《Architecture of a Database System》

（中文版）

（版本号：2013 年 12 月 4 日）

Joseph M. Hellerstein, Michael Stonebraker and James Hamilton

翻译：林子雨

厦门大学数据库实验室

http://dblab.xmu.edu.cn

论文中文版网址：http://dblab.xmu.edu.cn/node/459

厦门大学计算机科学系教师 林子雨 翻译作品

http://www.cs.xmu.edu.cn/linziyu

2013 年 9 月

前言

本文翻译自经典英文论文《Architecture of a Database System》，原文作者是Joseph M. Hellerstein, Michael Stonebraker 和 James Hamilton。该论文可以作为中国各大高校数据库实验室研究生的入门读物，帮助学生快速了解数据库的内部运行机制。

本文一共包括 8 章，分别是：第 1 章概述，第 2 章进程模型，第 3 章并行体系结构：进程和内存协调，第 4 章关系查询处理器，第 5 章存储管理，第 6 章事务：并发控制和恢复，第 7 章共享组件，第 8 章结束语。

本文翻译由厦门大学数据库实验室林子雨老师团队合力完成，其中，林子雨老师负责统稿校对，刘颖杰同学负责翻译第 1 章、第 2 章和第 6 章，罗道文同学负责翻译第 3 章和第 4 章，谢荣东同学负责翻译第 5 章，蔡珉星同学负责翻译第 7 章和第 8 章，并负责对林子雨老师校对结果进行二次校对。

如果对本文翻译内容有任何疑问，欢迎联系林子雨老师。

林子雨的E-mail是：ziyulin@xmu.edu.cn。

林子雨的个人主页是：http://www.cs.xmu.edu.cn/linziyu。

厦门大学数据库实验室网站是：http://dblab.xmu.edu.cn。

本文中文版的网址是：http://dblab.xmu.edu.cn/node/459。

林子雨于厦门大学海韵园

2013 年 9 月

摘 要

数据库管理系统（DBMS）广泛存在于现代计算机系统中，并且是其重要的组成部分。它是学术界以及工业界数十年研究和发展的成果。在计算机发展史上，数据库属于最早开发的多用户服务系统之一，因此，它的研究也催生了许多为保证系统可扩展性以及稳定性的系统开发技术，这些技术如今被应用于许多其他的领域。虽然许多数据库的相关算法和概念广泛见于教科书中，但关于如何让一个数据库工作的系统设计问题却鲜有资料介绍。本文从体系架构角度探讨数据库设计的一些准则，包括处理模型、并行架构、存储系统设计、事务处理系统、查询处理及优化结构以及具有代表性的共享组件和应用。当业界有多种设计方式可供选择时，我们以当前成功的商业开源软件作为参考标准。

目 录

第 1 章 概述

数据库管理系统（DBMS）是一种复杂的、关键任务软件系统。今天的数据库管理系统包含了学术界和工业界数十年的研究以及大量的企业软件开发成果。数据库管理系统属于最早期广泛应用的在线服务系统之一，因此，具备前沿的设计方法，这些设计方法涵盖数据管理、计算机应用、操作系统以及网络服务等方面。早期的数据库管理系统是计算机科学领域最具影响力的软件系统之一，而且，那些因为数据库研究而产生的理念和实现技术也被广泛地借鉴和创新。

由于诸多原因，数据库管理系统架构的相关介绍并没有像它应该的那样被人们广泛地熟知。首先，应用数据库群体较小。由于市场只能支撑几个高水平的竞争者，因此，只有一小部分成功的数据库产品存在。从事数据库设计和应用的人们彼此联系紧密，他们往往来自于同一所学校，研究同样的知名项目，然后合作开发几个相同的产品。另一方面，数据库管理系统的教学领域往往忽视对体系架构问题的讲解。数据库教材一直关注那些易于教学、研究和考试的算法和理论知识点，没有从应用角度对数据库架构有一个全局的讲解。总而言之，关于如何构建一个数据库方面的知识，并不是保密的，可是，它并没有被系统地写下来并供人们讨论交流。

本文中，我们希望通过几个方向的讨论，介绍清楚现代数据库系统架构的主要方面。这些内容部分见于教材中，我们会给出合适的注释。另外有些内容埋藏于用户手册中以及一些数据库相关团体的口头交流中。在适当的情况下，我们使用商业开源软件作为复杂多样的数据库架构的实例。当然，受篇幅所限，在这至少有十年历史的数百万行代码中，它们的特性以及一些好的创新点就不能一一列举了。我们的重点在于整个系统的架构设计，并着重讲解那些没有被教科书着重谈到的、但是却使那些广为人知的算法发挥作用的系统环境。我们希望读者已经熟悉主流的数据库教材，并且对现代操作系统如 UNIX、Linux 以及 Windows 有基本的操作能力。在下一节整体介绍完一个数据库管理系统的架构之后，我们在第 1.2 节为每一部分组件提供一些参考资料作为背景阅读。

1.1 关系数据库：查询的命脉

迄今为止，发展成熟并且得到最广泛应用的数据库类型是关系数据库（RDBMS）。它们广泛应用于许多应用系统中，比如电子商务、医疗文件、人力资源、工资系统、客户联系管理以及供应链管理等等。网络社区的出现也使得它得到更广泛的应用。关系数据库几乎被用作所有的网络交易及在线内容管理系统的数据存储方式，如博客、维基百科、社交网络等等。关系型数据库不仅是重要的软件设施，它也是帮助我们理解未来可能发生的数据库变革的很好的切入点。因此，本文将始终以关系型数据库为例。

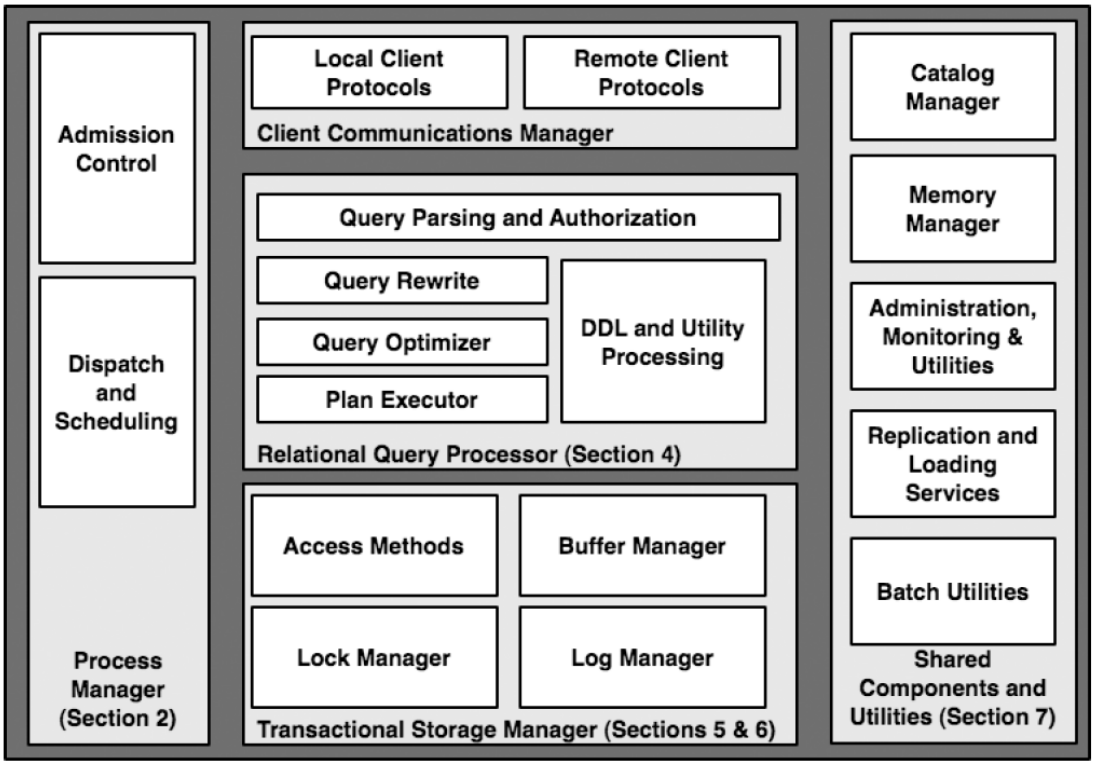

图 1-1 一个 DBMS 的主要组件

如图 1-1 所示，关系数据库主要由五部分组成。为了分别介绍这五个部分的内容以及它们之间如何相互合作，我们先来看看一个查询语句在数据库系统是如何被处理的。这同时也作为本文内容的概览。

让我们来考虑这样一个简单却很典型的数据库在机场的应用实例：查询某次航班的所有旅客名单。这个操作所引发的的查询请求大致按如下方式被处理：

1、机场登机口的 PC 机（客户端）调用 API 与 DBMS 的客户端通信管理器（Client Communications Manager）建立网络连接。在一些情况下，客户端直接通过 ODBC 或 JDBC 连接协议与数据库服务器建立这种连接。这种处理方式被称为“两层”或者“客户端-服务

器”。还有一些情形中，客户端与“中间层服务器”建立连接，如 web 服务器、事务处理系统等，它们通过协议代理原本直接建立在客户端与数据库服务器之间的连接，这被称为“三层”模式。在一些网络应用中，还会再多一个应用服务器架设于网络服务器和 DBMS 之间，这被称为“四层”模式。为了适用于如此多的环境，一个 DBMS 需要兼容许多不同客户端以及中间系统的多种连接协议。不过实际上，DBMS 中负责多种协议的管理器基本上是相同的：为调用者（客户端或者中间件）建立连接并记录其连接地址，对客户端的 sql 语句做出回应，并在适当的时候返回数据以及控制信息。在本文例子中，通信管理器还将为客户端建立安全证书，为新的连接细节以及客户端 sql 命令分配空间，并将客户端的请求传送到 DBMS 更底层进行处理。

2、在收到客户端的第一个请求之后，DBMS 必须为之分配一个计算线程。系统必须确保该线程的数据以及控制输出是通过通信管理器与客户端连接的。这些工作交由 DBMS 的进程管理器（Process Manager）来管理（图 1-1 左）。在这一部分中，DBMS 所做的主要工作是准入控制，即系统是否应该立即处理该查询，或是等待系统有足够资源时再处理该查询。我们将在第二章详细介绍进程管理器。

3、在分配控制进程之后，登机口的查询便可以被处理了。处理工作借助于关系查询处理器（Relational Query Processor，图 1-1 中间部分）中的代码来实现。这些模块检查用户是否有权进行该查询，然后将用户的 sql 查询语句编译为中间查询计划。在编译之后，结果查询计划被交给查询执行器。查询执行器包含一系列处理查询的操作（关系型算法实现）。典型的处理查询任务的操作包括：连接、选择、投影、聚集、排序等等，当然也包括从底层读取需要的数据。在我们的例子中，包括优化查询的操作在内，调用了一个操作集合来解决用户的查询问题。在第四章我们将讨论查询处理问题。

4、在登机口代理的查询计划的底层，由若干操作从数据库请求数据。这些操作通过调用(call)来从 DBMS 的存储管理器（transactional storage manager，图 1-1 底部）中收集数据；存储管理器负责所有的数据接口（读）和操作调用（建立、更新、删除）。存储系统包括用于管理磁盘数据的基本算法和数据结构，比如基本的表和索引。它还包括一个缓冲管理器，用来控制内存缓冲区和磁盘之间的数据传输。回到我们的例子中，在获取数据的过程中，登机口客户端的查询必须调用事务管理代码来保证“ACID”性质（将在第 5.1 节讨论）。在获取数据之前，需要通过锁管理器来确保并发情况下运行的正确性。如果登机口客户端的查询包含对数据库的更新操作，那么，它需要与日志系统进行交互，来确保更新操作的持久性以及撤销操作的完整性。在第 5 章，我们会讨论存储与缓冲管理的更多细节，第 6 章介绍

业务一致性结构。

5、在查询的这一时期，查询操作已经开始获取数据并准备好用它们来为客户端计算结果。这一步通过展开我们之前提到的所有操作的堆栈来完成。访问方法把控制权交给查询处理器，查询处理器将数据库的数据组织成结果元组；结果元组生成后被放入客户通信管理器的缓冲区中，然后该通信管理器将结果发送给调用者。对于较大的结果集合，客户端会发送更多的请求来获取更多的数据，这也导致了通信管理器、查询处理器和存储管理器的循环操作。在我们的例子中，在查询操作的最后，事务结束，连接关闭；事务管理器中的结果被清空，进程管理器释放无用的数据结构，通信管理器将连接状态清空。

我们通过这个查询的例子讨论了 RDBMS 的许多关键组件，但还有一些没有涉及到。图 1-1 右边一侧有许多共享组件和工具，它们对于一个功能完整的 DBMS 而言，同样是十分重要的。目录和存储管理器在传输数据时被作为工具来调用，在我们的例子中也是这样。在认证、分解以及查询优化过程中，查询处理器都会用到目录。同样，存储管理器也广泛应用于整个 DBMS 运行过程中动态分配和释放内存的场合。在图 1-1 中最右边列出的其余组件，独立运行于任何查询，它们使数据库保持稳定性和整体性。我们将在第 7 章讨论这些共享的组件和工具。

1.2 本文内容介绍

本文的大部分篇幅着重介绍支撑数据库核心功能实现的架构原理。我们没有过多涉及数据库算法，在文档中你可以很方便地找到它们。对于现代 DBMS 系统的衍生品，我们也很少讨论，它们虽然提供了一些不同于数据库核心部分的特征，但是，这对于数据库架构的改变并不大。然而，在本文的很多章节中，我们感兴趣的内容有些超出了文章应该涉及的范围，我们将尽可能地为这些额外的知识提供参考资料。

我们开篇谈到了数据库系统的整体架构。在所有服务器架构中第一个问题便是服务器整个的进程架构，我们会介绍各种切实可行的方案，首先是单处理机，然后是多种并行处理架构。关于核心服务器系统架构的讨论，是适合于许多其他系统的，但是，在很大程度上是在 DBMS 设计中最早得到应用的。随后，我们探讨 DBMS 的专有组件。我们从一个简单查询的角度开始，重点关注关系数据库查询处理器。在这之后，我们研究存储架构和事务存储管理设计。最终，我们介绍一些大部分 DBMS 都会有的共享组件和工具，这一点在教材中很少被提及。

第 2 章 进程模型

在设计多用户服务器时，必须要尽早决定并发用户请求的执行以及它们是如何被映射到操作系统的进程和线程的。这个决定将对之后的系统软件架构以及运行效率、稳定性、轻便性产生深远影响。在这一章，我们测试一些 DBMS 的运行数据，它们也可以作为其他高并发系统的参考模版。我们考虑一个简化的框架，假设系统支持线程并且仅考虑单片机的情况。然后我们进一步阐述现代 DBMS 在简化的框架下是如何实现它们的进程模型的。在第三章，我们将介绍集群技术以及多进程系统和多核系统。

我们接下来的讨论基于以下定义：

- 一个**操作系统进程**包含了一个正在执行的程序的活动单元以及专属的地址空间。为一个进程维护的状态，包括系统资源句柄和安全性上下文环境。程序执行的单元被操作系统内核管理，每一个进程有其自己的地址空间。
- 一个**操作系统线程**是操作系统程序执行单元，它没有私有的地址空间和上下文。在多线程系统中，每个线程都可以共享地访问所属进程的地址空间。线程的执行是由系统内核来管理的，通常被称为内核线程或者 k-线程。
- **轻量级线程包**是一个应用层次上的结构，它支持单系统进程中的多线程。不同于由操作系统内核调度的线程，轻量级线程由应用级线程调度程序来负责调度。它们之间的区别是，轻量级线程仅在用户空间内调度而没有内核调度程序的参与。轻量级线程仅运行在一个进程中，从操作系统调度程序的角度来看只有一个线程在运行。轻量级线程相对于系统线程而言，可以更快地切换，因为轻量级线程不必通过系统内核来切换调度。轻量级线程也有它的缺点，任何线程中断操作，比如 I/O 中断，都会打断进程中的其他线程。这使得一个线程中断等待系统资源时，其它线程就不能运行。轻量级线程包通过以下两点来避免该情况的发生：（1）只接受无中断的异步 I/O 操作；（2）不调用可以导致中断的系统操作。总的来说，轻量级线程相对于系统进程和系统线程，提供了一个更困难的编程模型。一些 DBMS 系统实现了它们自己的轻量级线程包，这些是轻量级线程包的特殊实现。当需要区别于其它线程时，我们将这些线程称为 DBMS 线程和简单线程。

- **DBMS 客户端**是应用程序实现与数据库系统通信 API 的软件组件，JDBC、ODBC 和 OLE/DB 是我们熟知的例子。除此之外，还有许多其他种类的 API 集合。有一些程序使用了嵌入式 SQL，这是一种混合了数据库访问表的编程方法。这种方法最初存在于 IBM COBOL 和 PL/I 中，后来 SQL/J 也为 Java 实现了嵌入式 SQL。嵌入式 SQL 会被预处理器处理，从而把嵌入式 SQL 转换为对数据访问 API 的直接调用。不管客户端程序使用什么语法，最终的结果就是一系列针对数据访问 API 的直接调用。这些调用由 DBMS 客户端组件整理然后通过通信协议与 DBMS 交互。这些协议通常拥有专利但是没有正式规范文档。在过去，开源组织 DRDA 曾试图将客户端-数据库通信协议规范化，但并没有被广泛使用。
- **DBMS 工作者**是指 DBMS 中为客户端工作的线程。它与客户端是一一对应的，即一个 DBMS 工作者处理来自一个 DBMS 客户端的所有 SQL 请求。DBMS 客户端发送 SQL 请求给 DBMS 服务器。DBMS 工作者处理每一条请求并将结果返回客户端。在下文中，我们将研究商业数据库把 DBMS 工作者映射到系统线程或者进程的不同方法。当需要区分进程和线程的区别时，我们将分别称之为工作者线程或者工作者进程，如果不需要，那么我们直接用 DBMS 工作者来表示。

2.1 单处理机和轻量级线程

在这一部分，我们介绍一个简单的 DBMS 进程模型的分类方法。尽管先进的 DBMS 很少完全按照我们将介绍的内容来构建，但是在这个基础上，我们可以更详细地讨论新一代产品。现今的每一款数据库系统的核心都是下列所述模型的扩展或者增强。

我们首先做两个简单的假设（在后文的讨论中我们将放宽条件）：

- **系统线程支持**：我们假设操作系统内核有效地支持线程，并且一个进程可以拥有很多线程。我们假设线程存储空间很小，所以，它们在切换的时候并不需要太多的代价。在现代操作系统中这个假设基本是成立的，但是，在 DBMS 设计之初却一定不是这样的。因为，在很多平台上，系统线程或者不支持，或者效率落后，所以，很多 DBMS 系统在开发时没有用到系统线程支持。
- **单处理机**：我们假设我们的设计针对仅拥有一个 CPU 的一台机器。因为多核系统已经广泛存在，所以即便是在低端机型方面，这个假设也是不现实的。但是，该假设可以简化我们最初的讨论。

通过上下文我们了解到，DBMS 拥有三个天然的进程模型性质。从最简单到最复杂依次为：（1）每个 DBMS 工作者拥有一个进程；（2）每个 DBMS 工作者拥有一个线程；（3）进程池。尽管该模型被简化了，但是，今天的商业 DBMS 系统都遵循这三条性质。

2.1.1 每个 DBMS 工作者拥有一个进程

每个 DBMS 工作者拥有一个进程的模型（如图 2-1 所示），早期被用于 DBMS 开发，并且今天仍被一些商业系统所采用。这个模型由于 DBMS 工作者被直接映射到系统进程，因而相对容易实现。系统调度管理 DBMS 工作者的运行时间，而 DBMS 编程人员也可以利用系统的一些保护措施来排除标准错误，如内存溢出。而且，不同的编程工具，比如调试器和存储检测器，都与这种进程模型相适应。使这个模型变复杂的是 DBMS 连接所需要的内存中的数据结构，包括锁机制和缓冲池（在第 6.3 节和第 5.3 节将详细讨论）。这些共享的数据结构必须分配在所有 DBMS 进程都可以访问的共享内存中。这需要系统支持（一般系统均支持）和 DBMS 相关的代码。实际中，由于这种模型需要广泛使用共享内存，使得地址空间分离的优势被削弱了。

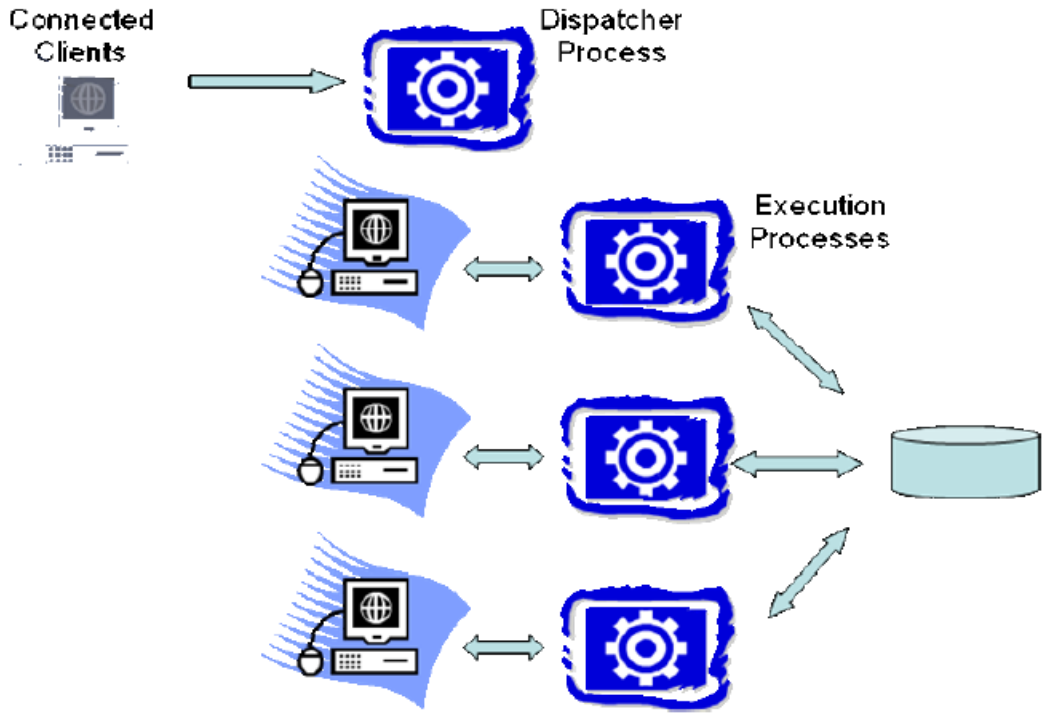

图 2-1 每个 DBMS 工作者拥有一个进程的模型

如果把“扩展到大量并发连接”作为衡量标准，那么，每个 DBMS 工作者拥有一个进程的模型并不十分有效。这是因为，进程相对于线程而言，拥有更多的环境变量并且消耗更

多的存储空间。进程切换需要切换安全的上下文环境、存储空间变量、文件和网络句柄列表以及其他一些进程上下文。这在线程切换时是不需要的。尽管如此，每个 DBMS 工作者拥有一个进程的模型，还是比较受欢迎的，并得到 IBM DB2、PostgreSQL 和 Oracle 的支持。

2.1.2 每个 DBMS 工作者拥有一个线程

在每个 DBMS 工作者拥有一个线程的模型（图 2.2）中，一个多线程进程负责所有的 DBMS 工作者的工作。一个调度线程监听新的 DBMS 客户端连接。每个连接都被分配一个新的线程。当每个客户端提交 SQL 请求时，该请求都由对应的 DBMS 工作者线程来执行。这个线程在 DBMS 进程中运行，一旦运行结束，结果将返回给客户端，然后该线程等待来着这个客户端的下一个请求。

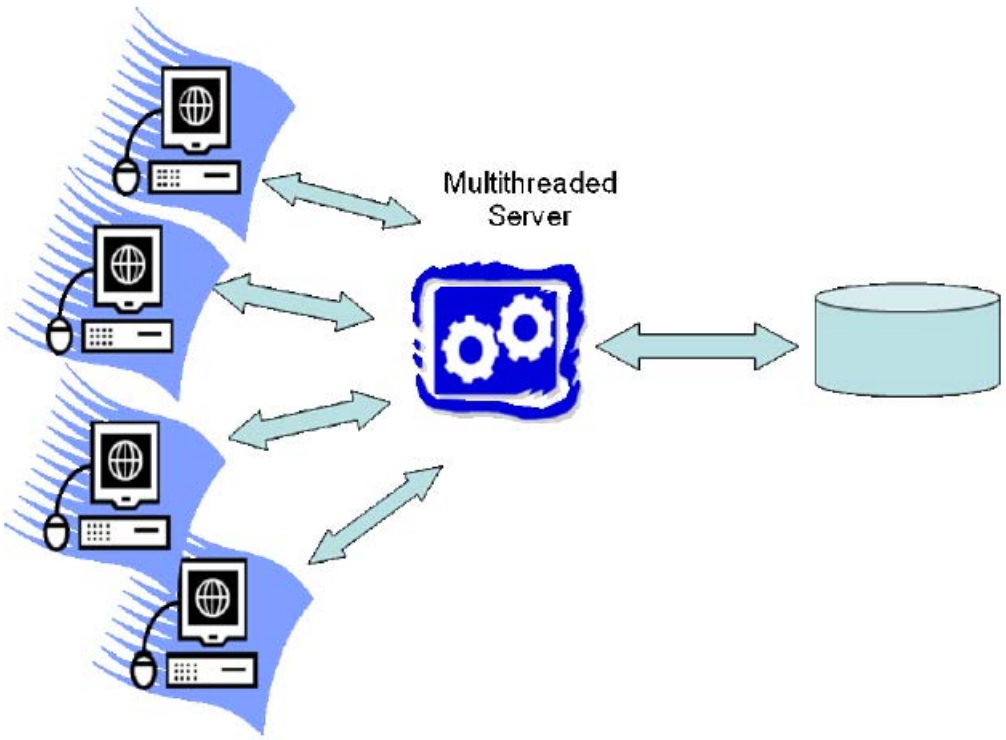

图 2-2 每个 DBMS 工作者拥有一个线程的模型

多线程编程在这种架构下有以下困难：操作系统对线程不提供溢出和指针的保护；调试困难，尤其是在运行情况下更是如此；由于不同操作系统在线程接口和多线程扩展性方面的不同，使得软件可移植性较差。“每个 DBMS 工作者拥有一个线程模型”中的很多编程问题，同样存在于每个 DBMS 工作者拥有一个进程模型中，因为它们都需要共享内存的使用。尽管不同操作系统在线程 API 的多样性方面，近年来不断缩小，但不同平台之间微小的差

