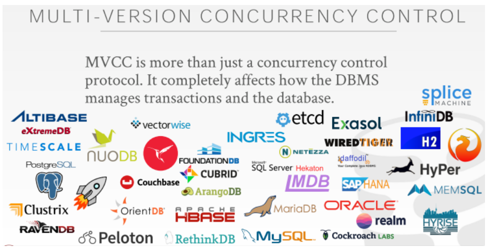
19–02


MULTI-VERSION CONCURRENCY CONTROL

MVCC is more than just a concurrency control protocol. It completely affects how the DBMS manages transactions and the database.

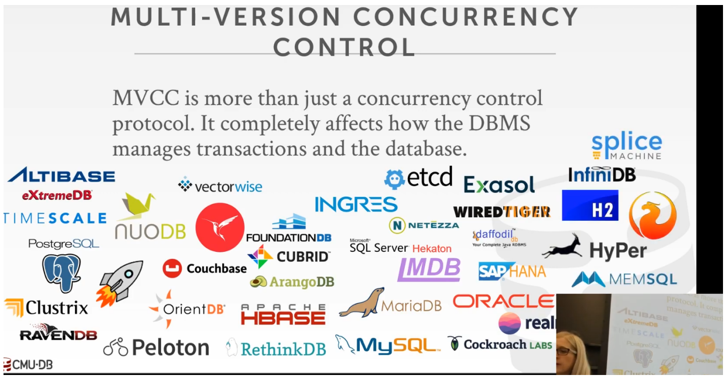
ALTIBASE
eXtremeDB
TIMESCALE
PostgreSQL
Clustrix
RAVENDB
NUODB
vectorwise
Couchbase
FOUNDATIONDB
CUBRID
ArangoDB
OrientDB
Peloton
APACHE HBASE
RethinkDB
INGRES
NETEZZA
SQL Server Hekaton
IMDB
MariaDB
MySQL
etcd
Exasol
WIREDTIGER
daffodil
SAP HANA
ORACLE
realm
Cockroach LABS
splice MACHINE
InfiniDB
H2
HyPer
MEMSQL
HYRISE

20:03 – 20:04
all right

20.04–20.04
so as I mentioned
So，正如我提到的
20.04–20.06
~~if you say~~ oh yes
请讲
~~20:28 – 20:28~~
~~And you say~~

20:41 – 20:45
oh well so this again this is a very high–level example right now
Oh，Well，这是一个高级层面的例子
20:45 – 20:51
and there’s actually we’re going to go into how you actually store this information later on in this lecture
实际上，我们稍后会在这节课上讨论实际如何存储这些信息
20:51 – 20:55
~~so that’s with you like ,~~so that will answer the question
So，我觉得你听了之后，就知道这个问题的答案了
20:56 – 20.56
basically
简单来讲
20.56–20.58
it’s in some cases
在某些例子中
20.58–21.01
yes, you do need to consider locks
你确实需要思考锁方面的问题
21:01 – 21:05
it really depends on how you’re actually storing this version information
这取决于你实际是如何保存这些版本信息
21:06 – 21:09

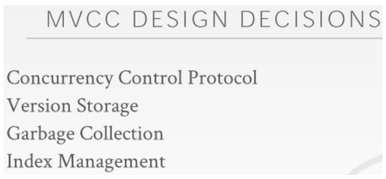
so we don’t cover it in a few slides
So，我们不会用几张幻灯片来介绍下它的话
21.09–21.11
, because there’s multiple ways to do this
因为我们可以通过多种方法来做到这点
21:11 – 21:13
so I don’t want to just list them all out now
So，现在我并不想将它们都列出来
21.13–21.15
we don’t cover it in few slides
我们不会通过一些幻灯片来介绍它
21.15–21.16
 please ask your question again
请再次提下你的问题


MULTI-VERSION CONCURRENCY CONTROL

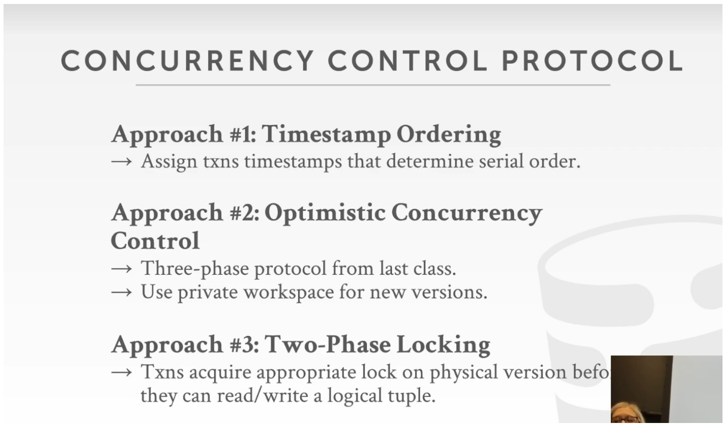
MVCC is more than just a concurrency control protocol. It completely affects how the DBMS manages transactions and the database.

21:22 – 21:23
all right

21.23–21.30
so um so again like MVCC or its variants are used in almost all new database systems
So，几乎所有新的数据库系统都使用了MVCC或者它的变体
21:30 – 21:35
and these are just you know some examples of the systems that use MVCC
这些是使用了MVCC的一些数据库系统
21:36 – 21:40
but what we really want to emphasize for the rest of this lecture, is that
但我们想在这节课上剩余时间里强调的东西是
21.40–21.46
 MVCC is a lot more than just maintaining the timestamps that I showed you in the previous two examples
除了我们前两个例子中维护时间戳这方面事情以外，MVCC还有很多其他东西
21:47 – 21:53
there’s a whole bunch of other design decisions, that you have to make in order to actually implement a system that supports MVCC
实际上，为了实现一个支持MVCC的系统，你还需要做一些其他方面的设计决策
21:53 – 21:55
so we’re gonna go over those next
So，接下来，我们会去看下这些东西


