



2024年春季学期

# 第10章 事务与恢复



# 课程知识结构

## Chp. 1 数据库系统概述

Chp. 2 数据库系统体系结构

Chp. 3 关系数据模型

Chp. 10 事务与恢复

Chp. 4 SQL

Chp. 6 关系数据库模式设计

Chp. 11 并发控制

Chp. 5 过程化SQL

Chp. 7 数据库设计

Chp. 12 数据库安全性

Chp. 8 数据库索引

Chp. 13 数据库完整性

\* Chp. 9 数据库应用开发

Chp. 14 高级主题

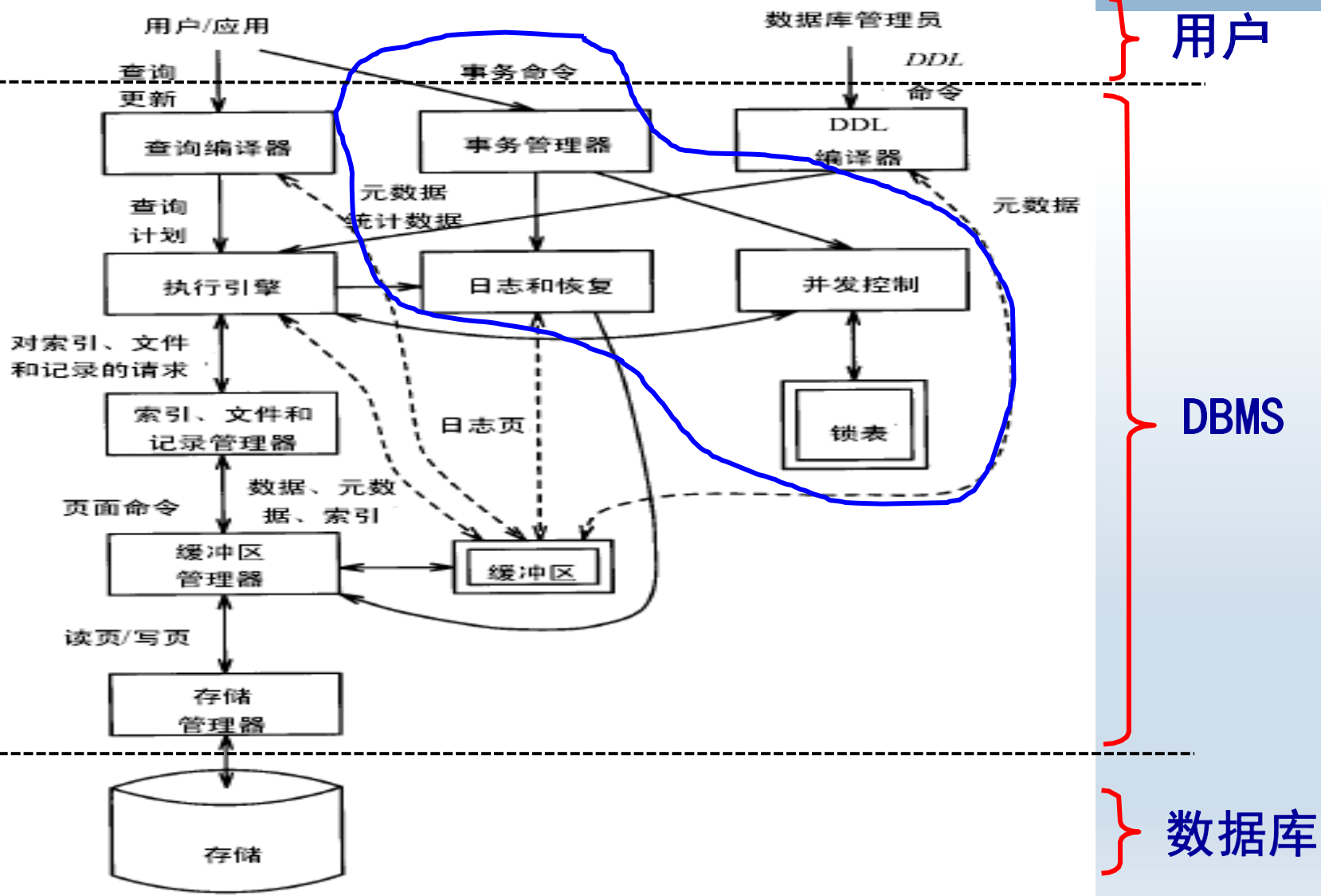


# Database Protection

- **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





# 主要内容

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



# 一、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

- 银行转账：A账户转账到B账户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，要么都不做



# 1、事务

## □ 事务的ACID性质

### □ 原子性 Atomicity

- 事务是不可分的原子，其中的操作要么都做，要么都不做

### □ 一致性 Consistency

- 事务的执行保证数据库从一个一致状态转到另一个一致状态

### □ 隔离性 Isolation

- 多个事务一起执行时相互独立

### □ 持久性 Durability

- 事务一旦成功提交，就在数据库永久保存





## 2、事务的状态 [in logs]

- **<Start T>**
  - **Transaction T has started**
- **<Commit T>**
  - **T has finished successfully and  
all modifications are reflected to disks**
- **<Abort T>**
  - **T has been terminated and  
all modifications have been canceled**



