

# 第7章 事务管理

2012.04



# 目录 Contents

## ■ 7.1 数据库恢复

- 恢复的基本技术
- 日志结构与机制
- 更新事务的执行与恢复
- 各类失效的具体恢复对策

## ■ 7.2 并发控制

- 并发控制概述
- 加锁协议
- 多粒度封锁与意向锁
- 死锁的检测、处理和预防



- 事务是DBMS中的执行单位，事务能（也只有事务能）将数据库从一种一致状态变为另一种一致状态，从而保证数据库中的数据始终是完整的、正确的。这样的数据库才是一种信息资源。
- 事务为何能这样呢？
  - 因为事务满足ACID准则。
- 保证事务始终满足ACID准则的一系列技术措施称事务管理(Transaction Management)，包括两个方面
  - 当系统发生故障时的技术措施，称数据库恢复(Database Recovery)。
  - 当多个事务并发执行时的技术措施，称并发控制(Concurrency Control)。



# 目录 Contents

## ■ 7.1 数据库恢复

- 恢复的基本技术
- 日志结构与机制
- 更新事务的执行与恢复
- 各类失效的具体恢复对策

## ■ 7.2 并发控制

- 并发控制概述
- 加锁协议
- 多粒度封锁与意向锁
- 死锁的检测、处理和预防



# 7.1.1 恢复的基本技术

- 数据库系统同其它任何（计算机）系统一样，不可能一直正常运作，总可能在某个时候发生故障(Fault)而导致数据库失效(Failure)，这种情况一旦发生，就难于保证事务满足ACID准则，而对用户来说，数据变得不可靠了、甚至丢失了，这不能不说是一个巨大损失。
- 故障是不可避免的
  - 计算机硬件故障
  - 系统软件和应用软件的错误
  - 操作员的失误
  - 恶意的破坏
- 故障的影响
  - 运行事务非正常中断
  - 破坏数据库



# 7.1.1 恢复的基本技术

- 数据库管理系统对故障的对策
  - DBMS提供恢复子系统
  - 保证故障发生后，能把数据库中的数据从错误状态恢复到某种逻辑一致的状态
  - 保证事务ACID
- 恢复技术是衡量DBMS优劣的重要指标



# 7.1.1 恢复的基本技术—三类故障

## ■ 三类故障

- **事务失效(Transaction Failure)**: 指事务因不可预知的原因而夭折, 这些原因可能是:
  - 事务因运行时出错 (Run-time Error) 而无法继续执行;
  - 用户突然要求撤销事务;
  - 系统调度时强行中止事务的运行, etc。
  - **特征: 发生在事务提交完成前。**
- **系统失效(System Failure)**: 指掉电或系统(OS或DBMS)发生故障而导致数据库系统无法继续正常运行下去, 此时必须重新启动(Restart), 从而导致一切事务无条件中止运行。
  - **特征: 内存数据全部丢失, 但外存上的数据库未遭破坏。**



# 7.1.1 恢复的基本技术—三类故障

## ■ 三类故障(cont.)

- 介质失效(Media Failure): 指数据库存储介质(通常是磁盘)发生故障而导致不可读/写盘或盘中数据丢失。
  - 特征: 外存上的数据库已遭破坏, 一切已提交的事务对数据库的影响全部丢失。
  - 介质故障比前两类故障的可能性小得多, 但破坏性大得多。

## ■ 当发生以上故障后, 怎么办?

- 如果平常没有保障措施的话, 那将是束手无策的! 当数据库发生以上故障时, 可通过一定的技术措施将数据库恢复到发生故障前的、最近的一致状态——称这样的技术措施为数据库恢复。





# 7.1.1 恢复的基本技术—三种恢复技术

## ■ 恢复操作的基本原理

- 利用数据冗余原理，将数据库中的数据在不同的存储介质上进行冗余存储，当数据库本身受到破坏时，可以利用这些冗余数据来重建数据库中已被破坏或不正确的那部分数据，进行恢复。

## ■ 恢复的实现技术

- 复杂：一个大型数据库产品，恢复子系统的代码要占全部代码的10%以上。

## ■ 恢复机制涉及的关键问题

- 如何建立冗余数据
- 如何利用这些冗余数据实施数据库恢复

## ■ 常用技术

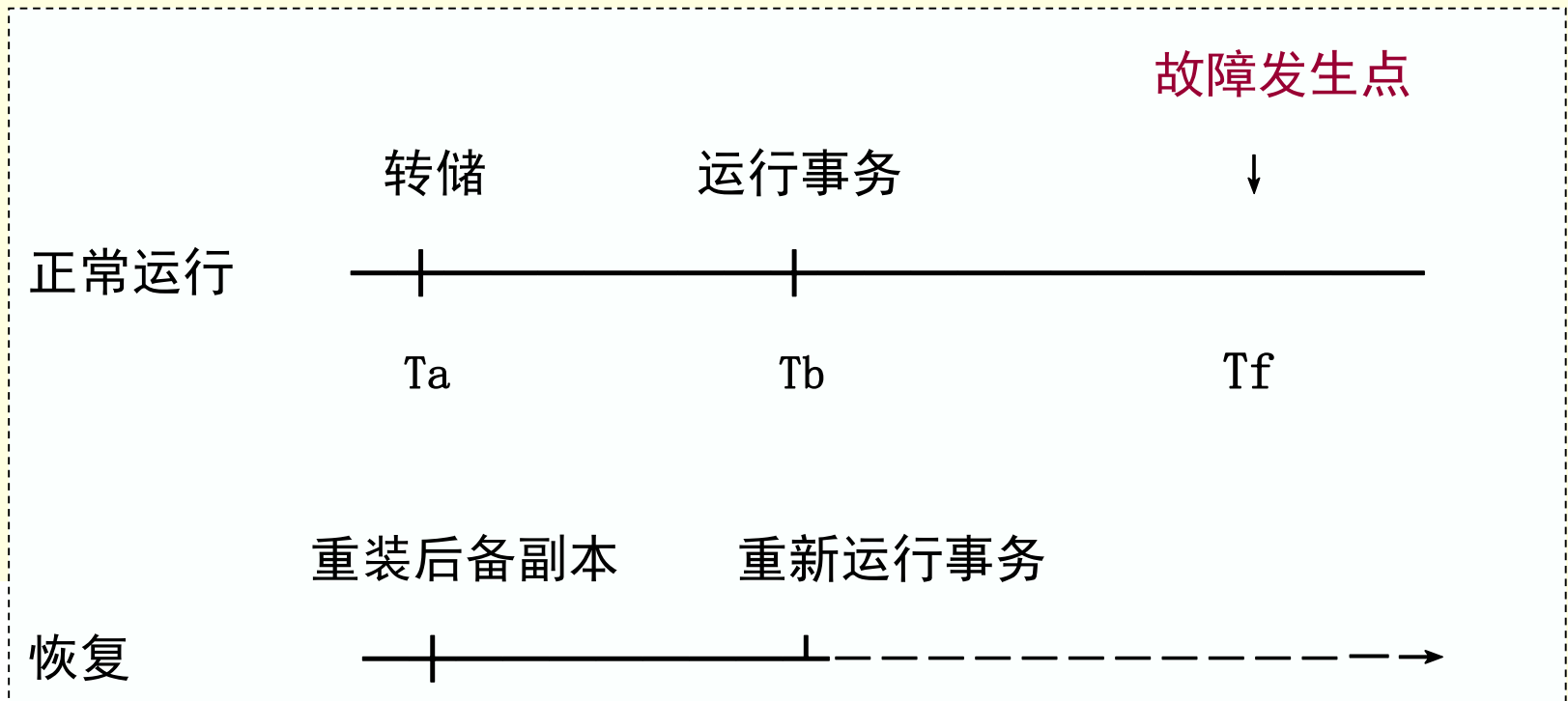
- 数据后备复本 (backup)
- 运行记录/日志 (logging)
- 多复本



# 7.1.1 恢复的基本技术—三种恢复技术

## ■ 单纯以后备复本为基础的恢复技术

- 后备复本 (Backup): 周期性地把 (磁盘上的) 数据库物理文件复制 (称转储 Dumping) 到磁带上, 建立副本。
- 方法: 一旦发生故障, 利用后备复本重装数据库, 从而达到恢复目的。



## 7.1.1 恢复的基本技术—三种恢复技术

### ■ 单纯以后备复本为基础的恢复技术(cont.)

- **注：**做后备时，数据库须保持一致状态。通常需暂停系统的运行。如果数据量很大，后备工作量很大。因此，后备周期不可能太频繁。但由于“后备”只能将数据库恢复到后备时的一致状态，从最近后备到发生故障之间的数据库更新将全部丢失。
- 为弥补不足产生了**增量转储(Incremental Dumping, ID)**(原先的后备技术相对称**海量转储**)，以减少更新丢失量。



## 7.1.1 恢复的基本技术—三种恢复技术

### ■ 单纯以后备复本为基础的恢复技术(cont.)

#### ■ 特点：

- 实现技术简单。
- 不会持久影响数据库系统的运行性能，但备份与重装时工作量很大！
- 不能将数据库恢复到最近一致状态（会丢失更新）。
- 注：备份要消耗系统资源，尤其是联机备份(Online Backuping)；脱机备份(Off-line Backuping)只是在实施备份时暂停系统的运行，其它时间不影响系统。

- 结论：这是文件系统那里继承来的恢复技术，(大型)数据库系统一般不用此技术。



## 7.1.1 恢复的基本技术—三种恢复技术

### ■ 以后备复本和运行记录为基础的恢复技术

- 运行记录/日志(Log): 从数据库建立并开始运行起, 记录全部提交事务对数据库的所有更新操作的文件, 包括以下内容:
  - 事务及其状态:
    - 事务标识 (TID)
    - 提交了(commit) / 回滚了(rollback)?
  - 前像 & 后像
    - 前像(Before Image, BI): 每次更新时, 数据块的旧值。(插入时, BI为空)。
    - 后像(After Image, AI): 每次更新时, 数据块的新值。(删除时, AI为空)。



## 7.1.1 恢复的基本技术—三种恢复技术

### ■ 以后备复本和运行记录为基础的恢复技术(cont.)

#### ■ 方法：

- 有了BI，如果需要，可使DB恢复到更新前的状态（可撤销更新）——称撤销(Undo)；
- 有了AI，如果需要，可使DB恢复到更新后的状态（可重做更新）——称重做(Redo)。
- 因此，有了日志，当发生故障时，可通过以下方法完全恢复DB：
  - 若数据库未遭破坏，则从最近一致状态开始，对未提交事务进行Undo—向后恢复(Backward Recovery)；对已提交事务进行Redo—向前恢复(forward Recovery)。
  - 若数据库遭破坏，则先重装最近的Backup，再对Backup以来的所有已提交事务进行Redo。



