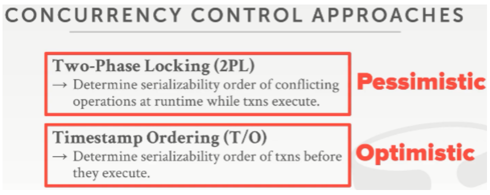
18–01

18 – Timestamp Ordering Concurrency Control (CMU Databases Systems _ Fall 2019)


CONCURRENCY CONTROL APPROACHES

Two-Phase Locking (2PL)
→ Determine serializability order of conflicting operations at runtime while txns execute.

Timestamp Ordering (T/O)
→ Determine serializability order of txns before they execute.

Pessimistic
Optimistic

00:20 – 00:27

all right so last lecture, I think Matt covered two–phase locking

So，我相信Matt在上节课的时候已经给你们介绍了两阶段锁方面的内容

00:28 – 00:36

And two–phase locking is a mechanism that the database can use to essentially generate serializable schedules at runtime

两阶段锁是一种机制，数据库可以通过它在运行时生成Serializable schedule

00:36 – 00:39

and it relies on locks to be able to do that

它依靠锁才能做到这些

00:39 – 00:43

today we’re going to be talking about is a collection of protocols that don’t rely on locks,

今天我们要讨论的是一些不依靠锁的协议

0.43–0.45

but instead rely on on timestamps

相反，它们依靠的是时间戳

00:47 – 00:47

at a high level

从一个高级层面来讲

0.47–0.50

maybe with a good way to think about this is that two phase locking

兴许，我们使用二阶段锁来思考它会是一种不错的方式

0.50–0.58

assumes that there’s gonna be a lot of contention inside of the transactions that are executing inside in the database

假设数据库中所执行的事务里面存在着大量的争抢情况

00:58 – 01:01

right so if there’s a lot of contention

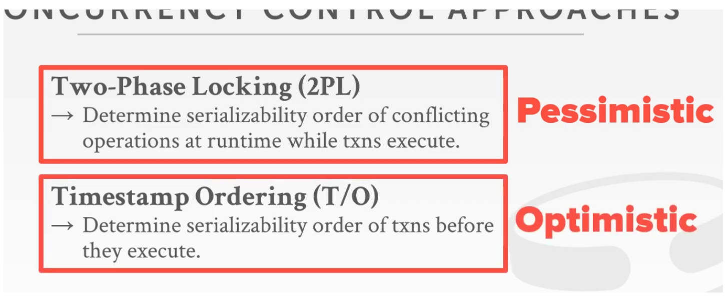
So，如果其中存在着大量的争抢情况

1.01–1.06

then it’s obviously I’ve edges to be defensive and take a lot of locks

那么，很明显，我就会变得很保守，并且使用大量的锁

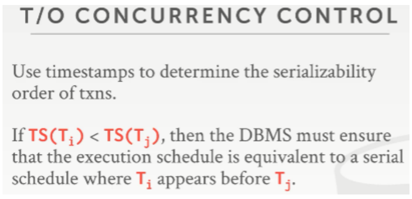
01:06 – 01:09

so anytime you want to read or write into into a database object
So，每当你想对某个数据库对象执行读或者写操作时
1.09–1.11
You acquire these locks
你就会去获取这些锁
01:12 – 01:12
so in that sense
So，在这种情况下
1.12–1.13
 it’s quite pessimistic
这是一种相当悲观的情况

Two-Phase Locking (2PL)
→ Determine serializability order of conflicting operations at runtime while txns execute.

Timestamp Ordering (T/O)
→ Determine serializability order of txns before they execute.

Pessimistic
Optimistic

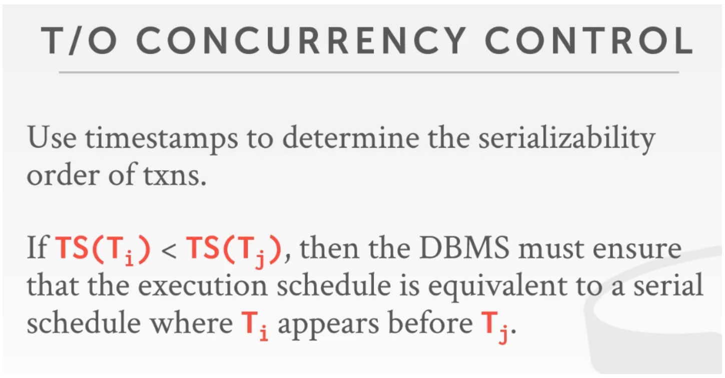
1.13–1.14
 on the other hand
另一方面
1.14–1.19
 you can view timestamp ordering based techniques as more optimistic right
你可以将基于时间戳顺序的方案视为一种更加乐观的方案
01:20 – 01:24
You allowed the database to operate ,and read and write data without actually acquiring locks
你允许数据库在不获取锁的情况下对数据进行读和写操作
01:24 – 01:25
and at the end of the day
最后
1.25–1.31
you’re able to correctly reconcile the correct serializable schedule at the end of the day
你能够正确地调整出正确的Serializable schedule
01:31 – 01:34
and we’ll talk about how this is actually done inside of the database
我们稍后会讲数据库内部是如何做到这点的
01:36 – 01:41
there’s actually going to be two times time ordering protocols, we’re going to talk about in this lecture
实际上，这节课上我们会讨论两种基于时间顺序的协议
01:41 – 01:44
One of them is actually called timestamp ordering or basic timestamp ordering
其中一种协议叫做timestamp ordering，或者叫做basic timestamp ordering
01:44 – 01:48
and another one is gonna be called the optimistic concurrency control
另一种叫做乐观并发控制
01:48 – 01:49

so it’s a little bit confusing
So，这有点令人困惑
1.49–1.50
, because they’re both optimistic
因为它们两个都属于乐观方案
1.50–1.51
and they’re both timestamp ordering
并且它们两个都是根据时间戳来做的
1.51–1.55
,this is just the the nomenclature that the community has come up with
这其实是社区所提出的术语

T/O CONCURRENCY CONTROL

Use timestamps to determine the serializability order of txns.

If $TS(T_i) < TS(T_j)$, then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where T_i appears before T_j.

01:56 – 1.57
all right

1.57–1.57
so let get started
So，我们开始上课吧
1.57–2.02
,the basic idea for these timestamp based protocols is that
这些基于时间戳的协议的基本思路是
2.02–2.03
it’s a mechanism
它是一种机制
2.03–2.07
that the database can use to assign numeric values to timestamps
数据库可以通过这种机制来分配时间戳
2.07–2.11
that predefined the commit order of these transactions
以此来预定义这些事务的提交顺序

T/O CONCURRENCY CONTROL

Use timestamps to determine the serializability order of txns.

If $TS(T_i) < TS(T_j)$, then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where T_i appears before T_j.

