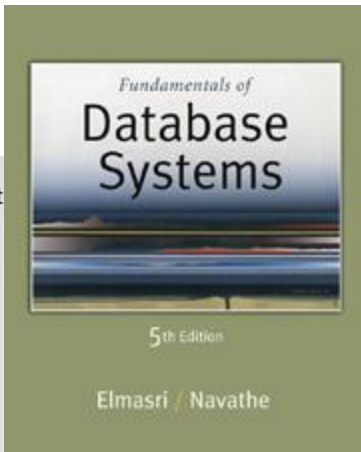
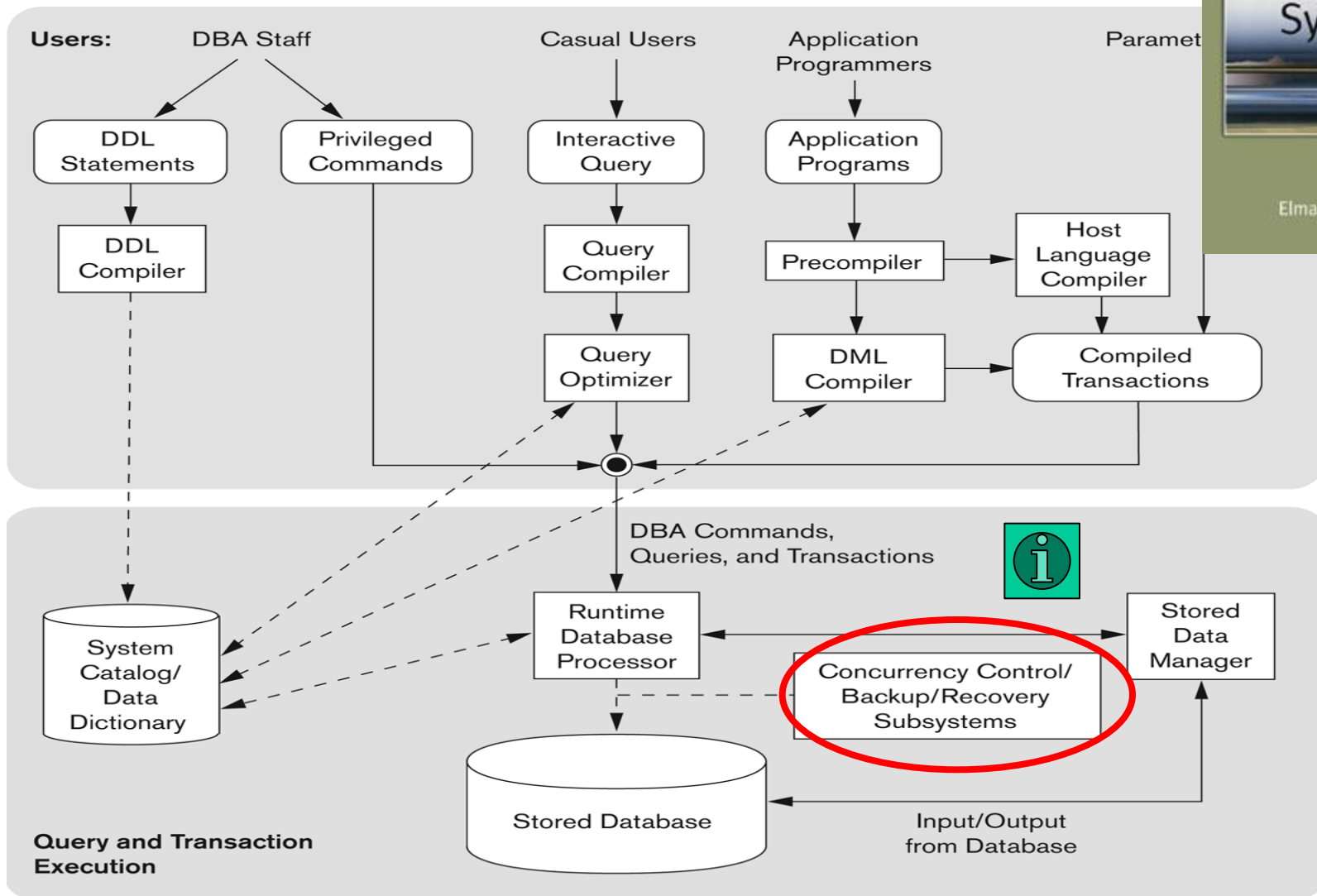


# DBMS体系结构



**Figure 2.3** 1

Component modules of a DBMS and their interactions.

# 第七章 事务处理技术

---

- 事务的概念
- 数据库恢复技术
- 并发控制技术



**James Nicholas Gray**

(January 12, 1944 – January 28, 2007? )

# 事务的定义

---

- **事务(Transaction)**是用户定义的数据库操作序列，这些操作要么都做，要么都不做，是一个不可分割的工作单位。
  - **事务与应用程序**是两个概念，一般来说，一个应用程序可以包含多个事务。
  - **事务的开始与结束可以由用户显式控制**。如果用户没有显式定义事务，则由DBMS按缺省规定自动划分事务。

# 示例

---

例：银行转帐：事务T从A帐户过户50元到B帐户。

read(X)：从数据库传送数据项X到事务的工作区中。

write(X)：从事务的工作区中将数据项X写回数据库。

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

# 事务的特性 (1)

---

- 原子性(Atomicity)
  - 事务中包括的所有操作要么都做，要么都不做
- 一致性(Consistency)
  - 事务执行的结果必须是使数据库从一个一致性状态，变到另一个一致性的状态

# 事务的特性 (2)

---

- 隔离性(Isolation)

- 一个事务的执行不能被其它事务干扰。即一个事务内部的操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能互相干扰

- 持久性(Durability)

- 一个事务一旦提交之后，它对数据库的影响必须是永久的。其它操作或故障不应该对其执行结果有任何影响

# 事务的特性 (3)

---

- 事务的**ACID**特性对于数据库数据的正确、有效具有重要意义。但事务的特性有可能遭破坏，主要有两种情况：
  - 多个事务并行运行时，不同事务的操作交叉进行；
  - 事务在运行过程中被强行停止。

# 事务的特性 (4)

---

- 利用数据库并发控制机制以及数据库恢复机制保证事务的特性不被破坏，从而保证数据库数据的正确、有效
  - 原子性由恢复机制实现
  - 一致性是由事务的原子性保证的
  - 隔离性通过并发控制机制实现
  - 持久性通过恢复机制实现
- 事务是数据库恢复和并发控制的基本单位

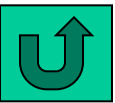


# SQL中事务的定义

---

- 在SQL中定义事务的语句有三条：

事务开始	BEGIN TRANSACTION	
事务结束	提交事务，正常结束。	COMMIT
	撤消全部更新，回滚到事务开始时状态。非正常结束。	ROLLBACK



# 数据库恢复技术

---

- 什么是数据库恢复
- 故障的种类
- 恢复的实现技术
- 恢复的策略
- 具有检查点的恢复技术
- 数据库镜像

# 数据库恢复技术

---

- 什么是数据库恢复技术
  - 数据库管理系统必须具有把数据库从错误状态恢复到某一已知正确状态的功能，这就是数据库的恢复。
  - 数据库恢复是通过数据库管理系统的恢复子系统完成的。
- 数据库恢复子系统的意义
  - 保证事务的原子性。实现事务非正常终止时的回滚。
  - 当系统发生故障以后，数据库能够恢复到正确状态。



