高级数据库系统及其应用

第2部分　关系数据库系统实现

第8章 事务并发控制

xshxie@ustc.edu.cn

LOGO

第8章 事务并发控制

8.1 事务并发执行管理概述

8.2 基于封锁的并发控制

8.3 死锁及其处理

8.4 扩展封锁处理技术

8.5 基于优化的并发控制

8.6 SQL-92的事务支持

8.1 事务并发执行管理概述

8.1.1 事务的概念与基本特性

8.1.2 调度的基本概念

8.1.3 事务的并发执行

8.1.4 优先图

8.1.5 视可串行化

DB 事务

DBMS内核会自动将来自一条或多条SQL语句的用户读写DB请求，划分为若干个事务。每个事务有自己的局部内存，由一系列读/写DB数据元素的动作构成。

DB(数据)元素与DB动作

DB元素

- 泛指DB关系表、数据页，或元组等；
- 具体要视应用场合或上下文而定。

DB动作(actions)

- 读一个DB元素X---R(X)
  - X：磁盘页➔DB缓冲区➔局部内存(变量t中)。
- 写一个DB元素X---W (X)
  - X：磁盘页←DB缓冲区←局部内存(变量t中)。

☆读/写DB元素，涉及三个地址空间：

- 磁盘空间，DB缓冲区，(程序/事务)局部内存。

- 其它动作辅助动作

8.1.1 事务的基本概念与特性

❖事务（Transaction）定义

- 指来自DBMS中一个用户会话的、具有一定偏序的DB动作序列。
  - 实际DB事务中也可能含有一些不影响并发的、事务管理一般不显式列出的其它动作，如读写OS文件、内部赋值计算等。
  - 所谓**偏序**，是指事务中有些关键动作顺序很重要，但也有些动作的前后相对顺序并不重要。

并发事务的串行与并发执行

- 串行执行(确保并发事务正确执行的保守做法)
  - 想象用户A/B/C依次先后提交事务，则安全的做法应按{T_a动作序列}{T_b动作序列} {T_c动作序列}执行
  - 合理逻辑：B应用到了A的结果[除非A出错]，但没有用C的结果 ；同一事务连读同一个元素，结果应相同
- 并发执行(是DBMS提高系统性能的必然要求)
  - 这可能导致DB状态不一致. 四种典型异常情形：
    - 读[先启动事务]未提交的数据（WR冲突） （脏读）
    - （同一事务内出现）不可重复的读（WR冲突）
    - （前后事务间出现）重写未提交的数据（WW冲突）
    - （并发事务中）包含有中止事务 （WW冲突）
- 但无论如何,必须确保合理逻辑—单事务的ACID

单个事务的ACID特性

- **原子性(atomic)**
  - 事务的执行具有原子性。要么成功，所有事务动作都被正常执行；要么失败，任何一个事务动作都未被执行。如果执行中间出错或被中止，则之前已执行过的那些动作必须全部撤销，事务回滚(roll back)。
- **一致性(consistency)**
  - 当事务独立执行时，必须能保持DB状态的一致性。
- **孤立性(isolation)**
  - 尽管DBMS内核通常以并发形式管理事务，但对用户而言，仍是以事务被‘孤立’执行，来理解其执行效果。
- **持久性(durability)**
  - 当事务被成功执行后，它对DB数据造成的影响应是持久性的；即使是发生系统突然崩溃时，也应如此。

8.1.2 调度的基本概念

- 调度(Scheduling)定义
  - 指对一组可能来自多个不同事务动作流的一种偏序序列安排
  - 常用隐含时间轴的列表方法直观表示。
- 完全调度complete schedule
  - 相关事务含Commit或Abort动作
- 串行调度（series schedule）
  - 来自不同事务的动作没有交替。

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(B)$
	$W_2(B)$
$R_1(C)$	
$W_1(C)$	

图8.1 一个含两事务的调度

可恢复调度与可串行调度

严格调度

- 对调度的任意事务T，要求T所写的值，在T提交或中止之前，不会被S中其它事务读或重写

可恢复调度

- 如果一个调度S只包含满足以下规定的事务T_i，则称调度S是可恢复的。
  - 令事务T_i读取了DB元素集$\{X\}$，而$\{T_j \mid j \neq i\}$是至少改变了元素集合$\{X\}$中一个元素的所有事务集，则T_i必须在$\{T_j \mid j \neq i\}$中所有事务都提交后才能提交。

可串行化调度

- 当一个包含一组提交事务的调度执行时，如果对任何一致性DB造成的影响，与某个包含同组事务的串行调度相同。

冲突等价与冲突可串行化

- 冲突定义
  - 称两个动作是冲突的或不可交换的，如果它们属于同一事务；或如果它们是来自不同事务的、都对同一DB元素操作的两动作，且其中至少有一个是写。
- 称两个调度**S1**和**S2**冲突等价，如果满足：
  - 在S1与S2中，都包含了同一组事务；
  - S1中任意两个提交事务的每个冲突动作对的先后顺序，与S2中发生的情况相同。
- 冲突可串行化
  - 称调度S是冲突可串行化的，如果S与某个串行调度的冲突等价。

视可串行化(view serializability)

- 视等价定义：令两调度**S1**和**S2**包含同样的事务集，如果满足以下三个条件，则称它们视等价。
  - 若在$S1$中事务T_i读元素A的初值，则在$S2$中事务T_i也必须读元素A的初值；
  - 若在$S1$中事务T_i读被事务T_j写过的元素A值，则在$S2$中事务T_i也必须读被事务T_j写过的元素A值；
  - 若$S1$中由某事务T_i执行了对DB元素A的最后写操作，则在$S2$中，T_i也须是对A执行最后写操作的事务。
- 如果一个调度视等价于某串行调度，则称该调度是视可串行化的。
- 视可串行化是一种比冲突可串行化弱的可串行化条件。

例8.2

- 给定调度S： $R_1(A)$，$W_2(A)$，COMMIT(T_2)，$W_1(A)$，COMMIT(T_1)，$W_3(A)$，COMMIT(T_3)

  试分析它的可串行性。

  - 它等价于串行调度T_1, T_2, T_3（两调度中，T_1读到的A值相同，且A值最终值都由T_3写），因而是可串行调度；
  - 但这个调度显然不是冲突可串行化的，它与串行调度T_1, T_2, T_3在冲突上不等价。
  - 本例中，调度S虽不满足冲突可串行化，但却是一个视可串行化的调度。

