兰州理工大学

LANZHOU UNIVERSITY OF TECHNOLOGY

毕业设计

题目 基于 Hadoop 的个性化书籍推荐系统的设计与实现

学生姓名 毛烨辉

学　　号 13270212

专业班级 13 级软件工程 2 班

指导教师 张秋余、崔略

学　　院 软件学院

答辩日期 2017 年 6 月 14 日

基于 Hadoop 的个性化书籍推荐系统的设计与实现

Design and implementation of personalized book recommendation system based on Hadoop Platform

毛烨辉（MAO Ye-Hui）

13270212

序 言

互联网的出现和普及给用户带来了大量的信息，满足了用户在信息时代对信息的需求，但随着网络的迅速发展而带来的网上信息量的大幅增长，使得用户在面对大量信息时无法从中获得对自己真正有用的那部分信息，对信息的使用效率反而降低了，这就是所谓的信息超载（informationoverload）问题。

解决信息超载问题一个非常有潜力的办法是个性化推荐系统 ，它是根据用户的信息需求、兴趣等，将用户感兴趣的信息、产品等推荐给用户的个性化信息推荐系统。和搜索引擎相比推荐系统通过研究用户的兴趣偏好，进行个性化计算，由系统发现用户的兴趣点，从而引导用户发现自己的信息需求。一个好的推荐系统不仅能为用户提供个性化的服务，还能和用户之间建立密切关系，让用户对推荐产生依赖。

个性化图书推荐系统作为一种人机交互系统，主要应用信息检索、信息过滤、数据挖掘、人工智能等多种技术和方法为用户提供“书籍推送”服务，帮助用户在互联网海量信息中筛选符合其个性化需求的书籍信息资源，为用户带来全新的服务体验。

随着电子商务、Web2.0 和社交网络的流行和发展，信息推荐系统作为信息服务科学的一个重要研究领域，已得到国内外学者、研究机构和企业界的广泛关注。因此，系统地探讨个性化图书推荐系统的基本原理、技术以及研究热点，无疑将从理论和实践上推动个性化推荐系统的进一步发展。

目录

摘 要

基于 Hadoop 平台的个性化书籍推荐系统是一个能够根据用户兴趣爱好并向用户精准推荐书籍的人机交互系统。该系统使用了 Hadoop 分布式文件系统（HDFS）存储数据。数据的清洗和算法的执行依赖 MapReduce 分布式计算框架。系统每天定时在 Hadoop 平台上进行分布式计算，然后保存运算结果到数据库中，并向用户推送最新的信息和资源。

该系统主要采用了基于协同过滤的推荐技术，能够及时地分析用户的收藏情况，并能向他们推荐具有类似收藏经历的其他用户收藏的书籍。同时还能对用户的收藏行为进行预测，并反馈出该顾客还可能对哪些类型的商品感兴趣，从而把用户的潜在需求转化为真实需求，有效地提高了站点的访问量。

关键词：Hadoop；个性化；推荐系统；书籍；协同过滤算法

Abstract

Personalized Book Recommendation System Based on Hadoop platform is a human-computer interaction system which can recommend books accurately according to the user's interests and interests. The system uses the Hadoop distributed file system (HDFS) to store data. Data cleaning and algorithm execution rely on the MapReduce distributed computing framework. The system computes periodically on the Hadoop platform, and then the results are saved to the database and the latest information and resources are pushed to the user.

The system mainly adopts the recommendation technology based on collaborative filtering, which can analyze the user's collection in a timely manner, and can recommend other books which have similar collection experience to other users. At the same time, it can predict the user's behavior and feedback what kind of goods the customer may be interested in, thereby transforming the potential demand of the user into real demand, and effectively improving the amount of the site's access.

Keywords: Hadoop、Personalise、Recommendation system、Book、Collaborative filtering algorithm

第 1 章 绪论

1.1 个性化推荐系统现状分析

推荐系统最早是来自于国外学者的研究，其推荐技术也比国内的研究深入先进。1997 年 Resnick 和 Varian 首次提出了推荐系统的概念 ，即“推荐系统是利用电子商务网站向客户提供商品信息和建议，帮助用户决定应该购买什么产品，模拟销售人员帮助客户完成购买过程”。直到目前为止该概念仍被广泛引用。自此以后越来越多的学者和研究机构开始关注推荐系统的研究，这些研究工作主要是对推荐系统技术的改进，使推荐系统能适应不同的推荐场所。无论从哪个角度来改进都需要从全局着手，对推荐系统的整体结构要有一个完整的认识。

通常，一个健全的推荐系统由 3 部分组成 ，包括收集用户信息和分析用户偏好、确定推荐算法和推荐的实施。其中，推荐算法是整个推荐系统最核心最重要的部分，是影响推荐系统性能好坏与否的的关键。一般依据推荐算法的不同，将推荐系统分为四种：基于内容的推荐（content-based)、协同过滤推荐（collaborative filtering)、基于网络结构的推荐(network-based)和混合推荐(hybrid)。

协同过滤推荐技术与目前流行的社会化网络研究有交叉点，有丰富的研究基础和广阔的研究前景，是目前研究最多的个性化推荐技术，比较著名的有文档推荐系统 Tapestry、图书推荐系统 Amazon.com、新闻推荐系统 PHOAKS 和电影推荐系统 MoviesLens 等。

协同过滤技术又是最基本的数据挖掘技术。数据挖掘的一个关键问题是数据量。典型的数据挖掘问题包括一个大的数据库，需要从中提取有用的信息。

1.2 个性化书籍推荐系统的目的及意义

网络技术的快速发展将人类带进了一个网络经济的时代，给人类生活的方方面面都带来了巨大的影响。企业利用网络发放海量的商品信息，供用户选择的商品种类和数量也成呈爆炸式增长，面对海量的信息资源，用户无法从中快速有效发现自己感兴趣的商品。因此，用户需要个性化推荐系统能根据用户的喜好针对不同用户推荐其可能感兴趣的商品，这就方便用户找到自己所需的商品。在这种情况下，推荐系统应运而生，它可以利用用户的历史行为

记录来找到用户潜在可能喜欢的资源推荐给用户。个性化书籍推荐系统将书籍作为推荐对象，为用户提供其可能喜欢的书籍。总体而言，个性化图书推荐系统需要具备以下几个方面的功能：

1. 能挖掘潜在用户：很多用户在浏览书籍期刊类网站时并没有想要阅读某一特定书籍，个性化推荐系统应该能够向用户推荐他们可能感兴趣的书籍对象，以增强用户购买的心理需求，从而完成商品购买。

2. 具备网站间交叉销售的能力：用户在购买商品时系统可以向用户推荐其它有价值的商品，这些商品是用户需要但是一时未想起需要购买的商品，这就有效提高了电子商务网站的交叉销售能力。

3. 提高用户对网站的忠诚度：不同于传统的商务模式，个性化推荐系统网站给用户提供了海量的商品以便选择，用户可以很方便的快速查看不同的电子商务系统。个性化推荐系统分析用户的历史购买信息，分析用户的喜好从而向其推荐有价值的商品信息。如果推荐的质量很高，用户就会对个性化推荐系统产生依赖，有利于维持稳定的客源，并提高客户的忠诚度。

1.3 个性化书籍推荐系统的主要内容

随着网络技术的爆炸式增长，人们不得不面对“信息超载”问题。个性化推荐技术是当前解决这个问题最有效的方法之一。然而，当前的很多推荐方法都不可避免的受冷启动问题，用户兴趣随时间变化等问题的困扰，大大影响了推荐系统的性能。该系统将基于协同过滤推荐技术实现精准书籍推荐的功能。

该系统主要做的功能如下：

1. 对书籍种类进行分类，并按照热门程度，进行批量模糊推送。

2. 对书籍发布日期进行排序，批量推送最新书籍。

3. 对用户收藏书籍进行协同过滤计算，并将计算结果按推荐向量的模从大到小排序，将排序前四位精准推送给用户。

4. 对用户关注的好友进行协同过滤计算，并将计算结果按推荐向量的模从大到小排序，将排序结果推送给用户。用户可以查看关注人收藏的书籍，以达到辅助推荐的效果。

第 2 章 系统可行性分析

2.1 技术可行性分析

2.1.1 协同过滤算法理论基础

协同过滤（collaborative filtering)又称为社会过滤，有 Godberg 等研究人员与 1992 年提出，是目前最广泛的推荐技术。与基于内容过滤的推荐方法不同，协同过滤的基本思想是计算用户间偏好的相似度，在用户群中寻找目标用户的相似用户，在相似用户的基础上自动地为目标用户进行信息资源的推荐和预测，即它基于这里的假设：如果用户对一些资源（项目）的评分比较相似，则他们对其他资源（项目）的评分也会比较相似。因此，协同过滤推荐方法针对某类信息资源，首先找出品味和偏好相似的用户，形成用户群；通过分析该群体用户的共同兴趣和对资源的评分，在此基础上对目标用户产生相关的推荐。协同过滤推荐方法的思想非常直观，并且从日常生活中可以找出它所蕴含的道理。

例如：用户 A、B 和 C 都共同选择过《人性记录》、《羊脂球》和《呼啸山庄》这三本书，说明三个用户都对这些书感兴趣。

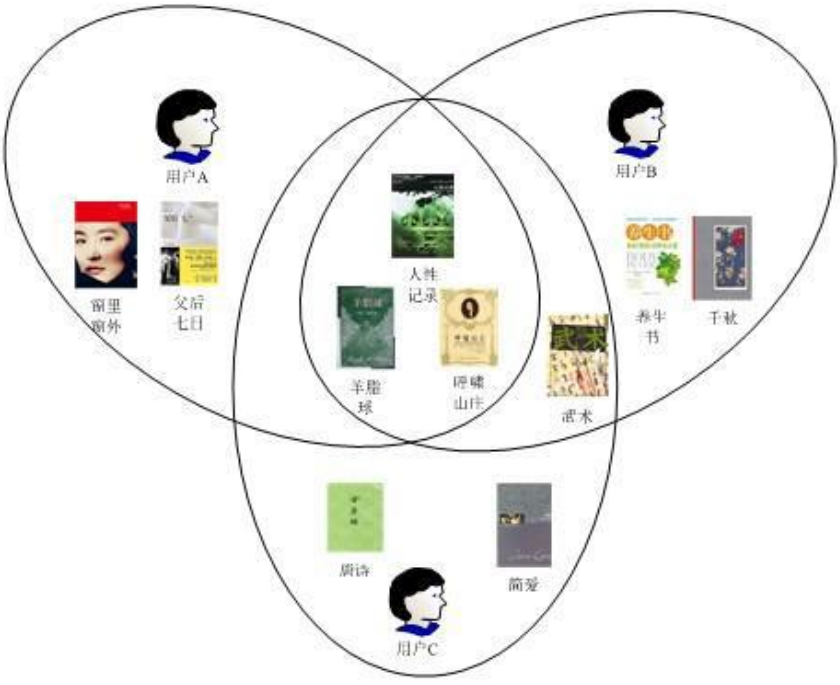


图 2.1 协同过滤原理图

图中重合的部分表示他们的兴趣有相似的部分。所以，当用户 A 是目标用户时，其邻近用户就是用户 B 和用户 C，由图可以知道系统将《武术》这本书推荐给用户 A 的几率最大，因为用户 B 和用户 C 都喜欢这本书。

目前，大部分协同过滤推荐系统使用用户对商品的评分数据作为推荐基础，用户的评分数据分为显示评分（explicit rating）和隐式评分(implicit rating)两类。显示评分通过专门的人机接口直接请求用户显示输入对某些资源的数值评分，该评分表示用户对该资源的偏好程度（一般分值越高代表用户越感兴趣）；隐式评分则不需要用户直接对资源评分，而是根据用户浏览网页是的行为特征，并结合行为科学和心理学的研究成果来分析用户对该资源的偏爱程度，从而预测用户对该资源的评分。显然，显示评分方式存在较大的缺陷，用户必须暂停当前浏览或阅读行为，转而输入对资源的评分，从而导致用户评分数据的稀缺性。

我们可以将兴趣图谱想象成对人们与他们的兴趣之间的关系进行建模的一种方式。兴趣图谱在数据挖掘领域提供了大量可能性，这些主要涉及测量事物之间的相关性从而进行智能推荐，或涉及机器学习中的其他应用。

我们还可以利用用户搜索的书本名称、作家、类别等信息建立统计语言模型。统计语言模型以概率论和数理统计理论为基础，用来计算自然语言序列的概率，使得正确序列的概率大于错误序列的概率。

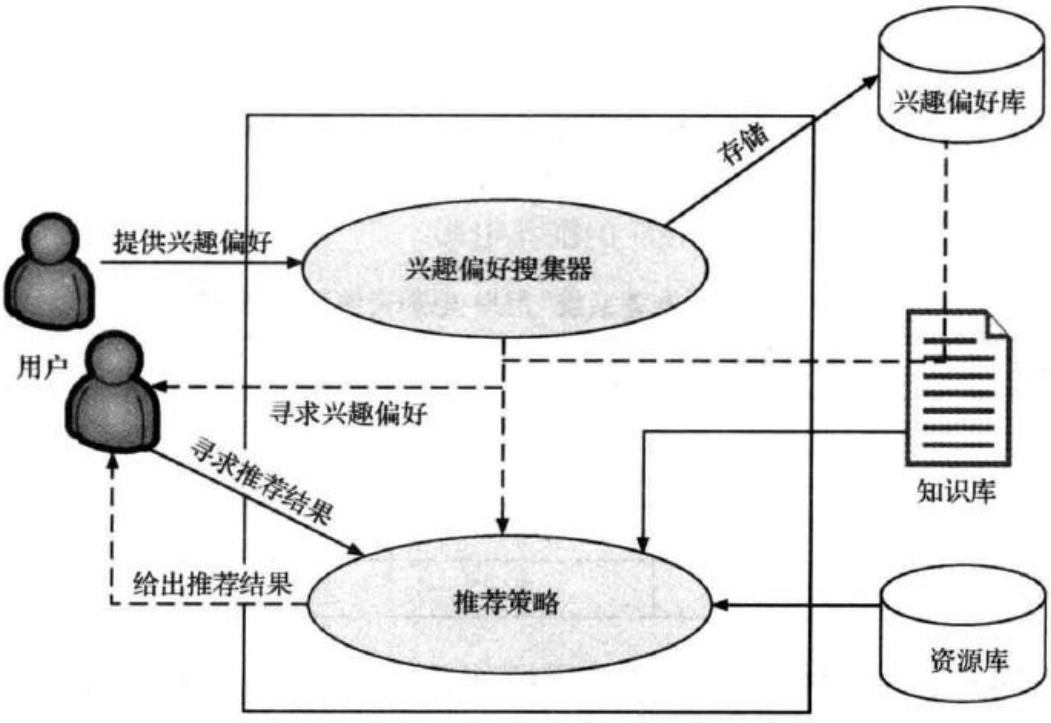


图2.2 推荐系统通用模型

2.1.2 协同过滤技术的优点

1) 在协同过滤中，用户在信息流中决定对一个项目的兴趣或质量，因此可以过滤一些难以通过计算机技术进行内容分析的信息，像书籍、音乐、电影等资源。

2) 可以共享其他人的经验，避免了内容分析的不完全和不精准；同时，可以有效使用其他相似用户的反馈信息，从而减少用户的信息反馈量，加快个性化学习的速度。

3) 具有推荐新信息的能力，可以为目标用户推荐潜在的感兴趣的信息资源。协同过滤推荐主要是根据用户的相似性来推荐资源，这与基于内容过滤的信息推荐有所不同，它比较的是用户描述的文件，而不是资源与用户的描述文件。由于协同过滤推荐是根据相似用户进行推荐的，所以可以为目标用户推荐出新的感兴趣的资源，从而拓宽了信息的推荐范围。

2.1.3 协同过滤推荐算法的流程

传统的信息协同过滤推荐系统用 User 和 Item 来处理推荐事务。User 表示用户，Item 表示资源或项目。信息协同过滤推荐算法的目标是为当前目标用户推荐满足其偏好的新资源(项目)或预测用户对某一未评分资源的评分值。具体推荐过程从一个初始的评分矩阵 User×Item 开始，该评分矩阵记载了 User 对 Item 的主观评分。矩阵 User×Item 要么由用户显示确定，要么由系统隐式推断得出。

一般地，信息协同过滤推荐模型可以描述为：假设 U 是所有用户集合，$U = \{u_1, u_2, \ldots\ldots, u_h\}$，项目对象集合为 $I = \{i_1, i_2, i_3, \ldots\ldots, i_f\}$。每个用户 u_i 都存在一个对资源的评价向量 I_{ui}，用来记录该用户对所有资源的历史评分值。选取目标用户 $u_i \in U$，经典的协同过滤算法包括如下三个过程，如图 2.3 所示。

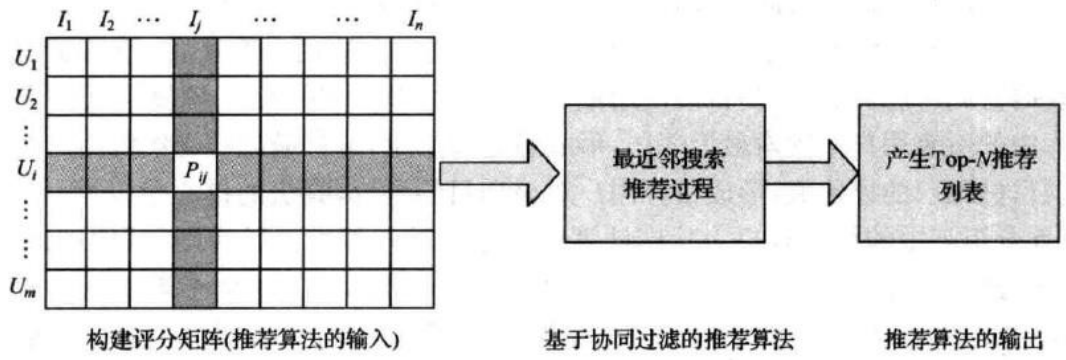


图 2.3 协同过滤推荐算法的流程

1. 构建“用户-资源”评分矩阵

该过程是协同过滤推荐算法的输入过程，主要是收集用户对信息资源的评分、评价行为等，并进行数据清洗、转换，最终形成用户 $U_i \in$ 对各种资源（项目）$I_j(1 \le j \le n)$ 的评分 R_{ij} 代表第 i 个用户 U_i 对资源 I_i 的评分，表示用户 U_i 对资源 I_i 的偏好程度。

2. 最近邻搜索

当用户输入对信息资源的评价并形成评价矩阵后，系统就需要利用推荐算法进行最近邻选择。最近邻可以是当前目标用户的最近邻，也可以是当前目标资源的最近邻。该过程计算目标用户（资源）与系统数据库中各个用户（资源）之间的相似度，将相似度最高的若干个用户（资源）作为最近邻居集。因此，最近邻搜索过程是协同过滤算法的核心。

3. 产生推荐列表

该过程是协同过滤推荐算法的输出过程。系统通过相关推荐算法，在最近邻居集的评价基础上产生推荐，即选择前 N 个目标用户最感兴趣的资源项目，将其主动推送给目标用户。

2.2 经济可行性分析

由于 Apache Hadoop 是开源的项目，所以基于 Hadopp 分布式平台开发软件不会产生任何费用。Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台，使用 Eclipse 开发 Java WEB 项目也时完全免费的。

此外，互联网规模和信息资源的迅猛增长使得人们从互联网中有效获取信息日益困难，尤其是具有携带传播信息功能的书籍。个性化图书推荐系统能够针对用户的信息需求，高效地为其提供高质量的信息获取服务不仅为用户节约了时间成本，还成为当前互联网可持续发展的关键。所以该系统具有经济可行性。

2.3 社会因素可行性分析

随着互联网上数字信息资源的迅速增长，用户通常需要不断手工构造复杂的查询条件以减少无关的返回结果。该系统减少了用户盲目的网上搜索的时间，减轻了用户的负担，提高了信息检索效率和信息获取服务的质量。所以该系统应用市场前景广阔。

该系统面向人群广泛，目标用户定位无年龄段限制，无收入限制，各学历水平均可推广。使用该系统的用户主要集中在青少年，其特点是爱好阅读，性格安静。

法律方面的可行性问题很多，如合同责任、侵犯专利权、侵犯版权等方面，该系统涉及内容和开发过程符合法律规范，可以对外发布。

第 3 章 系统需求分析

3.1 需求分析

软件需求分析就是把软件计划期间建立的软件可行性分析求精和细化，分析各种可能的解法，并且分配给各个软件元素。需求分析是软件定义阶段中的最后一步，是确定系统必须完成哪些工作，也就是对目标系统提出完整、准确、清晰、具体的要求。

深入描述软件的功能和性能，确定软件设计的约束和软件同其他系统元素的接口细节，定义软件的其他有效性需求，借助于当前系统的逻辑模型导出目标系统逻辑模型，解决目标系统“做什么”的问题。

需求分析可分为需求提出、需求描述及需求评审三个阶段。

开发软件系统最为困难的部分就是要准确说明开发什么。最为困难的概念性工作便是要编写出详细的技术需求，这包括所有面向用户、面向机器和其它软件系统的接口。如果做错，这将是会最终给系统带来极大损害的一部分，并且以后再对它进行修改也极为困难。所以我们应当明确我们的需求。

3.1.1 用户需求

Internet 的发展带来了书籍信息资源的极大丰富，称为人们获取书籍信息的重要来源，但是网络信息资源的无序多样、异构，以及冗余度高度等特点也给用户从 Internet 网络中获取信息带来了困难。人们通常借助各种网络信息获取工具，如门户网站、搜索引擎、专业数据索引、RSS 等从网络获取信息。这种拉取服务方式是基于网页的全文检索服务，其优点是信息量大，更新及时；缺点是返回信息过多没有个性化考虑，用户仍需要从检索出的信息中手工选择自己所需要的信息。例如：某用户喜欢读小说，在搜索引擎中搜索“小说”，会拉取到大量信息，但是搜索引擎并不知道用户的兴趣爱好，更不知道用户喜欢读哪一类型的小说，所以用需要通过手工选择的方式在繁杂的信息中再次检索。这个过程不能从根本上帮助用户解决“信息过载”的问题。

