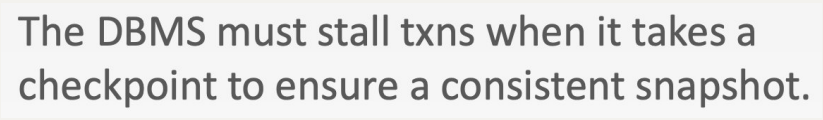
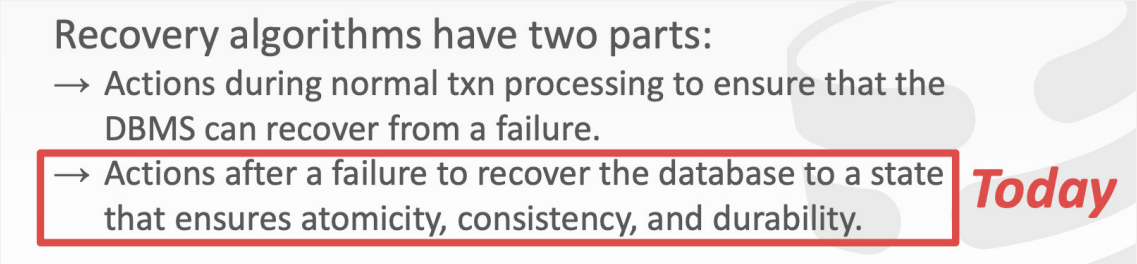
首先补充一下上一个Lecture的笔记里有疏漏的地方：只有当DBMS中已有的事务都结束，而且新的事务没有开始，也就是数据库中没有任何事务在进行，整个数据库是静态的时候，在这样的一个“空档期”才能做checkpoint，这样才能保证实现一致性的快照



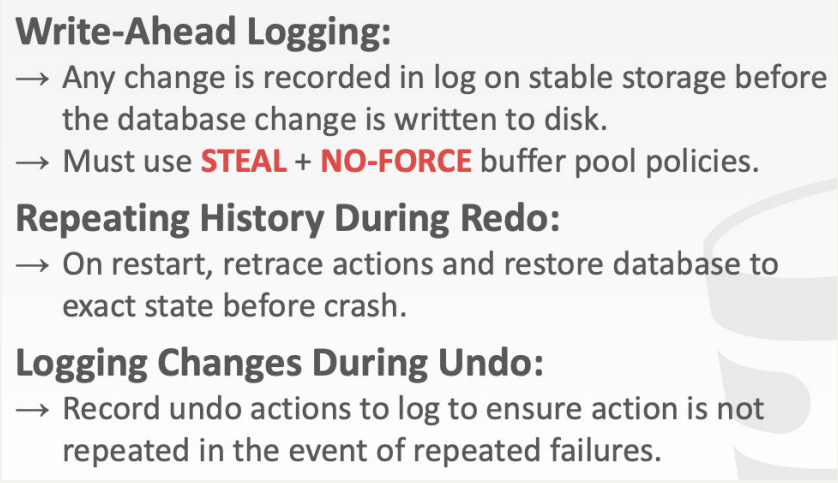
本Lecture将详细地介绍数据库在宕机后做恢复的时候会干什么，有没有什么 比checkpoint更好的恢复方法?



本Lecture将介绍ARIES算法，

翻译成中文是“数据库恢复原型算法”，这个算法是用来做数据库恢复的，但只是一个原型，提供了一个基本的理念，工业界中各个数据库基于这个理念给出了各种不同的实现

ARIES算法主要的想法如下：



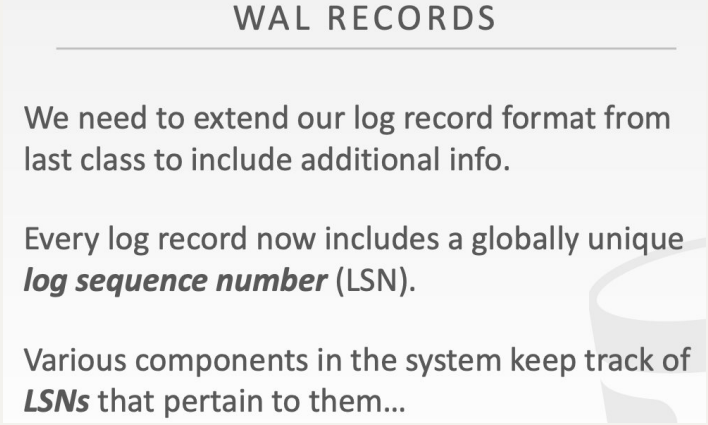
- WAL要先于数据落盘

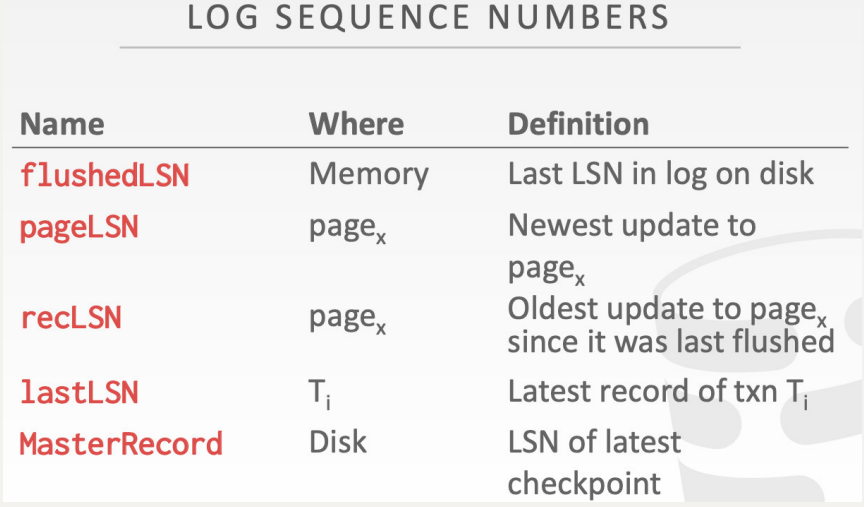
- 必须使用 Steal + No-Force 的缓存池管理策略
- 当 DBMS 重启时，按照日志记录的内容做回放，恢复到故障发生前的状态
- 在 undo 过程中记录 undo 操作到日志中，确保在恢复期间再次出现故障时不会执行多次相同的 undo 操作

Log Sequence Numbers

LSN，日志序列号

WAL中的每条日志记录都需要包含一个全局唯一且一般是单调递增的log sequence number (LSN)。而DBMS中的不同部分都需要记录某些相关的LSN信息





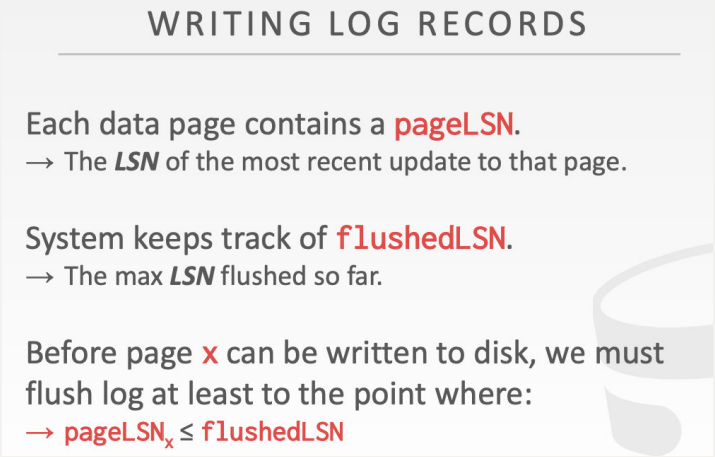
- flushedLSN：最后落盘的那个 LSN，用于记录现在有哪些日志记录已经被刷入磁盘

