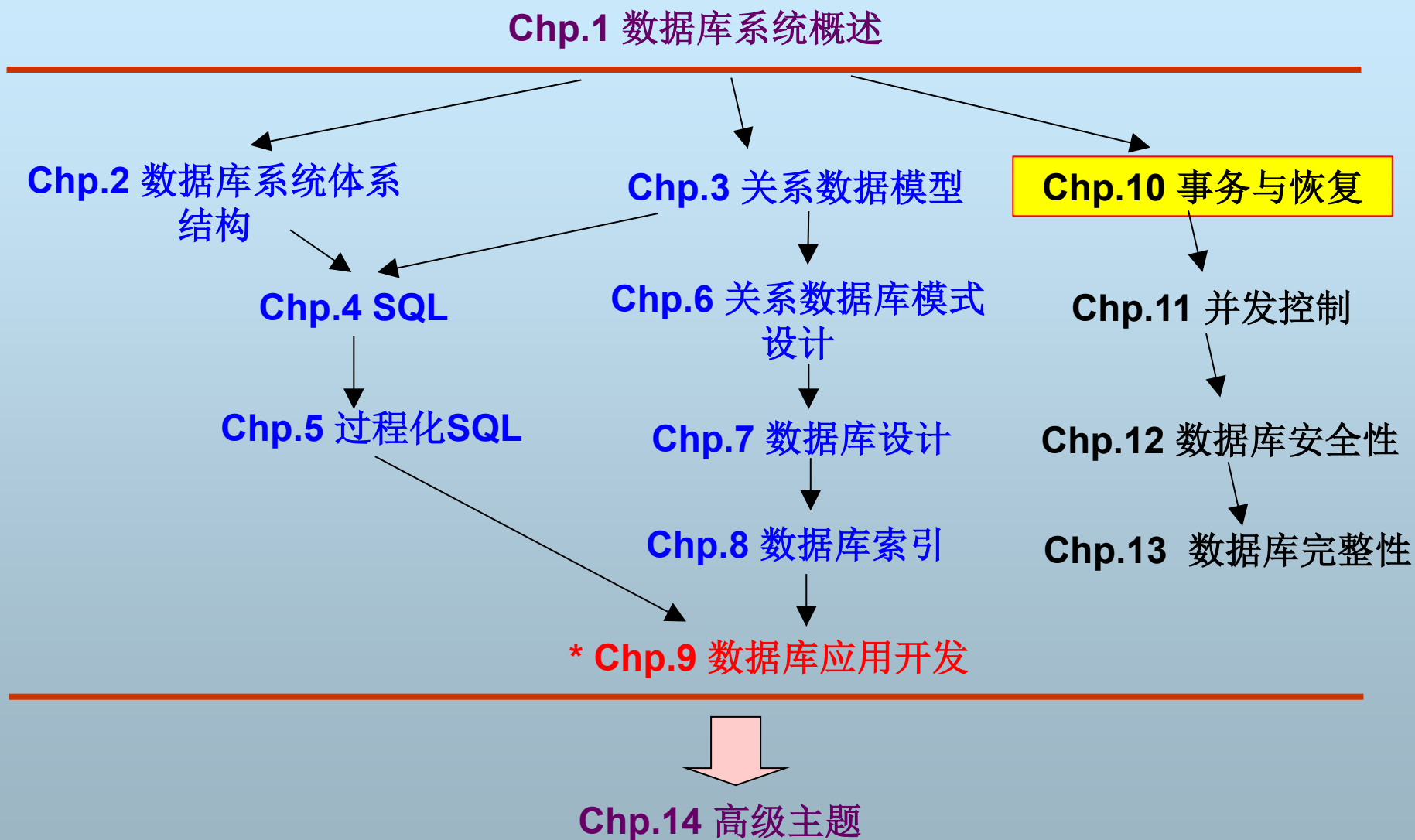


# 第10章 事务与恢复

# 课程知识结构

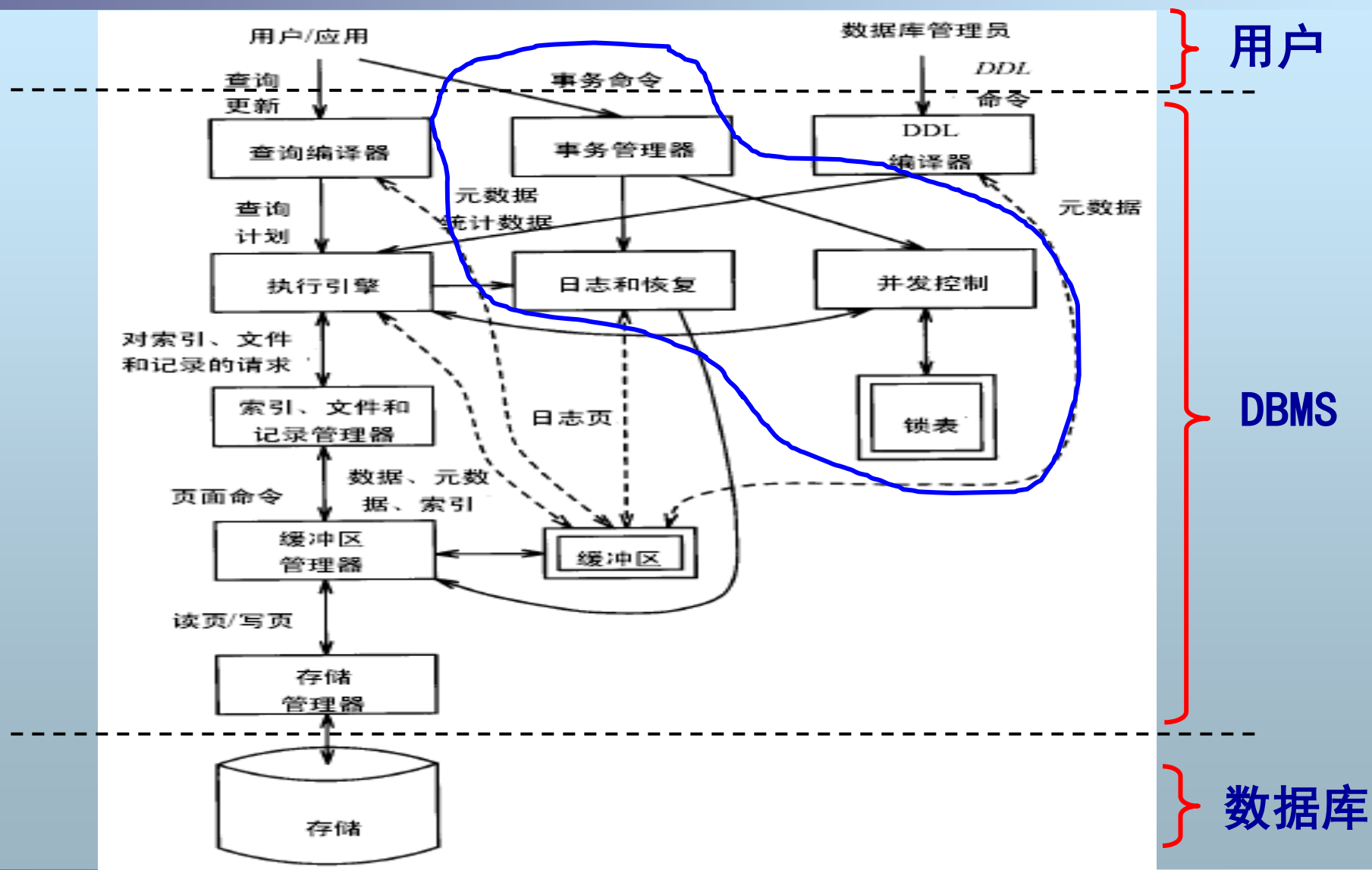


# Part 2: 故障恢复

# 一、数据库保护技术概述

- **数据库保护**：**预防**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**
  - **乐观机制**：事后恢复
  - **悲观机制**：事前预防
- 数据库保护通过四个方面来实现
  - **完整性控制技术**
    - ◆ Enable constraints
  - **安全性控制技术**
    - ◆ Authorization and authentication
  - **数据库的恢复技术**
    - ◆ Deal with failure
  - **并发控制技术**
    - ◆ Deal with data sharing

# DBMS架构



## 二、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

## 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**

# 4、数据库系统故障恢复策略

## ■ 目的

- 恢复DB到最近的一致状态

## ■ 基本原则

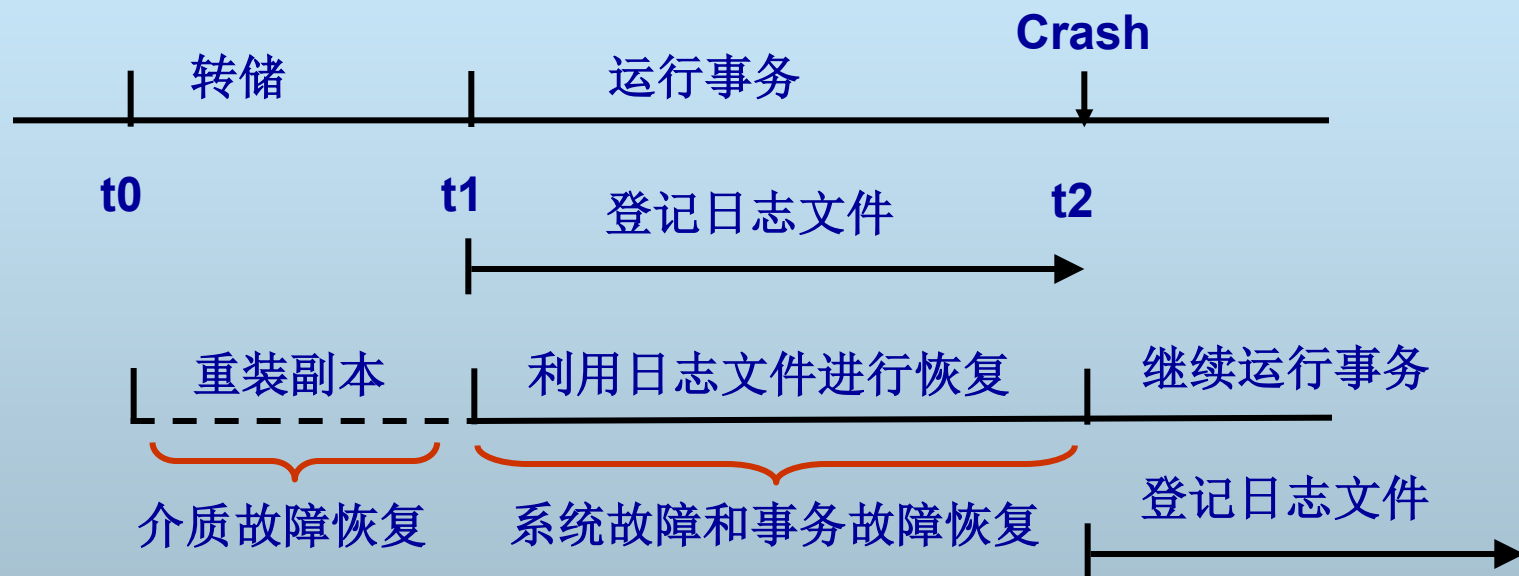
- 冗余 (Redundancy)

