基础和趋势® in
数据库
第1卷，第2期（2007年）141-259

DOI: 10.1561/1900000002

now
the essence of knowledge

数据库系统的架构

**Joseph M. Hellerstein[1], Michael Stonebraker[2]
和James Hamilton[3]**

1 加利福尼亚大学伯克利分校，美国，*hellerstein@cs.berkeley.edu*
2 麻省理工学院，美国
3 微软研究，美国

摘要

数据库管理系统（DBMS）是现代计算的普遍和关键组成部分，是几十年来学术界和工业界研究和开发的结果。 从历史上看，DBMS是最早开发的多用户服务器系统之一，因此在可扩展性和可靠性方面开创了许多系统设计技术，现在在许多其他环境中使用。 虽然DBMS使用的许多算法和抽象是教科书材料，但在文献中对使DBMS工作的系统设计问题的研究相对较少。 本文介绍了DBMS设计原则的架构讨论，包括进程模型、并行架构、存储系统设计、事务系统实现、查询处理器和优化器架构，以及典型的共享组件和实用程序。 成功的商业和开源系统被用作参考点，特别是当不同团体采用了多种替代设计时。

1

介绍

数据库管理系统（DBMS）是复杂的、至关重要的软件系统。 今天的DBMS体现了几十年的学术和工业研究以及公司软件开发的成果。

数据库系统是最早广泛部署的在线服务器系统之一，因此在数据管理、应用程序、操作系统和网络服务等方面开创了设计解决方案。 早期的DBMS是计算机科学中最有影响力的软件系统之一，DBMS的思想和实现问题被广泛复制和重新发明。

由于一些原因，数据库系统架构的经验教训并不像应该那样广为人知。首先，应用数据库系统的社区规模相对较小。 由于市场力量只支持少数几个高端竞争对手，成功的DBMS实现只有少数几个。 参与设计和实现数据库系统的人员群体非常紧密：许多人在同一所学校就读，参与同一项有影响力的研究项目，并在同一款商业产品上合作。 其次，学术界对数据库系统的处理通常忽略了架构问题。 传统的数据库系统教材主要关注算法

而不涉及系统架构的全面实现的综合讨论，这些讨论是自然的教学、研究和测试的理论问题。 总之，关于如何构建数据库系统的许多常识是可用的，但很少有人将其写下来或广泛传播。

在本文中，我们试图捕捉现代数据库系统的主要架构方面，并讨论一些高级主题。 其中一些出现在文献中，我们会提供适当的参考文献。 其他问题埋藏在产品手册中，有些只是社区口头传统的一部分。 在适当的情况下，我们使用商业和开源系统作为讨论的各种架构形式的示例。 然而，由于篇幅限制，我们无法列举这些多百万行代码库中出现的例外和细微差别，其中大部分已经存在了十多年。我们的目标是关注整体系统设计和强调通常不在教科书中讨论的问题，为更广为人知的算法和概念提供有用的背景。 我们假设读者熟悉教科书数据库系统材料（例如，[72]或[83]）以及现代操作系统（如UNIX、Linux或Windows）的基本功能。在下一节中介绍DBMS的高级架构后，我们提供了一些参考文献，以了解第1.2节中每个组件的背景知识。

1.1 关系系统：查询的生命周期

目前在生产中最成熟和广泛使用的数据库系统是关系数据库管理系统（RDBMSs）。 这些系统可以在许多世界应用基础设施的核心找到，包括电子商务、医疗记录、计费、人力资源、工资单、客户关系管理和供应链管理等。网络商务和面向社区的网站的出现只增加了它们的使用量和广度。关系系统几乎是所有在线交易和大多数在线内容管理系统（博客、维基、社交网络等）的记录库。 除了作为重要的软件基础设施外，关系数据库系统还充当

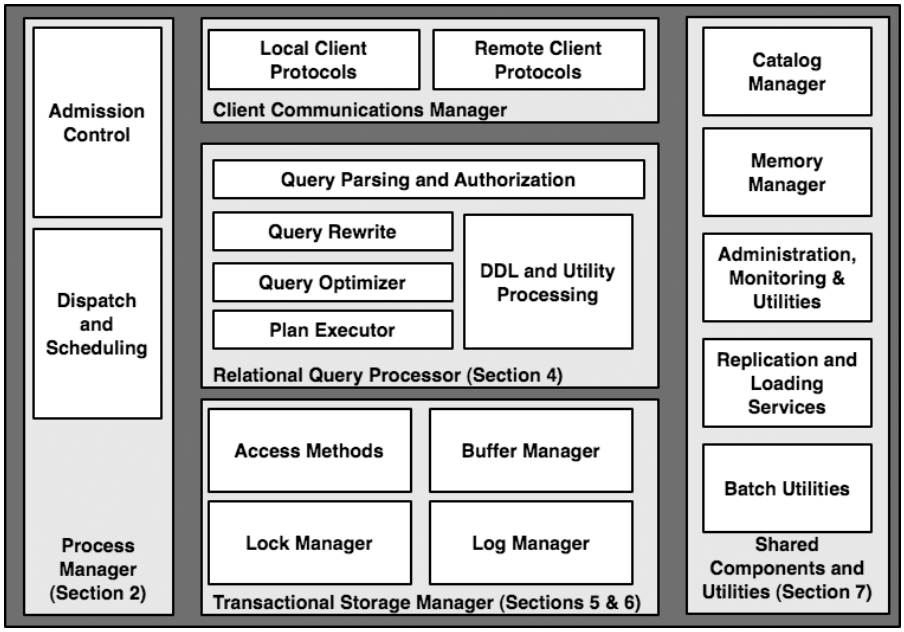


图1.1 DBMS的主要组件。

因此，我们在本文中重点关注关系数据库系统。 因此，我们在本文中重点关注关系数据库系统。

从根本上说，一个典型的关系型数据库管理系统（RDBMS）有五个主要组件，如图1.1所示。作为对这些组件及其相互关系的介绍，我们将逐步介绍数据库系统中查询的生命周期。 这也是本文剩余部分的概述。

考虑一个简单但典型的机场数据库交互，其中一个登机口代理点击一个表单来请求某个航班的乘客名单。 这个按钮点击会导致一个单查询事务的执行，大致如下：

1. 机场登机口的个人电脑（“客户端”）调用一个API，该API通过网络与数据库管理系统的客户端通信管理器（图1.1顶部）进行通信。在某些情况下，这个连接

是直接在客户端和数据库服务器之间建立的，例如通过ODBC或JDBC连接协议。
这种安排被称为“两层”或“客户端-服务器”系统。 在其他情况下，客户端可能与“中间层服务器”（Web服务器、事务处理监视器或类似的服务器）进行通信，后者再使用协议来代理客户端和数据库管理系统之间的通信。这通常被称为“三层”系统。 在许多基于Web的场景中，Web服务器和数据库管理系统之间还有另一个“应用服务器”层，从而形成四层。 鉴于这些不同的选择，一个典型的数据库管理系统需要与许多不同的连接协议兼容，这些协议由各种客户端驱动程序和中间件系统使用。 然而，在基础层面上，无论使用哪种协议，数据库管理系统的客户端通信管理器的责任基本上是相同的：建立和记住调用者（无论是客户端还是中间件服务器）的连接状态，响应调用者的SQL命令，并适当地返回数据和控制消息（结果代码、错误等）。 在我们的简单示例中，通信管理器将建立客户端的安全凭据，设置状态以记住新连接和当前SQL命令的详细信息，并将客户端的第一个请求转发到数据库管理系统以进行处理。

2. 在接收到客户端的第一个SQL命令时，数据库管理系统（DBMS）必须为该命令分配一个“计算线程”。 它还必须确保线程的数据和控制输出通过通信管理器连接到客户端。 这些任务是DBMS的进程管理器（图1.1左侧）的工作。在查询阶段，DBMS需要做出的最重要决策是关于准入控制的：系统是否应立即开始处理查询，还是推迟执行，直到有足够的系统资源可用于该查询。 我们在第2节中详细讨论进程管理。

3. 一旦被准入并分配为控制线程，门户代理的查询就可以开始执行。 它通过调用关系查询处理器（图1.1中心）中的代码来实现。这组模块检查用户是否有权运行查询，并将用户的SQL查询文本编译成内部查询计划。 一旦编译完成，生成的查询计划将通过计划执行器处理。 计划执行器由一套“操作符”（关系算法实现）组成，用于执行任何查询。 典型的操作符实现了关系查询处理任务，包括连接、选择、投影、聚合、排序等，还包括从系统的较低层请求数据记录的调用。 在我们的示例查询中，查询优化过程组装了一小部分这些操作符，以满足门户代理的查询。 我们在第4节中讨论查询处理器。

4. 在门控代理的查询计划的基础上，存在一个或多个操作符来请求数据库中的数据。 这些操作符调用DBMS的事务存储管理器（图1.1，底部）来获取数据，该管理器负责所有数据访问（读取）和操作（创建、更新、删除）的调用。 存储系统包括用于组织和访问磁盘上的数据的算法和数据结构（“访问方法”），包括表格和索引等基本结构。 它还包括一个缓冲管理模块，用于决定何时以及何种数据在磁盘和内存缓冲区之间传输。 回到我们的例子，在访问访问方法中的数据过程中，门控代理的查询必须调用事务管理代码来确保事务的“ACID”属性[30]（在第5.1节中详细讨论）。在访问数据之前，从锁管理器获取锁以确保在面对其他并发查询时正确执行。 如果门控代理的查询涉及对数据库的更新，它将与日志管理器交互，以确保事务在提交时是持久的，并在中止时完全撤消。

在第5节中，我们将更详细地讨论存储和缓冲管理；第6节涵盖了事务一致性架构。

5. 在示例查询的生命周期中，它已经开始访问数据记录，并准备使用它们来计算客户端的结果。 这是通过“展开堆栈”来完成的，我们在这一点上进行了描述。 访问方法将控制权返回给查询执行器的操作员，它们协调从数据库数据计算结果元组；生成结果元组时，它们被放置在客户端通信管理器的缓冲区中，该管理器将结果返回给调用者。 对于大型结果集，客户端通常会进行额外的调用，逐步从查询中获取更多数据；这导致通过通信管理器、查询执行器和存储管理器进行多次迭代。 在我们的简单示例中，查询结束时，事务完成并关闭连接；这导致事务管理器清理事务的状态，进程管理器释放查询的任何控制结构，通信管理器清理连接的通信状态。

我们对这个示例查询的讨论涉及了关系数据库管理系统中的许多关键组件，但并非全部。 图1.1的右侧描绘了许多共享组件和实用程序，这些对于全功能数据库管理系统的运行至关重要。目录和内存管理器在任何事务中都被作为实用程序调用，包括我们的示例查询。 目录在身份验证、解析和查询优化过程中由查询处理器使用。 内存管理器在数据库管理系统中的任何时候需要动态分配或释放内存时都会被使用。 图1.1最右边的框中列出的剩余模块是独立于任何特定查询运行的实用程序，使整个数据库保持良好调整和可靠。 我们在第7节中讨论了这些共享组件和实用程序。

1.2 范围和概述

在本文的大部分内容中，我们的重点是支持核心数据库功能的架构基础。 我们不试图提供对已在文献中广泛记录的数据库算法的全面回顾。 我们也只提供了对现代数据库管理系统中许多扩展的最少讨论，其中大部分提供了超出核心数据管理的功能，但并不显著改变系统架构。 然而，在本文的各个部分中，我们注意到了超出本文范围的感兴趣的主题，并在可能的情况下提供了额外阅读的指引。

我们从整体上研究数据库系统的架构开始讨论。 任何服务器系统架构的第一个主题是其整体进程结构，我们探索了各种可行的选择，首先是针对单处理器机器，然后是针对当今各种并行架构。 这个关于核心服务器系统架构的讨论适用于各种系统，但在数据库管理系统设计中得到了很大程度的开创。 在此之后，我们开始讨论数据库管理系统更具领域特定性的组件。我们从单个查询对系统的视图开始，重点关注关系查询处理器。 在此之后，我们进入存储架构和事务存储管理设计。 最后，我们介绍了大多数数据库管理系统中存在但很少在教科书中讨论的一些共享组件和实用工具。

2

进程模型

在设计任何多用户服务器时，需要尽早决定并发用户请求的执行方式以及如何将其映射到操作系统进程或线程。 这些决策对系统的软件架构以及其性能、可扩展性和跨操作系统的可移植性有着深远的影响。[1] 在本节中，我们对DBMS进程模型提供了一些选项，这些选项可以作为许多其他高度并发的服务器系统的模板。 我们从一个简化的框架开始，假设操作系统对线程有良好的支持，并且最初只针对单处理器系统。 然后，我们扩展这个简化的讨论，以应对现代数据库管理系统实现其进程模型的现实情况。 在第3节中，我们讨论了利用计算机集群以及多处理器和多核系统的技术。

下面的讨论依赖于以下定义：

- 操作系统进程将操作系统（*OS*）程序执行单元（控制线程）与一个

[1] 许多但不是所有的DBMS都设计成可在各种主机操作系统上移植。 值得注意的OS特定的DBMS有zSeries的DB2和Microsoft SQLServer。这些产品不仅使用广泛可用的OS设施，还可以利用其单个主机的独特设施。

进程的私有地址空间。 进程的状态中包括OS资源句柄和安全上下文。 这个单独的程序执行单元由OS内核调度，每个进程都有自己独特的地址空间。

- 操作系统线程是一个没有额外私有OS上下文和没有私有地址空间的OS程序执行单元。 每个OS线程都可以完全访问在同一个多线程OS进程中执行的其他线程的内存。线程的执行由操作系统内核调度器调度，这些线程通常被称为“内核线程”或k-线程。

- 轻量级线程包是一个应用级构造，支持在一个操作系统进程中运行多个线程。 与由操作系统调度的操作系统线程不同，轻量级线程由应用级线程调度器调度。 轻量级线程与内核线程的区别在于，轻量级线程在用户空间中调度，不涉及或不知晓内核调度器的参与。 用户空间调度器及其所有轻量级线程组合在一个操作系统进程中运行，并对操作系统调度器呈现为单个执行线程。

  与操作系统线程相比，轻量级线程具有更快的线程切换速度，因为不需要进行操作系统内核模式切换来调度下一个线程。 然而，轻量级线程的缺点是，任何线程的阻塞操作（如同步I/O）都会阻塞进程中的所有线程。

  这会阻止其他线程在一个线程等待操作系统资源时取得进展。
  轻量级线程包通过(1)仅发出异步(非阻塞)I/O请求和(2)不调用任何可能阻塞的操作来避免这种情况。 一般来说，轻量级线程比基于操作系统进程或操作系统线程编写软件提供了更困难的编程模型。

- 一些数据库管理系统实现了自己的轻量级线程(LWT)包。 这些是一般LWT包的特例。 我们将这些线程称为数据库管理系统线程，简称线程，当讨论数据库管理系统、一般LWT和操作系统线程之间的区别不重要时。

- 数据库管理系统客户端是实现应用程序与数据库管理系统通信所使用的API的软件组件。一些示例数据库访问API包括J*DBC*、O*DBC*和OLE/*DB*。此外，还有各种各样的专有数据库访问API集。 一些程序使用嵌入式*S*QL编写，这是一种将编程语言语句与数据库访问语句混合的技术。 这最初是在I*BM*的*C*O*B*OL和PL/I中实现的，后来在实现了Java的*S*QL/J中实现。嵌入式*S*QL由预处理器处理，将嵌入式*S*QL语句转换为对数据访问API的直接调用。无论客户端程序中使用的语法是什么，最终结果都是一系列对数据库管理系统数据访问API的调用。对这些API的调用由数据库管理系统客户端组件进行编组，并通过某种通信协议发送到数据库管理系统。 这些协议通常是专有的，而且通常没有文档记录。 过去，有过几次努力标准化客户端到数据库通信协议，其中最著名的是Op*en* Group *D*R*D*A，但没有一个取得广泛的采用。

- *DBMS*工作线程是*DBMS*中代表*DBMS*客户端执行工作的线程。*DBMS*工作线程与*DBMS*客户端之间存在一对一的映射关系：*DBMS*工作线程处理来自单个*DBMS*客户端的所有*S*QL请求。*DBMS*客户端将*S*QL请求发送到*DBMS*服务器。 工作线程执行每个请求并将结果返回给客户端。 接下来，我们将研究商业*DBMS*使用的不同方法，将*DBMS*工作线程映射到操作系统线程或进程上。当区分明显时，我们将称之为工作线程或工作进程。

否则，我们简称为工作线程或DBMS工作线程。2.1 单处理器和轻量级线程

在本小节中，我们概述了一个简化的DBMS进程模型分类。目前，很少有领先的DBMS完全按照本节描述的方式进行架构，但这些材料为我们更详细地讨论当前一代生产系统奠定了基础。

如今，每个领先的数据库系统都是至少一个这里介绍的模型的扩展或增强。 每个领先的数据库系统今天都是至少一个这里介绍的模型的扩展或增强。

我们首先做出两个简化的假设（我们将在后续章节中放宽这些假设）：

1.操作系统线程支持：我们假设操作系统为我们提供了高效的内核线程支持，并且一个进程可以拥有非常多的线程。 我们还假设每个线程的内存开销很小，并且上下文切换的代价很低。 这在现代操作系统中可能是正确的，但在大多数数据库管理系统最初设计时并不成立。 由于某些平台上没有可用的操作系统线程或者操作系统线程的扩展性较差，许多数据库管理系统在实现时没有使用底层的操作系统线程支持。

2.单处理器硬件：我们将假设我们设计的是一台只有一个CPU的单机。考虑到多核系统的普及，即使在低端系统中，这也是一个不现实的假设。然而，这个假设将简化我们的初始讨论。

在这个简化的背景下，DBMS有三种自然的过程模型选择。 从最简单到最复杂，它们分别是：（1）每个DBMS工作进程，（2）每个DBMS工作线程，和（3）进程池。
尽管这些模型是简化的，但是商业DBMS系统今天仍然在使用这三种模型。

2.1.1 每个DBMS工作进程

每个*DBMS*工作进程模型（图2.1）在早期的*DBMS*实现中被使用，并且今天仍然被许多商业系统使用。 这个模型相对容易实现，因为*DBMS*工作进程直接映射到操作系统进程。 操作系统调度程序管理*DBMS*工作进程的时间共享，*DBMS*程序员可以依赖操作系统的保护机制来隔离标准错误，如内存溢出。 此外，各种编程工具，如调试器和内存检查器，非常适合这个进程模型。 复杂化这个模型的是跨*DBMS*连接共享的内存数据结构，包括锁表和缓冲池（在6.3节和5.3节中详细讨论）。 这些共享的数据结构必须在操作系统支持的共享内存中显式分配，可由所有*DBMS*进程访问。 这需要操作系统的支持（广泛可用）和一些特殊的*DBMS*编码。实际上，

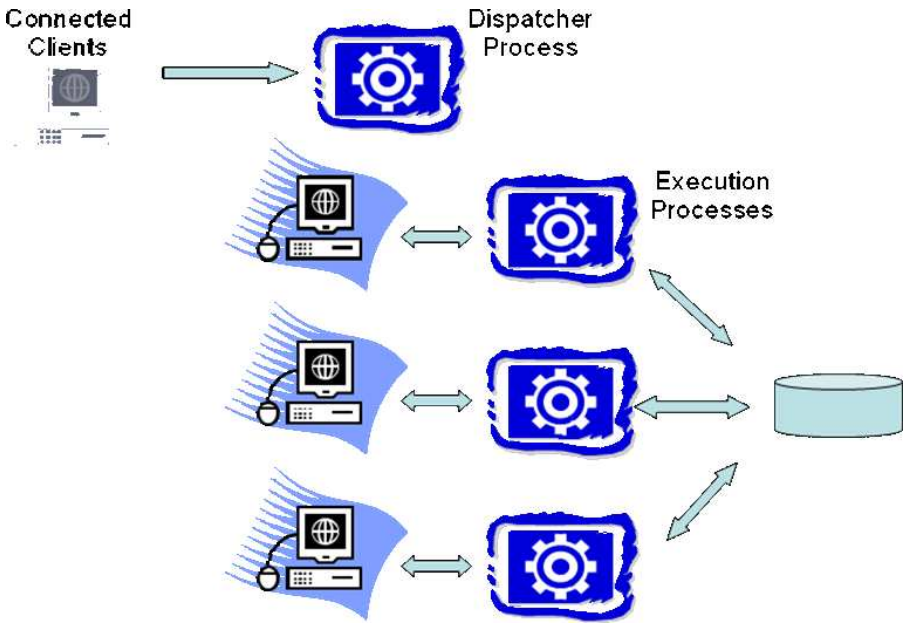


图2.1 每个DBMS工作模型的进程：每个DBMS工作进程都实现为一个操作系统进程。

在这个模型中，对共享内存的广泛使用减少了地址空间分离的一些优势，因为相当大一部分“有趣”的内存是跨进程共享的。

就并发连接数量的扩展性而言，每个*DBMS*工作进程模型并不是最有吸引力的进程模型。 扩展性问题是由于进程比线程具有更多的状态，因此消耗更多的内存。 进程切换需要切换安全上下文、内存管理器状态、文件和网络句柄表以及其他进程上下文。 线程切换不需要这样做。 尽管如此，每个*DBMS*工作进程模型仍然很受欢迎，并得到I*BM DB*2、Postgre*S*QL和Oracle的支持。

2.1.2 每个DBMS工作线程

在每个*DBMS*工作线程模型（图2.2）中，一个多线程进程承载所有的*DBMS*工作活动。一个调度器

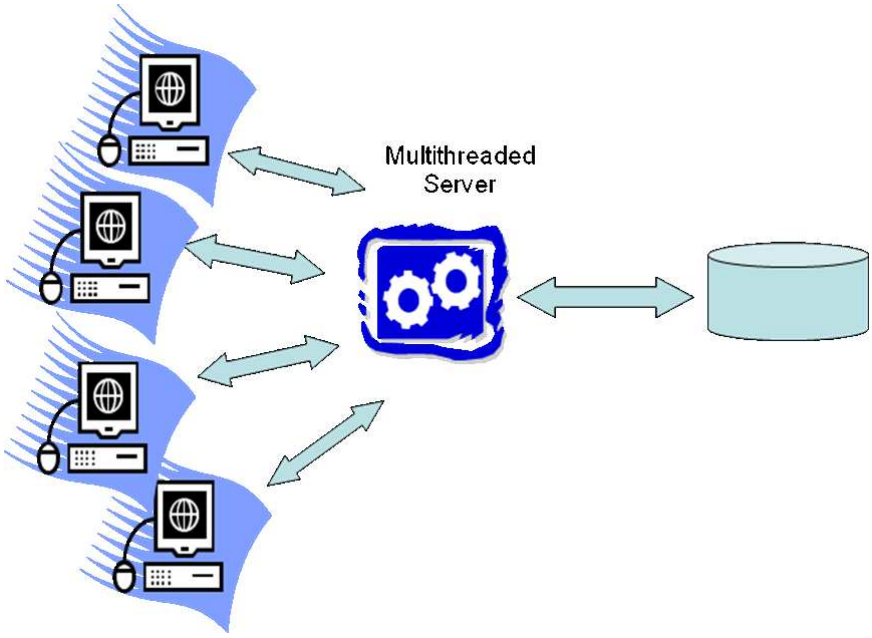


图2.2 每个DBMS工作线程模型：每个DBMS工作线程都是作为一个操作系统线程实现的。

线程（或少量这样的线程）监听新的DBMS客户端连接。 每个连接都分配一个新的线程。 每个客户端提交SQL请求时，该请求完全由其对应的线程执行DBMS工作。该线程在DBMS进程内运行，一旦完成，结果将返回给客户端，并且线程将在该相同客户端的连接上等待下一个请求。 在这种架构中，通常会出现多线程编程的挑战：操作系统无法保护线程免受彼此的内存溢出和错误指针；调试很棘手，尤其是在存在竞争条件时；由于线程接口和多线程扩展的差异，软件在不同操作系统之间的移植可能会很

由于大量使用共享内存，在每个*DB*MS工作线程模型中，许多多程序编程的挑战也存在于每个*DB*MS工作进程模型中。 由于大量使用共享内存，在每个*DB*MS工作线程模型中，许多多程序编程的挑战也存在于每个*DB*MS工作进程模型中。

尽管近年来，操作系统之间的线程API差异已经减小，但是平台之间的微妙差别仍然导致调试和优化方面的麻烦。 忽略这些实现上的困难，每个数据库管理系统工作线程模型在大量并发连接的情况下具有良好的扩展性，并且被一些当前生产的数据库管理系统使用，包括I*B M DB*2，*M*icrosoft SQL Server，*My*SQL，Informix和Sybase。

2.1.3 进程池

这个模型是每个数据库管理系统工作线程的一个变体。 回想一下，每个数据库管理系统工作线程的优势在于其实现的简单性。 但是，每个连接需要一个完整的进程的内存开销是一个明显的劣势。 通过进程池（图2.3），不再为每个数据库管理系统工作线程分配一个完整的进程，而是由一个进程池来托管它们。 一个中央进程持有所有的数据库管理系统客户端连接，并且当每个*S*QL请求从客户端进来时，请求会被分配给进程池中的一个进程。 *S*QL语句会执行到完成，结果会返回给数据库客户端，并且进程会返回到进程池中以便分配给下一个请求。

