中国科学技术大学

博士学位论文

云计算基础软件架构的研究和实现

作者姓名：代 栋

学科专业：计算机系统结构

导师姓名：周学海 教授

完成时间：二零一三年五月

University of Science and Technology of China

A dissertation for doctor degree

Research on Software Infrastructure of Cloud Computing

Author :	Dong Dai
Speciality :	Computer System
Supervisor :	Prof. Xuehai Zhou
Finished Time :	May, 2013

云计算基础软件架构的研究和实现

计算机科学学院　代栋　中国科学技术大学

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____　　签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

□ 公开　□ 保密 _____ 年

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

摘 要

过去的五年里，云计算得到了长足的进步，在工业界的应用范围越来越广。随着各类应用的增多，云计算也使得计算和存储成为一种资源，像水电一样逐渐渗透到人们生活的各个方面。这些应用的出现离不开云计算基础软件架构领域的发展，尤其是分布式存储技术和分布式计算技术的进步。然而，新型的应用不断出现，比如实时搜索，在线推荐系统，社交网络分析等等，这些应用具有一些共同的特点：对数据量的要求大，比如实时搜索，我们希望能够搜索到最新的来自实时媒体，比如微博、新闻网站、社交网络中出现的信息，并且将这些结果和传统搜索引擎的结果混合 (*mesh-up*) 来提供用户感兴趣的结果。这种情况下，实时搜索需要处理的数据集只会更大。另外一个特点是对数据随机访问的速度要求很高。实时应用所处理的数据往往是无结构的小数据，比如实时搜索处理的爬取的页面信息，在线推荐系统所处理的用户评分和浏览信息等，这些信息多半是未经处理的小的原始数据，无法将其处理成连续读取的大数据块，因此应用需要对它们进行随机的读写，再加上实时应用需要在较快的时间内产生结果，因此对底层存储系统的访问速度要求越来越高。第三个特点是应用的复杂度也越来越高，对新数据的响应要求越来越快。典型的推荐系统，搜索引擎，社交分析的过程中都需要大规模的迭代处理，这些迭代之间通常有强依赖性，这就需要计算模型的支持以简化这类应用的编写提高应用的执行速度；另外既然是实时应用，其对于新的数据就需要能够尽快的进行处理。在应用的驱动下，近年来，不断有新型的存储系统和计算系统应用这些应用中，但是这个领域中依旧存在需要问题亟待解决。

本文基于这些应用的需求，对云计算的基础软件架构进行了深入的研究，包括对现有架构的优化配置以提高其性能，为海量数据的高速随机写设计并实现了一个完全基于内存的分布式存储系统，为实时应用复杂的计算模式设计并实现了一个基于触发器的通用计算模型。本文的主要工作和贡献如下:

- 提出了一种基于模糊逻辑的 Hadoop 集群异构配置工具，该工具使用模糊逻辑算法, 将正在运行 Hadoop 系统的异构集群中服务器的各种硬件参数以及历史运行数据作为模糊输入，根据模糊规则自动生成参数配置来提高 Hadoop 集群本身的执行效率。通过将传统的 Hadoop 集群配置中优化参数的方法转变成了优化规则的方法，极大的降低了配置集群的成本。实验方法表明，该模糊规则工具根据异构集群的多项指标生成的参数配置能够有效的提高应用的执行速度。

- 在基于内存的分布式存储系统 Sedna 中，我们提出了一种基于层次化的集群管理方案，并且和一种新的分布式哈希算法结合起来，极大的提高了集群的可扩展性以及进行动态负载均衡的灵活性。除此之外，我们还在传统的存储系统访问 API 的基础之上，设计并且实现了一组专用于实时应用的实时访问 API，来进一步提高存储系统对实时应用的支持。实验证明，Sedna 存储系统具有和内存缓存系统接近的速度，能够保证数据的持久性，在大量数据写入的情况下，研究能够很好的保证服务器之间的负载均衡，是一个有效的分布式存储系统。

- 我们将基于触发器的编程模型扩展到分布式计算领域，提出了一种完全基于触发器的通用编程模型及其运行时系统 Domino。在该系统的设计和实现中，我们提出了一种最终同步的模型，很好的解决了分布式的纯异步的触发器模型如何进行数据同步的问题；通过引入多种同步模型（完全异步、最终同步、严格同步），我们为开发人员提供了灵活的选择方案。在 Domino 中，我们还提出了实时恢复的概念，对于执行过程中的错误可以做到容忍和实时恢复，极大的提高了计算模型的扩展性和可用性。通过将多个典型的复杂应用在 Domino 上进行实现并且比较性能，我们证明了 Domino 具备非常好的扩放性并且在复杂的计算应用中，其性能上远优于传统的基于 MapReduce 模型的方案。

关键词： 云计算基础软件架构，分布式存储系统，编程模型，计算框架，实时应用

ABSTRACT

In the past five years, cloud computing has been dramatically developed because their widely usage in industry. Along with variable kinds of applications began to appear in cloud, cloud computing became more and more common in people's diary life. However, some new applications like realtime searching, online recommendation, social network analysis etc. still give us challenges: 1) all these applications need to process huge amount of dataset, which give the storage systems lots of presure on scalaibity. For example, the realtime search engine needs to process information from different sources and *mesh-up* them to generate results that users may be interested with, so all this info needs to be stored. 2) They needs a higher random data access speed as they need to produce final results in in *realtime* fashion. The random data access pattern is neccesary as most input datasets are small, fragile, raw data, it is hard to construct these data pieces into a continous large data block. 3) The computation is much more complex than traditional applications. Most of these applications included machine learning or data mining algorithms, which need lots of iterative and incremental computations. Besides, due to the *realtime* requirement of these applications, they need to be more sensitive to the new data. Based on these challenges, there are some new storage systems and programming models appearing recently, however, the main problems still have not solved well.

In this paper, we study the cloud software infrastucture in different aspects to build a complete framework for these applications, the main works and contributions of this paper include:

- We propose a new auto-configuration tool for heterogeneous Hadoop cluster. This tool collect all the hardware parameters and history execution information as the input of our fuzzy algorithm, and produce a collection of corrent Hadoop

configuration to accelerate the MapReduce job execution speed in Hadoop. Our solution change the way of configuring Hadoop from optimzing the parameter to optimizing the fuzzy rules. The experiments show our tool improve the Hadoop cluster performance dramatically especially for the heterogenous cluster.

- In our memory based distributed keyvalue storage system (Senda), we propose a new hierarchical architecture for distributed stroage systems. Woring with the new distribtued hash algorithm proposed in Sedna, this new architecture improved the scalability and the flexibility of load balance in Sedna. Besides, we propose a new API suit for realtim applications, which is much more sensitive to data modification than traditional API. The experiments show that Domino can archieve a much better performance than current disk-based distributed storage systems and be comparable with the widely used memory cache system.

- We extend the long history trigger-based programming model into the distributed computing area with some new ideas. Domino is a trigger-based genenral distributed programming model in cloud. To overcome the limitations of traditional trigger-based models, we propose a eventually synchronous model to solve the problem that how different actions synchronize their executions. Besides, through introducing different synchronization models (asynchronous, evenutally synchronous, strict synchronous), Domino provides developers flexbile solutions for their needs. In Domino, we also propose a *realtime* recovery concepts and implementation which we beleive dramatically improve the scalaibility and speed of distributed computation. We implement different applications in Domino and compare their performance, the experiment results show that our Domino framework keep a good scalability and better performance than traditional MapReduce based solutions.

Keywords: Cloud Computing, Software Infrastructure, Distributed Storage, Programming Model, Computation Framework, Realtime Applications

目 录

表格

插图

第一章　绪论

1.1　云计算的基本概念

进入新世纪以来，计算机基础技术上的长足进步，特别是在半导体技术、存储技术、网络技术以及处理器技术上不断取得巨大进步，使得人们有能力建造具有大量计算能力和存储能力的数据中心或者大规模集群。在这些大规模集群的基础上，人们开始将计算机资源看做是像水资源和电资源一样的公共资源，希望提供一种随需随得，按需付费，多人共享，可监督可测量的新型使用模型。在 2006 年的搜索引擎大会上，Google 时任 CEO 施密特首次完整的提出云计算的概念（这个概念最早成为“云端计算”，来源于 Google 工程师的 Google 101 项目），立刻被学术界和工业界所广泛接受，成为了描述当前基于互联网的计算模式的标准概念。

云计算的概念本身包括的内涵非常广泛，学术界也一直没有产生一个标准的定义。维基百科将云计算描述为一种基于互联网的计算方式，通过这种方式共享的软硬件资源和信息可以按需提供给计算机和其他设备。这是一种描述性的概念，从学术的角度来看，本文认为云计算 [3] 是一门包括了分布式计算 (distributed computing)、并行计算 (parallel computing)、效用计算 (utility computing)、分布式存储 (distributed storage)、虚拟化 (virtualization) 以及数据中心网络 (data center) 等学科在内的，关联了计算机系统软件、数据库和网络的综合学科。

按照服务模式划分，云计算可以被分为三个层次：IaaS、PaaS、以及 SaaS，按照服务的对象可以划分为公有云，私有云以及混合云。这里我们将按照服务模式的区别来划分云计算的不同类型。

- IaaS(Infrastructure as a Service，即基础架构即服务)，主要关注虚拟化技术，通过对硬件资源 (包括 CPU、I/O、网络等) 的虚拟化，为用户提供更为灵活，低成本的基础架构服务，因此，虽然 IaaS 主要依靠软件技术来实现，

但本文中我们仍称之为云计算的基础硬件架构。其中典型的研究内容如各种虚拟技术、资源调度算法、虚拟机节能等；而典型的商业应用如 Amazon 的 EC^2[4](Elastic Computing Cloud)，RackSpace[5] 等。用户在这种服务中不仅仅可以按需获得计算和存储资源还可以控制防火墙、网络拓扑等，然而所有这些操作都是在虚拟化的基础之上完成，用户无法接触到真正的物理设备。这也是 IaaS 与传统的 ISP 托管服务的标志性区别。

- PaaS(Platform as a Service，即平台即服务)，主要关注并行和分布式技术，通过将大规模分布式集群抽象成为统一的存储和计算服务来为用户提供分布式系统平台。用户在这种服务模式中获得的是对应用程序运行环境的控制，用户无法接触到实际的物理机器以及虚拟机器，所有的资源都通过 API 抽象给用户程序使用，比较典型的例子如 Googe App Engine[6]，或者 Amazon DynamoDB[7] 等服务。由于效率和速度的因素，提供此类服务时多直接使用物理机，通过在跨域部署的数据中心上构建一层分布式的软件系统来提供这种稳定的、性能可扩放的平台服务。我们称这类软件系统就为：**云计算的基础软件架构**，也正是本文研究的课题。

- SaaS(Software as a Service，即软件即服务)，主要为用户提供各式各样的基于云计算基础软硬件平台的软件服务，这些软件不同于传统的单机部署的私有软件，在云计算技术的支撑下，这些软件具备了更好的随需扩放，随处访问等能力，比如 Saleforce 公司提供的在线 ERM 套件等。在这种服务模式下，用户直接使用服务提供商提供的应用程序，并不会接触到物理机器、虚拟机、操作系统、数据库访问 API 等层的数据。

云计算一经提出便发展迅速，大量的企业利用 PaaS 以及 IaaS 提供的服务为最终用户提供 SaaS 服务，这也被证明是一种非常有效的商业模式。不过相比之下，在 IaaS 层和 PaaS 层上，不仅仅出现了新的按需服务的商业模式，更重要的是出现了许多新的技术和思想，不断提高着人们对于计算机技术本身的理解和掌握。特别在 PaaS 领域，由于需要通过管理大量的服务器为用户提供统一的存储和计算能力，这都对底层操作系统、文件系统、存储系统、以及计算模型等设计都提出了新的挑战，因此在云计算基础软件方向，新的技术更是层出不穷。

1.2　研究背景和意义

本论文的研究内容是云计算中 PaaS 层的一个子方向：云计算中分布式存储及分布式计算模型。这两者也是我们刚才所提到的**云计算软件体系架构**中的核心组件。

1.2.1　分布式存储系统的发展和新的需求

分布式存储通过管理多台计算机协同提供存储能力。相比较传统存储服务它能够提供更好的存储容量、数据备份和容错能力也能够提供更好的数据访问速率。按照其存储数据类型的区别，可以分为分布式文件系统和分布式对象存储系统；按照应用场景的不同，也可以分为网络文件系统、并行文件系统，以及分布式存储系统。

网络文件系统 (network file system) 是将数据维护至分布式环境下的一种解决方案。它是一种典型的客户机/服务器的系统: 数据存储在存储节点上，客户机通过和访问本地文件系统相同的方式来访问位于存储节点中的数据，通常情况下存储节点位于网络的另一个位置。比较典型的系统包括 Sun 公司的 NFS 系统 [8] 以及 CMU 开发的 AdnrewFS 系统 [9]。

由于受到设计思路的限制，这些存储系统往往存储能力有限，面对海量的数据以及大量的服务器构成的机群情况下很难处理，且客户机/服务器的模式使得客户机和存储节点之间的网络带宽成为系统最大的瓶颈。在这种情况下，Google File System(GFS)[10] 被提出来解决这个问题。GFS 基于大量的廉价 PC 以及其上附带的存储设备构建了一个可扩展的文件系统。与 NFS 等传统的存储系统不同的地方在于，GFS 在设计之初就考虑到了海量服务器和廉价 PC 带来的高失败率，通过多备份和对数据写操作进行限制极大的提高了分布式存储系统的可用性和扩展性。在 GFS 的基础之上，2006 年 Google 实现了 BigTable[11]。Bigtable 不是一款文件系统而是一个基于 GFS 实现的分布式对象存储系统，其中所存储的数据是键值对数据。Bigtable 提供了与传统数据库类似的表结构，通过降低对数据读写的 ACID 要求提供了传统关系数据库所不能提供的良好的扩放性和容错性。

Hadoop[12] 是 Apache 基金会开发的开源 GFS 的实现。它不仅仅包括了 GFS

这一分布式存储系统 (在 Hadoop 中被称为 HDFS)，也包括了 Bigtable 的开源实现 (HBase)，以及后面会讲到的 MapReduce 的实现。通过对 Google 核心系统的开源实现，Hadoop 实现了基于廉价 PC 设备的大规模存储能力，并且通过在 Yahoo!, Amazon, Facebook, Twitter 等公司的大量应用已经成为了业界标准。

GFS 以及 Hadoop 的大规模使用使得人们开始重新考虑云计算对分布式存储系统的要求和影响，从而产生了新的设计思路：1）抛弃传统的单一高性能节点或者专用硬件来提高速度的做法，转而考虑如何在大量的廉价设备上实现高性能的存储能力；2）扩放行和容错性的重要性被前所未有的提高。扩放性和容错性是衡量一个云存储系统的核心指标；3）基于对象的存储系统越来越重要，因为基于对象的存储系统介于文件系统和数据库之间，能够为数据提供结构化的语义。

因此，从 2006 年开始，大量的分布式存储系统被应用于生产系统中。Dynamo[13] 是 Amazon 公司设计实现的一种高可用的键值存储系统，它通过高可用的特性保证了写操作始终成功。与其说其是一款分布式存储系统不如说它事实上定义了分布式存储系统的分布式管理架构，而真正的数据存储则可以简单的由各式各样的本地文件系统实现并且接入。Dynamo 的设计中引入了点对点网络的概念，使得集群的扩放性极佳。相比较 GFS，Dynamo 提供了基于键值对的数据模型，更加适合存储小数据。类似的系统还有 Cassandra[14]，其结合了 Dynamo 和 Bigtable，为应用提供了结构化的数据语义。

类似 Dynamo 的点对点式的分布式存储系统很好的解决了存储系统的扩放性和容错性，但是却没有能够为上层应用提供足够快速的数据访问能力。RamCloud[2] 是斯坦福大学提出的一种完全基于内存的分布式存储系统，首次提出了在云计算环境下完全使用内存构建持久的分布式存储的可行性和基本技术，并且对于读写性能的提高给出了非常乐观的估计。RamCloud 主要解决的是数据备份和恢复在基于内存的系统中如何实现的问题，在其后续文章 [15] 中，作者提出了异步备份，并行恢复的策略，被认为能够有效的解决内存存储系统的持久性问题。

分布式存储系统作为云计算基础软件架构的核心组件之一，在云计算时代到来之初面临了一系列重大的改革，然而这种变革依然在继续。我们知道传统的分布式存储系统把数据存储在网络中不同服务器的硬盘中，这样当用户发起

一个请求时首先将请求分拆，通过网络并发请求给不同的服务器，服务器从本地硬盘中检索并读取数据，最后通过网络返回到请求节点中。根据这个流程，图 1.1粗略估计了当前工艺下普通 PC 磁盘以及不同网络条件下，一次请求的时间估计，从中可以容易的看到磁盘寻道 (平均 10ms) 在请求的数据量较小 (1Kb) 的情况下所花费的时间远远超过了网络传输和磁盘数据传输的时间；随着传输的连续数据量逐渐变大，寻道时间占据的比重越来越低。这也就意味着，当前分布式存储系统的速度极限受到磁盘本身性能的限制，也受到了数据访问模式的限制。当主要的数据访问是随机的小数据访问时，保证系统性能就成了不可能的事情。然而在实际应用中这种对小数据的海量存储访问的模式却普遍存在着。比如大量传感器构成的传感器网络，它们不断搜集着数据并且需要持久存储。在这种情况下，一个能够存储海量小数据，并且支持随机访问的云平台下数据访问将极大的提高云计算平台的应用范围。

表 1.1 数据中心请求各个步骤的时间估计

请求的连续数据	网络传输时间 *(1Gb/10Gb)*	寻道时间	磁盘读取时间	总时间
1Kb	$10^{-6}/10^{-7}$s	10^{-2}s	10^{-5}s	$\approx 10^{-2}$s
1Mb	$10^{-3}/10^{-4}$s	10^{-2}s	10^{-2}s	$\approx 2*10^{-2}$s
1Gb	$1/10^{-1}$s	10^{-2}s	10s	$\approx$ 11s

1.2.2　分布式计算模型的发展和新的需求

分布式计算模型是云计算软件体系架构的另一个核心组件，它负责为用户提供基本的编程语义，运行时系统支持，以及任务调度、容错、安全审计等。在云计算之前，大量的分布式系统是通过使用消息传递的方式来实现的，其中比较常用的如高性能计算中常见的 MPI，OpenMPI 等；企业级开发中的 RMI(远程过程调用) 等。2006 年 Google 提出了一种通用的用于大规模分布式计算的 MapReduce[16] 模型。该模型将用户程序抽象成为 map 和 reduce 两个阶段组成的一轮计算组成，多轮计算得到最终结果，运行时系统会自动将 map 和 reduce 拆分到不同的服务器执行，并且支持错误恢复，动态任务调度等功能。MapReduce 模型和 GFS 一起构成了 Google 的大规模计算的基石。Hadoop 项目中包括了一个 Java 实现的 MapReduce 模型，用来和 HDFS 一起工作以对大规模

数据进行处理。MapReduce 是一种批处理模型，map 和 reduce 都需要较长的时间启动执行，map 和 reduce 之间还存着这强制的数据同步。根据 Google 内部的统计，在 Google 的数据中心中每天有超过 1 千个 MapReduce 在运行，短的需要几十秒，长的可能需要运行一个星期，而平均的运行时间超过 10 分钟。

鉴于 MapReduce 的批处理特性，其不支持对数据的实时处理和持续处理。为了支持大规模实时应用，来自 Yahoo! 的数据分析科学家提出了 S4[17] 流式计算模型。该模型将应用程序分拆成基于 actor 实现的小的计算单元，将数据组织成流通过这些计算单元并且实时的得到结果。类似思路的系统还包括了来自 Twitter 工程师提出的 Storm 系统，该系统的逻辑架构类似 MapReduce，它将计算拆分成'spouts' 和'bolts' 两部分，前者消耗流式数据，后者产生新的数据流。通过将数据引入'spouts' 来启动 storm 的执行。流式计算的一个典型缺陷在于容错的复杂性，由于流式计算任务常驻，任务之间的关系通过数据流产生和维持，当节点传输时数据丢失或者节点计算过程中失败，任务和当前计算的状态很难在别的节点上恢复，并且恢复时间过长的话更会导致系统长时间等待。在实际应用，Storm 这样的实时计算模型通常和 MapReduce 模型一起工作，这样即便计算出现错误，最终也能通过 MapReduce 任务返回正确结果。

Spark[18] 是由 Berkeley AMP 实验室提出的分布式计算模型，基于 Scala 语言实现。同样是将用户程序拆分成 map 和 reduce 两个阶段，Spark 通过 RDD 的概念极大的提高了 MapReduce 的任务执行速度，其性能在全内存的支持下甚至可以打到 MapReduce 任务的 100 倍。RDD(分布式离散数据集) 是对输入数据的一个抽象，用户在构造 RDD 的时候，Spark 自动的将相关的数据载入到内存中，这个数据集类似 GFS 一样不能修改。所有基于该数据集的 map，reduce 任务都可以更快的完成。GraphLab[19] 则是一个专用于图处理的计算模型，它将分布式的计算抽象成对图中节点和边的处理。每一个节点包含一个计算过程，不同计算过程之间的数据传递则沿着节点之间的边移动。通过定义不同层次的读写锁，GraphLab 克服了传统的图处理模型只能描述异步程序的限制，成为了一个通用的分布式编程模型。当然除了它们之外还有很多计算模型不断出现，比如 Dryad[20], Piccolo[21]，Pregel[22] 等。

从云计算编程模型发展的历程来看，最开始简单的 MapReduce 模型专注与如何将复杂的应用抽象成分布式的基本组件，并且能够在大量的服务器上容错

的运行。然而随着云计算环境中应用种类的增加，MapReduce 这样简单的模型对于复杂的实时应用、迭代和递增应用的不适应性越来越明显。比如电子商务网站的推荐系统，它们能够根据用户的历史行为为用户推荐有需求的商品。这个算法本身就包含了迭代直至收敛的计算过程，再加上用户的行为在浏览网站的时候也在不断变化，如果希望更准确的预测用户的需求就需要考虑这些最近发生的行为。这样的问题在 MapReduce 模型上确实无从求解，如何提出并且实现一个高性能的云平台的计算模型能够支持这类应用就是本文中计算模型研究的根本出发点。

1.3　本文的研究内容

我们希望提高云计算平台中基础软件架构的性能以使其能够支持复杂的实时应用的需求。在分布式应用中，*实时*并非指任务必须在某个时间点完成，而是根据 SLA 的要求尽快给出结果的软实时。比如用户载入自己的社交网络首页，服务器应该能在 500 毫秒时间内返回给用户完整的页面。这里所说的*复杂*不是指算法设计上的复杂性，而是是指计算过程的复杂性。在分布式系统中，如果一个计算过程内部没有依赖关系并且不需要全局信息，这个计算就可以被拆分成子任务并行执行并获得最大的加速比，可惜的是，在大部分情况下都是不可能的。那么计算过程复杂性的区别就体现在了依赖关系的复杂性和所需的全局信息的数量上。复杂的应用指的是这样一类计算过程：在应用计算的过程中，每一个阶段都需要来自全局的信息，并且该全局信息是不断变化；与此同时，计算包括多个迭代，迭代之间具有数据依赖关系；更加复杂的情况是，果输入数据集在计算的过程中不断改变，在这种场景下试图得到实时的结果，是现有云计算基础软件架构所不能完成的工作。

前面我们提到了*迭代计算*和*递增计算*的概念。迭代计算 (iterative computation) 指应用对同一个数据集进行多次处理直到收敛 (达到某种条件)，这其中包含了对重复的数据集进行重复的计算过程。递增计算 (incremental computation) 指应用对不断变化 (递增) 的数据集进行多次处理直到收敛 (达到某种条件) 的过程，这其中包含了变化的数据集和基本重复的计算过程。这两种计算的模式在非常多的算法中都作为核心过程出现，比如 PageRank[23] 中的迭代收敛过程以

及典型的 MLDM(Machine Learning, Data Mining) 算法。

本文的研究内容集中在对云计算基础软件架构特别是分布式存储和计算模型的研究上。我们首先通过对现有云计算基础平台的优化配置来提高现有系统性能 (二)；在认识到应用的需求和系统瓶颈后，通过对现有系统的重新设计，为大规模的实时计算提供完整的软件基础架构支撑 (三, 四)。我们的主要应用对象为：需要在快速相应的在线的、包含复杂的迭代甚至递增计算模式复杂的的大规模应用程序。其中比较典型的应用包括：社交网络、数据挖掘算法、机器学习算法、深度神经网络算法等等。

1.4　本文的主要贡献

本研究的主要贡献可以归纳为以下两部分:

- 本文通过对完全基于内存的分布式存储系统 Sedna 的设计和实现为云计算中实时应用提供了远远超过传统分布式存储系统的数据访问速率。在设计和实现这一新型的分布式存储系统时，提出并实现一种基于层次化的数据中心存储系统架构；提出了一种基于动态可调整的分布式哈希方案提高了存储节点的均衡；提出并实现了新的专用于实时应用的数据访问 API。

- 本文首次提出了一种完全基于触发器的云平台下的分布式编程模型 Domino。面对触发器模型带来的种种挑战，本文提出了最终同步的概念使得应用可以在不进行同步等待的前提下完成同步计算；结合触发器模型，本文还设计了一套完整的容错和恢复模块，以较小的资源使用率使得触发器的执行可以实时恢复。

1.5　论文结构

本文的组织结构如下:

- 第二章我们介绍一种通过模糊算法自动的优化异构 Hadoop 集群的方法。通过对已部署的以后 Hadoop 集群的配置的自动优化，我们能够在不改变软件架构的基础之上提高其性能。

- 第三章我们将介绍基于内存的分布式存储系统 Sedna。该系统是我们设计并实现的适用于实时云计算平台的键值存储系统。比较现有的分布式存储系统，Sedna 提供远超过现有存储系统的数据访问速率，更好的扩放性以及容错恢复的速度，并且支持特有的实时数据访问接口。
- 第四章我们将介绍一种基于触发器的分布式编程模型 Domino。该模型作为一个通用的分布式编程模型，不仅通用与多种应用，更是特别适合迭代和递增计算的实时应用。相比较现有的实时计算模型，Domino 不仅仅具有更快的执行速度、更好的对递增计算的支持，它更是提供了完善的错误容忍和恢复的功能，能以最小的粒度实时的恢复出错的程序。
- 最后在第五章对本文进行了总结。

第二章　基于模糊逻辑的异构 Hadoop 集群配置优化

在本研究课题中，我们使用模糊逻辑的算法，根据正在运行 Hadoop 系统的异构集群中服务器的各种硬件参数作为模糊输入，设计并且实现了一个自动的对大规模云计算异构集群进行自动配置的工具。该工具在集群运行的过程中一直搜集集群运行状态并且根据这些历史数据作为参考来进行模糊分析。通过模糊逻辑的方案进行动态参数配置改变了传统 Hadoop 集群的配置方法，将过去优化参数的方法转变成了优化规则的方法。我们相信，相比较优化参数，优化规则更能够适应不同的部署环境，更加具有通用性。实验证明，本方法不仅仅降低了云计算异构集群的维护成本，还极大的提高了任务的执行效率，进一步提高了 Hadoop 集群的速率。

2.1　问题介绍

Hadoop 项目 [12] 包含了 Apache 基金会下的一系列开源项目：MapReduce、HDFS、HBase[24]、Hive[25] 等。由于它的完整性和高可用性，一出现就成为了工业界和学术界进行大规模计算研究和应用的标准平台。大量的公司 2.1内部往往运行着数以万计的服务器组成的 Hadoop 集群来完成各种各样的工作。

表 2.1 典型的 Hadoop 集群的规模 [1]

Institution	*Scale*
Ebay	532 节点，4256 核心
Facebook	1100/300 节点，8800/2400 核心
百度	2000 节点，超过 8000 核心
Yahoo!	4000 节点，超过 16000 核心
...	..., ...

