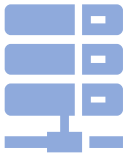
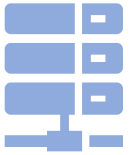
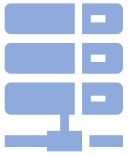


第十章 备份与恢复



备份恢复分类

- 逻辑备份恢复
  - 利用SQL从数据库中提取数据，并将其存入文件中。这些数据可重新导入原来的数据库，也可以导入到其他数据库中
  - 通常使用DBMS提供的导出/导入工具包实现（如MySQLDump）
  - 备份效率低，影响数据库运行性能，数据量大时不适用
  - 常用于小型数据库迁移和数据维护
- 物理备份恢复
  - 基于数据文件（和日志文件）的备份恢复
  - 备份效率高，对数据库运行性能影响小
  - 是保证数据库安全性的重要手段
  - 备份策略复杂
  - 本章主要讨论物理备份恢复的方法

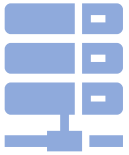


备份与恢复

- 数据库管理系统发生故障是不可避免的
- 备份与恢复策略是应对故障的对策
- 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
- 恢复技术是系统安全性的重要保证
  - 保证事务的原子性和持久性
    - 原子性：事务操作要么全做，要么全不做
    - 持久性：已提交事务对数据库影响是持久的，未提交事务对数据库没有影响

第六章 备份与恢复

1. 故障的种类
2. 恢复的实现技术
3. 恢复的策略



一、事务故障

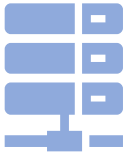
■什么是事务故障

- 某个事务在运行过程中由于种种原因未运行至正常终止点就夭折了

■事务故障的常见原因

- 输入数据有误
- 运算溢出
- 违反了某些完整性限制
- 某些应用程序出错
- 并行事务发生死锁
- 。。。。

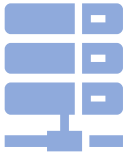
只影响故障事务（程序）本身



一、事务故障

■事务故障的恢复

- 发生事务故障时，夭折的事务可能已把对数据库的部分修改写回磁盘
- 事务故障的恢复
  - 撤消事务（UNDO）
    - ✓ 对未提交事务进行强行回滚（ROLLBACK），清除该事务对数据库的所有修改，消除未提交事务的影响
    - ✓ 保障事务原子性



二、系统故障

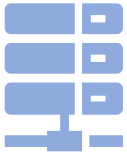
■什么是系统故障

- 整个系统的正常运行突然被破坏
- 所有正在运行的事务都非正常终止
- 内存中数据库缓冲区的信息全部丢失
- 外部存储设备上的数据未受影响

■系统故障的原因

- 操作系统或DBMS代码错误
- 操作员操作失误
- 特定类型的硬件错误（如CPU故障）
- 突然停电

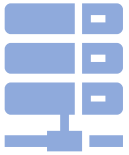
影响到当前正在运行的事务或部分已经运行完结的事务



二、系统故障

■系统故障的恢复

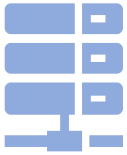
- 清除尚未完成的事务对数据库的所有修改
  - ✓ 系统重新启动时，恢复程序要强行撤消（UNDO）所有未完成事务
- 将缓冲区中已完成事务提交的结果写入数据库
  - ✓ 系统重新启动时，恢复程序需要重做（REDO）所有已提交的事务
- 实现上，需要日志文件的帮助



三、介质故障

- 硬件故障使存储在外存中的数据部分丢失或全部丢失
- 介质故障比前两类故障的可能性小得多，但破坏性大得多
- 硬件故障
  - 磁盘损坏
  - 磁头碰撞
  - 操作系统的某种潜在错误
  - 瞬时强磁场干扰
  - 。。。

影响到内存中的数据和磁盘上的数据



三、介质故障

■介质故障的恢复

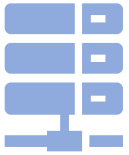
- 装入数据库发生介质故障前某个时刻的数据副本
- 重做自此时始的所有成功事务，将这些事务已提交的结果重新记入数据库
- 实现上，需要数据转储副本（即备份数据）和日志文件的帮助

第六章 备份与恢复

1 故障的种类

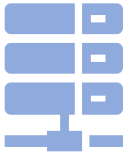
2 恢复的实现技术

3 恢复的策略



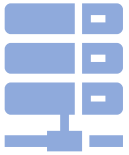
恢复技术

- 恢复操作的基本原理：冗余
  - 利用存储在系统其它地方的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
- 恢复技术涉及的关键问题
  - 1. 如何建立冗余数据
    - ✓ 数据转储（backup）---或称数据备份
    - ✓ 登录日志文件（logging）
  - 2. 如何利用这些冗余数据实施数据库恢复



恢复技术

- 数据转储
  - 转储是指DBA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，或备份。
- 数据转储方法
  - 静态转储与动态转储
  - 全量转储与增量转储



静态转储

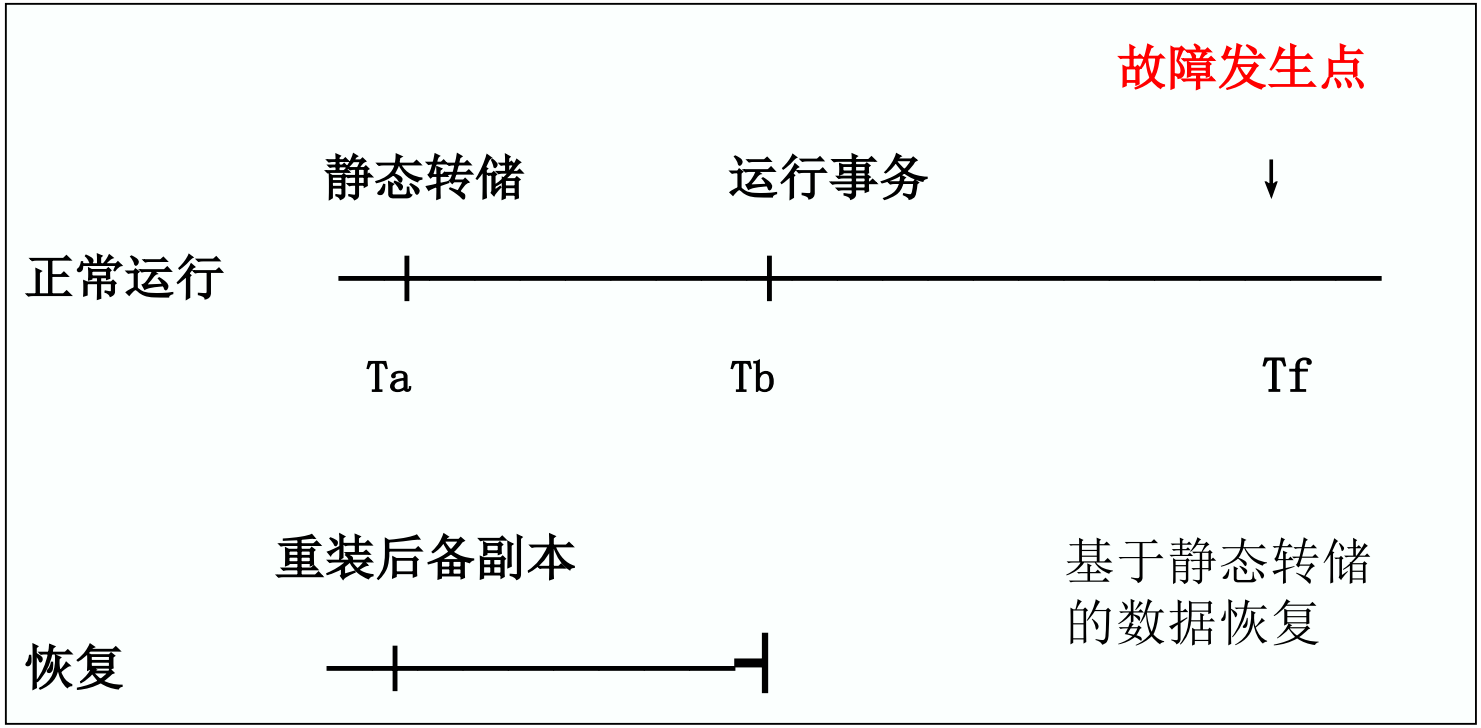
■在系统中无运行事务时进行转储

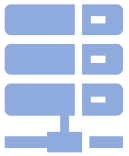
- 转储开始时数据库处于一致性状态
- 转储期间不允许对数据库的任何存取、修改活动
- 优点：实现简单
- 缺点：降低了数据库的可用性
  - ✓ 转储必须等用户事务结束
  - ✓ 新的事务必须等转储结束

