第15章 并 发

15.1 引言

并发和恢复紧密相关，都是事务管理的重要组成部分，本章将重点讨论并发控制。DBMS通常允许多个事务同时存取相同的数据库，这就是并发。在这样的系统中，必须提供某种并发控制机制以确保并发事务间互不干扰。在 15.2节将给出一些事务间相互干扰的例子，这都是在缺乏必要的控制情况下发生的。

本章的结构如下：

- 15.2节说明若不进行适当的并发控制将会出现什么样的问题。
- 15.3节介绍处理上述问题的传统方法：锁。注：锁并不是解决并发控制问题的唯一方法，但却是实际运用最广泛的方法。其他的方法可参考对文献 [15.1]、[15.3]、[15.6~15.7]以及[15.14~15.15]的注释。
- 15.4节解释了锁机制如何用于解决 15.2节中描述的问题。
- 15.5节讨论了锁自身所引发的问题，其中最著名的就是死锁。
- 15.6节描述了可串行性的概念，其通常作为并发事务执行正确性的准则。
- 15.7节和15.8节对锁的基本思想作了进一步的讨论，即隔离级别和意向锁。
- 15.9给出了SQL的相关表示。
- 15.10给出了小结和几点结论。

注意：第14章引言部分的两点说明在这里仍然适用：

- 并发与恢复一样，与底层系统是关系型还是其他类型无关。但是这一领域早期的理论工作大多是在特定的关系型环境下进行的 [15.5]。
- 并发这一主题的范围很广，本章只是对其中最重要的和最基本的思想进行了介绍。练习、答案以及本章最后对参考文献的注释涉及了对这一主题的更深层次的讨论。

15.2 三个并发问题

首先考虑任何一个并发控制机制必须提及的一些问题：在三种情况下必定将发生错误，即在这三种情况下，一个事务即使自身是正确的，如果其他的某个事务以某种方式与之交错运行，也可能引发错误。注意：其他事务本身也可能是正确的，正是未对两个正确运行事务的交错在一起的操作进行控制产生了全局不一致的结果。关于事务“自身正确”的概念，其含义是指不违背黄金法则，参看第 8章。三个问题是：

- 丢失更新问题；
- 未提交依赖问题；
- 不一致分析问题。

下面将依次进行讨论。

1. 丢失更新问题

考虑图15-1的情形。图例的意思是：事务*A*在时间*t1*检索元组*t*；事务*B*在时间*t2*检索同一元组*t*；事务*A*在时间*t3*更新元组*t*（基于时间*t1*所看到的值）；事务*B*在时间*t4*更新元组*t*（基于时间*t2*所看到的值，与*t1*时间的值相同）。事务*A*的更新在*t4*时间丢失，因为事务*B*甚至都没看它就将其覆盖了。

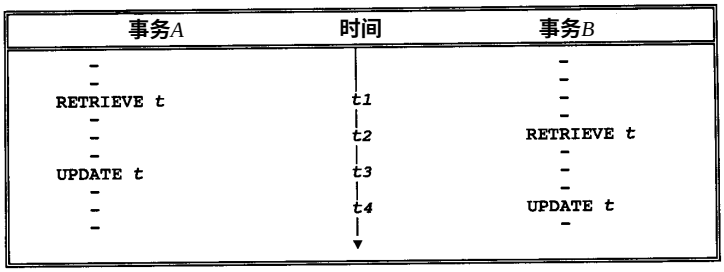

图15-1 事务*A*在*t4*时间丢失更新

2. 未提交依赖问题

如果一个事务允许检索，或更糟糕的是，更新一个已被其他事务更新但未提交的元组，这将引起未提交依赖问题。因为如果事务尚未提交，总有可能其将不再提交而是回滚，这种情况下，第一个事务所看到的数据当前并不存在，某种意义上，从未存在过。参看图 15-2和图15-3。

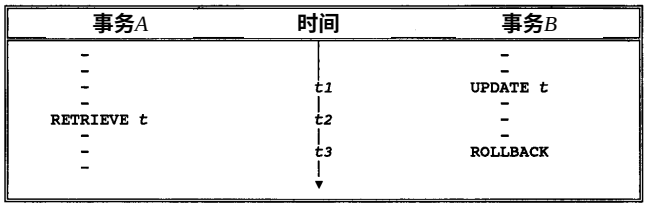

图15-2 事务*A*在时间*t2*依赖未提交的变化

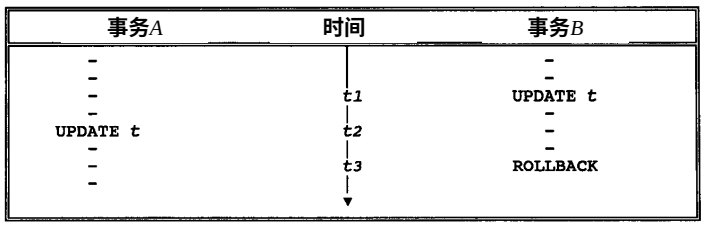

图15-3 事务*A*在时间*t2*更新未提交的变化，并在时间*t3*丢失了该更新

在第一个例子（图 15-2）中，事务*A*在时间*t2*看到一个未提交的更新（也称作未提交的变化），该更新在时间*t3*被撤消。事务*A*的操作基于错误的假设，即假设元组 *t*具有时间*t2*时看到的值，但实际上为时间 *t1*前的值，结果事务 *A*产生了不正确的输出结果。注意，事务 *B*的回滚可能并不是*B*本身的故障造成的，举个例子，可能是系统故障的结果，此时事务 *A*可能已终止，该故障并不引起*A*的回滚。

第二个例子情况更糟。事务 *A*不仅依赖时间*t2*未提交的变化，而且实际上在时间 *t3*还丢失了更新，因为时间 *t3*时的回滚使得元组 *t*恢复为时间*t1*以前的值。这是丢失更新问题的又一形式。

3. 不一致分析问题

图15-4表示了事务*A*和*B*对帐户（ACC）元组的操作：事务*A*对帐户余额进行求和，事务*B*将帐户3的金额10转到帐户1上。*A*产生的结果为110，显然不正确。如果*A*将该结果写回到数据库，实际上将使数据库处于不一致的状态[⊖]。事务*A*看到了数据库的不一致的状态，并因此进行了不一致的分析。注意该例与前例的不同：这里不存在*A*依赖未提交的更新问题，因为*B*在*A*看到ACC 3前已提交了所有更新。

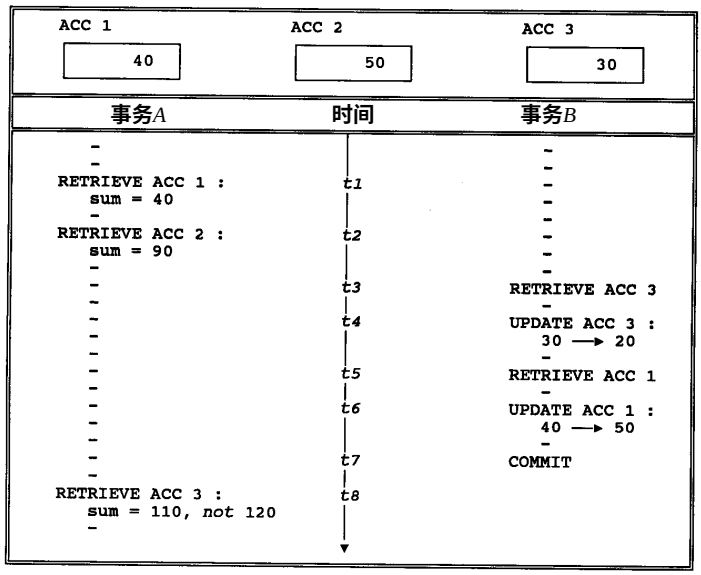

图15-4 事务*A*进行了不一致的分析

15.3 锁

正如15.1节中指出的，可采用锁这一并发控制技术解决15.2节的所有问题。基本思想很简单：当一个事务需要确保其感兴趣的对象（通常是一数据库元组）在其完成时不会发生某种形式的改变，它必须获得该对象上的一个锁。锁的作用就是锁住对象使得其他事务无法访问，尤其是阻止其他事务改变该对象。因此第一个事务就能知道该对象将保持稳定的状态，并顺利执行。