8.1.3 用优先图识别冲突可串行化调度

❖调度S优先图构成：

- S中的每个事务对应图中的一个节点。
- 若T_i动作A先于T_j的动作B，且动作A与B存在冲突，则有一条从T_i对应节点到T_j对应节点的弧。

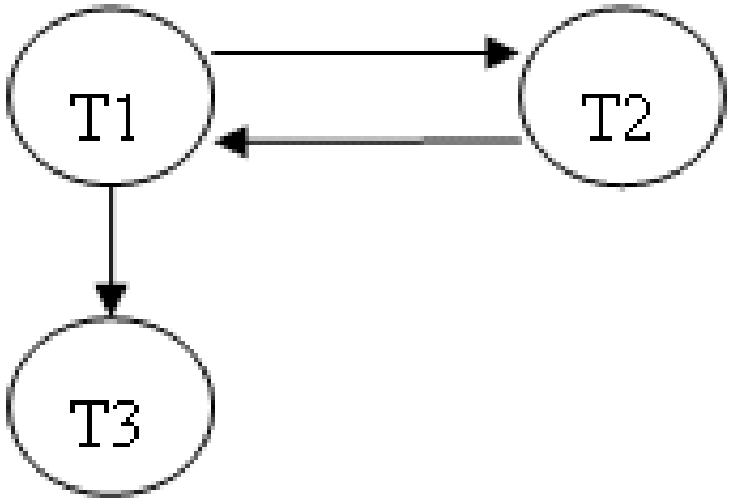


图 8.3 优先图示例

定理8.1 一个调度S是冲突可串行化的，当且仅当它的优先图是无环的。

8.2 基于封锁的并发控制

8.2.1 严格两阶段封锁协议

8.2.2 一般两阶段封锁协议

8.2.3 封锁管理

8.2 基于封锁的并发控制

- 通常，**DBMS**必须能确保：
  - 只允许可串行化和可恢复的调度，
  - 在撤销事务时，不会导致其它已提交事务动作效果丢失。
  - DBMS常基于**封锁协议**调度来实现这一目标。
    - 封锁协议：指每个事务都必须遵循的一组规则。
- 调度的一致性（~宪法）
  - 在基于封锁的调度器中，事务某DB元素X，需先获得X的锁，才可访问X；访问结束事务释放元素X锁。
  - T_i: …, T_i.lock(X); …,W-R(X);…; T_i.unlock(X)
- 调度的合法性（~法律）
  - 调度中的所有事务，都严格按封锁协议来获得DB元素锁

8.2.1 严格两阶段封锁协议 (Strict-2PL)

- 严格-2PL:指包含以下两条规则的基本封锁协议
  - 有两种元素锁：共享锁[读锁]—sl(X)；
    排它锁[读写锁]—xl(X)。
  - 事务所持有的所有锁只有在它完成时，才允许释放。
  - 完成---指commit或abort
- 若调度S的所有事务都遵循这两规则，则称S是严格-2PL调度。
- 由DBMS调度器负责自动往事务动作流中，插入获得锁和释放锁的请求。
- 一个请求锁的事务将被阻塞挂起（延迟），直到系统同意授予所请求的锁为止。

一个演示严格-2PL协议起作用的示例

T1	T2
$R_1(A)$ $W_1(A)$ $R_1(B)$ $W_1(B)$	 $R_2(A)$ $W_2(A)$ $R_2(B)$ $W_2(B)$
(a) 一个不满足一致性的调度	

Strict -2PL ⇒

T1	T2
$xl_1(A)$ $R_1(A)$ $W_1(A)$ $xl_1(B)$ $R_1(B)$ $W_1(B)$ commit	 $xl_2(A)$ | | | $R_2(A)$ $W_2(A)$ $xl_2(B)$ $R_2(B)$ $W_2(B)$ commit
(b)在 2PL 作用下被强制为串行调度	

T1	T2
$sl_1(A)$ $R_1(A)$ xl(A) | | | | $W_1(A)$ $xl_1(B)$ $R_1(B)$ $W_1(B)$ commit	 $sl_2(A)$ $R_2(A)$ $xl_2(B)$ $R_2(B)$ $W_2(B)$ commit
(c) 一个具有动作交替、但符合严格-2PL 的调度	

图 8.4　一个演示严格-2PL 协议起作用的示例

8.2.2 一般两阶段封锁协议 (2PL)

