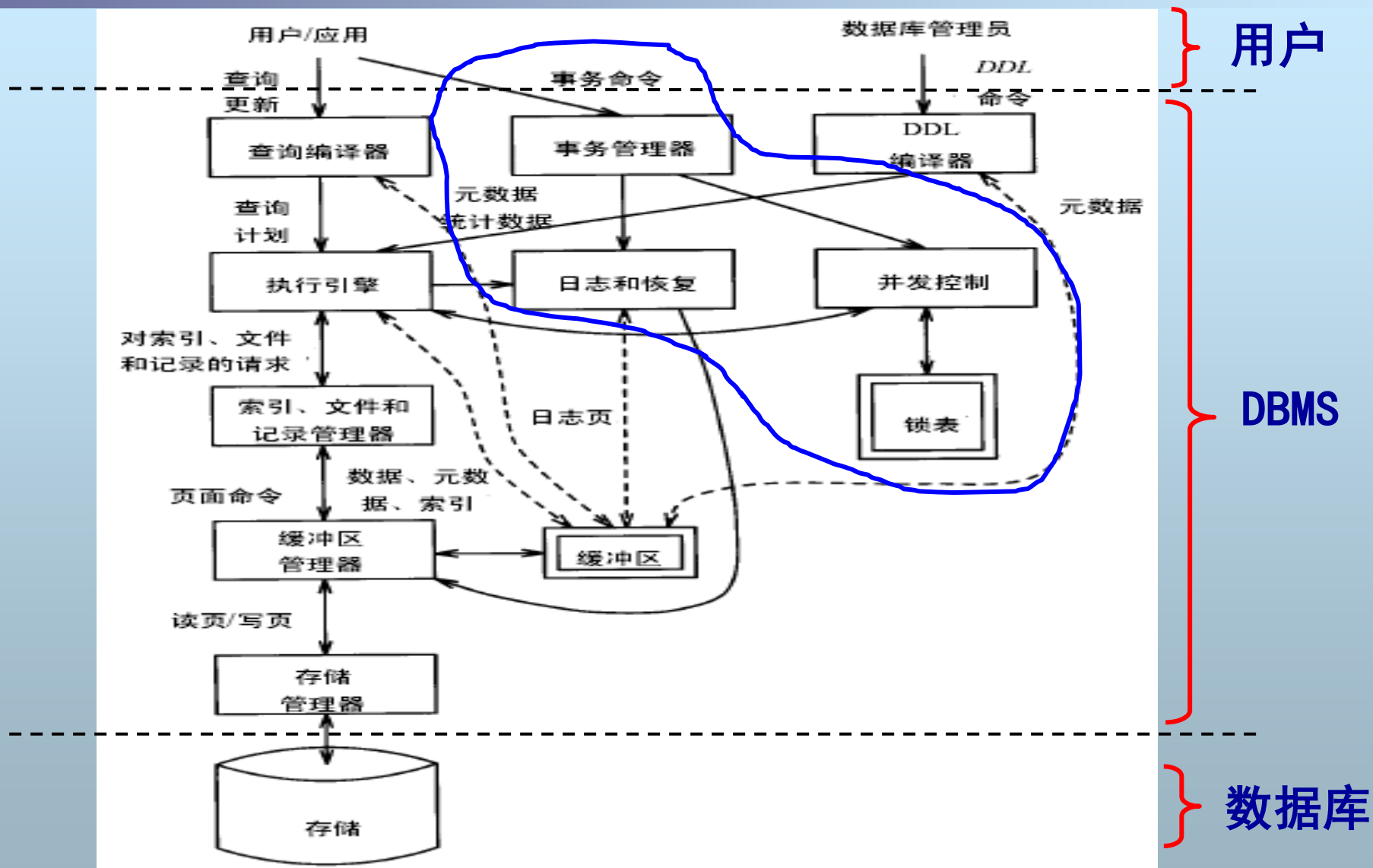


Log & Recovery

A thick, wavy orange line that spans the width of the slide, positioned below the title.

回顾



Databases protection

- 数据库保护：排除和防止各种对数据库的干扰破坏，确保数据安全可靠，以及在数据库遭到破坏后尽快地恢复
- 数据库保护通过四个方面来实现
 - 数据库的恢复技术 [this chapter]
 - ◆ Deal with failure
 - 并发控制技术 [Chp. 18 & 19]
 - ◆ Deal with data sharing
 - 完整性控制技术
 - ◆ Enable constraints
 - 安全性控制技术
 - ◆ Authorization and authentication

主要内容

- 数据库的一致性
- 事务的状态及原语操作
- 数据库系统故障分析
- Undo日志
- Redo日志
- Undo/Redo日志
- Checkpoint
- Log Rotation

一、数据库的一致性

Integrity or consistency constraints

- **Predicates data must satisfy**
- **Examples:**
 - **x is key of relation R**
 - **$x \rightarrow y$ holds in R**
 - **$\text{Domain}(x) = \{\text{Red, Blue, Green}\}$**

一、数据库的一致性

- **Consistent state**: satisfies all integrity constraints
- **Consistent DB**: DB in consistent state

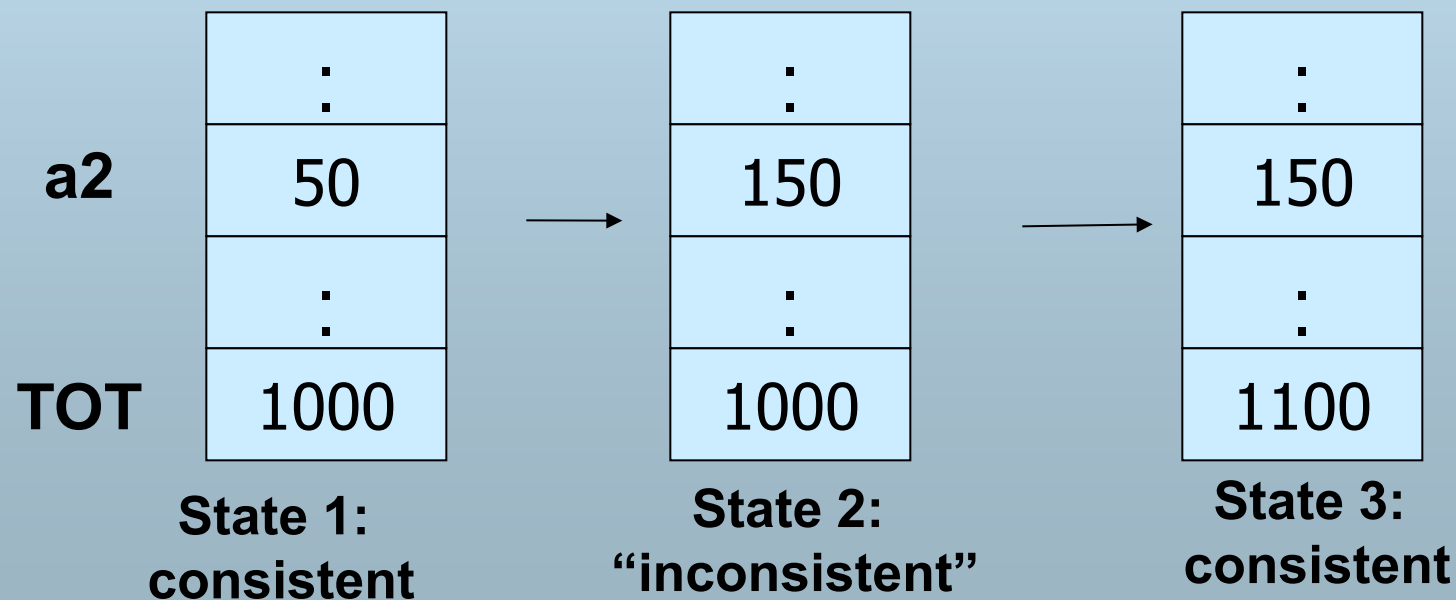
一、数据库的一致性

■ DB will not always satisfy constraints

Example: $a1 + a2 + \dots + a_n = \text{TOT}$ (constraint)

Transaction: Deposit \$100 in a2: $a2 \leftarrow a2 + 100$

$\text{TOT} \leftarrow \text{TOT} + 100$

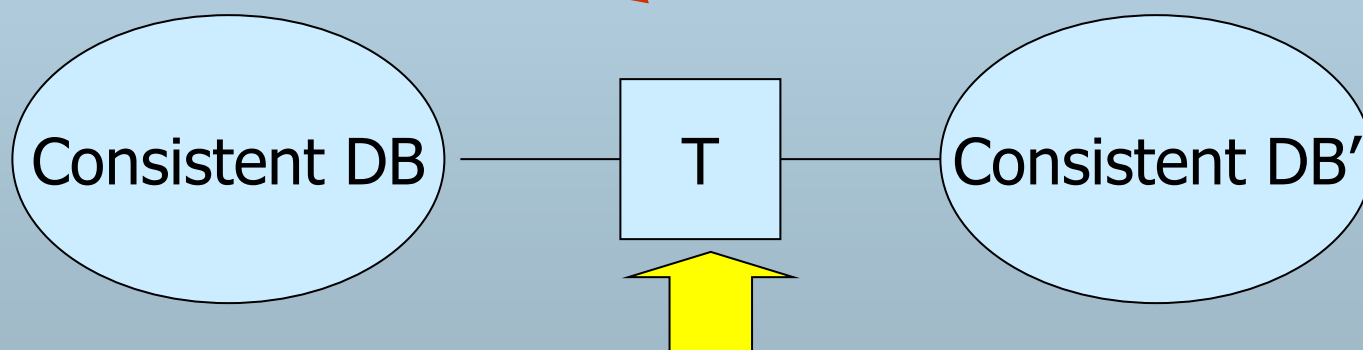


一、数据库的一致性

consistency of transaction

事务的ACID性质

Atomicity, Consistency, Isolation, Durability



但事务内部不保证DB的一致性

二、事务的状态及原语操作

■ 事务(transaction)