# 数据库系统中故障的种类 (1)

---

- 事务内部的故障

- 可预期的：事务根据内部的测试条件，确定是否回滚。
- 不可预期的：指不能由应用程序处理的事务故障，如死锁，运算溢出，违反完整性规则等。

- 系统故障

- 是指造成系统停止运行的任何事情，使得系统要重新启动。如硬件错误，操作系统故障，停电等。
- 这类故障打断所有正在运行的事务，使事务都异常中止，但不会破坏数据库。



# 数据库系统中故障的种类 (2)

---

- 介质故障

- 介质故障指外存故障，如磁盘损坏，瞬时强磁场干扰等
- 这类故障将破坏全部或部分数据库，并影响正在存取这部分数据的所有事务。

- 计算机病毒

- 计算机病毒是一种人为的破坏或故障，已成为数据库系统的主要威胁之一。
- 多数病毒对数据进行非法修改。



# 恢复的实现技术

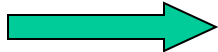
---

- 故障对数据库系统的两种影响
  - 数据库本身被破坏，如介质故障
  - 数据库没有破坏，但数据可能不正确。
    - 如事务故障、系统故障，由于事务的非法终止造成；
    - 计算机病毒
- 数据库恢复的原理
  - 数据库恢复的基本原理为冗余。即利用存储在系统别处的冗余数据来重建或恢复修正数据库。

# 恢复的实现技术

---

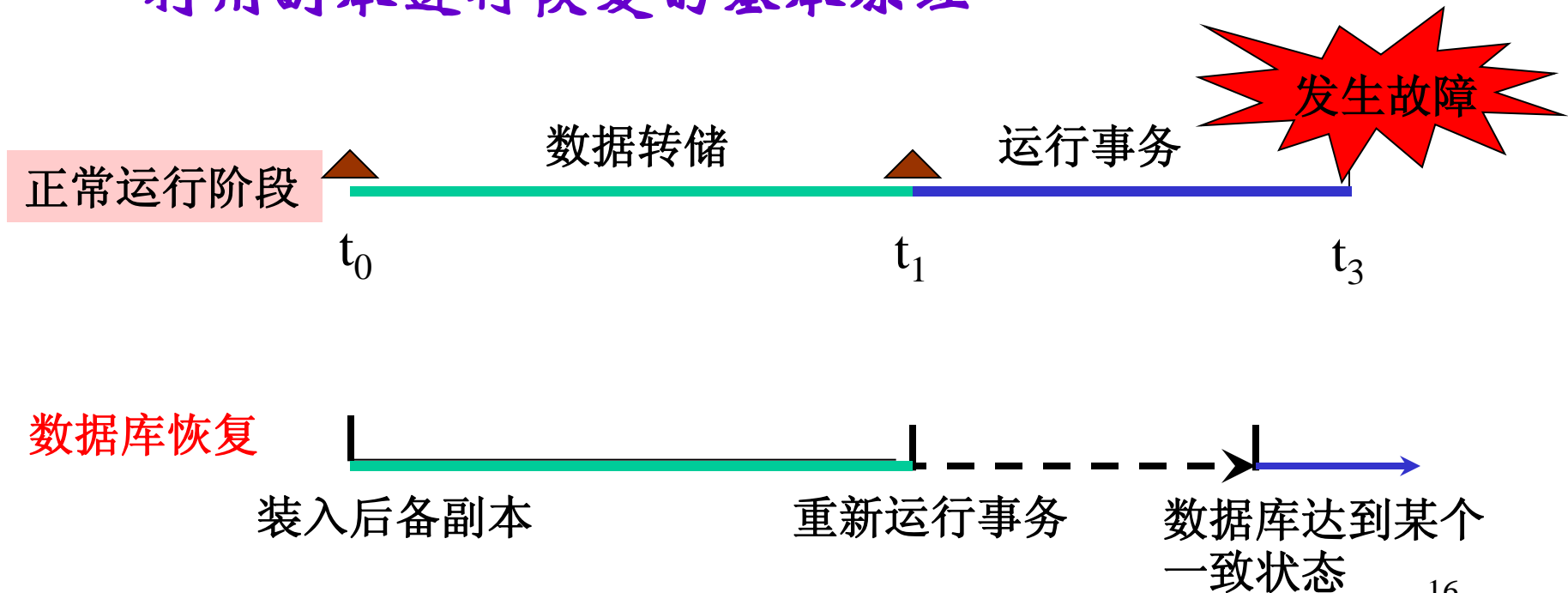
- 数据库恢复的关键问题

- 如何建立冗余       数据转储与登录日志文件

- 如何利用冗余实施数据库恢复

# 数据转储 (1)

- 是DBA定期地将整个数据库复制到磁带或另一个磁盘上保存起来的过程。这些备用的数据文本称为**后备副本**或**后援副本**。
- 利用副本进行恢复的基本原理





# 数据转储 (2)

---

- 数据转储分为两种转储状态
  - **静态转储**：系统中无事务运行时进行的转储操作。并且转储过程中，不允许对数据库进行任何存取、修改。
    - 优点：保证副本的数据一致性；
    - 缺点：由于转储必须等待正在运行的事务结束才能开始，而新的事务必须等待转储结束才能执行，降低了数据库的可用性。
  - **动态转储**：转储期间允许对数据库进行存取或修改。
    - 优点：不影响数据库的可用性
    - 缺点：不能保证副本上的数据正确、有效。还必须把转储期间各事务对数据库的修改记录下来，建立日志文件。**后援副本加上日志文件**就能把数据库恢复到某一时刻的正确状态。

---

- 数据转储可有两种转储方式

- 海量转储

- 海量转储指每次转储全部数据库。

- 增量转储

- 增量转储指每次只转储上一次转储后更新过的数据。

# 日志文件的建立与使用

---

- 日志文件格式与内容

日志是用来记录事务对数据库更新操作的文件。日志文件主要有两种格式：以记录为单位和以数据块为单位

- 记录为单位的日志文件

- 记载的内容

- 各个事务的开始标记
      - 各个事务的结束标记
      - 各个事务的所有更新操作

- 每个日志记录中包含的信息项

- 事务标识（标明是哪个事务）
      - 操作的类型（插入、删除或修改）
      - 操作对象（记录的内部标识）
      - 更新前数据的旧值（对插入操作，此项为空）
      - 更新后数据的新值（对删除操作，此项为空）

- 以数据块为单位的日志文件，日志记录的内容包括事务标识以及更新前和更新后的数据块。

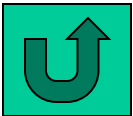
---

- 日志文件的作用

- 事务故障和系统故障恢复必须使用日志文件。
- 在动态转储方式中必须建立日志文件，后备副本和日志文件综合起来才能有效地恢复数据库。
- 在静态转储方式中，用日志文件恢复转储结束时刻到故障点间的事务。

- 日志文件的写入规则

- 登记的次序严格按并发事务执行的时间顺序；
- 必须先写日志文件，后写数据库。



# 故障的恢复策略

---

- 事务故障的恢复—UNDO，即撤消事务在不影响其它事务的情况下，强行回滚，撤消已做的修改。具体步骤：
  - 反向扫描日志文件，查找该事务的更新操作；
  - 对该事务的更新操作（插入、删除、修改）执行逆操作，即将日志记录中的“更新前的值”写入数据库；
  - 如此处理下去，直到读到该事务的开始标志。

