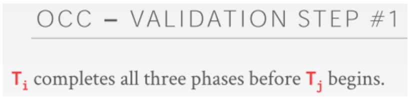
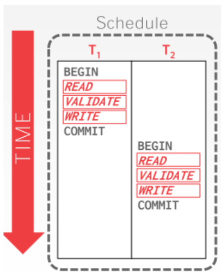
18-03

40:02 - 40:06
so it's a it should now be installed into the system
So，它现在应该被落地到系统中了
40:11 - 40:12
all right

40:14 - 40:14
So okay

40.14-40.15
so for the rest of this lecture
So，在这节课剩下的时间里
40.15-40.18
,let's just assume that we're always doing forward validation
假设，我们使用的始终是forward validation
40:18 - 40:19
so anytime you want to validate a transaction
So，每当你想对一个事务进行验证时
40.19-40.21
you find all newer transactions than you
你会找到所有比该事务时间戳更大的事务
40.21-40.23
and you perform this validation step
你就会执行这个验证步骤

OCC – VALIDATION STEP #1

T_i completes all three phases before T_j begins.
40:26 - 40:29
okay,so there's a couple of different scenarios that we have to cover
So，我们需要介绍两种不同的情况
40:29 - 40:30
okay the first scenario is that
Ok，第一种情况是
40.30-40.32
if you find a transaction TJ
如果你找到一个事务Tj
40:34 - 40:42
~~where all your phases your read your would sorry,~~your work your that your validation in your write steps happen before anything that TJ has done
你原有事务中的写操作要在Tj这个时间戳更大的事务执行任何操作前完成



40:42 – 40:45

all right this is sort of like the simple happy path case here here

这就如这里这个简单的例子所展示的

40:45 – 40:47

so this is an example

So，在这个例子中

40.47–40.48

you have t1 and t2

我们有两个事务，即T1和T2

40.48–40.52

,t1 completes all of its steps before t2 is then anything

在T2执行任何操作之前，T1就完成了它的所有步骤

40:53 – 40:55

okay so this is a really simple example

Ok，这是一个非常简单的例子

40.55–40.58

you could essentially collapse this into one and now you have a serial execution

本质上来讲，你可以将这些事务合并为一个，然后你就得到了一个Serial execution

40:59 – 41:00

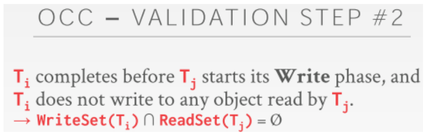
so you dont only have to do anything

So，你无需去做任何事情

41.00–41.04

all that interesting here this is sort of given to you naturally

这里所有你感兴趣的东西都是默认很自然的提供给你的



OCC – VALIDATION STEP #2

T_i completes before T_j starts its **Write** phase, and T_i does not write to any object read by T_j.

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

41:06 – 41:11

the second scenario that we have to handle, is that

我们需要处理的第二种情况是

41.11–41.17

if ~~t1 completes before TJ ,sorry~~ TI completes before TJ begins its write phase

如果Tj在开始执行它的写阶段之前，Ti就完成了它的工作

41:18 – 41:19

okay

41.19–41.21

and we have to make sure in this scenario

我们需要确保，在这种情况下

41.21–41.28

that the stuff that we write the the write set in our transaction ,doesn't intersect with the read set of the other transaction

我们事务中写操作要处理的东西不会与其他事务读操作涉及的东西相交

41:28 – 41:32

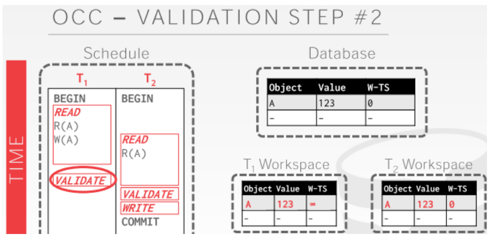
so the other transaction hasn't read anything that we're going to that we've written

即其他事务不会去读取任何我们要写入的东西

41:32 – 41:33

okay why is this important

为什么这点很重要呢?



