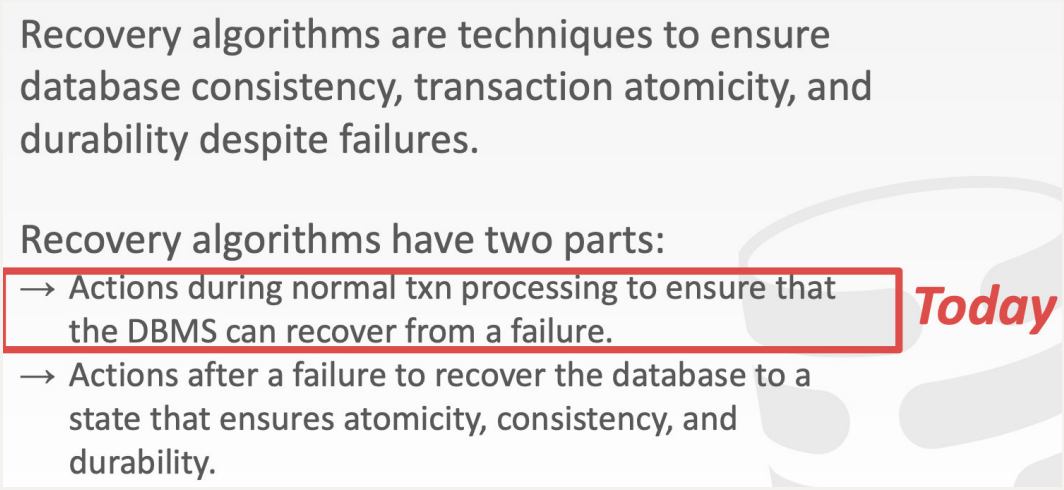
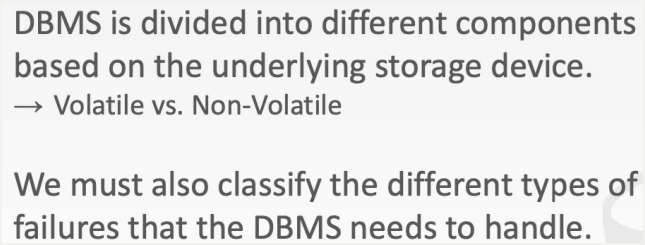
本Lecture和下一个Lecture将研究数据库的恢复



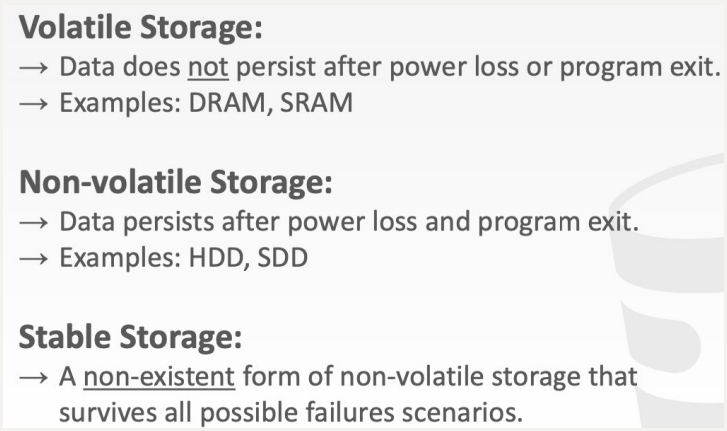
Failure Classification

故障的级别

DBMS的不同组件依赖于不同的存储设备，比如说缓存池是放在RAM这样的易失性存储上的，数据库里的数据本身是在硬盘这样的非易失性存储器上。

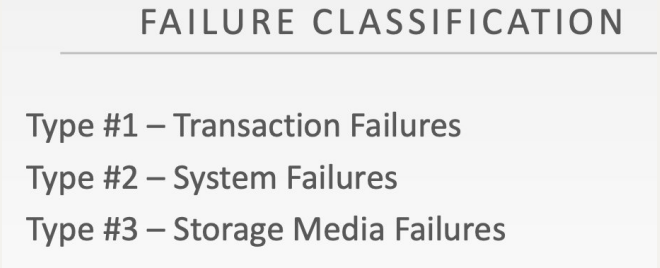


存储设备有如下的分类



Stable Storage是一种理想中的存储器：永远不会坏的非易失性存储，只存在于分析问题时的假设中

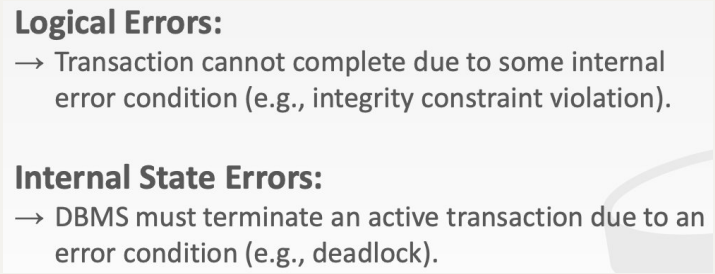
DBMS中的故障分为如下三级：



- 第一级，事务级别的故障

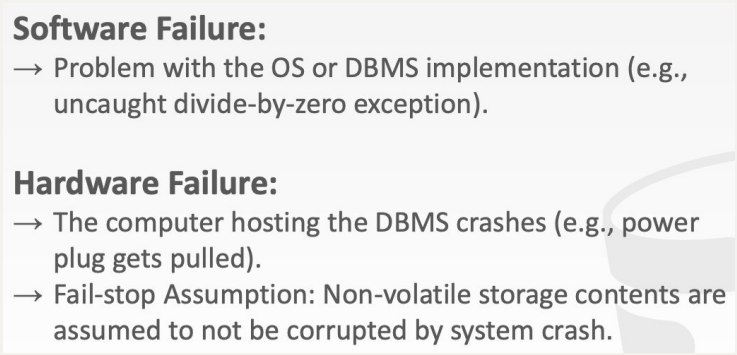
  这又分为逻辑错误（比如说用户要求回滚或者OCC的校验阶段发现不可串行而被迫回滚）和内部状态错误（比如说两个事务之间构成死锁，不得不把其中的一个回滚）

  事务级别的故障是数据库正常运行所不可避免的，因此这也是数据库开发者所必须考虑的

- 第二级，系统级的故障

  这又分为软件异常和硬件异常，软件异常可能是DBMS或者OS的bug，硬件故障可能是断电，CPU烧坏了这种，但不包括硬盘故障，硬盘的故障属于第三级故障。

- 第三级，存储介质的故障

  这类故障一般是无法修复的，数据库开发者无需考虑这些

Non-Repairable Hardware Failure:

→ A head crash or similar disk failure destroys all or part of non-volatile storage.

→ Destruction is assumed to be detectable (e.g., disk controller use checksums to detect failures).

No DBMS can recover from this! Database must be restored from archived version.

Buffer Pool Policies

DBMS需要特定的缓存池管理方案来实现如下的Undo/Redo操作，从而保证事务的原子性和持久化

Undo: The process of removing the effects of an incomplete or aborted txn.

Redo: The process of re-instating the effects of a committed txn for durability.

How the DBMS supports this functionality depends on how it manages the buffer pool...

