

西北工业大学

Northwestern Polytechnical University

数据库系统原理

Database System

第十一章 数据库恢复技术

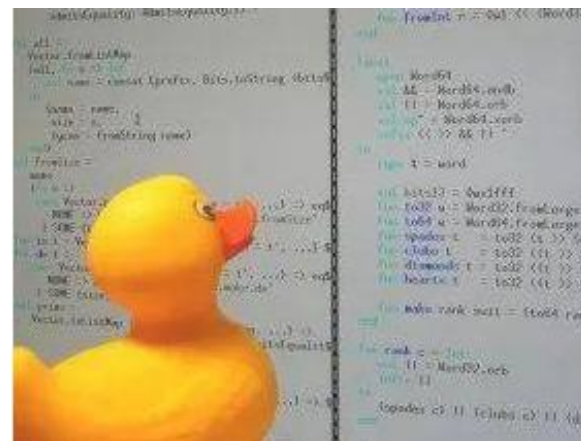
2024.11

精髓：以讲代学



物理学家

理查德·费曼



程序员文化：小黄鸭

查询处理

查询优化

事务处理

数据存储

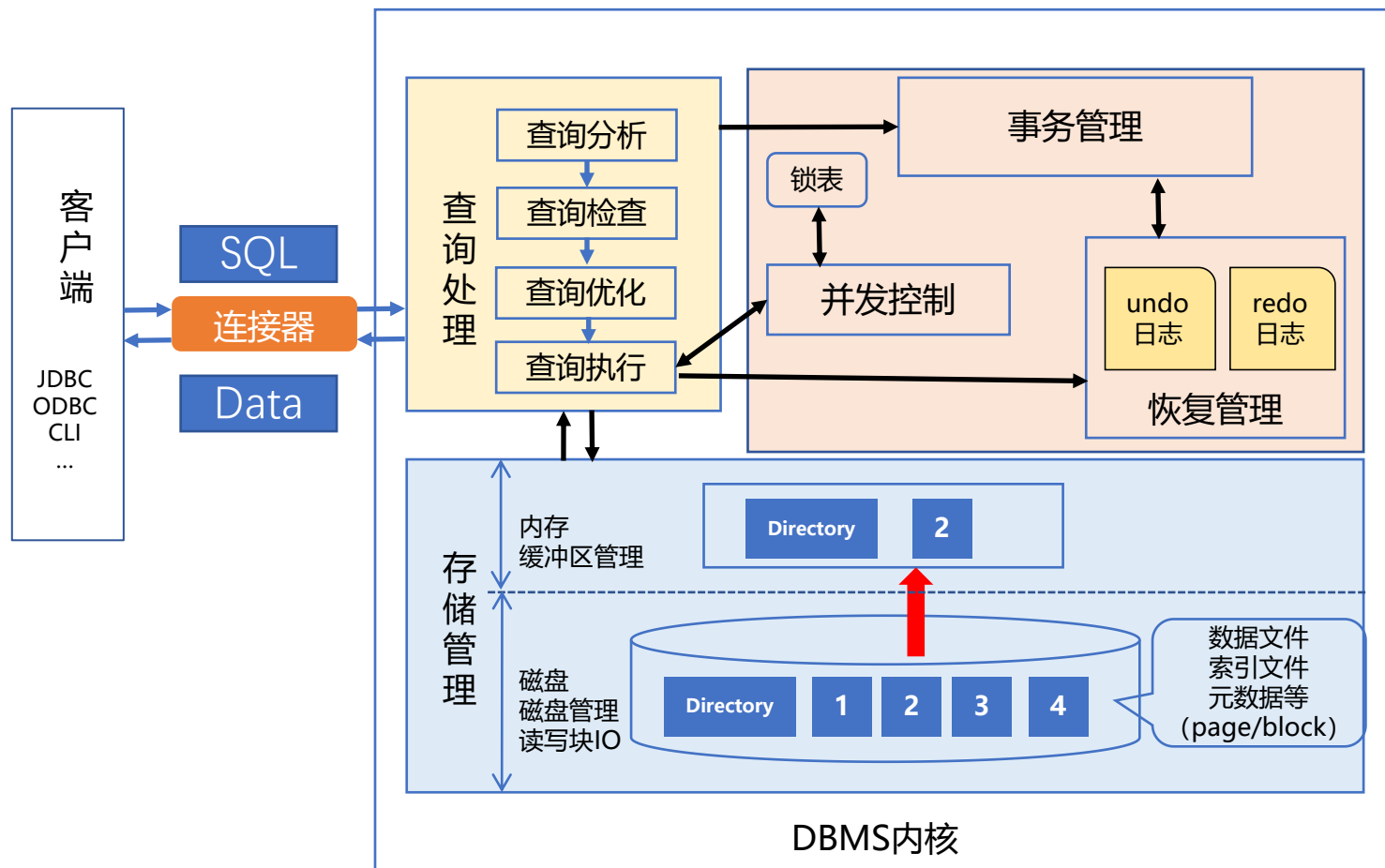


第9章 数据存储与索引

- DBMS的数据存储架构
- 记录、页（变长与定长记录）
- 文件的组织形式
- 缓冲区管理（Buffer Pool）
- 索引结构（B+树、Hash索引）

第10章 查询处理与查询优化

- 查询分析（词法、语法、语义）
- 查询处理（选择算子、连接算子）
- 查询优化（代数优化、物理优化）
- 查询计划执行（火山模型）



- 11.1 事务的基本概念**
- 11.2 数据库恢复概述**
- 11.3 故障的种类**
- 11.4 恢复的实现技术**
- 11.5 恢复策略**
- 11.6 具有检查点的恢复技术**
- 11.7 数据库镜像**
- 11.8 MySQL中的日志**



阿里天猫双11的数据，猜猜看2020年双11，支付宝每秒支付笔数！

- ☐ A 33.8万
- ☐ B 56.8万
- ☒ C 58.3万
- ☐ D 68.3万

| 年度 | 订单创建/秒 | 支付峰值 |
|------|---------|-------|
| 2020 | ? | ? |
| 2018 | 48 万笔/秒 | ? |
| 2015 | 14万 | 8.59万 |
| 2012 | 1.4万 | 3,850 |
| 2009 | 400 | 200 |



提交

双11 “万亿交易额” 背后，浪潮信息助力银行扛住交易洪流

浪潮企业级解决方案 2023-11-08 14:18 北京

双十一，不仅是网络购物的狂欢，更是中国支付清算业务的大考。举目望去，双十一的台前幕后可谓是“不一样的精彩”。一方面台前的主角是消费者，全球超200个国家和地区的人们捧着手机、电脑，在阿里、京东、抖音、拼多多等电商平台上拼秒杀、抢红包，通过“买买买”创造了超万亿元的交易额，带动了零售、物流业的发展；另一方面，双十一是对金融支付清算系统基础设施的“大考”：当数十亿人同时拿起手机下单，将汇聚起排山倒海的数据洪流，冲向网联、银联、商业银行等金融机构的关键交易平台，让服务器、存储等金融基础设施的负载瞬间飙到高峰。

这里以央行公开披露数据为例，2021年“双十一”期间（11月1日至11月11日），联、银联共处理支付交易270.48亿笔，**金额22.32万亿元**。其中11日当天，网联、银联合计最高业务**峰值9.65万笔/秒**。



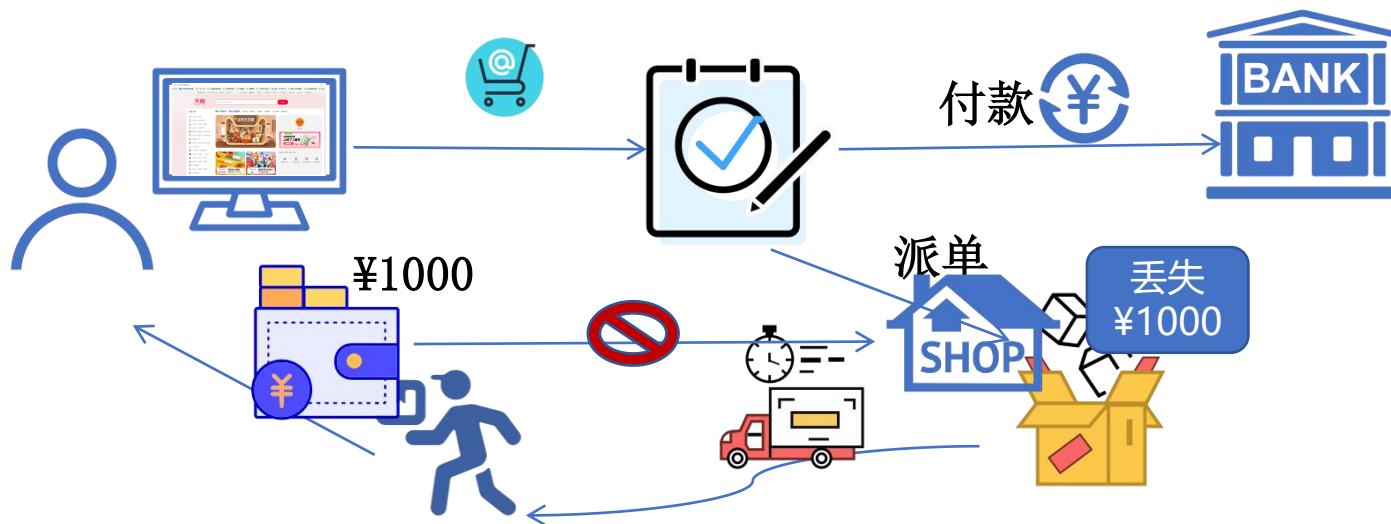
据阿里巴巴官网，11月1日0点至11月11日0点30分，天猫双十一实时成交额破3723亿，订单创建**峰值达58.3万个/秒**。

澎湃新闻记者 王嘉仪 实习生 林美汕 素材来源 阿里巴巴官网 责任编辑：李蕊

11.1 事务的基本概念



一个“双十一”订单与DBMS的关联



- 电商平台的数据库
- 银行的数据库
- 物流公司的数据库
-

1. 事务的概念

- 一个数据库操作序列
- 一个不可分割的工作单位（要么全做，要么全不做）
- 恢复和并发控制的基本单位

在关系数据库中的事务和程序：

- 一个事务可以是一条SQL语句，一组SQL语句或整个程序。
- 一个程序通常包含多个事务。

2. 事务的特性(ACID)

要么全做，要么全不做

➤ 原子性 (Atomicity)

➤ 一致性 (Consistency)

事务执行的结果必须是使数据库
从一个一致性状态变到
另一个一致性状态

➤ 隔离性 (Isolation)

一个事务的执行不能被
其他事务干扰

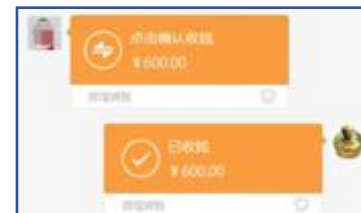
➤ 持续性 (Durability)

事务一旦提交，它对数据库中数据
的改变就应该是永久性的。



事务典型例子 —— **银行转帐**:

从帐号A中取出1000元，存入帐号B。



定义一个事务，该事务包括两个操作：

| A | B |
|------------|------------|
| $A=A-1000$ | |
| | $B=B+1000$ |

- ① 从帐户A减去一百元 ($A=A-1000$)；
- ② 给帐户B加上一百元 ($B=B+1000$)。

该事务应该符合：

- ① 这两个操作要么全做，要么全不做 (**原子性**)
- ② 全做或者全不做，数据库都处于**一致性**状态；
如果只做一个操作，数据库就处于不一致性状态，这是不允许的。

■ 隔离性

一个事务的执行不能被其他事务干扰，而影响它对数据的正确使用和修改。

例：（右图）

事务 T_1 对数据A的修改被 T_2 所影响，造成 T_1 的修改丢失。

| T_1 | T_2 |
|--------------------------------|------------------------------|
| ① 读A=16 | 读A=16 |
| ② | |
| ③ $A \leftarrow A-1$ 写回A=15 | |
| ④ | $A \leftarrow A-3$ 写回A=13 |

■ 如何定义事务？

➤ 显式定义方式

```
start transaction/begin
SQL 语句1
SQL 语句2
。 。 。 。 。
commit
```

- 事务正常结束；
- 提交事务的所有操作；
- 事务中所有对数据库的更新写回磁盘上的物理数据库中。

```
start transaction/begin
SQL 语句1
SQL 语句2
。 。 。 。 。
rollback
```

- 事务异常结束；
- 事务运行的过程中发生了故障，不能继续执行；
- 系统将事务中对数据库的所有已完成的操作**全部撤销**。
- 事务滚回到**开始**时的状态

➤ 隐式方式

当用户没有显式地定义事务时，DBMS按缺省规定自动划分事务。默认每条SQL语句是一个事务。

MySQL 事务- Commit



```
mysql> select * from sc;
```

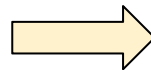
| sno | cno | grade |
|------|-----|-------|
| 2001 | 1 | 92 |
| 2001 | 2 | 85 |
| 2001 | 3 | 90 |
| 2002 | 1 | 98 |
| 2002 | 2 | 78 |
| 2002 | 3 | 84 |
| 2002 | 4 | 91 |
| 2009 | 1 | 90 |

8 rows in set (0.00 sec)

```
mysql> select @@autocommit;
```

| @@autocommit |
|--------------|
| 0 |

1 row in set (0.00 sec)



```
mysql> start transaction;
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> update sc set grade=92 where sno=2009 and cno=1;
```

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

```
mysql> select * from sc;
```

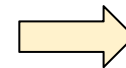
| sno | cno | grade |
|------|-----|-------|
| 2001 | 1 | 92 |
| 2001 | 2 | 85 |
| 2001 | 3 | 90 |
| 2002 | 1 | 98 |
| 2002 | 2 | 78 |
| 2002 | 3 | 84 |
| 2002 | 4 | 91 |
| 2009 | 1 | 92 |

8 rows in set (0.00 sec)



```
mysql> commit;
```

Query OK, 0 rows affected (0.01 sec)



```
mysql> select * from sc;
```

| sno | cno | grade |
|------|-----|-------|
| 2001 | 1 | 92 |
| 2001 | 2 | 85 |
| 2001 | 3 | 90 |
| 2002 | 1 | 98 |
| 2002 | 2 | 78 |
| 2002 | 3 | 84 |
| 2002 | 4 | 91 |
| 2009 | 1 | 92 |

8 rows in set (0.00 sec)

