ARIES

事务恢复算法

1990s 由 IBM 开发

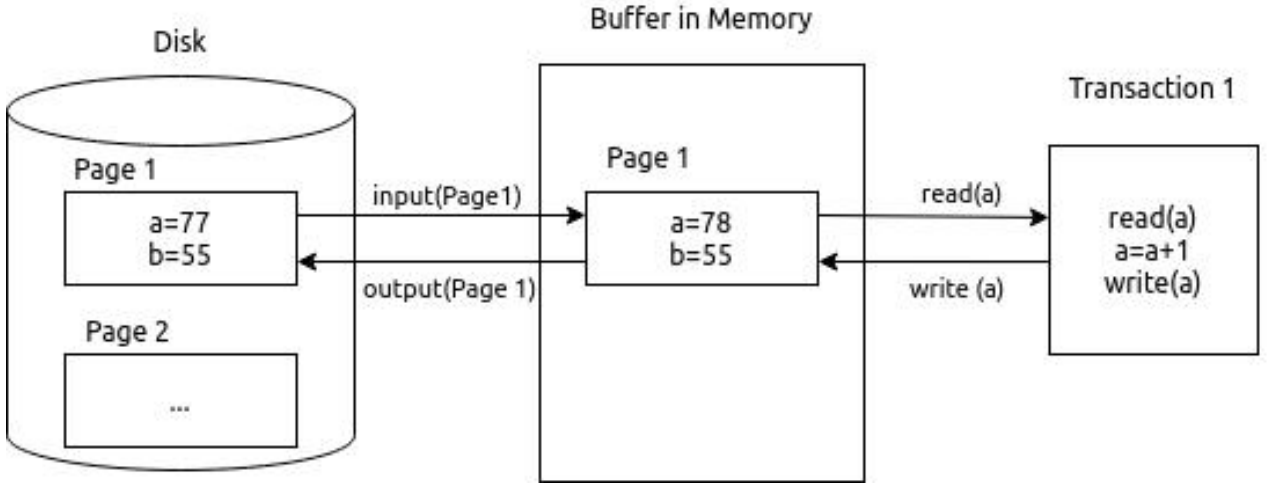
不是所有的系统都实现了论文中的 ARIES 算法，但是都是类似的

主要内容

- 页模型
- 基本思想
- 执行流程
- 模糊检查点
- 恢复流程

页模型

- 在页模型(Page Model)中，数据库包含一个有限数据页的集合，数据页不可分且不相交。
- 所有对数据的高层操作最终都要转化成为对数据页的读写操作。
- 事务被看成是对页面的一组操作步骤

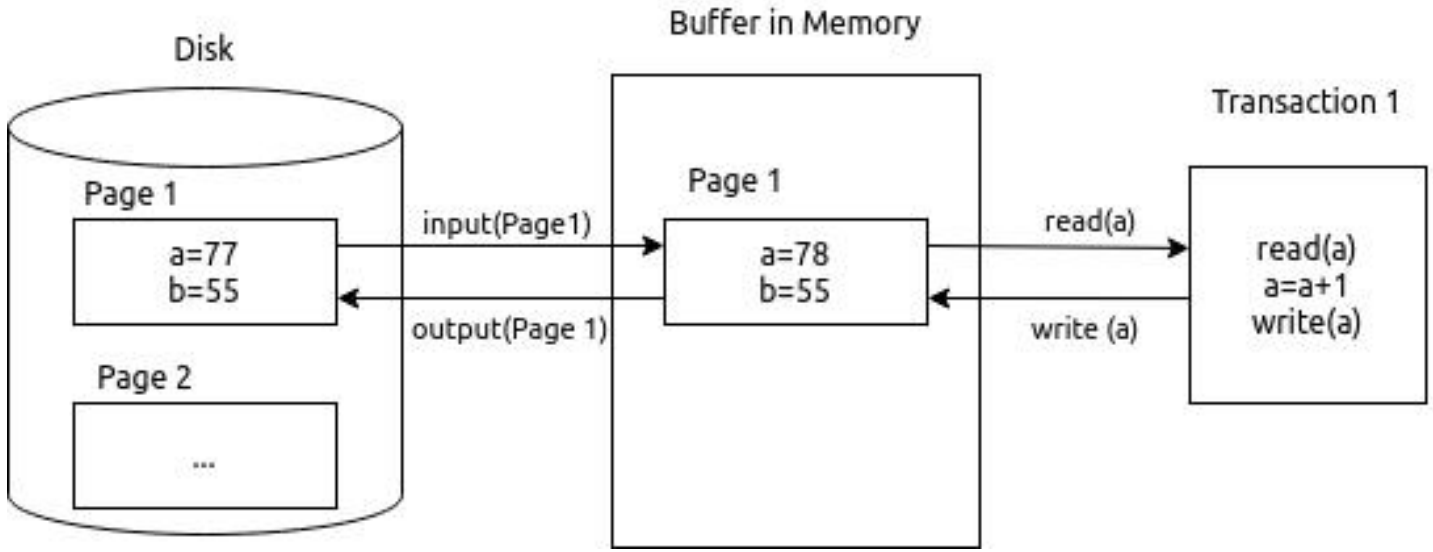


- 数据库系统常驻于非易失性的存储器（通常是磁盘），并以页为存储单位。
- 内存中有一个缓冲区用于临时存放磁盘上的页。
- 每个事务T拥有自己的私有工作区，事务通过在其工作区和缓冲区传送数据来和数据库系统进行交互。

页模型

如果事务系统崩溃：

- 内存缓冲区中的页面会丢失
- 活动的事务会失败



事务的特性：

- 原子性
- 一致性
- 隔离性
- 持久性

在系统重启的时候：

需要把数据恢复到一致的状态

主要保证事务的原子性和持久性

页模型

缓冲策略

- Force

  事务提交前，强制将它所做的修改都写回磁盘

- Steal

  事务提交前，就可以把它对页面的修改写回磁盘

ARIES支持Steal和No-Force的缓冲策略：

- 这给了Buffer Manager 最大的灵活性
- 增加了系统的复杂性
  - No-Force 导致系统崩溃时，已提交事务的修改可能还没有被写回磁盘
  - Steal 导致系统崩溃时，失败的事务可能已经把一些更新写入了磁盘