随着云计算的发展，大规模的数据中心越来越多，越来越多的公司需要管理这些大量的服务器。这些由廉价个人计算机组成的集群由于设备淘汰率高，更换设备的频率也更高，从而使得集群内部大量存在着异构的服务器。这些异

构的服务器有些拥有不同的硬件配置，有些运行着不同版本的操作系统，有些甚至可能运行在不同体系结构下。那么在每次更新或者升级数据中心之后，如何动态的对 Hadoop 集群进行重新配置以使达到最佳的执行效率就成为数据中心维护人员的一项重要工作。

Hadoop 本身的对集群配置的功能非常弱，配置往往只能在安装时进行一次配置，之后所有的配置都通过同步的方式传输到所有节点中，并在启动后开始起作用；如果管理员希望在 Hadoop 集群运行的过程中修改某些参数，那么只能要求集群的使用人员在提交任务的时候以命令行参数的形式将新的参数传入 Hadoop。但是这种方法只能修改整个 Hadoop 可配置参数集中少量的部分。如果希望对别的参数进行修改则需要重启整个集群，这在大规模的生产环境中是不可以接受的。

除此之外，Hadoop 本身作为一个复杂的分布式系统软件，其中包含了大量的可配置参数，比如版本 0.19.2 中包括了大约 170 个配置参数，而这些参数中很大一部分对所部属的 Hadoop 性能有较大影响。从 Hadoop 项目的邮件列表中也可以看出，在日常使用中如何根据自己集群中服务器的实际配置选择合适的参数是用户在配置集群时面临的主要问题，而现有的解决方案则是通过一些艺术化的指导原则来进行的 [26]，比如：*如果集群服务器数目较大，应该将 dfs.namenode.handler.count 配置更大的值 [27]*。这种指导性的配置方法显然不适合大规模的集群管理，特别是当集群中存在异构服务器的情况下。

2.2 技术背景

2.2.1 模糊逻辑

模糊逻辑由 Berkeley 大学的 L.A.Zadeh 与 1965 年引入的一种基于模糊集合论对布尔逻辑的扩展理论。首先我们简单描述一下模糊集合。在集合论中，我们称一个元素是否属于某集合是确定的，也就是说它要么属于，要么不属于，此为是非二元判断，不过在模糊逻辑中对于元素对集合的从属关系却可以描述为部分属于集合。并且元素属于集合的程度用可以隶属度 (likelyhood) 来描述。隶属度函数是一个 0-1 之间的集合成员关系值，在自然语言中，通常可以被描述成为*稍微*，*非常*等概念。常见的隶属度函数如：三角隶属度函数、梯形隶属

度函数和高斯隶属度函数等。

与布尔逻辑类似，模糊逻辑也定义了一系列逻辑操作：IF/THEN，AND, OR，和 Not 等。其中 AND，OR，NOT 被称为 Zadeh 运算符，它们的定义如公式 2.1所示：

$$\begin{aligned} NOT \quad x &= (1 - truth(x)) \\ x \quad AND \quad y &= minimum(truth(x), truth(y)) \\ x \quad OR \quad y &= maximum(truth(x), truth(y)) \end{aligned} \tag{2.1}$$

而 IF/ELSE 规则相比较布尔逻辑也有所不同，它的 IF 需要根据元素对多个模糊子集的隶属度大小来最终判断是否属于某一个集合。

模糊逻辑作为不确定推理的一种，在人工智能中具有非常重要的意义，特别对于复杂的工控和专家系统来说，它的效果非常好。对于复杂的 Hadoop 集群的自动配置来说，模糊推理是一个可行的方案，主要原因在于影响 Hadoop 集群的参数众多且关联性强，无法建立一个准确的数学模型来描述。

2.2.2　Hadoop 参数分析

Hadoop 0.19.2 中共提供了 170+ 个可配置的参数，这些参数中大部分是对于系统能够正常工作的配置，比如 NameNode 的端口配置，ip 配置等等，在本文中我们对于这种不需要改变，并且不会影响到集群性能的配置不予考虑，我们主要分析的是那些对系统性能会产生较大影响的参数，经过分析共有超过 70 个此类参数。我们按照他们的作用时间将其分为 3 个类：

- 在 Hadoop 系统启动的时候会读取一系列初始化参数并且存储在每一个服务器的 Hadoop 进程所在的地址空间中，这些参数在 Hadoop 运行中都不会再次读入，如果需要对这些参数进行改变，现有系统的唯一做法是重启 Hadoop 集群。

- Hadoop 启动之后主要向客户端提供基于 HDFS 的数据访问功能和基于 MapReduce 的任务执行功能。MapReduce 中任务执行的基本单位是 Job，Job 由用户提交的可运行的代码和配置文件组成，Hadoop 集群在调度 MapReduce 的 Job 时会向 JobConf 中加入系统的配置参数，这些参数会被

用户建立 Job 时写入的配置覆盖，通常情况下这些参数会改变任务执行效率。

- 参数仅仅在一个 Job 中的多个 Task 中起作用。Task 是比 Job 更底层的概念，一个 Job 通常包括两类 Task：Map 任务和 Reduce 任务，而且每一类 Task 在系统中都同时运行着多个实例。这些 Task 运行在服务器上的 Task Slot 上，而每一个 slot 其实就是一个 JVM 进行。每一次家在 Task 执行的时候，系统都会动态的加载这些参数。

通过对参数生命周期的分类可以将我们的主要研究目标放入到生命周期为 Job 的参数，这些参数与 Hadoop 的基本组件相关，并且也会对系统性能产生显著的影响，我们对这些参数进一步细分可以得到如下类型：

- 定义时间类型的参数。这些参数定义了事件发生的最小或者最大时间。如果设置的过小，可能会导致系统过早的因为意外而进入所谓的伪失败状态，而如果设置过高，则容易导致系统对错误响应不及时。

- 定义最大尝试次数。很多参数会定义网络行为中最大尝试次数，重复次数的多少通常体现了系统对于容错能力的考量。

- 定义系统并发度。并发度包括了网络发送的并发度、任务执行的线程并发度、网络接受数据的并发度等等。它们描述了 Hadoop 集群的能力，特别是集群规模扩大的时候，并发度也应该随之提高。

- 定义资源使用限额。比如允许 map 任务在本地内存中缓存数据的上限，这些参数定义了系统对硬件资源的占用情况。这些参数跟服务器的硬件资源情况、集群的拓扑结构和网络贷款都有非常紧密的关系。

表2.2展示了 Hadoop 参数的具体分类情况。

2.2.3　Hadoop 参数加载流程

在介绍 Hadoop 加载参数之前我们先对 Hadoop 架构做一个简单的介绍。图2.1给出了包括 HDFS 和 MapReduce 两个模块的标准 Hadoop 集群的架构。主节

表 2.2 Hadoop 中参数的统计分析情况

参数类型	相关参数数目	典型参数实例
时间类	10	fs.checkpoint.period dfs.df.interval dfs.heartbeat.interval
重复次数	11	dfs.replication mapred.map.max.attempts
并发类参数	14	io.sort.factor mapreduce.reduce.parallel.copies
资源使用类	25	io.sort.mb mapred.child.ulimit
其他	5	mapred.jobtracker.taskScheduler

点上同时运行了 *NameNode* 和 *JobTracker* 两个服务，前者负责为分布式存储系统 HDFS 提供元数据位置信息，后者为 MapReduce 任务执行系统提供任务管理和分配服务。主节点之外的子节点运行着 *DataNode* 和 *TaskTracker* 服务，分别负责数据存储和任务执行。当 Hadoop 集群启动的时候，首先会启动主节点，之后向所有的子节点同时发送启动命令，子节点启动时需要向主节点请求元数据信息。

在这个启动过程中，Hadoop 集群中的每一台服务器都需要载入配置参数。这些配置参数并未维护在一个集中的位置而是存储在每一台服务器上。用户通过手动的同步操作将配置文件同步到集群的所有服务器上。在每一台服务器上，*NameNode/DataNode* 会首先启动，在本地配置文件在进程空间里生成一个 Configuration 对象，紧接着 *JobTracker/TaskTracker* 启动，生成 JobConf 对象。当用户编写一个在 Hadoop 上运行的 MapReduce 应用程序的时候，都需要生成一个 JobConf 对象并且传递给 *JobTracker*，此 JobConf 中的配置会覆盖掉之前 *JobTracker* 加载的 JobConf 对象；最后，当用户的任务已经提交给 *JobTracker*，那么集群中的 *TaskTracker* 都会通过信条协议向 *JobTracker* 请求最新的可执行的任务，当获得任务后将根据其种类 (Map 任务或 Reduce 任务) 来启动对应的任

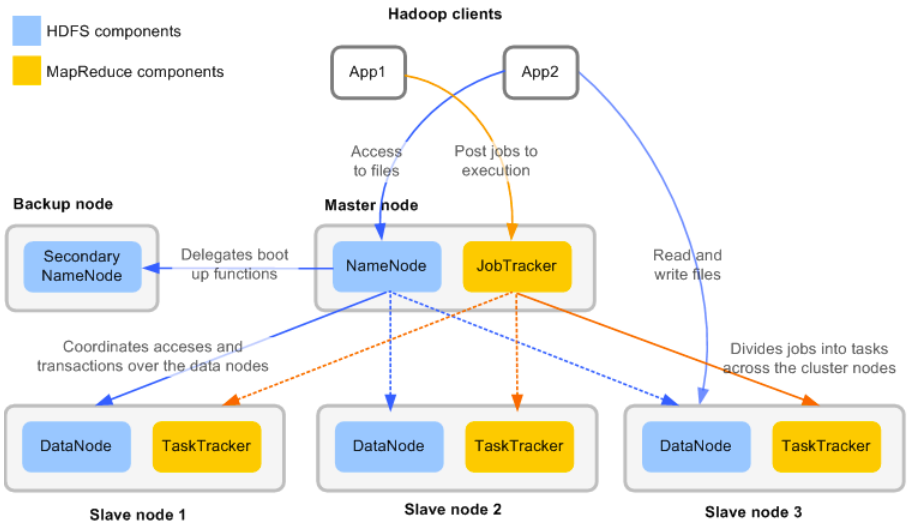

图 2.1 Hadoop 系统架构

务，启动的过程中会生成一个 JobConf 对象，该对象加载 *TaskTracker* 本地的配置文件以及用户提交的 JobConf 中的数据。

事实上，动态的对于那些已经加载到 Hadoop 运行时系统中的参数进行修改非常困难。比如参数 *mapred.job.tracker.handler.count* 值会决定 *JobTracker* 中启动的服务线程的数目。而这些服务线程是在 *JobTracker* 启动的时候开辟的线程池。因此即便我们在运行时修改了这个参数，也不会对当前运行中的系统行为产生任何影响。本文中，对 Hadoop 集群参数的修改是通过将新生成的参数写入到本地配置文件中，在下一次启动的时候让 Hadoop 自动加载实现的。

2.3　基于模糊逻辑的配置算法

2.3.1　运行时数据搜集

我们通过修改 Hadoop 源码以及分布式的资源检测工具来获得集群的历史运行数据和所有服务器的硬件指标。并且使用这些信息作为模糊逻辑方案的输入数据来来计算参数。

- 所有节点的 Map 运行时间的平均值和本节点 Map 任务执行时间的差

- 所有节点的 Reduce 运行平均时间和本节点上 Reduce 任务执行平均时间差
- 服务器在执行任务过程中的 CPU 平均负载

同时我们也需要集群中服务器的硬件指标，包括:

- 服务器的 CPU 频率，以及核心数目。
- 服务器的内存容量
- 服务器的硬盘参数
- 服务器的网络带宽以及接入机架信息

2.3.2 模糊控制器的实现

2.3.2.1 隶属度函数

我们使用了混合的隶属度函数来描述 Hadoop 集群的各种硬件信息，其中包括高斯型隶属度函数、梯形隶属度函数。梯形隶属度函数能够更好的体现离散输入的特点，比较适合 Hadoop 集群中服务器硬件指标数据。

三个隶属度函数分别为:

$$\begin{aligned} y_{Slow} &= gaussmf(x,[sig,c])[sig=2,c=0] \\ y_{Average} &= trapmf(x,[a,b,c,d])[a=3,b=4.5,c=5.5.,d=6.2] \\ y_{Flat} &= smf(x,[a,b])[a=4.5,b=9] \end{aligned} \tag{2.2}$$

其中高斯和梯形隶属度函数的表达式如下:

$$\begin{aligned} f(x;\sigma,c) &= e^{\frac{(x-c)^2}{2\sigma^2}} \\ f(x;a,b,c,d) &= max(min(\frac{x-a}{b-a},1,\frac{d-x}{d-c}),0) \end{aligned} \tag{2.3}$$

实际使用中，服务器 CPU 频率的变化范围设定为 [0,16GHz]。当前主流的 CPU 为 4 核，单核频率 2GHz，因此 8GHz 作为中间值，超过 16GHz 的单服务器计算能力称为 很快的 CPU。模糊子集分别为 *Slow, Average, Fast*，其隶属度函

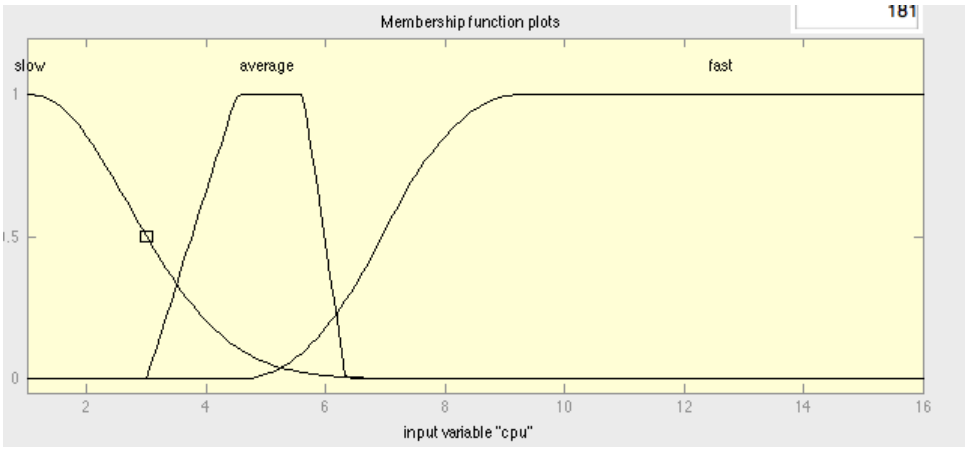

图 2.2 对 CPU 建模的隶属度函数分布图

数在 Matlab 中可以表示为图2.2所示。单机 CPU 频率无上限，因此使用 S 型隶属度函数来描述 *Fast*。

之后分别对内存，网络进行建模，生成如下隶属度函数图 (2.3和2.4)

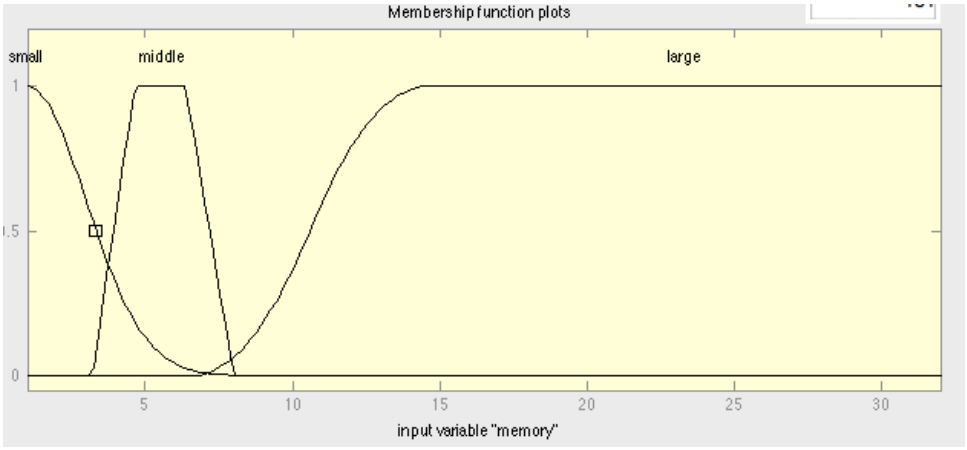

图 2.3 对内存建模的隶属度函数分布图

另一个输入来源就是执行的历史信息。在本文中我们认为任务执行的历史信息是由多种原因造成的，特别是临时的错误，调度器的优先级等。因此它们并不适合作为参数直接来影响模糊推理的结果，我们用它们作为最后的微调参数。

模糊逻辑的输出值为 Hadoop 配置参数中的值，我们在这里考虑了 10 个对系统性能影响较大的参数 [26–28].

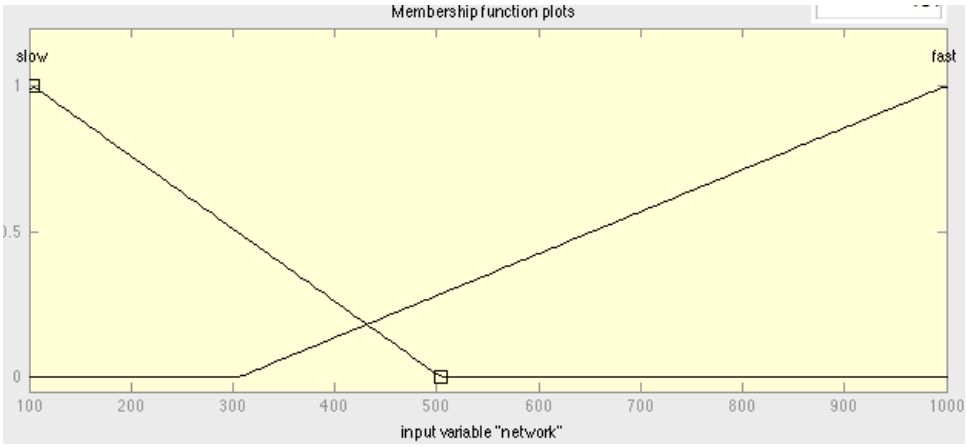

图 2.4 对网络建模的隶属度函数分布图

表 2.3 模糊逻辑输出参数列表

参数名称	取值范围
io.sort.mb	10-48-96-192-384-768 M
io.sort.factor	10-40-80-120-240
io.sort.split.percent	0-0.2-0.4-0.6-0.8-1.0
mapred.reduce.parallel.copies	1-5-10-15-20
mapred.min.split.size	64-128-256-512 M
mapred.tasktracker.reduce/map .task.maximum	1(1)-2(3)-3(5)-4(6)-5(9)
dfs.block.size	64-128-256-512 M
mapred.map.tasks.speculative.execution	False -> True
mapred.reduce.task.speculative.execution	False -> True
mapred.job.shuffle.merge.percent	0-0.2-0.4-0.6-0.8-1.0
mapred.job.shuffle.input.buffer.percent	0-0.2-0.4-0.6-0.8-1.0

2.3.2.2　模糊推理规则

通过系统的调研 Hadoop 邮件列表中关于优化配置的参数和场景，我们为模糊算法设定了合理的模糊规则。由于输入变量数目较多，输出变量更多，我们仅仅列出其中一个模糊规则作为示例。*io.sort.mb* 指 Hadoop 的 MapReduce 任务执行过程中对键值对进行排序时候可以使用的内存 buffer 的容量，单位为 MB。*io.sort.mb* 的隶属度集合如图2.5所示。

其对应的规则如图2.6所示。

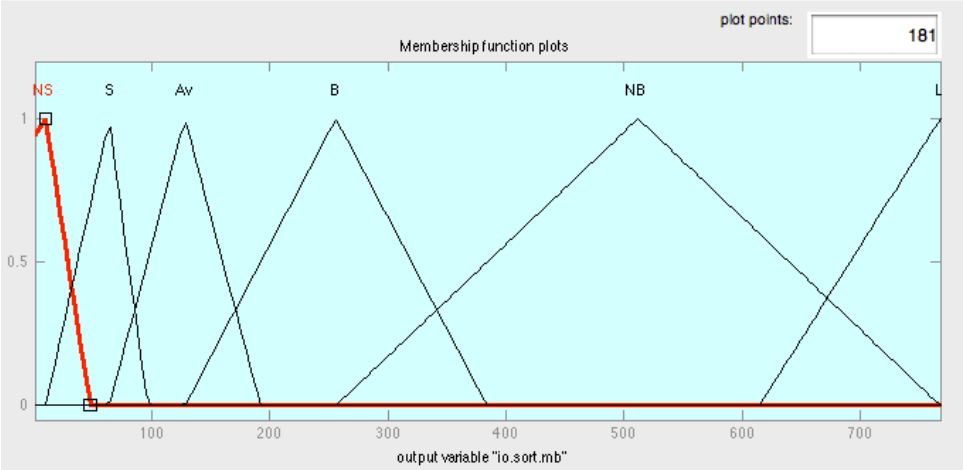

图 2.5 *io.sort.mb* 的隶属度函数

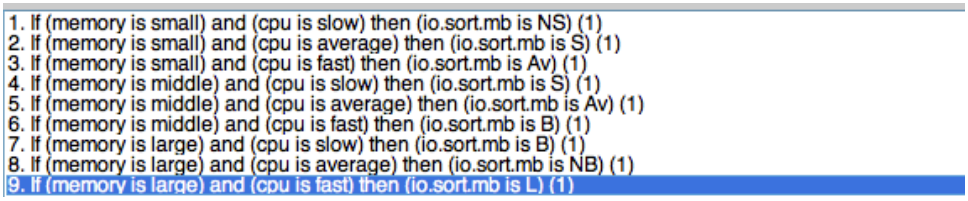

图 2.6 对 *io.sort.mb* 的模糊规则表

2.4　实验和分析

2.4.1　Matlab 仿真

利用 Matlab 实现的模糊控制系统架构图 (FIS) 如下图2.7所示。利用 sufview

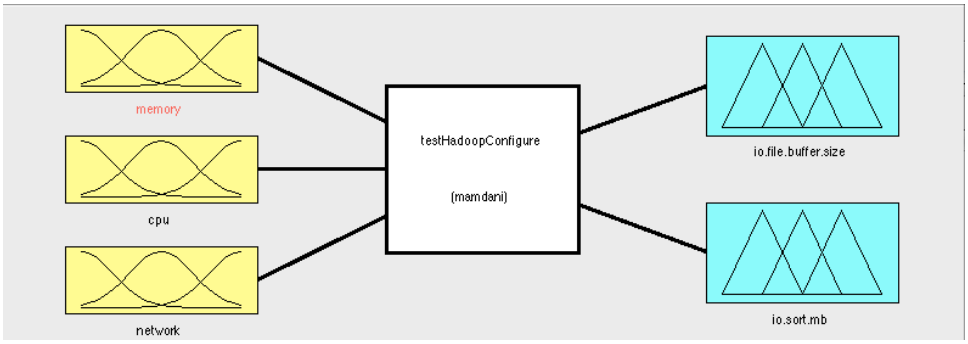

图 2.7 系统的模糊控制系统总体图 (仅列出了三个输入参数和两个输出参数)

可以得到 *io.sort.mb* 和服务器中内存、CPU 参数的输出曲面，图2.8所示。

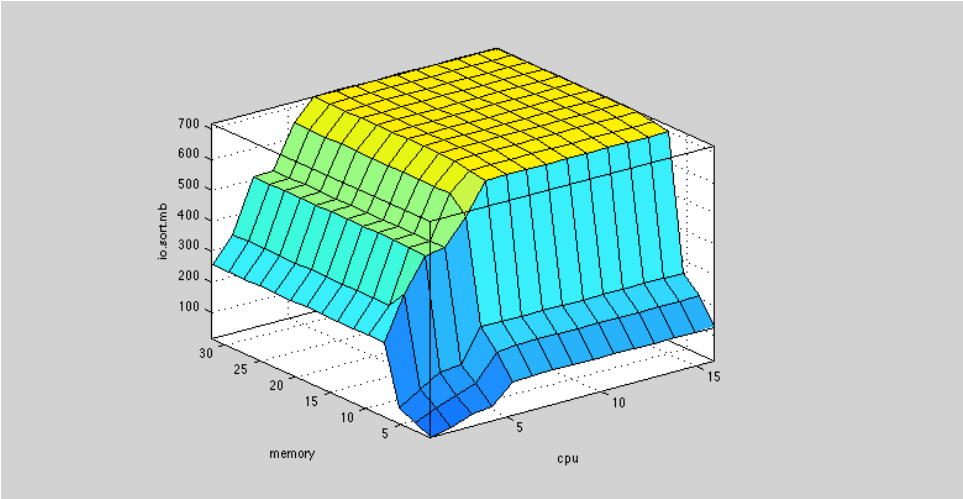

图 2.8 参数 *io.sort.mb* 的模糊规则输出三维示意图

2.4.2　Hadoop 集群实验

通过修改 Hadoop 的源代码，我们为 *JobTracker* 和 *TaskTracker* 实现了自动配置代码，通过检查历史运行数据来判断是否需要重新配置，并将根据模糊逻辑生成的参数写入到本地配置文件中。

2.4.2.1　同构环境实验

所有的实验在一个由 4 台服务器构成的 Hadoop 集群上进行，其中 3 台 slave 节点，一台 master 节点，所有节点性能配置如表2.4所示。

表 2.4 同构集群的节点参数

属性	参数
CPU	Xeon W3530 2.53GHz
Memory	6G 1066MHz ECC
Disk	1T SATA Raid 1
Network	100MB Ethernet

都运行 Hadoop 发行版中自带的 Terasort 应用，之后使用自动化配置后得到的参数再次运行 Terasort。输入数据为 25GB，每个实验执行 10 次求平均值，最后结果如表2.5所示。

表 2.5 运行 25G Terasort 所花费的时间

	Default	*Autoconfigured*
Tera-Sort	44min 376maps, 16min 1reduce, 43min	19min,40s 94maps, 1min 3reduce, 19min

采用默认配置的 Hadoop 集群运行一次任务需要 2640s，其中共运行了 376 个 map 任务共耗时 960s，一个 reduce 任务，耗时 2580 秒。采用我们的模糊推理得到的参数运行同一个任务只需要 1180s，其中公允性了 94 个 map 任务耗时 60 秒，3 个 reduce 任务，耗时 1140 秒，加速比为 2.23. 此实验结果表明使用模糊逻辑配置的工具能够得到较好的集群性能。

2.4.2.2 异构环境实验

我们通过认为降低集群中某一台服务器的内存，并且使其只启动一个 CPU 核来构造异构的环境，该服务器的配置如下表2.6所示。

表 2.6 同构集群的节点参数

属性	参数
CPU	OneCore 2.53GHz
Memory	2G 1066MHz ECC
Disk	1T SATA Raid 1
Network	100MB Ethernet

同样使用 Terasort 作为基准测试，这次我们将和之前同构环境下测试时生成的配置文件进行比较。执行 10 次后平均时间如表2.7所示。

表 2.7 运行 25G Terasort 同构和异构配置的性能对比

	homo-configured	*heter-configured*
Tera-Sort	33min 94maps, 11min 3reduce, 31min	27min 94maps, 7min 3reduces, 25min

结合表2.5和2.7，我们可以得出结论，采用同构环境下计算出来的结果来配

置异构集群会使得集群性能降低，并且这个性能降低 (从 19min 到 33min) 远远超过了一台服务器性能折半带来的性能降低。这说明了在异构环境下使用正确的参数配置的重要性。通过实验结果也可以看出来，采用异构配置的集群其性能明显优于同构的配置参数。

2.5　小结

在本研究中，我们使用模糊逻辑的方法对大规模部署的异构 Hadoop 集群进行参数配置，并且通过试验证明了本方法能够明显的提高 Hadoop 集群中 MapReduce 任务执行的速率。然而面临着不断出现的对系统反应时间有着更高要求的应用，Hadoop 和 MapReduce 模型本身的设计上的限制决定了其无法适用于这类应用。从下一章开始，我们将介绍新型的分布式存储系统以及新的计算模型来应对实时应用的需求。

第 三 章　基于内存的分布式键值存储系统

3.1　引言

云计算平台上的存储系统和传统的网络存储系统不同，它不仅仅要对外服务接受来自客户端的读写请求，更重要的是为云计算平台中运行的分布式任务提供并发的数据读写能力。图3.1展示了一个典型的云计算环境下，数据中心内部存储的架构图。存储通过大量的廉价个人计算机构成，存储系统不仅仅接受来自外界的请求，更需要支撑数据中心内部运行的应用程序的读写请求。

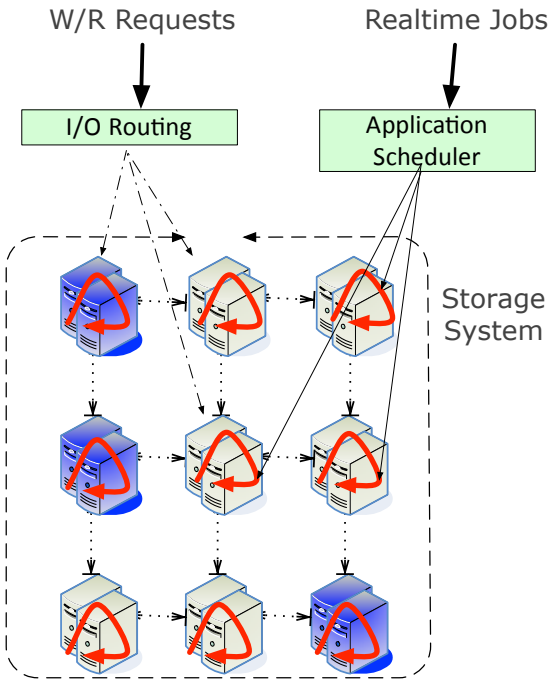

图 3.1 典型的云计算环境下数据中心中存储架构图

为了存储各种不同类型的数据以及支撑不同类型的应用程序，在云计算领域出现了多种分布式存储系统。从早期的类似 Google File System[29] 的块式系统，它们适合与存储基本不会被修改只会被追加的较大的文件，适合连续

读取；到后来出现了分布式的键值存储系统，比如 Dynamo[13] 等用来存储较小的数据，较好的支持随机读写；以及后来被称为 NoSQL 的结构化存储系统，如 Bigtable[11]、HBase[24] 以及 Cassandra[14] 等，它们用来存储具有结构信息的小数据，这些数据按照表的方式组织起来，表中每一个单元存储的数据都较小并且支持随机的读写；在对存储读写性能要求苛刻的场景下，Redis[30]，Memcached[31] 等分布式缓存系统得到了极大的应用，它们通过在内存中构建不持久的哈希表为应用和用户提供高速数据访问能力，它们不能脱离前面所讨论的分布式系统而存在；与内存缓存系统类似，RamCloud[2] 是一个基于内存的存储系统，但是它却能够提供持久存储的能力，并且具有快速数据恢复能力可以作为独立的高速分布式系统使用。从分布式存储的发展路径可以看出，它始终随着外界应用的要求而发展：云计算早期海量数据一般集中在搜索引擎公司，它们需要廉价且稳定的海量数据存储能力，而对这些数据的处理则是在后台以批处理的方式进行，因此只需要提供高性能的顺序读功能即可，因此 GFS 完美的负担起了这一角色。然而随着社交网络，电子商务，在线广告系统等产品的发展，人们开始发现那些小的、零散的原始数据现在已经成为应用程序最主要的数据来源。比如 Twitter 这个流行的微型博客系统，主要存储的就是不超过 140 个字符的短消息，然而在 Twitter 当前规模下，每天都会有超过 4 亿用户从世界各地提交超过超过三千五百亿条短消息；不仅仅数据变成了小数据，应用对数据读写的频率和速度的要求也远远超过了传统应用。在这种情况下大量的类 Memcached 系统被应用在云计算平台中，而由于其数据的不稳定性，类似 RamCloud 这样的持久分布式内存存储也就成为了那些需要应对实时应用的分布式存储系统的最终归宿。

在本章中，我们将介绍一个完全基于内存实现的持久化的分布式键值存储系统——Sedna。面对我们的应用对存储系统的存储容量、随机访问速度、持久性、数据一致性的要求带来的挑战，Senda 最终能够为用户提供一个简单、有效的接口以支持各类实时应用。在3.2中，我们首先介绍存储系统的相关工作，介绍内存存储的合理性和主要挑战。随后在3.3中详细的描述 Sedna 的架构和实现细节介绍了如何解决这些挑战，在3.4节中通过实验着重分析了 Sedna 的读写性能，在3.5对本章进行了一个小结。

3.2　相关工作介绍

3.2.1　相关分布式存储系统介绍

Google File System 由 Google 公司与 2003 年提出的一种基于块的大规模分布式文件系统，其基本架构如图3.2所示，包括一个 master 和多个 chunkserver 组成。文件系统的层次结构和名字空间都存放在 GFS Master 节点中，客户端应用程序首先通过查询 GFS master 来获得欲访问的文件所在节点信息 (chunk location)，然后通过向特定的节点发送读写请求来完成数据传输过程。

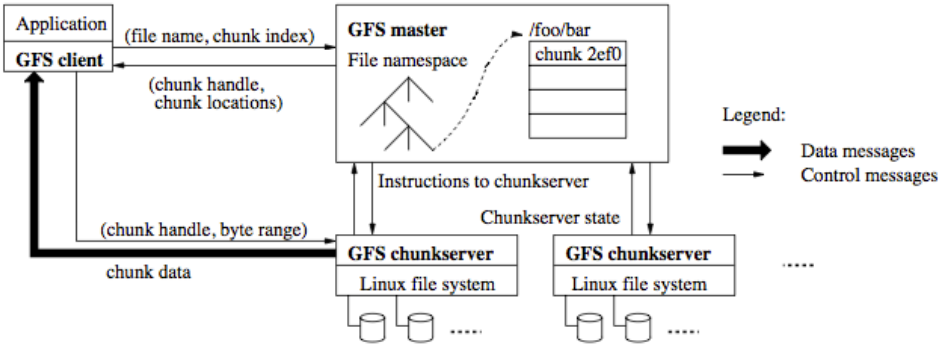

图 3.2 GFS 文件系统的架构

Hadoop Distributed File System(HDFS) 作为 GFS 的开源实现，结构上与 GFS 基本一致，master 和 chunkserver 分别对应 NameNode 和 DataNode。在 HDFS 上，任何一个文件都由预定义大小 (默认 64MB) 的块组成，这些块被冗余存储于多个 DataNode 上（默认情况下，每一个块会存储在三个节点中)。由于 HDFS 多运行在由海量普通计算机构成的数据中心中，节点的失效和网络问题非常普遍，通过多备份数据极大的提高了系统的可用性和容错性。每一个 DataNode 需要和 NameNode 维持一个周期心跳用来检测节点是否失效以及网络是否可用，当 NameNode 发现某节点失效的时候会利用备份节点的数据来提供服务，并且尝试恢复失效节点，以维持每一个数据块三备份的状态。由于每一个客户端在访问 HDFS 中的数据时都需要向 NameNode 查询相关数据所在的 DataNode 位置信息，为了保证多个客户端程序读写的性能，HDFS 的实现中会将所有的命名空间信息都存放在 NameNode 的内存中。HDFS 这种基于数据块的设计方法极大

的提高了系统对于海量文件的存储能力。通过一个 NameNode 的管理，我们可以轻易的管理成千上万服务器构成的存储集群，同时也带来了一些问题，首先是 NameNode 的单点故障特性，其次是单名字空间节点限制了节点的存储容量。