---

## • 系统故障——UNDO+REDO

- 系统故障造成数据库不一致状态的原因有两个：
  - 一是未完成的事务对数据库的更新可能已经写入数据库；
  - 二是已提交事务对数据库的更新可能还留在缓冲区未写入数据库。
- 因此恢复操作就是要撤销(UNDO)故障发生时未完成的事务，重做(REDO)已完成的事务。



---

- 系统故障恢复具体步骤:

- 正向扫描日志文件，找出故障发生前已经提交的事务，将其事务标识记入重做 (REDO) 队列。同时找出故障发生时尚未完成的事务，将其事务标识记入撤销 (UNDO) 队列；
- 对撤销队列中的各个事务进行UNDO处理；
- 对重做队列中的各个事务进行REDO处理。

---

- 介质故障的恢复，恢复的方法：

- 装入最新的数据库后备副本，使数据库恢复到最近一次转储时的一致状态。对于动态转储的副本，还需要装入转储开始时刻的日志文件副本，将数据库恢复到一致状态；
- 装入转储以后的日志文件副本，重做已经完成的事务。





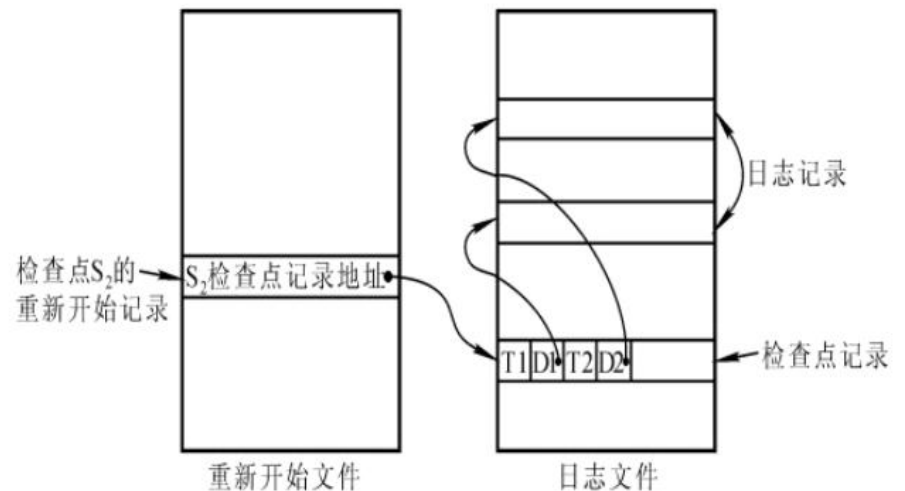
# 具有检查点的恢复技术

---

- 利用日志技术进行恢复时，恢复子系统通常**需要检查大量日志记录**，存在的问题是：
  - 搜索日志耗费大量时间
  - 不必要重做某些事务
- **检查点技术**可以改善效率，使得在检查点之前提交的事务，在数据库恢复处理时不必重做。

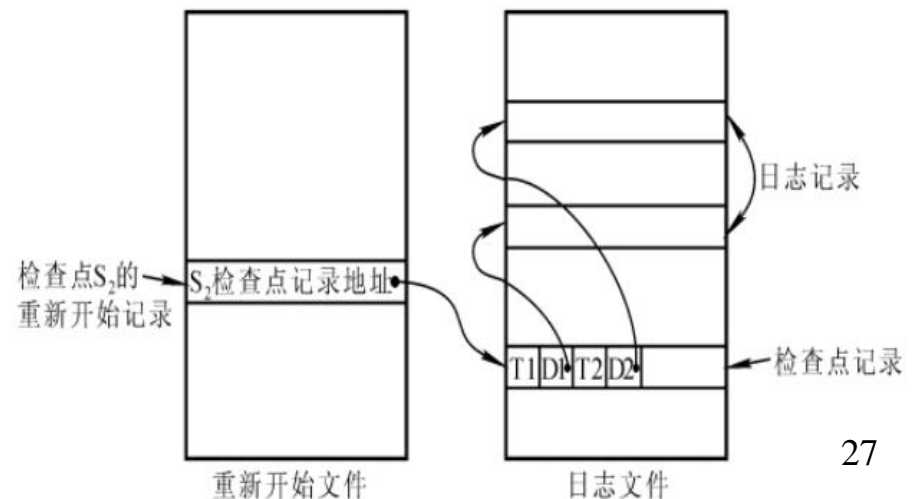
# 检查点技术 (1)

- 在日志文件中增加检查点 (checkpoint) 记录
- 检查点记录的内容包括：
  - 建立检查点时刻所有正在执行的事务清单
  - 这些事务最近一个日志记录的地址
- 系统中增加一个重新开始文件，用来记录各个检查点记录在日志文件中的地址



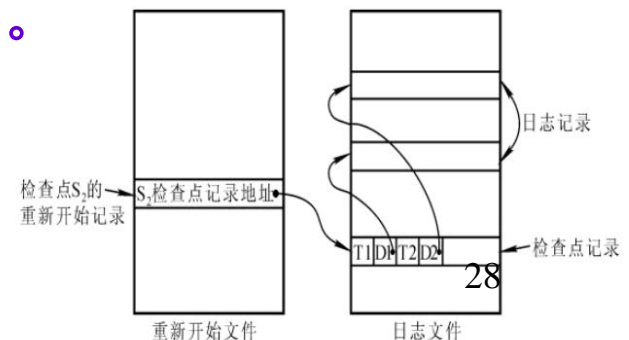
# 检查点技术 (2)

- 恢复子系统动态维护日志文件，即周期性地执行如下操作：
  - 将当前日志缓存中的所有日志记录写入磁盘的日志文件上
  - 在日志文件上写入一个检查点记录
  - 将当前数据缓存的所有数据记录写入磁盘的数据库中
  - 把检查点记录在日志文件中地址写入重新开始文件