别还是不断带来调试上的麻烦。如果不考虑实现上的困难，那么，每个 DBMS 工作者拥有一个线程的模型，还是可以很好地扩展到高并发系统中的，并且这种模型在一些现代 DBMS 产品中也被使用了，如 IBM DB2、微软 SQL Server、MySql、Informix 和 Sybase。

2.1.3 进程池

这个模型是“每个 DBMS 工作者一个进程”这种模型的变体。我们知道，每个 DBMS 工作者拥有一个进程的模型的优点是编程实现较为简单，但是，每一个连接都需要一个进程却是一个缺点。使用进程池（如图 2-3 所示）后，不必每个 DBMS 工作者都分配一个进程，而是由进程池来管理所有 DBMS 工作者。一个中央进程控制所有的 DBMS 客户端连接，每个从客户端到来的 SQL 请求将被分配一个进程池中的进程。SQL 请求处理完之后，结果返回客户端，进程回到缓冲池中准备分配给下一个请求。缓冲池的大小是一定的，且通常不可变。如果一个请求到来而没有进程空闲，那么新的请求必须等待进程。

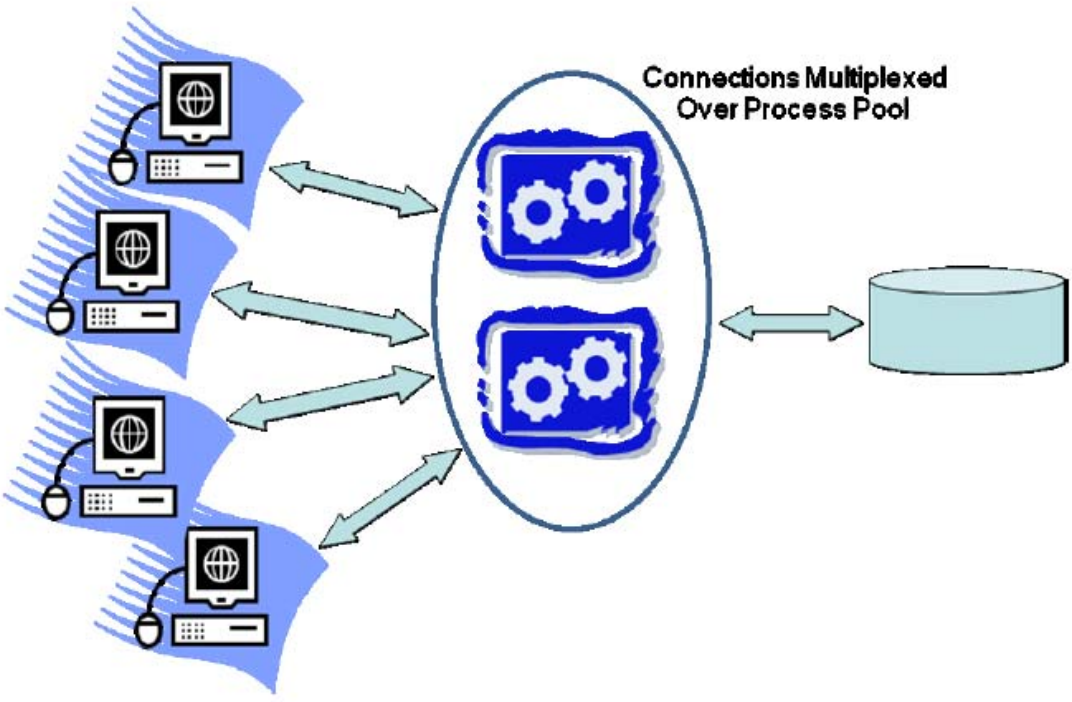

图 2-3 进程池

进程池拥有每个 DBMS 工作者一个进程模型的所有优点，而且，由于只需要少量的进程，所以，其内存使用效率也很高。进程池一般被设计成大小可动态变化的，以此应对大量的高并发请求。当请求负载较低时，进程池可以缩减为较少的等待进程。跟每个 DBMS 工

作者一个进程一样，很多现代 DBMS 产品也支持进程池。

2.1.4 共享数据和进程空间

上述的每一个模型都试图尽可能地独立处理并发客户端请求。但是，DBMS 工作者完全地独立工作是不可能的，因为它们处理的是共享的数据库。对于每个 DBMS 工作者拥有一个线程的模型，数据共享较为简单，因为，所有线程共享内存空间。在其它模型中，共享内存被用来存放共享数据结构。在所有三个模型中，数据必须从 DBMS 传送给客户端。这使得所有的 SQL 请求都必须传给服务器来处理，然后得到的结果需要返回给客户端。要实现这个功能，简单地说，就是使用不同的缓冲区。两种主要的缓冲区是磁盘 I/O 缓冲区和客户端通信缓冲区。我们简要地介绍一下它们以及它们的管理方式。

磁盘 I/O 缓冲区：最常见的工作者之间的数据依赖是读写共享的数据。所以，工作者之间的 I/O 中断是必要的。有两种不同的 I/O 中断需要分别考虑：(1)数据库请求；（2）日志请求。

- **数据库 I/O 中断请求：缓冲池**。数据库提交的数据存放在数据库缓冲池中。对于每个 DBMS 工作者拥有一个线程的模型，缓冲池中的数据以堆结构的形式存放，允许共享该存储空间的所有线程来访问。在另外两种模型中，缓冲池被作为所有进程共享的空间。总的来说，三种模型都是将缓冲池中的数据共享给所有的线程或进程。当一个线程需要从数据库中读取一个数据页时，它产生一个 I/O 请求并且获得空闲的内存空间来存放得到的数据。当需要将缓冲区中的页存入磁盘时，线程产生一个 I/O 请求将缓冲池中的页存入磁盘中的目标地址处。在第 4.3 节中我们将详细讨论缓冲池。
- **日志 I/O 请求：日志尾部**。数据库日志是存储在硬盘上的一组条目（entry）。一个日志条目是在事务处理过程中产生的，它们会被暂时存储在内存队列中，然后，周期性地按 FIFO 顺序刷新至日志磁盘中。这个队列常被称为日志尾部。在许多系统中，有一个独立的进程或者线程负责将它刷新至磁盘。

对于每个 DBMS 工作者拥有一个线程的模型，日志尾部只是一个堆数据结构。在另外两种模型中，它们的设计方式比较类似。一种方法是一个独立的进程负责管理日志。日志记录通过共享内存或者其他有效的通信协议与日志管理器进行交互。另一种方式是，日志尾部像上文中提到的缓冲池那样分配给共享内存。关键的一点是，所有的线程或者进程在处理客

户端请求时，需要写日志记录并将日志尾部刷新至磁盘。

一种重要的日志刷新方法是提交事务刷新。一个事务在日志记录被刷新到日志存储器之前不能被成功提交。这意味着，客户端代码必须等待日志刷新，DBMS 服务端在刷新之前必须保存所有资源。日志刷新请求可能会被推迟一段时间，这样可以实现在一个 I/O 请求中批量提交记录。

客户端通信缓冲区：SQL 通常被用于“拉”（pull）模型，也就是说，客户端通过重复发送 SQL FETCH 请求，不断地获取结果元组，SQL FETCH 每个请求会获取一个或多个元组。大多数 DBMS 尽可能地在 FETCH 流到来之前做一些数据预提取工作，以此确保在客户端请求之前将结果存入队列。

为支持预提取功能，DBMS 工作者使用客户端通信套接字作为元组队列。将来更复杂的方法是实现客户端游标缓存功能，并使用 DBMS 客户端存储近期即将被访问的结果，而不是依赖于操作系统通信缓冲区。

锁表：锁表由所有的 DBMS 工作者共享，由锁管理器实现数据库锁机制。锁共享技术类似于缓冲池共享，并可以用类似的方法实现 DBMS 开发需要的其他数据结构共享。

2.2 DBMS 线程

前文中我们简单描述了 DBMS 进程模型。我们假设系统线程有良好的的性能，DBMS 着眼于单处理机系统。在本节剩余的篇幅中，我们放宽第一个假设，然后来看看它对 DBMS 实现的影响。接下来讨论多进程和并行化操作。

2.2.1 DBMS 线程

当前的大多数 DBMS 技术，都离不开 19 世纪 70 年代开始的系统研究和 19 世纪 80 年代开始的商业化发展。在数据库开发的早期阶段，标准操作系统的技术无法用到数据库开发中。高效的操作系统线程支持就是其中一种技术。直到 19 世纪 90 年代，操作系统线程的出现和广泛应用，才使得数据库开发发生了很大的变化。即便是今天，操作系统线程开发都没有很好地支持 DBMS 的工作负载[31,48,93,94]。

受历史发展的影响以及其他一些原因，很多广泛应用的 DBMS 在开发上并不依赖于操作系统线程。一些完全不涉及线程，而是使用每个 DBMS 工作者拥有一个进程或者进程池模型。其他的 DBMS，比如一个 DBMS 工作者拥有一个线程的模型，需要一个方案来应对

那些没有好的内核线程的系统。一些技术领先的 DBMS 解决该问题的方式是，开发了自己的高效轻量级的线程包。这些轻量级线程或者说 DBMS 线程，取代了上文提到的系统线程。每个 DBMS 线程都管理自己的变量，通过非中断的异步接口来编写中断操作，并通过调度器来调度任务。

轻量级线程并不是一个新的概念，在文献[49]中以一种怀旧的方式介绍了它，这个概念如今在针对用户接口的循环事务编程方面被广泛应用。此概念在操作系统相关文献[31,48,93,94]中也被不断研究。这个架构为我们提供了快速的任务切换和易移植性，它的代价是需要在 DBMS 中重复实现许多操作系统逻辑。

2.3 标准练习

在现今主要的 DBMS 中，我们可以看到 2.1 节中介绍的所有三种架构以及它们的一些有趣的变化。IBM DB2 是支持四种进程模型的很有趣的例子。对于线程支持良好的操作系统，DB2 默认的模型是每个 DBMS 工作者拥有一个线程，它也支持 DBMS 工作者多路复用一个线程池。当运行在没有线程支持的系统上时，DB2 默认的模型是每个 DBMS 工作者拥有一个进程，同时也支持 DBMS 工作者多路复用一个进程池。

我们对 IBM DB2、MySql、Oracle、PostgreSQL 和 Microsoft SQL Server 所支持的进程模型做一个总结：

每个 DBMS 工作者拥有一个进程：

这是最直接的进程模型，同时也得到了广泛的应用。DB2 在不支持线程的系统上，默认使用每个 DBMS 工作者拥有一个进程的模型，在支持线程的系统上，默认使用每个 DBMS 工作者拥有一个线程的模型。这也是 Oracle 进程模型的默认设计。Oracle 也支持上文提到的进程池。PostgreSQL 在所有的操作系统上都只运行每个 DBMS 工作者拥有一个进程模型。

每个 DBMS 工作者拥有一个线程：

该模型有两个种类，在目前来看十分高效：

1. 每个 DBMS 工作者拥有一个系统线程：IBM DB2 运行在有良好系统线程支持的系统上时，默认使用该模型；MySQL 同样使用该模型。
2. 每个 DBMS 工作者拥有一个 DBMS 线程：在这个模型中，DBMS 工作者的调度是通过系统进程或线程去调度轻量级线程来实现的。这个模型规避了系统调度的一些

问题，但是，它开发代价较高，没有良好的开发工具，也需要运营商的长期维护。它有两种子模型：

①通过系统进程来调度 DBMS 线程：轻量级线程的调度是由一个或多个系统进程来完成的。Sybase 和 Informix 支持该模型。现今很多系统采用这个模型来开发实现 DBMS 线程的调度，以此调度 DBMS 工作者发挥多处理器的作用。然而，并不是所有采用这个模型的系统都实现了线程迁移：将 DBMS 线程再分配给不同的系统进程（如考虑到负载均衡）。

②通过系统线程来调度系统进程：微软 SQL Sever 以可选择的方式支持这种模型（默认的模型为 DBMS 工作者多路复用线程池）。SQL Sever 中这个选项叫做 Fibers，被用做应对高频事务处理，但是很少被使用。

进程/线程池：

在这个模型中，DBMS 工作者共用一个进程池。随着系统线程支持性能的不断提高，依赖线程池而不是进程池的新的变种开始出现。在后面这种变种模型中，DBMS 工作者复用系统的一个系统线程池：

1. DBMS 工作者共用进程池：这个模型相对于每个 DBMS 工作者拥有一个进程的模型而言，有更高的内存使用效率。在系统没有良好的线程支持的情况下，它也易于与系统对接，并且在大量用户的情况下表现稳定。这个模型是 Oracle 数据库的可选项，Oracle 建议在大量用户并发操作的情况下使用。Oracle 默认的模型是每个 DBMS 工作者拥有一个进程。这两个选项对于 Oracle 所适用的大多数操作系统而言都是支持的。
2. DBMS 工作者共用线程池：微软 SQL Server 默认使用该模型，且超过 99%的 SQL Server 产品使用该模型来运行。为有效支持上万的并发连接用户，SQL Server 提供可选支持，允许系统线程调度 DBMS 线程。

我们将在下一章中谈到，现今大多数商业数据库支持内部查询并行化：并行多线程执行一条查询语句或者该语句的一部分。这一节中我们之所以提到这一点，是因为并行化内部查询，使得多个 DBMS 工作者暂时分配给一条 SQL 请求。除了一个客户端连接将拥有多个 DBMS 工作者来处理其请求之外，之前提到的进程模型与该情况并无冲突。

2.4 准入控制

随着多用户系统负载不断升高，吞吐量将达到上限。在这一点上，当系统挂起时应当减少吞吐量。对于操作系统来说，“挂起”经常是由内存问题造成的：DBMS 无法保证在缓冲池中存放数据库页数据的工作空间，导致出现不断的页替换情况。在数据库系统中，一些特殊的操作比如排序和哈希连接往往会造成大量的内存消耗。有时候，系统挂起也会发生在锁竞争上：事务处理发生死锁需要回滚并重启。因此，任何好的多用户系统都有准入控制机制，在系统没有充足资源的情况下，新的任务不被接受。如果拥有一个好的准入控制器，系统将在过载情况下发生比较优雅的性能衰退：事务延迟将随着到达率的增加而适当增加，但吞吐量一直保持在峰值。

DBMS 的准入控制可以在两个层面上来实现：

第一个层面是，一个简单的准入控制可以通过进程来确保客户端连接数处于一个临界值之下。这就避免了网络连接数这类基础资源的过度消耗。在一些 DBMS 中，这种控制并没有被提供，我们可以假设该功能被系统的其他部分实现了，如应用层、事务处理层或者网络服务层。

第二个层面是，直接在 DBMS 内核关系查询处理器上实现。准入控制器在查询语句转换和优化完成后执行这一步操作，由该操作来决定，是否要推迟执行一个查询，是否要使用更少的资源来执行查询，以及是否需要额外的限制条件来执行查询。准入控制器依靠查询优化器的信息来执行，如查询所需的资源以及系统并发资源等信息。特殊情况下，优化器的查询计划可以：（1）确定查询所需要的磁盘设备以及对磁盘的 I/O 请求数；（2）根据查询中的操作以及要求的元组数目判断 CPU 负载；（3）评估查询数据结构的内存使用情况，包括在连接和其他操作期间的排序和哈希所消耗的内存。正如之前描述的那样，上面的第 3 点对于准入控制而言是最为关键的，因为，内存压力通常是引起“抖动”的主要原因。因此，许多 DBMS 把内存使用情况和活跃的 DBMS 工作者的数量作为主要的准入控制标准。

2.5 讨论及其他资料

进程模型的选择对 DBMS 的表现有很大的影响。因此，商业数据库往往支持多个进程模型。从工程师的角度来看，在所有操作系统以及各种负载级别上使用单进程，很显然会更加简单明了。但是，由于操作系统和应用场合的多样性，三种 DBMS 都选择了支持多种模

型。

展望未来，由于硬件瓶颈的变化以及网络的规模和多样性，近年来很多人致力于为服务器系统研究新的进程模型[31,48,93,94]. 在这些设计中，有一种想法是把一个服务器系统分解成一系列独立调度的引擎，并且在这些引擎之间进行异步和批量的消息传输。这有一些类似于上文中提到的进程池模型，即在多个请求之间重用工作者单元。这类最新研究的主要创新是，用一种更加小范围的、面向特定任务的方式来执行一些功能。这使得 DBMS 工作者与请求的关系变为多对多的，即一个查询的工作由多个 DBMS 工作者来完成，一个 DBMS 工作者负责多个查询的同样任务。这种结构带来了更为灵活的调度方式，比如，它允许一个工作者完成许多查询的任务（可能提高系统的整体吞吐量），或者允许一个查询由多个工作者共同完成（可能减少查询操作的延时）。在一些例子中，它在处理器局部性方面显得更有优势，在硬件未命中的情况下，能更好地防止 CPU 空闲。更多关于这一点的研究可以参考 StagedDB 的研究项目[35]，这也是很好的阅读材料。

第 3 章 并行架构：进程和内存协调

并行硬件在现代的服务器中已经是一个不争的事实，它们存在于各种各样的配置中。在这一章，我们将总结标准的数据库管理系统术语，接着分别讨论进程模型和内存协调（memory coordination）问题。

3.1 共享内存

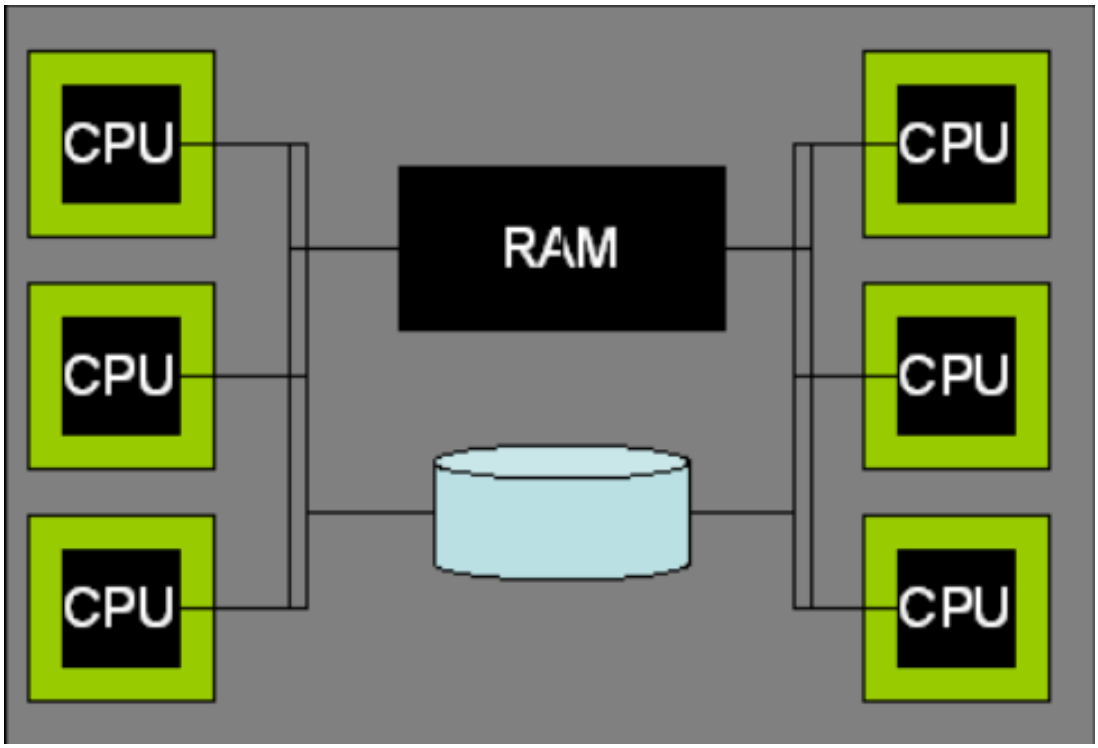

图 3-1 共享内存体系架构

在一个共享内存的并行系统中（如图 3-1 所示），所有的处理机可以使用相同的内存和硬盘，并且拥有大致相同的性能。这个架构现在已经是一个标准。大多数服务器硬件都包含 2 个到 8 个的处理器。高端的机器可以包含数十个处理器，可以提供更高的处理器资源，但是，也往往价格更加昂贵。高度并行共享内存的机器是硬件行业最后剩下的摇钱树之一，并

大量使用在高端在线事务处理应用程序中。服务器硬件的代价与管理系统的代价相比，不免就相形见绌了，所以，购买少量大型的、昂贵的系统有时候被看作是可以接受的折中方案。

多核处理器在单一芯片上支持多个处理内核和共享一些基础结构，如高速缓存（cache）和内存总线。这使得它们在编程方面非常类似于共享内存的架构。如今，几乎所有的数据库部署都涉及到多个处理器，并且每个处理器都不只一个 CPU。DBMS 架构需要充分利用这种潜在的并行。幸运的是，在第二章描述的所有三个 DBMS 的架构，可以在现代共享内存硬件架构上运行得很好。

共享内存机器的进程模型很自然地遵循单处理机的方式。事实上，大多数数据库系统都是从他们最原始的单处理机器实现，然后演化为共享内存实现。在共享内存的机器中，操作系统通常支持作业（进程或线程）被透明地分配到每个处理器上，并且共享的数据结构继续可以被所有的作业所访问。所有三模型可以在这些系统上运行良好，而且支持多个独立的 SQL 并行请求的执行。面临的主要挑战是修改查询执行层，从而利用将一条单一的查询语句并行到多个处理器上能力。我们将推迟到第 5 章介绍。

3.2 无共享

一个无共享的并行系统是由多个独立计算机的集群组成的，这些计算机可以高速地通过网络进行互连通信，或者逐渐频繁地在商业网络组件上通信。对于给定的一个系统，无法直接访问另一个系统的内存和硬盘。

无共享系统并没有提供抽象的硬件共享，协调不同机器的任务完全留给了 DBMS。DBMS 支持这些集群最常用的技术就是在集群中的每台机器或每个节点上运行它们标准的进程模型。每一个节点都能够接受客户端 SQL 请求、访问需要的元数据、编译 SQL 请求、进行数据访问，正如上述描述的单一共享内存一样。主要的区别就是，集群中的每个系统仅仅保存一部分的数据。对于每一个 SQL 请求，无共享系统中的每一个节点不只是单独执行查询本地的数据，这个请求会被发送到集群中的其他成员，然后所有的计算机并行地执行查询本地所保存的数据。每个表格会通过*水平数据分区*传播到集群的多个系统中，因此，每个处理器可以独立于其他处理器执行。

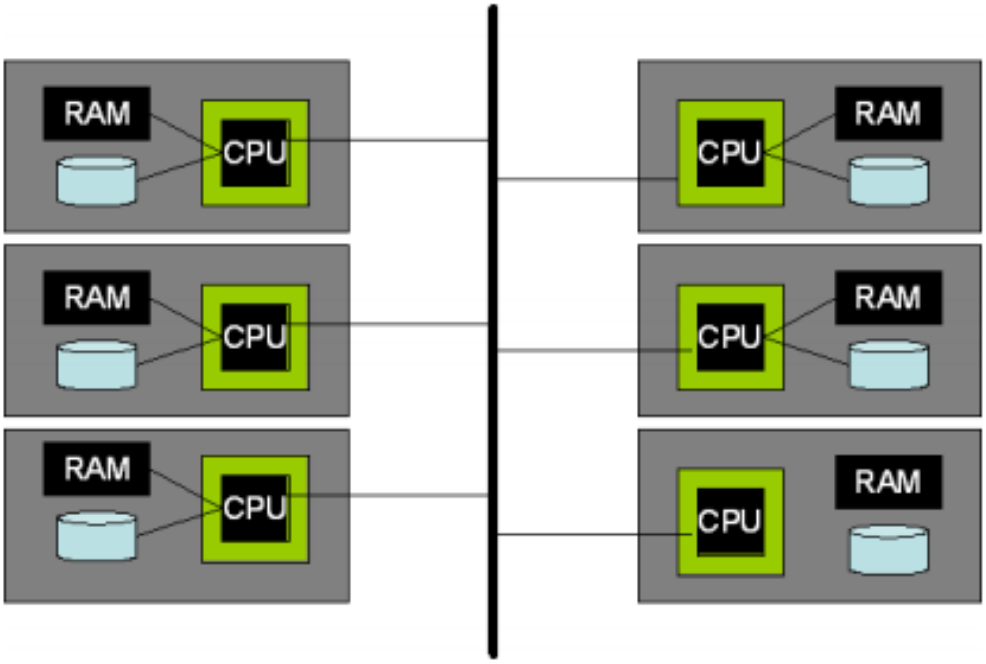

图 3-2 无共享体系架构

数据库中的每一个元组被分配到到单独的机器，因此，每一张表被水平地切分，然后分布到整个系统。典型的数据分区方案包括：元组属性基于哈希的分区，元组属性基于范围的分区，round-robin，以及混合型分区方案（前两种方案的混合）。每一个单独的计算机都负责访问、锁定、记录本地磁盘上的数据。在查询执行期间，查询优化器会选择如何对表进行重新水平分区，然后把中间结果分布到各个计算机上以满足查询的需要，并且给每一台机器分配一个作业的逻辑分区。不同的计算机上的查询执行部件，将数据请求和元组传输到其他计算机中，但是，没必要传输任何线程状态和其他低级的信息。数据库元组采用基于值的分区导致的一个很自然的结果就是，在这些系统中，只需要最少的协调工作。然而，为了得到很好的性能，就需要很好地数据分区。这就给数据库管理员一个重大的负担——合理明智地确定表的分布。同时给查询优化器一个重大的负担——需要在负载分区方面做得很好。

这简单分区方案并不能处理 DBMS 上所有的问题。例如，必须采取明确的跨处理器协调来处理事务完成，提供负载平衡以及支持某种维护任务。例如，对于一些像分布式死锁检测和两阶段提交[30]等问题，处理器之间必须交换明确的控制信息。这就需要额外的逻辑，而且如果做得不好，就可能是个性能瓶颈。

同时，在无共享系统中，局部故障是必须被妥善管理的。在一个无共享系统中，一个处理器发生故障通常会导致整个系统的停止运行，因此，整个 DBMS 也会停止运行。在一个无共享系统中，集群中一个单一的节点发生故障，并不一定会影响到其他的节点。但是，这

肯定会影响 DBMS 的整体运行，因为，失败的节点里寄存着数据库中部分的数据。在这种情形下，至少有三种可行的方法。第一种办法就是，如果有任何一个节点发生故障，就停止运行所有的节点；这本质上是模拟在一个共享内存系统中将会发生的事情。第二种方法，在 Informix 上称为“数据跳跃”，允许在正常的节点上继续执行查询，而跳过故障节点的数据。这在数据的可用性比结果的完整性更重要的情况下是很有用的。但是，竭尽全力的结果，不具备良好定义的语义，对于许多负载而言，这不是一个有用的选择——尤其是因为在多层系统中 DBMS 通常被用作“记录存储库”，需要在更高的层面（通常是一个应用服务器）来权衡可靠性和一致性。第三种方法是采用冗余方案，范围从完整的数据库失败恢复（需要两倍的计算基数量和软件数量）到类似于链式分簇的细粒度冗余[43]。在这后面的技术中，元组副本会分布在集群中的多个节点。链式分簇比其他更简单机制的优势是：(a)需要部署更少的机器，就可以保证比其他简单机制获得更好的可用性；(b)当一个节点发生故障时，系统负载将会相当均匀地分配到剩下的节点：剩下的 n-1 个节点，每个完成原来工作的 n/(n-1)。这种形式的线性性能下降，会随着节点的失败而持续。实际上，许多通用的商业化系统是介于中间的，既不是粗粒度的全数据库冗余，也不是细粒度的链式分簇。