- 一个不可分割的操作序列，其中的操作要么都做，要么都不做

1、事务

■ 事务的例子

- 银行转帐：A帐户转帐到B帐户100元。该处理包括了两个更新步骤
 - ◆ $A = A - 100$
 - ◆ $B = B + 100$
- 这两个操作是不可分的：要么都做，要么都不作

1、事务

■ 事务的ACID性质

- 原子性 **Atomicity**
- 一致性 **Consistency**
- 隔离性 **Isolation**
- 持久性 **Durability**

2、事务的状态 [in logs]

■ <Start T>

- Transaction T has started

■ <Commit T>

- T has finished successfully and all modifications are all reflected to disks

■ <Abort T>

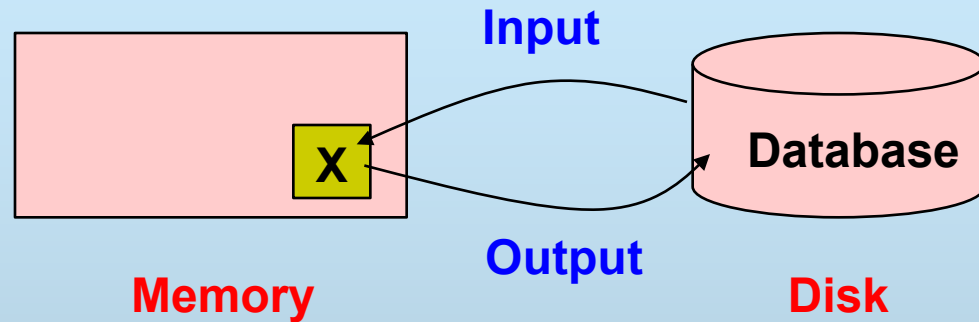
- T has been terminated and all modifications have been canceled

3、事务的原语操作

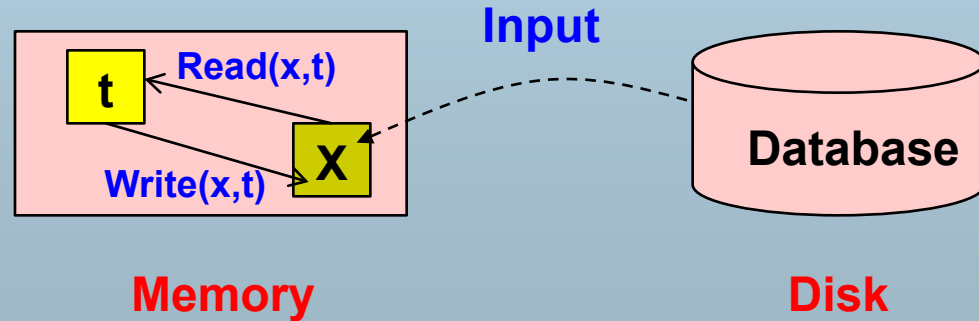
- **Input (x):** disk block with $x \rightarrow$ memory
- **Output (x):** buffer block with $x \rightarrow$ disk
- **Read (x,t):** do input(x) if necessary
 $t \leftarrow$ value of x in buffer
- **Write (x,t):** do input(x) if necessary
value of x in buffer $\leftarrow t$

3、事务的原语操作

- **Input (x)**
- **Output (x)**



- **Read (x,t)**
- **Write (x,t)**



4、事务例子

A bank transfer

```
T1:  Read (A,t);  
      t ← t - 100;  
      Write (A,t);  
      Read (B,t);  
      t ← t + 100;  
      Write (B,t);  
      Output (A);  
      Output (B);
```

In general:

T1: **r1(A)w1(A)r1(B)w1(B)**

5、SQL对事务的支持

- **SQL标准提供了三个语句，允许应用程序声明事务和控制事务**
 - **Begin Transaction**
 - **Commit Transaction**
 - **Rollback Transaction**
- **Oracle**
 - **Commit或Commit Work**
 - **Rollback或Rollback Work**
 - **没有Begin Transaction语句，一旦连接数据库建立会话，就认为是一个事务的开始**

Oracle中的事务设置

```
SQL>set autocommit=on
```

```
--设置为每次语句执行都自动Commit
```

```
SQL>set autocommit=off
```

```
SQL>update student set age=age-1;
```

```
SQL>rollback; -- 取消前面的更新操作
```

```
SQL>update student set age=age-1;
```

```
SQL>commit; --提交, 修改生效不能再回退
```

Oracle PL/SQL中使用事务

- 若在程序中使用了Rollback或Commit，则自动将Rollback或Commit之前的操作视为事务。

```
Create or Replace Procedure Transfer
(sender varchar2, receiver varchar2, amount number, val varchar2)
AS
    a Number:=0;
    b Number:=0;
    exp Exception;
Begin
    Select balance into a Where ID=sender;
    Select count(*) Into b From accounts where ID=receiver and name=val;
    If a<=amount or b=0 then
        Rollback;
    Else
        Update account Set balance=balance+100 where ID='B';
        Update account Set balance=balance-100 where ID='A';
    End If
    Commit;
End;
```

三、数据库系统故障分析

■ Consistency of DB 可能由于故障而被破坏

- 事务故障
- 介质故障
- 系统故障

1、事务故障

■ 发生在单个事务内部的故障

- 可预期的事务故障：即应用程序可以发现的故障，如转帐时余额不足。由应用程序处理
- 非预期的事务故障：如运算溢出、死锁等，导致事务被异常中止。应用程序无法处理此类故障，由系统进行处理

可预期事务故障的处理

```
Create or Replace Procedure Transfer
(sender varchar2, receiver varchar2, amount number, val varchar2)
AS
    a Number:=0;
    b Number:=0;
    exp Exception;
Begin
    Select balance into a Where ID=sender;
    Select count(*) Into b From accounts where ID=receiver and name=val;
    If a<=amount or b=0 then
        Raise exp; --生成一个异常;
    Else
        Update account Set balance=balance+100 where ID='B';
        Update account Set balance=balance-100 where ID='A';
    End If
    Commit;
EXCEPTION
    When exp Then
        Rollback;
        Raise_Application_Error(-20001, '余额不足或接收方账号有误');
End;
```

2、介质故障

- **硬故障（Hard Crash）**，一般指磁盘损坏
 - 导致磁盘数据丢失，破坏整个数据库

3、系统故障

- **系统故障：软故障（Soft Crash），由于OS、DBMS软件问题或断电等问题导致内存数据丢失，但磁盘数据仍在**
 - **影响所有正在运行的事务，破坏事务状态，但不破坏整个数据库**

1. 未完成事务对数据库的更新可能已部分写入数据库
2. 已提交事务对数据库的更新可能还留在缓冲区还没来得及写入数据库

4、数据库系统故障恢复策略

■ 目的

- 恢复DB到一致状态

■ 基本原则

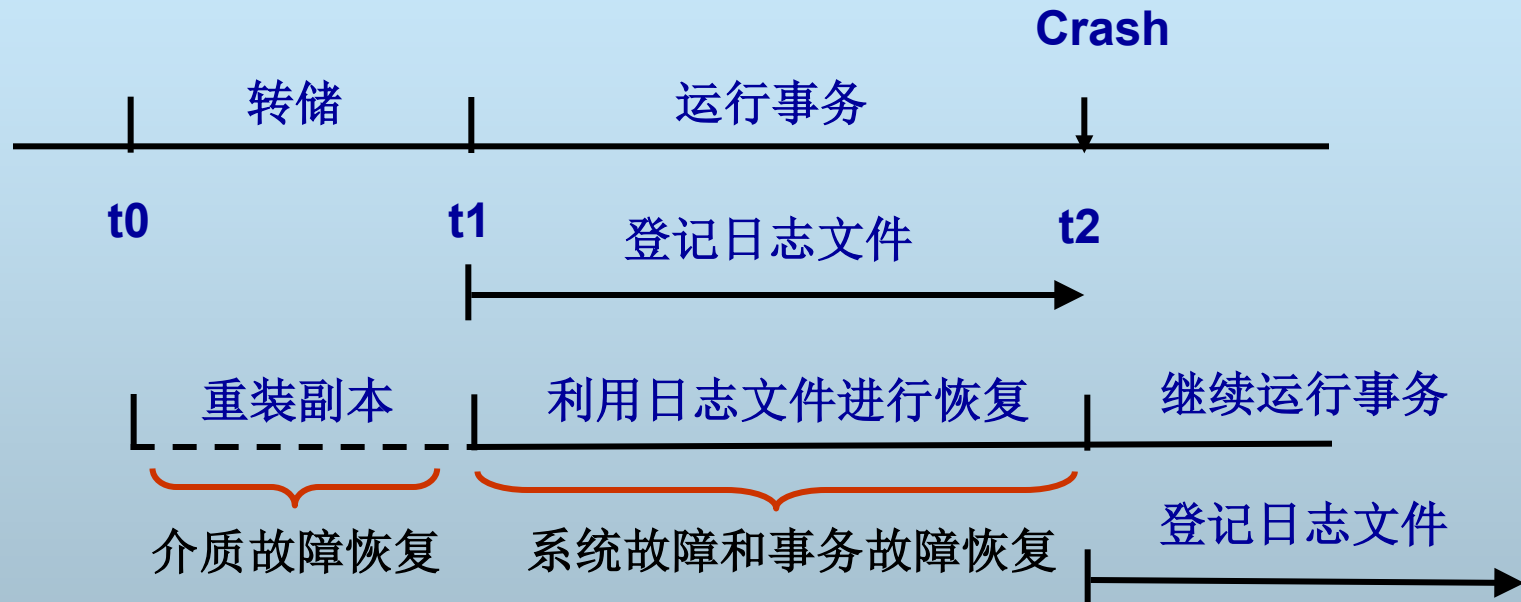
- 冗余 (Redundancy)

