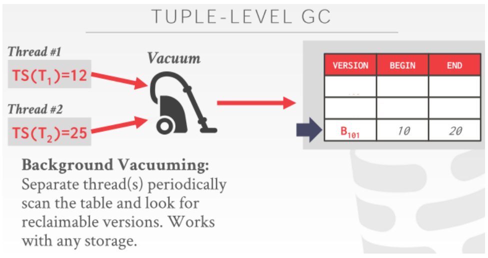
19–03

40:05 – 40:08

so I could potentially use that value

So，我可能可以使用这个值



TUPLE-LEVEL GC

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

40:11 – 40:12

and at this point

此时

40.12–40.15

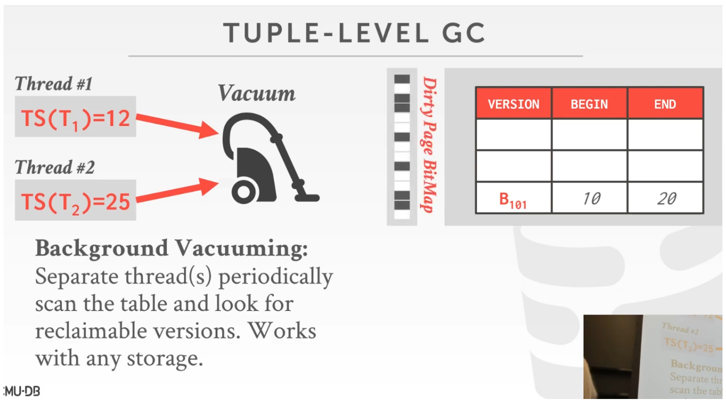
we know those two tuples are safe to reclaim

我们知道我们可以安全回收这两个tuple

40:15 – 40:16

so we go ahead and do,So um

So，我们继续



TUPLE-LEVEL GC

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

40:19 – 40:20

so one optimization here

So，这里我们可以对其进行一种优化

40:28 – 40:31

well one obvious optimization here that we can do is

Well，我们这里可以使用的一种优化方式是

40.31–40.36

we can actually maintain a bitmap for dirty pages

实际上，我们可以为那些dirty page维护一个bitmap

40:36 – 40:38

and so anytime you modify it

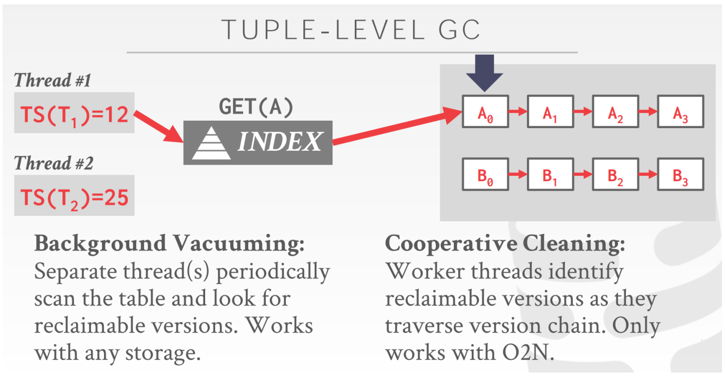
So，每当你更新数据时

40.38–40.41

 ,you can just flip the bit of the page that you modified
你可以翻转你所修改的那个page对应的bit
40:41 – 40:47
so again we're maintaining a bitmap for all of the pages in the database pages specifically
So，我们会为数据库中的所有page去维护一个bitmap
40:48 – 40:49
if we modify a page
如果我们修改了某个page
40.49–40.53
 we'll flip that particular bit which indicates that that page is dirty
我们就会翻转它所对应的bit，以此来表示这个page变dirty了
40:54 – 40:58
so this you know takes a little bit extra storage
So，你知道的，这需要占些额外存储空间
40:58 – 41:01
But it's just a single bit for all of the pages in the database
但对于数据库中的所有page来说，它们共用一个bitmap
41:02 – 41:06
~~And anytime you want it in~~ ,so when the vacuumer comes around
So，当清理器开始清理的时候
41:06 – 41:09
it immediately knows which pages it actually needs to vacuum right
它就能立刻知道哪些page需要被清理
41:10 – 41:12
so it will go ahead and vacuum that page
So，它就会去清理这个page
41.12–41.13
and then reset the bit to zero
并将该page对应的bit重置为0
41:17 – 41:25
so vacuuming again is typically ran as sort of a cron job that runs periodically
So，垃圾清理通常是一个定时任务
41:25 – 41:28
but in some database systems for example Postgres
但在某些数据库系统中，以PostgreSQL为例
41:29 – 41:36
you can actually invoke vacuum manually from the SQL prompt for example
实际上，你可以在SQL prompt中手动调用vaccum进行垃圾清理
41:36 – 41:41
And it also has configuration parameters that you can set
你也可以对它里面的一些配置参数进行设置
41.41–41.51
 such that ,it will the system will basically start up a vacuum thread，if over you know 20% of the pages are dirty for example
简单来讲，比如说，如果超过20%的page是dirty的，系统就会启动一个vaccum线程对这些page进行清理