OCC – VALIDATION STEP #2

Database

Object	Value	W-TS
A	123	0
-	-	-

T1 Workspace

Object	Value	W-TS
A	123	∞
-	-	-

T2 Workspace

Object	Value	W-TS
A	123	0
-	-	-

41.33–41.35

so let's walk through an example

So，我们来看个例子

41:35 – 41:38

so in this scenario we have two transactions

So，在这个场景下，我们有两个事务

41.38–41.39

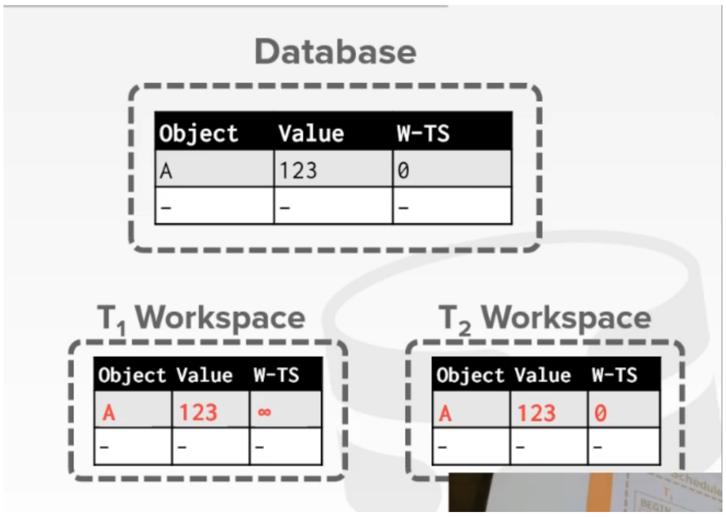
t1 is reading and writing a

T1对A进行读取和写入操作

41.39–41.42

t2 is reading a and doing nothing else

T2除了读取A以外，其他什么事也不做



Database

Object	Value	W-TS
A	123	0
-	-	-

T1 Workspace

Object	Value	W-TS
A	123	∞
-	-	-

T2 Workspace

Object	Value	W-TS
A	123	0
-	-	-

41:42 – 41:46

so so this is the state of the database

So，这就是数据库的状态

41.46–41.50

so t1 has read a and written to it assume that is writing the same value 123

So，T1对A进行读取，并将它的值修改为123

41:51 – 41:52

and it's timestamp is infinity,

它的时间戳是无穷大

41.52–41.54

 T2 is read a

T2对A进行读取

41.54–41.56

so it has a timestamp of 0 there

So，它的时间戳就是0

41.56–41.59

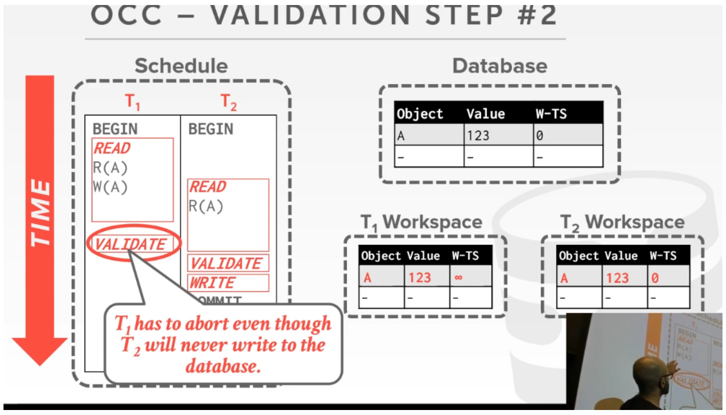
,and it has a local copy in its in its private workspace

在它的私有工作空间中，它保存了A的一份本地副本

41:59 – 42:02

and then you want to perform the the validation here

接着，我们想在这里执行下验证操作



OCC – VALIDATION STEP #2

Schedule

T1	T2
BEGIN	BEGIN
READ	
R(A)	
W(A)	
	READ
	R(A)
VALIDATE	
	VALIDATE
	WRITE
	COMMIT

Database

Object	Value	W-TS
A	123	0
–	–	–

T1 Workspace

Object	Value	W-TS
A	123	∞
–	–	–

T2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

T_1 has to abort even though T_2 will never write to the database.

42:03 – 42:05

right so t1 has to abort

So，这里需要中止T1

42.05–42.06

 because it’s read set

图标为它是read set

42:06 – 42:09

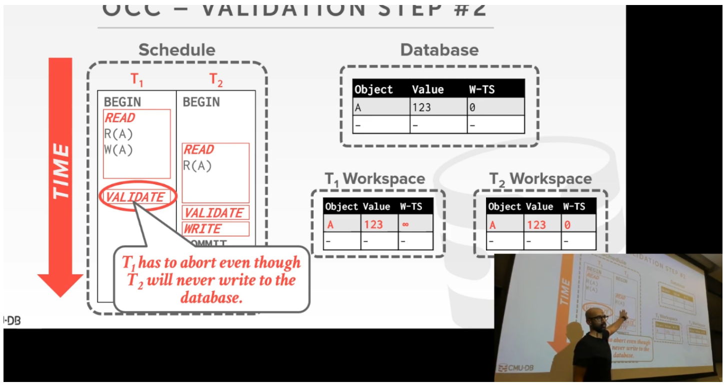
so it has a lower timestamp than t2

So，它的时间戳要比T2小

42.09–42.11

,because it begins validation first

因为它先开始进行验证操作



OCC – VALIDATION STEP #2

Schedule

T1	T2
BEGIN	BEGIN
READ	
R(A)	
W(A)	
	READ
	R(A)
VALIDATE	
	VALIDATE
	WRITE
	COMMIT

Database

Object	Value	W-TS
A	123	0
–	–	–

T1 Workspace

Object	Value	W-TS
A	123	∞
–	–	–

T2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

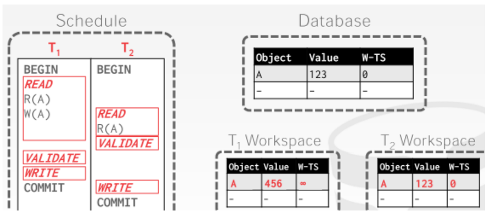
T_1 has to abort even though T_2 will never write to the database.

42:11 – 42:14

So it’s write set intersects with the read set here

So，它的write set与此处的read set相交了

42:15 – 42:17
all right so that violates the invariant that we had before
So，这违反了我们之前拥有的不变量
42.17–42.21
so we have to abort this transaction
So，我们需要中止这个事务（T1）
42:22 – 42:23
okay

42.23–42.24
in the serial order
在Serial Order的情况下
42.24–42.26
t1 has to begin before t2,
T1必须在T2之前开始
42.26–42.28
but t2 has read a stale value
但T2读取到了一个过时的值
42:29 – 42:32
so we have to abort
So，我们需要中止事务
42:34 – 42:37
I think it’s a pretty simple example
我觉得这是一个相当简单的例子


