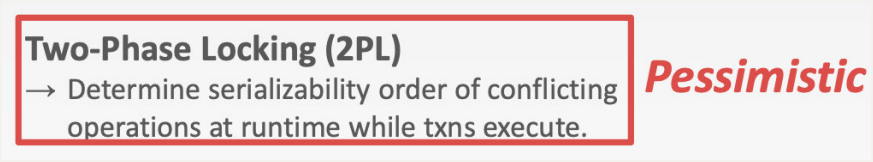
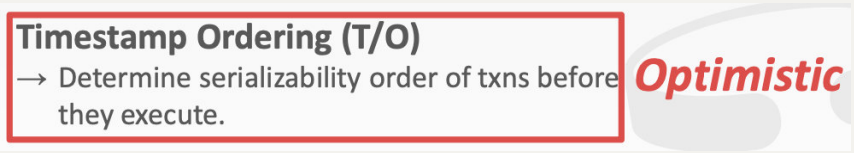
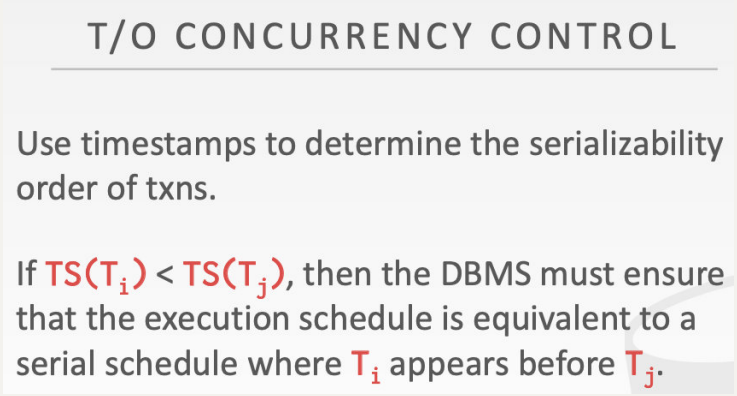
本Lecture继续探索并发控制理论的具体实现，上一个Lecture介绍了二阶段锁这一种实现方式，二阶段锁协议有一个不可避免的缺点：锁的存在会导致并行性的下降，进而影响性能。二阶段锁协议属于一种悲观的并发控制方法：总是假设未来可能出现数据竞争，因此总是给共享的对象上锁



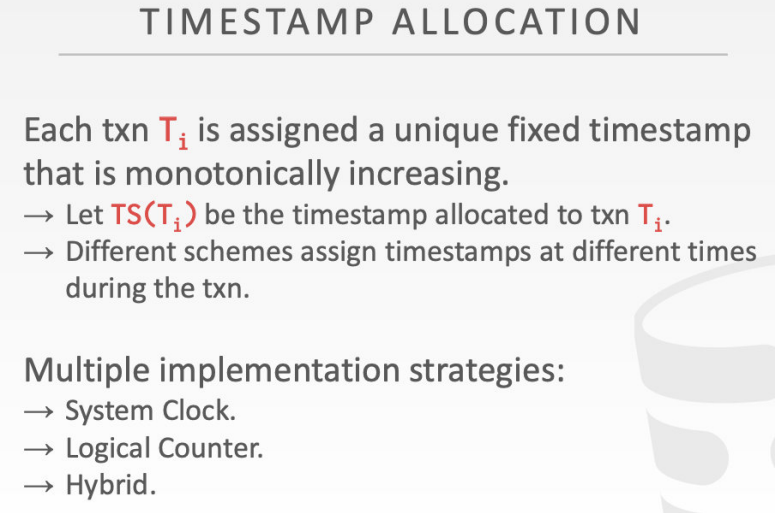
与之相对的也存在一些乐观的并发控制方法，比如说接下来要介绍的基于时间戳顺序的并发控制（Timestamp Ordering），它的基本原理是，给每个事务一个时间戳，根据事务的时间戳来决定它们的顺序以及出现冲突操作时该如何处理



如果事务Ti的时间戳在Tj之前，那么DBMS要保证这两个事务都被提交之后的效果相当于Ti先执行，Ti执行完了之后Tj再执行



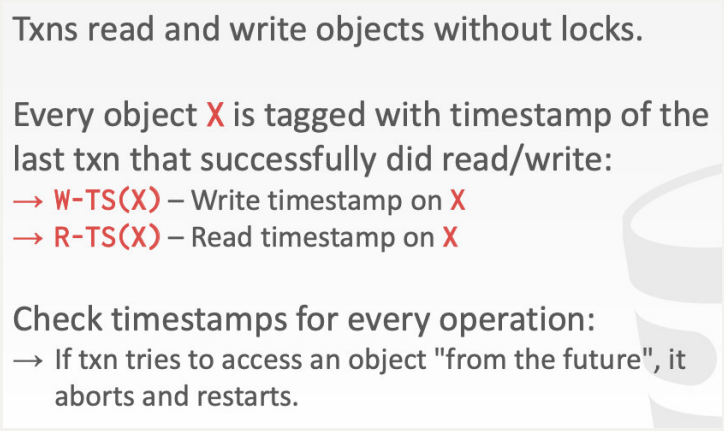
首先要解决的问题是，事务的时间戳是怎么设置的呢？



- 可以通过系统时钟，但有些时候这会出问题，因为系统时钟不是完全精确的，它会每隔一段时间和服务器通信，进行同步，因此在同步的时候就会出现“时间突然被调慢了一分钟”这种情况，这就可能导致系统时间校准之前到达的事务和系统时间校准的之后到达的事务的时间戳顺序发生了混乱
- 由于系统时钟并不完全可靠，因此在很多单节点的数据库中使用Logical Counter（逻辑计数器），第一个到达的事务标记为1号，第二个到达的标记为2号...，通过简单的计数完成时间戳的排列，从而避免系统时钟的跳变
- 对于分布式系统来说，Logical Counter很难校准（因为在各个节点之后依赖于网络的通讯比较慢，如果A节点和B节点相距很远的话，有可能某一时刻到达A节点的事务和到达B节点的事务对应同一个counter，因为它们之间不会立即获取到对方的counter更新）。因此就有了系统时钟和逻辑计数器混合使用的时间戳确定办法，具体的实现方法在本课程中并未提及