无共享架构如今已是相当普遍，并且拥有无与伦比的可拓展性与成本特性。它主要用在极高端应用中，通常是决策支持应用和数据仓库。在一个硬件架构的有趣组合中，一个无共享的集群通常是由许多节点构成的，而每个节点都是共享内存的多处理器。

3.3 共享磁盘

如图 3-3 所示，在一个共享磁盘的并行系统中，所有的处理器可以访问具备大致相同性能的磁盘，但是，不能访问彼此的 RAM。这种架构是相当普遍的，两个突出的例子是 Oracle RAC 和 DB2 for zSeries SYSPLEX。随着存储区域网络（SAN：Storage Area Networks）的普及，共享磁盘在最近几年已经变得越来越普遍。一个 SAN 允许一个或多个逻辑磁盘被安装到一个或多个宿主系统上，这样可以使得创建共享磁盘配置变得较为容易。

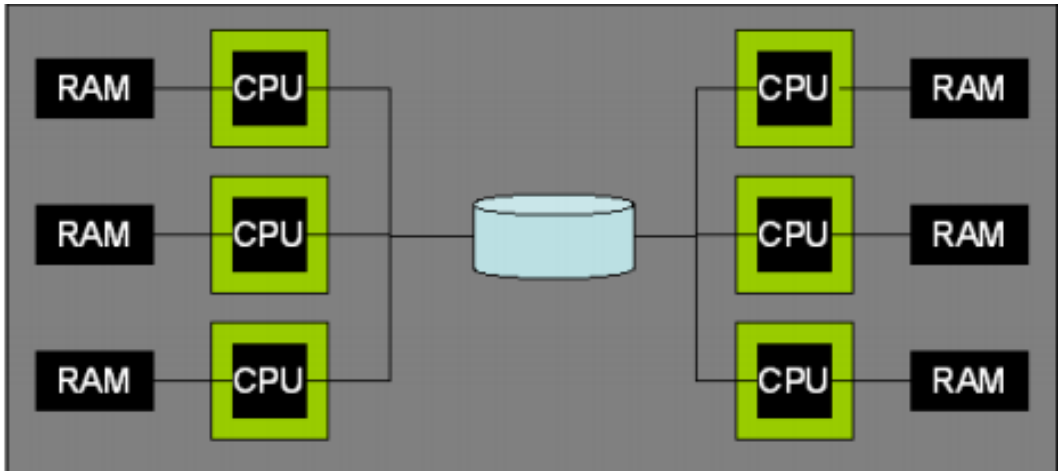

图 3-3 共享磁盘体系架构

共享磁盘系统相对于无共享系统而言的一个潜在的优势就是，它们管理成本较低。共享磁盘系统的 DBAs 没必要为了实现并行而考虑对表进行分区并分布到不同计算机上。但是，大型的数据库仍然需要分区，所以，对于大规模数据库，二者的区别不是很显著。共享磁盘架构的另一个引人注目的特征是，单个 DBMS 处理节点发生故障不会影响其他节点访问整个数据库。这一点既不同于共享内存系统（作为一个整体发生故障），也不同于无共享系统（由于某个节点发生故障会导致至少无法访问一些数据，除非使用一些其他数据冗余机制）。然而，即使有这些优势，共享磁盘系统仍然容易受到单点故障。当数据未到达存储子系统时，如果数据由于硬件或软件故障而损坏或发生其他方式的毁坏，则系统所有的节点只能访问这些损坏的页面。如果存储子系统使用 RAID 或者其他数据冗余技术，则损坏的页面将会被冗余储存，但所有的副本仍然是损坏的。

因为在一个共享磁盘系统中不需要对数据进行分区，所以数据可以被复制到 RAM，并在多个计算机上进行修改。与共享内存系统不同的是，共享磁盘系统不存在很自然的内存位置来协调数据的共享，每个计算机都有为锁和缓冲池页面准备的本地内存。因此，需要在多台计算机之间进行显示的数据共享协调。共享磁盘系统依赖于一个分布式锁管理设备和一个管理分布式缓冲池的高速缓存一致性协议。这些都是复杂的软件组件，并且对于具备竞争性的负载而言会成为瓶颈。有些系统是由硬件子系统来实现锁管理的，比如 IBM zSeries SYSPLEX。

3.4 非均衡内存访问

非均衡内存访问（NUMA：Non-Uniform Memory Access）系统，在一个拥有独立内存

的集群系统中，提供了共享内存编程模型。集群中的每个系统都能快速访问本地内存，然而高速集群中每个计算机之间的远程访问，会存在一定程度上的互连延迟。这个架构的名字来自于这种内存访问时间的不一致。

NUMA 的硬件架构是处于无共享系统和共享内存系统之间的一个有趣的中间地带，它们比无共享集群更容易编程，同时，比共享内存系统拥有更大规模的处理器，这样就可以避免共享点的争用，例如共享内存系统总线。

NUMA 集群在商业上并没有获得广泛的成功，但是，NUMA 设计概念正在被共享内存多处理器领域所采用。伴随着共享内存多处理器系统已经扩展到大量处理器的规模，它们在内存架构上已经展示了逐渐增长的非均衡性。通常大规模的共享内存多处理器系统的内存被分为多个部分，每个部分与系统中处理器一个小子集相关联。每一个内存和 CPU 相结合的子集通常被称为一个“pod”。每一个处理器访问本地的 pod 内存的速度，要稍快于访问远程 pod 的内存。NUMA 设计模式的使用已经允许共享内存系统扩展到更多数量处理器的规模，从而使得 NUMA 共享内存多处理器如今已经非常普遍了，然而，NUMA 集群并没有成功取得任何显著的市场份额。

DBMS 可以在 NUMA 共享内存系统上运行的一种方式是，通过忽略内存访问的非均衡性。如果非均衡性较小的时候，这种方式还勉强可以接受。但是，当近程内存访问时间与远程内存访问时间的比率从 1.5:1 升到 2:1 的范围时，DBMS 需要使用优化，以避免内存访问瓶颈。这些优化可以有不同的形式，但是，全部都要遵循相同的基本方法：（1）当为一个处理器分配内存使用的时候，尽量使用本地的内存（避免使用远程内存）；（2）如果可能的话，保证一个 DBMS 使用者被安排到与之前相同的硬件处理器。这种组合允许 DBMS 负载在更大的规模上运行良好，同时，共享内存系统拥有一些内存访问时间的非均衡性。

尽管 NUMA 集群已经几乎全部消失了，但是，编程模式和优化技术对于当代的 DBMS 系统仍然很重要，因为许多大规模的共享内存系统在内存访问性能上有着显著的非均衡性。

3.5 线程和多处理器

当我们除去第 2.1 节中关于单处理机硬件的第二个简化假设的时候，使用 DBMS 线程来实现每个 DBMS 工作者一个线程时，一个潜在的问题就会立即变得非常明显。第 2.2.1 节描述的轻量级 DBMS 线程包的自然实现方式就是，所有线程运行在一个单一的操作系统进程中。不幸的是，一个单一的进程一次只能在一个处理器上执行。因此，在一个多处理器系

统中，DBMS 一次只能用一个处理器，而系统中其他的处理器将会被闲置。早期的 Sybase SQL Server 架构就受到这样的限制。在 90 年代，随着共享内存多处理器越来越受欢迎，Sybase 很快就更改了系统架构，开发出了多系统进程的架构。

当在多进程上运行多个 DBMS 线程的时候，就会出现这样的时刻，一个进程拥有大部分的工作，其他进程（处理器也一样）就被闲置了。在这种情形下，为了让这种模式工作得更好，DBMS 必须实现在进程之间迁移线程。

当 DBMS 线程映射到多个操作系统进程的时候，需要作出以下决定：（1）需要使用多少个操作系统进程；（2）怎么将 DBMS 线程分配到操作系统线程；（3）怎么分配到多个操作系统进程。 一个好的经验法则就是每个物理处理器产生一个进程。这样就可以最大化硬件固有的并行度，同时最小化每个进程的内存开销。

3.6 标准的操作规程

对于并行度的支持，趋势就和上节所述类似，即大多数主流的 DBMS 支持多种模式的并行。 由于共享内存系统（SMPs，多核系统和二者的结合）在商业上的普及，所有主要的 DBMS 供应商都支持共享内存的并行。但是，我们开始看到，在多节点的集群并行方面（可以采用共享磁盘或者无共享的设计），不同厂商提供了不同的支持。

- *共享内存*：所有主要的商业 DBMS 提供对共享内存并行的支持，包括：IBM DB2、Oracle 和 Microsoft SQL Server；
- *无共享*：这种模型被 IBM DB2、Informix、Tandem 和 NCR Teradata 所支持；Greenplum 提供了一种 PostgreSQL 的定制版本，可以支持无共享并行；
- *共享磁盘*：这种模型被 Oracle RAC、RDB(是甲骨文从数字设备公司收购而来的)和 IBM DB2 的 zSeries 所支持。

IBM 销售不同的 DBMS 产品，选择在一些产品上实现共享磁盘支持，在其他产品上支持无共享模型。到目前为止，没有一款领先的商业系统在一个单一的代码库中同时支持无共享模式和共享磁盘模式。Microsoft SQL Server 也没有实现。

3.7 讨论与附加材料

上述的设计代表了在不同服务器系统中所使用的一些硬件/软件架构模式。虽然这些设计率先出现在 DBMS 中，但是，在其他数据密集领域，包括像 Map-Reduce（有越来越多的

用户使用它来处理各种各样的自定义数据分析任务）这样低级别的可编程的数据处理后端，这些设计理念正在获得越来越多的认可。然而，尽管这些设计理念正在更加广泛地影响计算，数据库系统并行性的设计仍然有新的问题产生。

并行软件架构在未来十年将迎来一个重要挑战，这个挑战来自于处理器供应商开发出新一代“众核”架构的构想。这些设备将会引进一种新的硬件设计理念，即在一个单一的芯片上拥有数十、数百甚至数千的处理单元，通过高速芯片网络互联，但是，仍然保留了许多现有的瓶颈，比如访问芯片外的内存和磁盘。这将会导致在磁盘和处理器的内存路径上产生新的不平衡和瓶颈，这就必定需要重新审视 DBMS 架构，以满足硬件性能潜力。

在面向服务的计算领域，一些相关的架构转向更大的规模已经是可以预见的。这里的核心理念是，由数以万计的电脑组成的大数据中心来为用户安排处理（硬件和软件）。在这样的规模下，只有实现高度自动化，才能负担得起应用程序和服务器管理。没有什么管理任务可以随着服务器的数量一直扩展。而且，集群中通常使用更不可靠的商业服务器，故障是不可避免的，从故障中恢复就需要完全的自动化操作。在这样规模下的服务，将会每天出现磁盘故障，每星期出现服务器故障。在这样的环境中，管理数据库的备份通常被整个数据库的冗余在线副本所取代，这些副本维护在不同磁盘上不同的服务器中。根据数据的价值，冗余副本可能保存在不同的数据中心。仍然可以采用自动离线备份，从而可以从应用程序、管理工作或者用户错误中恢复。然而，从最常见的错误和故障中恢复是将故障快速转移到冗余的在线副本。冗余可以用多种方式实现：(a)在数据存储级的复制（存储区域网络）；(b)在数据库存储引擎级别的复制（将在第 7.4 节讨论）；(c)由查询处理器执行冗余查询（第 6 章）；或者(d)在客户端软件级别自动生成冗余数据库请求（例如，WEB 服务器或应用程序服务器）。在一个更高的解耦水平，在实际应用中为多个具备 DBMS 功能的服务器进行分层部署是很普遍的，以减少“DBMS 记录”的请求率。这些机制包括各种形式的、针对 SQL 查询的中间层数据库缓存（包括专业的内存数据库，比如 Oracle TimesTen），以及更多被配置成服务于这个目的的传统数据库。在部署层次的更高层面，许多支持编程模型（比如 Enterprise Java Bean）的、面向对象的“应用-服务器”架构，可以配置成为配合 DBMS 工作的、应用对象的事务缓存。然而，这些各式各样的方案的选择、设置和管理，仍然是不标准的，并且很复杂，普遍接受的优秀模型仍然难以实现。

第 4 章 关系查询处理器

前面的章节强调了一个 DBMS 的宏观架构设计问题。现在，在接下来的章节中我们开始以更细的角度讨论设计问题，依次研究每一个主要的 DBMS 组件。接着第 1.1 节的讨论，我们开始从系统的顶部查询处理器开始，然后在随后的几个章节，我们向下介绍存储管理、事务处理以及实用工具。

一个关系查询处理器以一个 SQL 语句作为输入，然后进行验证，优化成为一个程序数据流执行计划，并且在获得准入许可以后可以代表一个客户程序执行数据流程序。然后，客户程序获取（“拉”）结果元组，通常一次一个元组或一小批元组。关系查询处理器的主要组件已经在图 1-1 介绍过了。在本章内容中，我们关注查询处理器和“存储管理器的访问方法的一些非事务处理方面”。一般而言，关系查询处理可以被看作是一个单用户、单线程任务。正如后面第五章描述的那样，并发控制是由系统较低层透明控制的。这个规则唯一的例外就是，当 DBMS 操作缓冲池页面的时候，DBMS 必须明确“固定”(pin)和“不固定”(unpin)缓冲池页面，这样就可以使它们在简短并且关键的操作执行时驻留在内存中，我们将在第 4.4.5 节讨论这点。

在本章中，我们将重点放在常见的 SQL 命令：数据操作语言（DML: Data Manipulation Language）语句包括 SELECT、INSERT、UPDATE 和 DELETE。像 CREATE TABLE 和 CREATE INDEX 这样的数据定义语言语句（DDL: Data Definition Language）通常是不被查询优化器处理的。这些语句通常是由静态 DBMS 逻辑通过调用存储引擎和目录管理器（在第 6.1 节描述）来实现的。一些 DBMS 产品也已经开始优化 DDL 语句的一个小子集，我们期待这个趋势将会持续。

4.1 查询解析和授权

给定一个 SQL 语句，SQL 解析器主要任务是： (1) 检查这个查询是否被正确地定义；(2) 解决名字和引用；(3) 将这个查询转化为优化器使用的内部形式；(4) 核实这个用户是否被授权执行这个查询。一些 DBMS 将一些或者全部安全检查延后到查询执行时才去做，但是，即使是在这些系统中，解析器仍然负责为查询执行时的安全检查收集所需要的数据。

给定一个 SQL 查询，解析器首先考虑的是在 FROM 子句中每个表的引用。解析器把每个表名规范化为“服务器. 数据库. 模式. 表名”，这被称为“四部分名称”。不支持跨越多个服务器执行查询的系统，只需要把表名规范化为“数据库. 模式. 表名”，而对于每个 DBMS 只支持一个数据库的系统来说，只需要把表名规范化为“模式. 表名”。这种规范化是必须的，因为，用户必须依赖上下文的默认值，这就使得在查询具体化过程中允许单个部分的名称被使用。某些系统支持一个表可以拥有多个名字，称为表的别名，而这些同样需要被完全的表名所代替。

在规范化表名之后，查询处理器开始调用目录管理器，检查表是否被注册到系统目录中。在这一步中，处理器可能也将表的元数据缓存到内部的查询数据结构。基于表格的信息，处理器接着使用目录来保证属性引用是正确的。属性的数据类型被用来消除那些存在于重载函数表达式、比较操作符和常量表达式中的逻辑含义模糊性。例如，考虑表达式（EMP.salary*1.15）<75000。乘法函数和比较操作符的代码，以及假定的数据类型和字符串“1.15”及“75000”的内部格式，都将取决于 EMP.salary 属性的数据类型。这数据类型可能为一个整数，一个浮点数或者一个“money”的值。

其他的标准 SQL 语法检查也被使用，包括元组变量的一致性使用，通过集合运算符（UNION/INTERSECT/EXCEPT）相结合的多个表之间的兼容性，在聚合查询中 SELECT 列表中的属性的使用，以及子查询的嵌套等等。

如果查询被成功解析，下一阶段就是授权检查，以确保用户对查询中引用到的这些表、用户自定义的函数以及其他对象具有适当的权限（SELECT/DELET/INSERT/UPDATE）。一些系统在语句解析阶段执行授权检查。然而这并不总是可行的。例如，支持行级安全检查的系统，直到执行时间才能进行完全的安全检查，因为，安全检查可能依赖于数据值。尽管理论上授权可以在编译时间内静态验证，但是，推迟授权验证到查询计划执行时间是有好处的。将安全检查推迟到执行时间的查询计划，可以在用户之间共享，而且，当安全条件变化时，不需要重新对查询进行编译。因此，部分安全检查通常被推迟到查询计划执行的时候。

在编译期间对约束常量表达式进行约束检查，也是可能的。例如，一个 UPDATE 命令可能有一个形如 SET EMP.salary = -1 的子句。如果约束条件指定为正值，查询甚至没必要执行。但是，把这个检查工作推迟到执行时间，也是相当普遍的。

如果一个查询已经被解析并且通过了验证，那么这个查询的内部形式就会被传递到查询的重写模块进行更深入的处理。

4.2 查询重写

查询重写模块，或重写器，负责简化和标准化查询，而无需改变查询语义。它只能依靠查询目录中的元数据，而不能访问表中的数据。尽管我们说是重写查询，但大多数重写实际操作的是查询的内部表示，而不是原始 SQL 语句文本。查询重写模块通常输出一个查询的内部表示，这种输出形式和它接受作为输入的内部格式相同。

许多商业系统的重写器是一个逻辑组件，它的实际执行要么发生在查询解析的后期阶段，要么发生在查询优化的前期阶段。例如，在 DB2 中，重写器是一个单独的组件，然而在 SQL 服务器中，查询重写是作为一个查询优化器的前期阶段完成的。尽管如此，单独考虑重写器是很有用的，即使在所有的系统中并不存在显示的架构界线。

重写器的主要职责是：

- **视图重写**：处理视图是重写器主要的传统角色。对于在 FROM 子句中出现的每一个视图引用，重写器都会从目录管理器中检索出视图定义。然后重写查询，用这个视图所引用的表和谓词来替换这个视图，以及将任何对这个视图的引用替换为对这个视图中的表的列引用。这个过程是递归的，直到这个查询表达式里只有表、没有视图。这种视图重写技术，率先是 INGRES[85]为基于集合的 QUEL 语言提出的，在 SQL 上需要一些额外手段去正确处理重复消除、嵌套查询、空值和一些其他棘手的细节。
- **常量运算表达式**：查询重写可以简化常量运算表达式，例如，R.x<10+2+R.y 被重写为 R.X<12+R.Y。
- **谓词逻辑重写**：逻辑重写是应用在基于 WHERE 子句中的谓词和常量的。简单的布尔逻辑往往是用来改进“表达式”和“基于索引的访问方法的能力”这二者之间的匹配程度。例如，一个诸如 NOT Emp.Salary>100000 的谓词，可能被重写为 Emp.Salary<=100000。通过简单的满足性测试，这些逻辑重写甚至可能导致短路查询执行。例如，表达式 Emp.salary<75000 AND Emp.salsary >1000000，可以被 FALSE 替换。这就允许系统返回一个空的查询值，而无需访问数据库。不可满足的查询可能看起来令人难以置信，但是回想一下，谓词可以被“隐藏”在视图定义中，而且不被外部查询的作者知道。例如，上述的查询，可能由于在一个称为“高管”的视图上查询“工资收入较低的员工”而导致的。在 Microsoft SQL Server 并行安装中，不可满足的谓词也形成了“分区消除”的基础：当一个关系通过区间谓词被水平跨

磁盘进行分区时，如果它的区间分区谓词和查询谓词的合取是不可能满足的，那么查询就没必要在一个卷上运行。另一个重要的逻辑重写使用谓词传递性来引入新的谓词。例如，R.x<10 AND R.x=S.y ，暗示着额外添加了一个谓词“AND S.y<10”。增添这些传递谓词增加了优化器选择方案的能力，这些方案在执行的早期阶段就可以过滤数据，尤其是通过使用基于索引的访问方法。

- **语义优化**：在许多情况下，模式的完整性约束是储存在目录中的，可以被用来帮助重写一些查询。这种优化的一个重要例子就是冗余连接消除。这种情形发生在一个外键约束把一个表的某一列（例如，Emp.deptno）绑定到另一个表（Dept）的时候。给定一个这样的外键约束，我们知道，对于每一个 Emp，只有一个 Dept 与之相对应，而且当缺少与 Emp 相对应的 Dept 元组（父母）时，Emp 元组是不可能存在。考虑一个连接两个表但没有使用 Dept 列的查询：

  SELECT Emp.name，Emp.salary

  FROM Emp,Dept

  WHERE Emp.deptno=Dept.dno

  这种查询可以被重写，从而删除 Dept 表（假定 Emp.deptno 被限制为非空），因此，就可以删除这个连接操作。同样，这种看起来令人难以置信的情形通常很自然地发生在视图上。例如，一个用户可能提交一个员工属性的查询，这个属性来自于连接两个表的视图 EMPDEPT。像 Siebel 这样的数据库应用程序使用非常宽的表，它们的底层数据库不支持足够宽度的表，它们就使用多个表和一个基于这些表的视图。如果缺少冗余连接消除机制，这个以视图的方式实现宽表，将会表现出很差的性能。当表的约束条件与查询谓词不兼容时，语义优化器也可以完全避免查询执行。

- **子查询的平面化和其他启发式重写**：在当代的商业数据库管理系统中，查询优化器是最复杂的组件之一。为了把这种复杂性控制在一定程度以内，大多数优化器都独立地优化单个 SELECT-FROM-WHERE 查询块，并不跨块优化。因此，许多系统把查询重写为一种更适合优化器的形式，而不是去想办法使优化器变得更加复杂。这种转变有时候称为查询规范化。规范化的一种类型就是把语义等价查询重写为规范化的形式，尽量确保语义等价查询被优化后可以产生相同的查询计划。另一个重要的启发式方法就是平面化嵌套查询，这样就可以最大程度地为查询优化器单块优化提供机会。由于重复语义、子查询、空值和相关性等问题，在 SQL 中的一些情况下，这将会是非常棘手的。在早期的时候，子查询平面化是一个纯粹的启发式重写，但是，现在一些产品

已经将重写决策建立在代价分析基础之上。其他跨块查询的重写也是可能的。例如，谓词传递性允许谓词在子查询之间被复制[52]。平面化相关的子查询，对于在并行架构上实现良好的性能是尤其重要的：相关子查询会导致“嵌套循环”式的、查询块之间的比较，这将会序列化执行子查询，尽管有可用的并行资源。

4.3 查询优化器

查询优化器的工作就是将一个内部的查询表示转化为一个高效地查询执行计划（如图4-1 所示）。一个查询计划可以被认为是一个数据流图，在这个数据流图中，表数据会像在管道中传输一样，从一个查询操作符（operator）传递到另一个查询操作符。在许多系统里，查询首先被分解为 SELECT-FROM-WHERE 查询块。每个单独的查询块的优化都是使用一些技术来完成的，这些技术与作家 Selinger et al 在 System R 优化器[79]的论文中描述的技术类似。在完成的时候，一些简单的操作符通常被添加到每个查询块的顶部，作为后处理来计算 GROUP BY、ORDER BY、HAVING 和 DISTINCT 子句。然后，不同的块就用一种简单的方式拼合在一起。

生成的查询计划可以表示成多种方式。原始的 System R 的原型系统，将查询计划编译成机器码，而早期的 INGRES 原型系统则生成一种可解释的查询计划。在 19 世纪 80 年代，INGRES 的作者在他的综述论文[85]里，将“可解释的查询计划”视作一个“错误”，但是，摩尔定律和软件工程在一定程度上证明 INGRES 的这种观点是正确的。讽刺的是，在 System R 项目中，一些研究员将“编译成机器码”视作一个错误。

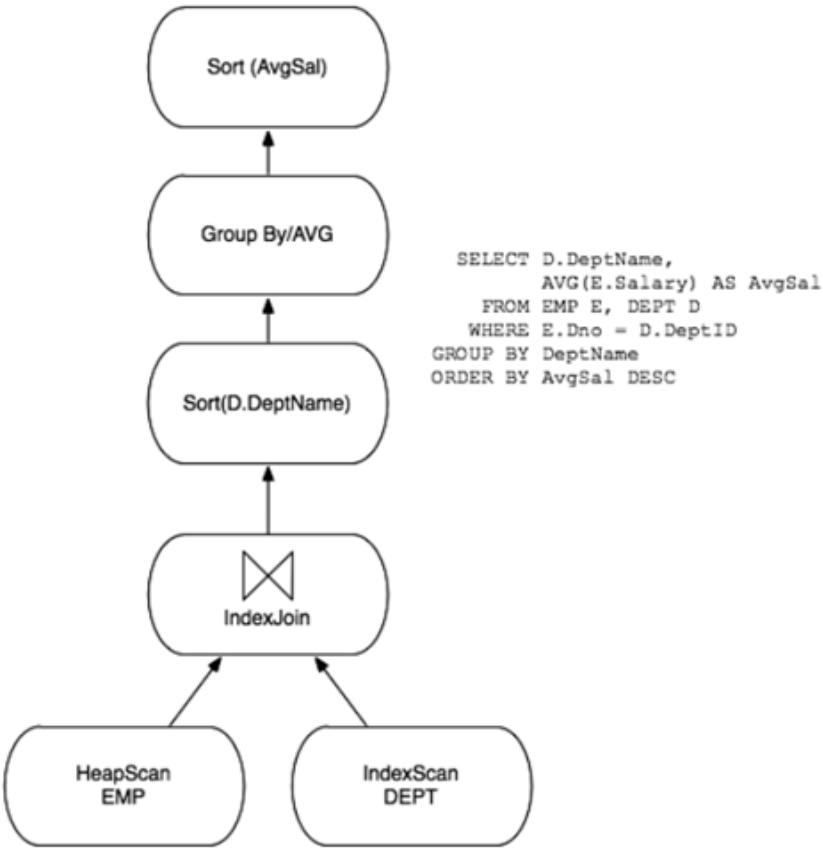

图 4-1 一个查询计划

当系统的代码库被制作成商业的数据库管理系统（SQL/DS）的时候，开发团队做出的第一个改变就是用一个解释器来替换机器代码执行器。

为了能够实现跨平台的可移植性，现在每一个主要的 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。在某些系统中，查询计划是一个非常轻量级的对象，未必不可能是一个关系代数表达式（是由访问方法名称和连接算法等来表示的）。其他系统使用更低层次的“操作码”语言，相比于关系代数表达式，这种语言在思想上更像 Java 字节码。为了简单起见，在之后的讨论中，我们专注于类代数查询表示。