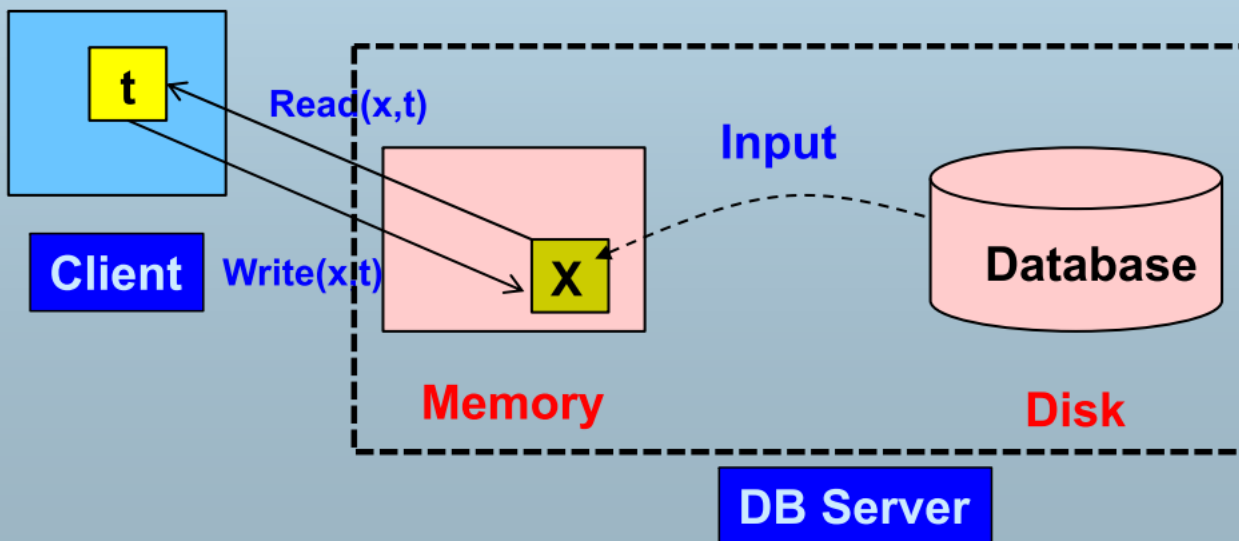
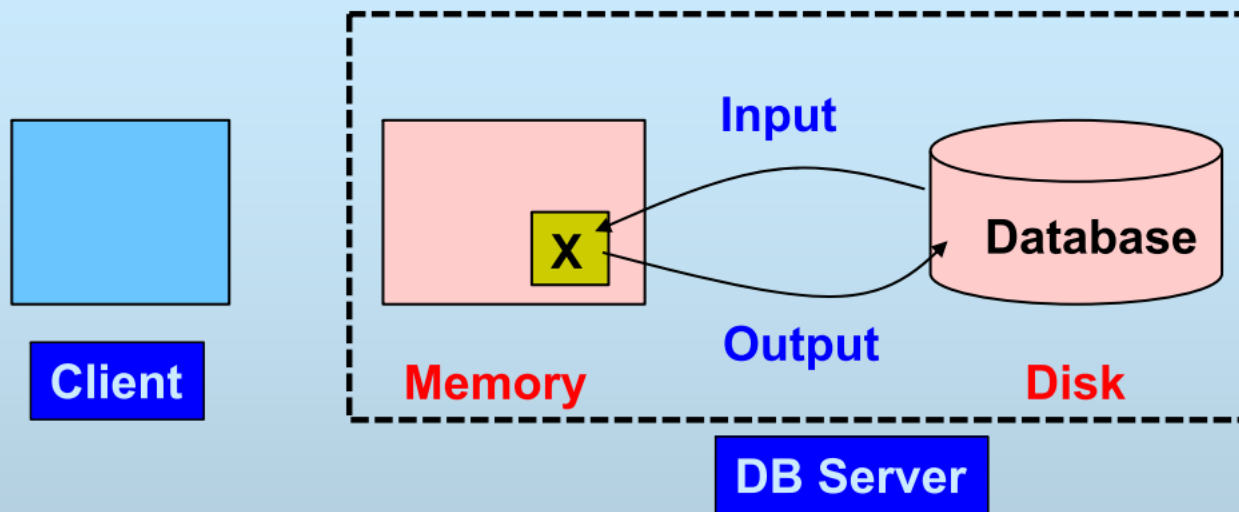
### 3、事务的原语操作

- **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);
```

这两个操作是  
Client端的本地  
操作

DBMS的事务管  
理器实际上是不  
知道这些操作的

此处只为了便于  
大家理解事务的  
操作过程



## 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**
- **MySQL**
  - **Start transaction/commit/rollback**
- **Oracle**
  - **Commit或Commit Work**
  - **Rollback或Rollback Work**



## 6、存储过程中使用事务