■ 实现方法

- 定期转储整个数据库
- 建立事务日志 (log)
- 通过备份和日志进行恢复

4、数据库系统故障恢复策略

The recovery process



当发生故障时:

- (1) 若是介质故障, 则首先重装副本
- (2) 利用日志进行事务故障恢复和系统故障恢复, 一直恢复到故障发生点

四、Undo日志

- 事务日志记录了所有更新操作的具体细节
 - **Undo日志、Redo日志、Undo/Redo日志**
- 日志文件的登记严格按事务执行的时间次序
- **Undo日志文件中的内容**
 - 事务的开始标记（<Start T>）
 - 事务的结束标记（<Commit, T>或<Abort T>）
 - 事务的更新操作记录，一般包括以下内容
 - ◆ 执行操作的事务标识
 - ◆ 操作对象
 - ◆ 更新前值（插入为空）

1、Undo日志规则

- 事务的每一个修改操作都生成一个日志记录 $\langle T, x, \text{old-value} \rangle$
- 在 x 被写到磁盘之前，对应该修改的日志记录必须已被写到磁盘上
- 当事务的所有修改结果都已写入磁盘后，将 $\langle \text{Commit}, T \rangle$ 日志记录写到磁盘上

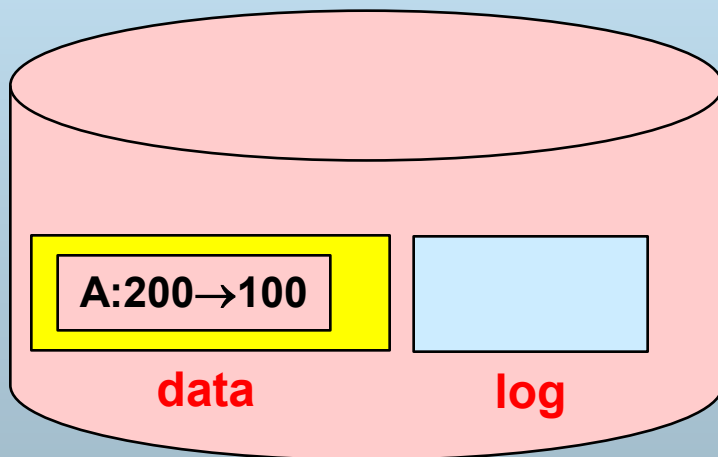


Write Ahead Logging (WAL日志) 先写日志

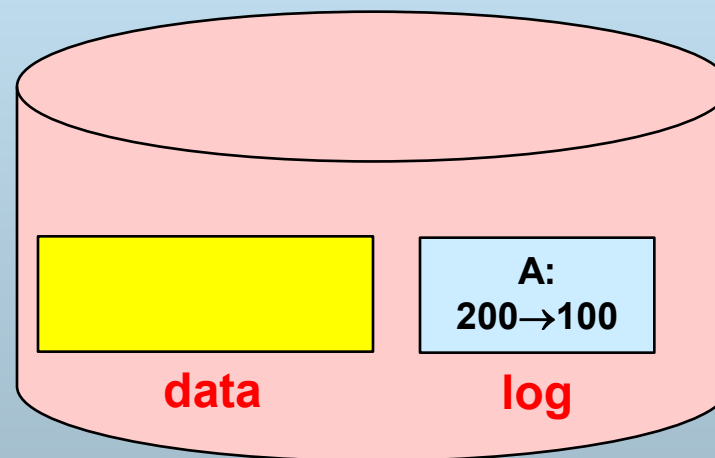
Why WAL?

