

第7章 数据库原子性和持久性 实现及故障恢复

苏州大学 费子成

feizicheng@suda.edu.cn

<https://web.suda.edu.cn/feizicheng/>

大纲

1. 事务原子性和持久性的实现
2. 数据库故障恢复机制概述
3. 单机系统崩溃恢复方法
4. ARIES恢复算法
5. 数据库备份技术
6. 数据库多机恢复
7. 小结

目录

- 1. 事务原子性和持久性的实现**
2. 数据库故障恢复机制概述
3. 单机系统崩溃恢复方法
4. ARIES恢复算法
5. 数据库备份技术
6. 数据库多机恢复

事务原子性和持久性

□ 事务的**原子性**规定了事务是一个不可再分的基本操作单元。

- “事务全部成功或者失败”就是事务原子性的要求。“不可再分”这一性质则意味着对于不管是单条语句还是多条语句组合的事务，其中所有语句需要同时生效，或者同时被数据库拒绝。
- 数据库在事务的执行过程中发生故障而无法完成事务，也需要消除已经完成的部分语句的影响，使数据库**不会处于只有部分语句执行成功的状态**。

□ 事务的**持久性**：一旦某个事务提交以后，即使数据库发生故障，该事务的**执行结果也不会丢失**，可以被正确地恢复，仍然对后续事务可见。

事务原子性和持久性的实现

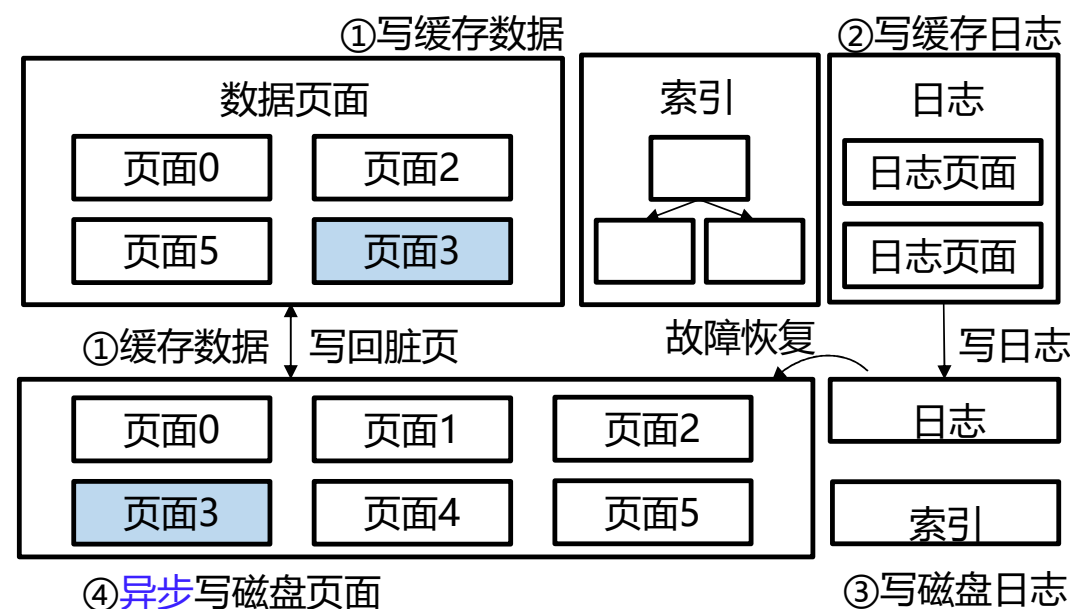
□ 故障系统重启后，内存数据丢失，磁盘数据不丢失

□ 原子性

- ① 事务运行期间不刷盘，故障系统重启后自动保证原子性；
- ② 事务运行期间刷盘，故障系统重启需回滚该事务

□ 持久性

- ① 事务完成 (commit、abort) 时刷盘，故障系统重启后自动保证持久性；
- ② 事务完成时不刷盘，故障系统重启后需重做该事务



数据库故障类别

- ❑ 事务故障：数据库事务因为资源冲突或者死锁等原因导致执行失败。
- ❑ 系统崩溃：数据库自身或操作系统的故障导致数据库进程意外退出。
- ❑ 磁盘故障：数据因为磁盘（其他非易失性存储）损坏导致无法被读取。
- ❑ 自然灾害：自然灾害对数据库系统所在的环境造成了彻底性破坏。

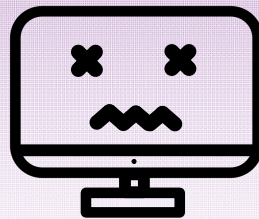


执行失败了!

事务故障



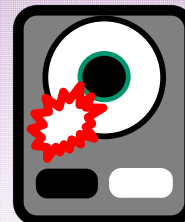
原子性



系统崩溃



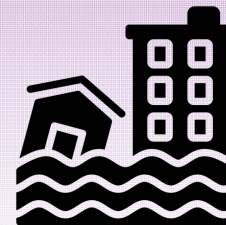
原子性、持久性



磁盘故障



持久性



自然灾害



持久性

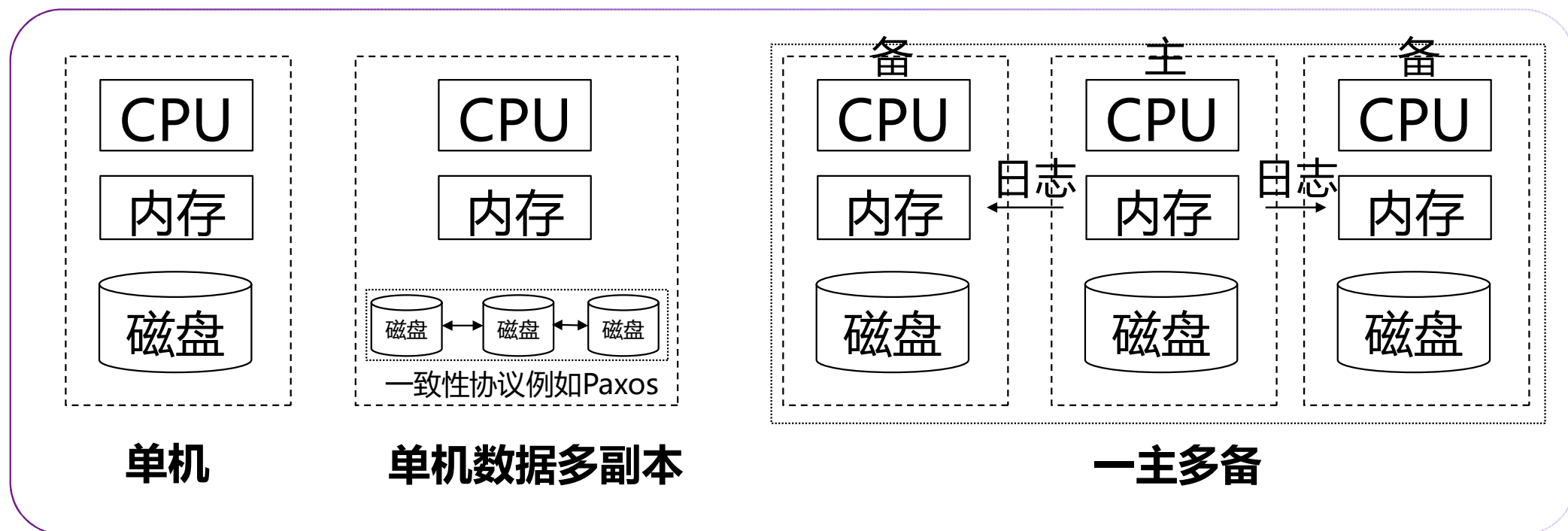
目录

1. 事务原子性和持久性的实现
- 2. 数据库故障恢复机制概述**
3. 单机系统崩溃恢复方法
4. ARIES恢复算法
5. 数据库备份技术
6. 数据库多机恢复

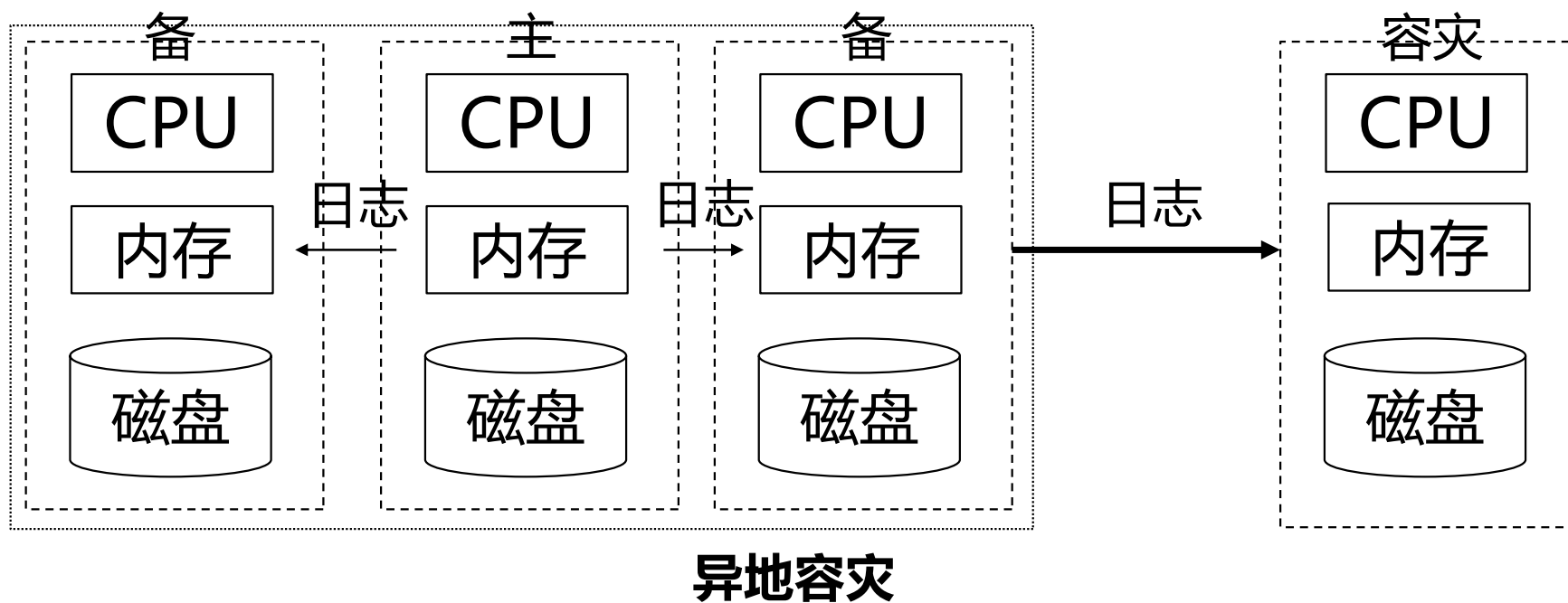
数据库恢复机制架构

- **无故障事务回滚（例如账户小于0）：影响原子性，撤销该事务已做操作**
- **故障失误回滚（例如死锁杀死事务）：影响原子性，撤销该事务已做操作**
- **系统故障（例如重启）：内存数据丢失，影响原子性和持久性**
 - 原子性：撤销未结束（不带Commit、Abort标记）的事务
 - 持久性：重做已经结束（带Commit或Abort标记）的事务
- **系统崩溃不能重启：不能提供服务，影响持久性**
 - 一主多备：主备之间通过日志保持一致性，发生故障后切换到其他系统
- **磁盘故障：磁盘数据丢失，影响持久性**
 - 磁盘数据多副本；数据备份机制：创建数据备份、日志备份
- **自然灾害：系统宕机不能重启，影响持久性**
 - 异地多机容灾：多机实时传输日志保证一致性，发生故障后切换到其他系统

数据库恢复机制架构



数据库恢复机制架构



数据库恢复机制对应关系

问题类型	出现频率	对事务的影响	解决方法
无故障下事务回滚	高	原子性	单机数据库恢复
事务故障	较高	原子性	
系统崩溃能重启	中等	原子性/持久性	
磁盘故障	低	持久性	数据多副本
系统崩溃不能重启	低	持久性	一主多备
自然灾害	极低	持久性	异地多机恢复

Durability → High Availability
磁盘 → 多机高可用

高可用指标

□通用高可用指标：

- 平均故障间隔时间 MTBF (Mean Time between Failures)：系统在两相邻故障间隔期内正确工作的平均时间
- 平均恢复时间 MTTR (Mean Time to Repair)：系统平均从故障中恢复需要的时间。
- 平均损坏时间 MTTF (Mean Time to Failure)：系统出现损坏的平均时间。

□数据库容灾指标：

- 恢复点目标 RPO (Recovery Point Objective)：业务系统在系统故障后所能容忍的数据丢失量。
- 恢复时间目标 RTO (Recovery Time Objective)：业务系统所能容忍的业务停止服务的最长时间。
- 一般用n个9来表示
 - 例如四个九99.99%表示一年99.99%时间可用，即不可用时间为 $365*24*60*0.01\%=52.56$ 分钟

目录

1. 事务原子性和持久性的实现
2. 数据库故障恢复机制概述
- 3. 单机系统崩溃恢复方法**
4. ARIES恢复算法
5. 数据库备份技术
6. 数据库多机恢复

系统崩溃恢复

□系统崩溃诱因

- 数据库进程被操作系统中止
- 管理员错误操作
- 软件故障、死锁

□特点

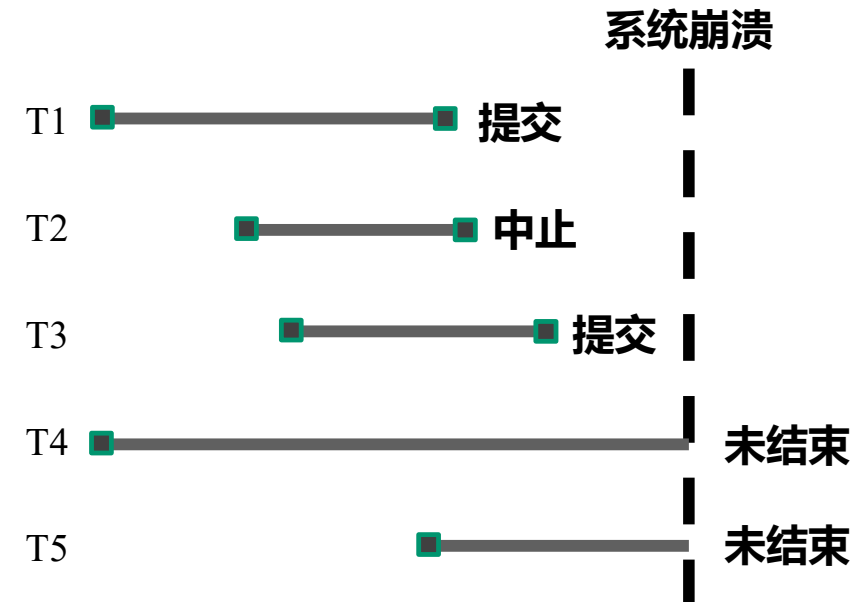
- 出现频率较高
- 缓冲区数据丢失、磁盘数据不完整

□影响

- 原子性
- 持久性

系统崩溃恢复

- ❑ **脏页**：内存页面已更新，磁盘页面未更新
- ❑ **刷脏**：将内存脏页刷到磁盘
- ❑ T1、T3在崩溃时刻前已经提交
 - 如果**未刷脏**，则影响持久性，需要**重做**
 - 如果**已刷脏**，则不影响持久性
- ❑ T2在崩溃时刻前已经中止（已经完成了回滚）
 - 如果期间刷过脏，但是abort时未刷脏，则影响持久性，需要**重做**
 - 如果已刷脏，则不影响持久性
- ❑ T4、T5在崩溃时刻前未结束
 - 如果已刷脏，则影响原子性，需要**回滚**
 - 如果未刷脏，则不影响原子性



	已刷脏	未刷脏
提交Commit	😊	重做
中止Abort	😊	重做
未完成	回滚	😊

系统崩溃下的事务

□ 已完成的事务

- 已提交的事务（Commit标记）：它们对数据库的修改可能没有写回磁盘，缓冲区数据丢失后这些修改无法找回。事务的持久性受到了影响，需要重做这些事务。
- 已中止的事务（Abort标记）：系统对这些事务的撤销可能没有写回磁盘，因此在重启之后这些撤销内容会丢失。事务的持久性受到了影响，需要重做这些事务。

□ 未完成的事务（只有Start，没有Commit、Abort标记）：它们可能已经对数据库造成了修改，但是没有被系统提交，重启之后的数据库也没有撤销这些修改。事务的原子性受到了影响，需要回滚这些事务