## 7.1.1 恢复的基本技术—三种恢复技术

### ■ 以后备复本和运行记录为基础的恢复技术(cont.)

#### ■ 特点:

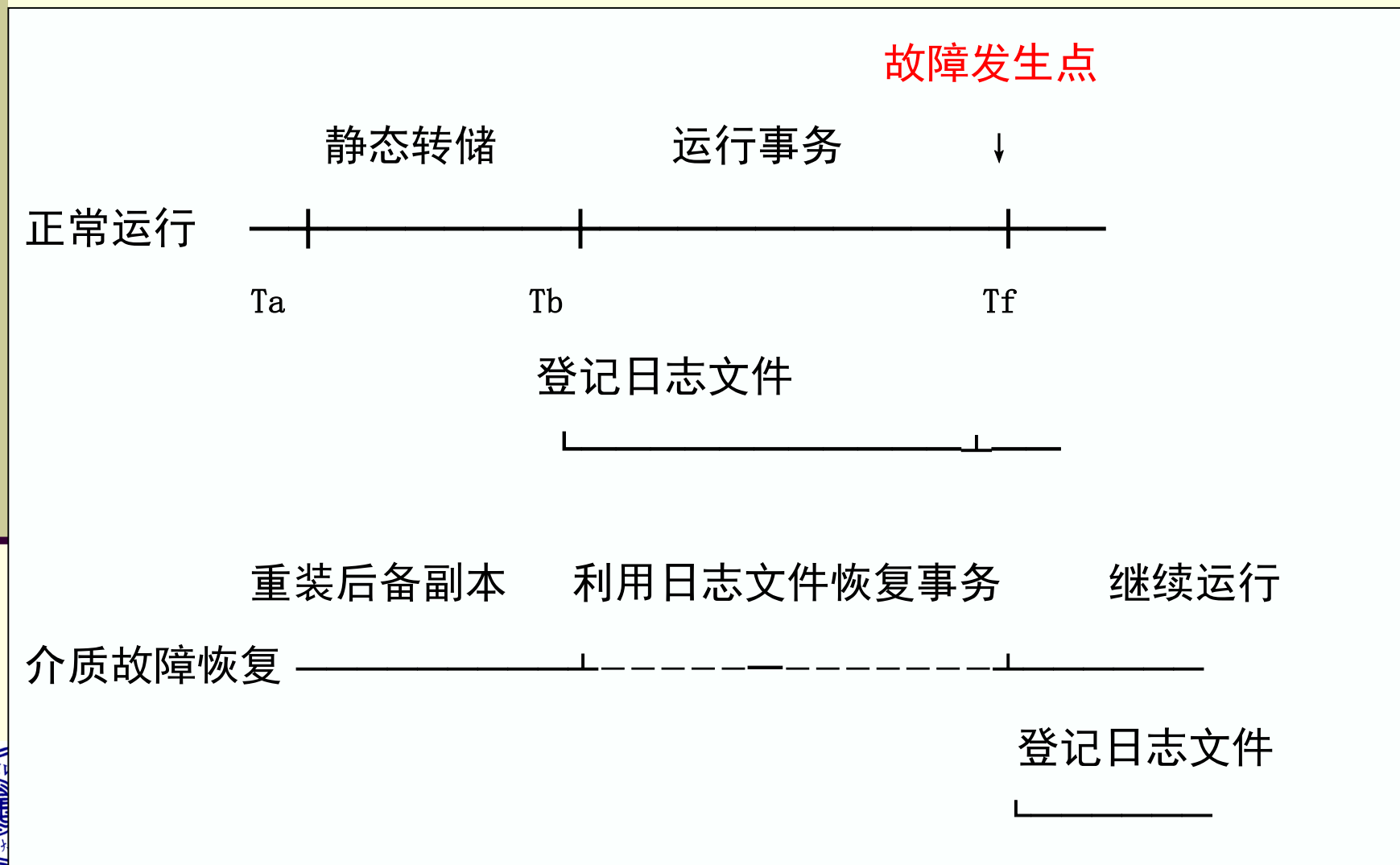
- 实现技术较复杂。
- 建立日志的过程持久地影响数据库系统的运行性能，也花费较大的存储空间。
- 能将数据库恢复到最近一致状态（从不丢失更新）。

- 结论：Backup+Log是最典型/常用的技术，绝大多数商品化DBMS均采用这种技术。



## 7.1.1 恢复的基本技术—三种恢复技术

### ■ 以后备复本和运行记录为基础的恢复技术(cont.)





# 7.1.1 恢复的基本技术—三种恢复技术

## ■ 基于多复本的恢复技术

- 独立失效模式(Independent failure Mode): 不致因同一故障而一起失效（因为支持环境是独立的）。
- 方法: 系统中保持多个具有独立失效模式的数据库复本, 互为备份、互为恢复的依据。
  - 例如: 镜像 (Mirroring) 技术, Mirrored Disks / Mirrored Files  
“同时写, 任选读” —对性能影响不大, 有时可能反而提高性能。



# 7.1.1 恢复的基本技术—三种恢复技术

## ■ 基于多复本的恢复技术(cont.)

### ■ 特点：

- 开销较大（用户要购置双倍/多倍硬件）。
- 恢复基本上由系统自动进行；总能保持DB的一致状态。
- 可能冒这样的（不太可能发生，但还是有可能发生）危险：“全军覆没” → “无可救药”。

### ■ 结论：

- 绝大多数商品化DBMS不支持这种技术，如Oracle不支持镜像文件功能（控制文件及日志文件例外）；
- 较适合运用于分布式数据库环境：如：DB Server 双工 / 镜像。



# 目录 Contents

- 7.1 数据库恢复
  - 恢复的基本技术
  - 日志结构与机制
  - 更新事务的执行与恢复
  - 各类失效的具体恢复对策
- 7.2 并发控制
  - 并发控制概述
  - 加锁协议
  - 多粒度封锁与意向锁
  - 死锁的检测、处理和预防



## 7.1.2 日志结构与机制

### ■ 日志存储的基本内容

- 活动事务表(ATL): 记录还在执行但尚未提交的事务标识(TID)。
- 提交事务表(CTL): 记录已提交事务的标识。
  - 当一个事务被提交结束时, 做: ①TID→CTL; ②Remove TID from ATL
- 前像文件: 记录所有被更新的数据块之旧值BI, 用于进行Undo操作。
  - Undo操作满足幂等性:  $\text{Undo}(\text{Undo}(\text{Undo} \dots (x))) = \text{Undo}(x)$
- 后像文件: 记录所有被更新的数据块之新值AI, 用于进行Redo操作。
  - Redo操作同样满足幂等性:  $\text{Redo}(\text{Redo}(\text{Redo} \dots (x))) = \text{Redo}(x)$



## 7.1.2 日志结构与机制

### ■ 在Oracle中，日志及相关机制

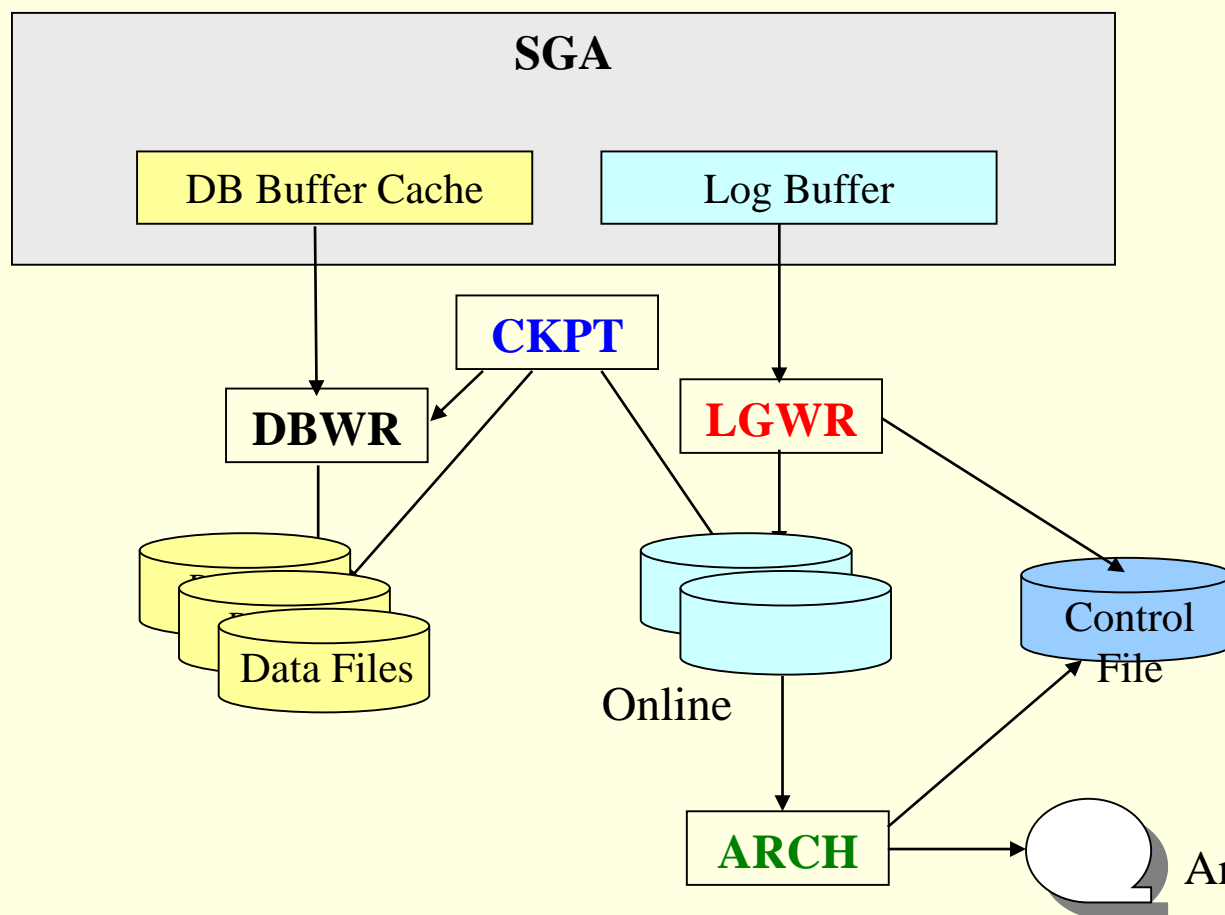
- ATL, (CTL), BI文件→回滚段文件(Rollback Segment File)。
- AI文件称重做日志文件(Redo Log File)。
- 每个Oracle重做日志是一组文件。系统中总是保持至少两个文件组，以循环方式（即一个填满就用另一个）被写入日志项，这些日志文件称在线日志(Online Redo Log)。当一个在线日志文件被填满后，可以将其复制到离线存储设备（磁带）上，称为归档日志(Archived Redo Log)。
- Oracle在线日志文件可以被镜像！



