事务的定义与特性

> ## 为何数据库需要进行事务管理
>
> 通常一个SQL语句是关系数据库的一个原子操作
>
> 修改原子操作的定义, 将多个SQL语句定义为一个原子操作
>
> - 完整性约束检测时机是在原子操作之前或之后
> - 单条SQL语句不触发完整性约束检测, 保证数据库的一致性

- 事务的定义

> 事务 是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位.
>
> 定义事务的SQL语句
>
> ```
> BEGIN TRANSACTION
> COMMIT
> ROLLBACK
> ```

- 事务的特性
  - 原子性(Atomicity)
    事务包含的一组更新操作是原子不可分的，即更新操作对于数据库而言，要么全做，要么全不做，不能部分地完成
  - 一致性(Consistency)
    当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态
  - 隔离性(Isolation)
    系统允许的任何交错操作调度等价于某一个串行调度，即可串行性
  - 持久性(Durability)
    当事务发出提交语句后系统返回到程序逻辑时，必须保证该事务是可恢复的
- 破坏ACID特性的因素

  破坏事务ACID特性的因素有：

  (1) 为了有效地利用系统资源, 多个事务并发(concurrency).
  - 若并发事务操作相同的数据并且这些操作是冲突的, 会破坏事务的隔离性.
  - DBMS需要对事务进行并发控制, 保证事务的隔离性, 从而保证数据库中数据的一致性.

  (2) 事务的运行过程中被强行停止
  故障恢复

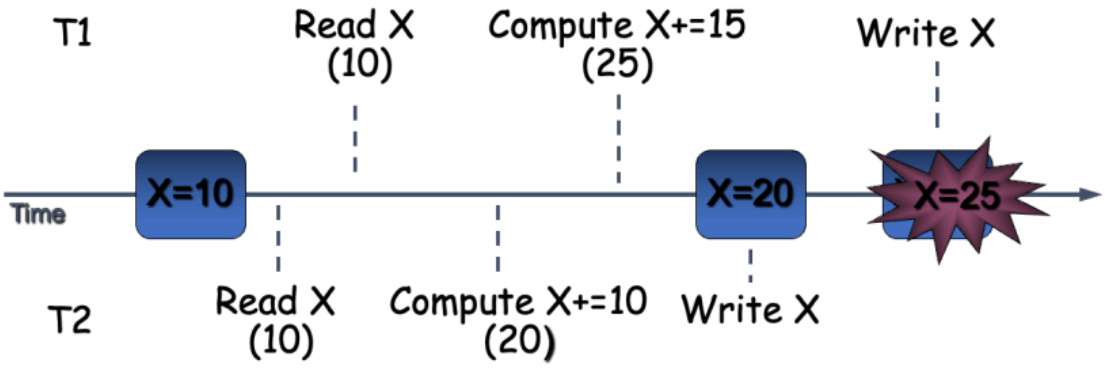
事务并发控制

事务并发导致的不一致性

- 丢失修改(Lost Update)

丢失修改(Lost Update)

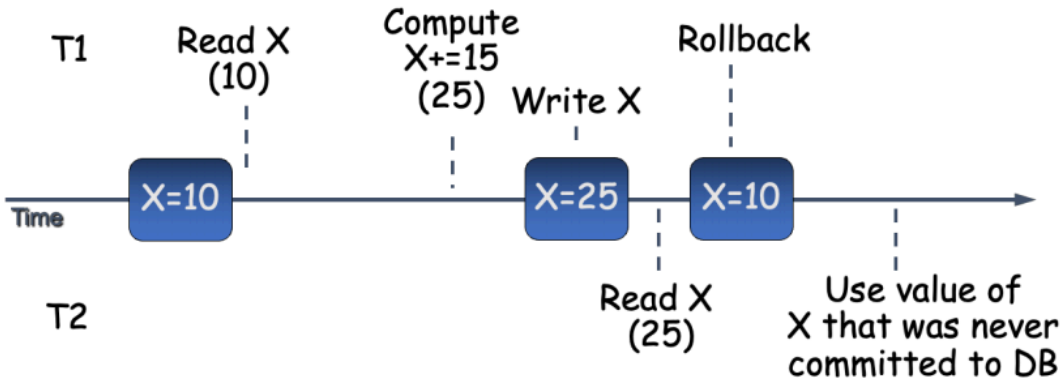
例 两个事务T_1和T_2读入同一数据并修改



- 脏读(Dirty Read)

