数据库系统原理与设计

（第 3 版）

《第五项修练》

- 第一项修练：自我超越
- 第二项修练：改善心智模式
- 第三项修练：建立共同愿景
- 第四项修练：团队学习
- 第五项修练：系统思考

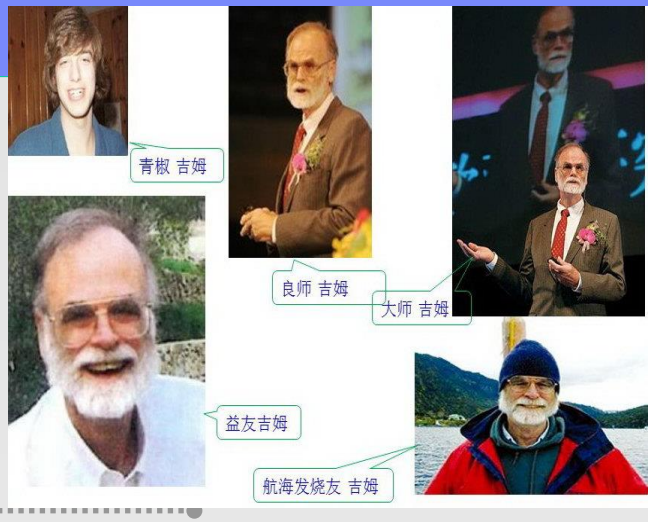
—— 作者：美国商业周刊推崇为当今最有影响力的管理大师之一

数据库系统原理与设计

（第 3 版）

第10章 事务管理与恢复

目　录



2007年1月28日，数据库界的第三位图灵奖得主James Gray（昵称Jim Gray），要去做一件善事--百善孝为先--陪伴他的母亲。然后，然后他们就失联了。带着在数据库事务处理和系统实现方面的功勋，走了！

问题背景

买票,转账

- 现实应用中，数据库的操作与操作之间往往具有一定的语义和关联性。数据库应用希望将这些有关联的操作当作一个逻辑工作单元看待，**要么都执行，要么都不执行**。
- **[例10.1]** 飞机订票系统有两个表Sale和Flight，分别记录各售票点(agentNo)的售票数及全部航班的剩余票数：

  Sale(agentNo, *flightNo*, *date*, saledNumber)

  Flight(flightNo, date, remainNumber)

- 现有A0010售票点(agentNo)欲出售F005航班(flightNo)2018年8月8日(date)机票2张。

问题背景

■ [例10.1] 飞机订票系统有两个表Sale和Flight，分别记录各售票点的售票数及全部航班的剩余票数：

Sale(agentNo, *flightNo*, *date*, saledNumber)

Flight(flightNo, date, remainNumber)

■ 可编制如下程序：

查询F005航班2018年8月8日剩余票数A; *(1)*

if *(A<2)*

拒绝操作，并通知票源不足;

else

更新A0010售票点的售票数;

更新F005航班的剩余票数; } *(2)*

如何用SQL语句分别实现语句(1)和语句(2)？

■ 语句(1)可用SQL语句表示为：

```
SELECT remainNumber
FROM Flight
WHERE flightNo='F005' AND date='2018-08-08'
```

问题背景

- 语句(2)是在当F005航班2018年8月8日的剩余票数大于请求票数时更新Sale和Flight表。该更新包括两个update操作：

```
UPDATE Sale
 SET saledNumber=saledNumber+2
 WHERE  agentNo='A0010' AND flightNo='F005'
       AND date='2018-08-08'

UPDATE Flight
 SET remainNumber=remainNumber–2
 WHERE flightNo='F005' AND date='2018-08-08'
```

如果第一个UPDATE语句执行成功，而第二个UPDATE语句执行失败，会发生什么问题呢？？？

Sale(agentNo, flightNo, date, saledNumber)

agentNo	flightNo	date	saledNumber
A0010	F005	2018/8/8	20
..	..	..	..

Flight(flightNo, date, remainNumber)

flightNo	date	remainNumber
F005	2018/8/8	2
...	...	...

- 假设F005航班共有200个
(其中被A0010售出20张)
- 当第1个UPDATE语句执
为22。当系统发生故障重
求出售F005航班2018年8月8日机票2张，由于F005的剩余票数未更新(仍为2)，因此满足其要求又出售了2张。
- 结果多卖了2张票！

出现上述问题的原因是什么？

出现故障后，系统重新提供服务时数据库状态与现实世界状态出现了不一致。对于机票系统来说，一航班的剩余票数加上已售出票数应等于该航班全部座位数。而重新提供服务时，F005的已售票数与剩余票数之和为202(不等于200!)，导致多买了2张。

问题背景

- 为解决上述问题，数据库管理系统引入了**事务**概念，它将这些有内在联系的操作当作一个逻辑单元看待，并采取相应策略保证一个逻辑单元内的全部操作**要么都执行成功，要么都不执行**。
- 对数据库用户而言，只需将具有完整逻辑意义的一组操作正确地定义在一个事务之内即可。

事务的概念是什么？

```
SELECT remainNumber
FROM Flight
WHERE flightNo='F005' AND date='2008-08-08'

UPDATE Sale
 SET saledNumber=saledNumber+2
 WHERE  agentNo='A0010' AND flightNo='F005'
      AND date='2008-08-08'

UPDATE Flight
 SET remainNumber=remainNumber–2
 WHERE flightNo='F005' AND date='2008-08-08'
```

事务概念

- 对于**用户**而言，**事务**是具有完整逻辑意义的数据库操作序列的集合。
- 对于**数据库管理系统**而言，**事务**则是一个读写操作序列。这些操作是一个不可分割的逻辑工作单元，**要么都做，要么都不做**。

事务结束语句

■ 事务结束的两种类型：

- **事务提交(commit)**：将成功完成事务的执行结果(即更新)永久化，并释放事务占有的全部资源。
- **事务回滚(rollback)**：中止当前事务、撤销其对数据库所做的更新，并释放事务占有的全部资源。

DBMS很多，事务模式有所区别，如SQL Server事务模式是这样的。

SQL Server事务模式

- SQL Server数据库提供了三种类型的事务模式：
  - **显式事务**是指用户使用Transact-SQL事务语句所定义的事务，其事务语句包括：
    - 事务开始：BEGIN TRANSACTION
    - 事务提交：COMMIT TRANSACTION，COMMIT WORK
    - 事务回滚：ROLLBACK TRANSACTION，ROLLBACK WORK
  - **隐式事务**是指事务提交或回滚后，系统自动开始新的事务。该类事务不需要采用BEGIN TRANSACTION语句标识事务的开始。
  - **自动定义事务**：当一个语句成功执行后，它被自动提交，而当执行过程中出错时，则被自动回滚。

SQL Server事务定义举例

- [例10.2] 利用SQL Server提供的显式事务模式定义例10.1中的数据库更新事务。

```
BEGIN TRANSACTION
    UPDATE Sale
    SET saledNumber=saledNumber+2
    WHERE agentNo='A0010' AND flightNo='F005'
        AND date='2008-08-08'

    UPDATE Flight
    SET remainNumber=remainNumber-2
    WHERE flightNo='F005' AND date='2008-08-08'
COMMIT TRANSACTION
```

事务特性

- 为了保证事务并发执行或发生故障时数据库的一致性（完整性），事务应具有以下ACID特性：
  - 原子性(atomicity)。事务的所有操作要么全部都被执行，要么都不被执行。
  - 一致性(consistency)。一个单独执行的事务应保证其执行结果的一致性，即总是将数据库从一个一致性状态转化到另一个一致性状态。
  - 隔离性(isolation)。当多个事务并发执行时，一个事务的执行不能影响另一个事务，即并发执行的各个事务不能互相干扰。
  - 持久性(durability)。一个事务成功提交后，它对数据库的改变必须是永久的，即使随后系统出现故障也不会受到影响。

DBMS保证事务特性措施

- **原子性**也称为**故障原子性**或(**故障**)**可靠性**
  - 由DBMS通过撤销未完成事务对数据库的影响来实现。
- **一致性**是指**单个事务的一致性**，也称为**并发原子性**或**正确性**
  - 由编写该事务代码的应用程序员负责，但有时也可利用DBMS提供的数据库完整性约束(如触发器)的自动检查功能来保证。
- **隔离性**也称为**执行原子性**或**可串行化**，可以看作是多个事务并发执行时的一致性或正确性要求
  - 由DBMS的并发控制模块保证。
- **持久性**也称为**恢复原子性**或**恢复可靠性**
  - 它是利用已记录在稳固存储介质(如磁盘阵列)中的恢复信息(如日志、备份等)来实现丢失数据(如因中断而丢失的存放在主存中但还未保存到磁盘数据库中去的数据等)的恢复。
  - 它是由DBMS的恢复管理模块保证。

事务并发执行

- 数据库管理系统允许多个事务并发执行：
  - 优点
    - 增加系统吞吐量(throughput)。吞吐量是指单位时间系统完成事务的数量。当一事务需等待磁盘I/O时，CPU可去处理其它正在等待CPU的事务。这样，可减少CPU和磁盘空闲时间，增加给定时间内完成事务的数量。
    - 减少平均响应时间(average response time)。事务响应时间是指事务从提交给系统到最后完成所需要的时间。事务的执行时间有长有短，如果按事务到达的顺序依次执行，则短事务就可能会由于等待长事务导致完成时间的延长。如果允许并发执行，短事务可以较早地完成。因此，并发执行可减少事务的平均响应时间。
  - 缺点
    - 若不对事务的并发执行加以控制，则可能破坏数据库的一致性。

事务并发执行可能出现的问题

- 读脏数据,丢失更新,不可重复读。

事务并发执行问题举例

- [例10.3] 设A航班的剩余票数为10张，有两个事务T_1和T_2同时请求出售该航班机票2张和3张。它们各自的执行序列如图10-1所示(这里只考虑对航班剩余票数的更新)。

T_1	T_2
R(*A*)	R(*A*)
A=*A*−2	*A*=*A*−3
W(*A*)	W(*A*)

图10-1　更新事务T_1和T_2

事务并发执行问题举例

- 如果是串行执行，则不管是T_1先执行再执行T_2(图10-2(a))，还是T_2先执行再执行T_1(图10-2(b))，都可得到正确的执行结果，即剩余票数都为5。