- pageLSN：每一个数据页都有一个pageLSN，代表着对这个页进行最近一次修改的

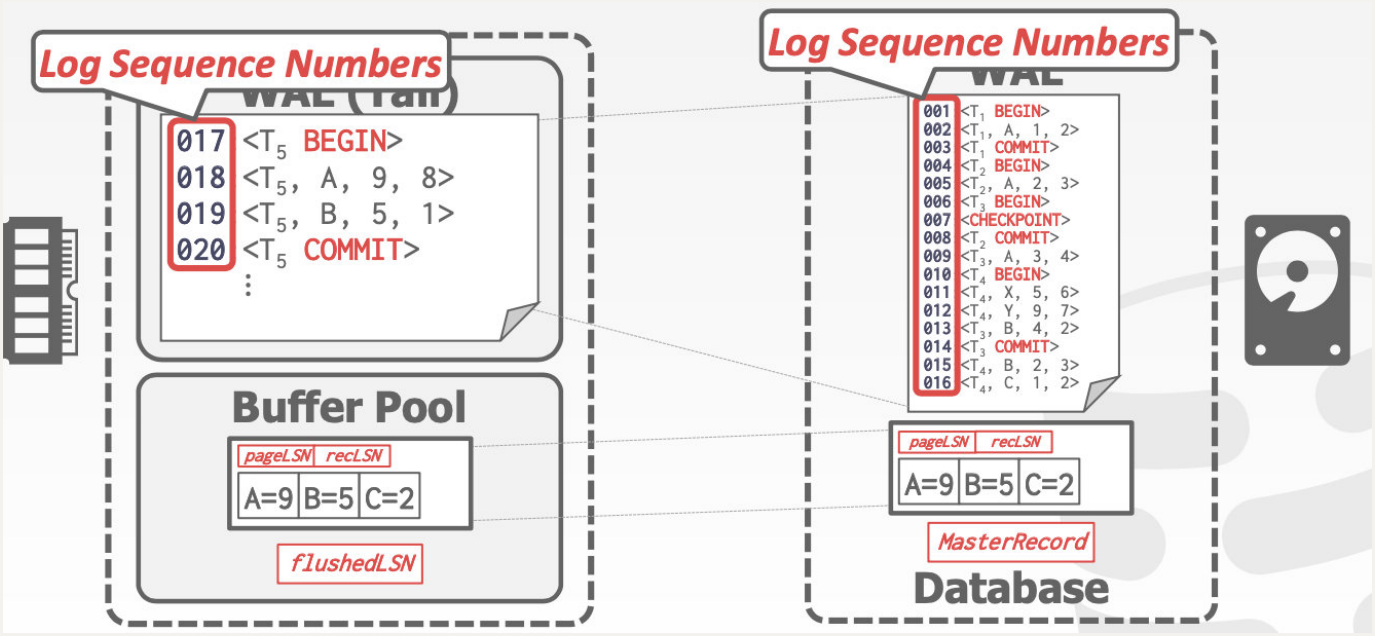
SQL语句对应的日志的LSN，也就是记录的是缓存当中这个页最新的修改

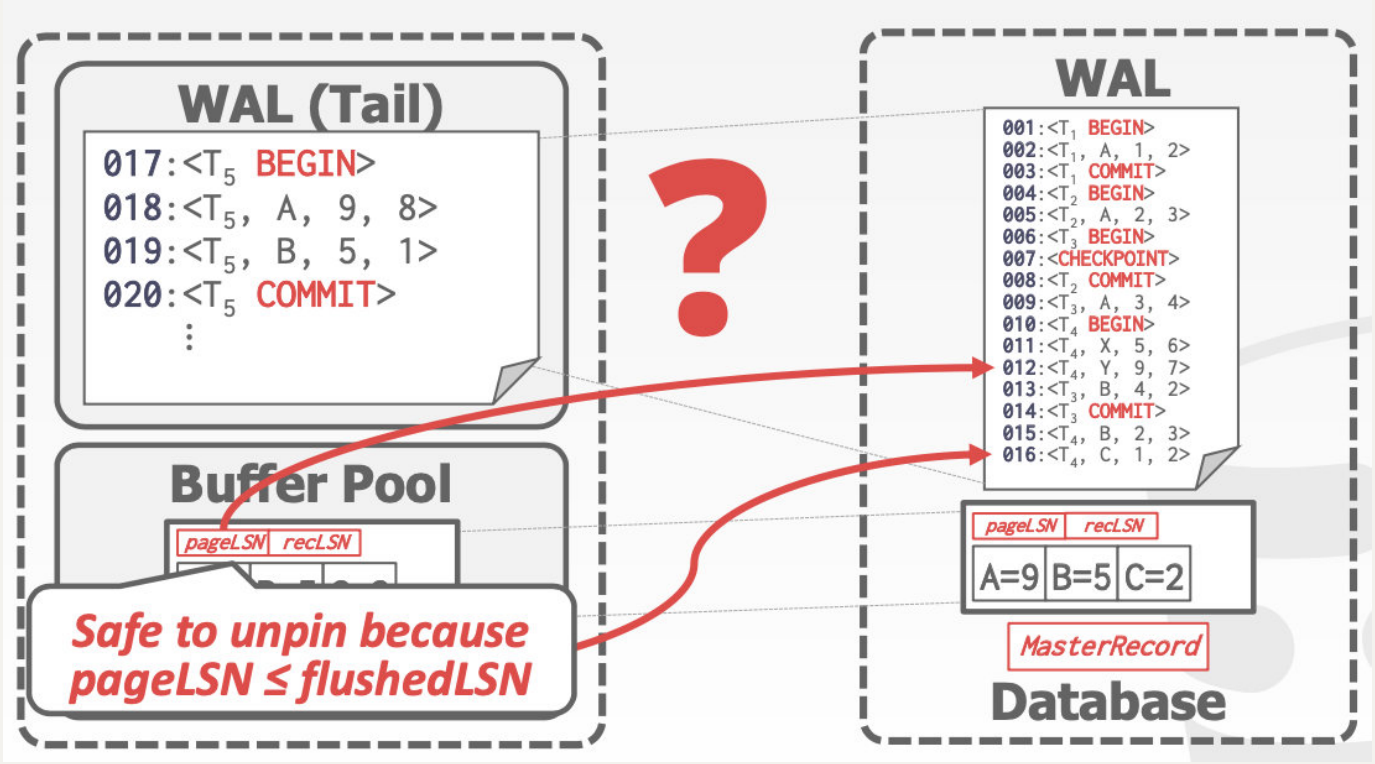
- recLSN：也是在数据页里面，用于记录自从这个页上次被刷入磁盘之后第一个修改这个页的SQL语句对应的日志的LSN，也就是记录的是缓存里面比磁盘上的页新的第一个版本，或者说是内存里面这个页最老的修改，pageLSN和recLSN是缓存里面对于这个页的修改的上限和下限
- lastLSN：用于记录到目前为止某个事务留下的最后一条日志的 LSN
- MasterRecord：上一次打的checkpoint点对应的LSN