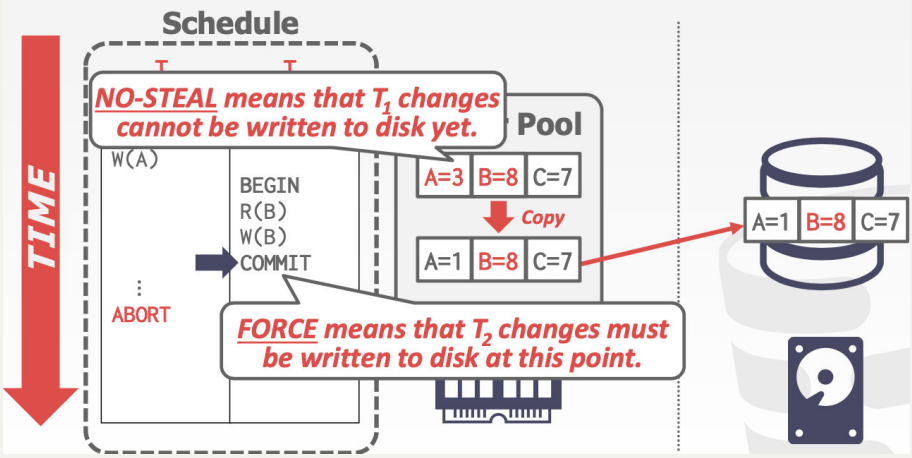
如果DBMS要求事务在commit的时候必须把它所做的更新写入磁盘，这就对应的是Force策略，否则就是No-Force策略（Steal/No-Steal参考slides即可）

Whether the DBMS requires that all updates made by a txn are reflected on non-volatile storage before the txn can commit.

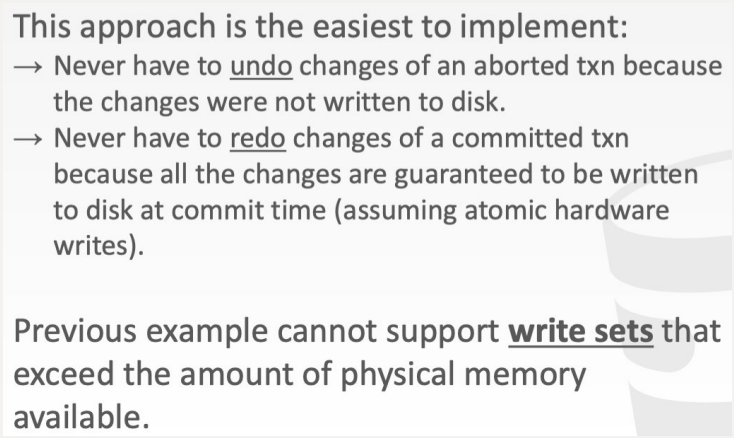
FORCE: Is required.

NO-FORCE: Is not required.

No-Steal+Force策略下，在如下的例子里，T2会申请一个新的页，这个页里只包含T2自己的数据更新，之后把这个页写入磁盘



No-Steal+Force这个策略的优点是它很容易实现，回滚时不需要做undo操作，因为被回滚的事务所做的更新没有被其他已经提交了的事务所连带着写入硬盘，只需把缓存池改回到原来的状态即可；重启恢复的时候也不需要做redo操作，因为事务commit的时候就已经完成了持久化



这个策略也有一定的问题：

- 刷盘操作过于频繁，性能不佳：每次commit的时候都会有硬盘I/O，导致用户也要随之等待；
- 事务在对数据库进行更新的过程中所读写过的所有东西在其提交之前都要一直暂存到缓存池里，否则如果有一部分被修改过的页提前被踢出缓存池，那么就破坏了事务的原子性。特别是全表扫描然后更新的这种场景下，缓存池就会非常紧张。也就是说，每个事务可修改的数据的量严重受缓存池大小限制（xv6的日志系统中也存在这样的问题）

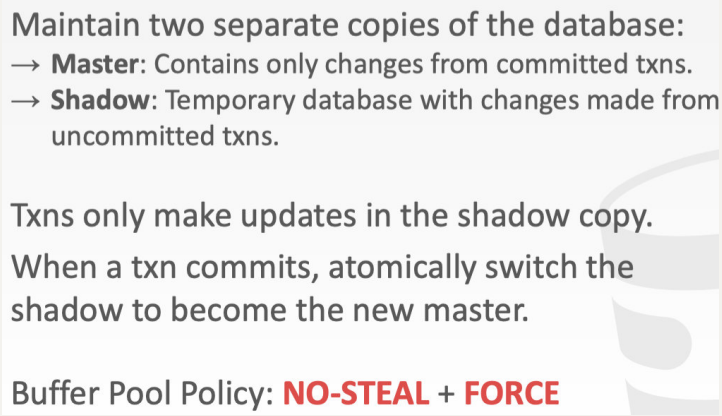
Shadow Paging

Shadow Paging是上文No-Steal+Force策略的一个具体实现

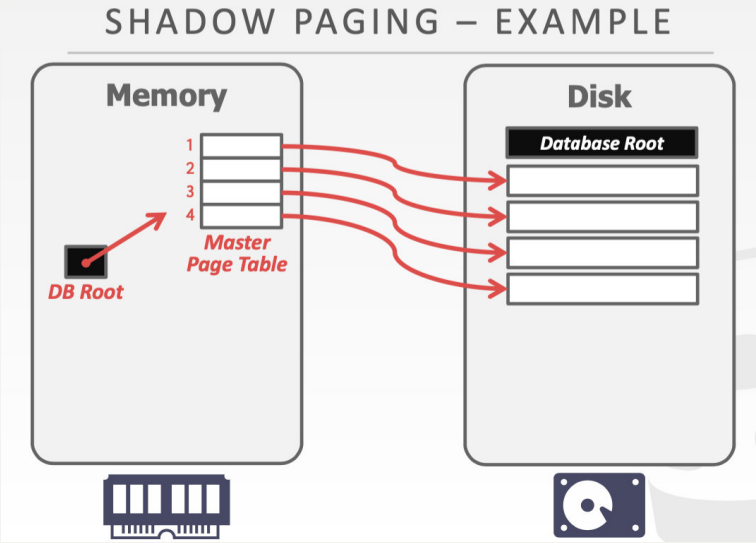
它会维护两份数据库数据：

- Master：包含所有已经提交事务的数据
- Shadow：在 Master 之上增加未提交事务的数据变动

正在执行的事务都只将修改的数据写到 shadow copy 中，当事务提交时，再原子地把 shadow copy 修改成新的 master。为了提高效率，DBMS 不会复制整个数据库，只需要复制有变动的部分即可。



