce 连接算法领域中，广播连接算法是基于 Map 的连接中研究最多的内容之一。目前，广播连接算法已存在较多的改进方案。本节主要围绕候广播连接算法进行归纳总结。

(1) 单路广播等值连接算法

单路广播等值连接算法的原理是在 Map 阶段分割大表后分布式处理大表的部分内容，而小表由缓存的形式载入内存并生成 hash 表，由此来提高等值连接的查询速率，此算法具有可分布性和速度快等优点，但也存在缓存表受到内存大小限制的局限性，在两表梯度较大时具有明显的速度优势，但随着缓存表的增大，其劣势的一面会越来越明显。具体的基于单路等值连接的广播连接算法描述见算法 2。

算法 2：广播等值连接

预处理阶段：从分布式文件系统下载表 R，若是小表将元组中的连接键为 hash 值缓存到结构为<连接键，valuelist>的小表临时 hash 表

```
Map(key:null,value1:来自表 L 一个 split 数据块中的一个元组){
    If(表 R 为小表并生成了 hash 表){
        根据 value1 中提取的连接键，从 hash 表中取出相同连接键的元组
        For(value2 in hash 表中每个有相同连接键的元组){
            Emit<null,连接后的元组(value1,value2)>
        }
    }else{
        将 value1 根据提取出的连接键加入小表 hash 表
    }
}
结束阶段：
If(表 R 不是小表){
    For(value1 in  表 R){
        For(value2 in hash 表中每个有相同连接键的元组){
            Emit<null,连接后的元组(value1,value2)>
        }
    }
}
```

(2) 单路广播非等值连接算法

广播非等值连接算法的分布式原理与等值连接基本相同，不同的是载入内存后由于两表间要做笛卡尔积遍历操作，没有生成 hash 表的必要，分布式的优势依然存在，缓存局限性也同等值连接一样。在两表梯度较大时具有明显的速度优势，但随着缓存表的增大，其劣势的一面会越来越明显从而导致速度变得很慢。具体的基于单路非等值连接的广播连接算法描述见算法 3。

算法 3：广播非等值连接

预处理阶段：从分布式文件系统下载表 R，若为小表缓存到小表临时表

```
Map(key:null,value1:来自表 L 一个 split 数据块中的一个元组){
    If(表 R 是小表){
        For(元组 value1 in 大表){
            For(元组 value2 in 小表临时表){
                //非等值连接连接键之间需要判断连接键是否满足对应关系
                若 value1 和 value2 满足连接条件 Emit<null,连接后的元组(value1,value2)>//因为广播连接只有 Map 阶段，此处 Emit 为最后的结果输出。
            }
        }
    }else{
        缓存 value1 到小表临时表
    }
}
```

结束阶段：如果表 R 不是小表

```
For(value1 in 表 R){
    For(value2 in 小表临时表){
        若 value1 和 value2 满足连接条件 Emit<null,连接后的元组(value1,value2)>
    }
}
```

2.5 小结

本章详细介绍了 MapReduce 架构以及其开源实现 Hadoop，便于后面基于 Hadoop 的二次研发。此外，还介绍了几种基于 MapReduce 的连接算法，并对他们在单路等值和非等值连接的使用场景中存在的问题做了详细的描述，这些算法对单路非等值连接的支持存在一定的局限，但他们是本文提出算法的基础，将会在后面的章节对这些算法进行比较和分析。

3 单路非等值连接算法的改进

3.1 新算法的思想

经过上述对各种连接算法的研究和探讨，单路等值连接与预分配算法结合的很好，很难有质的改进，本文研究主要针对改进非等值连接。

要通过 MapReduce 架构来得到分布式计算的性能提升，又要避免完全缓存小表引起的一系列问题，就必须放弃重分配和广播的部分思想，突破 MapReduce 数据发送一对一的限制。

在研究中发现，某些环境下不同的数据划分导致的数据 I/O 开销是不同的，以例 3.1 为例，这方面的 I/O 开销是可以得到优化的，所以新算法的思想就是减少数据发送上的 I/O 开销。

例 3.1：大表和小表都为 1GB 的数据，Mapper 或 Reducer 设置 16 个。

这样的情况下，广播连接的 I/O 开销是大表的 1GB 开销加上小表的 1GB*16 的开销，一共 17GB 的开销，而通过 4*4(两表各划分成 4 份)的数据划分，计算量不变，可以只产生大表 1GB*4 的开销加上小表 1GB*4 的开销，一共 8GB 开销，优化数据划分的开销降低是显而易见的。

以此给出一种新的单路非等值连接算法思路，基本思想为利用划分算法对大小表进行二维分割，动态地对两表数据进行划分，以此优化分布式数据发送上的 I/O 开销，Map 阶段将数据一对多向 bucket(划分桶)分发（Partition），Reduce 阶段收集分配到各个 Reducer 的数据以笛卡尔积遍历进行非等值连接操作。

广播连接将小表分发到每个 Map 节点的思想，其实是本文算法思想在大小表梯度大时(小表因为相对大表太小而不被划分)的边界情况，但在大小表梯度较小的情况下，广播连接在 I/O 开销上的劣势(整个小表的反复发送)就凸显出来，本文提出的算法思想的优势(优化 I/O 开销)就能比较明显的体现。

由于每个表的数据都要发送另一个表划分数量个 bucket，单路连接的理论 I/O 开销公式为(3.1)：

$$\text{I/O 开销} = A * y + B * x \tag{3.1}$$

其中 A 代表大表数据量，y 代表小表划分数，B 代表小表数据量，x 代表大表划分数。可以通过基本不等式(3.2)求解 I/O 开销的最小值：

$$A * y + B * x \geq 2\sqrt{ABxy} \tag{3.2}$$

此不等值取最小值的条件为(3.3)：

$$A * y = B * x \tag{3.3}$$

同时有最大划分数限制条件(3.4)：

$$x * y = C \tag{3.4}$$

其中 C 为预先设定的总划分个数。

理论上当同时满足(3.3)(3.4)条件时，就可以在一定的划分数量条件下得到较少的 I/O 开销。

3.2 数据划分新算法

新提出的非等值连接算法汲取了过去各种经典算法的种种思想和经验，在引入数据划分思想后使得非等值连接操作对 MapReduce 的分布式架构支持更好，总体 I/O 开销变得更小，并接近理想 I/O 最小值。对于分割算法，过去的资料中有提出使用几何平均值作为单个划分(桶)的边长大小，而本文提出的新算法是以 Reducer 个数的因子对计算出 I/O 分发最小的划分比例，至于两种算法的异同，将在本节详细阐述。

