大型集群上的快速和通用数据处理架构

An Architecture for Fast and General Data Processing on Large Clusters

Matei Zaharia 著

CSDN CODE 翻译社区 译

加州大学伯克利分校电气工程和计算机科学系技术报告
编号：UCB/EECS-2014-12
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html

CSDN CODE 翻译社区项目地址：http://code.csdn.net/translations/15

版权声明

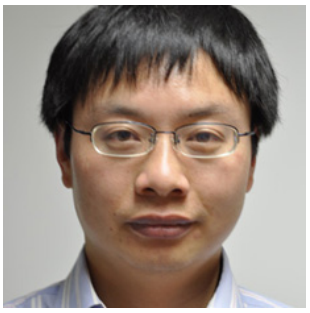
译者名录

本论文翻译由 CSDN CODE 翻译平台（http://code.csdn.net/translations）组织，网友自愿报名参与。共有 35 名译者，7 名审校先后报名参与本论文的翻译工作。最终有 29 名译者、6 名审校完整跟进并完成翻译工作。在此，我们对这些译者、审校以及项目经理吴小然表示诚挚的谢意。

感谢 CSDN CODE 翻译平台及北京语智云帆科技有限公司提供翻译平台和技术支持。

以下列出了完整跟进此项目至完成的译者、审校和项目经理名单。

项目经理：

	CSDN ID: xiaoran27 **昵称/姓名：**吴小然 **个人简介：**美一天进步一点点，尽人事，听天命。

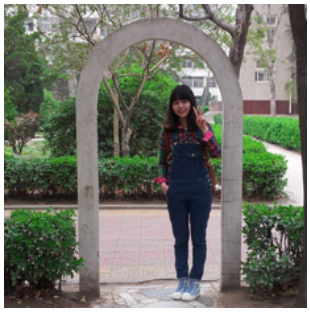
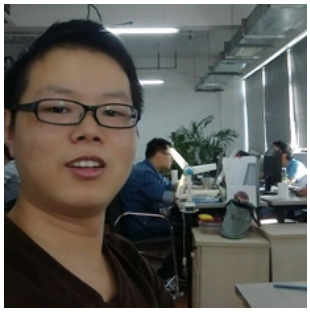
主审校：

	CSDN ID: aiuyjerry **昵称/姓名：**邵赛赛 **个人简介：**邵赛赛，开发工程师，专注于大数据领域，开源爱好者，现从事 Spark 相关工作，Spark 代码贡献者。
	CSDN ID: liyezhang556520 **昵称/姓名：**张李晔 **个人简介：**英特尔大数据研发工程师，apache spark contributor

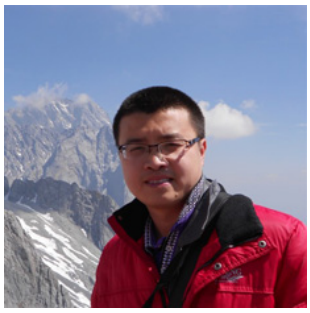
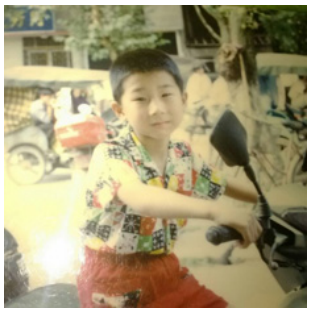
审校：

	CSDN ID: u011278817 **昵称/姓名：**余根茂 **个人简介：**心若没有栖息的地方，到哪里都是在流浪。
	CSDN ID: u012969795 **昵称/姓名：**Ali **个人简介：**很高兴能和大家一起走过来，谢谢。要有到深圳来玩的，吱个声，聚聚~
	CSDN ID: lance_123 **昵称/姓名：**王联辉 **个人简介：**Hadoop/Hive/Spark Contributor，2009 年开始从事 Hadoop 相关的工作，经历了 Hadoop 千台规模的扩张及解决方案。对 Hadoop,Hive,HBase,Yarn,Storm,Spark 等项目有丰富的实践经验且熟悉其核心代码，热衷于大数据开源项目与技术。
	CSDN ID: derek12344321 **昵称/姓名：**马继 **个人简介：**大家好，我叫马继，目前在亚信从事 spark 相关研究工作，希望能在这个平台认识更多的 spark 爱好者，一起为社区贡献力量。

初译（按工作量排名）：

	**CSDN ID：Aylee_Liu** **昵称/姓名：**Ayleeliu **个人简介：**我不认同“不以物喜，不以己悲”，但并不代表我要大喜大悲，遇到开心的事要笑，对自己的缺点不避讳；我喜欢向日葵，不是因为她高傲，而是她可以一直面对阳光，作为一个小人物，我只信奉：做好眼前的事，未来一定有惊喜。
	**CSDN ID： qfdai2** **昵称/姓名：**代其锋 **个人简介：**沉迷 Spark 已有半载，被 Spark 的设计原理和强大功能所深深吸引，这次能有幸参与 Spark 主要作者 Matei Zaharia 博士的毕业论文让我不仅对作者开发 Spark 的思路脉络有了清晰认识，更让自己能站在一个更高视角了解大数据的发展和趋势。
	**CSDN ID：**shiyuzh2007 **昵称/姓名：**AlexZhou **个人简介：**平和 追求 希望 珍惜
	**CSDN ID：caidaoqq** **昵称/姓名：**潘义文 **个人简介：**妹子，能交个朋友吗？哈哈.....

	CSDN ID：u011582658 **昵称/姓名：**雷力明 **个人简介：**国内某小二本（XJTU）一个，正在上研一。平时喜欢读书，有时写点代码，有时看看论文，有时出去户外运动，有时看看电影，还喜欢打游戏，Braid 死忠粉。
	CSDN ID：sun7545526 **昵称/姓名：**孙爱华 **个人简介：**之前几年一直接触 j2ee，最近从事云计算的研究，范围包括 openstack,ceph,hadoop 等技术，初出茅庐的 spark 其魅力让我无法抗拒，相信它一定会有更好的前景。
	CSDN ID：litao471625wo **昵称/姓名：**栗涛 **个人简介：**非常幸运可以参与到 Spark 论文的翻译工作，也收获了很多理解和研究论文的经验。不能像阅读论文的时候，遇到不太理解的词语、概念，可以跳过去，翻译的过程更像一个研究的过程，要理解上下文，来表达某些语句的技术重点。希望以后还可以更多的参与到类似的翻译工作，一起和大家交流学习。
	CSDN ID：zhangkan1983 **昵称/姓名：**张侃 **个人简介：**希望能多为开源社区做一些贡献，正从事大数据/车联网相关工作，欢迎交流。
	CSDN ID：laizx **昵称/姓名：**赖正兴 **个人简介：**一名热爱软件开发技术的老程序员！

	**CSDN ID：luogankungmail** **昵称/姓名：**PK 时发型不乱 **个人简介：**PK 时发型不乱
	**CSDN ID：lvhaozhi** **昵称/姓名：**吕浩志 **个人简介：**感谢 CSDN 给了我开阔眼界的机会。
	**CSDN ID：jacty0219** **昵称/姓名：**陈骏 **个人简介：**一个略微忧郁的英语爱好者兼码农，正在慢慢得朝着笔译之路前行。
	CSDN ID：wuyang630 **昵称/姓名：**武扬 **个人简介：**从大公司起步，到小公司创业，无论是谈技术还是谈事业希望能与更多志同道合的同学交流
	CSDN ID：yuangeqingtian **昵称/姓名：**yuangeqingtian **个人简介：**下次有这种项目，记得叫上我

	CSDN ID：lazyman500 **昵称/姓名：**Dongxu **个人简介：**这个人很懒，什么都没有留下。。
	**CSDN ID：**liuchao_9 **昵称/姓名：**刘超 **个人简介：**感谢 CSDN 发起这次协作翻译，以及参与协调的工作人员。很多优秀的技术文档都是英文的，平时也是直接看英文的，也觉得自己可以读懂，没有什么问题，但当要翻译成中文，贡献给读者时才发现很难。 一句话可能要仔细琢磨好多次，在不改变原作者意思
	**CSDN ID：**ljkang1990 **昵称/姓名：**刘见康 **个人简介：**大家好，我叫刘见康，人称康帅博，健康的康，帅气的帅，博学的博。我的理想是成为一名德智体美劳全面发展的暴栈工程师，因为不会弹吉他的摄影师不是好程序员。平时喜欢看书、听音乐、摄影，弹弹吉他唱唱歌，篮球羽毛球打的不错，代码写的也还可以，不约，谢谢！
	**CSDN ID：**qwewegfd **昵称/姓名：**杨志斌 **个人简介：**爱老婆，爱儿子，我爱我家。
	**CSDN ID：**usen521 **昵称/姓名：**张冰 **个人简介：**在业余时间能有机会结合自己的兴趣爱好做点积极的事情，是一件很有乐趣的事。参与翻译活动纯属偶然，但很高兴得到这么一个机会，认真的翻译认真的玩，不求多么完美，自己满意就好。

	**CSDN ID：xhz1234** **昵称/姓名：**徐洪志 **个人简介：**没伞的孩子，拼命跑
	**CSDN ID：pastgift** **昵称/姓名：**周逸灵（本本乱） **个人简介：**周逸灵，男，汉，1987 年生，2010 年日语毕业；籍贯江苏，现居上海；后端开发，熟悉 C、Python、Docker；热爱技术、涉猎广泛。
	**CSDN ID：Martin19870726** **昵称/姓名：**周项勇(Martin Zhou) **个人简介：**致力于实时/离线大数据分析！实时大数据分析系统 Druid 拓荒者。热心开源事业，Zookeeper 管理系统 ZookeeperEdit、Zookeeper 集群一键安装脚本 Zookeeper-Cluster Installer 开发者。现从事在线广告业务数据分析，和 DSP(Demand-Side Platform)系统研发工作。
	**CSDN ID：u011941712** **昵称/姓名：**籽皓 **个人简介：**谢谢 CSDN 的这次活动，让我了解了自己曾经不知道的技术，希望下次还可以参加类似的活动。
	CSDN ID：fancylee0808 **昵称/姓名：**李奕飞 **个人简介：**

	CSDN ID：LinuxCoder **昵称/姓名：**LinuxCoder **个人简介：**美国拿到计算机硕士学位，在国外从事 7 年的技术工作。
	**CSDN ID：S1012W2** **昵称/姓名：**叶秋 **个人简介：**丘吉尔曾说过，"We make a living by what we get, but we make a life by what we give." 我虽拥有的不多，但也希望能发挥自己所长做些有意义的事情。不给自己设限，世界就没有边界。
	CSDN ID：wanghua13717589807 **昵称/姓名：**王华 **个人简介：**
	CSDN ID：ytfuestc **昵称/姓名：**袁腾飞 **个人简介：**爱生活，爱程序，爱篮球，正在奋力研究 spark
	**CSDN ID：zhang177** **昵称/姓名：**张刚 **个人简介：**大家好，我叫张刚，2011 年硕士毕业，现任职于某研究院从事项目管理及软件设计工作， 爱好笛子，游泳，心理咨询等，业余时间热衷于公益活动。我相信一个人的视野决定他的深度，一个人的思维决定他的高度，所以不断学习，不断挑战，将是我不变的追求。

以下译者对本文亦有贡献：

CSDN ID: u012830490 **昵称/姓名：**私家宅院 **个人简介：**为了让生活更精彩!
CSDN ID: u014388509 **昵称/姓名：**OopsOutOfMemory
CSDN ID:harryxujiao **昵称/姓名：**馬小喬
CSDN ID: pandonghua_de **昵称/姓名：**pandonghua_de
CSDN ID:lu8000 **昵称/姓名：**木风卜雨
CSDN ID:kevenking **昵称/姓名：**kevenking
CSDN ID: mqshen **昵称/姓名：**mqshen

摘要

基于大型集群的快速通用数据处理架构

由

计算机科学博士 Matei Alexandru Zaharia

加州大学伯克利分校教授、主席 Scott Shenker 撰写

过去的几年中，计算系统经历着重大的变革，为了满足不断增长的数据量和处理速度需求，越来越多的应用向分布式系统扩展。如今，从互联网到企业运作，再到科技设备，不尽其数的数据源都在产生大量的、有价值的数据流。然而，单一的机器处理能力并没有跟上数据增长的速度，使得这些有价值的数据越来越难以被使用。以至于越来越多的组织——不仅仅是互联网公司，还有一些传统企业和研究室——迫切需要将他们重要的计算能力扩展到成百上千台机器上去。

在这同时，数据处理所需的速度和复杂性也在逐渐增加。在许多领域中，除了简单的查询，像机器学习和图分析这样的复杂算法也得到日益广泛的应用。另外，除了批量处理，一些组织还需要在实时数据源上进行流分析，以保证能够及时采取行动。未来的计算平台不仅需要能满足常规作业的扩展，同时也需要对新的应用有更好的支持。

针对上述的各种问题，本文提出了一种集群计算架构，能够解决这些新出现的数据处理作业的需求，同时还可以应对越来越大规模的扩展。虽然早期的集群计算系统，如 MapReduce，已经能够进行批量处理，但我们的架构更支持流处理和交互查询，并且拥有和之前系统相同的可扩展性和容错性。然而当前所部署的大部分的系统仅支持简单的单路运算(例如,聚合或SQL查询)，而我们的系统针更为复杂的分析（例如，机器学习的迭代算法）扩展到了对多路算法的支持。最后，与处理特定工作的专有系统不同的是，我们的架构允许这些算法相互结合，从而实现更丰富的新应用。例如，流处理和批量处理，或 SQL 和复杂分析之间的相互结合。

为了实现上述的各种特性，我们通过简单的扩展 MapReduce，为其增加了数据共享原语，也就是所谓的弹性分布式数据集（RDDs）。我们发现，这样的扩展足以能够有效地覆盖大部分作业的需求。在开源的 Spark 系统中我们实现了 RDDs，同时使用了模拟测试程序和真实的用户应用对其进行评估。在许多应用领域中，Spark 已经接近或是超过了专有系统的性能，同时提供更强

大的容错保证，并允许这些作业之间能够进行结合。我们从理论建模和实践的角度去探索 RDDs 的通用性，来解释为什么这样的扩展可以覆盖大范围的不同作业需求。

致谢

感谢我的导师 Scott Shenker 和 Ion Stoica 教授，在我博士期间对我孜孜不倦的指导。他们都是非常卓越的研究者，总是能够将想法往前推进一步，为我们提供完成任务所需的条件，以及分享他们做研究的经验。特别荣幸能和他们两人一起工作，他们的观点让我受益。

本论文的工作是与其他很多人合作的结果。第 2 章是和 Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Mike Franklin, Scott Shenker 及 Ion Stoica 一起合作的 [118]。第 3 章中介绍的 Shark 项目部分，是与 Reynold Xin, Josh Rosen, Mike Franklin, Scott Shenker 及 Ion Stoica 一起开发的[113]。第 4 章是与 Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker 及 Ion Stoica 一起合作的[119].。更广泛的说，很多在 AMPLab 以及参与 Spark 相关项目如 GraphX [112] 和 MLI [98]的人，都对文中的思想形成和完善有所贡献。

除了这个项目中的直接合作者，很多人对我博士期间的工作都做出了贡献，这些都促成 Berkeley 成为了难忘的经历。在多次的喝茶过程中，Ali Ghodsi 在研究和开源方面都提出了一些特别好的建议和想法。和 Ben Hindman, Andy Konwinski，Kurtis Heimerl 在一起非常有趣，他们在一些好的想法上都是非常棒的合作者。Taylor Sittler 让我和 AMPLab 的很多人对生物学产生了非常大的兴趣，这促成了我和 Bill Bolosky, Ravi Pandya, Kristal Curtis, Dave Patterson 及其他很多人在 AMP-X 加入的最有趣的小组之一。在其它项目中，我也有幸与 Vern Paxson, Dawn Song, Anthony Joseph, Randy Katz 和 Armando Fox 一起合作，并从他们的见解中学到知识。最后，AMPLab 和 RADLab 是一个奇妙的组织，无论是 Berkeley 的成员，还是工业界的一些接触者，都在不断地给我们建议。

我也非常荣幸参与到早期的开源大数据社区工作。在 Facebook，Dhruba Borthakur 以及 Joydeep Sen Sarma 引导我开始为 Hadoop 做出贡献，同时，与 Eric Baldeschwieler, Owen O'Malley, Arun Murthy, Sanjay Radia 和 Eli Collins 参与的很多讨论，让我们的研究想法得到实现。在我们开始开发 Spark 项目后，我一直都被那些才华横溢和热情的贡献者所折服。Spark 和 Shark 的贡献者目前已经超过 130 人了，非常感谢每个人，使得这些项目变成现实。当然，这些项目的用户也做出了巨大的贡献，他们持续的提出了很多好的建议，并一直推动系统朝新的方向上发展，这些都影响着核心设计。在其中，我特别感谢该项目的早期用户，他们包括 Lester Mackey, Tim Hunter, Dilip Joseph, Jibin Zhan, Erich Nachbar, 以及 Karthik Thiyagarajan。

最后，感谢我的家人及朋友在我读博士期间对我坚定的支持。

目录

第 1 章 简介

在过去的几年里已经看到了计算机系统的重大变革，随着数据量的不断增长越来越多的应用需要扩展到大型集群。在商业和科学领域，新的数据源和工具*(例如,* 基因测序仪，RFID 和 Web)正在生产越来越多的信息。不幸的是，单机的处理能力和 I/O 性能并没有跟上这种增长。这样一来，越来越多的企业不得不向外扩展他们的计算至集群模式。

可编程的集群环境会带来一些挑战。第一个是并行化：这需要以并行的方式重写应用程序，同时这种编程模型能够处理范围广泛的的计算。然而，与其他并行平台相比，集群的第二个挑战是容错:在大规模的情况下节点故障和 *straggler* （慢节点）将变得很常见，而且可以极大地影响应用程序的性能。最后，集群通常在多个用户之间共享，因此需要在运行时可以动态地扩展和缩减计算资源，而且加剧了应用互相干扰的可能性。

因此，各种各样针对集群的新的编程模型已经被设计出来。起初，谷歌的 MapReduce[36]提出了一种简单通用而且能够自动处理故障的批处理计算模型。然而，MapReduce 并不适合其他类型的计算任务，以至于出现了大量的与 MapRedeuce 有显著不同的特制的编程模型。例如,在谷歌，Pregel[72] 提供了一个 bulk-sunchronous parallel（BSP）并行迭代图计算模型；F1[95]是一个快速但没有容错的 SQL 查询系统；MillWheel[2] 支持连续地流式处理。谷歌之外，像 Storm [14], Impala [60], Piccolo [86] and GraphLab [71]系统提供了相似的模型。随着每年新模型持续地出现，集群计算势必需要一系列的解决不同的计算工作的方案。

本论文讨论的刚好相反，我们可以设计一个*统一* 的编程抽象，不仅可以处理这些不同的计算任务，而且能使新的应用更好的编程。特别的是，我们将展示 MapReduce 的一个简单扩展，称为弹性分布式数据集（RDDS）,它增加了高效的数据共享元语，以及大大增加了它的通用性。由此产生的架构比当前系统有几个关键优势：

1. 在相同的运行环境下，它支持批处理、交互式、迭代和流计算，*结合*这些模式提供丰富的应用编程，并且相对于单一模式的系统能更好的发挥其性能。
2. 它以很小的代价在这些计算模式上提供结点故障和 straggler 的容忍功能。事实上，在一些地方（如流和 SQL），基于 RDD 产生的新系统比现有的系统有更强的容错性。
3. 它实现的性能往往比 MapReduce 高 100 倍，并可媲美各个应用领域的专业系统。
4. 这很适合多组织用户管理，允许应用程序弹性地扩缩容和响应式地共享资源。

我们实现了基于 RDD 的架构，在这个开源系统栈里包括作为公共组件的 Apache Spark;处理 SQL 的 Shark;和处理分布式流的 Spark Streaming（图 1.1）。我们使用了真实的用户应用案例和传统的基准测试来评估这些系统。我们的实现为传统和新的数据分析工作提供了很好的性能，并成为第一个使得用户可以组合这些计算任务的平台。

从更长远的角度来看，我们也讨论了在 RDD 上实现各种数据处理的通用技术，以及证实为什么 RDD 是如此通用。随着集群的应用程序变得越来越复杂，我们相信通过 RDD 提供的这种统一处理架构将在性能和易用性变得越来越重要。

论文声明：*基于弹性分布式数据集的单个执行模型可以有效地支持不同的分布式计算。*

在本章的其余部分，我们说明了 RDD 设计的一些动机，然后突出展示我们的主要成果。

1.1 专业系统相关的问题

今天的集群计算机系统越来越多地专门针对特定的应用领域。虽然像 MapReduce 和 Drayed[36, 61]这样的系统模型目标是在于覆盖相当通用的计算，然而研究员和从业者已经为新的应用领域研发了越来越多的专业系统。

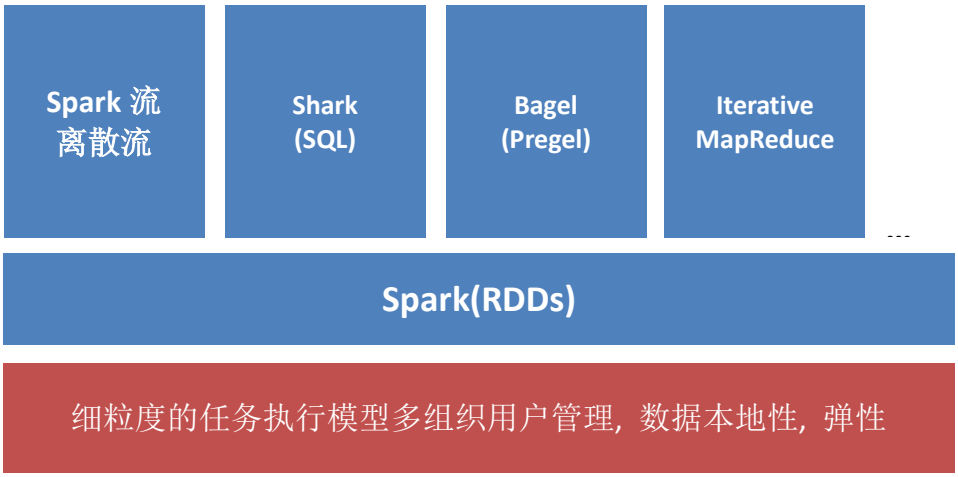


图 1.1 本论文中计算栈的实现

最近通过 Spark 的 RDDs（弹性分布式数据集）的实现，我们建立起了其他的计算模型，如流式计算，SQL 和图计算，所有这些都可以混杂在 Spark 程序中。RDDs 本身利用一系列的细粒度的任务来执行应用程序，能有效的共享资源。

其中包括交互式 SQL 查询系统 Dremel 和 Impala[75,60],图计算处理系统 Pregel[72],机器学习系统 GraphLab，等等。

虽然这些专业系统似乎天然地减少了那些在分布式环境中具有挑战性的问题，但他们也有一些缺点：

1. 重复工作：许多专业系统仍然需要解决同样的潜在问题，如分布式执行和容错性。举个例子，分布式 SQL 引擎或机器学习引擎都需要执行并行聚合。对于独立的系统，针对每个领域也是需要解决这些问题。

2. 组成：不同系统的组合计算既昂贵也笨重。尤其是对于"大数据"应用，中间处理过程的数据集是庞大的且难以移动的。为了使得在各个计算引擎之间共享数据，当前的环境需要将数据导出到稳定且多备份的存储系统中，通常这比实际计算要多出更多的消耗。因此，相比于一栈式的系统，由多个系统组成的管道常常是低效的。

3. 范围限制：如果应用程序不符合专业系统的编程模型，用户要不修改程序以适应当前的系统，要不就针对该程序写一个新的运行系统。

4. 资源共享：在计算引擎之间动态共享资源是很困难的，因为大多数引擎在应用程序运行期间都假定独自拥有一组机器。

5. 管理和管理员：相对单一的系统，独立的系统需要更多的工作用于管理和部署。对于用户来说，它们需要学习多种 API 和执行模型。

由于这些限制，集群计算的统一抽象在易用性和性能方面都有显著的好处，特别是对于复杂的应用程序和多用户环境下。

1.2 弹性分布式数据集（RDDS）

为了解决这个问题，我们引入一个新的概念，*弹性分布式数据集*（RDDs），它是 MapReduce 模型一种简单的扩展和延伸。进一步说，虽然乍一看那些不适合 MapReduce 的计算任务*（例如，*迭代，交互性和流查询）之间存在着明显的不同，但他们却都有一个功能特性，也是 MapReduce 模型的缺陷：在并行计算阶段之间能够高效地*数据共享，*这正是 RDD 具有真知灼见的地方。运用高效的数据共享概念和类似于 MapReduce 的操作方式，使得所有这些计算工作都可以有效地执行，并可以在当前特定的系统中获得关键性的优化。RDDs 以一种既高效有能容错的方式为广泛的并行

计算提出这样一个抽象。

特别提出的是，以前的这些集群容错处理模型，像 MapReduce、Dryad，将计算转换为一个有向非循环图（DAG）的任务集合。这使得它们能够高效地重复执行 DAG 里的其中一部分任务来完成容错恢复。但对于一个独立的计算，（*例如*在一个迭代过程中），这些模型除了可复制的文件系统外没有提供其他存储的概念，这就导致因为在网络上进行数据复制而增加了大量的消耗。RDDs 是一个可以*避免*复制的容错分布式存储概念。取而代之，每一个 RDD 都会记住由构建它的那些操作所构成的一个图，类似于批处理计算模型，可以有效地*重新*计算因故障丢失的数据。由于创建 RDDS 的操作是相对粗粒度的，即单一的操作应用于许多数据元素，该技巧比通过网络复制数据更高效。RDDs 很好地运用于当前广泛的数据并行算法和处理模型中，所有的这些对多个任务使用同一种操作。

现在它看起来很神奇，只是增加数据共享却极大地提高了 MapReduce 的通用性，那就让我们从几个方面探讨为什么会这样。首先，从表现力的角度来说，我们了解到 RDDs 可以效仿*任何*一种分布式系统，并且会在容许网络延迟的条件下做的非常高效。这是因为，一旦增加了快速数据共享机制，MapReduce 可以效仿并行计算中的 Bulk Synchronous Parallel (BSP) [108] 模型，而主要的缺陷是每个 MapReduce 的阶段会有延迟。根据经验，在我们的 Spark 系统中，这可以低至 50～100 毫秒。其次，从系统的角度来说，不像普通的 MapReduce，RDDs 在大多数集群计算中会给应用足够的控制以便优化资源瓶颈（特别是网络和存储 I/O）。因为这些资源经常占据主要的执行时间，通常仅控制它们(*例如，* 通过控制数据位置）就能达到使用相同资源的独立系统的性能。

除了这种探索，我们还实证研究表明，使用 RDDs 我们可以实现多种目前使用的专用模型，以及新的编程模型。我们的实现能达到专业系统的性能，同时提供丰富的容错特性和组合。

1.3 基于 RDD 机制实现的模型

我们使用 RDD 机制实现了多类模型，包括多个现有的集群编程模型和之前模型所没有支持的新应用。在这些模型中，RDD 机制不仅在性能方面能够和之前系统相匹配，在其他方面，他们也能加入现有的系统所缺少的新特性，比如容错性，straggler 容忍和弹性。我们讨论以下四类模型。

迭代式算法 一种目前已经开发的针对特定系统最常见的的工作模式是迭代算法，比如应用于图处理，数值优化，以及机器学习中的算法。RDD 可以支持广泛类型的各种模型，包括 Pregel[72]，像 HaLoop 和 Twister 这类的迭代式 MapReduce 模型[22， 37]，以及确定版本的 GraphLab 和 PowerGraph 模型[71,48]。

关系查询 在 MapReduce 集群中的首要需求中的一类是执行 SQL 查询，长期运行或多个小时的批量计算任务和即时查询。这促进了很多在商业集群中应用的并行数据库系统的发展[95, 60, 75]。MapReduce 相比并行数据库在交互式查询[84]有非常大的缺陷，例如 MapReduce 的容错机制模型，而我们发现通过在 RDD 操作中实现很多常用的数据库引擎的特性（*比如，*列处理），这样能够达到相当可观的性能。由上述方式所构建的系统，Shark[113]，提供完整的容错机制，能够在短查询和长查询中很好的扩展，同时也能在 RDD 之上提供复杂分析函数的调用*(例如,* 机器学习)。

MapReduce RDD 通过提供 MapReduce 的一个超集，能够高效地执行 MapReduce 程序，同样也可以指向比如 DryadLINQ 这样常见的机遇 DAG 数据流的应用[115]。

流式数据处理 我们的系统与定制化系统最大的区别是我们也使用 RDD 实现了流式处理。流式数据处理已经在数据库和系统领域进行了很长时间研究，但是实现大规模流式数据处理仍然是一项挑战。当前的模型并没有处理在大规模集群中频繁出现的 straggler 的问题，同时对故障恢复的方式也非常有限，需要大量的复制或浪费很长的恢复时间。特别是，当前的系统是基于一种*持续操作*的模型，这就需要长时间的有状态的操作处理每一个到达的记录。为了恢复一个丢失的节点，当前的系统需要保存每一个操作符的两个副本，或通过一系列耗费大量开销的串行处理来对上游的数据进行重放。

我们提出了一个新的模型，*离散数据流*(D-Streams),来解决这样的问题。对使用长期状态处理的过程进行替换，D-Streams 把流式计算的执行当做一系列短而确定性的批量计算的序列，将状态保存在 RDD 里。D-Stream 模型通过根据相关 RDD 的依赖关系图进行并行化恢复，就能达到快速的故障恢复，*这样不需要通过复制*。另外，它通过推测(Speculative)来支持对 straggler 迁移执行[36]，例如，对那些慢任务运行经过推测的备份副本。尽管 D-Stream 将计算转换为许多不相关联的 jobs 来运行从而增加了部分延迟，然而我们证明了 D-Stream 能够被达到次秒级延时的实现，这样能够达到以前系统单个节点的性能，并能线性扩展到 100 个节点。D-Stream 的强恢复特性让他们成为了第一个处理大规模集群特性的流式处理模型，并且他们基于 RDD 的实现使得应用能够有效的整合批处理和交互式查询。

通过将这些模型整合到一起，RDD 还能支持一些现有系统不能表示的新的应用。例如，许多数据流应用程序还需要加入历史数据的信息；通过使用 RDD 可以在同一程序中同时使用批处理和流式处理，这样来实现在所有模型中数据共享和容错恢复。同样的，流式应用的操作者常常需要在数据流的状态上执行即时查询；在 D-Stream 中的 RDD 能够如静态数据形式进行查询。我们使用一些在线机器学习 (第 4.6.3 节) 和视频分析(第 4.6.3 节)的实际应用来说明了这些用例。更一般

的说，每一个批处理应用常常需要整合多个处理类型：比如，一个应用可能需要使用 SQL 提取一个数据集，在数据集上训练一个机器学习模型，之后对这个模型进行查询。由于计算的大部分时间花在系统之间共享数据的分布式文件系统的 I/O 开销上，因此使用当前多个系统组合而成的工作流的效率非常的低下。使用一个基于 RDD 机制的系统，这些计算可以在同一个引擎中紧接着执行，而不需要额外的 I/O。

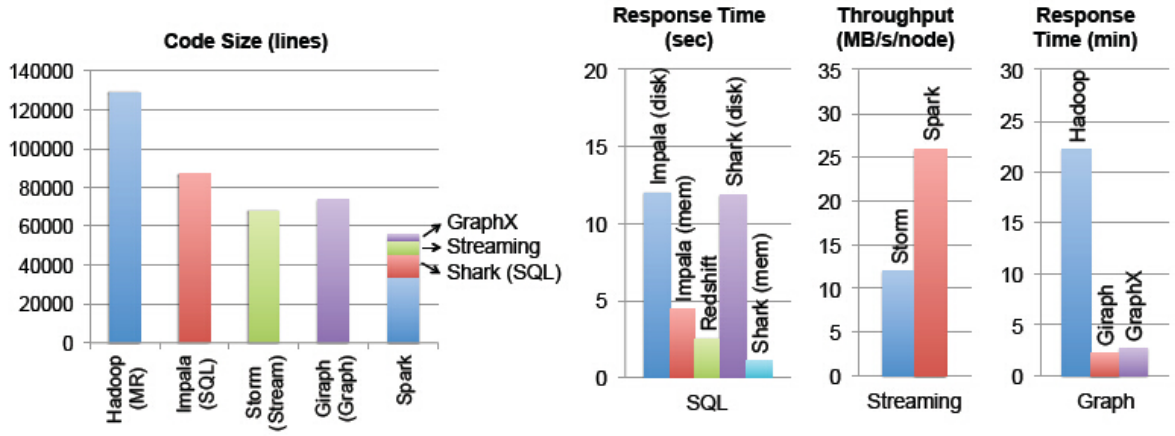


图 1.2.Spark 栈和定制化系统在代码量和性能上的比较

Spark 的代码量和定制化系统是相近的，然而这些模型在 Spark 上的实现代码量明显要少。尽管如此，在选定的应用中的 Spark 的性能可以和定制化系统相媲美。

1.4 总结

我们在托管于 Apache 孵化器而且已经用于多个商业部署的开源系统 Spark 中实现了 RDDs。尽管 RDD 很通用，但 Spark 相对较小:共 34，000 行 Scala(公认的高级语言)代码，在同一范围内把它作为专业的集群计算系统。更重要的是，建立于 Spark 上的专业模型比它们单独运行的时候小得多:我们用几百行代码实现 Pregel 和交互性的 MapReduce,8000 行代码实现了离散 Stream,12000 行代码实现一个以 Apache Hive 作为 Spark 前段进行查询的 SQL 系统 Shark。这些基于 spark 的系统比单独的特定实现小几个数量级且支持各种方法的混合模型，但是在性能上仍然比得上专业系统。简短总结一下,图 1.2 从性能和代码规模上对 Spark 及建立于 Spark 上的 3 个系统(Shark,Spark Streaming,GraphX）[113,119,112]，和广受欢迎的专业系统（Impala，Amazon Redshift—处理 SQL 的 DBMS;Storm—流处理;Giraph—图处理）[60,5,14,10]进行了比较。

除了这些实际的结果，我们也包括通过 RDD 实现复杂处理函数的通用技术以及讨论为什么 RDD 模型如此受欢迎。尤其是在 1.2 章节中表述的那样，我们发现 RDD 模型可以与任何分布式系统竞争，且

有着比 MapReduce 更高效的表现。而实际上，RDD 接口比起专业系统在集群瓶颈资源方面，给予了应用足够的自由控制，而且仍然可以实现自动容错恢复和高效的组合。

1.5 论文计划

本文组织结构如下。第 2 章介绍了 RDD 抽象并涵盖了一些简单的编程模型的应用。第 3 章介绍了 Shark SQL 系统基于 RDDs 实现的更高级的存储和处理模型的技术。第 4 章介绍了如何使用 RDDs 开发离散的流，这是一种新的流式处理模型。第 5 章则介绍了为什么 RDD 模型在这些应用中如此通用，同时介绍它的限制和扩展性。最后，在第 6 章，我们总结和讨论一些未来工作的可能方向。

