并发控制

在第14章中我们了解了事务最基本的特性之一是隔离性。然而，当数据库中有多个事务并发执行时，事务的隔离性不一定能保持。为保持事务的隔离性，系统必须对并发事务之间的相互作用加以控制；这种控制是通过一系列机制中的一个称为*并发控制*的机制来实现的。第26章讨论允许非可串行化调度的并发控制机制。在这一章中，我们考虑并发执行事务的管理，并且我们忽略故障。第16章将讨论系统如何从故障中恢复。

诚如我们将会看到的，并发控制有许多种机制。没有哪种机制是明显最好的；每种机制都有优势。在实践中，最常用的机制有*两阶段封锁*和*快照隔离*。

15.1 基于锁的协议

确保隔离性的方法之一是要求对数据项以互斥的方式进行访问；换句话说，当一个事务访问某个数据项时，其他任何事务都不能修改该数据项。实现该需求最常用的方法是只允许事务访问当前该事务持有锁(lock)的数据项。14.9节介绍过锁的概念。

15.1.1 锁

给数据项加锁的方式有多种，在这一节中，我们只考虑两种：

1. **共享的**(shared)：如果事务 T_i 获得了数据项 Q 上的**共享型锁**(shared-mode lock)(记为S)，则 T_i 可读但不能写 Q。
2. **排他的**(exclusive)：如果事务 T_i 获得了数据项 Q 上的**排他型锁**(exclusive-mode lock)(记为X)，则 T_i 既可读又可写 Q。

我们要求每个事务都要根据自己将对数据项 Q 进行的操作类型**申请**(request)适当的锁。该事务将请求发送给并发控制管理器。事务只有在并发控制管理器**授予**(grant)所需锁后才能继续其操作。这两种锁类型的使用可以让多个事务读取一个数据项但是限制同时只能有一个事务进行写操作。

更普遍地说，对于给定的一个锁类型集合，我们可在它们上按如下方式定义一个**相容函数**(compatibility function)：令 A 与 B 代表任意的锁类型，假设事务 T_i 请求对数据项 Q 加 A 类型锁，而事务 T_j($T_i \neq T_j$)当前在数据项 Q 上拥有 B 类型锁。尽管数据项 Q 上存在 B 类型锁，如果事务 T_i 可以立即获得数据项 Q 上的锁，则我们就说 A 类型锁与 B 类型锁是**相容的**(compatible)。这样的一个函数可以通过矩阵方便地表示出来。本节所用的两类锁的相容关系由图15-1所示的矩阵comp给出。当且仅当类型 A 与类型 B 是相容的，该矩阵的一个元素 **comp**(A, B)具有 *true* 值。

注意共享型与共享型是相容的，而与排他型不相容。在任何时候，一个具体的数据项上可同时有(被不同的事务持有的)多个共享锁。此后的排他锁请求必须一直等待直到该数据项上的所有共享锁被释放。

	S	X
S	true	false
X	false	false

图15-1 锁相容性矩阵comp

一个事务通过执行 **lock-S**(Q)指令来申请数据项 Q 上的共享锁。类似地，一个事务通过执行 **lock-X**(Q)指令来申请排他锁。一个事务能够通过 **unlock**(Q)指令来释放数据项 Q 上的锁。

要访问一个数据项，事务 T_i 必须首先给该数据项加锁。如果该数据项已被另一事务加上了不相容类型的锁，则在所有其他事务持有的不相容类型锁被释放之前，并发控制管理器不会授予锁。因此，T_i 只好**等待**(wait)，直到所有其他事务持有的不相容类型锁被释放。

事务 T_i 可以释放先前加在某个数据项上的锁。注意，一个事务只要还在访问数据项，它就必须拥有该数据项上的锁。此外，让事务在对数据项作最后一次访问后立即释放该数据项上的锁也未必是可取的，因为有可能不能保证可串行性。

举一个例子，再考虑我们在第 14 章中介绍的银行业务系统。令 A 与 B 是事务 T_1 与 T_2 访问的两个账户，事务 T_1 从账户 B 转 50 美元到账户 A 上(见图 15-2)，事务 T_2 显示账户 A 与 B 上的总金额，即 $A+B$(见图 15-3)。

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

图 15-2 事务 T_1

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

图 15-3 事务 T_2

假设账户 A 与 B 的金额分别为 100 美元与 200 美元。如果这两个事务串行执行，以 T_1、T_2 或 T_2、T_1 的顺序执行，则事务 T_2 将显示的值为 300 美元。然而，如果两个事务并发地执行，则有可能出现如图 15-4 所示的调度 1。在这种情况下，事务 T_2 显示 250 美元，这是不对的。出现这种错误的原因是由于事务 T_1 过早释放数据项 B 上的锁，从而导致事务 T_2 看到一个不一致的状态。

T_1	T_2	并发控制管理器
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

图 15-4 调度 1

该调度显示了事务执行的动作以及并发控制管理器授权加锁的时刻。申请加锁的事务在并发控制管理器授权加锁之前不能执行下一个动作。因此，锁的授予必然是在事务申请锁操作与事务的下一动作的间隔内。至于在此期间内授权加锁的准确时间并不重要；我们不妨假设锁正好在事务的下一动作前获得。因此，在本章余下部分所描述的调度中我们将去掉并发控制管理器授予锁的那一栏。我们让读者自己去推断何时授予锁。