虽然 Selinger 的论文被广泛地认为是查询优化领域的“圣经”，但是，毕竟也只是初步的研究。所有系统在许多不同的角度上都显著地拓展了这篇论文的工作，主要的拓展有：

- **计划空间**：System R 优化器通过只专注于“left-deep”查询计划（一个连接操作的右手边的输入必须是一个基表），以及“推迟笛卡尔积”（保证在数据流中，求笛卡尔积的操作是出现在所有的连接之后），来限制查询计划空间。在当今的商业系统中，在一些情况下，“浓密的树”（具有嵌套式的右手边输入）和尽早计算笛卡

尔积是很有用的。因此，在大多数系统中，这两个选择在某些情况下是被考虑使用的。

- **选择性估算（selectivity estimation）**：Selinger 论文中的选择性估算技术是基于简单的表和索引基数，如果按照当今系统的标准，这种选择性估算技术是很初级的。如今，大多数的系统利用直方图和其他简易的统计数据来分析和概括属性值的分布。因为这涉及到访问每一个列值，所以代价比较昂贵。因此，一些系统使用抽样技术来得到大概的属性值分布，而无需付出完全扫描每一列值的代价。适合于基本表连接操作的选择性估算，可以通过把连接列上的直方图连接起来来实现。为了摆脱单列直方图的束缚，一些包含列之间的依赖性等问题的、更复杂方案[16,69]最近被提出来了。这些创新已经出现在商业产品上，但还有待取得更多的进步。这些机制被缓慢采用的一个原因是，在许多工业测试基准（benchmark）中存在一个长期的缺陷：如 TPC-D 和 TPC-H 基准数据发生器所生成的每一例，在数值的分布上具有统计独立性，因此，就不提倡采用那些用来处理“真实数据分布”的技术。这个测试基准的缺陷已经在 TPC-DS 测试基准[70]中得到解决。尽管被采用的步伐很慢，但是，改进的选择性估算所带来的好处还是被广泛认可的。Ioannidis 和 Christtodoulakis 注意到，在优化过程的早期的选择性估算错误，会被查询计划树成倍放大，以至于最后会得到一个非常糟糕的估算结果[45]。
- **搜索算法**：在一些商业系统中，特别是 Microsoft 和 Tandem，放弃了 Selinger 动态规划优化方法，转而支持一种基于级联式技术[25]的、目标导向的、自顶而下的搜索方案。在一些情况下，自顶向下搜索可以降低一个优化器需要考虑的计划的数量[82]，但是，同时产生了负面影响，即增加了优化器内存消耗。如果实践成功与否是衡量一个技术的质量的标准，那么，选择自顶向下搜索，还是选择动态规划优化方法，都是无关紧要的，因为，二者都被证实可以在先进的优化器上运行得很好，同时，不幸的是，它们的运行时间和内存需求，都与一次查询中所涉及表的数量成指数关系。针对涉及到太多表的查询，一些系统会采用启发式查询机制。尽管随机查询优化启发式方法的研究文献[5,18,44,84]是很有趣的，但是，在商业系统中使用的启发式方法是被申请了专利的，它们显然不同于那些研究文献中的随机查询优化启发式方法。一个教育实践就是去检查开源 MySQL 引擎的查询优化器，它在最后的检查是完全启发式的，而且大部分依赖于利用索引和键/外键约束。这让人想起了早期的、臭名昭著的 Oracle 版本。在一些系统中，在 FROM 子句中涉及到太多

表的查询只能在以下情况下才能运行，即用户必须明确指示优化器如何选择一个方案(通过嵌入在 SQL 中所谓的优化器“暗示”来实现)。

- **并行**：如今每一个主流的商业数据库管理系统都对并行处理有一定的支持。大多数也支持“查询内”并行：通过多处理器的使用来加速一个查询。查询优化器需要参与决定，如何在多个 CPU 之间以及在多个独立的计算机之间（在无共享或共享磁盘的情况下）调度操作符（被并行化的操作符）。Hong 和 Stonebraker[42]选择避开并行优化复杂性问题，使用两个阶段来实现：首先，传统的单系统优化器被调用来选择最好的单系统方案；然后，这个计划在多个处理器和机器上被调度。尽管不明确这些结果会在多大程度上影响当前的实践，但是，关于第二个优化阶段的研究文献[19,21]已经被发表。一些商业系统实现了上述描述的两阶段方法。其他一些系统则努力对集群网络拓扑结构以及集群机器之间的数据分布进行建模，然后，在一个阶段内产生一个最好的方案。虽然单阶段方法可以被证明在某些情况下产生更好的方案，但是，有一点仍然不明确，那就是，使用单阶段的方法产生的查询方案质量改进，相对于由此带来的额外的优化器复杂性而言，是否是值得的。因此，许多当前的系统，在实现时仍然倾向于采用两阶段方法。当前，这个领域看起来更像是艺术而非科学。Oracle OPS(如今称为 RAC)共享磁盘集群使用两阶段优化器。IBM DB2 并行版本（如今称为 DB2 数据库分区特色）一开始使用两阶段优化器来实现，但是，如今已经演变为单阶段来实现。
- **自动调优（tuning）**：各种各样正在进行的工业研究努力，尝试着改善 DBMS 来自动执行调优决策。这些技术中的一部分是基于收集查询负载，然后通过各种“what-if”分析来使用优化器来确定查询计划的代价，例如，如果其他索引已经存在时该怎么办。正如 Chaudhuri 和 Narasayya 描述的那样[12]，一个优化器需要在一定程度上被调整来高效地支持本次查询活动。Markl 等人的学习优化器（LEO）的工作[57]，也是这种类型。

4.3.1 一个查询编译和重新编译的标注

SQL 支持“预处理”查询的能力：将查询传递到解析器、重写器和优化器，把生成的查询执行计划存储起来，并且在后续的“执行”语句中使用。这可能同样适用于用程序变量代替查询常量的动态查询（例如，来自网页表单）。唯一的麻烦就是在选择性估算期间，由

对于那些由表单提供的变量，优化器将会采用“典型”的值。当无代表性的“典型”值被选择时，就会导致很差的的查询执行计划。查询预处理对于表单驱动以及在一些可预测数据的查询是很有用的。当应用程序被编写，查询会被预处理，当应用程序上线时，用户就没有必要经历解析、重写和优化，不会产生这些方面的代价。

尽管在编写一个程序时进行查询预处理是可以提高性能的，但是，这是一个非常有局限性的应用程序模型。许多应用程序开发者，以及像 Ruby on Rails 这样的工具箱，在程序执行期间动态构建 SQL 语句，因此，无法提供预编译。因为这种情况非常普遍，DBMSs 就在查询计划的缓存中存储这些动态查询执行计划。如果相同的（或者非常相似的）语句在随后被提交，那么就使用缓存里的版本。这项技术近似于预编译静态 SQL 的性能，不会受到应用程序模型的限制，因此，被大量使用。

随着数据库的演化，通常需要重新优化预编译计划。至少当一个索引被删除的时候，任何使用这个索引的执行计划都必须从存储计划缓存中删除，以保证在下次调用的时候，选择一个新的执行计划。

其他关于重新优化计划的设计决定是更加微妙的，显示了数据库厂商之间的设计理念区别。一些数据库厂商（例如 IBM）非常努力地在多次调用中获得可预测的性能，而不是每次调用查询执行计划都获得最优的性能。因此，在类似删去索引的情况下，供应商不会重新优化一个查询执行计划，除非该计划不再执行。其他供应商（例如 Microsoft）非常努力地使它们的系统实现自我调优，而且将会更加积极地重新优化执行计划。例如，如果一个表的基数发生了显著的变化，在 SQL 服务器中就会触发重新编译过程，因为，这种变化可能会影响到索引和连接顺序的最优使用。一个自我调优的系统是难以预测的，但是在动态环境中更加有效。

这种设计理念的区别主要来自于这些产品的历史客户群体基础的不同。IBM 传统上注重于有熟练技能的数据库管理员这样的高端用户和应用程序开发者。在这样的高预算的 IT 商店，从数据库中获得可预测性能是非常重要的。经过几个月的调优数据库设计和设置，数据库管理员不希望优化器不可预测地改变这些设置。相比之下，微软公司则战略性地进入了数据库的低端市场，他们的客户群体趋向于拥有较低的 IT 预算和专业知识，希望 DBMS 尽可能地“自我调优”。

随着时间的推移，这些公司的商业战略和客户基础逐渐融合，因此，这些公司就发生了直接的竞争，这使得他们的方法开始走向融合。微软拥有大规模的企业用户，这些用户想

要完全地控制和查询计划稳定性。IBM 有一些客户没有数据库管理员，所以，需要完全的自动控制。

4.4 查询执行器

查询执行器操作一个完全具体的查询计划。这通常是一个把很多操作连接在一起的数据流图，而这些操作封装了基本表（base table）的访问和各种查询执行算法。在一些系统中，这个数据流图已经被优化器编译成低级的操作码。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，查询执行器接收到一个数据流图的表示，然后递归调用基于图布局的操作程序。我们专注于后面这种情况，因为，操作码的方法在本质上就是把我们在这里描述的逻辑编译为一个程序。

大多数当代的查询执行器使用迭代器模型，该模型曾经使用在最早期的关系型系统中。迭代器大多数仅仅被描述为面向对象的形式。图 4-2 展示了一个迭代器简化的定义。每一个迭代器都规定了它的输入，即数据流图的边。

```
class iterator {
  iterator &inputs[];
  void init();
  tuple get_next();
  void close();
}
```

图 4-2 一个迭代器超类的伪代码定义

在一个查询计划中的所有操作——数据流图中的节点——都会被实现为迭代器类的一个子类。在一个典型的系统中，子类的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、重复消除和分组聚类。迭代器模型的一个重要特性就是，迭代器任何一个子类可以作为其他子类的输入来使用。因此，在数据流图，每一个迭代器的逻辑是独立于它的子类和父母类的，不需要使用专门的代码对这些迭代器进行组合。

Graefe 在他的查询执行综述论文[24]中提供了更多关于迭代器的细节。建议感兴趣的读者去研究开源 PostgreSQL 代码库。PostgreSQL 使用了适度复杂的迭代器实现，而这些迭代器适用于大多数标准查询执行算法。

4.4.1 迭代器讨论

迭代器的一个重要性能就是，它们连接了数据流和控制流。get_next()调用是一个标准过程调用，该调用通过调用堆栈给调用者返回一个元组引用。因此，当一个控制返回的时候，一个元组就返回给数据流图的父类。这意味着，只需要一个单一的 DBMS 线程来执行一个完整的查询图，迭代器之间的队列和速率匹配是不需要的。这就使关系型查询执行器可以顺利地实现功能，并且易于调试，同时，也与其他环境中的数据流架构形成了鲜明的对比，例如，网络就需要在并发的生产者和消费者之间设计各种各样的队列和反馈协议。

单线程迭代器架构对于单系统（非集群）查询执行同样是高效的。在大多数数据库应用程序中，判别是否高效的性能指标是查询完成的时间，但是，也可能采用其他优化目标。例如，最大化 DBMS 的吞吐量是另一个合理可行的目标。在很多交互式应用程序的数据库系统中，会把到达第一行的时间作为性能指标。在一个单一处理器的环境中，当所有资源被完全利用的时候，把一个给定查询计划的完成时间作为优化目标是可以实现的。在一个迭代器模型中，因为其中一个迭代器总是处于活跃状态，所以，资源利用率是最大化的。

正如我们之前提到的，大多数当代的 DBMS 支持并行查询执行。幸运的是，基本上可以不用对迭代器模型和查询执行架构做任何修改就可以实现这种支持。并行性和网络通信可以被封装在 Graefe 描述的特殊交换迭代器中[23]。这些也实现了网络式的数据“推”操作，并且实现方式对于 DBMS 迭代器而言是不可见的，这些迭代器保留了“拉”式的 get_next() API。一些系统也在它们的查询执行模型中显示地确定“推”操作。

4.4.2 数据在哪里？

为了讨论方便，我们对迭代器的讨论已经避开了任何关于正在使用的数据的内存分配问题。我们既没有详细说明元组是如何在内存中储存的，也没有说明数据是如何在迭代器之间传递的。实际上，每一个迭代器都预分配了固定数量的元组描述符（tuple descriptor），每个输入对应一个元组描述符，输出对应一个描述符。一个元组描述符通常是一个关于“列引用”的数组，这个引用数组的每一个列引用，包括对内存中某个元组的引用和那个元组的列偏移。基本的迭代器超类的逻辑程序，永远不会动态地分配内存。这就提出了一个问题，被引用的真实元组到底被存储在内存中什么地方了。

对于这个问题，有两种可能的答案。第一种是，那个元组驻留在缓冲池的页面中，我们称这些为缓冲池元组。如果一个迭代器构造了一个引用缓冲池元组的元组描述符，那么它就必须增加那个元组所在的缓冲池页面的引脚数（pin count），即在那个页面上的元组的活跃（active）引用数量。当元组描述符被销毁的时候，迭代器就减少引脚数。第二种可能就是，一个迭代器实现可能为在内存堆中的元组分配空间。我们称这个为内存元组。一个迭代器可能通过复制缓冲池的列来构造一个内存元组，或者通过求查询中的表达式来构造一个内存元组（例如，像“EMP.sal*0.1”这样的表达式）。

一个通用的方法就是总是将缓冲池中的数据复制到内存元组中。这个设计使用内存元组作为唯一的查询时使用的元组结构，从而简化了执行器代码。这个设计也可以避免一些缺陷，这些缺陷来自于在缓冲池中分开执行 pin 和 unpin 调用的时间间隔很长（比如由于包含多行代码）。一种常见的错误是完全忘记对页面执行 unpin 操作（产生缓冲区泄漏）。不幸的是，正如第 4.2 节描述的那样，专门内存元组会成为一个主要性能问题，因为，在一个高性能系统中，内存副本通常是一个瓶颈。

另一方面，在某些方面构造内存元组是很有意义的。只要一个迭代器直接引用一个缓冲池元组，这个缓冲池元组所在的页面必须在缓冲池中固定不动。这就消耗一个页的缓冲池内存，而且束缚了缓冲替换策略作用的发挥。如果一个元组将长期被引用，那么将该元组从缓冲池复制出来是很有好处的。

该讨论的要点就是，同时支持缓冲池元组和内存元组的元组描述符是最有效方法。

4.4.3 数据修改语句

到目前为止，我们只讨论查询，即只读 SQL 语句。另一种数据操作语言是为修改数据而存在的，包括 INSERT、DELETE 和 UPDATE 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，即把单个访问方法作为源头，一个数据修改操作符作为数据流管道的尾部。

然而，在一些情况下，这些计划同时查询和修改同一个数据。这种针对同一个表的读和写混合操作（可能多次），需要格外小心。一个简单的例子就是声名狼藉的“万圣节问题”，因为，它是在由 System R 小组在 10 月 31 日发现的。万圣节问题是由像“给每个工资低于 20K 美元的人增加工资 10%”这样的语句的特殊执行策略而产生的。这个查询的朴素执行计划，会把 Emp.salary 域上的索引扫描迭代器以管道的方式输入给一个更新迭代器（如图

4-3 的左边所示，图 4-3 中左边的计划是容易发生万圣节问题的，右边计划是安全的，因为在执行任何更新之前，它会首先确定所有需要进行更新的元组）。这个管道提供了很好的 I/O 局部性，因为它只在元组被从 B+-树中获取到以后才对元组进行修改。然而，这个管道也可能导致索引扫描在修改结束之后，“重修发现”一个之前修改过的元组在 B+树上向右移动，导致每一个员工多次加薪。在我们的例子中，所有低薪员工将会收到重复的加薪，直到他们的收入超过 20K 美元。这不是这个语句的真实意图。

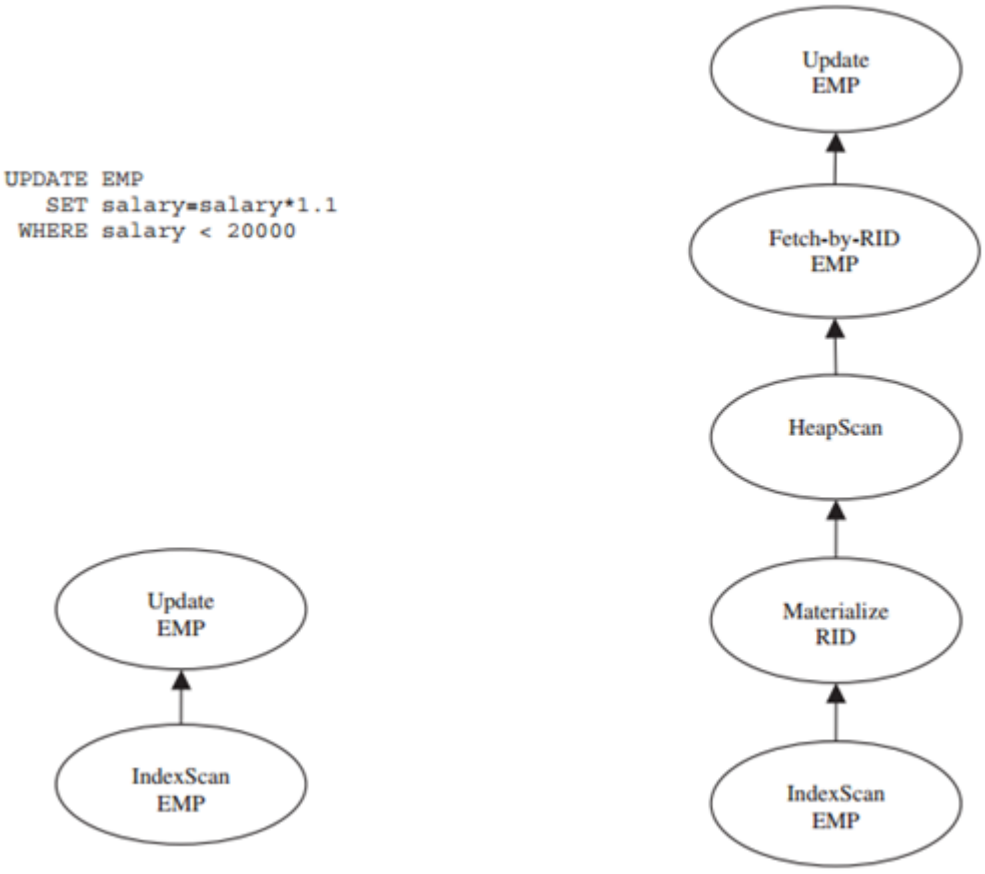

图 4-3 通过 IndexScan 更新一个表的两种查询计划

SQL 语义禁止这种行为：一个单一的 SQL 语句是不被允许“看到”自己的更新的。一定要小心保证遵守这个“可视”规则。一个简单、安全的实现，可以使得查询优化器所选择的计划能够避免对更新列进行索引。这在某些情况下是很低效的。另一种技术是使用批量的“先读后写”方案。这就需要在数据流图中（图 4.3 的右边），在索引扫描和数据修改操作符之间插入记录 ID 物化（materialization）操作符和数据抓取操作符。物化操作符接收所有需要修改的元组的 ID，并将它们存储在临时文件中。接着，物化操作符扫描临时文件，通过 RID（记录 ID）获取每一个物理元组 ID，并将结果元组提供给数据修改操作符。如果优化器选择一个索引，在大多数情况下，这会意味着只有一小部分元组发生改变。因此，这种

技术的明显低效率是可以被接受的，因为，临时表很可能完全保留在缓冲池中。管道化的更新机制也是可能的，但是，需要来自存储引擎的多版本支持[74]。

4.5 访问方法

访问方法是用来对系统支持的基于磁盘的数据结构的访问进行管理的，通常包括无序的文件（“堆”）和各种各样的索引。所有的商业系统都实现了堆和 B+树索引。Oracle 和 PostgreSQL 同时都支持“平等查找”（equality search）的哈希索引。一些系统开始引入对类似 R-树这样的多维索引的初步支持。PostgreSQL 支持一种叫做 Generalized Search Tree[39]的可扩展性索引，当前使用它来实现多维数据的 R-树索引以及针对文本数据的 RD-树索引[40]。第 8 个版本的 IBM UDB 引入了多维分簇索引[66]，通过多个维度上的区间来访问数据。以读操作为主的数据仓库负载通常采用专用的、索引的位图变种，正如我们在第 4.6 节描述的那样。

访问方法提供的基本 API 是一种迭代器 API。Int()例程会被扩展，从而可以接受一种列操作符常量形式的“搜索参数”（或者在 System R 术语中被称为 SARG）。一个 NULL SARG 被看成一个扫描表中所有元组的请求。当再也没有满足搜索参数的元组时，在访问方法层调用 get_text()将返回 NULL 值。

这里有两个原因将 SARG 传递给访问方法层。第一个原因是很清晰的，像 B+-树这样的索引访问方法需要 SARG 来高效地运行。第二个原因是更细微的性能问题，但是，适用于堆扫描和索引扫描。假定 SARG 是由调用访问方法层的例程来检查的。那么，每次访问方法从 get_next()返回时，它必须：（a）返回一个指向“驻留在缓冲池的某个帧中的元组”的句柄，并且固定住那个帧中的页面，从而避免发生页的替换；或者（b）复制一份元组。如果调用者发现 SARG 不满足，它就要负责：（a）减少那个页面上的 pin 数量，或者（b）删除复制的元组。然后必须重新调用 get_next()，以尝试在页面中的下一个元组。这个逻辑程序在函数“调用/返回”对（pair）上，消耗了较多数量的 CPU 周期，将会导致在缓冲池中“固定”（pin）了不必要的页面（产生了缓冲池帧的不必要的竞争），或者不必要地创建和销毁元组副本——当流式地通过数百万的元组的时候，这会是一个巨大的 CPU 开销。需要注意的是，一个典型的堆扫描将会访问给定页面中所有的元组，导致每个页面中这种交互的多次迭代。相比之下，如果所有这种逻辑在访问方法层就完成了，那么重复的“调用/返回对”和“销毁/不销毁或者复制/删除”，可以通过下面的方式来加以避免，即在测试 SARG

时一次测试一页，并且对于满足该 SARG 的元组，只从 get_next()调用返回。SARG 在存储引擎和关系型引擎中保着清晰的边界，同时保持着极好的性能。因此，许多系统支持非常丰富的 SARG 而且广泛使用它们。在主题层面上，这是关于在集合中的多个项目之间分摊工作的标准 DBMS 智慧的很好的一个实例，但是，在这种情况下，它是为了获得更好的 CPU 性能，而不是磁盘性能。

所有的 DBMS 需要某个方法来指向基本表的行，这样索引条目就可以恰当地引用行。在许多 DBMS 中，这是通过使用直接的行 ID（RID）来实现的，这些行 ID 是基本表的行在磁盘中的物理地址。这样做的好处就是速度快，但是，也带来了负面影响，即造成基本表的行在移动时开销较大，因为，所有指向这行的二级索引都需要更新。查找和更新这些行的代价都是很高的。当一个更新改变了行的大小，导致当前页面空间无法容纳更新后的行时，行就需要移动。当一个 B+-树分裂时，许多行也是需要移动的。DB2 使用一个前向指针来避免第一个问题。这就需要第二次 I/O 来找到移动的页面，但是，避免了更新二级索引。DB2 仅仅通过“不支持 B+-树作为基本表元组的主要存储”来避免第二个问题。Microsoft SQL Server 和 Oracle 支持 B+-树最为主要存储，就必须能够高效解决行的移动问题。采取的方法就是避免在二级索引中使用一个物理行地址，而是使用行主键（如果表没有一个主键，就提供额外的系统位来保证行的唯一性）而不是物理 RID。当使用二级索引来访问基本表行的时候，这会牺牲一些性能，但是，却可以避免一些由行的移动而导致的问题。Oracle 通过使用一个物理指针和主键来避免在某些情况下这种方法带来的性能损失。如果行没有移动，那么使用物理指针就可以很快地找到该行。但是，如果行移动了，那么将使用缓慢主键技术。Oracle 通过允许行跨越页面来避免在堆文件中移动行。因此，当一行被更新为更长的值，以至于不再适合存储在原来页面的时候，Oracle 并没有迫使去移动该行，而是采取如下措施：存储在原来页面的部分仍继续存储在原来页面，剩下的部分可以存储在下一个页面。

与所有其他的迭代器相比，访问方法和并发性、事务恢复逻辑等都有很深的交互，正如在第四章描述的那样。

4.6 数据仓库

数据仓库是服务于决策支持的、包含大量历史数据的数据库，OLTP 系统中的数据更新会被周期性地加载到数据仓库中。数据仓库已经发展到需要专业查询处理的支持，在下一章

内容中，我们将会总结一些它们需要具备的关键特性。数据仓库作为相关话题放在这里讨论，主要有两个原因：

- 数据仓库是 DBMS 技术的一项非常重要的应用。有些人声称数据仓库占了所有数据库管理活动的 1/3[26,63]。
- 本章内容到目前为止所讨论的传统的查询优化和执行引擎，在数据仓库上无法获得较好地性能。因此，为了获得更好的性能，有必要对它们进行扩展和修改。

关系型数据库管理系统最早构建于 20 世纪 70 年代到 20 世纪 80 年代之间，用来满足业务数据处理应用需求，因为，这是那个时代最主要的需求。在 20 世纪 90 年代早期，出现了数据仓库和业务分析的市场，并且从那时起增长很快。

在 20 世纪 90 年代，联机事务处理（OLTP）已经取代了批量业务数据处理成为数据库使用的主导形式。此外，大部分 OLTP 系统有很多计算机操作员提交事务，要么来自和终端客户的电话交流，要么把纸上的数据进行输入。自动柜员机已经广泛流行，允许客户不用操作员干预就能直接进行交互。此类交易的响应时间对于生产力来说是至关重要的。这种响应时间需求只会变得更加迫切，并且在多样化的今天，互联网正在以终端客户的自助服务来取代操作员。同时，企业在零售领域想要捕捉所有的历史销售业务，并存储它们一到两年。买家可以通过这样的历史销售数据来找出什么是热销的，什么不是热销的。这些信息可以用来影响采购模式。同样，这些数据可以用来决定哪些商品用来促销，哪些商品寄回厂家。在使用数据仓库时，大家的共识就是，在销售领域采用历史数据仓库，可以进行更好的库存管理、货架管理和商店布局管理，由此带来的收益，会超过数据仓库本身的投入。

