毕业设计（论文）报告

题目　集群缓存系统负载均衡算法的设计与实现

计算机科学与工程 院（系）计算机科学与技术 专　业

学　　号　09015131

学生姓名　郑云川

指导教师　王威

顾问老师　肖卿俊

起讫日期　2018 年 12 月—2019 年 6 月

设计地点　HKUST

东南大学毕业（设计）论文独创性声明

本人声明所呈交的毕业（设计）论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得东南大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

作者签名：_____

日 期：_____

东南大学毕业（设计）论文使用授权声明

东南大学有权保留本人所送交毕业（设计）论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括刊登）论文的全部或部分内容。论文的公布（包括刊登）授权东南大学教务处办理。

作者签名：_____

导师签名：_____

日 期：_____

集群缓存系统负载均衡算法的设计与实现

09015131　郑云川
指导教师　王威

摘　要

希腊字母源自腓尼基字母。腓尼基字母只有辅音，从右向左写。希腊语的元音发达，希腊人增添了元音字母。因为希腊人的书写工具是蜡板，有时前一行从右向左写完后顺势就从左向右写，变成所谓“耕地”式书写，后来逐渐演变成全部从左向右写。字母的方向也颠倒了。罗马人引进希腊字母，略微改变变为拉丁字母，在世界广为流行。希腊字母广泛应用到学术领域，如数学等。

希腊字母是希腊语所使用的字母，是世界上最早的有元音的字母，也广泛使用于数学、物理、生物、天文等学科。俄语等使用的西里尔字母也是由希腊字母演变而成。希腊字母进入了许多语言的词汇中，英语单字“alphabet”（字母表），源自拉丁语“alphabetum”，源自希腊语“αλφαβητον”，即为前两个希腊字母 α（“Alpha”）及 β（“Beta”）所合成，三角洲（“Delta”）这个词就来自希腊字母 Δ，因为 Δ 是三角形。

关键词：希腊字母，腓尼基字母，语言，深度学习

Design and Implementation of Load Balance Algorithms in Cluster Cache Systems

09015131 Yunchuan Zheng
Advisor Wei Wang

Abstract

The Greek alphabet has been used to write the Greek language since the late 9th century BC or early 8th century BC It was derived from the earlier Phoenician alphabet, and was the first alphabetic script to have distinct letters for vowels as well as consonants. It is the ancestor of the Latin and Cyrillic scripts.Apart from its use in writing the Greek language, in both its ancient and its modern forms, the Greek alphabet today also serves as a source of technical symbols and labels in many domains of mathematics, science and other fields.

In its classical and modern forms, the alphabet has 24 letters, ordered from alpha to omega. Like Latin and Cyrillic, Greek originally had only a single form of each letter; it developed the letter case distinction between upper-case and lower-case forms in parallel with Latin during the modern era.

KEY WORDS: Greek Alphabet, Phoenician Alphabet, Language, Deep Learning

目 录

第一章 前言

本章是课题的前言部分。在此章中，首先介绍了课题的实际背景，接下来是课题的目的和意义，并且对当前的研究现状做了简要调研与分析，最后介绍了课题的主要研究内容。

1.1 课题背景

1.1.1 大数据

因为技术的不断发展，包括物联网，云计算的崛起[1]、智能设备的流行等，在当今时代各种各样不同的领域（例如健康领域、政府、社交网络、营销、财务），每一天都在以前所未有的速度产生大量的数据[2]。从海量的数据中，我们能够挖掘出大量有用的规律，对人们的生活产生积极的影响。在大数据革命之前，大公司很难将他们的数据存档保存较长时间，也难以管理庞大的数据集。传统技术存储能力有限，管理工具很昂贵，它们缺乏大数据背景所需要的灵活性、可扩展性和性能。事实上，大数据管理需要大量资源，新方法和强大技术，进一步来说，大数据需要清洗，处理，分析，保护数据，并提供对大量不断发展的数据集的细粒度访问[2]。为了应对大数据带来的机遇和挑战，学界和业界开展了大量的研究与开发工作，发展出来众多技术，提供了很多成熟的模型、框架、软件。典型的互联网大数据平台（如图 1.1）从上至下大致可分为三个部分：

- 数据采集：将应用程序产生的数据和日志等同步到大数据系统，同步时数据可能还需经过清洗、转化等过程；
- 数据处理：包括大数据存储、离线计算和流式计算等；
- 数据输出与展示：大数据经过处理后将有价值的信息存入数据库，通过数据库给用户提供所需信息，或者给运营、决策人员提供所需信息。

此外，将三个部分整合起来的是大数据任务调度管理系统，它会管理数据的同步、集群资源的分配等等。

在大数据平台中（图 1.1），数据处理是非常重要的一环，实现对数据的高效清洗、存储和挖掘是这个环节的目标。对计算能力的需求随着数据量的急剧增加而增加，但是单机的处理能力和 I/O 性能并没有跟上这种增长，越来越多的企业不得不向外扩展他们的计算至集群模式[3]。集群环境对编程平台提出了更高的要求，主要有三方面，一是程序需要并行化执行，二是需要强大的容错能力，三是动态地扩展和缩减计算资源，为此，越来越多的编程模型被设计出来。在存储方面，谷歌提出了分布式文件系统 GFS（Google File System）[4]，对应的开源实现是 HDFS（Hadoop File System）[5]，它实现了对成千上万台机器上的大规模数据进

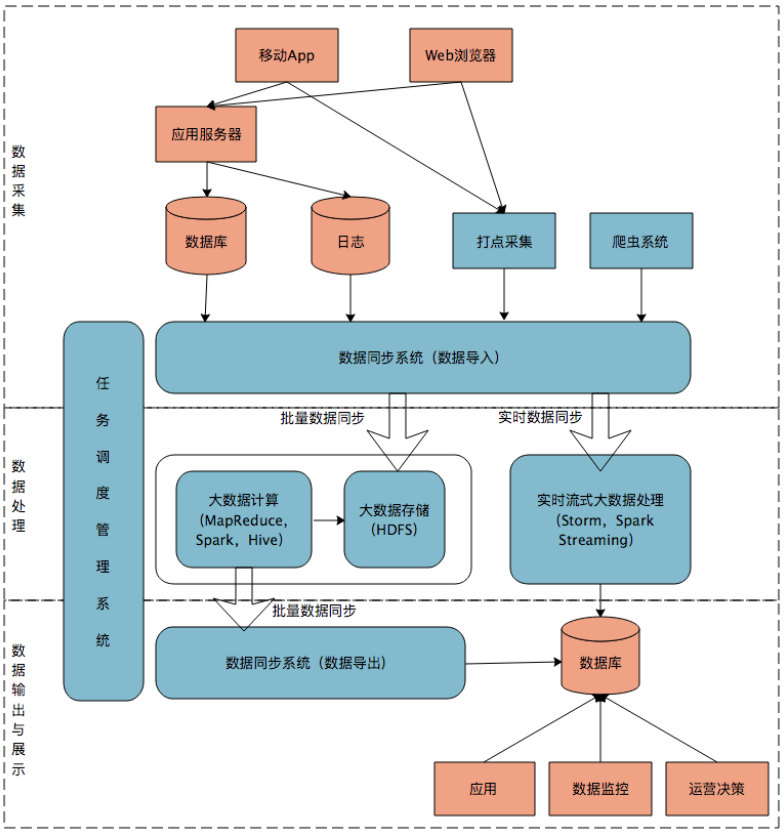


图 1.1 典型的互联网大数据平台架构。

行高效地存储、访问，并且具有高度容错能力。计算框架方面，起初，谷歌的 MapReduce[6] 提出了一种简单通用而且能够自动处理故障的批处理计算模型，但是它将中间以及最终结果保存在磁盘上，消耗大量 I/O 时间，不利于重用计算结果。Spark[3]、Pregel[7] 等采用内存计算方案来加速计算，实现数据的重复利用。