MySQL 事务 - rollback



```
mysql> select * from sc;
```

| sno | cno | grade |
|------|-----|-------|
| 2001 | 1 | 92 |
| 2001 | 2 | 85 |
| 2001 | 3 | 90 |
| 2002 | 1 | 98 |
| 2002 | 2 | 78 |
| 2002 | 3 | 84 |
| 2002 | 4 | 91 |
| 2009 | 1 | 90 |

8 rows in set (0.00 sec)

```
mysql> rollback;
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> select * from sc;
```

| sno | cno | grade |
|------|-----|-------|
| 2001 | 1 | 92 |
| 2001 | 2 | 85 |
| 2001 | 3 | 90 |
| 2002 | 1 | 98 |
| 2002 | 2 | 78 |
| 2002 | 3 | 84 |
| 2002 | 4 | 91 |
| 2009 | 1 | 90 |

8 rows in set (0.00 sec)

```
mysql> start transaction;
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> update sc set grade=92 where sno=2009 and cno=1;
```

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

```
mysql> select * from sc;
```

| sno | cno | grade |
|------|-----|-------|
| 2001 | 1 | 92 |
| 2001 | 2 | 85 |
| 2001 | 3 | 90 |
| 2002 | 1 | 98 |
| 2002 | 2 | 78 |
| 2002 | 3 | 84 |
| 2002 | 4 | 91 |
| 2009 | 1 | 92 |

8 rows in set (0.00 sec)

start transaction;

insert into t values(1);

insert into t values(2);

savepoint s1;

insert into t values(3);

insert into t values(4);

savepoint s2;

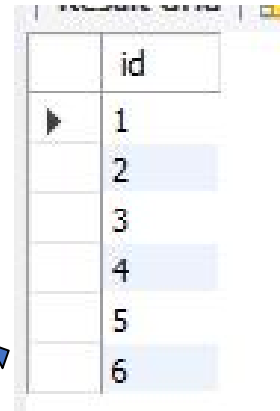
insert into t values(5);

insert into t values(6);

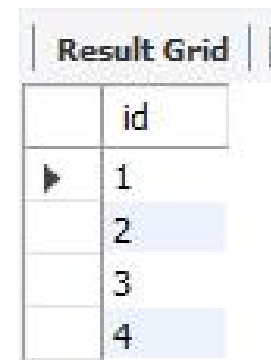
select * from t;

rollback to savepoint s2;

select * from t;



| id |
|----|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

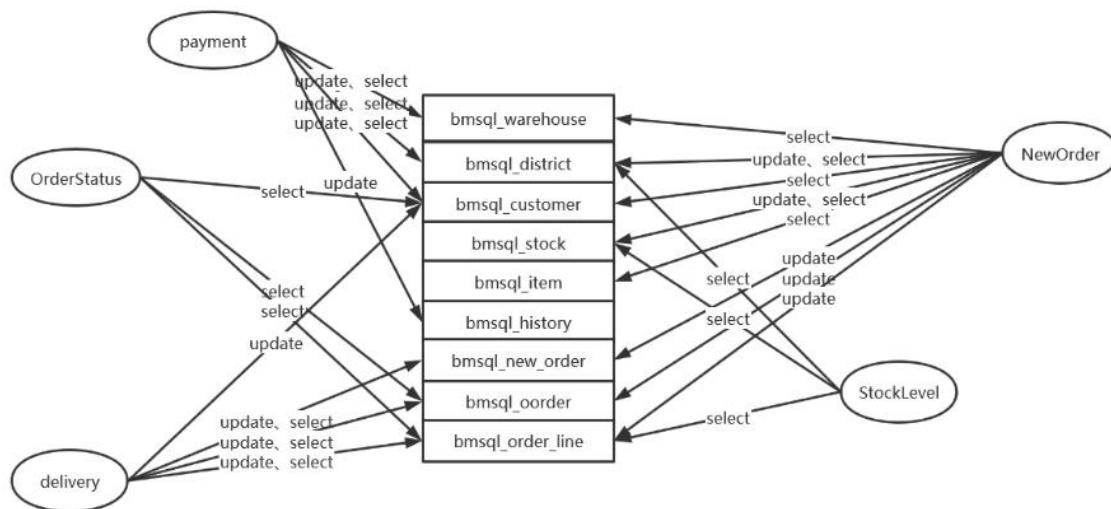


| id |
|----|
| 1 |
| 2 |
| 3 |
| 4 |

■ 数据恢复的基本单位：事务（ACID）

该系统需要处理的交易为以下几种：

- 1) New-Order: 客户输入一笔新的订货交易；默认45%
- 2) Payment: 更新客户账户余额以反映其支付状况；默认43%
- 3) Delivery: 发货(模拟批处理交易)；默认4%
- 4) Order-Status: 查询客户最近交易的状态；默认4%
- 5) Stock-Level: 查询仓库库存状况，以便能够及时补货。默认4%



TPCC中的事务示例

■ 数据恢复的基本单位：事务（ACID）

TPCC中的事务示例

| | | | | | | | |
|---------------|-------------------|-----------------|-----------------------------|--------|-----------------------------------|--|--|
| <100:newOrder | newOrderWeight=45 | executeNewOrder | stmtNewOrderSelectDist | Select | bmsql_district | Select for update SELECT d_tax, d_next_o_id FROM bmsql_district WHERE d_w_id = ? AND d_id = ? FOR UPDATE: | 从地区表中根据地区id和地区-仓库id, 获得地区税率和下一个订单的id, 并将数据加锁。 |
| | | | stmtNewOrderSelectWhseCust | Select | bmsql_customer bmsql_warehouse | SELECT c_discount, c_last, c_credit, w_tax FROM bmsql_customer JOIN bmsql_warehouse ON (w_id = c_w_id) WHERE c_w_id = ? AND c_d_id = ? AND c_id = ? | 从顾客表和仓库表中获取客户折扣、客户名、客户信誉和仓库税率 |
| | | | stmtNewOrderUpdateDist | Update | bmsql_district | 非主键更新 UPDATE bmsql_district SET d_next_o_id = d_next_o_id + 1 WHERE d_w_id = ? AND d_id = ? | 更新地区表, 令下一个订单号+1 |
| | | | stmtNewOrderInsertOrder | Update | bmsql_oorder | INSERT INTO bmsql_oorder (o_id, o_d_id, o_w_id, o_c_id, o_entry_d, o_ol_cnt, o_all_local) VALUES (?, ?, ?, ?, ?, ?, ?); | 向订单表插入新的订单信息 |
| | | | stmtNewOrderInsertNewOrder | Update | bmsql_new_order | INSERT INTO bmsql_new_order (no_o_id, no_d_id, no_w_id) VALUES (?, ?, ?); | 向新订单表插入仓库id、地区id、订单id |
| | | | stmtNewOrderSelectItem | Select | bmsql_item | SELECT i_price, i_name, i_data FROM bmsql_item WHERE i_id = ?; | 从物品表根据物品id选择物品的价格、名字、数据 |
| | | | stmtNewOrderInsertOrderLine | Update | bmsql_order_line | INSERT INTO bmsql_order_line (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_dist_info) VALUES (?, ?, ?, ?, ?, ?, ?, ?); | 向订单状态表插入订单id、订单-地区id、订单-仓库id等信息 |
| | | | stmtNewOrderUpdateStock | Update | bmsql_stock | 非主键更新 UPDATE bmsql_stock SET s_quantity = ?, s_ytd = s_ytd + ?, s_order_cnt = s_order_cnt + 1, s_remote_cnt = s_remote_cnt + ? WHERE s_w_id = ? AND s_i_id = ?; | 根据仓库id和物品id, 更新库存状态表的物品数量、年度余额、订单数量+1、remote_cnt信息 |
| | | | rollback | trans | | | |
| | | | stmtNewOrderSelectStock | Select | bmsql_stock | Select for update SELECT s_quantity, s_data, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05, s_dist_06, s_dist_07, s_dist_08, s_dist_09, s_dist_10 FROM bmsql_stock WHERE s_w_id = ? AND s_i_id = ? | 从库存状态表根据仓库id和物品id选择商品数量、商品数据等信息, 并对数据加锁 |
| | | | stmtNewOrderUpdateStock | Update | bmsql_stock | 非主键更新 UPDATE bmsql_stock SET s_quantity = ?, s_ytd = s_ytd + ?, s_order_cnt = s_order_cnt + 1, s_remote_cnt = s_remote_cnt + ? WHERE s_w_id = ? AND s_i_id = ? | 根据仓库id和物品id, 更新库存状态表的物品数量、年度余额、订单数量+1、remote_cnt信息 |
| | | | stmtNewOrderInsertOrderLine | Update | bmsql_order_line | INSERT INTO bmsql_order_line (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_dist_info) | 向订单状态表插入订单id、订单-地区id、订单-仓库id等信息 |
| | | | commit | trans | | | |





事务的ACID非常重要，如果一个事务的执行结果被另外一个事务干扰了，违反了事务的哪个性质？

- ☐ A 原子性
- ☐ B 一致性
- ☒ C 隔离性
- ☐ D 持久性

11.1 事务的基本概念

11.2 数据库恢复概述

11.3 故障的种类

11.4 恢复的实现技术

11.5 恢复策略

11.6 具有检查点的恢复技术

11.7 数据库镜像

11.8 MySQL中的日志

■ 故障是不可避免的

- 系统故障：计算机软、硬件故障
- 人为故障：操作员的失误、恶意的破坏等。



■ 故障的影响

- 运行中事务非正常中断，影响数据库中数据的正确性。
- 破坏数据库，全部或部分丢失数据。

■ 数据库的恢复

把数据库从**错误状态**恢复到**某一已知的正确状态** (亦称为一致状态或完整状态)。

恢复技术是衡量系统优劣的重要指标

11.1 事务的基本概念

11.2 数据库恢复概述

11.3 故障的种类

11.4 恢复的实现技术

11.5 恢复策略

11.6 具有检查点的恢复技术

11.7 数据库镜像

11.8 MySQL中的日志

11.3 故障的种类



1. 事务故障
2. 系统故障
3. 介质故障



■ 什么是事务故障

某个事务在运行过程中
由于种种原因未运行至正常
终止点就夭折了

■ 事务故障的常见原因

- 输入数据有误
- 运算溢出
- 违反了某些完整性限制
- 并行事务发生死锁
-

■ 事务故障恢复

- 对于可预见的错误，由事务程序来处理(上例)
- 对于不可预见的错误，由DBMS强行回滚该事务。

例：银行转账事务的伪代码

```
start transaction;  
读取帐户甲的余额balance;  
balance = balance - amount;  
if (balance < 0) then  
    {提示‘余额不足，不能转帐’;  
    rollback; }  
else  
    写回balance到数据库;  
    {读帐户乙的余额balance1;  
    balance1= balance1+ amount  
    写回balance1  
    commit; }
```

■ 什么是系统故障

- 整个系统的正常运行突然被破坏
- 所有正在运行的事务都非正常终止
- 内存中数据库缓冲区的信息全部丢失
- 外部存储设备上的数据未受影响

■ 系统故障的常见原因

- 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）
- 突然断电




■ 什么是介质故障

介质故障是指存储数据库的设备(如硬盘)发生故障,使存储在其上的数据部分丢失或全部丢失。介质故障又称为硬件故障。

■ 介质故障的产生原因

- 磁盘损坏
- 磁头碰撞
- 操作系统的某种潜在错误
- 瞬时强磁场干扰



可能性小,
但破坏性大!

■ 介质故障的恢复

- (1) 装入数据库发生介质故障前某个时刻的数据副本
- (2) 重做自此时开始的所有成功事务,将这些事务已提交的结果重新记入数据库。

11.1 事务的基本概念

11.2 数据库恢复概述

11.3 故障的种类

11.4 恢复的实现技术

11.5 恢复策略

11.6 具有检查点的恢复技术

11.7 数据库镜像

11.8 MySQL中的日志

■ 恢复操作的基本原理：冗余

利用存储在系统别处的冗余数据来重建数据库中已被破坏或不正确的那部分数据。

■ 恢复技术中的关键技术：

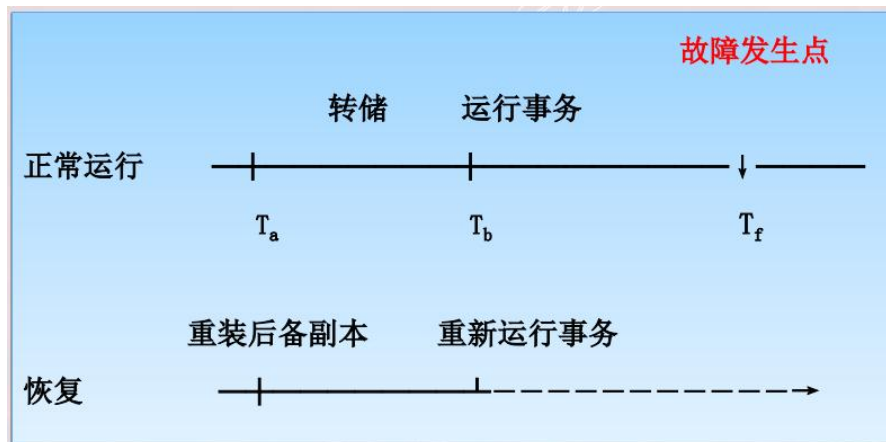
- 如何建立冗余数据
 - 数据转储 (backup)
 - 日志文件 (logging)
- 如何利用这些冗余数据实施数据库恢复