MVCC DESIGN DECISIONS

Concurrency Control Protocol
Version Storage
Garbage Collection
Index Management

21:55 – 21:58
so what exactly are these design decisions
So，这些设计决策有哪些呢?
21:59 – 22:03
specifically it’s what concurrency protocol you’re going to use
特别是，你要去使用什么并发协议呢?
22:03 – 22:06
how you’re going to maintain and store the different versions
你要如何维护并存储这些不同的数据版本
22.06–22.08
, which relates to the question that was previously asked
这也和你之前问的问题相关
22:10 – 22:15
how are you’re going to clean up the old versions ,once they’re not visible to any transactions anymore
一旦某些老版本数据不再对这些事务可见，我们该如何清理这些老版本数据呢?
22:16 – 22:20
and how you’re going to ensure that the indexes point to the correct version
你该如何确保这些索引指向的是该数据的正确版本呢?

CONCURRENCY CONTROL PROTOCOL

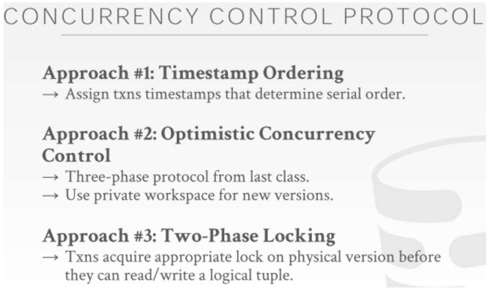
Approach #1: Timestamp Ordering
→ Assign txns timestamps that determine serial order.

Approach #2: Optimistic Concurrency Control
→ Three-phase protocol from last class.
→ Use private workspace for new versions.

Approach #3: Two-Phase Locking
→ Txns acquire appropriate lock on physical version befo
they can read/write a logical tuple.

22:23 – 22:24
all right

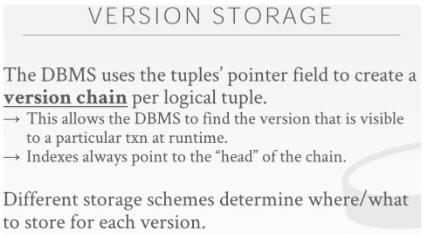
22.24–22.27
so the first thing we’ll cover is
So，首先我们要介绍的是
22.27–22.30
a skip that one okay
我点错了
22:30 – 22:36
so that’s what I thought okay, so the first thing we’re going to cover is concurrency control protocol
So，我们首先要介绍的是并发控制协议
22:41 – 22:45
right so this is basically I’m looking at the ,sorry

抱歉，我点错幻灯片了
22:46 – 22:48
I'm not used to this presenter view my bad ,okay
我不应该使用这种演讲者模式，这是我的错

CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering
→ Assign txns timestamps that determine serial order.

Approach #2: Optimistic Concurrency Control
→ Three-phase protocol from last class.
→ Use private workspace for new versions.

Approach #3: Two-Phase Locking
→ Txns acquire appropriate lock on physical version before they can read/write a logical tuple.

22:51 – 22:52 ！！
concurrency control protocol
来说下并发控制协议
22.52–22.56
this is our first consideration right,for our design decisions
这是我们设计决策中要首先考虑的一个东西
22:57 – 23:05
so these are the concurrency control protocols that you guys have been studying for the past two weeks in the past two lectures
So，你们在前两周的课上已经学过了这些并发控制协议
23:05 – 23:09
And again when you encounter a write write conflict
当你遇上Write-Write Conflict时
23.09–23.11
you need to use one of these protocols
你需要使用其中一种协议
23:11 – 23:15
whether it be two phase locking，OCC or timestamp ordering
它可以是两阶段锁、OCC或者是Timestamp Ordering
23.15–23.21
to figure out which transaction should be allowed to write to that object
以此来弄清楚哪个事务能对该对象进行写入操作
23:21 – 23:23
and what isolation level you're running at
以及你使用的隔离级别是什么
23:23 – 23:26
so we're not going to go into much detail on this
So，我们不会对此太过深入
23.26–23.29
since you've just been covering it very recently
因为你们最近已经学过了

VERSION STORAGE

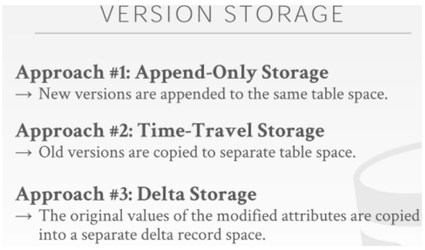
The DBMS uses the tuples' pointer field to create a **version chain** per logical tuple.
→ This allows the DBMS to find the version that is visible to a particular txn at runtime.
→ Indexes always point to the "head" of the chain.

Different storage schemes determine where/what to store for each version.

23:30 – 23:33
so the next consideration is version storage

So，我们下一个要考虑的东西是Version Storage（版本存储）
23:34 – 23:37
so for version storage what we need to do is
So，对于版本存储来说，我们要做的事情是
23.37–23.44！！！
figure out for a particular tuple version ,what should actually be visible to us, right
我们需要弄清楚某个tuple的哪个版本对我们来说是可见的
23:45 – 23:50
so let’s assume for now that we’re doing a sequential scan on the entire table
So，假设我们要对整张表进行循序扫描
23.50–23.55
, and we want to know where to find the version of a tuple that we want
我们想知道我们该在哪找到我们想要的那个版本的tuple
23:55 – 23.56
so the way we’re going to implement this is
So，我们实现这个功能的方式是
23.56–23.59
we’re going to maintain an internal pointer field
我们会去维护一个internal pointer字段
23.59–24.03
, that will allow us to find the previous or next version
这使得我们能找到该tuple的前一个版本或者下一个版本
24:03 – 24:07
we’ll go into that more for this particular logical tuple
我们会对这个逻辑tuple的不同版本进行深入

