面向NVM的
数据库故障恢复机制

汇报人：董创轼

思路

1. 为什么需要故障恢复机制?
2. 传统的故障恢复机制有哪些?
   - Shadow Page、WAL
   - ARIES
3. NVM的特点有哪些?
   - 大容量、字节寻址
   - 面向这些特点有哪些思路?
4. 基于NVM的优化
   - MARS
   - WBL
5. 总结

目录

01 ／ 问题与背景

01 为何需要故障恢复

事务的特性：

1. A —— 原子性（ Atomicity ）
2. C —— 一致性（ Consistency ）
3. I —— 隔离性（ Isolation ）
4. D —— 持久性（ Durability ）

故障类型：

1. 事务本身失败
2. 系统失败
3. 存储失败

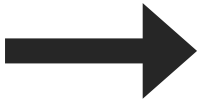
为何需要故障恢复

事务的特性：

1. A —— 原子性（ Atomicity ）
2. C —— 一致性（ Consistency ）
3. I —— 隔离性（ Isolation ）
4. D —— 持久性（ Durability ）

故障类型：

1. 事务本身失败
2. 系统失败
3. 存储失败

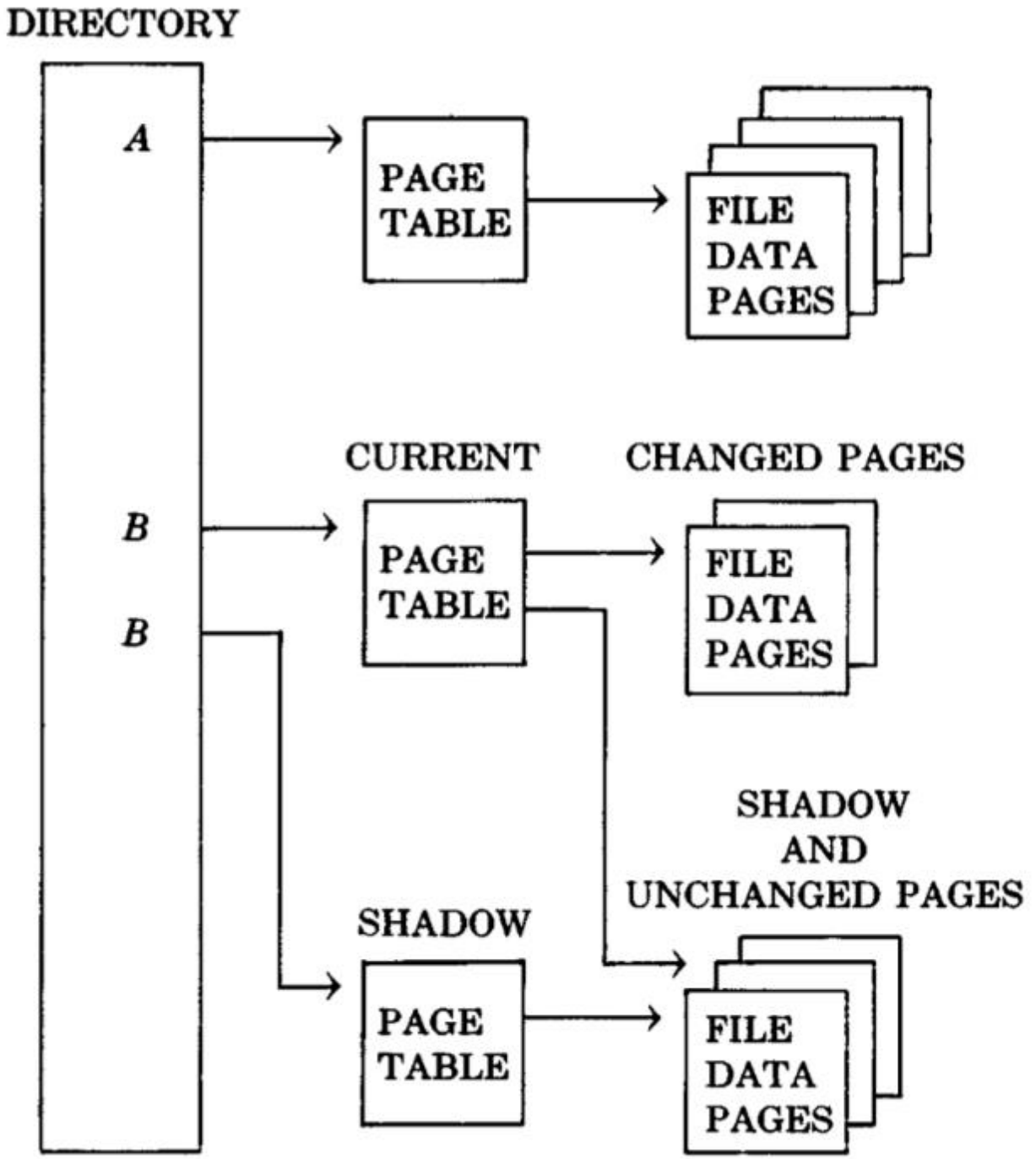


故障恢复的存在就是在出现故障的情况下，依然能保证事务的**持久性与原子性**

01 Shadow Page

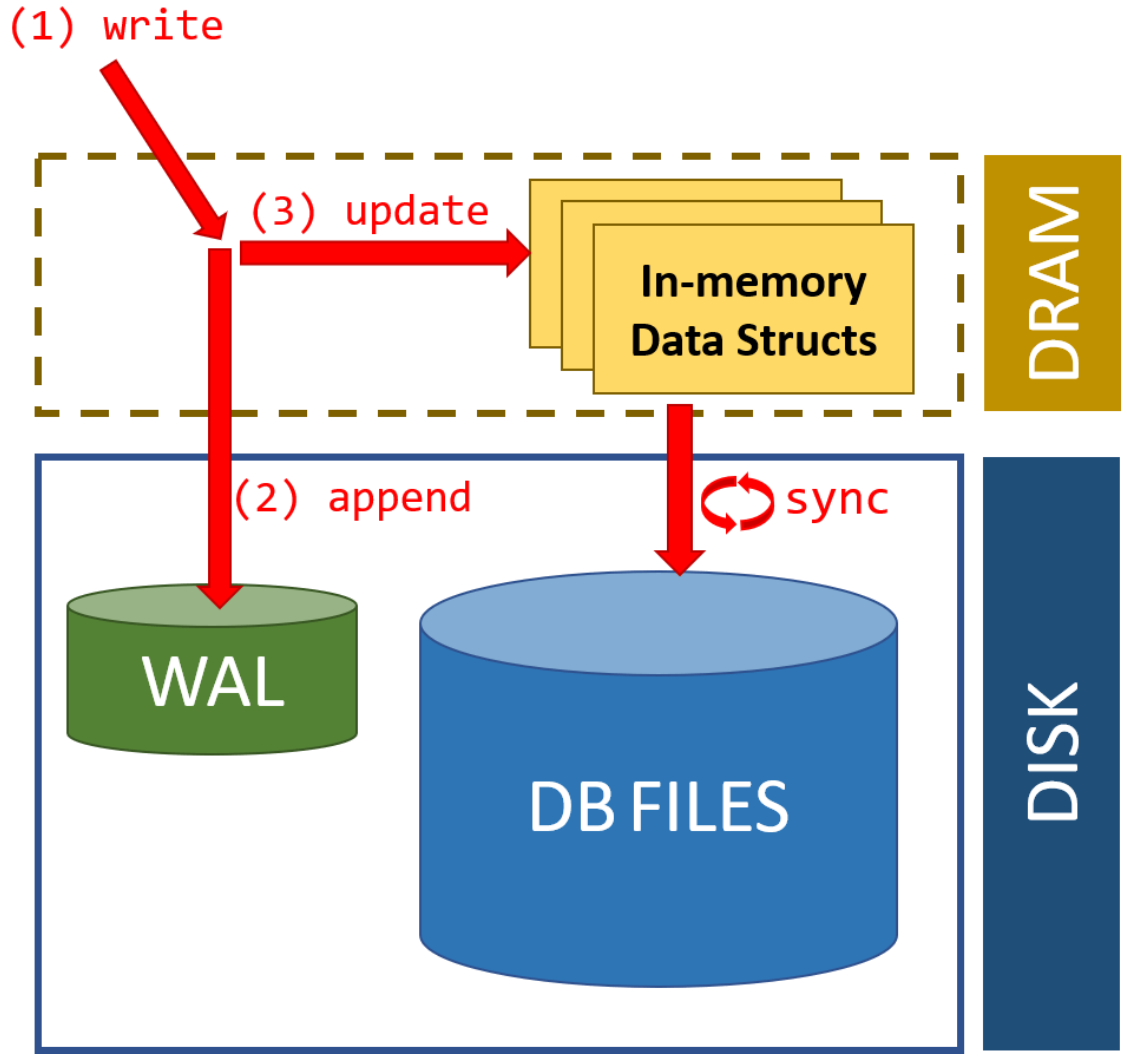
- 事务修改page的时候，数据库复制原来的page
- 事务提交时，原子性合并page
- 事务回滚时，丢弃修改的page

- 不支持page内并发
- 不断修改page，相关的页局部性差
- 垃圾回收负担大



01 WAL

- 事务所有日志都写入持久存储设备，才视为成功
- 先写入日志，再写入page
- 基于磁盘顺序IO远高于随机IO
- 日志文件依靠checkpoint回收



01 Buffer Pool策略

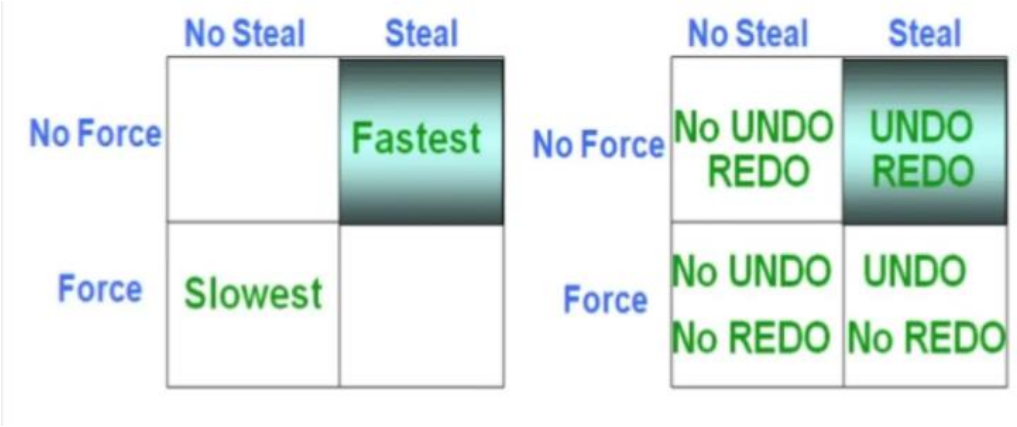
Steal — 事务提交之前，允许写的数据刷入磁盘?