## 7.1.2 日志结构与机制

### ■ 几个后台进程：

- **DBWR**进程—负责DB写入
- **LGWR**进程—负责形成在线日志
- **ARCH**进程—负责形成归档日志
- **CKPT**进程—负责产生检查点



# 目录 Contents

- 7.1 数据库恢复
  - 恢复的基本技术
  - 日志结构与机制
  - 更新事务的执行与恢复
  - 各类失效的具体恢复对策
- 7.2 并发控制
  - 并发控制概述
  - 加锁协议
  - 多粒度封锁与意向锁
  - 死锁的检测、处理和预防



## 7.1.3 更新事务的执行与恢复

■ 为保证数据库是可恢复的，更新事务在系统具有日志机制后，其执行应遵守下列**两条规则**：

### ■ 提交规则(Commit Rule)

- 后像AI必须在事务提交前写入非易失存储器(即数据库或日志文件)。
- 通常是立即写入DB Buffer Cache和Log文件，以便当发生故障时，能通过Redo而恢复。

### ■ 先记后写规则(Log Ahead Rule)

- 如果AI在事务提交前写入数据库，则必须首先把BI记入日志。
- **写日志文件操作**：把表示这个修改的日志记录写到日志文件
- **写数据库操作**：把对数据的修改写到数据库中
- 以便一旦事务在提交前瞬间失败时，能通过Undo而恢复。





## 7.1.3 更新事务的执行与恢复


### ■ 为什么要先记后写？

- 写数据库和写日志文件是两个不同的操作。
- 在这两个操作之间可能发生故障。
- 如果先写了数据库修改，而在日志文件中没有登记下这个修改，则以后就无法恢复这个修改。
- 如果先写日志，但没有修改数据库，按日志文件恢复时只不过是多执行一次不必要的UNDO操作，并不会影响数据库的正确性。



## 7.1.3 更新事务的执行与恢复

- 根据后像(AI)写入数据库的时间不同, 有**三种可选方案**
  - **后像在事务提交前完全写入数据库, 步骤如下**
    - <1> TID→ATL
    - <2> BI→Log      /\* 先记后写规则 \*/
    - <3> AI→DB      /\* 提交规则      \*/      /\* AI直接写入数据库 \*/
    - <4> TID→CTL
    - <5> 从ATL删除TID
    - 当事务执行中发生故障时, 根据ATL和CTL中是否有该事务的TID, 采取不同的恢复措施

ATL	CTL	事务所处状态	恢复措施
有	—	(1)已完成,但(4)尚未完成	(1) 若 BI已写入日志, 则UNDO,否则无需UNDO (2) 从ATL删除TID
有	有	(4)执行完	从ATL删除TID
	有	(5)执行完	无需处理

## 7.1.3 更新事务的执行与恢复

- 后像在事务提交后才写入数据库，步骤如下：
  - <1> TID→ATL
  - <2> AI→Log            /\* 提交规则    \*/   /\* AI先写入日志文件 \*/
  - <3> TID→CTL
  - <4> AI→DB            /\* AI再写入数据库 \*/
  - <5> 从ATL删除TID
- 后像在事务提交前未写入数据库，根据先记后写原则，不必记入前像。

ATL	CTL	事务所处状态	恢复措施
有	—	(1)已完成,但(3)尚未完成	从ATL删除TID
有	有	(3)已完成,但(5)尚未完成	(1) REDO (2) 从ATL删除TID
—	有	(5)执行完	无需处理



## 7.1.3 更新事务的执行与恢复

- 后像在事务提交前后写入数据库，步骤如下：
  - <1> TID→ATL
  - <2> AI、BI→Log    /\* 满足两条规则 \*/    /\* AI先写入日志 \*/
  - <3> AI<sup>P</sup>→DB    /\* AI部分写入DB \*/
  - <4> TID→CTL
  - <5> AI<sup>F</sup>→DB    /\* AI全部写入DB/
  - <6> 从ATL删除TID

ATL	CTL	事务所处状态	恢复措施
有	—	(1)已完成,但(4)尚未完成	(1) 若 BI已写入日志, 则UNDO,否则 无需UNDO (2) 从ATL删除TID
有	有	(4)已完成,但(6)尚未完成	(1) REDO (2) 从ATL删除TID
—	有	(6)执行完	无需处理



## 7.1.3 更新事务的执行与恢复

### ■ 方案之间的比较

方案	redo	undo	AI	BI	执行的并发度	使用情况
1	—	√	—	√	差	很少
2	√	—	√	—	中	较多
3	√	√	√	√	好	最多



# 目录 Contents

- 7.1 数据库恢复
  - 恢复的基本技术
  - 日志结构与机制
  - 更新事务的执行与恢复
  - 各类失效的具体恢复对策
- 7.2 并发控制
  - 并发控制概述
  - 加锁协议
  - 多粒度封锁与意向锁
  - 死锁的检测、处理和预防



## 7.1.4 各类失效的具体恢复对策

在Oracle中，以下两种失效的恢复称**实例恢复(Instance Recovery)**

### ■ 事务失效的恢复

- 正如前所述，由于事务失效必然发生在事务提交之前，故恢复措施/步骤：
  - 如果需要，则进行undo操作（因为BI已写入Log）；
  - 从ATL中删除该事务的TID，释放其所占的资源。
- 以上步骤由系统自动进行，无需DBA介入。

### ■ 系统失效的恢复

- 首先要使系统恢复正常进行；其次在失效发生时，可能有已提交事务的更新丢失了，可能有未提交的事务夭折了。因此，故障恢复措施/步骤：
  - 重新启动OS和DBMS；（由DBA进行）
  - 利用日志中的前像（BI）对未提交事务进行undo操作（向后恢复），利用日志中的后像（AI）对已提交事务进行Redo操作（向前恢复）。（由系统自动进行）



## 7.1.4 各类失效的具体恢复对策

### ■ 注：

- 在故障发生时，未提交的事务是有限的，故undo工作量较少；但已提交的事务是大量的，故Redo工作量很大。而且，到底从那一个点开始Redo呢？（—太前了很浪费，太后了会失更新）。
- 一般DB系统中都设置检查点(Check point)机制。每隔一段时间（如一分钟）产生一个CP点，在CP点上，DBMS强制将内存中DB Buffer Cache中已修改的数据块(后像AI)写入到数据库的Data Files中。
- 这样，在最近一个CP点之前提交的事务就不必在恢复时Redo了，CP同时写入日志文件中 — 大大减少了向前恢复的工作量！
- Oracle中CKPT进程就承担此工作。





## 7.1.4 各类失效的具体恢复对策

### ■ 介质失效的恢复

- 在Oracle中，称介质恢复（Media Recovery）
- 由于介质失效的特点是DB已不可用了，故必须首先利用Backup来恢复DB，然后，再利用日志进行向前恢复。因此，故障恢复措施/步骤：
  - 修复系统，必要时更换磁盘；（由系统管理员或DBA进行）
  - 重启系统OS和DBMS；（由系统管理员或DBA进行）
  - 加载最近后备（Backup）；（由DBA进行）
  - 利用日志中的后像（AI），重做（Redo）自Backup后的所有已提交事务的更新操作。（由系统自动进行）



# 小结

- DBMS必须对事务故障、系统故障和介质故障进行恢复
- 恢复中最经常使用的技术：数据后备复本和日志
- 恢复的基本原理：利用存储在后备副本、日志文件和数据库镜像中的冗余数据来重建数据库
- 常用恢复措施
  - 事务故障的恢复: UNDO
  - 系统故障的恢复: UNDO + REDO
  - 介质故障的恢复: 重装备份并恢复到一致性状态 + REDO
- 提高恢复效率的技术
  - 检查点技术
    - 可以提高系统故障的恢复效率; 在一定程度上提高利用动态转储备份进行介质故障恢复的效率



# 目录 Contents

## ■ 7.1 数据库恢复

- 恢复的基本技术
- 日志结构与机制
- 更新事务的执行与恢复
- 各类失效的具体恢复对策

## ■ 7.2 并发控制

- 并发控制概述
- 加锁协议
- 多粒度封锁与意向锁
- 死锁的检测、处理和预防



## 7.2.1 并发控制概述

### ■ 并发访问与并发控制

- 数据库是一个多用户共享系统
- 多个事务的执行方式
  - 串行访问(Serial Access): DBMS一次只可接纳一个事务, 事务串行地被执行, 即一个结束另一个才开始。
    - 不能充分利用系统资源, 发挥数据库共享资源的特点
  - 并发访问(Concurrent Access): DBMS可以同时接纳多个事务, 事务可以在时间上重叠地执行。
    - 交叉并发方式(interleaved concurrency): 并行事务的并行操作轮流交叉运行, 是单处理机系统中的并发方式。能够减少处理机的空闲时间, 提高系统的效率。
    - 同时并发方式(simultaneous concurrency): 多处理机系统中, 每个处理机可以运行一个事务, 多个处理机可以同时运行多个事务, 实现多个事务真正的并行运行。最理想的并发方式, 但受制于硬件环境。更复杂的并发方式机制。



## 7.2.1 并发控制概述

### 并发访问与并发控制(cont.)

- 数据库系统中，事务必然是并发访问的。

- 性能方面

- 多个用户往往同时存取数据库，实际上，即往往同时有多个事务向DBMS提出申请，如果DBMS串行服务，则必然有“排队等待”，这样极大地影响系统性能。特别是“短事务”，更感觉“响应时间太慢”。（supermarket）

- 资源利用率方面

- 一个事务在执行过程中，不同执行阶段需不同的资源(CPU、I/O、Comm...)。如果串行执行，必然有“资源的闲置”，影响资源利用率；如果并行执行，既充分利用了系统资源，又可缩短响应时间。

- 但是，并发访问环境下，还能保证所有事务都满足ACID准则吗？—回答是：如果不加“控制”，那么就不能！



## 7.2.1 并发控制概述

- 并发访问与并发控制(cont.)
  - 并发所引起的不一致问题
    - 丢失更新 (Lost Update)
      - 源于：写—写冲突 (Write—Write Conflict)
    - 读脏数据 (Dirty Read)
      - 源于：读—写冲突 (Read—Write Conflict)
    - 读值不可复现 (Unrepeatable Read)
      - 源于：读—写冲突



## 7.2.1 并发控制概述

### ■ 并发所引起的不一致问题 1

#### ■ 丢失更新 (Lost Update)

- 丢失更新是指事务1与事务2从数据库中读入同一数据并发地写入，事务2的提交结果破坏了事务1提交的结果，导致事务1写入的数据被丢失。

#### 现象

一个事务的修改结果破坏了另一个事务的更新结果

#### 原因

对多个事务并发更新同一个数据对象的情况未加控制

$T_1$	$T_2$
① 读A=16	
②	读A=16
③ $A \leftarrow A+1$ 写回A=17	
④	$A \leftarrow A*2$ 写回A=32