T_1	T_2	A
R(A)		10
A=A−2		
W(A)		8
	R(A)	8
	A= A-3	
	W(A)	5

(a)

T_1	T_2	A
	R(A)	10
	A=A−3	
	W(A)	7
R(A)		7
A=A−2		
W(A)		5

(b)

图10-2 T_1和T_2串行执行

事务并发执行问题举例

- **读脏数据**：在图10-3中，事务T_1在T_2读取其更新值(8)后回滚，而T_2仍然使用读到T_1修改后的值进行运算，得到的结果是5。但实际上T_1未执行成功，系统只出售了3张票，余票数应为7。

T_1	T_2	A
R(A)		10
A=A −2		
W(A)		8
	R(A)	8
rollback		
	A=A−3	
	W(A)	5

- **读脏数据**。如果事务T_2读取事务T_1修改但未提交的数据后，事务T_1由于某种原因中止而撤销，这时事务T_2就读取了不一致的数据。数据库中将这种读未提交且被撤销的数据为读“脏数据”。

图10-3 T_2读脏数据

事务并发执行问题举例

- **不可重复读**：如图10-4所示，T_3第一次读时余票数为10张，第二次读时为8张，两次读结果不一致。

T_1	T_3	A
	R(A)	10
R(A)		10
$A=A-2$		
W(A)		8
	R(A)	8

图10-4　T_3不可重复读

- **不可重复读**。是指事务T_i两次从数据库中读取的结果不同，可分为三种情况：
  - 事务T_i读取一数据后，事务T_j对该数据进行了更改。当事务T_i再次读该数据时，则会读到与前一次不同的值。
  - 事务T_i按某条件读取数据库中某些记录后，事务T_j删除了其中部分记录。当事务T_i再次按相同条件读取时，发现记录数变少了。（幻影现象1）
  - 事务T_i按某条件读取数据库中某些记录后，事务T_j插入了新的记录。当事务T_i再次按相同条件读取时，发现记录数变多了。（幻影现象2）

■ **丢失更新**。两个或多个事务都读取了同一数据值并修改，最后提交事务的执行结果覆盖了前面提交事务的执行结果，从而导致前面事务的更新被丢失。

事务并发执行问题举例

■ **丢失更新**：在图10-5中，T_1和T_2都读到余票数为10，由于T_1后于T_2提交，导致T_2的更新操作没有发生作用，被T_1的更新值(8)覆盖。

T_1	T_2	A
R(A)		10
	R(A)	10
	$A=A-3$	
	W(A)	7
$A=A-2$		8
W(A)		8

图10-5　T_2更新丢失

是不是所有并发执行事务都会出现这些问题呢？

事务并发执行得到正确的结果

启示：如果一组并发执行事务的执行结果与它们串行执行得到的结果是相同的，那么就可以认为该并发执行的结果是正确的。

T_4	T_5	A	B
R(A)		10	
$A=A-2$			
W(A)		8	
	R(A)	8	
	$A=A-3$		
	W(A)	5	
R(B)			15
$B=B-2$			
W(B)			13
	R(B)		13
	$B=B-3$		
	W(B)		10

图10-6 T_4和T_5并发执行

- 可以验证图10-6中并发执行的结果与T_4、T_5按先后顺序串行执行的结果是一样的，也是正确的。

T1:a1a2a3a4a5a6

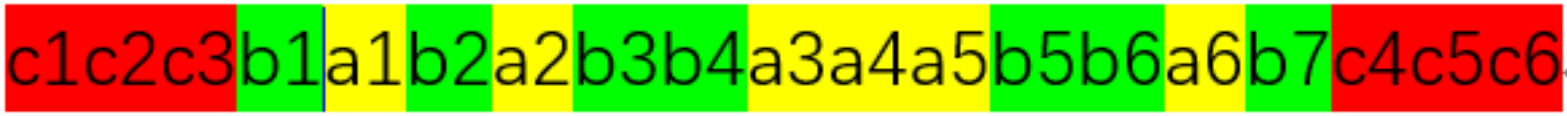
T2: b1b2b3b4b5b6b7

T3: c1c2c3c4c5c6

■ 事务

三个事务的其中一个调度:

成的



■ 对由一组事务操作组成的调度序列而言，应满足下列条件：

- 该调度应包括该组事务的全部操作；
- 属于同一个事务的操作应保持在原事务中的执行顺序。

串行调度

- **定义10.1(串行调度)** 在调度S中，如果属于同一事务的操作都是相邻的，则称S是**串行调度**。
- 对由*n*个事务组成的一组事务而言，共有*n*!种有效串行调度。
- 事务**串行执行**可保证数据库的**一致性**，如果能判断一个并发调度的执行结果等价于一个串行调度的结果，就称该并发调度可保证数据库的一致性。

冲突操作与冲突等价

- 假设调度S包含两个事务T_i与T_j，若两个相邻操作$O_i \in T_i$，$O_j \in T_j$访问不同的数据对象，则交换O_i与O_j不会影响调度中任何操作的结果。若O_i与O_j访问相同的数据对象，并且有一个为写操作时，则不能改变它们被调度执行的顺序。
- **定义10.2** 在一调度S中，如果O_i与O_j是不同事务在相同数据对象上的操作，并且其中至少有一个是写操作，则称O_i与O_j是**冲突操作**；否则称为非冲突操作。
- **定义10.3** 如果一调度S可以经过交换一系列非冲突操作执行的顺序而得到一个新的调度S'，则称S与S'是**冲突等价**的(conflict equivalent)。

冲突可串行化

- **定义10.4** 如果一调度*S*与一串行调度是冲突等价的，则称*S*是**冲突可串行化**的(conflict serializable).
- 在图10-6的调度中，通过交互非冲突操作可得到图10-7所示的串行调度<T_4, T_5>，故图10-6的调度是冲突可串行化的。

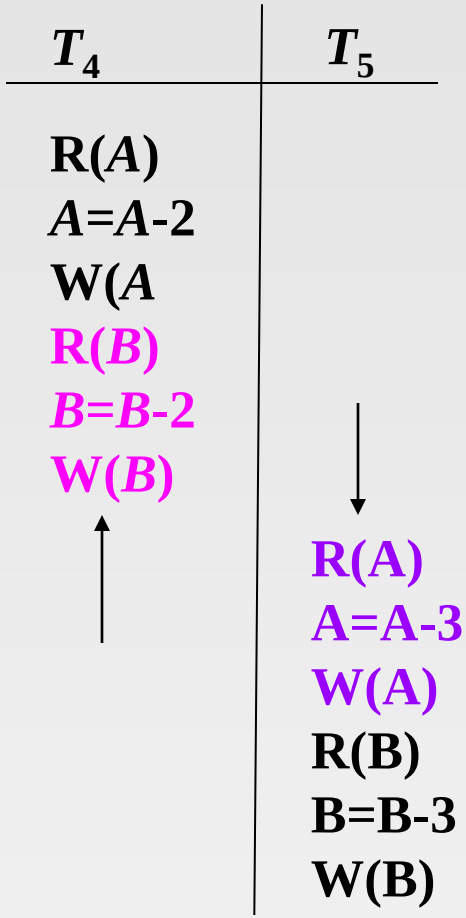


图10-7 交换图10-6的非冲突操作后得到的串行调度<T_4, T_5>

冲突可串行化

- 冲突可串行化仅仅是正确调度的充分条件，并不是必要条件，即冲突可串行化调度执行结果一定是正确的，而正确的调度不一定都是冲突可串行化的。

T_4	T_6	A	B
R(A)		10	
$A=A-2$			
W(A)		8	
	R(B)		15
	$B=B-3$		
	W(B)		12
R(B)			12
$B=B-2$			
W(B)			10
	R(A)	8	
	$A=A-3$		
	W(A)	5	

图10-8 不满足冲突可串行化的正确调度

优先图

- 设S是一个调度。由S构造一个有向图，称为**优先图**，记为$G=(V, E)$，其中V是顶点集，E是边集。顶点集由所有参与调度的事务组成，边集由满足下列3个条件之一的边$T_i \rightarrow T_j$组成：
  - T_i执行了$W_i(Q)$后T_j执行$R_j(Q)$；
  - T_i执行了$R_i(Q)$后T_j执行$W_j(Q)$；
  - T_i执行了$W_i(Q)$后T_j执行$W_j(Q)$。

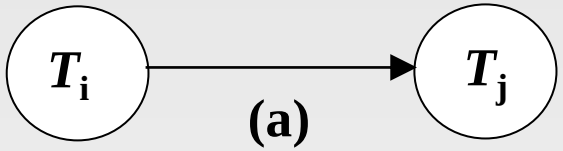


图10-9　图10-2的优先图

优先图

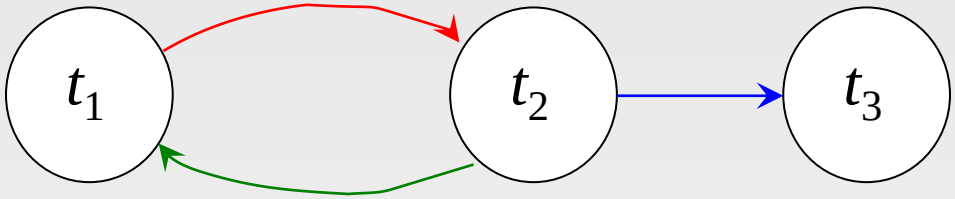
设S是一个调度。由S构造一个有向图，称为优先图，记为$G=(V, E)$，其中V是顶点集，E是边集。顶点集由所有参与调度的事务组成，边集由满足下列3个条件之一的边$T_i \rightarrow T_j$组成：

- T_i执行了$W_i(Q)$后T_j执行$R_j(Q)$；
- T_i执行了$R_i(Q)$后T_j执行$W_j(Q)$；
- T_i执行了$W_i(Q)$后T_j执行$W_j(Q)$。