下面将对锁的工作方式做更详细的解释：

1) 首先假定系统支持两种锁，排它锁（exclusive lock简记为X锁）和共享锁（shared lock简记为S锁），定义将在下面两个段落给出。注意：X锁和S锁有时也分别称作写锁和读锁。在进一步讨论之前，假设X锁和S锁是仅有的类型，在15.8节还有其他类型的锁的例子；同时假设元组是仅有的可锁的对象，在15.8节还给出了其他类型的可锁的对象。
2) 如果事务*A*在元组*t*上具有排它锁（X），则来自某个不同事务*B*对*t*的任何一种类型的锁的请求将被拒绝。

⊖ 关于这种可能性（即将结果写回到数据库），显然必须假设没有完整性约束以防止这样的写操作。

3) 如果事务*A*在元组*t*上具有共享锁（S），则：

- 来自某个不同事务*B*对*t*的X锁的请求将被拒绝；
- 来自某个不同事务*B*对*t*的S锁的请求将被满足，因为*B*也将具有*t*上的S锁。

这些规则可用锁类型相容矩阵表示，见图 15-5。该矩阵解释如下：考虑某个元组 *t*；假定事务*A*当前具有*t*上的锁，用左边列表示（破折号 =没有锁）；假定某个不同的事务*B*发出了对*t*的锁请求，用最上面的一行表示（为完整起见，仍包括“没有锁”的情况）。“N”表示冲突，即*B*的请求不能被满足，将处于等待状态；“Y”表示相容，即*B*的请求被满足。矩阵显然是对称的。

	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

图15-5 锁类型X和S的相容矩阵

下面介绍数据存取协议（或称为锁协议），其运用X锁和S锁保证了15.2节描述的问题不会发生[⊖]：

1) 事务在检索元组前需获得该元组上的S锁。

2) 事务在更新元组前需获得该元组上的X锁。相应地，如果其已经具有了该元组上的S锁（因为通常是RETRIEVE-UPDATE的顺序），必须将其从S锁升级到X锁。

注意：事务对元组的锁请求通常都是隐式的，“检索元组”的请求通常隐含着对相关元组的S锁请求，“更新元组”的请求通常隐含着对相关元组的X锁请求。当然，术语“更新”包括INSERT、DELETE以及UPDATE，但是对INSERT和DELETE操作，规则需进一步的提炼，这里略去了细节。

3) 如果事务*B*的锁请求因为与事务*A*已具有的锁冲突而被拒绝，事务*B*将处于等待状态，直到*A*的锁被释放掉。注意：系统必须保证*B*不会永远处于等待状态（这是有可能出现的，通常称作活锁）。避免活锁的简单方法是采用先来先服务的策略。

4) X锁将一直保持到事务结束（COMMIT或ROLLBACK）。S锁通常也保持到那时，例外情形可参看15.7节。

15.4 重提三个并发问题

本节解释上一节所介绍的方法如何用于解决15.2节中描述的问题。

1. 丢失更新问题

图15-6是图15-1的修改形式，表示图15-1在锁协议下事务交错执行的情况。事务*A*在时间*t3*的UPDATE未被接受，因为其隐含着对*t*的X锁请求，该请求与事务*B*拥有的S锁冲突，因此*A*进入等待状态。类似的原因，*B*在时间*t4*也进入等待状态。这样两个事务都无法继续执行，因此不存在任何丢失更新问题。锁解决了丢失更新问题，但同时又引起另外一个问题。但至少其已解决了最初的问题，新问题称作死锁，将在15.5节讨论。

2. 未提交依赖问题

图15-7和图15-8分别是图15-2和图15-3的修改形式，表示图15-2和图15-3在锁协议下事务交错执行的情况。事务*A*在时间*t2*的操作（图15-7中为RETRIEVE，图15-8中为UPDATE）在两种情况下都未被接受，因为其隐含着对*t*的请求，这与*B*所拥有的X锁相冲突，因此*A*进入等待状态。其一直处于等待状态直到事务*B*的执行结束（或者是COMMIT，或者是ROLLBACK），当*B*的锁

⊖ 这里描述的协议是两阶段锁(具体在15.6节中讨论)的一个例子。

释放掉后，*A*才能继续执行，而且在此时*A*所看到的是提交后的值（如果*B*以回滚结束，则为*B*之前的值；否则为*B*之后的值）。任何一种情况下，*A*都不再依赖未提交的更新。

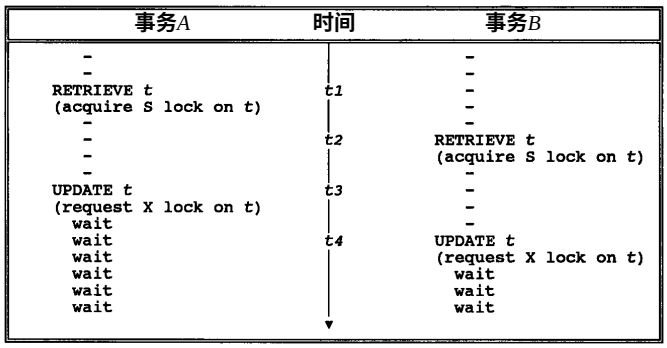

图15-6 更新未丢失，但在时间*t4*发生了死锁

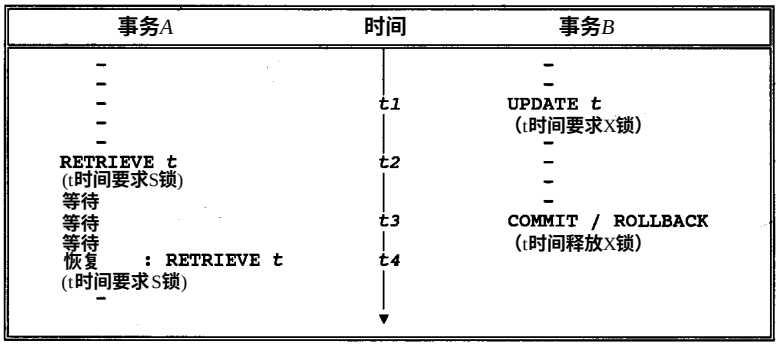

图15-7 事务*A*在时间*t2*无法检索未提交的更新

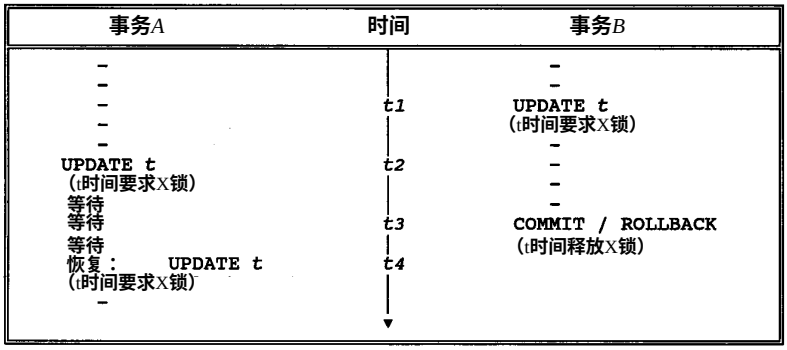

图15-8 事务*A*在时间*t2*无法修改未提交的更新

3. 不一致分析问题

图15-9是图15-4的修改形式，表示图15-4在锁协议下事务交错执行的情况。事务*B*在时间*t6*的UPDATE未被接受，因为其隐含着对ACC 1的X锁请求，该请求与事务*A*拥有的S锁冲突，因此*B*进入等待状态。同样事务在时间*t7*的RETRIEVE也未被接受，因为其隐含着对ACC3的S锁请求，该请求与事务*B*拥有的X锁冲突，因此*A*也进入等待状态。因此，锁方法解决了最初的不一致分析问题，同时引起了死锁。