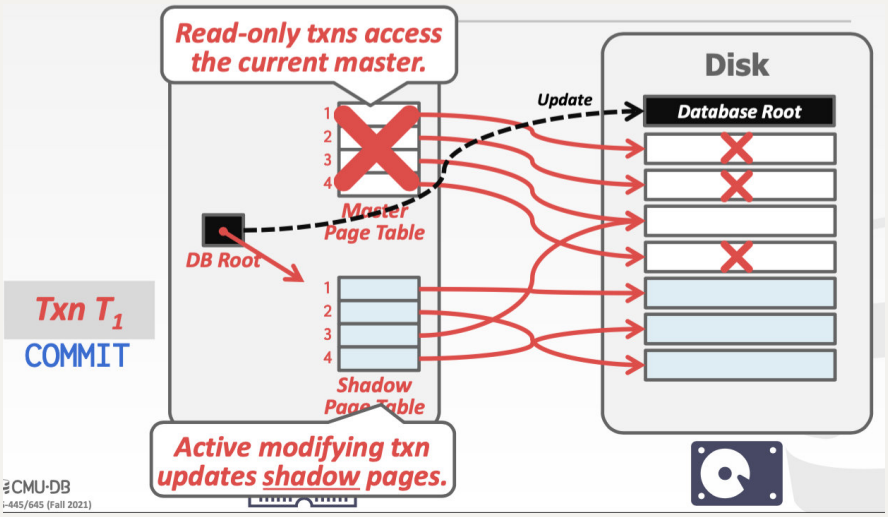
具体场景如下所示，DB Root这个指针指向缓存池中当前对其他事务visible的数据页，在写事务执行时，会生成一个 shadow page table，并复制需要修改的数据页，在其上完成相应操作。在提交写事务时，DBMS 将 DB Root 的指针指向 shadow page table，并将对应的数据页全部落盘。



To install the updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow:

→ Before overwriting the root, none of the txn's updates are part of the disk-resident database

→ After overwriting the root, all the txn's updates are part of the disk-resident database.



SHADOW PAGING – DISADVANTAGES

Copying the entire page table is expensive:

→ Use a page table structured like a B+tree.

→ No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes.

Commit overhead is high:

→ Flush every updated page, page table, and root.

→ Data gets fragmented.

→ Need garbage collection.

→ Only supports one writer txn at a time or txns in a batch.

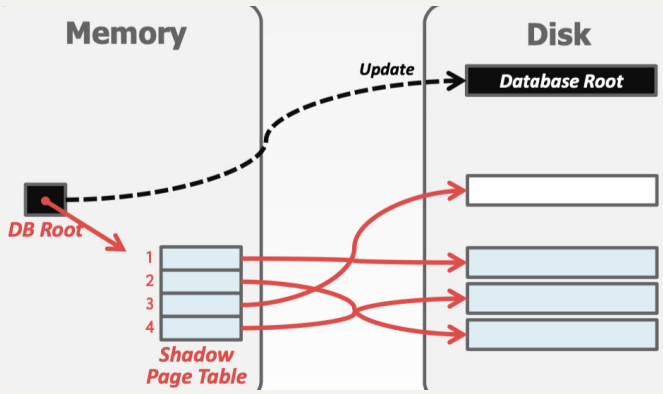
Shadow Paging相比于纯粹的no-steal+force策略的好处在于，新的shadow page在事务执行的过程中可以被刷到磁盘中，因为会刷到磁盘里新分配的页，其他事务看不到这些页，直到这个事务提交之后。也正是因此，在这个机制下可以"install changes that are bigger than the buffer pool size"。

Shadow Paging也有一定的问题：

磁盘I/O次数增多，因为在事务进行的过程中就要往磁盘里写数据。

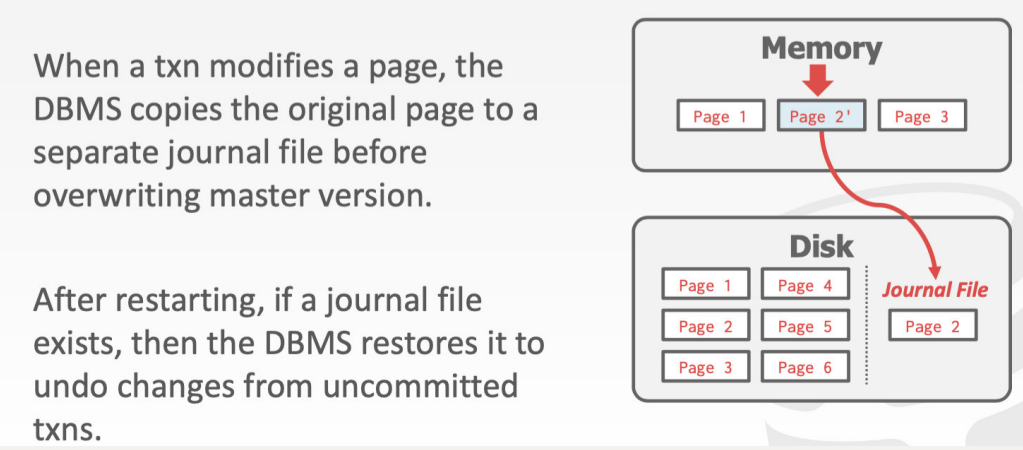
在工业界实现当中，一个shadow page里面可能有成百上千条的数据（MySQL的一个页是16KB），如果我们只修改数据库文件某个页里的一条数据的话，那么还是要对这整个页做一个拷贝，因此就会有很大的冗余，拷贝的开销不小。

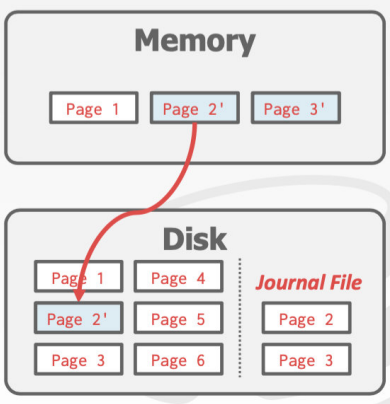
而且在commit时所做的事情太多：把shadow page都刷入磁盘，修改DB Root，对之前的内存和磁盘里的旧的页做垃圾清理。而且这容易造成数据的碎片化：清理掉了很多旧页之后，磁盘上出现了外部碎片/空穴（如下所示），数据的存储可能就因此变得不连续



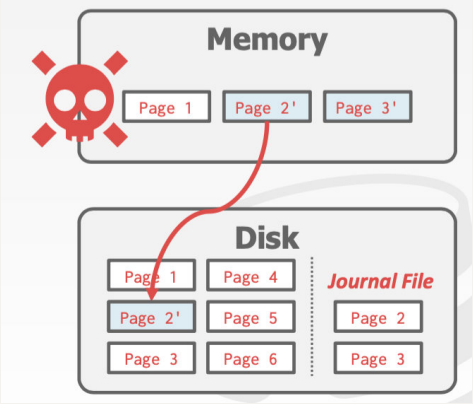
SQLite曾经使用的就是shadow paging的策略，但其实是改进后的

硬盘上journal file里存储的是事务要修改的页的原始版本，commit时会把缓存池里修改后的页刷入磁盘里对应的地方（非jouranl file区）

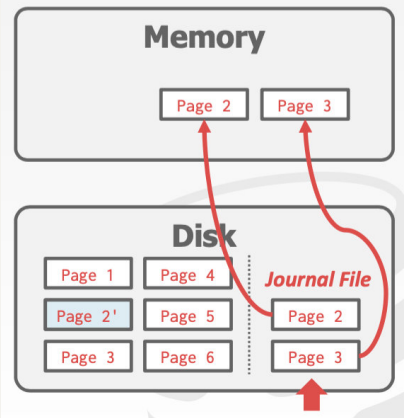




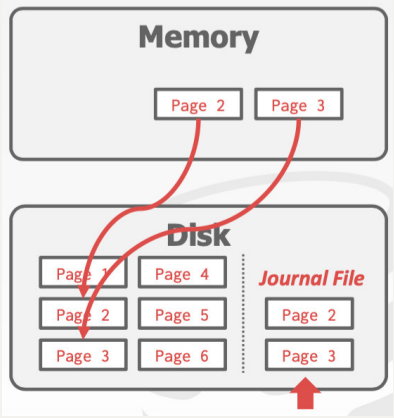
如果刷入Page 2'之后停电了，机器重启后会把journal file加载入缓存，然后把它写入磁盘，这就完成了对断电时未提交的事务的回滚，保证了事务的原子性



->next->



->next->



Shadowing page requires the DBMS to perform writes to random non-contiguous pages on disk.