对于单点故障问题，2011 年 Borthakur[32] 提出了使用 AvatarNode 代替 NameNode 和 SecondaryNameNode 的方案，系统架构如图 (3.3) 所示。这样，一个 HDFS 集群就包含了两个 avatar 节点，active avatar 和 standby avatar。任一个 avatar 节点实际上都是一个正常的 NameNode 的包裹。HDFS 文件系统镜像和日志都存放在 NFS 中，而不是本地。活动的 avatar 节点将所有的事务写入在 NFS 中的 log 文件中，与此同时，standby avatar 节点将从 NFS 中打开相同的 log 文件，读入并且不断的将新写入的事务应用到本地存储的命名空间中，从而使得本地的命名空间和当前活跃 avatar 节点上的命名空间保持尽量一致。所有的 DataNode 不仅仅和 active avatar 节点交流还会和 standby avatar 节点交流，这样就使得 standby 的 avatar 节点时刻保持着最新的块位置信息，以保证在活跃节点失效的时候，standby 节点能够在一分钟内开始提供服务。

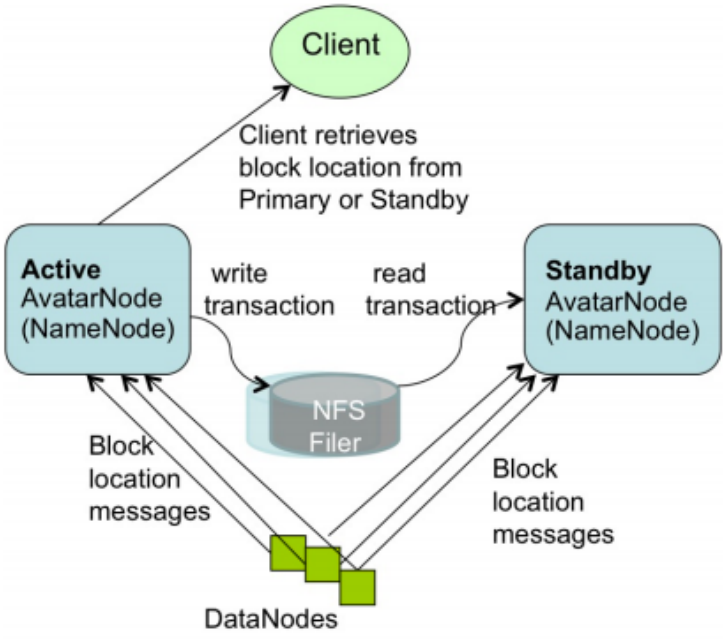

图 3.3 Avatar Node 架构

容量瓶颈问题同样严重，Konstantin V. Shavachko 发表论文详细探讨了

HDFS 文件系统的瓶颈。通过对当时最大的 HDFS 集群的实际运行数据的搜集和分析，给出了单个 NameNode 服务器上内存容量和整个 HDFS 集群存储容量上限的关系：每一个块的位置信息都需要在 NameNode 中占据超过 200 个字节的内存，而在一个文件平均包括 1.5 个块的集群中，每一个文件就大概需要 600 个字节的内存空间。这样，如果要存储超过 1 亿个文件，那么 NameNode 至少就要有 60GB 的内存。如果该 HDFS 集群中每一个块为 128MB，并且备份三分，那么总的存储空间就是 60PB(NameNode 需要 60GB 内存)，这也就意味着 HDFS 的架构不具备线性扩展性。论文 [33] 提出了一种新的 GFS Namespace 服务器的架构。这种新的架构包含了数以百计的 Namespace 服务器，每一个服务器上最多支持超过 1 亿个文件。由于 NameNode 容量的提高，每一个文件可以被划分为更加细粒度的块 (由 64MB 可以减少到 1MB)，用以支持小数据存储。HDFS-dhh 就是该思想的一个具体实现，在该实现中，它使用了 HBase 来代替 HDFS 中的单 NameNode，试验结果表明改进后的 HDFS 集群的写性能可以做到线性扩展，极大的提高了 HDFS 的应用范围。

Dynamo[13] 是 2006 年 Amazon 公司提出的一种高可用的键值存储系统。从存储内容来说，Dynamo 是一种存储键值对的存储系统；从应用场景来说，Dynamo 主要应用于 Amazon 特有的‘永远可写’的场景中。比如在 Amazon 网站的购物车应用中，我们永远希望用户任意一次的“加入购物车”行为都能够成功，不管这次请求时数据中心内不是否发生了错误；从架构上来说，Dynamo 彻底抛弃了 GFS 的单中心节点架构，转而提出了一套完全没有中心节点的 P2P 架构：所有的节点都等价的提供服务，节点和节点之间松耦合：单个节点失效不会影响到别的节点。为了达到近乎苛刻的应用场景需求，Dynamo 整合了一些有效的分布式系统的技术，其中包括：采用动态哈希表和一致性哈希来进行数据划分和复制备份；使用多版本和矢量时钟技术来提供一致性；多副本之间的一致性由仲裁算法来保持一致性；采用反熵 (anti-entropy based recovery) 的恢复策略；采用基于 gossip 的分布式故障检测及 token 协议。

Megastore[34] 是由 Google 于 2011 年公开的应用在公司内部的跨数据中心存储系统。它在保证扩放性的前提下，提供了传统 RDBMS 具备的易用性，包括强一致性保证、高可用性，并且在细粒度的数据分片中提供了 ACID 的语义。这听起来似乎和 CAP 定理 [35] 有矛盾，事实上 Megastore 是通过给应用程序

细粒度的管理数据分片和本地性的能力来避免 CAP 的限制：在 Megastore 中以 EntityGroup 作为基本的独立数据集合，一个 EntityGroup 中的数据并且会在不同的数据中心间同步复制。除此之外，Megastore 还使用 Paxos[36] 协议避免了之前所提到的分布式主节点需要将写前日志 (write-ahead-log) 复制到多个节点的耗时操作。

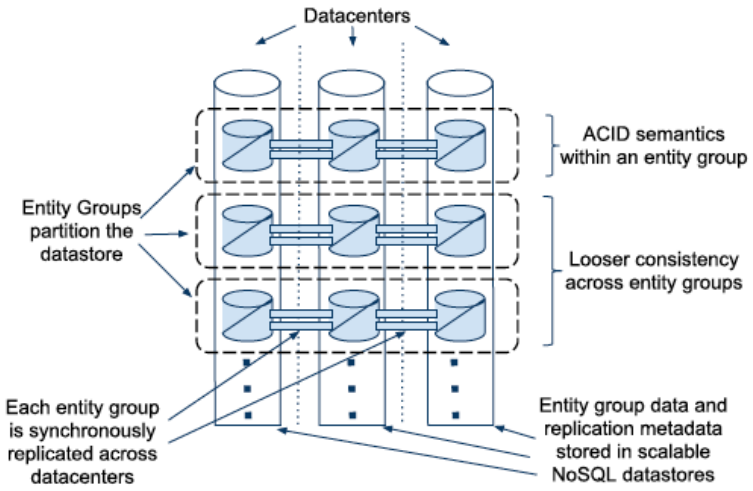

图 3.4 Megastore 架构图

图 (3.4) 展示了 EntityGroup 的结构以及它们是如何在不同的数据中心之间进行复制的。在一个 EntityGroup 内部的所有操作都依赖于 Paxos 算法来确保其 ACID 的语义，而跨 EntityGroup 的操作则依赖于比较笨重的两阶段提交协议，或者使用异步消息的最终一致方案。在一个数据中心内部，Megastore 使用 Bigtable 来做基础的容错存储。Megastore 的创新点主要体现在下面几个方面：1）设计了 EntityGroup 方案给用户提供了足够的一致性的灵活性；2）使用 Paxos 算法来保证 EntityGroup 内部数据的强一致性。3）基于已存的系统：Bigtable 和 GFS 并且和 SQL 模型相结合。尽管 Megastore 在 Google 内部已经开始大规模使用，然而在 Google 之外却没有引起太多的关注，核心原因应在于 Megastore 引入了过多的不是很成熟的优化，使得该架构更像是一个专用存储系统。

3.2.2　内存存储的可行性和相关系统介绍

实时应用在大规模数据处理应用中比重越来越大，这得益于半导体技术的发展使得数据产生的速率越来越高，数据存储的成本越来越低。应用对数据实时性的敏感度也越来越高，同时也就要存存储系统能够提供更高速度的读写能力。面对这些挑战，传统的基于磁盘的存储系统在随机读写上的局限性使其无法应对。因此在实际应用中，大量基于内存的缓存系统被放置在磁盘之上来提供告诉随即读写的能力。比如世界上最大的社交网络公司 Facebook，就大量的使用了 Memcached 作为内存缓存。以 2009 年 8 月的数据来看，大约其所有在线数据的 25% 是保存在 memcached 集群的内存中的，并且提供了 96.5% 的平均命中率。如果再算上数据库服务器的内存缓存，那么整个数据集 (除去图片) 大约有 75% 的数据是存放在内存中的。从这个角度来看，在海量数据的场景下，完全使用内存来作为基本的存储系统是非常可能的。

当然，作为基于磁盘之上的缓存解决方案似乎能够在较高的命中率的基础上以较少的内存代价 (如 facebook 中的数据，25%) 来提供较高的平均访问时间。然而，事实上磁盘和内存之间处理时间之间差超过 1000 倍，使得即便是非常高的命中率前提下产生一次未命中都会导致非常明显的访问延迟，而且 cache 机制带来的冷启动、数据抖动都埋下了性能急剧降低的隐患。

因此在数据中心内部完全使用内存代替磁盘作为主要的存储介质，而磁盘仅仅作为备份存储介质正逐渐成为提高存储系统性能的主要手段。使用内存作为海量存储的主要介质的可能性随着内存价格的逐步降低已经越来越大，图3.5展示了十年来存储每一 Gb 的数据，需要的成本对比图，从图中我们可以看出来，随着半导体技术的发展，DRAM 的成本有了一个极为明显的下降过程，按照这个趋势，单位 GB 内存价格将持续走低，未来单机大规模内存会成为主流。这点，可以结合图3.6看出来，出去数据中心中网络和机架成本，仅仅计算 DRAM 的成本，到 2020 年，我们可以通过假设 4,000 台服务器的集群来提供超过 1PB 内存数据存储能力，而每 GB 的存储成本只有 $6。

RamCloud[2]2009 年由 Stanford 大学的研究组提出的一种将数据中心中全部数据存储在内存中的分布式存储方案。图3.7展示了一个典型的 RamCloud 机群的结构。它包含了大量的存储服务器，每一台存储服务器包含两个部分:

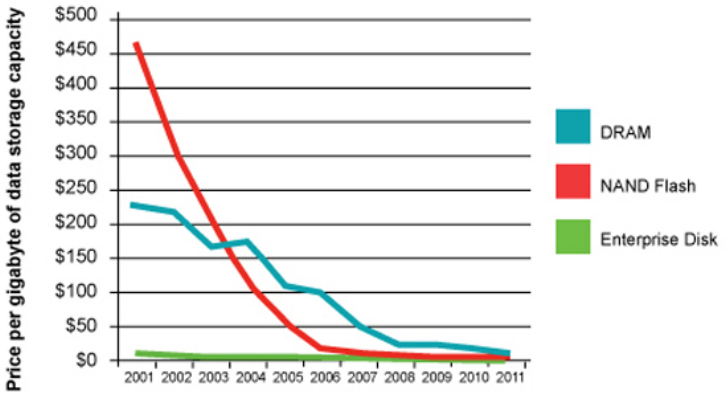

图 3.5 最近十年 (2001-2011) 内存，磁盘，Flash 单位 Gb 的价格趋势分析。

	2010	2020
# servers	2,000	4,000
Capacity/server	24GB	256GB
Total capacity	48TB	1PB
Total server cost	$3.1M	$6M
Cost/GB	$65	$6
Total ops/sec.	2×10^9	4×10^9

图 3.6 2012 年完全使用内存构造海量存储的成本以及 2020 年的预估 [2]

1）主服务，其负责在内存中管理 RAMCloud 对象，并且负责处理客户端的请求；2）备份服务，其存储了来自别的主节点的备份数据，并且将其存放到磁盘和 Flash 中。每一个 RAMCloud 实例都会包括一个单独的服务器，成为协作者 (Coordinator)。其负责管理配置信息，比如存储服务器的网络地址，以及对象的位置信息。协作者将对象放置到不同的存储服务器以：一个连续的 key 域存储在一个表中。较小的表会存放在一个存储节点中，较大的表会被分割并且存储与多个节点。客户端程序不会控制表的配置，而写作者会存放表和存储服务器之间的映射信息。RAMCloud 客户端库将会保存一些缓存信息以防多次访问协作者。RamCloud 通过写备份的方式来保证数据的持久性 [15]，如图3.8所示，当一个存储节点中的主服务收到一个写请求的时候，它将首先更新自己内存中的 Hash 表，并且将新的数据转发到多个备份节点中，它们将把这些新数据缓存在它们的内存中，而最终被写到本地磁盘中。备份节点必须使用备份电源来确保即便在断点时，缓存中的数据也能被写入到本地磁盘中。当需要恢复数据的时

候则通过多个节点并发恢复的策略来减少恢复时间。

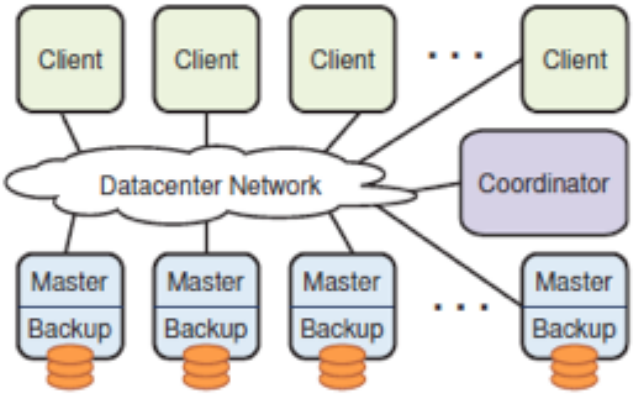

图 3.7 RamCloud 架构图

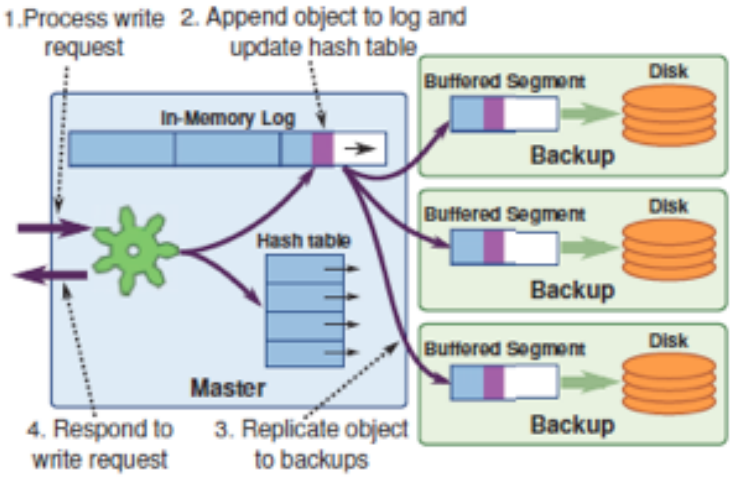

图 3.8 RamCloud Log 访问的结构图

RAMCloud 的设计思路和目标以及遇到的一些问题都非常具有启发性。由于内存在随机访问上的优势，能够极大的降低系统的访问延时，提高吞吐量，非常适应与移动云计算环境对存储系统的要求。因此未来将大量的活跃数据放置于内存之上是非常有必要的。虽然前面提到的 Cassandra 和 Bigtable 等系统都很大程度上利用了内存的高速特性，但都是作为缓存来用的，这与 RAMCloud 中所提出的将内存完整的视作存储介质不同。RAMCloud 不仅需要考虑读，更

要考虑写。比如在写效率同样高于磁盘百倍的前提下，完成数据的可用性和持久性的设计。这也是基于内存的存储系统的关键。

3.2.3　前人工作的主要不足

即便在半导体技术带来的成本降低、大量实时应用带来的需求推动之下，完全基于内存的存储系统如果要推广使用，依然面临着很多的问题。首当其冲的就是数据的安全性。由于磁盘是持久存储介质，即便服务器崩溃数据也不会丢失。而内存作为易失性介质，在掉电后所有的数据都会消失。因此基于内存的存储系统第一要解决的问题就是如何持久存储数据。这个问题上，RamCloud提出了一个非常有效的策略，它将内存中的数据备份到别的节点的内存和磁盘中，并且设计了一套快速恢复的策略，使得数据丢失后能够在几秒的时间内完全恢复。除了数据的持久性问题之外，我们认为还存在几个重要的基础性问题需要解决:

1. 扩放性问题。由于单机内存的存储容量远远小于磁盘，因此内存集群若要达到和磁盘集群同样的存储能力需要更多服务器。因此，分布式内存存储系统需要具有更好的扩放性;

2. 读写效率问题。之所以使用内存作为存储介质是希望获得更好的对数据的访问效率从而满足实时应用的需求。传统的数据访问接口对于实时数据改变不够敏感，因此我们需要一个对数据改变响应更快的数据访问能力。

3.3　基于内存的分布式键值存储系统

我们认为一个基于内存的分布式存储系统应当具备的特点:

- 需要能够存储海量的小数据。小的、零散的原始数据现在已经成为云计算下应用程序最主要的数据来源。比如 Twitter 这个流行的微型博客系统，主要存储的就是不超过 140 个字符的短消息，然而在 Twitter 当前规模下，每天都会有超过 4 亿用户从世界各地提交超过超过三千五百亿条短消息。因此一个实时存储系统应该有能力存储下如此多的小数据。

- 需要保持高的写入效率。依然以 Twitter 来举例，考虑到其每天数以千亿计的消息以及更高量级的交互信息，要保存并且处理这些不断产生的数据需要非常高的写入效率。

- 需要高随机读速率。任何时候新的数据写入到存储系统的时候，我们需要马上对它们进行处理，这样才能满足实时应用的要求。

- 需要能够帮助开发人员实现实时应用。典型的实时应用需要对用户动作以在秒的级别产生相应，而这些计算一般包括了一系列复杂的分布式计算过程。因此我们希望存储系统能够为编写这样的应用提供更加简洁的接口。

为了应对上面所提出的对存储系统的要求，我们设计了实现 Sedna。相比较传统的基于磁盘的文件系统，Sedna 能够提供更快的数据存储和访问速率；而相比较现有的内存系统，Sedna 在保证数据的持久性的基础上，提供了更好的扩放性，并且通过引入触发器的机制为用户提供了更加简单的编写实时应用的数据访问接口。在具体介绍 Sedna 的设计和实现细节之前，我们首先列举 Sedna 的一些基本设计原则：

- Sedna 将所有的数据以键值对 (Key-Value) 的形式存储于服务器的内存中，并且扩展了键值对的键 (key) 来提供了层次化的数据空间。

- Sedna 使用了层次化的整体架构来避免单点故障以及单点性能平静。并且通过这种方式提供了更好的扩放性

- Sedna 提供了读写触发 API 能够帮助用户编写实时应用。通过向用户程序推送最近的数据改变并且触发用户的动作，Sedna 能够减少编写复杂的实时应用的难度。当然，这些 API 也特别适合用于基于 Sedna 的流处理应用。

3.3.1　Sedna 总体架构

一个生产环境下的分布式存储系统是比较复杂的，至少包括：数据持久模块，扩展性解决方案，数据持久性解决方案，稳定性解决方案，数据均衡，服务器管理，错误检测，错误恢复，多备份管理，状态控制，并发机制等等。在本节中，我们将集中在几个 Sedna 的核心概念和设计上，主要包括：数据分割，

表 3.1 Sedna 存储系统使用的主要技术列表

Problems	**Technique**	**Advantages**
Partitioning	Consistent Hashing	Incremental Scalability
Replication	Eventually Consistency	Higher R/W Speed Flexible policy
Node Management	ZooKeeper Sub cluster	Avoid the single node failure
Read&Write	Read Trigger and Lock-Free Writes	Speed up the Processing by data push and low latency
Failure Detection and Handling	Heart-beat protocol and Active detection	Reduce the failure detection time
Persistency Strategy	Periodically flush or write-ahead logs	Different speed and availability according users' needs

备份，数据读写策略，以及 ZooKeeper 的使用。表3.1展示了 Sedna 中使用的几项核心的技术及其优点，我们会在后面逐一详述它们。

图3.9展示了 Sedna 存储系统的总体架构图，左边表示一个来自外部的请求以及来自内部运行着的实时应用对 Sedna 的请求；右侧则显示了 Sedna 中两类不同的节点所运行的组件的差别。Sedna 机群的所有服务器被分为两类：位于上层的子机群服务器，它们运行着子集群管理服务；以及位于下层的普通存储节点。数据中心中的所有服务器都运行着基本相同的组件，除了在子集群管理模块有少许的差别。

一台服务器上运行的一个完整的 Sedna 实例可以从逻辑上被分为本地部分和分布式部分。本地部分包括本内内存管理层以及持久化存储层，而分布式部分则包括了基于 ZooKeeper 实现的，帮助我们管理整个机群一致性信息的组件 (cluster status manager)。最上面的层次是不同的可插入的组件，用来实现 Sedna 的可定制的功能，比如接入不同的 replica 管理策略，或者接入不同的同步策略。

ZooKeeper 实现了 Zab 协议，能够为分布式应用提供式强一致协议的分布式服务，后面我们会更详细的介绍 ZooKeeper 以及其在 Sedna 中实现的优化。总

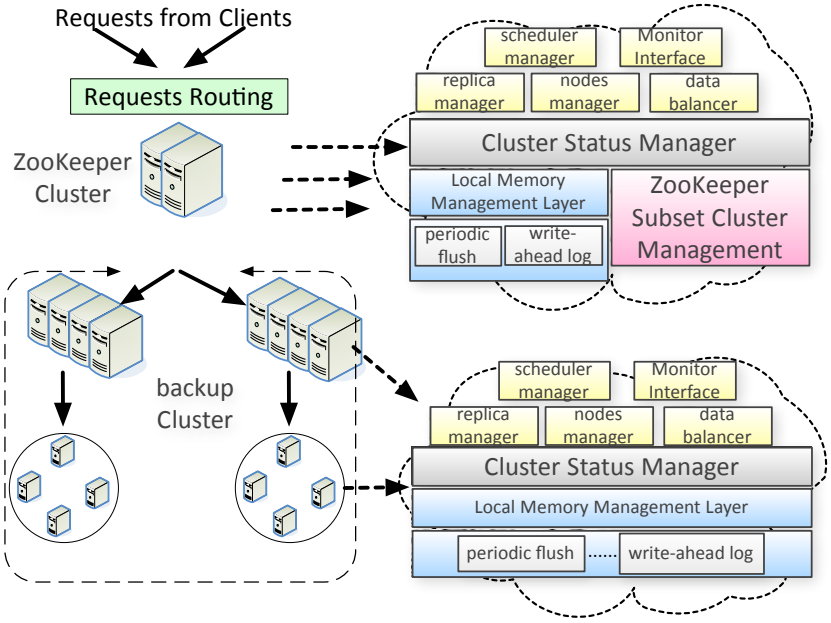

图 3.9 Sedna 机群的总体架构

体来说，Sedna 在下面功能处使用 ZooKeeper 提供的服务：

- Sedna 集群在首次冷启动的时候，使用 ZooKeeper 的锁服务来选举一个节点进行所有的初始化工作。

- Sedna 中所有虚节点和物理节点之间的对应关系通过 ZooKeeper 来保存。对每一个物理节点都会在 ZooKeeper 中新建一个节点，并且将其所存储的虚拟节点信息写入该节点。

- Sedna 中所有物理节点的心跳通过 ZooKeeper 来维护。所有节点上的 Sedna 实例在初始化时都会向 ZooKeeper 注册一个临时 *znode* 节点，并且向 ZooKeeper 维持心跳联系。当节点丢失后，ZooKeeper 能够最先反应并且提醒 Sedna 处理。

- 所有需要所有节点同步的状态都通过 ZooKeeper 维护。

3.3.2　数据分割

前面已经介绍，Sedna 是一个键值存储系统：所有的数据都以键值对的形式存储，通过对键的结构进行扩展，Sedna 支持一定程度的数据空间划分。不过对于任意一个键值存储的系统来说，一个非常根本的问题就是如何划分不同的键值对到不同的存储节点上去，Sedna 使用分布式哈希表的方式来进行数据划分，具体来说就是采用带虚节点的一致性哈希算法来进行数据划分。

如图3.10所示，Sedna 使用的一致性哈希算法首先将整个键哈希后的区间分为数以百万计的大小相等的区间。假设整个哈希后的区间为整数区间，那么分割后的每一个小区间都代表了一个连续的整数集。这个连续的区间成为虚拟节点，每一个虚拟节点都由一个序列号。当一个键值对到来的时候，它的键会首先被哈希为一个整数，之后会被指派给一个虚拟节点。每一个虚节点的数据都一定存储在一个服务器中 ($r1$)，并且被备份到另外两个物理节点中 (如图3.10所示的$r2$, $r3$ 中)。集群中所有的物理节点被称为实节点，以便和虚节点对应起来。

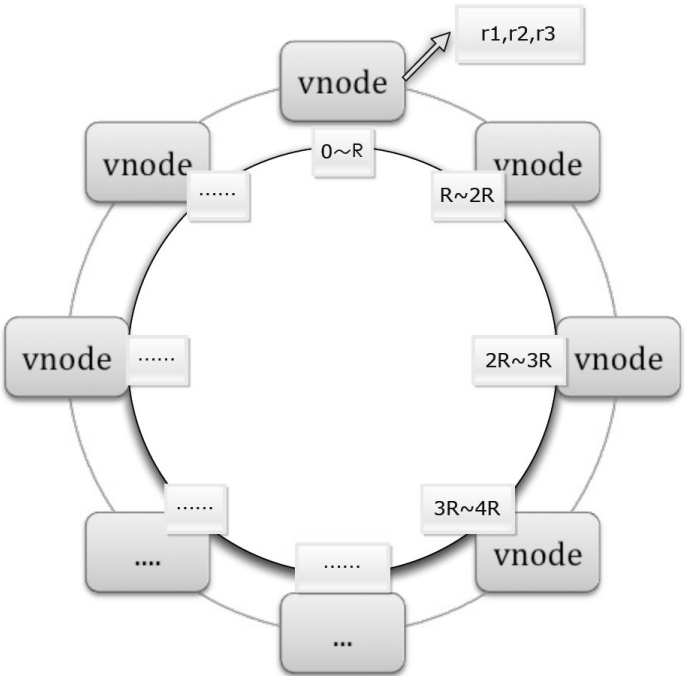

图 3.10 Sedna 虚节点环结构。其中存储着 0-R 的数据的徐节点被存储到实际物理节点 r1,r2 和 r3 上

我们之所以没有用最基本的一致性哈希算法主要是因为负载均衡的考虑。

当节点不断加入或者离开集群的时候，简单的一致性哈希会导致节点之间存储的数据范围有很大差别，从而使得集群性能不均衡，这对于大规模的内存存储集群来说问题更大。使用虚节点策略则能够有效减少 Sedna 集群中数据的不均衡性。在 Sedna 中，虚节点是数据存储的最小连续块。我们会记录所有的虚节点的状态包括它当前存储的实际数据的容量、读写的频率等，并且根据这些信息在 Sedna 中维护一个全局一致的均衡表，均衡表中回记录所有存储这些虚节点的实际物理节点的负载情况。虚节点是全局一致的，因此在 Sedna 中，它被存储在 ZooKeeper 子集群中。考虑到分布式读写的性能问题，均衡表不需要每次系统进行读写数据之后就更新，更新首先都是在每一个物理节点本地进行，所有的更新在本地经过计算和分析后，周期性的更新到均衡表中。

当系统发现某些物理节点的负载更高的时候，首先在系统中找到当前负载最低的物理节点，之后从负载较高的物理节点上找到和负载之差最为接近的访问频率所在的虚节点，并且将该虚节点迁移到负载较低的物理节点上。

由上面的负载均衡算法可知，Sedna 中虚节点和实节点的对应关系不是事先指定而是动态可变的，因此这也就需要我们有一个全局一致的存储来维护整个对应关系。在 Sedna 中，这些信息保存在图3.9所示的 ZooKeeper Cluster 中。这部分数据的存储和读取对系统性能影响很大，我们将在后面具体介绍它的设计和优化，并且通过实验证明我们所采用方案对系统性能的影响很小。

3.3.3 数据备份

Sedna 系统中每一个数据都至少包括两个备份以保证数据的可靠性。多个备份之间由于节点的崩溃，网络延迟等原因，存在不一致的可能，在 Sedna 中我们使用基于议会选举 (quorum) 的最终一致性来维护不同拷贝之间的一致性。

假设分布式系统中，每一个节点有 N 个备份，并且假设当我们从这个 N 个备份中读取到超过 R 个一致的数据的时候，我们才认为读成功，并且返回给用户；当从 N 个备份中写入成功 W 个，才认为写成功返回。那么当 N, R, W 满足下面公式 (3.1) 的时候，我们就可以保证每次成功读写都得到一致的结果。比如，如果每一个数据有三个备份 (这也是通常情况下数据中心中备份的情况)，设置 $R = 2$，$W = 2$，此时公式3.1得到了满足，那么在这种情况下，如果一次写成功，那么就意味着至少有两个节点上已经写入了正确的结果，如果此时发出读

请求，由于至少在两个节点上得到一致的结果才会返回，那么一旦返回的时候，我们可以确定返回的是刚刚写入的值。

$$\begin{aligned} R + W &> N \\ W &> \frac{N}{2} \end{aligned} \tag{3.1}$$

由于采用了 Quorum 算法，Sedna 中的读写都是通过一个协调节点实现的。来自客户端或者内部应用程序的请求通过路由器会被发送到 Sedna 机群中任意一台服务器，此时这台服务器就成为这次读写请求的协调节点。协调检点负责查询所需要的数据所在的物理节点所在的位置，并且发送请求，根据请求返回的数据的时间戳来判断是否有足够数目的一致结果。在得到足够数量的一致结果之后马上返回结果给调用者。使用协调节点能够有效的均衡客户端请求带来的压力，并且协调节点除了负责执行 Quorum 算法外，更重要的是还要负责起数据恢复和最终一致性的功能。

由于我们需要在 Sedna 中为每一个数据保持多个备份，同时对一个大规模的集群来说，服务器的离线崩溃是经常发生的事情，因此系统中会长时间存在着备份不足或者备份之间数据不一致的情况。Sedna 使用' 读恢复' 策略来修复这些问题，读恢复发生在协调节点上。当协调节点向多个备份发送读请求的时候，那些离线或者崩溃的节点会出现超时的情况，而对于那些由于网络丢包而导致数据不一致的节点来说，会返回给协调节点不一致的数据。协调节点在返回给客户端正确的请求数据之后会开始数据恢复工作。对于超时或者拒绝服务的返回值，协调节点会再次向 ZooKeeper 集群确认该节点是否真的离线或仅仅是协调者本身出了问题。如果确认对方节点确实已经离线，协调节点会异步的启动一个数据备份任务，将数据从已存的物理节点中拷贝到新制定的节点中去，并且在拷贝完成之后修改数据的映射信息。