现在假定事务结束后才释放锁。事务 T_3 对应于 T_1，它延迟了锁释放(见图 15-5)，事务 T_4 对应于 T_2，它延迟了锁释放(见图 15-6)。

```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

图 15-5 事务 T_3
(事务 T_1 延迟释放锁)

你可以验证，在调度1中导致显示不准确总和250美元的读写顺序，在事务 T_3 与 T_4 的调度中没有再出现的可能。我们也可以用别的一些调度，在任何调度当中，T_4 将不会显示不一致的结果；稍后我们将明白其中缘由。

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

图15-6 事务 T_4（事务 T_2 延迟释放锁）

遗憾的是，封锁可能导致一种不受欢迎的情形。考察如图15-7所示的有关事务 T_3 与 T_4 的部分调度，由于 T_3 在 B 上拥有排他锁，而 T_4 正在申请 B 上的共享锁，因此 T_4 等待 T_3 释放 B 上的锁；类似地，由于 T_4 在 A 上拥有共享锁，而 T_3 正在申请 A 上的排他锁，因此 T_3 等待 T_4 释放 A 上的锁。于是，我们进入了这样一种哪个事务都不能正常执行的状态，这种情形称为**死锁**(deadlock)。当死锁发生时，系统必须回滚两个事务中的一个。一旦某个事务回滚，该事务锁住的数据项就被解锁，其他事务就可以访问这些数据项，继续自己的执行。15.2节将再讨论死锁处理这个问题。

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

图15-7 调度2

我们如果不使用封锁，或者我们对数据项进行读写之后立即解锁，那么我们可能会进入不一致的状态。另一方面，如果在申请对另一数据项加锁之前如果我们不对当前锁住的数据项解锁，则可能会发生死锁。在某些情形下有办法避免死锁，我们将在15.1.5节讨论。然而，一般而言，如果我们为了避免不一致状态而采取封锁，则死锁是随之而来的必然产物。产生死锁显然比产生不一致状态要好，因为它们可以通过回滚事务加以解决，而不一致状态可能引起现实中的问题，这是数据库系统不能处理的。

我们将要求在系统中的每一个事务遵从称为**封锁协议**(locking protocol)的一组规则，这些规则规定事务何时对数据项们进行加锁、解锁。封锁协议限制了可能的调度数目。这些调度组成的集合是所有可能的可串行化调度的一个真子集。我们将讲述几个封锁协议，它们只允许冲突可串行化调度，从而保证隔离性。在此之前，我们需要先给出几个定义。

令 $\{T_0, T_1, \cdots, T_n\}$ 是参与调度 S 的一个事务集，如果存在数据项 Q，使得 T_i 在 Q 上持有 A 型锁，后来，T_j 在 Q 上持有 B 型锁，且 $\mathbf{comp}(A, B) = \text{false}$，则我们称在 S 中 T_i **先于**(precede) T_j，记为 $T_i \rightarrow T_j$。如果 $T_i \rightarrow T_j$，这一居先意味着在任何等价的串行调度中，T_i 必须出现在 T_j 之前。注意这个图与14.6节中用于检测冲突可串行性的优先图是类似的。指令之间的冲突对应于锁类型之间的不相容性。

如果调度 S 是那些遵从封锁协议规则的事务集的可能调度之一，我们称调度 S 在给定的封锁协议下是**合法的**(legal)。当且仅当其所有合法的调度为冲突可串行化时，我们称一个封锁协议**保证**(ensure)冲突可串行性；换句话说，对于任何合法的调度，其关联的→关系是无环的。

15.1.2 锁的授予

当事务申请对一个数据项加某一类型锁，且没有其他事务在该数据项上加上了与此类型相冲突的锁，则可以授予锁。然而，必须小心防止出现下面的情形。假设事务 T_2 在一数据项上持有共享锁，另一事务 T_1 申请在该数据项加排他锁。显然，事务 T_1 必须等待事务 T_2 释放共享锁。同时，事务 T_3 可能申请对该数据项加共享锁，加锁请求与已授予 T_2 的锁是相容的，因此可以授权 T_3 加共享锁。此时，T_2 可能释放锁，但 T_1 还必须等待 T_3 完成。可是，可能又有一个新的事务 T_4 申请对该数据项加共享锁，并在 T_3 完成之前授予锁。事实上，有可能存在一个事务序列，其中每个事务申请对该数据项加共享锁，每个事务在授权加锁后一小段时间内释放锁，而 T_1 总是不能在该数据项上加排他锁。事务 T_1 可能永远不能取得进展，这称为**饿死**(starved)。

我们可以通过按如下方式授权加锁来避免事务饿死：当事务 T_i 申请对数据项 Q 加 M 型锁时，并发控制管理器授权加锁的条件是：

1. 不存在在数据项 Q 上持有与 M 型锁冲突的锁的其他事务。
2. 不存在等待对数据项 Q 加锁且先于 T_i 申请加锁的事务。

因此，一个加锁请求就不会被其后的加锁申请阻塞。

15.1.3 两阶段封锁协议

保证可串行性的一个协议是**两阶段封锁协议**(two-phase locking protocol)。该协议要求每个事务分两个阶段提出加锁和解锁申请。

1. **增长阶段**(growing phase):事务可以获得锁,但不能释放锁。
2. **缩减阶段**(shrinking phase):事务可以释放锁,但不能获得新锁。

最初,事务处于增长阶段,事务根据需要获得锁。一旦该事务释放了锁,它就进入了缩减阶段,并且不能再发出加锁请求。

例如,事务 T_3 与 T_4 是两阶段的。与此相反,事务 T_1 与 T_2 不是两阶段的。注意,解锁指令不必非得出现在事务末尾。例如,就事务 T_3 而言,我们可以把 **unlock**(B)指令放在紧跟指令 **lock-X**(A)之后,并且仍然保持两阶段封锁特性。

我们可以证明两阶段封锁协议保证冲突可串行化。对于任何事务,在调度中该事务获得其最后加锁的位置(增长阶段结束点)称为事务的**封锁点**(lock point)。这样,多个事务可以根据它们的封锁点进行排序,实际上,这个顺序就是事务的一个可串行化顺序。我们将此证明留为习题(见实践习题15.1)。

两阶段封锁并不保证不会发生死锁。不难发现事务 T_3 与 T_4 是两阶段的,但在调度2中(如图15-7所示),它们却发生死锁。

回想一下,在14.7.2节中,除了调度可串行化外,调度还应该是无级联的。在两阶段封锁协议下,级联回滚可能发生。作为示例,考虑如图15-8所示的部分调度。每个事务都遵循两阶段封锁协议,但在事务 T_7 的 **read**(A)步骤之后事务 T_5 发生故障,从而导致 T_6 与 T_7 级联回滚。

级联回滚可以通过将两阶段封锁修改为**严格两阶段封锁协议**(strict two-phase locking protocol)加以
667 避免。这个协议除了要求封锁是两阶段之外,还要求事务持有的所有排他锁必须在事务提交后方可释放。这个要求保证未提交事务所写的任何数据在该事务提交之前均以排他方式加锁,防止其他事务读这些数据。

另一个两阶段封锁的变体是**强两阶段封锁协议**(rigorous two-phase locking protocol),它要求事务提交之前不得释放任何锁。我们很容易验证在强两阶段封锁条件下,事务可以按其提交的顺序串行化。

考察下面两个事务,我们只写出了其中较为重要的 **read** 与 **write** 指令:

T_8: **read**(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: **read**(a_1);
read(a_2);
display($a_1 + a_2$).

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

图15-8 在两阶段封锁下的部分调度

如果我们采用两阶段封锁协议,则 T_8 必须对 a_1 加排他锁。因此,两个事务的任何并发执行方式都相当于串行执行。然而,请注意,T_8 仅在其执行结束,写 a_1 时需要对 a_1 加排他锁。因此,如果 T_8 开始时对 a_1 加共享锁,然后在需要时将其变更为排他锁,那么我们可以获得更高的并发度,因为 T_8 与 T_9 可以同时访问 a_1 与 a_2。

以上观察提示我们对基本的两阶段封锁协议加以修改,使之允许**锁转换**(lock conversion)。我们将提供一种将共享锁升级为排他锁,以及将排他锁降级为共享锁的机制。我们用**升级**(upgrade)表示从共享到排他的转换,用**降级**(downgrade)表示从排他到共享的转换。锁转换不能随意进行。锁升级只能发
668 生在增长阶段,而锁降级只能发生在缩减阶段。

回到我们的例子,事务 T_8 与 T_9 可在修改后的两阶段封锁协议下并发执行,如图15-9所示,其中的调度是不完整的,它只给出了一些封锁指令。

注意，事务试图升级数据项 Q 上的锁时可能不得不等待。这种情况发生在另一个事务当前对 Q 持有共享锁时。

像基本的两阶段封锁协议一样，具有锁转换的两阶段封锁协议只产生冲突可串行化的调度，并且事务可以根据其封锁点作串行化。此外，如果排他锁直到事务结束时才释放，则调度是无级联的。

对于一个事务集，可能存在某些不能通过两阶段封锁协议得到的冲突可串行化调度。然而，如果要通过非两阶段封锁协议得到冲突可串行化调度，我们或者需要事务的附加信息，或者要求数据库中的数据项集合有一定的结构或顺序。在本章后面当我们考虑其他封锁协议时，我们将看到例子。

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

图 15-9 带有锁转换的不完整调度

严格两阶段封锁与强两阶段封锁(含锁转换)在商用数据库系统中广泛使用。

这里介绍一种简单却广泛使用的机制，它基于来自事务的读、写请求，自动地为事务产生适当的加锁、解锁指令：

- 当事务 T_i 进行 **read**(Q)操作时，系统就产生一条 **lock-S**(Q)指令，该 **read**(Q) 指令紧跟其后。
- 当事务 T_i 进行 **write**(Q)操作时，系统检查 T_i 是否已在 Q 上持有共享锁。若有，则系统发出 **upgrade**(Q)指令，后接 **write**(Q)指令。否则系统发出 **lock-X**(Q)指令，后接 **write**(Q)指令。 669
- 当一个事务提交或中止后，该事务持有的所有锁都被释放。

15.1.4 封锁的实现

锁管理器(lock manager)可以实现为一个过程，它从事务接受消息并反馈消息。锁管理器过程针对锁请求消息返回授予锁消息，或者要求事务回滚的消息(发生死锁时)。解锁消息只需要得到一个确认回答，但可能引发为其他等待事务的授予锁消息。

锁管理器使用以下数据结构：锁管理器为目前已加锁的每个数据项维护一个链表，每一个请求为链表中一条记录，按请求到达的顺序排序。它使用一个以数据项名称为索引的散列表来查找链表中的数据项(如果有的话)；这个表叫做**锁表**(lock table)。一个数据项的链表中每一条记录表示由哪个事务提出的请求，以及它请求什么类型的锁，该记录还表示该请求是否已授予锁。

图 15-10 是一个锁表的示例，该表包含 5 个不同数据项 I4、I7、I23、I44 和 I912 的锁。锁表采用溢出链，因此对于锁表的每一个表项都有一个数据项的链表。每一个数据项都有一个已授予锁或等待授予锁的事务列表，已授予锁的事务用深色阴影方块表示，等待授予锁的事务则用浅色阴影方块表示。为了简化图形我们省略了锁的类型。举个例子，从图 15-10 中可以看到，T23 在数据项 I912 和 I7 上已被授予锁，并且正在等待在 I4 上加锁。

虽然图 15-10 上没有标示出来，但锁表还应当维护一个基于事务标识符的索引，这样它可以有效地确定给定事务持有的锁集。

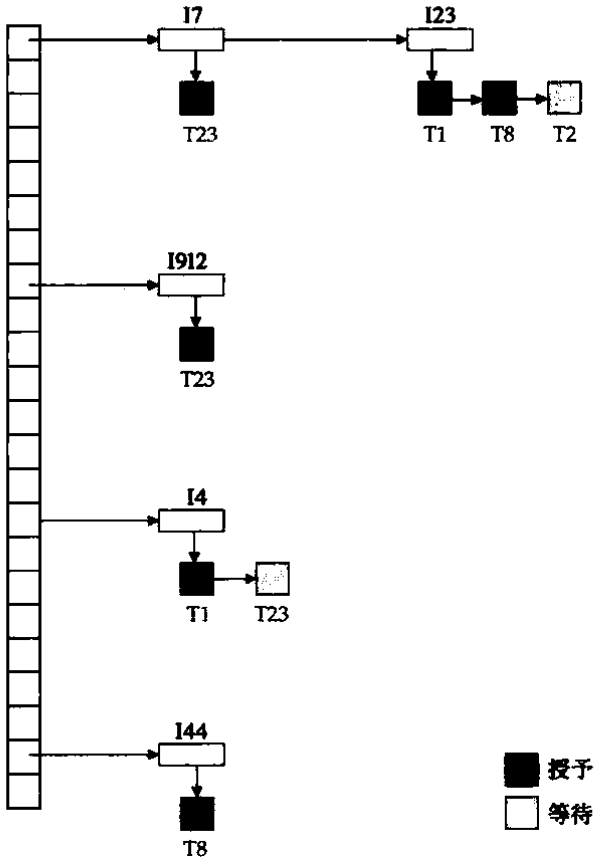

图 15-10 锁表

锁管理器这样处理请求：

- 当一条锁请求消息到达时，如果相应数据项的链表存在，在该链表末尾增加一个记录；否则，新建一个仅包含该请求记录的链表。

  在当前没有加锁的数据项上总是授予第一次加锁请求，但当事务向已被加锁的数据项申请加锁时，只有当该请求与当前持有的锁相容，并且所有先前的请求都已授予锁的条件下，锁管理器才为该请求授予锁，否则，该请求只好等待。
- 当锁管理器收到一个事务的解锁消息时，它将与该事务相对应的数据项链表中的记录删除，然后检查随后的记录，如果有，如前所述，就看该请求能否被授权，如果能，锁管理器授权该请求并处理其后记录，如果还有，类似地一个接一个地处理。
- 如果一个事务中止，锁管理器删除该事务产生的正在等待加锁的所有请求。一旦数据库系统采取适当动作撤销该事务(见 16.3 节)，该中止事务持有的所有锁将被释放。

这个算法保证了锁请求无饿死现象，因为在先前接收到的请求正在等待加锁时，后来者不可能获得授权。我们稍后将在 15.2.2 节学习检测和处理死锁。17.2.1 节阐述另一个实现——在锁申请/授权上利用共享内存取代消息传递。

15.1.5 基于图的协议

正如 15.1.3 节提到的，如果我们要开发非两阶段协议，我们需要有关每个事务如何存取数据库的附加信息。有多种不同模型可以为我们提供这些附加信息，每一种所提供的信息量大小不同。最简单的模型要求我们事先知道访问数据项的顺序。在已知这些信息的情况下，有可能构造出非两阶段的封锁协议，并且无论如何，仍然保证冲突可串行性。

670 ~ 671

为获取这些事先的知识，我们要求所有数据项集合 $\mathbf{D} = \{d_1, d_2, \cdots, d_n\}$ 满足偏序→：如果 $d_i \rightarrow d_j$，则任何既访问 d_i 又访问 d_j 的事务必须首先访问 d_i，然后访问 d_j。这种偏序可以是数据的逻辑或物理组织的结果，也可以只是为了并发控制而加上的。

偏序意味着集合 **D** 可以视为有向无环图，称为**数据库图**(database graph)。在本节中，为了简单起见，我们只关心那些带根树的图。我们将给出一个称为树形协议的简单协议，该协议只使用排他锁。其他更复杂的基于图的封锁协议可以参阅文献注解中给出的有关文献。

在树形协议(tree protocol)中，可用的加锁指令只有 **lock-X**。每个事务 T_i 对一数据项最多能加一次锁，并且必须遵从以下规则：

1. T_i 首次加锁可以对任何数据项进行。
2. 此后，T_i 对数据项 Q 加锁的前提是 T_i 当前持有 Q 的父项上的锁。
3. 对数据项解锁可以随时进行。
4. 数据项被 T_i 加锁并解锁后，T_i 不能再对该数据项加锁。

所有满足树形协议的调度是冲突可串行化的。

为了说明这个协议，考察如图 15-11 所示的数据库图。下面 4 个事务遵从该图的树形协议。我们只列出了加锁、解锁指令：

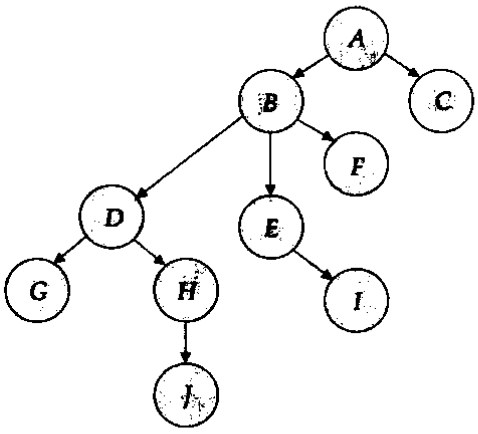

图 15-11 树形结构数据库图

672

T_{10}: **lock-X**(B); **lock-X**(E); **lock-X**(D); **unlock**(B); **unlock**(E); **lock-X**(G); **unlock**(D); **unlock**(G).

T_{11}: **lock-X**(D); **lock-X**(H); **unlock**(D); **unlock**(H).

T_{12}: **lock-X**(B); **lock-X**(E); **unlock**(E); **unlock**(B).

T_{13}: **lock-X**(D); **lock-X**(H); **unlock**(D); **unlock**(H).

这 4 个事务参与的一个可能的调度如图 15-12 所示。注意，在执行时，事务 T_{10} 持有两棵互相分离的子树的锁。

不难发现如图 15-12 所示的调度是冲突可串行化的。可以证明树形协议不仅保证冲突可串行性，而且保证不会产生死锁。