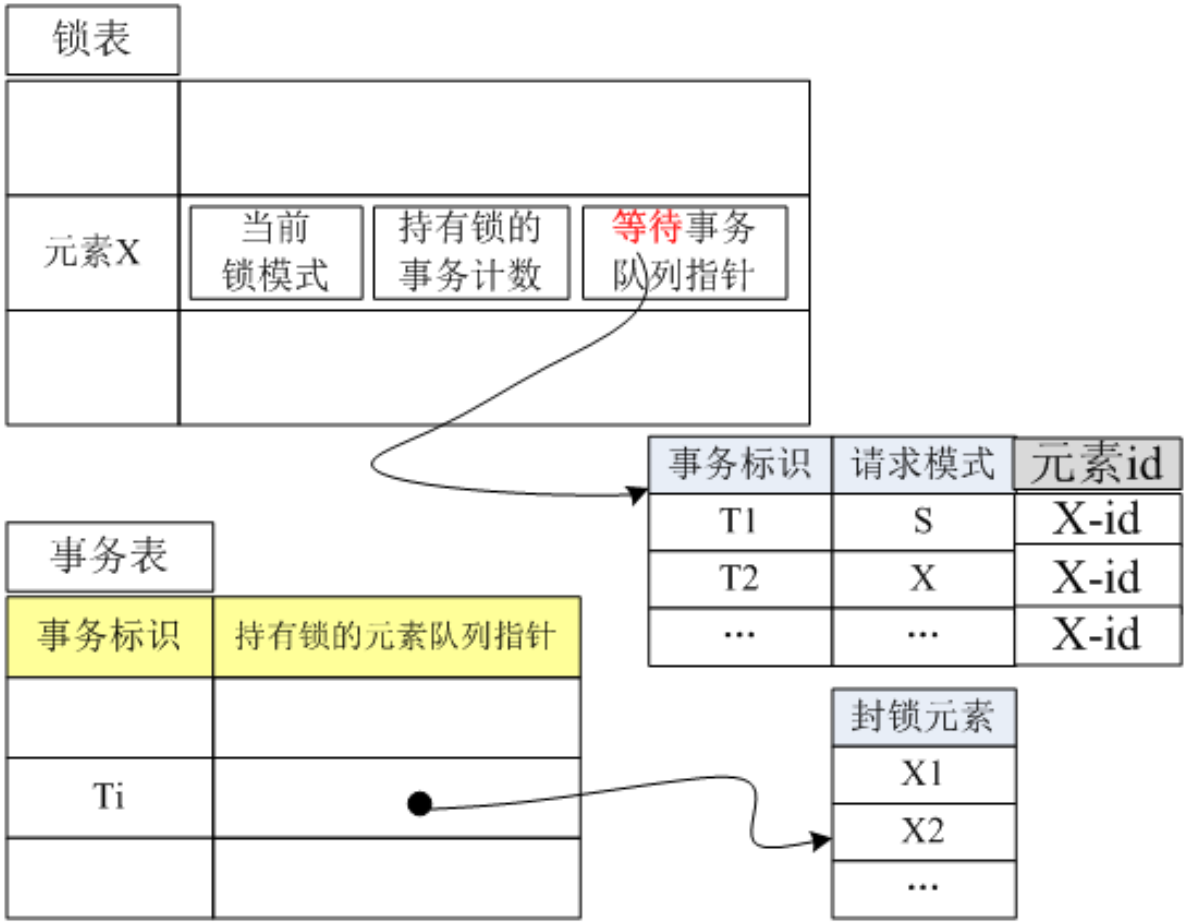
- 一般2PL是严格-2PL协议的一种变体，它放松了严格2PL的第二条规则：允许事务在结束(提交或中止)之前释放锁。
  - 2PL(规则1)：与严格-2PL相同。
  - 2PL(规则2)：当一事务已经释放了一个锁之后，就不能再允许请求额外的锁。
- 称释放第一个锁之前的阶段为“增长阶段(Growing phase)”，把之后的阶段称为“萎缩(Shrinking phase)”阶段。
- 定理：由若干一致的两阶段封锁事务构成的任意合法调度S，肯定是冲突等价的可串行调度。

strict-2PL与一般2PL对比

- 严格调度（定义）
  - 一个调度S被称为严格的，如果被一个S中事务T写的值在它提交或中止之前，不被S中其它事务读或重写。
  - 严格调度是可恢复的，且可避免级联中止。
- 严格-2PL增强了2PL，能保证每个允许的调度都是严格的，且是冲突可串行化的。
  - 这是因为当一个事务在严格-2PL下写一个元素时，它将持有排它锁直到提交或中止——不会有其它事务能‘看见’或修改这个元素的值。

8.2.3 封锁管理（数据结构及算法）

- 封锁管理器工作的基本数据结构；



- 封锁管理基本算法

8.3 死锁及其处理

8.3.1 预防死锁

8.3.2 死锁检测

8.3.3 基于封锁的并发控制性能

8.3 死锁及其处理

死锁(Deadlock)

- **定义**：在一组事务中，如果每一个都在等待其它事务占有的资源(持有的锁)，使得每个事务都无法向前推进。
- 死锁是所有事务都被无限长延迟的极端阻塞情况
  - 即使是2PL事务，也可能发生死锁

T1	T2
$sl_1(A)$	
	$sl_2(A)$
$xl_1(A)$被拒绝	
	$xl_2(A)$被拒绝

图 8.6 直接升级共享锁可能导致死锁

处理死锁的两种基本方法：

- 预防法(Deadlock Prevention)
- 检测法(Deadlock Detection)

8.3.1 预防死锁

- 根据事务启动的先后，赋予事务不同的时间戳
  - 小时间戳(更早或较老)事务，优先权较高。
- 如果一个事务T_i请求一个锁，而事务T_j持有这个锁，则封锁管理器可用如下两种策略之一来预防死锁发生：
  - 等待-死亡(Wait-die) (弱势等待者)
    - 如果事务T_i有较高优先级，则允许它等待；
    - 否则中止T_i（自死亡）；
  - 伤害-等待(Wound-wait) (强势等待者)
    - 如果T_i有较高的优先级，则会中止(伤害)T_j；
    - 否则T_i等待；

附加策略：“被中止而重新启动事务，保持原有的时间戳”，可保证每个事务最终变为足够老，成为具有高优先权的事务。

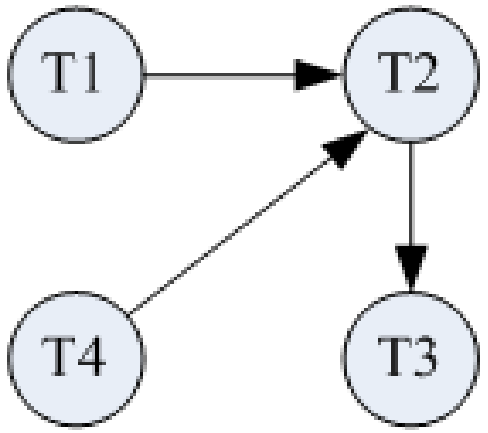
8.3.2 基于等待图的死锁检测

- 等待图的拓扑结构：
  - 每个活跃事务对应图中的一个节点；
  - 如果事务T_i正等待事务T_j所持有的某个锁，则有一条从T_i对应节点指向T_j对应节点的边。
- 它可清晰表达事务等待其它事务持有锁的情况。
- 封锁管理器通过维护等待图来检测死锁循环。

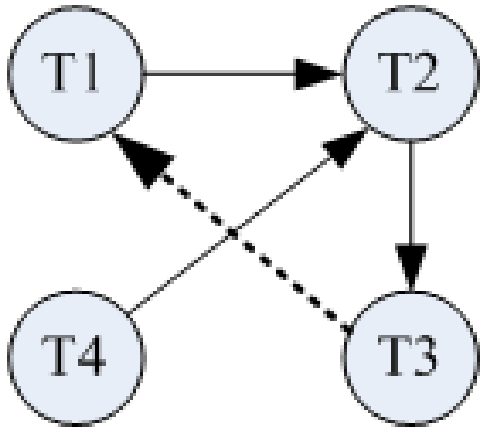
一个会导致死锁的调度及其等待图(图8.7)