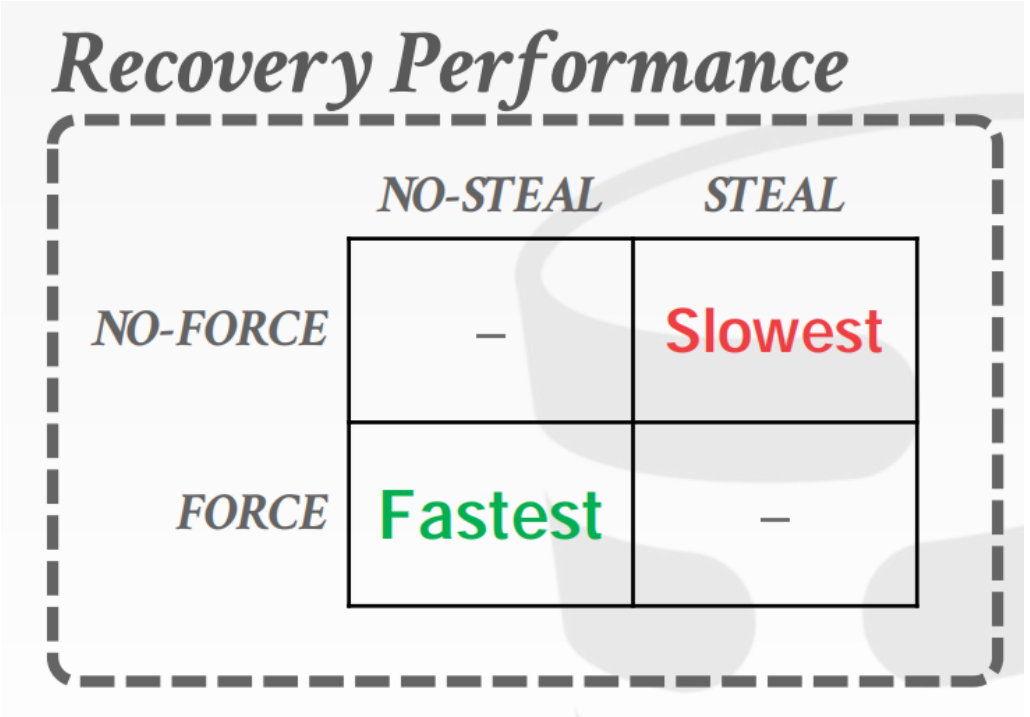
- steal：允许
- no-steal：不允许

Force — 事务提交的时候，是否强制把他所有的修改都刷回到磁盘?

- force：强制刷
- no-force：不强制刷

绝大多数数据库方案都是采用steal + no-force





01 Aries

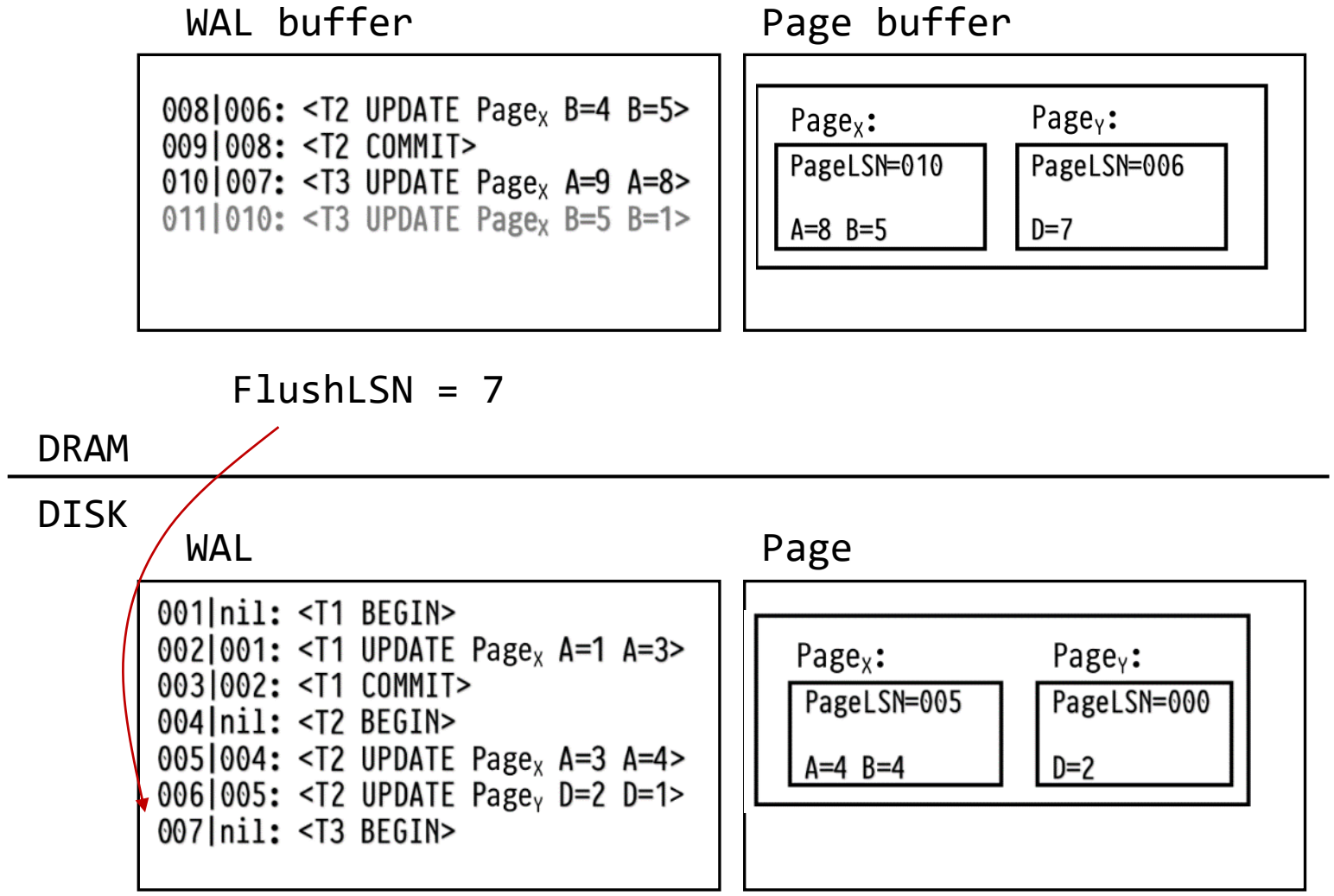
- 本质是基于undo-redo WAL的实现
- no-force + steal

LSN —— 全局唯一的编号

- PageLSN
  - 内存中每个page最新操作对应的日志号
- FlushLSN
  - 刷入磁盘中最大的LSN

对于一个页面：

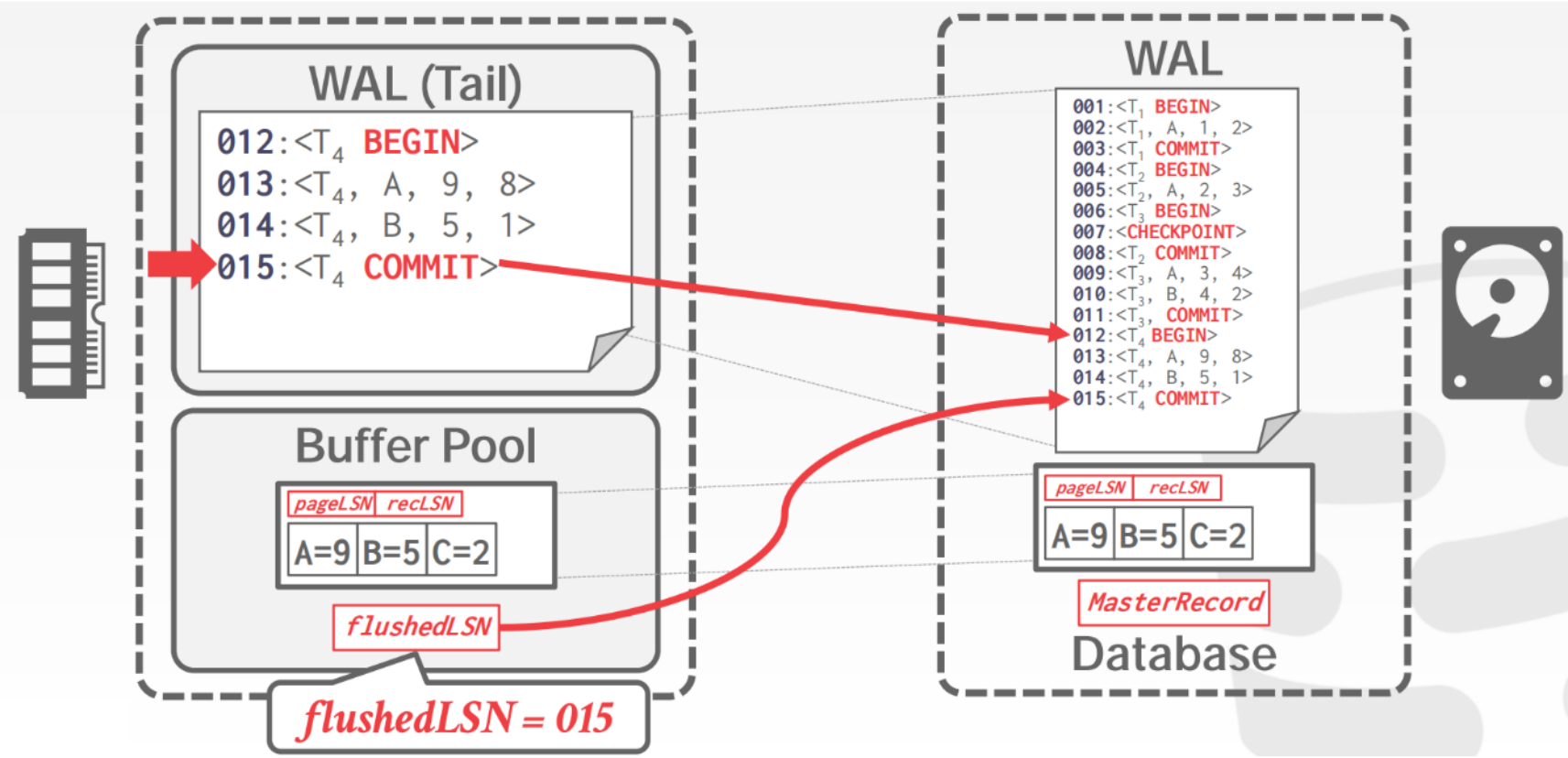
pageLSN x ≤ FlushedLSN



01 Aries

事务提交

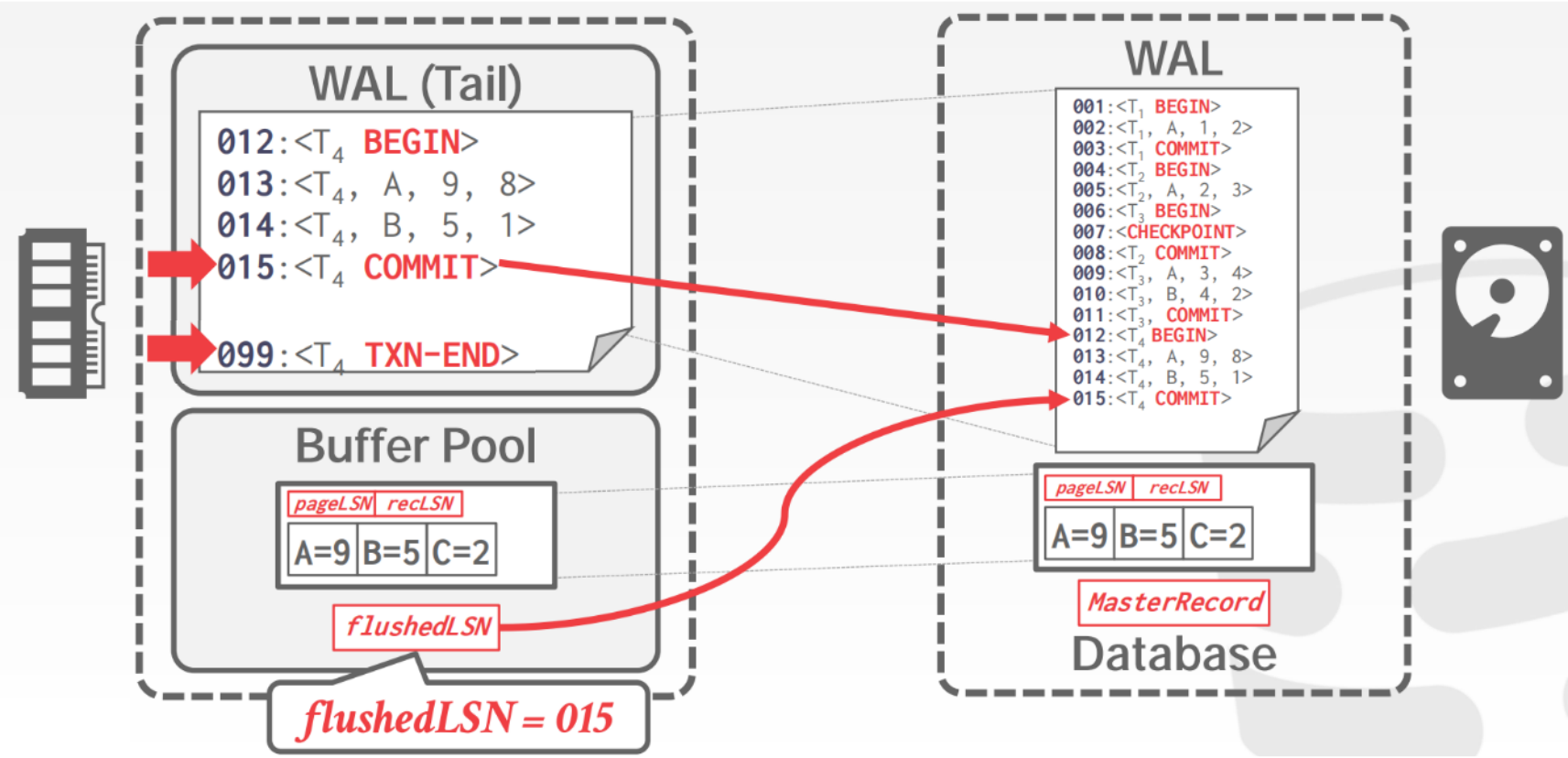
1. 事务T执行过程中，对于每一个更新操作：
   1. 写一条日志记录
   2. 更新缓冲区数据
2. 事务T准备提交，写一条Commit日志
3. 当FlushedLSN >= Commit日志的LSN，事务提交成功
4. 对应的page刷盘后，为事务T写一条TXN-END日志



