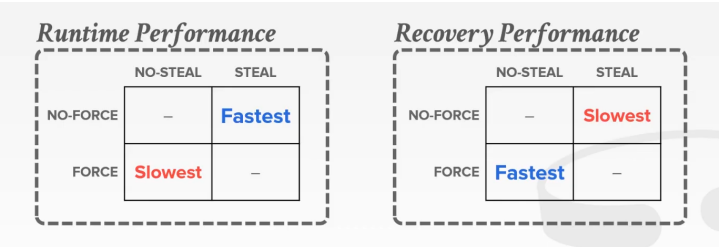
20-04

59:58 – 01:00:00
because I don't do anything extra
因为我并没有做更多事情
1.00.00–1.00.05
I just come back and my DBMS root points to my shadow page or the consistent master
我的database root指向的是我的shadow page或者master
01:00:05 – 01:00:07
the most recently committed version
即那个最近提交的版本
1.00.07–1.00.08
and I'm done
这样我就完事了
01:00:08 – 01:00:10
but under write ahead logging
但如果使用的是预写式日志
1.00.10–1.00.11
that's actually slower
实际上，它的速度更慢
1.00.11–1.00.15
because I have to replay the log up to some some point
因为我需要重新执行日志中的操作，并执行到日志中的某个点为止
01:00:15 – 01:00:18
which she was sort of alluding to with garbage collection
这和垃圾回收有点相关联

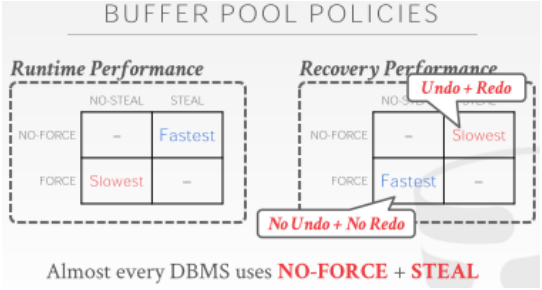


01:00:19 – 01:00:23
so because of this trade–off between performance and recovery time
So，因为性能和恢复时间上的这种权衡关系
01:00:24 – 01:00:29
most database system implementations, choose the the write ahead logging, choose the No–Force+steal
大部分数据库系统选择去实现的是预写式日志，采用的策略是Np–Force+Steal
01:00:29 – 01:00:31
because they'd rather be faster at runtime
因为它们更希望在运行时速度更快

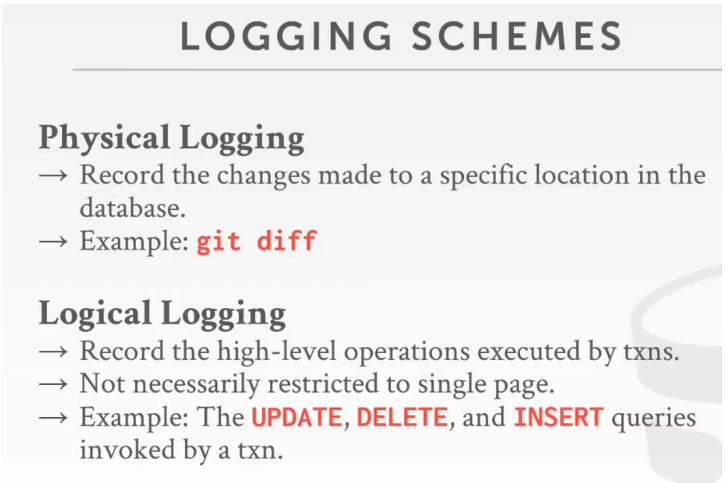
1.00.31–1.0034
and assume failures are gonna be rare
并假设出现故障的频率非常稀少
1.00.34–1.00.37
which you know in the grand scheme of things they are

01:00:37 – 01:00:41
your DBMS not crashing every minute if you do， we have other Problems
你的DBMS不会每分钟都发生崩溃，如果你遇上这种情况，说明我们遇上了其他问题
01:00:41 – 01:00:42
so therefore
因此
1.00.42–1.00.46
they're willing to trade off faster runtime performance in exchange for slower recovery
他们更希望用较快的运行时性能来交换更慢的恢复速度
01:00:48 – 01:00:56
~~there's only one system that I'm aware of~~ except for the ones that do shadow paging like the old SQLite
除了老版的SQLite会使用shadow paging以外
01:00:56 – 01:01:03
there's only one system that I can make the trade–off for having faster recovery time in exchange for running slower at runtime