*T*1	*T*2	*T*3	*T*4
sl(A)			
R(A)			
	xl(B)		
	W(B)		
sl(B)			
被拒		sl(C)	
		R(C)	
	xl(C)		
	被拒		xl(B)
		xl(A)	被拒
		被拒	

(a)一个会导致死锁的调度



(b) 死锁发生之前的等待图



(c) 死锁发生之后的等待图

8.3.3 基于封锁的并发控制性能

- 采用“阻塞(blocking)”和“中止(aborting)”两种机制来解决并发事务间的动作冲突。
- 两种处理机制都有性能惩罚因素
  - 阻塞事务可能因持有某些锁，而致更多事务阻塞等待
  - 中止和重启动，则会浪费事务已完成的工作。
- **DBMS**中，要设计一个良好的、基于封锁的并发控制机制，必须在以下方面作出选择：
  - 是使用死锁预防还是死锁-检测机制？
  - 如果使用死锁-检测机制，则多长时间间隔检测一次？
  - 检测到了一个死锁，应中止等待环中的那个事务？

检测方法与预防方法比较

- 实现难易程度
  - 预防法更易实现。当DB为分布式时，构造等待图必须通过各个节点上的锁表集合来构造。
- 在死锁发生不频繁的场合，基于检测死锁方案通常效果更好、更为实用。
  - 等待图能使由于死锁而必须中止事务的次数最少。除非真的存在死锁，否则系统决不会中止事务。而预防法常会在并不存在死锁的情况下中止事务。
  - 在封锁竞争程度很大的场合，死锁的生命期会变得很长，这时预防法效果可能会更好些。

8.4 扩展封锁处理技术

- 前面已介绍的各种封锁协议，都是将DB抽象为一组相互独立的、DB元素固定的集合。
- 本节我们将放宽这个限制，并讨论相关扩展处理方法。
- 方法:通过进一步细化封锁协议,以获得更好性能

针对元素集
大小可变扩展

讨论：
著名的幻象(phantom)问题

针对元素间
非独立扩展1

讨论：
树形结构对象的封锁问题

针对元素间
非独立扩展2

讨论：
具有嵌套包含关系元素组的多粒度封锁问题

8.4 扩展封锁处理技术

8.4.1 动态数据库与幻象问题

8.4.2 B+树的并发控制

8.4.3 多粒度封锁

8.4.1 动态数据库与幻象问题

实例分析：

- 事务*T1*扫描Sailors关系，
  - ①检索职级$rating$ =1的最大年龄水手age=71。
  - ④检索职级$rating$ =2的最大年龄水手age=63。
- *事务T2*
  - ②插入一个$rating$=1, age=96的新元组
  - ③删除$rating$ =2, age=80的最大年龄水手
- 讨论不同的动作执行顺序对结果的影响
- 如果事务封锁了一组DB元素后，其它事务还能导致组元素增加(enlarging)，则冲突可串行化并不能真正保证可串行化。

处理幻象问题的方法—扩大封锁

- 封锁整个关系文件
  - 如果没有索引且文件所有页必须被扫描，则必须封锁整个关系文件。
- 采用谓词封锁技术
  - 如果有特定字段上的稠密索引，则直接封锁指定键值范围的索引项。
  - 索引封锁实际上是谓词封锁中的一种特殊情形。谓词封锁是一种代价相对昂贵的封锁方法，实际系统中并不常用。

8.4.2 B+树的并发控制

- 将B树索引当作一般文件，以页为单位，采用2PL协议进行并发控制管理，将会导致很强的封锁竞争。
  - 访问B+树，都要过根节点
- 已有更有效的封锁协议，如树协议，能充分利用B+树的层次结构特征
  - 在确保可串行化和可恢复性的同时，有效提高系统的并发性能。

针对B+树结构特点的两点观察

- 树的内层只是用来帮助搜索，只有叶节点上才存储的数据记录或记录指针；
- 对搜索操作
  - 必须获得沿路径（从根至叶）的各节点共享锁；此外，B+树搜索从不需要回溯。
  - 这个观察说明 “**一旦达到并封锁了某节点的子节点，则该节点上的锁即可释放**”。
- 对插入操作
  - 如果因叶节点修改分裂，可能传播到的上层节点，必须以排它模式封锁。