24:07 – 24:10
so you can think of this is sort of a linked list
So，你可以将其想作是一个链表
24.1024.17
where you know you can jump on into you jump on it and land on the head
你知道的，你可以跳到这个链表中，你可以找到它的头结点
24:17 – 24:25
and then you can follow the the pointers in the linked list to find all of the different versions they’re currently being maintained
接着，你可以通过该链表中的指针来找到它们当前所维护的该数据的不同版本
24:29 – 24:33
so indexes always point to the head of the chain
So，索引指向的始终是该链表的头节点
24:36 – 24:40
and oh my I did it again ,I’m so sorry okay
抱歉，我又出错了
24:44 – 24:46
technical difficulties
技术性问题emmm
24:52 – 24:54
all right~~ so um~~

24:55 – 24.58 ！！
indexes like it says here will always point to the head of the chain,
正如我说的，索引指向的始终是该链的头节点
24.58–25.03
 whether the head is the oldest version that the newest version of that tuple depends on the implementation
取决于具体实现，这个头节点可以是该tuple的最老版本，也可以是它的最新版本
25:04 – 25:09
so there’s different approaches determine how we’re going to store these different versions
So，不同的方案决定了我们存储这些不同版本的方式
25:09 – 25:11
so we’ll go more into that next
So，我们稍后会对此进行深入

VERSION STORAGE

Approach #1: Append-Only Storage
→ New versions are appended to the same table space.

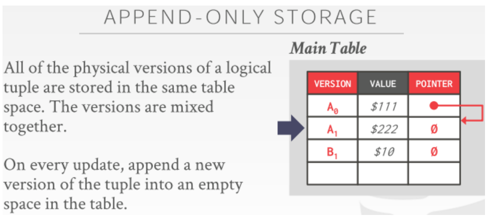
Approach #2: Time-Travel Storage
→ Old versions are copied to separate table space.

Approach #3: Delta Storage
→ The original values of the modified attributes are copied into a separate delta record space.

25:19 – 25:24
so the first simplest approach is called **append-only storage**
So，首先要讲的最简单方案叫做Append-Only Storage
25:24 – 25:24
all right

25.24–25.27
so this just means that
So，这意味着
25.27–25.30
every time we create a new version
每当我们创建某个数据的新版本时
25:30 – 25:36
we just copy the old version as a new physical tuple in our tablespace and update it
我们只需复制该tuple的老版本，并将该副本作为我们表空间中的一个新物理tuple，并对其进行更新
25:37 – 25:40
so then we update pointers to say
So，接着，我们更新指针并说
25.40–25.40
here’s the next version
这是该tuple的下一个版本
25.40–25.44
and we’re going to go over examples of all three of these in the next few slides
在接下来的一些幻灯片中，我们会去看下关于这三种方案所对应的案例
25:45 – 25:49
so the next approach is called **time-travel storage**
So，下一种方案叫做Time-Travel Storage
25.49–25.52

 and this is where you have one master version table
在这种方案中，我们有一个master version table
25:54 – 25:59
that there’s always duringstoring the latest version of the object or tuple
它里面所保存的始终是对象或者tuple的最新版本
26:00 – 26:04
then you copy out older versions into a separate table
然后，我们会将这些老版本数据复制到一张单独的表上
26.04–26.06
that we’re going to call the time–travel table
我们将这张表叫做Time–Travel表
26:06 – 26:08
and then at that point
在此时
26.08–26.14
you just maintain the pointers from the master version of the table with the latest tuples to the time–travel table
你需要做的就是去维护master version表中指向Time–Travel表的指针
26:16 – 26:22
and so the last approach which is the one Andy prefers and things as best is called delta storage
So，最后一种是Andy所倾向的一种方案，同时它也是最佳方案，它叫做Delta Storage
26:23 – 26:27
so you can think of this as a Gives and get
So，你可以将它想象为是一种Gives and Get策略
26.27–26.31
, we’re instead of just copying the old version every single time and updating it
我们不用每次都去复制这些老版本数据，然后在它们的副本上进行更新
26:31 – 26:37
, you’re just going to maintain you know a small delta of the modifications from the previous version .
我们只需去维护那些对前一个版本所做的修改即可



APPEND-ONLY STORAGE

All of the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.

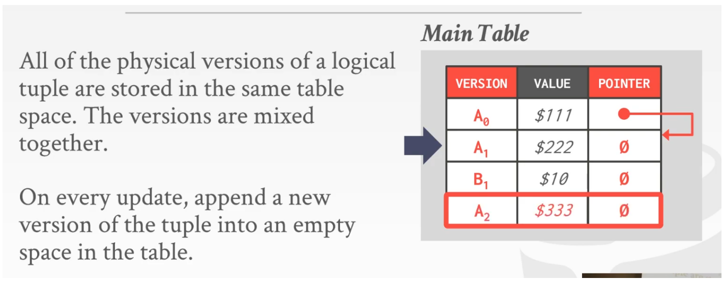
On every update, append a new version of the tuple into an empty space in the table.

Main Table

VERSION	VALUE	POINTER
A_0	$111	●
A_1	$222	Ø
B_1	$10	Ø

26:39 – 26:43
so well first go over in an example of the **append–only storage**
So，首先我们来看下这个关于Append–Only Storage的例子
26:44 – 26:45
so again this is the simplest approach
So，这是一种最简单的方案
26.45–26.47
and this is also what Postgres uses
这也是PostgreSQL所使用的方案
26:48 – 26:54

so each physical version is just a new tuple in the main table
So，每个物理版本其实就是Main Table中的一个新tuple
26:56 – 27:00
so let's say we have a transaction here that wants to update object a ,right
So，假设我们有一个事务，它想去更新对象A
27:01 – 27:03
So the first thing that's going to do is
So，首先它要去做的事情是
27.03–27.08
it's going to find an empty slot in the tablespace
它会在表空间中找到一个空的slot
27:08 – 27:12
and then copy the values from the current value of A which is A1
接着，它会去复制A的当前值，即A1
27.12–27.13
, all right that's the most recent value
All right，这是A的最近值
27:14 – 27:17
I into that table slot
并将这个值放入表中这个空的slot中


All of the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.