因此，用户需要有一个个性化书籍推荐系统能够根据用户的偏好需求从互联网海量书籍信息中寻找满足用户需求的书籍信息，进而采取主动“推送”模式传给用户。

3.1.2 系统需求

系统需要通过在数据库中的所有用户中搜索最近邻居，形成最近邻居集，通过最近邻居集内用户对书籍资源的评价值，形成目标用户对该书籍的评价。以“用户-书籍”评分矩阵中的行（用户）为基础计算用户之间的相似性，即通过用户之间对资源的评分来产生推荐。

系统需要每日定时利用 Sqoop 从 MySQL 数据库中抽取数据到 HDFS 上，并启动 MapReduce 程序计算推荐结果，并定时将推荐结果保存至数据库中，即时个性化书籍推荐信息将被推送至用户下一次访问。

此外，系统还需要每周定时重新计算各类榜单的排行结果，并将其保存至数据库中。

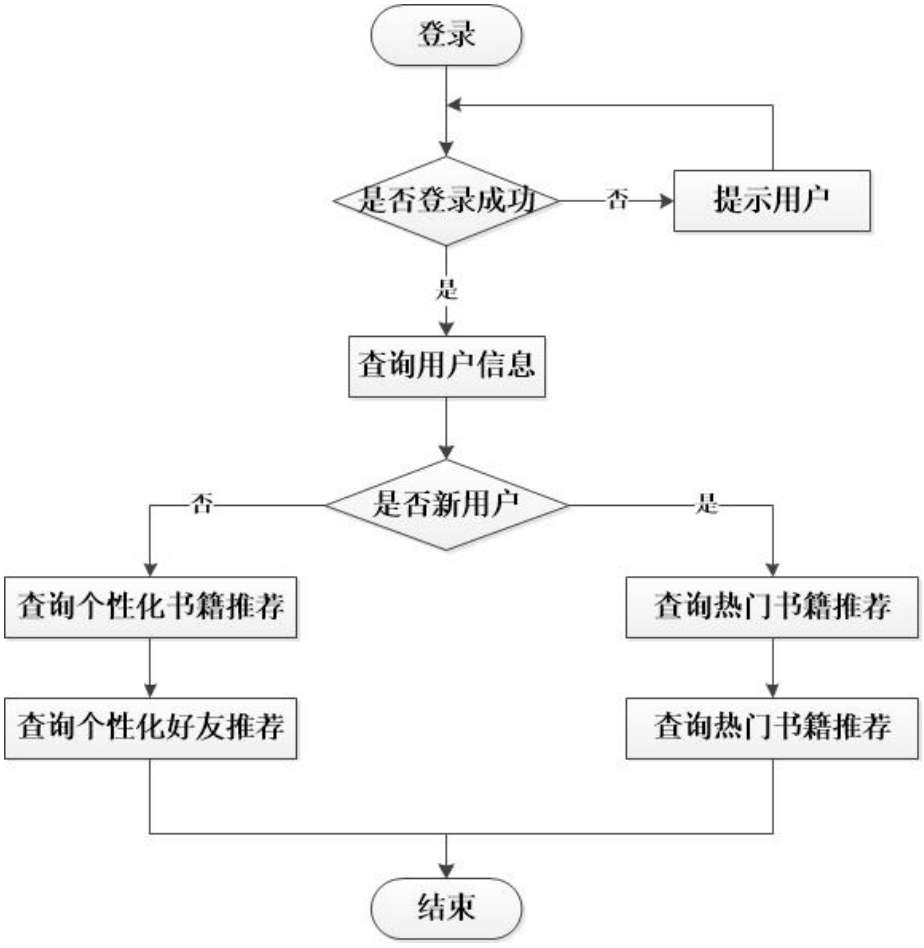


图 3.1 个性化推荐系统流程图

3.1.3 功能需求

通过与用户沟通交流，结合实际情况，明确个性化书籍推荐系统的功能需求，在尽可能

完成用户需求的基础上，使用合理的技术解决方案，多使用图片、图标等友好地展示方式，使其具有友好地界面和良好的用户体验。本系统根据用户需求，遵循软件工程原则开发，采用模块化设计，便于设计管理、维护升级。系统功能模块图如下图 3.2 所示。

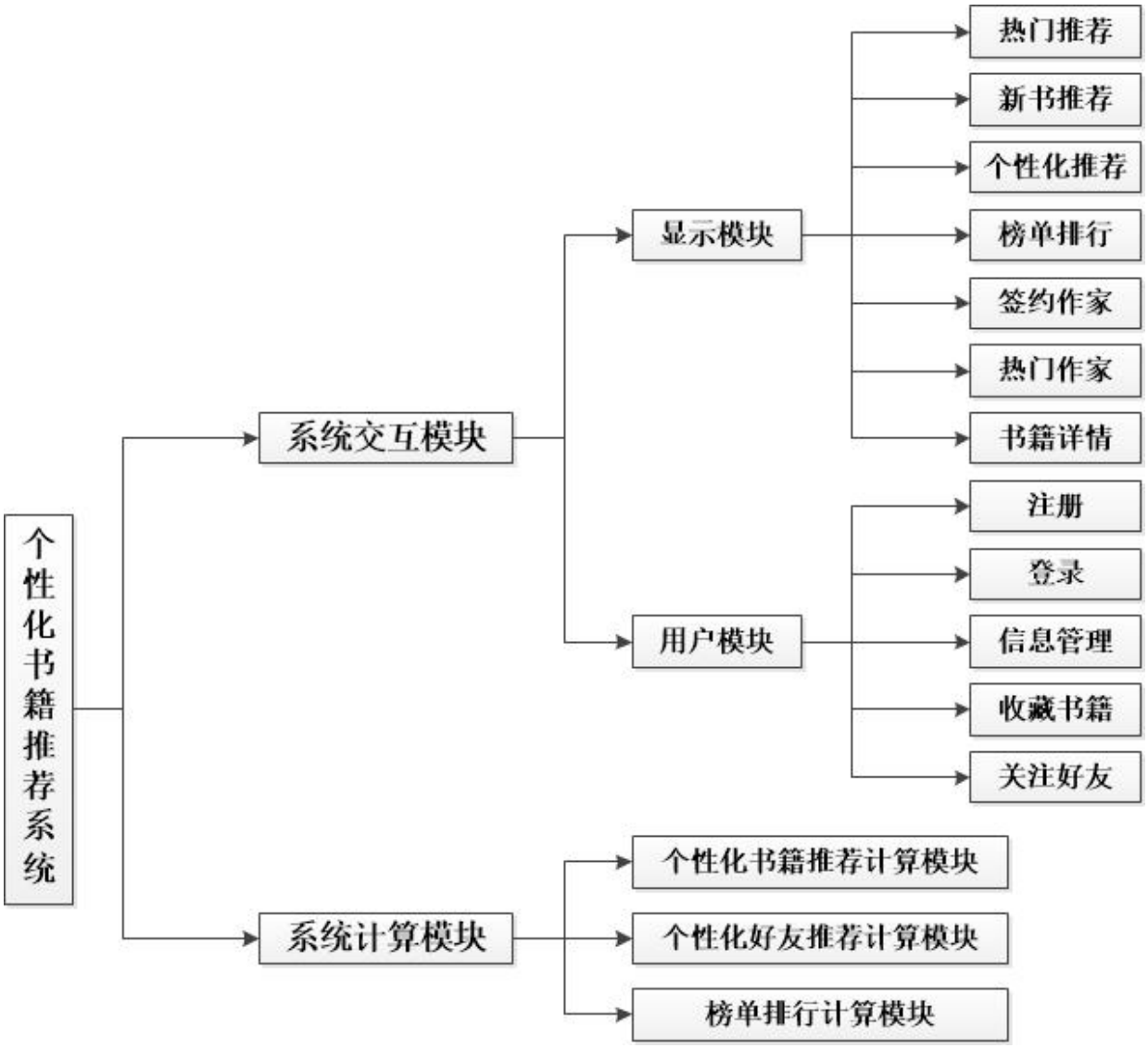


图 3.2 个性化书籍推荐系统功能模块图

具体功能模块描述如下：

个性化推荐系统：划分为系统交互模块和系统计算模块。

系统交互模块：用与用户与系统的交互功能的实现。又可划分为显示模块和用户模块。

系统计算模块：用于系统在 Hadoop 分布式集群上计算推荐结果。可以划分为个性化书籍推荐计算模块、个性化好友推荐计算模块和榜单排行计算模块。

显示模块：用于向用户展示书籍和作家信息。包括热门推荐、新书推荐、个性化推荐、

榜单排行、签约作家、热门作家和书籍详情等。

用户模块：用于用户行为动作的操作。包括注册、登录、信息管理、收藏书籍、关注好友等。

参与者和用例的关系如图 3.3 所示的用例图：

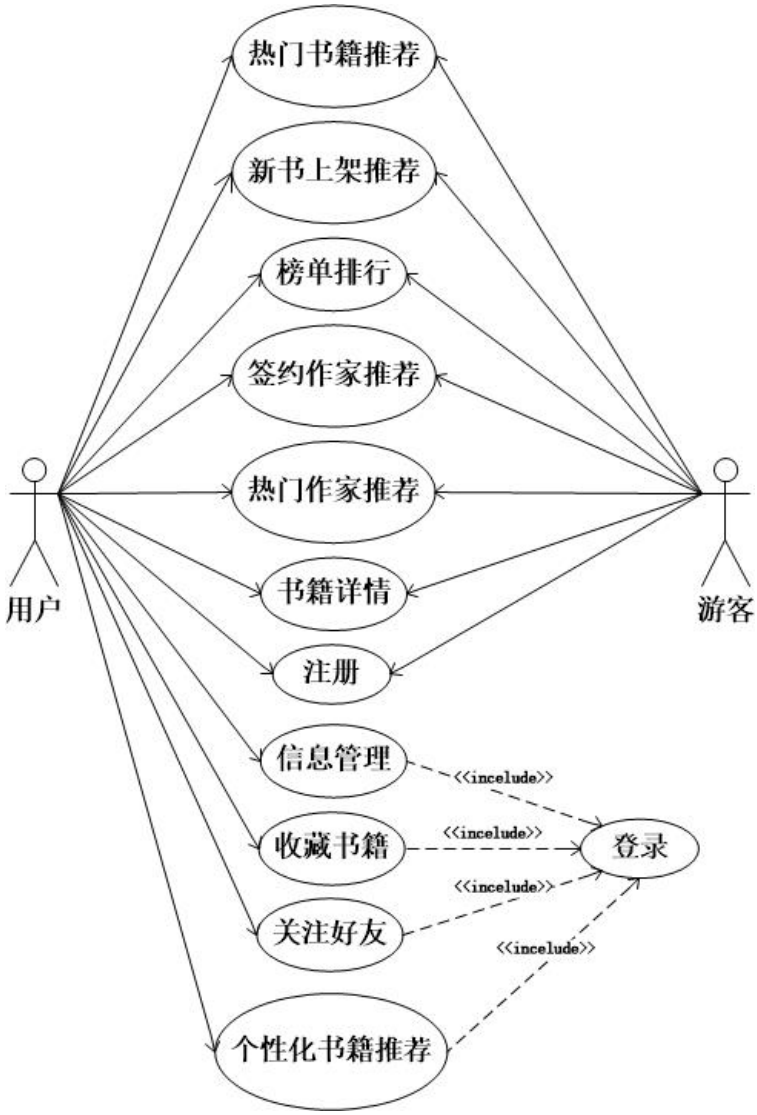


图 3.3 用例图

3.2 系统目标

个性化书籍推荐系统充分考虑用户需求，在尽可能满足用户需求的同时，在界面、用户体验上也做出了努力。个性化书籍推荐系统可以为用户提供热门书籍、新鲜书籍、榜单排行、热门作家、签约作家等信息。用户登录后可以可以收藏书籍，关注好友，查看个性化推荐结

果。界面简洁大方，操作简单。为此，开发的个性化书籍推荐系统应该满足以下条件：

1) 界面简洁大方，美观，数据显示应该比较直观，操作简单易行；

2) 用户登录后推送个性化书籍推荐信息；

3) 用户查看“以文会友”时，推动个性化好友推荐信息；

4) 用户删除收藏书籍后，刷新已收藏的书架书籍；

5) 用户关注粉丝后，刷新粉丝为相互关注；

6) 用户删除关注后，刷新关注人列表信息；

该系统开发在尽可能完成用户需求的前提下，对软件的界面、性能等进行优化，友好地向用户展示其所需要的数据，提升用户体验。

第 4 章 系统采用的主要关键技术

本毕业设计基于 Apache 软件基金会的 Hadoop 分布式基础架构的个性化书籍推荐系统，设计与实现了一套自动推送个性化需求服务信息的系统。系统采用的关键技术有 HDFS、YARN、MapReduce、ZooKeeper、Sqoop 和 Python 等。

4.1 HDFS

Hadoop 分布式文件系统(HDFS)被设计成适合运行在通用硬件(commodity hardware)上的分布式文件系统。它和现有的分布式文件系统有很多共同点。但同时，它和其他的分布式文件系统的区别也是很明显的。HDFS 是一个高度容错性的系统，适合部署在廉价的机器上。HDFS 能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。HDFS 放宽了一部分 POSIX 约束，来实现流式读取文件系统数据的目的。HDFS 在最开始是作为 Apache Nutch 搜索引擎项目的基础架构而开发的。HDFS 是 Apache Hadoop Core 项目的一部分。

HDFS 有着高容错性（fault-tolerant）的特点，并且设计用来部署在低廉的（low-cost）硬件上。而且它提供高吞吐量（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。HDFS 放宽了（relax）POSIX 的要求（requirements）这样可以实现流的形式访问（streaming access）文件系统中的数据。

HDFS 是一个主从结构，一个 HDFS 集群是由一个名字节点，它是一个管理文件命名空间和调节客户端访问文件的主服务器，当然还有一些数据节点，通常是一个节点一个机器，它来管理对应节点的存储。HDFS 对外开放文件命名空间并允许用户数据以文件形式存储。

内部机制是将一个文件分割成一个或多个块，这些块被存储在一组数据节点中。名字节点用来操作文件命名空间的文件或目录操作，如打开，关闭，重命名等等。它同时确定块与数据节点的映射。数据节点负责来自文件系统客户的读写请求。数据节点同时还要执行块的创建，删除，和来自名字节点的块复制指令。

namenode 管理文件系统的命名空间。它维护着文件系统树及整棵树内所有的文件和目录。这些信息以两个文件形式永久保存在本地磁盘上：命名空间镜像文件和编辑日志文件。namenode 也记录着各个块所在数据节点信息，但它并不永久保存块的位置信息，因为这些信息会在系统启动时由数据节点重建。

datanode 是文件系统的工作节点。它们根据需要存储并检索数据块(受客户端或 namenode

调度)，并定期向 namenode 发送他们所存储的块的列表。

没有 namenode，文件系统将无法使用。如果运行 namenode 服务的机器毁坏，文件系统上所有的文件将会丢失，因为我们不知道如何根据 datanode 的块重建文件。

4.2 YARN

Apache Hadoop YARN （Yet Another Resource Negotiator，另一种资源协调者）是一种新的 Hadoop 资源管理器，它是一个通用资源管理系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。

YARN 的基本思想是将 JobTracker 的两个主要功能（资源管理和作业调度/监控）分离，主要方法是创建一个全局的 ResourceManager（RM）和若干个针对应用程序的 ApplicationMaster（AM）。这里的应用程序是指传统的 MapReduce 作业或作业的 DAG（有向无环图）。

YARN 分层结构的本质是 ResourceManager。这个实体控制整个集群并管理应用程序向基础计算资源的分配。ResourceManager 将各个资源部分（计算、内存、带宽等）精心安排给基础 NodeManager（YARN 的每节点代理）。ResourceManager 还与 ApplicationMaster 一起分配资源，与 NodeManager 一起启动和监视它们的基础应用程序。在此上下文中，ApplicationMaster 承担了以前的 TaskTracker 的一些角色，ResourceManager 承担了 JobTracker 的角色。

NodeManager 管理一个 YARN 集群中的每个节点。NodeManager 提供针对集群中每个节点的服务，从监督对一个容器的终生管理到监视资源和跟踪节点健康。MRv1 通过插槽管理 Map 和 Reduce 任务的执行，而 NodeManager 管理抽象容器，这些容器代表着可供一个特定应用程序使用的针对每个节点的资源。YARN 继续使用 HDFS 层。它的主要 NameNode 用于元数据服务，而 DataNode 用于分散在一个集群中的复制存储服务。

要使用一个 YARN 集群，首先需要来自包含一个应用程序的客户的请求。ResourceManager 协商一个容器的必要资源，启动一个 ApplicationMaster 来表示已提交的应用程序。通过使用一个资源请求协议，ApplicationMaster 协商每个节点上供应用程序使用的资源容器。执行应用程序时，ApplicationMaster 监视容器直到完成。当应用程序完成时，ApplicationMaster 从 ResourceManager 注销其容器，执行周期就完成了。

4.3 MAPREDUCE

MapReduce 是一种计算模型，该模型可将大型数据处理任务分解成很多单个的、可以在服务器集群中并行执行的任务。这些任务的计算结果可以合并在一起来计算最终的结果。概念"Map（映射）"和"Reduce（归约）"，是它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。它极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。 当前的软件实现是指定一个 Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的 Reduce（归约）函数，用来保证所有映射的键值对中的每一个共享相同的键组。

MapReduce 任务过程分为两个处理阶段：map 阶段和 reduce 阶段。每个阶段都以键值对作为输出和输入，其类型有程序员来选择。程序员还需要写两个函数：map 函数和 reduce 函数。

Map 阶段的输入是原始数据。我们选择文本格式作为输入格式，将数据集的每一行作为文本输入。键是某一行起始位置相对于文件起始位置的偏移量。

Map 函数的输出经由 MapReduce 框架处理后，最后发送到 reduce 函数。这个处理过程基于键来对键值对来进行排序和分组。如图 4.1 所示。

Job 对象指定作业执行规范。我们可以用它来控制真个作业的运行。我们再 Hadoop 集群上运行这个作业时，要把代码打包成一个 jar 文件（Hadoop 在集群上发布这个文件）。不必明确指定 jar 文件的名称，在 Job 对象的 setJarByClass()方法中传递一个类即可。构造 Job 对象之后，需要指定输入和输出路径的数据。调用 FileInputFormat 类的静态方法 addInputPath()来定义输入数据的路径，这个路径可以是单个文件或目录。可以多次调用 addInputPath()来实现多路径的输入。调用 FileOutputFormat 类中的静态方法 setOutputFormat()来指定输出路径（只能有一个输出路径）。在运行作业前该目录是不应该存在的，否则 Hadoop 会报错并拒绝运行作业。这种预防措施的目的是防止数据丢失。

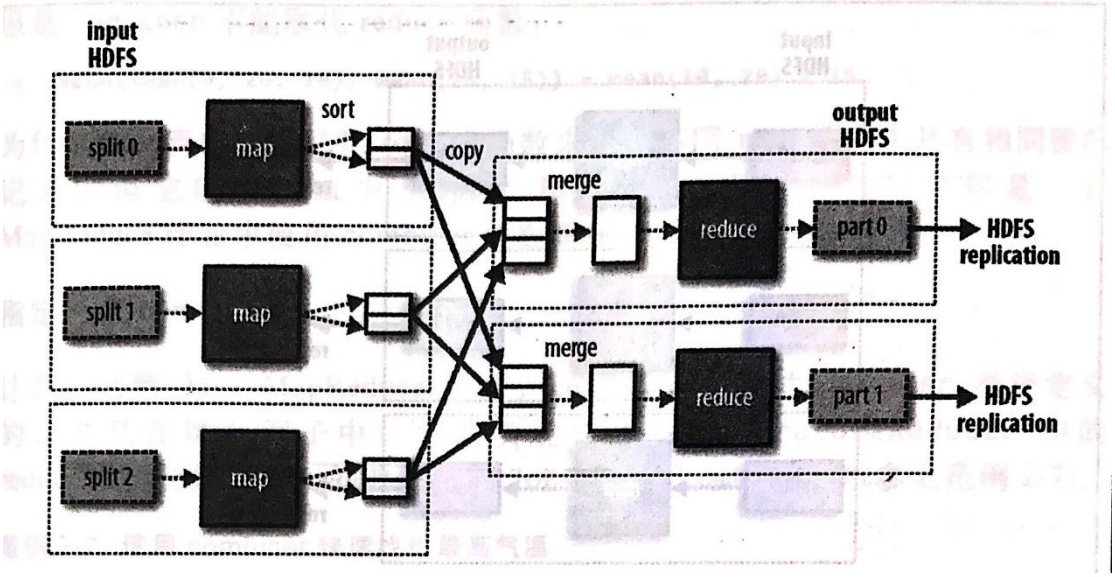


图 4.1 多个 reduce 任务的数据流

4.4 ZOOKEEPER

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，是 Google 的 Chubby 一个开源的实现，是 Hadoop 和 Hbase 的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

ZooKeeper 是以 Fast Paxos 算法为基础的，Paxos 算法存在活锁的问题，即当有多个 proposer 交错提交时，有可能互相排斥导致没有一个 proposer 能提交成功，而 Fast Paxos 作了一些优化，通过选举产生一个 leader (领导者)，只有 leader 才能提交 proposer。

ZooKeeper 的基本运转流程：

1) 选举 Leader。
2) 同步数据。
3) 选举 Leader 过程中算法有很多，但要达到的选举标准是一致的。
4) Leader 要具有最高的执行 ID，类似 root 权限。
5) 集群中大多数的机器得到响应并 follow 选出的 Leader。

4.5 SQOOP

Sqoop是一款开源的工具，主要用于在Hadoop(Hive)与传统的数据库间进行数据的传递，可以将一个关系型数据库（例如：MySQL，Oracle，Postgres 等）中的数据导进到 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导进到关系型数据库中。

Sqoop 项目开始于 2009 年，最早是作为 Hadoop 的一个第三方模块存在，后来为了让使用者能够快速部署，也为了让开发人员能够更快速的迭代开发，Sqoop 独立成为一个 Apache 项目。

