



本讲简要说明

授课目的与要求： 掌握基于时标和乐观的并发控制，了解一般的恢复机制。

授课重点： 基于时标的并发控制、基于日志的恢复。

作业安排： p.263 3,4,6,15,16

8.2 并发控制

- 调度、串行调度、并行调度
- 可串行调度
 - 视图（或称计算）可串行
 - 冲突可串行（冲突操作对）
- 调度S是冲突可串行当且仅当前驱图G中无环

8.3 基于锁的并发控制机制

- 二值锁
- 共享互斥锁
- 两段锁
 - S满足2PL协议 \Rightarrow S是冲突可串行调度
 - 如果 T_1, T_2, T_3, \dots 是放锁的顺序，则调度冲突等价于
$$S_s = T_1 T_2 T_3 \dots$$
- 死锁

8.4 其它并发控制机制

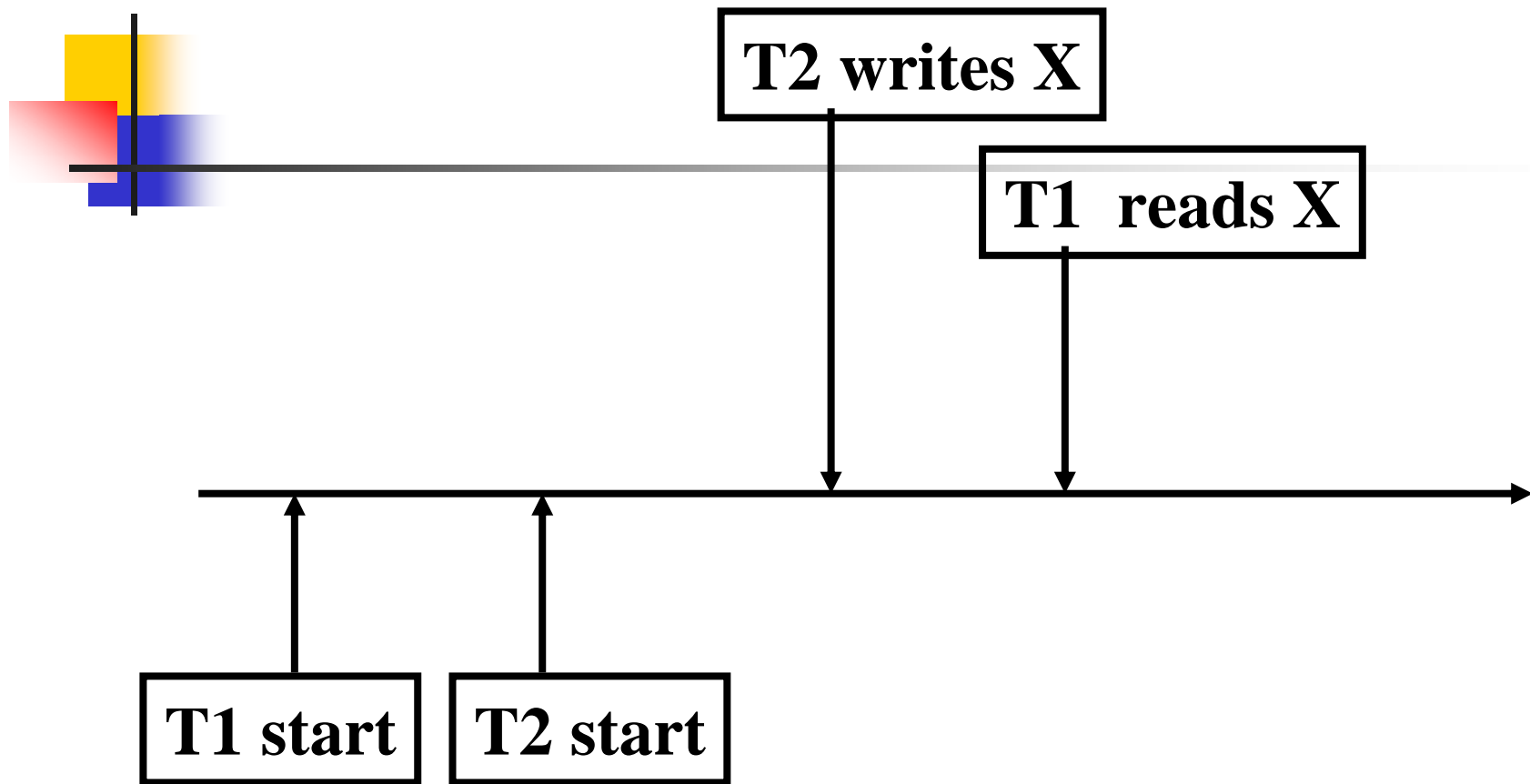


1) 基于时标的并发控制机制

基本思想:基于时标大小解决操作冲突，保证按时间顺序执行冲突操作。

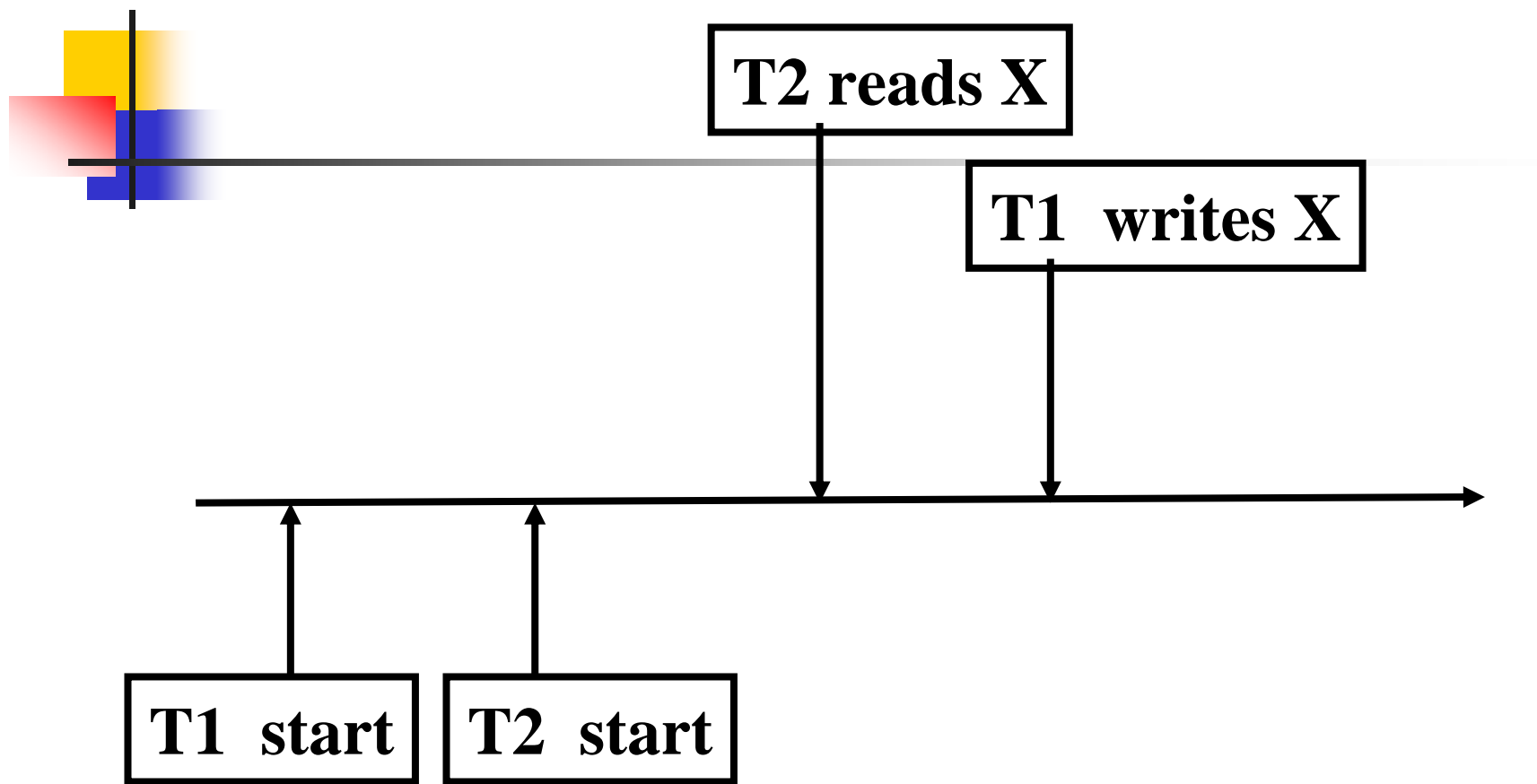
即日常生活中的先来后到。

情景分析



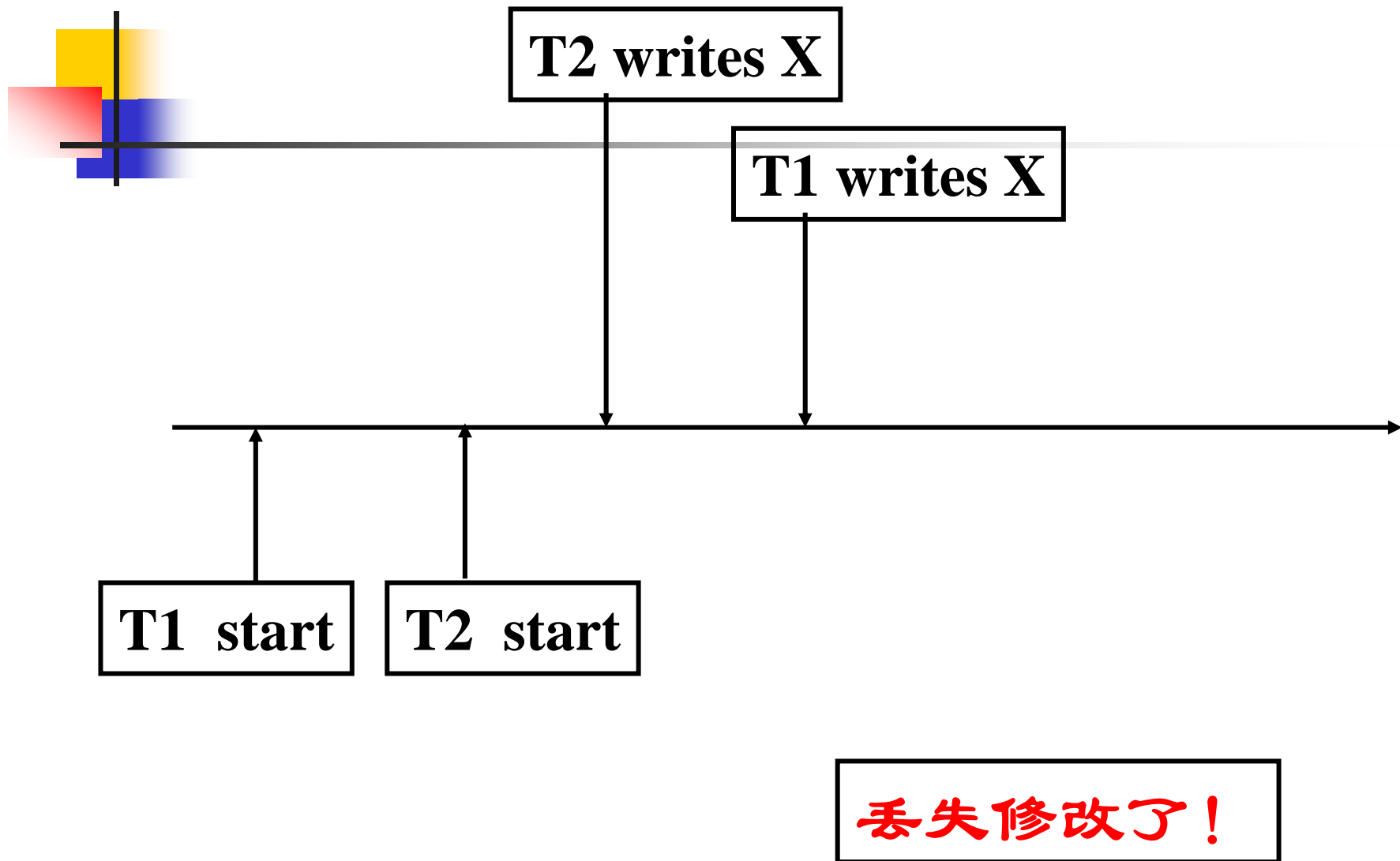
读得太晚了！

情景分析



写得太晚了！

情景分析



1) 基于时标的并发控制机制

- a. 每个事务**T**开始执行时，系统给它打上时标**t**（一般为当前时间）。事务越老，时标越小。
- b. 系统中维持每个数据项**x**的最近读时标**RTM(x)**
和最近写时标**WTM(x)**。
- c. 当执行到某事务**T**的某操作**O(x)**时，比较事务**T**的时标**t**与操作对象**x**的最近读写时标，并按如下规则运作：

1) 基于时标的并发控制机制

① 当O是读操作时, 读得太晚!

如果 $t < \text{WTM}(\mathbf{x})$, 不能读, 拒绝O;

并将T的时标赋一新值, 重新启动T。

如果 $t \geq \text{WTM}(\mathbf{x})$, 执行O, 修改x的读时标, 使得:
 $\text{RTM}(\mathbf{x}) = \max\{t, \text{RTM}(\mathbf{x})\}$

② 当O是写操作

写得太晚!

丢失修改!

如果 $t < \text{RTM}(\mathbf{x})$ 或 $t < \text{WTM}(\mathbf{x})$, 不能写, 拒绝O,

并将T的时标赋一新值, 重新启动T。

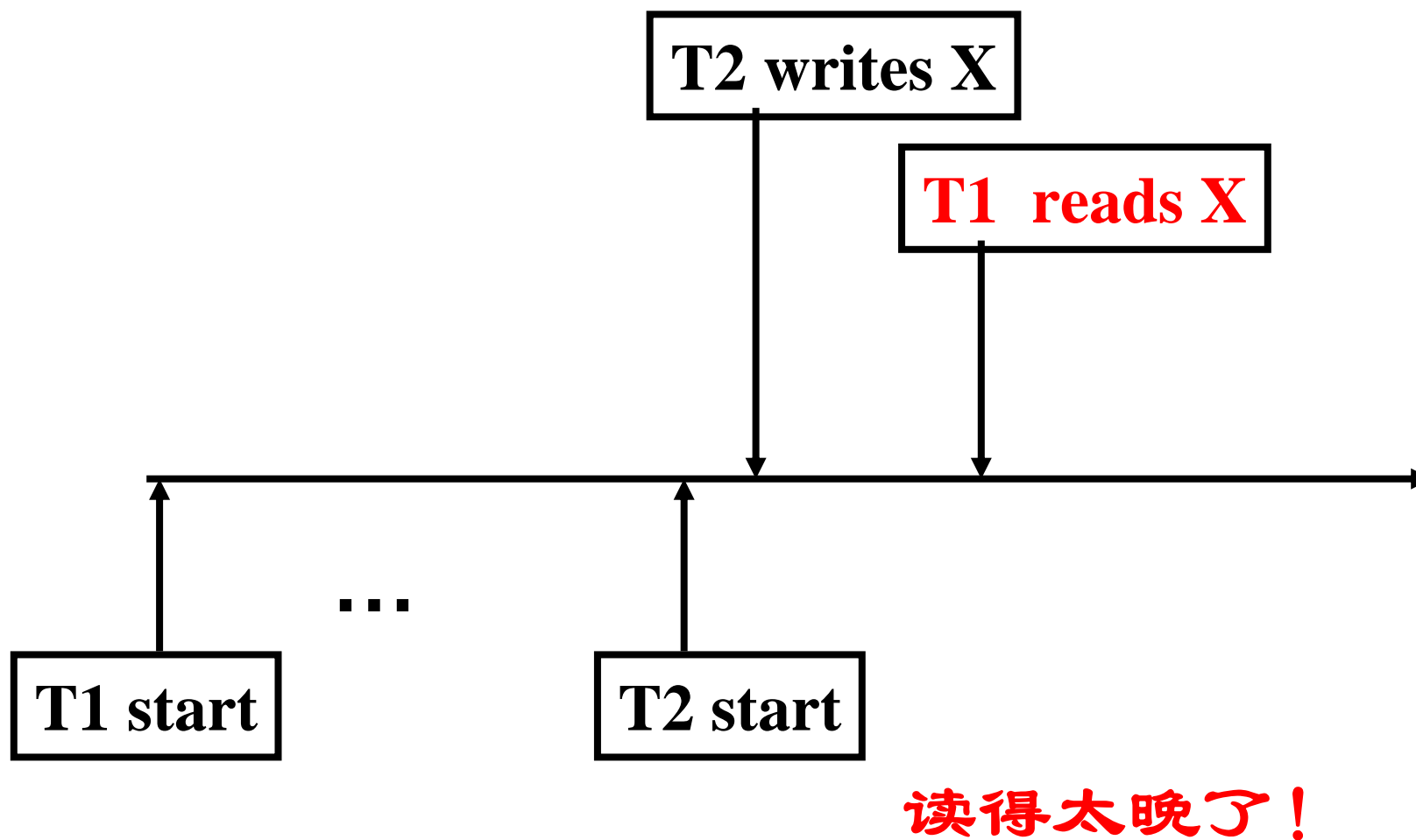
否则, 执行O, 修改x的写时标, 使得 $\text{WTM}(\mathbf{x}) = t$