On every update, append a new version of the tuple into an empty space in the table.

Main Table

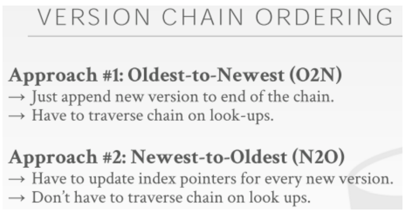
VERSION	VALUE	POINTER
A_0	$111	●
A_1	$222	Ø
B_1	$10	Ø
A_2	$333	Ø

27:18 – 27:24
and then next it's going to copy the modified value into that table slot
接着，它会将这个修改后的值放入表中这个slot中
27:26 – 27:27
and are we done yet
我们完事了吗?
27.27–27.27
not quite
还没有
27.27–27.29
the final thing we need to actually do is
实际上，我们需要做的最后一件事情就是
27.29–27.37
update the pointer to point from the older version to the newest version that we currently installed
我们要去更新指针，并让它指向我们当前插入的最新版本数据
27:51 – 27:51
okay

27.51–27.56
so another aspect we must consider your in order to store this version

So，为了保存这些版本，我们必须考虑的另一个方面是
~~27.56=~~
~~so~~

27:56 – 27.57
Oh in this example
在这个例子中
27.57–28.05
A is considered the head of the version chain
A被认作是该version chain的头节点
28:06 – 28:07
and in this example
在这个例子中
28.07–28.11
we’re specifically ordering these oldest to youngest ,right
我们会根据从旧到新的顺序对这些版本进行排序

28:11 – 28:15
so an alternative would be you can order them youngest to oldest
So，你也可以按照从新到旧的顺序对它们进行排序
28:17 – 28:22
so if you’re looking for the newest tuple in this case
So，在这个例子中，如果你要查找某个tuple的最新版本
28:22 – 28:29
you actually get to the point of where your get to version A zero
实际上，进入这张表的时候，你所拿到的版本是A0
28:29 – 28:33
and again you have to follow the pointers ,all the way down to the newest version A2
你必须沿着这些指针一路向下，找到A的最新版本A2
28.33–28.34
so make sense?
So，懂了吗

VERSION CHAIN ORDERING

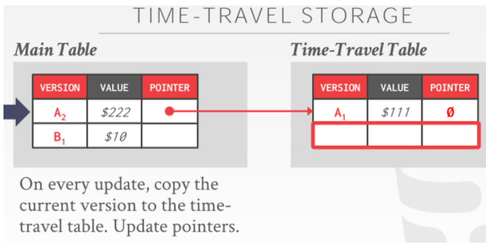
Approach #1: Oldest-to-Newest (O2N)
→ Just append new version to end of the chain.
→ Have to traverse chain on look-ups.

Approach #2: Newest-to-Oldest (N2O)
→ Have to update index pointers for every new version.
→ Don't have to traverse chain on look ups.

28:38 – 28:39
all right

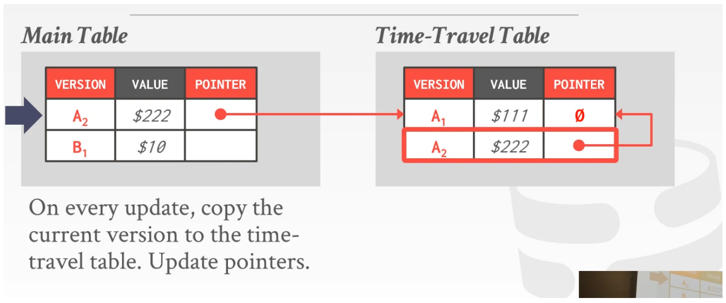
28.39–28.42
so um so like I just said
So，正如我说的
28.42–28.45
the previous example used oldest to newest
上一个例子中，我们是按照从旧到新的顺序进行排序
28:45 – 28:47
but you could also use newest to oldest
但你也可以按照从新到旧的顺序进行排序

28.47–28.51
and there’s performance implications and trade offs for both of them right
对于它们来说，这存在着性能上的影响和权衡
28:52 – 28.53
so with oldest to newest
So，如果你用的是从旧到新这种
28.53–28.57
,all you need to do
你所需要做的事情是
28.57–29.02
when there’s a new version is to just append to the end of the version chain right
当该数据的新版本追加到这个version chain的末尾时
29:02 – 29:10
this is very simple append the new tuple and update the pointer to point to the the newer version from the older version to the newer version
我们可以很容易地追加这个新tuple，让原本指向老版本的指针更新，从较老版本指向该tuple的新版本
29:11 – 29:14
and this is a really easy operation to do
这是一个很容易执行的操作
29:17 – 29:18
but if you do newest to oldest
但如果你使用的是从新到旧的方式
29.18–29.21
then what this means is that
那么，这意味着
29.21–29.23
 you have to add the entry
你添加这个条目的时候（即这个tuple的新版本）
29.23–29.26
,and update its pointer to point to the old head ,right
要将该条目的指针指向原来那个头节点
29:26 – 29:31
but now you have to actually update all of the indexes to point to your new version
但现在你需要更新全部索引，让它们指向你的新版本
29.31–29.33
so it’s again like we said a few slides ago
So，就像我们前几张幻灯片中所讲的那样
29:34 – 29:38
indexes always point to the head of the version chain, right
索引指向的始终是version chain的头节点
29:38 – 29:41
so this means a lot more updates in some cases
So，这意味着，在某些情况下，我们要做大量更新



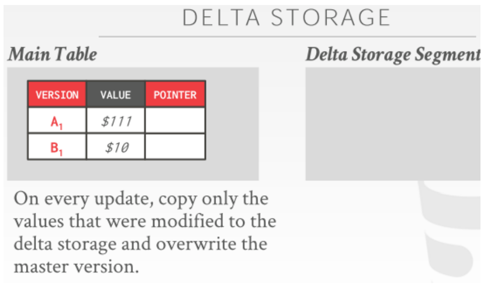
On every update, copy the current version to the time-travel table. Update pointers.