- $T=\{ t_1: R_1(X)\ W_1(X)\ C_1;$
  $t_2: R_2(X)\ R_2(Y)\ W_2(X)\ C_2;$
  $t_3: R_3(Y)\ W_3(Y)\ C_3\}$

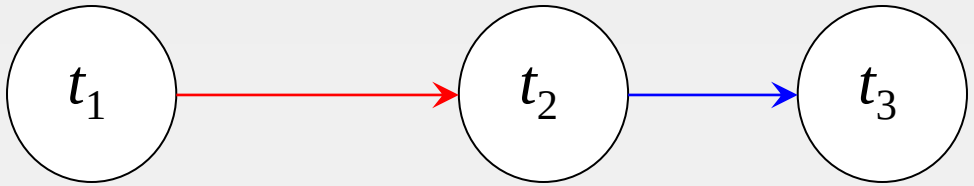
➢ $H_T = R_1(X)\ R_2(X)\ W_1(X)\ R_2(Y)\ C_1\ W_2(X)\ R_3(Y)\ C_2\ W_3(Y)\ C_3$

➢ $SG(H_T)$:



➢ $H_T = R_1(X)\ W_1(X)\ R_2(X)\ R_2(Y)\ C_1\ W_2(X)\ R_3(Y)\ C_2\ W_3(Y)\ C_3$

➢ $SG(H_T)$:



基于优先图的冲突可串行化判别

- **基于优先图的冲突可串行化判别准则：如果优先图中无环，则S是冲突可串行化的；如果优先图中有环，则S是非冲突可串行化的。**
- **测试冲突可串行化的算法：**
  - 构建S的优先图；
  - 采用环路测试算法（如基于深度优先搜索的环检测算法）检测S中是否有环；
  - 若S包含环，则S是非冲突可串行化的，否则调度S是冲突可串行化的。

基于优先图的冲突可串行化判别

■ [例10.5] 设并发调度事务集为：

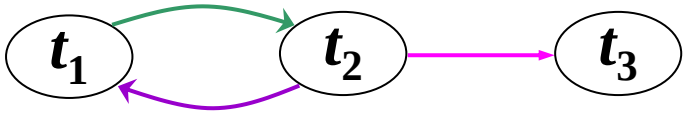
T={t_1: $R_1(X)$ $W_1(X)$, t_2: $R_2(X)$ $R_2(Y)$ $W_2(X)$, t_3: $R_3(Y)$ $W_3(Y)$}

两个并发调度分别为：

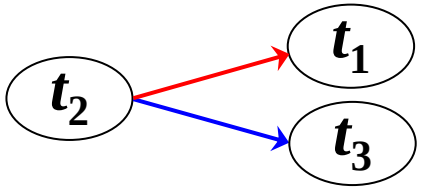
H_T=$R_1(X)$ $R_2(X)$ $W_1(X)$ $R_2(Y)$ $W_2(X)$ $R_3(Y)$ $W_3(Y)$

H_T'=$R_2(X)$ $R_3(Y)$ $R_2(Y)$ $W_2(X)$ $R_1(X)$ $W_3(Y)$ $W_1(X)$

试判断这两个调度是否是冲突可串行化调度？



(a) 调度H_T的优先图



(b) 调度H_T'的优先图

因此，调度H_T不是冲突可串行化的；而调度H_T'是冲突可串行化的，它冲突等价于串行调度<t_2, t_1, t_3>或<t_2, t_3, t_1>。

小结

- 事务四个特点，事务的问题？如何克服问题？，事务可串行化，判断方法？

目 录

并发控制概述

- 并发控制机制大体上可分为**悲观的**和**乐观的**两种。
- **悲观的并发控制方法**认为数据库的一致性经常会受到破坏，因此在事务访问数据对象前须采取一定措施加以控制，只有得到访问许可时，才能访问数据对象，如基于封锁的并发控制方法。——事前控制
- **乐观的并发控制方法**则认为数据库的一致性通常不会遭到破坏，故事务执行时可直接访问数据对象，只在事务结束时才验证数据库的一致性是否会遭到破坏，如基于有效性验证方法。——事后验证
- 本章介绍基于封锁的并发控制方法。

基于封锁方法的基本思想

- **基本思想**：当事务*T*需访问数据对象*Q*时，先申请对*Q*的锁。如批准获得，则事务*T*继续执行，且此后不允许其他任何事务修改*Q*，直到事务*T*释放*Q*上的锁为止。
- 基本锁类型：
  - **共享锁**（shared lock, 记为S）：如果事务*T*获得了数据对象*Q*的共享锁，则事务*T*可读*Q*但不能写*Q*。
  - **排它锁**（eXclusive lock, 记为X）：如果事务*T*获得了数据对象*Q*上的排它锁，则事务*T*既可读*Q*又可写*Q*。

锁相容性

- “锁相容”是指如果T_i已持有数据对象Q的某类型锁后，事务T_j也申请对Q的封锁。如果允许事务T_j获得对Q的锁，则称事务T_j申请锁类型与事务T_i的持有锁类型相容；否则称为不相容。
- 基本锁类型的封锁相容性原则：
  - 共享锁与共享锁相容
  - 排它锁与共享锁、排它锁与排它锁是不相容的。

T_j \ T_i	S	X
S	+	—
X	—	—

“+”表示相容

“—”表示不相容

图10-12 基本锁类型的相容性矩阵

单个事务封锁举例

- 申请和释放锁操作：
  - SL(Q)——申请数据对象Q上的共享锁；
  - XL(Q)——申请数据对象Q上的排它锁；
  - UL(Q)——释放数据对象Q上的锁。

T_4	T_5	A	B
R(A)		10	
A=A−2			
W(A)		8	
	R(A)	8	
	A=A−3		
	W(A)	5	
R(B)			15
B=B−2			
W(B)			13
	R(B)		13
	B=B−3		
	W(B)		10

图10-6 T_4和T_5并发执行

封锁能否保证并发执行事务的冲突可串行化？

T_4

XL(A)
R(A)
A=A−2
W(A)
UL(A)

其他事务在此期间不允许访问数据对象A

XL(B)
R(B)
B=B−2
W(B)
UL(B)

其他事务在此期间不允许访问数据对象B

COMMIT

[例10.6] 假设事务在访问完数据对象后立即释放锁，则添加了封锁操作的事务T_4操作序列如图10-13所示。由于事务T_4要对A、B进行读写操作，因此访问A和B之前都使用XL操作申请排它锁。这样事务T_4在释放A(或B)的封锁之前，其他事务不能访问A(或B)。

图10-13 增加了封锁的事务T_4操作序列

并发事务封锁举例

- [例10.7] 考虑并发事务T_1、T_2和T_3，它们申请锁和释放锁的规则是：
  - 访问数据对象前根据操作类型申请锁；
  - 访问完后**立即释放锁**；
  - 当一个事务释放锁后，由等待时间较长的事务优先获得锁。

它们的一个可能的并发执行过程如图10-14所示。

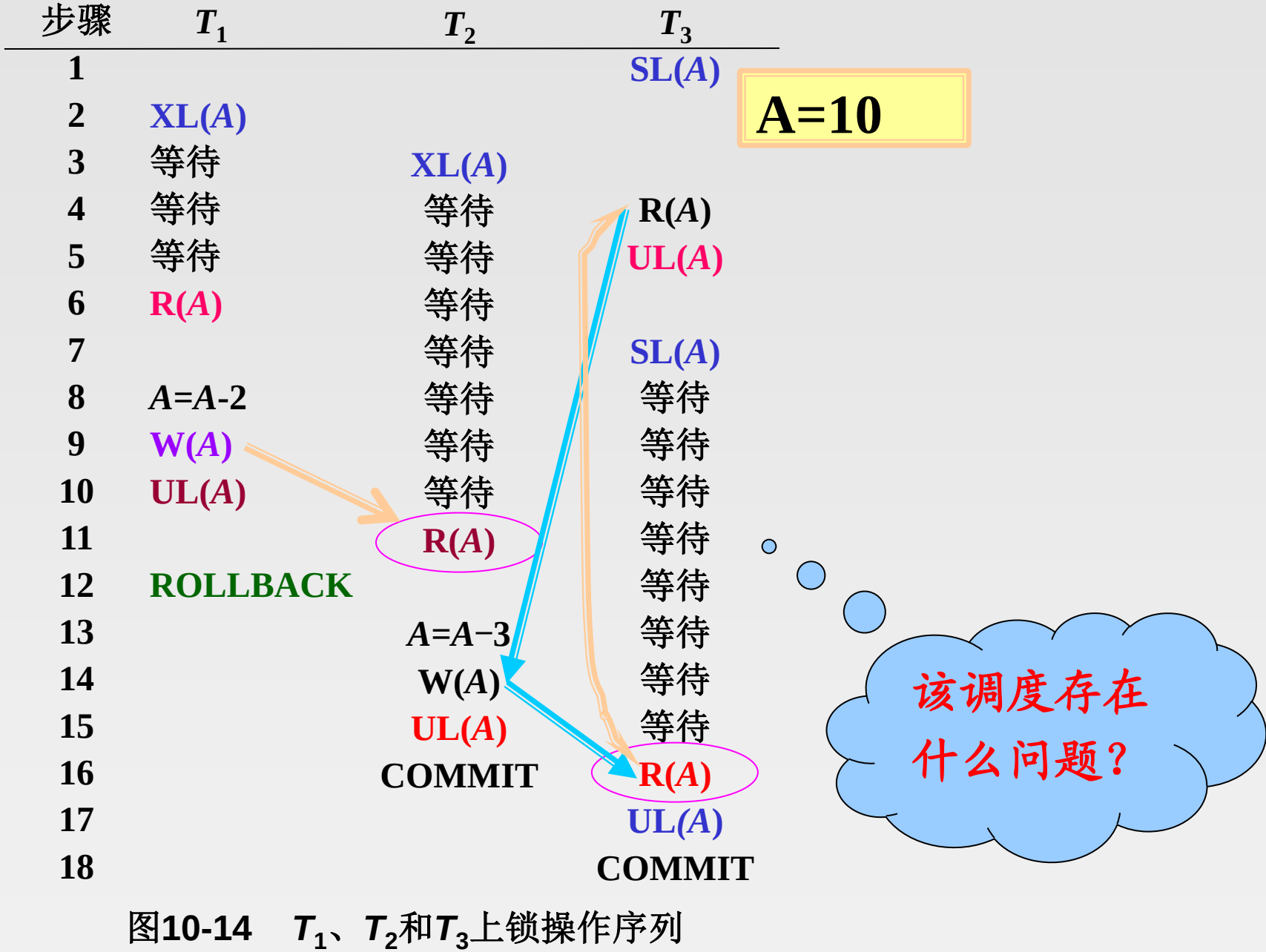


图10-14 T_1、T_2和T_3上锁操作序列

并发事务封锁举例

- 该调度避免了丢失更新，即不会有多个写事务读取同一数据对象的相同值，因为一个数据对象任何时候只能有一个排它锁。

步骤	T_1	T_2	T_3
1			SL(*A*)
2	XL(*A*)		
3	等待	XL(*A*)	
4	等待	等待	R(*A*)
5	等待	等待	UL(*A*)
6	R(*A*)	等待	
7		等待	SL(*A*)
8	*A*=*A*-2	等待	等待
9	W(*A*)	等待	等待
10	UL(*A*)	等待	等待
11		R(*A*)	等待
12	ROLLBACK		等待
13		*A*=*A*−3	等待
14		W(*A*)	等待
15		UL(*A*)	等待
16		COMMIT	R(*A*)
17			UL(*A*)
18			COMMIT