例： 设有两个事务T1: R(A)W(A)
T2: R(A)W(A),
它们具有时标 **t1** < t2。

RTM (A)	WTM (A)	调度操作序列
0	0	
t1	0	T1: R(A)
t2	0	T2: R(A)
t2	0	T1: W(A), T1回退
t2	t2	T2: W(A)
t3	t2	T1: R(A)
t3	t3	T1: W(A)

若事务时标的顺序为 T1, T2, T3, ...
则调度冲突等价于 $S_s = T1\ T2\ T3...$

2) 多版本 (Multiversion) 并发控制技术



3) 多版本 (Multiversion) 并发控制技术

系统对数据项 x 维持 k 个版本 x_1, x_2, \dots, x_k , 对每个版本保存:

read_TS(x_i): x_i 的最近读时标

write_TS(x_i): x_i 的最近写时标

使用两条规则控制事务 T (时标为 t) 对数据项 x 的读写:

(1) 读: 设 x 的所有版本中写时标小于等于 t 的最后版本为 x_i , 返回该版本的值,并修改其读时标。 (读总可行)

(2) 写: 设 x 的所有版本中具有最大写时标的版本为 x_i ,
if write_TS(x_i) \leq t < read_TS(x_i)
then 终止并回退 T

4) 乐观的并发控制

8.4 其它并发控制机制

在乐观并发控制中，对事务的执行过程不作任何检查，对数据库的修改也不立即进行，而是到事务结束时进行有效性检查。当事务的执行不破坏可串行性时，交付事务；否则将事务撤消，并回退，重新启动。

适用场景：

- 冲突较少
- 系统资源丰富
- 应用系统有实时要求

当冲突操作很多时，引起大量重启，降低效率。

4) 乐观的并发控制

事务有三个阶段:

(1) Read

- all DB values read
- writes to temporary storage
- no locking

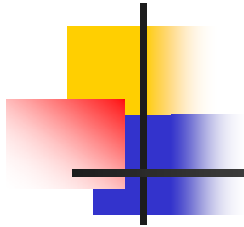
(2) Validate

- check if schedule so far is serializable

(3) Write

- if validate ok, write to DB

4) 乐观的并发控制



- 每个事务 T_i 维持:

$RS(T_i) = \{T_i \text{ 读的所有数据项} \}$

$WS(T_i) = \{T_i \text{ 写的所有数据项} \}$

4) 乐观的并发控制

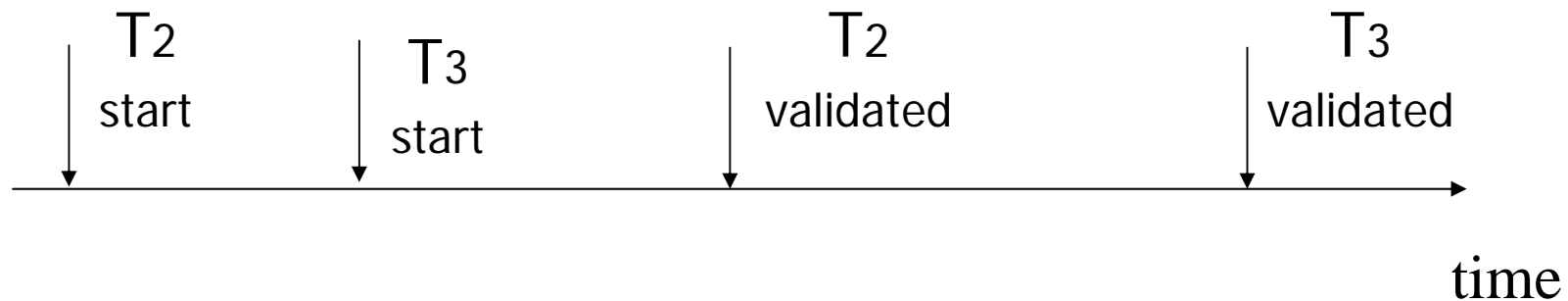
校验必须阻止的情形:

$RS(T_2) = \{B\}$

$WS(T_2) = \{B, D\}$

$RS(T_3) = \{A, B\} \neq \phi$

$WS(T_3) = \{C\}$



4) 乐观的并发控制

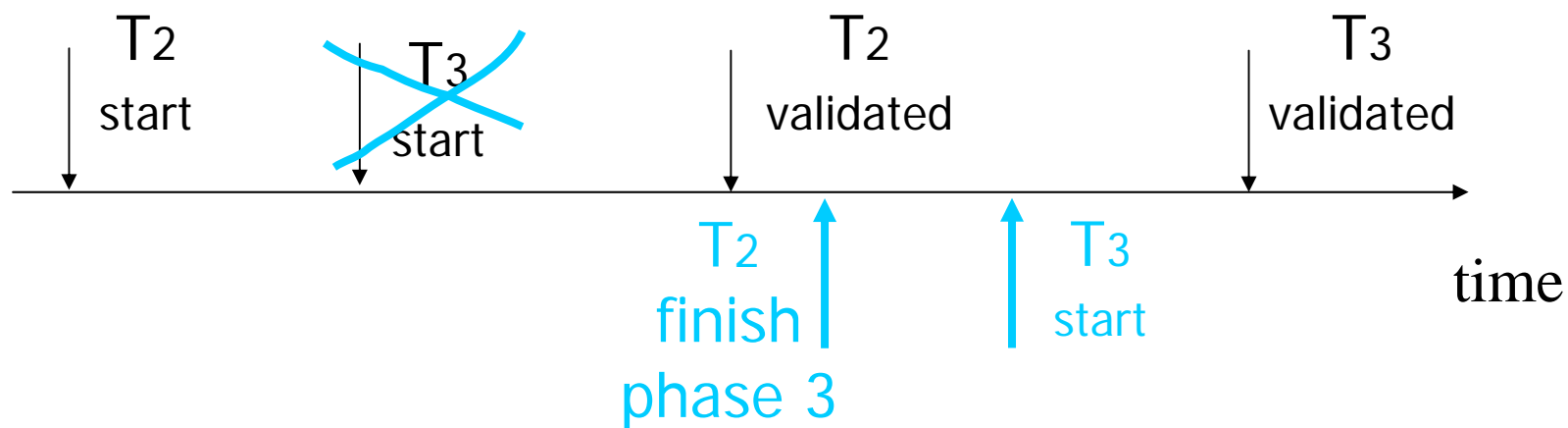
校验通过的情形:

$RS(T_2) = \{B\}$

$WS(T_2) = \{B, D\}$

$RS(T_3) = \{A, B\} \neq \phi$

$WS(T_3) = \{C\}$



4) 乐观的并发控制

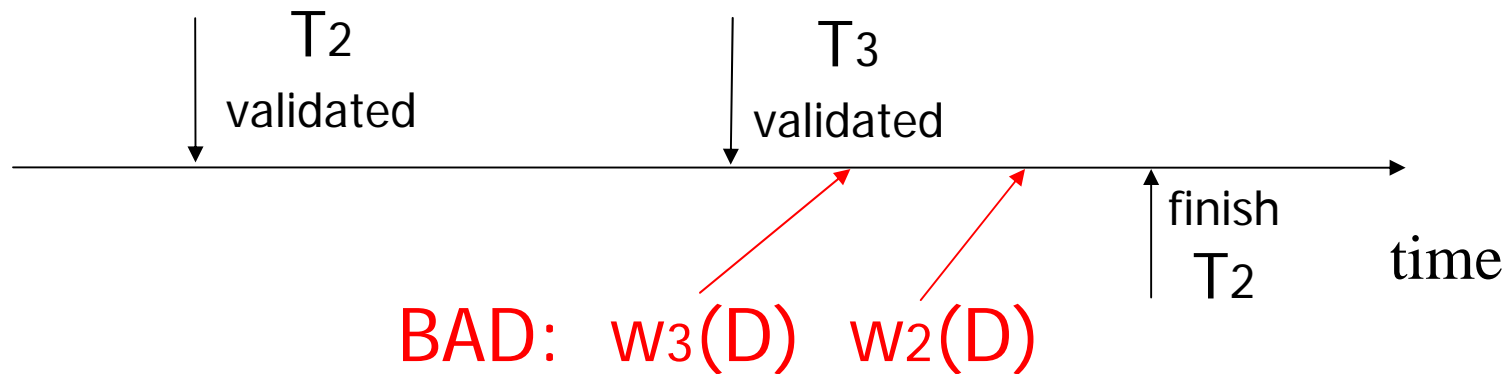
校验必须阻止的另一情形:

$$RS(T_2) = \{A\}$$

$$RS(T_3) = \{A, B\}$$

$$WS(T_2) = \{D, E\}$$

$$WS(T_3) = \{C, D\}$$



4) 乐观的并发控制

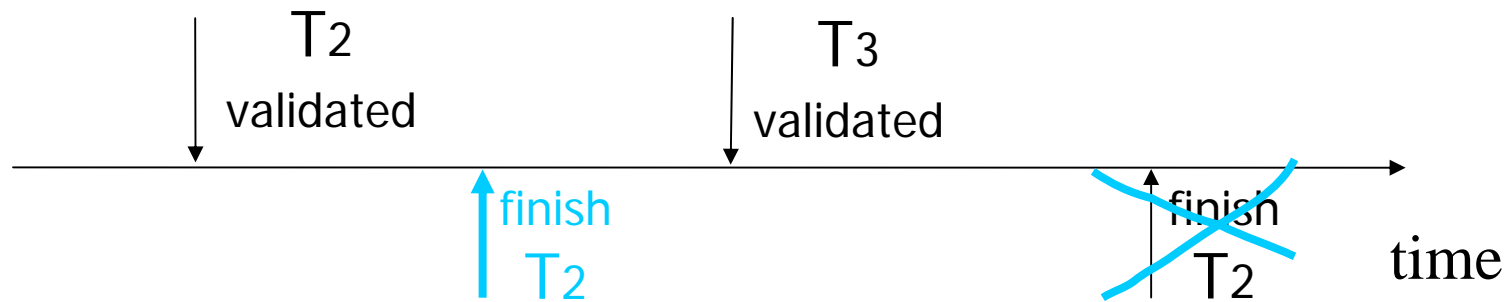
校验通过的情形:

$$RS(T_2) = \{A\}$$

$$WS(T_2) = \{D, E\}$$

$$RS(T_3) = \{A, B\}$$

$$WS(T_3) = \{C, D\}$$



4) 乐观的并发控制

为此，系统保持两个集合：

- **FIN** = 已做完三个阶段的事务（全做完）
- **VAL** = 已成功完成第二阶段(校验)的事务

4) 乐观的并发控制

Validation rules for T_j :

(1) When T_j starts phase 1:

$\text{ignore}(T_j) \leftarrow \text{FIN}$ {此时已在FIN中的事务
校验时无需再考虑}

(2) at T_j Validation:

if **check (T_j)** then

[$\text{VAL} \leftarrow \text{VAL} \cup \{T_j\}$; {通过校验}

do write phase;

$\text{FIN} \leftarrow \text{FIN} \cup \{T_j\}$ {全做完}]

Check (T_j):

检查跟T_j的读并发了的写，这一刻它可能完了也可能没完

For T_i ∈ VAL - IGNORE (T_j) DO

IF [WS(T_i) ∩ RS(T_j) ≠ ∅ OR

(T_i ∉ FIN AND WS(T_i) ∩ WS(T_j) ≠ ∅)]

THEN RETURN false;

RETURN true;

检查将跟T_j的写并发的写，只有这一刻尚未完的事务才可能

例子:

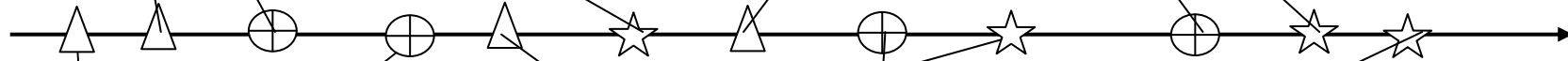
△ start
⊕ validate
☆ finish

U: $RS(U) = \{B\}$

$WS(U) = \{D\}$

W: $RS(W) = \{A, D\}$

$WS(W) = \{A, C\}$



T: $RS(T) = \{A, B\}$

$WS(T) = \{A, C\}$

V: $RS(V) = \{B\}$

$WS(V) = \{D, E\}$

4) 乐观的并发控制



关键点:

- 保证校验是原子的;
- 如果 T_1, T_2, T_3, \dots 是通过校验的序, 则调度冲突等价于 $S_s = T_1 T_2 T_3 \dots$

8.5 数据库的恢复

- 数据库在运行过程中，总是从一种完整状态转变为另一种完整状态。
- 但是，由于种种原因(软硬故障) 事务可能被撤消、重启，使得数据库停留在某一转变过程中间（不完整状态）。
- 为了清除被撤消事务对数据库造成的影响，保证已交付事务的结果不丢失（达到新的完整状态），以及在系统故障修复后使数据库恢复到可用状态，需要使用恢复机制。



关键问题

未完事务

Example

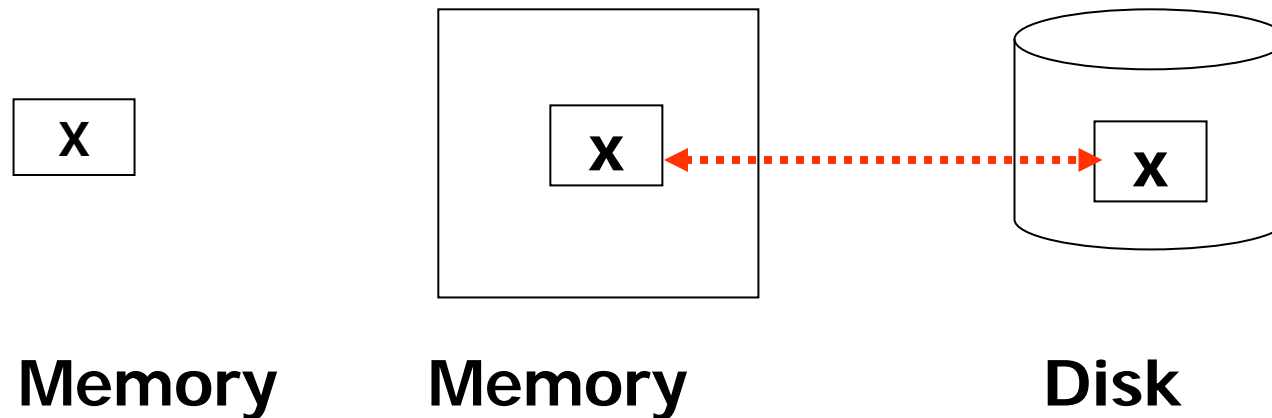
Constraint: $A=B$

$T_1: A \leftarrow A \times 2$


$B \leftarrow B \times 2$

8.1 事务管理的基本概念

Storage hierarchy



- The space of disk blocks
- The virtual or main memory address space
- The local address space of the transaction



T1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
~~Read (B,t); $t \leftarrow t \times 2$~~
Write (B,t);
Output (A);
Output (B);

失败!

