数据库系统读本

第五版

编辑
彼得·贝利斯
约瑟夫·M·赫勒斯坦
迈克尔·斯通布雷克

数据库系统读本

第五版（2015年）

由彼得・贝利斯、约瑟夫・M・赫勒斯坦和迈克尔・斯通布雷克编辑

目录

前言

自从上一版的数据库系统读本出版以来的十年里，数据管理领域发生了爆炸性的变化。 今天的数据库和数据密集型系统在数据量上前所未有地运行，这在很大程度上得益于“大数据”的崛起以及存储和计算成本的大幅降低。 云计算和微架构趋势使分布和并行几乎成为无处不在的关注点。

数据以越来越多样化和异构的格式和来源以及越来越大的规模进行收集，并且用于越来越广泛的任务。 因此，商品数据库系统在多个维度上都有了显著的发展，从使用新的存储介质和处理器设计，到查询处理架构、编程接口以及事务处理和分析中新兴的应用需求。 这是一个令人兴奋的时代，市场上变化很大，研究中涌现出许多新的想法。

在这个快速变化的时代，我们对传统的“红皮书”进行了更新，旨在为领域的核心概念提供基础，并对选定的趋势进行评论。 一些新技术与几十年前的前辈非常相似，我们认为让读者熟悉主要来源是有用的。 与此同时，技术趋势要求重新评估数据库系统的几乎所有维度，许多经典设计需要修订。 我们在这个系列中的目标是展示重要的长期经验和基础设计，并突出我们认为最新颖和相关的新思想。

因此，我们选择了一系列经典的、传统的论文，这些论文在早期数据库文献中具有影响力，以及在最近的发展中最具影响力的论文，包括事务处理、查询处理、高级分析、Web数据和语言设计。

每个章节都附有一篇简短的评论，介绍了论文的内容，并解释了我们选择该论文的原因。 每篇评论都由编辑之一撰写，但所有编辑都提供了意见；我们希望评论不乏观点。

在选择读物时，我们寻找符合一系列核心标准的主题和论文。 首先，每个选择都代表了数据管理的主要趋势，这一点可以从研究兴趣和市场需求得到证明。
其次，每个选择都是经典或接近经典的；我们寻找每个主题最具代表性的论文。
第三，每个选择都是一份原始资料。 关于这个集合中许多主题有很好的调查报告，我们在评论中引用了这些报告。 然而，阅读原始资料可以提供历史背景，让读者了解塑造有影响力的解决方案的思考过程，并有助于确保我们的读者在这个领域有扎实的基础。 最后，这个集合代表了我们目前的品味，我们希望读者以批判的眼光看待这个集合。

与《红皮书》以前的版本相比，一个主要的变化是我们对分析和数据集成这两个部分的处理方式。在研究和市场上都清楚，这两个问题是当今数据管理中最大的问题之一。 它们也是研究和实践中发展迅速的主题。 鉴于这种不稳定的状态，我们发现很难就这些主题达成一致的“经典”读物。 在这种情况下，我们决定省略正式的读物，而是提供评论。 这显然导致了对领域内正在发生的事情的高度偏见。 因此，我们不建议将这些部分作为《红皮书》传统上试图提供的“必读”内容。 相反，我们将这些部分视为可选的附录：“对移动目标的偏见观点”。 读者应该对这两个部分持保留态度（甚至比对本书其他部分使用的保留态度更大）

我们以无限制的非商业再分发的许可证，免费提供这本红皮书的第五版。 我们没有获得推荐论文的版权，而是提供了链接到Google学术搜索的方式，以帮助读者找到相关论文。 我们预计这种电子格式将允许更频繁的“书”的版本。我们计划根据需要不断更新这个收藏。

最后一点：这个收藏自1988年以来一直存在，并且我们预计它将有一个长久的未来。 因此，我们在这些灰发编辑中添加了一些“年轻的血液”。 随着时间的推移，这个收藏的编辑可能会进一步发展。

彼得・贝利斯
约瑟夫・M・赫勒斯坦
迈克尔・斯通布雷克

第一章：背景

由迈克尔・斯通布雷克引入

精选读物：

Joseph M. Hellerstein和Michael Stonebraker。历史循环。《数据库系统读物》第四版（2005年）。

Joseph M. Hellerstein, Michael Stonebraker, James Hamilton. 一个数据库系统的架构。数据库基础与趋势, 1 , 2 (2007).

我很惊讶这两篇论文是在仅仅十年前写的！我对解剖学论文的惊讶在于细节在几年后发生了很大变化。 我对数据模型论文的惊讶在于似乎没有人从历史中学到任何东西。

让我们先谈谈数据模型论文。

十年前，XML是所有人关注的焦点。供应商们都希望将XML添加到他们的关系型引擎中。 工业分析师（以及不少研究人员）都在宣扬XML是“下一个大事”。 十年后，它成为了一个小众产品，领域已经发展了。在我看来，（正如论文中预测的）它受到了以下因素的影响：

- 过于复杂（没有人能够理解）
- 关系引擎的复杂扩展，并不表现出良好的性能
- 没有令人信服的广泛应用场景

有点讽刺的是，在论文中预测X将通过成功简化XML而赢得图灵奖。这个预测完全错误！最终关系型数据库胜出，XML失败。

当然，这并没有阻止“新手”重新发明轮子。 现在是JSON，可以从三个方面来看待：

- 通用的分层数据格式。 任何认为这是个好主意的人都应该阅读数据模型论文中关于IMS的部分。
- 稀疏数据的表示。 考虑员工的属性，并假设我们希望记录兴趣爱好数据。对于每个爱好，记录的数据都是不同的，而且兴趣爱好基本上是稀疏的。在关系型数据库管理系统中对此进行建模很简单，但会导致非常宽、非常稀疏的表。 这对于基于磁盘的行存储来说是灾难性的，但在列存储中运行良好。 这对于基于磁盘的行存储来说是灾难性的，但在列存储中运行良好在前一种情况下，JSON是“爱好”列的合理编码格式，最近一些关系数据库管理系统已经添加了对JSON数据类型的支持。
- 作为“读取时模式”的机制。 实际上，模式非常宽且非常稀疏，几乎所有用户都希望对此模式进行某种投影。 当从宽且稀疏的模式中读取时，用户可以在运行时指定他想要看到的内容。 从概念上讲，这只是一个投影操作。 因此，“读取时模式”只是对JSON编码数据的一种关系操作。

总之，JSON是稀疏数据的合理选择。 在这种情况下，我预计它会有相当多的“生命力”。 另一方面，作为一种通用的分层数据格式，它是一场灾难的酝酿。 我完全预料到关系数据库管理系统将JSON仅仅作为其系统中的一种数据类型（众多数据类型之一）来吸收。 换句话说，这是一种合理的编码稀疏关系数据的方式。

毫无疑问，红皮书的下一个版本将会摧毁一些由那些站在前人肩膀上的人发明的新的分层格式。

在过去的十年中，另一个引起很大关注的数据模型是Map-Reduce，它是由谷歌专门为支持他们的网络爬虫数据库而构建的。 几年后，谷歌停止使用Map-Reduce来处理该应用程序，转而使用Big Table。现在，世界其他地方也开始意识到谷歌早就发现的事实：Map-Reduce并不适用于广泛的架构。 相反，Map-Reduce市场已经变得更加-

进入了HDFS市场，并且似乎有望成为关系型SQL市场。 例如，Cloudera最近推出了Impala，这是一个SQL引擎，构建在HDFS之上，而不是使用Map-Reduce。最近，在HDFS领域还有另一个推动力

值得讨论，即“数据湖”。 一个合理的使用HDFS集群的方式（现在大多数企业都已经投资并希望找到对它们有用的东西）是作为已经摄入的数据文件的队列。 随着时间的推移，企业将会找出哪些值得花费精力进行清理（数据整理；本书第12章介绍）。 因此，数据湖只是一个暂时的“杂物抽屉”。 此外，在第5章中我们还将对HDFS、Spark和Hadoop进行更多的讨论。

总结一下，在过去的十年里，似乎没有人注意到“因果循环”的教训。 新的数据模型已经被发明出来，只是变成了基于表的SQL。
分层结构被重新发明，结果却是失败。 我不会对下一个十年出现更多相同的情况感到惊讶。 人们似乎注定要重新发明轮子！

关于解剖学论文，仅仅十年后，我们可以看到DBMS构建方式发生了重大变化。 因此，细节发生了很大变化，但是论文中描述的整体架构仍然基本正确。 该论文描述了大多数传统的DBMS（例如Oracle，DB2）的工作方式，十年前，这是主流实现方式。 现在，这些系统已经成为历史文物；在任何方面都不太好。 例如，在数据仓库市场上，列存储已经取代了本论文中描述的行存储，因为它们的速度快了1-2个数量级。

在OLTP世界中，具有非常轻量级事务管理的主内存SQL引擎正在迅速成为常态。 这些新发展在本书的第4章中有详细记录。 现在很难找到传统行存储在应用领域中有竞争力的地方。

因此，它们应该被送到“退休软件之家”。 很难想象“一刀切”会再次成为主导架构。

因此，“大象”们面临着一个糟糕的“创新困境”问题。 在克莱顿·克里斯蒂森的经典著作中，他认为传统技术的供应商很难转变到新的结构而不失去他们的客户基础。 然而，大象们将如何尝试已经显而易见。例如，SQLServer 14至少是两个引擎（Hekaton主内存OLTP系统和传统的SQLServer - 传统行存储）在一个共同的解析器下统一。 因此，微软的策略显然是在他们的传统解析器下添加新的引擎，然后支持将数据从一个老旧引擎移动到更现代化的引擎，而不会干扰应用程序。 还有待观察这将会有多成功。 这将取决于微软的执行能力。

然而，这些新系统的基本架构仍然遵循了论文中描述的解析/优化器/执行器的结构。 此外，线程模型和进程结构至今仍然与十年前一样相关。因此，读者应该注意并发控制、崩溃恢复、优化、数据结构和索引的细节正在快速变化，但DBMS的基本架构仍然保持不变。

此外，这些传统系统要消失需要很长时间。事实上，目前仍然有大量的IMS数据在生产中使用。 因此，任何学习这个领域的学生都应该了解（一段时间内）这些系统的架构。

此外，随着计算架构的发展，本论文的某些方面可能在未来变得更加相关。 例如，即将到来的NVRAM可能为新的架构概念提供机会，或者重新出现旧的概念。

第二章：传统关系型数据库管理系统

由迈克尔・斯通布雷克引入

精选读物：

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, Vera Watson. System R: 关系型数据库管理的方法.*ACM Transactions on Database Systems*, 1(2), 1976, 97-137.

Michael Stonebraker 和 Lawrence A. Rowe. POSTGRES 的设计. *SIGMOD*, 1986.

David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990, 44-62.

在这个部分中，有关于（可以说是）最重要的三个实际数据库管理系统的论文。 我们将在本介绍中按时间顺序讨论它们。

System R项目始于1972年左右，由IBM研究部门的Frank King领导。
那时，Ted Codd的开创性论文已经发表了18个月，很多人都认识到应该建立一个原型来测试他的想法。 不幸的是，Ted没有被允许领导这个项目，他离开去考虑自然语言接口与数据库管理系统。System R很快决定实现SQL，该语言从1972年的一个干净的块结构语言[34]演变成了一种更复杂的结构，本文中有详细描述[33]。 关于SQL语言设计的评论，请参见[46]，该评论是十年后撰写的。

System R被分为两个组，即“下半部分”和“上半部分”。 它们并不完全同步，因为下半部分实现了链接，而上半部分不支持链接。 为了支持下半部分团队的决策，很明显他们在与IMS竞争，而IMS具有这种结构，所以自然而然地包含了它。上半部分只是没有让优化器为这种结构工作。

事务管理器可能是该项目中最大的遗产，显然是吉姆・格雷的作品。他的设计在商业系统中仍然存在很多。 第二名是Sys-tem R优化器。 基于动态规划的成本优化方法仍然是优化器技术的黄金标准。

我对System R最大的抱怨是团队从未停下来整理SQL。因此，当“上半部分”简单地粘贴到VSAM上形成DB2时，语言水平保持不变。 所有令人讨厌的语言特性至今仍然存在。SQL将成为2020年的COBOL，一种我们被困在其中的语言，每个人都会抱怨。

我第二大的抱怨是System R使用了一个子程序调用接口（现在是ODBC）将客户端应用程序与DBMS耦合在一起。我认为ODBC是地球上最糟糕的接口之一。 要发出一个查询，必须打开一个数据库，打开一个游标，将其绑定到一个查询，然后为数据记录发出单独的获取操作。 运行一个查询需要一页相当晦涩的代码。 Ingres [150]和Chris Date [45]都有更清晰的语言嵌入方式。 此外，Pascal-R [140]和Rigel [135]也是将DBMS功能包含在编程语言中的优雅方式。 直到最近，随着Linq [115]和Ruby on Rails [80]的出现，我们才看到更清晰的语言特定嵌入方式的复兴。

在System R之后，吉姆・格雷离开去了Tandem工作，致力于Non-stop SQL和Kapali Eswaren进行关系型创业。 团队中的大部分成员留在IBM并继续从事各种其他项目的工作，包括R*。

第二篇论文涉及Postgres。这个项目始于1984年，当时明显继续使用学术Ingres代码库进行原型开发是没有意义的。 Postgres的历史回顾可在[147]中找到，读者可以在那里获得开发过程中的详细回顾。

然而，我认为Postgres的重要遗产是其抽象数据类型（ADT）系统。
使用Postgres模型，用户定义的类型和函数已被添加到大多数主流关系型数据库管理系统中。 因此，这个设计特性一直延续至今。该项目还尝试了时间旅行，但效果不佳。 我认为不覆盖存储将在更快的存储技术改变数据管理经济学的情况下迎来它的时代。

还应该注意到，Postgres的重要性很大程度上应归功于一个强大且高效的开源代码线。 这是一个开源社区模型的最佳实践示例。 一支由志愿者组成的团队在1990年代中期接管了伯克利的代码线，并一直推动其发展至今。

Postgres和4BSD Unix [112] 在使开源代码成为首选的代码开发机制方面起到了重要作用。

Postgres项目在伯克利大学持续进行，直到1992年，商业公司Illustra成立以支持商业代码线。参见[147]。

描述Illustra在市场上经历的起起落落，请参阅[147]。

除了ADT系统和开源分发模型之外，Postgres项目的一个重要遗产是一批经过高度培训的DBMS实施者，他们在构建其他几个商业系统方面发挥了重要作用。

本节中的第三个系统是Gamma，在1984年至1990年间在威斯康星州建立。 在我看来，Gamma推广了共享无分区表的多节点数据管理方法。 尽管Teradata也有相同的并行思想，但是Gamma才是推广这些概念的人。 此外，在Gamma之前，没有人谈论哈希连接，所以应该归功于Gamma（以及Kitsuregawa Masaru）提出了这类算法。

基本上所有的数据仓库系统都采用了Gamma风格的架构。 几乎没有人再考虑使用共享磁盘或共享内存系统。 除非网络延迟和带宽能够与磁盘带宽相媲美，否则我预计当前的共享无分区架构将继续存在。

第三章：每个人都应该了解的技术

由Peter Bailis引入

精选读物：

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price. 在关系数据库管理系统中的访问路径选择。S*I*GM*O*D，1979年。

C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz. ARIES: 一种支持细粒度锁定和部分回滚的事务恢复方法，使用预写式日志记录。ACM 数据库系统事务，17(1)，1992年，94-162。

Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger. 共享数据库中的锁粒度和一致性程度。, IBM, 1975年9月。

Rakesh Agrawal, Michael J. Carey, Miron Livny. 并发控制性能建模：选择和影响。ACM 数据库系统事务，12(4)，1987年，609-654。

C. Mohan, Bruce G. Lindsay, Ron Obermarck. R* 分布式数据库管理系统中的事务管理。*A*CM 数据库系统事务，11(4)，1986年，378-396。

在本章中，我们介绍了数据库系统设计中几个最重要的核心概念的主要和次要来源：查询计划、并发控制、数据库恢复和分布。 本章中的思想对现代数据库系统来说是如此基础，以至于几乎每个成熟的数据库系统实现都包含它们。 本章中的三篇论文是各自主题上的经典参考文献。 此外，与前一章相比，本章侧重于广泛适用的技术和算法，而不是整个系统。

查询优化

在关系数据库架构中，查询优化非常重要，因为它是实现数据独立查询处理的核心。 Selinger等人关于System R的基础论文通过将问题分解为三个不同的子问题：成本估计、定义搜索空间的关系等价性和基于成本的搜索，实现了实际的查询优化。

优化器为查询的每个组件执行的成本提供了估计，以I/O和CPU成本为度量。 为了做到这一点，优化器依赖于关于每个关系内容的预计算统计信息（存储在系统目录中）以及一组用于确定查询输出的基数（大小）的启发式算法（例如，基于估计的谓词选择性）。 作为练习，考虑这些启发式算法在细节：何时它们有意义，以及在什么输入上它们会失败？ 它们如何改进？

使用这些成本估计，优化器使用动态规划算法为查询构建计划。 优化器定义了一组实际操作符，用于实现给定的逻辑操作符（例如，使用完整的“段”扫描还是索引查找元组）。 使用这个集合，优化器迭代地构建一个“左深”操作符树，然后使用成本启发式来最小化运行操作符所需的总估计工作量，考虑到上游消费者所需的“有趣顺序”。 这避免了考虑所有可能的操作符顺序，但在计划大小上仍然是指数级的；正如我们在第7章中讨论的那样，现代查询优化器仍然难以处理大型计划（例如，多路连接）。 此外，虽然Selinger等人的优化器提前执行编译，但其他早期系统（如Ingres [150]）解释了查询计划-实际上，逐个元组地解释。

像几乎所有的查询优化器一样，Selinger等人的优化器实际上并不是“最优”的-没有保证优化器选择的计划是最快或最便宜的。 关系优化器更接近于现代语言编译器中的代码优化例程（即，将进行最佳努力搜索），而不是数学优化例程（即，将找到最佳解决方案）。 然而，今天的许多关系引擎采用了基本的方法论来自

论文，包括使用二进制运算符和成本估计。

并发控制

我们关于事务的第一篇论文，来自Gray等人，介绍了两个经典的思想：多粒度锁定和多个锁定模式。 实际上，这篇论文可以看作是两篇独立的论文。

首先，论文介绍了多粒度锁定的概念。 这里的问题很简单：给定一个具有分层结构的数据库，我们应该如何执行互斥操作？ 我们应该在粗粒度（例如整个数据库）还是细粒度（例如单个记录）上进行锁定，以及如何同时支持对层次结构的不同部分进行并发访问？ 尽管Gray等人的分层布局（包括数据库、区域、文件、索引和记录）与现代数据库系统略有不同，但除了最基本的数据库锁定系统外，几乎所有数据库锁定系统今天都采用了他们的建议。

其次，本文提出了多个孤立程度的概念。正如Gray等人所提醒我们的，并发控制的目标是维护数据的“一致性”，即遵守一些逻辑断言。 经典地，数据库系统使用可串行化事务来强制保持一致性：如果各个事务都将数据库保持在“一致”的状态下，那么可串行化的执行（等效于某些事务的串行执行）将保证所有事务观察到数据库的“一致”状态[57]。

Gray等人的“三级”协议描述了经典的（严格的）“两阶段锁定”（2PL），它保证了可串行化的执行，并且是事务处理中的一个重要概念。

然而，可串行性通常被认为是太昂贵的。 为了提高性能，数据库系统通常使用非可串行化隔离来执行事务。 在本文中，持有锁是昂贵的：在冲突的情况下等待锁需要时间，并且在发生死锁的情况下可能需要永远的时间（或导致中止）。 因此，早在1973年，IMS和System R等数据库系统就开始尝试非可串行化策略。 在基于锁的并发控制系统中，这些策略通过缩短持有锁的时间来实现。 这样可以增加并发性，减少死锁和系统引起的中止，并且在分布式环境中，可能允许更高的操作可用性。

在本文的后半部分，Gray等人提供了这些基于锁的策略行为的基本形式化。 如今，它们非常普遍；正如我们在第6章中讨论的那样，非可串行化隔离是商业和开源关系数据库管理系统的默认设置，而一些关系数据库管理系统根本不提供可串行化隔离。2级通常被称为可重复读隔离，1级被称为读已提交隔离，而0级很少使用[27]。 该论文还讨论了可恢复性的重要概念：在不影响其他事务的情况下，可以中止（或“撤消”）事务的策略。 除了0级事务外，所有事务都满足这个属性。