对于某些 NoSQL 数据库它也提供了连接器。Sqoop，类似于其他 ETL 工具，使用元数据模型来判断数据类型并在数据从数据源转移到 Hadoop 时确保类型安全的数据处理。Sqoop 专为大数据批量传输设计，能够分割数据集并创建 Hadoop 任务来处理每个区块。

4.6 PYTHON

Python, 是一种面向对象的解释型计算机程序设计语言，由荷兰人 Guido van Rossum 于 1989 年发明，第一个公开发行版发行于 1991 年。

Python 是纯粹的自由软件，源代码和解释器 CPython 遵循 GPL(GNU General Public License)协议。

Python 语法简洁清晰，特色之一是强制用空白符(white space)作为语句缩进。

Python 具有丰富和强大的库。它常被昵称为胶水语言，能够把用其他语言制作的各种模块（尤其是 C/C++）很轻松地联结在一起。常见的一种应用情形是，使用 Python 快速生成程序的原型（有时甚至是程序的最终界面），然后对其中有特别要求的部分，用更合适的语言改写，比如 3D 游戏中的图形渲染模块，性能要求特别高，就可以用 C/C++重写，而后封装为 Python 可以调用的扩展类库。需要注意的是在您使用扩展类库时可能需要考虑平台问题，某些可能不提供跨平台的实现。

4.7 SSM

SSM（Spring+SpringMVC+MyBatis）框架集由 Spring、SpringMVC、MyBatis 三个开源框架整合而成，常作为数据源较简单的 web 项目的框架。

Spring 是一个开源框架，Spring 是于 2003 年兴起的一个轻量级的 Java 开发框架，由 Rod Johnson 在其著作 Expert One-On-One J2EE Development and Design 中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。Spring 使用基本的 JavaBean 来完成以前只可能由 EJB 完成的事情。然而，Spring 的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何 Java 应用都可以从 Spring 中受益。简单来说，Spring 是一个轻量级的控制反转（IoC）和面向切面（AOP）的容器框架。

Spring MVC 属于 SpringFrameWork 的后续产品，已经融合在 Spring Web Flow 里面。Spring MVC 分离了控制器 、模型对象 、分派器以及处理程序对象的角色，这种分离让它们更容易进行定制。

MyBatis 本是 Apache 的一个开源项目 iBatis，2010 年这个项目由 Apache Software Foundation 迁移到了 Google Code ，并且改名为 MyBatis。MyBatis 是一个基于 Java 的持久层框架。iBATIS 提供的持久层框架包括 SQL Maps 和 Data Access Objects（DAO），MyBatis 消除了几乎所有的 JDBC 代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML 或注解用于配置和原始映射，将接口和 Java 的 POJOs（Plain Old Java Objects ，普通的 Java 对象）映射成数据库中的记录。

第 5 章 数据库设计

5.1 数据库概述

数据库系统（DataBase System，简称 DBS）是由数据库及其管理软件组成的系统。是为了适应数据处理的需要而发展起来的一种数据处理系统，也是一个为实际可运行的存储、维护和应用系统提供数据的软件系统，是存储介质 、处理对象和管理系统的集合体。

DBMS 是数据库系统软件的核心，它是用户的应用程序和物理数据库之间沟通的桥梁。用户对数据的一切操作都是在 DBMS 的指挥、调度、控制下进行展开的，而且只能借于 DBMS 实现其数据管理功能。数据库设计是指对于给定特定的应用环境，构造（设计）出某种数据库管理系统所支持的优化的数据库逻辑模式和物理结构，并据此建立数据库及其应用系统，使之能够有效地存储和管理数据，满足各种用户对于应用的各种需求，其中包括信息管理要求和数据处理要求。数据库已经成为现代信息系统的基础和核心组成部分，而数据库设计的好坏直接影响到整个系统设计的效率和质量。

5.2 数据库需求分析

由于个性化推荐系统获得数据的主要方式是用户与服务器端进行交互产生的用户数据和数据库中的历史数据，所以在数据库中存储的数据主要的实体信息有：

1) 书籍的基本信息：包括书籍标识，书籍名称，作家标识，ISBN 号，出版社，出版日期，发布日期，是否免费，价格，综合评分，书籍类别，字数，书籍目录，书籍简介，图片路径等；

2) 作家的基本信息：包括作家标识，作家姓名，作家照片，出生日期，国籍，作家简介，热门度，是否签约，作家标签等；

3) 书单的基本信息：包括书单标识，，书单简介，创建人标识，图片路径等；

4) 用户的账号信息：包括用户账号，用户密码，用户权限，最后登录时间等；

5) 用户的基本信息：包括用户标识，用户昵称，用户签名，用户性别，职业，用户照片，用户省份，用户城市；

6) 用户的拓展信息；用户标识，手机，微信，QQ，等级，是否会员，会员等级，余额等。

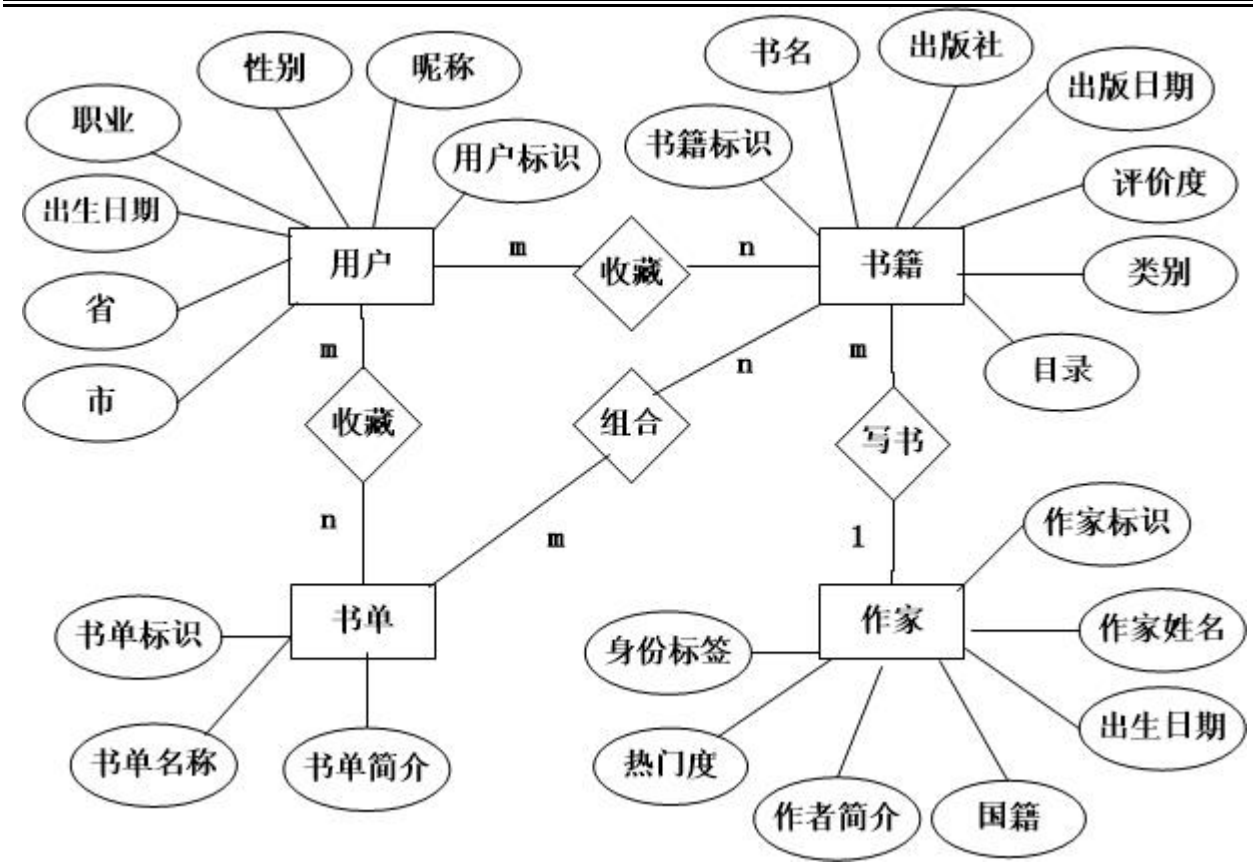


图 5.1 E-R 图

数据流程图（DFD）是在系统分析员在系统设计阶段，对实际构建的系统分析综合后，提取逻辑模型的一个过程，它更关注于过程内数据的处理，而把具体处理数据的物理过程，物理分布忽略。实际上，最初始的数据流程图标准图元只有四个！实体，过程，数据流，数据的存储。并且，数据流的分析过程是逐步对实际过程求精的，从顶层数据流图，到分层数据流图，数据流，过程类型也逐步增加，直到形成最后的数据字典和底层数据流图。需要注意的是数据流图和程序设计中的程序流程图（Flow Chat）是不同的，数据流图关心的是企业业务系统中的数据处理加工的客观过程，并不关心未来电子化处理的加工过程；数据流图中流动的只是数据，并没有控制过程，但在程序流程图当中，必须有控制逻辑。

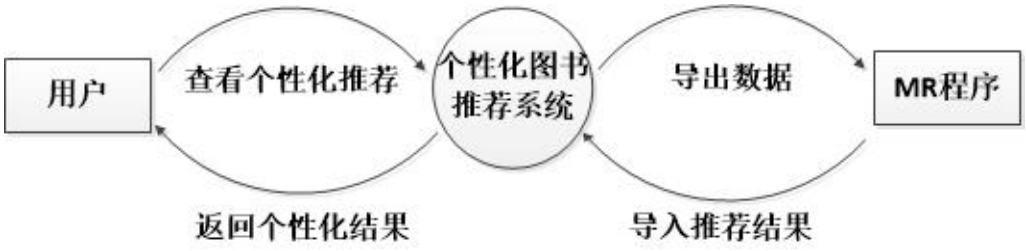


图 5.2 个性化书籍推荐系统顶层图

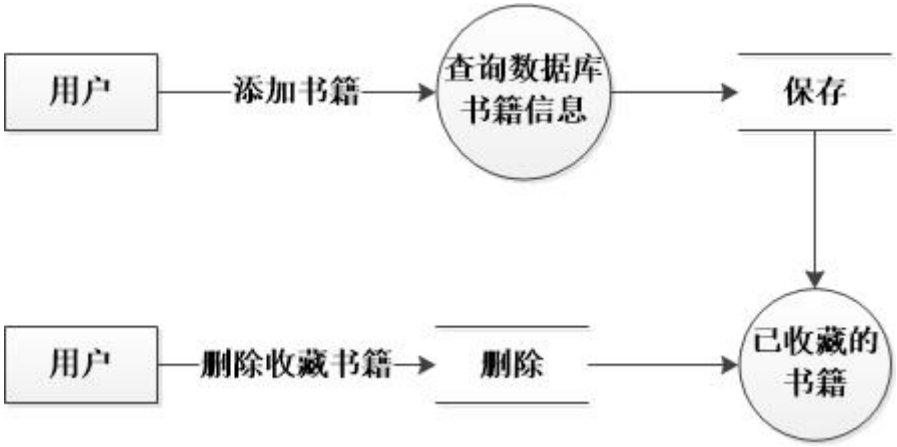


图 5.3 用户书籍操作数据流图

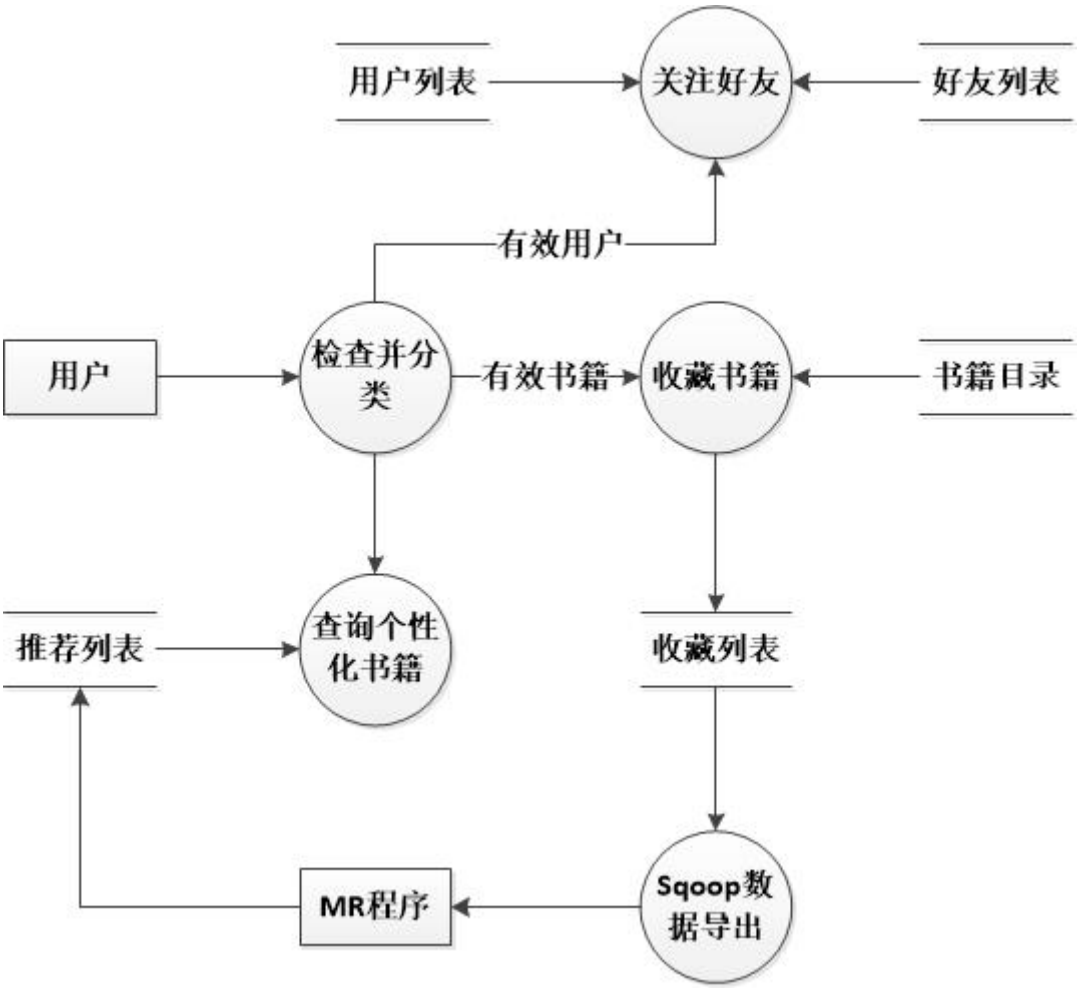


图 5.4 个性化书籍推荐系统

5.3 数据字典

1）管理员账号数据表

表 5.1 表名：administrator

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☑	☐
2	account	账号	varchar	20	☐	☐
3	password	密码	varchar	20	☐	☐
4	power	权限	varchar	10	☐	☐
5	last_login_time	最后登录时间	date		☐	☐

2）用户账号数据表

表 5.2 表名：user

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☑	☐
2	account	账号	varchar	20	☐	☐
3	password	密码	varchar	20	☐	☐
4	power	权限	varchar	10	☐	☐
5	last_login_time	最后登录时间	datetime		☐	☐

3）用户基本信息数据表

表 5.3 表名：user_info

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☑	☐
2	name	昵称	varchar	20	☐	☐
3	signature	个性签名	varchar	100	☐	☑
4	gender	性别	char	1	☐	☐
5	career	职业	varchar	10	☐	☐
6	picture	头像	varchar	100	☐	☑
7	birthday	出生日期	date		☐	☐
8	province	省	varchar	20	☐	☐
9	city	市	varchar	20	☐	☐

4）用户拓展信息数据表

表 5.4 表名：user_ext

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☑	☐
2	cellphone	手机号码	varchar	20	☐	☑
3	wechat	微信	varchar	20	☐	☑
4	qq	QQ	varchar	20	☐	☑
5	level	用户等级	varchar	20	☐	☐
6	if_vip	是否会员	tinyint	1	☐	☐
7	vip_level	会员等级	varchar	10	☐	☑
8	balance	余额	double		☐	☐

5）用户收藏书籍数据表

表 5.5 表名：collection_books

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☑	☐
2	book_id	书籍标识	int	20	☑	☐
3	evaluation	评价	varchar	200	☐	☑
4	reaction_title	读后感标题	varchar	20	☐	☑
5	reaction	读后感	text		☐	☑
6	if_public_reaction	是否公开读后感	tinyint	1	☐	☐
7	recommendation_level	推荐度	int		☐	☑

6）用户搜索书籍数据表

表 5.6 表名：search_books

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☐	☐
2	context	搜索内容	varchar	20	☐	☐
3	search_time	搜索时间	datetime		☐	☐

7）用户好友数据表

表 5.7 表名：friends

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☐	☐
2	friend_id	好友标识	int		☐	☐
3	degree	好友度	int		☐	☐

8）用户黑名单数据表

表 5.8 表名：blacklist

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☐	☐
2	black_id	好友标识	int		☐	☐

9）用户书籍推荐数据表

表 5.9 表名：recommendation_books

序号	字段名	中文名	类型	长度	主键标识	允许空
1	rec_day	推荐日期	date		☐	☐
2	id	用户标识	int		☐	☐
3	book_id	书籍标识	int		☐	☐
4	rec_degree	推荐度	double		☐	☐

10）书籍信息数据表

表 5.10 表名：books_info

序号	字段名	中文名	类型	长度	主键标识	允许空
1	book_id	书籍标识	int		☑	☐
2	title	书名	varchar	20	☐	☐
3	writer_id	作家标识	int		☐	☐
4	isbn	ISBN 号	varchar	25	☐	☑
5	press	出版社	varchar	30	☐	☑
6	publish_date	出版日期	date		☐	☑
7	if_free	是否免费	tinyint	1	☐	☐
8	price	售价	double		☐	☐
9	evaluation_level	评价度	double		☐	☑
10	category	类别	varchar	50	☐	☐
11	num_of_words	字数	int		☐	☑
12	context	目录	text		☐	☑
13	brief_introduction	内容简介	text		☐	☑
14	pictrue	图片	varchar	100	☐	☑

11）热门书籍排名数据表

表 5.11 表名：hot_books

序号	字段名	中文名	类型	长度	主键标识	允许空
1	rec_day	推荐日期	date		☐	☐
2	book_id	书籍标识	int		☐	☐
3	hot_degree	热门度	double		☐	☐

12）最新书籍排名数据表

表 5.12 表名：fresh_books

序号	字段名	中文名	类型	长度	主键标识	允许空
1	rec_day	推荐日期	date		☐	☐
2	book_id	书籍标识	int		☐	☐
3	writer_id	作家标识	int		☐	☐
4	fresh_degree	新鲜度	double		☐	☐

13）作家信息数据表

表 5.13 表名：writers_info

序号	字段名	中文名	类型	长度	主键标识	允许空
1	writer_id	作家标识	int		☑	☐
2	writer_name	作家姓名	varchar	20	☐	☐
3	picture	图片	varchar	100	☐	☑
4	birthday	出生日期	date		☐	☑
5	nation	国籍	varchar	20	☐	☑
6	brief_introduction	作者简介	text		☐	☑
7	hot_degree	热门度	double		☐	☐
8	if_contract	是否签约	tinyint		☐	☐
9	identity_lable	身份标签	varchar	50	☐	☐

14）热门作家排名数据表

表 5.14 表名：hot_writers

序号	字段名	中文名	类型	长度	主键标识	允许空
1	rec_day	推荐日期	date		☐	☐
2	writer_id	作家标识	int		☐	☐
3	hot_book_id	畅读书籍标识	int		☐	☐
4	hot_degree	热门度	double		☐	☐

15）书单信息数据表

表 5.15 表名：booklist_info

序号	字段名	中文名	类型	长度	主键标识	允许空
1	booklist_id	书单标识	int		☑	☐
2	booklist_name	书单名称	varchar	20	☐	☐
3	booklist_synopsis	书单简介	varchar	200	☐	☐
4	creater_id	创建人标识	int		☐	☐
5	picture	书单封面	varchar	100	☐	☐

16）书单数据表

表 5.16 表名：booklist

序号	字段名	中文名	类型	长度	主键标识	允许空
1	booklist_id	书单标识	int		☐	☐
2	book_id	书籍标识	int		☐	☐

17）书单收藏数据表

表 5.17 表名：collection_booklist

序号	字段名	中文名	类型	长度	主键标识	允许空
1	id	用户标识	int		☑	☐
2	booklist_id	书单标识	int		☑	☐

5.4 数据库逻辑结构设计

逻辑设计是将各局部的 E-R 图进行分解、合并后重新组织起来形成数据库全局逻辑结构，包括所确定的关键字和属性、重新确定的记录结构、所建立的各个数据之间的相互关系。