02:11 – 02:14
you can assume that there’s going to be a new function here called TS
假设这里有一个叫做TS的新函数
2.14–2.18
,that given a transaction gives you the timestamp for that transaction
该函数会返回给我们该事务的时间戳

02:18 – 02:21

And so what the database tries to guarantee is that

So，数据库试着保证的东西是

2.21–2.25

 if a transaction (Ti) has a timestamp that’s less than the transaction (Tj)

如果事务Ti的时间戳小于事务Tj的时间戳

02:25 – 02:26

then in the serial schedule

那么，在Serial Schedule中

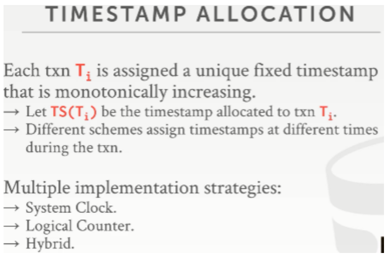
2.26–2.30

it’s as if (Ti) had occurred before (Tj) inside of the database

在数据库中，Ti会在Tj之前执行

2.30–2.30

okay



TIMESTAMP ALLOCATION

Each txn T_i is assigned a unique fixed timestamp that is monotonically increasing.
→ Let $TS(T_i)$ be the timestamp allocated to txn T_i.
→ Different schemes assign timestamps at different times during the txn.

Multiple implementation strategies:
→ System Clock.
→ Logical Counter.
→ Hybrid.

02:32 – 02:34

So how is this done

So，这是如何做到的呢?

2.34–2.35

like what are these timestamps look like

这些时间戳长啥样呢?

2.35–2.40

the timestamps are sort of unique fixed numeric values

这些时间戳其实是唯一且固定数字

02:40 – 02:43

And they have a couple of interesting characteristics

并且，它们还有两种令我们感兴趣的特性

2.43–2.45

 that characteristics that the database has to maintain

数据库必须维护这些特性

02:46 – 02:47

The first one is that

第一种特性就是

2.47–2.50

these timestamps have to be monotonically increasing

这些时间戳必须是单调增加的

02:50 – 02:53

okay so they have to always go forward in time and increase in time

So，它们的值必须随着时间的流逝而增加

02:54 – 02:55

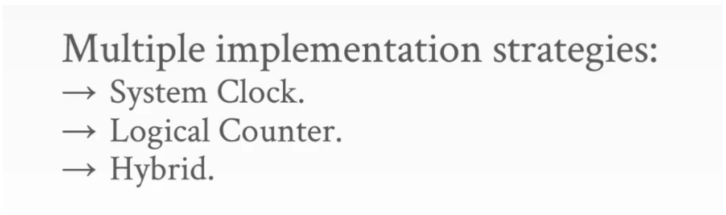
the second thing is that

第二种特性则是

2.55–2.57

 they have to be unique values right

它们的值必须是唯一的
02:57 – 03:00
so you can’t ever have two transactions have the same timestamp
So，你永远不可以拥有两个具备相同时间戳的事务
03:01 – 03:03
because of this monotonically increasing characteristic
因为它的特性就是单调增加
03:04 – 03:08
so again assume that the database now has this new function this TS function
So，假设数据库现在有一个叫做TS的新函数
3.08–3.12
, that’s able to take a transaction ID and return to you the timestamp for that transaction
它通过接收一个事务ID，它能够给我们返回该事务的时间戳
03:13 – 03:19
and these different timestamp protocols have different mechanisms
不同的时间戳协议它们拥有的机制也不同
3.19–3.25
~~by which~~ at time points at which they actually assign these timestamps to the transactions
从事务执行的时候，它们会将这些时间戳分配给事务
03:25 – 03:27
an important characteristic is that
一项重要的特性就是
3.27–3.30
 these timestamps don’t necessarily have to correspond to the wall clock time
这些时间戳不一定要和挂钟时间所对应
03:32 – 03:36
because they could be assigned to the transaction at any point during its execution
因为我们可以在事务执行期内的任意时间点给事务分配时间戳
3.36–3.38
not necessarily when it enters the system
不一定是要在刚拿到事务的时候给它分配时间戳
3.38–3.39
not even necessarily when it’s about to commit
也不一定要在事务提交的时候分配时间戳
03:40 – 03:41
right so different protocols have different mechanisms
So，不同的协议有着不同的机制
3.41–3.45
 by at time points at which they actually assign timestamps to transactions
它们会在某个时间点将时间戳分配给事务
03:47 – 03:52
 there’s a few different ways on how you actually source a timestamp for a transaction
实际上，你可以通过一些不同的方法来确定某个事务的时间戳是什么


Multiple implementation strategies:
→ System Clock.
→ Logical Counter.
→ Hybrid.

03:53 – 03:55
okay and I've listed this a few listed here,
Ok，这里我已经列出了一些
3.55-3.57
 the simplest thing you could do is just ask the CPU
最简单的办法就是去问CPU
03:57 – 03.58
okay what is the current time right,
当前时间是什么
3.58-3.59
because time is always increasing
因为时间总是在流逝的
3.59-4.02
you can assume that this kind of makes sense
你可以假设它是具备一定意义的
04:02 – 04:03
all right
4.03-4.05
but there's a few drawbacks
但这也存在着一些缺点
4.05-4.11
can anybody think about a few drawbacks for using wall clock time or a real time as a timestamp
如果使用挂钟时间或真实时间作为时间戳的话，你们能思考下这里面存在着哪些缺点吗?
04:11 – 04:11
yeah
请讲
04:17 – 04:19
so that doesn't really matter
So，这并不重要
4.19-4.20
,as long as you're going to the same computer
只要你用的是同一台机器
4.20-4.23
and as long as the time is monotonically increasing
并且它上面的时间是单调增加的
4.23-4.24
 then you should be okay
那么这就没什么问题
04:25 – 04:26
right so that's a good point right
So，你讲的不错
4.26-4.29
so if you have distributed database system
So，如果你使用的是分布式数据库系统
4.29-4.31
 and it's difficult to keep these time points in sync
你就很难保证时间是同步的

04:31 - 04:33
what’s another problem with using wall clock time
使用挂钟时间的另一个问题是什么?
04:37 - 04:39
you can turn back the clock
你可以将时间往回调
4.39-4.42
 when would you actually turn back the clock
你在什么情况下才会将时间往回调?
04:48 - 04:48
Sure yeah
说的没错
4.48-4.52
so there could be some skew in the actual granularity which you track time
So，在你机器进行同步时间的时候，时间轴可能回调
4.52-5.06
is there another and another possible drawback, yeah
还有其他缺点吗？请讲
05:06 - 05:08
so you don’t actually have to keep duration of time
So，实际上，你无须去保存这些持续时间
5.08-5.13
, you just need like one point like this is time point one time point two and it’s increasing
你只需保存时间点即可，比如这是时间点1、时间点2，以此类推，它们的值是单调增加的
05:15 - 05:17
so it’s not necessarily by duration is just about a point in time
So，你不用去保存这个事务持续的时间，你只需保存时间点即可
05:19 - 05:21
okay so a clue is that it’s gonna happen this weekend
So，就好比这周的夏令时
05:23 - 05:24
yeah exactly
没错
5.24-5.29
yeah so a day like say day that daylight savings right, so it could be the case that you know you’re operating on the weekend
So，就拿这周的夏令时为例，你要在周末的时候调整下你的时钟
05:29 - 05:31
And then at random at a at a random point
在随机某个时间点
5.31-5.33
your clock back goes back an hour
你的时钟往回调了1小时
5.33-5.35
and your timestamps are pretty much screwed
那么你的时间戳就完蛋了
5.35-5.35
okay
懂了吧

