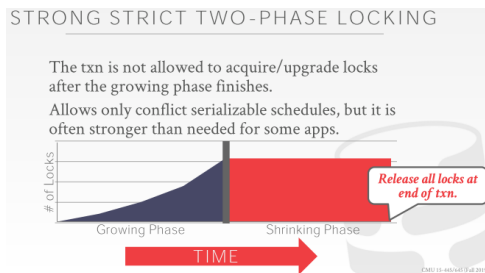


17-02



20:06 – 20:07

so we're going to tackle the first 1 first ,
So, 我们先来应对下第一个例子

20.07-20.10

we're going to talk about strong strict two-phase locking
我们要来讨论下强严格两阶段锁 (SS2PL)

20:13 – 20:14

it's sort of a misnomer

这里有点用词不当

20.14-20.17

because the second phase doesn't really exist anymore
因为这里第二个阶段并不存在了

20:18 – 20:20

all strong strict two phase locking says is

强严格两阶段锁的意思是

20.20-20.24

you don't release any of your locks until the end of the transaction ,when you're going to commit

当你执行完该事务，要提交该事务时，你才会去释放你的锁

20:25 – 20:27

the growing phase is exactly the same

这里的growing阶段和之前完全一样

20.27-20.29

you keep acquiring locks as you need them

根据需要，你去不断地获取锁

20:29 – 20:34

but you don't release any locks until commit time

直到你要提交事务的时候，你才会去释放这些锁

20:35 – 20:41

and this is going to allow us to prevent any ~~unrepeatable or~~ dirty reads from propagating across transactions

这可以防止我们遇上跨事务时所遇到的脏读问题

20.41-20.44

which is also going to solve the cascading aborts problem for us

这也会为我们解决cascading aborts问题

20:47 – 20:49

so yeah the figure is kind of updated

So, 这张图确实被换过了

20:49–20:50

but as you can see like

但你们可以看到的是

20:50–20:51

the shrinking phase

在shrinking阶段中

20:51–20:57

wherever you want to define it as like the last lock you acquired is the start of the shrinking phase

你获取最后一把锁的时间就是shrinking阶段开始的时间

20:58 – 20:59

but basically there is no shrinking phase

但简单来讲, 这里没有shrinking阶段

20:59–21:02

everything gets released at once at the end of the transaction

所有的锁都会在事务结束的时候释放掉

21:05 – 21:05

~~yeah~~

STRONG STRICT TWO-PHASE LOCKING

A schedule is **strict** if a value written by a txn is not read or overwritten by other txns until that txn finishes.

Advantages:

- Does not incur cascading aborts.
- Aborted txns can be undone by just restoring original values of modified tuples.

21:08 – 21:13

and the the word strict does have a specific meaning in when we're talking about concurrency control

当我们在讨论并发控制的时候, strict这个单词拥有特定的意义

21:13 – 21:14

it basically means

简单来讲, 它的意思是

21:14–21:27

~~anything that you wrote~~ none of your writes are going to be visible to ~~to any other asset~~

any other area of the system, any other transactions things like that until you commit

直到你提交事务的时候, 该事务所做的修改才会被系统中其他执行的事务所看到

21:27 – 21:28

so so in this context

So, 在这个语境中

21:28–21:30

strict has a very specific meaning

strict有一个非常特别的意思

21:30–21:32

and like I said that

正如我所说的

21:32–21:34

solves our cascade cascading aborts problem

这解决了我们的cascading aborts问题

21:34 – 21:38

because no other transactions are gonna see values that aren't committed to the system yet

因为没有任何事务会看到那些还未提交给系统的值

21:38 – 21:40

so they're only looking at committed data

So, 他们看到的只会是这些提交后的数据

21:40–21:42

everyone's happy

所有人都很高兴

21:44 – 21:46

and and like I mentioned before

就像我之前提到过的

21:46–21.48

this simplifies your abort logic

这简化了你的中止逻辑

21.48–21.52

because aborted transactions only have to put back their values

因为那些中止的事务必须要将它们所操作的值变回原样

21:52 – 21:58

you don't need to worry about all these transactions in the system potentially reading uncommitted data

这样的话, 你就无须去担心系统中的这些事务会去读取那些未提交的值了

21:59 – 22:08

and then they each kind of have to store their own ,metadata almost or methods to be able to roll back the the work that they've done

否则, 它们就得保存它们自己的元数据, 或者通过某种方法回滚它们之前所做的修改

22:08 – 22:12

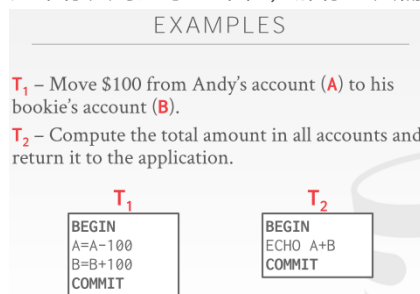
The simplifies things a lot by not having to reason about multiple versions in the system

在不探究系统中数据多版本的情况下, 这简化了太多东西

22:12 – 22:15

you only have one undo to do if you if you wrote a value

如果你只写入了一个值, 那你也只需撤销它即可



22:19 – 22:22

so we're gonna look at a simple example

SO, 我们来看个简单案例

22:23 – 22:25

andy owes his bookie money I guess

emmm, 我猜Andy欠了点赌债

22:26 – 22:29

so he's gonna move a hundred dollars to is to his bookies account

So, 他会转100美金到他的博彩账户中去

22:29 – 22:32

and then the second transaction is just gonna gonna compute the sum

接着, T2会去计算总和

22:32 – 22:35

This echo command is is made up it's not a real statement

这个ECHO我们编造的一条命令, 它并不是一条真正的命令

22:35–22:39

he just wants to demonstrate that you're reading these values out

他只是想去演示你们读取了这些值

22:40 – 22:42

we could have done something more complicated,

我们可以做些更为复杂的事情

22:42–22:45

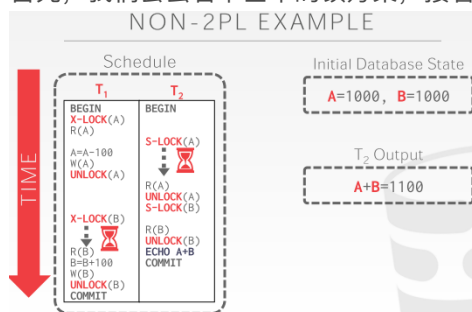
but we want to keep the example fairly simple

但我们想让这个例子简单些

22:45 – 22:50

first we're gonna look at it with I think just basic locking, then two-phase locking, and then strong strict 2 phase locking .