在Gray等人关于基于锁的可串行化的开创性工作之后，出现了各种替代的并发控制机制。 随着硬件、应用需求和访问模式的变化，并发控制子系统也发生了变化。 然而，并发控制的一个属性几乎可以确定：在并发控制中没有单方面的“最佳”机制。

最优策略取决于工作负载。 为了说明这一点，我们包括了Agrawal、Carey和Livny的一项研究。尽管有些过时，但这篇论文的方法论和广泛结论仍然准确。 这是一个思考周到、不依赖具体实现的性能分析工作的绝佳例子，随着时间的推移，它可以提供宝贵的经验教训。

在方法论上，能够进行所谓的“信封背面”计算是一项宝贵的技能：使用粗略的算术快速估算感兴趣的度量，以便在正确值的数量级内得出答案，可以节省数小时甚至数年的系统实施和性能分析时间。 这是数据库系统中的一项长期而有用的传统，从“五分钟规则”[73]到谷歌的“每个人都应该知道的数字”[48]。 虽然从这些估计中得出的一些教训是暂时的[69, 66]，但通常结论提供了长期的教训。

然而，对于复杂系统（如并发控制）的分析，模拟可以成为估算和全面系统基准测试之间的有价值的中间步骤。 Agrawal研究是这种方法的一个例子：作者使用精心设计的系统和用户模型来模拟锁定、基于重启和乐观并发控制。

评估的几个方面特别有价值。 首先，几乎每个图表都有一个“交叉点”：没有明确的赢家，因为最好的-

性能机制取决于工作负载和系统配置。 相比之下，几乎每个没有交叉点的性能研究都很无聊。 如果一个方案“总是胜出”，研究应该包含分析分析，或者最好是证明为什么这是这样的。 其次，作者考虑了各种系统配置；他们研究并讨论了模型的几乎所有参数。 第三，许多图表呈现非单调性（即，不总是向上和向右移动）；这是垃圾回收和资源限制的产物。 正如作者所示，假设资源无限会导致截然不同的结论。 一个不够谨慎的模型，假设这个假设是隐含的，将会不太有用。

最后，研究的结论是合理的。 基于重启的方法的主要成本是在冲突事件中的“浪费”工作。 当资源充足时，推测是有意义的：浪费的工作成本较低，并且在资源无限的情况下是免费的。

然而，在资源有限的情况下，阻塞策略将消耗更少的资源并提供更好的整体性能。 同样，没有单一最优选择。 然而，论文的结论已经被证明是有远见的：计算资源仍然稀缺，实际上，今天很少有商品系统完全采用基于重启的方法。 然而，随着技术比率 - 磁盘、网络、CPU速度的变化，重新审视这种权衡是有价值的。

数据库恢复

事务处理中的另一个主要问题是保持持久性：事务处理的影响应该在系统故障后仍然存在。 保持持久性的一种几乎无处不在的技术是执行日志记录：在事务执行期间，事务操作被存储在容错介质（例如硬盘或固态硬盘）中的日志中。每个在数据系统中工作的人都应该了解预写式日志记录的工作原理，最好是有一些细节。

实现“无强制、偷窃”基于WAL的恢复管理器的规范算法是IBM的ARIES算法，我们下一篇论文的主题。 （资深数据库研究人员可能会告诉你，类似的想法在Tandem和Oracle等地方同时发明。）在ARIES中，数据库不需要在提交时将脏页写入磁盘（“无强制”），并且数据库可以在任何时候将脏页刷新到磁盘（“偷窃”）[78]；这些策略可以实现高性能。

几乎所有商业RDBMS都采用了这些策略，但同时也增加了数据库的复杂性。 ARIES中的基本思想是通过三个阶段进行崩溃恢复。 首先，ARIES通过向前回放日志来执行分析阶段，以确定崩溃时正在进行的事务。 其次，ARIES通过再次回放日志并执行崩溃时正在进行的任何事务的效果来执行重做阶段。 第三，ARIES通过向后播放日志并撤消未提交事务的效果来执行撤消阶段。 因此，ARIES的关键思想是“重复历史”来执行恢复；实际上，撤消阶段可以执行与正常操作中中止事务相同的逻辑。

ARIES应该是一篇相当简单的论文，但它可能是这个系列中最复杂的论文。 在研究生数据库课程中，这篇论文是一个必经之路。
然而，这些内容是基础的，所以理解它们很重要。幸运的是，Ramakrishnan和Gehrke的本科教材[127]以及Michael Franklin的一篇调研论文[61]都提供了较为温和的解释。
我们在这里包含的完整ARIES论文在其对替代设计决策的缺点的离题讨论中变得复杂。在第一遍阅读时，我们鼓励读者忽略这些内容，只关注ARIES方法。

替代方案的缺点很重要，但应该在更仔细的第二或第三次阅读时再来讨论。 除了组织结构外，ARIES协议的讨论还涉及管理内部状态的讨论，如索引（即嵌套顶级操作和逻辑撤销日志 - 后者也用于像Escrow事务[124]这样的奇特方案）以及在恢复过程中最小化停机时间的技术。 在实践中，恢复时间尽可能短是很重要的，但这很难实现。

分布

本章的最终论文涉及分布环境中的事务执行。这个主题在今天尤为重要，因为越来越多的数据库是分布式的 - 要么是复制的，数据在不同的服务器上有多个副本，要么是分区的，数据项存储在不同的服务器上（或者两者都有）。 尽管分布提供了容量、持久性和可用性方面的好处，但也引入了一系列新的问题。

服务器可能会故障，网络连接可能不可靠。
在没有故障的情况下，网络通信可能

是昂贵的。

我们专注于分布式事务处理中的核心技术之一：原子提交(AC)。 非常不正式地说，给定在多个服务器上执行的事务(无论是多个副本、多个分区还是两者兼有)，AC确保事务在所有服务器上要么提交要么中止。 实现AC的经典算法可以追溯到20世纪70年代中期，称为两阶段提交(2PC；不要与2PL混淆！)[67, 100]。 除了提供2PC和提交协议与WAL之间相互作用的良好概述外，本文还包含了两个改进AC性能的变体。 假设中止变体允许进程避免将中止决策强制写入磁盘或确认中止，从而减少磁盘利用率和网络流量。 假设提交优化类似，当更多事务提交时，优化空间和网络流量。 请注意2PC协议、本地存储和本地事务管理器之间的相互作用的复杂性；细节很重要，正确实现这些协议可能具有挑战性。

故障的可能性大大复杂化了AC（以及分布式计算中的大多数问题）。 例如，在2PC中，如果协调者和参与者在所有参与者发送了他们的投票之后但在协调者收到失败的参与者的消息之前都失败了，会发生什么？ 剩下的参与者将不知道是提交还是中止事务：失败的参与者是投票YES还是投票NO？ 参与者无法安全地继续。 实际上，任何一个在不可靠网络上运行的AC实现都可能阻塞或无法取得进展[28]。 结合可串行化并发控制机制，阻塞的AC意味着吞吐量可能会停滞。 因此，一组相关的AC算法在放松的情况下研究了AC。

关于网络（例如，通过假设同步网络）[145]和服务器可用信息（例如，使用“故障检测器”来确定节点故障）[76]的假设。最后，许多读者可能熟悉一致性问题或听说过一致性实现，例如Paxos算法

。 在一致性中，可以选择任何提案，只要所有进程最终都会同意它。（相比之下，在AC中，任何个体参与者都可以投票否决，之后所有参与者必须中止。）这使得一致性问题比AC问题“更容易”[75]，但是，与AC一样，一致性的任何实现在某些情况下也可能阻塞[60]。 在现代分布式数据库中，一致性通常被用作复制的基础，以确保副本按照相同的顺序应用更新，这是状态机复制的一个实例（参见Schneider的教程[141]）。 AC通常用于执行跨多个分区的事务。 Lamport的Paxos [99]是最早的一致性实现之一（也是最著名的之一，部分原因是其复杂的演示）。 然而，Viewstamped Replication [102]、Raft [125]、ZAB [92]和Multi-Paxos [35]算法在实践中可能更有帮助。

这是因为这些算法实现了分布式日志抽象（而不是像原始的Paxos论文中的'共识对象'）。

不幸的是，数据库和分布式计算社区在某种程度上是分离的。 尽管在复制数据和思想传递方面存在共同兴趣，但两者之间的交流多年来一直有限。 在云和互联网规模的数据管理时代，这种差距已经缩小。
例如，Gray和Lamport在2006年合作开发了Paxos Commit [71]，这是一个结合了AC和Lamport的Paxos的有趣算法。在这个交叉领域还有很多工作要做，而且这个领域中“每个人都应该知道的技术”的数量也在增加。

第四章：新的DBMS架构

由迈克尔・斯通布雷克引入

精选读物：

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik. C-store: A Column-oriented DBMS. *SIGMOD*, 2005.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling. Hekaton: SQL Server的内存优化OLTP引擎。 *SIGMOD*, 2013.

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker. 通过镜子看OLTP，我们在那里发现了什么。 *SIGMOD*, 2008.

在数据库管理系统领域，最重要的事情可能就是"一刀切"的消亡了。 直到2000年代初，传统的基于磁盘的行存储架构无处不在。 实际上，商业供应商只有一个解决方案，而所有问题都是钉子。

在过去的十五年里，发生了几次重大变革，我们将依次讨论。

首先，社区意识到列存储在数据仓库市场上明显优于行存储。 数据仓库在面向客户的零售环境中得到了早期的接受，并迅速扩展到了面向客户的数据。仓库记录了客户交易的历史信息。 实际上，这是每个客户互动的"谁-什么-为什么-何时-何地"。

传统的智慧是围绕一个中央事实表来构建数据仓库，在这个事务信息被记录下来。 围绕这个事实表的是维度表，记录可以从事实表中分解出来的信息。 在零售场景中，有关于商店、顾客、产品和时间的维度表。结果就是一个所谓的星型模式[96]。如果商店被分组成区域，那么可能会有多层维度表，形成一个雪花模式。

关键观察是事实表通常是"肥"的，通常包含一百个或更多的属性。
显然，它们也是"长"的，因为有很多很多的事实要记录。 一般来说，对数据仓库的查询是重复请求的混合（生成按店铺的月销售报告）和"即席"的请求。 例如，在零售仓库中，人们可能想知道什么在东北地区销售时发生了暴风雪，在大西洋沿岸销售时发生了飓风。

此外，没有人运行select *查询来获取事实表中的所有行。 相反，他们总是指定一个聚合函数，从表中检索出六个属性。 下一个查询检索不同的数据集，过滤条件之间几乎没有局部性。

在这种情况下，很明显列存储将从磁盘移动的数据量比行存储少16倍（6列对比100列）。 因此，它具有不公平的优势。 此外，考虑一个存储块。在列存储中，该块上只有一个属性，而行存储上有100个属性。 压缩在一个属性上明显比在100个属性上效果更好。 此外，行存储在每个记录的前面有一个头部（在SQLServer中显然是16字节）。 相比之下，列存储非常小心地没有这样的头部。

最后，基于行的执行器有一个内部循环，用于检查输出的有效性。
因此，内部循环的开销很大，每个记录检查都要付出代价。 相比之下，列存储的基本操作是检索列并挑选出符合条件的项。 因此，内部循环的开销是每个列检查一次，而不是每行检查一次。 因此，列执行器在CPU时间上更高效，并且从磁盘中检索的数据更少。 在大多数实际环境中，列存储比行存储快50-100倍。

早期的列存储包括Sybase IQ [108]，

它出现在1990年代，还有MonetDB [30]。 然而，这项技术可以追溯到1970年代 [25, 104]。 在2000年代，C-Store/Vertica作为一家资金充裕的初创公司出现，并具有高性能的实现。 在接下来的十年里，整个数据仓库市场从行存储世界转变为列存储世界。

可以说，如果Sybase在技术上投入更多，并进行多节点实现，Sybase IQ可能早些时候就可以做到这一点。 列执行器的优点在[30]中有详细讨论，尽管它很细节化且难以阅读。

第二个重大变革是主存价格的急剧下降。 目前，可以以大约$25,000购买1TB的存储器，而具有几TB的高性能计算集群可能售价约为$100K。 关键的洞察力是OLTP数据库并不是那么大。1TB是一个非常大的OLTP数据库，适合部署在主存中。 正如本节中的“透视镜”论文所指出的，当数据适合主存时，不希望运行基于磁盘的行存储，因为开销太高。

实际上，OLTP市场现在正在成为主存储DBMS市场。 再次，传统的基于磁盘的行存储方式无法竞争。 为了良好运行，需要针对并发控制、崩溃恢复和多线程开发新的解决方案，我预计OLTP架构在未来几年会发展。

我目前最好的猜测是没有人会使用传统的两阶段锁定。 基于时间戳排序或多版本的技术很可能会占主导地位。
本节的第三篇论文讨论了Hekaton，它实现了一种最先进的MVCC方案。

还必须处理崩溃恢复。 一般来说，提出的解决方案是复制和在线故障转移，这是Tandem二十年前首创的。

传统的智慧是先写日志，然后将日志传输到备份站点，然后在备份站点上进行前滚。
这种主备架构在[111]中被证明比主主架构差了3倍，主主架构只需在每个副本上运行事务。 如果使用主主架构，则必须确保每个副本上的事务按相同顺序运行。 不幸的是，MVCC无法做到这一点。 这导致人们对确定性并发控制方案产生了兴趣，在端到端系统中，这些方案可能比MVCC快得多。

无论如何，OLTP将转移到主内存部署，并且一种新的主内存DBMS类正在发展以支持此用例。

第三个展开的现象是“无SQL”运动。 本质上，有大约100个DBMS，它们支持各种数据模型，并具有以下两个特点：

1. “开箱即用”体验。 对于程序员来说，它们很容易上手并且能够做出有生产力的事情。 相比之下，RDBMS非常庞大，需要事先定义模式。

2. 对半结构化数据的支持。 如果每个记录都可以有不同属性的值，那么传统的行存储将具有非常宽的元组，并且非常稀疏，因此效率低下。

这是对商业供应商的警示，要求他们制作更易于使用并支持半结构化数据类型（如JSON）的系统。总的来说，我预计No SQL市场将随着RDBMS对上述两点的反应而与SQL市场合并。

第四次变革是Hadoop/HDFS/Spark环境的出现，在第6章中进行了讨论。

第5章：大规模数据流引擎

由Peter Bailis引入

精选读物：

Jeff Dean和Sanjay Ghemawat。MapReduce：大规模集群上的简化数据处理。 *OSDI*，2004年。

Yuan Yu，Michael Isard，Dennis Fetterly，Mihai Budiu。DryadLINQ：一种通用分布式数据并行计算系统，使用高级语言。 *OSDI*，2008年。

在过去十年中，数据管理领域发展了许多，其中MapReduce和随后的大规模数据处理系统是最具颠覆性和最具争议性的。廉价的商品存储和不断增长的数据量使得许多互联网服务供应商放弃了传统的数据库系统和数据仓库，而是构建了定制的自家引擎。 谷歌在其大规模系统上的一系列出版物，包括Google文件系统[62]，MapReduce，Chubby [32]和BigTable [37]，在市场上可能是最著名和最有影响力的。 几乎所有情况下，这些新的自家系统只实现了传统数据库中的一小部分功能，包括高级语言，查询优化器和高效执行策略。 然而，这些系统和由此产生的开源Hadoop生态系统在许多开发人员中非常受欢迎。

这导致了在这些平台上的大量投资、营销、研究兴趣和发展，这些平台至今仍在不断变化，但作为一个生态系统，它们已经开始类似于传统的数据仓库，但有一些重要的修改。 我们在这里反思这些趋势。

历史和继任者

我们的第一篇阅读是2004年的原始Google MapReduce论文。 MapReduce是一个用于简化Google规模下的并行分布式计算的库，特别是用于从爬取的页面中批量重建网络搜索索引。 当时，传统的数据仓库很难处理这样的工作量。 然而，与传统的数据仓库相比，MapReduce提供了一个非常低级的接口（两阶段数据流），与一个容错的执行策略（两阶段数据流之间的中间材料化）密切相关。同样重要的是，MapReduce被设计为一个并行编程库，而不是一个端到端的数据仓库解决方案；例如，MapReduce将数据存储委托给Google文件系统。当时，数据库界的成员谴责这种架构过于简单、低效且有限[53]。

虽然最初的MapReduce论文发布于2003年，但在2006年之前，谷歌之外几乎没有其他活动，直到雅虎开源了Hadoop MapReduce实现。 随后，出现了大量的兴趣：在一年内，包括Dryad（微软）[89]、Hive（Facebook）[156]、Pig（雅虎）[123]在内的一系列项目都在开发中。 这些系统，我们将其称为后MapReduce系统，得到了开发者的广泛关注，这些开发者主要集中在硅谷，并获得了大量的风险投资。 涉及系统、数据库和网络领域的大量研究调查了调度、故障处理、UDF查询优化和替代编程模型等问题[16]。

几乎立即，MapReduce后系统扩展了它们的接口和功能，包括更复杂的声明性接口、查询优化策略和高效的运行时。 如今的MapReduce后系统已经实现了传统关系型数据库管理系统的越来越多的功能集。最新一代的数据处理引擎，如Spark [163]、F1 [143]、Impala [98]、Tez [1]、Na-iad [119]、Flink/Stratosphere [9]、AsterixDB [10]和Drill [82]经常地：i）提供更高级的查询语言，如SQL；ii）更先进的执行策略，包括处理一般的操作符图；iii）在可能的情况下使用索引和其他结构化输入数据源的功能。在Hadoop生态系统中，数据流引擎已经成为一套更高级功能和声明性接口的基础，包括SQL [15, 156]、图处理 [64, 110]和机器学习 [63, 146]。

对流处理功能的兴趣也越来越大，重新审视了数据库社区在2000年代开创的许多概念。 一个不断发展的商业和开源生态系统已经开发出了与各种结构化和半结构化数据源的“连接器”，目录功能（例如HCatalog）以及数据服务和有限的事务能力（例如HBase）相结合的功能。这些框架中的许多功能，例如典型的查询优化器，与许多成熟的商业数据库相比还很基础，但正在快速发展。

DryadLINQ，我们在本节中选择的第二篇读物，可能最有趣的是它的接口：一组嵌入式语言绑定，用于与微软的.NET LINQ 无缝集成，以提供并行化的集合库。 DryadLINQ 通过早期的 Dryad 系统 [89] 执行查询，该系统使用基于重放的容错机制实现了任意数据流图的运行时。 虽然 DryadLINQ 仍然限制程序员使用一组无副作用的数据集转换（包括“类似 SQL”的操作），但它提供了比 Map Reduce 更高级的接口。DryadLINQ 的语言集成、轻量级容错和基本查询优化技术在后来的数据流系统中产生了影响，包括 Apache Spark [163] 和微软的 Naiad [119]。

影响和遗产

MapReduce现象至少有三个持久的影响，否则可能不会发生。 这些想法-就像分布式数据流本身一样-不一定是新颖的，但是后MapReduce数据流和存储系统的生态系统广泛增加了它们的影响力：

1.)模式灵活性。也许最重要的是，传统的数据仓库系统是封闭的花园：摄取的数据是完美的，经过策划的，并且有结构。相比之下，M*a*pR*e*duc*e*系统处理任意结构化的数据，无论是干净的还是脏的，经过策划的还是不经过策划的。没有加载步骤。这意味着用户可以先存储数据，然后再考虑如何处理数据。再加上存储成本（例如，在H*a*doop文件系统中）比传统数据仓库要便宜得多，用户可以承担更长时间的数据保留。这是与传统数据仓库的重大转变，也是“大数据”崛起和聚集的关键因素之一。越来越多的存储格式（例如，Avro，P*ar*quet, RCFile) 与列式布局等存储技术相结合，使半结构化数据和存储方面的进展（例如列式布局）相结合。与XML相比，这种最新的半结构化数据波动更加灵活。因此，提取-转换-加载（ETL）任务是后MapReduce引擎的主要工作负载。
很难过高估模式灵活性对于数据管理的现代实践的影响，从分析师到程序员和分析供应商，我们相信它在未来将变得更加重要。 然而，这种异构性并非免费：维护这样的“数据湖”是昂贵的（比存储成本更高），这是我们在第12章中深入讨论的一个主题。