基础的T/O协议

Basic Timestamp Ordering(T/O) Protocol

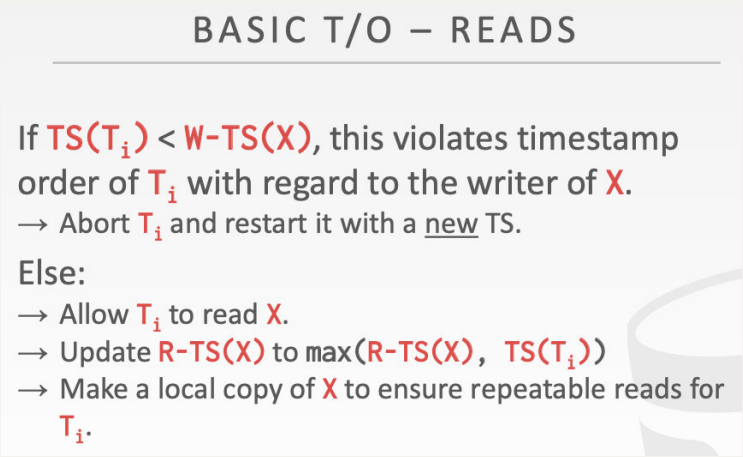


基础的T/O协议中，事务在读/写对象的时候是不加锁的

数据库中的所有对象（一般就是指tuple这种对象）上面要附带两个时间戳，一个读时间戳（上一次读这个对象的事务的时间戳/事务号），一个写时间戳（上一次写这个对象的事务的时间戳/事务号）

每次读/写数据的时候都要检查时间戳（比较当前事务的时间戳和操作过当前正在进行读/写的对象的事务的时间戳），要求是“不能操作未来的数据”

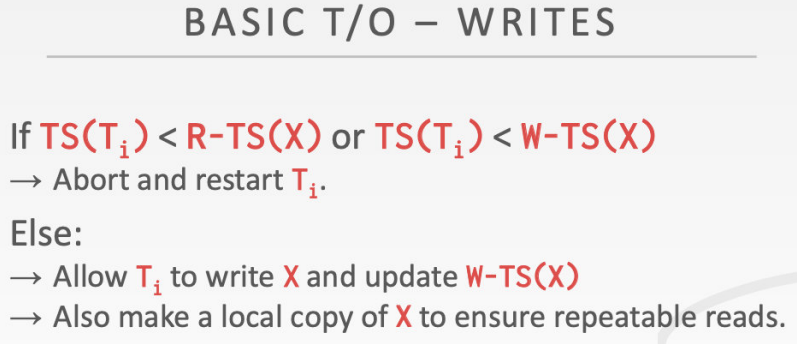
如何实现“不能操作未来的数据”呢？首先分析事务读对象的流程：



如果发现当前事务（上图中的Ti）的时间戳比要读的对象的写时间戳（上次修改它的事务的时间戳）早，这就属于“操作来自未来的数据”了，无法满足“执行调度的结果等效于先到达的事务完整地在后到达的事务之前发生”，那么当前的Ti事务就不能再操作这个共享的对象，于是abort，之后系统给这个abort的事务一个新的时间戳，事务重启。

否则就是合法的，当前事务可以读共享的对象，读完了之后更新这个对象的读时间戳（如果这个对象的读时间戳比当前事务的时间戳早，那就把读时间戳修改成当前事务的时间戳），然后把对象从数据库拷贝一份到事务本地（因为对象可能会在接下来被其他事务修改）

事物写对象的流程如下：

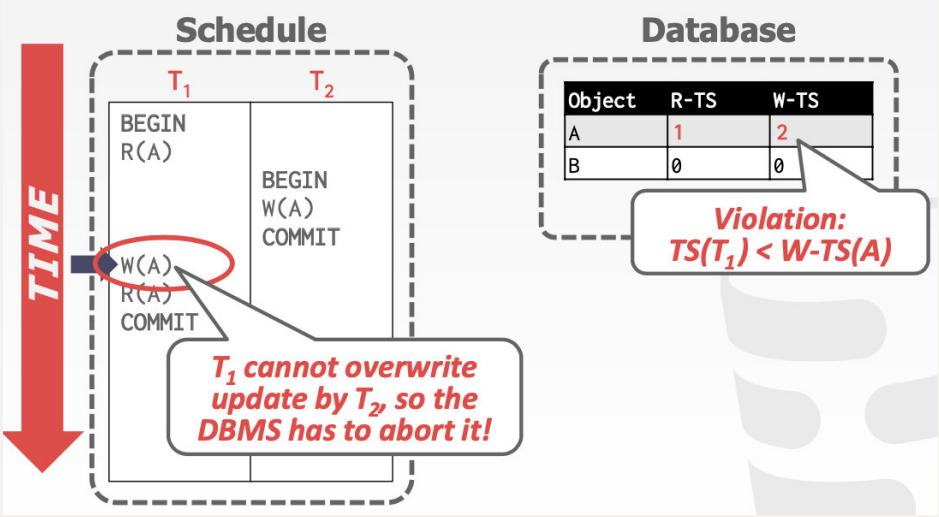


如果当前事物的时间戳早于共享对象的读时间戳或者当前事物的时间戳早于共享对象的写时间戳，那么当前事物abort并重启（因为不能更新未来的数据）

否则（也就是共享对象的上一次读或写都由该在事物之前到达的事物完成）就可以进行对共享对象的写操作，并且将其拷贝一份放在本地以便可重复读

基础的T/O协议执行的过程可以结合slides中的例子来理解（

在出现abort情况的Example 2中



如果T1的W(A)不写入数据库，而是只写在事务本地，随后的R(A)也只是读本地的值，这样的话，这个执行调度的结果就和串行化后的结果是一样的，而且T1不用abort，这个优化方法就是托马斯写规则

THOMAS WRITE RULE

If $TS(T_i) < R\text{-}TS(X)$:
→ Abort and restart T_i.

If $TS(T_i) < W\text{-}TS(X)$:
→ **Thomas Write Rule**: Ignore the write to allow the txn to continue executing without aborting.
→ This violates timestamp order of T_i.

Else:
→ Allow T_i to write X and update $W\text{-}TS(X)$

如果当前事物写共享对象的时候，它的时间戳早于这个对象的读时间戳，那么就表明：未来有事务读取这个对象当前的值，那就不能让当前事物进行对这个共享对象的写操作，因此需要abort当前事务；但是如果仅仅是当前事务的时间戳早于共享对象的写时间戳，这就表明未来有事务会修改这个共享对象的值，那么其实当前事务不对这个对象进行写操作也没事，因为这等效于当前事务完成了写操作但写的结果在未来被覆盖了

如果不考虑托马斯写规则带来的优化，基础T/O协议会生成冲突可串行化的执行调度，它的优点是没有采用锁，因此不可能构成死锁，但也有一个问题：较长的事务（比如说有几百条SQL语句）有可能会饥饿，因为很有可能它执行了一段时间之后，想要访问的数据都是被比它更“年轻”的事物修改过的，那它只能abort，重启之后又迎来同样的结局...