```
1
2 delimiter //
3 CREATE PROCEDURE transfer(IN id_from INT, IN id_to INT, IN amount INT, OUT state INT)
4 BEGIN
5     DECLARE s INT DEFAULT 0;
6     DECLARE a INT;
7     DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET s = 1;
8     START TRANSACTION;
9     SELECT count(*) FROM account WHERE id = id_from or id=id_to INTO a;
10 IF a < 2 THEN -- 至少有一个账户不存在
11     SET s = 2;
12 END IF;
13
14 SELECT balance FROM account WHERE id = id_from INTO a;
15 IF a < amount THEN -- 余额不足
16     SET s = 3;
17 END IF;
18 UPDATE account SET balance = balance - amount WHERE id = id_from;
19 UPDATE account SET balance = balance + amount WHERE id = id_to;
20 IF s = 0 THEN
21     SET state = 0;
22     COMMIT;
23 ELSE
24     SET state = -1000;
25     ROLLBACK;
26 END IF;
27 END //
28 delimiter;
```



## 7、ADO中使用事务编程

- **ADODB.Connection**对象主要的方法
  - **Open、Close**
  - **Execute** 可执行SQL语句
  - **BeginTrans、ConnectTrans、RollbackTrans** 用于事务编程
- **示例**

```
Dim cnn as New ADODB.Connection  
cnn.ConnectionString="DSN=Mysql; SERVER=192.168.1.11; UID=root;  
PWD=root; PORT=3306; DATABASE=mydb"  
Cnn.CursorLocation=adUseClient  
Cnn.Open
```



## 7、ADO中使用事务编程

cnn.Open

On Error Goto RollbackAll '错误陷阱

cnn.BeginTrans '此连接下的所有操作现在开始都属于一个事务