复杂：一个大型数据库产品，恢复子系统的代码要占全部代码的10%以上

■ 转储

指数据库管理员定期地将整个数据库复制到磁带、磁盘或其他存储介质上保存起来的过程。

备用的数据文本称为后备副本 (backup) 或后援副本。



转储状态：
静态转储与动态转储

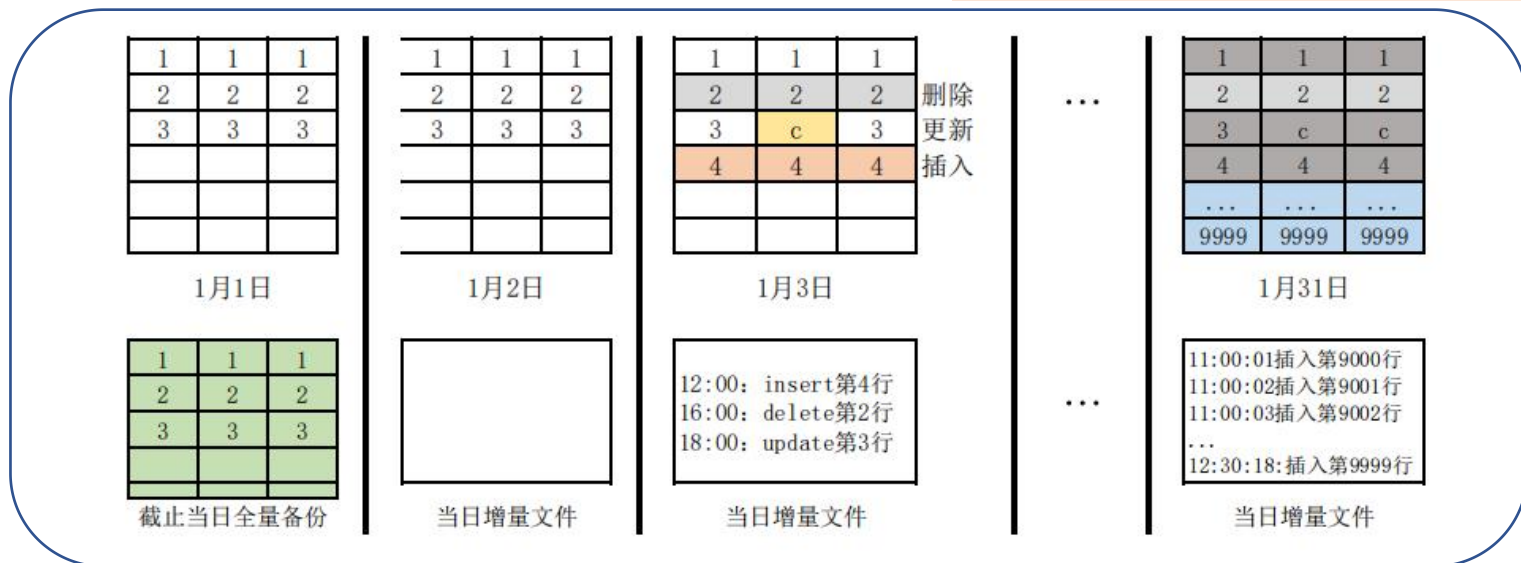
转储方式：
海量转储与增量转储

■ 静态转储与动态转储

| | 静态转储 | 动态转储 |
|----|---|---|
| 概念 | 系统中 无运行事务时 进行的转储操作 | 转储操作与用户事务并发 进行 |
| | 转储期间不允许对数据库的任何存取、修改活动 | 转储期间允许对数据库进行存取或修改 |
| 优点 | 实现简单 | 提高数据库的可用性 1. 不用等待正在运行的用户事务结束; 2. 不会影响新事务的运行 |
| 缺点 | 降低了数据库的可用性: 1. 转储必须等待正运行的用户事务结束 2. 新事务必须等转储结束 | 不能保证副本中的数据正确有效: 例如在转储过程中, 如果数据被修改, 则副本上的数据是过时的数据 |

■ 数据恢复到任意时刻?

增量通常是日志备份



| | | |
|------|------|------|
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | c | c |
| 4 | 4 | 4 |
| ... | ... | ... |
| 9999 | 9999 | 9999 |

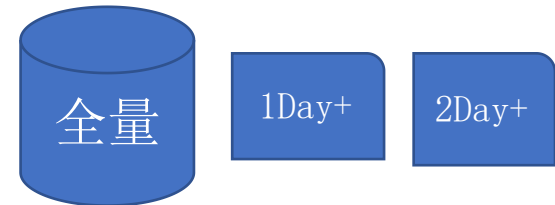
2月1日

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |
| | | |
| | | |

截止当日全量备份

- 数据恢复到1月1日?
- 数据恢复到1月2日?
- 数据恢复到1月3日13:00:00?
- 数据恢复到1月23日12:00:00?

■ 海量转储与增量转储



➤ 海量转储：每次转储全部数据库

➤ 增量转储：只转储上次转储后更新过的数据

■ 海量转储与增量转储的比较

➤ 从恢复角度看：

使用海量转储得到的后备副本进行恢复往往更方便

➤ 从实用角度看：

如果数据库很大，事务处理又十分频繁，则增量转储方式更实用更有效。

转储方法分类

| | | 转储状态 | |
|------|------|--------|--------|
| | | 动态转储 | 静态转储 |
| 转储方式 | 海量转储 | 动态海量转储 | 静态海量转储 |
| | 增量转储 | 动态增量转储 | 静态增量转储 |

转储策略

- 应定期进行数据转储，制作后备副本。
- 但转储又是十分耗费时间和资源的，不能频繁进行。
- DBA应该根据数据库使用情况确定适当的转储周期和转储方法。

例：

- (1) 每天晚上进行动态增量转储
- (2) 每周进行一次动态海量转储
- (3) 每月进行一次静态海量转储

数据同步工具：
CDC工具



数据库需要进行恢复时，逻辑备份快还是物理备份快呢？

- ☐ A 逻辑备份快
- ☒ B 物理备份快
- ☐ C 一样快
- ☐ D 不确定

| 备份管理 | | | | | | | | | | |
|--------------------------|---------------------------------------|-----------------------------------|--------------|------|----------|----------------------------|------|---------|----|--------------|
| 数据库同区域备份 | | | | | 数据库跨区域备份 | | | | | |
| | | | | | 全部引擎 | | | | | |
| | | | | | 备份名称 | | | | | |
| | | | | | 请输入关键字 | | | | | |
| <input type="checkbox"/> | 备份名称/ID | 实例名称/ID | 数据库引擎版本 | 备份类型 | 备份方式 | 备份开始/结束时间 | 状态 | 大小 | 描述 | 操作 |
| <input type="checkbox"/> | mysql-mysqld80-2022-3049d4d925f45... | mysql80-2022-d8f5da241ab40f0bd... | MySQL 8.0.25 | 自动 | 物理备份 | 2022/11/02 00:40:23 - 2... | 备份完成 | 2.11 GB | -- | 恢复 下载 复制 |
| <input type="checkbox"/> | mysql-mysqld80-2022-a3c0f71c17d74t... | mysql80-2022-d8f5da241ab40f0bd... | MySQL 8.0.25 | 自动 | 物理备份 | 2022/11/01 00:40:24 - 2... | 备份完成 | 2.11 GB | -- | 恢复 下载 复制 |

提交

■ 日志文件

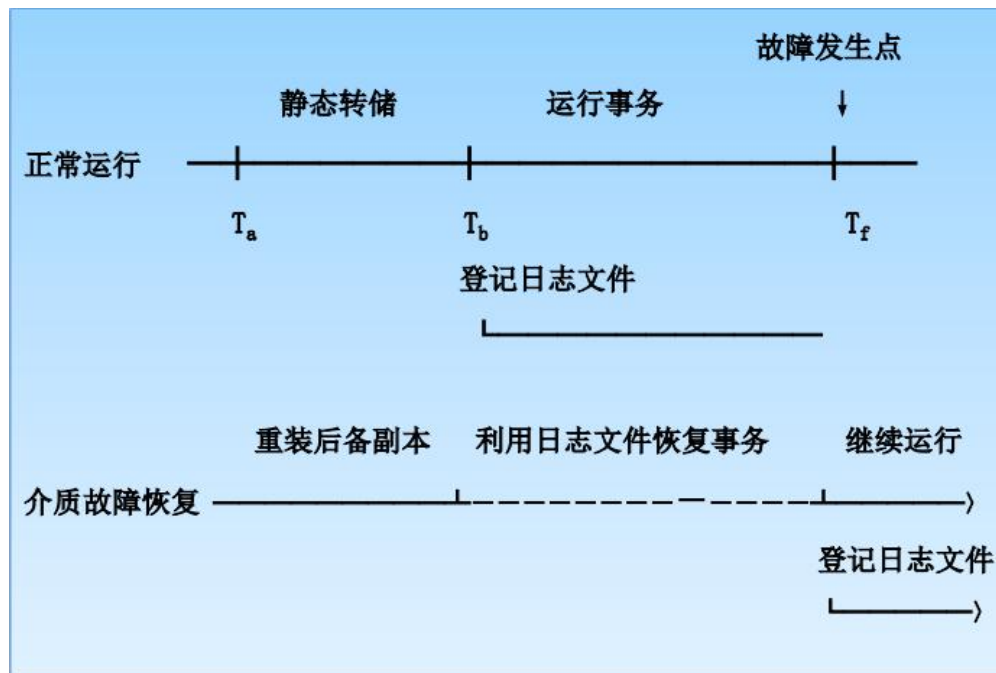
用来记录事务对数据库的更新操作的文件，只能**顺序**追加，不同事务的日志记录交错存储，按照时间顺序记录。

■ 日志文件格式

- 以记录为单位：事务开始、事务结束、各个事务的更新操作
 - 事务标识（标明是哪个事务）
 - 操作类型（插入、删除或修改）
 - 操作对象（记录ID、 Block NO.）
 - 更新前数据的旧值（对插入操作而言，此项为空值）
 - 更新后数据的新值（对删除操作而言，此项为空值）
- 以数据块为单位：
事务标识、被更新的数据块

■ 日志文件的作用

- **故障恢复**必须使用日志文件。
- 动态转储方式：利用备份副本与日志结合起来才能恢复
- 静态转储方式：利用日志恢复转储结束到故障发生之间的数据



■ 日志相关的几个重要概念

- 事务提交日志

当一个事务的提交日志记录 $\langle T_i, \text{commit} \rangle$ 输出到稳定存储器（磁盘）后，我们说该事务被提交了。

（大部分基于日志的恢复技术，并不要求在事务提交时必须将该事务所做的修改的数据从主存缓冲中输出到磁盘中 => 不同缓冲区策略）

- 检查点日志

$\langle \text{checkpoint} (T_2, T_3) \rangle$ ：在日志中写入该记录时，表明在 T_2, T_3 之前的事务（如 T_1 ）已经完全修改完成。

- 事务回滚日志

$\langle T_i, \text{abort} \rangle$ ：在日志中写入该记录时，表明事务 T_i 的回滚（rollback）已经结束。

■ 事务提交的不同缓冲区策略

➤ 从未提交事务的角度来看

- STEAL: 允许未提交事务的写落盘
- NO-STEAL: 不允许未提交事务的写落盘

➤ 从已提交事务的角度来看

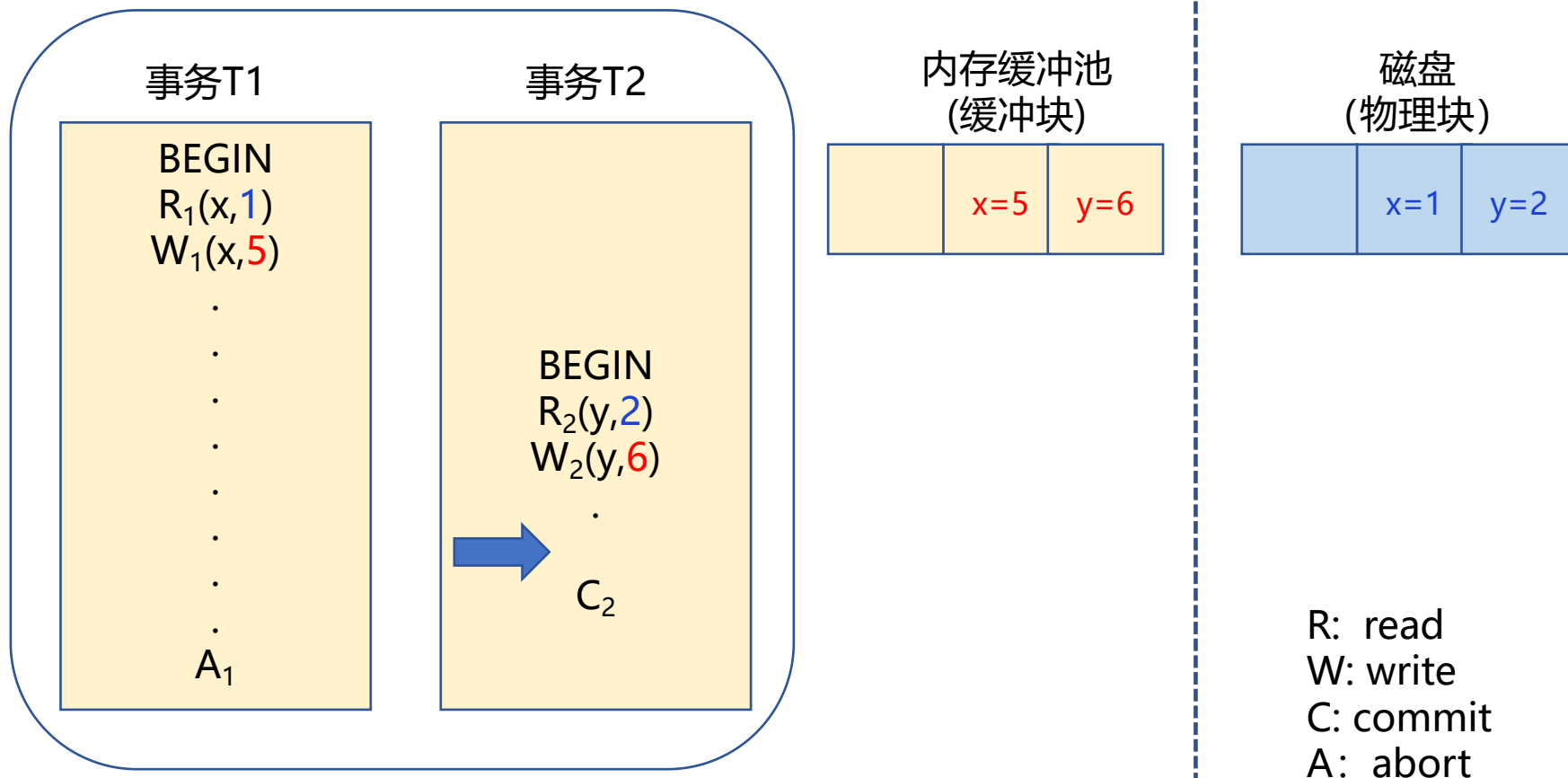
- FORCE: 事务一旦提交, 强制同步该事务的写落盘
- NO-FORCE: 事务一旦提交, 不强制同步该事务写落盘