01:01:04 – 01:01:06
and it was this system I don't know the name of it
我不记得这个系统的名字叫什么
1.01.06–1.01.12
,it was a database system built in the 1970s for the Puerto Rico's electrical system
这个数据库系统是在1970年代为Puerto Rico电力系统所构建的
01:01:12 – 01:01:15
because in Puerto Rico in the 1970s they had power outages like every hour
因为在1970年代，Puerto Rice每小时都会遇上停电的情况
01:01:15 – 01:01:18
so the DBMS was crashing literally every single hour
So，他们的DBMS每小时都会遇上故障
01:01:18 – 01:01:19
so for them
So，对于他们来说
1.01.19–.01.21
it was much better to be slow at runtime
运行时的速度慢点会更好
1.01.21–1.01.24
such that for every hour ,when you crashed ,when you lose power
这样的话，当每小时你断电的时候
01:01:25 – 01:01:27
you could recover the database immediately afterwards
你就可以立即恢复你的数据库



01:01:29 – 01:01:33
right another were there anything like this at the high level to is
另一个高级层面的东西就是
1.01.33–1.01.34
this is with no undo no redo
这里没有Undo，也没有Redo
1.01.34–1.01.35
because there's nothing to reverse
因为这里没有东西要恢复
1.01.35–1.01.37
and nothing to reapply
也没有东西要重新提交
1.01.37–1.01.39
with write ahead logging，you need undo and redo
如果使用的是预写式日志，那你就需要Undo和Redo



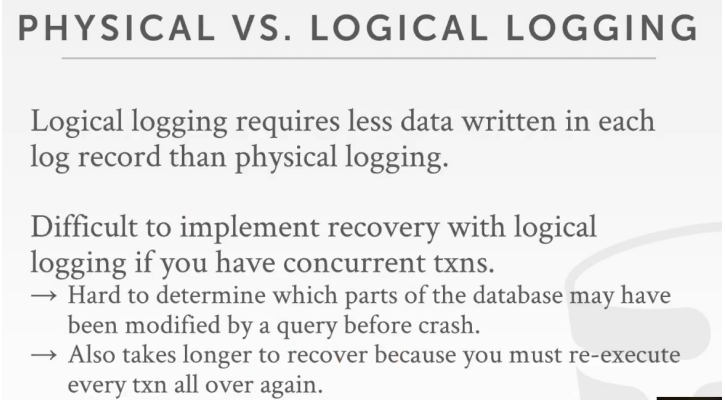
01:01:42 – 01:0143
all right

1.01.43–1.01.47
so at a high level what these log records are
So，从一个高级层面来看，这些日志记录是什么样子呢
1.01.47–1.01.49
that there's an object ID
这里面有一个Object ID
1.01.49–1.01.52
and then there's a before value and the after value
这里面有一个before value，以及一个after value
01:01:53 – 01:01:55
but what how this is actually implemented
但这实际是如何实现的呢

01:01:56 – 01:01:57
so there's a couple different approaches
So，这里有两种不同的方案
01:01:58 – 01:02:01
so the one would be what it's called physical logging
So，其中一种方案叫做Physical Logging
1.02.01–1.02.03
which is basically what I've talked about so far
基本上来讲，这也是我目前所讨论的东西
01:02:04 – 01:02:13
its we're recording the low level byte changes to a specific location as some object in the database that you made to and you know how to reverse it
我们会去记录你对数据库中某个特定位置所做的低级层面的字节修改，并且你知道如何撤销你的修改