41:52 – 41:54

so there's different way to implement this
So，我们可以通过不同的方式来实现这点
41.54–41.58
there's different ways to optimize it for different workloads
我们可以针对不同的workload使用不同的方式进行优化
42:04 – 42:05
okay



TUPLE-LEVEL GC

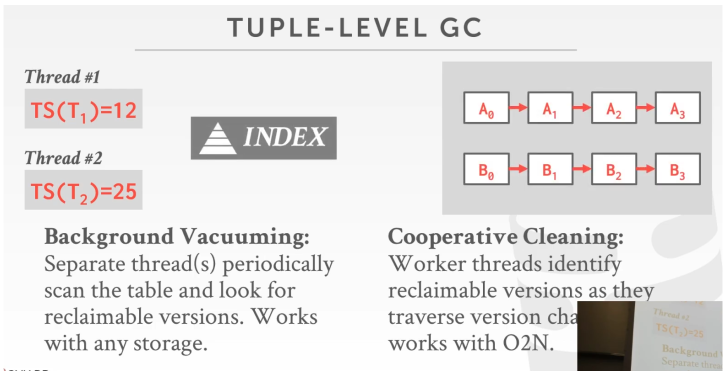
Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

42.05–42.13
so the other approach we're going to look at is cooperative cleaning alright
So，我们要查看的另一种方案就是Cooperative Cleaning

42:14 – 42:18
so this is basically where the threads as they're executing queries
So，简单来讲，当这些线程在执行查询的时候
42.18–42.20
when they come across old versions
当它们遇到旧版本数据的时候
42:21 – 42:23
that they know are not visible to anybody else
它们知道，对于其他人来说，这些数据是不会再用的
42.23–42.27
 it's their job to actually clean them up as they go along
它们的工作就是，当它们遇上这些旧版本数据的时候，就把这些数据清理掉
42:28 – 42:32
so again these are threads are actually executing transactions
So，实际上，当这些线程正在执行事务的时候
42:33 – 42:44
they're going to actually check the versions ,that they that they Traverse across whether that space is ready to be reclaimed
当它们在遍历的时候，它们会对这些数据的版本进行检查，并看看能否去回收这些空间
42:45 – 42:48
because they're not visible to any transactions anymore
因为这些数据不再对任何事务可用
42.48–42.48
and if it is
如果确实如此的话
42.48–42.50
 they will go ahead and reclaim that space
这些线程就会回收这些空间

42:51 – 42.54
so one thing to note is
So，这里要注意的一件事情是
42.54–43.01
 if you consider the two orderings that we discussed earlier ,oldest to newest ,and newest to oldest
如果你思考下我们之前讨论过的两种顺序，即从旧到新，从新到旧
43.01–43.05
would would this approach work for both of those
这种方案是否适用于这两种顺序呢?
43:06 – 43:07
no
答案是No
43.07–43.08
 right why is that
原因是什么呢?
43:16 – 43:18
that’s it that’s exactly right yeah
说的没错
43:18 – 43:21
so in the case of newest to oldest
So，如果是按照从新到旧的顺序
43.21–43.24
 ,you’re not going to be looking at any of the old transactions
你就没办法去查看那些旧的事务了(知秋注:直接找到目标值了，干嘛还要无用功遍历其他数据)
43:24 – 43:27
so you will actually never end up reclaiming those
So，实际上，你也就永远没法去回收这些空间了
43:28 – 43:33
so it’s important to note that a cooperative cleaning only works with oldest to newest ordering
So，你们要记住Cooperative Cleaning只适用于从旧到新这种顺序，这点很重要
43:34 – 43:35
all right



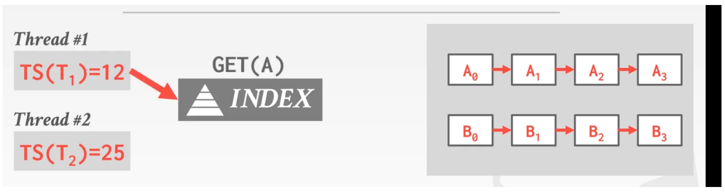
TUPLE-LEVEL GC

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

Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version cha
works with O2N.