图 15-12 所示的树形协议不保证可恢复性和无级联回滚。为了保证可恢复性和无级联回滚，可以将协议修改为在事务结束前不允许释放排他锁。直到事务结束前一直持有排他锁降低了并发性。这里有一个提高并发性的替代方案，但它只保证可恢复性：为每一个发生了未提交写操作的数据项，我们记录是哪个事务最后对它执行了写操作，当事务 T_i 执行了对未提交数据项的读操作，我们就在最后对该数据项执行了写操作的事务上记录一个 T_i 的**提交依赖**(commit dependency)，在有 T_i 的提交依赖的所有事务提交完成之前，T_i 不得提交。如果其中任何一个事务中止，T_i 也必须中止。

与两阶段封锁协议不同，树形协议不会产生死锁，因此不需要回滚，这一点树形封锁协议要优于两阶段封锁协议。树形封锁协议的另一个优点是可以在较早时候释放锁。较早解锁减少了等待时间，增加了并发性。

然而，在某些情况下，该协议也有其缺点，事务有可能必须给那些根本不访问的数据项加锁。例如，如果某事务要访问图 15-11 所示数据库图中的数据项 A 与 J，则该事务不仅要给 A 与 J 加锁，而且还要给 B、D 与 H 数据项加锁。这种额外的封锁导致封锁开销增加，可能造成额外的等待时间，并且可能引起并发性降低。此外，如果事先没有得到哪些数据项需要加锁的知识，事务将必须给树根加锁，这会大大降低并发性。

对于一个事务集，可能存在某些不能通过树形封锁协议得到的冲突可串行化调度。事实上，一些两阶段封锁协议中可行的调度在树形封锁协议下是不可行的，反之亦然。习题中对这样的调度的例子进行了探讨。

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

图 15-12 在树形协议下的可串行化调度

15.2 死锁处理

如果存在一个事务集，该集合中的每个事务在等待该集合中的另一个事务，那么我们说系统处于死锁状态。更确切地说，存在一个等待事务集$\{T_0, T_1, \cdots T_n\}$，使得 T_0 正等待被 T_1 锁住的数据项，T_1 正等待被 T_2 锁住的数据项，…，T_{n-1}正等待被 T_n 锁住的数据项，且 T_n 正等待被 T_0 锁住的数据项。在这种情况下，没有一个事务能取得进展。

此时，系统唯一的补救措施是采取激烈的动作，如回滚某些陷于死锁的事务。事务有可能只部分回滚，即事务回滚到这样的点之前，它在该点得到一个锁，而释放该锁就可以解决死锁。

处理死锁问题有两种主要的方法。我们可以使用**死锁预防**(deadlock prevention)协议保证系统永不进入死锁状态。另一种方法是，我们允许系统进入死锁状态，然后试着用**死锁检测**(deadlock detection)与**死锁恢复**(deadlock recovery)机制进行恢复。正如我们将要看到的那样，两种方法均有可能引起事务回滚。如果系统进入死锁状态的概率相对较高，则通常使用死锁预防机制；否则，使用检测与恢复机制会更有效。

注意，检测与恢复机制带来各种开销，不仅包括在运行时维护必要信息及执行检测算法的代价，还要包括从死锁中恢复所固有的潜在损失。

673 / 674

15.2.1 死锁预防

预防死锁有两种方法。一种方法是通过对加锁请求进行排序或要求同时获得所有的锁来保证不会发生循环等待。另一种方法比较接近死锁恢复，每当等待有可能导致死锁时，进行事务回滚而不是等待加锁。

第一种方法下最简单的机制要求每个事务在开始之前封锁它的所有数据项。此外，要么一次全部封锁要么全不封锁。这个协议有两个主要的缺点，(1)在事务开始前通常很难预知哪些数据项需要封

锁。(2)数据项使用率可能很低，因为许多数据项可能封锁很长时间却用不到。

防止死锁的另一种机制是对所有的数据项强加一个次序，同时要求事务只能按次序规定的顺序封锁数据项。我们曾经在树形协议中讲述过一个这样的机制，其中采用一个偏序的数据项。

该方法的一个变种是使用数据项与两阶段封锁关联的全序。一旦一个事务锁住了某个特定的数据项，它就不能申请顺序中位于该数据项前面的数据项上的锁。只要在事务开始执行之前，它要访问的数据项集是已知的，该机制就容易实现。如果使用了两阶段封锁，潜在的并发控制系统就不需要更改：所需要的是，保证锁的申请按照正确的顺序。

防止死锁的第二种方法是使用抢占与事务回滚。在抢占机制中，若事务 T_j 所申请的锁已被事务 T_i 持有，则授予 T_i 的锁可能通过回滚事务 T_i 被**抢占**(preempted)，并将锁授予 T_j。为控制抢占，我们给每个事务赋一个唯一的时间戳，系统仅用时间戳来决定事务应当等待还是回滚。并发控制仍使用封锁机制。若一个事务回滚，则该事务重启时保持原有的时间戳。已提出利用时间戳的两种不同的死锁预防机制：

1. **wait-die** 机制基于非抢占技术。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 T_i 的时间戳小于 T_j 的时间戳(即，T_i 比 T_j 老)时，允许 T_i 等待。否则，T_i 回滚(死亡)。

例如，假设事务 T_{14}、T_{15}及 T_{16}的时间戳分别为 5、10 与 15。如果 T_{14}申请的数据项当前被 T_{15}持有，则 T_{14}将等待。如果 T_{16}申请的数据项当前被 T_{15}持有，则 T_{16}将回滚。

2. **wound-wait** 机制基于抢占技术，是与 wait-die 相反的机制。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 T_i 的时间戳大于 T_j 的时间戳(即，T_i 比 T_j 年轻)时，允许 T_i 等待。否则，T_j 回滚(T_j 被
675 T_i伤害)。

回到前面的例子，对于事务 T_{14}、T_{15}及 T_{16}，如果 T_{14}申请的数据项当前被 T_{15}持有，则 T_{14}将从 T_{15}抢占该数据项，T_{15}将回滚。如果 T_{16}申请的数据项当前被 T_{15}持有，则 T_{16}将等待。

两种机制都面临的主要问题是可能发生不必要的回滚。

另一种处理死锁的简单方法基于**锁超时**(lock timeout)。在这种方法中，申请锁的事务至多等待一段给定的时间。若在此期间内未授予该事务锁，则称该事务超时，此时该事务自己回滚并重启。如果确实存在死锁，卷入死锁的一个或多个事务将超时并回滚，允许其他事务继续。该机制介于死锁预防(不会发生死锁)与死锁检测及恢复之间，死锁检测与恢复在 15.2.2 节讲述。

超时机制的实现极其容易，并且如果事务是短事务并且长时间等待很可能由死锁引起的，该机制运作良好。然而，一般而言很难确定一个事务超时之前应等待多长时间。如果已发生死锁，等待时间太长导致不必要的延迟。如果等待时间太短，即便没有死锁，也可能引起事务回滚，造成资源浪费。该机制也可能会产生饿死。因此，基于超时的机制应用有限。

15.2.2 死锁检测与恢复

如果系统没有采用能保证不产生死锁的协议，那么系统必须采用检测与恢复机制。检查系统状态的算法周期性地激活，判断有无死锁发生。如果发生死锁，则系统必须试着从死锁中恢复。为实现这一点，系统必须：

- 维护当前将数据项分配给事务的有关信息，以及任何尚未解决的数据项请求信息。
- 提供一个使用这些信息判断系统是否进入死锁状态的算法。
- 当检测算法判定存在死锁时，从死锁中恢复。

本节详细讲述上述问题。

15.2.2.1 死锁检测

死锁可以用称为**等待图**(wait-for graph)的有向图来精确描述。该图由 $G=(V, E)$对组成，其中 V
676 是顶点集，E 是边集。顶点集由系统中的所有事务组成，边集 E 的每一元素是一个有序对 $T_i \rightarrow T_j$。如果 $T_i \rightarrow T_j$ 属于 E，则存在从事务 T_i 到 T_j 的一条有向边，表示事务 T_i 在等待 T_j 释放所需数据项。

当事务 T_i 申请的数据项当前被 T_j 持有时，边 $T_i \rightarrow T_j$ 被插入等待图中。只有当事务 T_j 不再持有事务 T_i 所需数据项时，这条边才从等待图中删除。

当且仅当等待图包含环时，系统中存在死锁。在该环中的每个事务称为处于死锁状态。要检测死锁，系统需要维护等待图，并周期性地激活一个在等待图中搜索环的算法。

为说明这些概念，考虑图 15-13 所示的等待图，它说明了下面这些情形：

- 事务 T_{17} 在等待事务 T_{18} 与 T_{19}。
- 事务 T_{19} 在等待事务 T_{18}。
- 事务 T_{18} 在等待事务 T_{20}。

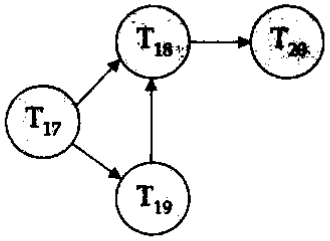

图 15-13 无环等待图

由于该等待图无环，因此系统没有处于死锁状态。

现在假设事务 T_{20} 申请事务 T_{19} 持有的数据项。边 $T_{20}\rightarrow T_{19}$ 被加入等待图中，得到图 15-14 所示的系统新状态。此时，该等待图包含环：

$$T_{18}\rightarrow T_{20}\rightarrow T_{19}\rightarrow T_{18}$$

意味着事务 T_{18}、T_{19} 与 T_{20} 都处于死锁状态。

由此，引出下面的问题：我们何时激活检测算法？答案取决于两个因素：

1. 死锁发生频度怎样？
2. 有多少事务将受到死锁的影响？

如果死锁频繁发生，则检测算法应比通常情况下激活得更频繁。分配给处于死锁状态的事务的数据项在死锁解除之前不能为其他事务获取。此外，等待图中环的数目也可能增大。在最坏的情况下，我们要在每个分配请求不能立即满足时激活检测算法。 677

15.2.2.2 从死锁中恢复

当一个检测算法判定存在死锁时，系统必须从死锁中**恢复**(recover)。解除死锁最通常的做法是回滚一个或多个事务。需采取的动作有三个。

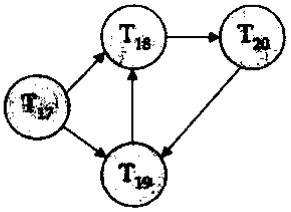

图 15-14 有一个环的等待图

1. **选择牺牲者**：给定处于死锁状态的事务集，为解除死锁，我们必须决定回滚哪一个(或哪一些)事务以打破死锁。我们应使事务回滚带来的代价最小。可惜“最小代价”这个词并不准确。很多因素影响事务回滚代价，其中包括：
   a. 事务已计算了多久，并在完成其指定任务之前该事务还将计算多长时间。
   b. 该事务已使用了多少数据项。
   c. 为完成事务还需使用多少数据项。
   d. 回滚时将牵涉多少事务。
2. **回滚**：一旦我们决定了要回滚哪个事务，就必须决定该事务回滚多远。最简单的方法是**彻底回滚**(total rollback)：中止该事务，然后重新开始。然而，事务只回滚到可以解除死锁处会更有效。这种**部分回滚**(partial rollback)要求系统维护所有正在运行事务的额外状态信息。确切地说，需要记录锁的申请/授予序列和事务执行的更新。死锁检测机制应当确定，为打破死锁，选定的事务需要释放哪些锁。选定的事务必须回滚到获得这些锁的第一个之前，并取消它在此之后的所有动作。恢复机制必须能够处理这种部分回滚。而且，事务必须能够在部分回滚之后恢复执行。有关参考文献见文献注解。 678
3. **饿死**：在系统中，如果选择牺牲者主要基于代价因素，有可能同一事务总是被选为牺牲者。这样一来，该事务总是不能完成其指定任务，这样就发生**饿死**(starvation)。我们必须保证一个事务被选为牺牲者的次数有限(且较少)。最常用的方案是在代价因素中包含回滚次数。

15.3 多粒度

到目前为止所讲的并发控制机制中，我们将一个个数据项作为进行同步执行的单元。

然而，某些情况下需要把多个数据项聚为一组，将它们作为一个同步单元，因为这样会更好。例如，如果事务 T_i 需要访问整个数据库，使用的是一种封锁协议，则事务 T_i 必须给数据库中每个数据项加锁。显然，执行这些加锁操作是很费时的。要是 T_i 能够只发出单个封锁整个数据库的加锁请求，那会更好。另一方面，如果事务 T_j 只需要存取少量数据项，就不应要求给整个数据库加锁，否则并发性就丧失了。

我们需要的是一种允许系统定义**多级粒度**(granularity)的机制。这通过允许各种大小的数据项并定义数据粒度的层次结构，其中小粒度数据项嵌套在大粒度数据项中来实现。这种层次结构可以图形化地表示为树。注意，这里所描述的树与树形协议(见 15.1.5 节)所用的树的概念完全不同。多粒度树中的非叶结点表示与其后代相联系的数据。在树形协议中，每个结点是一个相互独立的数据项。

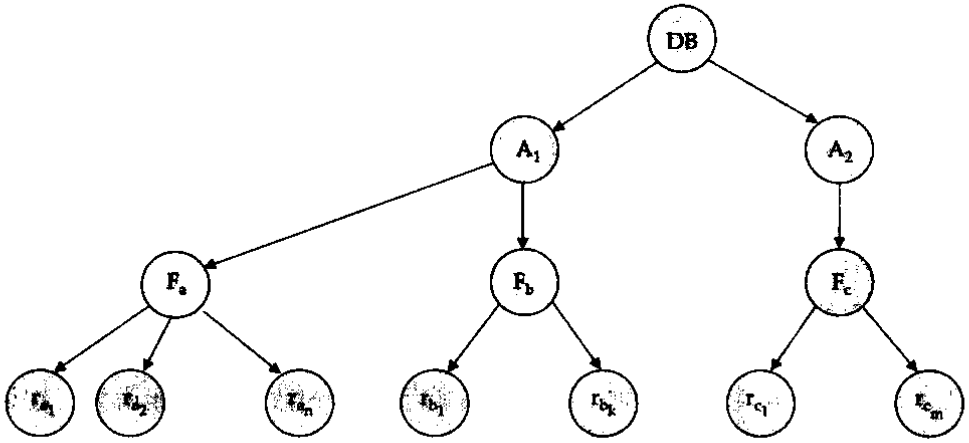

图 15-15 粒度层次图

作为例子，考虑图 15-15 所示的树。它由 4 层结点组成，最高层表示整个数据库，其下是 *area* 类型的结点，数据库恰好由这些区域组成。每个区域又以 *file* 类型结点作为子结点，每个区域恰好由作为其子结点的文件结点组成。没有任何文件处于一个以上区域中。最后，每个文件由 *record* 类型的结点组成。和前面一样，文件恰好由作为其子结点的记录组成，并且任何记录不能同时属于多个文件。

树中每个结点都可以单独加锁。正如我们在两阶段封锁协议中所做的那样，我们将使用**共享**(shared)锁与**排他**(exclusive)锁。当事务对一个结点加锁，或为共享锁或为排他锁，该事务也以同样类型的锁隐式地封锁这个结点的全部后代结点。例如，若事务 T_i 给图 15-15 中的 F_b **显式**(explicit lock)地
679 加排他锁，则事务 T_i 也给所有属于该文件的记录**隐式**(implicit lock)地加排他锁。没有必要显式地给 F_b 中的单条记录逐个加锁。

假设事务 T_j 希望封锁文件 F_b 的记录 r_{b_6}。由于 T_i 显式地给 F_b 加锁，因此意味着 r_{b_6} 也被加锁(隐式地)。但是，当 T_j 发出对 r_{b_6} 加锁的请求时，r_{b_6} 没有显式加锁！系统如何判定是否 T_j 可以封锁 r_{b_6} 呢？T_j 必须从树根到 r_{b_6} 进行遍历，如果发现此路径上某个结点的锁与要加的锁类型不相容，则 T_j 必须延迟。

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

图 15-16 相容性矩阵

现假设事务 T_k 希望封锁整个数据库。为此，它只需给层次结构图的根结点加锁。不过，请注意 T_k 给根结点加锁不会成功，因为目前 T_i 在树的某部分持有锁(具体地说，对文件 F_b 持有锁)。但是，系统是怎样判定根结点是否可以加锁呢？一种可能的方法是搜索整棵树。然而，这个方法破坏了多粒度封锁机制的初衷。获取这种知识的一个更有效的方法是引入一种新的锁类型，即**意向锁类型**(intention lock mode)。如果一个结点加上了意向锁，则意味着要在树的较低层进行显式加锁(也就是说，以更小的粒度加锁)。在一个结点显式加锁之前，该结点的全部祖先结点均加上了意向锁。因此，事务不必搜索整棵树就能判定能否成功地给一个结点加锁。希望给某个结点(如 Q)加锁的事务必须遍历从根到 Q 的路径。在遍历树的过程中，该事务给各结点加上意向锁。

与共享锁相关联的是一种意向锁，与排他锁相关联的是另一种意向锁。如果一个结点加上了**共享**

型意向(intention-shared (IS) mode)锁，那么将在树的较低层进行显式封锁，但只能加共享锁。类似地，如果一个结点加**排他型意向**(intention-exclusive(IX) mode)锁，那么将在树的较低层进行显式封锁，可以加排他锁或共享锁。最后，若一个结点加上了**共享排他型意向锁**(shared and intention-exclusive (SIX) mode)，则以该结点为根的子树显式地加了共享锁，并且将在树的更低层显式地加排他锁。这些锁类型的相容函数如图 15-16 所示。 680

多粒度封锁协议(multiple-granularity locking protocol)采用这些锁类型保证可串行性。每个事务 T_i 要求按如下规则对数据项 Q 加锁：

1. 事务 T_i 必须遵从图 15-16 所示的锁类型相容函数。
2. 事务 T_i 必须首先封锁树的根结点，并且可以加任意类型的锁。
3. 仅当 T_i 当前对 Q 的父结点具有 IX 或 IS 锁时，T_i 对结点 Q 可加 S 或 IS 锁。