Database

Object	Value	W-TS
A	123	0
-	-	-

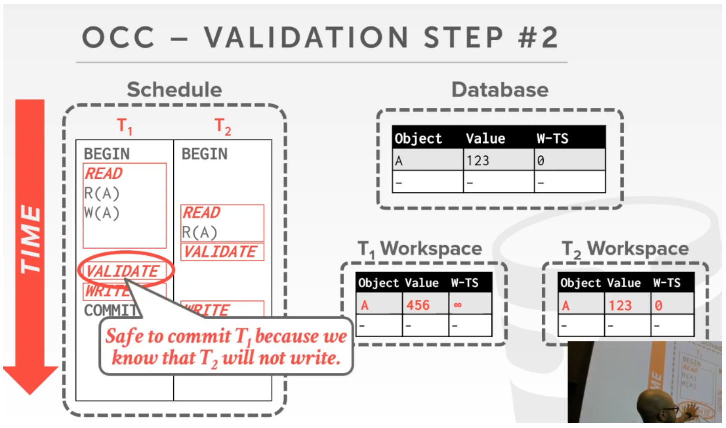
T1 Workspace

Object	Value	W-TS
A	456	∞
-	-	-

T2 Workspace

Object	Value	W-TS
A	123	0
-	-	-

42:39 – 42:42
But now let’s that slightly tweak it
现在让我们来稍微调整下这个例子
42.42–42.46
 and now we’ll have T2 begin validation before t1
我们让T2在T1之前先执行验证
42:46 – 42:50
so in the serial order t2 happens before t1
So，在Serial Order中，T2在T1之前执行
42:51 – 42:53
what’s gonna happen in this in this case right
在这个例子中，这会发生什么呢?
42.53–42.58
 ,so t1 reads a ,begins validation, it doesn’t have a write set
So，T1对A进行读取，接着开始验证，这里它并没有write set
42:58 – 43:01
so it doesn’t it doesn’t have anything to intersect with this transaction
So，它也就没有任何与这个事务相交的部分
43.01–43.02
,so so it commits

So，我们可以提交T1
43:03 – 43:04
t2 does the same thing
T2也做了相同的事情


Object	Value	W-TS
A	123	0
-	-	-

T_1 Workspace

Object	Value	W-TS
A	456	∞
-	-	-

T_2 Workspace

Object	Value	W-TS
A	123	0
-	-	-

43:09 – 43:17
it’s write set ,doesn’t intersect with the the read set of T, it doesn’t intersect with any other concurrent transaction
T2的write set不与其他并发事务中的操作相交
43:18 – 43:20
because it’s what this one’s all revalidated
因为它已经被重新验证了
43:21 – 43:23
so ~~another thing at like~~ nothing happens right
So，这里什么也没发生
43:23 – 43:23
So when the serial order
So，当我们是Serial Order的情况下
43.23–43.24
t2 happens before t1
T2在T1之前开始执行
43.24–43.28
it reads the value of a that initially existed before
它读取了A之前一开始就存在的值
43:28 – 43:31
This transaction is going to read the value of a there
该事务就会去读取A的值
43.31–43.34
make a local copy read write to it,
它会去制作A的一份本地副本，并对该副本进行读写操作
43.34–43.34
and everything’s okay
所有东西都是ok的
43:39 – 43:43
all right,questions, yes
有问题吗？请问
43:50 – 43:51
so in this scenario
So，在这种情况下

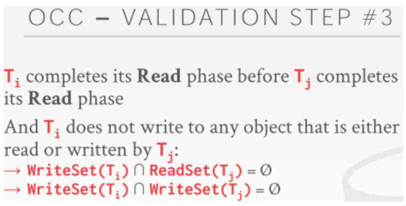
43.51-43.54
 because T2 begins validation first
因为T2先开始进行验证
43.54-43.56
it appears first in the serial order
它会先出现在Serial Order中
~~44:04 - 44:10~~
~~yeah yeah~~

44:16 - 44:20
so T1 has to abort
So，我们需要中止T1
44.20-44.24
because in the serial order what should have happened is that
因为在Serial Order中，这里应当会发生的事情是
44.24-44.27
 T2 because it begins validation first
因为T2先开始进行验证
44.27-44.29
, it's the first to recognize that there's something wrong
它会首先注意到这里出了问题
44:30 - 44:36
so it recognizes something's wrong as soon as it begins a validation
So，只要当它开始验证时，它就会意识到这里有东西出了错
44:36 - 44:37
because it gets assigned a timestamp,
因为它被分配了一个时间戳
44.37-44.40
 it sees a concurrent transactions that's read stale value
它看到了一个并发的事务，该事务读取到了一个过时的值
44.40-44.41
so it aborts
So，该事务被中止了
44:45 - 44:47
well when T2 goes to validate,
Well，当T2开始验证的时候
44.47-44.49
 T1 note is no longer a valid transaction
T1被打上标记，它不再是一个有效的事务
44:49 - 44:51
So it won't see any concurrent transactions,
So，它不会看到任何并发的事务
44.51-44.55
there's nothing to intersect with also it's a read-only transaction
这样也就没有其他事务会与其相交，即便它是一个只读事务
44:55 - 44:56
so there's nothing to intersect with
So，这里不会有任何事务与它相交
45:16 - 45:22

~~T2 couldn’t,~~ no not ~~until the~~ stalls it’s it’s changes
No，T2不会去暂停它所做的修改
45.20-45.22
because it’s a private private thing
因为它是一个私有的东西
45:25 – 45:28 – 45:28
yeah yeah yeah question
请讲
45:54 – 46:00
~~Yeah so when~~ when this T1 begins its validate phase
你是说当T1开始执行它的验证阶段的时候?
46:00 – 46:02
you don’t get a timestamp
你不会拿到时间戳
46.02-46.03
yeah I don’t get I don’t know 1 or 2 or something
你不会拿到1或者2之类的时间戳
46:04 – 46:07
but this T2 doesn’t have a write timestamp yet
但T2目前还没有write timestamp
46:08 – 46:09
yeah I mean it’s infinity.
我的意思是，它是无穷大的
46.09-46.11
we doesn’t have any timestamp
我们还没有分配任何时间戳
46.11-46.12
because it hasn’t begun validation yet
因为它还未开始验证操作
46:15 – 46:18
no no no so no no no no
No No No，你讲的不对
46:19 – 46:21
so the timestamp is at assigned to the transaction
So，我们会将时间戳分配给这个事务
46.21-46.30
~~it isn’t assumed that~~ it doesn’t take whatever timestamp is from the the tuple
我们并不会去使用tuple所携带的任何时间戳
46:30 – 46:32
yeah so that’s just a local copy
So，这只是一个本地副本
46:33 – 46:34 – 46:40 – 46:41
okay yeah yeah yeah
请说
46:56 – 46.57
are you saying
你说的是不是这么一回事
46.57-47.01
if t2 performs a write to this object A