NO-STEAL + FORCE

11.4.2 登记日志文件



NO-STEAL + FORCE



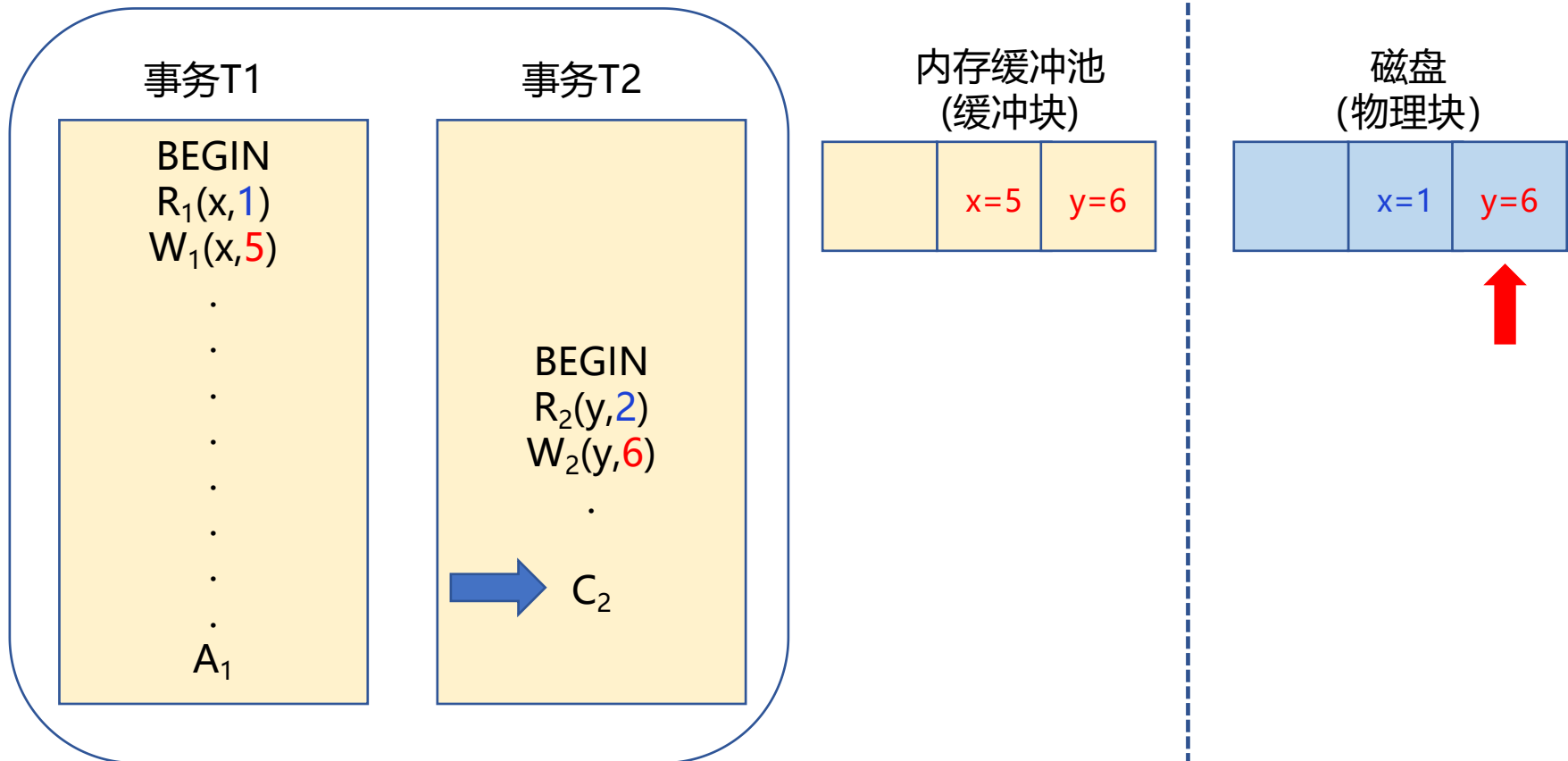
FORCE: 事务一旦提交, 强制同步该事务的写落盘

NO-STEAL: 不允许未提交事务的写落盘

11.4.2 登记日志文件



NO-STEAL + FORCE



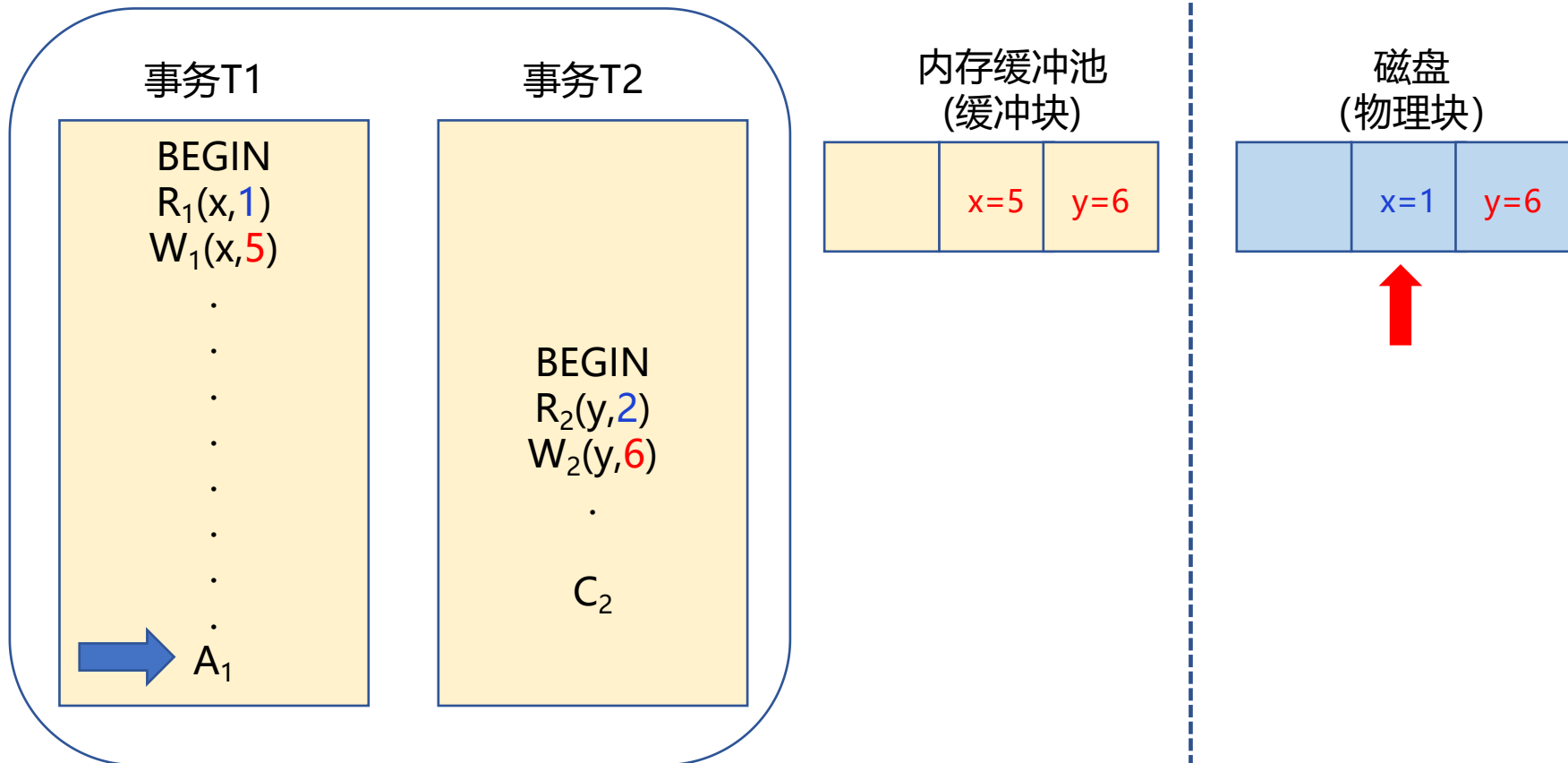
FORCE: 事务一旦提交, 强制同步该事务的写落盘

NO-STEAL: 不允许未提交事务的写落盘

11.4.2 登记日志文件



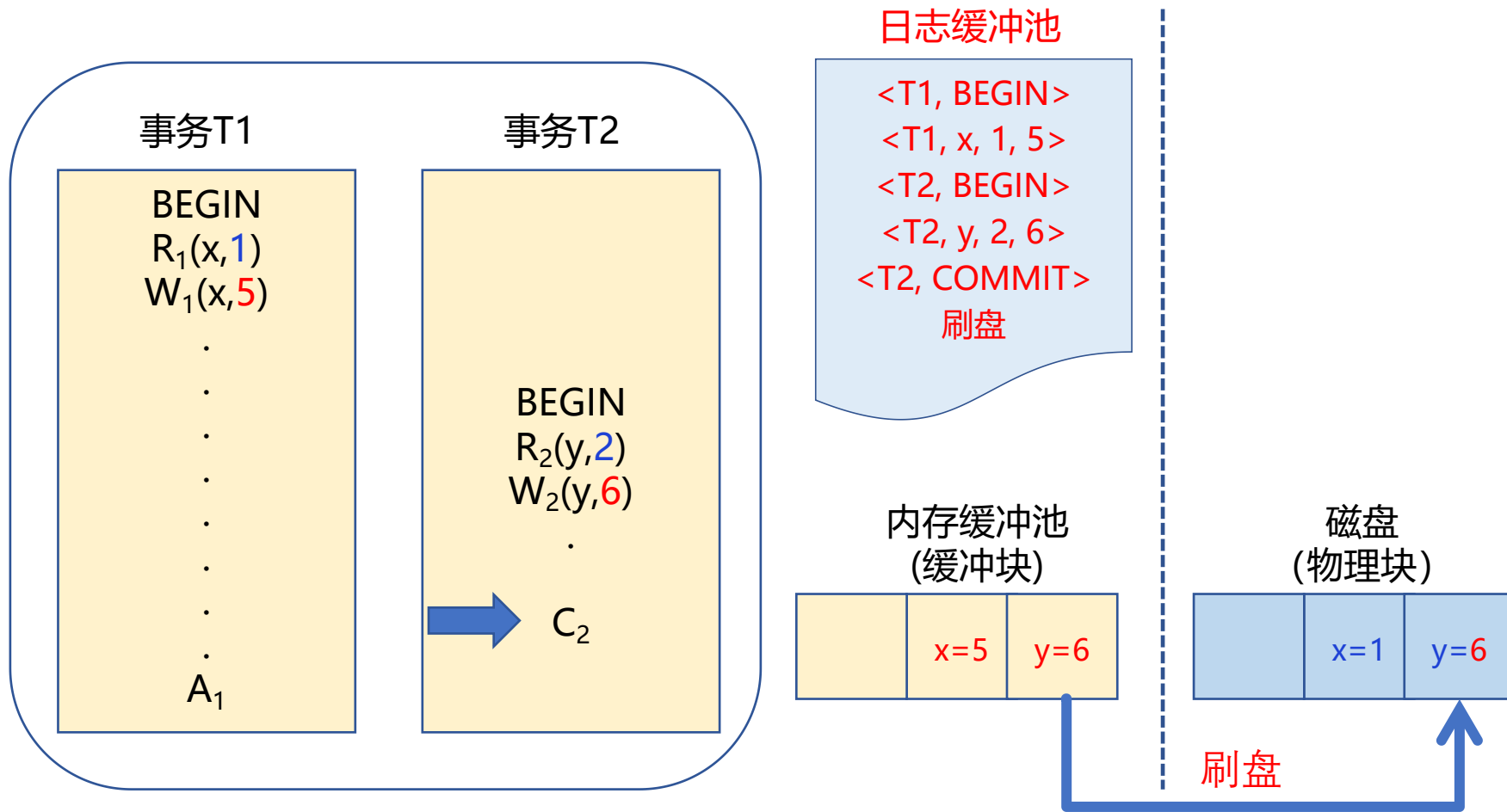
NO-STEAL + FORCE



FORCE: 事务一旦提交, 强制同步该事务的写落盘

NO-STEAL: 不允许未提交事务的写落盘

日志文件示例 (NO-STEAL + FORCE)



■ 登记日志文件需要遵循的原则

- 登记的次序严格按并发事务执行的时间次序
- 必须先写日志文件，后写数据库

WAL日志: write ahead log

■ 为什么先写日志后写数据库？

- 写日志与写数据之间发生故障怎么办？
- 先写数据库会出现什么情况？先写日志呢？

Insert into sc values (1, 1, 100);

插入: T(grade, null, 100)

更新: T(grade, 80, 100) (old, new)

删除: T(grade, 100, null)

■ 几乎所有的商用数据库采用：NO-FORCE+STEAL 策略

NO-FORCE：事务提交，不强制同步写落盘

STEAL：允许未提交事务的写落盘

正常执行时的性能

NO-STEAL STEAL

NO-FORCE

redo日志

redo/undo日志
最快

FORCE

最慢

undo日志

故障恢复时的性能

NO-STEAL STEAL

NO-FORCE

redo日志

redo/undo日志
最慢

FORCE

最快

undo日志

**redo
undo**

**无需redo
无需undo**

11.1 事务的基本概念

11.2 数据库恢复概述

11.3 故障的种类

11.4 恢复的实现技术

11.5 恢复策略

11.6 具有检查点的恢复技术

11.7 数据库镜像

11.8 MySQL中的日志

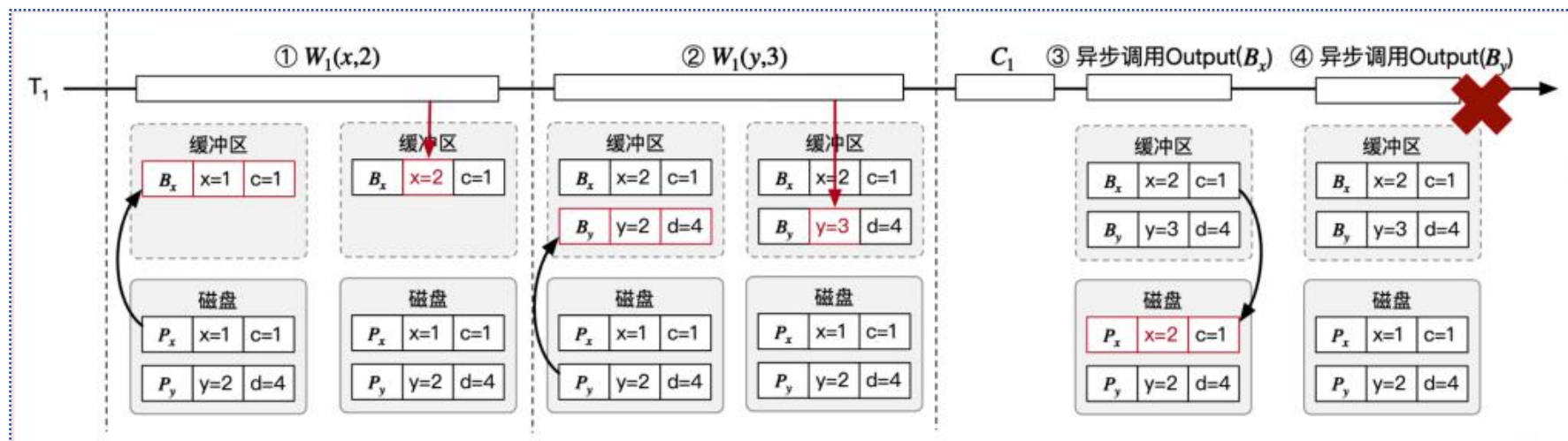
- 事务故障的恢复
- 系统故障的恢复
- 介质故障的恢复



11.5.1 事务故障的恢复



■ 事务故障可能造成的数据不一致



引自 中国人民大学 数据库课程课件

■ 事务故障恢复方法

由恢复子系统利用日志文件撤消（UNDO）此事务已对数据库进行的修改

注意：事务故障的恢复由系统自动完成，
对用户是透明的，不需要用户干预！

■ 恢复步骤

- 1) 反向扫描文件日志，查找该事务的更新操作；
- 2) 对该事务的更新操作执行逆操作，即将日志记录中“更新前的值”写入数据库。
 - 插入：“更新前的值”为空，则进行删除操作
 - 删除：“更新后的值”为空，则利用删除前的值进行插入操作
 - 更新：用修改前值代替修改后值
- 3) 继续反向扫描日志文件，查找该事务的其他更新操作，直到读到此事务的开始标记。