图10-14　T_1、T_2和T_3上锁操作序列

并发事务封锁举例

- 但仍然存在以下问题：
  - 读脏数据。如T_2在步骤11读了T_1修改后的数据，而T_1在步骤12需ROLLBACK。

步骤	T_1	T_2	T_3
1			SL(*A*)
2	XL(*A*)		
3	等待	XL(*A*)	
4	等待	等待	R(*A*)
5	等待	等待	UL(*A*)
6	R(*A*)	等待	
7		等待	SL(*A*)
8	*A*=*A*-2	等待	等待
9	W(*A*)	等待	等待
10	UL(*A*)	等待	等待
11		R(*A*)	等待
12	ROLLBACK		等待
13		*A*=*A*−3	等待
14		W(*A*)	等待
15		UL(*A*)	等待
16		COMMIT	R(*A*)
17			UL(*A*)
18			COMMIT

图10-14　T_1、T_2和T_3上锁操作序列

并发事务封锁举例

- 但仍然存在以下问题：
  - 不可重复读。如T_3两次读到A的值不同。

步骤	T_1	T_2	T_3
1			SL(A)
2	XL(A)		
3	等待	XL(A)	
4	等待	等待	R(A)
5	等待	等待	UL(A)
6	R(A)	等待	
7		等待	SL(A)
8	A=A-2	等待	等待
9	W(A)	等待	等待
10	UL(A)	等待	等待
11		R(A)	等待
12	ROLLBACK		等待
13		A=A−3	等待
14		W(A)	等待
15		UL(A)	等待
16		COMMIT	R(A)
17			UL(A)
18			COMMIT

图10-14　T_1、T_2和T_3上锁操作序列

并发事务封锁举例

- 但仍然存在以下问题：
  - 不可串行化。无论如何交换非冲突操作，上述调度都不能等价于T_1、T_2和T_3的任何一个串行调度。

出现上述问题的原因是事务过早释放了其持有的锁！

步骤	T_1	T_2	T_3
1			SL(*A*)
2	XL(*A*)		
3	等待	XL(*A*)	
4	等待	等待	R(*A*)
5	等待	等待	UL(*A*)
6	R(*A*)	等待	
7		等待	SL(*A*)
8	*A*=*A*-2	等待	等待
9	W(*A*)	等待	等待
10	UL(*A*)	等待	等待
11		R(*A*)	等待
12	ROLLBACK		等待
13		*A*=*A*−3	等待
14		W(*A*)	等待
15		UL(*A*)	等待
16		COMMIT	R(*A*)
17			UL(*A*)
18			COMMIT

图10-14　T_1、T_2和T_3上锁操作序列

两阶段封锁协议

- **两阶段封锁协议**要求每个事务分两个阶段完成封锁操作：**增长(申请锁)**阶段和**缩减(释放锁)**阶段：
  - **增长阶段**：事务可以获得锁，但不能释放锁；
  - **缩减阶段**：事务可以释放锁，但不能获得新锁。
- **两阶段封锁协议**能保证**冲突可串行化**。对于任何事务，调度中该事务获得其最后加锁的时刻(增长阶段结束点)称为事务的**封锁点**。这样，多个事务可以根据它们的**封锁点**进行排序，而这个顺序就是并发事务的一个冲突可串行化顺序。

并发控制

- 两段锁协议(Two-phase Locking)并发控制非常重要的协议。
  - 内容：

①在对任何数据进行读写之前，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。

即事务分为两个阶段：

生长阶段：获得封锁。

收缩阶段：释放封锁。

  - 定理：若所有事务均遵从两段锁协议，则这些事务的所有并行调度都是可串行化的。

两段锁协议(Two-phase Locking)

- 内容：

①在对任何数据进行读写之前，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。

即事务分为两个阶段：

生长阶段：获得封锁。

收缩阶段：释放封锁。

- 定理：若所有事务均遵从两段锁协议，则这些事务的所有并行调度都是可串行化的。

两阶段封锁协议举例

■ **[例10.8]** 图10-15采用了两阶段封锁，允许事务T_4在获得全部锁后（A和B上的排它锁）提前释放部分锁（如步骤7释放了A上的排它锁），事务T_5得以提前执行，从而提高了事务T_4和T_5的并发度。

冲突可串行化，它等价于$<T_4, T_5>$串行调度。

步骤	T_4	T_5
1	XL(A)	
2		XL(*A*)
3	R(A)	等待
4	A=A-2	等待
5	W(A)	等待
6	XL(B)	等待
7	UL(A)	等待
8		R(*A*)
9		A=*A*−3
10	R(B)	
11	B=B-2	
12		W(*A*)
13		XL(*B*)
14	W(B)	等待
15	UL(B)	等待
16		UL(*A*)
17		R(*B*)
18		*B*=*B*−3
19		W(*B*)
20		UL(*B*)

图10-15 T_4和T_5的两阶段封锁

两阶段封锁协议存在的问题

■ 问题一：可能导致死锁

- 持有锁事务出现相互等待都不能继续执行。

T_4	T_6
XL(A)	
R(A)	
	XL(B)
	R(B)
$A=A-2$	
W(A)	
	$B=B-3$
	W(B)
XL(B)	
(等待T_6释放B上的排它锁)	
	XL(A)
	(等待T_4释放A上的排它锁)

图10-16 采用两阶段封锁时T_4和T_6出现死锁

两阶段封锁协议存在的问题

- 问题二：不能避免读脏数据（T_2读取了T_1的未提交更新结果）

导致的后果是“级联回滚”！

T_1	T_2	T_7
R(A)		
A=A−2		
W(A)		
	R(A)	
	A=A-3	
	W(A)	
ROLLBACK		
		R(A)
		A=A−4
		W(A)

图10-17 由于读脏数据引起的级联回滚

两阶段封锁协议变体

- 对于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加以避免。
- **严格两阶段封锁协议**除了要求封锁是两阶段之外，还要求事务持有的**所有排它锁**必须在事务提交后方可释放。这个要求保证了*未提交事务所写的任何数据在该事务提交之前均以排它方式加锁，防止了其他事务读取这些数据。*
- 另一个两阶段封锁的变体是**强两阶段封锁协议**，它要求事务提交之前不得释放任何锁(包括共享锁和排它锁)。

小结

- 锁的作用
- 两阶段封锁协议

目 录

事务运行过程？

恢复系统

911,灾难已经给人们留下了巨大的伤痛，但这远远没有结束，当重建工作遭遇数据灾难恢复难题时，数据丢失带来的二次灾难正在上演。据统计，“9.11”事故一年后，重返世贸大厦的企业由原先的350家变成150家，另外200家企业由于重要信息系统的破坏，关键数据的丢失而永远的消失了。再来看看国外一些数据灾难恢复研究机构的统计：金融业在灾难停机两天内所受损失为日营业额的50%；如果在两星期内无法进行数据灾难恢复方案，75%的公司将业务信[illegible]42%的公司将再也无法开业；没有实施数[illegible]的公司60%将在灾难后2-3年间破产。[illegible]据灾难恢复方案对灾难后社会的正常[illegible]关键的作用。

我自己电脑ghost的时候，格式化的数据盘！

事务访问数据方式

- 对于一个事务而言，它是通过三个地址空间同数据库进行交互：
  - 保存数据库记录的磁盘块空间——**物理数据库**；
  - 缓冲区管理器所管理内存地址空间——**数据缓冲区**；
  - 事务的局部地址空间——**事务工作区**。

故障分类、特征及恢复策略

■ 事务故障

- 事务在运行过程中由于种种原因，如输入数据的错误、运算溢出、违反了某些完整性限制、某些应用程序的错误以及并发事务发生死锁等，使事务未运行至正常终止点就夭折了，这种情况称为**事务故障**。
- 特征：系统的软件和硬件都能正常运行，内存和磁盘上的数据都未丢失和破坏。
- 恢复策略：强行回滚（ROLLBACK）夭折事务，清除其对数据库的所有修改，使得该事务好象根本没有启动过一样，称这类恢复操作称为事务撤销（UNDO）。

故障分类、特征及恢复策略

■ 系统故障

- 该类故障是指系统在运行过程中，由于某种原因，如操作系统或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)、突然停电等造成系统停止运行，致使所有正在运行的事务都以非正常方式终止。
- 特征：数据库缓冲区的信息全部丢失，但存储在外部存储设备上的数据未被破坏。
- 恢复策略
  - UNDO所有未完成事务——原子性的要求。
  - 重做（REDO）所有已提交的事务——持久性的要求。

故障分类、特征及恢复策略

■ 介质故障

- 该类故障是指系统在运行过程中，由于某种硬件故障，如磁盘损坏、磁头碰撞，或操作系统的某种潜在错误，瞬时强磁场干扰等，致使存储在外存中的数据部分丢失或全部丢失。这类故障比前两类故障的可能性小得多，但破坏了磁盘上的数据，危害性最大。
- 特征：存储在外部存储设备上的数据被破坏。
- 恢复策略：需要装入发生介质故障前某个时刻的数据库数据副本，并重做（REDO）自备份相应副本数据库之后的所有成功事务，将这些事务已提交的更新结果重新反映到数据库中去。无需UNDO操作！

故障分类、特征及恢复策略

■ 其它故障

- 黑客入侵、病毒、恶意流氓软件等引起的事务异常结束、篡改数据等不一致性。
- 特征：恶意破坏。
- 恢复策略：通过数据库的安全机制、审计机制等实现对数据的授权访问和保护 。