■ 每次写操作需要执行两次write

- Write data
- Write log

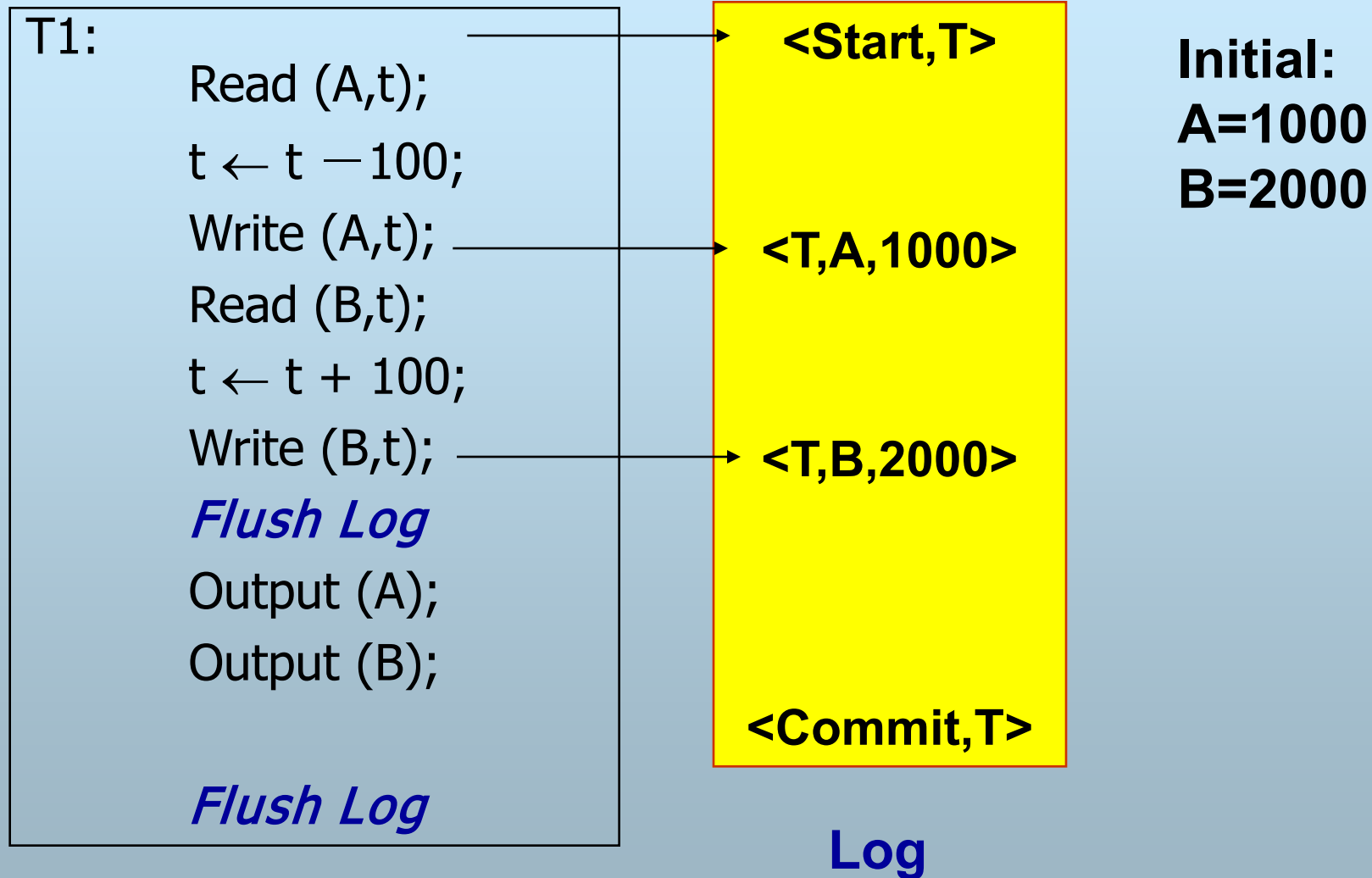


如果先写data后写log，一旦写log之前发生故障，DBMS无法恢复

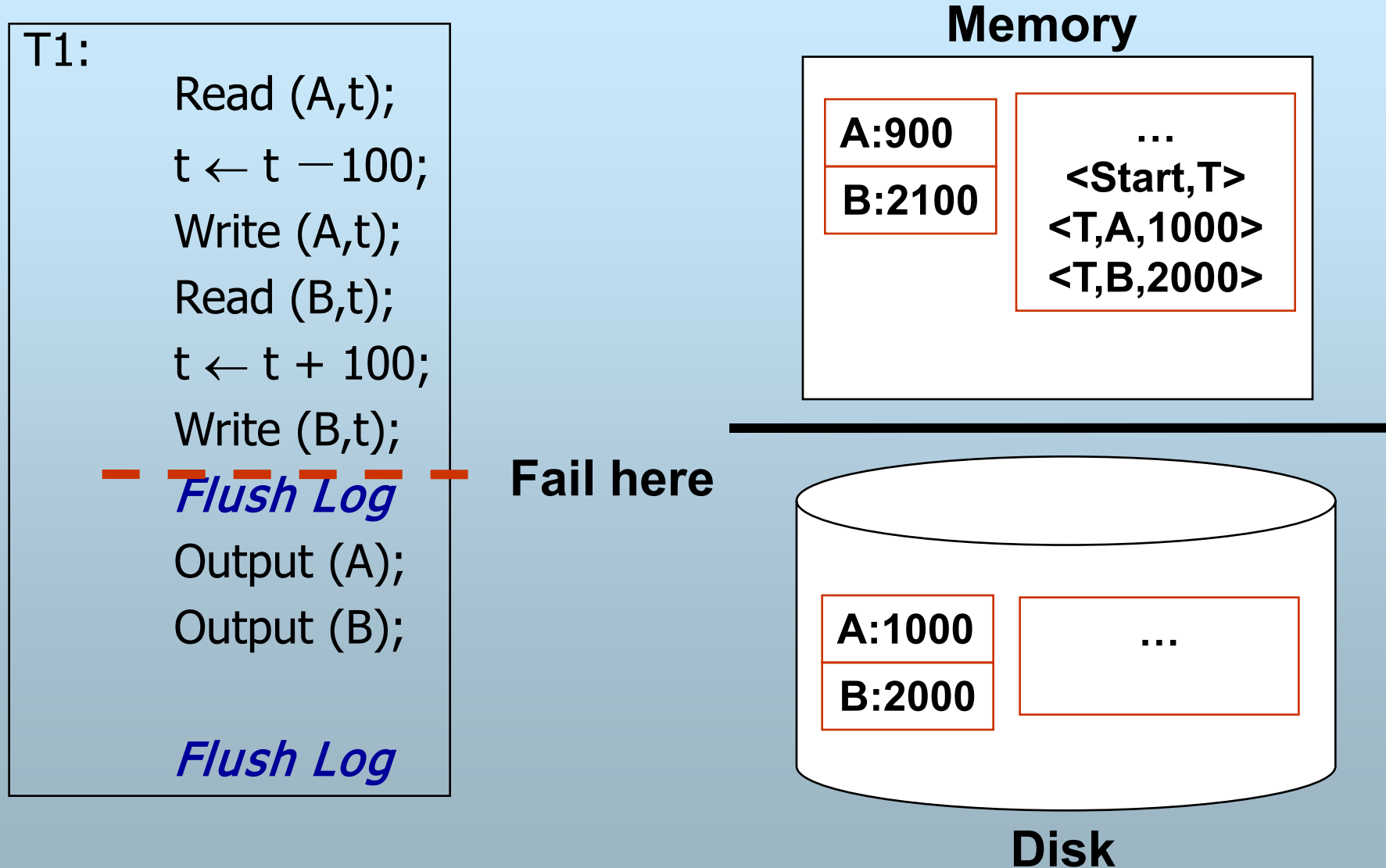


如果先写log后写data，即使写data之前发生故障，DBMS可以借助log知道此时数据库处于不一致状态并恢复

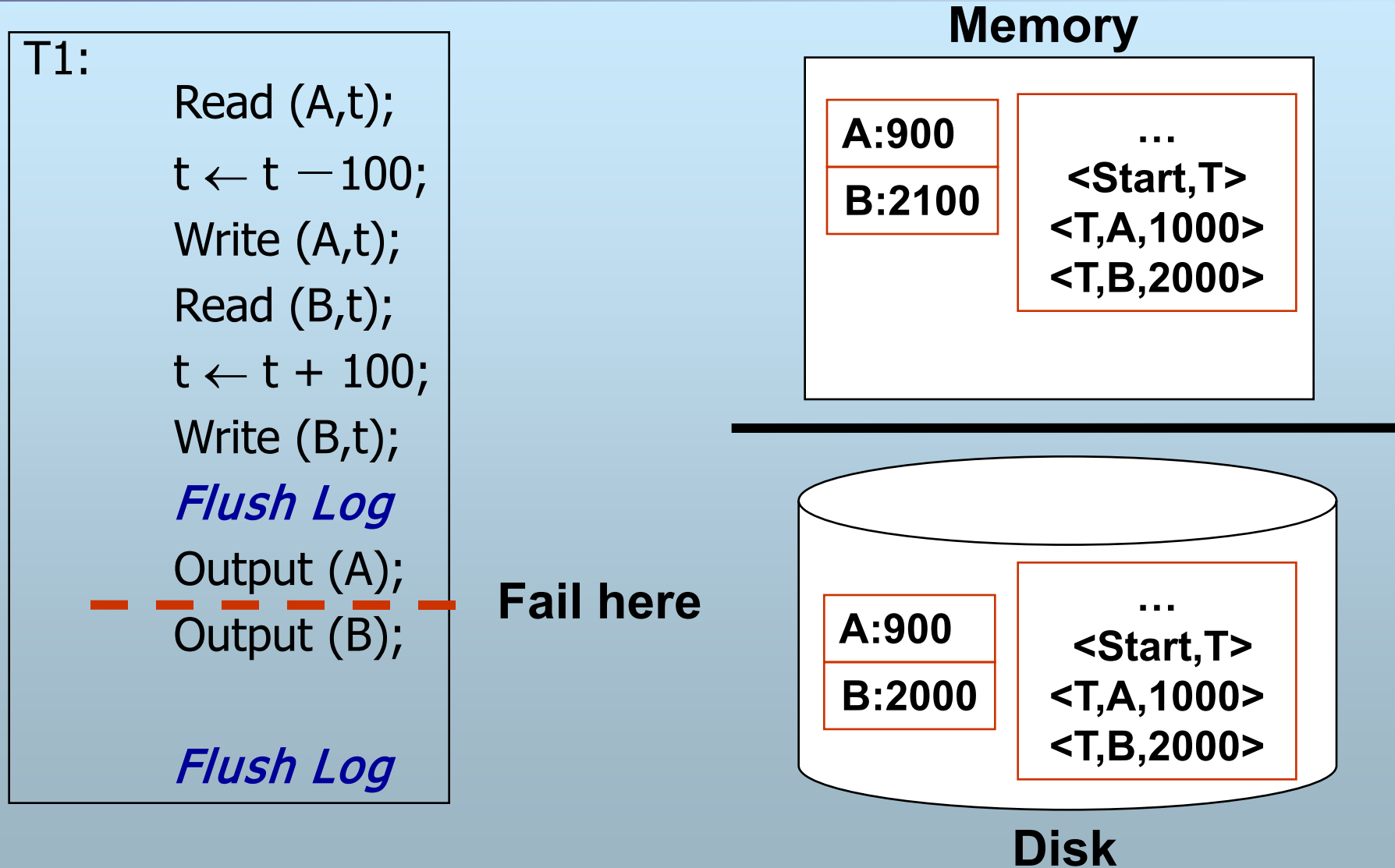
1、Undo日志规则



1、Undo日志规则



1、Undo日志规则



1、Undo日志规则

T1:

```
Read (A,t);  
t ← t - 100;  
Write (A,t);  
Read (B,t);  
t ← t + 100;  
Write (B,t);  
Flush Log  
Output (A);  
Output (B);
```

Flush Log

Fail here

Memory

A:900

B:2100

...

<Start,T>

<T,A,1000>

<T,B,2000>

<Commit,T>

A:900

B:2100

...

<Start,T>

<T,A,1000>

<T,B,2000>

Disk

1、Undo日志规则

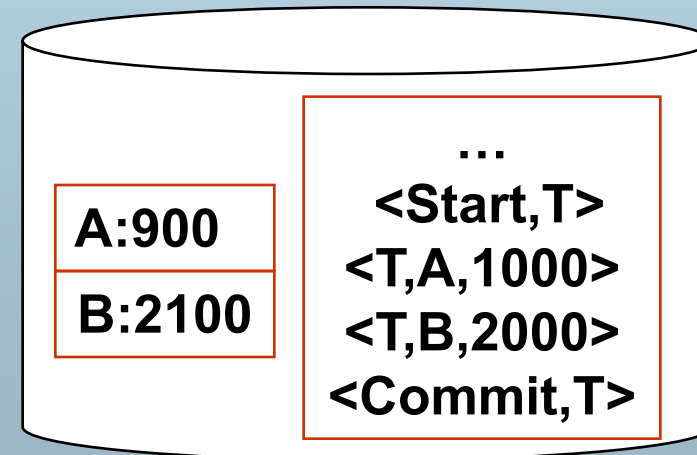
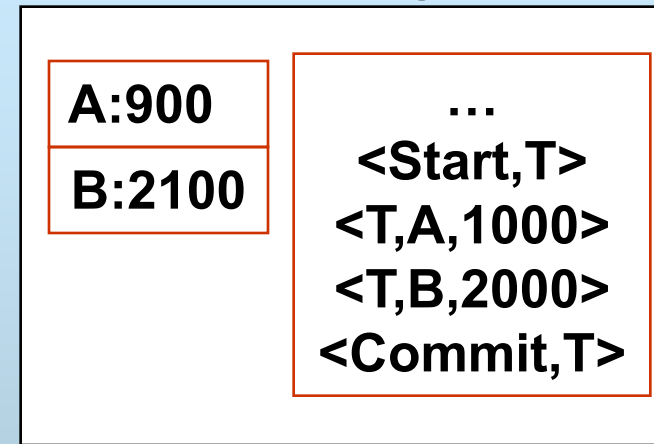
T1:

```
Read (A,t);  
t ← t - 100;  
Write (A,t);  
Read (B,t);  
t ← t + 100;  
Write (B,t);  
Flush Log  
Output (A);  
Output (B);
```

Flush Log

Success!