很明显，一个数据仓库应部署在独立于 OLTP 系统之外的硬件里。使用该方法，漫长的（通常是不可预测的）商务智能查询，就不会破坏 OLTP 的响应时间。同时，数据的本质是非常不同的，数据仓库处理历史数据，OLTP 处理“现在”的数据。最后，往往会发现历史数据模式经常和当前数据模式不匹配，这就需要进行数据模式的转换。由于这些原因，人们构建了工作流系统，从 OLTP 系统不断“刮取”（scrape）数据，装载到数据仓库中。这样的系统被命名为“提取、转换和加载”（ETL）系统。受欢迎的 ETL 产品包括来自 IBM 的 Data Stage 和来自 Informatica 的 PowerCenter。在过去的十年里，ETL 供应商已经将产品扩展到数据清洗工具、重复数据删除工具和其它以质量为中心的服务。接下来，我们必须讨论几个在数据仓库环境中必须解决的问题。

4.6.1位图索引

B+树为记录的快速插入、删除和更新进行了优化。相比之下，一个数据仓库执行原始的负载，然后数据就几个月或几年都是静态的。而且，数据仓库经常包含只有少数量值的列，例如，考虑一下存储用户的性别，性别只有两种值（男或女），而且在位图上，性别可以用一个位记录来描述。相比之下，一个 B+树的每一个记录将需要（值，记录指针）对，而且通常情况下，每一个记录需要消耗 40 位。

位图在合取谓词过滤器上也很有优势，例如，Customer.sex=”F”,和 Customer.state=”California”。在这种情况下，结果集是由相交的位图决定的。有许多更加精致的位图算法技巧，可以被使用来提高常见的分析查询的性能。对于位图处理的讨论，有兴趣的读者可以参考文献[65]。

在当前的产品中，位图索引为 Oracle 中的 B+树索引提供了很好的补充，而 DB2 提供了一个更为有限的版本。Sybase IQ 大量使用位图索引。当然，位图的缺点就是它们昂贵的更新代价，所以，它们只限制在仓库环境中使用，因为，数据仓库一般是只读的，数据写入后就不会发生更新。

4.6.2 快速下载

通常情况下，数据库在半夜加载白天的数据交易。零售场所只在白天开放是一个显而易见的策略。实行夜间批量加载的第二个原因是，为了避免用户交易过程中出现更新。考虑一下，一个业务分析师希望制定一些即席查询的方案，也许会查询飓风对顾客购买模式的影响。这个查询的结果可能要求后续的查询，如调查大风暴中的购买模式。这两个查询的结果应该是兼容的，也就是说，应该基于相同的数据集来计算得到答案。如果数据同时被加载，就会导致包含最近历史的查询出现问题。

因此，数据仓库的快速批量加载至关重要。虽然人们可以用 SQL 插入语句来编写数据仓库加载程序，但是，从来没有人在实践中使用过这种战术。相反，人们一般利用批量加载机（loader）把大量记录存储到数据仓库中，它没有 SQL 层的开销，并充分利用了面向 B+-树的特殊批量加载方法的优点。批量加载比 SQL 插入的速度大约快一个数量级，并且，所有主要的数据库厂商都提供了一个高性能的批量加载机。

随着电子商务和 24 小时营业的全球化，这种批量加载的战术意义不大。但是，在“实时”仓库的道路上还存在两个问题。首先，插入操作（无论是批量加载还是事务中的插入操作），必须设置写锁，如第 6.3 节中讨论的那样。这就会和查询所获得的读锁发生冲突，可能导致仓库“冻结”。其次，设置如上所述的查询之间提供一致的答案也是有问题的。

避免就地更新，提供历史查询，可以避免这两个问题。如果保持更新之前和之后的值，那么，适当的时间戳可以提供在最近的时间的查询。运行相同历史时间的集合查询，将会提供一致的答案。此外，同样的历史查询，可以在没有设置只读锁的时候运行。如在第 5.2.1 节中讨论的那样，一些数据厂商（特别是 Oracle 公司）提供了多版本（MVCC）隔离级别。由于实时仓库变得越来越流行，其它厂商大概会效仿。

4.6.3 物化视图

数据仓库通常是巨大的，并且，连接多个大表的查询似乎有种要运行到“永远”的倾向。为了提高热门查询的性能，大多数厂商提供物化视图。不像本章前面讨论过的纯粹的逻辑视图，物化视图是可以查询的实际表，而不是建立在真正基本数据表基础之上的逻辑视图表达式。一个物化视图的查询，可以避免在运行时执行视图表达式中的连接操作。物化视图必须保持最新，因为更新一直在进行。

物化视图的应用主要有三个方面：（a）选择要物化的视图；（b）保持视图的更新；（c）考虑在即席查询中使用物化视图。其中，（a）是自动数据库调优的一个先进方面，我们在第 4.3 节中已经提到；（c）在各种产品上实施的程度不同，这个问题在理论上具有挑战性，甚至对于简单的单块查询[51]而言也是如此，对于包含聚集和子查询的通用 SQL 而言更是如此；至于（b），大多数厂商提供多个刷新技术，比如，每次当物化视图所基于的源头表的数据发生更新时，物化视图都要执行更新，再比如，定期丢弃一个物化视图然后重新创建新的物化视图。这些策略提供了一个在运行时的开销和物化视图数据一致性之间的权衡。

4.6.4 OLAP 和 Ad-hoc 查询支持

一些仓库工作负载有可预见的查询。例如，在每个月的月底，一份总结报告的运行可能提供在零售链每个销售地区的部门的销售总额。很明显地，可以通过适当地构建物化视图来支持可预测的查询。更普遍的是，由于大多数业务分析查询需要聚合（aggregation）操作，

因此，人们可以为每家商店的销售总额按照部门来进行聚合计算，为计算结果生成物化视图。然后，如果上面的区域查询被选定，我们就可以对每个区域每个商店进行“上卷”操作来满足这个查询的要求。这种聚合结果往往被称为数据立方体（data cube），是物化视图中一个有趣的类。20 世纪 90 年代早期的产品，如 Essbase，提供了定制工具，采用优先立方体格式存储数据，并且提供了基于数据立方体的用户界面来浏览数据。这个功能后来被称为联机分析处理（OLAP）。随着时间的推移，数据立方体支持已经被添加到全功能的关系数据库系统中，通常被称为关系型 OLAP（ROLAP）。很多提供 ROLAP 的 DBMS 已经发展到可以在内部实现一些早期的 OLAP 类型的存储机制，这种机制有时会简称为 HOLAP（混合 OLAP）机制。

显然，数据立方体为可预见的有限的查询类提供了很高的性能。然而，它们对于即时查询而言一般没有什么帮助。

4.6.5 雪花模式查询的优化

许多数据仓库遵循一个特定的模式设计方法。一般而言，它们会存储一系列事实表，在零售业环境中，事实表通常就是简单的记录，例如“客户 X 在时间 T 从商店 Z 购买了商品 Y”。一个核心的事实表记录了每个事实的信息，例如购买价格、账户、销售汇率信息等等。同时，在事实表中还有维度集合的外键。维度包括客户、产品、商店、时间等等。这种形式的方案经常被称为“星型方案”，因为，它在中心有一个事实表，这个事实表周围是很多维表，每一个维表与事实表之间都存在 1-N 的主外键关联。画成实体关系图的形式，这样的模式就是星型的。

许多维度是自然地具有层级结构的。例如，如果商店可以被聚合到地区中，那么商店的“维表”就增加一个外键指向地区维表。对于涉及到时间（月/日/年）和管理级别等属性而言，通常也存在类似的层级结构。在这些情形中，会出现一个多层次星型模式，或者称为雪花模式。

基本上所有数据仓库查询需要对雪花模式的一个或多个维度，在这些维表的一些属性上进行过滤，然后将过滤结果与中央的事实表进行连接操作，接着根据事实表或维表的一些属性进行分组，最后计算一个 SQL 聚合结果。

随着时间的推移，供应商在他们的优化器中拥有特殊的查询类，因为它是如此的流行，为如此长时间执行的命令选择一个好的计划是很重要的。

4.6.6 数据仓库：结论

正如我们看到的那样，数据仓库需要提供不同于 OLTP 环境的能力。除了 B+-树，数据仓库还需要位图索引。数据仓库需要将重点放在基于雪花模式的聚合查询，而不是一个通用的优化器。数据仓库需要物化的视图而不是常规的视图。数据仓库需要快速的批量加载而不是快速的事务更新等等。在文献[11]中有内容更加丰富的、关于数据仓库操作实践的阐述。

主要的关系型数据库厂商从面向 OLTP 的架构开始，随着时间的推移，供应商已经增加了面向数据仓库的功能。另外，已经有各种各样的小型供应商在这个领域提供了 DBMS 解决方案。这包括 Teradata 和 Netezza，他们提供了在他们的 DBMS 上运行的、无共享的专有硬件。而且，这个领域还有 Greenplum（PostSQL 的并行性）、DATAllegro 和 EnterpriseDB，以上这些都运行在传统的硬件上。

最后，一些人（包括本文的其中一个作者）认为，相对于传统的行存储引擎（即存储单元是表的行）而言，列存储在数据仓库空间中拥有巨大的优势。当表是“很宽的”（很多的列），而且仅仅趋向于访问一些列时，单独存储每一列是尤其高效的。列存储也可以带来简单而有效的磁盘压缩，因为，列中所有的数据来自于同一种类型，具有更高的数据压缩率。列存储面临的挑战是，表格中每一行的位置需要与所有列保持一致，或者需要额外的机制来把这些列重新连接起来得到一个完整的记录。这对于 OLTP 是一个大问题，但是，对于像数据仓库或者系统日志库这样的追加数据库而言，就不是一个主要问题。像 Sybase、Vertica、Sand、Vhayu 和 KX 等数据库厂商，都提供了列存储。该架构讨论的更多细节可以在[36,89,90]中找到。

4.7 数据库扩展性

传统上，人们都认为关系型数据库存储的数据类型是有限的，主要集中在企业和行政机构记录保存中使用的“事实和数字”。然而，如今，关系型数据库支持主流的编程语言描述的多种数据类型。这是通过用多种方式使核心的关系型数据库管理系统进行扩展来实现的。在这节，我们简要地总结一下广泛使用的扩展方式，重点阐述一些在实现这种扩展性中产生的架构问题。这些特性不同程度地出现在如今大多数商业数据库管理系统中，也包括开源的 PostgreSQL 数据库关系系统。

4.7.1 抽象数据类型

原则上，关系模型对可放置在模式列上的标量数据类型是一无所知的。但是，最初的数据库系统只支持一组静态的字母数字列类型，而且，这种限制与关系模型本身是相关的。一个关系型数据管理系统在运行时可以被扩展支持新的抽象数据类型，正如在早期 IngresADT 中阐述的，在后来的 Postgres 系统中表现更为明显。

为了实现这点，DBMS 类型系统——以及解析器——必须从系统目录中驱动，系统目录保存了系统已知的类型列表，以及指向操作类型的“方法”（代码）的指针。在这种方法中，DBMS 不解释数据类型，它仅仅在表达式计算时恰当地调用它们的方法；因此叫作“抽象数据类型”。作为一个典型的例子，DBMS 可以注册一个 2 维空间“矩形”的类型，以及像矩形相交或合并的操作方法。这也意味着系统必须为用户自定义的代码提供一个运行时引擎，而且安全地执行那个代码，没有一丝导致数据库服务器崩溃或毁坏数据的风险。如今所有主要的数据库管理系统，都允许用户采用现代 SQL 的“存储过程”子语言来定义函数。除了 MySQL，大多数数据库管理系统至少支持一些其他语言，通常是 C 和 Java。在 Windows 平台上，Microsoft SQL Server 和 IBM DB2 支持代码编译到 Microsoft .Net Common Language Runtime，它可以用多种语言进行编写，较为普遍的是 Visual Basic,C++和 C#。PostgreSQL 本身就支持 C、Perl、Python 和 Tcl，而且允许在运行时向系统添加对新语言的支持——流行的第三方 Ruby 插件和开源 R 统计包。

为了使抽象数据类型在 DBMS 上高效地运行，查询优化器必须在选择和连接谓词上解释“昂贵”的用户自定义代码，在一些情况下推迟“选择”操作直到“连接”操作完成[13,37]。为了使抽象数据类型更加高效，在它们上定义索引是很有用的。至少，B+-树需要扩展到可以为抽象数据类型上的表达式进行索引，而不只是对列进行索引（有时候称为“函数索引”），必须对优化器进行扩展，从而可以使用这些建立在表达式上的索引。对于不是线性命令（<，>，=）的谓词，B+-树是不够的，系统需要支持一种可扩展的索引机制。文献中记载的两种方法是：原始 Postgres 可扩展性访问方法接口[88]和 GiST[39]。

4.7.2 结构化类型和 XML

ADTs（抽象数据类型）被设计成完全兼容关系模型——它们不以任何方式改变基本的关系代数，只改变属性值表达式。然而，在过去几年，出现了许多积极改变数据库来支持关

系型结构类型的建议：例如，嵌套集合类型，像数组、集合、树和嵌套元组以及/或者关系。如今，与这些提议最相关的就是通过 XPath 和 XQuery 等语言提供对 XML 的支持。大约有三种方法来处理 XML 这种结构化数据类型。第一种方法就是建立一个自定义数据库，该数据库可以操作结构化类型的数据。历史上，在传统关系型 DBMS 内部容纳这种结构化数据类型的方法，已经取代了上面所说的这种方法，而且这种趋势在 XML 出现之后继续延续着。第二种方法就是将复杂的数据类型看成一个 ADT。例如，可以定义一个包含 XML 类型的列的关系表，每一行存储一个 XML 文档。这就意味着，搜索 XML 的表达式——XPath 匹配树模式——需要以一种对于查询优化器而言很难懂的方式来执行。对于 DBMS 的第三种方法就是将嵌套结构规范化为一个关系集合，用外键来连接子类和它们的父类。这种技术有时候称为“分割”XML 文档，在一个关系框架内部将数据的所有结构暴露给 DBMS，但是，增加了存储开销，而且在查询时需要“连接”操作重新连接数据。大多数 DBMS 供应商提供 ADT 和存储分割选项，而且允许数据库设计者在它们之间做出选择。对于 XML 这种情形，在“分割”方案中提供去除同一级别的 XML 嵌套元素之间的的顺序信息的功能是很普遍的，它可以通过允许重排序和其他关系型优化来提高查询性能。

一个相关的问题是扩展关系型模型来处理嵌套表和元组以及数组。例如，这在 Oracle 安装上是广泛使用的。设计的取舍与处理 XML 的权衡在很多方面是相似的。

4.7.3 全文检索

传统上，关系型数据库在处理丰富的文本数据以及伴随使用到的关键词搜索方面是出了名的差的。原理上，在数据库上建立自由文本模型是一个简单的存储文档问题，可以定义一个“倒排文件”关系，关系中的每个元组采用（word,documentID,position）这种形式，并且在 word 列上建立一个 B+-树索引。这大致是在任何文本搜索引擎中所发生的事情，此外，还会建模一些单词的语言规范以及利用一些额外元组属性来辅助排序搜索结果。

除了这个模型，大多数文本索引引擎针对这种模式实现了许多性能优化，这在典型的 DBMS 上是没有实现的。这些优化措施包括对模式进行非规范化，例如，（word，list<documentID，position>），从而使得每个 word 在出现列表中只出现一次。这就可以支持列表的增量压缩，对于文档中的单词具备扭曲（Zipfian）分布的情形而言，增量压缩是很重要的。而且，文本数据库趋向于以数据库仓库类型来被使用，规避了任何 DBMS 事务逻

辑。一般而言，大家普遍认为，在一个 DBMS 中简单采用文本搜索引擎，要比采用定制的文本索引引擎大致慢一个数量级。

然而，如今大多数 DBMS 要么包含文本索引的子系统，要么附带一个单独的引擎来做这项工作。文本索引功能通常可以适用于全文文档和元组中较短的文本属性的搜索。在大多数情况下，全文索引一般采用异步更新，而不是采用事务性的更新维护。在这一点上，PostgreSQL 的做法有点不同寻常，它为事务更新情形提供了全文索引。在一些系统中，全文索引是存储在 DBMS 之外的，因此需要独立的工具来备份和还原。在关系型数据库中处理全文搜索，一个重要挑战就是在关系型语义（结果是一个无序的、完整的集合）与排序文档搜索（结果是有序的、不完整的）之间建立语义桥梁。例如，当两个关系都有一个关键字搜索谓词，如何对一个查询的、来自两个关系的结果进行排序，是不明确的。这在当前实践中仍然是临时确定的。给定一个查询输出语义，关系型查询优化器的另一个挑战就是分析文本索引的选择性和代价，以及判断一个查询的合适的代价模型（这个查询的结果集在用户界面上是有序和分页的，可能不能完全地检索）。根据各种报道，许多主流的 DBMS 正在积极地解决上述的最后一个挑战。

4.7.4 额外的可扩展性问题

除了数据库可扩展性的三个驱动使用场景，我们这里讨论引擎中的两个核心组件，它们经常被扩展后服务于各种各样的用途。

有许多关于可扩展查询优化器的建议，包括支持 IBM DB2 优化器[54,68]的设计和支持 Tandem 与 Microsoft 优化器[25]的设计。所有这些方案提供规则驱动的子系统，该子系统产生和修改查询计划，以及允许新的优化规则独立地注册。当新的功能被添加到查询执行器或者当关于查询重写或计划优化方面的新想法产生时，这些技术对于更容易地扩展优化器而言是很有用的。这些通用的架构在上述描述的许多具体的可扩展类型的功能方面是很重要的。

另一个在早期系统中出现的可扩展性的形式是数据库把远程数据源“打包”（wrap）到自身模式（schema）内的能力，从而使得这些远程数据似乎就是本地的表，而且可以在查询处理过程中访问它们。在这方面的一个挑战就是需要优化器去处理那些“不支持扫描、但是会响应把值赋给变量的请求”的数据源；这就需要对那些“可以把索引 SARG 匹配到查询谓词”的查询优化器进行扩展[33]。执行器的另一个挑战就是高效地处理远程数据源，该数

据源在产生输出方面可能缓慢，也可能很快；使查询执行器执行异步的磁盘 I/O 是一个很大的设计挑战，它会使得访问时间可变性[22,92]增加了一个以上数量级。

4.8 标准实践

基本上所有的关系数据库查询引擎的底层架构，看起来都与 System R 的原型很相似[3]。过去这些年，查询处理的研究和发展都把重点放在了在这个架构范围内的创新，从而可以增加更多种类的查询和模式。不同系统之间的设计区别，主要体现在优化器搜索策略（自顶向下，还是自底向上）和查询执行器控制流模型（尤其是对无共享和共享磁盘的并行性，应该采用迭代器和交换操作模型，还是采用异步的生产者/消费者模式)。在一个更细粒度层面上，在优化器、执行器、访问方面等方面，都有大量的不同机制组合方案，从而可以使得在不同的工作负载下都能获得好的性能，这些负载类型包括 OLTP、数据仓库和 OLAP。这个商业产品的秘方决定了它们在特殊情况下可以表现得有多好；几乎所有的商业系统都认为自己可以在各种不同类型的工作负载下做得很好，但是，实际上可能会在某种特殊的工作负载上面看起来比较慢。

在开放源码领域，PostgreSQL 有一个具有合理复杂性的查询处理器，它拥有一个传统的基于代价的优化器、一个执行算法集合和大量的还没有在商业产品中发现的扩展功能。MySQL 的查询处理器更简单，采用了基于索引的嵌套循环连接。MySQL 的查询优化器着重于对查询进行分析，从而确保常用操作是轻量级和高效的，尤其是主外键连接，“外连接到连接”的重写，以及只要求结果集的前几行的查询。仔细阅读 MySQL 手册和查询处理的代码、并把它与更多传统的设计进行比较，是有借鉴意义的（要知道，MySQL 在实践方面具有很高的市场普及率），同时也要了解 MySQL 执行哪些任务比较出色。

4.9 讨论和附加材料

因为查询优化和执行形成了比较清晰地模块，在过去这些年，这个领域里已经研究出了大量的算法、技术、技巧，并且关系型查询处理的研究到今天还在持续。值得高兴的是，大部分已被用于实践的想法（很多还没有），都可以在已经发表的研究文献中找到。如果想从事查询优化研究，一个比较好的研究起始点是阅读 Chaudhuri 的简短综述文献[10]。对于查询处理研究而言，Graefe 提供了一个很全面的综述论文[24]。

除了传统的查询处理，近几年有大量工作开始把丰富的统计方法融合到处理大型数据集中。一个很自然的扩展就是使用抽样或汇总统计为聚合查询提供数值估算[20]，可能是以一种持续改进的、在线的方式[38]。然而，尽管有相当成熟的研究成果，但是，市场在采纳这些研究成果方面仍然步伐缓慢。Oracle 和 DB2 都提供了简单的基表采样技术，但是，没有提供统计上包括多个表的、鲁棒性较好的查询估算。他们没有把重点放在这些特性上，相反，大部分数据库厂商把主要精力放在丰富他们的 OLAP 功能上面，这就约束了那些可以被很快回答、并且可以为用户提供百分之百准确性的查询的数量。

另一个重要但更根本的扩展，就是把数据挖掘技术引入 DBMS 领域。流行的技术包括统计聚类、分类、回归和关联规则[14]，除了文献中研究的这些技术的独立实现以外，把这些技术和关系查询的体系架构进行整合，也是一个很大的挑战[77]。

最后，值得注意的是，最近，数据并行使得整个计算机研究领域变得异常活跃，比较有代表性的是，Google 的 Map-Reduce、Microsoft 的 Dryad、被 Yahoo 所支持的开源 Hadoop 代码。这些系统很像无共享、并行的关系查询执行器，使用应用程序开发者自己编写的自定义的查询操作符。它们包含了简单而明智的设计方法，用来管理参与节点的失败，在大规模集群中，这种节点失败是一种常见的情况。或许这一领域最有趣的方面就是，被创造性地应用于很多种类的计算领域数据密集问题，包括文本和图像处理、统计方法等。我们将会很有趣地看到，是否其他来自于数据库引擎的方法会被这些框架的使用者借鉴，例如，在 Yahoo，早期的工作曾经对 Hadoop 进行扩展，使它拥有声明性查询和优化器。在这些框架上的创新，反过来，也可以被融合到数据库引擎中。

第 5 章 存储管理

在今天的商业应用中，主要有两种基本类型的 DBMS（数据库管理系统）存储管理器：（1）DBMS 直接与底层的面向磁盘的块模式设备驱动程序进行交互（通常称为原始模式访问）；（2）DBMS 使用标准的 OS 文件系统设施。这个决定会在空间和时间上同时影响 DBMS 控制存储的能力。下面我们先分别考虑这两个维度（空间和时间），然后继续深入地讨论存储层次的使用。

5.1 空间控制

从磁盘中读取和写入数据时，顺序读写带宽要比随机读写带宽快 10 到 100 倍，并且这个差距还在增加。磁盘密度每 18 个月翻一番,带宽增长速度与磁盘密度呈平方根的关系(与旋转速度呈线性关系)。但是，磁盘机械臂移动速度的增长率则较低，约为每年 7% [67]。因此，对于 DBMS 存储管理器来说，如何把数据块放置在磁盘上就显得尤其重要，从而使得需要访问大量数据的查询可以顺序地访问磁盘。因为 DBMS 能够比底层操作系统更理解它的工作负载访问模式，所以，完全由 DBMS 设计师来确定如何把数据库块放置到磁盘上，是有意义的。

对于 DBMS 而言，控制数据空间局部性的最好的方式，就是将数据直接存储到“原始”磁盘设备中，完全绕过文件系统。这种做法是可行的，因为，原始设备地址通常对应于存储位置的物理临近性。为了获得峰值性能，大部分商业数据库系统能够很好地提供这项功能。这种技术虽然有效，但还是有一些缺点。第一，它需要数据库管理员将整个磁盘分区都分配给数据库管理系统，这就使得这些磁盘空间无法提供给那些需要使用文件系统接口的工具。第二，“原始磁盘”的访问接口往往是与特定操作系统相关的，这使得 DBMS 的可移植性变得更差。这是一个难点，不过近来大多数商业 DBMS 厂商都已经解决这一问题。最后，随着存储行业的发展，RAID、存储区域网络和逻辑卷（volume）管理器，已经非常普及。现在我们所处的情况是，在许多应用场景中，“虚拟”磁盘设备已经成为规范——“原始”设备接口实际上已经被很多硬件和软件拦截掉，它们会在一个或多个物理磁盘之间重新定位数据。因此，随着时代的发展，由 DBMS 显式地控制磁盘所带来的收益已经越来越少。我们

将在第 7.3 节进一步讨论这个问题。

原始磁盘访问的一种替代方式是，由 DBMS 在操作系统的文件管理系统中创建一个非常大的文件，然后采用数据在文件中的地址偏移量来定位数据。这个文件在本质上可以视为磁盘页面的一个线性阵列。这可以避免原始设备访问的一些缺点，并且仍然能够提供相当好的性能。在大多数流行的文件系统中，如果你分配一个非常大的文件到一个空磁盘上，文件中的地址将会和存储位置的物理临近性非常吻合。因此，这是一个原始磁盘访问的很好的近似方法，而不需要直接访问原始设备接口。大多数虚拟化存储系统在设计时，都会把文件中临近的地址放置在临近的物理位置中。因此，随着时间的推移，使用大文件而不是原始磁盘的相对控制代价，已经变得越来越不明显了。使用文件系统接口对时间控制有其他后果，我们将在接下来的内容中讨论。

我们最近在一个中等规模的系统中，使用一个主流的商业 DBMS，对直接原始磁盘访问和大型文件访问这两者进行了比较，我们发现，当运行 TPC-C 测试基准[91]时，只有 6%的性能降低，而对于较少包含密集 I /O 的工作负载而言，几乎没有负面影响。DB2 的报告显示，当使用直接 I / O（DIO）和它的变体比如并发 I / O(CIO)时，文件系统开销可以低至 1%。因此，数据库管理系统厂商通常不再推荐原始数据存储，而且很少用户会使用这种配置。一些主流商业系统还在支持这种特性，主要是用于测试基准（benchmark）。

一些商业 DBMS 还允许自定义数据库页面大小，使它能够适合预期的工作负载。IBM DB2 和 Oracle 等都支持这个功能。其他的商业系统，比如 Microsoft SQL Server，不支持多种页面大小，因为这会增加管理的复杂性。如果支持页面大小可调，那么可供选择的页面尺寸应该是一个文件系统（如果使用原始 I / O，这里就是原始设备）所使用的页面尺寸的倍数。在“5 分钟法则”的论文中，讨论了如何选择合适的页面大小，这个法则后来又被更新为“30 分钟法则”[27]。如果使用文件系统而不是原始设备访问，就需要特定的接口来写入那些与文件系统页面大小不同的页面，例如 POSIX 的 mmap / msync 调用，就提供了这种支持。