05:38 - 05:41
another option is actually to use these logical counters right
另一个选项则是使用这些逻辑计数器
05:41 - 05:50
so you can think of having just a register and the CPU dedicated to having a mono time monotonically increasing 32-bit 64-bit value
So，假设你CPU中有一个专门用来保存时间的寄存器，它里面的值是单调增加的，这些值的长度是32位或者64位

05:50 - 05:54
are there any drawbacks or potential downfalls for this approach
这种方案是否存在任何缺点呢?
5.54-5.55
yeah
请讲
06:03 - 06:06
yeah so the distribution aspect is still an important factor
So，分布式方面依然是一个重要因素
06:07 - 06:09
but assume that there's one counter for the CPU
但假设CPU中有一个计数器
6.09-6.12
,and it's really fast to increment it without requiring locks
在不需要用到锁的情况下，它可以快速增加它的值
06:13 - 06:15
like you can do an atomic addition or something like that
比如，你可以进行原子性相加或者类似的操作
6.15-6.18
are there any problems with using this logical counter
在使用逻辑计数器时，还有什么问题吗
06:23 - 06:25
so I said 32 bits or 64 bits
So，我说了它的值长度是32位或者64位的
06:27 - 06:28
what happens if you run out of 32-bit values
如果你的值超过了32位，这会发生什么呢?
6.28-6.33
then you saturate your edition and you roll back right
那么，你就会用完你的时间戳，你可以将它回滚
06:33 - 06:34
so your counter is now going backwards in time
So，到了那个时候，你的计数器就会往回走了
06:35 - 06:37
so that's one of the problems with this approach
So，这是该方案中存在的其中一个问题
06:39 - 06:40
so most systems actually use this hybrid approach
So，实际上，大部分系统使用的都是混合方案
6.40-6.42

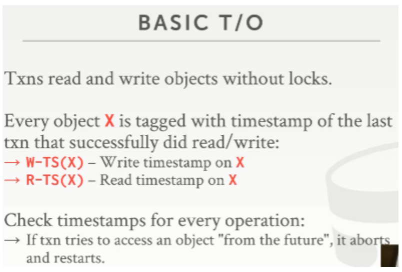
 which is like a physiological thing
这就像是生理上的一种东西
6.42-6.47
and it sort of matches both the physical counter, and a logical counter to make sure that everything sort of works out
通过让时间戳和物理计数器以及逻辑计数器进行匹配，以此确定所有东西都是正常工作的
06:52 - 06:55
so the system clock one problem is you have daylight savings times right
So，如果使用系统时钟，你会遇到的其中一个问题就是夏令时和冬令时导致的时间调整
06:55 - 06.58
so at a point in time one day like say daylight savings occurs
So，比如夏令时发生的那天
6.58-7.02
your time you time move backs or moves back an hour
你的时间就会往回调1小时
07:03 - 07:05
so now your time is not monotonically increasing it's not going back in time
So，现在你的时间就不再是单调增加的了，因为它往回调了一小时
07:08 - 07:08
okay

TODAY'S AGENDA

Basic Timestamp Ordering Protocol
Optimistic Concurrency Control
Partition-based Timestamp Ordering
Isolation Levels
07:11 - 07:15
right so just to give you an idea of what the agenda is gonna be today
So，来给你们看下我们今天要讲的内容
07:15 - 07:18
So we're gonna talk about something called the **basic timestamp ordering protocol**
So，我们会先讲一个叫做**basic timestamp ordering protocol的东西**
7.18-7.22
,then we're going to talk about **optimistic concurrency protocol**
接着，我们会讨论乐观并发协议
7.22-7.24
,which is also a timestamp based protocol
它也是一个基于时间戳的协议
07:24 - 07:27
and then we're going to talk about a **partition-based timestamp ordering** protocol
接着，我们会讨论一个叫做**partition-based timestamp ordering** protocol的东西
7.27-7.31
which alleviates some of the bottlenecks than regular timestamp ordering protocols have
它缓解了那些基于时间戳顺序的协议所带来的瓶颈
07:31 - 07:33
and then we're going to talk about **isolation levels**
然后，我们会去讨论隔离级别
7.33-7.34
okay

07:35 - 07:35

so let’s get started
So，我们开始吧


BASIC T/O

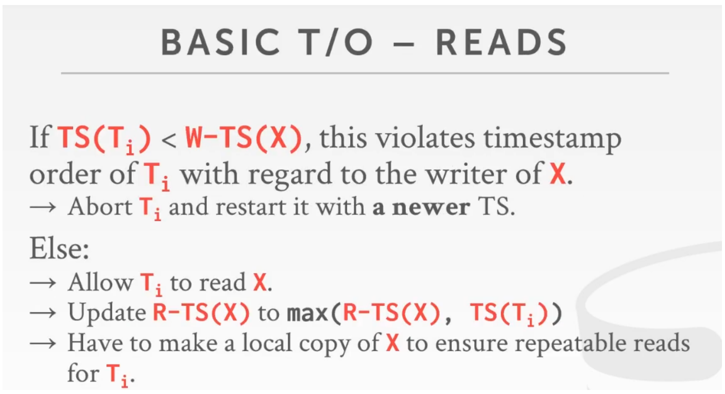
Txns read and write objects without locks.

Every object X is tagged with timestamp of the last txn that successfully did read/write:
- → W-TS(X) – Write timestamp on X
- → R-TS(X) – Read timestamp on X

Check timestamps for every operation:
- → If txn tries to access an object "from the future", it aborts and restarts.

07:36 – 07:40
so with these timestamp ordering schemes, the general idea is that
So，这些基于时间戳顺序的方案的基本思想是
7.40–7.45
you want the transactions to proceed in the system ,reading and writing objects without actually acquiring locks
在不获取锁的情况下，你想让系统中的事务能够对数据库中的对象进行读写操作
07:46 – 07:48
okay to make this possible
Ok，为了让它成为可能
7.48–7.53
you know you have to add some extra metadata to all of the database objects to make this possible
你需要往所有的数据库对象中添加一些额外的元数据，以此来让它成为可能
07:53 – 07:58
specifically you have to add two extra timestamps to every single tuple in the system
特别是，你需要往数据库系统中每个tuple上都要添加两个时间戳
07:58 – 08:06
you have to add a read timestamp which represents timestamp of the transaction the most recent transaction then read this item
 你需要往该对象上添加一个read timestamp，它表示的是最近读取该对象的事务的时间戳

08:06 – 08:07
and a write timestamp
接着，你还要添加一个write timestamp
8.07–8.12
which is the timestamp of the most recent transaction ,that wrote into this this tuple and in the system
它表示的是该系统中最近对该tuple进行写入操作的那个事务的时间戳
08:12 – 08:16
And then as the transaction is going through its operations
接着，当事务执行它里面的操作时
8.16–8.22
it’s just going to make sure that it can actually read this tuple by leveraging the timestamps that are associated with this tuple
它需要确保它可以利用与该tuple相关的时间戳来读取这个tuple



BASIC T/O – READS

If $TS(T_i) < W\text{-}TS(X)$, this violates timestamp order of T_i with regard to the writer of X.
→ Abort T_i and restart it with **a newer** TS.

