

東南大學

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

起止日期 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。

论文作者签名：_____ 日期：_____ 年 _____ 月 _____ 日

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。

论文作者签名：_____ 导师签名：_____
日期：_____ 年 _____ 月 _____ 日 日期：_____ 年 _____ 月 _____ 日

摘要

目前越来越多的数据密集型集群部署内存计算方案来提高 I/O 性能，在集群内存使用缓存是一种普遍做法。然而负载不均这一集群缓存中常见的问题则会损害缓存带来的好处。学术界应对这一问题的方法包括将热门的文件复制多份、用存储编码为文件创建同等分区、选择性地将文件分割成多份来分散存储。然而这些方法均关注一般意义上的文件的负载均衡，而本文关注的是结构化数据的负载均衡，根据数据表中各个列的访问热度倾斜实现列级别的负载均衡。我们的解决方案称为 CW-Cache，将数据表中访问热度排在前 K 的列一起复制 r 份缓存在集群中。我们通过数学建模，在数据表各个列的热度确定的情况下，高效地计算出最优的 K 和 r ——二者太小不足以实现负载均衡，太大则会增大内存开销。我们在分布式内存文件系统 Alluxio 上实现了 CW-Cache，本文最后提出了 CW-Cache 目前存在的问题，为下一步研究提出方向。

关键词：集群缓存，列级别，负载均衡，结构化数据

Abstract

Nowadays, more and more data-intensive clusters employ in-memory solutions to improve I/O performance and a common approach is using cache in cluster memories. Nevertheless, routinely observed load imbalance will degrade the benefits of caching. Ways the academic deal with this problem include copying multiple replicas of hot files, creating parity chunks using storage codes and selectively partitioning files and caching them across the clusters. Yet, they focus on load balance of general files, while this paper cares about load balance of structured data. We aimed to achieve column-wise load balance based on the skewed popularities of columns. Our solution, called CW-Cache, bundle columns with their popularities in the top K list with a replicative factor of r . Through modeling, we effectively calculate the optimal K and r given popularities of each column – too small leads to load imbalance, while too large results in high memory costs. We implemented CW-Cache atop Alluxio, a popular in-memory distributed storage for data-intensive clusters. At last, this paper points out several problems of CW-Cache and gives directions of future research.

KEY WORDS: Cluster Cache, Column-wise, Load Balancing, Structured Data

目 录

摘要	I
Abstract	II
第一章 前言	1
1.1. 课题背景	1
1.1.1. 大数据	1
1.1.2. 集群缓存	2
1.1.3. 负载均衡	3
1.2. 研究现状	4
1.2.1. 选择复制	5
1.2.2. 纠删码	5
1.2.3. 选择分割	5
1.2.4. 更细粒度负载均衡	6
1.3. 研究的目的与内容	6
1.4. 论文结构	7
第二章 研究动机	8
2.1. 列的访问规律	9
2.1.1. 实验设置	9
2.1.2. 文件内列访问频率偏差	9
2.1.3. 热门的列被共同访问的规律	10
2.2. 列之间的数据 shuffle	10
2.2.1. 实验设置	11
2.2.2. 每一列的数据 shuffle	13
2.3. 数据 shuffle 的影响	13
2.3.1. 实验设置	13
2.3.2. 度量指标	14
2.3.3. 不同网络带宽下的 shuffle 开销	14
2.4. 总结	14
第三章 Column-aware 方案与缺点	16
3.1. Column-aware 方案	16
3.2. 现有条件	17
3.2.1. Parquet 文件格式	17
3.2.2. Alluxio	18
3.2.3. 分析	19
3.3. 总结	20

第四章 CW-Cache: 设计与分析	22
4.1. 问题建模	22
4.1.1. 符号定义	22
4.1.2. 目标	23
4.2. 算法	23
4.3. 通过列的热门度估算共同访问模式	25
4.3.1. 符号定义	25
4.3.2. 推测	26
4.3.3. 分析	26
第五章 实现	30
5.1. 架构概览	30
5.2. 实现细节分析	31
5.2.1. Reporter	31
5.2.2. Recorder	32
5.2.3. Replication Manager	33
5.2.4. 周期性检查与复制	34
5.3. 系统额外开销	34
第六章 测试与评估	36
6.1. 实验方法	36
6.1.1. 实验环境设置	36
6.1.2. 负载	36
6.1.3. 基准	36
6.2. 实验结果	37
6.2.1. 读取延迟	37
6.2.2. 负载	37
第七章 总结与展望	39
7.1. 工作总结	39
7.2. 未来展望	39
参考文献	41
附录 A Bundle-K 方案核心代码	44
致谢	49

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

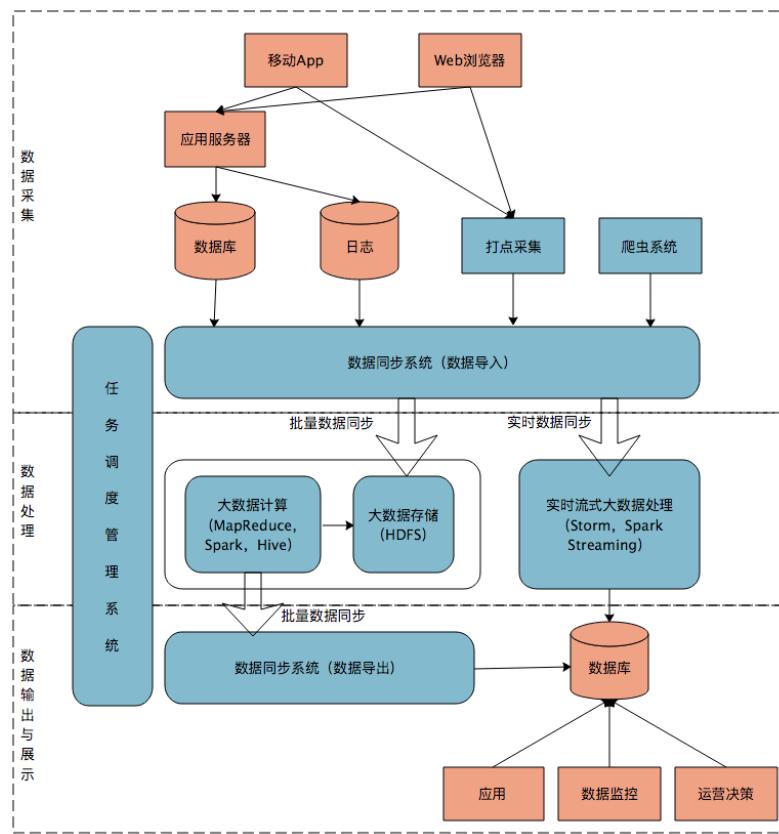


图 1.1 典型的互联网大数据平台架构。

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

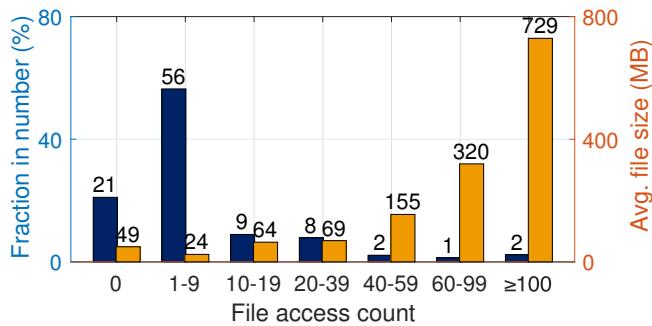


图 1.2 Yahoo! 集群上观察到的文件热门程度（蓝色）和文件大小（橙色）的分布^[29]。

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、Apache 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（~ 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，

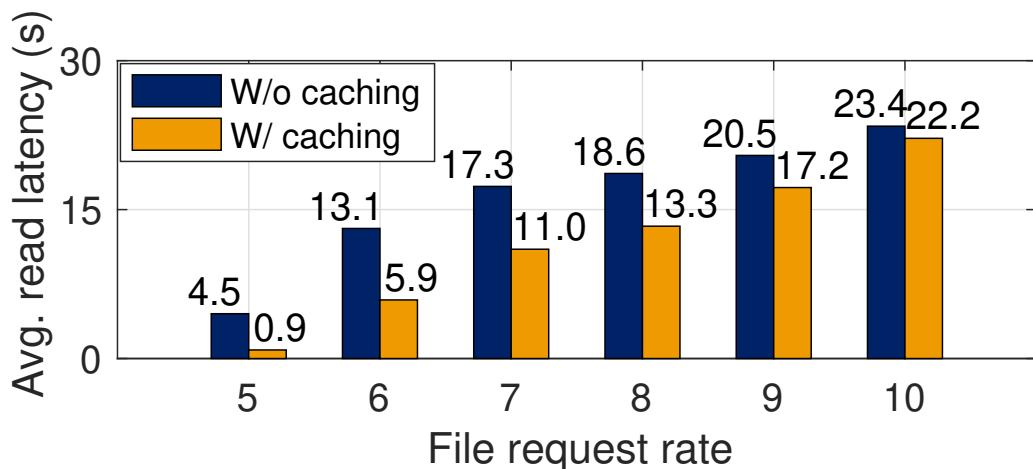


图 1.3 有/无缓存的情况下平均读延迟随着负载增加而增加。

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

本文一共七章，下面分别介绍每一章的主要内容。

第一章是前言，主要介绍了本研究课题的背景，集群缓存与负载均衡的现状，接着对当前最新的研究进展做了简介，最后提及此课题的研究目的和主要内容。

第二章是研究动机，用三个实验分别探究列的访问热度倾斜规律，数据表内部的 shuffling 和 shuffling 的影响，为课题进行提供基础。

第三章是 Column-aware 方案与缺点，介绍了一个易想到的直接的方案，然后分析了我们现有的条件，指出此方案的确定和不实用，为我们的正式方案提供参考。

第四章是 CW-Cache: 设计与分析，这一章我们提出采用 Bundle-K 方案的 CW-Cache 系统，并进行数学建模与仿真实验，论证以列的热度信息估算列的共同访问模式，最终求出最优的复制的列数 K 和复制的份数 r 的可行性。

第五章是实现，介绍了 CW-Cache 系统的框架，各部分的细节以及系统的额外开销。

第六章测试与评估，

第七章是总结与展望，此章总结了本项目完成的不足，提出 CW-Cache 存在的缺陷，指出未来的研究工作应该改进完善的地方。

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。

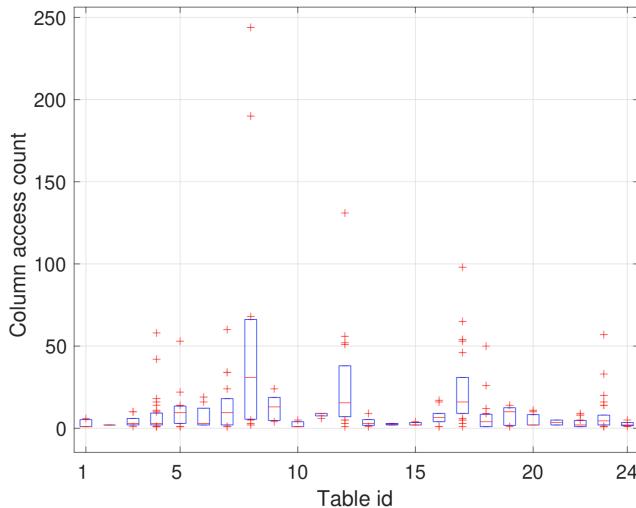


图 2.1 TPC-DS 标准测试程序中所有数据表的列访问计数的分布。箱形图展示了每张数据表里 25th 分位，中位数和 75th 分位的列访问次数计数。红色标记表示离群值。