2.)接口灵活性。如今，大多数用户都使用类似SQL的语言与大数据引擎进行交互。然而，这些引擎还允许用户使用多种范式进行编程。例如，一个组织机构可能使用命令式代码进行文件解析，使用SQL进行列投影，并使用机器学习子程序对结果进行聚类 - 所有这些都在一个框架内完成。像D*rya*dL*I*NQ这样的紧密、习惯用法的语言集成是司空见惯的，进一步提高了可编程性。虽然传统数据库引擎历史上支持用户定义函数，但这些新引擎的接口使用户定义计算更容易表达，并且还使用户定义计算的结果与使用传统关系构造（如SQL）表达的查询结果集成更容易。接口灵活性和集成是数据分析产品的一个强大卖点；在一个系统中结合ETL、分析和后处理的能力对于程序员来说非常方便，但对于传统B*I*工具的用户来说可能并非如此，这些工具使用传统的JDBC接口。

3.) 架构灵活性。对关系数据库管理系统（RDBMS）的一个常见批评是它们的架构过于紧密耦合：存储、查询处理、内存管理、事务处理等等紧密交织在一起，在实践中它们之间缺乏清晰的接口。相比之下，由于其自下而上的开发方式，H*a*doop生态系统已经有效地构建了一个数据仓库作为一系列模块。今天，组织可以针对原始文件系统（例如HDFS），任意数量的数据流引擎（例如Sp*ar*k），使用高级分析包（例如G*raph*L*ab* [105]，P*arame*t*er* S*erver* [101]）或通过SQL（例如Imp*ala* [98]）编写和运行程序。这种灵活性增加了性能开销，但在这个规模上混合和匹配组件和分析包的能力是前所未有的。

这种架构灵活性可能是最有趣的

对于系统构建者和供应商来说，这给他们在设计基础设施时增加了额外的自由度。

总结一下，当今分布式数据管理基础设施的主题是灵活性和异构性：存储格式的灵活性、计算范式的灵活性以及系统实现的灵活性。 其中，存储格式的异构性可能是影响最大的，因为它对新手、专家和架构师都有影响。 相比之下，计算范式的异构性主要影响专家和架构师，而系统实现的异构性主要影响架构师。 这三个方面都是数据库研究中相关且令人兴奋的发展，但市场影响和持久性仍然存在一些问题。

展望未来

从某种意义上说，MapReduce是一个短暂的、极端的架构，它打开了一个设计空间。 这种架构简单且高度可扩展，在开源领域取得了成功，让许多人意识到存在对替代解决方案和所体现的灵活性的需求（更不用说基于开源的更便宜的数据仓库解决方案的市场机会了）。 由此产生的兴趣仍然令许多人感到惊讶，这是由于许多因素，包括社区精神、巧妙的营销、经济和技术变革。 有趣的是思考这些新系统与关系型数据库之间的哪些差异是根本的，哪些是由于工程改进造成的。

如今，关于大规模数据处理的适当架构仍存在争议。 以Rasmussen等人的研究为例，他们强烈论证了中间故障容忍除了在非常大规模（100+节点）的集群中并非必要[132]。 以McSherry等人的研究为另一个例子，他们生动地说明了许多工作负载可以通过单个服务器（或线程！）高效处理，完全不需要分布式[113]。 最近，像GraphLab项目[105]这样的系统提出了领域特定系统对于性能是必要的；后续的研究，包括Grail [58]和GraphX [64]，则认为这并非必要。 最近一波的提议还提出了用于流处理、图处理、异步编程和通用机器学习的新接口和系统。这些提议是否实际上，是否需要专门的系统，还是一个分析引擎可以统治所有？ 时间会告诉我们，但我感觉有一种向整合的推动力。

最后，我们不得不提到Spark，它只有六年的历史，但在开发者中越来越受欢迎，得到了风投支持的初创公司（例如Databricks）和Cloudera和IBM等知名公司的大力支持。虽然我们将DryadLINQ作为一个后MapReduce系统的例子，因为它具有历史意义和技术深度，但Spark论文[163]是在项目初期编写的，最近的扩展包括SparkSQL [15]，也值得一读。

与Hadoop一样，Spark在相对早期的成熟阶段引起了广泛的兴趣。 今天，Spark在功能集方面仍有一段路要走，才能与传统数据仓库相媲美。 然而，它的功能集正在迅速增长，人们对Spark作为Hadoop生态系统中MapReduce的继任者的期望很高；例如，Cloudera正在努力用Spark替代Hadoop生态系统中的MapReduce [81]。 时间会告诉我们这些期望是否准确；与此同时，传统数据仓库和后MapReduce系统之间的差距正在迅速缩小，从而产生了与传统系统一样擅长数据仓库的系统，但也更加强大。

评论：迈克尔·斯通布雷克

*2015*年*10*月*26*日

最近，数据分析作为“大数据”营销宣传语的一部分引起了相当大的兴趣。 从历史上看，这意味着商业智能（BI）分析，并由BI应用程序（Cognos，Business Objects等）与关系型数据仓库（如Teradata，Verica，Red Shift，Greenplum等）进行交互。最近，它与“数据科学”联系在一起。 在这个背景下，让我们从十年前的Map-Reduce开始，这是Google为支持他们的网络爬行数据库而专门构建的。 然后，市场人员接管了基本论点：“Google很聪明；Map-Reduce是Google的下一个大事，所以一定很好”。

Cloudera，Hortonworks和Facebook在炒作Map-Reduce（以及其开源仿制品Hadoop）方面处于先锋地位。几年前，市场上充斥着Map-Reduce的热潮。 大约在同一时间，Google停止使用Map-Reduce用于其专门构建的应用程序，转而使用Big Table。大约5年后，世界其他地方才看到Google早就发现的东西；Map-Reduce不是具有广泛规模的架构。

适用性。

实际上，Map-Reduce存在以下两个问题：

1. 它不适合作为构建数据仓库产品的平台。 任何商业数据仓库产品中都没有类似Map-Reduce的接口，这是有充分理由的。 因此，数据库管理系统不希望使用这种平台。

2. 它不适合作为构建分布式应用程序的平台。 Map-Reduce接口不仅对于分布式应用程序来说不够灵活，而且使用文件系统的消息传递系统速度太慢，不具有足够的吸引力。

当然，这并没有阻止Map-Reduce供应商的步伐。他们只是将自己的平台重新命名为HDFS（一个文件系统），并基于HDFS构建了不包含Map-Reduce的产品。例如，Cloudera最近推出了Impala，它是一个SQL引擎，不是基于Map-Reduce构建的。事实上，Impala实际上也没有使用HDFS，而是选择直接读写底层的本地Linux文件。HortonWorks和Facebook也有类似的项目正在进行中。 因此，Map-Reduce的拥护者已经转向了SQL，作为一个接口，Map-Reduce已经成为历史。 当然，当SQL引擎使用HDFS时会存在严重的问题，所以目前还不清楚它是否会成功，这还有待观察。 无论如何，Map-Reduce-HDFS市场将与SQL-数据仓库市场合并，最好的系统将获胜。 总之，作为一个分布式系统平台，Map-Reduce已经失败了，供应商们正在将HDFS作为数据仓库产品的文件系统使用。

这将引导我们到Spark。Spark的原始论点是它是Map-Reduce的更快版本。它是一个具有快速消息传递接口的主内存平台。因此，当用于分布式应用程序时，它不应该遭受Map-Reduce的性能问题。 然而，根据Spark的首席作者Matei Zaharia的说法，超过70%的Spark访问是通过SparkSQL进行的。实际上，Spark被用作SQL引擎，而不是分布式应用程序平台！在这种情况下，Spark存在身份问题。 如果它是一个SQL平台，那么它需要一些机制来持久化、索引、共享用户之间的主内存、元数据目录等，以在SQL/数据仓库领域具有竞争力。 看起来Spark很可能会成为一个数据仓库平台，沿着Hadoop的同样道路发展。

另一方面，30%的Spark访问不是通过SparkSQL进行的，主要是来自Scala。这可能是一个分布式计算负载。在这种情况下，Spark是一个合理的分布式计算平台。 然而，还有一些问题需要考虑。 首先，平均数据科学家会进行数据管理和分析的混合。 更高的性能来自于两者的紧密耦合。在Spark中，由于Spark的数据格式在这两个任务中并不一定是共同的。 其次，Spark目前仅支持主内存。 随着时间的推移，可扩展性要求将被解决。 因此，看到Spark未来的发展将是有趣的。

总结一下，我想提供以下要点：

- 仅仅因为谷歌认为某个想法很好
  does not mean you should adopt it.
- 不要相信所有的营销宣传，要弄清楚任何给定产品的实际好处。这尤其适用于性能声明。
- 程序员社区对“下一个闪亮的东西”有一种热恋。 这很可能会在你的组织中引起“流失”，因为闪亮的东西的“半衰期”可能非常短。

第六章：弱隔离和分布

由Peter Bailis引入

精选读物：

Atul Adya, Barbara Liskov, and Patrick O'Neil. 广义隔离级别定义. *ICDE*, 2000.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin,Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon的高可用键值存储. *SOSP*, 2007.

Eric Brewer. CAP十二年后："规则"如何改变。*IEEE*计算机，45，2（2012年）。

传统的数据库智慧认为，可串行化事务是并发编程问题的标准解决方案，但在实际数据库中很少出现这种情况。 实际上，数据库系统主要实现非串行化的并发控制，使用户的事务可能不会按照某种串行顺序执行。 在本章中，我们讨论为什么使用所谓的"弱隔离"如此普遍，这些非串行化的隔离模式实际上是什么，以及为什么对它们进行推理如此困难。

概述和普及

即使在数据库系统的早期（参见第3章），系统构建者也意识到实现串行化是昂贵的。 事务必须按顺序执行的要求对数据库的并发程度产生了深远的影响。 如果事务访问数据库中的不相交数据集，串行化实际上是"免费"的：在这些不相交的访问模式下，可串行化的调度允许数据并行处理。 然而，如果事务争用相同的数据项，在最坏的情况下，系统无法以任何并行方式处理它们。 这个属性对于串行化是基本的，与实际实现无关：因为事务不能在所有工作负载下安全地独立进行进展（即，它们必须协调），串行化的任何实现可能实际上需要串行执行。 实际上，这意味着事务可能需要等待，降低吞吐量同时增加延迟。 事务处理专家Phil Bernstein表示，与最常见的单节点数据库相比，串行化通常会导致性能下降三倍。

被称为读提交的弱隔离级别[29]。 根据实现的不同，可串行化也可能导致更多的中止、重新启动的事务和/或死锁。 在分布式数据库中，这些成本会增加，因为网络通信是昂贵的，增加了执行串行关键部分所需的时间（例如，持有锁）；我们观察到在不利条件下会有多个数量级的性能惩罚[20]。

因此，数据库系统设计者通常不实现可串行化，而是实现了更弱的模型。 在弱隔离下，事务不能保证观察到可串行化的行为。 相反，事务将观察到一系列异常（或"现象"）：这些行为在串行执行中是不可能发生的。 具体的异常取决于提供的模型，但示例异常包括读取另一个事务生成的中间数据，读取中止的数据，在执行同一事务期间读取同一项的两个或更多不同的值，以及由于对同一项的并发写入而"丢失"某些事务的效果。

这些弱隔离模式出奇地普遍。 在最近对十八个SQL和"NewSQL"数据库的调查中[18]，我们发现只有三个数据库默认提供串行化，并且有八个数据库（包括Oracle和SAP的旗舰产品）根本不提供串行化！这种情况进一步复杂化，因为术语的使用经常不准确：例如，Oracle的"串行化"隔离保证实际上提供的是快照隔离，一种弱隔离模式[59]。

供应商之间也存在着竞争。 据传，当供应商A（事务处理市场的主要参与者）将默认隔离模式从串行化更改为读提交时，供应商B仍然默认为串行化，开始在与供应商A的竞争中失去销售合同。 供应商B的数据库明显较慢，那么为什么客户会选择B而不是A呢？

毫不奇怪，供应商B现在也默认提供读提交隔离。
毫不奇怪，供应商B现在也默认提供读提交隔离。

关键挑战：对异常进行推理

弱隔离的主要问题是，根据应用程序的不同，弱隔离的异常可能导致应用程序级的不一致性：在可串行化执行中，每个事务保持的不变量在弱隔离下可能不再成立。 例如，如果两个用户同时尝试从一个银行账户中取款，并且他们的事务在允许并发写入相同数据项的弱隔离模式下运行（例如常见的读提交模型），则用户可能成功地取出比账户中实际金额更多的钱（即，每个用户读取当前金额，计算出他们取款后的金额，然后将“新”的总额写入数据库）。 这不是一个假设的情景。 在最近的一个生动例子中，攻击者系统地利用了Flexcoin比特币交易所中的弱隔离行为；通过反复和程序化地触发Flexcoin应用程序中的非事务性读取-修改-写入行为（在读提交隔离下的漏洞和在更复杂的访问模式下的快照隔离下），攻击者能够取出比应有的比特币更多的数量，从而使交易所破产[2]。

也许令人惊讶的是，我与开发人员交谈时，他们对于事务的使用并不知道自己正在运行在非可串行化隔离级别下。 事实上，在我们的研究中，我们发现许多基于开源ORM的应用程序都假设了可串行化隔离级别，这在部署在通用数据库引擎上时可能导致一系列应用完整性违规问题[19]。 那些意识到弱隔离级别的开发人员往往会在应用程序层面采用一系列替代技术，包括显式获取锁（例如SQL的“SELECT FOR UPDATE”）和引入虚假冲突（例如在快照隔离下写入虚拟键）。这种做法容易出错，并且抵消了事务概念的许多好处。

不幸的是，关于弱隔离级别的规范通常是不完整、模糊甚至不准确的。
这些规范有着悠久的历史，可以追溯到1970年代。 虽然它们随着时间的推移有所改进，但它们仍然存在问题。

最早的弱隔离模式是以操作方式指定的：正如我们在第3章中所看到的，流行的模型如“读已提交”最初是通过修改读锁的持续时间来发明的[72]。 “读已提交”的定义是：“读锁持续时间短，写锁持续时间长。”

ANSI SQL标准后来试图提供对几种弱隔离模式的实现无关描述，不仅适用于基于锁的机制，还适用于多版本和乐观方法。然而，正如Gray等人在[27]中所描述的那样，SQL标准既模糊又不明确：对于英语描述，存在多种可能的解释，并且形式化未能捕捉到基于锁的实现的所有行为。 此外，ANSI SQL标准并未涵盖所有隔离模式：例如，供应商已经开始提供提供快照隔离（并将其标记为可串行化！）的生产数据库，而Gray等人在其1995年的论文中定义了它。 （可悲的是，截至2015年，ANSI SQL标准仍然没有改变。）为了进一步复杂化问题，Gray等人在1995年修订的形式化方法也存在问题：它侧重于与锁相关的语义，并排除了在多版本并发控制系统中可能被认为是安全的行为。 因此，Atul Adya在他1999年的博士论文[6]中引入了迄今为止

最好的弱隔离形式化方法。 Adya的论文将多版本序列化图的形式化方法应用于弱隔离领域，并根据这些图的限制描述异常情况。 我们包括Adya在ICDE 2000会议上的相应论文，但是对于隔离爱好者来说，应该参考完整的博士论文。 不幸的是，Adya的模型在某些情况下仍然不明确（例如，如果没有涉及读取，那么G0到底意味着什么？），并且这些保证的实现在不同的数据库中有所不同。

即使有完美的规范，弱隔离仍然是一个真正的挑战。 为了确定弱隔离是否“安全”，程序员必须在脑海中将他们的应用级一致性问题转化为低级读写行为[11]。 即使对于经验丰富的并发控制专家来说，这也是非常困难的。 事实上，人们可能会想知道，如果串行化被破坏，事务的好处还有什么？

为什么理解读提交隔离比没有隔离更容易？ 考虑到有多少像Oracle这样的数据库引擎在弱隔离下运行，现代社会是如何正常运作的 - 无论用户是预订航班、管理医院还是进行股票交易？ 文献中很少提供线索，这对于事务概念在实践中的成功提出了严重的问题。

我遇到的最有说服力的论点是，为什么弱隔离在实践中似乎是“可以接受的”是因为今天很少有应用程序经历高并发。 没有并发性，大多数实现弱隔离的结果是可串行化的。 这反过来又产生了一系列有益的研究结果。 即使在分布式环境中，弱隔离的数据库也能提供“一致”的结果：例如，在Facebook，他们的最终一致性存储返回的结果中只有0.0004%是“过时的”[106]，其他人也发现了类似的结果[23, 159]。 然而，尽管对于许多应用程序来说，弱隔离显然没有问题，但它可能存在问题：正如我们的Flexcoin示例所示，鉴于错误的可能性，应用程序编写者必须在考虑（或明确忽略）与并发相关的异常时保持警惕。

弱隔离、分布和“NoSQL”

随着互联网规模服务和云计算的兴起，弱隔离变得更加普遍。 正如我之前提到的，分布加剧了串行化的开销，并且在部分系统故障（例如服务器崩溃）的情况下，事务可能会无限期地停滞。 随着越来越多的程序员开始编写分布式应用程序并使用分布式数据库，这些问题变得越来越普遍。

过去十年见证了一系列针对分布环境进行优化的新数据存储的引入，统称为“NoSQL”。"NoSQL"这个标签不幸地被过度使用，涵盖了这些存储的许多方面，从缺乏字面SQL支持到更简单的数据模型（例如键值对）和几乎没有事务支持。 如今，与MapReduce类似的系统（第5章）一样，NoSQL存储正在添加许多这些功能。 然而，一个显著的基本区别是，这些NoSQL存储经常专注于通过更弱的模型提供更好的操作可用性，并明确关注容错性。 （有点讽刺的是，尽管NoSQL存储通常与非串行化的保证相关联，但是一些关系型数据库管理系统默认不提供串行化功能。

作为这些NoSQL存储的一个例子，我们包括了一篇关于亚马逊的Dynamo系统的论文，该论文在SOSP 2007上发表。 Dynamo系统旨在为亚马逊的购物车提供高可用性和低延迟的操作。 这篇论文在技术上非常有趣，它结合了多种技术，包括仲裁复制、Merkle树反熵、一致性哈希和版本向量。 该系统完全不支持事务，不提供任何类型的原子操作（例如比较和交换），并依赖于应用程序编写者来协调不一致的更新。 在极限情况下，任何节点都可以更新任何项（通过暗示式转交）。

通过使用合并函数，Dynamo采用了一种“乐观复制”的策略：先接受写入，然后再协调不一致的版本。 一方面，向用户呈现一组不一致的版本比像Read Committed隔离级别那样简单地丢弃一些并发更新更友好。 另一方面，程序员必须思考合并函数的问题。

这引发了许多问题：什么是适合应用程序的合并方式？ 我们如何避免丢弃已提交的数据？ 如果一个操作本来就不应该同时进行，怎么办？ 一些开源的Dynamo克隆，如Apache Cassandra，不提供合并操作符，而是根据时间戳选择“获胜”的写入。 其他的，如Basho Riak，采用了自动合并数据类型的“库”，比如计数器，称为可交换复制数据类型[142]。

Dynamo也不对读取的新鲜度做出承诺。 相反，它保证，如果写入停止，最终所有数据项的副本将包含相同的写入集合。 这种最终一致性是一种非常弱的保证：从技术上讲，一个最终一致性的数据存储可以在无限的时间内返回过时的（甚至是垃圾的）数据[22]。 实际上，数据存储部署经常返回最新的数据[159, 23]，但是，用户仍然必须考虑非可串行化的行为。 此外，在实践中，许多存储提供了称为“会话保证”的中间隔离形式，确保用户读取自己的写入（但不包括其他用户的写入）；有趣的是，这些技术在20世纪90年代早期作为Bayou移动计算项目的一部分开发，并最近再次引起关注[154, 153]。

权衡和CAP定理

我们还包括了Brewer对CAP定理的12年回顾。CAP定理最初是在Brewer建立Inktomi，第一个可扩展的搜索引擎之一的时候提出的，它简洁地描述了协调（或“可用性”）和强一致性等强保证之间的权衡。 尽管早期的研究已经描述了这种权衡[91, 47]，但CAP定理成为了2000年代中期开发者的口号，并产生了相当大的影响。 Brewer的文章简要讨论了CAP的性能影响，以及在不依赖协调的情况下保持一些一致性标准的可能性。

可编程性和实践

正如我们所看到的，弱隔离是一个真正的挑战：尽管我们对其行为知之甚少，但其性能和可用性的好处使其在部署中非常受欢迎。 即使有一个完美的规范，现有的弱隔离形式仍然非常难以理解。 要决定弱隔离是否“安全”，程序员必须将他们的应用级一致性问题转化为低级的读写行为[11]。 即使对于经验丰富的并发控制专家来说，这也是非常困难的。