Else:
→ Allow T_i to read X.
→ Update $R\text{-}TS(X)$ to $\max(R\text{-}TS(X), TS(T_i))$
→ Have to make a local copy of X to ensure repeatable reads for T_i.

08:23 – 08:25
Okay now and I'll talk about how this is done
Ok，现在我来谈论下它是如何做到的
08:26 – 08:38
so for reads
So，对于读操作来说
8.38–8.33
there's an invariant that you have to make sure before you're actually allowed to read a value from the database system
实际上，在你被允许从数据库系统中读取一个值前，你必须确保它是一个不变量
08:34 – 08:36
you have to ensure that the timestamp
你必须确保这个时间戳
8.36–8.38
,so you read your own timestamp for this transaction Ti
So，你要去读取事务Ti中你自己的时间戳
08:38 – 08:43
you have to make sure that it's less than the write timestamp for this tuple in the system
你需要确保这个时间戳不小于该系统中这个tuple所对应的write timestamp
08:44 – 08:46
okay does that make sense
Ok，你们懂了吗?
8.46–8.47
what does that mean,
这意味着什么呢
8.47–8.48 ！！！！…
that means
这意味着
8.48–8.52
that's essentially making sure that there's no other transaction that's written into the system
本质上来讲，这确保了该系统中没有其他事务对该tuple进行写入操作

08:54 – 8.59
yeah there's no other transaction in the system ,that's wrote into this tuple,
没错，该系统中没有其他事务对该tuple进行写入操作
8.59–9.05
~~that should have read your ,but who's to say~~ you're not reading a value of the tuple from the future

你不会读取到该tuple未来的值
09:05 – 09:13
right so there’s a transaction in the future ,that’s overwritten the database value that you should not be reading ~~that’s Essentially~~
So，在未来，某个事务会去覆写掉数据库中的这个值，你不应该读取到这个覆写后的值
09:13 – 09:14
but that’s a problem
但这是一个问题
9.14-9.15
and when this happens
当这种情况发生的时候
9.15-9.16
 you essentially have to abort
你就必须中止该事务
09:16 – 09:17
and when you abort
当你中止该事务的时候
9.17-9.18
 you have to make sure that
你需要确保
9.18-9.24
you start with a newer transaction a newer timestamp, than the one that you had when you initially began the transaction
你所开始执行的新事务所携带的新时间戳要比你一开始执行该事务时的时间戳要新
09:24 – 09:28
does everybody sort of understand why you need to assign a newer timestamp
你们是否明白我们为什么需要分配一个较新的时间戳?
09:29 – 09:32
what would happen if you had the same timestamp you had before
如果你使用的是和之前相同的时间戳，这会发生什么呢?
09:34 – 0934
exactly
没错
9.34-9.35
 you’d run into the same problem right,
你会遇上同样的问题
9.35-9.39
~~you have to that’s because mod~~ because you have to ensure timestamps are monotonically increasing
因为你必须确保时间戳是单调增加的
9.39-9.42
you have to get a new timestamp to avoid this problem, yeah
你必须使用一个新时间戳来避免这个问题
09:52 – 09:55
so if your timestamp is the same as a write timestamp
So，如果你的时间戳和write timestamp相同
9.55-9.56
 what does that mean
这意味着什么呢?

09:58 – 09:58
exactly
没错
10:00 – 10:04
you can read it,you can read it right, it should be repeatable reads, it’s perfectly fine, yeah
你可以进行读取，这种情况是可重复读，这没什么问题
10:14 – 10:15
so in this scheme
So，在这种方案下
10.15–10.19
our timestamps are assigned when you begin that’s when begin the transaction
我们会在事务开始的时候给它分配时间戳
10:21 – 10:21
okay

10.21–10.23
so if this invariant is invalidated
So，如果这个不变量无效了
10.23–10.25
you essentially have to abort
也就是说，你需要中止该事务
10:25 – 10:26
but on the other hand
但在另一方面
10.26–10.29
if you’re actually allowed to perform the read
实际上，如果你被允许去执行这个读操作
10.29–10.40
then you modify the read timestamp for this tuple to take the maximum of whatever the read timestamp is right now and what your own timestamp is
那么，你就会对该tuple的read timestamp进行修改，你会使用现在的read timestamp和你自己的时间戳这两者间的最大值作为该tuple新的read timestamp

10:40 – 10:42
does anybody know why you have to take the maximum
有人知道我们为什么采用的是两者间的最大值么
10:47 – 10:49
timestamp have to be monotonically increasing right
时间戳必须是单调增加的
10.49–10.53
you could have a transaction that’s newer than you, update the read timestamp
你可以使用一个较新的事务来更新read timestamp
10:53 – 10:56
but you can’t set back the timestamp back to what you were
但你不能将时间戳设定为以前的时间
10.56–10.57
because you’re older, right

因为你现在的时间戳是老时间戳
10:57 – 10:59
timestamp have to be monotonically increasing
时间戳需要是单向增加的
11:00 – 11:05
so you have to take the maximum or whatever it is right at this point in time when you read it and what’s your own timestamp is
你就必须在read timestamp和你自己的时间戳中选出那个最新的时间戳来进行更新
11:06 – 11:08
so this is important okay
So，这很重要
11:09 – 11:11
so once you’ve updated the timestamp
So，一旦你更新了时间戳
11.11–11.17
you now have to make a copy of this tuple into some local private workspace that’s only visible to you
你必须将该tuple的副本保存到一个只有你可见的本地私有工作空间中去
11:17 – 11:19
so that you can ensure that you get repeatable reads
这样的话，你就可以确保你可以做到可重复读
11:51 – 11:55
so assume that there’s some tuple that you want to read right
So，假设你想读取某个tuple
11:55 – 12:59
and the write timestamp is from a transaction that’s newer than you that’s in the future
该事务的write timestamp要比你新，它的时间点是在未来
12:00 – 12:02
right you shouldn’t be able to read that value right
你就不能读取这个值
12.02–12.06
you should be reading the value that existed before it wrote before the new transaction wrote it
你所读取到的值应该是这个新事务执行写操作前就已经存在的值
12:06 – 12:08
so that’s why it validates
So，这就是为什么它有效的原因
12:13 – 12:17
because you now appear in a different order in the serial order right
~~因为你在按顺序执行时，使用了一个不同的执行顺序~~
因为你在Serial Order中出现了一个不同的执行顺序
12:17 – 12:21
~~you shouldn’t~~ because your timestamp is newer than the one that wrote to it before .
因为你的时间戳要比之前它执行写操作时来得新
12.21–12.24
you logically appear after this transaction in the serial order
从逻辑上来讲，在Serial Order中，你是在这个事务之后出现的
12:29 – 12:31
yeah go ahead

