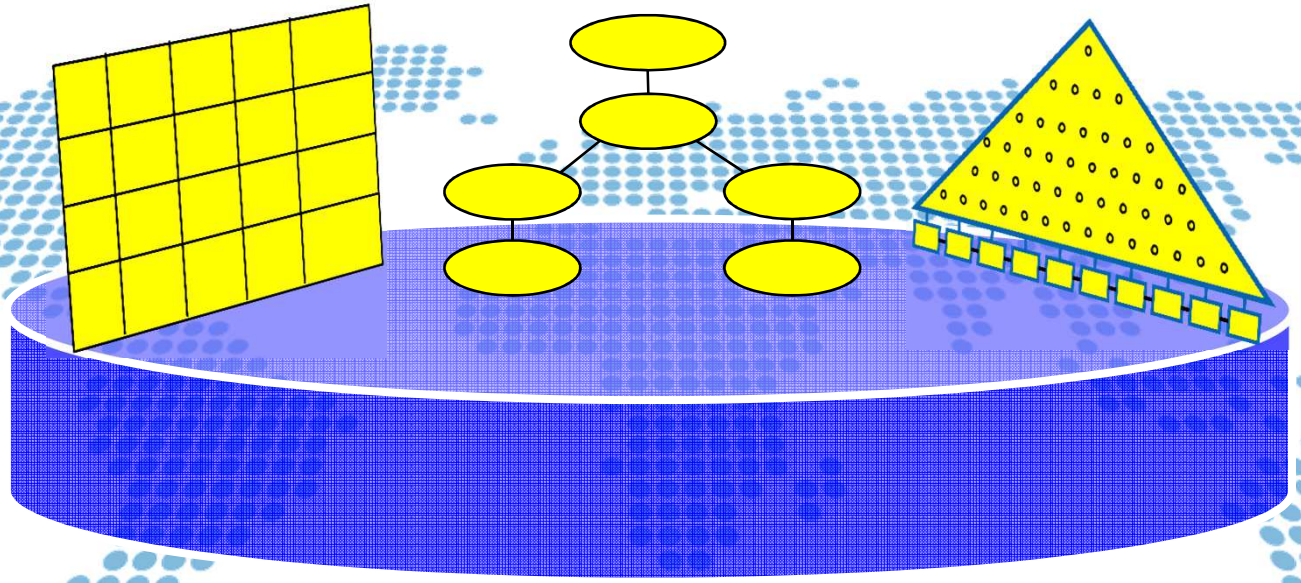
数据库系统

事务处理

陈世敏

（中科院计算所）

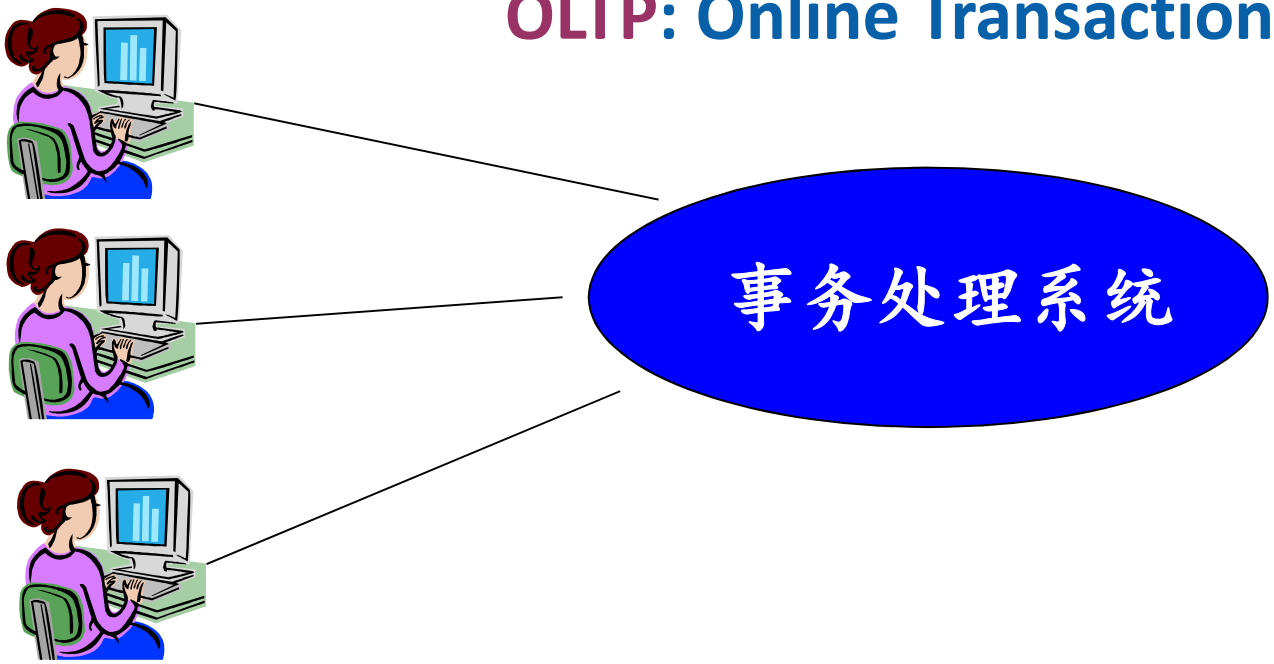


Outline

- 事务的概念和ACID
- Concurrency Control (并发控制)
- Crash Recovery (崩溃恢复)

事务处理 (Transaction Processing)

OLTP: Online Transaction Processing



- 典型例子：银行业务，订票，购物等
- 大量并发用户，少量随机读写操作

什么叫事务？（Transaction）

- 一个事务可能包含多个操作
  - select
  - insert/delete/update
  - 等
- 事务中的所有操作满足ACID性质

事务的表现形式

- 没有特殊设置
  - 那么每个SQL语句被认为是一个事务
- 使用特殊的语句
  - 开始transaction
  - 成功结束transaction
  - 异常结束transaction

开始一个Transaction

- 不同的系统使用的命令不同，但是意思基本一样
- 我们以PostgreSQL为例进行介绍

```
START TRANSACTION [ transaction_mode [, ...] ];
```

或者

```
BEGIN TRANSACTION [ transaction_mode [, ...] ];
```

其中*transaction_mode*可以是：

- ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
- READ WRITE | READ ONLY
- 等

成功结束一个Transaction

```
COMMIT TRANSACTION;
```

- 当前的事务成功结束
- 数据库系统保证事务的任何写操作都不丢失

异常终止一个Transaction

```
ROLLBACK TRANSACTION;
```

- 当前的事务进行中，发现某些条件不满足，需要主动进行异常终止
- 数据库系统丢弃事务所有的修改信息，返回事务初始的状态

Transaction

成功的事务

```
begin transaction;
……
commit transaction;
```

可以用rollback回卷事务

```
begin transaction;
……
rollback transaction;
```

ACID：DBMS保证事务的ACID性质

- Atomicity（原子性）
  - all or nothing
  - 要么完全执行，要么完全没有执行
- Consistency（一致性）
  - 从一个正确状态转换到另一个正确状态
    （正确指：constraints, triggers等）
- Isolation（隔离性）
  - 每个事务与其它并发事务互不影响
- Durability（持久性）
  - Transaction commit后，结果持久有效，crash也不消失

一致性和隔离性

- 对于单个执行的事务，在没有其他干扰的情况下
  - 应该满足一致性
  - 保证各种完整性约束条件是正确的

- 对于并发执行的多个事务
  - 不同事务之间应该互不影响，不能看到事务内部中间状态
  - 每个事务仍然需要满足一致性

☞Concurrency Control (并发控制)

原子性和持久性

- 三种原因可能导致事务非正常结束
  - 数据库系统内部异常
    - 例如，由于多事务竞争等原因
  - 掉电或其它原因导致系统崩溃
  - 事务逻辑本身决定没有达到预期，需要非正常终止

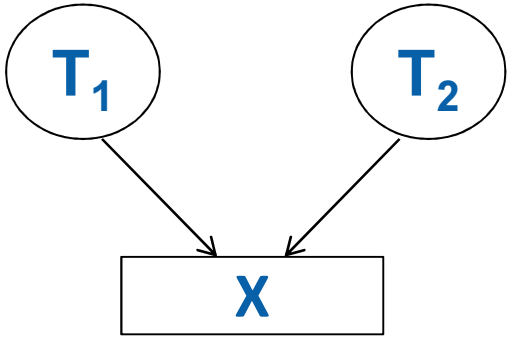
- 前两种情况需要自动恢复

☞Crash Recovery (崩溃恢复)

Outline

- 事务的概念和ACID
- Concurrency Control (并发控制)
  - 数据冲突和可串行化
  - 加锁的并发控制
  - 乐观的并发控制
- Crash Recovery (崩溃恢复)