脏读(Dirty Read)

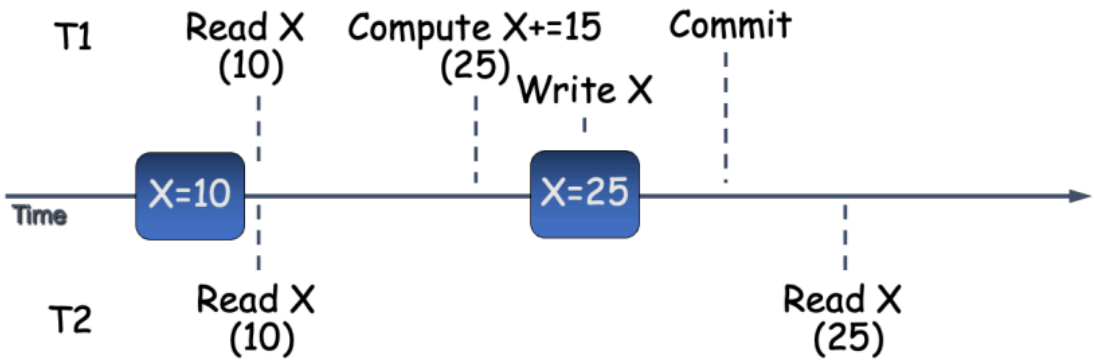
例 事务T_1修改数据X, 事务T_2读取X后, T_1被撤消, 并将X恢复原值, T_2读到的值与数据库不一致



- 不可重复读(Non-Repeatable Read)

不可重复读(Non-Repeatable Read)

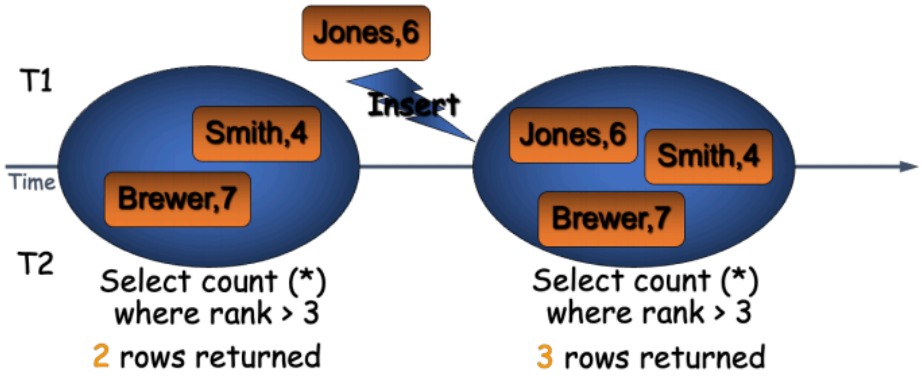
例 事务T_2读取数据X后, 事务T_1更新该数据X, T_2再读X时, 得到的结果不同



- 幻读(Phantom Read)

幻读(Phantom Read)

例 事务T_1按一定条件读数据库后, 事务T_2删除或插入了某些记录, 当事务T_1再次按相同条件读数据库时, 发现某些记录神秘消失或出现

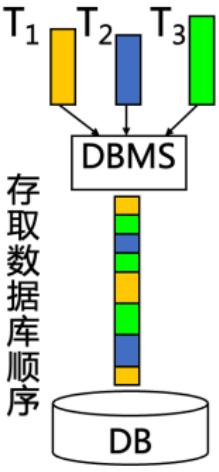


解决方法

出现不一致性的原因:
不同事务间冲突操作的无序执行

如何控制冲突操作的无序?
并发控制的方法

- 对事务规定一定的顺序, 时间戳
- 对共享的资源进行控制, 加锁



封锁及封锁协议

- 封锁的定义

  封锁的定义

  事务T在对某个数据对象例如表、记录等操作之前, 先向系统发出请求, 对其加锁. 加锁后事务T就对该数据对象有了一定的控制, 在事务T释放它的锁之前, 其他事务对该数据的操作受到一定限制.

