第14章
Database System Concepts, 6E

事　务

通常，从数据库用户的观点来看，数据库中一些操作的集合被认为是一个独立单元。比如，从顾客的立场来看，从支票账户到储蓄账户的资金转账是一次单一的操作；而在数据库系统中，这是由几个操作组成的。显然，有一点是最基本的，这些操作要么全都发生，要么由于出错而全不发生。资金从支票账户支出而未转入储蓄账户的情况是不可接受的。

构成单一逻辑工作单元的操作集合称作**事务**(transaction)。即使有故障，数据库系统也必须保证事务的正确执行——要么执行整个事务，要么属于该事务的操作一个也不执行。此外，数据库系统必须以一种能避免引入不一致性的方式来管理事务的并发执行。在资金转账的例子中，一个计算顾客总金额的事务可能在资金转账事务从支票账户支出金额之前查看支票账户余额，而在资金存入储蓄账户之后查看储蓄账户余额。结果，它就会得到不正确的结果。

本章介绍事务处理的基本概念。并发事务处理和故障恢复的细节问题分别在第15章和第16章讨论。事务处理的更进一步话题在第26章讨论。

14.1　事务概念

事务是访问并可能更新各种数据项的一个程序执行**单元**(unit)。事务通常由高级数据操纵语言(代表性的是SQL)或编程语言(例如，C++或Java)通过JDBC或ODBC嵌入式数据库访问书写的用户程序的执行所引起。事务用形如 **begin transaction** 和 **end transaction** 语句(或函数调用)来界定。事务由 **begin transaction** 与(**end transaction** 之间执行的全体操作组成。

这些步骤集合必须作为一个单一的、不可分割的单元出现。因为事务是不可分割的，所以要么执行其全部内容，要么就根本不执行。因此，如果一个事务开始执行，但是由于某些原因失败，则
627 事务对数据库造成的任何可能的修改都要撤销。无论事务本身是否失败(例如，如果它除以零)，或者操作系统崩溃，或者计算机本身停止运行，这项要求都要成立。正如我们将会看到的，确保这个要求是困难的，因为对数据库的一些修改可能仅仅存在事务的主存变量中，而另一些已经写入数据库并存储到磁盘上。这种"全或无"的特性称为**原子性**(atomicity)。

此外，由于事务是一个单一的单元，它的操作不能看起来是被其他不属于该事务的数据库操作分隔开的。尽管我们希望表现在事务的用户级上，但是我们知道事实是有相当大的区别的。即使单条SQL语句也会涉及许多分开的数据库访问，并且一个事务可能会由多条SQL语句构成。因此，数据库系统必须采取特殊处理来确保事务正常执行而不被来自并发执行的数据库语句所干扰。这种特性称为**隔离性**(isolation)。

即使系统能保证一个事务的正确执行，如果此后系统崩溃，结果系统"忘记"了该事务，那么这项工作的意义也不大了。因此，即使崩溃后事务的操作也必须是持久的。这种特性称为**持久性**(durability)。

因为上述三个特性，事务就成了构造与数据库的交互的一种理想方式。这使我们必须加强对事务本身的要求。事务必须保持数据库的一致性——如果一个事务作为原子从一个一致的数据库状态开始独立地运行，则事务结束时数据库也必须再次是一致的。这种一致性要求超出我们此前看到的数据完整性约束(例如主码约束、参照完整性、**check** 约束等)。相反，事务会被期望得更多，以保证太过复杂而不能用SQL构建数据完整性并且依赖于程序的一致性约束。如何实现这个则是编写事

务的程序员的职责。这种特性称为**一致性**(consistency)。

将上述内容更简明地重新描述，我们要求数据库系统维护事务的以下性质。

- **原子性**：事务的所有操作在数据库中要么全部正确反映出来，要么完全不反映。
- **一致性**：隔离执行事务时(换言之，在没有其他事务并发执行的情况下)保持数据库的一致性。
- **隔离性**：尽管多个事务可能并发执行，但系统保证，对于任何一对事务 T_i 和 T_j，在 T_i 看来，T_j 或者在 T_i 开始之前已经完成执行，或者在 T_i 完成之后开始执行。因此，每个事务都感觉不到系统中有其他事务在并发地执行。
- **持久性**：一个事务成功完成后，它对数据库的改变必须是永久的，即使出现系统故障。 628

这些性质通常称为 **ACID 特性**(ACID property)，这一缩写来自 4 条性质的第一个英文字母。

正如我们此后看到的，确保隔离性有可能对系统性能造成较大的不利影响。由于这个原因，一些应用在隔离性上会采取一些妥协。我们将在学习严格执行 ACID 特性后学习这些妥协。

14.2 一个简单的事务模型

因为 SQL 是一种强大而复杂的语言，所以我们采用一种简单的数据库语言来开始学习事务，该语言关注数据何时从磁盘移动到主存以及何时从主存移动到磁盘。这样，我们忽略了 SQL 插入和删除操作，推迟到 15.8 节再去考虑它们。在简单语言中，对数据的实际操作仅限于算术操作。后面，我们会在一个有更丰富的操作集合并且基于 SQL 的真实环境中讨论事务。在简单模型中数据项只包含一个单一的数据值(在例子中是一个数字)。每个数据项由一个名字所标识(在例子中通常是一个字符，例如 A、B、C 等)。

我们将采用一个由几个账户和一个访问和更新账户的事务集合构成的简单的银行应用来阐明事务的概念。事务运用以下两个操作访问数据。

- **read**(X)：从数据库把数据项 X 传送到执行 **read** 操作的事务的主存缓冲区的一个也称为 X 的变量中。
- **write**(X)：从执行 **write** 的事务的主存缓冲区的变量 X 中把数据项 X 传回数据库中。

重要的是要知道一个数据项的变化是只出现在主存中，还是已经写入磁盘上的数据库。在实际数据库系统中，**write** 操作不一定立即更新磁盘上的数据；**write** 操作的结果可以临时存储在某处，以后再写到磁盘上。但是目前我们假设 **write** 操作立即更新数据库。我们将在第 16 章回到这个话题。

设 T_i 是从账户 A 过户 \$50 到账户 B 的事务。这个事务可以定义为：

```
Ti: read(A);
    A := A - 50;
    write(A);
    read(B);
    B := B + 50;
    write(B).
```

629

现在让我们逐个考虑 ACID 特性(为了便于讲解，我们不按 A－C－I－D 的次序来讲述它们)。

- **一致性**：在这里，一致性要求事务的执行不改变 A、B 之和。如果没有一致性要求，金额可能会被事务凭空创造或销毁！容易验证，如果数据库在事务执行前是一致的，那么事务执行后数据库仍将保持一致。

  确保单个事务的一致性是编写该事务的应用程序员的责任。完整性约束的自动检查给这项工作带来了便利，正如我们已经在 4.4 节讨论的。

- **原子性**：假设事务 T_i 执行前账户 A 和账户 B 分别有 \$1000 和 \$2000。现在假设在事务 T_i 执行时系统出现故障，导致 T_i 的执行没有成功完成。我们进一步假设故障发生在 **write**(A)操作执行之后 **write**(B)操作执行之前。在这种情况下，数据库中反映出来的是账户 A 有 \$950，而账户 B 有 \$2000。这次故障导致系统丢失了 \$50。特别地，我们注意到 $A+B$ 的和不再维持原状。

这样，由于故障，系统的状态不再反映数据库本应描述的现实世界的真实状态。我们把这种状态称为**不一致状态**(inconsistent state)。我们必须保证这种不一致性在数据库系统中是不可见的。但是请注意，系统必然会在某一时刻处于不一致状态。即使事务 T_i 能执行完，也仍然存在某一时刻账户 A 的金额是 \$950 而账户 B 的金额是 \$2000，这显然是一个不一致状态。然而这一状态最终会被账户 A 的金额是 \$950 且账户 B 的金额是 \$2050 这个一致的状态代替。这样，如果一个事务或者不开始，或者保证完成，那么这样的不一致状态除了在事务执行当中以外，在其他时刻是不可见的。这就是需要原子性的原因：如果具有原子性，某个事务的所有动作要么在数据库中全部反映出来，要么全部不反映。