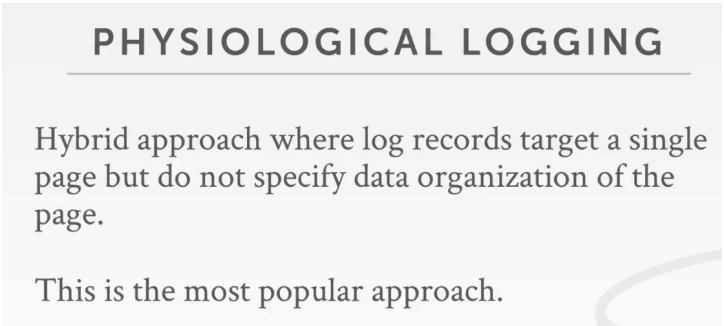
01:02:13 – 01:02:15
so thinking this is like if you run diff or git diff
So，你们可以去思考下git diff
01:02:16 – 01:02:21
it's but allows you to get out the before and after of the change
这使得你可以获取修改前后的数据
01:02:22 – 01:02:23
but the downside of this is again
但它的缺点在于
1.02.23–1.02.27
if I update of you know billion tuples in my transaction
如果我在我的事务中要更新10亿条tuple
01:02:28 – 01:02:32
I have to have a billion log records that corresponds to all those low–level physical changes
那么，我就需要使用10亿条与这些低级层面物理修改所对应的日志记录
01:02:34 – 01:02:36
another approach is do logical logging
另一种方案就是使用Logical Logging
1.02.36–1.02.41
where you just record the high–level operation that which of the change you made to the database
即你只记录那些你对数据库所做的高级层面的修改
01:02:42 – 01:02:46
and that's enough for you to be able to reapply it at run time
这足以让你在运行时重新执行这些操作
01:02:46 – 01:02:48
undo is a lot more tricky
Undo的话，则更为棘手
1.02.48–1.02.52

~~based if you have you know~~ based on what the actual query is
这取决于实际的查询是什么
1.02.52–1.02.52
but let's ignore that for now
但我们现在将它忽略
01:02:54 – 01:03:00
~~so I think the way thing like this is~~ this is storing like in the diff of the change you made
So，这其实就是去保存你对数据修改前后的不同之处
01:03:00 – 01:03:04
this is storing actually the SQL query that of the change you made
实际上，这保存的是你在执行SQL查询时所做的修改



01:03:05 – 01:03:08
so the advantage or disadvantage of each of them are that
So，这两者的优缺点分别是
1.03.08–1.03.12
logical logging allows you to record more changes with less data
Logical Logging允许你使用更少的数据去记录更多的修改
01:03:13 – 01:03:15
I update a billion tuples with a single update statement
我可以使用一条更新语句去更新10亿条tuple
1.03.15–1.03.17
I only know of that one update statement
我只需要知道这一条更新语句即可
01:03:18 – 01:03:22
the downsides gonna be with logical logging though is that
Logical Logging的缺点是
01:03:23 – 01:03:32
it's gonna be tricky for me to figure out what changes I potentially may made to the database that I've got written to disk, before the crash
对于我来说，我难以去弄清楚在系统发生崩溃前，我对数据库所做的修改实际有哪些已经被写入磁盘了
01:03:33 – 01:03:35
Because I don't have that low-level information
因为我并没有这些低级层面的信息
01:03:36 – 01:03:41
I update a billion tuples over a billion pages maybe half of them got written out the disk
比如说：我更新了10亿个page上的10亿个tuple，可能只有一半的tuple被写入到了磁盘
01:03:41 – 01:03:44

how do I know which ones I need to update and replay
我如何知道我该更新哪个page，我该重新执行哪些操作?
01:03:46 – 01:03:47
the other issues gonna be
其他的问题则是
1.03.47–1.03.51
~~you'll see that~~ however long it took me update the database
不管我更新数据库数据时所花的时间有多长
1.03.51–1.03.54
the first time when I ran the query with a logical logging scheme
当我第一次使用Logical Logging方案来执行查询时
01:03:54 – 01:03:56
this is gonna take that same amount of time the second time
它所花的时间和第二次执行该查询时所花的时间相同
01:03:56 – 01:03.58
so my query took an hour to run
So，比如说：我这个查询要花1小时才能执行完毕
1.03.58–1.04.00
during recovery to take an hour to run again
在恢复数据库的时候，执行这个查询还是要花1小时
01:04:01 – 01:04:01
there's no magic
这里并没有什么奇特的地方
1.04.01–1.04.03
because I'm in recovery mode that's gonna make that go faster
当我处于恢复模式时，这会使其运行得更快
01:04:05 – 01:04:08
so although the scheme storing I'm storing less information of logical logging
So，虽然Logical Logging这种方案需要保存的数据较少
1.04.08–1.04.09
it's going to make recovery more expensive
但这会让数据库系统恢复时所付出的成本更加昂贵
1.04.09–1.04.12
and most systems do not make that change
大部分数据库系统不会做出这种改变