## ■ 实现方法

- 定期备份整个数据库
- 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

# 三、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，对应此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) 先写日志

# 先写日志(Write-Ahead Log)原则

- 在数据被写到磁盘之前，对应此修改的日志记录必须已被写到磁盘上

先写日志

<T1, Begin Transaction>

<T1,A,1000,900>

后写日志

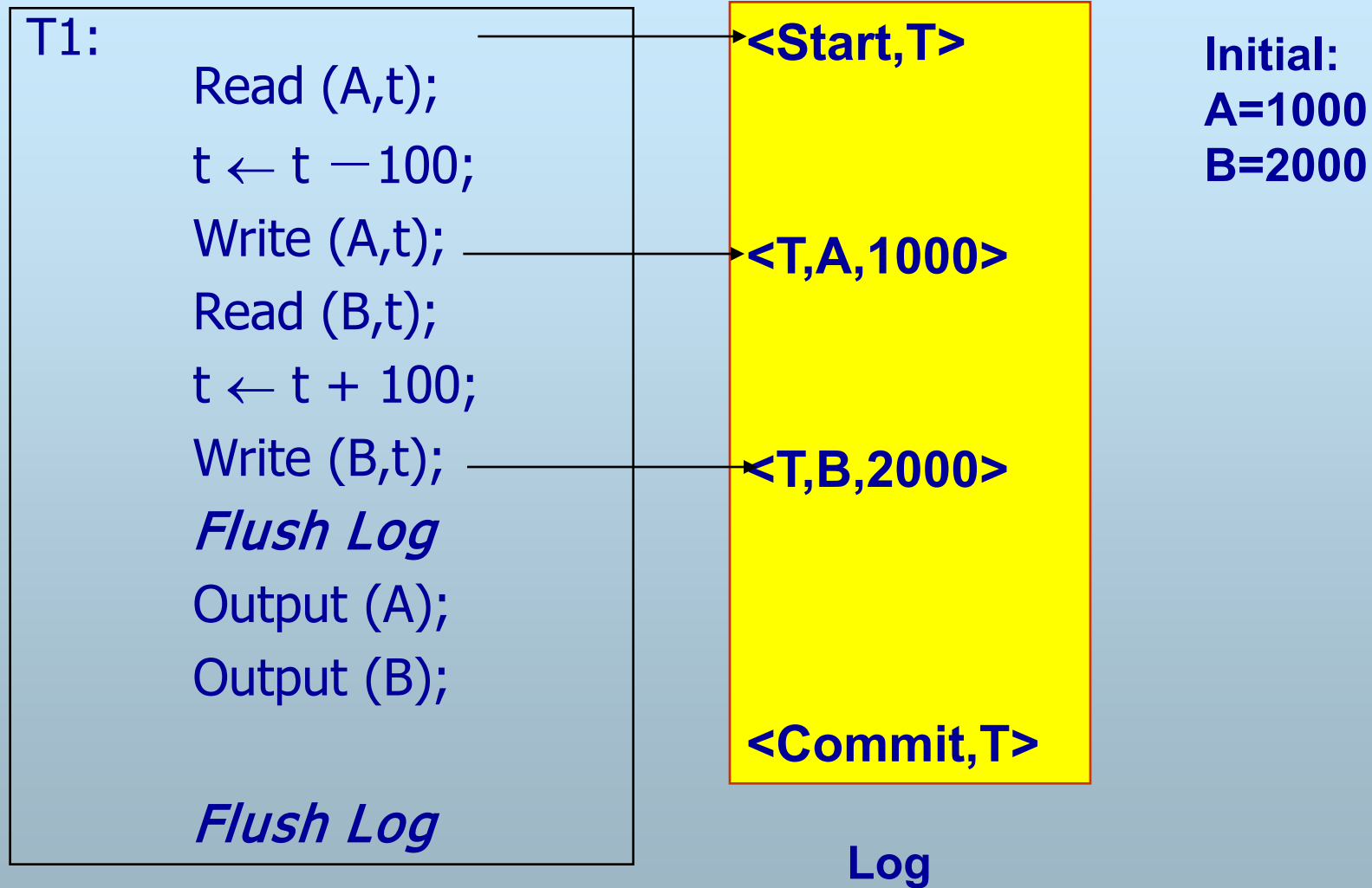
<T1, Begin Transaction>

设T1将A修改为900时发生故障。设此时900已写到数据库，但还未来得及写到日志中。

根据恢复策略，T1在恢复应UNDO，但此时由于后写日志，A的更新操作在日志中没有记录，因此无法将A恢复到1000