3.3.4　节点管理

Sedna 对于节点加入和退出都是自动处理的。如果集群中新加服务器，只需要在其上启动 Sedna 实例即可。每一个刚启动的 Sedna 实例都首先启动本地的

内存管理模块，之后申请连接 ZooKeeper Cluster 来同步状体。如果 ZooKeeper 服务还没有初始化完成，这意味着整个 Sedna 集群还没有启动，该 Sedna 实例会申请一个全局锁，并初始化整个集群。对于 ZooKeeper 已经成功启动的情况，所有新加入的节点都会通过在 ZooKeeper 的 $real\_nodes$ 目录下创建一个临时 Znode 节点 ($ephemeral$) 来表明自己的存在；完成后，本地会启动一定数目的线程 (根据及其性能来配置) 来从当前负载较高的服务器上请求虚节点数据，并且存储到本地。当该动作完成后，它会改变集群中虚节点到物理节点的映射关系。因此它需要最后向 ZooKeeper 子集群申请修改这部分数据。

当一个节点意外的离线或者崩溃的时候，之前维持的节点和 ZooKeeper 集群之间的心跳信息丢失。这个动作会提醒 ZooKeeper 有节点离线了。此时 Sedna 集群并不会立即做出反应，节点丢失的恢复工作主要由读操作的时候指派的协调者来执行。

在 Sedna 中，虚节点的数目是在集群启动前写入到配置参数中的，一旦集群启动后，不能够更改。前面提到节点在加入系统之后会启动一定数目的线程来从别的物理节点拉取数据，这个线程数跟虚节点的数目有极大的关系。典型的，如果在一个实节点中最终将存储 100 个虚节点，也就是说集群公有 1,000 台服务器，而虚节点的数目是 100,000，此时数据拉取的线程数目最好设置为 10 个。

3.3.5　ZooKeeper 子机群

Sedna 使用 Zookeeper 子集群来实现分布式锁服务以及元数据管理的功能。Apache Zookeeper 是一个开源的用于开发大规模分布式程序中分布式协调的组件。它是一个中心化的服务，可以用来存放全局一致的配置信息、名字空间信息、提供分布式的同步或者组服务。ZooKeeper 基于 ZAB 协议实现，又称为 ZooKeeper Atomic Broadcaset 协议。ZAB 是一种非常类似 Paxos 的在不同的服务器之间保持一致性的协议，并且通过使用 TCP 并且减少提交时所需要的不同阶段数目，它的性能也好于传统的 Paxos 实现。

ZooKeeper 服务会在 Sedna 集群的一部分节点上运行，这部分节点就构成了图 3.9中的 ZooKeeper Cluster。该子集群中节点的数目是不断改变的，当 Sedna 集群规模较小的时候，ZooKeeper Cluster 规模也应该较小以保持较高的写能力；

然而随着 Sedna 集群的扩大，子集群的节点也可以不断扩大。ZooKeeper Cluster 在 Sedna 中的作用非常重要，本节中我们将分析 Sedna 中 ZooKeeper 的使用，并且针对性的设计优化的策略。

图3.11展示了 ZooKeeper 版本 3.2 运行在拥有双核 2Ghz 的 Xeon 以及双 SATA 磁盘上的性能图。读请求共 1K 个，写请求也为 1K，服务器 (Servers) 表明了 Zookeeper 所管理的服务器数目，大约超过 30 个额外的的服务器模拟客户端请求。试验中配置 ZooKeeper 首领不会接收来自客户端的请求。

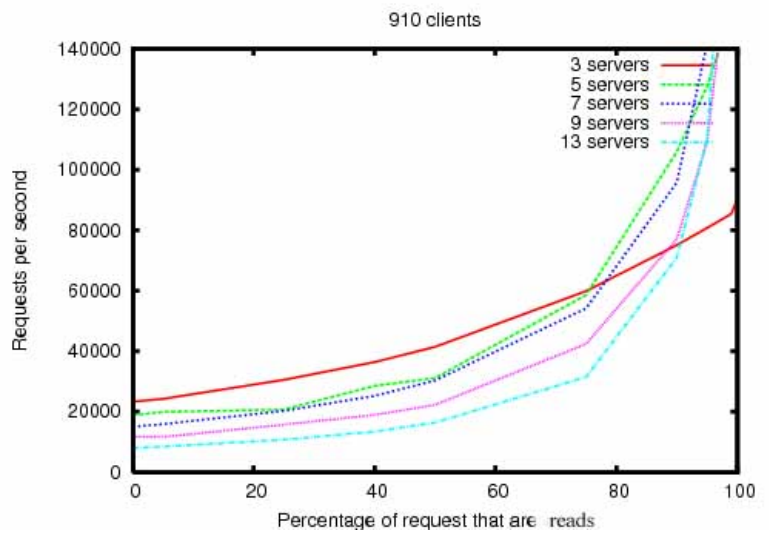

图 3.11 ZooKeeper 性能特征图

通过上述试验以及很多公开的结果，我们可以给出 ZooKeeper 的优缺点：

1. 请求中读占的比重越大，ZooKeeper 集群能够响应的请求数目越多。这点很容易理解，因为客户端读数据的时候仅仅需要连接 ZooKeeper 集群中任意一台服务器即可。然而，写请求需要传播到 ZooKeeper 集群中的多台机器，从而是的效率较低。

2. 对于读请求，ZooKeeper 的性能随着服务器数目的增加而提升；对于写请求，性能反而会随着服务器数目的增加而降低。

根据上面优缺点可见 ZooKeeper 本身是一个写效率较低而读效率较高的系统，因此在 Sedna 中对 ZooKeeper 将尽量使用读而不是写，需要写的主要有两种情况：

- 当 Sedna 集群首次启动的时候，所有的物理节点都需要向 ZooKeeper 写入代表自己的 znode 信息。除此之外，负责初始化的节点首先还需要单独向 ZooKeeper 写入代表所有虚节点的 znode 信息。由于典型的 Sedna 配置中包括百万条虚节点，因此这些动作会产生大量的写操作。

- 当物理节点加入或者离开继去年的时候，Sedna 会自动更新系统中表示物理节点的 znode 的数据。当我们因为数据均衡或者数据一致性需要对数据进行迁移的时候，需要修改虚节点和物理节点之间的对应关系，这是通过写 ZooKeeper 中数据实现的。

前一种情况虽然非常耗时，但是由于其只会在集群启动的时候执行一次，因此我们不会将其作为性能瓶颈来考虑。后一种情况在系统运行时会不断发生，但是发生频率还是远远低于毫秒级，而 ZooKeeper 典型的写入时间始终保持在毫秒级，我们认为即便集群规模扩大的情况下，这些操作不会降低 Sedna 的性能，在实验中我们也验证了这一论断。

ZooKeeper 的读性能或许是 Sedna 的另一个性能瓶颈。为了避免这个瓶颈，我们使用了如下三个策略：1）大量使用本地缓存。Sedna 中协调者进行读写操作的时候优先从本地缓存中获取数据，当缓存数据被证明失效的时候才会从 ZooKeeper 中更新新的数据。2）为了减少由于缓存失效带来的长时间延迟，我们在每一个 Sedna 实例中还会启动一个周期线程负责从 ZooKeeper 中更新数据的映射信息。该线程执行的周期成为租约时间。租约时间是不断变化的：当最近对于 ZooKeeper 中的数据有写入，那么租约时间就会减半；如果在上一个租约时间内 ZooKeeper 中没有数据改变，那么下次租约时间就会延长一半。3）每次对 ZooKeeper 中数据进行更新的时候，系统会将其记录在单独的 znode 目录下，这样无论在写入和读出的时候都能够避免锁在 ZooKeeper 中我们关闭监控 (watch) 功能因为大量的节点都需要监控同一个 znode 节点，那么任何一个改变都会导致一次不可控的网络风暴。

3.3.6　基本数据访问 API

3.3.6.1　写操作

由于在 web2.0 应用中，随即写是一种非常普遍的操作，Sedna 存储系统允许来自不同客户端对相同的键进行无锁并发写操作。Sedna 中所有的键值对数据都有时间戳的保护，并且每一个键所对应的值都不是单一的一个值，而是一个列表，列表中保存了来自不同的协调者所写入的最新的值以及当时的时间戳，列表的第一个元素则维护着来自不同的协调者写入的所有数据中时间戳最新的那个值。Sedna 为写操作提供了两个不同的 API：**write_latest()** 和 **write_all()**，他们两个都是无锁写。这个无锁是通过多版本数据管理实现的。

用户调用 **write_lastest()** 函数时，Sedna 会将请求的时间戳和当前该数据最新的时间戳进行比较，对于试图以较旧的时间戳更新当前教新的数据的请求会返回一个 $outdated$，而那些具有更新时间戳的写请求将直接覆盖当前数据，并且返回给用户 ok。而如果用户调用 **write_all()**，那么 Sedna 只会将请求的时间戳与该数据上次来自相同协调者所写的版本号做比较。如果当前请求比较新的话则会更新那个来自，否则返回 $outdated$。

任何程序调用写 API 都会得到三个不同的返回值：ok 代表当前数据被成功写入并且覆盖原有数据；$outdated$ 表示当前数据比较老，未能覆盖掉原有数据；$failure$ 表示由于网络等原因写失败。

3.3.6.2　读操作

所有键值存储系统的读 API 都非常简单，给定一个键，返回当前键对应的数据。由于 Sedna 中所有的值都根据它们来源和时间戳进行管理，我们也对应提供了 **read_latest()** 和 **write_all()** 两个 API 给应用程序快速的获取值。如字面意思所示，**read_latest()** 会返回最新的数据，不管当前读操作所在的协调者是什么状态；而 **read_all()** 则会返回当前键对应的来自不同协调者写操作的所有数据，这些数据被维护成一个列表发送给客户端。如何处理这些数据的不一致将是客户端软件的工作。

3.3.7　持久化策略

Sedna 实现了两种不同的持久化策略：周期刷新和写前日志。写前日志不是默认的持久化方式，它只是对于特定的应用可以启动，用来应对非常严格的数据持久化要求，而周期刷新作为日常情况下数据的持久化策略，已经足够保证 Sedna 中数据的高可用性。

我们首先分析一下周期刷新策略的可用性。在一个有 n 个服务器的集群中，假设所有的节点都有相同的 MTBF[37] 值为 m，那么每一个服务器在时间间隔 t 内发生失败的概率为 $\frac{t}{m}$。如果当前 Sedna 集群中公有 v 个虚节点，那么每一个实节点应平均包括了 $\frac{n}{v}$ 个虚节点，因此从某一个虚节点中读取数据的概率为:$\frac{1}{v}$。假设周期刷新的周期为 T_f，以及整个 Sedna 集群的读写的带宽为 tp/s.

Senda 出现丢失数据的条件是比较苛刻的，首先一个数据的三个备份都在短时间内发生故障，其次这个短时间内没有发生过任何数据刷新到磁盘的动作，并且这个短时间内也没有任何来自客户端的对该失效节点上任何一个虚节点进行访问。那么发生这三个事件的概率是可以计算的:

$$P_f = (1-\frac{1}{v})^{tp*T_f}(\sum_{i=3}^{n}(\frac{T_f}{m})^i(1-\frac{T_f}{m})^{n-i}(1-\frac{C_v^{ik}C_{ik}^k\cdots C_{2k}^k}{(C_v^k)^i}))$$

根据上面的公式所示，P_f 是一个非常小的数。假设一台服务器的 $m = 50,000$ 小时，$n = 1,000$，并且 $T_f = 600$，那么 P_f 小于 $2e^{-20}$. 因此我们认为在 Sedna 中使用三个备份，并且周期性的将数据刷新到磁盘中是足够保持数据的高可用性和持久性的。

3.3.8　Sedna 实时数据访问接口

之前我们已经提到了 Sedna 是为实时应用设计的分布式存储系统，我们除了将数据完全放入内存中来提高系统的读写速度，并且设计了新型的基于层次的存储架构来提高系统的扩放性外，更重要的是我们考虑设计了实现了一套实时数据访问接口帮助用户编写复杂的实时应用。

为了支持用户多种多样的实时应用的需求，Sedna 增加了基于触发的 API 作为其基础功能。触发适合实时处理的关键在于，它对于数据的改变极为敏感。

每当数据发生变化的时候，触发快速执行需要的动作，能够很快的给出用户需要的结果。触发机制在数据库中作为辅助工具已经广泛使用多年，这些功能在数据库中往往是作为确保数据完整性、合法性检查而出现的，不过我们在 Sedna 中设计的触发 API 和数据库中的触发器还有所不同，主要差别体现在 Sedna 运行在大规模的分布式集群中，其面临着更加复杂的调度和管理任务：不断执行的多个触发之间如果有相关性的话如何在存储层进行控制，如何帮助上层应用进行负载均衡等。

一个基于 Sedna 的触发 API 实现的典型的实时应用程序通常包含了若干个对数据的触发访问来完成计算任务。如图3.12所示，我们用三个触发器 A、B、C 来构成一个计算任务。触发器 A 会监测初始数据集的并且产生中间结果写入 Sedna 系统中，然后这些中间结果的改变又会触发令恶意个触发器 C 执行，以此往复。A、B、C 共同构成了一个循环任务的三个子部分。

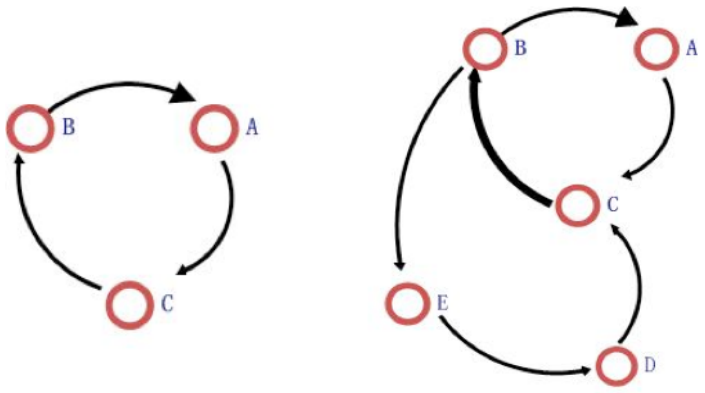

图 3.12 Domino 提供的基于触发的实时数据访问接口的典型应用实例

3.3.8.1　流控制

Sedna 中基于触发的 API 简单易懂，然而在一个分布式存储中广泛的提供这种 API 却面临着种种问题，其中最重要的挑战在于如何解决由多个触发 API 调用带来的饿可能的涟漪问题。我们依然使用图3.12来描述这个问题。图 3.12中右图展示了 Sedna 之上的应用调用了 A-E 共 5 个 API，我们称应用使用了 5 个触发器 A-E。执行过程中，A、D 两个触发后执行的结果一步激活了触发器 C，

而 C 的执行结果反过来又导致 A 和 D 的执行，由触发器相互激活形成的环可以用来支撑一些复杂的应用，然而在本例中，一旦循环开始，每一轮下来所有触发器被激活的概率都会提高一倍并且最终将整个集群的资源占完。并且由于 Sedna 是作为一个存储系统存在的，多个独立的用户会共同使用同一个集群，由于互相之间并不了解对方的应用，因此出现这种情况的概率就更高了。

我们通过在 Sedna 的触发 API 中引入 token 机制来平衡触发执行的速率。每一个触发器的执行都需要一定的周期来获得 token，对于那些在没有获得 token 的时候数据改变，API 将会自动忽略，并且不会返回数据给用户程序。

下面的代码展示了如何利用 Sedna 的实时 API 编写一个简单的应用 (基于 Java)。由于 Java 不是函数语言，所以开发者需要将执行的动作封装成一个简单的 Java 类，并且提交给 Sedna 实时 API。对于 Sedna 的实时 API 来说，每次调用都需要指定 API 所监控的键或者键的范围，数据发生改变时需要接管的类。每一个传给实时 API 的类都必须是 Action 类的子类，其中关键的 action 函数将获得新写入的数据的值，并且负责进行处理。

3.3.8.2　Sedna 实时 API 应用的示意

```
1
2  public class myAction extends Action<Key, Iterator<Value>, Result>{
3
4    protected void action(Key k,  Iterator <Value> vs, Result){
5      // user codes here
6    }
7
8    protected boolean  filter (){
9      // user codes here
10   }
11
12   public  static  void main(String[]  args){
13     SednaClient instance = new SednaClient();
14     instance.ReadTrigger(KeySet, myAction.class);
15   }
16 }
```

在上面例子中，首先我们实现了一个 Action 的子类 myAction 来负责执行用户逻辑，并且在主函数中调用 SednaClient 的实时触发 API($ReadTrigger$)，其中指定了需要处理的键的范围以及执行的动作类。在示例中还有一个 filter 函数，

该函数也是 Action 基类中的函数之一，用户应该继承它来根据自己的需求设置停止条件。Filter 机制基本包括两个方面：其一，filter 函数是基于每一个键值对来调用的，Sedna 运行时系统会根据 filter 函数是否返回 true 来决定是否执行用户动作；第二是 filter 函数使用的参数中包括了当前改变的数据以及改变前的数据，这样设计使得很多迭代收敛的任务能够更容易的基于 Sedna 实现。

3.4　实验和性能分析

本节中我们将对 Sedna 存储系统的性能进行实验分析。在 Sedna 中，我们使用了修改版本的 Memcached 作为本地内存存储，因此它的性能受到 Memcached 性能的限制。然而通过跟 Memcached 的性能比较我们可以看出，即便作为一个分布式的存储系统，Sedna 依然具有和 Memcached 可比的性能特征，并且能够提供 Memcached 这类缓存系统所不具备的持久性、扩放性等特征。在试验中，我们并没有将 Sedna 和别的基于内存的键值数据库来比较，比如 MemBase，主要原因是这些系统的并非基于 Sedna 所依赖的 Memcached 实现的，并且设计的主要目标也不同，使得这种比较不够公平。我们相信，通过和广泛应用与业界的 Memcached 性能进行比较，我们能为 Sedna 系统的性能设定一个标准的比较值。别的系统通过和 Memcached 性能进行比较就可以相比较出和 Sedna 的性能差别。

本节所有试验都基于一个有 9 台物理服务器的实验集群，急群中每一台服务器都使用 Xeon 双核 2.53Ghz 处理器以及 6GB 的内存。所有的物理机器都通过一个 GB 的以太网互联，节点之间没有划分机架，所有节点之间的消息传递的时间在毫秒以内。所有 Sedna 实例服务器分为两种，一种单纯运行数据存储功能，我们为每一台节点预先开辟 4GB 内存作为本地数据存储；另外一种节点构成了 ZooKeeper 子集群，因此其上还需要运行 Zookeeper 服务，我们为每一台节点开辟 3GB 内存存储本地数据。同时我们还使用 Memcached 在相同的实验集群上配置了相同的内存容量来进行比较

3.4.1 单客户端性能

着这个测试中，系统中只有一个客户端分别对 Sedna 集群和 Memcached 集群来进行读写请求。所有要存储的键值都非常小，因为数据的大小并不是影响 Sedna 系统性能的核心因素，数据大小仅仅会对网络带宽有影响。为了让所有的测试可比较，我们每次都随机产生一个 20 字节的键，比如 $test-000000000000000$，以及一个 20 字节大小的固定值写入 Sedna 中。

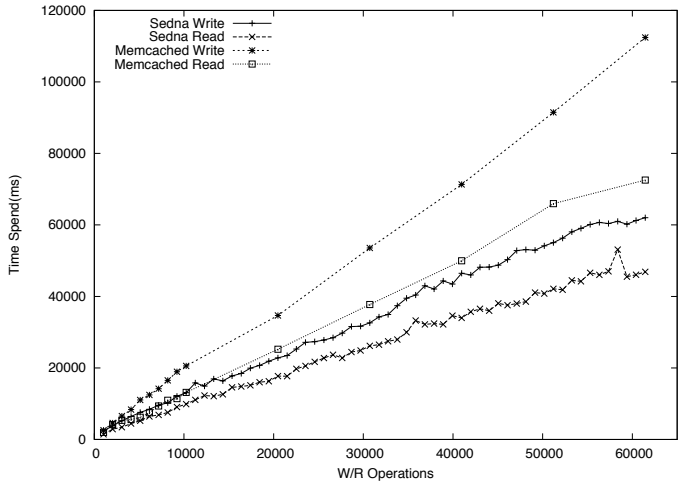

图 3.13 Memcached 客户端向不同的节点读写三次数据和 Sedna 正常读写的性能对比

前面我们也描述了，为了给出比较标准的对比性能，方便与其他存储系统做比较，所有的试验都以 Memcached 作为标杆。运行分布式的 Memcached 集群不需要在服务器端做任何的皮质，很多 Memcached 的客户端本身就支持以一致性哈希的方法将数据分割到不同的节点上去。当然了，Sedna 本身作为一个支持持久存储的内存存储系统，它和 Memcached 这样的缓冲系统还是有一个非常重要且会影响性能的区别的：Sedna 会将一个数据并发的向不同节点写入至少 3 处，并且在其中 2 处返回成功的时候返回。而 Memcached 只需要写入一份数据即可。为了使得这两个系统的比较更加清晰，我们先后强制要求 Memcached 的客户端读写三次数据和读写一次数据与 Sedna 读写请求做一次对比，最终的结果如图3.13和 3.14所示。

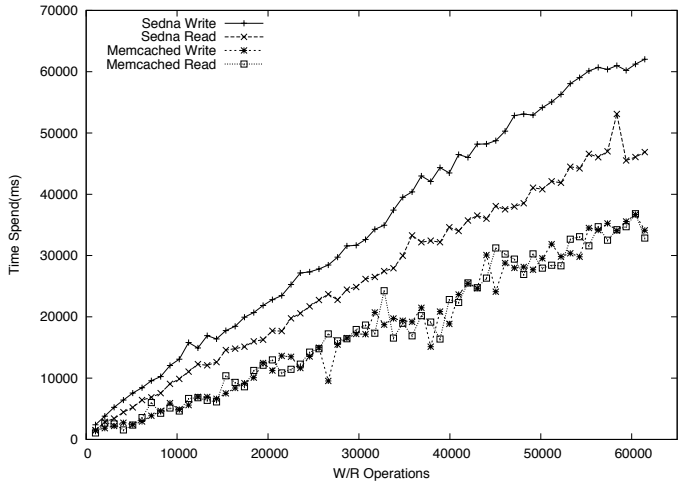

图 3.14 Memcached 客户端向不同的节点读写一次数据和 Sedna 正常读写的性能对比

尽管从图3.13中显示 Sedna 的读写性能都优于 Memcached，实际上这里面有一些不公平的地方：Memcached 的三次读写是顺序进行的，而在 Sedna 中，我们使用并发读写，并且使用了 Quorum 算法使得不需要等待最慢的读写完成，因此性能要优于 Memcached。之所以没有为 Memcached 设置并发读写，原因主要是如果用户需要使用多次写入不同节点来保证数据的安全性，在客户端往往是通过依次写成功来保证的。初次之外我们还在3.14中比较了 Memcached 只读写一次的性能和 Sedna 的对比，可以看出来 Sedna 的性能比较稳定，一直保持高速的读写能力，并且其速度比单纯 memcached 缓存系统稍慢，却明显快于在 memcached 中用 client 实现多备份的做法。

3.4.2　多客户端性能

作为一个云计算平台下的分布式存储系统，通常不只一个客户端在进行读写请求，系统中可能同时存在数以大量的应用从不同的节点请求读写。本节中，我们将在每一个服务器上启动一个客户端同时发起读写请求。在这种情况下，网络带宽有可能成为系统的性能平静，在我们的试验中，所有的服务器都通过 1GB 以太网互联。

图3.15展示了 9 个客户端同时请求时 Sedna 的性能，图中的读写次数是所有客户端读写速率的平均值。我们可以看出随着客户端数目的增加，每一个节点的 IO 性能都有所下降。这很容易理解，因此多客户端导致的服务器负载增加，网络带宽拥塞等。考虑同时有 9 个客户端在进行读写请求，Senda 系统整体的读写带宽是远远超过单客户端的。通过这个试验我们也可以看出，Sedna 在客户端数目增加的时候，由于引入了协调者的概念，从而具备了很好的扩放行。

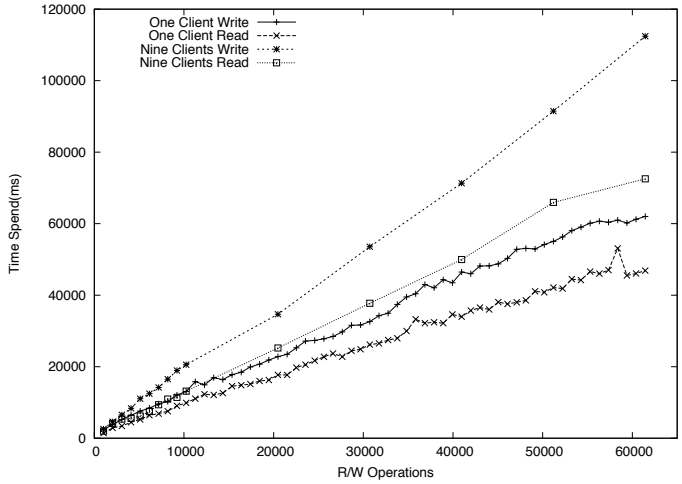

图 3.15 多客户端和单客户端的读写性能对比 (多客户端数据来自所有客户端的性能平均)

3.4.3　ZooKeeper 性能分析

首先我们对本试验集群中运行的 ZooKeeper 集群进行一系列性能测试。从图 3.16中可以看出一个同步的写操作在我们的集群中耗费了大约 10ms 的时间，并且这个时间随着 ZooKeeper 节点的增加有所增加但是并不明显。图3.17展示了当 ZooKeeper 集群中节点数目增加时读延迟的变化，我们可以看出当 ZooKeeper 读性能远远好于写性能。

为了证实 ZooKeeper 到底对 Sedna 系统性能的影响，我们在本实验中通过对启动阶段性能和正常阶段的性能来比较。当 Sedna 集群第一次启动的时候，我们称这个阶段为*启动模式*。这个模式下，大部分的虚节点还没有指派给任何

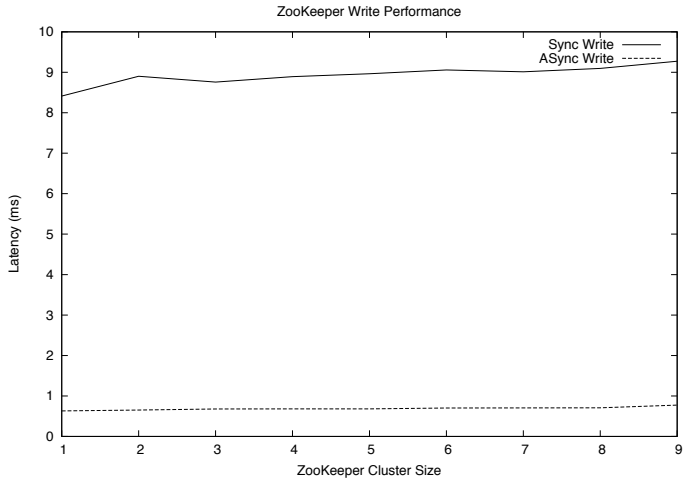

图 3.16 ZooKeeper 写延迟随着服务器数目的变化图

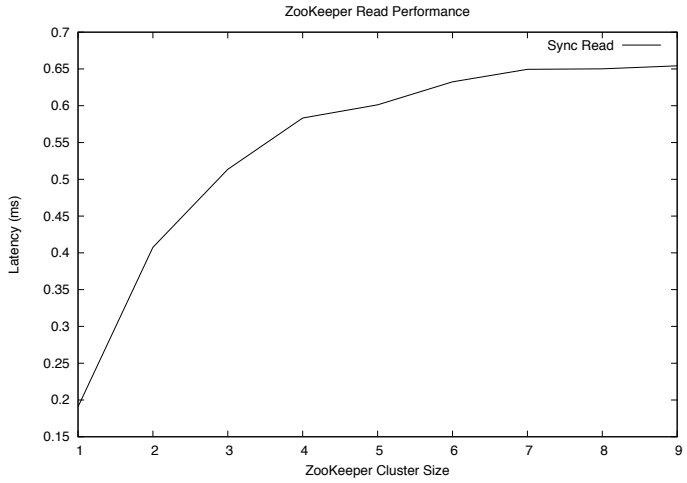

图 3.17 ZooKeeper 读延迟随着服务器数目的变化图

一个实节点，此阶段需要进行虚节点的指派，需要非常频繁的写 ZooKeeper。而这些写操作又必须是顺序的，并且使得所有节点的本地缓存无效。这些动作会极大的降低 Sedna 系统的性能。启动模式完成后，Sedna 进入安静模型，在这个模式下，大部分的虚节点都有了存储位置且相对稳定，且节点的缓存都是有效

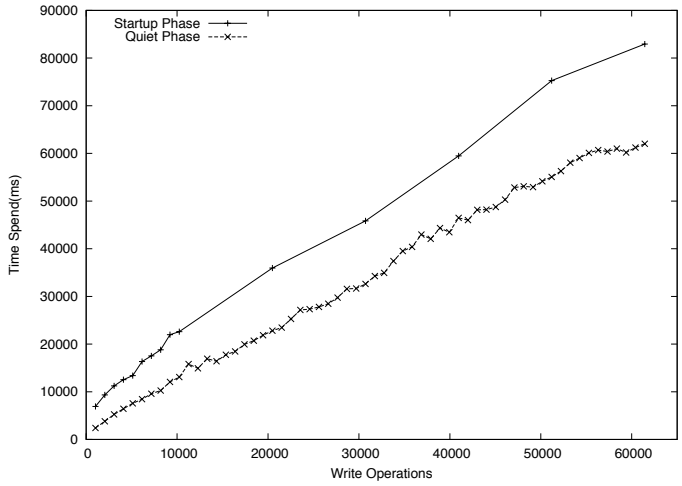

图 3.18 启动模式和安静模式下单客户端性能对比图

的，此时读写性能达到峰值。通过观察这两个阶段的性能差别，我们可以看出最坏情况下，ZooKeeper 是如何影响 Sedna 系统性能的。从图3.18中可以看出，这两个阶段的性能确实有一个比较明显的区别，不过这个性能差别是一个常数，并不会随着集群规模的扩大而增大。

3.5　小结

分布式存储系统作为云计算基础软件架构的核心对系统的性能起着至关重要的作用，为了支撑实时的云计算应用，我们设计并实现了一种完全基于内存的持久存储系统 Sedna。Sedna 能够在保证数据的持久性的前提下，为用户提供接近传统的内存缓存的性能。通过层次化的架构模型使得 Sedna 能够部署更多的服务器以提供与传统存储模型相同的存储容量。通过提供实时的 API，我们进一步丰富了 Sedna 对实时应用的支持。通过对实时应用下存储系统的研究，我们认为使用内存作为优化只是解决了问题的一部分 (提供了更快的数据的访问速度)，而实际上我们面对的实时应用需要对数据的改变进行快速的反应，能够处理复杂的应用类型，这就需要计算模型的配合。因此，我们通过设计和实现了一种基于触发器的通用编程模型来解决这个问题。

第 四 章　基于触发器的计算模型

4.1　引言

随着大数据时代的到来，人们每天能够搜集和存储的数据越来越多，并且这些数据中富含了各种有用的信息。比如很多电子商务网站会记录用户的浏览历史，甚至是用户在网站中浏览时鼠标停留的位置和时间。对每一个用户来说，这些记录的数据量并不大，但是对于通常能够拥有上亿用户的电子商务网站来说，记录这些数据就需要足够的资金来建立海量存储系统。然而公司愿意付出成本来记录这些信息，并不是单纯的为了应用' 海量' 存储系统，而是因为这些数据确实能为公司带来利润。事实上通过对用户行为的分析，公司能够非常准确的识别出每一个用户的特征、喜好、需求，并且针对性的给予推荐商品，从而极大的提高广告到实际购买的转化率。因此存储和处理这些数据也成为了此类公司的核心技术。除了存储处理这些本身就在赛博空间 (Cyber Space) 存在的数据之外，越来越多的物理世界数据开始被数字化并且搜集以加以利用。比如实时交通路况数据，街景信息，停车位信息，甚至是热门饭店的排位信息都在不断的产生，转化，存储。在这些数据被存储之后，更重要的就是如何处理以利用他们。