第二章 弹性分布式数据集

2.1 简介

在本章中，我们提出了弹性分布式数据集（RDD）的抽象概念，论文其余部分基于此建立了一个通用的集群计算栈。RDD 对 MapReduce [36] 和 Dryad [61]提出的数据流编程模型进行了扩展,这些模型是目前大数据分析使用最为广泛的编程模型。数据流系统取得了成功，很重要的因素是用户通过使用比较高级的操作进行计算而无需担心任务分布和系统的容错问题。然而，随着集群负载的增加，数据流系统在很多重要的应用场景出现了低效率问题，比如迭代算法，交互式查询和流式处理。这引发了大量针对这些应用而定制的计算框架的发展[72, 22, 71,95, 60,14, 2]。

我们的工作源于观察到很多数据流模型不适用的应用场景所共有的一个特征： 在计算过程中都需要高效率的*数据共享*。例如，迭代算法，如 PageRank, K-means 聚类，或逻辑回归，都需要进行多次访问相同的数据集；交互数据挖掘经常需要对于同一数据子集进行多个特定的查询；而流式应用下则需要随时间对状态信息进行维护和共享。不幸的是，尽管数据流框架支持大量的计算操作运算,但是它们缺乏针对数据共享的高效原语。在这些框架中，实现计算之间(例如，两个的 MapReduce 作业之间)数据共享只有一个办法，就是将其写到一个稳定的外部存储系统，如分布式文件系统。这会引入数据备份、磁盘 I/O 以及序列化，这些都会引起大量的开销，从而占据大部分的应用执行时间。

事实上，在针对这些新应用而定制的框架进行研究的过程中，我们的确有发现它们会对数据共享进行优化。例如，Pregel[72]是一种针对图迭代计算的系统，它会将中间状态保存在内存中。而 HaLoop[22]是一种迭代 MapReduce 的系统，它会在各步骤中都以一种高效率的方式对数据进行分区。不幸的是，这些框架只能支持特定的计算模式（*例如* ，循环一系列的 MapReduce 的步骤），并对用户屏蔽了数据共享的方式。它们不能提供一种更为通用的抽象模式，*例如*，允许一个用户可以加载几个数据集到内存中并进行一些跨数据集的即时查询。

相反，我们所提出的弹性分布式数据集（RDDs），这种全新的抽象模式令用户可以直接控制数据的共享。RDD 具有可容错和并行数据结构特征，这使得用户可以指定数据存储到硬盘还是内存、控制数据的分区方法并在数据集上进行种类丰富的操作。他们提供了一个简单

高效的编程接口，可以同时满足现有的特定模型和全新的应用场景。RDD 设计时的最大挑战在于定义一个能提供*高效*容错能力的编程接口。现有的基于集群的内存存储抽象，比如分布式共享内存[79],键-值存储[81],数据库,以及 Piccolo[86],提供了一个对内部状态基于细粒度更新的接口(*例如,*表格里面的单元).在这样的设计之下,提供容错性的方法就要么是在主机之间复制数据，要么对各主机的更新情况做日志记录。这两种方法对于数据密集型的任务来说代价很高，因为它们需要在带宽远低于内存的集群网络间拷贝大量的数据，同时还将产生大量的存储开销。

与上述系统不同的是，RDD提供一种基于*粗粒度*变换（*如，* map, filter, join）的接口，该接口会将相同的操作应用到多个数据集上。这使得他们可以通过记录用来创建数据集的变换（*lineage*），而不需存储真正的数据，进而达到高效的容错性。[1] 当一个RDD的某个分区丢失的时候，RDD记录有足够的信息记录其如何通过其他的RDD进行计算，且只需重新计算该分区。因此，丢失的数据可以被很快的恢复，而不需要昂贵的复制代价。

尽管基于粗粒度变换的接口显得很局限，但 RDD 针对很多应用都有很好的普适性，因为这些应用可以自然地对多个数据项使用同样的操作。事实上，RDD 可充分表达多种现有的集群编程模型。这些模型之间相互独立，它们包括 MapReduce、DryadLINQ、SQL、Pregel 和 HaLoop 以及一些这些系统无法涵盖的新需求，如交互式数据挖掘。RDD 对新计算需求的适用能力是对 RDD 抽象优势的最佳见证。在这之前，这些新的需求在只能通过创建新的计算框架才能得到满足。

RDD 已经在一个名为 Spark 的系统中实现，该系统正广泛应用于 UC Berkeley 和其他公司的研究和生产环境下。与 DryadLINQ[115]类似，Spark 提供了一种便捷的语言集成编程接口。该接口用 Scala[92]语言实现。此外，借助 Scala 解释器，Spark 提供对大数据集进行交互式查询的功能。我们相信 Spark 是首个支持用通用编程语言来在集群上实现交互速度级的内存数据挖掘的系统。

我们基准程序和用户程序中的衡量指标对 RDD 和 Spark 进行了评估。结果表明，Spark 在迭代性的应用上比 Hadoop 最高可快 80 倍，真实的数据分析应用上快 40 倍，而且在 1TB 的数据上实现 5-7 秒内的交互式扫描。最后，为展现 Spark 的通用性，我们在 Spark 实现了 Pregel 和 HaLoop 编程模型。这些实现包括它们各自的分布优化，并以相对较小的库(每个约 200 行代码)来提供这些功能。

本章首先会分别对 RDD(章节 2.2) 和 Spark (章节 2.3)进行概述。随后会详细讨论 RDD 的内部描述(章节 2.4)，具体实现过程(章节 2.5)以及实验结果(章节 2.6)。最后，我们会

[1]当 lineage 增加到做够大时，对某些 RDD 中的数据进行检查或将变得有意义。相关细节我们将会在 2.5.5 节进行的讨论。

讨论 RDD 如何适用于现有的编程模型 (章节 2.7)，阐述相关的工作 (章节 2.8)，并最终得出结论。

2.2 RDD 概述

本节提供 RDDs 的概述。首先我们看下 RDD (§2.2.1)的概念以及它们在 Spark (§2.2.2)中的编程接口。然后比较下 RDD 与细粒度共享内存 (finer-grained shared memory)。最后我们讨论 RDD 模型的局限性。

2.2.1 概念

从形式上看，RDD是一个分区的只读记录的集合。RDD只能通过在（1）稳定的存储器或（2）其他RDD的数据上的确定性操作来创建。我们把这些操作称作*变换*以区别其他类型的操作。例如 *map, filter,* 和 join。[2]

RDD 在任何时候都不需要被"物化"(进行实际的变换并最终写入稳定的存储器上)。实际上，一个 RDD 有足够的信息描述着其如何从其他稳定的存储器上的数据生成。它有一个强大的特性：从本质上说，若 RDD 失效且不能重建，程序将不能引用该 RDD。

最后，用户可以控制 RDD 的其他两个方面：*持久化*和*分区*。用户可以选择重用哪个 RDD，并为其制定存储策略*(比如，* 内存存储)。也可以让 RDD 中的数据根据记录的 key 分布到集群的多个机器。 这对位置优化来说是有用的，比如可用来保证两个要 Jion 的数据集都使用了相同的哈希分区方式。

2.2.2 Spark 编程接口

Spark 通过一种类似于 DryadLINQ [115] 和 FlumeJava [25]集成语言 API 来对外提供 RDD 的功能。具体来说，每一个数据集都会表示为一个对象，而各种变换则通过该对象相应方法的调用而实现。

[2]尽管单个的 RDDS 是不可变的，但可以通过多个 RDDs 来表示一个数据集的多个版本来实现可变。这种性质（不可变）使得描述其 lineage(获取 RDD 所需要经过的变换)变得容易。可以这样理解，RDD 是版本化的数据集，并且可以通过变换记录追踪版本。

在最开始，编程人员通过对稳定存储上的数据进行变换操作*(e.g., map* 和 *filter).*而得到一个或多个 RDD。之后，他们可以调用这些 RDD 的 *actions*(动作)类的操作。这类操作的目的或是返回一个值，或是将数据导入到存储系统中。动作类的操作如 *count*(返回数据集的元素数),*collect*(返回元素本身的集合)和 *save*(输出数据集到存储系统)。与 DryadLINQ 一样，Spark 直到 RDD 第一次调用一个动作时才真正计算 RDD。这也就使得 Spark 可以按序缓存多个变换。

此外，编程人员还可以调用 RDD 的 *persist*(持久化)方法来表明该 RDD 在后续操作中还会用到。默认情况下，Spark 会将调用过 *persist* 的 RDD 存在内存中。但若内存不足，也可以将其写入到硬盘上。通过指定 *persist* 函数中的参数，用户也可以请求其他持久化策略并通过标记来进行 persist，比如仅存储到硬盘上，又或是在各机器之间复制一份。最后，用户可以在每个 RDD 上设定一个持久化的优先级来指定内存中的哪些数据应该被优先写入到磁盘。

例如：控制台日志挖掘

假设一个 Web 服务遇到错误，操作员要在 Hadoop 文件系统（HDFS [11]）里搜索 TB 级大小的日志，以查找原因。通过 Spark，操作员可以只把日志中的错误信息加载到多个节点的内存中，并进行交互式查询。可以先键入以下 Scala 代码：

```
lines = spark.textFile("hdfs://. . . “)
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

第 1 行定义了以一个 HDFS 文件（由数行文本组成）为基础的 RDD。第 2 行则从它派生了一个过滤后的 RDD。第 3 行要求 **errors** 在内存中持久化，以便它可以通过查询共享。需要注意的是 *filter* 的参数用的是 Scala 闭包的语法。

到此，集群上还没有工作被执行。但是，用户现在已经可以在动作(action)中使用 RDD。*例如：*计算消息的数量：

```
errors.count()
```

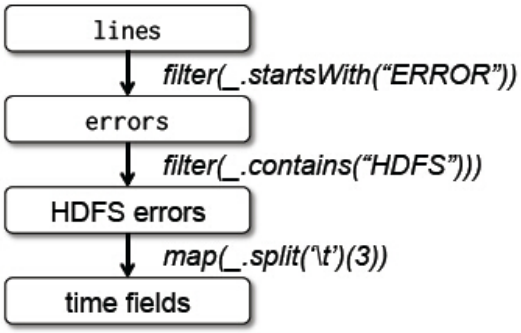


图 2.1 示例中第 3 个查询的 lineage 示意图方块代表 RDD，箭头表示的转换。

用户也可以在 RDD 上进行进一步的转换，并使用他们的结果，如下面的几行：

```
// 关于 MySQL 的错误计数:
errors.filter(_.contains("MySQL")).count()

// 将提及 HDFS 的错误的 time 字段作为数组返回，假设
// time 字段是用制表符分隔的格式:
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

在第一个涉及 errors 的动作运行后，Spark 会在内存中存储 errors 对应的分区，极大的加速了其后在上面运行的计算。需要注意的是首个 RDD——lines——*不会*加载到 RAM 中。这是值得的，因为错误消息可能只是数据（小到足以放入内存）的一小部分。

最后，为了阐明我们的模型如何实现容错，我们在图表 2.1 中展示了第 3 个查询所对应的 RDD 的 lineage。在这个查询里，我们最开始得出 erros 所对应的 RDD lines，接着对 lines 进行 filter(过滤)操作，之后再次进行过滤、做 Map 操作，最后进行 *collect*(收集)。Spark 的调度器会对最后的那个两个变换操作流水线化，并发送一组任务给那些保存了 errors 对应的缓存分区的节点。另外，如果 errors 的某个分区丢失，Spark 将只在该分区对应的那些行上执行原来的 filter 操作即可恢复该分区。

2.2.3 RDD 模型的优点

方面	RDDs	分布式共享内存
读	粗细粒度	细粒度
写	粗粒度	细粒度
一致性	不重要（不可变）	取决于应用程序/运行
容错性	细粒度并且使用 lineage 开销低	需要检查点和程序回滚
straggler 缓解	可以使用后备任务	难
工作分配	根据数据局部性自动分配	取决于应用程序（运行时透明）
当没有足够的 RAM 时行为	类似于现有的数据流系统	性能不佳（交换？）

表 2.1 RDDs 与共享内存（DSM）的比较

为了更好地理解 RDD 分布式内存抽象的好处，我们在表 2.1 中对分布式共享内存（DSM）进行了比较。在 DSM 系统，应用程序读取和写入全局地址空间的任意位置。值得注意的是，我们在这里所说的 DSM 不仅包括传统的共享内存系统[79]，还包括其他能让应用程序执行细粒度“写”共享状态的系统，例如提供共享 DHT 的 Piccolo[86]和分布式数据库。DSM 是一个非一般的抽象，但这种一般性使得它更难在普通集群中以一种有效且容错的方式实现。

RDDs和DSM之间的主要区别在于，RDDS只能通过粗粒转换创建（“写”），而DSM允许对每个存储单元读取和写入。[3]这使得RDD在批量写入主导的应用上受到限制，但增强了其容错方面的效率。具体来说，因为可以使用lineage恢复数据，RDD不需要检查点的开销。[4]此外，当出现失败时，RDDs的分区中只有丢失的那部分需要重新计算，而且该计算可在多个节点上并发完成，不必回滚整个程序。

RDDs 的第二个优点是，不可变性让系统像 MapReduce [36]那样用后备任务代替运行缓慢的任务来减少缓慢节点（stragglers）的影响。因为在 DSM 中任务的两个副本会访问相同

[4] 注意，RDDs 的*读*也可以细粒度进行。比如，应用程序可以把 RDDs 当作一个只读的查找表。

[4] 正如我们在第 2.5.5 节讨论的一样，在某些应用中也可以用长的 lineage 来进行节点检查。然而，节点检查可在后台完成，因为 RDDs 是不可变的。同样也没必要像 DSM 那样对*整个*应用程序创建快照。

的存储器位置和受彼此更新的干扰，这样后备任务在 DSM 中很难实现。

最后，RDDs 还具备了 DSM 的两个优点。首先，在 RDDs 上的批量操作过程中，任务的执行可以根据数据的所处的位置来进行优化，从而提高性能。其次，只要所进行的操作是只基于扫描的，当内存不足时，RDD 的性能下降也是平稳的。不能载入内存的分区可以存储在磁盘上，其性能也会与当前其他数据并行系统相当。

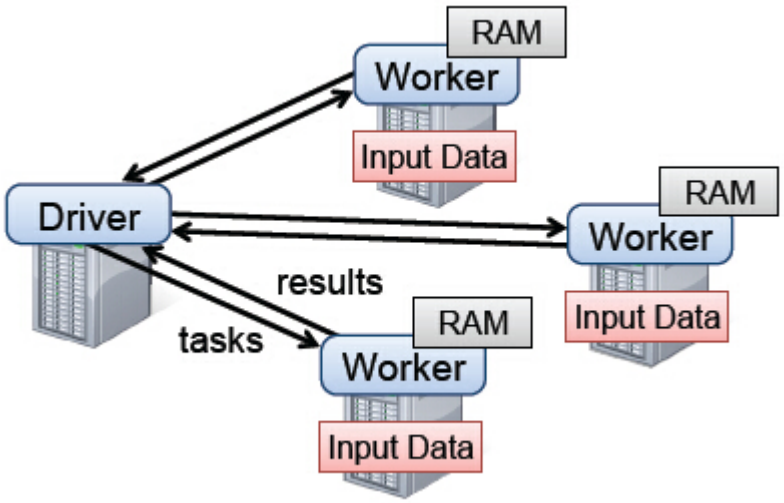


图 2.2　Spark 运行时，用户的驱动程序启动多个 worker，worker 从分布式文件系统中读取数据模块，并且可以将计算好的 RDD 分区持久化到内存中。

2.2.4　不适合 RDDs 的应用

正如在引言中讨论的，RDDS 最适合对数据集中所有的元素进行相同的操作的批处理类应用。在这些情况下，作为整个 lineage 图中的其中一步，RDD 高效地记住每一次变换，从而不需要对大量数据做日志记录便可恢复失效分区。RDDS 不太适用于通过异步细粒度更新来共享状态的应用，比如针对 Web 应用或增量网络爬虫的存储系统。对于这些应用，那些传统的更新日志和数据检查点的系统会更有效，例如数据库，RAMCloud [81]，Percolator [85] 和 Piccolo [86]。我们的目标是为批量分析提供一个高效的编程模型，这些异步应用仍然交由定制系统来处理。但是，第 5 章会提供一些把这些类型的应用与 RDD 模型结合起来的可能方法，比如批量更新。

2.3 Spark 编程接口

Spark 通过类似于 Scala [92](一种基于 Java 虚拟机的静态类型函数式的编程语言)中 DryadLINQ [115]的语言集成 API 来提供 RDD 的抽象定义，我们选择 Scala 是因为它的简洁（易于交互）和效率（静态类型），而不是 RDD 的抽象需要一种函数式语言。

为了使用 Spark,开发者需要写一个 *driver program* 来连接到 *workers* 集群，如图 *2.2.* 所示。驱动程序定义一个或多个 RDDs 以及相关的一些 action 操作。驱动上的 spark 代码也跟踪记录 RDDs 的继承关系，即 lineage。Worker 是一直运行着的进程，它将经过一系列操作后的 RDD 分区数据保存在内存中。

正如我们在 2.2.2 节的日志挖掘例子中所看到的,用户通过传递闭包的方式将参数传递给*map*等操作。在Scala中每个闭包都代表一个Java对象，这些对象可以被序列化，也可以通过网络将闭包传递给其他节点并加载。Scala会将闭包中的所有变量转义成Java对象的属性域。例如，我们可以写类似 `var x = 5; rdd.map(_ + x)` 的代码来将 5 加到RDD的每个元素上.[5]。

RDDs 是一个通过元素类型参数化的静态类型对象。*例如* RDD[Int]是一个整数型 RDD。然而，我们大多数的例子中忽略类型是因为 Scala 可以进行类型推断。

虽然我们在 Scala 中暴露 RDDS 的方法很简单，但是我们必须用反射来解决 Scala 的闭包对象问题[118]。为了使 Spark 在 Scala 的解释器中可用，我们还需要做更多的工作，我们将在 2.5.3.节讨论然而，我们没必要修改 Scala 解释器。

我们在闭包（closure）创建的时候就保存它，这样在这个例子中的 map 操作不管 X 是否变化都会对它加 5.

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2.2. Some transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

表 2.2 Spark 中 RDD 的一些 transform 操作和 action 操作。Seq[T] 表示 T 类型的元素序列。

2.3.1 Spark 中 RDD 的操作

表 2.2 列出了 Spark 中 RDD 一些主要的 transform 操作和 action 操作。我们给出了每个操作的方法签名，方括号中显示的是类型参数。我们可以将 *transformations* 操作理解成一种惰性操作，它只是定义了一个新的 RDD，而不是立即计算它。相反，*actions* 操作则是立即计算，并返回结果给程序，或者将结果写入到外存储中。

请注意某些操作，例如 *join* 只适合键值对类型的 RDDs。此外，函数名的选择符合 Scala 等函数式语言的 API 命名规范。例如 *map* 表示一对一的映射，而 *flatMap* 则表示一对多的映射（类似于 MapReduce 中的 map）

除了这些操作，用户还可以持久化(persist)一个 RDD。此外，用户可以得到一个 RDD 的划分顺序，它由 Partitioner 类表示，并且根据它划分另一个数据集。一些操作例如 *groupByKey*, *reduceByKey* 以及 *sort*，会自动产生一个基于哈希或者范围划分的 RDD。

2.3.2应用示例

这里，我们对 §2.2.2 节的数据挖掘应用示例补充了两个迭代型应用的例子：logistic regression(逻辑回归)和 PageRank。后者还展示了如何控制 RDDs 的划分来提升性能。

Logistic Regression(逻辑回归)

很多机器学习算法本质上是迭代型的，因为它们要运行迭代式的优化算法，比如采用梯度下降法最大化目标函数。因此，通过将数据缓存在在内存中可以加速运行。

作为一个例子，下面的程序实现了 logistic regression[53]，它是一种常用的分类算法，目的是找到一个超平面 w，以最好地将两个集合点分开（*比如*, 分类垃圾邮件和非垃圾邮件）。该算法使用梯度下降法：首先对 w 取一个随机值，然后在每一步迭代中，在数据集上计算 w 函

数的和，然后沿着梯度方向移动 w 来改进它。

```
val points = spark.textFile(...).map(parsePoint).persist()
var w =  // random initial vector

for (i < 1 to ITERATIONS) {
  val gradient = points.map { p =>
    p.x * (1/ (1+ exp(-p.y * (w dot p.x))) - 1) * p.y
  }.reduce((a,b) => a + b)
  w -= gradient
}
```

一开始，我们定义了一个持久化的 RDD 称之为 **points** ，它是通过对 text 文件做 *map* 转换（对每一行文本解析得到一个 Point 对象）得到的结果。然后我们在每一步都重复的对 **points** 执行 *map* 操作 和 *reduce* 操作来计算梯度，梯度是通过对当前 w 的函数求和得到。

如 2.6.1.节所述，在每一步迭代中，通过将 **points** 缓存在内存中能够获得 20 多倍的速度提升。

PageRank

在 PageRank 中有一个更复杂的数据共享模式[21]。该算法迭代地对每篇文档更新 *rank* 值：通过对链接到该文档的其它文档的贡献值求和。在每一次迭代中，每个文档发送一个贡献值 r/n 到其邻近结点，其中 r 表示该文档的 rank，n 为其邻居节点数。然后文档更新其 rank 值为：$\alpha/N + (1-\alpha)\sum c_i$，这里的求和是对所有接收到的贡献值求和，$N$ 表示总的文档数，a 是一个平滑参数。我们用 Spark 实现的 PageRank 代码如下：

```
//Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()
var ranks =  // RDD of (URL, rank) pairs

for (i < 1 to ITERATIONS) {
// 用每个也页面发送过来的贡献值来创建一个(targetURL, float)对的 RDD
val contribs = links.join(ranks).flatMap { case (url, (links, rank))=> links.map(dest => (dest, rank/links.size))
}
```

// 根据 URL 对贡献值求和从而获取新的排名

ranks = contribs.reduceByKey((x,y) => x+y).mapValues(sum => a/N + (1-a)*sum)_

}

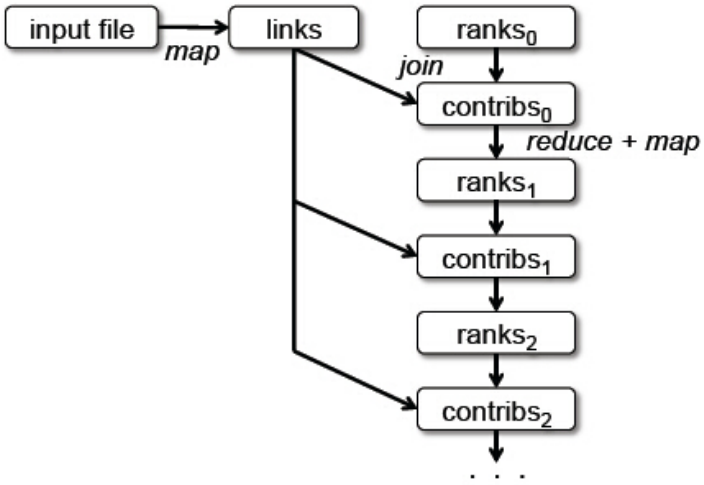


图 2.3 PageRank 中数据集的 lineage

该程序生成的RDD lineage如图 2.3 所示。在每一步迭代中，我们基于contribs和上一步迭代的ranks，以及静态的links数据集 [6]建立了一个新的ranks数据集。一个有趣特点是lineage图会随着迭代次数变长。因此，在一个有多次迭代的作业中，可能需要复制 **ranks** 的某几个版本以减少故障恢复的时间。 [66].用户能够调用带*RELIABLE* 参数的*persist*接口来做到这点。然而，需要注意**links** 数据集*不*需要复制，因为它的分片可以通过对输入文件块执行*map*操作来重建。links的数据集通常比ranks大很多，因为每个文档有很多链接，但是只有一个rank数值，使用lineage来恢复它会比对程序的内存状态做checkpoint要节省时间。

最后，通过控制 RDD 的*划分策略* ，我们能够优化 PageRank 中的通信开销。如果我们对 links 采用了某种划分策略*（比如，*在所有节点上对 link 列表进行 hash 分片），
我们可以对 **ranks** 用同样的方式分片，保证 links 和 ranks 的 *join* 操作不需要通信（因为每个 URL 的 rank 和 link 列表会在相同的机器上）。我们也能够写一个定制的 Partitioner 类将相互链接的页面分在一组*(比如，根据域名对 URL 进行*分片）。
这两种优化都能在我们定义 links 是调用 partitionBy 来实现：

```
links = spark.textFile(...).map(...)
                .partitionBy(myPartFunc).persist()
```

[6]注意，尽管 RDDs 是不可变的，程序中的变量 ranks 以及 contribs 在每轮迭代中都指向不同的 RDDs

经过这个初始化后，links 和 ranks 的 *join* 操作将自动将每个 URL 的贡献值聚合到 link lists 所在的节点上，计算新的 rank 值并和它的 links 做 join 操作。这种迭代间的一致性划分策略是一些特定框架的主要优化方法，例如 Pregel。RDDs 允许用户直接实现这个目标。

操作	含义
partitions()	返回分片对象列表
preferredLocations(p)	根据数据的本地特性，列出分片 p 能够快速访问的节点。
dependencies()	返回依赖列表
iterator(p, parentIters)	给定 p 的父分片的迭代器，计算分片 p 的元素
partitioner()	返回说明 RDD 是否是 hash 或 range 分片的元数据

表 2.3 Spark 中用于表示 RDDs 的接口

2.4 抽象 RDDs

抽象 RDDs 的一个挑战是如何在经过一系列 transform 操作后追踪其继承关系。理想情况下，一个实现了 RDD 的系统必须尽可能多地提供各种变换操作，（*如*表 2.2 中所示），并允许用户随意进行组合。我们提出了一种基于图的方式来抽象 RDD，它可以实现上述目标。我们已经在 Spark 中使用了这种表现形式来提供各种 transform 操作，而无需为每个 transform 操作的调度增加额外的逻辑。这极大度简化了系统的设计。

简而言之，我们提供了一个通用接口来抽象每个 RDD，并提供 5 种信息：一组*分区*，他们是数据集的最小分片；一组 *依赖关系*，指向其父 RDD；一个函数，基于父 RDD 进行计算；以及划分策略和数据位置的元数据。例如：一个表现 HDFS 文件的 RDD 将文件的每个文件块表示为一个分区，并且知道每个文件块的位置信息。同时，对 RDD 进行 *map* 操作后具有相同的划分。当计算其元素时，将 map 函数应用于父 RDD 的数据。我们在表 2.3 中总结了这个接口。

在设计接口的过程中，最有趣的问题在于如何表示 RDD 之间的依赖关系。我们发现，比较合理的方式是将依赖关系分成两类：*窄*依赖：每个父 RDD 的分区都至多被一个子 RDD 的分区使用；*宽*依赖：多个子 RDD 的分区依赖一个父 RDD 的分区。例如，*map* 操作是一种窄依赖，而 *join* 操作是一种宽依赖（除非父 RDD 已经基于 Hash 策略被划分过了）。图 2.4 中展示了一些其他例

子。

这两种依赖的的区别从两个方面来说比较有用。首先，窄依赖允许在单个集群节点上流水线式执行，这个节点可以计算所有父级分区。例如，可以逐个元素地依次执行 *filter* 操作和 *map* 操作。相反，宽依赖需要所有的父 RDD 数据可用并且数据已经通过类 MapReduce 的操作 shuffle 完成。其次，在窄依赖中，节点失败后的恢复更加高效。因为只有丢失的父级分区需要重新计算，并且这些丢失的父级分区可以并行地在不同节点上重新计算。与此相反，在宽依赖的继承关系中，单个失败的节点可能导致一个 RDD 的所有先祖 RDD 中的一些分区丢失，导致计算的重新执行。

RDD 的这种通用接口使得在 Spark 中使用不到 20 行的代码来实现大多数 transform 操作。事实上，即使是 Spark 的新用户也能实现新的 transform 操作 *（如：抽样和各种类型的 join）* 而不必了解调度细节。下面是一些 RDD 实现的概略图。

HDFS 文件：在我们的例子中，HDFS 文件作为输入 RDD。对于这些 RDD，*partitions* 代表文件中每个文件块的分区（包含文件块在每个分区对象中的偏移量），*preferredLocations* 表示文件块所在的节点，而 *iterator* 读取这些文件块。

map：在任何一个 RDD 上调用 map 操作将返回一个 MappedRDD 对象。这个对象与其父对象具有相同的分区以及首选地点（preferredLocations），但在其迭代方法（iterator）中，传递给 map 的函数会应用到父对象记录。

union：在两个RDD上调用*union*操作将返回一个RDD，这个RDD的分区为原始两个RDD的父RDD的分区进行union后的结果。每个子分区都是通过窄依赖于同一个父级分区计算出来的。[7]

sample：抽样类似于映射。不同之处在于，RDD 会为每一个分区保存一个生成随机数的种子来对确定如何对父级记录进行抽样。

join：连接两个 RDD 可能会产生两个窄依赖，或两个宽依赖，或一个窄依赖和一个宽依赖。如果两个 RDD 都是基于相同的 Hash/范围划分策略，那么就会产生窄依赖；如果一个父 RDD 具有某种划分策略而另一个不具有，则会同时产生窄依赖和宽依赖。无论哪种情况，结果 RDD 都具有一个划分策略（要么继承自父 RDD，要么是一个默认的 Hash 划分策略）。

需要注意的是 union 操作不会丢弃那些重复的值。

2.5 实现

我们用约 3 万 4 千行 Scala 代码实现了 Spark。该系统可运行于多种集群管理器之上，包括 Apache Mesos [56]，Hadoop YARN [109]，Amazon EC2 [4]，以及其内置的集群管理器。

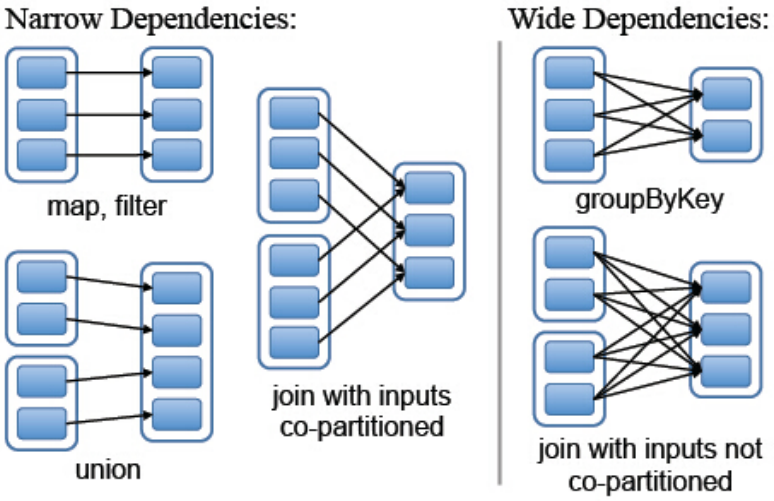


图 2.4，宽依赖和窄依赖的样例。每一个方框表示一个 RDD，其内的阴影矩形表示 RDD 的分区。

每一个 Spark 程序都以一个独立的应用在集群上运行，它有它自己的驱动节点(主节点,Master)和工作节点(Workers)。各个应用之间的资源共享则通过集群管理器来控制。

Spark 可以从任何 Hadoop 的输入源（例如使用 Hadoop 的 HDFS 和 HBase）中使用 Hadoop 的现有输入插件 APIs 读取数据，并且在未更改的 Scala 版本上运行。

现在我们描述了几种在系统中有趣的技术：我们的作业调度程序，多用户支持，Spark 解析器的交互式使用，内存管理，并且检查点支持。

2.5.1 作业调度

Spark 的调度器针对我们在 2.4 节所描述的 RDD。

总的来说，我们的调度器与 Dryad 的[61]类似，但它额外会考虑被持久化(persist)的 RDD 的那个分区保存在内存中并可供使用。当用户对一个 RDD 执行 Action*(如 count* 或 *save)*操作时， 调度器会根据该 RDD 的 lineage，来构建一个由若干 *阶段(stage)* 组成的一个 DAG(有向无环图)以执行程序，正如 2.5 所示。 每个 stage 都包含尽可能多的连续的窄依赖型转换。各个阶段之间的分界则是宽依赖所需的 shuffle 操作，或者是 DAG 中一个经由该分区能更快到达父 RDD 的已计算分区。之后，调度器运行多个任务来计算各个阶段所缺失的分区，直到最终得

出目标 RDD。

调度器向各机器的任务分配采用延时调度机制[117]并根据数据存储位置(本地性)来确定。若一个任务需要处理的某个分区刚好存储在某个节点的内存中，则该任务会分配给那个节点。否则，如果一个任务处理的某个分区，该分区含有的 RDD 提供较佳的位置(例如，一个 HDFS 文件)，我们把该任务分配到这些位置。

"对应宽依赖类的操作 *{比如〃* shuffle 依赖)，我们会将中间记录物理化到保存父分区的节点上。这和 MapReduce 物化 Map 的输出类似，能简化数据的故障恢复过程。"

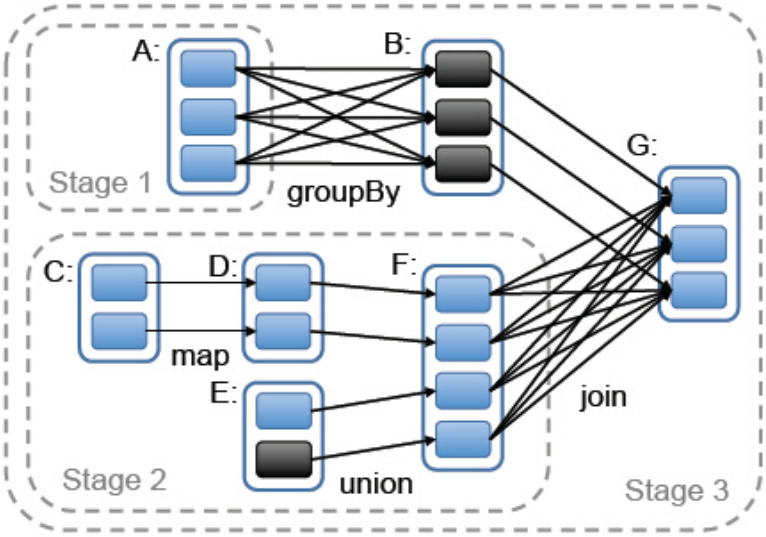


图 2.5.Spark 如何计算 job 的 stage 的例子。实线圆角方框标识的是 RDD。阴影背景的矩形是分区，若已存于内存中则用黑色背景标识。RDD G 上一个 Action 的执行将会以宽依赖为分区来构建各个 stage，对各 stage 内部的窄依赖则前后连接构成流水线。在本例中，stage 1 的输出已经存在 RAM 中，所以直接执行 stage 2 ，然后 stage 3。

对于执行失败的任务，只要它对应 stage 的父类信息仍然可用，它便会在其他节点上重新执行。如果某些 stage 变为不可用*(例如，*因为 shuffle 在 map 阶段的某个输出丢失了)，则重新提交相应的任务以并行计算丢失的分区。

针对调度器器自身失败的容错，拷贝相应 RDD 的 lineage 是比较直接的解决之道。但现阶段我们并不提供该类容错特性。

若某个任务执行缓慢 *(即"*落后者"straggler)，系统则会在其他节点上执行该任务的拷贝

--这与 MapReduce 做法类似，并取最先得到的结果作为最终的结果。

最后，虽然目前在 Spark 中所有的计算都是为了对驱动程序中调用动作的响应而执行，我们也试验让集群上的任务（*如*映射）调用*查找*操作，它根据键值能够随机访问散列分区的 RDDs 的元素。在这种设计下，如果任务所需要的分区丢失了，则该任务需要告知调用器去重新计算该分区。

2.5.2多用户管理

RDD 模型将计算分解为多个相互独立的细粒度任务，这使得它在多用户集群上能支持多种资源共享算法。特别地，每个 RDD 应用可以在执行过程中动态增长，并且可以轮询访问每台设备，或者可以被高优先级的应用占用。Spark 应用中大多数的任务的执行周期在 50 毫秒到数秒之间，这使得共享请求能得到快速响应。

虽然多用户共享算法并非本论文的主题，但如下我们列出了所支持的那些具体算法，以给读者一个感性的认识。

- 在每个应用程序中，Spark 允许多线程同时提交作业，并通过一种等级公平调用器来实现多个作业对集群资源的共享。这种调用器和 Hadoop Fair Scheduler [117]类似。 此特性主要用于创建基于针对相同内存数据的多用户应用，例如：Shark SQL 引擎有一个服务模式支持多用户并行运行查询。公平共享确保作业彼此分离，同时短的作业能在即使长作业占满集群资源的情况下也可尽早完成。

- Spark 的公平调度也使用延迟调度[117]，通过轮询每台机器的数据，在保持公平的情况下给予作业高的数据本地性。在本章几乎所有的试验中，内存本地化访问(Memory Locality)为 100%。Spark 支持多级本地化访问策略(本地性)，包括内存、磁盘和机架，以降低在一个集群里不同方式下的数据访问的代价。

- 由于任务相互独立，调度器还支持取消作业来为高优先级的作业腾出资源。[62].