Shadow Defender 是一款影子系统，从其名称也能看出来。Shadow Defender 是由国人开发的软件，有简体中文。Shadow Defender 支持多分区，支持转储，支持排除，界面及设置选项直观。Shadow Defender 具有防止所有病毒和恶意软件对电脑的攻击；安心上网，消除不必要的痕迹；保护隐私；消除系统停机时间和维护成本；重新启动后电脑系统便可以还原回其原始状态等等特点，特别适合软件测试和安全防护用。

Shadow Defender 简明使用教程【一站式资源】_虚拟机讨论区_反入侵区 卡饭论坛 - 互助分享 - 大气谦和!
http://bbs.kafan.cn/thread-1546629-1-1.html

- **恢复的本质**是利用存储的冗余数据（如日志、影子、备份副本等）来重建数据库中已经被破坏或已经不正确的那部分数据。
- **DBMS中的恢复管理模块由两部分组成：**
  - 正常事务处理过程中：系统需记录冗余的恢复信息，以保证故障发生后有足够的信息进行数据库恢复；
  - 故障发生后：利用冗余信息进行**UNDO或REDO等操作**，将数据库恢复到一致性状态。

日志及特点

- 日志是DBMS记录数据库全部更新操作的序列文件。
- 主要特点有：
  - 日志文件记录了数据库的全部更新顺序。
  - 日志文件是一个追加（append-only）文件。
  - DBMS允许事务的并发执行导致日志文件是“交错的”。
  - 属于单个事务的日志顺序与该事务更新操作的执行顺序是一致的。
  - 日志记录通常是先写到日志缓冲区中，然后写到稳固存储器（如磁盘阵列）中。在不影响讨论的情况下，本章假设日志记录在生成时是直接写到稳固磁盘中。

日志记录类型

- 数据库中的日志记录有两种类型：
  - 记录数据更新操作的日志记录，包括UPDATE、INSERT和DELETE操作；
  - 记录事务操作的日志记录，包括START、COMMIT和ABORT操作。

日志记录格式

- $<T_i, A, V_1, V_2>$表示事务T_i对数据元素A执行了更新操作，V_1表示A更新前的值(前映像)， V_2表示A更新后的值(后映像)。对于插入操作，V_1为空；对于删除操作，V_2为空。
- $<T_i$, START>表示事务T_i已经开始。此时DBMS完成对事务的初始化工作，如分配事务工作区等。
- $<T_i$, COMMIT>表示事务T_i已经提交，即事务T_i已经执行成功（该事务对数据库的修改必须永久化）。事务提交时其更新的数据都写到了数据缓冲区中，但是由于不能控制缓冲区管理器何时将缓冲块从内存写到磁盘。因此当看到该日志记录时，通常不能确定更新是否已经写到磁盘上。
- $<T_i$, ABORT>表示事务已经中止，即事务执行失败。此时，如果T_i所做的更新已反映到磁盘上，DBMS必须通过UNDO操作来消除T_i对磁盘数据库的影响。

先写日志规则

- 为了保证数据库能运用日志进行恢复，要求日志文件必须放到稳固存储器(如磁盘阵列)上，并且要求每条日志记录必须在其所包含数据记录的更新值写到外存储器之前先写到稳固存储器上，即先写(write-ahead)日志规则。

UNDO操作

- 对于要UNDO的事务*T*，日志中记录有<*T*, START>以及*T*对数据库的所有更新操作的日志记录。
- UNDO过程为：从*T*的最后一条更新日志记录开始，从日志尾向日志头(反向)依次将*T*更新的数据元素值恢复为旧值(V_1)。

- 0:a=0
- 1:a=1
- 2:a=a*10
- 3:a=100 X

REDO操作

- 与UNDO相反，REDO操作是对已提交事务进行重做，将数据库状态恢复到事务结束后的状态。
- 对于要REDO的事务*T*，日志中已经记录了<*T*, START>、*T*的所有更新操作日志以及<*T*, COMMIT>。
- REDO过程为：从*T*的第一条更新日志记录开始，从日志头向日志尾(顺向)依次将*T*更新的数据元素值恢复为新值(V_2)。

基于日志恢复策略举例

- [例10.10] 考虑订票事务T_1和T_2，除更新航班的剩余票数A外，还分别需更新售票点的售出票数X和Y。假设先执行T_1，再执行T_2，几种可能的日志记录如图10-19所示。

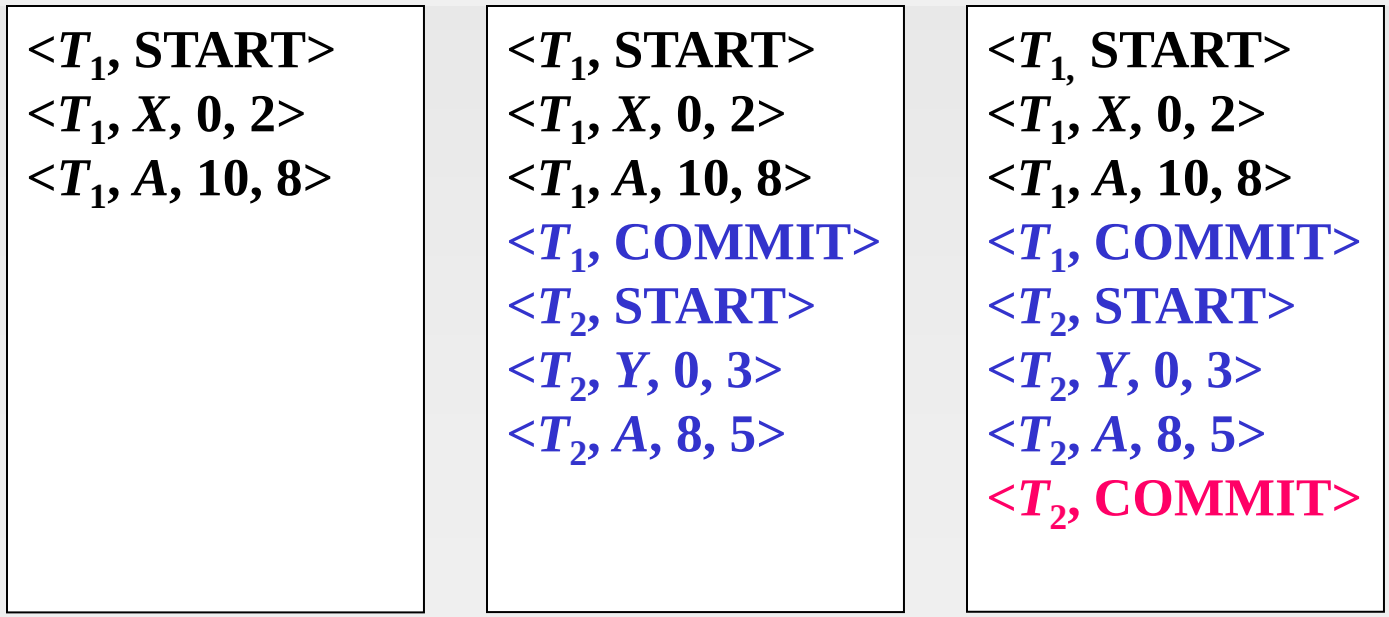


图10-19 T_1和T_2串行执行的三种日志情形

基于日志恢复策略举例

- 情形(a)： T_1完成WRITE(A)后系统发生崩溃。当系统重启动时，检查到有<T_1, START>，但没有<T_1, COMMIT>。因此恢复时执行UNDO(T_1)

<T_1, START>
<T_1, X, 0, 2>
<T_1, A, 10, 8>

(a)

将X和A的值分别恢复为0和10。

图10-19 T_1和T_2串行执行的三种日志情形

基于日志恢复策略举例

- 情形(b)：T_2完成WRITE(A)后系统发生崩溃。这时需分别执行两个恢复操作UNDO(T_2)和REDO(T_1)，因为T_1既有<T_1, START>，又有<T_1, COMMIT>日志，而T_2只有<T_2, START>日志。

<T_1, START> <T_1, X, 0, 2> <T_1, A, 10, 8>

(a)

<T_1, START> <T_1, X, 0, 2> <T_1, A, 10, 8> <T_1, COMMIT> <T_2, START> <T_2, Y, 0, 3> <T_2, A, 8, 5>

(b)

恢复完成后X、Y和A的值分别为2、0和8。

图10-19 T_1和T_2串行执行的三种日志情形

基于日志恢复策略举例

- 情形(c)：T_2完成提交后系统发生崩溃。这时也需执行两个恢复操作REDO(T_1)和REDO(T_2)，因为T_1和T_2都有START和COMMIT日志。　恢复完成后X、Y和A的值分别为2、3和5。

(a)	(b)	(c)
<T_1, START>	<T_1, START>	<T_1, START>
<T_1, X, 0, 2>	<T_1, X, 0, 2>	<T_1, X, 0, 2>
<T_1, A, 10, 8>	<T_1, A, 10, 8>	<T_1, A, 10, 8>
	<T_1, COMMIT>	<T_1, COMMIT>
	<T_2, START>	<T_2, START>
	<T_2, Y, 0, 3>	<T_2, Y, 0, 3>
	<T_2, A, 8, 5>	<T_2, A, 8, 5>
		<T_2, COMMIT>

图10-19　T_1和T_2串行执行的三种日志情形

并发执行事务的基本恢复过程

- 三个阶段：
  - **分析阶段**：从日志头开始顺向扫描日志，确定重做事务集（REDO-set）和撤销事务集（UNDO-set）。将既有<*T*, START>又有<*T*, COMMIT>日志记录的事务*T*加入REDO-set；将只有<*T*, START>没有<*T*, COMMIT>日志记录的事务*T*加入UNDO-set。
  - **撤销阶段**：从日志尾反向扫描日志，对每一条属于UNDO-set中事务的更新操作日志依次执行UNDO操作。
  - **重做阶段**：从日志头顺向扫描日志，对每一条属于REDO-set中事务的更新操作日志依次执行REDO操作。
- UNDO与REDO必须是幂等的，即重复执行任意次的结果与执行一次的结果是一样的。

引入检查点目的

- 利用日志文件恢复主要有两个问题：
  - 日志扫描过程太耗时。因为日志文件必须保存在磁盘中，而且随着时间的不断推进，日志文件在不断扩大，扫描的时间也就变得越来越长。
  - 许多要求REDO事务的更新实际上在恢复时都写入了磁盘的物理数据库中。尽管对它们做REDO操作不会造成不良后果，但会使恢复过程变得更长，导致数据库系统停止服务延长，从而降低了数据库的可用性。
- 为了减少扫描开销和提高恢复效率，引入了检查点技术。