5.2 时间控制:缓冲

除了控制数据在磁盘上的放置位置，一个 DBMS 还必须控制数据什么时候被物理地写入到磁盘中。正如我们将在后面内容讨论的那样，DBMS 包含了关键的逻辑程序，它可以判断什么时候把数据写入磁盘。大多数操作系统的文件系统还提供内置的 I / O 缓冲机制，

来决定何时读取和何时写入文件块。如果 DBMS 在执行写操作时使用标准的文件系统接口，操作系统缓冲机制将会打乱 DBMS 逻辑程序的意图，因为，操作系统缓冲机制会悄悄地推迟 DBMS 写操作或者重新排序写操作。这可能会给 DBMS 带来大问题。

第一类大问题是，数据库的 ACID 事务承诺的正确性：如果不能对磁盘写操作的时间和顺序进行显式的控制，那么，在发生软件或硬件失败后，DBMS 不能保证原子恢复。正如我们将在第 5.3 节讨论的那样，写前日志（write ahead logging）要求一个写操作在写入到数据库相应的设备之前，必须先写入日志设备，并且在提交日志记录没有写入日志设备之前，不能给用户返回提交请求。

OS 缓冲问题的第二个方面是性能问题，但是没有对正确性提出要求。现代操作系统的文件系统，通常提供了内置功能支持“预读取（read-ahead）”（随机的读取）和“后写入（write-behind）”(延迟，批处理)。这些对于 DBMS 访问模式而言，通常都不适合。文件系统的逻辑程序，会根据文件中的物理字节偏移量连续性来做出提前读取的决定。DBMS 级别的 I/O 设施，可以根据未来的读请求来做出逻辑预测 I/O 决定，这些未来的读请求在 SQL 查询处理层面是可以知道的，但是在文件系统层面上则很难做到。例如，当扫描不一定被连续存储的 B+-树叶子节点时（行被存储在一个 B +-树的叶子节点中），就可能会发生逻辑的、DBMS 层面的预读取请求。逻辑预读取是很容易在 DBMS 逻辑中实现的，只要让 DBMS 在它产生实际需求之前就发出 I / O 请求。查询执行计划包含了关于数据访问算法的相关信息，并且拥有关于这个查询的未来访问模式的完整信息。类似地，DBMS 可能想对何时刷新日志尾部做出自己的决定，它会综合考虑锁冲突和 I / O 吞吐量。DBMS 可以获得这些详细的未来访问模式信息，而操作系统的文件系统则无法获得这些信息。

最后的性能问题是“双缓冲”和内存拷贝的昂贵的 CPU 开销。鉴于 DBMS 必须妥善管理自己的缓冲从而保证正确性，因此，任何由操作系统提供的额外缓冲都是多余的。这种冗余会引起两个代价。第一，它浪费了系统内存，因为它显著地减少了系统中可用于有用工作的内存。第二，它浪费了时间和处理资源，因为它会导致额外的复制步骤：执行读取操作时，数据会首先从磁盘复制到操作系统的缓冲区，然后再复制到 DBMS 缓冲池。而写数据时，这两者都刚好相反，即先把数据写入到 DBMS 缓冲池，然后再复制到操作系统的缓冲区，最后写入磁盘。

在内存中复制数据将会是一个严重的性能瓶颈。复制导致延迟，消耗 CPU 周期，并且使 CPU 的数据内存溢出。这对于一个从没有操作或者实现过数据库系统的人来说，常常是一个令人惊讶的事，他们往往以为，相对于磁盘 I / O 而言，内存操作是“免费”的。但是

在实践中，在一个经过调优后的事务处理 DBMS 中，吞吐量通常不受 I /O 的限制。这在高端 DBMS 配置中，可以通过购买足够的磁盘和内存来实现，从而使得这些重复的页面请求都被缓冲池吸收，磁盘 I/O 会在不同磁盘机械臂之间进行共享，从而使得磁盘 I/O 的速率可以满足数据系统中所有处理器对数据的需求。一旦实现了这种“系统平衡”，I / O 延迟将不再是系统吞吐量的主要瓶颈，剩下的内存瓶颈则会成为系统吞吐量的主要限制因素。内存拷贝正成为计算机体系结构的主要瓶颈：这是由于在性能演变过程中，每秒每美元的原始 CPU 周期（会遵循摩尔定律）和 RAM 的存取速度（明显是在追随着摩尔定律）之间的“鸿沟”越来越大[67]。

操作系统缓冲的问题已经在数据库研究文献中被广泛熟知[86]，被业界所熟知也有一段时间了。大多数现代操作系统现在提供钩子（比如，POSIX mmap 套件调用或者特定于平台的 DIO 和 CIO API 集合），从而使得程序（比如数据库服务器）可以规避对文件的双缓冲。这可以确保当写操作被请求时直接被写入到磁盘，避免了双缓冲，而且 DBMS 可以控制页面置换策略。

5.3 缓冲管理

为了提供对数据库页面的有效访问，每个数据库管理系统会在自己的内存空间中实现一个大型共享缓冲池。在早期，缓冲池是被静态设置为管理员设定的值，但是，现在大多数商业 DBMS 会根据系统需要和可用资源来动态调整缓冲池大小。缓冲池会被组织成一个帧数组，其中，每一帧是内存中的一段区域，帧的大小是数据库磁盘块的大小。块从磁盘直接复制到缓冲池中，不会发生格式的变化，在内存中也是以这种原生的格式进行修改操作，然后，写回磁盘。这种不需要转换的方法，避免了在向磁盘写入数据和从磁盘读取数据过程中的 CPU 瓶颈。也许更为重要的是，固定大小的帧，避免了通用技术导致的外部碎片和压缩方面的内存管理复杂性。

和缓冲池中的帧数组相关联的是一个哈希表，它会对以下内容进行哈希映射：（1）把内存中当前的页面编号映射到它们在帧表中的位置；（2）页面在备份磁盘存储中的位置；（3）关于该页面的一些元数据。

元数据包括一个“脏”标记位，用来表示页面在从磁盘中读取出来以后是否已经发生改变；元数据还包括页面替换策略所需要的任何信息，当缓冲区满的时候，页面替换策略会选择被驱逐出缓冲区的页面。大多数系统还包括一个引脚计数器（pin count），来标记该页面

是没有资格参与页面替换算法的。当引脚数是非零时，页面被“钉”（pin）在内存中，不会被强行写入到磁盘或丢失。这允许 DBMS 的工作线程在操作一个页面之前，通过增加引脚数来把页面“钉”在缓冲池中，操作结束后，再减少引脚计数器的值。这样做的目的是，在任何时间点，只让少量的缓冲池空间被“钉”住。有些系统还提供了管理选项，允许把一个表“钉”在内存中。对于较小的、频繁访问的表而言，它可以改进访问时间。但是，被“钉住”的页面，也减少了可以供正常缓冲池活动的页面数量，并且，当被“钉住”的页面百分比增加时，会对性能造成负面影响。

早期的关系系统的许多研究都是集中在设计页面置换政策，因为，DBMS 中数据访问模式的多样性使得简单的技术变得无效。例如，某些数据库操作往往需要全表扫描，当被扫描的表远远大于缓冲池时，这些操作往往会清除缓冲区中所有常用的数据。对于这样的访问模式而言，如果把近期的引用情况作为判断未来引用情况的依据，那将会是非常糟糕的；因此，操作系统中经常使用的页面替换机制（比如 LRU 和 CLOCK），在许多数据库访问模式下面都表现得非常糟糕[86]，这是众所周知的。研究人员提出了各种替代方案，比如，一些方案试图利用查询执行计划信息来调整替换策略[15]。今天，大多数系统使用简单的增强 LRU 方案来进行全表扫描。一个出现在研究文献中并且已经运用到商业系统中的方案是 LRU-2[64]。商业系统中使用的另一种机制是，根据页面类型来确定替换策略，例如，B+-树的根节点的替换策略，可能和堆文件中的页面替换策略不同。这不禁让人想起了 Reiter’s Domain Separation 方案[15,75]。

最近的硬件发展趋势（包括 64 位寻址和内存价格的下降），使得使用非常大的缓冲池变得经济可行。这开启了利用大内存提高效率的新机遇。但是反过来，一个大的、非常活跃的缓冲池，在重启恢复速度和高效的检查点等方面也带来更多的挑战，当然问题还不仅仅是这些。这些主题将在第六章进一步讨论。

5.4 标准实践

在过去的十年里，商业文件系统已经进化到可以很好地支持数据库存储系统。在标准的使用模型中，系统管理员在每个磁盘上创建一个文件系统，或者在 DBMS 的逻辑卷上创建一个文件系统。然后，DBMS 为每一个文件系统分配一个单一的大文件，然后通过低层次的接口（如 mmap 套件）来控制数据的放置。DBMS 基本上把每个磁盘或逻辑卷当作一个(几乎)连续的数据库页面的线性数组。在这个配置中，现代文件系统为 DBMS 提供了合理的空

间和时间控制，并且这种存储模型可在所有数据库系统中实现。在大多数数据库系统中，对原始磁盘的支持仍然是一个常见的高性能选项，然而，它的应用范围迅速变窄，现在一般只用于性能基准测试中。

5.5 讨论以及附加材料

数据库存储子系统是一个很成熟的技术，但是，在最近几年，数据库存储方面又出现了许多新的考虑因素，它有可能在许多方面改变数据管理技术。

一个关键的技术变化是闪存的出现，它已经是一种经济可行的、支持随机访问和持久存储的技术[28]。自从数据库系统研究的早期阶段，就一直在讨论，新的存储技术取代磁盘会引起 DBMS 设计的巨大变化。闪存具有技术上和经济上的可行性，并且具有广泛的市场支持，它的性价比介于磁盘和 RAM 之间。在近 30 年里，闪存是第一个成功的、新的持久性存储介质，因此，它的特性将可能显著影响未来 DBMS 的设计。

最近走向前台的另一个传统话题是数据库数据的压缩。早期文献的主题主要集中在磁盘压缩，从而最小化读取时的磁盘延迟，并且最大化数据库缓冲池的容量。由于处理器性能已经改进很多，而内存延迟却没有保持同步改进, 这使得数据压缩（即使在计算时）变得越来越重要，从而最大化处理器缓存中的数据延迟。但是，这需要压缩数据表示形式是适合用来进行数据处理的，同时还要能够适用于对压缩数据进行操作的查询处理。另一个关系型数据库压缩的难题是，数据库是可排序的元组集合，而大部分关于压缩的研究工作集中在字节流，而不考虑重新排序。最近关于这一主题的研究表明，在不久的将来可以很好地实现对数据库的压缩[73]。

最后，在传统的关系数据库市场之外，大家对大规模但稀疏的数据的存储技术的兴趣，正在日益增强，这些数据在逻辑上有成千上万个列，但对于某个给定的行，其中的大部分列都是空的。这些场景通常采用某种属性-值对或三元组来表示。实例包括谷歌的 BigTable[9]，Microsoft’s Active Directory and Exchange 产品所使用的标签列，以及为语义网提出的资源描述框架（RDF）。这些方法的共同点是，在使用存储系统方面，都采用数据表列的方式而不是行的方式来组织磁盘存储。在最近的一系列数据库研究成果[36,89,90]中，关于面向列的存储思想又复苏起来，并且有了一些详细的研究。

第 6 章 事务：并发控制和恢复

数据库系统常被认为是一种庞大的复杂的软件，很难被分割为多个可重用部分。但在实际中，数据库开发以及维护的团队确实是将数据库系统按照明文规定的接口将数据库系统拆分为多个部分来设计的。这一点在关系查询处理器和事务存储引擎之间的接口上表现尤为明显。在大多数商用系统中，这些组件由不同的团队开发并且彼此之间有良好定义的接口。

数据库系统真正庞大而复杂的部分是事务存储管理器，该部分由四个彼此紧密关联的组件组成：

- 用于并发控制的锁管理器；
- 用于事务恢复的日志管理器；
- 数据库 I/O 缓冲池；
- 用于组织磁盘数据的访问方法。

很多文章分析过数据库系统中的事务管理算法以及协议的诸多细节。读者如果想了解更多内容，可以阅读基本的本科教材[72]、关于 ARIES 日志协议的期刊文章[59]以及至少一篇介绍事务索引并发控制和日志的论文[46、58]。对以上内容已经有一定了解的读者，我们推荐阅读 Gray 和 Reuter 在事务处理方面的教材[30]。不过想要成为真正的专家，在阅读的同时还要在实现方面下一些功夫。我们将不在具体算法和协议上花费太多笔墨，主要介绍的是各个组件的角色和作用。我们将重点介绍在教材中被忽略的基础架构问题，并且着重分析各个组件间的内部依赖关系，这些依赖关系在实现简单的协议时导致了许多复杂的细节问题。

6.1 ACID

许多人对 Haerder 和 Reuter 提出的“ACID 事务”很熟悉[34]。ACID 表示事务的原子性、一致性、独立性和持久性。但是，它们并不是为保证事务正确而被正式定义并经得起数学论证的术语。因此，我们不必刻意去分析它们的不同以及彼此之间的关系。尽管 ACID 并不是正式术语，但是，它对于我们组织讨论事务系统是很有用的，而且，它非常重要，因此，

我们在这里再回顾一下：

- **原子性**：是对事务“全部做或者全部不做”的保证——即事务的所有行为或者全部提交或者全部不做；
- **一致性**：是应用层面的一个保证；SQL 语句的完整性约束通常就是用于在数据库系统中保证一致性的。给定一个由约束条件集提供的一致性定义，只有当一个事务在完成时可以使得整个数据库仍保持一致性状态的时候，这个事务才能被提交；
- **隔离性**：使得对于应用开发者而言，两个并发的事务看不到彼此正在进行的更新操作。这样，应用开发者就不必因担心其他事务的脏数据而采取防御性编程，可以当做只有自己在访问数据库；
- **持久性**：是指保证一个成功提交的事务对数据库的更新是永久的，即便之后发生软件或者硬件故障，除非另一个提交的事务将它重写。

简单地说，现代数据库管理系统通过锁协议来实现隔离性。持久性通过日志和恢复技术来实现。隔离性和原子性由锁（使瞬时数据库状态不可见）和日志（确保可见数据的正确性）来保证。一致性由查询处理器运行时的检查来管理：如果一个事务违反一个 SQL 一致性约束，这个事务就会被终止，并且返回错误信息。

6.2 可串行化的简单回顾

在开始讨论事务之前，我们首先简单回顾一下数据库并发控制的目的什么。在接下来的一节中，我们将讲述两个最重要的用来在多用户事务存储管理器中实现并发概念的模块：（1）锁；（2）锁存器。

可串行化是并发事务正确性的一个很好的书面定义。它意味着，多个事务相互交错的一组并发执行，必须与该组事务的一个串行执行结果相对应——即执行结果与没有并发的结果相同。可串行化是描述一组事务行为约束的一种说法。从单个事务的角度来看，隔离性有相同的意思。如果一个事务在执行时看不到其它并发的行为，我们就说该事务是隔离执行的，这就是 ACID 中的 I。

可串行化是由 DBMS 并发控制模型来实现的。这里有三种并发控制的技术，它们在教材和早期的文件中都有详细描述，这里我们再简短回顾一下：

- **严格的两段锁（2PL）**：事务在读任何数据之前需要一个共享锁，而在写之前需要一个排他锁。一个事务所拥有的锁，会一直保持到事务结束时才自动释放。当一个

事务等待锁时，会中断然后移动到等待队列。

- **多版本并发控制（MVCC）**：事务不使用锁控制机制，我们为过去某一时间点的数据库状态保存一个一致的副本，即便在某一固定时间点之后数据库状态发生了改变，我们也可以读到数据库的一个过去的状态。
- **乐观并发控制（OCC）**：该方法允许多个事务在无阻塞的情况下读或者更新一个数据项。事务会保存自身的读写历史，在一个事务提交前，必须通过检查其读写历史来判断是否发生了隔离性冲突；若发生，则发生冲突的其中一个事务必须回滚。

大多数商业关系数据库管理系统通过 2PL 机制来实现可串行化。锁管理器是提供 2PL 机制的代码模块。

为减少锁请求和锁冲突，一些数据库管理系统支持 MVCC 或者 OCC，将它们作为 2PL 的一个补充。在 MVCC 模型中，读锁不是必要的，但是，这种实现方式的代价是，无法提供完全可串行化。为了避免读操作后的写操作被阻塞，在之前的数据项版本已保存或者可以很快获得的情况下，写操作可以被允许执行。同时，正在执行的读操作事务则继续使用早期的数据项值，这样就仿佛被读取的数据已经被加锁了一样。在 MVCC 的商业实现方面，这种稳定的读操作数值，或者被设定为读事务开始时的数据值，或者被设定为该事务最近的一个 SQL 语句开始时的数据值。

由于 OCC 避免了锁等待，因此，当事务之间真正发生冲突时将会产生很高的惩罚代价。在处理事务间的冲突方面，OCC 和 2PL 比较类似，除了它会将 2PL 中的锁等待操作转换为了事务回滚。在针对部分特殊的冲突时，OCC 表现得很好，它避免了过度保守的等待时间。但是，当冲突频繁发生时，过多的回滚和重试会严重降低性能，这时，OCC 是一个较差的选择[2]。

6.3 锁(locking)和锁存器(latching)

数据库锁是系统中常用的名字，它可以表示数据库系统管理的物理项（如磁盘页）或者逻辑项（如元组、文件、卷）。我们可以看到，任何一个名字都可以有一个与之相关联的锁，即便这个名字只是一个抽象的概念。锁管理器提供一个可以为这些名字申请锁或者检查锁的地方。每个锁都关联到一个事务，同时，每一个事务都有一个自己的事务 ID。锁有不同的类型，并且存在一个与这些类型关联的锁类型兼容表。在大多数系统中，锁的逻辑以 Gray 关于锁粒度的论文[29]中所介绍的锁类型为基础。这篇论文也介绍了商业系统中是如何实现

分层锁的。分层锁机制允许一个针对整个表的锁，同时，在该表中还可以高效并准确地使用行级别粒度的锁。

锁管理器支持两个基本的函数：lock(lockname,transactionID,mode)和remove_transaction(transaction-ID)。需要注意的是，由于遵循严格 2PL 协议，因此，我们不能单独地释放某个资源锁——函数 remove_transaction()会释放与一个事务相关的所有资源锁。然而，如同我们在第 5.2.1 节中所说的那样，SQL 标准允许较低水平的事务隔离性，因此，我们也需要函数 unlock(lockname,transactionID)。还有一个函数 lock_upgrade(lockname,transactionID,newmode)，事务通过调用它来将锁升级到一个较高的锁类型（如从共享锁升级到排他锁），这就不必释放锁并重新申请一个锁了。此外，一些系统也支持 conditional_lock(lockname,transactionID,mode)函数。这个函数在调用后立即返回并指出是否成功得获得了锁。如果获取锁失败，则被调用的 DBMS 线程将不再排队等待锁。关于索引并发所需的条件锁的使用，在文献[60]中有详细介绍。

锁管理器提供了两个数据结构来支持这些函数。一个全局的锁表用来记录锁的名字以及它们的相关信息。锁表是以锁名字为键值的动态哈希表。每个锁都有一个类型标识位来表示锁的类型，还有一个等待队列来记录锁请求信息（transactionID，mode）。此外，锁管理器还有一个以 transactionID 为键值的事务表，表中的每个事务有两项信息：（1）一个指向该事务的 DBMS 线程的指针，这使得事务因请求锁而等待时可以进行线程调度；（2）指向锁表中该事务所有锁请求的指针的链表，这会有利于移除某个事务的所有锁（如当事务提交或者中止）。

从内部实现来看，锁管理器利用死锁检测器这个 DBMS 线程来周期性地检查锁表中是否存在等待环（环中每一个 DBMS 工作者都在等待下一个，因而形成环）。根据对死锁的检测，死锁检测器会中止死锁中的某一个事务。具体是哪个事务被中止，则由启发式算法决定，文献[76]中有关于这些算法的详细描述。在无共享或共享磁盘系统中，一个分布式死锁检测[61]或一个更原始的超时死锁检测是必需的。关于锁管理器具体的实现细节，读者可以去阅读 Gray 和 Reuter 的文档[30]。

轻量级的锁存器(latch)作为数据库锁的一种补充，也被应用于处理互斥问题。锁存器相对于锁而言，更类似于一种监视器或者信号量；它们被用来实现数据库内部数据结构的互斥存取。举例来说，缓冲池页表中每一个页（帧）都有一个对应的锁存器，这样可以保证在任何时刻只有一个 DBMS 线程替换给定的页。锁存器也被用于锁的实现中，在内部数据结构可能被并发地改变时，可以依靠锁存器来暂时地确保这些内部数据结构的稳定性。

锁存器在以下几个方面不同于锁：

- 锁被保存在锁表中并通过哈希表进行定位；锁存器位于内存中更靠近它要保护的资源，它通过地址直接访问。
- 在严格两段锁实现中，锁受严格两段锁协议支配。在一个事务中，锁存器可以基于特定应用的内部逻辑来使用或放弃。
- 锁的获得完全是由数据访问情况决定的，因此，获得锁的顺序及锁的持续时间，在很大程度上是由应用和查询优化器来决定的。锁存器则由 DBMS 内部特定的代码来获取，因此，DBMS 内部代码决定了锁存器的获取和释放。
- 锁可以允许产生死锁，死锁可以被检测并通过事务重启来解决。锁存器的死锁则是不允许发生的，锁存器死锁的发生，意味着 DBMS 代码出现了一个 bug。
- 锁存器的实现主要通过原子的硬件指令操作，但是，在极少数情况下，当不存在原子硬件指令操作时，也会通过系统内核的互斥来实现。
- 锁存器的调用只需要几十个 CPU 时钟周期，而锁请求则需要几百个 CPU 时钟周期。
- 锁管理器跟踪一个事务所有的锁，并且在事务抛出异常时自动释放它们。但是，DBMS 在应对锁存器时，必须谨慎地追踪它们，包括把手动清除也作为一种异常处理方式。
- 锁存器不被追踪，所以，当任务失败时，不会被自动释放。

锁存器 API 支持的函数有 latch(object,mode)，unlatch(object)和 conditional_latch(object,mode)。在大多数 DBMS 系统中，锁存器的可选类型只有共享或者排他。锁存器维持一个类型，同时，也维护一个等待获得该锁存器的 DBMS 线程队列。锁存器的添加和释放，正如我们预期的那样工作。Conditional_latch()调用类似于前文说过的 conditional_lock()，它也被用于索引并发[60]。

6.3.1 事务隔离性级别

在早期的事务概念中，为了提高并发性，人们尝试了很多相对于可串行化而言更弱的定义。这些尝试最大的困难是如何给出具有鲁棒性的定义。这方面最具影响力的成就来自于 Gray 早期关于“一致性程度”的研究[29]。这项工作试图说明一致性程度的定义，并通过锁的形式来实现它。受该工作的影响，ANSI SQL 标准定义了四个“隔离等级”：

- **未提交读**：一个事务可以读任何已提交或未提交的数据。这可以通过“读操作不需

要请求任何锁”来实现。

- **已提交读**：一个事务可以读任何已提交的数据。对于同一个对象的重复读可能导致读到不同版本的数据。实现方式是，读数据前必须首先获得一个读操作锁，一旦数据读取之后该锁被立即释放。
- **可重复读**：一个事务只能读取一个已提交数据的一个版本；一旦该事务读取了一个对象，那么，它将只能读取该对象的同一个版本。实现方式是，事务在请求读数据之前必须获得一个锁，并且保持该锁直到事务结束。
- **可串行化**：保证完全的可串行化。

乍一看可重复读似乎保证了完全的可串行化，但是，其实并不是这样。在早期的 R 系统[3]中发生了一个“幽灵问题”。在幽灵问题中，一个事务使用同样的谓词多次访问了同一个关系，但是，最近的访问却得到了最初访问时没有发现的新的“幽灵元组”。原因在于，元组级的两段锁并不能阻止往表中插入元组。表级别的两段锁可以防止幽灵问题，但是，当事务通过索引访问表中的几个元组时，表级别的两段锁是被限制的。

商业系统通过锁机制的并发控制来实现上述四种隔离性级别。但不幸的是，正如 Bernson 等人[6]指出的那样，不论是早期的 Gray 的工作，还是 ANSI 标准，都没能提供真正意义上的定义。它们都只是基于一个假设：锁方案用于实现并发控制，而不使用乐观[47]或是多版本[74]并发方案。感兴趣的读者可以去阅读 Berenson 关于讨论 SQL 标准技术规范问题的论文，也可以阅读 Adya 等人[1]的关于提出一种新的、清晰的解决方案的研究工作。

除了 ANSI SQL 隔离级别以外，很多开发商提供了其它一些可应用于特殊情况的隔离性级别：

- **游标稳定**：这个等级是为了解决已提交读的更新丢失问题。假设有两个事务 T1 和 T2。T1 以“已提交读”模式运行，读取数据项 X（假设是银行账户值），记录这个值，然后根据记录的值重写数据项 X（假设为原始账户增加￥100）。T2 同样读写了 X（假设从账户取走￥300）。如果 T2 的行为发生在 T1 的读和写之间，那么 T2 对于账户的修改将丢失，即对于我们的例子而言，该账户最终将增加￥100 而不是减少￥200。游标稳定中的事务将根据查询游标在最近读取的数据项上加一个锁，当游标移走（如数据被提取）或者事务中止时释放该锁。游标稳定允许事务对个别数据项目按照“读—处理—写”的顺序来操作，其间避免了其他事务的更新干扰。

- **快照隔离**：一个以快照隔离方式运行的事务，只对自身开始时的数据版本进行操作，不受在这个时间点后发生的其他事务对该数据的改变的影响。这是 MVCC 在数据库产品中的主要应用之一。当事务开始时，它从一个单调递增的计数器中得到一个开始时间戳，当它成功提交时得到一个终止时间戳。对于一个事务 T 而言，只有当具有与 T 重叠的开始/结束时间戳的其他事务不去写事务 T 要写的数据时，事务 T 才会提交。这种隔离模型更依赖于多版本并发的实现，而不是锁机制。当然了，在支持快照隔离的系统中这些方案可以共存。
- **读一致**：这是 Oracle 定义的一种 MVCC 形式，它相对于快照隔离有一些不同。在 Oracle 中，每个 SQL 语句（一个事务中会有很多 SQL 语句）会看到语句开始之前最近的已提交数据版本。对于需要从游标处取数据的语句，游标取值的版本以它打开的时间为准。这个级别的实现是通过保存元组的多个逻辑版本来实现的，这意味着一个事务可能引用一个元组的多个版本数据。但是，Oracle 并不保存每一个可能需要的版本，它只存储最近的版本。当需要一个旧版本的数据时，Oracle 通过对现有版本依据日志记录进行回滚处理来得到旧版本。修改的顺序由长期写锁来保证，当两个事务需要写同一个数据项时，第二个提出写请求的事务将等待第一个提出写请求的事务完成后才能进行自己的写操作。相对而言，在快照隔离中，第一个已提交的事务（而不是第一个提出写请求的事务）将首先写数据。