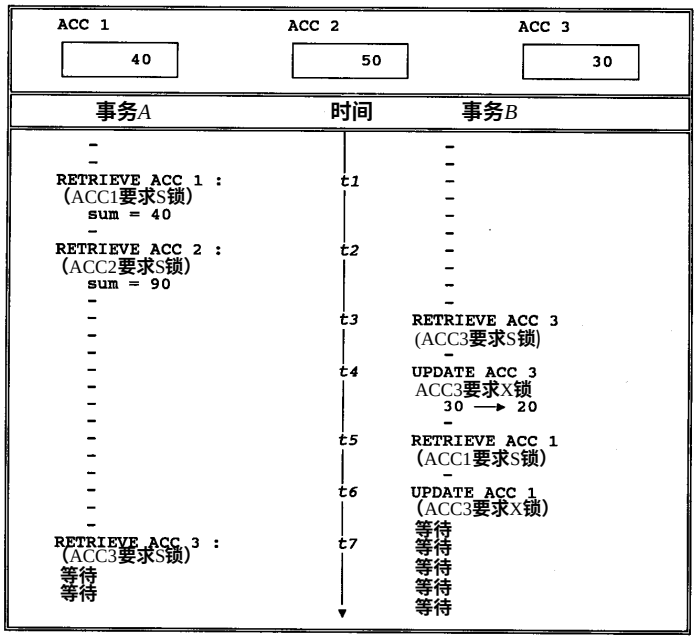

图15-9 防止了不一致分析，又发生了死锁

15.5 死锁

到目前为止，我们已知道锁是如何用来解决三个基本的并发问题的。不幸的是，锁自身也引发了问题，主要为死锁。前一节已给出了两个死锁的例子。图 15-10给出了更一般的表示形式，图中的*r1*和*r2*（*r*表示资源）表示任何可锁的对象，不必仅为数据库的元组（参看 15.8节），语句“ LOCK ...EXCLUSIVE ”表示任何申请（排它）锁的操作，可以为显式或隐式的请求。

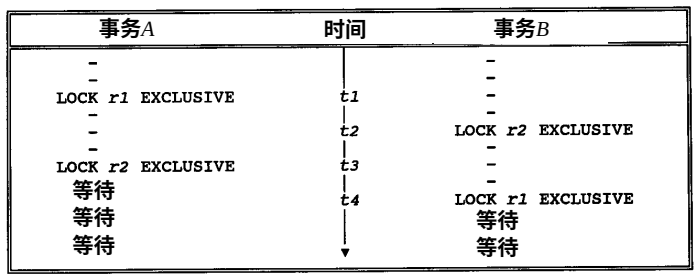

图15-10 死锁示例

死锁发生时两个或更多的事务同时处于等待状态，每个事务都在等待其它的事务释放锁使其可继续执行[⊖] 。图15-10中死锁发生时涉及两个事务，但涉及三个、四个甚至更多事务都是可能的，至少原理上是这样。但是 System R的实验表明，实际上死锁几乎从未涉及两个以上的事务[15.8]。

⊖ 死锁也被形象地称作死死拥抱(deadly embrace)。

死锁发生时，系统必须能检测并解除它。检测死锁就是检测等待图（即“谁在等待谁”的图——参看练习15.4）中的环。解除死锁就是选出一个死锁的事务，即图中环上的一个事务，将之回滚，从而释放其所拥有的锁使得其他一些事务可继续执行下去。注意：实际上，并不是所有的系统都进行死锁检测，有些系统采用超时机制，简单地假设在预定时间内不工作的事务发生了死锁。

可以看出，被选择回滚的事务自身并未发生错误，有些系统将自动重启这样的事务，当然这是基于最初引起死锁的条件可能不再出现的假设。其他的系统则简单地向应用程序返回“死锁受害者”的代码，以便应用程序以某种得体的方式处理该情形。从程序员的角度看，前一种方法显然更可取。但是，即使有时需要程序员进行必要的处理，将问题隐藏起来不让最终用户感知总是很必要的。

15.6 可串行性

有了前面的基础，现在可解释一个重要概念——可串行性。可串行性通常作为多个事务执行的正确性准则。更精确的说法是，多个事务的某个执行过程是正确的，当且仅当其是可串行化的（如果多个事务的某个执行过程与同样的这些事务的某个串行执行过程产生的结果相同，则称其为可串行化的）。判断方法如下：

1) 各单个事务如能将数据库从一个正确状态转变为另一个正确状态，就认为其是正确的。
2) 按任何一个串行顺序依次运行多个事务也认为是正确的。这里串行顺序可为“任何一个”，是基于各单个事务彼此间相互独立的假定。
3) 交错执行过程是正确的，当且仅当其与某个串行执行过程等价，即是可串行化的。

从15.2节中的例子（图15-1~图15-4）可看出任何一种情形的问题都在于交错执行过程是不可串行化的，也就是说，其与先A后B或先B后A的串行执行过程都不等价。15.3节所讨论的锁模式正是强制了每种情形的可串行性。图15-7和图15-8中的交错执行过程等价于先B后A。图15-6和图15-9中发生了死锁，这暗示了两个事务中的一个将被回滚，假设其稍后将再次执行。如果A被回滚，则交错执行过程等价于先B后A。

术语：多个事务，其任何一个执行过程，无论是否交错都被称作调度。依次执行这些事务，不相互交错，构成了一个串行调度。不是串行的调度称作交错调度，或简单地称作非串行调度。两个调度若保证产生同样的结果，与数据库的初始状态无关，则称其是等价的。因此一个调度是正确的（即可串行化的），当且仅当其与某一串行调度等价。

这里强调一下，相同的多个事务的两个不同的串行调度可能产生不同的结果，因此这些事务的两个不同的交错调度也可能产生不同的结果，但两者都是正确的。举个例子，假设事务A为“x加上1”，事务B为“将x翻番”，这里x为数据库中的某项。同时假设x的初始值为10。则先A后B的串行调度结果为x=22，而先B后A的串行调度结果为x=21。这两个结果都是正确的。任何一个保证与先A后B或先B后A等价的调度同样都是正确的。参看本章的练习15.3。

可串行性的概念最初是由Eswaran在参考文献[15.5]中提出的，尽管使用的不是该名称。论文中还证明了一个重要的理论，称作两阶段锁理论，下面简单地介绍一下[⊖]。

如果所有的事务都遵守“两阶段锁协议”，则所有可能的交错调度都是可串行化的。

⊖ 两阶段锁与两阶段提交无关，它们只是有类似的名字而已。

两阶段锁协议为：

1) 在对任何一个对象（如一个数据库元组）进行操作之前，事务必须获得对该对象的锁；

2) 在释放一个锁之后，事务不再获得任何其他锁。

遵守该协议的事务分为两个阶段：获得锁阶段，也称为“扩展”阶段；释放锁阶段，也称为“收缩”阶段。注意：实际系统中收缩阶段通常被压缩为事务结束时的单个操作COMMIT或ROLLBACK。实际上，15.3节的数据存取协议可看作两阶段锁协议的加强形式。

可串行性的概念有助于对事务并发的更清楚的理解，这里将给出另外一些结论。假设 I为涉及事务$T1$，$T2$，...，Tn的交错调度。如果I可串行化，则存在某个涉及事务 $T1$，$T2$，...，Tn的串行调度S，使得I等价于S。S为I的串行调度（serialization）。正如前面已讨论的，S不一定唯一，即某一交错调度可具有多个串行调度。

假定Ti和Tj为$T1$，$T2$，...，Tn事务集中的任意两个事务。不失一般性，假定在串行调度 S中Ti在Tj之前执行。因此，在交错调度I的效果就像Ti确实在Tj之前执行一样。换句话说，可串行性的非正式但却很有用的特征为：如果 A和B为某个可串行化调度的任意两个事务，则该调度中或者逻辑上A在B之前或者逻辑上B在A之前，即或者B看到A的输出结果或者A看到B的输出结果。如果A更新资源$r,s,...,t$，并且如果B可看到其中的任一资源，则B或者在所有的资源被A更新后看到这些资源，或者在A更新所有的资源之前就看到这些资源，总之不可能出现两者相混合的情况）。相反，如果效果不像A在B之前执行或B在A之前执行，则调度是不可串行化的，因而也是不正确的。

最后必须强调一点，如果某个事务A不是两阶段的，即不遵守两阶段锁协议，则总能构造某个其他的事务B以某种方式与A交错执行，产生不可串行化的不正确的全局调度。为了减少资源冲突，从而改善性能和吞吐量，实际系统通常构建不是两阶段的事务，即事务提早释放锁（在COMMIT前），并继续获得更多的锁。显然这样的事务冒着很大的风险，实际上，允许某一给定的事务A为非两阶段的，赌的是在系统中不存在与 A交错运行的事务B，否则系统很可能将产生错误的答案。