Memory



Disk

2、基于Undo日志的恢复

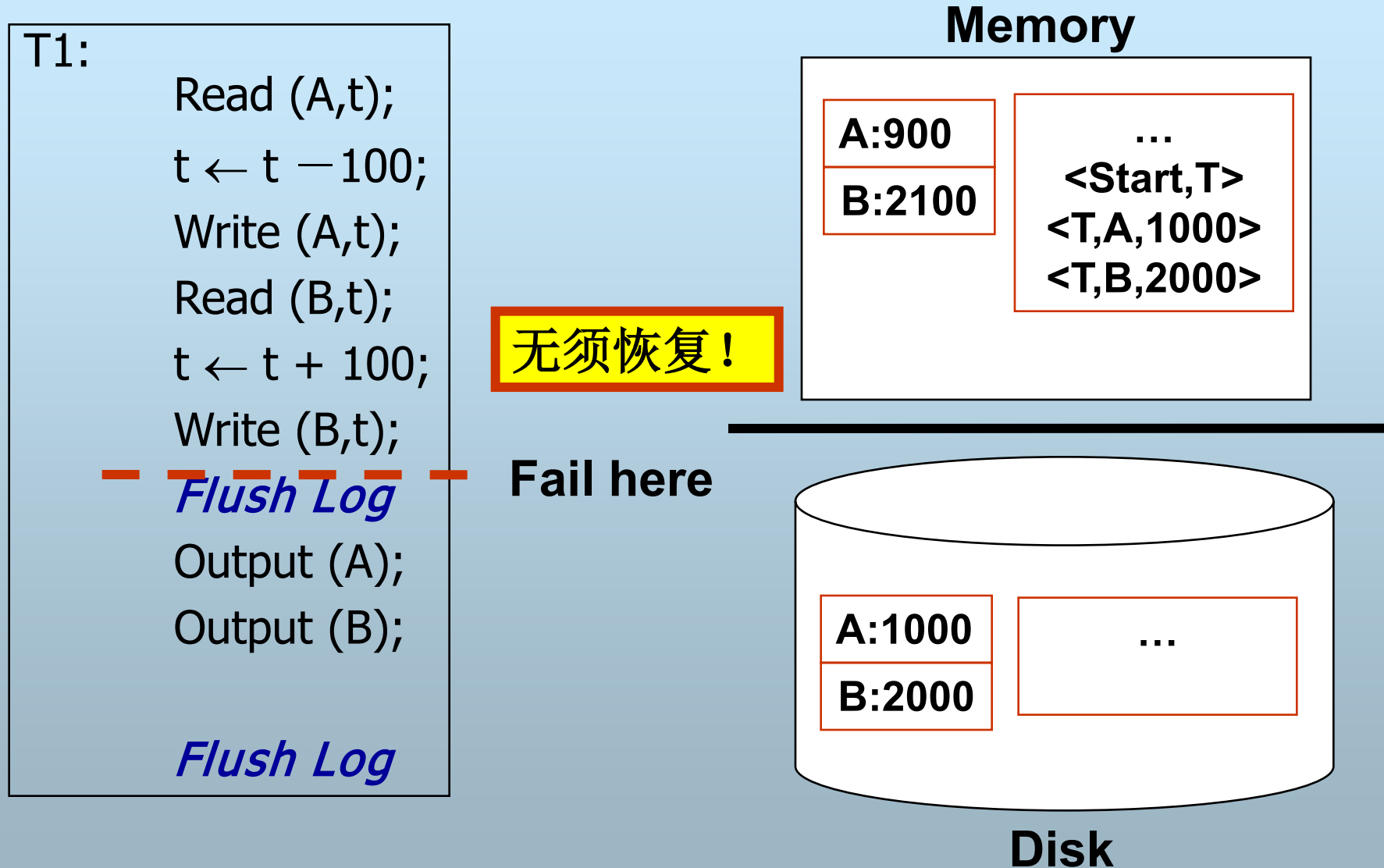
The recovery process



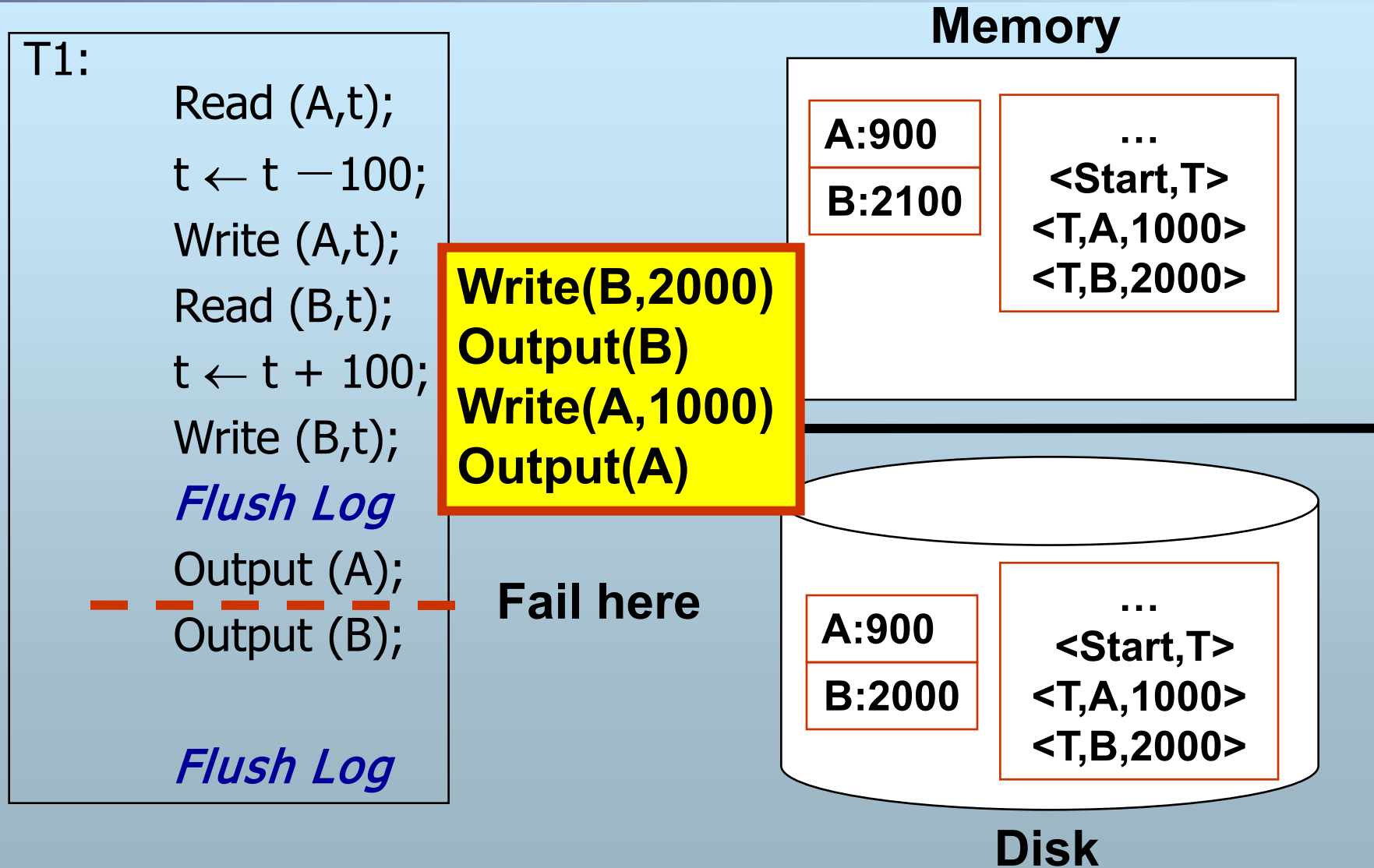
2、基于Undo日志的恢复

- 从头扫描日志，找出所有没有<Commit,T>或<Abort,T>的所有事务，放入一个事务列表L中
- 从尾部开始扫描日志记录<T,x,v>,如果 $T \in L$ ，则
 - write (X, v)
 - output (X)
- For each $T \in L$ do
 - write <Abort,T > to log