基本思想

预写日志 WAL

1. 对任何更新操作，先记录日志。将更新写回磁盘之前，必须先把日志写到稳定的存储器上
2. 一个事务的所有日志（包括commit日志）写入磁盘后才可以被提交

Redo阶段重放历史

系统重启的时候，ARIES重放历史(Repeat History)，将系统带到崩溃时刻的状态，然后undo那些崩溃时仍然活动的事务（它们失败了）

记录Undo时所做的修改

撤销事务所做的操作也会被记录在日志，用来保证重复崩溃下的undo操作不会被重复执行

执行流程

- WAL
- 事务 Commit
- 事务 Abort

WAL

ARIES算法为每个日志分配一个全局唯一的编号Log Sequence Number (LSN)。系统中不同的组件跟踪了与它们相关的LSN。

- PageLSN

  位置：每个页面（磁盘上的物理页面和内存缓冲区中的页面）

  作用：PageLSN_X 是对页面Page_X最后的更新，类似于版本号。每当有更新操作发生在该页面上时，就将PageLSN更新为这条日志的LSN

- FlushedLSN

  位置：内存

  作用：FlushedLSN 记录被写入磁盘的最后一条日志的LSN

WAL

- LSN

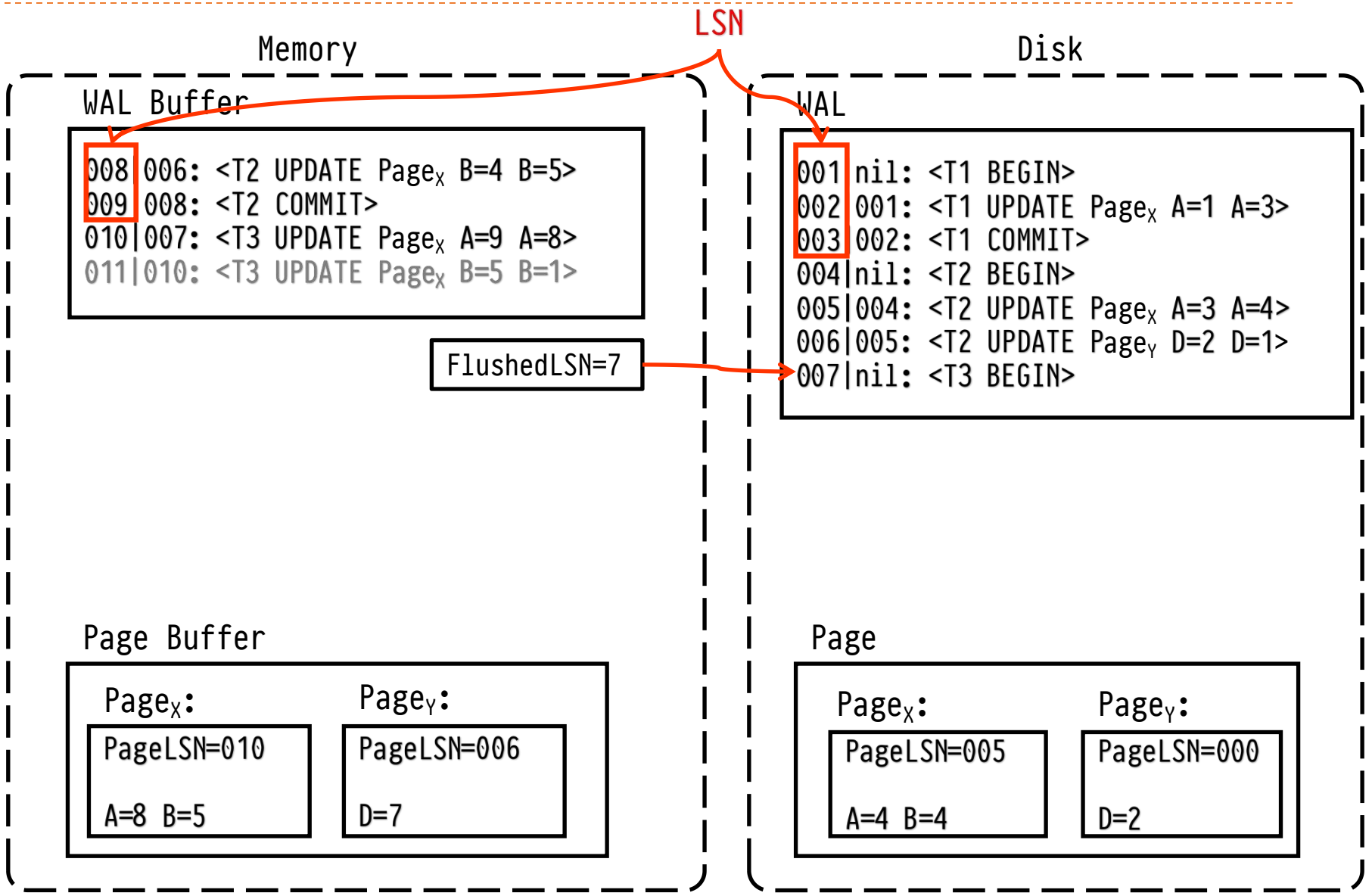
  日志顺序号

- PageLSN

  对页面的最后更新

- FlushedLSN

  被写入磁盘的最后一条日志



WAL

- 将脏页面X写回磁盘前，必须满足：

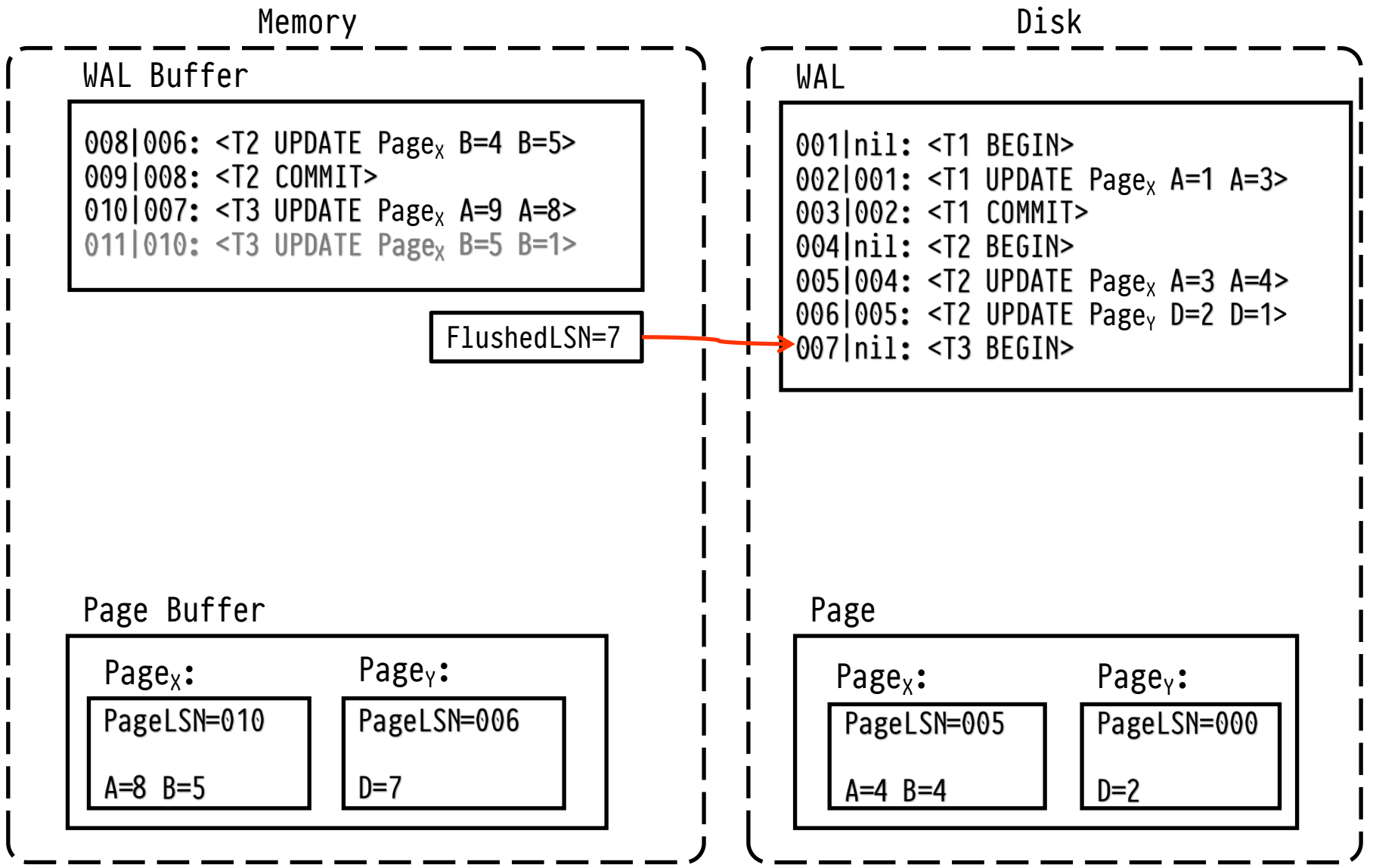
pageLSN x ⩽ FlushedLSN

- 图中的状态下：

  页面X不能被写回磁盘

  页面Y可以被写回磁盘

- 写数据随机IO转换为了写日志顺序IO和写回累计更新的脏页面



事务提交

事务提交流程

1. 事务T执行过程中，对于每一个更新操作：
   写下一条更新日志记录
   更新缓冲区相应的数据
2. 事务T准备提交，写一条Commit日志
3. 当FlushedLSN >= Commit日志的LSN，事务提交成功
4. 为事务T写一条TXN-END日志