29:43 – 29:44
all right

29.44–29.47
so for time–travels storage
So，来讲下Time–Travel Storage
29.47–29.49
this is the next approach we’ll cover
这是我们要讲的下一种方案
29.49–29.51
and here we’re going to have a main table
这里我们有一个Main表
29.51–29.54
that always has to latest version of each tuple
它上面保存的始终是每个tuple的最新版本
29:55 – 29:58
and then we’ll have another table called the time travel table
接着，我们还有另一张叫做Time–Travel的表
29:59 – 30:06
and this is where we’re going to maintain older versions and copy older versions as they get modified in the database,right
当数据库中的tuple被修改的时候，我们要将旧版本数据复制到这张表上，并维护这些旧版本数据
30:07 – 30:08
so for this example
So，在这个例子中
30.08–30.15
let’s say the transaction wants to update object A again same as last example
和上一个例子一样，假设事务想去更新对象A
30:15 – 30:21
then we’re going to copy A2 into the free spot in the time travel table
接着，我们将A2复制到Time–Travel表的空闲位置
30:21 – 30:26
and then update the version pointer to point to the oldest version of tuple a
接着，我们要去更新版本指针，以此来让它指向tuple A的最老版本


On every update, copy the current version to the time-travel table. Update pointers.

30:28 – 30:32 ！！！！！
then we’re going to overwrite the master version in the main table to be the new version value
接着，我们会覆盖Main表中的master version，将它变为新的版本号
~~30:34 – 30:39~~
~~and finally we need to update the air diversion by~~

30:34 – 30:49
and then finally we need to update the pointer to point from the new version A3 to the version that we just installed in the time travel table which is A2
最后我们需要更新指针，将指向A3的指针指向我们刚在Time–Travel表中插入的A2
30:57 – 30:57
yes
请讲
31:04 – 31:06
it would be an append table
它会有一个Append表

DELTA STORAGE

Main Table

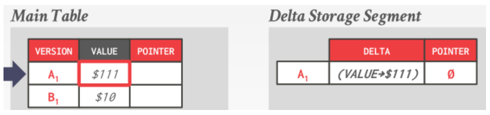
VERSION	VALUE	POINTER
A_1	$111	
B_1	$10	

Delta Storage Segment

On every update, copy only the values that were modified to the delta storage and overwrite the master version.
31:12 – 31:13
all right

31.13–31.20
so now we’ll move on to the the last approach that we’re going to consider which is Delta storage
So，现在我们来看下最后一种方案，即Delta Storage
31:21 – 31:24
which again this is used by both MySQL and Oracle
MySQL和Oracle都使用了这种方案
31.24–31.26
, and like I mentioned
正如我所提到的
31.26–31.29
it’s the one Andy thinks is the best option
Andy觉得这才是最佳方案
31:30 – 31:31
so what’s gonna happen here is
So，这里所发生的事情是
31.31–31.32
every time you do an update
每当你要更新数据时
31.32–31.39
you’re just going to copy the values that were modified into this separate Delta storage segment that you see over here

你只需将那些修改过的值复制到这张单独的Delta Storage Segment中即可



31:40 – 31:43

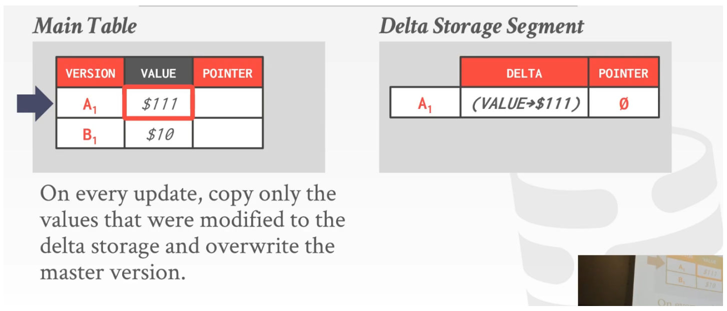
so to update a ,

So，为了更新A

31.43–31.46

we’re first going to update its version value

我们首先要去更新它的版本号



On every update, copy only the values that were modified to the delta storage and overwrite the master version.

31:49 – 31:52

into the Delta storage right, so we’re gonna copy over its value

So，我们要将这个值复制到右边的Delta Storage Segment表上

31:52 – 31.54

so instead of storing the entire tuple

So，我们无须去保存完整的tuple

31.54–31.03

~~we’re just going to call~~ we’re just going to create a delta the States, you know which part which attributes in the tuple were actually modified

我们只需去创建一个delta值，这里面存放的是该tuple中实际被修改的属性值

32:03 – 32:04

so in this case

So，在这个例子中

32.04–32.05

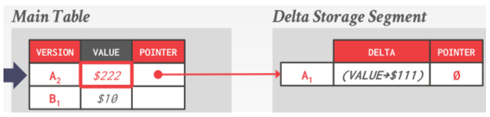
there’s one attribute

我们只有一个属性

32.05–32.10

,so that that was now reflected in the Delta storage segment

So，这一点已经在Delta Storage segment中反映出来了



32:11 – 32:15

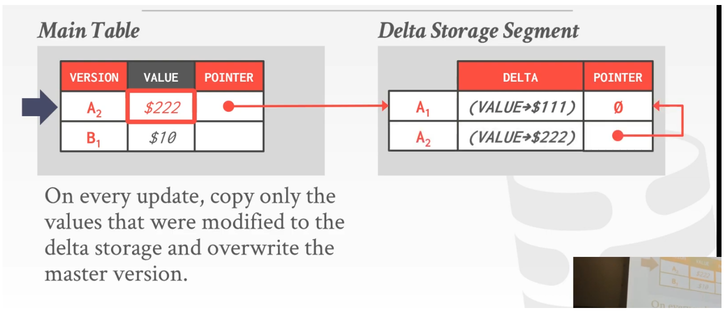
then we’re going to update the actual value in the main table

接着，我们会在Main表中更新实际的值

32.15–32.21

,and also update the pointer from the new value into our Delta storage

并且，我们要去更新指针，让它从这个新值指向我们的Delta Storage对应的位置



On every update, copy only the values that were modified to the delta storage and overwrite the master version.