首先, 我们会去看下基本的锁方案, 接着是, 两阶段锁, 然后是强严格两阶段锁



22:51 – 22:56

so with the two locks that we defined at the beginning, and we're not if we're not using two phase locking

So, 我们先来看下不使用两阶段锁会是什么样子

22:57 – 23:02

~~we can see spoiler alert~~ you're gonna get a you're gonna get a wrong output from this
你会得到一个错误的输出结果

23:03 – 23:05

So we're start with a thousand dollars in each account

So, 在一开始的时候, 这两个账户中分别都有1000美金

23:08 – 23:10

t1 already gets the exclusive lock does a read

T1已经拿到了A所对应的exclusive lock, 并对A进行读取

23:10–23:14

because it needs to decrement his his balance by a hundred bucks

因为T1需要需要对他的银行存款减去100美金

23:16 – 23:17

t2 wants to get the shared lock

T2想去获取A对应的shared lock

23:17–23:19

because it's trying to compute the sum

因为T2要试着计算出总和

23.19–23.20

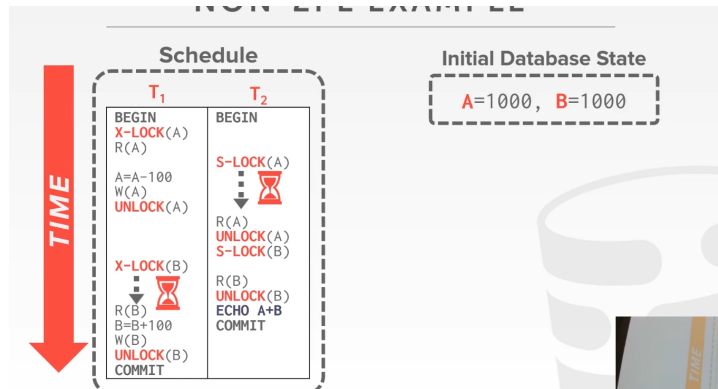
it can't get it

它没法获得这个lock

23:20 – 23:21

so it just starts waiting

So, 它只能开始等待获取这个lock



23:22 – 23:26

t1 finishes his operation to decrement and unlocks a

T1结束了它的操作，并释放了A对应的锁

23:26–23:27

at that time

与此同时

23:27–23:30

t2 gets the lock on a and performs its read

T2拿到了A对应的shared lock，并对A的值进行读取

23:31 – 23:32

And then unlocks it

接着，T2释放了锁

23:32–23:33

, because again we're not in two phase locking here

再说一遍，因为我们这里并没有使用两阶段锁

23:34 – 23:36

so you're free to acquire and release locks as you need them

So，根据你的需要，你可以自由地去获取和释放锁

23:39 – 23:41

it also t2 also gets the shared lock on B

T2也会去获得B对应的shared lock

23:41–23:45

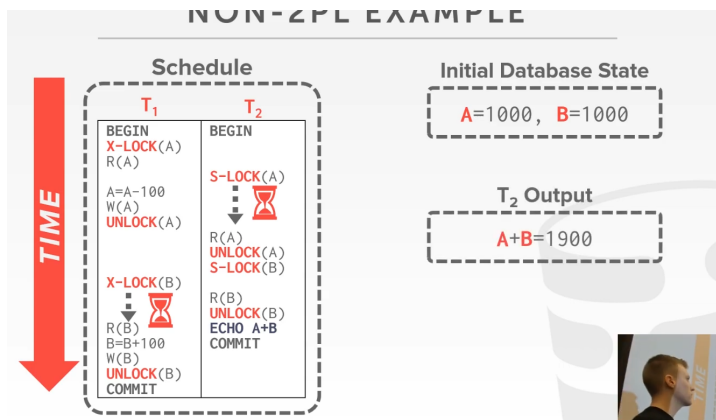
, because it needs to do a read on that which leads t1 to stall

因为它需要去读取B的值，这会导致T1停下来

23:44 – 23:48

Because it can't do the operation or it can't access that tuple yet

因为T1无法执行该操作，或者说它无法访问该tuple



23:49 – 23:52

eventually t2 releases the lock

最终，T2释放了这把锁

23:52–23:54

t1 gets the lock on B

T1拿到了B对应的exclusive lock

23:54–23:58

finishes moving the money over to the bookies account unlocks commits

它将钱移动到博彩账户，接着释放锁并提交事务

23:59 – 24:02

and t2 gives us a wrong output

这里T2给了我们一个错误的输出结果

24:02–24:05

,because did it read an inconsistent state

因为它读取到了一个不一致的状态

24:05 – 24:09

It read part of the work that t1 had done

它读取到了T1已经完成的部分工作结果

24:09–24.12

and t1 leaked that information to the rest of the system

T1将该信息泄露到了系统的其他地方

24.12–24.14

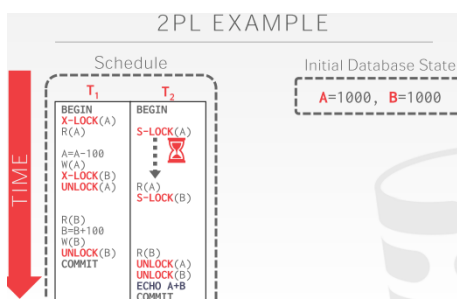
,and said okay this bank account balances 900 bucks

并说：该银行账户中的余额是900美金

24:15 – 24:20

But it that hundred dollars was missing at that point when t2 came along and got the locks that it needed