15.7 隔离级别

隔离级别用来不严格地表示一个事务在与其它事务并发执行时所能容忍干扰的程度。如果可串行性得到了保证，根本就不存在可能被接受的干扰，换句话说，就是此种情况的隔离级别为可能的最高级别。然而，正如前一节最后指出的，由于各种实用的原因，实际系统通常允许事务在低于最高级别的隔离级别下操作。

注意：正如前一段落所表明的，隔离级别通常被认为是事务所具有的属性。实际上没有任何理由说明为什么一个事务不应该同时对数据库的不同部分在不同的级别进行操作，至少理论上没有给出原因。但是为简化起见，仍将隔离级别作为事务的属性。

至少可定义五个不同的隔离级别，但参考文献 [15.9]以及SQL标准分别只定义了四个，DB2当前支持两个。通常，隔离级别越高，干扰越少，并发程度越低；隔离级别越低，干扰越多，并发程度越高。作为示例，可考虑 DB2所支持的两个级别，分别为游标稳定性（cursor stability）和可重复读（repeatable read）。可重复读（RR）是最高级别，如果所有的事务都操

作在该级别，则所有的调度都是可串行化的（ 15.3节和15.4节的解释默认假设了该级别）。而在游标稳定性（CS）级别，如果事务*T1*

- 获得了对某个元组*t*的可寻址性[⊖]，因此
- 获得对*t*的锁，接着
- 未对元组更新并放弃了对*t*的可寻址性，因此
- 未将锁升级为X锁，随后
- 不必等到事务结束即可释放该锁。

但是，其它的某个事务*T2*现在可对*t*进行更新并提交该改变。如果事务*T1*又再次访问*t*，将看到数据库的不一致状态。而在可重复读（ RR）级别，所有对元组的锁，不仅仅是 X锁，都将保持到事务结束，从而前面提到的问题将不会产生。

几点说明：

1) 前面提到的问题并不是在CS级别产生的唯一问题，只不过其最容易解释。但是不幸的是，这表明只有在一个事务需要访问元组两次这一相比较而言不大可能发生的情况下才需要RR级别。相反，有观点认为 RR总是比CS更优的选择，在CS级别运行的事务不是两阶段的，因此无法保证可串行性。当然，相应的观点认为 CS比RR的并发程度高（可能但不一定）。
2) 支持低于最高级别的隔离级别的系统实现通常提供一些显式的并发控制能力（一般为显式的LOCK语句），从而允许用户利用其编写自己的应用程序，以保证系统自身未提供保证的安全性。DB2提供了显式的LOCK TABLE语句，允许用户在CS级别进行操作以获得显式的锁，CS以上的级别DB2将自动获得锁。但是，注意：SQL标准不包括这样的显式的并发控制机制，参看15.9节。

注意：DB2实现中的可重复读被作为最高级别，但 SQL标准使用同样的术语“可重复读”表示的级别严格低于最高级别（参看15.9节）。

15.8 意向锁

到目前为止，一直假设锁单元是单个元组。但理论上没有任何理由能说明为什么锁不能应用在更大或更小的数据单元，如整个关系变量甚至整个数据库，或特定元组的特定部分。由此将引出锁粒度这一概念[15.9~15.10]。一般折衷考虑：粒度越细，并发程度越高；粒度越粗，需要设置和测试的锁就越少，负载越低。举例来说，如果一个事务具有一个关系变量上的X锁，则没有必要对该变量的各单个元组设置 X锁；另一方面，没有并发事务可获得该关系变量上的锁或该变量的元组上的锁。

假设某个事务*T*确实申请某关系变量*R*上的X锁。在接受*T*的请求之前，系统必须能判断出其它的事务是否已拥有*R*的任何一个元组上的锁，如果确实拥有，则 *T*的请求此时无法满足。系统怎样才能检测出这样的冲突？显然，检查每个元组看是否有元组被其它的某个事务锁住，或检查每个已存在的锁看是否有锁是施加在*R*的元组上的方法都是不明智的。由此将介绍另一个协议，即意向锁协议。根据该协议，事务在允许对某一元组锁之前必须首先获得对包含该

⊖ 正如在第4章提到的，通过设置一指向元组的游标获得对该元组的可寻址性，这也是“游标稳定性”名字的由来。提醒一下，在DB2中T1申请的锁是“更新”(U)锁，而不是S锁(参看[4.20])。

元组的关系变量上的锁，即意向锁（参见下面）。该例中的冲突检测由此将变得很简单，只需看事务在关系变量级是否有冲突的锁。

前面已暗指X锁和S锁对关系变量和对单个元组一样有意义，根据参考文献[15.9~15.10]，这里将再介绍三种新的锁，称为意向锁，这些锁只对关系变量有意义，而对单个元组无意义。这三种锁分别称为意向读（IS）、意向写（IX）、共享意向排它锁（SIX）。它们非形式化的定义如下（假定事务*T*请求获得关系变量*R*上的指定类型的锁，为完整起见，这里也包括X锁和S锁的定义）：

- IS：*T*意欲对*R*的元组设置S锁，以保证这些元组被处理时的稳定性。
- IX：与IS类似，必须补充一点，*T*可能对*R*的元组进行更新，从而将对这些元组设置X锁。
- S：*T*允许并发的读操作，但不允许并发的更新操作。*T*自身在*R*的任意元组上都无更新操作。
- SIX：综合了S和IX锁。
- X：*T*根本就不允许任何对*R*的并发存取，*T*自身可能对*R*的元组进行更新操作。

这五种锁类型的正式定义可通过前面15.3节所提到的锁类型相容矩阵的扩展形式表示。如图15-11。

	X	SIX	IX	S	IS	–
X	N	N	N	N	N	Y
SIX	N	N	N	N	Y	Y
IX	N	N	Y	N	Y	Y
S	N	N	N	Y	Y	Y
IS	N	Y	Y	Y	Y	Y
–	Y	Y	Y	Y	Y	Y

图15-11 扩展后包括意向锁的相容矩阵

下面给出意向锁协议的更精确的表述：

1) 事务在获得某一元组上的S锁之前必须首先获得包含该元组的关系变量上的IS或更强的锁；
2) 事务在获得某一元组上的X锁之前必须首先获得包含该元组的关系变量上的IX或更强的锁。

但是这还不是完整的定义，参看[15.9]的注释。

前面协议中提到的相对锁强度的含义解释如下（参见图15-12）：锁类型*L2*强于（在图中高于）锁类型*L1*，当且仅当在相容矩阵某一行中*L1*列对应的那项为“N”（冲突）。在同一行中*L2*对应的那项也为“N”（参看图15-11）。若某一锁类型的请求无法满足，对比之更强的锁类型的请求必定也无法满足。这一事实表明使用比要求的锁类型更强的锁总是安全的。S和IX彼此都不比对方更强。

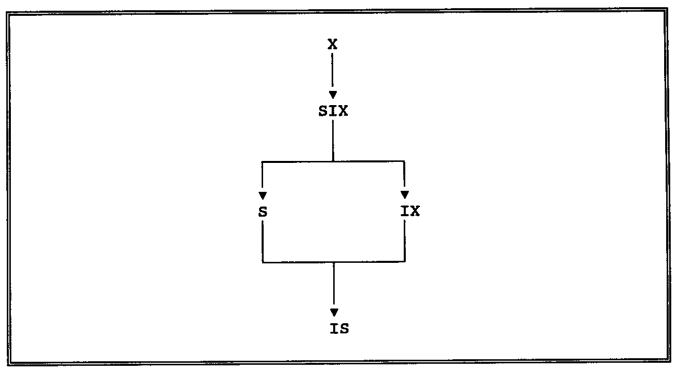

图15-12 锁类型前驱图

意向锁协议所要求的关系变量级的锁通常被隐式地获得。例如，对某一只读事务，系统可能对事务存取的每一个关系变量都隐式地获得IS锁。而对某一更新事务，则可能获得IX锁。但系统也可能提供某种显式的LOCK语句以允许事务获得关系变量上的S、X、SIX锁。DB2就支持这样的语句，虽然只有S和X锁，没有SIX锁。

最后讨论锁升级（lock escalation）的概念。锁升级在许多系统中得以实现，用于平衡高并发度和低锁管理开销之间的冲突。其基本思想就是当系统达到预定的阈值时，系统将自动用一个粗粒度的锁取代一组细粒度的锁。例如，取代一组单个元组级的S锁，将包含这些元组的关系变量上的IS锁改变为S锁。该技术在实际应用中很有效。