Dim rst1, rst2 as New ADODB.Recordset '执行记录的增删改

rst1.Open "account", cnn, adUseClient, adOpenKeyset, adLockOptimistic, adCmdTable

rst1.AddNew '增加新记录

.....

rst2.Open "summary", cnn, adUseClient, adOpenKeyset, adLockOptimistic, adCmdTable

..... '更新关联的summary表

'当发生任何预期错误时, RollbackTrans

If rst2.EOF and rst2.BOF Then

Goto RollbackAll

End If

.....

cnn.CommitTrans '成功到达事务尾部时, 提交事务

cnn.Close

RollbackAll: 'Rollback事务的操作统一进行处理

cnn.RollbackTrans

cnn.Close





## 二、数据库的一致性和正确性

- 一致性 (Consistency)
- 正确性 (Correctness)



# 1、Consistency

## Integrity or consistency constraints

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



# 1、Consistency

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



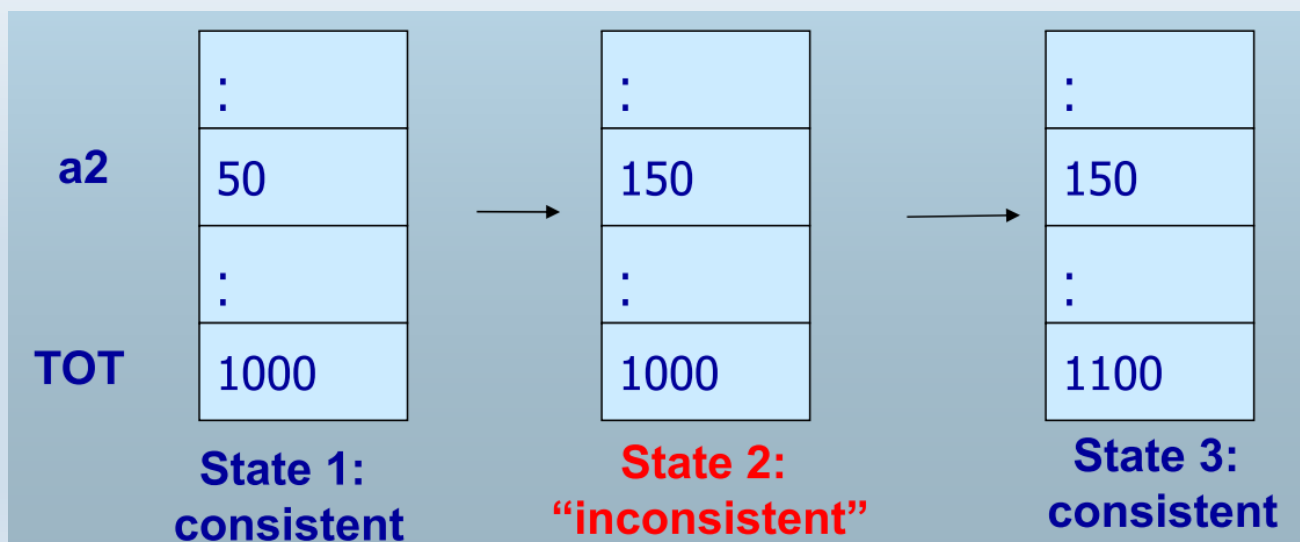
# 1、Consistency

- DB will not always satisfy constraints

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

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

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



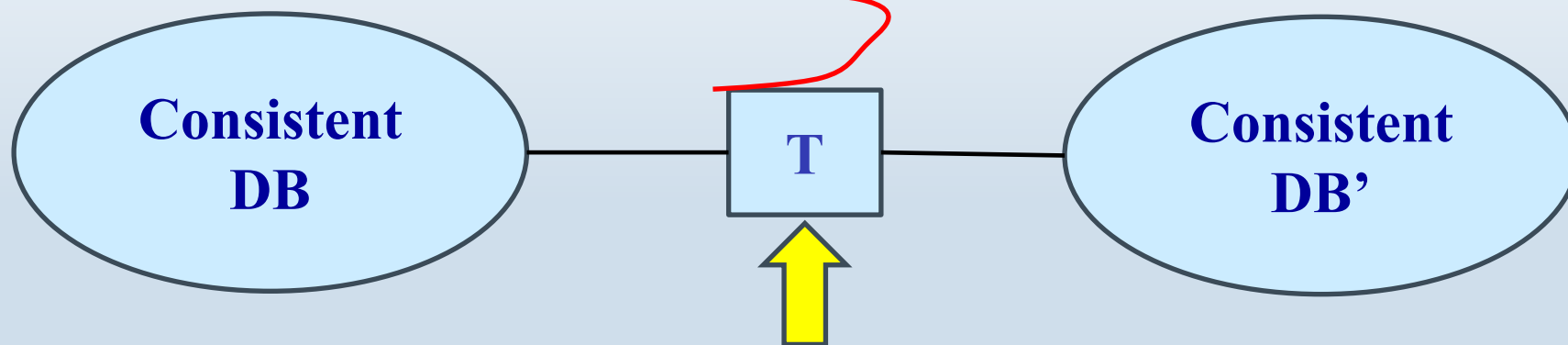


# 1、Consistency

## Consistency of Transaction

### □ 事务的ACID性质

□ Atomicity, **Consistency**, Isolation, Durability

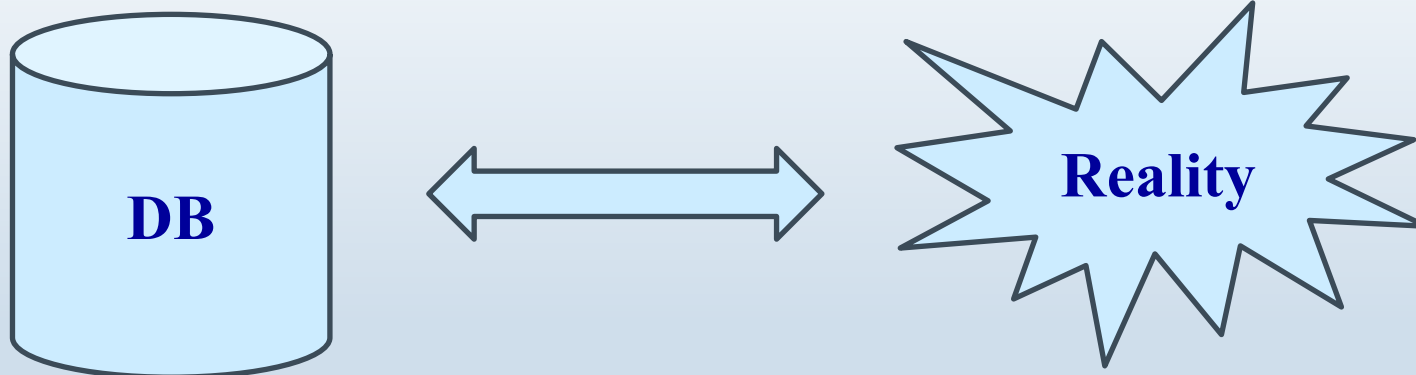


但事务内部允许不保证DB的一致性