43.35–43.38
so now we’ll just go through a similar example here
So，我们来看个类似的例子
43:38 – 43:39
so let’s say that

So，假设



43.39-43.46

 the wave in index in transaction t1 wants to do a look-up on object A now

T1想通过索引来找到对象A

43:48 - 43:52

so again it's going to land on the head of the version chain

So，它会落在version chain的头节点处

43.52-43.54

, which is the oldest value

即A的最旧值处

43:55 - 44:00

and then it's going to scan along until it figures out which versions are actually visible to it

接着，它会沿着这个对象的version chain进行扫描，来弄清楚实际该数据的哪个版本对其可见

44:01 - 44:12

~~so if it recognizes a version that it's looking~~ if it recognizes that one of the versions that it's currently traversing it's not visible to any other transactions

如果T1意识到它正在遍历的某个版本对于其他事务来说，都是无视的

44:13 - 44:16

then we'll go ahead and mark them as deleted and reclaim the space

那么，我们就会将这些标记为deleted，并回收它们所占用的空间

44:17 - 44:25

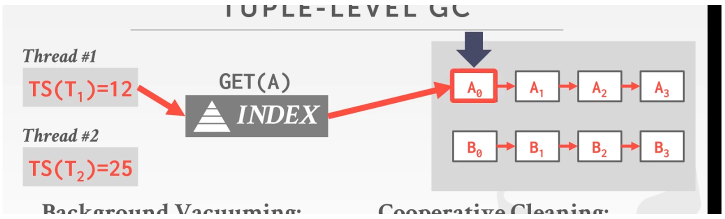
and then at the very end of us also update the index to point to the new head of the version chain

在最后，我们会更新索引，让它指向这个version chain的新头节点

44:25 - 44:28

so we'll just go through these steps

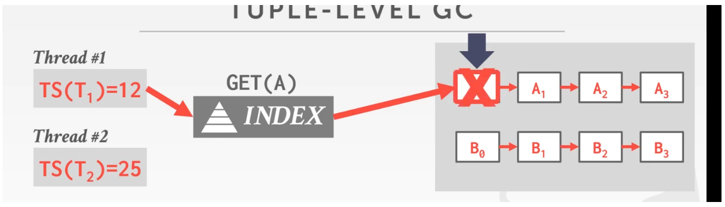
So，我们会来看下这些步骤



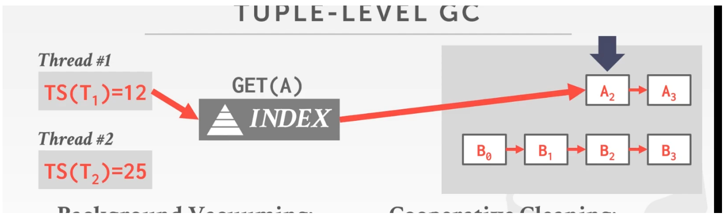
44.28-44.31

so here we find the value

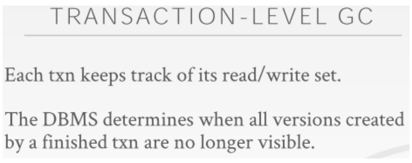
So，我们在此处找到了值



44:33 – 44:39
right and so we can see the version A1 is can be reclaimed
So，我们可以看到A1是可以被回收的



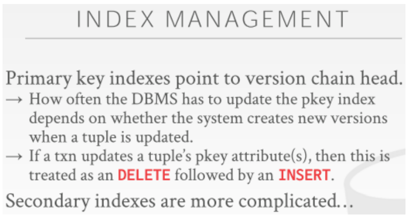
44:41 – 44:48
~~and then we reattach~~ and then we recreate the pointer from the index to the new version head right
接着，我们重新创建一个指针，让它从索引指向这些version chain的新头节点
44:49 – 44:52
so ordering is is actually important here
So，实际上，顺序在这里很重要
44.52–44.57
,and the ordering that’s actually done on the that’s actually on this slide is not quite correct
实际上，幻灯片上所展示的这些顺序并不完全正确
44:57 – 45:02
so when you actually perform these operations
So，当你实际执行这些操作的时候
45.02–45.10
~~what you would do first is actually update,~~ so the first thing you would do is mark them as deleted right
So，首先你要做的就是将这些对其他事务不可见的版本数据打上deleted标记
45:10 – 45:12
but you’re not actually reclaiming the space yet
但你实际还不会去回收这些空间
45:12 – 45:14
the next important thing is that
接下来一件重要的事情是
45.14–45.21
 you actually update the index pointer to point to A2 before physically deleting them, right or claiming that space
在物理删除这些数据或者回收这些空间之前，实际上你可以更新索引指针，让它指向A2，
45:21 – 45:25
because otherwise if you have other transactions running concurrently
否则，如果你同时有其他的事务正在执行
45.25–45.29
, they might find an empty pointer that points to nothing
它们可能会看到一个什么也不指向的空指针
45.29–45.30
yes
请讲
45:43 – 45:44
Yes yeah essentially yeah
说的没错

45:45 – 45:46
so it’s going to maintain some information
So，它会去维护一些信息
45.46–45.47
so they can figure out
So，它们可以去弄清楚。。。
45:47 – 45:56
so it’s going to know the set of active transactions, and be able to compare those timestamps with the begin and end timestamps that in the version table
~~So，通过这些处于活跃状态的事务，我们能够去比较version表中那些begin timestamp和end timestamp来比较时间戳~~
通过这些活跃的事务，我们可以拿到它们的timestamps，来和version表中的beign timestamp和end timestamp进行比较
45.55–45.56
correct
你说的没错

TRANSACTION-LEVEL GC

Each txn keeps track of its read/write set.