■ 事务故障恢复方法 例子

具体实现的一种例子（参考英文版教材 数据库系统概念）

对单纯事务回滚的恢复：

从后往前扫描日志，对于日志 $\langle T_i, X_j, V1, V2 \rangle$ （ $V1$:旧值； $V2$:新值）：

1. 将一个旧值 $V1$ 写入数据项 X_j
2. 往日志里面输出一条特殊日志： $\langle T_i, X_j, V1 \rangle$ （有时也称补偿日志）
3. 继续往回扫描日志，直到发现 $\langle T_i, \text{start} \rangle$ 的日志记录，停止扫描，并往日志中写一条 $\langle T_i, \text{abort} \rangle$ 的记录。

插入： $(T1, \text{grade}, \text{null}, 100)$

更新： $(T2, \text{grade}, 100, 200)$

删除： $(T1, \text{grade}, 200, \text{null})$

$\langle T2, a1, 800 \rangle$: //补偿日志

$\langle T2, b1, 600 \rangle$: //补偿日志

$\langle T2, \text{abort} \rangle$: 回滚结束

表明T2事务回滚了a1为800, b1为600

■ 系统故障可能造成的数据不一致

- 系统故障发生前，数据库的**一致性状态**是什么？

系统中**未提交事务**的所有操作都未发生，且**已提交事务**的写操作都已经应用在缓冲区和磁盘上时，数据库的一致性状态。

- 故障发生后，会有**哪些一致性状态**被破坏？
 - 系统故障发生后，系统需要重启，导致**缓存区中的所有数据丢失**，所有运行事务都被中止。
 - **未提交事务的写**：故障发生时，未提交事务的执行被中断，而这些**未提交事务中可能有一些事务的写已经写入到磁盘**。
 - **已提交事务的写**：**已提交事务写入的数据可能部分或全部留在缓冲区中**，系统故障导致**缓冲区中的数据丢失**，而这些数据尚未来得及写到磁盘。

■ 系统故障造成数据库不一致状态的原因

- **未完成事务**：数据可能已经部分写入，但后来的故障导致这个事务需要终止。
- **已提交事务**：日志已经写入磁盘，但对数据库的更新可能还留在缓冲区没来得及写入磁盘

■ 系统故障恢复方法

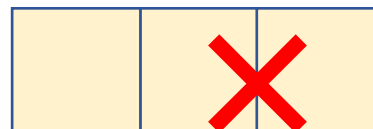
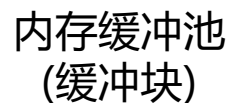
- 故障发生时**未完成**的事务：Undo
- 故障发生时**已完成**的事务：Redo

注意：系统故障的恢复由系统在重新启动时自动完成，不需要用户干预！

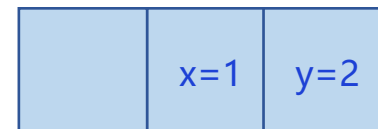
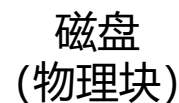
■ 系统故障的恢复步骤

- 1) 正向扫描日志文件（即从头扫描日志文件）；
 - 构造重做队列(RED0-List)：故障发生前已经提交事务
该类事务：BEGIN TRANSACTION + COMMIT
 - 构造撤销队列(UNDO-List)：故障发生时尚未完成的事务
该类事务：BEGIN TRANSACTION
- 2) 对撤销(UNDO)队列事务进行撤销(UNDO)处理
 - 反向扫描日志文件，将“更新前的值”写入数据库
- 3) 对重做(RED0)队列事务进行重做(RED0)处理
 - 正向扫描日志文件，将“更新后的值”写入数据库

UNDO: T1

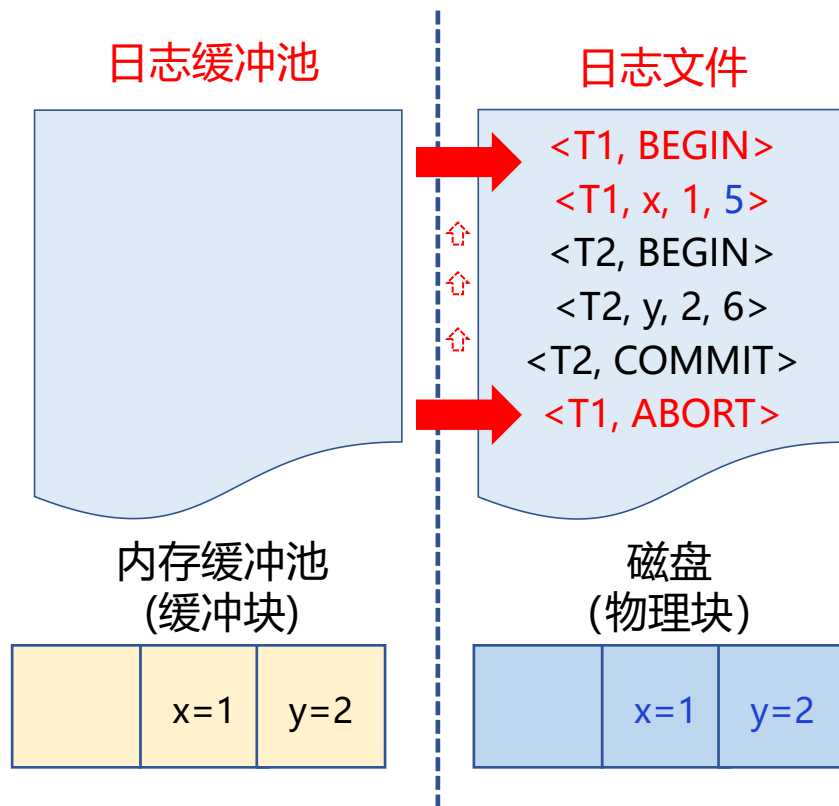
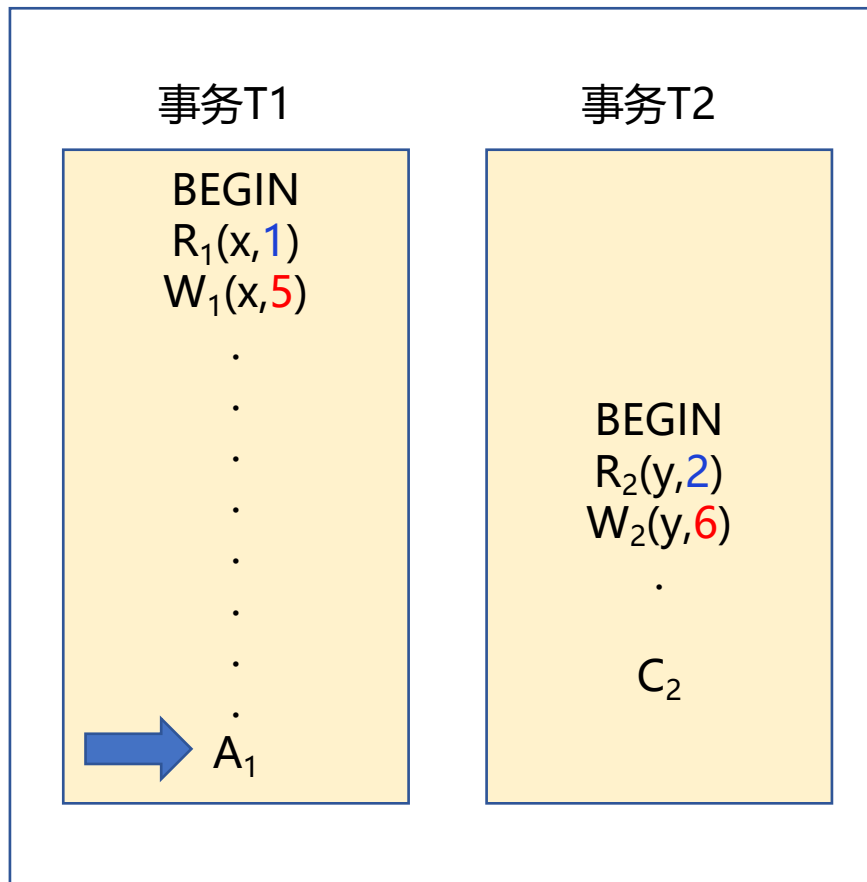


日志文件



1. 正向扫描日志文件构造REDO/UNDO列表
2. 反向扫描日志，执行UNDO
3. 正向扫描日志，执行REDO

UNDO: T1



1. 正向扫描日志文件构造REDO/UNDO列表
2. 反向扫描日志, 执行UNDO
3. 正向扫描日志, 执行REDO

■ 介质故障是最严重的一种故障

- 重装副本
- 重做已完成的事务

■ 恢复步骤

- 1) 恢复至离发生故障最近的一个副本
- 2) 装入相应的日志文件副本（转储结束时刻）
 - 首先扫描日志文件，构造重做队列(REDO-List)
 - 正向扫描日志文件，重做事务，即将“更新后的值”写入数据库



需要DBA将相应文件进行装入，然后执行对应的命令

■ 数据库故障类型与恢复技术

| 故障类型 | 特点和问题 | 恢复技术 |
|------|---------------------------------------|--|
| 事务故障 | 事务故障时内存与磁盘数据均可正常访问，仅事务内部需要回滚 | 利用日志进行undo (此时可能尚未写入磁盘的缓存中的日志信息) |
| 系统故障 | 系统故障时内存数据丢失，磁盘数据虽未丢失但无法确认缓冲区数据是否已写入磁盘 | 日志：最后一个检查点前的不用恢复（检查点机制确保之前数据已写入磁盘），之后的故障时已提交事务redo，未提交事务undo |
| 介质故障 | 故障时磁盘数据破坏（不用考虑系统故障的复杂性） | 1. 重装故障前最后一个转储副本 2. 日志：从头到尾做redo即可 |

11.1 事务的基本概念

11.2 数据库恢复概述

11.3 故障的种类

11.4 恢复的实现技术

11.5 恢复策略

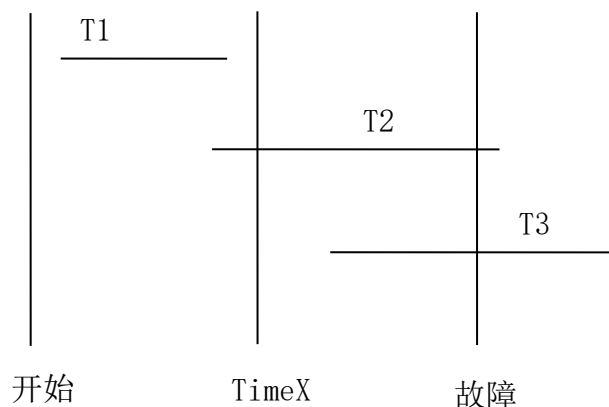
11.6 具有检查点的恢复技术

11.7 数据库镜像

11.8 MySQL中的日志

■ 系统故障恢复时存在的问题

- (1) 搜索整个日志将耗费大量的时间
- (2) REDO处理：重新执行，浪费了大量时间 => 通过某种方式记录？



T1在故障发生之前已经commit, 对数据库的修改已经写入数据库, 恢复时仍然需要Redo

■ 解决方案：具有检查点（checkpoint）的恢复技术

- 1) 在日志文件中增加检查点记录（checkpoint）
- 2) 增加重新(Redo)开始文件
- 3) 恢复子系统在写日志文件期间动态地维护日志

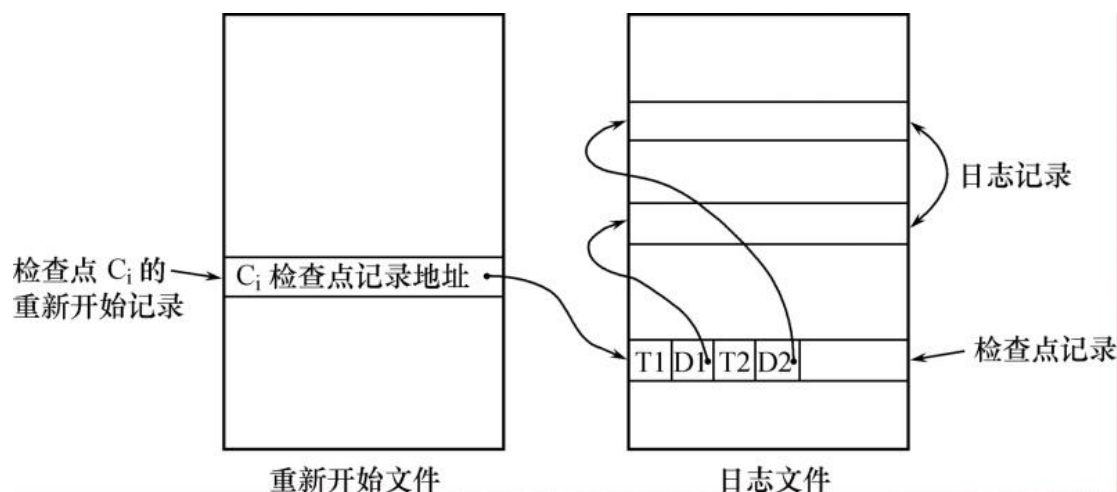


■ 日志文件中的检查点记录

- (1) 建立检查点时刻所有正在执行的事务清单 (T_1, T_2, \dots)
- (2) 这些事务最近一个日志记录的地址 (D_1, D_2, \dots)