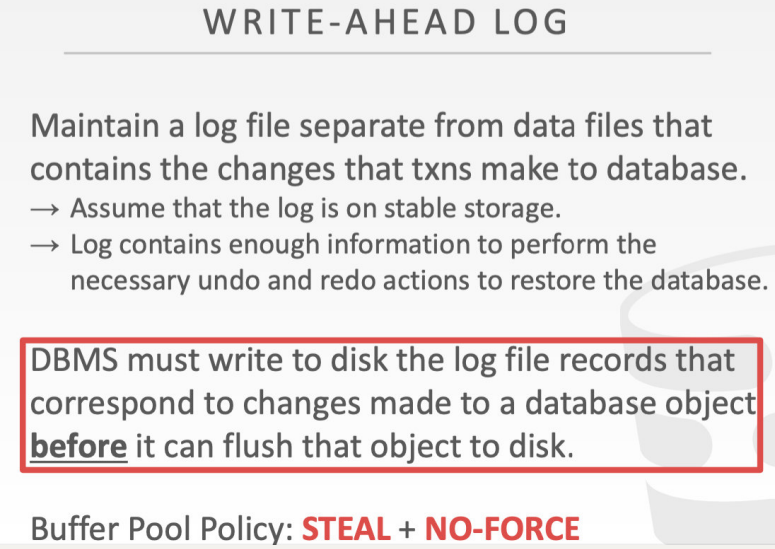
We need a way for the DBMS convert random writes into sequential writes.

Shadow Paging策略会造成对硬盘的大量的随机访问，这会降低性能，DBMS需要一个方案去把对磁盘的随机的写转换成连续的写，这就是下面要介绍的WAL（Write-Ahead Log）

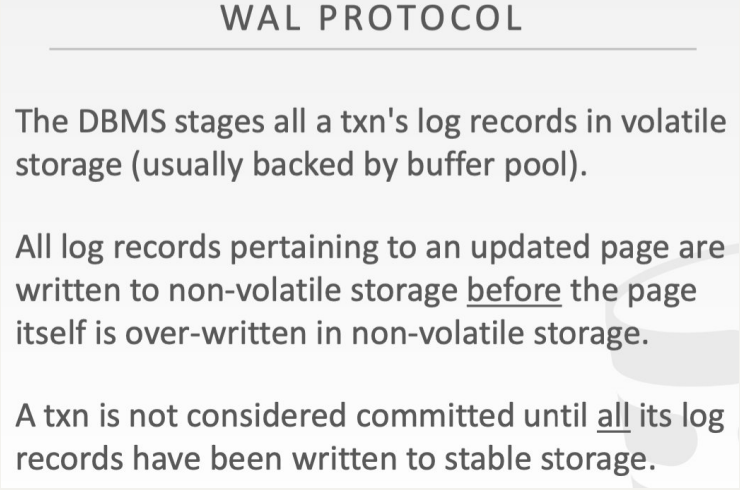
Write-Ahead Log

WAL，翻译成中文是“预写日志”，也就是要先写日志，再写数据。在这个策略里，磁盘里面会单独开辟一块类似于前面所讲的journal file的区域，称为log file，用于保存事务对于数据的修改。为了便于分析问题，我们假设日志文件在可靠的不容易坏的存储设备上存储，日志里面有足够多的信息让DBMS完成Undo/Redo操作

DBMS在把用户所修改过的缓存池中的页刷入磁盘之前，它需要先把预写日志写入磁盘中的日志文件，也就是说先完成预写日志的持久化，再完成缓存池里dirty page的持久化

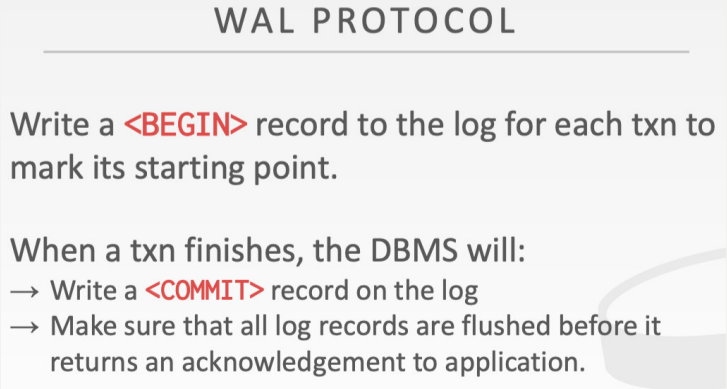


WAL协议的具体内容如下：

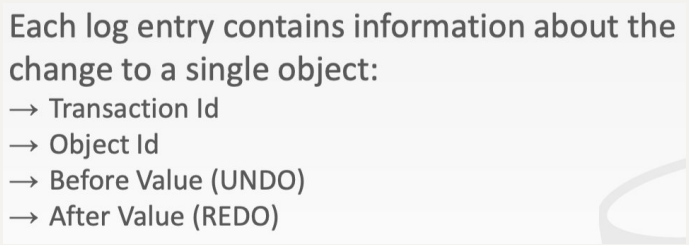


修改数据库文件的数据页的时候，自然要把修改先写入缓存池，日志也会先被写在内存里，内存里一般会开辟一个专用的缓存专门用来写日志。在将数据页落盘前，所有的日志记录必须先落盘，在日志落盘之后，事务就被认为已经提交成功。

日志中往往会插入如下的记录：事务刚开始的时候会写一个begin record，事务提交的时候会写一个commit record，这两个record之间记录的就是事务所做的全部操作，只有当日志里的记录全部被刷入硬盘，才可以把“提交成功”的消息返回给上层的应用程序或者用户