A: ~~8~~ 16
B: ~~8~~ 16

memory

A: ~~8~~ 16
B: 8

disk



atomicity: 要不执行事务全部动作要不一个动作都不执行。

8.5.3 主要恢复机制

DBMS应维持足够的信息用以将数据库恢复到一致的状态。

(1) 日志 (log)

日志内容一般包括：

<start_transaction, T>：事务**T**开始执行的标记。

<write_item, T, x, old_value, new_value>：

事务**T**将数据项**x**的值由**old_value**修改为**new_value**。

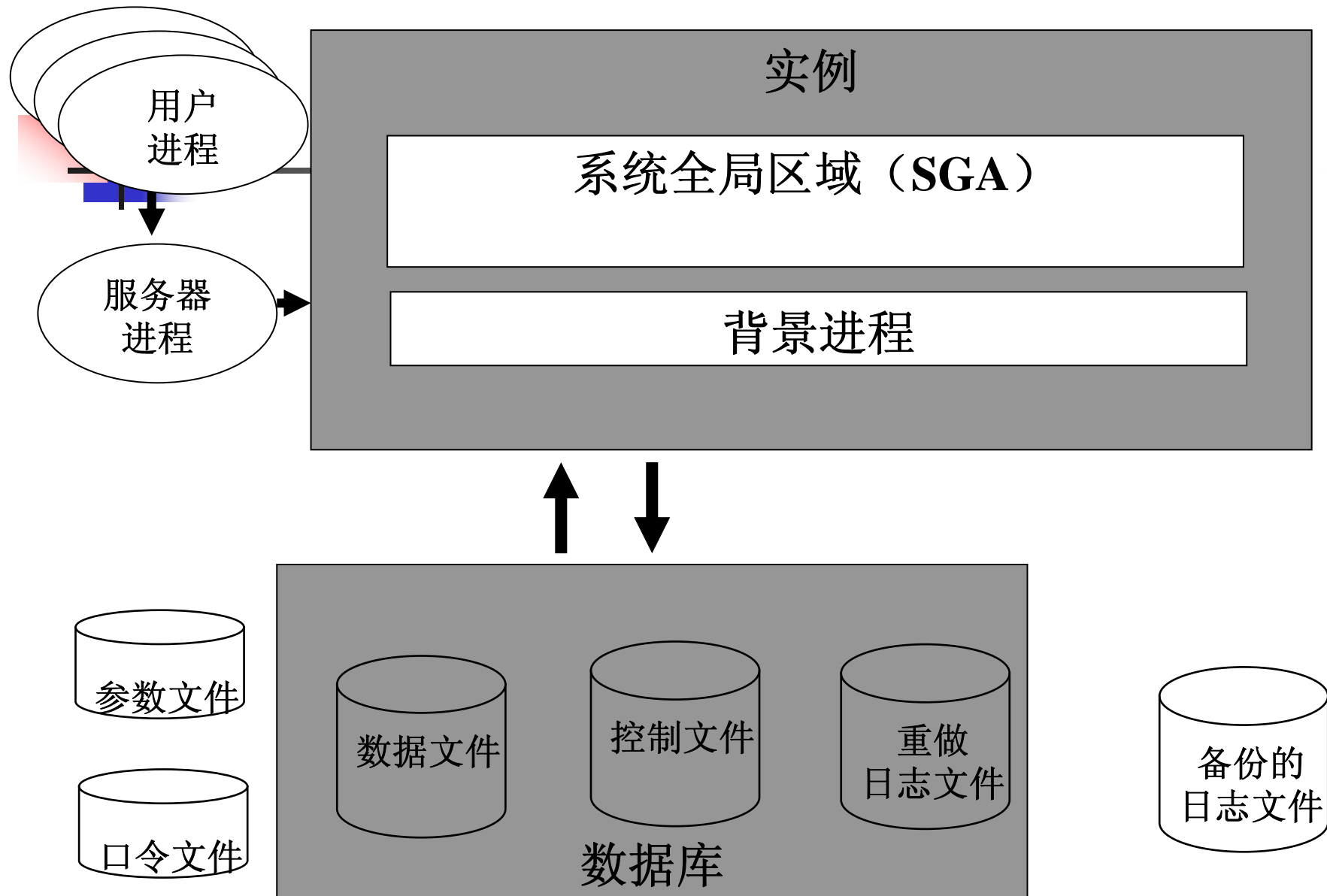
<read_item, T, x>：事务**T**读取数据项**x**的值。

<commit_transaction, T>：

事务**T**完成所有操作，交付。

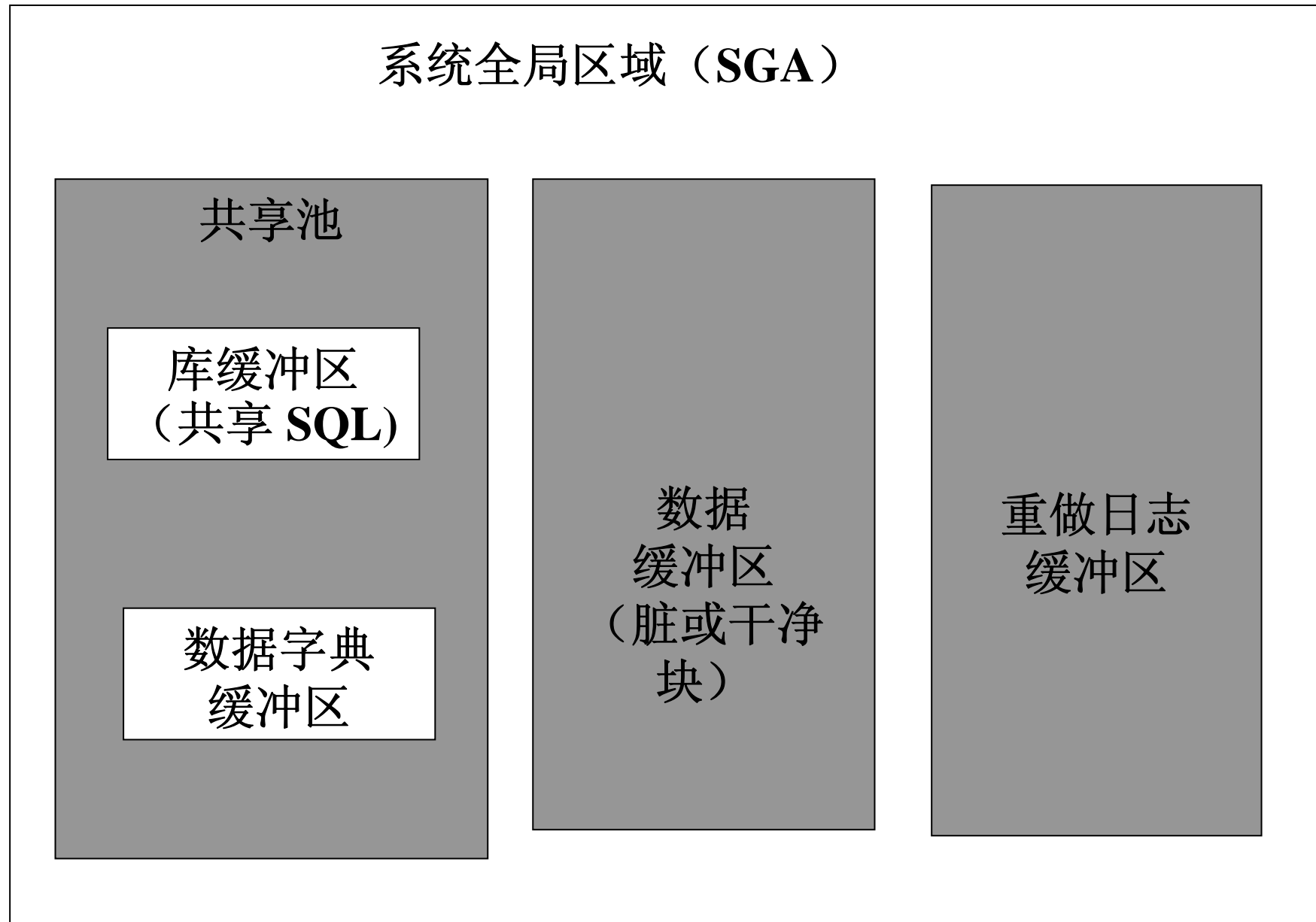
<abort_transaction, T>：事务撤销。

Oracle体系结构

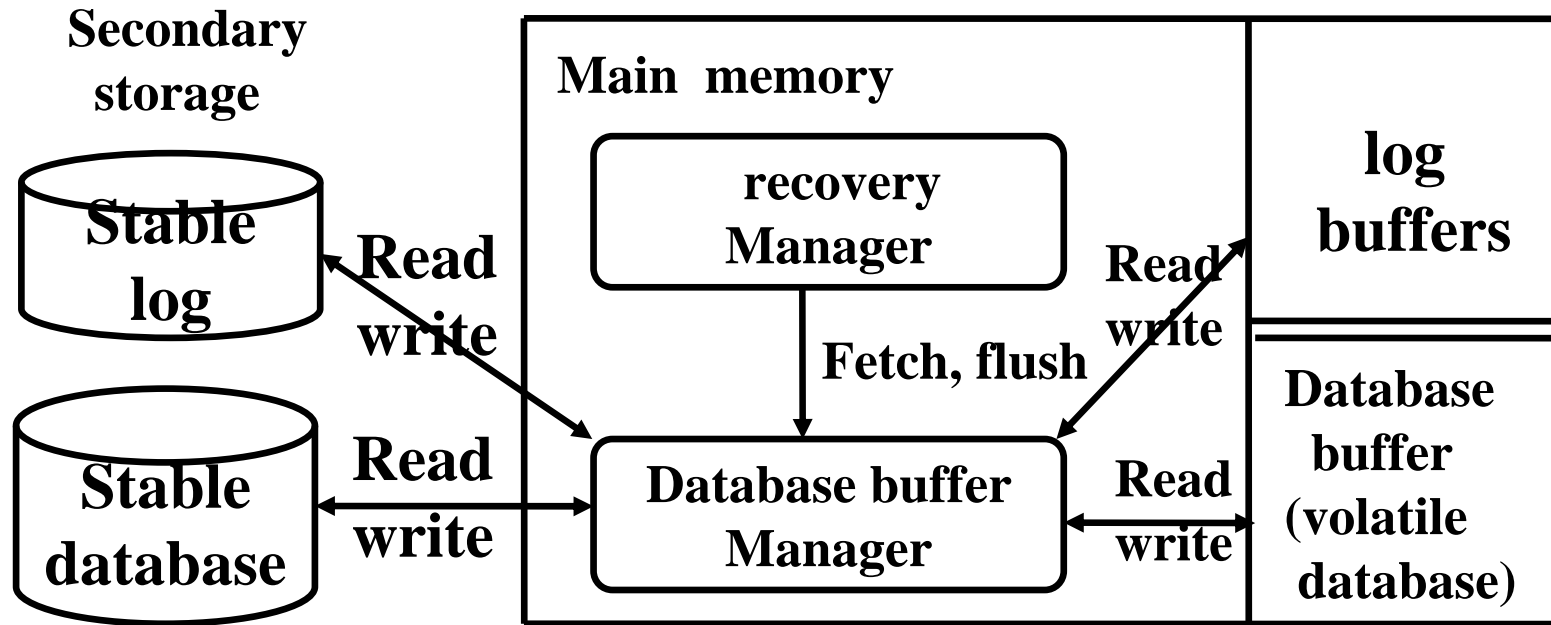


Oracle体系结构

系统全局区域（SGA）



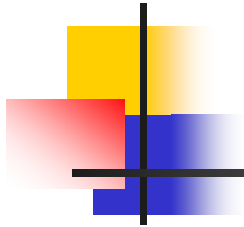
日志接口



同步写日志：在添加每个日志记录时，要求把日志记录从主存写入稳定存储。

异步写日志：日志记录周期地或缓冲满时写入稳定存储。

(2) Redo和Undo



Redo和**Undo**是针对事务的一次操作。
Redo就是重新执行这次操作，**Undo**则是消除这次操作的影响，将事务执行点回退到这次操作之前。

(3) 回退程序

事务的回退（**rollback**）不仅意味着事务程序的重新运行，还包括清除事务已经对数据库造成的影响，完成事务回退工作的程序为回退程序。

事务的回退过程就是由若干**Undo**操作组成，其执行顺序与被**Undo**操作在日志中出现的顺序正好相反。