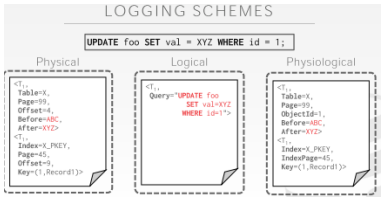
01:04:13 – 01:04:17
the hybrid approach that most people use is call it physiological logging
大部分人所使用的混合方案叫做Physiological Logging
01:04:18 – 01:04:23

where you're not going to store the low level byte information about the changes you're making to the database
即你无须去保存你对数据库所做的那些低级层面
01:04:23 – 01:04:30
you're still sort like it's a low level enough to say here at this page, I'm modifying this object

01:04:30 – 01:04:34
but you're not you're not really taking a diff as you would under physical logging

01:04:34 – 01:04:38
you're just saying here's this logical thing I want you to make up make a change to

01:04:38 – 01:04:40
so this is what most systems actually implement



01:04:41 – 01:04:48
so let's say we have this update query here, so in a physical log it would be like at this page of this offset here's the before and after image

01:04:49 – 01:04:55
and we haven't talked about indexes as well, but indexes you basic at the log in the at the same time you're making changes to the database

01:04:55 – 01:05:01
because if my of my index doesn't fit memory ,then I don't want to have to rebuild it from scratch upon recovery

01:05:01 – 01:05:06
so most DBMS also record the log the changes you makes it indexes

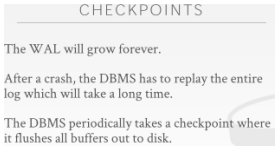
01:05:06 – 01:05:10
logical login of query again you just store the SQL statement

01:05:10 – 01:05:18
physical logical logging is you're saying at this page at this slot number, here's the change I want you to make these these low–level attributes

01:05:20 – 01:05:27
and this allows you to not have to ,but having these in this extra indirection sort of like the slotted pages

01:05:27 – 01:05:38
it doesn't allows you to reorder that the the replay operations, in such a way that the the DBMS mean a byte for byte copy yet before and after the crash

01:05:38 – 01:05:45
you have some wiggle room to actually reapply these in different ways, and restored still restore back to the correct state



01:05:46 – 01:05:48
all right,

1.05.48–1.05.50
so the getting to her question is
So，这就和她的问题有关了
1.05.50–1.05.54
the issue we talked about so far is that
我们目前为止所讨论的问题是
1.05.54–1.05.56
these write ahead logs are gonna grow forever
这些预写式日志的体积会不断变大
01:05:57 – 01:05.59
if my database system is running for a year
如果我的数据库系统运行了一年
1.05.59–1.06.01
I'm gonna have a years of logs
我就会拥有一年份的日志
01:06:01 – 01:06:06
so now if I have to crash ,if I crash and come back into replay this log
So，如果我遇上崩溃的情况，当系统恢复后重新执行该日志上的内容
1.06.06–1.06.08
I potentially have to replay the entire year's worth of data
我可能需要重播这一整年的日志数据
01:06:10 – 01:06:12
so logical logging that would be terrible
So，Logical Logging方案可能就会很可怕
01:06:12 – 01:06:16
right because if the query takes the same amount of time it took the first time as it does during recovery

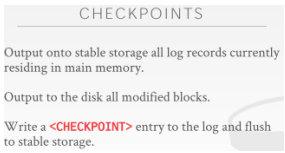
01:06:17 – 01:06:20
so I have a year's worth of data of logical log records

1.06.20–
if I crash and come back

I pretend to take a year for me to recover the database

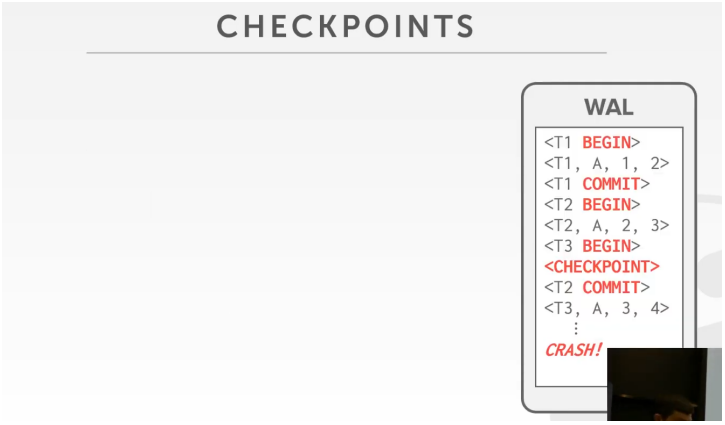
01:06:26 – 01:06:27
so that's bad

So，这很糟糕
1.06.27–1.06.34
so the way we can truncate the log is to what are called checkpoints
So，我们截断日志的方式叫做checkpoints
01:06:34 – 01:06:36
and the idea of a checkpoint is that
checkpoints的思路是
1.06.36–1.06.41
we're gonna flush out all the pages that are dirty in our buffer pool out the disk
我们会将我们buffer pool中所有dirty page刷出到磁盘