保证原子性的基本思路如下：对于事务要执行写操作的数据项，数据库系统在磁盘上记录其旧值。这个信息记录在一个称为**日志**的文件中。如果事务没能完成它的执行，数据库系统从日志中恢复旧值，使得看上去事务从未执行过。14.4 节将进一步讨论这些想法。保证原子性是数据库系统本身的责任；具体来说，这项工作由称作**恢复系统**(recovery system)的一个数据库组件处理，这个将在第 16 章详细讲述。

630

- **持久性**：一旦事务成功地完成执行，并且发起事务的用户已经被告知资金转账已经发生，系统就必须保证任何系统故障都不会引起与这次转账相关的数据丢失。持久性保证一旦事务成功完成，该事务对数据库所做的所有更新就都是持久的，即使事务执行完成后出现系统故障。

  现在我们假设计算机系统的故障将会导致内存中的数据丢失，但已写入磁盘的数据决不会丢失。第 16 章将会讨论预防磁盘上的数据丢失。我们可以通过确保以下两条中的任何一条来达到持久性：

  1. 事务做的更新在事务结束前已经写入磁盘。
  2. 有关事务已执行的更新信息已写到磁盘上，并且此类信息必须充分，能让数据库在系统出现故障后重新启动时重新构造更新。

  第 16 章将介绍的数据库**恢复系统**负责除了保证原子性之外还保证持久性。

- **隔离性**：如果几个事务并发地执行，即使每个事务都能确保一致性和原子性，它们的操作会以人们所不希望的某种方式交叉执行，这也会导致不一致的状态。

  正如我们先前看到的，例如，在 A 至 B 转账事务执行过程中，当 A 中总金额已减去转账额并已写回 A，而 B 中总金额加上转账额后还未写回 B 时，数据库暂时是不一致的。如果另一个并发运行的事务在这个中间时刻读取 A 和 B 的值并计算 $A+B$，它将会得到不一致的值。更进一步，如果第二个事务基于它读取的不一致值对 A 和 B 进行更新，即使两个事务都完成后，数据库仍可能处于不一致状态。

  一种避免事务并发执行而产生问题的途径是串行地执行事务——一个接一个地执行。然而，我们将会在 14.5 节看到，事务并发执行能显著地改善性能。因此人们提出了许多其他的解决方法，它们允许多个事务并发地执行。

  14.5 节讨论事务并发地执行所引起的问题。事务的隔离性确保事务并发执行后的系统状态与这些事务以某种次序一个接一个地执行后的状态是等价的。14.6 节将进一步讨论隔离的原则。确保隔离性是数据库系统中称作**并发控制系统**(concurrency-control system)的部件的责

631

任，这个稍后将在第 15 章讨论。

14.3 存储结构

为了理解如何确保事务的原子性和持久性，我们需要更好地理解数据库中各种数据项如何存储和访问。

在第 10 章中我们看到存储介质可以通过它们的相对速度、容量、故障弹性区分为易失性存储器或非易失性存储器。我们回顾这些概念，并介绍另一种新的存储器，即**稳定性存储器**。

- **易失性存储器**(volatile storage)：易失性存储器中的信息通常在系统崩溃后不会幸存。这种存储器的例子包括主存储器和高速缓冲存储器。易失性存储器的访问非常快，一方面是因为内存访问本身的速度，另一方面是因为可以直接访问易失性存储器中的任何数据项。
- **非易失性存储器**(nonvolatile storage)：非易失性存储器中的信息会在系统崩溃后幸存。非易失性存储器的例子包括用于在线存储的二级存储设备(如磁盘和闪存)，以及用于存档存储的三级存储设备(如光介质和磁带)。根据目前的技术，非易失性存储器比易失性存储器慢，特别是对于随机访问。然而，二级存储设备和三级存储设备容易受到故障的影响，导致信息丢失。
- **稳定性存储器**(stable storage)：稳定性存储器中的信息永远不会丢失。(应该对永远持有怀疑态度，因为理论上永远不能保证。例如，尽管可能性很小，也有可能出现黑洞吞噬地球从而永久地销毁所有数据!)尽管稳定性存储器在理论上不可能获得，但是可以通过技术近似使得数据丢失的可能性微乎其微。为了实现稳定性存储器，我们可以复制几种非易失性存储器介质(通常是磁盘)中的信息，并且采用独立故障模式。更新必须很小心以保证更新过程中稳定性存储器的故障不会导致信息丢失。16.2.1节将讨论稳定性存储器的实现。

这几种不同存储器类型之间的区别在实际中没有我们介绍得这么明显。例如，某些系统提供备用电池使得一些主存可以在系统崩溃或电源故障中幸存下来，例如某些RAID控制器。

为了一个事务能够持久，它的修改应该写入稳定性存储器。同样，为了一个事务是原子的，日志记录需要在对磁盘上的数据库做任何改变之前写入稳定性存储器。显然，一个系统保证的持久性和原子性的程度取决于稳定性存储器的实现到底有多稳定。在某些情况下，磁盘的一个单一拷贝是足够的，但是对于其数据非常有价值和事务非常重要的应用程序需要多个拷贝，或者换句话说，更接近于理想化的稳定性存储器。 632

14.4 事务原子性和持久性

正如我们先前所注意到的，事务并非总能成功地执行完成。这种事务称为**中止**(aborted)了。我们如果要确保原子性，中止事务必须对数据库的状态不造成影响。因此，中止事务对数据库所做过的任何改变必须撤销。一旦中止事务造成的变更被撤销，我们就说事务**已回滚**(rolled back)。恢复机制负责管理事务中止。典型的方法是维护一个**日志**(log)。每个事务对数据库的修改都首先会记录到日志中。我们记录执行修改的事务标识符、修改的数据项标识符以及数据项的旧值(修改前的)和新值(修改后的)。然后数据库才会修改。维护日志提供了重做修改以保证原子性和持久性的可能，以及撤销修改以保证在事务执行发生故障时的原子性的可能。第16章将会详细讨论基于日志的故障恢复。

成功完成执行的事务称为**已提交**(committed)。一个对数据库进行过更新的已提交事务使数据库进入一个新的一致状态，即使出现系统故障，这个状态也必须保持。

一旦事务已提交，我们不能通过中止它来撤销其造成的影响。撤销已提交事务所造成影响的唯一方法是执行一个**补偿事务**(compensating transaction)。例如，如果一个事务给一个账户加上了$20，其补偿事务应当从该账户减去$20。然而，我们不总是能够创建这样的补偿事务。因此，书写和执行一个补偿事务的责任就留给了用户，而不是通过数据库系统来处理。第26章包含对补偿事务的讨论。

我们需要更准确地定义一个事务成功完成的含义。为此我们建立了一个简单的抽象事务模型。事务必须处于以下状态之一。

- **活动的**(active)：初始状态，事务执行时处于这个状态。
- **部分提交的**(partially committed)：最后一条语句执行后。
- **失败的**(failed)：发现正常的执行不能继续后。
- **中止的**(aborted)：事务回滚并且数据库已恢复到事务开始执行前的状态后。
- **提交的**(committed)：成功完成后。

事务相应的状态图如图14-1所示。只有在事务已进入提交状态后，我们才说事务已提交。类似 633

地，仅当事务已进入中止状态，我们才说事务已中止。如果事务是提交的或中止的，它称为**已经结束的**(terminated)。

事务从活动状态开始。当事务完成它的最后一条语句后就进入了部分提交状态。此刻，事务已经完成执行，但由于实际输出可能仍临时驻留在主存中，因此一个硬件故障可能阻止其成功完成，于是事务仍有可能不得不中止。