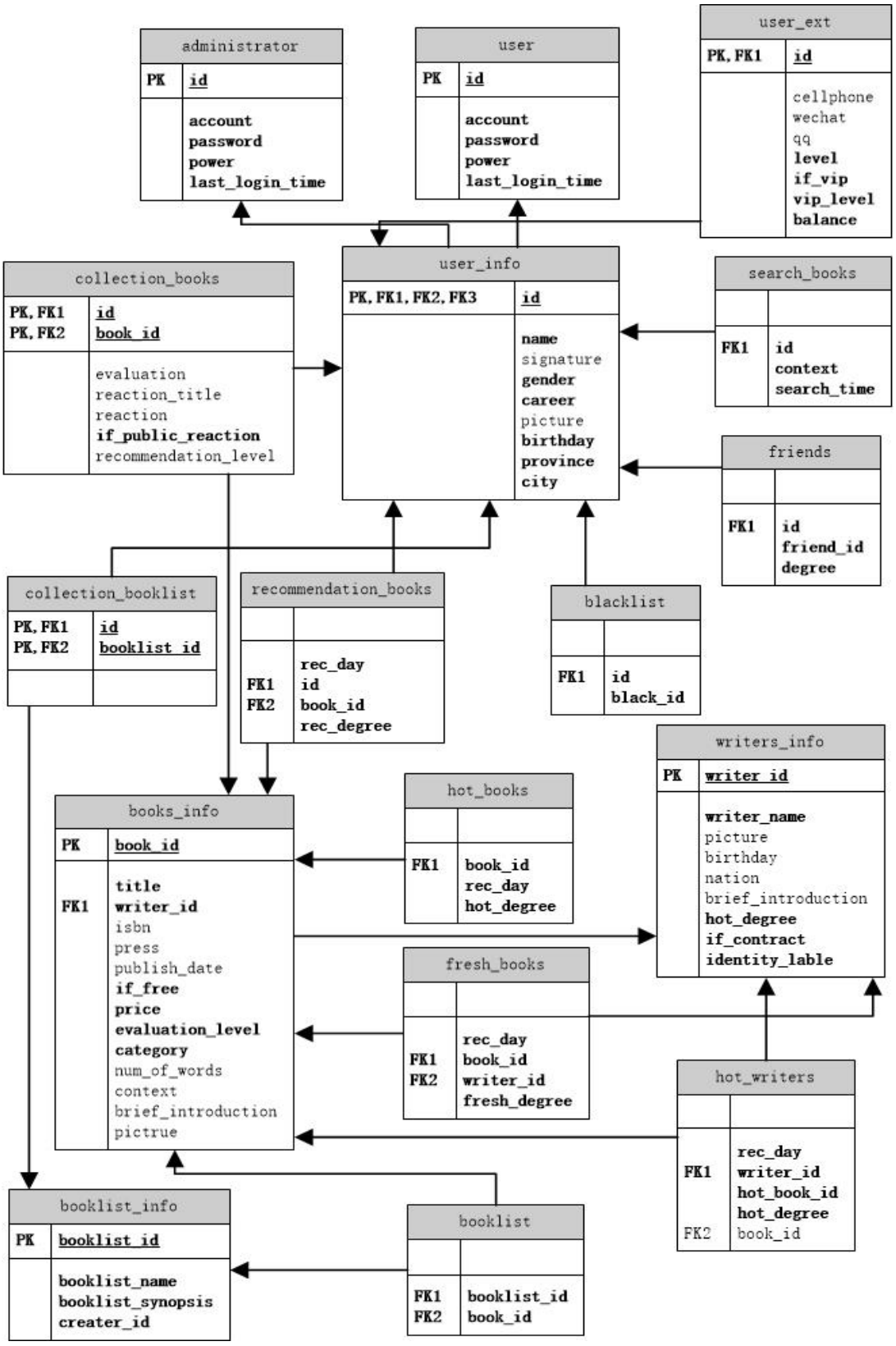

administrator
PK id
account
password
power
last_login_time
user
PK id
account
password
power
last_login_time
user_ext
PK, FK1 id
cellphone
wechat
qq
level
if_vip
vip_level
balance
collection_books
PK, FK1 id
PK, FK2 book_id
evaluation
reaction_title
reaction
if_public_reaction
recommendation_level
user_info
PK, FK1, FK2, FK3 id
name
signature
gender
career
picture
birthday
province
city
search_books
FK1 id
context
search_time
friends
FK1 id
friend_id
degree
collection_booklist
PK, FK1 id
PK, FK2 booklist_id
recommendation_books
FK1 rec_day
FK2 id
book_id
rec_degree
blacklist
FK1 id
black_id
writers_info
PK writer_id
writer_name
picture
birthday
nation
brief_introduction
hot_degree
if_contract
identity_lable
hot_books
FK1 book_id
rec_day
hot_degree
books_info
PK book_id
FK1 title
writer_id
isbn
press
publish_date
if_free
price
evaluation_level
category
num_of_words
context
brief_introduction
pictrue
fresh_books
FK1 rec_day
FK2 book_id
writer_id
fresh_degree
hot_writers
FK1 rec_day
writer_id
hot_book_id
hot_degree
FK2 book_id
booklist_info
PK booklist_id
booklist_name
booklist_synopsis
creater_id
booklist
FK1 booklist_id
FK2 book_id

图 5.5 数据库模型图

第 6 章 系统功能实现

6.1 系统实施环境

- 开发环境：Eclispe for J2EE
- 开发语言：Java，JDK1.8
- 应用服务器：Tomcat 8.0
- 集群服务器：Ubuntu Server 14.10
- 集群客户端操作系统：Ubuntu 14.10
- 数据库：MySQL 5.7
- 部署环境：Hadoop 2.6
- 软件项目管理工具：Maven 3.3.3

6.2 用户书籍数据导出

1. 使用 Sqoop 将 collection_books 从 Mysql 中导入到 HDFS 中。
2. 解压 Sqoop1.4.4.tar 到指定目标
3. 将 MySQL-connector-Java-5.1.16-bin.jar 文件复制到./sqoop/lib 文件夹下
4. 从 MySQL 数据库中将 collection_books 导出 sqoop import --connect jdbc:mysql://192.168.56.1:3306/brs --username root --password 123456 --table collection_books --fields-terminated-by '\t'
5. 导出文件地址为 HDFS 上提交命令用户的家目录下

6.3 协同过滤推荐算法实现

6.3.1 创建 Maven 项目并配置 pom 文件

pom 配置文件如下：

```
<dependencies>
    <dependency>
```

```
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>2.6.0</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.38</version>
    </dependency>
    <dependency>
        <groupId>org.apache.zookeeper</groupId>
        <artifactId>zookeeper</artifactId>
        <version>3.4.6</version>
    </dependency>
</dependencies>
```

6.3.2 解析原始数据

从数据库导出的文件中提取用户标识、书籍标识和用户对书籍评分，剔除无效数据。数据清洗时数据科学项目的第一步，往往也是最重要的一步。许多灵巧的分析最后功败垂成，原因就是分析的数据存在严重的质量问题，或者数据中的某些因素使分析产生偏见，或使数据科学家得出根本不存在的规律。

Map 函数如下：

```
protected void map(LongWritable key, Text value,
```

```
        Mapper<LongWritable, Text, IntWritable, IntWritable>.Context context)
        throws IOException, InterruptedException {
    Parser parser = new Parser();
    IntWritable k = null;
    IntWritable v = null;
    if(parser.parse(value.toString())){
        k = new IntWritable(parser.getId().get());
        v = new IntWritable(parser.getBook_id().get());
        context.write(k, v);
    }
}
```

Reduce 函数如下：

```
protected void reduce(
        IntWritable key,
        Iterable<IntWritable> values,
        Reducer<IntWritable, IntWritable, IntWritable, Text>.Context context)
        throws IOException, InterruptedException {
    String s = "";
    for(IntWritable v : values){
        s = s + v.toString() + ",";
    }
    context.write(key, new Text(s));
}
```

执行命令如下：

```
yarn jar target/BookRecommendation-0.0.1-SNAPSHOT.jar book_rec.step1.BookRecTest -Dinput=/user/maoyh/collection_books -Doutput=/BookRec/step1
```

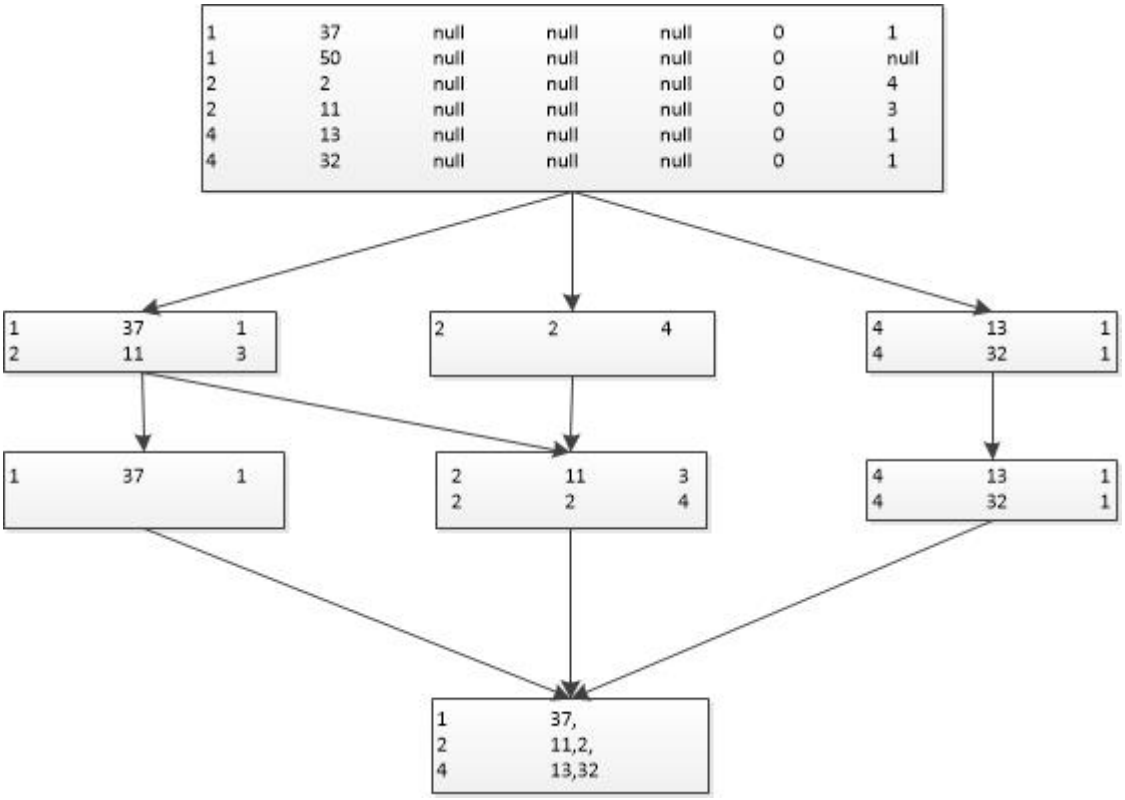


图 6.1 原始数据处理过程图

6.3.3 计算商品共现列表

Map 函数如下：

```
protected void map(LongWritable key, Text value,
        Mapper<LongWritable, Text, Text, IntWritable>.Context context)
        throws IOException, InterruptedException {
    String[] token = value.toString().split("\t");
    String[] splits = token[1].substring(0, token[1].length()-1).split(",");
    for(int i = 0; i < splits.length; i++){
        for(int j = 0; j < splits.length; j++){
            k.set(splits[i].trim() + "\t" + splits[j].trim());
            context.write(k, v);
        }
    }
```

```
}
```

Reduce 函数如下：

```
protected void reduce(Text key, Iterable<IntWritable> values,
        Reducer<Text, IntWritable, Text, IntWritable>.Context context)
        throws IOException, InterruptedException {
    int sum = 0;
    for(IntWritable v : values){
        sum += v.get();
    }
    context.write(key, new IntWritable(sum));
}
```

执行命令如下：

```
yarn jar target/BookRecommendation-0.0.1-SNAPSHOT.jar book_rec.step2.CoMatrixTest -Dinput=/BookRec/step1 -Doutput=/BookRec/step2
```

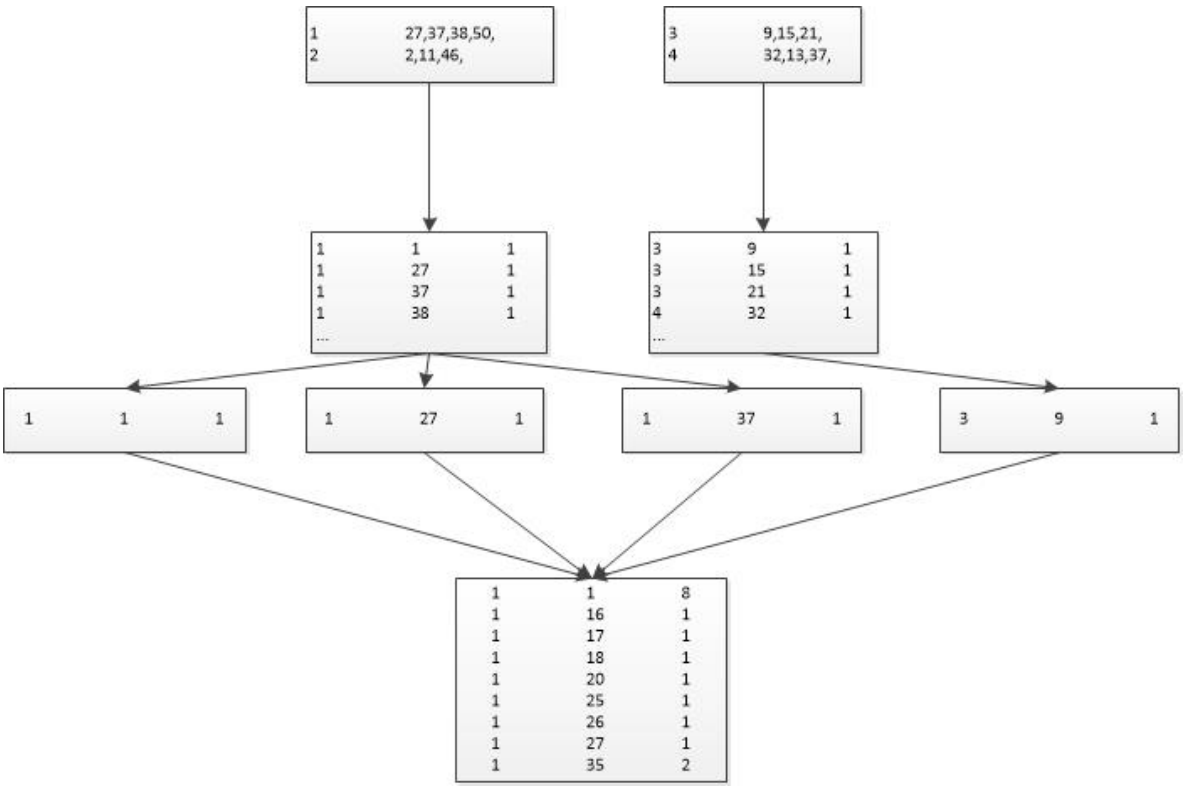


图 6.2 商品共现矩阵列表数据运算图

6.3.4 计算商品共现稀疏矩阵

Map 函数如下：

```
protected void map(Text key, Text value,
        Mapper<Text, Text, Text, BookVector>.Context context)
        throws IOException, InterruptedException {
    String[] split = value.toString().split("\t");
    v.setBook_id(Integer.valueOf(split[0].trim()));
    v.setCount(Double.valueOf(split[1].trim()));
    context.write(key, v);
}
```

Reduce 函数如下：

```
protected void reduce(Text key, Iterable<BookVector> values,
        Reducer<Text, BookVector, Text, Text>.Context context)
        throws IOException, InterruptedException {
    set.clear();
    for(BookVector v : values){
        set.add((new BookVector(v)).toString());
    }
    context.write(key, new Text(set.toString()));
}
```

执行命令如下：

```
yarn jar target/BookRecommendation-0.0.1-SNAPSHOT.jar book_rec.step3.FinalCoMatrixTest -
Dinput=/BookRec/step2 -Doutput=/BookRec/step3
```

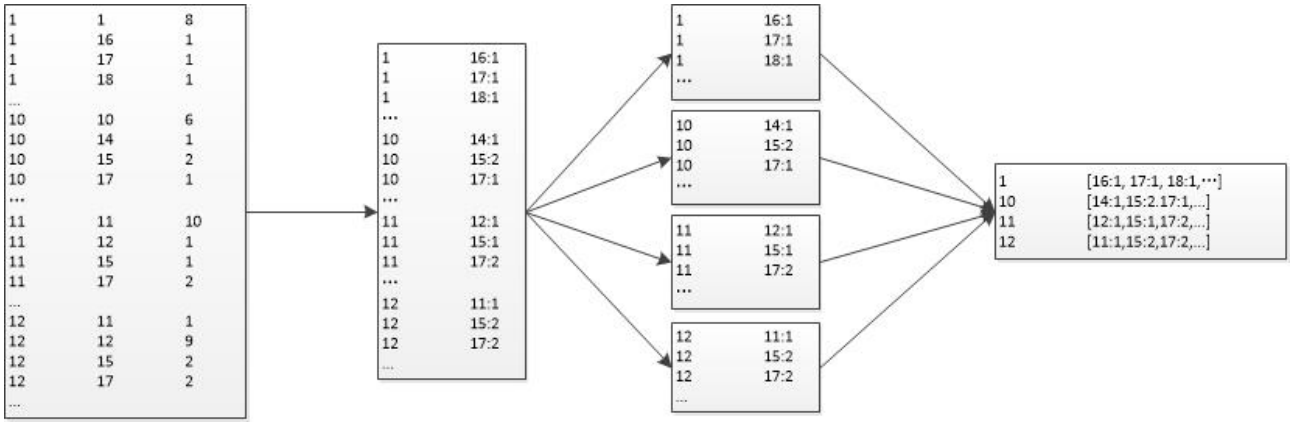


图 6.3 商品共现稀疏矩阵数据运算图

6.3.5 计算用户-书籍参考矩阵

Map 函数如下：

```
protected void map(LongWritable key, Text value,
        Mapper<LongWritable, Text, IntWritable, Preference>.Context context)
        throws IOException, InterruptedException {
    parser.parse(value.toString());
    k.set(parser.getBook_id().get());
    v.setId(parser.getId().get());
    v.setPref(parser.getRecommendation_level());
    context.write(k,v);
}
```

Reduce 函数如下：

```
protected void reduce(IntWritable key, Iterable<Preference> values,
        Reducer<IntWritable, Preference, Text, Text>.Context context)
        throws IOException, InterruptedException {
    l.clear();
    for(Preference v : values){
        l.add(new Preference(v).toString());
    }
```

```
    value.set(l.toString());
    context.write(new Text(key.toString()), value);
}
```

执行命令如下：

```
yarn jar target/BookRecommendation-0.0.1-SNAPSHOT.jar book_rec.step4.ReferenceMatrixTest
-Dinput=/user/maoyh/collection_books -Doutput=/BookRec/step4
```

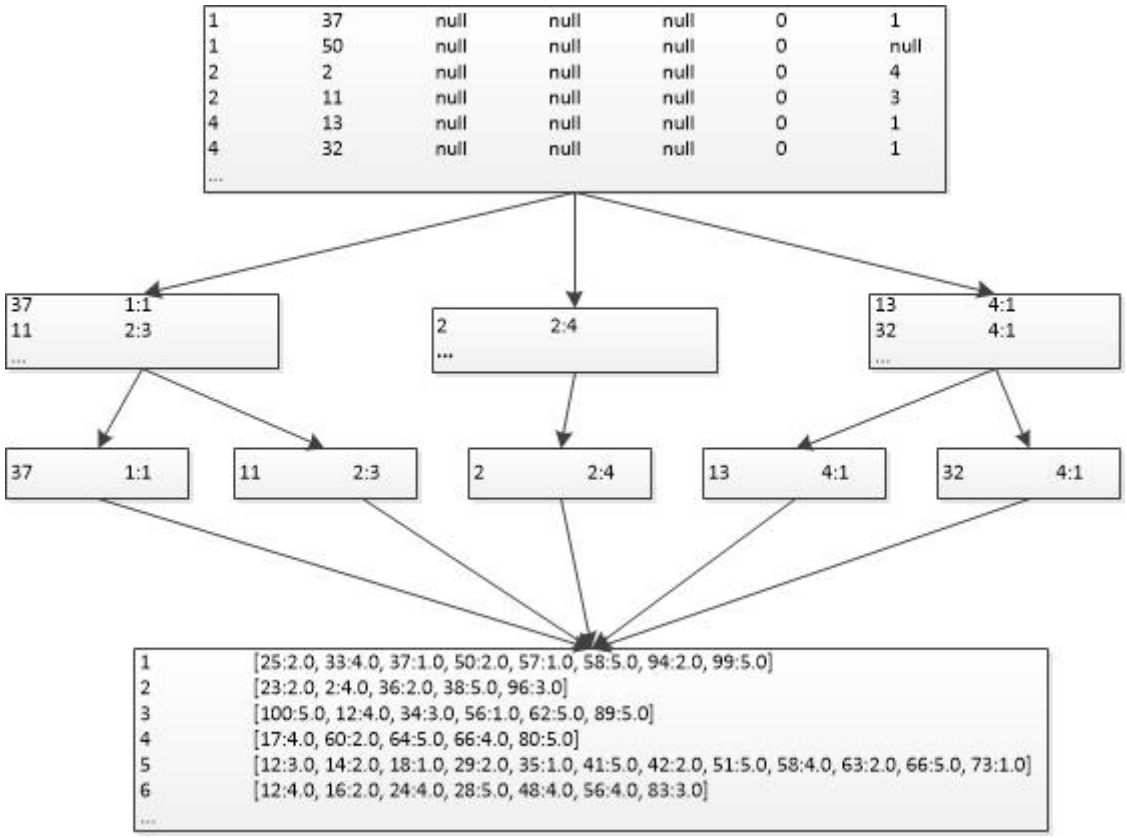


图 6.4 用户-书籍参考矩阵数据运算图

6.3.6 计算用户-书籍推荐列表

Partitioner 分区函数如下：

```
public int getPartition(KeyTuple key, Text value, int numPartitions) {
    return Math.abs(key.getId().hashCode() * 127) % numPartitions;
}
```

Map1 函数如下：

```
protected void map(
        LongWritable key,
        Text value,
        Mapper<LongWritable, Text, KeyTuple, VectorWritable>.Context context)
        throws IOException, InterruptedException {
    String[] split = value.toString().split("\t");
    v.clear();
    if(split != null && split.length == 2){
        k.setId(split[0].trim());
        k.setTag("1");
        parser.parse(split[1]);
        if(parser.isValid()){
            v = parser.getVector();
            context.write(k, v);
        }
    }
}
```

Map2 函数如下：

```
protected void map(
        LongWritable key,
        Text value,
        Mapper<LongWritable, Text, KeyTuple, VectorWritable>.Context context)
        throws IOException, InterruptedException {
    String[] split = value.toString().split("\t");
    v.clear();
    if(split != null && split.length == 2){
        k.setId(split[0]);
```

```
            k.setTag("0");
            parser.parse(split[1]);
            if(parser.isValid()){
                v = parser.getVector();
                context.write(k, v);
            }
        }
}
```