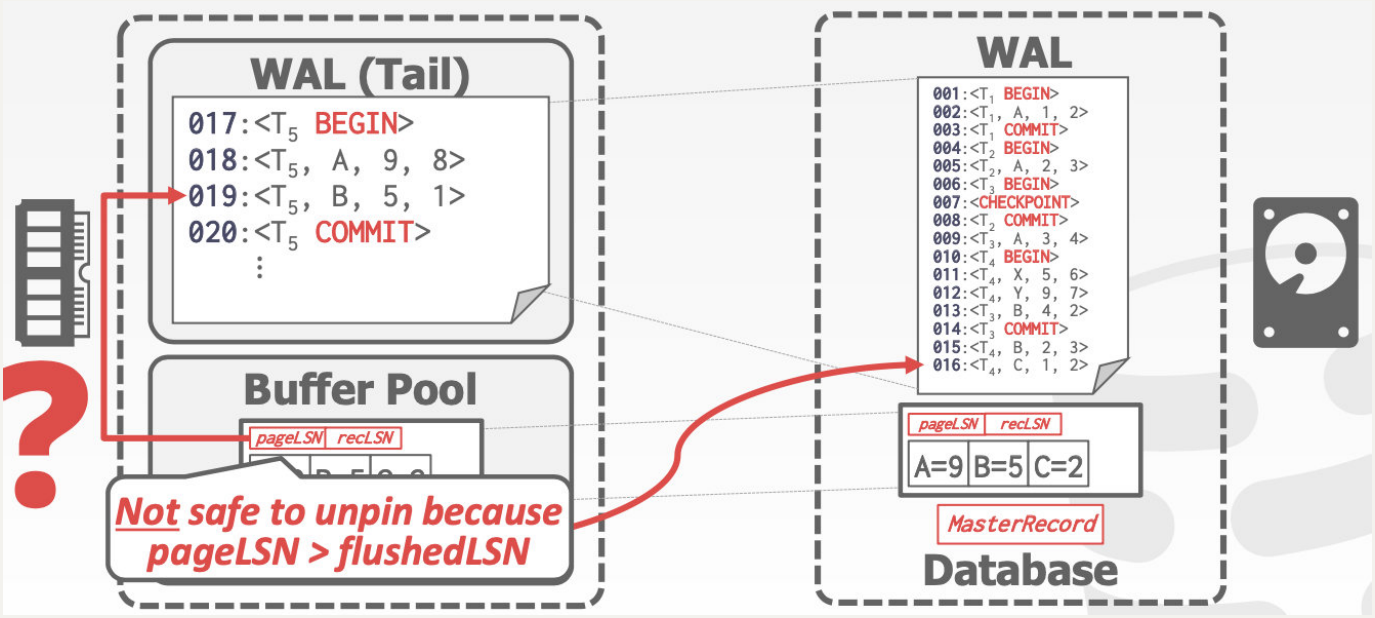
将脏页中的数据更新刷入磁盘的时候要保证修改过这个脏页的所有操作所对应的所有日志记录已经被刷入磁盘，并且比这些日志记录的LSN小的日志记录也都要被刷进磁盘，也就是满足如下的关系：



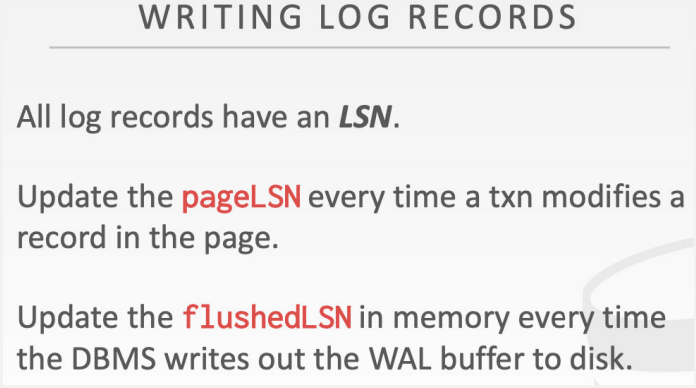
结合slides中的例子理解：







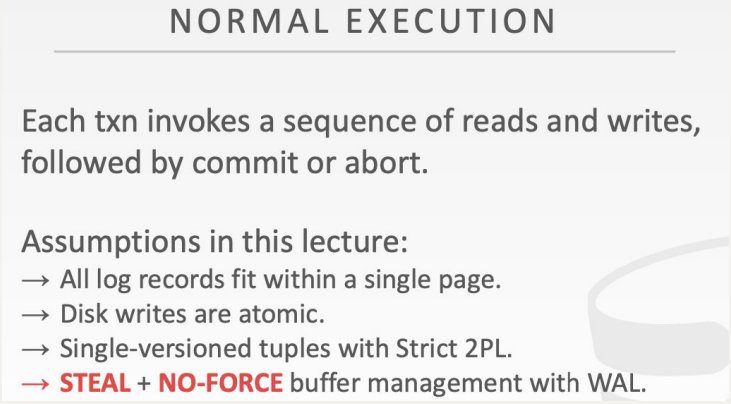
事务每一次修改缓存里面的数据页，都要顺带修改pageLSN，每次把内存里的log刷入磁盘时也要顺带更新flushedLSN



Normal Commit & Abort Operations

在事物正常地commit与回滚的时候要做的操作

为了便于研究问题，我们首先做如下的假设：



事务commit时向磁盘中写入日志的过程是连续的写（因为就是在磁盘中的日志文件后面不断地追加），而且是同步的（synchronous writes to disk，线程会一直阻塞，直至把日志记录全部写入磁盘，全部写入后会返回给用户commit成功）

在事务完全成功（在将来某一时刻，除了相关的日志以外的数据更新也都完成了落盘）的时候会再写一条'TXN-END'这样的日志记录，但它不需要马上落盘

TRANSACTION COMMIT

Write COMMIT record to log.

All log records up to txn's COMMIT record are flushed to disk.

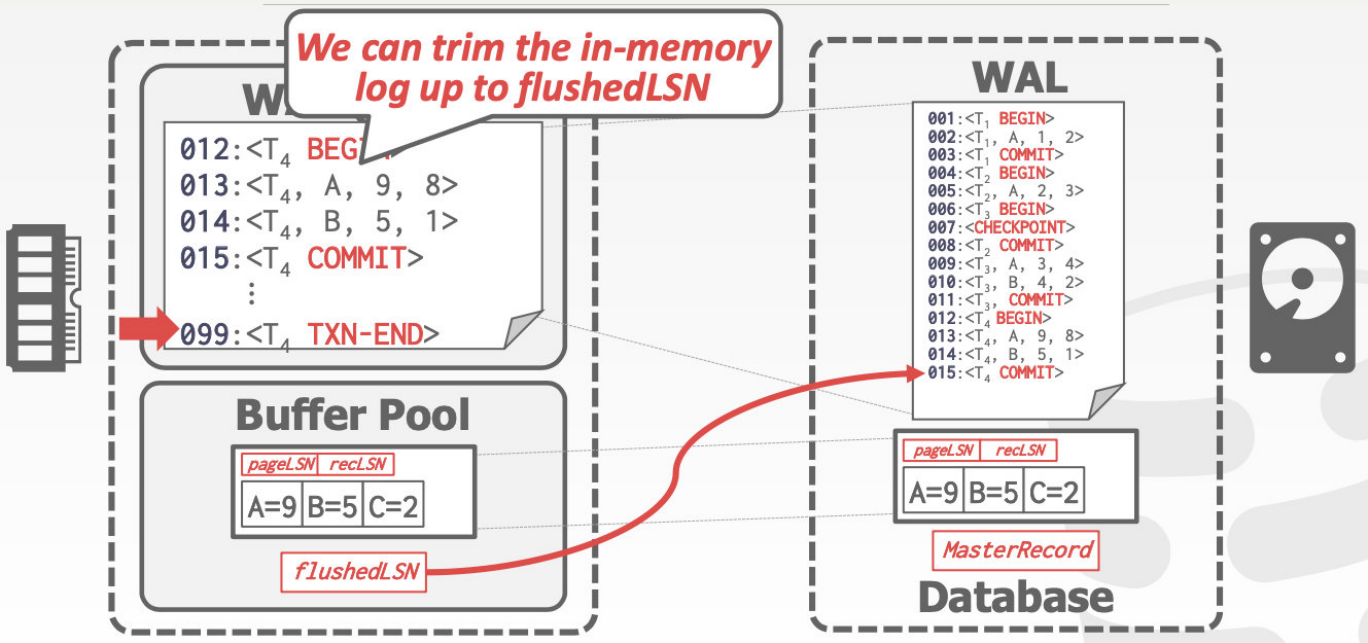
→ Log flushes are sequential, synchronous writes to disk.

→ Many log records per log page.

When the commit succeeds, write a special TXN-END record to log.

→ This does not need to be flushed immediately.