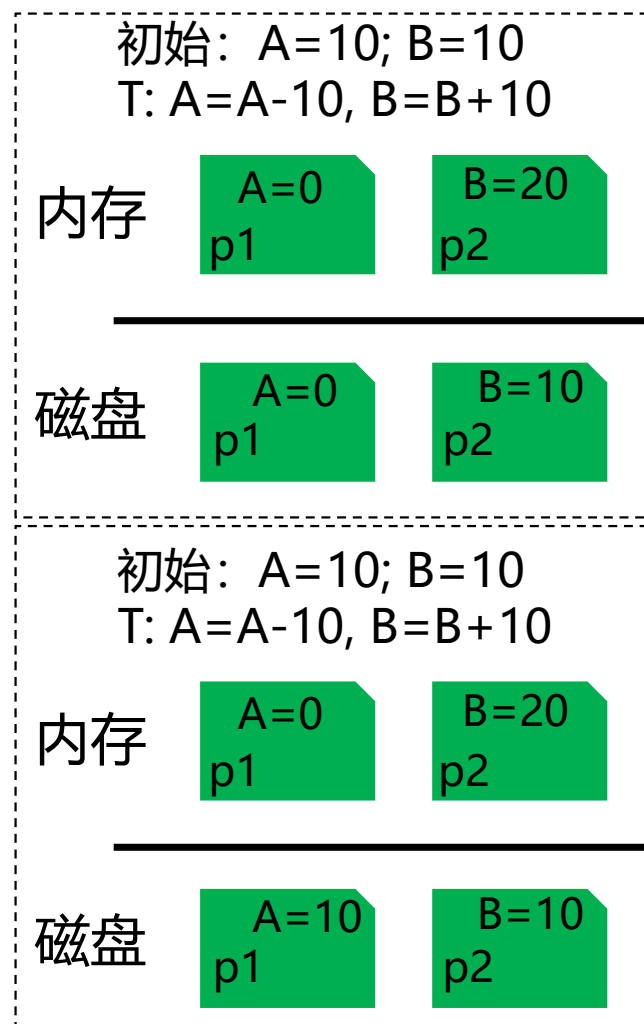
崩溃恢复策略设计

□ 原子性保证

- 选择①: NO-STEAL (非窃取)
 - 未结束事务不能将脏页写入磁盘, 不存在原子性问题
- 选择②: STEAL (窃取)
 - 未结束事务能将脏页写入磁盘, 影响原子性, 需要回滚

□ 持久性保证

- 选择①: Force (强制)
 - 已完成事务强制将脏页写入磁盘, 不存在持久性问题
- 选择②: No-Force (非强制)
 - 已完成事务不强制将脏页写入磁盘, 影响持久性, 需要重做



保证持久性和原子性的方案选择

□ FORCE条件：事务完成强制刷脏 vs 不强制刷脏

- Force缺点：每次事务提交都必须刷新脏页，消耗大量IO读写资源
- 解决方法：使用NO-FORCE模式，利用Redo日志重做事务
- Redo日志：记录事务对数据库的所有影响

□ NO-STEAL条件：事务中间可以刷脏 vs 不可以刷脏

- No-Steal缺点：事务执行过程中不能刷新磁盘，必须占有较大的缓冲区空间，不利于多个事务的并发执行
- 解决方法：使用STEAL模式，利用Undo日志撤销事务
- Undo日志：记录撤销事务所需的内容

数据库恢复算法分类

要解决原子性和持久性问题，各分别存在两种方法，两两组合起来产生四种数据库恢复算法。

	FORCE 强制（事务提交强制刷盘）	NO-FORCE非强制（事务提交非强制刷盘）
NO-STEAL非窃取 (执行期间不刷盘)	无redo日志 无undo日志	有redo日志 无undo日志
STEAL窃取 (执行期间可刷盘)	无redo日志 有undo日志	有redo日志 有undo日志

数据页面写回磁盘时机

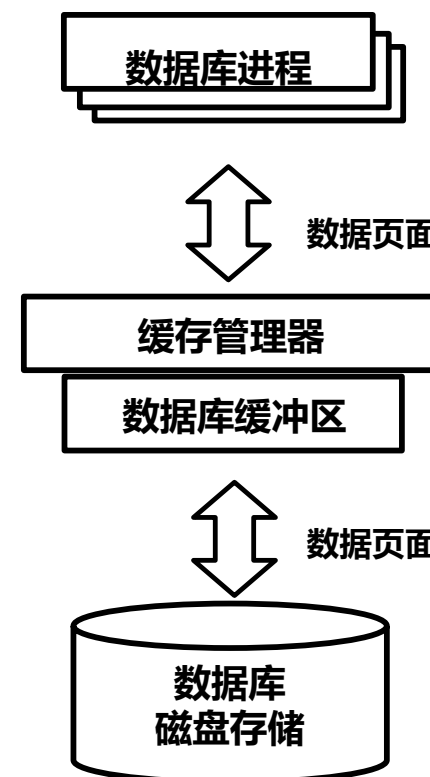
□ 同步写内存数据页面，但是异步写回磁盘

- 脏页：内存修改但是未写回磁盘的页面
- 刷脏：异步将脏页写回磁盘

□ 刷脏的时机：

- 数据库关闭时，缓冲区中的所有脏页需要写回磁盘
- 缓冲区中的数据页面已经满了，如果还需要继续读入数据页面，就必须将被替换的脏页写回磁盘
- 数据库会设置一个单独线程定时刷脏
 - 全量、增量

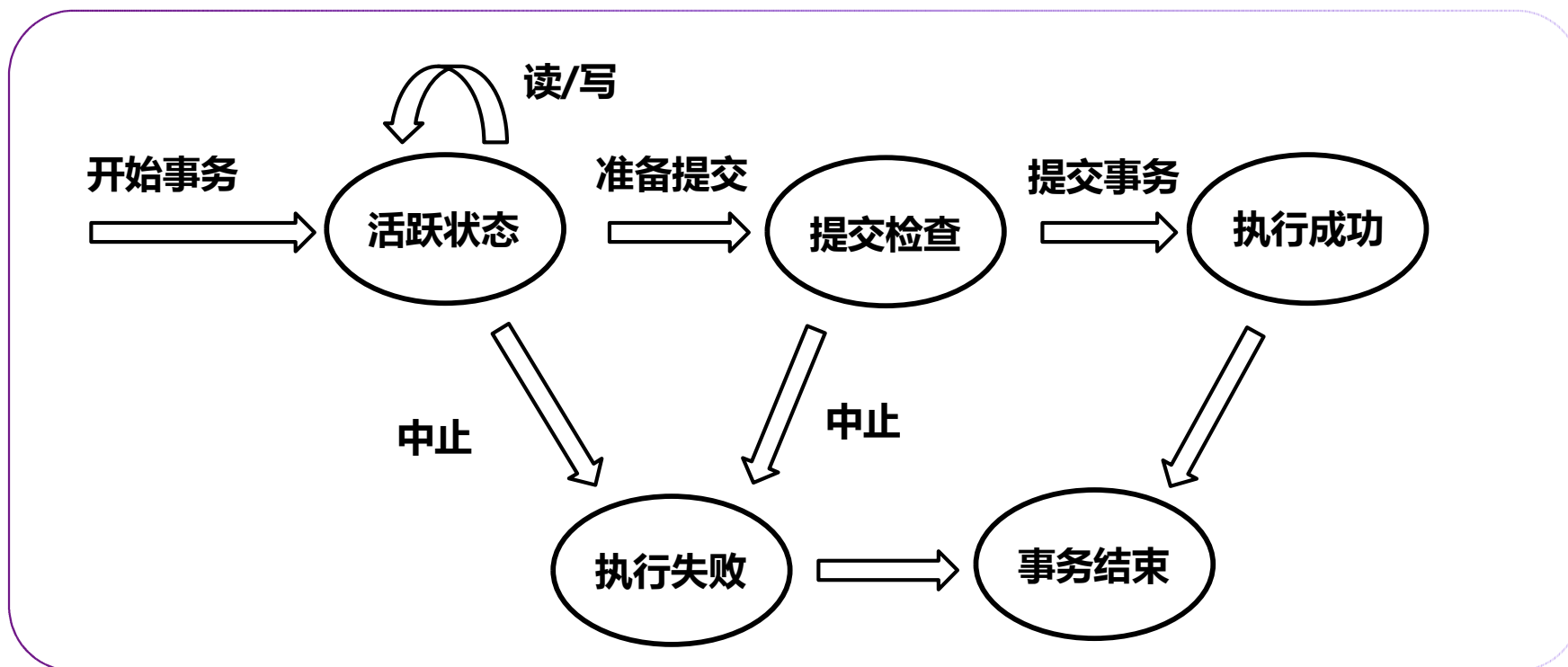
□ 难点：故障时如何恢复？需要回放（重做或回滚）哪些日志？



数据库日志

- ❑ 数据库日志是数据库系统内一系列执行事件的记录，它与数据库事务是密切相关的，事务的执行过程会反映在日志中，数据库可以通过对日志的分析实现对事务的回滚（原子性）或重做（持久性）。
- ❑ 日志是日志记录（log record）的序列。日志记录是数据库系统活动记录的最小单位，每一条记录反映了数据库系统的一次操作。日志记录不仅包含了数据的更新，还包含了数据库事务开始/结束的逻辑。
- ❑ 日志的内容在写入磁盘以后是不会被修改的，因此所有的日志内容可以顺序写入磁盘，这保证了高效的写入速度。该特性也是建立高可用恢复机制的前提。即数据的随机读写转换为日志的连续读写。

事务控制日志记录



事务开始日志记录 < START T >

事务提交日志记录 < COMMIT T >

事务中止日志记录 < ABORT/ROLLBACK T >

Undo回滚日志

□ 格式: $\langle T, X, v_{old} \rangle$

- T : 事务的唯一标识符
- X : 数据项
- v_{old} : 数据项修改以前的值

□ 产生时机: 当数据 T 修改数据项 X (Write(X)) 时产生

□ 作用: 实现事务回滚

□ 注意: 一般还包含一个日志序号log sequential number (LSN)

Undo回滚日志

□事务 T_0 读写操作

read(A);
 $A = A - 200$;
write(A);
read(B);
 $B = B + 200$;
write(B).

数据项变化

A: 1500 \rightarrow 1300
B: 2000 \rightarrow 2200

□事务 T_1 读写操作

read(C);
 $C = C + 500$;
write(C).

数据项变化

C: 500 \rightarrow 1000

$\langle T_0 \text{ start} \rangle$
 $\langle T_0, A, 1500 \rangle$
 $\langle T_0, B, 2000 \rangle$
 $\langle T_0 \text{ commit} \rangle$
 $\langle T_1 \text{ start} \rangle$
 $\langle T_1, C, 500 \rangle$
 $\langle T_1 \text{ commit} \rangle$

Redo重做日志

□ 格式: $\langle T, X, v_{new} \rangle$

- T : 事务的唯一标识符
- X : 数据项
- v_{new} : 数据项修改以后的值

□ 产生时机: 当数据 T 修改数据项 X (Write(X)) 时产生

□ 作用: 实现事务重做

Redo重做日志

□ 事务 T_0 读写操作

read(A);
A = A - 200;
write(A);
read(B);
B = B + 200;
write(B).

数据项变化

A: 1500 → 1300
B: 2000 → 2200

□ 事务 T_1 读写操作

read(C);
C = C + 500;
write(C);

数据项变化

C: 500 → 1000

< T_0 start >
< T_0 , A, 1300 >
< T_0 , B, 2200 >
< T_0 commit >
< T_1 start >
< T_1 , C, 1000 >
< T_1 commit >

预写日志WAL

- ❑ 日志只有在持久存储中才能发挥作用
- ❑ 数据库日志需要满足预写日志（Write Ahead Logging，简称WAL）条件，即**日志必须比数据更早的写入磁盘**
- ❑ 日志写回磁盘的顺序必须和日志生成的时间相一致
- ❑ 对于事务原子性保证，每当页面写回磁盘时，**和事务相关的undo日志需要先写回磁盘**
- ❑ 对于事务持久性保证，每当事务提交的时候，**和事务相关的redo日志需要先写回磁盘**

日志实现方式

□按照功能分类日志

- Undo回滚日志
- Redo重做日志

□按照性质分类日志

- 物理日志
- 逻辑日志
- 物理逻辑日志

日志记录方案

□ 逻辑日志

- 记录事务中高层抽象的逻辑操作
- 举例：记录日志中UPDATE、DELETE和INSERT的文本信息
 - 例如小明的年龄由20改成21。

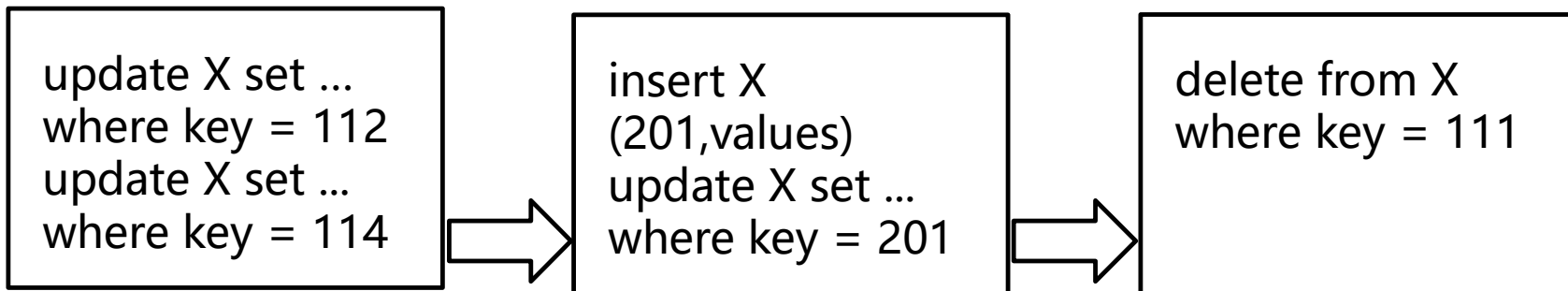
□ 物理日志

- 记录数据库具体物理变化
- 举例：记录一个被查询影响的数据项前后的值
 - 例如第10个页面第100偏移量的值由20改成21。

逻辑日志

□ 逻辑日志

- 记录事务中高层抽象的逻辑操作
- 举例：记录日志中UPDATE、DELETE和INSERT的文本信息

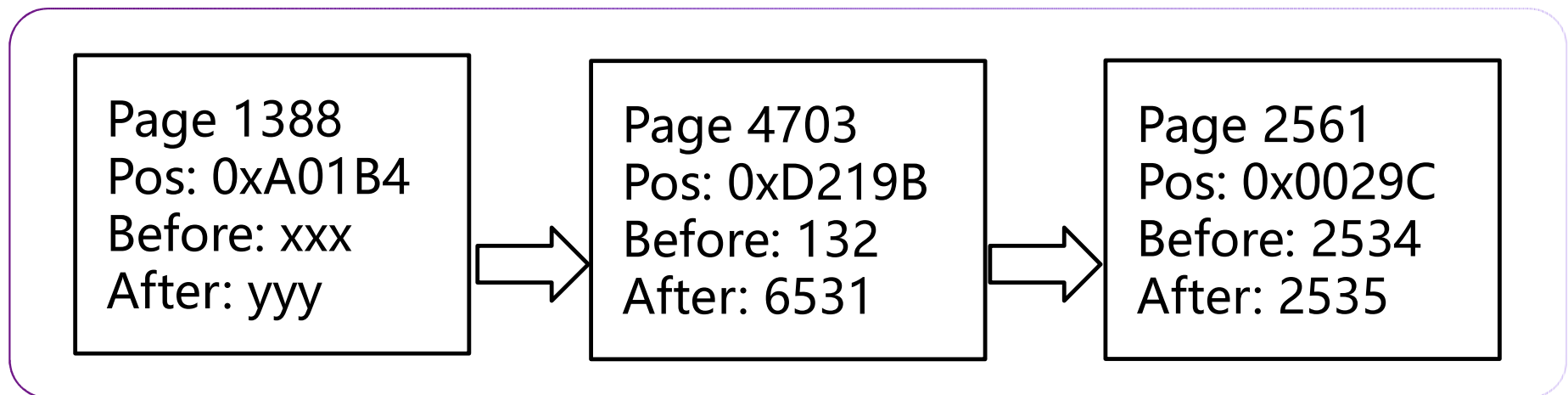


逻辑日志序列

物理日志

□ 物理日志

- 记录数据库中数据项的具体物理变化
- 举例：记录一个被查询影响的数据项前后的值



物理日志序列

物理逻辑日志

□物理逻辑日志

- 一种结合了物理日志和逻辑日志混合方法
- 日志记录中包含了数据页面的物理信息，但是页面以内目标数据项的修改信息则是以逻辑方式记录
 - 例如第100个页面（物理）的小明年龄值由20改成21（逻辑）

三种日志对比

UPDATE Student SET Sname= "Mike" WHERE Sno = "1" ;

物理日志

```
< T1,  
  Table= Student,  
  Page=99,  
  Offset=4,  
  Before=James,  
  After=Mike >  
< T1,  
  Index=X_PKEY,  
  Page=45,  
  Offset=9,  
  Key=(1,Record1) >
```

逻辑日志

```
< T1,  
  Query= "UPDATE  
    Student SET Sname  
    = Mike WHERE Sno =  
    1" >
```