The DBMS determines when all versions created by a finished txn are no longer visible.

46:01 – 46:02
all right

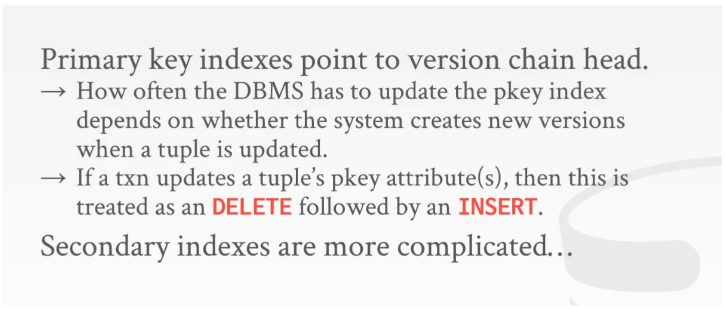
46.02–46.07
so again **transaction-level GC**
So，我们来讲下事务级别的垃圾回收
46:06 – 46:10
we here we just maintain the read/write sets of transactions
我们只需去维护事务的read/write set
46:10 – 46:13
and we use them to figure out what versions are not visible anymore
我们通过它们来弄清楚哪些版本数据不再可用
46.16–46.16
and then we claim the space
然后，我们去回收这些空间
46.16–46.22
and that’s really all we’re going to say about about transaction-level GC
关于事务级别的垃圾回收，这就是我们要讲的东西
46:22 – 46:26
so any questions on GC or anything else up until this point
So，对于垃圾回收方面或者到目前为止我们所讲的内容，你们有任何问题吗

INDEX MANAGEMENT

Primary key indexes point to version chain head.
→ How often the DBMS has to update the pkey index depends on whether the system creates new versions when a tuple is updated.
→ If a txn updates a tuple's pkey attribute(s), then this is treated as an **DELETE** followed by an **INSERT**.

Secondary indexes are more complicated…

46:28 – 46:29
all right

46.29–46.38
so now we're going to move on to our final topic in design decision which is index management
So，现在我们要讲的就是最后一个主题，即索引管理
46:39 – 46:41
so as I mentioned before
So，我之前提到过
46.41–46.45
the primary key index is always going to point to the head of the version chain
主键索引指向的永远是version chain的头节点
46:46 – 46:49
anytime we create a new version
每当我们去创建某个新版本数据时
46.49–46.51
we have to update the version chain
我们需要去更新该数据对应的version chain
46:51 – 46:57
~~or we have to update~~ well we have to update the index to point to the new heads at the version chain right
Well，我们需要对索引进行更新，以让它指向该version chain的新头节点
46:57 – 47:00
so this gets tricky when updating the primary key
So，当更新主键时，这会变得有点棘手
47:00 – 47:07
because now it's actually possible that you could have two version chains for the same logical tuple
因为对于同一个逻辑tuple来说，实际上，我们是有可能会出现两个version(新的和之前的旧的)
47:08 – 47:10
the way you implement this is
你实现这个的方式是
47.10–47.16
~~,when if you want to delete the primary~~ ,when you want to update the primary key
当你想去更新主键时


Primary key indexes point to version chain head.
→ How often the DBMS has to update the pkey index depends on whether the system creates new versions when a tuple is updated.
→ If a txn updates a tuple's pkey attribute(s), then this is treated as an DELETE followed by an INSERT.
Secondary indexes are more complicated…

47:16 – 47:21
you did this as a delete followed by an insert of a new logical tuple
你先执行delete，紧接着再插入一个新的逻辑tuple
47:21 – 47:24
and there's some bookkeeping you need to maintain and
你需要维护一些bookkeeping信息
47:26 – 47:30
and you also need to understand like how and when to rollback when necessary

你也需要理解，当你需要的进行回滚的时候，该怎么去做

47:31 – 47:33

but for **secondary indexes**

但对于Secondary index来说

47.33–47.35

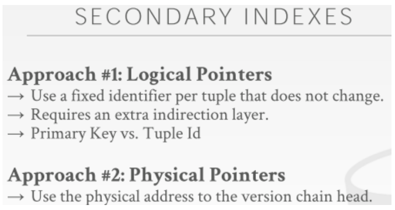
this is actually more complicated

这实际上更为复杂

47.35–47.40

 and this will be what we’ll talk a little bit more about

我们会稍微讲下关于这方面的内容



SECONDARY INDEXES

Approach #1: Logical Pointers
→ Use a fixed identifier per tuple that does not change.
→ Requires an extra indirection layer.
→ Primary Key vs. Tuple Id

Approach #2: Physical Pointers
→ Use the physical address to the version chain head.