静态转储

■在系统中无运行事务时进行转储

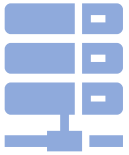
- 转储开始时数据库处于一致性状态
- 转储期间不允许对数据库的任何存取、修改活动





动态转储

- 转储操作与用户事务并发进行
- 转储期间允许对数据库进行存取或修改
- 优点
  - 不用等待正在运行的用户事务结束
  - 不会影响新事务的运行
- 动态转储的缺点
  - 不能保证副本中的数据正确有效



动态转储

- 利用动态转储得到的副本进行故障恢复
  - 需要把动态转储期间各事务对数据库的修改活动登记下来，建立**日志文件**
  - **后备副本**加上**日志文件**才能把数据库恢复到某一时刻的正确状态

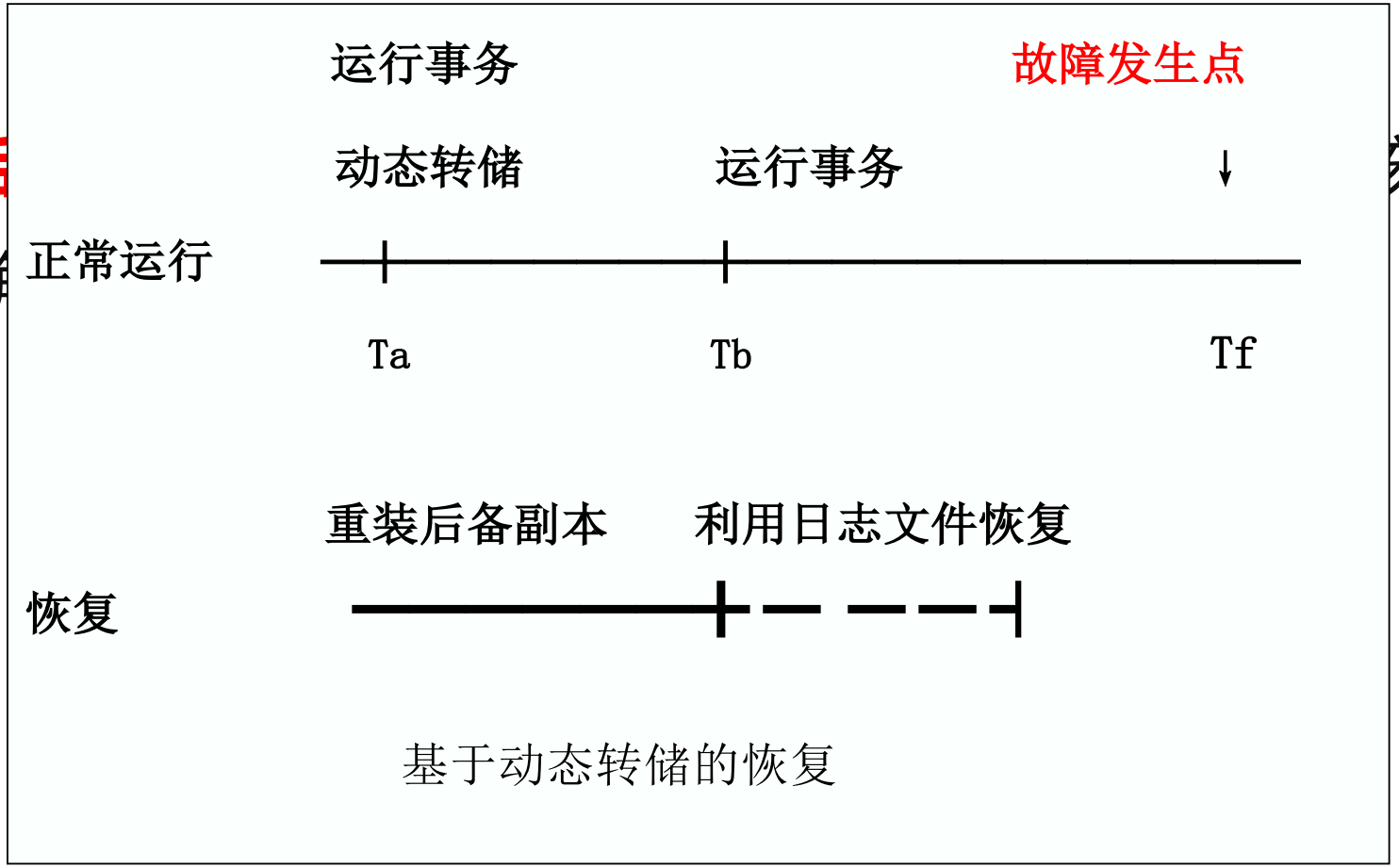
动态转储

■利用动态转储得到的副本进行故障恢复

■需要把动态转储期间各事务对数据库的修改活动登记下来，

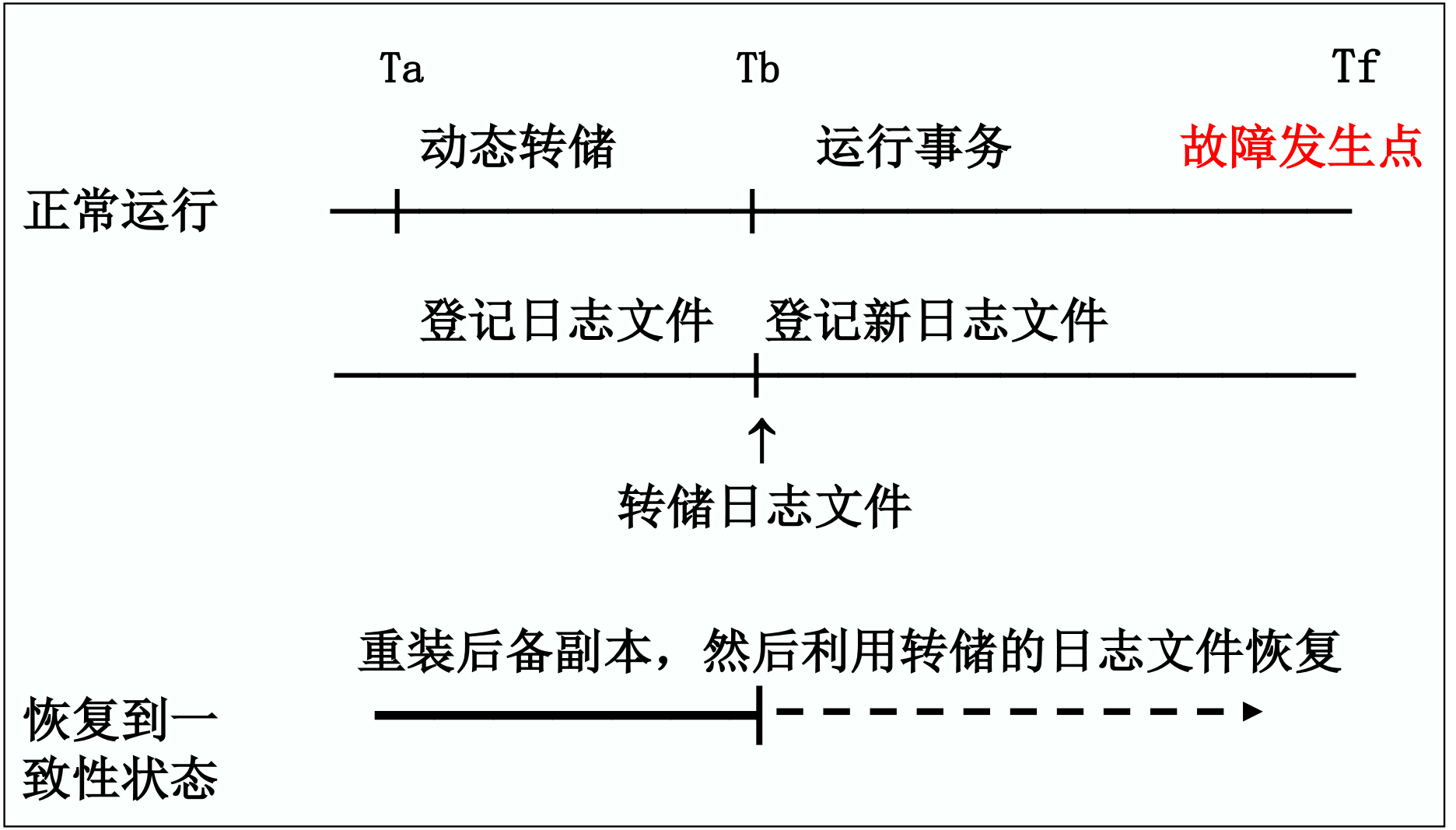
■后

刻的正确

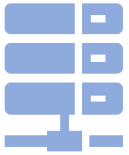


基于动态转储的恢复

动态转储



基于动态转储的恢复



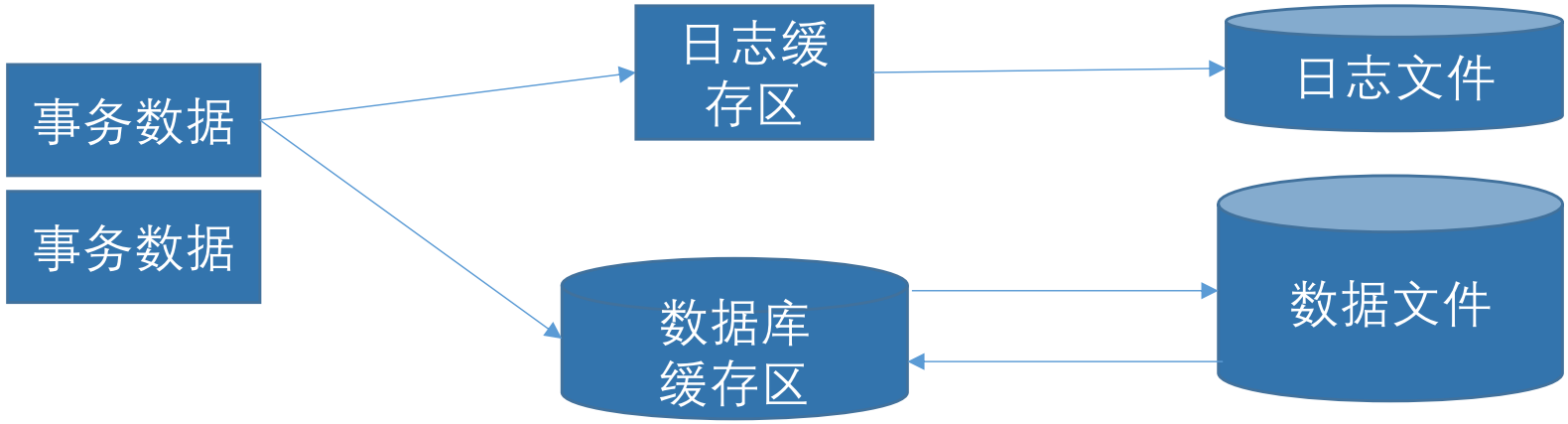
全量转储与增量转储

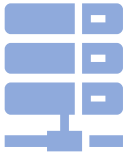
- 全量转储
  - 每次转储全部数据文件
  - 恢复方便，但会影响数据库运行
  - 通常隔一段时间做一次全量转储，做为增量转储和恢复的基准
- 增量转储
  - 只转储上次转储后更新过的数据
  - 恢复策略复杂，但备份时对数据库影响小

恢复技术

■登记日志文件（Log File）

- 日志文件是用来记录事务对数据库的更新操作的文件
- 不同事务的日志记录交错存储
- 顺序写入，写入效率高（数据文件是随机写入，效率低）
