

18-04

18-04

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.

01:00:00 – 01:00:06

so I have to look at the local working copies for all the other transactions as part of my validation

So, 在我验证的时候, 我需要查看其它所有事务的本地数据副本

01:00:07 – 01:00:09

and I do that and in order to do that in a consistent way

为了以一致的方式做到这点

1.00.09–1.00.10

I have to acquire latches

我需要去获取latch

1.00.10–1.00.13

,and this latch is these latches can have some overhead

使用这些latch会给我们带来一些开销

01:00:14 – 01:00:14

yeah

请讲

01:00:27 – 01:00:32

yeah there's this ,yeah there's there's a lot of techniques that you can use to reduce the latch contention

我们可以使用很多技巧来减少这种latch的争抢

01:00:32 – 01:00:35

But at the end of the day like if you even if you have a parallel sighting

但最终如果你以并行的角度来看待这个的话

1.00.35–1.00.36

,there is some overhead ,right

这其实还是有一定开销的

01:00:40 – 01:00:44

so what if we take a slightly different view right

So, 如果我们以一种略微不同的角度来看呢?

01:00:44 – 01:00:49

what if we partition the entire database

如果我们对整个数据库进行分区会怎么样呢?

01:00:50 – 01:00:51

so that

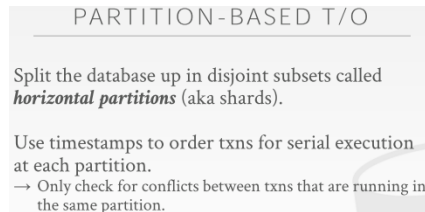
这样的话

1.00.51–1.00.51

all of the stuff that one transaction has to work on is only within one transaction
一个事务需要处理的所有东西都是只局限在这个事务中的
一个事务所有的工作都只在一个分区上

01:00:54 – 01:01:00

and then I can remove all of my locks and latches altogether ,is that even possible
那么我就可以移除掉我所有的lock和latch, 这种做法可行吗?



01:01:00 – 01:01:02

so it turns out that this is a valid technique

So, 事实证明, 这是一项有效的技术

1.01.02–1.01.06

and it's called **partition-based T/O**

它叫做Partition-Based Timestamp Ordering

01:01:06 – 01:01:07

the idea is

它的思路是

1.01.07–1.01.13

that I want to split up my database into these **horizontal partitions**, right

我想将我的数据库拆分为这些水平分区

01:01:13 – 01:01:19

and then I want to be able to use timestamps to order the transactions on a partition in serial order

接着, 我想能够通过时间戳来对该分区所涉及的事务进行排序, 让它们以Serial Order的顺序执行

01:01:20 – 01:01:23

right so if I have transactions executed executing serially

So, 如果我的事务按顺序执行

1.01.23–1.01.28

~~then there's no reason~~, I'm sorry within a database there's no reason to have locks and latches at all

那么在一个数据库中, 我们根本也就没理由去使用lock和latch了

01:01:28 – 01:01:29

because there's no concurrent activity

因为这里面不存在并发

01:01:31 – 01:01:33

right if they're operating single-threaded,

如果它们是单线程执行的话

1.01.33–1.01.36

~~I don't need lights~~, I don't need locks, I don't need latches

我也就不需要用到lock和latch了

01:01:39 – 01:01:44

and then it gets a little bit complicated ,if you have to access multiple of these database partitions

如果你需要访问多个数据库分区，那么情况就会变得有些复杂

01:01:44 – 01:01:47

but if you're only accessing all the data within one partition

但如果你访问的只是一个分区中的所有数据

1.01.47–1.01.50

then it can be potentially really really fast

那么，它的速度就会变得很快很快

01:01:50 – 01:01:53

and this is sort of what partition based time step ordering is trying to achieve

这就是Partition-Based Timestamp Ordering所试着做到的事情

01:01:54 – 01:01:56

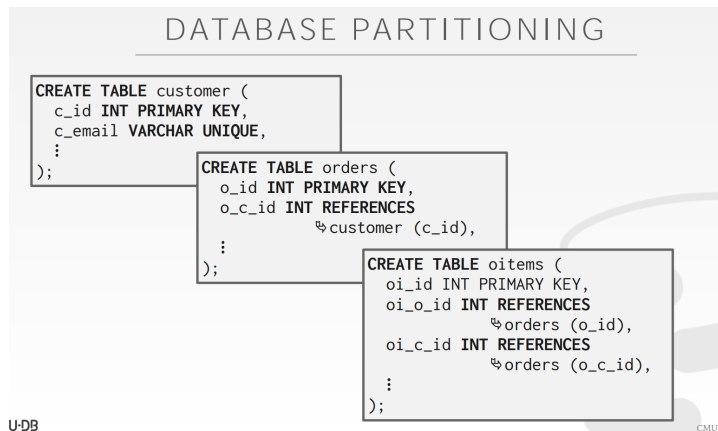
let's let's walk through an example

我们来看个例子

1.01.56–1.01.56

,so imagine that

So，想象一下



1.01.56–1.02.00

I have this the schema

我拥有这样一个schema

1.0200–1.02.02

that represents some online store that I'm running

它表示的是我所运营的那个在线商城

01:02:02 – 01:02:06

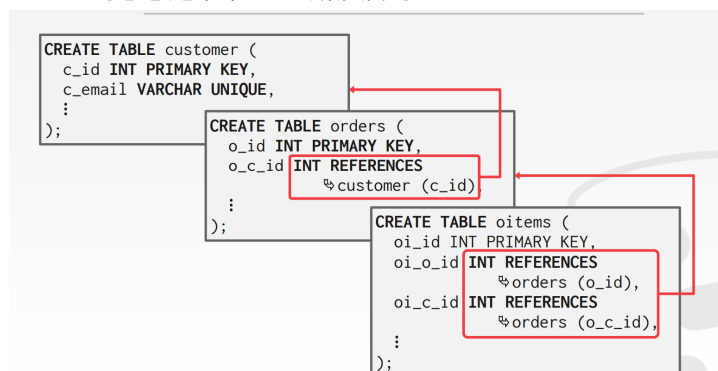
I have **customers**, **orders**, and the the **oitems**,