弱一致性级别相对于完全可串行化而言可以提供更高的并发性能。因此，一些系统甚至默认设置成弱一致性级别。比如 Microso SQL Server 将“已提交读”设置为默认级别。但是，隔离性（ACID 中隔离性的含义）不能够得到保证。因此，应用开发者需要使用正确的方案来确定他们的事务运行正确。由于需要根据具体操作来定义这种机制的语义，因此，这种机制实现较复杂，并且使得应用很难在不同 DBMS 之间迁移。

6.4 日志管理器

日志管理器的主要功能包括：保持已提交事务的持久性、协助中止事务的回滚以确保原子性、在系统崩溃或非正常关机时使系统恢复。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拥有一个数据结构集。为支持系统崩溃后的恢复，内存中数据结构需要通过日志和数据库中的数据进行重建。

数据库日志是一个十分复杂并且细节繁多的话题。关于数据库日志的官方参考书目是

ARIES 上的期刊文章[59]，任何数据库领域的专家都应该对这些文章很熟悉。ARIES 的文章不仅解释了日志协议，而且也给出了关于其他设计方法及其引发的问题的讨论。这虽然有些啰嗦，但并不影响它们是一些好文章。如果想获取更多的概要性介绍，读者可以选择 Ramakrishnan 和 Gehrke 的教材[72]，其中对于基本的 ARIES 协议给出了说明并且没有额外的讨论和描述。这里我们讨论关于恢复的几个基本思想，并将试着解释教材和期刊之间对于该问题描述的不同。

数据库标准恢复机制使用写前日志（WAL: Write Ahead Log）协议。WAL 协议有三个规则：

- 对于数据库页的每一次更新都会产生一个日志记录，在数据库页被刷新到磁盘之前，该日志记录必须被刷新到日志系统中。
- 数据库日志记录必须按顺序刷新，即在日志记录 r 被刷新时，必须保证所有 r 之前的日志记录都已经被刷新了。
- 如果一个事务提出提交请求，那么在提交返回成功之前，该提交日志记录必须被刷新到日志设备上。

很多人只记得第一个规则，但是，实际上为了保证正确执行，这三个规则都是必须的。

第一条规则保证了事务中止时，未完成的事务可以被撤销，这保证了原子性。第二和第三条规则保证了持久性：系统崩溃后，如果一个已提交事务并没有反映在数据库上，那么该事务可以被重做。

虽然上面三条规则很简单，但是，我们会惊奇地发现，高效的数据库日志系统实际上包含了许多细节。在实际应用中，由于需要提供高效的数据库性能，上述简单的规则会变得非常复杂。我们面临的挑战是，在保证事务提交高效进行的情况下，同时保证高效的事务回滚和中止操作，并且能在数据库崩溃的情况下快速恢复。当需要增加与特定应用相关的优化功能时，日志将更为复杂，比如，对于只增或只减类型的字段提供更好的性能。

为了最大化快速通道的速度，大多数商用数据库系统以 Haerder 和 Reuter 称为“DIRECT,STEAL/NOT-FORCE”[34]的模式运行：（a）数据项原地更新，（b）未被“钉住”(pin)的缓冲页，即便在包含未提交数据的情况下也可以被“偷走”（原缓冲页会被写入磁盘），（c）当提交请求成功返回给用户时，缓冲页不必被强制刷新到数据库。这些规则保证 DBA 选定的数据位置不变，并且给予缓冲管理器和磁盘调度器充分的自由来管理内存和 I/O 策略，而不必考虑事务的正确性。这些特点虽然可以有较多的性能提升，但是，也需要日志管理器能够高效地处理已中止事务的缓冲页被刷新到磁盘的情况，对它们执行撤销操作，并且

需要日志管理器处理已提交事务的未被刷新到磁盘的缓冲页在崩溃后丢失的情况，对它们执行重做操作。一些数据库使用的优化方法是，把 DIRECT,SREAL/NOT-FORCE 系统的可扩展性优势和 DIRECT,NOT-STEAL/NOT-FORCE 系统的性能优势这二者进行结合。在这种系统中，缓冲区中的页不会被偷走，除非缓冲区中不再有干净页，这种情况下，系统退化为 STEAL 策略并拥有上文所说的优势。

另一个日志系统快速通道的问题是保证日志记录尽可能小，这样可以提高日志 I/O 的吞吐量。一种容易想到的优化方案是记录逻辑操作（如 insert(Bod,$25000) into EMP）而不是物理操作（如元组插入后的字节范围情况，包括堆文件以及索引块中的字节）。这样做的代价是，使得逻辑上的撤销和重做操作变得十分复杂。这会在事务中止或者数据库恢复时严重降低性能。在实际操作中，我们使用物理操作和逻辑操作混合的日志模式（被称为物理逻辑日志）。在 ARIES 中，物理日志被用来支持重做操作，而逻辑日志被用来支持撤销操作。这是 ARIES 规则中用于恢复的“重复历史”方法的一部分，即首先到达崩溃状态点，然后从那个点开始回滚事务。

崩溃恢复是必须的，它可以在数据库崩溃或者非正常关闭后将数据库恢复到一个一致性状态。正如之前所说的，恢复在理论上是将从第一条日志记录开始直至最后一条记录的历史重演。这个理论本身没错，但是，考虑到日志可能很长，它还是不够高效。我们不需要从第一条日志开始恢复，我们可以选择从以下两条日志中选择最老的一条开始恢复，也能够得到正确的结果：（1）缓冲池中描述对于最旧的一个脏页的最早更改的日志记录，（2）表示系统中最老的一个事务开始的日志记录。这个点的序号被称为恢复日志序号（recovery LSN）。由于计算并记录恢复日志序号依旧耗费时间，既然已知该序号是递增的，那么我们不必一直计算。我们可以周期性地选择一个时间点（称为检查点）作为计算时间。

一个单纯的检查点将刷新缓冲区中所有的脏页，并且计算、存储恢复日志序号。当缓冲区很大时，这将导致几秒的 I/O 延迟。所以，我们需要一个更为“模糊”的检查点，同时，还要有一套逻辑，可以使得检查点可以跟上最新的一致状态，并且尽可能少地处理日志。ARIES 使用一个很聪明的模式，它所采用的检查点记录非常小，仅仅包含了一些必要的信息，这些信息可以用来发起日志分析过程，并且可以用来重建在崩溃时丢失的内存中的数据结构。在 ARIES 的模糊检查点策略中，恢复日志序号在计算时不会导致脏页被同步刷新。当然了，这种策略需要另一个方案来决定异步刷新脏页的时间。

我们注意到回滚是需要写日志页的。这会导致正在执行的事务因得不到日志页空间而不能执行但是又不能回滚的情况。这种情况可以通过空间保留策略来避免，但是，当遇到不同

版本的系统时，这种策略难以保证正确。

最后我们要说，考虑到数据库不仅仅是磁盘页上的一个个元组，它也包括了很多控制磁盘数据结构的物理信息，因此，日志和恢复的任务将变得更加复杂。我们将在下一节讨论日志索引的情况。

6.5 关于索引的锁和日志

索引是数据库中为快速获得数据而使用的物理存储结构。索引本身对于数据库应用开发者是看不到的，除非他们需要提高索引性能或加强约束条件。开发者和应用程序不能直接看到或控制索引的接口。这使得索引可以通过更为高效（也很复杂）的事务机制来管理。唯一不变的是，关于索引的并发和恢复，需要保证索引返回的总是数据库中事务一致性的元组。

6.5.1 B+-树中的 latch 机制

有一个关于 B+-树 latch 机制的很好的例子（译者注：latch 是一种轻量级锁）。B+-树包含了通过缓冲池存取的数据库磁盘页，就像数据页一样。有一种索引并发控制是对索引页使用两段锁协议。这意味着，每一个访问索引的事务都需要在提交之前一直锁住 B+-树的根，这是一种有限的并发。也有许多基于封锁机制的方案在这个问题上没有对索引页使用任何事务锁。这些方案的真正核心问题是：在保证所有并发事务都能找到叶子节点上正确的数据的情况下，关于树的物理结构的改变（如节点分裂）可以采用一种非事务方式。大概包含三种方法：

- **保守方案**：如果多个事务需要访问同一个页面，那么，只有当它们能够保证在使用页面内容不发生冲突时才被允许访问。列举一个冲突的例子是：当一个读事务正在读取一整个页面时，另一个插入事务在该页面中节点的下方正在进行插入操作，这可能导致该页面分裂[4]。相对比下面列举的较新的方案，这些保守的方法牺牲了太多的并发性。
- **联结锁(latch-coupling)方案（蟹行协议）**：在遍历每一个树节点之前首先添加逻辑锁，当遍历到一个节点 u 时，首先将其锁住，只有当下一个需要访问的节点已经被锁住时，才释放节点 u 的锁。这个方案也被称为“蟹行”加锁，因为它的工作方式类似于螃蟹行走：锁住一个节点，抓住它的孩子，然后释放父母，并且一直重复。联结锁应用于许多商业系统中；IBM 的 ARIESIM 版本对该协议有很好的描述[60]。ARIES-IM 包括一些相当复杂的细节和小案例——如在分裂发生后必须重新

开始遍历，甚至要对整棵树锁住。

- **右链接协议**：我们在 B+-树中添加了一些简单的结构，来尽可能地减少锁和重新遍历的情况。在该协议中，我们对每一个节点添加一个指向其右边邻居的链接。在遍历过程中，右链接协议不必使用联结锁——每一个节点都可以被锁住、读取然后释放锁。右链接协议的主要灵感来自于，如果一个遍历事务跟随一个指针找到节点 n，而节点 n 同时发生了分裂，那么，遍历事务在确认分裂后可以通过右链接向右移动来找到树中的正确位置[46,50]。一些系统出于同样的考虑提供了右链接来支持反向遍历。

Kornacker 等人[46]提供了关于蟹行协议和右链接协议的细节讨论，并指出了蟹行协议只适用于 B+-树，对于更复杂的索引树将不起作用，比如没有单一线性规律的地理数据。PostgreSQL 广义查询树（GiST）的实现，就是基于 Kornacker 等人针对右链接协议的扩展。

6.5.2 物理结构的日志

除了特殊情况并发逻辑以外，索引也使用特殊情况下的日志逻辑。这些逻辑使得日志记录和恢复十分高效，但也增加了代码的复杂性。其主要想法是，在相关的事务中止后，结构索引所发生的改变并不需要重做，因为，这样的改变往往并不影响其他事务处理数据库元组。举例来说，如果一个 B+-树的页在一个插入事务执行的过程中发生了分裂，但该事务突然中止了，那么，在事务中止处理中我们不必取消分裂操作。

这就面临一个挑战，那就是，我们必须为一些日志记录贴上 redo-only 的标签。在任何日志的撤销过程中，贴有 redo-only 标签的改变将不会被改动。ARIES 为这种情况提供一个简洁的定义——nested top actions——允许恢复进程在恢复时跳过关于物理结构更改的日志记录而不使用任何特殊的代码。

在其它一些情况下也会用到相同的办法，比如堆文件。对一个堆文件进行插入操作可能会引起文件被扩展到磁盘上。为了解决该问题，这些改变必须反映在文件 extent map 上（译者注：现代很多文件系统都采用了 extent 替代 block 来管理磁盘），这是磁盘上一个指向组成文件的连续磁盘块的数据结构。如果插入事务中止，这些针对 extent map 的改变不需要取消，因为，这个文件增大是一个对事务不可见的副作用，这有可能对终止将来的插入操作有用。

6.5.3 Next-Key 锁定：逻辑性能的物理代理

让我们以最后一个索引并发问题来结束本章，该问题解释了一些虽然细微却意义重大的想法。这个问题是：在允许元组级别的锁并使用索引的情况下，我们如何提供完全的可串行

化（包括防止幽灵问题出现）。需要注意的是，这种技术仅适用于提供完全可串行化的情况，而在一些宽松的隔离性模型中，这种技术就不需要考虑了。

幽灵问题会在事务通过索引访问元组时出现。在这种情况下，事务仅会锁住它需要通过索引访问的元组，而不是锁住整个表（如 Name BETWEEN ‘Bob’ AND ’Bobby’）。在没有锁住整个表的情况下，其它的事务可以向表中插入新的元组（如”Name = ‘Bobbie’”）。当插入的数据符合查询谓词所限定的范围时，它们将出现在该谓词查询的结果中。需要注意的是，幽灵问题关系到数据库中元组的可见性，因此，这不仅仅是锁存器的问题，也是关于锁的问题。理论上来说，我们需要的是锁住原始查询谓词所表示的逻辑区域，如按照词典顺序落入“Bob”和“Bobby”之间的所有字符串。不幸的是，由于谓词锁需要比较许多谓词的重叠区域，它的实现代价太昂贵了。我们不可能用一个哈希锁表来完成这项任务[3]。

一个通用的解决 B+-树中的幽灵问题的办法是 next-key 锁定。在 next-key 锁定中，索引插入算法经过了这样的修改：当索引键值为 k 的元组被插入时，必须为索引中拥有大于 k 的最小键值的下一个元组分配一个排他锁。该协议使得后续的插入操作不会出现在之前活动的事务所获得的两个元组的中间。这也使得元组不能被插入到之前被返回的具有最低键值的元组后面。比如，如果在第一次访问中没有发现“Bob”键，那么，在同一个事务的后续操作中也不可能发现它。还有一种情况：元组正好被插入到之前返回的具有最大键值的元组的上面。为了防止该情况的发生，next-key 锁定协议要求读事务在下一个元组上添加一个共享锁。在这种情况下，next-key 元组将是不符合查询谓词的、具有最小键值的元组。尽管存在优化的可能性，并且也常常存在优化方法，但是，更新操作仍然通常表现为逻辑上的删除操作后面跟随一个插入操作。

Next key 锁定虽然有效，但是在一些特殊负载的情况下也会引发封锁过多的情况。比如，如果我们从关键字 1 扫描到关键字 10，但是，有索引的关键字只有 1、5、100。这种情况下，从 1 到 100 的整个区间都会被锁住。

Next key 锁定并不仅仅是一个简单的小技巧。它是使用物理对象（一个当前存储元组）来作为一个逻辑概念（谓词）的代理（surrogate）的例子。这样做的好处是，简单的系统结构（例如哈希锁表）可以被用来实现更为复杂的功能，比如更改锁协议。复杂软件的设计者应该将这种逻辑替代的普适方法作为自己的常备技巧以便解决类似问题。

6.6 事务存储的相互依赖

我们早在本节开头就提到事务存储系统是庞大而错综复杂的系统。在这一部分中，我们来谈一谈事务存储系统的三个主要部分的相互依赖性：并发控制、恢复管理和存取方法。如果世界没那么复杂，我们应该可以准确地定义模块之间的 API，这样便可以使得接口的实现方式更为独立。但是，我们即将举出很多例子来证明这并不容易做到。我们不会在这里给出一个详尽的接口列表，因为这是一项艰巨的任务。但是，我们会尽可能说明事务存储方面的一些复杂的逻辑，进而说明商业 DBMS 中为何会用如此庞大复杂的实现方式。

我们首先仅考虑并发控制和恢复机制而忽略更复杂的存取方法。即便经过如此简化，各个组件之间的关系还是错综复杂的。并发控制和恢复机制的复杂关系，一方面表现在写前日志对于锁协议做了隐含假设。写前日志需要在严格两段锁协议下才能正确执行。为了说明这一点，我们首先来看一下当一个中止事务回滚时会发生什么。恢复代码开始执行中止事务的日志记录，撤销它所做的改变。这通常会改变事务已经更改过的页或元组。为了完成这些改变，事务需要在这些页和元组上加锁。在非严格两段锁协议中，如果一个事务在中止前取消了任何锁，那么，它将不能够重新申请该锁来完成回滚操作。

存取算法使得事情更为复杂。实现教材中的存取算法（如线性哈希[53]或者 R-树[32]）并且在事务系统中为其实现一个高并发可恢复的版本，是一个非常技巧性并且工程性的挑战。因此，很多先进的 DBMS 系统仍然仅实现了堆文件和 B+-树来作为事务存取方法；但是，PostgreSQL 的 GiST 的实现是一个例外。正如我们之前提到的 B+-树一样，高效的事务索引的实现，包括复杂的关于 latch、锁以及日志的协议。正规 DBMS 中的 B+-树在发生并发调用以及恢复时，会遭遇很多挑战。即便简单的类似堆文件的存取方法，也会在描述其内容的数据结构（如 extent map）方面面临难以捉摸的并发和恢复问题。这些逻辑并不是普遍存在于所有的存取方法中——会因为存取方法的特定逻辑和特殊实现而不同。

并发控制只有在锁方案出现后才得到了很好的发展。其他并发方案（如乐观或者多版本并发控制）较少考虑到存取方法，或者只是随便不切实际地提了一下[47]。所以，将不同并发机制跟一个给定的存取方法的实现配搭起来是很困难的。

存取方法中的恢复逻辑也是系统定制的：日志记录的存取方法的时间设置和内容依赖于恢复协议良好的细节，包括结构更改的处理（如事务回滚时相关日志是否应该被撤销，如果不应撤销该怎样避免），以及针对物理日志和逻辑日志的使用。即便对于特殊的存取算法例如 B+-树，它的恢复和并发逻辑也是错综复杂的。一方面，恢复逻辑依赖于并发策略：如果

恢复管理器必须恢复树的物理一致性状态，那么，它需要知道可能出现的不一致状态是怎样的，然后通过日志将这些状态适当归类以保持原子性（如通过 nested top action）。在另一方面，存取算法的并发协议也依赖于恢复逻辑。例如，B+-树的右链接策略假设树的数据页在分裂后永远不会重新合并。这种想法需要恢复机制采取比如 nested top action 这样的机制去避免因事务中止而撤销分裂操作。

在整个架构图中很显著的一点是，缓冲区管理跟其他组件的关系是比较独立的。只要保证钉住页面的动作正确执行，缓冲区管理器就可以将它剩下的逻辑打包起来，然后在需要的时候重新实现。例如，缓冲区管理器可以自由地选择替换哪个页（因为有 STEAL 性质），也可以自由地调度页面刷新（感谢 NOT FORCE 性质）。当然了，之所以能取得这样的隔离效果，是因为复杂的并发和恢复策略。所以，这一点可能不像它看起来那样显著。

6.7 标准实践

今天所有的数据库产品都支持 ACID 事务。作为一个规则，它们使用写前日志来保证持久性，使用两段锁协议来保证并发控制。但是，PostgreSQL 是个例外，它自始至终使用多版本并发控制协议。Oracle 首先有限度地使用了多版本并发控制，并用锁机制加以配合，来提供较为宽松的一致性模型，如快照隔离和读一致性；这些模型深受用户喜爱，也促使了更多商业数据库系统去实现它们，在 Oracle 中这些模型是默认的。B+-树索引在数据库产品中已经成为了一种标准，大多数商业数据库也会在系统本身或者插件中提供一些其他的多维索引结构。只有 PostgreSQL 通过实现 GiST 来提供了高并发多维的文本索引。

MySQL 有一点跟其他数据库不同，它支持许多不同的底层存储管理器，在这一点上，DBA 可以在同一个数据库中为不同的表选择不同的存储引擎。它的默认存储引擎 MyISAM 只支持表级别的锁，但是，在以读操作为主的负载情况下，它将是一个高并发的选择。对于读/写工作负载，我们推荐 InnoDB 存储引擎，它提供行级别的锁（InnoDB 在几年前被 Oracle 购买，但目前仍保持对用户开源并免费）。但是，MySQL 并没有哪一款存储引擎提供著名的针对 R 系统的分层锁(hierarchical locking)机制[29]，尽管该机制在其它数据库系统中应用广泛。这使得 MySQL 的 DBA 在选择 InnoDB 或者 MyISAM 时非常痛苦，在一些混合工作负载的情况下，没有哪个引擎可以提供好的锁粒度。于是，DBA 必须通过多重表或者数据库复制开发一种物理设计，从而可以支持扫描和高选择度的索引方法。MySQL 也支持针对主存和聚类存储的存储引擎，一些第三方组织也提供了支持 MySQL 的存储引擎，但是，

当今 MySQL 的使用还是集中在 MyISAM 和 InnoDB。

6.8 讨论及相关资料

事务处理机制目前已经是一个很成熟的话题了，在许多年里，大多数可能的技巧已经被尝试了。新的设计试图将现有的方法进行排列组合。在这个领域力，最明显的改变也许就是 RAM 价格的不断下降。这使得我们可以将更大比例的数据库中的“热数据”放入内存，并以内存速度执行，这也使得“如何把数据足够经常地刷新至永久性存储器以使得重启时间最短”这项工作变得更加复杂。事务管理中闪存的角色也是该平衡工作的一部分。

近年来一个有趣的发展是，写前日志在操作系统领域得到广泛应用，尤其是在日志文件系统下。这些已成为今天所有操作系统中的一种标准。由于这些文件系统并不支持针对文件数据的事务，它们如何通过使用写前日志来实现持久性以及原子性，是一件很有趣的事情。感兴趣的读者可以阅读文献[62,71]以获得更多信息。就这一点而言，另一个有趣的方向是 *Stasis*[78]中的一项工作——试图更好地模块化 ARIES 风格的日志和恢复理论以使得它能够被系统程序员更广泛地使用。

第 7 章 共享组件

本章内容介绍一些在几乎所有的商业 DBMS 中都存在、却很少在研究文献中讨论的共享组件和工具。

7.1 目录管理器

数据库的目录管理器以元数据的格式保存了系统中数据的相关信息。目录管理器记录了系统中基本实体（用户、模式、表、列、索引等）的名称及它们的关系，并且以一系列表的形式存储在数据库中。元数据保持与数据一样的格式，可以使得系统在使用上更加紧凑、简单：用户可以使用同一种语言和工具来研究其他数据的元数据，而且，内部系统用于管理元数据的代码大部分与管理其他表的代码一样。代码和语言的复用是重要的经验，但在早期的实现中常常被忽略，这对后来的开发者来说是一个重大的遗憾。在过去的十年里，本文的其中一位作者在工业界中一再目睹这种错误的出现。

出于效率原因，我们通常采用不同的方式来处理基本的目录数据和通常的表。目录中经常被访问的部分常常根据需求物化在内存中，所采用的数据结构，一般是对目录的扁平关系结构进行非规范化处理，转换成主存中的对象网络形式。内存中缺少的数据独立性是可以接受的，因为，内存中的数据结构只供查询解析器和优化器以程序化的方式使用。额外的目录数据在解析时会被缓存在查询计划中，这些数据常常也还是采用适用于查询的非规范化格式。此外，目录表通常服务于特殊情况的事务处理技巧，用来减少事务处理中的热点（hot spots）。

目录在商业应用中能变得非常庞大。例如，一个大型 ERP（企业资源计划）应用有超过 60,000 个表，每个表有 4 到 8 列，且通常每个表有 2 或 3 个索引。

7.2 内存分配器

教材介绍的 DBMS 内存管理往往全部集中在缓冲池上。实际上，数据库系统也分配了大量的内存用于其他任务。正确地管理这些内存不仅是编程的负担，也关系到性能问题。例如，Selinger 风格的查询优化可使用大量的内存用来建立动态编程过程中的状态。查询操作

符如哈希连接、排序在运行时会分配大量内存。商业系统中的内存分配可通过使用基于上下文的内存分配器来更有效、更容易地调试。

内存的上下文是内存中的数据结构，维护了一个连续的虚拟内存的区域列表，通常称其为内存池。每个区域都可以有一个小的头部，头部包含了一个上下文标签或一个指向上下文头部结构的指针。基本的内存上下文 API 包括如下调用：

- *创建给定名字或类型的上下文。*上下文的类型可以让分配器知道如何有效地处理内存分配。例如，用于查询优化器的上下文所使用的内存数量会以较小的增量逐渐增加，而用于哈希连接的上下文是以批量的方式来分配内存的。基于这样的知识，分配器会每次选择分配更大或更小的区域。
- *分配上下文中的一块内存。*这种分配方式会返回一个指针，指向内存（很像是传统的 malloc()调用）。这部分内存可能来自上下文中已存在的区域，或者，如果任何区域都不存在这样的空间，分配器会向操作系统请求一个新区域的内存，为它做标记，并链接到上下文中。
- *删除上下文中的一块内存。*这未必能使上下文删除相应的区域。内存上下文的删除有些不寻常，更为典型的行为是删除整个上下文。
- *删除上下文。*这首先会释放所有与上下文相关联的区域，然后删除上下文头部。
- *重置上下文。*这会保留上下文，但返回原始创建时的状态，往往是通过重新分配所有之前已分配区域的内存实现的。

内存上下文提供了重要的软件工程的优势。最重要的是它们作为底层的服务，是一种程序员可控的垃圾回收方式。例如，开发人员编写的优化器可以为特定的查询在优化器上下文中分配内存，而不用担心之后如何去释放内存。当优化器选择了最佳计划，它就可以针对查询从一个单独的执行器上下文中拷贝计划到内存中，然后删除查询的优化器上下文。这就不需要编写代码来小心地遍历所有优化器数据结构来删除它们的组件。它也避免了因代码的 BUG 而引起的棘手的内存泄漏。这个特征对于具有自然的“阶段性”的查询执行过程而言，是很有用的，因为，在查询执行过程中，从解析器到优化器再到执行器的整个控制流程，每个阶段的开始都会生成上下文并伴随着大量的内存分配，每个阶段的结束都会发生上下文的删除并伴随着大量的内存回收。

需要注意的是，内存上下文实际上为开发人员提供了比大多数垃圾回收器更多的控制，因为，开发人员可以在重新分配内存时控制空间和时间的局部性。上下文机制本身提供了空间控制，允许程序员将内存划分成多个逻辑单元。时间控制是指允许程序员在适当的时候发

起上下文删除操作。相反地，垃圾回收器通常在一个程序的所有内存上工作，并且会自己决定何时运行。这是尝试用 Java 来编写服务器级别的高质量代码时会遇到的一个挫折[81]。

内存上下文也为 malloc()和 free()开销相对高的平台提供了性能优势。尤其是内存上下文可以使用关于内存怎样被分配和重新分配的语义知识（通过上下文类型），然后相应地调用调用 malloc()和 free()来最小化操作系统开销。一些数据库系统的组件（比如解析器和优化器）分配了大量的小对象，然后通过上下文删除一次性释放它们。调用 free()来释放大量的小对象在多数平台上会带来很大的开销。实际上，内存分配器可以调用 malloc()来分配大区域内存，并将该内存分配给其调用者。这样做就不再需要内存重新分配，这也就意味着，malloc()和 free()所使用的压缩逻辑（compaction logic）也就不再需要了。当上下文删除时，移除大区域内存只需要少数的 free()调用。