1.1.2 集群缓存

由于近来数据中心架构的改进 [8] 和高速网络设备的出现[9–11]，网络带宽和存储器 I/O 带宽之间的差距正在迅速减小 [12–15]，因此，云计算系统的性能瓶颈正迅速从网络转变为存储器 I/O。先前的工作证明从本地硬盘读取数据相比网络远程读取并没有显著的优势 [16]，这个结论同样适用于固态硬盘（SSD）。最近的一个研究 [17] 表明将数据存储在 EC2 实例的一个本地 SSD 上甚至比把数据写到 Amazon S3 [18] 上还要慢，Amazon S3 是一个远程的提供了 PUT/GET 接口的对象存储服务。当磁盘本地化变得无关紧要，云端对象存储如 Amazon S3 [18]、Windows Azure Storage [19]、和 OpenStack Swift [20] 等，逐渐取代与计算同地的存储（尤其是 HDFS [21]），作为数据密集型应用的首选存储方式。

然而，云端对象存储在磁盘 I/O 上依然是瓶颈 [22]，因为从磁盘读数据比从内存读数据慢至少两个数量级，考虑到这个问题，集群缓存系统，例如 Alluxio [23]、Memcached [24] 和 Redis [25]，被越来越多的云端对象存储系统部署来提供内存速度级别的低延迟数据访问，而集群缓存系统面对的一个很大的挑战便是如何实现负载均衡。

一个能够实现分布式内存缓存的系统是 Alluxio。Alluxio[26] 是开源的分布式内存文件系统，旨在作为上层繁多的计算框架（如 MapReduce、Apache Spark、Apache Storm、Apche

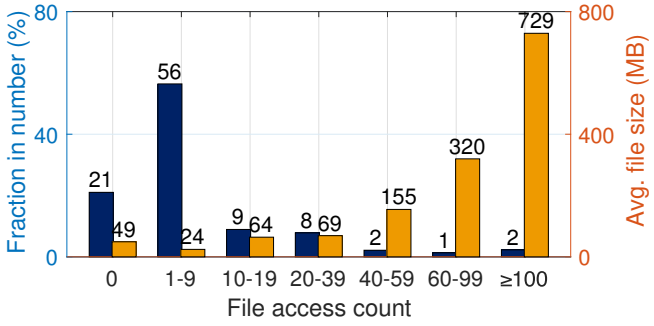


图 1.2 Yahoo! 集群上观察到的文件热门程度（蓝色）和文件大小（橙色）的分布 [29].

Mahout 等）与底层存储层（如文件系统、对象存储、键值对存储等）的中间层，提供统一的文件读写的接口，实现全局的数据访问、高效的内存数据共享、跨应用的数据管理、高效的网络带宽利用，它借助“血缘关系”、检查点机制提供强大的容错能力。鉴于这些特点，alluxio 也非常适合作为内存缓存系统，进一步加速数据分析应用。

1.1.3 负载均衡

上一小节提及的 Apache Spark 等内存计算方案的一个重要挑战是负载不均，在先前工作 [22, 27] 中，研究人员已经发现了生产集群中负载不均的两个来源：文件热门程度差别和网络流量不均。

在数据中心中，我们普遍观察到，文件（数据对象）热门程度差别极大，并且遵循 Zipf 分布 [22, 26–28]，也就是说，数据访问的大部分请求是由一小部分非常热门的文件贡献的。图 1.2描述了 Yahoo! 集群查询数据集 [29] 中文件的热门程度和文件大小的分布，从这个数据集可以得到某两个月内对超过四千万个文件的访问的统计数据。我们发现绝大多数文件（$\sim 78\%$）存储的是冷门数据，很少被访问（< 10 times），只有 2% 的文件有高访问量（≥ 100），这些文件通常比那些冷门的文件大很多（15-30×）。由于这些文件较大的体积和较高的访问量，缓存这些文件的服务器很容易负荷过重。

这个问题由于网络负载不均而加重，这在生产环境的数据中心非常普遍 [22, 30–32]，例如，在研究 [22] 中，研究者测量了 Facebook 一个集群所有上行和下行链路中最大利用率和平均利用率的比值，结果表明这个比值在半数以上的时间里保持在 4.5 以上，这意味着严重的负载不均。在 SP-Cache [33] 的研究中，研究人员分别测量了在有无内存缓存的情况下，不同请求速率下的文件的平均读延迟，结果表明 1.3当集群负载不重时（每秒 5 个请求），内存缓存带来了显著性好处，降低平均读延迟达 5×，然而，当负载骤然增大，集群中的热点机器变得突出，缓存带来的好处迅速减少，尤其当请求速率大于 9，读延迟就由热点服务器的网络拥塞决定，内存缓存就变得无关紧要。

1.2 研究现状

当今的数据并行集群依赖内存计算方案来进行高性能的数据分析工作[3, 24, 26, 34–36]，通过将数据对象缓存在内存中，I/O 密集型应用相对于传统的磁盘解决方案能够获得数量级的性能提升[3, 26, 35]

然而，内存计算方案面对的一个严峻的挑战是缓存服务器之间严重的负载不均衡。在生

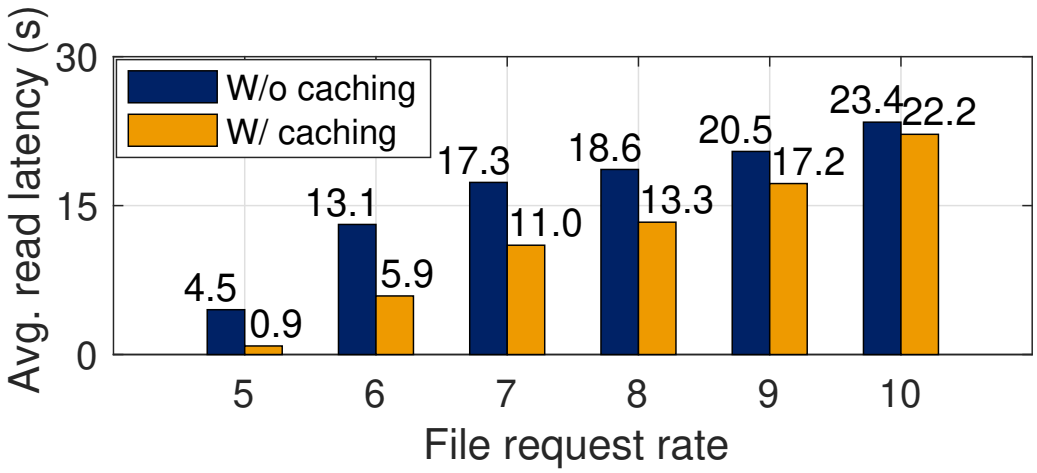


图 1.3 有/无缓存的情况下平均读延迟随着负载增加而增加。

产集群中，数据对象有严重的热门程度差别，这意味着对一小部分非常热门的文件访问占据了总访问的很大一部分[22, 27, 28]。存储有热门文件的缓存服务器因此变为访问热点，这个问题因为网络的负载不均衡而进一步恶化。据报道，在 Facebook 的一个集群中，负载最重的链路的利用率在 50% 的时间里比平均链路利用率高出 4.5 倍[22]。访问热点和网络负载不均导致了 I/O 性能极大下降，这甚至可能会抵消内存计算带来的性能提升。

因此，保证负载均衡是提高集群缓存性能的关键，这方面的解决方案包括选择性复制[27]、纠删码[22] 和选择分割[33]，前二者是借助缓存冗余来减缓访问热点机器的负担，第三个是根据文件的热门程度将文件分割成不同份数，并随机放置在不同服务器上，分散请求负载。

1.2.1 选择复制

选择复制方案基于文件的热门程度对文件进行复制 [27, 37]，也就是说，一个文件的访问频率越高，它会越多被复制，并分散在集群中，一个文件的读请求就能随机选择一台含有这个文件副本的服务器提供服务。这样，读请求的负载就被均匀分散，提高负载均衡。