我有三张表，即customer表，orders表以及oitems表

1.02.06–1.02.10

the items in the order for an order

oitems的意思是某个order所涉及的item



01:02:10 – 01:02:12

what I could potentially do is

我可以做的事情是

1.02.12–1.02.14

because I have these foreign key references

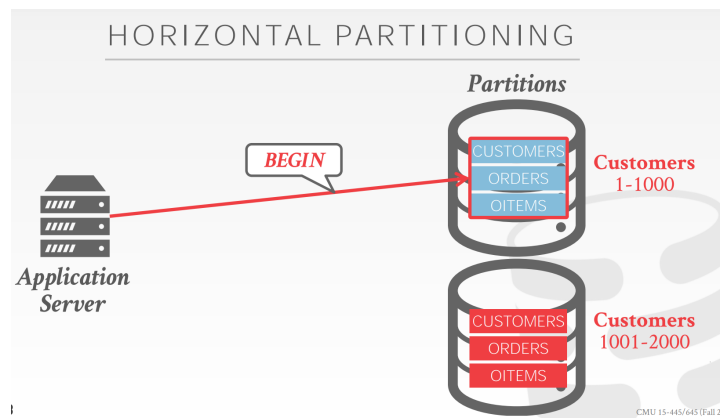
因为这里面我有一些外键引用

01:02:15 – 01:02:25

I can store within one partition of the table a set of customers all of their orders and all

of the items for their orders, by using this foreign key reference structure

通过使用这种外键引用结构，我可以将一些客户信息、他们所对应的订单信息以及订单中所涉及的商品信息都保存在一个分区中



01:02:27 – 01:02:27

okay

1.02.27–1.02.28

so imagine that

So, 想象一下

1.02.28–1.02.29

I have two databases now

现在我有两个数据库

1.02.29–1.02.36

where I have customers 1 – 1000 and 1001– 2000 in a separate database

我将c_id从1到1000的客户信息放在一个数据库中，c_id从1001到2000的客户信息放在另一个数据库中

01:02:36 – 01:02:39

~~And I want to~~ let's say, I want to update a customer's name

假设，我想去更新某个客户的名字

1.02.39–1.02.43

I want to add in order to one customer's order

我想对某个客户的订单信息进行修改

01:02:43 – 01:02:45

right what do I do

我该怎么做呢？

01:02:46 – 01:02:49

so ~~a transaction~~ I have an application here

So，这里我有一个应用程序

1.02.49–1.02.50

it begins a transaction

它开启了一个事务

1.02.50–1.02.50

and let's say that

假设

1.02.50–1.02.55

it's trying to update customer one ,it falls into this partition here

它试着去更新c_id为1的客户信息，它会落到这个分区中

01:02:56 – 01:02:59

so okay it's missing something

Ok, 这里少写了点东西

01:02:59 – 01:03:06

so assume that in this line ,there's a there's an operation that says get me the name of the customer whose ID is one

So, 假设这条箭头上写了这样一个操作，该操作想去获取c_id为1的那个客户的名字

01:03:06 – 01:03:09

okay that's an operation that obviously falls into this partition

Ok, 显然这个操作是在这个分区中执行的

1.03.09–1.03.10

because this customer belongs in this partition

因为这个客户属于这个分区

01:03:12 – 01:03:14

Right and then it does commit

接着，它提交事务

1.03.14–1.03.17

and it can safely do that right

它能够安全地做到这些

01:03:17 – 01:03:22

because all of these transactions get queued up and operated in a single thread in this database

因为所有的事务都会放在一个队列中，在数据库中，我们会以单线程的方式来执行这些事务

01:03:22 – 01:03:24

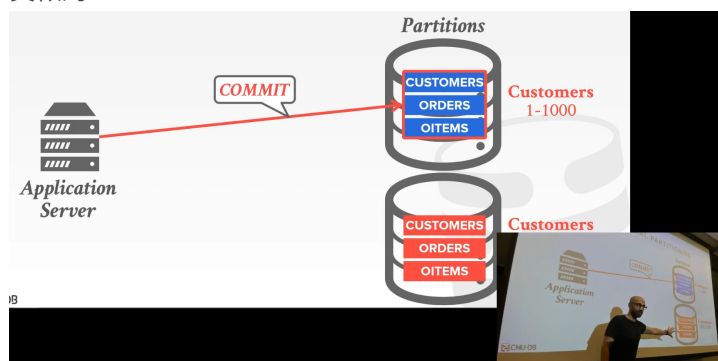
there's no concurrent activity at all

这里面并不存在任何并发行为

01:03:25 – 01:03:25

Similarly

类似的



1.03.25–1.03.33

I can have another application server that wants to update the customer 1004 that'll go here

接着，我有另一个应用程序服务器，它想去更新c_id为1004的customer，它应该跑到下面这个分区这里

01:03:35 – 01:03:38

and these two transactions are totally disjoint

这两个事务是完全不相交的

1.03.38–1.03.39

they're operating on different customers

它们操作的是不同的customer

1.03.39–1.03.44

,that's one way that we get parallelism even though each database is operating single threaded

这就是我们获得并行能力的方式，即便是每个数据库都只能操作一条线程，我们也可以做到

01:03:44 – 01:03:50

okay the finer grain that we can have these customers ,we can store these these database partitions, the more parallelism we can get

如果我们将数据库分区的粒度越细，那么我们所获得的并行性就越高

01:03:51 – 01:03:54

and that way each individual transaction runs much faster

这样的话，每个事务执行的速度就会变得更快

1.03.54–1.03.56

because we don't have to acquire locks and latches

因为我们不需要再去获取lock和latch了

01:03:57 – 01:04:02

and we get parallelism by having more of these transactions when it concurrently on disjoint sets of the data

当更多事务以并发的方式去访问那些不相交的数据集时，我们的并行性就越高

01:04:04 – 01:04:05

is that clear

懂了吗

01:04:06 – 01:04:06

all right

PARTITION-BASED T/O

Txns are assigned timestamps based on when they arrive at the DBMS.