同一个数据元素被并发访问

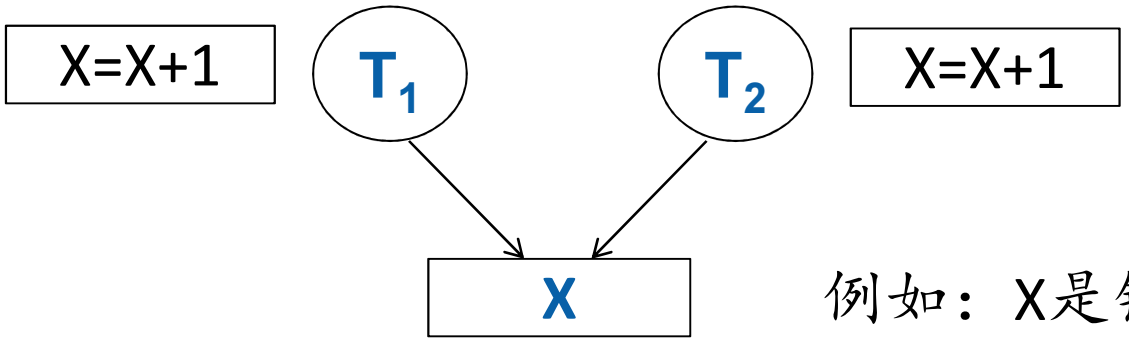


例如：X是银行账户余额

- 会有什么问题？

数据竞争(Data Race)

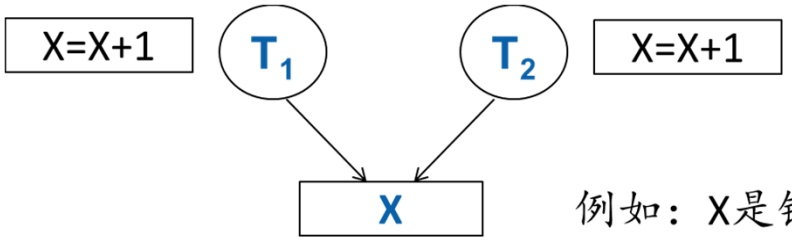
- 当两个并发访问都是写，或者一个读一个写时



例如：X是银行账户余额

- 场景：同一个账户两笔转帐并发发生会怎样？
  - 初始：X=100
  - 最终X=?

Schedule
(调度/执行顺序)



例如：X是银行账户余额

初始： X=100

T1	T2
Read(X)	
	Read(X)
	Write(X)
Write(X)	

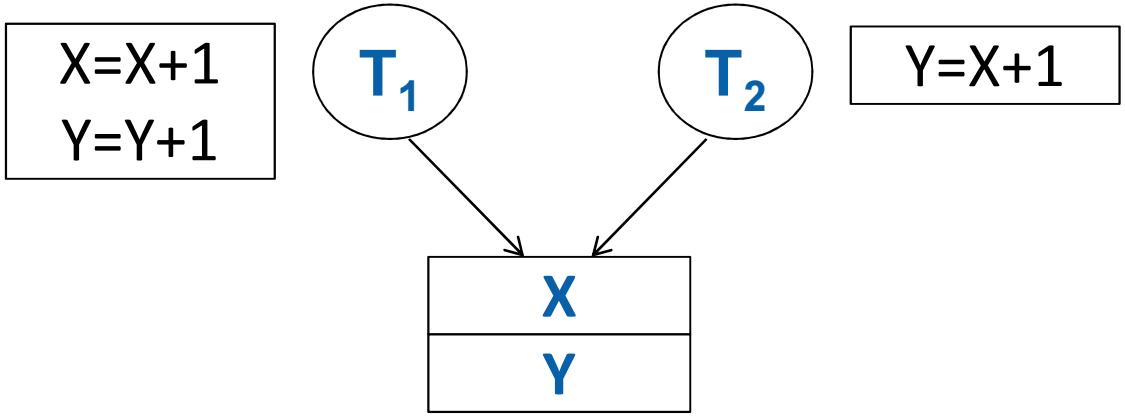
X=101

T2的write被覆盖了

T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

X=102

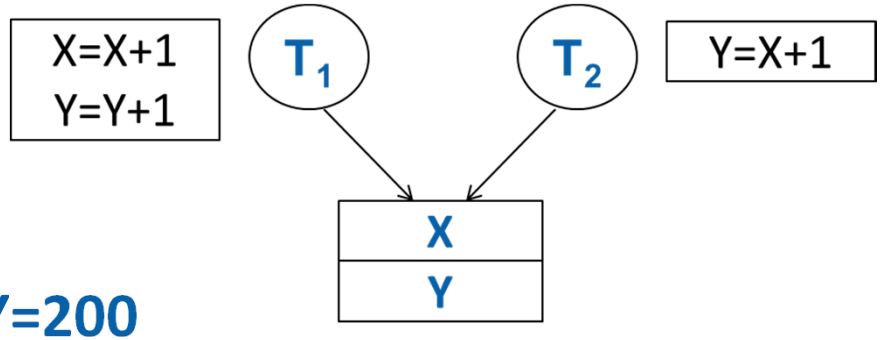
更复杂的情况



- 两个Transactions并发访问多个共享的数据元素
- 实际上，真实情况更加复杂

Schedule

(调度/执行顺序)



初始： X=100, Y=200

T1	T2
Read(X)	
	X=100
	Read(X)
	Write(Y)
	Y=101
Write(X)	
Read(Y)	
Write(Y)	

Y=102

T1	T2
Read(X)	
Write(X)	
	X=101
	Read(X)
	Write(Y)
	Y=102
Read(Y)	
Write(Y)	

Y=103

T1	T2
Read(X)	
	X=100
	Read(X)
Write(X)	
Read(Y)	
Write(Y)	
	Write(Y)

Y=101

……

正确性问题

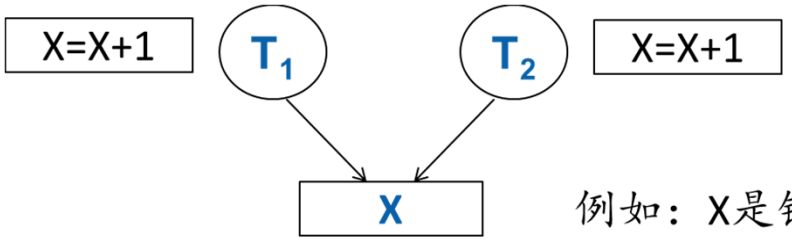
- 提出解决方案前，我们必须提问
- 如何判断一组Transactions正确执行？

- 存在一个顺序，按照这个顺序依次**串行执行**这些Transactions，得到的结果与**并行执行**相同

Serializable(可串行化)

- 判断一组并行Transactions是否正确执行的标准

Serializable?



例如：X是银行账户余额

初始： X=100

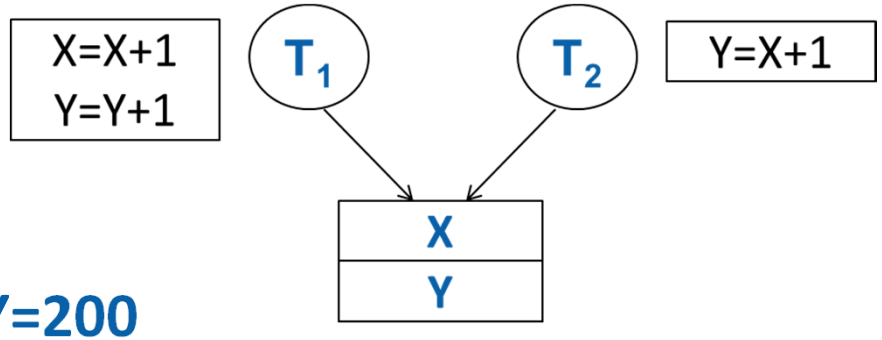
T1	T2
Read(X)	
	Read(X)
	Write(X)
Write(X)	

X=101

T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

X=102

Serializable?



初始： X=100, Y=200

T1	T2
Read(X)	
	Read(X)
	Write(Y)
Write(X)	
Read(Y)	
Write(Y)	

Y=102

T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(Y)
Read(Y)	
Write(Y)	

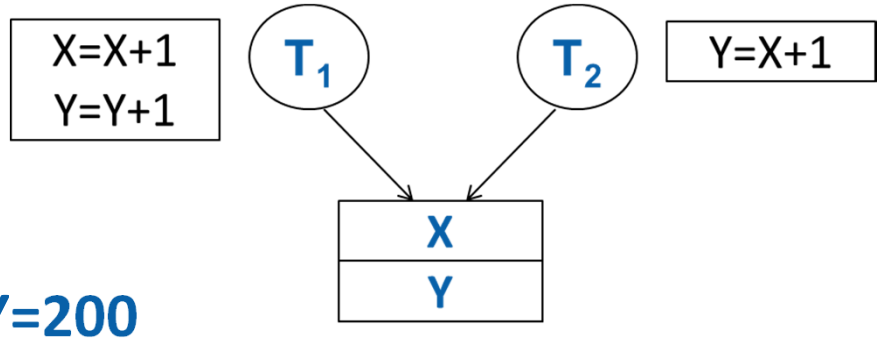
Y=103

T1	T2
Read(X)	
	Read(X)
Write(X)	
Read(Y)	
Write(Y)	
	Write(Y)

Y=101

……

Serializable?