你说
12:34 – 12:36
so this is not I don’t think it’s covered in the book
So，我不觉得书中有对它进行介绍
12.36–12.38
,but you have to make sure that you have to make a local copy
但你必须确保你制作了一份本地副本
12.38–12.40
,so that you can issue repeatable reads
这样的话，你就可以发起可重复读了
12:44 – 12:45
so you can imagine that
So，你可以想象一下
12.45–12.48
another transaction comes in and updates the system
当另一个事务进来并对系统中的数据进行更新操作时
12:49 – 12:51
but you have to be able to read the same value that you read initially
你必须能够读取到你最初读取的值
12:53 – 12.57
but if you allow another transaction to update the value here
但如果你允许另一个事务去更新这里的值
12.57–13.00
then you would invalidate this this this invariant at top
那么，你就要无效化顶部这个不变值
13:00 – 13:01
and you wouldn’t able to read it
你不能读取这个值
13.01–*13.02
but you actually should be able to read it right
但实际上你应该能够读取这个值
13:03 – 13:06
because~~ you wrote it ~~you read it in a transaction inconsistent state
因为你在这个事务中读取到的是一个不一致状态
13:08 – 13:08
yeah
请讲
13:20 – 13:22
Yeah that’s a that’s a very good point
你说的这点非常不错
13.22–13.28
 I think you’re alluding to the fact that there could be starvation where you have consistently consistent in aborts
~~我认为你所暗示的是这里会存在starvation的情况~~
在这里出现一致性中止的时候，我认为这里暗示了存在着starvation的情况

13:28 – 13:29
that’s a drawback of this approach

这是该方案的一个缺点

13.29–13.33

and we’ll get to that later in the presentation

我们稍后会提到它

13:33 – 13:35

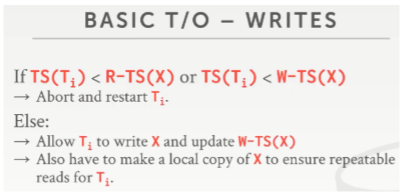
okay so this is just for reads

Ok，这里讲的都是关于读操作的

13.35–13.37

you have a similar similar story for writes

对于写操作也是类似情况



BASIC T/O – WRITES

If $TS(T_i) < R\text{-}TS(X)$ or $TS(T_i) < W\text{-}TS(X)$

→ Abort and restart T_i.

Else:

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

→ Also have to make a local copy of X to ensure repeatable reads for T_i.

13:38 – 13:39

Okay

13.39–13.48

so if your transaction I if your timestamp is less than the the read timestamp of the the object that you’re trying to write into

So，如果你事务的时间戳小于你试着写入的那个对象所携带的read timestamp

13:49 – 13:50

that means it’s a newer transaction

这意味着，它是一个较新的事务

13.50–13.53

that read a stale value

它会读取到一个过时的值

13.53–13.56

a value that you that should have been coming from you as a transaction

这个值你应该在之前的事务中就已经看到过了

13.56–13.57

but as is not anymore

但它现在就不应该存在了

13:57 – 14:00

So that’s a violation of the of this timestamp ordering protocol

So，这就违反了这个timestamp ordering协议

14:01 – 14:02

Similarly

类似的

14.02–14.10

if your timestamp is less than the write timestamp of another object

如果你的时间戳小于另一个对象的write timestamp

14.10–14.14

then again there’s a new a transaction ,that essentially overrode your value

接着，一个新事务会覆盖掉你的值

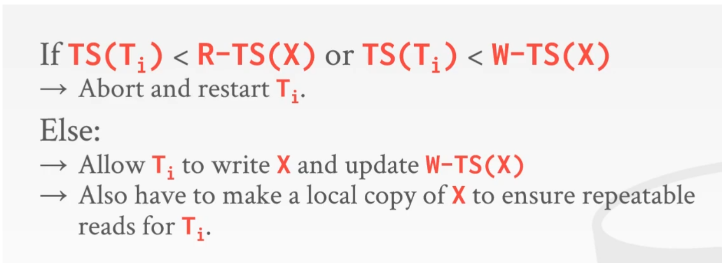
14:14 – 14:15

and again that’s a violation

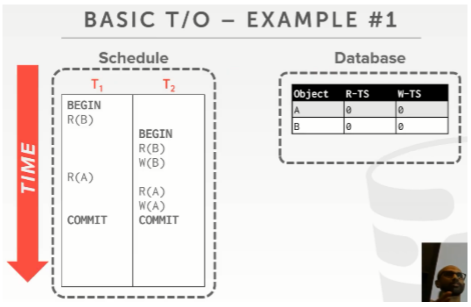
这种情况违反了这个协议

14.15–14.16

 and the idea is that
这里的思路是


If $TS(T_i) < R\text{-}TS(X)$ or $TS(T_i) < W\text{-}TS(X)$
→ Abort and restart T_i.
Else:
→ Allow T_i to write X and update W-TS(X)
→ Also have to make a local copy of X to ensure repeatable reads for T_i.

14.16–14.19
 if either of these conditions are true
如果其中一个条件为true
14:19 – 14:23
you have to abort and again start with a newer timestamp value and begin the entire process again
那你就需要中止并重启该事务，你需要将新的时间戳分配给它，然后再次执行整个过程
14:25 – 14:26
if on the other hand
如果为false
14.26–14.29
these this is a it’s a valid write
那么，这就是一次有效的写操作
14.29–14.32
then you have to update the write timestamp for the tuple
那么，你就需要去更新该tuple的write timestamp
14:32 – 14:39
and you essentially done you have to also make a local copy here in order to support repeatable read
本质上来讲，这里你也必须要制作一个本地副本，以此来支持可重复读
14.39–14.44
, you read your local copy instead of going back to the in in to the databases global state
你会去读取你本地副本中的值，而不是回到数据库中去读取数据库中的值
14:46 – 14:46
okay okay


BASIC T/O – EXAMPLE #1

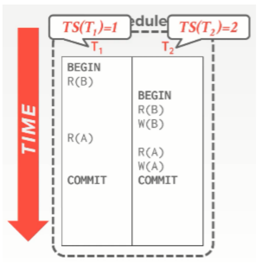
Schedule

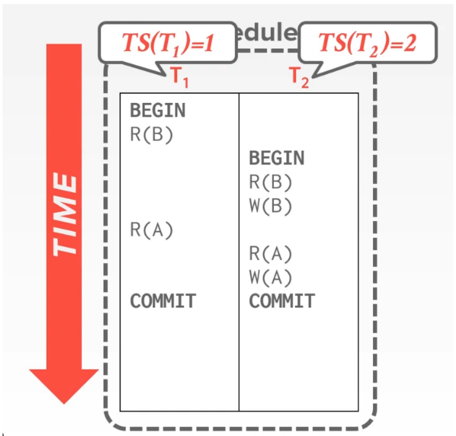
T_1	T_2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
	W(A)
COMMIT	COMMIT