01 Aries

事务提交

1. 事务T执行过程中，对于每一个更新操作：
   1. 写一条日志记录
   2. 更新缓冲区数据
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. 对应的page刷盘后，为事务T写一条TXN-END日志



01 Aries

事务回滚

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
   1. 写一条CLR
   2. 恢复缓冲区数据
3. 为事务T写一条TXN-END日志

LSN	prevLSN	TxnId	Type	Object	Before	After	UndoNext
001	nil	T_1	BEGIN	–	–	–	–
002	001	T_1	UPDATE	A	30	40	–
⋮							
011	002	T_1	ABORT	–	–	–	–
⋮							
026	011	T_1	CLR-002	A	40	30	001

The LSN of the next log record to be undone.

01 Aries

模糊检查点

checkpoint → <checkpoint-begin, checkpoint-end>

活动事务表（ ATT ）

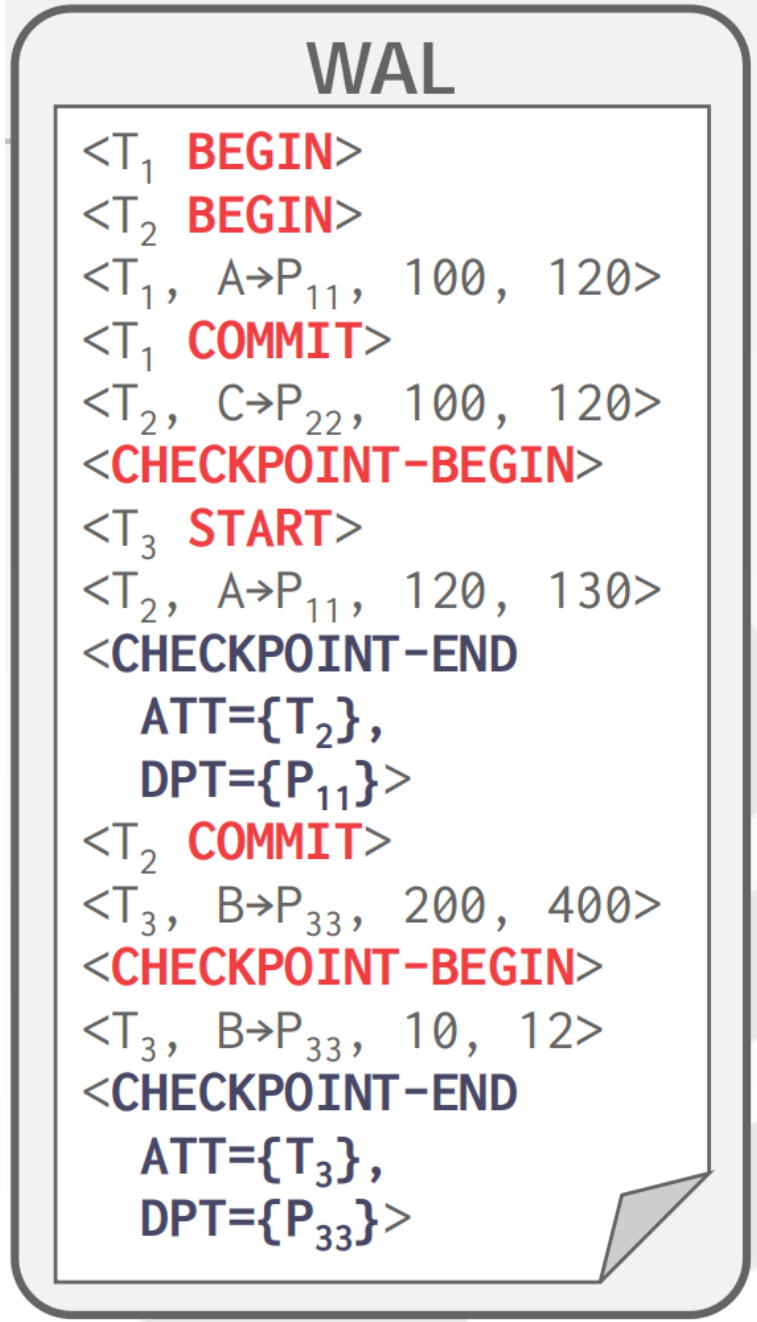
脏页表（ DPT ）

checkpoint-begin

- 表示检查开始
  - 在这个点之后开始的事务都不会记录到ATT中
- 开始分析生成两种表

checkpoint-end

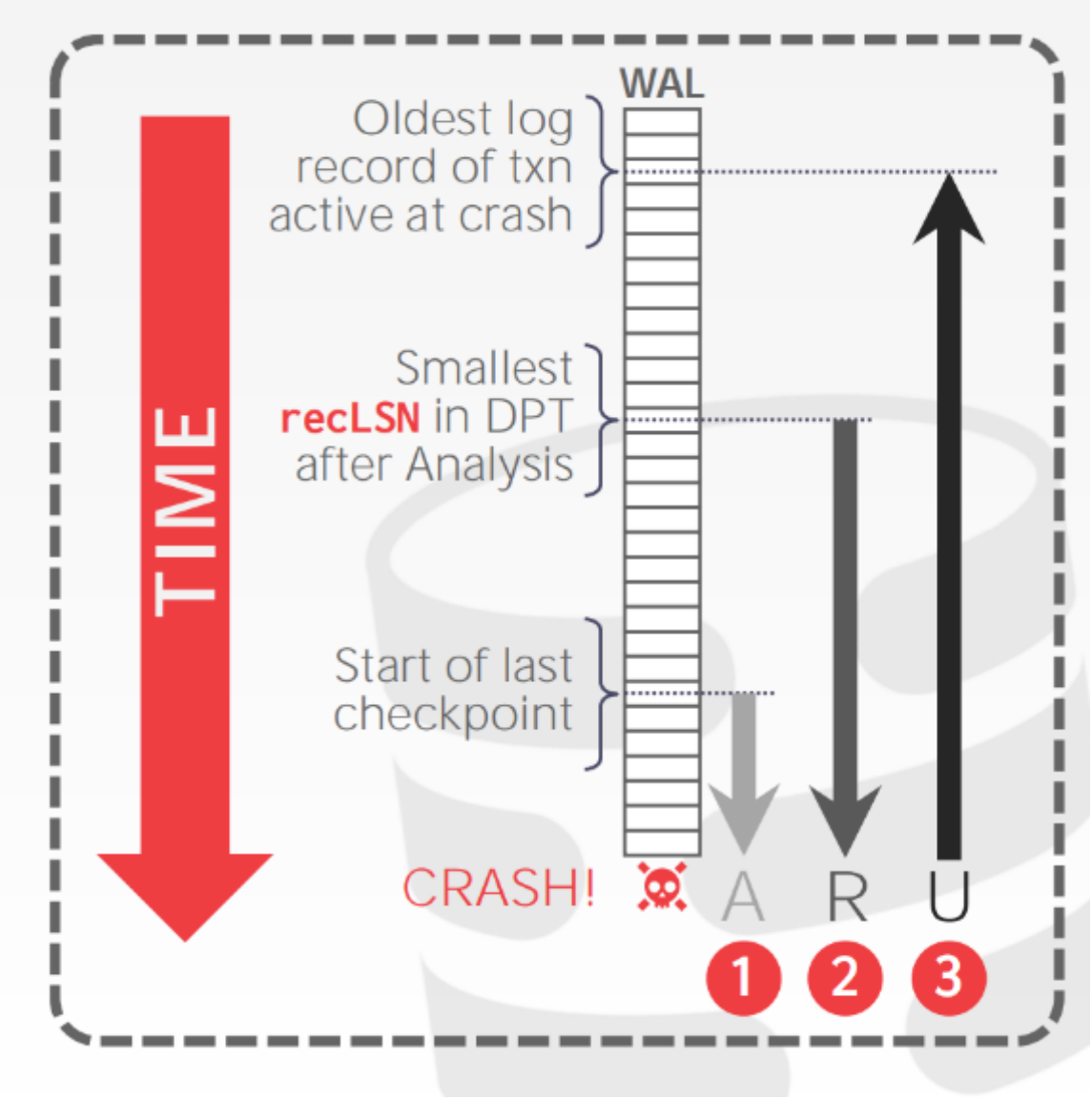
- 检查结束，生成最终的ATT与DPT



01 Aries

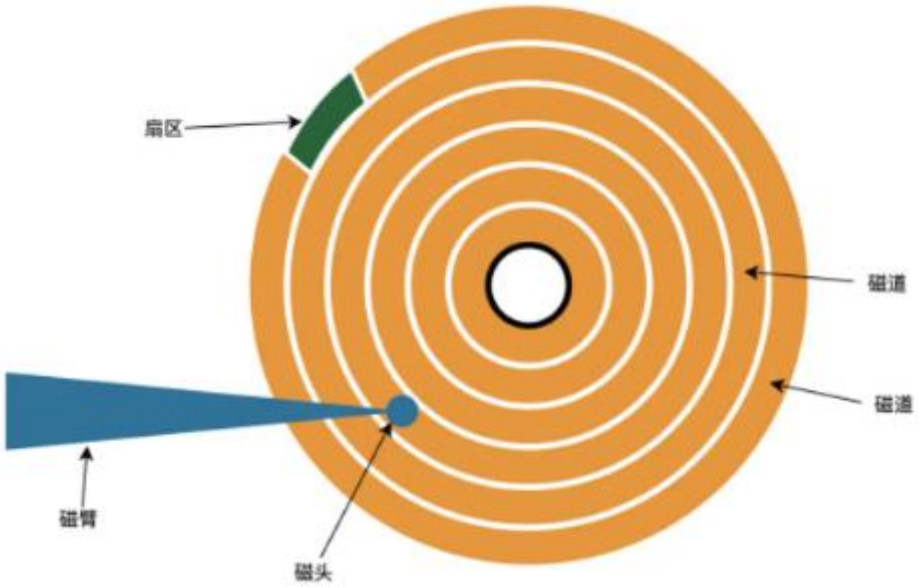
恢复

1. Analysis:
    - 从上一个成功创建的检查点开始分析WAL
    - 恢复到系统崩溃点
        - 根据脏页面，得到redo起始位置
        - 根据未提交事务，确定需要undo的事务
2. Redo:
    - 重做日志，包括CLR
3. Undo:
    - 撤销失败的事务的影响