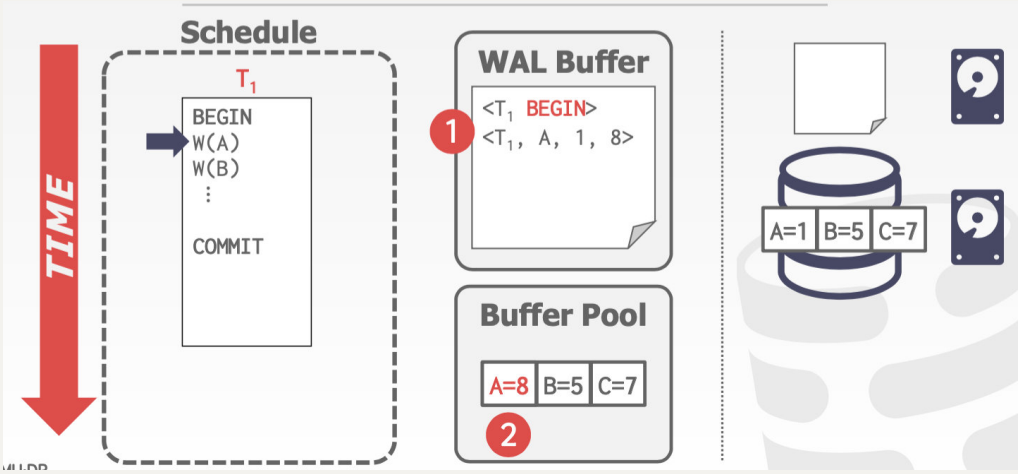
WAL策略下，日志的每一条entry包含如下的内容：



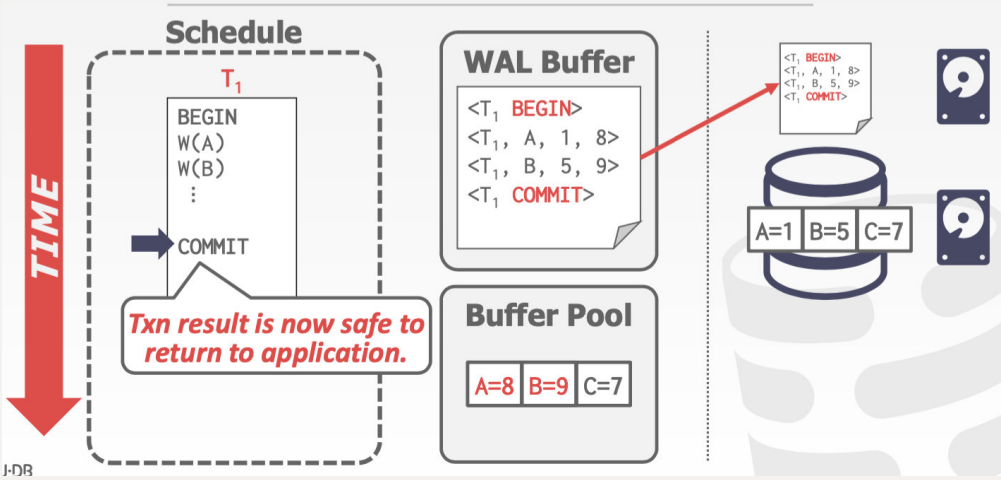
object是被修改的数据库中的对象的ID，before value就是这个对象的数据被修改之前的值，它是Undo操作的时候要用到的；after value就是这个对象的数据被修改后的值，它是Redo操作的时候要用到的

MySQL的write ahead log被分成了两部分，分为undo log和redo log，当然在很多工业界DBMS的实现中，还是把这两种log合并成了一个log，去集中管理全部的日志。MySQL把WAL分成undo/redo log的好处是 undo log可以单独拿出来用来做delta storage（这属于MVCC相关的内容），回推出来数据的上一个版本，从而实现MVCC，并且把undo/redo log分开存储也更安全。MySQL中，undo log和数据库中的数据本身被存储在一个文件里（不和数据库的数据在同一个页里），redo log是被存储在单独的文件中的。

结合如下例子理解WAL：



->next->...->

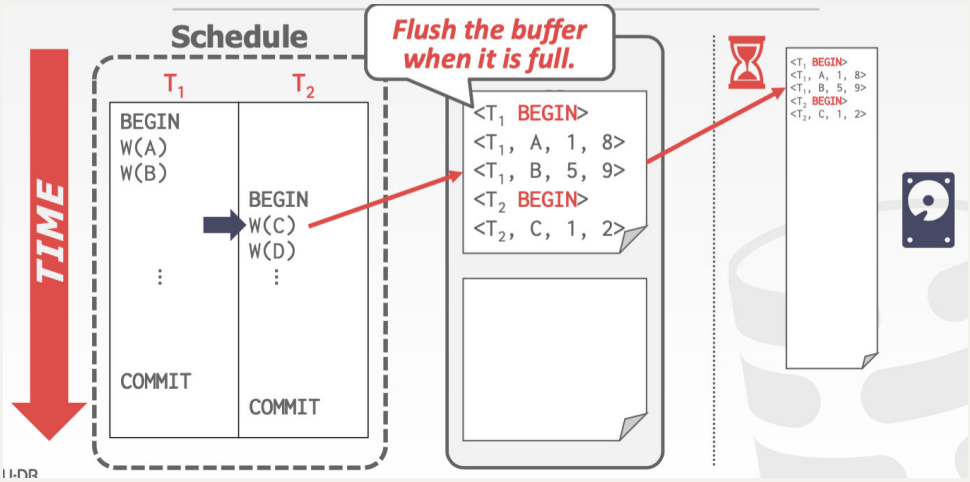


WAL协议规定要在事务commit的时候把它的日志写入硬盘，在实际的DBMS实现当中，有一个优化-group commit，“组提交”：在数据库系统中，磁盘I/O的开销是非常巨大的，如果每提交一个事务就要进行一系列的磁盘I/O，那么性能不是太好。不妨让用户多等待一会儿，把多个事务的日志攒到一起提交（第一个到达的事务执行到commit时先阻塞住，然后多攒几个要commit的事务，之后一次性地把它们的日志全部刷入磁盘，然后同时将“事务成功提交”返回给所有等待着的用户），这样的话就会把多个事务的磁盘I/O合并到一起来处理，减少磁盘I/O的次数与开销。（xv6的日志系统就是使用的组提交，一次性地提交多个和

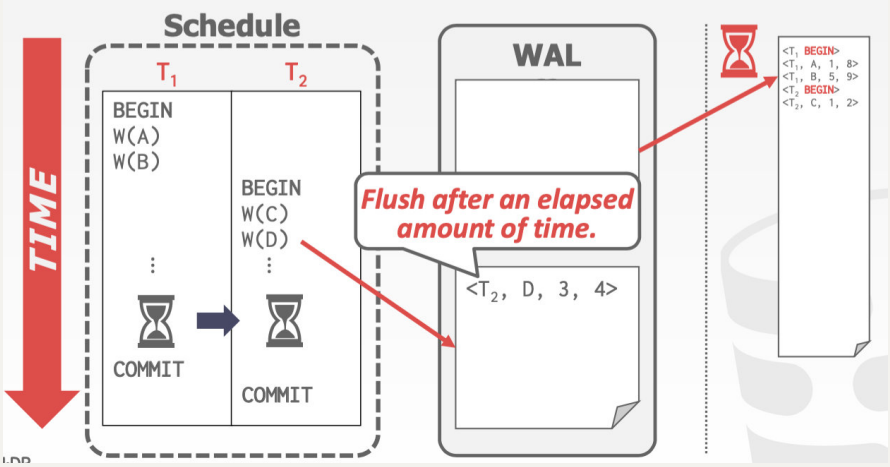
文件系统有关的系统调用的事务日志，并且在执行相关的系统调用而陷入内核时先做检查，检查当前攒批的所有事务所要写的日志的大小是否已经超过日志容量的上限，如果没有的话就可以执行本次系统调用，否则就要等待这波攒批提交完之后再继续执行系统调用）