如果T2对A执行写操作
47:02 – 47:08
 it’ll modify thewrite timestamp of its local copy to infinity
它会将它本地副本上的write timestamp修改为无穷大
47:10 – 47:10
yeah

47:25 – 47:26
Yeah so if it makes a write
So，如果它执行了这个写操作
47.26–47.29
~~it will get a local time~~ it’ll get a timestamp in’ validation
它会在验证的时候拿到一个时间戳
47.29–47.34
 and it’ll update the write timestamp there，If there’s no conflicts yeah
如果这里没有任何冲突的话，它会更新这里的write timestamp

47:58 – 47.58
no no
No No
47.58–48.00
you have to have this write be atomic
你所拥有的这个写操作必须是原子性的
48.00–48.04
so you write all all your changes back into into the database in this phase
So，你会在这个阶段将你所做的所有修改都写回数据库
48:11 – 48:14
that’s just because of the schedule that we have here
这只是因为我们这里所拥有的这个schedule
48:14 – 48:17
~~so we have the device~~ you can think of it as like we begin the validation
So，你可以想象一下，当我们开始验证的时候
48.17–48.19
we finish and do context switch here
当我们执行完毕，进行上下文切换到T1这里的时候
48.19–48.28
 and now we begin the validation and the write phase
我们就会开始执行验证，以及这个写阶段

OCC – 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:

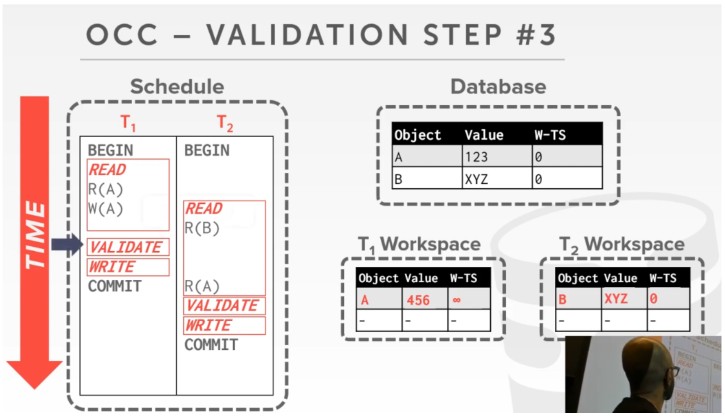
→ $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \emptyset$

→ $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j) = \emptyset$

48:28 – 48:28
all right

48.28–48.30
so this is the second example
So，这是第二个例子

48.30–48.33
there’s still one more to go
我们还要再看一个例子
48:34 – 48:34
in this one
在这个例子中
48.34–48.44
the transaction you have to ensure that the transaction TI completes its read phase before transaction TJ,begins its read phase
你需要确保在Tj开始它的读阶段开始之前，Ti完成了它的读阶段
48:44 – 48:55
so we have to ensure that the write set of my transaction doesn’t intersect with the the read set and the write set of all these of the transactions that fall into this category or the read phases are overlapping
So，我们需要确保我们事务中的write set不与其他事务的read set和write set相交，或者不与读阶段重叠


OCC – VALIDATION STEP #3

Database

Object	Value	W-TS
A	123	0
B	XYZ	0

T_1 Workspace

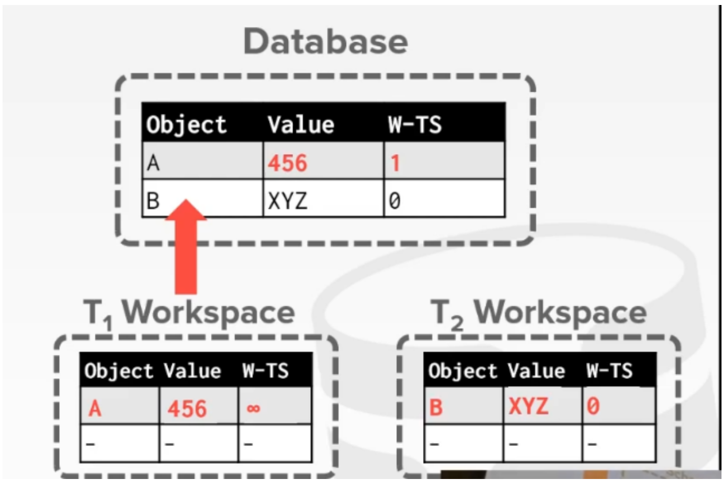
Object	Value	W-TS
A	456	∞
-	-	-

T_2 Workspace

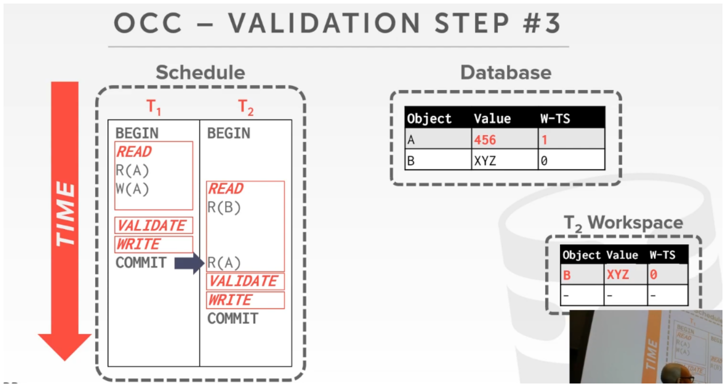
Object	Value	W-TS
B	XYZ	0
-	-	-

48:56 – 49:01
and this is another example of how we can go through this
这是我们要看的另一个例子
49:02 – 49:04
so let’s see in this scenario
So，在这个场景下
49.04–49.06
t1 is reading and writing a
T1对A进行读写操作
49:07 – 49:08
oops
不好意思，按错了
49.08–49.10
t2 is reading B and reading a,
T2先读取B，再读取A
49.10–49.14
t1 and begin that validation phase
T1开始执行它的验证阶段
49:14 – 49:16
so it gets a timestamp of 1
So，它拿到的时间戳是1
49.16–49.19