2、基于Undo日志的恢复



2、基于Undo日志的恢复



2、基于Undo日志的恢复

恢复后的Undo日志

<Start,T>
<T,A,1000>
<T,B,2000>
<Abort T>

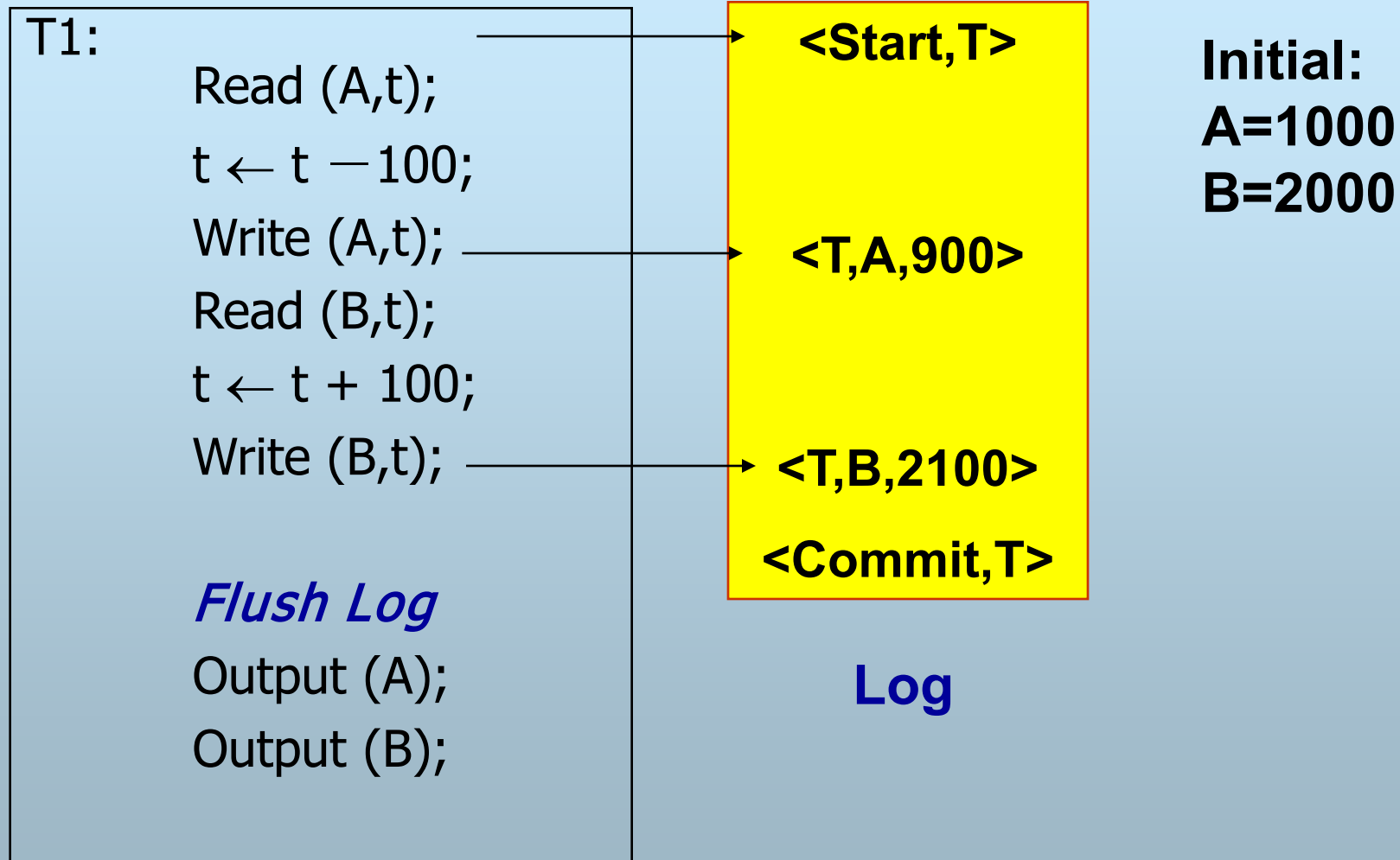
3、Undo日志总结

- $\langle T, x, v \rangle$ 记录修改前的旧值
- 写入 $\langle \text{Commit}, T \rangle$ 之前必须先将数据写入磁盘
- 恢复时忽略已提交事务，只撤销未提交事务
 - 有 $\langle \text{Commit}, T \rangle$ 的事务肯定已写回磁盘

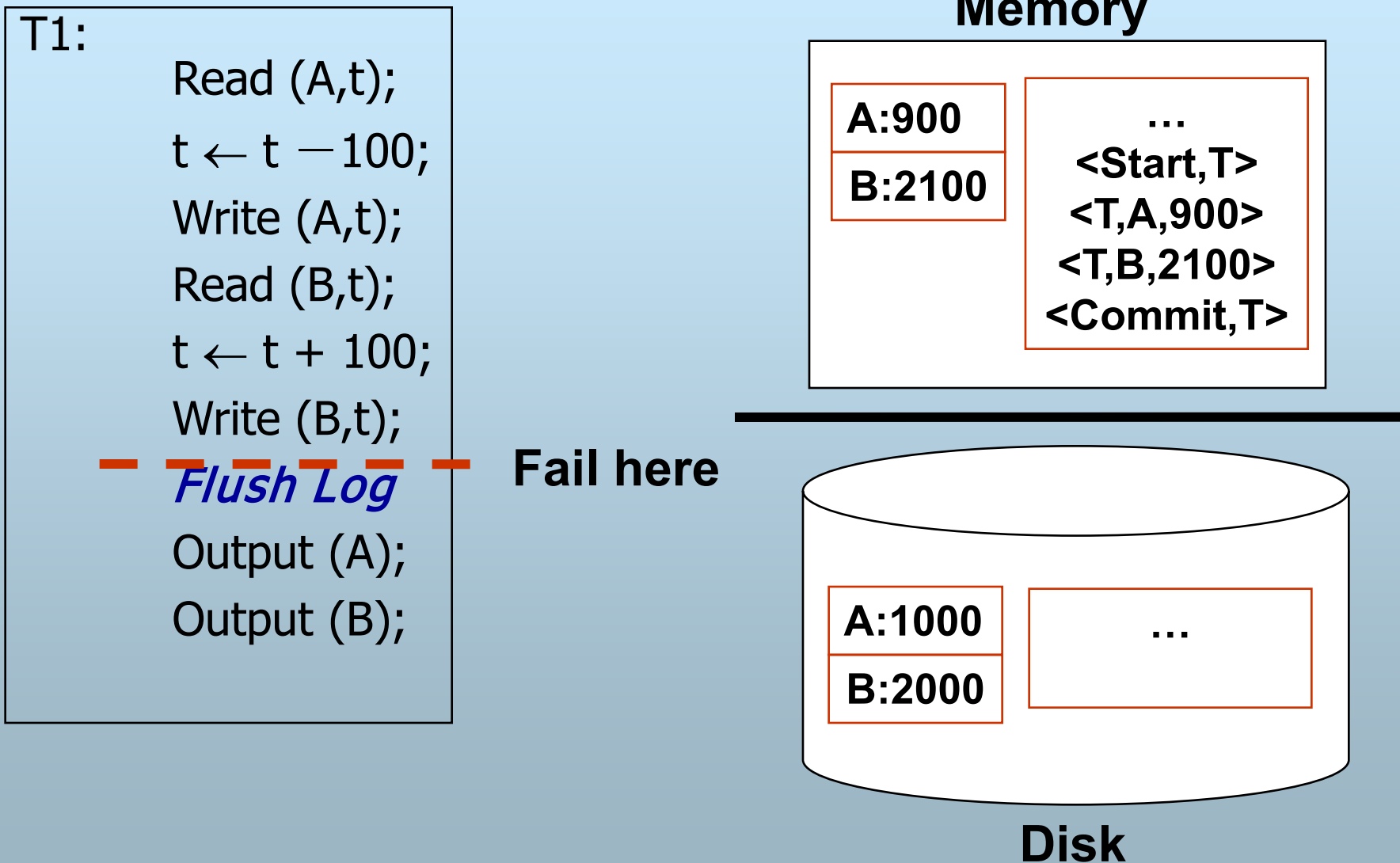
五、Redo日志

- 在 x 被写到磁盘之前，对应该修改的Redo日志记录必须已被写到磁盘上 (WAL)
- 在数据写回磁盘前**先写** $\langle \text{Commit}, T \rangle$ 日志记录
- 日志中的数据修改记录
 - $\langle T, x, v \rangle$ - - **Now v is the new value**

1、Redo日志规则



1、Redo日志规则

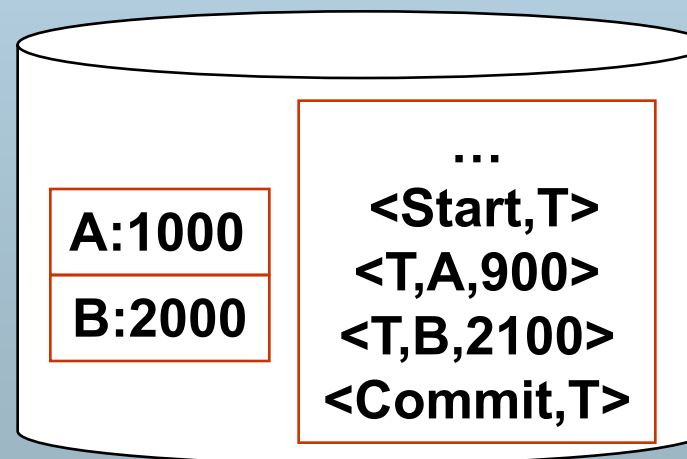
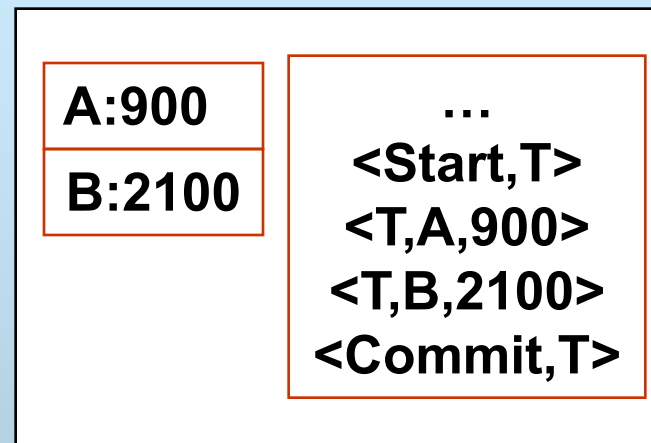