进程池的大小是有限的，通常是固定的。如果有一个请求进来

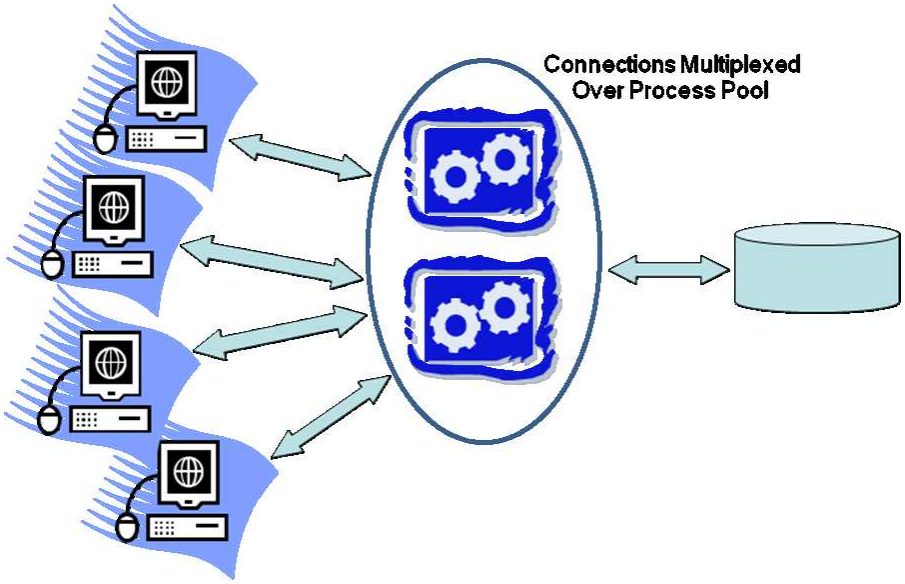


图2.3 进程池：每个数据库管理系统的工作进程被分配到一个操作系统进程池中当客户端发送工作请求时，进程从池中分配，并在请求处理完成后返回到池中。

如果所有进程都正在处理其他请求，新的请求
必须等待一个进程可用。

进程池具有每个数据库管理系统工作进程的所有优点但是，由于需要的进程数量较少，内存利用率更高。进程池通常使用动态可调整大小的进程池实现当大量并发请求到达时，进程池可能会增长到某个最大数量。 当请求负载较轻时，进程池可以减少等待进程的数量。 与每个数据库管理系统工作线程一样，进程池模型也被当前几代数据库管理系统所支持今天使用。

2.1.4 共享数据和进程边界

上述所有模型的目标都是尽可能独立地执行并发客户请求。 然而，完全的数据库管理系统工作人员的独立性和隔离性是不可能的，因为它们在同一个共享的数据库上操作。

数据库。 在每个数据库管理系统工作线程模型中，数据共享很容易，因为所有线程都在同一个地址空间中运行。 在其他模型中，共享内存用于共享数据结构和状态。 在这三种模型中，数据必须从数据库管理系统移动到客户端。 这意味着所有的SQL请求都需要移动到服务器进程中，并且所有返回给客户端的结果都需要移动出来。这是如何完成的？ 简短的答案是使用了各种缓冲区。

两种主要类型是磁盘I/O缓冲区和客户端通信缓冲区。 我们在这里描述这些缓冲区，并简要讨论管理它们的策略。

磁盘*I/O*缓冲区：最常见的跨工作器数据依赖关系是对共享数据存储的读取和写入。 因此，DBMS工作器之间的*I/O*交互很常见。 有两种独立的磁盘*I/O*场景需要考虑：（1）数据库请求和（2）日志请求。

- 数据库*I/O*请求：缓冲池。 所有持久化数据库数据都通过*DB*MS缓冲池（第5.3节）进行分阶段。对于每个*DB*MS工作器，缓冲池只是一个可供共享*DB*MS地址空间中的所有线程使用的堆内存数据结构。 在另外两个模型中，缓冲池分配在对所有进程可用的共享内存中。 在所有三个*DB*MS模型中，最终结果是缓冲池是一个大型的共享数据结构，可供所有数据库线程和/或进程使用。 当一个线程需要从数据库中读取一个页面时，它会生成一个*I/O*请求，指定磁盘地址，并提供一个空闲内存位置（帧），用于存放结果。 要将缓冲池页面刷新到磁盘上，线程会生成一个包含缓冲池中页面当前帧和目标磁盘地址的*I/O*请求。 缓冲池的详细讨论请参见第4.3节。

- 日志*I/O*请求：日志尾。 数据库日志（第6.4节）是存储在一个或多个磁盘上的条目数组。 在事务期间生成日志条目

处理过程中，它们被暂存在内存队列中，以FIFO顺序定期刷新到日志磁盘。
这个队列通常被称为日志尾。 在许多系统中，一个单独的进程或线程负责定期将日志尾刷新到磁盘。

使用每个*DBMS*工作线程，日志尾只是一个驻留在堆中的数据结构。 在另外两种模型中，常见的是两种不同的设计选择。 在一种方法中，一个单独的进程管理日志。日志记录通过共享内存或任何其他高效的进程间通信协议与日志管理器通信。 在另一种方法中，日志尾在共享内存中分配，方式与缓冲池处理方式类似。 关键点是，执行数据库客户端请求的所有线程和/或进程都需要能够请求写入日志记录和刷新日志尾。

一种重要的日志刷新类型是提交事务刷新。 直到提交日志记录被刷新到日志设备，事务才能被报告为成功提交。 这意味着客户端代码要等待直到提交日志记录被刷新，而DBMS服务器代码也必须在此期间持有所有资源（例如锁）。
日志刷新请求可能会被延迟一段时间，以允许将提交记录批量处理在一个I/O请求中（“组提交”）。

客户端通信缓冲区：SQL通常使用“拉取”模型：客户端通过重复发出SQL FETCH请求来从查询游标中获取结果元组，每个请求检索一个或多个元组。
大多数DBMS尝试在FETCH请求流之前工作，
预先排队客户端请求的结果。
为了支持这种预取行为，DBMS工作线程可以使用客户端通信套接字作为生成的元组的队列。 更复杂的方法实现了客户端端游标缓存，并使用DBMS客户端存储可能被获取的结果。

在不久的将来，而不是依赖于操作系统通信
缓冲区。

锁表：锁表由所有数据库管理系统工作者共享，并且由锁管理器（第6.3节）用于实现数据库锁定语义。 共享锁表的技术与缓冲池的技术相同，这些技术可以用于支持数据库管理系统实现所需的任何其他共享数据结构。

2.2 数据库管理系统线程

前一节提供了数据库管理系统进程模型的简化描述。 我们假设有高性能的操作系统线程，并且数据库管理系统仅针对单处理器系统。 在本节的其余部分，我们放宽了这些假设中的第一个，并描述了对数据库管理系统实现的影响。 多处理和并行性将在下一节中讨论。

2.2.1 数据库管理系统线程

今天的大多数数据库管理系统(DBMSs)都源自20世纪70年代的研究系统和20世纪80年代的商业化努力。 当最初的数据库系统构建时，我们今天认为理所当然的标准操作系统功能通常是不可用的。 高效、高规模的操作系统线程支持可能是其中最重要的特性之一。

直到20世纪90年代，操作系统线程才被广泛实现，并且在存在的情况下，实现方式差异很大。 即使在今天，一些操作系统线程的实现无法很好地支持所有数据库管理系统的工作负载[31, 48, 93, 94]。

因此，出于遗留、可移植性和可扩展性的原因，许多广泛使用的数据库管理系统(DBMS)在其实现中不依赖于操作系统线程。 有些系统完全避免使用线程，而是采用每个数据库管理系统工作进程或进程池模型。 选择剩余的进程模型之一，即每个数据库管理系统工作线程模型，需要解决那些没有良好内核线程实现的操作系统的问题。 一些领先的数据库管理系统采用了解决这个问题的方法

是为了实现自己专有的、轻量级的线程包。
这些轻量级线程，或者称为DBMS线程，取代了前一节中描述的操作系统线程的角色。 每个DBMS线程都被编程来管理自己的状态，通过非阻塞、异步接口执行所有可能的阻塞操作（例如I/O），并经常将控制权交给调度例程，在这些任务之间进行调度。

轻量级线程是一个古老的概念，在[49]中以回顾的方式进行了讨论，并广泛用于用户界面的事件循环编程。 这个概念在最近的操作系统文献中经常被重新讨论[31, 48, 93, 94]。 这种架构提供了快速的任务切换和易于移植性，但代价是在DBMS中复制了大量的操作系统逻辑（任务切换、线程状态管理、调度等）[86]。

2.3 标准实践

在当今领先的数据库管理系统中，我们发现了我们在2.1节中介绍的三种架构的代表，以及一些有趣的变体。 在这个维度上，IBM DB2可能是最有趣的例子，因为它支持四种不同的进程模型。 在具有良好线程支持的操作系统上，DB2默认为每个数据库管理系统工作线程，可选择支持线程池上的多路复用的数据库管理系统工作线程。

在没有可扩展线程支持的操作系统上运行时，DB2默认为每个数据库管理系统工作进程，并可选择支持进程池上的数据库管理系统工作进程的多路复用。

总结IBM DB2、MySQL、Oracle、PostgreSQL和Microsoft SQL Server支持的进程模型：

每个数据库管理系统工作进程：
这是最直接的进程模型，至今仍被广泛使用。 DB2在不支持高质量、可扩展操作系统线程的操作系统上默认为每个数据库管理系统工作进程，在支持的操作系统上默认为每个数据库管理系统工作线程。这也是Oracle的默认进程模型。 Oracle还支持下面描述的进程池作为可选模型。
PostgreSQL在所有支持的操作系统上独占地运行每个*DBMS*工作进程模型。

每个*DBMS*工作线程：这是一种高效的模型，目前有两个主要的变体在使用中：

1.每个*DBMS*工作的操作系统线程：I*BM DB*2在运行在具有良好操作系统线程支持的系统上时，默认使用此模型，*My*SQL也使用此模型。2.每个*DBMS*工作的*DBMS*线程：在这个模型中，*DBMS*工作线程由轻量级线程调度器在操作系统进程或操作系统线程上调度。 这个模型避免了潜在的操作系统调度器扩展或性能问题，但以高实现成本、差的开发工具支持和*DBMS*供应商的大量长期软件维护成本为代价。 这个模型有两个主要的子类别：

a.*DBMS*线程调度在操作系统进程上：一个或多个操作系统进程托管一个轻量级线程调度器。 *S*yba*se*和I*nfo*rmix支持这个模型。使用这个模型的当前一代系统实现了一个*DBMS*线程调度器，它在多个操作系统进程上调度*DBMS*工作线程以利用多个处理器。 然而，并不是所有使用这个模型的*DBMS*都实现了线程迁移：即将现有的*DBMS*线程重新分配给不同的操作系统进程（例如，用于负载平衡）。

b.*DBMS*线程在操作系统线程上调度：*Mi*cro*s*oft*S*QL *Ser*v*er*支持这种模型作为非默认选项（默认是*DBMS*工作线程在下面描述的线程池上复用）。 这个*S*QL *Server*选项，称为纤程，在一些高规模的事务处理基准测试中使用，但在其他情况下使用较少。

进程/线程池：
在这个模型中，DBMS工作线程在一个进程池上复用。
随着操作系统线程的支持改善，出现了这个模型的第二个变体。

基于线程池而不是进程池的模型已经出现。 在这个后一种模型中，DBMS工作线程在一个操作系统线程池上复用：

1.*DBMS*工作线程在一个进程池上复用：这个模型比每个*DBMS*工作线程一个进程更节省内存，易于在没有良好操作系统线程支持的操作系统上移植，并且可以很好地扩展到大量用户。 这是O*racle*支持的可选模型，也是他们推荐用于大量并发连接用户的系统。 O*racle*的默认模型是每个*DBMS*工作线程一个进程。 O*racle*支持的这两个选项都很容易在他们所针对的众多不同操作系统上支持（曾经O*racle*支持超过80个目标操作系统）。2.*DBMS*工作线程在一个线程池上复用：*Microsoft*SQL *Server*默认采用这个模型，超过99%的*S*QL *Server*安装都是这样运行的。正如上面提到的，为了高效地支持成千上万个并发连接用户，*S*QL *Server*还可选择*DBMS*线程调度在操作系统线程上。

正如我们在下一节中讨论的那样，大多数当前的商业DBMS支持查询内部并行性：即能够并行地在多个处理器上执行单个查询的全部或部分。 在本节中，我们讨论的目的是查询内部并行性是将多个DBMS工作进程临时分配给单个SQL查询。 这个特性对底层的进程模型没有任何影响，除了一个客户端连接可能有多个DBMS工作进程在其代表执行。

2.4 入场控制

我们在本节中结束时还有一个与支持多个并发请求相关的问题。 随着任何多用户系统的工作负载增加，吞吐量将增加到某个最大值。 超过这个点，随着系统开始抖动，吞吐量将急剧下降。

与操作系统一样，抖动通常是内存压力的结果：

DBMS无法将数据库页面的“工作集”保留在缓冲池中，并且花费所有时间替换页面。 在DBMS中，这尤其是在排序和哈希连接等查询处理技术中，往往会消耗大量的主内存。 在某些情况下，DBMS抖动也可能是由于锁争用引起的：事务不断地死锁，并需要回滚和重新启动。 因此，任何一个良好的多用户系统都有一个入场控制策略，除非有足够的DBMS资源可用，否则不接受新的工作。 通过一个良好的入场控制器，系统在超载情况下将显示出优雅的退化：事务延迟将与到达率成比例增加，但吞吐量将保持在峰值。

DBMS的准入控制可以分为两个层次。 首先，在调度程序过程中可以使用简单的准入控制策略，以确保客户端连接数保持在阈值以下。

这样可以防止基本资源（如网络连接）的过度消耗。 在某些DBMS中，不提供此控制，假设由多层系统的另一层处理，例如应用服务器、事务处理监视器或Web服务器。

准入控制的第二层必须直接在核心DBMS关系查询处理器中实现。 这个执行准入控制器在查询被解析和优化之后运行，并确定查询是否被延迟、以较少的资源开始执行，或者以无额外约束开始执行。 执行准入控制器通过查询优化器提供的信息来辅助，该优化器估计查询所需的资源和系统资源的当前可用性。 特别是，优化器的查询计划可以指定（1）查询将访问的磁盘设备以及每个设备的随机和顺序I/O数量的估计，（2）基于查询计划中的运算符和要处理的元组数量的查询的CPU负载估计，以及最重要的是（3）关于查询数据结构的内存占用的估计，包括在连接和其他查询执行任务期间对大型输入进行排序和哈希的空间。 如上所述，这个最后的度量通常是准入控制器的关键，因为内存压力通常是抖动的主要原因。因此

许多数据库管理系统使用内存占用和活动数据库管理系统的数量作为入场控制的主要标准。

2.5 讨论和附加材料

进程模型的选择对数据库管理系统的扩展性和可移植性有重要影响。因此，三个广泛使用的商业系统在其产品线上都支持多个进程模型。从工程角度来看，在所有操作系统和所有规模的情况下使用单一进程模型显然更简单。但是，由于使用模式的广泛多样性和目标操作系统的不均匀性，这三个数据库管理系统选择支持多个模型。

展望未来，近年来对服务器系统的新进程模型产生了重大兴趣，这是由硬件瓶颈的变化和互联网工作负载的规模和变异性所驱动的 [31, 48, 93, 94]。这些设计中出现的一个主题是将服务器系统分解为一组独立调度的“引擎”，这些引擎之间以异步和批量方式传递消息。这有点像上面的“进程池”模型，其中工作单元在多个请求之间被重复使用。这项最新研究的主要创新是以比以前更狭义的任务特定方式来分解工作的功能颗粒。这导致了工作者和SQL请求之间的多对多关系 - 单个查询通过多个工作者的活动来处理，并且每个工作者为多个SQL请求执行其自己的专门任务。这种架构可以实现更灵活的调度选择 - 例如，它允许在允许单个工作者完成多个查询的任务之间进行动态权衡（可能提高整个系统的吞吐量），或者允许查询在多个工作者之间取得进展（以提高该查询的延迟）。在某些情况下，这已经被证明在处理器缓存局部性和在硬件缓存未命中期间保持CPU忙碌方面具有优势。

在DBMS环境中进一步研究这个想法通常以StagedDB研究项目[35]为典型，这是一个很好的额外阅读起点。

3

并行架构：进程和内存协调

并行硬件是现代服务器中不可或缺的一部分，有各种不同的配置。 在本节中，我们总结了标准的DBMS术语（在[87]中介绍），并讨论了每个模型中的进程模型和内存协调问题。

3.1 共享内存

共享内存并行系统（图3.1）是指所有处理器都可以以大致相同的性能访问相同的RAM和磁盘。 这种架构在今天相当标准 - 大多数服务器硬件配备了两到八个处理器。 高端机器可以配备数十个处理器，但相对于提供的处理资源，它们往往以较高的溢价出售。 高度并行的共享内存机器是硬件行业中仅存的“摇钱树”之一，并且在高端在线事务处理应用中被广泛使用。 服务器硬件的成本通常被系统管理成本所掩盖，因此成本的

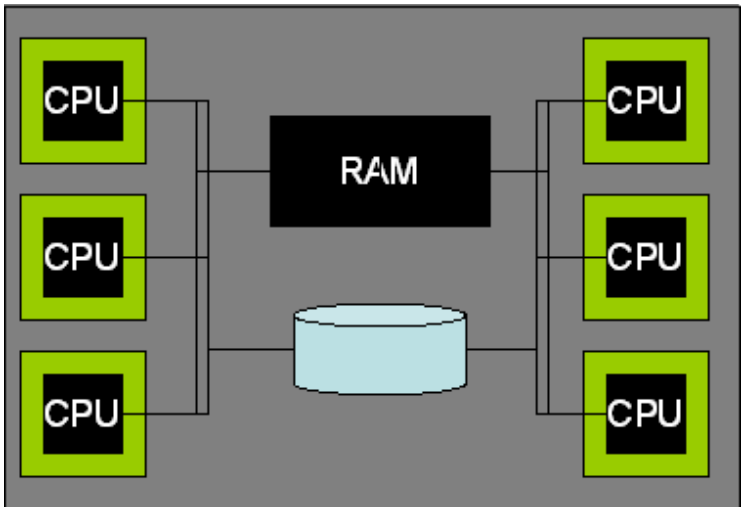


图3.1 共享内存架构。

购买少量大型、非常昂贵的系统有时被视为可以接受的权衡。[1]

多核处理器在单个芯片上支持多个处理核心，并共享一些基础设施，如缓存和内存总线。从编程模型上看，它们与共享内存架构非常相似。 如今，几乎所有严肃的数据库部署都涉及多个处理器，每个处理器都有多个CPU。DBMS架构需要能够充分利用这种潜在的并行性。幸运的是，第2节中描述的三种DBMS架构在现代共享内存硬件架构上运行良好。

共享内存机器的进程模型与单处理器方法非常相似。 事实上，大多数数据库系统从最初的单处理器实现逐渐演变为共享内存实现。 在共享内存机器上，操作系统通常支持对工作进程（进程或

[1]对于数据库管理系统（DBMS）的客户来说，主要成本通常是支付合格的人员来管理高端系统。 这包括配置和维护DBMS的数据库管理员（DBA）以及配置和维护硬件和操作系统的系统管理员。

线程）跨处理器，并且共享的数据结构仍然可以被所有人访问。这三种模型在这些系统上运行良好，并支持并行执行多个独立的SQL请求。主要挑战是修改查询执行层以利用在多个CPU上并行化单个查询的能力；我们将此推迟到第5节。

3.2 共享无

共享无并行系统（图3.2）由一组独立的机器组成，它们通过高速网络互连或者越来越频繁地通过商品化网络组件进行通信。 给定系统无法直接访问另一个系统的内存或磁盘。

共享无系统不提供硬件共享抽象，将各个机器的协调完全交给DBMS处理。DBMS用于支持这些集群的最常见技术是在集群中的每台机器或节点上运行它们的标准进程模型。 每个节点都能够接受客户端SQL

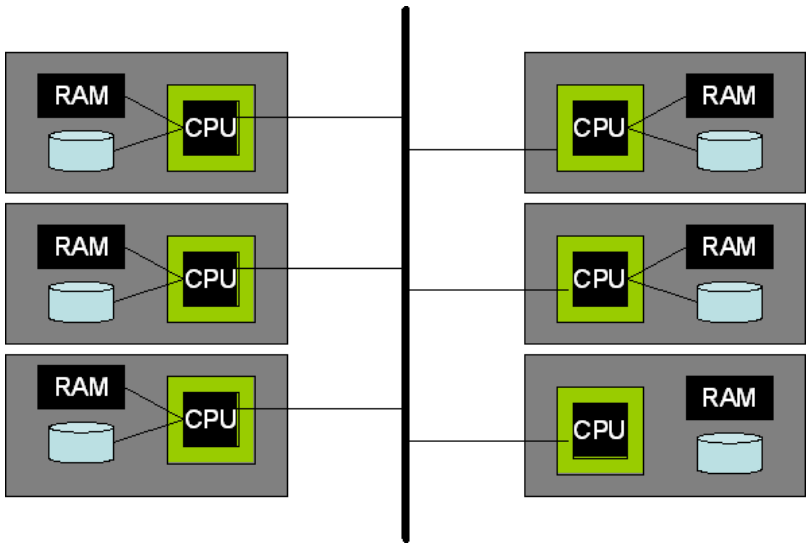


图3.2 共享无架构

请求，访问必要的元数据，编译SQL请求，并像在单个共享内存系统上一样执行数据访问。 主要区别在于集群中的每个系统仅存储部分数据。 与仅针对本地数据运行接收到的查询不同，请求被发送到集群的其他成员，并且所有涉及的机器并行执行查询以针对它们存储的数据。 表在集群中的多个系统上分布，使用水平数据分区，以使每个处理器能够独立执行。

数据库中的每个元组都分配给一个单独的机器，因此每个表都被“水平”切片并分布在机器上。 典型的数据分区方案包括基于哈希的元组属性分区，基于范围的元组属性分区，轮询和混合分区，后者是基于范围和哈希的组合。 每个单独的机器负责访问、锁定和记录其本地磁盘上的数据。 在查询执行期间，查询优化器选择如何在机器之间水平重新分区表和中间结果以满足查询，并为每个机器分配工作的逻辑分区。 各个机器上的查询执行器将数据请求和元组发送给彼此，但不需要传输任何线程状态或其他低级信息。 由于数据库元组的基于值的分区，这些系统需要最少的协调。 然而，为了获得良好的性能，需要对数据进行良好的分区。 这给数据库管理员（DBA）带来了重要的负担，需要智能地布局表，以及查询优化器需要做好工作来分区工作负载。

这种简单的分区解决方案不能解决DBMS中的所有问题。例如，必须进行显式的跨处理器协调来处理事务完成、提供负载平衡和支持某些维护任务。 例如，处理器必须交换显式的控制消息来处理分布式死锁检测和两阶段提交[30]等问题。 这需要额外的逻辑，并且如果不小心处理，可能成为性能瓶颈。

此外，部分故障是一个需要在共享无关系统中管理的可能性。 在共享内存系统中，一个节点的故障

处理器通常会导致整个机器关机，因此整个数据库管理系统（DBMS）也会关机。在共享无关系统中，单个节点的故障不一定会影响集群中的其他节点。但是它肯定会影响数据库管理系统的整体行为，因为故障节点托管了数据库中的一部分数据。在这种情况下，至少有三种可能的方法。第一种方法是如果任何一个节点失败，则关闭所有节点；这本质上模拟了共享内存系统中的情况。I*n*formix称之为“数据跳过”的第二种方法允许在任何正常运行的节点上执行查询，跳过故障节点上的数据。这在数据可用性比结果完整性更重要的情况下非常有用。但是，尽力而为的结果没有明确定义的语义，并且对于许多工作负载来说，这不是一个有用的选择——特别是因为数据库管理系统通常用作多层系统中的“记录库”，可用性与一致性之间的权衡往往在更高的层次（通常在应用服务器中）完成。第三种方法是采用从完全数据库故障转移（需要双倍的机器和软件许可证）到细粒度冗余（如链式分布）的冗余方案。在后一种技术中，元组副本分布在集群中的多个节点上。链式分布比简单方案具有以下优点：（a）它所需的部署机器比朴素方案少，以确保可用性；（b）当一个节点失败时，系统负载会相对均匀地分布在剩余节点上：剩余的n-1个节点每个执行原始工作的$n/(n\text{-}1)$部分，这种性能的线性退化会随着节点的故障而继续。实际上，大多数当前的商业系统介于两者之间，既不像完全数据库冗余那样粗粒度，也不像链式分布那样细粒度。

共享无关架构在今天相当常见，并具有无与伦比的可扩展性和成本特性。它主要用于极高端，通常用于决策支持应用和数据仓库。在硬件架构的有趣组合中，共享无关集群通常由许多节点组成，每个节点都是共享内存多处理器。

3.3 共享磁盘

共享磁盘并行系统（图3.3）是一种所有处理器都可以以大致相同的性能访问磁盘，但无法访问彼此的RAM的系统。这种架构在最近几年变得越来越常见，其中两个著名的例子是Oracle RAC和DB2 for zSeries SYS-PLEX。随着存储区域网络（SAN）的日益流行，共享磁盘变得更加普遍。 SAN允许一个或多个逻辑磁盘被一个或多个主机系统挂载，从而轻松创建共享磁盘配置。