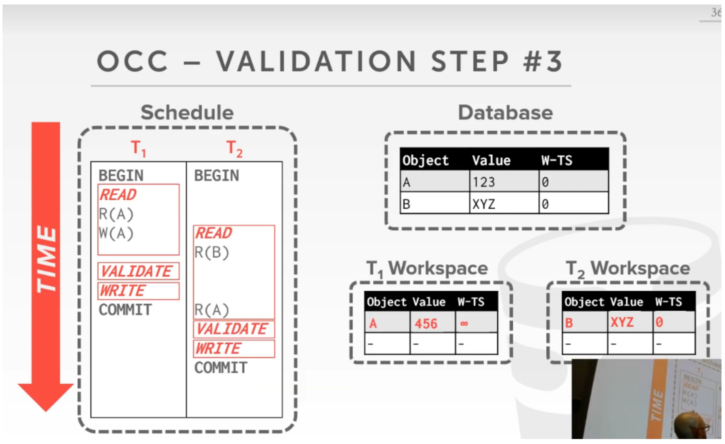
,and now I can commit it
我可以提交T1
49.19–49.27
,because my write set doesn’t intersect with the read set of transaction t2 at this point in time
因为T1中的write set此时并没有与T2中的read set相交
49:29 – 49:30
okay

49.30–49.34
 so I can install my changes back into the into the database
So，我可以将我所做的修改落地到数据库中


49:35 – 49:38
I update 4 5 6 or update A with value 4 5 6
我将A的值更新为456
49.38–49.39
and I set my write timestamp to 1
并将它的write timestamp设置为1
49.39–49.45
which is the timestamp that I was assigned at validation phase
这也是我在验证阶段所分配的时间戳
49:45 – 49:45
okay

49.45–49.51
then when we come back down into we do a context switch back into t2
接着，我们将上下文切换到T2


49:51 – 49:54
t2 does a R(B) or sorry a R(A)
接着，T2对A进行读取
49.54–49.58
,and now ~~it can go back to the main~~ it goes to the to the main database
现在，它跑到主数据库那里
49:58 – 50:03
and does a local copy of the updated value of A made by t1
读取到由T1更新过的A的值的本地副本
50:04 – 50:06
and then it can validate and write successfully
接着，它可以进行验证，并成功写入数据
50.06–50.08
,because there are no concurrent transactions
因为这里并没有任何其他的并发事务
50:11 – 50:13
Good yeah
都懂了吧，请讲
50:22 – 50:26
so if you let’s go back to this example
So，我们回到这个例子



OCC – VALIDATION STEP #3

Schedule

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	
W(A)	READ
	R(B)
VALIDATE	
WRITE	
COMMIT	R(A)
	VALIDATE
	WRITE
	COMMIT

Database

Object	Value	W-TS
A	123	0
B	XYZ	0

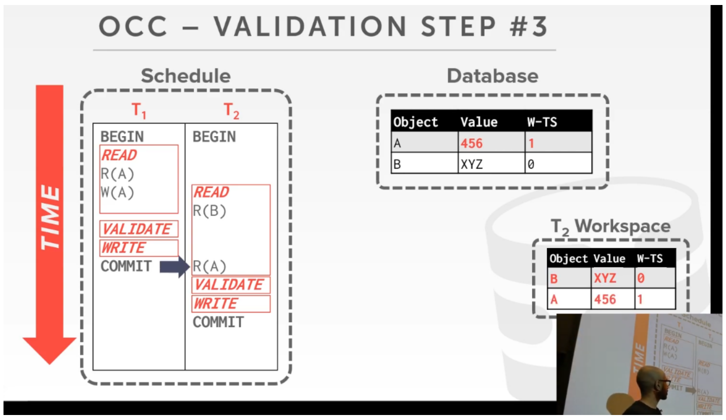
T_1 Workspace

Object	Value	W-TS
A	456	∞
-	-	-

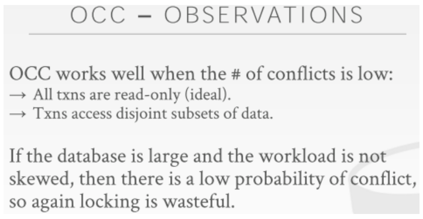
T_2 Workspace

Object	Value	W-TS
B	XYZ	0
-	-	-

50:28 – 50:30
so when you want to perform a write
So，当你想执行一次写操作的时候
50.30–50.36
 right you make a local copy of the database object into your local private workspace
你会制作该数据库对象的一份副本，并保存到你的本地私有工作空间中
50:36 – 50:38
and then that’s why you set the timestamp to infinity
这就是你将时间戳设置为无穷大的原因所在
50:38 – 50:39
you set it to infinity
你将它设置为无穷大的原因是
50.39–50.41
 ,because you don’t know what your timestamp is going to be
因为你不知道你的时间戳会是什么
50:41 – 50:44
because you haven’t been assigned one yet, right

因为你还未分配这个时间戳
50:45 – 50:49
it’s zero here only for mainly for illustrative purposes
这里我们之所以写0，只是出于展示原因
50.49–50.55
, you can think about like there was some transaction with timestamps 0 ,that bulk loaded all of these objects into the database
你可以想象一下这种场景，某个时间戳为0的事务将所有的对象加载到数据库中
50:55 – 50:57
so that’s why the write timestamp is 0
So，这就是为什么write timestamp是0的原因了
51:00 – 51:00
when you read it
当你读取它的时候
51.00–51.11
you read it whatever timestamp is associated with the object at that point
该对象的read timestamp就与你读取它的时间相关