Reduce 函数如下：

```
protected void reduce(KeyTuple key, Iterable<VectorWritable> values,
            Reducer<KeyTuple, VectorWritable, Text, Preference>.Context context)
            throws IOException, InterruptedException {
        Iterator<VectorWritable> iterator = values.iterator();
        refMatrix = iterator.next().toList();
        coMatrix = iterator.next().toList();
        for(Preference r : refMatrix){
            k.set(r.getId().toString());
            for(Preference c : coMatrix){
                v.setId(c.getId());
                v.setPref(c.getPref().get() * r.getPref().get());
                context.write(k, v);
            }
        }
}
```

执行命令如下：

```
yarn jar target/BookRecommendation-0.0.1-SNAPSHOT.jar book_rec.step5.ReduceJoinTest
```

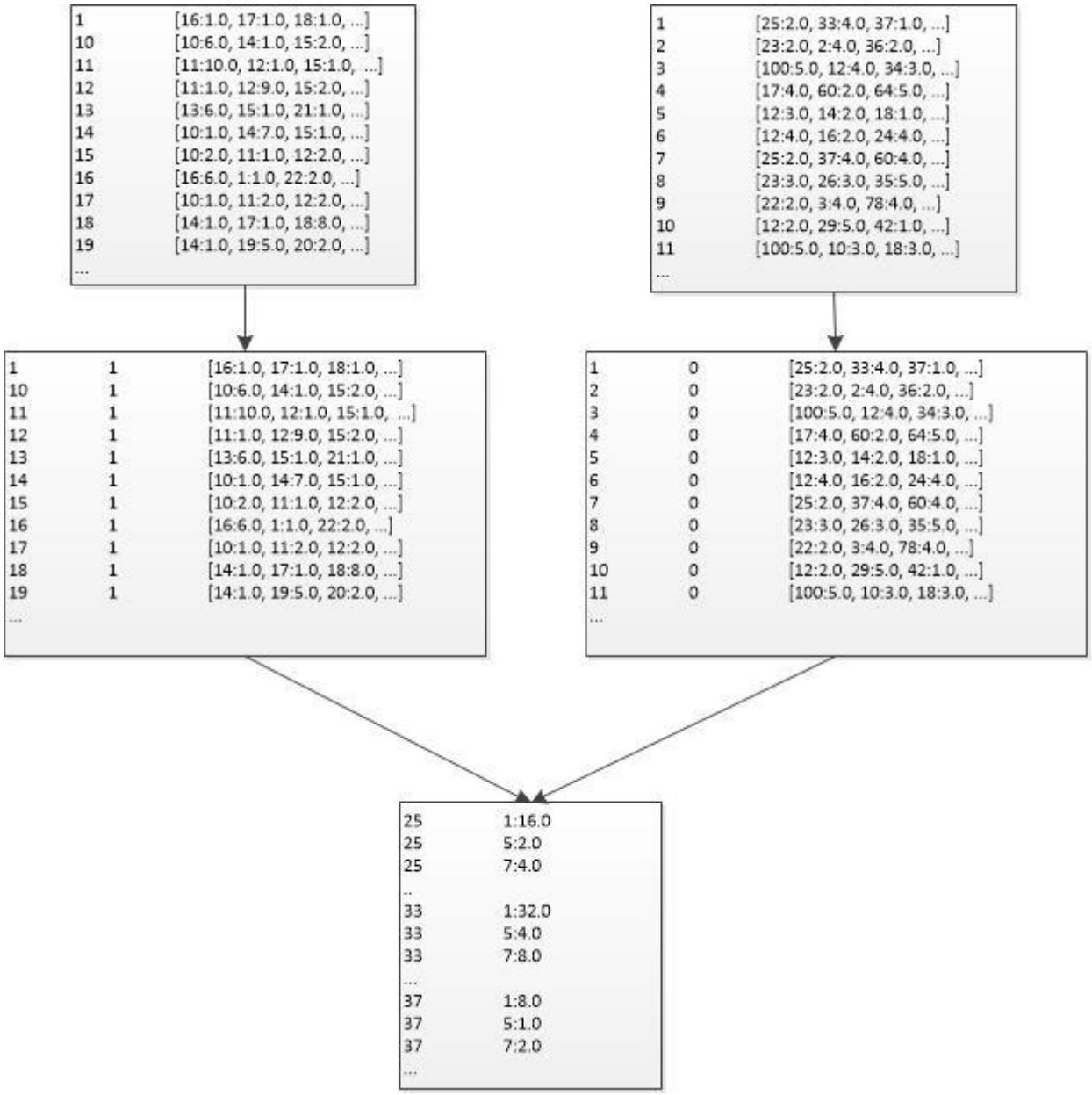


图 6.5 用户-书籍推荐列表数据运算图

6.3.7 计算用户-书籍推荐矩阵

Map 函数如下：

```
protected void map(LongWritable key, Text value,
        Mapper<LongWritable, Text, Text, Text>.Context context)
        throws IOException, InterruptedException {
    String[] split = value.toString().split("\t");
```

```
    String[] token = split[1].split(":");
    k.set(split[0].trim() + "\t" + token[0].trim());
    v.set(token[1].trim());
    context.write(k, v);
}
```

Reduce 函数如下：

```
protected void reduce(Text key, Iterable<Text> values,
        Reducer<Text, Text, Text, Text>.Context context)
        throws IOException, InterruptedException {
    double sum = 0.0;
    for (Text v : values) {
        sum += Double.parseDouble(v.toString());
    }
    String[] split = key.toString().split("\t");
    k.set(split[0].trim());
    v.set(split[1].trim() + ":" + sum);
    context.write(k, v);
}
```

执行命令如下：

```
yarn jar target/BookRecommendation-0.0.1-SNAPSHOT.jar book_rec.step6.MergeTest -Dinput=/BookRec/step5 -Doutput=/BookRec/step6
```

6.3.8 消除用户已收藏书籍

Map1 函数如下：

```
protected void map(LongWritable key, Text value,
        Mapper<LongWritable, Text, KeyTuple, Text>.Context context)
```

```
            throws IOException, InterruptedException {
        String[] split = value.toString().split("\t");
        k.setId(split[0].trim() + "\t" + split[1].trim());
        k.setTag("1");
        v.set(split[6].trim());
        context.write(k, v);
}
```

Map2 函数如下：

```
protected void map(LongWritable key, Text value,
            Mapper<LongWritable, Text, KeyTuple, Text>.Context context)
            throws IOException, InterruptedException {
        String[] split = value.toString().split(":");
        k.setId(split[0].trim());
        k.setTag("1");
        v.set(split[1].trim());
        context.write(k, v);
}
```

Reduce 函数如下：

```
protected void reduce(KeyTuple key, Iterable<Text> values,
            Reducer<KeyTuple, Text, Text, Text>.Context context)
            throws IOException, InterruptedException {
        int count = 0;
        for (Text v : values) {
            count++;
            l.add(v);
        }
        if(count == 1){
            context.write(key.getId(), l.get(0));
```

```
    }
}
```

执行命令如下：

```
yarn jar target/BookRecommendation-0.0.1-SNAPSHOT.jar book_rec.step7.EliminateTest -Dorigin=/user/maoyh/collection_books -Dmatrix=/BookRec/step6 -Doutput=/BookRec/step7
```

6.3.9 推荐列表保存到数据库

Map 函数如下：

```
protected void map(LongWritable key, Text value,
        Mapper<LongWritable, Text, KeyPair, ValueTuple>.Context context)
        throws IOException, InterruptedException {
    String[] split = value.toString().split("\t");
    k.setId(split[0].trim());
    k.setRef(Double.valueOf(split[2].trim()));
    v.setBook_id(split[1].trim());
    v.setPref(Double.valueOf(split[2].trim()));
    context.write(k, v);
}
```

Reduce 函数如下：

```
protected void reduce(
        KeyPair key,
        Iterable<ValueTuple> values,
        Reducer<KeyPair, ValueTuple, book_rec.step8.RecommendationDB, NullWritable>.Context context)
        throws IOException, InterruptedException {
    int count = 0;
```

```
    for (ValueTuple value : values) {
        if (count < 5) {
            k.setId(key.getId());
            k.setBook_id(value.getBook_id());
            k.setPreference(value.getPref());
            context.write(k, NullWritable.get());
            count++;
        }
        else
            break;
    }
}
```

执行函数如下：

```
public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    Path input = new Path(conf.get("input"));

    Job job = Job.getInstance(conf);
    job.setJarByClass(this.getClass());
    job.setJobName("Filtrate & Wirte to DB Test");

    job.setMapperClass(FiltrateMapper.class);
    job.setMapOutputKeyClass(KeyPair.class);
    job.setMapOutputValueClass(ValueTuple.class);

    job.setReducerClass(FiltrateReducer.class);
    job.setOutputKeyClass(RecommendationDB.class);
    job.setOutputValueClass(NullWritable.class);
```

```
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(DBOutputFormat.class);

        TextInputFormat.addInputPath(job, input);

        DBConfiguration.configureDB(job.getConfiguration(), "com.mysql.jdbc.Driver",
                "jdbc:mysql://192.168.56.1:3306/brs", "root", "123456");
        DBOutputFormat.setOutput(job, "recommendation_books", "rec_day", "id", "book_id",
"rec_degree");

        job.setPartitionerClass(KeyPairPartitioner.class);
        job.setSortComparatorClass(KeyPairComparator.class);
        job.setGroupingComparatorClass(KeyPairGroupComparator.class);

        return job.waitForCompletion(true) ? 0 : 1;
}
```

执行命令如下：

```
yarn jar target/BookRecommendation-0.0.1-SNAPSHOT.jar book_rec.step8.FiltrateTest -Dinput=/BookRec/step7 -libjars /opt/maven/repository/mysql/mysql-connector-java/5.1.38/mysql-connector-java-5.1.38.jar
```

第 7 章 系统测试

7.1 基于 HADOOP 的个性化书籍推荐系统的测试方案

软件测试指为了发现软件中的错误而执行软件的过程。它的目标是尽可能多地发现软件中存在的错误，将测试结果作为纠错的依据。软件测试是软件开发过程中的一个重要组成部分，是贯穿整个软件开发生命周期、对软件产品（包括阶段性产品）进行验证和确认的活动过程。

软件测试是一个规则的过程，包括测试设计、测试执行以及测试结果比较等。

1. 测试设计：根据软件开发各阶段的文档资料和程序的内部结构，利用各种设计测试用例技术精心设计测试用例。

2. 测试执行：利用这些测试用例执行程序，得到测试结果。

3. 测试结果比较：将预期的结果与实际测试结果进行比较，如果二者不符合，对于出现的错误进行纠错，并修改相应文档。修改后的程序还要进行再次测试，直到满意为止。如果测试发现不了错误，可能由于测试配置考虑不周到，应考虑重新制定测试方案，设计测试用例。

软件测试技术是多种多样的，从不同角度有多种测试技术：

1. 从是否需要执行被测软件的角度分类：

静态测试——通过对被测程序的静态审查，发现代码中潜在的错误。它一般用人工方式脱机完成，故亦称人工测试或代码评审（Code Review）;也可借助于静态分析器在机器上以自动方式进行检查，但不要求程序本身在机器上运行。

动态测试——使用和运行被测软件，通常意义上的测试。动态测试的对象必须是能够由计算机真正运行的被测试的程序，它包含黑盒测试和白盒测试。

2. 从软件测试用例设计方法的角度分类：

黑盒测试——从用户角度出发的测试，又称为功能测试、数据驱动测试和基于规格说明的测试。把被测试程序当作一个黑盒，忽略程序内部的结构的特性，测试者在只知道该程序输入和输出之间的关系或程序功能的情况下，依靠能够反映这一关系和程序功能需求规格的说明书，来确定测试用例和推断测试结果的正确性。

白盒测试——基于产品的内部结构来进行测试，又称为结构测试，逻辑驱动测试或基于程序的测试。主要检查内部操作是否按规定执行，软件各个部分功能是否得到充分利用，即根据被测程序的内部结构设计测试用例，测试者需要预先了解被测试程序的结构。

3. 从软件测试的策略和过程的角度分类：

单元测试——针对每个单元的测试，它确保每个模块能正常工作，主要采用白盒测试方法，用以发现内部错误。

集成测试——对已测试过的模块进行组装后的测试，主要检验与软件设计相关的程序结构问题。主要采用黑盒测试和白盒测试两种方法，来验证多个单元模块集成到一起后是否能够协调工作。

确认测试——检验所开发的软件能否满足所有功能和性能需求的最后手段，通常采用黑盒测试方法。

系统测试——是检测被测软件与系统的其他部分的协调性，通常采用黑盒测试方法。

验收测试——是软件产品质量的最后一关，主要从用户的角度着手，其参与者主要是用户和少量的程序开发人员，通常采用黑盒测试方法。

7.2 测试内容

对此次开发完成的单点登录系统，测试的主要内容如下：

1. 功能测试中，要测试该系统的个性化推荐功能。当游客访问页面时，要能够看到热门推荐、新书上架、书籍排行等内容。当用户登录时，要能够推送根据用户兴趣爱好推送个性化推荐信息。如果用户没有收藏过书籍，就推送热门书籍信息。

2. 代码测试中，要测试：当输入错误数据进行相关操作时系统的错误提示。如在注册用户时，如果该用户名已存在，系统将提示：“用户名已被占用!”

3. 测试当选择相应的操作时，系统显示的内容是否与当前用户的操作相匹配，例如，当用户点击“关注”按钮时，如果在已关注的好友列表可以查询到关注的好友，那么就说明关注的功能是正常的。

7.3 测试实例的研究与选择

首先，我们要充分掌握软件测试的流程、原则、测试用例分析设计及自动化测试方案设计，分析产品需求，建立测试环境和计划，保证产品质量以及测试工作的顺利进行，按照软

件工程规范和项目管理流程，实施并管理不同阶段的测试，书写测试用例，提交测试报告。做好测试工作的各项准备。

1. 测试方案

掌握好软件测试的方法是提升软件企业工作质量的基础，也是软件工程的一个重要阶段，系统的问题越早发现，改正成本越低，破坏性越小，所以，在系统发布前，要尽量多地把问题找出来，其手段就是有计划、有组织地进行充分的测试。本系统的测试方案主要采用黑盒测试方法，白盒测试方法作为辅助。

在测试过程中，先对系统的各个子模块进行测试，其次对子系统进行测试，并处理好各个接口的问题，最后对系统进行测试和维护。

2. 测试项目

1) 各模块之间的接口测试
2) 系统测试
3) 计算机软硬件兼容性测试
4) 容错性测试
5) 性能与效率测试
6) 易用性测试

3. 测试过程

测试工作开始之前，要认真书写测试文档，详细陈列要测试的内容和测试的标准，设计全面、合理的测试用例，以便更好地对该项目进行测试和维护。要认真阅读该项目的设计说明书，熟悉设计的思想，编程思路，了解系统的流程和各个模块的功能，做到心中有数，其次，要认真阅读测试文档，熟悉测试环境，按照开发阶段划分，软件测试可分为：代码审查、单元测试、集成测试、系统测试和验收测试，按照这一流程，开始测试。

4. 代码审查

代码审查，在项目的早期发现缺陷，将损失降至最低。同时由于还有至少另外一个人看过这部分代码，项目就不会严重依赖于某个人。该流程中，我和同学首先确定了编码规范，努力发现了一些不符合编码规范的代码，并做了调整。对不合理的部分通过协商，最终修改。其次，找了经验丰富的同学对系统的代码进行了点评，他也提出了一些建议，优化了项目结构和代码规范。

5. 单元测试

单元测试是指对软件中的最小可测试单元进行检查和验证，是对项目中的各个模块进行测试。在编码过程中，利用 Junit 对类中重要的方法进行测试执行，对于执行异常的部分，随时根据测试结果做相应的修改，确保每个模块能正常工作。在单元测试阶段，以代码检查法、逻辑覆盖法、基本路径测试法等白盒测试方法为主。

6. 集成测试

集成测试也称为组装测试、联合测试。在单元测试的基础上，根据概要设计的要求对程序模块采用适当的集成测试策略组装起来，对系统的接口以及集成后的功能进行正确校验的测试。

根据软件的模块结构图，按控制层次从高到低的顺序对模块进行集成，也就是从最顶层模块向下逐步集成，并在集成的过程中进行测试，直至组装成符合要求的最终软件系统。这样可在测试早期发现在主要控制方面存在问题，实现并验证系统主要功能。

7. 确认测试

确认测试又称为有效性测试。它的任务是验证软件的功能和性能及其特性是否达到需求规格说明书的要求。在测试规格说明书中，对需求规格说明中的要求做进一步的细化，用于指导确认测试的进行。确认测试一般不由软件开发人员执行，而应由软件企业中独立的测试部门或第三方测试机构来完成。

在模拟的环境下，通过执行黑盒测试，验证被测软件是否满足需求规格说明书中的需求。包括对功能、性能、文档、安全性、健壮性、兼容性等的要求。

8. 系统测试

软件只是计算机系统的一个元素，软件最终要与其他系统元素（如硬件、信息等）相结合。系统测试是将已经集成好的软件系统，作为整个计算机系统的一个元素，与计算机硬件、外设、某些支持软件、数据和人员等其他系统元素结合在一起，在实际运行环境下，对计算机系统进行一系列的组装测试和确认测试。

在系统测试阶段，依次对系统进行了恢复测试（容错测试）、安全测试、压力测试、负载测试、性能测试、容量测试、正确性测试等。

9. 验收测试

验收测试是软件正式交付使用之前的最后一道测试工序，检验最终软件产品与用户预期需求是否一致，决定软件是否可被用户接受。软件开发的最终目的是满足用户的需要，所以验收测试通常更突出客户的作用，是以用户为主进行的测试，同时软件开发人员、SQA 人员也应参加进来。

7.4 测试环境与测试条件

该系统实现的是一个基于 Hadoop 的个性化推荐系统，提供了个性化推荐的功能，开发工作主要分为 Server 端和 Client 端。Server 端提供个性化推荐的计算和推送功能，Client 端即请求服务。

运行环境如下：

1. JDK 的版本是 1.8，Tomcat 服务器的版本是 8.0
2. MySQL 数据库的版本是 5.7
3. Apache Hadoop 的版本是 2.6

Hadoop 实现了一个分布式文件系统（Hadoop Distributed File System），简称 HDFS。HDFS 有高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上；而且它提供高吞吐量（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。HDFS 放宽了（relax）POSIX 的要求，可以以流的形式访问（streaming access）文件系统中的数据。