接着数据库系统往磁盘上写入足够的信息，确保即使出现故障时事务所做的更新也能在系统重启后重新创建。当最后一条这样的信息写完后，事务就进入提交状态。

正如先前所提到的，我们现在假设故障不会引起磁盘上的数据丢失。磁盘上数据丢失的处理技术将在第 16 章讨论。

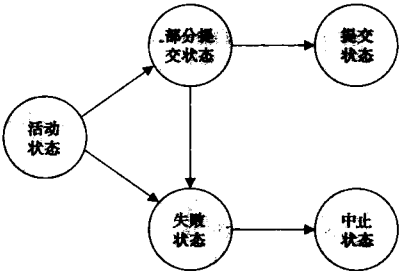

图 14-1 事务状态图

系统判定事务不能继续正常执行后(例如，由于硬件或逻辑错误)，事务就进入失败状态。这种事务必须回滚。这样，事务就进入中止状态。此刻，系统有两种选择。

- 它可以**重启**(restart)事务，但仅当引起事务中止的是硬件错误或不是由事务的内部逻辑所产生的软件错误时。重启的事务被看成是一个新事务。
- 它可以**杀死**(kill)事务，这样做通常是由于事务的内部逻辑造成的错误，只有重写应用程序才能改正，或者由于输入错误，或所需数据在数据库中没有找到。

在处理**可见的外部写**(observable external write)，比如写到用户屏幕，或者发送电子邮件时，我们
634 必须要小心。由于写的结果可能已经在数据库系统之外看到，因此一旦发生这种写操作，就不能再抹去。大多数系统只允许这种写操作在事务进入提交状态后发生。实现这种模式的一种方法是在非易失性存储设备中临时写下与外部写相关的所有数据，然后在事务进入提交状态后再执行真正的写操作。如果在事务已进入提交状态而外部写操作尚未完成之时，系统出现了故障，数据库系统就可以在重启后(用存储在非易失性设备中的数据)执行外部写操作。

在某些情况下处理外部写操作会更复杂，例如，我们假设外部动作是在自动取款机上支付现金，并且系统恰好在支付现金之前发生故障(我们假定现金能自动支付)，当系统重新启动时再执行现金支付将毫无意义，因为用户可能已经离开。在这种情况下，重新启动时系统应该执行一个补偿事务，比如将现金存回用户的账户。

作为另一个例子，考虑一个用户在 Web 上进行预订。很可能在预订事务刚刚提交后数据库系统或应用服务器发生崩溃，也有可能在预订事务刚刚提交后用户的网络连接丢失。在上述任意一种情况下，即使事务已经提交，外部写也并没有发生。为了处理这种情况，应用程序必须设计成当用户再次连接到 Web 应用程序时，她可以看到她的事务是否成功。

对于某些特定应用，允许处于活动状态的事务向用户显示数据也许是我们所期望的，特别对将运行几分钟或几小时的长周期事务来说。遗憾的是，除非牺牲事务原子性，否则我们不能允许这种可见的数据输出。第 26 章会讨论另外的事务模型，它们支持交互式长周期事务。

14.5 事务隔离性

事务处理系统通常允许多个事务并发地执行。正如我们先前看到的，允许多个事务并发更新数据引起许多数据一致性的复杂问题。在存在事务并发执行的情况下保证一致性需进行额外工作；如果我们强制事务**串行地**(serially)执行将简单得多——一次执行一个事务，每个事务仅当前一事务执行完后才开始。然而，有两条很好的理由允许并发。

- **提高吞吐量和资源利用率**。一个事务由多个步骤组成。一些涉及 I/O 活动；还有一些涉及 CPU
635 活动。在计算机系统中 CPU 与磁盘可以并行运作。因此，I/O 活动可以与 CPU 处理并行进行。

利用 CPU 与 I/O 系统的并行性，多个事务可并行执行。当一个事务在一张磁盘上进行读写时，另一个事务可在 CPU 上运行，第三个事务又可在另一张磁盘上进行读写。所有这些技术增加了系统的**吞吐量**(throughput)——即给定时间内执行的事务数增加。相应地，处理器与磁盘**利用率**(utilization)也提高；换句话说，处理器与磁盘空闲或者没有做有用的工作的时间较少。

- **减少等待时间**。系统中可能运行着各种各样的事务，一些较短，一些较长。如果事务串行地执行，短事务可能得等待它前面的长事务完成，这可能导致难以预测的延迟。如果各事务针对数据库的不同部分进行操作，让它们并发地执行会更好，它们之间可以共享 CPU 周期与磁盘存取。并发执行可以减少执行事务时不可预测的延迟。此外，也可减少**平均响应时间**(average response time)：即一个事务从提交到完成所需的平均时间。

在数据库中使用并发执行的动机在本质上与操作系统中使用**多道程序**(multiprogramming)的动机是一样的。

当多个事务并发地执行时，可能违背隔离性，这导致即使每个事务都正确执行，数据库的一致性也可能被破坏。这一节将讲述调度的概念，以帮助读者识别哪些是可以保证一致性的执行序列。

数据库系统必须控制事务之间的交互，以防止它们破坏数据库的一致性。系统通过称为**并发控制机制**(concurrency-control scheme)的一系列机制来保证这一点。第 15 章将研究并发控制机制，现在我们集中考虑正确的并发执行这一概念。

我们再来看看 14.1 节中的简化银行系统，其中有多个账户以及存取、更新这些账户的一组事务。设 T_1、T_2 是将资金从一个账户转移到另一个账户的两个事务，事务 T_1 是从账户 A 过户 \$50 到账户 B 的事务，它定义为：

T_1: **read**(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

636

事务 T_2 是从账户 A 将存款余额的 10% 过户到账户 B 的事务，它定义为：

T_2: **read**(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

并发性趋势

当前计算领域的一些发展趋势带来大量可能的并发性。数据库系统利用并发性提高系统的整体性能，使得并发运行的事务数量可能越来越多。

早期的计算机只有一个处理器。因此，计算机中没有真正意义的并发性。唯一表现出的并发性是操作系统使几个不同的任务或进程共享处理器。现代计算机可能有很多个处理器，这将使得一个计算机中有真正不同的进程。然而，即使是一个处理器，如果它有多核，也能够同时运行多个进程。英特尔酷睿双核处理器就是一个众所周知的多核处理器例子。

数据库系统有两种方法利用多处理器和多核的优势。一种是发现单个事务或查询内的并行性，另一种是支持大量并发的事务。

许多服务提供商现在采用一批计算机而不是大型主机来提供服务。他们做出这样的选择是基于这种方案的低成本。这样的结果是带来了更大限度的并发性支持。

文献注解介绍了计算机架构和并行计算的进展。第 18 章将介绍利用多处理器和多核构建并行数据库系统的方法。

假设账户 A 和账户 B 当前的值分别是 $1000 和 $2000。假设两个事务一个一个地执行，先是 T_1，然后是 T_2。该执行顺序如图 14-2 所示。在图 14-2 中，指令序列自顶向下按时间顺序排列，T_1 的指令出现在左栏，T_2 的指令出现在右栏。按图 14-2 的顺序执行后，账户 A 与 B 中最终的值分别为 $855 与 $2145。因此，账户 A 与 B 的资金总数(即 $A+B$)在两个事务执行后保持不变。

637

类似地，如果事务一个一个地执行，先是 T_2，然后是 T_1，那么相应的执行顺序如图 14-3 所示。同样，正如所预期的，$A+B$ 之和仍维持不变。账户 A 与 B 中最终的值分别为 $850 与 $2150。

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

图 14-2 调度 1：一个串行调度，T_2 跟在 T_1 之后

前面所描述的执行顺序称为**调度**(schedule)。它们表示指令在系统中执行的时间顺序。显然，一组事务的一个调度必须包含这一组事务的全部指令，并且必须保持指令在各个事务中出现的顺序。例如，在任何一个有效的调度中，事务 T_1 中指令 **write**(A)必须在指令 **read**(B)之前出现。请注意，我们在调度中包括了 **commit** 操作来表示事务已经进入提交状态。在下面的讨论中，我们将称第一种执行顺序为调度 1(T_2 跟在 T_1 之后)，称第二种执行顺序为调度 2(T_1 跟在 T_2 之后)。