02 NVM特点

02 NVM特点



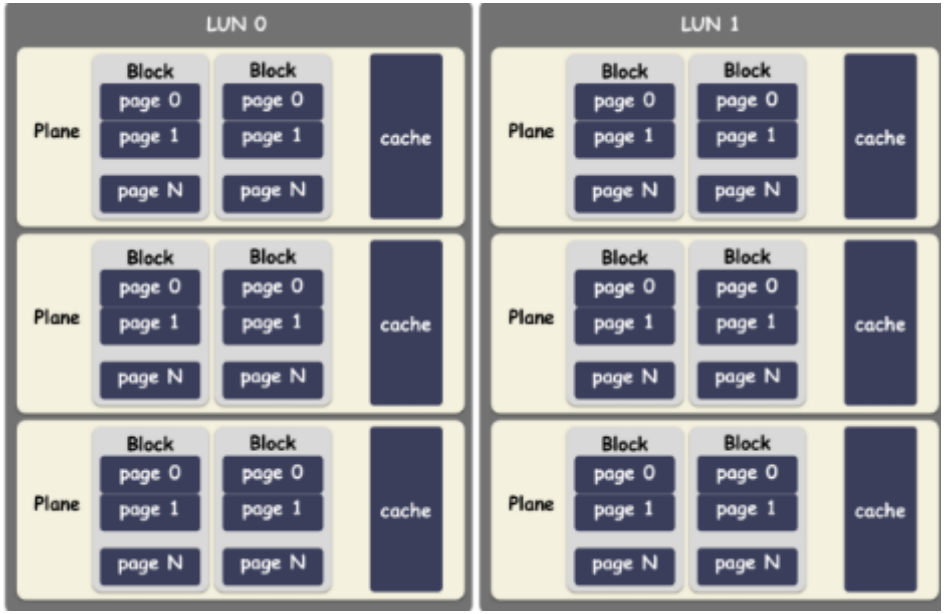
优点：

1、廉价、存储量大、寿命长、断电后数据也能保存很久

2、是最擅长顺序IO的存储介质

缺点：

1、顺序IO和随机IO性能差异大



优点：

1、闪存结构，速度快

2、随机IO和顺序IO差异没那么大

缺点：

1、写前擦除效率影响IO

2、寿命短

02 NVM特点

1．整体性能与DRAM接近

2．随机读写与顺序读写差别不大

3．可字节寻址

Table 1　Characteristics NVM and other storage technologies

	DRAM	NVM	SSD	HDD
读延迟	60 ns	250 ns	25 us	10 ms
写延迟	60 ns	500 ns	300 us	10 ms
寻址单元	字节	字节	块	块
易失性	易失	非易失	非易失	非易失
容量	1x	4x	4x	—
写寿命	10^{16}	10^{10}	10^{5}	10^{16}
价格	>150x	>60x	>8x	1x

读粒度/KB	延迟		比值
	随机写	顺序写	
1	538 ns	503 ns	1.07
4	2 006 ns	2 004 ns	1.00
16	8 000 ns	8 089 ns	0.99
64	31 597 ns	32 115 ns	0.98
256	0.13 ms	0.13 ms	1.00
1 024	0.51 ms	0.51 ms	1.00
4 096	2.61 ms	2.60 ms	1.00
16 384	10.37 ms	10.50 ms	0.99

读粒度/KB	延迟		比值
	随机读	顺序读	
1	611 ns	589 ns	1.04
4	2 361 ns	2 326 ns	1.04
16	7 368 ns	7 273 ns	1.03
64	29 086 ns	28 850 ns	1.01
256	0.12 ms	0.12 ms	1.00
1 024	1.09 ms	1.09 ms	1.00
4 096	5.68 ms	5.68 ms	1.00
16 384	22.81 ms	22.82 ms	1.00

02 原有问题

1. No-Force And Steal
   - 需要维护redo、undo，加之原数据，造成三倍写放大，换来的是磁盘的顺序写性能
2. Page管理方式
   - 基于持久化设备为块设备，且顺序写远大于随机写，面对随机设备NVM可否有其他高效的方式?

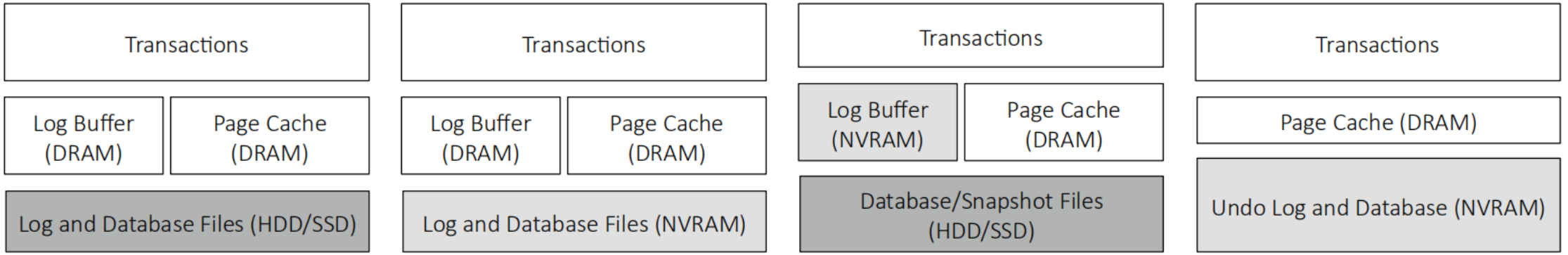
02 优化思路

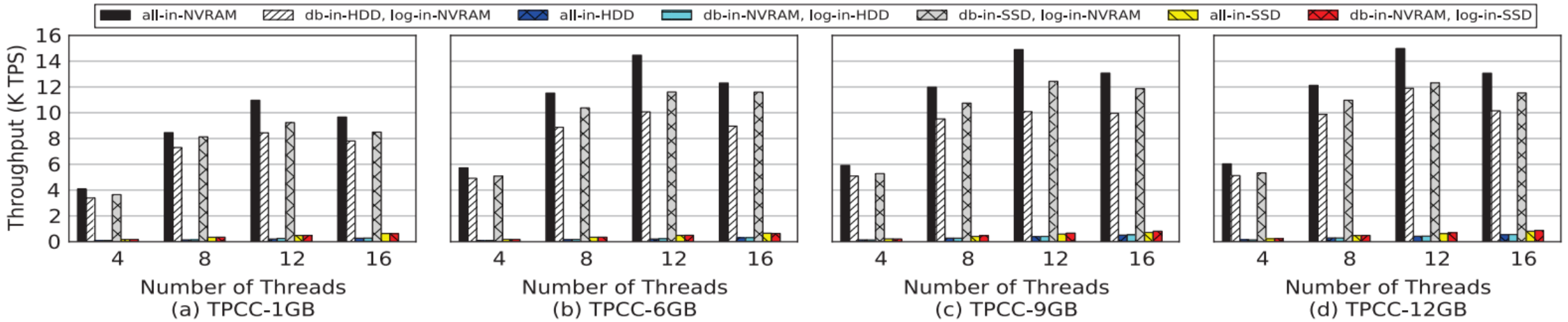
1. 架构方面
    - 如何利用NVM加速故障恢复机制?
    - NVM与磁盘和DRAM如何结合?
2. 算法方面
    - aries的steal + no-force如何在NVM优化?

03 ／ 优化

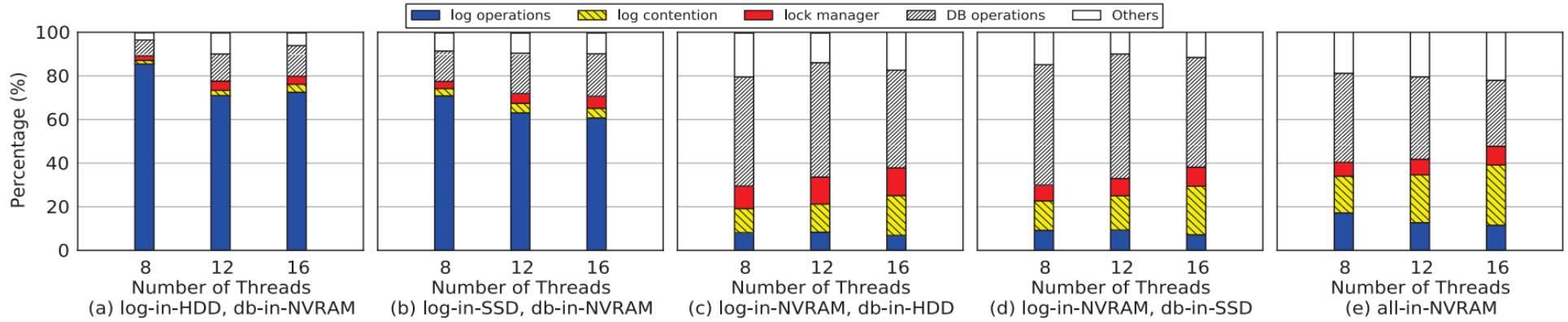
03 NV-logging — NVRAM-aware Logging in Transaction Systems

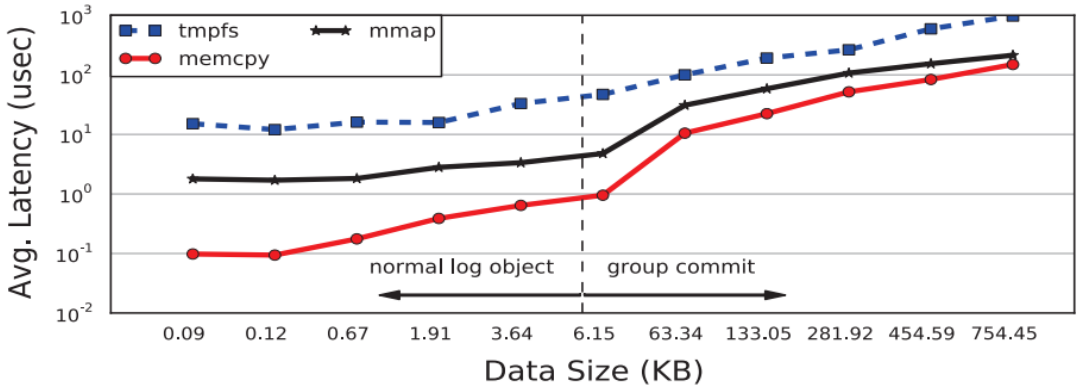
如何更高效的使用NVM?