虽然选择复制已被证明对于基于磁盘的存储系统是有效的 [27]，但是因为复制带来了高额的内存开销，它在集群缓存上表现不佳 [22, 38]。研究 [33] 的实验得出，内存开销的 线性增加（热门文件副本数增加）换来了读延迟的 亚线性降低，而且热门文件的体积通常比较大（图 1.2）。

1.2.2 纠删码

有研究利用纠删码 [39, 40] 来实现缓存服务器之间的负载均衡且避免产生高额的内存开销。一个 (k, n) 的纠删码方案能够将一个文件均匀地切分成 k 份，然后计算同样大小的 $n-k$ 个奇偶校验分区，原始文件能通过解码 n 份中的任意 k 份来获得，从而使得读请求的负载被分散到 n 台服务器上。内存的额外开销是 $(n-k)/k$，在实际设定中比选择复制低（至少 $1\times$）。

这个方法的一个有效实现是 EC-Cache [22]，它在读取文件时通过迟绑定来减轻落后机器的影响，换句话说，EC-Cache 随机读取文件分区中的 $k+1$ 份，等待其中的 k 份完成读取，而不是恰好读取 k 份。EC-Cache 在读文件的中位和尾延迟都比选择复制低很多 [22]。然而，EC-Cache 在读（写）时带来巨大的解码（编码）的额外开销，即使有高度优化的编码和实现方案，解码的开销仍会对读请求产生高达 30% [22] 延迟。

1.2.3 选择分割

研究 [33] 中提出了 SP-Cache 来实现集群缓存的负载均衡，同时避免高额的内存和计算开销。它选择性地将热门文件根据其大小和热门程度，分割成一定数目的文件分区，随机缓存到不同的缓存服务器上，这样分散了读请求的负载，同时读操作可以并行，提升性能。SP-Cache 建立了一个上限分析来量化平均延迟 [33]，并基于这个推导设计了一个高效的算法来决定每个文件的最佳分区数量，文件分割的数目太小则不足以缓解热点机器的压力，分割的数目太大则容易受到慢机器的影响。此外它采集一段时间内集群中文件的访问数据，周期性地调整各个文件的分区数目。选择分割在不产生高额内存和计算开销的情况下实现可负载均衡，但因为其分割的特性，缓存无冗余，容错性依赖底层文件系统，且读取文件必须读取所有分区，会受到慢机器的影响。

1.2.4 更细粒度负载均衡

以上方案都是针对一般意义上的文件来考虑负载均衡的，优点是非常通用，毋需考虑文件的语义，对于任何格式的文件都可以使用。它们负载均衡的粒度是文件，那么问题来了，能否在更细的粒度进行负载均衡，提高缓存效率呢？对于语义清晰的结构化数据，比如 Parquet 文件 [41]，如果在文件的内部列与列存在热门程度差异，列之间被共同查询的概率也存在差异，那么就没有必要去分割或者复制整个文件，只要对一个文件热门的这一部分，例如其中一列或者多列进行复制或者分割就行。这样能够节约内存，提高使用效率，因为内存总是有限的，而且一部分内存需要给计算任务使用，那么留作缓存的就更少了。这个目标的挑战在于底层分布式文件系统需要了解文件的语义，与上层的应用通信来获得这部分信息，可能产生一定程度的耦合，同时我们需要明智地决定对文件的哪些列进行复制或分割，在哪些机器上进行缓存。

1.3 研究的目的与内容

当前的大数据系统主要采用复制的方式来进行容错和负载均衡，而服务器内存的容量往往有限，缓存会产生不可忽视的内存开销。根据本项目的前期调研，生产集群中结构化数据（数据表）的不同列之间，热门程度（被访问热度）存在差异，列与列之间共同被查询的概率也存在差异，我们希望复制数据表中比较热门的列，而不是全表，并基于列与列之间被共同查询的概率设计一定的放置策略，实现以更少的内存，来获得相似的负载均衡效果的目标，从而节约资源，提高缓存效率。

总的来说，本课题的主要研究内容是上文提出的针对结构化数据文件的更细粒度（列级别）的负载均衡方案，具体来说：

- 利用具有代表性的基准查询数据集，如 TPC-DS, TPC-H 等，测量数据表中列之间的查询频率，以及列与列之间被共同查询的频率，分析其中的统计及其他客观规律，为本项目的可行性奠定理论基础。

- 通过实验探究 SQL 查询过程中，数据的 shuffle 过程对任务执行时间的影响，证明数据表中相关列“捆绑放置”（bundle）的有效性，进一步强化项目的理论基础。

- 搭建 Spark SQL [42]，Alluxio [23]，HDFS [21] 为主的集群系统，探索各组件之间通信协作机制，为项目方案实现奠定基础。
- 基于已有条件，主要在 Alluxio [23] 基础上添加模块，使得 Alluxio 能够获取热门度以及关联性等信息，查阅相关文献，设计实现细粒度的负载均衡算法，并在 AWS EC2 搭建实验环境，进行测试。

1.4 论文结构

第二章　研究动机

过往的很多工作研究了文件被访问热度（skewed file popularities）差别很大的情况下集群缓存系统的负载均衡，在这些工作中，文件复制是被广泛应用的方法。例如，HDFS 默认把一份文件复制为 3 份；当缓存服务器上发生缓存缺失（cache miss），Alluxio 弹性增加热门文件在内存中的副本数。

然而文件复制会带来较高的内存开销，最终损害缓存带来的好处。在数据分析的工作中，批处理任务分析结构化数据情况比较多，结构化数据相比一般意义上的文件具有更多的上下文信息，其中列式存储的文件格式，比如 Parquet[41]，得到越来越多的应用，因为在数据分析的大部分任务中，通常需要读取相关列，而不是整行数据，且列式存储把相同类型的数据归在一起，压缩比可以很高。那么问题来了，针对采用列式存储的结构化数据，我们是否能够根据其特有的性质，保证负载均衡的效果的同时，降低文件复制的开销呢？以往的研究工作表明，对一小部分热门文件（被高频访问的文件）访问占据了集群总访问量的大部分，我们猜想，那么对于结构化数据来说，例如具体一张数据表，列与列之间是否存在热门程度的差异呢？如果猜想成立，直观上来说，对于一张表，我们可以只复制最热门的几列，就能达到接近复制全表的负载均衡效果，同时能够降低缓存开销。

如果按照上文所述，仅复制最热门的若干列，缓存在不同的机器上，那么在执行分布式 SQL 任务的时候，极有可能发生表内部的数据 shuffle，我们需要探究 shuffle 对任务执行时间的影响。直观上来说，数据 shuffle 带来网络通信上的开销，会降低任务执行的效率，那么在考虑被复制的列在集群里的放置策略时，我们也需要考虑列与列之间被共同访问的概率，如果两列有很大可能性会被一起访问，那么可以考虑将它们“捆绑”（bundle）在一起放置。

在本章 2.1节中，我们通过对标准基准数据集 TPC 系列的分析，来证明数据表中列与列之间，被访问频率存在差异，并且当考虑两两之间被共同访问的概率，两列各自的被访问频率越高，它们被共同访问的频率也越高。在 2.2节中，我们通过实验证明数据 shuffle 会对 SQL 任务的执行会显著降低任务的执行效率。

2.1 列的访问规律

首先我们研究了具有代表性的基准标准测试程序 TPC 系列中的 TPC-H，TPC-DS，TPC-xBB 中的列的访问规律。

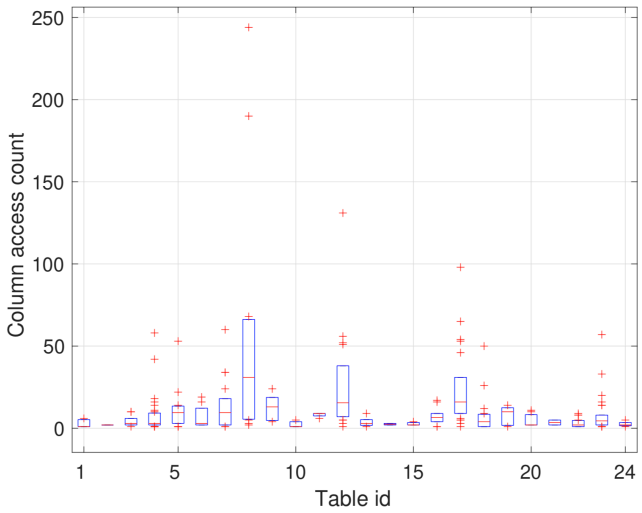


图 2.1 TPC-DS 标准测试程序中所有数据表的列访问计数的分布。箱形图展示了每张数据表里 25th 分位，中位数和 75th 分位的列访问次数计数。红色标记表示离群值。