事务提交

更新日志记录

事务对数据库进行任何修改之前，先写更新日志记录

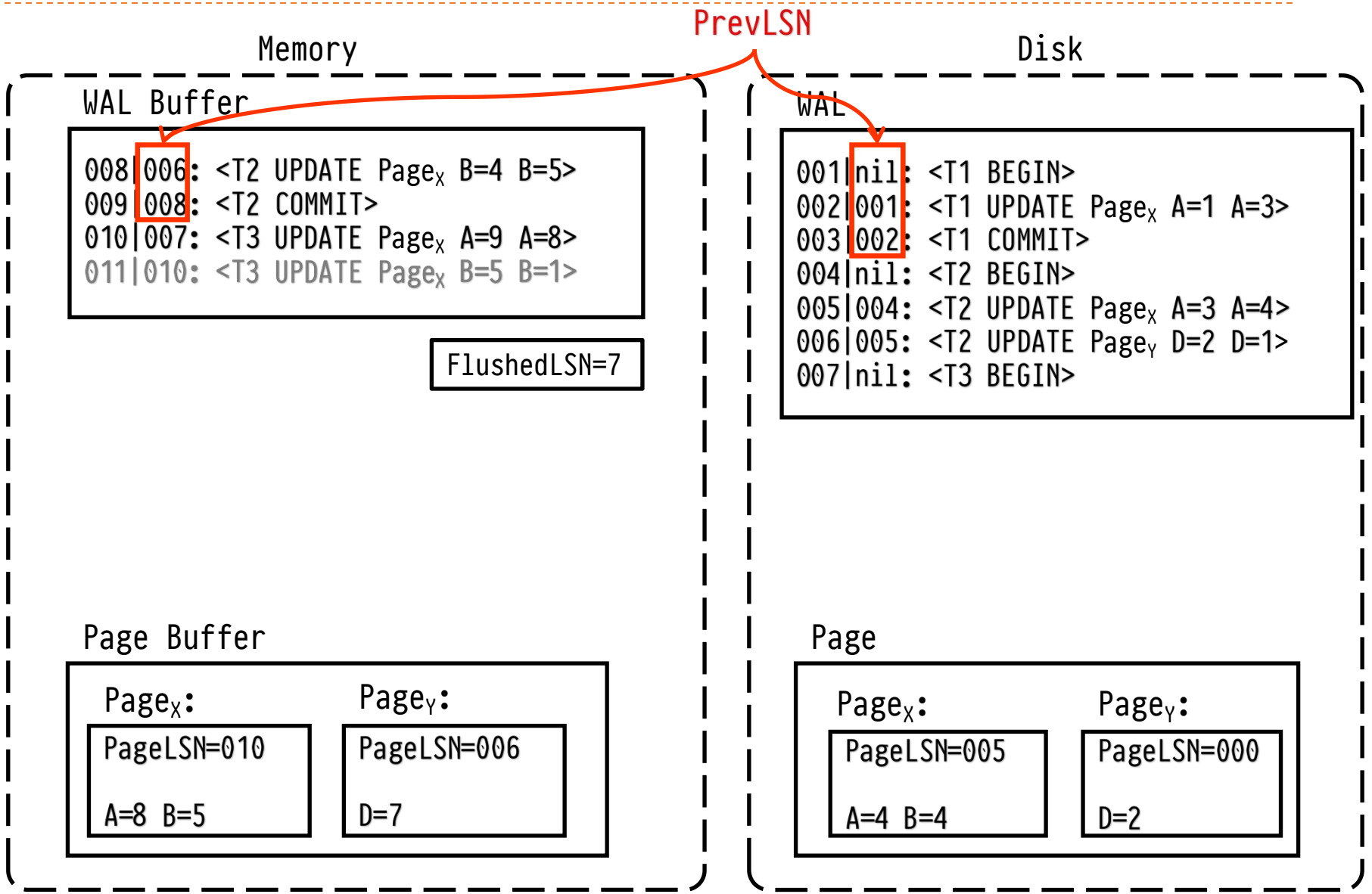
```
LSN: [prevLSN, TransID, "update", pageID, redo Info, undo Info]
```

- LSN (Log Sequence Number)每条日志拥有一个全局唯一单调递增的LSN。
- PrevLSN：当前事务的前一条日志的LSN。如果已经是第一条日志，那么PrevLSN为空
- TransID：产生这条日志的事务ID
- PageID：对应的更新操作作用的页面ID
- redo Info：记录如何redo这次更更新的操作
- undo Info：记录如何undo这次更新的操作

事务提交

LSN: [prevLSN, TransID, "update", pageID, undo Info, redo Info]

- 图中的状态下：
  - 事务T1可以提交
  - 事务T2、T3不能提交



事务回滚

单个事务回滚流程

1. 事务T执行了一些更新操作后被abort，写一条Abort日志
2. 倒序遍历事务T的日志，对它的每一个更新日志记录：

   写一条补偿日志记录CLR

   将缓冲区中相应的数据恢复成旧值
3. 为事务T写一条TXN-END日志

事务回滚

补偿日志记录 （Compensation Log Record，CLR）

对事务的每个更新操作进行撤销时，也会对数据库进行修改。按照WAL的原则，也需要先写日志

```
LSN: [prevLSN, TransID, "CLR", redoTheUndo Info, undoNextLSN]
```