物理逻辑日志

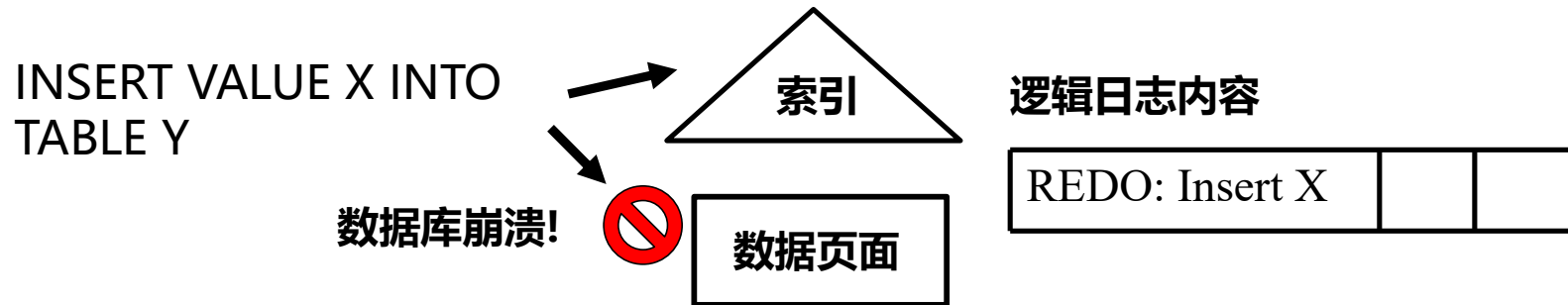
```
< T1,  
  Table= Student,  
  Page=99,  
  ObjectId=1,  
  Before=James,  
  After=Mike >  
< T1,  
  Index=X_PKEY,  
  IndexPage=45,  
  Key=(1,Record1) >
```

物理/逻辑日志性质比较

□ 有关数据库日志的三个重要性质：

- 幂等性：一条日志记录无论执行一次或多次，得到的结果都是一致的。
 - 例如： $x=x+1$ 不幂等； $x=0$ 幂等
 - 物理日志满足幂等性；逻辑日志不满足
- 失败可重做性：一条日志执行失败后，是否可以重做一遍达成恢复目的。
 - 例如：插入一条记录失败，再次插入成功。
 - 物理日志满足失败可重做性；逻辑日志不满足：例如插入数据页面成功，而插入索引失败，重做插入这个逻辑日志失败。
- 操作可逆性：逆向执行日志记录的操作，可以恢复原来状态（未执行这批操作时的状态）
 - 例如第10个页面第100偏移量的值由20改成21，逆操作由21改成20
 - 物理日志不可逆（页面偏移量位置可能被后续记录修改），逻辑日志可逆。

逻辑日志缺陷



- ✓ 逻辑日志不具有幂等性
- ✓ 逻辑日志不具有失败可重做性
 - 一条逻辑日志记录可能对应多项数据修改（表、索引），崩溃时X对索引造成影响，但没有影响数据页面。
- ✓ 是否能用逻辑日志重做日志记录？
 - 重做/不重做都有问题！
- ✓ 因此redo日志须用物理日志

物理日志缺陷

数据页面10

P1	100	P2	300	P3	500	P4	700	P5
----	-----	----	-----	----	-----	----	-----	----

旧数据页面日志

A: 0x00030

P1	100	P2	300	P3
----	-----	----	-----	----

P4	400	P5	500	P6	700	P7
----	-----	----	-----	----	-----	----

新数据页面

A: 0x00020

问题：无法找到
数据项500的位
置

- ✓ 物理日志不具有可逆性，无法处理数据项位置变化的情况
 - 页面分裂后A的地址发生了变化，撤销的时候无法定位数据项
- ✓ 因此物理日志一般不能用于回滚，Undo日志须用逻辑日志

日志性质比较及使用场景

	解析速度	日志量	可重做性	幂等性	可逆性	应用场景
物理日志	快	大	是	是	否	重做日志
逻辑日志	慢	小	否	否	是	撤销日志
物理逻辑日志	较快	中	是	否	否	撤销日志

注：物理逻辑日志用于回滚时，特别是索引页面分裂，可通过页面前后指针来完成回滚。

数据库恢复算法概述

□ 解决系统崩溃后事务原子性/持久性的问题

□ 每种问题对应于两种解决方法：
是否使用redo、undo日志

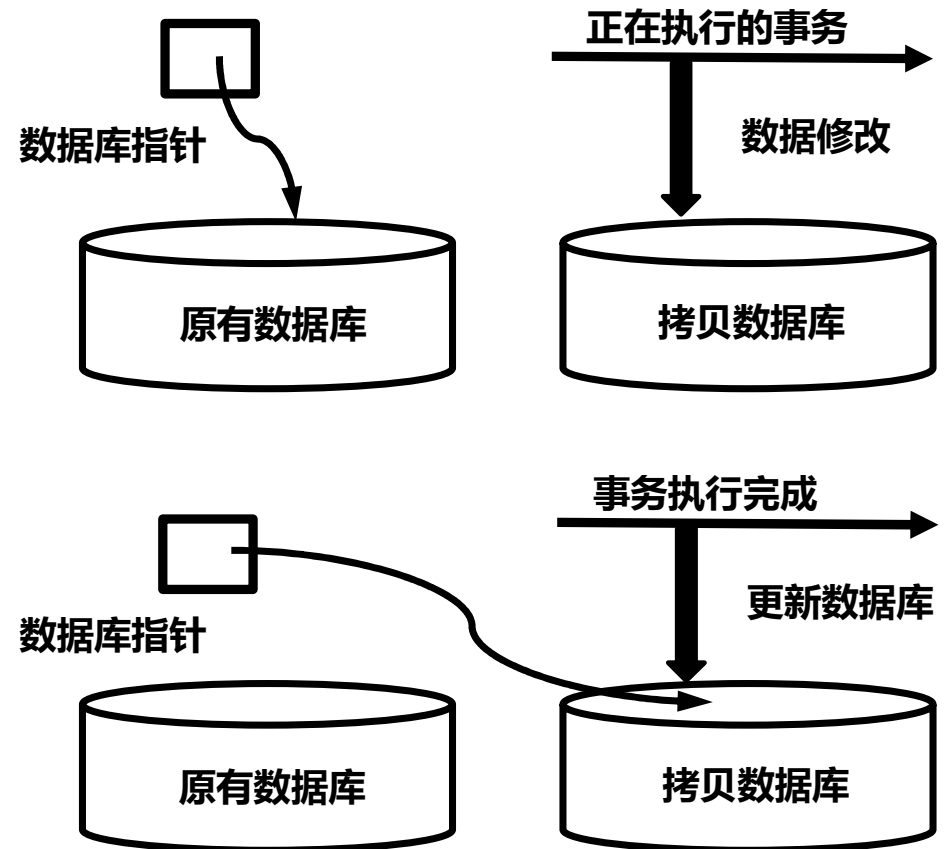
□ 四种恢复算法：

- 影子拷贝方法
- 基于undo日志的恢复方法
- 基于redo日志的恢复方法
- 基于undo/redo日志的恢复方法

	FORCE强制 (事务提交强制刷盘)	NO-FORCE非强制 (事务提交非强制刷盘)
NO-STEAL非窃取 (执行期间不刷盘)	无redo日志 无undo日志	有redo日志 无undo日志
STEAL窃取 (执行期间可刷盘)	无redo日志 有undo日志	有redo日志 有undo日志

影子拷贝方法

- ✓ No-STEAL/FORCE性质的算法
- ✓ 事务修改在拷贝数据库上(或者影子拷贝页面上)
- ✓ 提交事务时，切换数据库指针
- ✓ 优化：影子页面，仅拷贝修改的页面
- ✓ 缺点：效率低，难以支持事务并发



基于undo日志的恢复

- STEAL/FORCE性质的算法
- 不需要处理事务重做，需要依靠undo日志来回滚事务
- 故障恢复时，需要完成的任务：
 - 找到所有未完成的事务
 - 回滚这些未完成的事务
 - 写入该事务中止的日志

基于undo日志的恢复

□ 正常事务T的执行流程

- 开始事务：向日志中写入事务开始记录 $\langle T \text{ start} \rangle$
- 修改数据项X：向日志中写入undo日志记录 $\langle T, X, v_{old} \rangle$ ， v_{old} 代表数据项修改前的值（修改过的脏页允许刷盘）
- 提交事务：将T关联的脏页写入磁盘，写入事务提交记录 $\langle T \text{ commit} \rangle$ ，并且将脏页相关undo日志刷盘
- Abort事务：将T关联的脏页及undo日志刷盘，写入记录 $\langle T \text{ abort} \rangle$

□ 页面淘汰时：将淘汰的页面和对应的undo日志刷盘

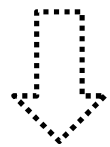
□ 回滚事务执行流程

- 反向扫描T相关的undo日志，执行回滚
- 将回滚中更新的脏页刷盘，写入事务中止记录 $\langle T \text{ abort} \rangle$

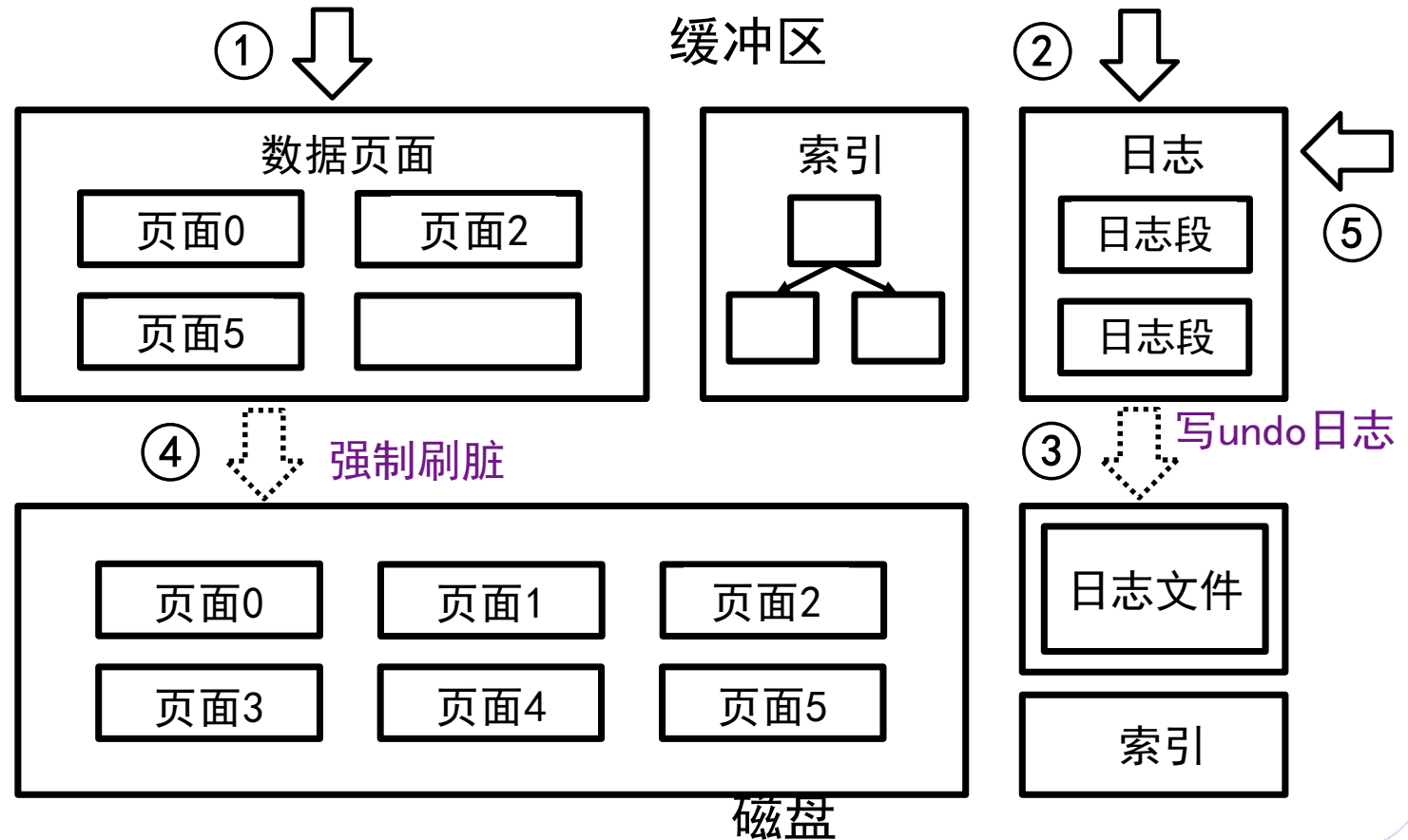
基于undo日志的恢复

1. 写入数据项
2. 写入日志记录
3. 将日志写回磁盘
4. 将脏页写回磁盘
5. 返回提交成功

淘汰页面时刷新页面和相关undo日志



需要保证事务提交前完成



基于undo日志的恢复流程

- 识别崩溃发生时刻数据库中事务的状态，恢复系统会扫描整个日志并且条件识别出需要回滚的事务（即未结束的事务）。
- 结束的事务（提交或者中止，即包含 $\langle T, \text{start} \rangle$ 且包含 $\langle T, \text{commit} \rangle$ 或 $\langle T, \text{abort} \rangle$ 的事务），不需要被回滚。
- 未结束的事务（即只包含 $\langle T, \text{start} \rangle$ 但不包含 $\langle T, \text{commit} \rangle$ 或 $\langle T, \text{abort} \rangle$ 的事务），需要被回滚/撤销（利用undo日志）。

基于undo日志的恢复流程

□ 回滚所有未结束的事务

□ 恢复系统逆序扫描整个undo日志，找出其中的数据更新日志记录，如果该日志记录属于未结束的事务，根据undo日志记录将数据项恢复为原值。

- 例如遇到属于该事务的更新日志记录 $\langle T, X, v_{old} \rangle$,
- T 属于未结束的事务，将数据项 X 的值置为 v_{old} 。

□ 将恢复阶段的数据更新刷回磁盘。对于每一个回滚事务 T ，写入一个事务中止记录 $\langle T, abort \rangle$

□ 至此，恢复过程结束。

基于undo日志的恢复

✓ 四个事务，五个数据项

T_1
A: 100 → 200

T_2
A: 200 → 300

T_3
A: 300 → 400
B: 400 → 200

T_4
X: 500 → 600
Y: 900 → 700
B: 200 → 300
C: 100 → 200

- ✓ T_1, T_2 在崩溃以前就已经提交，因此不需要回滚
- ✓ T_3, T_4 在崩溃之前没有结束，因此需要被处理
 - 日志008、010、011、012需要被回滚

001: < T_1 start >
002: < T_1 , A, 100 >
003: < T_1 commit >
004: < T_2 start >
005: < T_2 , A, 200 >
006: < T_3 start >
007: < T_2 commit >
008: < T_3 , A, 300 >
009: < T_4 start >
010: < T_4 , X, 500 >
011: < T_4 , Y, 900 >
012: < T_3 , B, 400 >

----- ↗
缓冲区刷盘，
写入A、X、Y的新值

基于undo日志的恢复

缓冲区状态	A: 100 B: 400 C: 100 X: 500 Y: 900
磁盘状态	A: 100 B: 400 C: 100 X: 500 Y: 900



T_1, T_2 提交

缓冲区状态	A: 300 B: 400 C: 100 X: 500 Y: 900
磁盘状态	A: 300 B: 400 C: 100 X: 500 Y: 900



数据库刷盘，写入未提交事务(T_3, T_4)修改记录

缓冲区状态	A: 400 B: 400 C: 100 X: 600 Y: 700
磁盘状态	A: 400 B: 400 C: 100 X: 600 Y: 700



数据库崩溃，进入不一致状态

缓冲区状态	无
磁盘状态	A: 400 B: 400 C: 100 X: 600 Y: 700



恢复完成，回到一致状态

缓冲区状态	A: 300 B: 400 C: 100 X: 500 Y: 900
磁盘状态	A: 300 B: 400 C: 100 X: 500 Y: 900

T_1
A: 100 → 200

T_2
A: 200 → 300

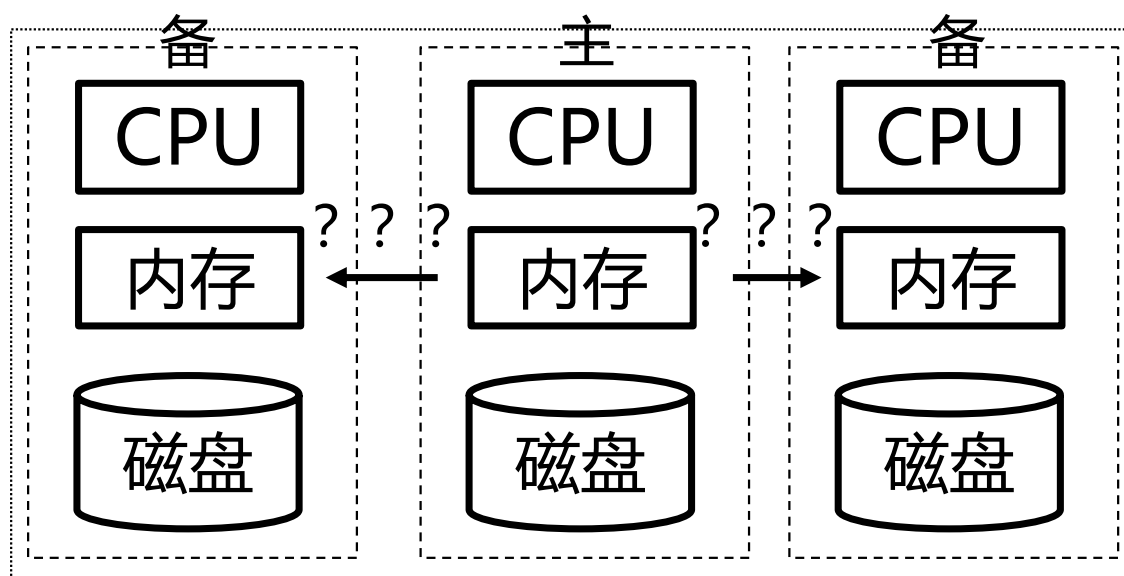
T_3
A: 300 → 400
B: 400 → 200

T_4
X: 500 → 600
Y: 900 → 700
B: 200 → 300
C: 100 → 200

- ✓ 磁盘数据和缓冲区的状态变化
- ✓ 提交之后缓冲区的修改立即在磁盘中生效
- ✓ 恢复阶段重做日志会复原缓存池中数据的状态，这些修改最终会被写入磁盘

基于undo日志的问题

- 每次事务提交都需要强制刷盘，造成随机页面读写多，性能差！
- 难以实现主备之间同步



一主多备

基于redo日志的恢复

- NO-STEAL/NO-FORCE性质的算法
- 不需要处理事务回滚，需要依靠redo日志处理事务重做
- 故障恢复时，需要：
 - 找到所有已提交的事务
 - 重做这些已完成的事务
 - 写入该事务结束的日志