2.1.1 实验设置

我们通过 2.4.0 版本的 Spark SQL 执行三种标准测试程序提供的查询任务。对于每一种，我们生成 1 GB[1]的数据，数据存为 Parquet 格式，然后将查询任务依次提交。三种标准测试程序各自包含的数据表的数量和查询任务的数量总结在表格 2.1中。当执行查询任务的时候，我们记录对列的访问数据，以此来分析列级别的数据访问的性质。我们从结果中观察到以下两个现象。

2.1.2 文件内列访问频率偏差

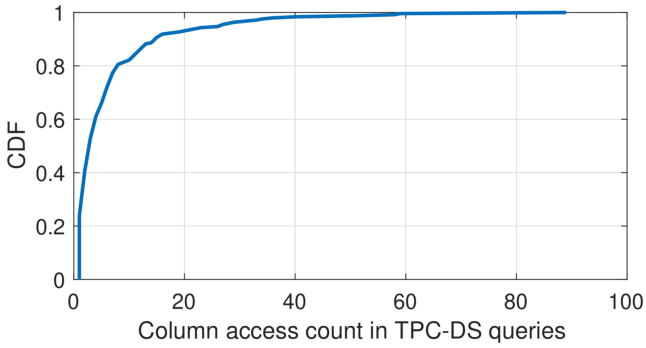
在上述三种标准测试程序中，我们观察到，每张数据表内，列的访问频率存在显著差异，即每张表里只有一小部分列被经常访问，而其他的列访问频次比较低。为证明这点，我们对三种标准测试程序中各数据表的列的访问进行了计数。图 2.1展示了 TPC-DS 标准测试程序中每张表中列访问计数的分布，箱形图展示了每张数据表里 25th 分位，中位数和 75th 分位的列访问次数计数。每个红色标记表示异常值，特别地，在箱形图上方的红色标记代表该表中访问频率特别高的列。从图上可以看出，对于 TPC-DS 的多数表，箱形图上方的红色标记远远高于箱形图顶部，这些“热门”的列被访问的次数远远超过均值，这说明文件内列访问频率存在显著偏差。此外，对于各个表而言，表越“热门”（它的列整体上访问频率高），列之间访问频率的差异越大。

相似的性质也能在另外两种标准测试程序中看到，图 2.2展示了三种基准测试程序中所有列的访问计数的总体分布。我们发现，多数的列是“冷门的”，有很多列的访问次数是 1，甚至是 0，这在 TPC-DS（图 2.2a）和 TPC-xBB（图 2.2c）中表现比较明显，而一小部分列有非常高的访问计数。例如，在 TPC-DS 中，最“热门”的列被访问了多达 89 次。

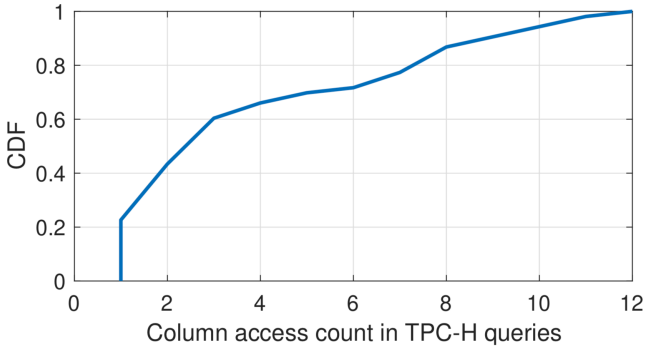
[1]这个实验与数据量无关，因为数据表的个数和每个表的访问规律不回随着表的规模而改变

表 2.1 三种标准测试程序的数据

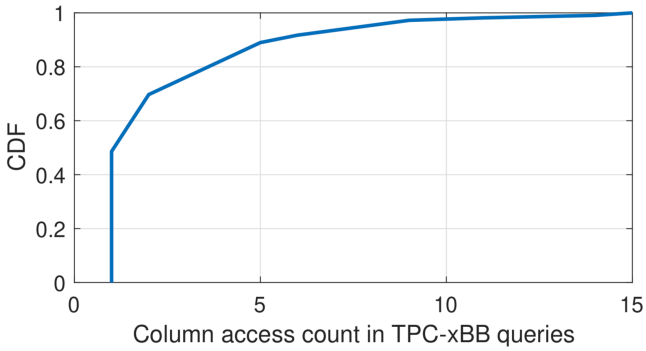
标准测试程序	表的数量	查询任务的数量
TPC-DS	24	99
TPC-H	8	22
TPC-xBB	19	30



(a) TDC-DS.



(b) TDC-H.



(c) TDC-xBB.

图 2.2 三种标准测试程序中列访问计数的 CDF。

2.1.3 热门的列被共同访问的规律

我们观察到的另一个现象是在 SQL 查询中，“热门”的列有很大概率会被共同访问。为了展示这一点，我们按照列的“热门程度”（访问频次）对列进行排序，绘制访问热图。我们从三个标准测试程序中各选出了一张具有代表性的表，并把结果展示在图 reffig:heatmap 中。在热图里，格 (i, i) (也就是对角线上的格子) 代表第 i^{th} 热门的列的访问计数，格子 (i, j) 表示第 i^{th} 热门和第 j^{th} 热门的列在同一个查询任务中被共同访问的概率。从图中可以看出，每张表中越是热门的列，被共同访问的概率越高。

2.2 列之间的数据 shuffle

从 2.1节中观察到的现象我们获悉，每张数据表中一小部分的列非常热门，访问频率很高。如果我们复制这一小部分热门的列，并将它们缓存在不同的机器上，那么当 SQL 查询任务在分布式环境中执行时，这些热门的列很容易引起集群节点之间的数据 shuffling。我们推测，这种 shuffling 给任务执行时间带来的影响是不可忽视的，为了展示列这一级别的网络开销，我们做了一个实验，测量一个小集群中列的热门程度与数据 shuffling 的关系。

2.2.1 实验设置

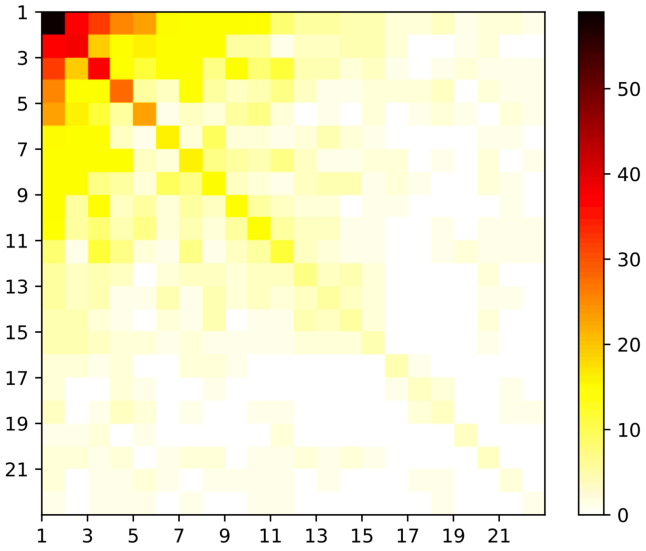
我们部署了一个含有 1 个 master 和 2 个 worker 的小集群，所用的实例是 c5.4xlarge，每一个有 32 GB 内存和 16 个 CPU 核，通过 iperf3 测试，小集群的网络带宽是 10 Gbps。我们在集群上部署了 Alluxio 以及 Spark，运行 TPC-H 标准测试程序。在实验中，只有一台 worker 缓存有 6 GB 的 Parquet 格式的数据，因此集群里每次执行查询任务都会引发从有数据的机器到没有数据的机器的数据 shuffling。我们关闭了 Spark 和 Alluxio 的数据被动缓存功能，一个一个按顺序执行标准测试程序里的查询任务，保证每一次执行都会有数据 shuffling。我们会记录每一列的总 shuffle 量。