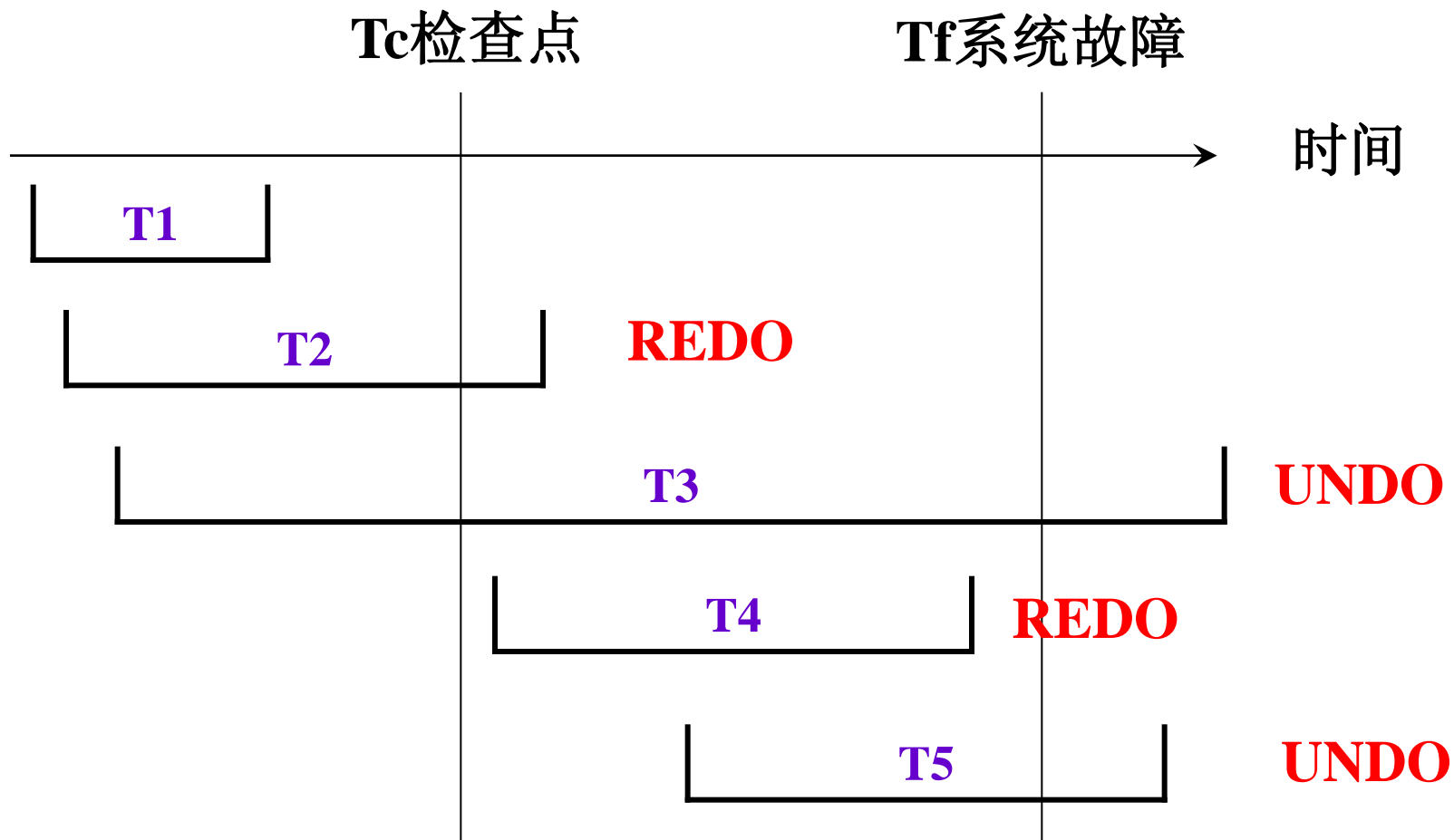


# 利用检查点技术进行恢复

- 利用重新开始文件定位最近检查点记录：在重新开始文件中找到最后一个检查点记录在日志文件中的地址。
- 找到检查点时刻运行事务清单：由该检查点记录得到检查点建立时刻所有正在运行的事务清单ACTIVE-LIST，把ACTIVE-LIST暂时放入UNDO-LIST。
- 确定需要撤消和重做的事务：从检查点开始正向扫描日志文件，做如下处理，直到文件结束。
  - 如果有新开始的事务 $T_i$ ，把 $T_i$ 暂时放入UNDO-LIST；
  - 如果有提交的事务 $T_j$ ，把 $T_j$ 从UNDO-LIST队列移入到REDO-LIST队列；
- 执行撤消或重做动作：对UNDO-LIST中的每一个事务执行UNDO操作，对REDO-LIST中的每个事务执行REDO操作。

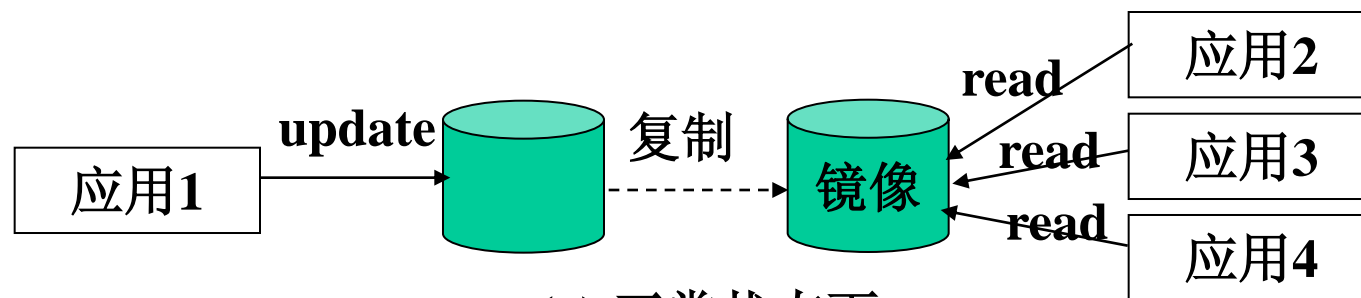


# 示例

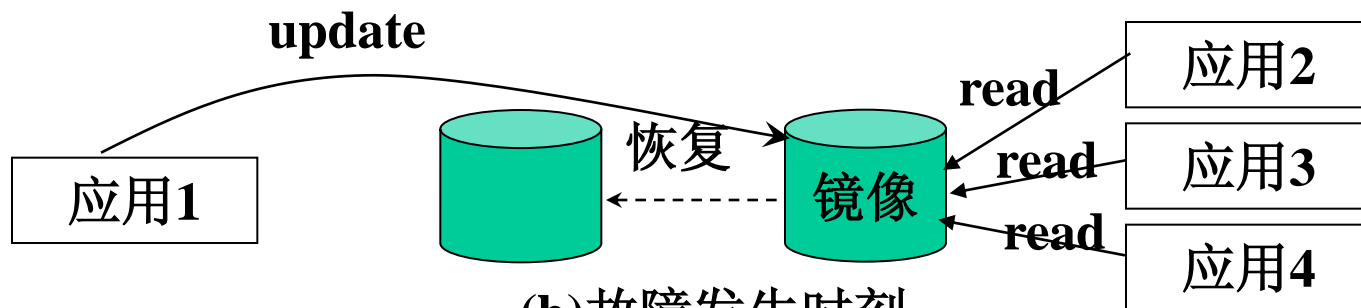


# 数据库镜像

- 根据DBA的要求，自动把整个DB或其中的关键数据复制到另一个磁盘上，由DBMS自动保证镜像数据库与主数据库的一致性。



(a) 正常状态下



(b) 故障发生时刻



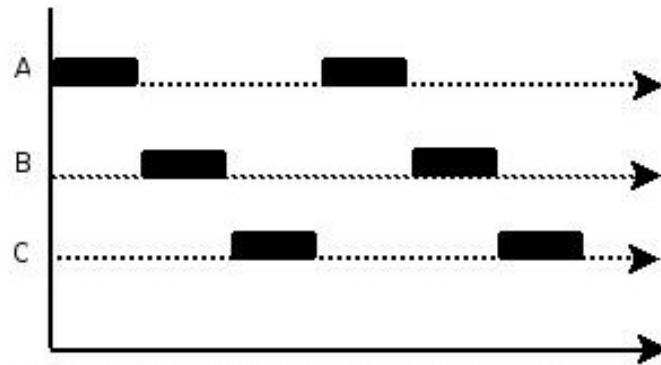
# 并发控制

---

- 为什么需要并发控制
- 并发控制的主要方法
  - 两种基本类型封锁
  - 封锁协议
  - 多粒度封锁
- 活锁与死锁
- 事务的可串行化调度

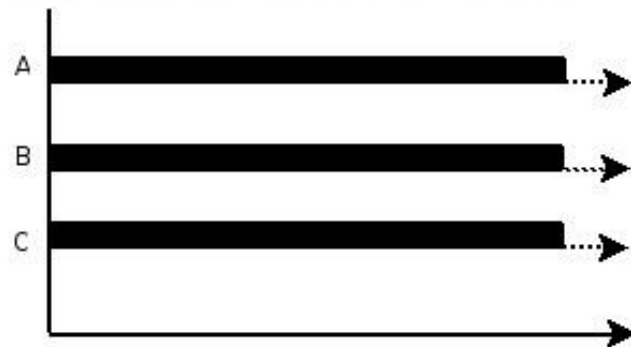
# 并发与并行

- 并发:



Concurrency : 1. Single Processor  
2. logically simultaneous processing

- 并行:



Parallelism : 1. Multiprocessores, Multicore  
2. Physically simultaneous processing



# 并发控制

---

- 事务并发执行的优点

- 一个事务由不同的步骤组成，所涉及的系统资源也不同。这些步骤可以并发执行，以提高系统的吞吐量。
- 系统中存在着周期不等的各种事务，串行会导致难以预测的时延。如果各个事务所涉及的是数据库的不同部分，采用并发会减少平均响应时间。

- 事务并发执行带来的问题

- 多个事务同时存取同一数据时，如不加控制就可能会读取或存储不正确的数据，破坏数据库的一致性。

# 并发控制的必要性

- 例：假设存款余额 $X=1000$ 元，事务甲取走300元，事务乙取走200元，存款余额最终更新后应该是 $X=500$ 元。

时间	事务甲	事务乙
...	...	...
$t_1$	读 $X$ (1000)	
...	...	
$t_2$		读 $X$ (1000)
...		...
$t_3$	更新： $X=X-300$ (700)	...
...	...	更新： $X=X-200$ (800)



!错误

# 并发操作导致的数据不一致性(1)

---

- 丢失更新(Lost Update)

- 两个事务 $T_1$ 和 $T_2$ 读入同一数据并修改， $T_2$ 提交的结果破坏了 $T_1$ 提交的结果，导致 $T_1$ 的修改被丢失。

$T_1$	$T_2$
READ(A) //A=16	
<b>A=A-1</b>	READ(A) //A=16
<b>WRITE(A) //A=15</b>	
	<b>A=A-1</b>
	<b>WRITE(A) //A=15</b>

# 并发操作导致的数据不一致性(2)

- “脏”数据的读出(Dirty Read)

- 事务 $T_1$ 修改某一数据，并将其写回磁盘，事务 $T_2$ 读取同一数据后， $T_1$ 由于某种原因被撤销，这时 $T_1$ 已修改过的数据恢复为原值， $T_2$ 读到的数据就与数据库中的不一致，则 $T_2$ 读到的数据就为“脏”数据。

$T_1$	$T_2$
READ(C) //C=100	
C=C*2	
WRITE (C)	
	READ(C) //C=200
ROLLBACK	
//C=100	

# 并发操作导致的数据不一致性(3)

- 不能重复读(Non-Repeatable Read)

- 事务 $T_1$ 读取数据后，事务 $T_2$ 执行更新（修改、插入、删除）操作，使 $T_1$ 无法再现前一次读取的结果。

$T_1$	$T_2$
READ(A) READ(B) Sum=A+B	
	READ(B) B=B*2 WRITE(B)
READ(A) READ(B) Sum=A+B	

( a )

$T_1$	$T_2$
select * from SC where CNO = C01 and SNO = S01	
	Delete From SC Where CNO =C01 and SNO =S01
select * from SC where CNO = C01 and SNO = S01	

(b)

# 事务并发操作中可能存在的问题

---

- 丢失更新(Lost Update)
- “脏”数据的读出(Dirty Read)
- 不能重复读(Non-Repeatable Read)



# 并发控制基本思想

---

- 并发控制就是要合理调度并发事务，避免并发事务之间的互相干扰造成数据的不一致性。
- 并发控制的主要方法是采用封锁机制。

# 并发控制的基本手段——封锁

---

- **封锁**就是事务T在对某个数据对象如表、记录等操作之前，先向系统发出请求，对其**加锁**，从而对该数据对象有了一定的控制权。
- 基本的封锁有两种类型：
  - 排它锁 (**X锁**，e**X**clusive lock)
  - 共享锁 (**S锁**，Share lock)



# 封锁的类型

---

- **排它锁 (X锁)**：事务T对数据对象R加上X锁，则只允许T读取和修改R，其它事务对R的任何封锁请求都不能成功，直至T释放R上的X锁。
- **共享锁 (S锁)**：事务T对数据对象R加上S锁，则事务T可以读取但不能修改R，其它事务只能对R加S锁，而不能对R加X锁，直到T释放R上的S锁。

# 基本锁的相容矩阵

---

- 以相容矩阵表示两种基本锁的控制：

$T_1 \backslash T_2$	X	S
X	N	N
S	N	Y

# 封锁协议(1)

---

- 运用X锁和S锁这两种基本封锁，可以建立不同的约定，形成不同级别的封锁协议，以保证事务并发执行过程中的数据的一致性。
  - 一级封锁协议——防止丢失修改
  - 二级封锁协议——防止读“脏”数据
  - 三级封锁协议——保证数据可重复读

# 封锁协议(2)

- 一级封锁协议

- 协议内容：事务T在修改数据R之前必须对其加X锁，直到事务结束才释放。事务结束包括正常结束(COMMIT)和非正常结束(ROLLBACK)。

T <sub>1</sub>	T <sub>2</sub>
X LOCK A	
...	
A=A-1	X LOCK A
...	等待
COMMIT	·
UNLOCK A	·
	获得

一级封锁协议可以防止丢失修改，并保证事务T是可恢复的。但在一级封锁协议中，不要求读数据操作对数据加锁，因此它不能保证可重复读和不读“脏”数据。

# 封锁协议(3)

## • 二级封锁协议

- 协议内容：一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，读完后即可释放S锁。

T <sub>1</sub>	T <sub>2</sub>
X LOCK C READ(C) //C=100 C=C*2 ... ROLLBACK UNLOCK C ...	S LOCK C 等待 ... READ(C) //C=100 UNLOCK(C) ... COMMIT

二级封锁除了防止丢失更新，还可以进一步防止读“脏”数据。但由于读完后即可释放S锁，所以不能保证可重复读。

# 封锁协议(4)

- 三级封锁协议

- 协议内容：一级封锁协议加上事务T在读取R之前必须对其加S锁，直到事务结束才释放。

T <sub>1</sub>	T <sub>2</sub>
S LOCK A	
S LOCK B	
...	
READ(A) //A= 50	X LOCK B
READ(B)// B= 100	等待
Sum=A+B //Sum=150	.
...	.

T <sub>1</sub>	T <sub>2</sub>
READ(A)	等待
READ(B)	.
Sum=A+B //Sum=150	.
COMMIT	.
UNLOCK(A)	.
UNLOCK(B)	.
	获得
	WRITE(B)

- 三级封锁协议除了防止丢失修改和读“脏”数据以外，还进一步防止了不可重复读。

# 封锁协议小结

- 三级协议的主要区别在于何时加锁以及何时放锁。

	X锁		S锁		一致性保证		
	操作结束释放	事务结束释放	操作结束释放	事务结束释放	不丢失修改	不读“脏”数据	可重复读
一级封锁协议		√			√		
二级封锁协议		√	√		√	√	
三级封锁协议		√		√	√	√	√