基于redo日志的恢复

□ 正常事务T的执行流程

- 开始事务：向日志中写入事务开始记录 $\langle T \text{ start} \rangle$
- 修改数据项X：向日志中写入redo日志记录 $\langle T, X, v_{new} \rangle$ ， v_{new} 代表数据项修改后的值；未提交事务修改过的脏页不允许刷盘。
- 提交事务：写入事务提交记录 $\langle T \text{ commit} \rangle$ ，并且将日志刷盘。事务T关联的脏页（且该脏页无相关未提交事务）允许刷盘。
- Abort事务：invalidate（失效）缓冲区内该事务修改的页面

□ 淘汰页面：只允许淘汰无相关未提交事务（活跃事务）的页面

□ 回滚事务T执行流程

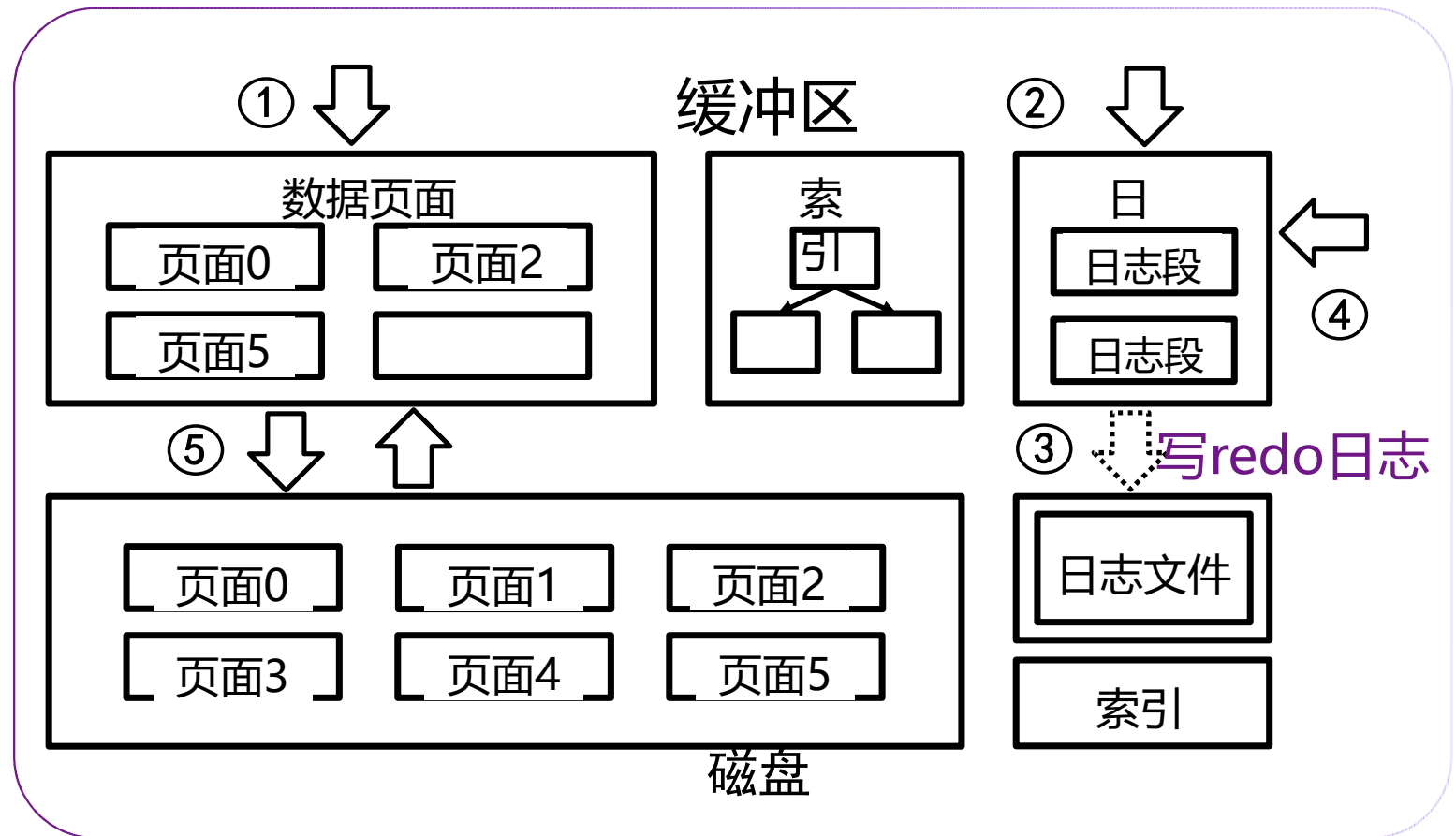
- 废弃T相关的脏页
- 写入事务中止记录 $\langle T \text{ abort} \rangle$

基于redo日志的恢复

1. 写入数据项
2. 写入日志记录
3. 将日志写回磁盘
4. 返回提交成功
5. 异步将脏页写回磁盘

注意：事务未提交前不能写磁盘；事务提交后只能写不含活跃事务的页面

 需要保证事务提交前完成



基于redo日志的恢复流程

□ 恢复子系统会从数据库日志末尾向前扫描日志，对于扫描过程中出现的事务 T ：

- (1) 如果出现 $\langle T \text{ commit} \rangle$ 的日志记录，说明该事务已经被提交了，系统需要对其进行重做。
 - 如果已Commit的事务已经刷盘，可以不用重做。
 - 后续通过checkpoint检查点机制来判断是否刷盘，从而可以实现刷盘的事务不需要重做。
- (2) 如果出现 $\langle T \text{ abort} \rangle$ 或者没有找到事务提交记录，那么系统不需要处理任何该事务关联的日志记录。

基于redo日志的恢复流程

□ 扫描过程结束以后，恢复子系统会从日志头部开始向后扫描日志。对于遇到的每一条形如 $\langle T, X, v_{new} \rangle$ 的更新日志记录，系统根据事务的性质进行不同的处理：

- (1) 如果 T 是未提交的事务，直接跳过。
- (2) 如果 T 是提交的事务，则将数据项 X 置为 v_{new} 。

□ 扫描结束后，对每个未完成的事务 T ，在日志中写入一个 $\langle T \text{ abort} \rangle$ 记录并刷新日志。

□ 至此，恢复算法结束。

基于redo日志的恢复

001: $\langle T_1, \text{start} \rangle$
002: $\langle T_1, A, 200 \rangle$
003: $\langle T_1, \text{commit} \rangle$
004: $\langle T_2, \text{start} \rangle$
005: $\langle T_2, A, 300 \rangle$
006: $\langle T_3, \text{start} \rangle$
007: $\langle T_2, \text{commit} \rangle$
008: $\langle T_3, A, 400 \rangle$
009: $\langle T_4, \text{start} \rangle$
010: $\langle T_4, X, 600 \rangle$
011: $\langle T_4, Y, 700 \rangle$
012: $\langle T_3, B, 200 \rangle$

✓ 四个事务，五个数据项

T_1	T_2
A: 100 \rightarrow 200	A: 200 \rightarrow 300

T_3	T_4
A: 300 \rightarrow 400	X: 500 \rightarrow 600
B: 400 \rightarrow 200	Y: 900 \rightarrow 700
	B: 200 \rightarrow 300
	C: 100 \rightarrow 200

✓ T_1, T_2 在崩溃以前就已经提交，因此需要被重做

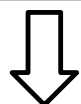
- 日志002、005需要被重做

✓ T_3, T_4 在崩溃之前没有结束，因此不需要处理

注：缓冲区未向磁盘写入任何页面

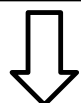
基于redo日志的恢复

缓冲区状态	A: 100 B: 400 C: 100 X: 500 Y: 900
磁盘状态	A: 100 B: 400 C: 100 X: 500 Y: 900



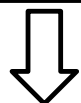
T_1, T_2 提交

缓冲区状态	A: 300 B: 400 C: 100 X: 500 Y: 900
磁盘状态	A: 100 B: 400 C: 100 X: 500 Y: 900



数据库崩溃, 进入不一致状态

缓冲区状态	无
磁盘状态	A: 100 B: 400 C: 100 X: 500 Y: 900



恢复完成, 恢复到一致状态

缓冲区状态	A: 300 B: 400 C: 100 X: 500 Y: 900
磁盘状态	A: 300 B: 400 C: 100 X: 500 Y: 900

001: $\langle T_1, \text{start} \rangle$
002: $\langle T_1, A, 200 \rangle$
003: $\langle T_1 \text{ commit} \rangle$
004: $\langle T_2 \text{ start} \rangle$
005: $\langle T_2, A, 300 \rangle$
006: $\langle T_3 \text{ start} \rangle$
007: $\langle T_2 \text{ commit} \rangle$
008: $\langle T_3, A, 400 \rangle$
009: $\langle T_4 \text{ start} \rangle$
010: $\langle T_4, X, 600 \rangle$
011: $\langle T_4, Y, 700 \rangle$
012: $\langle T_3, B, 200 \rangle$

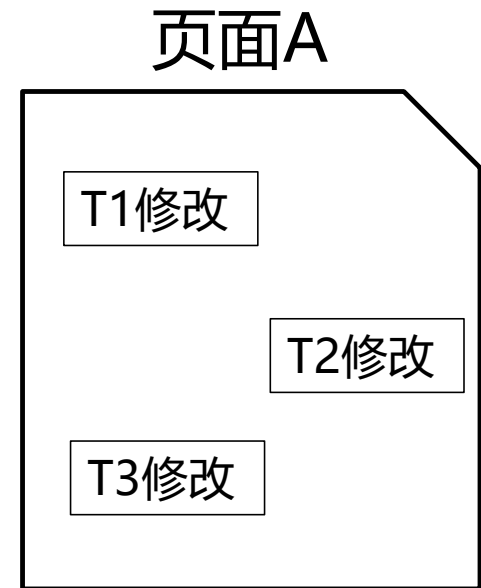
- 磁盘数据和缓冲区的状态变化
- 提交之后缓冲区的修改不会立即写到磁盘
- 恢复阶段重做日志会复原缓存池中数据的状态, 这些修改最终会被写入磁盘

基于redo日志的问题

□ 事务执行期间不能刷盘，造成内存空间占用大；
buffer缓冲池满时，由于不能淘汰未完成的事务，需要等待

□ 事务并发受限

- 考虑一个并发场景
 - 一个提交的事务T1更改了页面A → 可以将A刷盘
 - 一个未提交的事务T2也更改了页面A → 不能将A刷盘
- 如何处理页面A？是否允许刷盘？



基于undo/redo日志的恢复

□ STEAL/NO-FORCE性质的算法

- 需要依靠undo日志处理事务回滚
- 需要依靠redo日志处理事务重做

□ 故障恢复时，需要：

- 找到所有需要重做以及需要回滚的事务（分析阶段）
- 重做这些已完成的事务（重做阶段）
- 回滚未结束的事务（撤销阶段）

基于undo/redo日志的恢复

□ 正常事务T的执行流程

- 开始事务：向日志中写入事务开始记录 $\langle T \text{ start} \rangle$
- 修改数据项X：向日志中写入日志记录 $\langle T, X, V_{old}, V_{new} \rangle$ ，修改过的脏页允许刷盘
- 提交事务：写入事务提交记录 $\langle T \text{ commit} \rangle$ ，并且将日志刷盘，页面可不刷盘
- Abort事务：写入事务中止记录 $\langle T \text{ abort} \rangle$ ，并且将日志刷盘，页面可不刷盘

□ 页面淘汰时：此页面关联事务的redo/undo日志必须已经刷盘

□ 回滚事务执行流程：

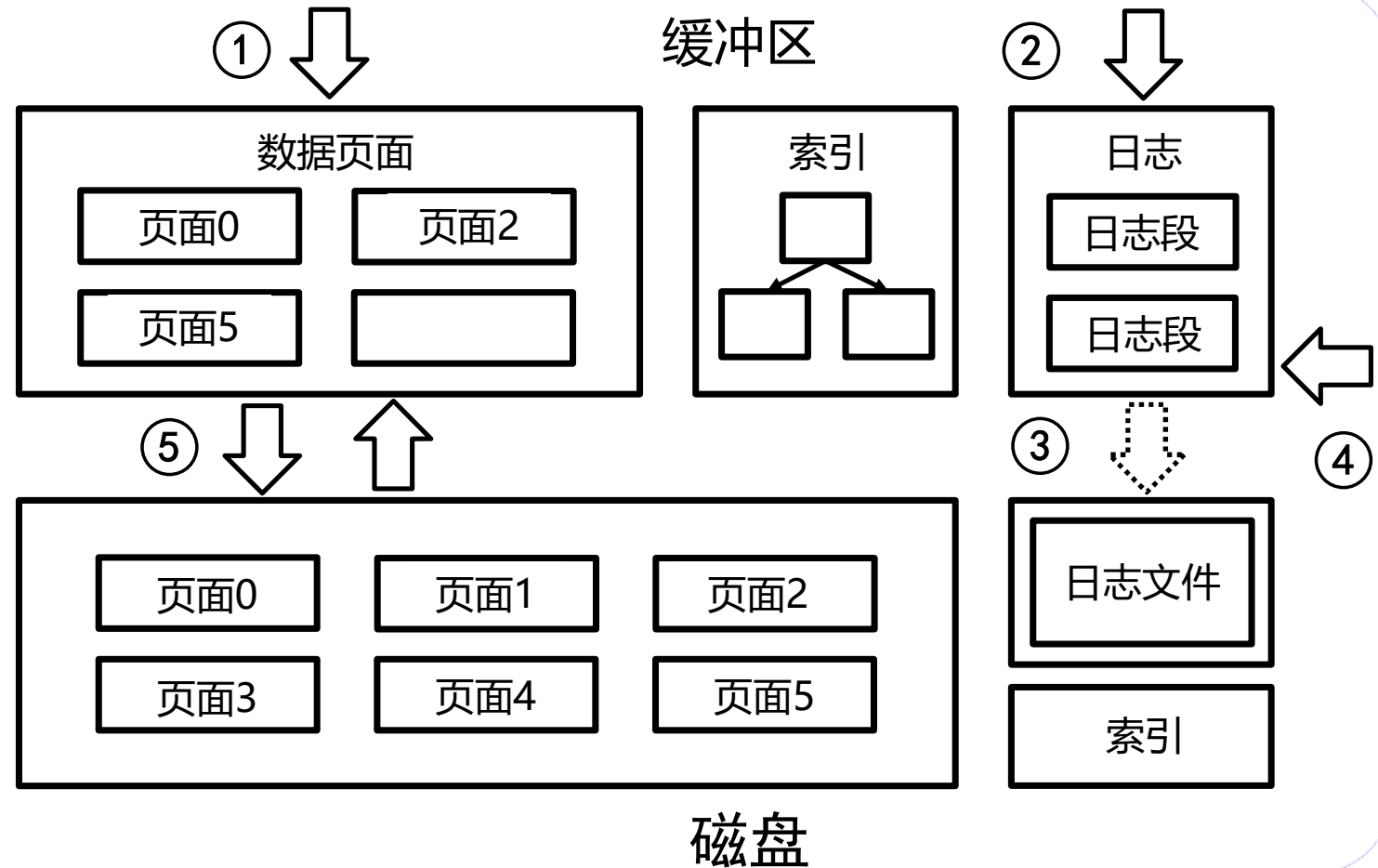
- 反向扫描T相关的undo日志，执行回滚
- 写入该事务中止的记录 $\langle T \text{ abort} \rangle$ ，并将日志刷盘

