《Architecture of a Database System》

（中文版）

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

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 6 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拓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。

第 6 章 事务:并发控制和恢复

厦门大学计算机科学系教师 林子雨 编著

个人主页：http://www.cs.xmu.edu.cn/linziyu

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

2013 年 9 月

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。

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