3.2.1 开方划分算法

对(3.3)变换形式，可以得到：

$$\frac{A}{x} = \frac{B}{y} = \text{单个划分边长} \tag{3.5}$$

这表示大小表划分后的单个数据块中的两表划分的数据量是相等的。所以开方划分算法认为理想状态下每个 Reducer 应该是长宽相等的正方形，结合(3.4)(3.5)先计算出这个正方形的边长，然后通过大小表的条数除以边长来决定大小表的划分。

可推导单个划分边长计算公式(3.6)如下：

$$\text{边长} = \sqrt{\frac{\text{大表条数} * \text{小表条数}}{\text{reducer个数}}} \tag{3.6}$$

例 3.2：大表条数为 6000 条，小表条数为 1500 条，Reducer 个数为 16 个。

边长 = 750，所以大表划分为 8，小表划分为 2。

当然这是个比较理想状态的情况，实际情况中如例 3.3，开方划分方案往往取不到整数划分，这个时候就需要向上或者向下取整，取整后的一组划分可能改变 C 的值，这个时候 C 的值就不再是原来的设置数（如测试中开始的设置数为 16，而实际任务执行时规划了 18 个 Reducer）。

例 3.3：大表条数为 4800 条，小表条数为 1875 条，Reducer 个数为 16 个。

边长 = 750，这时计算结果出现小数，大表划分为 6.4，小表划分为 2.5。

若采取划分结果向上取整 C 的值将被重设为 21，由此可能引发水桶效应，使计算效率下降，若采取划分结果向下取整 C 的值将被重设为 12，这可能导致硬件资源利用不充分，分布式计算性能下降。

开方划分的特点是对 x 和 y 的求解相对独立，x 和 y 的解的相互约束关系较弱，当 x 和 y 产生误差时，无法修正两值之间的关系，导致最终结果并无法满足预先提出的条件。

3.2.2 比例划分算法

比例划分为了强调 x 和 y 之间的关联性，对 x 和 y 的解范围做出了限制，使 x 和 y 的解必须落在 C 的因子对中。这样做可以使 x 和 y 在误差修正后仍然满足(3.4)的关系。

首先，每个任务事先设定好 C 的值（一般根据可用 cpu core 的最大数来设置，保证 cpu 资源最大化地利用，测试脚本中设置为 16），然后根据大小表数据条数的比例（测试数据中大小表数据条数比例为 4:1），计算出最接近这个比例的 C 值的二维因子对（比如 16 的因子对有[16,1],[8,2],[4,4]），在这里比较显然，[8,2]是最接近测试数据的比例的，所以对数据的划分如图 3.1，整个矩阵代表所有的连接可能解，竖轴代表大表，横轴代表小表，矩形内数字代表划分后的 Reducer 号码，对每个表的数据按照所得划分均分后发送到相应行或列的 Reducer（比如大表均分为 8 份，小表均分 2 份，大表的第一份数据按照图中竖向虚线箭头发送到 1,2 号 Reducer，小表的第一份数据按照图中横向虚线箭头发送到奇数号 Reducer）

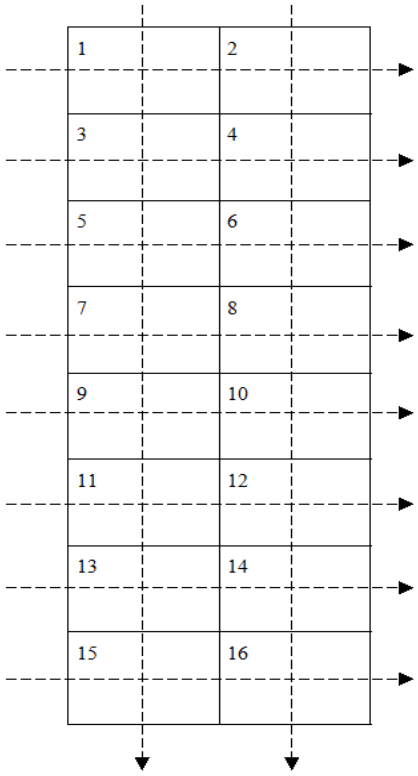

图 3.1 Reducer 划分图

上面的情况是比例比较接近的情况，如果比例不显而易见的时候，可以通过计算与数据比例邻近的两组因子对，具体公式(3.7)，通过因子对产生的 I/O 开销，来选择开销更小的因子对来作为优化的划分，具体优化划分公式见(3.8)。通过例 3.4 和例 3.5 具体说明两种情况的选择计算过程。

比例邻近因子对 = 邻近大小表比例的 Reducer 个数的二维因子对(如果不是恰好存在相等和边界情况，一般有对应的上邻近和下邻近两组) (3.7)

优化划分 = min{比例邻近因子对 I/O 开销}中的因子对 (3.8)

例 3.4：大表条数为 6000 条，小表条数为 1500 条，Reducer 个数为 16 个。

测试数据大小表比例是 4:1，可得：

比例邻近因子对 = 8 : 2

优化划分 = [8,2]

I/O 开销 = 6000 * 2 + 1500 * 8 = 24000

例 3.5：大表条数为 8100 条，小表条数为 1500 条，Reducer 个数为 16 个。

测试数据大小表比例是 27:5，可得：

比例邻近因子对 = {[8,2]和[16,1]}

[8,2]的 I/O 开销 = 8100 * 2 + 1500 * 8 = 28200

[16,1]的 I/O 开销 = 8100 * 1 + 1500 * 16 = 32100

优化划分 = min{28200,32100}的因子对 = [8,2]