2.1 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。

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 GB¹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

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

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。

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。

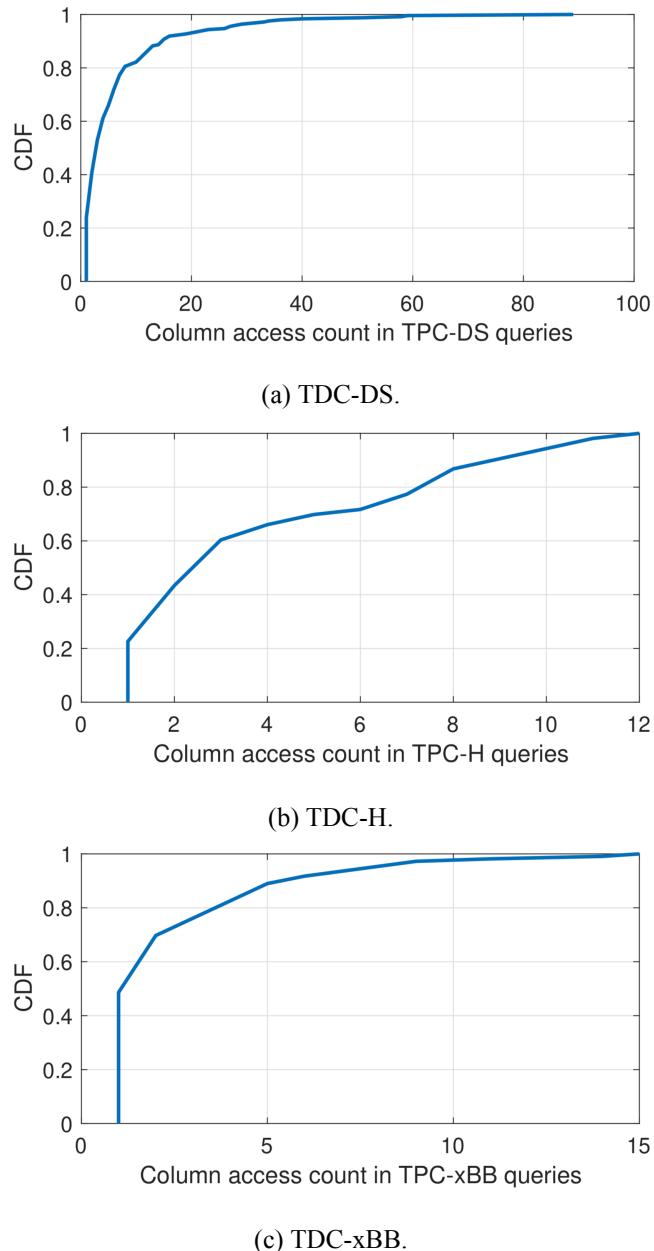


图 2.2 三种标准测试程序中列访问计数的 CDF。

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。

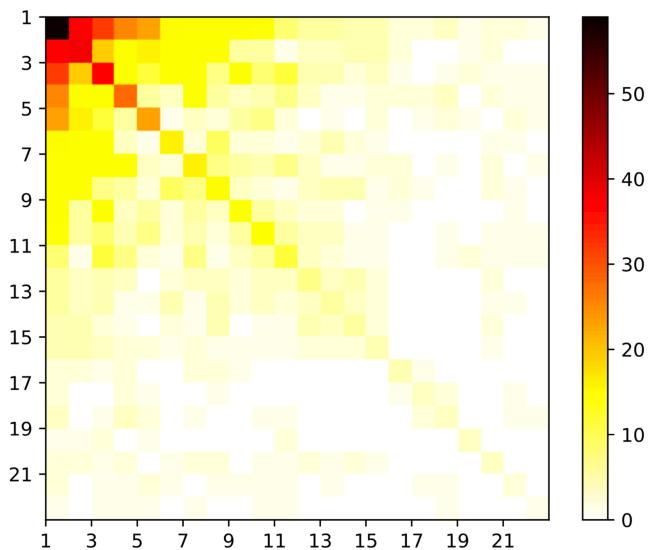
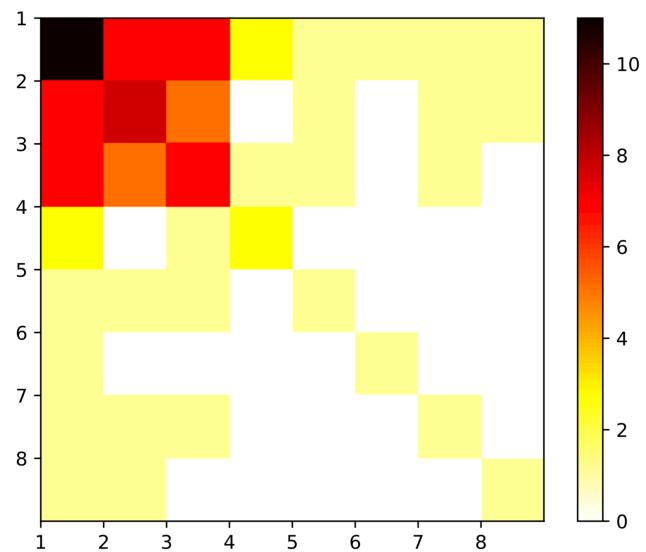
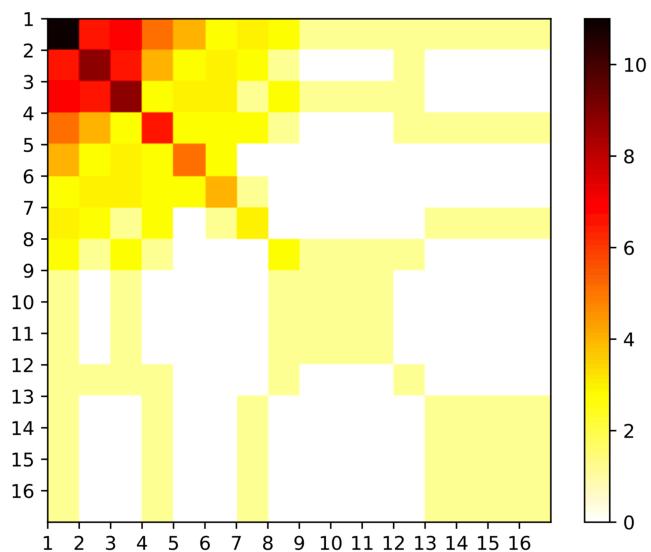
(a) The *store_sales* table in TDC-DS.(b) The *orders* table in TDC-H.(c) The *store_sales* table in TDC-xBB.

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

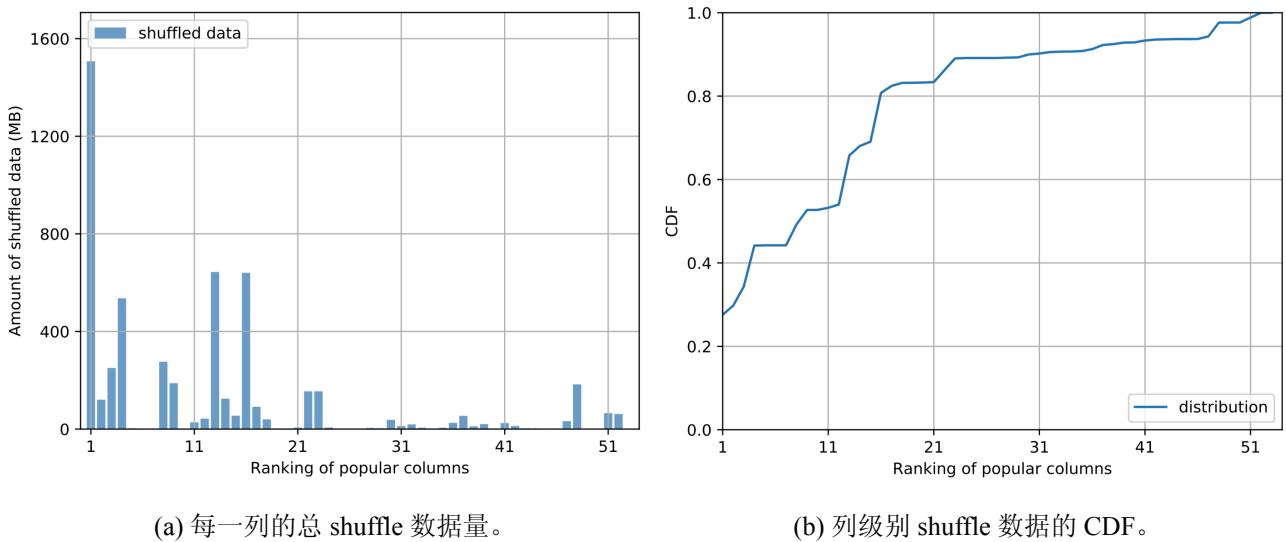


图 2.4 TPC-H 标准测试程序的列级别数据 shuffle。

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。

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：

$$\text{Slowdown} = \frac{L_S - L_N}{L_N}, \quad (2.1)$$

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，*slowdown* 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。

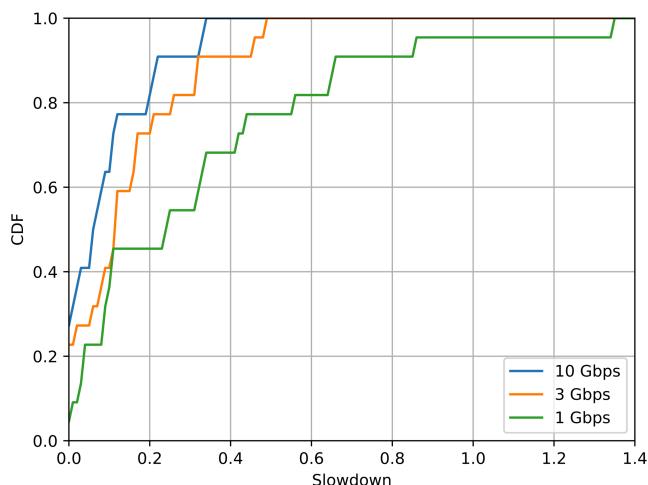


图 2.5 *slowdown* 在不同网络带宽下的分布。

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 Column-aware 方案与缺点

在本章中我们会讨论一个容易想到的直接的方案，取名 Column-aware 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，分析这个方案存在的缺点，不能在实际中应用的原因，为我们的第 4 章提出的 CW-Cache 方案提供参考。

3.1 Column-aware 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 Column-aware 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]。

这个采用 Column-aware 方案的负载均衡系统 CW-Cache 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；