## 2、Correctness

**DB should reflect real world**





## 2、Correctness

**DB should reflect real world**

**Example: A telephone number**

**63600110 – Correct**

**abdcefgH – Not correct**

**Can be preserved by explicit constraints!**

**CHECK (tel LIKE '[1-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]')**



## 2、Correctness

**DB should reflect real world**

**Example: A telephone number  
63600110 – Correct  
90000000 – is it correct?**

**Not correct in reality, but  
can DB know this?**

**Answer: NO!**





## 2、Correctness

□ **Correctness of DB  $\neq$  Correctness of reality**

### **Correctness of DB**

如果数据库在事务开始执行时是一致的，并且事务执行结束后数据库仍处于一致状态，则数据库满足正确性。

**Consistency of DB + ACID of transaction**



**Correctness of DB**

通常数据库中使用“一致性”来表示数据库的正确性，很少使用“正确性”



## 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

- 即应用程序可以发生的故障，如转账时余额不足。  
由应用程序处理

### □ 非预期的事务故障

- 如运算溢出，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)

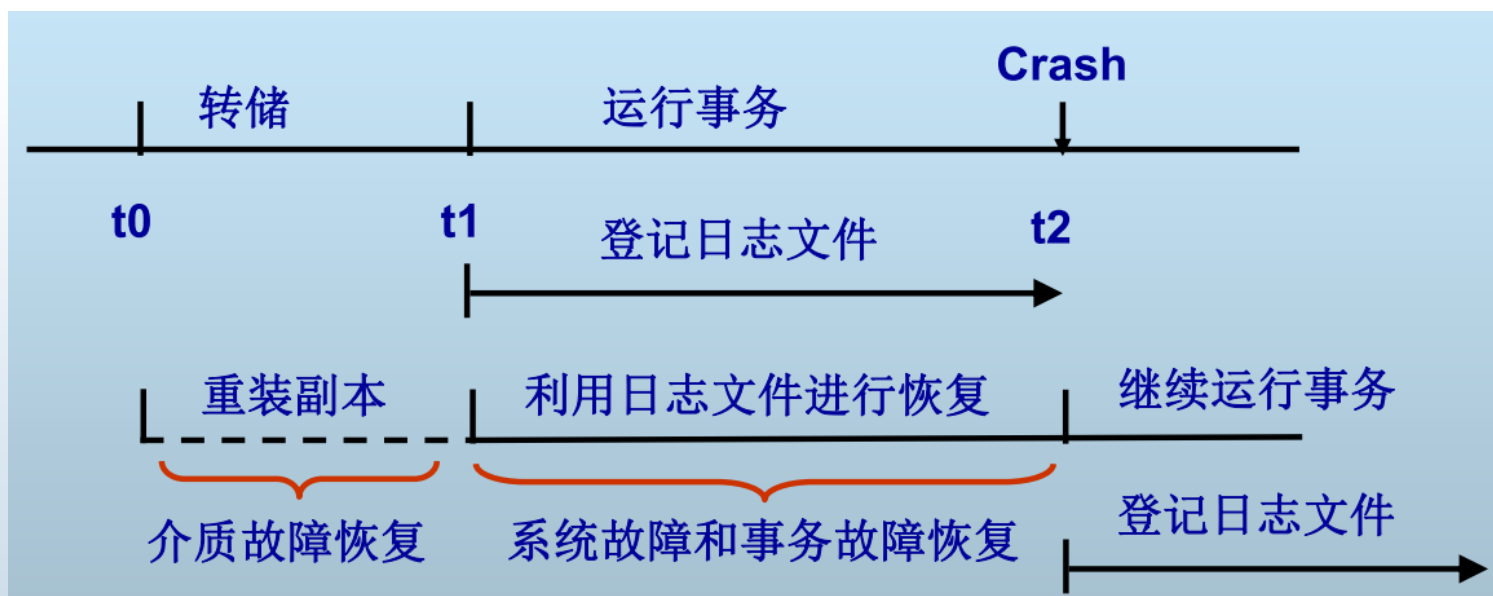
### □ 实现方法

- 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



## 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，此时900已写入到数据库，但还未来得及写到日志中。

根据恢复原则，T1在恢复时应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

T1:

Read (A,t);