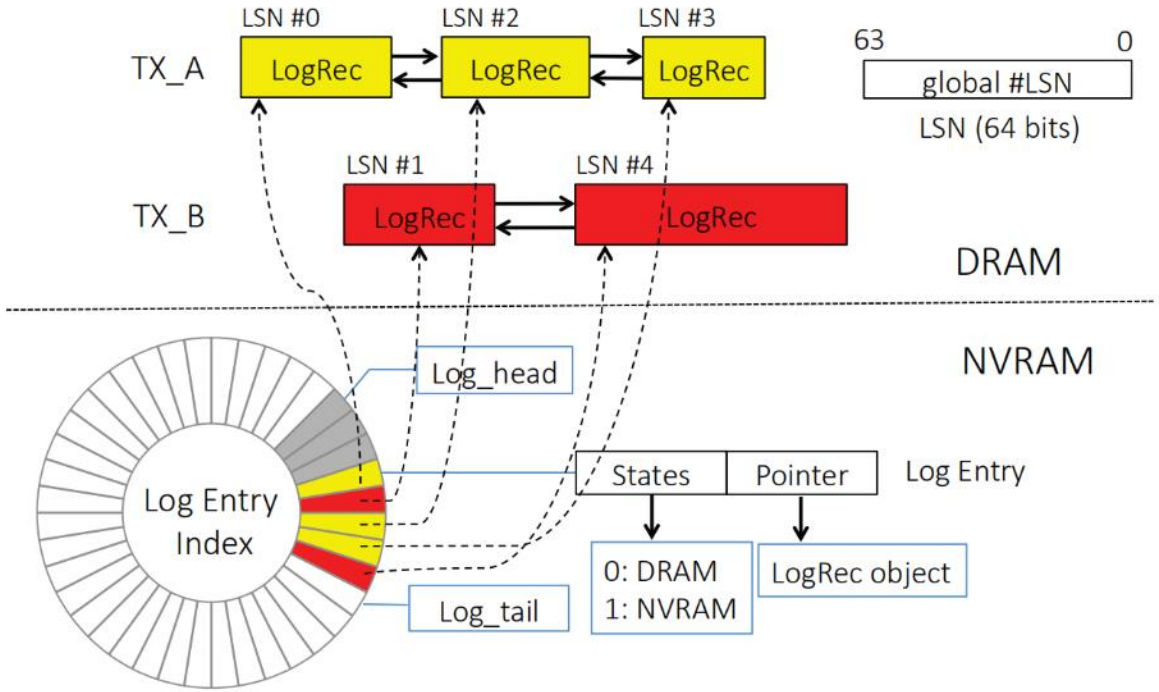




03 NV-logging







03 MARS

From ARIES to MARS: Transaction Support for Next-Generation, Solid-State Drives

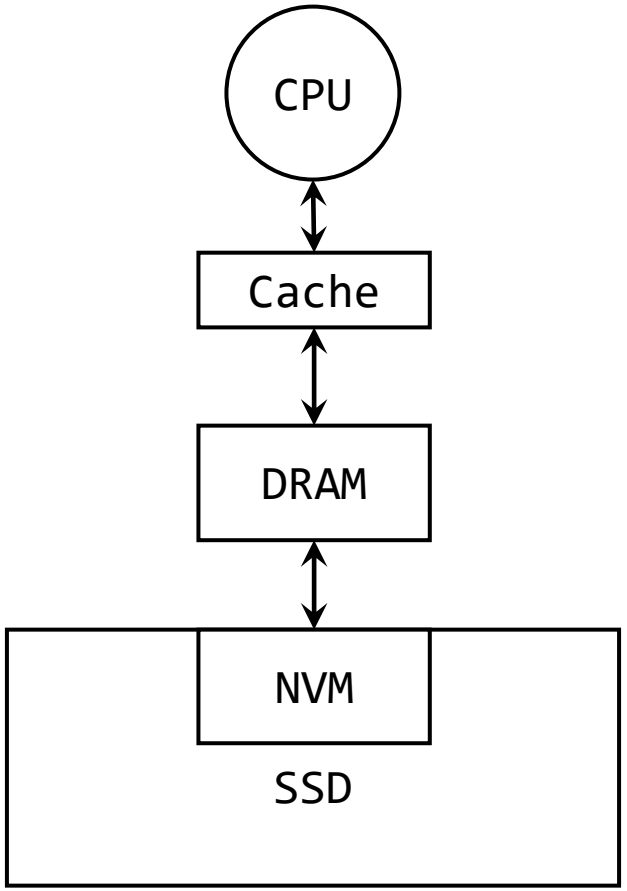
Aries问题:

- 针对原来DRAM-Disk两级系统设计，针对新兴的NVM
  性能提升不大
- 无法利用NVM字节寻址且持久化的特点

整体架构:

- NVM上保存Redo日志
- 采用no-force + no-steal策略
- 没有pages，直接操作对象
- 采用硬件与软件协同实现

本质上是一种Redo-Only的实现方式



03 MARS

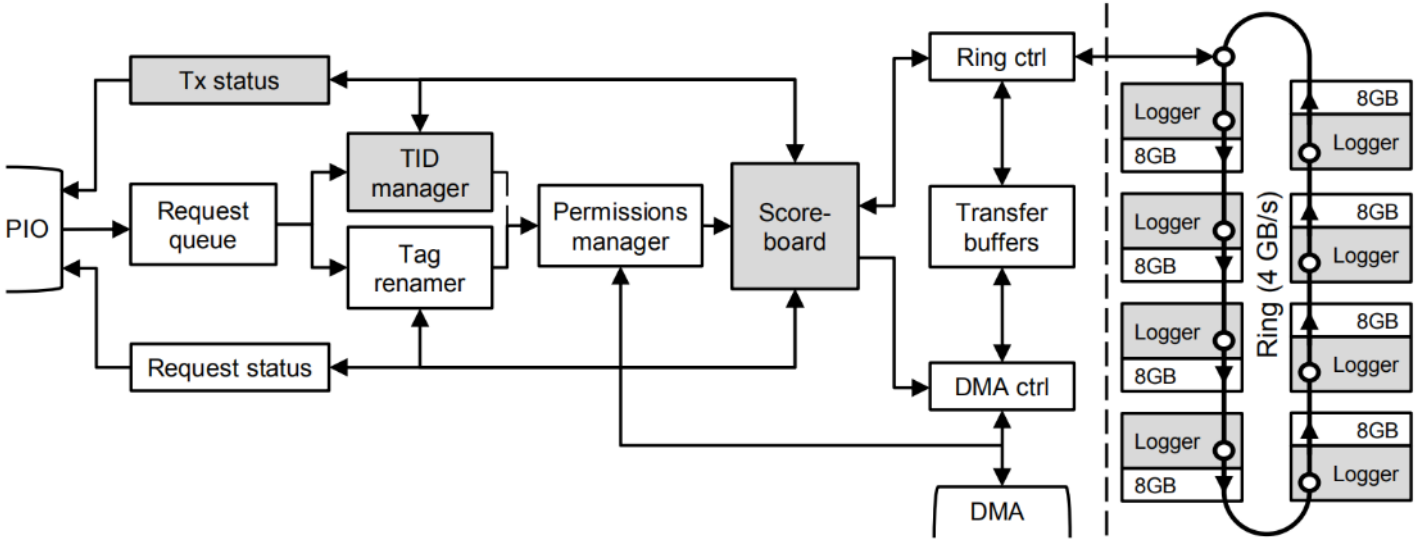
软硬件结合

硬件

1. TID manager：映射用户TID到具体TID
2. Score-Board：追踪事务，约束事务顺序
3. Tx status：事务状态

软件

1. EAW (editable atomic writes)
   - LogWrite、Commit接口实现redo日志可编辑与日志apply
2. 文件系统支持Log存储



Command	Description
LogWrite(TID, file, offset, data, len, logfile, logoffset)	Record a write to the log at the specified log offset. After commit, copy the data to the offset in the file.
Commit(TID)	Commit a transaction.
AtomicWrite(TID, file, offset, data, len, logfile, logoffset)	Create and commit a transaction containing a single write.
NestedTopAction(TID, logfile, logoffset)	Commit a nested top action by applying the log from a specified starting point to the end.
Abort(TID) PartialAbort(TID, logfile, logoffset)	Cancel the transaction entirely, or perform a partial rollback to a specified point in the log.

03 MARS

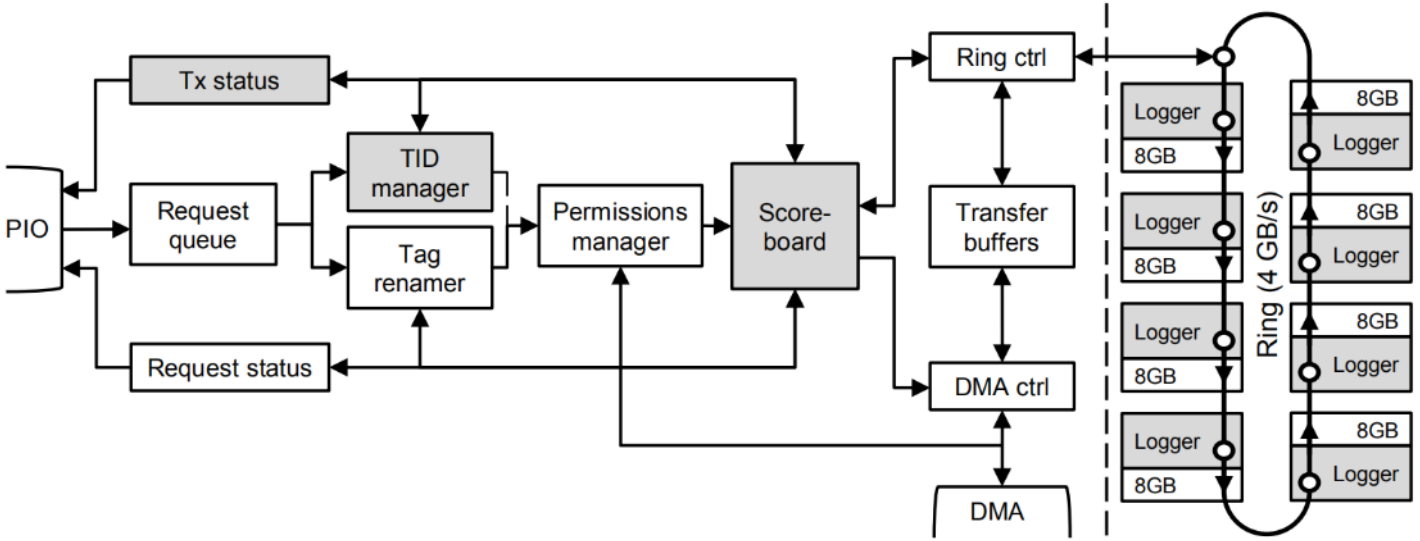
软硬件结合

硬件

1. TID manager：映射用户TID到具体TID
2. Score-Board：追踪事务，约束事务顺序
3. Tx status：事务状态

软件

1. EAW (editable atomic writes)
   - LogWrite、Commit接口实现redo日志可编辑与日志apply
2. 文件系统支持Log存储



Command	Description
LogWrite(TID, file, offset, data, len, logfile, logoffset)	Record a write to the log at the specified log offset. After commit, copy the data to the offset in the file.
Commit(TID)	Commit a transaction.
AtomicWrite(TID, file, offset, data, len, logfile, logoffset)	Create and commit a transaction containing a single write.
NestedTopAction(TID, logfile, logoffset)	Commit a nested top action by applying the log from a specified starting point to the end.
Abort(TID) PartialAbort(TID, logfile, logoffset)	Cancel the transaction entirely, or perform a partial rollback to a specified point in the log.