Database

Object	R-TS	W-TS
A	0	0
B	0	0

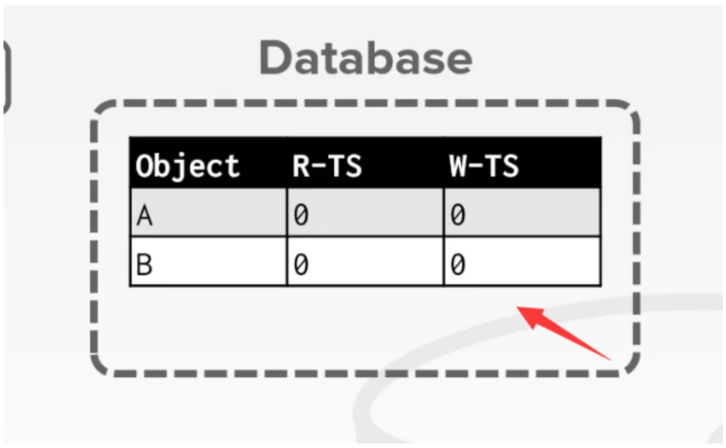
14:54 – 14:57
so let’s walk through an example hopefully this will clear things up a little bit
So，我们来看个例子，希望这能让你们明白这些东西
14:58 – 15:00
so we have two transactions here
So，这里我们有两个事务
15.0–15.04
,and it’s just assumed that you can only execute one transaction one operation at a time

假设你一次只能执行这些事务中的一个操作
15:04 – 15:05
So you can assume single core single thread
So，假设我们这里的场景是单核和单线程的情况
15:06 – 15:16
and in this database we now have annotated all of the the tuples we have, all of the objects here with the read timestamp and a write timestamp
我们现在已经列出了该数据库中我们所拥有的所有tuple，以及它们所对应的read timestamp和write timestamp
15:16 – 15:17
okay so let’s get started
Ok，我们开始吧


15:17 – 15:19
So we have t1 and t2
So，我们有T1和T2这两个事务
15.19–15.22
when they entered the system
当它们进入系统时
15:22 – 15:23
they were assigned a timestamp
我们就会给它们分配一个时间戳


15.23–15.26
assume that T1 is assigned a timestamp of 1
假设我们分配给T1的时间戳是1
15.26–15.28
,and T2 is assigned a timestamp of 2
分配给T2的时间戳是2
15.28–15.30
,right pretty simple
很简单吧

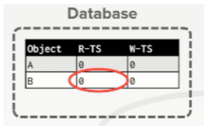
15:30 – 15:31
then you do the read
接着，我们执行读操作
15.31–15.34
so T1 does a read of B
So，T1对B进行读取



Database

Object	R-TS	W-TS
A	0	0
B	0	0

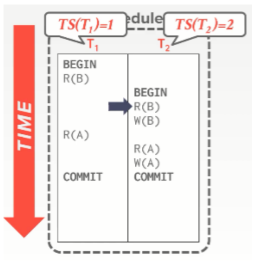
15.34–15.36
you look at the write timestamp for B ,it’s 0
可以看到B的write timestamp的值是0
15.36–15.37
1 is greater than 0
1比0大



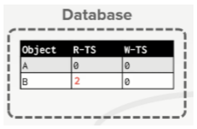
Database

Object	R-TS	W-TS
A	0	0
B	0	0

15:38 – 15:40
so you update and the read timestamp to 1
So，你将B的read timestamp值更新为1



15:42 – 15:45
then we do a context switched into transaction 2
接着，我们将上下文切换到T2
15.45–15.48
and T2 now wants to do a read of B
T2现在想对B进行读取

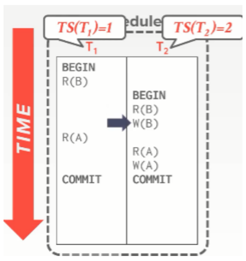


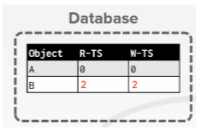
Database

Object	R-TS	W-TS
A	0	0
B	2	0

15:48 – 15:49
if you look at the write timestamp
如果你去看下B的write timestamp
15.49–15.51
and you’re good to go

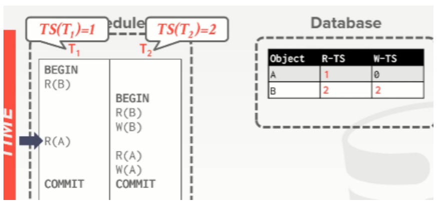
看了下，没问题，我们可以继续执行我们的操作
15.51–15.57
 so you update the read timestamp to take the maximum of what it was which is 1 and the new timestamp 2 and you get 2
So，我们就可以对它的read timestamp进行更新，我们取它当前read timestamp和新时间戳这两者间的最大值作为它的新read timestamp，即2





Object	R-TS	W-TS
A	0	0
B	2	2

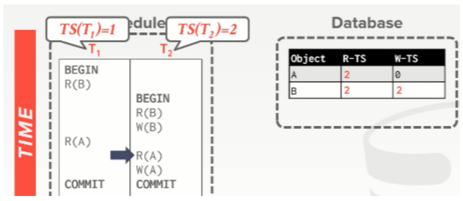
15:58 – 15.58
all right

15.58–16.01
now you do a write
现在，我们来执行写操作
16.01–16.03
,so T2 wants to do a write of B
So，T2想对B进行读取
16:04 – 16:06
you look at the write timestamp and the real timestamp your greater than both
通过查看B的write timestamp，我们发现我们真实的时间戳值要大于它
16.06–16.09
,so you update the write timestamp of B to 2
So，我们将B的write timestamp更新为2


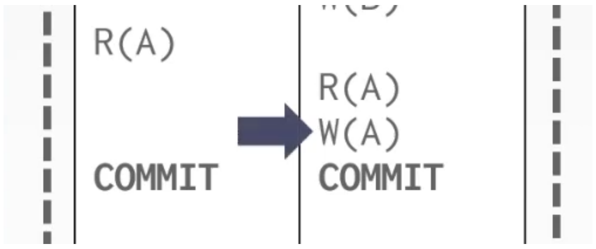
Object	R-TS	W-TS
A	1	0
B	2	2

16:11 – 16:15
Okay, now you do a context switch back into t1
现在，我们切换到T1
16.15–16.17
 t1 wants to do a read of a
T1想对A进行读取
16.17–16.20
,it looks at the the write timestamp of a
它会去查看A的write timestamp
16.20–16.21
1 is greater than 0
它发现1比0大
16:22 – 16:30
so you update the read timestamp to be the maximum of 0 1 and you're good to go
sorry yeah you update the read timestamp to 1

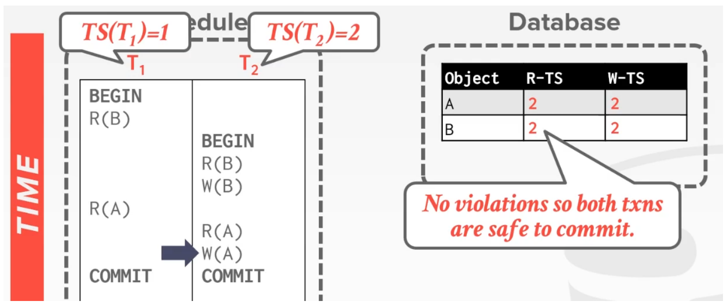
So，这里的read timestamp应该是取0和1这两者间的最大值，我们将A的read timestamp更新为1