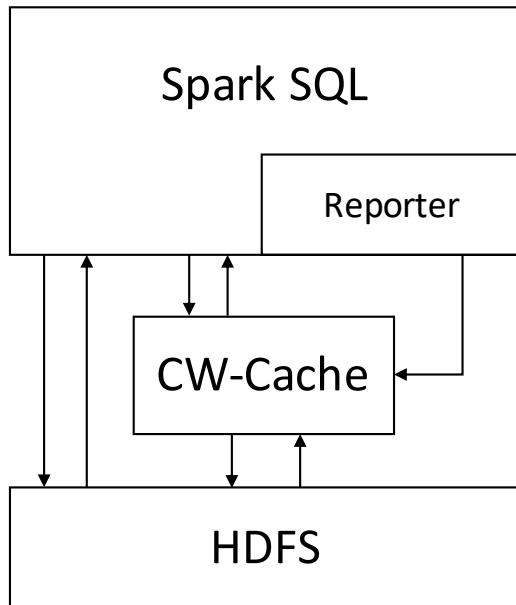


图 3.1 Column-aware 方案的架构设计。

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具有代表性的列式存储的文件格式，我们的方案针对 Parquet 实现，所以在这里我们具体讨论一下 Parquet 格式。

Parquet 文件是以二进制方式存储的，因此不能够像文本文件一样直接读取，Parquet 中包括该文件的元数据（metadata）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 文件包括数据和元数据，数据分散地存储在若干个 HDFS 块 (Block) 中。
- **HDFS 块 (Block):** HDFS 块 (Block) 是 HDFS 上最小的副本 (replica) 单位，HDFS 会把一个 Block 作为一个文件存储在本地，并且维护分散在不同的机器上的多个副本（默认每个文件 3 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):** Parquet 按照行 (Row) 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，每一个 HDFS 文件至少存储一个行组，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容量决定的，也就是说记录（Record）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，同一列储存在一个列块中，行组中的所有列依次连续地存储在这个行组中。同一个列块中的值的类型是相同的，不同的列块可能使用不同的压缩算法进行压缩。
- **页 (Page):** 每一个列块划分为多个页，页是最小的编码的单位，同一个列块中的不同页可能使用不同的编码方式。

一般情况下，在存储 Parquet 数据的时候会根据下层文件系统的块（Block）大小来设置行组的大小，在 MapReduce 计算框架中，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块（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。

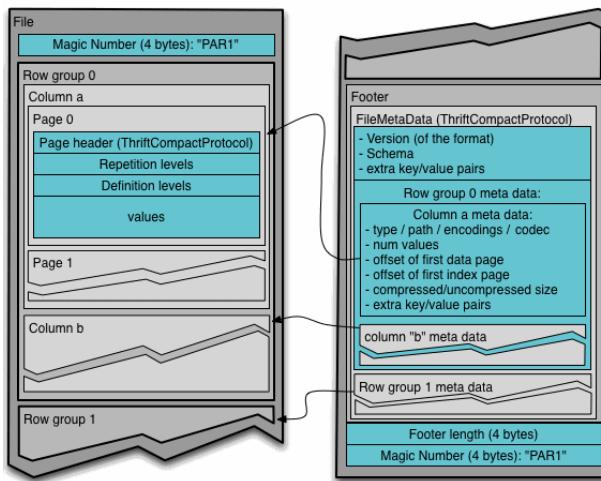


图 3.2 Parquet 文件格式。

3.2.2 Alluxio

Alluxio 原名 Tachyon，是一个基于内存的分布式文件系统，它是架构在底层的分布式文件系统（如 Amazon S3、Apache HDFS 等）和上层分布式计算框架（如 Spark、MapReduce、Hbase、Flink 等）之间的中间层，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，而如果将 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，那么 Alluxio 可以为这些大数据应用提供一个数量级的加速，而且它提供通用的数据访问接口，所以能够很方便地切换底层的分布式文件系统。

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，

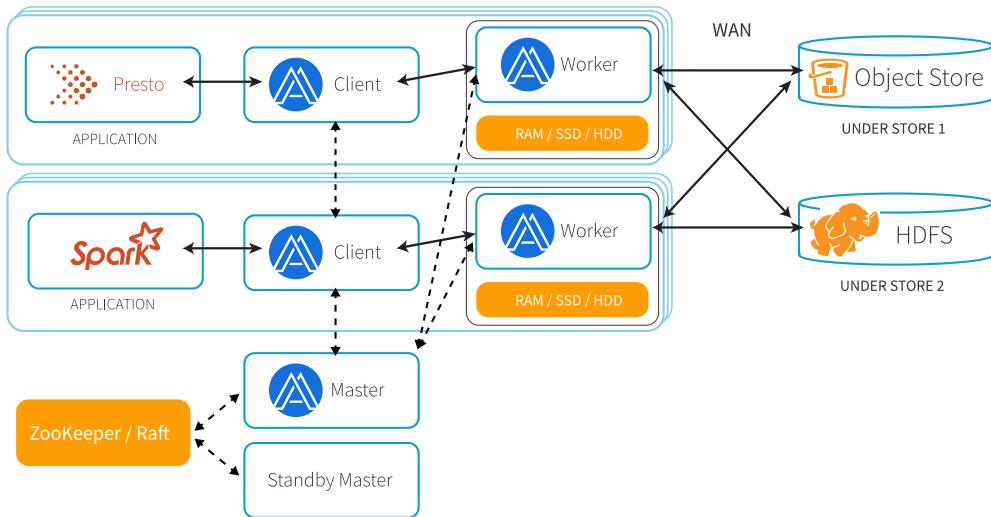


图 3.3 Alluxio 架构。

负责管理全局的文件系统的元数据，比如文件系统树等；从节点运行 Worker 进程，负责管理本节点的数据存储服务；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。

当应用程序需要访问 Alluxio 中存储的文件，先通过 Client 客户端与主节点 Master 通讯，待 Master 返回存储有应用需要的文件的 worker 列表，再根据一定的策略和对应 Worker 节点通讯，进行文件的存取操作。所有的 Worker 会周期性地发送心跳消息给 Master，维护文件系统的元数据信息，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 Column-aware 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，但是考虑到 Parquet 文件的文件格式，我们认为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”，它们甚至不是完整的列块（column chunk），可能只是其中的一小部分，有时为了高效读取数据，parquet-mr 甚至并行地一个一个字节读取文件。本章 Column-aware 方案需要 alluxio 获得应用读取 Parquet 文件时的关于列的语义信息，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 Spark 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，将数据表以列为单位分开缓存且分开放置会在查询任务中引起数据表内部的 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 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），Master 端不会维护状态，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。

3.3 总结

本章主要讨论了一个不考虑系统实现，只针对列级别负载均衡目标的集群缓存系统 Column-aware 方案，它需要上层计算框架将访问的列的信息发送给中间层 CW-Cache，CW-Cache 借此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提取出来进行复制，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 Parquet 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，为多用户提供服务，不保存状态信息，不去识别请求来自哪个 Client，这意味着 alluxio 难以知晓对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，本章描述的采用 Column-aware 方案的系统开发难度较大，结构设计不合理，实用性相对较差，但对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 章中我们会介绍本项目中 CW-Cache 系统实际采用的 Bundle-K 方案的设计与分析。

第四章 CW-Cache：设计与分析

根据第 2 章的实验和第 3 章对 Column-aware 方案的分析，我们了解到方案的一些需求：

- 方案需要根据数据表中各个列的热度（访问频率）决定哪一些列需要被复制，复制多少份。
- 如果复制后的列的副本分开存储，执行 SQL 查询任务时会引起 shuffling，且 shuffling 会对任务执行时间产生不可小觑的影响，我们设计的方案需要尽力避免。
- 方案应该尽可能减少上层计算框架和中间缓存系统的耦合度，缓存系统利用的额外信息要尽可能少。

我们给系统取名为 CW-Cache，CW 是 Column-wise 两个单词的首字母，取自它实现列级别负载均衡的目标。吸取 Column-wise 方案的经验教训，我们提出了另一个用于 CW-Cache 系统的方案，我们称这个方案为“Bundle-K”，正如字面意思所示，我们经过统计后将数据表中的各个列按照热度（访问频率）进行排序，然后“捆绑”前 K 个列进行复制。在执行查询任务的时候，如果查询任务涉及的所有列均在这复制出来的前 K 个列中，那么就由副本为查询任务提供数据，否则就由原表为任务提供数据。那么问题来了，这个 K 取多少合适呢？即使确定了 K ，那么这前 K 列要复制多少份比较合适呢？这并不是拍脑袋就能想出来的，接下来我会对此方案进行数学建模与分析，用公式进行推导。

4.1 问题建模

4.1.1 符号定义

这里我们对一张数据表内的列级别的共同访问模式进行描述。假设这张表共有 n 列，记为 $\{c_0, c_1, \dots, c_{n-1}\}$ ，它们所占空间的归一化分别是 $\{s_0, s_1, \dots, s_{n-1}\}$ ，则 $\sum_{i=0}^{n-1} s_i = 1$ ，一些查询任务会访问一组特定的列，我们称这一组列是一个共同访问模式。假设有 m 个共同访问模式 $T = \{t_0, t_1, \dots, t_{m-1}\}$ ，对于某一个特定的访问模式 t_i ，它的热度 p_i 代表了有这个访问模式的查询任务的数量， ts_i 是这个访问模式中所有列的大小之和。这 m 组共同访问模式的负载分别是 $\{l_0, l_1, \dots, l_{m-1}\}$ ，其中 $l_i = p_i ts_i$ 。

记 k 为复制的最热门的列的数量， S_k 是这复制的 k 列组成的副本能够覆盖的访问模式的集合，那么所有的访问模式就被分成了两组， S_k 和 $T - S_k$ 。记 L_h 和 L_c 分别是 S_k 和 $T - S_k$ 承担的总负载，则有 $L_h = \sum_{i \in S_k} l_i$ 以及 $L_c = \sum_{i \notin S_k} l_i$ 。

假设我们的集群有 N 台服务器，我们的目标是找到 k 和副本的数量 r 来最小化任意一台服务器的负载的方差和复制的代价。

4.1.2 目标

记 X 是任意一台服务器的负载，那么有：

$$X = a_0 \frac{L_h}{r} + a_1 L_c, \quad (4.1)$$

其中 a_0 和 a_1 是二元随机变量，表示一个副本/原表是否被放置在这台机器上。由于我们将副本和原表随机放置在集群中， a_0 和 a_1 服从伯努利分布且相互独立，因此我们能够得出：

$$\begin{aligned} \text{Var}(X) &= \frac{L_h^2}{r^2} \frac{r}{N} \left(1 - \frac{r}{N}\right) + L_c^2 \frac{1}{N} \left(1 - \frac{1}{N}\right) \\ &= \frac{1}{N} \left(\frac{L_h^2}{r} + L_c^2 \right) - \frac{1}{N^2} \left(L_h^2 + L_c^2 \right) \end{aligned} \quad (4.2)$$

记 C 为复制的代价，即缓存这些副本占用的内存空间，我们有：

$$C = r \sum_{i=0}^{k-1} s_i \quad (4.3)$$

我们的目标是使得负载的方差与复制的代价的加权和最小化。假设权重为 w ，那么：

$$\begin{aligned} \min \quad & \frac{1}{N} \left(\frac{L_h^2}{r} + L_c^2 \right) - \frac{1}{N^2} \left(L_h^2 + L_c^2 \right) + w \times r \sum_{i=0}^{k-1} s_i \\ \text{s.t.} \quad & k \in \{1, \dots, n\} \\ & r \in \{1, \dots, N\} \end{aligned} \quad (4.4)$$