大量像 Amazon EC2 或者 Windows Azure 等 IaaS 基础设施服务的出现使得按需获取大规模计算资源成为可能，这也使得利用这些海量的数据成为可能。然而，正如过去几十年计算机的发展所揭示的那样，设计和实现不同种类的可扩放的应用程序依旧非常困难，特别是对于需要大规模计算的领域专家而言，他们除了需要考虑应用本身的分布式实现之外，还需要考虑竞争条件、死锁、分布式状态管理、同步等非常复杂的技术问题。为了把应用开发人员和这些繁琐复杂的分布式问题分隔开来，大量的分布式编程模型开始大量的出现。

在现有云计算平台的多种计算模型的基础上，本章将介绍一种基于触发器的新型云计算平台中的计算模型——Domino。该计算模型包括了编程模型、运

行时支持、以及容错、调度、检测等部件，并且通过与现有的云计算存储系统以及前章所讲 Sedna 存储系统的整合，为开发人员提供了一个简单、直观、高效的大规模应用开发环境。Domino 通过整合 HBase 存储系统来为应用开发人员提供了完整的存储和处理基础设施来更加容易的编写那些包含了大规模的迭代和递增处理的应用程序。我们基于 Domino 模型也实现了一系列的类似的应用，比如 PageRank 排序、一些比较典型的机器学习和数据挖掘算法 (K-means, 协同过滤等)、分布式的爬虫等。通过这些应用的编写和性能对比试验，我们认为 Domino 具备了比现有的云计算模型更加灵活的特点，并且相比较传统的 MapReduce 以及其扩展模型，更是具备了更好的效率和简洁的编程接口，更重要的是，Domino 还为我们的大规模分布式应用程序提供了实时的容错恢复能力，这一点是现有的模型很难做到的。

本章将首先介绍现有的编程模型以及它们的问题，之后介绍详细的介绍基于触发器的编程模型 Domino。在4.3中，我们将综合 PageRank 的例子来介绍什么叫做“基于触发器的”编程模型以及为什么使用触发器模型来进行编程；在4.3.3中，我们将着重介绍 Domino 中的同步模型；4.4具体的介绍 Domino 的实际设计与实现的细节；4.5将会就几个典型的应用介绍如何在 Domino 中实现不同种类的应用；最后在4.6中通过一系列的性能对比来验证 Domino 本身的性能以及其上运行的应用程序的性能。

4.2　相关工作介绍

4.2.1　MapReduce 模型及其问题

MapReduce 自从 2006 年由 Google 提出，并且之后由 Apache 基金会以 Hadoop 项目开源实现，当前已经成为云计算平台中的标准配置。全世界数以千计的数据中心时刻运行着百万个 MapReduce 任务。作为一个批处理的计算模型，MapReduce 非常适合于进行日志分析、离线推荐、或者单轮的 PageRank 算法。然而 MapReduce 依旧存在很多的问题:

- 不能很好的支持小的任务或者交互式任务。来自 Google 内部的经验，一个 MapReduce 任务平均需要 8 分钟的时间来执行，而一个 PageRank 算法重新运行一轮甚至需要几天的时间。这导致现有的 MapReduce 十分不适

合以下几种任务：实时搜索、在线推荐或者广告系统。可惜的是这些应用恰好是互联网收入的主要来源。再比如，在线推荐系统中，我们需要根据当前用户点击和浏览数据来计算新的推荐数据，这应该在几秒内实现。很明显的，现有的 MapReduce 是不符合的。

- 不适合迭代式的数据处理。许多数据分析技术都需要交互式计算，包括新型的 PageRank 算法，HITS（Hypertext-Induced Topic Search），递归关系查询，神经网络分析，社交网络分析等。这些技术都有一个共同的特点：数据都是被迭代处理，直到计算收敛或者到达停止条件。MapReduce 不能很好的支持这种迭代式的数据分析程序，开发人员只能通过编写驱动程序来手动的启动和管理多轮 MapReduce 工作。而这种开发方式显然不是一个好方法，1)，每一轮迭代中，数据必须不断的重新载入，重新处理，这将极大的浪费 I/O，网络带宽和 CPU 资源；2)，停止条件或者收敛判断往往还需要一个额外的 MapReduce 来进行判断，这将引入非常大的延迟。

- MapReduce 不适合进行数据源不断增加的分析任务。许多数据分析技术都需要对数据集进行递增计算，比如实时搜索引擎使用 PageRank 算法来对所有抓取到的页面排序，当爬虫爬取到部分新增网页，如果对所有页面一起运行完整的 PageRank 算法则极大的浪费了计算资源。可惜的是，这正是当前的实际情况。

- 使用 MapReduce 的编程框架，应用程序不能在线访问计算的任何中间状态，只能通过将不同的数据流进行聚合来模拟传统的共享内存结构。

为了解决这些问题，人们提出了许多改进和设计，我们将首先介绍其中的典型代表，并最终介绍我们提出的基于触发器的计算模型。

4.2.2　迭代处理模型

Haloop[38] 是一个基于 Hadoop MapReduce 模型针对迭代任务的的改进版本。Haloop 不仅仅通过扩展 MapReduce 模型使其支持迭代应用，并且提出了迭代感知的调度器以及引入了迭代任务之间的缓存，极大的提高了迭代应用的执行效率。

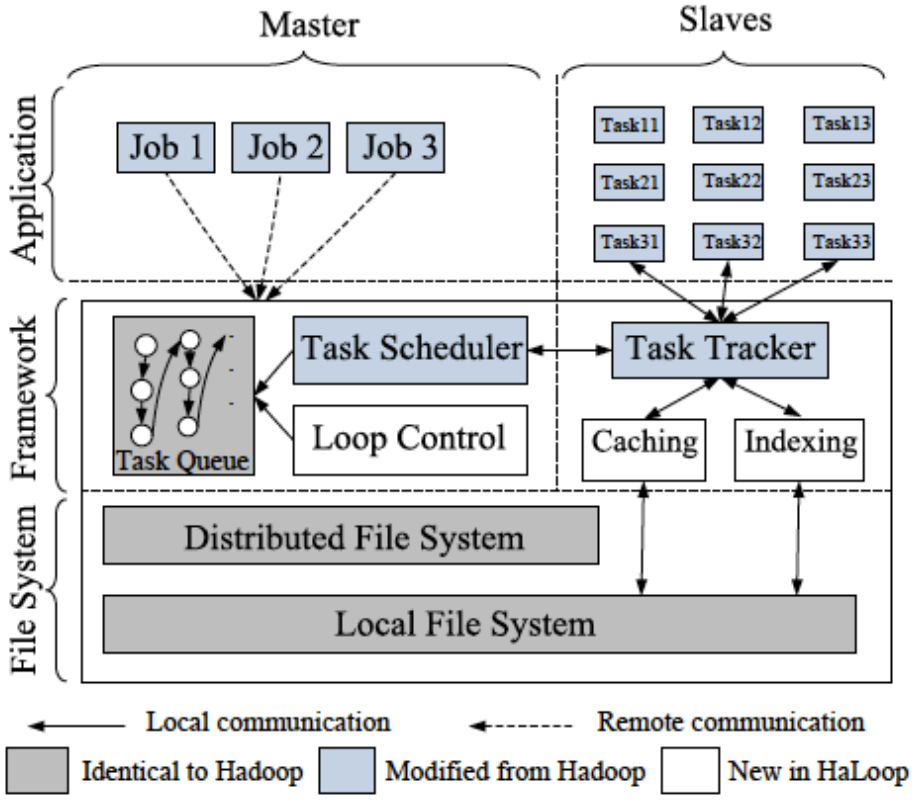

图 4.1 Haloop 架构图

图4.1展示了 Haloop 的整体架构。与 Hadoop 架构非常类似，Haloop 集群中包括了一个主节点和若干个从属及诶单。客户端向主节点提交任务，对每一个提交的任务，主节点会建立并且调度一系列子任务并发的在从属节点上执行。与 Hadoop 架构不同的地方包括如下几个地方：Haloop 的主节点包含了一个循环控制模块，能够不断的自动启动用户写入在循环体内的 map-reduce 任务直到用户指定的循环条件不再满足自动停止；其次，Haloop 使用了新的任务调度器，该调度器能够感知到迭代计算的任务，并且通过将数据优先调度到数据所在的节点来提高数据的局部性。最后，Haloop 能够主动的在从属节点中缓存迭代过程中产生的不变量，同时也会缓存 reduce 的输出以加速下一个迭代的执行速率。

Haloop 支持的迭代程序可以用下面的公式来描述：

$$R_{i+1} = R_0 \cup (R_i \bowtie L) \tag{4.1}$$

其中 R_0 代表初始的结果，L 代表计算过程中的不变量。

在 Haloop 中编写程序时候，程序员应当指定循环体、循环终止条件以及循环不变量。为了帮助编程人员实现终止条件，Haloop 提供了一些函数:*SetFixedPointThreashold, ResultDistance, SetMaxNumOfIterations...* 来提高编程效率。

Twister[39] 是另外一个支持迭代计算的 MapReduce 模型扩展。不同于 Haloop，它使用了一种基于 *publish/subscribe* 的消息传递的架构来进行数据传输和通讯以支持哪些长时间运行的 mapreduce 任务。

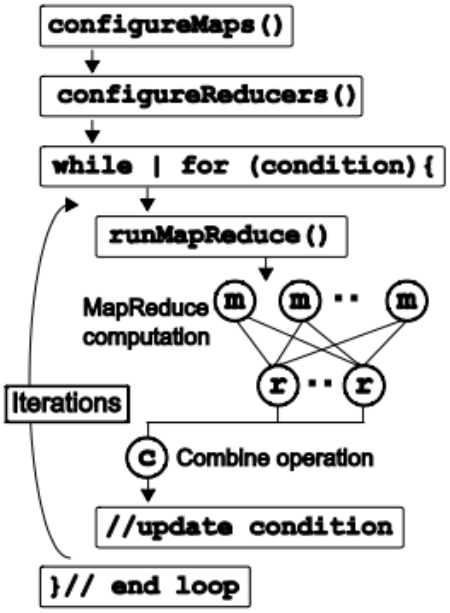

图 4.2 Twister 下迭代 MapReduce 编程模型示意图

图4.2描述了 Twister 的扩展计算模型。Twister 为 mapreduce 任务引入了配置阶段，在这个阶段里用户可以家在任何的静态数据以供后面的 map 和 reduce 阶段使用。那些长时间运行的 mapreduce 会将这些静态数据加载如内存以供迭代中不断使用。

图4.3进一步描述了 Twister 运行时的具体架构，其主要包括三个组件：1）客户端驱动程序，驱动了整个迭代程序的执行；2）所有的 worker 节点上运行的 Twister 守护进程；3）Broker 网络节点。所有的 worker 节点启动后 Twister 守护

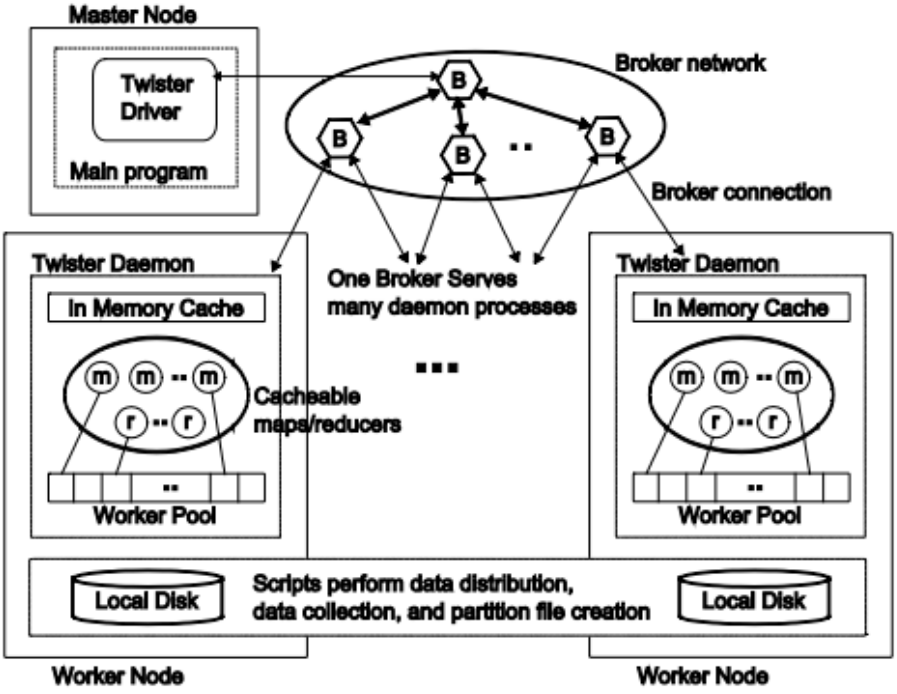

图 4.3 Twister 系统的总体架构图

进程会向 Broker 网络发起连接请求来接受命令和数据，该守护进程负责管理分配给自己的 map 和 reduce 任务，通知节点状态，并且最终需要对控制命令给予合适的相应。客户端驱动程序为用户提供了编程 API 并且将用户提交的 API 调用转变为控制命令和输入数据通过 Broker 网络发送给 worker 节点。

4.2.3　递增计算模型

Incoop[40] 是一个为了递增计算而修改的 MapReduce 模型，它不仅仅扩展了 MapReduce，还结合了底层分布式文件系统，使得 Incoop 本身能够检测到输入数据的改变，并且通过一些细粒度的数据重用策略尽量避免不需要的计算，从而提高递增计算的效率。为了达到这个目的，Incoop 对 Hadoop 系统做了几处修改：1）不再使用 HDFS 来存储输入数据和 MapReduce 任务的输出数据，Incoop 基于一个修改版的 Inc-HDFS 来存储数据，并且其能够很好的识别出多次循环执行中输入数据中的相似部分；2）为了控制任务的粒度，以尽可能的复用之前的计算结果，Incoop 引入了一个新的 Contraction 阶段；3）Incoop 提出了一

个基于亲和性的调度算法，以减少 task 在多次执行过程中数据迁移。

Inc-HDFS 改变了之前 HDFS 使用的基于固定大小进行分块的做法，转而使用基于内容的方式来进行分块。在 Inc-HDFS 中，引入了 maker 的概念，一个 maker 意味着一个特定的模式。当扫描输入数据进行分块的时候，如果当前所扫描的数据符合某一个 maker，那么就将块边界定在此处。当然为了避免过大或者过小的块，其也设置了一些大小限制。

Incoop 中的 MapReduce 任务执行至少有 3 个阶段：Map、Contraction、Reduce。所有的 map 都会将结果持久存储以来，并且把存放的位置信息和 map 任务的相关信息存放在单独的备忘服务器中，以供日后重复使用。为了减少 Reduce 任务的粒度，引入了 Contraction 阶段。Contraction 是基于原 MapReduce 中的 combiner 实现的。Contraction 阶段的结果可以被更好的重用。

Incoop 引入了基于备忘服务器的亲和性调度算法。它试图在尽量提高局部性和减少 straggler 之间取得平衡。该调度算法为每一个物理节点维护一个单独的任务队列，每一个任务队列都包含了根据备忘服务器的数据而最应该在本节点执行的任务集合。每次取任务都从本节点的任务队列中优先取出，直到为空，方从别的节点“偷取”任务执行。

Percolator 是由 Google 于 2010 年提出的一种事件驱动的处理模型，专用于递增计算。Percolator 在 Google 内部被广泛应用于实时搜索引擎业务。通过实际使用，Percolator 相比 MapReduce 方案可以提供相同的数据处理量，而将数据处理的更小效率提高了一倍。

Percolator 是专为搜索引擎而设计的。以 Google 为例，如果爬虫爬取了一部分新的互联网页面，如何处理并且建立索引呢？仅仅因为这一些新数据就重新运行一个 MapReduce 任务是不合适的，效率太低了。因此递增计算是一个较好的解决方案。Percolator 提供了对多达几个 PB 的数据集的随机访问能力，ACID 的事务机制。为了帮助程序员来追踪递增计算的状态，它还提供了观察者机制：一段用来观察指定的数据域改变从而触发执行的代码。一个典型的 Percolator 程序就是一系列观察者的序列，每一个观察者都可以通过修改数据从而触发下一个观察者。

Percolator 最主要的贡献在于它提供了两个非常重要的实现来支持递增计算：分布式事务以及通知机制。在一个可以随机存取的存储系统之上提供 ACID

事务是非常困难的，而且该事务是跨行的、跨表的。为了提高效率，Percolator中的事务是通过独立快照实现的，他不能提供串行读写保证，但是却更高效。Percolator 主要是通过 Bigtable 来实现包括独立快照、读写锁等机制的。锁信息存放在 Bigtable 的某一列中，为了处理由于节点失效可能导致的死锁，Percolator又引入了主锁的概念。通知机制也是 Percolator 的一个创新，其中的观察者作为Percolator 的基本单元以组成一个应用程序。每一个观察者都会完成一个任务并且通知下一个观察者来执行。该通知机制有点类似于数据库中的触发器，但职责不同。Percolator 的观察者实现具有几个优化的地方，首先保证多次触发同一个观察者不会导致任务重复执行；其次提供了一个非常有效的方法来寻找表中改变的项目。

4.2.4　实时处理模型

随着 Web 2.0 站点和移动互联网的兴起，用户开始尝试在互联网上共享更多的数据，而且这些数据的生命期也变得越来越短。对用户提交的新数据进行快速的分析需要能够进行实时计算的编程方法。当然，实时计算的范围相比迭代计算和递增计算可能更宽一点，它有可能包括这两种计算模式中的一部分，然而，实时计算最基本的特点是其对计算时间和计算延迟的关注。

论文 *Analytics for the Real Time Web* 中提到了一种扩展已有的 Key-Value 存储架构以提供实时计算模型的方法。该论文扩展了 Cassandra 存储系统，加以基于推送的处理协议，增加了对任务执行事务性的特点。总体上来说还是一种基于 MapReduce 风格的计算模型，不同的是其修改了 Reduce 函数，变为 Reduce* 函数。该函数能够递增的接受来数据输入，这也意味着用户可以不断的将数据推送给 Reduce* 函数，而须一定按照 MapReduce 框架所规定的那样，只能由 Map 或者 Combiner 函数来发送数据。唯一的局限在于并非所有的 Reduce 函数都可以转化为可以递增计算的 Reduce* 函数。

图4.4展示了一个如何利用本系统进行实时计算的流程。每当一个 key-value 对被插入到 Cassandra 系统中的时候，该节点就会执行一个分布式的事务：1）复制该 key-value 到多个节点；2）选择一个备份节点作为执行 map 程序的协调者；3）将 map 任务放入本地的任务队列中。而 map 任务的执行也是一个分布式的事务：1）将任务从队列中移除；2）将 map 输出写入到对应的 reduce* 节

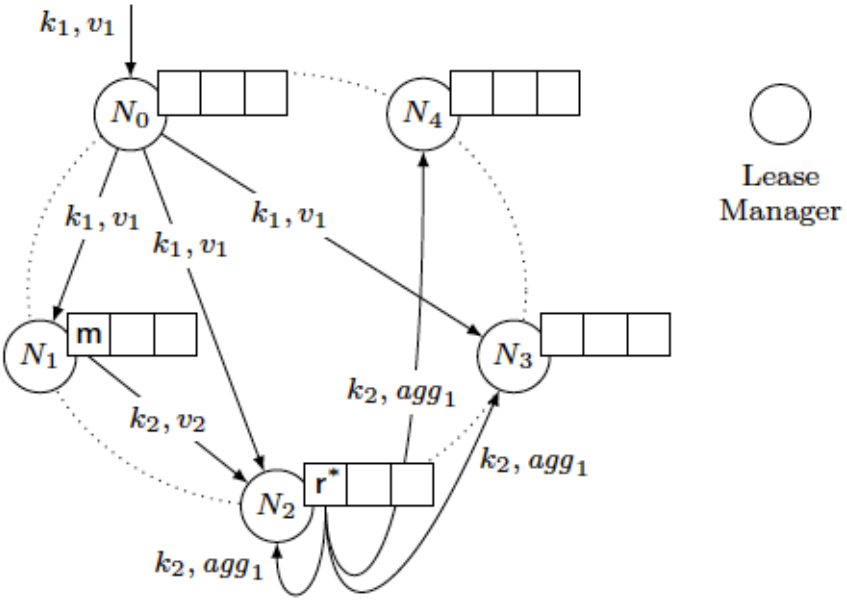

图 4.4 实时 MapReduce 模型下 map/reduce* 的执行流程

点；3）将 reduce* 任务放入到本地任务队列。而 map 输出将相同的 key 值路由到相同的节点中，reduce* 的输出会将结果保存到多个节点中。事实上 reduce* 的执行过程也被视作一个分布式的事务，包括将数据写入多个节点中，以及将 reduce* 任务正确的从任务队列中移除。

4.2.5　前人工作的不足

虽然上一节我们已经简单的介绍了很多分布式计算模型，然而实际上分布式计算模型的种类远远超过了本文的章节，我们在这里按照数据访问的模型对他们进行一个简单的分类并且分别讨论他们的缺点和不足。

上面我们所描述的编程模型中很大一部分是同步的数据流模型，这些模型针对的是那些面向数据流，并且对数据流进行同步的多轮处理的应用程序。比较典型的代表如 MapReduce[16] 以及 Dryad[20] 模型，以及一些基于它们的扩展模型 [38, 39, 41]。这些模型非常适合处理那些不需要访问全局共享状态的批处理应用，像 *wordcount* 或者 *sorting* 能够被很容易的在该模型下实现，这其中的一些扩展模型能够处理迭代计算，不过 MapReduce 模型本身的数据流模型使得

访问计算中间状态变得非常复杂，这个限制使得这类编程模型在复杂的迭代计算场景下还是显得无能为力。

一些编程模型提供了面向数据的模型，比如 Piccolo[21] 以及 Pregel[22]，这些模型提供面向对象的数据访问语义，允许用户直接访问任意的数据而不须将数据抽象为数据流来处理。由于没有数据流的概念，然而为了保证并发执行时数据的一致，该类模型需要引入全局屏障 (global barrier) 以同步程序的执行。同步的计算模型抽象虽然简单，但是却会极大的提高程序的执行时间，因为每一个阶段的执行时间都是由最慢的执行所决定。

与同步面向数据的模型相对应的是提供了异步语义的面向数据的编程模型，比如 Oolong[42] 以及 GraphLab[43] 等，其中 GraphLab 是一个针对图应用设计的编程模型，用户需要将其需要解决的问题抽象成为图算法来解决，这在某种程度上也限制了其应用范围。

Oolong 和 Percolator 同属于递增编程模型，它们针对递增处理应用而设计，从模型的角度来说很好的解决了递增计算和迭代计算对模型的需求，然而却在同步和错误处理上存在较大的问题。对于数据同步，Percolator 选择使用分布式事务来管理程序执行的状态，Oolong 则是通过用户提交的聚合函数作为全局屏障来同步程序的执行。Percolator 的问题在于大量的事务和锁非常容易使得开发人员写出效率低下的分布式代码；而 Oolong 使用的聚合函数过于简单，对于稍微复杂的同步需求就无能为力，这类面向递增处理应用而设计的编程模型太过简单，并且非常有针对性，无法作为一种通用的编程模型出现。

面对这些编程模型的问题，我们提出了一种基于触发器的通用编程模型，称为 Domino，它特别适用与高性能的迭代和递增程序中，并且作为一种通用模型可以用于类型广泛的各种应用中。Domino 的应用是通过组合一系列的触发器实现的，这些触发器是由用户编写的代码块组成，并且在特定的事件发生的时候被触发执行，同步或者异步程序都可以很容易的在 Domino 中加以实现。Domino 同时提供了一个面向数据的语义。我们认为 Domino 具备一下几点创新性:

- Domino 是一种通用的基于触发器的编程模型，它同时支持同步或者异步的应用程序。

- Domino 作为一种触发器的编程模型为用户提供了实时的容错恢复的能力，并且通过一系列的优化为高性能应用提供性能支持。

- Domino 和其他编程模型之间的合作能够进一步完善云计算基础软件架构。

4.3　基于触发器的编程模型

4.3.1　触发器模型

触发器的概念在计算机科学中出现已经超过 30 年了。很多传统系统中，组件需要对刚刚更新的对象进行识别并且针对性的执行一系列的操作。普遍认为这种方式能够提供更好的软件模块化能力，因为更新模块和处理该更新的模块可以独立起来。比如，一个雷达物体扫描程序会不断的更新敌军飞行器的位置，那么我们系统中应该同时有一个基于敌机出现或者位置改变发射火箭进行摧毁的模块来响应雷达扫描程序的输出。通常有两种做法来完成此类工作：一种是该模块不断的检查 (轮询) 感兴趣的对象，并且当发现对象发生改变时执行特定的动作。此方案最大的缺陷就是浪费资源，并且响应时间完全受限与轮询的时间间隔，而若设置较小的时间间隔，会浪费更多的资源。另外一种则是模块在某处等待直到感兴趣的对象发生改变才被激活执行特定的动作，我们也称之为触发模型或者在某些上下文语境下也成为异步模型。

从上个世纪 90 年代起，数据库领域就出现了一波研究主动数据库 (Active database[44–49]) 的热潮，核心概念就是提出一种面向数据库的主动执行的概念来简化编写数据库上应用。在主动数据库中，数据库操作或者外部的事件在满足一定条件的基础之下都能够触发特定动作执行。从概念上来说，主动数据库中的执行流程满足 ECA 规则，ECA 即 Event-Condition-Action。当一个事件发生的时候 (On Event)，并且特定的条件被满足了 (IF CONDITION)，然而指定的动作就将被执行 (THEN ACTION)。主动数据库后来被广泛实现于商用数据库系统中 (一般都是通过触发器或者触发器过程来实现)，比如 Postgres[50]，HiPAC[48]，Sybase[51]，VBase[52] 以及 OOPS[53] 等。然而直到今天，我们可以发现，虽然在理论上已经有很多关于 ECA 规则的研究，然而在主动数据库中，触发器依然是作为一个保持数据一致性、完整性或者安全检查的辅助工具存在

于数据库中，一直没有成为比较通用的编程模型，其中主要原因是其复杂度太高。虽然相比较传统的面向程序的编程模型，基于触发器的编程模型能够在触发器数目较小的情况下提供明显的响应速度提升，并且通过将较大的应用分解成为一个个小的触发器来模块化应用并加以管理，不过，由于触发器对数据库领域最为核心的 ACID 特性，特别是事务机制的挑战，使得其一直没有成长为一个通用的模型。

不过情况在分布式环境下开始发生变化。根据之前介绍的 CAP 所描述，在一个分布式的系统无法同时提供一致性、可用性以及分割容忍三个特性。而且现有的云环境中，我们更加重视可用性和分割容忍性，因此一致性往往成为我们牺牲的唯一选择。在新的 NoSQL 或者 NewSQL 的分布式存储系统中，我们基本上放弃了 ACID 的特性，转而尝试提供最终一致或者单节点内的原子性等。在这种情况下，触发器模型也逐渐成为一个好的选择。

4.3.2　Domino 的编程模型

Domino 严格遵守了触发器模型，它将一个用户程序抽象为：事件监控、条件判断、以及动作执行三个部分。为了更清楚的展示如何使用 Domino 模型来编写应用程序的，我们这里使用一个简单的分布式爬虫作为例子来介绍。

> **[分布式爬虫实例]** 最基本的分布式网页爬虫非常简单，它开始于几个简单的网页 URL 地址，爬虫会不断的将这些网页读取下来并且进行分析，通过分析网页的外链，爬虫将会获得更多的网页 URL 地址，周而复始直到爬取到某种限额或者整个网络。当然，一个可以工作的网络爬虫不止这么简单，它需要遵守 Robots 协议，需要考虑到网络带宽，需要考虑页面更新等等。而在我们的这个简单的例子中，我们并不会把重心放在这些功能上，而是更加关注网络爬虫的核心功能：爬取页面，分析页面。
>
> 我们的爬虫并不是爬取互联网上的网站数据，而是爬取一个新型的社交网络上的信息 (新浪微博)。微博是一种新型的社交媒体，所有的用户都可以发表公开的不超过 140 字的消息或者评论、转发他人的消息。由于其简单且交互性强而越来越活跃。我们不希望爬取

所有的微博上的信息，而是希望能够爬取最新的消息，并且希望我们的爬虫可以以更高的优先级爬取那些更重要的消息。这里的更重要使用了一个评论数加转发数的方式来定义，如果一个消息的评论数和转发数更多，那么它就更加重要。

基于 Domino 实现的网络爬虫程序总体结构如下：需要两个触发器的相互配合，触发器 *WBContentTrigger* 负责监控微博消息表，当其中出现新的微博的时候，需要该触发器获取该微博的所有评论或者转发的用户的信息，并且写入到微博用户信息表中；而触发器WBUserTrigger负责监控微博用户信息表，当该表中某一个用户状态发生改变，意味着最近该用户曾经评论或者转发过别的用户信心的时候，该触发器将负责爬取这个用户最近的微博消息，并且写入到微博消息表中。

4.3.2.1　事件 (Event)

那么触发器模型中，首要的元素就是 ECA 中的 Event 即事件。事件的种类非常多，按照之前在主动数据库中的分类，事件一般可以被划分为简单事件和复杂事件。其中简单事件主要包括了：时间事件 (包括绝对时间事件和相对时间事件)，方法事件 (比如某个方法被调用而产生的事件)，事务事件 (比如事务提交或者准备事件)，以及数据事件 (比如数据被读取或者写入产生的事件) 等。而复杂事件则是简单事件的组合或者是使用简单事件加上某些特定语义来组合，比如在某一个时间周期内未出现某个事件就被认为是一个复杂事件等等。然而在 Domino 中，我们主要关注的应用场景就是希望对存储在系统中的数据变化进行监控，因此我们主要提供了数据事件，特别是写数据事件的实现。当然时间事件也是很应用很广泛的事件源，不过由于分布式场景下时间的不一致导致我们无法给出一个用户一个统一的准确的时间线，而且正是由于时间的不稳定，使得有意义的时间事件一定是粗粒度的，比如每个一个小时、每隔一天。对于这种应用，我们可以很简单的将时间事件委派给周期任务去执行，而不是通过 Domino 来实现。因此，Domino 中主要关注了数据写事件。

众所周知，在 HBase[11] 中，数据是以表为单位存储的。表中的元素通过使用行关键字来区分，每一个行关键字都可以是任意一个字节串，而且 HBase 保

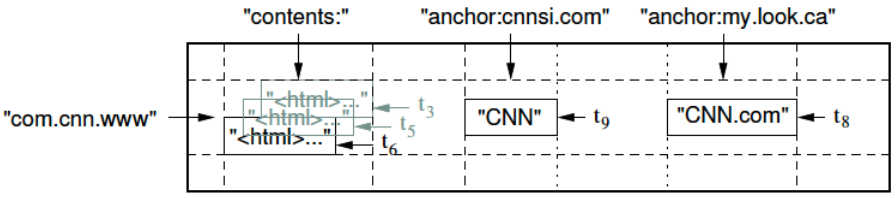

图 4.5 HBase 的基本数据单位：表的组成

证了所有针对字节串的读写都是原子的。如图4.5所示，HBase 中的每一个表通常包含了数以亿行的数据，每一个行最多可以存储上千个列族以及上百万个列。列是以列族的方式组织起来的，每一个列族中可能包含不定数目的列，这些列可以在程序运行的过程中随时增加，列族也是进行访问控制的最小单元。每一个行列定位一个 cell，其中存储着多个版本的数据。数据版本是一个 64bit 的整数。在 cell 中所有数据按照数据版本顺序以堆栈的方式堆叠起来。HBase 中，如果想要使用某一个列族，开发人员必须事先新建该列族，或者通过改变表的属性来增加列族，列族被创建之后列才能被使用，用户通过指定*列族: 列*来制定一个 cell。前面提到了 Domino 是通过修改 HBase 来实现的，在 Domino 中，我们要求用户提交触发器的时候指定所要监控的事件，默认情况下就是数据更新事件。这个事件中必须制定所监测的表、列族，列则是可选的，如果不指定列，那么所有该列族中的列上的更改都会产生数据更新事件。

> **[分布式爬虫实例]** 在 Domino 网络爬虫的实现中，我们需要监控两个数据写事件。一个来自由于爬取到新的微博数据而修改的 WBContent 表的列: *Content:zh*，另一个来自于由于猜测到某些用户可能最近更新了微博页面的 WBUser 表的列:*Activity:recently*. (触发器 WBContentTrigger 所监控的表如4.1所示，而触发器 WBUserTrigger 所监控的表如4.2所示)。

4.3.2.2　条件 (Condition)

条件是一组用户自定义的函数，该函数的传入参数是封装好的事件，返回值为 true 或 false 来表明当前用户制定的条件是否满足。在 Domino 中，条件

(condition) 包括了三种，分别负责三个关键的责任：首先，由于触发器一旦提交到系统中就会持久存在，除非用户显式的取消该触发器，然而在很多情况下，我们希望触发器能够在到达某些条件的时候自动停止，在这种情况下，用户就可以通过编写停止条件 (stop condition) 来实现判断。除此之外，由于 Domino 中的事件都是由数据写入引起的，如果短期内写入频率非常高，对它们进行快速处理就会极大的占据系统资源，在这种情况下，用户就可以通过编写间隔条件 (interval condition) 来为每一个触发器指定执行的周期。其实在 Domino 中，每一个活跃的触发器都是有一个最大执行频率的，用户可以通过编写条件函数覆盖该设置以加入自己的频率控制语句。最后，Domino 还引入了选择条件 (select condition) 来对事件进行选择，仅仅处理其中一部分事件，该条件对于很多高效应用非常重要。

停止条件 (stop condition) 中非常的应用在那些迭代收敛的应用中。比如 PageRank 或者很多机器学习算法，这些算法中，往往需要判断连续两次迭代中计算的值之差是否足够小。如果足够小，程序就可以停止运行。Domino 对这种应用做了特别的优化，每一个传入条件函数的封装好的事件都包括了更新前的值和更新后的值。

[分布式爬虫实例] 对于爬虫来说，我们不需要特别的停止语句，因此我们将默认使用 Domino 本身提供的条件函数 (如.1.1的 filter 函数所示)。

4.3.2.3　动作 (Action)

动作函数 (Action) 是由用户编写的一段代码块来执行所需要的代码逻辑。Domino 为用户的动作函数提供了完整的变量封装以提高用户代码执行的效率，如.1.1中的 HTriggerEvent 对象。在该对象中，Domino 为用户程序封装了多个相

表 4.1 HBase 中表 *WBContent* 的结构

MessageId	*Content:zh*	...,...
Message-1	$content_1$	...,...
Message-2	$content_2$	...,...
...	...	...