这两个调度是**串行的**(serial)。每个串行调度由来自各事务的指令序列组成，其中属于同一事务的指令在调度中紧挨在一起。回顾组合数学中一个众所周知的公式，我们知道，对于有 n 个事务的事务组，共有 $n!$ 个不同的有效串行调度。

当数据库系统并发地执行多个事务时，相应的调度不必是串行的。若有两个并发执行的事务，操作系统可能先选其中的一个事务执行一小段时间，然后切换上下文，执行第二个事务一段时间，接着又切换回第一个事务执行一段时间，如此下去。在多个事务的情形下，所有事务共享 CPU 时间。

多种执行顺序是有可能的，因为来自两个事务的各条指令可能是交叉执行的。一般而言，在 CPU 切换到另一事务之前准确预测 CPU 将执行某个事务的多少条指令是不可能的。⊖

回到前面的例子，假设两个事务并发执行。一种可能的调度如图 14-4 所示，当它执行完成后，我们到达的状态与先执行 T_1 后执行 T_2 的串行调度一样，$A+B$ 之和保持不变。

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

图 14-3 调度 2：一个串行调度，T_1 跟在 T_2 之后

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

图 14-4 调度 3：等价于调度 1 的一个并发调度

不是所有的并发执行都能得到正确的结果。举个例子，考虑如图 14-5 所示的调度，该调度执行后，到达的状态是账户 A 与 B 中最终的值分别为 $950 与 $2100。这个最终状态是一个不一致状态，因

⊖ 包含 n 个事务的事务组的可能调度数量非常大。有 $n!$ 个不同的串行调度。考虑所有的事务的步骤可以交错的方式，调度的总数要比 $n!$ 大得多。

为在并发执行过程中多出了 $50。实际上，两个事务执行后 $A + B$ 之和未能保持不变。

如果并发执行的控制完全由操作系统负责，许多调度都是可能的，包括像上述调度那样使数据库处于不一致状态的调度。保证所执行的任何调度都能使数据库处于一致状态，这是数据库系统的任务，数据库系统中负责完成此任务的是**并发控制**(concurrency-control)部件。

在并发执行中，通过保证所执行的任何调度的效果都与没有并发执行的调度效果一样，我们可以确保数据库的一致性。也就是说，调度应该在某种意义上等价于一个串行调度。这种调度称为**可串行化**(serializable)调度。

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

图 14-5 调度 4：一个导致不一致状态的并发调度

14.6 可串行化

在我们考虑数据库系统并发控制部件如何保证串行化之前，我们考虑如何确定一个调度是可串行化的。显然，串行调度是可串行化的，但是如果许多事务的步骤交错执行，则很难确定一个调度是否是可串行化的。由于事务就是程序，因此要确定一个事务有哪些操作、多个事务的操作如何相互作用是有困难的。由于这个缘故，我们将不会考虑一个事务对某一数据项可执行的各种不同类型的操作，而只考虑两种操作：**read** 和 **write**。我们这样假设，在数据项 Q 上的 **read**(Q) 和 **write**(Q) 指令之间，事务可以对驻留在事务局部缓冲区中的 Q 的拷贝执行任意操作序列。按这种模式，从调度的角度来看，事务唯一重要的操作是 **read** 与 **write** 指令。commit 操作尽管相关，但是我们到 14.7 节才考虑它。因此，我们在调度中通常只显示 **read** 与 **write** 指令，正如图 14-6 所示调度 3 中所表示的那样。

本节讨论不同形式的等价调度，但是重点关注其中一种称为**冲突可串行化**(conflict serializability)的形式。

我们考虑一个调度 S，其中含有分别属于 I 与 J 的两条连续指令 I_i 与 I_j ($i \neq j$)。如果 I 与 J 引用不同的数据项，则交换 I 与 J 不会影响调度中任何指令的结果。然而，若 I 与 J 引用相同的数据项 Q，则两者的顺序是重要的。由于我们只处理 **read** 与 **write** 指令，因此需考虑以下 4 种情形：

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

图 14-6 调度 3：只显示 read 与 write 指令

1. I = **read**(Q)，J = **read**(Q)。I 与 J 的次序无关紧要，因为不论其次序如何，T_i 与 T_j 读取的 Q 值总是相同的。

2. I = **read**(Q)，J = **write**(Q)。若 I 先于 J，则 T_i 不会读取由 T_j 的指令 J 写入的 Q 值；若 J 先于 I，则 T_i 读到由 T_j 写入的 Q 值。因此 I 与 J 的次序是重要的。

638
|
641

3. I = **write**(Q)，J = **read**(Q)。I 与 J 的次序是重要的，其原因类似前一情形。

4. I = **write**(Q)，J = **write**(Q)。由于两条指令均为 **write** 指令，因此指令的顺序对 T_i 与 T_j 没有什么影响。然而，调度 S 的下一条 **read**(Q)指令读取的值将受到影响，因为数据库里只保留了两条 **write** 指令中后一条的结果。如果在调度 S 的指令 I 与 J 之后没有其他的 write(Q)指令，则 I 与 J 的顺序直接影响调度 S 产生的数据库状态中 Q 的最终值。

因此，只有在 I 与 J 全为 **read** 指令时，两条指令的执行顺序才是无关紧要的。

当 I 与 J 是不同事务在相同的数据项上的操作，并且其中至少有一个是 **write** 指令时，我们说 I 与 J **是冲突**(conflict)的。

为了说明冲突指令的概念，我们考虑图 14-6 中的调度 3。T_1 的 **write**(A)指令与 T_2 的 **read**(A)指令相冲突。然而，T_2 的 **write**(A)指令与 T_1 的 **read**(B)指令不冲突，因为两条指令访问不同的数据项。

设 I 与 J 是调度 S 的两条连续指令。若 I 与 J 是属于不同事务的指令且不冲突，则可以交换 I 与 J 的顺序得到一个新的调度 S'。S 与 S'等价，因为除了 I 与 J 外，其他指令的次序与原来相同，而 I 与 J 的顺序无关紧要。

在图 14-6 的调度 3 中，由于 T_2 的 **write**(A)指令与 T_1 的 **read**(B)指令不冲突，因此可以交换这些指令得到一个等价的调度——图 14-7 所示的调度 5。不管系统初始状态如何，调度 3 与调度 5 均得到相同的最终系统状态。

我们继续交换非冲突指令如下：

- 交换 T_1 的 **read**(B)指令与 T_2 的 **read**(A)指令。
- 交换 T_1 的 **write**(B)指令与 T_2 的 **write**(A)指令。
- 交换 T_1 的 **write**(B)指令与 T_2 的 **read**(A)指令。

经过以上交换的结果是一个串行调度，即图 14-8 所示的调度 6。注意调度 6 和调度 1 完全一样，但是后者只显示了 **read** 和 **write** 指令。这样，我们说明了调度 3 等价于一个串行调度。该等价性意味着不管初始系统状态如何，调度 3 将与某个串行调度产生相同的最终状态。

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

图 14-7 调度 5：交换调度 3 的一对指令得到的调度

如果调度 S 可以经过一系列非冲突指令交换转换成 S'，我们称 S 与 S'是**冲突等价**(conflict equivalent)的。[⊖]

不是所有的串行调度相互之间都冲突等价。例如，调度 1 和调度 2 不是冲突等价的。

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

图 14-8 调度 6：与调度 3 等价的一个串行调度

由冲突等价的概念引出了冲突可串行化的概念：若一个调度 S 与一个串行调度冲突等价，则称调度 S 是**冲突可串行化**(conflict serializable)的。那么，因为调度 3 冲突等价于串行调度 1，所以调度 3 是冲突可串行化的。

最后，考虑图 14-9 所示的调度 7；该调度仅包含事务 T_3 与 T_4 的重要操作(即 read 与 write)。这个调度不是冲突可串行化的，因为它既不等价于串行调度 $< T_3, T_4 >$，也不等价于串行调度 $< T_4, T_3 >$。