$t \leftarrow t - 100;$

Write (A,t);

Read (B,t);

$t \leftarrow t + 100;$

Write (B,t);

***Flush Log***

Output (A);

Output (B);

***Flush Log***

**<Start,T>**

**<T,A,1000>**

**<T,B,2000>**

**<Commit,T>**

**Initial:**

**A=1000**

**B=2000**

Log



# 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>

A:1000

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***

Fail here

Memory

A:900

B:2100

...

<Start,T>

<T,A,1000>

<T,B,2000>

A:900

B:2000

...

<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**

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

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>

<Commit,T>

Disk



# 2、基于Undo日志的恢复

## The recovery process



**Recovery Here**





## 2、基于Undo日志的恢复

- 从头扫描日志，找出所有不存在 $\langle \text{Commit}, T \rangle$ 或 $\langle \text{Abort}, T \rangle$ 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 $\langle T, x, v \rangle$ ，如果 $T \in L$ ，则
  - $\text{write}(X, v)$
  - $\text{output}(X)$
- For each  $T \in L$  do
  - $\text{write} \langle \text{Abort}, T \rangle$  to log



# 2、基于Undo日志的恢复

T1:

Read (A,t);  
t ← t − 100;  
Write (A,t);  
Read (B,t);  
t ← t + 100;  
Write (B,t);  

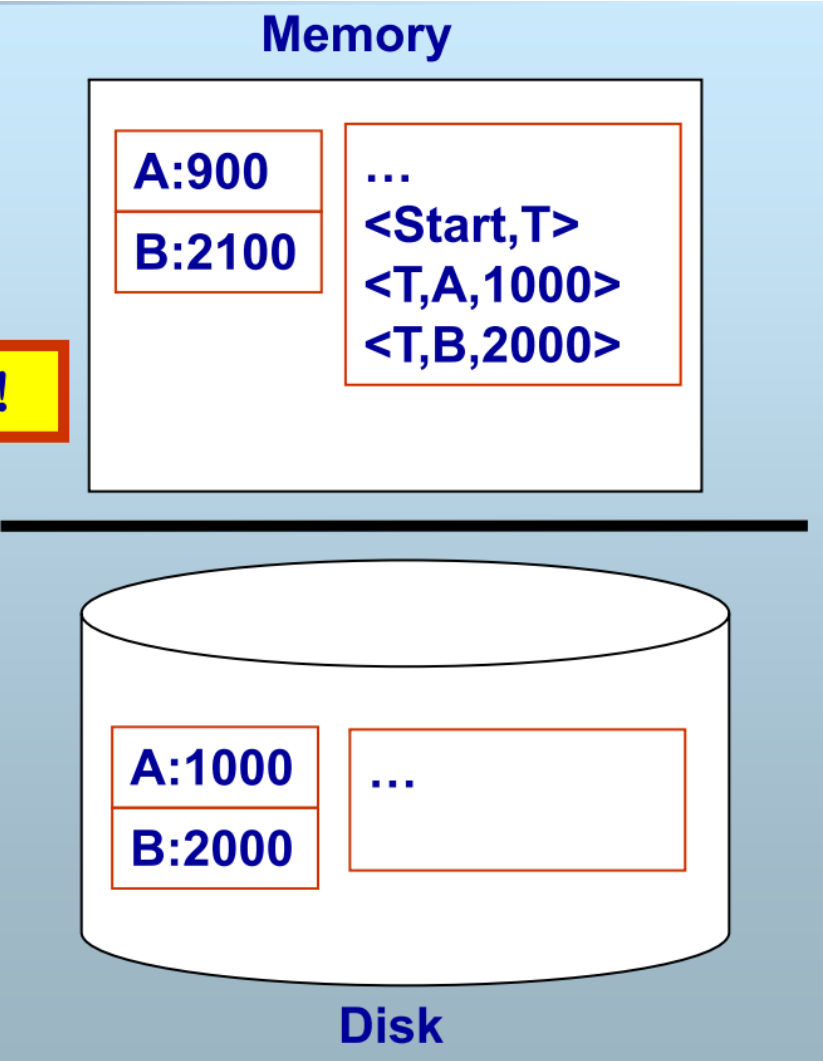
Flush Log

  