2.2.2 每一列的数据 shuffle

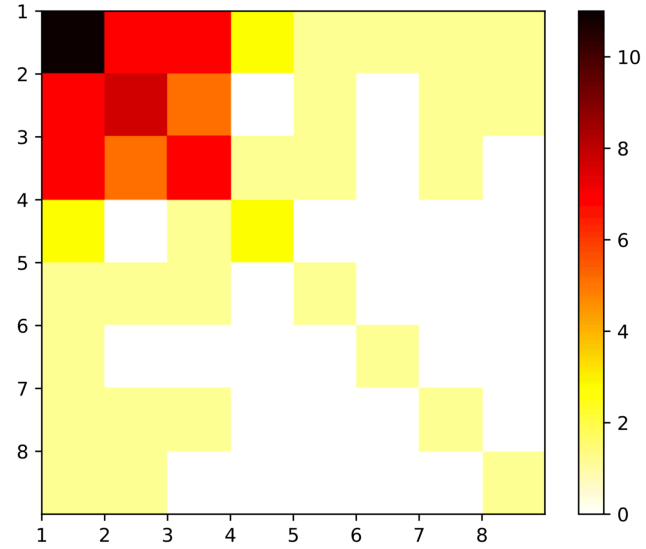
图 2.4展示了上述实验的结果，其中图 2.4a展示了列级别的数据 shuffle 量，TPC-H 标准格式程序中 53 列按照它们的热门程度排序。因为热门的列被频繁访问，并且我们发现通常来说热门的列比冷门的列的体积更大，所以热门的列相比冷门的列引起更多的数据 shuffle。根据图 2.4b展示的分布，总体来说，查询任务产生的数据 shuffle 主要来自热门的列，比如接近 90% 的数据 shuffle 量是由 30% 最热门的列贡献的。

2.3 数据 shuffle 的影响

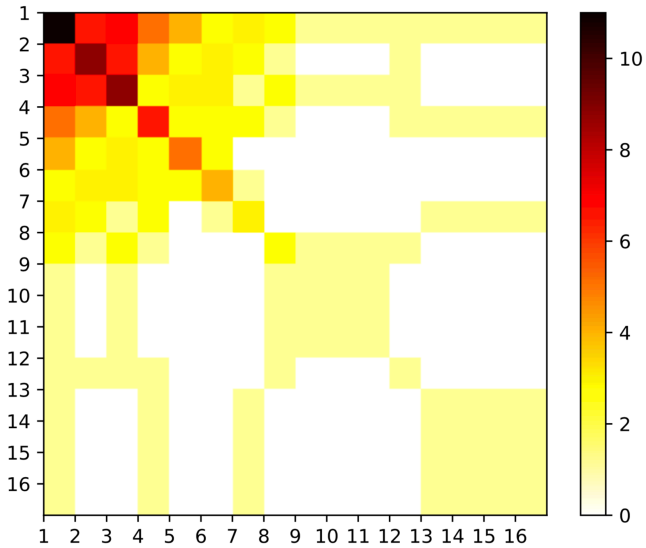
2.2节实验证明了热门的列很容易引起数据 shuffle，本节中我们会用实验证明数据 shuffle 会降低执行查询任务的性能。



(a) The $store\_sales$ table in TDC-DS.

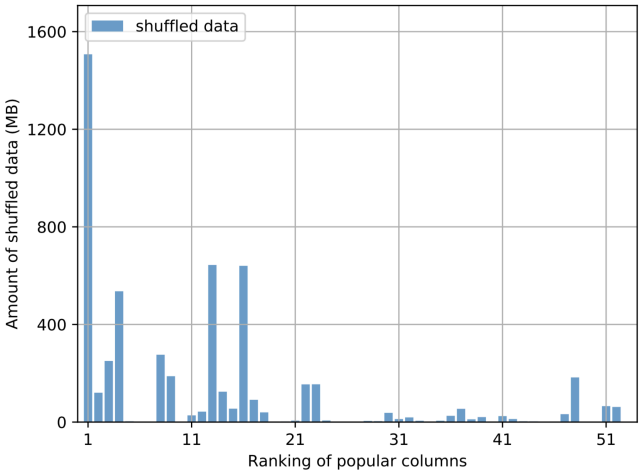


(b) The $orders$ table in TDC-H.

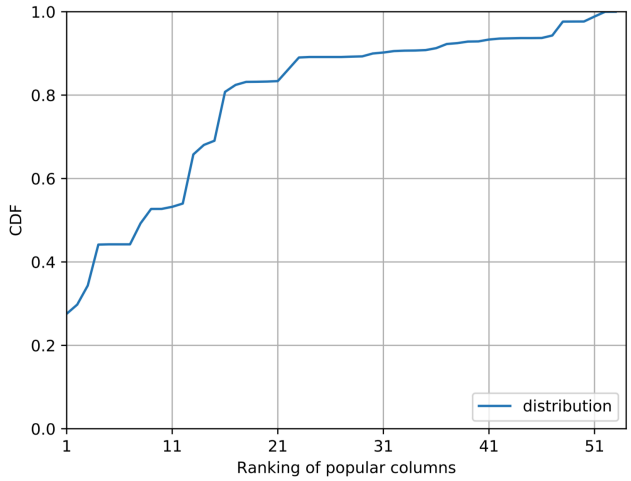


(c) The $store\_sales$ table in TDC-xBB.

图 2.3 Access heat maps of representative tables in the three benchmarks.



(a) 每一列的总 shuffle 数据量。



(b) 列级别 shuffle 数据的 CDF。

图 2.4 TPC-H 标准测试程序的列级别数据 shuffle。

2.3.1 实验设置

这个实验所用的集群与 2.2.1小节中描述的集群一致。我们设置了对照实验，其中实验组的设置与 2.2.1小节一致，只把数据缓存在一台 worker 上，这一组会产生数据 shuffle；另外一组中，我们将相同的数据在两台 worker 上均进行缓存，保证不会产生数据 shuffle。我们对比两组中查询任务的执行时间，以此测量 shuffle 的开销。

2.3.2 度量指标

我们使用任务的平均执行延迟 *slowdown* 作为衡量指标：

$$\text{Slowdown} = \frac{L_S - L_N}{L_N}, \tag{2.1}$$

其中 L_S 和 L_N 分别是由 shuffle 和没有 shuffle 的实验中查询任务的执行时间。*slowdown* 的值越大表明其降低查询任务执行的性能的影响越显著。

2.3.3 不同网络带宽下的 shuffle 开销

按照 2.3.1小节的设定，我们依次执行了 TPC-H 标准测试程序提供的查询任务并计算了每个查询任务的 *slowdown*（ 2.3.2）。图 2.5展示了网络带宽被限制为 1 Gbps，3 Gbps 和 10 Gbps 的情况下，*salowdown* 的分布。从图中可以看出，对于分布式环境下执行的 SQL 查询任务，网络是瓶颈，所以数据 shuffle 大大影响了任务执行的性能。例如，即便是在 10 Gbps 的带宽下，40% 的查询任务的延迟由于数据 shuffle 会上升 10%。此外，当网络带宽变得越小，shuffle 带来的通信开销会更加明显，任务的性能的下降也会更加显著。