检查点技术

- **检查点**是周期性地向日志中写一条检查点记录并记录所有当前活跃的事务，为恢复管理器提供信息，以决定从日志的何处开始恢复。
- 检查点工作主要包括：
  - 将当前位于日志缓冲区的所有日志记录输出到磁盘上；
  - 将当前位于数据缓冲区的所有更新数据块输出到磁盘上；
  - 记录日志记录<Checkpoint *L*>并输出到磁盘上，其中*L*是做检查点时活跃事务的列表。

静态检查点技术

- 在检查点执行过程中，不允许事务执行任何更新动作，如写缓冲块或写日志记录，称其为静态检查点技术。
- 如果事务*T*在做检查点之前就已提交，那么它的<*T*, COMMIT>记录一定出现在<Checkpoint *L*>记录前，并且其更新在做Checkpoint时都已写到磁盘中，因此不需要对*T*做任何恢复操作，这样可大大减少恢复工作量。
- 如果数据缓冲区及日志缓冲区中缓存的更新数据很多时，就会导致系统长时间不能接受事务处理，这对响应时间要求较严格的系统来说是不可忍受的。为避免这种中断，可使用模糊检查点(fuzzy checkpoint)技术，允许在做检查点的同时接受数据库更新操作。

基于检验点恢复方法

- 图10-20是系统崩溃时的不同事务状态类型，其中t_c为完成最近检查点时刻，t_f为故障发生时刻。

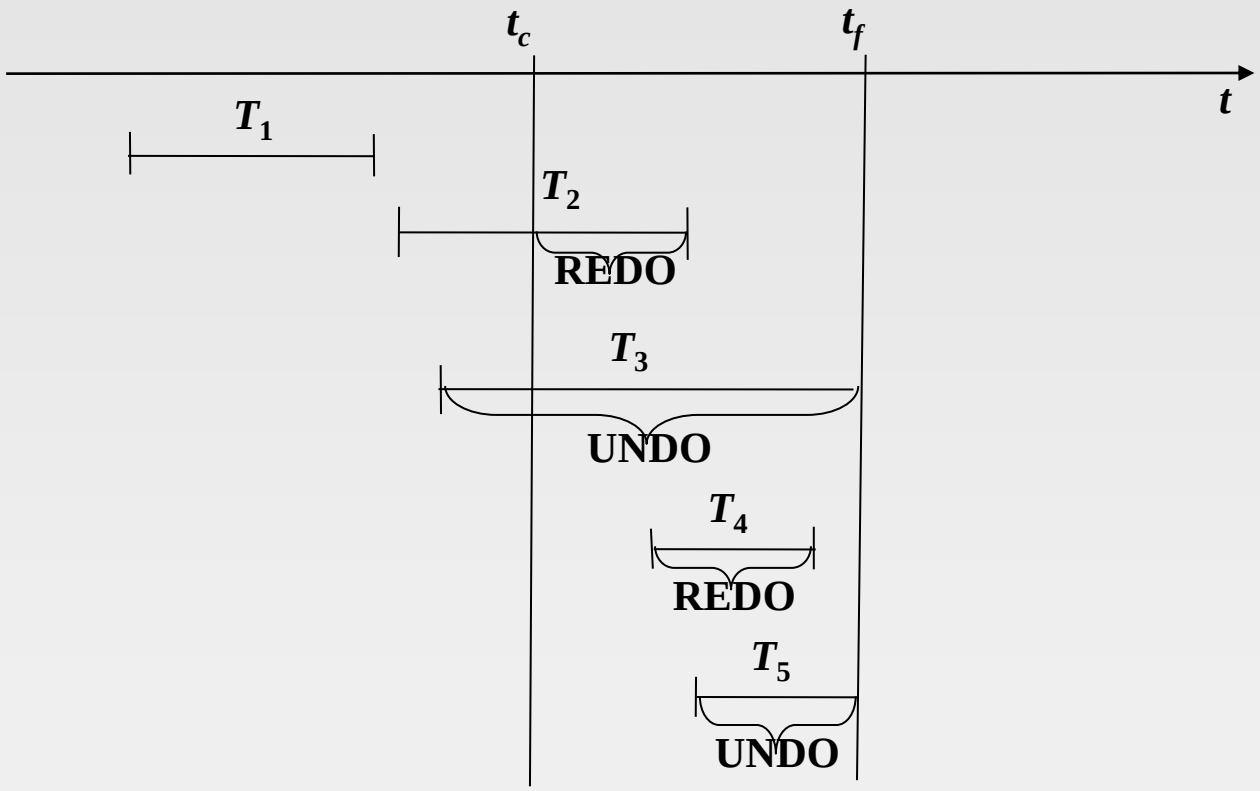


图10-20 系统崩溃时不同状态事务的不同恢复处理

基于检验点恢复方法

- T_1类事务的更新在做检查点之前已经写到磁盘上，故不用做任何恢复操作。

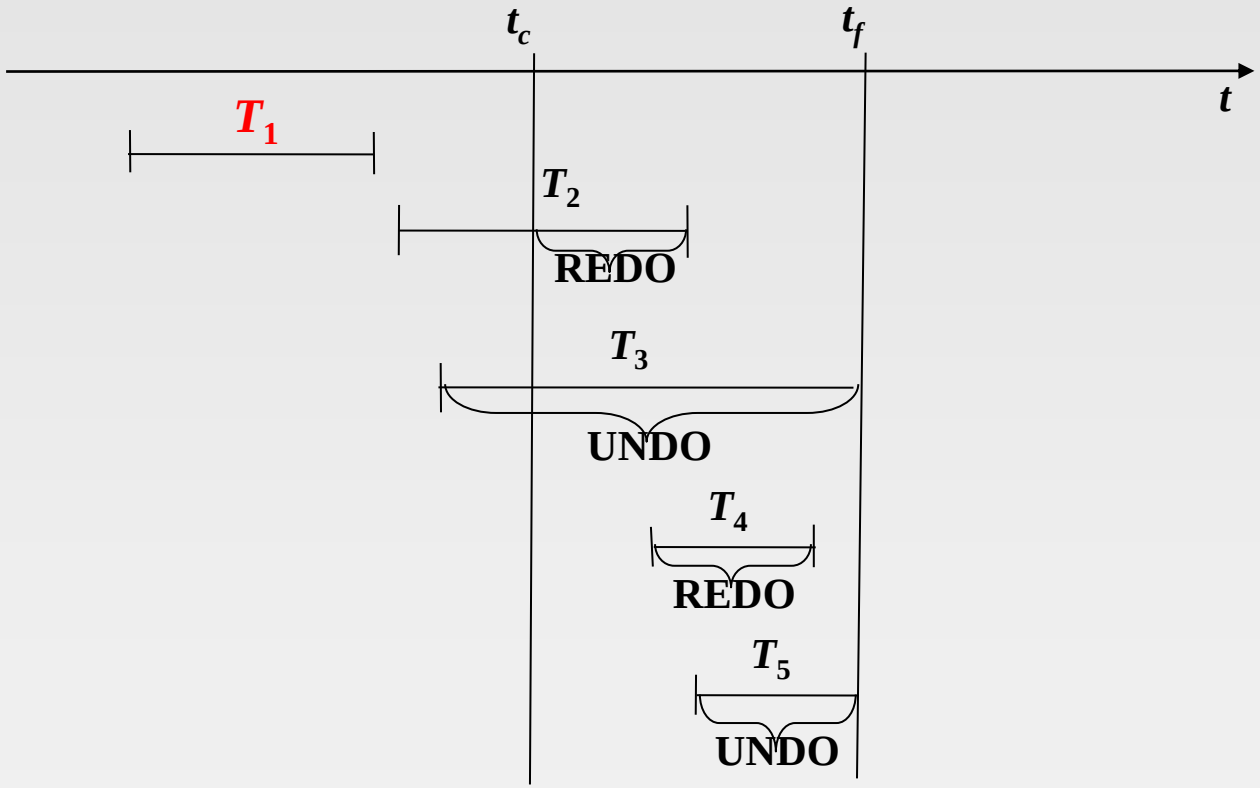


图10-20 系统崩溃时不同状态事务的不同恢复处理

基于检验点恢复方法

- T_2类事务在t_c前的更新已写到磁盘，故重做时只需根据t_c之后的日志记录进行REDO即可。

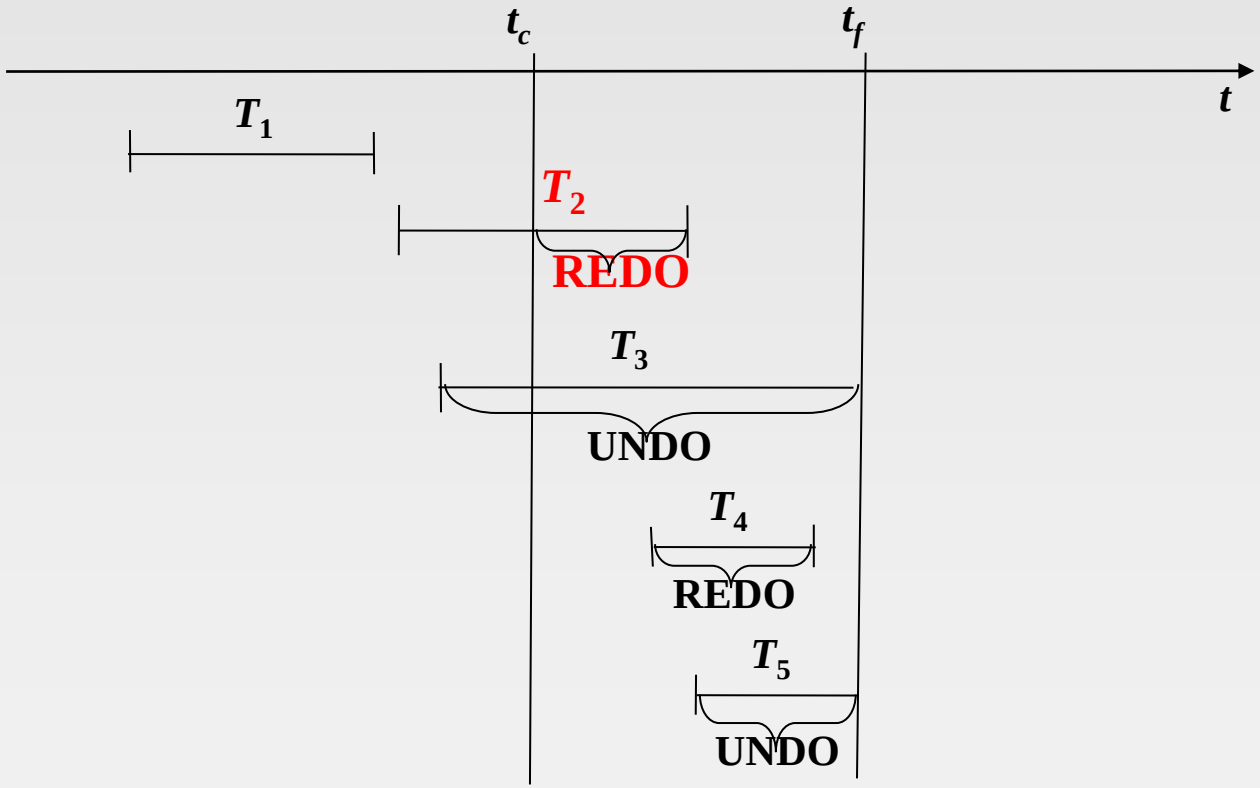


图10-20 系统崩溃时不同状态事务的不同恢复处理

基于检验点恢复方法

- T_3类事务为在t_c之前开始且在t_f之前仍未结束的事务，这类事务需要全部撤销。

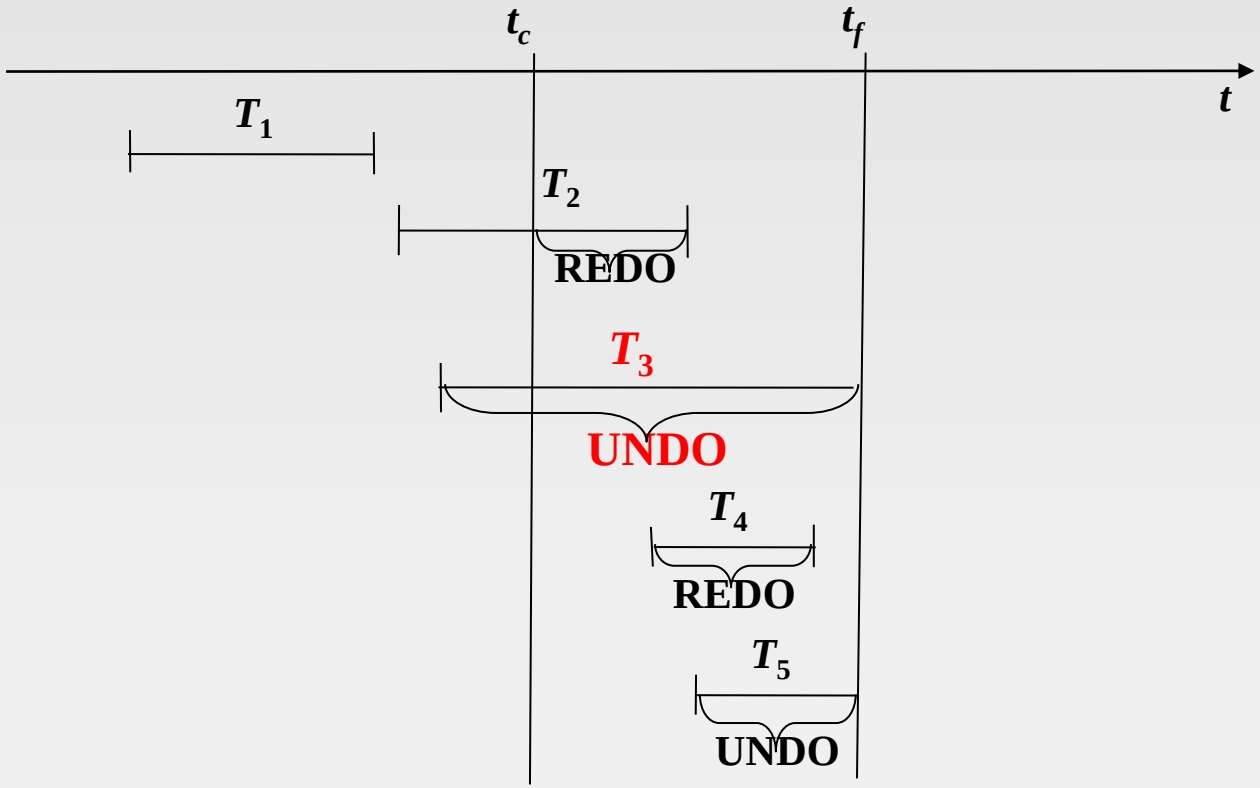


图10-20 系统崩溃时不同状态事务的不同恢复处理

基于检验点恢复方法

- T_4类事务为在t_c之后开始且在t_f之前已完成的事务，这类事务需要全部重做。

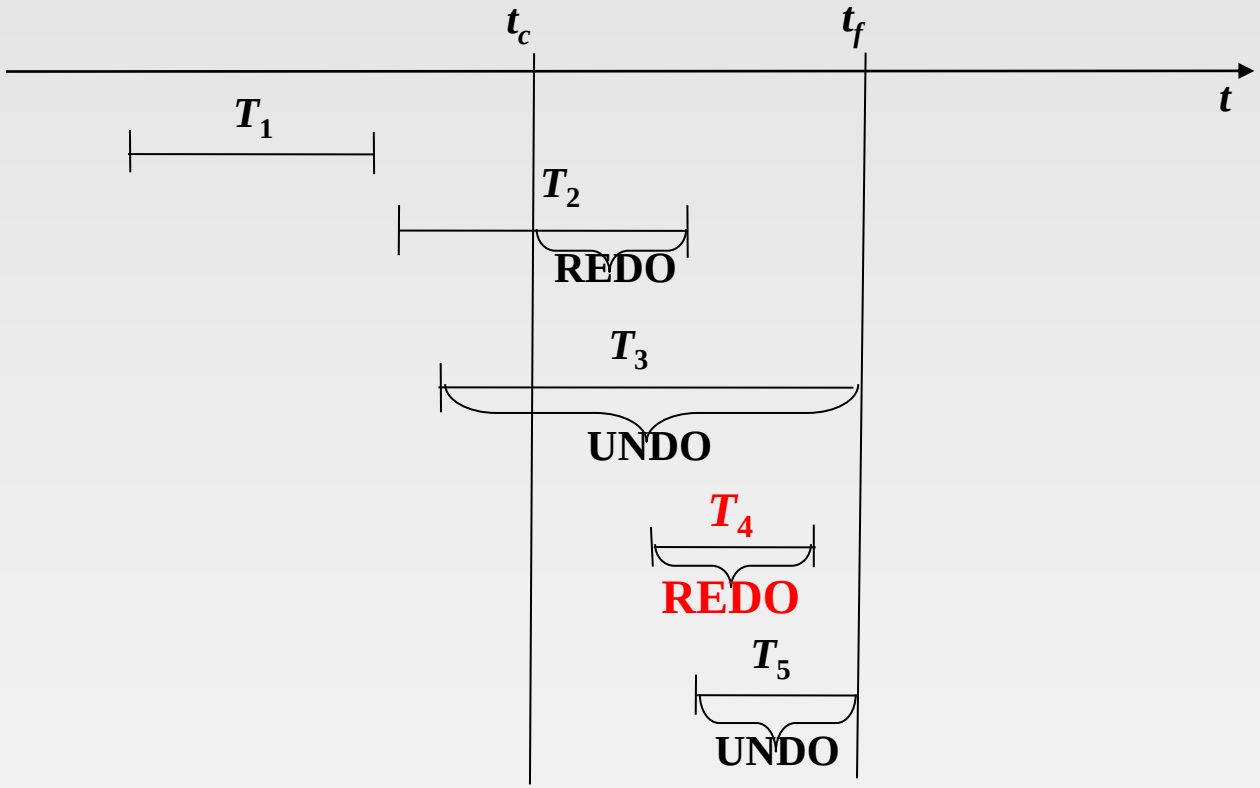


图10-20 系统崩溃时不同状态事务的不同恢复处理

基于检验点恢复方法

- T_5类事务为在t_c之后开始且在t_f之前仍未完成的事务，这类事务应全部撤销。

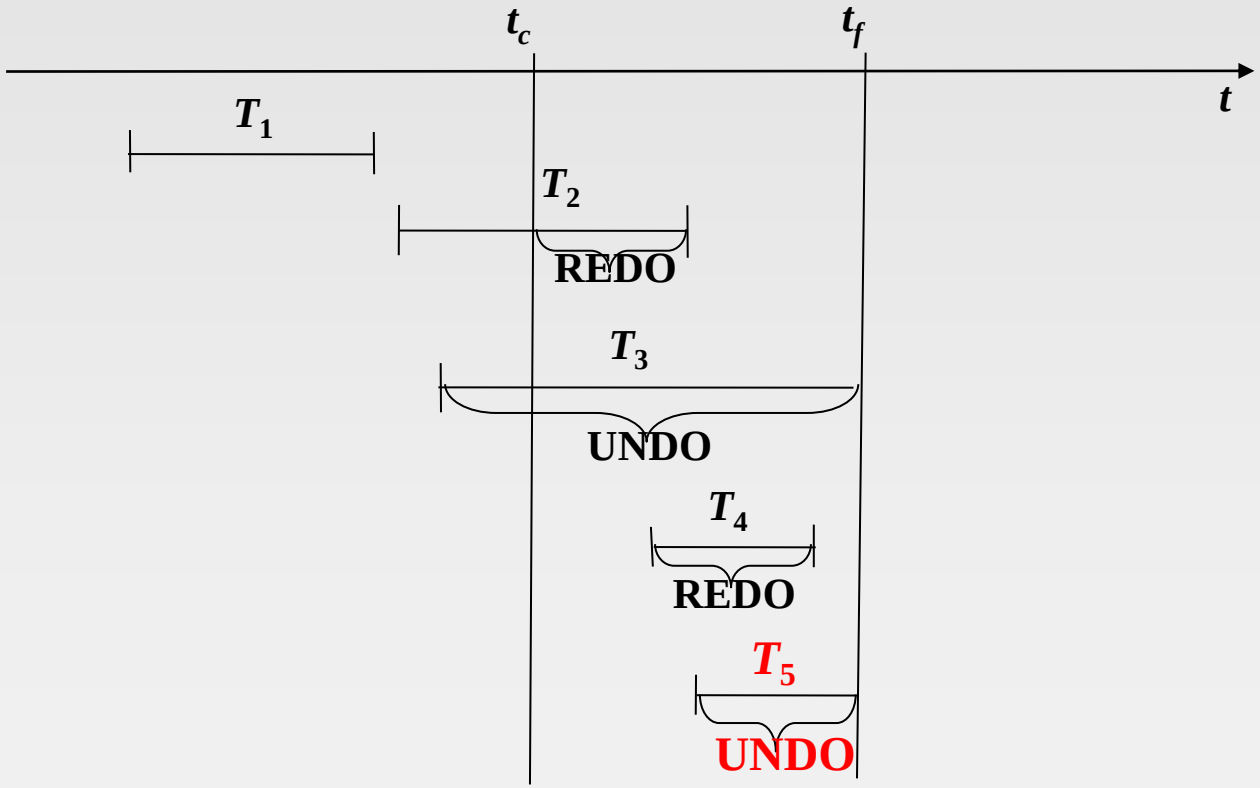


图10-20 系统崩溃时不同状态事务的不同恢复处理

检查点技术

- [例10.11] 系统崩溃时日志文件记录内容如图10-21所示，试写出系统重启后恢复处理的步骤及恢复操作(指UNDO, REDO操作)，并指明*A*, *B*, *C*, *D*恢复后的值分别是多少？

- **分析阶段**：从最后一次检查点开始顺向扫描日志，确定重做事务集REDO-set和撤销事务集UNDO-set。
  - 将既有<*T*, START>又有<*T*, COMMIT>日志记录的事务*T*加入REDO-set;
  - 将只有<*T*, START>没有<*T*, COMMIT>日志记录的事务*T*加入UNDO-set。

<T_0, START>
<T_1, START>
<T_0, *A*, 2, 12>
<T_2, START>
<T_0, COMMIT>