4.2 算法

我们可以通过遍历所有可能的值，即从 1 到 n ，来寻找最佳的 k 。具体来说，对于每一个可能的 k 值，我们更新 S_k 来计算 L_h 和 L_c ，接着我们计算当前的 k 值下最优的 r 值。用 $f(r)$ 表示目标函数，我们能够发现：

$$f'(r) = -\frac{L_h^2}{Nr^2} + w \sum_{i=0}^{k-1} s_i \quad (4.5)$$

算法 1 Find optimal k and r

- T : 所有的共同访问模式
- $l[0 \cdots m - 1]$: m 个访问模式的负载
- w : 最小化目标函数（等式 4.4）中的权重

```

1: function UpdateSk( $k, S_k, O_k$ ) ▷ update  $S_k$ 
2:   for all  $t \in O_k$  do
3:     if  $k$  hottest columns contain  $t$  then
4:       move  $t$  from  $O_k$  to  $S_k$ 
5:     end if
6:   end for
7: end function
8: function FindOpt
9:    $S_k \leftarrow \{\}$  ▷ 初始化  $S_k$ 
10:   $O_k \leftarrow T$  ▷ 初始化  $T - S_k$ 
11:   $opt\_k \leftarrow -1$  ▷ 初始化最优值  $k$ 
12:   $opt\_r \leftarrow -1$  ▷ 初始化最优值  $r$ 
13:   $opt\_obj \leftarrow +\infty$  ▷ 初始化最优目标
14:  for all  $k \in \{1, \dots, n\}$  do
15:    UpdateSk( $k, S_k, O_k$ )
16:     $L_h \leftarrow \sum_{i \in S_k} l_i$ 
17:     $L_c \leftarrow \sum_{i \notin S_k} l_i$ 
18:     $r \leftarrow$  get integer  $r$  using equation 4.6
19:     $r \leftarrow \min\{N, r\}$  ▷  $r \leq N$ 
20:     $obj \leftarrow$  calculate objective in equation 4.4
21:    if  $obj < opt\_obj$  then ▷ 更新  $k, r, obj$ 
22:       $opt\_k \leftarrow k$ 
23:       $opt\_r \leftarrow r$ 
24:       $opt\_obj \leftarrow obj$ 
25:    end if
26:  end for
27:  return  $opt\_k, opt\_r$ 
28: end function

```

当 $f'(r) = 0$, 可以得到:

$$r = \sqrt{\frac{L_h^2}{Nw \sum_{i=0}^{k-1} s_i}}, \quad (4.6)$$

在这个例子中, 我们的目标函数 $f(r)$ 能够求到最小值。

算法 1 展示了求解最优的 k 和 r 的过程。时间复杂度为 $O(n + m)$, 其中 n 是列的数量而 m 是访问模式的数量。

算法 2 Update S_k for sorted T

```

1: function UpdateSk( $k, S_k, O_k$ )
2:   for all  $t \in O_k$  do
3:     if  $k$  hottest columns contain  $t$  then
4:       move  $t$  from  $O_k$  to  $S_k$ 
5:     else
6:       return
7:     end if
8:   end for
9: end function

```

算法 1中最耗时的步骤是对每一个 k 值更新 S_k ，把所有的访问模式 T 排个序是可选的降低开销的方法。具体来说，对于访问模式 t_i ，记 e_i 是其热门度最低的列，我们把 T 中的 m 个访问模式按照它们热门度最低的列的热门度进行排序 $\{e_0, e_1, \dots, e_{m-1}\}$ 。这样，随着 k 的增加，排序后的 T 中的访问模式就会依次被覆盖。

算法 2展示了当 T 被排好序后，更新 S_k 的过程。基于排序后的 T 来搜索最佳的 k 和 r ，时间复杂度是 $O(n + m)$ ，假设此排序过程的时间复杂度是 $O(m \log m)$ （例如快速排序），那么总的时间复杂度是 $O(m \log m + n)$ 。

4.3 通过列的热门度估算共同访问模式

上一节我们的建模引入了列的共同访问模式，对于一张表来说，如果这张表有 N 列，理论上它能产生 2^N 种访问模式，当 N 比较大的时候，访问模式的数量是我们无法接受的。要想获得访问模式，一种方法是 Master 在内存中维护每一种出现过的访问模式的计数器，即统计它们的热度信息。问题是对于列比较多的表，要想维持这么多计数器是非常消耗内存资源的，内存资源本就紧缺，应该用在刀刃上。于是我们设想，能否利用列的热度信息来推测访问模式的热度？下面我们进行数学建模并加以实验验证。

4.3.1 符号定义

对于某张数据表，假设它有 n 个列 $\{c_1, c_2, \dots, c_n\}$ ，它们的归一化的体积为 $\{s_1, s_2, \dots, s_n\}$ ，那么 $\sum_{i=1}^n s_i = 1$ 。假设现在有 m 个查询任务访问这些列。这些列的热度记为 $P = \{p_1, p_2, \dots, p_n\}$ ，它们的负载是 $L = \{l_1, l_2, \dots, l_n\}$ ，其中 $l_i = p_i s_i$ 。用 L_a 来表示负载的总和，那么我们有 $L_a = \sum_{i=1}^n l_i$ 。

在 4.1 节中，我们假设确定复制前 k 个最热门的列，然后计算呢它们能覆盖的访问模式，并把它们的负载累加起来得到 L_h 。在本节中，我们放弃详细的访问模式的信息，仅仅通过列

的热度信息来估计 L_h 。

4.3.2 推测

我们可以根据列的热度随机生成查询任务来估算访问模式的分布。对于一个查询任务，它访问列 c_i 的可能性是 $\frac{p_i}{m}$ 。假设查询任务是否访问每一个列是相互独立的，那么 n 个列被一个查询任务访问的概率是 $B = \left\{ \frac{p_1}{m}, \frac{p_2}{m}, \dots, \frac{p_n}{m} \right\}$ 。

为了估计访问模式的分布，首先我们给每一个列分配等同于它们热度的“配额”，接着我们按照 B 中的比例生成查询任务，直到所有配额用尽。基于这些生成的查询任务和访问模式，我们能够研究 L_h 和 k 之间的关系。

测量实验 1 首先我们通过 TPC-DS 标准测试程序测试了上文提出的方法。对于每一张表，我们统计每个列的热度以及访问过该表的查询任务的数量。接着我们生成查询任务，通过蒙特卡罗方法估计 k 取不同的值下的 L_h 。为了简化问题，我们假设所有的列的大小相等。

图 4.1 展示了 TPC-DS 中三张具有代表性的数据表中列的热度及归一化的 L_h ，即 $\frac{L_h}{L_a}$ 和 k 之间的关系。为估算 L_h ，我们对比了蒙特卡罗方法（Monte Carlo method(MC)）和实际情况（ground truth(GT)）。从图 4.1 上可以看出，两种方法对应的曲线吻合程度是比较高的，也就是说，就 L_h 和 k 之间的关系而言，我们可以通过随机生成查询任务这种方法来估算访问模式的分布。

4.3.3 分析

为分析这个方法，我们简单地假设共有 m 个查询任务，每一个有 $B = \left\{ \frac{p_1}{m}, \frac{p_2}{m}, \dots, \frac{p_n}{m} \right\}$ 的概率访问 n 个列。我们的目标是给定 k ，估算 L_h 。

考虑一个特定的查询任务 q_i ，给定 k 值，记 X_i 为一个二元随机变量，表示 q_i 涉及的列能否被前 k 个最热门的列覆盖。我们有：

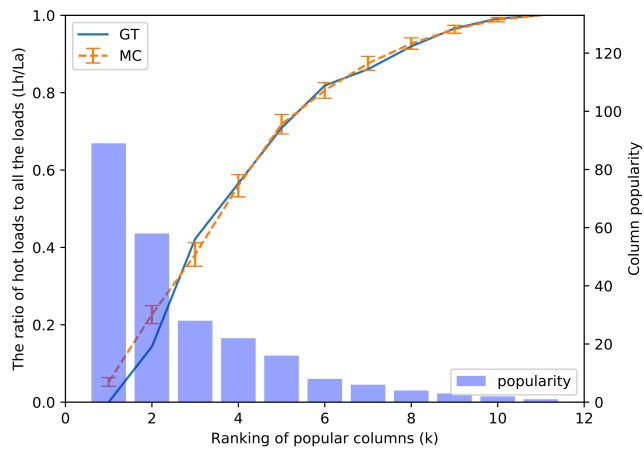
$$\Pr [X_i = 1] = \prod_{i=k+1}^n \left(1 - \frac{p_i}{m}\right) \quad (4.7)$$

假设来自查询任务 q_i 的负载是 d_i ，其中 d_i 是其访问的列的大小之和。如果 $X_i = 1$ ，那么 d_i 的期望值为：

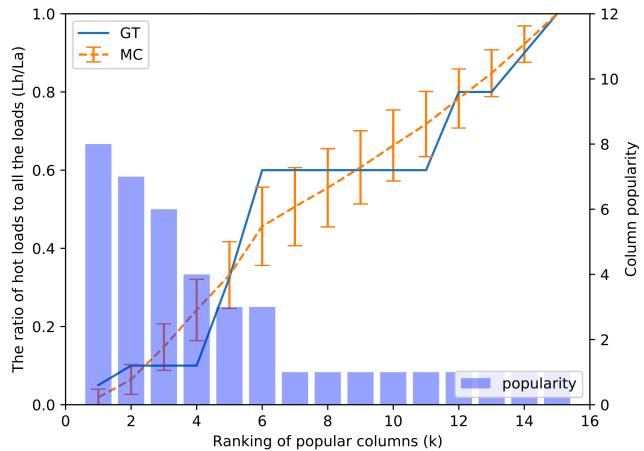
$$E [d_i] = \sum_{i=1}^k \frac{p_i s_i}{m} = \frac{\sum_{i=1}^k l_i}{m} \quad (4.8)$$

记 X 为可以由前 k 个最热门的列承担的查询任务的数量，则：

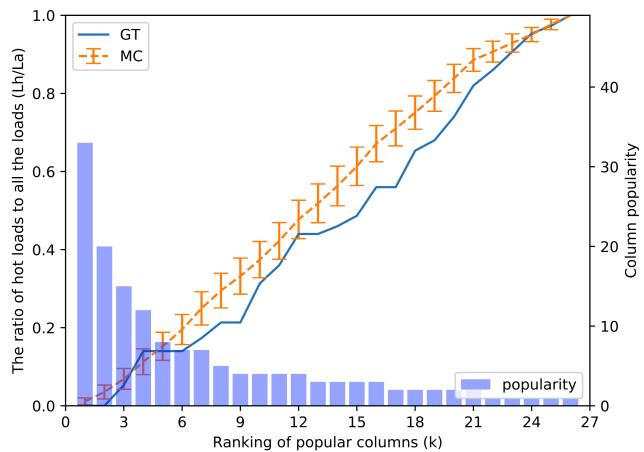
$$X = \sum_{i=1}^m X_i = m \prod_{i=k+1}^n \left(1 - \frac{p_i}{m}\right) \quad (4.9)$$