## 7.2.1 并发控制概述

### ■ 并发所引起的不一致问题 2

#### ■ 读脏数据(Dirty Read)

- 事务1修改某一数据，并将其写回磁盘。事务2读取同一数据后，事务1由于某种原因被撤消，这时事务1已修改过的数据恢复原值。事务2读到的数据就与数据库中的数据不一致，是**不正确的数据**，又称为**“脏”数据**。

#### 现象

读到了错误的数据（即与数据库中的情况不相符的数据）

#### 原因

一个事务读取了另一个事务未提交的修改结果

$T_1$	$T_2$
① 读 $C=100$ $C \leftarrow C * 2$ 写回 $C$	读 $C=200$
②	
③ ROLLBACK $C$ 恢复为100	





## 7.2.1 并发控制概述

### ■ 并发所引起的不一致问题 3

#### ■ 读值不可复现 (Unrepeatable Read)

- 指事务1读取数据后，事务2执行更新操作，使事务1无法再现前一次读取结果。
- 三类读值不可复现，事务1读取某一数据后：
  - 事务2对其做了修改，当事务1再次读该数据时，得到与前一次不同的值。
  - 事务2删除了其中部分记录，当事务1再次读取数据时，发现某些记录神秘地消失了。
  - 事务2插入了一些记录，当事务1再次按相同条件读取数据时，发现多了一些记录。
- 后两种不可重复读有时也称为幻影现象 (phantom row)



## 7.2.1 并发控制概述

- 并发所引起的不一致问题 3
  - 读值不可复现 (Unrepeatable Read)

### 现象

在一个事务的执行过程中, 前后两次读同一个数据对象所获得的值出现了不一致。

### 原因

在两次‘读’操作之间插入了另一个事务的‘写’操作

$T_1$	$T_2$
① 读A=50 读B=100 求和=150	
②	读B=100 B←B*2 写回B=200
③ 读A=50 读B=200 求和=250 (验算不对)	



## 7.2.1 并发控制概述

### ■ 并发访问与并发控制(cont.)

- 并发控制(Concurrency Control): 控制事务的并发访问, 以避免访问冲突所引起的数据不一致, 保证DB始终处于一致状态。
- DBMS必须提供并发控制机制。
- 并发控制机制是衡量一个DBMS性能的重要标志之一。



## 7.2.1 并发控制概述

### 并发控制的正确性准则

- **调度(Schedule)**: 在某一时刻, DBMS对并发访问的一组事务  $\{T_1, T_2, \dots, T_n\}$  的所有操作步骤的顺序的一个安排, 可形式地表示成:

$$S = \dots R_i(x) \dots W_j(x) \dots R_k(y) \dots \quad i, j, k \in \{1, 2, \dots, n\},$$

$R_i(x)$ 表示事务 $T_i$ 对数据 $x$ 的一个读操作,  $W_j(x)$ 表示事务 $T_j$ 对数据 $x$ 的一个写操作,  $R_k(y)$ 表示事务 $T_k$ 对数据 $y$ 的一个读操作。

- 可见对同一个  $\{T_1, T_2, \dots, T_n\}$  有多种调度 $S$ 。



## 7.2.1 并发控制概述

T1	T2
-----	
READ(A,t)	READ(A,s)
t:=t+100	s:=s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t:=t+100	s:=s*2
WRITE(B,t)	WRITE(B,s)

约束条件：A=B



## 7.2.1 并发控制概述

### Schedule A

		A	B
T1	T2	25	25
Read (A); $A \leftarrow A+100$		125	
Write (A);			
Read (B); $B \leftarrow B+100$			125
Write (B);			
	Read (A); $A \leftarrow A \times 2$ ;	250	
	Write (A);		
	Read (B); $B \leftarrow B \times 2$ ;		250
	Write (B);		
		250	250



## 7.2.1 并发控制概述

### Schedule B

T1	T2	A	B
		25	25
	Read (A); $A \leftarrow A \times 2$ ;		
	Write (A);	50	
	Read (B); $B \leftarrow B \times 2$ ;		
	Write (B);		50
Read (A); $A \leftarrow A + 100$ ;			
Write (A);		150	
Read (B); $B \leftarrow B + 100$ ;			
Write (B);			150
		150	150



## 7.2.1 并发控制概述

### ■ 串行调度(Serial Schedule)

- 导致事务串行访问的调度。
- 对于任何两个事务 $T$ 和 $T'$ ，如果 $T$ 的某个action在 $T'$ 之前，那么 $T$ 的所有actions都在 $T'$ 之前，这样的调度称为串行的。
- 只要一致性得到保证，最终的结果值并不是主要的。（在两个调度中，最终结果值可能会不一样）





# Schedule C

T1	T2	A	B
		25	25
Read (A); $A \leftarrow A+100$			
Write (A);		125	
	Read (A); $A \leftarrow A \times 2$ ;		
	Write (A);	250	
Read (B); $B \leftarrow B+100$			
Write (B);			125
	Read (B); $B \leftarrow B \times 2$ ;		
	Write (B);		250
		250	250

一个非串行的可串行化调度



# Schedule D

T1	T2	A	B
		25	25
Read (A); $A \leftarrow A+100$			
Write (A);		125	
	Read (A); $A \leftarrow A \times 2$ ;		
	Write (A);	250	
	Read (B); $B \leftarrow B \times 2$ ;		
	Write (B);		50
Read (B); $B \leftarrow B+100$			
Write (B);			150
		250	150

一个非可串行化调度



## 7.2.1 并发控制概述

- 要得到正确的调度，不需要考虑
  - 初始状态
  - 事务的语义

- 仅需要考察读写操作的次序

$$Sc = R1(A)W1(A)R2(A)W2(A)R1(B)W1(B)R2(B)$$

- 等价的调度：给定对同一组事务的两个调度S1和S2，如果在DB的任何初始状态下，所有从DB中读出的数据都是一样的，留给DB的最终状态也是一样的，则称S1和S2是等价的调度。这种等价也称目标等价(View Equivalence)。



## 7.2.1 并发控制概述

### ■ 并发控制的正确性准则(cont.)

- 有两种操作对是冲突的，分别：
  - 读—写冲突：表示为： $R_i(x)$ 和 $W_j(x)$
  - 写—写冲突：表示为： $W_i(x)$ 和 $W_j(x)$
- 另三种操作对是不冲突的，分别是：
  - $R_i(x)$ 和 $R_j(x)$ ， $R_i(x)$ 和 $R_j(y)$ ， $W_i(x)$ 和 $W_j(y)$
- 不冲突的操作之间可以相互调换次序，不会影响执行结果。
- 冲突等价(Conflict Equivalence)：通过调换(不同事务间的)不冲突操作次序而得到的新调度。
- 可见，目标等价更普遍，冲突等价的调度→目标等价的调度。



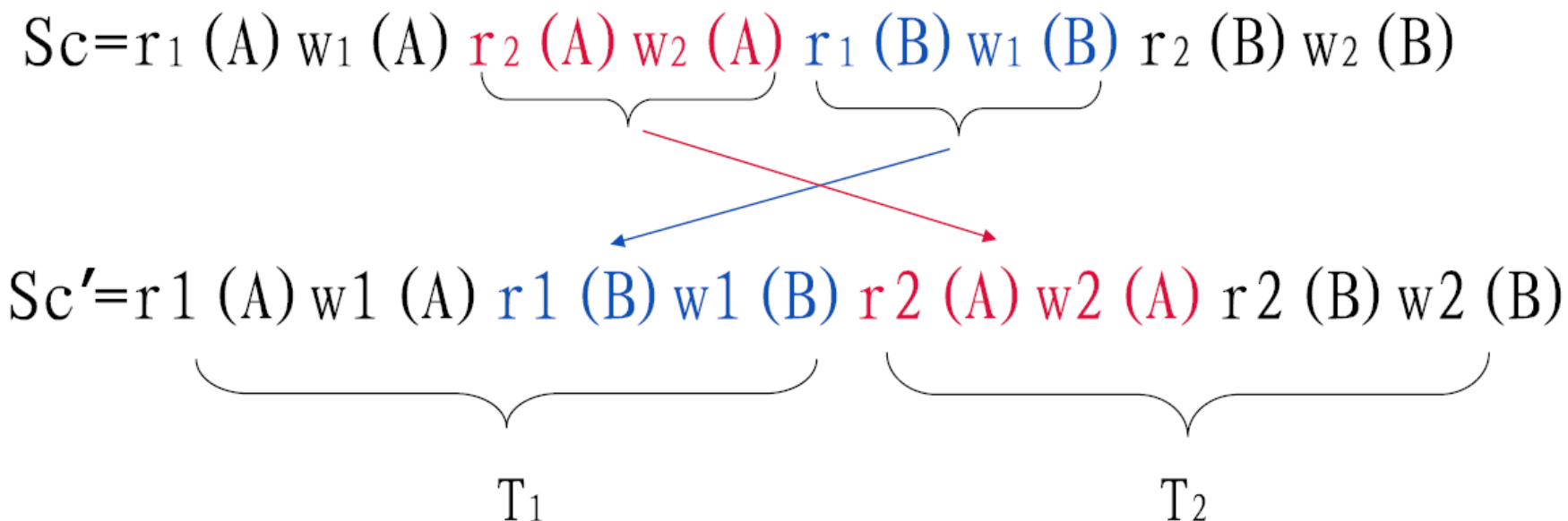
## 7.2.1 并发控制概述

### 并发控制的正确性准则(cont.)

- 串行调度(Serial Schedule): 导致事务串行访问的调度。
- 并行调度(Concurrent Schedule): 导致事务并行访问的调度。
- 可串行化调度(Serializable Schedule): 如果一个调度与某个串行调度是等价的, 则称它为可串行化调度。
- 同样有: 冲突可串行化、目标可串行化
  - 冲突可串行化→目标可串行化
- 由于串行调度导致事务的串行访问(事务间不会有相互影响), 总能保持数据库的一致性。可串行化调度与串行调度等价, 因此, 可串行化调度也总能保持数据库的一致性。
- 可串行化是并事务正确性的唯一准则



## 7.2.1 并发控制概述




$Sc$  is "equivalent" to a serial schedule  
(in this case  $T_1, T_2$ )

可串行化调度



## 7.2.1 并发控制概述

$S_d = r_1(A) w_1(A) r_2(A) w_2(A) r_2(B) w_2(B) r_1(B) w_1(B)$



$S_d$  cannot be rearranged into a serial schedule

$S_d$  is not "equivalent" to any serial schedule

$S_d$  is "bad"



## 7.2.1 并发控制概述

- **例1:** 设  $T_1 : R_1(y)$   $T_2 : R_2(x), W_2(y)$   $T_3 : W_3(x)$   
则  $\{T_1, T_2, T_3\}$  的一个并行调度:

$$\begin{aligned} Sc &= R_2(x) \overline{W_3(x)} \overline{R_1(y)} W_2(y) \\ &\longleftrightarrow \overline{R_2(x)} \overline{R_1(y)} \overline{W_3(x)} \overline{W_2(y)} \\ &\longleftrightarrow R_1(y) R_2(x) W_2(y) W_3(x) = T_1 T_2 T_3 = Ss \text{ (串行调度)} \end{aligned}$$