4. 仅当 T_i 当前对 Q 的父结点具有 IX 或 SIX 锁时，T_i 对结点 Q 可加 X、SIX 或 IX 锁。
5. 仅当 T_i 未曾对任何结点解锁时，T_i 可对结点加锁（也就是说，T_i 是两阶段的）。
6. 仅当 T_i 当前不持有 Q 的子结点的锁时，T_i 可对结点 Q 解锁。

多粒度协议要求加锁按自顶向下的顺序(根到叶)，而锁的释放则按自底向上的顺序(叶到根)。

作为该协议的例子，考虑图 15-15 所示的树及下面的事务：

- 假设事务 T_{21} 读文件 F_a 的记录 r_{a_2}。那么，T_{21} 需给数据库、区域 A_1，以及 F_a 加 IS 锁(按此顺序)，最后给 r_{a_2} 加 S 锁。
- 假设事务 T_{22} 要修改文件 F_a 的记录 r_{a_1}。那么，T_{22} 需给数据库、区域 A_1，以及文件 F_a(按此顺序)加 IX 锁，最后给记录 r_{a_1} 加 X 锁。
- 假设事务 T_{23} 要读取文件 F_a 的所有记录。那么，T_{23} 需给数据库和区域 A_1(按此顺序)加 IS 锁，最后给 F_a 加 S 锁。 681
- 假设事务 T_{24} 要读取整个数据库。它在给数据库加 S 锁后就可读取。

我们注意到事务 T_{21}、T_{23} 与 T_{24} 可以并发地存取数据库。事务 T_{22} 可以与 T_{21} 并发执行，但不能与 T_{23} 或 T_{24} 并发执行。

该协议增强了并发性，减少了锁开销。它在由如下事务类型混合而成的应用中尤其有用：

- 只访问几个数据项的短事务。
- 由整个文件或一组文件来生成报表的长事务。

类似的封锁协议可以用于数据粒度按有向无环图组织的数据库系统中，参考文献见文献注解。与在两阶段封锁协议中一样，在多粒度协议中也可能存在死锁。有多种技术可以用来减少多粒度协议中死锁发生的频度，甚至可以彻底消除死锁。文献注解给出了这些技术的相关资料。

15.4 基于时间戳的协议

到目前为止我们所讲述的封锁协议中，每一对冲突事务的次序是在执行时由二者都申请的，但类型不相容的第一个锁决定的。另一种决定事务可串行化次序的方法是事先选定事务的次序。其中最常用的方法是**时间戳排序机制**。

15.4.1 时间戳

对于系统中每个事务 T_i，我们把一个唯一的固定时间戳和它联系起来，此时间戳记为 $TS(T_i)$。该时间戳是在事务 T_i 开始执行前由数据库系统赋予的。若事务 T_i 已赋予时间戳 $TS(T_i)$，此时有一新事务 T_j 进入系统，则 $TS(T_i) < TS(T_j)$。实现这种机制可以采用下面这两个简单的方法：

1. 使用**系统时钟**(system clock)的值作为时间戳；即，事务的时间戳等于该事务进入系统时的时钟值。

2. 使用**逻辑计数器**(logical counter)，每赋予一个时间戳，计数器增加计数；即，事务的时间戳等于该事务进入系统时的计数器值。 682

事务的时间戳决定了串行化顺序。因此，若 $TS(T_i) < TS(T_j)$，则系统必须保证所产生的调度等价于事务 T_i 出现在事务 T_j 之前的某个串行调度。

要实现这个机制，每个数据项 Q 需要与两个时间戳值相关联：

- **W-timestamp**(Q)表示成功执行 **write**(Q)的所有事务的最大时间戳。
- **R-timestamp**(Q)表示成功执行 **read**(Q)的所有事务的最大时间戳。

每当有新的 **read**(Q) 或 **write**(Q)指令执行时，这些时间戳就更新。

15.4.2 时间戳排序协议

时间戳排序协议(timestamp-ordering protocol)保证任何有冲突的 **read** 或 **write** 操作按时间戳顺序执行。该协议运作方式如下：

1. 假设事务 T_i 发出 **read**(Q)。
   a. 若 TS(T_i) < W-timestamp(Q)，则 T_i 需要读入的 Q 值已被覆盖。因此，read 操作被拒绝，T_i 回滚。
   b. 若 TS(T_i) ≥ W-timestamp(Q)，则执行 read 操作，R – timestamp(Q)被设置为 R-timestamp(Q)与 TS(T_i)两者的最大值。
2. 假设事务 T_i 发出 **write**(Q)。
   a. 若 TS(T_i) < R-timestamp(Q)，则 T_i 产生的 Q 值是先前所需要的值，且系统已假定该值不会再产生。因此，**write** 操作被拒绝，T_i 回滚。
   b. 若 TS(T_i) < W-timestamp(Q)，则 T_i 试图写入的 Q 值已过时。因此，**write** 操作被拒绝，T_i 回滚。
   c. 其他情况，系统执行 **write** 操作，将 W-timestamp(Q)设置为 TS(T_i)。

如果事务 T_i 由于发出 **read** 或 **write** 操作而被并发控制机制回滚，则系统赋予它新的时间戳并重新启动。

683 为说明这个协议，我们考虑事务 T_{25} 与 T_{26}。事务 T_{25} 显示账户 A 与 B 的内容：

T_{25}: **read**(B);
read(A);
display($A + B$).

事务 T_{26} 从账户 B 转 50 美元到账户 A，然后显示两个账户的内容：

T_{26}: **read**(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

在说明时间戳协议下的调度时，我们将假设事务在执行第一条指令之前的那一刻被赋予时间戳。因此，在如图 15-17 所示的调度 3 中，TS(T_{25}) < TS(T_{26})，这是满足时间戳协议的一个可能的调度。

我们注意到前面的执行过程也可以由两阶段封锁协议产生。不过，存在满足两阶段封锁协议却不满足时间戳协议的调度，反之亦然(见习题 15.29)。

时间戳排序协议保证冲突可串行化，这因为冲突操作按时间戳顺序进行处理。

该协议保证无死锁，因为不存在等待的事务。但是，当一系列冲突的短事务引起长事务反复重启时，可能导致长事务饿死的现象。如果发现一个事务反复重启，与之冲突的事务应当暂时阻塞，以使该事务能够完成。

该协议可能产生不可恢复的调度。然而，该协议可以进行扩展，用以下几种方法之一来保证调度可恢复：

- 在事务末尾执行所有的写操作能保证可恢复性和无级联性，这

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

图 15-17 调度 3

些写操作必须具有下述意义的原子性：在写操作正在执行的过程中，任何事务都不许访问已写完的任何数据项。

- 可恢复性和无级联性也可以通过使用一个受限的封锁形式来保证，由此，对未提交数据项的读操作被推迟到更新该数据项的事务提交之后(见习题15.30)。
- 可恢复性可以通过跟踪未提交写操作来单独保证，一个事务 T_i 读取了其他事务所写的数据，只有在其他事务都提交之后，T_i 才能提交。15.1.5节概述过的提交依赖可以用作这个目的。

15.4.3 Thomas 写规则

我们现在给出对时间戳排序协议的一种修改，它允许的并发程度比15.4.2节中的协议要高。让我们考虑图15-18所示的调度4，并应用时间戳排序协议。由于 T_{27} 先于 T_{28} 开始，因此假定 TS(T_{27}) < TS(T_{28})。T_{27} 的 **read**(Q)操作成功，T_{28} 的 **write**(Q)操作也成功。当 T_{27} 试图进行 write(Q)操作时，我们发现 TS(T_{27}) < W-timestamp(Q)，因为 W-timestamp(Q) = TS(T_{28})。所以，T_{27} 的 write(Q) 操作被拒绝且事务 T_{27} 回滚。

虽然事务 T_{27} 回滚是时间戳排序协议所要求的，但这是不必要的。由于 T_{28} 已经写入了 Q，因此 T_{27} 想要写入的值将永远不会被读到。满足 TS(T_i) < TS(T_{28})的任何事务 T_i 试图进行 **read**(Q)操作时均回滚，因为 TS(T_i) < W-timestamp(Q)。满足 TS(T_j) > TS(T_{28})的任何事务 T_j 必须读由 T_{28} 写入的 Q 值，而不是 T_{27} 想要写入的值。

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

图15-18 调度4

根据以上分析，我们可以修改时间戳排序协议而得到一个新版本的协议，该协议在某些特定的情况下忽略过时的 **write** 操作。协议中有关 **read** 操作的规则保持不变；但 **write** 操作的协议规则与15.4.2节中的时间戳排序协议略有区别。 684 / 685

时间戳排序协议所做的这种修改称为 **Thomas 写规则**(Thomas' write rule)：假设事务 T_i 发出 **write**(Q)。

1. 若 TS(T_i) < R-timestamp(Q)，则 T_i 产生的 Q 值是先前所需要的值，且系统已假定该值不会再产生。因此，**write** 操作被拒绝，T_i 回滚。
2. 若 TS(T_i) < W-timestamp(Q)，则 T_i 试图写入的 Q 值已过时。因此，这个 **write** 操作可忽略。
3. 其他情况，执行 **write** 操作，将 W-timestamp(Q)设置为 TS(T_i)。

以上规则与15.4.2节中规则的区别在第二条，时间戳排序协议在 T_i 发出 **write**(Q)且 TS(T_i) < W-timestamp(Q)时要求 T_i 回滚。而在修改后协议对这种情况的处理是，在 TS(T_i) ≥ R-timestamp(Q)时，我们忽略过时的 **write** 操作。

通过忽略写，Thomas 写规则允许非冲突可串行化但是正确的调度。这些允许的非冲突可串行化调度满足视图可串行化调度的概念(见示例框)。Thomas 写规则实际上是通过删除事务发出的过时的 **write** 操作来使用视图可串行性。对事务的这种修改使得系统可以产生本章中其他协议所不能产生的可串行化调度。例如，图15-18所示调度4是非冲突可串行化的，因此在两阶段封锁协议、树形封锁协议或时间戳排序协议中都是不可能的。在 Thomas 写规则下，T_{27} 的 **write**(Q)操作将忽略，产生视图等价于串行调度 < T_{27}，T_{28} > 的一个调度。

视图可串行化

有一种等价形式比冲突等价的限制要宽松，但这种等价形式与冲突等价一样只基于事务的 **read** 与 **write** 操作。

考虑两个调度 S 与 S'，参与两个调度的事务集是相同的，若满足下面三个条件，调度 S 与 S' 就称为视图等价(view equivalent)的：

1. 对于每个数据项 Q，若事务 T_i 在调度 S 中读取了 Q 的初始值，那么在调度 S' 中 T_i 也必须读取 Q 的初始值。

2. 对于每个数据项 Q，若在调度 S 中事务 T_i 执行了 **read**(Q)并且读取的值是由事务 T_j 执行 **write**(Q)操作产生的；则在调度 S' 中，T_i 的 **read**(Q)操作读取的值 Q 也必须是由 T_j 的同一个 **write**(Q)操作产生的。

3. 对于每个数据项 Q，若在调度 S 中有事务执行了最后的 write(Q)操作，则在调度 S′中该事务也必须执行最后的 write(Q)操作。

条件 1 与 2 保证在两个调度中的每个事务都读取相同的值，从而进行相同的计算。条件 3 与条件 1、2 一起保证两个调度得到相同的最终系统状态。

视图等价的概念引出了视图可串行化的概念。如果某个调度视图等价于一个串行调度，则我们说这个调度是**视图可串行化**(view serializable)的。

举个例子，假设我们在调度 4 中增加事务 T_{29}，由此得到以下视图可串行化的调度(调度 5)：

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

实际上，调度 5 视图等价于串行调度 $<T_{27}, T_{28}, T_{29}>$，因为在两个调度中 read(Q)指令均是读取 Q 的初始值，两个调度中 T29 均最后写入 Q 值。

每个冲突可串行化调度都是视图可串行化的，但存在非冲突可串行化的视图可串行化调度。事实上，调度 5 就不是冲突可串行化的，因为每对连续指令均冲突，从而交换指令是不可能的。

注意，在调度 5 中，事务 T_{28} 与 T_{29} 执行 **write**(Q)操作之前没有执行 **read**(Q)操作。这样的写操作称作**盲目写操作**(blind write)。盲目写操作存在于任何非冲突可串行化的视图可串行化调度中。

15.5 基于有效性检查的协议

在大部分事务是只读事务的情况下，事务发生冲突的频率较低。因此，许多这样的事务，即使在没有并发控制机制监控的情况下执行，也不会破坏系统的一致状态。并发控制机制带来代码执行的开销和可能的事务延迟，采用开销较小的机制是我们所希望的。减小开销面临的困难是我们事先并不知道哪些事务将陷入冲突中。为获得这种知识，我们需要一种监控系统的机制。

686
|
687

有效性检查协议(validation protocol)要求每个事务 T_i 在其生命周期中按两个或三个阶段执行，这取决于该事务是一个只读事务还是一个更新事务。这些阶段顺序地列在下面。

1. **读阶段**(read phase)：在这一阶段中，系统执行事务 T_i。各数据项值被读入并保存在事务 T_i 的局部变量中。所有 **write** 操作都是对局部临时变量进行的，并不对数据库进行真正的更新。

2. **有效性检查阶段**(validation phase)：对事务 T_i 进行有效性测试(下面将会介绍)。判定是否可以执行 **write** 操作而不违反可串行性。如果事务有效性测试失败，则系统终止这个事务。

3. **写阶段**(write phase)：若事务 T_i 已通过有效性检查(第 2 步)，则保存 T_i 任何写操作结果的临时局部变量值被复制到数据库中。只读事务忽略这个阶段。

每个事务必须按以上顺序经历这些阶段。然而，并发执行的事务的三个阶段可以是交叉执行的。

为进行有效性检测，我们需要知道事务 T_i 的各个阶段何时进行。为此，我们将三个不同的时间戳与事务 T_i 相关联。

1. **Start**(T_i)：事务 T_i 开始执行的时间。

2. **Validation**(T_i)：事务 T_i 完成读阶段并开始其有效性检查的时间。

3. **Finish**(T_i)：事务 T_i 完成写阶段的时间。

我们利用时间戳 Validation(T_i)的值，通过时间戳排序技术决定可串行性顺序。因此，值 TS(T_i) = Validation(T_i)，并且若 TS(T_j) < TS(T_k)，则产生的任何调度必须等价于事务 T_j 出现在 T_k 之前的某个串行调度。我们选择 Validation(T_i)而不是 Start(T_i)作为事务 T_i 的时间戳是因为在冲突频度很低的情况

下期望有更快的响应时间。

事务 T_i 的**有效性测试**(validation test)要求任何满足 TS(T_k) < TS(T_i)的事务 T_k 必须满足下面两条件之一：

1. Finish(T_k) < Start(T_i)。因为 T_k 在 T_i 开始之前完成其执行，所以可串行性次序得到了保证。

2. T_k 所写的数据项集与 T_i 所读数据项集不相交，并且 T_k 的写阶段在 T_i 开始其有效性检查阶段之前完成 (Start(T_i) < Finish(T_k) < Validation(T_i))。这个条件保证 T_k 与 T_i 的写不重叠。因为 T_k 的写不影响 T_i 的读，又因为 T_i 不可能影响 T_k 的读，从而保证了可串行性次序。

688

作为例子，我们再一次考虑事务 T_{25} 与事务 T_{26}。假设 TS(T_{25}) < TS(T_{26})，则有效性检查阶段成功地产生调度 6，如图 15-19 所示。注意，只有在 T_{26} 的有效性检查阶段之后才写实际的变量。因此，T_{25} 读取 B 与 A 的旧值，且该调度是可串行化的。