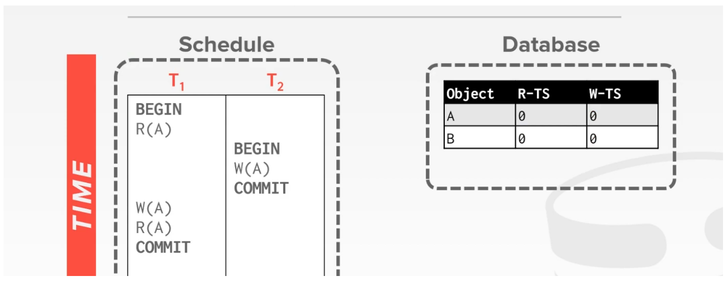
Object	R-TS	W-TS
A	2	0
B	2	2

16:31 – 16:33
now you come back over to transaction 2
现在，我们切换到T2
16.33–16.36
and T2 wants to do a read of a
T2想对A进行读取
16:36 – 16:37
you look at the write timestamp of a,
我们看了下A的write timestamp
16.37–16.39
2 is greater than 0
2比0大
16.39–16.42
you update read timestamp to 2 now
我们将A的read timestamp更新为2


16:42 – 16:47
And then finally, t2 wants to do a write of a
接着，最后，T2想对A进行写入操作
16:47 – 16:51
so it looks at both the read timestamp on the write timestamp of a, it’s greater than both of them
So，它会去查看A的read timestamp和write timestamp，我们的时间戳要比这个两个时间戳都来得大
16:51 – 16:53
so that that’s for the write is valid
这样的话，这个写操作就是有效的
16:53 – 16:58
~~and yeah so no validations exist~~ no no no violations exists
So，这里不存在违反协议的情况



16:59 – 17:01
so both transactions are safe and you can commit both of them
So，这两个事务都是安全的，你可以提交它们

17:03 – 17:04
Alright this is this clear
你们懂了吗?
17:05 – 17:06
okay good
Ok，不错
17.06–17.10
 so let’s walk through another example
So，我们来看下另一个例子


17:11 – 17:13
so it’s the same setup here
So，它和之前的设置完全相同
17.13–17.14
all the initial timestamps are 0
所有的时间戳都是0
17.14–17.17
 T1 and T2 enter the system
T1和T2进入了系统
17:17 –17:20
T1 gets a timestamp 1
T1分配到的时间戳是1
17–.20–17.21
, T2 gets timestamp 2
T2分配到的时间戳是2
17:21 – 17:22
All right

17.22–17.23

so in this scenario

So，在这种情况下

17.23–17.25

T1 wants to do a read

T1想对A执行一次读操作

17.25–17.27

it’s a good read

这一步读取操作并没有什么问题

17:27 – 17:30

so it updates the the read timestamp

So，T1会去更新read timestamp

17.30–17.31

do a context switch

接着，我们切换到T2

17:32 – 17:34

and now T2 wants to do a write of a

现在，T2想对A进行写入操作

17.34–17.36

it checks the read timestamp and the write timestamp,

它会检查A的read timestamp和write timestamp

17.36–17.37

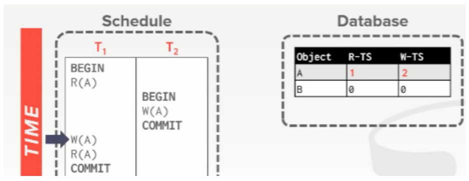
 it’s valid

它们是有效的

17.37–17.39

 so it updates the write timestamp to 2

So，它会将A的write timestamp更新为2



17:41 – 17:44

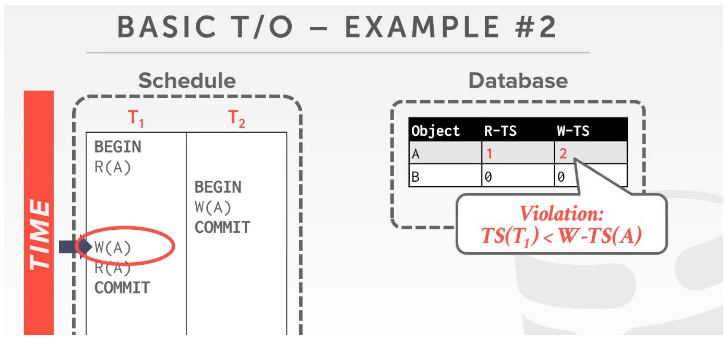
T1 now is trying to do a write of a

T1现在试着要对A进行写入操作

17:44 – 17:47

so checks the write timestamp and read timestamp

So，它会去检查A的write timestamp和read timestamp

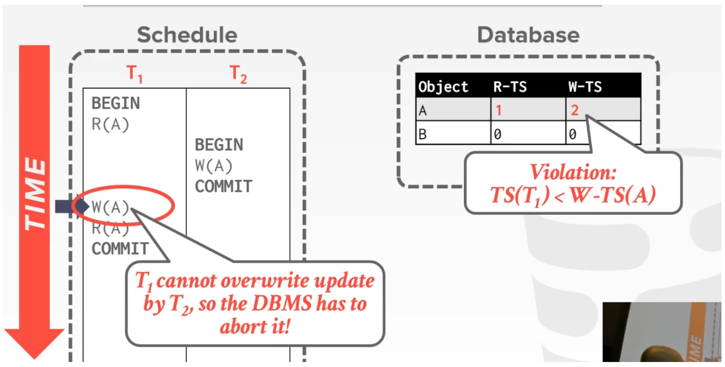


17.47–17.48

it’s no longer valid

它不再有效

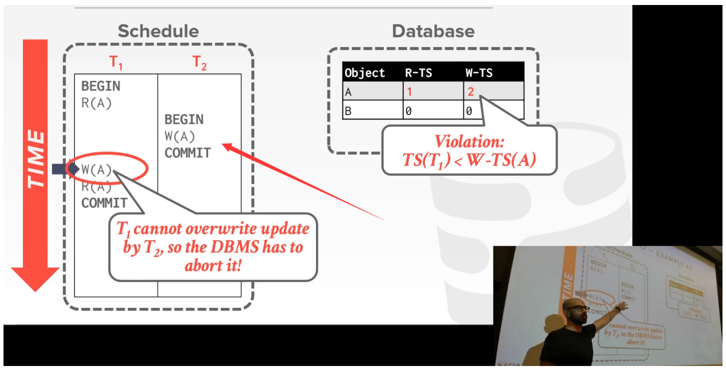
17.48–17.50
,because you have it the timestamp of 1
因为A的read timestamp是1
17.50–17.54
 ,which is 1 is less than the write timestamp of a, which is 2
1小于A的write timestamp 2
17.54–17.57
this is a violation
这里就违反了我们说的东西
17:57 – 18:00
so T1 actually can’t commit it has to abort
So，实际上，T1不能进行提交，我们需要中止它
18:03 – 18:04
It`s that clear
你们懂了吗
18.04–18.07
you can think about this as like in the serial order
你们可以想象一下，在Serial Order中
18.07–18.09
because T1 has timestamp 1 and T2 has timestamp 2
因为T1的时间戳是1，T2的时间戳是2
18:09 – 18:12
T1 should appear before T2 in the serial order
在Serial Order中，T1应该在T2之前出现，即T1的时间戳比较小


Schedule
Database
TIME
T_1
T_2
BEGIN
R(A)
W(A)
R(A)
COMMIT
BEGIN
W(A)
COMMIT
Violation: $TS(T_1) < W\text{-}TS(A)$
T_1 cannot overwrite update by T_2, so the DBMS has to abort it!