15.9 SQL的支持

SQL标准不提供任何显式的锁支持，实际上，根本就未提及锁。但是，实际系统必须提供相关措施以免并发执行事务相互干扰。特别地，它要求事务 *T1*的更新对其他不同的事务 *T2*不可见，直到事务 *T1*以提交更新结束。以提交更新结束使得事务所做的所有更新对其他事务都是可见的，以回滚更新结束使得事务所做的所有更新都被取消。

注意：前面假设所有的事务隔离级别为 READ COMMITED(读提交)、REPEATABLE READ(可重复读)以及SERIALIZABLE(可串行化的)。对READ UNCOMMITED级别的事务给予了特别的考虑，该级别允许事务“读脏数据”，但事务必须被定义为READ ONLY（在第14章有说明）。

隔离级别

第14章中提到SQL包含这样的语句SET TRANSACTION，用于定义下一个事务的特性，其中一个特性就是隔离级别。可能的级别有 READ UNCOMMITED、READ COMMITED、REPEATABLE READ以及SERIALIZABLE[⊖]。缺省为SERIALIZABLE。如果指定了其他三个级别中的任何一个，系统将设置为某一较高的级别。这里“较高”是根据这样的排序定义：SERIALIZABLE>REPEATABLE READ>READ COMMITED>READ UNCOMMITED。

如果所有事务都执行在隔离级别SERIALIZABLE（缺省），则任意一组事务的交错执行过程都是可串行化的。但是，如果有事务执行在较低的隔离级别，则可能存在多种不同的违背可串行性的情形。标准定义了三种情形：读脏数据、不可重复读和幻影。

- 读脏数据：假设事务*T1*在某行上执行了更新操作，然后事务 *T2*检索了该行，但事务 *T1*后来以回滚结束。那么事务*T2*看到了一个已不存在的行，也可以说是从未存在的行（因为事务*T1*可能决不再运行）。
- 不可重复读：假设事务*T1*检索了某行，然后事务 *T2*更新了该行，此后事务 *T1*再次检索“同一”行。事务*T1*检索“同一”行两次但是看到的结果却不同。
- 幻影：假设事务*T1*检索到一个符合某些条件的行集（例如：所在城市为巴黎的所有的供应商），另假设事务*T2*又插入了满足同一条件的新的一行。那么如果这时事务 *T1*发出相同的检索请求，它将看到一个以前不存在的行——“幻影”。

不同的隔离级别是根据各个级别对前面说明的串行化违背的允许程度来定义的。图 15-13给出了所有的四个隔离级别。其中， Y表示指定的违背可能发生， N则表示指定的违背不

⊖ SERIALIZABLE用在这里并不特别合适，因为是调度可串行化，而不是事务。更准确的术语可以是 TWO PHASE，其含义在于事务将遵循（或将被迫遵循）两阶段锁协议。

会发生。

隔离级别	读脏数据	不可重复读	幻影
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

图15-13 SQL隔离级别

那么，系统如何才能阻止“幻影”的发生呢？解决的方法就是系统必须对可能使用的数据的存取路径加锁。例如，对于前面提到的例子中的巴黎的供应商，如果存取路径恰好是建立在供应商所在城市的索引上，则系统必须锁定索引中的巴黎这一项，这样的锁将阻止幻影的产生。原因在于幻影的产生需要对存取路径（本例中为索引项）进行更新。关于这一问题的更深层次的讨论可以阅读文献[14.12]。

这里要重复的一点就是：SQL的可重复读（REPEATABLE READ）和DB2中的“可重复读（RR）”不是一回事。事实上，DB2中的RR与SQL中的“可串行化的”的隔离级别相同。

15.10 小结

本章考虑了有关并发控制的问题。首先介绍了并发事务的交替执行中没有并发控制时会产生的三个问题，即：丢失更新、未提交依赖和不一致分析。产生这些问题的调度都不是可以串行化的，即不能等价于涉及相同事务的某个串行调度。

解决这些问题的最广泛的技术是锁。锁的基本类型有两种，即共享锁（S锁）和排它锁（X锁）。如果某个事务在某个对象上施加了S锁，则其他事务也能在该对象上申请S锁，但不能对该对象申请X锁；如果某个事务在某个对象上施加了X锁，则其他事务不能对该对象申请如何锁。为了保证不发生丢失修改，可以引入这样一个锁协议：对任何要检索的对象申请S锁，对任何要更新的对象申请X锁，并将锁保持到事务结束。该协议可以实现事务的串行化。

上面所描述的协议是两段锁协议的一个较强但很常见的形式。由此得出两段锁定理：如果所有的事务都遵循该协议，那么所有的调度都是可串行的。可串行调度意味着：如果*A*和*B*是该调度所涉及的两个事务，那么或者*A*看到*B*的输出，或者*B*看到*A*的输出。遗憾的是，两段锁协议可能导致死锁的发生。打破死锁的办法是选择死锁事务中的某一个事务作为牺牲者（victim），然后将该事务回滚（从而释放牺牲者所锁定的全部资源）。

一般说来，如果事务未达到完全可串行性，则不能保证一定安全。但系统一般都允许事务在某个实际上并不安全的隔离级别上运行，这样可以减少资源的竞争并提高事务吞吐量。本章将这样一个“不安全”级别描述为游标稳定性（这是DB2中的术语，而在SQL中则用READ COMMITTED来表示）。

接下来主要说明锁粒度的问题及相应的意向锁思想。意向锁的基本观点是：事务在某些对象（如数据库中的元组）上获得某种锁前，它必须先获得在该对象的“父对象”上的一个适当的意向锁。例如，对于一个元组则要在包含该元组的关系变量上获得一个意向锁。这样的意向锁通常以隐式申请方式实现，就像元组上的S锁和X锁以隐式方式申请一样。但是也应该提供某些意向锁的显式锁语句以允许事务在必要时能够获得较隐式方式获得的锁

更强的锁。

本章最后给出了SQL对并发控制支持的几个要点。SQL基本上不提供任何的显式锁能力，但它支持多种隔离级别，即：读未提交、读已提交、可重复读和可串行化的等四种。这些不同的隔离级别可以由DBMS通过非显式锁方式加以实现。

对于完整性的重要性问题，这里进一步给出一个简要的说明。在较泛泛地讲数据库正确时，一般指数据库没有违背任何已知的完整性约束。因此容易看出，对于未提供较多完整性支持的系统而言，它只能提供较弱意义上的数据库“正确”。在第14章中提到的恢复指的是恢复到以前的某个已知的“正确状态”，而本章中的并发（或者说并发控制）则指的是一个可串行调度将把数据库从一个“正确状态”转移到另一个“正确状态”。由此可见，完整性是比恢复和并发更为基本的问题。事实上，完整性是单用户系统中也要考虑的问题。

练习

15.1 定义可串行性。

15.2 说明下列概念：

a) 两段锁协议。

b) 两段锁定理。

15.3 给定事务*T1*、*T2*、*T3*、执行下列操作：

T1：将*A*加1；

T2：将*A*加倍；

T3：在屏幕上输出*A*，并将*A*置为1，其中*A*为数据库中的某个数据项。

a) 假设*T1*、*T2*、*T3*可以并发执行。若*A*初值为0，那么存在多少种可能的正确结果？列举之。

b) 各个事务的内部结构如下表所示。若事务执行不施加任何锁，则有多少种可能的调度？

T1	*T2*	*T3*
R1: RETRIEVE A INTO t1 ; t1 := t1 + 1 ; U1: UPDATE A FROM t1 ;	R2: RETRIEVE A INTO t2 ; t2 := t2 * 2 ; U2: UPDATE A FROM t2 ;	R3: RETRIEVE A INTO t3 ; display t3 ; U3: UPDATE A FROM 1 ;