Partitions are protected by a single lock:

- Each txn is queued at the partitions it needs.
- The txn acquires a partition's lock if it has the lowest timestamp in that partition's queue.
- The txn starts when it has all of the locks for all the partitions that it will read/write.

kx VOLTDB FAUNA H-Store

01:04:09 – 01:04:11

so this turns out to be a very popular protocol

So，事实证明，这是一个很流行的协议

01:04:12 – 01:04:19

you still have to assign transactions and ID and then you queue them up by these IDs

你依然需要给这些事务分配id，接着，你需要根据这些id对这些事务进行排序

01:04:20 – 01:04:26

And the this type of partition based protocol has been actually very very very successful
实际上，这种基于分区的协议非常成功

01:04:27 – 01:04:29

so Andy worked on this system called **H-store** which is a commercial system

So, Andy以前开发的H-Store, 它是一个商用数据库系统

01:04:31 – 01:04:33

that now get that got commercialized into **VOLTD**

它的商业化产物就是VoltDB

1.04.33–1.04.37

and they're still using this type of partition based T/O

他们使用的依然是这种Partition-Based Timestamp Ordering

01:04:37 – 01:04:41

Andy worked with the someone called a professor called **Daniel Abadi**

Andy当时和一个叫做Daniel Abadi的教授一起工作

1.04.41–1.04.41

now,who's at I think Maryland

他是马里兰大学的教授

1.04.41–1.04.44

now who did some initial work on a system called Calvin

他为一个叫做Calvin的系统做了些初步工作

1.04.44–1.04.49

,that's now been commercialized into fauna that also uses something very similar to this

它的商业化产物叫做Fauna, 该系统也使用了和它相类似的东西

01:04:49 – 01:04:51

And there's a system called KxDB

这里还有一个叫做KxDB的东西

1.04.51–1.04.52

which is of like a financial database

它是金融领域用的数据库

01:04:53 – 01:04:54

and they also use this technique

它们也使用了这种技术

1.04.54–1.04.59

and it's actually pretty successful if your work votes of supports this type of partition based operation

实际上, 如果你的系统也支持这种Partition-Based Timestamp Ordering的话, 你的系统就会非常成功

01:05:02 – 01:05:04

the way that you can view this is that

你看待这项技术的方式是

1.05.04–1.05.07

every single database essentially has a giant lock or a latch around it

本质上来讲, 每个数据库上都被加了一个巨大的lock或者latch

01:05:07 – 01:05:09

and when a transaction is ready to commit

当一个事务准备好提交的时候

1.05.09–1.05.10

,it acquires this latch

它会获取这个latch

1.05.10–1.05.17

,and it begins execution within the database fully single threaded bare metal speed
它会以单线程的方式来执行对该数据库的操作，执行速度和裸机速度一样

01:05:17 – 01:05:19

And then as other transactions come up

当其他事务出现的时候

1.05.19–1.05.21

they queue up on this latch

它们会排队获取这个latch

1.05.21–1.05.22

, and they essentially get assigned a timestamp

本质上来讲，我们会给它们分配一个时间戳

01:05:23 – 01:05:25

And then as their timestamp gets gets ringed up

当轮到它们执行的时候

1.05.25–1.05.29

,they begin execution into this database into this into this database partition

它们就会开始对该数据库分区进行操作

PARTITION-BASED T/O – READS

Txns can read anything that they want at the partitions that they have locked.

If a txn tries to access a partition that it does not have the lock, it is aborted + restarted.

01:05:34 – 01:05:36

okay yeah so that's it

Ok，这块就讲完了

1.05.36–1.05.38

so for reads

So，我们再来看下读操作方面的事情

01:05:41 – 01:05:47

The database that the transactions can essentially read whatever they want right now safely without requiring latches

本质上来讲，在不获取latch的情况下，事务可以读取任何它们想读取的东西

01:05:47 – 01:05:51

because they came **Caron** the database system guarantees, that there's no other transaction running in the system

因为数据库系统保证此时并没有其他事务在系统中运行

01:05:53 – 01:05:57

this is great if you only want to read stuff within one database partition

如果你只想在一个数据库分区中读取数据，那这就很棒

01:05:57 – 01:06:04

it gets complicated when you want to try to read rows that exist across partitions

当你想试着去读取存在于多个数据库分区中的几行数据时，情况就会变得复杂

01:06:04 – 01:06:11

so in the application server we had before, let's say that you want to modify within one transaction customers from two different partitions ,that becomes complicated

So，假设我们之前的应用程序服务器想通过一个事务来对两个不同分区内的customer信息进行修改，那么情况就会变得复杂

01:06:11 – 01:06:17

because now I have to acquire the lock for one partition ,and acquire a lock for the other partition ,before I can do any sort of operation

在我可以执行任何操作之前，我必须要获取这个分区的lock，以及另一个分区的lock

01:06:17– 01:06:18

okay

1.06.18–1.06.23

and oftentimes it's not even possible to know a priori all of the partitions and have to touch it

有时，它们甚至不用知道有哪些分区，也不用去接触它们

01:06:23 – 01:06:31

so some systems what they actually do is they'll run the system in sort of like a like a speculative or like reconnaissance mode to figure out all the partitions I need to access So, 有些系统在它们运行的时候，它们会使用一种推测或者侦查模式来弄清楚我需要访问的分区有哪些

01:06:31 – 01:06:36

and then rollback acquire all the locks ahead of time and then begin execution

接着，如果回滚的话，它提前需要获取到所有的lock，接着开始执行

01:06:36 – 01:06:39

all right,yeah

PARTITION-BASED T/O – READS

Txns can read anything that they want at the partitions that they have locked.