BASIC T/O

Generates a schedule that is conflict serializable if you do **not** use the Thomas Write Rule.
→ No deadlocks because no txn ever waits.
→ Possibility of starvation for long txns if short txns keep causing conflicts.

Permits schedules that are not recoverable...

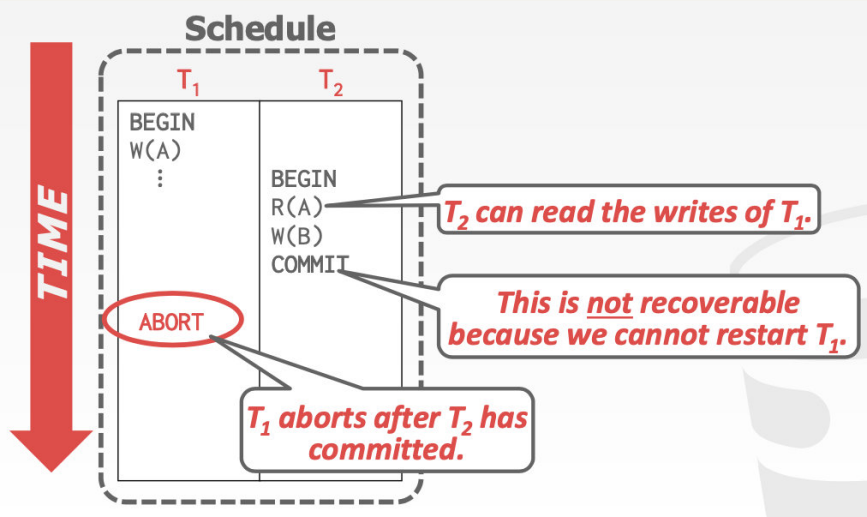
RECOVERABLE SCHEDULES

A schedule is **recoverable** if txns commit only after all txns whose changes they read, commit.

Otherwise, the DBMS cannot guarantee that txns read data that will be restored after recovering from a crash.

并且采用基础T/O协议的执行调度是不可恢复的，这里涉及到了另外的一个概念：“可恢复的执行调度”，它的意思是说，事务在修改共享对象时，都是在其他事务已提交对共享对象的更新的基础上去修改。如果不能保证修改共享对象时都是在更新已提交的基础上去修改，那DBMS很难保证在崩溃后恢复时数据的一致性

这段话过于抽象，举个例子具体分析，如下所示

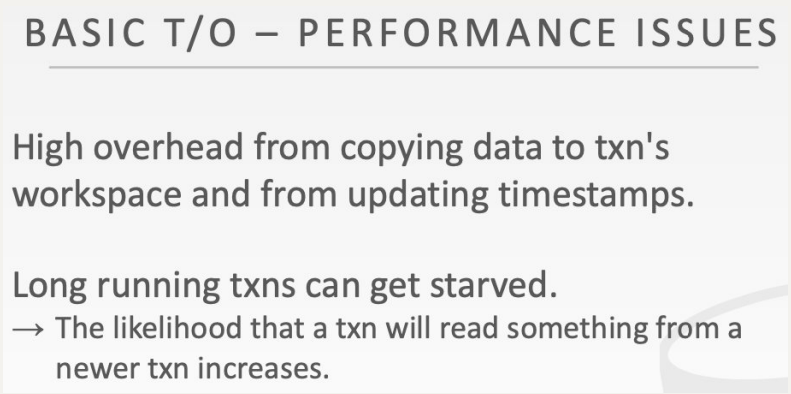


上图这个执行调度就是不可恢复的：

T2执行W(B)后立即提交，并且T2提交时T1仍处于活跃状态，但T1在提交前发生了故障。T2已经读取了T1写入的数据A的值，因此，我们必须终止T2以保证事务的原子性，但T2已经提交，不能再中止。这样就出现了T1发生故障之后不能正确恢复的情形。

基础T/O协议下的执行调度里，很有可能一个长事务执行半天然后abort了,短事务都commit很多了，就像上面的场景一样，这种执行调度是不可恢复的

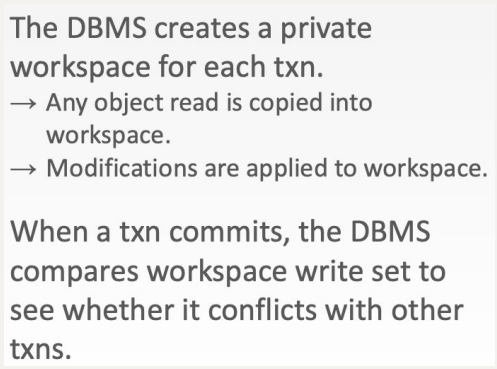
基础的T/O协议还有一些性能问题，事务在读任何数据的时候都要往本地拷贝一份，这会带来不小的开销（比如说全表扫描的时候就要把整个表复制一份）；以及还有长事务容易饥饿的问题