Output (A);  
Output (B);  
  

Flush Log

无须恢复!

Fail here





## 2、基于Undo日志的恢复

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**

Write(B,2000)  
Output(B)  
Write(A,1000)  
Output(A)

Fail here

Memory

A:900

B:2100

...

<Start,T>

<T,A,1000>

<T,B,2000>

A:900

B:2000

...

<Start,T>

<T,A,1000>

<T,B,2000>

Disk



## 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

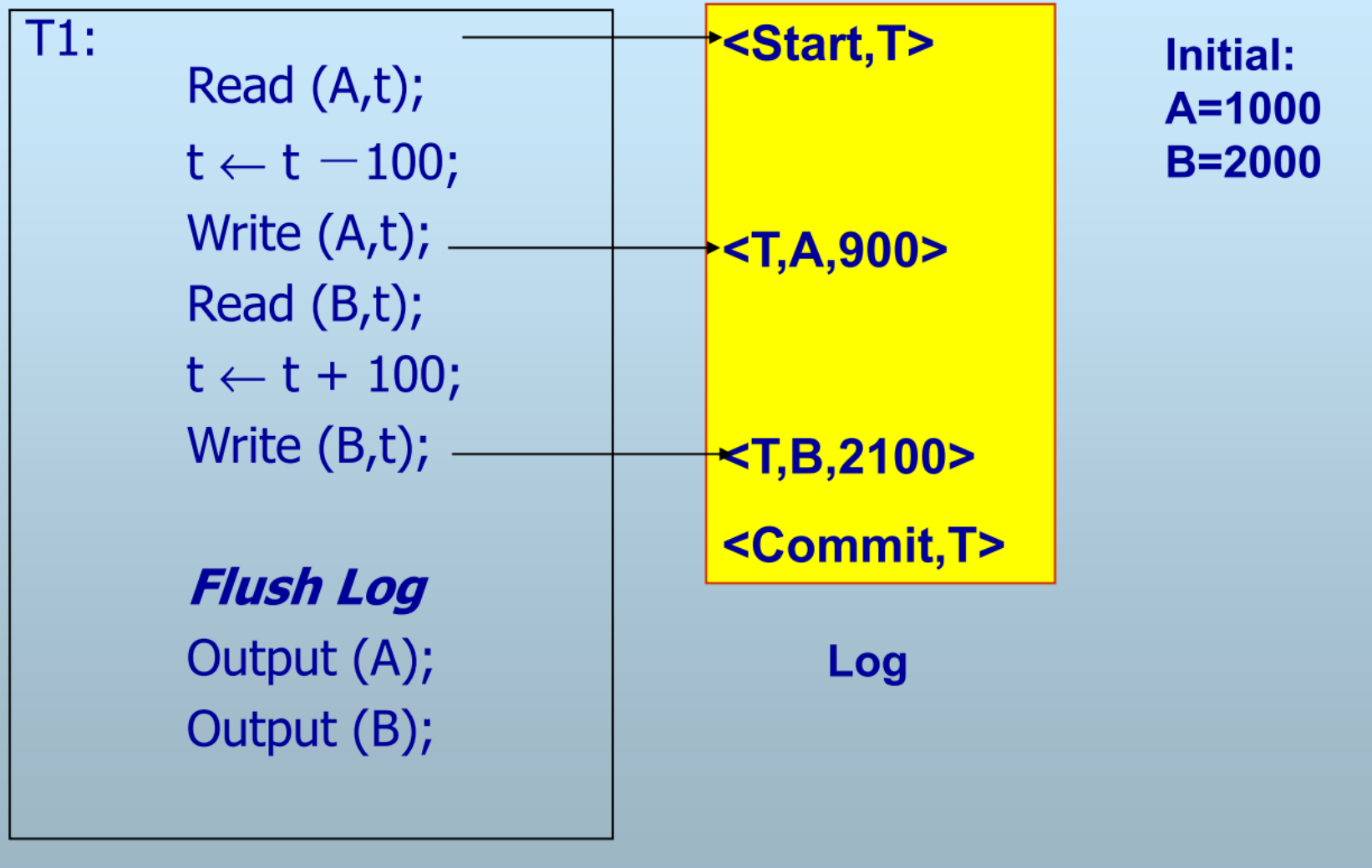


## 五、Redo 日志

- 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

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

Memory

A:900

B:2100

...

<Start,T>

<T,A,900>

<T,B,2100>

<Commit,T>

A:1000

B:2000

...

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

A:900
B:2100

...  
<Start,T>  
<T,A,900>  
<T,B,2100>  
<Commit,T>

A:1000
B:2000

...  
<Start,T>  
<T,A,900>  
<T,B,2100>  
<Commit,T>

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

A:900

B:2100

...

<Start,T>

<T,A,900>

<T,B,2100>

<Commit,T>

A:900

B:2000

...

<Start,T>

<T,A,900>

<T,B,2100>

<Commit,T>

Disk



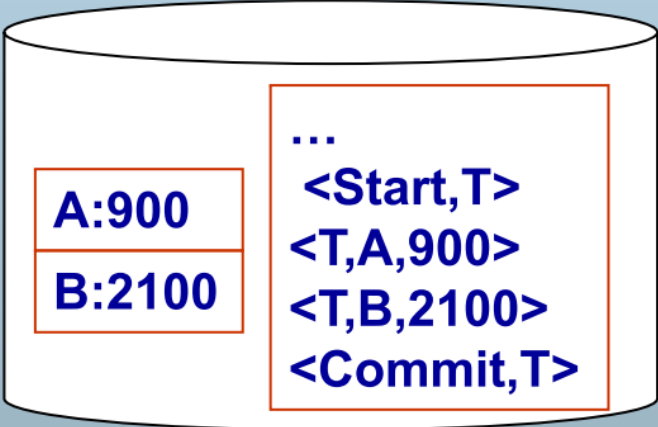
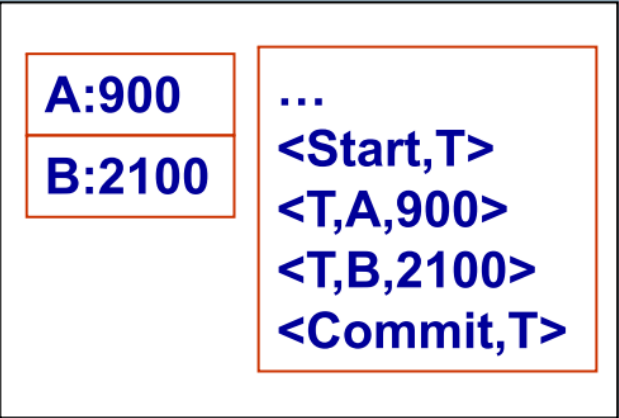
# 1、Redo 日志规则

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，找出所有存在 $\langle \text{Commit}, T \rangle$ 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