- 基本锁的类型

基本的封锁类型

- 排它锁(Exclusive Locks, 简称X锁, 也称写锁)
- 共享锁(Share Locks, 简称S锁, 也称读锁)

封锁类型相容矩阵

- N, 不相容
- Y, 相容

	T_2 X	T_2 S	T_2
T_1 X	N	N	Y
T_1 S	N	Y	Y
T_1	Y	Y	Y

- 三种封锁协议

一、二、三级封锁协议的一致性保证

	X锁		S锁		一致性保证		
	操作结束释放	事务结束释放	操作结束释放	事务结束释放	不丢失修改	不读"脏"数据	可重复读
一级封锁协议		√			√		
二级封锁协议		√	√		√	√	
三级封锁协议		√		√	√	√	√

一级封锁协议

一级封锁协议就是加了一个X锁，也就是写锁，再事务结束后才释放，因此可以解决丢失修改这个问题

一级封锁协议

事务T在**修改**数据D之前必须先对其加X**锁**, 直到**事务结束**才释放. 事务结束包括正常结束(COMMIT)和非正常结束(ROLLBACK).

一级封锁协议 举例

	T_1	T_2
1）	读A=16	
2）	A←A–1 写回A=15	申请A的X锁，等待
3）		读A=15
4）		A←A–1 写回A=14

丢失修改

避免 √

	T_1	T_2
1）	读A=16 A←A*2 写回A=32	
2）		读A=32
3）	ROLLBACK A恢复为16	

读"脏"数据

×

	T_1	T_2
1）	读A=16	
2）		读A=16 A←A*2 写回A=32
3）	读A=32	

不可重复读

×

二级封锁协议

二级封锁协议就是在第一封锁协议的基础上，加上了S锁，但是要注意这个S锁是在读完后就解锁。S锁与X锁互斥，因此可以解决脏读这个问题

二级封锁协议

在一级封锁协议基础上，事务T在**读取**数据D之前必须先对其加S**锁，读完后即可释放**S**锁**.

二级封锁协议 举例

	T_1	T_2
1）	读A=16	
2）	A←A–1 写回A=15	申请A的X锁，等待
3）		读A=15
4）		A←A–1 写回A=14

丢失修改

避免 √

	T_1	T_2
1）	读A=16 A←A*2 写回A=32	
2）		申请A的S锁，等待
3）	ROLLBACK A恢复为16	

读"脏"数据

√

	T_1	T_2
1）	读A=16	
2）		读A=16 A←A*2 写回A=32
3）	读A=32	

不可重复读

×

三级封锁协议

三级封锁协议和二级封锁协议类似，但是注意的是S锁在事务结束后才释放，这就解决了不可重复读的问题，但是因为限制太高，因此系统效率也会下降

三级封锁协议

在一级封锁协议基础上，事务T在**读取**数据D之前必须先对其加S锁，**直到事务结束才释放**.

封锁及封锁协议

三级封锁协议 举例

	T_1	T_2
1）	读A=16	
2）		读A=16
3）	A←A–1 写回A=15	
4）		A←A–1 写回A=15

丢失修改

避免 √

	T_1	T_2
1）	读A=16 A←A*2 写回A=32	
2）		申请A的X锁，等待
3）	ROLLBACK A恢复为16	

读"脏"数据

√

	T_1	T_2
1）	读A=16	
2）		申请A的X锁，等待
3）	读A=16	

不可重复读

√

并发调度的可串行性

- 不可串行化调度，A=3,B=3

> 系统对并发事务中并发操作的调度是随机的，哪个（些）调度是正确的?
> 能将数据库置于一致状态的调度
>
> 将所有事务串行起来的调度策略一定是正确的调度
>
> 并发事务正确性准则: 多个事务的并发执行是正确的, 当且仅当其结果与某一次序串行地执行它们时的结果相同, 称这种调度策略为可串行化的调度.