如果先写日志，则即使没有写到数据库中，也只不过多执行一次UNDO操作，不会影响数据库的一致性。

# 1、Undo日志规则



# 1、Undo日志规则

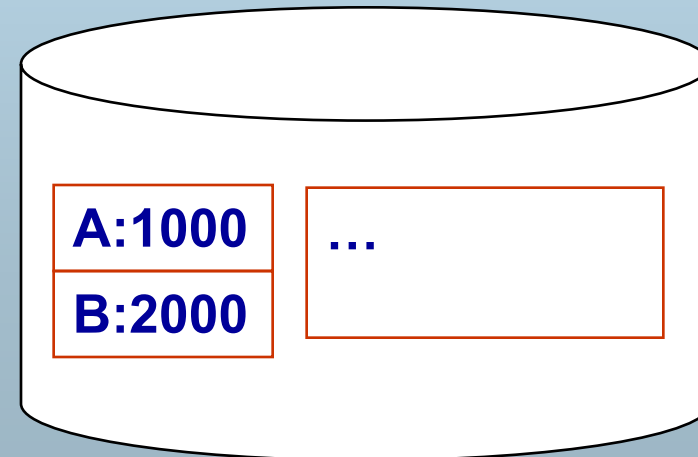
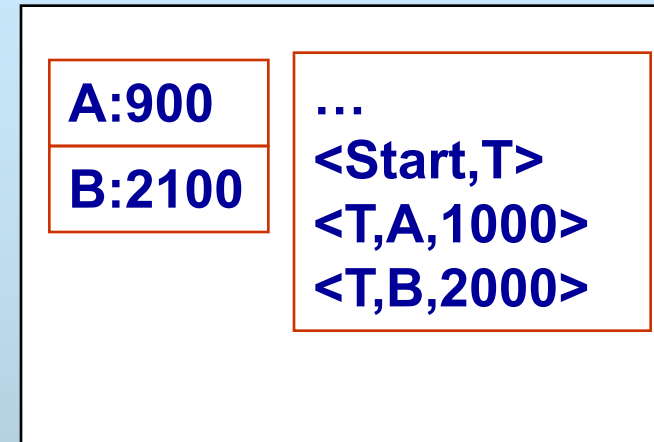
T1:

```
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

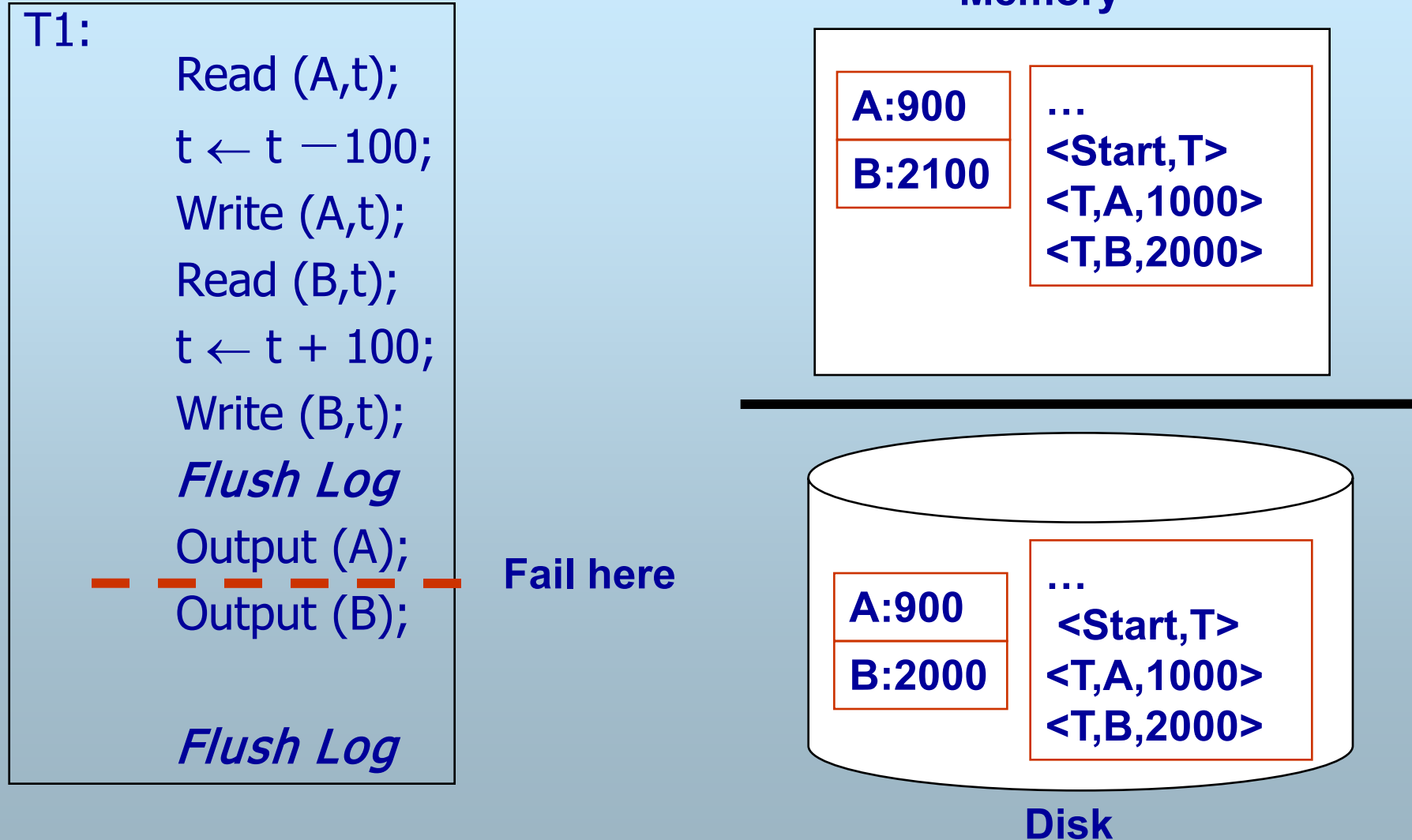
Memory



Disk



# 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

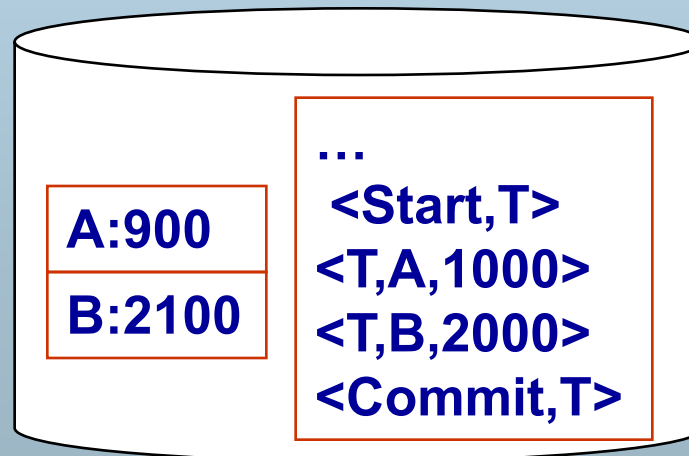
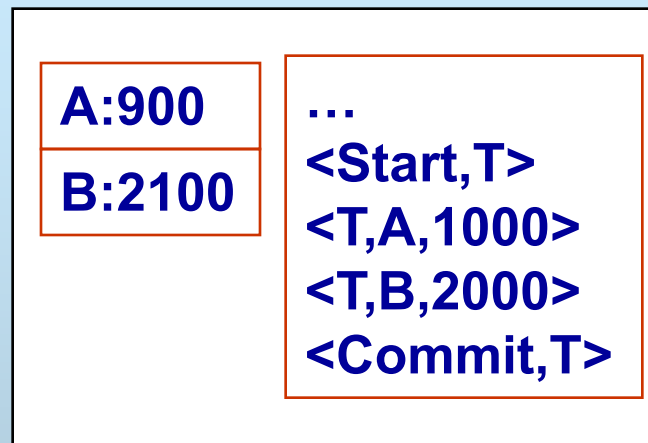
T1:

```
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

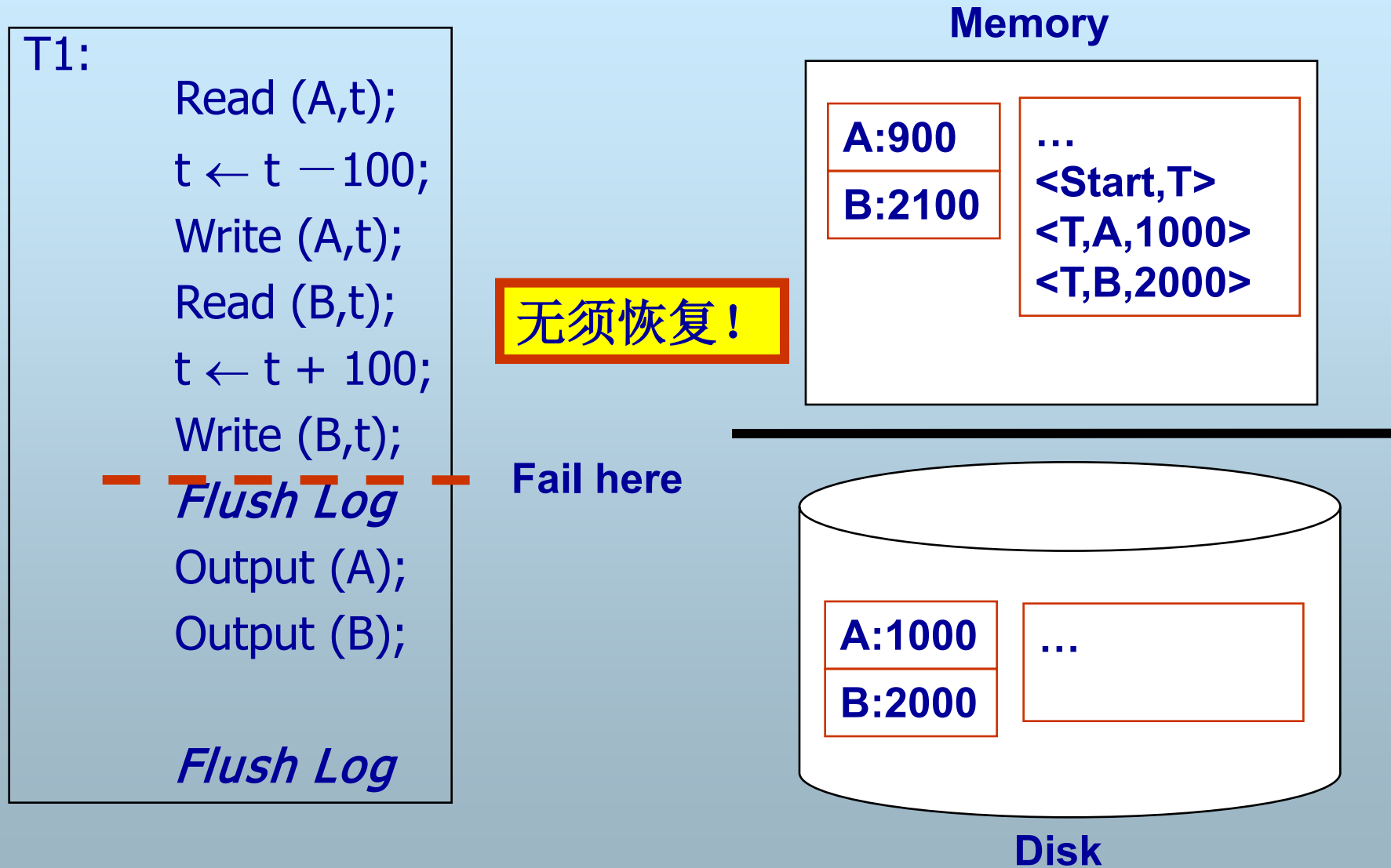
### The recovery process



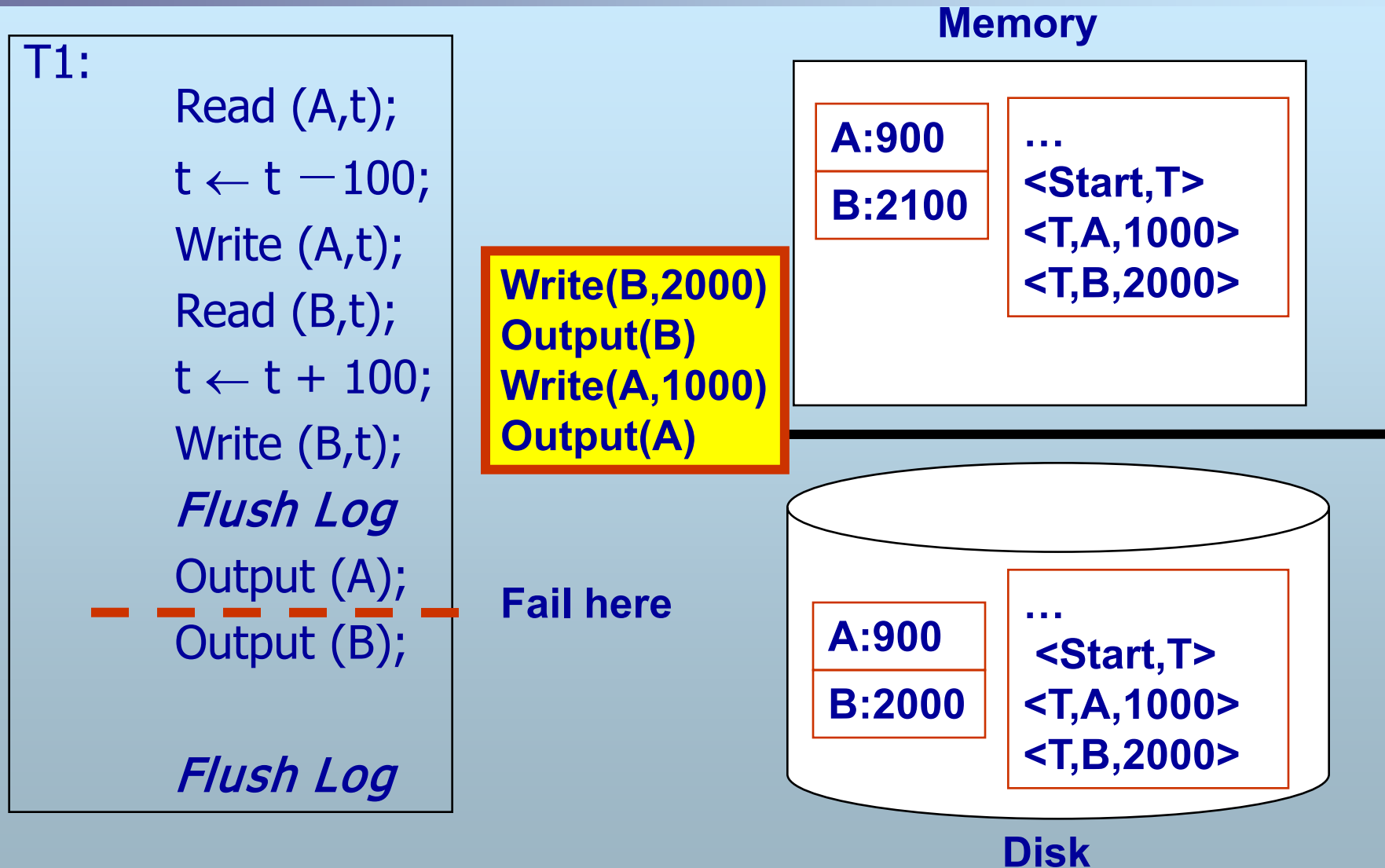
## 2、基于Undo日志的恢复

- 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