结合如下的例子理解group commit:

日志文件可以有很多个页，如果日志的buffer满了的话可以让后台的线程把部分日志页写入磁盘，这不会太影响当前正在执行的事务的性能



如果某段时间内DBMS中的事务数量不多，很长的一段时间内都没有新的事务到达，那么也没有必要等到事务攒够一定的数量再group commit，不然会导致等待的时间过长。可以设置一个等待时长上限，到了这个上限之后就把当前所有想要commit的事务的日志写入硬盘。



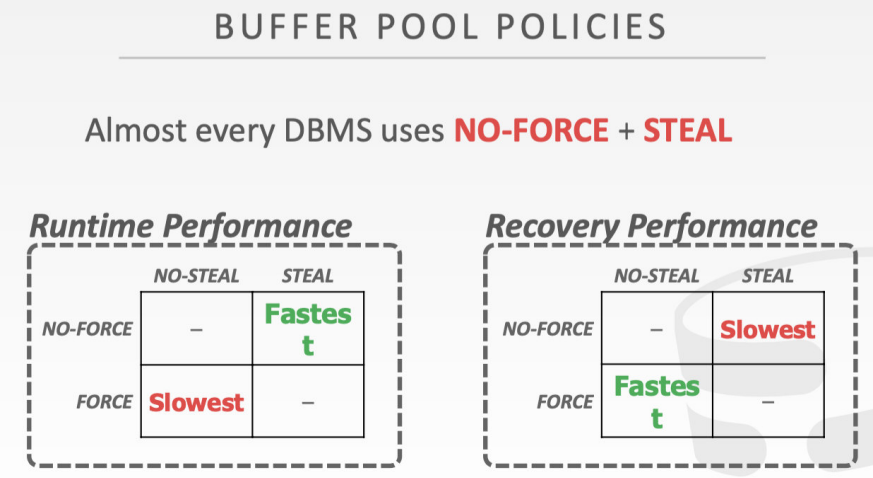
前面讨论的都是和日志落盘有关的问题，那么在WAL策略的实现当中，什么时候会完成脏数据的落盘呢?

When should the DBMS write dirty records to disk?

→ Every time the txn executes an update?

→ Once when the txn commits?

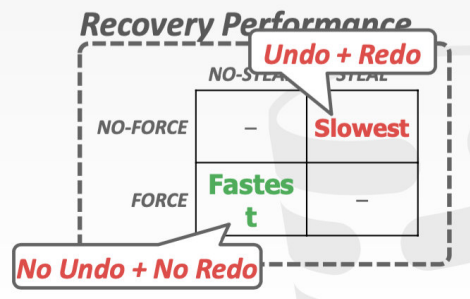
这与缓存池的策略有关，各种缓存池策略下的性能对比如下所示



runtime performance是事务正常执行的时候的性能，no-force+steal策略的runtime performance是最好的，因为steal策略会把多次的刷dirty page回磁盘的I/O合并成一次，no-force策略下没有强制要求用户在commit的时候刷盘；与之相反，force+no-steal的runtime performance最差。

recover performance是数据库恢复的时候的性能，no-force+steal策略下的recover performance最差，因为no-force策略会导致宕机后重启时用log重放事务（因为事务commit的时候没有立即把dirty page刷回磁盘），steal策略会导致往磁盘里刷了好多没有还没有提交的事务所做的数据更新，因此DBMS需要执行undo操作把crash的时候没执行完的

事务回滚。

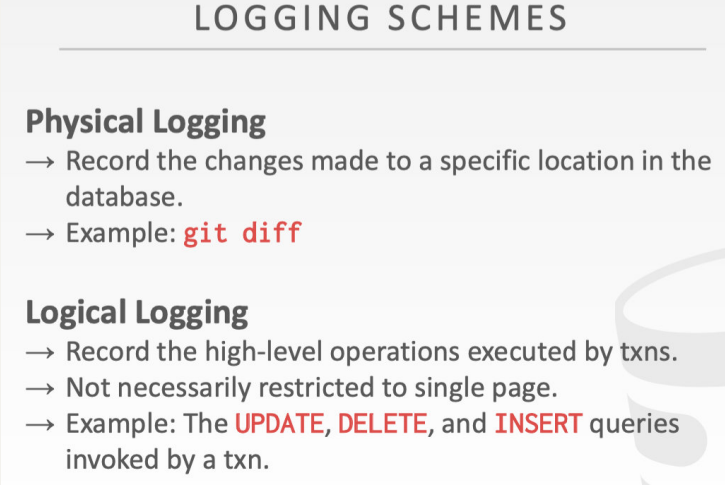


但是DBMS的开发者们更在意它的运行性能，进而选择no-force+steal的缓存池策略，因为crash后恢复的场景是相对比较罕见的，可能DBMS运行几年，几个月才有一次宕机恢复的情况发生。

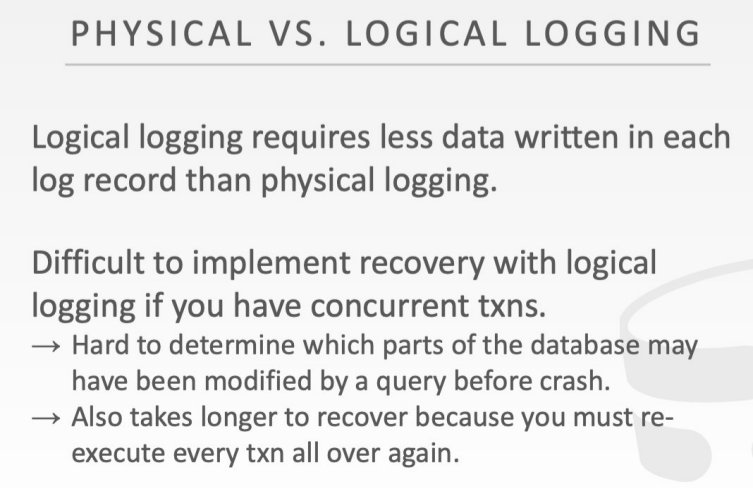
Logging Schemes

接下来将深入分析WAL（write ahead log）的格式，也就是WAL所记录的具体内容

WAL的具体内容有如下两种：



- 物理日志：记录每一个页具体到二进制级别的变化（e.g. 某个页的xxx偏移量位置处发生了xxx变化）
- 逻辑日志：记录事务所执行的操作，比如说事务所执行的SQL语句



物理日志的缺点是，如果事务所执行的操作是对很多很多的数据进行修改（比如说修改全表的某个attribute的值），那么物理日志将会非常庞大，如果用逻辑日志来记录，要记录的可能仅仅是一条SQL语句