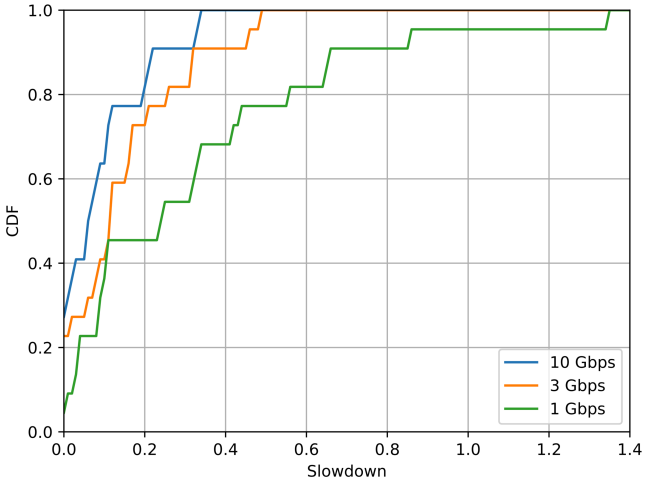


图 2.5 slowdown 在不同网络带宽下的分布。

2.4 总结

我们从本章第 2.1节得知，一张数据表中不同列之间热门程度（访问频率）存在明显的差异，且当考虑两两之间被共同访问的概率时，两列的热门程度越高，二者被共同访问的频率越高。在一张表中，热门的列是少数，其余多数是冷门的，不经常被访问。我们这些规律可以推断，相比对整张数据表进行复制，理论上复制数据表里相对热门的若干列能够达到接近复制整表的负载均衡的效果。因为冷门的列本身访问次数不多，在较长的一段周期内，热门的列的访问负载由其副本承担，而冷门的没有被复制的列的访问负载由原表承担，直观来说，这能够起到不错的负载均衡效果。与此同时，复制更少的列，降低缓存开销。提高缓存效率。

将热门的列分别复制，如果随机放置在缓存服务器上，那么一个查询任务很容易引起表内部的数据 shuffle，因为各个列的副本很有可能不在同一台服务器上。第 2.2节显示，通常来说，热门的列引起的数据 shuffle 的量更大，第 2.3节证明，表内部的数据 shuffle 对于查询任务的执行时间的影响是不可小觑的。

以上总结告诉我们，设计方案时我们需要考虑：第一，哪些列是热门的列，需要复制多少热门的列；第二，复制以后，这些列在集群里如何放置，这涉及到“捆绑”（bundle）放置的问题。

第三章　简单方案与缺点

在本章中我们会讨论一个想法直接的简单的方案，此方案没有考虑工程实现的难度，仅考虑我们的目标。然后本章会讨论我们的现有条件，分析这个简单方案存在的缺点，不能在实际中应用的原因，为我们的实际方案提供参考。

3.1 简单方案

由第 2章我们知道，设计方案需要考虑如何决定复制多少热门的列，以及列的“捆绑”（bundle）放置问题。那么，根据之前的分析，直观上来说，想要基于列的访问热度对集群缓存系统进行列级别的负载均衡，我们的系统需要获得 SQL 查询任务具体访问的列，才方便对列的热度进行统计，并且根据此热度信息对热门的列进行复制。应用访问数据表的列的信息是由上层计算框架（如 Spark SQL）掌握，而 alluxio 是不知道的，需要计算框架提供给它，拿到这些信息之后，alluxio 进行统计，计算列的访问热度，根据热度，计算列需要拷贝的副本数，访问到来时，alluxio 根据一定的策略，返回副本中的一个（如果被复制）或者是原表。

图 3.1所示即为本章描述的简单方案的架构的设计。该架构主要有三大组件，最上层计算框架为 Spark SQL [43]（也可以更换为其他的），中间是基于 alluxio[23] 实现的列级别的负载均衡系统 CW-Cache，底层是分布式文件系统 HDFS（Hadoop File System）[5]。

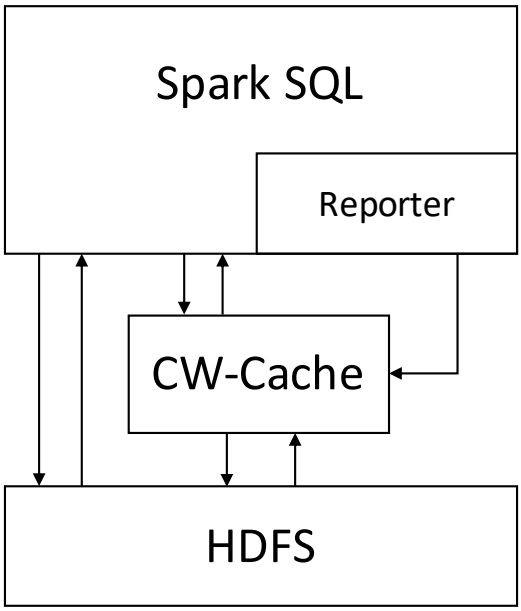


图 3.1 简单方案的架构设计。

这个列级别的集群缓存系统负载均衡方案工作流程大致如下：当用户给 Spark SQL 提交一个查询任务，经过一系列转换后，Spark SQL 得到具体要访问的列的信息，Reporter 负责

将这些信息传递给 CW-Cache，CW-Cache 记录下这些信息，并且将对应列的访问计数器加 1。在经过一段时间后，CW-Cache 对于缓存的数据表的各个列，均维持有计数器，从中获得各个列的热度（访问频率），然后它根据一定的算法计算出哪些列需要进行复制，复制多少份，哪些列需要“捆绑”在一起放置。当查询任务再此到来，CW-Cache 得到应用访问的列，CW-Cache 根据一定的策略，在缓存副本（如果有）或者原表中选择相应的列的信息返回给应用，尽可能使得负载比较分散，并且尽力避免出现 shuffle，同时更新相关列的访问计数器。以上步骤重复进行。

这个系统是根据我们的目标的很直接简单的一种思路，但是它是不实用的，这样的设计不够通用，比如图 3.1是针对 Spark SQL 进行了修改的，如果更换 SQL 引擎又需要重新实现；其次，这样的设计需要对上层计算层和中间层同时做修改，增加了二者的耦合度，不利于软件开发与维护。下面我们会分析现有条件 Parquet 和 alluxio 来解释以上原因。

3.2 现有条件

3.2.1 Parquet 文件格式

列式存储有多种格式，Parquet 是其中一种被广泛使用的，我们的方案针对 Parquet 实现，所以这里具体讨论一下 Parquet 格式。

Parquet 文件是以二进制方式存储的，因此是不能够直接读取的，Parquet 中包括该文件的元数据和数据，所以 Parquet 格式的文件是自解析的。在 Hadoop File System 文件系统和 Parquet 文件中有以下几个概念。

- HDFS 文件 (File)：一个 HDFS 的文件包括数据和元数据，数据分散地存储在多个 HDFS 块（Block）中。
- HDFS 块 (Block)：它是 HDFS 上的最小的副本（replica）单位，HDFS 会把一个 Block 存储成本地的一个文件，并且维护分散在不同的机器上的多个副本，Block 的大小可以根据需求由用户自己配置，Hadoop 早期版本默认一个 Block 大小是 128M，Hadoop 2.7.3 以及之后的版本默认一个 Block 的大小为 128M。
- 行组 (Row Group)：按照行将数据从物理上划分为多个单元，每一个行组包含一定的行数，在一个 HDFS 文件中至少存储一个行组，Parquet 读写的时候会将整个行组缓存在内存中，所以每一个行组的大小是由内存大的小决定的，例如记录占用空间比较小的 Schema 可以在每一个行组中存储更多的行。
- 列块 (Column Chunk)：在一个行组中每一列保存在一个列块中，行组中的所有列连续的存储在这个行组文件中。一个列块中的值都是相同类型的，不同的列块可能使用不同的算法进行压缩。
- 页 (Page)：每一个列块划分为多个页，一个页是最小的编码的单位，在同一个列块的不同页可能使用不同的编码方式。

一般情况下，在存储 Parquet 数据的时候会按照块（Block）大小设置行组的大小，由于一般情况下每一个 Mapper 任务处理数据的最小单位是一个 Block，这样可以把每一个行组由