### ■ What if failure during recovery?

**No problem !  
Just re-execute the recovery!  
Because each recovery has same effect!**



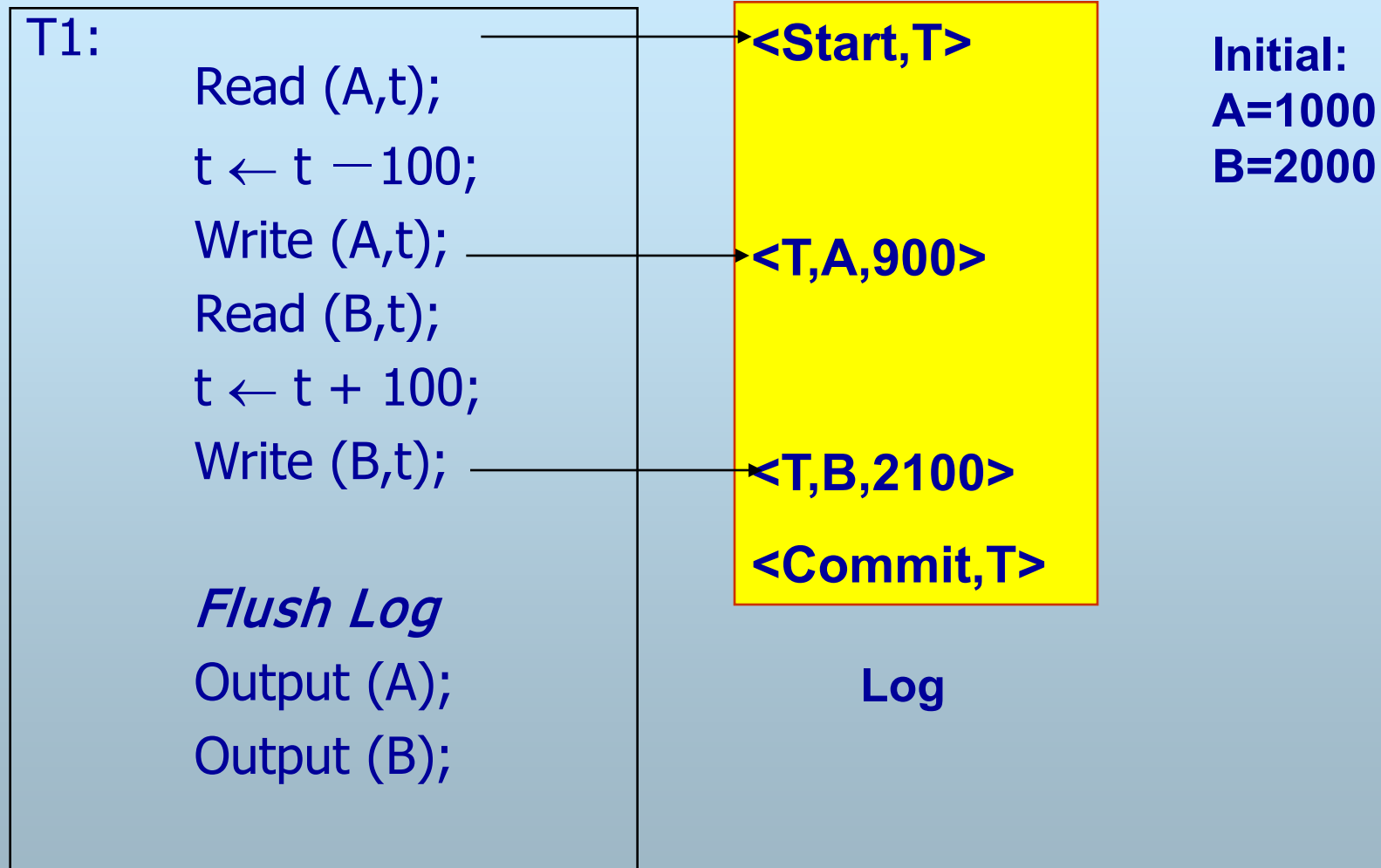
### 3、Undo日志总结

- $\langle T, x, v \rangle$  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