3.2.3 两种划分算法的区别

由于新提出的两种划分算法是基于相同的目的(减少分布式数据发送的 I/O 开销)，所以在数据梯度很小时性状相似。

而在数据梯度较大时比例划分因为对 x 和 y 的解有因子对范围限定比较稳定，而开方划分在不做边缘检测的情况下会因为 Reduce 个数的剧烈增加使得效率变低。

在表数据比例不理想的情况下，比例划分不会改动实现配置好的 Reducer 数量，而开方划分的 Reducer 个数，如果解向上取整一般都会比预先设置的大，而如果解向下取整则一般都会比预先设置的小并且要进行边缘检测防止取 0 情况的发生，Reducer 可能导致效率变低。

所以可以把开方划分看成比例划分在数据梯度较小情况下的一种算法子集，虽然取几何平均数是数学上理想的结果，但是在实际中没有小数个 Reducer 或节点，所以 x 和 y 取整后可能无法再满足原先提出的条件，误差其实是很大的，I/O 开销并不能达到理论值，开方划分通过对小数取整，即使可能使得 I/O 更理想化，但是会改动原有的与环境相吻合的 Reducer 个数设置，从而引发一些问题，这些问题将在下一章中结合测试案例具体分析。

3.3 连接实现方法流程

这里必须提出的是，运用原始 Hadoop 架构是不足以实现 Map 和 Reduce 一对多数据分发的，重写 Hadoop 的 Partition 模块，达到了一份 Map 数据发送到多个 Reducer 的目的。

如图 3.2 所示，上图为重构的模块 Partition 在 shuffle 阶段的内部结构图，对应 MapReduce 底层流程中的具体位置见图 2.1 的 spill 阶段，原 Hadoop 的 shuffle 阶段中，每个 key 只能划分到一个 Partition，如 P1，P2，P3，在重构后实现了每个 key 可划分到任意个数的指定 Partition 中。