■ 重新开始文件

- 记录各个检查点记录在日志文件中的地址

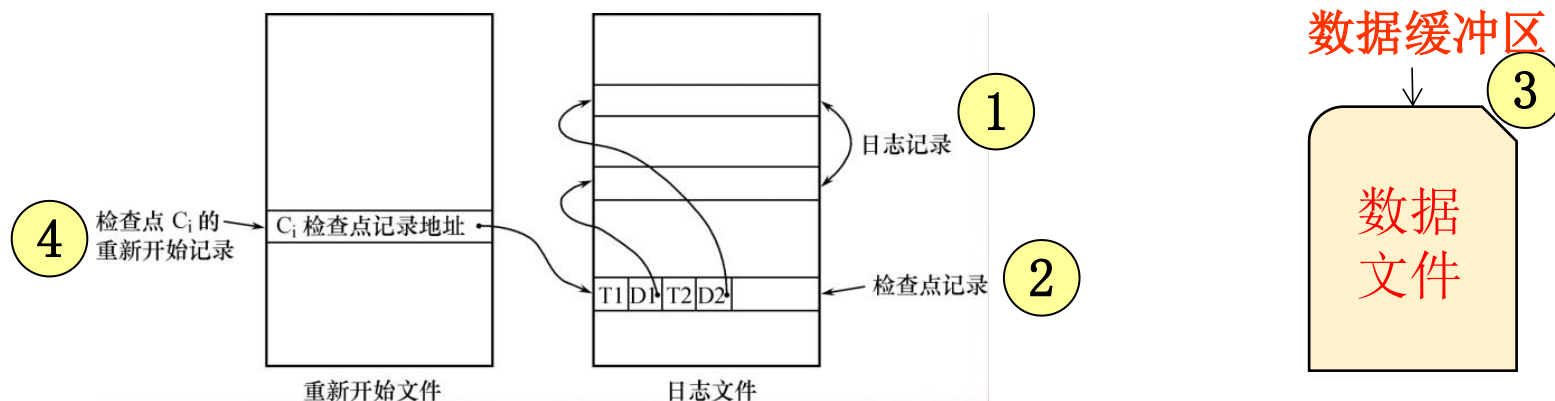


11.6 具有检查点的恢复技术



- 动态维护日志文件的具体步骤(周期性建立检查点):
 - 1) 将当前日志缓冲区中的所有日志记录写入磁盘的日志文件上
 - 2) 在日志文件中写入一个检查点记录
 - 3) 将当前数据缓冲区的所有数据记录写入磁盘的数据库中
 - 4) 把检查点记录在日志文件中的地址写入一个重新开始文件

检查点记录产生时，可以确保日志和数据都已经落盘！

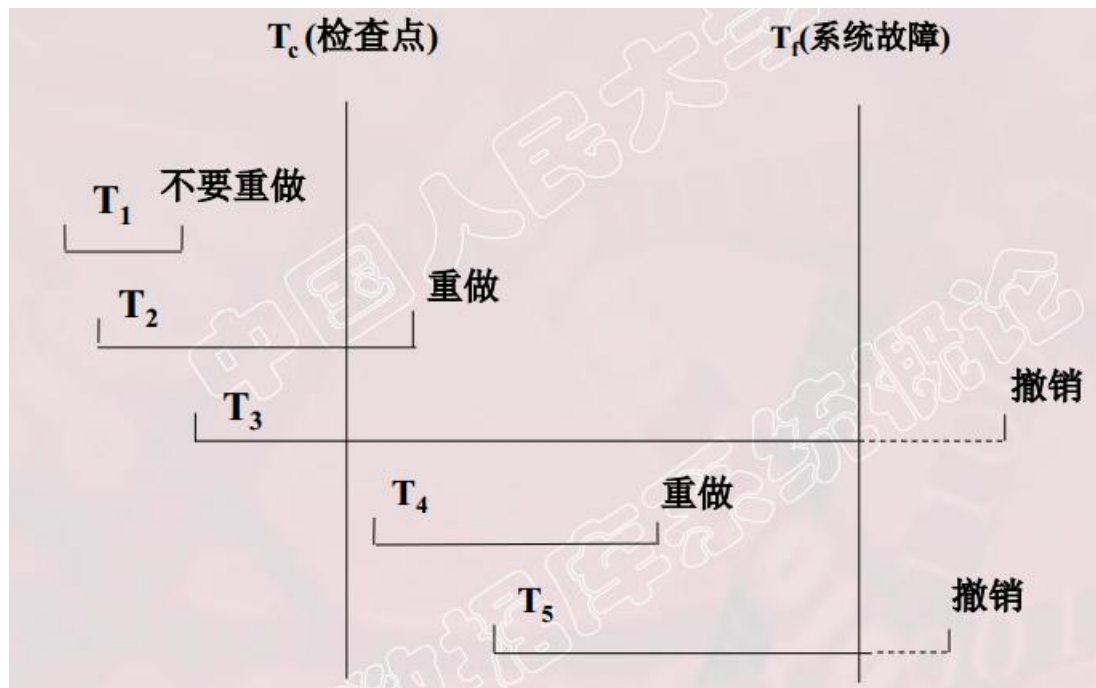


周期性：可以是定时或者按照一定的规则(如到达日志一半)建立检查点

11.6 具有检查点的恢复技术



- 恢复子系统根据事务的不同状态采用不同的恢策略



故障发生时刻:

已提交事务: 重做Redo

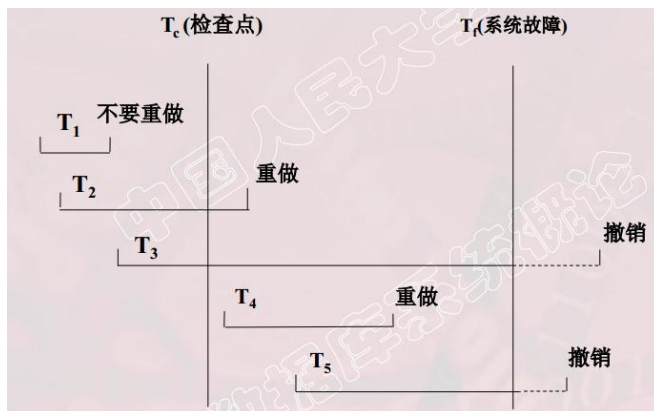
未提交事务: 撤销Undo

11.6 具有检查点的恢复技术



■ 恢复子系统利用检查点恢复的步骤:

- 1) 从重新开始文件中找到**最后一个检查点记录地址**，在日志文件中找到**最后一个检查点记录**。
- 2) 得到检查点记录建立时，所有正在执行的事务清单Active-list，将其加入UndoList (T2, T3)
- 3) 从检查点开始正向扫描日志文件：
 - 有新开始的事务T， T → UndoList (T4, T5)
 - 有提交的事务，将该事务从UndoList → RedoList (T2, T4)
- 4) 分别针对UndoList (T3, T5) / RedoList (T2, T4) 执行Undo和Redo



11.6 具有检查点的恢复技术



西北工业大学
NORTHWESTERN POLYTECHNICAL UNIVERSITY

■ 恢复子系统利用检查点恢复的步骤：动画演示



11.6 具有检查点的恢复技术



- 恢复子系统利用检查点的优化恢复策略：优化的体现？
(参考英文版数据库系统概念教材)

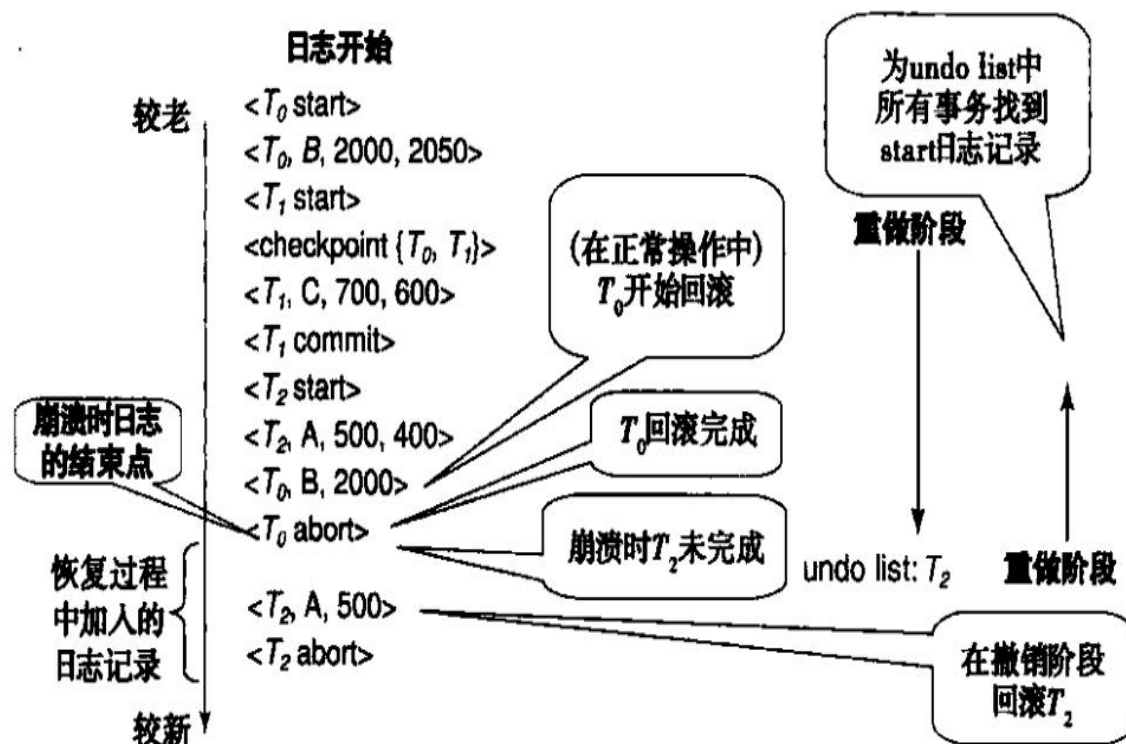


图 16-5 记录在日志中的动作和恢复中的动作的例子

11.6 具有检查点的恢复技术

恢复子系统利用检查点的优化恢复策略 (参考英文版数据库系统概念教材)

Step1:重做(Redo)阶段

1) 找到最后一个检查点记录: $\langle \text{checkpoint } (T_0, T_1) \rangle$

构造初始list: 初始值=检查点时刻活动的事务: $\{T_0, T_1\}$

2) 从检查点开始逆向扫描日志, 找到 $\langle T_0, \text{start} \rangle$ 和 $\langle T_1, \text{start} \rangle$ 为止

3) 从 $\langle T_0, \text{start} \rangle$ 开始正向扫描日志:

- 读到 $\langle T_0, \text{start} \rangle$: 把T0加入undo-list
 - 读到 $\langle T_0, B, 2000, 2050 \rangle$: 执行redo (写2050写给数据项B)
 - 读到 $\langle T_1, \text{start} \rangle$: 把T1加入undo-list
 - 读到 $\langle \text{Checkpoint } \{T_0, T_1\} \rangle$: 什么都不做
 - 读到 $\langle T_1, C, 700, 600 \rangle$: 执行redo (写600写给数据项C)
 - 读到 $\langle T_1, \text{commit} \rangle$: 把T1从undo-list 去除
 - 读到 $\langle T_2, \text{start} \rangle$: 把T2加入undo-list
 - 读到 $\langle T_2, A, 500, 400 \rangle$: 执行redo (写400给数据项A)
 - 读到 $\langle T_0, B, 2000 \rangle$: 执行redo (写2000给数据项B) (本特殊日志说明T0执行回滚了)
 - 读到 $\langle T_0, \text{abort} \rangle$: 表明T0回滚完成, 把T0从undo-list 去除
- (此时, undo-list只剩下T2)

正向扫描日志结束。

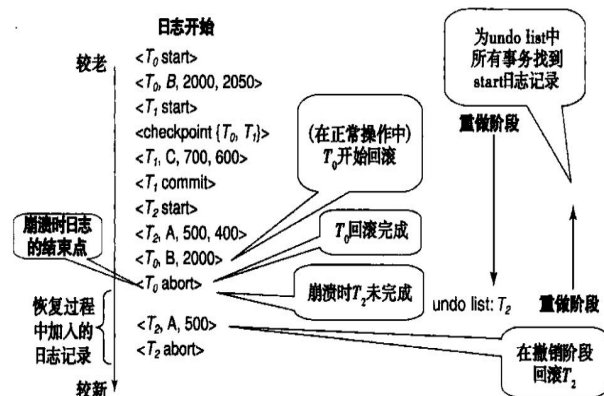


图 16-5 记录在日志中的动作和恢复中的动作的例子

■ 恢复子系统利用检查点的优化恢复策略 (参考数据库系统概念教材)

与优化前的区别？

1. 一遍扫描+直接执行redo操作
2. 在扫描日志过程中构建undo
3. 针对undolist执行undo操作

Step2: 撤销(Undo)阶段

- 1) 针对undo-list中的事务进行逐一Undo (本例: 仅T2)
- 2) 针对T2, 从日志尾部开始反向扫描执行回滚
 - <1> 发现 <T2, A, 500, 400>: 执行Undo(写500给数据项A)
 - <2> 写补偿日志 <T2, A, 500>: 记录该回滚动作
 - <3> 继续发现扫描, 直到发现<T2, start>
 - 写日志 <T2, abort>
 - 将T2从undo-list 去除

循环针对其他undo-list中的事务做同样的处理

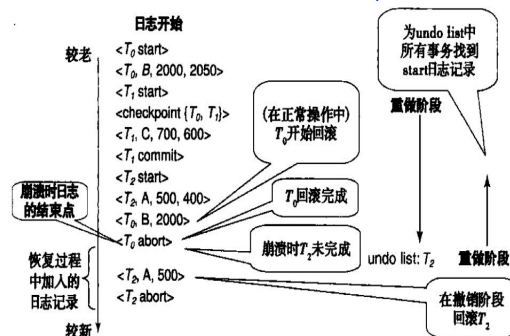


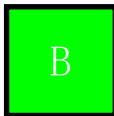
图 16-5 记录在日志中的动作和恢复中的动作的例子

对以下日志记录，若故障在9之后，哪些事务会被undo？

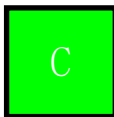
| 序号 | 日志 |
|----|---------------|
| 1 | T1: 开始 |
| 2 | T1: 写 A, A=10 |
| 3 | T2: 开始 |
| 4 | T2: 写 B, B=9 |
| 5 | T1: 写 C, C=11 |
| 6 | T1: 提交 |
| 7 | T2: 写 C, C=13 |
| 8 | T3: 开始 |
| 9 | T3: 写 A, A=8 |
| 10 | T2: <u>回滚</u> |
| 11 | T3: 写 B, B=7 |
| 12 | T4: 开始 |
| 13 | T3: 提交 |
| 14 | T4: 写 C, C=12 |