但当T2执行并获取它需要的lock时，这100美金消失了



24:22 – 24:23

so with two-phase locking

So, 在两阶段锁中

24.23–24.29

the key thing to notice here is

这里要注意的关键在于

24.29–24.31

it starts the same,

它们在同一时间开始执行

24.31–24.35

t1 gets its exclusive lock on a ,performs the operation it needs

T1获取了A对应的exclusive lock, 并执行了它需要执行的操作

24:35 – 24:37

But before it unlocks a

但在它释放A对应的这把锁之前

24.37–24.39

,which would put it into the shrinking phase

即在T1进入shrinking阶段之前

24:39 – 24:42

it acquires the lock it needs on B all the way down here

它去获取了它需要用到的B对应的exclusive lock

24:43 – 24:45

so it gets the exclusive lock on b

So, 它拿到了B对应的exclusive lock

24.45–24.48

, t2 stuck waiting around waiting for the locks that it needs

T2原地等待获取它需要的那些lock

24:50 – 24:52

because this isn't strong strict two-phase locking

因为这里使用的并不是强严格两阶段锁

24:53 – 24:58

t1 actually unlocks ,it well yeah

实际上, T1释放了锁

24:59 – 25:00

if this were strong strict

如果这里使用的是强严格两阶段锁

25.00–25.02

it would be down here at commit time

它会到提交的时候再释放锁

25:02 – 25:06

so it dont unlocks a, finishes this operation on on object B

So, 它不会释放A对应的锁, 直到它对B的操作执行完毕

25:07 – 25:08

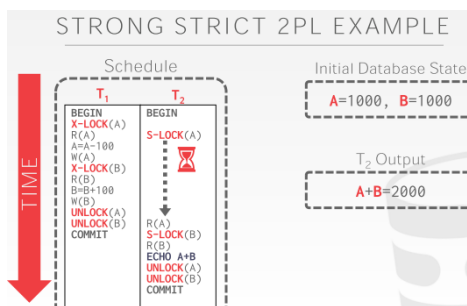
and then releases the locks

然后, 它才会去释放锁

25.05–25.11

,and this actually gives us a correct output

实际上, 这会给我们一个正确的输出结果



25:13 – 25:16

Same example with strong trick two-phase locking like I said

我们来看个相同的例子，这里使用的是强严格两阶段锁，正如我说的

25:16–25:19

the unlocking happens at the end right before commit,

释放锁的操作会在事务最后提交前执行

25:19–25:21

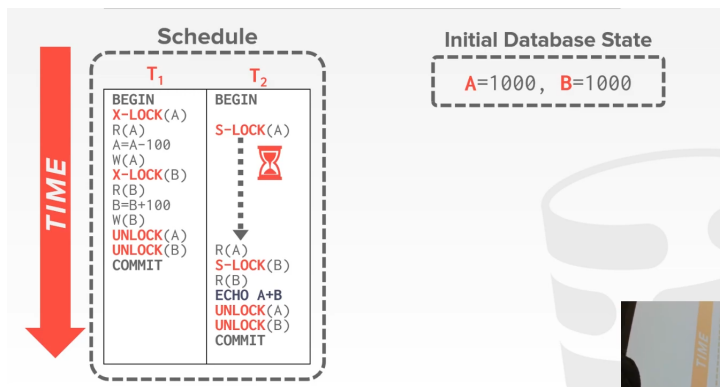
so yeah t1 gets its lock

So, T1拿到了锁

25:21 – 25:23

t2 has to wait the entire time

T2需要等待一整个T1事务执行时间（等到T1被提交时才行）



25:25 – 25:32

you can sort of see how strong strict two-phase locking is effectively forcing a serial ordering for these transactions

你们可以看到强严格两阶段锁有效地强制了事务的有序执行

25:33 – 25:37

by basically acquiring all your locks ,holding on to them until you get to commit time

简单来讲，事务会去获取你所需要用到到的所有锁，一直拿着这些锁，直到你提交该事务的时候

25:37 – 25:44

you're guaranteeing that that any of the operations that that t2 would have that would conflict are going to be forced in a serial ordering

你会保证T2中所存在的任何会引起冲突的操作会被强制按照某种顺序执行

25:46 – 25:47

make sense

懂了吗？

25:51 – 25:52

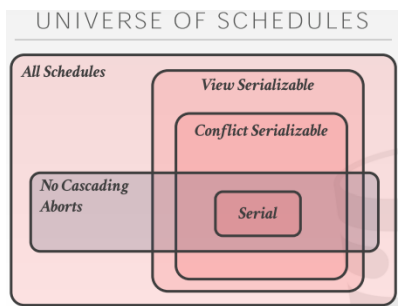
I'll go with yes

我猜你们懂了

25:54 – 25:57

and then the correct output again as well

那么，正确的结果就是2000



25:57 – 26:03

so I think I think Andy showed this slide before in the universe of schedules

So, 我相信Andy之前已经向你们展示过这个张图了

26:03 – 26:09

he showed Serial schedules ,conflict serializable schedules ,View serializable

他展示了serial schedule、Conflict Serializable Schedule、View Schedule

26.09–26.11

and then I don't know if he had cascading aborts in there

我不知道他有没有向你们展示这个cascading aborts

26:11 – 26:15

so this slide actually I think is he mentions it last year is actually incomplete

So, 实际上我认为Andy去年提到的这个幻灯片是处于未完成状态

26.15–26.23

what he mentioned to show is where two-phase locking ,and strong strict two phase locking live in this hierarchy

在这个层次结构图中, 他想展示的是两阶段锁和强严格两阶段锁

26:23 – 26:27

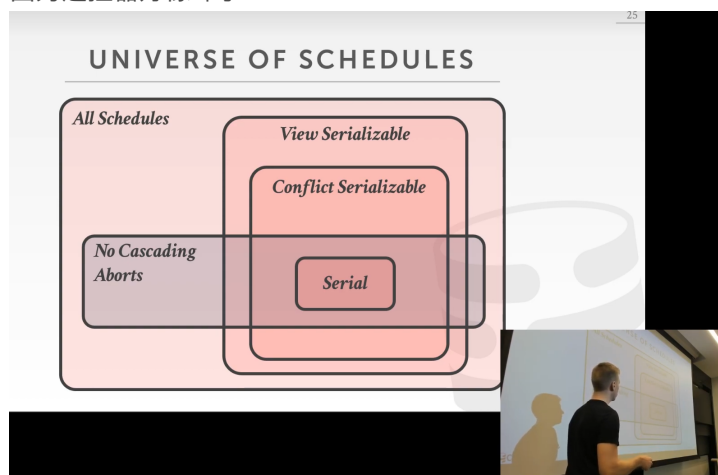
and I'll see, if I can at least use the laser pointer

如果我能使用激光笔就好了

26.27–26.31

because a clicker doesn't seem to be doing anything ,nope

因为遥控器好像坏了



26:32 – 26:38

so two-phase locking is gonna live in kind of this area right here

So, 两阶段锁所处的位置是在这个区域

26:39 – 26:41

it's guaranteed to generate conflict serializable schedules

它会保证生成Conflict Serializable的Schedule

26:42 – 26:44

but it's going to be susceptible to cascading aborts

但它也容易导致cascading aborts的情况发生

26:47 – 26:51

I'll see if we can get this slide updated for the ones that actually get published on the site