47:40 – 47:47

so with **secondary indexes** the two approaches

So，对于Secondary index，我们有两种方案

47:48 – 47:58

we use to make sure that our indexes reflect the the correct value in the version chain are to maintain a logical pointer

我们通过维护一个逻辑指针来确保我们的索引反映了version chain中的正确值

47:59 – 48:08

and here so here you have ~~some kind of false identifier for the tuple or~~ some kind of unique identifier for the tuple, that does not change

So，每个tuple都会有一个唯一标识符，它是不会改变的

48:08– 48:11

and then you have some layer of indirection

接着，你会有一个indirection layer（间接层）

48.11–48.16

 an indirection layer that map’s the logical ID to the physical location in the database

间接层所做的事情是将tuple的逻辑id映射到数据库中的物理位置

48.16–48.19

and anytime you update the version chain

每当你要更新version chain的时候

48:19 – 48:25

you just have to update the indirection layer rather than actually updating every single index

你只需要去更新这个间接层即可，而无需去更新每个索引

48:25 – 48:26

all right

48.26–48.35

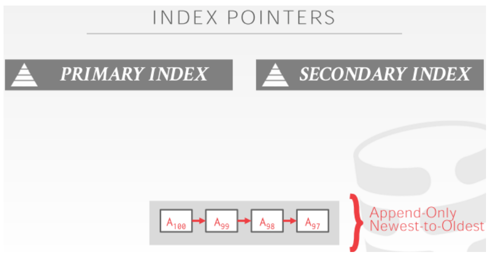
so the actual approach is I think it was I used in some of the slides earlier I think, which is to actually use physical pointers

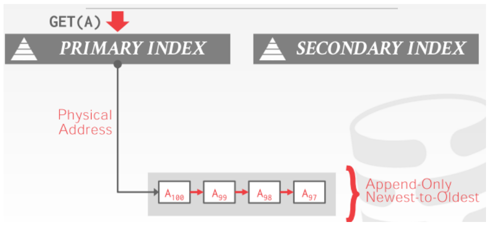
我觉得我在前几张幻灯片中使用过一种方案，即物理指针这种方案

48:36 – 48:39

which is when you just point directly to the head of a new version chain

即它直接指向的是version chain的头节点
48:40 – 48:42
so every time the version chain gets updated
So，每当version chain更新的时候
48.42–48.44
you have to update every single index ,right
你需要更新每个索引
48:45 – 48:53
so the difference between the physical between using physical pointers and logical is you basically have this indirection table
So，简单来讲，物理指针和逻辑指针之间的区别就是逻辑指针中有一个间接表
48:53 – 48.56
and the benefit of the indirection table is that
这种间接表的好处在于
48.56–49.03
you do not have to update every single index， every time you update your version chain
每当你更新你的version chain的时候，你无须去更新每个索引

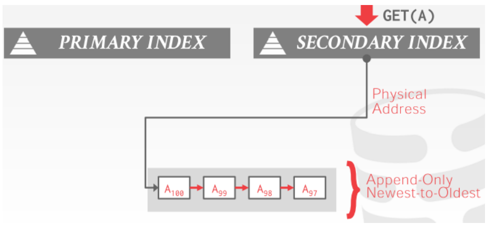


49:04 – 49:09
so all right so in this example
So，在这个例子中
49:09 – 49:11
we'll say we have a simple database
假设，我们有一个简单数据库
49.11–49.17
and we're using a pendant version chain which is running newest to oldest right
这里我们使用version chain将该数据的所有版本串联在一起，从最新到最老


49:21 – 49:27
so for the primary key index if I'm going to do a lookup on object a
So，如果我使用主键索引来查找对象A

49:27 – 49:36
then this will just be a physical address right for the for the primary key which will be just a page ID and offset
对于主键来说，它其实就是一个物理地址，它由page id和offset所组成

49:36 – 49:40

so you know which page to go to and then you take the offset that’s typically what this is
So，通过page id，我们知道这个对象在哪个page上，通过offset，我们知道它在该page的哪个位置上
49:40 – 49:43
it’s going to point again to the head of the version chain right
它会指向版本链的头节点
49:44 – 49:45
and anytime you create a new version
每当你创建某个数据的新版本时
49.45–49.49
you always update that the primary key ,right
你始终要去更新你的主键


49:54 – 49:55
all right

49.55–50.00
so for secondary indexes again you could use the physical address
So，你也可以使用物理地址来作为secondary index（非主键索引）使用
50:00 – 50:02
but there’s the same issue
但这里我们也会遇上相同的问题
50.02–50.10
anytime you update the tuple， you have to update the secondary index to point to this
每当你更新某个tuple时，你必须对非主键索引进行更新，让它也指向这条链的头结点