因此，我认为有一个严肃的机会来研究那些不受串行化性能和可用性开销限制的语义，而且比现有的保证更直观、可用和可编程。 弱隔离在历史上一直是非常具有挑战性的，但这并非必然。 我们和其他人发现，包括索引和视图主要在内的几个高价值用例维护、约束维护和分布式聚合等操作，实际上并不需要协调来实现“正确”的行为；因此，对于这些用例，串行化是过度的 [17, 21, 136, 142]。 也就是说，通过为数据库提供关于应用程序的额外知识，数据库用户可以两全其美。进一步识别和利用这些用例是一个值得研究的领域。

结论

总之，弱隔离由于其许多好处而普遍存在：较少的协调、更高的性能和更大的可用性。 然而，其语义、风险和使用在学术环境中甚至都不被很好地理解。 这尤其令人困惑，考虑到许多人认为可串行化事务处理已经是一个“解决的问题”。弱隔离甚至更值得得到彻底的处理。 正如我所强调的，仍然存在许多挑战：现代系统如何工作，用户如何编写弱隔离的程序？ 目前，我提供以下要点：

- 非可串行隔离在实践中很常见
  （无论是传统的关系型数据库管理系统还是最近的NoSQL新兴技术）由于其与并发相关的优势。
- 尽管如此，许多现有的非可串行隔离公式规范不清楚且难以使用。
- 对新形式的弱隔离的研究表明如何在不牺牲串行性的情况下保留有意义的语义并提高可编程性。

第七章：查询优化

由乔・海勒斯坦引入

精选读物：

Goetz Graefe和William J. McKenna。Volcano优化器生成器：可扩展性和高效搜索。 *ICDE*，1993年。

Ron Avnur和Joseph M. Hellerstein。Eddies：持续自适应查询处理。 *SIGMOD*，2000年。

Volker Markl，Vijayshankar Raman，David Simmen，Guy Lohman，Hamid Pirahesh，Miso Cilimdzic。通过渐进优化实现鲁棒查询处理。 *SIGMOD*，2004年。

查询优化是数据库技术的标志性组成部分之一 - 连接声明性语言与高效执行的桥梁。 查询优化器被认为是实现良好的DBMS中最困难的部分之一，因此对于成熟的商业DBMS来说，它们仍然是一个明显的区别因素。相比之下，开源关系数据库优化器中最好的优化器受到限制，并且一些优化器相对较为简单，只适用于最简单的查询。

重要的是要记住，没有一个查询优化器真正产生“最优”的计划。 首先，它们都使用估算技术来猜测真实计划的成本，众所周知，这些估算技术的错误可能会膨胀 - 在某些情况下甚至糟糕到随机猜测[88]。 其次，优化器使用启发式算法来限制选择的计划搜索空间，因为这个问题是NP难的[86]。 最近受到重视的一个假设是传统的使用2表连接运算符；在某些情况下，这被证明在理论上劣于新的多路连接算法[121]。

尽管存在这些注意事项，关系查询优化已经被证明是成功的，并且使关系数据库系统能够在实践中很好地服务于各种常见用例。 数据库供应商已经投入了多年的时间来确保他们的优化器在各种用例上可靠地运行。 用户已经学会了在连接数量上的限制。

优化器在很大程度上仍然使声明式SQL查询成为大多数情况下比命令式代码更好的选择。

除了难以构建和调优之外，严重的查询优化器还有一个趋势，随着时间的推移变得越来越复杂，因为它们不断发展以处理更丰富的工作负载和更多的边界情况

数据库查询优化的研究文献实际上是一个独立的领域，充满了技术细节，其中许多细节已经在成熟供应商如IBM和Microsoft的文献中被研究人员讨论过，他们与产品团队密切合作。 对于本书，我们关注的是大局观：已经考虑过的主要查询优化架构以及它们如何随时间重新评估。 对于本书，我们关注的是大局观：已经考虑过的主要查询优化架构以及它们如何随时间重新评估。

Volcano/Cascades

我们从最先进的技术开始。在数据库研究的早期，有两种查询优化的参考架构，涵盖了今天大多数严肃的优化器实现。 第一种是Selinger等人在第三章中描述的System R优化器。 System R的优化器是教科书上的材料，在许多商业系统中实现；每个数据库研究人员都应该详细了解它。 第二种是Goetz Graefe及其合作者在一系列研究项目中改进的架构：Exodus、Volcano和Cascades。Graefe的工作在研究文献或教科书中没有像System R的工作那样频繁出现，但在实践中被广泛使用，特别是在Microsoft SQL Server中，但据说也在其他商业系统中使用。 Graefe关于这个主题的论文有一些内部人的特色，针对那些了解并关心实现查询优化器的人。 我们选择了Volcano论文作为这本书中最易理解的代表作，但热衷者也应该阅读Cascades论文[65]——它不仅提出并解决了Volcano的一些详细缺陷，而且是最新（因此是标准）的方法参考。 最近，出现了两个开源的Cascades风格的优化器：Greenplum的Orca优化器-

timizer现在是Greenplum开源项目的一部分，并且Apache Calcite是一个优化器，可以与多个后端查询执行器和语言一起使用，包括LINQ。

Graefe的优化器架构有两个显著的特点。 首先，它明确设计为可扩展的。 Volcano值得赞赏，因为它在MapReduce和大数据堆栈出现之前就已经展示了数据流的概念对于各种数据密集型应用非常有用。 因此，Graefe的优化器不仅仅用于将SQL编译为数据流迭代器的计划。 它们可以为其他输入语言和执行目标进行参数化；这在最近几年中非常相关，因为专门的数据模型和语言在一方面得到了发展（参见第2章和第9章），而专门的执行引擎在另一方面得到了发展（第5章）。 这些优化器中的第二个创新是使用自上而下或目标导向的搜索策略来找到最便宜的计划。 这种设计选择与Graefe的设计中的可扩展性API相关，但这并非固有的：Starburst系统展示了如何为Selinger的自下而上算法实现可扩展性。 这种“自上而下”与“自下而上”的查询优化辩论在双方都有支持者，但没有明确的胜者；类似的自上而下/自下而上辩论在递归查询处理文献中也出现，并且没有明确的胜者。 热衷者会注意到，这两个文献领域-递归查询处理和查询优化器搜索-在Evita Raced优化器中直接连接在一起，该优化器通过使用递归查询作为实现优化器的语言来实现自上而下和自下而上的优化器搜索。

自适应查询处理

到了20世纪90年代末，一些趋势表明查询优化的整体架构值得进行重大反思。这些趋势包括：

- 流数据上的连续查询。
- 数据探索的交互式方法，如在线聚合。
- 查询数据源位于DBMS之外且不提供可靠统计信息或性能。
- 不可预测和动态的执行环境，包括弹性和多租户设置以及传感器网络等广泛分布的系统。
- 查询中的不透明数据和用户定义函数，统计信息只能通过观察行为来估计。

此外，对于多操作符查询，计划成本估计经常不稳定，这也是一个持续存在的实际关注点[88]。 由于这些趋势，人们对处理查询的自适应技术产生了兴趣，其中执行计划可以在查询过程中更改。 我们在自适应查询处理的设计空间中提出了两个互补的观点；还有一篇更全面的综述[52]。

Eddies

关于涡流的工作，由我们的第二篇论文代表，对适应性的问题进行了深入探讨：如果查询“重新规划”必须在执行过程中发生，为什么不完全消除规划和执行之间的架构区别呢？ 在涡流方法中，优化器被封装为一个数据流操作符，它本身被插入到其他数据流边缘中。 它可以监视沿着这些边缘的数据流速率，因此它对它们的行为有动态的了解，并记录任何它关心的历史。 通过持续的信息流，它可以通过数据流路由动态控制查询规划的其他方面：交换操作符的顺序是通过元组通过操作符的路由来确定的（我们在这里包括的第一篇涡流论文的重点）物理操作符的选择（例如连接算法、索引选择）是通过在流中路由元组到多个备选的、潜在冗余的物理操作符中来确定的[129, 51]操作符的调度是通过缓冲输入并决定将输出传递给下一个操作符来确定的[131]。 作为扩展，可以通过干预它们的流程并共享公共操作符来安排多个查询[109]。 涡流在查询操作符的进行中拦截正在进行的数据流，将数据从输入传输到输出。因此，涡流路由的实现效率很重要；Deshpande在这方面进行了实现增强[50]。 这种流水线方法的优点是，涡流可以在执行过程中自适应地改变策略。

流水线操作符，例如连接操作符，对于查询操作符非常长寿（如流式系统）或非常糟糕的选择，在运行完成之前应该被放弃。 有趣的是，原始的Ingres优化器也具有根据每个元组做出某些查询优化决策的能力[161]。

渐进优化

IBM在本节中的第三篇论文代表了一种更具进化性的方法，它通过增加适应性功能来扩展了Sys-R风格的优化器；这种通用技术由Kabra和DeWitt [93]首创，但在这里得到了更全面的处理。 虽然Eddies专注于操作符内部的重新优化（当数据“在运动”时），但这项工作专注于操作符之间的重新优化（当数据“静止”时）。 一些传统的关系操作符，包括排序和大多数哈希连接，是阻塞的：它们在产生任何输出之前会消耗它们的整个输入。 这在输入被消耗后提供了一个机会，可以将观察到的统计数据与优化器的预测进行比较，并使用传统的查询优化技术重新优化查询计划的“剩余”部分。 这种方法的缺点是，在操作符消耗其输入时不进行重新优化，因此不适用于流式查询、对称哈希连接等流水线操作符，也不适用于初始计划中存在选择不当操作符的长时间运行的关系查询，例如从DBMS外部访问数据源而不提供有用统计信息的情况[116, 157]。

值得注意的是，这两种自适应架构原则上可以共存：涡旋只是一个数据流操作符，意味着传统优化器可以生成一个查询计划，其中涡旋连接一组流操作符，并且在数据流的阻塞点进行重新优化，就像我们的第三篇论文中所描述的那样。

讨论

这引出了对当前数据流架构的讨论，特别是在开源大数据堆栈中的趋势。 谷歌MapReduce将数据在运动中的适应性的讨论推迟了十年，通过将阻塞操作符作为执行模型的一部分。

容错机制。 在2000年代中后期，几乎不可能就优化数据流管道进行理性对话，因为这与Google/Hadoop的容错模型不一致。

在过去几年中，关于大数据执行框架的讨论突然大开，部署了各种各样的数据流和查询系统，它们之间的相似性大于差异性（Tenzing、F1、Dremel、DryadLINQ、Naiad、Spark、Impala、Tez、Drill、Flink等）。请注意，上述自适应优化的动机问题在当今的大数据讨论中非常热门，但并未得到很好的处理。

更一般地说，我认为“大数据”社区在研究和开源方面对查询优化的关注太慢了，这对当前系统和查询优化领域都是不利的。 首先，手动规划的MapReduce编程模型的讨论时间比应有的要长。 Hadoop和系统研究社区花了很长时间才接受像SQL或LINQ这样的声明性语言作为一个好的通用接口，即使同时保持低级别的MapReduce风格的数据流编程作为一种特殊情况的“快速路径”。 更令人困惑的是，即使社区开始构建像Hive这样的SQL接口，查询优化仍然是一个很少讨论且实现不佳的话题。 也许是因为查询优化器比查询执行器更难构建得好。 或许这是商业和开源数据库之间历史上质量差异的后果。 MySQL是过去十年中开源数据库技术的事实上的参考，具有天真的启发式优化器。 也许由于这个原因，许多（大多数？）开源大数据开发人员不理解或不信任查询优化器技术。

无论如何，这个潮流正在大数据社区中转变。 声明性查询已经成为大数据的主要接口，并且在几乎所有项目中都在努力构建至少一个1980年代的优化器。 鉴于我上面提到的问题列表，我相信在未来几年中，我们还将看到更多创新的查询优化方法在新系统中得到应用。

第8章：交互式分析

由乔・海勒斯坦引入

精选读物：

Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman. 高效实现数据立方体。 *SIGMOD*，1996年。

Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton. 一种基于数组的同时多维聚合算法。 *SIGMOD*，1997年。

Joseph M. Hellerstein, Ron Avnur, Vijayshankar Raman. 在线查询处理的Informix控制。
数据挖掘与知识发现，4(4)，2000年，281-314。

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. *EuroSys*, 2013.

几十年来，大多数数据库工作负载被分为两类：(1)许多小型的“事务处理”查询，在大型数据库中查找和更新少量项目，以及(2)较少的大型“分析”查询，用于汇总大量数据进行分析。 本节关注的是加速第二类查询的思路，特别是以交互速度回答这些查询，并允许对数据进行汇总、探索和可视化。

多年来，工业界一直在进行大量的噱头游戏，以捕捉这种后期工作负载的一部分或全部，从“决策支持系统”(DSS)到“在线分析处理”(OLAP)，再到“商业智能”(BI)，再到“仪表盘”，以及更一般的“分析”。 这些标签已经与数十亿美元的收入联系在一起，因此多年来，营销人员和行业分析师努力定义、区分，有时试图颠覆它们。 到现在为止，术语有点混乱。 有兴趣的读者可以查阅维基百科，评估这些流行词汇的常识以及它们可能的不同之处；请注意，这不是一个科学上令人满意的练习。

在这里，我将尽量保持事情简单和技术上的基础。

人类认知无法处理大量的原始数据。 为了让人类理解数据，数据必须以某种方式被“提炼”为相对较少的一小组记录或视觉标记。 通常通过对数据进行分区并在分区上运行简单的算术聚合函数来完成，将SQL的“GROUP BY”功能视为一个典型的模式[1]。 随后，通常需要将数据可视化，以便用户将其与手头的任务相关联。

本章我们要解决的主要挑战是以交互速度运行大规模的分组/聚合查询，即使在无法迭代处理与查询相关的所有数据的情况下。

我们如何使查询运行时间少于查看数据所需的时间？ 实际上只有一个答案：在不查看（全部）数据的情况下回答查询。 这个想法有两个变体：预计算：如果我们事先了解查询工作负载的一些

信息，我们可以以各种方式提炼数据，以便支持对某些查询的快速答案（准确或近似）。 这个想法的最简单版本是预先计算一组查询的答案，并且只支持这些查询。

我们在下面讨论更复杂的答案。 采样：如果我们无法提前预测查询，我们唯一的选择就是在查询时查看数据的子集。 这相当于从数据中进行采样，并根据样本来近似得出真实答案。

本节的论文重点讨论这两种方法之一或两种方法的结合。

我们的前两篇论文讨论了数据库的什么问题。

[1]精通数据库的人对GROUP BY和聚合视为理所当然。 在统计编程包（例如R的plyr库或Python的pandas）中，这显然是一个相对较新的问题，被称为“分割-应用-合并策略”。在这种情况下的一个问题是需要同时支持数组和表格表示法。

社区将其称为“数据立方体”[DataCubes]。 数据立方体最初由独立的查询/可视化工具支持，称为在线分析处理（OLAP）系统。 这个名称是由关系数据库先驱Ted Codd提出的，他被聘为OLAP供应商Essbase（后来被Oracle收购）的顾问。这不是Codd更学术的努力之一。

早期的OLAP工具采用了纯粹的“预计算”方法。 它们摄取了一个表，并计算并存储了该表上一系列GROUP BY查询的答案：每个查询都按不同的列子集进行分组，并计算非分组数值列的汇总聚合。 例如，在一张汽车销售表中，它可以显示按制造商的总销售额，按型号的总销售额，按地区的总销售额，以及按这些属性的任意2或3个组合的总销售额。 图形用户界面允许用户交互地浏览分组查询结果空间，这个查询空间后来被称为数据立方体[2]。 最初，OLAP系统被宣传为独立的“多维数据库”，与关系数据库有根本的不同。 然而，Jim Gray和关系数据库行业的一些作者联合解释了数据立方体的概念如何适应关系上下文[68]，该概念随后作为单个查询构造进入了SQL标准中：‘CUBE BY’。还有一种标准的SQL替代方案，称为MDX，用于OLAP目的，更简洁。 数据立方体的一些术语已经成为常用语，特别是“向下钻取”到细节和“向上卷取”到更粗略的摘要。

一个天真的关系预计算方法对于预先计算完整的数据立方体不具有良好的可扩展性。 对于具有k个潜在分组列的表，这种方法需要运行并存储2^k个GROUP BY查询的结果，每个子集的列都需要一个查询。 每个查询都需要对表进行完整的遍历。

我们的第一篇论文由Harinarayan、Rajaraman和Ullman撰写，它减少了这个空间：它选择了立方体中值得预计算的一部分查询；然后使用这些查询的结果来计算立方体中任何其他查询的结果。 这篇论文是该领域中引用最多的论文之一，部分原因是它早期观察到了数据立方体的结构。

问题是一个包含关系格。 这个格结构是他们解决方案的基础，并且在许多其他数据立方体论文（包括我们的下一篇论文）以及某些数据挖掘算法（如关联规则，也称为频繁项集）[7]中反复出现。 在OLAP领域工作的每个人都应该读过这篇论文。

我们的第二篇论文由赵、德什潘德和诺顿撰写，重点关注在立方体中实际计算的结果。 该论文采用了一种“基于数组”的方法：即假设数据存储在类似Essbase的稀疏数组结构中，而不是关系表结构，并提出了一种利用该结构的非常快速的算法。 然而，令人惊讶的是，即使对于关系表，将表格转换为数组以运行此算法也是值得的，而不是运行（效率远低于此的）传统关系算法。 这大大扩展了查询引擎的设计空间。 这意味着您可以将数据模型与查询引擎的内部模型解耦。 因此，特定用途的“引擎”（在这种情况下是多维OLAP）可以通过嵌入到更通用的引擎（在这种情况下是关系型）中增加价值。 在OLAP战争几年后，斯通布雷克开始主张数据库引擎“一刀切”不适用于所有情况，因此专用的数据库引擎（类似Essbase）确实很重要[149]。 这篇论文是这种推理的一个例子：聪明的专用技术得到了发展，如果它们足够好，它们也可以在更一般的环境中取得回报。 在这条线的两侧进行创新-专业化和泛化-多年来取得了良好的研究成果。 同时，任何构建查询引擎的人都应该记住数据和操作的内部表示可以是API表示的超集。

与这个问题相关的是，由于数据库压缩和摩尔定律的推进，分析型数据库在过去十年中变得更加高效。斯通布雷克告诉我，列存储使得OLAP加速器变得无关紧要。 这是一个有趣的论点，尽管市场尚未验证。 供应商仍然构建立方引擎，商业智能工具通常将它们作为加速器实现在关系数据库和Hadoop之上。我们第一篇论文中的缓存技术仍然相关。但是实时查询处理的权衡

[2]请注意，这个想法并不是起源于数据库。 在统计学和电子表格中，有一个古老而众所周知的交叉表或交叉制表（crosstab）的概念。

高性能分析型数据库技术和数据立方技术之间的关系可能值得重新考虑。

我们关于“在线聚合”的第三篇论文从与OLAP相反的领域开始探索，试图通过产生逐步细化的近似答案来快速处理即席查询，而无需预先计算。 这篇论文的灵感来自于人们每天在收集证据做决策时所进行的分类，我们通常对何时停止评估并采取行动做出定性决策。具体的数据中心示例包括选举报道中提供的“初步结果”，或者在低带宽连接上的图像多分辨率传递，在这两种情况下，我们对可能发生的事情有足够清晰的认识，远在过程完成之前。

在线聚合通常利用抽样来逐步改进结果。 这不是数据库抽样提供近似查询答案的第一次（也不是最后一次）使用。 （Frank Olken的论文[122]是数据库抽样的一个早期必备参考。）但是在线聚合帮助开启了一个持续进行的关于近似查询处理的工作序列，这在大数据和按需结构的当前时代尤为重要。

我们在这里包括了关于在线聚合的第一篇论文。 要欣赏这篇论文，重要的是要记住，当时的数据库长期以来一直在“正确性”的神话下运行，这在今天的研究环境中有点难以理解。 直到21世纪左右，计算机被普通大众和商业界视为准确、确定性计算的引擎。 像“垃圾进，垃圾出”这样的短语是为了提醒用户将“正确”的数据输入计算机，以便它能够完成工作并产生“正确”的输出。 一般来说，计算机不被期望产生“粗糙”的近似结果。

因此，本文首先讨论的是大规模分析查询中完全准确性的重要性，并且用户应该能够以灵活的方式平衡准确性和运行时间。这种思路很快导致了三个需要协同工作的研究方向：快速查询处理、统计近似和用户界面设计。 这三个主题之间的相互依赖构成了一个有趣的设计空间，研究人员和产品至今仍在探索其中。