回退程序还须考虑到事务的隔离性，如果并发控制与恢复机制没有在事务处理中保证事务的隔离性，回退的结果必将导致若干事务的依次回退(**cascading rollback**)，甚至需要破坏事务的持久性来保证事务的正确执行。

（4）检查点

（5）像本

像本是整个数据库或某些部分在某一时刻的副本。像本的使用可以有效地解决由于磁盘故障引起的问题。建立像本就是把数据库中的数据转储到一个后援副本上，这一过程称为倒库（**dump**），一般定期进行。（备份工具名称：**BUCKUP,COPY,DUMP,EXPORT**）



8.5.4 基于立即修改的恢复技术

1. undo logging

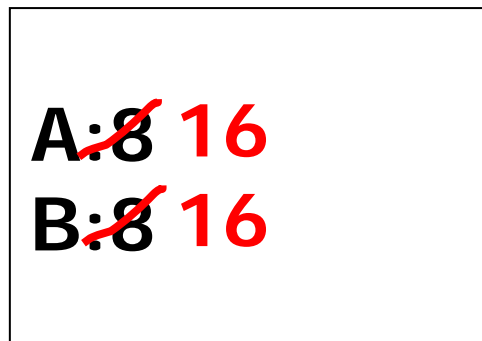
(immediate modification)

· <write_item, T , x, old_value>

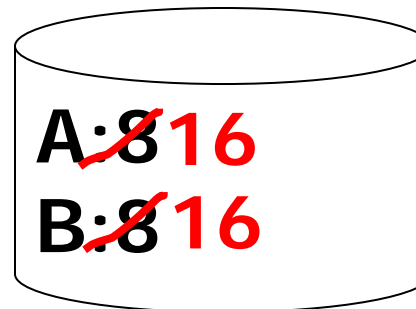
Undo logging (Immediate modification)

T1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);

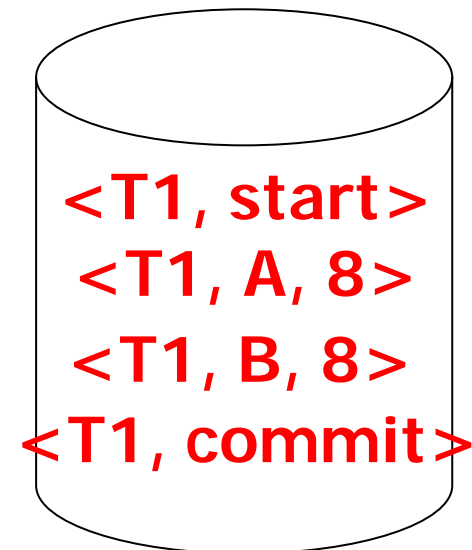
A=B



memory



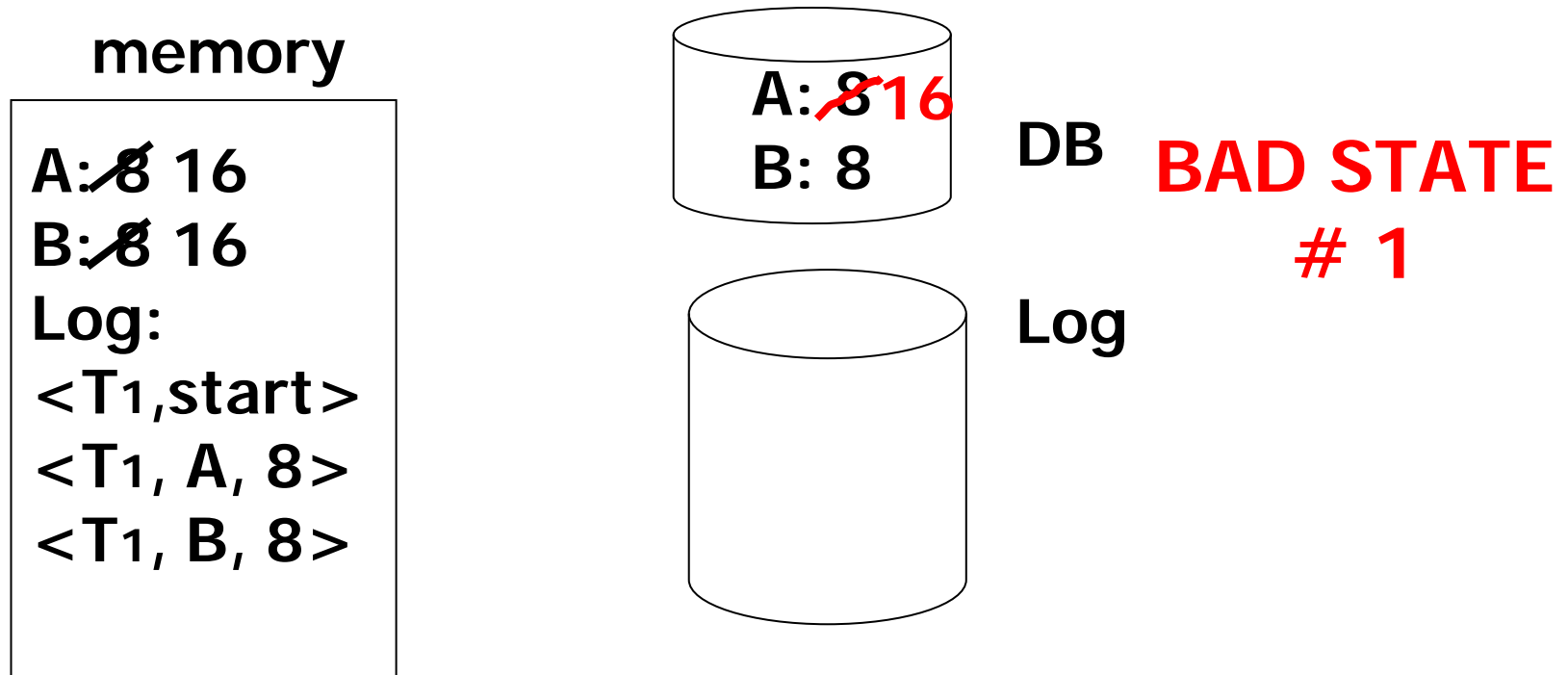
disk



log

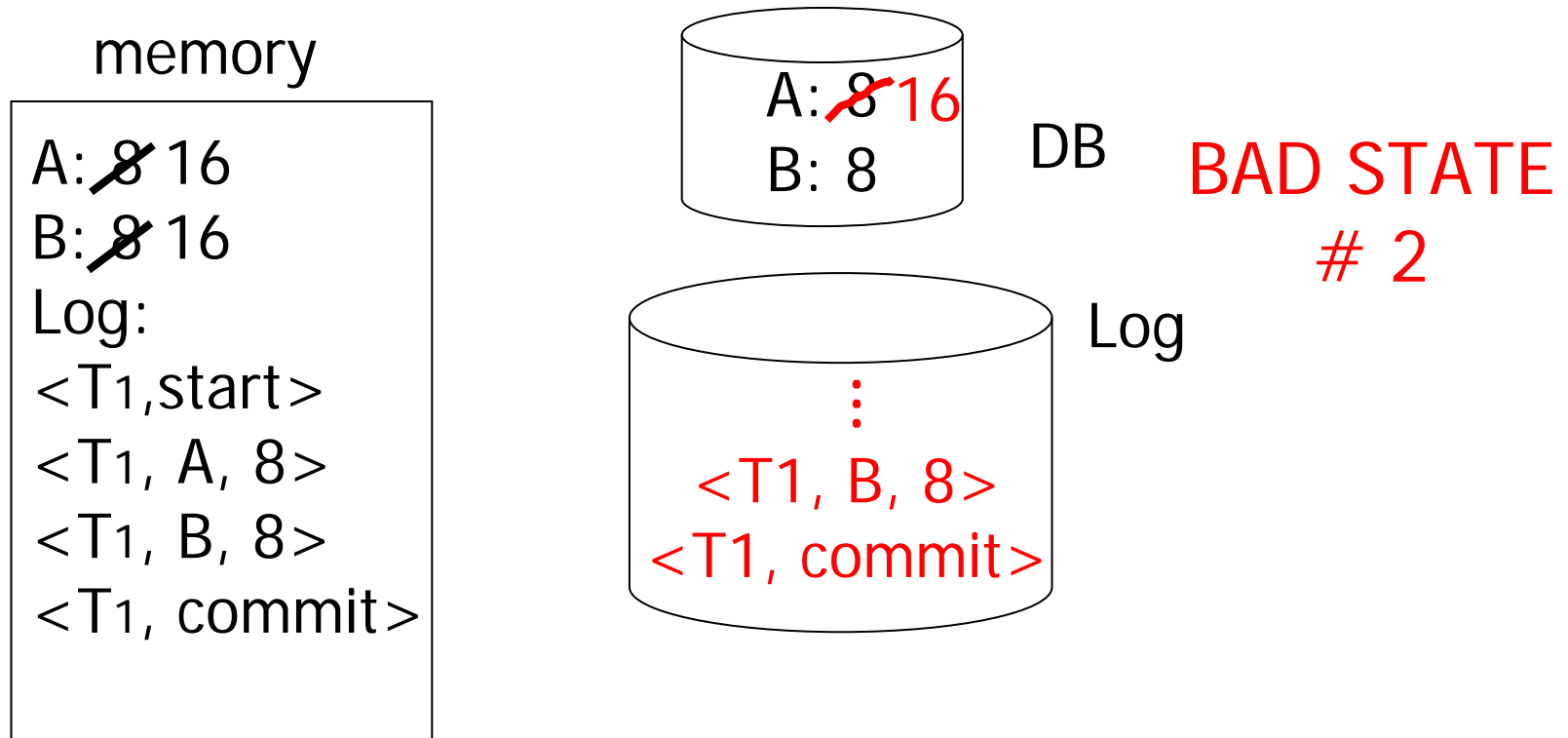
问题1

- Log is first written in memory
- Not written to disk on every action



问题2

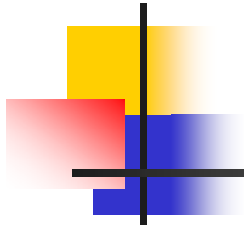
- Log is first written in memory
- Not written to disk on every action