<T_1, *C*, 6, 16>
<T_2, *B*, 4, 14>
<Checkpoint {T_1, T_2}>
<T_1, *A*, 12, 20>
<T_3, START>
<T_1, COMMIT>
<T_2, *B*, 14, 40>
<T_3, *A*, 20, 60>
<T_4, START>
<T_3, *D*, 8, 18>
<T_3, COMMIT>
<T_4, *C*, 16, 30>

图10-21 日志文件

■ 分析过程如下：

	UNDO-set	REDO-set
➢ <Checkpoint {T_1, T_2}>	{T_1, T_2}	{ }
➢ <T_3, START>	{T_1, T_2, T_3}	{ }
➢ <T_1, COMMIT>	{T_2, T_3}	{T_1}
➢ <T_4, START>	{T_2, T_3, T_4}	{T_1}
➢ <T_3, COMMIT>	{T_2, T_4}	{T_1, T_3}

■ **分析阶段**：从最后一次**检查点**开始**顺向扫描**日志，确定**重做事务集**REDO-set和**撤销事务集**UNDO-set。

- 将既有<*T*, START>又有<*T*, COMMIT>日志记录的事务*T*加入REDO-set;
- 将只有<*T*, START>没有<*T*, COMMIT>日志记录的事务*T*加入UNDO-set。

<T_0, START>
<T_1, START>
<T_0, *A*, 2, 12>
<T_2, START>
<T_0, COMMIT>
<T_1, *C*, 6, 16>
<T_2, *B*, 4, 14>
<Checkpoint {T_1, T_2}>
<T_1, *A*, 12, 20>
<T_3, START>
<T_1, COMMIT>
<T_2, *B*, 14, 40>
<T_3, *A*, 20, 60>
<T_4, START>
<T_3, *D*, 8, 18>
<T_3, COMMIT>
<T_4, *C*, 16, 30>

图10-21 日志文件

■ 撤销过程如下：

<$T4$，C, 16, 30>：C=16

<$T2$，B, 14, 40>：B=16

<$T2$，B, 4, 14>：B=4

■ 撤销后的结果为：B=4，C=16。

■ **撤销阶段**：首先，从**日志尾反向扫描**日志文件至遇到最后一次**检查点**止，对每一条属于UNDO-set={T_2, T_4}中事务的**更新操作**日志依次执行UNDO操作。

● 其次，对于同时出现在UNDO-set={T_2, T_4}与<Checkpoint {T_1, T_2}>列表中的事务集{T_2}，从最后一次**检查点**开始继续**反向扫描**日志至遇到这些事务的START止，对属于{T_2}中事务的更新操作日志依次执行UNDO操作。

<T_0，START>
<T_1，START>
<T_0，A, 2, 12>
<T_2，START>
<T_0，COMMIT>
<T_1，C, 6, 16>
<T_2，B, 4, 14>
<Checkpoint {T_1, T_2}>
<T_1，A, 12, 20>
<T_3，START>
<T_1，COMMIT>
<T_2，B, 14, 40>
<T_3，A, 20, 60>
<T_4，START>
<T_3，D, 8, 18>
<T_3，COMMIT>
<T_4，C, 16, 30>

图10-21 日志文件

检查点技术

- [例10.11] 系统崩溃时日志文件记录内容如图10-21所示，试写出系统重启后恢复处理的步骤及恢复操作(指UNDO, REDO操作)，并指明*A*, *B*, *C*, *D*恢复后的值分别是多少？