32:22 – 32:23

so similarly

So，类似地

32.23–32.30

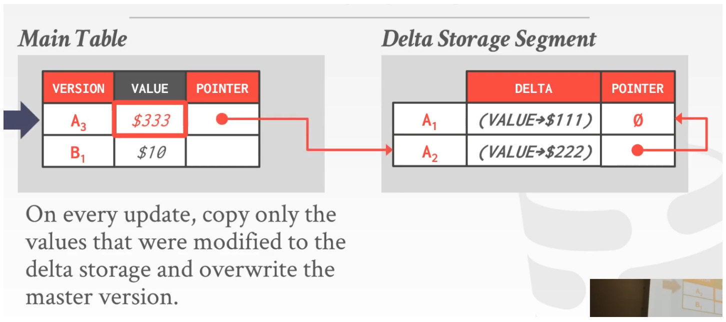
if we want to now install a new value ,And new version

如果我们现在想插入一个新值和一个新版本号

32:30 – 32:34

then we need to do something similar to the time travel table scenario

那么，我们需要做些和Time-Travel Table那种情况类似的事情



On every update, copy only the values that were modified to the delta storage and overwrite the master version.

32.34–32.38

, which is specifically we append the new version

我们会去追加新的版本号

32.38–32.39

, update the value, again

并更新值

32.39–32.42

and now we're seeing its version A3, right

现在，我们看到它的值就变成了A3

32:42 – 32:50

~~but we have~~ but we also need to update the pointer from A3 to now point to the most current value of A2

但我们也需要更新A3的指针，让它指向A2的当前值

32:50 – 32:51

and additionally

此外

32.51–32.56

we need to update the pointer of A2 to the point to the older version now A1

我们需要去更新A2的指针，让它指向A的老版本，即A1

32.56–33.03

This is the time travel example

这就是关于Time-Travel的例子

33:03 – 33:08

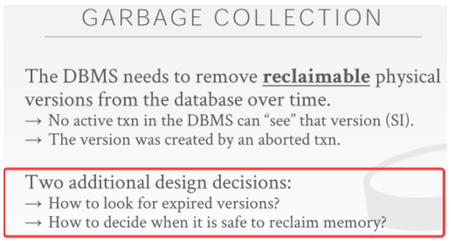
so when you want to read an old version
So，当你想读取某个数据的老版本时
33.08-33.11
, well you really what you essentially have to do is
Well，你本质上必须要做的事情是
33.11-33.17
you have to replay the deltas to put the tuple back into its original form
你必须通过这些delta值来将tuple恢复原状
33:17 - 33:19
so in this case
So，在这个例子中
33.19-33.22
if we wanted to read A1
如果我们想读取A1
33.22-33.25
we would start with the value of A3
我们就需要从A3值开始处理
33:25 - 33:28
and then we would follow the pointer that A2 apply the delta on A2
接着，我们会沿着指针找到A2，然后使用delta值来对A2进行处理
33:28 - 33:32
and then apply the Delt A1
接着，使用delta值来处理A1
33.32-33.34
 and that would get us back to the original value
这就会让我们知道该值一开始是什么
33:38 - 33:43
so this is another good example of the trade-offs between reads and writes
So，这是另一个关于在读操作和写操作间进行取舍的例子
33:43 - 33:47
so reading old versions and the append-only approach is really easy
So，从读取旧版本数据来看，这种append-only方案是非常简单的
33.47-33.49
which is one nice thing about it
这是该方案中一个很nice的地方
33.49-33.50
 it's easy to implement right
并且实现起来也很容易
33:52 - 33:57
because you just find the version and the tuple is already ready to be turn
因为你只需找到该版本以及tuple就行了
33:57 - 33.59
so in addition to being easy to implement
So，除了易于实现以外
33.59-34.02
 ,you also don't have to put the tuple back together
你也不需要将tuple变回原样
34.02-34.06
,you don't have to apply Delta's to get it back to its correct state

你也无须通过delta值来让它变回正确的状态
34:06 – 34:10
but with Delta storage， writes are going to be much faster
但如果使用delta storage，那么写操作的速度就会更快
34:10 – 34:16
because we don't have to copy the entire tuple, if we only make a change to a subset of the attributes
因为如果我们只对该tuple的一小部分属性进行修改，我们也就不需要复制整个tuple
34:16 – 34:19
so you know if you just have one attribute like we do here
So，如果你只有一个属性，就像我们这里例子中所展示的那样
34.19–34.22
, this is you know clearly a trivial optimization
这显然是一个微不足道的优化
34:22 – 34:23
but in many tables
但在很多表中
34.23–34.27
you might have you know dozens of columns
你可能会有数十个列
34.27–34.35
in which case this can matter a lot
在这种情况下，这种优化就变得很重要了
34:35 – 34:40
but again yes with the Delta storage the that is the benefit
但这就是Delta storage所带来的好处
34:40 – 34:43
but the disadvantage is that
但它的缺点在于
34.43–34.49
you have to replay the deltas again to put the tubule back together into its correct value
你必须重演这些delta值，以此来将tuple变回原来正确的值
34:49 –34:51
so one takeaway you can go from this is
So，你们从中能学到的一点是
34.51–34.54
like we mentioned earlier Postgres
正如我们之前提到过的PostgreSQL
34:56 – 34:58
Postgres will be faster for reads right
PostgreSQL在处理读操作方面会更快
34.58–35.04
because
因为。。。
35:05 – 35:07
Well PostgreSQL will be faster for reads
Well，PostgreSQL在处理读操作方面更快
35.07–35.12
and the and MySQL will be faster for writes for this exact reason

出于这个理由，MySQL处理写操作的速度会更快

35:15 – 35:15

alright



GARBAGE COLLECTION

The DBMS needs to remove **reclaimable** physical versions from the database over time.
→ No active txn in the DBMS can "see" that version (SI).
→ The version was created by an aborted txn.