OCC

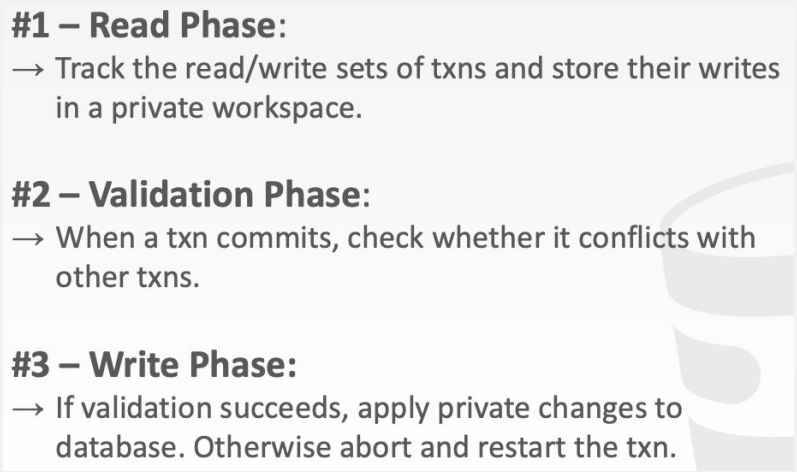
Optimistic Concurrency Control

OCC是基于事务之间发生冲突的概率低，并且事务都比较短这个假设提出的优化策略，它希望通过无锁化来对事务之间无冲突的场景做出优化



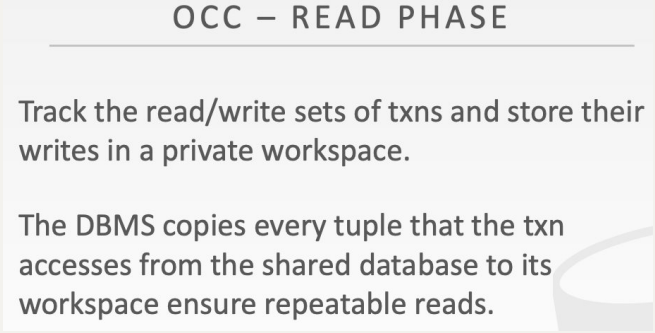
OCC策略的概述如上图：每个事务都有一个本地保存临时数据的workspace，读的所有的数据都要保存在本地，对数据的修改不会写入数据库中，而是只写到本地。等到事务即将commit的时候，DBMS会将要提交的数据更新和其他事务要提交的更新进行比较，如果它们之间没有冲突的话，DBMS会一次性地把事务所做的所有更新都提交，如果出现了冲突，那么事务会abort

OCC有如下三个阶段：



- 读阶段

  进行SQL语句所描述的读写操作。这个阶段之所以称为读阶段，是对于数据库来说的它是一直只被读的：因为OCC策略下，事务想写数据时不会直接写到数据库里，而是写在本地，因此数据库中的数据就是一直在被读



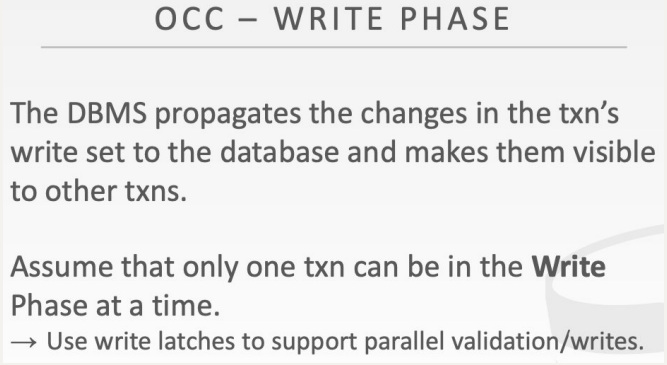
- 校验阶段

  事务执行完了准备提交时，把它完成的数据更新和其他事务相比较。如果校验通过，那么就转至下一个阶段，否则abort并重启事务

- 写阶段

  把事务本地所记录的更新提交到数据库中

  实现OCC的DBMS一般会在写阶段锁全表，除了当前事务以外没有其他线程可以修改数据库，也就不能多个事务并发地写，虽然这牺牲了并发性，但由于前面已经准备好了要写的数据，所以写操作的时间并不长，因此开销可以接受



OCC策略下数据库中的对象只有写时间戳，先进入校验阶段的事务会先拿到较早的时间戳，在进入校验阶段之前事务都没有自己的时间戳（可以结合slides中的例子来理解）

校验阶段做的事情很复杂，接下来详细介绍

当事务调用commit的时候就进入了校验阶段，然后会给它分配一个时间戳，校验时DBMS要保证serializable的原则：执行调度的结果等效于时间戳早的事务在时间戳晚的事务之前串行发生，因此它会检查RW冲突与WW冲突，判断相应的Dependency Graph有没有成环

When txn T_i invokes **COMMIT**, the DBMS checks if it conflicts with other txns.

→ The DBMS needs to guarantee only serializable schedules are permitted.

→ Checks other txns for RW and WW conflicts and ensure that conflicts are in one direction (e.g., older→younger).

Two methods for this phase:

→ Backward Validation

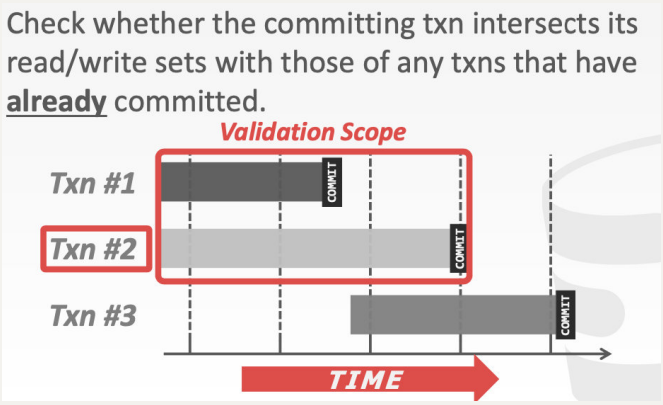
→ Forward Validation

OCC校验阶段的具体实现方式有两种

- 向后校验

  即向已经发生过的事务校验

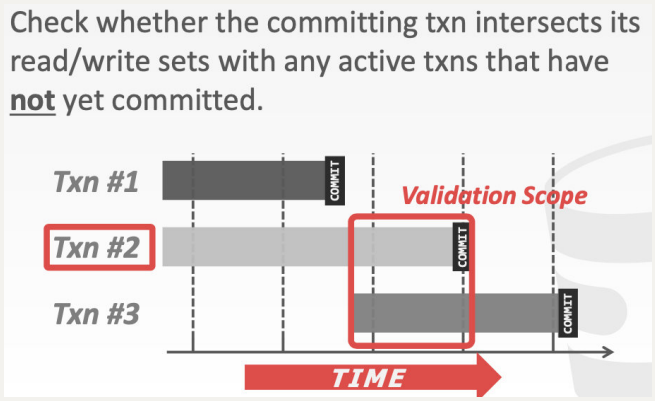
  对于下图事务2来说，它走到commit这一步的时候，需要向已经commit过的事务校验，因此会判断事务2和事务1之间的冲突操作是否成环，如果成环了的话，那么事物2 abort