正如前面所说，逻辑日志的优点是所记录的数据的量比物理日志要少。但逻辑日志也有其缺点：如果逻辑日志是记录事务所执行的SQL语句，那么如果事务执行了“记录数据库当前的时间”这样的SQL指令（可能会通过SQL的`now`函数来实现），在redo的时候就会出现问题。逻辑日志还有其他的问题，比如说，如果SQL语句当中含有`limit`，这是不会保证每次执行时吐出的数据集都一样的（比如说`limit100`，限制输出100个tuple，实际执行的时候被优化器安排做加塞扫描，或者还有可能是在MySQL这种的主备复制的情况下，备库中没有主库的索引，在主库上走主库的索引输出100个tuple，但备库上没有索引，走了主索引输出了100个tuple，它们输出的数据集合不一样）总结一下，也就是说，如果只采用逻辑日志，在redo重放日志的时候可能会重放出来错误的结果，或者说是出现逻辑问题。

并且在事务并发执行的情况下，逻辑日志很难实现一致的事务并发控制。由于逻辑日志难以携带并发执行顺序的信息，当同时有多个事务产生更新操作时，数据库内部会将这些操作调度为串行化序列执行，需要机制来保障每次回放操作的执行顺序与调度产生的顺序一致。

物理日志和逻辑日志都有各自的优缺点，因此也便有了结合它们各自的优点的混合式策略-*phsilogical logging*

PHYSIOLOGICAL LOGGING

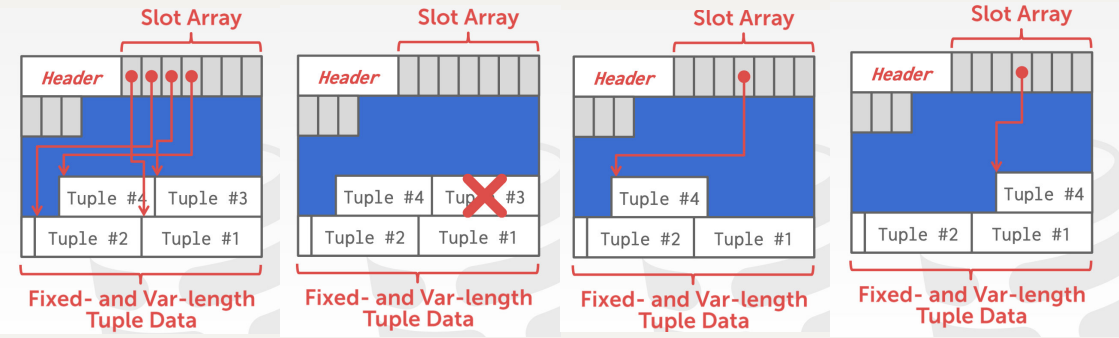
Hybrid approach where log records target a single page but do not specify organization of the page.

→ Identify tuples based on their slot number.

→ Allows DBMS to reorganize pages after a log record has been written to disk.

This is the most popular approach.

这个策略大体上和物理日志差不多，但记录的是某个页的第几号槽（slot）指向的tuple内的数据的变化而非在偏移量xxx处的变化，这就不会受存储引擎整理数据页中的tuple时造成的影响（如下所示，删除相关的SQL指令在页中删除tuple后导致空穴和碎片，因此DBMS会整理tuple进而清理碎片，在整理前后，同一个tuple在这个页中的偏移量会发生变化，因此就在清理碎片的同时需要维护物理日志，如果日志里记录的是slot号而非偏移量，那么就省去了这个维护操作带来的开销）

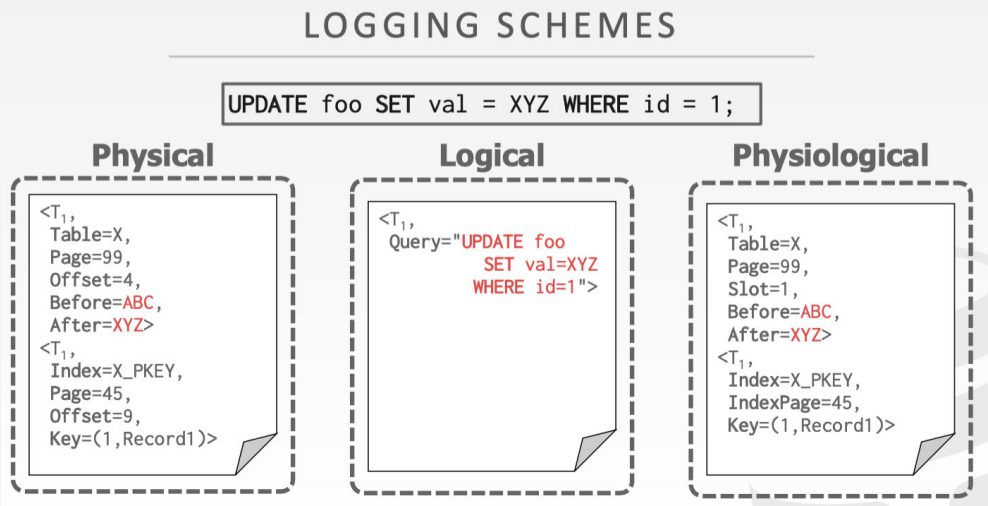


*phsilogical logging*是最为常见的WAL实现方式，MySQL官方文档中所说的物理日志也是这个（纯物理日志在工业界DBMS中不常见）

虽然但是，*phsilogical logging*还是没有彻底解决“访问的数据过多导致日志过于庞大”的问题，因此在此基础上还是有一些改善的策略：MySQL中的Mixed Logging，在这个策略中，如果DBMS检测到*phsilogical logging*日志太长，并且经过分析可以得出只记逻辑日志的话在redo时不会出现问题（也就是不涉及前面所提及的那些特殊情况），那么可以混合地记日志：一会儿记*phsilogical logging*日志，一会儿记逻辑日志，这样的话就更好地结合了各种日志策略的优点，但也在前面判断是否可以记混合日志这个阶段有一点额外的开销。

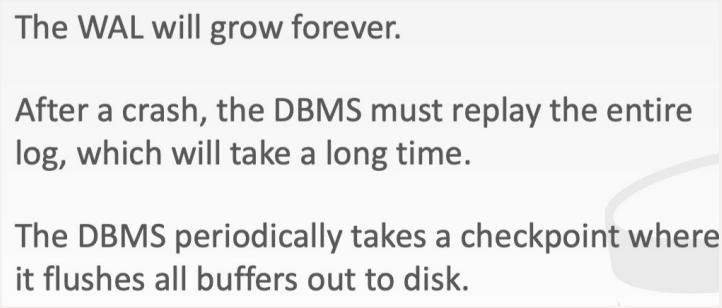
而且*phsilogical logging*相比于物理日志，在redo/回放时会多一些开销，因为需要根据slot号推断出具体的偏移量是多少，逻辑计划在redo的时候开销最大，因为需要把SQL语句经过查询优化器重新执行