我们在这里包括了关于在线聚合的第一篇论文。 要欣赏这篇论文，重要的是要记住，当时的数据库长期以来一直在“正确性”的神话下运行，这在今天的研究环境中有点难以理解。 直到21世纪左右，计算机被普通大众和商业界视为准确、确定性计算的引擎。 像“垃圾进，垃圾出”这样的短语是为了提醒用户将“正确”的数据输入计算机，以便它能够完成工作并产生“正确”的输出。 一般来说，计算机不被期望产生“粗糙”的近似结果。 在这篇论文中，它还为标准SQL聚合提供了统计估计器，并展示了如何使用标准RDBMS通过“查询调参”实现分层抽样，以使GROUP BY查询中以不同的速率对不同的组进行抽样。

该领域的后续论文探讨了将在线聚合与查询处理中的许多其他标准问题集成在一起，其中许多问题令人惊讶：连接、并行处理、子查询，以及最近的MapReduce和Spark等大数据系统的具体细节。IBM和Informix在上世纪90年代末都进行了在线聚合的商业努力，微软也在近似查询处理方面有研究议程。 但是，这些努力都没有进入市场。当时的一个原因是“数据库客户不会容忍错误答案”的顽固观念。 另一个更有说服力的原因与用户界面与查询引擎和近似处理的耦合有关。当时，许多商业智能供应商独立于数据库供应商。因此，数据库供应商通常无法在一般情况下“拥有”最终用户体验，并且无法

通过标准API直接向用户提供在线聚合功能。例如，传统的查询游标API不允许对同一查询进行多次近似，并且不支持与聚合列相关的置信区间。 当时市场的结构不支持同时涵盖后端和前端的激进新技术。 数据库供应商通常无法在一般情况下“拥有”最终用户体验，并且无法通过标准API直接向用户提供在线聚合功能。例如，传统的查询游标API不允许对同一查询进行多次近似，并且不支持与聚合列相关的置信区间。 当时市场的结构不支持同时涵盖后端和前端的激进新技术。 当时市场的结构不支持同时涵盖后端和前端的激进新技术。

这些因素中的许多已经发生了变化，而在线聚合正在研究和工业界中受到重新关注。 第一个动机毫不奇怪地是对大数据的兴趣。大数据不仅在容量上很大，而且在格式和用途上有广泛的“多样性”，这意味着在用户想要进行分析之前可能无法解析和理解。 对于大数据的探索性分析，大数据的大量和按需模式的结合使得预计算变得不可行或不具吸引力。

但是即时抽样仍然廉价且有用。

此外，自上世纪90年代以来，行业的结构和其接口发生了变化。 从底层开始，查询引擎标准今天通常通过开源开发出现和演变，而获胜的项目（例如Hadoop和Spark）变得紧密。

[3]这尤其具有讽刺意味，因为一些供应商提供的抽样支持是有偏差的（通过对块进行抽样而不是对元组进行抽样）。

足够垄断他们的API可以决定客户端设计。 同时，从上到下，云中托管的数据可视化产品通常是垂直集成的：前端体验是主要关注点，并且由（通常是特定目的的）后端实现驱动，而不关注标准化。 在这两种情况下，可以通过从引擎到应用程序的堆栈传递在线聚合等独特功能。

在这种背景下，我们介绍了该领域中最广泛阅读的最新论文之一，即BlinkDB。该系统利用了Olken所称的“物化样本视图”：在基本表上预先计算的样本，存储以加快近似查询的速度。 与早期的OLAP论文一样，BlinkDB提出只需预先计算少数GROUP BY子句即可在（稳定的）工作负载上获得良好的性能。 早期AQUA项目的作者也提出了类似的论点，即通过预先计算的概要（“草图”）而不是物化样本视图作为其基本近似机制。 BlinkDB论文还提出了在其视图中进行分层的理由，以捕捉小组，这让人想起在线聚合论文中的索引跨越。 BlinkDB在业界引起了兴趣，Spark团队最近提出了在预先计算的样本上进行即时抽样的增强方法-这是一种尽可能高效地实现在线聚合的合理混合技术。 最近的商业BI工具如ZoomData 似乎也使用在线聚合（他们称之为“查询优化”）

在所有这些关于在线聚合的讨论中，值得对当前市场现实进行一次快照。 自从广泛引入以来的25年里，OLAP风格的预计算已经支撑起了现在价值数十亿美元的商业智能行业。 相比之下，用户界面上的近似几乎不存在。 因此，对于那些根据收入生成进行计分的人来说，预计算这个简单的解决方案是目前的赢家。 何时以及是否将近似成为实践中的常用技术仍然是一个悬而未决的问题。 从技术层面上讲，采样的基本好处似乎是不可避免的，并且围绕数据增长和探索性分析的技术趋势使其在大数据市场上具有吸引力。 但是今天这仍然是一项超前的技术。

一个最终的算法注释：通过草图近似查询实际上在今天的工程师和数据科学家中被广泛使用作为分析的基础。 除了这里涵盖的系统工作之外，精通数据库的学生还应该熟悉像CountMin草图、HyperLogLog草图、Bloom过滤器等技术。该领域的全面调查可以在[44]中找到；各种草图的实现可以在许多在线语言中找到，包括在第11章提到的MADlib库中作为用户定义的函数。

第9章：语言

由乔·海勒斯坦引入

精选读物：

Joachim W. Schmidt. 一些用于关系型数据的高级语言构造。*ACM Transactions on Database Systems*, 2(3), 1977, 247-261.

Arvind Arasu, Shivnath Babu, Jennifer Widom. CQL连续查询语言：语义基础和查询执行。 *The VLDB Journal*, 15(2), 2006, 121-142.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak. 在Bloom中的一致性分析：一种CALM和集中的方法。 *CIDR*, 2011.

从阅读数据库论文，你可能会期望典型的数据库用户是数据分析师、业务决策者或IT人员。 实际上，大多数数据库用户是软件工程师，他们构建数据库支持的应用程序，这些应用程序在堆栈的更高层次上使用。 尽管SQL最初是为非技术用户设计的，但人们很少直接与数据库通过SQL这样的语言进行交互，除非他们正在编写一个数据库支持的应用程序。

所以，如果数据库系统主要只是软件开发的API，它们为程序员提供了什么？
像大多数优秀的软件一样，数据库系统提供了强大的抽象。

有两个突出的特点：

1. 事务模型为程序员提供了一个抽象的单进程、顺序执行的机器，永远不会在任务中间失败。 这保护了程序员免受巨大的复杂性的困扰，即现代计算机的内在并行性。 今天的单个计算机机架上有数千个核心在数十台机器上并行运行，可以独立故障。 然而，应用程序员仍然可以毫不费力地编写顺序代码，就像是1965年，他们正在将一叠打孔卡加载到一台大型机中逐个执行一样。

2. 像SQL这样的声明性查询语言为程序员提供了操作数据集的抽象。 众所周知，声明性语言使程序员不必考虑如何访问数据项，而是让他们专注于返回哪些数据项。 这种数据独立性也保护了应用程序员从底层数据库的组织变化中保护数据库管理员免于参与应用程序的设计和维护。

这些抽象在过去的时间里有多有用？ 它们在今天有多有用？

1. 作为一种编程结构，可串行化的事务具有很大的影响力。程序员可以很容易地用BEGIN和COMMIT/ROLLBACK来包围他们的代码。不幸的是，正如我们在第6章中讨论的那样，事务是昂贵的，并且经常受到妥协。 “放松”的事务语义打破了用户的串行抽象，并将应用程序逻辑暴露给竞争和/或不可预测的异常的可能性。 如果应用程序开发人员想要解决这个问题，他们必须处理并发性、故障和分布式状态的复杂性。 对于缺乏事务的常见反应是追求“最终一致性”[154]，正如我们在弱隔离性部分所讨论的那样。 但正如我们在第6章中所讨论的，这仍然将所有正确性的负担转移到应用程序开发人员身上。 在我看来，这种情况代表了现代软件开发中的一场重大危机。

2. 声明性查询语言也取得了成功 - 显然比之前的导航语言要好，这些导航语言导致了需要每次重新组织数据库时都需要重写的意大利面代码。 不幸的是，查询语言与程序员通常使用的命令式语言非常不同。查询语言消耗和生成简单的无序“集合类型”（集合、关系、流）；编程语言通常对复杂的结构化数据类型（树、哈希表等）进行有序执行指令。数据库应用程序的程序员被迫弥合这种所谓的“阻抗不匹配”程序和数据库查询之间的差

> 这对于数据库程序员来说一直是一个麻烦，自关系数据库诞生以来就存在。 这对于关系数据库的程序员来说一直是一个麻烦，自关系数据库诞生以来就存在。

数据库语言嵌入：Pascal/R

本节中的第一篇论文展示了解决第二个问题的一个经典示例：帮助命令式程序员解决阻抗不匹配问题。 论文首先定义了一些操作，这些操作我们现在（40多年后！）可能会认识到是熟悉的集合类型：Python中的“字典”类型，Java或Ruby中的“映射”类型等。然后，论文耐心地介绍了在应用程序中似乎反复出现的各种语言结构的可能性和陷阱，这些结构在几十年间一直存在。 一个关键主题是希望区分枚举（用于生成输出）和量化（用于检查属性）-如果你明确指出后者，往往可以进行优化。 最后，论文提出了一种嵌入到Pascal中的声明性、类似SQL的子语言，用于关系类型。结果相对自然，与今天一些更好的接口相似。

尽管这种方法现在似乎很自然，但这个主题花了几十年才引起广泛关注。 在这个过程中，像ODBC和JDBC这样的数据库“连接”API成为了C/C++和Java的救命稻草-它们允许用户将查询推送到DBMS并遍历结果，但类型系统仍然是分离的，从SQL类型到宿主语言类型的桥接是令人不愉快的。 也许像Pascal/R这样的思想的最佳现代演变是微软的LINQ库，它提供了语言嵌入的集合类型和函数，以便应用程序开发人员可以在各种后端数据库和其他集合（XML文档、电子表格等）上编写类似查询的代码。我们在第5章的DryadLINQ论文中包含了一小部分LINQ语法的味道。

在2000年代，像社交媒体、在线论坛、互动聊天、照片共享和产品目录等网络应用程序被实现和重复实现在关系数据库后端。现代脚本语言用于Web编程的语言比Pascal更方便，通常包含良好的集合类型。 在这种环境中，应用程序开发人员最终在他们的代码中看到了被认可的模式，并将它们编码成现在被称为对象关系映射（ORM）的东西。 Ruby on Rails是最有影响力的ORM之一，尽管现在有很多其他的ORM。 每种流行的应用程序编程语言至少有一个ORM，并且在功能和理念上有所不同。 嵌套的读者可以参考维基百科的“对象关系映射软件列表”维基页面。

ORM对于Web程序员来说有一些方便的功能。首先，它们提供了与集合类似的语言原生基元，就像Pascal/R一样。其次，它们可以使内存中的语言对象的更新在数据库支持的状态中透明地反映出来。 它们通常为熟悉的数据库设计概念（如实体、关系、键和外键）提供一些语言原生的语法。 最后，一些ORM（包括Rails）提供了很好的工具，用于跟踪数据库模式随时间演变以反映应用程序代码的变化（在Rails术语中称为“迁移”）。

这是一个数据库研究社区和工业界应该更加关注的领域：这些是我们的用户！ORM和数据库之间存在一些令人惊讶和令人不安的脱节[19]。 例如，Rails的作者是一个名叫David Heinemeier Hansson（“DHH”）的有趣人物，他相信“有主见的软件”（当然反映了他的观点）。 他曾经说过以下的话：

> 我不希望我的数据库聪明！……我认为存储过程和约束是破坏一致性的恶劣行为。
> 不，数据库先生，你不能拥有我的业务逻辑。 你的过程性野心将毫无收获，你必须从我的冷漠的面向对象的手中夺走那个逻辑…… 我只想要一个聪明的层次：我的领域模型。

这种不愿相信DBMS的态度导致了Rails应用程序中的许多问题。 针对ORM编写的应用程序通常非常慢——ORM本身并没有对查询生成进行优化。 因此，Rails程序员经常需要

为了学习以不同的方式编程，以鼓励Rails生成高效的SQL，就像在Pascal/R论文中讨论的那样，他们需要学会避免循环和逐表迭代。 Rails应用程序的典型演变是先以简单的方式编写，观察性能较慢，研究生成的SQL日志，然后重新编写应用程序以说服ORM生成“更好”的SQL。Cheung和同事最近的研究探索了程序合成技术可以自动生成这些优化的想法[38]；这是一个有趣的方向，时间将会告诉我们它能自动化多少复杂性。 数据库和应用程序之间的分离也可能对正确性产生负面影响。 例如，Bailis最近展示了[19]许多现有的开源Rails应用程序由于在应用程序内部（而不是数据库内部）未正确执行强制执行而容易出现完整性违规。

尽管存在一些盲点，ORM在数据库支持的应用程序的可编程性方面通常是一个重要的实际进步，并且验证了追溯到Pascal/R的一些想法。一些好的想法需要时间才能被接受。

流查询：CQL

我们关于CQL的第二篇论文是一种不同的语言工作 - 这是一篇查询语言设计论文。 它介绍了一种新的声明性查询语言的设计，用于流数据模型。 这篇论文有几个有趣之处。 首先，它是一个清晰、易读且相对现代的查询语言设计示例。

每隔几年就会出现一群人，提出一个新的数据模型和查询语言：例如对象和OQL、XML和XQuery，或者RDF和SPARQL。这些练习通常以一种断言开始，声称某个数据模型“改变了一切”，从而呈现出一种看起来很熟悉但又奇怪不同于SQL的新查询语言。CQL是一种令人耳目一新的语言设计示例，因为它恰恰相反：它强调了通过正确的视角查看流数据实际上几乎没有改变。 CQL对SQL进行了足够的演变，以便区分“静止”表和“移动”流之间的关键区别。 这使我们清楚地了解到，在谈论流时，从语义上讲真正不同的是什么；许多其他当前的流语言比CQL更加临时和混乱。

除了这篇论文是一个很好的范例之外深思熟虑的查询语言设计，它也代表了数据库文献中受到很多关注的研究领域，并且在实践中仍然具有吸引力。 从2000年代初的第一代流数据研究系统[3, 120, 118, 36]来看，它们在开源方面或者在各种初创公司中都没有得到广泛应用。 然而，近年来，流查询的话题在工业界再次引起了兴趣，像Spark-Streaming、Storm和Heron这样的开源系统得到了应用，而像Google这样的公司也强调了连续数据流作为现代服务的重要性[8]。

我们可能会看到流查询系统在金融服务领域占据比目前更大的市场份额。

CQL有趣的另一个原因是流数据在数据库和“事件”之间的一种中间地带。 数据库存储和检索集合类型；事件系统传输和处理离散事件。

一旦你将你的事件视为数据，那么事件编程和流编程看起来非常相似。 考虑到事件编程是某些领域（例如用户界面的Javascript，分布式系统的Erlang）中广泛使用的编程模型，事件编程语言（如Javascript）和数据流系统之间应该存在相对较小的阻抗不匹配。 这种方法的一个有趣的例子是Rx（反应式扩展）语言，它是LINQ的流式扩展，使编程事件流感觉像编写功能查询计划；或者正如其作者Erik Meijer所说，“你的鼠标是一个数据库”[114]。

无事务编程正确的应用程序：Bloom

Bloom的第三篇论文与上述几点有关；它在应用程序级别具有状态的关系模型，并且有一个与CQL流相关的网络通道概念。 但主要目标是帮助程序员在本章开头介绍的第一个抽象的丧失时进行管理；我将其描述为一个重大危机。 对于现代开发人员来说，一个重要的问题是：在不使用事务或其他昂贵的方案来控制操作顺序的情况下，你能否找到一个正确的分布式实现方案？

布鲁姆对这个问题的答案是给程序员一个“无序”的编程语言：一个阻止他们意外使用排序的语言。

布鲁姆的默认数据结构是关系；它的基本编程构造是可以以任意顺序运行的逻辑规则。 简而言之，它是一种类似于关系查询语言的通用编程语言。 出于相同的原因，SQL查询可以进行优化和并行化而不改变输出，简单的布鲁姆程序具有明确定义的（一致的）结果，与执行顺序无关。 对于这个直觉的例外是布鲁姆代码中的“非单调”行，它们测试随着时间的推移可以在真和假之间振荡的属性（例如“NOT EXISTS x”或“HAVING COUNT() = x”）。这些规则对执行和消息顺序敏感，并且需要通过协调机制进行“保护”。

CALM定理正式化了这个概念，明确回答了上面的问题：只有当程序的规范是单调的时候，你才能找到一个一致的、分布式的、无协调的实现[84, 14]。 Bloom论文还说明了编译器如何在实践中使用CALM来确定Bloom程序中协调的需求。 CALM分析还可以应用于数据流语言，例如在像Storm这样的系统中，借助程序员的注解[12]。 关于这个领域的理论结果的调查在[13]中给出。

关于避免协调的相关语言工作已经有了一系列的研究：一些论文提出使用关联、交换、幂等操作[83, 142, 42]；这些操作本质上是单调的。 另一组工作研究了替代的正确性标准，例如只确保数据库上的特定不变量。

状态[20]或使用替代程序分析来提供可串行化的结果，而不实现传统的读写并发[137]。这个领域还很新颖；论文有不同的模型（例如，一些有事务边界，一些没有），并且通常对“一致性”或“协调”没有一致的定义。

（CALM将一致性定义为全局确定性结果，协调作为无论数据分区或复制都需要的消息[14]。）在这里更清楚和更多的想法非常重要 - 如果程序员不能使用事务，则他们需要在应用程序开发层面得到帮助。

Bloom也是数据库研究中一个反复出现的主题的例子：通用声明性语言（也称为“逻辑编程”）。 Datalog是标准的例子，在数据库研究中有着悠久而有争议的历史。 Datalog是20世纪80年代数据库理论家们最喜欢的话题之一，但在当时的系统研究者中引起了激烈的反对，认为它在实践中无关紧要[152]。 最近，它在数据库和其他应用领域的（年轻）研究人员中引起了一些关注[74] - 例如，Nicira的软件定义网络堆栈（被VMWare以10亿美元收购）使用了Datalog语言来进行网络转发状态[97]。 在访问数据库状态时，可以使用声明性子语言，也可以使用像Bloom这样用于指定应用逻辑的声明性编程的非常激进的用法之间存在一个谱。 时间将告诉我们在各种情境中，包括基础设施、应用程序、Web客户端和移动设备，这种声明性-命令性边界对程序员的影响如何变化。

第十章：网络数据

由Peter Bailis引入

精选读物：

Sergey Brin和Larry Page。一个大规模超文本网络搜索引擎的解剖。 *WWW*，1998年。

Eric A. Brewer。结合系统和数据库：搜索引擎回顾。数据库系统读物，第四版，2005年。

Michael J. Cafarella，Alon Halevy，Daisy Zhe Wang，Eugene Wu，Yang Zhang。WebTables：探索网络表格的力量。 *VLDB*，2008年。

自从这个系列的上一版以来，万维网毫无疑问地解决了关于其长寿和全球影响的任何疑问。 包括Google和Facebook在内的几个拥有数十亿用户的服务已经成为现代生活中的核心，而与互联网和网络相关的技术已经渗透到商业和个人互动中。 毫无疑问，网络至少在可预见的未来是不可或缺的。

Web数据系统带来了一系列新的挑战，包括高规模、数据异构性以及复杂且不断演化的用户交互模式。 经典的关系型数据库系统设计并没有考虑到Web的工作负载，并不是在这种情况下的首选技术。 相反，Web数据管理需要一系列技术的融合，涵盖信息检索、数据库内部、数据集成和分布式系统。 在本节中，我们包括了三篇论文，重点介绍了Web数据管理中固有问题的技术解决方案。

我们的前两篇论文描述了搜索引擎和索引技术的内部机制。 我们的第一篇论文来自Google的联合创始人Larry Page和Sergey Brin，描述了Google早期原型的内部机制。这篇论文从历史和技术的角度都很有趣。最初的Web索引，如Yahoo！，由人工策划的“目录”组成。 尽管目录策划证明是有用的，但目录很难扩展，并且需要大量的人力来维护。 因此，一些搜索引擎，包括Google和Inktomi（由第二篇论文的作者Eric Brewer共同创建），寻求自动化的方法。 这些引擎的设计在概念上很简单：一组爬虫下载Web数据的副本，并构建（和维护）只读的索引。用于计算相关性评分的索引。 反过来，查询由一个前端网络服务处理，该服务从索引中读取并呈现一组按评分函数排序的结果。