故  $Sc$  是（冲突）是串行化的。





## 7.2.1 并发控制概述

- **例2:** 设  $T_1 : R_1(x), W_1(x)$   $T_2 : W_2(x)$   $T_3 : W_3(x)$  则  $\{T_1, T_2, T_3\}$  的一个并行调度:
  - $Sc = R_1(x) W_2(x) W_1(x) W_3(x)$  无法调换(均是冲突操作)
  - 但是根据 “**目标等价**” 定义,  
 $Ss = R_1(x) W_1(x) W_2(x) W_3(x) = T_1 T_2 T_3$  (串行调度)
- 由上两例说明:
- 冲突等价/冲突可串行化调度  $\iff$  目标等价/目标可串行化调度



## 7.2.1 并发控制概述

### 并发控制的正确性准则(cont.)

#### 结论

1. 由于“可串行化调度”能保持DB的一致状态，因此，一般DBMS都以可串行化作为并发控制的正确性准则
2. “目标可串行化”比“冲突可串行化”更普遍，但由于其测试算法是NP完全问题，而后者可通过简单算法(如前超图(Precedence Graph)拓扑排序法)来判断，故总是以“冲突可串行化”作为具体的正确性准则。  
(即使明知会漏掉一些可串行化调度)



# Precedence Graph

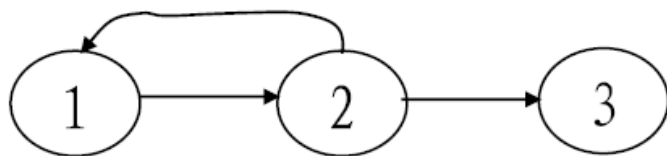
S1:  $r_2(A)$ ;  $r_1(B)$ ;  $w_2(A)$ ;  $r_2(B)$ ;  $r_3(A)$ ;  $w_1(B)$ ;  $w_3(A)$ ;  $w_2(B)$ ;

从  $w_2(A)$ ;  $r_3(A)$  可以看出:  $T_2 <_S T_3$ , 因此  $T_2 \rightarrow T_3$

从  $r_1(B)$ ;  $w_2(B)$  可以看出:  $T_1 <_S T_2$ , 因此  $T_1 \rightarrow T_2$

从  $r_2(B)$ ;  $w_1(B)$  可以看出:  $T_2 <_S T_1$ , 因此  $T_2 \rightarrow T_1$

S1的调度优先图:



调度优先图有环, S1不是冲突可串行化的。



## 7.2.1 并发控制概述

### ■ 并发控制的正确性准则(cont.)

#### ■ 结论

3. 然而在实际系统中，不可能“通过检测是否可串行化”来控制并发访问、保证正确性准则。而是通过遵守“加锁协议”(Locking Protocol)的办法来保证以上正确性准则的。(因为：事务是随机到达和退出的，没有一个固定的事务集等待DBMS调度/或者事务集一直在动态变化，难道频繁地“检测”吗？)
4. 一用“加锁”，即可能产生活锁(Live Lock)和死锁(Dead Lock)之问题，由此即产生了如何检测和如何预防的问题。



# 目录 Contents

## ■ 7.1 数据库恢复

- 恢复的基本技术
- 日志结构与机制
- 更新事务的执行与恢复
- 各类失效的具体恢复对策

## ■ 7.2 并发控制

- 并发控制概述
- 加锁协议
- 多粒度封锁与意向锁
- 死锁的检测、处理和预防



## 7.2.2 加锁协议

- 在事务并发调度时，遵守“加锁协议”，即在执行任何事务的任一操作（R/W）之前对操作对象（数据）进行加锁，并遵循一定的协议，就可保证并发调度是（冲突）可串行化的，从而达到并发控制的目标。



## 7.2.2 加锁协议

### ■ 加锁

#### ■ 使用加锁技术的前提

- 在一个事务访问数据库中的数据时，必须先获得相应的访问对象上的加锁，以保证数据访问操作的正确性和一致性。

#### ■ 加锁的作用

- 在一段时间内禁止其它事务在被加锁的数据对象上执行某些类型的操作。

- 由加锁的类型决定

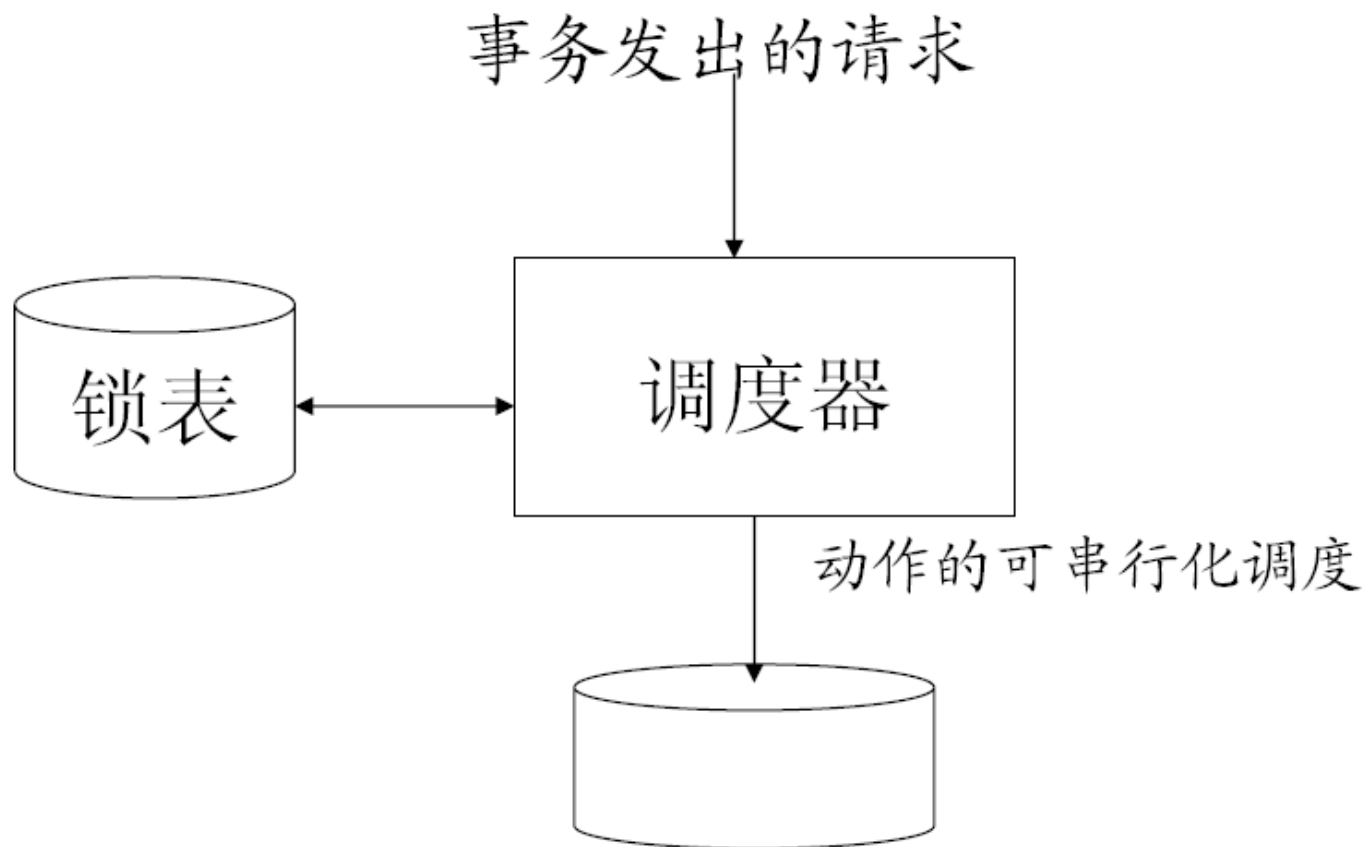
- 同时也表明：持有该加锁的事务在被加锁的数据对象上将要执行什么类型的操作。

- 由系统所采用的加锁协议来决定

- ‘加锁’是多用户环境中最常采用的一种并发控制技术



## 7.2.2 加锁协议



使用锁表的调度器





## 7.2.2 加锁协议

### ■ 使用X锁的加锁协议

- X锁: 排它锁(eXclusive Lock)
- X锁的特性

- 只有当数据对象A没有被其它事务加锁时, 事务T才能在数据对象A上施加X锁;
- 如果事务T在数据对象A上施加了X锁, 则其它任何事务都不能在数据对象A上再施加任何类型的锁。
- 加锁协议可以用一个相容矩阵(Compatibility Matrix)来定义

	其他事务已拥有的锁	
	NL	X
本事务锁请求 X	Yes	No



## 7.2.2 加锁协议

### ■ 使用X锁的加锁协议(cont.)

#### ■ 协议

- ① 任何事务的任何操作（R/W）在能够执行之前，必须获得此操作对象的X锁（称加X锁），操作完成后要释放X锁。
  - 注：仅有①会产生问题：
    - 读脏数据
    - 读值不可复现
    - 连锁回滚(Cascading Rollback)。
- ② 每个X锁必须保持到该事务结束（End of Transaction, EOT）时才能释放 — 此条协议同样适合于以下任何加锁协议。



## 7.2.2 加锁协议

### 使用X锁的加锁协议(cont.)

- 仅有①会产生问题：读脏数据、读值不可复现、连锁回滚

$T_1$	$T_2$
.	.
.	.
<b>Lock (D)</b>	.
<b>Write1 (D)</b>	<b>Lock (D)</b>
<b>Unlock (D)</b>	<b>Wait</b>
.	.
.	<b>Lock (D)</b>
.	<b>Read2(D)</b>
<b>(rollback)</b>	<b>Unlock (D)</b>

$T_1$	$T_2$
.	.
<b>Lock (D)</b>	.
<b>Read1 (D)</b>	.
<b>Unlock (D)</b>	.
.	<b>Lock (D)</b>
.	<b>Write2(D)</b>
.	<b>Unlock (D)</b>
<b>Lock (D)</b>	
<b>Read'1 (D)</b>	
<b>Unlock (D)</b>	



## 7.2.2 加锁协议

### ■ 使用X锁的加锁协议(cont.)

#### ■ 作用

- 如果一个事务T申请在数据对象A上施加X锁并得到满足，则事务T自身可以对数据对象A作读、写操作，而其它事务则被禁止访问数据对象A。
  - 这样可以让事务T独占该数据对象A，从而保证了事务T对数据对象A的访问操作的正确性和一致性。
    - 缺点：降低了整个系统的并行性
  - X锁必须维持到事务T的执行结束