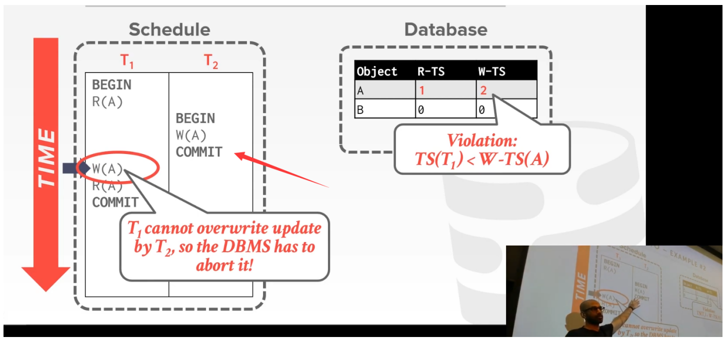
Object	R-TS	W-TS
A	1	2
B	0	0

18:15 – 18:18
so this is obviously going to be a violation
So，显然，这违反了我们的规则
18.18–18.20
because this read here it’s gonna be aborted
因为这里的读操作会被中止
18:22 – 18:24
okay ~~sorry~~

18:26 – 18:31
so there’s actually an optimization that we can make here to avoid aborting in this specific scenario
So，实际上，这里我们可以进行优化，以防止在这种特定情况下出现事务中止的情况
18:32 – 18:35
right you can think about it

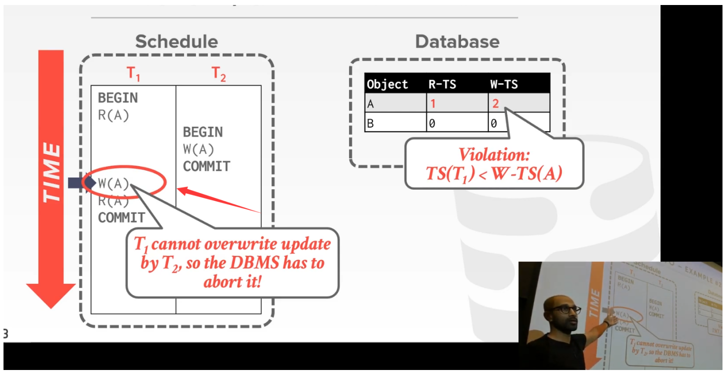
你们可以这样想
18.35–18.36
as in physical time what’s happening
在物理时间上，这里发生了什么呢
18:36 – 18:38
right T2 is writing to something
T2正对某个东西进行写入操作
18.38–18.41
, but then it’s being overwritten by T1
但接着，它所做的修改又被T1覆盖掉了

Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
R(A)
COMMIT
BEGIN
W(A)
COMMIT
T_1 cannot overwrite update by T_2, so the DBMS has to abort it!
Database
Violation: $TS(T_1) < W\text{-}TS(A)$

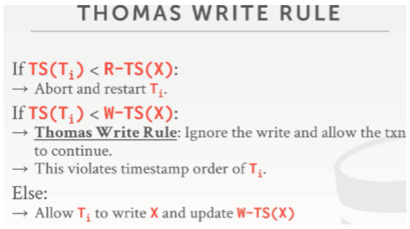
Object	R-TS	W-TS
A	1	2
B	0	0

18:41 – 18:43
so do we really need this write
So，我们是否真的需要这个写操作呢?
18:44 – 18:47
no I think the observation is you actually don’t need it
我觉得从我们的观察结果来看，我们实际并不需要它
18:48 – 18:49
what you could have is
我们所能做的事情是
18.49–18.54
because we’ve maintaining a local every transaction is maintaining a local copy of the soup of the tuples
因为每个事务会为它所操作的tuple制作一份本地副本
18:54 – 18:57
this write here can can essentially be ignored by the system
本质上来讲，这个写操作就可以被系统所忽略了

Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
R(A)
COMMIT
BEGIN
W(A)
COMMIT
T_1 cannot overwrite update by T_2, so the DBMS has to abort it!
Database
Violation: $TS(T_1) < W\text{-}TS(A)$

Object	R-TS	W-TS
A	1	2
B	0	0

18:58 – 19:02
right because externally ,this write is what’s valid
因为从外部的角度来讲，这个写操作是有效的
19:03 – 19:05

and as long as this write is externally valid
从外部来看，只要这个写操作是有效的

Schedule
Database
TIME
T_1
T_2
BEGIN
R(A)
W(A)
R(A)
COMMIT
BEGIN
W(A)
COMMIT
Object R-TS W-TS
A 1 2
B 0 0
Violation: $TS(T_1) < W\text{-}TS(A)$
T_1 cannot overwrite update by T_2, so the DBMS has to abort it!

19.05–19.07
then you don’t actually need this one
那么，你实际就不需要这个写操作了
19:07 – 19:08
externally
从外部角度来讲是这样的
19.08–19.08
 within the transaction
在事务内部
19.08–19.09
 you still need this write
你依然需要这个写操作
19.09–19.11
because you have to be able to read your own writes
因为你需要能够读取到你自己写操作所做的修改


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 and allow the txn to continue.
→ This violates timestamp order of T_i.
Else:
→ Allow T_i to write X and update $W\text{-}TS(X)$

19:12 – 19:20
so this observation leads to an optimization that you you can apply in these timestamp based systems call the Thomas Wright rule
So，我们从中观察到的一种能用于这种基于时间戳的系统所做的优化叫做托马斯写入规则（Thomas Write Rule）
19:21 – 19:22
and the idea is that
它的思路是
19.22–19.24
if you’re trying to write into an object X
如果你试着对对象X进行写入操作
19:26 – 19:27
as before
和之前一样
19.27–19.30
 if your timestamp is less than the the read timestamp for that object
如果你的时间戳小于该对象的read timestamp

19.30–19.33
you still have to abort and start with a new a timestamp
你依然需要中止该事务，并开启一个携带新时间戳的事务
19:34 – 19:38
but if the timestamp is less than the write timestamp of the object
但如果该时间戳小于该对象的write timestamp
19.38–19.38
which means
这意味着
19.38–19.41
there’s a newer transaction that wrote into this object
有一个较新的事务已经修改过该对象了
19:41 – 19:44
you can actually just ignore the write altogether
实际上，你可以忽略掉这个写操作
19:44 – 19:49
you have a local copy of the write that you can now read
你现在可以读取这个对象的本地存储副本
19.47–19.52
but externally it’s okay to ignore this write
但从外界来看，将这个写操作忽略掉是Ok的
19:52 – 19:55
yeah yeah yes
请讲