50:10 – 50:18
and you know this is again like this is similar to some of the two sort of the Delta storage idea that we saw a few slides ago
你知道的，这和我们之前几张幻灯片上看到的Delta Storage思想有点类似
50:19 – 50:21
yes if you have one attribute
如果你有一个属性
50.21–50.24
or if you have one index or one secondary index
或者，你有一个索引或者一个非主键索引
50.24–50.25
then this is not a big deal
那么这不是什么大问题
50:26 – 50:38
but it’s very common for OLTP databases in particular to have many some secondary indexes on a single table
但对于OLTP数据库来说，一张表上有多个非主键索引，这种事情非常常见

50:38 – 50:40

so every time you update the version chain

So，每当你更新version chain的时候

50.40–50.44

you have to update all of those secondary indexes

你就需要更新所有的非主键索引

50.44–50.45

which for OLTP

对于OLTP来说

50.45–50.51

you can imagine might be 12 or you know a few dozen mmm

你可以想象得出一张表上会有12或者数十个非主键索引

50:51 – 50.53

um and this of course is expensive

Of course，这种做法成本很高

50.53–50.55

, because for example

例如

50.55–50.56

if it’s a B+ tree

如果它是一个B+ Tree

50.56–51.01

, then youare traversing the B+ tree you’re taking latches as you go

当你遍历B+ Tree的时候，你就会去获取latch

51:01 – 51:03

and then finally you have to apply the update

接着，最后你需要提交这些更新

51:05 – 51:15

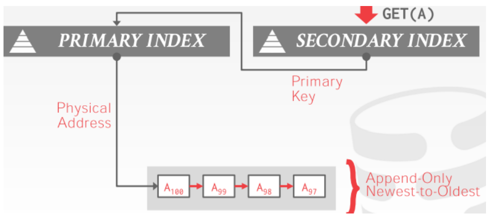
so again like we said the previous slide instead of storing the physical address in the secondary index, we’re going to look at two alternatives

so 就像我们前一张幻灯片所说的，与其在非主键索引中存储的是物理地址，不如存逻辑指针，我们来看第二种方案

51:15 – 51:18

so the first is to just store the primary key

So，第一种方案就是只保存主键



51.18–51.24！！！！！！

which is literally just a copy of the primary key as the value

从字面上来讲，就是将主键副本作为值保存

51:24 – 51:34

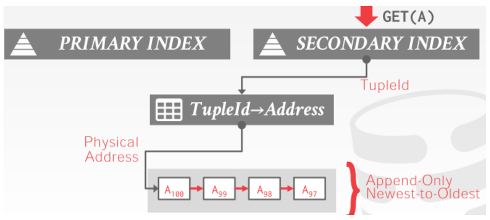
and the secondary index ~~first of all the dress,~~ here we go

对于Secondary index来讲，如图所示

51:34 – 51:42

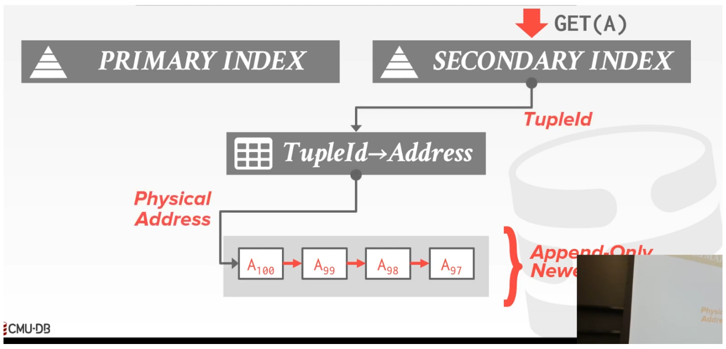
right so here we’re going to have the actual value that we’re going to store in the secondary index is going to be a pointer to the primary key index
~~So，这里我们在非主键索引中保存的值会指向主键索引~~
So，这里我们通过非主键索引来指向主键索引
51:44 – 51:46
so now when you want to find a tuple
So，当你现在想找一个tuple的时候
51.46–51.52
um you just get you first got the primary key index other than the secondary index
首先你需要通过主键索引查找，而不是非主键索引
51:52 – 51:59
and then you do a lookup on the primary key index just as you would to figure out what the physical address is
当你想知道物理地址是什么的时候~~事情~~，你需要在主键索引中进行查找
51:59 – 52:03
and then every proceeds as in the first example with the physical address
所有的操作流程都与第一个例子中的物理地址相关部分相同
52:05 – 52:10
so anytime I update the tuple and the head of the version chain
So，每当我更新tuple和它所对应的version chain的头节点时
52:10 – 52:16
you can just update the primary index and automatically updates all of the secondary indexes right
你可以只更新主键索引，也就自动更新了所有的非主键索引
52:16 – 52:21
so this is one example of logical pointers
So，这就是逻辑指针的一个例子
52:23 – 52:25
and this is what MySQL does
这也是MySQL的做法
52.25–52.29
 and Postgres actually stores the physical address
实际上，PostgreSQL保存的是物理地址
52:32 – 52:32
yes
请讲
52:37 – 52:46
~~um,a secondary index see~~ if your your primary key index right which stores like the key for your immediate table
如果你看下你的主键索引，它保存的其实是你表中的这些key
52:46– 52:53
,so your secondary index is going to I guess be a reference to that case
So，我猜你的非主键索引会引用这些东西
~~52:54 – 52:54~~
~~so~~