- 纵观 Spark 的应用，Spark 仍然使用 Mesos 中资源提供的概念来支持细粒度共享，[56]它让不同的应用使用相同的 API 发起细粒度的任务请求。这使得 Spark 应用能相互之间或在不同的计算框架（例如 Hadoop）之间实现资源的动态共享。.延迟调度仍然能够在资源提供模型中提供数据本地性。

- 最后，Spark 使用 Sparrow *系统扩展支持分布式调度*[83].。该系统允许多个 Spark 应用

以去中心化的方式在同一集群上排队工作，同时提供数据本地性、低延迟和公平性。通过以去主节点方式来进行多任务提交时，分布式调度可极大地提升系统的可扩展性。

在现实环境中，绝大多数集群针对多用户环境设计，而且各个应用之间的交互程度也在增加。这样，相较于传统静态分区的集群，上述特性使得 Spark 的性能能得到显著的提升。

2.5.3 解析器集成

与 Ruby 和 Python 类似，Scala 也提供了一个交互式 Shell(解析器)。借助内存数据所带来的低延迟特性，我们希望让用户也能通过解析器来运行 Spark 并对大数据集进行交互式查询。

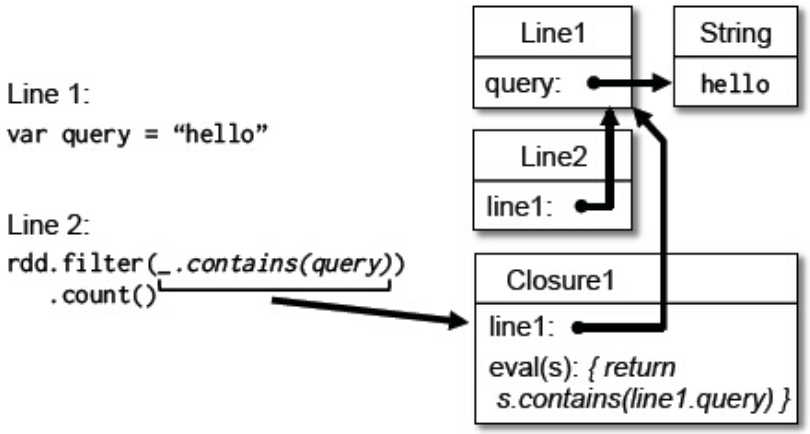


图 2.6.Spark 解析器将用户输入的多行命令解析为相应 Java 对象的示例。

Scala 解析器通常会为用户输入的每一行生成一个类，把它导入 JVM ，调用上面的一个函数。Scala 解析器的解析通常有如下组成：1. 将用户输入的每一行编译出其所对应的一个类；2. 将该类载入到 JVM 中；3. 调用该类的某个函数。这个类包含一个单例对象，对象中包含当前行的变量或函数，在初始化方法中包含运行该行的代码。例如，如果用户键入 var x = 5，换一行再键入 println(x)，那解析器会定义一个叫 Line1 的类，该类包含 x。第二行编译成 println(Line1.getInstance().x)。

Spark 中我们做了两个改变。

1. *类传输：*为了让工作节点能够从各行生成的类中获取到字节码，我们让解析器通过 HTTP 来为类提供服务。

为能让 Worker 节点能获取到各行对应类的字节码，我们让解析器通过 HTTP 来提供这些类。

*2. 代码生成器的改动：*通常，各种代码生成的单例对象是经由其相应类的一个静态方法来访问的。也就是说，当我们序列化一个引用了上一行中定义的变量的闭包（例如上面例子中的 Line1.x） 时，Java 不会通过检索对象树的方式去传输包含 x 的 Line1 实例。因此，工作节点不能得到 x。我们修改了代码生成器的逻辑，让各行对象的实例可以被直接引用。

图 2.6. 显示了我们修改之后解析器是如何把用户键入的每一行变成 Java 对象的。

我们发现，对于我们研究的部分成果和对 HDFS 上数据进行处理后构成的大量历史记录，Spark 解析器能用于对它们的处理。 我们还计划用它来交互式地执行更高级别的查询语言，*比如* SQL。

2.5.4内存管理

Spark提供了三种对持久化RDD的存储策略：未序列化Java对象存于内存中、序列化后的数据存于内存及磁盘存储。第一个选项的性能表现是最优秀的，因为可以直接访问在JAVA虚拟机内存里的RDD对象。在空间有限的情况下，第二种方式可以让用户采用比JAVA对象图更有效的内存组织方式，代价是降低了性能。[8]第三种策略适用于RDD太大难以存储在内存的情形，但每次重新计算该RDD会带来额外的资源开销。

对于有限可用内存，我们使用以 RDD 为对象的 LRU 回收算法来进行管理。当计算得到一个新的 RDD 分区，但却没有足够空间来存储它时，系统会从最近最少使用的 RDD 中回收其一个分区的空间。除非该 RDD 便是新分区对应的 RDD，这种情况下，Spark 会将旧的分区继续保留在内存，防止同一个 RDD 的分区被循环调入调出。这点很关键--因为大部分的操作会在一个 RDD 的所有分区上进行，那么很有可能已经存在内存中的分区将会被再次使用。到目前为止，这种默认的策略在我们所有的应用中都运行很好，当然我们也为用户提供了“持久化优先级”选项来控制 RDD 的存储。

最后，Spark 集群中的每一个实例都有其自己独立的内存空间。在后续的工作中，我们计划通过一个统一的内存管理器来实现多个 Spark 实例之间的 RDD 共享。Berkeley 正在进行的 Tachyon[68] 项目便是朝着这个目标。

整个的花销依赖于应用对每个字节的数据要做多少计算。对于一些轻量级的处理，这可能会道道 2 倍的计算量。

2.5.5检查点支持

虽然 lineage 可用于错误后 RDD 的恢复，但对于很长的 lineage 的 RDD 来说，这样的恢复耗时较长。由此，将某些 RDD 进行检查点操作(Checkpoint)保存到稳定存储上，是有帮助的。

通常情况下,对于包含宽依赖的长血统的 RDD 设置检查点操作是非常有用的,比如 PageRank 例子 (§2.3.2)中的排名数据集；在这种情况下，集群中某个节点的故障会使得从各个父 RDD 得出某些数据丢失，这时就需要完全重算[66]。相反，对于那些窄依赖于稳定存储上数据的 RDD 来说，对其进行检查点操作就不是有必要的。这样的 RDD 如 logistic 回归的例子(§2.3.2)和 PageRank 中的链接列表。

如果一个节点发生故障，RDD 在该节点中丢失的分区数据可以通过并行的方式从其他节点中重新计算出来，计算成本只是复制整个 RDD 的很小一部分。

Spark 当前提供了为 RDD 设置检查点(用一个 **REPLICATE 标志来持久化**)操作的 API,让用户自行决定需要为哪些数据设置检查点操作。但是，我们也正在对检查点操作自动化进行研究。因为调度器知道每个数据集的大小以及计算它的消耗的时间，那它应该可以选出所需 Checkpoint 的那些 RDD 以最小化系统恢复所需时间[114].。

最后，由于 RDD 的只读特性使得比常用的共享内存更容易做 checkpoint.由于不需要关心一致性的问题，RDD 的写出可在后台进行，而不需要程序暂停或进行分布式快照。

2.6 性能评估

我们通过在 Amazon EC2 上进行一系列实验和用户应用程序的基准测试对 Spark 和 RDDs 进行了性能评估。总体而言，我们的测试结果显示如下：

- 在迭代机器学习和图计算中，Spark 性能要比 Hadoop 模型好 80 倍这些性能提升来自于将数据以 java 对象存入内存从而减少系统 IO 和反序列化的开销
- 用户应用程序同样有很好的性能和扩展性。尤其，我们使用 Spark 来运行一个原本运行在 Hadoop 上的分析报告的应用，相较于性能 Hadoop 提升了 40 倍.。
- 当出现节点故障时，Spark 可以快速地恢复那些丢失的 RDD 分区。

- Spark 可以在 5-7 秒内交互式地查询 1TB 的数据。

我们首先使用迭代机器学习（§2.6.1)）和 PageRank 算法（§2.6.2）这类基准测试与 Hadoop 进行了对比。然后评估了在 spark 的容错性(§2.6.3)和数据不能完全存入内存(§2.6.4)时的状况。最后，我们讨论了 spark 在交互式数据挖掘（§2.6.5）和一些真实用户应用程序（§2.6.6）的表现。

除非另有说明，我们在测试中使用 **ml.xlarge** EC2 节点，配有 4 核 CPU 和 15 GB 的内存。我们使用 HDFS 来存储数据，每个文件 block 是 256MB.每次测试之前，我们都会清除操作系统缓冲区，从而得到更精确的 I/O 开销。

2.6.1 迭代式机器学习应用

我们实现了两种迭代式机器学习应用，逻辑回归和 k-均值算法（k-means），同以下系统进行性能对比：

- *Hadoop：* Hadoop 0.20.2 稳定版
- *HadoopBinMem：*一种 Hadoop，在首轮迭代中将输入数据转换成为开销较低的二进制格式，从而削减了后续迭代中文本解析的开销，并将数据存储在基于内存的 HDFS 中。
- *Spark：* 我们的 RDDs 的实现版本。

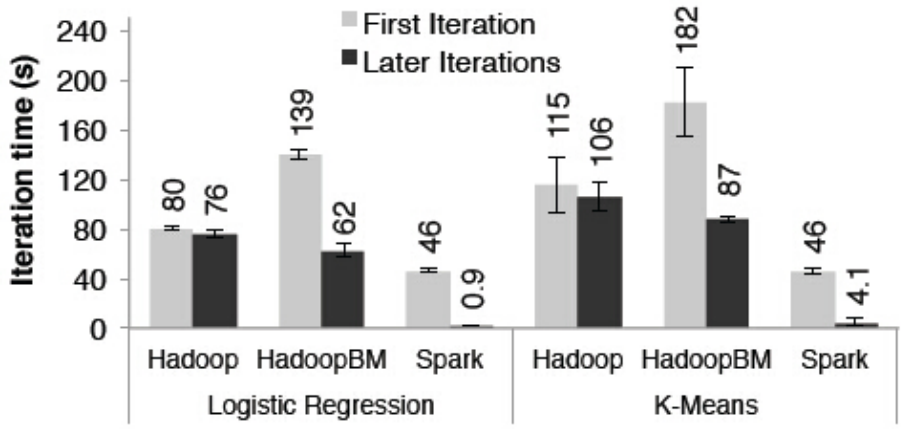


图 2.7 在 Hadoop 的迭代期间,HadoopBinMem 和 Spark 的逻辑回归和 K-means 都运行在一个 100 节点集群上处理 100GB 数据。

我们使用 25-100 台机器来运行两种算法，在 100GB 数据集上迭代 10 次。两个应用的关键区别在于对每个字节数据的计算量不同。K-means 的迭代时间取决于计算量，逻辑回归是非计算密集型的，对反序列化和 I/O 开销更加敏感。

由于典型的机器学习算法需要数十轮迭代直到收敛，我们分别统计了首轮迭代和后续迭代的耗时。我们发现通过 RDDs 共享数据极大地加快了后续迭代的速度。

首轮迭代： 在首轮迭代过程中，所有 3 个系统都是从 HDFS 中读取文本数据作为输入数据。如图 2.7 中的浅色条所示，整个实验中 Spark 都要比 Hadoop 快一些。主要是因为 Hadoop 中的 Master 和 Slave 之间基于心跳协议的信令开销。HadoopBinMem 是最慢的，因为它运行了一个额外的 MapReduce 作业来将数据转换成二进制格式，并且它需要通过网络传输将数据在内存式 HDFS 中进行备份。

后续迭代：图 *2.7* 显示了后续迭代的平均耗时。对于逻辑回归，在 100 台机器上运行，Spark 分别比 Hadoop 和 HadoopBinMem 快 85 和 70 倍。对于计算密集型的 K-means，Spark 仍然分别比 Hadoop 和 HadoopBinMem 快 26 和 21 倍。注意在所有这些案例中，程序通过同样的算法计算出了同样的结果。

理解速度提升： 我们非常惊奇地发现，Spark 甚至胜过了基于内存存储二进制数据的 Hadoop（HadoopBinMem）高达 85 倍之多。

在 HadoopBinMem 中，我们使用 Hadoop 的标准二进制格式（序列文件）和 256MB 文件块，并且我们强制使 HDSF 的数据直接加载到内存文件系统中。然而，Hadoop 仍然运行的比较慢，是由于以下几个原因：

1. Hadoop 软件栈的最低开销

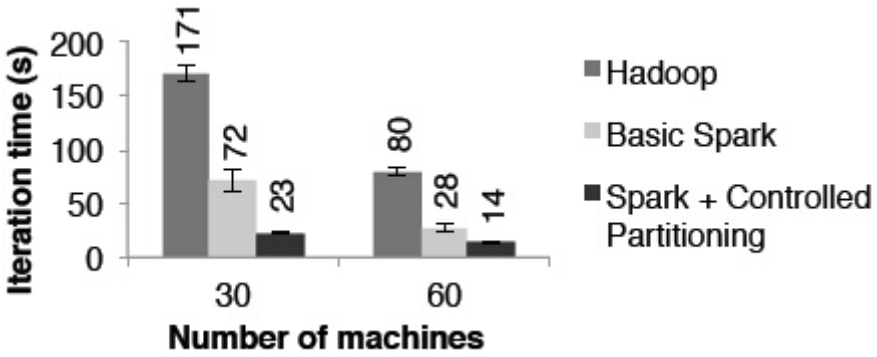


图 2.8 用 Hadoop 和 Spark 执行 PageRank 的性能

2. HDFS 读取数据的开销

3. 将二进制记录转反序列化为可用的在内存中的 Java 对象的代价

为了确认这些因素，我们执行单独的微基准测试。例如，为了测试 Hadoop 的启动开销，我们运行空操作的 Hadoop 作业，观察到仅仅完成作业的最小需求：设置、启动任务、清理工作就至少耗时 25 秒。至于 HDFS 的开销，我们发现为了维护每一个数据块，HDFS 进行了多次内存复制以及校验和计算。最后，我们发现，即使是在内存中的二进制数据，通过 Hadoop 的 SequenceFileInPutFormat 读取数据来反序列化这一步，也比逻辑回归的计算花费了更多时间，这解释了为什么 HadoopBinMem 更慢。

2.6.2 PageRank

我们对一个 54GB 的维基百科数据进行 PageRank 来比较 Spark 和 Hadoop 的性能。我们对 PageRank 算法迭代 10 次，处理一个大约有 400 万词条的连接图由图 2.8 可见在 30 节点的集群上，基于内存存储的 Spark 相较于 hadoop 获得了 2.4 倍的性能提升。此外如 2.3.2 章节所述，控制 RDDs 的分区划分使其在每次迭代中保持一致可以将性能提升到 7.4 倍。同样，当节点扩展到 60 个时，性能也保持着一种线性增长。

我们也评测了一种基于 Spark Pregel 实现的 PageRank（将在 2.7.1 小节描述）迭代次数和图 2.8 类似，但大约长了 4 秒钟。这是因为每轮迭代中，Pregel 都需要额外的运算让顶点“投票”决定是否结束作业。

2.6.3 故障恢复

我们评估了在 K-Means 作业中当一个节点出现故障后 Spark 通过 RDD 的继承关系重建分区的开销。图 2.9 显示了在一个 75 节点集群正常运行场景下，对 K-Means 算法迭代 10 次的运行时间。在这些节点中，有个节点会在第六轮迭代的开始出现故障。

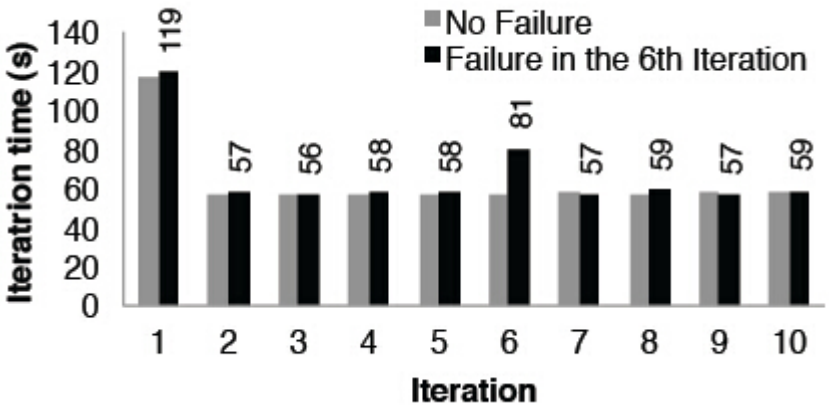


图 2.9 出现一次故障时 K-Means 的迭代时间。由于一台机器在第六次迭代开始就被终止了，这就导致 RDD 会通过其继承关系来进行部分重建。

在正常操作情况下，75 节点群集的 k-means10 次迭代过程中，有一个节点在第六次迭代的开始失败。当没有任何故障时，每轮迭代包含 400 个运行在 100GB 数据上的任务。

直到第 5 次迭代结束，迭代时间约为 58 秒。在第 6 次迭代，一台机器被杀掉，导致了该机器丢失了运行其上的任务和缓存的 RDD 分区。Spark 将并发地在其他机器上重新执行这些任务。这些任务重新读取相应的输入数据并根据继承关系重建 RDDs，导致迭代时间增长到 80 秒。一旦丢失的 RDD 分区被重建完成，迭代时间将会降回至 58 秒。

注意，基于设置检查点的故障恢复机制，恢复可能会需要重新运行几轮迭代，而迭代的次数取决于设置检查点的频率。此外，该系统可能需要在整个网络中备份 100GB 工作集（文本数据被转换成二进制数据），这将会 1.要么消耗两倍内存将数据备份在内存中，2.要么等待直到将数据集写入到磁盘中。相反，我们例子中 RDDs lineage 图的数据量小于 10 KB。

2.6.4 内存不足的情况

目前为止，我们集群中使用的机器都有足够的内存来缓存迭代过程中所有的 RDDs。一个很自然的问题是 Spark 如何在没有足够内存存储的的情况下运行。在这个实验中，我们配置 Spark 集群只能使用节点的一定比例的内存来缓存 RDDs。我们在图 2.10 中展示了逻辑回归算法在使用不同内存比例情况下的表现。我们发现，当使用较少内存空间时性能急剧下降。

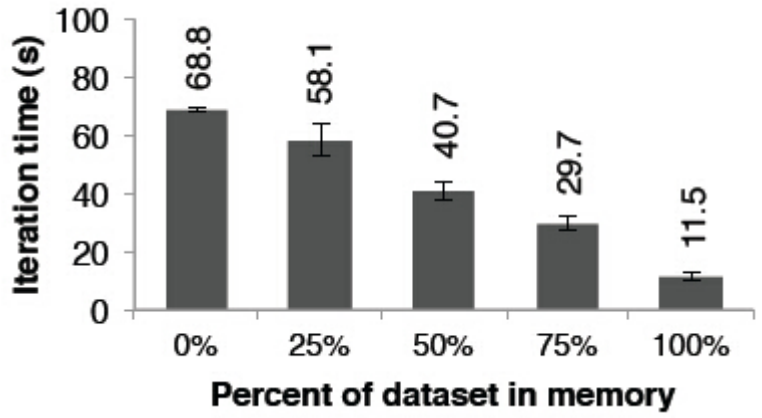


图 2.10 缓存不同数量数据在内存中时逻辑回归算法的性能（100GB 数据以及 25 个节点）

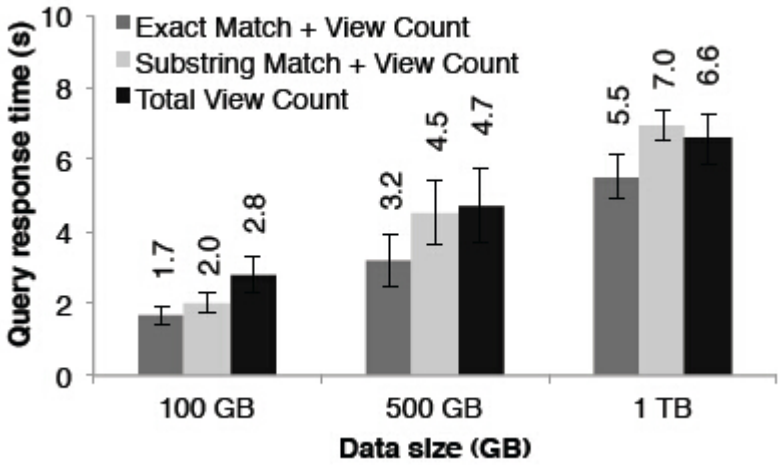


图 2.11 在 Spark 上交互式查询的响应时间，在 100 台机器的集群上扫描逐步增大数据量。

2.6.5 交互式数据挖掘

为了说明 Spark[7]可以对大数据集进行交互式查询的能力，我们用它来分析 1TB 的维基百科页面日志（2 年数据）。在这个实验中，我们使用了 100 个 m2.4xlarge EC2 节点，配有 8 个 CPU 核和 68GB 内存。我们通过查询来找出整个视图（View），这些视图包括（1）所有页面，（2）标题与指定关键字完全匹配的页面以及（3）标题与指定关键词部分匹配的页面。每个查询都会扫描所有的输入数据。

图 2.11 给出了在完整数据集，50%数据集，和 10%数据集上查询操作对应的响应时间，即使在 1TB 数据上，Spark 查询操作也仅花了 5-7 秒。相比于对磁盘数据进行查询，这提升了不止一个数量级；例如，从磁盘上查询 1TB 的文件需要花费 170 秒。这证明了 RDDs 可以使得 Spark

成为一个有力的交互式数据挖掘工具。

2.6.6 实际应用

内存分析：一家视频分发公司，Conviva Inc，用 Spark 替代了 Hadoop 来加速一些数据分析报告应用。

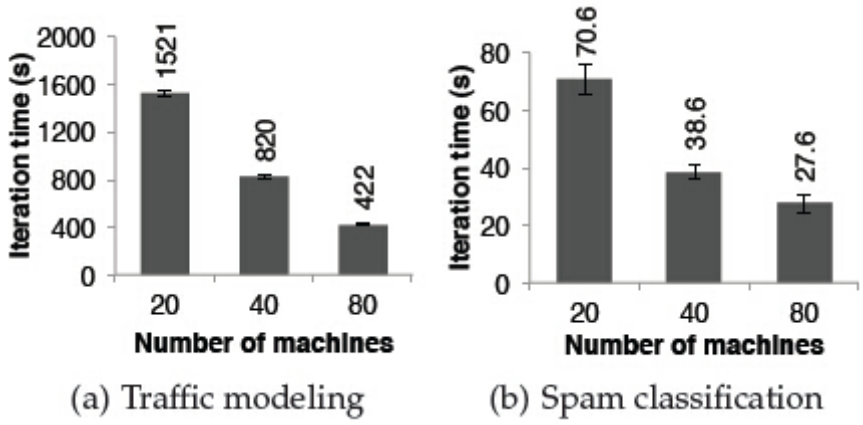


图 2.12 两个 Spark 应用的每轮迭代时间。误差表示为标准差

例如，其中一个报告应用进行一系列 Hive 查询，为消费者计算各种统计数据。这些查询都是在相同的数据子集（那些匹配用户提供的过滤条件的记录）上进行的，但是对不同的组域进行聚集（aggregation）操作（求平均，统计百分比和 统计非重复值），这些操作都需要独立的 MapReduce 作业来完成。通过在 Spark 上实现查询，并一次性加载数据子到 RDD 中，公司报告生成速度可以提高达 40 倍。使用 Hadoop 集群的话，处理 200G 的压缩数据要花费 20 个小时，而 Spark 只需要 2 台机器在 30 分钟内完成。另外，Spark 只需要 96G 的内存，因为它只需要将符合用户过滤规则的行和列存储在 RDD 中，而不是 整个解压缩文件。

交通建模： 伯克利大学的 *Mobile Millennium* 项目研究人员 并行化了一个基于汽车 GPS 数据推断道路拥堵情况的学习算法。数据源是城市中 10000 条道路交通网，以及 60 万个装有 GPS 设备的汽车点到点的运行样本（每个运行样本可能包括多条道路）。基于一种交通模型，该系统可以估算出经过某条特定线路需要花费的时间。研究人员使用期望最大化（EM）算法来训练这个模型，该算法会迭代两次 *map* 和 *reduceByKey* 操作.该应用测试了从 20 节点扩展到 80 个节点，每个节点配置 4 个 CPU 核，获得了接近线性的性能提升，如图 2.12(a)所示。

Twitter 垃圾分类： 伯克利大学的 Monarch 项目使用 Spark 来识别 twitter 消息中的垃圾信息。他们在 spark 上使用了类似于 2.6.1 中的逻辑回归分类器，但是他们使用了分布式的 *reduceByKey* 操作来并行地计算梯度向量总和。在图 2.12(b)中，我们展示了基于 50G 数据子集分类器训练的性能扩展结果：这些数据包括 25 万个网址以及和这些地址页面相关的千万级别的网络和内容的特性和维度。由于每轮迭代中有较高的固定网络开销，性能并没有获得线性的增长。

2.7 讨论

虽然 RDD 的不可变性质和粗粒度变换特质，使得其编程接口看上去能力有限，但实际中我们发现它们能适应的应用种类广泛。具体来说，RDD 可以表达大量的集群编程模型，而这些模型之前都是针对独立框架而提出。从而，RDD 使得用户 可以在一个程序中*组合*这些模型 *(比如,* 先运行一个 MapReduce 操作来构建一个图，而后对该图调用 Pregel)，并在它们之间共享数据。在本节中，我们将讨论 RDD 可以表达哪些编程模型，以及为什么它们被广泛应用(§2.7.1).另外，我们还将讨论 RDD 中 lineage 信息的另一个好处--它使得在这些模型之间的调试变得容易。(§2.7.3).

2.7.1 对现有编程模型的表达

RDD 可以*高效*的表现一些此前相对独立的集群编程模型。所谓“高效”，是指 RDD 不仅能得到与它们相同的输出结果，而且还囊括了对这些框架所进行的 *优化* ，比如将特定的数据保持在内存中、数据分区优化以降低网络通讯和高效率的故障恢复。可以用 RDD 表达的模型包括：

MapReduce： 这个模型可以由 SPARK 中的 *flatMap* 与 *groupByKey* 操作进行表达，而如果用到 combiner 时则可引入 *reduceByKey* 操作。

DryadLINQ： DryadLINQ 系统基于 Dryad 更通用的运行机制上，提供了比 MapReduce 更为丰富的操作。但这些操作都是批处理操作，且 Spark 中都有相应的 RDD 变换操作与之对应（如 *map*，*groupByKey*， *join* 等等）。

SQL： 与 DryadLINQ 的表达类似，SQL 查询对数据集的操作是基于数据并行的，故它们也可通过

RDD 变换而实现。在 第三章 ，我们会描述 Shark 组件，它是 SQL 在 RDD 上的一种高效实现。

Pregel：谷歌的 Pregel[72] 是一个专门针对迭代图型应用的模型。这种模型初看之下与其他系统面向集合的编程模型有很大的不同。在 Pregel 中，一个程序以一系列协调好的“superstep”运行。

在每一个 superstep 里，图内的各节点都通过执行一个用户定义的函数来实现对自身状态的更新和对图的拓扑结构的改变，并向其他节点发送包含它们在下一超步所需要的信息的消息。该模型可以表达许多图形算法，包括最短路径，二分匹配，和 PageRank。

注意，在 Pregel 的每次迭代中，它是将*相同的*用户自定义函数运用到所有节点上。 这是 RDD 可以表达 Pregel 模型的关键。具体来说，我们可以将各次迭代时的节点状态保存为一个 RDD，然后调用一个变换 *(flatMap)* 来执行用户自定义的函数，并生成上述消息所对应的 RDD。之后，通过将该 RDD 和节点状态的 RDD 进行 Jion 操作，便可实现消息的交换。同样值得注意的是，RDD 同时也能提供如 Pregel 那样将节点状态保存在内存中、控制节点分区策略来减少网络通讯以及出现故障时的部分恢复功能。我们在 Spark 上实现了一个 200-行的 Pregel 库，读者请参阅[118]

迭代式 MapReduce：近期所提出的系统，包括 HaLoop[22]和 Twister [37]， 提供了一种迭代式的 MapReduce 模型。在该模型下用户可以指定系统运行一系列的 MapReduce 任务。 这些系统可以保持每次迭代的数据分区一致性，其中 Twister 还可将讲数据保持在内存中。这些优化都可轻松地用 RDD 来表达，HaLoop 模型的实现对应一个 200 行左右的库。

2.7.2 解释 RDD 表达能力

为什么 RDD 能够表达这些不同的编程模型？原因就是 RDD 上的限制在许多并行应用程序中影响非常小。其原因在于，虽然 RDD 仅能通过批量变换来创建，但众多的并行程序本质上都是 *对多条记录执行相同的操作* ，而这点便使得它们易于表达。另外，RDD 的不变性也不会影响其表达，因为相同数据集的各个不同版本可以通过多个对应的 RDD 来表示。事实上，大多数当前的 MapReduce 应用所基于的文件系统，比如 HDFS，并不允许更新文件(译注:记录只能创建或删除，而不能修改)。在后续章节（3 和 5）中，我们会对 RDD 表达进行更为详细的阐述。

最后一个问题是，为什么之前的框架没有提供相同级别的通用性呢？我们认为，这是由于这些系统仅关注在 MapReduce 和 Dryad 所不擅长的特定问题上，比如迭代，而未能发现这些问

题的 *均是因为*缺乏对数据共享的抽象。

2.7.3 利用 RDD 来调试

RDD的最初始设计时能为容错进行确切的重算特性，该特性也方便了对其的调试。具体来说，通过记录在作业中创建的RDD的lineage，借助重算所依赖的的RDD分区，人们可以 (1)在后续中重建这些RDD，同时对其进行交互式查询，(2) 在一个单进程调试器中从该作业里运行任意一个任务。不同于传统的针对一般分布式系统的重放（replay）调试器[51]，需要对多个节点记录和推断出其事件的先后顺序，RDD只需要记录血谱图[9] 因此基本上不会引入任何记录开销。我们现在就是基于这些概念进行Spark调试器的开发工作[118]。

2.8 相关工作

集群编程模型 ：集群编程模型的相关工作可分为如下几类：首先，数据流模型例如 MapReduce [36]，Dryad [61]和CIEL [77]，他们都有丰富的操作集来处理数据，但通过稳定的存储系统来实现数据的共享。RDD则展现了一种比稳定存储更*高效*的数据共享抽象，因为它避免了数据备份、I/O操作与序列化的开销。[10]

其次，一些数据流系统的高级编程接口，如 DryadLINQ[115]与 FlumeJava[25]，提供了语言级集成的 API，可以让用户通过 *map* 与 *join* 等操作符处理“并行集合”。然而，在这些系统中，并行集合是指磁盘上的文件或者用于表达查询计划的临时数据集。在这些系统中，尽管对于同一个查询，数据可通过管道流转于各操作符之间*(例如，*一个 *map* 操作接另一个 *map 操作)，*但是*不同的*查询之间数据却不能有效共享。我们依据并行集合模型来构建 Spark 的 API，我们不会宣称我们创造了开发语言集成接口，但将 RDD 作为这个接口背后的存储机制，我们将可以支持更多类型的应用。

一些第三方系统为需要数据共享的*特殊*应用提供了高级接口。例如，Pregel[72]支持迭代图形应用，Twister[37]与 HaLoop[22]是迭代的 MapReduce 运行时。然而，这些框架只是隐性的为他们支持的计算模式进行了数据共享，并没有为用户提供一个通用的抽象，使得用户能够

[9] 与这些系统不同的是，一个 RDD-based 调试器不会重放用户函数中不确定的行为(如，一个不确定的 map 操作)，但它至少可以通过校验数据来报告。

[10] 注意，将 MapReduce/Dryad 在 RAMCloud 这样的内存数据存储中运行[81] ，仍然需要数据备份与序列化。这些开销对一些系统来说也是很大的，例如§2.6.1.中提到的。

利用这个抽象在他所选择的操作中共享数据。例如，一个用户不能用 Pregel 与 Twister 将数据先加载到内存中*再*决定如何对其进行处理。RDD 显示地提供了一个分布式的存储抽象层，从而支持这些特殊系统无法支持的应用，如交互式数据挖掘。

最后，一些系统暴露了共享的可变状态使得用户可以进行内存中的计算。例如，Piccolo[86]允许用户在分布式哈希表中运行一些并行读以及并行更新的函数。分布式共享内存（DSM）系统[79]和键值对存储系统如 RAMCloud[81]提供了一个类似的模型。RDD 与这些系统的区别有如下两点：首先，RDD 提供了更高级别的编程接口，这些接口基于操作符如 *map,sort* 与 *join*，而 Piccolo 与 DSM 的接口只能读和更新表的元素。第二，Piccolo 与 DSM 系统通过检查点与回滚实现恢复机制，这在很多应用中比 RDD 的数据溯源策略更为昂贵。最后，如 2.2.3 中讨论的那样，RDD 相对于 DSM 还具有其他的优势，如减轻 straggler(慢节点)的情况。