T1



T2



T3



T4

若按照英文版教材的恢复中的优化算法（第一次扫描时直接做redo, 将已完成的从undolist中删除），哪些事务会被undo？哪些事务会被redo？

提交

11.1 事务的基本概念

11.2 数据库恢复概述

11.3 故障的种类

11.4 恢复的实现技术

11.5 恢复策略

11.6 具有检查点的恢复技术

11.7 数据库镜像

11.8 MySQL中的日志

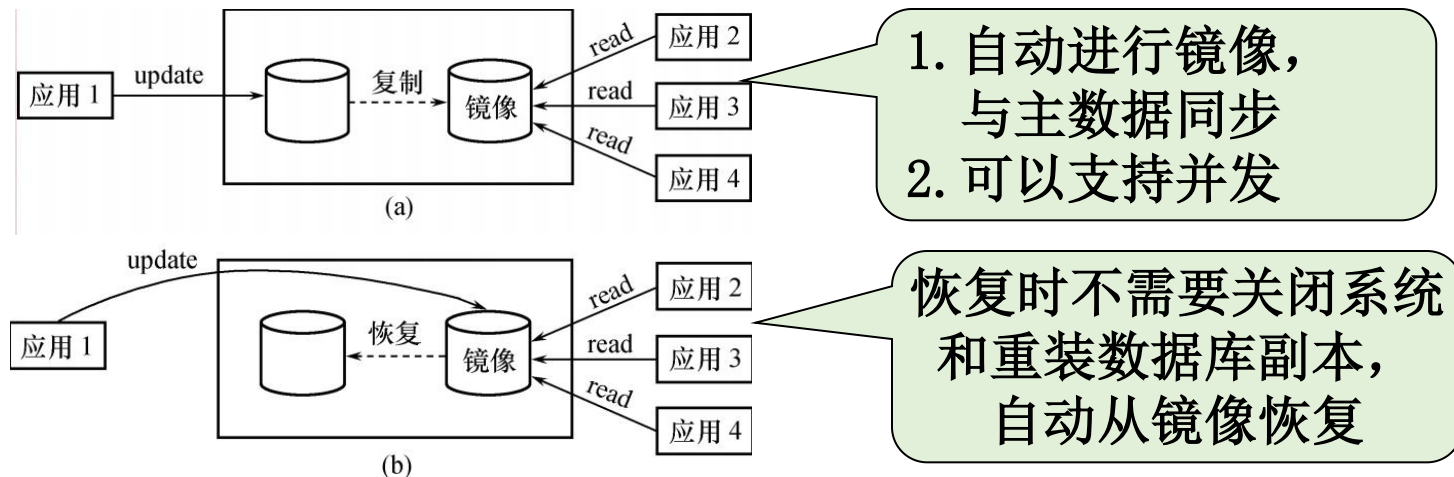
■ 介质故障严重影响数据库的可用性

- 介质故障恢复比较费时
- 为预防介质故障，DBA必须周期性地转储数据库

■ 数据库镜像

由DBMS自动把整个数据库或其中的关键数据复制到另一个磁盘上，DBMS自动保证镜像数据与主数据的一致性

■ 数据库镜像的用途



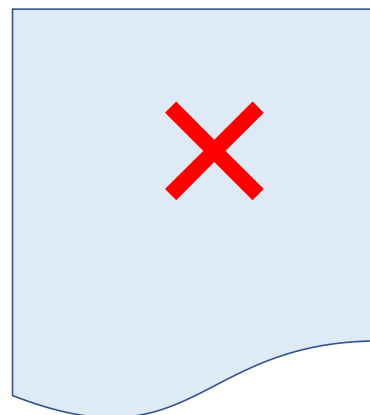


■ 数据库故障类型与恢复技术

| 故障类型 | 特点和问题 | 恢复技术 |
|------|---------------------------------------|--|
| 事务故障 | 事务故障时内存与磁盘数据均可正常访问，仅事务内部需要回滚 | 利用日志进行undo (此时可能尚未写入磁盘的缓存中的日志信息) |
| 系统故障 | 系统故障时内存数据丢失，磁盘数据虽未丢失但无法确认缓冲区数据是否已写入磁盘 | <ol style="list-style-type: none">1. 重装故障前最后一个转储副本2. 日志：最后一个检查点前的不用恢复（检查点机制确保之前数据已写入磁盘），之后的故障时已提交事务redo，未提交事务undo |
| 介质故障 | 故障时磁盘数据破坏（不用考虑系统故障的复杂性） | <ol style="list-style-type: none">1. 重装故障前最后一个转储副本2. 日志：从头到尾做redo即可 |

WAL: 必须先写日志文件, 后写数据库!!! 为什么?

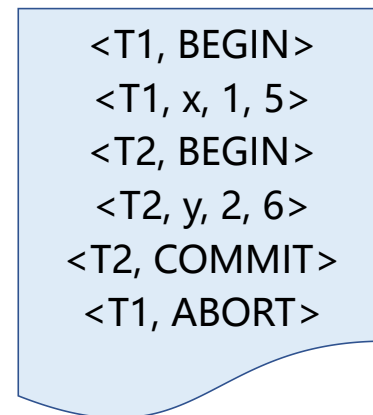
UNDO: T1



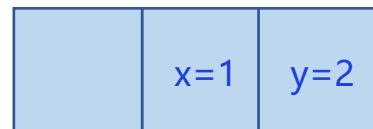
内存缓冲池 (缓冲块)



日志文件



磁盘
(物理块)



1. 正向扫描日志文件构造REDO/UNDO列表
2. 反向扫描日志，执行UNDO
3. 正向扫描日志，执行REDO

- 11.1 事务的基本概念
- 11.2 数据库恢复概述
- 11.3 故障的种类
- 11.4 恢复的实现技术
- 11.5 恢复策略
- 11.6 具有检查点的恢复技术
- 11.7 数据库镜像
- 11.8 MySQL中的日志**

11.8 MySQL中的日志文件



MySQL中日志文件类型：my.ini 或者 my.cnf

| 日志类型 | 日志标识 | 默认 | 说明 |
|--------|------------------|-----|---|
| 一般查询日志 | log | 不开启 | 记录所有的查询，占空间影响性能，默认不开 (general-log) |
| 错误日志 | log-err | 开启 | 记录mysql服务的错误 |
| 慢查询日志 | log-slow-queries | 开启 | 记录执行时间超过long_query_time设定时间阈值(秒)的SQL语句，mysqldumpslow。 |
| 二进制日志 | log-bin | 开启 | Mysql的server层变更日志，主要用于记录修改数据或有可能引起数据改变的mysql语句，可用于数据复制或者恢复。滚动文件，由.index文件管理。 |
| 中继日志 | relay log | 不开启 | 主从复制时使用的日志 |
| 事务日志 | innodb_log | 开启 | InnoDB特有的事务日志redo和undo日志，帮助提高事务的效率。存储引擎在修改表的数据时只需要修改其内存拷贝，再把修改行为记录到持久在硬盘上的事务日志中，而不用每次都修改的数据本身持久到磁盘。 |

11.8 MySQL中的日志文件



MySQL中日志文件配置: my.ini 或者 my.cnf

FLUSH LOGS;
mysqladmin flush-logs

```
# General and Slow logging.
log-output=FILE

general-log=0

general_log_file="THINKPAD_LINING.log"

slow-query-log=1

slow_query_log_file="THINKPAD_LINING-slow.log"

long_query_time=10

# Error Logging.
log-error="THINKPAD_LINING.err"

# ***** Group Replication Related *****
# Specifies the base name to use for binary log files. With binary logging
# enabled, the server logs all statements that change data to the binary
# log, which is used for backup and replication.
log-bin="THINKPAD_LINING-bin"
```