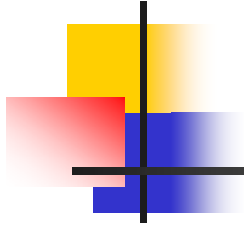
基于立即修改的恢复技术的协议：

- 直到事务的所有**Undo**型记录全部回写到磁盘后，它才可以修改物理数据库；
- 直到事务的所有**Undo**类型记录全部回写到磁盘后，才可以完全交付。



这种恢复技术中，事务的中间结果可能（与并发控制机制有关）会被其它事务引用，破坏了事务的隔离性，因此必须考虑到事务流产所引起的嵌套异常终止，在事务回退处理时可能需要处理许多相关的事务。

假设并发控制使用严格两段锁和死锁防止机制（那么事务执行到达终点时才会释放所有对数据项所加的锁）



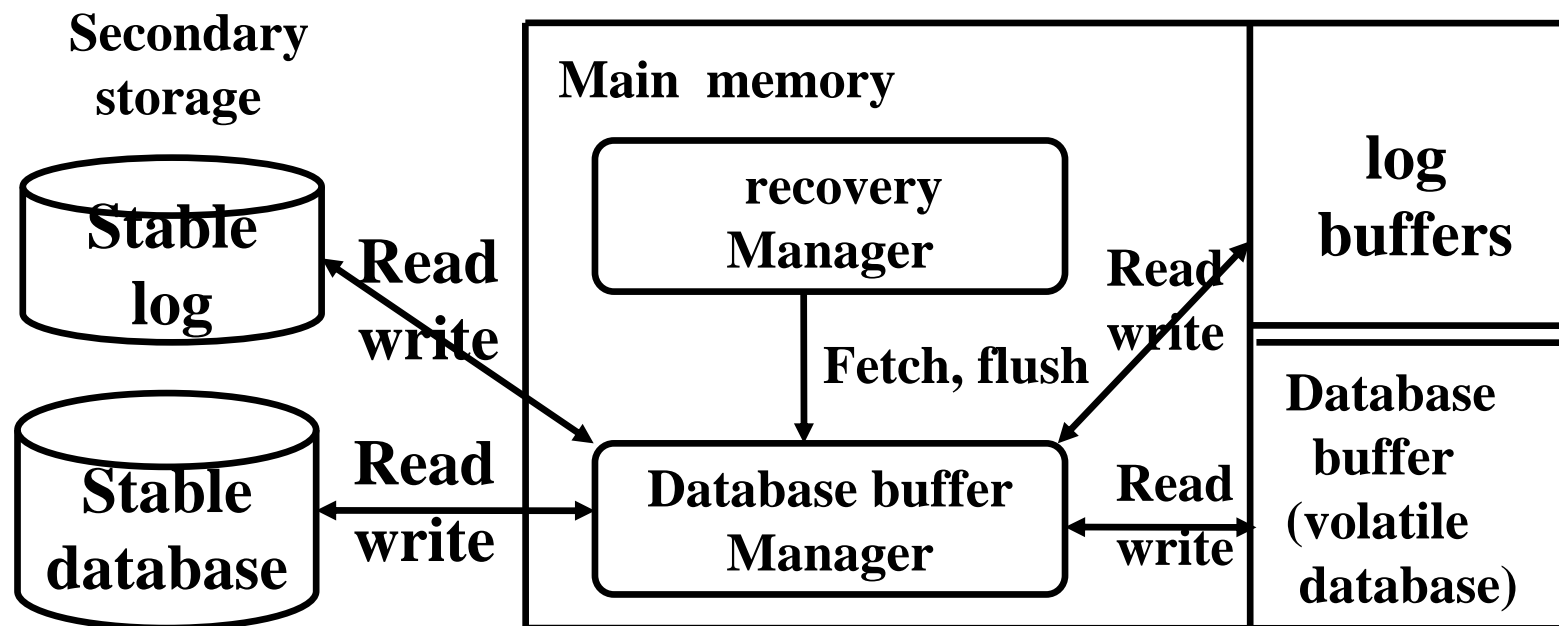
恢复过程中又出故障怎样?

No problem!



Undo idempotent

日志接口



8.3 基于锁的并发控制机制

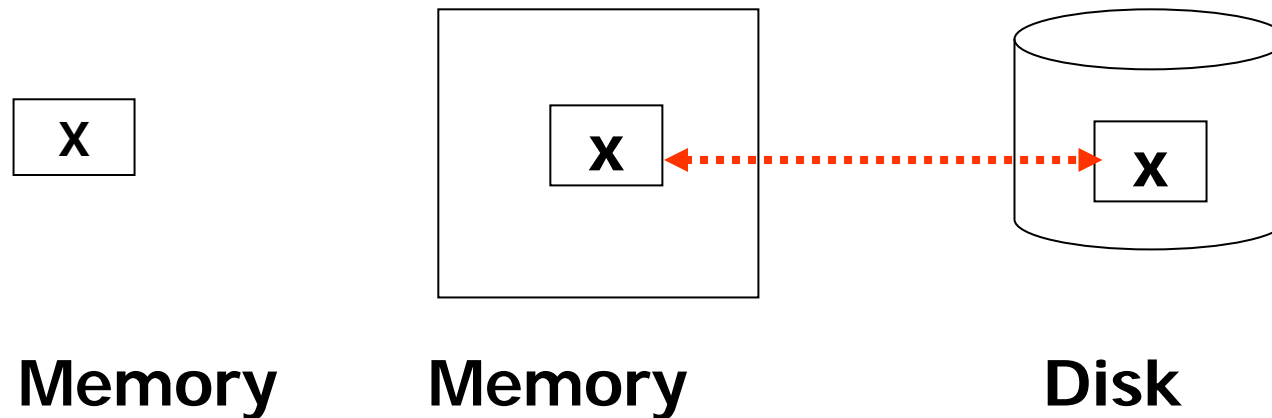
8.4 其它并发控制机制

- 基于时标的并发控制机制（多版本）
- 乐观并发控制

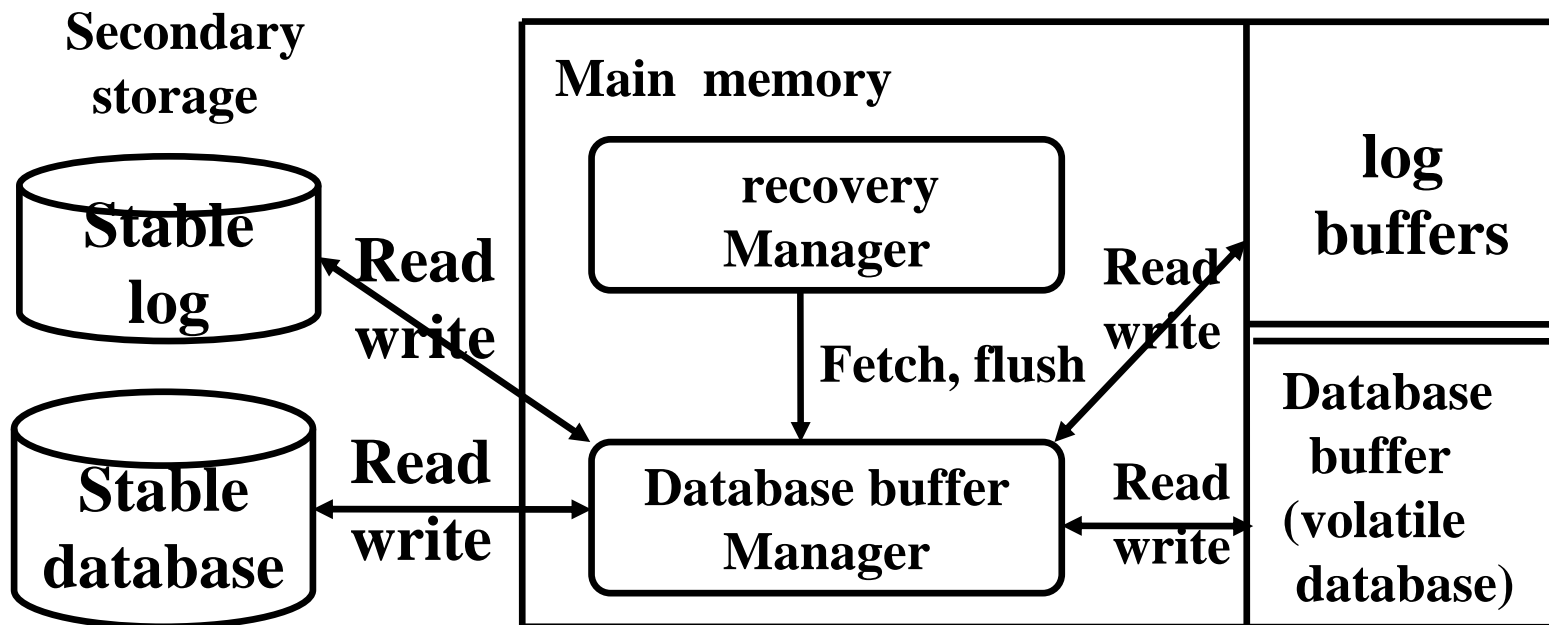
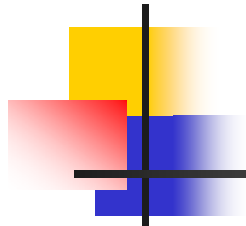
8.5 数据库的恢复

- 主要恢复机制
- 基于立即修改的恢复技术
 - Undo日志

Storage hierarchy



- The space of disk blocks
- The virtual or main memory address space
- The local address space of the transaction



Undo 日志规则

- (1) 每个写操作产生一个**undo**日志记录 (包含旧值);
- (2) 在磁盘上修改 x 之前, 关于修改 x 的日志记录必须先写到磁盘上 (先写日志: **WAL**);
- (3) **commit** 日志记录写到日志文件之前, 事务的所有写操作必须反映到磁盘上。

恢复规则（基于 Undo 日志）：

- (1) S = 日志中有 $\langle Ti, \text{start} \rangle$ 记录，但无 $\langle Ti, \text{commit} \rangle$ （或 $\langle Ti, \text{abort} \rangle$ ）记录的所有事务；
- (2) 逆序扫描日志（由后往前），对每个 $\langle Ti, X, v \rangle$ 记录：
 - 如果 $Ti \in S$ 则 $\left\{ \begin{array}{l} - \text{write}(X, v) \\ - \text{output}(X) \end{array} \right.$
- (3) 对每个 $Ti \in S$ ，
 - 在日志中写上 $\langle Ti, \text{abort} \rangle$ 记录