If a txn tries to access a partition that it does not have the lock, it is aborted + restarted.

01:06:39 – 01:06:41

so that's essentially what this is talking about,

So, 简单来讲，这就是这里所谈论的内容

1.06.41–1.06.45 ！！

if I have to access cross partitions across partition rows

如果我需要跨分区访问一些行数据

01:06:45 – 01:06:50

I have to abort and restart and acquire the new locks the new set of locks that I've discovered during execution

那我就需要中止并重启该事务，然后获取新的lock

01:06:51 – 01:06:52

and this can be wasteful

这就很浪费性能了

01:06:52 – 01:06:56

so if you you have to be kind of careful about when you want to apply partition based timestamp ordering

So, 当你使用这种Partition–Based timestamp ordering技术的时候，你就需要注意这个问题

PARTITION-BASED T/O – WRITES

All updates occur in place.

→ Maintain a separate in-memory buffer to undo changes if the txn aborts.

If a txn tries to write to a partition that it does not have the lock, it is aborted + restarted.

01:06:59 – 01:07:03

So in contrast to regular T/O and OCC

So, 与常规的Timestamp Ordering协议和OCC相比

1.07.03–1.07.05

, I can now apply all my updates in place

我可以直接在数据库中提交我的修改

01:07:06 – 01:07:09

right so when OCC and basic T/O

So, 当使用OCC和Basic Timestamp Ordering的时候

1.07.09–1.07.12

I had a private workspace that or data plot that I have to apply my updates

我需要在我的私有空间中对数据副本进行更新操作

01:07:12 – 01:07:18

so that I don't conflict with other transactions that are running concurrently that want to read the same stuff that I'm writing

So, 这样我就不会与其他并发执行的事务产生冲突, 它们想去读取我正在修改的对象数据

01:07:18– 01:07:19

okay

1.07.19–1.07.23

but here the goods the system guarantees that there's only one transaction running at a time

但在这里, 该系统保证同一时间只有一个事务在执行

01:07:24 – 01:07:25

I can apply my updates in place

我可以在数据库中直接更新数据

01:07:26 – 01:07:27

and of course

1.07.27–1.07.30

I have some extra logic to ensure that when I abort I undo those changes

我需要一些额外逻辑来确保当我中止事务时, 我要撤销这些已经执行的修改

01:07:30 – 01:07:32

but I can do this without making a local copy

但我可以在不需要制作本地副本的情况下做到这点

1.07.32–1.07.42

so I've reduced the data copying overhead that would normally exist in OCC systems

So, 这样我就减少了通常存在于OCC系统中那些数据复制所带来的开销

01:07:42 – 01:07:49

and in the case if I try to modify a tuple that exists in a different partition than when I'm running on right now

在某种情况下, 如果我试着对一个tuple进行修改, 它所在的分区与我事务所在的分区不是同一个分区

01:07:49 – 01:07:50

I'll abort

我就会中止该事务

1.07.50–1.07.51

restart

并重启该事务

1.07.51–1.07.56

get a lock on both database partitions and then begin my execution

接着，我会去获取这两个数据库分区的lock，然后开始执行该事务

01:07:56 – 01:07:56

yeah

请讲

01:08:13 – 01:08:14

no no

NO

1.08.14–1.08.17

because every database has only one transaction execution thread

因为每个数据库都只有一个事务执行线程

1.08.17–1.08.21

,every partition has one transaction execution thread

每个分区都有一个事务执行线程

01:08:23 – 01:08:26

right that's sort of one reason why you want to do this

这就是你为什么想这样做的其中一个理由

1.08.26–1.08.28

because you by having only one thread

因为如果你只有一条线程

1.08.28–1.08.32

,you don't have to copy, you don't have to have a locks and latches

你就不需要去复制数据，并且也不需要去获取lock和latch

01:08:32 – 01:08:32

yes

请讲问讲

01:08:48 – 01:08:51

yeah so I think they're ways into the remedy

So，我觉得这有办法进行补救

1.08.51–1.08.54

~~, this way as you have if you~~ if you have a deterministic order that you acquire the locks

So，如果你在获取lock时有确定的获取顺序

1.08.51–1.08.56

that's one way to alleviate this problem

这是减轻该问题的一种方式

01:08:56 – 01:08:58

but I think you're sort of alluding to the fact that

但我觉得你所暗示的东西是

1.08.58–1.08.59

,you could have the case

你可能会遇上这种情况

1.08.59–1.09.04

~~where you keep learning~~ during execution that you have that you're touching different partitions

在你执行事务的期间，你会去接触不同的分区

01:09:04 – 01:09:05

so you do a bit of work

So, 当你做了一些工作后

1.09.05–1.09.10

,realize you have to acquire a lock for a partition that you don't have

你意识到你需要去获取一个你还未接触分区所对应的lock

01:09:10 – 01:09:12

abort retry get further in the execution

接着你会中止该事务，并重新执行该事务，接着取得一些进展

1.09.12–1.09.15

realize you have to acquire a different lock and a different partition

你意识到你需要去获取一个不同分区所对应的lock

1.09.15–1.09.16

abort and then restart

你会中止并重启该事务

01:09:16 – 01:09:20

and this you hope sort of incrementally grow the set of locks that you have to acquire

你希望你逐步增加你需要获取的lock的数量

01:09:21 – 01:09:22

because you don't know a priority

因为你事先不知道要获取哪些lock

01:09:51 – 01:09:58

yeah but you see so you solve that problem by having that these transactions acquire the locks in a specific order ,and in a deterministic order