一个 Mapper 任务处理，提高任务执行并行度。Parquet 文件的格式如下图 3.2所示。

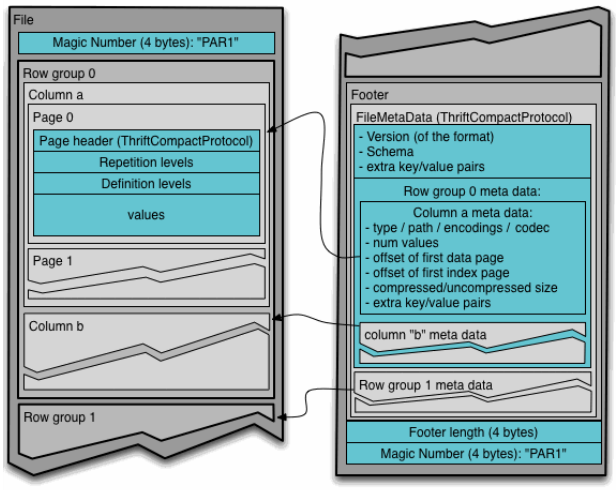


图 3.2 Parquet 文件格式。

3.2.2 Alluxio

Alluxio 原名 Tachyon，是一个基于内存的分布式文件系统，它是架构在底层分布式文件系统（如 Amazon S3、Apache HDFS 等）和上层分布式计算框架之间的一个中间件（如 Spark、MapReduce、Hbase、Flink 等），主要职责是以文件形式在内存或其它存储设施中提供数据的存取服务。在 Alluxio 出现以前，这些上层的分布式框架，往往都是直接从底层的分布式文件系统中读写数据，效率比较低，性能消耗比较大，而将 Alluxio 部署在二者之间，以文件的形式在内存中对外提供读写访问服务的话，那么 Alluxio 可以为那些大数据应用提供一个数量级的加速，而且它只要提供通用的数据访问接口，就能很方便的切换底层分布式文件系统。

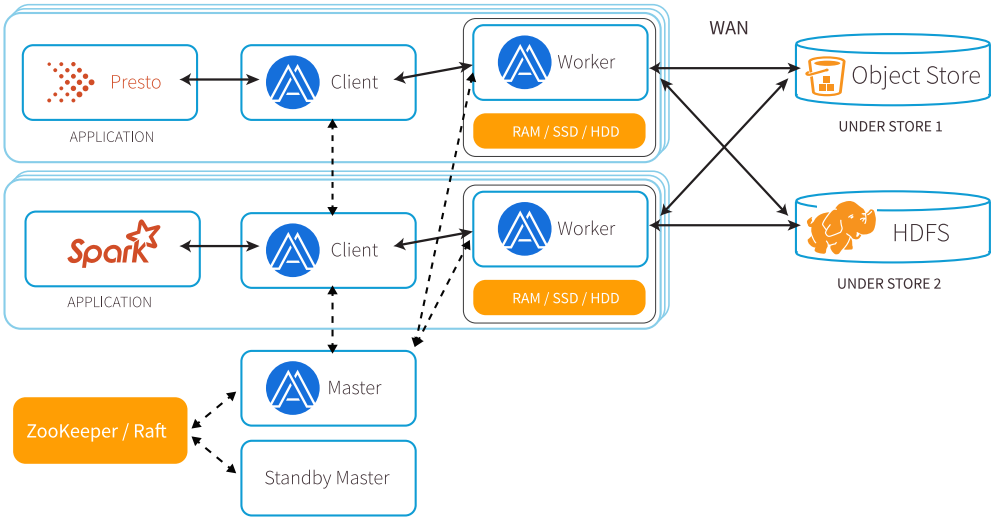


图 3.3 Alluxio 架构。

Alluxio 的架构如图 3.3所示。整体框架为主从结构，与 Hadoop[5] 等类似。主节点为 Master，负责管理全局的文件系统元数据，比如文件系统树等；从节点为 Worker，负责管理本节点数

据存储服务；Client 用于 Alluxio 与用户应用的交互，为用户提供统一的文件存取服务接口。

应用程序访问 Alluxio，先通过 Client 客户端与主节点 Master 通讯，获取对应文件的元数据，得到存储应用需要的文件的 worker，再和对应 Worker 节点通讯，进行文件存取操作。所有的 Worker 会周期性地发送心跳给 Master，维护文件系统元数据信息，确保自己被 Master 感知而仍然能在集群中正常提供服务。Master 不会主动发起与其他组件的通信，它只是以回复请求的方式与其他组件进行通信。这与 HDFS、HBase 等分布式系统设计模式是一致的。

3.2.3 分析

从上文的分析可以看出，Parquet 文件格式在存储数据表时，先将数据表按行进行“切割”出一个个行组（Row Group），行组内部再按列分成列块（Column Chunk），文件系统里物理存在的文件含有若干行组，同一列的数据物理上并没有存储在一起，并且不同的块（Block）有可能放置在不同的机器上。按照我们在本章提出的简单方案，CW-Cache 能够获得 Parquet 文件的语义信息，希望把热门的列单独提取出来，但是 Parquet 文件的文件格式决定了这个目标的实现不太容易。直观上来说，需要读取 Parquet 文件的元数据，找出需要复制的列所在的 Block，读取之后将它们拼装成一个新的文件，进行复制，这个过程的产生不可忽略的计算开销和网络通信开销，可能对系统性能造成比较大的影响，抵消负载均衡获得的性能提升。

Spark SQL 能解析基本的 SQL 语句并在分布式环境下高效执行，当 Spark SQL 解析出查询任务需要访问的列，具体的读取任务是交由 Parquet 文件格式的实现 parquet-mr[44] 来完成的，parquet-mr 会读取 Parquet 文件的 footer 获取文件的元数据，定位需要读取的列所在的文件块（block）、偏移（Offset）和读取的长度（Length）。然后它把这些信息传递给 alluxio，alluxio 将结果返回给 parquet-mr，parquet-mr 将数据解压缩（如果进行了压缩）、拼装之后交给应用进行处理，单独看这个过程，alluxio 拿到的信息是一个个“文件片段”，它们甚至不是完成的列块（column chunk），可能只是其中的一小部分，甚至是一个个字节。本章简单方案需要 alluxio 知道应用读取 Parquet 文件时的语义信息，而 alluxio 本身的架构决定了它并不支持这一点，很难将这些零碎的“文件片段”与 Parquet 存储的数据表的各个列建立关联。如果想要向 alluxio 传递文件的上下文信息，我们需要额外修改 Spark SQL 或者 parquet-mr 的相关模块，增加了上层计算框架和中间层的耦合度，同时因为只在 Spark SQL 或者 parquet-mr 进行修改，那么方案仅仅适配 Sparrk SQL 计算框架或者 Parquet 这种文件格式，没有通用性，是不好的软件设计。

此外，列的“捆绑”放置（bundling）也是非常困难的。根据 2.2和 2.3节，将数据表按照列的粒度缓存并且分开放置会在查询任务中产生表内部的 shuffle，且 shuffle 带来的网络通信开销会对查询任务的执行时间带来不可忽视的影响。 2.1节的实验结果表明在列的热度存在差异的基础上，不同热度的列相互之间被共同访问的概率也不一样。于是我们想到可以借助“捆绑”放置（bundling）来减少甚至避免同一张数据表内的数据 shuffle。然而这个问题是难以解决的：首先如何得到列的共同访问的模式是困难的。如果一张数据表有 N 列，那么列的共同访问的模式理论上有 2^N 种，在生产环境中是没有足够的资源来同时满足这么多共同访问模式。其次，从系统上来说，alluxio 作为通用的内存分布式文件系统，需要为多用户提供服务，只要用户通过 alluxio 的接口存取文件，alluxio 便执行相应的操作，将结果返回即可。alluxio 并不知晓每一次的请求来自哪个客户端（Client），它不会维护状态，区分不同客户端的请求。换句话说，不做修改的情况下，alluxio 无法得知哪些读访问请求来自同一个客户端

的同一个查询任务，那么列的“捆绑”（bundling）放置也就无从谈起。而如果要使得 alluxio 维护状态信息，首先需要客户端发送请求时附带自己的身份信息，同时 alluxio 的 master 需要记录并且进行匹配，大大增加系统的网络、存储、计算开销。