表 4.2 HBase 中表 *WBUser* 的结构

User Id	*Activity:recently*	...,...
User-1	$true$	...,...
User-2	$true$	...,...
...	...	...,...

关值供其操作，这些值包括：触发该事件的数据更新动作相关的行关键字的值；该触发器所监测的列族或者列中数据的新值以及更新操作之前的旧值；事件发生的时间和上次同样的事件发生的时间。除此之外，我们还封装了相关的环境变量。比如触发器执行时所在节点的信息以及某些用户提交触发器时显式设置的变量及其值。

Domino 为用户提供的是面向数据的编程模型，其允许用户在动作函数中任意访问需要的数据并且加以修改，并不需要像 MapReduce 或者 Storm 模型那样，必须按照流的概念来访问数据。这种方式简化了用户代码，不过也带来了问题。首当其冲的就是效率问题，允许用户在高频执行的动作函数对分布式存储系统 (HBase) 中的数据任意读写，特别是写，将给存储系统带来极大的压力，并且也会严重影响动作的执行速度。因此我们为这种情况专门提供了延迟写的异步语义，用户在动作函数中对分布式存储系统的写操作会被缓冲在本地的 WritePrepared 实例中，直到该动作函数退出前统一调用 flush 函数实施真正的写入操作。这一部分详细内容我们将在第**??**部分进一步详述。

> **[分布式爬虫实例]** 本应用中动作函数非常简单：对任意一条消息，找到所有曾经评论或者转发它的用户，并且将该用户的数据写入到 WBUser 表中。对每一个用户，爬取所有其最近发表的所有消息，并且存储到 WBContent 表中。具体的代码参见.1.1中两个触发器中的 action 函数。

Domino 模型在动作函数中一个关键的不同于现有的基于事件的分布式计算模型 (比如 Percolator 或者 Oolong) 的地方：如何实现聚合操作。分布式爬虫的实例中不存在聚合操作，为了说明聚合操作的作用和重要性，我们这里简单的扩展该爬虫：当从某消息中获得所有曾经评论或者转发它的用户后，且将这些用户数据写入到 WBUser 表之前，出于某种原因，我们需要判断该用户所评论或转发的字数之和是否超过一定阈值，如果超过才认为他是活跃的并且写入到 WBUser 表以进行爬取工作。实现该功能，就需要搜集到所有同关键字的数据，并且加以聚合，就需要使用到聚合操作。

在 Oolong 这样的事件处理系统中，用户可以使用一系列显式的，事先定义好的聚合函数，比如求和、最大 (小) 值等。不过这些函数都太过简单，限制了

实现很多复杂的逻辑的可能性。Google 的 Percolator 完全没有对聚合操作提供额外的支持，所有的聚合动作都由用户通过使用事务机制自己实现 (通常通过加分布式锁来实现)，这对于不熟悉分布式系统的开发人员来说并不是一个好的选择。不同于已有系统，Domino 设计了一个专用与聚合操作的设计模式：聚合模式。当用户需要聚合操作的时候，他可以完全按照聚合模式的指导来设计系统：首先，与原有触发器一起提交一个聚合触发器；其次，对所有需要聚合的触发器，修改其 WritePrepared 实例，引导结果写入到刚才提交的聚合触发器而不是 HBase 的表中；最后，通过修改聚合触发器中的动作函数来完成用户逻辑。通过这种方式，所有需要聚合的中间结果将被写入到一个隐式的表中 (t_{acc})，聚合触发器会自动检测该表上的变化，并且运行用户提交的动作函数。

引入聚合模式仅仅是解决了如何将来自不同的触发器动作产生的结果结合在一起的问题，紧接着的问题就是什么时候执行聚合模式中的动作函数：当来自不同服务器的触发器动作试图更新 (t_{acc}) 中的数据时，它们往往是不同步的。然而，编程框架无法判断是否应该在触发器动作处进行同步等待或者可以异步向下执行，这跟应用程序有很大的相关性，因此 Domino 为不同的应用类型提供了不同的同步模型。

4.3.3　同步模型

这里所说的同步和异步不同于 I/O 系统中的同步、异步模型。设想一下，在分布式程序中，如果当前计算需要分布运行的多个子任务产生的数据的时候，如何继续执行就是同步模型需要解决的问题。同步模型意味着计算的继续需要多有子任务的结果都已经产生；而异步模型意味着计算只需要发现有子任务产生了结果就可以继续运行。

4.3.3.1　严格同步

同步模型下运行的程序往往会带来很大的性能损失，这也很容易理解，因为程序的执行需要等待多个分布式子任务中最慢的那个完成才能继续，不过由于很多程序或者算法本身必须遵循该模型才能得到正确的结果，因此 Domino 中对该类应用提供了严格同步的原语以帮助实现这些算法。假设一个触发器在多

个服务器上独立运行，并且在某一个节点需要一个严格同步的场景，在 Domino 中可以简单的通过使用同步模式来实现：在节点处实现一个聚合触发器，该触发器所配备的表会存储所有需要同步的中间计算结果。此时问题的关键就在于聚合触发器上的用户动作什么时候开始执行，它是如何知道所有需要同步的任务都已经执行到同步点了呢?

在 Domino 实现中，我们提供了两个原语 ($register$ 和 $waitSync$) 来提供严格同步的功能。首先所有需要进行同步的触发器动作刚开始执行的时候都需要先调用 $register$ 来注册自己，每次注册成功后会在 ZooKeeper 中生成一个临时节点，节点的名字为 triggger 的 id 加当前执行序列 id，该节点中存储一个 count 值，每次注册的时候将 count 值加 1；当触发器动作执行完退出的时候会将 ZooKeeper 中对应的节点中的值减 1，如果减一后节点值为 0，那么就删除该节点。所有调用 $waitSync$ 的动作函数都根据当前执行序列 id，挂在 ZooKeeper 的对应节点上同步等待。当节点被删除的时候，$waitSync$ 返回，开始执行用户逻辑。此时可以保证所有需要同步的节点上的触发器动作都已经执行完毕了。

4.3.3.2　完全异步

与同步模型相比，异步模型是另外一个极端，它永远以分布式子任务中最快的那个为标准执行。相比之下明显具备更好的效率，并且很多算法都被证明异步结果和同步结果非常接近，并不会影响到结果的正确性。比如很多线性 [54] 的数据挖掘和机器学习算法 (belief propagation[55], expectation maximization[56], stochastic optimization[57, 58], 以及 PageRank 等)。Domino 本身的触发器语义天然的支持了这种全异步的模型：触发器被自动的指派到不同的节点上独立执行。

4.3.3.3　最终同步

为了平衡很多同步应用对性能的要求，Domino 除了提供同步异步模型外，还设计了一套最终同步的模型，专门用来提高那些需要较好的响应时间，且不能够简单改写为异步模型应用的性能。

最终同步模型中，程序的执行不需要等待所有的子任务都完成，程序可以提前向下执行，但是在足够的时间窗口后，最终还是会得到同步执行相同的结

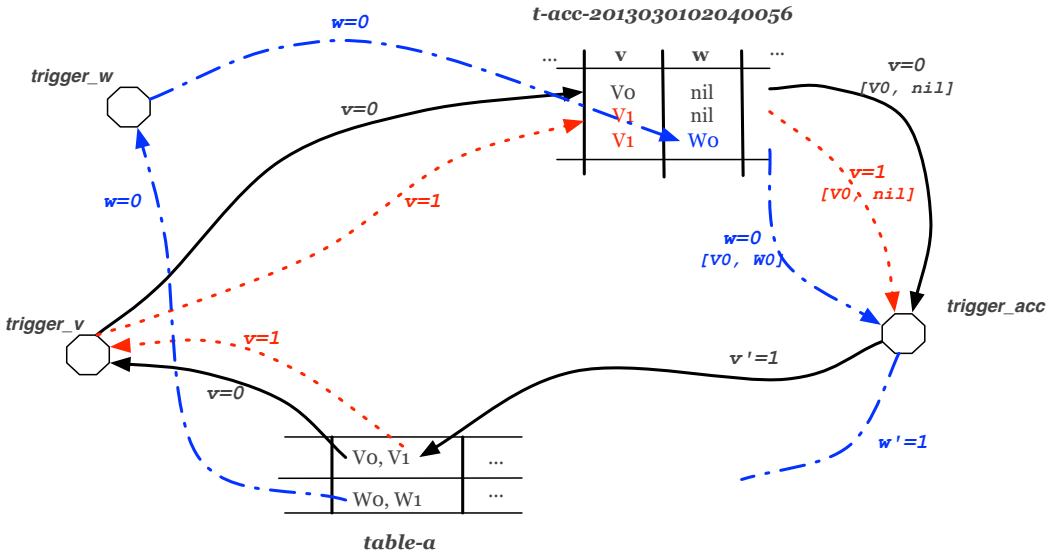

图 4.6 Domino 中最终同步模型下多版本执行的流程图

果。它不同于异步模型的地方在于，当前执行的程序的所有来自之前子任务的中间数据的版本号都必须满足一定的要求，以保证最终结果的正确性。因此，该模型下运行的应用能够具有较好的性能，并且有可能提前给出正确的结果。

Domino 中最终同步模型基于 HBase 的多版本表实现，它可以用于同步两个不同的触发器，也可以用于上一节我们描述的聚合模式。比如，当用户需要聚合操作，并且需要在聚合操作处进行同步的时候，只要伴随需要聚合的触发器再提交一个聚合触发器即可。系统会自动创建一个仅本应用可以访问的隐式表，该表存储的所有数据都是带有版本信息的。表中任一个 cell 上数据的版本信息等于将该数据写入表的触发器动作的执行序列 id，而触发器动作的执行序列 id 则是从 0 开始，每次执行自动加 1 的。聚合触发器的动作 (Action) 则根据表上的数据更新操作来执行：每次数据更新都会触发动作执行用户定义的聚合操作，那么这个动作的执行序列 id 就是更新后的数据的版本号。执行聚合操作需要读入表的数据，Domino 模型确保该函数只能读到小于或等于当前执行序列 id 的数据，最终产生的结果的版本 id 也是当前动作的执行序列 id 加 1。

图4.6展示了这样一个最终同步模型下多版本执行的流程。我们使用了两个触发器：$trigger_{[w|k]}$ 和 $trigger_{acc}$ 来做示例。其中 $triger_{[w|k]}$ 由表 $table_a$ 触发，而 $trigger_{acc}$ 用来帮助同步所有 $trigger_{[w|k]}$ 实例产生的中间结果。表名字为

t-acc-2013030102040056，由 Domino 系统自动创建，且仅在 $trigger_{acc}$ 中可以访问。由图中可以看出 $trigger_w$ 比 $trigger_v$ 慢一些，当 v 已经运行到执行序列为 1 的时候，w 刚刚开始它第 0 轮运行。那么在整个执行过程中，两个中间结果会被聚合触发器的动作函数根据输入 $[v_0, nil], [v_1, nil]$ 计算出来并且写入到 HBase 的表中。这些结果未必正确，但是它能够作为中间结果被访问和使用。如果我们的算法中这些部分结果是有意义的，那么这些部分结果就能够被最早的加以利用，而不是所有的节点都等待最慢节点的执行。当然，随着 w 的执行，其最终也会将结果写入到表中，聚合函数根据版本顺序，取到的输入正好是所有同版本的数据，从而得到正确的结果。

最终同步相比较严格同步来说，不会发生停止等待的情况，不需要分布式锁的参与。特别对于那些中间结果有意义的算法具有非常好的效果，且其本身也可以保证最终结果与同步模型结果完全相同。因此在实际应用中有很多优势。不过其问题也非常明显，其中最大的就是资源浪费问题，如果大量的中间结果是无意义的，那么浪费计算资源去计算它们就没有意义。因此对于这类算法，还是应当采用严格同步的模型。

由于在最终同步的实现中，所有的序列 id 都是本地维护的，这样就存在一种可能性：具有相同执行序列的动作函数都试图向同一个位置发起写操作。在这种情况下，我们需要对这些写操作指定一个顺序来保证结果的正确性。在 Domino 中，我们定义一个触发器的某个动作函数的写序向量如下：

$$V^T = (V_{tc_{fired}} : V_{tc_1} : V_{tc_2} : ...V_{tc_i}) \tag{4.2}$$

其中等式右边的 $V_{tc_{fired}}$ 代表触发该触发器动作执行的数据版本号，而 V_{tc_i} 则代表了触发器动作执行时访问的数据的版本号。当两个具有相同的触发器动作执行序列号的动作 (i 和 j) 试图向同一个位置写入数据的时候，系统将比较他们的写序向量：首先比较 $V^i_{tc_{fired}}$ 和 $V^j_{tc_{fired}}$，较大的动作写入；若相等，那么从前向后依次比较 V_{tc_i}，同样是较大的动作写入。

通过这种方式设计的写顺序，通过简单的分析就可以知道：任何时候，一个全部到达同步点的写入带来的触发器动作一定比所有未完全到达同步点的触发器动作的写序向量大，这样就保证了最终同步的结果不会被覆盖。

4.4　设计和实现

本节中我们将介绍 Domino 运行时系统的实现以及其主要挑战的解决方法。我们将 Domino 系统设计成为使用 Java 实现的基于 HBase 的计算模型，并且和 HBase 以一种插件的方式一起运行，然而事实上 Domino 的实现并不限制与 HBase，它可以与任何的分布式存储系统一起工作，只要存储系统提供持久性支持并且允许存储多版本的数据。我们现在正在将 Domino 移植到 Sedna 系统中。

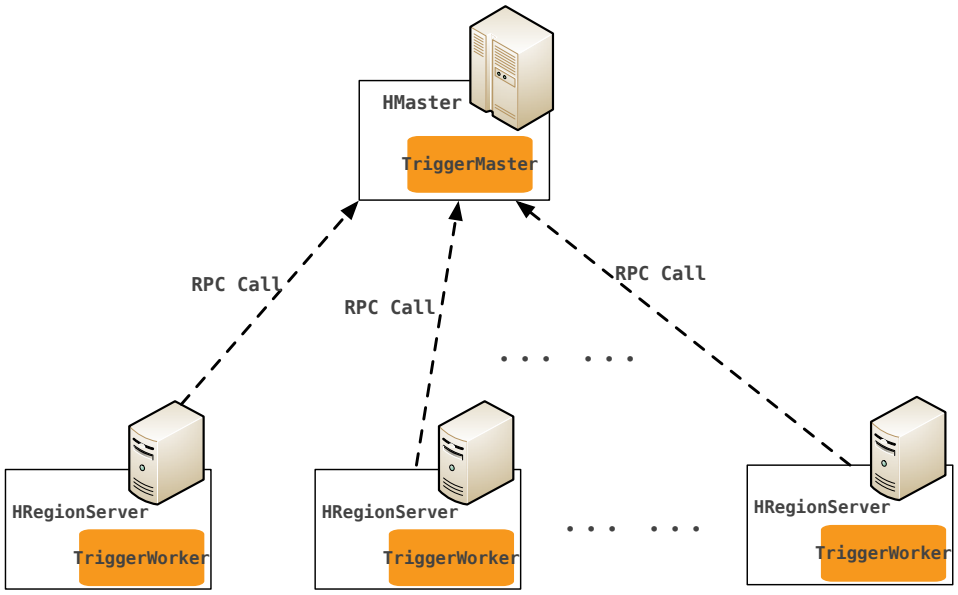

图 4.7 运行在 HBase 集群上的 Domino 集群的架构图

图4.7和4.8分别展示了 Domino 的系统架构以及不同逻辑模块之间的逻辑架构。Domino 本身是基于 HBase 实现的，因此在实际运行中其节点之间的架构和 HBase 一致，都是由 Master 节点和 Slave 节点组成。HBase 中的 Master 节点称为 $HMaster$，slave 节点也就是实际的 Region 存储的节点被称为 $HRegionServer$；于此对应的是 Domino 的节点，Domino 的 Master 节点 $TriggerMaster$ 与 HBase 的 $HMaster$ 运行在同一个物理节点上，而任务执行节点 $TriggerWorker$ 运行在 $HRegionServer$ 的节点上。所有的节点都运行着图4.8中的各个组件，唯一的区别是 $TriggerMaster$ 会运行其中的分布式管理 (distribted management) 部分，而 $TriggerWorker$ 则不会运行这部分。不同的 $TriggerWorker$ 之间通过同步的 RCP 远程调用来互相通信，而所有的 $TriggerWorker$ 也需要和 $TriggerMaster$

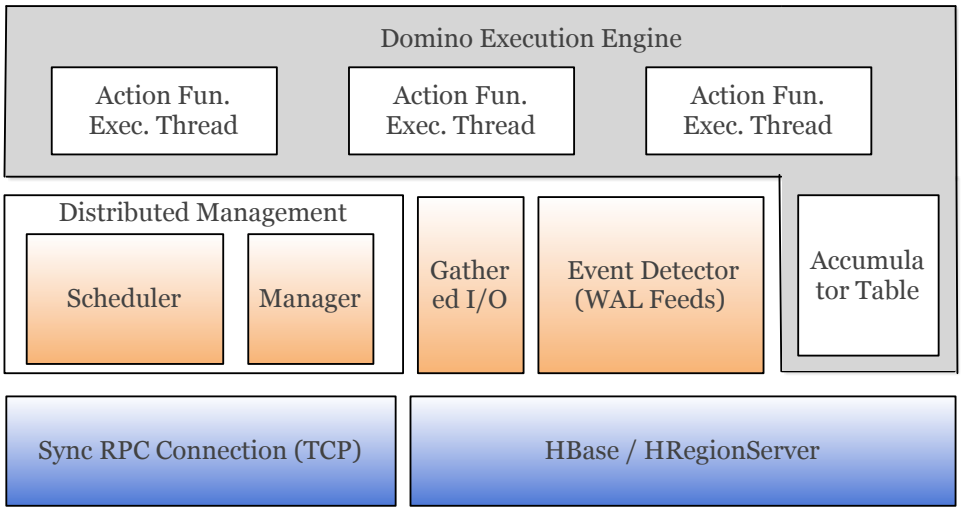

图 4.8 Domino 运行时系统的模块图，每一个模块都依赖于其下面模块提供的服务

保持一个心跳连接以汇报所有触发器的执行状态。需要注意的是，Domino 的 $HMaster$ 不是由用户手工指定的，而是在 Domino 启动的时候首次将自己注册成为 $TriggerMaster$ 的节点充当。因此在 Domino 中不存在单点故障或者单点性能瓶颈。

4.4.1　执行流程

用户想 Domino 系统提交触发器采用 MapReduce 任务提交类似的方式：首先将需要执行的触发器相关代码打包成 jar 包，然后通过 domino 命令行程序提交到系统中去。下面的代码块展示了如何将上文中实现的分布式触发器提交给 Domino 的命令。其中 WBCrawler 为一个包含了 main 入口的主类，其负责提交另外 WBContentTrigger 和 WBUserTrigger。

```
1  bin/domino trigger WBCrawler.jar wbcrawler.WBCrawler
```

用户向 Domino 提交触发器是通过新建并初始化一个 Trigger 对象，之后调用其 submit 方法实现的。下面的代码块展示了分布式爬虫实现中一个触发器 (WBUserTrigger) 的提交代码。用户需要新建一个 Trigger 对象，这个对象中必须设置触发器的名字，指定所监测的表、列族信息。如果这个触发器的检测对象具体到列族中的某个列，那么还需要调用配套的设置函数来设置，最后最重要

的是需要设置该触发器触发时执行的类。这个类中应当包括用户实现的条件函数和动作函数。

```
Trigger tg2 = new Trigger("WBUserTrigger", "WBUser", "Activity");
tg2.setTriggerOnColumn("recently");
tg2.setActionClassName("wbcrawler.WBUserTrigger");
tg2.submit();
```

调用 Trigger 对象的 submit 函数后，Domino 会首先从 $TriggerMaster$ 处获得一个全局唯一的触发器 id，之后将用户提交的 jar 包提交到 HDFS 存储系统由该触发器 id 组成的目录中。之后所有的 $TriggerWorker$ 都将 HDFS 的该位置下载并加载需要的类。Domino 会首先向 $TriggerMaster$ 提交触发器。提交成功之后，会询问 HBase 的.META. 表来获取本触发器所监测的表所在的 $RegionServer$ 的位置。之后会依次向这些 $RegionServer$ 提交触发器请求。只有所有的触发器请求都成功之后，提交才会返回。如果其中出错，Domino 会负责回滚之前的操作，并且返回用户提交失败的信息。

触发器提交成功之后就已经开始在 $RegionServer$ 处开始运行。此后，Domino 在每一台服务器上都运行着事件感知组件 (4.4.2) 负责监控数据更改，当遇到对象的数据修改的时候，用户提交的触发器代码就会被加载执行。触发器会因为停止条件 (stop condition) 而停止，也可以由用户提交命令显示的停止。在 Domino 中，用户程序可以使用 Trigger 对象的 stop() 方法来卸载一个触发器，也可以使用 shell 命令来完成：

```
bin/domino stop_trigger trigger-id
```

命令提交后，Domino 会首先根据用户提交的‘trigger-id’来查找对应的 trigger 实例。如果存在，就会先卸载运行在 $TriggerWorker$ 上的实例，所有实例卸载完成之后再卸载 $TriggerMaster$ 上的实例，完成后返回。

4.4.2　事件感知组件

Domino 的事件感知组件包括两部分。第一部分 (WAL Feeds) 作为主要的事件感知源用于在正常情况下对数据修改进行快速的感知；而另外一部分 (Sequential Scan) 则作为持久化事件的组件存在，当系统中出现错误的时候，该

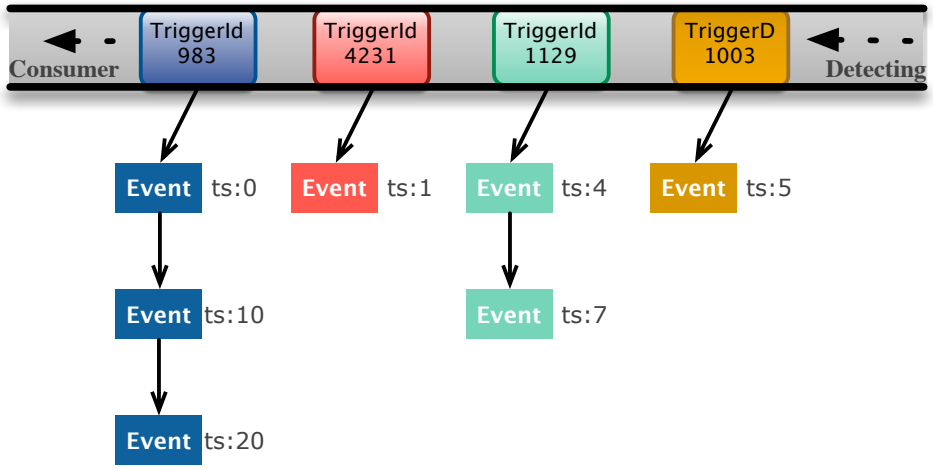

图 4.9 Domino 事件队列。如果两个事件属于同一个触发器，它们会整合被放入到队列中的同一个位置。队列中的顺序基于时间戳。消费者每次队列中取出属于同一个触发器的所有事件以减少频繁的进行线程切换带来的性能损失。

部分则保证所有未处理的事件都不会丢失。

4.4.2.1　WAL Feeds

HBase 是一个保证了数据持久存储的分布式存储系统，它不会因为少量节点的故障而丢失数据。这是因为其会将所有的数据都存储在磁盘中，而磁盘则利用 HDFS 提供的多备份来保证数据的安全性。然而由于磁盘的响应时间过慢，为了提高 IO 性能，HBase 会将所有的数据暂时存储在内存中，同时持久化在 HDFS 的 WAL(write-ahead-log) 中。为了保证数据不会因为在写操作的过程中节点突然崩溃而丢失，HBase 会先将数据写入到 WAL 中，并且在写入成功后才开始将数据写入到内存中。写入到 WAL 中的数据是按照其在内存中的结构组织来写，而是将每一条数据整理成包括：行关键字、列族、列、值、时间戳、类型的日志，以顺序的方式写入 WAL 文件中，以提高磁盘 I/O 性能。

Domino 的事件感知组件利用了 HBase 存储数据的特点，在每一个 $TriggerWorker$ 中，Domino 的事件感知组件都会将自己注册到 HBase 写入 WAL 文件的关键路径中，完全异步地对当前写入的日志数据进行判断，判断是否属于某个已注册的触发器所监测的范围。

一旦事件感知组件探测到了一个对 WAL 文件的修改，并且该修改正好属

于某一个已经注册的触发器的监测范围，它将封装一个包括了更新后的值以及更新前的值的事件 (HTriggerEvent) 并且将封装好的事件对象放入到事件队列中 (如图4.9) 所示。队列上有一个消费者不断的从队列中取出最新的事件来处理。最终，触发的事件会被发送到用户所写的条件函数和动作函数来处理。为了进一步提高效率，我们在每一个 $TriggerWorker$ 中预先开辟了一个线程池，且为每一个触发器启动了一个线程，并且空置等待着事件到来执行。在实际的实现中，我们扩展了 Java 的同步线程库，使得所有的线程都变得可管理。具体表现在所有的子线程都有自动的异常处理、执行状态记录、重启等功能。

4.4.2.2　Sequential Scan

WAL Feeds 组件对事件相应时间较低，适合作为主要的事件感知组件在 Domino 中运行。然而由于它的核心组件——事件队列是一直存在与单节点的内存中的，如果节点出现故障就无法恢复。这对于一个会长时间运行的分布式运行时系统来说是不可以接受的。为了在保证事件感知效率的前提下提高系统容错的能力，我们引入了 Sequential Scan 的策略来为 Domino 提供错误情况下的事件探测工作。

首先，由于 Domino 允许最小的监测单元是列，而 HBase 本身是将表按照列族分为独立的单元来管理的，因此同一行且同一个列族的数据一定在同一台服务器上存储着。我们修改 HBase 代码，为每一个列族默认的增加一列 (_events_)，每次 HBase 在向某一列族中的某一列写入数据的时候，系统会同时在 _events_ 列中存储此次写入的本地唯一 id，这个本地唯一 id 也会被存储到 $HTriggerEvent$ 对象中，传给动作函数。用这样的方式，在 WAL 文件中，就会生成连续的两条写入日志：数据列写入和 _events_ 列写入。另外一方面，当用户提交的触发器动作函数执行完毕的时候，Domino 会自动将 id 从 _events_ 列中删除。

如果 Domino 在运行过程中，某个节点崩溃，那么我们在 WAL Feeds 中所构造的事件队列全部丢失，此时触发器会在另外一个节点上重新启动，而所有的崩溃节点的数据也会在另外一个节点上由 WAL 文件进行恢复。在恢复的过程中，我们可以首先获得每一行数据，每一个列族的 _event_ 列最新数据。上面记录了在事件队列中丢失了的所有的写入时的 id。对每一个 id，从后向前，逆着

事件顺序搜索，当发现带有该 id 的 WAL 日志的时候，其前面那一条日志中存储的数据就是导致事件发生的，更新的数据的值。Domino 会强制重新产生这个事件，进而继续之前未能执行的触发器动作。

4.4.3 延迟写组件

Domino 为用户提供了面向数据的编程模型，它允许用户在自己编写的函数中自由的访问数据。但是这种模型也会带来很多问题。首当其冲的就是性能问题，因为访问非本地数据是非常耗时的，而用户写的函数中可能多次调用这样的写操作从而使得情况更加糟糕。另外一个问题就是数据的一致性问题，前面在介绍最终同步的时候，我们介绍了如何解决同一个触发器不同的动作之间写冲突时一致性的方案，而这里需要考虑的是不同触发器的写冲突。首先，对于单个写操作，在 Domino 中，我们始终保证后写成功。在这个前提下，我们面临一个更复杂的情况：由于我们允许用户自由的读写数据，那么有可能发生不同的触发器动作会进行交错的写操作。比如触发器 A 写入 loc_1，且成功，之后开始写 loc_2；而触发器 B 先写入 loc_2 成功后写入 loc_1，这样最后结果是 loc_1 被 A 写入，loc_2 被 B 写入，任何一个触发器内部都无法得到一个一致的状态。