Caching Systems:Nectar [50]可以通过使用程序分析[55]定位公共子表达式的方式复用 DrayLINQ 作业之间的中间结果。如果加入到以 RDD 为基础的系统中，这种功能将会变得非常引人注目。然而 Nectar 并不提供内存缓存（他将数据放入了一个分布式文件系统），而且他也不能让用户显示地去控制哪些数据集需要常驻内存以及如何分割数据集。CIEL[77] 和 FlumeJava[25]同样提供任务结果的缓存，但不提供内存缓存或显式的控制哪些数据需要被缓存。

Ananthanarayanan 等提出了增加内存缓存到分布式文件系统来充分利用数据访问的时间和空间局部性[6]。虽然这种解决方案可以更快速地访问那些已经在文件系统中的数据， 但是相对于 RDD 在一个应用运行过程中将*中间*结果进行共享这种方式，还不够高效， 因为它仍然需要在 stage 与 stage 之间将这些结果写到文件系统中去。

Lineage：获取数据的源头信息一直以来都是科学计算与数据库技术的热门研究课题。特别是针对一些应用比如解释结果、可以允许其他人进行重现以及在工作流中发现 bug 或者数据丢失的情况下对数据重新计算。我们建议读者参考[20] 和 [28]以获得更多的相关信息。RDDS 提供一种并行编程模型，在这种模型中细粒度的数据溯源可以轻松实现，这一特点可以用于故障恢复。

我们基于 lineage 的恢复机制与 MapReduce 和 Dryad 中的计算（作业）*中* 恢复机制类似，MapReduce 和 Dryad 中的恢复机制是跟踪 DAG 任务间的依赖关系。但是，这些系统中，所有 lineage 信息都在一个作业结束后就自动丢失，需要备份到存储系统中才能与*其他*计算作业共享。与此相反，RDD 在计算之间高效地将数据常驻内存来实现 lineage，通过这种方式省去了备份和磁盘 I/O 的开销。

关系型数据库 RDD 在概念上与数据库的视图类似，持久化的 RDD 与物化视图类似 [89].然而，就像 DSM 系统，数据库通常允许细粒度的读写访问所有记录，为了容错需要对所有操作与数据进行日志记录，并且需要额外的开销来保持一致性。这些开销粗粒度的 RDD 转化模型来说是不必要的。

2.9 总结

我们已经提出了弹性分布式数据集（RDDS），它是在集群应用中对共享数据的一种高效、通用和容错的抽象。

RDDS 可以表达各种各样的并行应用，包括许多已提出的专门用于迭代计算的的编程模型，以及这些模型不能涵盖的新的应用。不同于现有的集群存储抽象那样必须要进行数据备份来容错，RDDS 提供基于粗粒度的转换的 API,使用 lineage)的方式让数据恢复更加高效。我们在一个叫 Spark 的系统上实现了 RDDs，它在迭代应用中要比 Hadoop 快 80 倍，并且可以实现在数百 GB 的数据上进行交互式查询。

Spark 是目前托管在 Apache 软件基金会的一个开源项目，地址是 spark.incubator.apache.org。从项目诞生开始，已经被大量的开源社区所广泛使用和增强，截至在写本文之时，已经有 100 多名工程师给 Spark 贡献过代码。

虽然本章涵盖了一些可以很快在 Spark 上实现的现成的简单的编程模型，RDD 同样可以表达更加复杂的计算和很多模型（*例如,* 列式存储器）下比较流行的优化。接下来的两章会介绍这些模型。

第三章 基于 RDD 的模型

3.1 简介

尽管在之前的章节中已经介绍了一些简单的基于 RDD 的编程模型，例如 Pregel 和 MapReduce 的迭代计算。不过 RDD 的抽象模型可以用来实现更复杂的工作，包括专用引擎中关键的优化(*例*

如 列式存储的处理和索引)自 Spark 发布以来，我们和大家已经实现了如下的一些模型，如 SQL 引擎 Shark [113]，图计算系统 GraphX[112]，还有机器学习库 MLLib[96].我们将在这一章描述在这些系统中所实现的技术，会将 Shark 作为样例深入剖析。

简单回顾一下之前的章节，RDD 可以提供如下特性：

- 在一个集群中对于任意记录具有不变性的存储（在 Spark 中以 Java 对象的方式来表示）
- 通过每一条记录的 key 字段来控制数据分区
- 将粗粒度的操作用于分区的操作
- 利用内存存储的低延迟特性

接下来，我们将展示如何利用这些特性来实现更复杂的数据处理和存储。

3.2 一些在 RDDs 上实现其他模型的技术

在特定的引擎上，不仅仅优化了数据上的运算符，也优化了数据的存储格式和数据的访问方式。例如，像 Shark 这样的 SQL 引擎可能会按列来处理数据，但是像 GraphX 这样的图引擎按照索引来处理数据，使得效率表现很出色。下面我们讨论几个已经在 RDDs 上实现了的这些优化的常见方法，这些方法使得可以在享受 RDD 模型带来的容错等好处的同时，还能保持特定系统的性能。

3.2.1 RDDs 里的数据格式

虽然 RDD 存储的是简单、扁平的数据记录，但我们发现一个实现更丰富的存储格式的有效策略是通过在同一条 RDD 记录中存储*多个*数据项，并对每一条记录实施更加复杂的存储。用这样的方式批处理即使几千条记录所带来的效率就足以非常接近使用专门的数据结构，同时仍然保持了每个 RDD“记录”的大小为几兆字节。

例如，在分析型数据库中，一种常用的优化就是列式存储和压缩。在 Shark 系统中，我们在一条 Spark 记录中存储多条数据库记录并使用这些优化。对 10000 到 100000 条记录压缩的程度与将整个数据库存储成列格式的压缩程度非常接近。因此，这个设计使我们仍然获得显著的效果。举个比较高级的例子，GraphX [112]需要在数据集中进行非常稀疏的连接(JOIN)操作，这些数据集表示的是这个图中的顶点集合和边集合。它是这么操作的：在每个 RDD 的记录里存

储包含多个图记录的 *哈希表*，当和新的数据进行连接（JOIN）的时候，能够快速地查找到某个顶点和边。这点非常重要，因为在许多图算法中，最后几次迭代只包含几条边或几个点，但这些点和边仍然需要和整个数据集连接（JOIN）。同样的机制可以被用于更有效的来实现 Spark 的 Streaming 里的 *updateStateByKey* (Section 4.3.3).

RDD模型有两方面的因素使得这个方法非常有效。首先，RDD通常在内存中，因此对每个操作可以用*指针*来只读取整个“组合”记录中相关的部分。例如，一组用列表示的（整型，浮点型）记录可以用一个整型数组和浮点数组来实现。如果我们只想读取整数字段，我们可以根据指针找到第一个数组而不用在内存中扫描第二个数组。同样地，在上述哈希表中的RDD，我们也可以只查找所需要的记录。[11]

其次，一个很自然的问题便是，如果每个计算模型都有自己的批处理记录的表示方式，那么如何有效地把要处理的类型结合起来？幸运的是，RDD 底层接口是基于*迭代器*的（见 2.4 节中的*计算* 方法），这可以实现数据在不同格式中快速和流水线地转换。含有复合记录的 RDD 可以通过 *flatMap* 批量操作有效地在解压的记录上返回一个迭代器，并且这个迭代器可以被进一步地在解压后的记录上进行管道化的窄变换，或者被重新打包成另一种格式进行转换，从而使未提交的且未解压的数据量最小化。迭代器由于通常进行扫描操作，一般情况下是用于内存数据中的一个高效的接口。只要每个批记录能放在 CPU 缓存中，这使得数据能够快速地转换和转化。

3.2.2 数据分区

在特定模型中的第二个常见优化是在一个集群中用特定领域的方式对数据进行*划分*来提高应用程序的性能。例如，Pregel 和 HaLoop 使用了一个可能的用户自定义函数来划分数据，从而加快针对一个数据集的连接操作。并行数据库通常也提供了多种数据划分形式。

在 RDDs 里，对于每条记录都可以通过记录里的 key 值很容易地进行数据划分。（事实上，这是在 RDDs 拥有的一条记录元素的唯一标识。）注意到，即使 key 是对整个记录的，但含有多个潜在数据项的“复合”记录仍然可以有一个有意义的 key 值。例如，在 GraphX 中，每一个分区中的记录都有相同的散列码来取分区数的模，但是为了能够高效的查找仍然需要在内部进行散列。当系统使用复合记录来进行 shuffle 操作时，如 *groupBy*,它们可以将每个突出的记录散列为目标组合记录的 key 值。

最后一个有趣的划分用例是*复合*数据结构，数据结构的一些字段会随着时间被更新而另外

[11] 如果给磁盘记录添加一个 API 来提供类似的属性，就像数据库存储系统 API 一样，这将很有意思，只是我们还没做这个。

一些字段则不被更新。例如，在第 2.3.2 节的 PageRank 应用中，对每一个 page 有两块数据：一个不可变的 links 列表，和一个可变的 rank。我们用两个 RDDs 来表示，一个（ID,links）对和一个（ID，ranks）对，都是通过 ID 来进行划分。系统优先将 ranks 和 links 的每个 ID 记录都放置在相同机器上，但我们可以不需要改变 links 而分别对 ranks 进行更新。在 GraphX 中，类似的技术被用来保存点和边的状态。

一般情况下，用户可以认为 RDDs 是作为一个在集群环境里更为具体的内存抽象。当在单台机器上使用内存时，程序员主要为了优化查找和最大化提高常访问信息的集中式放置，而需要考虑数据的分布。RDDs 通过让用户选择一个划分函数和划分的数据集，来提供对分布式内存的控制，但 RDDs 避免了要求用户精确的指定每一个分区的*位置* 。因此，运行时系统可以基于可用资源对分区数据进行有效地放置，或在出现故障时对分区数据移动，而程序员仍然可以控制访问的性能。

3.2.3 关于不可变性

RDD 模型与大多数特定系统的第三个区别是 RDDs 是不可变的。不可变性对于 lineage 和错误恢复来说是很重要的，尽管它与为这些目的而具有可变的数据集和记录版本号没有本质上的不同。但是，可能有人会问这是否会导致性能低下。

虽然不可变性和容错性肯定会导致一些开销，但是我们发现这两项技术在很多情况下都能够表现出良好的性能。

1. 我们用多个 co-partitioned RDDs 来表示复合数据结构，就像上一节里提到的 PageRank 例子一样，只允许程序修改需要修改部分的状态。在很多算法中，尽管记录的其他字段在每次迭代中都会改变，但是有些字段是永远不变的，所以这种方式就可以取得了很好的性能。

2. 当内部的数据结构是不可变的时候，即使在一条记录中，我们也可以用指针来重复使用记录之前"版本"的状态。例如，在 Java 中字符串是不可变的，所以一个(Int, String)记录上的映射（*map*） 如果保持 String 不变，只改变 Int 值的话，我们只需要使用一个指向之前 String 对象指针，而不是去复制它。更笼统地说，在函数式编程中的*持久化数据结构*[64]可以用其他形式的数据上的增量更新(*例如，散列表*) 来表示之前版本的增量。令人很愉快的是，许多函数式编程中的想法可以直接帮助 RDDs。

在今后的工作中，我们将继续寻求其他方式来跟踪多个版本的状态，从而接近可变状态系统的性能。

3.2.4 实现自定义转换

最后，我们发现在一些应用中，使用低级的 RDD 接口实现自定义的依赖模式和转换是有用的（2.4 节）。该接口非常简单，实现他们仅仅需要依赖于父 RDDs 的列表和为 RDD 的分区从父 RDDs 给定的迭代器里进行迭代计算的一个函数。在 Shark 和 GraphX 的第一版中，我们实现了一些这样的自定义运算符，这导致了很多新的运算符也被加入到 Spark 中去。比如说，*mapPartitions*，是我们实现的一个有用的运算符，在给定一个 RDD[T]和一个计算迭代器函数（给定 Iterator[T]，计算出 Iterator[U]）的情况下，通过将这个函数作用到每一个分区上，最后能返回一个 RDD[U]。这是非常接近于 RDDs 最低级的接口，允许在每个分区里执行非功能性的操作(*例如*,使用可变状态)。在 Shark 的实现中同样包含 *join* 和 *groupBy* 的自定义版本，这是为了取代内置的相应运算符的工作。但是，请注意即使是实现了自定义转换的应用，这些应用依然能够自动地享受到 RDD 模型的容错、多租户和组合所带来的好处，并且使得开发将会比独立系统更加简单。

3.3 Shark:RDDs 上的 SQL

我们拿 Shark 系统作为在 RDDs 上实现高级的存储和处理技术的例子。Shark 在并行数据库的大量研究领域中表现良好，并且还提供容错能力和复杂分析的能力，而这正是传统数据库所不具备的。

3.3.1 动机

现在的数据分析面临着几个挑战。首先，数据量正在急剧增加，这也使得将计算任务分发到计算机集群中的不同机器上，然后这些机器并行执行任务成为一种需求。其次，这种分发增加了错误和 straggler（慢任务）的发生概率，并且使得并行数据库设计变得更加复杂。第三，现在数据分析的*复杂度*和以前已经不一样了：现在的数据分析采用了先进的统计分析方法，比如说机器学习算法,这种方法在汇总和分析能力上要远远超过传统企业所采取的数据仓库系统。最后，尽管数据的规模和复杂度在不断增加，用户却仍然希望查询能够以交互速度执行。

为了解决这个“大数据”问题，探索的方向分成两条主线。第一条主线，考虑到 MapReduce

[36]及其各种泛化版本[61, 27]提供了一个细粒度的适合大型集群的容错模型，在这种模型中，失败的或者运行很慢的节点上的任务最终都一定能在其他的节点上再执行。MapReduce 本身也非常泛化：它已经被证明能够表达许多统计和学习算法[31]。它同样也能支持非结构化的数据和"schema-on-read."但是，MapReduce 引擎缺少许多使数据库高效运行的特性，所以会表现出几十秒到几小时的高延迟。即使是对于那些已经针对 SQL 查询显著优化过 MapReduce 的系统，比如说谷歌的 Tenzing [27]，又或者是在每个节点上将 MapReduce 和传统数据库整合，比如说 HadoopDB [1]，最小的延迟也达到了 10 秒。因此，使用 MapReduce 的方法基本上不可能实现交互速度的查询[84]，所以即使是谷歌自己也正在开发适用于此类负载的新引擎[75, 95]。

相反，大部分的 MPP 分析数据库*(比如说，* Vertica, Greenplum, Teradata)和几个新的为 MapReduce 环境建立的低延迟引擎(*比如说，*，Google F1 [95], Impala [60]) 采用了一种更粗粒度的恢复模型，在这种模型中，如果一个机器失败了，整个查询必须要重新提交。这种粗粒度的模型很适合执行短查询，因为重新提交段短询的代价比较低，但是对于长查询，这种模型面临着几个重大挑战[1]。此外，这些系统缺少丰富的分析函数，比如说机器学习和图算法，而这些函数在 MapReduce 下是很容易实现的。使用 UDFs 实现这些函数的确是一种可能可行的途径，但是这些算法通常复杂度比较高，这会加剧对错误和 straggler（慢任务）恢复的需求。所以，大多数的企业倾向于结合其他系统和 MPP 数据库去处理复杂的分析。

我们相信，要实现一种更加高效的大数据分析环境，处理系统需要*同时*支持高效的 SQL 和复杂分析运算，还要能够为上述两种运算提供细粒度的恢复模型。我们提出了一个能够满足这些需求的新系统，称之为 Shark。

Shark 使用 RDD 模型来执行大部分的计算，这些计算是在内存中完成的，与此同时，Shark 还提供一个细粒度的容错模型。对于大规模的分析来说，在内存中进行运算正在变得越来越重要，这可以从以下两个方面来解释。第一，许多复杂的分析函数是迭代的，比如说机器学习和图算法。第二，即使传统 SQL 仓库的工作负载也表现出很强的时间和空间局部性，这是因为最近的表数据和小维度表数据经常频繁地被读取。在 Facebook 的 Hive 数据仓库和 Microsoft 的 Bing 分析集群上做的一项研究显示，两个系统中超过 95%的查询可以仅仅使用 64 GB/节点作为高速缓存就能完成，尽管这两个系统管理的总数据量都已经超过 100PB[7]。

但是，为了高效的运行 SQL，我们也必须扩展 RDD 的执行模型，这也引出了几个传统分析数据库中的概念以及一些新的概念。首先，为了高效地存储和处理关系数据，我们实现了在内存中按列压缩存储数据的技术。这种方式与一般存储记录的方式相比，能够减小数据规模，处理时间也能减少到原来的 1/5。第二，为了优化基于数据特征的 SQL 查询（即使在分析函数和 UDFs 存在的情况下），我们使用*局部 DAG 执行(PDE)*来扩展 Spark。在一个查询序列开始执行之后，Spark 可以基于观测到的统计数据，选择一个更好的连接策略或者更合适的并发度，从而实现

重新优化正在运行中的查询序列。第三，我们利用在传统的 MapReduce 系统中不存在而在 Spark 引擎中存在的其他特性，比如说控制数据划分。

我们实现的 Shark 和 Apache 的 Hive 是兼容的[104]，支持 Hive 的所有 SQL 语句和 UDFs，并且不经任何修改就可以在 Hive 数据仓库上使用。Spark 通过引进复杂分析函数增强了 SQL，这些复杂分析函数使用 Spark 的 Java，Scala 和 Python API 来实现。在单个执行计划中，这些函数可以和 SQL 组合在一起，从而为两种类型的处理提供内存数据共享和快速数据恢复的能力。

实验显示同时使用 RDD 和上面提到的所有优化，在 SQL 查询方面，Spark 的速度可以达到 Hive 的 100 倍，在运行迭代机器学习算法方面，Spark 的速度可以达到 Hadoop 的 100 倍，并且可以在几秒钟内从查询错误中恢复。在 Pavlo 等人使用的与 MapReduce 比较的基准测试中，Shark 的速度可以和 MPP 数据库相媲美[84]，但是 Shark 还能提供细粒度的恢复模型和复杂分析特性，这真是那些系统所欠缺的。

3.4 实现

Shark 在 Spark 上执行 SQL 查询的步骤与传统的 RDBMS 类似：查询解析，生成逻辑计划和生成物理计划的。

对于一个给定的查询，Shark 使用 Apache Hive 查询编译器来解析该查询并生成抽象语法树。然后语法树被转换成一个逻辑计划，并对该计划进行一些基本的逻辑优化，如采用谓词下推（pushdown）。到目前为止，Shark 和 Hive 都采用相同的方法。Hive 会将操作转换成由多个 MapReduce 阶段组成的物理计划。至于 Shark，它的优化器采用额外的规则优化，如推送 LIMIT 到各个分区，并创建一个由 RDDs 转换，而不是 MapReduce 任务组成的物理计划。我们可以使用许多 Spark 上已有的操作，例如 *map* 和 *reduce*，也可以使用一些为 Shark 定制的操作，如 broadcast joins（广播连接）。Spark 的 master 使用标准的 DAG 调度技术执行这个依赖图，如将 task 尽量保证数据本地性，重新运行丢失的任务，以及慢节点任务（straggler）迁移等（第 2.5.1 节）。

虽然这种基本方法能够在 Spark 上运行 SQL，但是让它更*高效*的执行仍然是具有挑战性的。在 Shark 中普遍存在 UDF 和复杂的解析函数，使它难以在编译时确定最优的查询计划，尤其是对于那些没有经过 ETL 处理的新数据。此外，即使采用这样的方案，直接在 RDDs 上执行它也可能是低效的。在本节中，我们将讨论在 RDD 模型中高效运行 SQL 的优化方法。

3.4.1 列式内存存储

内存中数据的表示既影响空间占用又影响读取吞吐量。一个原始的方法是简单的将磁盘中的数据按照原格式缓存，然后在查询处理中根据需求执行反序列化。这个反序列化成为了很大的瓶颈：在我们的研究中，我们看到现代的商业 CPUs 单核的序列化速率仅仅在 200MB/秒。

Spark 内存存储的默认方式是将数据分区作为 JVM 对象集合存储。由于查询处理器能直接使用这些对象，这可以避免反序列化，但会付出严重的存储空间代价。一般的 JVM 实现会使得每个对象增加 12 到 16 字节的开销。例如，存储 270MB 的 JVM 对象的 TPC-H 线项表大约要使用 971MB 的内存，而序列化表示则仅需 289MB，存储空间将近为原来的 1/3。但是，更重要的是垃圾收集（Garbage Collection, GC）的影响。在一个记录大小为 200 字节情况下，32GB 的堆栈能容纳 16 亿的对象。JVM 的垃圾收集耗时与堆栈中对象的数量呈线性相关关系，因此在一个大堆栈上执行一次完整的垃圾收集（GC）可能需要几分钟时间。这些不可预测的、昂贵的垃圾收集导致响应时间会有大的波动。

Shark 将基本类型的列以 JVM 的原始数组形式存储。Hive 支持的复杂数据类型，例如 map 和数组，通过序列化串接成一个字节数组。每一列仅创建一个 JVM 对象，可以带来快速的 GCs 和紧凑的数据表示。通过廉价的压缩技术，这种压缩技术基本上不需要 CPU 成本，可以将列数据的空间占用进一步减少。与列数据库系统类似，*e.g.*，C-store [100]，Shark 实现了高效的 CPU 压缩效率模式，例如字典编码、游程编码以及位填充。

列式数据表示可以带来更好的缓存行为，特别是对那些频繁在特定列上进行聚合计算的分析查询。

3.4.2 数据协同划分

在一些数据仓库工作中，两个表经常用来进行连接操作。例如，TPC-H 基准测试频繁地对 lineitem 表和 orders 表进行连接操作。MPP 数据库常用的技术是在数据加载过程中基于两个表的连接键进行协同划分。在 HDFS 等分布式文件系统中，因其存储系统是架构无关的，从而无法进行数据协同划分。Shark 允许两个表基于公共键进行共同划分，这可以在后续的查询中提供快速的连接操作。它在表的创建声明中增加了 DISTRIBUTE BY 语法，用来指定对某个列进行划分。

3.4.3 分区统计和映射修剪

通常情况下，数据是在一个或多个列上使用某种逻辑聚合进行存储的。例如，一个网站的流量日志数据项可能基于用户的物理位置信息进行分组的，因为日志首先是被存储在距离用户最佳地理位置的数据中心。在每一个数据中心内，日志只能被添加，并且按照大致时间顺序进行存储。作为一个不太明显的例子，一个新闻网站的数据可能包含具有很强相关性的新闻 ID（news_id）和时间（timestamp）列。对于分析查询，对这些列进行过滤和聚合操作是很典型的，*例如，*搜索特定时间段或者新闻标题的有关数据。

*映射修剪*是基于其自然聚合列对数据进行分区修剪的过程。由于 Shark 的内存存储将数据拆分成小的分区，每一个数据块在这些列上只包含一个或几个逻辑组，当数据块落在查询过滤条件外时，Shark 可以不用进行数据扫描。

为了利用列（column）在自然聚合的优势，Shark 在每一个工作节点上的的内存存储会在数据加载过程中附带收集统计信息。每个分区收集来的统计信息包含了每一个列的范围，当不同的值个数较少的时候，会包含所有不相同的值(*例如,* 枚举列)。所收集到的统计信息会被发送回驱动节点并存储在内存中，在查询的执行过程中用于修剪分区。当发出一个查询后，Shark 会针对查询的目标评估所有的分区统计信息，然后修剪掉那些没有匹配目标的分区。这是通过简单地创建只依赖于一些父分区的 RDD 来实现的。

我们从一个视频分析公司收集了一些基于 Hive 仓库的查询样本，在我们收集到的 3833 个查询中，至少有 3277 个 Shark 可以利用他们所包含的谓语来进行映射修剪。章节 3.5 中会提供相应工作的更多细节。

3.4.4 局部 DAG 执行 （PDE）

像 Shark 和 Hive 系统经常用来查询未经历过一个数据加载过程的新数据。这就排除了那些依赖精确数据统计的静态查询优化技术的使用，例如通过索引维持的统计信息。新数据统计的缺乏，再加上 UFD 的普遍使用，这就需要动态方法来进行查询优化。

在分布式环境中支持动态查询优化，我们扩展了 Spark 以支持*局部 DAG 执行* (PDE)，这是一项能够允许运行时收集数据统计信息进行动态地改变查询计划的技术。

目前，我们将局部 DAG 执行应用在阻断“shuffle”操作的边界上，在这个阶段数据被交换和重新划分。在 Shark 中，这些都是典型的消耗最大的操作。默认情况下 Spark 在每次 shuffle 前都会将 map 任务的结果物化到内存中，必要时会溢出到磁盘中。然后，reduce 任务会获取这些输出。

PDE 从两个方面修改了此机制。首先，它在全局上收集可定制的统计信息，以及物化 map 任务输出时每个分区的粒度。其次，它允许基于这些统计信息来改变 DAG，或通过选择不同的操作，或改变其参数（例如他们的并行度）。

这些统计数据可以通过使用简单的可插拔累加器 API 来自定义。一些例子的统计信息包括：

1. 分区大小和记录计数，可用于数据倾斜检测。
2. “重量级”列表，*也就是说,*那些经常出现在数据集中的项目。
3. 近似直方图，可以被用来估计分区的数据布局。

这些统计信息由每个 worker 传送给 master，然后它们在那里汇总并提交给优化器。为了提高效率，我们采用有损压缩记录统计信息，限制其大小为每个任务 1-2kB。例如，我们将分区的大小（以字节为单位）用对数编码来实现，可以只用一个字节来表示高达 32GB 的大小，这样误差最高也只有 10%。然后，master 可以使用这些统计信息来执行各种多样的运行优化，这些我们将接下来讨论。

使用当前在 PDE 中已经实现的优化的例子包括：

- Join 算法选择。当连接两个表时，Shark 使用 PDE 来选择运行 shuffle 连接（对两个集合的记录在全网对它们的 key 进行哈希）还是广播连接（将较小表广播到所有节点）。优化算法取决于表的大小：如果一个表比其他的小很多，广播连接就会使用较小的网络通信。因为表的大小可能不会被事先知道（*例如,*由 UDF 引起的），在运行时选择算法会做出更好的决策。
- 并行度。reduce 任务的并行度在类 MapReduce 的系统中有较大的性能影响：启动太少 reduce 任务会使得 reducer 的网络连接过载，并消耗完它们的内存，而如果启动太多的话可能会由于调度开销延长作业[17]。基于局部 DAG 执行，Shark 可以使用单个分区的大小决定运行时 reduce 任务的数目。通过将许多小的细粒度的分区合并为粗粒度的分区，来减少 reduce 任务的数目。
- 倾斜处理。以类似的方式，通过将 map 任务的结果预先划分成许多小块，可以帮助我们来选择 reduce 任务的数目，也可以帮助我们识别并特别处理一些特殊的 key。这些特殊的分区可以由单独的 reduce 任务来处理，那些其他块则可以合并起来形成较大的任务。

局部 DGA 执行实现了现有的一些在单机系统中典型的自适应查询优化技术，[16, 63,107]，因为我们可以使用现有的技术，来动态地优化每个节点内*的本地计划* ，并在阶段的边界使用

PDE 来优化执行计划的全局结构。细粒度的统计信息收集以及优化，使 PDE 区别于先前系统例如 DryadLINQ [115]中的图重写特性。

而 PDE 目前仅在我们的 Shark 原型中实现，未来我们计划将其添加到 Spark 的核心（Core）引擎中去，从而从这些优化中收益。

3.5 性能

我们使用四个数据集对 Shark 进行了评估：

1. Pavlo 等人的基准测试：我们用 2.1TB 的数据重现了 Pavlo 等人对 MapReduce 和分析数据库管理系统的比较[84]。
2. TPC-H 数据集:使用 DBGEN 程序产生了 100G 和 1TB 的数据集[106]。
3. 真实的 Hive 仓库：从一个早期的匿名 Shark 用户那里采集了 1.7TB 的 Hive 仓库数据样本。

总体来说，我们的结果表明，Shark 的执行速度要比 Hive 和 Hadoop 快上 100 倍。特别是，相对于 Pavlo 等人的比较报告中 MPP 数据库的结果，Shark 也有可比较的性能提升[84]。对于那些数据存储在内存中的例子，Shark 超过了报告中 MPP 数据库的性能。

强调一下，我们并不是说，Shark 从根本上超越了 MPP 数据库的速度；因为 MPP 引擎同样可以实现一套与 Shark 相同的优化处理方法。事实上，我们的实现相对于商业引擎来说还存在几个缺点，比如在 Java 虚拟机上运行。相反，我们旨在证明的是，当保留一个类似于 MapReduce 的引擎并同时留有细粒度的错误恢复特性的情况下，实现和商业引擎可比的性能也是有可能的。另外，Shark 可以利用这个引擎来在相同的数据上执行复杂的分析（*例如,* 机器学习），我们相信这将是未来分析工作中必不可少的。

3.5.1 方法和集群设置

除非另有说明，否则实验是在 Amazon EC2 上使用 100 个 m2.4xlarge 的节点进行的。每个节点有 8 个虚拟核，68 GB 内存，1.6 TB 的本地存储器。

集群运行在 64 位 Linux 上（内核版本为 3.2.28），Apache Hadoop 版本为 0.20.205，Apache

Hive 版本为 0.9。对于 Hadoop 的 MapReduce，每个节点的 map 任务数量以及 reduce 的任务数量都匹配节点虚拟核的数目，设定为 8，对于 Hive，我们对任务之间的 JVM 进行重用，并避免合并小的输出文件，这将需要每次查询后执行一个额外的步骤用来进行合并。

每个查询我们执行 6 次，第一次的运行结果会被丢弃，并报告余下的五次运行的平均值。我们放弃第一次运行的结果，是为了让 JVM 的 JIT（just-in-time）编译器去优化公共代码路径。我们认为这更真实的反映了实际的部署，在这样的部署中 JVM 将被多个查询重用。

3.5.2 Pavlo 等人的基准测试

Pavlo 等人对 Hadoop 与 MPP 数据库的性能进行了比较，结果显示 Hadoop 在数据导入方面比较擅长，但是在查询的执行 [84].方面不是很理想。我们使用了他们基准测试中使用的数据集和查询语句，来比较 Hive 和 Shark 的效率。

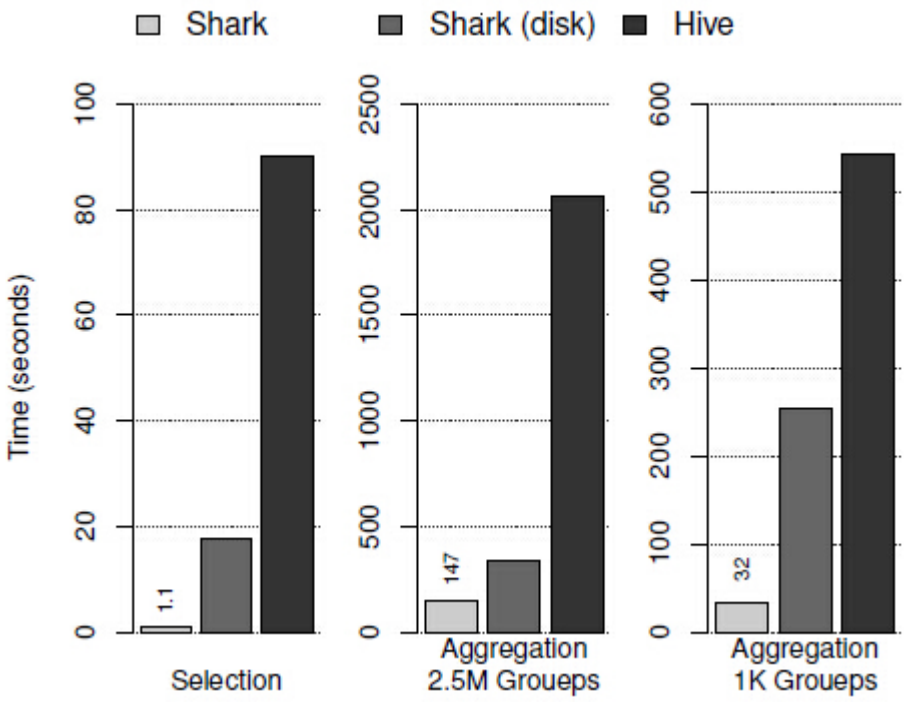


图 3.1. Pavlo 等人的.基准测试中选择和聚合的查询时间（秒）

基准测试使用了两张表：一个是 1GB/节点的 *rankings* 表，另一个是 20GB/节点的 *uservisits* 表。在我们 100 个节点的集群上，我们重建一个 100 GB 的 *rankings* 表，它包含 18 亿行记录，以及一张 2 TB 的 *uservisits* 表，它包含 155 亿行记录。在试验中我们用 Hive 以及 Shark 分别执行了这四个查询，结果的报告如图 3.1 和 3.2.在本小节中，我们手动调整了 Hive 中 reduce 任务的数量，以获得对 Hive 而言最佳的优化结果。尽管对 Hive 做了调整，

Shark 的性能在所有的例子中还是远远超过了 Hive。

选择查询： 第一个查询是在 *rankings* 表上做的一个简单的选择操作：

```
SELECT pageURL, pageRank
FROM rankings WHERE pageRank > X;
```

在[84]，因为针对 Vertica 创建了簇索引，Vertica 的性能超过了 Hadoop 的 10 倍。即使没有簇索引，如果数据存储在内存上，Shark 执行这个查询的速度比 Hive 快 80 倍以上，如果数据存储在 HDFS 上，Shark 也要比 Hive 快 5 倍以上。

聚合查询： Pavlo 等人的基准测试执行了两个聚合查询

```
SELECT sourceIP, SUM(adRevenue)
FROM uservisits GROUP BY sourceIP;
```

```
SELECT SUBSTR(sourceIP, 1，7)， SUM(adRevenue) FROM uservisits GROUP BY SUBSTR(sourceIP, 1，7)；
```

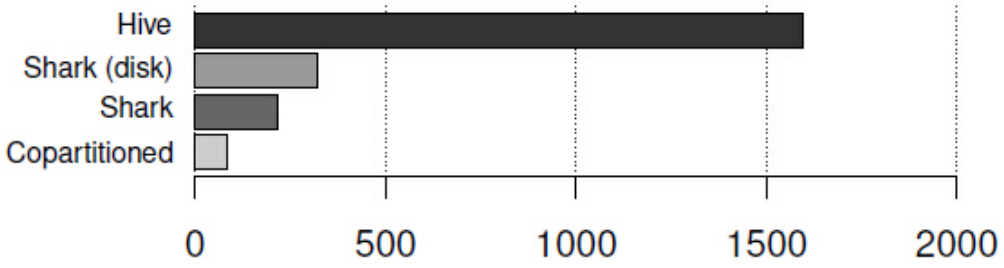


图 3.2 Pavlo 基准测试中 Join 查询的执行时间（秒）

在我们的数据集中，第一个查询有 200 万个 Group，第二个查询大约有一千个 Group。Shark 和 Hive 都采用了任务的本地汇聚和数据清洗（shuffle），以达到将最终的归并聚合操作并行化的目的。Shark 的性能再次大幅度的领先于 Hive。MPP 数据库的基准测试是在每个节点上执行本地聚合，然后将所有的聚合结果发送到一个单一的查询协调器上来做最终的合并;当 Group 的数量比较小的时候这种方式表现的非常出色，但有大量 Group 的时候执行效果反而更差。MPP 数据库选择计划的方法与 Shark/Hive 选择单个 reduce 任务的方法类似。