(a) TPC-DS 中的 data_dim 表



(b) TDC-DS 中的 web_returns 表



(c) TDC-DS 中的 web_sales 表

图 4.1 列的热度及归一化的 L_h 和 k 之间的关系。蒙特卡罗方法 vs. 实际情况 (Ground Truth)

因此，对于 L_h ，我们有：

$$\begin{aligned}
 L_h &= \frac{\sum_{i=1}^k l_i}{m} \cdot m \prod_{i=k+1}^n \left(1 - \frac{p_i}{m}\right) \\
 &= \sum_{i=1}^k l_i \prod_{i=k+1}^n \left(1 - \frac{p_i}{m}\right)
 \end{aligned} \tag{4.10}$$

根据等式 4.10，当各个列的负载差异程度很高（highly skewed）的时候，当 k 值较小时， L_h 随着 k 的增大迅速增大。

测量实验 2 我们做实验对比了用等式 4.10 求得的 L_h 的预测值（PV）和通过蒙特卡罗方法得到的 L_h 的值，结果展示在图 4.2 中，可以看到，两条曲线表现出相似的趋势。我们注意到这两种方法求出的 L_h 的值有误差，原因是假设生成的查询任务是相互独立的，然而这些生成的查询任务包含的列不能超出各个列的定额，这导致在查询任务生成过程后期产生的那些任务实际是有依赖关系的，具体来说，当列 c_i 的定额用尽，那么后面生成的查询任务访问列 c_i 的概率是 0 而不是 $\frac{p_i}{m}$ 。在实际操作中，我们需要生成的查询任务的数量超出 m ，以便消耗所有列的定额。

事实上，当生成的查询任务的数量变得很大时，公式预测的 L_h 的值能够接近蒙特卡罗方法得到的值。

测量实验 3 我们加入了更多测试来验证 L_h 和 k 之间的关系，其中用到的负载热度差异更大。具体来说，不同于之前对 TPC-DS 标准测试程序提供的查询任务只计数一次，我们对查询任务的热度（数量）进行了配置，使其服从 Zipf 分布（指数参数设置为 2）。这样一来，1) 不同列的热门度差异更大，2) 查询任务的数量更多。图 4.3 展示了先前提到的 3 个例子中归一化的 L_h 和 k 之间的关系。根据图中结果，我们发现 1) 因为大部分查询任务只访问很少一部分非常热门的列，大部分的负载可由复制的仅含少部分热门的列的副本承担；2) 当查询任务的数量增加，通过等式 4.10 预测的 L_h 的值更加接近蒙特卡罗方法得出的值，也就更加接近实际值。

总而言之，经过我们的评估，基于列的热度生成查询任务来估计访问模式的分布是可行的。此外，在列热度分布更加不均衡的情况下进行的测量实验（测量实验 3）也启发我们，把前 k 个最热门的列“捆绑”复制是有利可图的。

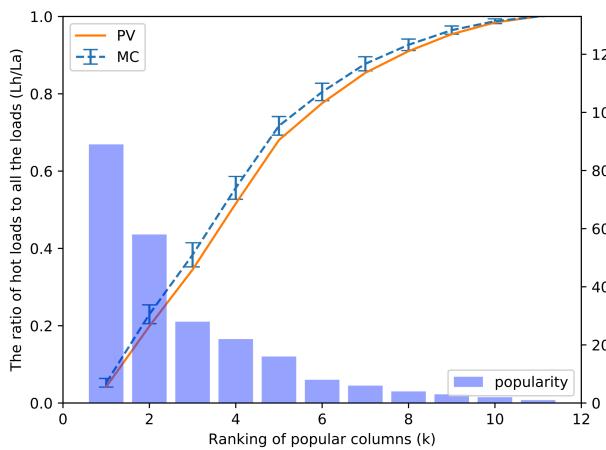
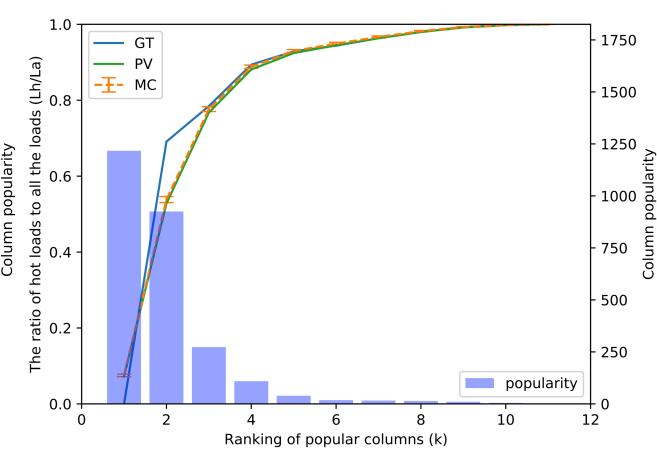
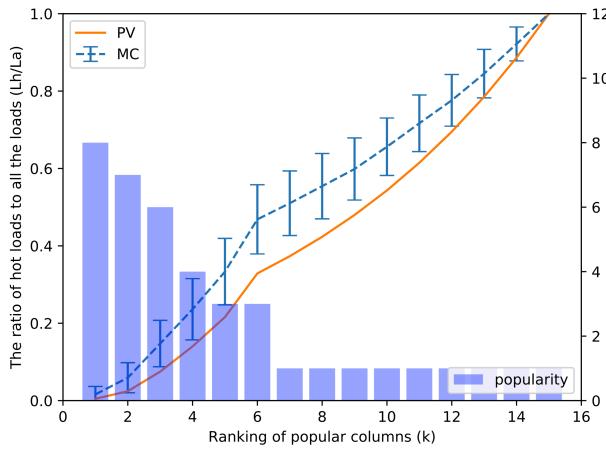
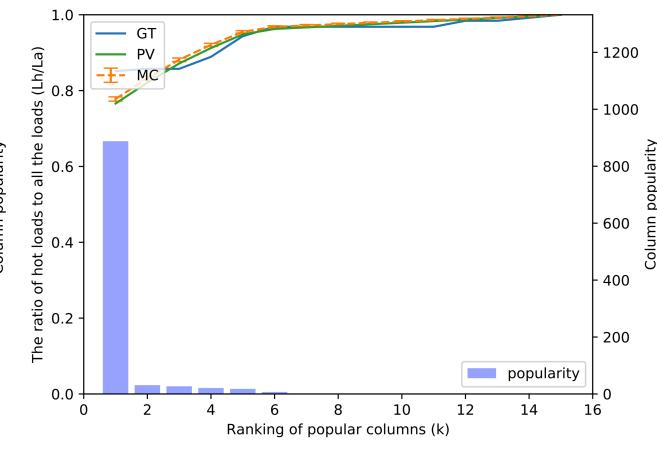
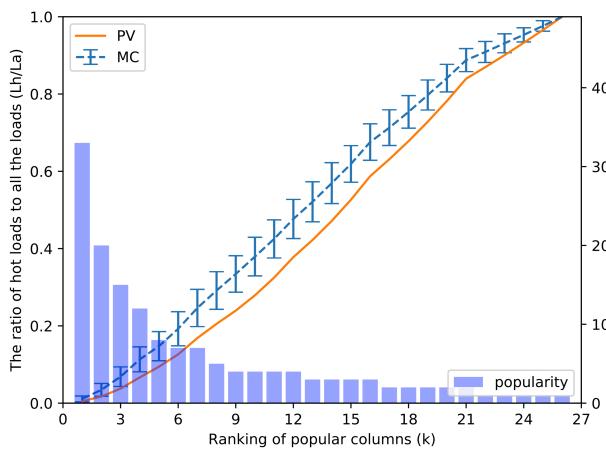
(a) TDC-DS 中的 *data_dim* 表。(a) TPC-DS 中 *data_dim* 表。(b) TDC-DS 中的 *web_returns* 表。(d) TPC-DS 中的 *web_returns* 表。(e) TDC-DS 中的 *web_sales* 表。

图 4.2 列的热度及归一化的 L_h 和 k 之间的关系。蒙特卡罗方法 vs. 公式预测

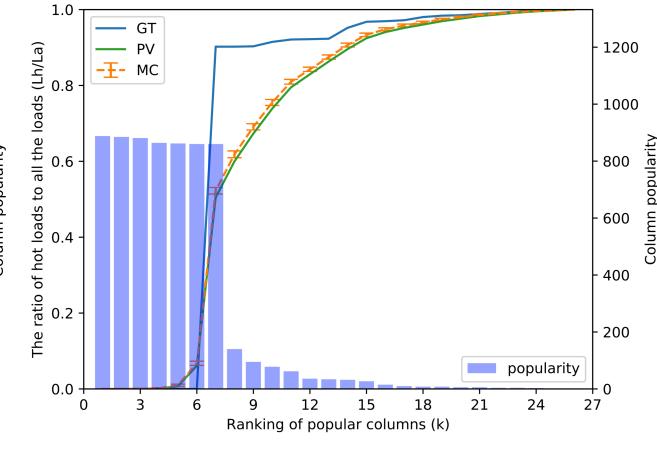
(f) TPC-DS 中的 *web_sales* 表。

图 4.3 负载差异大 (skewed workloads) 的情况下，蒙特卡罗方法、实际值和公式预测值的比较。

第五章 实现

在第 4 章中我们对问题进行了建模并且对我们提出的 Bundle-K 的方案做了理论分析，论证了我们的方案的可行性。在本章当中我们会介绍 CW-Cache 的具体实现。

5.1 架构概览

我们在 3.2.2 小节中介绍过的分布式内存文件系统 Alluxio 上实现了 CW-Cache，图 5.1 展示了 CW-Cache 的整体框架。CW-Cache 系统主要由两部分组成：CW-Master 和 CW-Client，它们分别是在 Alluxio Master 和 Alluxio Client 的基础上实现的。

CW-Master 实现了第 4 章中描述的 Bundle-K 方案的主要逻辑，除了原来 Alluxio Master 具有的响应 Client 的请求、维护全局的文件的元数据之外，还要记录保存查询任务对数据表中列的访问次数，运行 Bundle-K 方案中的逻辑对数据表进行列级别的复制，当 Client 访问数据表中的列的时候返回所有副本所在位置。

CW-Client 依旧是用户的应用与系统交互的桥梁。CW-Cache 接受应用的读/写请求，并与 CW-Master 交互获得文件所在位置，在执行 SQL 查询任务的时候，CW-Client 会发起远程调用，上传访问的文件的 URI、偏移量（Offset）和长度。读取数据表的列所在的文件时，它收到 CW-Master 传来的副本（原表）所在的位置，根据一定的策略自行决定从哪里读取。此外，图中的应用主要指上层应用框架，比如 Spark，Hive 等等，通过 Alluxio 提供的通用接口存取文件。Cache Server 上由 Alluxio Worker 来管理本地节点的文件或者对象。