初始： X=100, Y=200

T1	T2
Read(X)	
	Read(X)
	Write(Y)
Write(X)	
Read(Y)	
Write(Y)	

Y=102

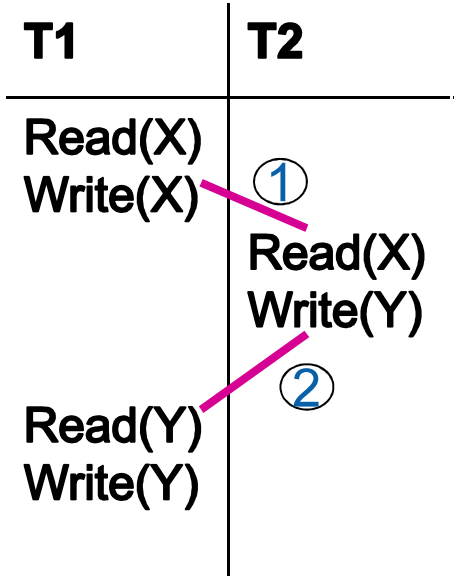
=

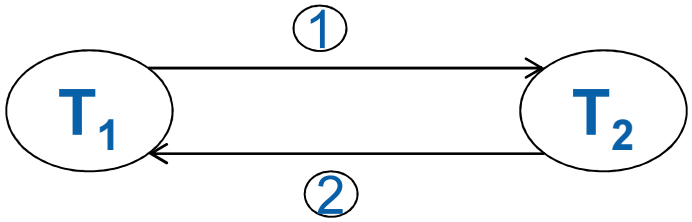
T1	T2
	Read(X)
	Write(Y)
Read(X)	
Write(X)	
Read(Y)	
Write(Y)	

优先图

- 确定一个并发执行是否可串行化
- 优先图的画法
  - 每个事务：图的顶点
  - 如果Ti的一个操作与Tj的一个操作相冲突，并且Ti的操作先于Tj的操作，那么就有一条从Ti指向Tj的边
- 判断是否可串行
  - 可串行当且仅当优先图中没有环

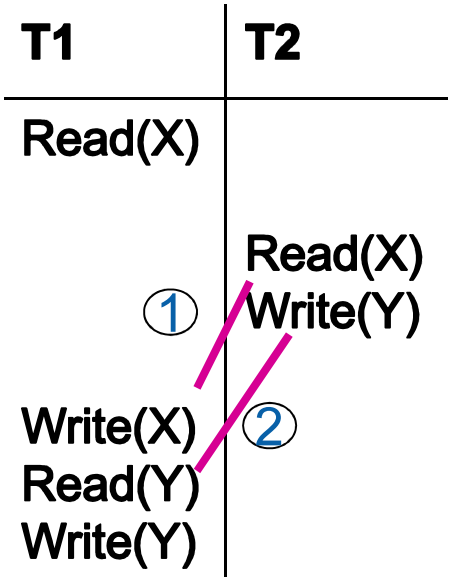
优先图举例

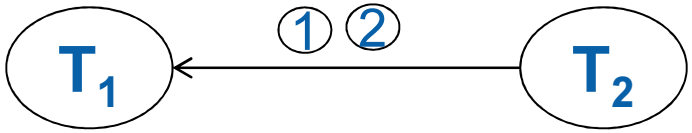




存在环，不可串行

优先图举例

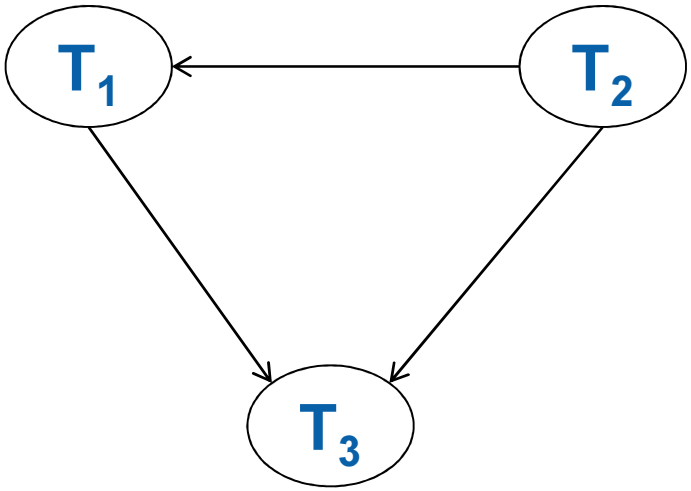


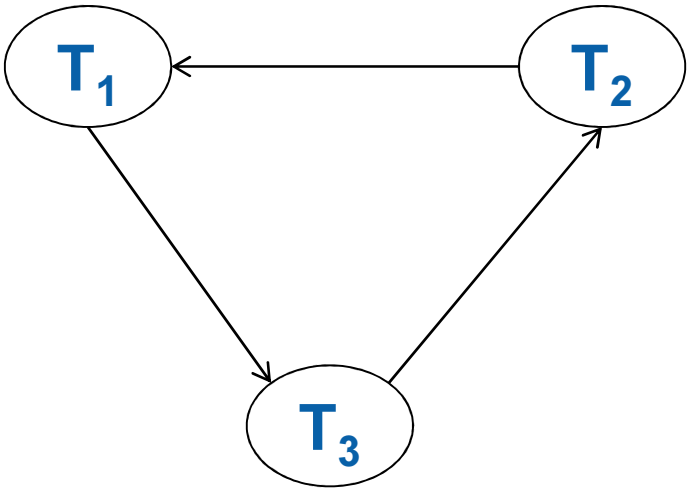


没有环，可串行

多个事物的复杂情况

- 优先图可以表达多个事务之间的复杂情况





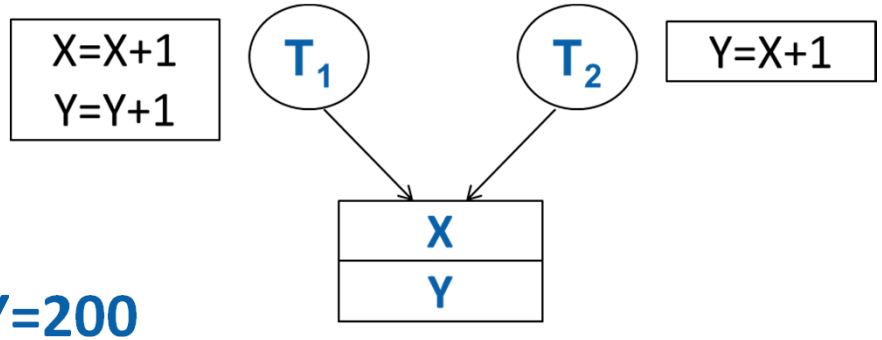
数据冲突引起的问题

- Read uncommitted data (读未提交的数据) (写读)
  - 在T2 commit之前，T1读了T2已经修改了的数据
- Unrepeatable reads(不可重复读) (读写)
  - 在T2 commit之前，T1写了T2已经读的数据
  - 如果T2再次读同一个数据，那么将发现不同的值
- Overwrite uncommitted data (重写未提交数据) (写写)
  - 在T2 commit之前，T1重写了T2已经修改了的数据

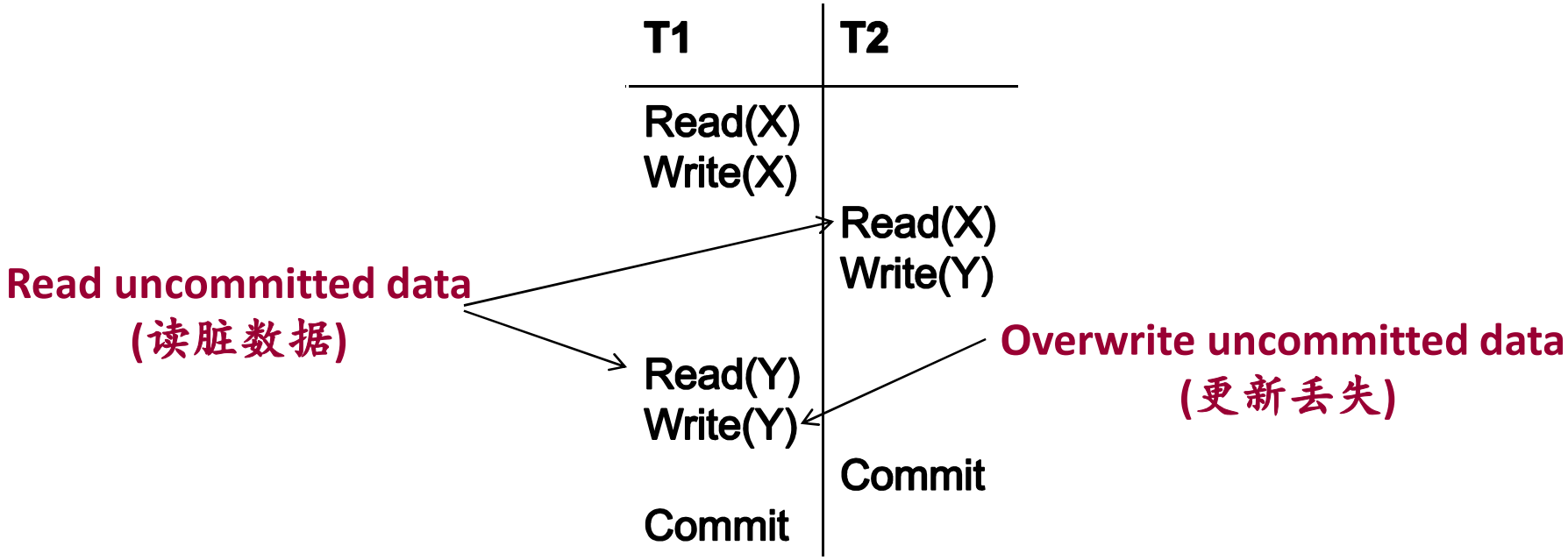
Isolation Level

	Read uncommitted data （写读）	Unrepeatable Read （读写）	Overwrite uncommitted Data （写写）
Serializable	no	no	no
Repeatable Read	no	no	possible
Read committed	no	possible	possible
Read uncommitted	possible	possible	possible