# 封锁的粒度

---

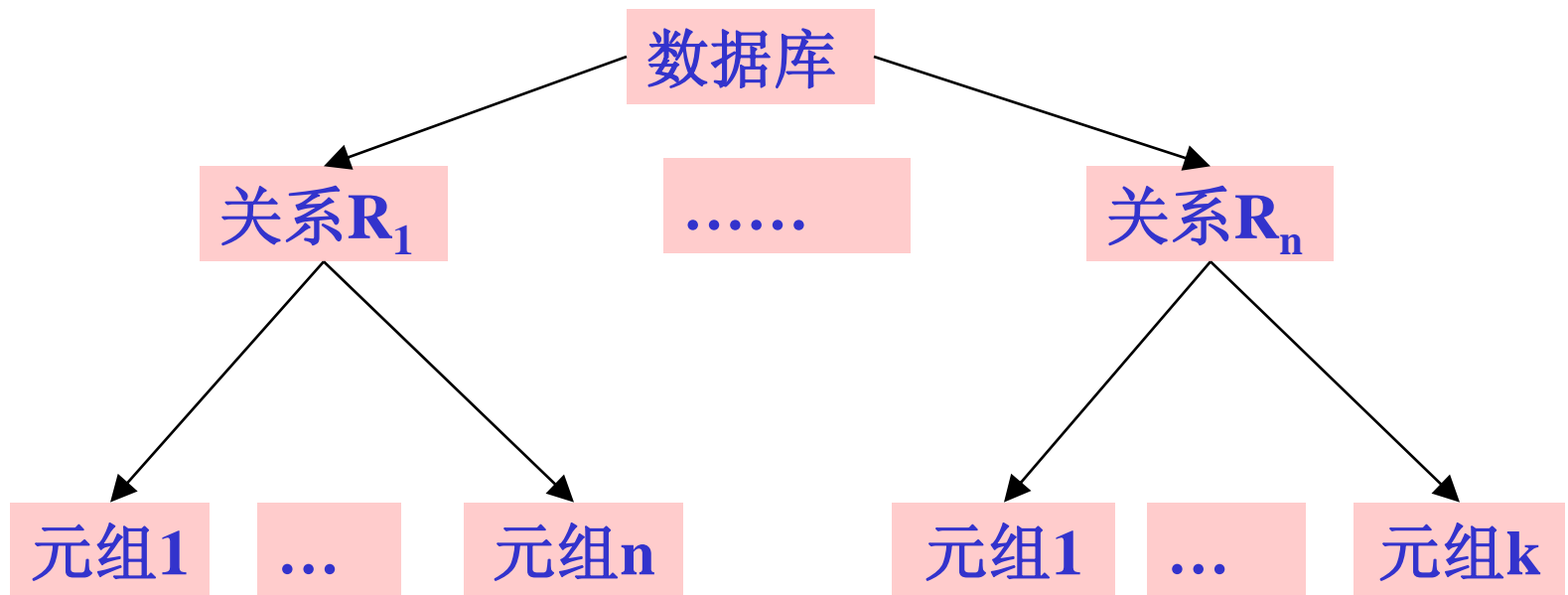
- 封锁对象的大小称为封锁粒度。
- 封锁对象：属性值、属性值集合、元组、关系、某索引项、整个索引、整个数据库、物理页、块等。
- 封锁粒度大，则并发度低，封锁机构简单，开销小。  
封锁粒度小，则并发度高，封锁机构复杂，开销高。
- 多粒度封锁：在一个系统中同时支持多种封锁粒度供不同的事务选择。选择封锁粒度时应同时考虑封锁开销和并发度两个因素，适当选择封锁粒度以达到最优效果。



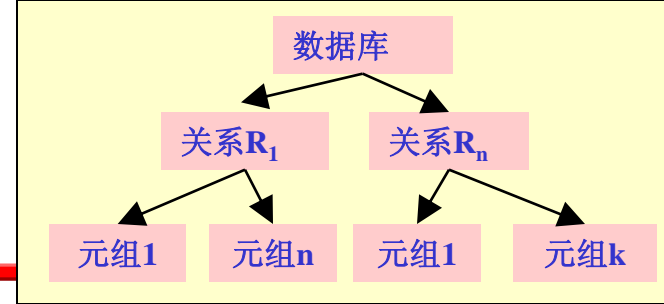
# 多粒度封锁 (1)

- 多粒度树

- 多粒度树的根结点是整个数据库，表示最大的粒度。叶结点表示最小的粒度。



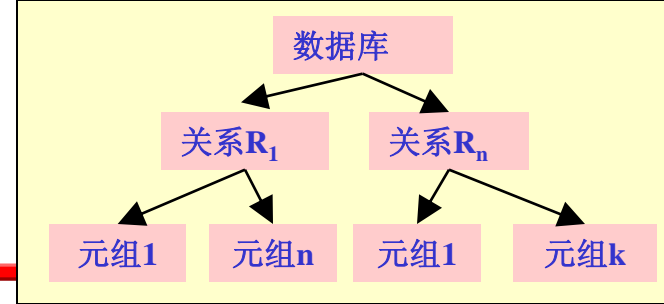
# 多粒度封锁 (2)



- 多粒度封锁协议

- 多粒度封锁协议允许多粒度树中的每个结点被独立地加锁。对一个结点加锁意味着这个结点的所有后裔结点也被加以同样类型的锁。因此，在多粒度封锁中一个数据对象可能以两种方式封锁，即：
  - 显式封锁是应事务的要求直接加到数据对象上的封锁。
  - 隐式封锁是该数据对象没有独立加锁，是由于其上级结点加锁而使该数据对象加上了锁。
- 在多粒度封锁方法中，显式封锁与隐式封锁的效果是一样的。

# 多粒度封锁 (3)

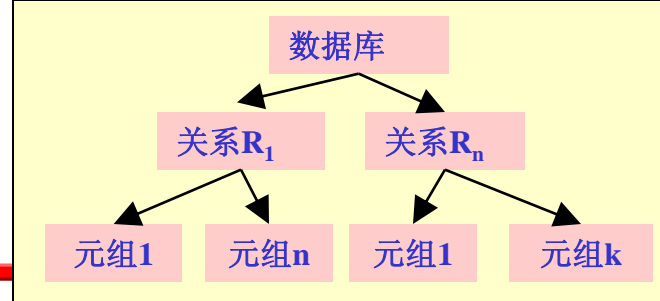


- 多粒度封锁中存在的问题

- 在多粒度封锁方法中，一般对某个数据对象加锁，系统要做如下检查：

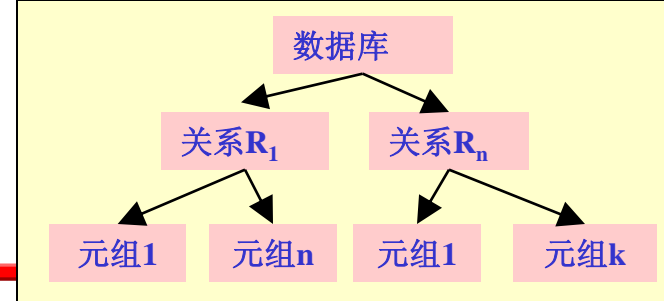
- 是否与该数据对象上的显式封锁冲突（检查对象本身）；
- 是否与该数据对象上的隐式封锁冲突（检查对象的所有上级结点）；
- 是否与该数据对象下级的显式封锁冲突（检查其所有下级结点）。

# 意向锁 (1)



- **意向锁**的含义是该结点的下层结点正在被加锁
- 对任意节点加锁时，必须先对其上级节点加意向锁
- 意向锁的好处是：在对象加锁时，**不再检查下级结点的封锁**，只需检查对象和它的上级结点

# 意向锁 (2)



- 三种常用的意向锁

- 意向共享锁 (Intent Share Lock, 简称IS锁)