c) 在*A*的初值给定为0时，能够产生“正确”结果但不可串行化的调度吗？

d) 如果这三个事务都遵循2PL(两段锁)协议，那么存在事实上可串行化但又不能形成的调度吗？

15.4 下面给出的是一个调度的事件序列。该调度包含*T1*, *T2*, ..., *T12*等12个事务，A, *B*..., *H*为数据库中的数据项。

```
time t0      ..........
time t1    (T1)    : RETRIEVE A ;
time t2    (T2)    : RETRIEVE B ;
    -      (T1)    : RETRIEVE C ;
    -      (T4)    : RETRIEVE D ;
    -      (T5)    : RETRIEVE A ;
    -      (T2)    : RETRIEVE E ;
    -      (T2)    : UPDATE E ;
    -      (T3)    : RETRIEVE F ;
    -      (T2)    : RETRIEVE F ;
    -      (T5)    : UPDATE A ;
    -      (T1)    : COMMIT ;
```

```
        -      (T6)    : RETRIEVE A ;
        -      (T5)    : ROLLBACK ;
        -      (T6)    : RETRIEVE C ;
        -      (T6)    : UPDATE C ;
        -      (T7)    : RETRIEVE G ;
        -      (T8)    : RETRIEVE H ;
        -      (T9)    : RETRIEVE G ;
        -      (T9)    : UPDATE G ;
        -      (T8)    : RETRIEVE E ;
        -      (T7)    : COMMIT ;
        -      (T9)    : RETRIEVE H ;
        -      (T3)    : RETRIEVE G ;
        -      (T10)   : RETRIEVE A ;
        -      (T9)    : UPDATE H ;
        -      (T6)    : COMMIT ;
        -      (T11)   : RETRIEVE C ;
        -      (T12)   : RETRIEVE D ;
        -      (T12)   : RETRIEVE C ;
        -      (T2)    : UPDATE F ;
        -      (T11)   : UPDATE C ;
        -      (T12)   : RETRIEVE A ;
        -      (T10)   : UPDATE A ;
        -      (T12)   : UPDATE D ;
        -      (T4)    : RETRIEVE G ;
    time tn      ..........
```

假定“RETRIEVE *R*”如果成功，则获得*R*上的一个S锁，而“UPDATE *R*”成功时，则将锁升级为X锁。同时假定所有锁都保持到事务结束。那么在时刻*tn*是否存在死锁？

15.5 重新考虑图15-1~图15-4中提出的并发问题。如果所有事务都在CS隔离级别而不是RR级上运行，会发生什么情况？

15.6 分别给出锁类型X、S、IX、IS和SIX的非形式化和形式化定义。

15.7 定义相对锁强度的概念，并给出相应的前驱图。

15.8 定义意向锁协议。说明该协议的意义。

15.9 SQL中定义了读脏数据、不可重复读和幻影等三个并发问题，它们和15.2节中说明的三个并发问题具有什么样的联系？

15.10 给出文献[15.1]中说明的多版本并发控制协议的一个概要的实现机制。

参考文献和简介

除了下面给出的参考文献和书目外，还可以参阅第14章中的[14.2]、[14.10]，尤其是[14.12]。

15.1 R. Bayer, M. Heller and A. Reiser: “Parallelism and Recovery in Database Systems,” *ACM TODS 5*, No. 2(June 1980).

第14章中提到，在包括硬件和软件工程在内的新兴应用领域中经常要涉及十分复杂的处理要求，而本章大部分及以前各章所描述的传统事务管理控制并不能很好地满足这些需求。其中一个很基本的问题就是复杂事务可能持续几小时甚至几天，而不是传统系统中所认为的几个毫秒，从而导致下面的后果：

1) 将事务完全回滚到事务开始时的状态可能导致不可接受的大量工作丢失。

2) 使用通常的锁可能产生不可接受的过长时延来等待放锁。

本文是阐述这些问题的文章之一（其它的文章可参阅[15.7]、[15.11~15.13]及[15.17]）。文中提出了多版本锁的并发控制技术，或者叫做多版本读技术。该技术在一些商用产品中已经得到实现。该技术的最大优点在于读操作不会等待，即任意数量的读者和一个写者能够在同一个逻辑对象上同时操作。尤其是：

1) 读决不会延迟；

2) 读决不会延迟更新；

3) 完全没有必要回滚只读事务；

4) 死锁只可能在更新事务之间发生。

这些优点在分布系统（见第 22章）中尤为明显，因为在分布系统中的更新可能持续很长一段时间，这时只读事务将被过度毫无意义地延迟（反之亦然）。多版本锁的的基本思想如下：

- 事务*T2*请求读一个对象时，如果事务 *T1*已经获得了在该对象上的更新存取路径，则系统为*T2*提供该对象的一个已提交版本的存取路径。这一版本必须作为恢复之用而保留在系统中（如保存在系统日志里）。
- 事务*T2*请求更新一个对象时，如果事务 *T1*已经获得了在该对象上的读存取路径，则*T2*获得该对象的存取路径。这时 *T1*仍然指向原来的对象版本（此时的对象是一个真正的“以前版本”）。
- 事务*T2*请求更新一个对象时，如果事务 *T1*已经获得了在该对象上的写存取路径，则*T2*进入等待状态[⊖]（如前所述的死锁和强制回滚仍有可能发生）。

15.2 Hal Berenson *et al*.: “A Critique of ANSI SQL Isolation Levels,” Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif.(May 1995).

这篇论文的主要内容是批评 SQL标准用可串行性违背来说明隔离级别的方式。文章认为：“（SQL的）定义不能准确地说明几个常见隔离级别的特点，包括其中涉及的几个级别的标准锁实现”。文章特别指出：标准不能防止写脏数据（写脏数据可以定义为两个事务*T1*和*T2*可能在两者终止前都在同一行上执行了更新）。

对于标准中没有显式地防止写脏数据这一点，看来是正确的。原文说明如下（略有变动）：

- “保证可串行隔离级别上的并发事务的执行是可串行的”。或者说，如果事务都在可串行隔离级别上执行，实现时必须要防止写脏数据，因为写脏数据必将违背可串行性。
- “四个隔离级别保证了……不会丢失更新”。这一说法只是一厢情愿，四个隔离级别的定义本身并未提供这样的保证，但它表明了标准定义者试图防止写脏数据。
- “一个事务所做出的修改直到该事务以提交方式结束时才能为其他事务（读未提交级别的事务除外）所感知”。这里的问题在于，“感知”的准确含义是什么？事务能够更新一个“脏数据”而不“感知”它吗？

注：上述注释摘自文献[4.19]。

15.3 Philip A. Bernstein and Nathan Goodman: “Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems,” Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada(October 1980).

文章讨论了一组基于时标而不是锁的并发控制方法。基本思想是：如果事务 *A*在事务*B*之前启动执行，则系统将从整体上把 *A*视为在*B*启动前执行完，也就是通常的串行调度。

⊖ 换言之，更新/更新冲突仍可能存在，这里假定使用锁来解决这一冲突。也可以使用时标 [15.3]技术来取代锁。

这样*A*无权看到*B*所作的任何更新。类似地，也决不允许*A*更新任何*B*已经看到的对象。这一控制方法按下面的方式实现。对任何给定的数据库请求，系统将把请求事务的时标和对所请求元组进行最近一次检索或更新的事务的时标相比较，如果发生冲突，则将发出请求的事务简单地重新启动，并指派一个新的时标（正如[15.14]中的所谓乐观方法）。

正如论文标题所隐含的那样，时标方法最初是在分布式环境中引入的。原因在于人们感到分布式系统中使用锁会由于检测消息、置时钟等原因而产生不可忍受的系统开销。当然，在一个非分布式系统中它并非很合适。事实上它在分布式系统中的可行性仍然值得怀疑。一个明显的问题是：每个元组必须保留最近检索（或更新）过它的事务的时标，这意味着每一个读转变为一个写操作！事实上，文献[14.12]中已经指出：时标模式恰恰是[15.14]中的乐观并发控制模式的一个退化方案，而乐观控制模式本身也具有自身的问题。

15.4 M. W. Blasgen, J. N. Gray, M. Mitoma and T. G. Price: “The Convoy Phenomenon,” *ACM Operating Systems Review 13*, No.2(April 1979).