redoTheUndo Info

记录如何执行一次undo操作

undoNextLSN

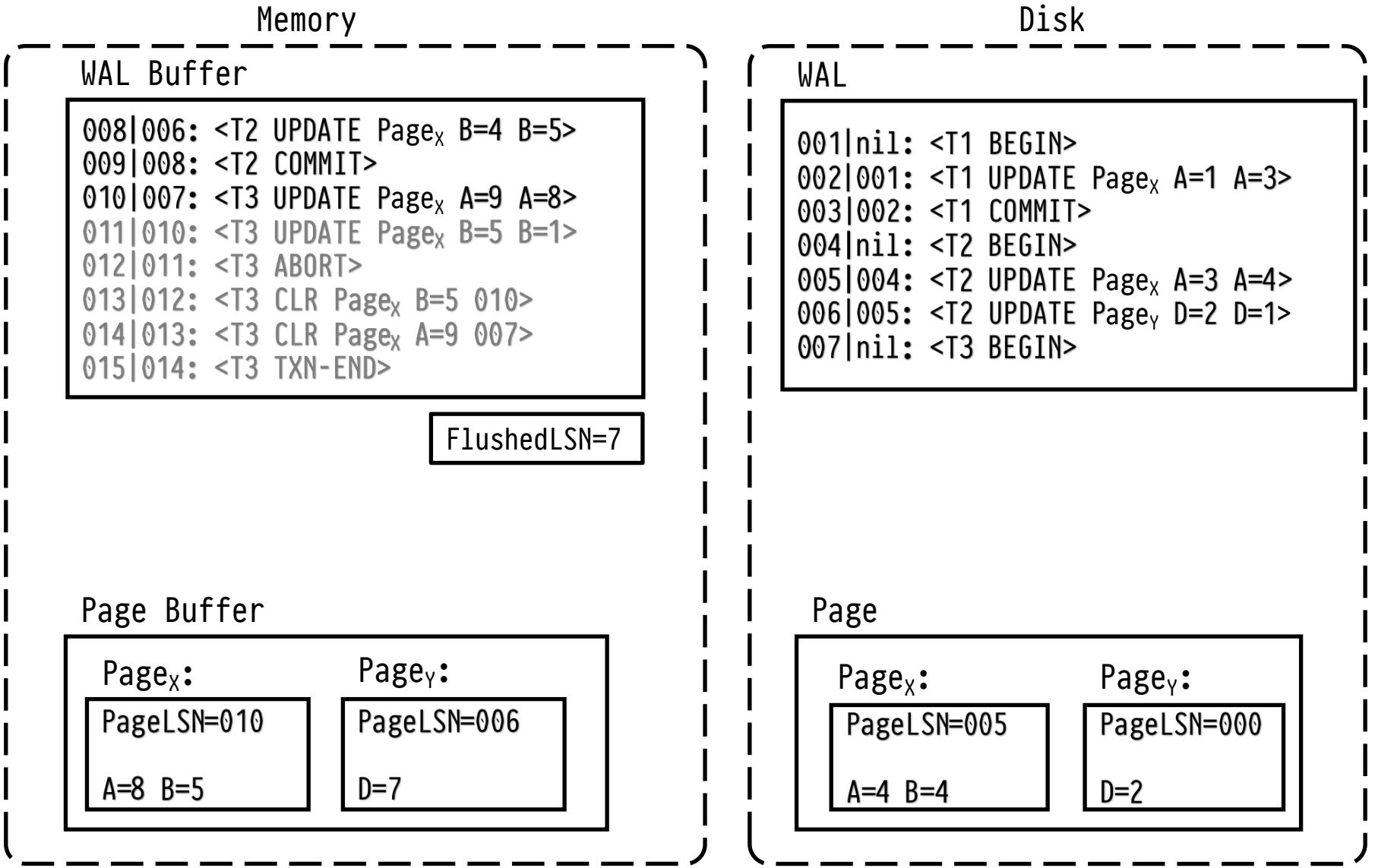
指向了下一条需要被撤销的日志

如果这条CLR是为了撤销更新日志记录LR，那么CLR.undoNextLSN=LR.prevLSN

事务回滚

LSN: [prevLSN, TransID, "CLR", redoTheUndo Info, undoNextLSN]

- 事务T3回滚



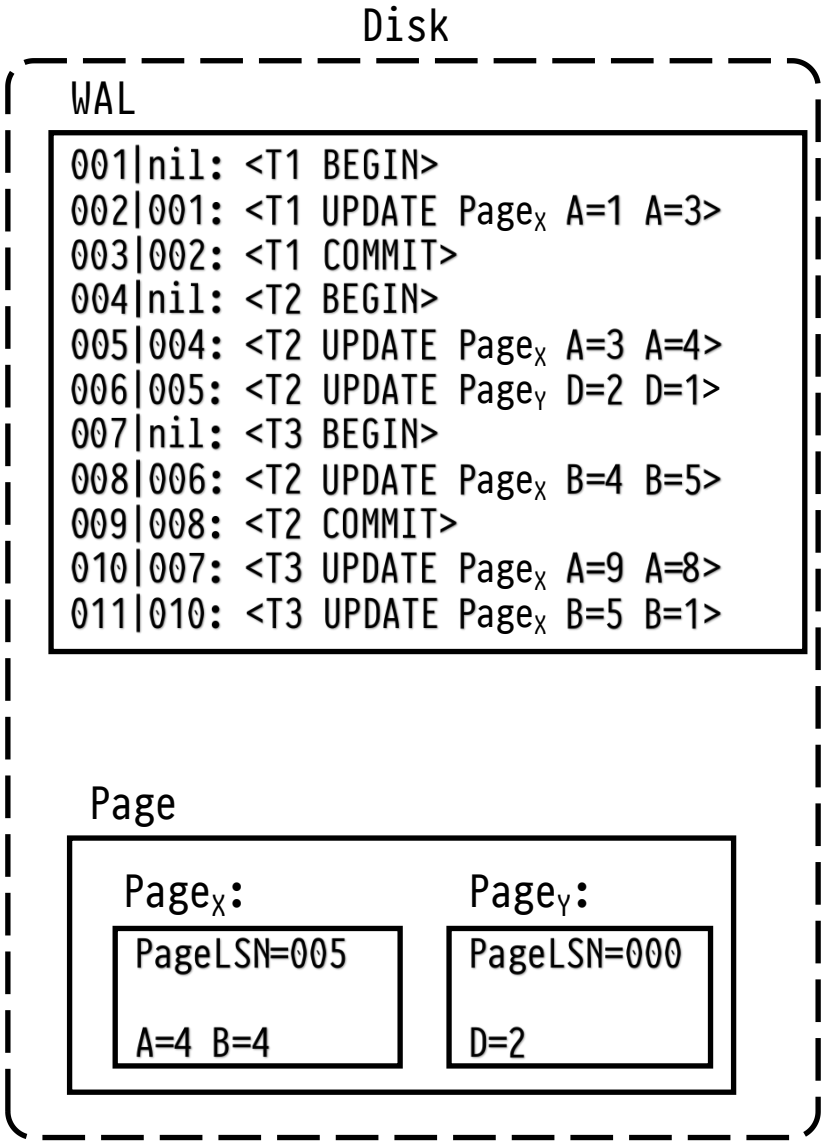
检查点

系统崩溃，重启时使用日志进行恢复

REDO 所有日志，让系统恢复到崩溃前的状态

UNDO 失败事务

使用检查点来减少需要分析的日志



Sharp检查点

Sharp检查点：创建检查点时让数据库进入一致的状态

- 阻止任何新事务的启动
- 等待所有事务执行完成
- 将所有脏页面刷到磁盘

可以从检查点开始进行恢复

- Checkpoint 之前提交的事务的更新操作一定被写回磁盘了
- Checkpoint 时没有执行的事务

- 同步写磁盘需要的时间?
- 长时间执行的事务?

模糊检查点

模糊检查点(fuzzy checkpoint)： 创建检查点时不暂停事务

- 允许未完成的事务继续执行
- 不强制将脏页面刷回磁盘
- 脏页面由BM持续写回磁盘，不影响事务执行和检查点的创建

- 必须在检查点中记录必要的状态信息
  - 活动事物表 Transaction Table
  - 脏页表 Dirty Page Table

模糊检查点

模糊检查点：创建检查点时不暂停事务

- 允许未完成的事务继续执行
- 不强制将脏页面刷回磁盘
- 脏页面由BM持续写回磁盘，不影响事务执行和检查点的创建

活动事务表 ATT

- 活动事务表(TT)记录了当前还在活动（未提交）的事务
- 对每个活动事务包含以下信息
  - TransID: 事务的 ID
  - LastLSN: 该事务所做最后修改的日志

模糊检查点

模糊检查点：创建检查点时不暂停事务

- 允许未完成的事务继续执行
- 不强制将脏页面刷回磁盘
- 脏页面由BM持续写回磁盘，不影响事务执行和检查点的创建

脏页表 DPT

- 脏页表记录了缓冲区中被修改，但还未被写回磁盘的页面信息。
- 对每个脏页面包含以下信息
  - PageID：脏页面的 ID
  - RecoveryLSN: 使当前页面变脏的第一条日志

模糊检查点

模糊检查点：创建检查点时不暂停事务

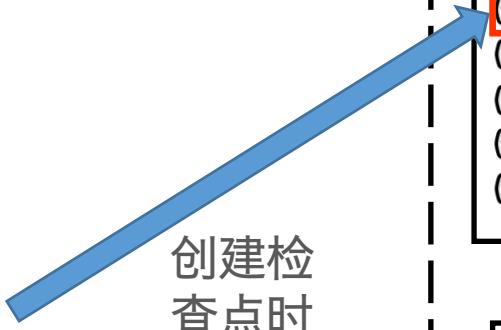
- 允许未完成的事务继续执行
- 不强制将脏页面刷回磁盘
- 脏页面由BM持续写回磁盘，不影响事务执行和检查点的创建