- 事务操作先写入日志文件，写成功后，再写入数据缓冲区，最后再择机写入数据文件

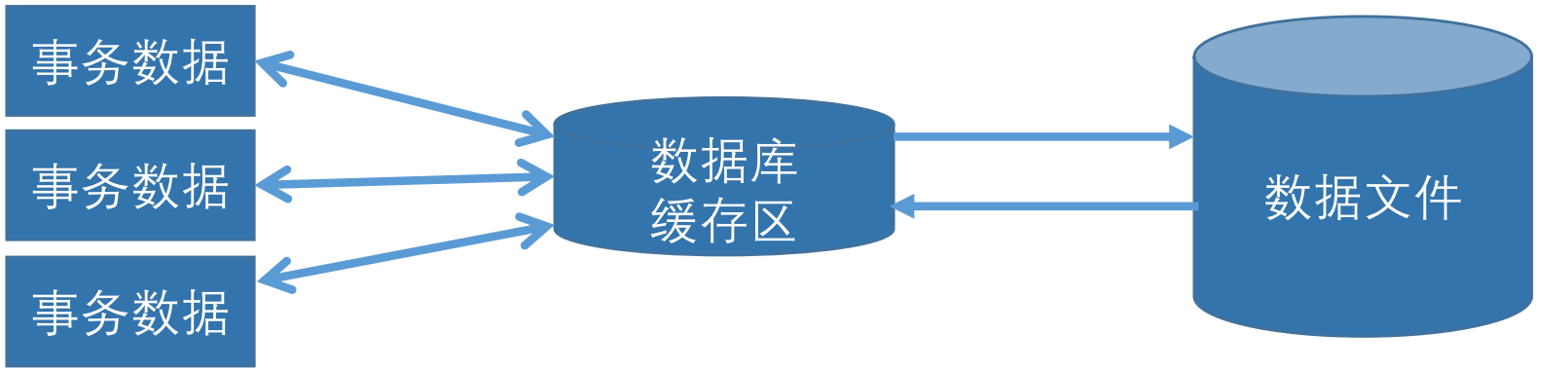




恢复技术

- 日志文件内容
  - 各个事务的开始标记(BEGIN TRANSACTION)
  - 各个事务的结束标记(COMMIT或ROLLBACK)
  - 各个事务的数据修改操作
  - 与事务有关的内部操作（如检查点等信息）
- 日志文件作用
  - 发生系统故障时，利用日志文件进行故障恢复
    - 故障时已提交事务进行重做（Redo）操作-持久性
    - 故障时未提交事务进行撤销（Undo）操作-原子性

备份与恢复的环境设定



内存：易失性存储　　磁盘：持久性存储

■事务原语

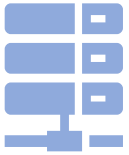
- INPUT(X)
  - ✓ 把数据项 X 读到内存缓冲区
- READ(X,t)
  - ✓ 把数据项 X 拷贝到事务的本地变量 t 中
- WRITE(X,t)
  - ✓ 把事务本地变量 t 拷贝回数据项 X
- OUTPUT(X)
  - ✓ 将数据项 X 写到磁盘里记录下来

备份与恢复的环境设定

READ(A,t); t := t*2; WRITE(A,t)

READ(B,t); t := t*2; WRITE(B,t)

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16



备份与恢复的环境设定

- 缓冲区内容和磁盘内容不一定一致
- 缓冲区处理策略
  - Force： 内存中的数据最晚在commit的时候写入磁盘
  - No Force： 内存中的数据可以一直保留，在commit之后过一段时间再写入磁盘（系统崩溃时数据可能还没写入磁盘，没能持久化）
  - No Steal： 不允许在commit之前把内存中数据写入磁盘
  - Steal ： 允许在事务commit之前把内存中的数据写入磁盘（系统崩溃时可能有未提交的数据被持久化了）