结合slides中的例子理解：



当事务提交时，DBMS 先写入一条 COMMIT 记录到 WAL ，然后将 COMMIT 及之前的日志落盘。一旦COMMIT记录安全地存储在磁盘上，DBMS就向应用程序返回事务已提交的确认信息，并将flushedLSN 被修改为 COMMIT 记录的 LSN。在将来某一时刻，DBMS 会将内存中 COMMIT 及其之前的日志清除，并将缓存池中的脏页刷回磁盘，完成后再写入一条 TXN-END 记录到 WAL 中，作为内部记录。

前面介绍了事务在正常commit的时候所做的操作，接下来介绍事务在回滚的时候所做的操作：

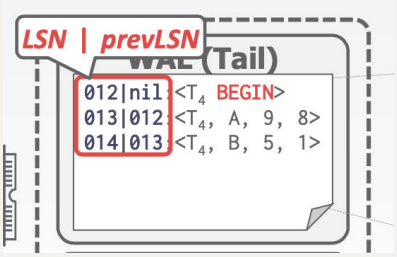
TRANSACTION ABORT

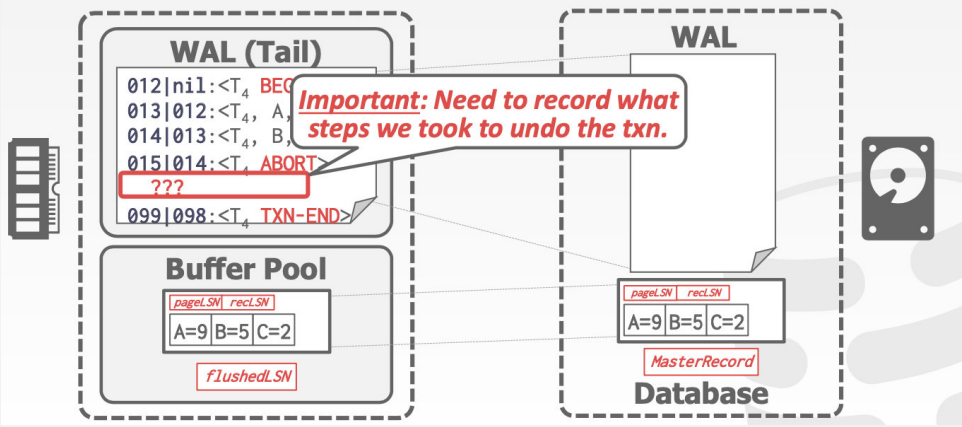
Aborting a txn is a special case of the ARIES undo operation applied to only one txn.

We need to add another field to our log records:
→ prevLSN: The previous ***LSN*** for the txn.
→ This maintains a linked-list for each txn that makes it easy to walk through its records.

事务回滚对于数据库来说是特殊情况，因此ARIES算法也会做特殊的处理：在日志记录中再加一个字段：prevLSN，加在每一条日志记录的后面，它代表着这一条日志的上一条日志的LSN（e.g. 15号日志记录的上一条日志记录不一定是14号日志记录，因为它们可能属于不同的事务），这就方便事务回滚的时候找到它的所有日志

结合slides中的例子理解：





在对事务做回滚，也就是进行undo操作的时候，要记录"compensation log records"（简称CLR），也就是把进行过的回滚操作也写入日志，从而防止在回滚过程中再次故障导致部分操作被执行多次。等待所有操作回滚完毕后，DBMS 再往 WAL 中写入 TXN-END 记录，意味着所有与这个事务有关的日志都已经写完，不会再出现相关信息。

CLR 记录的是 undo 操作，它除了记录原操作相关的记录，还记录了 undoNext 指针，指向下一个将要被 undo 的 LSN

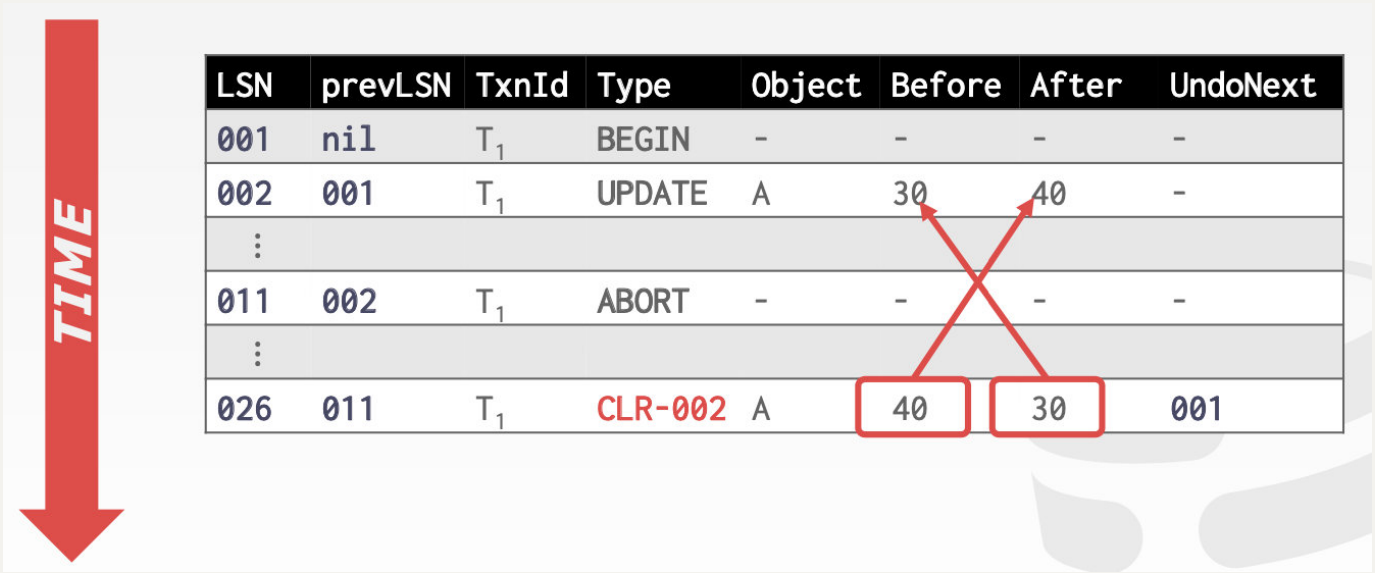
CLR日志不需要强制落盘，因为大不了就下次DBMS重启的时候重新回滚一遍，不影响事务的原子性（也就是说，在事务abort的时候可以直接告知用户事务abort了，而不必等到所有的undo操作都做完并且CLR都落盘）

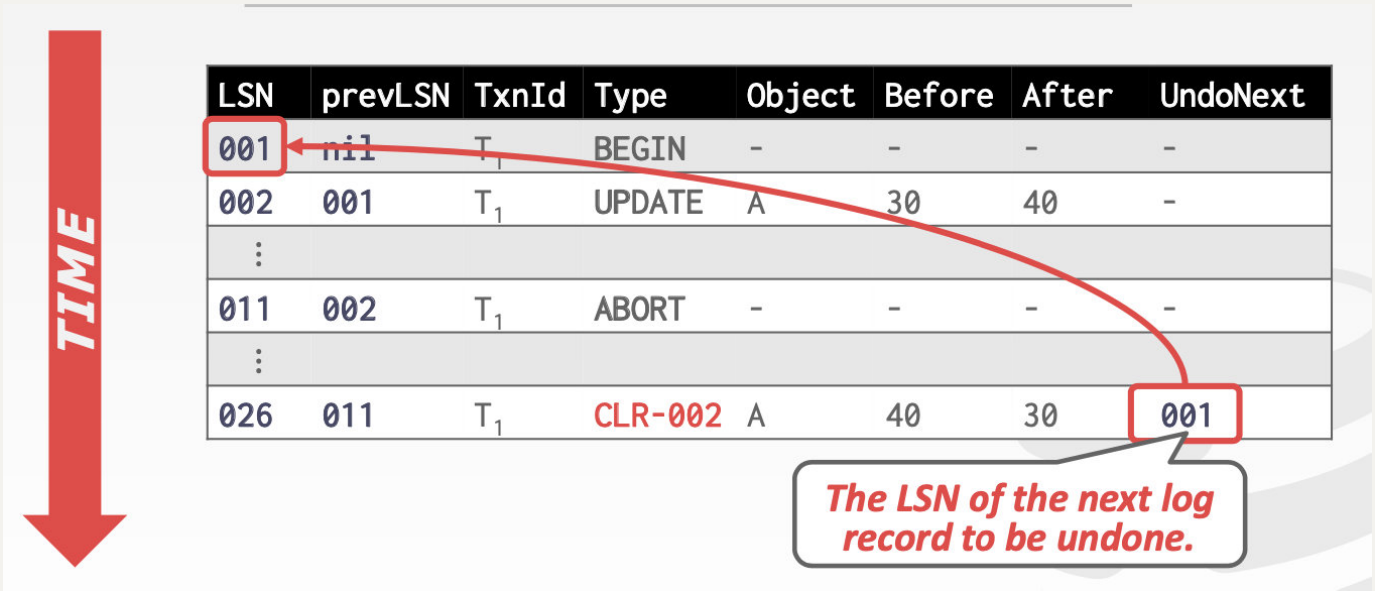
COMPENSATION LOG RECORDS

A **CLR** describes the actions taken to undo the actions of a previous update record.