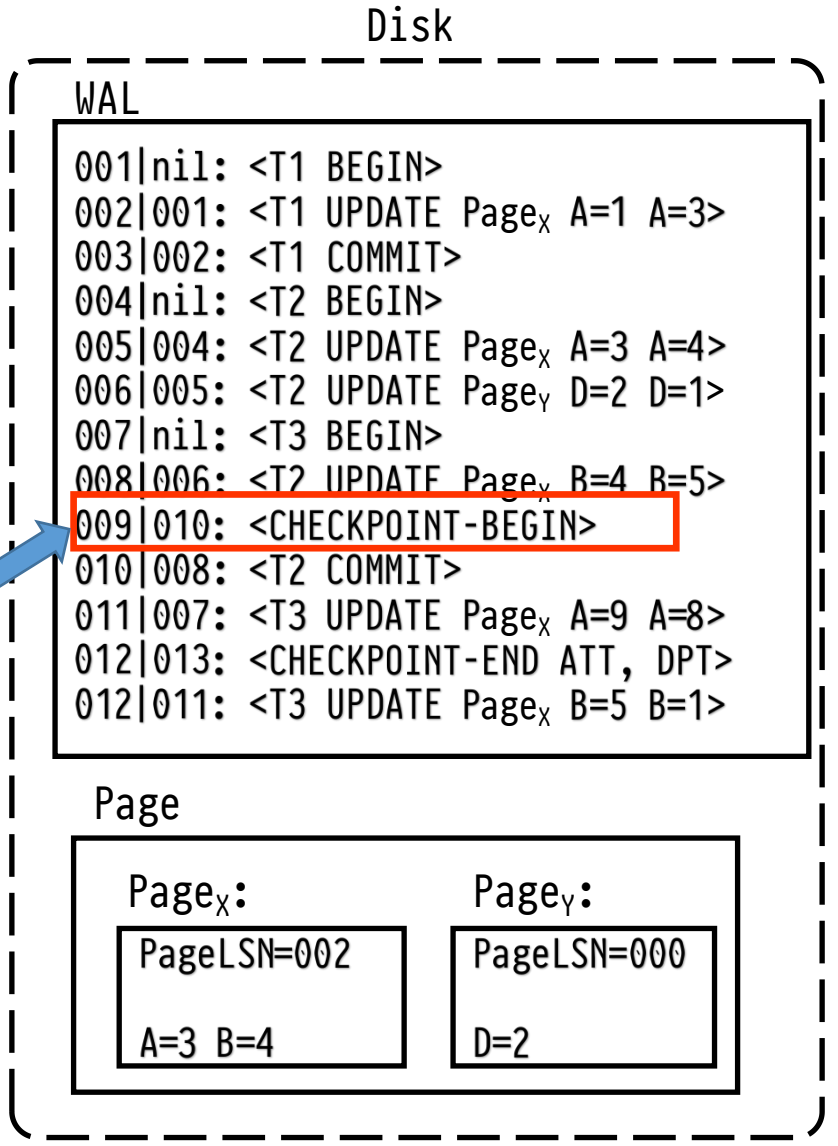
DPT

PageID	RecoveryLSN
x	005
y	006

ATT

TransID	LastLSN
T2	008
T3	007





模糊检查点

使用两个新的日志来记录检查点的边界

- CHECKPOINT-BEGIN

  指出检查点的位置

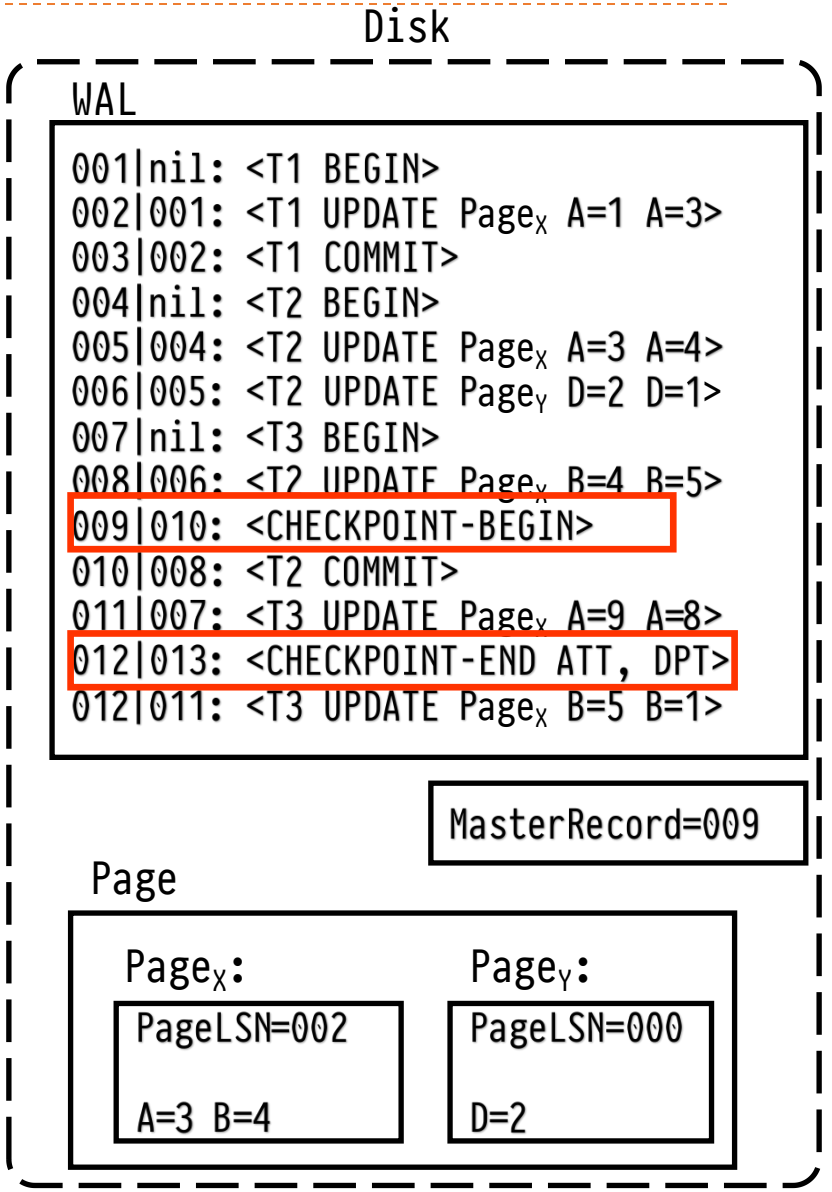
- CHECKPOINT-END

  记录ATT和DPT信息

CHECKPOINT-BEGIN 时刻创建DPT和ATT副本 （COW）

- 检查点写副本
- 事务正常执行更新另一份

在磁盘上记录最后一个成功的检查点 MasterReocrd



恢复流程

1. Analysis阶段

   从上一个成功的检查点开始分析WAL

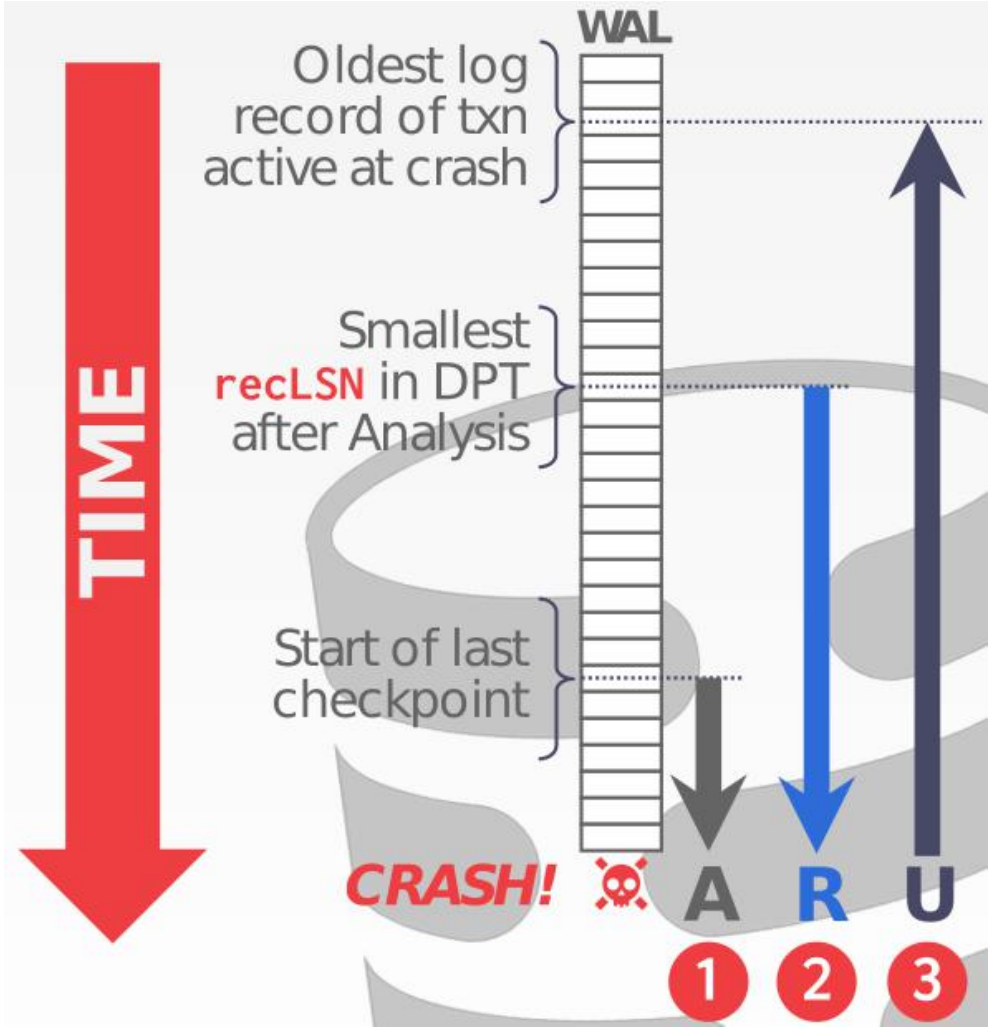
   - 找出脏页面，得到Redo起始位置
   - 找出未提交事务，确定需要undo的事务

2. Redo阶段

   从某个地方开始重做所有日志（包括失败的）

3. Undo阶段

   撤销崩前未提交的事务



恢复流程

1. Analysis阶段
    1. 根据MasterRecord找到最后一个成功的检查点
    2. 从Checkpoint-End中的数据初始化DPT和ATT
    3. 从Checkpoint-Begin开始，顺序扫描日志：
        - 发现属于不在ATT表中事务的日志