## 7.2.2 加锁协议

### ■ 使用X锁的加锁协议(cont.)

#### ■ 特点:

- 简单，只有一种锁。
- 并发性较差，即使是“读一读”也要用“排它”之锁。



## 7.2.2 加锁协议

### ■ 定义：合式事务

- 一个事务如果遵守“先加锁，后操作”的原则，则称此事务为合式事务(Well-formed Transaction)。
- 即：一个事务在访问数据库中的数据对象A之前按照要求申请对A的加锁，在操作结束后释放A上的加锁。
- 合式事务是保证并发事务的正确执行的基本条件。



T1	T2	A	B
L1 (A) ; r1 (A) ; A: =A+100; W1 (A) ; u1 (A) ;		25  125	25
	L2 (A) ; r2 (A) ; A: =A*2; W2 (a) ; u2 (A) ; L2 (B) ; r2 (B) ; B: =B*2; W2 (B) ; U2 (B) ;	250	50
L1 (B) ; r1 (B) ; B: =B+100; W1 (B) ; u1 (B)			150

- 一个合法调度，但是一个合法不可串行化的调度。



## 7.2.2 加锁协议

- 上页的调度尽管是合法的，却不是可串行化的。
- 保证合法冲突可串行性所需附件的条件是“两段封锁协议”
- 定义：两段事务和两段锁协议
  - 在一个事务中，如果所有加锁动作都在所有释放锁动作之前，则称这样的事务是两段事务(Two-Phase Transaction) — 隐含了X锁的加锁协议①。
  - 以上限制也称两段封锁协议(two-Phase Locking protocol, 2PL协议)
  - 例如：

$T_1 : \underbrace{Lock(A)Lock(C)Lock(B)}_{\text{锁增长阶段性(growing)}} \underbrace{Unlock(C)Unlock(A)Unlock(B)}_{\text{锁收缩阶段(shrinking)}}$





## ■ 不遵守2PL条件

T1	T2	A	B
$L1(A); r1(A);$ $A := A + 100;$ $W1(A); u1(A);$		25  125	25
	$L2(A); r2(A);$ $A := A * 2;$ $W2(a); u2(A);$ $L2(B); r2(B);$ $B := B * 2;$ $W2(B); U2(B);$	250	50
$L1(B); r1(B);$ $B := B + 100;$ $W1(B); u1(B)$			150



## ■ 遵守2PL条件

T1	T2	A	B
L1 (A); r1 (A); A: =A+100; W1 (A); 11 (B); u1 (A);		25  125	25
	L2 (A); r2 (A); A: =A*2; W2 (A); L2 (B) 被拒绝	250	
r1 (B); B: =B+100; W1 (B); u1 (B)			125
	L2 (B) ; u2 (A); r2 (b) B: =B*2; W2 (B); U2 (B);		250



## 7.2.2 加锁协议

- 定理：如果所有事务都是合式、两段事务，则它们的任何调度都是可串行化的。

- 注：①有的书上将“合式”作为事务的基本特性，则遵守两段封锁协议即可保证事务的任何调度都是可串行化的。(充分条件)

- 证明见教材(反证法)：前超图中是否有回路。

- 但是，2PL协议并不是调度可串行化的必要条件。

- 例：  $S = R_2(x)W_3(x)R_1(y)W_2(y)$ ，不是两段事务，却是可串行化的。

$$S' = R_1(y)R_2(x)W_2(y)W_3(x)$$

- ②上述结论，对以下加锁协议同样适用。



## 7.2.2 加锁协议

### ■ 使用(S, X)锁的加锁协议

#### ■ 使用两种锁：

- X锁：排它锁(eXclusive Lock)，用于写操作。
- S锁：共享锁(SHaring Lock)，用于读操作。

#### ■ S锁的特性

- 如果数据对象A没有被其它事务加锁，或者其它事务仅仅以S锁的方式来封锁数据对象A时，事务T才能在数据对象A上施加S锁。



## 7.2.2 加锁协议

### ■ 使用(S, X)锁的加锁协议(cont.)

#### ■ S锁的作用

- 如果一个事务T申请在数据对象A上施加S锁并得到满足，则事务T可以读数据对象A，但不能写数据对象A。
  - 不同事务所申请的S锁可以共存于同一个数据对象A上，从而保证了多个事务可以同时读数据对象A，有利于提高整个系统的并发性
    - 在持有加锁的事务释放数据对象A上的所有S锁之前，任何事务都不能写数据对象A
- S锁不必维持到事务T的执行结束



## 7.2.2 加锁协议

### ■ 使用(S, X)锁的加锁协议(cont.)

- 用一个相容矩阵(Compatibility Matrix)来定义(S, X)锁

		其它事务已持有的锁		
		X锁	S锁	—
当前事务申请的锁	X锁	No	No	Yes
	S锁	No	Yes	Yes

提高并发度（允许R-R并发），但可能发生“活锁”现象

- 每个(S, X)锁必须保持到该事务结束(End of Transaction, EOT)时才能释放



## 7.2.2 加锁协议

- 使用(S, X)锁的加锁协议(cont.)

- 活锁(Live Lock)

- 如果不断有事务申请对数据对象的S锁，而某个数据对象始终被S锁占有，而X锁申请迟迟不能获准，这种现象称为活锁。
- 活锁对系统的性能有不良影响。



## 7.2.2 加锁协议

- 使用(S, X)锁的加锁协议(cont.)
  - 活锁预防：在加锁协议中应规定先来先服务(First Come, First Serve)原则。
    - 当多个事务请求封锁同一数据对象时
      - 按请求封锁的先后次序对这些事务排队
      - 该数据对象上的锁一旦释放，首先批准申请队列中第一个事务获得锁。
  - 这样，比X锁后申请的S锁不会先于X锁获准了。





## 7.2.2 加锁协议

### ■ 使用(S, U, X)锁的加锁协议

#### ■ 使用三种锁：

- **X锁**：排它锁(e**X**clusive Lock)，用于**写**操作。
- **S锁**：共享锁(**S**haring Lock)，用于**读**操作。
- **U锁**：共享更新锁(Sharing **U**ppdate Lock, SU)，
  - 被事务用来“预置”对某个数据对象将进行**更新**的权力，并防止其它事务对这个数据对象进行写操作或加X锁。在最后更新写入前，系统将U升级为X而排它。
  - 事务在更新一个数据对象时，首先申请对它的U锁。数据对象加了U锁后，仍允许其他事务对它加S锁。待最后写入时，事务再申请把U锁升级为X锁。
  - **提高事务并发度**



## 7.2.2 加锁协议

### ■ 使用(S, U, X)锁的加锁协议(cont.)

- 可用一个相容矩阵(Compatibility Matrix)来定义它们:

		其它事务已持有的锁			
		X锁	U锁	S锁	—
当前事务申请的锁	X锁	No	No	No	Yes
	U锁	No	No	Yes	Yes
	S锁	No	Yes	Yes	Yes

提高并发度（允许W前操作-R并发）。  
但可能发生“活锁”现象（S一直占有，  
U久久不能升级为X）

- 为避免活锁，规定先来先服务原则。



## 7.2.2 加锁协议

### ■ 使用加锁的并发控制技术

- 加锁由DBMS统一管理。

- DBMS的加锁管理器中维护着一张锁表，以记录当前各个数据对象加锁的情况：

- 锁的持有情况

- 有哪些‘事务’在哪些‘数据对象’上持有什么类型的‘封锁’

- 锁的申请情况

- 有哪些‘事务’正在申请哪些‘数据对象’上的什么类型的‘封锁’



## 7.2.2 加锁协议

### ■ 使用加锁的并发控制技术(cont.)

- 事务如果需要对某些数据进行操作，必须向DBMS申请。DBMS根据锁表的状态和加锁协议，同意其申请或令其等待。
- 锁表是DBMS的公共资源，且访问频繁，通常置于公共内存区。如果系统失效，锁表内容也随之失效，无保留价值。



# 目录 Contents

## ■ 7.1 数据库恢复

- 恢复的基本技术
- 日志结构与机制
- 更新事务的执行与恢复
- 各类失效的具体恢复对策

## ■ 7.2 并发控制

- 并发控制概述
- 加锁协议
- 多粒度封锁与意向锁
- 死锁的检测、处理和预防



## 7.2.3 多粒度封锁与意向锁

### ■ 封锁粒度(Locking Granularity)

#### ■ 一把锁可以封锁的数据对象的大小

- 锁的封锁对象可以是数据库中的逻辑数据单元，也可以是物理数据单元。

- 以关系数据库系统为例，可以采用的封锁粒度有：

- 逻辑数据单元

- 属性值(集合)，元组，关系
- 索引项，索引文件
- 整个数据库

- 物理数据单元

- 页，块



## 7.2.3 多粒度封锁与意向锁

- 单粒度封锁(Single-granularity Locking)
  - 固定一种粒度，如“表”。
  - SGL简单。
  - 但是若G太大，则并发度较低；若G太小，锁太多（对大操作）、维护开销大。



## 7.2.3 多粒度封锁与意向锁

- **多粒度封锁(Multiple-granularity Locking)**: 可有多种粒度, 根据需要选用。
  - 如果在一个系统中同时支持多种封锁粒度供事务选择使用, 这种封锁方法被称为**多粒度封锁**。
  - 通过选择合适的封锁粒度来达到如下目的
    - 通过**加大‘封锁粒度’**来减少‘锁’的数量, 降低并发控制的开销。
    - 通过**降低‘封锁粒度’**来缩小一把锁可以封锁的数据范围, 减少封锁冲突现象, 提高系统并发度。
  - 可以按照封锁粒度的大小构造出一棵**多粒度树**, 以树中的每个结点作为封锁对象, 可以构成一个**多粒度封锁协议**。
  - MGL复杂。但灵活性较大。一般大型DBMS都支持MGL。