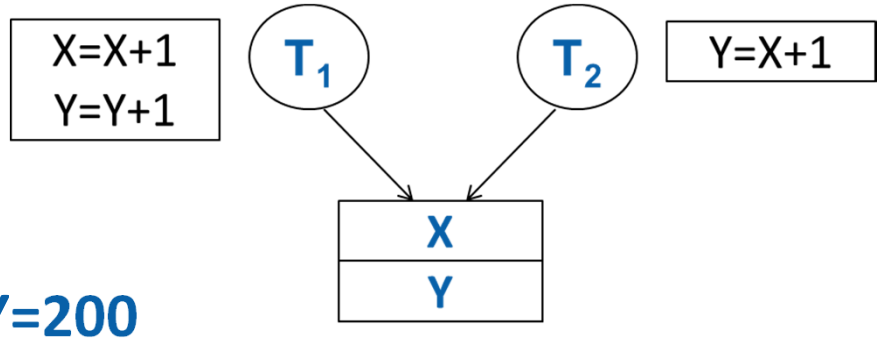
问题举例



初始： X=100, Y=200



问题举例



初始： X=100, Y=200

T1	T2
Read(X)	
	Read(X)
Write(X)	
Read(Y)	
Write(Y)	
	Write(Y)
	commit
commit	

Unrepeatable reads
(不可重复读)

Overwrite uncommitted data
(更新丢失)

解决数据冲突：两大类解决方案

- Pessimistic (悲观)
  - 假设：数据竞争可能经常出现
  - 防止：采用某种机制确保数据竞争不会出现
    - 如果一个Transaction T_1可能和正在运行的其它Transaction有冲突，那么就让这个T_1等待，一直等到有冲突的其它所有Transaction都完成为止，才开始执行。
- Optimistic (乐观)
  - 假设：数据竞争很少见
  - 检查：
    - 允许所有Transaction都直接执行
    - 但是Transaction不直接修改数据，而是把修改保留起来
    - 当Transaction结束时，检查这些修改是否有数据竞争
      - 没有竞争，成功结束，真正修改数据
      - 有竞争，丢弃结果，重新计算

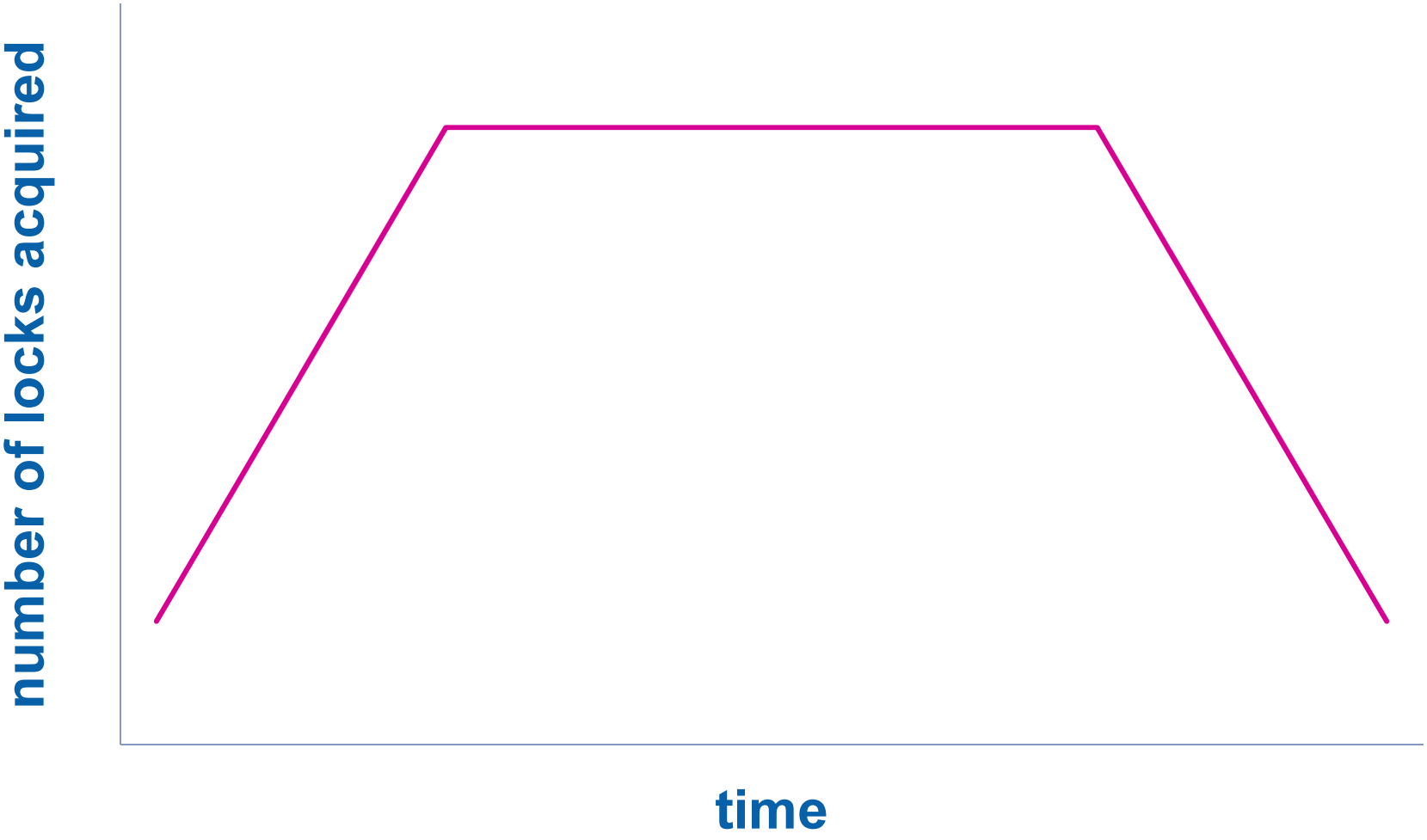
Pessimistic: 加锁

- 使用加锁协议来实现
- 对于每个事务中的SQL语句，数据库系统自动检测其中的读、写的数据
- 对事务中的读写数据进行加锁
- 通常采用两阶段加锁（2 Phase Locking）

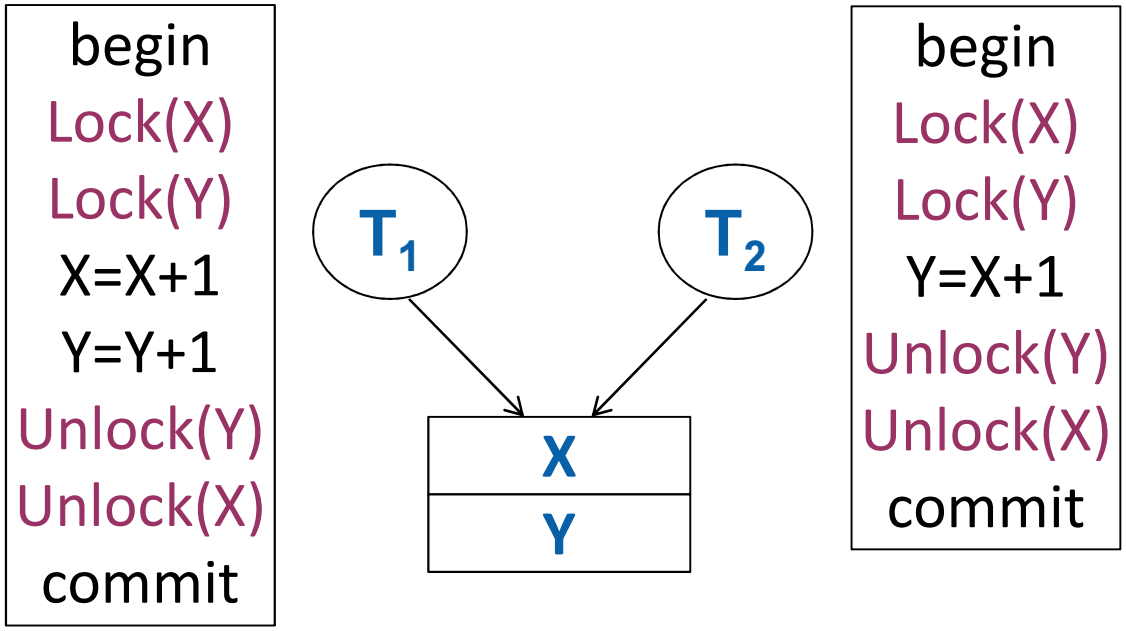
2 Phase Locking

- Pessimistic concurrency control
- 对每个访问的数据都要加锁后才能访问
- 算法如下
  - 在Transaction开始时，对每个需要访问的数据加锁
    - 如果不能加锁，就等待，直到加锁成功
  - 执行Transaction的内容
  - 在Transaction commit前，集中进行解锁
  - Commit
- 有一个集中的加锁阶段和一个集中的解锁阶段
  - 由此得名