基于undo/redo日志的恢复

1. 写入数据项
2. 写入日志记录
3. 将日志写回磁盘
4. 返回提交成功
5. 将脏页异步写回磁盘

注意：事务中间允许写磁盘

 需要保证事务提交前完成



undo/redo恢复的分析阶段

□ 系统从日志起始位置开始扫描整个日志，找出需要重做和需要回滚的事务：

- 在扫描过程中出现 $\langle T \text{ start} \rangle$ 的日志记录而没有 $\langle T \text{ commit} \rangle$ 或 $\langle T \text{ abort} \rangle$ ，那么该事务在数据库崩溃的时刻是未结束的，需要被回滚（标注回滚）。
- 在扫描过程中出现了 $\langle T \text{ commit} \rangle$ 或 $\langle T \text{ abort} \rangle$ ，那么事务是已经完成，需要被恢复子系统重做（标注重做）。

undo/redo恢复的重做阶段

- 系统按时间顺序**正向扫描日志**，如果出现了一条**标注重做**的日志记录，系统便重做它。
- 由于系统重做了所有的日志更新记录，这个过程和数据库的执行历史是相同的，因此该过程也被称作**重放历史**（repeating history）。

undo/redo恢复的撤销阶段

- 从**日志末尾反向扫描**整个日志，如果出现了一条标注撤销的日志记录，那么系统会撤销它（包括修改缓冲区）。
- 一旦事务撤销完成（即扫描中遇到了 $\langle T \text{ start} \rangle$ ），数据库会自动写入 $\langle T \text{ abort} \rangle$ ，代表该事务已经回滚完成。
- 问题：如果在恢复过程中，如果系统故障了，该如何处理？
- 由于undo日志是逻辑日志，不能多次执行一条undo日志，撤销过程中需要记录某条undo日志（补偿日志）是否被执行过。

基于undo/redo日志的恢复

□ 补偿日志：Undo日志的redo日志

- 每次执行undo日志记录后，数据库需要向日志中写入一条**补偿日志记录**（compensation log record, CLR），记录撤销的动作
- **CLR实现了undo日志的redo**，记录已经undo的日志，保证undo不被重复执行

基于undo/redo日志的恢复

□ T_1, T_2 在崩溃以前就已经提交, 因此需要被重做

- 日志002、003、005、007、010需要被重做

□ T_3 在崩溃之前没有结束, 需要撤销/回滚

- 日志009、012需要被撤销

001: $\langle T_1, \text{start} \rangle$
002: $\langle T_1, A, 100, 300 \rangle$
003: $\langle T_1, B, 200, 250 \rangle$
004: $\langle T_3, \text{start} \rangle$
005: $\langle T_1, C, 400, 300 \rangle$
006: $\langle T_2, \text{start} \rangle$
007: $\langle T_2, D, 300, 350 \rangle$
008: $\langle T_1, \text{commit} \rangle$
009: $\langle T_3, B, 250, 450 \rangle$
010: $\langle T_2, A, 300, 500 \rangle$
011: $\langle T_2, \text{commit} \rangle$
012: $\langle T_3, E, 400, 800 \rangle$

基于undo/redo日志的恢复

001: $\langle T_1, \text{start} \rangle$
002: $\langle T_1, A, 100, 300 \rangle$
003: $\langle T_1, B, 200, 250 \rangle$
004: $\langle T_3, \text{start} \rangle$
005: $\langle T_1, C, 400, 300 \rangle$
006: $\langle T_2, \text{start} \rangle$
007: $\langle T_2, D, 300, 350 \rangle$
008: $\langle T_1, \text{commit} \rangle$
009: $\langle T_3, B, 250, 450 \rangle$
010: $\langle T_2, A, 300, 500 \rangle$
011: $\langle T_2, \text{commit} \rangle$
012: $\langle T_3, E, 400, 800 \rangle$

013: $\langle T_3, E, \text{CLR}, 400 \rangle$
014: $\langle T_3, B, \text{CLR}, 250 \rangle$
015: $\langle T_3, \text{abort} \rangle$

系统崩溃后开始恢复

- 撤销过程中数据修改不会立刻写回磁盘，需要写入CLR日志
- 一旦撤销完成后系统再次崩溃，可以使用CLR重做撤销操作

检查点机制

- 数据库的日志会随着事务的执行不断变长，这会使恢复过程中系统需要扫描并处理更多的日志，恢复时间也相应地变长，需要压缩日志大小来降低恢复的时间。
- 数据库设计了一种检查点(checkpoint)机制，检查点定义了一个脏页刷盘的时刻，要求**检查点之前的日志记录对应的缓冲区数据页面修改已经刷新到磁盘。**

检查点机制

- 在有检查点的情况下，系统恢复首先定位到检查点时候的日志，并分为一下三种情况处理日志。
 - （1）在检查点之前**完成（commit/abort）**的事务不需要处理；
 - （2）在检查点之后**commit/abort的事务需要重做；**
 - （3）所有未完成的事务（不含commit/abort）需要回滚。
- 数据库管理系统往往会定时地执行检查点操作，用户也可以在终端手动输入“checkpoint”令数据库系统执行检查点操作。
- 当数据库开始检查点操作后，它会持续将缓冲区的脏页写入磁盘。

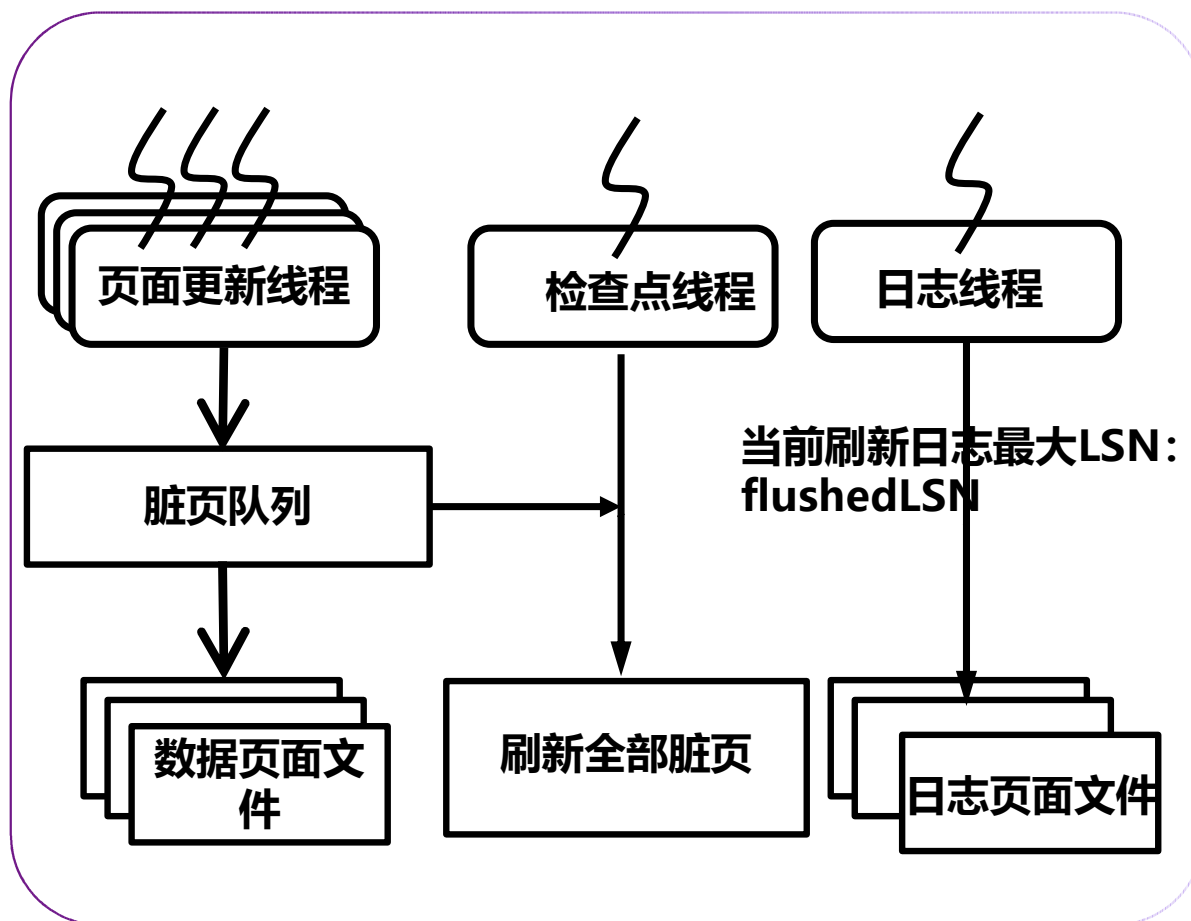
全量检查点

□刷新全量脏页

□记录checkpoint检查点

- 停止接受新的事务或修改请求，确保没有新的脏数据产生。
- 将当前所有未持久化的脏数据页写入磁盘，更新对应的数据文件。
- 记录检查点位置。
- 恢复接受新的事务或修改请求，继续正常的数据库操作。

□需要暂停业务（优化策略：非刷脏的页面可修改）



目录

1. 事务原子性和持久性的实现
2. 数据库故障恢复机制概述
3. 单机系统崩溃恢复方法
- 4. ARIES恢复算法**
5. 数据库备份技术
6. 数据库多机恢复

ARIES算法

- ARIES是一种state-of-the-art的恢复算法
- 最早由IBM公司数据库专家设计，应用于DB2数据库
- 基于undo/redo日志
- 应用于很多商业数据库系统

基于undo/redo日志的恢复算法缺陷

□ 部分redo日志无需重做

- 脏页表：已经刷脏的日志不需要重做

□ Undo 操作无需扫描全部日志

- 日志记录的前向指针

□ Undo日志恢复期间故障恢复问题

- Undo的redo

□ 故障恢复执行时间过长

- 模糊检查点和增量检查点

ARIES优化策略

□ ARIES针对上述四个问题，分别设计了解决方案

问题	优化策略
部分redo日志无需重做	引入脏页表，更新前通过比较检查，优化更新效率
Undo 操作无需扫描全部日志	引入活跃事务表和PrevLSN，跳过不相关的日志记录
Undo日志恢复期间故障恢复问题	引入UndoNextLSN字段，加速撤销过程执行
故障恢复执行时间过长	引入模糊检查点和增量检查点，在不牺牲性能的前提下优化了恢复速度

部分redo日志无需重做

刷新所有脏页

001: $\langle T_1, \text{start} \rangle$
002: $\langle T_1, A, 100, 300 \rangle$
003: $\langle T_1, B, 200, 250 \rangle$
004: $\langle T_3, \text{start} \rangle$
005: $\langle T_1, C, 400, 300 \rangle$
006: $\langle T_2, \text{start} \rangle$
007: $\langle T_2, D, 300, 350 \rangle$
008: $\langle T_1, \text{commit} \rangle$
009: $\langle T_3, B, 250, 450 \rangle$
010: $\langle T_2, A, 300, 500 \rangle$
011: $\langle T_2, \text{commit} \rangle$
012: $\langle T_3, E, 400, 800 \rangle$

□已经刷回磁盘的脏页不需要被重做

□假设日志记录005写入后，系统刷新了脏页，那么日志002、003、005都是不需要被重做

□重做日志记录开销较大（页面随机读写）

□挑战：如何记录脏页被刷新的时刻？

部分redo日志无需重做的解决方法

- redo日志冗余重做出现原因：在恢复过程中，数据库无法得知redo日志对应页面最后一次写回磁盘的时刻，即无法确认脏页是否被刷盘。
- 如果知道每一个脏页的刷盘时间，就可以避免不必要的重做。
- 需要通过引入新的结构来记录刷盘时间。
 - 脏页表：哪些页面更新了没刷盘，需要重做
 - 检查点：检查点之前已经刷盘不需重做

部分redo日志无需重做的解决方法

t_1

001: $\langle T_1, \text{start} \rangle$
002: $\langle T_1, A, 100, 300 \rangle$
>
003: $\langle T_1, B, 200, 250 \rangle$
004: $\langle T_3 \text{ start} \rangle$
005: $\langle T_1, C, 400, 300 \rangle$

t_2

>

脏页刷新

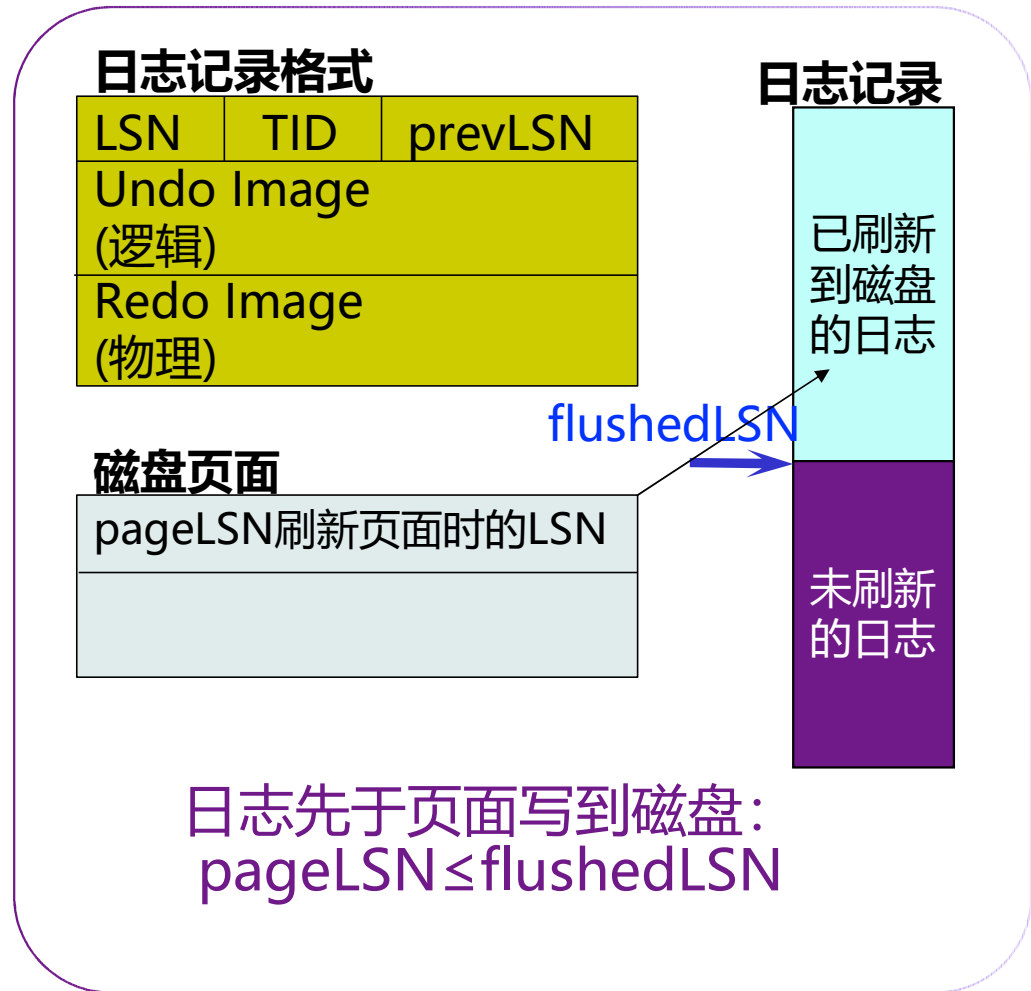
006: $\langle T_2 \text{ start} \rangle$
007: $\langle T_2, D, 300, 350 \rangle$
 t_3 008: $\langle T_1, \text{commit} \rangle$
009: $\langle T_3, B, 250, 450 \rangle$
010: $\langle T_2, A, 300, 500 \rangle$
011: $\langle T_2, \text{commit} \rangle$
012: $\langle T_3, E, 400, 800 \rangle$

日志生成时刻: t_1
页面刷脏时刻: t_2
 $t_1 < t_2$
日志记录不需要重做

日志生成时刻: t_3
页面刷脏时刻: t_2
 $t_2 < t_3$
日志记录需要重做

LSN和PageLSN

- ❑ ARIES算法采用逻辑时间判断事件发生的先后关系，为了构建逻辑时间，引入了LSN、PageLSN等一系列字段。
- ❑ LSN全称Log Sequence Number，代表日志序列号，表示日志记录出现的逻辑时间，LSN大小代表日志记录的顺序关系。
- ❑ PrevLSN，该事务的前一项LSN
- ❑ PageLSN是数据页面上新增字段，代表数据页面最近更新的时刻。
- ❑ 异步刷新日志记录FlushedLSN，刷新到磁盘的最大日志LSN



脏页表 (Dirty Page Table)