共享磁盘系统相对于共享无系统的一个潜在优势是其更低的管理成本。 共享磁盘系统的数据库管理员不需要考虑将表分区到不同机器上以实现并行处理。 但是非常大的数据库通常仍然需要进行分区，因此在这个规模上，差异变得不那么明显。

共享磁盘架构的另一个吸引人的特点是单个数据库管理系统处理节点的故障不会影响其他节点访问整个数据库的能力。 这与共享内存系统完全不同，共享内存系统会作为一个整体失败，而共享无系统在节点故障时将无法访问至少一些数据（除非使用其他数据冗余方案）。 然而，即使具有这些优势，共享磁盘系统仍然容易受到某些单点故障的影响。

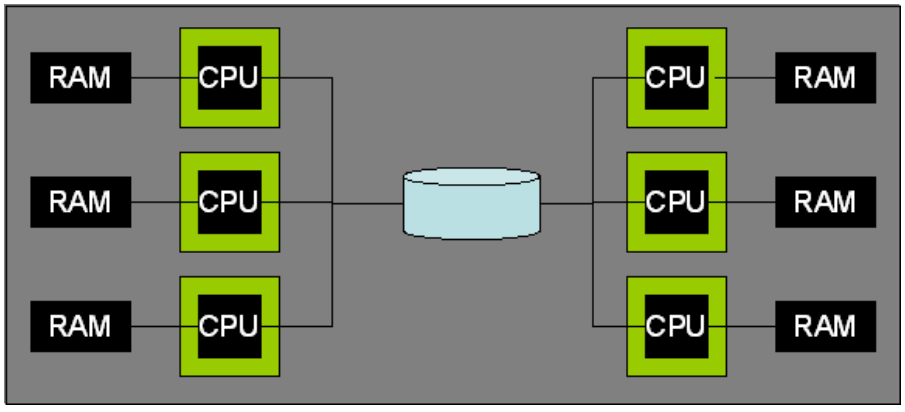


图3.3 共享磁盘架构。

故障点。 如果数据在到达存储子系统之前由硬件或软件故障损坏或其他方式损坏，那么系统中的所有节点只能访问到这个损坏的页面。 如果存储子系统正在使用RAID或其他数据冗余技术，那么这个损坏的页面将被冗余存储，但在所有副本中仍然是损坏的。

因为在共享磁盘系统中不需要对数据进行分区，数据可以被复制到内存中，并在多台机器上进行修改。
与共享内存系统不同，没有自然的内存位置来协调数据的共享 - 每台机器都有自己的本地内存用于锁和缓冲池页面。 因此，需要对机器之间的数据共享进行显式协调。 共享磁盘系统依赖于分布式锁管理器和用于管理分布式缓冲池的一致性缓存协议[8]。 这些都是复杂的软件组件，并且可能成为具有显著争用的工作负载的瓶颈。 一些系统，如IBM zSeries SYSPLEX，在硬件子系统中实现了锁管理器。

3.4 NUMA

非一致性内存访问（*NUMA*）系统提供了一个在具有独立内存的系统集群上的共享内存编程模型。 集群中的每个系统可以快速访问自己的本地内存，而通过高速集群互连进行的远程内存访问则稍有延迟。 这个架构名称来自于内存访问时间的非一致性。

NUMA硬件架构是共享无关和共享内存系统之间的一个有趣的中间地带。 它们比共享无关集群更容易编程，并且通过避免共享内存总线等共享争用点，也可以扩展到更多处理器。

NUMA集群在商业上并不广泛成功，但NUMA设计概念已经在共享内存多处理器中得到采用（第3.1节）。随着共享内存多处理器扩展到更多处理器，它们的内存架构显示出越来越不均匀。

通常，大型共享内存多处理器的内存被划分为几个部分，并且每个部分与系统中的一小部分处理器相关联。 每个组合的内存和CPU子集通常被称为一个pod。每个处理器可以比远程pod内存更快地访问本地pod内存。 NUMA设计模式的使用使得共享内存系统可以扩展到非常大的处理器数量。 因此，NUMA共享内存多处理器现在非常常见，而NUMA集群从未获得任何重要的市场份额。

DBMS可以在NUMA共享内存系统上运行的一种方式是忽略内存访问的非均匀性。 只要非均匀性不太严重，这种方法是可行的。 当近内存和远内存访问时间的比例超过1.5:1到2:1的范围时，DBMS需要采取优化措施以避免严重的内存访问瓶颈。 这些优化措施有多种形式，但都遵循相同的基本方法：(a)在为处理器分配内存时，使用与该处理器本地的内存（避免使用远内存），(b)确保给定的DBMS工作进程始终在同一硬件处理器上进行调度，如果可能的话。

这种组合使得DBMS工作负载能够在具有一定内存访问非均匀性的高规模共享内存系统上良好运行。

尽管NUMA集群已经几乎消失，但编程模型和优化技术对当前一代DBMS系统仍然非常重要，因为许多高规模共享内存系统的内存访问性能存在显著的非均匀性。

3.5 DBMS线程和多处理器

实现每个*DBMS*工作线程的潜在问题在于，当我们从第2.1节中删除我们的两个简化假设中的最后一个假设，即单处理器硬件时，问题立即显现。 在第2.2.1节中描述的轻量级*DBMS*线程包的自然实现是所有线程在一个单一的操作系统进程中运行。不幸的是，

单个进程一次只能在一个处理器上执行。 因此，在多处理器系统上，DBMS一次只能使用一个处理器，并使系统的其余部分空闲。

早期的Sybase SQL Server架构受到了这个限制的影响。 随着共享内存多处理器在90年代初变得更加流行，Sybase迅速进行了架构上的改变，以利用多个操作系统进程。

在多个进程中运行DBMS线程时，有时一个进程承担大部分工作，而其他进程（因此处理器）处于空闲状态。 为了在这种情况下使该模型良好运行，DBMS必须在进程之间实现线程迁移。 Informix在6.0版本发布时做得非常出色。

当将DBMS线程映射到多个操作系统进程时，需要做出关于使用多少个操作系统进程、如何将DBMS线程分配给操作系统线程以及如何在多个操作系统进程之间分配的决策。 一个好的经验法则是每个物理处理器有一个进程。 这样可以最大限度地发挥硬件中固有的物理并行性，同时最小化每个进程的内存开销。

3.6 标准实践

关于并行性的支持，趋势与上一节类似：大多数主要的DBMS都支持多种并行模式。 由于共享内存系统（SMP、多核系统和两者的组合）在商业上的普及，所有主要的DBMS供应商都对共享内存并行性提供了良好的支持。

我们开始看到支持的分歧是在多节点集群并行性方面，广泛的设计选择是共享磁盘和共享无。

- 共享内存：所有主要的商业DB*MS*提供商都支持共享内存并行性，包括：IB*M* DB2、O*racle*和*M*ic*ro*s*o*ft *S*QL *Ser*v*er*。
- 共享无关：这个模型由IBM DB2、I*nfor*m*ix*、T*ande*m和*N*C R T*eradata*等公司支持；绿色-

plum提供了一个定制版本的PostgreSQL，支持共享无关并行处理。

- 共享磁盘：这个模型由Oracle RAC、RDB（由Oracle从Digital Equipment Corp.收购）和IBM DB2 for z*Series*等公司支持。

IBM销售多种不同的DBMS产品，并选择在其中一些产品中实现共享磁盘支持，而在其他产品中实现共享无关支持。 到目前为止，没有任何一家领先的商业系统在单一代码库中同时支持共享无关和共享磁盘；Microsoft SQL Server两者都没有实现。

3.7 讨论和附加材料

上述设计代表了各种服务器系统中使用的硬件/软件架构模型的选择。 虽然它们在DBMS中首创，但这些想法在其他数据密集型领域中越来越流行，包括像Map-Reduce [12]这样的低级可编程数据处理后端，它们越来越多地用于各种定制数据分析任务。 然而，尽管这些想法正在广泛影响计算领域，但在数据库系统的并行设计中仍然出现了新的问题。

在接下来的十年中，并行软件架构面临的一个关键挑战是利用来自处理器供应商的新一代“多核”架构。 这些设备将引入一个新的硬件设计点，单个芯片上有数十个、数百个甚至数千个处理单元，通过高速芯片内网络进行通信，但在访问芯片外存储器和磁盘方面仍然存在许多现有的瓶颈。

这将导致内存路径在磁盘和处理器之间出现新的不平衡和瓶颈，几乎肯定需要重新审视数据库管理系统的架构，以满足硬件的性能潜力。

在更“宏观”的层面上，服务导向计算领域正在预见到一种相关的架构转变。 在这里，大型数据中心将托管数万台计算机的处理（硬件和软件）给用户。在这个规模上，应用程序-

只有高度自动化才能负担得起数据库管理和服务器管理。
没有任何管理任务能够随着服务器数量的增加而扩展。 而且，由于通常使用不太可靠的廉价服务器，并且故障更加常见，因此需要完全自动化地从常见故障中恢复。 在大规模服务中，每天都会发生磁盘故障，每周还会发生多次服务器故障。 在这种环境下，通常会用不同服务器上存储在不同磁盘上的整个数据库的冗余在线副本来替代管理数据库备份。 根据数据的价值，冗余副本甚至可以存储在不同的数据中心。 仍然可以使用自动离线备份来恢复应用程序、管理或用户错误。 然而，对于大多数常见错误和故障，快速故障转移到冗余的在线副本是一种快速恢复的方法。 可以通过多种方式实现冗余：(a) 在数据存储层面进行复制（存储区域网络），(b) 在数据库存储引擎层面进行数据复制（如第7.4节所讨论的），(c) 查询处理器通过冗余执行查询（第6节），或者 (d) 客户端软件层面自动生成冗余数据库请求（例如，由Web服务器或应用服务器生成）。

在更加解耦的层面上，实际上很常见的是将具有DBMS功能的多个服务器部署在层次结构中，以尽量减少对“记录型DBMS”的I/O请求速率。这些方案包括各种形式的中间层数据库缓存用于SQL查询，包括专用的主内存数据库，如Oracle TimesTen，以及更传统的数据库配置以满足此目的（例如，[55]）。 在部署堆栈的较高层次上，许多面向对象的应用服务器架构，支持企业级Java Bean等编程模型，可以与DBMS一起配置以进行应用对象的事务性缓存。然而，这些各种方案的选择、设置和管理仍然是非标准和复杂的，优雅且普遍认可的模型仍然难以实现。

4

关系查询处理器

前面的章节强调了DBMS中的宏观架构设计问题。现在我们开始一系列的章节，讨论稍微细粒度的设计，依次解决主要的DBMS组件。 根据我们在第1.1节中的讨论，我们从系统的顶部开始，即查询处理器，并在随后的章节中进入存储管理、事务和实用程序。

一个关系查询处理器接受一个声明性的SQL语句，验证它，并将其优化为一个过程化的数据流执行计划，然后（在接受控制的情况下）代表客户程序执行该数据流程序。 然后客户程序获取（“拉取”）结果元组，通常一次获取一个或者一小批。 关系查询处理器的主要组件如图1.1所示。在本节中，我们关注查询处理器和存储管理器访问方法的一些非事务性方面。 一般来说，关系查询处理可以被视为单用户、单线程的任务。 并发控制由系统的较低层透明地管理，如第5节所述。

唯一的例外是当DBMS必须显式地在操作缓冲池页面时“固定”和“解固”它们，以便

在我们讨论第4.4.5节时，它们在内存中保留在短暂的、关键的操作期间。

在本节中，我们重点关注常见的SQL命令：数据操作语言（DML）语句，包括SELECT、INSERT、UPDATE和DELETE。数据定义语言（DDL）语句，如CREATE TABLE和CREATE INDEX通常不会由查询优化器处理。这些语句通常通过对存储引擎和目录管理器（在第6.1节中描述）的显式调用在静态DBMS逻辑中进行实现。一些产品也开始优化DDL的一个小子集，我们预计这个趋势将继续下去。

4.1 查询解析和授权

给定一个SQL语句，SQL解析器的主要任务是（1）检查查询是否正确指定，（2）解析名称和引用，（3）将查询转换为优化器使用的内部格式，以及（4）验证用户是否有权执行查询。一些DBMS将一些或全部安全检查推迟到执行时，但即使在这些系统中，解析器仍负责收集执行时安全检查所需的数据。

给定一个SQL查询，解析器首先考虑FROM子句中的每个表引用。它将表名规范化为完全限定的形式，即server.database.schema.table。这也被称为四部分名称。不支持跨多个服务器的查询的系统只需要将其规范化为database.schema.table，而仅支持每个DBMS一个数据库的系统可以将其规范化为schema.table。这种规范化是必需的，因为用户具有上下文相关的默认值，允许在查询规范中使用单个部分名称。一些系统支持对表的多个名称，称为表别名，并且这些别名也必须替换为完全限定的表名。

在规范化表名之后，查询处理器会调用目录管理器来检查表是否在系统目录中注册。在此步骤中，它还可以在内部查询数据结构中缓存有关表的元数据。根据关于的信息

在查询执行之前，它使用目录来确保属性引用是正确的。 属性的数据类型用于驱动对重载函数表达式、比较运算符和常量表达式的消歧逻辑。 例如，考虑表达式(EMP.salary * 1.15) < 75000。 乘法函数和比较运算符的代码，以及字符串“1.15”和“75000”的假定数据类型和内部格式，将取决于EMP.salary属性的数据类型。 这个数据类型可以是整数、浮点数或“货币”值。 还会应用其他标准的SQL语法检查，包括元组变量的一致使用，通过集合运算符（UNION/INTERSECT/EXCEPT）组合的表的兼容性，聚合查询的SELECT列表中属性的使用，子查询的嵌套等等。

如果查询成功解析，下一阶段是授权检查，以确保用户对查询中引用的表、用户定义的函数或其他对象具有适当的权限（SELECT/DELETE/INSERT/UPDATE）。 一些系统在语句解析阶段执行完整的授权检查。

然而，这并不总是可能的。 支持行级安全性的系统，例如，直到执行时间才能进行完整的安全性检查，因为安全性检查可能依赖于数据值。 即使在理论上授权可以在编译时静态验证，将部分工作推迟到查询计划执行时间也有优势。 将安全性检查推迟到执行时间的查询计划可以在用户之间共享，并且不需要在安全性更改时重新编译。 因此，安全性验证的某些部分通常推迟到查询计划执行。

在编译时可以对常量表达式进行约束检查。 例如，一个UPDATE命令可能有一个形如SET EMP.salary = -1的子句。 如果完整性约束规定工资必须为正值，那么查询甚至不需要执行。 然而，将这项工作推迟到执行时间是非常常见的。

如果查询解析并通过验证，则将查询的内部格式传递给查询重写模块进行进一步处理。

4.2 查询重写

查询重写模块负责简化和规范化查询，而不改变其语义。它只能依赖于查询和目录中的元数据，并且无法访问表中的数据。尽管我们谈论“重写”查询，但大多数重写器实际上是在查询的内部表示上操作，而不是在原始的SQL语句文本上操作。查询重写模块通常以与其输入相同的内部格式输出查询的内部表示。

许多商业系统中的重写器是一个逻辑组件，其实际实现要么在查询解析的后期阶段，要么在查询优化的早期阶段。例如，在DB2中，重写器是一个独立的组件，而在SQL Server中，查询重写是查询优化器的早期阶段。尽管如此，即使在所有系统中都不存在显式的架构边界，单独考虑重写器仍然是有用的。

重写器的主要责任是：

- 视图展开：处理视图是重写器的主要传统角色。对于出现在FROM子句中的每个视图引用，重写器从目录管理器中检索视图定义。然后，它重写查询以(1)用视图引用的表和谓词替换该视图，并且(2)用视图中的表的列引用替换对该视图的任何引用。这个过程递归地应用，直到查询完全表示为表上的表达式，并且不包含任何视图。这种视图展开技术最初是为INGRES中的基于集合的QUEL语言提出的[85]，在SQL中需要一些注意事项，以正确处理重复消除、嵌套查询、NULL和其他棘手的细节[68]。

- 常量算术计算：查询重写可以简化常量算术表达式，例如，`R.x < 10+2+R.y`被重写为`R.x < 12+R.y`。

- 谓词的逻辑重写：基于WHERE子句中的谓词和常量进行逻辑重写。

简单的布尔逻辑通常用于改善表达式与基于索引的访问方法的匹配。 例如，谓词NOT Emp.Salary >1000000可以被重写为Emp.Salary <=1000000。 这些逻辑重写甚至可以通过简单的可满足性测试来短路查询执行。 例如，表达式Emp.salary < 75000 AND Emp.salary > 1000000可以被替换为FALSE。这可能允许系统在不访问数据库的情况下返回一个空的查询结果。 不可满足的查询可能看起来不太可能，但请记住，谓词可能被“隐藏”在视图定义中，对于外部查询的编写者来说是未知的。 例如，上面的查询可能是从一个名为“Executives”的视图上的一个查询中得到的。不可满足的谓词也是Microsoft SQL Server并行安装中“分区消除”的基础：当一个关系通过范围谓词在磁盘卷上进行水平范围分区时，如果其范围分区谓词与查询谓词共同不可满足，则查询不需要在该卷上运行。

另外，一个重要的逻辑重写使用谓词的传递性来引出新的谓词R.x < 10 ANDR.x = S.y，例如，建议添加额外的谓词“AND S.y < 10。” 添加这些传递性谓词增加了优化器在执行早期选择计划时过滤数据的能力，特别是通过使用基于索引的访问方法。

- 语义优化：在许多情况下，模式上的完整性约束存储在目录中，并且可以用于帮助重写一些查询。 一个重要的优化示例是冗余连接消除。 当外键约束将一个表的列（例如，E*mp*.d*eptno*）绑定到另一个表（D*ept*）时，就会出现这种情况。 给定这样一个外键约束，已知每个E*mp*都有一个对应的D*ept*，而且E*mp*元组在没有对应的D*ept*元组（父元组）的情况下是不存在的。

考虑一个连接两个表但不使用Dept列的查询:

```
SELECT Emp.name, Emp.salary
  FROM Emp, Dept
  WHERE Emp.deptno = Dept.dno
```

这样的查询可以重写以删除Dept表(假设Emp.deptno被约束为非空)，从而删除连接。 同样，这样看似不可能的情况经常通过视图自然地出现。 例如，用户可能提交一个关于员工属性的查询，该查询在一个连接这两个表的视图EMPDEPT上进行。 诸如Siebel之类的数据库应用使用非常宽的表，如果底层数据库不支持足够宽的表，则使用多个表并在这些表上创建视图。

如果没有冗余连接消除，基于视图的宽表实现性能会非常差。

当表的约束与查询谓词不兼容时，语义优化也可以完全绕过查询执行。

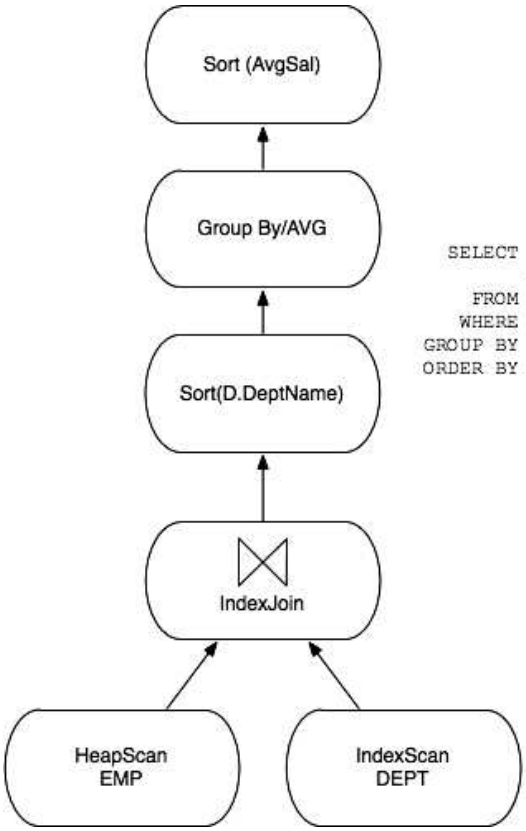
- 子查询展开和其他启发式重写：查询优化器是当前商业数据库管理系统中最复杂的组件之一。为了保持复杂性有限，大多数优化器在单独的*S*ELECT-FROM-WHERE查询块上操作，并且不跨块进行优化。 因此，许多系统将查询重写为更适合优化器的形式，而不是进一步复杂化查询优化器。 这种转换有时被称为查询规范化。 一种常见的规范化类别是将语义等价的查询重写为规范形式，以确保语义等价的查询将被优化为生成相同的查询计划。 另一个重要的启发式是在可能的情况下展开嵌套查询，以最大程度地暴露查询优化器的单块优化机会。 这在某些情况下在*S*QL中非常棘手，由于存在问题

像重复语义、子查询、NULL和关联性 [68, 80]。 在早期，子查询展开是一种纯启发式的重写方法，但现在一些产品已经开始基于成本分析来决定重写策略。 查询块之间还可以进行其他重写。 例如，谓词传递性可以允许谓词在子查询之间进行复制 [52]。 展开相关子查询对于在并行架构中实现良好性能尤为重要：相关子查询会导致查询块之间的“嵌套循环”式比较，这会使子查询的执行串行化，尽管并行资源是可用的。

4.3 查询优化器

查询优化器的任务是将内部查询表示转换为执行查询的高效查询计划（图4.1）。查询计划可以被视为将表数据通过查询操作符图进行传输的数据流图。 在许多系统中，查询首先被分解为SELECT-FROM-WHERE查询块。 然后，使用类似于Selinger等人在System R优化器上描述的技术来优化每个单独的查询块。 完成后，通常会在每个查询块的顶部添加一些操作符作为后处理，用于计算GROUP BY、ORDER BY、HAVING和DISTINCT子句（如果存在）。 然后，这些不同的块会以直接的方式连接在一起。

生成的查询计划可以用多种方式表示。
最初的System R原型将查询计划编译成机器代码，而早期的INGRES原型生成了可解释的查询计划。 查询解释在INGRES作者的回顾性论文中被列为一个“错误”，但是摩尔定律和软件工程在某种程度上证明了INGRES的决策是正确的。 具有讽刺意味的是，将编译成机器代码被System R项目的一些研究人员列为一个错误。 当System R代码库被转化为商业DBMS系统时




```
SELECT D.DeptName,
       AVG(E.Salary) AS AvgSal
  FROM EMP E, DEPT D
 WHERE E.Dno = D.DeptID
GROUP BY DeptName
ORDER BY AvgSal DESC
```

图4.1 查询计划。 只显示主要的物理运算符。

（SQL/DS）开发团队的第一个改变是用解释器替换机器代码执行器。

为了实现跨平台可移植性，现在每个主要的DBMS都将查询编译成某种可解释的数据结构。 它们之间唯一的区别是中间形式的抽象级别。 某些系统中的查询计划是一个非常轻量级的对象，类似于关系代数表达式，带有访问方法、连接算法等的注释。其他系统使用更低级别的语言，类似于Java字节码而不是关系代数表达式。 为了简化我们的讨论，我们在本文的剩余部分将重点放在类似于代数的查询表示上。

尽管Selinger的论文被广泛认为是查询优化的“圣经”，但它只是初步研究。 所有系统在许多方面都对这项工作进行了重要扩展。 主要的扩展包括：

- 计划空间：System R优化器通过仅关注“左深”查询计划（其中连接的右侧输入必须是基本表）并“推迟笛卡尔积”（确保笛卡尔积仅在数据流中的所有连接之后出现）来限制其计划空间。在今天的商业系统中，众所周知，“丛状”树（带有嵌套的右侧输入）和早期使用笛卡尔积在某些情况下是有用的。 因此，大多数系统在某些情况下都考虑了这两个选项。

- 选择性估计：*Selinger*论文中的选择性估计技术基于简单的表和索引基数，并且根据当前一代系统的标准来说是天真的。今天的大多数系统通过直方图和其他摘要统计信息分析和总结属性中的值分布。 由于这涉及访问每个列中的每个值，因此可能相对昂贵。 因此，一些系统使用抽样技术来估计分布，而无需进行详尽的扫描。

  通过在连接列上“连接”直方图，可以对基本表的连接性估计进行计算。
  为了超越单列直方图，最近提出了更复杂的方案，以解决列之间的依赖等问题[16, 69]。 这些创新已经开始出现在商业产品中，但仍然需要取得相当大的进展。 这些方案被缓慢采用的一个原因是许多行业基准测试中存在长期的缺陷：例如，TPC-D和TPC-H中的数据生成器在列中生成统计独立的值，因此并不鼓励采用处理“真实”数据分布的技术。

  TPC-DS已经解决了这个基准测试的缺陷。

基准测试[70]。 尽管采用率较慢，但改进的选择性估计的好处被广泛认可。 Ioan-nidis和Christodoulakis指出，在优化过程中选择性错误会在计划树中以乘法方式传播，并导致糟糕的后续估计[45]。

- 搜索算法：一些商业系统，尤其是微软和T*a*nd*em*的系统，放弃了*Sel*inger的动态规划优化方法，而采用了基于C*asca*d*es*技术的目标导向的“自顶向下”搜索方案[25]。 自顶向下搜索在某些情况下可以减少优化器考虑的计划数量[82]，但也可能导致优化器内存消耗增加的负面影响。 如果实际成功是质量的指标，那么自顶向下搜索和动态规划之间的选择是无关紧要的。 已经证明两者在最先进的优化器中都能很好地工作，但运行时间和内存需求都不幸地与查询中的表数量呈指数关系。