这些引擎的实现和实现是复杂的。 例如，评分算法是高度调整的，它们的实现甚至在今天的搜索引擎中也被视为商业机密：网络作者有很大的动机来操纵评分函数以获得自己的利益。 在Google论文中描述的PageRank算法（并在[126]中详细介绍）是评分函数的一个著名例子，它根据超链接图度量每个页面的“影响力”。 这两篇论文描述了在实践中如何使用大多数未指定的属性进行评分，包括“锚文本”（提供链接来源的上下文）和其他形式的元数据。 这些技术的算法基础，如关键词索引日期，可以追溯到1950年代[107]，而其他技术，如TFxIDF排名和倒排索引，可以追溯到1960年代[139]。 在构建互联网搜索引擎方面的许多关键系统创新涉及它们的扩展和处理脏、异构的数据源。

尽管这些论文的高层细节有助于理解现代搜索引擎的运作方式，但这些论文也因其对构建生产级Web搜索引擎过程的评论而引人入胜。 每篇论文都传达了一个核心信息，即Web服务必须考虑到多样性；Google的作者描述了在Web环境中典型信息检索技术所做的假设可能不再成立的情况（例如，“比尔·克林顿糟糕”网页）。 Web资源以不同的速度变化，因此需要优先抓取以保持索引的新鲜性。布鲁尔还强调了容错性和可用性的重要性。

他在构建Inktomi时的经验（也导致了概念的发展，包括收获和产量[31]以及CAP定理；请参阅第7章）也反映了操作的重要性。 布鲁尔概述了使用商品数据库引擎（例如，Informix比Inktomi的定制解决方案慢10倍）构建搜索引擎的困难。 然而，他指出，在这种情况下，数据库系统设计的原则，包括“自顶向下”设计、数据独立性和声明式查询引擎，是有价值的，只要适当地进行调整。

如今，网络搜索引擎被认为是成熟的技术。 然而，竞争服务不断通过添加额外的功能来改进搜索体验。 如今的搜索引擎远不止是用于文本数据网页的信息检索引擎；这两篇论文的内容只是像谷歌或百度这样的服务内部的一个小部分。这些服务提供了一系列的功能，包括定向广告、图片搜索、导航、购物和移动搜索。 在检索、实体解析和索引技术之间无疑存在着相互渗透，但每个领域都需要特定领域的适应。

作为大规模网络数据所带来的新型搜索的一个例子，我们包括了由谷歌的Alon Halevy领导的Web Tables项目的一篇论文。WebTables允许用户查询和理解存储在HTML表中的数据之间的关系。 由于缺乏固定的模式，HTML表的结构本质上是多样的。然而，在Web规模上聚合足够多的HTML表，并进行一些轻量级的自动化数据集成，可以实现一些有趣的查询（例如，可以将流感爆发地点的表与包含城市人口数据的表合并）。 挖掘这些表的模式，确定它们的结构和真实性（例如，实际上只有论文语料库中的1%的表是关系表），以及有效地推断它们之间的关系是困难的。 我们包含的论文描述了构建属性相关统计数据库（AcsDB）以回答关于表元数据的查询的技术，从而实现了包括模式自动完成在内的新颖功能。WebTables

bles项目以各种形式继续进行，包括Google表搜索和与Google核心搜索技术的集成；有关该项目的更新可以在[24]中找到。 在几个非传统领域中，包括移动、上下文和基于音频的搜索，能够产生结构化的搜索结果是可取的。

特别是WebTables论文突出了在规模化Web数据上工作的能力。 在2009年的一篇文章中，Halevy和同事描述了“数据的非理性有效性”，有效地论证了足够数量的数据可以捕捉足够的潜在结构，从而使建模变得更简单：相对简单的数据挖掘技术通常能够击败更复杂的数学统计模型[79]。 这个论点强调了通过大量数据和计算来解锁隐藏结构的潜力，无论是挖掘模式相关性还是进行语言之间的机器翻译。 有了足够大的干草堆，针变得很大。 即使只检查Web语料库中1%的表格，VLDB 2009论文也研究了1.54亿个不同的关系，这个语料库“比之前考虑的最大语料库大五个数量级”。

由于廉价的商品存储和云计算资源，执行大规模数据集和系统架构分析的障碍正在降低。 然而，要复制用户（例如，垃圾邮件发送者）和算法（例如，搜索排名算法）之间的反馈循环是困难的。互联网公司在为此反馈循环考虑的系统设计方面处于独特的位置。 随着数据库技术为更多的交互式领域提供动力，我们相信这种范式将变得更加重要。 也就是说，数据库市场和感兴趣的数据库工作负载可能会从类似的分析中受益。 例如，有趣的是对托管数据库平台（如Amazon Redshift和Microsoft SQL Azure）执行类似的分析，以实现包括索引自动调整、自适应查询优化、从非结构化数据中进行模式发现、查询自动完成和可视化推荐等各种功能。

第11章：对一个不断变化的目标的偏见观点：复杂分析

作者：迈克尔·斯通布雷克

在过去的5-10年中，出现了比典型的商业智能（BI）用例更复杂的新分析工作负载。 例如，互联网广告商可能想知道“过去四天内购买苹果电脑的女性与购买福特皮卡的女性在统计上有何不同？”下一个问题可能是：“在我们所有的广告中，基于点击概率，哪一个对女性福特买家来说最有利可图？”这些是当今数据科学家提出的问题，代表了与传统SQL分析由商业智能专家运行的非常不同的用例。 人们普遍认为，在未来十年或二十年内，数据科学将完全取代商业智能，因为它代表了一种更复杂的挖掘数据仓库以获得新见解的方法。 因此，本文档侧重于数据科学家的需求。

我将从描述数据科学家的工作描述开始这一部分。 在清洗和整理数据之后，他目前花费了大部分时间，这在数据整合部分有所讨论，他通常执行以下迭代：

```
直到（累了） {
    数据管理操作;
    分析操作;
}
```

换句话说，他有一个迭代的发现过程，他首先隔离出一个感兴趣的数据集，然后对其执行一些分析操作。这通常暗示着要么尝试在同一数据集上进行相同分析的不同数据集，要么在同一数据集上尝试不同的分析。总的来说，数据科学与商业智能的区别在于分析是预测建模、机器学习、回归等，而不是SQL分析。

一般来说，构成分析的计算有一个流水线。 例如，Tamr有一个模块，可以在规模上对一组记录（例如N条记录）执行实体合并（去重）操作。 为了避免暴力算法的N ** 2复杂度，Tamr识别出一组“特征”，将它们分成不太可能同时出现的范围，并基于这些范围为每个记录计算（可能是多个）“箱子”。 范围，重新排列数据并行分区按照bin号，去重每个bin，合并结果，最后从不同的重复集群构建复合记录。 这个流水线部分面向SQL（分区），部分面向数组分析。 Tamr似乎是典型的数据科学工作负载，有六个步骤的流水线。

一些分析流水线是“一次性”的，在一批记录上运行一次。 然而，大多数生产应用程序是增量的。 例如，Tamr在初始输入记录的批处理上运行，然后定期处理新的“增量”作为新的或更改的输入记录到达。 增量操作有两种方法。 如果增量以“小批量”的方式在（例如）一天的周期间隔内处理，可以将下一个增量添加到先前处理的批次中，并在每次输入更改时重新运行整个流水线。 这种策略会浪费大量的计算资源。 相反，我们将重点放在初始批处理操作之后必须运行增量算法的情况上。 这种增量算法需要在每个中间状态的分析中将其保存到持久存储中。 尽管Tamr流水线的长度约为6，但每个步骤都必须保存到持久存储以支持增量操作。 由于保存状态是一个数据管理操作，这使得分析流水线的长度为1。

最终的“实时”解决方案是通过流式平台连续运行增量分析服务，如新DBMS技术部分所讨论的。 根据新记录的到达速率，可以选择任一解决方案。

大多数复杂的分析都是面向数组的，即它们是在数组上定义的线性代数操作的集合。 一些分析是面向图的，例如社交网络分析。 很明显，数组可以在基于表的系统上模拟，而图可以在表系统或数组系统上模拟。 因此，在本文档的后面，我们将讨论为这个用例需要多少不同的架构。

一些问题涉及到稠密数组，即几乎所有单元格都有一个值。 例如，一个包含所有证券的随时间变化的收盘股价数组将是稠密的，因为每只股票每个交易日都有一个收盘价。另一方面，一些问题是稀疏的。

例如，将社交网络用例表示为矩阵将对每对相关人员有一个单元格值。显然，这个矩阵将非常稀疏。例如，将社交网络用例表示为矩阵将对每对相关人员有一个单元格值。显然，这个矩阵将非常稀疏。

稀疏数组上的分析与密集数组上的分析非常不同。

在本节中，我们将讨论这样的工作负载规模。如果想要在“小数据”上执行这样的流水线，则任何解决方案都可以正常工作。

数据科学平台的目标是支持这个迭代发现过程。 我们从一个令人沮丧的事实开始。 大多数数据科学平台都是基于文件的，与数据库管理系统无关。大多数分析代码都在R、MatLab、SPSS、SAS上运行，并且操作文件数据。 此外，许多Spark用户正在从文件中读取数据。 这种情况的一个例子是劳伦斯伯克利实验室的NERSC高性能计算（HPC）系统。该机器基本上专门用于复杂分析；然而，由于配置限制，我们无法使Vertica DBMS运行。 此外，大多数“大科学”项目从底层开始构建整个软件堆栈。这种情况可能会继续下去，DBMS在这个市场上不会成为一名参与者。 然而，有一些希望的迹象，例如遗传数据开始在DBMS上部署，例如1000个基因组计划[144]基于SciDB。在我看来，文件系统技术存在一些重大缺点。 首先，元数据（校准、时间等）通常无法捕获或以文件名编码，因此无法进行搜索。 其次，复杂的数据处理以完成数据科学工作负载的数据管理部分是不可用的，必须编写（以某种方式）。 第三，文件数据很难在同事之间

共享。 我知道有几个项目将其数据与解析程序一起导出。 接收者可能无法重新编译此访问程序，或者它会生成错误。 在本讨论的其余部分，我将假设数据科学家随着时间的推移希望使用DBMS技术。 因此，不会再讨论基于文件的解决方案。

在这个背景下，我们在表1中展示了数据科学平台的分类。 要执行数据管理部分，根据我们的假设，需要一个数据库管理系统（DBMS）。 这个DBMS可以是两种之一

类型。 首先，它可以是面向记录的，如关系型行存储或NoSQL引擎，或者是面向列的，如大多数数据仓库系统。 在这些情况下，DBMS的数据结构不是针对分析需求的，而分析需求基本上都是面向数组的，因此更自然的选择是数组DBMS。后一种情况的优势在于不需要从记录或列结构进行转换来执行分析。 因此，数组结构在性能上具有固有的优势。 此外，面向数组的存储结构在本质上是多维的，而表结构通常是一维的。同样，这可能会导致更高的性能。

第二个维度涉及分析和数据库管理系统之间的耦合。一方面，它们可以是独立的，可以运行查询，将结果复制到不同的地址空间，在那里运行分析。在分析管道的末尾（通常长度为1），结果可以保存回持久存储。 这将导致数据库管理系统和分析之间的大量数据变动。 另一方面，可以将分析作为用户定义的函数在与数据库管理系统相同的地址空间中运行。显然，紧密耦合的选择将减少数据变动，并应产生更好的性能。

在这种情况下，有四个单元格，如表1所示。在左下角，Map-Reduce曾经是典范；最近，Spark已经超过Map-Reduce成为最受关注的平台。 Spark中没有持久性机制，它依赖于Red-Shift或H-Base等来实现此目的。 因此，在Spark中，用户在某个数据库管理系统中运行查询以生成数据集，然后将其加载到Spark中进行分析。 Spark支持的数据库管理系统都是记录或列导向的，因此需要将其转换为数组表示形式进行分析。

在右下角的一个显著例子是MADLIB [85]，它是由RDBMS Greenplum支持的用户定义函数库。其他供应商最近也开始支持其他替代方案；例如，Vertica支持R中的用户定义函数。在右上角是具有内置分析功能的数组系统，例如SciDB [155]，TileDB [56]或Rasdaman [26]。

在本文的其余部分，我们将讨论性能影响。 首先，人们预计性能会随着在表1中从左下角移动到右上角而改善。 其次，大多数复杂的分析都可以简化为

	松耦合	紧耦合
数组表示		SciDB, TileDB, Rasdaman
表表示	Spark + HBase	MADLib, Vertica + R

表1：数据科学平台的分类

一个小的“内循环”操作集合，例如矩阵乘法、奇异值分解和QR分解。 所有这些都是计算密集型的，通常是浮点数代码。 大多数人都认为，硬件特定的流水线技术可以在这些类型的代码上提高近一个数量级的性能。 因此，调用硬件优化的Intel MKL库的库，如BLAS、LAPACK和ScaLAPACK，将比不使用硬件优化的代码快得多。 当然，在密集数组计算中，硬件优化将产生很大的差异，因为大部分工作都是浮点计算。 在稀疏数组中，硬件优化的意义将较小，因为索引问题可能主导计算时间。

第三，提供近似答案的代码比生成精确答案的代码要快得多。 如果你能处理近似答案，那么你将节省大量时间。

第四，高性能计算（HPC）硬件通常配置为支持大批量作业。 因此，它们通常被构造为计算服务器通过网络连接到存储服务器，程序必须在计算服务器缓存中预分配磁盘空间来满足其存储需求。 这显然与DBMS相悖，DBMS期望作为一个持续运行的服务。 因此，请注意在HPC环境中可能会遇到DBMS系统的问题。 一个有趣的探索领域是HPC机器是否能够同时处理交互式和批量工作负载而不降低性能。

第五，可扩展的数据科学代码通常在计算机网络中的多个节点上运行，并且通常受网络限制[55]。 在这种情况下，你必须仔细注意网络成本，TCP-IP可能不是一个好的选择。 一般来说，MPI是一个更高性能的替代方案。

第六，我们测试过的大多数分析代码都无法扩展到大型数据集大小，要么因为它们耗尽了主内存，要么因为它们生成的临时文件太大。 确保你测试任何你考虑在生产中运行的平台！你期望在生产中使用的数据集大小上运行的平台！

第七，你的分析流程的结构是关键的。如果你的流程只有一个步骤，那么紧密耦合几乎肯定是一个好主意。 另一方面，如果流程有10个步骤，松散耦合也能表现得几乎一样好。 在增量操作中，预计流程只有一个步骤。

总的来说，我们所知道的所有解决方案都存在可扩展性和性能问题。 此外，上述大部分示例都是快速发展的目标，因此性能和可扩展性无疑会得到改善。 总之，很有意思的是看看表1中哪些单元有腿，哪些没有。 商业市场将是最终的仲裁者！在我看来，复杂分析目前正处于“野西”阶段，我们希望红皮书的下一版能够确定一系列核心的开创

性论文。 与此同时，还有大量的研究需要进行。 具体而言，我们鼓励在这个领域进行更多的基准测试，以便发现现有平台的缺陷，并推动进一步的研究和开发，特别是涉及数据管理任务和分析任务的端到端任务的基准测试。 这个领域发展迅速，因此基准测试结果可能是暂时的。 这可能是件好事：我们正处于各个项目互相学习的阶段。

目前对于核心分析任务（如凸优化）的自定义并行算法引起了很大兴趣；其中一部分来自数据库界。 有趣的是，这些算法是否能够被纳入分析型数据库管理系统中，因为它们通常不遵循传统的数据流执行方式。 一个典型的例子是Hogwild！[133]，它通过允许无锁并行在共享内存中实现非常快的性能。 Google Downpour [49]和Microsoft的Project Adam [39]都将这个基本思想应用到分布式深度学习的上下文中。

另一个值得探索的领域是内存外算法。 例如，Spark要求你的数据结构要适应组合数量的

主内存，即你集群中的机器上的内存。
这样的解决方案会很脆弱，并且几乎肯定会有可扩展性问题。

此外，一个有趣的话题是理想的图分析方法。可以构建专用的图分析工具，如GraphX [64]或GraphLab [105]，并将它们连接到某种DBMS。或者，可以使用数组分析（如D4M [95]）或表格分析（如[90]中建议的方法）来模拟这些代码。 再次，愿解决方案空间繁荣，商业市场成为仲裁者！

最后，许多分析代码使用MPI进行通信，而DBMS通常使用TCP-IP。此外，诸如ScaLAPACK之类的并行密集分析包将数据组织成计算集群中节点之间的块循环组织[40]。 我不知道有哪个DBMS支持块循环分区。 消除分析包和DBMS之间的这种阻抗不匹配是一个有趣的研究领域，这是由英特尔赞助的大数据ISTC [151]所关注的目标之一。

评论：乔・海勒斯坦

*2015*年*12*月*6*日

从商业角度和研究机会来看，我对这个领域有一个与迈克截然不同的看法。 基本上，我建议在这个领域采取“大帐篷”方法。
数据库专家在这方面有很多贡献，但如果我们能与其他人合作得好，我们会做得更好。

让我们来看看这个行业。 首先，我们正在讨论的这种高级分析不会像迈克所建议的那样取代BI。 BI行业健康并且在增长。 更重要的是，正如著名统计学家约翰・图基在他关于探索性数据分析的基础工作中指出的那样，一个图表通常比一个复杂的统计模型更有价值。

尊重图表！

话虽如此，高级分析和数据科学市场确实在增长，并且准备进行变革。 但与商业智能市场不同，数据库技术在这个领域目前并不起重要作用。 这个领域的现任领导者是SAS，这是一家每年创造数十亿美元收入的公司，明显不是数据库公司。

当风险投资公司看这个领域的公司时，他们在寻找“下一个SAS”。SAS用户不是数据库用户。 而且像R这样的开源替代品的用户也不是数据库用户。 如果你像迈克一样假设“数据科学家将想要使用数据库管理系统技术” - 尤其是一种整体的

[4]Tukey, John. 探索性数据分析. Pearson, 1977.

“分析型数据库管理系统” - 你正在逆流而上

对于高级分析市场来说，更自然的方法是问自己：SAS面临着什么严重威胁？
谁能够大幅削减企业目前在那里花费的资金？ 以下是一些起点考虑：

1. 开源统计编程：包括R和Python数据科学生态系统（NumPy，SciKit-Learn，iPython Notebook）。这些解决方案目前不能很好地扩展，但正在积极努力解决这些限制。. 这个生态系统可能比SAS发展得更快。2. 与大数据平台的紧密耦合。. 当数据足够大时，性能要求可能会将用户“拖累”到一个新的平台，即已经托管其组织中的大数据的平台。

   因此，对于像Spark/MLLib、PivotalR/MADlib和Ver-tica dplyr这样的平台，对“DataFrame”接口的兴趣很大。 请注意，高级分析社区对开源非常偏爱。 云平台在这里也是一个有趣的平台，而且不是SAS具有优势的平台。

3. 分析服务。 我指的是以分析方法为核心的交互式在线服务：推荐系统、实时欺诈检测、预测设备维护等。这个领域对响应时间、请求扩展性、容错性和持续演进有着激烈的系统要求，而SAS等产品并没有解决这些问题。 如今，这些服务主要是使用自定义代码构建的。 这种方式在不同行业之间无法扩展 - 大多数公司无法招募到能够在这个层面上工作的开发人员。 因此，在大多数用例中，将这项技术商品化似乎存在机会。 但是，对于这个市场来说，现在还处于早期阶段 - 尚不清楚分析服务平台是否能够简化到足够适用于商品化部署。 如果技术得到发展，那么基于云的服务在这个领域也可能有重大机会进行颠覆。

在研究前沿，我认为思考超越数据库的范畴，并积极合作是至关重要的。 对我来说，这几乎是不言而喻的。几乎每个计算机科学的子领域都在以某种方式进行大数据分析，来自各个领域的聪明人们正在迅速学习有关数据和规模的教训。 我们可以和这些人一起玩得开心，或者我们可以无视他们而自食其果。

那么，在这个领域中，数据库研究可以产生重大影响的地方在哪里呢？ 我认为有一些看起来不错的可能性包括：

1. **可扩展性的新方法。我们已经成功地**

证明了并行数据流（比如MADlib、ML-lib或Teradata的Ordonez的工作[5]）可以在不对系统架构造成破坏的情况下实现可扩展的分析。这是有用的信息。 展望未来，我们能否做出比分区数据上的并行数据流更快、更可扩展的有用工作？这是必要的吗？ Hogwild! 在这里引起了很大的兴奋；请注意，这是跨数据库和机器学习社区的工作。

2. 用于分析服务的分布式基础设施.. 正如我上面提到的，分析服务是一个创新的机会。 在这方面，系统基础设施问题非常开放。 分析服务架构的主要组成部分是什么？ 它们是如何组合在一起的？ 在组件之间需要什么样的数据一致性？

   所谓的参数服务器是一个感兴趣的话题。现在，但只解决了难题的一部分。[6]关于在线服务、模型演化和部署已经有了一些初步的工作。[7]我希望会有更多的工作。

3. 分析生命周期和元数据管理.. 这是一个我同意迈克观点的领域。分析通常是一个人力密集型的工作，除了核心统计模型外，还涉及数据探索和转换。 数据库社区对这个领域有非常相关的观点，包括工作流管理、数据血统和物化视图维护。 **Vi**s-Tr**ail**s是这个领域的一个研究实例，正在实践中使用。[8] 这也是工业界迫切需要的领域，特别是考虑到现实世界中各种各样的分析工具和系统。

[5]例如，Ordonez, C. 使用SQL将K-means聚类与关系型数据库管理系统集成。TKDE 18(2) 2006. 还有Ordonez, C. 使用UDFs进行统计模型计算。TKDE 22(12), 2010.