Reporter 是增加在 Client 端用于将应用访问的文件的信息：文件 URI，偏移量和长度汇报给 Master；Recorder 是在 Master 端用于接收 Reporter 汇报的数据并将其存储在特定的数据结构中，维护热度的统计信息；Replication Manager 负责利用数据表的列的热度数据，执行第 4 章中的 Bundle-K 方案，计算出 K 和复制的份数 r ，对前 K 个最热门列进行“捆绑”复制。此外，Replication Manager 也会在 CW-Client 的读请求到来时返回所有副本（如果有）的位置，供其选择。

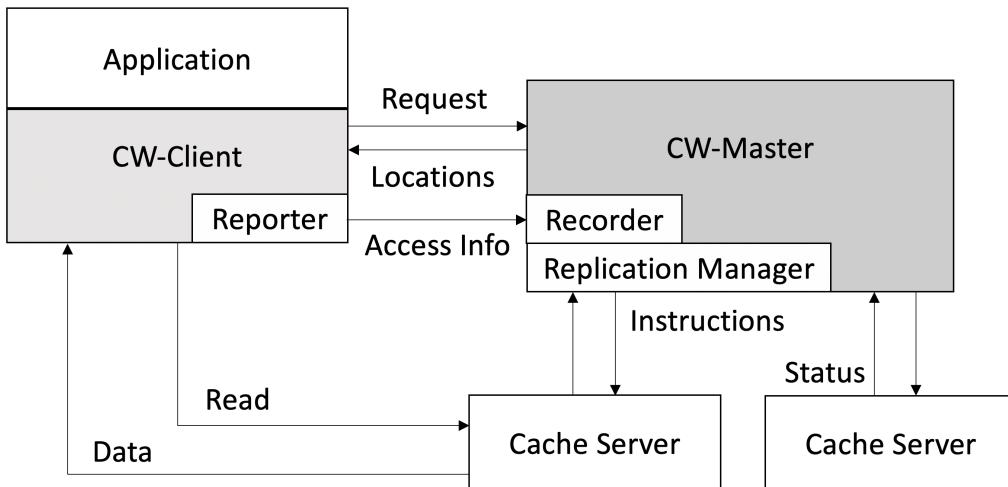


图 5.1 简单方案的架构设计。

5.2 实现细节分析

在本节中我们会介绍 CW-Cache 系统实现的细节，介绍我们在 alluxio 之上添加的部分代码，以及一些注意事项。

5.2.1 Reporter

Reporter 的功能是将读取文件的信息：文件 URI、偏移量、读取长度，上传给 CW-Master。第 3 章提到的简单方案是需要上层应用提供访问的列的语义信息，也就是使得 CW-Cache 得知应用具体访问的列。在 3.2.3 小节我们提到，我们放弃采用这种方法，是因为它需要上层计算框架与中间层的耦合，不符合软件设计模块化、低耦合的原则。我们想要在中间层实现列级别的负载均衡，利用尽可能少的信息，而 Alluxio 原本提供的读文件的接口就是接受文件 URI、偏移量、读取长度这三个参数。所以，为了方便代码实现，减少代码的耦合度，我们放弃了让上层将 Parquet 文件的语义信息传给 Alluxio 的想法，转为利用这三个参数构成的更加底层的信息，我们将之称为“文件片段”。Alluxio Client 在读取文件的时候，只是将文件 URI 通过远程调用传给 Master，Master 返回文件所在的 workers，Client 去相应的 worker 找到文件，根据偏移量和读取长度读取应用需要的部分，也就是说，偏移量和读取长度仅 Client 可见。为了实现这个 Reporter，我们添加了一个远程调用，将这三样信息一起传给 Master。Alluxio 1.8.0 版本使用 Apache Thrift 作为远程调用框架，我们在定义接口的 thrift 文件里定义：

```

1 struct UploadFileSegmentsAccessInfoTResponse {
2     1: list<fileSegmentInfo> replList
3 }
4 /**
5 */

```

```

5 * Upload UFSPath, offset and length to master for recording.
6 */
7 UploadFileSegmentsAccessInfoTResponse uploadFileSegmentsAccessInfo(
8     /** the UFSPath of the file */ 1: string UFSPath,
9     /** the offset in the file */ 2: i64 offset,
10    /** the length of bytes to be read */ 3: i64 len,
11 ) throws (1: exception.AlluxioTException e)

```

5.2.2 Recorder

Recorder 用于记录应用访问的列的信息，它实质是 Replication Manager 的一部分，我们把它作为 Replication Manager 的一个方法：

```

1 public Map<AlluxioURI, OffLenPair> recordAccess(AlluxioURI requestFile, long offset,
                                                 long length);

```

这个方法传入了之前提到的文件 URI、偏移量和读取长度三个参数，返回的是文件 URI 和偏移量、读取长度组成的配对的映射，这是因为这个方法是 Reporter 远程调用在 Master 端的实现，该远程调用也需要返回读取的文件片段的副本，以供 Client 选择。为了存储文件的访问信息，我们设计了 FileAccessInfo 类，它的成员变量和主要成员函数如下：

```

1 public class FileAccessInfo {
2     private AlluxioURI mFilePath;
3     private Map<OffLenPair, Long> offsetCount;
4     private long queryNum;
5     private long lastAccessTime; /* estimate query based on interval */
6     private long recordInterval;
7     private Set<OffLenPair> offsetWithinQuery;
8
9     public void incCount(OffLenPair offLenPair){
10         offsetCount.merge(offLenPair, (long) 1, Long::sum);
11         long currentTime = CommonUtils.getCurrentMs();
12
13         // check if it belongs to a new query
14         if (currentTime - lastAccessTime > recordInterval){
15             queryNum++;
16             offsetWithinQuery.clear();
17         }
18     }
19 }

```

```

17 }
18 else {
19     if (offsetWithinQuery.contains(offLenPair)) {
20         queryNum++;
21         offsetWithinQuery.clear();
22     }
23     else {
24         offsetWithinQuery.add(offLenPair);
25     }
26 }
27
28 lastAccessTime = currentTime;
29 }
30 ...
31 }

```

根据 4.3 节，对于一张数据表（即 `mFilePath` 标识）来说，用列的热度来估算共同访问模式，需要得到访问过这张表的查询任务的数量，在代码里对应成员变量 `queryNum`，以及每个列的访问次数，因为实现中用底层文件代替具有语义信息的列，因此代码里是 `Map<OffLenPair, Long> offsetCount` 来记录。成员函数 `public void incCount(OffLenPair offLenPair)` 是用于增加计数的，代码里引入 `lastAccessTime` 和 `recordInterval`，是因为我们在上传信息的时候没有附带 Client 的身份标识，这样 Master 对所有的请求一视同仁，所以我们设置时间窗口来近似估计哪些文件读取请求来自同一个查询任务。这样做的前提是我们认为同一个查询任务访问不同列的时间间隔很短，当然在多用户使用系统的情况下，得出的数据会损失部分准确性。

5.2.3 Replication Manager

Replication Manager 根据 Recorder 记录的统计信息，运行第 4 章中描述的关于 Bundle-K 方案中的算法，决定 K 和复制数量 r ，是 CW-Cache 系统的核心部分，并将需要复制的部分取出作为新的文件缓存在集群中。为此，我们在原 Alluxio Master 模块中添加了 `rep1` 模块，其核心是 `ReplManager` 类。在 5.2.2 小节中描述的 Recorder 实际是属于这个类的，这个类的成员变量主要有：

```

1 private FRClient frClient;
2 private ReplPolicy replPolicy;
3 private Map<AlluxioURI, FileAccessInfo> accessRecords;

```

```

5  private Map<AlluxioURI, FileRepInfo> fileReplicas;
6  private Map<AlluxioURI, AlluxioURI> replicaMap;
7  private Map<AlluxioURI, FileInfo> offsetInfoMap;
8  private int checkInterval; /* in seconds */

```

其中，`accessRecords` 是 `Recorder` 的实例；`fileReplicas`、`replicaMap`、`offsetInfoMap` 是用于复制；`replPolicy` 是复制策略的接口，目前我们的实现是 `Bundle-K`，`Bundle-K` 的代码见附录 A，后续我们会针对 `Bundle-K` 方案的不足之处开发新的策略，以及一些更加基础、简单的策略来和 `Bundle-K` 进行对比；`frClient` 是我们添加的一个特殊的 Client，它专门用于 `Master` 在对文件进行复制的时候，能够像普通的 Client 一样调用文件系统的功能来完成副本的复制、删除等操作。

5.2.4 周期性检查与复制

因为数据表的各个列的热度可能随着时间的推移而改变，`CW-Cache` 通过对缓存的数据表重新复制来使缓存服务器周期性地负载均衡。沿用论文 [27] 中推荐的方法，`CW-Cache` 每 12 小时基于过去 24 小时记录的各数据表的各个列的访问计数对文件重新重新复制。为了达到这个目的，`CW-Master` 在平常时间会持续对各个列的访问进行记录，同时它注册一项服务，每隔 12 小时调用 `Replication Manager` 的 `checkStats()` 成员函数，`master` 用章节 4 描述的方法估算列的共同访问模式，进一步计算最佳的要复制的列的数量 K 和复制的份数 r 。本身 `CW-Cache` 系统里缓存有原表和副本，我们只需要执行 `Bundle-K` 的方案计算新的 K 和 r ，如果 K 和 r 没有发生变化，那么什么都不用做；如果仅仅 K 改变，那么就通过复制或者删除副本使副本数量为 r ；如果两者均发生改变，那么就删除所有副本，重新复制即可。

我们有证据来支持周期性进行这个过程的有效性：在短期内（比如一天）生产集群中的文件热门程度是相对稳定的。事实上，研究人员观察到，在一个 Microsoft 的集群中，任何一天中被访问到的 40% 的文件，在之前与之后的四天里同样被访问 [27]。一个类似的结论也可以从 Yahoo! 集群数据集 [29] 中得到：接近 27% 的文件保持超过一周的热门程度。

5.3 系统额外开销

元数据的存储开销。为了统计文件片段的热度信息，`Master` 需要在内存中维持一部分元数据。对于每一个 `Parquet` 文件（可以是某一个 `Parquet` 文件的一部分）来说，`CW-Cache Master` 要维护它被访问过的文件片段（由偏移量和读取长度构成）的计数器，以及原文件片段和复制过后的文件片段（可能是多个）的映射。这部分元数据，我们仅保留一段时间的，

而不是历史累加，毕竟访问模式一直在变化，目前实现中是 12 小时复制分发一次，那么这些元数据就维护 12 小时，之后便重新统计。相对于 Alluxio 本身维护的文件的元数据来说，这些开销是很小的。

计算开销。求出复制的列的数量 K 和复制的份数 r 的这个过程是我们的系统实现中的主要开销。第 4.2 节的理论表明，算法的时间内复杂度是 $O(m \log m + n)$ ， m 是访问模式的数量， n 是列的数量，整个过程中也采用了不少近似处理，计算并不复杂。这个计算在目前的实现中是 12 小时进行一次，并且我们可以根据经验把这个时间设置为类似深夜这样的少有人使用系统的时间，这样的话计算开销的影响就极小了。