7.5 测试用例

1. 游客访问页面时，会展示热门推荐、新书上架、榜单、签约作家和热门推荐等信息。

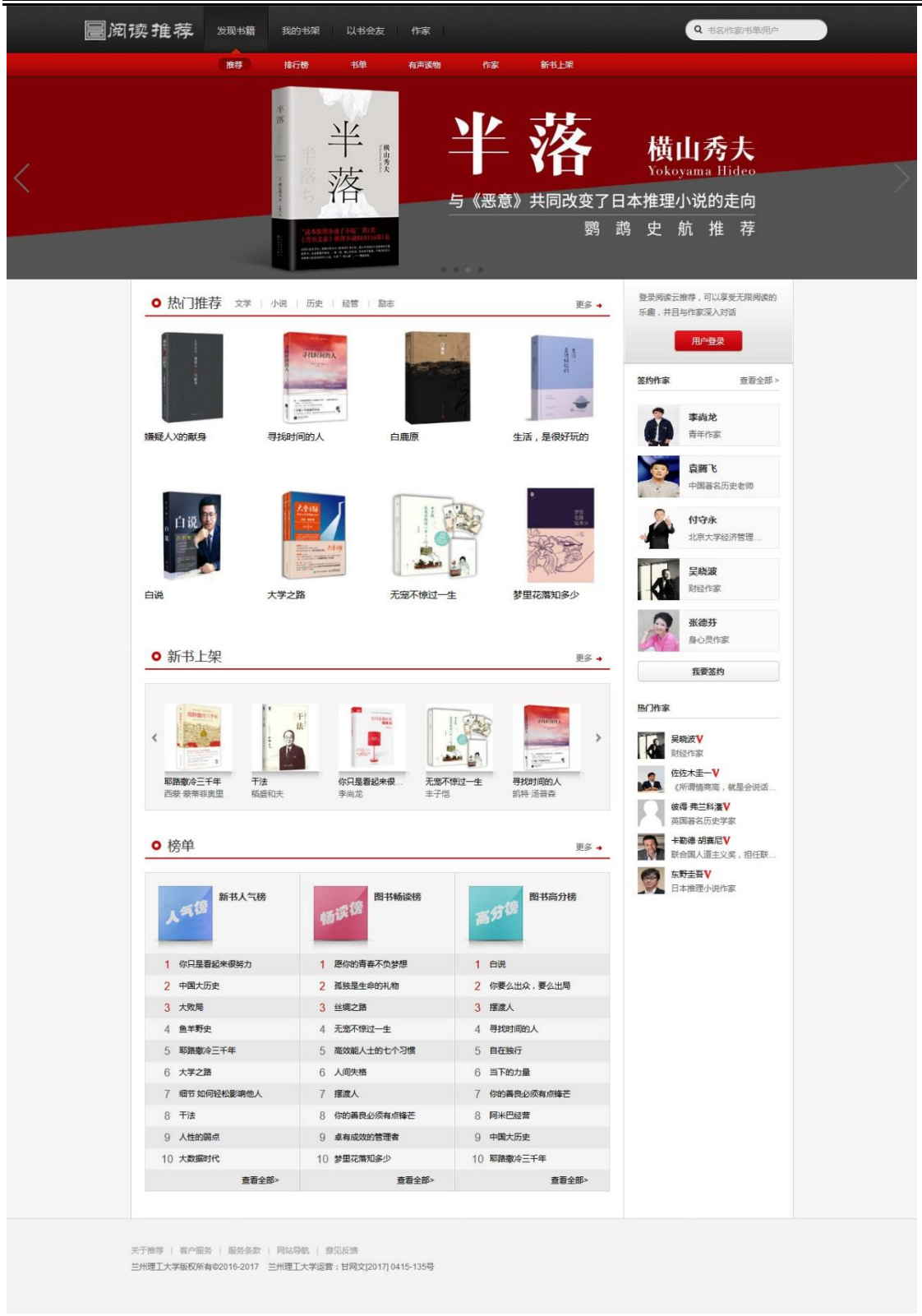


图 7.1 游客访问页面

2. 用户登录测试用例，用户在输入用户名和密码时，如果验证失败应该提示“用户名或密码错误”。

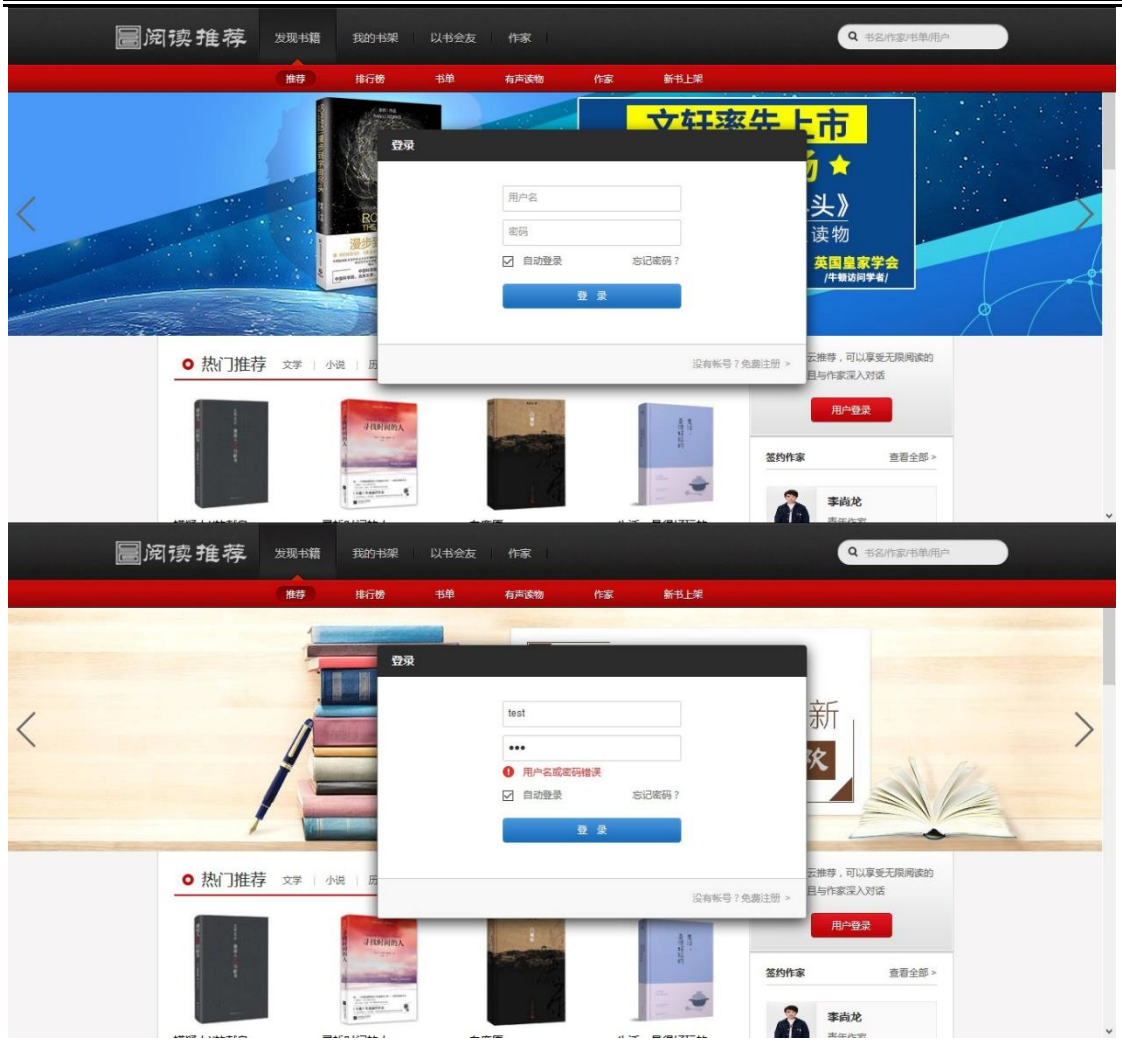


图 7.2 登录测试

3. 用户登录后应该会展示热门推荐、个性化推荐、新书上架、榜单、签约作家和热门推荐等信息。

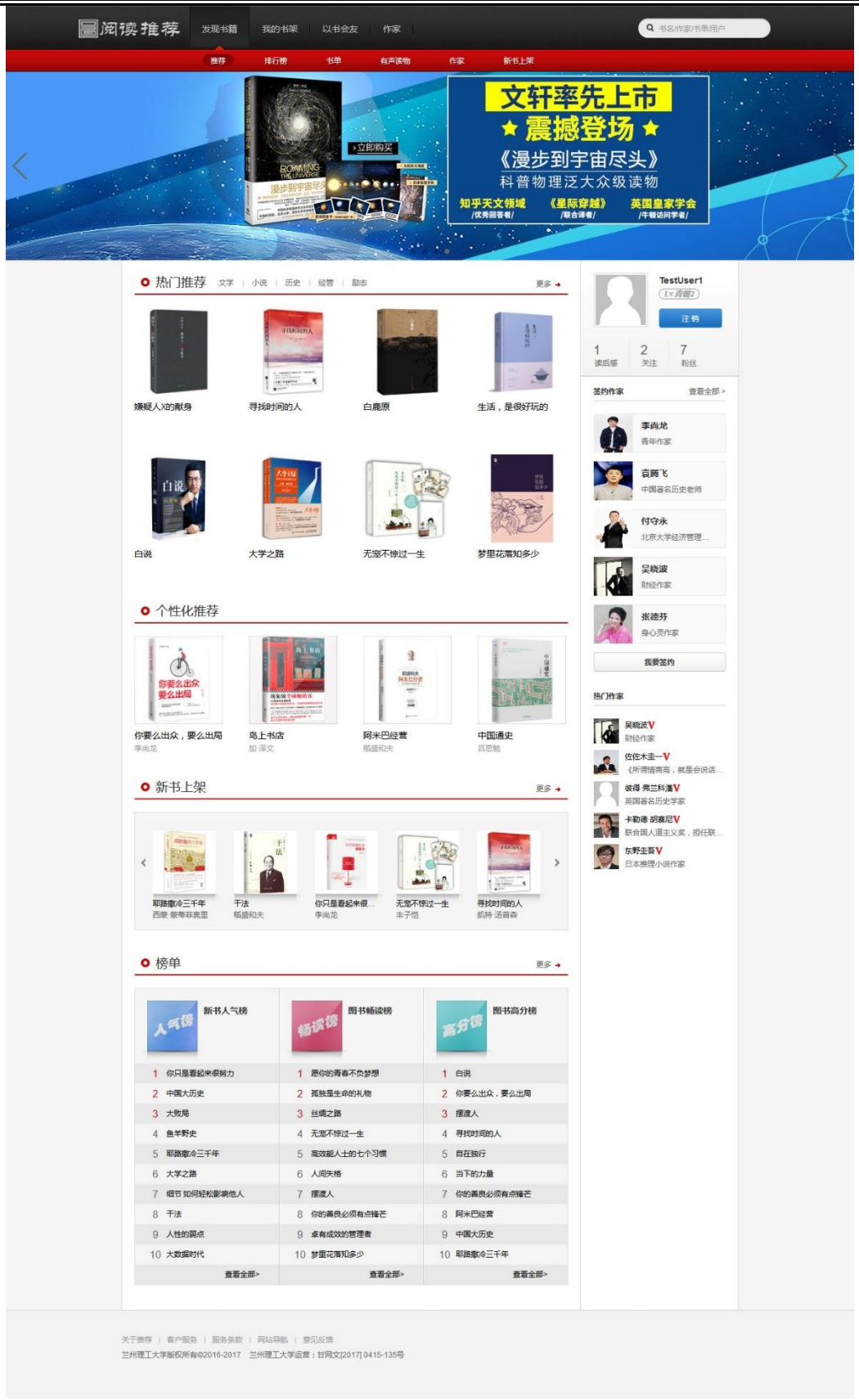

阅读推荐
发现书籍
我的书架
以书会友
作家
书名/作家/书单/用户
推荐
排行榜
书单
有声读物
作家
新书上架
文轩率先上市
震撼登场
《漫步到宇宙尽头》
科普物理泛大众级读物
知乎天文领域
《星际穿越》
英国皇家学会
立即购买
热门推荐
文学
小说
历史
经管
励志
更多
嫌疑人X的献身
寻找时间的人
白鹿原
生活，是很好玩的
白说
大学之路
无宠不惊过一生
梦里花落知多少
个性化推荐
你要么出众，要么出局
李尚龙
岛上书店
加·泽文
阿米巴经营
稻盛和夫
中国通史
吕思勉
新书上架
更多
耶路撒冷三千年
西蒙·蒙蒂菲奥里
干法
稻盛和夫
你只是看起来很...
李尚龙
无宠不惊过一生
毕子侣
寻找时间的人
凯特·汤普森
榜单
更多
新书人气榜
图书畅读榜
图书高分榜
1 你只是看起来很努力
2 中国大历史
3 大败局
4 鱼羊野史
5 耶路撒冷三千年
6 大学之路
7 细节 如何轻松影响他人
8 干法
9 人性的弱点
10 大数据时代
查看全部>
1 愿你的青春不负梦想
2 孤独是生命的礼物
3 丝绸之路
4 无宠不惊过一生
5 高效能人士的七个习惯
6 人间失格
7 摆渡人
8 你的善良必须有点锋芒
9 卓有成效的管理者
10 梦里花落知多少
查看全部>
1 白说
2 你要么出众，要么出局
3 摆渡人
4 寻找时间的人
5 自在独行
6 当下的力量
7 你的善良必须有点锋芒
8 阿米巴经营
9 中国大历史
10 耶路撒冷三千年
查看全部>
TestUser1
注销
1 读后感
2 关注
7 粉丝
签约作家
查看全部 >
李尚龙
青年作家
袁腾飞
中国著名历史老师
付守永
北京大学经济管理...
吴晓波
财经作家
张德芬
身心灵作家
我要签约
热门作家
吴晓波V
财经作家
佐佐木圭一V
《所谓情商高，就是会说话...
彼得·弗兰科潘V
英国著名历史学家
卡勒德·胡赛尼V
联合国人道主义奖，担任联...
东野圭吾V
日本推理小说作家
关于推荐
客户服务
服务条款
网站导航
意见反馈
兰州理工大学版权所有©2016-2017 兰州理工大学运营：甘网文[2017] 0415-135号

图 7.3 用户访问首页

4. 用户登录后可以关注粉丝，关注成功后会显示相互关注。并刷新当前一共关注的人数。

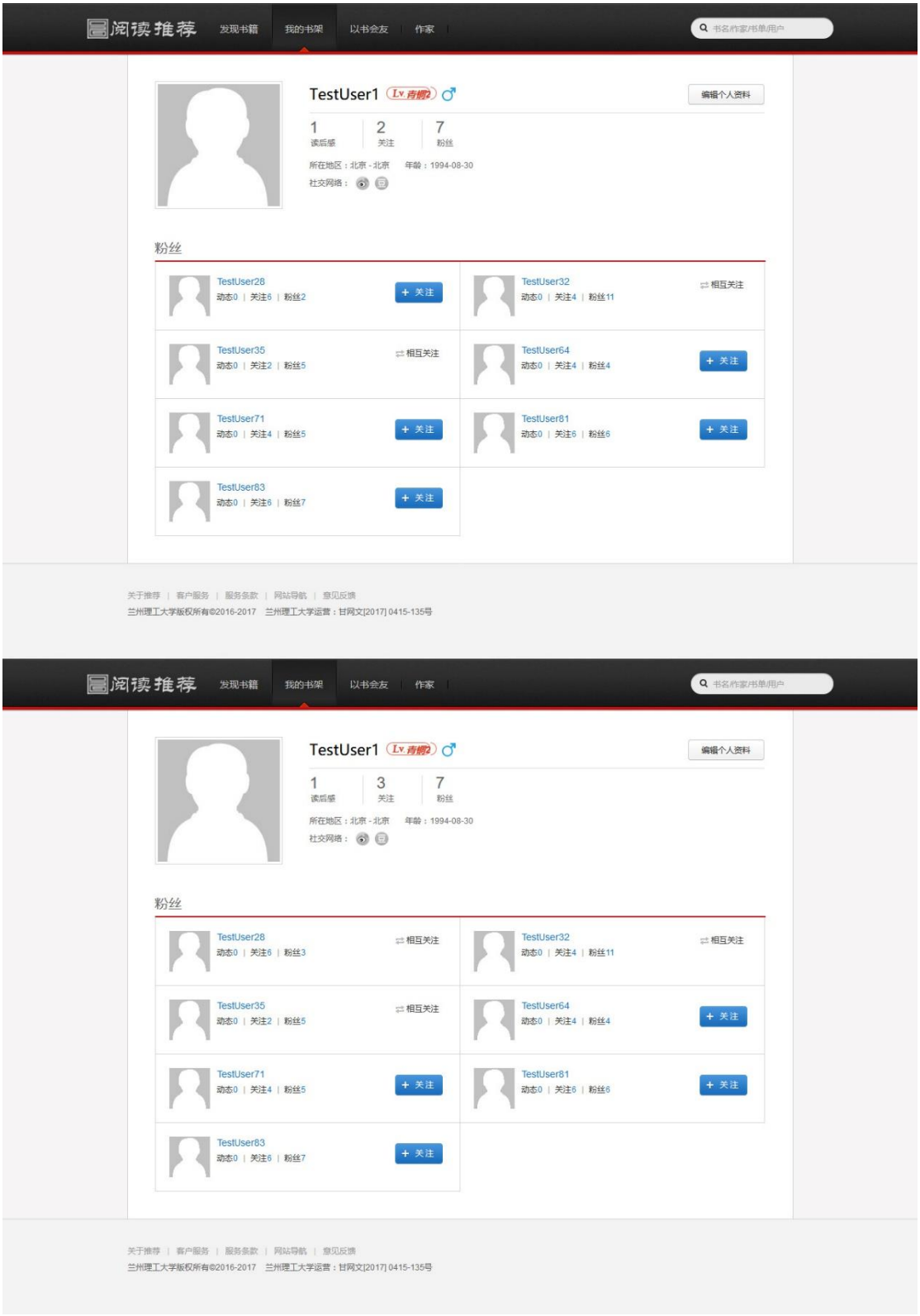


图 7.4 用户关注粉丝

5. 用户登录后，选择一本书籍查看详情，在详情页面点击收藏，即可收藏到我的书架。通过查看我的书架，我们即可看到我们刚才收藏的书籍。

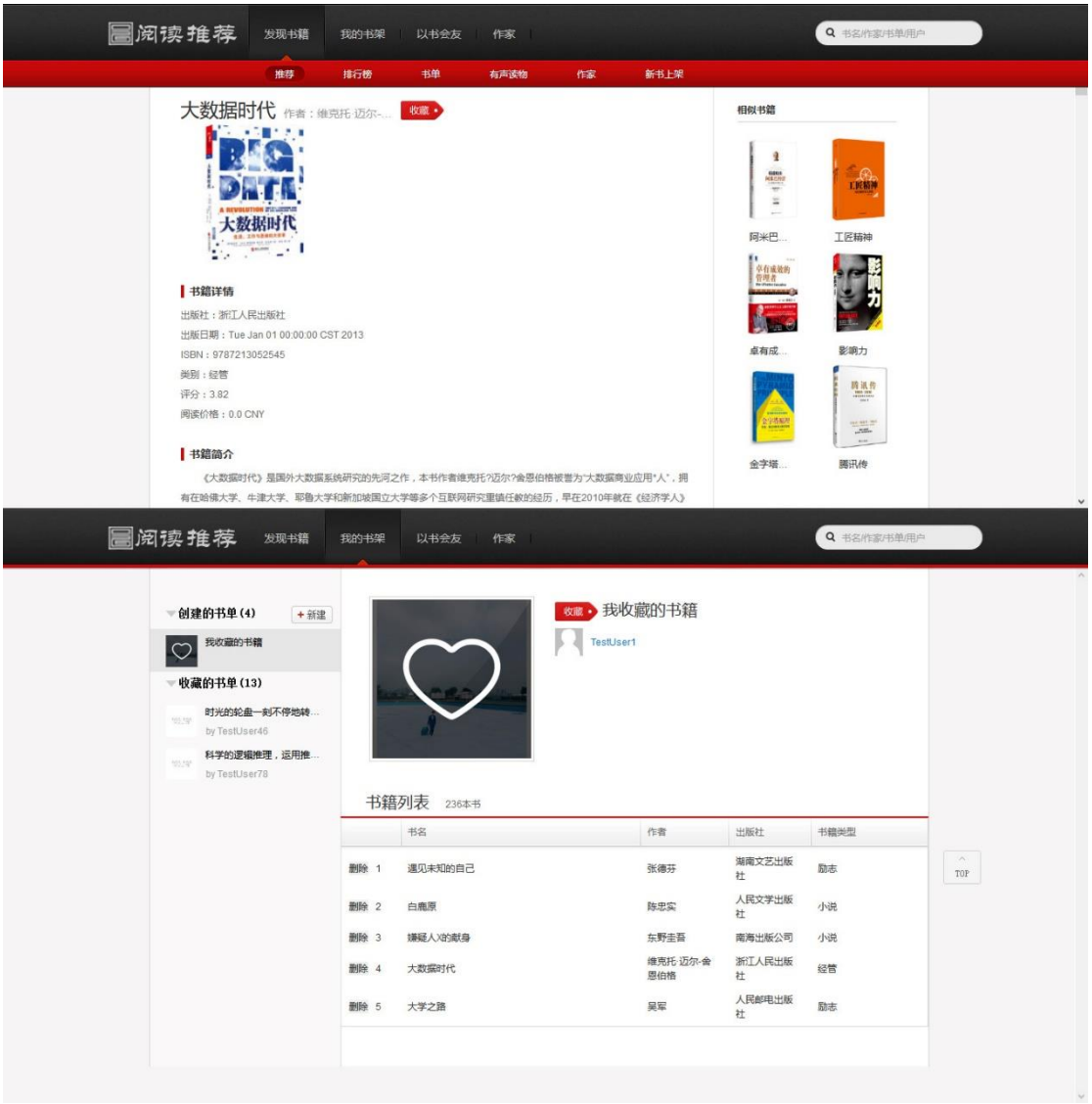


图 7.5 用户收藏书籍

6. 注册功能需要对用户提供的数据进行校验，如果校验不成功则提示用户重新检查输入。

编号	测试字段	校验规则	前提操作	实际输入	预期输出	实际输出
1	account	账号不能为空，不能重复	无		用户名不能为空	用户名不能为空
2				test1	用户名已被占用	用户名已被占用
3				test101	可以使用	可以使用
4	password	密码和确认密码一致，确认密码123qwe	无	123qwe	密码一致	密码一致
5				123456	前后两次密码不一致	前后两次密码不一致
6	name	昵称不能为空	无		昵称不能为空	昵称不能为空
7				test101	昵称可以使用	昵称可以使用
8	gender	性别不能为空	无		性别不能为空	性别不能为空
9				男	性别可以使用	性别可以使用
10				女	性别可以使用	性别可以使用
11	birthday	使用日期组件，不能为空	无		出生日期不能为空	出生日期不能为空
12				1994年8月30日	出生日期可用	出生日期可用
13	province	省份不能为空	无		省份不能为空	省份不能为空
14				浙江	省份可用	省份可用
15	city	城市不能为空	无		城市不能为空	城市不能为空
16				宁波	城市可用	城市可用
17	career	职业不能为空	无		职业不能为空	职业不能为空
18				学生	职业可用	职业可用