Join 查询： Pavlo 等人的基准测试的的最后一个查询是 join 一个 2TB 的 *uservisits* 表和一个 100GB 的 *rankings* 表

```
SELECT INTO Temp sourceIP, AVG(pageRank), SUM(adRevenue) as totalRevenue FROM rankings AS R,
```

```
uservisits AS UV WHERE R.pageURL = UV.destURL
AND UV.visitDate BETWEEN Date('2000-01-15') AND Date('2000-01-22')
    GROUP BY UV.sourceIP;
```

Shark 的性能再次在所有例子中都超过了 Hive。图 3.2 显示了对于该查询，如果提供的内存不足，则并不能获得比磁盘更好的性能。这是因为 Join 操作的开销在整个查询处理的过程中占主要地位。但是协同划分两个表，可以得到比较明显的性能提升，因为它避免了在 join 步骤中清洗（shuffle）2.1 TB 的数据的过程。

数据加载：从[84] 可以看出 Hadoop 擅长数据加载，它的数据加载吞吐量是 MPP 数据库的 5 到 10 倍。就像在第 3.4 节说明的那样，Shark 可用于直接查询 HDFS 中的数据，这意味着它的数据导入速率至少和 Hadoop 一样快。

生成了 2 TB 的 *uservisits* 表后，我们测量并比较了将它加载到 HDFS 与加载到 Shark 内存存储的时间 。我们发现，Shark 的内存存储导入数据的速率要比 HDFS 的高 5 倍以上。

3.5.3 微基准测试

为了了解影响 Shark 性能的因素，我们进行了一系列的的微基准测试。我们使用了 TPC-H [106]提供的 DBGEN 程序生成了 100 GB 和 1 TB 的数据。我们选择该数据集，是因为它包含了不同基数的表和列，并且可以用来为测试某个独立的操作创建大量的微基准测试。

在实验中我们发现 Hive 和 Hadoop MapReduce 对于作业设置的 reducer 的数量都是非常敏感的 Hive 的优化器会基于所估计数据的大小自动设置 reducer 的个数然而，我们发现 hive 的优化器经常做出错误的判断，导致很长的查询执行时间。我们基于特定查询，并通过试验和错误的特征对 Hive 手工调整了 reducer 的数量。我们分别给出了通过优化器确定 reducer 数量和手工调整 reducer 数量的 Hive 性能数据。另一方面，shark 对于 reducer 的数量的敏感程度要低于 hadoop mapreduce，几乎不需要进行调整。

聚合性能，我们通过在表 TPC-H *lineitem* 上的运行 group-by 查询来测试聚合性能。对于 100 GB 的数据集，*lineitem* 表有 6 亿行记录。而对于 1TB 的数据集，则包含 69 亿行记录。

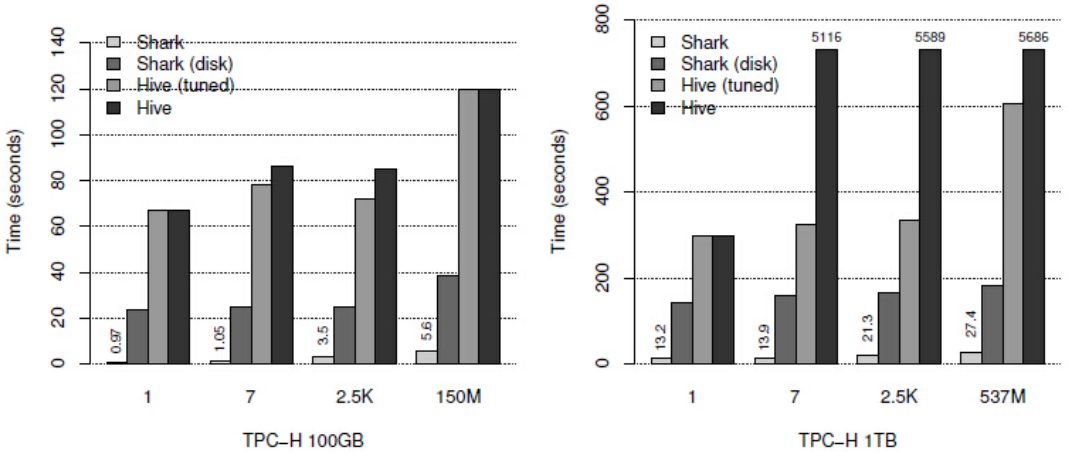


图 3.3。在 *lineitem* 表上的聚集查询。X 轴表示每次聚合查询 group 的数量

查询语句如下所示：

SELECT [GROUP —BY —COLUMN]，COUNT(*) FROM lineitem

GROUP BY [GROUP —BY —COLUMN]

我们运行了一个不带 group-by 的聚合查询（*例如,*一个简单的 count 函数），和三个带有 group-by 的聚合查询：SHIPMODE（7 组），RECEIPTDATE（2500 组），并 SHIPMODE（1.5 亿组，100 GB 和 5.37 亿 组，1 TB）。

对于 shark 和 hive 来说，首先在各分区上的进行聚合，聚合的中间结果经过 parition，然后发送到 reduce task 上，产生最终的聚合结果随着 group（组）的数量变大时，更多的数据需要通过网络进行 shuffle 操作

图 3.3 对 Shark 和 Hive 的性能进行了对比，并测试了 shark 分别使用内存数据和 HDFS 数据的性能。从图中可以看出，对于 group 数量较小的查询，shark 比手动调整过的 hive 快 80 倍；对于 group 数量较大的查询，shark 比手动调整过的 hive 快 20 倍。当 group 数量较大时，整个执行时间主要消耗在 shuffle 阶段。

在某种程度上我们会惊讶于所观察到的 Shark 对磁盘上的数据处理的性能提升。毕竟 shark 和 hive 都要从 hdfs 上读取数据，并为了查询处理要将数据反序列化。对于 shark 和 hive 的这种差异，可以解释为，shark 的任务启动开销很低，并优化了 shuffle 操作以及其他因素。shark 的执行引擎可以在一秒中内启动数千个任务，以最大化可用的并行度，对于聚合查询它只需要执行基于 hash 的 shuffle 操作。在 hive 中，hadoop mapreduce 提供的 shuffle 机制只有基于排序的，这要比基于 hash 的 shuffle 的计算量大。

运行时 Join 选择： 在这个实验中，我们测试了部分 DAG 执行是如何通过对查询计划在运行时进行重新优化来提升查询性能的。该查询在 1TB TPC-H 数据集中 join 了 *lineitem* 和 表 *supplier* 表，并用一个 UDF 对地址进行过滤来选出多感兴趣的供应商。在这种特定的例子下，

UDF 从 1000 万个供应商中选出了 1000 个。图 3.4 总结了这些结果。

```
SELECT * from lineitem l join supplier s ON
l.L_SUPPKEY = s.S 一 SUPPKEY WHERE
SOMEUDF(s.SADDRESS)
```

由于在 UDF 中缺乏良好的选择性估算，静态优化器对这两个表做 shuffle join，因为这两个表的初始化数据很大。通过利用部分 DAG 执行，在对两张表运行预先 shuffle 的 map 阶段后，Shark 的动态优化其会发现过滤后 *supplier* 表会变得很小。于是它会决定执行 map-join，将过滤后的 *supplier* 表复制到的所有节点，并在 *lineitem* 表上只是用 map 任务来执行 join 操作。

为了进一步提高执行效率，优化器会对逻辑计划进行分析，并推断出表 *supplier* 要比表 *lineitem* 小的可能行要大得多（因为原先表 *supplier* 就要小一点，并且对表 *supplier* 有一个过滤的谓语）。应此优化器选择只对表 *supplier* 预先 shuffle，从而避免了在 *lineitem* 上启动两波任务。通过结合静态查询分析

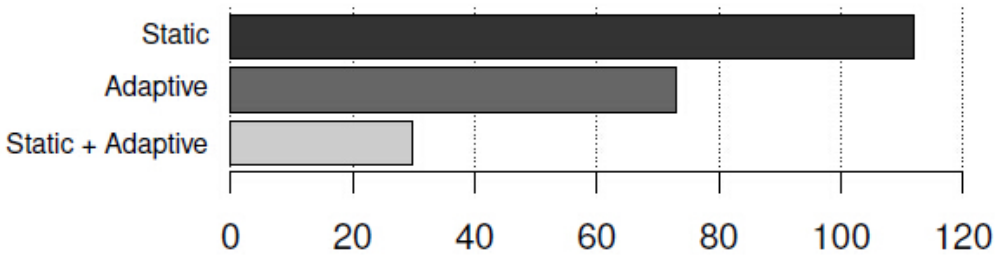


图 3.4 通过优化器选择 join 策略（秒）

和部分 DAG 执行，相对于简单的静态选择计划提升了 3 倍的性能。

3.5.4 容错

为了测试节点故障情况下 Shark 的性能，我们模拟故障并在故障恢复之前、期间、之后进行了查询性能的测试。图 3.5 总结了 5 个错误恢复的实验，这些实验运行在 EC2 集群上，是能够用了 50 个 m2.4xlarge 节点。

在出现节点错误的情况下，我们对 100GB *lineitem* 表使用 group by 查询来测测试查询性能。在将 *lineitem* 的数据加载到 shark 的内存存储后，我们切断了一台工作的机器并重新运行查询。Shark 能完美地从故障中恢复，并以并行的方式在其他 49 个节点上重新构建丢失的数据分块。这个恢复过程对性能有小许的影响，但比重新加载整个数据集并重新进行查询的成本

要低很多（14 vs 34 secs）。

恢复之后，后续查询操作继续使用已经恢复的数据集，尽管只有较少的机器。在图 3.5 中，恢复后的性能要比故障前的性能要好；我们认为，这是 JVM 的 JIT 编译器的副作用，因为在恢复后的查询运行的时候，越来越多的调度程序代码可能已经编译好了，这只是推论。

3.5.5 真实的 Hive 数据仓库查询

一个早期的工业用户（匿名）为我们提供了他们自己的 Hive 数据仓库和两年的 Hive 查询痕迹的一个样本。该用户是一个行业领先的视频分析公司，作为内容的提供商和发布商，他们大部分的分析栈是基于 Hadoop 的。我们拿到了 30 天视频会议数据的样本，解压后占 1.7 TB 的磁盘空间。该样本有一个 103 列的表，并大量使用复杂的数据类型，如数组和结构体。采样的查询日志中包含 3833 个分析查询，排序的频率。我们筛选出了那些调用私有的 UDF 的查询并从整个查询痕迹中选出了 4 个典型的被频繁使用的查询。这些查询基于不同的分片计算出总的视频质量指标。

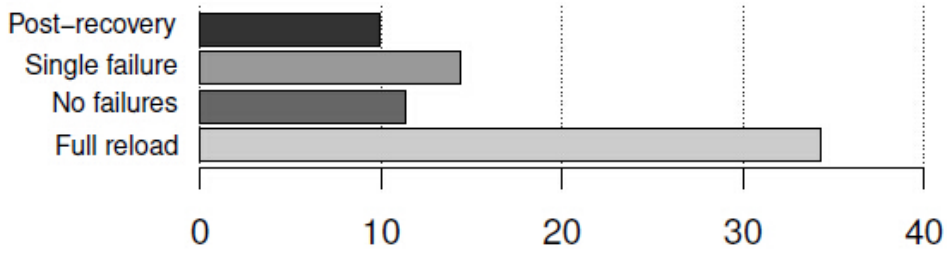


图 3.5 故障的查询时间（秒）

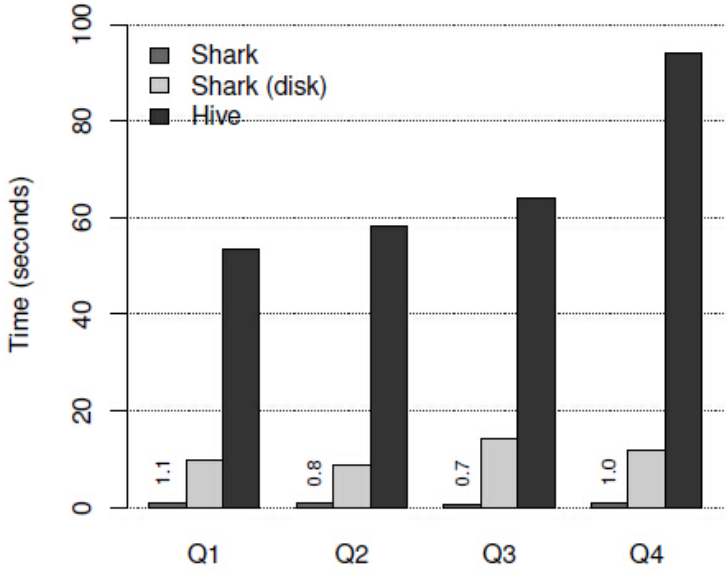


图 3.6 真实的 Hive 数据仓库的工作

1.查询 1 计算某一天特定客户用户的 12 维度汇总统计。

2.查询 2 给出对 8 个列使用谓语过滤并以国家进行分组所得出的不重复的“消费者/客户“组合和会话的个数。

3.查询 3 给出除了 2 个国家外的所有国家的会话个数和不重复的用户

4.查询 4 对一个列进行分组，计算出在 7 个维度上的汇总数据并以降序的方式显示那些最顶层的组。

图 3.6 比较 Shark 和 Hive 在这些查询上的性能。 由于 Shark 能够在亚秒级的时间里处理这些现实生活中的所有查询（除了其中一条），而 Hive 要使用 50 被甚至上百倍的时间来处理，这样的结果说明 Shark 很有前景。

这些查询表明，该数据显示出在 3.4.3 节中提到的自然聚类性质。 Map 修剪技术平均减少了 30 倍的数据扫描量。

3.6 与 SQL 相结合的复杂分析

Shark的一个关键设计目标是提供一个能够进行高效的SQL查询处理和高级机器学习的单系统。遵循将计算接近数据的原则，Shark 将机器学习作为其一级特性。设计决定了选择 Spark 作为执行引擎以及将 RDD 作为操作的主要数据结构。在本节中，我们将介绍 Shark 针对 SQL 和机器学习在语言和执行引擎上的集成。

其他研究项目已经证明，用 SQL 表达某些机器学习算法和避免将数据从数据库中移出是可行的[33,41]。然而，这些项目的实现涉及到 SQL，UDF 和用其它语言编写的驱动程序。系统变得难以理解和维护;此外，为了在传统的数据库引擎上执行代价巨大的并行数值计算，它们可能以牺牲性能的代价，因为传统的数据库引擎并不是为这项工作设计的。相比之下，shark 提供了数据库内的分析，将计算接近数据，shark 采取的方法是使用一个运行时和一个编程模型来实现的，该运行时特别针对这样的工作来优化的，同样的，该编程模型也是被特别设计用来表达机器学习算法的。

3.6.1 语言集成

Shark 复杂的分析能力以两种方式提供给用户。第一种，Spark 程序可以通过调用 Shark 提供的 Scala API 获取 Shark 数据作为 RDD。然后，用户在 RDD 上进行任何 Spark 计算，并使用 SQL 将它们自动串联起来。第二种，我们还扩展了 SQL 的 Hive 方言，允许在 RDD 上调用 Scala 函数，方便暴露现有的 Scala 类库给 Shark。

作为 scalar 集成的一个例子，清单 3.1 表示一个数据分析的流水线，它在用户数据库上使用 SQL 和 Scala 来执行逻辑回归[53]。逻辑回归是一种常用的分类算法，它搜索一个超平面来分离出两组数据点 w （例如，垃圾邮件和非垃圾邮件）。算法采用梯度下降优化算法，算法随机选择初始向量 w，并沿梯度方向迭代更新直至到达最优点。

```
def logRegress(points:RDD[Point]):Vector { var w = Vector(D， 一 => 2 * rand.nextDouble - 1) for
(i < 1 to ITERATIONS) { val gradient = points.map { p => val denom = 1 + exp(-p.y * (w dot p.x))
    (1 / denom - 1) * p.y * p.x }.reduce(一
    + 一) w -= gradient
}
w
}

val users = sql2rdd("SELECT * FROM user u JOIN comment c ON c.uid=u.uid")

val features = users.mapRows { row => new Vector(extractFeature1(row.getInt("age")),
                    extractFeature2(row.getStr("country")),
                    ...)}
val trainedVector = logRegress(features.cache())
```

清单 3.1 逻辑回归示例

该程序使用 sql2rdd 触发一个 SQL 查询获取用户信息，并作为一个 TableRDD。然后在查询行上执行特征抽取，并且在抽取的特征矩阵上执行逻辑回归。每次逻辑回归迭代对所有数据应用同一个 w 函数来产生梯度集合，并求和产生一个总梯度，用于更新 w.

Shark 和 Spark 在集群上自动并行执行 map，mapRows 和 reduce 函数，主程序仅仅收集 reduce 函数的运行结果，用于更新 w。SQL 连接操作顺序执行 reduce 步骤，通过类似 3.2.1 节讨论的迭代接口从 SQL 到 Scala 代码传递列向量数据。

我们还提供 API 从 SQL 中调用 Scala 方法。给定 RDD 的 Scala 函数，像 K-means 或 logistic 回归，用户可以通过 SQL 标记它们为可调用的，然后类似如下方式执行 SQL 代码：

```
CREATE TABLE user —features AS SELECT age， country FROM user;

GENERATE KMeans(user —features， 10) AS TABLE user —features —clustered;
```

在这个例子中，表 user_features_clustered 每行将包含年龄，国家和一个新的域，该域为集群 ID。10 是传给 KMeans 的集群数目。

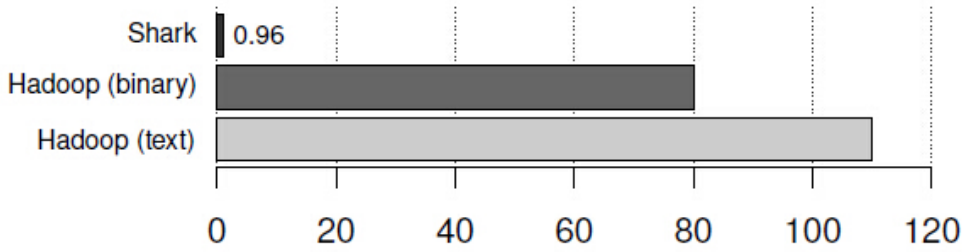


图 3.7.逻辑回归，每次迭代运行时间（秒）

3.6.2 执行引擎集成

除了语言集成，执行引擎集成是使用 RDD 作为操作的数据结构的另一个主要优点。这种通用的抽象允许机器学习计算和SQL查询在无需大量的数据移动的情况下共享worker和缓存数据。

由于 SQL 查询处理使用 RDD 实现，因此 lineage 保存了整个流水线，使得整个工作流具有端到端的容错能力。如果在机器学习阶段发生故障，故障节点上的数据分区会自动根据它们的 lineage 重新计算。

3.6.3 性能

我们实现了两个迭代机器学习算法，逻辑回归和 K-means，把 Shark 的性能与运行相同工作流程的 Hive 和 Hadoop 来进行比较。这个数据集被合成产生，含有 10 亿行，10 列，占用 100GB 空间。因此，该特征矩阵包含 10 亿个点，每个点有 10 个维度。这些机器学习实验在一个 100 个节点的 m1.xlarge EC2 集群上执行的。

数据最初以关系表的形式存储在 Shark 的内存和 HDFS 中。该工作流包括三个步骤：（1）使用 SQL 从仓库中选择感兴趣的数据，（2）提取特征，以及（3）应用迭代算法。在步骤 3 中，这两种算法都运行 10 次迭代。

图3.7和 3.8显示出执行逻辑回归和k-means的单次迭代分别用的时间。我们实现了Hadoop

的两个版本的算法，一个是把输入数据在 HDFS 中作为文本存储，另一个是使用序列化的二进制格式。二进制表示更紧凑，在记录反序列化时降低了 CPU 运行成本，从而提高性能。我们的研究结果表明，对于逻辑回归，Shark 比 Hive 和 Hadoop 快 100 倍，对于 K-均值算法，要快 30 倍。K-均值算法提速比较少，因为它在计算上比逻辑回归成本更高，因此使工作流程更加受限于 CPU。

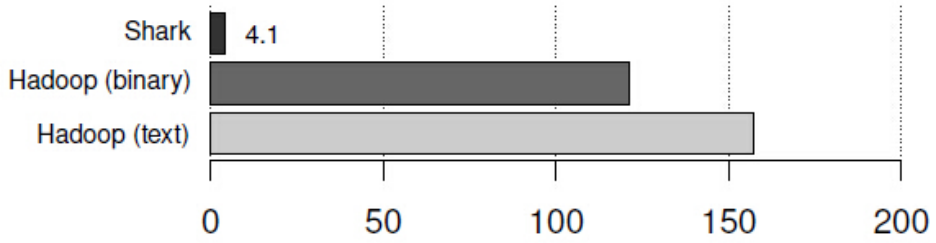


图 3.8 K-means 聚类，每次迭代运行时间（秒）

在 Shark 的案例中，如果数据最初驻留在内存中，步骤 1 和步骤 2 运行机器学习算法单次迭代花费的时间大略相同。如果数据没有被加载到内存中，这两种算法的第一次迭代都用了 40 秒。随后的迭代所花的时间，参见图 3.7 和 3.8。在 Hive 和 Hadoop 的的案例中，每一次迭代花的图上所显示较多的时间，因为每一次迭代数据都从 HDFS 中被加载。

3.7 总结

在本章中，我们介绍使用 RDD 的相关技术来实现更加复杂的业务处理和存储优化，并通过 Shark 的数据仓库系统例子来说明。类似的技术，包括对 Spark 记录内批量数据进行索引以及优化分区，已被用于像 GraphX[112]，MLlib [96]，MLI 的 [98]等项目。同时，对于各自领域中的特殊系统，这些技术也使得基于 RDD 的系统获得了类似的性能，并在类型的结合处理和类型的容错计算方面的应用中提供了更高的性能。

第四章 离散流

4.1 简介

本章讲述第二章里的 RDD 模型在一个有可能是最令人心动领域里的应用：大规模流处理。虽然其设计与传统流系统不同，但它提供丰富的故障恢复，以及强大的与其它处理类型融合的能力。

大规模流处理的动机很简单：大部分“大数据”都是实时获取的，并且到达之时最有价值。例如，社交网络或想检测出近几分钟内的热点话题，搜索网站会想对哪些用户会访问新网页进行建模，又或是服务运营商对程序日志进行秒级监控以实现实时故障侦测。要实现这些应用，就需要能扩展到大型集群的流处理模型。

然而，设计这样的模型并不容易，因为应用(*比如*实时日志处理或机器学习)所需的规模可达数百个节点。在这种规模下，系统故障和慢节点（straggler）问题会变得很严重[36]，并且流式应用尤其需要快速恢复。事实上，相比在批处理类应用中，快速恢复在流应用中更显重要：在批处理下，用 30 秒钟从系统故障或慢节点里恢复或许可以接受，而在流处理中，这 30 秒便可错过一个重要的决策。

不幸的是，现有的流系统对系统故障或慢任务（straggler）的应对能力有限。大多数分布式流式系统，包括 Storm[14]，TimeStream[87]，MapReduce Online[34]，和流式数据库[18，26,29]，都是基于连续操作模型。在这种模型中，长期运行的带有状态的操作会接收每条记录，更新内部状态，并且发送新的记录。这样的模型设计很自然，但也让它难以应对系统故障和慢任务问题。

特别的，给定连续操作模型，系统通过两种途径来进行恢复[58]：
*复制，*每个节点存在两个副本[18, 93]，或者*上行流备份（upstream backup），*节点缓存发送信息并在一个故障节点的新副本里重新执行。[87,34,14]。两种途径都不太适合大规模集群：复制要占用 2 倍的硬件，而上行流备份需要长时间的恢复，因为整个系统必须等待一个新的节点通过重新运行操作数据串行重建故障节点的状态。此外，这两种途经都不能处理慢任务问题：在上行流备份方案下，慢任务需要当作系统故障来处理，而这样的恢复代价高昂；在复制方案的系统里，采用如 Flux[93]的同步协议来进行副本之间的协调，恢复速度将受限于慢任务。

这里提出一种名为 *离散流*(D-Streams) 的新式流数据处理模型来克服上述问题。与管理长时间存在的操作不同，D-Streams 结构将各运算流化成为一系列短时间间隔的*无状态*、*确定性的*批计算。例如，我们可以将每秒钟(或每 100 毫秒)所接受的数据按照较短的时间间隔来分段，然后对每一段数据进行 MapReduce 操作来实现计数。同样的，可将各间隔求出的新的计数加到

旧的结果上而实现滚动计数。通过将计算按这样的方式来够造，D-Streams 可以确保（1）对于给定输入数据，每个时间间隔内的*状态*完全确定，而不需要同步协议；并且（2）当前状态和旧数据的*依赖关系*细粒度可见。我们表明，这样的设计能提供如批处理系统那样的强大的恢复机制，胜于复制和上行流备份方案。

实现 D-Stream 模型存在两方面的挑战。首先是要降低延时（间隔粒度）。传统批处理，如 Hadoop，在这方面有缺陷，因为它们任务间使用复制、磁盘储存方式保存状态。相反，我们建立在第二章提到的 RDD 数据结构上，可以在内存中保存数据，并且使用操作的 *lineage* 进行恢复，从而避免了复制。通过 RDD，我们证明可以达到低于秒级的端对端延迟。相信这足以满足许多实际大数据应用的需要，一般来说这些应用的时间尺度（*例如：*社会媒体的倾向） 要高得多。

第二个挑战是从故障和慢任务中快速恢复。这里我们通过 D-Streams 的确定性提供一种新的恢复机制，这种机制在以往的流系统中均未使用过。一个丢失节点状态的*并行恢复，*当某个节点失效时，集群中的各个节点都分担并计算出丢失节点 RDD 的一部分，从而使得恢复速度远快于上行流备份，且无复制开销。由于需要复杂的状态同步协议，即使是简单的复制操作（例如，Flux[93]），在连续处理系统中并行恢复也难以实现，但这对完全确定性的 D-stream 模型却变得简单。

与前一条类似，D-Stream 可通过推测性执行（speculative execution）[36]来从慢任务中恢复，而之前的系统均不处理该类任务。

基于 Spark，我们已经在 Spark Streaming 中实现了 D-Streams。这个系统在 100 个节点上能够处理超过 6000 万记录/秒，延迟在亚秒级，并且可以亚秒时间内从故障和慢任务中恢复。Spark Streaming 的单节点吞吐量可与商用流数据库相当，但同时提供百级节点上线性扩展的能力。它比开源的 Storm 和 S4 系统快 2-5 倍，还提供它们所没有的故障恢复保证。除了性能方面外，对 Spark Streaming 的阐述还将包括两个实际应用举例：一个是视频分发监控系统，另外一个是在线机器学习系统。

最后，因为 D-Streams 使用与批任务相同的处理模型和数据结构（RDD），该流处理模型能实现流查询和交互式计算以及批计算无缝结合。这是一个明显的优势。在 Spark Streaming 中，我们利用这个特性让用户使用 Spark 在流上进行即时（Ad-hoc）查询，或把流和已计算出的历史数据连接成一个 RDD。在实际中，这种特性很有价值，它使得用户通过单一 API 来整合以前彼此不同的计算。下文将阐述 D-Stream 是如何被用来桥接在线处理和离线处理处理的。

4.2 目标与背景

许多重要的应用需要对实时到达的大规模数据流进行处理。我们的工作目标是应用需要在几十到数百台机器上执行，并且可以容忍几秒钟的延迟。一些示例如下：

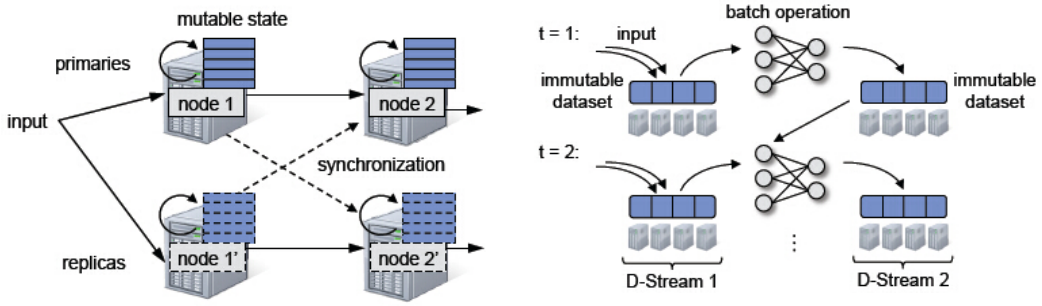
- 网站活动的统计数据：Facebook 建立了一个分布式聚合系统 Puma，来让广告者统计用户在 10-30 秒内点击他们网页的次数和处理时间[94]。
- 集群监控：数据中心运营商往往使用由数百个节点组成的[52]如 Flume[9]这样的系统，来对程序日志进行收集和挖掘以发现问题。
- 垃圾邮件检测：社交网络如 Twitter 可能希望利用统计学习算法 [102]来实时识别新的垃圾邮件活动。

对这些应用，我们认为，D-Streams 的 0.5-2 秒的延迟是足够的，因为该延迟远高于所监控的趋势的时间响应需求。我们特意不针对那些延迟要求低于几百毫秒的应用程序，如高频交易。

4.2.1 目标

为使得这类应用可大规模运行，理想的系统设计需满足如下四个目标：

1. 成百上千的可伸缩节点数目
2. 基本计算之外的开销最小--例如，不希望付出 2 倍的备份开销。
3. 二级延迟。
4. 从系统故障和慢节点恢复所带来的二级恢复。



（a）连续操作处理模型 每个节点连续地接收数据、更新内部状态并且发出新的记录。容错一般来说是通过复制数据来实现的，用类似于 Flux 或 DPC[93, 18]的同步协议来确保副本数据在每个节
点看来都是相同的顺序(例如，当它们有多个父节点时)。

(b) D-Stream 处理模型 在每个时间间隔，到达的记录被可靠的存储在集群中，形成一个不可变的分区的数据集。之后，这个数据集通过确定性的并行操作，计算其他表示程序输出或状态的分布式数据集。这些会作为下一个间隔里的输入。一个系列里的各个数据集构成一个 D-Stream。

图 4.1 对比传统连续性流处理(a)与离散流(b).