如下所示的SQL语句会导致在表以及对应的索引结构里修改数据，与之对应的各种日志形式如下图所示（这里是把undo log和redo log写到一起了）：

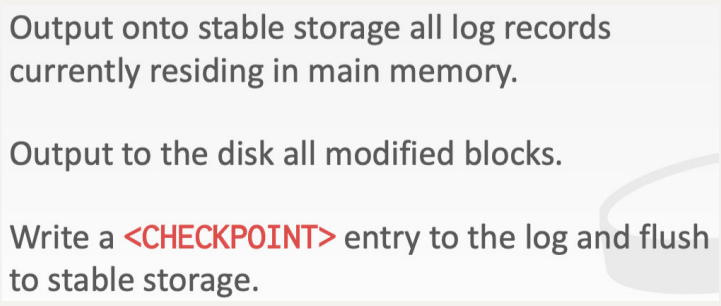


Checkpoints

这个和单机游戏中的“存档点”很类似

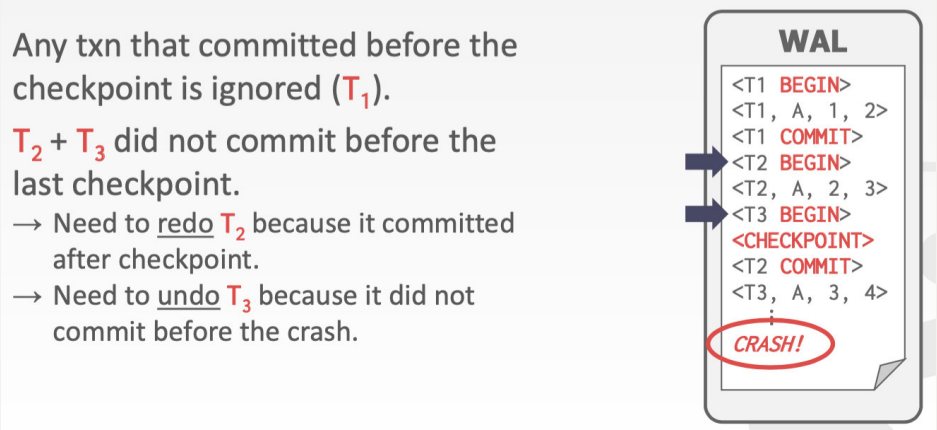


磁盘中的日志如果不清理的话它会越来越大，把磁盘占满。并且数据库崩溃的时候如果不知道该从哪个点开始恢复，就会造成类似于“玩游戏从来不存档”的后果。因此DBMS会在日志里面加一些存档点-'checkpoint'，有了存档点之后，可以把存档点之前的日志都删掉，并且崩溃恢复的时候，只需要回放存档点之后的日志即可

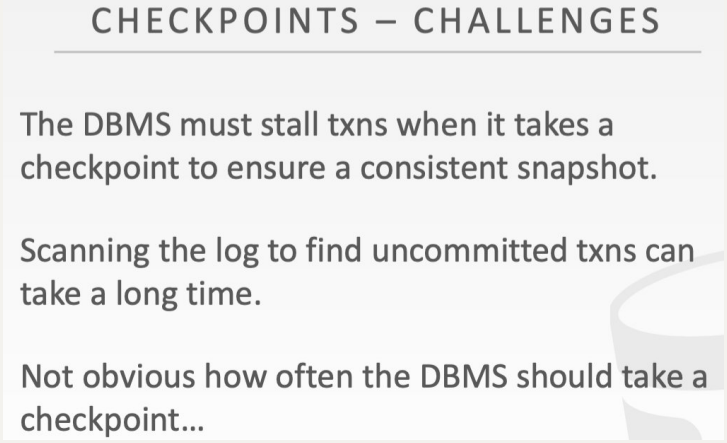


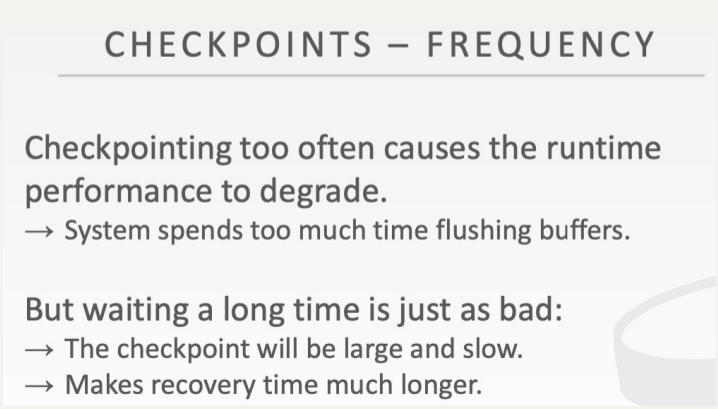
DBMS会定期地把全部的脏页和日志都被写入磁盘，然后在日志中打上checkpoint。也就是说，日志中checkpoint说明在它以上的所有东西已经全都被刷进了磁盘。

结合如下例子分析，在checkpoint之前就提交了的事务不用管。在checkpoint之前，T2和T3都没有commit，因此要对它们做redo或undo操作

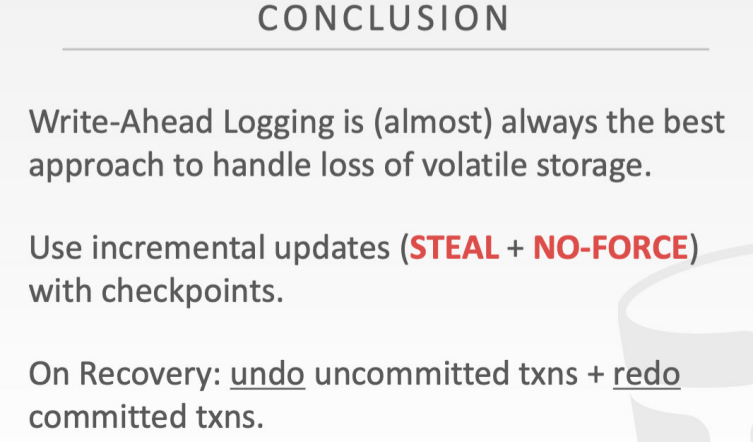


Checkpoint在实际实现当中存在一些问题:





Checkpoint本身肯定会带来性能损耗，存档需要花时间做磁盘I/O，事务的执行会被阻塞住。并且在数据库恢复的时候需要去根据日志判断哪些事务在crash前没有commit，进而需要扫描日志文件，因此也会带来开销。数据库多长时间存档一次也是一个玄学问题，如果存档过于频繁，性能会变差，如果存档频率极低，会导致每次存档时，有太多东西要写入磁盘，如果事务运行了一半的时候DBMS开始了存档，那么事务就会被阻塞很长时间。并且如果存档的频率过低，在数据库恢复的时候也比较麻烦，因为会有好多日志要回放。



总结一下，WAL之所以能让性能变好，是因为它会在日志文件里面进行连续的写，把日志写入磁盘之后就可以返回。WAL策略下一般会采用STEAL+NO-FORCE的缓存池策略，并且会通过checkpoints来提升数据库恢复的时候的性能

Reference:

https://zhuanlan.zhihu.com/p/482489302

https://www.bilibili.com/video/BV1yL411V7tS/?spm_id_from=pageDriver

https://www.youtube.com/watch?v=mD5znxu4Vq4&list=PLSE8ODhjZXjZaHA6QcxDfJ0SIWBzQFKEG&index=20

https://spongecaptain.cool/post/database/logicalandphicallog/