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
< validate>	
display($A + B$)	
	< validate>
	write(B)
	write(A)

图 15-19 调度 6，采用有效性检查得到的一个调度

有效性检查机制自动预防级联回滚，因为只有发出写操作的事务提交后实际的写才发生。然而，存在长事务饿死的可能，原因是一系列冲突的短事务引起长事务反复重启。为了避免饿死，与之冲突的事务应当暂时阻塞，以使该长事务能够完成。

在有效性检查机制中，由于事务乐观地执行，假定它们能够完成执行并且最终有效，因此也称为**乐观的并发控制**(optimistic concurrency control)机制。与之相反，封锁和时间戳排序是悲观的，因为当它们检测到一个冲突时，它们强迫事务等待或回滚，即使该调度有可能是冲突可串行化的。

15.6 多版本机制

目前为止讲述的并发控制机制要么延迟一项操作要么中止发出该操作的事务来保证可串行性。例如，**read** 操作可能由于相应值还未写入而延迟；或因为它要读取的值已被覆盖而被拒绝执行(即，发出 **read** 操作的事务必须中止)。如果每一数据项的旧值拷贝保存在系统中，这些问题就可以避免。

在**多版本并发控制**(multiversion concurrency control)机制中，每个 **write**(Q)操作创建 Q 的一个新**版本**(version)。当事务发出一个 **read**(Q)操作时，并发控制管理器选择 Q 的一个版本进行读取。并发控制机制必须保证用于读取的版本的选择能保持可串行性。由于性能原因，一个事务能容易而快速地判定读取哪个版本的数据项也是很关键的。

689

15.6.1 多版本时间戳排序

时间戳排序协议可以扩展为多版本的协议。对于系统中的每个事务 T_i，我们将一个唯一的静态时间戳与之关联，记为 TS(T_i)。如 15.4 节所述，数据库系统在事务开始前赋予该时间戳。

对于每个数据项 Q，有一个版本序列 < Q_1，Q_2，$\cdots Q_m$ > 与之关联，每个版本 Q_k 包含三个数据字段：

- **Content** 是 Q_k 版本的值。
- **W-timestamp**(Q)是创建 Q_k 版本的事务的时间戳。
- **R-timestamp**(Q)：所有成功地读取 Q_k 版本的事务的最大时间戳。

事务(如 T_i)通过发出 **write**(Q)操作创建数据项 Q 的一个新版本 Q_k，版本的 content 字段保存事务 T_i 写入的值，系统将 W-timestamp 与 R-timestamp 初始化为 TS(T_i)。每当事务 T_j 读取 Q_k 的值且 R-timestamp(Q_k) < TS(T_j)时，R-timestamp 的值就更新。

下面展示的**多版本时间戳排序机制**(multiversion timestamp-ordering scheme)保证可串行性。该机制运作如下：假设事务 T_i 发出 **read**(Q)或 **write**(Q)操作，令 Q_k 表示 Q 满足如下条件的版本，其写时间戳是小于或等于 TS(T_i)的最大写时间戳。

1. 如果事务 T_i 发出 **read**(Q)，则返回值是 Q_k 的内容。

2. 如果事务 T_i 发出 **write**(Q)，且若 TS(T_i) < R-timestamp(Q_k)，则系统回滚事务 T_i，另一方面，

若 TS(T_i) = W-timestamp(Q_k)，则系统覆盖 Q_k 的内容；否则(若 TS(T_i) > R-timestamp(Q_k))，创建 Q 的一个新版本。

规则 1 的理由是显然的，一个事务读取位于其前的最近版本。第二条规则规定当一个事务执行写操作"太迟"时将强迫中止，更确切地说，如果 T_i 试图写入其他事务应该已读取了的版本，则我们不允许该写操作成功。

690 不再需要的版本根据以下规则删除：假设有某数据项的两个版本 Q_k 与 Q_j，这两个版本的 W-timestamp 都小于系统中最老的事务的时间戳，那么 Q_k 和 Q_j 中较旧的版本将不会再用到，因而可以删除。

多版本时间戳排序机制有一个很好的特性：读请求从不失败且不必等待。在典型的数据库系统中，读操作比写操作频繁，因而这个优点对于实践来说至关重要。

然而这个机制也存在两个不好的特性。首先，读取数据项要求更新 R-timestamp 字段，于是产生两次潜在的磁盘访问而不是一次。其次，事务间的冲突通过回滚而不是等待来解决。这种做法开销可能很大。15.6.2 节讲述一个可以减轻这个问题的算法。

多版本时间戳排序机制不保证可恢复性和无级联性。按照与基本的时间戳排序机制一样的方法进行扩充，我们可以使之成为可恢复的和无级联的。

15.6.2 多版本两阶段封锁

多版本两阶段封锁协议(multiversion two-phase locking protocol)希望将多版本并发控制的优点与两阶段封锁的优点结合起来。该协议对**只读事务**(read-only transaction)与**更新事务**(update transaction)加以区分。

更新事务执行强两阶段封锁协议；即它们持有全部锁直到事务结束。因此，它们可以按提交的次序进行串行化。数据项的每个版本有一个时间戳，这种时间戳不是真正基于时钟的时间戳，而是一个计数器，我们称之为 **ts-counter**，这个计数器在提交处理时增加计数。

只读事务在开始执行前，数据库系统读取 **ts-coumter** 的当前值来作为该事务的时间戳。只读事务在执行读操作时遵从多版本时间戳排序协议。因此，当只读事务 T_i 发出 **read**(Q)时，返回值是小于 TS(T_i)的最大时间戳的版本的内容。

当更新事务读取一个数据项时，它在获得该数据项上的共享锁后读取该数据项最新版本的值。当更新事务想写一个数据项时，它首先要获得该数据项上的排他锁，然后为此数据项创建一个新版本。写操作在新版本上进行，新版本的时间戳最初置为∞，它大于任何可能的时间戳值。

当更新事务 T_i 完成其任务后，它按如下方式进行提交：首先，T_i 将它创建的每一版本的时间戳设置为 **ts-counter** 的值加 1；然后，T_i 将 **ts-counter** 增加 1。在同一时间内只允许有一个更新事务进行提交。

这样，在 T_i 增加了 **ts-counter** 之后启动的只读事务将看到 T_i 更新的值，而那些在 T_i 增加 ts-counter
691 之前就启动的只读事务将看到 T_i 更新之前的值。无论哪种情况，只读事务均不必等待加锁。多版本两阶段封锁也保证调度是可恢复的和无级联的。

版本删除类似于多版本时间戳排序中采用的方式。假设有某数据项的两个版本 Q_k 与 Q_j，两个版本的时间戳都小于或等于系统中最老的只读事务的时间戳。则两个版本中较旧的将不会再使用，可以删除。

15.7 快照隔离

快照隔离是一种特殊的并发控制机制，并且在商业和开源系统中广泛接受，包括 Oracle、PostgreSQL 和 SQL Server。14.9.3 节介绍了快照隔离。这儿将更详细地观察它如何工作。

从概念上讲，快照隔离在事务开始执行时给它数据库的一份"快照"。然后，事务在该快照上操作，和其他并发事务完全隔离开。快照中的数据值仅包括已经提交的事务所写的值。这种隔离对于只读事务来说是理想的，因为它们不用等待，也不会被并发管理器中止。当然，更新数据库的事务在将更新写入数据库之前，必须处理与其他并发更新的事务之间存在的潜在冲突。更新操作发生在事务的

私有工作空间中，直到事务成功提交，此时更新写入数据库。当允许事务 T 提交时，事务 T 变为提交状态，并且 T 对数据库的所有写操作都必须作为一个原子操作执行，以保证其他事务的快照要么包括 T 的所有更新，要么任何都不包括。

15.7.1 更新事务的有效性检验步骤

决定是否允许一个更新事务的提交需要小心。潜在地，两个并发执行的事务可能会更新同一个数据项。由于两个事务的操作用它们各自的私有快照隔离，因此任何事务都看不到对方所做的更新。如果两个事务都允许写入数据库，第一个更新的写入将被第二个覆盖。结果导致一个**更新丢失**(lost update)。显然地，这必须预防。快照隔离有两个变种，都防止了更新丢失。它们称为*先提交者获胜*和*先更新者获胜*。两种方法都基于检查事务的并发事务。一个事务称为**与 T 并发**(concurrent with T)，如果在从 T 开始到执行这个检查之间的任何时刻该事务是活跃的或者部分提交的。

按照**先提交者获胜**(first committer wins)方法，当事务 T 进入部分提交状态，以下操作作为一个原子操作执行： 692

- 检查是否有与 T 并发执行的事务，对于 T 打算写入的某些数据，该事务已经将更新写入数据库。
- 如果发现这样的事务，则 T 中止。
- 如果没有发现这样的事务，则 T 提交，并且将更新写入数据库。

这种方法称为“先提交者获胜”，因为如果事务发生冲突，第一个使用上述规则检查的事务成功地写入更新，而随后的事务则被迫中止。有关如何实现以上检查的细节在习题 15.19 中讨论。

按照**先更新者获胜**(first updater wins)方法，系统采用一种仅用于更新操作的锁机制(读操作不受此影响，因为它们不获得锁)。当事务 T_i 试图更新一个数据项时，它请求该数据项的一个写锁。如果没有另一个并发事务持有该锁，则获得锁后执行以下步骤：

- 如果这个数据项已经被任何并发事务更新，则 T_i 中止。
- 否则 T_i 能够执行其操作，可能包括提交。

然而，如果另一个并发事务 T_j 已经持有该数据项的写锁，则 T_i 不能执行，并且执行以下规则：

- T_i 等待直到 T_j 中止或提交。
  - 如果 T_j 中止，则锁被释放并且 T_i 可以获得锁。当获得锁后，执行前面描述的对于并发事务的更新检查：如果有另一个并发事务更新过该数据项，则 T_i 中止；否则，执行其操作。
  - 如果 T_j 提交，则 T_i 必须中止。

当事务提交或中止时，锁被释放。

这种方法称为“先更新者获胜”，因为如果事务发生冲突，允许第一个获得锁的事务提交并完成其更新。其后试图更新的事务中止，除非第一个更新者随后由于其他原因中止。(除了等待看第一个事务 T_j 是否中止，其后的更新者 T_i 可以在发现它所想要的写锁被 T_j 持有时立刻中止。)

15.7.2 串行化问题

快照隔离在实践中很有吸引力，因为其开销低并且没有中止发生，除非两个并发的事务更新同一个数据项。

然而，我们前面所给出的，以及在实践中实现的快照隔离机制存在一个严重的问题：*快照隔离不能保证可串行化*。即使在将快照隔离作为串行化隔离级别实现的 Oracle 中也是如此！下面举例说明快照隔离下可能的非可串行化执行，并且说明如何解决。 693

1. 假设有两个并发事务 T_i 和 T_j，以及两个数据项 A 和 B。假设 T_i 读取 A 和 B，然后更新 B，而 T_j 读取 A 和 B，然后更新 A。简单起见，假设没有其他并发事务。由于 T_i 和 T_j 是并发的，因此任何事务在其快照内不能看到对方的更新。但是，因为它们更新的是不同数据项，不管系统采用先提交者获胜或者先更新者获胜，两者都可以提交。

然而，优先图有一个环。优先图中从 T_i 到 T_j 有一条边，因为 T_i 在 T_j 写 A 之前读取 A 的值。同样，优先图中从 T_j 到 T_i 也有一条边，因为 T_j 在 T_i 写 B 之前读取 B 的值。由于优先图中存在环，因此结果

是非可串行化的。

一对事务中的每一个都读取对方写的数据，但是不存在两者同时写的数据，这种情况称为**写偏斜**(write skew)。举一个写偏斜的具体例子，考虑一个银行场景。假设银行通过完整性约束限制一个客户的支票账户和储蓄账户的余额之和不能为负。假设一个客户的支票账户余额和储蓄账户余额分别为100美元和200美元。假设事务 T_{36} 读这两个余额，进行完整性约束核实，从支票账户中取出200美元。假设另一个并发执行的事务 T_{37} 也经过完整性约束核实，从储蓄账户中取出200美元。由于每个事务都在其快照上进行完整性约束检验，如果它们同时执行，任何一方都会相信取款后的余额之和为100美元，因此取款并不违反完整性约束。由于两个事务更新不同的数据项，它们之间不存在更新冲突，因此在快照隔离下两者都可以提交。

遗憾的是，在 T_{36} 和 T_{37} 都提交后的状态中，余额之后为 -100 美元，违反了完整性约束。这种违反不可能出现在 T_{36} 和 T_{37} 的串行执行中。

值得注意的是，由数据库强制执行的完整性约束，例如主键和外键约束，不能在快照上进行检查；否则将可能有两个并发事务同时插入主键相同的记录，或者一个事务插入一个外键值，而另一个并发事务从引用的表中删除该值。相反，数据库系统必须在数据库的当前状态下检查约束，作为提交时有效性检验的一部分。

694 2. 在接下来的例子中，我们考虑两个并发更新事务本身是可串行化的且不存在任何问题，但是当一个只读事务出现在某个时刻时正好会造成问题。

假设有两个并发事务 T_i 和 T_j，以及两个数据项 A 和 B。假设 T_i 读取 B，然后更新 B，而 T_j 读取 A 和 B，然后更新 A。同时并发执行这两个事务没有问题。由于 T_i 只访问数据项 B，在数据项 A 上没有冲突，因此优先图中没有环。优先图中唯一的边是从 T_j 指向 T_i，因为 T_j 在 T_i 写入 B 前读取 B 的值。

然而，假设 T_i 提交时 T_j 仍然活跃。假设 T_i 提交后但 T_j 尚未提交时，一个新的只读事务 T_k 进入系统，而且 T_k 读取 A 和 B。它的快照包括 T_i 的更新，由于 T_i 已经提交。然而，由于 T_j 未提交，它的更新还未写入数据库，因此不包含在 T_k 的快照中。

考虑优先图中添加的与 T_k 相关的边。优先图中有一条从 T_i 到 T_k 的边，因为 T_i 在 T_k 读取 B 之前写入 B 的值。优先图中还有一条从 T_k 到 T_j 的边，因为 T_k 在 T_j 写 A 之前读取 A 的值。这导致优先图中出现环，说明调度是非可串行化的。

上述异常可能不像它们乍一看起来那样麻烦。回想串行化的原因是为了确保在事务并发执行时的数据库保持一致性。因为一致性是目标，所以我们可以接受潜在的非可串行化执行，如果我们确信那些可能出现的非可串行化执行不会导致不一致。上述第二个例子只有在提交只读事务(T_k)的应用关心 A 和 B 的更新出现乱序时才会出现问题。在这个例子中，我们没有指定每个事务必须保持的数据库一致性约束。如果我们是在处理一个金融数据库，T_k 读取非可串行的更新时会出现严重问题。而如果 A 和 B 是同一个课程的两次开课的注册数，则 T_k 不要求理想的可串行化，并且我们可以从应用中得知更新频率足够低，因此 T_k 读取中的不准确也是不重要的。

数据库必须在提交时而非快照上检查完整性约束这一事实也帮助避免在某些情况下的不一致。一些金融应用程序创建连续的序列号，例如在给账单编号时，将新账单号设置为当前最大账单号加1。如果两个事务并发执行，每一方在其快照中都会看到相同的账单集合，每一方都会创建具有相同账单
695 号的新账单。两个事务都通过快照隔离的有效性检验，因为它们不更新相同的元组。然而，执行是非串行化的。由此产生的数据库状态无法从两个事务的任何串行执行中得到。创建两个具有相同账单号的账单可能会导致严重的法律后果。

上述问题是一个幻象的例子，15.8.3节将描述它，因为任何一个事务执行的插入操作都会和另一个事务读取最大账单号的读操作冲突，但是这种冲突不能通过快照隔离检测到。[⊖]

幸运的是，在大多数应用程序中账单号会声明为主键，数据库系统会在快照之外检查主键冲突，

⊖ SQL标准中的术语幻象问题指非可重复谓词读，导致一些人声称快照隔离避免了幻象问题。然而，根据我们对幻象冲突的定义，该声称是无效的。

并且回滚事务中的一方。[⊖]

程序开发人员可以通过将以下 **for update** 子句附加到 SQL 查询语句之后来防止这种快照异常：