例 现在有两个事务T_1和T_2

- T_1: 读B; A=B+1; 写回A
- T_2: 读A; B=A+1; 写回B

假定A、B的初值均为2, 则

- 按T_1、T_2次序执行结果为A=3, B=4
- 按T_2、T_1次序执行结果为A=4, B=3

例

T_1	T_2
Slock B	
Y=B=2	
Unlock B	
Xlock A	
A=Y+1	
写回A(=3)	
Unlock A	
	Slock A
	X=A=3
	Unlock A
	Xlock B
	B=X+1
	写回B(=4)
	Unlock B

串行调度

T_1	T_2
	Slock A
	X=A=2
	Unlock A
	Xlock B
	B=X+1
	写回B(=3)
	Unlock B
Slock B	
Y=B=3	
Unlock B	
Xlock A	
A=Y+1	
写回A(=4)	
Unlock A	

串行调度

T_1	T_2
Slock B	
Y=B=2	
	Slock A
	X=A=2
Unlock B	
	Unlock A
Xlock A	
A=Y+1	
写回A(=3)	
	Xlock B
	B=X+1
	写回B(=3)
Unlock A	
	Unlock B

不可串行化调度

T_1	T_2
Slock B	
Y=B=2	
Unlock B	
Xlock A	
	Slock A
A=Y+1	等待
写回A(=3)	等待
Unlock A	等待
	X=A=3
	Unlock A
	Xlock B
	B=X+1
	写回B(=4)
	Unlock B

可串行化调度

两段锁协议

什么样的方法能保证调度的可串行化?

两段锁协议
所有事务必须分两个阶段对数据项加锁和解锁

- 获得（扩展）阶段
  在对任何数据进行读、写操作之前，首先要申请并获得对该数据的封锁;
- 释放（收缩）阶段
  在释放一个封锁后，事务不再申请和获得任何其他封锁。

两段锁协议是可串行化的充分条件, 而不是必要条件.

- 死锁举例

  死锁

  遵守两段锁协议的事务可能发生死锁

  如果事务T_1封锁了数据D_1, 事务T_2封锁了D_2.
  然后, T_1请求封锁D_2, 等待.
  接着T_2请求封锁D_1, 等待.
  这样就出现了T_1等待T_2, 而T_2又在等待T_1的局面.
  这两个事务永远不能结束, 形成死锁.

T_1	T_2
Slock B 读B=2	
	Slock A 读A=2
Xlock A 等待	
等待	Xlock B
	等待

死锁

例 假设在进行余额汇总交易的同时, 发生了一笔转账交易, 从101账户转给104账户400元, 满足提交读隔离级别和两段锁协议

时间	汇总事务	转账事务
t_1	LS(101), 读101账户余额	
t_2		LX(104), 更新104账户余额
t_3	LS(102), 读102账户余额	
t_4	LS(103), 读103账户余额	
t_5		LX(101), 更新101账户余额
t_6	LS(104), 读104账户余额	
t_7	阻塞	阻塞

- 解决死锁的方法

  如何解决死锁?

  诊断

  - 超时法
    超过规定的时间就认为发生死锁
    不足: 可能误判死锁, 也可能不能及时发现死锁
  - 等待图法
    事务为结点, 边表示事务等待的情况, 若T_1等待T_2, 则$T_1 \rightarrow T_2$.
    系统周期性地检测事务等待图, 若图中存在回路, 则出现死锁

  解除
  选择一个处理死锁代价最小的事务, 撤消

- 封锁的粒度 略

隔离级别

SQL-92定义了四种隔离级别

隔离级别	脏读	不可重复读	幻读	说明
未提交读 (Read Uncommited)	Y	Y	Y	最低级别, 仅可保证不读取物理损坏的数据
提交读 (Read Commited)	X	Y	Y	
可重复读 (Repeatable Read)	X	X	Y	
串行 (Serializable)	X	X	X	最高级别, 事务之间完全隔离

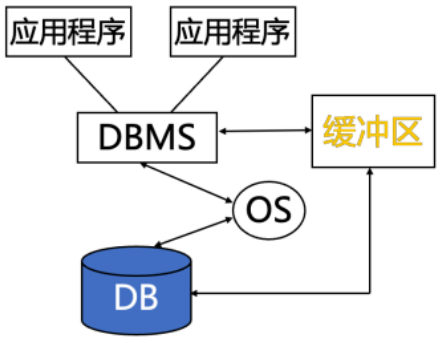
故障恢复

- 故障恢复的目标与任务

故障恢复的目标与任务

数据库发生故障时, 把数据库从错误状态恢复到某一个已知的正确状态(亦称为一致状态或完整状态).