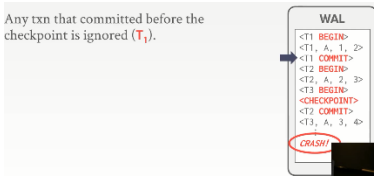
01:06:41 – 01:06:43
and add a entry to our log record to say
并往我们的日志记录中添加一个条目，以此表示
1.06.43–1.06.45
at this point
此时
1.06.45–1.06.48
there's no dirty pages that are not durable in disks
所有的dirty page都被持久化到了磁盘上
01:06:48 – 01:06:49
so therefore
因此
1.06.49–1.06.55
you don't need to replay that far in the past potentially from my checkpoint
你无须去重新执行过去那么多日志，你需要从我的checkpoint处开始执行即可
01:06:56 – 01:06:58
because I know all those changes have been persistent
因为我知道checkpoint前的这些修改已经被持久化了
01:06:59 – 01:07:00
again the idea here again
这里的思路是
1.07.00–1.07.05
because we're doing the steal policy or the no force policy
因为我们使用的是Steal+No–Force策略
01:07:05 – 01:07:10
we're not requiring that the dirty pages that transaction made has to be flushed out the disk before the transaction is committed
我们并不要求在事务被提交前，该事务所修改的这些dirty page被刷出到磁盘
01:07:11 – 01:07:16
so we don't know whether they actually made it to disk or not，if we crash
So，如果我们崩溃的话，我们并不清楚这些page是否被刷出到磁盘
01:07:16 – 01:07:17
whereas with the checkpoint
然而，如果使用的是checkpoint的话
1.07.17–1.07.19

when we know the checkpoint completes
我们知道，当这个checkpoint完成的时候
1.07.19–1.07.22
we know that at that point everything has been written a disk
我们知道在这个点上，所有东西都已经被写入到磁盘

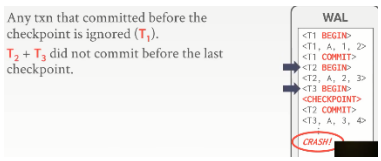


01:07:25 – 01:07:27
so look at really simple example here
So，我们来看个很简单的例子
01:07:27 – 01:07:28
so for this one
So，在这个例子中
1.07.28–1.07.30
I'm going to use a very simplistic checkpoint scheme
我会使用一种非常简单的checkpoint方案
1.07.30–1.07.39
that basically ~~stops the world~~ stops all transactions from running ,and flushes out all their changes all the dirty pages out to disk
简单来讲，就是停止所有正在执行的事务，并将它们的相关dirty page刷出到磁盘
01:07:39 – 01:07:41
and then once I know all the dirty pages are written
接着，一旦我知道所有的dirty page都落地到磁盘
1.07.41–1.07.43
then I let them to start running again
那么，我会让这些事务再次开始执行
01:07:44 – 01:07:48
this is called consistent checkpointing or blocking checkpoints
这种方式叫做Consistent Checkpoint或者叫做Blocking Checkpoint
01:07:48 – 01:07:50
most systems don't actually implement it this way
大多数数据库系统实际不会以这种方式去实现它
1.07.50–1.07.54
we'll see on Wednesday how to do it better and how things can run at the same time
我们会在周三的时候去看下如何将它变得更好，并让这些东西同时执行
01:07:54 – 01:08:00
but just understand the very basic how the basic protocol works assume that's the case
但我们要去理解下这个基础协议的工作方式
01:08:00 – 01:08:02
so I'm gonna add this checkpoint entry here

So，我会在这里添加个checkpoint条目
1.08.02–1.08.04
so what will happen is
So，这里所发生的事情是
1.08.04–1.08.05
when I take this checkpoint,
当我拿到这个checkpoint时
1.08.05–1.08.07
I stop all transaction from running
我会停止所有正在执行的事务
1.08.07–1.08.09
and I flush out any dirty pages
接着，我将所有的dirty page刷出到磁盘