感兴趣的读者可浏览 PostgreSQL 的开源代码，它使用了相当先进的内存分配器。

7.2.1 为查询操作符分配内存时的注意事项

数据库厂商们为空间密集型的操作符（如哈希连接和排序）所采用的内存分配方案各有不同。一些系统（比如 DB2 for zSeries）允许 DBA（数据库管理员）控制这些操作能使用的 RAM 数量，并且保证每个查询在执行时都能获得该数量的 RAM。准入控制策略确保了这一保证。在这样的系统中，操作符通过内存分配器从堆中分配它们的内存。这些系统提供了很好的性能稳定性，但是，会强制 DBA 决定如何去平衡各个子系统（如缓冲池和查询操作符）之间的物理内存使用。

其他系统（比如 Microsoft SQL Server），从 DBA 手中接管了内存分配任务，实现了内存分配的自动化管理。这些系统尽量智能地在查询执行的多个组件中分配内存，包括缓冲池中的页面缓存和查询操作符的内存使用。用于所有这些任务的内存池即缓冲池本身。因此，在这些系统中的查询操作符通过 DBMS 实现的内存分配器从缓冲池中取得内存，并且，只在连续地请求超过缓冲池页面大小的内存时才使用操作系统的分配器。

这个区别呼应了第 6.3.1 节中关于查询准备的讨论。前一类系统假设由 DBA 来完成复杂的调优工作，DBA 对系统内存各种参数进行仔细配置后，系统的工作负载将会服从于这些配置好的参数。在这些条件下，这样的系统应该总是能够像预期的那样很好地执行。后一类系统则假设 DBA 不能正确地设置这些参数，并努力以软件逻辑来代替 DBA 的手工调整。系统保留了自适应改变其相关分配的权力，这为在可变的工作负载上获得更好的性能提供了

可能性。如第 6.3.1 节所讨论的那样，从这个区别中不仅可以看出这些数据库厂商希望其产品如何被使用，也可以看出他们顾客的管理经验（和财务资源）。

7.3 磁盘管理子系统

DBMS 教材往往将磁盘视为同构对象。实际上，磁盘驱动器是复杂的异构硬件，在容量和带宽上大不相同。因此，每个 DBMS 都有一个磁盘管理子系统来处理这些问题，来管理表的分配和其他原始设备、逻辑卷或文件中的存储单元。

这个子系统的其中一个责任就是将表映射到设备和（或）文件上。表到文件一对一的映射听起来很自然，但在早期文件系统中会带来明显的问题。首先，传统的操作系统文件不能比磁盘大，而数据库表则可能需要跨越多个磁盘。其次，分配过多的操作系统文件被认为是不好的形式，因为，操作系统往往只允许少数的打开文件描述符，而且许多操作系统目录管理和备份的工具不能扩展到大量文件的情形。最后，许多早期的文件系统限制了文件大小上限为 2GB。如此小的表限制显然是无法接受的。许多 DBMS 厂商绕开了操作系统的文件系统而完全使用原始 IO，而其他厂商选择去解决这些限制。因此所有领先的商业 DBMS 可能会将一个表跨越多个文件，或在单个数据库文件中存储多个表。随着时间的推移，大多数操作系统的文件系统已改进解决了这些限制。但遗留的影响依然存在，而且现代 DBMS 往往仍将操作系统文件视为任意映射到数据库表的抽象存储单元。

更为复杂的是处理特定设备细节的代码，这些代码用来维护第 4 章所描述的时间和空间控制。基于复杂存储设备的产业今天依然存在，并且充满活力，它们把复杂存储设备伪装成磁盘驱动器，但实际上是一个大型硬件/软件系统，它的 API 还是遗留的磁盘驱动接口，如 SCSI。这些系统包括 RAID 系统和存储区域网络（SAN）设备，往往拥有超大容量和复杂的性能特征。管理员喜欢这些系统是因为它们易于安装，并且通常提供了易于管理的、位级（bit-level）的可靠性，可以支持快速失败恢复。这些特征为客户提供了重要的舒适感，远超 DBMS 恢复子系统的承诺。例如，大型 DBMS 配置一般使用存储区域网络。

不幸的是，这些系统使 DBMS 的实现变得更加复杂。例如，RAID 系统在发生错误之后和所有磁盘都正常运行时的性能表现大不相同。这潜在地使 DBMS 的 I/O 成本模型变得复杂。一些磁盘可以在开启写缓存的模式下操作，但在硬件故障时会导致数据损坏。先进的存储区域网络实现了大型的带后备电源的缓存，在一些情况下接近 TB 级，但这些系统本身具有超过百万行的代码和相当的复杂性。复杂性带来了新的失败模式，这些问题可能是非常难

以检测和正确诊断的。

RAID 系统在数据库任务上表现不佳，也使数据库的设计者失望。RAID 的构想是面向字节流的存储（Ia 后缀的 UNIX 文件），而不是用于数据库系统的面向页的存储。所以，与在多个物理设备上分区和复制的特定数据库解决方案相比较时，RAID 设备往往表现不佳。例如，Gamma 的“*chained declustering*”方法[43]，大体上与 RAID 方式一致，而且在 DBMS 环境中运行得更好。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 设备把多个设备隐藏在单一接口后面，破坏了这些命令。

当数据库在更加简单的方案如磁盘镜像（RAID1）下可以表现出非常好的性能时，许多用户会配置他们的 RAID 设备（RAID5），从而最小化空间开销。RAID5 有个特别不好的特征，那就是写入性能很差。这会对用户造成出人意料的瓶颈，而且，DBMS 厂商常常需要忙于向客户解释这个问题，或提供解决这些瓶颈的变通方法。无论如何，RAID 的使用（以及不当使用）实际上是商业 DBMS 必须考虑的。结果是，多数 DBMS 厂商花费大量的精力去调整他们的 DBMS，从而使其在领先的 RAID 设备上能够很好地工作。

在过去的十年，多数客户的部署是分配数据库存储到文件上，而不是直接分配到逻辑卷或原始设备上。但是，多数 DBMS 仍然支持原始设备访问，在运行大规模事务处理基准测试时，常常使用这种存储映射。而且，尽管有上面描述的一些缺点，多数企业 DBMS 存储如今还是使用 SAN。

7.4 备份服务

通过周期性更新来在网络上备份数据库常常是一种较为可行的方案。它被经常使用是为了额外的可靠性：在主服务器宕机的情况下，备份数据库可以作为稍微有些过时的“热备”（warm standby）。使热备处于不同的物理位置上，有利于在火灾或其他大灾难发生后能继续运行。备份也常常用于为大型的、地理上分散的企业提供了一个实际的分布式数据库功能。多数这样的企业将其数据库按大的地理区域（如国际或洲际）进行分区，并且在数据库的主副本上本地运行更新。查询也是本地执行的，但是，可以运行在混合的数据集合上，也就是一部分数据属于本地操作所获取的新数据，另一部分数据是从远程区域复制来的稍有些过时的数据。

如果忽略硬件技术（比如 EMC SRDF），那么，可以有三个用于备份的典型方案，但只有第三个方案提供了高端环境所需的性能和可扩展性，当然这也是最难实现的。

1. **物理备份**：最简单的方案是在每一个备份周期物理地备份整个数据库。考虑到传送数据的带宽和在远程站点重新设置时的成本，这个方案不能扩展到大型数据库上。此外，保证数据库的事务的一致快照是困难的。因此，物理备份只作为低端客户端上的变通方案。多数数据库厂商以不鼓励通过任何软件方式使用此方案。
2. **基于触发器的备份**：在这个方案中，触发器置于数据库表中，这样，当表中发生任何插入、删除或更新操作时，一条表示变化的记录便会被放置在一个特殊的备份表中。这个备份表传送到远程站点以后，相应的变化操作就会在远程站点被“回放”（即重新执行一次）。这个方案解决了上述物理备份所提到的问题，但带来的性能损失对一些工作负载来说是不能接受的。
3. **基于日志的备份**：在一些可行的场景，基于日志的备份是应该选择的备份解决方案。在基于日志的备份中，一个日志嗅探器进程截取日志写入，并将其传送到远程系统。基于日志的备份的实现使用了两个技术：（1）读取日志并建立 SQL 语句，并在目标系统上重新执行，或（2）读取日志记录并将其传送到目标系统，系统处于持续的恢复模式，当日志记录到达时重新执行。这两个机制都有其价值，所以，Microsoft SQL Server、DB2 和 Oracle 都实现了两者。SQL Server 称第一个为日志传送，称第二个为数据库镜像。

这个方案克服了前两个备选方案的所有问题：它是低开销的，在运行的系统上只会引起最小的或不引人注意的性能开销；它提供了增量更新，因此，可以优雅地扩展数据库的大小和更新速率；它复用了 DBMS 内置的机制而不用太多额外的程序；最后，它通过日志的内置程序自然地提供了事务一致性。

大多数主流的数据库厂商都为其系统提供了基于日志的备份。提供能在多个数据库厂商之间工作的基于日志的备份，要困难得多，因为，使数据库厂商在远程终端重新执行逻辑需要知道数据库厂商的日志格式。

7.5 管理、监控和工具

每个 DBMS 都提供了一系列工具来管理其系统。这些工具很少用于基准测试，但常常能影响系统的易管理性。技术上的挑战和格外重要的特征是要让这些工具能在线运行，也就是说，当用户查询和事务正在进行时去运行这些工具。这对于提供“24×7”服务而言是很

重要的，随着电子商务在全球范围内的发展，这种服务在近几年变得更为常见了。传统上的凌晨时的“维护窗口”通常已不再存在。所以，多数数据库厂商在这几年投入大量精力来提供在线工具。在此我们总结一下这些工具提供的功能：

- **优化统计信息收集**：每个主流的 DBMS 都有一些方法来扫描表并构建关于排序或其他操作的优化器统计信息。一些统计信息（如柱状图），如果没有超大容量的内存，是很难在一次扫描中构建的。例如，可以看看 Flajolet 和 Martin 在计算一列中不同值的个数时的工作[17]。
- **物理重组和索引建立**：随着时间的推移，存取方法会因为插入和删除的模式所留下的未使用空间而变得效率低下。而且，用户可能会偶尔地请求让表在后台进行重组，例如，在不同的列上重新聚集（排序）数据，或者在多个磁盘上重新分区。在线重组文件和索引是困难的，因为，维护物理的一致性必须避免在任意长的时间里一直保持加锁状态。从这个意义上说，它和第 5.4 节中所描述的用于索引的日志和锁协议比较类似。这已经是一些研究论文[95]和专利的主要研究内容。
- **备份/导出**：所有的 DBMS 都支持物理地转储数据库到备份存储的能力。同样，因为这是个长期运行的进程，它不能简单地设置锁。因此，大多数系统执行一些“模糊（fuzzy）”转储，并配以日志逻辑来确保事务一致性。相似的方案可用于以交换格式导出数据库。
- **批量加载**：在许多场景中，大量的数据需要快速载入到数据库中。数据库厂商提供了针对高速数据导入而优化过的批量加载工具，而不是每次插入一条数据。通常这些工具通过存储管理器的自定义代码来实现的。例如，针对 B+-树的、特定的批量加载代码，会比反复调用插入树的代码要快得多。
- **监控、调优和资源管理器**：即使在托管环境中，查询会消耗超过所需的资源也是寻常的。所以，多数 DBMS 提供了工具来协助管理者辨别和预防这些问题。它通常提供了基于 SQL 的接口来通过虚拟表访问 DBMS 的性能计数器，可以显示由查询或锁、内存、临时存储等资源所导致的系统状态。在一些系统中，它也有可能去查询这些数据的历史日志。许多系统允许注册一些报警，从而在查询超过了指定性能限制时发出警报，性能限制包括运行时间、内存或锁的获取。有时候触发一个警报会造成查询中止。最后，如 IBM 的预测资源管理器工具，会尽量预防资源密集型的查询被运行。

第 8 章 结束语

从本文中应该清楚的是，现代商业数据库系统是构建在两个基础之上的，一个是学术研究，另一个是为高端客户开发工业级别的产品所积累的大量经验。编写和维护一个高性能、全功能的关系型 DBMS 的任务，需要投入巨大的时间和精力。然而，许多关系型 DBMS 的经验已转化到新的领域。Web 服务、网络附加存储、文本和电子邮件库、通知服务和网络监控等，这些领域都可以从 DBMS 的研究和经验中受益。数据密集型的服务是当今计算的核心，数据库系统设计的知识是可以被广泛应用到各个领域的技能，既包括数据库领域也包括其他领域。这些新的应用方向，也带来了一些数据库管理方面的研究问题，这为数据库社区和其他计算领域之间的交互开辟了新的道路。

致谢

英文原文的作者对以下人员表示感谢： 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。

参考文献

[1] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in 16th International Conference on Data Engineering (ICDE), San Diego, CA, February 2000.

[2] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modelling: Alternatives and implications," ACM Transactions on Database Systems (TODS), vol. 12, pp. 609–654, 1987.

[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: Relational approach to database management," ACM Transactions on Database Systems (TODS), vol. 1, pp. 97–137, 1976.

[4] R. Bayer and M. Schkolnick, "Concurrency of operations on B-trees," Acta Informatica, vol. 9, pp. 1–21, 1977.

[5] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis, "A genetic algorithm for database query optimization," in Proceedings of the 4th International Conference 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, "A critique of ANSI SQL isolation levels," in Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 1–10, San Jose, CA, May 1995.

[7] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," ACM Computing Surveys, vol. 13, 1981.

[8] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, "The oracle universal server buffer," in Proceedings of 23rd International Conference on Very Large Data Bases (VLDB), pp. 590–594, Athens, Greece, August 1997.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra,

A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” in Symposium on Operating System Design and Implementation (OSDI), 2006.

[10] S. Chaudhuri, “An overview of query optimization in relational systems,” in Proceedings of ACM Principles of Database Systems (PODS), 1998.

[11] S. Chaudhuri and U. Dayal, “An overview of data warehousing and olap technology,” ACM SIGMOD Record, March 1997.

[12] S. Chaudhuri and V. R. Narasayya, “Autoadmin ‘what-if’ index analysis utility,” in Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 367–378, Seattle, WA, June 1998.

[13] S. Chaudhuri and K. Shim, “Optimization of queries with user-defined predicates,” ACM Transactions on Database Systems (TODS), vol. 24, pp. 177–228, 1999.

[14] M.-S. Chen, J. Hun, and P. S. Yu, “Data mining: An overview from a database perspective,” IEEE Transactions on Knowledge and Data Engineering, vol. 8, 1996.

[15] H.-T. Chou and D. J. DeWitt, “An evaluation of buffer management strategies for relational database systems,” in Proceedings of 11th International Conference on Very Large Data Bases (VLDB), pp. 127–141, Stockholm, Sweden, August 1985.

[16] A. Desphande, M. Garofalakis, and R. Rastogi, “Independence is good: Dependency-based histogram synopses for high-dimensional data,” in Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, February 2001.

[17] P. Flajolet and G. Nigel Martin, “Probabilistic counting algorithms for data base applications,” Journal of Computing System Science, vol. 31, pp. 182–209, 1985.

[18] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten, “Fast, randomized join-order selection — why use transformations?,” VLDB, pp. 85–95, 1994.

[19] S. Ganguly, W. Hasan, and R. Krishnamurthy, “Query optimization for parallel execution,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 9–18, San Diego, CA, June 1992.

[20] M. Garofalakis and P. B. Gibbons, “Approximate query processing: Taming the terabytes, a tutorial,” in International Conferenence on Very Large Data Bases, 2001. www.vldb.org/conf/2001/tut4.pdf.

[21] M. N. Garofalakis and Y. E. Ioannidis, “Parallel query scheduling and optimization with

time- and space-shared resources," in Proceedings of 23rd International Conference on Very Large Data Bases (VLDB), pp. 296–305, Athens, Greece, August 1997.

[22] R. Goldman and J. Widom, "Wsq/dsq: A practical approach for combined querying of databases and the web," in Proceedings of ACM-SIGMOD International Conference on Management of Data, 2000.

[23] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 102–111, Atlantic City, May 1990.

[24] G. Graefe, "Query evaluation techniques for large databases," Computing Surveys, vol. 25, pp. 73–170, 1993.

[25] G. Graefe, "The cascades framework for query optimization," IEEE Data Engineering Bulletin, vol. 18, pp. 19–29, 1995.

[26] C. Graham, "Market share: Relational database management systems by operating system, worldwide, 2005," Gartner Report No: G00141017, May 2006.

[27] J. Gray, "Greetings from a filesystem user," in Proceedings of the FAST '05 Conference on File and Storage Technologies, (San Francisco), December 2005.

[28] J. Gray and B. Fitzgerald, FLASH Disk Opportunity for Server-Applications. http://research.microsoft.com/～Gray/papers/FlashDiskPublic.doc.

[29] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in IFIP Working Conference on Modelling in Data Base Management Systems, pp. 365–394, 1976.

[30] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[31] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and 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, and J. Yang, "Optimizing queries across

diverse data sources," in International Conference on Very Large Databases (VLDB), 1997.

[34] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," ACM Computing Surveys, vol. 15, pp. 287–317, 1983.

[35] S. Harizopoulos and N. Ailamaki, "StagedDB: Designing database servers for modern hardware," IEEE Data Engineering Bulletin, vol. 28, pp. 11–16, June 2005.

[36] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases," in Proceedings of the 32nd Very Large Databases Conference (VLDB), 2006.

[37] J. M. Hellerstein, "Optimization techniques for queries with expensive methods," ACM Transactions on Database Systems (TODS), vol. 23, pp. 113–157, 1998.

[38] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in Proceedings of ACM-SIGMOD International Conference on Management of Data, 1997.

[39] J. M. Hellerstein, J. Naughton, and A. Pfeffer, "Generalized search trees for database system," in Proceedings of Very Large Data Bases Conference (VLDB), 1995.

[40] J. M. Hellerstein and A. Pfeffer, "The russian-doll tree, an index structure for sets," University of Wisconsin Technical Report TR1252, 1994.

[41] C. Hoare, "Monitors: An operating system structuring concept," Communications of the ACM (CACM), vol. 17, pp. 549–557, 1974.

[42] W. Hong and M. Stonebraker, "Optimization of parallel query execution plans in xprs," in Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS), pp. 218–225, Miami Beach, FL, December 1991.

[43] H.-I. Hsiao and D. J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," in Proceedings of Sixth International Conference on Data Engineering (ICDE), pp. 456–465, Los Angeles, CA, November 1990.

[44] Y. E. Ioannidis and Y. Cha Kang, "Randomized algorithms for optimizing large join queries," in Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 312–321, Atlantic City, May 1990.

[45] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in Proceedings of the ACM SIGMOD International Conference on Management

of Data, pp. 268–277, Denver, CO, May 1991.

[46] M. Kornacker, C. Mohan, and J. M. Hellerstein, “Concurrency and recovery in generalized search trees,” in Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 62–72, Tucson, AZ, May 1997.

[47] H. T. Kung and J. T. Robinson, “On optimistic methods for concurrency control,” ACM Tranactions on Database Systems (TODS), vol. 6, pp. 213–226, 1981.

[48] J. R. Larus and M. Parkes, “Using cohort scheduling to enhance server performance,” in USENIX Annual Conference, 2002.

[49] H. C. Lauer and R. M. Needham, “On the duality of operating system structures,” ACM SIGOPS Operating Systems Review, vol. 13, pp. 3–19, April 1979.

[50] P. L. Lehman and S. Bing Yao, “Efficient locking for concurrent operations on b-trees,” ACM Transactions on Database Systems (TODS), vol. 6, pp. 650–670, December 1981.

[51] A. Y. Levy, “Answering queries using views,” VLDB Journal, vol. 10, pp. 270–294, 2001.

[52] A. Y. Levy, I. Singh Mumick, and Y. Sagiv, “Query optimization by predicate move-around,” in Proceedings of 20th International Conference on Very Large Data Bases, pp. 96–107, Santiago, September 1994.

[53] W. Litwin, “Linear hashing: A new tool for file and table addressing,” in Sixth International Conference on Very Large Data Bases (VLDB), pp. 212–223, Montreal, Quebec, Canada, October 1980.

[54] G. M. Lohman, “Grammar-like functional rules for representing query optimization alternatives,” in Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 18–27, Chicago, IL, June 1988.

[55] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton, “Middle-tier database caching for e-business,” in Proceedings of ACM SIGMOD International Conference on Management of Data, 2002.

[56] S. R. Madden and M. J. Franklin, “Fjording the stream: An architecture for queries over streaming sensor data,” in Proceedings of 12th IEEE International Conference on Data Engineering (ICDE), San Jose, February 2002.

[57] V. Markl, G. Lohman, and V. Raman, “Leo: An autonomic query optimizer for db2,”

IBM Systems Journal, vol. 42, pp. 98–106, 2003.

[58] C. Mohan, “Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes,” in 16th International Conference on Very Large Data Bases (VLDB), pp. 392–405, Brisbane, Queensland, Australia, August 1990.

[59] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz, “Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging,” ACM Transactions on Database Systems (TODS), vol. 17, pp. 94–162, 1992.

[60] C. Mohan and F. Levine, “Aries/im: An efficient and high concurrency index management method using write-ahead logging,” in Proceedings of ACM SIGMOD International Conference on Management of Data, (M. Stonebraker, ed.), pp. 371–380, San Diego, CA, June 1992.

[61] C. Mohan, B. G. Lindsay, and R. Obermarck, “Transaction management in the r* distributed database management system,” ACM Transactions on Database Systems (TODS), vol. 11, pp. 378–396, 1986.

[62] E. Nightingale, K. Veerarghavan, P. M. Chen, and J. Flinn, “Rethink the sync,” in Symposium on Operating Systems Design and Implementation (OSDI), November 2006.

[63] OLAP Market Report. Online manuscript. http://www.olapreport.com/market.htm.

[64] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The lru-k page replacement algorithm for database disk buffering,” in Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 297–306, Washington, DC, May 1993.

[65] P. E. O’Neil and D. Quass, “Improved query performance with variant indexes,” in Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 38–49, Tucson, May 1997.

[66] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras, “Multi-dimensional clustering: A new data layout scheme in db2,” in ACM SIGMOD International Management of Data (San Diego, California, June 09–12, 2003) SIGMOD ’03, pp. 637–641, New York, NY: ACM Press, 2003.

[67] D. Patterson, “Latency lags bandwidth,” CACM, vol. 47, pp. 71–75, October 2004.

[68] H. Pirahesh, J. M. Hellerstein, and W. Hasan, “Extensible/rule- based query rewrite optimization in starburst,” in Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 39–48, San Diego, June 1992.

[69] V. Poosala and Y. E. Ioannidis, “Selectivity estimation without the attribute value independence assumption,” in Proceedings of 23rd International Conference on Very Large Data Bases (VLDB), pp. 486–495, Athens, Greece, August 1997.

[70] M. P¨oss, B. Smith, L. Koll´ar, and P.-˚A. Larson, “Tpc-ds, taking decision support benchmarking to the next level,” in SIGMOD 2002, pp. 582–587.

[71] V. Prabakharan, A. C. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Analysis and evolution of journaling file systems,” in Proceedings of USENIX Annual Technical Conference, April 2005.

[72] R. Ramakrishnan and J. Gehrke, “Database management systems,” McGraw-Hill, Boston, MA, Third ed., 2003.

[73] V. Raman and G. Swart, “How to wring a table dry: Entropy compression of relations and querying of compressed relations,” in Proceedings of International Conference on Very Large Data Bases (VLDB), 2006.

[74] D. P. Reed, Naming and Synchronization in a Decentralized Computer System. PhD thesis, MIT, Dept. of Electrical Engineering, 1978.

[75] A. Reiter, “A study of buffer management policies for data management systems,” Technical Summary Report 1619, Mathematics Research Center, University of Wisconsin, Madison, 1976.

[76] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, “System level concurrency control for distributed database systems,” ACM Transactions on Database Systems (TODS), vol. 3, pp. 178–198, June 1978.

[77] S. Sarawagi, S. Thomas, and R. Agrawal, “Integrating mining with relational database systems: Alternatives and implications,” in Proceedings of ACMSIGMOD International Conference on Management of Data, 1998.

[78] R. Sears and E. Brewer, “Statis: Flexible transactional storage,” in Proceedings of Symposium on Operating Systems Design and Implementation (OSDI), 2006.

[79] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, “Access path

selection in a relational database management system," in Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 22–34, Boston, June 1979.

[80] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in Proceedings of 12th IEEE International Conference on Data Engineering (ICDE), New Orleans, February 1996.

[81] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein, "Java support for data-intensive systems: Experiences building the telegraph dataflow system," ACM SIGMOD Record, vol. 30, pp. 103–114, 2001.

[82] L. D. Shapiro, "Exploiting upper and lower bounds in top-down query optimization," International Database Engineering and Application Symposium (IDEAS), 2001.

[83] A. Silberschatz, H. F. Korth, and S. Sudarshan, Database System Concepts. McGraw-Hill, Boston, MA, Fourth ed., 2001.

[84] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," VLDB Journal, vol. 6, pp. 191–208, 1997.

[85] M. Stonebraker, "Retrospection on a database system," ACM Transactions on Database Systems (TODS), vol. 5, pp. 225–240, 1980.

[86] M. Stonebraker, "Operating system support for database management," Communications of the ACM (CACM), vol. 24, pp. 412–418, 1981.

[87] M. Stonebraker, "The case for shared nothing," IEEE Database Engineering Bulletin, vol. 9, pp. 4–9, 1986.

[88] M. Stonebraker, "Inclusion of new types in relational data base systems," ICDE, pp. 262–269, 1986.

[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, and S. Zdonik, "C-store: A column oriented dbms," in Proceedings of the Conference on Very Large Databases (VLDB), 2005.

[90] M. Stonebraker and U. Cetintemel, "One size fits all: An idea whose time has come and gone," in Proceedings of the International Conference on Data Engineering (ICDE), 2005.

[91] Transaction Processing Performance Council 2006. TPC Benchmark C Standard

Specification Revision 5.7, http://www.tpc.org/tpcc/spec/tpcc current. pdf, April.

[92] T. Urhan, M. J. Franklin, and L. Amsaleg, “Cost based query scrambling for initial delays,” ACM-SIGMOD International Conference on Management of Data, 1998.

[93] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, “Capriccio: Scalable threads for internet services,” in Proceedings of the Ninteenth Symposium on Operating System Principles (SOSP-19), Lake George, New York, October 2003.

[94] M. Welsh, D. Culler, and E. Brewer, “Seda: An architecture for well- conditioned, scalable internet services,” in Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-18), Banff, Canada, October 2001.

[95] C. Zou and B. Salzberg, “On-line reorganization of sparsely-populated b+trees,” pp. 115–124, 1996.

附录 1:译者介绍

林子雨(1978－),男,博士,厦门大学计算机科学系助理教授,主要研究领域为数据库,数据仓库,数据挖掘.
主讲课程：《大数据技术基础》
办公地点：厦门大学海韵园科研 2 号楼
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu
数据库实验室网站： http://dblab.xmu.edu.cn