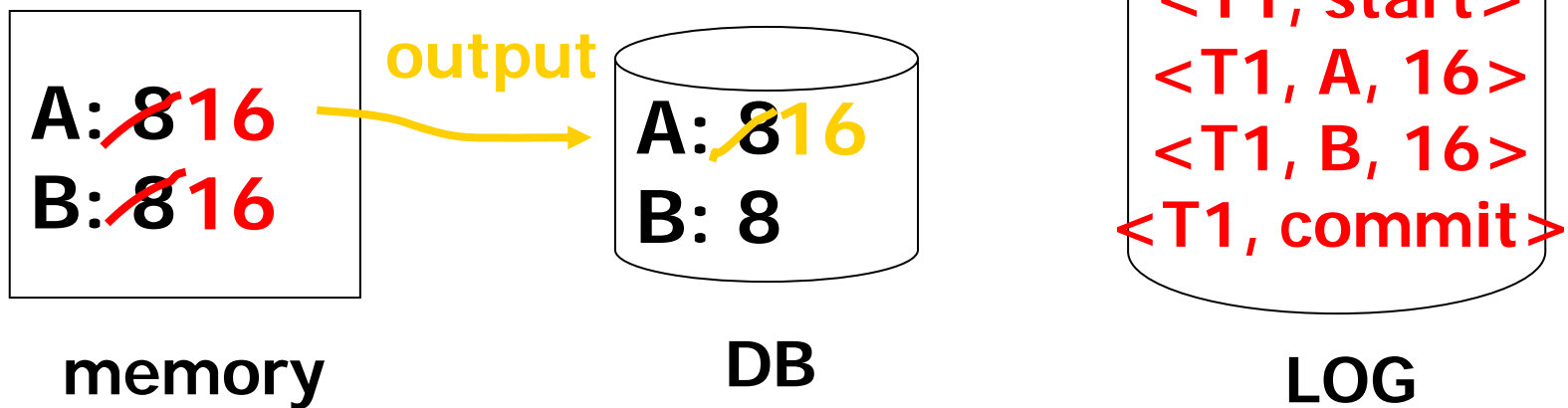


8.5.4 基于延迟修改的恢复技术

1. Redo logging **(deferred modification)**

- 基于延迟修改的恢复技术的基本思想是将事务中所有的写操作都延迟到终点完成。
- 在事务执行过程中，对数据的修改动作保存在日志和事务空间中（以利于事务本身引用操作结果）。当事务到达终点时成功交付，系统将日志写回磁盘，并将修改数据写到数据库。
- 如果事务在中途流产，由于事务对数据库的写操作实际上并没有进行，也就不会产生任何影响，因此就不需要恢复数据库内容。
- 此时，日志中的<write_item...>记录只需包含new_value。

**T1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)**





Redo 日志规则

- (1) 每个写操作产生一个**redo**日志记录
(包含新值);
- (2) 在磁盘上修改 **x** 之前, 关于修改 **x** 的
日志记录 (包括**commit**日志) 必须先写
到磁盘上;
- (3) **Commit**时所有日志写到磁盘上。

Note: 何时在磁盘(**DB**)上修改 **x** ?

恢复规则: (基于 Redo 日志)

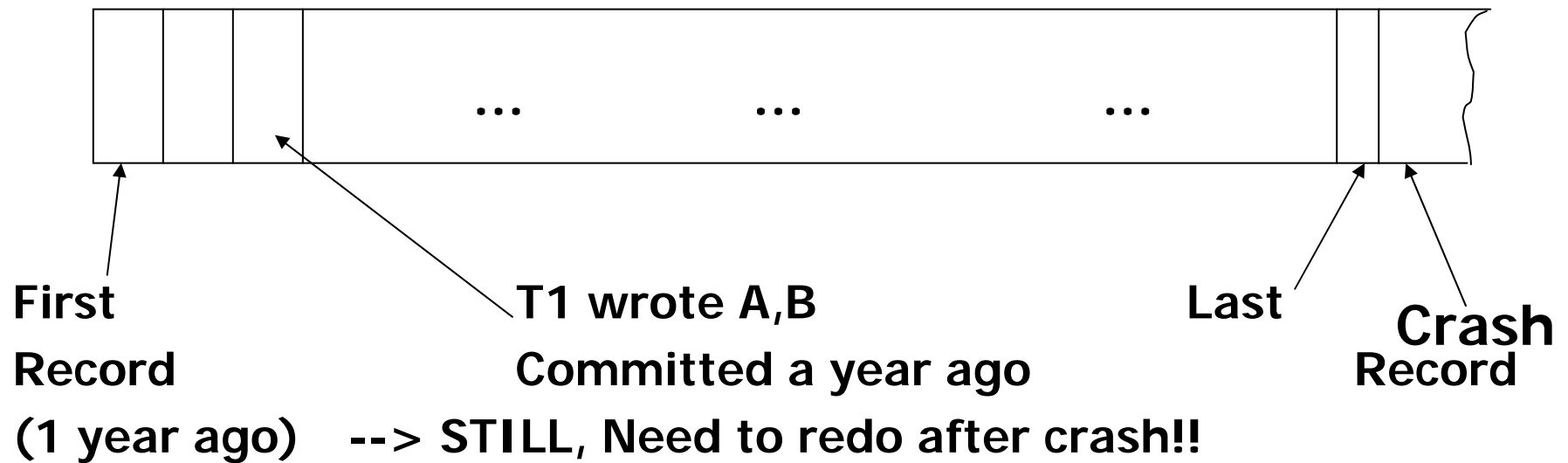
(1) S = 在日志中有 $\langle T_i, \text{commit} \rangle$ 记录的事务集合;

(2) 顺序扫描日志 (由前往后), 对每个 $\langle T_i, X, v \rangle$ 记录:

- 如果 $T_i \in S$ 则 $\begin{cases} \text{Write}(X, v) \\ \text{Output}(X) \text{ (可选)} \end{cases}$



Redo log:



Recovery is very, very **SLOW** !

Solution: Checkpoint

DBMS一般在某些时刻（称作检查点）将驻留在内存中的数据写回数据库并在日志中记录，避免系统停机等故障所造成的损失，也利于事务的恢复执行，此时无需大范围的查找日志和重做事务。

系统检查点：系统周期性检查的时刻；

事务检查点：用户在事务中设置的，要求系统记录事务的状态点。

Checkpoint (simple version)

Periodically:

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

恢复时做什么？

Redo log (disk):

⋮	<T1,A,16>	⋮	<T1,commit>	⋮	Checkpoint	⋮	<T2,B,17>	⋮	<T2,commit>	⋮	<T3,C,21>	Crash
---	-----------	---	-------------	---	------------	---	-----------	---	-------------	---	-----------	-------

Key drawbacks:

- *Undo logging*: increase the number of disk I/O's
- *Redo logging*: need to keep all modified blocks in memory until commit



5. undo/redo logging!

Update \Rightarrow $\langle Ti, Xid, \text{New } X \text{ val}, \text{Old } X \text{ val} \rangle$
page X

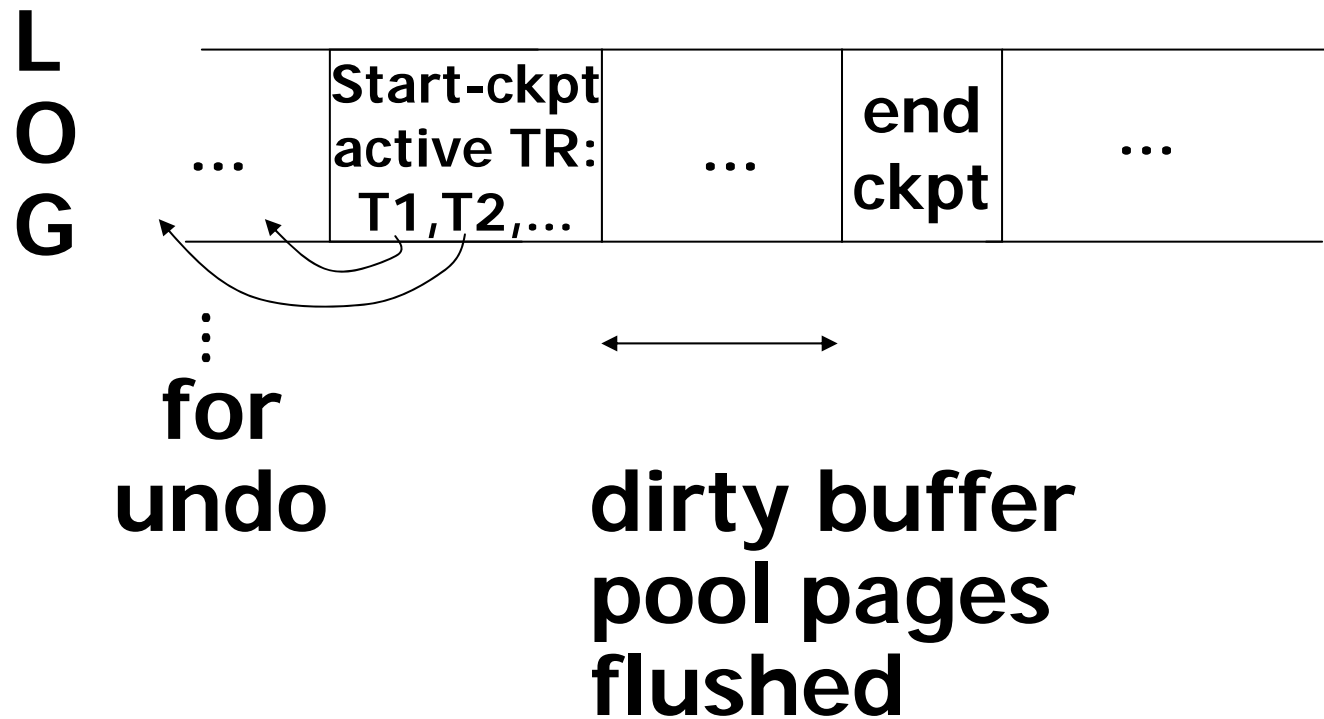


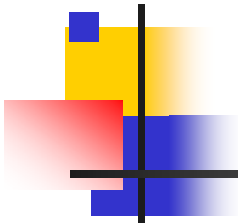
Rules

- Page X can be flushed before or after Ti commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