为确定一个调度是否冲突可串行化，我们这里给出了一个简单有效的方法。设 S 是一个调度，我们由 S 构造一个有向图，称为**优先图**(precedence graph)。该图由两部分组成 $G = (V, E)$，其中 V 是顶点集，E 是边集，顶点集由所有参与调度的事务组成，边集由满足下列三个条件之一的边 $T_i \rightarrow T_j$ 组成：

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

图 14-9 调度 7

- 在 T_j 执行 **read**(Q)之前，T_i 执行 **write**(Q)。
- 在 T_j 执行 **write**(Q)之前，T_i 执行 **read**(Q)。
- 在 T_j 执行 **write**(Q)之前，T_i 执行 **write**(Q)。

如果优先图中存在边 $T_i \rightarrow T_j$，则在任何等价于 S 的串行调度 S' 中，T_i 必出现在 T_j 之前。

a) b)

图 14-10 a)调度 1 与 b)调度 2 的优先图

例如，调度 1 的优先图如图 14-10a 所示，图 14-10a 中只有一条边 $T_1 \rightarrow T_2$，因为 T_1 的所有指令均在 T_2 的首条指令之前执行。类似地，图 14-10b 表示的是调度 2 的优先图，该图仅含一条边 $T_2 \rightarrow T_1$，因为 T_2 的所有指令均在 T_1 的首条指令之前执行。

调度 4 的优先图如图 14-11 所示。因为 T_i 执行 **read**(A)先于 T_2 执行 **write**(A)，所以图 14-11 含有边 $T_1 \rightarrow T_2$。又因 T_2 执行 **read**(B)先于 T_1 执行 **write**(B)，所以图 14-11 还含有边 $T_2 \rightarrow T_1$。

如果调度 S 的优先图有环，则调度 S 是非冲突可串行化的，如果优先图无环，则调度 S 是冲突可串行化的。

⊖ 们用冲突等价这个术语把刚刚定义的等价和本节后面将介绍的其他定义区别开。

串行化顺序(serializability order)可通过**拓扑排序**(topological sorting)得到，拓扑排序用于计算与优先图的偏序相一致的线性顺序。一般而言，通过拓扑排序可以获得多个线性顺序。例如，图14-12a有两种可接受的线性顺序，如图14-12b与图14-12c所示。

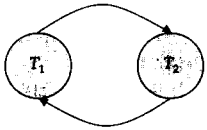

图14-11 调度4的优先图

因此，要判定冲突可串行化，需要构造优先图并调用一个环检测算法。环检测算法可在标准的算法课本中找到。环检测算法，例如那些基于深度优先搜索的环检测算法，需要n^2数量级的运算，其中n是优先图中顶点数(即事务数)。

回到前面的例子，注意到调度1与调度2的优先图(见图14-10)的确不包含环。而调度4的优先图(见图14-11)却有一个环，说明该调度不是冲突可串行化的。

有可能存在两个调度，它们产生相同的结果，但它们不是冲突等价的。例如，考虑事务T_3，它从账户B过户$10到账户$A$。设调度8如图14-13所示。我们说调度8不与串行调度$< T_1, T_3 >$冲突等价，因为在调度8中，T_3的**write**(B)指令与T_1的**read**(B)指令冲突。这在优先图中产生了一条$T_3 \rightarrow T_1$的边。类似地，我们看到T_1的**write**(A)指令与T_3的**read**指令冲突，产生了一条$T_1 \rightarrow T_3$的边。这表示优先图有环，即调度8不是可串行化的。然而，执行调度8或串行调度$< T1, T_3 >$后，账户A与B中最终的值是相同的，即分别为$960与$2040。

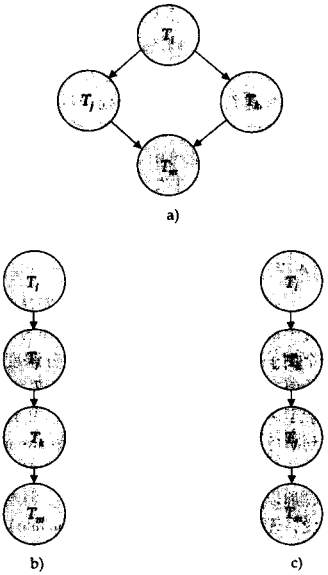

图14-12 拓扑排序示例

从这个例子可以看出，存在比冲突等价定义限制松一些的调度等价定义。对于系统来说，要确定调度8与串行调度$< T_1, T_3 >$产生的结果相同，系统必须分析T_1与T_3所进行的计算，而不只是分析其**read**和**write**操作。通常，这种分析难于实现并且计算代价很大。在该例子中，最后的结果和串行调度是一样的，因为从数学的角度递增和递减是可交换的。虽然这在该例子中比较简单，但是一般情况下并非如此，因为一个事务可能会表示为一条复杂的SQL语句，或一个有JDBC调用的Java程序等。

不过，存在一些别的纯粹基于**read**与**write**操作的调度等价定义。其中一个是*视图等价*，并且引出*视图可串行化*的概念。视图可串行化因为其计算的高度复杂性在实际中并不常用。[⊖]因此，我们推迟到第15章中再讨论视图可串行化，但是为了表述完整起见，这儿请注意，调度8的例子不是视图可串行化的。

T_1	T_3
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

图14-13 调度8

⊖ 判定调度是否视图可串行化的问题已被证明是属于NP-完全问题，因此几乎肯定不存在有效的判定视图可串行化的算法。

14.7 事务隔离性和原子性

642 ~ 646

至此，我们在隐含地假定无事务故障的前提下学习了调度。现在我们讨论在并发执行过程中事务故障所产生的影响。

不论什么原因，如果事务 T_i 失败了，我们必须撤销该事务的影响以确保其原子性。在允许并发执行的系统中，原子性要求依赖于 T_i 的任何事务 T_j（即 T_j 读取了 T_i 写的数据）也中止。为确保这一点，我们需要对系统所允许的调度类型做一些限制。

在下面两节中，我们从事务故障恢复的角度讲述什么样的调度是可接受的，第 15 章将讲述如何保证只产生这种可接受的调度。

14.7.1 可恢复调度

考虑图 14-14 所示的部分调度 9，其中事务 T_7 只执行一条指令：**read**(A)。我们称之为**部分调度**（partial schedule），因为在 T_6 中没有包括 **commit** 或 **abort** 操作。注意 T_7 执行 **read**(A) 指令后立即提交。因此 T_7 提交时 T_6 仍处于活跃状态。现假定 T_6 在提交前发生故障。T_7 已读取了由 T_6 写入的数据项 A 的值。因此，我们说 T_7 **依赖**（dependent）于 T_6。因此，我们必须中止 T_7 以保证事务的原子性。但 T_7 已提交，不能再中止。这样就出现了 T_6 发生故障后不能正确恢复的情形。

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

图 14-14 调度 9：一个可恢复的调度

调度 9 是不可恢复调度的一个例子。一个**可恢复调度**（recoverable schedule）应满足：对于每对事务 T_i 和 T_j，如果 T_j 读取了之前由 T_i 所写的数据项，则 T_i 先于 T_j 提交。例如，如果要使调度 9 是可恢复的，则 T_7 应该推迟到 T_6 提交后再提交。

14.7.2 无级联调度

即使一个调度是可恢复的，要从事务 T_i 的故障中正确恢复，可能需要回滚若干事务。当其他事务读取了由事务 T_i 所写数据项时就会发生这种情形。举一个例子，考虑图14-15所示的部分调度。事务 T_8 写入 A 的值，事务 T_9 读取了 A 的值。事务 T_9 写入 A 的值，事务 T_{10} 读取了 A 的值。假定此时事务 T_8 失败，T_8 必须回滚。由于 T_9 依
647 赖于 T_8，因此事务 T_9 必须回滚。由于 T_{10} 依赖于 T_9，因此 T_{10} 必须回滚。这种因单个事务故障导致一系列事务回滚的现象称为级联回滚（cascading rollback）。