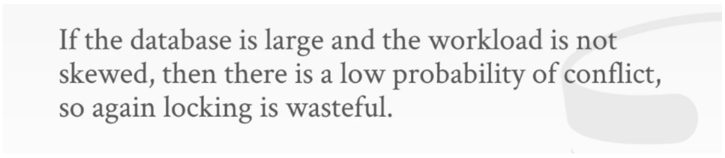
51:11 – 51:14
yeah so t2 does a R(A)
So，T2执行了R(A)
51.14–51.16
does a valid does a validation does a write
并执行了验证以及写操作
51.16–51.17
everything is good
这一切都没问题
51.17–51.18
because there’s no concurrent transactions
因为这里并没有其他的并发事务
51:19 – 51:19
yeah
请讲
51:23 – 51:26
yeah these are all for validation yeah correct
这些都是为了进行验证，说的没错
51:34 – 51:37
yeah you still need to rely on the same timestamps
没错，你依然需要依靠这些相同的时间戳


OCC – OBSERVATIONS

OCC works well when the # of conflicts is low:
→ All txns are read-only (ideal).
→ Txns access disjoint subsets of data.

If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.

51:44 – 51:53
yeah so that was essentially the I'm cup we've been kind of hind wavy on a couple of things on, how you actually install the write atomicly, and how you do the validations in a parallel way
So，我们已经讲了两个东西，即我们该如何保证我们的写操作原子性，以及我们该如何以并行的方式进行验证
51:53 – 51:54
because in a real system
因为在一个真正的系统中
51.54–51.56
 you have to do this parallel validation
你需要做这种并行验证
51:56 – 51:59
but I think hopefully the main idea has gotten across right
但我希望你们已经理解了这里面的主要思想
52:00 – 52:01
it's not too complicated
它并不是很复杂
52.01–52.03
there's some trickeries here
这里面有些棘手的地方
52.03–52.06
but generally it's it's pretty intuitive
但总的来讲，它还是比较直接的
52:07 – 52:10
so there's a few observations that we should try to make
So，我们从中学到了一些东西
52:11 – 52:18
all of these timestamp in optimistic concurrency protocols sort of work well when there's very little conflict right
当没有什么冲突的情况下，乐观并发协议中这些基于时间戳的协议的效果非常好
52:18 – 52:25
because you allowed the transactions to proceed without acquiring locks without doing that anything heavyweight it worked
因为这允许我们在不获取lock以及不做那些代价很高的操作的情况下，我们能够去处理这些事务
52.25–52.30
 and at the end you do a sort of lightweight semi lightweight validation to make sure that your transaction is still valid
在最后，我们会做些轻量级或者说半轻量级的验证来确保我们的事务是有效的
52:31 – 52:35
so if you have very few very few conflicts
So，如果你执行的事务中几乎没有什么冲突
52.35–52.41
even better if all your transactions are essentially just doing read-only work

更好的情况是，如果你的事务所做的都是只读型工作
52:41 – 52:43
and if the access disjoint sets of data
并且它们访问的都是不相交的数据集
52.43–52.45
then this these protocols work very well
那么，这些协议的效果就会非常好
52.45–52.48
because they’re actually not doing much work at all aside from doing some local data copying
因为它们除了会去制作本地数据副本以外，它们并不会做太多工作
52:49 – 52.53
Right，another way to think about this is that
思考它的另一种方法是

If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.

52.53–52.57
 if you have a very large workspace and very small transactions
如果你的数据库很大，并且你要执行的事务数量也不多
52.57–53.02
 and the probability of these transactions overlapping in the read and write set is very low
同时，这些事务在read set和write set上重叠的可能性又比较低

53:02 – 53:03
in this specific scenario
在这种特定情况下
53.03–53.06
 these optimistic concurrency protocols work very well
这些乐观并发控制协议的效果会非常好
53:07 – 53:08
on the other hand
另一方面
53.08–53.11
in very highly contentious workloads
在那些存在着很多冲突的workload中
53:11 – 53:12
what ends up happening is that
最终会发生的事情是
53.12–53.14
 you have transactions consistently restarting right
你的事务始终会反复重启
53:14 – 53:19
so they do a lot of work in their work phase
So，在它们执行阶段，它们会做大量的工作
53:20 – 53:22

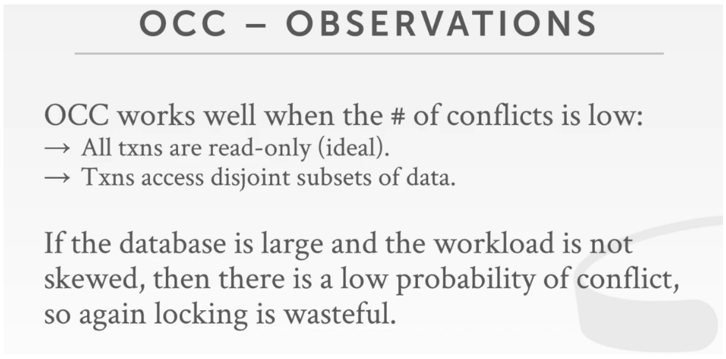
assuming if there is no contention
假设，如果这里面没有冲突的情况发生
53.22–53.27
 and then and only at the very end do they figure out oh crap all this work that I’ve done is kind of useless after we start
只有到了最后，它们发现：Oh，自我开始执行后，我所做的所有工作都是无谓的消耗
53:27 – 53:33
right so you’re kind of deferring a lot of the heavy lifting towards the end
So，你就会将很多繁重的工作都放到最后去做
53.33–53.35
 assuming that there won’t be a lot of heavy lifting
假设这里并没有那么多繁重的工作
53:35 – 53:38
but in the contentious work though there is heavy lifting
但在冲突很多的workload中，这里面就存在着那些繁重的工作
53.38–53.41
because your sense they have to abort everything you’ve done
因为它们必须中止你已经完成的所有东西
53:41 – 53:44
so really I think the the the research sort of shows that
So，这里的研究表明
53.44–53.46
 in contentious workloads
在这些冲突很多的workload中
53.46–53.53
both two-phase locking and optimistic concurrency protocols generally don’t work,
两阶段锁和乐观并发协议通常来讲都不可行
53.53–53.54
they’re sort of like almost equivalent
它们的效果几乎是一样的
53:54 – 53:58
so when 2PL ~~simply acquiring there’s~~ there’s a lot of contention on locks
So，在两阶段锁中，它里面存在着大量争抢锁的情况
53:59 – 54:02
so you have transactions waiting and these hot locks
So，你需要让事务去等待获取这些热门的lock
54.02–54.07
here you have transactions that are doing all their work almost wasteful work ,and then aborting at the very end
这里你会遇上那些做了很多无用功的事务，它们在最后的时候被中止
54:07 – 54:10
so neither really work for contentious workloads
So，这对于那些存在着冲突的workload来说，该协议的效果并不好
54:11 – 54:13
but if your workload is it has very low contention
但如果你的workload中所存在的冲突数量并不多
54.13–54.17
, then in general these optimistic protocols work better
那么，通常来讲，使用这些乐观协议的效果会很不错