一些系统对于具有“太多”表的查询采用启发式搜索方案。 尽管随机查询优化启发式算法的研究文献很有趣[5, 18, 44, 84]，但商业系统中使用的启发式算法往往是专有的，并且显然与随机查询优化文献不同。 一个教育性的练习是检查开源MySQL引擎的查询“优化器”，据最新检查，它完全是启发式的，并且主要依赖于利用索引和键/外键约束。 这让人想起了早期（臭名昭著的）Oracle版本。在某些系统中，如果FROM子句中的表太多，只有在用户明确指示优化器如何选择计划（通过嵌入在SQL中的所谓优化器“提示”）时，才能执行查询。

- 并行处理：现今的主要商业数据库管理系统都支持一定程度的并行处理。 大多数系统还支持“查询内部”并行处理：通过使用多个处理器来加速单个查询。 查询优化器需要

参与决定如何安排操作符 —以及并行化的操作符 — 跨多个CPU，以及（在共享无关或共享磁盘的情况下）跨多个独立计算机。 Hong和Stonebraker [42]选择避免并行优化的复杂性问题，并使用两个阶段：首先调用传统的单系统优化器来选择最佳的单系统计划，然后将该计划安排到多个处理器或计算机上。 已经发表了关于这个第二阶段优化的研究 [19, 21]尽管目前尚不清楚这些结果在多大程度上影响了当前的实践。

一些商业系统实现了上述的两阶段方法。 其他系统试图对集群的网络拓扑和数据分布进行建模，以在单个阶段中生成最优的计划。 虽然在某些情况下，单次遍历的方法可以产生更好的计划但目前尚不清楚使用单阶段方法可能带来的额外查询计划质量是否能够证明额外的优化器复杂性。 因此，许多当前的实现仍然倾向于使用两阶段方法。 目前这个领域更像是艺术而不是科学。 Oracle OPS（现在称为RAC）共享磁盘集群使用了两阶段优化器。 IBM DB2并行版（现在称为DB2数据库分区功能）最初使用了两阶段优化器，但后来逐渐向单阶段实现演进。

- 自动调优：各种正在进行的工业研究试图改进DBMS自动进行调优决策的能力。 其中一些技术基于收集查询工作负载，然后使用优化器通过各种“假设”分析来找到计划成本。 例如，如果存在其他索引或数据布局不同，会发生什么？ 优化器需要进行一些调整以有效支持这种活动，如Chaudhuri和Narasayya [12]所述。 Markl等人的学习优化器（LEO）工作也属于这一类别。

4.3.1 关于查询编译和重新编译的说明

SQL支持“准备”查询的能力：通过解析器、重写器和优化器处理查询，存储生成的查询执行计划，并在后续的“执行”语句中使用它。 这甚至适用于具有程序变量替代查询常量的动态查询（例如来自Web表单）。 唯一的问题是，在选择性估计期间，优化器假设由表单提供的变量具有“典型”值。 当选择了非典型的“典型”值时，可能会导致非常糟糕的查询执行计划。 查询准备对于基于表单驱动的、可预测数据的预定义查询特别有用：查询在应用程序编写时进行准备，在应用程序上线时，用户不会遇到解析、重写和优化的开销。

尽管在编写应用程序时准备查询可以提高性能，但这是一个非常限制性的应用程序模型。
许多应用程序员以及像Ruby on Rails这样的工具包在程序执行期间动态构建SQL语句，因此预编译不是一个选项。 由于这种情况非常普遍，数据库管理系统将这些动态查询执行计划存储在查询计划缓存中。 如果随后提交了相同（或非常相似）的语句，则使用缓存版本。 这种技术近似于预编译静态SQL的性能，而不受应用程序模型的限制，并且被广泛使用。

随着数据库随时间的变化，通常需要重新优化准备的计划。 至少，在删除索引时，必须从存储的计划缓存中删除使用该索引的任何计划，以便在下一次调用时选择新的计划。

关于重新优化计划的其他决策更加微妙，并且在供应商之间暴露了哲学上的区别。 一些供应商（例如IBM）非常努力地提供跨调用的可预测性能，以牺牲每次调用的最佳性能。 因此，除非计划不再执行（例如删除索引的情况），否则他们不会重新优化计划。 其他供应商（例如Microsoft）非常努力使其系统自我调整，并将重新优化计划。

更加积极地。 例如，如果表的基数发生了显著变化，SQL Server将触发重新编译，因为这种变化可能会影响索引和连接顺序的最佳使用。自我调整系统在动态环境中可能不太可预测，但更加高效。

这种哲学上的区别源于这些产品的历史客户群体的差异。 IBM传统上专注于高端客户，拥有熟练的数据库管理员和应用程序员。

在这些高预算的IT商店中，数据库的可预测性非常重要。 在花费数月时间调整数据库设计和设置之后，数据库管理员不希望优化器不可预测地进行更改。 相比之下，微软战略性地进入了低端数据库市场。因此，他们的客户往往拥有较低的IT预算和专业知识，并希望DBMS尽可能地“自我调整”。

随着时间的推移，这些公司的业务策略和客户群体已经趋于一致，以至于它们直接竞争，并且它们的方法也在一起发展。 微软拥有高规模企业客户，他们希望完全控制和查询计划的稳定性。 IBM有一些没有数据库管理员资源的客户，需要完全自动管理。

4.4 查询执行器

查询执行器在完全指定的查询计划上运行。这通常是一个连接封装了基表访问和各种查询执行算法的操作符的有向数据流图。在某些系统中，这个数据流图已经被优化器编译成低级操作码。在这种情况下，查询执行器基本上是一个运行时解释器。在其他系统中，查询执行器接收数据流图的表示，并根据图的布局递归地调用操作符的过程。 我们重点关注后一种情况，因为操作码方法实质上将我们在这里描述的逻辑编译成一个程序。

大多数现代查询执行器采用了最早的关系系统中使用的迭代器模型。 迭代器最简单地以面向对象的方式进行描述。 图4.2显示了一个简化的迭代器定义。 每个迭代器都指定了定义其输入的内容。

```
类迭代器 {
  迭代器和inputs[];
  void init();
  元组 get_next();
  void close();
}
```

图4.2迭代器超类伪代码。

数据流图中的边。 查询计划中的所有运算符——数据流图中的节点——都是迭代器类的子类实现的。 在典型系统中，子类集合可能包括文件扫描、索引扫描、排序、嵌套循环连接、合并连接、哈希连接、去重和分组聚合。 迭代器模型的一个重要特点是任何迭代器的子类都可以作为任何其他迭代器的输入。 因此，每个迭代器的逻辑与图中的子节点和父节点是独立的，并且不需要针对特定迭代器组合的特殊代码。

Graefe在他的查询执行调查[24]中提供了更多关于迭代器的细节。鼓励感兴趣的读者查看开源的PostgreSQL代码库。 PostgreSQL对大多数标准查询执行算法使用了适度复杂的迭代器实现。

4.4.1 迭代器讨论

迭代器的一个重要特性是它们将数据流与
控制流耦合在一起。 $get\ next()$调用是一个标准的过程调用，
通过调用堆栈将元组引用返回给调用者。 因此，当控制返回时，元组准确地返回给图中的父节点。
这意味着只需要一个单独的DBMS线程来执行整个查询图，不需要队列或迭代器之间的速率匹配。 这使得关系查询执行器易于实现和调试，并与其他环境中的数据流架构形成对比。 例如，网络依赖于各种协议用于并发生产者和消费者之间的排队和反馈。

单线程迭代器架构对于单系统（非集群）查询执行也非常高效。 在大多数数据库应用中，性能指标是查询完成时间，但也可以有其他优化目标。

在大多数情况下，性能评估的指标是查询完成时间，但也可以有其他优化目标。 例如，最大化DBMS吞吐量是另一个合理的目标。对于交互式应用程序，最受欢迎的目标是首行的时间。 在单处理器环境中，给定查询计划的完成时间是在资源充分利用时实现的。 在迭代器模型中，由于其中一个迭代器始终处于活动状态，资源利用率最大化。正如我们之前提到的，大多数现代DBMS支持并行查询执行。 幸运的是，这种支持可以在迭代器模型或查询执行架构基本不变的情况

下提供。 并行性和网络通信可以封装在特殊的交换迭代器中，如Graefe [23]所述；这些迭代器还实现了网络式的数据“推送”，对于DBMS迭代器来说是不可见的，它们保留了“拉取”式的“获取下一个”API。一些系统在其查询执行模型中也明确了推送逻辑。

-

4.4.2 数据在哪里?

我们对迭代器的讨论方便地回避了有关内存分配的问题。 我们既没有指定元组在内存中的存储方式，也没有指定它们在迭代器之间如何传递。 实际上，每个迭代器都预先分配了固定数量的元组描述符，一个用于每个输入，一个用于输出。 元组描述符通常是一个列引用的数组，其中每个列引用由一个指向内存中其他位置的元组的引用和该元组中的列偏移量组成。 基本迭代器超类逻辑从不动态分配内存。 这引发了一个问题，即实际引用的元组在内存中存储在哪里。

对于这个问题有两个可能的答案。 第一个答案是元组驻留在缓冲池中的页面中。 我们称这些为*BP*-元组。 如果一个迭代器构造了一个引用*BP*-元组的元组描述符，它必须

[1]这假设迭代器从不阻塞等待I/O请求。 在预取无效的情况下，迭代器模型中的低效率可能是由于在I/O上阻塞引起的。这在单系统数据库中通常不是问题，但在执行远程表或多系统集群中的查询时经常出现[23, 56]。

增加元组页的引用计数 - 该计数表示对该页上元组的活动引用数。 当元组描述符被清除时，它会减少引用计数。 第二种可能性是迭代器实现可能在内存堆上为元组分配空间。 我们称之为M元组。 迭代器可以通过从缓冲池复制列（复制由引用计数增加/减少对括号括起来），和/或通过计算表达式（例如，查询规范中的算术表达式如“EMP.sal $*0.1$”）来构造M元组。一种常见的方法是立即将数据从缓冲池复制到M元组中。这种设计将M元组作为唯一的在飞行中的元组结构，并简化了执行器代码。 该设计还规避了由于缓冲池引用计数和取消引用计数调用之间的长时间执行（和许多代码行）而导致的错误。

这种类型的一个常见错误是完全忘记取消固定页面（“缓冲泄漏”）。 不幸的是，正如第4.2节所指出的，独占使用M元组可能是一个主要的性能问题，因为内存复制在高性能系统中通常是一个严重的瓶颈。

另一方面，在某些情况下构建M元组是有意义的。 只要BP元组被迭代器直接引用，BP元组所在的页面必须保持固定在缓冲池中。 这将消耗一个页面大小的缓冲池内存，并限制了缓冲替换策略的选择。 如果元组在很长一段时间内仍然会被引用，从缓冲池中复制出一个元组可能是有益的。

这个讨论的结果是，最高效的方法是支持能够引用BP元组和M元组的元组描述符。

4.4.3 数据修改语句

到目前为止，我们只讨论了查询，也就是只读的SQL语句。 还有一类DML语句用于修改数据：插入、删除和更新语句。 这些语句的执行计划通常看起来像简单的直线查询计划，源头是一个访问方法，管道的末尾是一个数据修改操作符。

然而，在某些情况下，这些计划既查询又修改了相同的数据。读写同一张表（可能多次）的混合操作需要一些注意。一个简单的例子是臭名昭著的“Halloween问题”，因为它是由System R团队在10月31日发现的。Halloween问题源于一种特定的执行策略，用于类似“给工资低于[2]万美元的每个人加薪10%”的语句。对于这个查询，一个天真的计划将一个索引扫描迭代器管道化到一个更新迭代器（图4.3左侧）。这种管道化提供了良好的I/O性能。

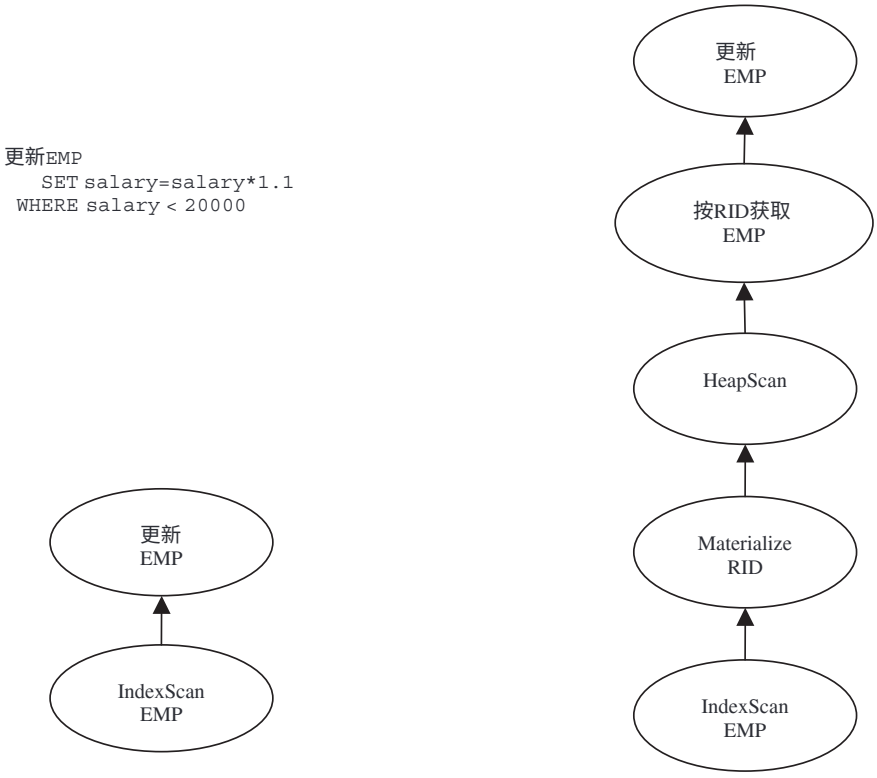


图4.3 通过IndexScan更新表的两个查询计划。左侧的计划容易受到Halloween问题的影响。右侧的计划是安全的，因为它在执行任何更新之前先确定要更新的所有元组。

[2]尽管名字非常相似，Halloween问题与第4.2.1节中的幽灵问题无关。

局部性，因为它在从B+-树中获取元组后立即修改它们。然而，这种流水线处理也可能导致索引扫描“重新发现”在修改后向右移动的元组，从而导致每个员工多次加薪。

在我们的例子中，所有低薪员工将会不断获得加薪，直到他们的收入超过2万美元。 这并不是该陈述的意图。

SQL语义禁止这种行为：一个单独的SQL语句不允许“看到”自己的更新。 需要注意一些细节，以确保遵守这个可见性规则。 一个简单、安全的实现方式是查询优化器选择避免在更新列上使用索引的计划。 在某些情况下，这可能非常低效。 另一种技术是使用批量读取-写入方案。 这在索引扫描和数据修改操作之间插入了记录ID材料化和获取操作符（图4.3右侧）。材料化操作符接收要修改的所有元组的ID，并将它们存储在临时文件中。 然后它扫描临时文件，并通过RID获取每个物理元组ID，并将结果元组提供给数据修改操作符。 如果优化器选择了一个索引，在大多数情况下，这意味着只有少数元组正在被修改。 因此，这种技术的表面上的低效可能是可以接受的，因为临时表很可能完全保留在缓冲池中。 还可以使用流水线更新方案，但需要存储引擎提供（有些奇特的）多版本支持。

4.5 访问方法

访问方法是管理系统支持的各种基于磁盘的数据结构的例程。 这些通常包括无序文件（“堆”）和各种类型的索引。 所有主要的商业系统都实现了堆和B+-树索引。 Oracle和PostgreSQL都支持哈希索引进行等值查找。

一些系统开始引入对多维索引的基本支持，例如R树[32]。 PostgreSQL支持一种可扩展的索引，称为广义搜索树（GiST）[39]，并且目前使用它来实现多维数据的R树和文本数据的RD树[40]。 IBM UDB Version 8引入了

多维聚集（MDC）索引用于通过多个维度的范围访问数据[66]。 针对主要是读取的数据仓库工作负载的系统通常还包括专门的位图变体索引[65]，如第4.6节所述。访问方法提供的基本API是迭代器API。init()例程扩展为接受“搜索参数”（或在System R的术语中，SARG ）的形式列运算符常量。 NULL SARG被视为请求扫描表中的所有元组。 访问方法层的get next()调用在没有更多满足搜索参数的元组时返回NULL。

-

将SARG传递到访问方法层有两个原因。
第一个原因很明显：像B+-树这样的索引访问方法需要SARG才能高效地运行。 第二个原因是一个更微妙的性能问题，但适用于堆扫描和索引扫描。 假设SARG由调用访问方法层的例程检查。 然后每次访问方法从get next()返回时，它必须要么（a）返回指向缓冲池中帧中的元组的句柄，并固定该帧中的页面以避免替换，要么（b）复制元组。 如果调用者发现SARG不满足，则负责要么（a）减少页面上的固定计数，要么（b）删除复制的元组。

然后必须重新调用 get_next()来尝试页面上的下一个元组。
这个逻辑在函数调用/返回对中消耗了大量的CPU周期，并且要么无谓地固定页面在缓冲池中（产生不必要的缓冲帧争用），要么无谓地创建和销毁元组的副本 —— 当流式处理数百万个元组时，这是一个显著的CPU开销。 请注意，典型的堆扫描将访问给定页面上的所有元组，导致每个页面上的多次迭代。 相比之下，如果所有这些逻辑都在访问方法层中完成，通过一次测试一个页面的SARGs，可以避免重复的调用/返回对和固定/解固定或复制/删除。只有满足SARG的元组才会从get next()调用中返回。SARGs在存储引擎和关系引擎之间保持了一个清晰的架构边界，同时获得了出色的性能。 因此，许多系统支持非常丰富的SARG支持并广泛使用它们。从主题上看，这是

标准数据库管理系统智慧的一个实例是将工作分摊到集合中的多个项目上，但在这种情况下，它被应用于CPU性能，而不是磁盘性能。

所有数据库管理系统都需要一种方法来“指向”基表中的行，以便索引条目可以适当地引用这些行。 在许多数据库管理系统中，这是通过使用直接行ID（RID）来实现的，这些RID是基表中行的物理磁盘地址。 这样做的好处是速度快，但缺点是使基表行的移动非常昂贵，因为所有指向该行的二级索引都需要更新。 查找和更新这些行都可能很昂贵。 当更新更改行大小并且当前页面上没有足够的空间来存储更新后的行时，行需要移动。当B+-树分裂时，许多行需要移动。 DB2使用转发指针来避免第一个问题。 这需要进行第二次I/O来查找移动的页面，但避免了更新二级索引的操作。 DB2通过不支持B+-树作为基表元组的主存储来避免第二个问题。 Microsoft SQL Server和Oracle支持将B+-树作为主存储，并且必须能够有效地处理行移动。 采取的方法是在二级索引中不使用物理行地址，而是使用行主键（如果表没有唯一键，则使用一些附加的系统提供的位来强制唯一性），而不是物理RID。这样做会在使用二级索引访问基表行时牺牲一些性能，但避免了行移动带来的问题。 Oracle通过在主键中保留物理指针来避免此方法在某些情况下的性能损失。如果行没有移动，将使用物理指针快速找到它。 但是，如果行已经移动，将使用较慢的主键技术。

Oracle通过允许行跨越页面来避免在堆文件中移动行。 因此，当一行被更新为一个不再适合原始页面的较长值时，他们不会强制移动行，而是将适合原始页面的部分存储在原始页面上，剩余部分可以跨越到下一个页面。

与所有其他迭代器相比，访问方法与围绕事务的并发和恢复逻辑有着深入的交互，如第4节所述。

4.6 数据仓库

数据仓库是用于决策支持的大型历史数据库，定期加载新数据，因此它们需要专门的查询处理支持，在下一节中，我们将概述它们通常需要的一些关键功能。 这个主题有两个主要原因：

1. 数据仓库是DBMS技术的一个非常重要的应用。 有人声称仓库占据了所有DBMS活动的1/3 [26, 63]。

2. 到目前为止，在本节中讨论的传统查询优化和执行引擎在数据仓库上效果不佳。 因此，需要进行扩展或修改以实现良好的性能。

关系型数据库管理系统（DBMS）最早是在20世纪70年代和80年代设计的，以满足商业数据处理应用的需求，因为那时是主要需求。 在20世纪90年代初出现了数据仓库和“业务分析”市场，并自那时以来发展迅速。

到20世纪90年代，联机事务处理（OLTP）已经取代了批处理业务数据处理，成为数据库使用的主要范式。 此外，大多数OLTP系统都有一组计算机操作员提交事务，要么通过与最终客户的电话交谈，要么通过从纸上进行数据输入。 自动取款机已经广泛使用，使客户能够直接进行某些交互而无需操作员干预。 对于这类交易来说，响应时间至关重要。 随着互联网迅速取代操作员，这类响应时间要求如今变得更加紧迫和多样化，最终客户通过自助服务进行操作。

大约在同一时间，零售领域的企业有了一个想法，即捕捉所有历史销售交易，并通常存储一两年。 买家可以利用这些历史销售数据来了解“热门和不热门”的情况。这些信息可以影响购买模式。 同样，这些数据可以用来决定哪些商品进行促销，哪些商品打折，以及

哪些商品退回给制造商。 当时的共识是，零售领域的历史数据仓库通过更好的库存管理、货架和店铺布局在几个月内就能回本。

当时很明显，数据仓库应该部署在与OLTP系统分离的硬件上。 使用这种方法，冗长（而且常常不可预测）的商业智能查询不会影响OLTP的响应时间。 此外，数据的性质非常不同；仓库处理历史数据，OLTP处理“现在”数据。最后，发现历史数据所需的模式通常与当前数据所需的模式不匹配，需要进行数据转换。

出于这些原因，工作流系统被构建出来，可以从操作性OLTP系统中“抓取”数据并将其加载到数据仓库中。 这样的系统被称为“抽取、转换和加载”(ETL)系统。 流行的ETL产品包括IBM的Data Stage和Informatica的PowerCenter。在过去的十年中，ETL供应商还通过数据清洗工具、去重工具和其他质量导向的产品扩展了他们的产品。

在数据仓库环境中，有几个问题必须解决，我们将在下面讨论。

4.6.1 位图索引

B+-树被优化用于快速插入、删除和更新记录。 相比之下，数据仓库执行初始加载，然后数据在几个月或几年内保持静态。 此外，数据仓库通常具有少量值的列。 例如，考虑存储客户的性别。 只有两个值，可以用一位来表示每个记录中的位图。 相比之下，B+-树将需要每个记录的(值，记录指针)对，并且通常每个记录将消耗40位。

位图对于合取过滤器也是有优势的，比如

$$\text{Customer.sex} = \text{“F”} \text{ and } \text{Customer.state} = \text{“California”}$$

在这种情况下，结果集可以通过交集
位图来确定。 还有一系列更复杂的位图运算

有一些技巧可以提高常见分析查询的性能。 对于位图处理的讨论，有兴趣的读者应该参考[65]。

在当前的产品中，位图索引与B+-树相辅相成用于索引存储的数据，而DB2提供了一个更有限的版本。 Sybase IQ广泛使用位图索引。 当然，位图的缺点是更新代价高昂，因此它们的实用性受到了仓库环境的限制。

4.6.2 快速加载

通常，数据仓库在夜间加载当天的交易数据。 这对于只在白天营业的零售机构来说是一种明显的策略。夜间批量加载的第二个原因是避免在用户交互期间出现更新。 考虑一个业务分析师希望制定某种形式的自由查询，也许是为了调查飓风对客户购买模式的影响。 这个查询的结果可能会提示一个后续查询，比如调查大风暴期间的购买模式。 这两个查询的结果应该是兼容的，即答案应该在相同的数据集上计算。 如果数据正在被并发加载，这可能会成为一个问题，尤其是对于包含最近历史的查询。

因此，数据仓库需要能够快速进行批量加载。 虽然可以使用一系列的SQL插入语句来编程仓库加载，但这种策略在实践中从未被使用过。 相反，使用了一个批量加载器，它可以将大量的记录流式传输到存储中，而不需要SQL层的开销，并利用了像B+-树这样的访问方法的特殊批量加载方法。粗略地说，批量加载器比SQL插入快一个数量级，并且所有主要供应商都提供高性能的批量加载器。

随着世界向电子商务和全天候销售的转变，这种批量加载策略变得不太合理。 但是转向“实时”仓库也存在一些问题。 首先，无论是来自批量加载器还是事务的插入操作，都必须像第6.3节中讨论的那样设置写锁。这些锁会与读锁发生冲突。

查询可能会导致数据仓库“冻结”。其次，如上所述，提供与查询集相兼容的答案是有问题的。

通过避免原地更新和提供历史查询，可以解决这两个问题。 如果保留更新之前和之后的值，并适当地加上时间戳，那么可以提供过去某个时间点的查询。 在同一历史时间点运行一系列查询将提供兼容的答案。 此外，可以在不设置读锁的情况下运行相同的历史查询。

如5.2.1节所讨论的，一些供应商（尤其是Oracle）提供了多版本（MVCC）隔离级别，如快照隔离。随着实时数据仓库的普及，其他供应商也将相应跟进。

4.6.3 材料化视图

数据仓库通常非常庞大，连接多个大型表的查询往往需要很长时间。为了加速热门查询的性能，大多数供应商提供了材料化视图。 与本节前面讨论的纯逻辑视图不同，材料化视图是实际可查询的表，但对应于对真实“基本”数据表的逻辑视图表达式。 对材料化视图的查询将避免在运行时执行视图表达式中的连接操作。 相反，必须在执行更新时保持材料化视图的最新状态。