如果这张幻灯片后续更新的话，那我会将它贴到网站上

26:51 – 26:54

so we actually see better than just a laser pointer on the video

So，这要比我用激光笔在上面指来指去要来得好

26:54 – 26:56

But yeah two phase locking would be here

但总之，两阶段锁是在这个位置

26:56–26:57

and then inside of this box

它在这个方框里面

26:57–27:00

and a round serial

在Serial这个方框周围

27:00–27:01

you would have strong trick two-phase locking

我们会有强严格两阶段锁

27:02 – 27:04

because it's guaranteed not to have cascading aborts

因为它保证我们不会出现cascading aborts的情况

27:04–27:06

we get conflict serializable schedules

我们会得到Conflict Serializable的schedule

27:06–27:08

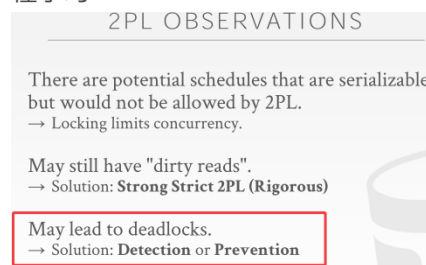
we're happy

我们会为此感到高兴

27:08–27:10

makes sense

懂了吗？



2PL OBSERVATIONS

- There are potential schedules that are serializable but would not be allowed by 2PL.
→ Locking limits concurrency.
- May still have "dirty reads".
→ Solution: **Strong Strict 2PL (Rigorous)**
- May lead to deadlocks.
→ Solution: **Detection or Prevention**

27:13 – 27:17

so now let's talk about the other problem with two-phase locking

So，我们现在来讨论下使用两阶段锁时遇到的另一个问题

27:17–27:19

which is that it leads to dead locks

它会导致死锁的出现

27:19–27:20

and like I mentioned before

正如我之前所提到的

27:20 – 27:23

there's a couple different ways we can try to solve this problem

我们可以通过两种不同的方式来试着解决死锁问题

27:23 – 27:31

we can be kind of lazy about it, and wait use a detection algorithm to to find a deadlock

我们可以对它进行懒处理，使用一种检测算法来发现死锁

27.31–27.33

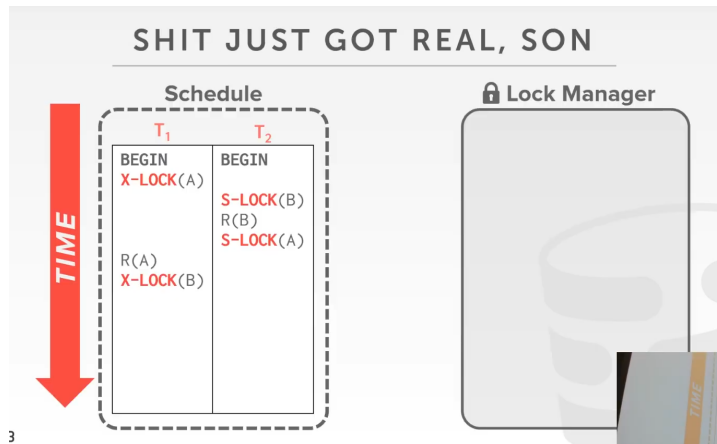
or we can kind of be a little bit more proactive about things

或者，我们可以主动出击

27.33–27.35

and try to prevent them from ever happening in the first place

在死锁问题发生之前就防范于未然



27:43 – 27:44

so you've probably seen dead locks before

So，你们之前可能已经见过死锁了

27.44–27.49

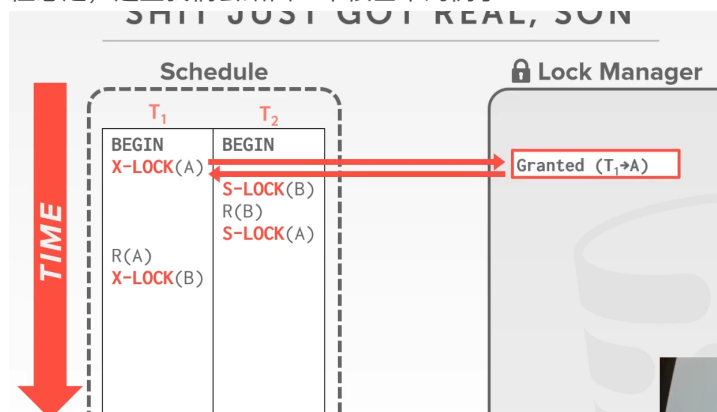
it's a fairly common concept in computer systems

它是计算机系统中一个相当常见的概念

27:50 – 27:52

But we'll give a basic example anyway

但总之，这里我们会给出一个很基本的例子



27:52 – 27:55

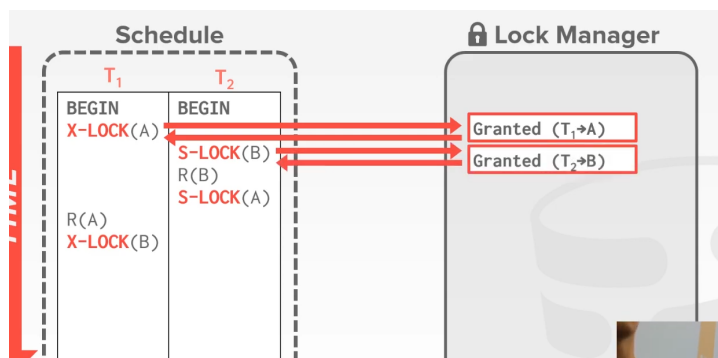
so t1 wants an exclusive lock on a

So，T1想去获取A所对应的exclusive lock

27.55–27.58

lock managers like sure you can have that no one else has that lock

lock管理器表示你可以去拿这个lock，现在没有人拿着它



27:58 – 28:00

t2 gets the lock on B,

T2拿到了B所对应的shared lock

28.00–28.01

because no one else has that lock

因为没有人拿着它

28:02 – 28:03

now t2 says

现在T2表示

28.03–28.04

I want the lock on a

我想获取A对应的shared lock

28.04–28.07

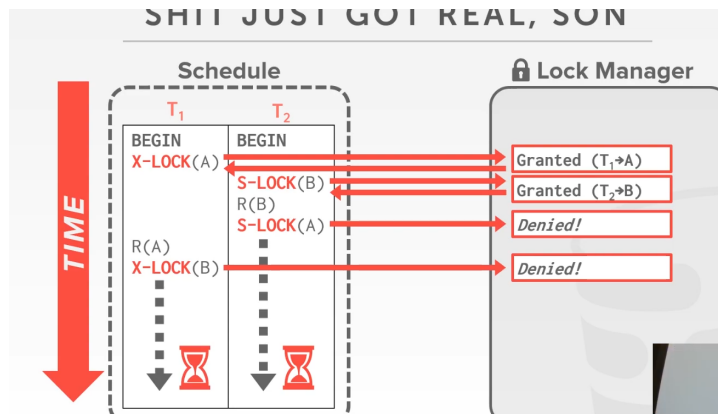
lock managers like no you don't get that