但你们能看到，如果我们以某种特定顺序来让这些事务去获取lock，我们就能解决这个问题

01:09:58 – 01:10:00

so you have let's say a partition a and B

So, 假设我们有两个分区，即分区A和分区B

1.10.00–1.10.03

you have t1 t2 try to access s a and B

接着，我们有T1和T2两个事务，它们要试着去访问A和B这两个分区

01:10:04 – 01:10:05

they realize that they need different partitions

它们意识到它们需要去获取不同分区对应的锁

1.10.05–1.10.07

then they restart the abort

接着，它们中止并重启事务

01:10:07 – 01:10:09

and now they all acquire a and they all acquire B

现在，它们就会去获取到A分区对应的lock以及B分区对应的lock

1.10.09–1.10.10

,and then they get queued up

接着，它们就会进入一个队列排队

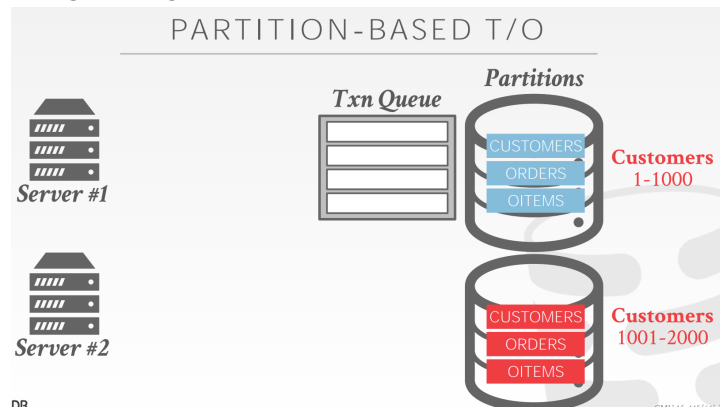
1.10.10–1.10.12

and they get executed in that order

它们就会以这个顺序执行事务

01:10:14 – 01:10:17

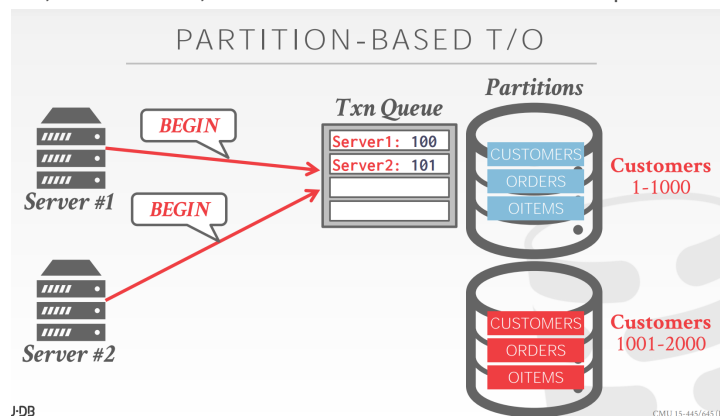
All right all right



1.10.17–1.10.21

so that's essentially partition based T/O

So, 本质上来讲, 这就是Partition Based Timestamp Ordering



01:10:21 – 01:10:23

This is sort of a visual illustration of what's going on

我们用图片来解释这里发生了什么

01:10:25 – 01:10:29

let's say you have two servers that are both trying to access some customer data in this first partition

假设这里两个服务器都要试着去访问第一个分区中的某个客户数据

01:10:30 – 01:10:32

the issue a begin request

它们发起了一个BEGIN请求

1.10.32–1.10.33

they get queued up in this transaction queue

它们就会进入一个事务队列进行排队

1.10.33–1.10.35

they get assigned timestamps

它们就会被分配时间戳

1.10.35–1.10.37

assume that server one gets timestamp 100

假设服务器1拿到的时间戳是100

1.10.37–1.10.39

server 2 it gets timestamp 101

服务器2拿到的时间戳是101

01:10:42 – 01:10:43

obviously

显然

1.10.43–1.10.48

the database system will take the transaction with the lowest timestamp in this case the the one from server1

数据库系统会先执行时间戳较低的那个事务，即这个例子中服务器1所发起的那个事务

01:10:48 – 01:10:49

it begins execution

它开始执行事务

1.10.49–1.10.53

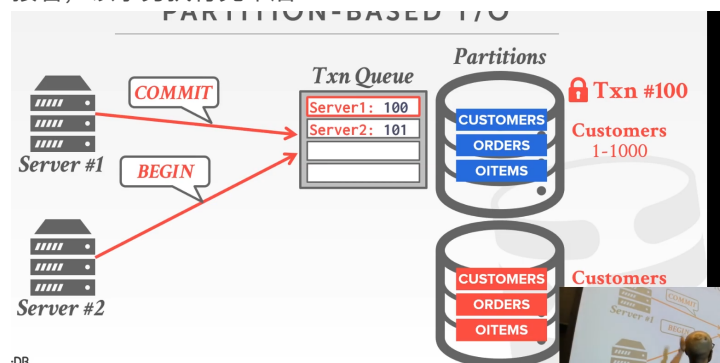
acquired by acquiring a lock on this on this database partition

它会去获取这个数据库分区对应的lock

1.10.53–1.10.57

, it finishes its execution

接着，该事务执行完毕后



01:10:57 – 01:11:02

~~it does it sorry~~ it's doing a bunch of work it's getting the customer ID, and it's updating the stuff the animations are a little bit wonky

它这里做了一系列工作，它拿到了customer id，它对该tuple进行更新，这里的动画有点不对

01:11:02 – 01:11:03

But then it does a commit

但接着，服务器1提交了该事务

1.11.03–1.11.06

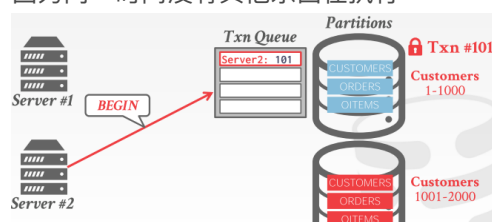
,it's safe

这样做是安全的

1.11.06–1.11.08

, because nothing else is running at the same time

因为同一时间没有其他东西在执行



01:11:08 – 01:11:15

so now this partition can now move up this this request from server2 up in the queue

So，现在服务器2所发出的这个请求就可以排到队列的前面

1.11.15–1.11.18

, begin execution acquire the lock

开始执行该事务，并获取lock

01:11:18 – 01:11:20

and then the whole cycle continues

然后执行这整个过程

01:11:22 – 01:11:22

all right

PARTITIONED T/O – PERFORMANCE ISSUES

Partition-based T/O protocol is fast if:
→ The DBMS knows what partitions the txn needs before it starts.
→ Most (if not all) txns only need to access a single partition.