[6]Ho, Q., 等人。通过陈旧同步并行参数服务器实现更有效的分布式机器学习。NIPS 2013.

[7]Crankshaw, D, 等人。复杂分析中缺失的一环：具有Velox的低延迟、可扩展的模型管理和服务。CIDR 2015. 另请参阅Schleier-Smith, J. 用于实时应用中敏捷机器学习的架构。KDD 2015.

[8]请访问http://www.vistrails.org.

第12章：对一个不断变化的目标的偏见观点：数据集成

作者：迈克尔·斯通布雷克

我将以数据集成中的两个主要主题的历史开始这篇论文。 在我看来，这个话题始于20世纪90年代的主要零售商将销售数据整合到数据仓库中。 为了做到这一点，他们需要从店内销售系统中提取数据，将其转换为预定义的公共表示（将其视为全局模式），然后将其加载到数据仓库中。 这个数据仓库保存了几年的历史销售数据，并被组织内的采购员用于库存轮换。 换句话说，采购员会发现宠物石头“过时了”，芭比娃娃“流行”。因此，他会与芭比娃娃工厂进行大订单，并将宠物石头提前并打折以清理库存。 一个典型的零售数据仓库通过更好的采购和库存轮换决策在一年内回本。 在20世纪90年代末和21世纪初，几乎所有企业都效仿零售商的做法，将面向客户的数据组织到数据仓库中。

一个新的行业应运而生，以支持数据仓库的加载，被称为提取、转换和加载（ETL）系统。 基本方法是：a）提前构建一个全局模式。b）派

遣一名程序员去每个数据源的所有者那里

，并让他弄清楚如何进行提取。从历史上看，编写这样的“连接器”是一项很大的工作，因为存在着晦涩的格式。希望随着更多开源和标准化的格式的出现，这个问题将变得不那么棘手。

c）让他编写转换脚本和必要的清理程序d）让他编写一个脚本将数据加载到数据仓库中

很明显，这种方法在处理大约十几个数据源时会遇到三个重大问题：

1. 需要提前准备一个全局模式。 大约在这个时候，许多企业都在推动为所有公司对象编写一个企业级模式。 一个团队负责完成这项工作，并将花费几年的时间。 在这段时间结束时，他们的成果已经落后了两年。

日期，并被宣布为失败。 因此，一个前期的全局模式对于广泛的领域来说非常难以构建。 这限制了数据仓库的合理范围。

2. 太多的人工劳动。 需要一个熟练的程序员来执行每个数据源的大部分步骤。

3. 数据集成和清洗基本上是困难的。 20世纪90年代的典型数据仓库项目超出预算的因素是两倍，延期的因素也是两倍。 问题在于规划者低估了数据集成挑战的难度。 有两个重要问题。首先，数据是脏的。 一个经验法则是你的数据中有10%是不正确的。 这是因为使用人名或产品的昵称，供应商的地址过时，人的年龄不正确等等。第二个问题是去重是困难的。 人们必须决定Mike Stonebraker和M. R. Stonebraker是同一个实体还是不同的实体。 同样具有挑战性的是同一地址上的两个餐厅。 它们可能在一个美食广场，一个可能已经取代了另一个独立的位置，或者这可能是一个数据错误。 在这种情况下弄清楚真相是昂贵的。

尽管存在这些问题，数据仓库对于面向客户的数据来说已经取得了巨大的成功，并且被大多数主要企业所使用。 在这种使用情况下，组合数据的组装带来的痛苦是合理的，因为它能带来更好的决策结果。 我几乎从来没有听到企业抱怨他们的数据仓库的运营成本。 相反，我听到的是业务分析师对更多数据源的渴望，无论是来自网络上的公共数据还是其他企业数据。 例如，平均大型企业拥有约5000个运营数据存储，而只有少数几个位于数据仓库中。

举个快速的例子，我曾经访问过一家主要的啤酒制造商。他拥有一个典型的数据仓库，记录了他的产品按品牌、经销商、时间段等销售情况。我告诉分析师，预测称那个冬天将会出现厄尔尼诺现象，西海岸会比正常年份更湿润、更温暖。

在东北地区。然后我问是否啤酒销售与温度或降水有关。 他们回答说：“我希望我们能回答这个问题，但天气数据不在我们的仓库中。” 使用ETL技术支持数据源的可扩展性非常困难。

快进到2000年代，新的时髦词汇是主数据管理（*MDM*）。 *MDM*的理念是标准化企业对重要实体（如客户、员工、销售、采购、供应商等）的表示，然后为每种实体类型精心策划一个主数据集，并让企业中的每个人都使用这个主数据集。 这有时被称为“黄金记录”。 实际上，以前的ETL供应商现在正在销售更广泛范围的*MDM*。

在我看来，MDM被过度炒作了。

让我从“谁会反对标准？”开始。当然不是我。然而，MDM存在以下问题，我将通过插图进行说明。

15年前我在Informix工作时，新任CEO在一次早期员工会议上问人力资源副总裁：“我们有多少员工？”她在下周回答说：“我不知道，也没有办法找到答案。”Informix在58个国家运营，每个国家都有自己的劳动法、员工定义等。没有员工的黄金记录。因此，回答CEO的问题的唯一方法是对这58个数据源进行数据集成以解决语义问题。 让58个国家经理同意这样做将是具有挑战性的，更加困难的是Informix甚至没有拥有所有涉及的组织。 新任CEO很快意识到这种做法的无效性。

那么为什么一家公司会允许发生这种情况呢？答案很简单：业务敏捷性。 Informix定期设立国家运营，并希望销售团队能够快速上手。 他们不可避免地会聘请一个国家经理，并告诉他“让它发生”。有时是分销商或其他独立实体。 如果他们说“这是你需要遵守的MDM黄金记录”，那么国家经理或分销商将花费数月时间试图将自己的需求与现有的MDM系统协调一致。 换句话说，MDM与业务敏捷性背道而驰。

显然，每个企业都需要在标准和敏捷性之间取得平衡。

第二个小插曲涉及一家大型制造企业。 出于业务敏捷性的原因，它们被分散成业务单元。 每个业务单元都有它自己的

拥有购买系统以指定业务单元与供应商互动的条款和条件。 这些系统大约有300个。 合并这些系统显然有回报率。 毕竟，维护的代码更少，企业可以获得比每个业务单元更好的合并条款。 那么为什么会有这么多采购系统？收购。这个企业主要通过收购来发展。 每次收购都成为一个新的业务单元，并带有自己的数据系统和合同。将所有这些数据系统合并到母公司的IT基础设施中通常是不可行的。 总之，收购破坏了主数据管理。

那么数据集成（或数据整理）包括什么？ 它包括以下步骤：

1. 摄取。必须找到并捕获数据源。 这需要解析用于存储的任何数据结构。
2. 转换。例如，欧元转换为美元或机场代码转换为城市名称。
3. 清理。必须找到并纠正数据错误。
4. 模式集成。你的工资就是我的薪水。
5. 合并（去重）。. M*ike* S*tone*br*aker*和M.R. S*ton e*br*aker*必须合并为一条记录。

ETL供应商以高成本和低可扩展性进行此操作。 MDM供应商具有类似的特点。
因此，存在着巨大的未满足需求。 大规模数据管理是一个“800磅的大猩猩在角落里”。那么，在这个领域中有哪些研究挑战呢？

让我们逐步进行这些步骤。摄取只是解析

数据源的问题。
其他人已经编写了这样的“连接器”，它们通常很昂贵。 一个有趣的挑战是半自动生成连接器。

数据转换也得到了广泛研究，主要集中在过去十年左右。[130, 54]中研究了用于指定转换的脚本/可视化工具。 数据整理工具[94]似乎是这个领域的最新技术，鼓励感兴趣的读者去了解一下。 此外，还有一些商业产品提供付费的转换服务（例如地址转换为（纬度，经度）或

公司到规范公司表示）。此外，关于从公共网络中找到感兴趣的转换的工作在[4]中有报道。

数据清洗已经使用各种技术进行研究。 [41]应用函数依赖性来发现错误数据并建议自动修复。

异常值检测（可能对应错误）已在许多情况下进行研究[87]。 [162, 158]是用于发现数据中有趣模式的查询系统。 这些模式可能对应错误。 [148]已研究过利用众包和专家资源来纠正错误，一旦它们被识别出来。 最后，有各种商业服务可以清理常见的领域，例如人员和日期的地址。 在我看来，数据清理研究必须利用真实世界的数据。 找到已注入到本来干净数据中的错误是不可信的。 不幸的是，真实世界的“脏”数据很难获得。

至少有20年来，模式匹配一直在广泛研究中。感兴趣的读者应该参考[117, 77, 134]以了解该领域的最新进展。

实体合并是在高维空间中（关于实体的所有属性，通常为25个或更多）找到接近的记录的问题。实际上，这是一个25维空间中的聚类问题。 这是一个N ** 2问题，在大规模情况下运行时间非常长。因此，明显使用近似算法是解决这个问题的方法。技术的调查出现在[87]中。

在我看来，真正的问题是一个端到端的系统。数据管理包括所有这些步骤，必须无缝集成，并且企业级系统必须能够进行大规模的管理。 一个有趣的端到端方法，似乎能够很好地扩展，是Data Tamer系统[148]。 另一方面，数据管理问题也发生在部门层面，一个个体贡献者想要整合一些数据源，上述的Data Wrangler系统似乎是一个有趣的方法。 有商业公司支持这两个系统，所以定期的改进应该会出现。

希望《红皮书》的下一版会有一系列关于这个领域的重要论文，以取代这个（自私的）行动呼吁。 在我看来，这是企业正在处理的最重要的话题之一。 我的一个警告是“实践出真知”。 如果你想在这个领域工作，你必须尝试你在真实世界企业数据上的想法。构建人工数据，注入错误，并且然后找到这些错误是不可信的。 如果你在这个领域有想法，我建议建立一个端到端的系统。 通过这种方式，你确保你正在解决一个重要的问题，而不仅仅是一个“点问题”，真实世界的用户可能会或可能不会感兴趣。

评论：乔・海勒斯坦

*2015*年*12*月*6*日

我同意迈克在这里的评估，但想要补充一下我对这个领域的看法，与他提到的“部门级别”问题有关。

根据与各种组织的用户的经验，我们发现数据转换越来越多地成为一个以用户为中心的任务，并且在很大程度上取决于用户体验：用于交互式评估和操作数据的界面和语言。

在今天的许多环境中，数据转换的正确结果严重依赖于上下文。 要了解数据是否有错误，你必须知道它应该是什么样子。 为了将数据转换为可用的形式，你需要了解它将被用于什么。即使在一个组织内部，数据的上下文和需求也会因人而异，随着时间的推移而变化。 将这一点添加到迈克对“黄金标准”的批评中-在许多现代用例中，尤其是在分析环境中，这根本就没有意义。

相反，你需要为最了解数据和使用情况的人设计工具，并使他们能够以敏捷、探索性的方式进行数据分析和转换。 计算机科学家往往为技术用户（IT专业人员和数据科学家）设计。 但在大多数组织中，了解数据和背景的用户更接近于“业务”而不是IT部门。 他们通常技术能力较弱。 与其使用传统的ETL工具，他们倾向于在电子表格和桌面数据库软件中操作数据，这两者都不适合评估数据质量或进行批量转换。 对于大型数据集，他们会在Microsoft Word中“编写代码”：他们用自然语言描述他们想要的工作流程，等待IT部门写代码，然后当他们得到结果时，通常会意识到结果并不完全符合他们的需求。 此时，他们对数据的需求往往已经发生了变化。 毫不奇怪，人们经常声称他们的时间的50-80%用于“准备数据”。（作为一个脚注，根据我的经验，现代的“数据科学家”倾向于使用Python、R或SAS DataStep中的临时脚本来处理数据，并且对代码质量非常放松。

以及这些脚本的修订控制。 因此，他们经常比老派ETL用户更糟糕！）商业用户选择图形工具是有原因的

：他们想要在数据被转换的过程中理解数据，并评估它是否越来越接近对他们的业务问题有用的形式。因此，数据库研究文献中的无人值守算法在这个领域中做得太少，没有解决关键问题。我相信最相关的解决方案将基于使人们能够直观地了解其数据状态并与算法合作以更好地用于目的的界面。这为创新提供了重要机会。

可视化和与数据交互的一般主题。 这是一个领域，数据库、人机交互和机器学习的思想可以结合起来，以创建解决实际、上下文特定问题的算法和人之间的互动合作。 为了支持这一点，我们需要能够提供即时数据概要（各种聚合）的交互式数据系统，以及能够在实时中预测用户之前提出多个可用的替代转换的系统。 近年来，我很高兴看到数据库社区在可视化和交互方面有更多的工作；这是一个很好的应用领域。

数据转换是探索的完美培养皿

[9]Heer, J., Hellerstein, J.M. 和 Kandel, S. ”数据转换的预测交互。” CIDR 2015.

所有读本列表

背景

Joseph M. Hellerstein 和 Michael Stonebraker. 循环往复.数据库系统读物,
第四版 (2005).

Joseph M. Hellerstein, Michael Stonebraker, James Hamilton. 数据库系统的架构.数据库基础与
趋势, 1, 2 (2007).

传统关系型数据库系统

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, Vera Watson. System R: 关系型数据库管理的方法.*ACM Transactions on Database Systems*, 1(2), 1976, 97-137.

Michael Stonebraker 和 Lawrence A. Rowe. POSTGRES 的设计. *SIGMOD*, 1986.

David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990, 44-62.

每个人都应该了解的技术

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price. Access path selection in a relational database management system. *SIGMOD*, 1979.

C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), 1992, 94-162.

Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger. 锁的粒度和一致性程度在共享数据
库中的应用. , IBM, 1975年9月.

Rakesh Agrawal, Michael J. Carey, Miron Livny. 并发控制性能建模：选择和影响。ACM 数据库系统事务，12(4)
，1987年，609-654。

C. Mohan, Bruce G. Lindsay, Ron Obermarck. R*分布式数据库管理系统中的事务管理.ACM数据库系统
交易, 11(4), 1986年, 378-396.

新的DBMS架构

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik. C-store: 一种列
式DBMS. *SIGMOD*, 2005年.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling. Hekaton: SQL Server的内存优化OLTP引擎. *SIGMOD*, 2013年.

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker. 通过镜子看OLTP，以及我们在那里
发现的东西。 *SIGMOD*，2008年。

大规模数据流引擎

Jeff Dean和Sanjay Ghemawat。MapReduce：大规模集群上的简化数据处理。 *OSDI*，2004年。

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu. DryadLINQ：通用分布式数据系统-

使用高级语言的并行计算。 *OSDI*，2008年。

弱隔离和分布

Atul Adya, Barbara Liskov, and Patrick O'Neil. 广义隔离级别定义. *ICDE*, 2000.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin,Swaminathan Sivasubramanian, Peter Vosshall, 和 Werner Vogels. Dynamo：亚马逊高可用键值存储。 *SOSP*，2007年。

Eric Brewer. CAP十二年后："规则"如何改变。*IEEE*计算机，45，2（2012年）。

查询优化

Goetz Graefe 和 William J. McKenna. Volcano优化器生成器：可扩展性和高效搜索。 *ICDE*，1993年。

Ron Avnur和Joseph M. Hellerstein。Eddies：持续自适应查询处理。 *SIGMOD*，2000年。

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, Miso Cilimdzic. 通过渐进优化实现鲁棒查询处理. *SIGMOD*, 2004.

交互式分析

Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman. 高效实现数据立方体. *SIGMOD*, 1996.

Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton. 一种基于数组的同时多维聚合算法. *SIGMOD*, 1997.

Joseph M. Hellerstein, Ron Avnur, Vijayshankar Raman. 在线查询处理的Informix控制. *Data Mining and Knowledge Discovery*, 4(4), 2000, 281-314.

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica. BlinkDB: 在大规模数据上具有有界误差和有界响应时间的查询. *EuroSys*, 2013.

语言

Joachim W. Schmidt. 关于数据类型关系的一些高级语言构造。*ACM* Transactions on Data*b*ase Systems，2(3)，1977，247-261。

Arvind Arasu, Shivnath Babu, Jennifer Widom. CQL连续查询语言：语义基础和查询执行。The *VLDB* Journa*l*，15(2)，2006，121-142。

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak. Bloom中的一致性分析：一种CALM和集中的方法。C*IDR*，2011。

Web数据

Sergey Brin和Larry Page。一个大规模超文本网络搜索引擎的解剖。 *WWW*，1998年。

Eric A. Brewer. 结合系统和数据库：搜索引擎回顾。数据库系统读物，第四版，2005。

Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, Yang Zhang. WebTables：探索网络表的力量。*VLDB*，2008。

参考文献

[1] Apache Tez. https://tez.apache.org/.

[2] Flexcoin: The Bitcoin Bank, 2014.http://www.flexcoin.com/; 原始来源：Emin G¨un Sirer.

[3] D. J. Abadi, D. Carney, U. C¸ etintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: 数据流管理的新模型和架构.*T*he *VLDB J*ournal*T*he *I*nterna-*tional Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[4] Z. Abedjan, J. Morcos, M. Gubanov, I. F. Ilyas, M. Stonebraker, P. Papotti, and M. Ouzzani. Dataxformer: 利用网络进行语义转换. 在 *CIDR*, 2015.

[5] S. Acharya, P. B. Gibbons, V. Poosala, 和 S. Ramaswamy. Aqua近似查询回答系统。在 *SIGMOD*, 1999年。

[6] A. Adya.弱一致性*:* 一个广义理论和分布式事务的乐观实现。博士论文, MIT, 1999年。

[7] R. Agrawal, T. Imieli´nski, 和 A. Swami. 在大型数据库中挖掘项集之间的关联规则。在 *SIGMOD*, 1993年。

[8] T. Akidau 等. 数据流模型: 在大规模、无界、乱序数据处理中平衡正确性、延迟和成本的实用方法。在 *VLDB*, 2015年。

[9] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, 等. 用于大数据分析的Stratosphere平台。V*LDB Jou*rnal, 23(6):939–964, 2014年。

[10] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, 等. Asterixdb: 一个可扩展的、开源的bdms. 在 *VLDB*, 2014年.

[11] P. Alvaro, P. Bailis, N. Conway, 和 J. M. Hellerstein. 无边界的一致性. 在 *SoCC*, 2013年.

[12] P. Alvaro, N. Conway, J. M. Hellerstein, 和 D. Maier. Blazes: 分布式程序的协调分析. 在*Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, 页码52–63. IEEE, 2014年.

[13] T. J. Ameloot. 声明性网络: 关于协调、正确性和声明性语义的最新理论工作.A*CM SIGMO*D Record, 43(2):5–16, 2014年.

[14] T. J. Ameloot, F. Neven, and J. Van den Bussche. 用于声明性网络的关系传感器。A*CM*期刊*(*JA*CM)*, 60(2): 15, 2013.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Spark中的关系数据处理。在 *SIGMOD*, 2015.

[16] S. Babu and H. Herodotou. 大规模并行数据库和MapReduce系统。数据库基础与趋势, 5(1):1–104, 2013.

[17] P. Bailis. 分布式数据库中的协调避免。 加州大学伯克利分校博士论文, 2015.