lock管理器表示：No，你拿不到这个lock

28:07 – 28:10

but t2 is gonna sit around and wait for that lock

但T2会坐在原地等待获取这个lock



28.10–28.13

and t1 now wants the exclusive lock on B

T1现在想去获取B对应的exclusive lock

28:14 – 28:15

and it's gonna wait as well

它也会原地等待获取这个lock

28.15–28.16

we have a problem now

我们现在就遇上了一个问题

28:17 – 28:22

both these transactions are waiting for locks that the other transaction holds

这两个事务都在等待获取彼此手上的lock

28:22 – 28:23

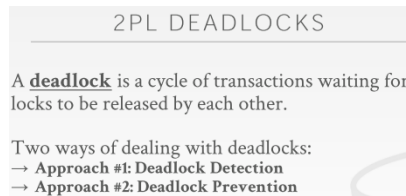
so we need to break this somehow

SO, 我们需要通过某种方式打破这种僵局

28:27 – 28:32

Yeah,nice animation we have a problem

我们摊上问题了



28:37 – 28:39

yeah so like I said

So, 就像我说的

28:39–28:49

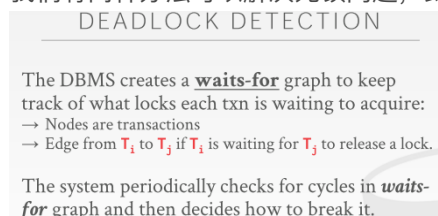
deadlocks are a when you have a dependency cycle between transactions with where they're holding locks and

死锁其实就是事务间彼此依赖成环, 它们都在等待对面释放它们所需要的锁

28:49 – 28:53

and we have a couple ways of dealing with them detection and prevention

我们有两种办法可以解决死锁问题, 即检测和预防



28:58 – 29:00

so with deadlock detection

So, 我们来讲下死锁检测

29:00–29:02

the systems gonna do with like a background thread

系统会使用一个后台线程来进行死锁检测

29:02 – 29:05

you're basically gonna look at the lock managers metadata

简单来讲, 你会去查看lock管理器中的元数据

29:05–29:07

and you're gonna build a waits-for graph

你会去构建一个waits-for图

29:07 – 29:08

so idea is

So, 这里的思路是

29:08–29:10

every node is it is a transaction

图中每一个节点就是一个事务

29:10–29:19

and every edge is pointing to another node that holds a lock that that transaction wants

每一条线会指向另一个节点, 该节点持有着另一端那个节点想要获取的锁

29:19 – 29:20

this is all going to be done in the background

这些都会在后台完成

29.20–29.23

you can balance out how frequently this gets done

你可以去平衡下这些操作执行的频率

29.23–29.25

we'll talk about that in a minute

我们稍后会对它进行讨论

29:25 – 29:26

but the idea is

但这里的思路是

29.26–29.27

it's a background task

它是一个后台任务

29.27–29.30

that inspects the state of the lock manager

它会去检查lock管理器的状态

29.30–29.36

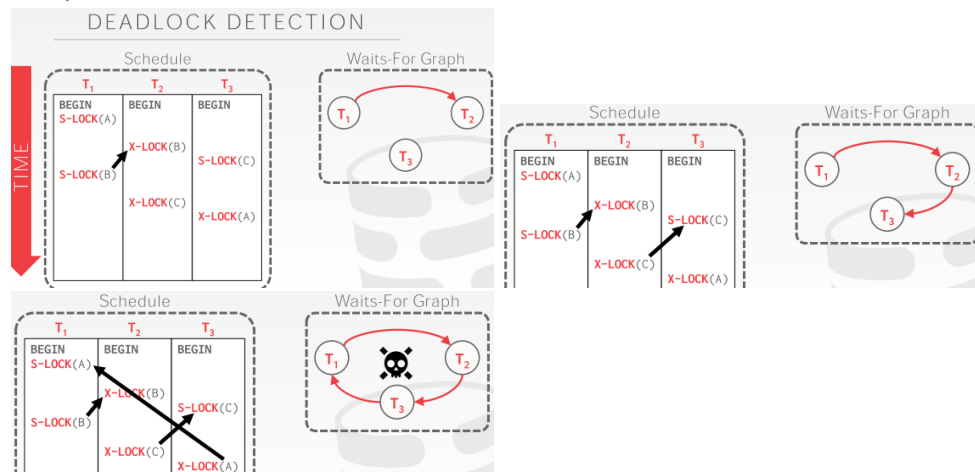
,and says hey are there any deadlocks use your favorite cycle detection algorithm

并说：hey，老兄，你用你喜欢的循环依赖检测算法是否找出了死锁？

29:36 – 29:41

And you have to decide what you're then gonna do with that deadlock

接着，你必须决定你接下来该如何处理这个找出的死锁



29:42 – 29:46

so let's look at a simple example of what these with these waits for graphs look like

So，我们来看个简单的例子，看看这些waits-for图是长啥样的

29:49 – 29:58

we'll start with T₁ wants a shared lock that T₂ already holds on holds an exclusive lock for