~~52:59 – 53:01~~
~~yeah so in this case so like if you~~

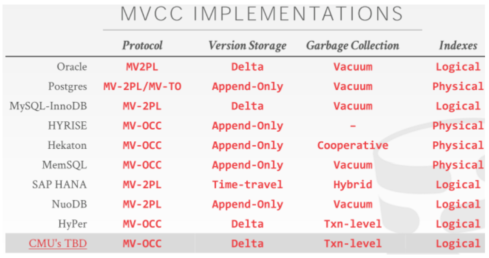
53:02 – 53:07
so if you have a table A right and the ID is your primary key
So，假设你有一张A表，id是你的主键
53:08 – 53:09
then maybe in table B,
接着，兴许，在B表中
53.09–53.15
 you have a reference to table a to to table a.ID,
它里面保存了一个指向A表id这个属性的引用
53.15–53.17
 right a the attribute column in that table, right
即A表中的某个属性
53:18 – 53:23
and so you might create a second this is what's called a secondary index on that particular item
So，你可能会在这个属性上创建一个非主键索引（secondary index）
53:23 – 53:26
so if you want to think about something more concrete
So，如果你想思考一些更具体的情况
53.26–53.28
you have a table of users your user has an ID
假设，你有一张user表，它里面有个id属性
53:29 – 53:33
your user has a list of items that it has purchased
每个用户会有一个列表，该列表上保存了他们所购买的东西
53:34 – 53:43
so for each of those items you might store the user's ID in it or you know something that it's just typically used for tracking
So，你可能会在这些商品中保存购买该商品的用户id或者其他一些用来跟踪的信息
53:43 – 53:46
okay any other questions
Ok，有任何疑问吗?


53:54 – 53:58
hmm think of my beginning of him I said here okay
我想下我这里开头所说的
53:58 – 54:02
so the last approach which is also another example of using a logical ID
So，来讲下最后一种方案，它是逻辑id的另一个案例
54:03 – 54:04
It`s basically
简单来讲
54.04–54.08
you just have some synthetic values it's like a tuple ID
你有那种像tuple id那样的合成值

54:08 – 54:15
so this would typically be you know an incrementing counter to serve as the tuple ID
So，这通常会有一个计数器来增加你的tuple id值
54:15 – 54:17
and then you have a hash table
接着，你还有一个hash table
54.17–54.22
 that says how to map from that tuple ID to the address
我们通过它将tuple id映射到对应的地址上
54:22 – 54:24
so basically
So，简单来讲
54.24–54.27
you're going to get the tuple ID out of the secondary index right
你会通过非主键索引来获取tuple id



54:31 – 54:35
~~buying a dress~~ yes, so you're going to get the tuple id of the secondary index
So，你会通过非主键索引来获取tuple id
54.35–54.39
and then you're gonna ,and then you're going to figure out where the physical address is
接着，你通过tuple id来弄清楚它的物理地址是什么
54:39 – 54:46
and then the hash table here will point you to the location of the physical address ,so you can read that value
这里的hash table会告诉你该tuple的物理地址是什么，So，你可以去读取这个值
54:48 – 54:57
and again similar to the approach we looked at where we were just storing the primary the primary key index
这种方案和我们只保存主键索引那个方案很类似
54:58 – 55:01
this is another example of logical pointers
这是逻辑指针的另一个例子
55.01–55.01
 which means that
这意味着
55.01–55.07
 if each time we have a new version or each time we update the version chain
每当我们要更新version chain的时候
55:07 – 55:12
we can actually avoid having to update all of the secondary index right

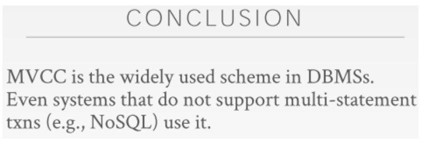
实际上，我们能够避免更新所有的非主键索引
55:12 – 55:18
so we the only thing we have to update in this case is the hash table ,and the pointers
So，在这个例子中，我们唯一要去更新的就是这个hash table和这些指针
55.18–55.20
,that make sense
懂了吗