2PL的执行过程



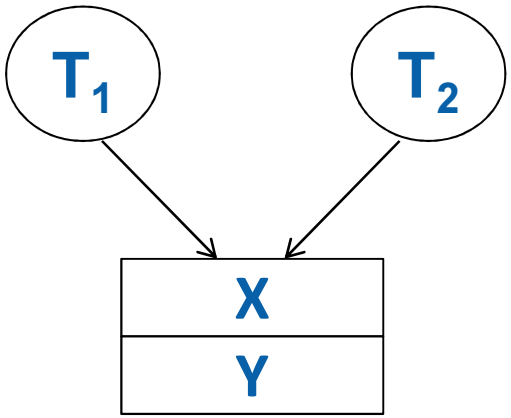
举例



- 这样一来，两个transactions不会同时执行

为什么一定要2-phase?

begin
Lock(X)
X=X+1
Unlock(X)
Lock(Y)
Y=Y+1
Unlock(Y)
commit



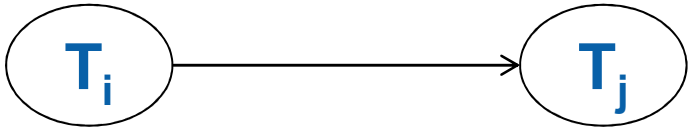
begin
Lock(X)
Lock(Y)
Y=X+1
Unlock(Y)
Unlock(X)
commit

- 有问题吗?

T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(Y)
Read(Y)	
Write(Y)	

证明2PL的可串行性

- 使用优先图可以证明2PL是可串行的



- 如果Ti->Tj，那么
  - 不存在反向边Tj->Ti
  - 并且只可能在Ti进行到commit时解锁阶段，Tj才可能开始
  - 并且Tj的执行不影响Ti的commit
  - 可以认为整个的Ti先于整个的Tj发生
- 环不存在
  - 一条路径T1->T2->T3...->Tk，T1先于T2发生，T2先于T3发生，...，必然有T1先于Tk发生，所以不可能有Tk->T1

实现细节1：读写的锁是不同的

- Shared lock(S)：保护读操作　　（共享锁）
- Exclusive lock(X)：保护写操作　　（互斥锁）

Lock Compatibility Matrix

	Shared Lock(S)	Exclusive Lock(X)
Shared Lock(S)	√	X
Exclusive Lock(X)	X	X

实现细节2： Lock Granularity

- 锁的粒度是不同的
  - Table?
  - Record?
  - Index?
  - Leaf node?
- Intent locks
  - IS(a)：将对a下面更细粒度的数据元素进行读
  - IX(a)：将对a下面更细粒度的数据元素进行写
- 为了得到S,IS: 所有祖先必须为IS或IX
- 为了得到X,IX: 所有祖先必须为IX

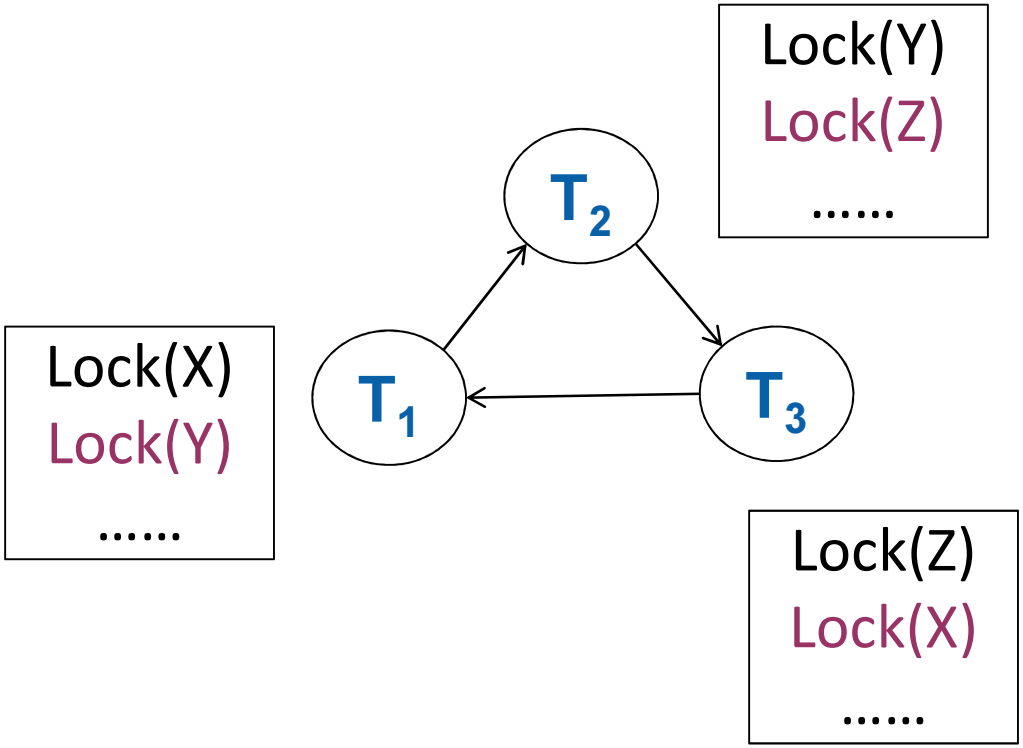
实现细节2： Lock Granularity

Lock Compatibility Matrix

	IS (intent shared)	IX (intent exclusive)	S (shared)	X (exclusive)
IS	√	√	√	X
IX	√	√	X	X
S	√	X	√	X
X	X	X	X	X

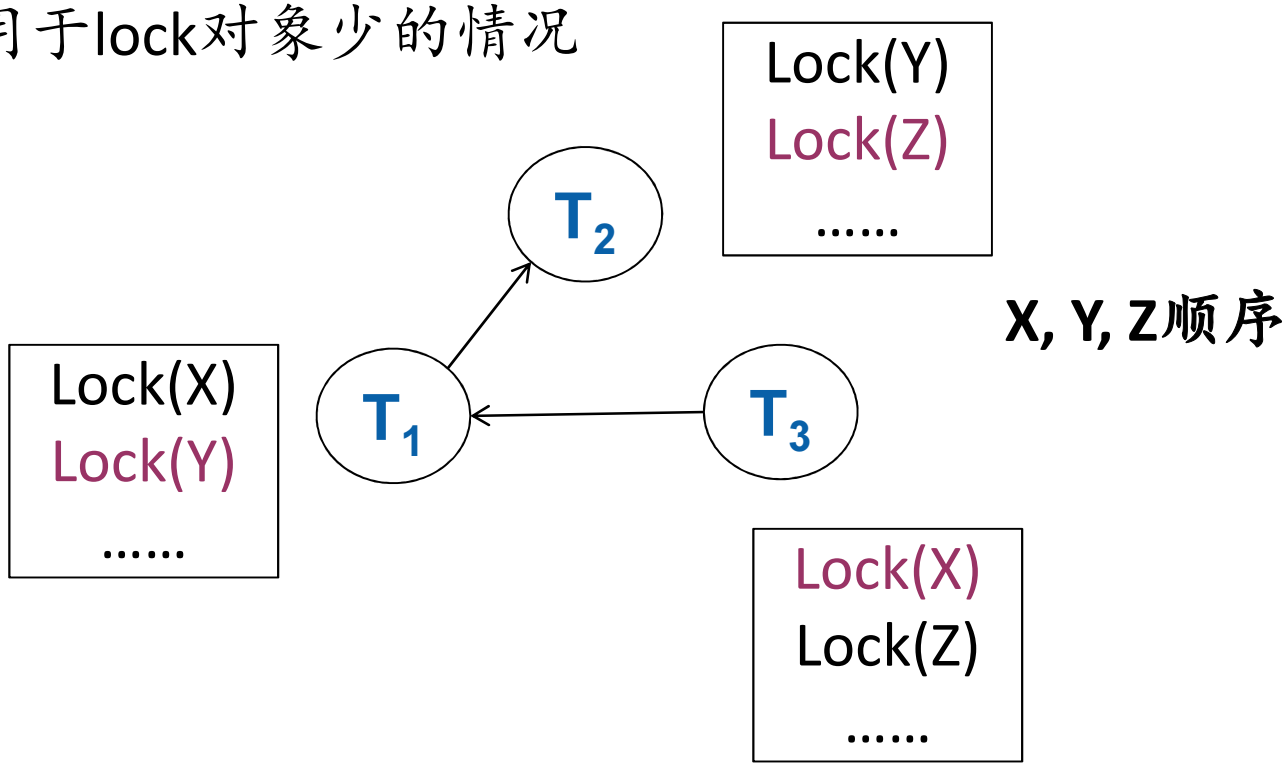
实现细节3: deadlock

- 什么情况下会出现死锁？
  - 最重要的条件：circular wait 循环等待（~~wait in a loop~~?）



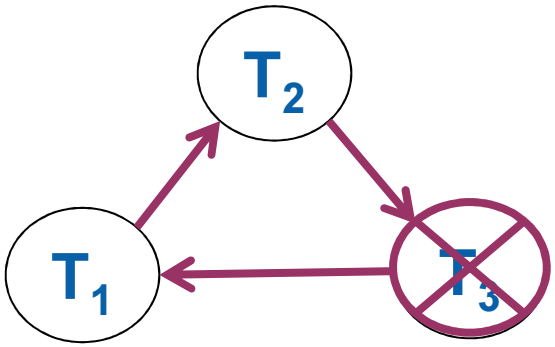
如何解决deadlock问题？

- 死锁避免
  - 规定lock对象的顺序
  - 按照顺序请求lock
  - 适用于lock对象少的情况



如何解决deadlock问题？

- 数据库的lock对象很多，不适合死锁避免
- 死锁检测
  - 周期地对长期等待的Transactions检查是否有circular wait
  - 如果有，那么就选择环上其中一个Transaction abort