03 MARS

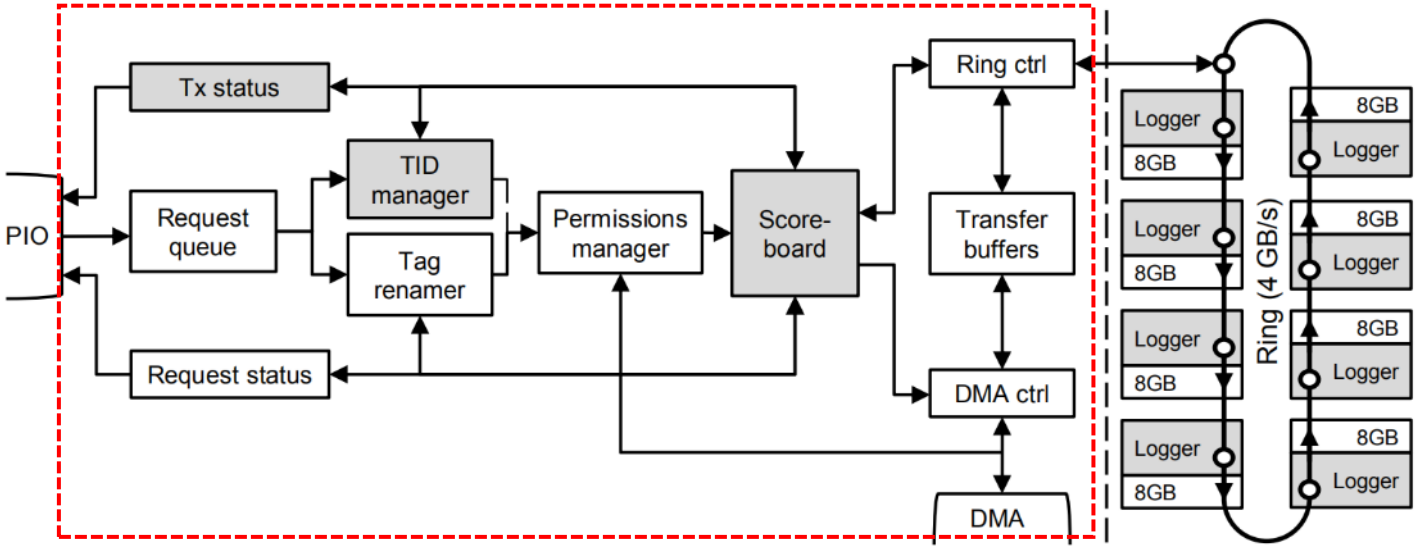
软硬件结合

硬件

1. TID manager：映射用户TID到具体TID
2. Score-Board：追踪事务，约束事务顺序
3. Tx status：事务状态

软件

1. EAW (editable atomic writes)
   - LogWrite、Commit接口实现redo日志可编辑与日志apply
2. 文件系统支持Log存储

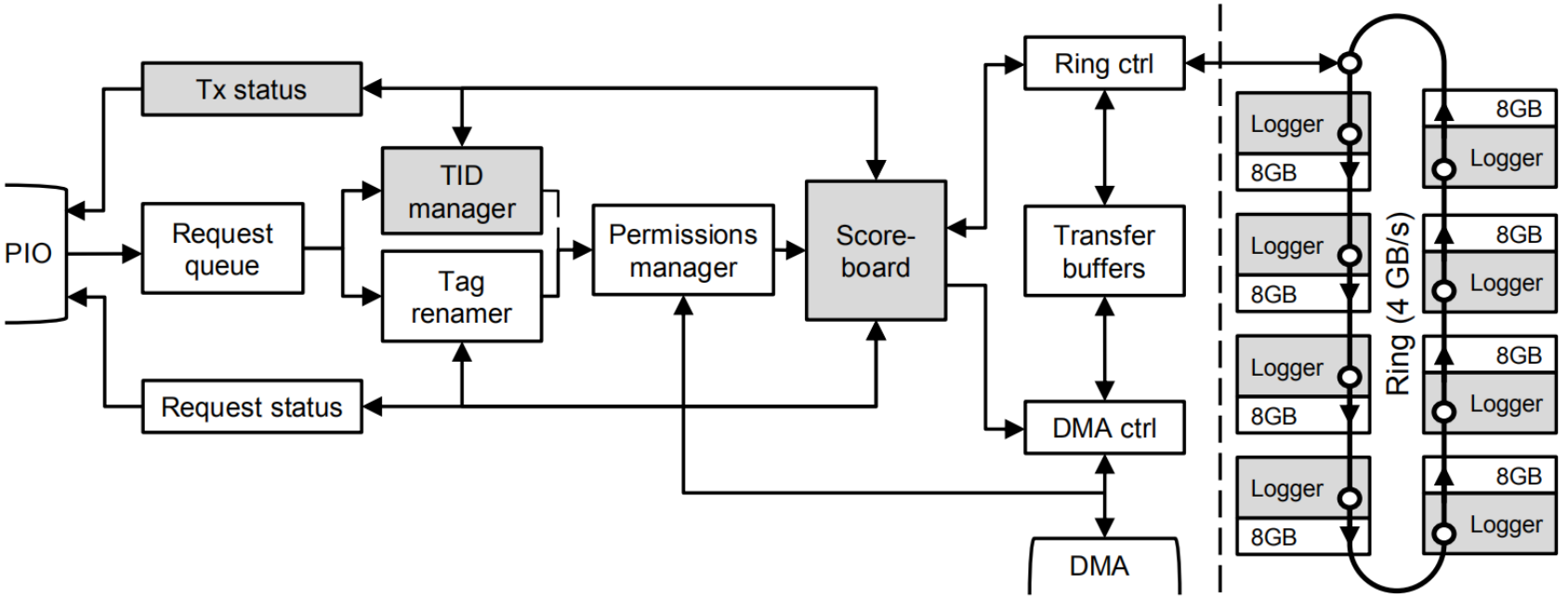


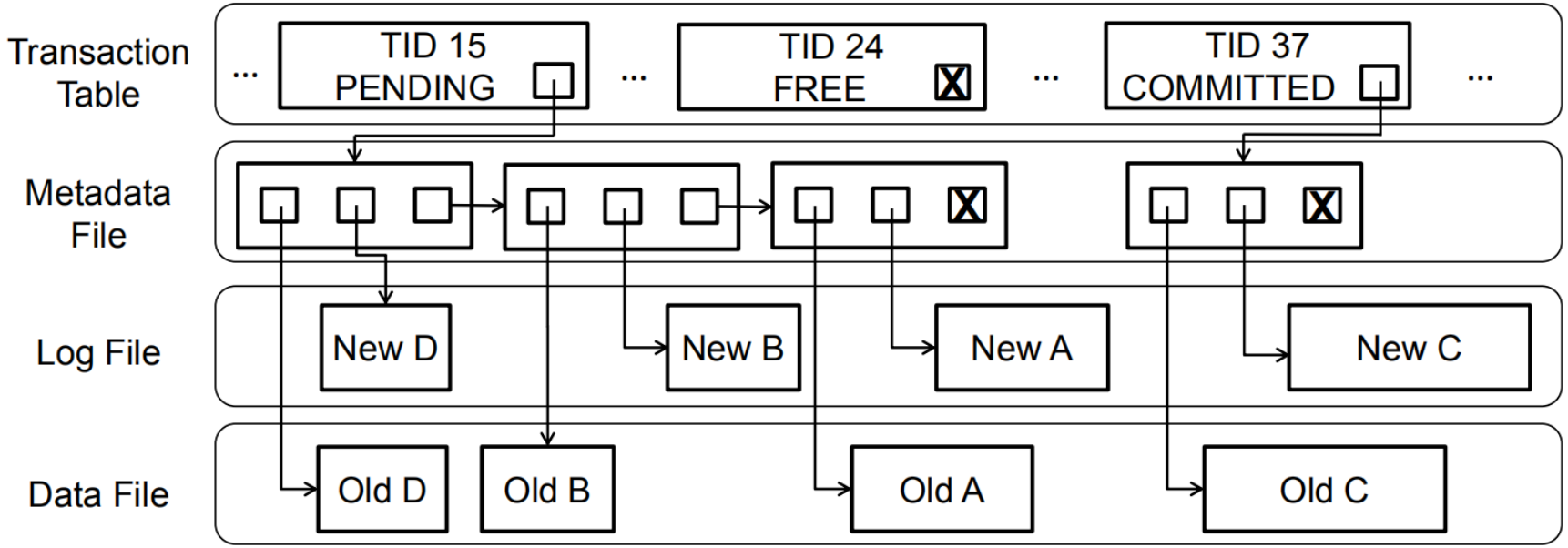
Command	Description
LogWrite(TID, file, offset, data, len, logfile, logoffset)	Record a write to the log at the specified log offset. After commit, copy the data to the offset in the file.
Commit(TID)	Commit a transaction.
AtomicWrite(TID, file, offset, data, len, logfile, logoffset)	Create and commit a transaction containing a single write.
NestedTopAction(TID, logfile, logoffset)	Commit a nested top action by applying the log from a specified starting point to the end.
Abort(TID) PartialAbort(TID, logfile, logoffset)	Cancel the transaction entirely, or perform a partial rollback to a specified point in the log.

03 MARS

事务提交

1. 开始事务
2. 通过LogWrite写入数据
3. 后续如果更改同一对象则直接修改log file
4. 用户commit

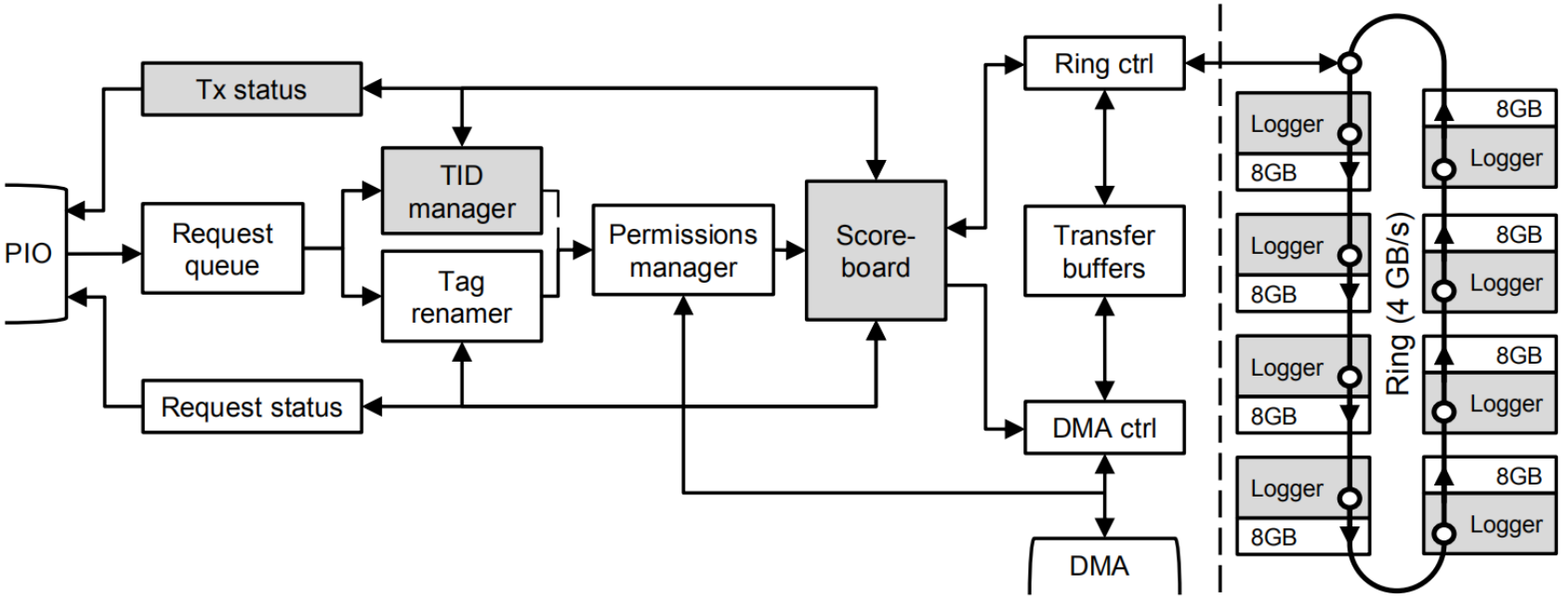




03 MARS

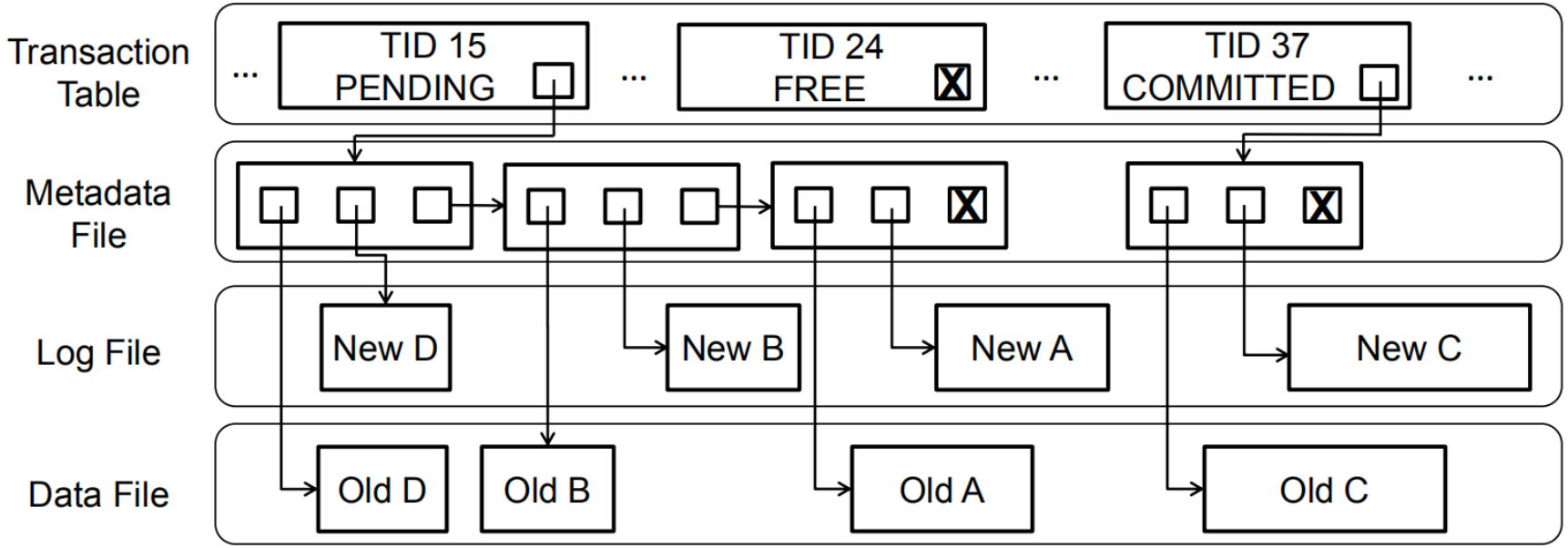
事务回滚

- 直接丢弃metadata file即可



恢复

1. 宕机恢复
2. 扫描各个logger的状态
3. 部分写入的回滚
4. 全部写入的直接commit



03 MARS

优缺点

优点:

- 使用多个logger并行刷入数据
- redo日志可编辑，进一步压缩日志量
- 充分利用NVM字节寻址特点

缺点:

- 硬件实现较为复杂，不易复现
- redo日志apply的时候断电，是否需要类似double write的方式保证对象写坏?