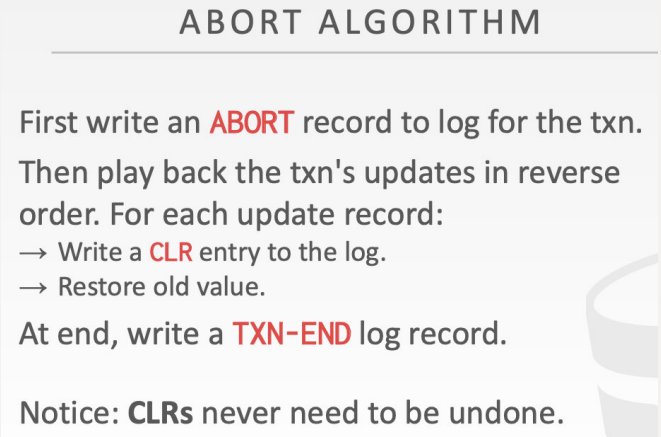
It has all the fields of an update log record plus the undoNext pointer (the next-to-be-undone LSN).

CLRs are added to log records but the DBMS does not wait for them to be flushed before notifying the application that the txn aborted.





总结一下abort操作的流程：

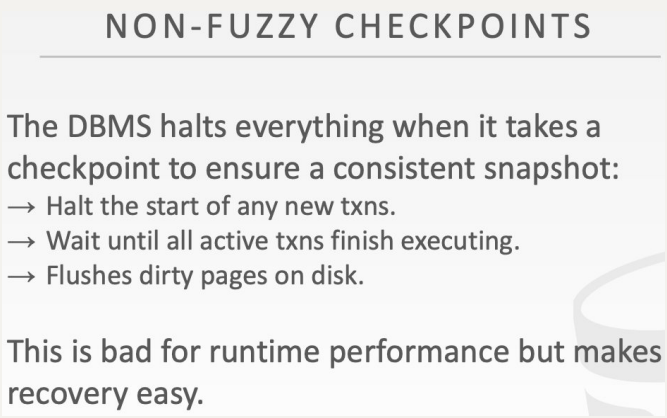


Fuzzy Checkpointing

Fuzzy Checkpointing是对之前介绍的checkpoint机制的优化

fuzzy即是“模糊”的意思

在前面所介绍的checkpoint实现方法中，DBMS在标记checkpoint的时候不让任何新的事务开始执行，并且把系统中还没有执行完的事务都执行完并且完成相应的数据更新的落盘。如果在还没执行完的那些事务里，有的事务非常的长，那么DBMS就会等待相当长的一段时间，在这段时间里不能接收新的事务，这会使得性能非常的差（虽然但是，这样也是有一些好处的，crash recovery的时候非常方便。在crash recovery的时候，checkpoint之前的所有事务所做的数据更新都已经落盘，因此数据库恢复的时候完全不用看checkpoint之前的日志记录，checkpoint之前的日志记录可以直接被删掉。）

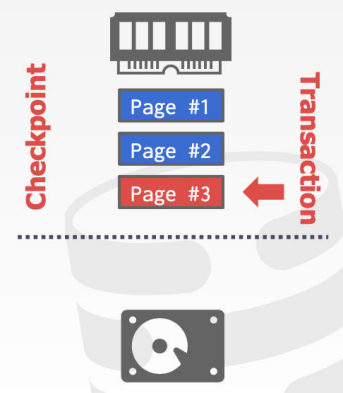


那么相应的就有一些改进方案：

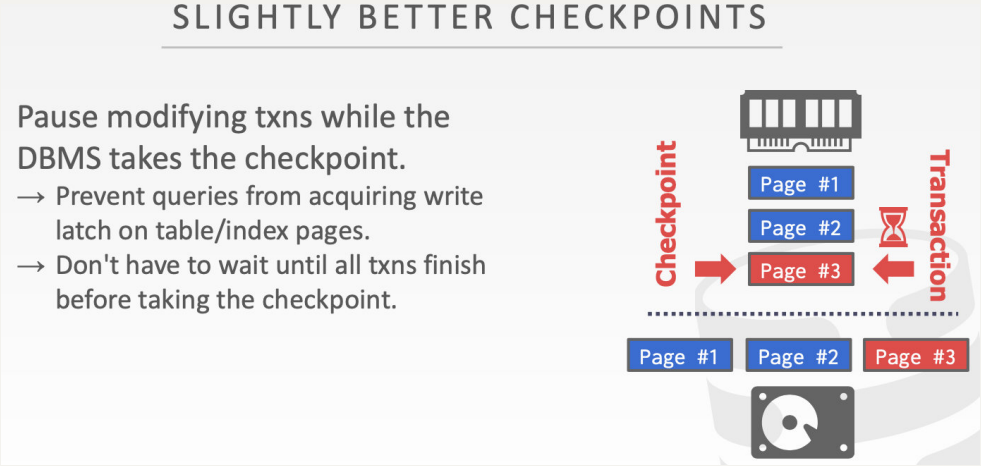
做checkpoint的时候不必等待当前还没执行完的事务去执行完，让它们停下即可，（如果事务是read-only的，那么可以让它继续）

让事务暂停的方法可以是在checkpoint开始后阻止事务获取数据或索引的写锁 (write latch)

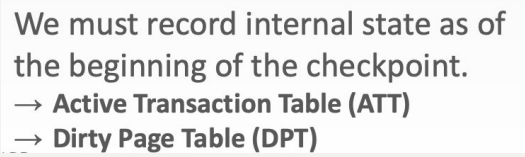
如下所示有两个线程，一个执行checkpointing，一个执行事务，并且事务已经修改了缓存池里的page 3，这时DBMS想要做一个checkpoint



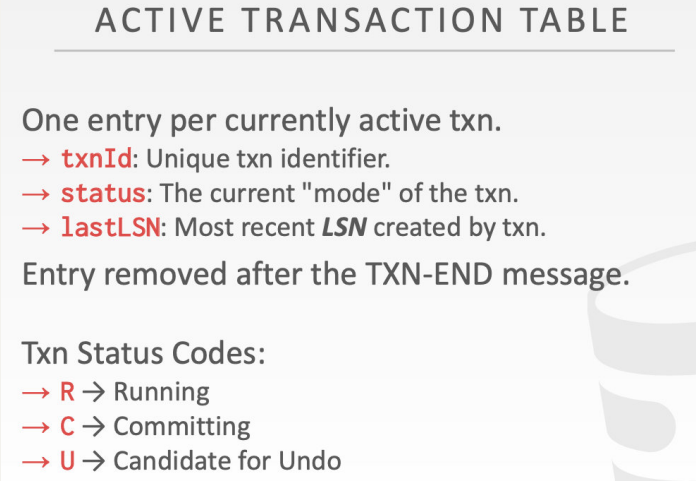
由于事务已经获取了page 3的写锁，它可以继续修改page 3，但不能再获取其它页的写锁，此时checkpoint线程扫描一遍缓存池中的所有页，将所有脏页落盘。checkpoint线程完成工作后，事务可以获取其他页的写锁，继续执行。



这时有些未commit的事务写入的数据可能会被checkpoint线程一起捎带落盘，因此这时磁盘中的数据 snapshot可能处于 inconsistent 的状态。但是只要我们在 checkpoint 的时候记录哪些活跃事务正在进行，哪些数据页是脏的，故障恢复时读取 WAL 就能知道存在哪些活跃事务的数据可能被部分写出，从而恢复 inconsistent 的数据。因此整个 checkpoint 过程需要两类信息：活跃事务表 (ATT)与脏页表(DPT)