## 7.2.3 多粒度封锁与意向锁

- 封锁粒度与系统并发度的关系
- 封锁粒度与并发控制的开销的关系

封锁粒度	系统并发度	并发控制的开销
大	低	小
小	高	大



## 7.2.3 多粒度封锁与意向锁

■ 例如：Oracle支持Table/Row两级封锁，因此有：

■ Locks：

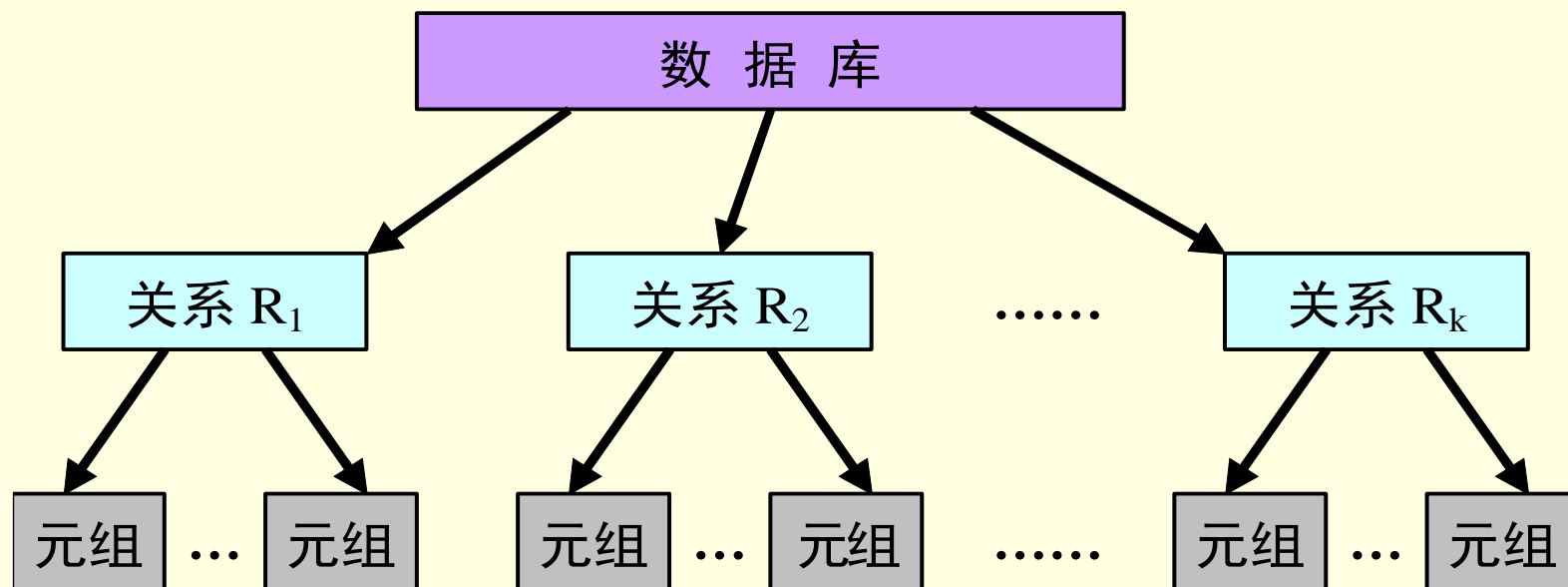
- (Table) Exclusive
- (Table) Share
- (Table) Share Update
- (Row) Share
- (Row) Exclusive



## 7.2.3 多粒度封锁与意向锁

### ■ 多粒度树

■ 例：以关系数据库为例



三个层次的多粒度树



## 7.2.3 多粒度封锁与意向锁

### ■ 多粒度封锁带来的问题及解决方法

#### ■ 一个数据对象可能以两种方式被封锁

##### ■ 显式封锁：Explicit Locking

- 可以对‘多粒度树’中的每个结点独立加锁；

##### ■ 隐式封锁：Implicit Locking

- 对一个结点加锁意味着该结点的所有子孙结点也被加以同样类型的锁。

#### ■ 隐式封锁与显式封锁的作用是一样的。



## 7.2.3 多粒度封锁与意向锁

- 对某个数据对象加锁时系统检查的内容
  - 该数据对象
    - 有无显式封锁与之冲突
  - 所有祖先结点
    - 检查本事务的显式封锁是否与该数据对象上的隐式封锁冲突：（由祖先结点封锁造成的）
  - 所有子孙结点
    - 看上面的显式封锁是否与本事务的隐式封锁（将加到子孙结点的封锁）冲突。



## 7.2.3 多粒度封锁与意向锁

- 如果希望对‘多粒度树’中的某个结点加锁，则需要在该结点的所有祖先结点和子孙结点中都进行锁的相容性检查，“锁冲突”的检查就复杂多了，这样的检查方法效率很低。
  - 为了解决此问题，提出了一种新的封锁类型：
    - 意向锁



## 7.2.3 多粒度封锁与意向锁

### ■ 意向锁(Intention Lock)

- 以简化检查过程，IBM System R 中首先采用
- 意向锁的使用规定
  - 对任一结点加锁时，必须先对它的上层结点加意向锁；
  - 如果对一个结点加意向锁，则说明该结点的下层结点正在被加锁。
  - 在多粒度封锁中，要为一个数据对象加锁，必须先对这个数据对象的所有上级加相应的意向锁。
  - 因此，在申请锁时：自上而下申请；释放锁时：自下而上释放。



## 7.2.3 多粒度封锁与意向锁

### ■ 意向锁(cont.)

- 例：对任一元组  $r$  加X锁，先对关系R加意向锁
  - 事务T要对关系R加X锁，系统只要检查根结点数据库和关系R是否已加了不相容的锁。
  - 不需要搜索和检查R中的每一个元组是否加了X锁。





## 7.2.3 多粒度封锁与意向锁

### ■ 常用意向锁

#### ■ 意向共享锁(Intent Share Lock, IS锁)

- 如果对结点N加IS锁，表示它的某些子孙拟加或已加了S锁。

#### ■ 意向排它锁(Intent Exclusive Lock, IX锁)

- 如果对结点N加IX锁，表示它的某些子孙拟加或已加了X锁。

#### ■ 共享意向排它锁( $SIX=S+IX$ , SIX锁)

- 如果对结点N加SIX锁，表示对结点N本身加S锁，并准备在它的某些子孙拟加或已加了X锁。



## 7.2.3 多粒度封锁与意向锁

### ■ 带有意向锁的锁相容矩阵

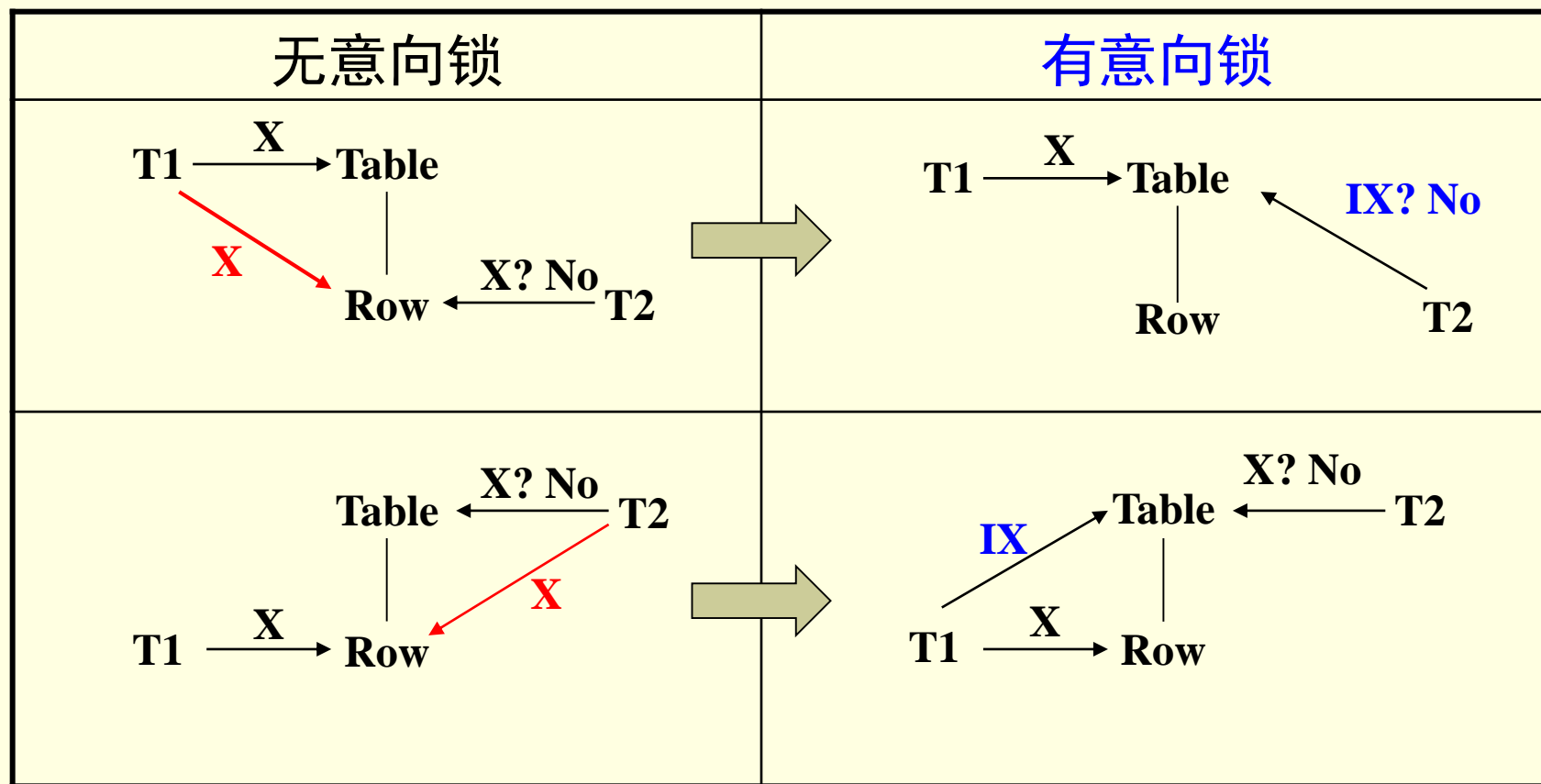
		其它事务已持有的锁				
		S锁	X锁	IS锁	IX锁	SIX锁
当前事务申请的锁	S锁	Yes	No	Yes	No	No
	X锁	No	No	No	No	No
	IS锁	Yes	No	Yes	Yes	Yes
	IX锁	No	No	Yes	Yes	No
	SIX锁	No	No	Yes	No	No

**Yes:** 表示相容的请求      **No:** 表示不相容的请求



## 7.2.3 多粒度封锁与意向锁

### ■ 例



## 7.2.3 多粒度封锁与意向锁

考虑关系：

Movie (title,year,length,studioName)

假设有整个关系和单个元组上的锁。事务 $T_1$ 有如下查询：

SELECT \*

FROM Movie

WHERE title = 'King Kong';