54:17 – 54:20
because they have no overhead
因为它们并没有什么开销
54.20-54.23
 and very very minimal override aside from some local data copy
并且它们对本地数据副本也只做最小程度的覆写
54:23 – 54:25
okay whereas in the 2PL protocol
然而在两阶段锁中
54.25-54.26
you actually have to acquire locks
实际上，你必须去获取这些lock
54.26-54.30
, even if there’s no logical reason that you should be acquiring these locks
即便没有什么逻辑上的理由，你也应该去获取这些lock
54:36 – 54:36
Yeah

54:53 – 54:56
so the timestamps are assigned at validation
So，我们会在验证的时候分配这些时间戳
54:57 – 54:57
yeah

55:19 – 55:22
I think what you’re describing is essentially basic timestamp order ,right
我觉得你所描述的本质上来讲就是Basic Timestamp Ordering
55:23 – 55:27
so in basic timestamp ordering，you assign the timestamps when the transaction begins
So，在Basic Timestamp ordering中，你会在事务开始的时候分配时间戳
55:27 – 55:28
In OCC
在OCC中
55.28-55.34
the protocol dictates that you assign the timestamps at validation time
该协议会让你在验证阶段的时候分配时间戳
55:34 – 55:39
because you want to defer the work of doing the actual checking to when you’re ready to commit
因为你会将这些实际的检查工作延迟到你准备提交事务的时候再做
55:39 – 55:46
whereas basic timestamps ordering is saying ,every operation I’m going to perform, I want to do a check to make sure that this is a valid operation
Basic timestamp ordering表示，我会对我要执行的每个操作进行检查，以确保这是一个有效操作
55:47 – 55:48
so just how the protocol works
So，这就是该协议的工作方式
55:50 – 55:50

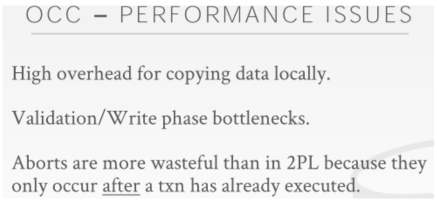
yeah

55:58 - 56:04
you still need to make a local copy even if you’re reading to ensure repeatable reads
即使你是在进行读操作的时候，你也依然需要制作一份本地副本以支持可重复读

OCC – OBSERVATIONS

OCC works well when the # of conflicts is low:
→ All txns are read-only (ideal).
→ Txns access disjoint subsets of data.

If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.

56:11 - 56:18
right so this slide is essentially just saying that for low contention workloads, these optimistic concurrency protocols work very well
So，这张幻灯片上的内容是，当你workload中存在的冲突数量不多时，这些乐观并发协议的效果就会很好
56:18 - 56:19
because they have very little overhead
因为它们几乎没有什么开销
56.19-56.22
 in comparison to two-phase locking
比起两阶段锁来说
56.22-56.26
which you know even if you have disjoint working sets between transactions
即使你在事务间的working set是不相交的
56:26 - 56:28
you still have to have transactions require locks
你依然会要求这些事务去获取锁
56.28-56.31
 ,whereas here there’s no locks at all
但如果是OCC的话，就根本不需要用到锁
56:45 - 56:47
this guy yeah you’re right
你说的没错
56:47 - 56:50
so you don’t have to acquire like a database of a lock
So，你不需要去获取数据库中的lock
56.50-56.51
, you still have to require latches
你依然需要去获取latch
56.51-5656
, you still have to make sure the integrity of the data structure is valid
你依然需要去确保该数据结构的完整性是有效的
56:56 - 56.59
Right you can’t just blindly overwrite stuff in memory
你无法盲目地去覆写内存中的数据

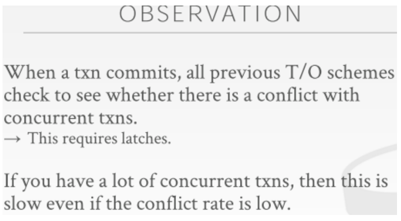
56.59-57.02
,because there could be concurrent access to these structures
因为这可能存在着并发访问这些结构的情况

OCC – PERFORMANCE ISSUES

High overhead for copying data locally.

Validation/Write phase bottlenecks.

Aborts are more wasteful than in 2PL because they only occur after a txn has already executed.