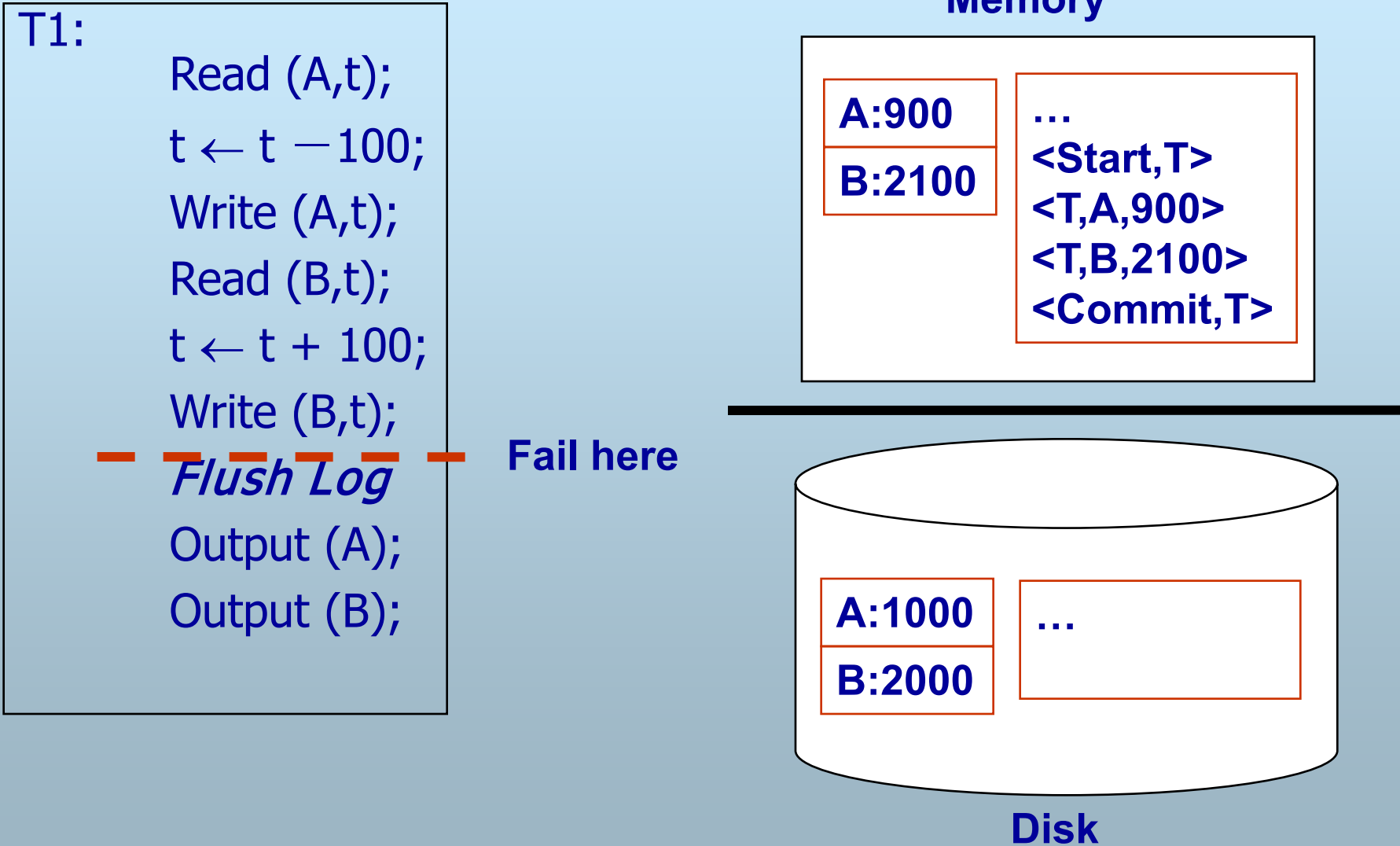
## 四、Redo日志

- 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

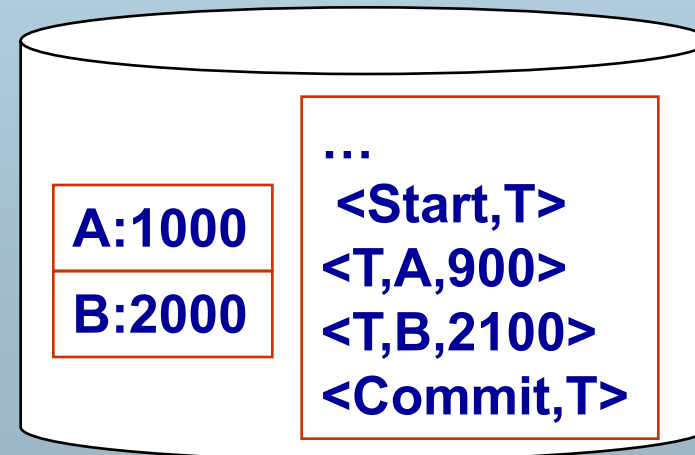
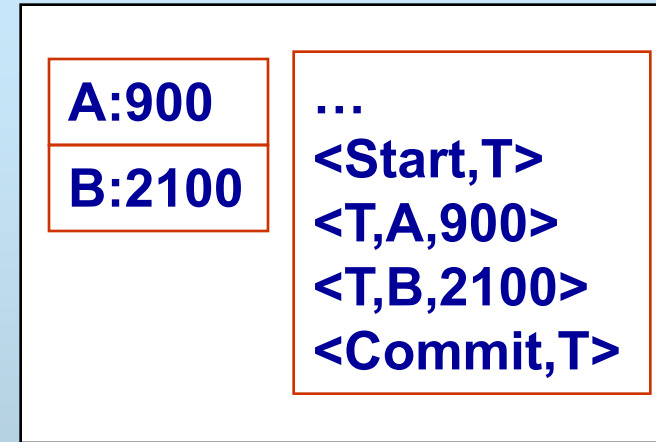