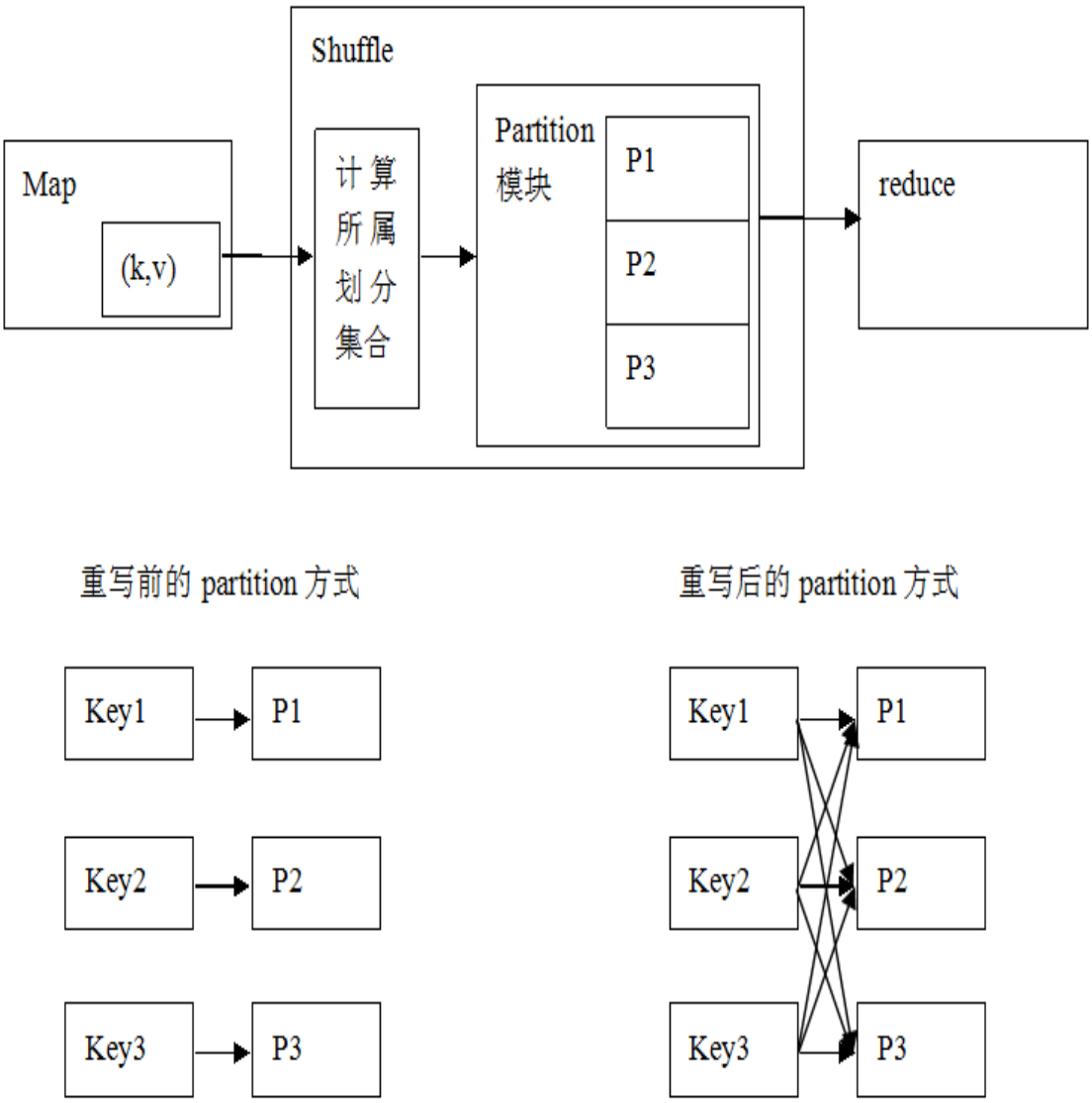

图 3.2 Partition 改写图

图 3.3 是改写后的算法流程图，介绍了改写后 Partition 模块的具体算法执行流程。

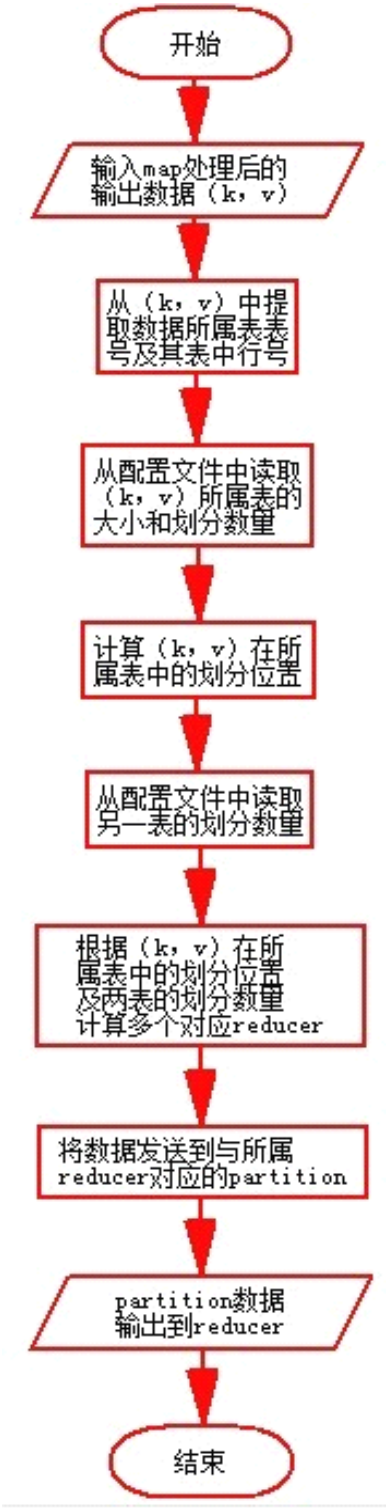

图 3.3 Partition 算法流程图

Reduce 阶段除了传统的缓存小表解决方案，在某些情况下出现小表过大无法缓存，对于这种情况，可以采取将小表存储为本地临时文件来解决，但考虑到一般无法缓存的表都比较大，转存为临时文件会产生不少 I/O 写入损失，所以通过对 ReduceContext(Reduce 阶段的上下文环境，相关处理文件也在其中，是 Hadoop 在 Reduce 中的一个重要类)的深拷贝保存 Context 状态和对 inputstream 的 NIO 读取方式的实现，完成对 Reduce 阶段的内部文件读取功能的改写，实现了 Reduce 相关处理文件的重定位反复读取（Hadoop 源码只实现了一次性的顺序 IO 读取），如此通过定位表头和对各表反复遍历的可行性实现单路非等值连接，虽然文件读取速度必然没有缓存读取快，但这个方案解决了单路非等值连接在一些维表过大的情况下无法缓存的问题。

如图 3.4 所示：T 表示表 table，文件中数据的排序方式按表号划分，原 Hadoop 在 Reduce 阶段中对文件的处理只支持一次性的顺序读操作，无法回头对数据二次读取，这种读取方式不对数据缓存无法对进行需要对表多次遍历的连接操作，通过修改 Reduce 模块中的 Context，实现文件位置 seek(定位)，从而实现对各个表表头位置的定位和存储，通过反复调用表头位置，实现表内容的反复读取，以此完成两表的数据连接。