Multi-partition txns causes partitions to be idle while txn executes.

01:11:26 – 01:11:29

as we've sort of talked about a little bit,

因为我们已经讨论过一些关于Partitioned Timestamp Ordering的东西了

1.11.29–1.11.32

these systems obviously are not a silver bullet

这些系统显然并不是什么万金油

01:11:32 – 01:11:34

right they have some performance issues

它们存在着一些性能上的问题

01:11:34 – 01:11:42

so these systems are very fast, if the database knows what partitions the transaction needs before even beginning execution

如果数据库在开始执行这些事务前就知道这些事务需要用到哪些分区，那么这些系统的速度就会很快

01:11:43 – 01:11:46

that's not really possible in a sort of an interactive transaction protocol right

在这种指令式事务协议中，这种做法并不现实

01:11:47 – 01:11:50

but you can remedy this by having stored procedures, right

但你可以通过存储过程来补救一下

01:11:50 – 01:11:51

in store procedures

在存储过程中

1.11.51–1.11.57

~~you declare everything on you~~ you don't have to do round trips back and forth

你无须来回发送请求

01:11:57 – 01:11:58

and the database runs everything on the server side,

数据库会在服务器端执行所有东西

1.11.58–1.12.06

so can determine much quicker all of the locks the system needs before enduring execution

So，这样可以更快地确定在事务执行过程中事务需要的lock有哪些

01:12:06 – 01:12:12

and if the transaction only touches one partition

如果该事务只接触一个分区

1.12.12–1.12.13

then obviously it's gonna be really fast

那么，显然这样就会非常快

01:12:13 – 01:12:15

but if it has to touch multiple partitions

但如果该事务要接触多个分区

1.12.15–1.12.20

and you have to do some execution abort retry by acquiring more locks

你就必须先执行一下该事务，接着中止该事务，获取更多的lock并重新执行该事务

1.12.20–1.12.21

,and then continue execution is again right

接着，继续执行该事务

01:12:22 – 01:12:22

so it becomes slower

So，这样系统就会变得更慢

01:12:23 – 01:12:26

But if your transaction only touches data within one partition

如果你的事务只接触一个分区中的数据

1.12.26–1.12.28

, then you're essentially running bare metal speed

本质上来讲，你的执行速度就是裸机的速度

01:12:28 – 01:12:31

because it's single threaded right

因为它就是单线程执行

01:12:31 – 01:12:33

there's no locks, no latches

这里面没有lock，也没有latch

01:12:34 – 01:12:35

the other drawback is that

另一个缺点则是

1.12.35–1.12.38

if you have this multi partition setup

如果你拥有这种多分区设置

1.12.38–1.12.42

you could have some partitions that are essentially idle

你的有些分区可能就处于闲置状态

1.12.42–1.12.45

because you have one hot partition

因为你只有一个分区在被人访问

DYNAMIC DATABASES

Recall that so far we have only dealing with transactions that read and update data.

But now if we have insertions, updates, and deletions, we have new problems...

01:12:49 – 01:12:49

okay

1.12.49–1.12.53

so that that was essentially partition based T/O

So，本质上来讲，这就是Partitioned–Based Timestamp Ordering

1.12.53–1.12.58

, try to get through this as fast as possible

我试着尽可能快地讲下这个

01:12:57 – 01:13:00

so one of the assumptions that we've been making throughout this work is that

So, 这里我们之前已经做了一个假设

1.13.00–1.13.03

the as transactions execute

当事务执行的时候

01:13:03 – 01:13:05

they don't really insert new data

它们不会真正地插入新数据

1.13.05–1.13.08

, they only modify data ,and they read data

它们只是修改数据和读取数据

01:13:08 – 01:13:08

okay

1.13.08–1.13.10

so when you now add the requirement

So, 当你要添加这样的需求时

1.13.10–1.13.17

the data can it can be inserted, and updated ,and deleted and inserted during execution

即在执行事务的过程中, 数据可以被插入, 被更新, 被删除

01:13:17 – 01:13:21

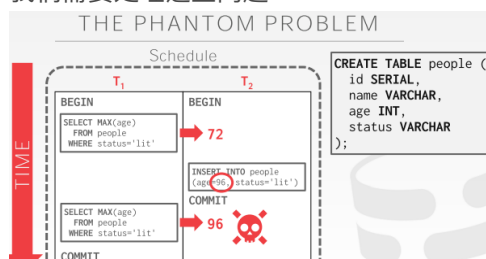
it violates some of the assumptions that we've been there that we've made in our protocols

这就违反了我们协议中所制定的那些假设

1.13.21–1.13.22

and we have to handle them

我们需要处理这些问题



01:13:22 – 01:13:23

okay

1.13.23–1.13.26

and specifically introduces an interesting problem

特别是, 这里面引入了一个我们感兴趣的问题

01:13:27 – 01:13:29

that's called **the phantom problem**

这叫做幻读问题

01:13:29 – 01:13:30

so imagine that

So, 想象一下

1.13.30–1.13.32

we have the same sequence of execution here,
这里我们拥有和之前相同的执行序列

1.13.32–1.13.35

but now I'm using SQL statements instead of regular read and write operations
但现在我这里使用的是SQL语句，而不是那些常规的读写操作

01:13:36 – 01:13:38

and we have these two transactions t1 and t2 ,
我们有两个事务，即T1和T2

```
CREATE TABLE people (  
  id SERIAL,  
  name VARCHAR,  
  age INT,  
  status VARCHAR  
);
```

1.13.38–1.13.44

and we have this database here of people that have you know **name** and **age** and some **status**