          就将其加入ATT，并更新LastLSN为日志的LSN
        - 发现一个结束的事务（已提交或者回滚完成）

          将其从TT表中删除
        - 发现一个更新了页面的日志记录

          如果这个页面不在DPT中，就将其添加进在DPT并设置recoveryLSN为该日志的LSN

恢复流程

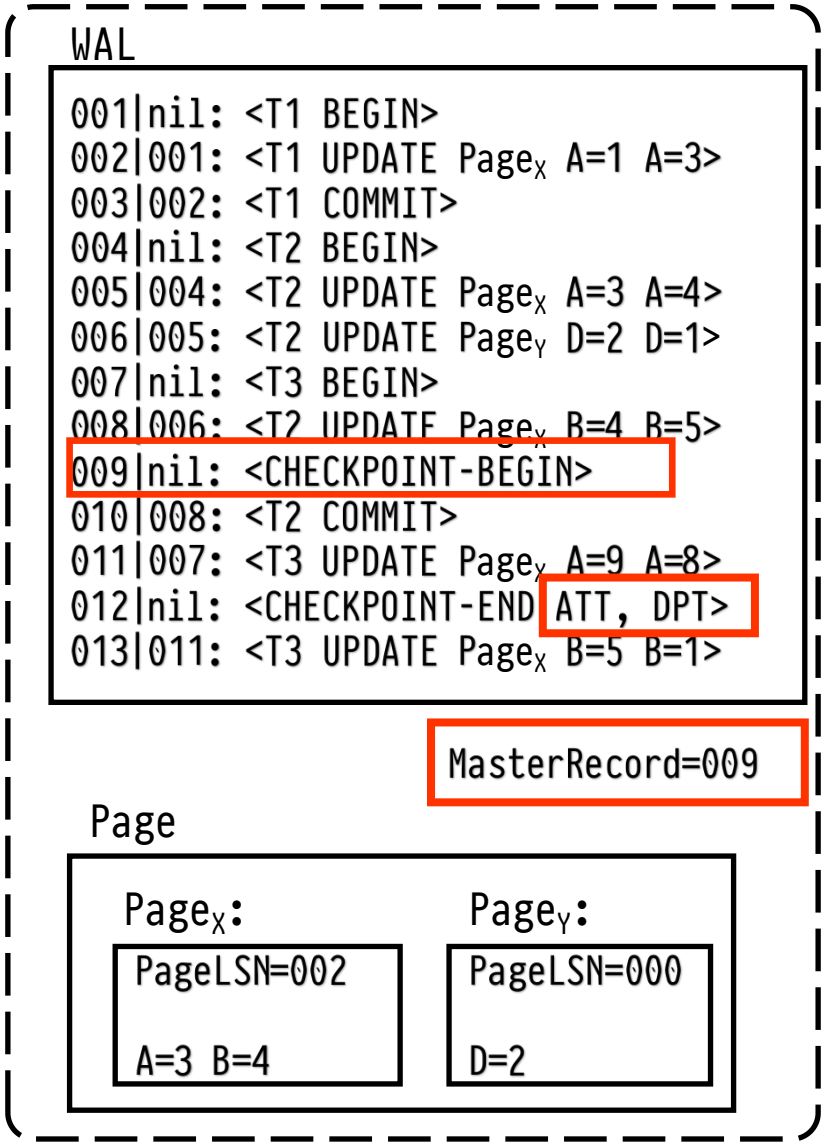
1. Analysis阶段

DPT

PageID	RecoveryLSN
x	005
y	006

ATT

TransID	LastLSN
T2	008
T3	007



恢复流程