图 7.6 注册功能测试用例

第 8 章 Hadoop 配置说明

Hadoop 主节点配置如下：

core-site.xml 配置主节点默认文件系统地址和端口号，在 configuration 标签内添加如下配置。

```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://192.168.56.191:9000</value>
</property>
```

hdfs-site.xml 配置 HDFS 的名称、块复制数量、secondery namenode 地址等，在 configuration 标签内添加如下配置。

```
<property>
    <name>dfs.nameservices</name>
    <value>hadoop-cluster1</value>
</property>
<property>
    <name>dfs.replication</name>
    <value>1</value>
</property>
<property>
    <name>dfs.namenode.secondary.http-address</name>
    <value>192.168.56.195</value>
</property>
<property>
        <name>dfs.namenode.name.dir</name>
        <value>/data/hadoop/hdfs/nn</value>
</property>
<property>
        <name>dfs.namenode.checkpoint.dir</name>
```

```
        <value>/data/hadoop/hdfs/snn</value>
</property>
<property>
        <name>dfs.namenode.checkpoint.edits.dir</name>
        <value>/data/hadoop/hdfs/snn</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>/data/hadoop/hdfs/dn</value>
</property>
```

yarn-site.xml 配置 resourcemanager 的地址，在 configuration 标签内添加如下配置。

```
<property>
    <name>yarn.resourcemanager.hostname</name>
    <value>192.168.56.191</value>
</property>
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>
<property>
    <name>yarn.nodemanager.local-dirs</name>
    <value>/data/hadoop/yarn/nm</value>
</property>
```

mapred-site.xml 配置 mapreduce 程序使用的运行框架

```
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

设计总结

本次毕业设计基于 Apache 的 Hadoop 项目，研发了一套个性化图书推荐系统，建立了个性化的、精准的、高效的获取书籍信息发的途径，并提供了用户管理功能。随着互联网技术的应用普及和电子商务的迅猛发展，充斥在网络中的信息资源数量呈现指数增长的态势。海量的信息同时呈现在用户面前，使得用户感觉无所适从，很难从中找到自己真正感兴趣的资源。本系统就很好的解决了这个问题，大大提升了用户体验度，提高工作效率。本系统采用基于 B/S 模式的三层架构的体系结构，结合 Spring、Spring MVC 和 MyBatis 等技术框架，利用 Cookie、HDFS、MapReduce 等技术，后台数据库服务器采用 MySQL5.7，应用服务器采用 Tomcat8.x，整个项目部署在 Hadoop 集群服务器上。因此，本系统具有如下特点：

1. 提升用户效率。系统采取主动推送模式传送给用户感兴趣的书籍，减少了用户浏览网页过滤信息的时间，提高了工作效率，极大的提升了用户体验。

2. 便于管理书籍 。系统对用户收藏书籍信息进行集中持久化保存，避免了用户数据丢失问题，降低了用户管理书籍的时间和空间成本。

3. 便于数据统计分析。系统建立在一系列真实数据之上的目标用户模型。通过用户调研去了解用户，根据他们的目标、行为和观点的差异，将他们区分为不同的类型，然后每种类型中抽取出典型特征，赋予名字、照片、一些人口统计学要素、场景等描述,就形成了一个人物原型。

4. 提升用户体验。系统不仅布局简约美观，而且呈现的内容丰富，推荐内容简单化、人性化，能够实现完整的交互和体验。

5. 易于扩展。本系统预留了大量弹性数据，可用于日后的数据分析和统计分析，做更加精确的机器学习。并且预留部分接口可用于系统的弹性扩展。

本系统的核心功能：

1) 个性化推荐。
2) 书籍收藏。
3) 好友关注。
4) 榜单排行。
5) 用户管理。

参考文献

[1] 曾子明 著. 信息推荐系统, 北京：科学出版社, 2013.

[2] (葡) Luis Torgo 著. 数据挖掘与 R 语言[M], 北京：机械工业出版社, 2008.

[3] (美) Matthew A. Russell 著. 社交网络的数据挖掘与分析[M]. 北京：机械工业出版社, 2015.

[4] 董启文 著. 基于语言处理技术的蛋白质结构和功能预测若干问题研究[M]. 博士论文,2007.

[5] (美) Tom White 著. 华东师范大学数据科学与工程学院(译). Hadoop 权威指南（第 3 版）[M]. 北京：清华大学出版社，2015.

[6] (美) Edward Capriolo Dean Wampler Jason Rutherglen 著, Hive 编程指南[M], 北京：人民邮电出版社, 2013.

[7] Katbleen Ting & Jarek Jarcec Cecbo 著, Apache Sqoop Cookbook, O'Reilly Media，2013

[8] (挪) Magnus Lie Hetland 著. Python 基础教程（第 2 版・修订版）[M]. 北京：人民邮电出版社, 2014.

[9] 赛奎春. JSP 工程应用与项目实践[M]. 北京：机械工业出版社, 2008.

[10] (美) Sandy Ryza Uri Laserson Sean Owen 著. Spark 高级数据分析[M], 北京：人民邮电出版社, 2015.

附录Ⅰ 英文资料原文

key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundredsof MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with userspecified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of

tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis for a rewrite of our production indexing system. Section 7 discusses related and future work.

Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
  map(String key, String value):
    // key: document name // value: document contents
for each word w in value:
     EmitIntermediate(w, "1");

  reduce(String key, Iterator values):
    // key: a word // values: a list of counts int
result = 0; for each v in values:
    result += ParseInt(v); Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

Types

Even thoughthe previous pseudo-codeis written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map `(k1,v1)` →`list(k2,v2)` reduce `(k2,list(v2))` →`list(v2)`

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to

the user code to convert between strings and appropriate types.

More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs h`URL`,`1`i. The reduce function adds together all values for the same URL and emits a h`URL`,`total count`i pair.

Reverse Web-Link Graph: The map function outputs h`target`,`source`i pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: h`target`,*list*(`source`)i

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of h*word,frequency*i pairs. The map function emits a h`hostname`,`term vector`i pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final h`hostname`,`term vector`i pair.

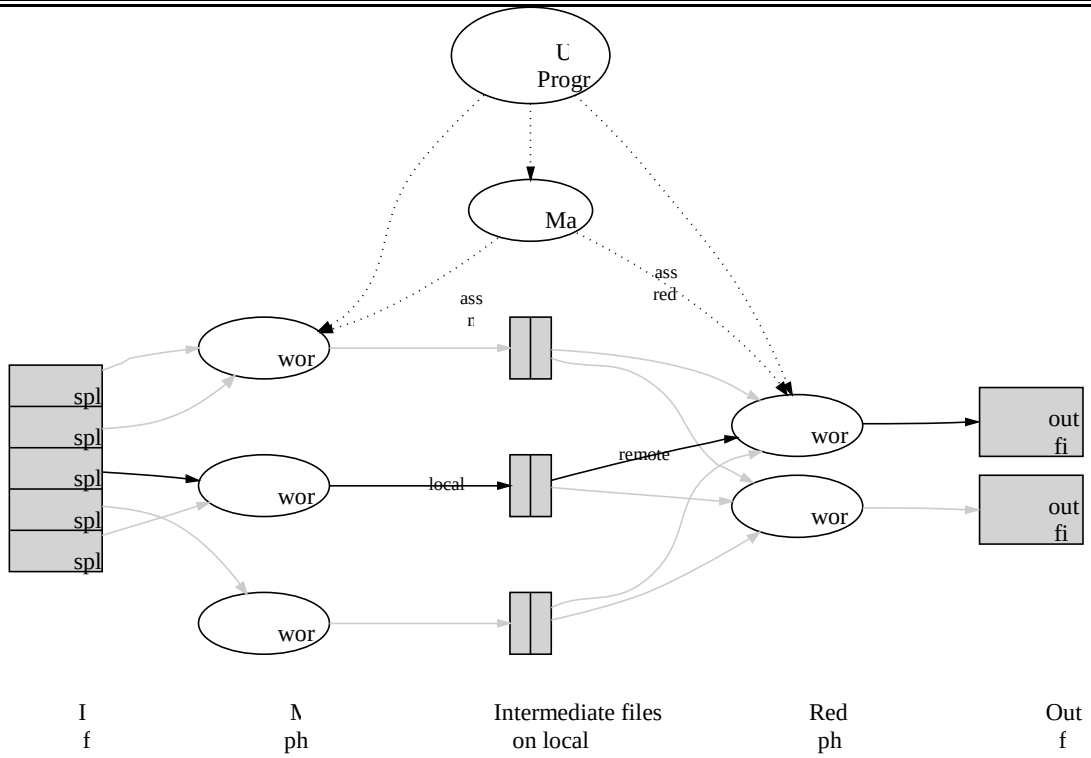


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of h`word`,`document ID`i pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a h`word`,*list*(`document ID`)i pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a h`key`,`record`i pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connectedtogether with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processorx86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developedin-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data into a set of *M splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into *R* pieces using a partitioning function (e.g., *hash*(*key*) **mod** *R*). The number of partitions (*R*) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into *M* pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are *M* map tasks and *R* reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into *R* regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values

to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the *R* output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these *R* output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the *R* intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker *A* and then later executed by worker *B* (because *A* failed), all workers executing reduce tasks are notified of the reexecution. Any reduce task that has not already read the data from worker *A* will read the data from worker *B*.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReducemaster simply re-executed the work done by the unreachable worker machines, and continuedto make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied*map* and *reduce* operatorsare deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces *R* such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the *R* temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of *R* files in a master data structure. When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmersto reason abouttheir program'sbehavior. When the *map* and/or *reduce* operators are nondeterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the non-deterministic program.

Consider map task *M* and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of *M* and $e(R_2)$ may have read the output produced by a different execution of *M*.

Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no

network bandwidth.

Task Granularity

We subdivide the map phase into *M* pieces and the reduce phase into *R* pieces, as described above. Ideally, *M* and *R* should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large *M* and *R* can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, *R* is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose *M* so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make *R* a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200{,}000$ and $R = 5{,}000$, using 2,000 worker machines.

Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (*R*). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "*hash*(*key*) **mod** *R*"). This tends to result in

fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "*hash*(*Hostname*(*urlkey*)) **mod** *R*" as the partitioning function causes all URLs from the same host to end up in the same output file.

Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the userspecified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form <the, 1>. All of these counts will be sent over the network to a single reduce task and then added togetherby the *Reduce*function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performsa map task. Typicallythe same codeis used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text" mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase; uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
    uppercase->Increment(); EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating countervalues, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

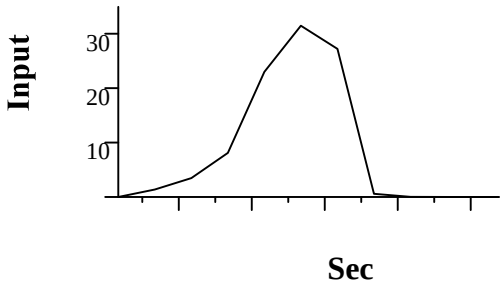
Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of



MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with HyperThreading enabled, 4GB of memory, two 160GB IDE

Figure 2: Data transfer rate over time

disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

Grep

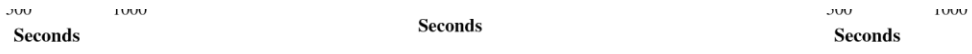
The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This programis modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the original text line as the



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions

passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program). As before, the input data is split into 64MB pieces

($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of *R* pieces.

Our partitioning function for this benchmarkhas builtin knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute splitpoints for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shufflingis doneabout 600seconds into the computation. The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

Machine Failures

In Figure 3 (c), we show an executionof the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler

immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

附录Ⅰ 英文资料译文

摘要

MapReduce 是一个编程模型，也是一个处理和生成超大数据集的算法模型的相关实现。用户首先创建一个 Map 函数处理一个基于 key/value pair 的数据集合，输出中间的基于 key/value pair 的数据集合；然后再创建一个 Reduce 函数用来合并所有的具有相同中间 key 值的中间 value 值。现实世界中有很多满足上述处理模型的例子，本论文将详细描述这个模型。

MapReduce 架构的程序能够在大量的普通配置的计算机上实现并行化处理。这个系统在运行时只关心：

如何分割输入数据，在大量计算机组成的集群上的调度，集群中计算机的错误处理，管理集群中计算机之间必要的通信。采用 MapReduce 架构可以使那些没有并行计算和分布式处理系统开发经验的程序员有效利用分布式系统的丰富资源。

我们的 MapReduce 实现运行在规模可以灵活调整的由普通机器组成的集群上：一个典型的 MapReduce 计算往往由几千台机器组成、处理以 TB 计算的数据。程序员发现这个系统非常好用：已经实现了数以百计的 MapReduce 程序，在 Google 的集群上，每天都有 1000 多个 MapReduce 程序在执行。

1 介绍

在过去的 5 年里，包括本文作者在内的 Google 的很多程序员，为了处理海量的原始数据，已经实现了数以百计的、专用的计算方法。这些计算方法用来处理大量的原始数据，比如，文档抓取（类似网络爬虫的程序）、Web 请求日志等等；也为了计算处理各种类型的衍生数据，比如倒排索引、Web 文档的图结构的各种表示形势、每台主机上网络爬虫抓取的页面数量的汇总、每天被请求的最多的查询的集合等等。大多数这样的数据处理运算在概念上很容易理解。然而由于输入的数据量巨大，因此要想在可接受的时间内完成运算，只有将这些计算分布在成百上千的主机上。如何处理并行计算、如何分发数据、如何处理错误？所有这些问题综合在一起，需要大量的代码处理，因此也使得原本简单的运算变得难以处理。

为了解决上述复杂的问题，我们设计一个新的抽象模型，使用这个抽象模型，我们只要表述我们想要执行的简单运算即可，而不必关心并行计算、容错、数据分布、负载均衡等复杂的细节，这些问题都被封装在了一个库里面。设计这个抽象模型的灵感来自 Lisp 和许多其

他函数式语言的 Map 和 Reduce 的原语。我们意识到我们大多数的运算都包含这样的操作：在输入数据的“逻辑”记录上应用 Map 操作得出一个中间 key/value pair 集合，然后在所有具有相同 key 值的 value 值上应用 Reduce 操作，从而达到合并中间的数据，得到一个想要的结果的目的。使用 MapReduce 模型，再结合用户实现的 Map 和 Reduce 函数，我们就可以非常容易的实现大规模并行化计算；通过 MapReduce 模型自带的“再次执行”（re-execution）功能，也提供了初级的容灾实现方案。

这个工作(实现一个 MapReduce 框架模型)的主要贡献是通过简单的接口来实现自动的并行化和大规模的分布式计算，通过使用 MapReduce 模型接口实现在大量普通的 PC 机上高性能计算。

第二部分描述基本的编程模型和一些使用案例。

第三部分描述了一个经过裁剪的、适合我们的基于集群的计算环境的 MapReduce 实现。

第四部分描述我们认为在 MapReduce 编程模型中一些实用的技巧。第五部分对于各种不同的任务，测量我们 MapReduce 实现的性能。

第六部分揭示了在 Google 内部如何使用 MapReduce 作为基础重写我们的索引系统产品，包括其它一些使用 MapReduce 的经验。

第七部分讨论相关的和未来的工作。

2 编程模型

MapReduce 编程模型的原理是：利用一个输入 key/value pair 集合来产生一个输出的 key/value pair 集合。

MapReduce 库的用户用两个函数表达这个计算：Map 和 Reduce。

用户自定义的 Map 函数接受一个输入的 key/value pair 值，然后产生一个中间 key/value pair 值的集合。

MapReduce 库把所有具有相同中间 key 值 I 的中间 value 值集合在一起后传递给 reduce 函数。

用户自定义的 Reduce 函数接受一个中间 key 的值 I 和相关的一个 value 值的集合。Reduce 函数合并这些 value 值，形成一个较小的 value 值的集合。一般的，每次 Reduce 函数调用只产生 0 或 1 个输出 value 值。通常我们通过一个迭代器把中间 value 值提供给 Reduce 函数，这样我们就可以处理无法全部放入内存中的大量的 value 值的集合。

2.1 例子

例如，计算一个大的文档集合中每个单词出现的次数，下面是伪代码段：

map(String key, String value): // key: document name // value: document contents for each word w in value:

EmitIntermediate(w, “1″);

reduce(String key, Iterator values):

// key: a word

// values: a list of counts int result = 0; for each v in values:

result += ParseInt(v); Emit(AsString(result));

Map 函数输出文档中的每个词、以及这个词的出现次数(在这个简单的例子里就是 1)。Reduce 函数把 Map 函数产生的每一个特定的词的计数累加起来。

另外，用户编写代码，使用输入和输出文件的名字、可选的调节参数来完成一个符合 MapReduce 模型规范的对象，然后调用 MapReduce 函数，并把这个规范对象传递给它。用户的代码和 MapReduce 库链接在一起

(用 C++实现)。附录 A 包含了这个实例的全部程序代码。

2.2 类型

尽管在前面例子的伪代码中使用了以字符串表示的输入输出值，但是在概念上，用户定义的 Map 和 Reduce 函数都有相关联的类型：

map(k1,v1) ->list(k2,v2)

reduce(k2,list(v2)) ->list(v2)

比如，输入的 key 和 value 值与输出的 key 和 value 值在类型上推导的域不同。此外，中间 key 和 value 值与输出 key 和 value 值在类型上推导的域相同。

我们的 C++中使用字符串类型作为用户自定义函数的输入输出，用户在自己的代码中对字符串进行适当的类型转换。

2.3 更多的例子

这里还有一些有趣的简单例子，可以很容易的使用 MapReduce 模型来表示：

分布式的 Grep：Map 函数输出匹配某个模式的一行，Reduce 函数是一个恒等函数，即把中间数据复制到输出。

计算 URL 访问频率：Map 函数处理日志中 web 页面请求的记录，然后输出(URL,1)。Reduce 函数把相同 URL 的 value 值都累加起来，产生(URL,记录总数)结果。

倒转网络链接图：Map 函数在源页面（source）中搜索所有的链接目标（target）并输出为(target,source)。

Reduce 函数把给定链接目标（target）的链接组合成一个列表，输出(target,list(source))。

每个主机的检索词向量：检索词向量用一个(词,频率)列表来概述出现在文档或文档集中的最重要的一些词。Map 函数为每一个输入文档输出(主机名,检索词向量)，其中主机名来自文档的 URL。Reduce 函数接收给定主机的所有文档的检索词向量，并把这些检索词向量加在一起，丢弃掉低频的检索词，输出一个最终的(主机名,检索词向量)。

倒排索引：Map 函数分析每个文档输出一个(词,文档号)的列表，Reduce 函数的输入是一个给定词的所有

（词，文档号），排序所有的文档号，输出(词,list（文档号）)。所有的输出集合形成一个简单的倒排索引，它以一种简单的算法跟踪词在文档中的位置。

分布式排序：Map 函数从每个记录提取 key，输出(key,record)。Reduce 函数不改变任何的值。这个运算

依赖分区机制(在 4.1 描述)和排序属性(在 4.2 描述)。

3 实现

MapReduce 模型可以有多种不同的实现方式。如何正确选择取决于具体的环境。例如，一种实现方式适用于小型的共享内存方式的机器，另外一种实现方式则适用于大型 NUMA 架构的多处理器的主机，而有的实现方式更适合大型的网络连接集群。

本章节描述一个适用于 Google 内部广泛使用的运算环境的实现：用以太网交换机连接、由普通 PC 机组成的大型集群。在我们的环境里包括：

1. x86 架构、运行 Linux 操作系统、双处理器、2-4GB 内存的机器。

2. 普通的网络硬件设备，每个机器的带宽为百兆或者千兆，但是远小于网络的平均带宽的一半。

3. 集群中包含成百上千的机器，因此，机器故障是常态。

4. 存储为廉价的内置 IDE 硬盘。一个内部分布式文件系统用来管理存储在这些磁盘上的数据。文件系统通过数据复制来在不可靠的硬件上保证数据的可靠性和有效性。

5. 用户提交工作（job）给调度系统。每个工作（job）都包含一系列的任务（task），调度系统将这些任务调度到集群中多台可用的机器上。

3.1 执行概括

通过将 Map 调用的输入数据自动分割为 M 个数据片段的集合，Map 调用被分布到多台机器上执行。输入的数据片段能够在不同的机器上并行处理。使用分区函数将 Map 调用产生的中间 key 值分成 R 个不同分区（例如，hash(key) mod R），Reduce 调用也被分布到

多台机器上执行。分区数量（R）和分区函数由用户来指定。

图 1 展示了我们的 MapReduce 实现中操作的全部流程。当用户调用 MapReduce 函数时，将发生下面的一系列动作（下面的序号和图 1 中的序号一一对应）：

1. 用户程序首先调用的 MapReduce 库将输入文件分成 M 个数据片度，每个数据片段的大小一般从

16MB 到 64MB(可以通过可选的参数来控制每个数据片段的大小)。然后用户程序在机群中创建大量的程序副本。

2. 这些程序副本中的有一个特殊的程序－master。副本中其它的程序都是 worker 程序，由 master 分配任务。有 M 个 Map 任务和 R 个 Reduce 任务将被分配，master 将一个 Map 任务或 Reduce 任务分配给一个空闲的 worker。

3. 被分配了 map 任务的 worker 程序读取相关的输入数据片段，从输入的数据片段中解析出 key/value pair，然后把 key/value pair 传递给用户自定义的 Map 函数，由 Map 函数生成并输出的中间 key/value pair，并缓存在内存中。

4. 缓存中的 key/value pair 通过分区函数分成 R 个区域，之后周期性的写入到本地磁盘上。缓存的 key/value pair 在本地磁盘上的存储位置将被回传给 master，由 master 负责把这些存储位置再传送给

Reduce worker。

5. 当 Reduce worker 程序接收到 master 程序发来的数据存储位置信息后，使用 RPC 从 Map worker 所在主机的磁盘上读取这些缓存数据。当 Reduce worker 读取了所有的中间数据后，通过对 key 进行排序后使得具有相同 key 值的数据聚合在一起。由于许多不同的 key 值会映射到相同的 Reduce 任务上，因此必须进行排序。如果中间数据太大无法在内存中完成排序，那么就要在外部进行排序。

6. Reduce worker 程序遍历排序后的中间数据，对于每一个唯一的中间 key 值，Reduce worker 程序将这个 key 值和它相关的中间 value 值的集合传递给用户自定义的 Reduce 函数。Reduce 函数的输出被追加到所属分区的输出文件。

7. 当所有的 Map 和 Reduce 任务都完成之后，master 唤醒用户程序。在这个时候，在用户程序里的对

MapReduce 调用才返回。

在成功完成任务之后，MapReduce 的输出存放在 R 个输出文件中（对应每个 Reduce

任务产生一个输出文件，文件名由用户指定)。一般情况下，用户不需要将这 R 个输出文件合并成一个文件－他们经常把这些文件作为另外一个 MapReduce 的输入，或者在另外一个可以处理多个分割文件的分布式应用中使用。

3.2 Master 数据结构

Master 持有一些数据结构，它存储每一个 Map 和 Reduce 任务的状态（空闲、工作中或完成)，以及 Worker 机器(非空闲任务的机器)的标识。

Master 就像一个数据管道，中间文件存储区域的位置信息通过这个管道从 Map 传递到 Reduce。因此，对于每个已经完成的 Map 任务，master 存储了 Map 任务产生的 R 个中间文件存储区域的大小和位置。当 Map 任务完成时，Master 接收到位置和大小的更新信息，这些信息被逐步递增的推送给那些正在工作的 Reduce 任务。

3.3 容错

因为 MapReduce 库的设计初衷是使用由成百上千的机器组成的集群来处理超大规模的数据，所以，这个库必须要能很好的处理机器故障。

3.3.1 worker 故障

master 周期性的 ping 每个 worker。如果在一个约定的时间范围内没有收到 worker 返回的信息，master 将把这个 worker 标记为失效。所有由这个失效的 worker 完成的 Map 任务被重设为初始的空闲状态，之后这些任务就可以被安排给其他的 worker。同样的，worker 失效时正在运行的 Map 或 Reduce 任务也将被重新置为空闲状态，等待重新调度。

当 worker 故障时，由于已经完成的 Map 任务的输出存储在这台机器上，Map 任务的输出已不可访问了，因此必须重新执行。而已经完成的 Reduce 任务的输出存储在全局文件系统上，因此不需要再次执行。

当一个 Map 任务首先被 worker A 执行，之后由于 worker A 失效了又被调度到 worker B 执行，这个“重新执行”的动作会被通知给所有执行 Reduce 任务的 worker。任何还没有从 worker A 读取数据的 Reduce 任务将从 worker B 读取数据。

MapReduce 可以处理大规模 worker 失效的情况。比如，在一个 MapReduce 操作执行期间，在正在运行的集群上进行网络维护引起 80 台机器在几分钟内不可访问了，MapReduce master 只需要简单的再次执行那些不可访问的 worker 完成的工作，之后继续执行未完成的任务，直到最终完成这个 MapReduce 操作。

3.3.2 master 失败

一个简单的解决办法是让 master 周期性的将上面描述的数据结构(alex 注：指 3.2 节）

的写入磁盘，即检查点（checkpoint）。如果这个 master 任务失效了，可以从最后一个检查点（checkpoint）开始启动另一个 master 进程。然而，由于只有一个 master 进程，master 失效后再恢复是比较麻烦的，因此我们现在的实现是如果 master 失效，就中止 MapReduce 运算。客户可以检查到这个状态，并且可以根据需要重新执行 MapReduce 操作。

3.3.3 在失效方面的处理机制

（alex 注：原文为”semantics in the presence of failures”）

当用户提供的 Map 和 Reduce 操作是输入确定性函数（即相同的输入产生相同的输出）时，我们的分布式实现在任何情况下的输出都和所有程序没有出现任何错误、顺序的执行产生的输出是一样的。

我们依赖对 Map 和 Reduce 任务的输出是原子提交的来完成这个特性。每个工作中的任务把它的输出写到私有的临时文件中。每个 Reduce 任务生成一个这样的文件，而每个 Map 任务则生成 R 个这样的文件（一个 Reduce 任务对应一个文件）。当一个 Map 任务完成的时，worker 发送一个包含 R 个临时文件名的完成消息给 master。如果 master 从一个已经完成的 Map 任务再次接收到到一个完成消息，master 将忽略这个消息；否则，master 将这 R 个文件的名字记录在数据结构里。

当 Reduce 任务完成时，Reduce worker 进程以原子的方式把临时文件重命名为最终的输出文件。如果同一个 Reduce 任务在多台机器上执行，针对同一个最终的输出文件将有多个重命名操作执行。我们依赖底层文件系统提供的重命名操作的原子性来保证最终的文件系统状态仅仅包含一个 Reduce 任务产生的数据。

使用 MapReduce 模型的程序员可以很容易的理解他们程序的行为，因为我们绝大多数的 Map 和 Reduce 操作是确定性的，而且存在这样的一个事实：我们的失效处理机制等价于一个顺序的执行的操作。当 Map 或 /和 Reduce 操作是不确定性的时候，我们提供虽然较弱但是依然合理的处理机制。当使用非确定操作的时候，一个 Reduce 任务 R1 的输出等价于一个非确定性程序顺序执行产生时的输出。但是，另一个 Reduce 任务 R2 的输出也许符合一个不同的非确定顺序程序执行产生的 R2 的输出。

考虑 Map 任务 M 和 Reduce 任务 R1、R2 的情况。我们设定 e(Ri)是 Ri 已经提交的执行过程（有且仅有一个这样的执行过程）。当 e(R1)读取了由 M 一次执行产生的输出，而 e(R2)读取了由 M 的另一次执行产生的输出，导致了较弱的失效处理。

3.4 存储位置

在我们的计算运行环境中，网络带宽是一个相当匮乏的资源。我们通过尽量把输入数据

(由 GFS 管理)存储在集群中机器的本地磁盘上来节省网络带宽。GFS 把每个文件按 64MB 一个 Block 分隔，每个 Block 保存在多台机器上，环境中就存放了多份拷贝(一般是 3 个拷贝)。MapReduce 的 master 在调度 Map 任务时会考虑输入文件的位置信息，尽量将一个 Map 任务调度在包含相关输入数据拷贝的机器上执行；如果上述努力失败了，master 将尝试在保存有输入数据拷贝的机器附近的机器上执行 Map 任务(例如，分配到一个和包含输入数据的机器在一个 switch 里的 worker 机器上执行)。当在一个足够大的 cluster 集群上运行大型 MapReduce 操作的时候，大部分的输入数据都能从本地机器读取，因此消耗非常少的网络带宽。

3.5 任务粒度

如前所述，我们把 Map 拆分成了 M 个片段、把 Reduce 拆分成 R 个片段执行。理想情况下，M 和 R 应当比集群中 worker 的机器数量要多得多。在每台 worker 机器都执行大量的不同任务能够提高集群的动态的负载均衡能力，并且能够加快故障恢复的速度：失效机器上执行的大量 Map 任务都可以分布到所有其他的 worker 机器上去执行。

但是实际上，在我们的具体实现中对 M 和 R 的取值都有一定的客观限制，因为 master 必须执行 O(M+R) 次调度，并且在内存中保存 O(M*R)个状态（对影响内存使用的因素还是比较小的：O(M*R)块状态，大概每对 Map 任务/Reduce 任务 1 个字节就可以了）。

更进一步，R 值通常是由用户指定的，因为每个 Reduce 任务最终都会生成一个独立的输出文件。实际使用时我们也倾向于选择合适的 M 值，以使得每一个独立任务都是处理大约 16M 到 64M 的输入数据（这样，上面描写的输入数据本地存储优化策略才最有效），另外，我们把 R 值设置为我们想使用的 worker 机器数量

的小的倍数。我们通常会用这样的比例来执行 MapReduce：M=200000，R=5000，使用 2000 台 worker 机器。

3.6 备用任务

影响一个 MapReduce 的总执行时间最通常的因素是“落伍者”：在运算过程中，如果有一台机器花了很长的时间才完成最后几个 Map 或 Reduce 任务，导致 MapReduce 操作总的执行时间超过预期。出现“落伍者” 的原因非常多。比如：如果一个机器的硬盘出了问题，在读取的时候要经常的进行读取纠错操作，导致读取数据的速度从 30M/s 降低到 1M/s。如果 cluster 的调度系统在这台机器上又调度了其他的任务，由于 CPU、内存、本地硬盘和网络带宽等竞争因素的存在，导致执行 MapReduce 代码的执行效率更加缓慢。我们最近遇到的一个问题是由于机器的初始化代码有 bug，导致关闭了的处理器的缓存：在这些机器上执

行任务的性能和正常情况相差上百倍。

我们有一个通用的机制来减少“落伍者”出现的情况。当一个 MapReduce 操作接近完成的时候，master 调度备用（backup）任务进程来执行剩下的、处于处理中状态（in-progress）的任务。无论是最初的执行进程、还是备用（backup）任务进程完成了任务，我们都把这个任务标记成为已经完成。我们调优了这个机制，通常只会占用比正常操作多几个百分点的计算资源。我们发现采用这样的机制对于减少超大 MapReduce 操作的总处理时间效果显著。例如，在 5.3 节描述的排序任务，在关闭掉备用任务的情况下要多花 44%的时间完成排序任务。

4 技巧

虽然简单的 Map 和 Reduce 函数提供的基本功能已经能够满足大部分的计算需要，我们还是发掘出了一些有价值的扩展功能。本节将描述这些扩展功能。

4.1 分区函数

MapReduce 的使用者通常会指定 Reduce 任务和 Reduce 任务输出文件的数量（R）。我们在中间 key 上使

用分区函数来对数据进行分区，之后再输入到后续任务执行进程。一个缺省的分区函数是使用 hash 方法(比如， hash(key) mod R)进行分区。hash 方法能产生非常平衡的分区。然而，有的时候，其它的一些分区函数对 key 值进行的分区将非常有用。比如，输出的 key 值是 URLs，我们希望每个主机的所有条目保持在同一个输出文件中。为了支持类似的情况，MapReduce 库的用户需要提供专门的分区函数。例如，使用“hash(Hostname(urlkey)) mod R”作为分区函数就可以把所有来自同一个主机的 URLs 保存在同一个输出文件中。

4.2 顺序保证

我们确保在给定的分区中，中间 key/value pair 数据的处理顺序是按照 key 值增量顺序处理的。这样的顺序保证对每个分成生成一个有序的输出文件，这对于需要对输出文件按 key 值随机存取的应用非常有意义，对在排序输出的数据集也很有帮助。

4.3 Combiner 函数

在某些情况下，Map 函数产生的中间 key 值的重复数据会占很大的比重，并且，用户自定义的 Reduce 函数满足结合律和交换律。在 2.1 节的词数统计程序是个很好的例子。由于词频率倾向于一个 zipf 分布(齐夫分

布)，每个 Map 任务将产生成千上万个这样的记录<the,1>。所有的这些记录将通过网络被发送到一个单独的

Reduce 任务，然后由这个 Reduce 任务把所有这些记录累加起来产生一个数字。我们允

许用户指定一个可选的 combiner 函数，combiner 函数首先在本地将这些记录进行一次合并，然后将合并的结果再通过网络发送出去。

Combiner 函数在每台执行 Map 任务的机器上都会被执行一次。一般情况下，Combiner 和 Reduce 函数是一样的。Combiner 函数和 Reduce 函数之间唯一的区别是 MapReduce 库怎样控制函数的输出。Reduce 函数的输出被保存在最终的输出文件里，而 Combiner 函数的输出被写到中间文件里，然后被发送给 Reduce 任务。

部分的合并中间结果可以显著的提高一些 MapReduce 操作的速度。附录 A 包含一个使用 combiner 函数的例子。

4.4 输入和输出的类型

MapReduce 库支持几种不同的格式的输入数据。比如，文本模式的输入数据的每一行被视为是一个 key/value pair。key 是文件的偏移量，value 是那一行的内容。另外一种常见的格式是以 key 进行排序来存储的 key/value pair 的序列。每种输入类型的实现都必须能够把输入数据分割成数据片段，该数据片段能够由单独的 Map 任务来进行后续处理(例如，文本模式的范围分割必须确保仅仅在每行的边界进行范围分割)。虽然大多数 MapReduce 的使用者仅仅使用很少的预定义输入类型就满足要求了，但是使用者依然可以通过提供一个简单的 Reader 接口实现就能够支持一个新的输入类型。

Reader 并非一定要从文件中读取数据，比如，我们可以很容易的实现一个从数据库里读记录的 Reader，或者从内存中的数据结构读取数据的 Reader。

类似的，我们提供了一些预定义的输出数据的类型，通过这些预定义类型能够产生不同格式的数据。用户采用类似添加新的输入数据类型的方式增加新的输出类型。

4.5 副作用

在某些情况下，MapReduce 的使用者发现，如果在 Map 和/或 Reduce 操作过程中增加辅助的输出文件会比较省事。我们依靠程序 writer 把这种“副作用”变成原子的和幂等的 。通常应用程序首先把输出结果写到一个临时文件中，在输出全部数据之后，在使用系统级的原子操作 rename 重新命名这个临时文件。

如果一个任务产生了多个输出文件，我们没有提供类似两阶段提交的原子操作支持这种情况。因此，对于会产生多个输出文件、并且对于跨文件有一致性要求的任务，都必须是确定性的任务。但是在实际应用过程中，这个限制还没有给我们带来过麻烦。

4.6 跳过损坏的记录

有时候，用户程序中的 bug 导致 Map 或者 Reduce 函数在处理某些记录的时候 crash

掉，MapReduce 操作无法顺利完成。惯常的做法是修复 bug 后再次执行 MapReduce 操作，但是，有时候找出这些 bug 并修复它们不是一件容易的事情；这些 bug 也许是在第三方库里边，而我们手头没有这些库的源代码。而且在很多时候，忽略一些有问题的记录也是可以接受的，比如在一个巨大的数据集上进行统计分析的时候。我们提供了一种执行模式，在这种模式下，为了保证保证整个处理能继续进行，MapReduce 会检测哪些记录导致确定性的 crash，并且跳过这些记录不处理。

每个 worker 进程都设置了信号处理函数捕获内存段异常（segmentation violation）和总线错误（bus error）。在执行 Map 或者 Reduce 操作之前，MapReduce 库通过全局变量保存记录序号。如果用户程序触发了一个系统信号，消息处理函数将用“最后一口气”通过 UDP 包向 master 发送处理的最后一条记录的序号。当 master 看到在处理某条特定记录不止失败一次时，master 就标志着条记录需要被跳过，并且在下次重新执行相关的

Map 或者 Reduce 任务的时候跳过这条记录。

4.7 本地执行

调试 Map 和 Reduce 函数的 bug 是非常困难的，因为实际执行操作时不但是分布在系统中执行的，而且通常是在好几千台计算机上执行，具体的执行位置是由 master 进行动态调度的，这又大大增加了调试的难度。为了简化调试、profile 和小规模测试，我们开发了一套 MapReduce 库的本地实现版本，通过使用本地版本的 MapReduce 库，MapReduce 操作在本地计算机上顺序的执行。用户可以控制 MapReduce 操作的执行，可以把操作限制到特定的 Map 任务上。用户通过设定特别的标志来在本地执行他们的程序，之后就可以很容易的使用本地调试和测试工具（比如 gdb）。

4.8 状态信息

master 使用嵌入式的 HTTP 服务器（如 Jetty）显示一组状态信息页面，用户可以监控各种执行状态。状态信息页面显示了包括计算执行的进度，比如已经完成了多少任务、有多少任务正在处理、输入的字节数、中间数据的字节数、输出的字节数、处理百分比等等。页面还包含了指向每个任务的 stderr 和 stdout 文件的链接。用户根据这些数据预测计算需要执行大约多长时间、是否需要增加额外的计算资源。这些页面也可以用来分析什么时候计算执行的比预期的要慢。

另外，处于最顶层的状态页面显示了哪些 worker 失效了，以及他们失效的时候正在运行的 Map 和 Reduce

任务。这些信息对于调试用户代码中的 bug 很有帮助。

4.9 计数器

MapReduce 库使用计数器统计不同事件发生次数。比如，用户可能想统计已经处理了多少个单词、已经索引的多少篇 German 文档等等。

为了使用这个特性，用户在程序中创建一个命名的计数器对象，在 Map 和 Reduce 函数中相应的增加计数器的值。例如：