护航现象（convoy phenomenon）是抢占式调度系统中使用高流量（high-traffic）锁时所遇到的问题。例如，向日志中写记录时所需要的锁就是一个高流量锁。注意，这里的“调度”是指为事务分配机器周期的问题，而不是本章中主要讨论的不同事务的数据库操作的交叉问题。

问题如下：如果事务*T*正持有一个高流量锁且这时被系统调度器所强占，如可能因时间片到期而被强制进入等待状态，那么系统将为那些等待获得高流量锁的事务形成一个护航。当*T*离开等待状态时，它将很快释放该高流量锁，但由于该锁是高流量的，*T*本身也可能在下一个事务未用完资源前而加入到护航中，因此不能继续处理下去，进而又进入一个等待状态。

问题的本质在于：大多数情况下（而非全部），调度器是底层OS而不是DBMS的一部分，因此它是基于另外一个不同的假设来设计的。作者观察到：护航一旦建立，将趋于稳定；系统将处于一个“锁颠簸”状态，这时大部分机器周期都用于处理切换而未做什么有意义的工作。一个不替换调度器的建议方案是将锁的授权方式不再基于“先来先服务”的原则，而采用随机方式。

15.5 K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger: “The Notions of Consistency and Predicate Locks in a Data Base System,” *CACM 19*，No. 11(November 1976).

该文首次将并发控制这一课题建立在一个合理的理论之上。

15.6 Peter Franaszek, and John T. Robinson: “Limitations on Concurrency in Transaction Processing,” *ACM TODS 10*, No.1(March 1985).

见文献[15.14]中的说明。

15.7 Peter A. Franaszek, John T. Robinson, and Alexander Thomasian: “Concurrency Control for High Contention Environments,” *ACM TODS 17*, No.2(June 1992).

该文指出，因为各种原因，未来的事务处理系统所面临的并发度情况可能大大超出目前系统所支持的并发度，因此在这些系统中可能发生大量的数据竞争。作者随之提出了“大量非锁并发控制概念和可用于高竞争环境中的事务调度技术”。据称这些技术是基于使用了模拟模型的实验的，并能够在这些环境中“带来实质性好处”。

15.8 J. N. Gray: “Experience with the System R Lock Manager,” IBM San Jose Research Laboratory internal memo (Spring 1980).

它实际上只是一些注释性说明，而不是一篇完整的论文，其中的某些思想如今可能有些过时了，但它也包括了一些有趣的话题，如：

- 锁在联机事务中带来10%的代价，而在批处理事务中只占1%。
- 支持多种锁粒度是值得的。
- 自动锁升级能够很好地工作。
- 实际情况中的死锁很少发生且决不会超过两个事务。几乎所有的死锁（超过 97%）能够通过支持U锁避免。U锁在DB2中得到支持而系统R则不支持（U锁定义为与S锁兼容而不能和U锁兼容，当然也不能和X锁兼容。细节可参阅文献[4.20]）。
- 可重复读（RR）比游标稳定性（CS）有效也更安全。

15.9 J. N. Gray, R. A. Lorie and G. R. Putzolu: “Granularity of Locks in a Large Shared Data Base,” Proc. 1st Int. Conf. on Very Large Data Bases, Framingham, Mass. (September 1975).

这篇论文引入了意向锁概念。和15.8中的解释一样，术语“粒度”是指能够被锁定的对象的大小。由于不同事务显然具有不同的特点和不同的要求，系统提供某个范围内的多种不同的锁粒度是值得的，而且大多数实际系统也是这样做的。这篇文章给出了基于意向锁的多粒度系统的一种实现机制。

由于本章中给出的解释显然有些简单，因此这里将给出关于意向锁协议的较详细的说明。首先，像前面假设的那样，可加锁对象类型不仅限于关系变量或元组。其次，这些锁对象甚至不必形成一个严格的层次；索引和其它存取结构的表现方式意味着它们应被视作有向无环图DAG。例如，供应商和零件数据库可能包含零件关系变量P（的某种存储形式）和一个建立在P#属性上的索引XP。如果要获得关系变量P中的元组，则必须先启动整个数据库，然后要么直接在关系变量中顺序扫描，要么先进入XP，然后再找到所需的元组。这样在相应的DAG中P有两个“父对象”，P和XP，而这两个对象都以数据库作为它的“父对象”。

下面给出协议的一般形式。

- 获得指定对象上的X锁时，隐式获得该对象的所有子对象上的X锁。
- 获得指定对象上的S或SIX锁时，隐式获得该对象的所有子对象上的S锁。
- 在事务获得指定对象上的S或IS锁前，它必须先获得该对象至少一个父对象上的IS（或更强的）锁。
- 在事务获得指定对象上的X、IX或SIX锁前，它必须先获得该对象的所有父对象上的IX（或更强的）锁。
- 在事务能释放某个指定对象上的一个锁前，它必须先释放掉它在所有子对象上保持的锁。

在实际应用中，该协议不会产生想像中那样的系统开销。原因在于在任何给定时刻，事务可能已经获得了所需的大多数锁。例如，在整个数据库上的一个IX锁可能只需要在程序初始启动时申请，然后该锁在程序的这个生存期内的所有事务中得到保持。

15.10 J. N. Gray, R. A. Lorie, G. R. Putzolu and I. L. Traiger: “Granularity of Locks and

Degrees of Consistency in a Shared Data Base," in G. M. Nijssen(ed.), *Proc. IFIF TC-2 Working Conf. on Modelling in Data Base Management Systems*. Amsterdam, Netherlands: North-holland/New York, N.Y.: Elsevier Science (1976).

这篇论文用一致性程度(degree of consistency)的形式介绍了隔离级别的概念。

15.11 Theo Härder and Kurt Rothermel: "Concurrency Control Issues in Nested Transactions," *The VLDB Journal 2*, No.1 (January 1993).

在第14章中提到，有些作者给出了关于嵌套事务思想的建议。这篇文章提出了适用于这些事务的一套较为恰当的锁协议。

15.12 J. R. Jordan, J. Banerjee, and R. B. Batman: "Precision Locks," Proc. 1981 ACMSIGMOD Int. Conf. on Management of Data, Ann Arbor, Mich. (April/May 1981).

精确锁是一种元组级的锁模式，它保证了只有那些需要加锁的元组才被真正锁定以满足可串行性，这些锁定元组也包括幻影。它实际上是 [15.5]的谓词锁的一种形式。它的工作机理是：(a) 检查更新请求，看要插入的元组是否满足其它并发事务较早发出的检索请求；(b) 检查检索请求，看其他并发事务已经插入或删除的元组是否满足查询中的检索请求。该模式不但精巧，而且作者同时还声称它实际上也比传统技术（一般有太多的锁）运行得更好。

15.13 Henry F. Korth and Greg Speegle: "Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model," *ACM TODS 19*, No.3 (September 1994).

正如文献[14.3]、[14.9]和[14.15~14.16]中所说明的，可串行性通常要求较严格的条件而无法在某些系统中实施。尤其在涉及人机交互和长周期事务的新兴应用领域中更是如此。这篇文章提出了一个叫做 NT/PV ("带谓词和视图的嵌套事务")的事务模型来阐述这些问题。此外，文章提出：具有可串行性的标准事务模型只是一个特例。文中定义了"新的且更具意义的正确性类"，并声称新的模型提供了"一个可解决长周期事务问题的合理的框架"。

15.14 H. T. Hung and John T. Robinson: "On optimistic Methods for Concurrency Control," *ACM TODS 6*, No.2 (June 1981).

锁模型可以采用悲观模式。这时，锁采用了最坏情况假设，即某指定事务所存取的每个数据都可能被其它的并发事务所申请，因此最好加以锁定。对应的，乐观模式（也可称做认证或校验模式）则采用了相反的假设，它认为冲突实际上很少发生。因此，乐观模式允许事务毫无阻碍地运行到结束。然后在提交时检查是否真的发生了冲突，如果有，则将不合理的事务简单地重新启动。在提交处理成功完成前，不能将任何更新写入数据库中，这样，重启事务时不需要对任何更新进行 UNDO处理。

在随后的论文[15.6]中，作者提出：在某些合理假设下，乐观方法在所支持的并发度（以同时执行的事务数目计）期望级别上比传统锁方法具有某些固有的优点，这意味着在拥有大量并行处理器的系统中，乐观方法可能更值得选择（但文献 [14.12]却认为乐观方法在"热点"条件下一般比锁方法更糟。热点是指被多个事务频繁更新的数据项。对于在热点上工作得较好的技术的讨论可参见文献 [15.15])。

15.15 Patrick E. O' Neil: " The Escrow Transactional Method," *ACM TODS 11*, No.4 (December

1986).