T₁想获取B所对应的一个shared lock，但T₂已经拿到了B所对应的exclusive lock

29:58 – 30:01

so T₁ gets an edge pointing to T₂

So，T₁会有一条指向T₂的线

30:03 – 30:05

T₂ is going to end up with an edge pointing to t₃

T₂最终会有一条指向T₃的线

30.05–30.06

because as you can see here

因为，你们这里可以看到

30.06–30.09

it wants an exclusive lock that T3 already holds a shared lock on

T2想去获取B所对应的exclusive lock，但T3已经拿到了B所对应的shared lock

30:10 – 30:11

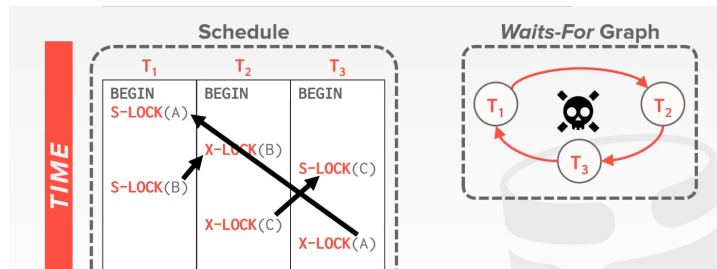
those are incompatible

它们是不兼容的

30.11–30.12

so we're just stuck waiting

So，我们只能进行等待了



30.12–30.15

,and then lastly T3 is gonna end up with an edge pointing to T1

在最后，T3会有一条指向T1的线

30:16 – 30:18

because it wants an exclusive lock that T1 has a shared lock on

因为T3想获取A所对应的exclusive lock，但T1已经拿到了A所对应的shared lock

30:19 – 30:22

we have a deadlock

我们遇上了死锁

30:23 – 30:25

so we have to do something about this

So，我们必须对它做点什么

30.25–30.27

questions

有问题吗？

30:30 – 30:31

what's that

你说的是啥？

DEADLOCK HANDLING

When the DBMS detects a deadlock, it will select a "victim" txn to rollback to break the cycle.

The victim txn will either restart or abort(more common) depending on how it was invoked.

There is a trade-off between the frequency of checking for deadlocks and how long txns have to wait before deadlocks are broken.

30:35 – 30:41

so deadlock handling is kind of simple

So，死锁处理有点简单

30.41–30.43

pick a victim, kill it

选择一个牺牲者（victim），干掉它，就像

30:44 – 30:46

you choose a transaction and you roll it back

你选择一个事务，然后将它回滚

30:46–30:51

how far you roll that back is a kind of implementation to find it's possible

回滚多少内容取决于具体实现

30:51 – 30:54

You don't and I think this is this is a later slide

你不需要去考虑这些，这个会在之后的幻灯片中出现

30:54 – 31:00

but you may not have to to abort the entire transaction ,may not have to undo all the queries that it did

你可能无须中止整个事务，也无须去撤销所有查询已经做的事情

31:00 – 31:09

you maybe only you may only need to partially rollback some of the queries to release the locks that you need to remove the deadlock , and make forward progress in the system

你可能只需回滚部分查询来释放锁，以此移除死锁，并在系统中取得进展

31:11 – 31:14

The the last point here is basically saying that

简单来讲，此处最后一点讲的是

31:14–31:19 ! ! !

you have a trade-off in the system with how frequently you're gonna build these waits-for graphs

在系统中你得权衡下构建这些waits-for图的频率

31:19 – 31:22

if the way you're dealing with deadlocks is detection

如果你检测处理死锁的方式是使用检测

31:23 – 31:26

you have this background task it's building these graphs checking for deadlocks

你需要运行这个后台任务来构建这些图，以用于检测死锁

31:27 – 31:30

it's up to you how frequently you want that task done

这个任务的执行频率取决于你

31:30–31:32

you could do it every microsecond, if you want

如果你想的话，你可以每微秒都执行它

31:32 – 31:38

but you're gonna burn a bunch of CPU cycles constantly building these graphs ,and potentially not finding any any deadlocks

但你可能会浪费CPU周期来构建这些图，并且你可能也检测不出什么死锁

31:39 – 31:41

so what you may want to reason about is

So，你可能想推断出的东西是

31:41 – 31:44

okay maybe I want to check for deadlocks less frequently

Ok，我想让检查死锁的频率变低一些

31:44 – 31:45

and if I do enter in a deadlock state

如果我陷入死锁的状态

31.45–31.50

you know how long do I want to make those transactions wait

我想让这些事务等待的时间有多长

31.50–31.57

what's an acceptable time out that I could sit in a deadlock state, without detecting it right away

在不检测死锁的情况下，我陷入死锁状态的可接受时常是多少

31:57 – 32:00

so these are always going to be tunable parameters in your database system

So，这些都是你可以在你数据库系统中进行调整的参数

32:00 – 32:04

because different workloads are gonna manifest different deadlock behaviors

因为不同的workload会表现出不同的死锁行为

32:04 – 32:12

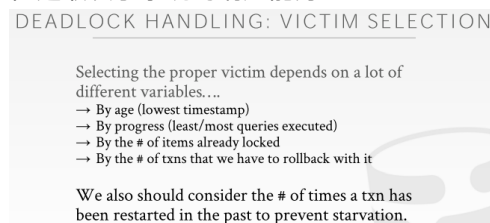
so we want to make sure, we're not being too aggressive, it may be fine just to leave the system in a deadlock state for ten seconds even

So，我们想确保我们不会太过激进，让系统处于死锁状态10秒可能也不是什么大事

32:12 – 32:15

but it depends on what the responsiveness of the system needs to be

但这取决于系统的响应能力



32:21 – 32:23

so victim selection

So，我们来讲下Victim Selection

32.23–32.27

there's a lot of different things you can look at here

你可以在这里看到很多不同的东西

32.27–32.31

and certain systems get very elaborate with what they do

某些系统会非常详细地描述它们所做的事情

32:31 – 32:33

But there's all sorts of different heuristics you can look at

但你可以看到这里有各种不同的启发式规则

32:34 – 32:39

the first is look at look at the age of the transaction, when you're trying to find a victim that you're going to kill

首先，当你要试着干掉某个victim（牺牲者）时，我们可以去看下时间戳最小的那个事务

32:39 – 32:41

Because at the end of the day you have this cycle ,you got to pick one

因为到最后，你会有一个cycle，你可以从中选一个事务来干掉

32:43 – 32:44

you can start with the timestamp

你可以从时间戳开始入手

Selecting the proper victim depends on a lot of different variables....

→ By age (lowest timestamp)

→ By progress (least/most queries executed)

32.44–32.50

, you can look about how much work it's done ,how many how many queries has it executed ,~~how big progress is a~~

你可以去查看下它已经完成的工作量，或者它已经执行的查询数量

32.50–32.54

you don't know how close it is being done

你不知道它距离完成还有多久

32:54 – 32:56

but you can at least reason about how much work it's done

但至少你能推断出它已经完成了多少工作量

Selecting the proper victim depends on a lot of different variables....