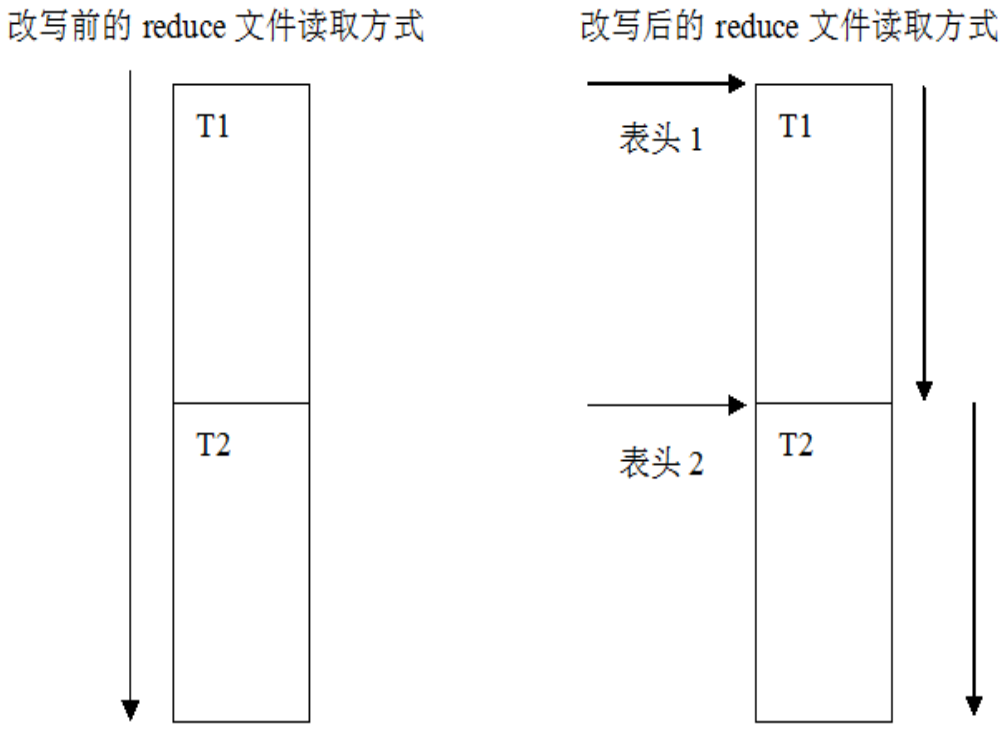

图 3.4 Context 改写图

图 3.5 是改写后的 Reduce 算法流程图，介绍了改写后 Reduce 阶段的具体算法执行流程。

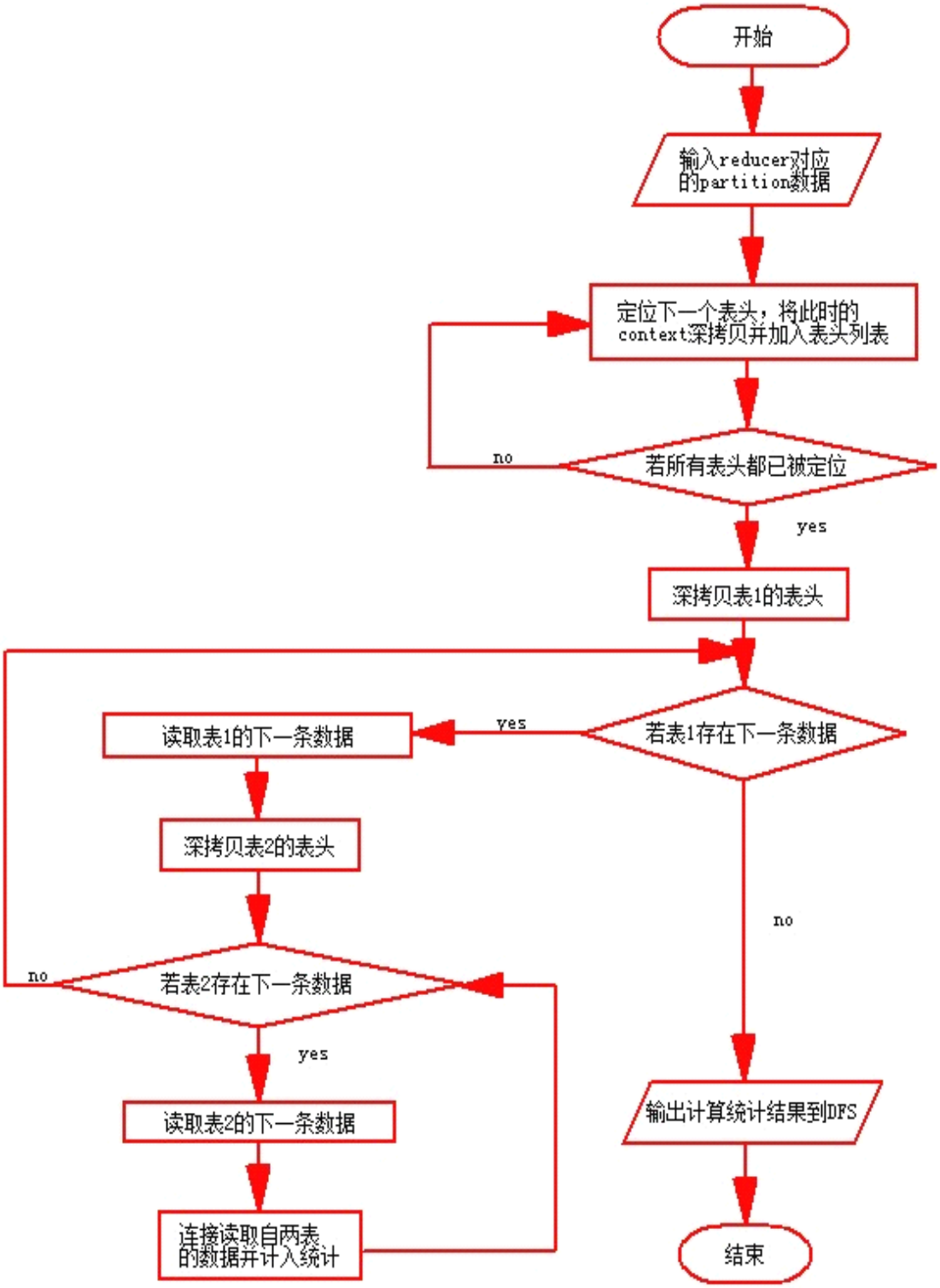

图 3.5 改写后 Reduce 阶段算法流程图

算法 4：基于比例划分的非等值连接算法

```
Map(key:null,value:来自表 L 或表 R 一个 split 数据块中的一个元组)
{
    获取 value 的表号和所在表中的位置
    根据 value 在所在表中的位置计算出所属划分(划分算法见上一小节)
    Emit<表号，value>到相应的 Reducer
    //Hadoop 源码改写后可以将一份数据 Emit 到多个 Reduce
}
Reduce(key:表号,valuelist:来自表 L 或表 R 一个划分的数据集合)
{
    先定位表 L 和表 R 的表头
    For(value1 in key 为表 L 的 values)
    {
    //改写后的 Hadoop 代码，可以不用缓存，通过保存表 R 和表 L 的表头对表 R 和表 L 中的所有元组进行多次遍历。
        For(value2 in key 为表 R 的 values)
        {
            //非等值连接连接键需要是否满足条件
            若 value1 和 value2 满足连接条件 Emit<null,连接后的元组(value1,value2)>
        }
    }
}
```

3.4 小结

本章首先对现存的 MapReduce 连接算法作存在的问题分析原因，简单分析案例后，对基于 I/O 开销的优化提出了针对数据划分的算法，对缓存的局限性提出了解决方案，结合之前已研究的连接算法提出新的非等值连接算法。这种算法，支持更多环境的非等值连接工作，提高了基于 MapReduce 的非等值连接的执行效率。

比例划分相比较于开方划分，环境参数 C(reducer 数量)更稳定可控，可以对硬件资源进行合理的预算，资源配置不会在任务执行中被修改，在硬件需要最大程度利用和任务需要最高效率执行的情况下对两者的权衡更好。开方划分在硬件资源足够充分的情况下可以得到很高的任务执行效率，可能会通过改变环境参数 C 来得到更优的数据划分，使 I/O 开销更小，但在硬件资源有限的环境下，开方

划分的表现没有比例划分稳定。

通过对 Hadoop 源码部分模块功能的改写支持新算法的实现，修改后新引入的数据操作机制出了对本文的作用，对今后进一步的研究也可能会有一些推进作用，见第 5 章。

4 测试结果比较与分析

本章将着重讲述在Hadoop集群环境下对各种基于MapReduce连接算法和本文提出的连接算法的测试结果的比较与分析。

首先，将针对参与比较分析的连接算法列出具体的实验测试结果数据表格。然后，将针对上面得到的实验结果数据来比较各种连接算法和本文提出的连接算法在实际环境和实际问题的处理上的速率差别。比较范围分别为基于单路等值连接的不同算法实现测试结果的比较和基于非等值连接的不同算法实现测试结果的比较。最后，将针对实验结果数据的比较来分析并且探究其两种等值连接算法背后形成这种实验结果的原因。分析范围分类为基于单路等值连接的不同算法实现测试结果的分析和基于非等值连接的不同算法实现测试结果的分析。