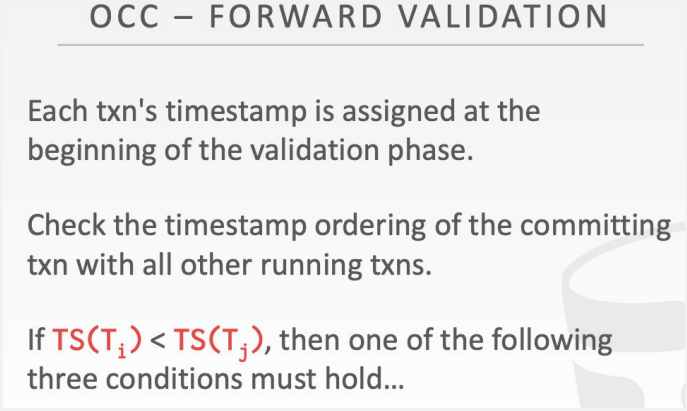
- 向前校验

  即向未来的事务校验

  还是同样的例子，事务2提交的时候，事务3还没有执行完，因此只会校验事务2和事务3有重叠的部分

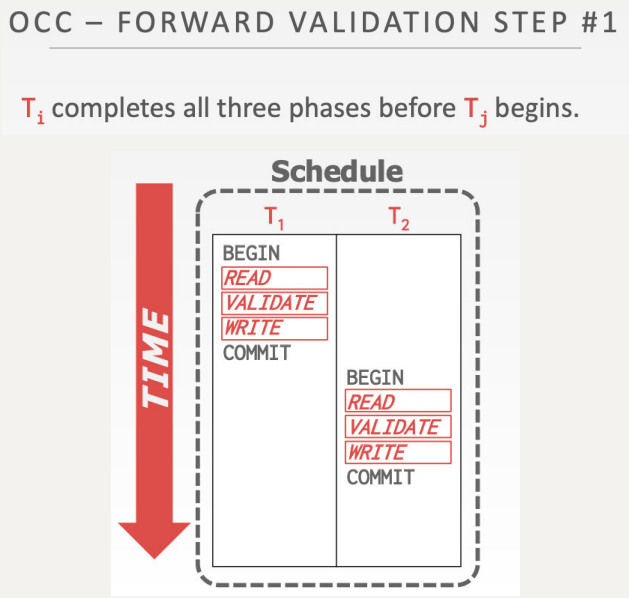


如果校验的那一部分中出现了成环的冲突操作，由于事务2和事务3（即当前要commit的事务和未来要commit的事务）还都没commit，因此可以根据情况选择它们中的一个来abort

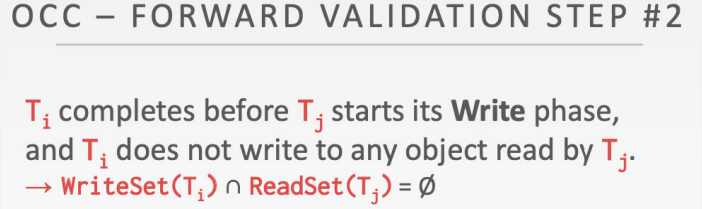


向前校验时有三种可以通过校验的情况（基于TS(Ti)<TS(Tj)），具备其中之一才能校验通过：

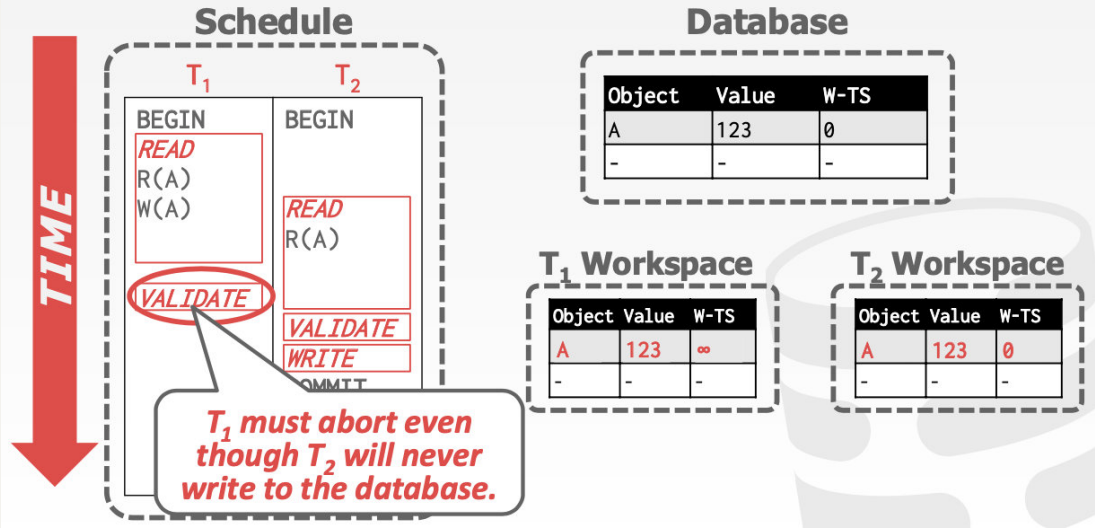
- 执行调度本身就是串行的



- Ti在Tj进入写阶段之前完成，并且Ti要修改的那些对象没有被Tj读过



如下所示的例子就是不满足这个条件的，



- Ti先于Tj完成读阶段

  Ti修改过的对象Tj没读过，Ti修改过的对象Tj没修改过

  OCC – FORWARD VALIDATION STEP #3

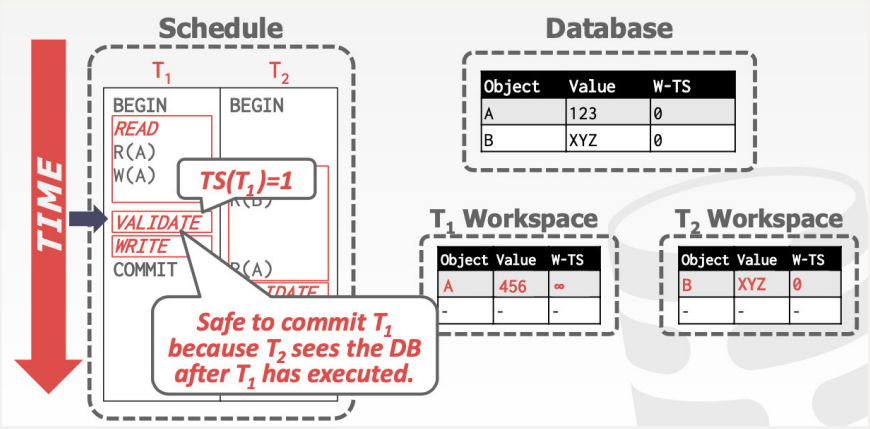
  T_i completes its **Read** phase before T_j completes its **Read** phase

  And T_i does not write to any object that is either read or written by T_j:

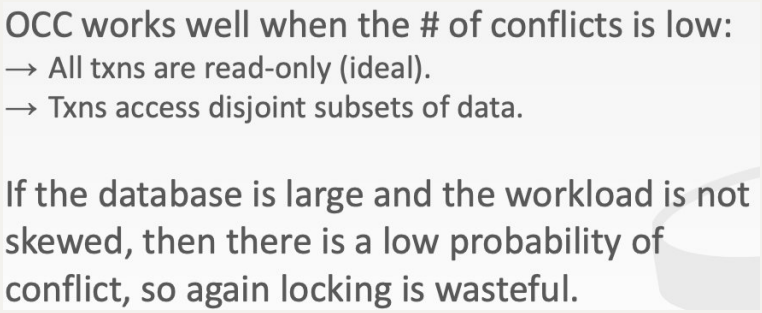
  → $WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$

  → $WriteSet(T_i) \cap WriteSet(T_j) = \emptyset$

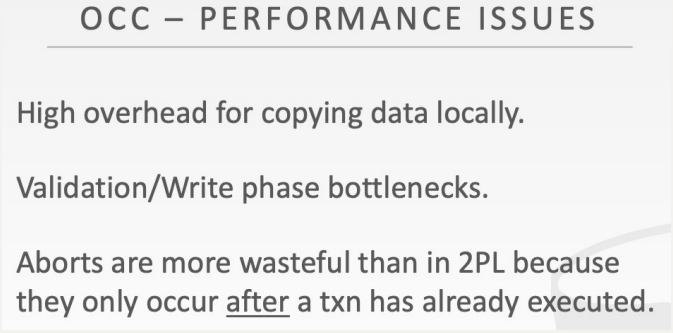
  结合下面的例子理解：



总结一下，OCC更适用于事务之间冲突不多的场景（最理想的情况下是所有事务都只读不写或者不同的事务访问的对象没有交集，这样的话性能远超二阶段锁）。如果数据库很大，事务里的查询是非倾斜的（也就是说没有特别热点的数据），出现事务间冲突的概率就比较小，可以考虑OCC，否则就会出现频繁的事务回滚，性能下降。

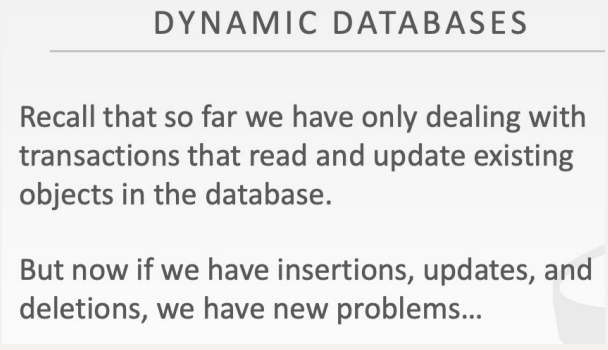


但是OCC也有一些性能问题，它还是会和基础T/O一样要求事务把要读写的数据拷贝到本地，并且校验这个步骤的逻辑很复杂，容易构成性能瓶颈，以及OCC的写阶段是串行的，不能并发，也容易成为瓶颈。OCC中如果校验阶段失败的话，那么前面所做的全部操作就都会前功尽弃，不像二阶段锁有死锁检测，这样可以在事务进行到一半的时候abort



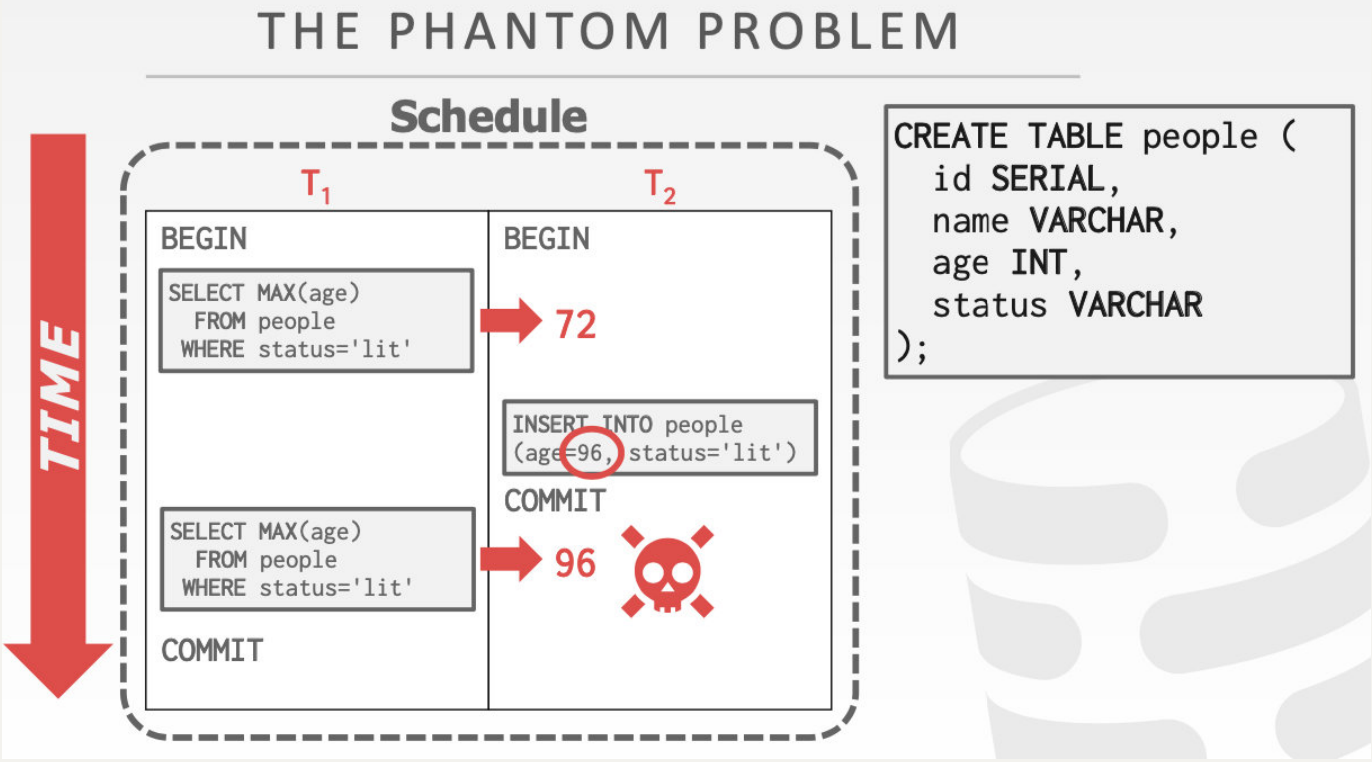
隔离级别

Isolation Levels



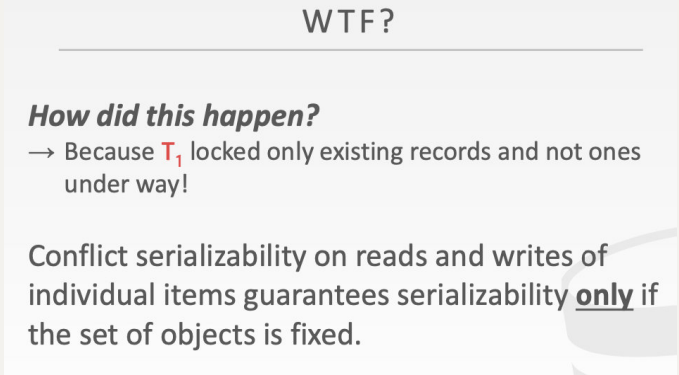
二阶段锁和基础T/O以及OCC其实都没有处理insert/delete时的并发问题