考虑下面的简单例子。设数据库中的数据项 *TC*表示“目前总现金数”，并假设系统中几乎所有事务都从 *TC*中减少一定数量（或者叫提取现金）来更新 *TC*，则*TC*是一个“热点”的例子，也就是说，当数据库中的数据项被系统中运行的很大比例的事务所存取时，它被称作“热点”。在传统锁下，热点很快成为一个“瓶颈”。因此，对 *TC*这样的数据项使用传统锁就显得强度太大。如果 *TC*初始值为一千万美元，而每个事务平均只减去十美元，则可以执行约一百万个这样的事务，这样在出现问题前可以按任意次序进行一百万次相应的缩减。对于 *TC*而言，不再需要传统锁，代之以只要确保当前值足以允许所需的减少量，就进行更新（如果随后事务失败了，则将减去的数量加回去）。

ESCROW方法适用前面所描述的情况，在这种情况下，更新是一种特殊形式，可以是完全随意的。系统必须提供一种新的更新语句（如“减去 *x*，当且仅当当前值大于 *y*”）。该语句通过将减少量放入“约定(escrow)”中执行更新，在事务结束时将该语句从“约定”中取出（如果事务以COMMIT结束，则提交修改；否则，在以ROLLBACK结束时，则将各数量放回原处）。

论文中描述了可使用ESCROW方法的大量示例。IBM的IMS Fast Path Version支持该技术。这一技术可以作为 [15.14]中的乐观并发控制的一个特例（但需注意，对提供特殊更新语句这一“特殊”特点是有争议的）。

15.16 Christos Papadimitriou: *The Theory of Database Concurrency Control*. Rockville, Md.: Computer Science Press (1986).

侧重于形式化理论的一本教科书。

15.17 Kenneth Salem, Hector Garcia-Molina and Jeannie Shands: “Altruistic Locking,” *ACM TODS 19*, No.1 (March 1994).

给出了一个两段锁的扩展协议。基于该扩展协议，如果事务 *T1*已经用完某些锁定的数据但根据2PL协议又不能释放数据上的锁时，*T1*可以将数据“捐献”给系统，进而允许其它事务获得在这些数据上的锁。这时称 *T2*“跟随(in the wake of)”*T1*，而所定义的协议用来阻止一个事务看到它的跟随事务所做的任何更新。利它锁（该术语源于“捐献”数据而使其他事务受益，而不是捐献者事务）能够提供较常规 2PL更大的并发度。

部分练习答案

15.3 a. 有6种可能的正确结果，相应的6个串行调度是：

```
Initially : A = 0
T1-T2-T3  : A = 1
T1-T3-T2  : A = 2
T2-T1-T3  : A = 1
T2-T3-T1  : A = 2
T3-T1-T2  : A = 4
T3-T2-T1  : A = 3
```

当然这6个可能的正确结果并非完全不同。事实上，在这个特殊例子中，由于 *T3*的性质，这些可能的正确结果与数据库的初始状态无关。

b. 有90个可能的不同调度。下面给出各种可能。其中 *Ri*、*Rj*、*Rk*分别表示三种检索操作 *R1*、*R2*、*R3*，顺序无关紧要；类似地，*Up*、*Uq*、*Ur*则分别表示更新操作 *U1*、*U2*、

U3，顺序也无关紧要。

```
Ri-Rj-Rk-Up-Uq-Ur : 3 * 2 * 1 * 3 * 2 * 1 = 36 possibilities
Ri-Rj-Up-Rk-Uq-Ur : 3 * 2 * 2 * 1 * 2 * 1 = 24 possibilities
Ri-Rj-Up-Uq-Rk-Ur : 3 * 2 * 2 * 1 * 1 * 1 = 12 possibilities
Ri-Up-Rj-Rk-Uq-Ur : 3 * 1 * 2 * 1 * 2 * 1 = 12 possibilities
Ri-Up-Rj-Uq-Rk-Ur : 3 * 1 * 2 * 1 * 1 * 1 =  6 possibilities
                                    TOTAL = 90 combinations
```

c. 是。例如，调度 *R1-R2-R3-U3-U2-U1*得到的结果与 6个可能的串行调度中的两个相同（练习：核实这一句），这样，对于给定的初始值 0恰好是“正确的”。但必须清楚这个“正确”只是凑巧，完全是因为初始值恰好为 0才得到的，对其他初始值则未必。作为反例，考虑初始值为 10而非0时的情况，上面给出的调度 *R1-R2-R3-U3-U2-U1*仍能够给出正确结果中的某一个吗？（这时真正的正确结果是什么？）如果不是则该调度是不可串行的。

d. 是。例如，调度*R1-R3-U1-U3-R2-U2*是可串行的（等价于串行调度 *T1-T3-T2*），但在*T1*、*T2*和*T3*都遵循2PL协议时不能出现。因为在这一协议下，操作 *R3*会代表事务 *T3*在*A*上获得*S*锁；这样事务 *T1*中的操作*U1*就无法在该锁释放前进行，且直到 *T3*结束时才会发生。事实上，事务*T3*和*T1*在操作*U3*到达时将发生死锁。

这个练习的重点是：对于给定的一个事务集合和一个数据库初态，（a）置*ALL*为涉及所有事务的全部可能调度的集合；（ b）置“ *CORRECT* ”为能保证产生准确终态或者从给定初态碰巧能够到达终态的调度集合；（ c）置*SERIALIZABLE*为所有可串行化的调度集；（d）置*PRODUCIBLE*为*2PL*协议下可产生的调度集，则一般有：

PRODUCIBLE≤ SERIALIZABLE≤ “CORRECT” ≤ ALL

这里的“ ≤ ”表示“……的子集”。

15.4 在*tn*时刻不会有任何事务执行有意义的工作！死锁涉及的事务有 *T2*、*T3*、*T9*和*T8*；另外，*T4*等待*T9*，*T12*等待*T4*，而*T10*和*T11*都在等待*T12*。这时的情形可以用等待图（WFG）的形式给出。等待图中的节点表示事务， *Ti*到*Tj*的有向边表示 *Ti*等待*Tj*（见图15-14）。边上的标识为数据项名和正等待的锁的级别。

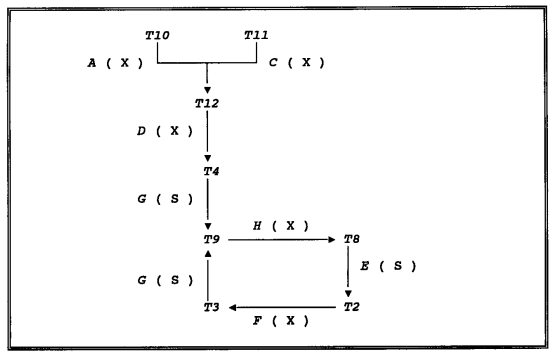

图15-14 练习15.4的等待图

15.5 隔离级别CS和RR在图15-1~图15-3中的问题上有相同的效果（但要注意，这句话不适用于DB2中实现的CS，因为DB2在[4.20]中提到的S锁处用的是U锁）。至于图15-4中的不一致分析问题，隔离级别 CS不能解决；事务 *A*必须在RR下执行，并将锁保持到事务结

束。但对其他问题仍将得到错误答案（当然也有其他方式。如果系统支持显式锁，则 *A* 可通过某个显式锁请求来锁定整个帐户关系变量。这时的情形在 CS和RR两个级别上都能工作）。

15.6 见15.8节。尤其要注意：形式化定义是以锁类型的兼容矩阵的形式给出的（见图 15-11）。

15.9 在15.2节中的三个并发问题是：丢失更新、未提交依赖和不一致分析。在这三者中：

- 丢失更新：SQL实现要求能够保证在任何情况下丢失更新决不会发生。
- 未提交依赖：仅仅是读脏数据的另一个名字。
- 不一致分析：这个术语包含了不可重复读和幻影。

15.10 下面的简要描述摘自文献[20.15]。最重要的是，系统必须保持：

1) 一个用于每个数据对象的已提交版本的栈。栈的项为对象的值和建立该值的事务的ID，也就是栈中每个项实际上包含一个指向日志中相应项的指针。这个栈为反时序序列，最近的项在栈顶。

2) 一个所有已提交事务的事务ID列表（或称作提交列表）。

当事务启动执行时，系统给事务一个提交列表的私有拷贝。在对象上的读操作指向私有列表上的事务所生成的最近对象版本。相应地，对于写操作，则指向实际对象（这也正是更新/更新冲突检测仍有必要的原因）。在事务提交时，系统更新提交列表，并适当更新数据对象的版本栈。