封锁过程：

- 首先需要获得整个关系上的IS锁
- 然后该事务转到单个元组（有两部名字为King Kong的影片），并在它们的每一个上获得S锁。



## 7.2.3 多粒度封锁与意向锁

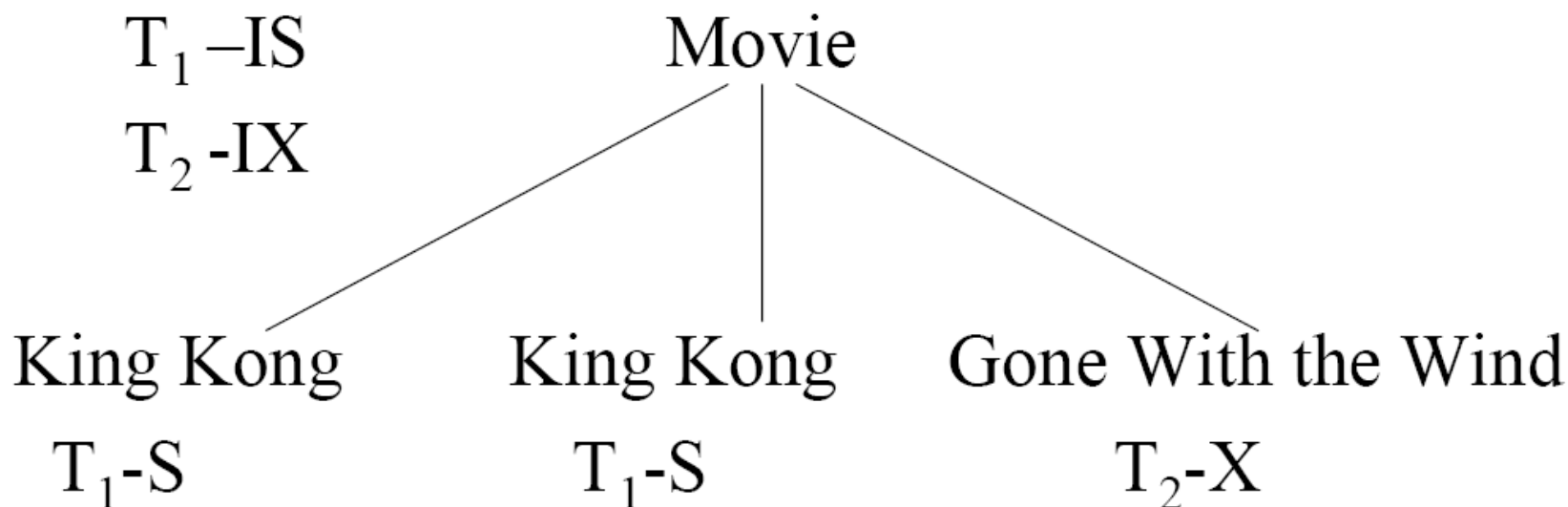
假设在执行T<sub>1</sub>的查询时，事务T<sub>2</sub>开始，它改变元组中的年属性：

UPDATE Movie

SET year = 1939

WHERE title = 'Gone With the Wind' ;

T<sub>2</sub>需要该关系上的一个IX锁。锁的集合如下图：



## 7.2.3 多粒度封锁与意向锁

- 具有‘意向锁’的多粒度封锁方法的优点
  - 减少了加锁和解锁的开销
  - 提高了系统的并发度



# 补充 在Oracle中，锁如何获得/释放

- 有一条扩充的SQL命令，事务可用来显式地获得封锁。

LOCK TABLE表[, 表...] IN **EXCLUSIVE** MODE  
[NOWAIT];

**SHARE**

**SHARE UPDATE**

**ROW EXCLUSIVE**

**ROW SHARE**



# 补充 在Oracle中，锁如何获得/释放

## ■ X锁

### ■ 获得

- 显式方式：LOCK TABLE命令
- 隐式方式：执行如下SQL命令：  
**INSERT INTO表 ...;**  
**UPDATE 表 ...;**  
**DELETE FROM表 ...;**

### ■ 释放

- 四种方式：
  - COMMIT/ROLLBACK（即事务结束时）；
  - DDL操作；
  - 程序终止运行；
  - LOGOFF（即终止会话时）





# 补充 在Oracle中，锁如何获得/释放

## ■ S锁

### ■ 获得：

- 显式方式：LOCK TABLE命令

### ■ 释放：

- 同X锁

## ■ SU锁

### ■ 获得：

- 显式方式：LOCK TABLE命令
- 隐式方式：执行 SELECT ... FOR UPDATE命令

### ■ 释放：

- 同X锁



# 目录 Contents

## ■ 7.1 数据库恢复

- 恢复的基本技术
- 日志结构与机制
- 更新事务的执行与恢复
- 各类失效的具体恢复对策

## ■ 7.2 并发控制

- 并发控制概述
- 加锁协议
- 多粒度封锁与意向锁
- 死锁的检测、处理和预防



## 7.2.4 死锁的检测、处理和预防

### ■ 死锁(deadlock)

- 每个事务都可能拥有一部分锁，并因申请其它事务所持有的锁而等待，因此产生的**循环等待现象被称为死锁**。
  - 也就是，两个事务对锁形成“循环等待”就产生了死锁。
- 死锁的例子
  - 假设在数据库中有两个数据对象：A和B，并存在下述两个事务：
    - **事务T1**：根据读到的A的值来修改B上的值，其数据库访问的操作流程如下：**R1(A);R1(B);W1(B);**
    - **事务T2**：根据读到的B的值来修改A上的值，其数据库访问的操作流程如下：**R2(B);R2(A);W2(A);**
  - 如果采用如下的调度来并发执行事务**T1**和**T2**将产生死锁现象：**R1(A);R1(B);R2(B);W1(B);R2(A);W2(A);**



## 7.2.4 死锁的检测、处理和预防

**R1(A);R1(B);R2(B);W1(B);R2(A);W2(A);**

	事务T <sub>1</sub>	事务T <sub>2</sub>	A的加锁状态	B的加锁状态
1	<b>slock<sub>1</sub>(A);</b>		S(T <sub>1</sub> )	
2	r <sub>1</sub> (A);			
3	<b>slock<sub>1</sub>(B);</b>			S(T <sub>1</sub> )
4	r <sub>1</sub> (B);			
5		<b>slock<sub>2</sub>(B);</b>		S(T <sub>1</sub> , T <sub>2</sub> )
6		r <sub>2</sub> (B);		
7	<b>xlock<sub>1</sub>(B);</b>			
8	Wait...	<b>slock<sub>2</sub>(A);</b>	S(T <sub>1</sub> , T <sub>2</sub> )	
9	Wait...	r <sub>2</sub> (A);		
10	Wait...	<b>xlock<sub>2</sub>(A);</b>		
11	Wait...	Wait...		
12	Wait...	Wait...		



## 7.2.4 死锁的检测、处理和预防

### ■ 死锁问题的处理办法

#### ■ 预防法 (“防患于未然” 法)

- 采用一定的锁申请操作方式以避免死锁现象的发生
  - 顺序申请法
  - 一次申请法

#### ■ 解除法 (“亡羊补牢” 法)

- 允许产生死锁，当系统通过死锁检测程序发现出现死锁现象时，可以调用解锁程序来解除死锁
  - 超时死锁检测法
    - 事务的执行时间超时
    - 锁申请的等待时间超时
  - 等待图法
  - 时间戳死锁检测法



## 7.2.4 死锁的检测、处理和预防

### ■ 死锁的检测和处理

#### ■ 检测方法一：超时法

- 规定一个“等待时限”，如果一个事务因等待获得锁而超过“等待时限”，则认为其死锁了。
- 优点：
  - 简单。
- 缺点：
  - 久等：早已发生死锁，非要到时限才能发现；
  - 误判：尚未死锁，再等一会即可推进，但时限到了，认为是死锁了。



## 7.2.4 死锁的检测、处理和预防

### ■ 死锁的检测和处理(cont.)

#### ■ 检测方法二：等待图法

- 使用一个有向图称“等待图”（Wait-for Graph）（OS中已学过）周期性地检测等待图中是否有回路，若有，则发生死锁了。

#### ■ 优点：

- 从不会“误判”，总能检测出来。

#### ■ 缺点：

- 久等：当检测周期较长时。
- 检测代价大：当检测周期较短时，频繁检测。



## 7.2.4 死锁的检测、处理和预防

### ■ 死锁的检测和处理(cont.)

#### ■ 处理方法

- 在循环等待的事务中，选一个事务作为牺牲品 (Victim) 让路
- 撤消牺牲者并在屏幕上通知用户，事后由用户重新提交；
- 暂时撤消，稍后由DBMS重启该事务再试。
- **选择牺牲品原则**：最迟交付事务；获得锁较少事务；撤消代价最小事务。





## 7.2.4 死锁的检测、处理和预防

### ■ 死锁的预防

#### ■ OS中防止进程死锁的思路：

- 一次申请所有锁
  - 规定一个封锁次序
- } 防止循环等待

#### ■ 但是这在DB中不太实际。

#### ■ DB中较实际的方法

- 选择性地“卷回重执” (Rollback and Retry)事务。



## 7.2.4 死锁的检测、处理和预防

### 死锁的预防(cont.)

- 每个事务被DBMS接纳时被赋一个“时间标记”(Time Stamp,TS)。
- 若 $ts(TA) < ts(TB)$ , 则TA是“年老”者, TB是“年轻”者。
- 设TB已持有某对象的锁, 当TA申请这一对象的锁而发生冲突时, 有如下策略:
  - 等待一死亡 (Wait-Die) 策略 (若老则等, 若幼则死)  
if  $ts(TA) < ts(TB)$  then TA waits;      /\* 老者等待 \*/  
else { Rollback TA;      /\* 幼者死亡 \*/  
Restart TA with the same  $ts(TA)$ ;      /\* 重执 \*/  
};
  - 年老者等待下去; 年轻者死亡, 稍后重执(其总会变得年老, 而不会总死亡)



## 7.2.4 死锁的检测、处理和预防

### ■ 死锁的预防(cont.)

- 击伤一等待 (Wound-Wait) 策略 (若幼则等, 若老则抢)

if  $ts(TA) > ts(TB)$  then TA waits;      /\* 幼者等待 \*/

else { Rollback TB;      /\* 幼者击伤 \*/

Restart TB with the same  $ts(TB)$ ;      /\* 重执

\*/ };

- 年轻者被年老者击伤后重执而等待 (总会变得年老, 而不会总被击伤)

- 以上策略的共性是: 年轻事务作为牺牲品, 卷回而重执。从而避免了循环等待!

- ——显然比一冲突就卷回这种“一触即退”方案要好些!



# 小结

---

- 数据共享与数据一致性是一对矛盾
- 数据库的价值在很大程度上取决于它所能提供的数据共享度。
- 数据共享在很大程度上取决于系统允许对数据并发操作的程度。
- 数据并发程度又取决于数据库中的并发控制机制
- 另一方面，数据的一致性也取决于并发控制的程度。施加的并发控制愈多，数据的一致性往往愈好。



# 小结（续）

- 数据库的并发控制以事务为单位
- 数据库的并发控制通常使用加锁机制
  - 两类最常用的锁
- 不同级别的加锁协议提供不同的数据一致性保证，提供不同的数据共享度。
  - 三种封锁协议



# 小结（续）

- 并发控制机制调度并发事务操作是否正确的判别准则是可串行性
  - 并发操作的正确性则通常由两段锁协议来保证。
  - 两段锁协议是可串行化调度的充分条件，但不是必要条件



# 小结（续）

- 对数据对象施加封锁，带来问题
- 活锁：先来先服务
- 死锁：
  - 预防方法
    - 一次封锁法
    - 顺序封锁法
  - 死锁的诊断与解除
    - 超时法
    - 等待图法