据我们所知，之前的系统无法满足这些目标：基于复制的系统开销很大，而基于上行流备份的系统则需数十秒来恢复丢失的状态信息[87,110]，另外两者均不处理慢节点问题。

4.2.2 以往的处理模型

虽然人们已经对分布式流处理进行了广泛的研究，但大部分现有系统均使用*连续操作* 处理模型。在该模型下，流计算被分隔为由多个有状态的算子(运算)的集合，而各算子以新到的记录为输入来更新自身状态*(比如*一个统计某个时间段内页面浏览次数的表)，从而完成对其的计算，并发出新的记录来作为回应。图 4.1(a)表示

尽管连续处理最小化了延迟，但是操作的状态化的特征和由于网络传输记录导致的不确定性，很难有效提供容错机制。具体来说，恢复的最大挑战在于操作状态在丢失节点或慢节点上的重建。之前的系统使用两种方案中的一种：*复制*或上行流备份[58]。这并不能在恢复开销和恢复时间之间进行良好平衡。

在复制模型下，这种模型是数据库系统中常用的模型，处理流程会有一个备份，而输入数据会都发给它们。然而，只是对节点做备份并不够，系统还需如 Flux 或者 Borealis's DPC 那样运行一个同步协议，来保证每个操作(含备份的)会以相同的顺序来对待上游发来的消息。

比如说：一个输出联合（union）两个父运算流的操作需要确保父运算流顺序相同，才能得出相同的输出流，所以操作与其拷贝之间需要协调。因此，复制方案虽然可以很快恢复，但是耗费大量资源。

上行流备份模式:每个节点在检查点时保存其所发出数据的副本。当一个节点失败之后，备用节点马上接管，父运算流会重新发送信息给备用节点来重建。这种方式需要花费大量恢复时间，因为通过运行一系列带状态的操作的代码来重新计算出丢失的状态只能在同一个节点上进行。TimeStream 和 MapReduce Online 使用的是这个模型。 主流的消息队列系统，比如 Storm，也是使用的这种模式，而且通常只保证“至少一次”发送消息，这依赖用户代码来实现处理状态恢复。

更重要的是，复制和上行流备份模式都不能应对慢任务问题。如果在复制模式下，如果一个节点运算很慢，为了确保复制能够保持同序，整个系统都会很慢。在上行流备份模式下，处理慢任务的唯一方法就是标记为失败，这就需要经历前面所提到的缓慢的状态恢复进程，对于偶尔发生的问题，这种规模是太过笨重。.因此，传统的流方法在小规模环境中工作良好，但是在大规模集群中会面对大量问题。[12]

4.3 离散流（D-Streams）

D-Streams 通过将计算构造为一组*短的，无状态的，确定性的任务*代替连续的，有状态的操作来避免传统流处理的问题。然后，它们将状态存放在内存中，再通过容错的数据结构（RDDs）可以重新计算出该状态。将计算分解成短任务并暴露其细粒度的依赖性，并允许像并行恢复和推测（speculate）这样强大的恢复技术。除了容错，D-Stream 模型提供了其他好处，比如与批处理相结合。

[12] 需要注意的是，如批处理中那样的推测执行在这里难以适用。这是因为算子已默认输入是连续的，从而使得即便对单个算法进行备份也因要从其上一个检查点恢复它而耗费大量时间。

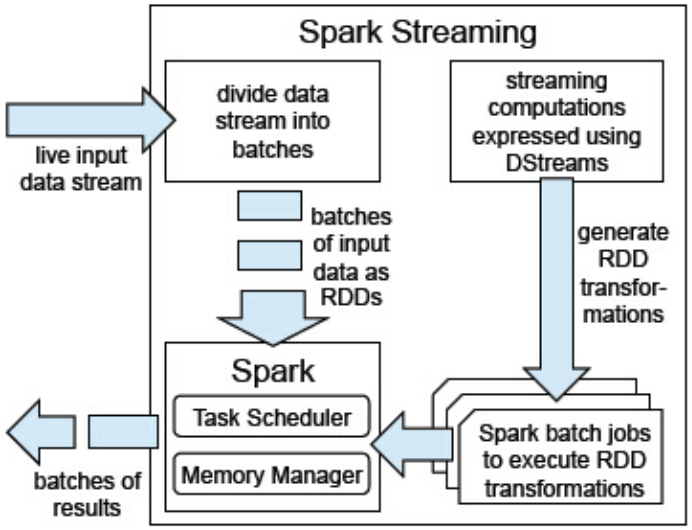


图 4.2 Spark 流系统的高级概述。Spark Streaming 把输入数据流分成批次，并将它们存储在 Spark 内存中，然后通过产生用来处理每个批次数据的 Spark 作业的方式来执行一个流应用程序。

4.3.1 计算模型

我们把流计算看作在一小段时间周期上进行的一系列确定性的批处理计算。对于每个时间周期收到的数据，通过集群可靠地存储成一个*输入数据集*。一旦时间周期完成时，该数据集便通过确定的并行操作来处理，例如通过 *map, reduce* 和 *groupBy* 等操作来产生新的数据集，该数据集可以表示程序输出或中间状态。对于前面的情况，结果可以以分布式的方式推送到一个外部系统。在后面的例子中，中间状态可以通过*弹性分布式数据集（RDDs）*的高效抽象的存储方式保存，这样可以避免为了恢复使用 lineage 而产生冗余。该状态数据集可以随同下一批输入数据一起处理，以产生一个新的数据集来更新中间状态。图 4.1（b）显示了我们的模型

基于这个模型，我们用 Spark Streaming 实现了这个系统。我们对每一批数据使用 Spark 作为批处理引擎。图 4.2 大致描绘了 Spark Streaming 上下文中的计算模型，后面我们会作更详细的解释。

在我们的 API 中，用户通过操纵对象来定义流程，我们称之为 *离散流（D-Streams）*。D-Stream 是一系列具有不可变性的分区数据集（RDDs），我们可以通过确定的*转换*对它们进行操作。这些转换生成新的 D-Streams，并且可以通过 RDDs 的形式创建中间*状态*

我们通过 Spark Steaming 流式计算运行的程序实现了这个想法。

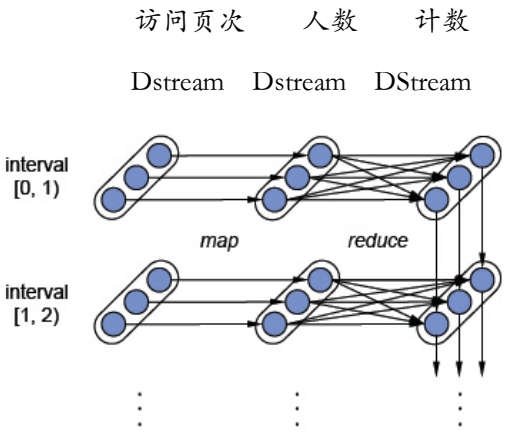


图 4.3 在 view count 程序里 RDDs 的 lineage。每个椭圆代表一个 RDD，分区用圆圈表示。每个 RDDs 的序列是一个 D-Stream。

通过URL计算访问事件。类似于LINQ[115,3]，Spark Streaming通过Scala语言的可编程的API暴露D-Streams.[13]我们的程序代码如下：

```
pageViews = readStream("http://…" "1s")

ones = pageViews.map(event => (event.url, 1))

counts = ones.runningReduce((a, b) => a + b)
```

这段代码创建了一个名叫 pageViews 的 D-Stream，通过从 HTTP 读取事件流将他们用 1 秒的时间周期来分组页面访问。然后将这个事件流通过建立（URL，1）这样的键值的变化对来形成新的 D-Stream，再通过一个状态相关的 *runningReduce* 转换来对他们进行计数操作。传入 *map* 和 *runningReduce* 的参数是 Scala 的函数文本。

为了执行这个程序，Spark Streaming 接收数据流，然后将其划分成秒级的批处理任务，并将其存储在 Spark 的 RDDs 内存中（见图 4.2）。同时，他也会调用 RDD 的转换操作如同 *map* 和 *reduce* 来对 RDD 进行处理。为了执行这些转换，首先 Spark 会启动 *map* 任务来对这些事件进行处理，同时生成 (url,1)这样的计数对。然后，会对 map 的结果和之前 reduce 得到的结果启动 *reduce* 任务，最后将结果存储在 RDD 里。这些任务会产生一个更新计数的新 RDD。程序中的每个 D-Stream 因此变成了一系列的 RDD。

最后，为了错误恢复和慢任务，D-Streams和RDDs要跟踪他们的lineage,，即用于生成他们的确定性的操作图。在每一个分布的数据集中，Spark会在分区的层面跟踪这些信息 ，如图 4.3 所示。如果一个节点任务失败，通过重新运行从集群中可靠存储的输入数据构建的任务来重新计算相应的RDD分区。这个系统还周期性的检查RDD的状态（*例如通过*异步的方式对每十个RDD

[13]其它接口，比如流 SQL 中，也将是可行的。

进行复制）[14] 以避免过度重算。但是不需要对所有数据都进行那样的操作，因为恢复总是很快的：丢失的分区可以在不同的节点上*并行*计算。类似的，当一个节点运行缓慢时，因为总会产生同样的结果，我们可以在其他节点上对任务的副本进行推测执行[36]。

我们发现在 D-Streams 中并行恢复比在上游备份具有更高的可用性，即使每个节点上执行了多个操作。D-Stream 从操作*分区*和*时间*两个方面展现了并行化：

1. 如同每个节点执行多个任务的批处理系统，每个节点在每个转换操作的时间片会产生多个 RDD 分区（*例如* 100 核的集群产生 1000 个 RDD 分区）。当节点出现故障时，我们可以在其它节点以并行方式重新计算其分区。

2. lineage 图通常可以使数据从不同的时间片并行地进行重建。如图 4.3 所示，如果一个节点出错，我们可能丢失一些时间片的 *map* 的输出，不同时间片的 map 任务可以并行的重新执行。假设需要执行一系列的操作，这在一个连续处理的系统中是无法实现那样的功能的。

依赖这些特性，当每 30 秒建立一次检查点时 （§4.6.2），D-Streams 仅用 1-2 秒就可以在数百个核上并行恢复。

我们将在本节的剩余部分更详细地介绍 D-Streams 的可靠性和编程接口。并在第 4.4.节中讨论如何实现。

4.3.2 时序方面的考虑

D-Streams 将每个记录按其*到达*系统的时间存入输入数据集。这样做可以确保系统总是可以及时开始一个新的批次，尤其是在那些记录从相同的地方里产生的应用中，*例如*，同一数据中心的服务产生的数据。这样分割处理的方式，在语义上不会产生错误。而在其他应用中，开发者可能希望基于事件发生的*外部时间戳*将记录分组，*例如*，基于用户点击某一个链接的时间。这样一来记录的到达可能是无序的。D-Streams 提供了两种方法来处理这种情况：

1. 系统可以在开始每个批次之前*等待*一个有限的“空闲时间”

2. 用户程序可以在*应用级别*上对晚到的记录进行纠正。例如，假设一个应用想要在 t 时刻与 $t+1$ 时刻间对某广告的点击数进行统计。一旦 $t + 1$ 时刻过去，应用就可以使用以一秒为周期的 D-Streams，对 t 时刻与 $t+1$ 时刻间接收的点击数进行统计。然后，在后面的时间周期里系统可以进一步收集 t 与 $t+1$ 时刻间的其他带有外部时间戳的事件并计算更新统计结果。例如，它可能将基于从 t 到 t+5 时间段内收到的记录，在 t+5 时刻产生一

[14]由于 RDD 具有不变性，所以设立检查点的操作不会阻塞任务。

个关于[t, $t+1$)时间区间的新的计数。这种计算可以应用一种高效的增量 *reduce* 操作，即在 $t+1$ 时刻的老计数基础上加上对之后新记录的计数，以避免重复计算。此方法类似与顺序无关处理[67].

这些时序性的考虑是流式处理系统所必须面对的，因为任何系统都会有外部延时。数据库领域对此已经进行了详细的研究 [67, 99]。一般来说,这些技术都可以通过D-Streams来实现，即将计算”离散化“到小批次数据的计算（相同批次的处理逻辑相同)。因此我们将不在本文中对这些方法做进一步的探讨。

4.3.3 D-Stream API

因为D-Streams是主要的执行策略（描述如何将一个计算分解成多个步骤)，因此他们被用在流式系统中实现了多个标准的操作，比如滑动窗口和增量式处理[29, 15]，以简单的对它们的执行批处理到各个小的时间间隔中。为了说明这一过程，我们描述了在 Spark Streaming 中的各个操作，也能支持其他的接口 *（例如，* SQL)。

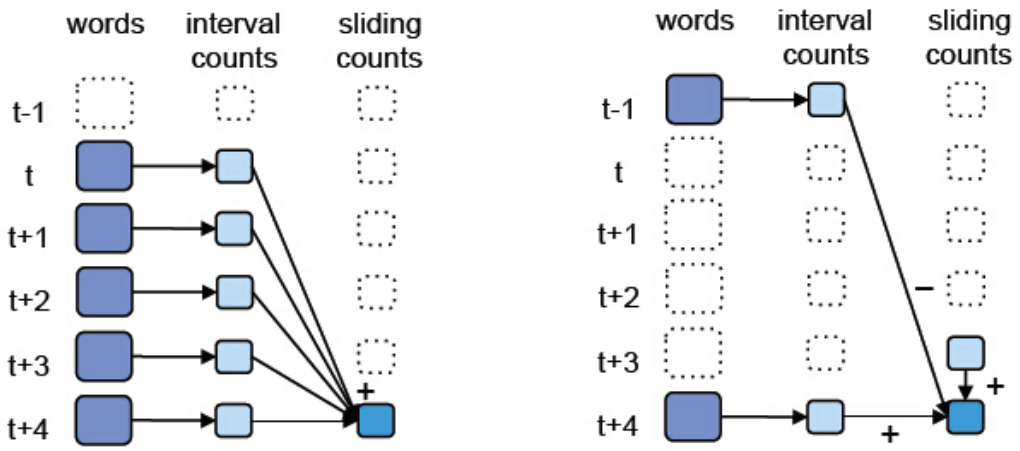
在 Spark Streaming 中，用户使用函数 API 来注册一个或多个数据流。程序可以将*输入*数据流定义为从外部系统中读取数据，该系统通过从对节点端口监听或周期性地从一个存储系统*(例如,*HDFS）加载来获取数据。它可以适用于两种类型对这些数据流的操作：

- *转换操作*，从一个或多个父数据流创建一个新的 D-Stream。这些操作可能是*无状态的*，在每个时间周期内对 RDD 分别进行处理，或它们可能跨越周期来创建状态。
- *输出操作,*，使得程序将数据写入外部系统。例如，*save* 操作将 D-Stream 中的每一个 RDD 输出到数据库。

D-Streams 支持在典型批处理框架中所拥有的无状态的转换操作[36,115], 包括 *map, reduce, groupBy,* 和 *join*。我们在 Spark 中提供了所有的操作。例如，一个程序使用以下的代码可以在 D-Stream 的每一个时间周期内，运行一个规范的 MapReduce word count 程序。

```
pairs = words.map(w => (w， 1))
counts = pairs.reduceByKey((a， b) => a + b)
```

此外，为了支持跨越多个周期的计算，D-Streams 提供了多个*有状态的*转换操作，这些操作是基于标准的数据流处理技术的基础上例如滑动窗口[29, 15]。这些操作包括：



（a）单一关联　　（b）关联和可逆

图 4.4.用于单一关联和关联+可逆版本的操作执行 的 *reduceByWindow*。这两个版本为每个时间间隔只进行一次计数的计算，但是第二个版本的操作避免了对每一个窗口进行重新求和。方框表示 RDDs，箭头表示用来计算窗口的操作\(t, i + 5)。

窗口：*window* 操作将每一个过去的时间周期的滑动窗口里的所有记录组合到一个 RDD。例如，调用上述代码中的 words.window("5s"),会产生一个包含周期内单词的 RDDs 的 D-Stream [0,5)，[1,6)，[2,7)，等。

增量式聚合：对于常用的聚合计算的用例，就像在一个滑动窗口上进行 count 或 max 操作，D-Streams 有增量 *reduceByWindow* 操作的几个变种操作。最简单的一个是仅仅用一个关联的合并函数来对值进行合并。例如，在上述代码中，用户可以写：

pairs.reduceByWindow(“5s”, (a, b) => a + b)

对于每一个时间周期只对该周期的计数进行一次计算，但不得不反复的对过去的 5 秒去添加计数，如图 4.4(a)所示。如果聚合函数也是*可逆的*,一个更加高效的版本还需要“减”值和增量式维护状态的一个函数（图 4.4(b)）:

pairs.reduceByWindow(“5s”, (a， b) => a+b, (a， b) => a-b)

状态跟踪：通常，应用程序为了对表示状态变化的事件流进行响应，需要对各类对象进行*状态*跟踪。例如，一个监控在线视频传输的程序可能会希望对活跃连接的数量进行追踪，一个连接表示从系统收到一个新客户端的“join”事件和当它收到“exit”的时间。然后，它能够提出这样的问题：“有多少个比特率大于 X 的会话”

D-Streams 提供了一个转换数据流的操作 *updateStateByKey*

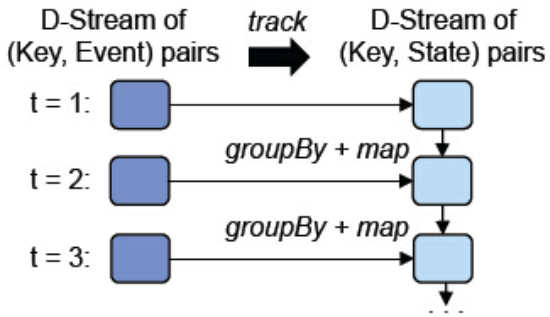


图 4.5.由 *updateStateByKey* 操作创建的 RDDs

基于三个参数记录(Key, Event) 到(Key, State) 记录的数据流中

- 一个从第一个事件中为新的键值创建一个 State 值的 *initialize* 函数。
- 一个从给定的一个旧状态和一个事件里为它的键值返回一个新的 State 值的 *update* 函数
- 一个用于删除旧的状态的 *timeout* 函数。

例如，用户可以从一个(ClientID, Event) 对的数据流中计算活跃的会话数，如下所示：

```
sessions = events.track(
    (key, ev) =>1，                                    // initialize function
(key, st, ev) => (ev == Exit ? null:1)，  // update 函数
    "30s")                                                  II 超时
counts = sessions.count()                          // 一个整型的数据流
```

这段代码给每一个活跃客户的状态设为 1，并且在它退出时通过从 update 中返回 null 来将它删掉。因此，会话对于每一个活跃客户含有一个(ClientID, 1) 元素，同时 counts 用来计算会话的总数。

在 Spark 中可以使用批处理的操作来实现这些操作，通过将批处理操作应用到来自父数据流中不同时间的 RDDs，例如，图 4.5 表示由 *updateStateByKey* 构建的 RDDs，通过对旧的状态和每个周期的新事件进行分组来实现。

最后，用户调用*输出操作符* 将 Spark Streaming 的结果发送到外部系统 *（例如，*展示在 dashboard 上)。我们提供了两个这样的操作：*save 操作，* 将 D-Stream 中的每一个 RDD 写入到一个存储系统 *（例如,* HDFS 或 HBase)，和 *foreachRDD,* 在每一个 RDD 上执行一段用户代码段（任意的 Spark 代码)。例如，用户可以用 counts.foreachRDD(rdd => print(rdd.top(K)))来打印 top K 的计数。

4.3.4 一致性语义

D-Streams 的一个好处是，它们具有真正的一致性语义。跨节点的状态一致性在以记录为基础的流式系统中是一个迫切的问题。例如，有这样一个系统，按国家来计算其页面访问量，每个页面的浏览事件被发送给负责汇总其国家统计数据的不同节点上。如果负责英格兰的节点落后于负责法国的节点，*例如，* 由于加载的原因，那么它们的状态快照将会出现不一致：英格兰的计数与法国的相比将会反映流的一个较老的状态，而且计数值通常会较低，从而混淆有关事件的推论。有些系统，像 Borealis [18], 其同步节点会避免这个问题。而其他的系统，像 Storm，却是忽略它。

在 D-Streams 中，一致性语义是非常明确的，因为时间会自然离散为时间周期，每个时间周期的输出 RDDs 反映当前时间周期内以及以前的时间周期收到的*所有* 输入。这是真实的，无论输出 RDDs 和状态 RDDs 是否分布在集群中，用户无需担心是否有节点在执行上落后。具体来讲，由于计算的确定性和不同时间周期的数据集的单独命名，每个输出 RDD 的计算效果相当于以前的时间周期上所有批量作业已经步调一致地运行，并且没有落后的和失败的。因此，D-Streams 提供一致的“恰好一次”处理。

4.3.5批处理与交互式处理的统一

因为 D-Streams 遵循与比处理系统相同的处理模型，数据结构（RDDs），和类似批处理系统的容错机制，因此两者可以无缝结合。Spark Streaming 提供了多种强大的功能来统一流式计算和批处理计算。

首先，D-Streams 能够使用标准的 Spark 作业与静态的 RDDs 结合进行计算。例如，我们可以将消息事件流和预先计算的垃圾过滤器进行*连接*操作，或者与历史数据进行比较。

其次，用户可以使用“批处理”的模式对历史数据运行一个 D-Stream 程序。这可以非常方便为历史数据计算一个新的数据流报告。

第三，附加一个 Scala 控制台到 Spark Streaming 程序里，用户可以在 D-Streams 上进行*交互式* 查询，并且在 RDDs 上运行任意的 Spark 操作。例如，用户可以查询在一个时间范围内最流行的词：

```
counts.slice("21:00", "21:05").topK(10)
```

与曾经编写过离线（基于 Hadoop）和在线处理应用的开发人员的讨论结果显示这些特性具有重要的实用价值。

方面	D-Streams	连续处理系统
延迟	0.5-2 s	1-100 ms ，除非为了一致性批次处理记录
一致性	记录在到达的时间间隔内原子性的进行处理	有些系统可以等待短暂的时间再继续执行同步操作[18, 87]
记录延迟	延迟时间或 app 级别的修正	延迟时间，乱序处理 [67, 99]
故障恢复	快速并行恢复	在单节点上复制或串行恢复
慢任务恢复	推测执行的可能	通常情况下没有处理
批处理混合操作	通过 RDD API 的简单统一	在一些数据库系统中[43]；在消息排队系统中没有

表 4.1 连续操作的系统和 D-Stream 的比较。

与流式系统和批处理系统拥有各自单独 API 的系统相比，共享同一份代码库可以节省许多开发时间。同时在流系统中交互式查询状态的能力则更加吸引人；它使得调试一个运行程序，或者在聚类操作的流式作业中查询未定义状态变得更加容易， *例如：*解决一个网站的问题。 如果没有这种特性，用户通常需要等待数十分钟来将数据导入到集群里，即使流式系统处理节点的内存具有所有相关的状态信息。

4.3.6总结

在介绍 D-Streams 的结尾部分，我们在表 4.1.中将它和连续处理系统进行了对比。它们的主要区别是，D-Streams 将任务划分成小的且确定性的批量操作。这会导致最小的延迟时间变长，但是可以让系统采取更高效的可恢复技术。事实上，一些连续处理系统，比如 TimeStream 和 Borealis [87, 18], *同样*也会将消息记录延迟，这是为了执行那些具有多个上游消息流的确定操作（通过等待流中的周期性“断点”）以及确保系统的一致性。这导致了延迟时间由过去的毫秒级变为 D-Streams 里面的秒级。

4.4 系统架构

我们已经在“Spark Streaming”上实现了 D-Stream，它是基于 Spark 处理引擎的一个修改版本。Spark Streaming 由三部分组成，如图 4.6 所示

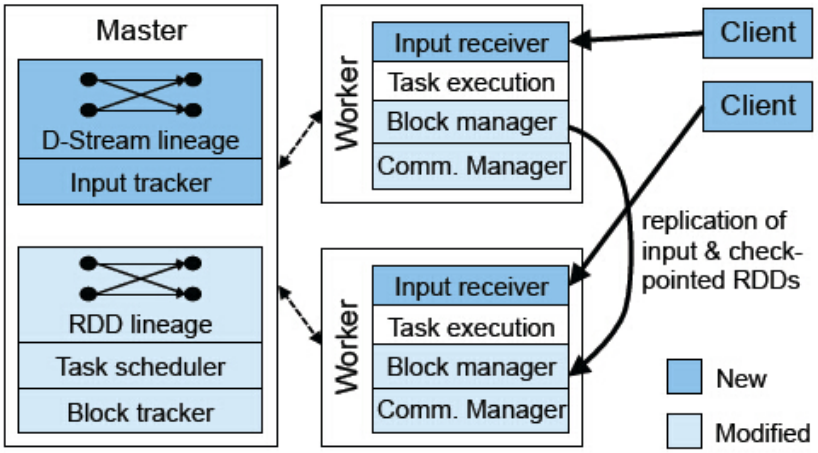


图 *4.6.*Spark Streaming 组件，显示了我们在 Spark 原版本上所作的修改。

- *master* 跟踪 D-Stream lineage，并调度*任务*来计算新的 RDD 分区。
- *工作节点*接收数据，保存输入分区和已计算的 RDD，并执行任务。
- *客户端*用于发送数据给系统。

如图中所示，Spark Streaming 重用了 Spark 的许多组件，但仍然需要修改和添加多个组件来支持流处理。这些变化将在第 4.4.2 节讨论。

从架构角度来看，Spark Streaming 和传统的流系统之间区别在于，Spark Streaming 将计算过程分解为小的、无状态的、确定的*任务*。每个任务都可以在集群中的任何节点或同时在多个节点运行。在传统系统的固定拓扑结构中，将部分计算过程转移到另一台机器是一个很大的动作。Spark Streaming 的做法，可以非常直接地在集群上进行负载均衡，应对故障或启动慢节点恢复。同理也能用于批处理系统——如 MapReduce。然而，由于 RDD 运行于内存中，Spark Streaming 的任务执行时间会短得多，一般只有 50-200 毫秒。

不同于以前系统将状态存储在长时间运行的处理过程中，Spark Streaming 中的所有状态都以容错数据结构（RDD）来保存。由于 RDD 分区被确定性地计算出来，它可以驻留在任何节点上，甚至可以在多个节点上进行计算。这套系统试图最大限度地提高数据局部性，同时这种底层的灵活性使得推测执行和并行恢复成为可能。

这些优势在批处理平台（Spark）上运行时可以很自然地获得。但依然需要进行显著的修改来支持流处理。在介绍这些修改之前，先讨论任务执行的更多细节。

4.4.1 应用程序执行

Spark Streaming 的应用从一个或多个输入流开始执行。系统加载数据流的方式，要么是通过直接从客户端接收记录数据，要么是通过周期性的从外部存储系统中加载数据，如 HDFS，外部的存储系统也可以被日志收集系统[9]所代替。在前一种方式下，由于 D-Streams 需要输入的数据被可靠地进行存储来重新计算结果，因此我们需要确保新的数据在向客户端程序发送确认之前，在两个工作节点间被复制。如果一个工作节点发生故障，客户端程序向另一个工作节点发送未经确认的数据。

所有的数据在每一个工作节点上被一个*块存储*进行管理，同时利用主服务器上的跟踪器来让各个节点找到数据块的位置。由于我们的输入数据块和我们从数据块计算得到的 RDD 的分区是不可变的，因此对块存储的跟踪是相对简单的：每一个数据块只是简单的给定一个唯一 ID，并所有拥有这个 ID 的节点都能够对其进行操作（*例如*,如果多个节点同时计算它）。块存储将新的数据块存储在内存中，但会以 LRU 策略将这些数据块丢弃，这在后面会进行描述。

为了确定何时开始一个新的时间周期，我们假设各个节点通过 NTP 进行了时钟同步，并且在每一个周期结束时每一个节点都会向主服务器报告它所接收到的数据块 IDs。主服务器之后会启动任务来计算这个周期内的输出 RDDs，*不需要*其他任何同步。和其他的批处理调度器一样[61]，一旦完成上个周期任务，它就简单地开始每个后续任务。

Spark Streaming 依赖于每一个时间间隔内 Spark 现有的批处理调度器，并加入了像 DryadLINQ[115]系统中的大量优化：

- 它对一个单独任务中的多个操作进行了管道式执行，如一个 *map 操作*后紧跟着另一个 *map 操作*。
- 它根据数据的本地性对各个任务进行调度。
- 它对 RDD 的各个*划分*进行了控制，以避免在网络中数据的 shuffle。例如，在一个 *reduceByWindow* 的操作中，每一个周期内的任务需要从当前的周期内*“增加”新的部分结果*（例如，每一个页面的点击数），和“删除”多个周期以前的结果。调度器使用相同的方式对不同周期内的状态 RDD 进行切分，以使在同一个节点的每一个 key 的数据(*例如*, 一个页面）在各时间分片间保持一致。更多的细节见 2.5.1 节。

4.4.2流处理优化

尽管 Spark Streaming 建立在 Spark 之上，我们仍然必须优化这个批处理引擎以使其支持流处理。这些优化包括以下几个方面：

网络通信：我们重写了 Spark 的数据层，通过使用异步 I/O 使得带有远程输入的任务，比如说 reduce 任务，能够更快地获取它们。

时间间隔流水线化：因为每一个时间间隔内的任务都可能没有充分地使用集群的资源(*比如说*，在每一个时间间隔的末端，可能只有很少的几个任务还在运行)，所以，我们修改了 Spark 的调度器，使它允许在当前的时间间隔还没有*结束*的时候调用下一个时间间隔的任务。例如，考虑我们在表 4.3 提到的 *map* + *runningReduce* 作业。我们之所以能够在时间间隔 1 的 reduce 操作结束之前就可以执行时间间隔 2 的 map 操作，这是因为每一步的 map 操作都是独立的。

任务调度：我们对 Spark 的任务调度器做了大量的优化，比如说手工调整控制消息的大小，使得每隔几百毫秒就可以启动上百个任务的并行作业。

存储层：为了支持 RDDs 的异步检查点和性能提升，我们重写了 Spark 的存储层。因为 RDDs 是不可变的，所以可以在不阻塞计算和减慢作业的情况下通过网络对 RDDs 设置检查点。在可能的情况下，新的数据层还会使用零拷贝。

lineage 截断：因为在 D-Streams 中 RDDs 之间的 lineage 可以无限增长，我们修改了调度器使之在一个 RDD 被设置检查点之后删除自己的 lineage，修改之后 RDDs 之间的 lineage 不能任意生长。类似地，对于 Spark 中的其他无限增长的数据结构来说，将会定期调用一个清理进程来清理它们。

Master 的恢复：因为流应用需要不间断运行 7 天 24 小时，我们给 Spark 加入对 master 状态恢复的支持(4.5.3 节)。

有趣的是，针对流处理所做的优化还提高了 Spark 在批处理标准测试上的性能，大概是之前的 2 倍。Spark 的引擎能够同时应对流处理和批处理，这是其强大之处。

4.4.3内存管理

在我们当前的 Spark Streaming 实现中，每个结点的块存储管理 RDD 的分片是以 LRU(最近最少使用)的方式，如果内存不够会依 LRU 算法将数据调换到磁盘。另外，用户可以设置最大的超时时间，当达到这个时间之后系统会直接将旧的数据块丢弃而不进行磁盘 I/O 操作（这个

超时时间必须大于检查点间隔的时间)。我们发现在很多应用中，Spark Streaming 需要的内存并不是很多，这是因为一个计算中的状态通常比输入数据少很多(很多应用是计算聚合统计)，并且任何可靠的流式处理系统都需要像我们这样通过网络来复制数据到多个结点。但是，我们还是会计划探索优化内存使用的方式。

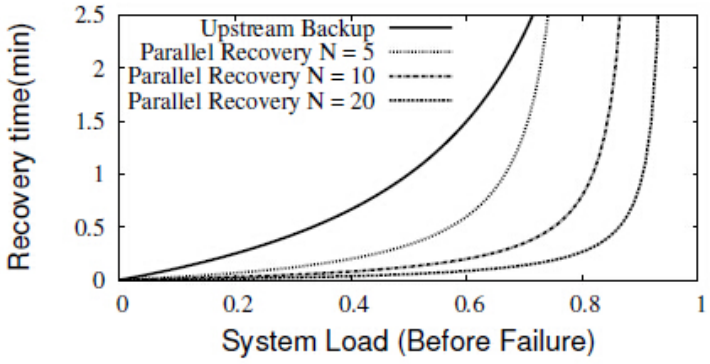


图 4.7.以失败之前的系统负载为函数，对比单点上行流恢复和 N 个节点并行恢复的恢复时间，我们假定自上次检查点为 1 分钟。

4.5 故障和慢节点恢复

D-Streams 的确定性使得可以使用两种有效却不适合常规流式系统的恢复技术来恢复工作节点状态：并行恢复和推测执行。此外，它也简化了主节点的恢复，我们接下来会讨论。

4.5.1 并行恢复

当一个节点失败,D-Streams允许节点上RDD分片的状态以及运行中的所有任务能够在其它节点并行地重新计算。通过异步地复制RDD状态到其它的工作节点，系统可以周期性地设置RDDs状态的*检查点*.[15] 例如，在运行时统计页面浏览数的程序中，系统可能对于该计算每分钟选择一个检查点。然后，如果一个节点失败了，系统会检查所有丢失的RDD分片，然后启动一个任务从上次的检查点开始重新计算。多个任务可以*同时启动* 去计算不同的RDD分片，使得整个集群参与恢复。如 4.3,节所述， D-Stream在每个时间片中并行地计算RDDs的*分区* 以及并行处理每个 *时间片* 中相互独立的操作(*例如*开始的*map*操作),因为可以从lineage中细粒度地获得依赖关系。

为了展示并行恢复的优点，图 4.7 使用一个简单的分析模型将它和单点上行流备份进行了比较。该模型假定系统从一分钟之久的检查点恢复。

在上行流备份中，单个闲置机器执行了所有的恢复，然后开始处理新的记录。在高负荷的系

[15]由于 RDDs 是不可变的，检查点不会阻止当前时间片的执行。

统中这需要很长时间才能跟上进度，这是因为在重建旧的状态过程中新的记录会持续到达。事实上，假设在失败之前的工作量是 λ，然后在恢复的每分钟中备份节点只能做一分钟的工作，但是会同时收到 λ 分钟的新任务。因此，要在 t_{up} 的时间内从上次失败结点中完全恢复 λ 个单元的任务，则可以得到：tup*1=λ+tup*λ.

$$t_{\text{up}} = \frac{\lambda}{1-\lambda}.$$

在其它线路中，所有的机器参与恢复，同时也处理新的记录。假定在失败前集群中有 N 台机器，剩余的 $N-1$ 台机器，现在每个机器需要恢复 A/N 个工作，同时接收数据的速率是 A. 它们追赶到来的数据流时间 tpar 满足 tpar • 1 =λ/N+tpar*Nλ/(N-1)

$$t_{\text{par}} = \frac{\lambda/N}{1-\frac{N}{N-1}\lambda} \approx \frac{\lambda}{N(1-\lambda)}.$$

因此，拥有更多的节点，并行恢复能够跟上到来的数据流，这比上行流备份要快得多。

4.5.2减缓慢结点的影响

除了节点故障，在大型集群中另一个值得关注的问题是运行较慢的节点[36].幸运的是，D-Streams 同样也可以让我们像批处理系统那样减少较慢节点的影响，这是通过推测性（speculative）地运行较慢任务的备份副本实现的.这种推测执行在连续的处理系统中可能很难实现，因为它需要启动一个结点的新副本，填充新副本的状态，并追赶上较慢的副本。事实上，流式处理中的复制算法，比如 Flux 和 DPC [93,18]，主要在于研究两个副本之间的同步 。

在我们的实现中，我们使用了一个简单的阈值来检测较慢的节点：如果一个任务的运行时长比它所处的工作阶段中的平均值高 1.4 倍以上，那么我们标记它为慢节点。我们将来也可能会采用更精细的算法，但是我们看到目前的方法仍然工作的很好，它能够在 1 秒内从较慢节点中恢复过来。

4.5.3Master 恢复

7*24 运行Spark Streaming的一个最终要求是能够容忍Spark master的故障。我们通过两个步骤来做到这些,第一步是当开始每个时序时可靠地记录计算的状态,第二步是当旧的master

失败时，让计算节点连接到一个新的master并且报告他们的RDD分区。D-Streams简化恢复的一个关键方面是 *如果一个给定的RDD被计算两次是没有问题的*。因为操作是确定的，这一结果与从故障中进行恢复类似。[16] 因为任务可以重新计算，这意味着当master重新连接时丢掉一些运行中的任务也是可以的。

我们目前的实现方式是将 D-Stream 元数据存储在 HDFS，记录(1) 用户的 D-Streams 图以及表明用户代码的 Scala 的函数对象，

(2) 最后的检查点的时间，还有(3) 自检查点开始的 RDD 的 ID 号，其中检查点通过在每个时序进行重命名（原子操作）来更新 HDFS 文件。恢复后，新 master 会读取这个文件找到它断开的地方，并重新连接到计算节点，以便确定哪些 RDD 分区是在内存中。然后再继续处理每一个漏掉的时序。虽然我们还没有优化恢复处理，但它是相当快了，100 个节点的集群可以在 12 秒内恢复。

4.6 评估

我们通过使用多个基准测试应用程序和移植两个实际应用来评估 Spark Streaming：商业视频分发监控系统和根据汽车的 GPS 数据估算交通状况的机器学习算法。从这两个案例中，我们也将看到 D-Stream 对批处理任务的良好结合能力--这点会在后文中予以详细讨论。[57]

4.6.1 性能

我们借助三个复杂度依次增加的应用来测试系统性能：1. Grep，在输入字符串中查找匹配模式的数量；2. WordCount，执行超过 30 次的滑动窗口计数；3. TopKCount，查找 K 个（超过 30 次）最频繁出现的词。后两种应用使用增量 *reduceByWindow* 算子。我们首先展示 Spark Streaming 的原始扩展性能，随后将其与 Yahoo! S4 和 Twitter Storm[78,14]这两种广泛使用的数据流系统进行比较。这些应用运行于 Amazon EC2 的 m1.xlarge 级节点上，每个节点都带有 4 个 CPU 核和 15GB 内存

图 4.8 表明，Spark Streaming 能在保持端到端延迟低于限定目标的同时，持续地最大化系统吞吐量。这里所说的“端到端延迟”是指记录从系统发出起到得出相应的结果所需的时间。因此，该延迟包括了等待一个新的输入批次开始的时间。对 1 秒的延迟目标，我们使用 500 毫秒的时间间隔输入，而对于一个 2 秒的目标，我们使用 1 秒的时间间隔。在这两种情况下，我

[16] 现在有一个微妙的问题是输出操作符；我们已经设计出诸如*保存*的幂等操作符，它会把每一时序的值输出到一个已知的路径，而且如果该时序已经计算出来，也不会覆盖掉先前的数据。

们使用 100 字节的输入记录。

我们看到，Spark Streaming 可以以近似线性的特性扩展到 100 个节点上。对于 Grep 应用，在 100 节点集群上以亚秒级时延可以处理多达 6 GB/s（64M 记录/s）。而对于其他更加消耗 CPU(CPU 密集型)的作业，也可以达到 2.3 GB/s（25M 记录/s）。若提高可延迟的时间，系统的吞吐量也只是稍有上升。事实上，亚秒级延迟时系统性能已经很高了。