一个用来分析树封锁方式的3阶B+树

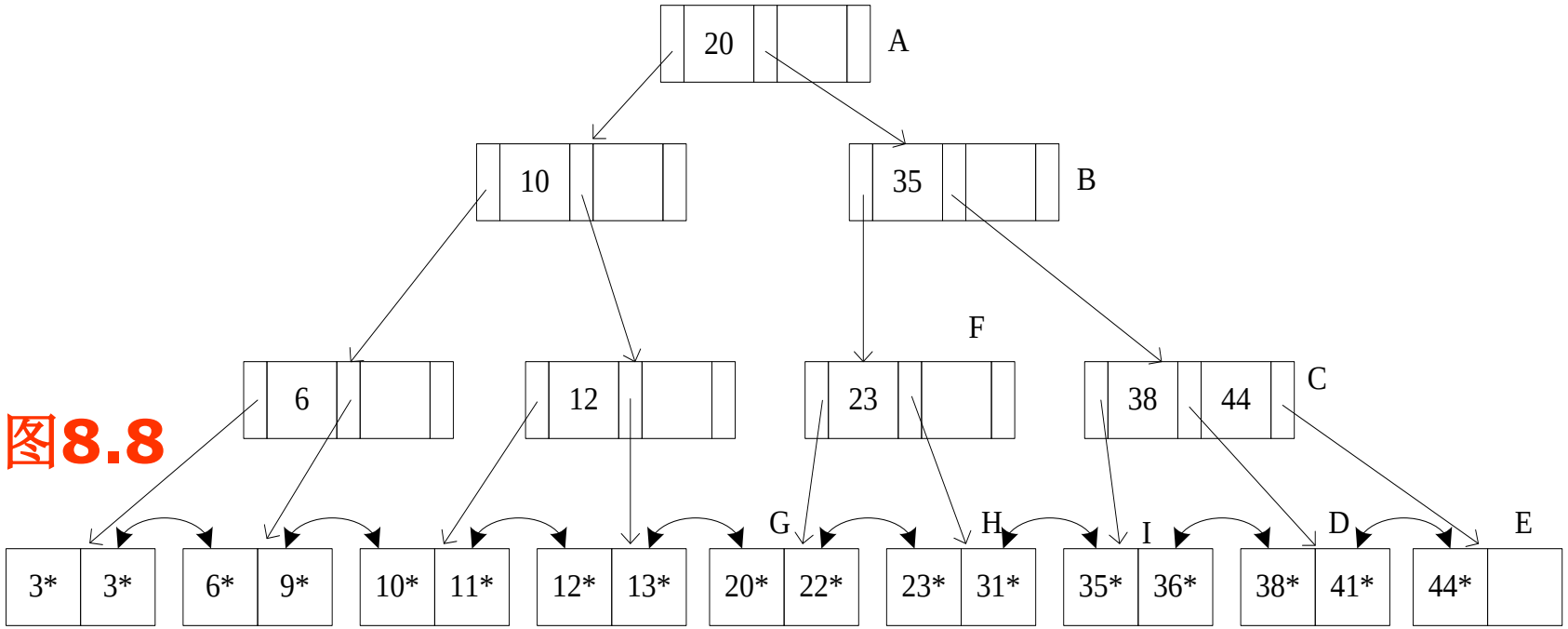


图8.8

再分析插入25*:事务最终会获得*F*上的S锁，*H*上的X锁。不幸的是，*H*是满的，插入新项后须分裂。分裂*H*，需修改*F*；但现在F上只持有S锁，？

树协议（一种2PL变体规则）

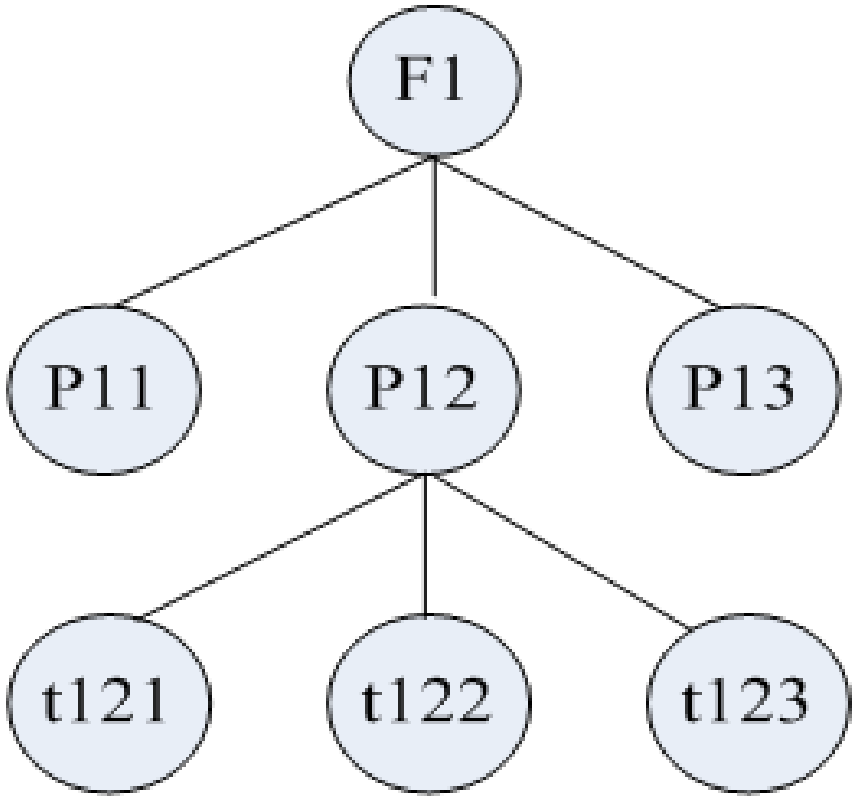
- 读模式访问某树节点，必须先获得该节点的***S***锁（共享锁）；读/写模式访问某节点，必须先获得该节点的***X***锁（排它锁）；但没有两阶段封锁限制。
- 对于树中的一个节点，事务只有在其上持有锁时，才能获得其后续子节点的锁。
- 事务的第**1**个锁可以在树的任何节点上。
- 节点可在任何时候解锁，无两阶段封锁限制。
- 事务不能对一个已经解锁的节点重新上锁，即使它在其父节点上仍持有锁。

树协议起作用的原因分析

- 调度S中事务T_i和T_j封锁一个共同的结点，而T_i先封锁该结点，则称T_i先于T_j（简记为$T_i<T_j$）。
- 对任意两个事务T_i和T_j，如果T_i先于T_j封锁根，那么T_i在T_j前封锁两者都共同封锁的结点(相当于强制一个串行调度)。
- 本质上看，可认为树协议在调度涉及的事务上强制实现一个串行顺序；
  - 可证明：满足树协议的调度优先图总是无环的。

8.4.3 多粒度封锁

- 不同DB元素，可能有不同粒度，且相互间可能还存在包含性的层次结构。
- 对某节点的锁，隐含了对该节点所有子节点的锁。



多粒度封锁时，需要考虑的一些特别因素

- 当一个事务已获得某层次节点锁时，必须能防止该节点的上层大粒度元素不会被其它事务以某种有冲突的方式封锁。
- 尽可能将较严格的封锁（X锁）局限在较小粒度元素上，这样会更有利于提高系统的并发性能。
- 如果必须封锁较大粒度元素（会隐含封锁下层所有子元素），应尽可能使用宽松的模式进行封锁。
- 多粒度封锁正是基于以上考虑而提出的一种封锁协议。

多粒度封锁协议

- 除了S锁、X锁
  在上层大粒度
  向锁：
  - 意向共享锁（
  - 意向排它锁（
  - 在包含它的上
    防止其它事务

○ 相容　　× 冲突

	IS	IX	S	X
IS	○	○	○	×
IX	○	○	×	×
S	○	×	○	×
X	×	×	×	×

- **IS与IX之间没有冲突，一个节点可以同时具有IS和IX锁。IS只与X锁冲突；IX则与S、X冲突。**

8.5 基于优化的并发控制

8.5.1 基于有效确认的并发控制

8.5.2 基于时间戳的并发控制

8.5.3 三种并发机制比较

基于封锁的并发控制

封锁协议采用了一种悲观保守的（**pessimistic**）方法来解决事务间冲突----使用中止或延迟事务封锁请求的方式来解决冲突。.