级联回滚导致撤销大量工作，是我们不希望发生的。我们希望对调度加以限制，避免级联回滚发生。这样的调度称为**无级联调度**。规范地说，**无级联调度**（cascadeless schedule）应满足：对于每对事务 T_i 和 T_j，如果 T_j 读取了先前由 T_i 所写的数据项，则 T_i 必须在 T_j 这一读操作前提交。容易验证每一个无级联调度也都是可恢复的调度。

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

图 14-15 调度 10

14.8 事务隔离性级别

可串行性是一个有用的概念，因为当程序员对事务编码时，它允许程序员忽略与并发性相关的问题。如果事务在独立执行时保证数据库一致性，那么可串行性就能确保并发执行时也具有一致性。然而，对于某些应用，保证可串行性的那些协议可能只允许极小的并发度。在这种情况下，我们采用较弱级别的一致性。为了保证数据库的正确性，使用较弱级别一致性给程序员增加了额外负担。

SQL 标准也允许一个事务这样规定：它可以以一种与其他事务不可串行化的方式执行。例如，一个事务可能在**未提交读**级别上操作，这里允许事务读取甚至还未提交的记录。SQL 为那些不要求精确结果的长事务提供这种特征。如果这些事务要在可串行化的方式下执行，它们就会干扰其他事务，造成其他事务执行的延迟。

SQL 标准规定的隔离性级别如下。

648

- **可串行化**（serializable）：通常保证可串行化调度。然而，正如我们将要解释的，一些数据库系统对该隔离性级别的实现在某些情况下允许非可串行化执行。

- **可重复读**(repeatable read)：只允许读取已提交数据，而且在一个事务两次读取一个数据项期间，其他事务不得更新该数据。但该事务不要求与其他事务可串行化。例如：当一个事务在查找满足某些条件的数据时，它可能找到一个已提交事务插入的一些数据，但可能找不到该事务插入的其他数据。
- **已提交读**(read committed)：只允许读取已提交数据，但不要求可重复读。比如，在事务两次读取一个数据项期间，另一个事务更新了该数据并提交。
- **未提交读**(read uncommitted)：允许读取未提交数据。这是 SQL 允许的最低一致性级别。

以上所有隔离性级别都不允许**脏写**(dirty write)，即如果一个数据项已经被另外一个尚未提交或中止的事务写入，则不允许对该数据项执行写操作。

许多数据库系统运行时的默认隔离性级别是已提交读。在 SQL 中，除了接受系统的默认设置，还可以显式地设置隔离性级别。例如，语句"**set transaction isolation level serializable;**"将隔离性级别设置为可串行化，其他隔离性级别可类似设定。Oracle、PostgreSQL 和 SQL Server 均支持上述语法。DB2 使用语法"**change isolation level**"以及其自身提供的隔离性级别缩写。

修改隔离性级别必须作为事务的第一条语句执行。此外，如果单条语句的自动提交默认打开，则必须关闭；可以采用 API 函数来做这件事，例如我们在 5.1.1.7 节中看到的 JDBC 方法 **Connection. setAutoCommit**(**false**)。此外，在 JDBC 方法中，**Connection. setTransactionIsolation** (**int level**)可以用来设置隔离性级别。更多细节请参阅 JDBC 手册。

一个应用程序设计者可能会为了提高系统性能而接受较弱的隔离性级别。正如我们将在 14.9 节和第 15 章所看到的，确保可串行化可能会迫使一个事务等待另一个事务，或者在某些情况下，由于该事务无法作为可串行化执行的一部分而被迫中止。虽然为了性能可能会带来短暂的数据库不一致风险，但是如果我们能确保这种不一致性是和应用程序无关的，则这种权衡是合理的。

实现隔离性级别有很多方法。只要实现确保可串行化，数据库应用程序的设计者或者应用程序的用户就不需要知道这些实现的细节，除非需要处理性能问题。遗憾的是，尽管隔离性级别设置为**可串行化**，但一些数据库系统实际上采用了较弱的隔离性级别来实现，这并不排除所有非可串行化的可能性。我们将在 14.9 节再次讨论这个问题。如果无论显式地或隐式采用较弱的隔离性级别，则应用程序的设计者必须知道一些实现细节，从而避免或者最小化由于缺乏可串行化保证所带来的不一致的可能性。 649

现实世界中的可串行化

可串行化调度是保证一致性的理想方法，但是在日常工作中，我们无需如此严格的要求。一个提供商品销售的网站可能会列出某个特售商品，当一个用户选择该商品并且在结账过程中，该商品是不可用的。从数据库的角度来看，这就是一个不可重复读。

作为另一个例子，考虑在航空旅行中选择座位。假设一个旅客已经预订好行程，并且正在为每次航班选择座位。许多航空公司的网站允许用户查看不同的航班来选择一个座位，然后要求用户确认该选择。而其他旅客也可能同时正在选择同样航班的座位或者更改选择的座位。因此，旅客看到的空余座位事实上是变化的，但是旅客所看到的只是当他开始座位选择流程时空余座位的一个快照。

即使两个旅客同时选择座位，他们很可能选择不同的座位，也不会发生冲突。然而，事务是非可串行化的，由于一个旅客读取的数据是此前其他旅客重新的，会导致优先图中的一个环。如果两个旅客同时选择了一个座位，其中的一个则不会获得他所选择的座位。不过，这种情况很容易解决，只要将更新的空余座位信息让一个或者多个旅客再次选择即可。

通过强制一个特别只允许一个用户选择某一个航班的座位，可保证可串行化。然而，这样做可能会带来很显著的延迟，因为旅客需要等待他的航班变成可以选择座位。特别地，如果一个旅客花费很长时间来选择一个座位，则会给其他旅客带来严重问题。另外的做法是，这类事务通常可以分成一个需要用户交互的部分以及一个专门在数据库上运行的部分。在上述例子中，数据库事务将检查旅客选中的座位是否仍然空闲，如果是，则更新数据库中的座位选择信息。只有在没有用户交互的情况下，数据库上运行的事务才能保证可串行化。

14.9 隔离性级别的实现

至此，我们已经知道一个调度应具有什么样的特性才能保证数据库处于一致状态，并保证事务故障得以安全地处理。

我们可以使用多种**并发控制机制**(concurrency-control scheme)来保证，即使在有多个事务并发执行
650 时，不管操作系统在事务之间如何分时共享资源(如 CPU 时间)，都只产生可接受的调度。

举一个并发控制机制的简单例子，考虑如下机制：一个事务在开始前获得整个数据库的**锁**(lock)，并在它提交之后释放这个锁。当一个事务持有锁时，其他事务就不允许获得这个锁，因此必须等待锁释放。由于采用了封锁策略，因此一次只能执行一个事务。所以只会产生串行调度。这样的调度很明显是可串行化的，并且容易证明也是可恢复的和无级联的。

这样的并发控制机制的性能低下，因为它迫使事务等到前面的事务结束后才能开始。换句话说，这种机制提供的并发程度很低。正如我们在 14.5 节中看到的那样，并发执行有诸多性能方面的益处。

并发控制机制的目的是获得高度的并发性，同时保证所产生的调度是冲突可串行化或视图可串行化的、可恢复的，并且是无级联的。

下面大致介绍一些重要的并发控制机制是如何工作的，然后第 15 章会介绍更多细节。

14.9.1 锁

一个事务可以封锁其访问的数据项，而不用封锁整个数据库。在这种策略下，事务必须在足够长的时间内持有锁来保证可串行化，但是这一周期又要足够短致使不会过度影响性能。对于我们将在 14.10 节中看到的数据项的访问依赖于一条 **where** 子句的 SQL 语句则情况更为复杂。第 15 章将介绍一个简单并且广泛用来确保可串行化的两阶段封锁协议。简单地说，两阶段封锁要求一个事务有两个阶段，一个阶段只获得锁但不释放锁，第二个阶段只释放锁但是不获得锁。(实际上，通常只有当事务完成所有操作并且提交或中止时才释放锁。)