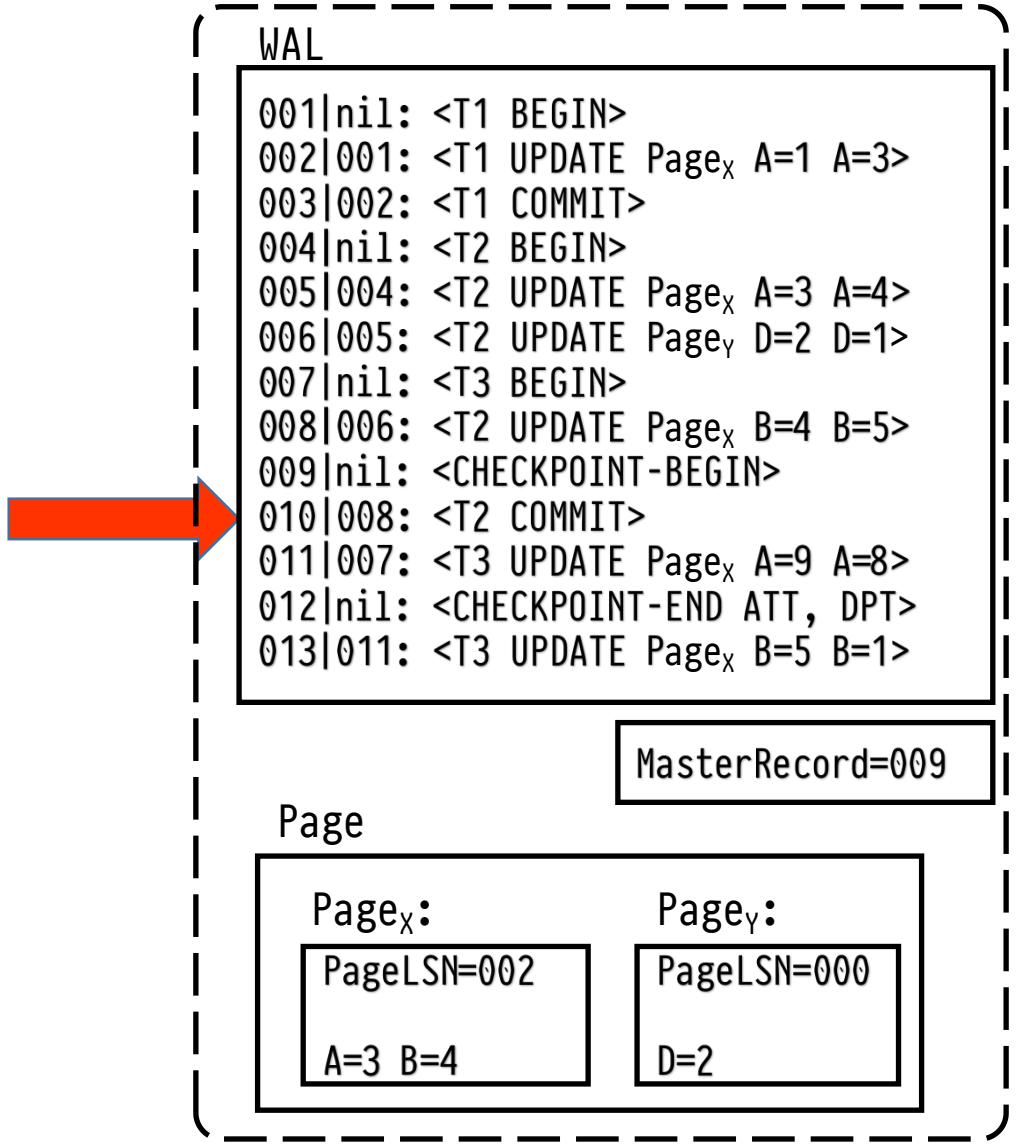
1. Analysis阶段

DPT

PageID	RecoveryLSN
x	005
y	006

ATT

TransID	LastLSN
T2	008
T3	007



恢复流程

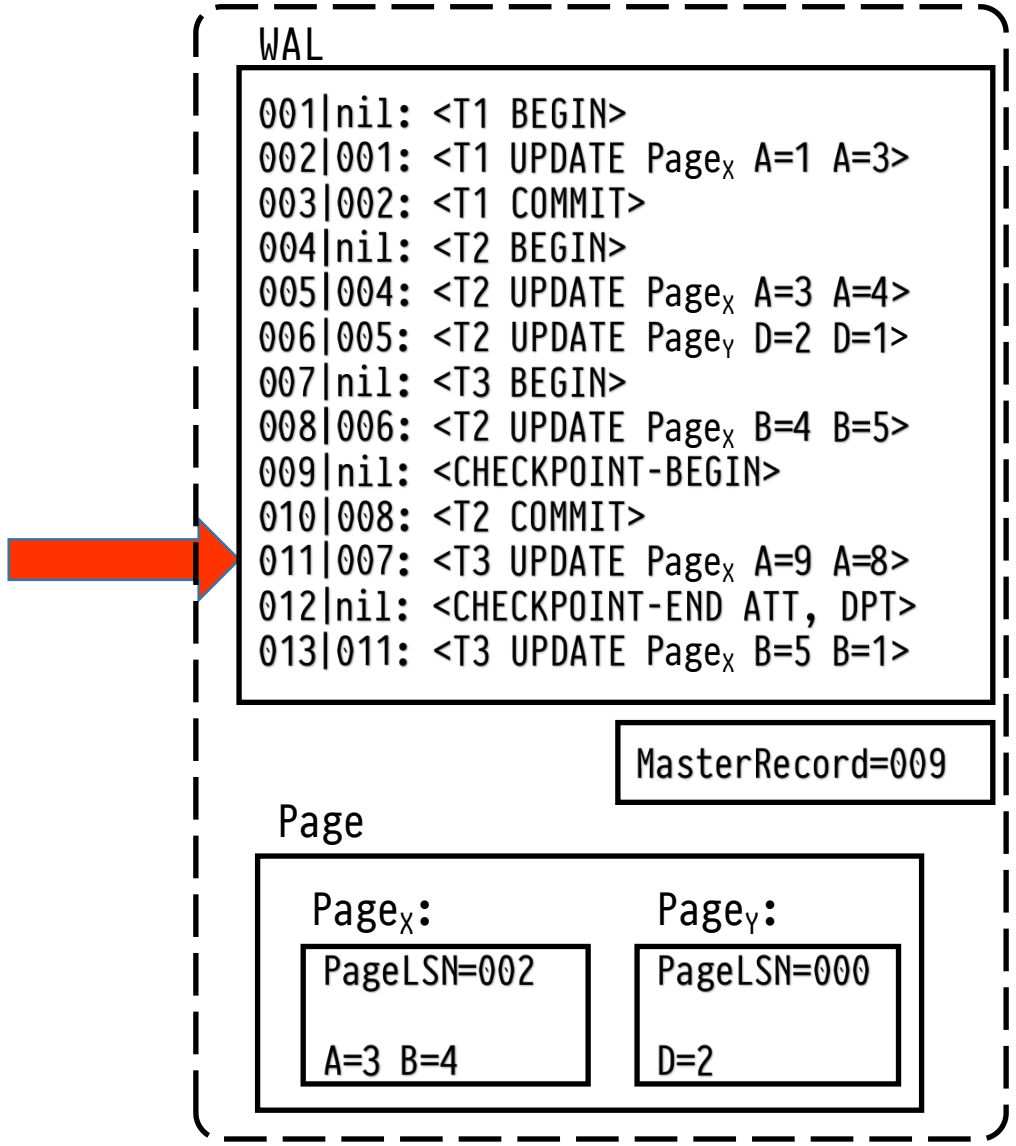
1. Analysis阶段

DPT

PageID	RecoveryLSN
x	005
y	006

ATT

TransID	LastLSN
T2	008
T3	011



恢复流程

分析阶段完成后

DPT：哪些页面是脏的？需要redo哪些日志？

RedoLSN = min(RecoveryLSN)