03 WBL

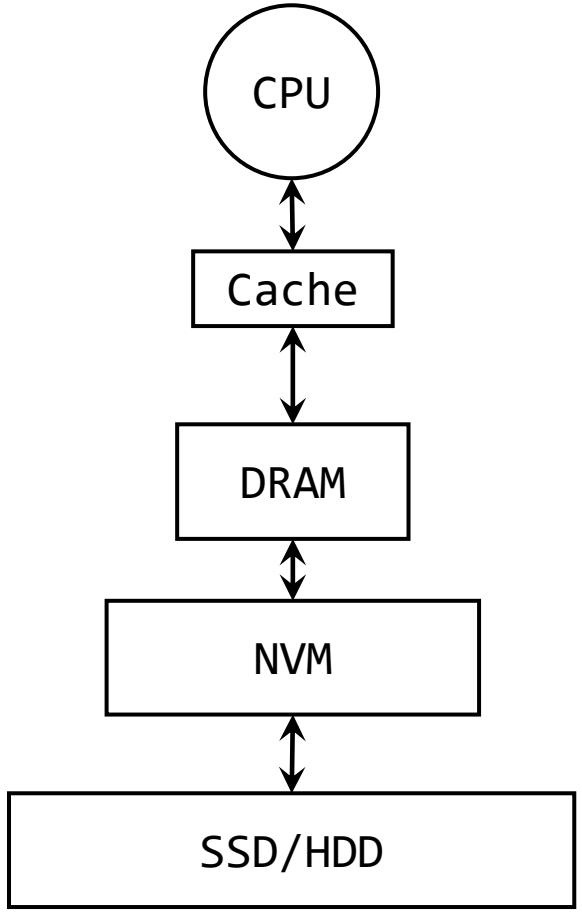
Write Behind Log

Aries问题:

- 原有算法针对块设备随机写与顺序写性能差别巨大而设计，NVM的顺序与随机则差别不大
- 无法利用NVM字节寻址且持久化的特点

整体架构:

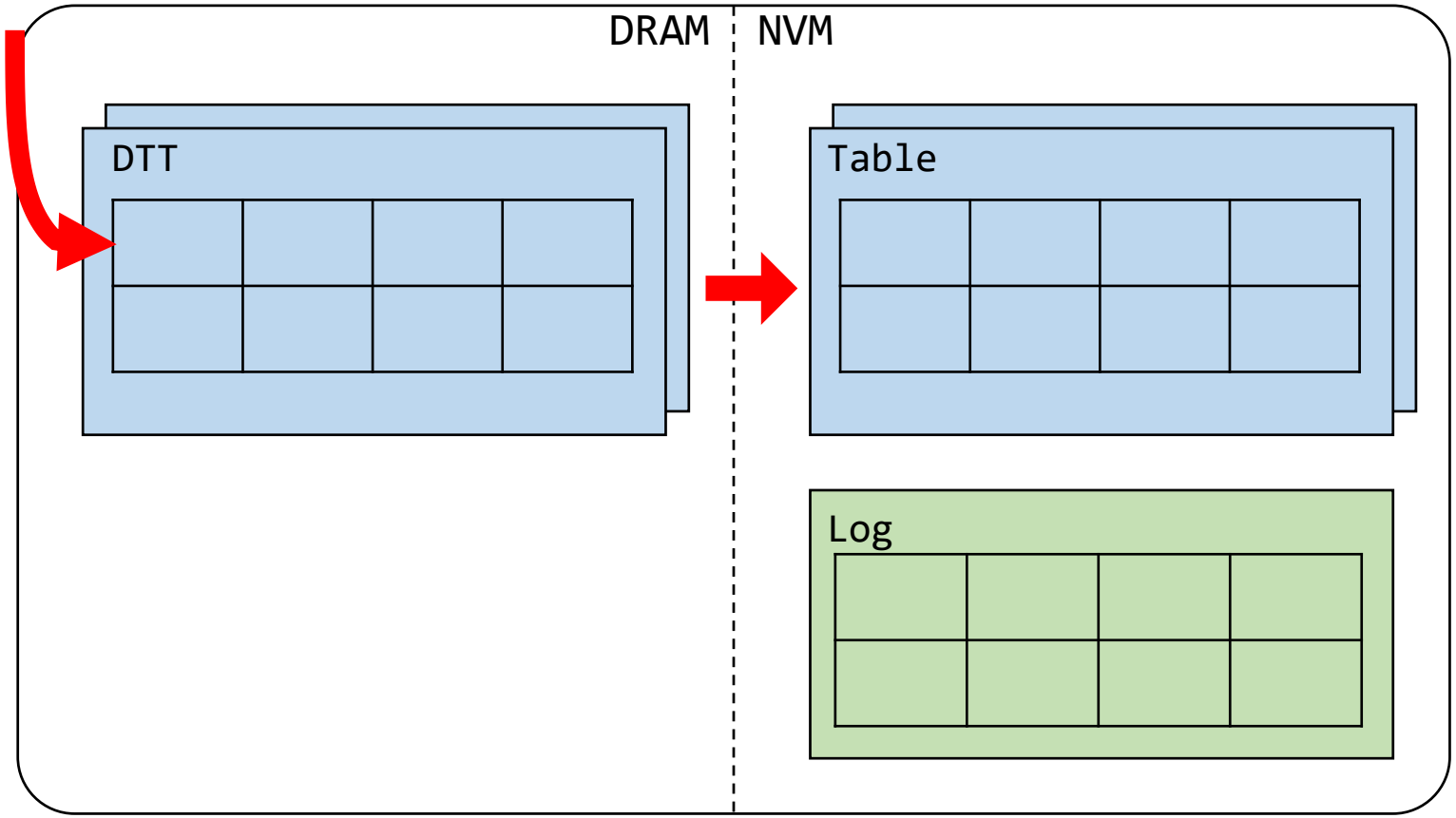
- DRAM与NVM直接交互
- NVM存放日志与tuple
- 先写入数据，再持久化Log



03 WBL

核心思路:

- 基于MVCC，无需复制原来的数据，对于新写入的数据在日志成功前不可见
- 写入即成功，无需redo log
- 仅保存undo log，并且需要维护的日志量极少



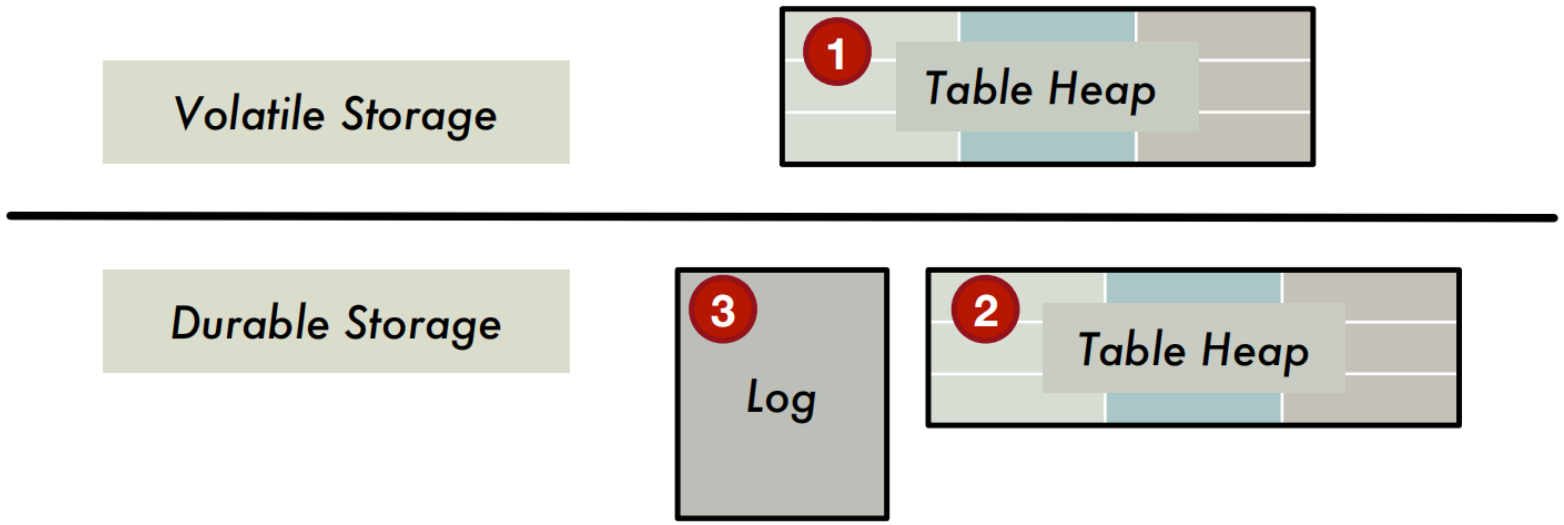
03 WBL

事务写入流程

1. 开始事务
2. 写入到DRAM中的tuple，完对数据库数据的修改和索引
3. 事务提交，内存DTT刷入到NVM中
4. NVM中记录log
5. 通知应用程序commit完成

Checksum	LSN	Log Record Type	Persisted Commit Timestamp(C_p)	Dirty Commit Timestamp(C_d)

(Cp, Cd)　　(Cl, (Cp, Cd))



03 WBL

事务恢复流程

回滚:

1．发现事务与其他事务冲突

2．事务管理器记录事务ID

3．后台回收线程负责垃圾回收写入的元组

4．后续事务对该事务时间戳的元组不可见

重启:

1．系统重启

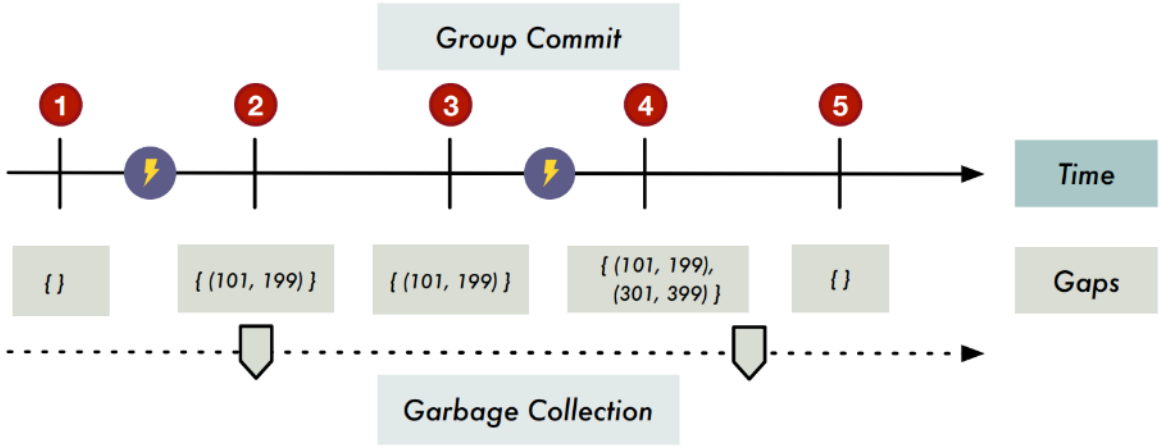
2．从前往后扫描日志

3．确定待undo的事务ID

4．同回滚3-4

LSN	WRITE BEHIND LOG
1	BEGIN CHECKPOINT
2	END CHECKPOINT (EMPTY CTG)
3	{ (1, 100) }
4	{ 2, (21, 120) }
5	{ 80, (81, 180) }
	SYSTEM FAILURE

Figure 12: WBL Example – Contents of the WBL during recovery.