3.2.4 总结

本章主要讨论了一个基于我们的列级别的负载均衡的集群缓存系统的目标而提出的简单直接的方案，它需要上层计算框架将访问的列的信息发给中间层，中间层用以统计各个列的访问热度并且按照一定的策略复制，在请求到来时选择合适的副本传给应用。这个方案有三点缺陷。

1) 因为 Parquet 文件格式是先按照行进行划分，然后再按列进行存储，如果要按照需求把某一列单独抽取出来进行复制，需要读取文件元数据、读取存储该列的各个 Block，然后拼装成新的 Parrquet 文件存放到其他机器，这一系列的操作开销较大；

2) Spark SQL 读取 Parquet 文件时，调用 parquet-mr，传给 alluxio 的信息仅有所需的文件 URI（统一资源标识符）、偏移量和读取长度，并未包含文件的语义信息。想要传递文件上下文信息需要额外对 Spark SQL 或者 Parquet-mr 做修改，增加了软件之间的耦合度；

3) Alluxio 作为通用的内存文件系统，为多用户服务，不保存状态信息，不去识别请求来自哪个 Client，这意味着 alluxio 难以知晓哪些列的访问是来自于同一个 Client 的同一个查询任务，不便于对这些列的缓存作“捆绑”放置（bundle）。如果访问文件时增加 Client 的身份标识，会大大增加通信开销，同时增加 Master 维持状态信息的存储、计算开销。

综上所述，本章描述的这个简单方案不具备实用条件，并且难以实现，但对于其缺陷的分析有助于我们更加深刻的理解问题，在现有条件的基础上调整系统设计的方向。在第 4章中我们会介绍实际 CW-Cache 的系统设计与分析。

第四章　CW-Cache：设计与分析

在本章我们会对我们的目标问题进行数学建模，

致　谢

这次的毕业论文设计总结是在我的指导老师 xxx 老师亲切关怀和悉心指导下完成的。从毕业设计选题到设计完成，x 老师给予了我耐心指导与细心关怀，有了莫老师耐心指导与细心关怀我才不会在设计的过程中迷失方向，失去前进动力。x 老师有严肃的科学态度，严谨的治学精神和精益求精的工作作风，这些都是我所需要学习的，感谢 x 老师给予了我这样一个学习机会，谢谢!

感谢与我并肩作战的舍友与同学们，感谢关心我支持我的朋友们，感谢学校领导、老师们，感谢你们给予我的帮助与关怀; 感谢肇庆学院，特别感谢计算机科学与软件学院四年来为我提供的良好学习环境，谢谢!

参考文献

[1] Botta A, De Donato W, Persico V, et al. Integration of cloud computing and internet of things: a survey[J]. Future generation computer systems, 2016, 56:684–700.

[2] Oussous A, Benjelloun F Z, Lahcen A A, et al. Big data technologies: A survey[J]. Journal of King Saud University-Computer and Information Sciences, 2018, 30(4):431–448.

[3] Zaharia M. An Architecture for Fast and General Data Processing on Large Clusters. USA: ACM Books, 2016.

[4] Ghemawat S, Gobioff H, Leung S T. The google file system. Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, 2003. 20–43.

[5] Apache hadoop. `https://hadoop.apache.org/`.

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

[7] Malewicz G, Austern M H, Bik A J, et al. Pregel: a system for large-scale graph processing. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010. 135–146.

[8] Singh A, Ong J, Agarwal A, et al. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. Proc. ACM SIGCOMM, 2015.

[9] Huawei. NUWA. `https://www.youtube.com/watch?v=OsmZBRB_OSw`.

[10] Asanovic K, Patterson D. FireBox: A hardware building block for 2020 warehouse-scale computers. USENIX FAST, 2014.

[11] Alistarh D, Ballani H, Costa P, et al. A high-radix, low-latency optical switch for data centers[J]. SIGCOMM Comput. Commun. Rev., 2015, 45(4):367–368.

[12] Scott C. Latency trends. `http://colin-scott.github.io/blog/2012/12/24/latency-trends/`.

[13] IEEE. Ieee p802.3ba 40 gbps and 100 gbps ethernet task force. `http://www.ieee802.org/3/ba/`.

[14] Han S, Egi N, Panda A, et al. Network support for resource disaggregation in next-generation datacenters. ACM HotNets, 2013.

[15] Gao P X, Narayan A, Karandikar S, et al. Network requirements for resource disaggregation. Proc. USENIX OSDI, 2016.

[16] Ananthanarayanan G, Ghodsi A, Shenker S, et al. Disk-locality in datacenter computing considered irrelevant. ACM HotOS, 2011.

[17] Jonas E, Venkataraman S, Stoica I, et al. Occupy the cloud: Distributed computing for the 99%. Proc. ACM SoCC, 2017.

[18] Amazon s3. https://aws.amazon.com/s3.

[19] Windows azure storage. https://goo.gl/RqVNmB.

[20] Openstack swift. https://www.swiftstack.com.

[21] Shvachko K, Kuang H, Radia S, et al. The Hadoop distributed file system. Proc. IEEE Symp. Mass Storage Syst. and Technologies, 2010, 2010.

[22] Rashmi K, Chowdhury M, Kosaian J, et al. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016. 401–417.

[23] Alluxio. http://www.alluxio.org/.

[24] Memcached. https://memcached.org/.

[25] Redis. https://redis.io/.

[26] Li H, Ghodsi A, Zaharia M, et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks. Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014. 1–15.

[27] Ananthanarayanan G, Agarwal S, Kandula S, et al. Scarlett: coping with skewed content popularity in mapreduce clusters. Proceedings of the sixth conference on Computer systems. ACM, 2011. 287–300.

[28] Ananthanarayanan G, Ghodsi A, Warfield A, et al. Pacman: Coordinated memory caching for parallel jobs. Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), 2012. 267–280.

[29] Yahoo! webscope dataset. https://goo.gl/6CZZCF.

[30] Kandula S, Sengupta S, Greenberg A, et al. The nature of data center traffic: measurements & analysis. Proc. ACM IMC, 2009.

[31] Chowdhury M, Kandula S, Stoica I. Leveraging endpoint flexibility in data-intensive clusters. Proc. ACM SIGCOMM, 2013.

[32] Greenberg A, Hamilton J R, Jain N, et al. VL2: a scalable and flexible data center network. Proc. ACM SIGCOMM, 2009.

[33] Yu Y, Huang R, Wang W, et al. Sp-cache: Load-balanced, redundancy-free cluster caching with selective partition. Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, Piscataway, NJ, USA: IEEE Press, 2018. 1:1–1:13.

[34] Presto. https://prestodb.github.io/.

[35] Power R, Li J. Piccolo: Building fast, distributed programs with partitioned tables. OSDI, volume 10, 2010. 1–14.

[36] Memsql. https://www.memsql.com/.

[37] Hong Y J, Thottethodi M. Understanding and mitigating the impact of load imbalance in the memory caching tier. Proc. ACM SoCC, 2013.

[38] Huang Q, Gudmundsdottir H, Vigfusson Y, et al. Characterizing load imbalance in real-world networked caches. ACM HotNets, 2014.

[39] Huang C, Simitci H, Xu Y, et al. Erasure coding in windows azure storage. Proc. USENIX ATC, 2012.

[40] Sathiamoorthy M, Asteris M, Papailiopoulos D, et al. Xoring elephants: Novel erasure codes for big data. Proc. VLDB Endowment, volume 6, 2013. 325–336.

[41] apache parquet. `https://parquet.apache.org/`, 2019.

[42] Armbrust M, Xin R S, Lian C, et al. Spark sql: Relational data processing in spark. Proceedings of the 2015 ACM SIGMOD international conference on management of data. ACM, 2015. 1383–1394.

[43] Apache spark sql. `https://spark.apache.org/sql/`.

[44] Parquet mr. `https://github.com/apache/parquet-mr`.

附录 A 1

hello1

附录 B　2

hello2