我们一张people表，它里面的字段有name，age以及status

T₁

```
BEGIN  
  
SELECT MAX(age)  
  FROM people  
 WHERE status='lit'
```

01:13:44 – 01:13:50

so a transaction t1 that's performing that's getting the maximum the the oldest person
whose status is lit, okay

So, T1所做的事情就是获取年纪最大且状态为lit的人

T₁ **T₂**

```
BEGIN  
SELECT MAX(age)  
  FROM people  
 WHERE status='lit'
```

→ 72

01:13:51– 01:13:55

so let's assume that I runs ,and it figures out that the maximum age is 72

So, 假设我们执行了这条SQL语句，并且知道了年纪最大值为72

01:13:55 – 01:13:56

and then we have this transaction t2

接着，我们还有一个事务T2

1.13.56–1.14.03

,that's essentially going to insert it into the same database with a new person whose age is 96 ,who also happens to be lit

本质上讲, T2会往同一个数据库中插入一个年龄为96, 状态为lit的新纪录

01:14:04 – 01:14:04

all right

1.14.04–1.14.06

and then we get back here

接着, 我们切换回T1

1.14.06–1.14.10

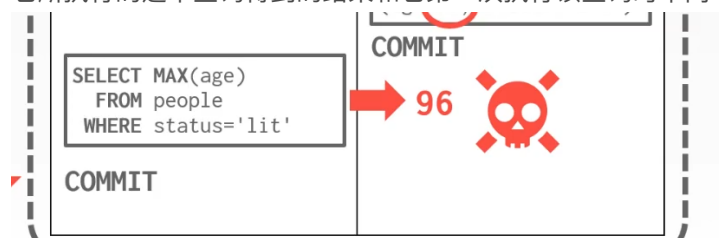
so it's the same t1 tries to re-execute the same query

So, T1试着去执行和刚才相同的查询

01:14:11 – 01:14:15

and it gets a different version it gets a different result from this query

它所执行的这个查询得到的结果和它第一次执行该查询时不同



01:14:15 – 01:14:16

so got 96 here

So, 它得到的结果是96

1.14.16–1.14.18

and it got 72 here

之前, 它所得到的结果是72

1.14.18–*1.14.20

today this is clearly a problem,right

很明显, 这就是一个问题

01:14:21 – 01:14:24

can2 phase locking solve this

两阶段锁能解决这个问题吗

01:14:36 – 01:14:37

assume tuple access

假设对tuple加锁

01:14:39 – 01:14:40

who said no

有谁说了No

1.14.40–1.14.41

,why do you say no

你为什么说No

01:14:49 – 01:14:52

yeah exactly that's it that's exactly right right

说得完全正确

WTF?

How did this happen?

→ Because T_1 locked only existing records and not ones under way!

Conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed.

01:14:52 – 01:14:56

you can only access locks on tuples that exists

只有当tuple上存在lock的时候，你才能访问该tuple

01:14:59 – 01:15:02

so because there was new transaction that inserted a new tuple

So, 因为这里有一个新事务，它要往数据库中插入一个新tuple

1.15.02–1.15.05

they didn't even have something to acquire a lock on, right

它们中的操作不需要用到lock

01:15:06 – 01:15:07

and that's clearly a problem

这很明显就是一个问题

1.15.07–1.15.08

and how do we solve this

我们该如何解决这个问题

1.15.08–1.15.10

if I want to ensure it's a realisability

如果我想确保它是可实现的

01:15:10 – 01:15:14

I have to ensure that I can ensure I can ensure repeatable reads, so if you saw this phantom problem

So, 如果我遇上这种幻读问题，那我需要确保这里可以进行可重复读

01:15:15 – 01:15:16

okay

1.15.16–1.15.19

so what if instead of acquiring locks on tuples

So, 我无须去获取tuple所对应的lock

1.15.19–1.15.22

,I can acquire locks on abstract objects

我可以去获取那些抽象对象对应的lock

01:15:22 – 01:15:28

what if I can acquire a lock on an expression status Lit ,would that solve the problem

如果我可以获取某个表达式上 (status = "lit") 的lock, 这能解决这个问题吗

01:15:29 – 01:15:33

if I had a lock on all tuples that satisfy the condition status is equal to lit

如果我拿到了所有满足该条件 (status='lit') tuple对应的lock

01:15:35 – 01:15:36

who says yes

谁说了Yes?

1.15.36–1.15.39

one person

有一个人

1.15.39–1.15.39

who says no

有谁说了No?

1.15.39–1.15.42

who says they don't know

你们中有谁说他们不知道的吗?

01:15:44 – 01:15:45

okay so the answer is yes

So, 答案是Yes

1.15.45–1.15.48

if you could have a powerful enough locking system

如果你有一个足够强大的locking系统

1.15.48–1.15.52

that you can have an expression that says status is equal to lid

那么你就可以通过一个表达式来表示status='lit'

01:15:52 – 01:15:54

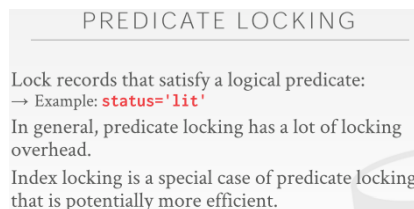
you can solve this problem

你就可以解决这个问题

1.15.54–1.15.55

that would solve it for you

这会为你解决这个问题



01:15:56 – 01:15:56

all right

1.15.56–1.16.00

~~but the problem is~~ and that the term that we're talking about here is called predicate locking

这里我们讨论的术语叫做条件锁 (predicate locking)

01:16:01 – 01:16:04

that it's very very expensive to be able to do this

实现它的成本非常非常的高

1.16.04–1.16.07

and in fact I don't think any systems actually do that type of really complex logic

事实上, 我并不相信有任何系统实际做到了这种复杂的逻辑

01:16:07 – 01:16:09

because it's very complex right

因为它实在是太复杂了

01:16:10 – 01:16:11

I gave you one example of status='lit'