- 一种内存数据结构
- 跟踪数据库缓冲区里被未提交事务修改的页面
- 每个数据项包含两个字段
 - pageID: 页面的标识符
 - RecLSN: 数据页面最早在缓冲区被修改的时刻
- 数据项更新时机
 - 页面第一次被事务修改: 添加到脏页表
 - 页面非第一次被事务修改: 无需处理
 - 页面被写回磁盘: 从脏页表中被删除

页面号	RecLSN
B	5
C	4

脏页表的作用

- 判断**日志记录是否需要被重做，以及从哪开始重做**
- 假设已经拥有了崩溃时刻的脏页表
 - 日志记录的pageID不在脏页表中，不需要重做
 - 日志记录LSN \leq RecLSN,数据修改已在磁盘中生效，不需要重做
- 崩溃时刻的脏页表真的存在吗？
- 如何构造模拟脏页表？

页面号	RecLSN
B	5
C	4

脏页表的作用

- 判断日志记录是否需要被重做
- 假设已经拥有了崩溃时刻的脏页表
 - 日志记录的pageID不在脏页表中，不需要重做
 - 日志记录LSN \leq RecLSN,数据修改已在磁盘中生效，不需要重做
- 崩溃时刻的脏页表真的存在吗？ **不存在，因为脏页表在内存**
- 如何构造模拟脏页表？ **扫描日志重建**

脏页表

2: Write(B)	3: Write(A)	4: Write(C)	5: Write(B)	6: Write(C)	7: Write(D)
-------------	-------------	-------------	-------------	-------------	-------------



刷脏



检查点



崩溃

检查点中的
脏页表

页面号	RecLSN
B	5
C	4

崩溃时刻
前的脏页
表

页面号	RecLSN
D	7
B	5
C	4

磁盘

页面	PageLSN
A	3
B	2

扫描日志
重构的脏
页表

页面号	RecLSN
D	7
B	5
C	4

□相比崩溃时的脏页表，
重构的脏页表包含更
多的项

□这是因为恢复过程中
无法知道具体的刷脏
时刻

Undo 操作无需扫描全部日志

001: $\langle T_1, \text{start} \rangle$
002: $\langle T_1, A, 100, 300 \rangle$
003: $\langle T_1, B, 200, 250 \rangle$
004: $\langle T_3, \text{start} \rangle$ ✓
005: $\langle T_1, C, 400, 300 \rangle$
006: $\langle T_2, \text{start} \rangle$
007: $\langle T_2, D, 300, 350 \rangle$
008: $\langle T_1, \text{commit} \rangle$
009: $\langle T_3, B, 250, 450 \rangle$ ✓
010: $\langle T_2, A, 300, 500 \rangle$
011: $\langle T_2, \text{commit} \rangle$
012: $\langle T_3, E, 400, 800 \rangle$ ✓

□ 恢复过程仅有事务3需要被撤销

□ 除去日志012、009、004，其他日志

记录在撤销过程中不需要被扫描

□ 挑战：如何在回滚阶段避免无效的扫描？

活跃事务表和PrevLSN

- 一种内存数据结构
- 跟踪数据库中所有**未结束的事务（活跃事务）**

- 每个数据项包含两个字段

- TID：数据库事务的标识符
- LastLSN：事务最后所关联日志记录的LSN

- 数据项更新时机

- 事务第一次修改数据：添加到事务表
- 事务结束时：从事务表中删除
- 事务执行：更新LastLSN，**决定从哪开始回滚**

- PrevLSN是日志记录中新增字段，表示所属事务前一条日志记录

活跃事务表

lastLSN	TID
13	3

活跃事务表和PrevLSN的作用

- 快速定位未完成的事务
- 快速找到未完成的事务最后一条日志记录

```
0001: < T2, start >  
0002: < T1, start >  
0003: < T1, A, 100,200 >  
... ...  
0102: < T1, commit >  
0103: < T2, X, 120,200 >  
0104: < T2, C, 300,301 >  
0105: < T9, start >  
... ...  
0522: < T2, B, 401,400 >  
0523: < T25, start >  
0524: < T2, Y, 100,100 >  
0525: < T20, abort >  
... ...  
1002: < T2, D, 200,300 >  
1003: < T30, commit >
```

- ✓ 需要回滚的事务: T2
- ✓ T2是一个长事务, 包含20条日志更新记录
- ✓ 总共约1000条日志记录
- ✓ 总共包含20条需要回滚的日志
- ✓ 通过PrevLSN, 获得50倍的性能提升

活跃事务表和PrevLSN

LSN	Type	Tid	prevLSN
1	SOT	1	
2	UP	1	1
3	UP	1	2
4	SOT	4	
5	SOT	3	
6	UP	1	3
7	SOT	2	
8	UP	4	4
9	UP	2	7
10	EOT	1	6
11	UP	3	5
12	UP	2	9
13	EOT	2	12
14	UP	3	11

Tid	lastLSN
3	14
4	8

- ✓ PrevLSN形成一个链表，不同事务之间的链表是没有交集的
- ✓ 事务表包含两项，在这里事务3和事务4还没有结束

Undo日志恢复期间故障恢复问题

001: $\langle T_1, \text{start} \rangle$
002: $\langle T_1, A, 100, 300 \rangle$
003: $\langle T_1, B, 200, 250 \rangle$
004: $\langle T_3, \text{start} \rangle$
005: $\langle T_1, C, 400, 300 \rangle$
006: $\langle T_2, \text{start} \rangle$
007: $\langle T_2, D, 300, 350 \rangle$
008: $\langle T_1, \text{commit} \rangle$
009: $\langle T_3, B, 250, 450 \rangle$
010: $\langle T_2, A, 300, 500 \rangle$
011: $\langle T_2, \text{commit} \rangle$
012: $\langle T_3, E, 400, 800 \rangle$

- ✓ 为了解决撤销阶段出现新故障的问题，可以添加undo日志的redo日志（即补偿日志）
- ✓ 记录哪些undo日志已经完成，故障恢复时可以根据补偿日志进行恢复（类似于断点续传）。
- ✓ 问题：此过程是否可以继续优化？

UndoNextLSN

□ **undoNextLSN**: 位于每条CLR中, 代表事务下一条需要回滚的日志记录的LSN

LSN	Type	Tid	prevLSN	Data
1	SOT	1		
2	UP	1	1	A
3	UP	1	2	B
4	UP	1	3	C
5	CLR	1	4	C, 3
6	CLR	1	5	B, 2
7	CLR	1	6	A, 1
8	EOT	1	7	

第一次崩溃

第二次崩溃

undoNextLSN

- ✓ 再次恢复时快速定位需要撤销的日志记录
- ✓ LSN 4写入磁盘后系统崩溃, 开始恢复
- ✓ 恢复过程写入LSN 5、LSN 6之后系统再次崩溃
- ✓ 再次恢复是根据LSN 6的UndoNextLSN找到LSN 2开始撤销, 写入LSN 7、LSN 8
- ✓ LSN 3、LSN 4的撤销被跳过了

ARIES算法的WAL

- ❑ ARIES算法使用了undo/redo日志，因此也需要满足预写日志的条件
- ❑ ARIES中含有FlushedLSN字段，表示写入磁盘日志最大的LSN
- ❑ 脏页写回磁盘时，需满足PageLSN \leq FlushedLSN
- ❑ 事务提交时需要满足LastLSN \leq FlushedLSN
- ❑ ARIES中日志记录会批量定时从内存写回磁盘，每次写入完成时会更新FlushedLSN

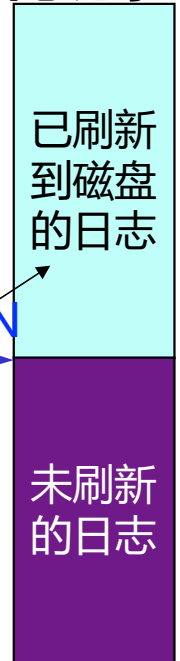
日志记录格式

LSN	TID	prevLSN
Undo Image (逻辑)		
Redo Image (物理)		

磁盘页面

pageLSN刷新页面时的LSN

日志记录



$pageLSN \leq flushedLSN$
 $lastLSN \leq flushedLSN$

故障恢复执行时间过长

- 普通的检查点机制尽管可以有效地缩减日志长度，从而减少恢复所需的时间。
- 然而，在设置检查点时，数据库需要把检查点之前的所有脏页写回磁盘，这会为数据库正常业务的运行带来巨大的开销。
- 挑战：如何减小检查点过程中的I/O开销？

数据库检查点

- 为了减少日志，数据库系统会定时执行检查点操作
- 将所有磁盘上的脏页刷回磁盘
- 全量检查点执行步骤
 - 等待
 - 缓冲区刷脏
 - 截断日志 – 已经刷盘，不需要的日志 – 可以回收
- 缺点：全量检查点过程中需要写回全部脏页面，占用大量IO资源

模糊检查点

- 一种对普通检查点的优化,解决需要全部刷脏的问题
- 整个检查点过程分成
 - 开始检查点<begin-checkpoint>
 - 结束检查点<end-checkpoint>
 - 中间过程写入脏页表和事务表
- 恢复时通过脏页表和事务表还原数据库状态,脏页表中存储了未被刷盘的脏页。

增量检查点

□不需全量脏页刷盘，增量刷新脏页，优化检查点执行速度

□单独线程刷新脏页

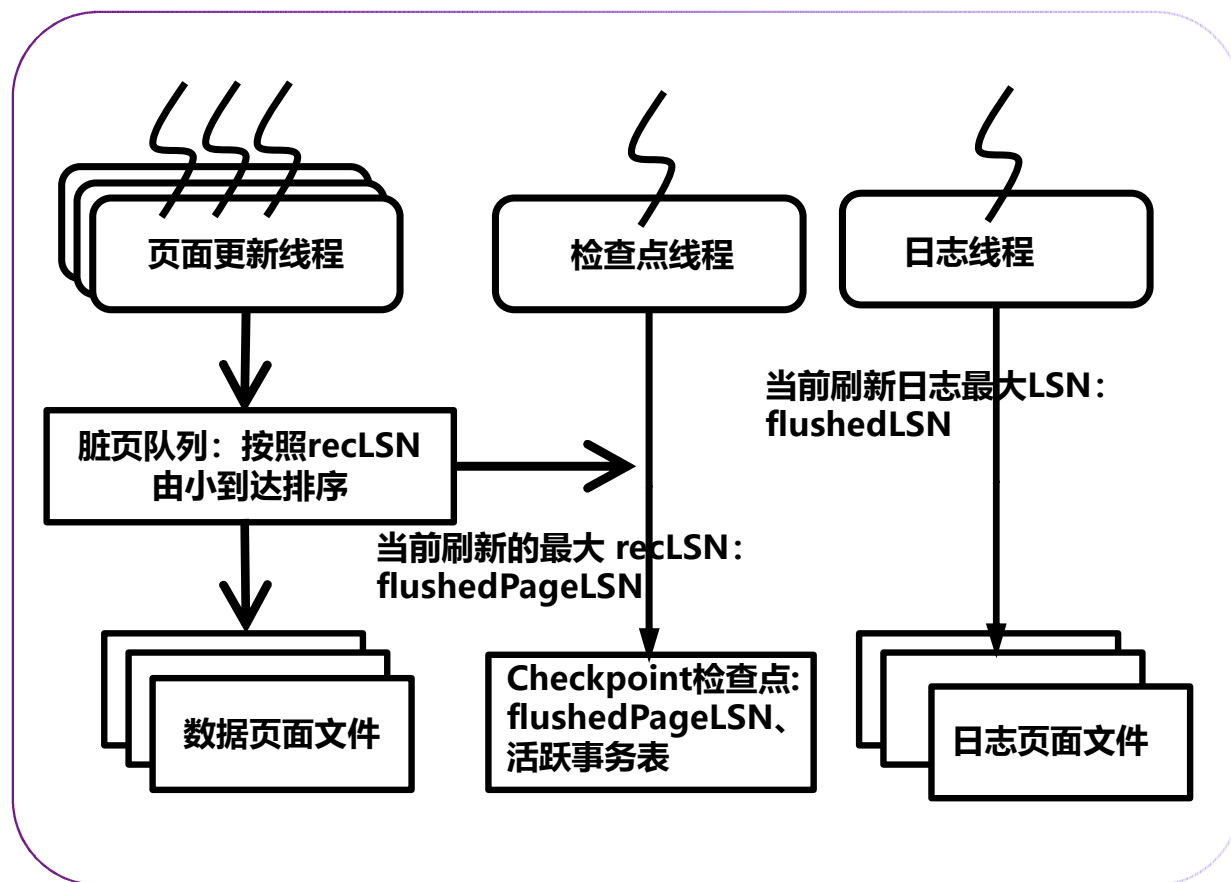
- 按照脏页的RecLSN排序
- 按照RecLSN由小到大刷新脏页

□在checkpoint检查点记录

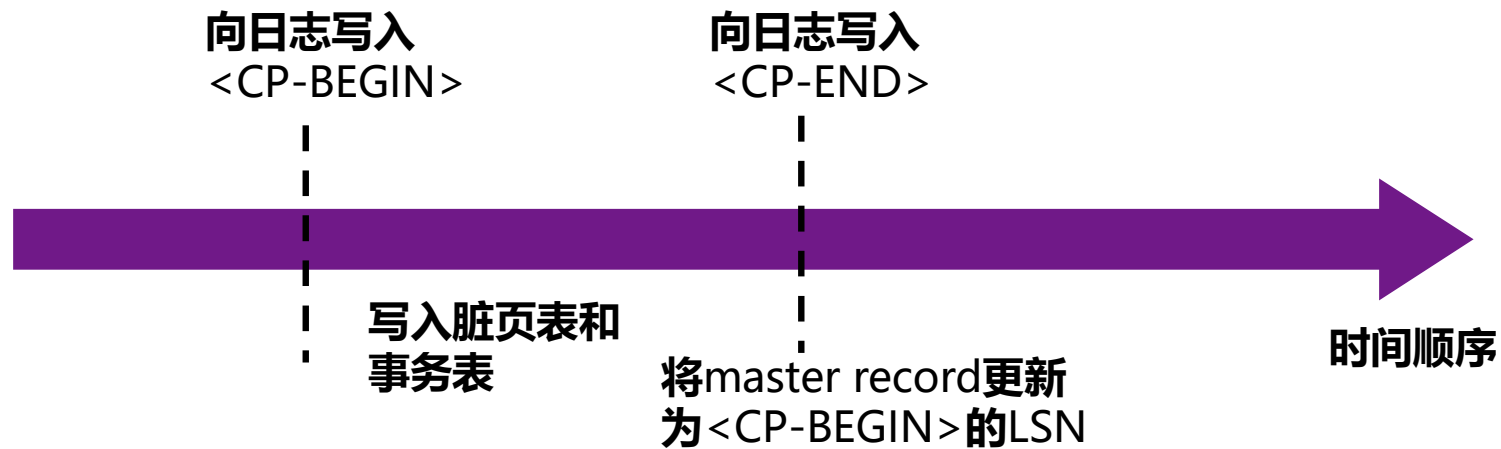
- 当前刷新脏页的最大LSN为FlushedPageLSN
- 活跃事务表ATT（事务提交日志）

□恢复时

- 从FlushedPageLSN日志开始重建脏页表和活跃事务表



ARIES算法检查点



- ✓ 由写入<CP-BEGIN>开始，由写入<CP-END>结束
- ✓ 中间过程将内存中的脏页表和事务表写入磁盘
- ✓ Master record: 一个特殊字段，位于磁盘，表示最新一次检查点的位置