举个例子，有如下的场景：

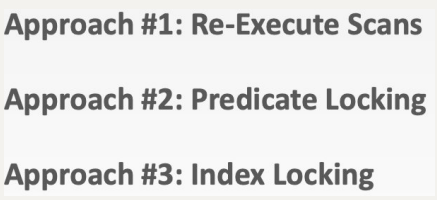


这就是幻读，第二次读的时候读到了第一次读的时候不存在的东西

之所以二阶段锁和OCC这些都预防不了幻读，是因为二阶段锁只是给已经存在了的对象（比如说tuple）上了锁，其他策略也都是只考虑已经存在的对象

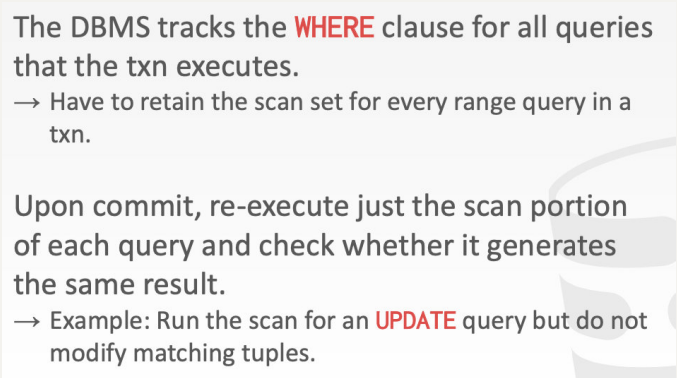


幻读问题有如下三种解决方案



- Re-Execute Scans

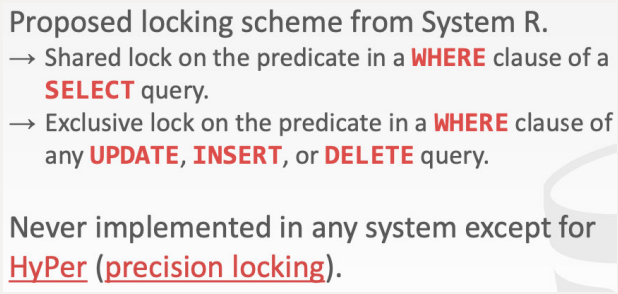
记录下来事务所有可能出现幻读的地方（像查最大值，平均值，最小值这些涉及到范围扫描的操作），为了防止察觉不到有其他事务在扫描完成后再向表里插入新数据，在事务提交之前会再执行一遍前面所记录的所有扫描