基于优化的并发控制

采用了一种乐观式(**optimistic**)假设性前提：认为绝大部分事务相互之间并不会有冲突。在此基本前提下，采用“先放任事务执行，出现问题后再处理”的思想.

8.5.1 基于有效确认的并发控制

- 有效确认是一种基于优化的并发控制，允许事务不经过封锁直接访问数据，并在“适当的时候”检查事务是否以可串行化的方式运转。
- 这个“适当时候”主要指事务开始写DB对象之前的、一个称被为“有效确认”的、很短的瞬间阶段。
- 基于有效确认的调度器数据结构——五个集合
  - 对每个事务：维护其读对象集$RS(T_i)$、写元素集$WT(T_i)$
  - 三个全局事务集：
    - 处于读阶段的事务集*START*；
    - 处于写阶段(已确认但写尚未完成)的事务集*VAL*；
    - 三个阶段都已完成的事务集*FIN*；

基于效确认规则的算法描述

- //判定T_j能否有效确认算法

validated =true ;

For each Ti ∈ { T_j之前已确认但仍活跃事务 }

if （T_j, T_i） 满足以下两个条件之一:

1） T_i 三个阶段已经完成

且 $RT(T_j) \cap WT(T_i)=\Phi$;

2） T_i已有效确认但尚未完成

且 $RT(T_j) \cap WT(T_i)=\Phi \wedge WT(T_j) \cap WT(T_i)=\Phi$ 则

//认为T_j不影响T_i的有效确认

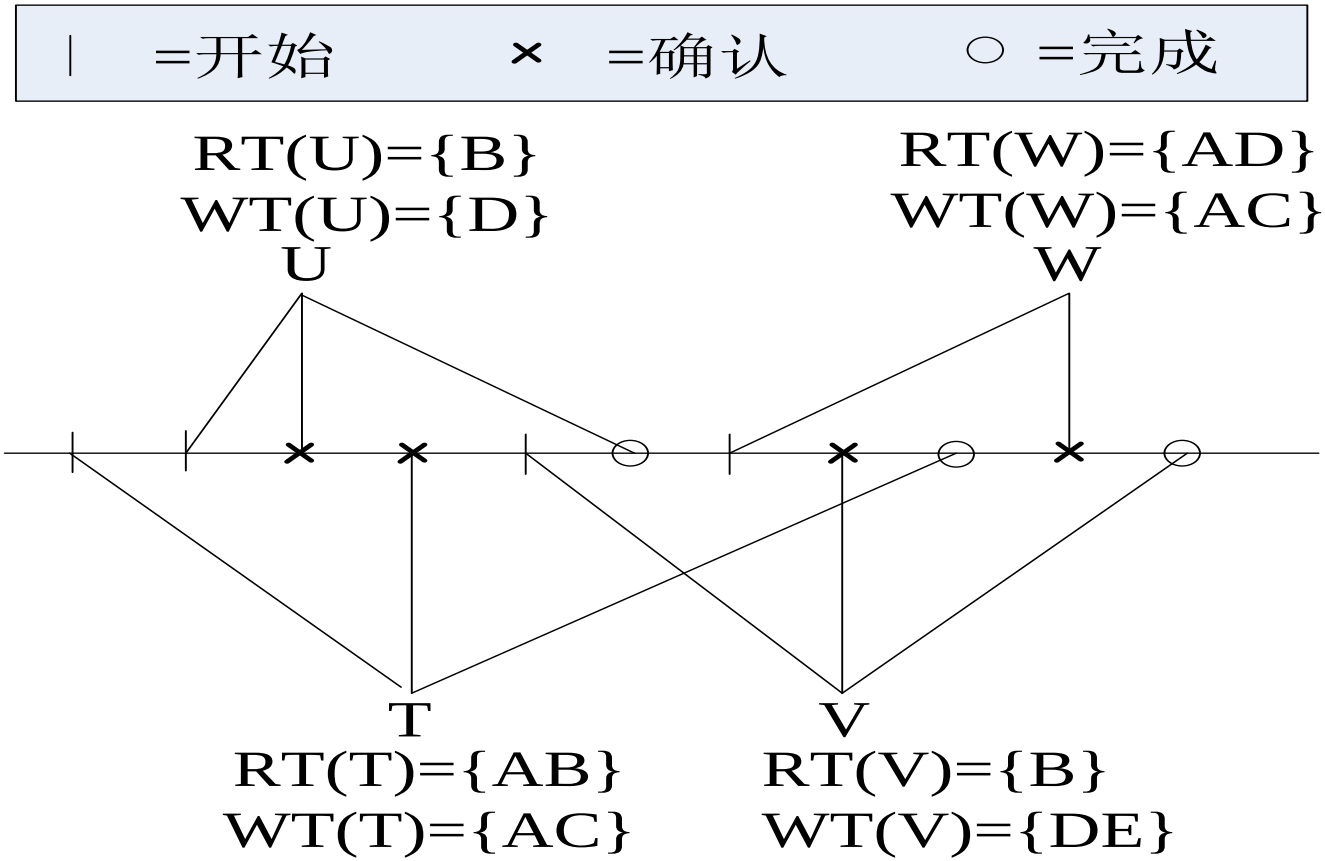
continue;

else {validated =false; break;}

End for

应用举例----例8.4

- 图8.11中给出了四个事务*T*、*U*、*V*、*W*试图执行并确认的时间点，每个事务的读写集已在图中标明。试分析它们是否能有效确认。



8.5.2 基于时间戳的并发控制

- 属乐观式的并发控制，事务执行被隐含了一个基于时间戳的顺序：
  - 执行时检查所有冲突动作是否能遵循该顺序
- 相关数据结构
  - 时间戳及其产生方法

需要标记时间戳的事件和动作：

- 对每个事务T_i，应维护一个启动时的时间戳TS(T_i)。
- 对每个DB对象O，应维护：
  - 一个最近被读的时间戳RT(O)；
  - 一个最近被写的时间戳WT(O)；
  - 一个最近被写的提交位C(O)。若最近写事务提交，则C(O)=true；否则C(O)=false。