1、Redo日志规则

T1:

```
Read (A,t);  
t ← t - 100;  
Write (A,t);  
Read (B,t);  
t ← t + 100;  
Write (B,t);  
Flush Log  
-----  
Output (A);  
Output (B);
```

Fail here



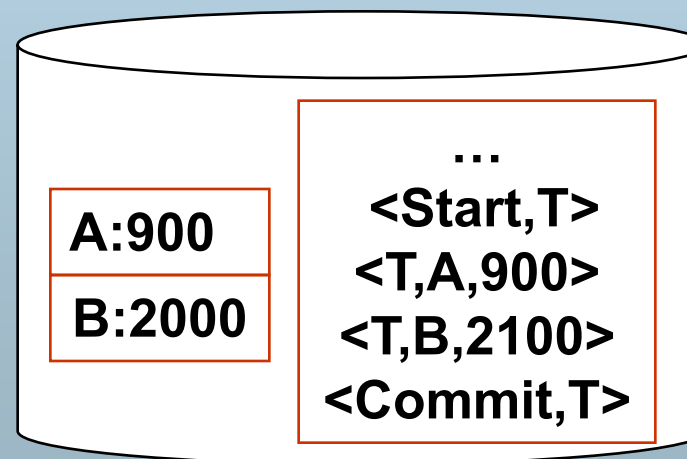
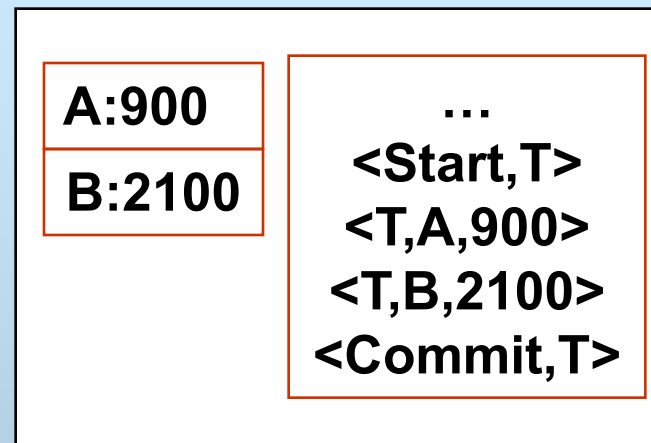
Disk

1、Redo日志规则

T1:

```
Read (A,t);  
t ← t - 100;  
Write (A,t);  
Read (B,t);  
t ← t + 100;  
Write (B,t);  
Flush Log  
Output (A);  
Output (B);
```

Fail here



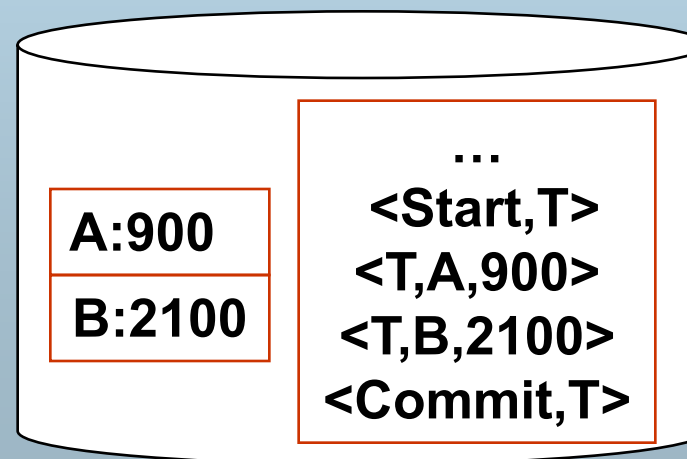
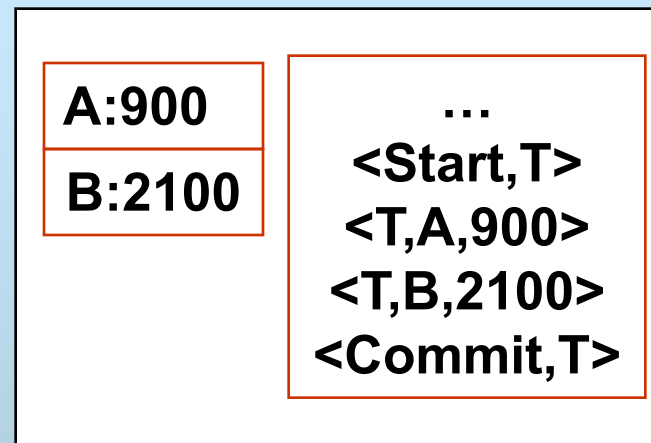
Disk

1、Redo日志规则

T1:

```
Read (A,t);  
t ← t - 100;  
Write (A,t);  
Read (B,t);  
t ← t + 100;  
Write (B,t);  
Flush Log  
Output (A);  
Output (B);
```

Fail here



Disk

2、基于Redo日志的恢复

- 从头扫描日志，找出所有有 $\langle \text{Commit}, T \rangle$ 的事务，放入一个事务列表L中
- 从首部开始扫描日志记录 $\langle T, x, v \rangle$, 如果 $T \in L$ ，则
 - **write (X, v)**
 - **output (X)**
- **For each $T \in L$ do**
 - **write $\langle \text{Abort}, T \rangle$ to log**

2、基于Redo日志的恢复

■ 恢复的基础

- 没有 $\langle \text{Commit}, T \rangle$ 记录的操作必定没有改写磁盘数据，因此在恢复时可以不理会
 - ◆ Differ from Undo logging
- 有 $\langle \text{Commit}, T \rangle$ 记录的结果可能还未写回磁盘，因此在恢复时要Redo
 - ◆ Still differ from Undo logging

Undo vs. Redo

Undo: 立即更新（乐观）

T1:
Read (A,t);
 $t \leftarrow t - 100$;
Write (A,t);
Output (A);
Read (B,t);
 $t \leftarrow t + 100$;
Write (B,t);
Output (B);



内存代价小



恢复代价高

Redo: 延迟更新（悲观）

T1:
Read (A,t);
 $t \leftarrow t - 100$;
Write (A,t);
Read (B,t);
 $t \leftarrow t + 100$;
Write (B,t);
Output (A);
Output (B);



恢复代价小



内存代价高

六、Undo/Redo日志

- 在x被写到磁盘之前，对应该修改的日志记录必须已被写到磁盘上 (WAL)
- 日志中的数据修改记录
 - $\langle T, x, v, w \rangle$
 - - v is the old value, w is the new value

1、基于Undo/Redo日志的恢复

- 正向扫描日志，将 $\langle \text{commit} \rangle$ 的事务放入Redo列表中，将没有结束的事务放入Undo列表
- 反向扫描日志，对于 $\langle T, x, v, w \rangle$ ，若T在Undo列表中，则
 - **Write(x,v); Output(x)**
- 正向扫描日志，对于 $\langle T, x, v, w \rangle$ ，若T在Redo列表中，则
 - **Write(x,w); Output(x)**
- 对于Undo列表中的T，写入 $\langle \text{abort}, T \rangle$

1、基于Undo/Redo日志的恢复

发生故障时的日志

<Start,T1>

<T1,B,2000,1900>

<Start,T2>

<T2,A,1000,900>

<Commit,T1>

<Start,T3>

<T3,C,3000,2000>

<T3,B,1900,1800>

<Commit,T2>

<Start,T4>

<T4,D,1000,1200>

1. Undo列表 {T3,T4}; Redo {T1,T2}

2. Undo
T4: D=1000
T3: B=1900
T3: C=3000

3. Redo
T1:B=1900
T2:A=900

4. Write log
<Abort,T3>
<Abort,T4>

1、基于Undo/Redo日志的恢复

■ 先Undo后Redo

发生故障时的日志

<Start,T1>

<Start,T2>

<T1,A,1000,1200>

<T2,A,1000,1100>

< Commit,T2>

T1要UNDO, T2要REDO

如果先**REDO**, 则**A=1100**; 然后再**UNDO**, **A=1000**。不正确

先**UNDO**, **A=1000**; 然后**REDO**, **A=1100**。正确

七、检查点(checkpoint)