如果我们有两种锁，则封锁的结果将进一步得到改进：共享的和排他的。共享锁用于事务读的数据项，而排他锁用于事务写的数据项。许多事务可以同时持有一个数据项上的共享锁，但是只有当其他事务在一个数据项上不持有任何锁(无论共享锁或排他锁)时，一个事务才允许持有该数据项上的排他锁。这两种锁模式以及两阶段封锁协议在保证可串行化的前提下允许数据的并发读。

14.9.2 时间戳

另一类用来实现隔离性的技术为每个事务分配一个**时间戳**(timestamp)，通常是当它开始的时候。对于每个数据项，系统维护两个时间戳。数据项的读时间戳记录读该数据项的事务的最大(即最近的)
651 时间戳。数据项的写时间戳记录写入该数据项当前值的事务的时间戳。时间戳用来确保在访问冲突情况下，事务按照事务时间戳的顺序来访问数据项。当不可能访问时，违例事务将会中止，并且分配一个新的时间戳重新开始。

14.9.3 多版本和快照隔离

通过维护数据项的多个版本，一个事务允许读取一个旧版本的数据项，而不是被另一个未提交或者在串行化序列中应该排在后面的事务写入的新版本的数据项。有许多多版本并发控制技术。其中一个是实际中广泛应用的称为**快照隔离**(snapshot isolation)的技术。

在快照隔离中，我们可以想象每个事务开始时有其自身的数据库版本或者快照。[⊖]它从这个私有版本中读取数据，因此和其他事务所做的更新隔离开。如果事务更新数据库，更新只出现在其私有版本中，而不是实际的数据库本身中。当事务提交时，和更新有关的信息将保存，使得更新被写入"真正的"数据库。

当一个事务 T 进入部分提交状态后，只有在没有其他并发事务已经修改该事务想要更新的数据项的情况下，事务进入提交状态。而不能提交的事务则中止。

⊖ 当然，在现实中，不会复制整个数据库。只有改变的数据项才会保留多个版本。

快照隔离可以保证读数据的尝试永远无须等待(不像封锁的情况)。只读事务不会中止;只有修改数据的事务有微小的中止风险。由于每个事务读取它自己的数据库版本或快照,因此读数据不会导致此后其他事务的更新尝试被迫等待(不像封锁的情况)。因为大部分事务是只读的(并且大多数其他事务读数据的情况多于更新),所以这是与锁相比往往带来性能改善的主要原因。

矛盾的是,快照隔离带来的问题是它提供了太多的隔离。考虑两个事务 T 和 T'。在一个串行化调度中,要么 T 看到 T'所做的所有更新,要么 T'看到 T 所做的所有更新,因为在串行化顺序中一个必须在另一个之后。在快照隔离下,任何事务都不能看到对方的更新。这是在串行化调度中不会出现的。在许多(事实上,大多数)情况下,两个事务的数据访问不会冲突,因此没有什么问题。然而,如果 T 读取 T'更新的某些数据项并且 T'读取 T 更新的某些数据项,则可能两个事务都无法读取对方的更新。结果可能会导致我们将会在第 15 章看到的数据库不一致状态,而这个在可串行化执行中当然是不会出现的。 652

Oracle、PostgreSQL 和 SQL Server 提供快照隔离的选项。Oracle 和 PostgreSQL 使用快照隔离来实现可串行化隔离级别。因此,它们的可串行化的实现在某些特殊的情况下会导致允许一个非可串行化的调度。而 SQL Server 在标准级别以外增加了一个称为快照的隔离性级别,来提供快照隔离选项。

14.10 事务的 SQL 语句表示

4.3 节介绍了 SQL 中标识事务开始和结束的语法。现在我们已经介绍过一些保持事务 ACID 特性的问题,我们已经准备好考虑如何在用一系列 SQL 语句表示事务时保证这些特性,而不像到目前为止我们仅限制了简单的读和写的模型。

在简单模型中,我们假设存在一些数据项集合。虽然我们允许数据项的值改变,但是不允许数据项被创建或删除。然而,在 SQL 中,**insert** 语句用来创建新数据,**delete** 语句用来删除数据。事实上,这两条语句是 **write** 操作,因为它们改变了数据库,但是它们与其他事务操作的交互与我们在简单模型中看到的不同。例如,考虑如下在大学数据库上的查询语句,查找所有工资超过 $9000 的教师。

```
select ID, name
from instructor
where salary > 90000;
```

采用示例的关系 *instructor*(见附录 A.3),我们发现只有 Einstein 和 Brandt 满足条件。现在假设在执行该查询的同一时间,另外一个用户插入一条新的名为"James"并且工资为 $10 000 的教师数据。

```
insert into instructor values ('11111','James','Marketing', 100000);
```

查询结果会取决于该插入是先于还是后于执行的查询而有所不同。在这两个事务的并发执行中,从直观上看它们是冲突的,然而这种冲突在简单模型中无法发现。这种情况称为**幻象现象**(phantom phenomenon),因为冲突存在于一个"幻象"数据上。

简单事务模型要求提供一个具体的数据项作为操作的参数来执行在该数据项上的操作。在简单模型中,我们从 **read** 和 **write** 步骤中就可以看到哪个数据项被使用。但是在一条 SQL 语句中,具体的数据项(元组)可能被一条 **where** 语句中的谓词所决定。因此,如果在事务多次运行之间数据库发生改变,那么即使是同一个事务,在多次不同运行中也可能会使用不同的数据项。 653

解决上述问题的一种方法是要认识到在并发控制时仅考虑事务访问的元组是不够的;并发控制还需要考虑找到事务访问元组所需的信息。这些用于寻找元组的信息可能会被插入和删除所更新,或者在有索引的情况下,该信息还可能由于搜索键属性更新而更新。例如,如果采用封锁机制来进行并发控制,则用于追踪关系中元组的数据结构,以及索引结构都必须适当地封锁。然而,这种封锁可能会在一些情况下导致低的并发度。最大化并发性的索引封锁协议将会在 15.8.3 节介绍,它在插入、删除、带有谓词的查询中都保证了可串行化。

让我们再次考虑查询:

```
select ID, name
from instructor
where salary > 90000;
```

和以下 SQL 更新：

```
update instructor
set salary = salary * 0.9
where name = 'Wu';
```

我们在判断该查询到底是否和这条更新语句冲突时面临一个有趣的现象。如果我们的查询读取整个 *instructor* 关系，则它读取了与'Wu'相关的元组，因此和更新冲突。然而，如果存在索引，使得该查询可以直接访问 *salary* > 90 000 的元组，则该查询根本不会访问'Wu'的元组，因为在该实例关系中'Wu'的初始工资为 $90 000，且更新后减少为 $81 000。

但是，使用上述方法，看起来一个冲突是否存在依赖于底层系统的查询处理决策，而与用户级两条 SQL 语句的含义无关！另一种并发控制方法是如果一次插入、删除、更新会影响一个谓词所选择的元组，则将其看作与关系上的谓词冲突。在上述例子的查询中，谓词是"*salary* > 90 000"，则一个将'Wu'的工资从 $90 000 更新为比 $90 000 更高的更新，或者将'Einstein'的工资从高于 $90 000 更新到低于或等于 $90 000 的更新都会和谓词发生冲突。基于这种思想的封锁称为**谓词锁**(predicate locking)。然而，这种封锁代价大，因此实际中很少使用。