# 1、Redo日志规则

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



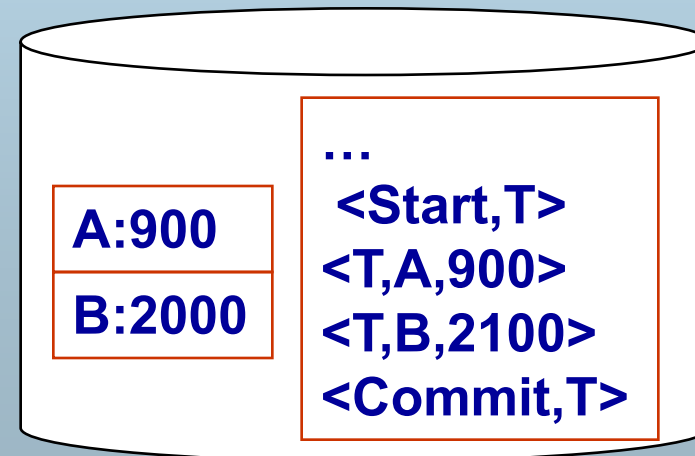
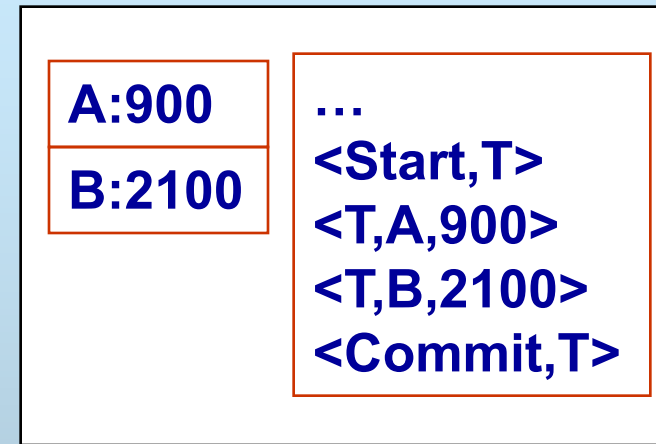
Disk

# 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

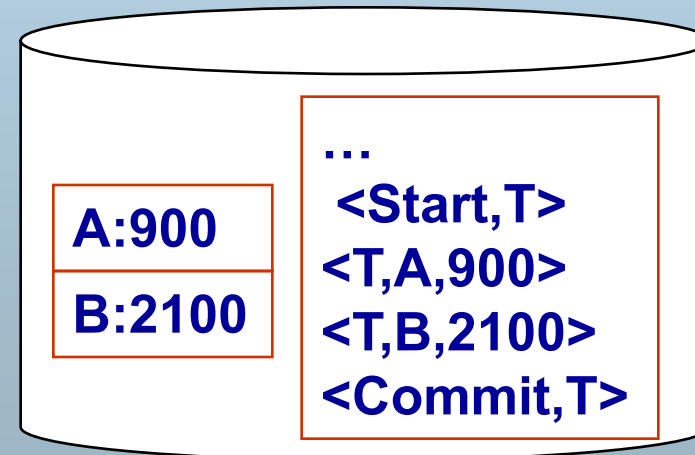
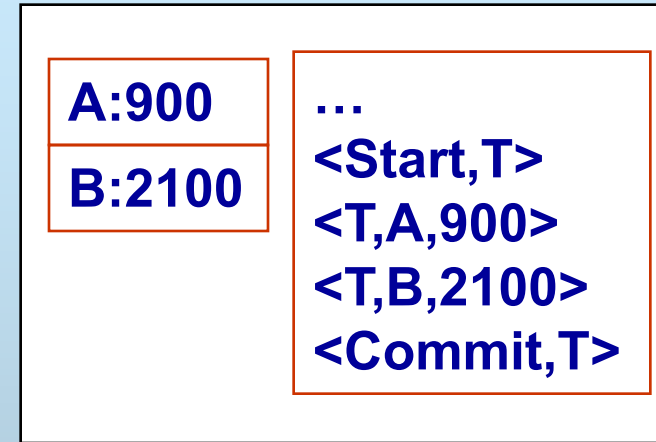
# 1、Redo日志规则

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  $T \notin L$  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

### 3、Undo vs. Redo

- **Undo**基于立即更新 (Immediate Update)
- **Redo**基于延迟更新 (Deferred Update)

# 3、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

## 五、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
- 可以立即更新，也可以延迟更新

# 1、基于Undo/Redo日志的恢复

- 正向扫描日志，将<commit>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<T,x,v,w>，若T在Undo列表中，则
  - Write(x,v); Output(x)
- 正向扫描日志，对于<T,x,v,w>，若T在Redo列表中，则
  - Write(x,w); Output(x)
- 对于Undo列表中的T，写入<abort,T>