DB与内存用户工作区之间的数据交换是通过缓冲区进行的, 这个交换一般是以缓冲区是否满来触发的.
因此, 有可能提交事务的数据仍在缓冲区而没有写到DB中, 而未提交事务的数据却写到了DB中.
所以, 故障恢复时, 既要REDO已经提交了的事务, 又要UNDO未提交的事务, 以保证事务的原子性.



- 基于备份数据的恢复

基于备份数据的恢复

将数据库复制、保存起来, 这些备用的数据文本称为后备副本或后援副本

- 静态
- 动态递增
- 双机热备份
- 区块链, 分布式数据库

- 基于日志的恢复

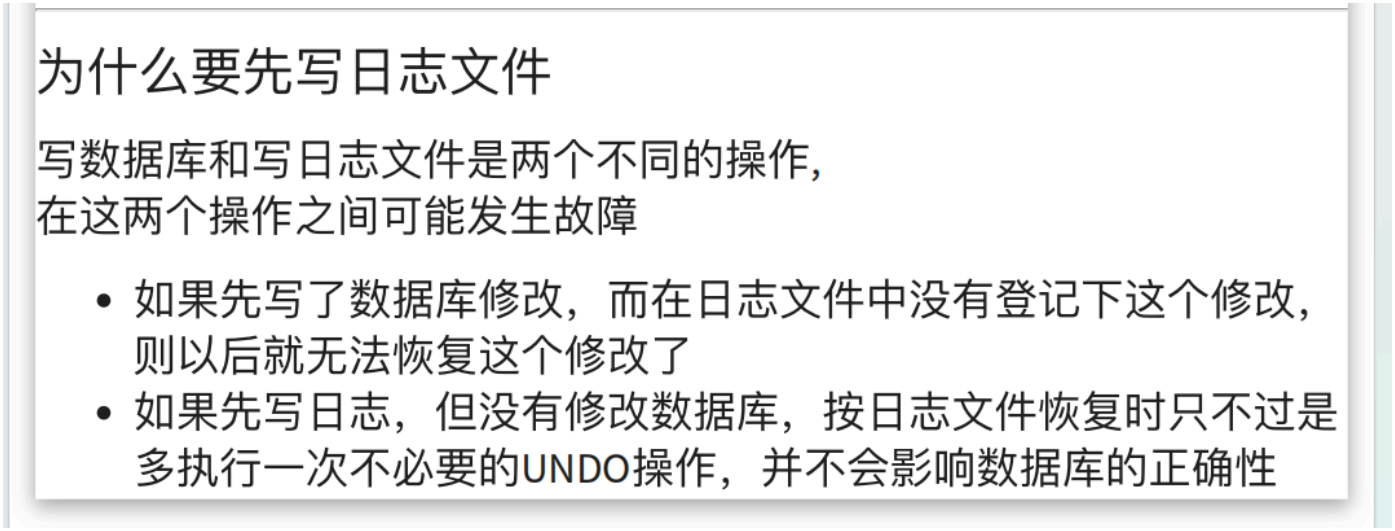
记录日志

- 各个事务的开始标志(BEGIN TRANSACTION)
- 各个事务的结束标志(COMMIT 或 ROLLBACK)
- 各个事务的所有更新操作

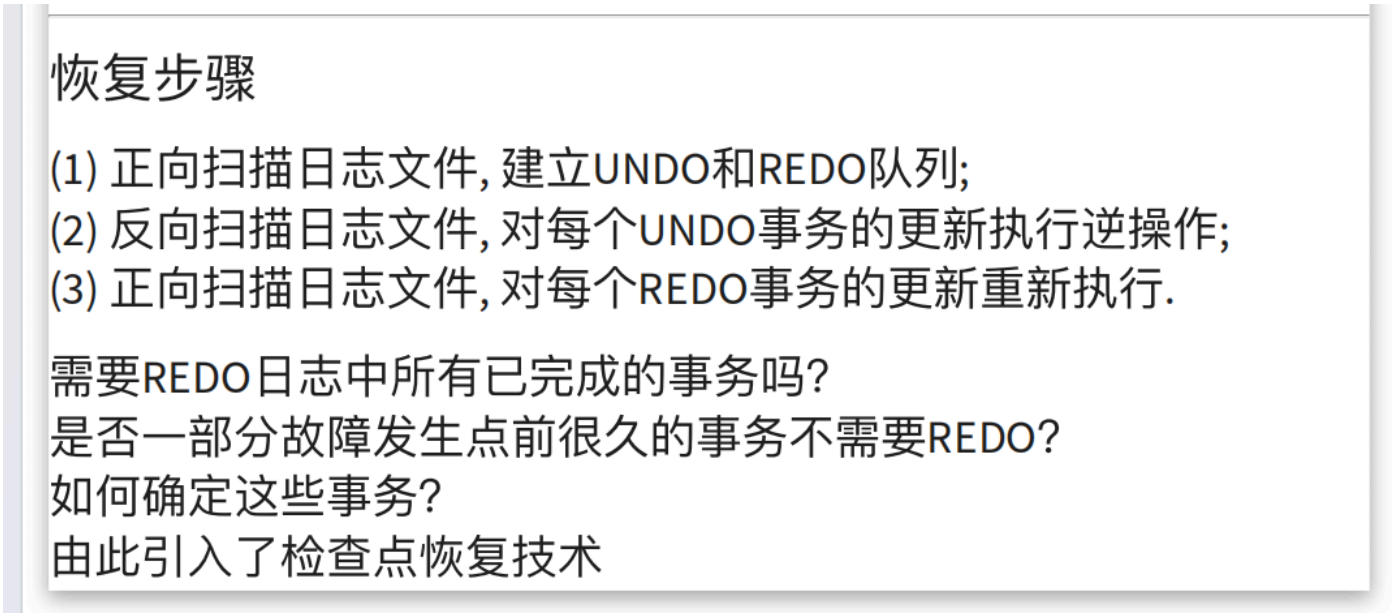
事务标识	操作类型	操作对象	旧值	新值
T_1	BEGIN			
T_2	INSERT	X	NULL	2
T_1	UPDATE	Y	5	7
T_1	DELTE	Z	abc	NULL
T_1	COMMIT			

记录日志 必须遵循两条原则

- 登记的次序严格按并行事务执行的时间次序
- 必须先写日志文件，后写数据库 (Write-Ahead Logging, WAL)