第六章 测试与评估

本章中我们对我们实现的采用 Bundle-K 方案的 CW-Cache 系统进行初步的测试与评估。

6.1 实验方法

6.1.1 实验环境设置

我们在一个 6 节点的 Amazon EC2 的集群中搭建 Spark、CW-Cache、HDFS 组成的平台，其中 1 台作为 Master，其他作为 Worker，每个节点是 r3.xlarge，各自有 4 个 CPU，，30.5 GB 内存。我们使用 iPerf 测得节点之间的网络带宽是 1Gbps。我们在平台上用 Spark 执行标准测试程序 TPC-H 提供的 SQL 查询任务，观察各项指标。应用均在 Master 上进行提交。

实验中所用的数据是利用 TPC-H 标准测试程序生成的规模为 10 的数据，我们利用 Spark 将其转换为了 Parquet 格式以进行实验。

6.1.2 负载

我们在第 4 章就观察到，执行 TPC-H 提供的任务便能够造成列的访问热度倾斜，在本评估实验中，我们将 TPC-H 的 22 个查询任务依次执行若干次，Master 在这个过程中会对访问的文件片段进行记录，一段时间内收集的信息便可供 CW-Cache 进行“列级别”的复制。在这之后，我们会控制应用程序随机启动 TPC-H 中的任务。

6.1.3 基准

本测试中，我们选择最基本的固定大小分块的方案进行对比，这个方法是很多分布式缓存/存储系统常常采用的，比如 HDFS、Windows Azure Storage 和 Alluxio。在固定大小分块方案中，文件被分成固定大小的文件块，存储在集群中。因为我们的方案是基于 Alluxio 构建的，因此我们采用原生 Alluxio 作为 CW-Cache 方案的对照。

6.2 实验结果

6.2.1 读取延迟

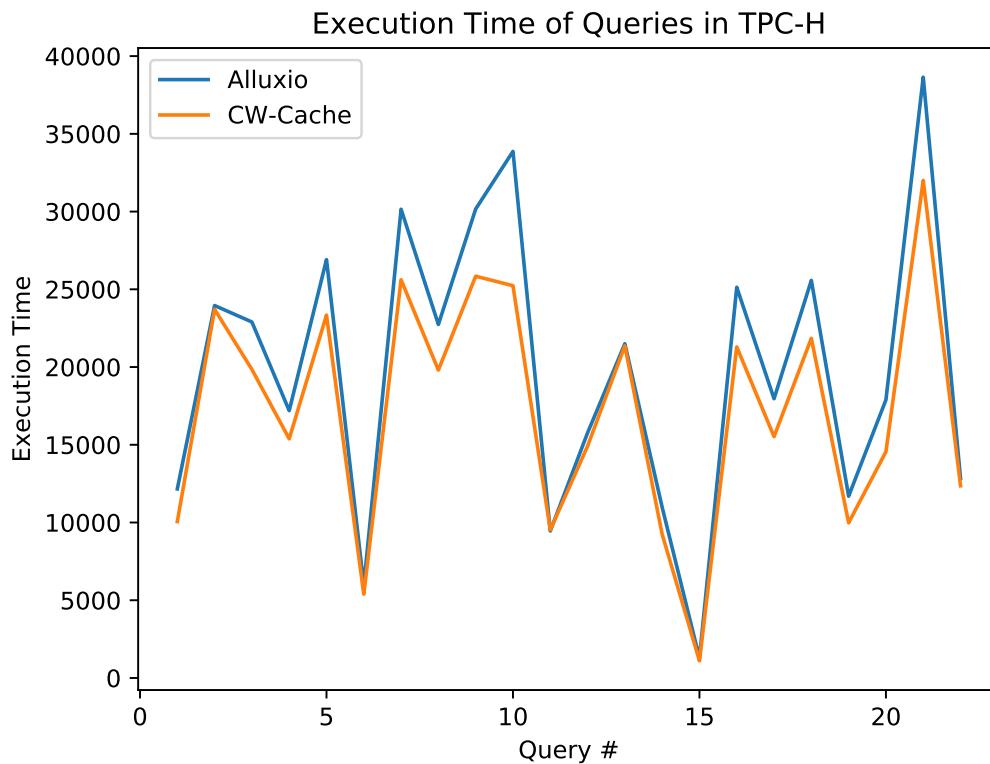


图 6.1 任务执行时间比较。

图 6.1 是我们得到的两个方案的查询任务的执行时间。由图可以看出，两条曲线的趋势基本一致，我们的 CW-Cache 比原生 Alluxio 系统要略好一点，最好的能降低延迟达 25%，有的则不如原生 Alluxio，我想可能的原因有：1) 有一部分任务属于计算密集型而不是 I/O 密集型，其瓶颈在于 CPU，我们的方案需要上传访问信息，反而增加 Client 与 Master 的通信开销，导致比原生 Alluxio 稍慢；2) 其次系统是初步实现，还未进行优化，本身性能还未达到令人满意的地步。

6.2.2 负载

图 6.2 展示了按照服务器负载大小排序后的服务器负载分布。由图可见，CW-Cache 优于原生方案。如果我们用不均衡因子衡量负载不均衡的程度，它被定义为：

$$\eta = \frac{L_{\max} - L_{\text{avg}}}{L_{\text{avg}}}, \quad (6.1)$$

其中 L_{\max} 和 L_{avg} 分别是服务器中最大的和平均负载，值更小的 η 表明更好的负载均衡。

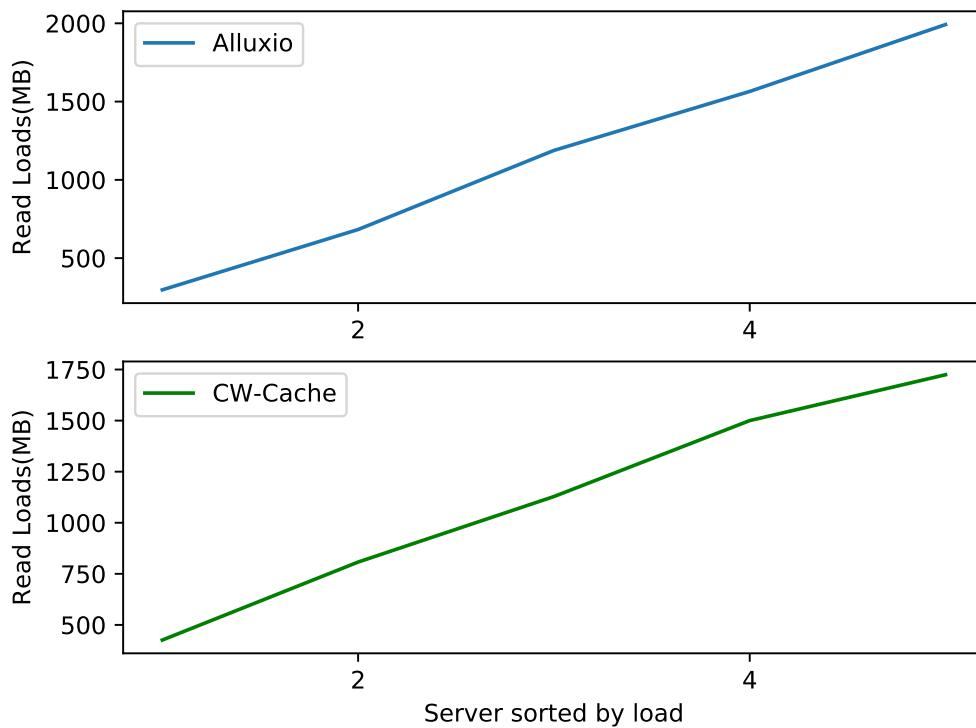


图 6.2 两种场景下的负载的分布。服务器的负载按照总的数据读取量计算。

那么我们的实验中 CW-Cache 方案比原生 Alluxio 方案好 26%。

第七章 总结与展望

本章对 CW-Cache 当前完成的工作进行总结，并对 CW-Cache 未来的研究方向与计划进行展望。

7.1 工作总结

我们注意到数据密集型集群中出现的负载均衡的问题，并且在文献调研中发现，研究人员几乎都关注一般意义上的文件的负载均衡，缺少对结构化数据的文件的特别关注。本文研究发现，集群里对结构化数据（数据表）的各个列的访问热度存在较大差异，且两列的热度越高，二者被共同访问的概率也越高。基于这个发现，本文设计、分析、开发了 CW-Cache 系统，希望实现结构化数据在列级别的负载均衡。目前 CW-Cache 实现了 Bundle-K 方案，即将热度排在前 K 的列“捆绑”在一起进行复制，复制 r 份。本文对 Bundle-K 方案进行了数学建模，给定一段时间内统计的各个列的热度和 SQL 查询任务的数量，求出当前最优的 K 和 r 。

我们在分布式内存文件系统 Alluxio 上实现了 CW-Cache，尽量只利用 Alluxio 能够获得的信息，避免与上层计算框架的耦合。

7.2 未来展望

在完成毕业论文的写作与答辩后，我仍将继续进行这个项目，采用 Bundle-K 方案的 CW-Cache 远远达不到完善的程度，还有很多工作于研究亟待进行，这个方案仅仅适用于一部分案例，还存在不少问题：

- CW-Cache 识别列是借助文件 URI、偏移量和读取长度，实质上已经放弃了文件存储的结构化数据的语义信息，是和系统实现的妥协。本身 Parquet 文件在存储时各个列在物理上并不连续，导致在 CW-Cache 系统看来，每一个列是由若干“文件片段”构成的。能否在不增加上下层耦合度的情况下再次引入列的语义信息，这是否能提高系统的性能，留待探索。
- 考虑列的共同访问模式，当前 CW-Cache 系统是设置一个时间窗口，来判断相邻的被访问的列是否同属一个 SQL 查询任务。现在我们认为当 Spark SQL 解析 SQL 语句后，会并发

地取获取需要访问的列，于是将这个窗口设置得比较小，但情况是否真的全是这样呢？此外，只有一个用户使用 CW-Cache 时还好，当多个用户同时使用时，结果将会不准确，我们需要想其他办法避开。

- Bundle-K 方案也许不够灵活，Bundle-K 有两个极端特例，一个是 Bundle-1，也就是把热度最高的列复制多份并分散在集群中缓存，如果这张表的访问热度倾斜很严重，那么 Bundle-1 可能取得好的效果；另一个是 Bundle-N，也就是全表复制，相当于现有的复制方案，这个可以完全避免表内数据 shuffle，但是复制成本高。我们现在的 Bundle-K 是这两个方案的折中，但是我们目前将这 K 列视作整体一起复制，复制的份数相同，不够灵活。Bundle-K 应用在图 4.2a、4.3a、4.3b、4.3c 所示的表上有不错的效果，因为当 K 比较小的时候，随着 K 的上升，前 K 列组成的副本能承担大量的负载，而图 4.2b、4.2c 的曲线接近线性增长，收益不大。

此外，数据 shuffling 与数据放置的位置、网络通信状况、任务调度等因素均有关，非常复杂，它产生的影响难以通过数学建模进行研究。我们非常希望有一个方法能够以较小的复制开销实现负载均衡，同时能够避免 shuffling，而这样的 one-size-fit-all 的方法几乎是不存在的，还是需要在不同方案间考虑不同场景进行取舍。

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[M]. USA: ACM Books, 2016.
- [4] Ghemawat S, Gobioff H, Leung S T. The google file system[C]. 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[C]. 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[C]. Proc. ACM SIGCOMM, 2015.
- [9] Huawei. NUWA. https://www.youtube.com/watch?v=0smZBRB_0Sw.
- [10] Asanovic K, Patterson D. FireBox: A hardware building block for 2020 warehouse-scale computers[C]. 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[C]. ACM HotNets, 2013.