- 如果要对一个数据对象加IS锁, 表示它的后裔结点拟 (意向) 加S锁。

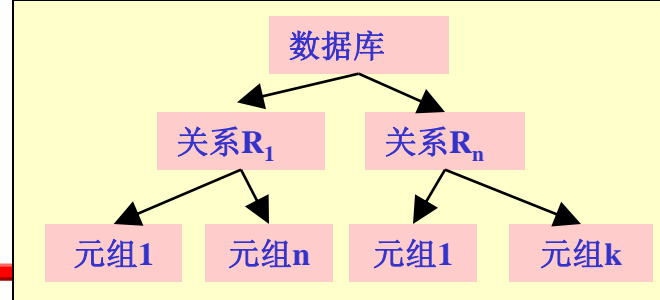
- 意向排它锁 (Intent Exclusive Lock, 简称IX锁)

- 如果要对一个数据对象加IX锁, 表示它的后裔结点拟 (意向) 加X锁。

- 意向共享排它锁 (Share Intent Exclusive Lock, 简称SIX锁)

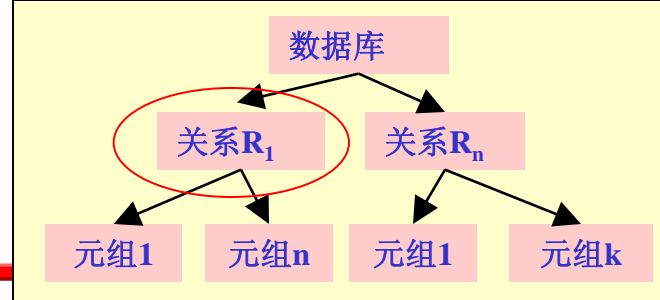
- 如果要对一个数据对象加SIX锁, 表示对它加S锁, 再加IX锁, 即 $SIX=S+IX$ 。

# 意向锁 (3)



- 具有意向锁的多粒度封锁方法中，任意事务T要对一个数据对象加锁，要先对它的上级对象结点加意向锁，申请封锁按自上而下的次序进行；释放封锁时，应按自下而上的次序进行。

# 意向锁 (4)



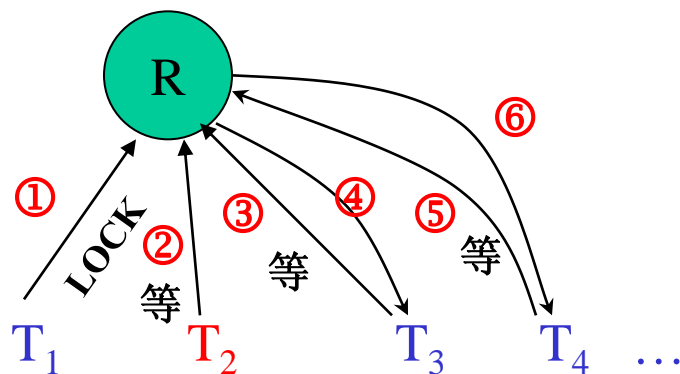
## • 意向锁的相容矩阵

$T_1 \backslash T_2$	S	X	IS	IX	SIX	—
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
—	Y	Y	Y	Y	Y	Y



# 活锁与死锁 (1)

- 活锁



避免活锁的简单方法是采用先来先服务的策略。

$T_2$  有可能永远等待，这就是活锁



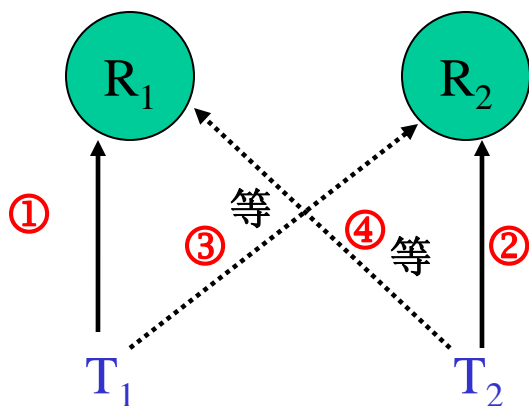
# 活锁与死锁 (2)

- 死锁

## 解决死锁的方法

- 预防死锁

- 死锁检测和解除



$T_1$ ,  $T_2$  永远不能结束,  
形成死锁。

# 活锁与死锁 (3)

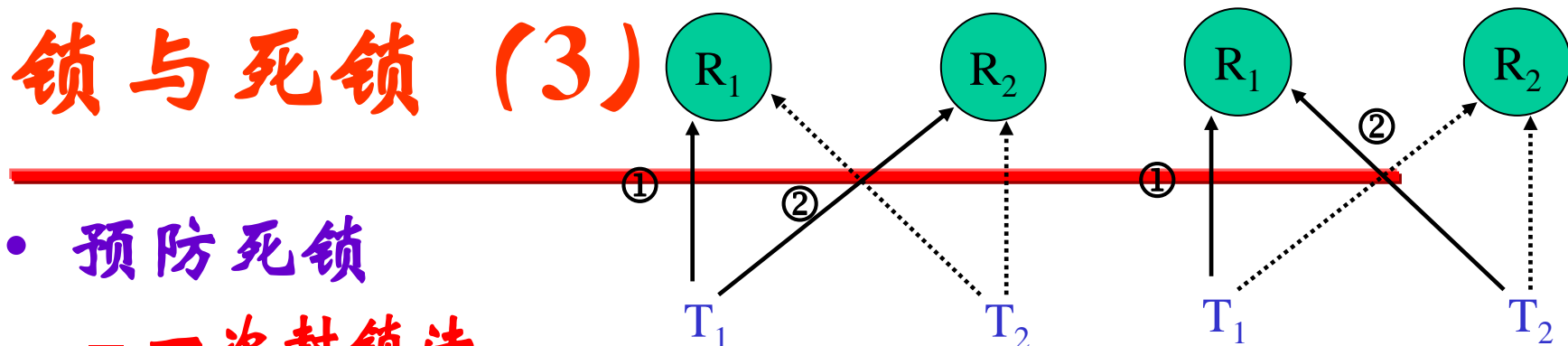
- 预防死锁

- 一次封锁法

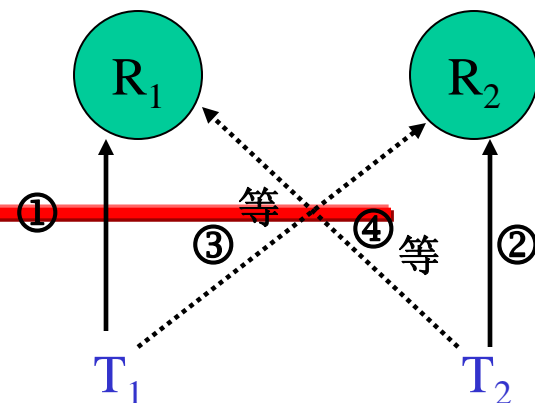
- 要求每个事务必须一次将其所有要使用的数据全部加锁，否则就不能执行。
    - 可以有效地防止死锁的发生，但由于需要扩大加锁的范围，因此降低了系统的并发度。

- 顺序封锁法

- 预先对数据对象规定一个封锁顺序，所有的事务都要按照这个顺序实行封锁。
    - 可以有效地防止死锁，但由于数据库中数据的不断变化和事务封锁要求的动态提出而实现难度大。



# 活锁与死锁 (4)



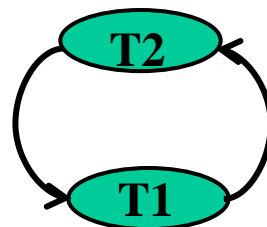
## • 死锁的检测

### — 超时法

- 如果一个事务的等待时间超过了规定的期限，就认为发生了死锁。

### — 等待图法

- 事务等待图是一个有向图 $G=(T, U)$ ,其中 $T$ 为结点集合,每个结点表示正在运行的事务, $U$ 为边集,每条边表示事务的等待情况。并发控制子系统周期性的检测事务等待图,如果发现图中存在回路,则表示系统出现死锁。