物化视图的使用有三个方面：(a) 选择需要物化的视图，(b) 保持视图的新鲜度，(c) 考虑在即席查询中使用物化视图。

主题(a)是自动数据库调优的高级方面，我们在4.3节中提到过。主题(c)在各种产品中以不同程度实现；即使对于简单的单块查询[51]，这个问题在理论上也具有挑战性，对于带有聚合和子查询的通用SQL来说则更加如此。 对于(b)，大多数供应商提供多种刷新技术，从对从物化视图派生的表进行每次更新的物化视图更新，到定期丢弃然后重新创建物化视图。

这些策略在运行时开销和物化视图的数据一致性之间进行权衡。

4.6.4 OLAP和自由查询支持

一些仓库工作负载具有可预测的查询。 例如，在每个月底，可能会运行一个汇总报告，为零售连锁店中的每个销售区域提供按部门的总销售额。 在这个工作负载中夹杂着即席查询，由业务分析师即时制定。

显然，可预测的查询可以通过适当构建的物化视图来支持。 更一般地说，由于大多数业务分析查询都要求聚合，可以计算一个物化视图，该视图是每个店铺按部门的销售总额。 然后，如果上述区域查询被指定，可以通过“卷起”每个区域中的个别店铺来满足该查询。

这种聚合通常被称为数据立方体，是一类感兴趣的物化视图。 在20世纪90年代初，诸如Ess*b*ase之类的产品提供了定制工具，用于将数据存储在优先级立方体格式中，同时提供基于立方体的用户界面来浏览数据，这种能力被称为在线分析处理（OLAP）。 随着时间的推移，数据立方体支持已经添加到全功能关系数据库系统中，并且通常被称为关系型OLAP（ROLAP）。 许多提供ROLAP的DBMS已经发展到在特定情况下内部实现一些早期的OLAP风格存储方案，并因此有时被称为混合OLAP（HOLAP）方案。

显然，数据立方体为可预测、有限的查询提供了高性能。 然而，它们通常对于支持即席查询不太有帮助。

4.6.5 雪花模式查询的优化

许多数据仓库遵循一种特定的模式设计方法。
具体来说，它们存储了一组事实，在零售环境中，通常是简单的记录，比如“客户X在时间T从商店Z购买了产品Y”。一个中心事实表记录了信息-

关于每个事实的信息，例如购买价格、折扣、销售税信息等。事实表中还包含了每个维度的外键。 维度可以包括客户、产品、商店、时间等。这种形式的模式通常被称为星型模式，因为它有一个中心事实表，周围是维度，每个维度与事实表有一个1-N的主键外键关系。 在实体关系图中绘制，这样的模式呈星形。

许多维度自然上是分层的。例如，如果商店可以聚合到地区中，那么“商店”维度表就会有一个额外的外键指向地区维度表。 类似的层次结构通常适用于涉及时间（月/日/年）、管理层次等属性。在这些情况下，多级星型或雪花型模式会产生。

基本上，所有数据仓库查询都涉及在这些表中的某些属性上对雪花型模式中的一个或多个维度进行过滤，然后将结果与中心事实表进行连接，按照事实表或维度表中的某些属性进行分组，然后计算一个SQL聚合。

随着时间的推移，供应商在其优化器中特别处理了这类查询，因为它非常受欢迎，并且选择一个好的计划对于这些长时间运行的命令至关重要。

4.6.6 数据仓库：结论

正如可以看到的，数据仓库需要与OLTP环境完全不同的能力。 除了B+-树，还需要位图索引。 与通用优化器不同，需要特别关注雪花模式下的聚合查询。 不再需要普通视图，而是需要物化视图。 不再需要快速事务更新，而是需要快速批量加载等。有关数据仓库实践的更详细概述可参见[11]。

主要的关系型数据库供应商从OLTP导向的架构开始，并随着时间的推移增加了面向数据仓库的功能。 此外，还有许多较小的供应商在这个领域提供DBMS解决方案。 其中包括Teradata和Netezza，它们提供基于专有硬件的共享无关的DBMS运行。此外，还有Greenplum（PostgreSQL的并行化）、DATAl-legro和EnterpriseDB等供应商，它们都在更传统的硬件上运行。

在这个领域还有Greenplum（PostgreSQL的并行化）、DATAl-legro和EnterpriseDB等供应商，它们都在更传统的硬件上运行。

最后，有些人（包括其中一位作者）声称列存储在数据仓库空间中相对于传统存储引擎具有巨大优势，其中存储单元是表行。将每个列单独存储在“宽”（高度）的表中尤其高效，并且访问通常只涉及几个列。 列存储还可以实现简单有效的磁盘压缩，因为列中的所有数据都是相同类型的。 列存储的挑战在于表中行的位置需要在所有存储的列中保持一致，否则需要额外的机制来连接列。 这对于OLTP来说是一个大问题，但对于仓库或系统日志存储库这样的追加式数据库来说不是一个主要问题。 提供列存储的供应商包括Sybase、Vertica、Sand、Vhayu和KX。有关此架构讨论的更多详细信息，请参阅[36, 89, 90]。

4.7 数据库可扩展性

传统上，关系数据库被认为在存储数据类型方面存在限制，主要关注企业和行政记录中使用的“事实和数字”。 然而，如今它们托管了各种以流行的编程语言表达的数据类型。 通过以多种方式使核心关系型DBMS可扩展来实现这一点。 在本节中，我们简要概述了广泛使用的扩展类型，并突出了在提供这种可扩展性时出现的一些架构问题。 这些功能在大多数商业DBMS中以不同程度出现，并且在开源的PostgreSQL DBMS中也有。

4.7.1 抽象数据类型

原则上，关系模型对可以放置在模式列上的标量数据类型的选择是不可知的。 但最初的关系数据库系统只支持一组静态的字母数字列类型，并且这种限制与关系模型相关联。

关系模型本身. 关系型数据库管理系统可以在运行时对新的抽象数据类型进行扩展，就像早期的Ingres-ADT系统和后续的Postgres系统[88]所示。 为了实现这一点，数据库管理系统的类型系统 - 因此解析器 - 必须从系统目录中获取驱动，该目录维护系统已知类型的列表，并指向用于操作这些类型的“方法”（代码）的指针。 在这种方法中，数据库管理系统不解释类型，只是在表达式求值中适当地调用它们的方法；因此被称为“抽象数据类型”。以二维空间“矩形”为典型例子，可以注册一个类型，并为矩形交集或并集等操作注册方法。 这也意味着系统必须提供一个运行时引擎来执行用户定义的代码，并在不崩溃数据库服务器或损坏数据的情况下安全地执行该代码。 今天的主要数据库管理系统都允许用户在现代SQL的命令式“存储过程”子语言中定义函数。除了MySQL外，大多数系统至少支持几种其他语言，通常是C和Java。在Windows平台上，Microsoft SQL Server和IBM DB2支持编译为Microsoft .Net公共语言运行时的代码，可以用多种语言编写，最常见的是Visual Basic、C++和C#。PostgreSQL支持C、Perl、Python和Tcl，并允许在运行时添加对新语言的支持 - 有流行的第三方插件支持Ruby和开源的R统计包。

为了使抽象数据类型在数据库管理系统中运行高效，查询优化器必须考虑选择和连接谓词中的“昂贵”用户定义代码，并在某些情况下推迟选择直到连接之后[13, 37]。 为了使抽象数据类型更加高效，能够在其上定义索引是很有用的。 至少需要扩展B+-树以索引抽象数据类型上的表达式，而不仅仅是列（有时称为“函数索引”），并且当适用时，必须扩展优化器来选择它们。 对于除线性顺序（$<, >, =$）之外的谓词，B+-树是不够的，系统需要支持可扩展的索引方案；文献中有两种方法，一种是原始的Postgres可扩展访问方法接口[88]，另一种是GiST [39]。

4.7.2 结构化类型和XML

ADT被设计为与关系模型完全兼容 -它们不以任何方式改变基本的关系代数，只改变属性值上的表达式。 多年来，有许多提议支持非关系结构化类型的更激进的数据库变更，例如嵌套的集合类型（如数组，集合，树和嵌套的元组和/或关系）。

也许今天最相关的这些提议之一是通过XPath和XQuery等语言支持XML。处理结构化类型（如XML）有大致三种方法。第一种方法是构建一个操作具有结构化类型数据的自定义数据库系统；从历史上看，这些尝试一直被传统关系型DBMS中容纳结构化类型的方法所掩盖，而在XML的情况下也是如此。 第二种方法是将复杂类型视为ADT。例如，可以定义一个具有XML类型列的关系表，每行存储一个XML文档。这意味着搜索XML的表达式（例如XPath树匹配模式）在查询优化器中以不透明的方式执行。 第三种方法是DBMS在插入时将嵌套结构“规范化”为一组关系，其中外键将子对象连接到其父对象。 这种技术有时被称为“剪切”XML，它在关系框架内公开了数据的所有结构，但增加了存储开销，并且在查询时需要连接数据。 大多数DBMS供应商今天都提供存储的ADT和剪切选项，并允许数据库设计人员在它们之间进行选择。 在XML的情况下，剪切方法通常还提供删除在同一级别嵌套的XML元素之间的排序信息的选项，这

[3] XML有时被称为“半结构化”数据，因为它对文档的结构没有任何限制。 然而，与自由文本不同，它鼓励结构化，包括非关系复杂性，如排序和嵌套集合。 在存储和查询XML文档时，复杂性往往来自于处理XML文档的结构，而不是处理文档内的非结构化文本问题。 事实上，许多用于XML查询处理的技术都源于对丰富结构化ODMG对象数据库模型及其OQL查询语言的研究。

通过允许连接重新排序和其他关系优化，可以提高查询性能。

相关问题是对关系模型进行更小的扩展，以处理嵌套表和元组，以及数组。 例如，在Oracle安装中广泛使用这些。 在许多方面，设计权衡与处理XML的方式相似。

4.7.3 全文搜索

传统上，关系数据库在处理富文本数据和通常伴随的关键字搜索方面一直很差。原则上，在数据库中建模自由文本是一个简单的问题，只需要存储文档，定义一个“倒排文件”关系，其元组形式为(单词，文档ID，位置)，并在单词列上构建一个B+-树索引。 这大致是任何文本搜索引擎中发生的情况，除了一些词语的语言规范化和一些额外的元组属性以帮助对搜索结果进行排序。

但是除了模型之外，大多数文本索引引擎还实现了一些特定于此模式的性能优化，这些优化在典型的DBMS中没有实现。其中包括将模式“去规范化”，使每个单词只出现一次，并带有每个单词的出现列表，即(单词，列表<文档ID，位置>)。这允许对列表进行积极的增量压缩（通常称为“倒排列表”），这在考虑到文档中单词的特征偏斜（Zipfian分布）非常重要。 此外，文本数据库往往以数据仓库的方式使用，绕过任何DBMS事务逻辑。 普遍的观点是，像上面那样在DBMS中进行文本搜索的天真实现大约比自定义文本索引引擎慢一个数量级。

然而，大多数DBMSs今天要么包含一个用于文本索引的子系统，要么可以与一个单独的引擎捆绑在一起来完成这项工作。文本索引功能通常可以同时用于全文文档和元组中的短文本属性。 在大多数情况下，全文索引是异步更新（“爬取”）的，而不是事务性地维护；PostgreSQL在提供具有事务性更新的全文索引选项方面是不寻常的。在某些系统中，

全文索引存储在DBMS之外，因此需要单独的工具进行备份和恢复。在处理关系数据库中的全文搜索时，一个关键挑战是将关系查询的语义（无序和完整的结果集）与使用关键字进行排名的文档搜索（有序且通常不完整的结果集）有用和灵活地结合起来。例如，在每个关系上都有关键字搜索谓词时，如何对两个关系的连接查询的输出进行排序是不清楚的。这个问题在当前实践中仍然是相当临时的。在为查询输出确定语义的基础上，另一个挑战是关系查询优化器对文本索引的选择性和成本估计进行推理，以及判断查询的适当成本模型，该查询的答案集在用户界面中是有序且分页的，并且可能不会完全检索。根据所有报告，这个最后的主题正在一些流行的DBMS中积极追求。

4.7.4 附加可扩展性问题

除了数据库可扩展性的三个主要使用场景外，我们还提出了引擎内的两个核心组件，这些组件通常可以为各种用途进行扩展。

关于可扩展查询优化器，已经提出了许多方案，包括支持IBM DB2优化器的设计[54, 68]，以及支持Tandem和Microsoft优化器的设计[25]。所有这些方案都提供了基于规则的子系统，用于生成或修改查询计划，并允许独立注册新的优化规则。当查询执行器添加新功能或针对特定查询重写或计划优化时，这些技术有助于更容易地扩展优化器。这些通用架构对于实现上述许多特定可扩展类型功能非常重要。

自早期系统以来，另一种横向可扩展性形式是数据库能够将远程数据源“包装”到模式中，就像它们是本地表一样，并在查询处理期间访问它们。在这方面的一个挑战是优化器如何处理不支持扫描的数据源，但仍能响应查询。

请求将值分配给变量；这需要将索引SARG与查询谓词进行泛化的优化器逻辑匹配[33]。
另一个挑战是执行器高效处理可能在产生输出时缓慢或突发的远程数据源；这泛化了查询执行器异步磁盘I/O的设计挑战，将访问时间的变异性增加一个数量级或更多[22, 92]。

4.8标准实践

基本上所有关系型数据库查询引擎的粗略架构看起来都类似于System R原型[3]。 多年来，查询处理的研究和开发集中在该框架内的创新，以加速越来越多的查询和模式。 系统之间的主要设计差异在于优化器搜索策略（自顶向下 vs. 自底向上）以及查询执行器控制流模型，特别是对于共享无关和共享磁盘并行性（迭代器和交换操作符 vs. 异步生产者/消费者方案）。 在更细粒度的级别上，优化器、执行器和访问方法中使用的方案组合在不同的工作负载（包括OLTP、数据仓库决策支持和OLAP）中实现良好的性能。商业产品中的“秘密配方”决定了它们在特定情况下的性能表现如何，但在大致上，所有商业系统在广泛的工作负载范围内表现都相当出色，并且可以在特定工作负载中看起来较慢。

在开源领域中，PostgreSQL具有一个相当复杂的查询处理器，具有传统的基于成本的优化器，广泛的执行算法以及一些商业产品中没有的可扩展性功能。 MySQL的查询处理器要简单得多，建立在索引上的嵌套循环连接之上。 MySQL查询优化器专注于分析查询，以确保常见操作轻量且高效，特别是键/外键连接、外连接到连接重写以及只请求结果集的前几行的查询。阅读MySQL的手册和查询处理代码并将其与更复杂的传统设计进行比较是很有启发性的，要记住MySQL在实践中的高采用率以及它似乎擅长的任务。

手册和查询处理代码，并将其与更复杂的传统设计进行比较，要记住MySQL在实践中的高采用率以及它似乎擅长的任务。

4.9 讨论和附加材料

由于查询优化和执行的清晰模块化，多年来在这个环境中已经开发出了大量的算法、技术和技巧，并且关系查询处理的研究至今仍在进行中。 令人高兴的是，大多数在实践中使用的想法（以及许多未使用的想法）都可以在研究文献中找到。 查询优化研究的一个很好的起点是Chaudhuri的简短调查[10]。 对于查询处理研究，Graefe提供了一份非常全面的调查[24]。

除了传统的查询处理外，近年来在处理大数据集时，已经进行了大量的工作，将丰富的统计方法纳入其中。 一个自然的扩展是使用抽样或摘要统计来提供聚合查询的数值近似[20]，可能以不断改进的在线方式[38]。 然而，尽管研究结果相对成熟，但在市场上的接受速度相对较慢。 Oracle和DB2都提供了简单的基表抽样技术，但在涉及多个表的查询中，它们不能提供统计上健壮的估计。 大多数供应商并没有专注于这些功能，而是选择丰富其OLAP功能，这限制了可以快速回答的查询类型，但提供了100%正确的答案给用户。

另一个重要但更基础的扩展是在DBMS中包含“数据挖掘”技术。流行的技术包括统计聚类、分类、回归和关联规则[14]。 除了研究文献中独立实现这些技术外，还存在将这些技术与丰富的关系查询集成的架构挑战[77]。

最后，值得注意的是，更广泛的计算社区最近对数据并行性产生了兴趣，如Google的Map-Reduce、Microsoft的Dryad和开放的-

由Yahoo支持的源Hadoop代码。这些系统非常类似于共享无关的并行关系查询执行器，由应用程序逻辑的程序员实现的自定义查询运算符。 它们还包括简单但合理设计的方法来管理参与节点的故障，这在大规模上是常见的。 也许这一趋势最有趣的方面是它在计算中被创造性地用于各种数据密集型问题，包括文本和图像处理以及统计方法。 有趣的是看到数据库引擎的其他想法是否被这些框架的用户借鉴 - 例如，Yahoo!正在早期工作中扩展Hadoop以支持声明性查询和优化器。 基于这些框架构建的创新也可以被纳入数据库引擎中。

5

存储管理

今天商业使用的DBMS存储管理器有两种基本类型：(1) DBMS直接与低级块模式设备驱动程序交互，用于磁盘(通常称为原始模式访问)，或者(2) DBMS使用标准操作系统文件系统功能。 这个决定影响了DBMS在空间和时间上控制存储的能力。

我们依次考虑这两个维度，并继续详细讨论存储层次结构的使用。

5.1 空间控制

磁盘与磁盘之间的顺序带宽比随机访问快10到100倍，这个比例还在增加。 磁盘密度每18个月翻一番，带宽大约与密度的平方根成正比(并且与旋转速度成线性关系)。 然而，磁盘臂的移动速度提高得要慢得多——大约每年提高7% [67]。 因此，对于需要大量数据的查询能够顺序访问，DBMS存储管理器将块放置在磁盘上至关重要。 由于DBMS能够比底层操作系统更深入地理解其工作负载访问模式，它

对于数据库管理系统（DBMS）的架构师来说，完全控制数据库块在磁盘上的空间位置是有意义的。

DBMS控制其数据的空间局部性的最佳方法是直接将数据存储到“原始”磁盘设备上，并完全避免使用文件系统。 这样做的原因是原始设备地址通常与存储位置的物理接近程度密切相关。 大多数商用数据库系统都提供此功能以实现最佳性能。

尽管这种技术有效，但也存在一些缺点。 首先，它要求数据库管理员将整个磁盘分区用于DBMS，这使得它们对需要文件系统接口的实用程序（备份等）不可用。 其次，“原始磁盘”访问接口通常是特定于操作系统的，这可能使得DBMS更难移植。 然而，大多数商用DBMS供应商在多年前就克服了这个障碍。最后，存储行业的发展，如RAID、存储区域网络（SAN）和逻辑卷管理器变得越来越流行。 现在，在大多数情况下，“虚拟”磁盘设备已经成为常态，实际上，“原始”设备接口实际上被设备或软件拦截，积极地将数据重新定位到一个或多个物理磁盘上。 因此，DBMS通过显式物理控制获得的好处随着时间的推移而被削弱。 我们在第7.3节进一步讨论了这个问题。

与直接访问原始磁盘相比，DBMS的另一种选择是在操作系统文件系统中创建一个非常大的文件，并管理数据在该文件中的偏移位置。 该文件实质上被视为一个线性数组的磁盘页面。 这样可以避免直接设备访问的一些缺点，并且仍然提供相当不错的性能。 在大多数流行的文件系统中，如果你在空磁盘上分配一个非常大的文件，那么该文件中的偏移位置将与物理存储区域的接近程度相当接近。 因此，这是对原始磁盘访问的一个很好的近似，而无需直接访问原始设备接口。 大多数虚拟化存储系统也被设计为将文件中的接近偏移位置放置在附近的物理位置。 因此，相对于使用原始磁盘而言，使用大文件而不是原始磁盘时所失去的相对控制在逐渐减少。 使用文件系统接口还涉及到时间控制方面的其他影响，我们将在下一小节中讨论。

作为一个数据点，我们最近在一个中型系统上使用一种主要商业DBMS之一进行了直接原始访问和大型文件访问的比较，并发现在运行TPC-C基准测试[91]时只有6%的性能下降，并且对于较少I/O密集型工作负载几乎没有负面影响。 DB2报告使用直接I/O（DIO）及其变种（如并发I/O（CIO））时，文件系统开销低至1%。 因此，DBMS供应商通常不再推荐使用原始存储，并且很少有客户在这种配置下运行。 它仍然是主要商业系统中支持的一个功能，主要用于基准测试。一些商业DBMS还允许根据预期工作负载设置数据库页面大小。 IBM DB2和Oracle都支持这个选项。 由于增加了管理复杂性，其他商业

系统（如Microsoft SQL Server）不支持多个页面大小。 如果支持可调整页面大小，则所选大小应为文件系统（或原始设备如果使用原始I/O）使用的页面大小的倍数。

在“5分钟规则”论文中讨论了适当选择页面大小的问题，后来更新为“30分钟规则”[27]。 如果使用文件系统而不是原始设备访问，则可能需要特殊接口来写入与文件系统不同大小的页面；例如，POSIXmmap/msync调用提供了这样的支持。

5.2 时间控制：缓冲

除了控制数据在磁盘上的位置之外，DBMS还必须控制数据何时被物理写入磁盘。
正如我们将在第5节讨论的那样，DBMS包含了关键的逻辑，用于决定何时将块写入磁盘。 大多数操作系统文件系统还提供了内置的I/O缓冲机制，用于决定何时进行文件块的读取和写入。 如果DBMS使用标准文件系统接口进行写入，则操作系统的缓冲机制可能会混淆DBMS逻辑的意图，通过暗中推迟或重新排序写入操作。 这可能会给DBMS带来重大问题。

第一组问题涉及数据库的正确性和ACID事务承诺：DBMS无法保证原子恢复-

在没有明确控制磁盘写入的时机和顺序的情况下，软件或硬件故障后的恢复是不可能的。 正如我们将在第5.3节中讨论的那样，写前日志记录协议要求将写入日志设备的操作必须在写入数据库设备之前进行，并且提交请求在提交日志记录已可靠地写入日志设备之前不能返回给用户。

与操作系统缓冲相关的第二组问题涉及性能，但对正确性没有影响。 现代操作系统文件系统通常具有一些内置的支持，用于预读（推测读取）和写后（延迟、批量写入）。 这些通常不适合DBMS的访问模式。 文件系统逻辑依赖于文件中物理字节偏移的连续性，以进行预读决策。 DBMS级别的I/O功能可以基于SQL查询处理级别已知但在文件系统级别不容易识别的未来读取请求，支持逻辑预测性I/O决策。 例如，在扫描B+-树的叶子节点时（行存储在B+-树的叶子节点中），可以请求逻辑DBMS级别的预读，这些叶子节点不一定是连续的。 通过在需要之前，DBMS可以提前发出I/O请求来轻松实现逻辑预读。 查询执行计划包含有关数据访问算法的相关信息，并且对于查询的未来访问模式具有完整的信息。 同样，DBMS可能希望根据混合了锁争用和I/O吞吐量等问题的考虑，自行决定何时刷新日志尾部。 这种详细的未来访问模式知识对于DBMS是可用的，但对于操作系统文件系统来说不可用。

最终的性能问题是“双缓冲”和内存复制的高CPU开销。考虑到DBMS必须仔细进行自己的缓冲以确保正确性，操作系统的任何额外缓冲都是多余的。 这种冗余导致了两个成本。 首先，它浪费了系统内存，有效地减少了可用于执行有用工作的内存。 其次，它浪费了时间和处理资源，因为它导致了额外的复制步骤：在读取时，数据首先从磁盘复制到操作系统缓冲区，然后再次复制到DBMS缓冲池。 在写入时，这两个复制都需要反向进行。

在内存中复制数据可能成为一个严重的瓶颈。 复制会导致延迟，消耗CPU周期，并可能使CPU数据缓存溢出。

缓存。 这个事实常常让那些没有操作或实现过数据库系统的人感到惊讶，并且他们认为与磁盘I/O相比，主存操作是“免费”的。但实际上，在经过良好调优的事务处理*DBMS*中，吞吐量通常不受I/O限制。通过购买足够的磁盘和RA*M*，高端安装可以通过缓冲池吸收重复的页面请求，并以能够满足系统中所有处理器的数据需求的速率共享磁盘I/O。 一旦实现了这种“系统平衡”，I/O延迟就不再是主要的系统吞吐量瓶颈，剩下的主存瓶颈成为系统的限制因素。 内存复制正在成为计算机体系结构中的主要瓶颈：这是由于原始CPU每秒每美元的性能演进与RA*M*访问速度之间的性能差距（RA*M*访问速度明显滞后于摩尔定律）[67]。

OS缓冲的问题在数据库研究文献[86]和行业中已经广为人知。 现代大多数操作系统现在提供了钩子（例如，POSIXmmap套件调用或平台特定的DIO和CIO API集），以便像数据库服务器这样的程序可以绕过双缓冲文件缓存。 这确保了在请求时写入磁盘，避免了双缓冲，并且数据库管理系统可以控制页面替换策略。

5.3 缓冲管理

为了提供对数据库页面的高效访问，每个数据库管理系统在自己的内存空间中实现了一个大型共享缓冲池。 在早期，缓冲池是静态分配给一个管理选择的值，但现在大多数商业数据库管理系统根据系统需求和可用资源动态调整缓冲池大小。