- [15] Gao P X, Narayan A, Karandikar S, et al. Network requirements for resource disaggregation[C]. Proc. USENIX OSDI, 2016.
- [16] Ananthanarayanan G, Ghodsi A, Shenker S, et al. Disk-locality in datacenter computing considered irrelevant[C]. ACM HotOS, 2011.
- [17] Jonas E, Venkataraman S, Stoica I, et al. Occupy the cloud: Distributed computing for the 99%[C]. 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[C]. Proc. IEEE Symp. Mass Storage Syst. and Technologies, 2010, 2010.
- [22] Rashmi K, Chowdhury M, Kosaijan J, et al. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding[C]. 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[C]. 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[C]. 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[C]. 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[C]. Proc. ACM IMC, 2009.
- [31] Chowdhury M, Kandula S, Stoica I. Leveraging endpoint flexibility in data-intensive clusters[C]. Proc. ACM SIGCOMM, 2013.

- [32] Greenberg A, Hamilton J R, Jain N, et al. VL2: a scalable and flexible data center network[C]. Proc. ACM SIGCOMM, 2009.
- [33] Yu Y, Huang R, Wang W, et al. Sp-cache: Load-balanced, redundancy-free cluster caching with selective partition[C]. 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[C]. 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[C]. Proc. ACM SoCC, 2013.
- [38] Huang Q, Gudmundsdottir H, Vigfusson Y, et al. Characterizing load imbalance in real-world networked caches[C]. ACM HotNets, 2014.
- [39] Huang C, Simitci H, Xu Y, et al. Erasure coding in windows azure storage[C]. Proc. USENIX ATC, 2012.
- [40] Sathiamoorthy M, Asteris M, Papailiopoulos D S, et al. Xoring elephants: Novel erasure codes for big data[J]. CoRR, 2013, abs/1301.3791.
- [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[C]. 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 Bundle-K 方案核心代码

```
1 package alluxio.master.repl.policy;
2
3 import alluxio.Configuration;
4 import alluxio.PropertyKey;
5 import alluxio.collections.Pair;
6 import alluxio.master.block.BlockMasterFactory;
7 import alluxio.master.repl.meta.FileAccessInfo;
8 import fr.client.utils.MultiReplUnit;
9 import fr.client.utils.OffLenPair;
10 import fr.client.utils.ReplUnit;
11 import org.slf4j.Logger;
12 import org.slf4j.LoggerFactory;
13
14
15 import java.util.Collections;
16 import java.util.List;
17 import java.util.stream.Collectors;
18
19 /**
20 *
21 */
22
23 public class BundleHottestKPolicy implements ReplPolicy {
24     private static final Logger LOG = LoggerFactory.getLogger(BundleHottestKPolicy.class);
25     ;
26
27     private double weight;
28     private int workNum;
29
30     public BundleHottestKPolicy() {
31         weight = Configuration.getDouble(PropertyKey.FR_REPL_WEIGHT);
32     }
33
34     private void updateWorkNum() {
35
36
37 }
```

```
33     workNum = BlockMasterFactory.getBlockMaster().getWorkerCount();  
34 }  
  
35  
36     private double calcObjective(double hotLoad, double coldLoad, double hotSize, int r){  
37         double hotSqu = hotLoad * hotLoad;  
38         double coldSqu = coldLoad * coldLoad;  
  
39  
40         return (hotSqu / r + coldSqu) / workNum - (hotSqu + coldSqu) / (workNum * workNum)  
41             + weight * r * hotSize;  
42 }  
  
43  
44     @Override  
45     public List<RepUnit> calcReplicas(FilePathInfo fileAccessInfo) {  
46         updateWorkNum();  
  
47         List<Pair<OffLenPair, Long>> allPops = fileAccessInfo  
48             .getOffsetCount()  
49                 .entrySet()  
50                     .stream()  
51                         .map( o -> new Pair<>(o.getKey(), o.getValue()))  
52                             .collect(Collectors.toList());  
  
53  
54         long totalSize = allPops  
55             .stream()  
56                 .mapToLong(o -> o.getFirst().length)  
57                     .reduce((long) 0, Long::sum);  
  
58  
59         List<Pair<OffLenPair, Double>> sortedLoads = allPops  
60             .stream()  
61                 .map( o -> new Pair<>(o.getFirst(), o.getFirst().length * o.getSecond() *  
62                     1.0 / totalSize))  
63                     .sorted((e1, e2) -> e1.getSecond() < e2.getSecond() ? 1 : -1)  
64                         .collect(Collectors.toList());  
  
65  
66         LOG.info("file: {}, sorted loads: {}", fileAccessInfo.getFilePath().getPath(),  
67             sortedLoads);  
  
68  
69         List<Pair<OffLenPair, Long>> sortedPops = sortedLoads  
70             .stream()  
71                 .map( o -> new Pair<>(o.getFirst(), fileAccessInfo.getOffsetCount().get(o  
72                     .getFirst()))))
```

```
71     .collect(Collectors.toList());  
  
73     LOG.info("query_num: {}", sorted_pops, fileAccessInfo.getQueryNum(),  
74             sortedPops);  
  
75     List<Double> sortedSizes = sortedPops  
76         .stream()  
77         .map( o -> o.getFirst().length * 1.0 / totalSize)  
78         .collect(Collectors.toList());  
  
79     double allLoad = sortedLoads.stream().mapToDouble(Pair::getSecond).reduce(0.0,  
80             Double::sum);  
  
81     int bestK = -1;  
82     int bestR = -1;  
83     // initial obj as no replicas  
84     // double bestObj = calcObjective(0, allLoad, 0, 0);  
85     double bestObj = Double.POSITIVE_INFINITY;  
  
86     for(int k = 0; k < sortedPops.size(); k++){  
  
87         double hotLoad = sortedLoads  
88             .stream()  
89             .limit(k + 1)  
90             .mapToDouble(Pair::getSecond)  
91             .reduce(0.0, Double::sum);  
  
92         double regret = sortedPops  
93             .stream()  
94             .skip(k + 1)  
95             .mapToDouble( o -> 1 - o.getSecond() * 1.0 / fileAccessInfo.  
96                     getQueryNum())  
97             .reduce(1.0, (d1, d2) -> d1 * d2);  
  
98         hotLoad = hotLoad * regret;  
  
99         double coldLoad = allLoad - hotLoad;  
  
100        double hotSize = sortedSizes  
101            .stream()  
102            .limit(k + 1)  
103            .reduce(0.0, Double::sum);  
104    }
```

```
111     double doubleR = Math.sqrt(hotLoad * hotLoad / (workNum * weight * hotSize));  
  
113 //         System.out.println("k=" + k + "; Lh=" + hotLoad + "; workNum=" + workNum +  
114 //         "; Sh=" + hotSize + "; r=" + doubleR);  
  
115     if (doubleR > workNum){  
116         double localObj = calcObjective(hotLoad, coldLoad, hotSize, workNum);  
117         if (localObj < bestObj){  
118             bestObj = localObj;  
119             bestK = k;  
120             bestR = workNum;  
121         }  
122     }  
123     else {  
124         int floorR = (int) Math.floor(doubleR);  
125         int ceilR = (int) Math.ceil(doubleR);  
  
127         double floorObj = calcObjective(hotLoad, coldLoad, hotSize, floorR);  
128         double ceilObj = calcObjective(hotLoad, coldLoad, hotSize, ceilR);  
129  
130         double localObj = Math.min(floorObj, ceilObj);  
131         if (localObj < bestObj){  
132             bestObj = localObj;  
133             bestK = k;  
134             bestR = floorObj < ceilObj ? floorR : ceilR;  
135         }  
136     }  
137 }  
138  
139 LOG.info("Opt K = {}; Opt R = {}; Opt Obj = {}", bestK, bestR, bestObj);  
140  
141 List<OffLenPair> replPairs = sortedPops  
142     .stream()  
143     .limit(bestK + 1)  
144     .map(Pair::getFirst)  
145     .collect(Collectors.toList());  
146  
147     return Collections.singletonList(new ReplUnit(replPairs, bestR));  
148 }
```

```
151     @Override  
152     public List<MultiRepUnit> calcMultiReplicas(List<FileAccessInfo> fileAccessInfos) {  
153         return null;  
154     }  
155 }
```

致 谢

逝者如斯夫，不舍昼夜。人生就是不断地拿笔为一段段故事郑重开头，也不断地为另一些故事画上句号，不论它们是否圆满，人生短暂，这一辈子能叙写的故事不会太多。我很有幸，很感激，在 18 岁到 22 岁这段最美好的年华，我的故事与东大一起书写，与南京一起书写。

大学四年，俯仰之间而已。还记得大一烈日炎炎下不屈不挠的军训让我的内心一点一点坚定；还记得和文教优秀的伙伴一起办过一场场讲座、大活动，和伙伴结下深厚友谊，同时感受人文熏陶；还记得四年三次和拉丁舞社的伙伴一起登上跨年舞台，为大学留下难忘的精力；还记得一次次比赛、一次次考试让我成长...尽管四年时光匆匆从指间溜走，但是回想起来，还是又那么多点点滴滴值得我珍藏，回味一生。

到了这一段，我也快要给自己的大学生活画上句号了。这篇论文背后的项目已经磕磕绊绊进行了五个多月，从开始不知所措到渐渐找到感觉，我经历了不少。科研的路上没有明确的路牌，在摸索中前进，也可能走入岔路，再掉头回来，踏上新的路，如此往复，坚持不懈，终可能有所获。这个项目到这里仅仅是一个阶段，距离最终成果还有不少距离，我仍然在路上。感谢项目组里余旻晨师兄的指导与辛勤付出，感谢项目组杨柏辰学弟为代码、测试贡献了不少力量，感谢王威老师对项目提出很多建设性意见，感谢余英豪师兄远程参与，感谢田黄石师兄、翁祈桢师兄、晏达师兄的宝贵建议。

回到东大，感谢我的校内指导老师肖卿俊老师的悉心指导，他带我进入科研的大门，让我参与课题组的项目，在组会上作报告锻炼我的能力，指导我们参加 SDN 大赛，为我写推荐信。感谢所有教过我的老师，是你们让我看到计算机科学丰富多彩的世界，激发我在这个世界继续探索的动力。感谢我的辅导员老师刘舒辰和魏敏娜老师，在我的大学生活中给予我帮助和鼓励。感谢与我并肩作战的舍友刘浩、王坤鹏、冷相宏同学，感谢 2015 级计科一班的同学，感谢计算机学院、东南大学。最最重要的，感谢我的父母郑群超先生和肖丽女士，你们养育我健康长大，你们的付出让我在求学之路上没有后顾之忧。

希望此篇成为我在计算机科学道路的起点，遇到困难能够不屈不挠，乘风破浪，不忘初心。与诸君共勉。