检查点开始记录（在日志中）一般包括：

- 活动事务表；
- 每一活动事务在日志中的第一条和最后一条记录的位置等。

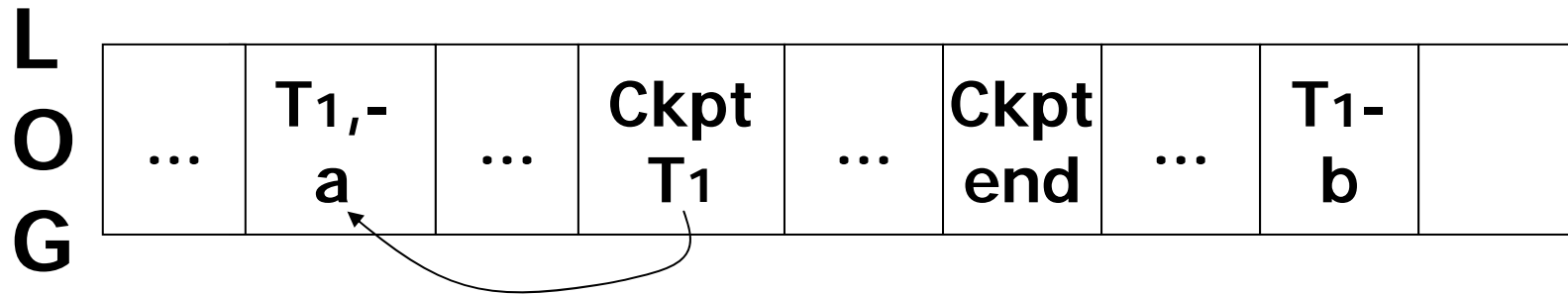




DBMS在检查点处的工作:

- ①产生检查点开始记录写到日志，暂停所有正在执行的事务；
- ②将在内存中驻留的所有数据回写到数据库；
- ③产生检查点结束记录写到日志；
- ④恢复所有被暂停的事务。

no T1 commit



☒ Undo T₁ (undo b, a)



Example

L

O

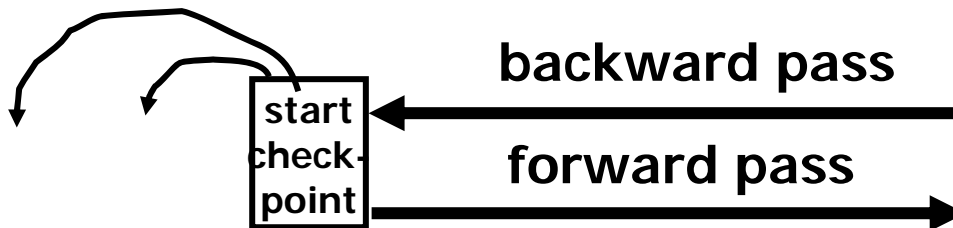
G

...	T1 a	...	ckpt-s T1	ckpt- end	...	T1 b	...	T1 cmt	...
-----	---------	-----	--------------	-----	--	-----	--------------	-----	---------	-----	-----------	-----

☒ Redo T1: (redo b)

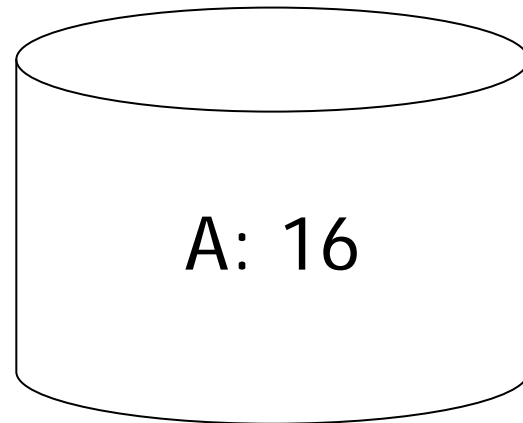
Recovery process:

- **Backwards pass** (end of log \Rightarrow latest checkpoint start)
 - construct set S of committed transactions
 - undo actions of transactions not in S
- **Undo pending transactions**
 - follow undo chains for transactions in (checkpoint active list) - S
- **Forward pass** (latest checkpoint start \Rightarrow end of log)
 - redo actions of S transactions



8.5.6. Media failure (loss of non-volatile storage)

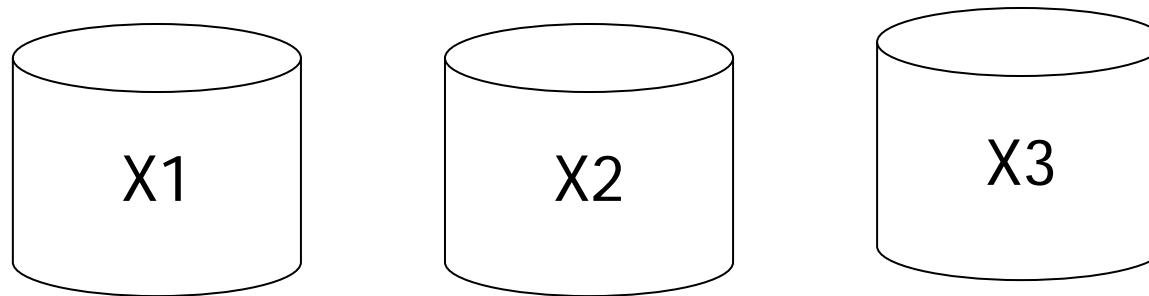




Solution: Make copies of data!

方案 1 Triple modular redundancy

- Keep 3 copies on separate disks
- Output(X) --> three outputs
- Input(X) --> three inputs + vote

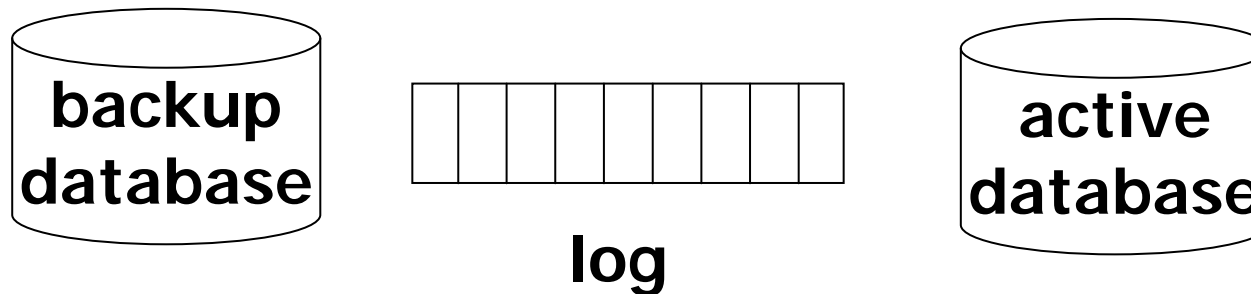




方案2 Redundant writes, Single reads

- Keep N copies on separate disks
 - Output(X) --> N outputs
 - Input(X) --> Input one copy
 - if ok, done
 - else try another one
- ⇔ Assumes bad data can be detected

方案3: DB Dump + Log



- If active database is lost,
 - restore active database from backup
 - bring up-to-date using redo entries in log
 - ARCHIVELOG or NOARCHIVELOG



数据库备份与恢复是DBA职责

- 至少两个本地副本
- 至少两代备份副本
- 先做磁盘副本，再移到磁带
- 除数据文件、日志文件和控制文件外，备份系统目录
- 备份完用**DBMS**工具检查
- 不能过于频繁又要足够频繁
- 完全备份与增量备份



Oracle的备份

- 逻辑备份（利用EXP/IMP导出和导入整个数据库或某个用户的模式或某个表）
- 物理备份（直接拷贝数据、日志、控制文件）
 - 冷备份（脱机即关闭数据库实例）
 - 热备份（联机且在**ARCHIVELOG** 方式）

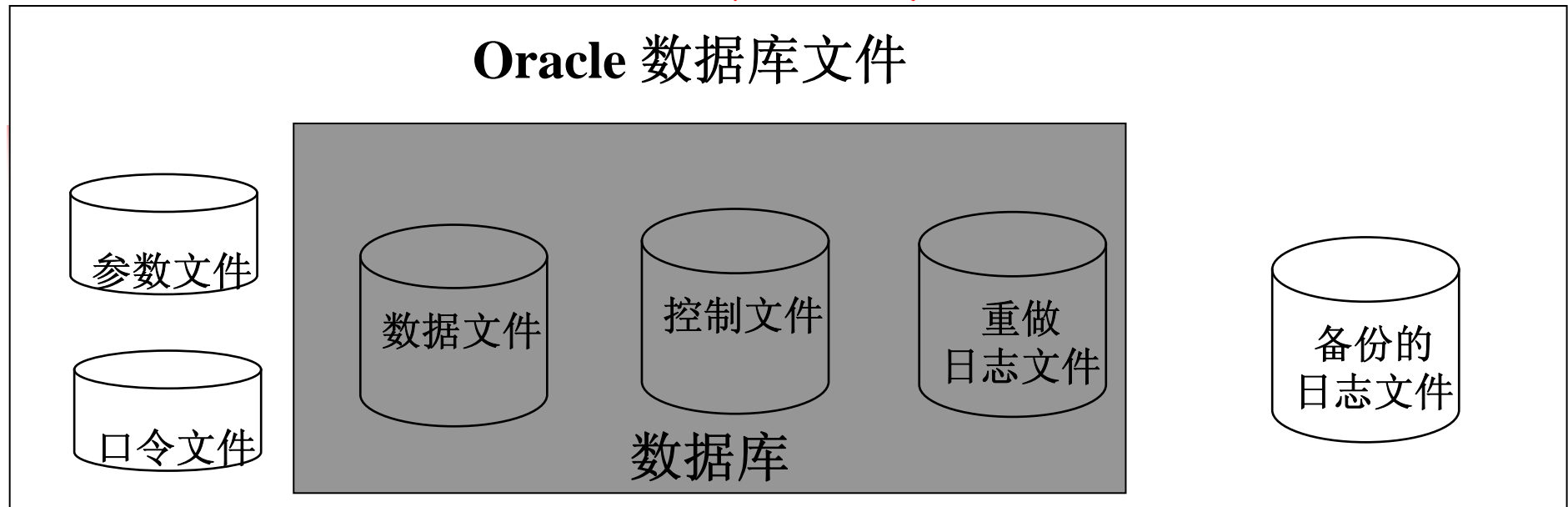
详见文档并上机实习



上机要求

- 掌握冷备份、热备份、导入和导出的概念，熟悉在Oracle环境下实现备份和恢复的基本操作。
- 掌握Oracle的内存结构和进程结构，了解其配置运行方案，掌握如何以手工方式启动和关闭数据库，初步理解数据库初始化配置文件。
- 了解Oracle的物理结构和逻辑结构，掌握管理表空间、日志、数据块、索引和数据库连接的操作命令，并力图在此基础上进一步理解数据库优化的必要性和可能性。

Oracle体系结构(4/4)



数据文件包含了存储在数据库中的实际数据。

控制文件包含了用来维护和验证数据库完整性的信息。

重做日志文件记录了用户对数据库所作的任何修改。

参数文件用来定义**Oracle** 实例的特征。

口令文件用来认证用户是否合法。

备份的日志文件是重做日志文件的脱机拷贝。