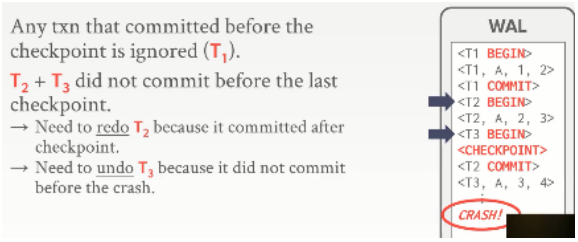


01:08:09 – 01:08:11
and so now if I have a crash .
So，现在如果我遇上崩溃
1.08.11–1.08.13
when I come back
当我系统恢复正常的时候
1.08.13–1.08.18
,I know that I don't need to look at any T1's changes
我知道我无须去查看T1所做的任何修改
01:08:18 – 01:08:22
because t1 committed before my checkpoint
因为T1在我的checkpoint之前就提交了
1.08.22–1.08.24
so I know all the t1 changes are written a disk
So，我知道T1所做的所有修改都落地到磁盘了
01:08:25 – 01:08:27
so I don't need to replay and look at it's log
So，我无须去查看并重新执行这段日志

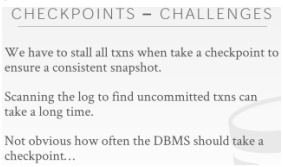


01:08:27 – 01:08:29
it's the other two guys t2 and t3
这里还有其他两个事务，即T2和T3
1.08.29–1.08.31
those guys get started
这两个事务开始执行
1.08.31–1.08.33
and could potentially make change before my checkpoint
它们可能在我的checkpoint之前就执行了修改
01:08:33 – 01:08:36

so I need to go back in the log up to that point
So，我需要回到日志中这个点的位置
1.08.36–1.08.40
and figure out what they actually did to put my data back in the correct state
并弄清楚它们实际做了什么事情，以此来让我的数据回归正轨
01:08:40 – 01:08:41
so in the case here
So，在这个例子中
1.08.41–1.08.44
T2 committed before the crash
T2在崩溃发生前就提交了
01:08:44 – 01:08:48
so I know I want to reapply it changes to redo their exchanges
So，我想去重新执行它的修改
01:08:48 – 01:08:50
t3 did not commit before the crash
T3并没有在崩溃前提交事务
1.08.50–1.08.52
so I would know I want to reverse its changes
So，我就知道我想去撤销它所做的修改



01:08:54 – 01:09:02
right so the checkpoints basically if your skin is telling us that we know that at this point in time all dirty pages from any transactions have been written to disk
简单来讲，checkpoint指的是，在这个时间点之前，任何事务所修改的dirty page已经被写入到磁盘了
01:09:03 – 01:09:12
and it's up for us to then figure out ,well what came slightly before the checkpoint and what came after the checkpoint to decide, who's allowed to actually have their changes persisted



01:09:15 – 01:09:25
so well game will talk about checkpoints more on Wednesday ,but in my simple example here, I stalled all the transactions to make my life easier

01:09:25 – 01:09:28
because if I have a transaction that's updating a bunch of pages
因为如果我有一个事务，它对一堆page进行更新
01:09:28 – 01:09:38
I don't want to have the case or it could be I had to do some actual work to figure out, well I'm updating 20 pages ,and my checkpoint flushed out the first 10 pages that transaction modified

01:09:38 – 01:09:42
but then while the checkpoint was running it modified these other ones I didn't flush those things out

01:09:42 – 01:09:45
so I don't want to figure out which ones actually should be around

01:09:47 – 01:09:51
the other tricky thing is gonna be it's not clear how often we should take a checkpoint

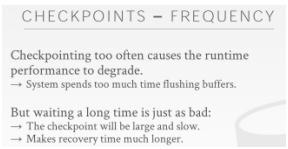
01:09:51 – 01:09:53
because these checkpoints aren't free

01:09:53 – 01:09:58
because we're writing out dirty pages and that's slowing slowing the disk,when we could be running out to a log

01:09:58 – 01:10:04
now in a lot of systems they'll have the disks, the log we stored in separate disks ,and the heat balls are sort of separate disks

01:10:04 – 01:10:07
so when you do a net syncs on both of them ,you're not slowing down each other

01:10:07 – 01:10:19
but again now my checkpoint is running out dirty pages, when I could have been you know evicting pages for disk to get new space in my buffer pool to have other transactions keep on running