```
select *
from instructor
where ID = 22222
for update;
```

添加 **for update** 子句会使系统出于并发控制的目的将读取的数据作为刚被更新来对待。在第一个关于写偏斜的例子中，如果 **for update** 子句附加到查询账户余额的查询语句之后，那么两个并发事务中只有一个允许提交，因为看起来两个事务都对支票账户和储蓄账户余额进行更新。

在第二个非可串行化执行的例子中，如果事务 T_k 的发起者希望避免异常，**for update** 子句可以附加到查询语句之后，尽管事实上没有更新。在该例子中，如果 T_k 采用 **select for update**，则当它读取 A 和 B 时作为对它们的更新。结果将是或者 T_k 或者 T_j 中止，并稍后作为一个新事务重试。这将会得到一个可串行化的执行。在这个例子中，另外两个事务中的查询不需要添加 **for update** 子句，不必要的 **for update** 子句会导致并发度的显著下降。

存在规范的方法(见文献注解)用来判断给定一个事务的混合运行，是否存在快照隔离下非可串行化执行的风险，并且决定引入何种冲突(例如采用 **for update** 子句)来保证可串行化。当然，这些方法只有在我们事先知道要执行什么事务时才有效。在一些应用中，所有事务都是预先确定的事务集中的事务，这使得这种分析成为可能。然而，如果应用允许不受限制的、临时的事务，则这种分析是不可能的。696

在三个广泛应用的支持快照隔离的系统中，SQL Server 提供了一个可串行化隔离级别选项，它能真正保证可串行化，以及一个快照隔离级别选项，它提供快照隔离的性能优势(连同上面讨论的潜在异常)。在 Oracle 和 PostgreSQL 中，可串行化隔离级别仅提供快照隔离。

15.8 插入操作、删除操作与谓词读

目前为止我们一直把注意力集中在 **read** 与 **write** 操作上。这就限制了事务只能处理已存在于数据库中的数据项。一些事务不仅要求访问已存在的数据项，而且要求创建新的数据项；还有一些事务要求能删除数据项。为考察这样的事务如何影响并发控制，我们引入如下操作。

- **delete**(Q)：从数据库中删除数据项 Q。
- **insert**(Q)：插入一个新的数据项 Q 到数据库中并赋予 Q 一个初值。

事务 T_i 试图在 Q 被删除后执行 **read**(Q)操作将导致 T_i 中的逻辑错误。类似地，事务 T_i 在 Q 被插入之前执行 **read**(Q)操作也会导致 T_i 中的逻辑错误。试图删除并不存在的数据项也是一个逻辑错误。

15.8.1 删除

要理解 **delete** 指令怎样影响并发控制，我们必须弄清 **delete** 指令何时与另一指令发生冲突。令 I_i 与 I_j 分别是 T_i 与 T_j 的指令，它们连续地出现于调度 S 中。令 I_i = **delete**(Q)。我们考虑几条指令 I_j。

- I_j = **read**(Q)：I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_j 可以成功执行 **read** 操作。
- I_j = **write**(Q)：I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_j 可以成功执行 **write** 操作。
- I_j = **delete**(Q)：I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_i 将出现逻辑错误。
- I_j = **insert**(Q)：I_i 与 I_j 冲突。假设数据项 Q 在执行 I_i 与 I_j 之前不存在。那么，若 I_i 出现在 I_j 之前，事务 T_i 将出现逻辑错误。如果 I_j 出现在 I_i 之前，则没有逻辑错误。类似地，如果 Q 在执行 I_i 与 I_j 之前已存在，则如果 I_j 出现在 I_i 之前会出现逻辑错误，反过来则不会。

⊖ 账单号重复问题在 I. I. T. Bombay 金融应用中实际发生过几次，其账单号不是作为主键(原因过于复杂，不在此讨论)，问题是由财务审计员发现的。

697 我们可以总结如下：

- 在两阶段封锁协议下，一数据项可以删除之前，在该数据项上必须请求加排他锁。
- 在时间戳排序协议下，必须执行类似于为 **write** 操作进行的测试。假设事务 T_i 发出 **delete**(Q)：
  - □ 如果 TS(T_i) < R-timestamp(Q)，则 T_i 将要删除的 Q 值已被满足 TS(T_j) > TS(T_i)的事务 T_j 读取。因此，**delete** 操作被拒绝，T_i 回滚。
  - □ 如果 TS(T_i) < W-timestamp(Q)，则满足 TS(T_j) > TS(T_i)的事务 T_j 已经写过 Q。因此，**delete** 操作被拒绝，T_i 回滚。
  - □ 否则，执行 **delete** 操作。

15.8.2 插入

我们已经看到 **insert**(Q)操作与 **delete**(Q)操作冲突。类似地，**insert**(Q)操作与 read(Q)操作或 **write**(Q)操作冲突。在数据项存在之前不能进行 **read** 或 **write** 操作。

由于 **insert**(Q)给数据项 Q 赋值，在并发控制中应当类似于处理 **write** 操作那样处理 **insert** 操作：

- 在两阶段封锁协议下，如果 T_i 执行 **insert**(Q)操作，就在新创建的数据项 Q 上赋予 T_i 排他锁。
- 在时间戳排序协议下，如果 T_i 执行 **insert**(Q)操作，就把 R-timestamp(Q)与 W-timestamp(Q)的值设置成 TS(T_i)。

15.8.3 谓词读和幻象现象

考虑事务 T_{30}，它在大学数据库上执行以下 SQL 查询：