Two additional design decisions:
→ How to look for expired versions?
→ How to decide when it is safe to reclaim memory?

35:18 – 35:22

so the third thing that we need to know about on our list is garbage collection

So，在我们的列表上我们需要知道的第三个东西就是垃圾回收

35:23 – 35:28

so all of these old versions are accumulating as transactions are running and finishing

当这些事务在执行和结束的时候，所有这些老版本数据都会累积在一起

35.28–35.29

and at some point

在某个时候

35:29 – 35:36

we know that the particular version is not being is not visible to any other active transactions ,right

我们知道某个特定版本数据不会对其他任何活跃的事务可见

35:36 – 35:37

so what this means is

So，这意味着

35.37–35.44

if you're thinking about the table with the begin and end timestamps and the timestamp version

如果你思考下表中的begin timestamp和end timestamp以及时间戳版本

35:44 – 35:45

it means that

这意味着

35.45–35.53

there are no active transactions with a timestamps that fit between that begin and end range right from older versions

就没有活跃事务的时间戳在这些老版本数据的begin timestamp和end timestamp的范围之内

35:53 – 35:54

so at this point

So，此时

35.54–35.58

we want to go ahead and garbage collect these versions in order to reclaim space

我们想去回收这些数据，以此来释放这些空间

36:01 – 36:04

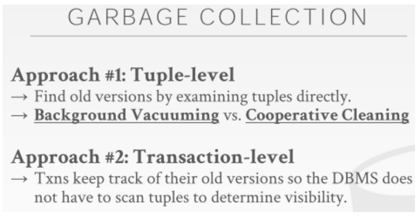
so two additional things that we have to worry about are

So，这里我们还需要关心两件事

36.04–36.06

how we're going to look for expired versions

即我们该如何查找那些过期的版本数据

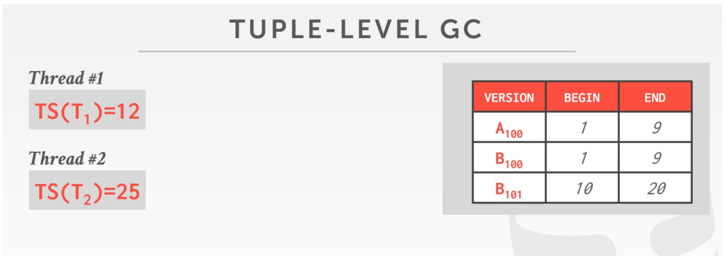
36.06-36.10
, and when it's safe to reclaim them
以及何时回收它们才是安全的
36:10 – 36:13
so these are topics that we're not going to cover in this class
So，我们不会在这门课上介绍这些内容
36.13-36.19
, but they are covered in the advanced class if you do choose to take it
但如果你选择上15-721的话，你们就会学到这些内容
36:19 – 36:21
so there's two approaches that
So，这里有两种方案

GARBAGE COLLECTION

Approach #1: Tuple-level
→ Find old versions by examining tuples directly.
→ **Background Vacuuming** vs. **Cooperative Cleaning**

Approach #2: Transaction-level
→ Txns keep track of their old versions so the DBMS does not have to scan tuples to determine visibility.
36:25 – 36:32
so these two approaches that we're going to look at specifically the first one is **tuple-level** garbage collection
So，我们要看的第一种垃圾回收是tuple级别的垃圾回收
36:32 – 36:33
and the second one is **transaction-level**
第二种则是事务级别的垃圾回收
36:34 – 36:36
so **tuple-level** means
So，tuple级别意味着
36.36-36.39
 that we're essentially going to do sequential scan on our tables
本质上来讲，我们会对我们的表进行循序扫描
36.39-36.46
and use the version timestamps and set of active transactions to figure out whether the version is expired
通过使用版本时间戳和那些活跃的事务来弄清楚这些版本是否过期
36:47 – 36:48
and if it is
如果它过期了
36.48-36.49
then we go ahead and prune it
那么我们就会将它清除
36:49 – 36:52
~~so the reason why~~ this is actually complicated
实际上，这很复杂
36.52-36.56
because we not only do we have to actually look at the pages in memory
因为我们不仅要去查看内存中的那些page数据
36:56 – 37:00
but we also need to look at the pages that we've swapped out to disk
我们还要去查看那些交换到磁盘上的那些page

37:01 – 37:04
because again we want to vacuum everything right
因为我们想去清除所有垃圾
37:04 – 37:10
so we’ll go for a background vacuuming and cooperative cleaning in the next slide
So，我们会在下一张幻灯片中讨论background vacuuming和cooperative cleaning
37:11 – 37:14
so the second approach is transaction–level
So，第二种方案是事务级别的垃圾回收
37.14–37.17
which we’re really not going to go into much detail about
我们不会对它太过深入

37:18 – 37:19
but the general idea is that
但它的基本思路是
37.19–37.22
you have transactions that maintain their read write sets
你的事务会去维护它们的read set和write set
37:23 – 37:24
and you know when they commit
当这些事务提交的时候
~~37.24–~~
~~;so the versions are right~~

37:27 – 37:29
so in this case
So，在这个例子中
37.29–37.30
 you have the transactions
你有这些事务
37.30–37.34
again they’re maintaining read write set
它们会去维护它们的read set和write set
37:34 – 37:36
so you know exactly when they commit
So，当这些事务提交的时候
37.36–37.40
and thus you can figure out when they’re no longer visible and can vacuum them
你可以弄清楚，当它们不再可见的时候，你就可以清理它们了