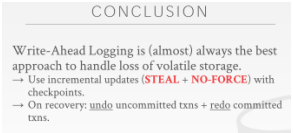


01:10:20 – 01:10:23
so how often you take a checkpoint can vary based on the implementation
So，你制作checkpoint的频率取决于你的具体实现
01:10:24 – 01:10:27
so the one approach is to say
So，其中一种方案是
1.10.27–1.10.31
every after every every minutes or seconds, take the checkpoint
比如每分钟或者每秒就制作一份checkpoint
01:10:32 – 01:10:33
if I do that
如果我这样做了
1.10.33–1.10.35
then my recovery time is much faster
那么，我的恢复时间就会短很多
1.10.35–1.10.39

because now I don't need to go as far back in the log to figure out what should be actually persisted
因为我就无须跑到日志很前面的地方去弄清楚哪些数据被持久化了
01:10:39 – 01:10:41
because my checkpoint is occurring more frequently
因为我记录checkpoint的频率非常频繁
01:10:42 – 01:10:45
because now the checkpoints are slowing me down ,so I'm like my runtime performance suffers
因为这个原因，这就会让我的运行时性能降低
01:10:46 – 01:10:47
another approach which I actually think is better
实际上，我觉得另一种方案要来得更好
1.10.47–1.10.52
is that the checkpoint only occurs after a certain amount of data that ran out the log
只有当日志中的数据满了的时候，我们才会去制作checkpoint
01:10:53 – 01:10:55
like after I've written out 250 megabytes of data to the log
比如，当我往日志中写入250MB大小的数据后
1.10.55–1.10.57
then I take a checkpoint
然后，我就制作了一份checkpoint
01:10:57 – 01:11:00
and that that bounds the amount of time you have to wait
这限制了你必须等待的时间量
1.11.00–1.11.07
,and you don't worry about whether you're taking checkpoints unnecessarily

01:11:07 – 01:11:14
because it's your I know that I only need to look at me but I'd most join fifty megabytes of the log before at the recover my database

01:11:15 – 01:11:18
again I'm going over this very very very fast cuz we're running out of time
我讲的速度非常非常快，因为我们没有太多时间了
01:11:18 – 01:11:22
but we'll cover this more detail when we talk about recovery on Wednesday
但我们会在周三讨论恢复的时候，对此进行更多介绍

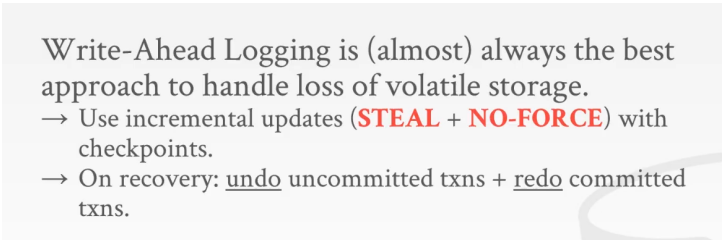


01:11:23 – 01:11:25
so any any high–level questions about checkpoints.
So，关于checkpoints，你们有任何高逼格的问题吗?
1.11.25–1.11.30
saying a checkpoint is like garbage collection for the write ahead log
checkpoints就像是对预写式日志进行垃圾回收
01:11:30 – 01:11:33
but i know that at that point of checkpoint

但我知道，在我制作checkpoint的那个点
01:11:34 – 01:11:37
I don't potentially need to look at anything it came before
我无须去查看该checkpoint之前的任何东西
01:11:40– 01:11:41
of course

1.11.41–1.11.43
in the extreme case
在极端情况下
1.11.43–1.11.45
if I have it if I would a transaction the runs for days
如果我有一个运行了好几天的事务
01:11:45 – 01:11:46
and I'm taking a checkpoint every five minutes
我会每五分钟设置一个checkpoint
1.11.46–1.11.51
, and I need to go back to when that transaction started to figure out what all changes actually made
我需要回过头去弄清楚该事务开始到现在实际做了哪些修改
01:11:53 – 01:11.54
okay

1.11.54–1.11.55
so as I said
So，正如我说的
1.11.55–1.12.06
write ahead logging is is almost always, the preferable approach that the better approach to handle avoiding data loss or to make sure that our database is durable on disk
预写式日志始终是一种更佳方案，它能用来避免数据丢失，或者确保我们的数据库持久化到了磁盘上
01:12:06 – 01:12:09
and the core idea what's gonna how it works is that,
它的核心思想是