4.1 测试环境

对本文所提出的方法的性能进行试验评估。实验环境包括四个连接千兆位以太网交换机的节点。其中两个节点都配备两个 2.13GHz 的四核 CPU（另外两个节点配备一个 2.0GHz 的四核 CPU），8GB 内存和 1TB 的硬盘，并在 CentOS5 操作系统下运行 ApacheHadoop1.0.4。其中，主节点默认最多可以调用其他 3 个从节点的 20 个 Mappers 和 20 个 Reducers。

在基于单路连接的测试设计中，选择 1MB、2MB、10MB、20MB、100MB、1GB、10GB、100GB 规格的 TCP-H 生成数据作为测试的数据组[15-17]。

为了避免多余条件对数据连接数据产生过滤作用（数据过滤并非本论文研究的重点）从而影响测试数据量，并未采用 TPC-H 的标准测试 SQL 语句，对于两表之间的等值连接操作简单的采用了如下语句：

```
select count(*) from lineitem l,orders o where l.orderkey = o.orderkey;
```

对于两表之间的非等值连接操作简单的采用了如下语句：

```
select count(*) from lineitem l,orders o where l.orderkey > o.orderkey;
```

其中实验测试的算法脚本包括，等值重分配连接、等值广播连接、非等值重分配连接、非等值广播连接和本文提出的非等值连接新算法脚本，比例划分解决方案、几何平均划分解决方案。

其中本文提出的算法中设置默认可划分 Reducer 个数为 16 个。

4.2 单路等值连接

4.2.1 单路等值连接的测试结果

部分数据由于测试时间过长的原因没有测出，但这并不影响对重分配和广播

连接算法进行比较和分析。

广播连接测试脚本是采用了自己编写的 Mapjoin 脚本，重分配连接采用的 Hadoop 自带的 datajoin 脚本。

列出的数据有两表参与连接的行数（字节数数量级），输出数统计（由于部分测试可能导致输出数据非常大，所以统一输出 count 后的结果），各个脚本执行后得到的执行总时间结果。具体见下表 4.1。

表 4.1 等值连接测试结果表

Lineitem(行)	orders(行)	统计结果输出(行)	广播连接的等值连接测试结果(秒)	重分配连接的等值连接测试结果(秒)
6000(1MB)	1500	6005	20	64
12000(2MB)	3000	11957	21	65
60000(10MB)	15000	60175	26	64
120000(20MB)	30000	120515	26	63
600000(100MB)	150000	688572	35	71
6000000(1GB)	1500000	6001215		86
60000000(10GB)	15000000	59986052		152
600000000(100GB)	150000000	600037902		1536

4.2.2 单路等值连接的算法比较

在基于单路等值连接的算法测试中，实验的测试算法有：基于单路等值连接的重分配连接算法和基于单路等值连接的广播连接算法。

由表 5.1 的测试结果，在连接表的数据量达到 100MB 这个数量级之前广播连接显得非常快，但当连接表的数量级达到 1GB 的时候测试结果由于内存不足(OOM)而无法被得到。与此同时重分配连接虽然在处理 1GB 以下数量级的连接表数据时任务运行处理时间相对广播连接要慢一倍左右，但重分配连接可以支持更大数量级的数据连接，100GB 数量级的连接表数据处理时间在 1500 秒左右，这是广播连接无法做到的。

4.2.3 单路等值连接的算法分析

从上节的基于单路等值连接的算法比较中，在数据较小的环境下广播连接在处理单路等值连接问题中要比重分配连接算法强得多，而在大数量级的数据连接中，广播连接甚至不能完成测试，形成这种结果的原因如下，重分配连接在处理单路等值连接问题的时候由于使用两表连接键作为 Map 输出 key，根据这个 key 对输出数据进行排序和聚合后，重分配连接算法在处理单路等值连接问题的时候

时间和空间复杂度都很小。而广播连接由于对小表进行了 hash 缓存，所以在处理连接操作时速度更快，但随着数据量的增大，对内存的要求越来越高，当数量级超过一定程度程序就会内存不足这使得广播连接对缓存表过大的情况无能为力。

4.3 单路非等值连接

4.3.1 单路非等值连接的测试结果

由于非等值连接测试时间普遍较长，只选择 1M、2M、10M、20M 规格的 TCP-H 生成数据作为测试的数据组。虽然输入数据看起来比较小，但从连接的结果(见表 4.2 中的统计结果输出)来看数据量其实很大。并且由于 TPCH 的查询语句往往具有筛选功能，会把不参与连接的数据筛选掉，这里只列出实际参与连接的数据量，这不影响对各种连接算法进行比较和分析。

广播连接测试脚本是采用了 hive 自带的 Mapjoin 语法，重分配连接也还是采用的 Hadoop 自带的 datajoin 脚本。

列出的数据有两表参与连接的行数（字节数数量级），输出数统计（由于部分测试可能导致输出数据非常大但毫无意义，所以统一输出 count 后的结果），各个脚本执行后得到的执行总时间结果。具体见下表 4.2。

表 4.2 非等值连接测试结果表

Lineitem (行)	orders (行)	统计结果输出(行)	广播连接的非等值连接测试结果(秒)	重分配连接的非等值连接测试结果(秒)	比例划分的非等值连接测试结果(秒)	开方划分的非等值连接测试结果(秒)
6000(1M)	1500	4515887	66	65	50	51
12000(2M)	3000	17983721	75	75	60	67
60000(10M)	15000	451776715	349	413	298	277
120000(20M)	30000	1806195592	1234	1474	1056	941/1803

注：lineitem 的行数去掉了后几位，尾数导致非等值开方的 20M 测试结果有不同环境下的两组，具体见后节详解。

4.3.2 单路非等值连接的算法比较

单路非等值连接的算法测试中，参与实验的测试算法有：基于单路非等值连接的重分配连接算法、基于单路非等值连接的广播连接算法、基于单路非等值连接的本文提出的基于比例划分的连接算法和基于单路非等值连接的基于开方划分的连接算法。