缓冲池被组织成一个帧的数组，每个帧是一个与数据库磁盘块大小相同的内存区域。 块被从磁盘复制到缓冲池中，不进行格式更改，在内存中以原生格式进行操作，然后再写回。 这种无需翻译的方法避免了在将数据从磁盘“编组”和“解组”时的CPU瓶颈；或许更重要的是，

固定大小的帧避免了外部碎片和压缩所引起的内存管理复杂性，这是通用技术所造成的。

与缓冲池帧数组相关联的是一个哈希表，它将(1)当前在内存中持有的页号映射到它们在帧表中的位置，(2)该页在后备磁盘存储上的位置，以及(3)关于该页的一些元数据。 元数据包括一个脏位，用于指示该页自从从磁盘读取以来是否发生了更改，以及由页面替换策略选择在缓冲池已满时要驱逐的页面的任何信息。 大多数系统还包括一个固定计数，用于表示该页不符合参与页面替换算法的条件。 当固定计数非零时，该页被“固定”在内存中，不会被强制写入磁盘或被抢占。

这允许DBMS的工作线程在操作页面之前通过增加引用计数来固定缓冲池中的页面，然后在操作之后递减引用计数。 其目的是在任何固定时间点上，只有缓冲池的一小部分被固定。 一些系统还提供将表固定在内存中作为管理选项的能力，这可以提高对小型、频繁使用的表的访问速度。

然而，固定页面会减少用于正常缓冲池活动的页面数量，并且随着固定页面的百分比增加，可能会对性能产生负面影响。

在关系系统的早期，许多研究都集中在页面替换策略的设计上，因为DBMS中发现的数据访问模式的多样性使得简单的技术无效。

例如，某些数据库操作往往需要对整个表进行扫描，当被扫描的表比缓冲池大得多时，这些操作往往会清除缓冲池中所有常用的数据。 对于这种访问模式，最近引用的频率不是未来引用概率的良好预测指标，因此像LRU和CLOCK这样的操作系统页面替换方案在许多数据库访问模式下表现不佳。 提出了各种替代方案，包括一些尝试通过查询执行计划信息来调整替换策略的方案。 如今，大多数系统使用对LRU方案进行简单增强来处理全表扫描的情况。 在研究文献中出现并在商业系统中得到实现的一种方案是LRU-2。另一个方案

在商业系统中使用的一种替换策略是根据页面类型来决定，例如，B+-树的根节点可能会采用与堆文件中的页面不同的策略。 这让人想起了Reiter的领域分离方案[15, 75]。

最近的硬件趋势，包括64位寻址和内存价格下降，使得非常大的缓冲池在经济上成为可能。
这为利用大容量主存储器提供了新的机会，以提高效率。 与此相反，一个大型且非常活跃的缓冲池也带来了重启恢复速度和高效检查点等其他问题。 这些主题将在第6节进一步讨论。

5.4标准做法

在过去的十年中，商业文件系统已经发展到可以很好地支持数据库存储系统的程度。 在标准使用模型中，系统管理员在DBMS的每个磁盘或逻辑卷上创建一个文件系统。然后，DBMS通过像mmap套件这样的低级接口，在每个文件系统中分配一个单独的大文件，并控制数据在该文件中的位置。 DBMS基本上将每个磁盘或逻辑卷视为一个（几乎）连续的数据库页面线性数组。 在这种配置中，现代文件系统为DBMS提供了合理的空间和时间控制，这种存储模型在几乎所有数据库系统实现中都可用。 原始磁盘支持仍然是大多数数据库系统中常见的高性能选项，然而，它的使用正在迅速缩小，仅限于性能基准测试。

5.5 讨论和附加材料

数据库存储子系统是一项非常成熟的技术，但近年来出现了一些新的考虑因素，这些因素有可能改变数据管理技术的多个方面。

一个关键的技术变革是闪存作为一种经济可行的随机访问持久存储技术的出现[28]。
自数据库系统研究的早期以来，就一直存在着

讨论由新的存储技术取代磁盘而引起的数据库管理系统设计的重大变革。 闪存似乎在技术上是可行的，并且得到了广泛市场的支持，它在成本/性能方面呈现出有趣的中间折衷。闪存是三十多年来首个在这方面取得成功的新型持久存储介质，因此它的细节可能对未来的数据库管理系统设计产生重大影响。

最近又出现了另一个传统话题，即数据库数据的压缩。 该话题的早期研究集中在磁盘上的压缩，以最小化读取时的磁盘延迟，并最大化数据库缓冲池的容量。 随着处理器性能的提高和RAM延迟的不跟上，即使在计算过程中也越来越重要考虑保持数据的压缩，以最大化数据在处理器缓存中的驻留。 但这需要适用于数据处理的压缩表示形式，以及操作压缩数据的查询处理内部。 关系数据库压缩中的另一个问题是数据库是可重新排序的元组集合，而大多数压缩工作都集中在不考虑重新排序的字节流上。

对于数据库压缩，最近的研究表明未来有很大的潜力[73]。

最后，在传统关系数据库市场之外，对于大规模但稀疏数据存储技术引起了增强的兴趣，其中逻辑上有数千列，对于任何给定的行，大部分列都为空。这些场景通常通过某种属性-值对或三元组的集合来表示。 实例包括Google的BigTable [9]，Microsoft的Active Directory和Exchange产品使用的标记列，以及为“语义Web”提出的资源描述框架（RDF）。这些方法的共同之处是使用以数据表的列为基础来组织磁盘的存储系统，而不是行。 列导向存储的想法在最近的一些数据库研究中得到了重新提起和详细探讨[36, 89, 90]。

6

事务：并发控制和恢复

数据库系统经常被指责为庞大的、整体的软件系统，无法分割成可重用的组件。 实际上，数据库系统以及实施和维护它们的开发团队是可以分解为独立的组件，并在它们之间有文档化的接口。 这在关系查询处理器和事务存储引擎之间的接口尤其如此。 在大多数商业系统中，这些组件由不同的团队编写，并且它们之间有明确定义的接口。

DBMS的真正整体部分通常包括四个紧密交织的组件：

1. 用于并发控制的锁管理器。
2. 用于恢复的日志管理器。
3. 用于数据库I/O缓冲的缓冲池。
4. 用于在磁盘上组织数据的访问方法。

关于数据库系统中事务存储算法和协议的细节已经有很多文字描述。 希望了解这些系统的读者至少应该阅读基本的本科数据库

教科书[72]，关于ARIES日志协议的期刊文章[59]，以及至少一篇关于事务索引并发和日志记录的重要文章[46, 58]。 更高级的读者会想要参考Gray和Reuter关于事务的教科书[30]。 要真正成为专家，需要在阅读后进行实现的努力。 我们在这里不详细讨论算法和协议，而是概述这些不同组件的角色。 我们专注于系统基础设施，这在教科书中经常被忽视，强调了导致简单协议可行性的各个组件之间的相互依赖关系的复杂性和微妙之处。

6.1 关于ACID的说明

许多人熟悉“ACID事务”这个术语，这是由Haerder和Reuter [34]提出的助记符。 ACID代表原子性、一致性、隔离性和持久性。 这些术语没有被正式定义，并且不是数学公理，不能保证事务一致性。 因此，仔细区分这些术语及其关系并不重要。 但尽管是非正式的，ACID缩写对于组织事务系统的讨论是有用的，并且非常重要，因此我们在这里进行回顾：

- 原子性是事务的“全有或全无”保证-事务的所有操作要么全部提交，要么全部不提交。

- 一致性是应用程序特定的保证；SQL完整性约束通常用于在DBMS中捕获这些保证。根据一组约束的一致性定义，只有在事务将数据库保持在一致状态时才能提交。

- 隔离性是对应用程序编写者的保证，即两个并发事务不会看到彼此的未提交更新。 因此，应用程序无需编写“防御性”代码来担心其他并发事务的“脏数据”；它们可以像程序员独占数据库一样编码。

- 持久性是已提交事务的更新在数据库中对后续事务可见的保证，独立于后续硬件或软件错误，直到被另一个已提交事务覆盖为止。

粗略地说，现代数据库管理系统通过锁定协议实现隔离。 持久性通常通过日志记录和恢复来实现。 隔离性和原子性通过锁定（以防止瞬态数据库状态的可见性）和日志记录（以确保可见数据的正确性）的组合来保证。 一致性由查询执行器中的运行时检查来管理：如果事务的操作违反了SQL完整性约束，则事务将被中止并返回错误代码。

6.2串行化的简要回顾

我们通过简要回顾数据库并发控制的主要目标开始讨论事务，并在下一节中描述了用于在大多数多用户事务存储管理器中实现此概念的两个最重要的构建块：锁定和闩锁。

串行化是并发事务的明确定义的正确性概念。 它规定多个提交事务的交错操作序列必须对应于某些事务的串行执行 - 就好像根本没有并行执行一样。串行化是描述一组事务所期望行为的一种方式。隔离性是从单个事务的角度来看的相同概念。 如果事务不看到任何并发异常，则称其在隔离中执行 - 这是ACID的“I”部分。串行化由DBM*S*并发控制模型强制执行。

并发控制执行有三种广泛的技术。
这些在教科书和早期调查报告中有很好的描述[7]，但是
我们在这里简要回顾一下：

1.严格的两阶段锁定（*2PL*）：事务在读取数据之前会获取共享锁，并且

在写入数据之前会获取独占锁。所有的锁都会一直持有，直到事务结束时才会原子性地释放。 在等待获取锁的过程中，事务会被阻塞在等待队列上。

2.多版本并发控制 $(MVCC)$：事务不持有锁，而是在过去的某个时间点上保证了对数据库状态的一致性视图，即使在那个固定时间点之后行发生了变化。

3.乐观并发控制 (OCC)：允许多个事务读取和更新一个项目而不会被阻塞。相反，事务会维护它们的读取和写入历史记录，并在提交事务之前检查它们的历史记录以查找可能发生的隔离冲突；如果发现任何冲突，其中一个冲突的事务将被回滚。

大多数商业关系型数据库管理系统通过2PL实现完全串行化。 锁管理器是负责提供2PL功能的代码模块。

为了减少锁定和锁冲突，一些数据库管理系统支持MVCC或OCC，通常作为2PL的附加功能。 在MVCC模型中，不需要读锁，但这往往会以不提供完全串行化为代价，我们将在第4.2.1节中讨论。为了避免写操作被读操作阻塞，写操作可以在前一版本的行被保存或者保证可以快速获取之后继续进行。 正在进行的读事务将继续使用之前的行值，就好像它被锁定并且无法更改一样。 在商业MVCC实现中，稳定的读取值被定义为读事务开始时的值或者该事务最近的SQL语句开始时的值。

虽然 OCC 避免了对锁的等待，但在事务之间发生真正的冲突时可能会导致更高的惩罚。 在处理跨事务的冲突时，OCC 类似于 2PL，只是将 2PL 中的锁等待转换为事务回滚。 在冲突不常见的情况下，OCC 的性能非常好，避免了过于保守的等待时间。 然而，在频繁冲突的情况下，过多的回滚和重试会对性能产生负面影响，使其成为一个不好的选择[2]。

过多的回滚和重试会对性能产生负面影响，使其成为一个不好的选择[2]。

6.3 锁定和锁定

数据库锁定只是系统内部约定使用的名称，用于表示 DBMS 管理的物理项（例如磁盘页）或逻辑项（例如元组、文件、卷）。 请注意，任何名称都可以与锁相关联 - 即使该名称表示一个抽象概念。 锁定机制只是提供了一个注册和检查这些名称的位置。 每个锁都与一个事务关联，每个事务都有一个唯一的事务 ID。锁有不同的锁定模式，并且这些模式与锁定模式兼容性表相关联。 在大多数系统中，这种逻辑是基于 Gray 在锁的粒度上引入的众所周知的锁定模式的论文[29]。 该论文还解释了商业系统中如何实现分层锁定。 分层锁定允许使用单个锁来锁定整个表，并同时有效且正确地支持表中的行粒度锁定。

锁管理器支持两个基本调用：`lock(lockname,transactionID,mode)`和`remove transaction(transaction-ID)`。 请注意，由于严格的2PL协议，不应该单独调用解锁资源的方法 - remove transaction()调用将解锁与事务相关的所有资源。 然而，正如我们在5.2.1节中讨论的那样，SQL标准允许较低程度的事务隔离，因此还需要一个`unlock(lockname,transactionID)`调用。 还有一个`lock upgrade(lockname, transactionID,newmode)`调用，允许事务以两阶段的方式“升级”到更高的锁模式（例如，从共享模式到独占模式），而无需放弃和重新获取锁。 此外，一些系统还支持 conditional lock (lockname, transactionID, mode) 调用。 条件锁()调用总是立即返回，并指示是否成功获取锁。 如果没有成功，调用的DBMS线程不会排队等待锁。 关于索引并发性的条件锁使用在[60]中讨论。

为了支持这些调用，锁管理器维护两个数据结构。 维护一个全局锁表来保存锁名称及其相关信息。 锁表是一个动态哈希表，以锁名称为键进行索引。 每个锁都有一个与之关联的模式标志，用于指示锁的模式，以及一个等待队列，其中包含锁请求的配对（事务ID，模式）。 此外，锁管理器还维护一个以事务ID为键的事务表，其中包含每个事务 T的两个项目：（1）指向 T的DBMS线程状态的指针，以便在获取任何等待的锁时重新调度 T的DBMS线程，以及（2）指向锁表中 T的所有锁请求的指针列表，以便在需要时删除与特定事务相关的所有锁（例如，在事务提交或中止时）。

在内部，锁管理器使用一个死锁检测器 DBMS线程，定期检查锁表以检测等待循环（DBMS工作线程之间的循环，每个线程都在等待下一个线程，形成一个循环）。 一旦检测到死锁，死锁检测器会中止其中一个死锁事务。 选择中止哪个死锁事务是基于已经在研究文献中进行研究的启发式算法 [76]。 在共享无关和共享磁盘系统中，需要分布式死锁检测 [61] 或更原始的基于超时的死锁检测器。 关于锁管理器实现的更详细描述可以在Gray和Reuter的著作 [30] 中找到。

作为数据库锁的辅助，还提供了更轻量级的互斥锁。 互斥锁更类似于监视器 [41] 或信号量，它们用于提供对内部DBMS数据结构的独占访问。 例如，缓冲池页面表中的每个帧都有一个与之关联的互斥锁，以确保在任何时候只有一个DBMS线程替换给定的帧。 互斥锁用于实现锁和短暂稳定内部数据结构，这些数据结构可能正在同时修改。

闩锁与锁在许多方面有所不同：

- 锁存储在锁表中，并通过哈希表定位；闩锁驻留在靠近它们保护的资源的内存中，并通过直接寻址访问。

- 在严格的2PL实现中，锁受到严格的2PL协议的约束。 根据特殊情况的内部逻辑，可以在事务期间获取或释放闩锁。

- 锁获取完全由数据访问驱动，因此锁获取的顺序和生命周期在很大程度上由应用程序和查询优化器控制。 闩锁由DBMS内部的专用代码获取，DBMS内部代码会策略性地发出闩锁请求和释放。

- 允许锁产生死锁，并通过事务重新启动来检测和解决锁死锁。 必须避免闩锁死锁；闩锁死锁的发生表示DBMS代码中存在错误。

- 闩锁是使用原子硬件指令或者在罕见情况下使用操作系统内核中的互斥实现的。

- 闩锁调用最多需要几十个CPU周期，而锁请求需要数百个CPU周期。
- 锁管理器跟踪事务持有的所有锁，并在事务抛出异常时自动释放锁，但是操作闩锁的内部DBMS例程必须仔细跟踪它们并包括手动清理作为异常处理的一部分。

- 闩锁不被跟踪，因此如果任务出错，无法自动释放。

闩锁API支持以下例程：闩锁(对象，模式)、解闩(对象)、条件闩锁(对象，模式)。 在大多数DBMS中，闩锁模式只包括共享或独占。

闩锁维护一种模式，并且有一个等待队列，其中包含等待闩锁的DBMS线程。 闩锁和解闩调用的工作方式与预期相同。 条件闩锁调用类似于上面描述的条件锁调用，并且还用于索引并发性[60]。

6.3.1 事务隔离级别

在事务概念的早期开发阶段，尝试通过提供“较弱”的语义来增加并发性

而不是串行化。 挑战是在这些情况下提供强大的语义定义。 在这方面最有影响力的工作是Gray早期关于“一致性程度”的研究[29]。 该工作试图提供一致性程度的声明性定义，并以锁定为基础进行实现。 受到这项工作的影响，ANSI SQL标准定义了四个“隔离级别”：

1.读未提交：事务可以读取任何版本的数据，无论是否提交。 在锁定实现中，通过读取请求而不获取任何锁来实现。[1]

2.读已提交：事务可以读取任何已提交的数据版本。 重复读取对象可能导致不同的（已提交的）版本。 这是通过在访问对象之前获取读锁，并在访问后立即解锁来实现的。

3.可重复读取：事务只会读取已提交数据的一个版本；一旦事务读取了一个对象，它将始终读取该对象的相同版本。

这是通过在访问对象之前获取读锁，并在事务结束之前一直持有该锁来实现的。

4.可串行化：保证完全可串行化访问。

乍一看，可重复读取似乎提供了完全可串行化，但事实并非如此。 在System R项目的早期阶段，出现了一个被称为“幻象问题”的问题。 在幻象问题中，一个事务在同一个事务中多次访问一个关系，并且在重新访问时看到了之前访问时没有看到的新的“幻象”元组。 这是因为以元组级别的两阶段锁定无法阻止向表中插入新的元组。 表级别的两阶段锁定可以防止幻象问题，但在某些情况下可能会有限制。

[1]在所有隔离级别中，写请求之前都会有写锁，直到事务结束。

其中事务只通过索引访问少量元组。 我们在第5.4.3节进一步研究了这个问题，讨论了索引中的锁定。

商业系统通过基于锁定的并发控制实现提供了上述四个隔离级别。不幸的是，正如Berenson等人[6]所指出的，早期的Gray的工作和ANSI标准都没有实现提供真正声明性定义的目标。 两者都以微妙的方式依赖于假设使用锁定方案进行并发控制，而不是乐观[47]或多版本[74]的并发方案。 这意味着提出的语义是不明确的。 鼓励感兴趣的读者阅读Berenson的论文，该论文讨论了SQL标准规范中的一些问题，以及Adya等人[1]的研究，该研究提供了一种新的、更清晰的方法来解决这个问题。

除了标准的ANSI SQL隔离级别外，各个供应商还提供了其他在特定情况下被证明受欢迎的级别。

- 游标稳定性：该级别旨在解决“丢失更新”问题。考虑两个事务$T1$和$T2$。$T1$以读提交模式运行，读取一个对象X（比如银行账户的余额），记住其值，并随后根据记住的值写入对象X（比如将原始账户余额增加100美元）。 $T2$也读取和写入X（比如从账户中减去300美元）。 如果$T2$的操作发生在$T1$的读取和$T1$的写入之间，那么$T2$的更新效果将会丢失 - 在我们的例子中，账户的最终值将增加100美元，而不是按预期减少200美元。 以游标稳定性模式运行的事务在查询游标上持有最近读取的项的锁定；当游标移动（例如通过另一个FETCH）或事务终止时，锁定会自动释放。

  CURSOR STABILITY允许事务在不受其他事务干扰的情况下对单个项目进行读取-思考-写入序列。

- 快照隔离：以事务开始时数据库的版本为基础运行的事务；其他事务的后续更新对该事务不可见。 这是生产数据库系统中多版本并发控制的主要用途之一。 事务启动时，从单调递增计数器获取唯一的开始时间戳；提交时从计数器获取唯一的结束时间戳。 只有在没有其他具有重叠的开始/结束事务对写入了该事务也写入的数据的情况下，事务才会提交。 这种隔离模式依赖于多版本并发实现，而不是锁定。 然而，在支持快照隔离的系统中，通常同时存在这些方案。

- 读一致性：这是*O*racle定义的一个MV*CC*方案，与快照隔离稍有不同。在*O*racle方案中，每个*S*QL语句（一个事务中可能有多个）在语句开始时看到的是最近提交的值。 对于从游标中获取的语句，游标集是基于打开时的值。 这是通过维护单个元组的多个逻辑版本来实现的，一个事务可能引用单个元组的多个版本。 *O*racle不存储可能需要的每个版本，而只存储最近的版本。 如果需要旧版本，它通过应用undo日志记录将当前版本“回滚”以产生旧版本。 通过长期写锁来维护修改，因此当两个事务想要写入同一个对象时，第一个写入者“获胜”，第二个写入者必须等待第一个写入者的事务完成后才能进行写入。 相比之下，在快照隔离中，第一个提交者“获胜”，而不是第一个写入者。

弱隔离方案可以提供比完全串行化更高的并发性。 因此，一些系统甚至使用弱一致性。

作为默认设置。 例如，Microsoft SQL Server默认为读取提交。缺点是隔离性（按ACID定义）不能保证。 因此，应用程序编写者需要考虑方案的细微差别，以确保其事务正确运行。

鉴于方案的操作定义语义，这可能很棘手，并且可能导致应用程序在不同的DBMS之间移动更加困难。

6.4 日志管理器

日志管理器负责维护已提交事务的持久性，促进中止事务的回滚以确保原子性，并从系统故障或非正常关闭中恢复。 为了提供这些功能，日志管理器在磁盘上维护一系列日志记录，并在内存中维护一组数据结构。 为了在崩溃后支持正确的行为，显然需要从日志和数据库的持久数据重新创建内存中的数据结构。

数据库日志是一个非常复杂和注重细节的主题。
关于数据库日志的规范参考是关于ARIES [59]的期刊论文，数据库专家应该熟悉该论文的细节。 ARIES论文不仅解释了日志协议，还提供了替代设计可能性的讨论以及可能引起的问题。 这使得阅读变得密集，但最终是有回报的。 作为更易理解的介绍，Ramakrish-nan和Gehrke的教材[72]提供了关于基本ARIES协议的描述，没有涉及其他讨论或改进。 在这里，我们讨论一些恢复的基本思想，并试图解释教材和期刊描述之间的复杂差距。

数据库恢复的标准主题是使用预写式
日志（WAL）协议。 WAL协议由三个非常
简单的规则组成：

1. 对数据库页面的每个修改都应生成一个日志记录，并且在刷新数据库页面之前必须将日志记录刷新到日志设备中。

2. 数据库日志记录必须按顺序刷新；在刷新之前，无法刷新日志记录 r之前的所有日志记录。
3. 在事务提交请求时，必须将提交日志记录刷新到日志设备在提交请求成功返回之前。

许多人只记得这三个规则中的第一个，但是所有三个规则都需要以确保正确的行为。

第一个规则确保在事务中止时可以撤销不完整事务的操作，以确保原子性。
规则（2）和（3）的组合确保持久性：在系统崩溃后，已提交事务的操作可以在数据库中反映之前重新执行。

鉴于这些简单的原则，令人惊讶的是，高效的数据库日志记录如此微妙和详细。然而，在实践中，上述简单故事由于需要极高的性能而变得复杂。
挑战在于保证事务提交的“快速路径”效率，同时为中止的事务提供高性能回滚和崩溃后快速恢复。 当添加应用程序特定的优化时，日志记录变得更加复杂，例如支持仅能递增或递减的字段的改进性能（“托管交易”）。

为了最大限度地提高快速路径的速度，大多数商业数据库系统采用Haerder和Reuter称之为“DIRECT，STEAL/NOT-FORCE”的模式：（a）数据对象在原地更新，（b）未固定的缓冲池帧可以被“窃取”（并将修改后的数据页写回磁盘），即使它们包含未提交的数据，（c）缓冲池页面在提交请求返回给用户之前不需要被“强制”（刷新）到数据库。 这些策略使数据保持在DBA选择的位置，并且它们使缓冲管理器和磁盘调度器能够自由决定内存管理和I/O策略，而不考虑事务正确性。 这些功能可以带来重大的性能优势，但需要日志管理器有效处理所有被中止事务窃取页面的刷新的微妙之处，并重新执行未强制页面的已提交事务的更改。

在崩溃时丢失。 一种由一些DBMS使用的优化方法是将DIRECT、STEAL/NOT-FORCE系统的可扩展性优势与DIRECT NOT-STEAL/NOT-FORCE系统的性能相结合。 在这些系统中，只有在缓冲池中没有干净的页面时，才会被窃取页面，此时系统会退化为具有上述额外开销的STEAL策略。

日志记录中的另一个快速路径挑战是尽可能使日志记录尺寸小，以增加日志I/O活动的吞吐量。
一种自然的优化是记录逻辑操作（例如，“将（Bob，$25000）插入EMP”），而不是物理操作（例如，通过元组插入修改的所有字节范围的后图像，包括堆文件和索引块上的字节）。这种权衡是，重做和撤销逻辑操作的逻辑变得非常复杂。 这可能会严重影响事务中止和数据库恢复的性能。[2]实际上，通常使用物理和逻辑日志记录（称为“生理”日志记录）的混合方式。 在ARIES中，通常使用物理日志记录来支持REDO，使用逻辑日志记录来支持UNDO。这是ARIES规则的一部分，在恢复过程中“重复历史”，然后从该点回滚事务。