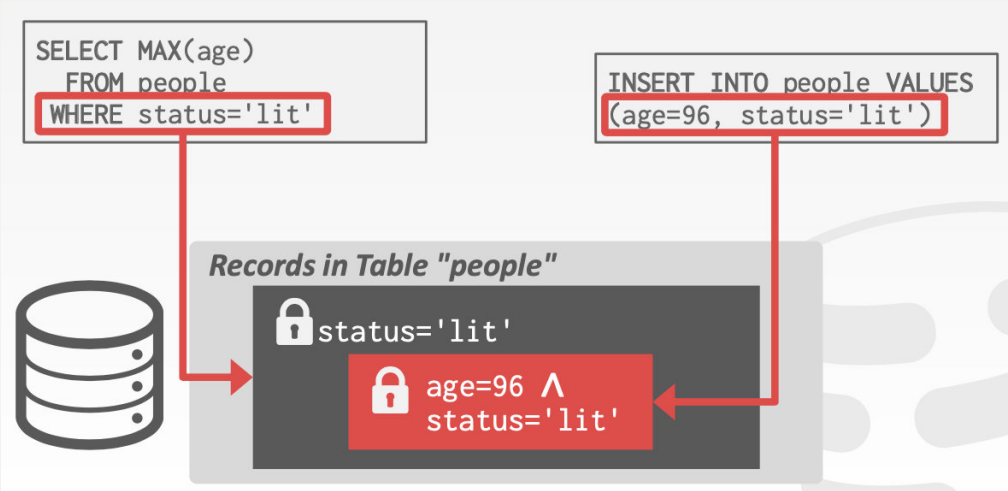
这显然性能不太好，属于简单粗暴的方法

- Predicate Locking

  谓词锁

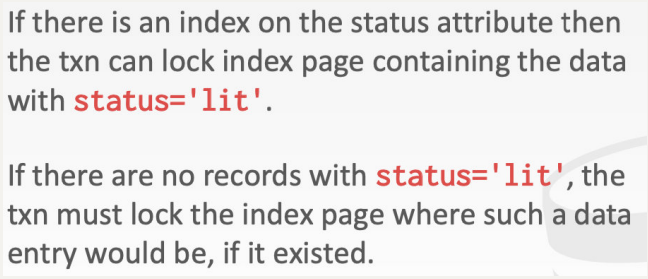


对于含有where clause的SQL语句：给select语句对应的谓词加共享锁，给update/insert/delete语句对应的谓词加独占锁



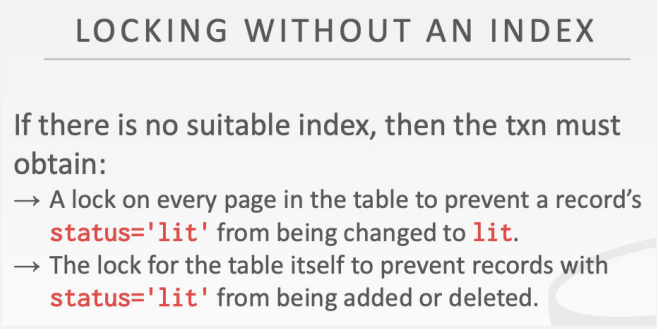
- Index Locking

  索引锁

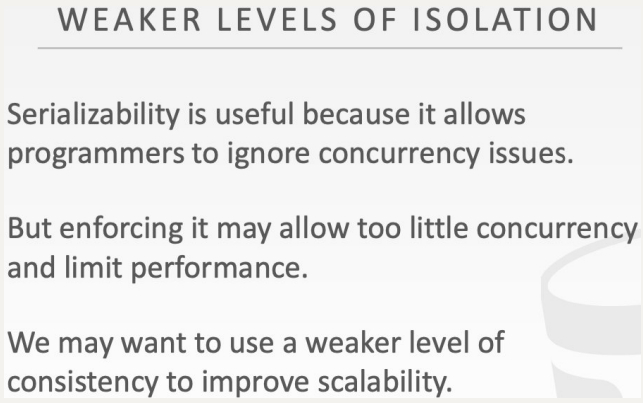


如果对谓词里的attribute已经构建了索引的话，那么DBMS可以给相关的索引页上锁

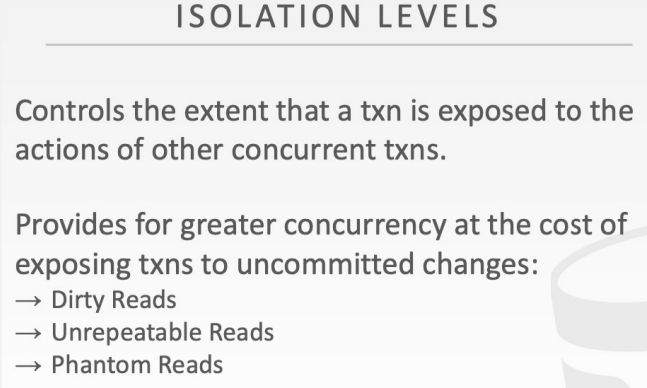
如果没有构建相关的索引，就使用表锁这种宽范围的锁



MySQL为了防止幻读实现了间隙锁：MySQL把索引里的内容分成了数据和间隙，也就是说表里同一列连续的数据之间存在间隙，在进行范围扫描的时候会同时锁数据和间隙，这样的话间隙里就不能插入数据（比如说事务里前面有关于最大值的查询，那么就可以在这个查询结束之后把索引中最大值后面的间隙锁住，这样就不会插入更大的值），这和索引锁类似

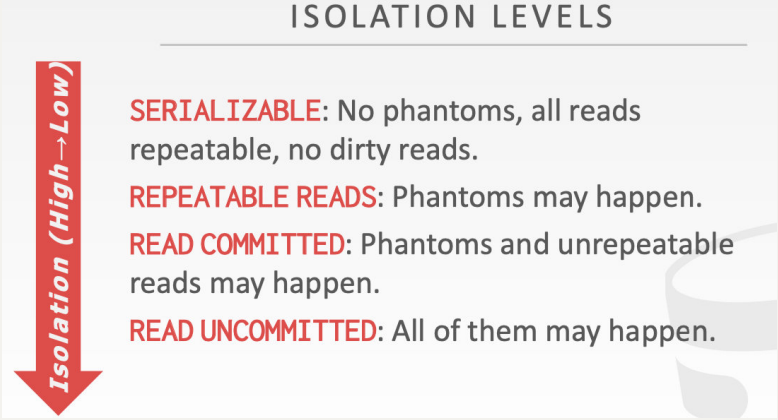


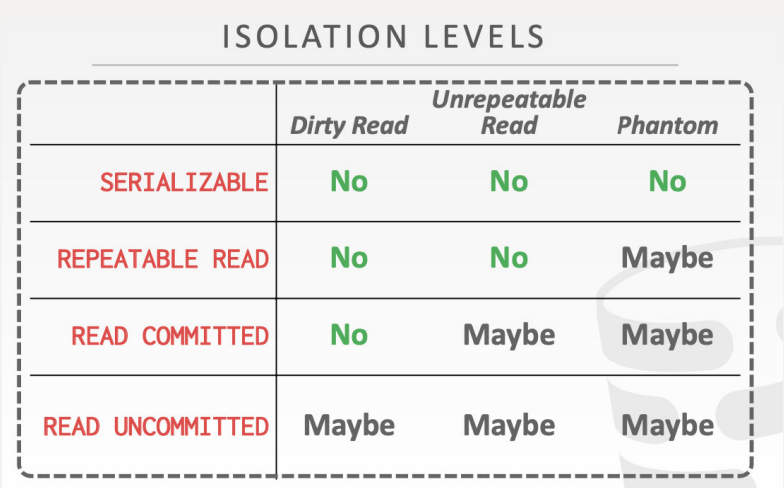
前面从2PL到T/O这些策略都是为了实现串行化这样一个最高的事务隔离级别，实际的业务场景里有些业务可以忍受非串行化的执行调度，而且非串行化时性能会更好，因此也存在一些比串行化更弱的隔离级别



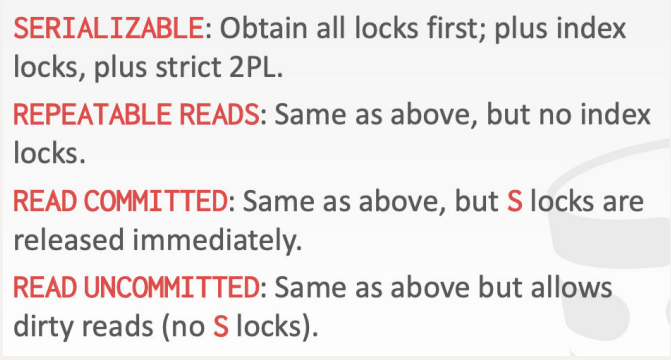
更低的隔离级别下事务之间会互相暴露，这也会引发一些问题：脏读（当前事务读的数据是其他事务修改过的但这个修改还没被提交），不可重复读（前后两次读同一个对象得到的数据不同，因为被其他事务修改过了），幻读（前后两次读同一个谓词下的数据集合，读到的数据规模不一样）

事务间的各种隔离级别如下，从上到下隔离级别越来越低，但性能也越来越好





各个隔离级别的实现方法：



Ref/参考自 :

https://www.jianshu.com/p/478c6dca1b74

https://www.bilibili.com/video/BV1ti4y197EG/?spm_id_from=333.788

https://blog.51cto.com/u_15103026/2645213

https://zhuanlan.zhihu.com/p/467927969

https://zhuanlan.zhihu.com/p/419056554

https://tech.zealscott.com/curriculum/adb/positive/