首先对基于单路非等值连接的重分配连接算法和基于单路非等值连接的广

播连接算法进行比较。

由表 4.2 的测试结果可以看出在这两种算法在处理单路非等值连接问题的速率基本在一个数量级上，广播连接的测试时间结果比预分配连接的测试时间结果快了 15%左右。

对基于单路非等值连接的传统连接算法和基于单路非等值连接的本文提出的连接算法进行比较。

由表 4.2 的测试结果可以看出在这两种算法在处理单路非等值连接问题的速率基本在一个数量级上，本文提出的基于比例划分的连接算法的测试时间结果比传统连接算法（广播连接）的测试时间结果快了 15%左右。

最后对基于单路非等值连接的本文提出的基于比例划分的连接算法和基于单路非等值连接的本文提出的基于开方划分的连接算法进行比较。

由表 4.2 的测试结果可以看出这两种算法在处理单路非等值连接问题的速率基本在一个数量级上，本文提出的基于比例划分的连接算法的测试时间结果和本文提出的基于开方划分的连接算法的测试时间结果在不同环境下有不同表现，但测试所用时间基本上相差无几，其中基于开方划分的连接算法在测试 20M 数量级连接表数据的时候由于设置环境(最多可调用 Reducer 个数)的不同，几乎相差了一倍，其中较快的一组测试数据相比较基于比例划分的连接算法的测试时间结果又要稍快一些，其中的种种原因在将在下节中来具体说明。

4.3.3 单路非等值连接的算法分析

从上节的基于单路非等值连接的重分配连接算法和基于单路非等值连接的广播连接算法的比较结果，可以看出广播连接的测试速度要稍快与重分配连接，这是因为广播连接是一个 Maponly 的任务（只有 Map 阶段而没有 Reduce 阶段）所以虽然两种算法的计算复杂度相差无几，但广播连接的任务开销要比重分配连接小很多，这也就使得广播连接算法必然会稍快于重分配连接算法。

从上节的基于单路非等值连接的传统连接算法和基于单路非等值连接的本文提出的连接算法的比较结果，可以看出基于本文提出的连接算法编写的测试脚本的执行速度要稍快与传统连接算法测试结果，这是因为本文提出的数据划分算法进一步将 I/O 开销降到了接近理论的优化值，从而使分布计算得到更合理的处理数据量，使得执行速率更快。

从上节的比例划分的连接算法和开方划分的连接算法的比较结果，可以看出两种解决方案的复杂度基本在一个数量级上，单纯看测试结果开方划分算法还要略好于比例划分算法，但其实这是和测试环境和测试数据量有一定关系的。由于测试数据并不是理想比例，低位会存在一些多出的数据，从而导致了开方划分在计算过程中会出现误差（开方误差是开方划分存在的一个很大的问题），存在误差后开方划分无法保持原定的 Reducer 个数，Reducer 个数被重新计算后由原定

的 16 个增加到了 18 个，而实验环境的 cluster 有可用 cpu core 20 个，Hadoop 环境配置里可执行的 Reducer 最大任务个数也是 20 个，是可以承担 18 个 Reducer 任务的，不存在 cpu 资源不足的情况，也不存在 Reducer 任务数不足的情况，在满足 cpu 资源可进一步利用并且 Reducer 任务个数未超出设置的最大个数的前提下，由于 Reducer 的个数增加，导致每个 Reducer 得到的划分数据变小，从而使得每个 Reducer 的计算量变小，而每个 Reducer 是并行计算的，相互之间没有干扰，这就导致了整体脚本执行总时间缩短，执行速度加快。

在将 Hadoop 环境修改成最大 Reducer 个数限定为 16 个后，再次执行开方划分脚本，则测试结果和原来的结果相差了接近一倍的时间，这是由于 18 个 Reducer 任务在 16 个最大 Reducer 限定的环境下要分两批进行计算，第一批先执行 16 个 Reducer 任务，剩下的两个多出的 Reducer 任务会在第二批执行，通过各个节点的进程执行情况也能观察到第二批 Reducer 任务执行时，只有两个 Reducer 进程在高度占用 cpu 进行密集计算，由此也能理解为什么换了一个环境测试时间结果就长了一倍的原因了。

4.4 小结

本章首先介绍了实验的测试环境，然后列出了实验的测试结果数据表，从基于单路等值连接和基于单路非等值了连接，简单对现存的 MapReduce 连接算法和本文提出的算法作出性能上的比较和数据背后详细的原因分析。

在基于单路非等值连接算法方面，实验进行了 3 组测试结果数据的对比和分析，对算法的优缺点进行了深入的分析和探讨，最终得出本文提出算法的优势所在和优势背后的原因。

5 总结与展望

基于 MapReduce 分布式计算架构的连接算法是大数据存储和大数据挖掘分析的交叉学科领域，是学术研究的热点。随着 MapReduce 分布式计算框架的流行和 Hadoop 开源社区的发展，使用 MapReduce 分布式计算框架来进行数据挖掘和数据分析的需求越来越多，如传统的数据统计，内容推荐，机器学习，用户行为分析等都逐渐应用到 MapReduce 处理中。应用这些技术，MapReduce 框架为数据挖掘数据分析提供了快速高效的分布式计算框架。

基于 MapReduce 的单路非等值连接是单路连接算法中较难突破的一部分，对单路非等值连接的研究会为 MapReduce 的单路等值连接提升可观的性能，具有巨大的实际应用和理论研究价值。本文在研究过去提出的 MapReduce 连接算法的基础上，提出了更适用于单路非等值连接的分布式计算算法，最后实现了任务执行速率的可观提升。

在本文实现的算法中主为基于单路非等值连接的本文提出的基于比例划分的连接算法和基于单路非等值连接的本文提出的基于开方划分的连接算法，对全文的的工作总结如下：

(1) 研究了有关 MapReduce 的单路连接的相关算法，包括重分配连接算法和广播连接算法。此外，对重分配连接算法和广播连接算法在处理等值连接上的表现作了对比，大体上重分配连接更适合处理等值连接这类问题；同时，对重分配连接算法和广播连接算法在处理非等值连接上的表现作了对比，大体上两者都可以进行很好的非等值连接处理，但广播连接稍快一些，在考虑存在的部分差异的基础上，可以将基于广播连接的部分算法思想应用到本文提出的新算法上。