  - $\text{write}(X, v)$
  - $\text{output}(X)$
- For each  $T \notin L$  do
  - $\text{write} \langle \text{Abort}, T \rangle$  to log



## 2、基于Redo日志的恢复

### □ 恢复的基础

- 没有<Commit, T>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<Commit, T>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 ← t - 100;  
Write (A,t);  
Output (A);  
Read (B,t);  
t ← 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 ← t - 100;  
Write (A,t);  
Read (B,t);  
t ← 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，与此次修改对应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，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<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>**



## 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



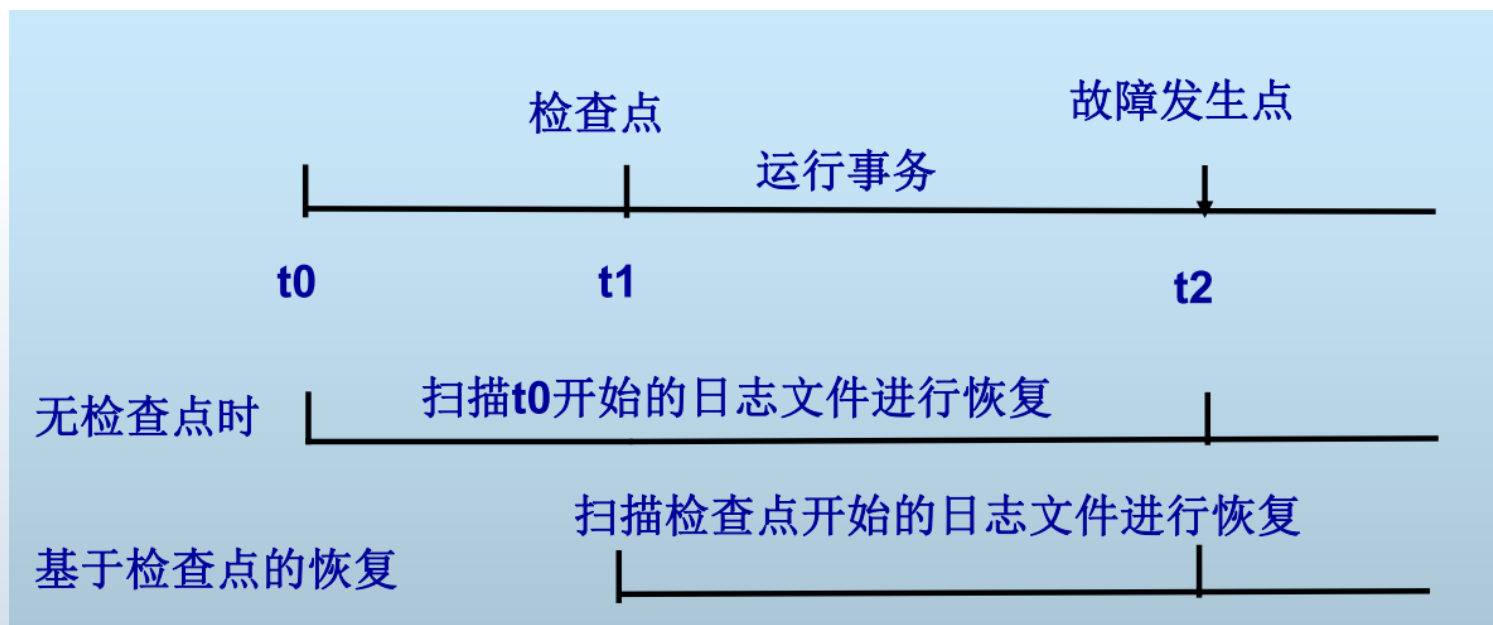
# 1、 Simple Checkpoint

## 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**



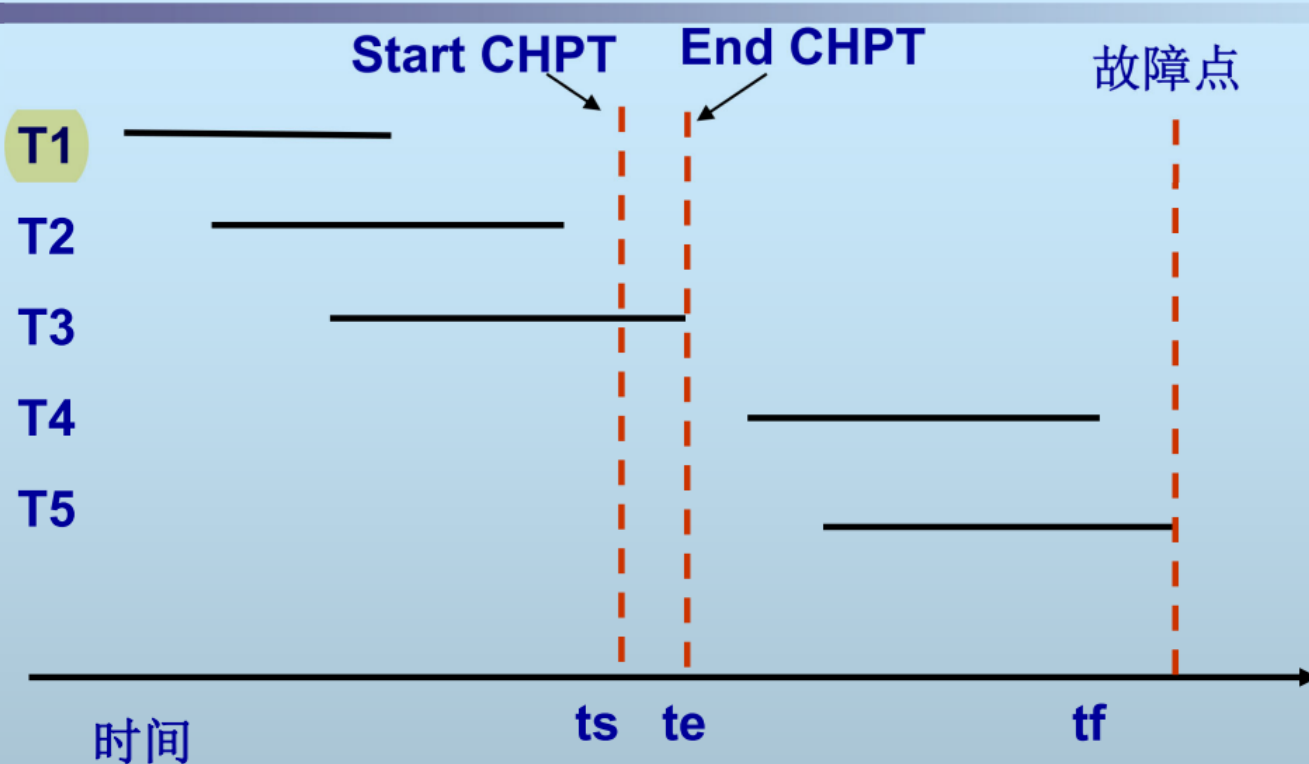
## 2、Checkpoint-Based Recovery



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、T2和T3由于在检查点之前已**Commit**或**Abort**,  
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>
...
```



# 本章小结

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