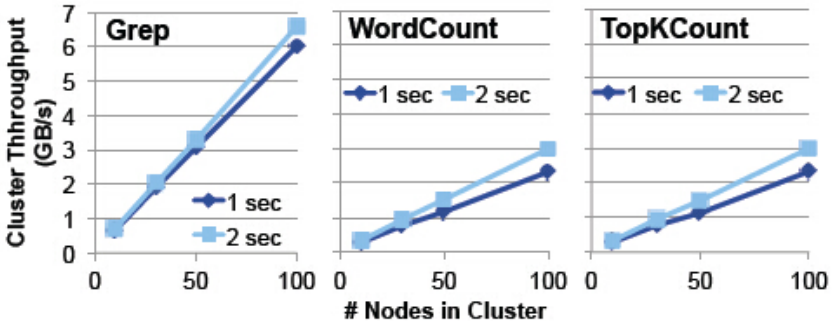


图 4.8。在给定延迟限制（1 秒或 2 秒）下，Spark Streaming 可达到的最大吞吐量。

与商用系统的对比： Spark Streaming 的单点吞吐量在 Grep 上为 640,000 条记录/秒，四核环境下 TopKCount 任务为 250,000 记录/秒。这样的性能能与其他商业化的单节点流系统已公开的性能相媲美。例如，Oracle CEP 公布的在 16 核机器上的吞吐量为 100 万条记录/秒[82]，而 StreamBase 公布的在 8 核环境下为 245,000 条记录/秒[105]，Esper 在四核环境下公开的吞吐量为 500,000 记录/秒[38] 。没有理由期望 D-Streams 在每个节点上要更慢或更快，但 Spark Streaming 主要优势在于其性能可以近似线性伸缩到 100 个节点。

与 S4 和 Storm 的对比： 我们还把 Sparkstreaming 同两个开源的分布式流系统，S4 和 Storm，进行了比较。两者都是连续操作的系统，它们不提供节点间的一致性保证，而且容错能力有限(S4 没有，而 Storm 保证记录至少有一次成功交付)。我们在这两个系统中编码实现上述三个应用，但发现 S4 上单节点每秒可处理的记录数有限（对 Grep 每秒最多 7500 条记录，WordCount 每秒最多 1000 条记录），这使得它比 Spark 和 Storm 至少慢 10 倍。因为 Storm 更快，我们在 30 个节点的集群上，同时使用 100 字节和 1000 字节的记录，对它进行了测试。

图 4.9 对 Storm 和 Spark 进行了比较，其中对于 Spark 是采用亚秒级时的吞吐量。从图中可以看出，记录较小时 Storm 的性能会降低。对于 100 字节的记录，Storm 只达到 115K 条记录/秒/节点，而 Spark 是 670K。这样的结果还是在我们对 Storm 实现进行过性能优化后得出的。这些优化包括：1. Grep 的实现中，对输入记录采用每 100 条批量方式发送；2. WordCount 和 TopK 的对应实现中，是按秒钟来发送新的计数，而不是每次记录发生改变就发送。在 1000 字节的记录上，Strom 的速率有所提升，但仍慢 Spark2 倍。

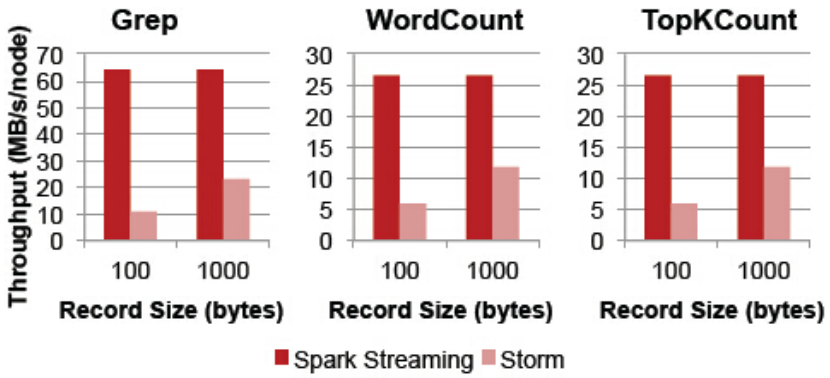


图 4.9 30 节点环境下与 Storm 在吞吐量上的对比。

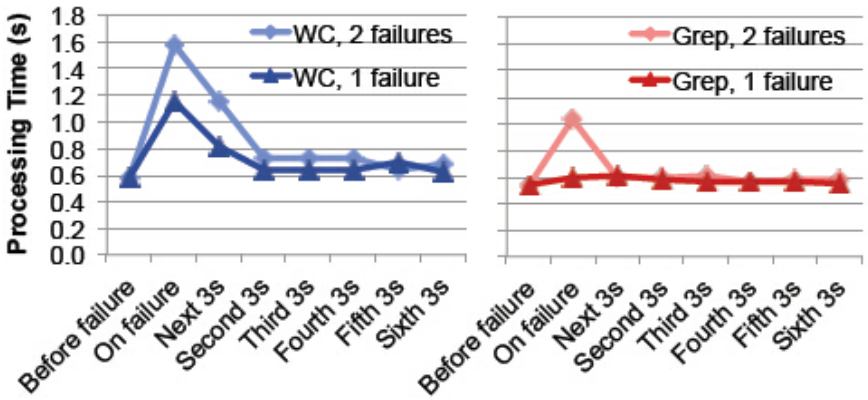


图 4.10 WordCount （WC）和 Grep 任务的故障处理时间间隔。这里展示了处理 1 秒钟的批量数据时，分别在故障前、故障中和故障后 3 秒内所需的平均耗时。平均值取自 5 次运行的结果。

4.6.2 故障和慢节点恢复

对故障恢复能力的评估，将从系统对单词计数(WordCount)和查找(Grep)两个应用在不同条件下的表现展开。输入采用的是秒级批处理数据，其原始数据存储在 HDFS 上。另外，数据传输速率的设定单词计数(WordCount)和查找(Grep)任务分别为 20MB/s/节点和 80MB/s/节点。这样的设定能使得单词计数(WordCount)和查找(Grep)任务的每次处理的时间近乎相同，具体分别为 0.58s 和 0.54s。单词计数(WordCount)任务执行一个增量式的*按键聚合（ReduceByKey）的*操作，这将使得它的 lineage 不确定的增长（因为每次处理都会去除过去 30 秒的数据）。鉴于此，这个任务会引入一个步幅为 10 秒的检查点操作。测试的平台是由 20 个四核节点构成的集群。每个作业都使用 150 个 Map 任务和 10 个 Reduce 任务。

首先，通过图 4.10， 我们呈现了在上述基本条件下恢复时间的情况。该图展现对于 1 个或 2 个并发失败时，系统在故障前、故障中和故障后 3 秒内，对单秒间隔窗口期数据的平均处理

时间。(在数据恢复时，这些后续处理因为受那些失败的时间区间影响而被延迟，所以我们接下来要展示系统如何再进入稳定状态。)从中可以看到，恢复速度比较理想：即便对于两次失败和 10 秒的检查点串口情况下，也最多只有 1 秒的延迟。单词计数(WordCount)任务中的恢复耗时相对较长。这是因为每次失败时，它需要追溯 lineage，从更加原始的数据开始重新计算。而查找(Grep)任务则与之相反，每个失败节点上它只丢失了四个任务。

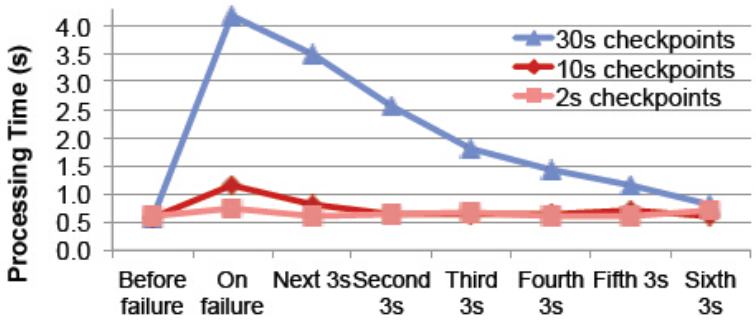


图 4.11 WordCount 任务中，不同检查点时间间隔窗口下的恢复耗时。

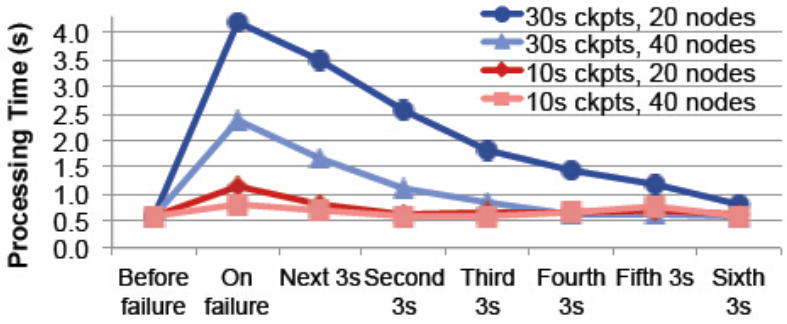


图 4.12 在 20 和 40 个节点集群上 WordCount 的恢复耗时。

调整检查点时间窗口　图 4.11 展示了不同检查点时间窗口对 WordCount 的影响。即便检查点窗口为 30 秒时，结果最多也就延迟 3.5 秒。窗口设定为 2 秒时，系统恢复需时仅为 0.15 秒--仍然快过全备份的策略。

调整节点数据　为评估并行程度对系统的影响，我们同样在一个 40 节点的集群上测试了 WordCount 应用。如图 4.12 所示，节点数目增加一倍会使得恢复时间减少一半。

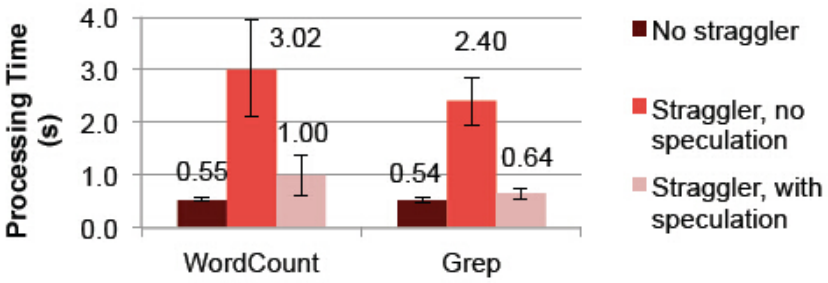


图 4.13 在延迟的情况下，花在常规操作上的 Grep 和 WordCount 的处理时间间隔，预测和非预测

慢节点恢复　最后，我们通过启动 60 个线程来过载 CPU 而不是直接移除的方式，来模拟出一个慢节点。图 4.13 分别展示了无慢节点时、有慢节点且禁用推测执行(备份任务)时，以及有慢节点且推测执行时，系统的单次处理耗时。预测执行能显著改善相应时间。

需要注意的是，我们的编码实现中并不记录执行过程中的慢节点有哪些，这样，性能的改善是在可能会多次在慢节点上发起新任务的情况下得到的。这就表明，即便是意外出现的慢节点也能被系统快速处理。一个比较完整的实现是将那些慢节点加入黑名单。

4.6.3 实际应用

我们借助两个实际应用来评估 D-Streams 的表达能力。这两个应用的复杂度都明显高于迄今所示的所有测试，并且它们都利用 D-Streams 来进行批处理、交互式处理以及流处理。

视频分发监控

Conviva 推出了一款用于视频在因特网上分发的商业管理平台。该平台中的一个功能是能实现不同地理区域、CDN、客户设备和 ISP 下的性能跟踪。这使得广播商能快速发现视频分发过程中的问题，并进行响应。系统从视频播放器接受事件信息，并利用它来计算不同分类下的 50 种以上的指标。这些指标包括如观众数，以及某次播放的视频缓冲率这样的复杂指标。

其当前的应用由两部分构成：一个定制的分布式流式系统用于处理实时数据，以及一个基于 Hadoop/Hive 的历史数据和即时(ad-hoc)查询支持系统。由于客户会想回调到某个历史时刻来进行系统调试，故同时支持实时数据和历史数据很重要。然而，这也增加了在这两个独立的系统之上来实现某个应用的挑战性。首先，两系统之间必需保持同步，以保证他们计算指标

（Metrics）的方式相同。其次，在数据导入 Hadoop 转成可供即时（ad-hoc）查询的形式的过程中，会有数分钟的迟延。

通过对 Hadoop 版本下相应的 *Map* 和 *Reduce* 函数进行封装，我们将上述功能移植到了 D-Streams 上。移植的实现包括一个 500 行的 Spark Streaming 程序和一个 700 行代码的额外的封装器。该封装器能让 Hadoop 程序在 Spark 下执行。

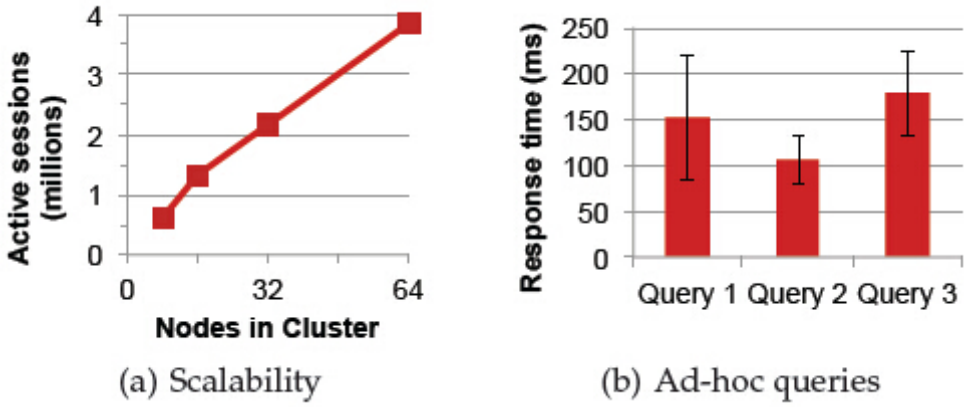


图 4.14。视频应用程序的结果，（a）显示支持的客户端会话数目 vs 集群大小，（b）显示 3 个从 Spark Shell 程序进行即席查询的性能：（1）所有的活动会话；（2）特定用户的会话；（3）已出现故障的会话。

从而所有的 Metrics(由两个 MapReduce 任务构成)可以在短短 2 秒内计算完。代码的实现使用了 4.3.3 章节所述的 *updateStateByKey* 操作，来对每一个客户端 ID 构造一个会话状态对象，并在收到事件时对该对象进行更新。而在该操作之后，会有一个滑动的 *reduceByKey* 过程 来对多个会话的 Metrics 进行聚合。

对该应用的可伸缩性测试表明，在 64 节点 4 核 EC2 集群下，其能处理足够多的事件来支撑 3.8 百万的同时在线观看。而这个量已经超过了 Conviva 现有的峰值记录。伸缩性测试见图 4.14(a)。

此外，我们使用 D-Streams 来添加一个原始应用程序中不存在的 *新* 功能：实时流状态的 ad-hoc 查询。如图 4.14（b），Spark Streaming 可以在不到一秒钟的时间内从 Scala Shell 完成对会话状态的 RDD 进行的 ad-hoc 查询。该集群的内存容量完全可以容下 10 分钟的数据，而这个长度接近历史数据和流处理之间的时间差。同时，人们可以在一套代码上对这两套数据进行处理。

众包交通流量估计

我们将 D-Streams 应用到 *Mobile Millennium* 交通信息系统[57]，该系统是一个基于机器

学习的用来估算城市的汽车交通状况的系统。测量高速公路上的交通很简单，因为高速公路上有专用的传感器，然而在*主干道*（城市里的道路）却且缺乏这类设施。*Mobile Millennium* 利用来自装有 GPS 的汽车*（如，*出租车）和运行特定程序的手机的 GPS 众包数据来解决这个问题。

从 GPS 数据进行流量估算是具有挑战性的，因为 GPS 数据存存有噪声（在高的建筑物附近 GPS 有误差）且呈稀疏特性（系统只从每车每分钟接受一次测量）。*Mobile Millennium* 通过一个计算复杂的期望最大化算法来做条件推断--利用 Markov Chain Monte Carlo 和一个交通流量模型来评估各个路段交通流量随时间的分布情况。

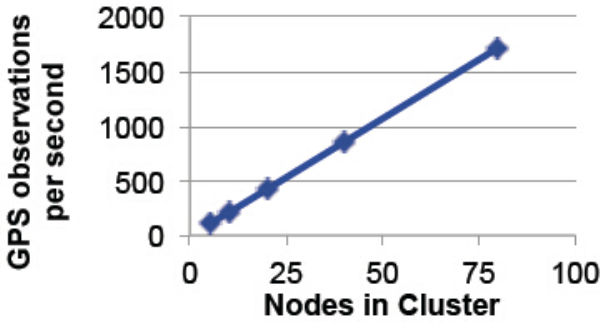


图 4.15 可扩展的 *Mobile Millennium* 作业。

对于每个路径连接。以前的实现[57]是一个迭代式的 Spark 批处理任务，该任务以 30 分钟为时间窗来对数据进行处理。

通过一个 EM 在线算法，我们将该应用移植到了 Spark Streaming。该算法每 5 秒钟生成新的数据。移植的实现为 260 行 Spark Streaming 代码，并封装了上述离线程序中的 *Map* 和 *Reduce* 函数。此外，我们发现 5 秒内的数据过于稀疏，从而可能引发过拟合。所以，我们通过利用 D-Streams 来结合*过去* 10 天里相同时间段数据的方式，来解决这个问题。

图 4.15 展示了 在 80 个四核 EC2 节点上这个算法的性能。由于该算法是CPU密集型，其性能几乎能随节点数线性扩展，同时其计算速度为原批处理版本的 10 倍。[17]

4.7 讨论

前面我们已经介绍了离散流（D-Streams）的相关知识，这是一个用于集群的崭新流处理模型。通过将复杂计算分解成短的，确定的任务和将状态存储在基于 lineage 的数据结构（RDDs）中，D-Streams 能够使用强大的恢复机制，这种恢复机制，有些类似于在某些批处理系统中对

[17]需要注意的是，该算法的原始速率-条/秒，比我们其他的算法更低，因为它的每一条记录都会进行远多于其他算法的工作--每个记录绘制 300 个 Markov Chain Monte Carlo 样本。

于错误和慢任务的处理。

因为批量数据的存在，D-Streams 的主要限制在于它有一个固定的最小延迟。但是，我们已经证明过总的延迟依然能够控制在 1-2 秒内，这对大多数的实际用例来说已经足够了。有趣的是，即使是在一些连续运算系统中，比如说 Borealis 和 TimeStream [18,87]，也会通过增加延迟来保证确定性。Borealis 的 SUnion 运算和 TimeStream^的 HashPartition 运算会在所谓的“heartbeat”边界上等待批量数据，这样一来，对于那些有多个父节点的运算，虽然其父节点的真实输入顺序是不确定的，但是可以将这些输入看作是以一定的顺序输入的。因此，D-Streams 的延迟在一定程度上和这些系统是类似的，但是 D-Streams 能够提供更高有效的恢复机制。

除了能够提供有效的恢复机制之外，我们认为 D-Streams 最重要的作用体现在它证明了流处理，批处理和交互式计算可以统一在同一平台上。由于大数据正成为某些应用可以操作的*唯一*数据规模（*例如*大型网站的垃圾邮件检测），一些组织将会在这些数据上使用这些工具来实现低延迟应用以及交互应用，而不仅仅批处理应用会用到这样规模的数据。D-Streams 深度整合了这些计算模式，这些计算模式不仅仅是调用相似的 API，还包括使用相同的数据结构和容错模型。这也使得使用 D-Streams 的系统能够具备丰富的特性，像流与离线数据的结合和在流状态上执行 ad-hoc 查询。

最后，虽然我们给出了 D-Streams 的一个基本实现，但是未来还有几个方面需要完善。

表现力：一般来说，既然 D-Streams 主要是一个*执行*策略，通过简单地把算法的执行划分成批处理步骤然后在这些步骤中间发送状态，就应该能够运行大多数的流数据算法。在 D-Streams 上调用流 SQL[15]和复杂事件处理模型[39]将会是一件很有意思的事情。

设置批处理间隔：给定任何一个应用，设置一个合适的批次间隔是非常重要的，因为这个批处理间隔直接决定了端对端的延迟与整个流负载吞吐量之间的权衡。目前来说，开发人员必须自己探索这个权衡并且手动确定批处理间隔。未来有可能实现系统自动调整。

内存使用：每处理完一批数据，我们的状态流处理模型就会生成一个新的 RDD 来存放每个运算的状态。在我们目前的实现中，相比于使用可变状态的连续运算，它会使用更多的内存。为了让系统能够执行基于 lineage 的错误恢复，必须要存储状态 RDDs 的不同版本。但是，可以通过存储不同状态 RDDs 之间的变化量来达到减少内存使用的目的。

检查点和容错策略：因为检查点的成本很高，所以选择一个合适的频率对每一个 D-Stream 自动设置检查点是很有价值的。另外，除了检查点之外，D-Streams 还允许使用大量的其他容错策略，比如说计算的*部分复制*，在部分复制中，任务的一个子集被复制（*例如*， 我们在 4.6.2 节中复制的 reduce 任务）。自动应用这些策略也是一件有趣的事。

近似的结果：除了重新计算丢失的工作，另外一种处理失败的方式就是返回近似的部分结果。

通过在父节点全部结束之*前*启动一个任务，并且提供用来推断*哪些*父节点丢失了的lineage 数据，D-Streams 提供了一个计算部分结果的机会。

4.8 相关工作

流数据库： 流数据库像 Aurora、Telegraph,、Borealis 和 STREAM[23, 26, 18, 15]是最早用于研究流数据的学术系统，率先提出来了像窗和增量操作等概念。然而，像 Borealis 等*分布式*流数据库使用复制或上行流备份来支持恢复功能[58]。我们在它们基础上提出了两点贡献。

首先，D-Streams 提供更有效的恢复机制，即并行恢复，它比上行流备份运行更快，并且没有复制开销。因为 D-Streams 将计算离散化为无状态且确定的任务，因此并行恢复是可行的。相比之下，流式数据库使用状态连续操作模型，因此需要为复制 (*e.g.*, Borealis's DPC [18] or Flux [93])和上行流备份[58]定制复杂的协议。我们所知的唯一并行恢复协议由 Hwang 等人提出[59]，但只能容忍单节点故障，并且不能处理慢任务。

其次，D-Streams 使用推测执行[36]可以容忍慢任务。慢节点恢复对连续操作模型非常难处理，因为每个节点状态可变，除非有耗时的序列重建过程，否则无法重建。

大规模流： 虽然最近一些系统使用类似 D-Stream 的高级别 API 支持流计算，但是它们缺少离散流模型的故障恢复和慢任务恢复的优势。

TimeStream [87]在微软 StreamInsight 的集群上[3]运行连续状态操作。它使用类似上行流备份的恢复机制，跟踪每个操作依赖哪个上行流数据，并且通过新的复制的操作顺序重放来进行恢复。因此对每个操作的恢复在单节点上发生，并且按操作处理窗的时间比例执行*(e.g.*，30 秒的滑动窗对 30 秒) [87]。相比之下，D-Streams 使用无状态变换，并明确把状态存放在数据结构（RDDs）中，这样可以

（1）异步的被设置检查点以限定恢复时间； (2) 并行重建，利用跨数据分区的并行处理和时间步长在亚秒级进行恢复。D-Stream 还能处理慢任务，但 TimeStream 不行。

Naiad [74, 76]用 LINQ 语言实现自动增量数据流计算并且能够*迭代增量*计算。然而，它使用传统的同步检查点进行容错，并且不能处理慢任务。

MillWheel [2]使用事件驱动 API 运行状态计算，但是通过保存所有状态到复制存储系统来保证可靠性，类似 BigTable。

MapReduce Online[34]是流式 Hadoop 运行时，它在 maps 和 reduces 之间推送数据，并且使用上行流备份保证可靠性。然而，它不能恢复有长期存在的 reduce 任务(用户必须手动执行对这些状态设置检查点并将数据放到外部系统中)，并且不能处理慢任务。Meteor Shower [110]也使用上行流备份，并且能够 10 秒左右恢复数据。iMR [70]为日志处理提供 MapReduce API，但可能在故障时丢失数据。Percolator [85]使用触发器运行增量运算，但是不能提供高级别操

作，像*映射*和*连接*.。

最后，据我们所知，除了 D-Streams，这些系统几乎都不支持批量和任意查询的*合并*流操作。一些流数据库支持将表和流合并 [43]。

消息队列系统 像 Storm, S4, 和 Flume 等系统[14, 78, 9] 提供消息传递模型，用户写状态码来处理记录，但是它们通常只有有限的容错保证。例如，Storm 确保使用上行流备份在源端“至少一次”的*消息*传递，但是要求用户手动处理*状态*的恢复，*例如，*通过将所有状态保存在备份的数据库中[101]。Trident [73]类似 LINQ，在 Storm 上层提供函数 API，自动管理状态。然而，Trident 通过将所有状态存储在另一个备份数据库中来提供容错机制，但这种方式成本很高。

增量处理 CBP[69]和 Comet [54]”在传统的 MapReduce 平台上提供“块式增量处理，通过每隔几分钟在新的数据上运行 MapReduce 作业实现。虽然这些系统*从*每个时间步的 MapReduce 扩展性和容错/慢任务中获益，但是它们复制和磁盘文件系统保存所有状态，从而导致高开销和几十秒到几分钟到延迟。相比之下，D-Streams 能够使用 RDD 在内存中保存未复制的状态，并且能够使用 lineage 按时间恢复它，产生低几个数量级的延迟。Incoop [19]改进 Hadoop 来支持输入文件更改时作业的增量重算输出，并且包含慢任务机制恢复，但在时间步长内它仍然使用备份磁盘存储，并且不提供显示流接口（类似 windows 的概念）上。

并行恢复： SEEP [24]是最近对流操作增加并行恢复的一个系统，它允许连续操作通过标准 API 显示和分离它们的状态。然而，SEEP 需要对 API 侵入式重写每个操作，并且不支持慢任务。

我们的并行恢复机制也类似与 MapReduce，GFS 和 RAMCloud [36, 44, 81]，它们都是对故障进行分区恢复。我们的贡献是如何跨数据分区和时间来构造流计算以实现这套机制，并且能够在一个足够小的时间尺度内实现流处理。

4.9 总结

我们提出了 D-Streams，一个用于分布式流计算的新模型，由于没有复制开销，能够快速地从故障和慢任务（straggler）中恢复过来（通常是不到一秒的时间）。D-Streams 不同于传统的流式设计，*它在每个时间片内进行批处理*。通过利用数据分区和时间上的并行性，它提供了高效的恢复机制。我们发现 D-Streams 可以实现丰富的操作，以及单节点高吞吐量，并且可以线性扩展到 100 个节点，达到亚秒级延迟和亚秒级故障恢复。最后，因为 D-Streams 使用的是与批量处理平台相同的执行模型，所以它们能无缝地集成批量处理和交互式查询。我们在 Spark

Streaming 中使用过这种功能，让用户用一些强大的方法将这些模型结合起来，并在两个实际应用中展示如何添加一些丰富的功能。

Spark Streaming 是开源的，并且目前已包含在 Apache Spark 项目中。该代码可以在 http://spark.incubator.apache.org. 中获取。

第五章 RDD 的通用性

5.1 简介

前面的四个章节涵盖了 RDD 模型和多个实现以前特定计算类型的 RDDs 应用程序。对模型进行一些相对简单的优化，以及通过降低延迟，使得 RDD 可以匹配特定系统的性能，同时提供更有效的组合。然而，问题依然存在：RDDs 为什么如此通用？为什么它们能够接近特定系统的性能？模型的瓶颈是什么？

在这一章中，我们从两个角度通过探索 RDDs 的通用性来研究这些问题。首先，从表述的观点看，RDDs 可以模拟*任何*分布式系统，并且在大多数情况下这样做都是 *高效的*，除非系统对网络延迟非常敏感。特别是，增加了数据共享的 MapReduce 使得这个模拟更高效。第二，从系统的角度来看，在集群环境中 RDDs 能给应用程序对常见的资源瓶颈加以控制（特别是网络和存储 I/O），这使得应用程序能够表达那些特定系统所具有的资源优化，并因此达到相似的性能。最后，RDDs 通用性的探索也决定了模型的一些局限性，即它可能不能有效地模仿其他分布式系统并导致一些扩展性方面的问题。

5.2 观点描述

为了从理论角度描述 RDDs，我们首先拿 RDDs 和从其派生和借鉴所得的 MapReduce 模型进行比较。MapReduce 最初是被用于大规模集群的计算，从 SQL[104]到机器学习 [12]，但是逐渐被其他特定系统取代。

5.2.1 MapReduce 所能涵盖的计算范围

我们关注的第一个问题是MapReduce本身能够表达哪些计算。尽管已经有很多关于MapReduce局限性的讨论，并且也已经有很多系统扩展其功能，但是令人惊奇的答案是*MapReduce可以模仿任何分布式计算任务。*[18]

为了证明这一点，注意任何分布式系统都由执行本地计算和偶尔地进行消息交换的节点组成。MapReduce 提供了 *Map* 操作用来执行本地计算和 *Reduce* 操作用来所有节点间相互通信。这样，通过将计算拆分成多个时间步长的方式，任何分布式系统都能够被模拟(或许有点低效率)，通过运行 Map 任务来执行每个时间步长上的本地计算任务，并在每个时间步长的最后进行消息的打包和交换。一系列的 MapReduce 步骤足以获得整个结果。图 5.1 显示了 这些步骤是怎样执行的。

两方面因素导致了这种模拟的效率低下。第一，就如我们在论文的其他部分讨论的那样，MapReduce 在时间步长间的 *共享数据*的方式是低效的，因为这种共享是基于可复制的外部存储系统。因此，由于每个时间步长都要输出自己的状态，我们模拟的分布式系统可能变得比较缓慢。第二，MapReduce 步骤的*延迟*决定了我们的模拟如何匹配一个真实的网络，并且大多数 MapReduce 的实现是为耗时几分钟到几小时的批量环境设计的。

RDD 架构和 Spark 系统解决了这两方面的限制。在数据共享方面，RDDs 的架构是通过避免复制的方式，使得数据快速共享。并且能够比较贴切地模拟跨越时间的 “内存数据共享”，这是由多个长驻进程组成的分布式系统所实现的。在延迟方面上，Spark 展示了在 100 多个节点组成的商业集群中执行 MapReduce 计算任务，有 100ms 的延迟——没有固有的 MapReduce 模型能够避免这种情况。然而，一些应用程序可能需要更细粒度的时间步长和通信，这样 100ms 的延迟已经足够实现许多数据密集型的计算，而在通信密集的情况下，大量的计算可以被批量执行。

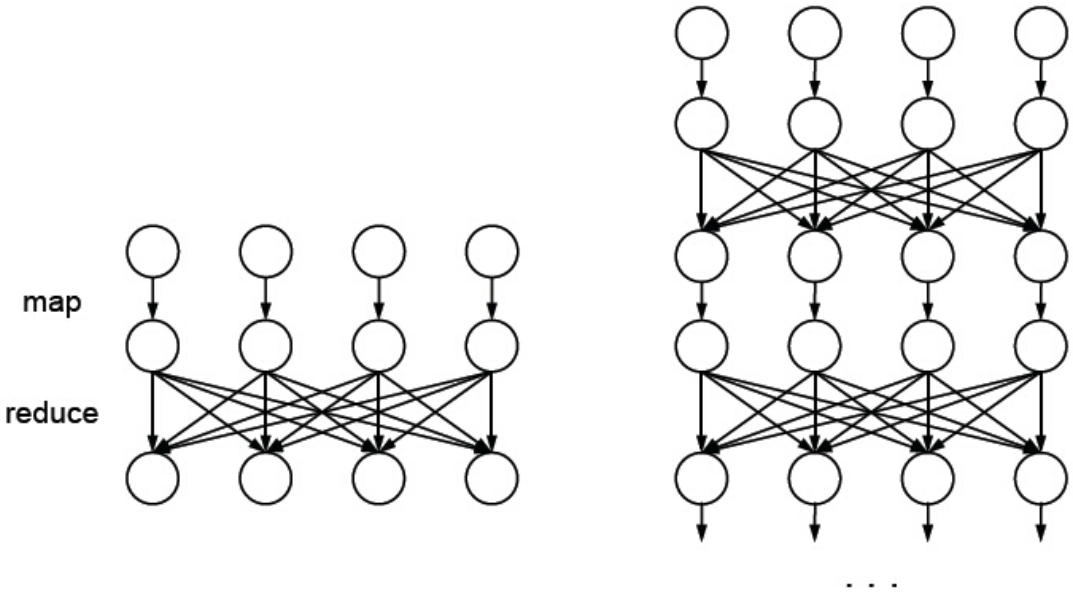
5.2.2 lineage 和故障恢复

上面基于 RDD 模拟的一个有趣特性是它也提供了故障容错。特别的，每个步骤的 RDD 计算仅仅比前面的步骤多一个*常数*大小的继承结构，这意味着存储 lineage 和执行故障恢复的代价很小。

想象一下我们模拟一个分布式系统，它包括多个单节点的处理，系统是通过执行固定数目

[18] 我们尤其关注 work-preserving 的模仿[88]，假设它们都有相同数目的处理器时，模仿机器上的总时间是被模仿机器的常数倍。

的步骤来完成信息的交换。



(a) 单个 MapReduce 步骤　　(b) 多个 MapReduce 步骤的通信

图 5.1。使用 MapReduce 模拟一个任意的分布式系统。如(a)所示，MapReduce 提供了本地计算以及所有结点相互间通信的原语。如(b)所示，通过将这些步骤链接在一起，我们能够模拟任意的分布式系统。这个模拟的主要代价是每一轮的延迟以及步骤之间状态传递的开销。

每个步骤的本地消息处理在“map”函数中循环进行（它的旧状态作为输入，新状态作为输出），然后在时间步长间使用“reduce”函数来进行消息的交换。只要将每个过程中的程序计数存储在它的状态里，这些map和reduce函数在每个时间步骤中是 *一样* 的：它们仅仅读取程序的计数，接受消息和状态，然后模拟执行过程。因此它们能够在常量空间内编码。由于在状态中增加了程序计数，这可能引起一些开销，这通常只是状态的一小部分，并且这些状态只是在节点本地共享。[19]

默认情况下，上面模拟过程的 lineage 可能使得状态恢复代价很高，这是因为每一步增加了一个新的所有节点相互间的 shuffle 依赖。但是，如果应用程序将计算语义表达的更精确*（比如*：说明某些步骤仅仅产生窄依赖），或者将每个节点的工作切分到多个任务，我们就可以在集群间来并行恢复。

[19] 这个方法最易于应用在一旦启动不会再从外部接受输入的系统。如果系统接受输入，我们需要这个输入被可靠的存储及与我们处理离散数据流一样的方式（第四章）来切分为时间片，如果一个系统需要可靠的对输入响应（比如，如果节点挂掉不会丢失任何输入信息），这个需求已经是必要的。

基于 RDD 模拟分布式系统的最后一个问题是在本地维护多个版本状态的代价，这些状态会被转换操作使用，以及为了便于故障恢复而维护对外发送消息的拷贝。这个代价不小，但是在很多应用中，我们可以通过一段时间执行异步的状态检查点（*比如*, 如果检查点的可用带宽比内存带宽低 10 倍, 我们可以在每 10 个步骤执行检查点）或者通过保存多个版本的差异(如 3.2.3 节所述)来限制它.只要每台机器上的“快速”存储足够储存对外发送的消息以及一些版本的状态，我们就可以达到原来系统的性能。