我之前给你们看过status='lit'这个例子

1.16.11–1.16.14

,but we're actually talking about it's a multi-dimensional problem

但我们实际讨论的是一个多维问题

1.16.14–1.16.17

because you could have any arbitrary complex expression

因为你可能会有某种任意复杂度的表达式

01:16:17 – 01:16:22

and you want to ensure that the expression you're evaluating doesn't intersect with this multi-dimensional space

你想确保你评估的这个表达式不与这种多维空间相交

01:16:23 – 01:16:24

so it's non-trivial to implement

So, 它实现起来并不容易

1.16.24–1.16.25

so most systems don't do that

So, 大部分数据库系统都不会去使用它

1.16.25–1.16.27

yes

请讲

01:16:32 – 01:16:36

yeah so you could acquire like a table level lock or a page level lock

So, 你可以去获取表级别的lock或者page级别的lock

01:16:43 – 01:16:48

yeah so you could use these hierarchy locks that's one way to definitely solve it

So, 你可以使用这种层级锁 (hierarchy lock) , 使用它你肯定能解决这个问题

01:16:48 – 01:16:48

the other way to solve it is

解决该问题的另一种方式是

1.16.48–1.16.51

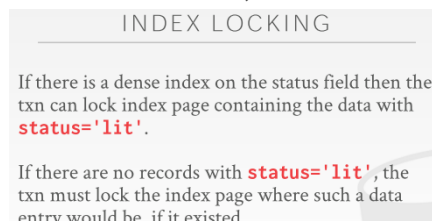
is this predicate locking that I was talking about which is more general-purpose

就是使用我刚才讨论过的条件锁, 它更加通用

01:16:53 – 01:16:56

And there's another one called index locking

这里还有另一个东西, 它叫做index locking



INDEX LOCKING

If there is a dense index on the status field then the txn can lock index page containing the data with **status='lit'**.

If there are no records with **status='lit'**, the txn must lock the index page where such a data entry would be, if it existed.

01:16:56 – 01:16:58

so if I have an index on the system

So, 如果在系统中, 我有一个索引

01:16:58 – 01:17:00

right let's say I have an index on this on the status attribute

假设, 我们在status这个字段上建立了索引

1.17.00–1.17.01

,what I could do is

我所能做的事情就是

1.17.01–1.17.06

take a lock on the slot in the index where status is equal to lit

我会对索引中status="lit"的slot加锁

01:17:06 – 01:17:10

so that any new insertions have to go through the index and make an update

So, 任何新的插入操作都需要去遍历索引, 并对索引进行更新

1.17.10–1.17.13

they can't acquire that lock

它们无法获取到那个lock

01:17:13 – 01:17:13

okay

1.17.13–1.17.15

so that's another way to solve it

So, 这是解决该问题的另一种方式

1.17.15–1.17.18

if the status lit doesn't even exist in the index

如果status="lit"并不存在该索引中

1.17.18–1.17.21

what I have to do is acquire what's called a **gap-lock**

我需要做的就是获取一个叫做Gap Lock (间隙锁) 的东西

01:17:21 – 01:17:24

so a gap in the index I acquire a lock on that gap

So, 在索引中有一个空隙, 我获取了该空隙对应的lock

01:17:25 – 01:17:30

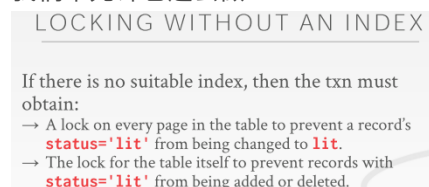
so that if another insertion comes in that tries to insert statuses is equal to lit in that space in the index

So, 如果另一个插入操作试图将符合 (status='lit') 条件的tuple插入索引中的这个空隙

01:17:31 – 01:17:32

it's not allowed to do that

我们不允许它这么做



01:17:34 – 01:17:40

okay so this that's that's two essentially two ways to do it

Ok, 本质上来讲, 我们可以通过这两种方式来做做到这点

01:17:43 – 01:17:46

yeah so this essentially goes back to the point that was made earlier ,

So, 本质上来讲, 这又回到我们之前说的一件事了

01:17:46 – 01:17:46

if you don't have an index

如果你没有索引

1.17.46–1.17.48

and you don't have predicate locks

如果你也没要Predicate lock (条件锁)

1.17.48–1.17.56

, you could do it by locking every every page ,or you could do it by locking the entire table ,or yeah hierarchical locks

那么你可以通过对每个page、或者整张表加锁来做到这点, 或者使用hierarchical lock (层级锁)

REPEATING SCANS

An alternative is to just re-execute every scan again when the txn commits and check whether it gets the same result.
→ Have to retain the scan set for every range query in a txn.
→ Andy doesn't know of any commercial system that does this (only just Silo?).

01:17:57 – 01:18:00

so I saw you know the last way to solve this problem is

So, 解决这个问题最后一种方法是

1.18.00–1.18.05

through just by repeating the scans ~~by by the committing~~ sorry before committing
在事务提交前，我们反复进行扫描

01:18:05 – 01:18:09

so I think one of the most popular systems that does this is **hekaton** which is from Microsoft

So, 其中一个使用了这种做法的系统就是微软的hekaton

01:18:09 – 01:18:11

so essentially everything before you commit

So, 本质上来讲，在你提交事务前

1.18.11–1.18.17

you have to make sure that everything you read before happens just before you commit
to make sure can see you have consistent reads

你必须确保你读取到的所有东西都是在你提交事务前读取的（知秋注：即读的数据在你提交这一刻，它就是最新的，若不是，就重启事务），这样可以确保你读取到的内容是一致的

01:18:17 – 01:18:18

I think we're out of time

我觉得我们的时间到了

1.18.18–1.18.24

, the last section was weaker isolation levels maybe we'll cover that in the next lecture
最后要讲的一部分就是Weaker Isolation Level，我们可能会在下节课的时候讲

01:18:25 – 01:18:28

all right all right that's it

All right, 我讲完了