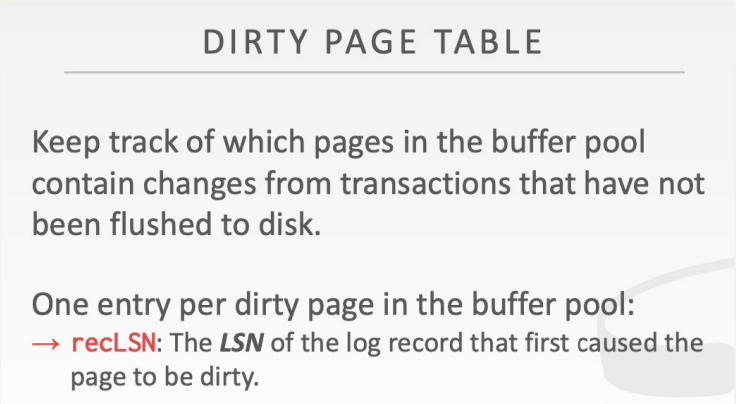
ATT中的每个entry代表着进行checkpoint的时候仍活跃的事务，entry中的内容包括如下所示的三项：



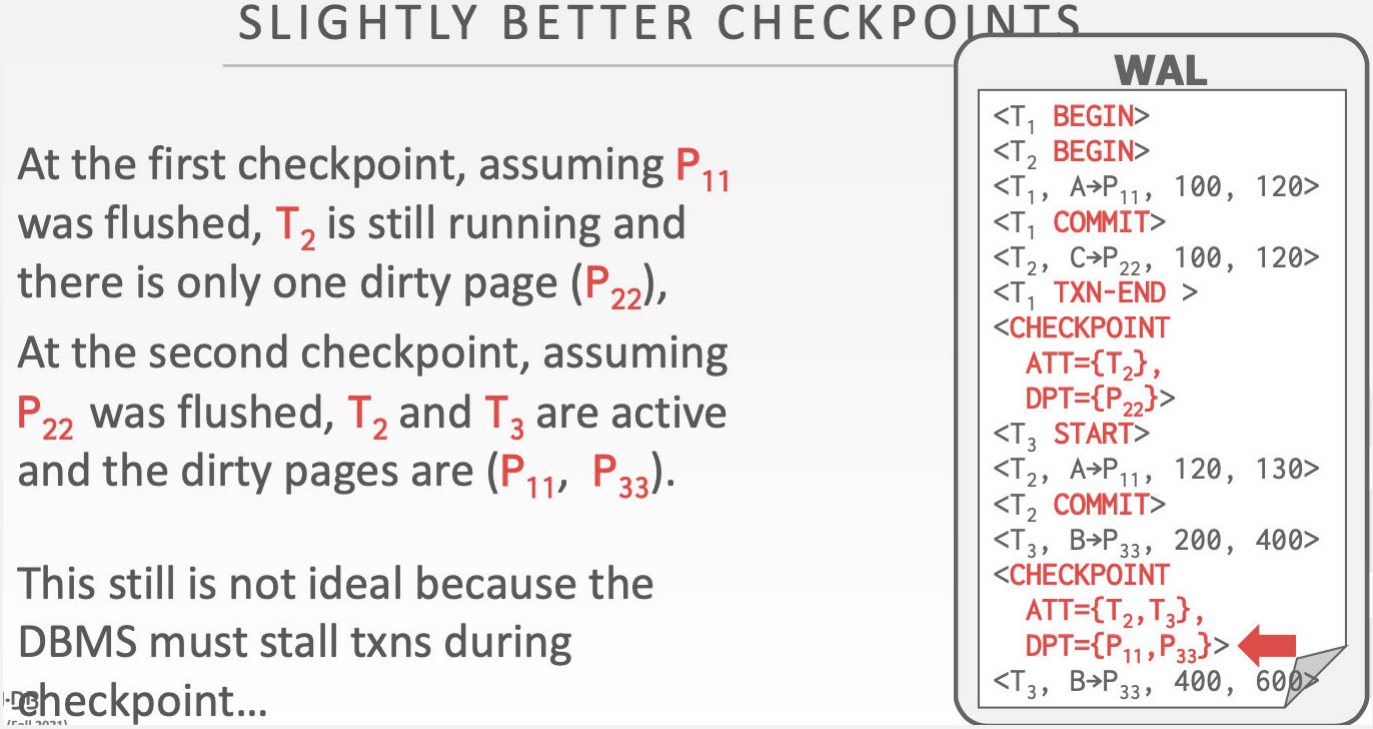
事务彻底完成的时候（也就是它所做的全部更新都落盘，即TXN-END被写入日志时），ATT表中它所对应的entry才可以被删除

（事务状态中的U可以理解为“还没提交”：如果数据库crash的时候这个事务还没提交，那么它就需要undo，所以称为"candidate for undo"）

DPT记录的是缓存池中当前还没落盘的脏页，每一个entry代表一个未落盘的脏页，entry中有这个脏页的recLSN（让这个页开始变脏的log record的LSN）



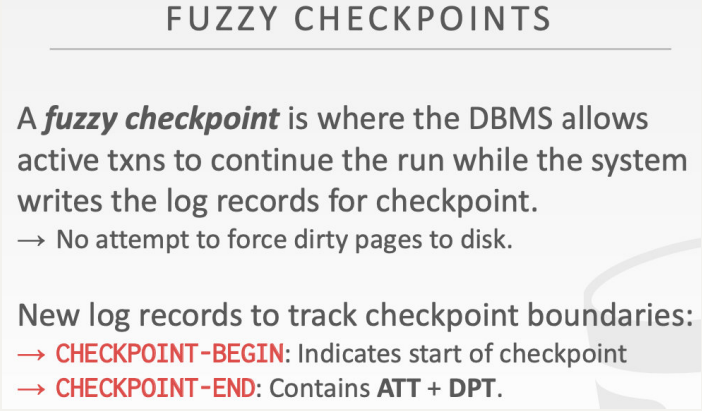
结合例子分析ATT和DPT的作用，如下所示，采取了*slightly better checkpoints*策略后的WAL日志如下所示，checkpoint存档点中多了ATT和DPT



这样的话其实也不算是很完美的策略，尽管比 Non-fuzzy 好一些，不需要等待所有活跃事务执行完毕，但仍然需要在 checkpoint 期间暂停执行所有写事务。

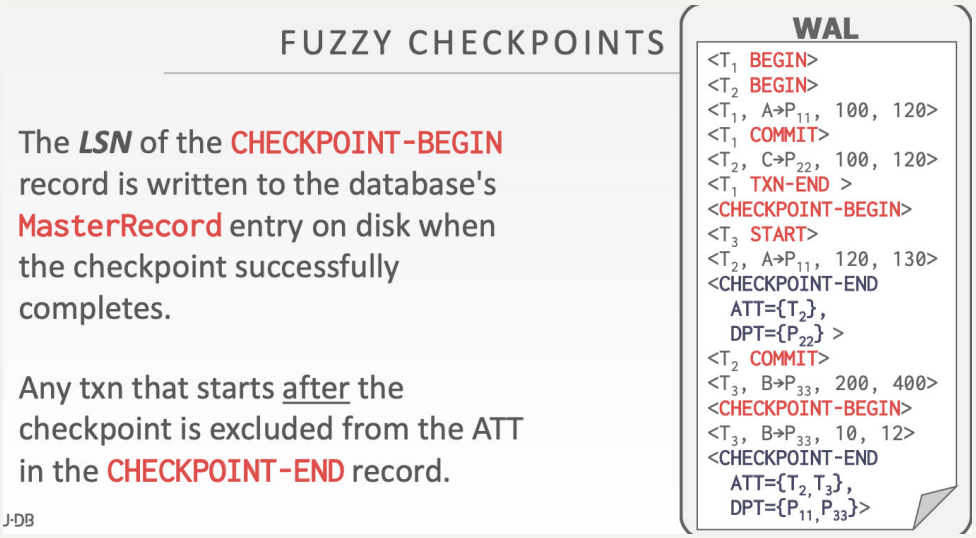
因此便有了最终的优化-**Fuzzy Checkpoints**

*fuzzy checkpoints*策略下，在做checkpoint的时候，所有的事务都可以正常工作，并且不会强制把所有的脏页都落盘



并且*fuzzy checkpoints*会把原先的日志中的checkpoint的一个时间点变成一个时间段：checkpoint阶段会写两条日志：checkpoint-begin与checkpoint-end，对应checkpoint阶段的开始与结束