为了解决这些问题，在 Domino 中，我们提供了一个延迟写组件 (WritePrepared) 帮助提高写效率，同时帮助在一个触发器内部维持一个一致的数据状态。WritePrepared 封装所有在触发器动作函数中发起的对 HBase 表格的写操作，并且通过在一次动作函数执行完毕的时候调用 flush 方法来一次性的将 WritePrepared 中缓存的数据真正写入到 HBase 的表格中。

WritePrepared 中的缓存数据写入 HBase 表时是有序的，顺序是由调用 flush 操作的时候从 ZooKeeper 获得的全局序列 id(G_{id}) 决定的。当两个具有不同全局 id 的写操作从 Domino 的触发器中发出的时候，如果两者并没有冲突 (即没有写入同一个位置)，那么 HBase 并不会感受到有什么不同；然而如果两者产生冲突，那么具有较大的全局序列 id 的写操作会被挂起直到前一个写操作完成。在挂起期间，所有之前部分写成功的数据都会被冲突的那个、具有较小的全局序列 id 的写操作覆盖。WritePrepared 的 flush 顺序并不意味着我们将所有的分布式写操作都强制排序了。强制对所有写操作排序会极大的降低写性能，我们只是通过

版本控制保证具有较低的全局 id 的 flush 操作开始后，它就不会被别的触发器抢占。

使用 WritePrepared 和 flush 操作，我们可以为触发器中的动作函数提供较高的 I/O 效率，并且保证所有的写操作不会互相交错。

4.4.4　容错和恢复组件

容错能力是云计算平台下的计算模型面临的的最大的挑战。很多流式计算模型，像 Storm 和 S4 这样的系统虽然具有较好的性能，但是其在容错和恢复上的短板同样极大的限制了其应用。然而，Domino 由于采用了基于触发器的模型，所有的中间数据都可以保存在持久存储的分布式存储环境中使得我们能够有机会提供一个让人印象深刻的容错能力以及一个近乎实时的错误恢复能力。

首先，Domino 的各个组件通过 HBase 提供的容错能力作为其静态容错的基础：所有提交的触发器信息都存储在 HBase 中，所有中间数据都被保存在 HBase 表中，触发器的执行状态也通过序列 id 区别保存，甚至聚合模式中自动创建的分布式表 (t_{acc}) 也持久存储在 HBase 中。这些信息都不会因为节点的意外崩溃而丢失。

除了静态容错外，由于触发器在 Domino 系统中时刻处于运行的状态，我们希望能够对执行中的状态实现容错和恢复。比如正在执行的触发器所在的节点崩溃，或者触发器本身线程崩溃，我们希望能够尽快的在备份节点上继续之前正在进行的计算。

Domino 节点中比较极端的情况是物理节点崩溃，这也意味着其上面运行的 HBase 服务也离线。那么节点崩溃之后的所有更新操作都会被重定向到自动选出的备份节点，并且在备份节点上会开始根据 WAL 中的数据重建内存数据结构。在备份节点恢复数据的同时，Domino 的触发器会随之重新在备份节点上初始化并且开始运行。这意味着，所有在节点崩溃之后新的数据写入操作都会如同没有发生错误一样在新的节点上被触发器监测并且处理。而对于那些崩溃发生时正在执行的动作函数来说，所有能影响到 HBase 的输出都被缓存在 WritePrepared 中。若崩溃时尚未执行到 flush，那么该动作函数的本次执行未完成，Domino 在备份节点上重新运行该动作函数即可 (只读入对应该执行序列 id 的输入数据)；若崩溃时已经开始执行 flush，那么这次动作函数的

执行其实已经完成，只剩下最后的 IO 操作。为了能够保证完成所有的缓存在 WritePrepared 中的 IO 操作，在 Domino 中，除了前面所讲每一个的 WritePrepared 都在 ZooKeeper 中申请全局唯一序列号 (G_{id}) 外，我们还为每一个 WritePrepared 实例都在 ZooKeeper 上映射了一个临时节点，其中存放序列化好的 HBase 写入操作序列以及当前执行完的操作 id，这样保证一旦 flush 开始就一定能够完整结束。

4.4.5　异常控制

在 Domino 中用户提交的触发器代码并不是像 MapReduce 中那样运行在独立的 JVM 中，而是运行在 $TriggerWorker$ 所在的 JVM 中。这样做主要是因为用户提交的触发器是常驻系统执行的，其执行时间长，粒度更低，对响应时间要求较高。如果每次提交触发器都要在相关的 $TriggerWorker$ 上生成新的 JVM 运行的话，那一台服务器所能处理的触发器数目就极为有限；并且每次在 Domino 中监测到的事件也需要通过进程间通讯的方式传输给在 JVM 中运行的触发器动作函数。

但是将用户提交的函数放在 Domino 所在的 JVM 执行需要面临不可控代码带来的错误和异常。在 Domino 中我们在每一个 $TriggerWorker$ 中实现了一个 $LocalThreadManager$ 类，它负责在初始化的时候预先开辟一个线程池供用户提交的触发器使用；除此之外，该类会记录所有的触发器和线程的对应关系；最重要的是其会尝试 catch 所有来自用户提交的函数产生的错误和异常 (通过捕获 Java 的 Throwable 对象实现)。这样当 $LocalThreadManager$ 发现某一个线程出现未处理的错误或者异常的时候，它会按照系统的要求进行处理使其不会影响到 $TriggerWorker$ 的正常执行。

4.4.6　优化

Domino 的设计和实现为面向需要快速给出答案的迭代和递增处理应用，尽管通过基于触发器的编程模型以及 HBase 的高随机读写的性能，我们已经提供了非常有竞争力的任务执行速度，但是我们依然希望能够进一步的提升系统执行速度，降低资源占有率。本章将主要介绍在 Domino 系统的两个优化。

4.4.6.1 内存加速

考虑 Domino 中的一个迭代应用，在触发器多次执行的过程中，我们需要将所有的中间结果写入到 HBase 的表中。这样做最主要的原因是我们需要这些针对 HBase 中表内容的修改才能触发下一轮的触发器执行，当然把这些中间结果写入到 HBase 的表中也能够提高系统日后容错恢复的能力。但实际情况是，在系统没有发生错误的情况下，我们对这些中间结果并不关心，我们只是希望尽快的得到最后的正确结果。因此，为了提升 Domino 中应用程序执行的速度，不将系统资源浪费在存储不关心的中间结果上，我们引入了内存加速的方案。

在 Domino 中，编程人员可以通过改变 WritePrepared 对象的参数来使得所有写的写操作不写入到 HBase 表中，而是写入到 Domino 自动创建的分布式内存表中。该分布式内存表本质上就是 HBase 的表，唯一的区别在于写操作不需要持久化到 WAL 中。因此其性能远优于 HBase，不过失去了持久性存储的能力。Domino 应用程序在迭代执行过程中产生的所有的中间结果都可以通过这种方式存储在 WritePrepared 所创建的分布式内存表中，而后续的触发器则可以通过监测这个分布式内存表来产生下一步的事件，并且加以处理。当触发器结束执行的时候，该触发器的 WritePrepared 对象所关联的分布式内存表将被转化成为真正的 HBase 表持久化储起来。

使用内存加速能够极大的提高 Domino 下应用程序的执行速度，它特别适合那些需要较快计算出结果的迭代例程。不过，我们在上面的描述中也同时指出了，内存中的数据都是不安全的，任何一个错误都会导致整个计算重新开始，因此对于那些需要长时间运行的应用程序，使用这种策略可能反而延长了计算时间。面对这种应用，我们可以将其分解成为若干步互相依赖的短的任务，并且对每一个短任务使用内存加速，这样就能够更快的计算出最终结果，并且对于计算中出现的错误也能够以较小的代价重新运行。

4.4.6.2 负载均衡

Domino 系统中运行的所有应用都严格遵守了数据局部性的原则：所有的计算 (触发器动作) 一定在引起该触发器运行的节点上运行 (尽管其执行过程中未必保证所有的读入数据都来自于本地节点)。这种特性对于提高计算性能是非常

有利的，不过却容易出现负载不均衡的情况：比如 HBase 的某表的特定行范围被更新的频率明显高于其它表或者表的其他范围，而这个范围恰好又落在某一台服务器上，那么这个服务器就会面临更多的触发器执行，从而导致相应速度变慢。这种情况下，我们称表的这个部分为热区 (hot spot)。在 Domino 中，我们考虑这种情况设计了简单的行负载均衡策略来对热区进行平均分配。

HBase 中表格存储是按照行进行分割的，最开始一个空表存储在某一台服务器上。随着不断增加新的数据，表的行数也在不断增加，当表的行数增加到一定程度的时候，HBase 会将这个表按照行进行分割，并且将两个子表存储在不同的服务器上。可以称这种做法为 HBase 的负载均衡。Domino 的热区均衡就是基于同样的流程：HBase 表的每一个区域 (Region) 的访问频度都被我们记录下来，当某一个表的大小超过的 HBase 允许的阈值，或者它的访问频度超过了 Domino 设置的访问频度上限，我们就将这个表进行分割，并且将产生的子表存储的不同的服务器上。表分割的同时，表上面设置的触发器也会跟着复制传输到另外一个节点。

4.5　应用实例

之前我们已经详细介绍了 Domino 的编程模型以及其实现的细节，按照我们在本章开始之处指出的，我们希望 Domino 是一个通用的编程模型而不是像数据库中的触发器那样仅仅作为一个辅助工具出现。为了表明，现在我们设计的 Domino 具有这个能力，本节我们将介绍如何在 Domino 中实现几个比较经典的分布式应用: 搜索引擎中常用的 PageRank 算法；推荐系统中常用的协同过滤算法 [59]；数据挖掘中常用的 K-means 算法。这三个算法各具特色，在某种程度上能够体现出 Domino 的通用性。需要注意的是，在实现这它们的过程中，我们都只关注了最核心的算法部分的实现且并未针对性的进行优化。

4.5.1　PageRank 算法

PageRank 算法是最早由 Google 的创始人提出的计算互联网中页面重要程度的算法。它的输入是一个稀疏图，图中的每一个节点代表一个页面，每一个有向边代表了页面之间通过超链接产生的连接关系。它的思想很简单：那些被

更多页面连接的页面更加重要。在实际计算中，我们将页面之间的连接关系抽象成一个网络图并且用一个矩阵 M 来表示，假设所有的页面的 PageRank 值最后组成一个向量 v，那么 PageRank 的计算公式如下所示：

$$v^{new} = \beta M v^{old} + (1-\beta)e/n \tag{4.3}$$

其中，β 是一个选定的常熟，通常在 0.8 到 0.9 之间，e 是一个全 1 向量，为公式合理性而加入的；n 代表了所有页面的个数。PageRank 算法会迭代的运行公式 4.3，直到 v_{new} 和 v_{old} 之间的差小于一个常数 ϵ。PageRank 算法的实现也非常直白：首先为所有的页面设置一个初始的 PageRank 值 pr_i，这样如果一个页面有 k_i 个向外的链接，那么每一个链接的权重就为 pr_i/n_i。其次对每一个页面求所有指向自己的链接的权重和，比如有 ins 个指向自己的链接，那么当前页面新的 pr_i 的值就为 $\sum_{k=0}^{ins}(pr_k/n_k)$。得到页面新的 PageRank 值之后，更新该页面所有向外链接的权重，继续执行下去。在得到页面新的 PageRank 值之后，是否马上更新所有链接权重，并且继续计所有别的页面的 PageRank 值体现了不同的实现策略。如果必须等待所有页面的新 PageRank 值计算完成之后才开始下一轮就成为同步实现，否则称为异步实现。实验和理论分析都证明，同步和异步的 PageRank 算法都能够得到正确的结果，并且异步算法速度优于同步算法，因此在本试验中，我们采用了异步实现。

在使用 Domino 模型编写 PageRank 之前，所有页面的信息首先应该存储在 HBase 中。表4.3展示了这样一个存储爬虫爬到的所有页面的信息表。表中每一行都是一个页面，每一个页面都有一个全局唯一的 id；第一列 pr 属于列族 prvalues，其中存储了当前页面的 pagerank 值；第二个列族存储了所有本页面中向外的链接指向的页面 id，其中列的数目不确定，其余与 PageRank 计算无关的没有在表中列出来。

表 4.3 HBase 中 *webpages* 表结构

WebPage	*prvalues:pr*	*outlinks:[* any linkout]*	...
p_1	0.5(default)	$p_{11}, p_{12}, p_{13}, \ldots$	...
p_2	0.5(default)	$p_{112}, p_{21}, p_{32}, \ldots$	...
...	...	... , ...	...

如表4.3所示，Domino 的 PageRank 实现只需要在 webpages 表的'prvalues:pr'列上设置一个触发器，每当一个页面的 PageRank 值发生改变的时候，我们就根据列族 outlinks 中存储的链接信息更新每一个链接的权重。考虑到停止条件(stop condition)，当任一个网页 wp_i 的新的 pagerank 值 $rank'_{wp_i}$ 和旧的 pagerank 值 $rank_{wp_i}$ 的差小于 ϵ，我们就可以停止继续运行。因此停止条件为:

$$Cond : true[r_{new} - r_{old} \leq \epsilon] \tag{4.4}$$

而根据当前页面的 pagerank 值更新所有的外链 (指向其他页面的链接) 权重的算法也非常简单，如算法4.1所示:

Require: Event object of current row (e)
  $n \leftarrow e.outedges$ //get out edges number
  $w \leftarrow e.rank/n$ //calculate out edge weight
  for all page ∈ e.outedge[] **do**
    lazy-write(pr-acc, page, curr-page, w) //write into accumulator
  end for
  Flush all written in lazy-write //flush to available to other actions

算法 4.1: PageRankDist

在获得每一个链接的权重之后，我们需要对每一个页面计算所有指向它的链接权重的和。这是一个典型的聚合操作，因此我们使用了 Domino 提供的聚合模式来实现该方法。算法4.1中使用 lazy-write 的时候，并不是将结果写入到 HBase 表中，而是写入到'pr-acc' 这个聚合触发器中。这个聚合触发器作用于表4.4中。它监测着表的列族 nodes，每当列族中数据发生改变的时候，聚合触发器就会通过求和得到当前页面的新的 pagerank 值，并且将新的 pagerank 值写入到表 webpages 的 prvalues:pr 中去。整个算法流程如4.2所示，具体的 Java 代码可见附录.1.2。

表 4.4 HBase 中表 *pr-acc* 的结构

WebPage	*nodes:[*any linkins]*	...
p_1	$p_{11}, p_{12}, p_{13}, \cdots$	...
p_2	$p_{112}, p_{21}, p_{32}, \cdots$	...
...	... , ...	...

Require: distributed table t_{acc}. page-id as the row-key.
Require: Each column represents the weights from one link-in edge.
  for all $weight_i \in t_{acc}.columns$ **do**
    pr += $weight_i$
  end for
  Flush pr back to *webpages* table

算法 4.2: accumulator actions of *pr-acc*

4.5.2 协同过滤算法

协同过滤算法是一种数据挖掘算法，主要用在推荐系统中。它能够根据已有的评分信息来预测尚未有评分信息的实体之间的关系。由于其有效性，大量应用在大型的互联网网站中。比如 Netflix 网上电影租赁系统会利用该算法根据用户已看过的电影以及评分信息来为用户推荐他们还未看过的电影。系统过滤算法也有很多种，本节我们将介绍一种称为 **alternationg least squares (ALS)** 的算法 [59] 在 Domino 上的实现，该算法曾应用在 Netflix 举办的第一届推荐系统大赛上，取得了非常好的效果。

ALS 算法通过将一个拥有百万行列的稀疏矩阵表示成为两个较低维度的矩阵的乘积来预测那些原矩阵中缺失的数据。ALS 的输入数据是稀疏矩阵 R，其中包括了所有已知的评分信息 (矩阵行代表了不同的用户、列代表不同的电影，每一个单元表示了某用户对某一部电影的评分信息)。ALS 算法会迭代的计算出一个低维的矩阵分解，如公式4.5所示。

$$R_{m,n} \approx U_{m,d} \times M_{d,n} \tag{4.5}$$

稀疏矩阵 R 是 $m \times n$ 维的，其被分解成为两个低维矩阵 U(m 行 d 列) 和 M(d 行 n 列)。具体的做法是先固定 M，然后求出最好的 U 矩阵；之后固定 U，求出最好的 M。不断迭代，直到对已知数据的误差平方差小于一个小常数 ϵ。这里面 d 是可变的，值越大计算复杂度越高，但是结果对于输入数据的拟合度越高。在实际的应用中，通常我们设置一个比较合理的值来控制，比如 20，来控制计算的时间。

不像 PageRank 算法，对于 ALS 算法本身的计算过程使用 Domino 的触发器

模型来考虑并不是非常直观。不过使用触发器模型来实现该问题确有非常大的优势。相比较现有的 ALS 算法，Domino 程序能够提供对不断增加、修改的用户评分信息提供实时的响应。在传统实现中，系统往往会选择周期性的重新执行 ALS 算法来处理过去一段时间增加的用户评价信息。这样，当输入数据集非常大，每次计算需要耗费大量资源的时候，就需要在推荐的实时性和资源使用成本之间做一个折中。使用 Domino，由于对增量更新的处理需要较少的计算资源，使得实时的推荐成为可能。

首先，矩阵 $R_{m,n}$ 被存储在 HBase 表 (*rating-table*) 中，这种大规模的稀疏表恰好是 HBase 存储的强项：每一个用户作为一行，其观看过的所有电影的评分信息存储在列'ratings:[* any movie]' 中，即都存储在 ratings 这个列族中，而每一部电影都是列族中单独的一个列，如表4.5所示。Domino 实现中，我们需要设置一个触发器监控 rating-table 的列族 ratings，每当用户其中数据进行了修改 (比如修改了 $r_{i,j}$)，我们就需要重新计算 U 的第 i 行和 M 的第 j 列的值，以使其能够拟合新的评分，并把新的 U 和 M 数据并且写入到聚合触发器 (对应的表结构如表4.6) 中。与 PageRank 类似，聚合触发器将根据新的 U 和 M 的值来重新计算 rating-table 中的值，当新的值写入 rating-table 的时候又会触发下一轮对 U 和 M 的生成。不同的是，判断迭代是否应该中止的条件不是判断连续两次数据之差是否小于 ϵ，而是判断新的值和最初始值的差是否小于 ϵ。由于 HBase 支持多版本数据存储，这一点也很容易实现。

表 4.5 HBase 中的 *rating-table* 表结构

Users	*ratings:[* any movie]*	...
U_1	$p_{11}, p_{12}, p_{13}, \ldots$	...
U_2	$p_{112}, p_{21}, p_{32}, \ldots$	...
...	..., ...	...

ALS 算法实现中需要特别主意的是 ALS 算法对执行顺序的敏感性：ALS 算法执行过程中，来自不同运算流程的更新如果没有序列化最终会影响 U 和 M 的收敛性。因此在 ALS 的 Domino 实现中需要使用严格同步模型来实现。

表 4.6 *ALS* 聚合表的结构

Entity	*vector:d-vector*	...
U_1	$(v_1^{u1}, v_2^{u1}, ...v_d^{u1})$	...
M_1	$(v_1^{m1}, v_2^{m1}, ...v_d^{m1})$	...
...	..., ...	...

4.5.3　K-means 算法

数据挖掘中，K-means 也是一个被高频使用的聚类分析工具，它能够将未打标签的 n 个观测数据集根据彼此之间“距离”的远近分割成为 k 个子类。K-means 的正规定义如下：给定一个观察数据集 $(x_1, x_2, ...x_n)$，其中每一个数据都是一个 d 维的实向量，k-means 算法的目标就是将这 n 个数据分割成为 k 个集合 $((k \leq n))$，S=$(S_1, S_2, ..., S_k)$，使得单个集合内部所有点距离平均点的举例平方和最小：

$$min\{\sum_{i=1}^{k}\sum_{x_j \in S_i} ||X_j - \mu_i||^2\} \tag{4.6}$$

K-means 本身是 NP 问题，常见的解法是采用迭代逐步逼近的贪心算法。首先随即指定 k 个初始的聚类，其中每一个聚类的均值点是 $m_1, ..., m_k$，接下来算法就迭代的进行下面两步：1）将每一个观测数据指派给它距离最近的集合；2）指派完所有的节点之后，计算新生成的集合的均值点。最终算法发现所有的观测数据的所属关系不再变化，就认为已收敛。

由上面的介绍我们可以知道 k-mean 算法是一个 CPU 密集型的计算任务。直观的来看，Domino 提供的基于触发器的模型似乎和这种简单的计算密集型任务的关系不是很明确，实现起来也无从下手。这里介绍一个技巧：从收敛条件开始分析。我们注意到当 k-mean 发现所有节点的归属关系不再发生改变的时候就收敛结束了，那么我们第一个触发器一定要负责监控所有节点的归属关系。只有这样，我们才有可能正确的在收敛时停止运行。确定了触发器的检测表以及列族的信息后，我们就可以推断出 HBase 表的结构。比如在本例中，我们需要创建表 *cluster-table* 来存储所有的 k 个聚类的中心点 (存储在 *centroids:value* 列中)，并且所有属于该聚类的观测点 (存储在 *clusterNodes* 列中)。

- 配置一个触发器 (*CentralTrigger*) 来监控表 *cluster-table* 中的列 *centroids:value*。

- 配置一个触发器 (*ClusterNodeTrigger*) 来监控同一个表的 *clusterNodes* 列。

程序刚开始执行的时候，首先产生随机的聚类中心点并且写入到表 *cluster-table* 中，当然了此时这些聚类总还不包括任何的观测数据。该随机中心点的更新会导致 *CentralTrigger* 中的动作执行，该动作会计算所有的存储在本地的观测单点和所有的聚类中心点的举例，得到最小的距离代表的聚类，并且将本观测点写入到聚类的 *clusterNodes* 列族中。此时 *ClusterNodeTrigger* 就会触发执行，它计算新的中心点，并且写入到列 *centroids:value* 中。在 k-means 的实现中，不同的迭代之间需要严格的同步来保证最后结果的正确性，因此在这里我们使用了 Domino 提供的严格同步模型来实现该代码。

通过上面的分析可以看出来，在 Domino 模型下实现类似 k-means 的算法是有一些技巧的，它不像使用触发器模型写一个分布式网络爬虫那么直观，并且性能也很大程度上依赖着数据在不同节点上的存储时的平衡度。不过，我们依旧可以看出作为一个通用的编程模型，Domino 依然能够为 k-means 这样的应用提供完善的支持。虽然直接计算 k-means 的效率和使用 MapReduce 模型相比优势并不明显但是如果考虑到不断变化的观察点求聚类的话，那么整个计算复杂度则依旧明显由于任何 MapReduce 的解决方案。

4.6　实验分析

4.6.1　实验环境设置

我们通过对之前实现的多个应用程序 (分布式爬虫、PageRank、协同过滤以及 k-means 算法) 对 Domino 模型和运行时系统进行性能评测，对应比较版本为 MapReduce 实现。

大部分的实验都基于一个 9 个节点的本地集群实现，所有的节点都包含一个 Xeon 双核 2.53GHz 处理器以及 6GB 的内存，通过 1Gb 的以太网交换机连接。除此之外，我们还在 Amazon EC2 环境下测试了 Domino 大规模扩放的性能。我们的 EC2 集群包括 64 台高性能集群实例 (cc 1.4xlarge)，每一台服务器都包括了两个四核 Xeon X5570 处理器，20GB 的内存以及 10Gb 的以太网连接。

在扩放性试验中，我们通过修改不同应用程序的输入数据来观察 Domino 应用的性能。图4.10展示了每一个参与比较的应用程序的默认输入数据集大小以

及最大的输入数据集。PageRank 算法的默认输入来自于我们的分布式爬虫爬取到的 1 百万用户的页面信息，不过在扩放性试验中，输入数据则来自于手工产生的 10 亿用户页面数据。协同过滤算法的输入数据来自于 Netflix 的公开数据及，而在扩放性试验中，我们使用随机生成的十亿条评分信息作为输入。

Application	Input Size	Generated Max Input Size
PageRank Distributed Crawler	**1M Persons** *Sina Microblog*	**1B Persons**
Collaborative Filtering	**10.5M vertex**	**10B vertex**
Domino K-Means	***n = 1M*** ***100 Clusters***	***n up to 1B*** ***100 Clusters***

图 4.10 Domino 性能及扩放性测试输入数据大小

4.6.2　HBase 性能比较

按照我们之前的描述，Domino 系统是基于对 HBase 的 *WAL* 写动作进行监控实现的事件本地监测，并且通过向每一个 *RegionServer* 添加本地周期扫描线程实现容错处理，除此之外，触发器的动作函数也是在 HBase 所运行的 JVM 中运行。因此这些操作一定会降低 HBase 本身的读写性能，本实验中我们将监测 Domino 实现本身对 HBase 系统性能的影响。本节所有实验都基于本地的 9 节点 Domino 集群实现。

HBase 的写性能测试中我们使用了 *PerformanceEvaluation*[**?**] 包，该包最早由 HBase-399 引入来测试 HBase 系统本身的读写性能。整个测试通过启动多个 MapReduce 任务完成，给定一个参数 n，测试程序会在 n 台客户机上同时启动 $10 * n$ 个 map 任务，同时对 HBase 进行插入操作，每一个客户端将插入 1 百万行，每一行正好是 1000 字节。简单的计算可知，每一个 map 任务需要插入十万条。Reduce 任务则比较简单，它会将所有的 map 中成功插入的条数做一个和，最后返回给用户。由于 Reduce 时间尽管没有对 HBase 进行写入，但依然会耗费大量的时间，因此在实际的比较中，我们将这一部分时间删去。为了更好的比较 Domino 和 HBase 的性能，我们根据整个 MapReduce 任务的执行时间计算出每一个 map 的平均执行时间，并且根据每一个 map 的平均写入条数来计算

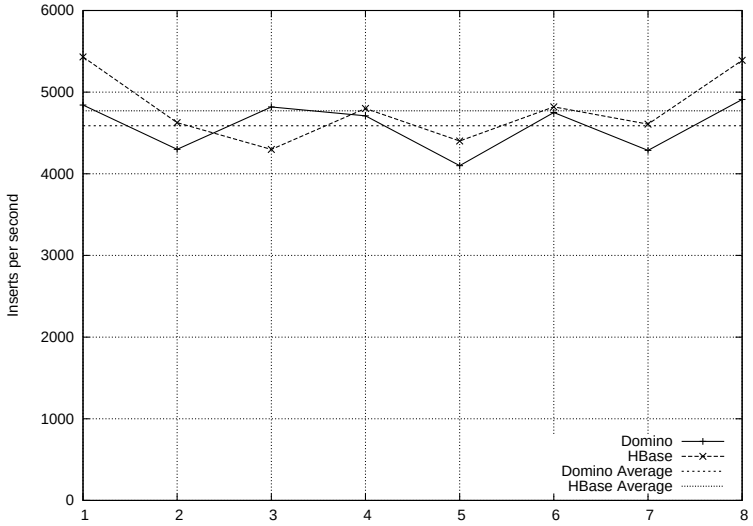

图 4.11 Domino 和 HBase 写性能的对比图 1(Domino 中无 Trigger 运行)

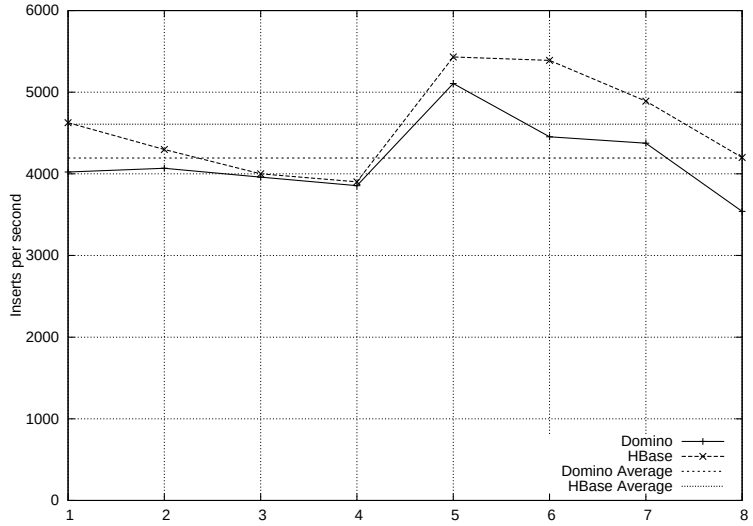

图 4.12 Domino 和 HBase 写性能的对比图 2(Domino 中 Trigger 高频率触发)

出单 map 的插入效率，通过乘以系统允许同时运行的 map 数目，最终得到整个 HBase 集群的写入速度。

图4.11展示了当 Domino 中系统中没有触发器运行时与 HBase 的写性能的多次 (共 8 次) 试验的比较数据，此时所有的性能降低来自于我们的 *WAL Feeds* 实现。我们发现 Domino 性能确有所降低，但是差别非常小 (2%)。图4.12则展示了当 Domino 系统中存在着频繁触发的触发器的时候系统的写性能与 HBase 的

多次比较的结果。测试时我们对 *PreformanceEvaluation* 所写的表设置了触发器，每次写入时，触发器都将执行。为了只检查触发本身的性能损失，触发器执行的动作非常简单，基本不占用 CPU 资源。从图中可以看出，即便在非常频繁的触发情况下，Domino 和 HBase 相比性能下降依然在 10% 以内。我们可以得出结论 Domino 本身实现带来的性能降低是非常小的。

4.6.3　扩展性分析

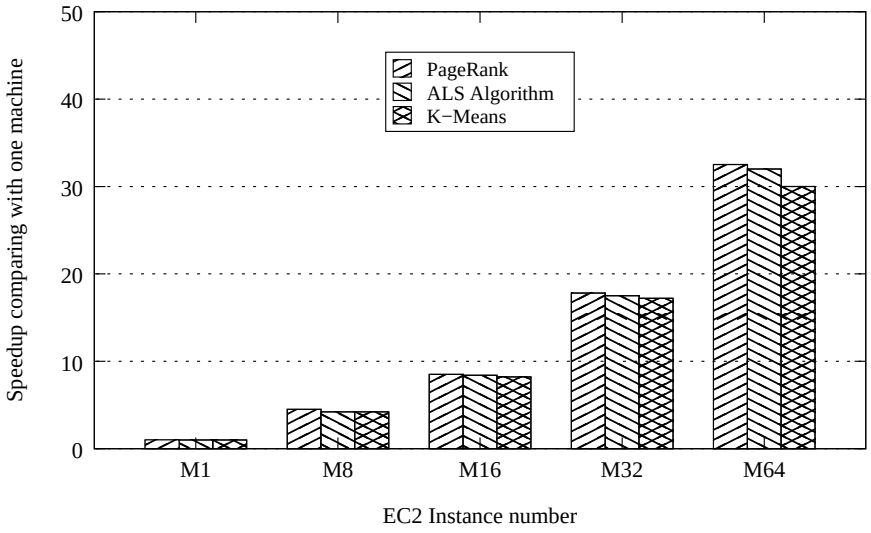

图 4.13 不同的 Domino 应用的性能随服务器数目变化图

图4.13展示了三个测试应用程序 (PageRank,ALS,k-means) 的性能随着服务器数目从 1 增加到 64，而数据数据集始终保持为默认值时的变化图。本实验基于 Amazon 的 EC2 服务实现。从图中可以看出，所有的应用随着服务器数目的增加都体现了很好的加速效果，不过需要大量同步的应用，比如 k-means 和 ALS 算法由于需要进行显式的同步引入了大量的等待，它们的执行速度受到了很大的影响。