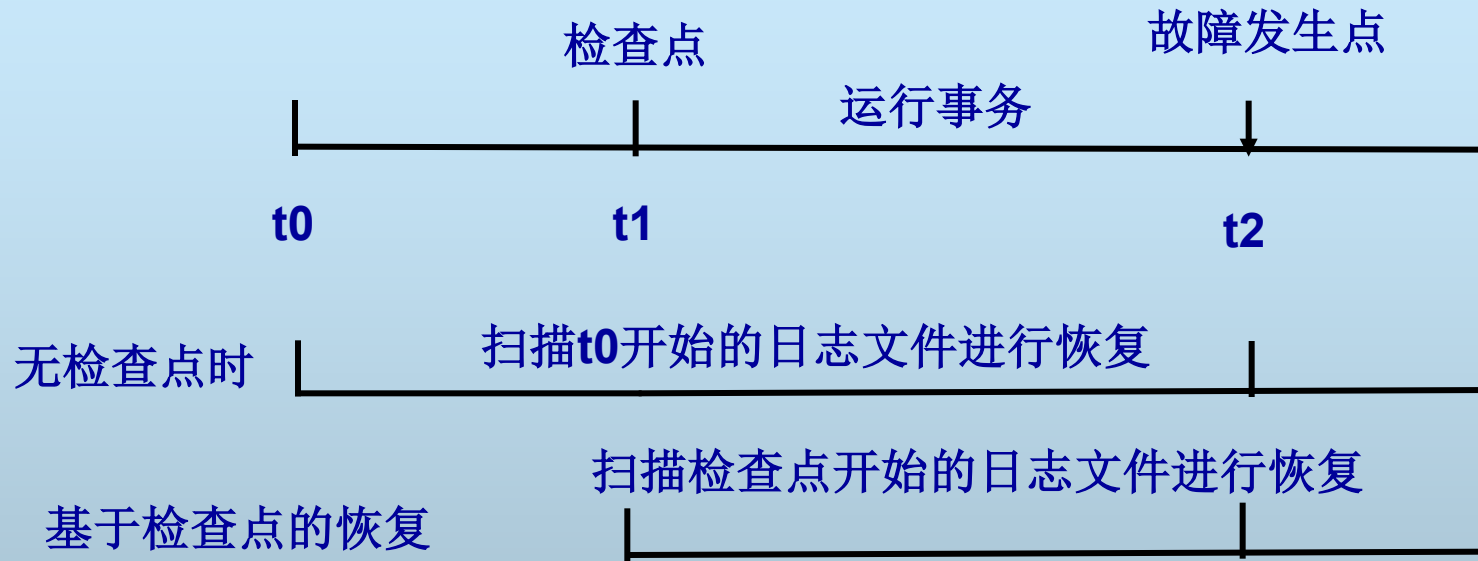
- 当系统故障发生时，必须扫描日志。需要搜索整个日志来确定**UNDO**列表和**REDO**列表
 - 搜索过程太耗时，因为日志文件增长很快
 - 会导致最后产生的**REDO**列表很大，使恢复过程变得很长

1、simple checkpoint

Periodically:

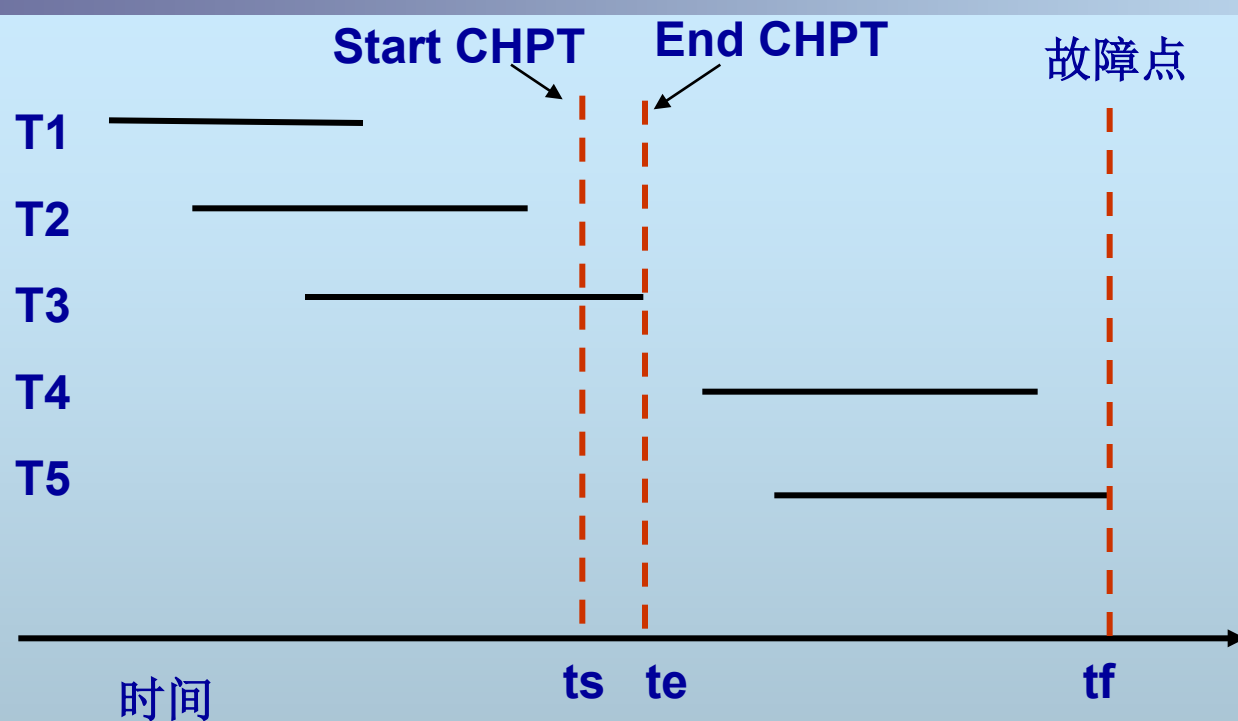
- (1) Do not accept new transactions**
- (2) Wait until all transactions finish (commit/abort)**
- (3) Flush all log records to disk (log)**
- (4) Flush all buffers to disk (DB)**
- (5) Write “checkpoint” record on disk (log)**
- (6) Resume transaction processing**

2、checkpoint-based recovery



检查点技术保证检查点之前的所有**commit**操作的结果已写回数据库，在恢复时不需**REDO**

2、checkpoint-based recovery



恢复时: **UNDO**={T5}, **REDO**={T4}

T1和**T3**由于在检查点之前已**Commit**,
因此不需要**REDO**

Log

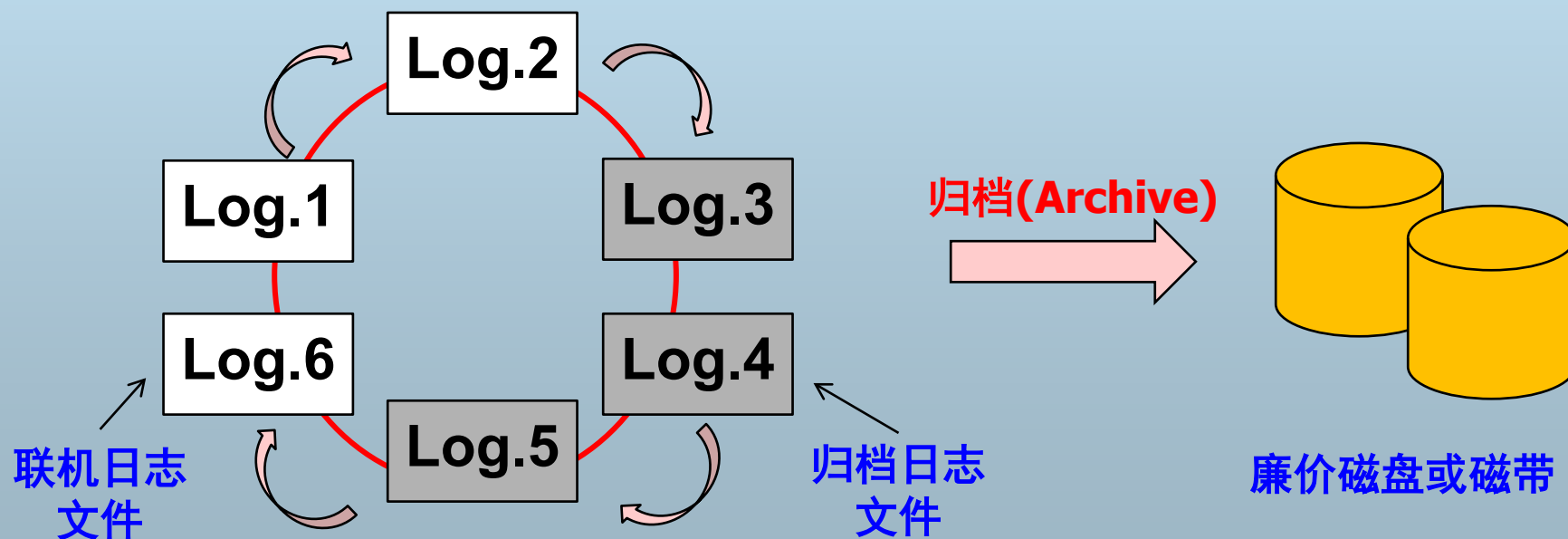
```
<start,T1>
...
<start,T2>
...
<start,T3>
...
<commit,T1>
...
<abort,T2>
...
<commit,T3>
<checkpoint>
<start,T4>
...
<start,T5>
...
<commit,T4>
...
```

2、checkpoint-based recovery

- If new transactions are allowed to start during a checkpoint
 - Non-quietescent checkpoint (非静止检查点)
 - See textbook 17.2.5,17.3.3,17.4.3

八、日志轮转技术

- 数据库日志产生很快，会占用很多的磁盘存储。但大部分日志随时间推移实际上已经失效了
- 采用Log Rotation节省存储



本章小结

- 数据库的一致性
- 事务的状态及原语操作
- 数据库系统故障类型
- 基于Undo日志的恢复过程
- 基于Redo日志的恢复
- 基于Undo/Redo日志的恢复
- 基于Checkpoint的恢复
- 日志轮转