- **重做阶段**：从最后一次检查点开始顺向扫描日志，对每一条属于REDO-set={T_1, T_3}中事务的更新操作日志依次执行REDO操作。

- **重做过程如下：**
  - <T_1，*A*, 12, 20>：*A*=20
  - <T_3，*A*, 20, 60>：*A*=60
  - <T_3，*D*, 8, 18>：*D*=18
- **重做后的结果为：*A*=60，*D*=18。**

<T_0，START>
<T_1，START>
<T_0，*A*, 2, 12>
<T_2，START>
<T_0，COMMIT>
<T_1，*C*, 6, 16>
<T_2，*B*, 4, 14>
<Checkpoint {T_1, T_2}>
<T_1，*A*, 12, 20>
<T_3，START>
<T_1，COMMIT>
<T_2，*B*, 14, 40>
<T_3，*A*, 20, 60>
<T_4，START>
<T_3，*D*, 8, 18>
<T_3，COMMIT>
<T_4，*C*, 16, 30>

图10-21 日志文件

检查点技术

- [例10.11] 系统崩溃时日志文件记录内容如图

恢复完成后的结果为：A=60，B=4，C=16，D=18。

明A, B, C, D恢复后的值分别是多少？

- **重做阶段**：从最后一次检查点开始顺向扫描日志，对每一条属于REDO-set={T_1, T_3}中事务的更新操作日志依次执行REDO操作。

- **重做过程如下：**

  <T_1，A, 12, 20>：A=20

  <T_3，A, 20, 60>：A=60

  <T_3，D, 8, 18>：D=18

- **重做后的结果为：A=60，D=18。**

<T_0，START>
<T_1，START>
<T_0，A, 2, 12>
<T_2，B, 4, 14>
<Checkpoint {T_1, T_2}>
<T_1，A, 12, 20>
<T_3，START>
<T_1，COMMIT>
<T_2，B, 14, 40>
<T_3，A, 20, 60>
<T_4，START>
<T_3，D, 8, 18>
<T_3，COMMIT>
<T_4，C, 16, 30>

图10-21 日志文件

数据备份

- **数据备份**(Backup)是数据库管理系统用来进行介质故障恢复的常用方法。它是由DBA周期性地将整个数据库的内容复制到其他外存储器上(通常为大容量的磁带或磁鼓)保存起来。

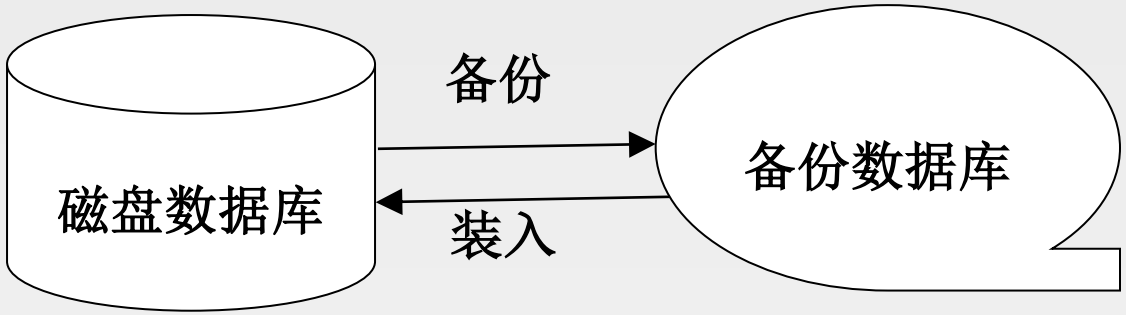


图10-20 数据备份与装入

静态备份和动态备份

- 数据备份操作可分为**静态备份**和**动态备份**。
- **静态备份**是在系统中无运行事务时进行的备份操作。优点是简单。但由于备份必须等待用户事务结束后才能进行，而新的事务必须等待备份结束后才能执行，因此会降低数据库的可用性。
- **动态备份**是指备份操作与用户事务的执行并发进行，备份期间允许对数据库进行存取或修改。动态备份克服了静态备份的缺点，它不用等待正在运行的用户事务结束，也不会影响新事务的运行。但它不能保证副本中的数据正确有效。

全备份与增量备份

- 具体进行数据备份时可以分**全备份**和**增量备份**。
- **全备份**是指每次备份全部数据库，而**增量备份**是只备份上次备份后更新过的数据。
- 从恢复角度看，一般说来使用全备份得到的后备副本进行恢复会更方便些。但如果数据库很大，事务处理又十分频繁，则增量备份方式更实用更有效。
- 不管是那种形式的备份及介质故障的恢复处理，都需要DBA介入。由于备份又是十分耗费时间和资源的，不能频繁进行。所以DBA应该根据数据库使用情况确定适当的备份周期和备份方法。