(2) 针对现有的 MapReduce 非等值连接算法的不足，把数据划分和 I/O 开销进行优化计算，利用修改 Hadoop 的 Partition 部分源码实现 Map 数据向 Reduce 的多次发送，在 Reducer 端通过数据大小判断选择使用缓存表解决方案，或者使用修改 Hadoop 源码 Reduce 中 Context 文件操作方式实现的 I/O 重定位反复读方案来对各个表的部分数据做笛卡尔积遍历筛选。从而提出了针对单路非等值连接的新算法，这种方法，完成了针对单路非等值连接任务的处理工作，提高了执行效率。

(3) 通过对测试时间结果的比较和研究，根据比例方案和开放方案在处理单路非等值连接问题上表现以及针对不同环境得到的不同结果，本文总结了两种算法的异同和对于不同环境的适应性和扩展性。

尽管本文的研究工作做出了一些有意义的研究，但对于基于 MapReduce 的

连接算法的整体性兼顾不够，以及局部算法的效率有待提高。以下几点可供论文工作进一步完善和改进作参考：

(1) 引入缓存机制

根据对环境的探测引入缓存机制，可让数据量在内存可接受的范围内时，减少效率相对较低的 I/O 操作，从而大大提升计算性能,如此任务的执行速度会变得更快。

虽然缓存机制先前由于存在一些问题，无法适应一些环境，但重写后 的 Context 文件读取机制，可以更好的支持缓存，原本内存无法全部容纳的数据，现在可以通过重写后的数据读取机制，将需要缓存的数据划分后分批次缓存，从而大大提高计算速度。

(2) 引入索引机制

Hadoop 的 HDFS 以文件形式存储数据，并没有完善的索引机制，这导致两表做非等值连接时必然要对两表的元组进行笛卡尔积遍历，这样的时间复杂度是很高的，如果可以引入类似数据库那样的索引机制，则非等值连接操作可以通过索引避免无谓多余的无效连接从而可提升一倍左右的效率。

参考文献

[1] Abelló A, Ferrarons J, Romero O. Building cubes with MapReduce. Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP. ACM, 2011: 17-24.

[2] Afrati F N, Ullman J D. Optimizing joins in a map-reduce environment. Proceedings of the 13th International Conference on Extending Database Technology. ACM, 2010: 99-110.

[3] Chandar J. Join Algorithms using Map/Reduce. Magisterarb. University of Edinburgh, 2010.

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

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

[6] Yang H, Dasdan A, Hsiao R L, et al. Map-reduce-merge: simplified relational data processing on large clusters. Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007: 1029-1040.

[7] Apache Hadoop, http://Hadoop.apache.org/, 2007.

[8] （美）怀特著，曾大聃，周傲英译，周敏审校.Hadoop 权威指南（中文版），2011

[9] 刘琨，董龙江.云数据存储与管理.计算机系统应用，2011，20(6):232-237

[10] 祝海通. MapReduce 环境中基于列存储的一种高效的星型连接方法.研究生学位论文，2012

[11] 潘巍，李战怀，陈群，索博，李卫榜. 面向 MapReduce 的非对称分片复制连接算法优化技术研究，计算机研究与发展，2012，49(suppl)，296-302

[12] Blanas S, Patel J M, Ercegovac V, et al. A comparison of join algorithms for log processing in mapreduce. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 975-986.

[13] Han H, Jung H, Eom H, et al. Scatter-Gather-Merge: An efficient star-join query processing algorithm for data-parallel frameworks. Cluster Computing, 2011, 14(2): 183-197.

[14] O’Neil P, O’Neil E, Chen X, et al. The star schema benchmark and augmented fact table indexing. Performance Evaluation and Benchmarking. Springer Berlin Heidelberg, 2009: 237-252.

[15] Tpc-h, http://www.tpc.org/tpch/, 2001.

[16] Peng M T, Huang H H. Aging of hypothalamic-pituitary-ovarian function in the rat. Fertility and sterility, 1972, 23(8): 535.

[17] Apache hive, http://hive.apache.org/, 2013

[18] White T. Hadoop: The definitive guide. O'Reilly Media, Inc., 2012.

[19] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system. Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on.

IEEE, 2010: 1-10.
[20] Borthakur D. The hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007, 11: 21.
[21] Zhang J, Sivasubramaniam A, Franke H, et al. Synthesizing representative i/o workloads for tpc-h. Software, IEE Proceedings-. IEEE, 2004: 142-142.
[22] Kandaswamy M A, Knighten R L. I/O phase characterization of TPC-H query operations. Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International. IEEE, 2000: 81-90.
[23] Shafer J, Rixner S, Cox A L. The Hadoop distributed filesystem: Balancing portability and performance. Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on. IEEE, 2010: 122-133.
[24] Leverich J, Kozyrakis C. On the energy (in) efficiency of hadoop clusters. ACM SIGOPS Operating Systems Review, 2010, 44(1): 61-65.
[25] Jiang D, Tung A K H, Chen G. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. Knowledge and Data Engineering, IEEE Transactions on, 2011, 23(9): 1299-1311.
[26] Wang F, Shi Y. An Improvement of Choosing Map-join Candidates in Hive. Procedia Computer Science, 2012, 9.
[27] Blanas S, Patel J M, Ercegovac V, et al. A comparison of join algorithms for log processing in mapreduce. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 975-986.
[28] Thusoo A, Sarma J S, Jain N, et al. Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2009, 2(2): 1626-1629.
[29] Kaldewey T, Shekita E J, Tata S. Clydesdale: Structured data processing on MapReduce. Proceedings of the 15th International Conference on Extending Database Technology. ACM, 2012: 15-25.
[30] Patel J M, DeWitt D J. Partition based spatial-merge join. ACM SIGMOD Record. ACM, 1996, 25(2): 259-270.