ATT：哪些事务要被Undo？

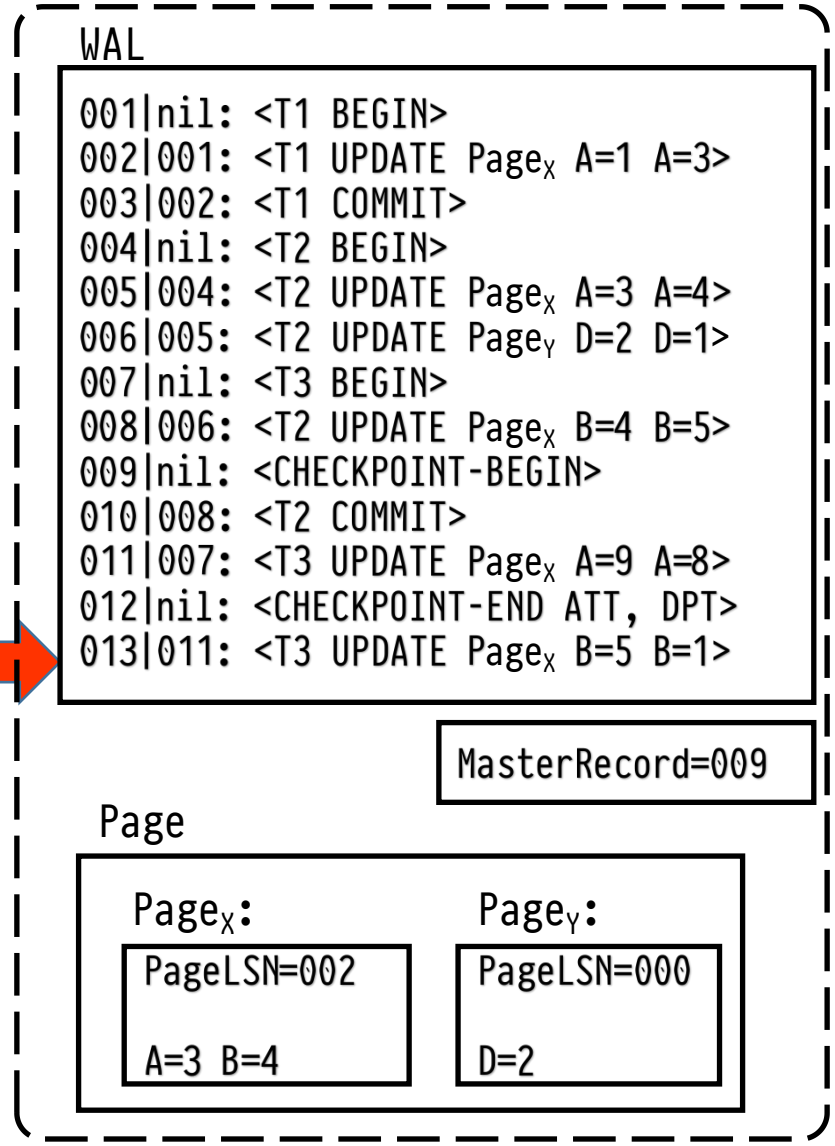
1. Analysis阶段

DPT

PageID	RecoveryLSN
x	005
y	006

ATT

TransID	LastLSN
~~T2~~	~~008~~
T3	013



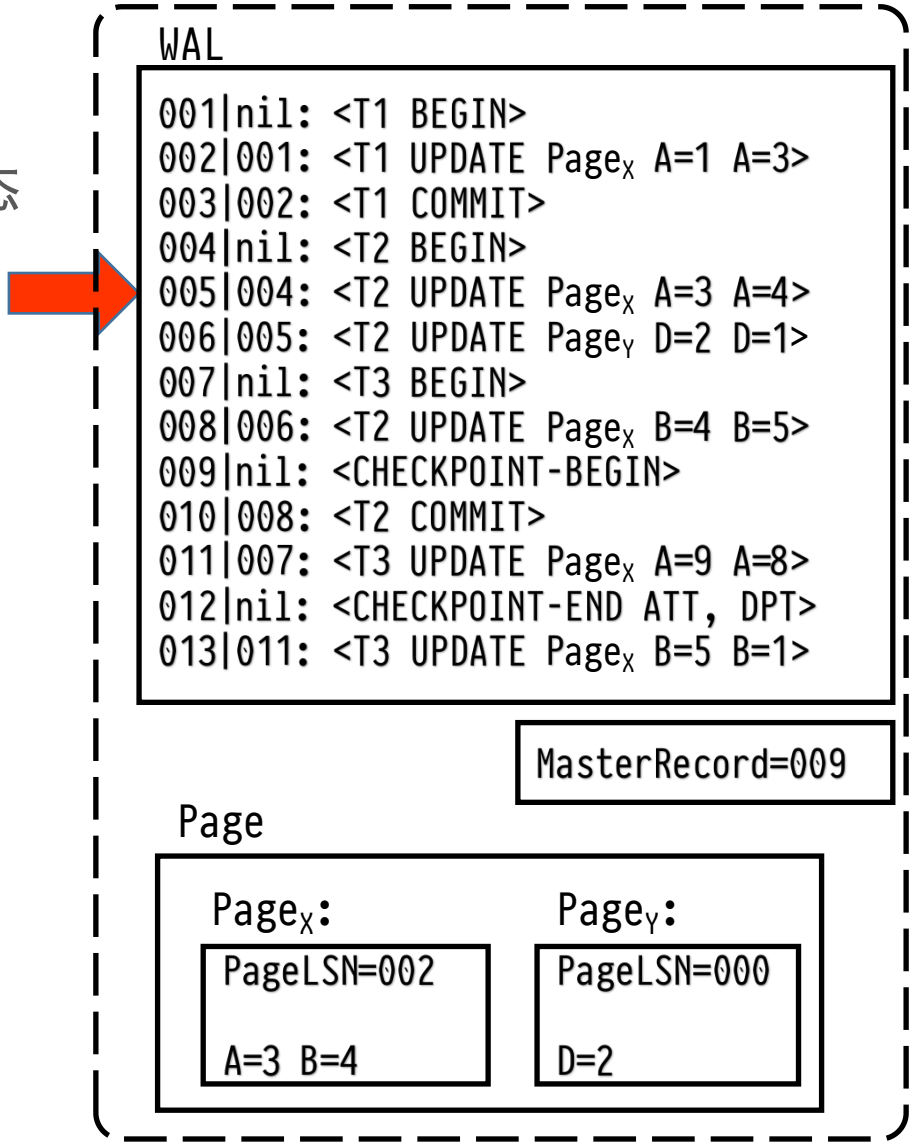
恢复流程

2. Redo阶段

- Redo 阶段repeat history，让数据库进入崩溃时的状态
- 从RedoLSN开始，按顺序重做日志（包括失败事务的）

对每个日志:

1. 检查，跳过以下日志
   - 所涉及页面不在DPT中
   - 所涉及页面的PageLSN大于日志LSN
2. 重做
   - 更新缓冲区页面
   - 更新缓冲区相应页面的PageLSN
   - 不强制写回



恢复流程

3. Undo阶段

- 撤销ATT中的事务
- 与撤销单个事务方法相同，使用CLR
- 按照LSN逆序操作
- 使用ATT中的LastLSN和PrevLSN直接定位

ATT

TransID	LastLSN
~~T2~~	~~008~~
T3	013

WAL

```
001|nil: <T1 BEGIN>
002|001: <T1 UPDATE Page_X A=1 A=3>
003|002: <T1 COMMIT>
004|nil: <T2 BEGIN>
005|004: <T2 UPDATE Page_X A=3 A=4>
006|005: <T2 UPDATE Page_Y D=2 D=1>
007|nil: <T3 BEGIN>
008|006: <T2 UPDATE Page_X B=4 B=5>
009|nil: <CHECKPOINT-BEGIN>
010|008: <T2 COMMIT>
011|007: <T3 UPDATE Page_X A=9 A=8>
012|nil: <CHECKPOINT-END ATT, DPT>
013|011: <T3 UPDATE Page_X B=5 B=1>
```

恢复流程

3. Undo阶段

- Undo阶段发生崩溃?

  再次重启，继续上一次未完成的撤销

ATT

TransID	LastLSN
T3	014

WAL

```
001|nil: <T1 BEGIN>
002|001: <T1 UPDATE PageX A=1 A=3>
003|002: <T1 COMMIT>
004|nil: <T2 BEGIN>
005|004: <T2 UPDATE PageX A=3 A=4>
006|005: <T2 UPDATE PageY D=2 D=1>
007|nil: <T3 BEGIN>
008|006: <T2 UPDATE PageX B=4 B=5>
009|nil: <CHECKPOINT-BEGIN>
010|008: <T2 COMMIT>
011|007: <T3 UPDATE PageX A=9 A=8>
012|nil: <CHECKPOINT-END ATT, DPT>
013|011: <T3 UPDATE PageX B=5 B=1>

014|013: <T3 CLR PageX B=5 011>

014|013: <T3 CLR PageX A=9 007>
015|014: <T3 TXN-END>
```

CRUSH

CRUSH AGAIN

UNDO阶段写CLR
使得在恢复的任何阶段崩溃，只需要重新执行恢复流程就可以了

总结

- WAL

  支持Steal和No-Force的缓冲策略

- 模糊检查点

  不暂停事务，也不刷脏页面，只记录状态(DPT，ATT)

- 从最早的脏页面开始Redo
- Undo写CLR，能够处理重复崩溃
- LSN

  日志LSN将事务的日志链接起来，减小撤销代价

  PageLSN允许对日志和页面进行比较（减小代价、并支持逻辑物理日志、Buffer管理）