→ By age (lowest timestamp)

→ By progress (least/most queries executed)

→ By the # of items already locked

32:58 – 32:59

How many locks it already holds

接着就是，它手上已经拿了多少个lock

33:02 – 33:04

let's see

我们再来看看。。。

→ By the # of items already locked

→ By the # of txns that we have to rollback with it

33.04–33.13

that's it's possible that you would have to look at the number of transactions, you have to roll back that would be in the case of,if you have cascading abort

如果你遇上cascading abort问题的话，你得去看下你需要回滚的事务数量

33:13 – 33:14

I don't think I mentioned before

我不觉得我之前有提到过

33.14–33.17

you don't have to do strong strict two-phase locking

你不一定要去使用强严格两阶段锁

33.17–33.19

some systems may just do two-phase locking

某些系统可能就只是做到了两阶段锁

33.19–33.23

and live with the possibility of a cascading abort

并且它们有可能会遇上cascading abort的问题

33:23 – 33:27

again it's gonna be workload dependant

这取决于workload是什么

33:29 – 33:35

you can decide if the workload is not very susceptible to cascading aborts or dirty reads
如果这些workload并不容易导致cascading aborts或者脏读的情况

33:35 – 33:37

you can just say two-phase locking is fine
那你就说，两阶段锁够用了

33:37–33:38

I can release my lock sooner
我可以以更快的速度去释放我的锁

33:38–33:40

that may result in higher throughput in the system
这可能会让系统的吞吐量变得更高

33:40 – 33:43

Again that's often gonna be something that that's configurable
我们通常可以通过配置某些选项来做到这点

33:43 – 34:43

yeah

请问

DEADLOCK HANDLING: VICTIM SELECTION

Selecting the proper victim depends on a lot of different variables...

- By age (lowest timestamp)
- By progress (least/most queries executed)
- By the # of items already locked
- By the # of txns that we have to rollback with it

We also should consider the # of times a txn has been restarted in the past to prevent starvation.

33:54–34:03

so we'll talk about a couple different ways to decide a little bit later on about about which transaction gets killed

So，我们之后会去讨论两种不同的方式来决定哪个事务该被干掉

34:03 – 34:07

there there are a couple different solutions
我们有两种不同的方案可以做到这点

34:07–34:09

but the other one is
但另一点是

34:14 – 34:16

yeah this is typically going to be a combination of things
通常我们会将这些东西结合起来使用

34:16 – 34:21

it's not necessarily always going to be just the age of the transaction is going to be what determines whether you get killed or not often
我们不一定总是根据时间戳来判断我们该杀死哪个事务

We also should consider the # of times a txn has been restarted in the past to prevent starvation.

34:22 – 34:28

one of the biggest things is going to be the number of times you've already been killed
which is, yeah then the last one says there

最重要的一点，也就是这里最后一点所说的东西，即我们应该考虑某个事务已经被重启的次数

34:28 – 34:30

because you do want to make sure you make progress in the system

因为你想去确保你在系统中取得了一些进展

34.30–34.32

you need to prevent starvation for these transactions

你需要去防止这些事务可能会导致的starvation情况

34:34 – 34:40

and I'm not saying like any one of these is exactly what you're going to use to kill a
transaction

我并没有说你们一定要根据上面列出的东西来干掉一个事务

34:40 – 34:51

the commercial systems get rather elaborate and can build predictive models even to try
to figure out like ,which transaction they should try to kill and answer a deadlock
scenario based on all these different heuristics

商用数据库系统相当复杂，它们甚至构建了预测模型来试着弄清楚它们该干掉哪个事务，并基于
所有这些不同的启发式规则来解决死锁场景

34:53 – 34:54

so it's typically a combination

So，我们通常会将它们结合起来使用

34.54–34.58

it's not just the timestamp I guess is a long way of answering hopefully your question

我猜他们使用的不仅仅只是时间戳，希望我的回答对你的问题有所帮助

35:22 – 35:23

okay,so your question is

So，你的问题是

35.23–35.27

~~why would you~~, sorry I want to make sure I repeat back for the video

为了确保录制的视频没问题，我要重复下你的问题

35:27 – 35:27

your question is

你的问题是

35.27–35.32

why would you always want to kill ,it what scenarios would you want to kill the
transaction with the lowest timestamp always

在哪种场景下，我们总是想去干掉时间戳最小的那个事务

35:39 – 35:51

yeah I suspect it's it's still always going to depend on the workload , depending on
what's causing the deadlock ,what sort of a situation is causing it ,but

我猜这依然取决于workload是什么、还有引起死锁的原因，以及在什么场景下会引起死锁问题

35:53 – 35:59

yeah we'll formalize a little bit like in a few slides on what gets killed and and why

我们会通过一些幻灯片来正式讲下它，即哪些事务会被干掉，以及为什么会被干掉

35:59 – 36:05

but I think in the case of Postgres in the example

以PostgreSQL为例

36.05–36.08

we'll see the highest time stamp gets killed

时间戳最大的那些事务会被干掉

36:08 – 36:10

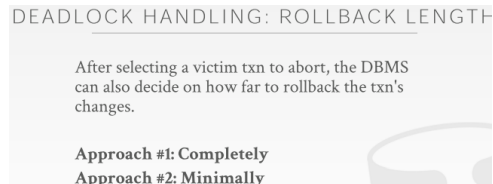
I think I think it's just it's just an example

这只是我举得一个例子

36.10–36.13

I'm just saying like one you're a stick you could look at.

我只是想表达的是，这是个你们要去看例子



36:17 – 36:19

And like I mentioned before

就像我之前提到的

36.19–36.22

it's possible you don't need to abort the entire transaction

你可能不需要去中止整个事务

36:22 – 36:23

you could completely abort it

你可以将它完全中止

36.23–36.25

or you could do some sort of minimal abort

或者，你可以最小程度地中止它

36.25–36.29

where you only rollback the number of queries in that transaction

即仅回滚该事务中的个别查询所做的修改

36.29–36.30

that will allow you to release the deadlock

这会让你脱离死锁的情况

36:31 – 36:32

And make forward progress in the system

并且能让你在系统中取得进展

36:33 – 36:37

again this is something that's going to depend on the workload, how much work you're throwing away

这取决于你的workload是什么，以及你要丢弃多少工作进度

36:38 – 36:42

and whether it just makes sense to have transactions resubmit all their queries

并让这些事务重新递交它们所有的查询这是否有意义

36:43 – 36:51

or if you can slowly unwind parts of it in order to to free the system up in the deadlock and make forward progress