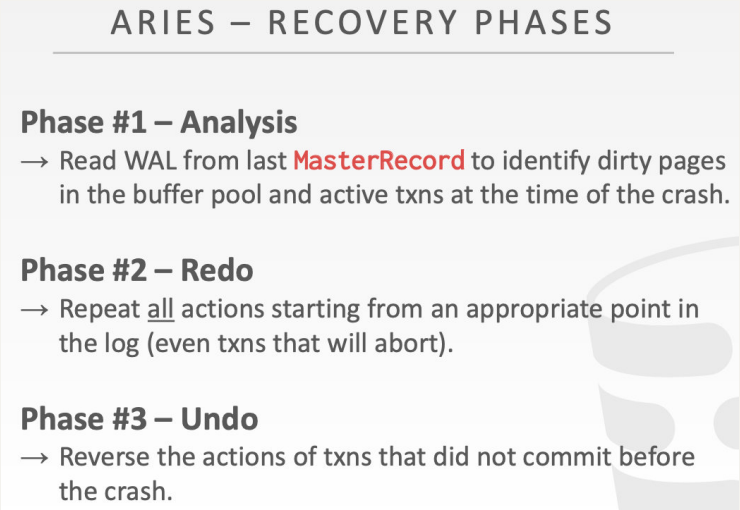
结合如下例子理解：



（T3是在checkpoint-begin之后开始的，因此不会被记入ATT）

Recovery Algorithm

ARIES分为如下三个步骤



- Analysis Phase:在DBMS宕机重启之后把磁盘上的WAL文件读入缓存，并找到其中的MasterRecord，定位到上一次checkpoint的存档点，之后分析它后面的所有日志，并且结合ATT和DPT中的信息，从而推断出DBMS在宕机时系统的情况，进而决定该undo哪些事务，redo哪些事务

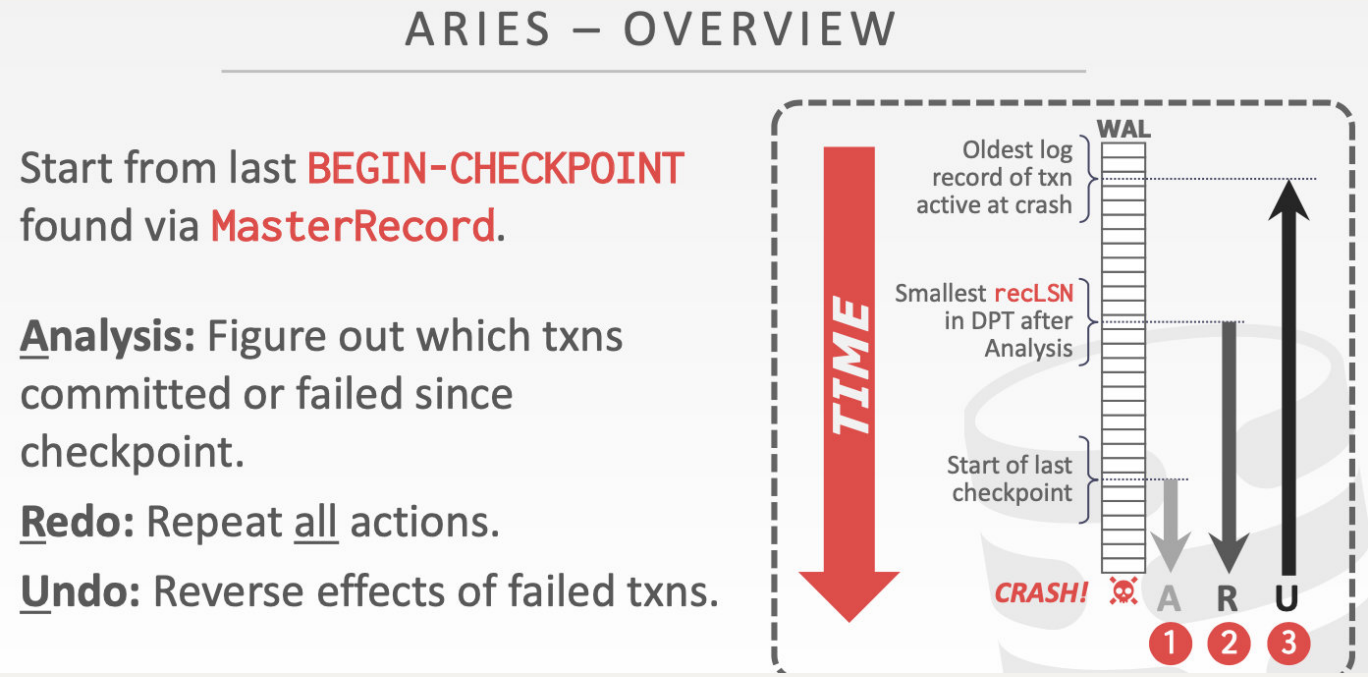
  （为了笔记的准确性，在接下来的内容里会把Andy老师的一些英文原话记下来）

- Redo Phase: In the redo phase you're jump to some appropriate location in the

log where you know there could be potential changes from transactions that did not make it safely to disk, and then you can strat reapplying those changes until you get to the end of the log. And you're gonna do this for every transaction you see, even ones that you know are gonna end up aborting.

- Undo Phase： In the undo phase you are gonna go back in reverse order, from the end of log uptil some point to reverse any changes from transactions that did not commit. And when the undo phase is done, then the database is now in a state that existed at the moment of crash with no partial updates from abortive transactions and all changes from committed transactions have been applied to disk.

结合具体例子理解ARIES的流程：



接下来详细分析Analysis阶段

ANALYSIS PHASE

Scan log forward from last successful checkpoint.

If you find a TXN-END record, remove its corresponding txn from **ATT**.

All other records:

→ Add txn to **ATT** with status UNDO.

→ On commit, change txn status to COMMIT.

For UPDATE records:

→ If page P not in **DPT**, add P to **DPT**, set its recLSN=LSN.