MVCC IMPLEMENTATIONS

	Protocol	Version Storage	Garbage Collection	Indexes
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-TO	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
HYRISE	MV-OCC	Append-Only	-	Physical
Hekaton	MV-OCC	Append-Only	Cooperative	Physical
MemSQL	MV-OCC	Append-Only	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
NuoDB	MV-2PL	Append-Only	Vacuum	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical
CMU's TBD	MV-OCC	Delta	Txn-level	Logical

55:26 – 55:27
all right

55.27–55.30
so this table is actually really interesting
So，实际上，我们对这张表中的内容非常感兴趣
55:31 – 55:37
um so this is a table from a paper that was published by Andy and a few other students
So，这张表是来自某篇paper，这篇paper是由Andy和一些其他学生一起完成的
55:38 – 55:39
I think a couple years ago
这篇paper发表已经有一两年了
55:40 – 55:41
so what they actually did is
So，实际上，他们在这篇paper中所做的事情是
55.41–55.45
they looked at a number of systems
他们查看了大量的系统
55.45–55.46
so they looked at some older systems
So，他们也看了一些比较老的系统
55.46–55.48
you know like Oracle, Postgres, MySQL
比如：Oracle、PostgreSQL和MySQL
55:49 – 55:53
and they also looked at some much newer systems within the past ten years
他们还查看了一些近十年来出现的一些较新的数据库系统
55:53 – 55:54
so for example
So，例如
55.54–55.56
 like HyPer and Nuodb
HyPer和NuoDB
55.56–55.59
HyPer would be an example of an academic system
Hyper是学术用数据库系统中的一个案例

55:59 – 56:02
so they tried to get a variety of systems here
So，他们试着对一系列数据库系统进行研究
56:02 – 56:09
and and the table lists which of these design decisions each of these database system makes
这张表中列出了他们所研究的每种数据库系统的设计决策
56:10 – 56:16
so let's see if Andy has any exciting things
So，我们看看Andy有没有放什么令我们兴奋的东西
56:16 – 56:23
so I guess he says the spoiler if you guys want the spoiler
So，如果你们想让我剧透下的话
56:25 – 56:38
is the or the take away the spoiler is that Oracle and MySQL the way they do in MVCC they actually found like Andy and some students actually found this way to be the fastest for OLTP workloads specifically
实际上，Andy和其他学生发现Oracle和MySQL这两者在MVCC方面所做的设计对于OLTP workload来说，是最快的

56:40 – 56:42
and actually found Postgres to be the slowest
他们发现PostgreSQL在这方面的速度是最慢的
56.42–56.48
although personally as both the user of MySQL and Postgres ,I like Postgres quite a bit
从我个人角度来说，虽然我是MySQL和PostgreSQL的用户，但我更喜欢PostgreSQL
56:48 – 56:57
but I'm also not running you know commercial database systems with the production workload traces
但我并没有在生产环境下对商用数据库系统进行workload方面的跟踪
56:57 – 56.59
so you know once you get at that scale
So，你知道的，一旦你的规模变得很大
56.59–57.00
it probably matters
这可能就会很重要
57.00–57.02
all well it definitely matters a lot
Well，应该是非常重要

CONCLUSION

MVCC is the widely used scheme in DBMSs.
Even systems that do not support multi-statement txns (e.g., NoSQL) use it.
57:03 – 57:06
so okay so this brings us to the conclusion
So，我们来做下总结
57:07 – 57:09
so today again we talked about MVCC
So，今天我们讨论了MVCC

57.09–57.13

and again as you just saw in the past few slides

就如你们在之前几张幻灯片中看到的那样

57:13 – 57:23

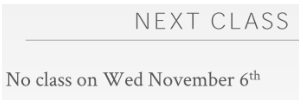
there’s a lot more to this than just figuring out you know what timestamps to assign ,and what versions are visible to the different transactions

除了如何分配时间戳，如何让不同的事务看到对应的版本这些事情以外，我们还需要了解很多东西

57:25 – 57:33

so you know of course you need to figure out how to store the versions how to update them how to update the indexes correctly ,and the other items that we covered here

Of course，你需要弄清楚该如何保存多版本，如何更新它们，如何正确地更新索引，等等。这一系列东西我们这节课上都介绍了



NEXT CLASS

No class on Wed November 6th

57:35 – 57:44

so right for next class just to just as a reminder don’t come to class on Wednesday, because nobody will be here

So，你们要记住，下节课你们不用来教室，因为没人在这里

57:44 – 57:46

so you guys have next ones day off

So，我们下一节课没课

57.46–57.51

and then I think the following week ,Andy will probably be back although that’s not certain

我认为Andy可能下周会回来，虽然我也不确定

57:51 – 57:53

but we will start logging in recovery

但我们会开始讲Logging和Recovery