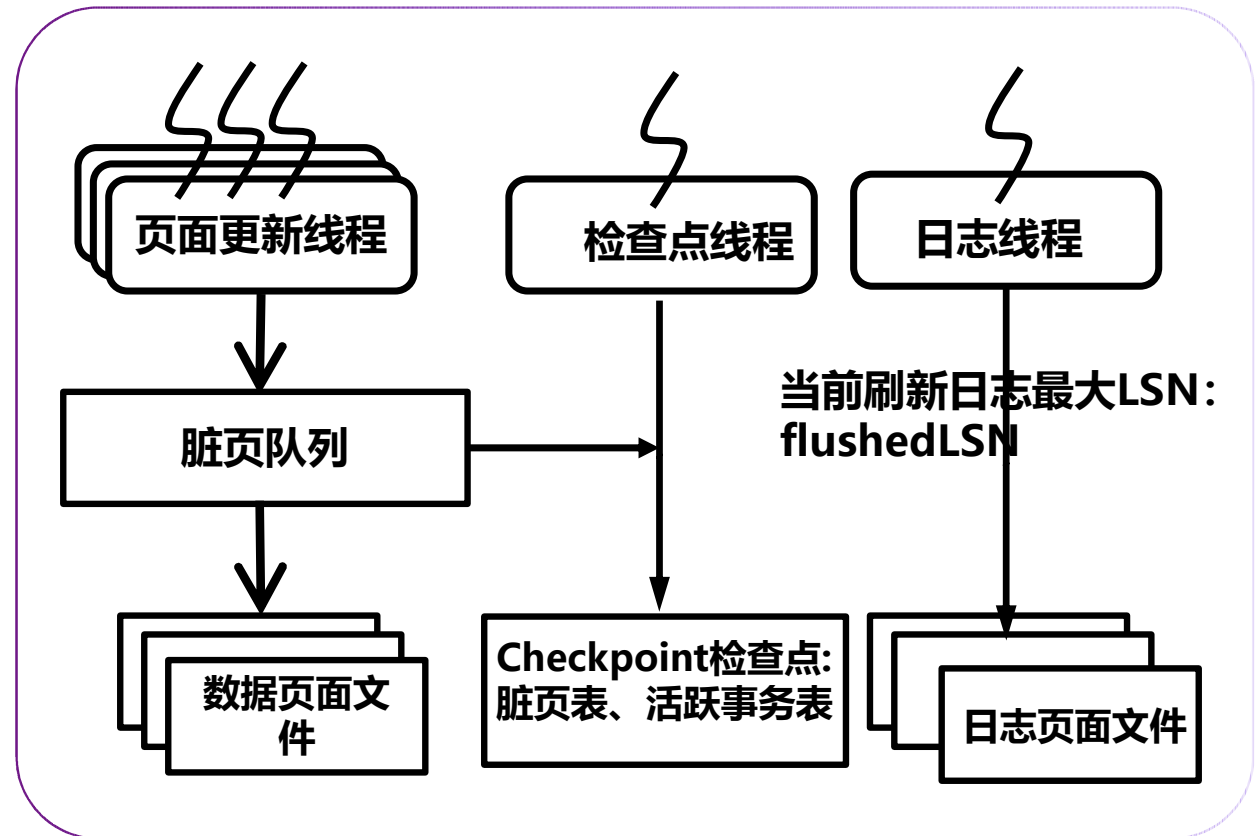
模糊检查点

□ 不需要全量脏页刷盘，单独线程刷新，优化检查点执行速度

- 脏页表
- 活跃事务表

□ 在checkpoint检查点记录

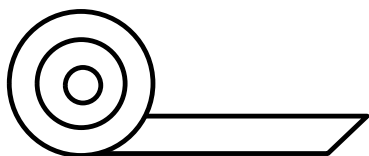
- 脏页表DPT
- 活跃事务表ATT



ARIES算法优化总结

问题	优化对策
重做阶段不必要的数据页面更新	引入脏页表，更新前通过比较检查，优化更新效率
撤销阶段不必要的日志扫描	引入活跃事务表和prevLSN，跳过不相关的日志记录
撤销阶段的故障恢复	引入补偿日志和undoNextLSN字段，确保撤销只执行一次
日志太多导致恢复时间过长	引入模糊检查点和增量检查点，优化了恢复速度

ARIES算法架构



日志记录

LSN(日志序号)

prevLSN(该事务前一条LSN)

undoNextLSN(补偿日志独有)

TID (事务ID)

Type (事务类型)

pageID (页面ID)

length (该日志记录长度)

offset (ObjectID)

before-image (修改前的值)

after-image (修改后的值)



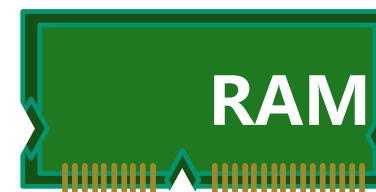
数据库

数据页面

每一个页面都带有
PageLSN (最后刷新
页面时刻的LSN)

超级记录 Master Record

(checkpoint pointer指
向检查点)



RAM

事务表

TID 事务ID

LastLSN 本事务最新一条LSN

脏页表

pageID

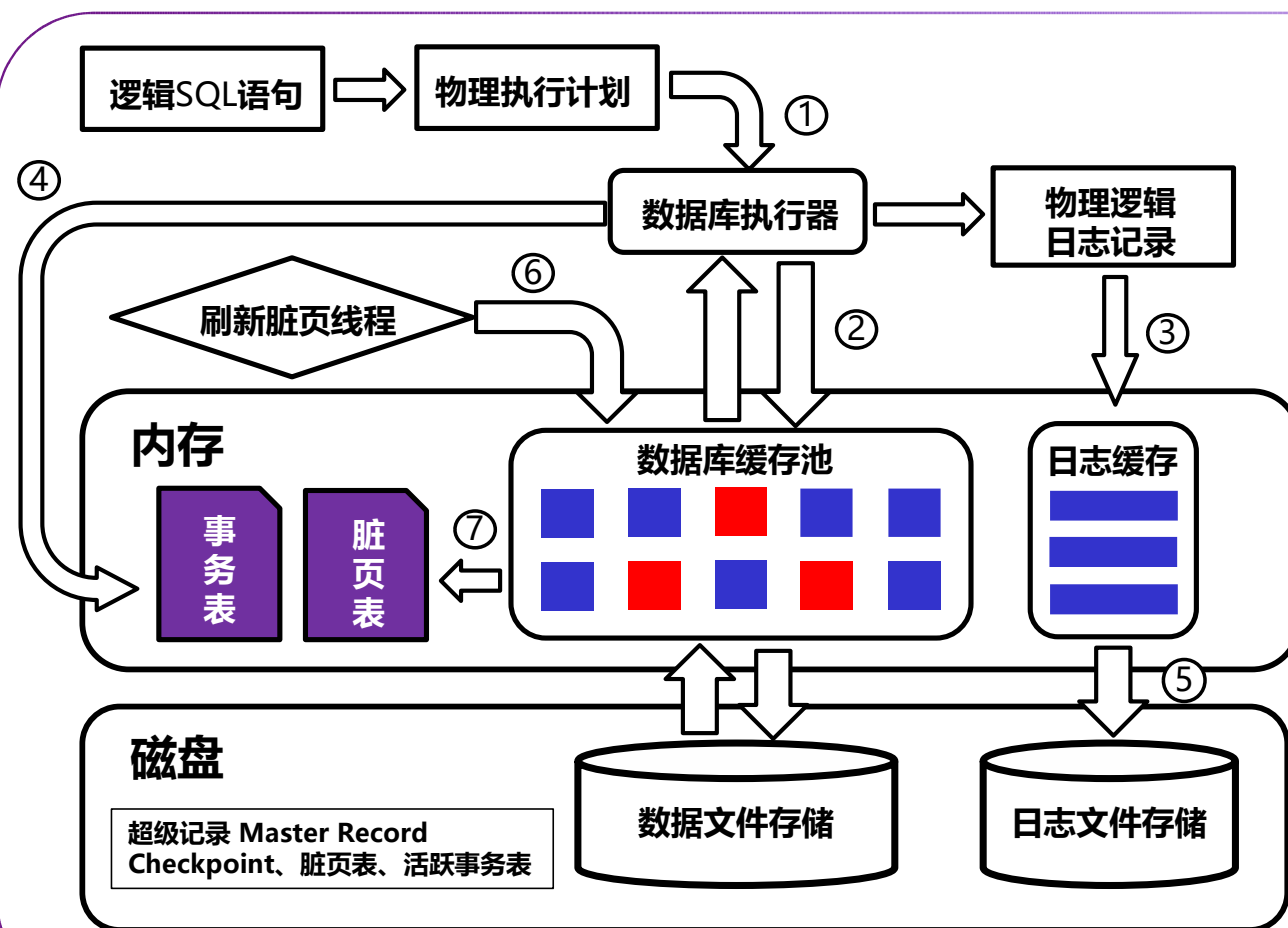
RecLSN 第一次造成脏页的LSN

日志末尾

FlushedLSN页面最新刷新的LSN

缓存池

ARIES正常流程



同步

1. SQL语句经过一系列变换转化成物理执行计划
2. 在缓冲区上读写数据页面
3. 日志记录写入缓存
4. 更新事务表和脏页表
5. 事务LSN刷新到磁盘

$\text{lastLSN} \leq \text{flushedLSN}$

异步

6. 刷脏线程定时写回脏页
7. 刷脏时更新脏页表

ARIES正常流程

□ 实时记录日志

□ 实时更新活跃事务表和脏页表

– 活跃事务表

- 事务开始时加入；事务结束时删除；
- 根据LSN更新LastLSN (**便于从最后这一条回滚**)

– 脏页表

- Page第一次修改时加入，并记录recLSN (**便于从这一条重做**)
- Page刷盘时删除（内存、磁盘一致时删除）

□ 异步刷新页面并更新FlushedLSN，并记录pageLSN

□ 异步增量检查点checkpoint

- Checkpoint记录当前脏页表和活跃事务表信息

活跃事务表

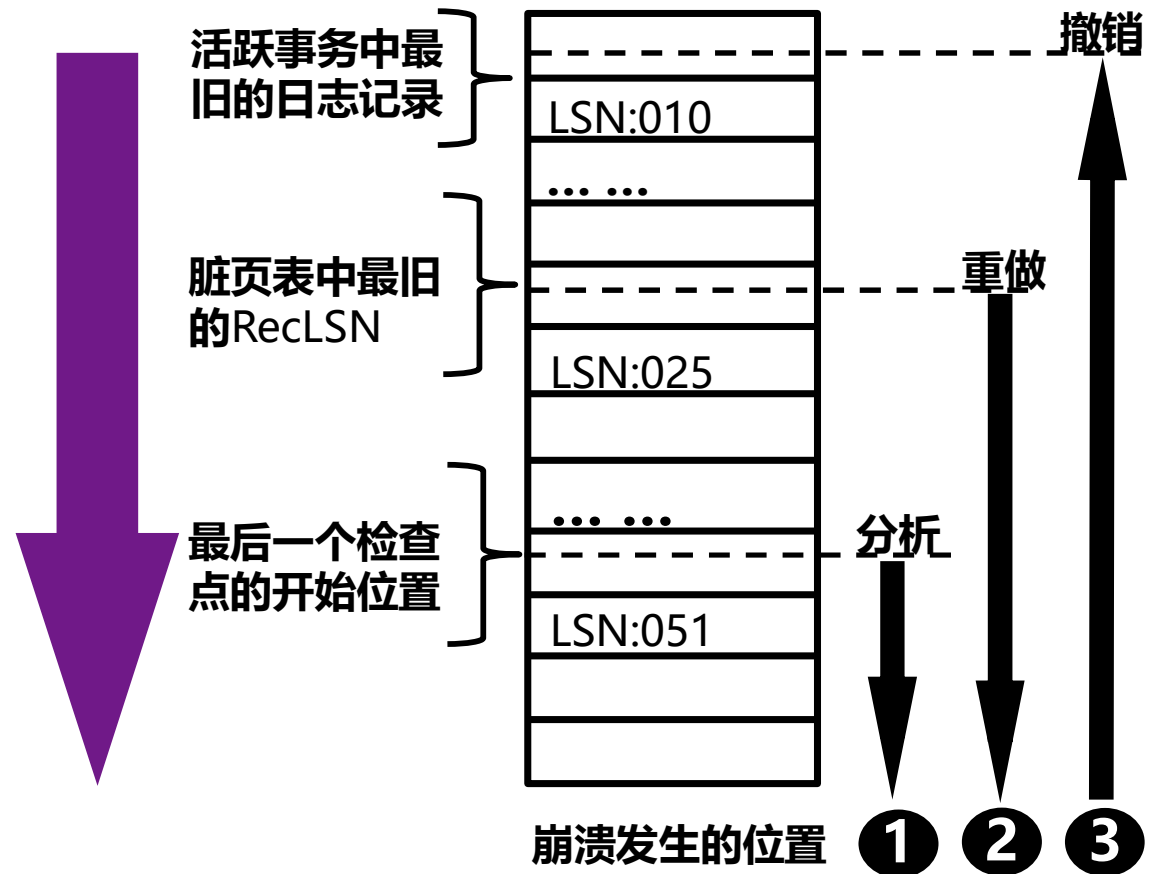
lastLSN	TID
13	3

脏页表

PageID	recLSN
A	2
B	3
C	6
D	8
E	13

ARIES恢复算法流程

- ❑ 分析阶段：利用checkpoint读取当时脏页表和事务表，扫描日志，重构脏页表和事务表
- ❑ 重做阶段：利用恢复的脏页表重放历史，将数据库恢复到崩溃前的状态
- ❑ 撤销阶段：利用活跃事务表回滚崩溃时刻未完成的事务
- ❑ 日志截断 (truncated) :撤销点以前的日志可以删除。
 - $\min(\min\text{LSN}(\text{脏页表}), \min\text{LSN}(\text{活跃事务}))$



分析阶段：流程

□ 主要任务

- 构造模拟脏页表和活跃事务表
- 利用脏页表确定重做开始的位置（重做）
- 利用活跃事务表找出所有未完成的事务（撤销）

□ 日志扫描模式：

- 从checkpoint读取当时的脏页表和活跃事务表
- 自checkpoint开始从前向后扫描日志记录，来构造崩溃时的脏页表和活跃事务表

分析阶段：流程

□ 从checkpoint开始扫描每一条日志log (LSN, TID, PageID, Op)

① 更新活跃事务列表

- 如果TID 在活跃事务列表
 - 如果OP是Commit, 把TID从活跃事务表删除
 - 否则更新活跃事务列表(TID, LastLSN) = (TID, LastLSN = LSN);
- 如果TID不在活跃事务列表
 - 将该事务加入活跃事务列表(TID, LastLSN = LSN)

② 更新脏页表

- 如果PageID不在脏页表将(Page, recLSN=LSN)加入脏页表
- 如果PageID在脏页表, 跳过不处理 (只需记录最早是该页面为脏页的LSN)
- PageID刷盘时, 从脏页表将PageID删除

分析阶段：根据脏页表确定重做开始位置

- 选择脏页表中最小的RecLSN作为重做阶段开始位置
- 如果脏页表为空，重做阶段开始位置为开始检查点记录的位置
- 正确性证明：对于任意日志记录，如果它小于 $\min(\text{RecLSN})$ ，那么说明它对应的页面没有出现在脏页表中，已经被刷回磁盘了，因此不需要被重做

分析阶段：构建活跃事务表找出未完成的事务

□ 对于每一条日志记录

- 读取事务ID，插入/更新事务表

□ 检查点结束日志记录

- 读取检查点中事务表，更新缓冲区中事务表

□ 事务结束记录

- 将事务表从数据项中删除

□ 其他日志记录

- 不处理

重做阶段：流程

□ 主要任务

- 重放历史，重做所有必须重做的日志记录，将数据库恢复到崩溃前一刻

□ 日志扫描模式

- 自脏页表中min(RecLSN)开始从前向后扫描，判断是否需要重做
 - 如果日志记录的pageID不在脏页表中，不需要重做
 - 如果日志记录LSN \leq RecLSN,数据修改已在磁盘生效，不需要重做
 - 如果日志记录LSN $>$ RecLSN,从磁盘读到缓冲区，如果PageLSN $>$ LSN，不需要重做
 - 以上条件均不满足，重做日志记录

□ 如果重做，完成以下操作

- 根据日志更新记录修改缓冲区数据项
- 更新脏页表和活跃事务表

撤销阶段

□ 主要任务

- 撤销所有未完成的事务
- 写入日志结束记录

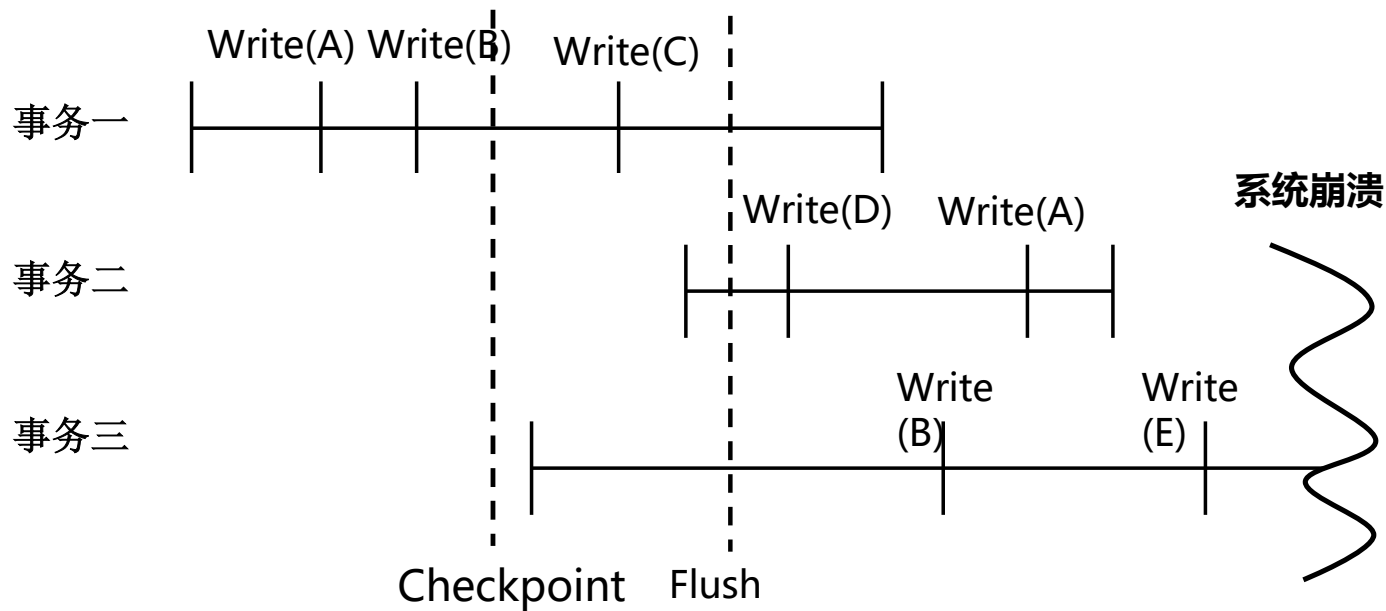
□ 日志扫描模式

- 借助LastLSN、PrevLSN以及UndoNxtLSN从后向前跳跃扫描
- 从活跃事务表根据事务ID找到LastLSN，根据LastLSN找到对应日志记录
- 对日志记录执行撤销操作，依次使用PrevLSN找到下一条要处理的日志
- 一旦PrevLSN = 0，代表到达事务的开始，回滚完成
- 写入该事务中止的日志记录