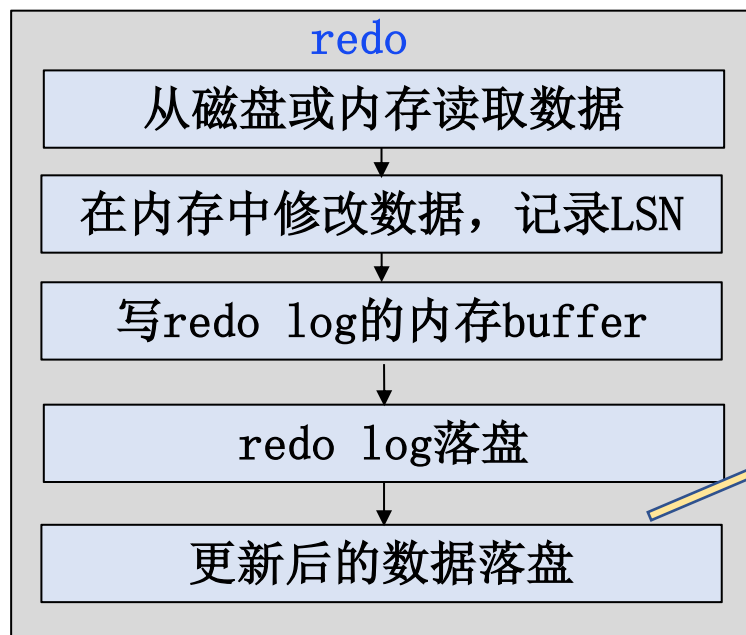
修改任何配置后，需要重新启动mysql服务

11.8 MySQL中的事务日志文件



■ 事务日志: redo log, undo log, binlog

SQL: `update t set name= 'b' where id = '3'`

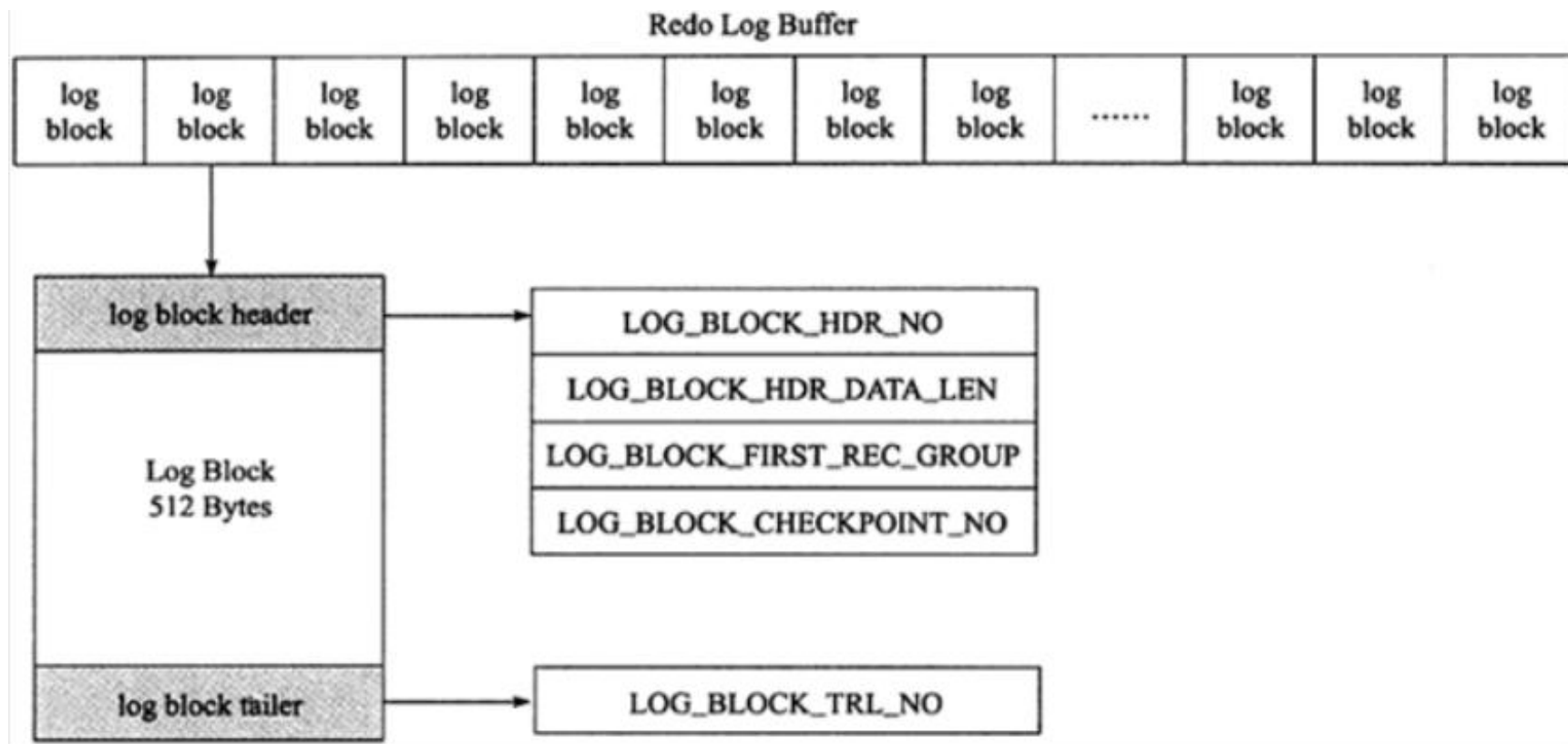


redo log优点: 顺序IO, 写入速度很快。并且redo log记载的是物理变化 (xxxx页做了xxx修改), 文件的体积很小, 恢复速度很快。

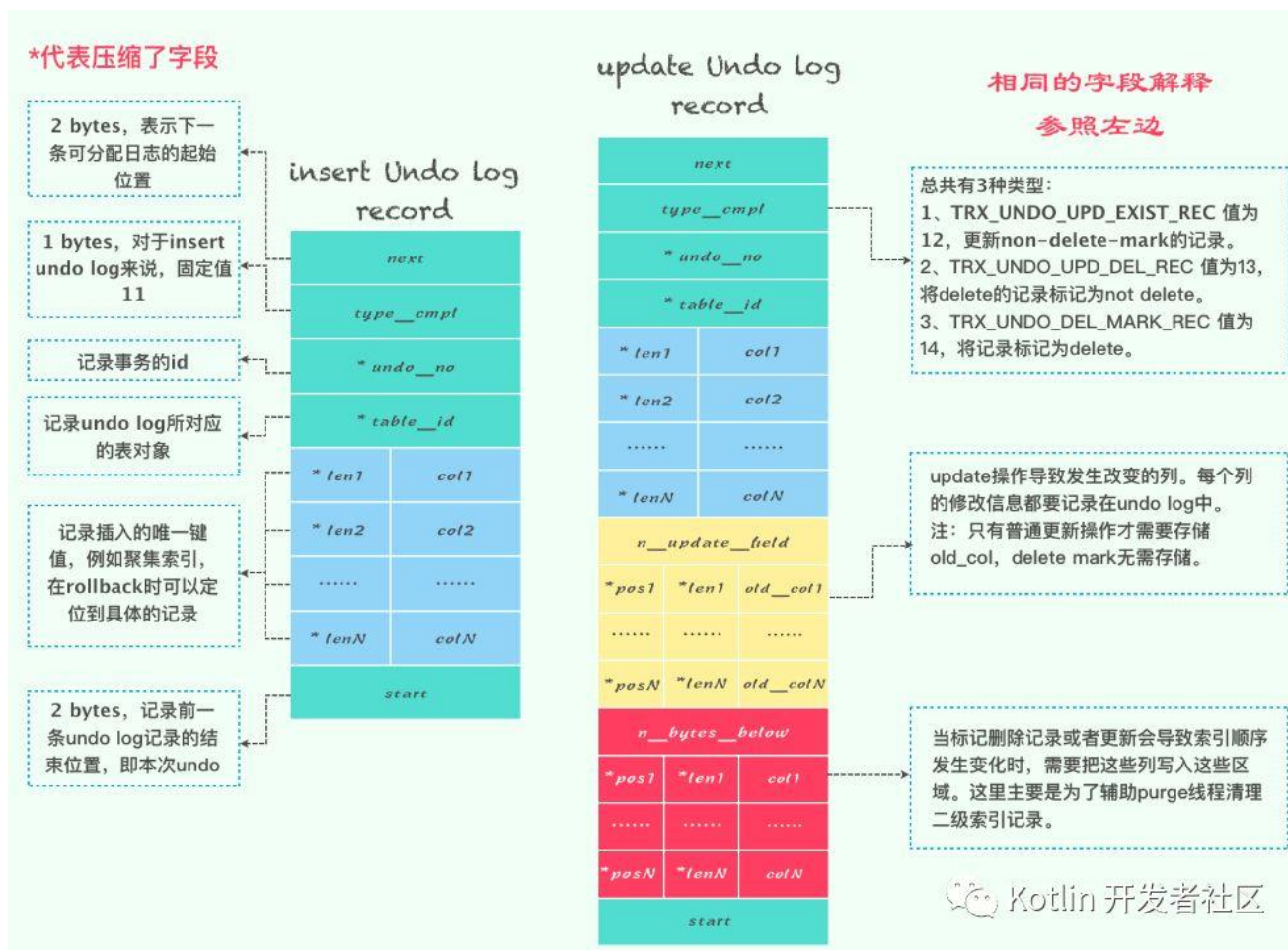
如果在内存中把数据改了, 还没来得及落磁盘, 而此时数据库宕机怎么办?

Redo事务开始时, 记录每次的变更信息。且仅记录内存还未刷新到硬盘的数据, 不是所有记录。循环日志。

■ 事务日志：redo log （每个块：512字节）



■ 事务日志: undo log (insert/update[含delete])



<https://blog.csdn.net/universsky2015/article/details/10251078310>

11.8 MySQL中的事务日志文件



- **redo log** : **物理日志**，提升事务**持久化**处理的性能。
只记录事务对数据页做了哪些修改。在磁盘上由名为 `ib_logfile0` 和 `ib_logfile1`。包含两部分：先写**内存** -> 后续某个时间一次性写多条到**磁盘**。

| | | | |
|--|------------------|----|-----------|
|  <code>ib_logfile0</code> | 2020/11/8 19:49 | 文件 | 49,152 KB |
|  <code>ib_logfile1</code> | 2020/10/30 20:06 | 文件 | 49,152 KB |

内存中的日志缓冲 (redo log buffer, 易失)

磁盘上的日志文件 (redo log file, 稳定)

- **undo log** : **逻辑日志**，用于保证数据的**原子性**。
保存事务发生之前的数据的一个版本，可以用于回滚。是 MVCC (多版本控制) 实现的关键。

| | | | |
|---|-----------------|----|-----------|
|  <code>undo_001</code> | 2020/11/8 19:49 | 文件 | 49,152 KB |
|  <code>undo_002</code> | 2020/11/8 19:49 | 文件 | 17,408 KB |

https://dev.mysql.com/doc/dev/mysql-server/8.0.11/PAGE_INNODB_REDO_LOG.html

二进制日志：Binlog文件：用于进行数据恢复的日志（主从同步或者恢复）

-- 显示binlog的基本信息

```
show variables like 'log_bin';    --确认binlog是否开启
show master logs;                -- 显示所有的binlog文件
show master status;              -- 显示最新的binlog的最后位置
show binlog events;              -- 显示所有的binlog
```

-- 显示指定binlog

```
show binlog events in 'THINKPAD_LINING-bin.000019' ;
```

-- 显示指定binlog中某个位置之后的log

```
show binlog events in 'THINKPAD_LINING-bin.000019' from 40080;
show binlog events in 'THINKPAD_LINING-bin.000019' from 40080 limit 10;
```

--刷新binlog，生成一个新的binlog文件

```
flush logs;
```

Binlog文件：用于进行事务恢复的日志

- Row: 不记录sql语句上下文相关信息，仅保存哪条记录被修改。
- Statement: 每一条会修改数据的sql都会记录在binlog中。
- Mixedlevel: 是以上两种的混合使用，一般的语句修改使用statement格式保存binlog，如一些函数；statement无法完成主从复制的操作，则采用row格式保存binlog. MySQL会根据执行的每一条具体的sql语句来区分对待记录的日志形式。

| Log_name | Pos | Event_type | Server_id | End_log_pos | Info |
|----------------------------|-------|----------------|-----------|-------------|---|
| THINKPAD_LINING-bin.000019 | 42065 | Table_map | 1 | 42130 | table_id: 129 (trans.icbc_card) |
| THINKPAD_LINING-bin.000019 | 42130 | Update_rows | 1 | 42202 | table_id: 129 flags: STMT_END_F |
| THINKPAD_LINING-bin.000019 | 42202 | Xid | 1 | 42233 | COMMIT /* xid=1851 */ |
| THINKPAD_LINING-bin.000019 | 42233 | Anonymous_Gtid | 1 | 42312 | SET @@SESSION.GTID_NEXT = 'ANONYMOUS' |
| THINKPAD_LINING-bin.000019 | 42312 | Query | 1 | 42397 | BEGIN |
| THINKPAD_LINING-bin.000019 | 42397 | Table_map | 1 | 42462 | table_id: 129 (trans.icbc_card) |
| THINKPAD_LINING-bin.000019 | 42462 | Update_rows | 1 | 42534 | table_id: 129 flags: STMT_END_F |
| THINKPAD_LINING-bin.000019 | 42534 | Xid | 1 | 42565 | COMMIT /* xid=1865 */ |
| THINKPAD_LINING-bin.000019 | 42565 | Anonymous_Gtid | 1 | 42644 | SET @@SESSION.GTID_NEXT = 'ANONYMOUS' |
| THINKPAD_LINING-bin.000019 | 42644 | Query | 1 | 42729 | BEGIN |
| THINKPAD_LINING-bin.000019 | 42729 | Table_map | 1 | 42794 | table_id: 129 (trans.icbc_card) |
| THINKPAD_LINING-bin.000019 | 42794 | Update_rows | 1 | 42866 | table_id: 129 flags: STMT_END_F |
| THINKPAD_LINING-bin.000019 | 42866 | Xid | 1 | 42897 | COMMIT /* xid=1874 */ |
| THINKPAD_LINING-bin.000019 | 42897 | Anonymous_Gtid | 1 | 42974 | SET @@SESSION.GTID_NEXT = 'ANONYMOUS' |
| THINKPAD_LINING-bin.000019 | 42974 | Query | 1 | 43087 | use `trans`; create table t (id int) /* xid=1920 */ |

11.8 MySQL中的事务日志文件



Binlog文件：用于binlog进行事务恢复

创建新的binlog日志文件

```
flush logs;  
show master status;  
假设：最新：mysql-bin.000022
```

执行常规SQL数据操作（含创建，增删改操作）

从日志中找回待恢复之前的SQL语句，导出为test000022.sql

1. mysqlbinlog.exe mysql-bin.000022 > test_000022.txt
2. 在txt日志中查找待恢复（如DROP TABLE）日志的位置（该语句的at 2413）
3. 导出binlog日志中'DROP TABLE'之前的SQL语句

```
mysqlbinlog mysql-bin.000022 -d db1 --skip-gtids --stop-position=2413 >  
test000022.sql
```

在mysql中执行以上SQL文件

```
source C:\ProgramData\MySQL\MySQL Server 8.0\Data\test000022.sql
```

Binlog文件：（输出为txt文件）

```
229 /*!80014 SET @@session.immediate_server_version=80021*//*!*/;
230 SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
231 # at 2413
232 #201103 21:21:37 server id 1 end_log_pos 2538 CRC32 0x599dd0ba Query thread_id=35 exec_time=1519 error_code=0 Xid = 2255
233 SET TIMESTAMP=1604409697/*!*/;
234 DROP TABLE `t1` /* generated by server */
235 /*!*/;
236 # at 2538
237 #201103 21:21:37 server id 1 end_log_pos 2617 CRC32 0xe5d082ed Anonymous_GTID last_committed=10 sequence_number=11 rbr_only=yes or
238 /*!50718 SET TRANSACTION ISOLATION LEVEL READ COMMITTED*//*!*/;
239 # original_commit_timestamp=1604411225757273 (2020-11-03 21:47:05.757273 中国标准时间)
240 # immediate_commit_timestamp=1604411225757273 (2020-11-03 21:47:05.757273 中国标准时间)
```

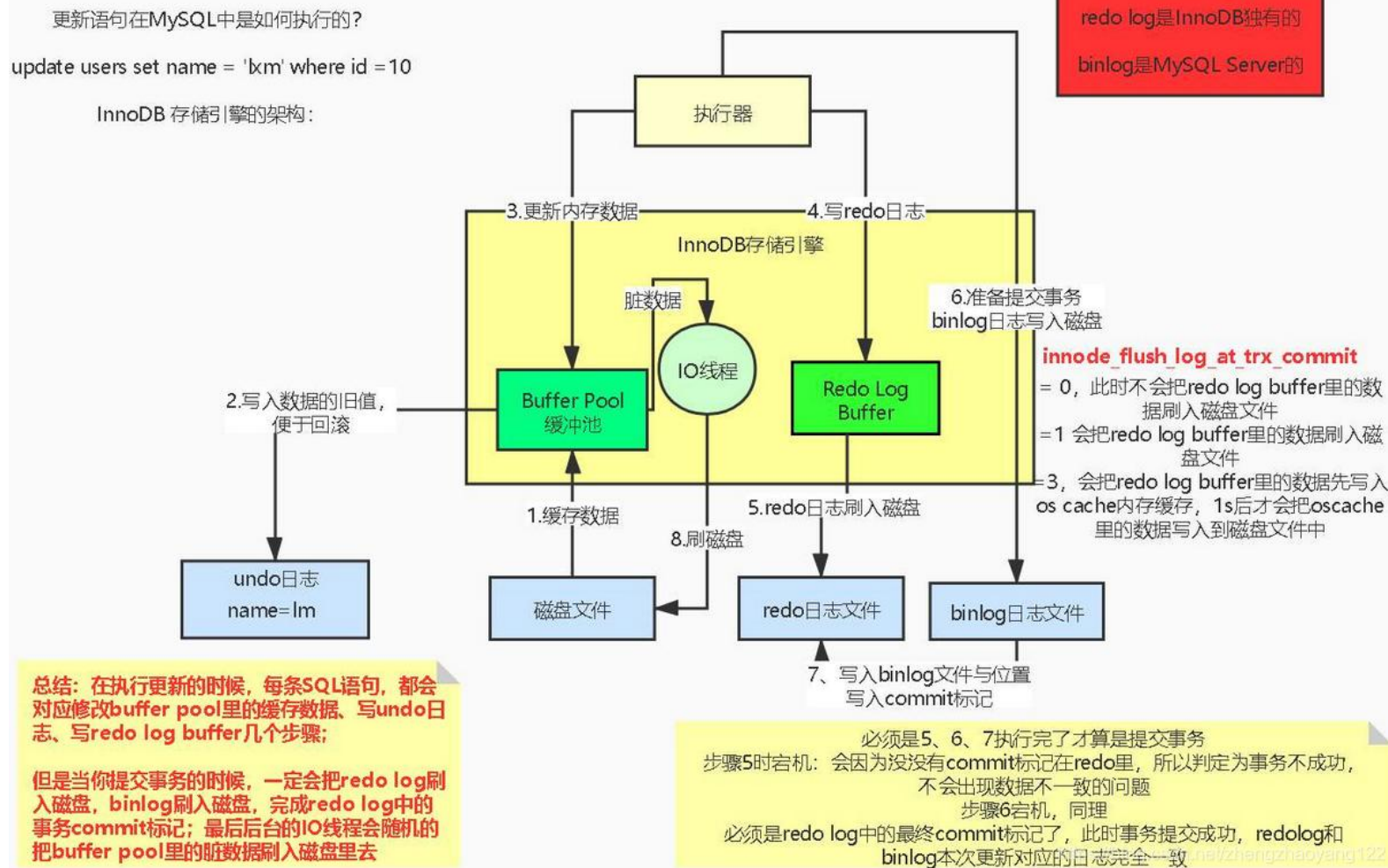
Window的日志默认路径：

C:\ProgramData\MySQL\MySQL Server 8.0\Data

11.8 MySQL中的事务日志文件



Update 语句执行过程的redo log, binlog, undo log



<https://blog.csdn.net/zhengzhaoyang122/article/details/109462404>



- redo log: (innodb)
事务更新已成功，但数据文件尚未写成功
- undo log: (innodb)
事务失败回滚，MVCC
- Binlog: (mysql)
备份，主从复制

■ 本章重点

- 事务的概念和性质
- 事务故障、系统故障和介质故障的恢复技术与原理
- Redo & Undo
- MySQL的日志文件

■ 本章难点

- 具有检查点的恢复技术

作业：第六版教材 本章课后第3，4，5，8题