```
select count(*)
from instructor
where dept_name = 'Physics';
```

事务 T_{30}需要访问 *instructor* 关系中与 Physics 系相关的所有元组。

考虑事务 T_{31}，它执行以下 SQL 插入：

```
insert into instructor
    values(11111,'Feynman','Physics', 94000);
```

698 令 S 是包含事务 T_{30}与 T_{31}的调度。由于下面的原因，我们预计冲突是可能存在的：

- 如果 T_{30}在计算 **count**(*)时使用 T_{31}新近插入的元组，则 T_{30}读取 T_{31}写入的值。因此，在等价于 S 的串行调度中，T_{31}必须先于 T_{30}。
- 如果 T_{30}在计算 **count**(*)时未使用 T_{31}新近插入的元组，则在等价于 S 的串行调度中，T_{30}必须先于 T_{31}。

这两种情况中的第二种让人感到奇怪。T_{30}与 T_{31}没有访问共同的元组，但它们相互冲突！事实上，T_{30}与 T_{31}在一个幻象元组上发生冲突。如果并发控制在元组级粒度上进行，该冲突将难以发现。结果是，系统也许不能防止出现非可串行化调度。我们把这种现象称为**幻象现象**(phantom phenomenon)。

除了幻象问题，我们还需要处理 14.10 节中我们看到的情况，一个事务利用索引查询 *dept_name* = "Physics"的元组，因此它不读取其他系的任何元组。如果另一个事务更新这些元组中的一个，把它的系名称改为 Physics，一个相当于幻象问题的问题则会出现。这些问题都是由谓词读导致的，并且有共同的解决方法。

为防止幻象现象，我们允许 T_{30}阻止其他事务在关系 *instructor* 上创建 *dept_name* = "Physics"的新元组，并且阻止将一个已有的 *instructor* 元组的系名更新为 Physics。

为找到关系 *instructor* 中满足条件 *dept_name* = "Physics"的所有元组，T_{30}必须搜索整个 *instructor* 关系，或至少搜索关系上的一个索引。到现在为止，我们隐含地假设事务所访问的数据项仅仅是元组。然而，事务 T_{30}是一个读取关系中有哪些元组这一信息的事务的例子，而事务 T_{31}则是更新这种信息的事务的例子。

显然，仅仅封锁所访问的元组是不够的；用来找出事务访问的元组的信息也需要封锁。

封锁用来找出要访问的元组的信息可以通过将一个数据项与关系关联在一起来实现；该数据项代

表一种信息，事务用它来查找关系中的元组。读取关系中有哪些元组这一信息的事务，如事务 T_{30}，必须以共享模式封锁对应于该关系的数据项。更新关系中有哪些元组这一信息的事务，如事务 T_{31}，必须以排他模式封锁对应于该关系的数据项加。这样，事务 T_{30} 与 T_{31} 将在一个真实的数据项上发生冲突，而不是在幻象上发生冲突。类似地，使用索引来检索元组的事务必须封锁索引本身。

不要将多粒度封锁中对整个关系的封锁与这儿讲的对对应于关系的数据项的封锁相混淆。通过封锁该数据项，一个事务只是阻止其他事务对关系中有哪些元组这一信息的修改。对元组的封锁仍是需要的。直接访问某个元组的事务可以被授予该元组上的锁，即使另一个事务在对应于关系本身的数据项上持有排他锁。 699

封锁对应于关系的一个数据项或者封锁整个索引的主要缺点是并发程度低——往关系中插入不同元组的两个事务也不能并发执行。

较好的解决方法是使用**索引封锁**(index-locking)技术，它避免封锁整个索引。在关系中插入元组的任何事务必须在该关系上维护的每一个索引中插入有关信息。通过使用索引的封锁协议，我们可以消除幻象现象。为简单起见，我们只考虑 B^+ 树索引。

正如我们在 11 章中看见的那样，每一个搜索码值与索引中的一个叶结点相关联。查询通常使用一个或多个索引来访问关系。插入操作必须在关系的所有索引中插入新元组。在该例子中，假定在关系 *instructor* 的 *dept_name* 上建立了索引。那么，事务 T_{31} 必须修改包含码值"Physics"的叶结点。如果 T_{30} 读取同一叶结点，查找有关 Physics 系的所有元组，则 T_{30} 与 T_{31} 在该叶结点上发生冲突。

通过将幻象现象实例转换为对索引叶结点上封锁的冲突，**索引封锁协议**(index-locking protocol)利用了关系上索引的可用性。该协议运作如下：

- 每个关系至少有一个索引。
- 只有首先在关系的一个或多个索引上找到元组后，事务 T_i 才能访问关系上的这些元组。为了达到索引封锁协议的目的，全表扫描看作一个索引上所有叶结点的扫描。
- 进行查找(不管是区间查找还是点查找)的事务 T_i 必须在它要访问的所有索引叶结点上获得共享锁。
- 在没有更新关系 r 上的所有索引之前，事务 T_i 不能插入、删除或更新关系 r 中的元组 t_i。该事务必须获得插入、删除或更新所影响的所有索引叶结点上的排他锁。对于插入与删除，受影响的叶结点是那些(插入后)包含或(删除前)包含元组搜索码值的叶结点。对于更新，受影响的叶结点是那些(修改前)包含搜索码旧值的叶结点，以及(修改后)包含搜索码新值的叶结点。
- 元组照常获得锁。
- 必须遵循两阶段封锁协议规则。

请注意，索引封锁协议并不关注索引内部结点的并发控制问题。最小化锁冲突的索引并发控制技术将在 15.10 节介绍。 700

封锁一个索引叶结点阻止了该结点上的任何更新，即使这个更新实际上并没有与谓词冲突。一个称为码值封锁的变种将在 15.10 节作为索引并发控制的一部分介绍，它可以使这样的假的锁冲突最小化。

如 14.10 节所指出的，事务之间的冲突看起来取决于低级别的系统查询处理决策，而与用户级别的事务含义无关。另一种并发控制的方法允许给查询中的谓词加共享锁，如关系 *instructor* 上的谓词"*salary* > 90000"。关系上的插入和删除操作都要检查是否满足谓词。若满足，则存在锁冲突，插入和删除操作要等待直到谓词锁被释放。对于更新操作，元组的初始值和最终值都要检查是否满足谓词。这些冲突的插入、删除和更新操作影响谓词选中的元组集合，因此不能允许它们与已获得(共享的)谓词锁的查询并发执行。我们称以上的协议为**谓词锁**(predicate locking)。[⊖]谓词锁在实践中不采用，因为相比较于索引封锁机制，它的实现代价很大，但又不能带来显著的额外好处。

⊖ 术语谓词锁用于对谓词使用共享锁和排他锁的协议的一个版本，因此更加复杂。我们在此给出的版本仅含有在谓词上的共享锁，因此也叫做**精确锁**(precision locking)

谓词锁技术存在一些变种，可以用来消除本章中其他并发控制协议下的幻象现象。然而，许多数据库，如 PostgreSQL（版本 8.1）和 Oracle（版本 10g）（就我们所知）都没有实现索引封锁和谓词锁，而且即使将隔离性级别设置为**可串行化**，也很容易由于幻象问题导致非可串行性。

15.9 实践中的弱一致性级别

14.5 节讨论了 SQL 标准中的隔离级别：可串行化、可重复读、已提交读和未提交读。本节首先简要介绍一些比可串行化级别弱的一致性级别的较老技术，并将它们与 SQL 标准级别相关联。然后我们将讨论在 14.8 节中已简要讨论过的涉及用户交互的事务并发控制问题。

15.9.1 二级一致性

二级一致性（degree-two consistency）的目的是在不必保证可串行性的前提下防止发生级联中止。二级一致性的封锁协议采用两阶段封锁协议中我们用过的同样的两类锁，共享锁（S）和排他锁（X）。事务必须持有适当的锁才能访问数据项，但是两阶段动作不是必需的。

与两阶段封锁的情况大不相同，这里共享锁可以在任何时候释放，并且锁可以在任何时间获得，
701 而排他锁只有在事务提交或中止后才能释放。该协议不保证可串行性。实际上，事务有可能两次读取同一数据项却得到不同结果，如在图 15-20 中，T_{32} 在 T_{33} 写 Q 值之前和之后读取 Q 值。

显然，读取不是可重复读，但是由于直到事务提交前一直持有排他锁，因此没有事务可以读取未提交的值。因此，二级一致性是已提交读隔离性水平的一种特殊实现。

T_{32}	T_{33}
lock-S(Q)	
read(Q)	
unlock(Q)	
	lock-X(Q)
	read(Q)
	write(Q)
	unlock(Q)
lock-S(Q)	
read(Q)	
unlock(Q)	

图 15-20 具有二级一致性的非可串行化调度

15.9.2 游标稳定性

游标稳定性（cursor stability）是二级一致性的一种形式，它是为利用游标对关系中的元组进行迭代的程序而设计的。它不封锁整个关系，游标稳定性保证：

- 正被迭代处理的元组被加上共享锁。
- 任何被更改的元组被加上排他锁，直至事务提交。

这些规则保证了二级一致性，不要求两阶段封锁，没有保证调度的可串行性。实践中游标稳定性用于频繁访问的关系，以作为一种提高并发性和改善系统性能的方法。无论是否存在非可串行化的调度，利用游标稳定性的应用程序必须在编码上确保数据库的一致性。所以，游标稳定性的用途局限于某些专门的并且具有简单一致性约束的情况。

15.9.3 跨越用户交互的并发控制

702 并发控制协议通常考虑的是不涉及用户交互的事务。现在考虑 14.8 节中涉及用户交互的航空公司座位选择的例子。假设我们从初始时将座位空闲情况显示给用户，直到确定座位选择的所有步骤看作一个事务。

如果使用两阶段封锁协议，飞机上所有座位都被加上共享锁，直到用户选择完座位，在这期间其他事务不允许更新座位分配情况。显然，这种锁是非常糟糕的，因为用户可能需要很长的时间来做出选择，甚至放弃交易但不显式地取消。可以采用时间戳协议或者有效性检验来代替，避免加锁出现的问题，但是这两种协议会在另一个用户 B 更新座位分配信息时中止用户 A 的事务，即使用户 B 选择的座位和用户 A 选择的座位并不冲突。快照隔离是在这种情况下最好的选择，因为只要用户 B 没有选择和用户 A 相同的座位，A 的事务就不会中止。

然而，快照隔离要求数据库记录一个事务的更新信息，即便该事务已经提交，但只要任何其他并发的事务仍然是活跃的，这对于长事务会存在问题。

另一种方法是将涉及用户交互的事务划分成两个或者更多的事务，使得没有事务跨越用户交互。如果我们的座位选择事务按照这样进行划分，则第一个事务将读取空闲座位，第二个事务将完成分配选择的座位。如果第二个事务编写不慎，会出现在分配座位给用户的时候，没有检查该座位是否同时分配给其他用户，导致更新丢失的问题。为避免这个问题，正如 14.8 节所述，第二个事务只有在座位

没有同时分配给其他用户时才能将其分配。

上述思想在另一种并发控制机制中概化了，该机制使用存储在元组中的版本号来避免更新丢失。每个关系模式中添加一个额外的版本号属性，当创建元组时初始化为0。当一个事务(第一次)读取它试图更新的一个元组时，它记录该元组的版本号。读操作作为一个独立的数据库事务执行，因此任何所获得的锁被立即释放。更新操作在本地完成，并作为提交过程的一部分拷贝到数据库中，采用作为原子执行的以下步骤(即作为数据库事务的一部分)：

- 对于每个更新的元组，事务检查当前的版本号是否等于第一次读取事务时的版本号。
  1. 如果版本号匹配，则执行数据库中该元组的更新，并且它的版本号加1。
  2. 如果版本号不匹配，事务中止，回滚它所执行的所有更新。 703

如果对于所有更新元组的版本号检查成功，事务提交。值得一提的是时间戳可以用来代替版本号，而对机制没有任何影响。

注意到上述机制和快照隔离的相似性。版本号检查实现了快照隔离中的先提交者获胜规则，而且在事务活跃时间很长的情况下也可使用。然而，与快照隔离不同，事务的读操作可能不对应数据库中的一个快照。与有效性检验协议不同，事务的读操作不进行检验。

我们将以上的机制称为**不做读有效性检查的乐观并发机制**(optimistic concurrency control without read validation)。不做读有效性检验的乐观并发控制机制提供了一种弱的串行化水平，并不保证可串行化。该机制的一个变种是在提交的时候，除了检验写操作外，还采用版本号来检验读操作，以保证事务读取的元组在初次读取之后没有更新。这种机制即我们前面看到的乐观并发控制机制。

上述机制已经被应用程序开发人员广泛地用在涉及用户交互的事务处理中。该机制的一个诱人的特点是它可以轻松地实现在数据库系统顶层。作为提交过程的一部分，检验和更新操作作为一个单独的数据库事务执行，并采用数据库的并发控制机制保持提交过程的原子性。以上机制同样用在Hibernate对象关系映射系统(见9.4.2节)和其他对象关系映射系统中，并称为乐观并发控制(即使默认读操作不检验)。在Hibernate中涉及用户交互的事务称为**对话**(conversation)，以区分它们和采用版本号的常规事务检验。对象关系映射系统还将数据库元组在内存中以对象的形式进行缓存，在缓存的对象上执行事务。对象上的更新在事务提交时转化为数据库上的更新。数据可能会在缓存中存在很长的时间，如果事务更新缓存的数据，则会有更新丢失的风险。因此，Hibernate和其他关系对象映射系统采用透明的版本号检查作为提交过程的一部分。(如果需要可串行化，Hibernate允许程序员绕开缓存，直接在数据库上执行事务。)

15.10 索引结构中的并发**

对索引结构访问的处理可以像处理访问其他数据库结构那样进行，并应用前面讲述过的并发控制技术。然而，由于索引访问频繁，它们将成为封锁竞争的集中点，从而导致低并发度。幸运的是，索引不必像其他数据库结构那样处理。事务在两次索引查找期间，发现索引结构发生了变化，这是完全可以接受的，只要索引查找返回正确的元组集。因此，只要维护索引的准确性，对索引进行非可串行 704
化并发存取是可接受的。

我们讲述两种并发访问B^+树的技术。关于处理B^+树的其他技术，以及处理其他索引结构的技术可参阅的相关文献请参见文献注解。

我们所讲述的处理B^+树的技术基于封锁机制，但既不采用两阶段封锁也不采用树形协议。用于查找、插入与删除的算法是第11章中使用的算法，只是做了小小的修改。

第一中技术叫作**蟹行协议**(crabbing protocol)：

- 当查找一个码值时，蟹行协议首先用共享模式锁住根结点。沿树向下遍历，它在子结点上获得一个共享锁，以便向更远处遍历，在子结点上获得锁以后，它释放父结点上的锁。它重复该过程直至叶结点。
- 当插入或删除一个码值时，蟹行协议采取如下行动：
  - 采取与查找相同的协议直至希望的叶结点，到此为止，它只获得(和释放)共享锁。

□ 它用排他锁封锁该叶结点，并且插入或删除码值。

□ 如果需要分裂一个结点或将它与兄弟结点合并，或者在兄弟结点之间重新分配码值，蟹行协议用排他锁封锁父结点，在完成这些操作后，它释放该结点和兄弟结点上的锁。

如果父结点需要分裂、合并或重新分布码值，该协议保留父结点上的锁，以同样方式分裂、合并或重新分布码值，并且传播更远。否则，该协议释放父结点上的锁。

该协议的名字来源于螃蟹走路的方式：先移动一边的腿，然后另一边的，如此交替进行。该协议的封锁过程，从上往下和从下往上(发生分裂、合并或重新分布的情况)，就像螃蟹移动一样。

一旦某个操作释放了一个结点上的锁，别的操作就可以访问该结点。在向下的搜索操作和由于分裂、合并或重新分布传播向上的操作之间有可能出现死锁，系统能很容易地处理这种死锁，它先让搜索操作释放锁，然后从树根重启它。

705 第二种技术使用一个改进版本的 B^+ 树，称为 **B-link 树**(B-link tree)，避免了在获取另一个结点的锁时还占有一个结点的锁，获得更多的并发性。B-link 树要求每个结点(包括内部结点，不仅仅是叶结点)维护一个指向右兄弟结点的指针，这个指针是必要的，因为在一个结点正在分裂时进行的查找可能不仅要查找该结点而且可能要查找该结点的右兄弟结点(如果存在的话)。我们以后将用一个例子来说明这个技术。但首先我们列出修改后的 **B-link 树封锁协议**(B-link-tree locking protocol)。

- **查找**：B^+ 树的每个结点在访问之前必须加共享锁。非叶结点上的锁应该在对 B^+ 树的其他任何结点发出加锁请求前释放。如果结点分裂与查找同时发生，所希望的搜索码值可能不再位于查找过程中所访问的某个结点所代表的那些值的范围内。在这种情况下，搜索码值在兄弟结点所代表的范围内，系统循着指向右兄弟结点的指针能找到该兄弟结点。不过，叶结点的封锁遵循两阶段封锁协议以避免幻象现象，如 15.8.3 节所述。
- **插入与删除**：系统遵循查找规则，定位要进行插入或删除的叶结点。该结点的共享锁升级为排他锁，然后进行插入或删除。受插入或删除影响的叶结点封锁遵循两阶段封锁协议，以避免幻象现象，如 15.8.3 节所述。
- **分裂**：如果事务使一个结点分裂，则按 11.3 节的算法创建新结点并作为原结点的右兄弟。设置原结点和新产生结点的右兄弟指针。接着，事务释放原结点的排他锁(假若它是一个内部结点；叶结点以两阶段形式加锁)，然后发出对父结点加排他锁的请求，以便插入指向新结点的指针。(没有必要对新结点加锁或解锁。)
- **合并**：执行删除后，如果一个结点的搜索码值太少，则必须将要与之合并的那个结点加上排他锁。一旦这两个结点合并，就发出对父结点加排他锁的请求，以便删除要被删除的结点。此时，事务释放已合并结点的锁。除非父结点也需再合并，不然释放其锁。

注意这个重要的事实：插入与删除操作可能封锁一个结点，释放该结点，然后又对它重新封锁。此外，与分裂或合并操作并发执行的查找可能发现所需搜索码值被分裂或合并操作移到右兄弟结点。

为说明这一点，考虑图 15-21 所示的 B-link 树。假设有两个针对该 B-link 树的并发操作：

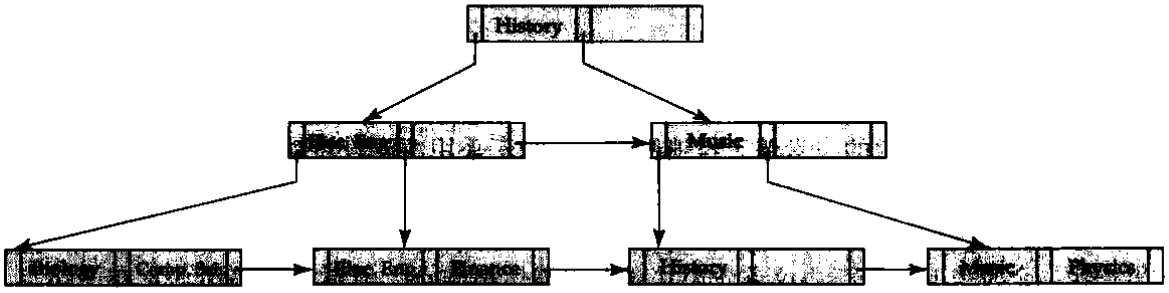

图 15-21 *department* 文件的 n = 3 的 B-link 树

1. 插入“Chemistry”。

706 2. 查找“Comp. Sci.”。

我们假设插入操作首先开始。它为“Chemistry”执行查找，发现要插入“Chemistry”的结点已满。于

是它将该结点的共享锁转换为排他锁，并创建新结点。这样，原结点包含搜索码值“Biology”与“Chemistry”。新结点包含搜索码值“Comp. Sci.”。

现在假设发生上下文切换，导致控制传递给查找操作。该查找操作访问根结点，然后沿指向根结点左子结点的指针而下。接着访问那个结点，并得到指向左子结点的指针。该左子结点原先包含搜索码值“Biology”与“Comp. Sci.”。由于该结点目前被插入操作以排他方式封锁，因此查找操作必须等待。注意，此时查找操作不持有任何锁！

插入操作现在释放了叶结点并重新对父结点加锁，这次以排他方式封锁。它完成了插入操作，得到图 15-22 所示的 B-link 树。查找操作继续进行。然而，它拥有的指针指向错误的叶结点。于是它顺着右兄弟结点指针定位下一个结点。如果该结点仍不正确，则继续顺着该结点的右兄弟指针查找。可以证明，如果查找操作拥有指向错误结点的指针，则沿着右兄弟结点指针，查找操作最终可以到达正确的结点。

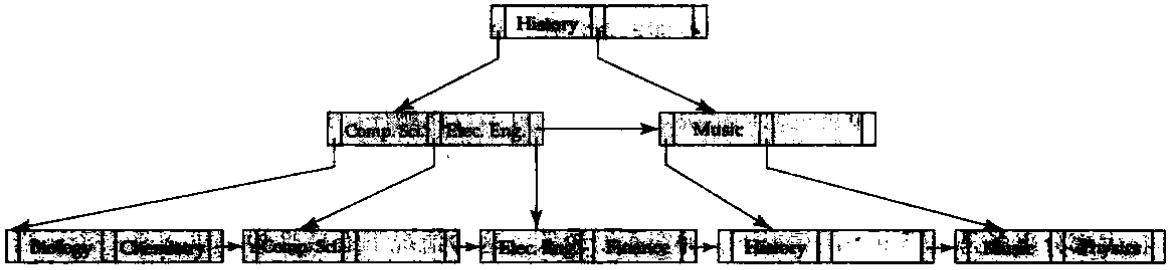

图 15-22 插入“Chemistry”到图 15-21 所示的 B-link 树中

查找与插入不会引起死锁。删除操作时的结点合并可能引起不一致性，因为查找操作可能在父结点更新前已经从父结点读取了指向被删除结点的指针，然后试图访问被删除的结点，查找操作将必须从根结点重新开始。不对结点进行合并可以避免这样的不一致性。这个解决方案使某些结点包含的搜索码值太少，违背了 B^+ 树的某些特性。然而，在大部分数据库中，插入操作比删除操作频繁，因此包含搜索码值太少的结点可能较快地得到更多的值。