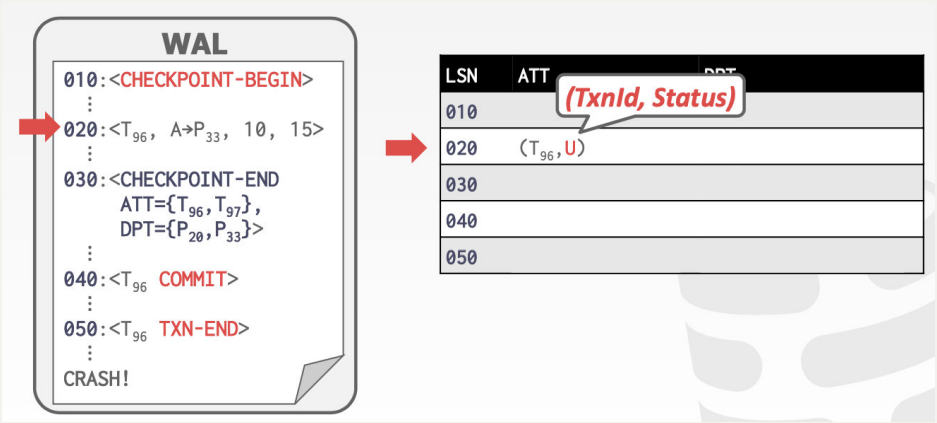
ANALYSIS PHASE

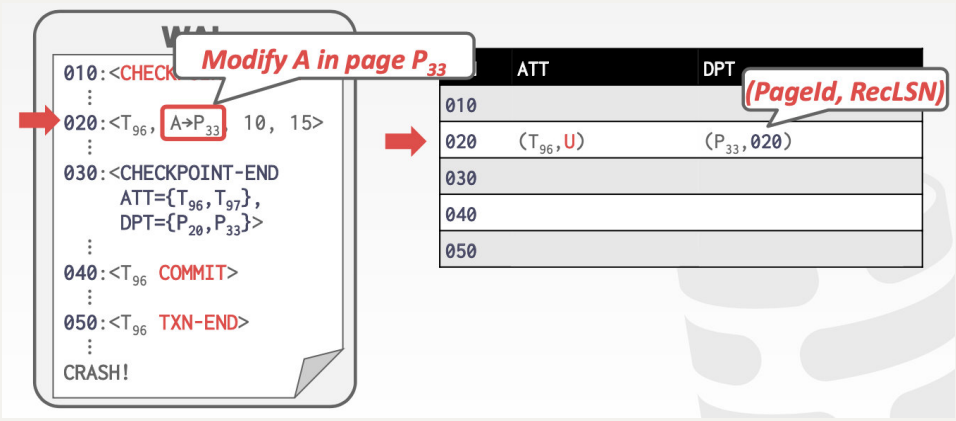
At end of the Analysis Phase:

→ **ATT** identifies which txns were active at time of crash.

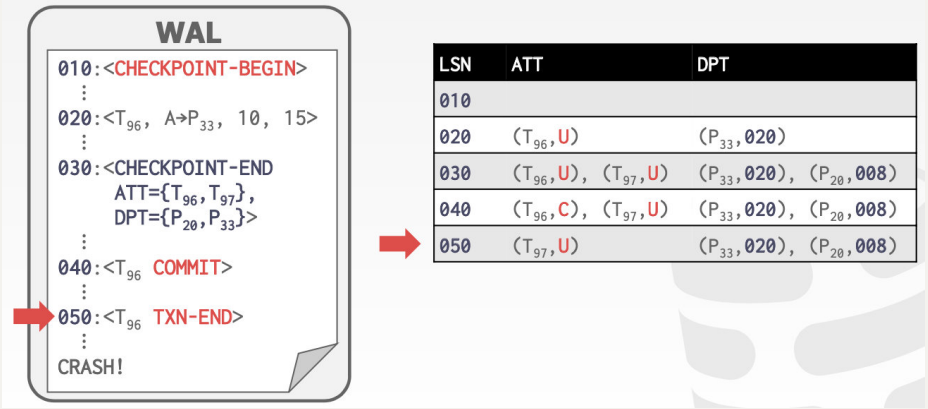
→ **DPT** identifies which dirty pages might not have made it to disk.

结合slides中的场景理解：





->next->...->



最终还原了DBMS宕机时的场景（推断出了有哪些事务未能完全提交，当时缓存池里有哪些可能未落盘的脏页）

然后就是redo阶段

The goal is to repeat history to reconstruct state at the moment of the crash:
→ Reapply all updates (even aborted txns!) and redo **CLRs**.

There are techniques that allow the DBMS to avoid unnecessary reads/writes, but we will ignore that in this lecture...

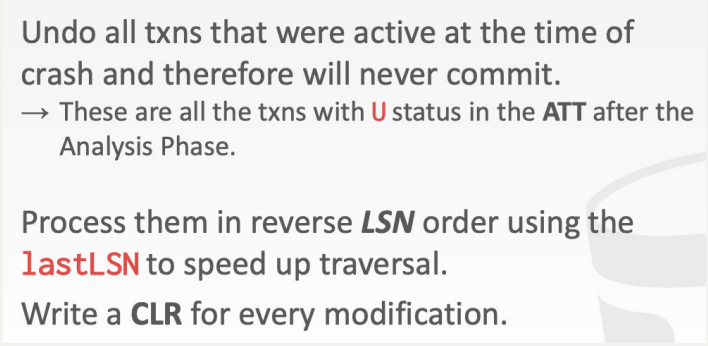
Scan forward from the log record containing smallest recLSN in **DPT**.

For each update log record or ***CLR*** with a given ***LSN***, redo the action unless:
→ Affected page is not in **DPT**, or
→ Affected page is in **DPT** but that record's ***LSN*** is less than the page's recLSN.

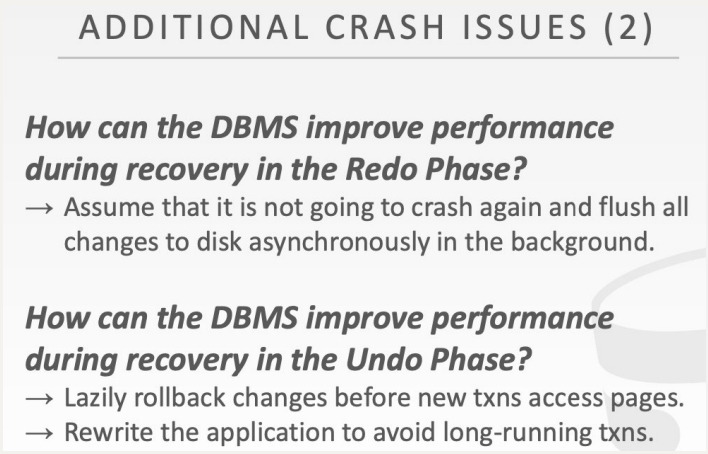
To redo an action:
→ Reapply logged action.
→ Set pageLSN to log record's ***LSN***.
→ No additional logging, no forced flushes!

At the end of Redo Phase, write TXN-END log records for all txns with status C and remove them from the **ATT**.

最后是undo阶段



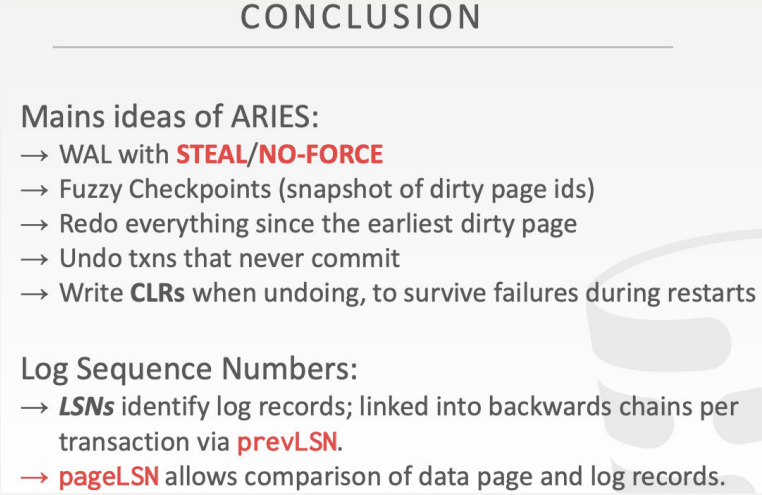
redo&undo阶段的流程可以结合slides里的例子理解（



有关于"Lazily rollback changes before new txns access pages." :

You do the analysis, you do the redo, then you figure out what you need to undo for every single page, but then rather than applying those changes, you just sort of keep them around somewhere in memory and then anytime a new transaction comes along and it wants to read that page, then you go ahead and apply the log. If you have a large database and for the undo phase, you only modify a small portion of it, rather than blocking access to the entire database while you undo this small number of pages, you merely come back right away and let everybody read whatever they want. It's just you block them when they try to read things you haven't rolled back yet.

（这和lazy allocation/copy on write都很像...）



Ref/参考自：

https://www.youtube.com/watch?v=4VGkRXVM5fk&list=PLSE8ODhjZXjbohkNBWQs_otTrBTrjyohi&index=21

https://zhuanlan.zhihu.com/p/482491814

https://zhenghe.gitbook.io/open-courses/cmu-15-445-645-database-systems/database-recovery

https://www.bilibili.com/video/BV1Rr4y1Y75X/?spm_id_from=333.788

https://www.youtube.com/watch?v=0wocj5Tp5dM&list=PLSE8ODhjZXjZaHA6QcxDfJ0SIWBzQFKEG&index=20&t=1799s