57:02 – 57:06
so I think you’re kind of hitting on the point that we will talk about it I think in this slide
So，我觉得你讲到我们要讲的重点了，应该是在这个幻灯片上
57:08 – 57:12
yeah so there is a little bit of work overhead in OCC right
So，使用OCC的时候，我们会有一些开销
57:12 – 57:15
because you have to maintain local copies of everything that you want to read and write
因为你需要去维护你想要进行读写操作的那些对象的本地副本
57:15 – 57:16
so if you’re updating a billion tuple
So，如果你要更新十亿个tuple
57.16-57.17
you have to make a billion copies
你就必须制作十亿个副本
57.17-57.20
every transaction has to make a billion copies
每个事务都必须制作十亿个副本
57:20 – 57:22
and so there’s a lot of overhead here
So，这里面就存在着大量的开销
57:23 – 57:27
the other thing is that the validation and the write phase is, they were happening serially right
另一件事情是，验证阶段和写阶段会按顺序执行
57:28 – 57:30
the only one transaction can be validating at a time
一次只会有一个事务进行验证
57:31 – 57:33
in reality you know in real systems
在真正的系统中
57.33-57.35
you have parallel validation and parallel writing
你可以进行并行验证和并行写
57:35 – 57:39
but again even those in those scenarios these phases become a big bottleneck
但在这些情况下，这些阶段会变成一个巨大的瓶颈
57:41 – 57:42
and lastly
最后
57.42-57.45

because we’re being optimistic
因为我们使用的是乐观方案
57.45–57.51
we’re assuming that we can perform all of the work with ,you know safely without being interrupted by other transactions
我们假设我们执行的所有操作都不会被其他事务所打断
57:51 – 57:53
But in contentious workloads
但在那种存在冲突的workload中
57.53–57.54
that’s not the case
这就不会是这种情况了
57.54–57.55
right that assumption is invalidated
这种假设就是无效的了
57:55 – 57:58
so in contentious workloads routing all this work upfront
So，在这些存在着冲突的workload中，我们会将所有的工作放在眼前
57:59 – 58:01
and then we could find out later that we actually have to abort
然后，我们就能找出我们需要中止的那些工作
58:01 – 58:03
so all of this work is wasted
So，所有这些被中止工作做的都是浪费的无用功
58.03–58.06
we’re being optimistic and we’re being wrong about that optimism
我们使用的是乐观方案，但我们对这种乐观的看法是错误的

OBSERVATION

When a txn commits, all previous T/O schemes check to see whether there is a conflict with concurrent txns.
→ This requires latches.

If you have a lot of concurrent txns, then this is slow even if the conflict rate is low.
58:11 – 58:13
so when a transaction commits
So，当一个事务要提交的时候
58.13–58.21
I think that the student was pointing this out correctly
我认为之前有学生正确地指出了这一点
58:21 – 58:25
 even if the transactions logically don’t overlap with each other
从逻辑上来讲，即使这些事务彼此不会重叠
58:26 – 58:31
you still have to make sure that you maintain the physical integrity of the data structure you’re gonna be looking at
你依然必须确保你要去维护你所查看的数据结构的物理正确性（知秋注：不会因为并发操作，使程序中的一个对象内部数据状态发生异常）
58:31 – 58:34
so as part of the validation phase
So，在执行验证阶段的时候

58.34–58.40
you have to look at oh I have you know all these other thousand transactions, I have to look inside read and write sets in a consistent way
你会看到所有这些数以千计的事务，我不得不深入其中通过一种可以保证一致性的操作方式来查看read和write set
58:41 – 58:45
I make this consistent by acquiring latches right
我通过获取latch来让它们保持一致
58:46 – 58:49
so this latch overhead it can actually play a big role
So，这种获取latch的开销实际上扮演了一个重要角色
58.49–58.52
even if they’re logically just disjoint
即使从逻辑上来说，它们是不相交的
58.52–58.55
 physically they’re still contending on the same data structures
从物理上来讲，它们依然在抢夺同一个数据结构
58:55 – 58:59
right there so contending on the read and write data sets，even though logically they’re disconnected
即使在逻辑上read set和write set是不相交的，但它们依然存在着争抢问题(知秋注:如果并发验证，那就会可能同时有多条线程查看同一个事务中本地保存的数据，这就会需要latch)
59:00 – 59:01
yes
请讲
59.01–59.03
 so when I’m doing my validation right
So，当我在执行验证的时候
59:04 – 59:06
I have one transaction and me ,I’m ready to validate
我有一个准备进行验证的事务
59.06–59.09
I have to go into his read and write transaction set to make sure that we don’t intersect
我必须去查看它的read set和write set，以确保它们是不相交的
59:10 – 59:10
all right

59.10–59.11
but he could be modifying it
但它可以对数据进行修改
59.11–59.14
because he’s still running ,he or she’s still running right
因为这个事务依然在执行
59:14 – 59:15
so I have to acquire latch
So，我需要去获取latch
59.15–59.20
read a consistent view of it， perform the intersection to make sure I’m okay
~~并读取到该数据的一个一致的状态~~

以这种一致性的方式来对它操作，以保证可以执行的很OK

59:20 – 59:24
and this other transaction can’t modify that that said while I’m reading it
当我正在读取这个数据的时候，其他事务无法对该数据进行修改
59:29 – 59:34
but I’m reading the local copy, I’m reading that transactions local copy of the working set
你说的是，如果我读取的是该事务自己的本地副本数据?
59:37 – 59:44
Because the table doesn’t have all of the updates that have been applied that this transaction has read and written to write
因为数据库表中并没有更新该事务所做的所有更新
59:44 – 59:46
the transaction has a local copy of reads and writes
该事务对本地副本进行了读写操作
59:46 – 59:50
so to see if this transaction has read a value that I’m writing to
So，为了弄清楚该事务读取到的值是否是我正在写入的那个值
59:51 – 59:53
I only know that by looking at his read write set
我只有通过查看该事务的read set和write set，我才能知道这个
59:54 – 59:55
I don’t know that by looking at the table
看数据库表，我是看不出这些的