加锁管理

- Lock Manager （锁管理器）
  - 维护一个哈希表
  - Key: 数据库中对象的标识
  - Value: 正在拥有这个锁的事务，等待这个锁的事务
- 加锁和解锁操作
  - 修改哈希表中对应的项
  - 必须保证是原子操作，采用底层的同步机制

乐观的并发控制：不采用加锁

- 事务执行分为三个阶段
  - 读：事务开始执行，读数据到私有工作区，并在私有工作区上完成事务的处理请求，完成修改操作
  - 验证：如果事务决定提交，检查事务是否与其它事务冲突
    - 如果存在冲突，那么终止事务，清空私有工作区
    - 重试事务
  - 写：验证通过，没有发现冲突，那么把私有工作区的修改复制到数据库公共数据中

- 优点：当冲突很少时，没有加锁的开销
- 缺点：当冲突很多时，可能不断地重试，浪费大量资源，甚至无法前进

多种乐观并发控制方案

- 具体的读、验证、写的机制不同
- 有多种方案，我们这里介绍
  - Time-stamp ordering基于时间戳的并发控制
  - Snapshot Isolation和MVCC

基于时间戳的并发控制

- 在事务开始时，给事务分配一个时间戳
  - 事务的时间戳都不等
  - 人为定义了一个串行化顺序
  - 要求所有读写冲突顺序必须符合时间戳顺序
- 每个数据库对象有一个读时间戳和一个写时间戳
  - RTS(O)：读时间戳，是读O的事务的时间戳
    - 所有读O的事务中，开始时间戳最晚的那个
  - WTS(O)：写时间戳，是写O的事务的时间戳
    - 所有写O的事务中，开始时间戳最晚的那个

事务T读数据对象O

- TS(T) <WTS(O)
  - 写读冲突
  - 假设最后一次写O的事务是S，TS(S)=WTS(O)
  - 事务T先于事务S，但是读O却晚于S写O
  - 出错了，终止T，重试T

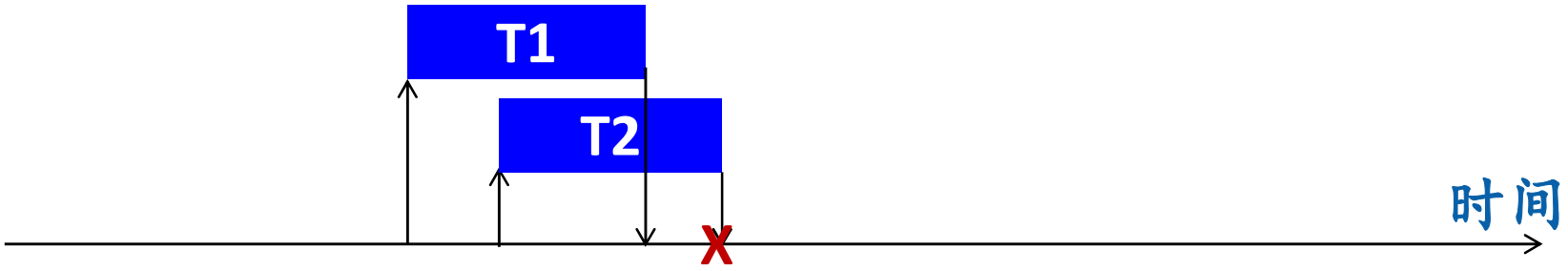
- TS(T)>WTS(O)
  - 正确
  - RTS(O)= max(TS(T), RTS(O))

事务T写数据对象O

- TS(T) < RTS(O)
  - 读写冲突
  - 假设最后一次读O的事务是S，TS(S)=RTS(O)
  - 事务T先于事务S，但是写O却晚于S读O
  - 出错了，终止T，重试T
- TS(T) < WTS(O)
  - 写写冲突
  - 假设最后一次写O的事务是U，TS(U)=WTS(O)
  - 事务T先于事务U，但是写O却晚于U写O
  - 所以，事务T的写会被U的写所覆盖：什么都不做
- TS(T) > RTS(O) and TS(T) > WTS(O)
  - 正确
  - WTS(O)= TS(T)

另一种并发控制方法:Snapshot Isolation

- Snapshot: 一个时点的数据库数据状态
- Transaction
  - 在起始时点的snapshot
  - 读：这个snapshot的数据
  - 写：先临时保存起来，在commit时检查有无冲突，有冲突就abort
    - First writer wins：两个互相竞争的事务，先commit的成功
- Multiversion concurrency control（多版本并发控制）是Snapshot Isolation一个实现



Snapshot Isolation不一定是可串行化的

- 在某些情况下, Snapshot Isolation不是Serializable的

初始： x=10; y=10

要求： x+y>=0
两个账户总和不能透支

T1

```
if (x + y >= 20) {
    x = x – 20;
}
```

T2

```
if (x + y >= 20) {
    y = y – 20;
}
```

结果： x= -10; y= -10

在Snapshot Isolation下，T1与T2可以同时正确执行。
注意：虽然在每个事物中，x+y>0，但总结果却不是了

Outline

- 事务的概念和ACID
- Concurrency Control (并发控制)
- Crash Recovery (崩溃恢复)
  - WAL
  - 崩溃恢复

Durability (持久性) 如何实现？

- Transaction commit后，结果持久有效，crash不消失

- 想法一
  - 在transaction commit时，把所有的修改都写回硬盘
  - 只有当写硬盘完成后，才commit

- 有什么问题？
  - 正确性问题：如果写多个page，中间掉电，怎么办？
    Atomicity被破坏了！
  - 性能问题：随机写硬盘，等待写完成

解决方案：WAL (Write Ahead Logging)

- 什么是Logging
- 什么是Write-Ahead
- 怎样保证Durability
- 怎么实现Write-Ahead Logging
- Crash Recovery

经典算法：ARIES

什么是Transactional Logging(事务日志)

- 事务日志记录(Transactional Log Record)
  - 记录一个写操作的全部信息

- 例如：记录的修改操作的日志记录
  - LSN: Log sequence number，是一个不断递增的整数，唯一代表一个记录；每产生一个日志记录，LSN加1
  - tID: transaction ID
  - prevLSN: 这个事务前一个日志的LSN
  - type: 写操作类型
  - pageID, slotID, columnID：定位到具体一个页的一个记录的一个列
  - old value, new value：旧值和新值

什么是Transactional Logging(事务日志)

- 对Transaction中每个写操作产生一个事务日志记录
- Transaction commit会产生一个commit日志记录
  - (LSN, tID, commit)
- Transaction abort会产生一个abort日志记录
  - (LSN, tID, abort)
- **日志记录被追加(append)到日志文件末尾**
  - 日志文件是一个append-only的文件
  - 文件中日志按照LSN顺序添加

什么是Write-Ahead Logging?