在系统故障或非正常关闭后，需要进行崩溃恢复以将数据库恢复到一致的状态。 如上所述，恢复理论上是通过重放历史记录并逐步遍历日志记录，从第一个记录到最近的记录来实现的。 这种技术是正确的，但由于日志可能任意长，效率不高。 与其从第一个日志记录开始恢复，不如从这两个日志记录中最旧的记录开始恢复：(1) 描述缓冲池中最旧脏页的最早更改的日志记录，以及(2) 表示系统中最旧事务开始的日志记录。

这个点的序列号被称为恢复日志序列号（recovery LSN）。由于计算和记录恢复LSN会产生开销，并且我们知道恢复LSN是

[2]还要注意的是，逻辑日志记录必须始终具有已知的反函数，如果它们需要参与撤消处理。

由于单调递增，我们不需要始终将其更新到最新状态。
相反，我们在称为检查点的周期间隔内计算它。

一个天真的检查点将强制所有脏缓冲池页面，并计算并存储恢复LSN。对于大型缓冲池，这可能导致延迟几秒钟来完成待处理页面的I/O。因此，需要一种更高效的“模糊”检查点方案，以及逻辑来正确将检查点带到最新的一致状态，尽可能少地处理日志。ARIES使用了一种非常聪明的方案，实际的检查点记录非常小，只包含足够的信息来启动日志分析过程，并在崩溃时重新创建丢失的主内存数据结构。在ARIES模糊检查点期间，计算恢复LSN，但不需要同步写出缓冲池页面。使用单独的策略确定何时异步写出旧的脏缓冲池页面。

请注意，回滚操作需要写入日志记录。这可能导致困难的情况，即正在进行的事务由于日志空间耗尽而无法继续进行，但也无法回滚。通常通过空间预留方案来避免这种情况，然而，随着系统在多个版本中的演变，这些方案很难得到并保持正确。

最后，日志记录和恢复的任务进一步复杂化，因为数据库不仅仅是磁盘上的一组用户数据元组，还包括各种“物理”信息，使其能够管理其内部基于磁盘的数据结构。我们将在下一节中讨论索引日志记录的上下文。

6.5 索引中的锁定和日志记录

索引是用于访问数据库中的数据的物理存储结构。索引本身对于数据库应用程序开发人员来说是不可见的，除非它们提高性能或强制唯一性约束。开发人员和应用程序无法直接观察或操作索引中的条目。这使得可以通过更高效（和复杂）的事务方案来管理索引。

索引并发性和恢复所需的唯一不变量是

服务的唯一要求是索引始终返回数据库中的事务一致的元组。

6.5.1 B+-树中的锁定

B+-树锁定中存在一个经过深入研究的问题的例子。 B+-树由通过缓冲池访问的数据库磁盘页面组成，就像数据页面一样。 因此，一种用于索引并发控制的方案是在索引页面上使用两阶段锁定。 这意味着每个触及索引的事务都需要在B+-树的根上锁定，直到提交时间 - 这是有限并发性的一种方法。 已经开发了各种基于锁的方案来解决这个问题，而不在索引页面上设置任何事务锁定。 这些方案的关键见解是树的物理结构的修改（例如，页面分割）可以以非事务性的方式进行，只要所有并发事务继续在叶子节点上找到正确的数据。 大致有三种方法：

- 保守方案：只有在可以保证不冲突使用页面内容的情况下，允许多个事务访问相同的页面。 其中一种冲突是一个读取事务想要遍历树的一个完全填充的内部页面，而一个并发插入事务正在该页面下方操作，并且可能需要拆分该页面[4]。 与下面的较新想法相比，这些保守方案牺牲了太多的并发性。

- 锁耦合方案：树遍历逻辑在访问每个节点之前锁定该节点，在成功锁定下一个要访问的节点之后才解锁该节点。 这种方案有时被称为锁“螃蟹”，因为它在树中“持住”一个节点，“抓住”它的子节点，释放父节点，并重复这个过程。 锁耦合在一些商业系统中使用；IBM的ARIES-IM版本有很好的描述[60]。 ARIES-IM包括一些相当复杂的细节和边界情况 - 有时它必须在拆分后重新启动遍历，甚至设置（非常短期的）整个树的锁。

- 右链接方案：在B+-树中添加简单的附加结构，以最小化对锁定和重新遍历的需求。 特别地，从每个节点到其右侧邻居添加了一个链接。 在遍历过程中，右链接方案不进行锁定耦合 - 每个节点都被锁定、读取和解锁。 右链接方案的主要思想是，如果遍历事务跟随指向节点n的指针，并发现n在期间被分裂，遍历事务可以检测到这一事实，并通过右链接“向右移动”，以在树中找到新的正确位置[46, 50]。 一些系统还支持使用反向链接进行反向遍历。

Kornacker等人[46]对锁定耦合和右链接方案之间的区别进行了详细讨论，并指出锁定耦合仅适用于B+-树，并且对于更复杂的数据（例如，没有单一线性顺序的地理数据）的索引树无效。 PostgreSQL的广义搜索树（GiST）实现基于Kornacker等人的可扩展右链接方案。

6.5.2 物理结构的日志记录

除了特殊情况下的并发逻辑外，索引还使用特殊情况下的日志记录逻辑。 这种逻辑使得日志记录和恢复更加高效，但代码复杂度增加。 主要思想是当相关事务被中止时，结构索引的变化不需要被撤销；这样的变化通常对其他事务所看到的数据库元组没有影响。 例如，如果在插入事务期间对B+-树页面进行了拆分，随后该事务被中止，那么在中止处理过程中没有迫切需要撤销拆分。

这就提出了将某些日志记录标记为仅重做的挑战。
在对日志进行任何撤销处理时，可以保留仅重做的更改。 ARIES为这些场景提供了一种优雅的机制，称为嵌套顶级操作，允许恢复过程在恢复期间“跳过”物理结构修改的日志记录，而无需任何特殊情况的代码。

这个相同的想法在其他情境中也被使用，包括堆文件。 插入到堆文件中可能需要在磁盘上扩展文件。
为了捕捉这个，必须对文件的范围映射进行更改。 这是一个在磁盘上的数据结构，指向构成文件的连续块的运行。 如果插入事务中止，对范围映射的这些更改不需要撤消。 文件变大的事实是一个在事务上不可见的副作用，实际上可能对吸收未来的插入流量有用。

6.5.3 下一个键锁定：物理替代品用于逻辑属性

我们在本节中以一个细微但重要的想法来结束索引并发问题的讨论。 挑战在于提供完全的可串行化性（包括幻像保护），同时允许元组级锁和索引的使用。 请注意，这种技术仅适用于完全的可串行化性，并不是放松隔离模型所必需或使用的。

幽灵问题可能在事务通过索引访问元组时出现。 在这种情况下，事务通常不会锁定整个表，只会锁定通过索引访问的表中的元组（例如，“Name BETWEEN 'Bob' AND 'Bobby'”）。 在没有表级锁的情况下，其他事务可以自由地向表中插入新的元组（例如，“Name='Bobbie'”）。 当这些新插入的元组落在查询谓词的值范围内时，它们将在后续通过该谓词的访问中出现。 请注意，幽灵问题涉及到数据库元组的可见性，因此它是锁的问题，而不仅仅是闩锁的问题。 原则上，需要的是能够以某种方式锁定原始查询的搜索谓词所表示的逻辑空间，例如，在字典顺序中介于“Bob”和“Bobby”之间的所有可能字符串的范围。 不幸的是，谓词锁定是昂贵的，因为它需要一种比较任意谓词重叠的方法。 这不能通过基于哈希的锁表来实现[3]。

在B+-树中，解决幽灵问题的一种常见方法被称为下一个键锁定。 在下一个键锁定中，索引插入代码被修改，以便使用索引键 k插入元组时必须分配一个

独占锁定下一个键元组，该元组存在于索引中，且其键值大于 k。 该协议确保后续的插入不能出现在之前返回给活动事务的两个元组之间。 它还确保不能在之前返回的最低键元组下方插入元组。 例如，在第一次访问中未找到“Bob”键，则在同一事务中的后续访问中也不应该找到。 还有一种情况：插入元组到上方之前返回的最高键元组。 为了防止这种情况发生，下一个键锁定协议要求读事务也要在索引中获取下一个键元组的共享锁。 在这种情况下，下一个键元组是不满足查询谓词的最小键元组。 逻辑上，更新操作相当于先删除再插入，尽管优化也是可能的和常见的。

尽管下一个键锁定是有效的，但它会遭受过度锁定的问题这在某些工作负载下可能会有问题。 例如，如果我们正在从键1扫描记录到键10，但是被索引的列只存储了键1、5和100，那么整个范围从1到100都将被读锁定，因为100是10之后的下一个键。

下一个键锁定不仅仅是一个巧妙的技巧。 这是一个使用物理对象（当前存储的元组）作为逻辑概念（谓词）的代理的例子。 好处是可以使用简单的系统基础设施，如基于哈希的锁表，用于更复杂的目的，只需修改锁协议。 当有这样的语义信息可用时，设计复杂软件系统的设计者应该将这种逻辑代理的一般方法保留在他们的“技巧包”中。

6.6 事务存储的相互依赖性

我们在本节早期声称，事务存储系统是单体的，紧密交织的系统。 在本节中，我们讨论了事务存储系统的三个主要方面之间的一些相互依赖关系：并发控制、恢复管理和访问方法。 在一个更幸福的世界中，可能可以在这些模块之间识别出狭窄的API，允许这些API背后的实现可以互换。 本节中的示例表明，这并不容易实现。 我们不打算提供详尽的

相互依赖性的列表；生成和证明这样一个列表的完整性将是一个非常具有挑战性的练习。 然而，我们希望能够说明事务存储的一些扭曲逻辑，并由此来证明商业DBMS中产生的单体实现的合理性。

我们首先考虑并发控制和恢复，而不考虑访问方法的进一步复杂性。 即使在这种简化情况下，组件之间也是紧密交织的。 并发和恢复之间关系的一个体现是，预写式日志对于锁定协议有隐含的假设。预写式日志需要严格的两阶段锁定，并且不能与非严格的两阶段锁定一起正确运行。 为了看到这一点，考虑一个中止事务的回滚过程中会发生什么。 恢复代码开始处理中止事务的日志记录，撤销其修改。 通常，这需要更改事务先前修改的页面或元组。 为了进行这些更改，事务需要对这些页面或元组进行锁定。 在非严格的2PL方案中，如果事务在中止之前释放任何锁定，它可能无法重新获取完成回滚过程所需的锁定。

访问方法进一步复杂化了事情。 将教科书访问方法算法（例如，线性哈希[53]或R树[32]）转化为事务系统中的正确、高并发、可恢复版本是一个重要的知识和工程挑战。因此，大多数领先的数据库管理系统仍然只实现堆文件和B+-树作为受事务保护的访问方法；PostgreSQL的GiST实现是一个值得注意的例外。 正如我们上面为B+-树所示，事务性索引的高性能实现包括复杂的锁定、锁定和日志记录协议。

严肃的数据库管理系统中的B+-树充斥着对并发性和恢复代码的调用。 即使像堆文件这样的简单访问方法也存在一些围绕描述其内容的数据结构（例如，扩展映射）的复杂并发性和恢复问题。 这种逻辑并不适用于所有访问方法 - 它非常定制于访问方法的特定逻辑及其特定实现。

对于访问方法的并发控制，锁定导向的方案已经得到了很好的发展。 其他并发方案（例如，

乐观或多版本并发控制）通常不考虑访问方法，或者只是随意提及它们而不切实际[47]。因此，对于给定的访问方法实现，混合和匹配不同的并发机制是困难的。

访问方法中的恢复逻辑特别依赖于系统特定的内容：访问方法日志记录的时间和内容取决于恢复协议的细节，包括结构修改的处理（例如，它们是否在事务回滚时被撤销，如果不是如何避免），以及物理和逻辑日志的使用。 即使对于像B+-树这样的特定访问方法，恢复和并发逻辑也是相互交织的。 在一个方向上，恢复逻辑取决于并发协议：如果恢复管理器必须恢复树的物理一致状态，则需要知道可能出现的不一致状态，以便通过日志记录来保证原子性（例如，通过嵌套顶级操作）。 在相反的方向上，访问方法的并发协议可能依赖于恢复逻辑。 例如，B+-树的右链接方案假设树中的页面在分裂后不会再次合并。 这个假设要求恢复方案使用诸如嵌套顶级操作之类的机制来避免撤销由中止事务生成的分裂。

这个图片中唯一的亮点是缓冲管理相对于存储管理器的其他组件来说是相对独立的。 只要页面被正确固定，缓冲管理器就可以自由地封装其余的逻辑并根据需要重新实现它。

例如，缓冲管理器在选择要替换的页面时具有自由度（由于STEAL属性），以及页面刷新的调度（由于NOT FORCE属性）。 当然，实现这种隔离性是并发性和恢复性复杂性的直接原因。 因此，这个亮点可能没有看起来那么明亮。

6.7 标准实践

今天的所有生产数据库都支持ACID事务。 作为一个规则，它们使用预写日志来保证持久性，并使用两阶段锁定来进行并发控制。

并发控制。 一个例外是PostgreSQL，它在整个系统中使用多版本并发控制。 Oracle开创了多版本并发控制与锁定并存的有限使用，作为提供松散一致性模型（如快照隔离和读一致性）的方式；这些模式在用户中的流行导致它们在多个商业DBMS中被采用，并且在Oracle中这是默认设置。 B+-树索引在所有生产数据库中都是标准的，大多数商业数据库引擎都提供一些形式的多维索引，无论是嵌入在系统中还是作为“插件”模块。 只有PostgreSQL通过其GiST实现提供高并发的多维和文本索引。

MySQL在积极支持各种不同的存储管理器方面是独一无二的，以至于数据库管理员经常为同一个数据库中的不同表选择不同的存储引擎。 它的默认存储引擎MyISAM只支持表级锁定，但被认为是适用于大多数只读工作负载的高性能选择。 对于读/写工作负载，推荐使用InnoDB存储引擎；它提供了行级锁定。 （InnoDB几年前被Oracle收购，但目前仍然是开源和免费的。）MySQL的这两个存储引擎都没有提供System R [29]中开发的众所周知的分层锁定方案，尽管它在其他数据库系统中被广泛使用。 这使得MySQL数据库管理员在InnoDB和MyISAM之间的选择变得棘手，在某些混合工作负载情况下，两个引擎都无法提供良好的锁粒度，需要数据库管理员使用多个表和/或数据库复制来开发物理设计，以支持扫描和高选择性索引访问。 MySQL还支持基于主内存和基于集群的存储引擎，并且一些第三方供应商已经宣布了与MySQL兼容的存储引擎，但目前MySQL用户群体中的大部分精力都集中在MyISAM和InnoDB上。

6.8 讨论和附加材料

事务机制现在是一个非常成熟的话题，多年来已经尝试了各种可能的技巧；新的设计往往涉及排列组合

现有思想的一部分。 在这个领域，最明显的变化可能是由于RAM价格的迅速下降。这增加了将数据库的“热”部分保持在内存中并以内存速度运行的动力，这使得将数据刷新到持久存储器的挑战变得更加复杂，以保持重启时间低。闪存在事务管理中的作用是这种不断演变的平衡的一部分。

近年来的一个有趣发展是操作系统社区相对广泛地采用了预写式日志，通常在日志文件系统的框架下。 这些已经成为几乎所有操作系统的标准选项。 由于这些文件系统通常仍然不支持文件数据上的事务，有趣的是看到它们如何以及在哪里使用预写式日志来实现持久性和原子性。 对于进一步阅读，可以参考[62, 71]。在这方面的另一个有趣方向是关于 *Stasis*[78]的工作，它试图更好地模块化ARIES风格的日志记录和恢复，并使其可用于系统程序员进行各种用途。

7

共享组件

在本节中，我们将介绍一些几乎所有商业数据库管理系统中存在但很少在文献中讨论的共享组件和实用工具。

7.1 目录管理器

数据库目录保存了系统中数据的信息，是一种元数据的形式。 目录记录了系统中基本实体（用户、模式、表、列、索引等）的名称和它们之间的关系，并且本身作为一组表存储在数据库中。 通过将元数据与数据保持相同的格式，系统既更紧凑又更简单易用：用户可以使用相同的语言和工具来查询元数据，就像查询其他数据一样，而用于管理元数据的内部系统代码与管理其他表的代码基本相同。 这种代码和语言的重用是一个重要的教训，在早期的实现中经常被忽视，通常会给开发人员带来重大的遗憾。在过去的十年中，其中一位作者在工业环境中再次目睹了这个错误。

基本目录数据在效率上与普通表有所不同。 目录的高流量部分通常根据需要在主内存中实现，通常以数据结构的形式“去规范化”目录的扁平关系结构，形成一个主内存网络对象。 内存中缺乏数据独立性是可以接受的，因为内存中的数据结构只由查询解析器和优化器以一种规范化的方式使用。

在解析时，附加的目录数据以查询计划的形式缓存，通常以适合查询的非规范化形式。 此外，目录表通常会使用特殊的事务处理技巧来最小化事务处理中的“热点”。

在商业应用程序中，目录可能变得非常庞大。
例如，一个主要的企业资源规划应用程序拥有超过60,000个表，每个表有4到8个列，通常每个表有两到三个索引。

7.2 内存分配器

数据库管理系统内存管理的教科书呈现往往只关注缓冲池。 实际上，数据库系统还为其他任务分配了大量的内存。 正确管理这些内存既是编程负担，也是性能问题。 例如，Selinger风格的查询优化在动态规划过程中可能使用大量内存来构建状态。

像哈希连接和排序这样的查询操作符在运行时分配了大量内存。 商业系统中的内存分配通过使用基于上下文的内存分配器变得更加高效和易于调试。

内存上下文是一种在内存中维护一组连续虚拟内存区域（通常称为内存池）的数据结构。
每个区域可以有一个小的头部，其中包含上下文标签或指向上下文头部结构的指针。 内存上下文的基本API包括以下调用：

- 使用给定的名称或类型创建上下文。 上下文类型可能会建议分配器如何高效处理内存分配。 例如，查询优化器的上下文

通过小增量增长，而哈希连接的上下文则会分配少量大批量的内存。 基于这样的知识，分配器可以选择一次分配更大或更小的区域。

- 在上下文中分配一块内存。 此分配将返回一个指向内存的指针（类似于传统的malloc（）调用）。 该内存可能来自上下文中的现有区域。 或者，如果任何区域中不存在这样的空间，则分配器将向操作系统请求新的内存区域，标记它，并将其链接到上下文中。

- 在上下文中删除一块内存。这可能会或者不会导致上下文删除相应的区域。
  从内存上下文中删除是相当不寻常的。更
  典型的行为是删除整个上下文。
- 删除一个上下文. 首先释放与上下文相关的所有区域，然后删除上下文头。
- 重置一个上下文. 保留上下文，但将其返回到原始创建状态，通常通过释放先前分配的所有内存区域来实现。

内存上下文提供了重要的软件工程优势。 最重要的是，它们作为一种低级别、可由程序员控制的垃圾回收的替代方案。 例如，编写优化器的开发人员可以在优化器上下文中为特定查询分配内存，而不必担心如何在以后释放内存。当优化器选择了最佳计划后，它可以将计划从单独的执行器上下文复制到内存中，然后简单地删除查询的优化器上下文。

这样可以避免编写代码来仔细遍历所有优化器数据结构并删除它们的组件。 它还避免了由于此类代码中的错误而导致的棘手的内存泄漏。这个特性对于查询执行的自然“分阶段”行为非常有用，其中控制从解析器到优化器再到执行器，每个上下文都有一些分配，然后是上下文删除。

请注意，内存上下文实际上比大多数垃圾收集器提供更多的控制，因为开发人员可以控制空间和时间

释放的局部性。 上下文机制本身提供了允许程序员将内存分离为逻辑单元的空间控制。 时间控制是由程序员在适当时候发出上下文删除请求而实现的。 相比之下，垃圾收集器通常在程序的所有内存上工作，并自行决定何时运行。这是在Java中尝试编写服务器质量代码时的挫折之一[81]。

在内存上下文中，还可以在malloc()和free()的开销相对较高的平台上提供性能优势。 特别是，内存上下文可以使用语义知识（通过上下文类型）来分配和释放内存，并相应地调用malloc()和free()以最小化操作系统开销。 数据库系统的某些组件（例如解析器和优化器）通过上下文删除一次性分配大量的小对象，然后释放它们。 在大多数平台上，释放许多小对象的调用是相当昂贵的。 内存分配器可以调用malloc()来分配大区域的内存，并将结果内存分配给调用者。 相对较少的内存释放意味着不需要malloc()和free()使用的压缩逻辑。 当上下文被删除时，只需要几个free()调用来移除大区域。

感兴趣的读者可能想浏览开源的PostgreSQL代码。 这利用了一个相当复杂的内存分配器。

7.2.1 关于查询操作的内存分配的说明

供应商在空间密集型操作（如哈希连接和排序）的内存分配方案上在哲学上存在差异。 一些系统（例如，DB2 for zSeries）允许DBA控制这些操作将使用的RAM数量，并保证每个查询在执行时获得该RAM数量。 准入控制策略确保了这一保证。 在这些系统中，操作员通过内存分配器从堆中分配内存。 这些系统提供了良好的性能稳定性，但迫使DBA（静态地）决定如何在各个子系统（如缓冲池和查询操作）之间平衡物理内存。

其他系统（例如，MS SQL Server）将内存分配任务从DBA手中拿走，并自动管理这些分配。 这些系统试图智能地分配内存，包括在缓冲池中进行页面缓存和查询操作内存使用。用于所有这些任务的内存池本身就是缓冲池。 因此，这些系统中的查询操作员通过DBMS实现的内存分配器从缓冲池中获取内存，并且仅在大于缓冲池页面的连续请求时使用操作系统的分配器。

这种区别呼应了我们在第6.3.1节中对查询准备的讨论。前一类系统假设数据库管理员（DBA）参与了复杂的调优，并且系统的工作负载可以通过精心选择的系统内存“旋钮”进行调整。在这些条件下，这样的系统应该始终表现出可预测的良好性能。 后一类系统则假设DBA要么不会，要么不能正确设置这些旋钮，并试图用软件逻辑来替代DBA的调优。 它们还保留了自适应更改相对分配的权利。 这为应对不断变化的工作负载提供了更好的性能可能性。 正如第6.3.1节中所讨论的，这种区别说明了这些供应商对其产品使用方式的期望，以及他们客户的管理专业知识（和财务资源）。

7.3磁盘管理子系统

数据库管理系统教材往往将磁盘视为同质的对象。 实际上，磁盘驱动器是复杂且异构的硬件设备，其容量和带宽差异很大。 因此，每个数据库管理系统都有一个磁盘管理子系统来处理这些问题，并管理表和其他存储单元在原始设备、逻辑卷或文件之间的分配。

这个子系统的一个责任是将表映射到设备和/或文件。 表到文件的一对一映射听起来很自然，但在早期文件系统中引发了重大问题。 首先，操作系统文件传统上不能大于一个磁盘，而数据库表可能需要跨多个磁盘。 其次，分配太多的操作系统文件是

被认为是不好的形式，因为操作系统通常只允许打开几个文件描述符，并且许多用于目录管理和备份的操作系统实用程序无法扩展到非常大量的文件。 最后，许多早期文件系统将文件大小限制为2 GB。这显然是一个无法接受的小表限制。 许多数据库管理系统供应商通过使用原始IO绕过操作系统文件系统，而其他供应商选择绕过这些限制。 因此，所有领先的商业数据库管理系统可以将表分布在多个文件中，或者将多个表存储在单个数据库文件中。 随着时间的推移，大多数操作系统文件系统已经超越了这些限制。 但是，传统影响仍然存在，现代数据库管理系统通常仍将操作系统文件视为抽象存储单元，可以任意映射到数据库表。

处理设备特定细节以维护时间和空间控制的代码更加复杂，如第4节所述。 今天存在着一个庞大而充满活力的行业，它基于复杂的存储设备，这些设备“假装”是磁盘驱动器，但实际上它们是大型硬件/软件系统，其API是像SCSI这样的传统磁盘驱动器接口。这些系统包括RAID系统和存储区域网络（SAN）设备，往往具有非常大的容量和复杂的性能特征。 管理员喜欢这些系统，因为它们易于安装，并且通常提供易于管理的位级可靠性和快速故障转移。 这些功能为客户提供了显著的安全感，超出了DBMS恢复子系统的承诺。 例如，大型DBMS安装通常使用SAN。

不幸的是，这些系统使DBMS的实现变得复杂。
以RAID系统为例，在故障发生后，它们的性能与所有磁盘正常工作时的性能非常不同。 这可能会使DBMS的I/O成本模型变得复杂。一些磁盘可以在启用写缓存模式下运行，但这可能会在硬件故障时导致数据损坏。 先进的SAN实现了大容量的电池备份缓存，有些接近一太字节，但这些系统带来了超过一百万行微代码和相当复杂的问题。 复杂性带来了新的故障模式，这些问题可能非常难以检测和正确诊断。

RAID系统也会让数据库设计人员感到沮丧，因为它们在数据库任务上表现不佳。 RAID最初是为面向字节流的存储（类似UNIX文件）而设计的，而不是为数据库系统所使用的面向页面的存储而设计的。因此，与针对分区和复制数据在多个物理设备上的数据库专用解决方案相比，RAID设备往往表现不佳。 例如，Gamma [43]的链式解聚方案与RAID的发明大致同时，对于DBMS环境来说性能更好。