或者你可以缓慢地释放部分锁，以此来让系统脱离死锁，并取得一些进展

=====

36:56 – 36:58

let's see if we can get a demo working

我们来看个demo

37:15 – 37:16

pardon

稍等一下

37:16–37:19

my typing on this surface keyboard

我用surface的键盘打字比较慢

37:32 – 37:34

cool looks like it's working

Cool, 看起来奏效了

A terminal window with a black background and white text. The title bar reads "SmartTTY - i-would-never-give-tk-your-unpublished-paper:2111". The prompt "mysql>" is visible on two lines, with a cursor on the second line.

```
mysql> _  
  
mysql>
```

37:37 – 37:38

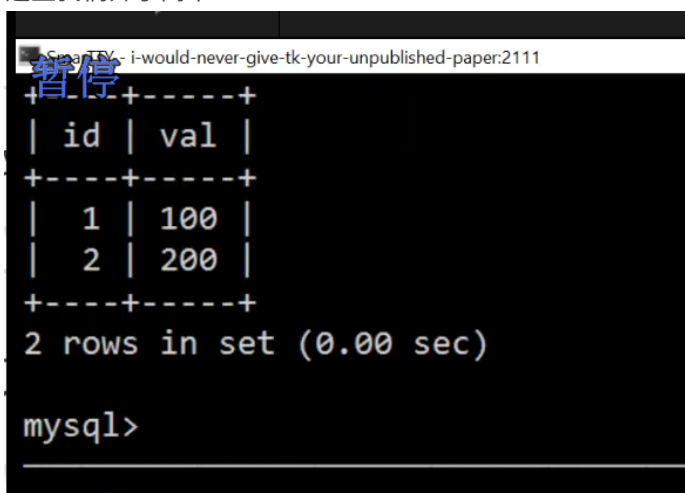
so we'll start with MySQL

So, 我们先从MySQL开始

37:38–37:43

we'll have two terminals open

这里我们开了两个terminal

A terminal window with a black background and white text. The title bar reads "SmartTTY - i-would-never-give-tk-your-unpublished-paper:2111". A blue "暂停" (Pause) watermark is visible. The output of a MySQL query is shown, including a table with 2 rows and a message "2 rows in set (0.00 sec)". The prompt "mysql>" is visible at the bottom.

```
+---+-----+  
| id | val |  
+---+-----+  
| 1 | 100 |  
| 2 | 200 |  
+---+-----+  
2 rows in set (0.00 sec)  
  
mysql>
```

37:43 – 37:48

We have a pretty basic table set up called transaction demo

这里我们有一个很简单的表, 它叫做txn_demo

37:48–37:49

, that's horrible

这有点糟糕

37:49–37:51

you can't see what's happening there

你们没法看到这里发生了什么

37:51 – 37:52

But basically we have two tuples in it
但基本上来讲，这张表中我们有两个tuple

37.52–37.55

primary key ID one and two

主键是id，里面有两个值，即1和2

37.55–37.57

,and the values 100 and 200

val分别是100和200

37.57– 37.58

pretty simple

这里面的东西相当简单

37.58–38.00

we have two tuples

我们有两个tuple

38.00–38.03

this should be really easy to put to put us into a deadlock State

这应该很容易让我们进入死锁状态

38:04 – 38:04

so

请问

38:06 – 38:10

I have no idea which ones those are

我也不清楚它们哪个是哪个

38:17 – 38:19

I'm glad I was standing next to you we got that on tape

我很高兴我站在你旁边

38:27 – 38:28

~~producer one everyone~~

38:35 – 38:39

so we've got these these two tuples in our table transaction demo

So，在txn_demo中，我们有2个tuple

```
2 rows in set (0.00 sec)

mysql> SET GLOBAL innodb_lock_wait_timeout = 10;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

38:39 – 38:44

first thing we're gonna do is set our timeout here

首先我们要做的是设置我们的超时时间

38:48– 38:49

so like I was saying before

So，像我之前说的

38.49–38.52

you can adjust things like like how frequently you're gonna detect for deadlocks

你可以去调整下你检测死锁的频率

38:52 – 38:53

so in this case

So，在这个例子中

38:53–38:58

we're gonna try to change the the lock_wait_timeout here

我们会试着去修改innodb_lock_wait_timeout的值

38:59 – 39:08

InnoDB is the storage engine for MySQL as of I don't know MySQL five I think or something like that

我记得InnoDB是MySQL 5.x的存储引擎来着

39:08 –39:11

so the first terminal will begin a transaction

So, 我们在第一个terminal中开始执行一个事务

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE txn_demo SET val = val + 1 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

39:12 – 39:18

the first thing we're gonna do is try to update the value of tuple one

我们首先要做的事情是, 试着去更新tuple 1中的值

```
mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 sec)

mysql> _
```

39:20 – 39:23

oh actually I missed a step

Oh, 实际上, 我漏了一个步骤

39:24 – 39:27

we need to explicitly tell MySQL

我们需要显式告诉MySQL

39:27 – 39:30

We want to run in the serializable isolation level

我们想在隔离级别为Serializable的情况下执行事务

39:30 – 39:33

so that's going to give us conflict serializable schedules

So, MySQL就会给我们Conflict Serializable的schedule

39:33 – 39:34

We haven't talked about isolation levels yet

我们目前还未讨论过隔离级别方面的东西

39:34–39:37

like I said I think that's going to be in the next lecture

我之前说过, 我会在下节课的时候讲它


```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.01 sec)

mysql>

mysql> SET GLOBAL innodb_lock_wait_timeout = 10;
Query OK, 0 rows affected (0.00 sec)

mysql> SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;_
```

39:38 – 39:44

but we need to tell the system, we want serializable isolation level

但我们需要告诉系统，我们想要的隔离级别是Serializable的

39:49– 39:50

go back

回到之前

39:50–39:52

begin our transaction

开始执行我们的事务

```
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE txn_demo SET val = val + 1 WHERE id = 1;_

mysql> SET GLOBAL innodb_lock_wait_timeout = 10;
```

39:52–40..00

and we're gonna like I said update the value on tuple one

假设，我要对tuple 1中的值进行更新

40:02 – 40:03

switch to our other terminal

切换到我们的另一个terminal

40:04 – 40:05

we're going to start a transaction

我们要开始执行一个事务

40:05–40.07

we're going to update the value on tuple two

我们要去更新tuple 2中的值