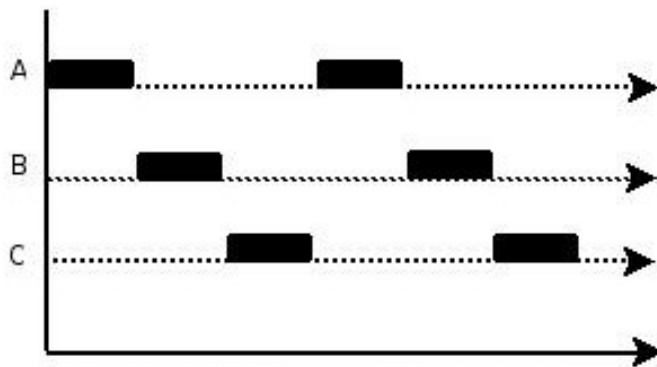
## • 死锁恢复

- 通常采用的方法是选择一个处理死锁代价最小的事务,将其撤销,释放此事务持有的所有锁,使其他事务得以继续运行下去。对于所撤销的事务所作的操作必须加以恢复。



# 事务的调度

- **N个事务的一个调度S**是N个事务所有操作的一个序列S，表示这些操作的执行顺序。并且这个序列满足：对于每个事务T，如果操作*i*在事务T中先于操作*k*执行，则在S中操作*i*也必须先于操作*k*执行。



Concurrency : 1. Single Processor  
2. logically simultaneous processing

# 串行调度与并行调度

T <sub>1</sub>	T <sub>2</sub>	T <sub>1</sub>	T <sub>2</sub>	T <sub>1</sub>	T <sub>2</sub>	T <sub>1</sub>	T <sub>2</sub>
Slock B			Slock A	Slock B		Slock B	
Y = R(B) = 2			X = R(A) = 2	Y = R(B) = 2		Y = R(B) = 2	
Unlock B			Unlock A		Slock A	Unlock B	
Xlock A			Xlock B		X = R(A) = 2	Xlock A	
A = Y + 1 = 3			B = X + 1 = 3	Unlock B			Slock A
W(A)			W(B)		Unlock A	A = Y + 1 = 3	等待
Unlock A			Unlock B	Xlock A		W(A)	等待
	Slock A	Slock B		A = Y + 1 = 3		Unlock A	等待
	X = R(A) = 3	Y = R(B) = 3		W(A)			X = R(A) = 3
	Unlock A	Unlock B			Xlock B		Unlock A
	Xlock B	Xlock A			B = X + 1 = 3		Xlock B
	B = X + 1 = 4	A = Y + 1 = 4			W(B)		B = X + 1 = 4
	W(B)	W(A)		Unlock A			W(B)
	Unlock B	Unlock A			Unlock B		Unlock B

(a) 串行调度

(b) 串行调度

(c) 不可串行化的调度

(d) 可串行化的调度

图 11.7 并发事务的不同调度

# 事务并发调度的正确性

---

- 对并发事务中操作的调度是随机的，而不同的调度可能产生不同的结果，但调度要保证事务执行的正确性
- 事务执行正确性的含义
  - 一个事务正常的或者预想的结果是没有其它并行事务干扰时得到的结果。因此一组事务的串行调度策略一定是正确的调度策略。
  - 虽然不同的串行顺序的结果会不同，由于各种结果都将保持数据库数据的一致性，所以都是正确的。

# 并发调度的可串行性

---

- 可串行化

- 多个事务的并发执行是正确的，当且仅当其结果与按某一次序串行执行它们时的结果相同，我们称这种调度策略为可串行化调度。

- 可串行性是并行事务正确性的准则

- 一个给定的并发调度，当且仅当它是可串行化的，才认为是正确调度。

# 可串行化调度的判定

- 可串行化调度判定的充分条件

- 一个调度 $Sc$ 在保证冲突操作次序不变的情况下，可以通过交换两个事务不冲突操作的次序，得到另一个串行调度 $Sc'$ ，则调度 $Sc$ 为冲突可串行化调度。

- 冲突操作

冲突操作是不同事务对同一数据的读-写操作以及写-写操作。表示为：

事务 $T_i$ 读 $x$ ， $T_j$ 写 $x$  ——  $R_i(x)$ 与 $W_j(x)$

事务 $T_i$ 写 $x$ ， $T_j$ 写 $x$  ——  $W_i(x)$ 与 $W_j(x)$

- 操作次序交换的约束条件

不同事务的冲突操作与同一个事务的两个操作不能交换

- 一个冲突可串行化调度，一定是可串行化调度。



# 可串行化调度的判定

- 示例

调度  $Sc1 = R1(A)W1(A)R2(A)W2(A)R1(B)W1(B)R2(B)W2(B)$ , 经交换操作得到  $Sc2 = R1(A)W1(A) R1(B) W1(B)R2(A)W2(A) R2(B)W2(B)$   
 $= T1, T2$

所以  $Sc1$  是可串行化调度。

$Sc1 = R1(A)W1(A)R2(A)W2(A)R1(B)W1(B)R2(B)W2(B) \rightarrow$   
 $R1(A)W1(A)R2(A) R1(B)W2(A) W1(B)R2(B)W2(B) \rightarrow$   
 $R1(A)W1(A) R1(B)R2(A) W2(A) W1(B)R2(B)W2(B) \rightarrow$   
 $R1(A)W1(A) R1(B)R2(A) W1(B)W2(A) R2(B)W2(B) \rightarrow$   
 $Sc2 = R1(A)W1(A) R1(B) W1(B)R2(A) W2(A) R2(B)W2(B)$

# 并发调度的可串行性

- 两段锁协议(Two-phase Locking)可保证并行事务的可串行性
  - 两段锁协议的内容
    - ①在对任何数据进行读、写操作之前，事务首先要获得对该数据的封锁。
    - ②在释放一个封锁之后，事务不再获得任何其它封锁。
  - 两段锁协议的含义
    - 事务分为两个阶段，第一个阶段是获得封锁，也称为扩展阶段；第二个阶段是释放封锁，也称为收缩阶段。
- 如：T<sub>1</sub>的封锁顺序是：
- S LOCK A ... S LOCK B...X LOCK C...UNLOCK A...UNLOCK C
- T<sub>2</sub>的封锁顺序是：
- S LOCK A ... UNLOCK A... S LOCK B... X LOCK C... UNLOCK C... UNLOCK B

# 并发调度的可串行性

---

- **定理：**若所有事务均遵从两段锁协议，则这些事务的所有并发调度都是可串行化的。  
按照这个定理，所有遵守两段锁协议的事务，其并行执行的结果一定是正确的。
- **注意的问题**
  - 事务遵守两段锁协议是可串行化调度的充分条件而不是必要条件。
  - 两段锁协议并不要求事务在执行任何数据库读、写操作之前就一次申请全部封锁，因此遵守两段锁的事务仍可能发生死锁。

# 小 结

---

- 事务的定义以及特性
- 数据库恢复的原理
- 故障的种类以及恢复策略
- 检查点技术
- 并发操作可能导致的问题
- 并发控制的原理以及三级封锁协议
- 多粒度锁概念
- 活锁与死锁的处理
- 可串行化调度的概念及两段锁协议