此外，大多数数据库提供了DBA命令来控制数据在多个设备上的分区，但RAID设备通过隐藏多个设备在单个接口后，破坏了这些命令的作用。

许多用户配置他们的RAID设备以最小化空间开销（“RAID级别5”），而数据库通过更简单的方案如磁盘镜像（“RAID级别1”）会表现得更好。 RAID级别5的一个特别不好的特性是写入性能较差。 这可能会给用户带来令人惊讶的瓶颈，DBMS供应商通常需要解释或提供解决这些瓶颈的方法。 无论好坏，RAID设备的使用（和误用）是商业DBMS必须考虑的事实。 因此，大多数供应商都会花费大量精力来调优他们的DBMS以在主流RAID设备上表现良好。

在过去的十年中，大多数客户部署将数据库存储分配给文件，而不是直接分配给逻辑卷或原始设备。
但大多数数据库管理系统仍然支持原始设备访问，并且在运行高规模事务处理基准测试时经常使用这种存储映射。 尽管上述一些缺点，大多数企业数据库管理系统的存储今天是基于存储区域网络（SAN）的。

7.4 复制服务

经常希望通过定期更新在网络上复制数据库。 这经常用于提供额外的可靠性：复制的数据库作为一个略微过时的“温暖备用”系统，以防主系统崩溃。 将温暖备用系统保持在物理上不同的位置是有优势的，以确保持续运行。

在火灾或其他灾难之后。 复制也经常用于为大型、地理分布的企业提供实用的分布式数据库功能。 大多数这样的企业将他们的数据库分区为大的地理区域（例如国家或大陆），并在数据的主要副本上本地运行所有更新。查询也在本地执行，但可以在来自本地操作的新鲜数据和从远程地区复制的稍旧数据的混合上运行。

忽略硬件技术（例如，EMC SRDF），使用了三种典型的复制方案，但只有第三种方案提供了高端环境所需的性能和可扩展性。 当然，这是最难实现的。

1.物理复制：最简单的方案是在每个复制周期物理复制整个数据库。 由于数据传输的带宽和在远程站点重新安装数据的成本，这种方案无法扩展到大型数据库。 此外，保证数据库的事务一致性快照是棘手的。 因此，物理复制仅在低端作为客户端解决方案使用。大多数供应商不通过任何软件支持鼓励使用此方案。

2.基于触发器的复制：在此方案中，触发器被放置在数据库表上，以便在对表进行任何插入、删除或更新时，在特殊的复制表中安装一个“差异”记录。 这个复制表被传输到远程站点，并在那里“重放”修改。

这个方案解决了上述物理复制的问题，但对于某些工作负载来说，带来了无法接受的性能惩罚。

3.基于日志的复制: 基于日志的复制是可行时的复制解决方案。 在基于日志的复制中，一个日志嗅探器进程拦截日志写入并将其传递给远程系统。 基于日志的复制使用两种广泛的技术: (1) 读取日志并构建要在目标系统上重放的SQL语句，或者

(2) 读取日志记录并将其发送到目标系统，目标系统处于持续恢复模式，按照日志记录的到达顺序重放日志记录。 这两种机制都有价值，因此Microsoft SQL Server，DB2和Oracle都实现了这两种机制。 SQL Server将第一种称为日志传送，将第二种称为数据库镜像。这个方案克服了以前的所有问题: 它的开销很低，在运行系统上几乎没有或者没有可见的性能开销; 它提供增量更新，因此可以与数据库大小和更新速率优雅地扩展; 它重用了DBMS的内置机制，没有显著的额外逻辑; 最后，它通过日志的内置逻辑自然地提供事务一致的副本。

大多数主要供应商为其自己的系统提供基于日志的复制。 提供跨供应商的基于日志的复制要困难得多，因为在远程端驱动供应商的重放逻辑需要理解该供应商的日志格式。

7.5 管理、监控和实用程序

每个数据库管理系统都提供一套用于管理其系统的实用程序。
这些实用程序很少进行基准测试，但通常决定了系统的可管理性。 一个技术上具有挑战性且特别重要的功能是使这些实用程序在线运行，即在用户查询和事务正在进行时运行。 这在由于电子商务的全球覆盖而变得越来越常见的24小时运营中非常重要。 传统的“重组窗口”通常在凌晨时段不再可用。 因此，大多数供应商近年来在提供在线实用程序方面投入了大量精力。 我们在这里介绍一些实用程序的概述：

- 优化器统计信息收集：每个主要的数据库管理系统都有一些方法来扫描表格并构建优化器统计信息。 一些统计信息，比如直方图，需要在一次遍历中进行非常复杂的构建，而不会占用过多内存。

例如，可以参考Flajolet和Martin在计算列中不同值数量方面的工作[17]。

- 物理重组和索引构建：随着时间的推移，访问方法可能由于插入和删除的模式而变得低效，从而导致未使用的空间。 此外，用户有时可能会要求在后台重新组织表格，例如根据不同的列重新聚集（排序）它们，或者在多个磁盘上重新分区它们。 在线文件和索引的重组可能会很棘手，因为必须避免长时间持有锁定以保持物理一致性。 从这个意义上讲，它与用于索引的日志记录和锁定协议有一些类似之处，如第5.4节所述。这已经成为几篇研究论文[95]和专利的主题。

- 备份/导出：所有数据库管理系统都支持将数据库物理转储到备份存储中。 再次，由于这是一个长时间运行的过程，它不能简单地设置锁定。 相反，大多数系统执行某种“模糊”转储，并通过日志记录逻辑来确保事务一致性。

  类似的方案可以用于将数据库导出到交换格式。

- 批量加载：在许多场景中，需要快速将大量数据导入数据库。 供应商提供了针对高速数据导入进行优化的批量加载实用程序，而不是逐行插入。 通常，这些实用程序由存储管理器中的自定义代码支持。

  例如，针对B+-树的特殊批量加载代码比重复调用树插入代码要快得多。

- 监控、调优和资源管理器：即使在托管环境中，查询消耗的资源可能超过预期也是常见的。 因此，大多数数据库管理系统提供了帮助管理员识别和预防这类问题的工具。 通常，通过“虚拟表”提供基于SQL的接口来访问数据库管理系统性能计数器，这些计数器可以按查询或按锁定、内存、临时存储等资源来显示系统状态。

年龄等等。 在一些系统中，还可以查询此类数据的历史日志。 许多系统允许在查询超过某些性能限制时注册警报，包括运行时间、内存或锁获取；在某些情况下，触发警报可能导致查询被中止。 最后，像IBM的预测资源管理器这样的工具试图完全阻止运行资源密集型查询。

8

结论

从本文可以清楚地看出，现代商业数据库系统既基于学术研究，也基于为高端客户开发工业级产品的经验。 从头开始编写和维护一个高性能、完全功能的关系型DBMS是一项巨大的时间和精力投资。 然而，关系型DBMS的许多经验教训也适用于新的领域。 Web服务、网络附加存储、文本和电子邮件存储库、通知服务和网络监视器都可以从DBMS的研究和经验中受益。 数据密集型服务是当今计算的核心，数据库系统设计的知识是一种广泛适用的技能，无论是在主要数据库公司的大厅内还是外部。 这些新的方向也提出了一些数据库管理方面的研究问题，并指引数据库社区与计算机领域的其他领域之间的新的互动。

致谢

作者们要感谢Rob von Behren、Eric Brewer、Paul Brown、Amol Deshpande、Cesar Galindo-Legaria、Jim Gray、Wei Hong、Matt Huras、Lubor Kollar、Ganapathy Krishnamoorthy、Bruce Lindsay、Guy Lohman、S. Muralidhar、Pat Selinger、Mehul Shah和Matt Welsh对本文初稿的背景信息和评论表示感谢。

参考文献

[1] A. Adya, B. Liskov, and P. O'Neil, "广义隔离级别定义"，于第16届国际数据工程会议（*ICDE*）, 加利福尼亚州圣地亚哥，2000年2月。

[2] R. Agrawal, M. J. Carey, and M. Livny, "并发控制性能建模：选择和影响"，ACM数据库系统交易（*TODS*）, 第12卷，第609-654页，1987年。

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. Frank King III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: 关系数据库管理方法," *ACM Transactions on Database Systems (TODS)*, vol. 1, pp. 97–137, 1976.

[4] R. Bayer and M. Schkolnick, "B树上的操作并发性," *Acta Informatica*, vol. 9, pp. 1–21, 1977.

[5] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis, "用于数据库查询优化的遗传算法," in*Proceedings of the 4th International Confer-ence on Genetic Algorithms*, pp. 400–407, San Diego, CA, July 1991.

[6] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil,"对ANSI SQL隔离级别的批评"，在A*CM* SI*GMOD*国际数据管理会议上，第1-10页，加利福尼亚州圣何塞，1995年5月。

[7] P. A. Bernstein和N. Goodman，"分布式数据库系统中的并发控制"，*ACM Com puting Surveys*，卷13，1981年。

[8] W. Bridge，A. Joshi，M. Keihl，T. Lahiri，J. Loaiza和N. MacNaughton，"Oracle通用服务器缓冲区"，在第23届国际大型数据库会议（*VLDB*）上，第590-594页，希腊雅典，1997年8月。

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: 一个分布式存储系统用于结构化数据," in操作系统设计和实现研讨会*(OSDI)*, 2006.

[10] S. Chaudhuri, "关系系统中查询优化的概述," in *ACM*数据库系统原理研讨会 *(PODS)*, 1998.

[11] S. Chaudhuri and U. Dayal, "数据仓库和*O*LAP技术概述," AC*M* S*IGMO*D记录, 1997年3月.

[12] S. Chaudhuri and V. R. Narasayya, "Autoadmin 'what-if'索引分析工具," inAC*M* S*IGMOD*国际数据管理会议, pp. 367–378, 西雅图, 华盛顿州, 1998年6月.

[13] S. Chaudhuri和K. Shim，"具有用户定义谓词的查询优化"，*ACM T*ransactions on *D*ata*b*ase S*y*stems *(T*O*D*S*)*，卷24，页177-228，1999年。

[14] M.-S. Chen，J. Hun和P. S. Yu，"数据挖掘：从数据库角度的概述"，*IEEE T*ransactions on *K*now*l*edge and Data *E*ngineering，卷8，1996年。

[15] H.-T. Chou和D. J. DeWitt，"关系数据库系统的缓冲区管理策略评估"，在第11届国际大数据会议*(VL*D*B)*论文集中，页127-141，瑞典斯德哥尔摩，1985年8月。

[16] A. Desphande，M. Garofalakis和R. Rastogi，"独立性是好的：基于依赖的高维数据直方图概要"，在第1*8*届国际数据工程会议论文集中，加利福尼亚圣何塞，2001年2月。

[17] P. Flajolet和G. Nigel Martin，"用于数据库应用的概率计数算法"，计算机系统科学杂志，第31卷，第182-209页，1985年。

[18] C. A. Galindo-Legaria，A. Pellenkoft和M. L. Kersten，"快速、随机的连接顺序选择-为什么使用转换？"，*VLDB*，第85-95页，1994年。

[19] S. Ganguly，W. Hasan和R. Krishnamurthy，"并行执行的查询优化"，在*ACM* S*IGMOD*国际数据管理会议论文集中，第9-18页，加利福尼亚州圣地亚哥，1992年6月。

[20] M. Garofalakis和P. B. Gibbons，"近似查询处理：驯服千兆字节，教程"，在非常大数据的国际会议上，2001年。 www.vldb.org/conf/2001/tut4.pdf。

[21] M. N. Garofalakis 和 Y. E. Ioannidis, "使用时间和空间共享资源的并行查询调度和优化," 在第2*3*届 *V*ery *L*arge *D*ata *B*ases *(VLDB)* 国际会议论文集中, pp. 296–*3*05, 希腊雅典, 1997年8月.

[22] R. Goldman 和 J. Widom, "Wsq/dsq: 一种结合数据库和网络的实用查询方法," 在 A*CM-SI*G*MOD* 国际数据管理会议论文集中, 2000年.

[23] G. Graefe, "在火山查询处理系统中封装并行性," 在 *ACM-SI*G*MOD* 国际数据管理会议论文集中, pp. 102–111, 大西洋城, 1990年5月.

[24] G. Graefe, “大型数据库的查询评估技术,” *Comp*uting *S*urveys, vol. 25, *pp.* 73–17 0, 1993年.
[25] G. Graefe，“查询优化的级联框架”，*IEEE*数据工程通报，第18卷，第19-29页，1995年。
[26] C. Graham，“市场份额：按操作系统划分的关系数据库管理系统，全球范围内，2005年”，Gartner报告编号：G00141017，2006年5月。
[27] J. Gray，“来自文件系统用户的问候”，在*FAST'05*文件和存储技术会议论文集中，（旧金山），2*005*年12月。
[28] J. Gray和B. Fitzgerald，“F*LASH*磁盘在服务器应用中的机会”。htt*p*://re*s*earch .mi*c*r*osof*t.*c*om/∼Gray/*pa*pe*r*s/Fla*s*h*Disk*P*u*bli*c*.d*oc*。[29] J. Gray，R. *A* . *L*orie，G. R. P*utzolu*和I. *L*. Traiger，“共享数据库中的锁的粒度和一致性程度”，在IFIP数据库管理系统建模工作会议上，第365-394页，1976年。

[30] J. Gray 和 A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
[31] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, 和 D. Culler, “Scalable, distributed data structures for internet service construction,” in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.
[32] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 47–57, Boston, June 1984.
[33] L. Haas, D. Kossmann, E. L. Wimmers, 和 J. Yang, “Optimizing queries across diverse data sources,” in *International Conference on Very Large Databases (VLDB)*, 1997.
[34] T. Haerder 和 A. Reuter, “面向事务的数据库恢复原则,” *ACM Computing Surveys*, vol. 15, pp. 287–317, 1983.
[35] S. Harizopoulos 和 N. Ailamaki, “StagedDB: 为现代硬件设计数据库服务器,” *IEEE Data Engineering Bulletin*, vol. 28, pp. 11–16, 六月2005.

[36] S. Harizopoulos, V. Liang, D. Abadi, 和 S. Madden, “在面向读取优化的数据库中的性能权衡,” in*Proceedings of the 32nd Very Large DatabasesConference (VLDB)*, 2006.
[37] J. M. Hellerstein, “对于具有昂贵方法的查询的优化技术,” *ACM Transactions on Database Systems (TODS)*, vol. 23, pp. 113–157,1998.

[38] J. M. Hellerstein, P. J. Haas, and H. J. Wang, “在线聚合”，在*ACM-SIGMOD*国际数据管理会议，1997年。

[39] J. M. Hellerstein, J. Naughton, and A. Pfeffer, “数据库系统的广义搜索树”，在*V*ery *L*arge *D*ata *B*ases *C*onference*(VLDB)*，1995年。

[40] J. M. Hellerstein and A. Pfeffer, “俄罗斯套娃树，一种用于集合的索引结构”，威斯康星大学技术报告TR1252，1994年。
[41] C. Hoare, “监视器：一种操作系统结构概念”，ACM通信杂志*(*CACM*)*，第17卷，第549-557页，1974年。

[42] W. Hong 和 M. Stonebraker, “在 xprs 中并行查询执行计划的优化,” 在第一届国际并行和分布式信息系统会议 *(PDI*S*)* 上, pp. 218–225, 迈阿密海滩, *F*L, 1991年12月.

[43] H.-I. Hsiao 和 D. J. DeWitt, “链式分布式: 用于多处理器数据库机器的新可用性策略,” 在第六届国际数据工程会议 *(*ICD*E)* 上, pp. 456–465, 洛杉矶, CA, 1990年11月.

[44] Y. E. Ioannidis 和 Y. Cha Kang, “用于优化大型连接查询的随机算法,” 在*AC*M*-S*I*GMOD*国际数据管理会议 上, pp. 312–321, 大西洋城, 1990年5月.

[45] Y. E. Ioannidis和S. Christodoulakis，“关于连接结果大小误差的传播”，在*AC*M SI*GMOD*国际数据管理会议的论文集中，第268-277页，1991年5月，科罗拉多州丹佛市。

[46] M. Kornacker，C. Mohan和J. M. Hellerstein，“广义搜索树中的并发和恢复”，在*A*CM *SIG*M*OD*国际数据管理会议的论文集中，第62-72页，1997年5月，亚利桑那州图森市。

[47] H. T. Kung和J. T. Robinson，“关于并发控制的乐观方法”，*ACM*数据库系统交易（TO*DS*），第6卷，第213-226页，1981年。

[48] J. R. Larus和M. Parkes，“使用队列调度提升服务器性能”，在U*SENIX*年会上，2002年。

[49] H. C. Lauer和R. M. Needham，“操作系统结构的二元性”，*A*CM *SIG*O*PS*操作系统评论，卷13，第3-19页，1979年4月。

[50] P. L. Lehman和S. Bing Yao，“B树上并发操作的高效锁定”，*ACM*数据库系统交易（*TOD*S），卷6，第650-670页，1981年12月。

[51] A. Y. Levy，“使用视图回答查询”，*V*L*DB Jo*urn*al*，卷10，第270-294页，2001年。

[52] A. Y. Levy，I. Singh Mumick和Y. Sagiv，“通过谓词移动进行查询优化”，在第2*0*届国际大型数据库会议论文集中，第96-1*07*页，圣地亚哥，1994年9月。

[53] W. Litwin，“线性哈希：文件和表寻址的新工具”，在第六届 *V*er*y* Large *D*ata *B*ases *(V*L*DB)*国际会议上，第212-223页，加拿大魁北克蒙特利尔，1980年10月。

[54] 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*AC*M *SIG*M*OD*国际数据管理会议*P*roceed*i*ngs上，第18-27页，伊利诺伊芝加哥，1988年6月。

[55] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, 和 J. F. Naughton，“面向电子商务的中间层数据库缓存”，在ACM SIGMOD国际数据管理会议P*rocee*d*ing*s上，2002年。

[56] S. R. Madden 和 M. J. Franklin，“流式传感器数据查询的架构”，在第*12*届*IEEE*国际数据工程会议*(ICDE)*上，圣何塞，*200*2年*2*月。

[57] V. Markl, G. Lohman, 和 V. Raman, “Leo: 一个自主查询优化器 for db2,” *IBM* 系统杂志 , vol. 42, pp. 98–106, 2003.

[58] C. Mohan, “Aries/kvl: 一种用于并发控制的键值锁定方法of multiaction transacti ons operating on b-tree indexes,” in第*16*届国际*V*ery *L*arge *D*ata *B*ases *(VLDB)*, pp. 392–405, 布里斯班, 昆士兰州, 澳大利亚, *1*990年8月.

[59] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, 和 P. M. Schwarz, “Aries: 一种支持细粒度锁定和 partial rollbacks using write-ahead logging的事务恢复方法,” *ACM* 数据库系统 *(TODS)*, vol. 17, pp. 94–162, 1992年.

[60] C. Mohan和F. Levine，“Aries/im: 一种使用预写式日志的高效且高并发的索引管理方法”，在ACM *SIG*-M*OD*国际数据管理会议（M. *S*tonebraker, 编辑），pp. 371–380, 圣地亚哥, 加利福尼亚, 1992年6月。

[61] C. Mohan, B. G. Lindsay和R. Obermarck，“r*分布式数据库管理系统中的事务管理”，*ACM Transactions on DatabaseSystems (TODS)*, vol. 11, pp. 378–396, 1986年。

[62] E. Nightingale, K. Veerarghavan, P. M. Chen和J. Flinn，“重新思考同步”，在操作系统设计和实现研讨会（*OSDI*），2006年11月。

[63] OLAP市场报告。在线手稿。 http://www.olapreport.com/market.htm.

[64] E. J. O’Neil, P. E. O’Neil和G. Weikum，“用于数据库磁盘缓冲的lru-k页面替换算法”，在*ACM SIGMOD*国际数据管理会议的论文集中，第297-306页，华盛顿，1993年5月。

[65] P. E. O’Neil和D. Quass，“使用变体索引提高查询性能”，在*ACM-SIGMOD*国际数据管理会议的论文集中，第38-49页，图森，1997年5月。

[66] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston和M. Huras，“多维聚类：db2中的新数据布局方案”，在ACM S*IGMOD*国际数据管理会议（加利福尼亚州圣地亚哥，2003年6月9-12日）S*IGMOD* '03的论文集中，第637-641页，纽约，纽约：ACM出版社，2003年。[67] *D.* Patterson，“延迟超过带宽”，CACM，第47卷，第71-75页，2004年10月。

[68] H. Pirahesh, J. M. Hellerstein, 和 W. Hasan, “可扩展/基于规则的查询重写优化在starburst中,” 在*ACM-SIGMOD*国际数据管理会议的论文集中, pp. 39–48, 圣地亚哥, 1992年6月.

[69] V. Poosala 和 Y. E. Ioannidis, “在不依赖属性值独立性假设的情况下进行选择性估计,” 在第*23*届国际大数据会议*(VLDB)*的论文集中, pp. 486–495, 雅典, 希腊, 1997年8月.

[70] M. P¨oss, B. Smith, L. Koll´ar, 和 P.-˚A. Larson, “Tpc-ds, 将决策支持基准提升到下一个级别,” 在*SIGMOD 2002*, pp. 582–587.

[71] V. Prabakharan, A. C. Arpaci-Dusseau, 和 R. Arpaci-Dusseau, “分析和演化的日志文件系统,” 在*USENIX*年度技术会议, 2005年4月.

[72] R. Ramakrishnan 和 J. Gehrke, “数据库管理系统,” McGraw-Hill, 波士顿, 马萨诸塞州, 第三版, 2003年.

[73] V. Raman 和 G. Swart, “如何将表格压缩到极限: 压缩关系的熵压缩和查询,” 在国际非常大的数据会议 *(*V*LDB)*, 2006年.

[74] D. P. Reed,在分散式计算机系统中的命名和同步. 博士论文, MIT, 电气工程系, 1978年.

[75] A. Reiter，“数据管理系统的缓冲管理策略研究”，数学研究中心技术摘要报告1619，威斯康星大学麦迪逊分校，1976年。

[76] D. J. Rosenkrantz, R. E. Stearns和P. M. Lewis，“分布式数据库系统的系统级并发控制”，*A*C*M*数据库系统交易（*TO*DS），第3卷，第178-198页，1978年6月。

[77] S. Sarawagi, S. Thomas和R. Agrawal，“将挖掘与关系数据库系统集成：选择和影响”，在A*CM*-SI*GMOD*国际数据管理会议论文集中，1998年。

[78] R. Sears和E. Brewer，“Statis：灵活的事务存储”，在操作系统设计和实现研讨会（*OSDI*）论文集中，2006年。

[79] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, “在关系数据库管理系统中的访问路径选择”，在ACM-S*I*GM*O*D国际数据管理会议论文集中，第22-34页，波士顿，1979年。

[80] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, “复杂查询的解耦”，在第*12*届*IEEE*国际数据工程会议*(ICDE)*论文集中，新奥尔良，*1*996年*2*月。

[81] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein, “用于数据密集型系统的Java支持：构建电报数据流系统的经验”，A*C*M S*IG*M*OD R*ecord，卷30，第103-114页，2001年。

[82] L. D. Shapiro，“利用上下界在自顶向下查询优化中的应用”，国际数据库工程与应用研讨会（*I*DE*A*S）, 2001年。

[83] A. Silberschatz, H. F. Korth, 和 S. Sudarshan,数据库系统概念. 麦格劳-希尔, 波士顿, 马萨诸塞州, 第四版, 2001年。

[84] M. Steinbrunn, G. Moerkotte, 和 A. Kemper, “启发式和随机化优化解决连接顺序问题”， *VLDB Journal*, 卷. 6, 页. 191–208,1997年。

[85] M. Stonebraker, “对数据库系统的回顾”，*ACM Transactions on Database Systems (TODS)*, 卷. 5, 页. 225–240, 1980年。

[86] M. Stonebraker，“操作系统对数据库管理的支持，”*A*C*M*通信（*CA*C*M*），第24卷，第412-418页，1981年。

[87] M. Stonebraker，“共享无关性的理由，”*IEEE*数据库工程公报，第9卷，第4-9页，1986年。

[88] M. Stonebraker，“关系数据库系统中新类型的包含，” *ICDE*，第262-269页，1986年。

[89] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran,

和S. Zdonik，“C-store：一种列式*DB*MS，”在非常大型数据库（*VLDB*）会议论文集，2005年。

[90] M. Stonebraker和U. Cetintemel，“一刀切：一个时代已经过去的想法”，在国际数据工程会议（*ICDE*），2005年。

[91] 事务处理性能委员会2006年。 TPC基准C标准规范修订5.7，http://www.tpc.org/tpcc/spec/tpcc current。 pdf，四月。

[92] T. Urhan，M. J. Franklin和L. Amsaleg，“基于成本的查询混淆用于初始延迟”，ACM-*SIG*M*OD*国际数据管理会议，1998年。

[93] R. von Behren，J. Condit，F. Zhou，G. C. Necula和E. Brewer，“Capriccio：互联网服务的可扩展线程”，在第十九届操作系统原理研讨会（S*OSP-19*），纽约州湖乔治，2003年*10*月。

[94] M. Welsh, D. Culler, and E. Brewer, “Seda: 一个适用于良好条件、可扩展的互联网服务的架构,” in*Proceedings of the 18th Symposium onOperating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

[95] C. Zou and B. Salzberg, “稀疏B+树的在线重组,” pp. 115–124, 1996.