  - 写日志文件操作：把表示这个修改的记录写到日志文件
  - 写数据库操作：把对数据的修改写到数据库中



- 恢复步骤，提交了的事务就是要REDO队列中的，没有提交的事务就是UNDO队列里面的，反向扫描对UNDO事务执行逆操作，正向扫描对REDO事务重新执行



- 检查点技术，主要是简化REDO队列，在CHECKPOIONT和CRASH之间提交的事务才进入REDO队列，而检查点之前的不用进入REDO队列

检查点技术(checkpoint) 最小化须REDO的事务数量

例 某数据库系统的日志记录如表所示

编号	内容
LSN1	〈T1, START〉
LSN2	〈T1, I, 22, 3〉
LSN3	〈T2, START〉
LSN4	〈T2, L, 32, 37〉
LSN5	〈T3, START〉
LSN6	〈T2, J, 45, 5〉
LSN7	〈T7, START〉

编号	内容
LSN8	〈T1, COMMIT〉
LSN9	〈T3, M, 53, 15〉
LSN10	〈T3, K, 9, 11〉
LSN11	CHECKPOINT
LSN12	〈T2, COMMIT〉
LSN13	CRASH

系统恢复时REDO事务列表: T2; UNDO事务列表: T3
数据项初始值为: I=22, J=45, K=9; 系统恢复后: I=3, J=5, K=9