[18] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, 和 I. Stoica. Highly Available Transactions: Virtues and limitations. In *VLDB*, 2014.

[19] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, 和 I. Stoica. Feral Concurrency Control: An empirical investigation of modern application integrity. In *SIGMOD*, 2015.

[20] P. Bailis, A. Fekete, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, 和 I. Stoica. Coordination avoidance in database systems. In *VLDB*, 2015.

[21] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, 和 I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.

[22] P. Bailis 和 A. Ghodsi. 今天的最终一致性: 限制、扩展和更多. A*CM Q*ueue, 11(3), 2013.

[23] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein 和 I. Stoica. 可概率有界陈旧度的实用部分投票. 在 *VLDB*, 2012.

[24] S. Balakrishnan, A. Halevy, B. Harb, H. Lee, J. Madhavan, A. Rostamizadeh, W. Shen, K. Wilder, F. Wu 和 C. Yu. 在实践中应用网络表格. 在 *CIDR*, 2015.

[25] D. S. Batory. 关于搜索转置文件.*ACM T*ransactions on Data*b*ase Syste*m*s *(T*ODS*)*, 4*(*4*)*, 1979年12月.

[26] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. 多维数据库系统 rasdaman. 在 *SIGMOD*, 1998年。

[27] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 对ANSI SQL隔离级别的批评。在 *SI GMOD*, 1995年。

[28] P. Bernstein, V. Hadzilacos, and N. Goodman.并发控制和数据库系统恢复, 卷 370。 Addison-Wesley 纽约, 1987年。

[29] P. A. Bernstein and S. Das. 重新思考最终一致性。在 *SIGMOD*, 2013年。

[30] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: 超流水线查询执行。在 *CIDR*, 2005年。

[31] E. Brewer等。巨型规模服务的教训。*I*EEE互联网计算, 5(4):46–55, 2001.

[32] M. Burrows。松散耦合分布式系统的Chubby锁服务。在 *OSDI*, 2006.

[33] D. D. Chamberlin。SQL的早期历史。*IEEE*计算历史年鉴, 34(4):78–82, 2012.

[34] D. D. Chamberlin和R. F. Boyce。Sequel: 一种结构化的英语查询语言。 在*1974*年AC*M* SI*G*FID*ET (*现在SI *GMO*D*)*数据描述、访问和控制研讨会论文集*,* 页码*249*–264. AC*M,*1974.

[35] T. D. Chandra, R. Griesemer和J. Redstone。Paxos的实时工程视角。在 *PODC*, 2007.

[36] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, et al. Telegraphcq: 连续数据流处理的不确定世界. 在 *CIDR*, 2003.

[37] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: 一个分布式结构化数据存储系统. 在 *OSDI*, 2006.

[38] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. StatusQuo: 使用程序分析使熟悉的抽象执行. 在 *CIDR*, 2013.

[39] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. 项目adam：构建高效可扩展的深度学习训练系统。在 *OSDI*, 2014年。

[40] J. Choi等。ScaLAPACK：一种用于分布式内存计算机的可移植线性代数库-设计问题和性能。 在应用并行计算物理、化学和工程科学, 1996年，第95-106页。Springer。

[41] X. Chu, I. F. Ilyas, and P. Papotti. 全面的数据清理：将违规放入上下文中。在 *ICDE*, 2013年。

[42] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. 可扩展的可交换性规则：为多核处理器设计可扩展软件。ACM计算机系统交易（T*OCS*）, 32*(4)*:10, 2015年。

[43] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *VLDB Endowment*会议论文, 1(1):1153–1165, 2008.

[44] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. 大规模数据的概要: 样本, 直方图, 小波, 轮廓.数据库基础与趋势, 4(1–3):1–294, 2012.

[45] C. J. Date. 高级语言数据库扩展的架构. 在 *SIGMOD*, 1976.

[46] C. J. Date. 对SQL数据库语言的批评.AC*M* S*IGMO*D *R*ecord, 14(3), 1984年11月.

[47] S. Davidson, H. Garcia-Molina, and D. Skeen. 分区网络的一致性.*A*CM CS*UR*, 17(3):341–370, 1985年.

[48] J. Dean. 设计、经验和建议：构建大型分布式系统的教训（主题演讲）。在 *LADIS*, 2009年。

[49] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, 等。大规模分布式深度网络。 在*Advances in Neural Information Processing Systems*, 2012年，第1223–1231页。

[50] A. Deshpande. Eddies的开销初步研究。A*CM SIGMO*D *R*ecord, 2004年，33(1):44–49。

[51] A. Deshpande 和 J. M. Hellerstein. 解放自适应查询处理的历史负担。在 *VLDB*, 2004年。

[52] A. Deshpande, Z. Ives 和 V. Raman. 自适应查询处理。 *F*oundations and *T*rends in Data*b*ases, 2007年，1(1): 1–140。

[53] D. DeWitt 和 M. Stonebraker. Mapreduce: 一个重大的倒退。数据库专栏, 2008.

[54] T. Dohzen, M. Pamuk, S.-W. Seong, J. Hammer, 和 M. Stonebraker. 在 morpheus 项目中通过转换重用进行数据集成。在 *SIGMOD*, 2006.

[55] J. Duggan 和 M. Stonebraker. 数组数据库的增量弹性。 在*Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 409–420. ACM, 2014.

[56] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, 等. BigDAWG 多存储系统的演示。在 *VLDB*, 2015.

[57] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 在数据库系统中一致性和谓词锁的概念。A*CM*通信, 19(11):624–633, 1976年。

[58] J. Fan, A. Gerald, S. Raj, and J. M. Patel. 反对专门的图分析引擎的案例。在 *CIDR*, 2015年。

[59] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. 使快照隔离可串行化。 A*C*M *TODS*, 30(2):492–528, 2005年6月。

[60] M. J. Fischer, N. A. Lynch, and M. S. Paterson. 一个有故障进程的分布式一致性不可能。*ACM*期刊*(JACM)*, 32(2):374–382, 1985.

[61] M. J. Franklin. 并发控制和恢复.计算机科学与工程手册, 页码 1–58–1077, 1997.

[62] S. Ghemawat, H. Gobioff, 和 S.-T. Leung. 谷歌文件系统. 在 *SOSP*, 2003.

[63] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, 和 S. Vaithyanathan. Systemml: 声明式机器学习在mapreduce上的应用. 在 *ICDE*, 2011.

[64] J. E. Gonzales, R. S. Xin, D. Crankshaw, A. Dave, M. J. Franklin, 和 I. Stoica. Graphx: 统一数据并行和图并行分析. 在 *OSDI*, 2014.

[65] G. Graefe. 查询优化的级联框架。*IEEE*数据工程通报。，18(3)：19-29，1995年。

[66] G. Graefe. 二十年后的五分钟规则，以及闪存如何改变规则。 *DaMoN*，2007年。

[67] J. Gray. 数据库操作系统笔记。 在操作系统：高级课程，卷60的计算机科学讲义，页393-481。 Springer Berlin Heidelberg，1978年。

[68] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. 数据cube：一个泛化group-by、交叉表和子总计的关系聚合运算符。 数据挖掘和知识发现，1(1)：29-53，1997年。

[69] J. Gray 和 G. Graefe. 十年后的五分钟规则，以及其他计算机存储规则。*ACM SIGMOD* 记录，26(4):63–68, 1997.

[70] J. Gray, P. Helland, P. ONeil, 和 D. Shasha. 复制的危险和解决方案。在 *SIGMOD*，1996年。

[71] J. Gray 和 L. Lamport. 事务提交的共识。*ACM* 数据库系统交易 *(TODS)*，31*(1)*:133–160, *M*ar. 2006.

[72] J. Gray, R. Lorie, G. Putzolu, 和 I. Traiger. 锁的粒度和一致性程度在共享数据库中。base. 技术报告，IBM，1976年。

[73] J. Gray 和 F. Putzolu. 5分钟规则用于交换内存和磁盘访问，以及10字节规则用于交换内存和CPU时间。在 *SIGMOD*，1987年。

[74] T. J. Green, S. S. Huang, B. T. Loo, 和 W. Zhou. Datalog 和递归查询处理.数据库基础和数据库趋势, 5(2):105–195, 2013.

[75] R. Guerraoui. 重新审视非阻塞原子提交和一致性之间的关系. 在 *WDAG*, 1995.

[76] R. Guerraoui, M. Larrea, 和 A. Schiper. 具有不可靠故障检测器的非阻塞原子提交. 在 *SRDS*, 1995.

[77] L. Haas, D. Kossmann, E. Wimmers, 和 J. Yang. 优化跨多样数据源的查询. 在 *VLDB*, 1997.

[78] T. Haerder 和 A. Reuter. 事务导向数据库恢复原则. A*CM* 计算调查 *(CSUR)*, 15(4):287–317, 1983.

[79] A. Halevy, P. Norvig, and F. Pereira. 数据的非理性有效性。*IEEE*智能系统，24(2):8–12，2009年3月。

[80] D. H. Hansson等。Ruby on Rails。http://www.rubyonrails.org。

[81] D. Harris. Forbes: 为什么Cloudera说“再见，MapReduce”，“你好，Spark”，2015年。 http://fortune.com/2015/09/09/cloudera-spark-mapreduce/。

[82] M. Hausenblas和J. Nadeau。Apache Drill：规模化的交互式即席分析。 大数据，1(2):100–104，2013年。

[83] P. Helland和D. Campbell。在CID*R*上构建在流沙上。

[84] J. M. Hellerstein。声明性命令：分布式逻辑的经验和猜想。*ACM SIGMOD Record*，39(1):5–19，2010年。

[85] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, 等. MADlib分析库：或者MAD技能，SQL。在 *VLDB*, 2012年。

[86] T. Ibaraki 和 T. Kameda. 计算n关系连接的最佳嵌套顺序。*ACM* Transactions on *Database Systems (TODS)*, 9(3):482–502, 1984年。

[87] I. F. Ilyas 和 X. Chu. 关系数据清理的趋势：一致性和去重。 Foundations and *Trends in Databases*, 5(4):281–393, 2012年。

[88] Y. E. Ioannidis 和 S. Christodoulakis. 关于连接结果大小误差的传播。在 *SIGMOD*, 1991年。

[89] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: 从顺序构建块中的分布式数据并行程序. 在 *EuroSys*, 2007.

[90] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: 你的关系友用于图分析！ 在 *VLDB*, 2014.

[91] P. R. Johnson and R. H. Thomas. Rfc 667: 重复数据库的维护. 技术报告，1 1975.

[92] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: 高性能主备广播系统. 在 *DSN*, 2011.

[93] N. Kabra and D. J. DeWitt. 高效的中间查询重新优化子优化查询执行计划. 在 *SIGMOD*, 1998.

[94] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: 交互式可视化数据转换脚本. 在 *CHI*, 2011.

[95] J. Kepner 等. 动态分布式维度数据模型 (D4M) 数据库和计算系统. 在声学、语音和信号处理 *(ICASSP), 2012 IEEE* 国际会议上, 页码 5349–5352. IEEE, 2012.

[96] R. Kimball 和 M. Ross.数据仓库工具包*:* 维度建模完全指南. John Wiley & Sons, 2011.

[97] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, 等. 多租户数据中心的网络虚拟化. 在*USENIX NSDI*, 2014.

[98] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, 等。Impala: 一个现代的、开源的Hadoop SQL引擎。在 *CIDR*, 2015年。

[99] L. Lamport. 兼职议会。*ACM*计算机系统交易（T*OCS*）, 16*(2)*:133–169, 1998年。

[100] B. Lampson 和 H. Sturgis. 分布式数据存储系统中的崩溃恢复。技术报告, 1979年。

[101] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, 和 B.-Y. Su. 使用参数服务器扩展分布式机器学习。在 *OSDI*, 2014年。

[102] B. Liskov和J. Cowling. 视图复制再探讨。 技术报告，麻省理工学院，2012年。

[103] G. M. Lohman. 类似语法的功能规则用于表示查询优化的替代方案。在 *SIGMOD*, 1988年。

[104] R. Lorie和A. Symonds. 用于交互应用的关系访问方法。*Cou*rant计算机科学研讨会，第*6*卷：数据库系统，1971年。

[105] Y. Low，D. Bickson，J. Gonzalez，C. Guestrin，A. Kyrola和J. M. Hellerstein. 分布式图形实验室：云中的机器学习和数据挖掘框架。在 *VLDB*，2012年。

[106] H. Lu，K. Veeraraghavan，P. Ajoux，J. Hunt，Y. J. Song，W. Tobagus，S. Kumar和W. Lloyd. 存在一致性：测量和理解Facebook的一致性。在 *SOSP*，2015年。

[107] H. P. Luhn. 为信息检索系统自动编码文档。文档的现代趋势，第45-58页，1959年。

[108] R. MacNicol和B. French。用于分析的Sybase iq多路复用设计。在 *VLDB*，2004年。

[109] S. Madden，M. Shah，J. M. Hellerstein和V. Raman。流上的连续自适应连续查询。在 *SIGMOD*，2002年。

[110] G. Malewicz，M. H. Austern，A. J. Bik，J. C. Dehnert，I. Horn，N. Leiser和G. Czajkowski。Pregel：用于大规模图处理的系统。 *SIGMOD*，2010年。

[111] N. Malviya，A. Weisberg，S. Madden和M. Stonebraker。重新思考主存储器OLTP恢复。在 *ICDE*，2014年。

[112] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman.*4.4 BSD*操作系统的设计与实现. Pearson Education, 1996.

[113] F. McSherry, M. Isard, and D. G. Murray. 可扩展性！但是代价是什么？在 *HotOS*, 2015.

[114] E. Meijer. 你的鼠标是一个数据库。 *Queue*, 10(3):20, 2012.

[115] E. Meijer, B. Beckman, and G. Bierman. Linq: 在.NET框架中协调对象、关系和XML。在 *SIGMOD*, 2006.

[116] J. Melton, J. E. Michels, V. Josifovski, K. Kulkarni, and P. Schwarz. Sql/med: 一个状态报告。*ACM SIGMOD Record*, 31(3):81–89, 2002.

[117] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. H. Ho, R. Fagin, and L. Popa. The clio project: managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[118] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[119] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. 在 *SOSP*, 2013年.

[120] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, 等. Niagara互联网查询系统.*IEEE*数据工程通报, 24(2):27–33, 2001年.

[121] H. Q. Ngo, E. Porat, C. R´e, 和 A. Rudra. 最坏情况下的最优连接算法:[扩展摘要]. 在数据库系统原理第*31*届研讨会上, 页码*37*–48. ACM, 2012年.

[122] F. Olken.从数据库中随机抽样. 加州大学伯克利分校博士论文, 1993年.

[123] C. Olston, B. Reed, U. Srivastava, R. Kumar, 和 A. Tomkins. Pig Latin: 一种不那么陌生的数据处理语言. 在 *SIGMOD*, 2008年.

[124] P. E. O'Neil. 托管交易方法。*ACM*数据库系统交易, 11(4):405–430, 1986.

[125] D. Ongaro和J. Ousterhout. 寻找可理解的共识算法。在*USENIX ATC*, 2014.

[126] L. Page, S. Brin, R. Motwani和T. Winograd. PageRank引用排名: 为网络带来秩序。斯坦福信息实验室技术报告, 1999. SIDL-WP-1999-0120.

[127] R. Ramakrishnan和J. Gehrke.数据库管理系统. 麦格劳希尔, 2000.

[128] R. Ramakrishnan和S. Sudarshan. 自上而下与自下而上的再探讨。 在国际逻辑编程研讨会论文集, 第321–336页, 1991.

[129] V. Raman, A. Deshpande, and J. M. Hellerstein. 使用状态模块进行自适应查询处理。在 *ICDE*。IEEE，2003年。

[130] V. Raman and J. M. Hellerstein. Potter's wheel: 一个交互式数据清理系统。在 *VLDB*，2001年。

[131] V. Raman and J. M. Hellerstein. 在线查询处理的部分结果。 在 *SIGMOD*，页码275-286。ACM，2002年。

[132] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: 一个I/O高效的MapReduce。在 *SoCC*，2012年。

[133] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: 一种无锁并行化随机梯度下降方法。 在*Advances in Neural Information Processing Systems*，页码693-701，2011年。

[134] M. T. Roth 和 P. M. Schwarz. 不要丢弃它，包装它！一个用于传统数据源的包装器架构。在 *VLDB*，1997年。

[135] L. A. Rowe 和 K. A. Shoens. RIGEL中的数据抽象、视图和更新。在 *SIGMOD*，1979年。

[136] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster 和 J. Gehrke. 自稳协议：通过程序分析避免事务协调。在 *SIGMOD*，2015年。

[137] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster 和 J. Gehrke. 自稳协议：通过程序分析避免事务协调。在 *SIGMOD*，2015年。

[138] Y. Saito 和 M. Shapiro. 乐观复制.*ACM* 计算机调查., 37(1), 2005年3月.

[139] G. Salton 和 M. E. Lesk. 索引和文本处理的计算机评估.*A*CM 通信 *(JA*CM*)*, 15(1):8–36, 1968年.

[140] J. W. Schmidt. 一些高级语言构造用于关系型数据.*ACM* 数据库系统交易., 2(3), 1977年9月.

[141] F. B. Schneider. 使用状态机方法实现容错服务: 教程. *ACM* 计算机调查 *(CSUR)*, 22(4):299–319, 1990年.

[142] M. Shapiro, N. Preguica, C. Baquero, 和 M. Zawirski. 收敛和交换复制数据类型的综合研究. INRIA TR 7506, 2011年.

[143] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, 等. F1: 一个可扩展的分布式SQL数据库. 在 *VLDB*, 2013年.

[144] N. Siva. 1000基因组计划.自然生物技术, 26(3):256–256, 2008年.

[145] D. Skeen. 非阻塞提交协议. 在 *SIGMOD*, 1981年.

[146] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. Jordan, T. Kraska, 等. Mli: 一个分布式机器学习的API. 在 *ICDM*, 2013年.

[147] M. Stonebraker. 土地鲨鱼在喋喋不休. 通信*AC*M. 即将出版.

[148] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, 和 S. Xu. 大规模数据整理: 数据驯服系统. 在 *CIDR*, 2013.

[149] M. Stonebraker 和 U. C¸ etintemel. “一刀切”：一个时代已经过去的想法. 在 *ICDE*, 2005.

[150] M. Stonebraker, G. Held, E. Wong, 和 P. Kreps. ingres的设计和实现.*AC*M数据库系统交易 *(TODS)*, 1(3):189–222, 1976.

[151] M. Stonebraker, S. Madden, 和 P. Dubey. Intel 大数据科学与技术中心的愿景和执行计划。*AC*M SI*G*M*O*D *R*ecord, 42(1):44–49, 2013.

[152] M. Stonebraker 和 E. Neuhold. 拉古纳海滩报告。 技术报告 1, 国际计算机科学研究所, 1989.

[153] D. Terry. 通过棒球解释复制数据一致性。*ACM* 通信, 56(12):82–89, 2013.

[154] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, 等. 弱一致性复制数据的会话保证。在 *PDIS*, 1994.

[155] SciDB 开发团队. SciDB 概述: 大规模数组存储、处理和分析。在 *SIGMOD*, 2010.

[156] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: 一个基于map-reduce框架的数据仓库解决方案. 在 *VLDB*, 2009年.

[157] T. Urhan, M. J. Franklin, and L. Amsaleg. 基于成本的查询混淆以减少初始延迟. ACM *SIGMOD Record*, 27(2):130–141, 1998年.

[158] M. Vartak, S. Madden, A. Parameswaran, and N. Polyzotis. Seedb: 自动生成查询可视化-izations. 在 *VLDB*, 2014年.

[159] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. 商业云存储中的数据一致性属性和权衡: 消费者的视角. 在 *CIDR*, 2011年.

[160] A. N. Wilschut和P. M. Apers. 数据流查询在并行主存环境中的执行。 在并行和分布式信息系统，1991年第一届国际会议论文集，第68-77页。
IEEE，1991年。

[161] E. Wong和K. Youssefi. 查询处理的分解策略。 *ACM*数据库系统交易（*TOD*S），1（3）：223-241，1976年。

[162] E. Wu和S. Madden. Scorpion：解释聚合查询中的异常值。在 *VLDB*，2013年。

[163] M. Zaharia，M. Chowdhury，T. Das，A. Dave，J. Ma，M. McCauley，M. J. Franklin，S. Shenker和I. Stoica。弹性分布式数据集：内存集群计算的容错抽象。在 *NSDI*，2012年。