已提交事务

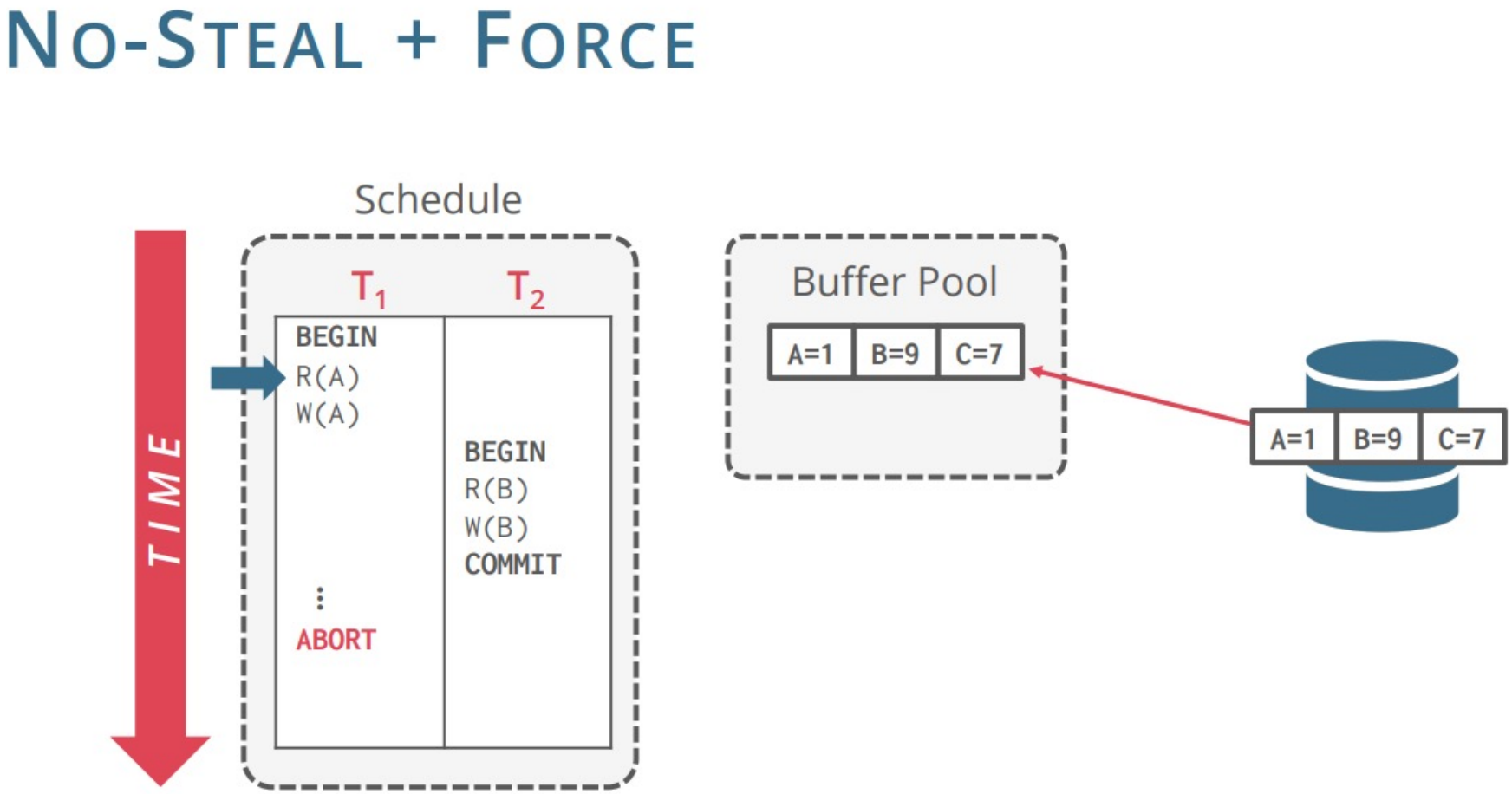
未提交事务

为提升数据读写性能，当前数据库常用No Force+Steal 策略

为此也增加了数据恢复策略的复杂性

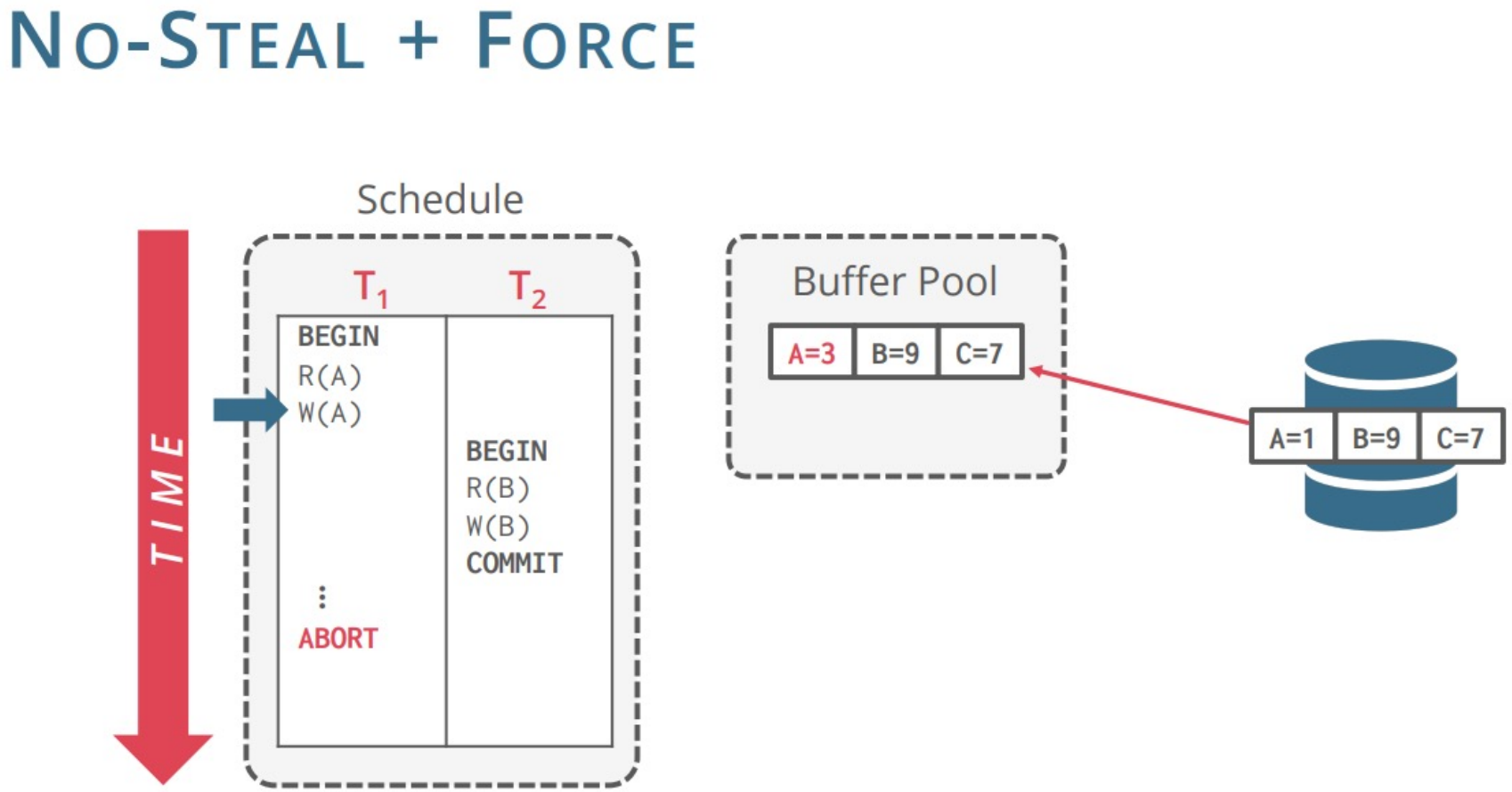
备份与恢复的环境设定

■缓冲区处理策略



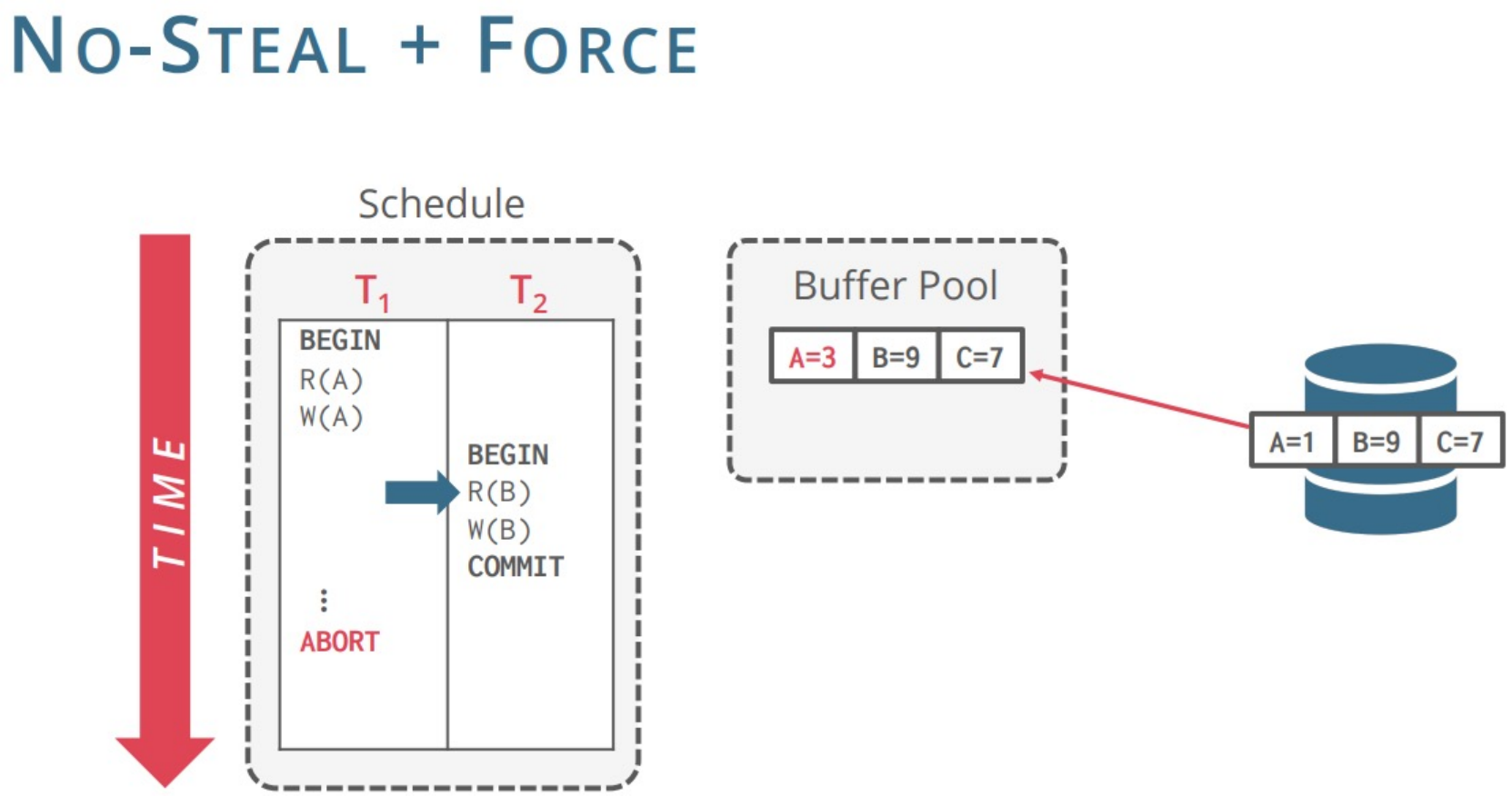
备份与恢复的环境设定

■缓冲区处理策略



备份与恢复的环境设定

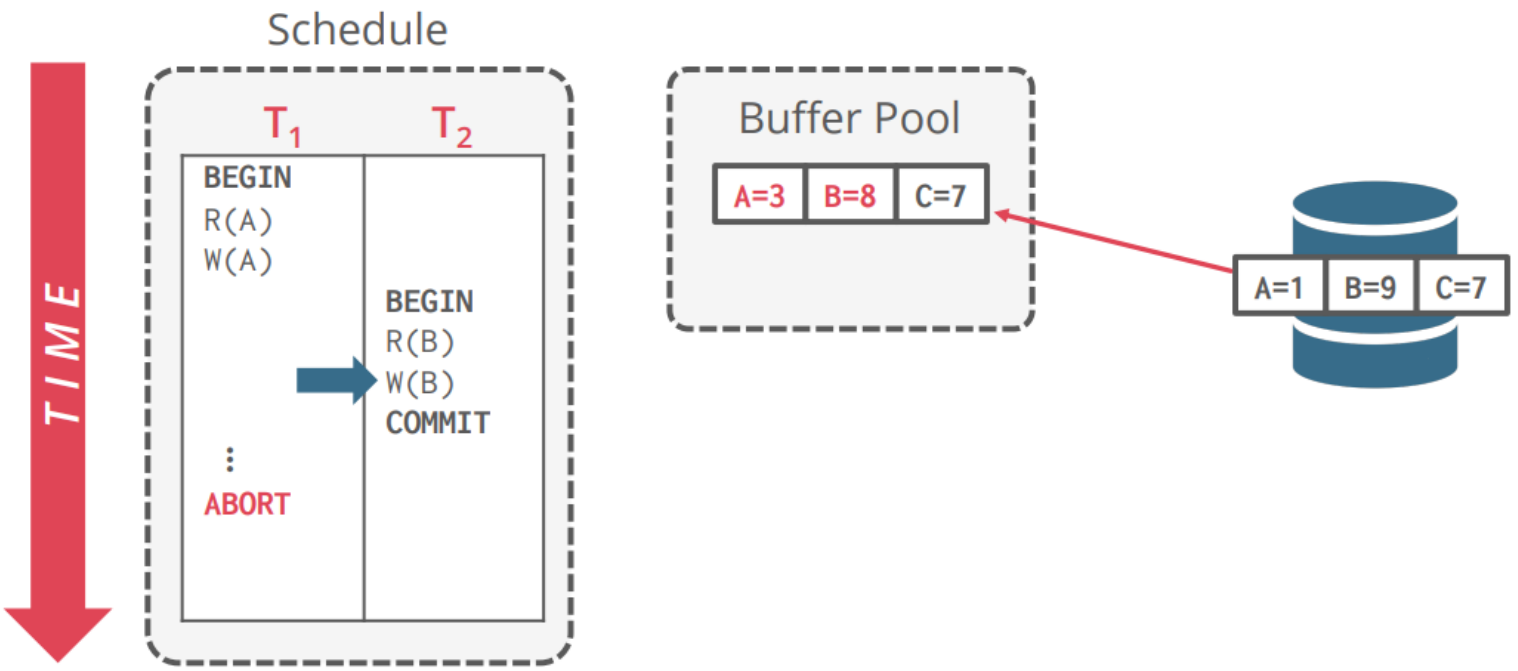
■缓冲区处理策略



备份与恢复的环境设定

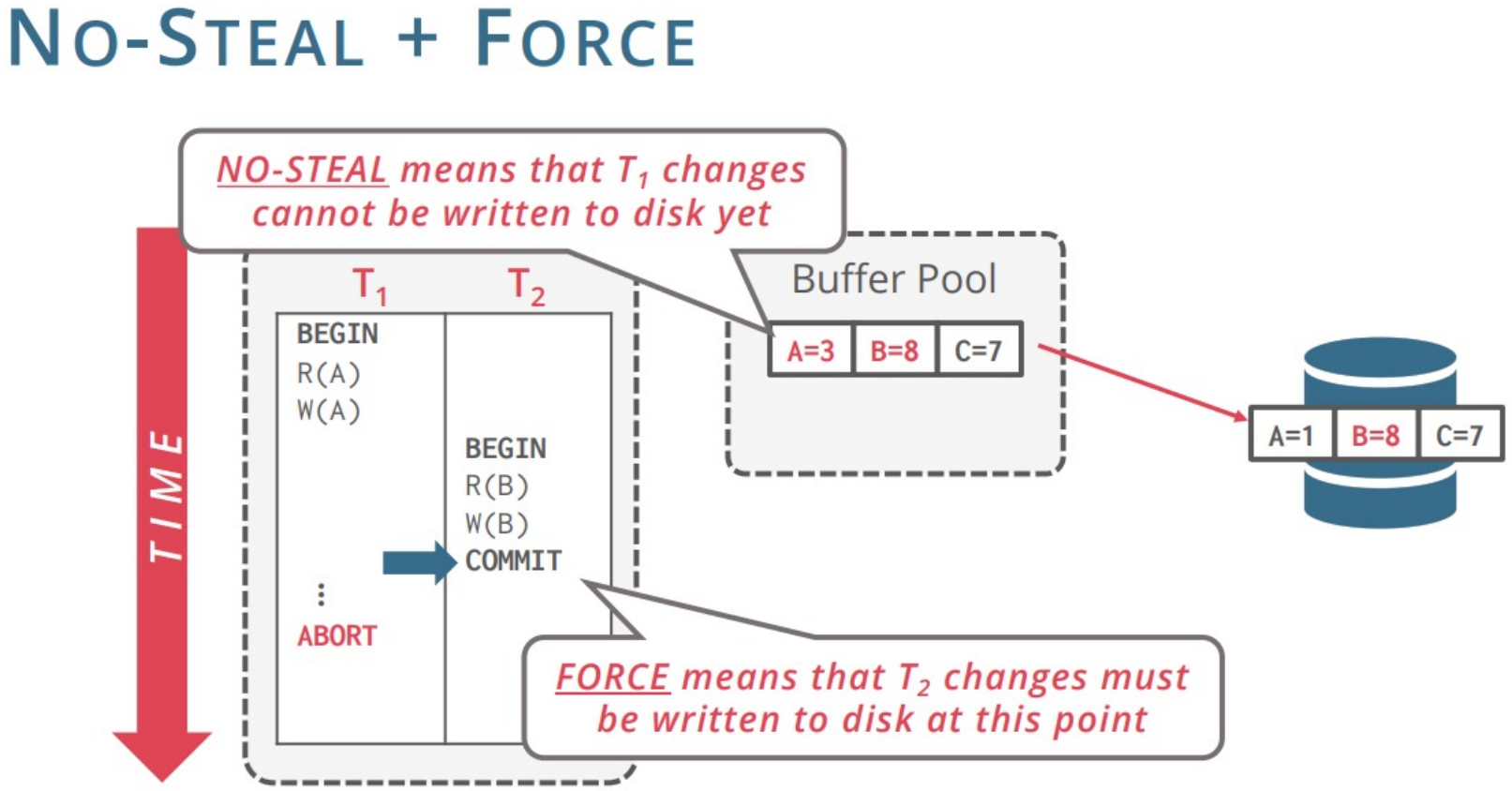
■缓冲区处理策略

NO-STEAL + FORCE



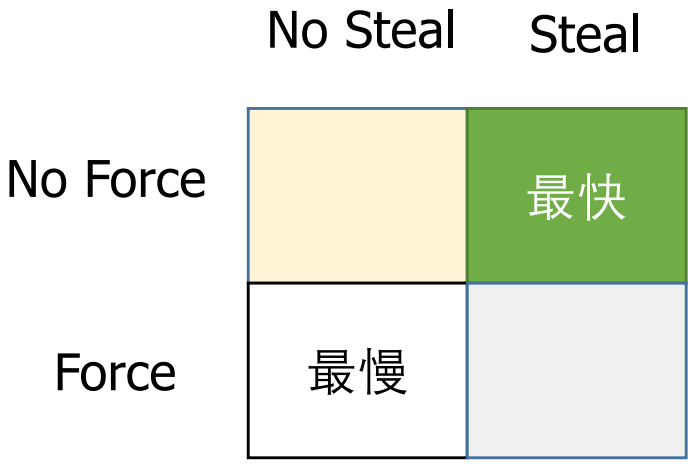
备份与恢复的环境设定

■缓冲区处理策略

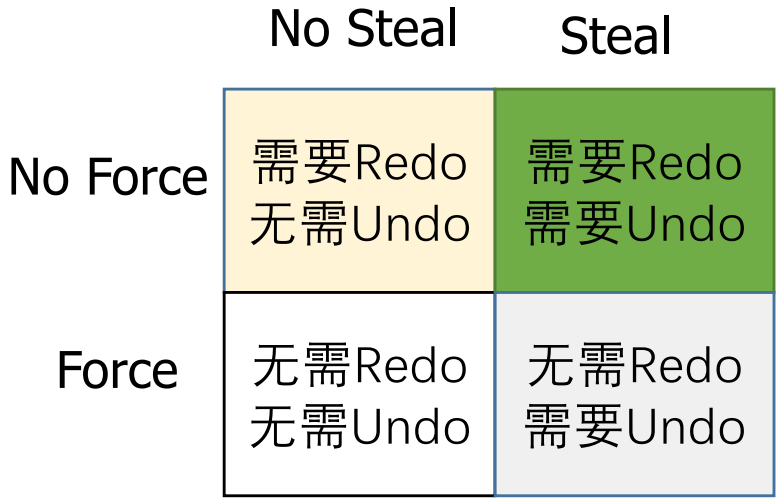