基于时间戳的并发控制原理

- 事务执行被隐含施加了一个基于时间戳的顺序，

如果有某个动作违反了这个串行顺序原则，则相关事务就必须被中止撤销。

- 这个原则性要求的具体算法描述：
  - 对∀<T_i,T_j>，

    if $T_i \rightarrow T_j$,即TS(T_i)<TS(T_j) then

    //须确保在执行期间

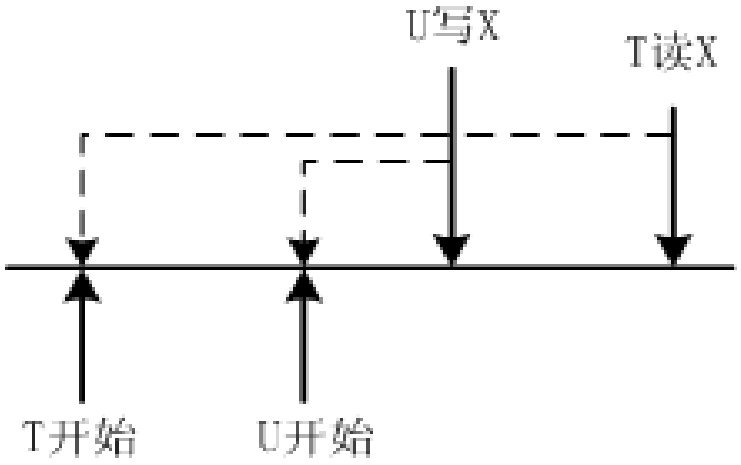
    for each < a_i,a_j > {T_i/T_j冲突动作对}

    总有$a_i \rightarrow a_j$成立;

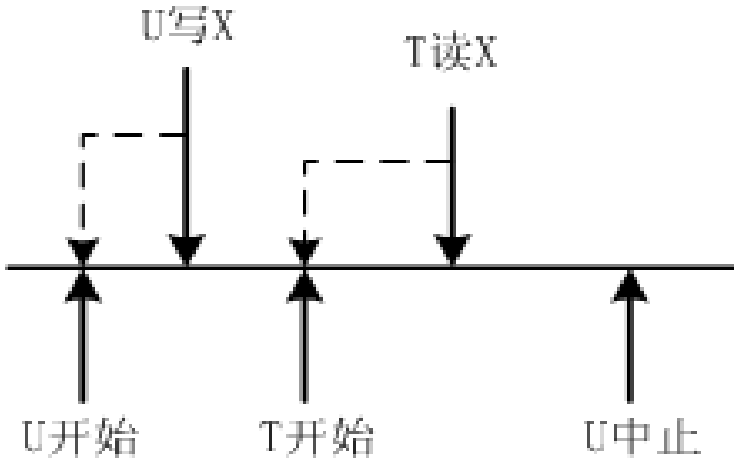
    end for;

    end if

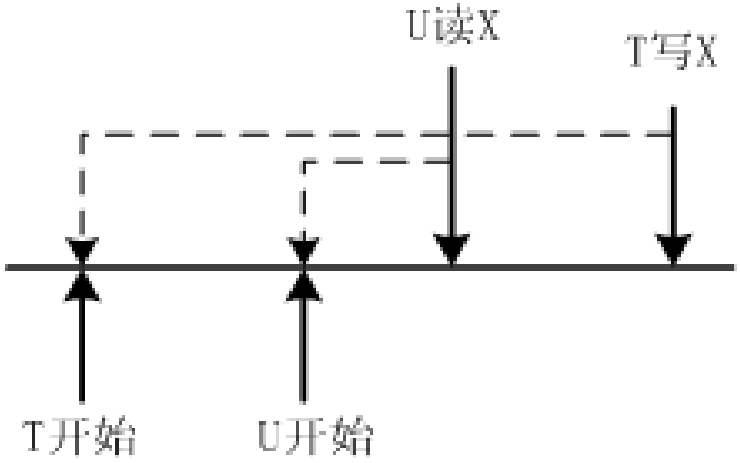
违反了这个串行原则的几种典型情况(图8.12)



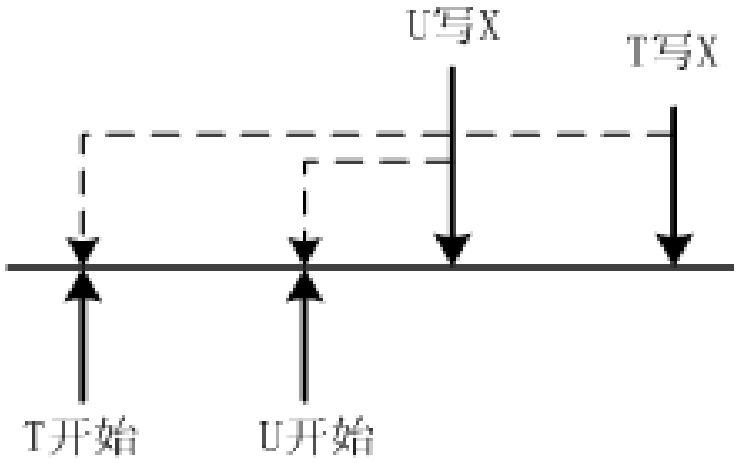
（a）事务T试图做过晚读



（b）事务T试图做脏读



（c）事务T试图做过晚写



（d）事务T试图做过时写

算法8.3 响应读请求的处理算法　　//T请求读DB对象O时调用

```
begin
    if  TS(T)<WT(O) then //以“过晚读”模式，违反了基于时间戳的串行顺序
        事务 T 被中止，并以一个更大、更新的时间戳重新启动；
    else
        if C(O)=false then
            事务 T 阻塞等待，直到 C(O)变为 true;
        end if
        事务 T 开始读对象 O；     //最近的写已提交
        RT(O)= max{ TS(T), RT(O) }; //更新对象 O 的读时间戳
    end if
end begin
```