5.2.3与 BSP 的比较

作为关于 MapReduce 与 RDDs 的通用性的第二个例子，我们注意到上述“本地计算和所有结点相互间通讯”模式与 Valian 的批量同步并行模型（BSP）[108]非常吻合。BSP 是一个“ 桥接模型，旨在捕捉真实的硬件上简单却最显着的特性（即通信具有延迟且同步是昂贵的）并对其进行简单的数据分析。因此，它不仅被直接用于设计一些并行算法，而且其成本（即通信的步骤数，每一步中的本地计算量以及每一步骤中各处理器之间通信的数据量）也是大多数并行应用中用来优化的自然因素。因此，我们可以预期与 BSP 吻合的算法都可以用 RDDs 进行有效的评估。

请注意，这个 RDDs 的仿真参数因此也可应用于基于 BSP 的分布式运行时，如 Pregel[72]。RDD 相比与 Pregel 增加了两个好处。首先，Google 论文[72]中描述的 Pregel 只支持‘检查点回滚’的系统错误恢复机制。随着系统规模的扩大，这使得系统扩展的效率降低并且节点失效会变得越来越频繁。该论文中确实介绍了一种还在开发中的‘限制性恢复’模式，它记录下传出的信息并且并行地恢复丢失的系统状态。这与 RDDs 的并行恢复机制类似。第二，因为 RDDs 有一个基于遍历器的接口，它们可以更有效地对不同类库编写的计算流水线化，这对于编写程序是非常有用的。更一般地，从编程接口的角度来看，我们发现 RDD 允许用户使用更高层次的抽象（*例如*, 将状态分割为多个分区数据集或允许用户建立可窄可宽的的依赖模式而不需要在每一步都进行所有结点相互间的通信），同时还提供一个简单通用的接口使数据可以按上述讨论进行共享。

5.3 系统角度

完全不同于仿真的方法来表征 RDD 的特性，我们可以采取一种系统方法：在大多数集群计算中资源的瓶颈是什么，能否用 RDD 来有效的解决这些问题？从这个角度来看，大多数集群应

用最明显的瓶颈是通信和存储。RDD 的分区和本地特性使得应用有足够的控制力来对这些资源进行优化，从而使得在许多应用中达到类似的性能。

5.3.1 瓶颈资源

虽然集群应用是多种多样的，但是它们都受到相同的底层硬件的限制。目前的数据中心有一个非常不合理的存储层次结构，这将会因相同的原因限制大多数应用。例如，现在一个典型的数据中心可能有以下硬件特性：

- 每个节点的本地内存大约有 50 GB / s 的内存带宽以及多个磁盘（通常在 Hadoop 集群中为 12-24，[80]）。也就是说，假设有 20 个磁盘，每个磁盘带宽 100 MB/s，那么将意味着本地存储带宽约为 2 GB/s。
- 每个节点都有一个 10 Gbps (1.3 GB/s) 的网络输出带宽，大约比内存带宽小 40 倍，比它的磁盘总带宽小 2 倍。
- 20-40 台机器节点组成机架，机架间的带宽为 20-40 Gbps，这比机架内部的网络性能要低 10 倍。

鉴于这些特性，许多应用所关心的最重要的性能指标就是控制网络布局和通信。幸运的是，RDDs 提供了这样的条件：其接口在运行时将计算调度在离数据最近的节点，就像 MapReduce 的 Map 任务（其实在 2.4 章节的定义中，RRDs 有一个“优先位置”的 API），并且 RDDs 还提供了数据分区和共存。不像 MapReduce 中的数据共享，总是隐式地需要经过网络传输。而 RRDs 是不会造成网络流量的，除非用户明确调用了一个跨节点操作，或是对数据集设置检查点。

从这个角度看，如果大部分的应用都有网络带宽限制，那么一个节点（*例如,*数据结构或 CPU 开销 ）的本地效率的影响将小于网络通信的效率。以我们的经验，很多 Spark 应用都有带宽限制的，特别是如果数据可以放进内存中。当数据无法放进内存中时，应用受 I/O 限制，并且数据本地性将是最重要的一个因素。CPU 密集型应用通常更容易执行，*（例如,*许多应用在 MapReduce 上也都做得很好）。就像在上一章节的讨论中，RDDs 明显增加成本的地方就是网络延迟，但是 Spark 的工作表明，这种延迟对很多应用来说可能会足够小，甚至小到足够支持数据流。

5.3.2 容错的开销

最后从系统的角度要说明的一点是，由于其容错性，基于RDD的系统产生了一些额外的开销。例如，在Spark中，每个shuffle操作中的“map”任务将他们的输出保存到本地文件系统中，所以之后“reduce”任务可以重复获取。另外，Spark（就像原生的MapReduce）在shuffle阶段执行了一个“barrier”，所以“reduce”任务不会启动，直到所有的map任务完成。相比于直接从map任务以管道的方式直接推送到reduce任务，这简化了容错的复杂性。[20]

尽管移除一些低效率之处会加快系统运行速度，并且我们也打算在以后的工作中这样做，但是即便如此 Spark 依旧性能突出。最主要的原因是前一节中的一个说法：许多应用程序都受 I/O 所限制， *例如,*通过网络传输大量数据，或从磁盘中读取数据，除此之外，如流水线技术仅增加了一个少量的改进。例如，可以考虑从 map 任务直接推送数据到 reduce 任务，而不是等待所有的 map 任务执行完再调度 reduce 任务：最理想情况下，如果 map 任务的 CPU 计算时间刚好与网络传输时间重叠，那么这将使速度加快 2 倍。当然，这是一个非常有用的优化，但不像将 map 任务调度到数据所在节点或者避免中间状态的复制那么重要。此外，如果运行过程中其他部分占主要开销（*例如,*map 任务花费很长时间从磁盘中读取，或 shuffle 过程比计算时间慢很多），那么效益就会降低。

最后我们注意到，即使故障*并* 不经常发生，Spark 和类 MapReduce 的设计将任务划分成细粒度的独立的任务会有其他的好处。首先，它可以缓解慢节点（straggler）问题，这在传统的基于推送的数据流设计中显得异常复杂。(类似于我们第 4 章中比较 D-streams 和连续操作). 甚至在较小的集群上，慢节点问题也比故障更常见，尤其是在虚拟化环境。[120].其次，独立的任务模式有利于*多用户管理：*来自不同用户的多个应用程序可以动态共享资源，从而实现多用户的交互执行。我们大部分并行编程模式的工作，已经使动态资源在集群用户之间进行共享，大型群集必然有很多用户，并且在这些集群中所有的应用都需要实现快速定位和数据本地化[117, 56,116].基于这些原因，一个细粒度的任务设计可能会让*大多数用户*在多用户环境中有更好的性能体验，即使单一应用的性能还比较差。

鉴于这些容错成本和其他独立任务模型的优点，我们认为，集群系统设计者应该考虑容错性和弹性因素，即使仅仅是针对短期工作。提供这些特性的系统将会更容易得扩展到大规模查询以及多用户环境。

[20] 除了这些，还有其他的一些好处，尤其是在多用户环境中。如果 map 任务没有快速产生数据，它会避免一些机器执行 reduce 任务。 [116].

5.4 限制与扩展

虽然先前的章节讲述了 RDD 能够有效模拟分布式系统，但它也有无法做到的情况。现在，我们研究一些关键限制，并讨论几种可能绕过它们的模型扩展。

5.4.1 延迟

正如前面几章说明的那样，基于 RDD 分布式系统的仿真与实际系统之间差距的主要系能指标就是延迟。因为 RDD 操作在整个集群中是确定且同步的，又由于启动每个“时步(timestep)”计算存在固有的延迟，所以大量的时步计算导致系统更慢了。

这类应用有两种设计方式，低延迟流式系统（*如*毫秒级请求）和细粒度时间片模拟（*如*，科学模型 ）。而我们发现，在实践中 RDD 操作可以低至 100ms 的延迟，这对一些应用程序还不够。从“人”的时间尺度来看，延迟已经足够低来跟踪事件(*如* Web 点击率和社交媒体的趋势)，并且符合互联网大范围的延迟。在仿真应用中，对抖动容忍的最新研究工作非常适用于 RDD [121]。这项研究工作在每台机器的本地网络区域上模拟多个时步。在大多数模拟中，信息需要花费几个时间片在整个网络中转移，并需要更多的时间来与其他节点进行同步。它被专门用来处理慢任务倾向（straggler-prone）的云环境。

5.4.2 通信模式

5.2.1 章节表明 MapReduce 和 RDD 可以仿真一个分布式系统。当使用 reduce 操作时，系统的节点之间通过点对点通信，虽然这是一个常见的场景，但实际网络也有其他有效的原语，如广播或者网络内聚合。一些数据密集型应用能够从这些中显著受益（如：在机器学习算法将当前模型广播出去，或者收集反馈结果)这些原语仅仅通过点对点的消息传输是非常难以效仿的，因此在 RDDs 之上直接支持这些原语是有帮助的。比如，spark 已经包含了一个有效的操作，“广播”，它通过 BitTorrent 来实现[30]

5.4.3 异步

RDD 操作，例如多对多的 *reduce* 操作，都是通过同步来提供确定性。当节点间的工作不均

衡或者某些节点是慢节点时，这可能会减缓计算速度。最近，一些集群计算系统提出了让节点 *异步*发送消息，这使得即使存在慢节点，计算块也可以继续[71, 48, 32]。尽管仍然保留故障恢复，一个类似 RDD 的模型是否能够支持这样的操作，这个话题值得探讨。例如，节点可能可靠地记录每次迭代计算的信息 ID，这可能仍然比记录信息更节省。请注意，对一般的计算，从丢失故障中重建不同的状态通常没有用，因为在丢失节点上后续计算可能已经使用了丢失状态。

一些算法，例如统计优化，能够从没有完全丢失所有过程的损坏状态[71]继续执行。这种情况下，一个基于 RDD 的系统可以在一个特殊模式下运行这些算法，它不执行恢复，但对结果执行检查点，允许在其上进行后续计算 *(例如,* 交互查询) 以得到一个一致的结果。

5.4.4 细粒度更新

由于记录每个操作的lineage代价很高，用户对RDDs进行许多细粒度更新时是不高效的。例如，Spark并不适合实现一个分布式的key-value存储系统。在某些情况下，我们或许可以将多个*操作*聚合起来一次处理以及将多个更新操作合为一个粗粒度操作。就如同在D-streams中的“离散化”流处理。[21]当然在目前的Spark系统中，对于一个key-value存储系统来说还算不上低延迟，如果能在一个低延迟的系统中实现这个模型还是很有趣的。这个方法类似于Calvin分布式数据库[103]，通过批处理事务以及确定性执行来获得更好的扩展性和可靠性。

5.4.5 不变性和版本追踪

正如本章之前讨论的，不变性可能会增加开销，因为更多的数据需要在基于 RDD 的系统中进行复制。RDDS 被设计为不可变的主要原因是为了跟踪不同版本的数据集的依赖关系，并恢复依赖于旧版本数据集的状态。但是，在任务执行时仍可通过别的途径来跟踪这些依赖关系，以及使用基于 RDD 抽象的可变状态。当处理流数据或者细粒度更新时，这将会是个很有趣的补充。正如 3.2.3 节所讨论的，用户可以使用其他的方法来手动把状态分割为多个 RDD，或是借助存储的(持久化)数据来对旧数据进行重用。

[21]需要注意的是，在这种情况下更新序列仍然需要副本来保证可靠性，不过这种情况在所有可靠的 key-value 存储中的写操作都需要副本来保证。

5.5 相关工作

有一些其它的项目尝试让 MapReduce 模型通用，以高效地支持更多的计算。例如，Dryad [61] 扩展了该模型，实现了对任意 DAG 任务的支持；CIEL [77] 允许任务基于它读取的数据在运行时修改 DAG *(比如.,* 创建其它任务)。然而，据我们所知，这些研究工作都没有试图正式地说明其模型具有某种能力 (*比如.,* 通过模拟一般的并行机器)，只是进一步说明了某些应用能运行的更快。

本章将重点关注并行机器模型方面的研究工作，比如 PRAM[42]、BSP[108]、QSM[46]以及 LogP[35]。这些工作大部分是集中在定义更类似于真实机器的新模型，以及确定哪个模型可以模拟另外的模型。比如，BSP 已被证明在有足够的并行松弛时，它可在一个常量因子(复杂度)下实现对 PRAM 的模拟[108]。另外，也有研究证明，QSM、BSP 和 LogP 可在多项式因子(复杂度)内实现相互之间的模拟。[88]在下文中，我们将证明 RDD 能高效表达 BSP 模型，而且表明大多数情况下它能增加应用的网络延时(*比如*对基于 LogP 的模型的应用)通俗来讲，这将意味着基于这些模型所设计的算法都很可能可以用 RDD 来实现。

使用 RDD 来实现这些应用是有趣的。其中一个原因就是 RDD 具有容错特性。另外，在存在并行松弛(*即*多个任务运行在同一个物理处理器上)时它可以在多个节点上进行并行恢复。在更为传统的各类计算模型中，我们所知道的唯一实现了这点的是 Savva 和 Nanya[91]。该模型基于 BSP 实现了可容错的共享内存模型，但这种实现是通过复制内存数据而实现。

最近，一些理论文章在讨论 MapReduce 的复杂度度量方法和表达能力 [40, 65, 49, 90, 47]. 这些工作主要集中在研究仅使用少量的 MapReduce 步骤能完成哪些计算，毕竟加入新步骤的代价不低。

其中的某些工作表明了 MapReduce 可以表达 BSP 或 PRAM 模型[65, 49]。RDD 是一个实用的计算抽象，它能减少多轮计算所引入的代价。那么，使用 RDD 时，这些分析的情况是否会有所不同是个很有趣的问题。

最后，也存在一些计算系统能实现 BSP 模型，包括 Pregel [72], 迭代式 MapReduce 模型 [37, 22], 以及 GraphLab [71].然而，就我们所知，这些系统没有被用于直接模拟一般的计算。

5.6 小结

本节从两个角度探讨了为什么 RDD 能够成为一个通用的模型：一是从对其他分布式系统的

表达能力方面，二是从用户对集群性能的关键因素是否可控这一实用角度。在第一方面的阐述中，我们表明了 RDD 能模拟*所有的*分布式系统，其代价主要只是网络延迟成本的增加--而这种代价在许多应用中看来是可以接受的。从第二个观点来看，RDD 是成功的，因为它们让用户来控制网络和数据配置这些与以前特定系统所有的优化最为相同的方面。借助 RDD 的性质，我们还能获得了对模型进行进一步扩展的可能。

第六章 总结

我们应该如何为大规模并行集群的新时代设计计算框架？该篇论文证明了在很多情况下答案是可以是相当简单的：一种对计算的单一抽象，基于粗粒度操作的高效数据共享，就能够在一系列普遍的作业模式中取得最先进的性能，同时也会提供一些先前系统所缺少的特性。

无疑集群计算系统将会继续演变，我们希望在这里提出的 RDD 架构，最起码是一个有用的参考。目前，Spark 引擎仅有 34000 行代码（首次发布的版本是 14000 行），并且建立在它之上的其他模型的代码量也要比一个独立的系统小上一个数量级。在应用需求快速发展的领域，我们相信在集群计算最困难（容错，调度，多租户）部分上的小而通用的抽象必然可以实现快速创新。

即使不是为了性能而采用 RDD 模型，我们相信其主要贡献是使以前完全不同的集群作业模式 可以*组合*起来。随着集群应用复杂性的增加，往往需要结合不同类型的计算和处理模型*(例如,* 机器学习和 SQL)和 *(例如,* 交互式查询和流操作)。由于不同系统之间共享数据的限制，这些计算与特定模型的结合必然会付出高昂的成本。而通过基于 lineage 的故障恢复 ，RDDs 避免了数据共享的开销，实现了高速的数据分享。并且在集群级别上，细粒度的执行使基于 RDD 的应用能够有效共存，为所有的用户提高了生产力。在实际部署中，这种多租户模式所带来的效率提升远远超过任何的单点计算。

在本章的其余部分，我们总结了一些影响这项工作的经验。然后讨论了它对于工业界的冲击。最后，我们将试图勾画出未来的工作领域。

6.1 经验总结

数据共享的重要性。我们工作的基本主线是数据共享对于性能的重要性，数据共享无论是对单一模式的计算(*例如*，迭代算法或数据流作业）应用，还是多种计算模式交错的应用都非常重要。特别是对于“大数据”的应用程序，数据集迁移代价是非常高的，所以对应用开发者来说，有效共享是很关键的。然而，以前的系统大多集中在实现特定的数据流模式，而 RDDs 使数据集成为一等原语，为用户提供了足够的机制来控制其属性(*例如*,分区和持久性)，同时其足够抽象的接口能够自动提供容错功能。

由于每台机器的网络带宽，存储带宽和计算能力之间的差异，我们认为数据共享在大多数分布式应用中，仍备受关注，并行处理平台仍将需要解决这一问题。

在共享环境中衡量性能，而不是基于单一应用。虽然针对特定应用的进行执行引擎优化是有益的，但我们所得到的另一个总结是，现实中的部署往往是比较复杂的，而在这些复杂的设置中衡量性能则是最重要的。特别是：

- 大多数工作流程会*结合*不同形式的处理，*例如*，使用 MapReduce 解析一个日志文件，然后在其上运行一个机器学习算法。
- 大多数部署会在多个应用之间*共享* ，需要执行引擎能够动态资源共享、撤销和重执行。

例如，假设一个机器学习算法的专门实现，使用一个像MPI的这样的执行模型（在整个应用运行过程中资源是静态分配的)，比Spark 执行快上 5 倍。然而在一个*端到端*的工作流程中这样的专有系统仍然会比较慢，这个流程包括使用MapReduce脚本的解析数据文件，然后运行学习算法。为了衔接这两个过程，将会需要把解析所得的数据集额外输出到一个可靠的存储系统中，从而来实现系统之间的共享。并且在一个多用户集群中，专有系统需要预先为应用选择一个固定的分配，这或将导致应用出现排队状况，又或是没有充分利用资源，并且与RDDs这样的细粒度执行模式相比，降低了集群中的所有用户的响应能力。[22]

我们认为，由于观点和上面所说的第一个经验相同（数据迁移比较昂贵）集群将会被动态地分享,这需要应用横向或是纵向积极地扩展以及轮流访问每个节点上的数据。在这些环境中，我们认为计算机系统将不得不为了这样的共享应用而进行 *优化*,从而在大多数部署中获得一定的性能优势。

[22] 虽然没有包括在本文中，但作者还是参与了在细粒度任务模型中的一些调度算法，以证明资源之间高效和动态地共享其实是可以实现的[117, 45, 56]。

瓶颈优化相当重要。一个有趣的经验是，如何设计通用处理引擎还要看瓶颈在哪里。在很多情况下，一些资源最终限制了整个应用的性能，所以给用户优化这些资源的控制力能够得到良好的性能。例如，当 Cloudera 发布 Impala SQL 引擎时，伯克利 AMPLab 发现，与 Shark 相比，在许多查询中，性能几乎相同 [111]。这是为什么呢？这些查询要么是 I/O，要么是网络瓶颈，这两个系统都使可用带宽达到了饱和。

这是一个有趣的方法来处理通用性问题，因为这意味着一般不需要低级抽象。例如，RDDs 通过控制分区给用户优化网络使用（最常见的瓶颈）的能力。但是，他们是使用通用的模式来做到这一点的(*例如,* 分区)，而不需要用户手动选择哪台机器上的每块数据，因此可以自动处理再平衡和容错能力。

简单的设计之间复合。最后，使 Spark 和 RDDs 两者如此通用的原因是，它们是建立在一组小的核心接口上的。在接口级别上，RDDs 只有两种类型间的转换（窄和宽）可区分。在执行过程中，它们通过一个单一标准接口（一个记录迭代器）来遍历数据，允许不同数据格式和处理功能的高效融合。关于任务调度的目的，它们可以细分到一个非常简单的模型（细粒度的、无状态的任务），基于此已经开发出的一套广泛的算法提供公平的资源共享 [45],数据局部性 [117], 慢节点缓解 [120, 8],以及可扩展的执行的算法 [83]。这些调度算法*自身之间相互复合*，在一定程度上归因于细粒度任务模型的简单性。其结果是整个系统可以同时从最新的并行处理算法和最强大的运行时算法的执行中获益。

6.2 更深远的影响

自从 2010 年 Spark 发布以来，spark 开源社区得到了快速的发展，已有来自 25 个公司超过 100 个开发人员参与了该项目。同时 spark 软件栈中的其他项目也吸引了非常多的爱好者以及贡献者（比如自 2011 年起，Shark 项目已有 29 个开发人员参与其中）。

这些外部的开发人员为核心模块贡献了非常多的重要特性、思路以及测试用例，这也助推了 spark 的设计。对于 Spark 在行业中的的应用案例的介绍，读者可以查看 2013 年 Spark 峰会中的相应的介绍[97]。

6.3 未来的工作

如 5.4,节所述，我们对 RDDs 的分析也表现出了模型的局限性，这是我们未来研究以进一步泛化该模型的兴趣点。这里重述下这部分内容，未来扩展的主要领域包括：

- 通信延迟：基于 RDD 模拟任意分布式系统的一个主要缺点是，它需要额外的延迟去同步每一个步骤以使得计算是确定性的。未来在这方面有两个有趣的方向。第一个是系统方面的挑战，我们要研究一个基于内存集群计算系统其延迟时间能够达到的水平 - 新的数据中心网络可能达到微秒级的延迟，另外，对于一个优化好的代码库，每一步的延迟可能只需要几毫秒。(在当前的基于 JVM 的 Spark 系统中，java 程序运行时间会使得延迟更高）第二个工作是在 RDDs 中使用延迟隐藏技术[121]用以执行需要紧密同步的应用，它是通过将工作划分为不同的区块或是推测其它机器的响应时间来实现的，
- 新的通信模式：RDDs 目前只在通信节点之间提供了点对点的 shuffle 模式，但也有些并行计算采用其他的通信模式会带来更好的结果，比如广播或者多对一的聚和。研究发掘这些模式也许能提高应用程序的性能，以及创造新的运行时优化和故障恢复方法的机会。
- 异步：虽然基于 RDD 的计算是同步和确定性的，但它可能也适用于在模型内执行异步计算步骤，同时在这些步骤之间提供故障恢复保障。
- 细粒度的更新：RDDs 擅长于粗粒度和数据并行的操作，但是，如第 5.4.4, 所述，使用细粒度的操作来模拟这样一个系统也是可行的，如读写一个键值对数据时，通过将这些操作分组来批量执行。特别是在更低的延迟运行时间下，可能非常有趣的是这种方法对比传统数据库设计究竟能有多快，以及通过运行事务和分析工作共存的情况下能带来什么好处。
- 版本跟踪：RDDs 定义为不可变的数据集,以允许依赖关系执行具体的版本，但是在这个抽象框架中通过使用可变的存储和更高效的版本追踪方法还有很大的提升空间。

除了以上这些方面来扩展 RDD 的编程模型外,我们在 Spark 方面的经验还指出了用户面临的实际系统相关的问题，这些可能也是未来研究工作的兴趣点。一些重要的领域有：

- 正确性调试：分布式应用的调试和正确性测试是复杂的，尤其是操作大量的，没有对比的数据集。在 Spark 中我们已经探索过的一个想法是使用 RDDs 的依赖关系信息高效地*重现调试*应用程序的一部分*(例如*，异常引起任务的崩溃，或产生某个特定输出的执行图的

一部分)。该工具还可以在第二次运行时修改 lineage 图，*比如*，在用户的函数中添加日志记录或增加错误跟踪记录。

- 性能调试：在 Spark 的邮件列表中，最常咨询的调试问题是关于性能的而不是程序正确性的。分布式应用程序的调优是非常困难的，一部分原因是用户对于什么是好的性能缺少直觉。如果一个 PageRank 的实现在有 5 个节点的集群上，处理 100 GB 的数据需要 30 分钟，这个性能好吗？这个应用能够只使用 2 分钟，或 10 秒吗？诸多因素如通信成本，各种数据表示的存储开销和数据倾斜等，都可能明显的影响并行应用程序的性能。开发一些可以自动检测这些低效率因素的工具，或者甚至是能够给用户提供足够的关于应用程序性能信息以辨别问题的监控工具，都是有趣且具有挑战性的工作。

- 调度策略：虽然 RDD 模型非常灵活的支持运行时细粒度任务的调度，并且 RDD 已被用于实现了一些调度机制，如公平共享调度，但是在用这种模型编写的应用中找到一个正确的调度策略仍然是一个有挑战的问题。例如，在 Spark 的流式应用中，在有多个数据流的情况下，我们该如何调度计算以满足任务的执行期限或优先级？如果同样的应用同时在数据流上执行交互式查询呢？同样的，给定一个 RDDs 或 D-Streams 的图结构，我们能够自动确定检查点以减少预期的执行时间吗？随着应用变的越来越复杂以及用户要求更多的响应接口，这些策略对于维持良好的性能是很重要的。

- 内存管理：在大多数集群中分配有限的内存是一个有趣的挑战，同时也取决于应用程序定义的优先级和使用模式。问题之所以特别有趣，是因为有不同的“层次”的存储，需要权衡内存大小与访问速度。例如，在内存中的数据能够被压缩，这可能使它需要更大的计算开销，但所需的内存更少；或者数据也可以被换出到
  SSD 中，或者是磁盘上。有一些 RDDs 可能没有任何持久化会更高效，它是通过在运行过程中对前一个 RDD 运行 map 函数来重新计算（对于大多数用户而言计算可能足够快了，这对于节省空间是值得的)。一些 RDDs 的计算可能要很大开销，因此 RDDs 一直需要被复制。特别的，因为 RDDs 总是能够可以从头开始计算丢失的数据，所以为存储管理策略留下了很大的优化空间。

我们希望在开源系统 Spark 上的持续经验将有助于我们应对这些挑战，并设计出适用于 Spark 和其他集群计算系统的解决方案。

参考文献

[1] Azza Abouzeid et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *VLDB*, 2009.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB,* 2013.

[3] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP server and online behavioral targeting. *Proc. VLDB Endow.*, 2(2):1558, August 2009.

[4] Amazon EC2. http://aws.amazon.com/ec2.

[5] Amazon Redshift. http://aws.amazon.com/redshift/.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS '11,* 2011.

[7] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *NSDI'12,* 2012.

[8] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI'10,* 2010.

[9] Apache Flume. http://incubator.apache.org/flume/.

[10] Apache Giraph. http://giraph.apache.org.

[11] Apache Hadoop. http://hadoop.apache.org.

[12]Apache Mahout. http://mahout.apache.org.

[13]Apache Spark. http://spark.incubator.apache.org.

[14] Apache Storm. http://storm-project.net.

[15] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford stream data management system. In *SIGMOD,* 2003.

[16] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[17] Shivnath Babu. Towards automatic optimization of MapReduce programs. In *SoCC'10,* 2010.

[18] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. Database Syst.,* 2008.

[19] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: MapReduce for incremental computations. In *SOCC'*11,2011.

[20] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys,* 37:1-28, 2005.

[21] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW,* 1998.

[22] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.,* 3:285296, September 2010.

[23] Don Carney, Ugur Cˆetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *VLDB '02,* 2002.

[24] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD,* 2013.

[25] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[26] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vi- jayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR,* 2003.

[27] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. Tenzing a SQL implementation on the MapReduce framework. In *Proceedings of VLDB,* 2011.

[28] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases,* 1(4):379- 474, 2009.

[29] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[30] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM,* 2011.

[31] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS '06,* pages 281-288. MIT Press, 2006.

[32] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *HotOS*, 2013.

[33] J. Cohen, B. Dolan, M. Dunlap, J.M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *VLDB*, 2009.

[34] Tyson Condie, Neil Conway, Peter Alvaro, and Joseph M. Hellerstein. Map- Reduce online. *NSDI*, 2010.

[35] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a

realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1-12, July 1993.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *oSdI,* 2004.

[37] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *HPDC '10,* 2010.

[38] EsperTech. Performance-related information. http://esper.codehaus.org/esper/performance/performance.html, Retrieved March 2013.

[39] EsperTech. Tutorial. http://esper.codehaus.org/tutorials/tutorial/tutorial.html, Retrieved March 2013.

[40] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, and Cliff Stein. On distributing symmetric streaming computations. In *SODA*, 2009.

[41] Xixuan Feng et al. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, 2012.

[42] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC,* 1978.

[43] M.J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network- effect world. *CIDR*, 2009.

[44] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings ofSOSP '03,* 2003.

[45] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andrew Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI,* 2011.

[46] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Can shared- memory model serve as a bridging model for parallel computation? In *SPAA,* pages 72-83, 1997.

[47] Ashish Goel and Kamesh Munagala. Complexity measures for map-reduce, and

comparison to parallel computing. *CoRR,* abs/1211.6526, 2012.

[48] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI'12,* 2012.

[49] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the MapReduce framework. In Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation,* volume 7074 of *Lecture Notes in Computer Science,* pages 374-383. Springer Berlin Heidelberg, 2011.

[50] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI '10,* 2010.

[51] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: an application-level kernel for record and replay. In *OSDI,* 2008.

[52] Jeff Hammerbacher. Who is using flume in production? http://www.quora.com/Flume/Who-is-using-Flume-in-production/answer/Jeff-Hammerbacher.

[53] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer Publishing Company, New York, NY, 2009.

[54] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: batched stream processing for data intensive distributed computing. In SoCC, 2010.

[55] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN Notices,* pages 311-320, 2000.

[56] Benjamin Hindman, Andrew Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained

resource sharing in the data center. In *NSDI,* 2011.

[57] Timothy Hunter, Teodor Moldovan, Matei Zaharia, Samy Merzgui, Justin Ma, Michael J. Franklin, Pieter Abbeel, and Alexandre M. Bayen. Scaling the Mobile Millennium system in the cloud. In *SOCC '11,* 2011.

[58] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[59] Jeong hyon Hwang, Ying Xing, and Stan Zdonik. A cooperative, selfconfiguring high-availability solution for stream processing. In *ICDE*, 2007.

[60] Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[61] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys,* 2007.

[62] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, November 2009.

[63] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD,* 1998.

[64] Haim Kaplan. Persistent data structures. In Dinesh Mehta and Sartaj Sanhi, editors, *Handbook on Data Structures and Applications.* CRC Press, 1995.

[65] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *SODA*, 2010.

[66] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On availability of intermediate data in cloud computations. In *HotOS '09,* 2009.

[67] S. Krishnamurthy, M.J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD,* 2010.

[68] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Eric Baldeschwieler, Scott Shenker, and Ion Stoica. Tachyon: Memory throughput I/O for cluster computing frameworks. In *LADIS,* 2013.

[69] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, 2010.

[70] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth Yocum. In-situ MapReduce for log processing. In *USENIXATC,* 2011.

[71] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Ky- rola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud, April 2012.

[72] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD,* 2010.

[73] Nathan Marz. Trident: a high-level abstraction for realtime computation. http://engineering.twitter.com/2S12/S8/trident-high-level-abstraction-for.html.

[74] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR),* 2013.

[75] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330-339, Sept 2010.

[76] Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *SOSP '13,* 2013.

[77] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI,* 2011.

[78] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed

stream computing platform. In *Intl. Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud)*, 2010.

[79] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52 -60, Aug 1991.

[80] Kevin O'Dell. How-to: Select the right hardware for your new hadoop cluster. Cloudera blog, http://blog.cloudera.com/blog/2S13/S8/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/.

[81] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[82] Oracle. Oracle complex event processing performance. http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/cepperformancewhitepaper-128S6S.pdf, 2008.

[83] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP '13*, 2013.

[84] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09*, 2009.

[85] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[86] Russel Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proc. OSDI 2010*, 2010.

[87] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *EuroSys '13*, 2013.

[88] Vijaya Ramachandran, Brian Grayson, and Michael Dahlin. Emulations between qsm, bsp and logp: A framework for general-purpose parallel algorithm design. *J. Parallel Distrib. Comput.*, 63(12):1175-1192, December 2003.

[89] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill, Inc., 3 edition, 2003.

[90] Anish Das Sarma, Foto N. Afrati, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *VLDB'13,* 2013.

[91] A. Savva and T. Nanya. A gracefully degrading massively parallel system using the bsp model, and its evaluation. *Computers, IEEE Transactions on*, 48(1):38-52, 1999.

[92] Scala Programming Language. http://www.scala-lang.org.

[93] Mehul Shah, Joseph Hellerstein, and Eric Brewer. Highly available, fault- tolerant, parallel dataflows. *SIGMOD,* 2004.

[94] Zheng Shao. Real-time analytics at Facebook. XLDB 2011, http://www-conf.slac.stanford.edu/xldb2S11/talks/xldb2S11_tue_0940_facebookrealtimeanalytics.pdf.

[95] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.,* 6(11), August 2013.

[96] Spark machine learning library (MLlib). http://spark.incubator.apache.org/docs/latest/mllib-guide.html.

[97] Spark summit 2013. http://spark-summit.org/summit-2S13/.

[98] Evan Sparks, Ameet Talwalkar, Virginia Smith, Xinghao Pan, Joseph Gonzalez, Tim Kraska, Michael I. Jordan, and Michael J. Franklin. MLI: An API for distributed machine learning. In *ICDM,* 2013.

[99] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS,* 2004.

[100] Mike Stonebraker et al. C-store: a column-oriented dbms. In *VLDB'05,* 2005.

[101] Guaranteed message processing (Storm wiki). https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing.

[102] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy*, 2011.

[103] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD '12*, 2012.

[104] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

[105] Richard Tibbetts. Streambase performance & scalability characterization. http://www.streambase.com/wp-content/uploads/downloads/StreamBase_White_Paper_Performance_and_Scalability_Characterization.pdf, 2009.

[106] Transaction Processing Performance Council. *TPC BENCHMARK* H.

[107] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. In *SIGMOD*, 1998.

[108] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103-111, August 1990.

[109] Vinod K. Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Ma- hadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Sid- dharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[110] Huayong Wang, Li-Shiuan Peh, Emmanouil Koukoumidis, Shao Tao, and Mun Choon Chan. Meteor shower: A reliable stream processing system for commodity data centers. In *IPDPS '12*, 2012.

[111] Patrick Wendell. Comparing large scale query engines. https://amplab.cs.berkeley.edu/2S13/S6/S4/comparing-large-scale-query-engines/.

[112] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx:

A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems,* GRADES '13, 2013.

[113] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data,* SIGMOD'13, 2013.

[114] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM,* 17:530-531, Sept 1974.

[115] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general- purpose distributed data-parallel computing using a high-level language. In *OSDI '08,* 2008.

[116] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, University of California, Berkeley, Apr 2009.

[117] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 10,* 2010.

[118] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, University of California, Berkeley, 2011.

[119] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP,* 2013.

[120] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI 2008,* December 2008.

[121] Tao Zou, Guozhang Wang, Marcos Vaz Salles, David Bindel, Alan Demers, Johannes Gehrke, and Walker White. Making time-stepped applications tick in the cloud. In *SOCC '11,* 2011