恢复技术

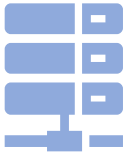
- 缓冲区处理策略与日志/恢复策略的关系



读写性能



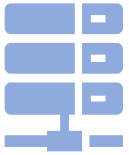
日志/恢复策略



恢复技术

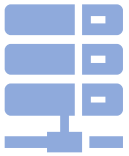
■No Steal和Force模式的恢复

- 未提交事务不可以写入磁盘，提交事务必须写入磁盘
- 系统故障时，没有数据一致性问题，无需恢复
  - ✓ 未提交的事务修改的数据页还停留在内存中，断电后重启，内存中的数据自然也就消失了，自动回滚
  - ✓ 已经提交的事务，其修改的数据页已经写入到磁盘进行持久化，并没有受到影响
- 问题：每次事务都要进行磁盘随机写入（提交事务修改数据），性能很差



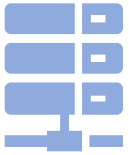
恢复技术

- No Steal和No Force模式的恢复
  - 即未提交事务不可以写入磁盘，提交事务可以暂不写入磁盘
  - 系统故障时
    - 未提交事务并不会受到影响，自动回滚
    - 已提交的事务的修改数据可能还未写入到数据文件中，日志文件与数据文件不一致，持久性未能保证
    - 为此需引入Redo日志文件，解决内存数据丢失的问题。



恢复技术

- Steal和 Force模式的恢复
  - ■即未提交事务可以写入磁盘，提交事务必须写入磁盘
  - ■系统故障时
    - ■已提交的事务，其修改的数据已经写入到磁盘进行持久化，并没有受到影响
    - ■未提交事务，其修改的数据可能已经被持久化到了数据文件中，日志文件与数据文件不一致
    - ■为此需引入Undo日志文件，清除数据文件中的未提交数据。



恢复技术

- Steal和 No Force模式的恢复
  - ■即未提交事务可以写入磁盘，提交事务可以暂不写入磁盘
  - ■性能最好
  - ■系统故障时
    - ■已提交的事务的修改数据可能还未写入到数据文件中
    - ■未提交事务，其修改的数据可能已经被持久化到了数据文件中
    - ■为此需引入Undo日志文件和Redo日志文件，共同解决数据一致性和持久化问题。