- Write-Ahead
  - Logging 总是先于实际的操作
  - Logging 相当于意向，先记录意向，然后再实际操作
- 写操作
  - 先Logging
  - 然后执行写操作
- Commit
  - 先记录commit 日志记录
  - 然后commit

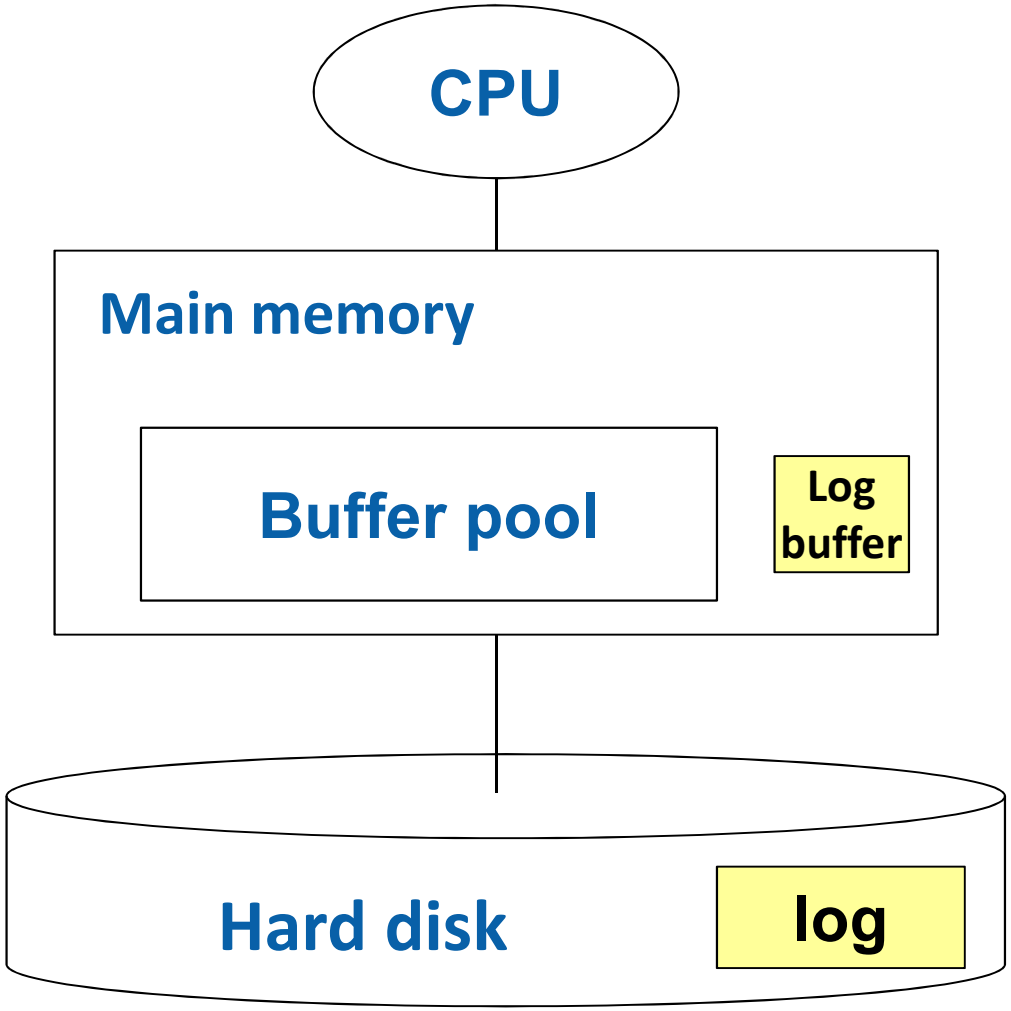
WAL怎样保证Durability？

- 条件：日志是Durable的
- 当出现掉电时，可以根据日志发现所有写操作
  - 总是先记录意向，然后实际操作
  - 所以只有存在日志记录，相应的操作才有可能发生
- 对于一个Transaction，寻找它的commit日志记录
  - 如果找到，那么这个transaction 已经commit了
  - 如果没找到，那么这个transaction 没有完成
- 已Commit
  - 根据日志记录，确保所有的写操作都完成了
- 没有commit
  - 根据日志记录，对每个写操作检查和恢复原值

如何保证日志Durable?

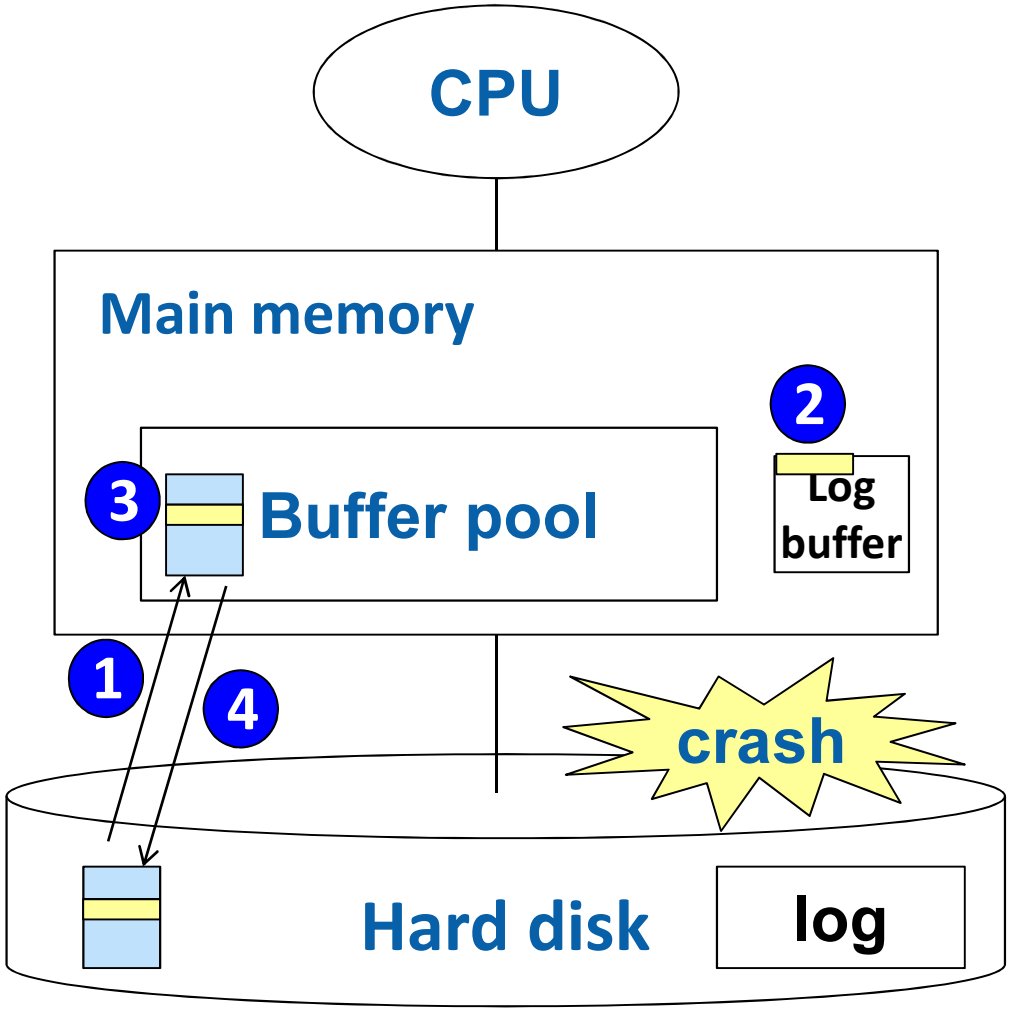
- 简单方法：写日志记录时保证日志记录Durable
  - write + flush
  - 必须执行一个写操作，然后用flush保证写操作确实写到硬盘上了，并且等待flush结束
  - 这个过程通常需要~10ms
- 简单计算
  - 假设每个transaction需要修改10个记录
  - 那么上述写日志就需要~100ms
  - 硬盘同时只能执行一个写操作
  - 所以系统的throughput为10 transaction/second
    - 如果每个transaction修改100个记录，会怎么样？
- 太慢了！

实现： WAL (Write Ahead Logging)



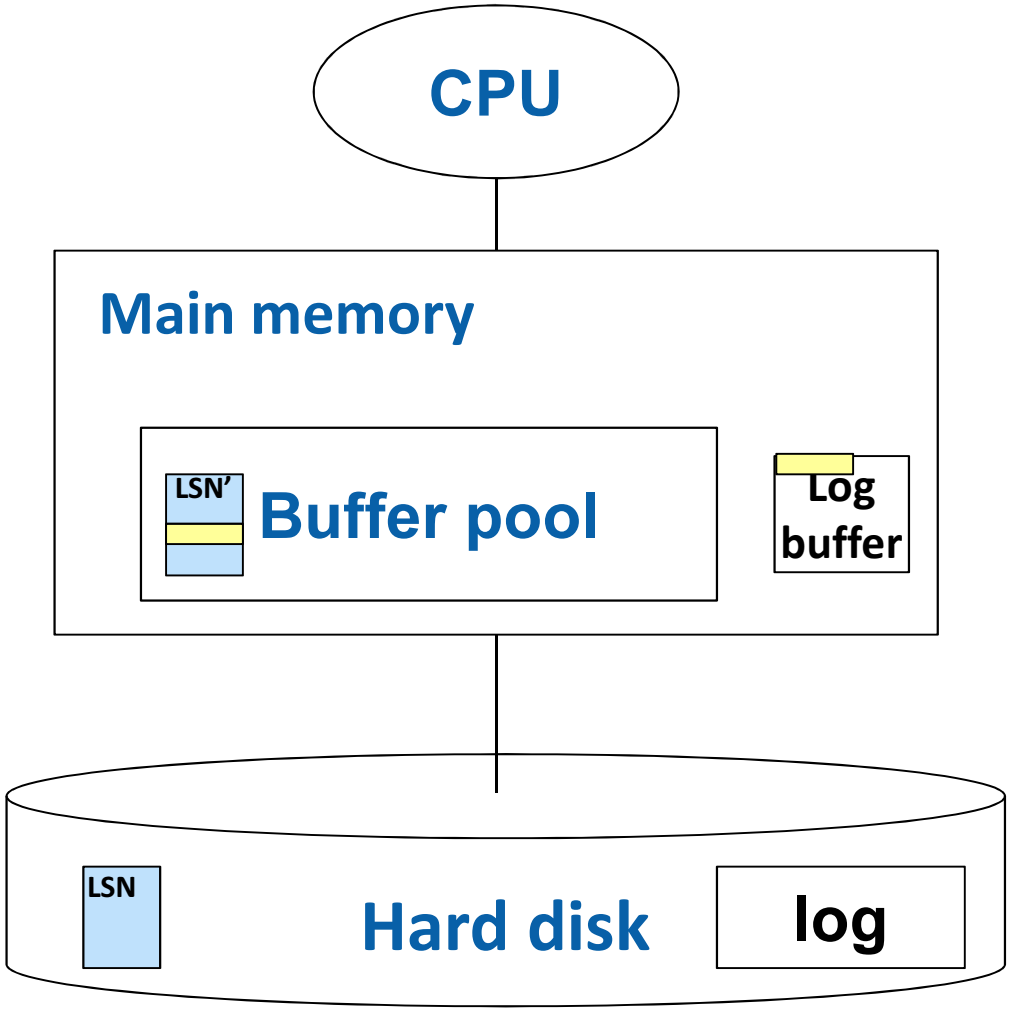
- **Log**: 硬盘上日志文件
- **Log buffer**：在内存中分配一个缓冲区