TUPLE-LEVEL GC

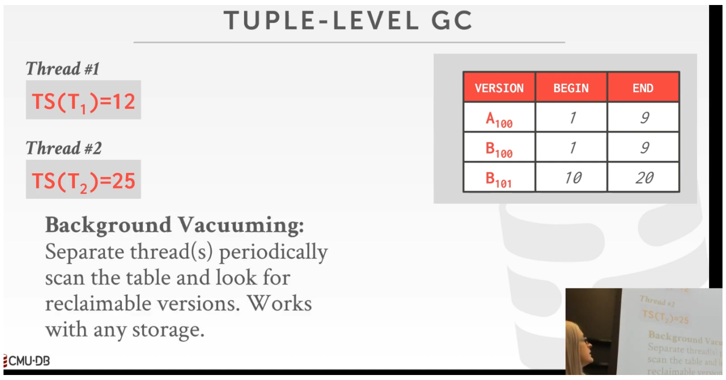
Thread #1
TS(T_1)=12

Thread #2
TS(T_2)=25

VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

37:42 – 37:47
so the first will first go over an example of how a tuple level garbage collection works
So，我们先要看的例子是关于tuple级别的垃圾回收是如何工作的

37:48 – 37:53
so let's say we have two threads running in the system where
So，假设我们系统中有两条线程正在执行

37:54 – 37.58
so transaction t1 is assigned the timestamp of 12
So，T1的时间戳是12
37.58–38.03
,and transaction t2 is assigned timestamp of 25
T2的时间戳是25
38:03 – 38:04
and then over in our version table
在我们的version表中
38.04–38.07
 you can see we have object a
你可以看到，我们有一个对象A
38.07–38.09
 ,which is assigned version 100
它的版本号是100
38.09–38.12
has a begin timestamp of 1 and an end timestamp of 9
它的begin timestamp是1，end timestamp是9

TUPLE-LEVEL GC

Thread #1
TS(T_1)=12

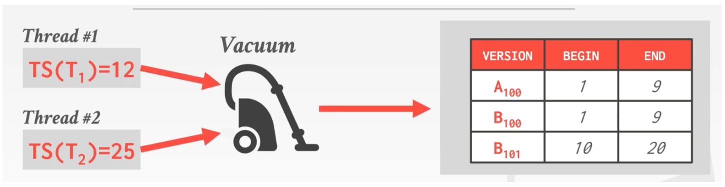
Thread #2
TS(T_2)=25

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

38:13 – 38:17
and then we have a few other versions in there for object B
接着，我们所拥有的对象B还有一些其他版本
38:19 – 38:22
so with background vacuuming
So，通过使用background vacuuming
38.22–38.23
 what we're going to do is
我们要做的事情是
38.23–38.27
we have sort of a set of threads that run in the background
我们在后台有一些线程在运行
38:27 – 38:30
and they perform this vacuuming
它们会执行这种清理
38.30–38.33
 where they periodically just do a full table scan of the table

它们会定期对表进行全表扫描
38:35 – 38:39
and look for which versions are cleanable
并查看哪些版本是可以被清理的
38:40 – 38:43
and it works with any type of storage here
它可以和任意类型的Storage一起使用
38:44 – 38:47
~~So for background~~ so here
So，此处
38:49 – 38:50
see have background thread
这里会有一些后台线程
38.50–38.57
 there goes to the transaction thread in and says what
它们会去查看这些事务，并表示
38:57 – 39:01
and it basically queries what the current transaction timestamps are
简单来讲，它会去查询当前事务的时间戳
39:01 – 39:02
so in this case
So，在这个例子中



VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

39.02–39.04
 it’s gonna be 12 and 25, right
T1的时间戳是12，T2的时间戳是25
39:06 – 39:14
then it’s going to do a sequential scan on the table to figure out whether the tuples would ever be visible to them
它会对这张表进行循序扫描，以此来弄清楚这些tuple是否对它们可见

39:14 – 39:15
All right

39.15–39.18
so A100 be visible to them
So，A100对它们可见
39:22 – 39:22
hard to say
这里有点难讲
39.22–39.26
because we don’t know yet ,what they’re reading or writing
因为我们还不知道它们所读取或写入的是什么东西
~~39.23–39.26~~

~~but~~

39:32 – 39:34

oh I see okay sorry

Oh，我懂了

39.34–39.34

let me back up

让我恢复一下

39:35 – 39:37

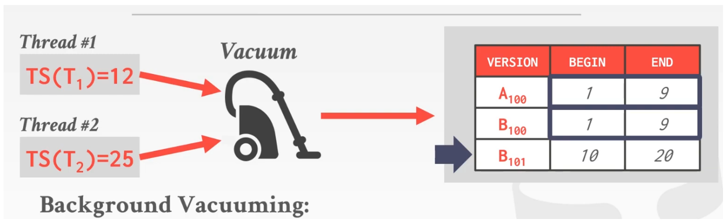
so again sorry in this example

So，在这个例子中

39.37–39.41

we’re just looking at at the beginning end timestamps here

我们会去查看这里的begin timestamp和end timestamp



VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

39:41 – 39:49

so here we gather the timestamps of 12 and 25 from these two transactions

So，我们从这两个事务中拿到了两个时间戳，即12和25

39:49 – 39:52

and then we again look at the beginning and end timestamp

接着，我们会去查看begin timestamp和end timestamp

39:52 – 39:56

so they will never be able to use A100 or B100

So，它们永远不能够使用A100或者B100

39.56–39.58

because the timestamp does not fall between 1 and 9

因为这两个时间戳不属于1到9这个范围内

39:59 – 40:05

~~whereas they do follow~~ well the timestamp of transaction t1 falls between 10 and 20

Well，T1的时间戳是在10和20之间