# 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; 然后在UNDO, A=1000。不正确

先UNDO, A=1000; 然后REDO, A=1100。正确

## 六、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

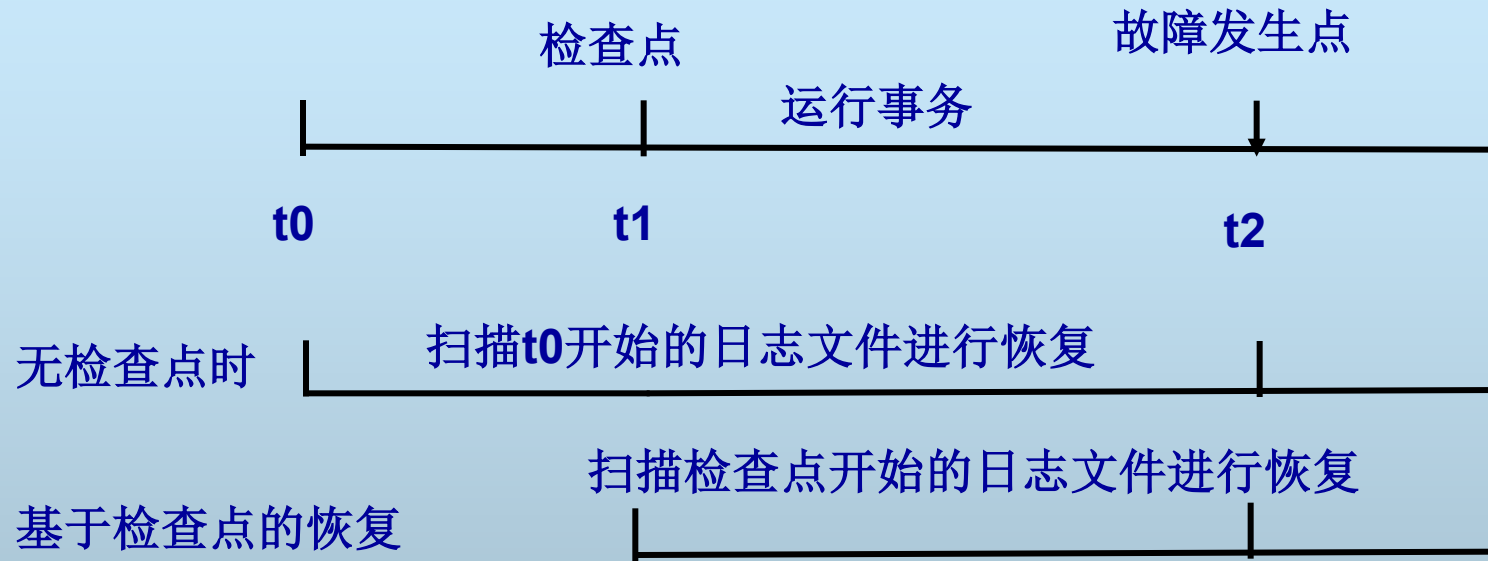


# 1、Simple Checkpoint

## Periodically:

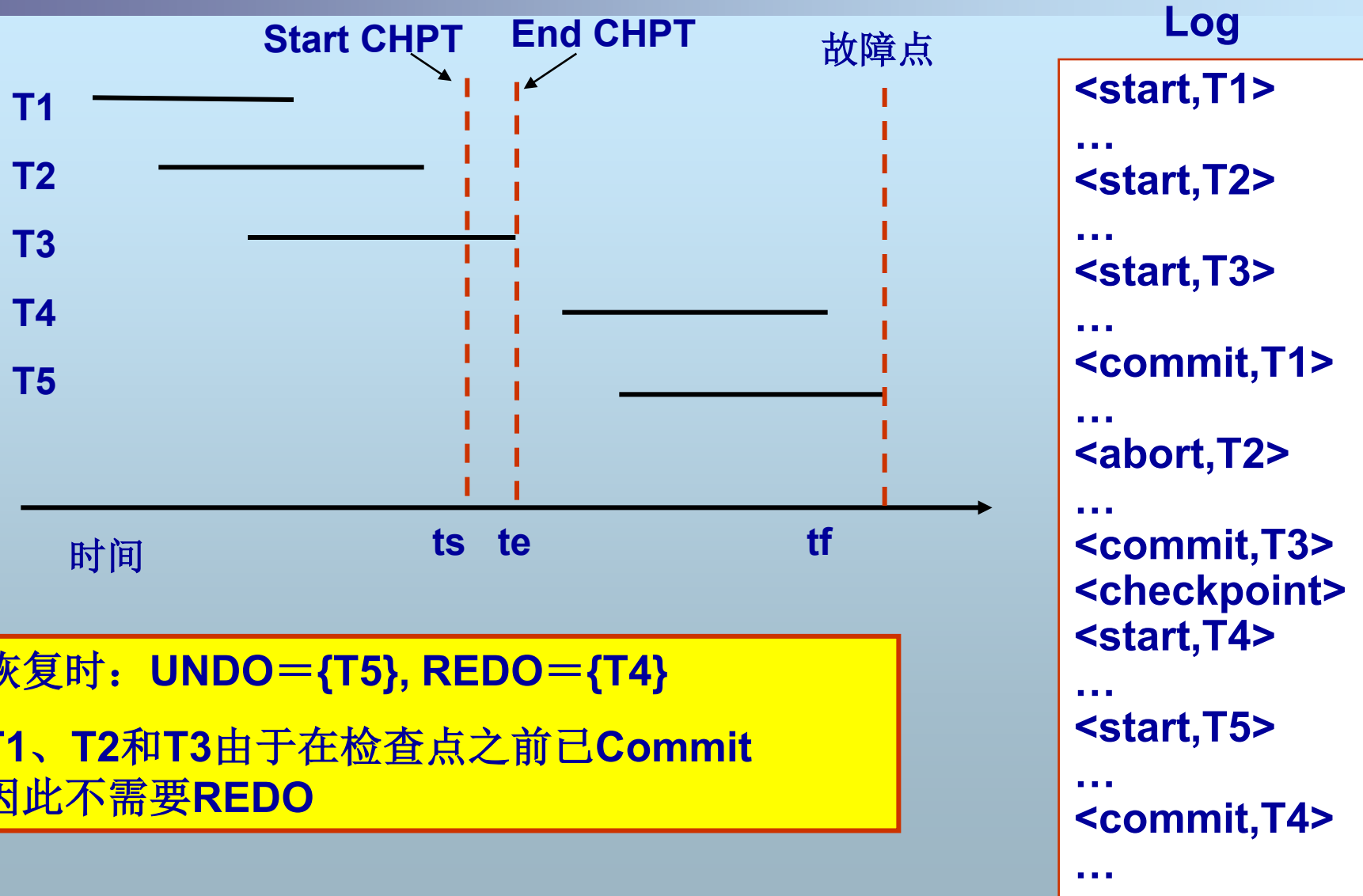
- (1) Do not accept new transactions**
- (2) Wait until all transactions finish (commit/abort)**
- (3) Flush all log records to disk (log)**
- (4) Flush all buffers to disk (DB)**
- (5) Write “checkpoint” record on disk (log)**
- (6) Resume transaction processing**

## 2、Checkpoint-Based Recovery



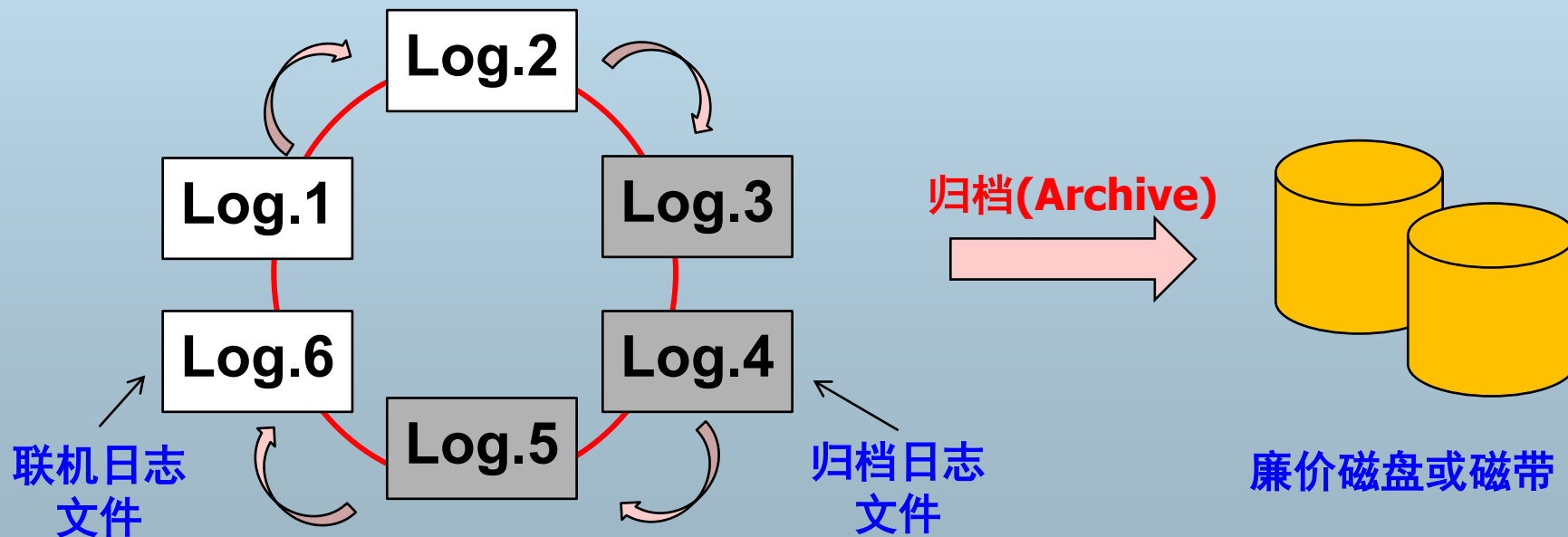
检查点技术保证检查点之前的所有**commit**操作的结果已写回数据库，在恢复时不需**REDO**

## 2、Checkpoint-Based Recovery



# 七、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

- 事务
- 数据库一致性
- 数据库系统故障分析
- Undo日志
- Redo日志
- Undo/Redo日志
- Checkpoint
- 日志轮转技术