为了进一步测试 Domino 上应用的扩放性，检测由于网络通讯带来的性能损失，我们根据服务器数目的提高来增加输入数据集的规模，并最终保证每一台服务器保持处理相同的数据量。对于那些完全独立的子任务来说，理想的情况应该是整个应用的执行时间始终不变。从图4.14可以看出，所有应用都不能达到理想状态，最坏的情况下大约是理想情况的 2 倍，我们认为这种情况是不可

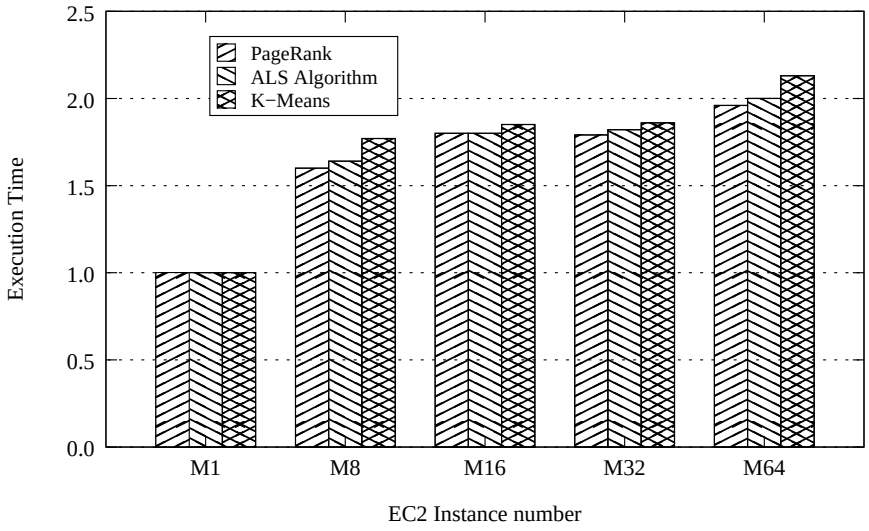

图 4.14 Domino 应用的性能随服务器变化图 (所有数据输入数据集同时增加)

避免的，主要原因是在多轮同步计算中，大量的网络通讯和同步的存在，不过依然可以看出的是，随着服务器数目的扩张 (N)，我们依然可以以同样的时间处理 ($N/2$) 倍的输入数据，Domino 上运行的应用能够保持着非常好的扩放性。

4.6.4　与 MapReduce 比较

我们通过在 Hadoop 中实现了 PageRank 算法用以和 Domino 版本进行比较，而对于 ALS 算法和 k-means 算法我们采用了 Mahout[60] 中的实现。Mahout 是一个基于 Hadoop 实现的饿开源大规模机器学习库，它包括了许多被高度优化的机器学习算法以及它们在 Hadoop 上基于 MapReduce 的实现。由于分布式爬虫没有任何可以基于 MapReduce 模型实现的必要，因此在本文中我们没有将其加入到比较中。

所有的实验都基于本地的 9 节点集群实现，所有的输入数据都是默认的输入数据，并且 Domino 使用了内存加速优化。在实验中，我们分别在 3 个节点和 9 个节点的集群上运行同样的应用以观察 Domino 应用和 MapReduce 应用的性能差别，以及进一步比较两种模型下应用的扩放性。从图4.15可以看出，在 1 百万个点的输入数据下，Domino 的 PageRank 算法性能至少达到了基于 Hadoop 的 PageRank 实现的 10 倍以上的性能。而对于 9 个节点的集群来说，Domino 下性能提升较 Hadoop 的提升也更大。

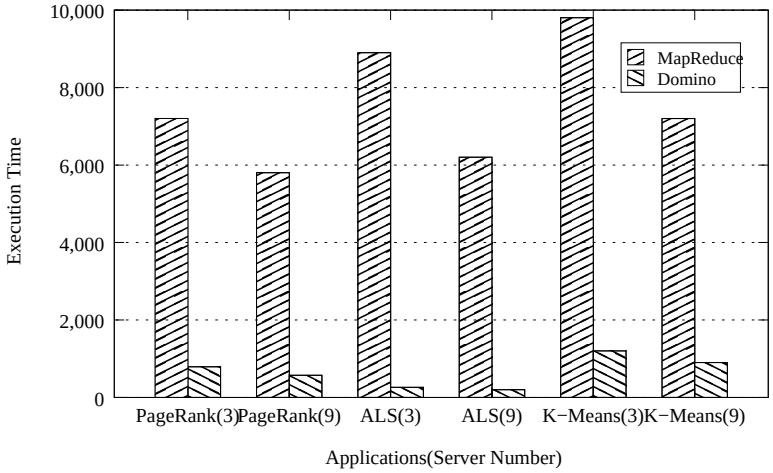

图 4.15 PageRank、ALS、K-means 算法的 MapReduce 实现和 Domino 实现的性能差别 (3/9 个节点)

从图**??**也可以看出 ALS 和 k-means 算法的性能提升同样非常明显，特别是 ALS 算法。这也很容易理解，由于 Mahout 中的 ALS 实现的每一个迭代包括了 3 个连续的 MapReduce 任务，这些 MapReduce 任务中所有的 map 函数并不进行计算，仅仅是为了将数据分发到不同的 reduce 函数中进行计算，这一切在 Domino 中都是不必须的，并且使用基于数据的模型是的 Domino 版本的 ALS 基本上不会进行多余的数据通讯。

4.6.5　递增计算性能

由于采用触发器模型，Domino 模型最大的优势在于对递增计算的支持。PageRank 的应用场景能够很好的体现这一点：通常情况下网络爬虫会不断的爬取新的网页交由 PageRank 进行排序。

图4.16展示了 Domino 和 MapReduce 方案对于部分改变网页的计算性能对比。y 轴表示了部分计算所花费的时间和全部都进行计算所花费时间的比率。基本的 MapReduce 语义中，无论几个页面发生了改变，为了得到正确的结果都需要对整个数据集运行一次完整的 MapReduce 程序。然而在 Domino 中，应用程序只需要对那些发生改变的页面以及与这些页面相关联的页面进行计算，因此 Domino 的递增计算性能远远好于 MapReduce 等系统：对于少量的输入数据集改变，可以立即得到新的结果。

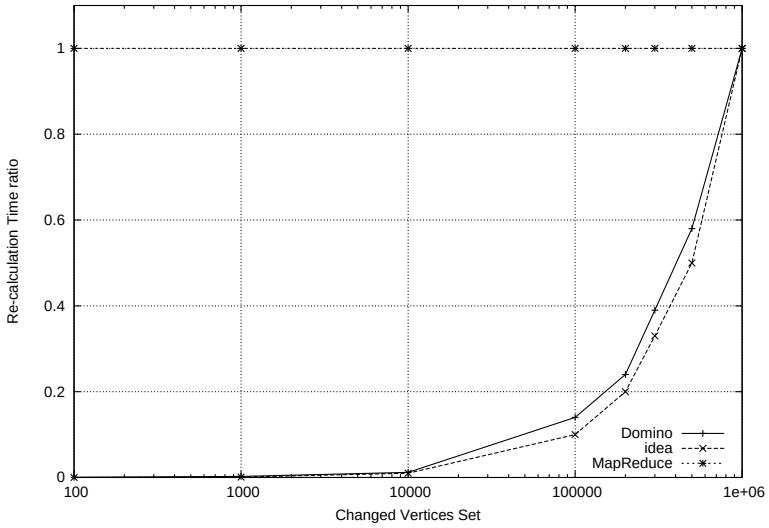

图 4.16 递增的 PageRank 在 Domino 和 MapReduce 下实现的性能对比

4.7　小结

传统意义上认为触发器不足以作为通用的编程模型，尤其在分布式场景下。而 Domino 作为一个通用的基于触发器的编程模型改变了人们对于这一模型的局限性的认识。通过引入聚合模式，我们在触发器这个纯异步模型的基础之上实现了同步语义；通过多版本数据管理，我们在 Domino 中引入了最终同步、严格同步、以及触发器本身的异步模型，为程序开发人员提供了根据他们应用可选的模型。在多种应用程序上的实现也充分证明了该模型的有效性和通用性。我们整合 HBase 存储系统实现了 Domino 运行时系统并且详细描述了各种优化策略，多个应用上的性能试验也证明了 Domino 模型本身的高性能和高扩放性。Domino 并不对底层存储系统提出针对性的需求，只是需要底层存储系统能够提供数据持久化以及数据多版本存储的能力，这说明了 Domino 本身可以和多种存储系统整合起来以面向不同的应用场景，我们也正在进行这方面的工作，我们希望一个和存储系统紧密结合的触发器模型能够作为云计算软件基础架构的核心组件存在。

第五章　结束语

5.1　研究工作总结

博士期间工作主要分为三部分：

- 首先我们对现有的云计算基础软件架构的一个重要实现 Hadoop 进行了系统的研究和应用测试，通过提出一种基于模糊逻辑的异构 Hadoop 集群的配置工具，我们希望能够进一步的提升 Hadoop 模型上应用程序的执行速度。虽然实验证明通过对 Hadoop 大规模集群的配置参数进行自动配置能够很大的提高系统性能，但是由于 MapReduce 模型以及 HDFS 存储系统本身的种种限制，当面对许多基于海量数据的实时应用的时候，显得不太适宜。特别是这些应用需要底层存储平台提供高速的随机读写能力，同时需要对复杂的计算模式进行支持，比如迭代计算和递增计算。

- 因此，我们开始对现有云计算平台中基础软件架构的两个核心组件：存储系统和计算模型进行重新设计和实现，目标就是针对海量数据上的实时应用。Sedna 作为一个完全基于内存的分布式键值存储系统，它极大地提高了分布式存储系统的读写响应速度和读写带宽，实现了和基于磁盘的存储系统一样的数据持久保存的能力，通过层次话的数据中心内存储系统的架构设计，Sedna 能够部署在更多台服务器上支持更大的存储能力，并且更好的支持负载均衡。我们还首次在 Sedna 中引入了基于触发器的文件读原语，通过监控数据的改变来帮助应用程序实时检测到数据的改变。

- 将这个思想进一步放大，我们提出了一个基于触发器的通用编程模型 Domino，它能够原生的支持迭代、递增计算模式。通过引入聚合模式，最终同步等方法，我们进一步解决了在触发器模型下的数据同步问题，极大的扩宽了该模型的应用范围。在不同应用上的实例也证明了触发器模型作为一种独立的编程模型的可行性。

5.2　对未来工作的展望

通过之前的工作，我们建立了一个结合了 Sedna 和 Domino 的云计算环境的机会。通过扩展 Sedna 系统使其支持多版本数据的存储和管理，我们可以轻易的将现基于 HBase 的 Domino 编程模型移植到 Sedna 中。由于 Sedna 是一个完全基于内存的存储系统，通过和 HBase 的简单性能对比也可以看出来，其性能远远超过 HBase，我们相信和基于 Sedna 的分布式存储系统的结合将进一步的提升 Domino 的计算速度。

其次 Domino 模型本身作为一中触发器模型特别适合递增计算，通过我们的之前的描述也容易注意到，对于某些复杂的大数据的数学计算其并不是非常适合，需要进行问题的转换后才能在 Domino 模型下实现。因此我们希望能够将 Domino 模型和现有的基于 MapReduce 的模型进行结合。对于存在大量数据时的首轮计算将采用 MapReduce 的方式计算，之后的改变则使用 Domino 模型进行计算。

另外，对于一个部署了 Domino 和 Sedna 的大规模集群来说，一样存在着资源管理优化配置的问题。我们相信基于模糊逻辑的 Hadoop 异构集群中提出的模糊逻辑思路对于提高集群的效率能够起到很好的指导作用。通过设计 Domino 和 Sedna 使其对动态改变配置更加友好，我们相信未来能够设计和实现一个运行时动态配置大规模集群的工具，用于进一步优化集群性能。

参考文献

[1] HadoopFoundation. http://wiki.apache.org/hadoop/PoweredBy.

[2] Ousterhout J, Agrawal P, Erickson D, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. ACM SIGOPS Operating Systems Review, 2010, 43(4):92–105.

[3] CloudComputing. http://en.wikipedia.org/wiki/Cloud_computing.

[4] Amazon. http://aws.amazon.com/ec2/.

[5] RackSpace. http://www.rackspace.com/.

[6] GAE. Google App Engine, https://developers.google.com/appengine/.

[7] Amazon. Amazon DynamoDB. aws.amazon.com/dynamodb/.

[8] Sandberg R, Goldberg D, Kleiman S, et al. Design and Implementation or the Sun Network Filesystem, 1985.

[9] Howard J H, Kazar M L, Menees S G, et al. Scale and performance in a distributed file system. ACM Transactions on Computer Systems (TOCS), 1988, 6(1):51–81.

[10] Ghemawat S, Gobioff H, Leung S T. The Google file system. Proceedings of ACM SIGOPS Operating Systems Review, volume 37. ACM, 2003. 29–43.

[11] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2008, 26(2):4.

[12] Hadoop. http://hadoop.apache.org/.

[13] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store. Proceedings of ACM SIGOPS Operating Systems Review, volume 41. ACM, 2007. 205–220.

[14] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. Operating systems review, 2010, 44(2):35.

[15] Ongaro D, Rumble S, Stutsman R, et al. Fast crash recovery in RAMCloud. Proceedings of Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011. 29–41.

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

[17] Neumeyer L, Robbins B, Nair A, et al. S4: Distributed stream computing platform. Proceedings of Data Mining Workshops (ICDMW), 2010 IEEE International Conference on. IEEE, 2010. 170–177.

[18] Zaharia M, Chowdhury M, Franklin M, et al. Spark: cluster computing with working sets. Proceedings of Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. USENIX Association, 2010. 10–10.

[19] Low Y, Bickson D, Gonzalez J, et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment, 2012, 5(8):716–727.

[20] Isard M, Budiu M, Yu Y, et al. Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Operating Systems Review, 2007, 41(3):59–72.

[21] Power R, Li J. Piccolo: building fast, distributed programs with partitioned tables. Proceedings of Proceedings of the 9th USENIX conference on Operating systems design and implementation. USENIX Association, 2010. 1–14.

[22] Malewicz G, Austern M, Bik A, et al. Pregel: a system for large-scale graph processing. Proceedings of Proceedings of the 2010 international conference on Management of data. ACM, 2010. 135–146.

[23] Page L, Brin S, Motwani R, et al. The PageRank citation ranking: bringing order to the web. 1999..

[24] HBase Project. Proceedings of http://hbase.apache.org.

[25] Apache. http://hive.apache.org/.

[26] Sharma S. http://www.slideshare.net/ImpetusInfo/ppt-on-advanced-hadoop-tuning-n-optimisation.

[27] [Cluster Setup:Configure Hadoop] http://hadoop.apache.org/docs/r0.19.1/cluster_setup.html.

[28] Murthy A C. Speeding up Hadoop. http://developer.yahoo.com/blogs/ydn/posts/2009/09/hadoop_summit_speeding_up_hadoop.

[29] Ghemawat S, Gobioff H, Leung S. The Google file system. Proceedings of ACM SIGOPS Operating Systems Review, volume 37. ACM, 2003. 29–43.

[30] Redis. http://redis.io/.

[31] Memcached. memcached - a distributed memory object caching system, memcached.org/.

[32] Borthakur D, Gray J, Sarma J, et al. Apache Hadoop goes realtime at Facebook. Proceedings of Proceedings of the 2011 international conference on Management of data. ACM, 2011. 1071–1080.

[33] McKusick M, Quinlan S. Gfs: Evolution on fast-forward. ACM Queue, 2009, 7(7):10–20.

[34] Baker J, Bond C, Corbett J, et al. Megastore: Providing scalable, highly available storage for interactive services. Proceedings of Proc. of CIDR, 2011. 223–234.

[35] Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 2002, 33(2):51–59.

[36] Lamport L. Paxos made simple. ACM SIGACT News, 2001, 32(4):18–25.

[37] MTBF. http://en.wikipedia.org/wiki/Mean_time_between_failures.

[38] Bu Y, Howe B, Balazinska M, et al. HaLoop: Efficient iterative data processing on large clusters. Proceedings of the VLDB Endowment, 2010, 3(1-2):285–296.

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

[40] Bhatotia P, Wieder A, Rodrigues R, et al. Incoop: MapReduce for incremental computations. Proceedings of Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011. 7.

[41] Zhang Y, Gao Q, Gao L, et al. Priter: a distributed framework for prioritized iterative computations. Proceedings of Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011. 13.

[42] Mitchell C, Power R, Li J. Oolong: asynchronous distributed applications made easy. Proceedings of Proceedings of the Asia-Pacific Workshop on Systems. ACM, 2012. 11.

[43] Low Y, Bickson D, Gonzalez J, et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow., 2012, 5(8):716–727.

[44] Rabuzin K, Maleković M, Lovrenčić A. The Theory of Active Databases vs. The SQL Standard. Proceedings of The Proceedings of 18th International Conference on Information and Intelligent Systems, 2007. 49–54.

[45] McCarthy D, Dayal U. The architecture of an active database management system. ACM Sigmod Record, 1989, 18(2):215–224.

[46] Jaeger U, Obermaier J. Parallel event detection in active database systems: The heart of the matter. Active, Real-Time, and Temporal Database Systems, 1999. 159–175.

[47] Gehani N, Jagadish H. Ode as an active database: Constraints and triggers. Proceedings of Proceedings of the Seventeenth International Conference on Very Large Databases (VLDB), 1991. 327–336.

[48] Dayal U, Blaustein B, Buchmann A, et al. The HiPAC project: Combining active databases and timing constraints. ACM Sigmod Record, 1988, 17(1):51–70.

[49] Chakravarthy S, Krishnaprasad V, Anwar E, et al. Composite events for active databases: Semantics, contexts and detection. Proceedings of Proceedings of the international conference on very large data bases. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1994. 606–606.

[50] Stonebraker M, Hanson E, Potamianos S. The POSTGRES rule manager. Software Engineering, IEEE Transactions on, 1988, 14(7):897–907.

[51] Darnovsky M, Bowman J. Transact-sql user's guide. Sybase Inc., Doc, 1987. 3231–2.

[52] Andrews T, Harris C. Combining language and database advances in an object-oriented development environment. Proceedings of ACM Sigplan Notices, volume 22. ACM, 1987. 430–440.

[53] Schlageter G, Unland R, Wilkes W, et al. OOPS-an object oriented programming system with integrated data management facility. Proceedings of Data Engineering, 1988. Proceedings. Fourth International Conference on. IEEE, 1988. 118–125.

[54] Bertsekas D, Tsitsiklis J. Parallel and distributed computation. 1989..

[55] Gonzalez J, Low Y, Guestrin C. Residual splash for optimally parallelizing belief propagation. Aistats, 2009.

[56] Neal R, Hinton G. A view of the EM algorithm that justifies incremental, sparse, and other variants. NATO ASI SERIES D BEHAVIOURAL AND SOCIAL SCIENCES, 1998, 89:355–370.

[57] Macready W, Siapas A, Kauffman S. Criticality and parallelism in combinatorial optimization. SCIENCE-NEW YORK THEN WASHINGTON-, 1996. 56–58.

[58] Smola A, Narayanamurthy S. An architecture for parallel topic models. Proceedings of the VLDB Endowment, 2010, 3(1-2):703–710.

[59] Zhou Y, Wilkinson D, Schreiber R, et al. Large-scale parallel collaborative filtering for the netflix prize. Algorithmic Aspects in Information and Management, 2008. 337–348.

[60] Mahout Project. Proceedings of http://mahout.apache.org.

[61] Al-Fares M, Radhakrishnan S, Raghavan B, et al. Hedera: Dynamic flow scheduling for data center networks. Proceedings of Proceedings of the 7th USENIX conference on Networked systems design and implementation, 2010. 19–19.

[62] Ananthanarayanan G, Ghodsi A, Wang A, et al. PACMan: Coordinated memory caching for parallel jobs. Proceedings of Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012. 20–20.

[63] Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. Communications of the ACM, 2010, 53(4):50–58.

[64] Benson T, Akella A, Maltz D A. Network traffic characteristics of data centers in the wild. Proceedings of Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, New York, NY, USA: ACM, 2010. 267–280.

[65] Burrows M. The Chubby lock service for loosely-coupled distributed systems. Proceedings of Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006. 335–350.

[66] Chakaravarthy V, Choudhury A, Sabharwal Y. Improved algorithms for the distributed trigger counting problem. Proceedings of Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International. IEEE, 2011. 515–523.

[67] Chowdhury M, Zaharia M, Ma J, et al. Managing data transfers in computer clusters with orchestra. SIGCOMM-Computer Communication Review, 2011, 41(4):98.

[68] Chu C, Kim S, Lin Y, et al. Map-reduce for machine learning on multicore. Advances in neural information processing systems, 2007, 19:281.

[69] Dai D, Li X, Wang C, et al. Sedna: A Memory Based Key-Value Storage System for Realtime Processing in Cloud. Proceedings of Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on. IEEE, 2012. 48–56.

[70] HBasePerformanceEvaluation. http://wiki.apache.org/hadoop/Hbase/PerformanceEvaluation.

[71] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for fine-grained resource sharing in the data center. Proceedings of Proceedings of the 8th USENIX conference on Networked systems design and implementation. USENIX Association, 2011. 22–22.

[72] Jiang D, Ooi B, Shi L, et al. The performance of mapreduce: An in-depth study. Proceedings of the VLDB Endowment, 2010, 3(1-2):472–483.

[73] Libenzi D. Linux epoll patch, 2006.

[74] Nishtala R, Fugal H, Grimm S, et al. Scaling Memcache at Facebook. NSDI, 2013.

[75] Oliner A, Ganapathi A, Xu W. Advances and Challenges in Log Analysis. Queue, 2011, 9(12):30:30–30:40.

[76] Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications. Proceedings of Proceedings of the 9th USENIX conference on Operating systems design and implementation. USENIX Association, 2010. 1–15.

[77] Saab P. Memcache Scaling in Facebook. https://www.facebook.com/note.php?noteid=39391378919, 11, 2008. https://www.facebook.com/note.php?noteid=39391378919.

[78] Stoica I, Morris R, Karger D, et al. Chord: A scalable peer-to-peer lookup service for internet applications. Proceedings of ACM SIGCOMM Computer Communication Review, volume 31. ACM, 2001. 149–160.

[79] Vogels W. Eventually consistent. Communications of the ACM, 2009, 52(1):40–44.

[80] Weil S, Brandt S, Miller E, et al. Ceph: A scalable, high-performance distributed file system. Proceedings of Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), 2006. 307–320.

[81] Xie J, Yin S, Ruan X, et al. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. Proceedings of Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on. IEEE, 2010. 1–9.

[82] Xu W, Huang L, Fox A, et al. Online System Problem Detection by Mining Patterns of Console Logs. Proceedings of Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on, 2009. 588 –597.

[83] Xu W, Huang L, Fox A, et al. Detecting large-scale system problems by mining console logs. Proceedings of Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principle, New York, NY, USA: ACM, 2009. 117–132.

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

[85] Zaharia M, Borthakur D, Sen Sarma J, et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. Proceedings of Proceedings of the 5th European conference on Computer systems. ACM, 2010. 265–278.

[86] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 2011.

[87] Zaharia M, Hindman B, Konwinski A, et al. The datacenter needs an operating system. Proceedings of Proceedings of the 3rd USENIX conference on Hot topics in cloud computing. USENIX Association, 2011. 17–17.

[88] Zaharia M, Konwinski A, Joseph A, et al. Improving mapreduce performance in heterogeneous environments. Proceedings of Proceedings of the 8th USENIX conference on Operating systems design and implementation, 2008. 29–42.

[89] Storm Project. Proceedings of http://storm-project.net/.

.1 代码节选

.1.1 [分布式爬虫实例]

```
/**
 * WBContentTrigger.java
 */
package wbcrawler;

//所有提交的触发器都需要实现这个基类HTriggerAction
public class WBContentTrigger extends HTriggerAction{

  //访问微博的凭证
  private final String accessToken = @access_token;
  private WritePrepared writer = null;

  //初始化读写其他表中数据所用的实例。WritePrepared
  public WBContentTrigger(){
    byte[] tableName = "WBRelation".getBytes();
    this.writer = new WritePrepared(tableName);
  }

  private ArrayList<String> getUsersByAPI(String msgId);

  //封装好的事件对象，其中包括了触发事件的表、行以及相关列族的内容数据。
  @Override
  public void action(HTriggerEvent hte) {
    byte[] msgId = hte.getRowKey();
    byte[] msgContent = hte.getNewValue();
    String msgContentStr = new String(msgContent);

    ArrayList<String> aus = this.getUsersByAPI(new String(msgId));
    for (String userId : aus){
      Put p = new Put(userId.getBytes());
      p.add("Activity".getBytes(), "recently".getBytes(),
                  "true".getBytes());
      this.writer.append(p);
    }
    this.writer.flush();
  }
  //过滤器，用来判断这个事件是否应该使触发器执行。
  @Override
  public boolean filter(HTriggerEvent hte) {
    return true;
  }
}
```

```
/**
 * WBUserTrigger.java
 */
package wbcrawler;
public class WBUserTrigger extends HTriggerAction{
   ...
   ...
  @Override
  public void action(HTriggerEvent hte) {
          byte[] userId = hte.getRowKey();
          Timeline tl = new Timeline();
          tl.client.setToken(this.accessToken);
          StatusWapper status = tl.getUserTimelineByUid(new String(userId));
          for (Status s:status.getStatuses()){
                  Put p = new Put(msgId);
                  p.add("Content".getBytes(), "zh".getBytes(), s.getText().
                      getBytes());
                  writer.append(p);
          }
          this.writer.flush();
  }
  @Override
  public boolean filter(HTriggerEvent hte) {
    return true;
  }
}
```

.1.2 [PageRank 实现实例]

```

public class PageRankDist extends HTriggerAction{

  private WritePrepared writer = null;
  private ReadPrepared reader = null;

  public PageRankDist(){
    boolean isAcc = true;
     writer = new WriterPrepared('PageRankAcc'.getBytes(), isAcc);
    reader = new ReadPrepared('wbpages'.getBytes[]);
  }

  @Override
  public void action(HTriggerEvent hte) {
```

```
15      byte[] currentPageId = hte.getRowKey();
16      float fvalue = Float.parseFloat(hte.getNewValue());
17
18      Get g = new Get(currentPageId);
19      g.addFamily("outlinks".getBytes());
20      Result r = reader.get(g);
21
22      outlinks = r.getFamilyMap("outlinks".getBytes());
23      int n = outlinks.size();
24      float weight = fvalue / n;
25
26      for (byte[] link : outlinks.values()){
27        Put p = new Put(link);
28        p.add("nodes".getBytes(), currentPageId, weight);
29        writer.append(p);
30      }
31      this.writer.flush();
32    }
33
34    @Override
35    public boolean filter(HTriggerEvent hte) {
36      byte[] nvalue = hte.getNewValue();
37      byte[] oldValue = hte.getOldValue();
38      float fnv = Float.parseFloat(new String(nvalue));
39      float fov = Float.parseFloat(new String(oldValue));
40      if (Math.abs((fnv - fov)) < 0.001){
41        return false;
42      }
43      return true;
44    }
45  }
```

```
1
2   public class PageRankAcc extends HTriggerAction{
3
4     private WritePrepared writer = null;
5     private ReadPrepared reader = null;
6
7     public PageRankAcc(){
8       writer = new WriterPrepared('wbpages'.getBytes());
9       reader = new AccModeReader();
10    }
11
12    @Override
13    public void action(HTriggerEvent hte) {
14      byte[] pageId = hte.getRowKey();
15      Get g = new Get(pageId);
```

```
16      g.addFamily("nodes".getBytes());
17      Result r = this.reader.get(g);
18
19      nodes = r.getFamilyMap("nodes".getBytes());
20      for (byte[] weight:nodes.values()){
21        String sw = new String(weight);
22        float fw = Float.parseFloat(sw);
23        sum += fw;
24      }
25      Put p = new Put(pageId);
26      p.add("prvalues".getBytes(), "pr".getBytes(), sum);
27      this.writer.append(p);
28      this.writer.flush();
29    }
30    @Override
31    public boolean filter(HTriggerEvent hte) {
32      return true;
33    }
34  }
```

致　谢

在中国科技大学完成本科和硕博连读学业的九年里，我所从事的学习和研究工作，都是在导师以及系里其他老师和同学的指导和帮助下进行的。在完成论文之际，请容许我对他们表达诚挚的谢意。

首先感谢导师 XXX 教授和 XXX 副教授多年的指导和教诲，是他们把我带到了计算机视觉的研究领域。X 老师严谨的研究态度及忘我的工作精神，X 老师认真细致的治学态度及宽广的胸怀，都将使我受益终身。

感谢班主任 XXX 老师和 XX 老师多年的关怀。感谢 XXX、XX、XX 等老师，他们本科及研究生阶段的指导给我研究生阶段的研究工作打下了基础。

感谢 XX、XXX、XXX、XX、XXX、XXX、XXX、XX 等师兄师姐们的指点和照顾；感谢 XXX、XX、XXX 等几位同班同学，与你们的讨论使我受益良多；感谢 XXX、XX、XXX、XX、XXX 等师弟师妹，我们在 XXX 实验室共同学习共同生活，一起走过了这段愉快而难忘的岁月。

感谢科大，感谢一路走过来的兄弟姐妹们，在最宝贵年华里，是你们伴随着我的成长。

最后，感谢我家人一贯的鼓励和支持，你们是我追求学业的坚强后盾。

刘青松

2013 年 4 月 30 日

在读期间发表的学术论文与取得的研究成果

已发表论文:

Conference **Dong Dai**, Xi Li, Junneng Zhang, Chao Wang, Xuehai Zhou. Detecting Associations in Large Dataset on MapReduce. *International Symposium on Parallel and Distributed Processing with Applications(IWCDM), 2013*

Conference **Dong Dai**, Xi Li, Chao Wang, Mingming Sun, and Xuehai Zhou. Sedna: A Memory Based Key-value Storage System for Realtime Processing in Cloud. *In the proceeding of IEEE Cluster'12 (IASDS 2012), 24-28, Sep, Beijing*

Conference Kun Lu, **Dong Dai**. HDFS+: Concurrent Write Improvements for HDFS. *The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2013*

Conference **Dong Dai**, Xi Li, Chao Wang, and Xuehai Zhou. Cloud Based Short Read Mapping Service. *In the proceeding of IEEE Cluster'12, 24-28, Sep, Beijing*

Conference Chao Wang, Xi Li, **Dong Dai**, Gangyong Jia, and Xuehai Zhou. Phase detection for loop-based programs on multicore architectures. *In the proceeding of IEEE Cluster'12, 24-28, Sep, Beijing*

Conference Tao Chen, **Dong Dai** et al. MapReduce On Stream Processing. *International Conference on Intelligent Computing and Intelligent System, 2011*

Conference Gangyong Jia, Xi Li, Xuehai Zhou, **Dong Dai**. Architecture Support Predicting Method for CMP Scheduling. *In Proceeding of International Conference on Computer Research and Development(ICCRD), 2011*

Journal **Dong Dai**, Xuehai Zhou, Feng Yang, and Chao Wang. An auto-configuration tool for heterogeneous hadoop cluster. *In Journal of the Graduate School of the Chinese Academy of Sciences, 2011*

投稿中论文:

Conference **Dong Dai**, Xi Li, Kun Lu, Chao Wang, Xuehai Zhou. Domino: A trigger-based Programming Framework in Cloud. *IEEE Super Computing, 2013*

专利申请情况

专利 **基于 Hadoop 集群的分布式监控系统及其监控方法，受理号:201110060308.1**

参与的科研项目

Period	**07/2011 — 07/2012**
Name	**RFP-07-999 Cisco Research Foundation**
Period	**03/2010 — 08/2011**
Name	**Jiangsu Research Project-BY128**