一些索引并发控制机制不是以两阶段形式封锁索引叶结点，而是对个别的码值使用**码值封锁**(key-value locking)，允许其他码值从同一个叶结点插入或删除，这样码值封锁提供了增强了的并发性。但是，朴素地使用码值封锁可能引发幻象现象；为防止幻象现象可以采用**下一码封锁**(next-key locking)技术。在这个技术中，每一次索引查找不仅封锁查找范围内的多个码(或单个码，在点查找时)，而且封锁下一个码值——也就是刚好比范围内最后一个码值大的码值；并且，每一次插入必须不仅封锁要插入的值，而且包括下一个码值。这样，如果一个事务试图插入一个值到另一个事务的索引查找范围之内时，这两个事务将在插入码值的下一个码值上冲突。同样，删除也必须封锁被删除值的下一个码值，来保证检测得到它与别的查询的查找范围的并发冲突。

15.11 总结

- 当多个事务在数据库中并发地执行时，数据的一致性可能不再维持。系统有必要控制各事务之间的相互作用，这是通过称为*并发控制机制*的多种机制中的一种来实现的。
- 为保证可串行性，我们可以使用多种并发控制机制。所有这些机制要么延迟一个操作，要么中止发出该操作的事务。最常用的机制是各种封锁协议、时间戳排序机制、有效性检查技术与多版本机制。
- 封锁协议是一组规则，这些规则阐明了事务何时对数据库中的数据项进行加锁和解锁。
- 两阶段封锁协议仅在一个事务未曾释放任何数据项上的锁时才允许该事务封锁新数据项。该协议保证可串行性，但不能避免死锁。在没有关于数据项访问方式信息的情况下，两阶段封锁协议对于保证可串行性既是必要的又是充分的。
- 严格两阶段封锁协议要求事务持有的所有排他锁必须在事务结束时方可释放，其目的是保证结果调度的可恢复性和无级联性，强两阶段封锁协议要求事务持有的所有锁必须在事务结束时方可释放。
- 基于图的封锁协议对访问数据项的顺序加以限制，从而不需要使用两阶段封锁还能够保证可串行性，

而且又能够保证不会产生死锁。

- 许多种封锁协议都不能防止死锁。一种可以防止死锁的方法是使用数据项的一种顺序，并且按与该顺序一致的次序申请加锁。
- 另一种防止死锁的方法是使用抢占与事务回滚。为控制抢占，我们给每个事务赋予一个唯一时间戳。这些时间戳用于决定事务是等待还是回滚。如果一个事务回滚，它在重启时保持原有时间戳。wound-wait 机制是一个抢占机制。
- 如果没有预防死锁，系统必须用死锁检测与恢复机制来处理它们。为此，系统构造了一个等待图。当且仅当等待图包含环时，系统处于死锁状态。当一个检测算法判定死锁存在，系统必须从死锁中恢复。系统通过回滚一个或多个事务来解除死锁。
- 某些情况下把多个数据项聚为一组，将它们作为聚集数据项来处理，其效果可能更好，这就导致了粒度的多个级别。我们允许各种大小的数据项，并定义数据项的层次，其中小数据项嵌套于大数据项之中。这种层次结构可以图形化地表示为树。封锁按从根结点到叶结点的顺序进行；解锁则按从叶结点到根结点的顺序进行。该协议保证可串行性，但不能避免死锁。
- 时间戳排序机制通过事先在每对事务之间选择一个顺序来保证可串行性。系统中的每个事务对应一个唯一的固定时间戳。事务的时间戳决定了事务的可串行化顺序。这样，如果事务 T_i 的时间戳小于事务 T_j 的时间戳，则该机制保证产生的调度等价于事务 T_i 出现在事务 T_j 之前的一个串行调度。该机制通过回滚违反该次序的事务来保证这一点。
- 在大部分事务是只读事务的情形下，冲突频度较低，这种情况下有效性检查机制是一个适当的并发控制机制。系统中的每个事务对应一个唯一的固定时间戳。串行性次序是由事务的时间戳决定的。在该机制中，事务不会延迟。不过，事务要完成必须通过有效性检查。如果事务未通过有效性检查，则该事务回滚到初始状态。
- 多版本并发控制机制基于在每个事务写数据项时为该数据项创建一个新版本。读操作发出时，系统选择其中的一个版本进行读取。利用时间戳，并发控制机制保证按确保可串行性的方式选取要读取的版本。读操作总能成功。
  - □ 在多版本时间戳排序中，写操作可能引起事务的回滚。
  - □ 在多版本两阶段封锁中，写操作可能导致封锁等待或死锁。
- 快照隔离是一种基于有效性检验的多版本并发控制协议，与多版本两阶段封锁协议不同，它不需要将事务声明为只读或更新的。快照隔离不保证可串行化，但是许多数据库系统仍然支持它。
- 仅当要删除元组的事务在该元组上具有排他锁时，**delete** 操作才能够进行。在数据库中插入新元组的事务在该元组上被授予排他锁。
- 插入操作可能导致幻象现象，这时插入操作与查询发生逻辑冲突，尽管两个事务可能没有存取共同的元组。如果封锁仅加在事务访问的元组上，这种冲突就检测不到。关系中用于查找元组的数据需要加锁，索引封锁技术要求对某些索引结点加锁来解决这个问题。所加的这些锁保证所有事务在实际的数据项上发生冲突，而不是在幻象上。
- 弱级别的一致性用于一些应用中，在这些应用中，查询结果的一致性不是至关重要的，而使用可串行性会使查询对事务的处理起反作用。二级一致性是这种弱级别的一致性之一，游标稳定性是二级一致性的一个特例，而且已广泛应用。
- 跨越用户交互的事务并发控制是一个有挑战性的任务。应用程序通常实现一种基于采用元组中存储的版本号来验证写操作的机制。这种机制提供了弱可串行化水平，而且可以实现在应用层，而无需修改数据库。
- 可以为特殊的数据结构开发特殊的并发控制技术。通常，特殊的技术用到 B^+ 树上，以允许较大的并发性。这些技术允许对 B^+ 树进行非可串行化访问，但它们保证 B^+ 树结构是正确的，并保证对数据库本身的存取是可串行化的。

709
|
710

术语回顾

- 并发控制
- 锁类型
  - 共享(S)锁
  - 排他(X)锁
- 锁
  - 相容性
  - 申请
  - 等待
  - 授予
- 死锁
- 饿死
- 封锁协议
- 合法调度
- 两阶段封锁协议
  - 增长阶段
  - 缩减阶段
  - 封锁点
  - 严格两阶段封锁
  - 强两阶段封锁
- 锁转换
  - 升级
  - 降级
- 基于图的协议
  - 树形协议
  - 提交依赖
- 死锁处理
  - 预防
  - 检测
  - 恢复
- 死锁预防
  - 顺序加锁
  - 抢占锁
  - wait-die 机制
  - wound-wait 机制
  - 基于超时的机制
- 死锁检测
  - 等待图
- 死锁恢复
  - 全部回滚
  - 部分回滚
- 多粒度
  - 显式锁
  - 隐式锁
  - 意向锁
- 意向型锁
  - 共享型意向(IS)
  - 排他型意向(IX)
  - 共享排他型意向(SIX)
- 多粒度封锁协议
- 基于时间戳的协议
- 时间戳
  - 系统时钟
  - 逻辑计数器
  - W-timestamp(Q)
  - R-timestamp(Q)
- 时间戳排序协议
  - Thomas 写规则
- 基于有效性检查的协议
  - 读阶段
  - 有效性检查阶段
  - 写阶段
  - 有效性测试
- 多版本并发控制
- 多版本时间戳排序
- 多版本两阶段封锁
  - 只读事务
  - 更新事务
- 快照隔离
  - 更新丢失
  - 先提交者获胜
  - 先更新者获胜
  - 写偏斜
  - select for update
- 插入和删除操作
- 幻象现象
- 索引封锁协议
- 谓词锁
- 弱一致性级别
  - 二级一致性
  - 游标稳定性
- 不做读有效性验证的乐观并发控制
- 对话
- 索引中的并发
  - 蟹行协议
  - B-link 树
  - B-link 树封锁协议
  - 下一码封锁

实践习题

15.1 证明两阶段封锁协议保证冲突可串行化，并且事务可以根据其封锁点串行化。

15.2 考虑下面两个事务：

T_{34}: **read**(A);
read(B);
if $A=0$ **then** $B:=B+1$;
write(B);

T_{35}: **read**(B);
read(A);
if $B=0$ **then** $A:=A+1$;
write(A).

给事务 T_{34} 与 T_{35} 增加加锁、解锁指令，使它们遵从两阶段封锁协议。这两个事务会引起死锁吗？

15.3 强两阶段封锁协议带来什么好处？它与其他形式的两阶段封锁协议相比有何异同？

15.4 考虑一个按有根树方式组织的数据库。假设我们在每对结点之间插入一个虚结点。证明，如果我们在

711
|
712

由此构成的新树上遵从树形协议，我们可以得到的并发度比在原始树上遵从树形协议的更高。

15.5 用例子证明：存在在树形封锁协议下可行，而在两阶段封锁协议下不可行的调度，反之亦然。

15.6 考虑以下对树形封锁协议的扩展，它既允许使用共享锁又允许使用排他锁：

- 事务可以是只读事务，在这种情况下它只申请共享锁；也可以是更新事务，此时只申请排他锁。
- 每个事务必须遵从树形协议规则。只读事务可以首先封锁任何数据项，而更新事务必须首先封锁根结点。

证明该协议保证可串行性并能避免死锁。

15.7 考虑以下基于图的封锁协议，它只允许加排他锁，并且在带根有向无环数据图上运作：

- 事务首先可以封锁任何结点。
- 要封锁任何其他的结点，事务必须在该结点的大部分父结点上持有锁。

证明该协议保证可串行性并能避免死锁。

15.8 考虑以下基于图的封锁协议，它只允许加排他锁，并且在带根有向无环数据图上运作：

- 事务首先可以封锁任何结点。
- 要封锁任何其他的结点，事务必须已经访问该结点的所有父结点，并且必须在该结点的一个父结点上持有锁。

证明该协议保证可串行性并能避免死锁。

15.9 在持久化程序设计语言中封锁不是显式进行的。访问对象（或相应页）时必须加锁。大部分现代操作系统允许用户对页面设置访问保护（不许访问、读、写），并且违反存取保护的内存访问将导致违反保护错误（如，参见 UNIX 的 mprotect 命令）。说明访问保护机制在持久化程序设计语言中如何用于页级封锁。

15.10 考虑除 **read** 与 **write** 操作之外还包含原子操作 **increment** 的一个数据库系统。令 V 是数据项 X 的值。操作：

$$\textbf{increment}(X) \text{ by } C$$

在一个原子步骤中将 X 的值设为 $V + C$。如果事务不执行 read(X)，则事务不能获知 X 的值。图 15-23表示三种锁类型的锁相容阵：共享型、排他型和增量型。

a. 证明：如果所有事务按相应的类型封锁它们所访问的数据项，则两阶段封锁保证可串行性。

b. 证明：包含增量型锁可以增加并发度。（提示：在所举的银行例子中，考虑支票的票据交换事务）。

15.11 在时间戳排序中，**W-timestamp**(Q) 表示成功执行 **write**(Q) 的所有事务的最大时间戳。现在，假设我们将之定义为最近成功执行 **write**(Q) 的事务的时间戳。这种措辞上的变化会带来什么不同？解释你的答案。

15.12 采用多粒度封锁机制比采用单封锁粒度的等价系统需要更多或更少的锁。针对每种情况各举一例，并比较所允许的相对并发量。

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

图 15-23 锁相容矩阵

15.13 考虑 15.5 节基于有效性检查的并发控制机制。证明：若选择 Validation(T_i) 而不是 Start(T_i) 作为事务 T_i 的时间戳，则如果事务间发生冲突的次数确实很低，我们可望有较好的响应时间。

713 ~ 714

15.14 对于下面的每个协议，说明促使你使用某个协议的实际应用原因以及不使用的原因：

- 两阶段封锁。
- 具有多粒度封锁的两阶段封锁。
- 树形协议。
- 时间戳排序。
- 有效性检查。
- 多版本时间戳排序。
- 多版本两阶段封锁。

15.15 解释为什么使用下述事务执行技术会比仅仅使用严格两阶段封锁能够得到更好的性能：跟基于有效性检查技术中一样，首先执行事务而无须获得任何锁，也不向数据库执行任何写操作。但跟有效性

检查技术中不一样的是它既不检查有效性也不在数据库上执行写操作，而是采用严格两阶段封锁重新运行事务。（提示：考虑磁盘 I/O 等待。）

15.16 考虑时戳排序协议，以及两个事务，一个执行写两个数据项 p 和 q，另一个执行读这两个数据项。试给出一个调度，使得第一个事务写操作的时间戳测试失败，引起该事务重启，并依次引起另一个事务的级联中止。并说明是怎样导致这两个事务都饿死的。（两个或多个进程执行，但它们都由于与其他进程的交互作用而无法完成它们的任务，这种情形叫做活锁（livelock）。）

15.17 设计一个基于时间戳的能避免幻象现象的协议。

15.18 假设我们采用 15.1.5 节中的树形协议来管理对 B^+ 树的并发访问，由于在影响到根结点的插入中也可能发生分裂，因此看来插入操作在整个操作完成之前都不能释放锁。在什么情况下有可能提前释放锁呢？

15.19 快照隔离采用有效性检验步骤，当事务 T 写数据项之前，检查 T 是否有其他并发事务已经写过该数据项。

a. 一种简单的实现方式对于每个事务采用一个开始时间戳和一个提交时间戳，另外还有一个更新集合来记录事务更新的数据项。解释如何利用事务时间戳和更新集合来实现先提交者获胜机制中的检验。你可以假设检验和其他提交过程是串行执行的，即一次只有一个事务。 715

b. 解释如何修改上述机制，不用更新集合，而为每个数据项分配一个写时间戳，来实现先提交者获胜机制下有效性检验步骤作为提交过程的一部分。同样，你可以假设检验和其他提交过程是串行执行的。

c. 先提交者获胜机制可以采用上述时间戳来实现，除了当获得排他锁时立刻执行有效性检验，而不是在提交时执行。

i. 解释如何给数据项分配写时间戳来实现先提交者获胜机制。

ii. 说明由于锁机制，如果提交时重复有效性检验，结果不会发生变化。

iii. 解释在这种情况下，为什么没有必要串行地执行有效性检验和其他提交过程。

习题

15.20 严格两阶段封锁协议带来什么好处？会产生哪些弊端？

15.21 大部分数据库系统实现采用严格两阶段封锁协议。说明该协议流行的三点理由。

15.22 考虑树形协议的一个变种，它称为森林协议。数据库按有根树的森林的方式组织。每个事务 T_i 必须遵从以下规则：

- 每棵树上的首次封锁可以在任何数据项上进行。
- 树上的第二次以及此后的封锁申请仅当被申请结点的父结点上有锁时才能发出。
- 数据项解锁可在任何时候进行。
- 事务 T_i 释放数据项后不能再次封锁该数据项。

证明森林协议不保证可串行性。

15.23 在什么条件下避免死锁比允许死锁发生然后检测的方式代价更小？

15.24 如果通过死锁避免机制避免了死锁后，饿死仍有可能吗？解释你的答案。

15.25 在多粒度封锁中，隐式封锁与显式封锁有什么不同？

15.26 尽管 SIX 锁在多粒度封锁中很有用，但排他共享意向（XIS）锁则无用。为什么？ 716

15.27 多粒度协议中的规则指出，仅当事务 T_i 当前对 Q 的父结点持有 IX 或 IS 锁时，T_i 对结点 Q 可加 S 或 IS 锁。已知 SIX 和 S 锁比 IX 和 IS 锁更强，为什么协议不允许当父结点持有 SIX 或 S 锁时对该结点加 S 或 IS 锁？

15.28 当一个事务在时间戳排序协议下回滚，它被赋予新时间戳。为什么它不能简单地保持原有时间戳？

15.29 证明：存在满足两阶段封锁协议却不满足时间戳协议的调度，反之亦然。

15.30 在时间戳协议的一个修改版中，我们要求测试提交位以判定 read 请求是否必须等待。解释提交位如何防止级联中止。为什么该测试对 **write** 请求是不必要的。

15.31 如练习 15.19 讨论的，快照隔离可以通过时间戳有效性检验的形式来实现。然而，与保证可串行化的多版本时间戳排序机制不同，快照隔离不保证可串行化。解释导致这种差异结果的两种协议之间

关键的区别。

15.32 列出练习 15.19 描述的基于时间戳实现的先提交者获胜版本的快照隔离与 15.9.3 节描述的不做读有效性检验的乐观并发控制的主要相似点和区别。

15.33 解释幻象现象。为什么尽管采用两阶段封锁协议，该现象仍可能导致不正确的并发执行？

15.34 解释使用二级一致性的原因，这种方法有什么缺点？

15.35 请给出码值封锁调度的例子，说明如果查找、插入、删除操作中的任何一个不对下一码值进行封锁，都可能出现未发现的幻象。

15.36 许多事务更新一个公共数据项(例如，一个支行的现金余额)和若干私有数据项(例如，多个个人账户余额)。解释如何通过对事务操作进行排序来提高并发度(和吞吐量)。

15.37 考虑下面这个封锁协议：所有数据项都被编号，并且当解锁一个数据项时，只有标号更大的数据项才可以加锁。锁可以在任何时候释放。只能使用排他锁。用例子说明这个协议不能保证可串行化。

717

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，包括并发控制的概念和实现细节。Bernstein 和 Newcomer[1997] 在其教科书中讨论了事务处理的多个方面，包括并发控制。

两阶段封锁协议由 Eswaran 等[1976]引入。树形协议来自 Silberschatz 与 Kedem[1980]。其他的工作在更一般的图上的非两阶段封锁协议由 Yannakakis 等[1979]、Kedem 与 Sileberschatz[1983]，以及 Buchley 与 Silberschatz[1985]提出。kovth[1983]探讨了由基本的共享和排他锁方式可以得到的多种封锁方式。

实践习题 15.4 来自 Buckley 与 Silberschatz[1984]。实践习题 15.6 来自 Kedem 与 Silberschatz[1983]。实践习题 15.7 来自 Kedem 与 Silberschatz[1979]。实践习题 15.8 来自 Yannakakis 等[1979]。实践习题 15.10 来自 Korth[1983]。

多粒度数据项封锁协议来自 Gray 等[1975]。Gray 等[1976]给出了详细的描述。Kedem 和 Silberschatz[1983]对任意封锁方式(允许更多的语义而不仅仅是读和写)的多粒度封锁做了规范化。这种方法包括称为*更新型锁*的一类锁，以处理锁转换。Carey[1983]将多粒度的想法扩展到基于时间戳的并发控制。为保证不产生死锁而产生的一种扩展协议由 Korth[1982]给出。

基于时间戳的并发控制机制来自 Reed[1983]。Buckley 与 Silberschatz[1983]给出了一种不须回滚且保证可串行化的时间戳算法。有效性检查的并发控制机制来自 Kung 与 Robison[1981]。

多版本时间戳排序由 Reed[1983]引入。Silberscatz[1982]中给出了一种多版本树封锁协议。

二级一致性在 Gray 等[1975] 中介绍，SQL 中的一致性(或隔离性)的级别在 Berenson 等[1995]中做了解释和评论。许多商业数据库系统使用与加锁相结合的基于版本的方法。PostgreSQL、Oracle、和 SQL Server 全部支持在 15.6.2 节中提到的快照隔离协议。细节请分别参考第 27、28 和 30 章。

需要注意的是，在 PostgreSQL (版本 8.1.4)和 Oracle (版本 10g)中将隔离级别设置为串行化的结果是采用快照隔离，并不保证可串行性。Fekete 等[2005]介绍如何通过重写事务引入冲突，使得在快照隔离下保证可串行化执行。这些冲突保证在快照隔离下事务不能并发执行。Jorwekar 等[2007]介绍了一种方法，给定一个运行在快照隔离下的(参数化)事务集合，该方法可以检验这些事务是否存在非串行化的风险。

Bayer 与 Schkolnick[1977]和 Johnson 与 Shasha[1993]对 B^+ 树中的并发作了研究。15.10 节所给技术基于 Kung 与 Lehman[1980]，以及 Lehman 与 Yao[1981]。ARIES 系统中采用的码值封锁技术为 B^+ 树访问提供了极高的并发性，在 Mohan[1990a]、Mohan 和 Narang[1992]中描述。Ellis[1987]给出了一个用于线性散列的并发控制技术。

718 ~ 720