654

14.11 总结

- 事务是一个程序执行单位，它访问且可能更新不同的数据项。理解事务这个概念对于理解与实现数据库中的数据更新是很关键的，只有这样才能保证并发执行与各种故障不会导致数据库处于不一致状态。
- 事务具有 ACID 特性：原子性、一致性、隔离性、持久性。
  - 原子性保证事务的所有影响在数据库中要么全部反映出来，要么根本不反映；一个故障不能让数据库处于事务部分执行后的状态。
  - 一致性保证若数据库一开始是一致的，则事务(单独)执行后数据库仍处于一致状态。
  - 隔离性保证并发执行的事务相互隔离，使得每个事务感觉不到系统中其他事务的并发执行。
  - 持久性保证一旦一个事务提交后，它对数据库的改变不会丢失，即使系统可能出现故障。
- 事务的并发执行提高了事务吞吐量和系统利用率，也减少了事务等待时间。
- 计算机中不同的存储介质包括易失性存储器、非易失性存储器和稳定性存储器。易失性存储器(例如 RAM)中的数据当计算机崩溃时丢失。非易失性存储器(如磁盘)中的数据在计算机崩溃时不会丢失，但是可能会由于磁盘崩溃而丢失。稳定性存储器中的数据永远不会丢失。
- 必须支持在线访问的稳定性存储器与磁盘镜像或者其他形式的提供冗余数据存储的 RAID 接近。对于离线或归档的情况，稳定性存储器可以由存储在物理安全位置的数据的多个磁带备份所构成。
- 多个事务在数据库中并发执行时，数据的一致性可能不再维持。因此系统必须控制各并发事务之间的相互作用。
  - 由于事务是保持一致性的单元，所以事务的串行执行能保持一致性。
  - 调度捕获影响事务并发执行的关键操作，如 **read** 和 **write** 操作，而忽略事务执行的内部细节。

655

  - 我们要求事务集的并发执行所产生的任何调度的执行效果等价于由这些事务按某种串行顺序执行的效果。
  - 保证这个特性的系统称为保证了可串行化。
  - 存在几种不同的等价概念，从而引出了冲突可串行化与视图可串行化的概念。
- 事务并发执行所产生的调度的可串行化可以通过多种并发控制机制中的一种来加以保证。
- 给定一个调度，我们可以通过为该调度构造优先图及搜索是否无环来判定它是否冲突可串行化。然而，有更好的并发控制机制可用来保证可串行化。
- 调度必须是可恢复的，以确保：若事务 a 看到事务 b 的影响，当 b 中止时，a 也要中止。
- 调度最好是无级联的，这样不会由于一个事务的中止引起其他事务的级联中止。无级联性是通

过只允许事务读取已提交数据来保证的。

- 数据库的并发控制管理部件负责处理并发控制机制。第 15 章阐述并发控制机制。

术语回顾

- 事务
- ACID 特性
  - □ 原子性
  - □ 一致性
  - □ 隔离性
  - □ 持久性
- 不一致状态
- 存储器类型
  - □ 易失性存储器
  - □ 非易失性存储器
  - □ 稳定性存储器
- 并发控制系统
- 故障恢复系统
- 事务状态
  - □ 活动的
  - □ 部分提交的
  - □ 失败的
  - □ 中止的
  - □ 提交的
  - □ 已结束的
- 事务
  - □ 重启
  - □ 杀死
- 可见的外部写
- 并发执行
- 串行执行
- 调度
- 操作冲突
- 冲突等价
- 冲突可串行化
- 可串行化判定
- 优先图
- 可串行化顺序
- 可恢复调度
- 级联回滚
- 无级联调度
- 并发控制机制
- 封锁
- 多版本
- 快照隔离

实践习题

14.1 假设存在永远不出现故障的数据库系统，对这样的系统还需要故障恢复管理器吗？

14.2 考虑一个文件系统，比如你最喜欢的操作系统的文件系统，

a. 创建和删除文件分别包括哪些步骤，向文件中写数据呢？

b. 试说明原子性和持久性问题与创建和删除文件以及向文件中写数据有什么关系。

14.3 数据库系统实现者比文件系统实现者更注意 ACID 特性，为什么会这样？

14.4 论证下面的说法：必须从(慢速的)磁盘获取数据或执行长事务时，事务的并发执行更为重要，而当数据存在于主存中且事务非常短时，没那么重要。

14.5 既然每一个冲突可串行化调度都是视图可串行化的，我们为什么强调冲突可串行化而非视图可串行化呢？

14.6 考虑图 14-16 所示的优先图，相应的调度是冲突可串行化的吗？解释你的回答。

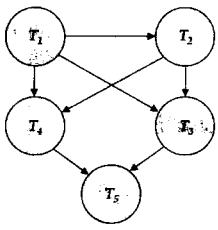

图 14-16 实践习题 14.6 的优先图

14.7 什么是无级联调度？为什么要求无级联调度？是否存在要求允许级联调度的情况？解释你的回答。

14.8 **丢失更新**(lost update)异常是指如果事务 T_j 读取了一个数据项，然后另一个事务 T_k 写该数据项(可能基于先前的读取)，然后 T_j 写该数据项。于是 T_k 所做的更新丢失了，因为 T_j 的更新覆盖了 T_k 写入的值。

a. 给出一个表示丢失更新异常的调度实例。

b. 给出一个表示丢失更新异常的调度实例，表明在**已提交读**隔离性级别下该异常也可能存在。

c. 解释为什么在**可重复读**隔离性级别下丢失更新异常不可能发生。

14.9 考虑一个采用快照隔离的银行数据库系统。描述一个出现非可串行化调度会为银行带来问题的特定场景。

14.10 考虑一个采用快照隔离的航空公司数据库系统。描述一个出现非可串行化调度但是航空公司为了更好的整体性能而愿意接受的特定场景。

14.11 一个调度的定义假设操作可以完全按时间顺序排列。考虑一个运行在多处理器系统上的数据库系统，它并不总是能对于运行在不同处理器上的操作确定一个准确的顺序。但是，一个数据项上的操作全部可以排序。

以上情况是否对冲突可串行化的定义造成问题？解释你的答案。

习题

14.12 列出 ACID 特性，解释每一特性的用途。

14.13 事务从开始执行直到提交或终止，其间要经过几个状态。列出所有可能出现的事务状态序列，解释每一种状态变迁出现的原因。

14.14 解释串行调度和可串行化调度的区别。

14.15 考虑以下两个事务：

T_{13}: **read**(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B);

T_{14}: **read**(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A);

设一致性需求为 $A = 0 \vee B = 0$，初值是 $A = B = 0$。

a. 说明包括这两个事务的每一个串行执行都保持数据库的一致性。

b. 给出 T_{13} 和 T_{14} 的一次并发执行，执行产生不可串行化调度。

c. 存在产生可串行化调度的 T_{13} 和 T_{14} 的并发执行吗？

14.16 给出两个事务的一个串行化调度的例子，其中事务的提交顺序与串行化序列不同。

14.17 什么是可恢复调度？为什么要求调度的可恢复性？存在要求允许出现不可恢复调度的情况吗？解释你的回答。

14.18 为什么数据库系统支持事务的并发执行，尽管需要额外编程工作来确保并发执行不会引起任何问题？

14.19 解释为何已提交读隔离性级别保证调度是无级联的。

14.20 对于以下隔离性级别，给出一个满足该隔离性级别但不是可串行化的调度实例：

656 ~ 659

a. 未提交读

b. 已提交读

c. 可重复读

14.21 假设除了 **read** 和 **write** 操作，我们允许一个操作 **pred_read**(r, P) 读取关系 r 中所有满足谓词 P 的元组。

a. 给出一个使用 **pred_read** 的调度实例来展示一个有幻象现象的非可串行化调度。

b. 给出一个调度实例，其中一个事务在关系 r 上用 **pred_read**，另一个事务删除 r 中的一个元组，但是调度中没有幻象冲突。（为此，你需要给出关系 r 的表结构，并且显示被删除元组的值）。

文献注解

Gray 和 Reuter[1993]在其教科书中全面讨论了事务处理的概念、技术和实现，包括并发控制和恢复问题。Bernstein 和 Newcomer[1997] 在其教科书中讨论了事务处理的多个方面。

Eswaran 等[1976]对可串行化的概念进行了形式化描述，这是同他们在 System R 的并发控制上的工作相关联的。

660 事务处理各个具体方面（如并发控制和故障恢复）的参考文献见第 15、16 和 26 章。