```
Counter* uppercase;
uppercase = GetCounter("uppercase"); map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

这些计数器的值周期性的从各个单独的 worker 机器上传递给 master（附加在 ping 的应答包中传递）。master 把执行成功的 Map 和 Reduce 任务的计数器值进行累计，当 MapReduce 操作完成之后，返回给用户代码。

计数器当前的值也会显示在 master 的状态页面上，这样用户就可以看到当前计算的进度。当累加计数器的值的时候，master 要检查重复运行的 Map 或者 Reduce 任务，避免重复累加（之前提到的备用任务和失效后重新执行任务这两种情况会导致相同的任务被多次执行）。

有些计数器的值是由 MapReduce 库自动维持的，比如已经处理的输入的 key/value pair 的数量、输出的 key/value pair 的数量等等。

计数器机制对于 MapReduce 操作的完整性检查非常有用。比如，在某些 MapReduce 操作中，用户需要确保输出的 key value pair 精确的等于输入的 key value pair，或者处理的 German 文档数量在处理的整个文档数量中属于合理范围。

5 性能

本节我们用在一个大型集群上运行的两个计算来衡量 MapReduce 的性能。一个计算在大约 1TB 的数据中进行特定的模式匹配，另一个计算对大约 1TB 的数据进行排序。

这两个程序在大量的使用 MapReduce 的实际应用中是非常典型的 — 一类是对数据格式进行转换，从一

种表现形式转换为另外一种表现形式；另一类是从海量数据中抽取少部分的用户感兴趣的数据。

5.1 集群配置

所有这些程序都运行在一个大约由 1800 台机器构成的集群上。每台机器配置 2 个 2G 主频、支持超线程的 Intel Xeon 处理器，4GB 的物理内存，两个 160GB 的 IDE 硬盘和一个千兆以太网卡。这些机器部署在一个两层的树形交换网络中，在 root 节点大概有 100-200GBPS 的传输带宽。所有这些机器都采用相同的部署（对等部署），因此任意两点之间的网络来回时间小于 1 毫秒。

在 4GB 内存里，大概有 1-1.5G 用于运行在集群上的其他任务。测试程序在周末下午开始执行，这时主机的 CPU、磁盘和网络基本上处于空闲状态。

5.2 GREP

这个分布式的 grep 程序需要扫描大概 10 的 10 次方个由 100 个字节组成的记录，查找出现概率较小的 3 个字符的模式（这个模式在 92337 个记录中出现）。输入数据被拆分成大约 64M 的 Block（M=15000），整个输出数据存放在一个文件中（R=1）。

图 2 显示了这个运算随时间的处理过程。其中 Y 轴表示输入数据的处理速度。处理速度随着参与 MapReduce 计算的机器数量的增加而增加，当 1764 台 worker 参与计算的时，处理速度达到了 30GB/s。当 Map 任务结束的时候，即在计算开始后 80 秒，输入的处理速度降到 0。整个计算过程从开始到结束一共花了大概 150 秒。这包括了大约一分钟的初始启动阶段。初始启动阶段消耗的时间包括了是把这个程序传送到各个 worker 机器上的时间、等待 GFS 文件系统打开 1000 个输入文件集合的时间、获取相关的文件本地位置优化信息的时间。

5.3 排序

排序程序处理 10 的 10 次方个 100 个字节组成的记录（大概 1TB 的数据）。这个程序模仿 TeraSort benchmark[10]。

排序程序由不到 50 行代码组成。只有三行的 Map 函数从文本行中解析出 10 个字节的 key 值作为排序的 key，并且把这个 key 和原始文本行作为中间的 key/value pair 值输出。我们使用了一个内置的恒等函数作为 Reduce 操作函数。这个函数把中间的 key/value pair 值不作任何改变输出。最终排序结果输出到两路复制的

GFS 文件系统（也就是说，程序输出 2TB 的数据）。

如前所述，输入数据被分成 64MB 的 Block（M=15000）。我们把排序后的输出结果分区后存储到 4000 个文件（R=4000）。分区函数使用 key 的原始字节来把数据分区到 R 个片

段中。

在这个 benchmark 测试中，我们使用的分区函数知道 key 的分区情况。通常对于排序程序来说，我们会增加一个预处理的 MapReduce 操作用于采样 key 值的分布情况，通过采样的数据来计算对最终排序处理的分区点。

图三(a)显示了这个排序程序的正常执行过程。左上的图显示了输入数据读取的速度。数据读取速度峰值会达到 13GB/s，并且所有 Map 任务完成之后，即大约 200 秒之后迅速滑落到 0。值得注意的是，排序程序输入数据读取速度小于分布式 grep 程序。这是因为排序程序的 Map 任务花了大约一半的处理时间和 I/O 带宽把中间输出结果写到本地硬盘。相应的分布式 grep 程序的中间结果输出几乎可以忽略不计。

左边中间的图显示了中间数据从 Map 任务发送到 Reduce 任务的网络速度。这个过程从第一个 Map 任务完成之后就开始缓慢启动了。图示的第一个高峰是启动了第一批大概 1700 个 Reduce 任务(整个 MapReduce 分布到大概 1700 台机器上，每台机器 1 次最多执行 1 个 Reduce 任务)。排序程序运行大约 300 秒后，第一批启动的 Reduce 任务有些完成了，我们开始执行剩下的 Reduce 任务。所有的处理在大约 600 秒后结束。 左下图表示 Reduce 任务把排序后的数据写到最终的输出文件的速度。在第一个排序阶段结束和数据开始写入磁盘之间有一个小的延时，这是因为 worker 机器正在忙于排序中间数据。磁盘写入速度在 2-4GB/s 持续一段时间。输出数据写入磁盘大约持续 850 秒。计入初始启动部分的时间，整个运算消耗了 891 秒。这个速度和 TeraSort benchmark[18]的最高纪录 1057 秒相差不多。

还有一些值得注意的现象：输入数据的读取速度比排序速度和输出数据写入磁盘速度要高不少，这是因

为我们的输入数据本地化优化策略起了作用 — 绝大部分数据都是从本地硬盘读取的，从而节省了网络带宽。排序速度比输出数据写入到磁盘的速度快，这是因为输出数据写了两份（我们使用了 2 路的 GFS 文件系统，写入复制节点的原因是为了保证数据可靠性和可用性)。我们把输出数据写入到两个复制节点的原因是因为这是底层文件系统的保证数据可靠性和可用性的实现机制。如果底层文件系统使用类似容错编码[14](erasure coding)的方式而不是复制的方式保证数据的可靠性和可用性，那么在输出数据写入磁盘的时候，就可以降低网络带宽的使用。

5.4 高效的 backup 任务

图三（b）显示了关闭了备用任务后排序程序执行情况。执行的过程和图 3（a）很相似，

除了输出数据写磁盘的动作在时间上拖了一个很长的尾巴，而且在这段时间里，几乎没有什么写入动作。在 960 秒后，只有 5 个 Reduce 任务没有完成。这些拖后腿的任务又执行了 300 秒才完成。整个计算消耗了 1283 秒，多了 44% 的执行时间。

5.5 失效的机器

在图三（c）中演示的排序程序执行的过程中，我们在程序开始后几分钟有意的 kill 了 1746 个 worker 中的 200 个。集群底层的调度立刻在这些机器上重新开始新的 worker 处理进程（因为只是 worker 机器上的处理进程被 kill 了，机器本身还在工作）。

图三（c）显示出了一个“负”的输入数据读取速度，这是因为一些已经完成的 Map 任务丢失了（由于相应的执行 Map 任务的 worker 进程被 kill 了），需要重新执行这些任务。相关 Map 任务很快就被重新执行了。

整个运算在 933 秒内完成，包括了初始启动时间。

致谢

本人的毕业设计是在我的导师张秋余老师和崔略老师的亲切关怀和悉心指导下完成的。他严肃的科学态度，严谨的治学精神，精益求精的工作作风，深深地感染和激励着我。从毕业设计题目的选择到项目的最终完成，张老师和崔老师都始终给予我悉心的指导和不懈的支持。在此谨向温老师致以诚挚的谢意和崇高的敬意。

四年大学生活即将结束，回顾几年的历程，我还要感谢在一起愉快的度过大学生活的每个可爱的同学们和尊敬的老师们，正是由于你们的帮助和支持，我才能克服一个一个的困难和疑惑，直至本文的顺利完成。老师们严谨的治学，优良的作风和敬业的态度，为我们树立了为人师表的典范。在此，谨向老师们致以衷心的感谢和崇高的敬意。