算法8.4 响应写请求的处理算法　　//T请求写DB对象O时调用

```
begin
    if   TS(T)<RT(O) then //  “过晚写”
        事务 T 被中止，并以一个更大、更新的时间戳重新启动；
    else if TS(T)<WT(O) then   //  “过时写”
        忽略事务 T 对元素 O 的写动作,并继续执行; //用“Thomas 写规则”
    else
        事务 T 写 DB 对象 O;
        WT(O)= max{ TS(T), WT(O) }; //更新对象 O 的写时间戳;
    end if
End;
```

基于时间戳的算法应用举例

T_1	T_2	T_3	A	B	C
TS=200	TS=150	TS=175	RT=0; WT=0	RT=0; WT=0	RT=0; WT=0
R1(B) W1(B) W1(A)	R2(A)	R3(C)	RT=150 WT=200	RT=200 WT=200	RT=175
	W2(C) 过晚写,T2 中止				
		W3(A) 被忽略 (Thomas 写)			

多版本时间戳并发控制

- 基本目标：希望事务只读DB元素时不需要等待。
- 基本做法：是维护最近被修改对象的多个版本(每个版本都带有一个写时间戳)，当读事务*T*到来时，让它读*TS*(*T*)之前的最近版本。
  - 单版本时，会因“过晚读”而回滚的读事务，在多版本下，总能读到适合它的版本值（符合“串行要求”），从而避免回滚。
  - 如果DB元素是页，这种方法特别有用，因为这时所需做的只是让缓冲区管理器保留一些对当前活跃事务可能有用的页。

多版本时间戳并发控制

- 相比于单版本，多版本时间戳调度器，需要在以下几个方面进行增强管理：
  - 当新的写请求$WT(X)$合法时，DB对象的一个新版本被创建，其写时间为$t=TS(T)$；写时间与对象的版本有关，且永远不改变。
  - 当新的读请求$RT(X)$到来时，调度器找到不违反“过晚读”的最新对象版本。
  - 当某版本的写时间t小于任何活跃事务的启动时间时，就可以删除比t更早的所有其它版本。

三种并发机制比较

表 8.2 三种并发控制方法需要的额外存储空间代价对比

并发控制方式	需要的辅助存储空间情况
封锁	需要增加锁表存储空间，锁表空间与被封锁的 DB 对象个数成正比。
时间戳	简单实现时：每个 DB 对象要维护读/写时间戳和提交位，不管是否被写过。这可能需要很大的额外存储空间。 更精细实现：将最早活跃事务以前的所有时间戳统一按 0 或-∞处理而不记录它们，并按类似锁表那样，将最近被访问过的相关对象时间戳记录在一张专门的表中。
有效确认	对当前活跃事务以及少量几个已提交的更早事务：记录每个事务启动/确认/完成时间戳，以及每个事务的读写元素集。 该方法的一个潜在问题是，事务的写集合必须在写发生之前知道（这可能需要先扫描一次事务的所有动作）。

不推迟事务执行的性能比较

- 封锁法一般只推迟事务没有回滚事务；但为处理死锁，偶尔也会有回滚事务的情况。
- 优化法不推迟事务，但会导致事务回滚。回滚是更严重的推迟形式，会造成较大的资源和已完成工作浪费。
- 在高冲突的竞用情况下，优化方法的回滚会很频繁，这时封锁法往往性能更优。
- 当回滚不可避免时，时间戳比有效确认能更早捕获到潜在问题，并且在考虑一个事务是否必须回滚前，允许让其完成所有的内部工作。——回滚造成的工作浪费会少些。

8.6 SQL-92*的事务支持

8.6.1 事务的基本特征

8.6.2 事务与约束

8.6.1 SQL-92*的事务基本特征

- 诊断大小(diagnose size)
  - 用于决定可能被记录的最大错误条件号，这里不讨论
- 存取模式
  - 包括只读(read only)和读写(read write)两种模式。如果是只读模式，则事务不允许修改DB。
  - 在只读模式下执行的事务，只需要获得共享锁，有利于增强系统的并发性能。
- 孤立级
  - 是控制事务内部操作暴露给其它并发执行事务程度的特征参数，有四种设置选项：
    - 未提交读 (READ UNCOMMITTED)
    - 提交读 (READ COMMITED)
    - 可重复读 (REPEATABLE READ)
    - 可串行化 (SERIALIZABLE)

四种孤立级对事物操作的影响

表 8.3 四种孤立级对事务操作的影响

隐含风险 / 孤立级	脏读问题	不可重复读问题	幻象问题
未提交读	可能有	可能有	可能有
提交读	无	可能有	可能有
可重复读	无	无	可能有
可串行化	无	无	无

8.6.2 事务与约束

- 完整性约束：表示**DB**状态必须满足的条件。通过**SQL**的**DDL**语句，允许用户在定义**DB**对象时，附加给出有关**DB**对象的完整性约束构造。
  - 在缺省情况下，在DBMS系统在执行每个可能违反约束的SQL语句时检查，拒绝违反语句。
- **SQL-92**允许推迟约束检查以提高应用灵活性。
  - 设置约束检查时机的两个选项：DEFERRED和IMMEDIATE。有关设置命令为：

    SET CONSTRAINT ConstraintFoo

    DEFERRED|IMMEDIATE.

- “可串行化”是SQL建议大部分事务使用的、最安全的一种孤立级。然而，一些事务允许运行在较低的孤立级，这样需要请求的锁数量可更少，有利于改善系统性能。
  - 例如，当一个关系中元组数很多时，计算该关系中某数值型字段均值的统计查询，就可用提交读或未提交读来运行。因为元组数很大时，少数几个不正确的或遗留的值对结果影响并不会很大。
- 在基于封锁的实现时，可通过遵循严格-2PL协议规则来获得可串行事务。

- 孤立级和存取模式可通过**SET TRANSACTION**命令来设置。
  - 例如，下面的命令声明当前事务以可串行化和只读模式运行。

    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY.
- 当一个事务启动时，缺省设置是：**SERIALIZABLE READ WRITE**。

T1.SLOCK(已有的rating=1所有页);

==>max_age1=71;

T2.XLOCK(新页).insert(rating=1;age=96);

T2.XLOCK(已有的rating=1所有页)

T2.delete(rating=2的age最大元祖-- age=96);

T2.commit;

T1.SLOCK(已有的rating=2所有页);

==>max_age2=63;

T1.commit;