实现：WAL (Write Ahead Logging)



- 日志写在log buffer中
- 当commit时write+flush log buffer
- 这样性能好了，但有什么问题？
  - Dirty page可能被写回硬盘！
  - 掉电后，硬盘上数据已经修改，但是log没有记录！

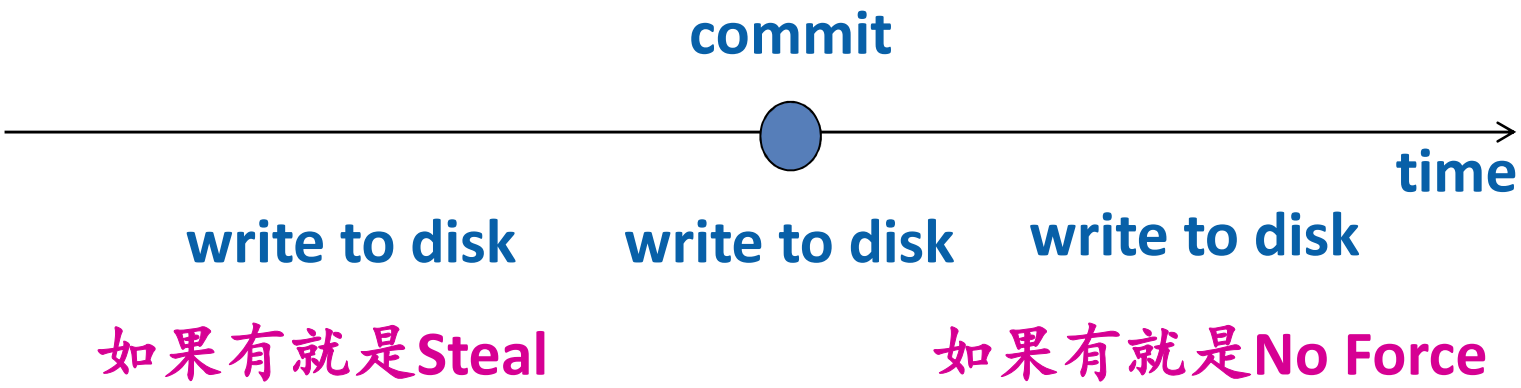
实现：WAL (Write Ahead Logging)



- 日志写在log buffer中
- 当commit时write+flush log buffer
- 解决方法：
  - Page header记录本page 最新写的LSN
  - Buffer pool在替代写回一个dirty page时，必须保证page LSN之前的所有日志已经flush过了
- **保证：日志记录一定是先于修改后的数据出现在硬盘上**

缓冲池策略比较

- Force vs. No Force
  - Force: 在事务commit时，把修改过的数据页都flush到硬盘
- Steal vs. No Steal
  - No Steal: 在事务commit前，修改过的数据页必须缓冲，不能成为victim被替换



缓冲池策略与Logging的类型

	Force	No Force
Steal	修改的数据页在commit前可能写出，在commit时一定写出 undo logging	修改的数据页在commit前可能写出，在commit时不一定写出 undo/redo logging
No Steal	修改的数据页在commit前不会写出，在commit时一定写出 不需要logging，但是性能代价极大	修改的数据页在commit前不会写出，在commit时不一定写出 redo logging

最通用

日志记录的种类

- Logical log
  - 记录大的操作: insert, delete, update
- Physical log
  - 记录修改前和修改后的整个数据页
- Physiological log
  - 记录被修改的页号（Physical）
  - 页内部记录修改的操作（Logical）
    - insert, delete, update
    - 旧值，新值
- 通常采用Physiological log（我们这里也是如此）

Checkpoint（检查点）

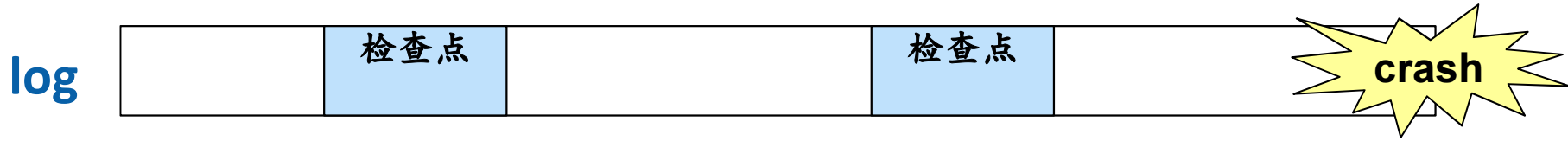
- 为什么要用checkpoint?
  - 为了使崩溃恢复的时间可控
  - 如果没有checkpoint，可能需要读整个日志，redo/undo很多工作
- 定期执行checkpoint
- checkpoint的内容
  - 当前活动的事务表：包括事务的最新日志的LSN
  - 当前脏页表：每个页最早的尚未写回硬盘的LSN

Log Truncation

- Log file不能无限地增长
- 什么情况下一个日志记录不需要了？
  - 对应的transaction完成了
  - 对应的写操作已经在硬盘上了
- 如果LSN之前的所有日志记录都不需要了，那么就可以删除LSN之前的Log – log truncation
  - Hot page 问题：一个page经常被更新，总是在buffer pool中，很长时间也不写回硬盘，而硬盘上对应的page很长时间没有更新，使得log truncation 难以进行
  - 定期地把长时间缓存在buffer pool中的dirty page写回（例如，在检查点时做）
  - LSN= min(脏页最早的尚未写回硬盘的LSN)
    - 这个LSN之前的所有日志都可以丢弃

Crash Recovery

- 系统定期把当前活跃的Transaction信息(tID, earliest LSN)记录在log中



- Crash后重新启动
- ARIES算法
  - 分析阶段
  - redo阶段
  - undo阶段

崩溃恢复：分析阶段

- 找到最后一个检查点
  - 检查点的位置记录在硬盘上一个特定文件中
  - 读这个文件，可以得知最后一个检查点的位置
- 找到日志崩溃点
  - 如果是掉电等故障，必须找到日志的崩溃点
  - 当日志是循环写时，需要从检查点扫描日志，检查每个日志页的校验码，发现校验码出错的位置，或者LSN变小的位置
- 确定崩溃时的活跃事务和脏页
  - 最后一个检查点时的活跃事务表和脏页表
  - 正向扫描日志，遇到commit, rollback, begin更新事务表
    - 同时记录每个活动事务的最新LSN
  - 遇到写更新脏页表
    - 同时记录每个页的最早尚未写回硬盘的LSN

崩溃恢复：Redo阶段

- 目标：把系统恢复到崩溃前瞬间的状态
- 找到所有脏页的最早的LSN
- 从这个LSN向日志尾正向读日志
  - Redo每个日志修改记录
- 对于一个日志记录
  - 如果其涉及的页不在脏页表中，那么跳过
  - 如果数据页的LSN>=日志的LSN，那么跳过
    - 数据页已经包含了这个修改
  - 其它情况，修改数据页

崩溃恢复：Undo阶段

- 目标：清除未提交的事务的修改
- 对于所有在崩溃时活跃的事务
  - 找到这个事务最新的LSN
  - 通过反向链表，读这个事务的所有日志记录
- undo所有未提交事务的修改
  - Undo时，比较数据页的LSN和日志的LSN
  - if (数据页LSN>=日志LSN) 时，才进行undo

介质故障的恢复

- 如果硬盘坏了，那么日志可能也损坏了
  - 无法正常恢复
- 硬件的方法：RAID
- 如果整个RAID坏了，怎么办？
- 需要定期replicate备份数据库
  - 备份数据库数据
  - 更频繁地备份事务日志
  - 那么就可以根据数据和日志恢复数据库状态
  - 例如：双机系统

Outline

- 事务的概念和ACID
- Concurrency Control (并发控制)
  - 数据冲突和可串行化
  - 加锁的并发控制
  - 乐观的并发控制
- Crash Recovery (崩溃恢复)
  - WAL
  - 崩溃恢复