1.12.09–1.12.12
it's gonna use Steal+ No force buffer pool management policy
它使用的是Steal+No–Force buffer pool管理策略
01:12:13 – 01:12:19
it's gonna flush all changes that transactions made to their log records to disk, before we tell the outside world that a transaction is committed
在我们告诉外界该事务已被提交前，它会将该事务对它们日志记录所做的修改全都刷到磁盘上

01:12:20 – 01:12:24
and then in the background of some later point we can flush out there's those dirty pages
在稍后某个时间点，我们会在后台将这些dirty page刷出去
01:12:24 – 01:12:28
but we have to write the log records first before we can write out the dirty pages that they modify
在我们可以将这些修改过的dirty page写出之前，我们必须先写到日志记录上
01:12:29 – 01:12:34
and so on recovery ,we just undo any changes from uncommitted transactions
So，在recovery这个过程中，我们只需撤销那些未提交事务所做的修改
01:12:34 – 01:12:39
and they ~~redo the commit changes the committed to~~ redo the change of any committed transactions to make sure that they get applied
它们会重新执行任何已被提交事务所做的修改，以确保它们被应用
01:12:40 – 01:12:40
yes
请问
01:12:45 – 01:12:48
this question is we have to undo potentially changes on recovery
他的问题是，我们需要撤销恢复过程中所做的修改
1.12.48–1.12.49
again we'll cover that on Wednesday
我们会在周三的时候介绍这个
01:12:50 – 01:12:57
because changes from uncommitted transactions could have those dirty pages could be written out the disk
因为这些未提交事务所修改的这些dirty page可能会被落地到磁盘上
1.12.55–1.12.57
because we're using the steal policy
因为我们使用了Steal策略
01:13:05 – 01:13:08
his question is
他的问题是
1.13.08–1.13.12
other scenarios where upon recovery we would not have to undo

01:13:12 – 01:13:15
because we can look at data art changes actually make it out the disk
因为我们关注的是那些实际已经落地到磁盘的修改

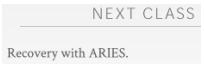
01:13:23 – 01:13:24
No
No

1.13.24–1.13.27
the spoiler would be for Wednesday ,you redo everything

01:13:28 – 01:13:30

you could go through the log of multiple times
你可能会对日志进行多次遍历
01:13:30 – 01:13:31
you're gonna redo everything
你会重新执行所有操作
1.13.31–1.13.33
, but then as you redo
但当你重新执行操作的时候
1.13.33–1.13.35
you say oh I see this transaction didn't commit
你表示：Oh，我看到这个事务并未被提交
1.13.35–1.13.39
then you go back and reverse on the log ,and you undo anything that is on

1.13.39–1.13.43
,so you play it safe and you always undo

01:13:43 – 01:13:46
there are some optimizations that I don't think most people do them
我们可以对它们进行一些优化，但我不觉得大部分人会做


01:13:49 – 01:13:50
all right

01:13:51 – 01:13:53
So，on Wednesdays class
在周三的课上
1.13.53–
again it would be the second part of what we talked about for logging recovery, is two things we do after a crash right after a restart

01:13:58 – 01:14:01
how do we use the write ahead log, how to use the check points to put us back in the correct State
你我们该如何使用预写式日志，以及该如何使用checkpoint来让我们的数据库回归正轨
01:14:01 – 01:14:05
so this is probably the third hardest part of database systems
So，这可能是数据库系统中第三难的地方
01:14:06 – 01:14:08
so the thing we're talking about is Aries
So，我们要讨论的东西是ARIES
1.14.08–1.14.11
,Aries is the gold standard of how you do database recovery
ARIES是你做数据库恢复的时的黄金准则
1.14.11–1.14.13
I don't know what the textbook calls it Aries
我不清楚教科书上为什么将它叫做ARIES
01:14:14 – 01:14:19

most systems that implement write ahead of all are not gonna call but they're doing Aries
大部分系统实现预写式日志的数据库系统，它们使用的方法其实都是ARIES
01:14:19 – 01:14:23
but everybody that does write ahead logging it's me based on the IBM's protocol from the 1990s
但所有人包括我实现预写式日志的方式都是基于1990年代IBM所提出的那个协议
01:14:23 – 01:14:26
whether or not they know that they're using their basically using Aries
不管他们知不知道他们使用的就是ARIES
01:14:27 – 01:14:27
Okay

01:14:28 – 01:14:29
all right

1.14.29–1.14.32
I'm having office hours now at 1:30
我1:30的时候会在办公室
1.14.32–1.14.35
,and see you guys on Wednesday
周三再会