03 WBL

元组可见性与单版本实现

可见性

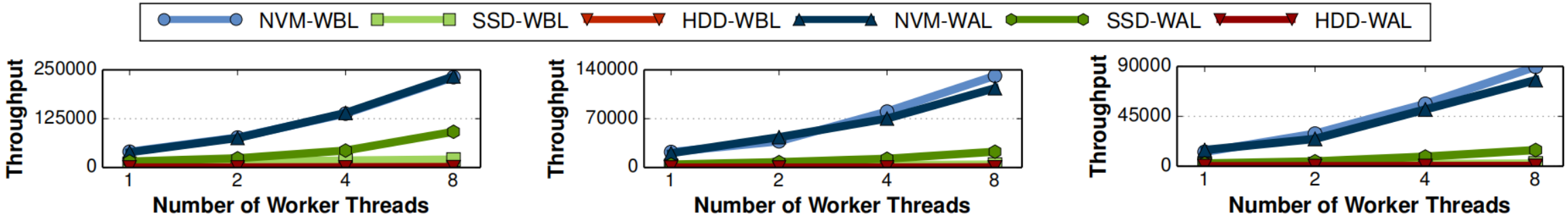
1. 回滚的事务
2. 重启之后位于（Cp，Cd）之间的事务与长事务
3. 与Last Cp之间的关系
    1. Cp之前的，除去长事务与回滚掉的事务，都是逻辑上可见的，
    2. Cp之后的，需要判断事务是否存活，与当前事务时间戳的关系

单版本

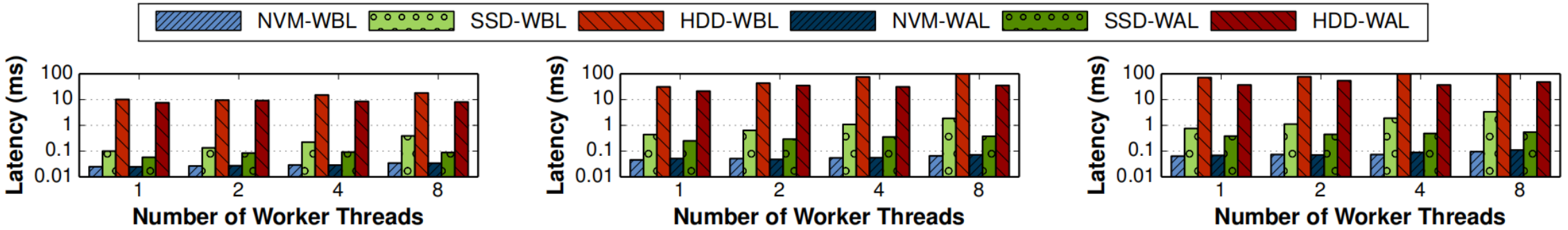
1. 先复制tuple
2. 写入日志
3. 根据时间戳判断事务可见性

03 WBL

性能



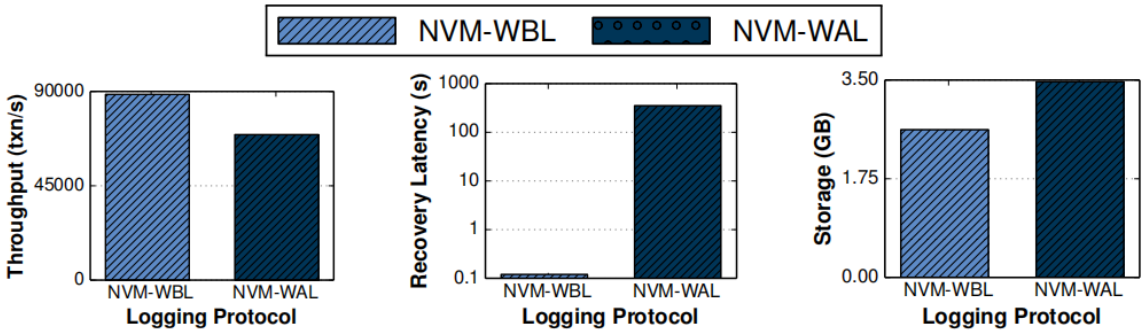
(a) Read-Heavy Workload (b) Balanced Workload (c) Write-Heavy Workload



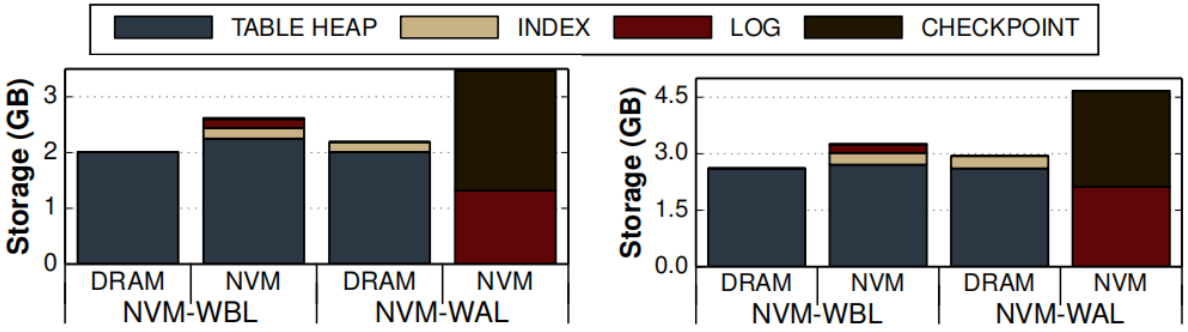
(a) Read-Heavy Workload (b) Balanced Workload (c) Write-Heavy Workload

03 WBL

性能



(a) Throughput (b) Recovery Time (c) Storage Footprint



(a) YCSB Storage Footprint (b) TPC-C Storage Footprint

Figure 2: WBL vs. WAL – The throughput, recovery time, and storage footprint of the DBMS for the YCSB benchmark with the write-ahead logging and write-behind logging protocols.

03 WBL

优缺点

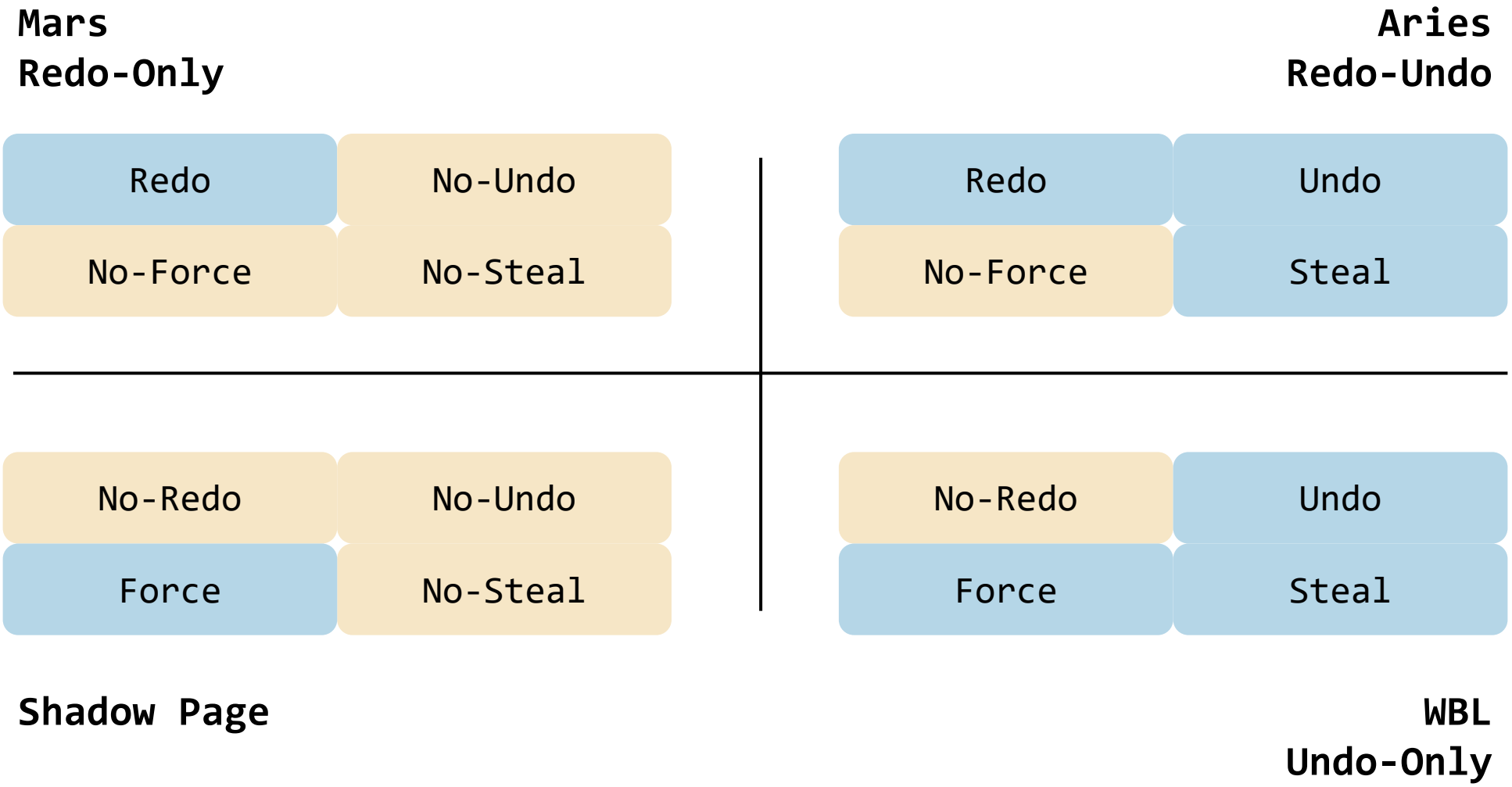
优点：

- 日志量少
- 恢复时间少
- 充分利用NVM特点

缺点：

- 垃圾回收复杂，导致可见性也比较复杂
- 数据和Log全部放在NVM上，需要后台线程不断垃圾回收

04 总结


Mars
Redo-Only
Aries
Redo-Undo
Redo
No-Undo
Redo
Undo
No-Force
No-Steal
No-Force
Steal
No-Redo
No-Undo
No-Redo
Undo
Force
No-Steal
Force
Steal
Shadow Page
WBL
Undo-Only

04 总结

	思路	优点	缺点
Aries	1. Redo-undo模式 2. Fuzzy checkpoint 3. No-force + Steal	1. 接口丰富，适合绝大多数场景 2. Redo + Undo获得最大程度并发量	1. 实现较为复杂 2. 写入放大较高
MARS	1. Redo-only模式 2. 软硬件协同实现 3. No-force + No-steal	1. 多个Logger模块并行加速写入 2. Redo日志可编辑，压缩日志空间	1. 硬件设计复杂 2. 是否需要引入double write机制
WBL	1. Undo-only模式 2. Group commit 3. Force + No-steal	1. 日志量极少 2. 恢复时间少 3. 充分利用NVM特点	1. 可见性较复杂 2. 对垃圾回收要求高

04 参考文献

1. http://catkang.github.io/2019/01/16/crash-recovery.html
2. https://courses.cs.washington.edu/courses/cse550/09au/papers/CSE550.GrayTM.pdf
3. https://cs.stanford.edu/people/chrismre/cs345/rl/aries.pdf
4. https://jianh.web.engr.illinois.edu/papers/jian-vldb15.pdf
5. http://mesl.ucsd.edu/pubs/SOSP2013-MARS.pdf
6. https://www.vldb.org/pvldb/vol10/p337-arulraj.pdf
7. https://github.com/CDDSCLab/Weekly-Group-Meeting-Paper-List/blob/main/meeting-summary/2020-09-25-Aries.md