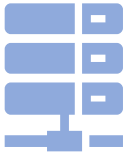
---三种日志文件： Undo日志， Redo日志和Undo/Redo日志

第六章 备份与恢复

1 故障的种类

2 恢复的实现技术

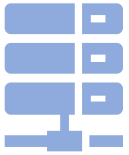
3 恢复的策略



基于Undo日志的恢复策略

■Undo日志记录的内容

- <START T>
  - ✓ 标记事务的开始
- <COMMIT T>
  - ✓ T 事务已经提交
- <ABORT T>
  - ✓ T 事务已被回滚
- <T,X,v>
  - ✓ T 事务已经更新数据项 X, 其更新前的旧值是 v



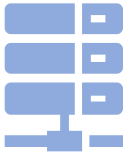
基于Undo日志的恢复策略

■Undo日志的规则

- 规则1： 如果事务 T 更新了数据项 X，则日志记录 <T,X,v> 必须在 X 被写入磁盘之前写入磁盘
- 规则2：如果事务 T 提交, 则日志记录 <COMMIT T> 只有在 T 所做的所有修改都被写入数据文件后才能写入日志文件

更新操作：日志先写，数据文件后写

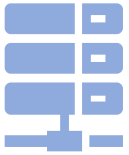
Commit操作：数据文件先写，日志文件后写



基于Undo日志的恢复策略

■Undo日志示例

Action	T	Mem A	Mem B	Disk A	Disk B	Log
INPUT(A)						<START T>
REAT(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>



基于Undo日志的恢复策略

■在系统故障后，进行基于Undo日志的恢复的思路

- Step 1: 查看Undo日志，确定每个事务T是否已经完成
  - ✓ <START T>....<COMMIT T>....　= yes
  - ✓ <START T>....<ABORT T>.....　= yes
  - ✓ <START T>.....　= no
- Step 2: 撤销每个未完成的事务对数据的修改
  - 进行反向操作，如对于delete操作，执行对应的insert操作

基于Undo日志的恢复策略

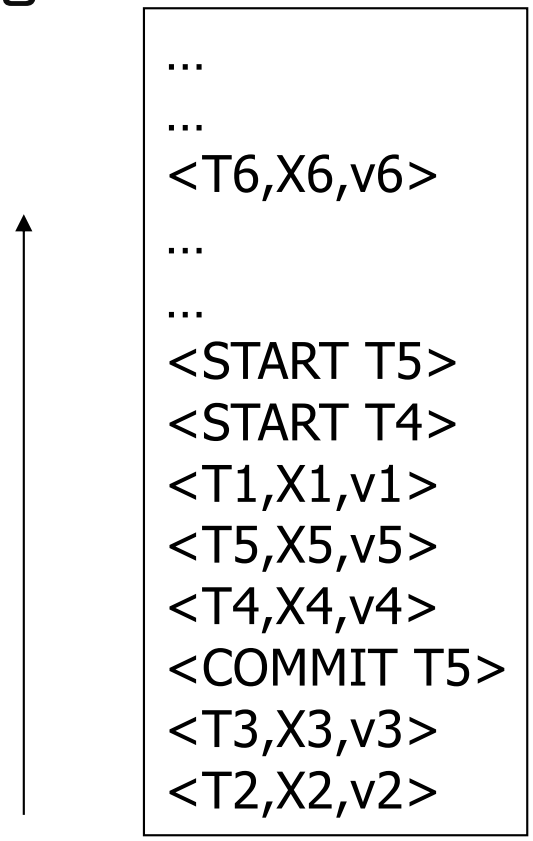
■基于Undo日志的恢复步骤

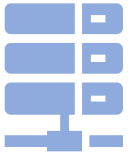
- 从日志文件尾部开始读取日志记录
- 若读到<COMMIT T>：标记 T 为结束状态
- 若读到<ABORT T>: 标记 T 为结束状态
- 若读到<T,X,v>:

  if T 不是结束状态

  then 将 X=v 写入磁盘数据文件

  else 忽略该记录
- 若读到<START T>: 忽略该记录

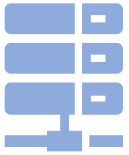




基于Undo日志的恢复策略

■Undo日志恢复的每种操作都是幂等的(*idempotent*)

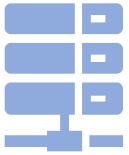
- 对于幂等的操作，我们多次执行的结果和执行一次的结果是一样的，多次重复执行不会对原系统产生危害
- 这意味着如果我们在Undo日志恢复过程中再次遇到系统故障，只需重新启动系统再次从头进行恢复，多次恢复不会影响系统数据



基于Redo日志的恢复策略

■Redo日志记录的内容

- <START T>
  - ✓ 标记事务的开始
- <COMMIT T>
  - ✓ T 事务已经提交
- <ABORT T>
  - ✓ T 事务已被回滚
- <T,X,v>
  - ✓ T 事务已经更新数据项 X, 其新值是 v



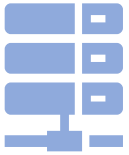
基于Redo日志的恢复策略

■Redo日志的规则

- 如果事务 T 更新了数据项 X，则日志记录 <T,X,v> 和 <COMMIT T>必须在 X 被写入数据文件之前写入日志文件

更新操作和Commit操作：

Redo日志文件先写，数据文件后写



基于Redo日志的恢复策略

■Redo日志示例

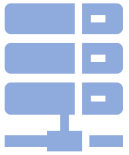
Action	T	Mem A	Mem B	Disk A	Disk B	Log
INPUT(A)						<START T>
REAT(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

基于Redo日志的恢复策略

■在系统故障后，进行基于Redo日志的恢复

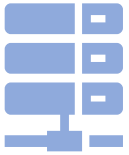
- Step 1: 查看Redo日志，确定每个事务T是否已经完成
  - ✓ <START T>....<COMMIT T>.... = yes
  - ✓ <START T>....<ABORT T>..... = yes
  - ✓ <START T>..... = no
- Step 2: 从Redo日志的开头开始读取日志记录，对于每一个已完成的事务，对它进行重做操作。

<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>
…
…



Undo日志和Redo日志的比较

- Undo日志
  - 每次有事务结束都会先对磁盘进行数据文件写入，导致性能较差
  - 通常在事务故障时根据Undo日志进行回滚操作（恢复旧值），保证事务的原子性
- Redo日志
  - 已提交事务对数据的修改可以延后写入数据文件，提升了数据库运行性能。
  - 通常在系统故障的时候，根据Redo日志进行已提交事务的重做，保证事务的持久性
  - 但是不允许Steal模式，也导致灵活性差
- No Force+Steal 模式下，需要Undo/Redo日志



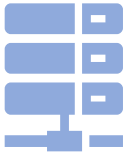
基于Undo/Redo日志的恢复策略

■Undo/Redo日志内容

- <START T>
  - ✓ 标记事务的开始
- <COMMIT T>
  - ✓ T 事务已经提交
- <ABORT T>
  - ✓ T 事务已被回滚
- <T,X,u,v>
  - ✓ T 事务已经更新数据项 X, 其新值是 v，旧值是u

■Undo/Redo日志规则

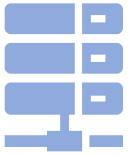
- 如果事务 T 更新了数据项 X，则日志记录 <T,X,u,v> 必须在 X 被写入数据文件之前写入日志文件；Commit则之前、之后皆可



基于Undo/Redo日志的恢复策略

■在系统故障后，进行基于Redo日志的恢复

- Step 1: 查看Undo/Redo日志，确定每个事务T是否已经完成
  - ✓ <START T>....<COMMIT T>....　= yes
  - ✓ <START T>....<ABORT T>.....　= yes
  - ✓ <START T>.....　= no
- Step 2: 从日志的开头开始读取日志记录，对于每一个已完成的事务，对它进行Redo操作。
- Step3: 从日志的末尾开始读取记录，对于每一个未提交事务，对它进行Undo操作。



基于检查点的恢复策略

- 若从头开始进行基于整个日志文件的恢复，会耗费大量时间
- 需要引入基于检查点（check point）的恢复策略
  - 在日志文件中增加检查点记录（checkpoint record）
  - 增加一个“重新开始文件”,记录日志文件维护情况
  - 以最新的检查点作为下次恢复工作的起点

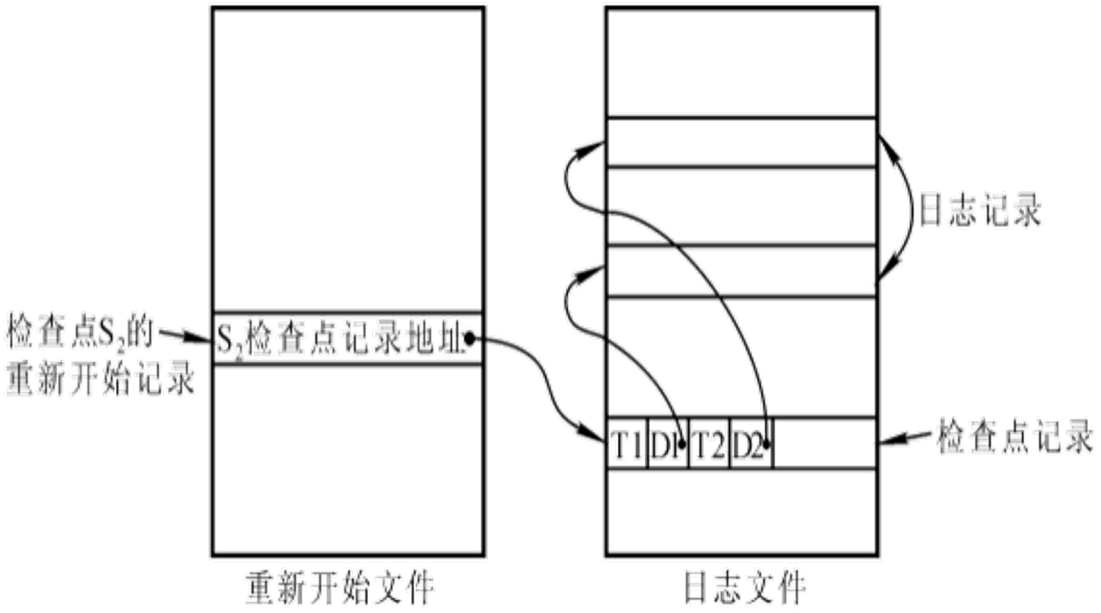
基于检查点的恢复策略

■检查点记录的内容

- 1. 建立检查点时刻所有正在执行的事务清单
- 2. 这些事务最近一个日志记录的地址

■ 重新开始文件的内容

- 记录各个检查点记录在日志文件中的地址



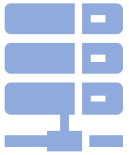
基于检查点的恢复策略

■在检查点维护日志文件

- 1. 将当前日志缓冲区中的所有日志记录写入磁盘的日志文件上。
- 2. 在日志文件中写入一个检查点记录。
- 3. 将当前数据缓冲区的所有数据记录写入磁盘的数据库中。
- 4. 把检查点记录在日志文件中的地址写入一个重新开始文件。

■ 建立检查点

- 按照预定周期，或按照规则条件 (如日志被写满一半)



基于检查点的恢复策略

■检查点之前提交的事务不需要恢复

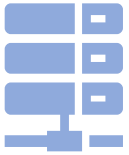
■对于检查点之后才提交的事务，其恢复过程

1. 从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
2. 由该地址在日志文件中找到最后一个检查点记录
3. 由该检查点记录得到检查点建立时刻所有正在执行的事务
4. 从检查点开始正向扫描日志文件，直到日志文件结束，对读到的事务建立已提交和未提交队列
5. 对已提交事务进行Redo操作，对未提交事务进Undo操作

ARIES算法

■Algorithms for Recovery and Isolation Exploiting Semantics（基于语义的恢复与隔离算法），多数数据库采用的数据恢复策略

- 1. Write-Ahead Logging—WAL策略
  - ✓ 出于性能上的考虑，数据的修改都是在内存中进行，并将这些修改操作记录到日志（redo log和undo log）中，然后异步将内存中的数据写入到磁盘；日志也并非立刻写入至磁盘，而是先写入到 log buffer，再按照相应的参数配置进行磁盘的写入操作（立即或定期）
  - ✓ 数据文件是随机写，性能差；日志文件是顺序写，效率高
  - ✓ 事务提交的信息只要写入日志文件即可保证持久性: no force

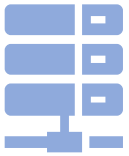


ARIES算法

- 2. 系统故障时，先进行分析，利用Checkpoint及日志中的信息找出所有活跃事务列表
- 3. 随后通过Redo log恢复数据
  - ✓ 对于列表中所有事务，重做事务操作，保证事务持久性
    - ✓ 未提交事务也重做，保证事务所修改数据没有逻辑不一致性（例如故障时事务删了表里的一条数据，但还没更新索引）
  - ✓ 把已重做的已经提交的事务从待处理事务列表中删除
- 4. 最后通过Undo log回滚数据
  - ✓ 对于事务列表中剩下的事务（未提交事务）进行回滚操作，保证事务原子性

作业

- 课本第305页第3，4，5，6，7，8，9题



练习

- Consider the following log:

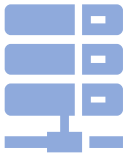
  ⟨START T⟩ ⟨T,A,10,11⟩ ⟨START U⟩ ⟨U,B,20,21⟩

  ⟨T,C,30,31⟩ ⟨U,D,40,41⟩ ⟨COMMIT U⟩

  ⟨T,A,10,11⟩: T is the transaction, A is the modified object, 10 is the old value, 11 is the new value.

- After a crash:
  - What values might/must have been changed?
  - How does the recovery manager get the database back to a consistent state?

Discuss for Undo-, Redo-, and Undo/Redo- logging



练习

Solution (Undo/Redo)

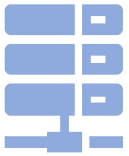
$$\langle \text{START } T\rangle \langle T, A, 10, 11\rangle \langle \text{START } U\rangle \langle U, B, 20, 21\rangle$$
$$\langle T, C, 30, 31\rangle \langle U, D, 40, 41\rangle \langle \text{COMMIT } U\rangle$$

We first identify the transactions that we need to redo and those that we need to undo. U must be redone, while T must be undone.

By reading the log we can conclude that:

Steal / no force

- A *might* have had its value changed on disk.
- B *might* have had its value changed on disk.
- C *might* have had its value changed on disk.
- D *might* have had its value changed on disk.



练习

Solution (Undo/Redo)

$$\langle \text{START } T\rangle\langle T, A, 10, 11\rangle\langle \text{START } U\rangle\langle U, B, 20, 21\rangle$$
$$\langle T, C, 30, 31\rangle\langle U, D, 40, 41\rangle\langle \text{COMMIT } U\rangle$$

Starting from the *end* of the log (undo):

- Ignore changes of transaction U altogether.
- Write value 30 for C.
- Write value 10 for A.
- Append $\langle \text{ABRT } T\rangle$ to the log.

Then, starting from the *beginning* of the log (redo):

- Ignore changes of transaction T altogether.
- Write value 21 for B.
- Write value 41 for D.

练习

Consider the content of the following undo log:
(LSN: log sequence number日志记录的序列号)

LSN1 <START T1>
LSN2 <T1 X 5>
LSN3 <START T2>
LSN4 <T1 Y 7>
LSN5 <T2 X 9>
LSN6 <START T3>
LSN7 <T3 Z 11>
LSN8 <COMMIT T1>
LSN9 <START CKPT(T2,T3)>
LSN10 <T2 X 13>
LSN11 <T3 Y 15>
*C*R*A*S*H*

Questions:

(a) Show how far back in the recovery manager needs to read the log. Write below the earliest LSN that the recovery manager reads.

(b) Show below the actions of the recovery manager during recovery

(c) What is the value of X at the end of the recovery