撤销阶段：撤销单条日志记录

- 根据日志记录内容修改数据项：通过记录中Undo内容修改，由于是逻辑日志，具体操作由数据库完成
- 更新脏页表：修改数据项后，将对应页面插入脏页表
- 写入补偿日志记录
- 撤销过程中，事务表每一个数据项含有一个新字段UndoNxtLSN，表示事务下一条需要撤销的日志
- 事务每撤销一条日志记录，事务表中的UndoNxtLSN更新为该条日志记录的PrevLSN

案例说明



- 三个事务、五个数据项
- 系统崩溃时事务一、事务二已经提交，事务三未结束

案例说明（完整日志记录）

LSN	Type	Tid	prevLSN	Data
1	SOT	1		
2	UP	1	1	A
3	UP	1	2	B
4	BEGIN-CP			
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

事务表

lastLSN	Tid
3	1

脏页表

页面号	recLSN
A	2
B	3

案例说明

事务表

LastLSN	TID
14	3

脏页表

页面号	RecLSN
D	9
B	11
A	12
E	14

检查点

事务表

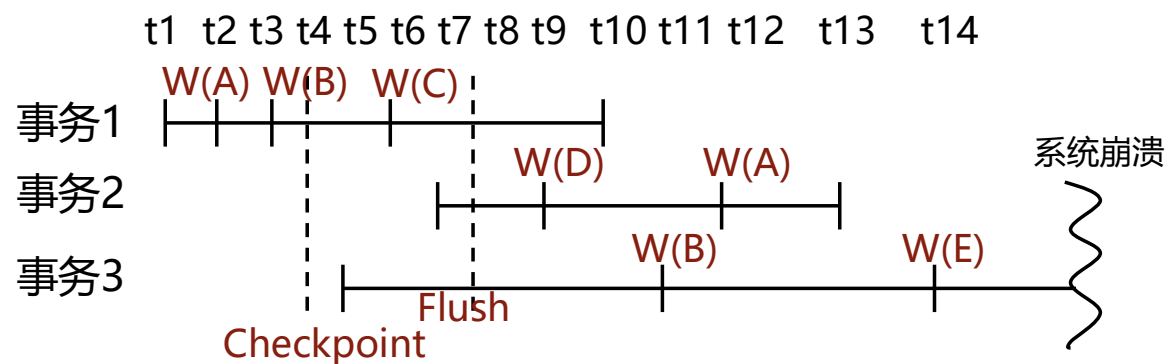
LastLSN	TID
3	1

脏页表

页面号	RecLSN
A	2
B	3

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?



案例说明（分析阶段）



LSN	Type	Tid	prevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

Flush刷脏

事务表

LastLSN	TID
3	1
5	3

脏页表

页面号	RecLSN
A	2
B	3

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

案例说明（分析阶段）



LSN	Type	Tid	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

Flush刷脏

事务表

LastLSN	TID
6	1
5	3

脏页表

页面号	RecLSN
A	2
B	3
C	6

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

案例说明（分析阶段）



LSN	Type	Tid	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

Flush刷脏

事务表

LastLSN	TID
6	1
5	3
7	2

脏页表

页面号	RecLSN
A	2
B	3
C	6

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

案例说明（分析阶段）



LSN	Type	Tid	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

Flush刷脏

事务表

LastLSN	TID
6	1
5	3
9	2

脏页表

页面号	RecLSN
D	9

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

案例说明（分析阶段）

LSN	Type	Tid	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

Flush刷脏



事务表

LastLSN	TID
14	3

脏页表

页面号	RecLSN
D	9
A	12
B	11
E	14

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

案例说明（重做阶段）



LSN	Type	TID	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A ✕
3	UP	1	2	B
4	BEGIN-CP			
5	SOT	3		
6	UP	1	3	C
7	SOT	2		Flush刷脏
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

脏页表

页面号	RecLSN
A	12
B	11
D	9
E	14

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

Redo UNLESS

- ① Page 不在脏页表中
- ② LSN < RecLSN: 页面在检查点之前已刷新
- ③ LSN ≤ PageLSN: 页面在检查点之后已刷新

案例说明（重做阶段）



LSN	Type	TID	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A ×
3	UP	1	2	B ×
4	BEGIN-CP			
5	SOT	3		
6	UP	1	3	C
7	SOT	2		Flush刷脏
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

脏页表

页面号	RecLSN
A	12
B	11
D	9
E	14

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

Redo UNLESS

- ① Page 不在脏页表中
- ② LSN < RecLSN: 页面在检查点之前已刷新
- ③ LSN ≤ PageLSN: 页面在检查点之后已刷新

案例说明（重做阶段）



LSN	Type	TID	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A ✕
3	UP	1	2	B ✕
4	BEGIN-CP			
5	SOT	3		
6	UP	1	3	C ✕
7	SOT	2		Flush刷脏
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

脏页表

页面号	RecLSN
A	12
B	11
D	9
E	14

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

Redo UNLESS

- ① Page 不在脏页表中
- ② LSN < RecLSN: 页面在检查点之前已刷新
- ③ LSN ≤ PageLSN: 页面在检查点之后已刷新

案例说明（重做阶段）



LSN	Type	TID	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A ✕
3	UP	1	2	B ✕
4	BEGIN-CP			
5	SOT	3		
6	UP	1	3	C ✕
7	SOT	2		Flush刷脏
8	END-CP			
9	UP	2	7	D ✓
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

脏页表

页面号	RecLSN
A	12
B	11
D	9
E	14

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

Redo UNLESS

- ① Page 不在脏页表中
- ② LSN < RecLSN: 页面在检查点之前已刷新
- ③ LSN ≤ PageLSN: 页面在检查点之后已刷新

案例说明（重做阶段）

脏页表

页面号	RecLSN
A	12
B	11
D	9
E	14

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

Redo UNLESS

- ① Page 不在脏页表中
- ② LSN < RecLSN: 页面在检查点之前已刷新
- ③ LSN ≤ PageLSN: 页面在检查点之后已刷新

LSN	Type	TID	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A ✕
3	UP	1	2	B ✕
4	BEGIN-CP			
5	SOT	3		
6	UP	1	3	C ✕
7	SOT	2		Flush刷脏
8	END-CP			
9	UP	2	7	D ✓
10	EOT	1	6	
11	UP	3	5	B ✓
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E

案例说明（重做阶段）

脏页表

页面号	RecLSN
A	12
B	11
D	9
E	14

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?


Redo UNLESS

- ① Page 不在脏页表中
- ② LSN < RecLSN: 页面在检查点之前已刷新
- ③ LSN ≤ PageLSN: 页面在检查点之后已刷新

LSN	Type	TID	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A ✕
3	UP	1	2	B ✕
4	BEGIN-CP			
5	SOT	3		
6	UP	1	3	C ✕
7	SOT	2		Flush刷脏
8	END-CP			
9	UP	2	7	D ✓
10	EOT	1	6	
11	UP	3	5	B ✓
12	UP	2	9	A ✓
13	EOT	2	12	
14	UP	3	11	E



案例说明（重做阶段）



LSN	Type	TID	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A ✕
3	UP	1	2	B ✕
4	BEGIN-CP			
5	SOT	3		
6	UP	1	3	C ✕
7	SOT	2		Flush刷脏
8	END-CP			
9	UP	2	7	D ✓
10	EOT	1	6	
11	UP	3	5	B ✓
12	UP	2	9	A ✓
13	EOT	2	12	
14	UP	3	11	E ✓

脏页表

页面号	RecLSN
A	12
B	11
D	9
E	14

磁盘

页面	PageLSN
A	2
B	3
C	6
D	?
E	?

不需要Redo 的情况

- ① Page 不在脏页表中
- ② LSN < RecLSN: 页面在检查点之前已刷新
- ③ LSN ≤ PageLSN: 页面在检查点之后已刷新

其他都需要redo

案例说明（撤销阶段）

LSN	Type	TID	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E



事务表

LastLSN	TID
14	3

案例说明（撤销阶段）

LSN	Type	TID	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E
15	CLR	3	14	E, 11

事务表

LastLSN	TID
15	3

案例说明（撤销阶段）

LSN	Type	TID	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E
15	CLR	3	14	E, 11
16	CLR	3	15	B, 5

事务表

LastLSN	TID
16	3

案例说明（撤销阶段）

LSN	Type	TID	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	END-CP			
9	UP	2	7	D
10	EOT	1	6	
11	UP	3	5	B
12	UP	2	9	A
13	EOT	2	12	
14	UP	3	11	E
15	CLR	3	14	E, 11
16	CLR	3	15	B, 5
17	EOT	3	16	

事务表

LastLSN	TID

目录

1. 事务原子性和持久性的实现
2. 数据库故障恢复机制概述
3. 单机系统崩溃恢复方法
4. ARIES恢复算法
- 5. 数据库备份技术**
6. 数据库多机恢复

数据库备份

□ 问题：ARIES只能处理内存数据丢失一类的故障，无法处理磁盘数据丢失（此时日志数据产生了损坏）

□ 解决方法：RAID技术、Paxos技术

□ 问题：机器不能重启了？

- 备份数据库
- 创建数据副本
- 当数据损坏时使用副本还原数据

常用备份技术

□ 冷备份

- 备份前需要结束所有数据库中的事务
- 限制较多，影响在线业务

□ 热备份

- 备份过程中事务
- 在备份数据的同时也备份数据库重做日志

□ 全量备份：备份过程中备份所有数据

□ 增量备份：在之前备份的基础上备份增量数据

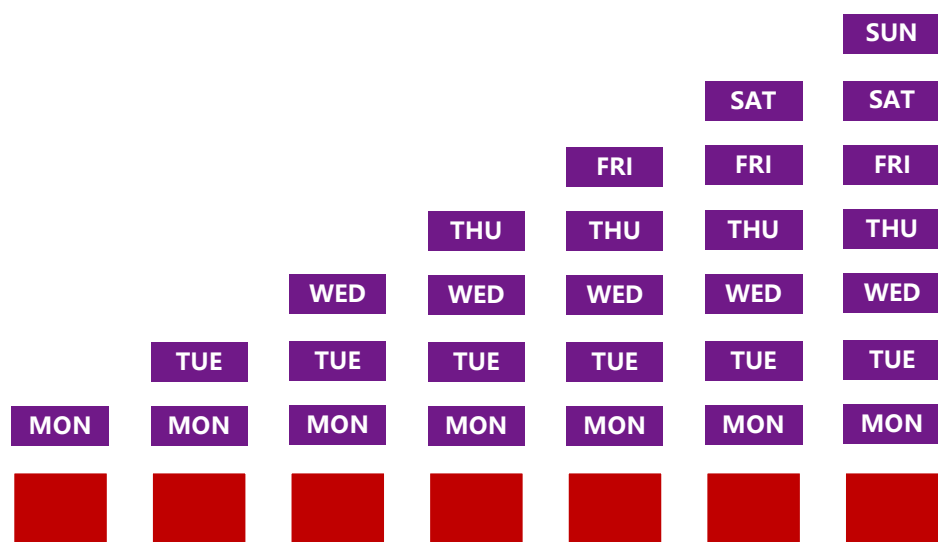
□ 差异备份：在全量备份的基础上备份差异的数据

增量备份

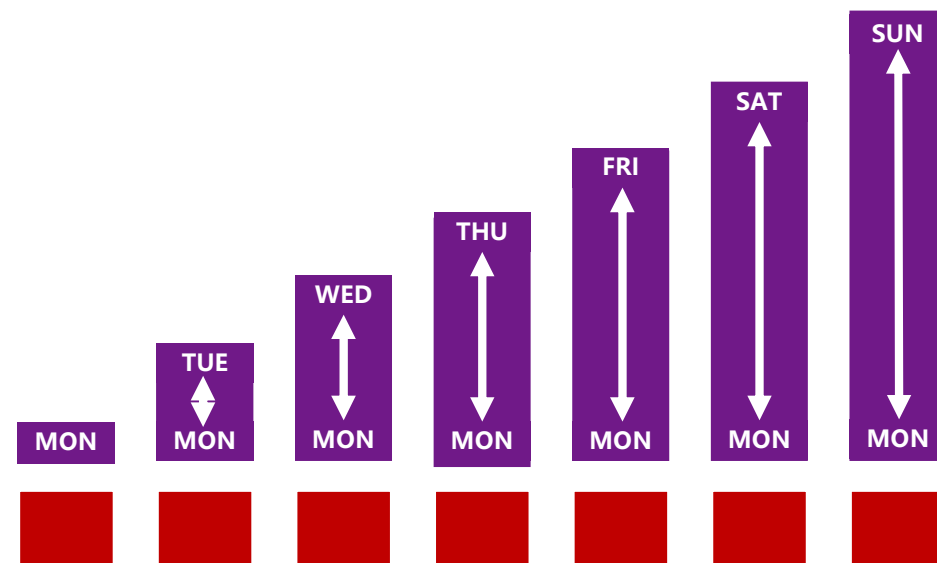


- 页面一、三被数据库修改，打上了标记
- 增量备份只需单独备份页面一、三，不需要备份页面二、四、五

增量备份 vs 差异备份



历史备份 增量备份



历史备份 差异备份

备份恢复

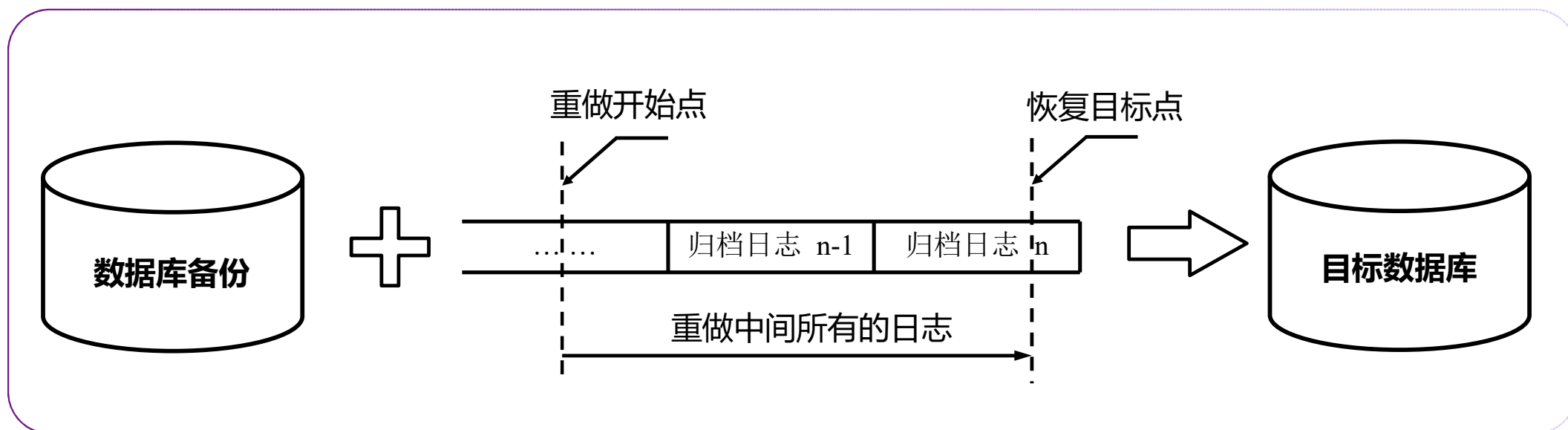
□ 利用数据和日志恢复数据

□ 具体步骤

- (1) 找到最近的数据库全量备份, 并根据它来恢复数据库 (即将备份拷贝到数据库)。
- (2) 如果有后续的增量备份, 按照从前往后做的顺序, 根据各个增量备份修改数据库。

基于时间点的恢复

- 一种灵活的恢复技术,支持将数据库回推到任意时刻
- 使用数据备份（全量和增量）和带时间戳的重做日志
- 首先，通过数据备份到恢复某一天
- 然后，通过日志恢复到某一个时刻



基于时间点恢复举例

原始表

Name	ID	Grade
Mike	13213	90
James	23419	80
Bob	55217	75

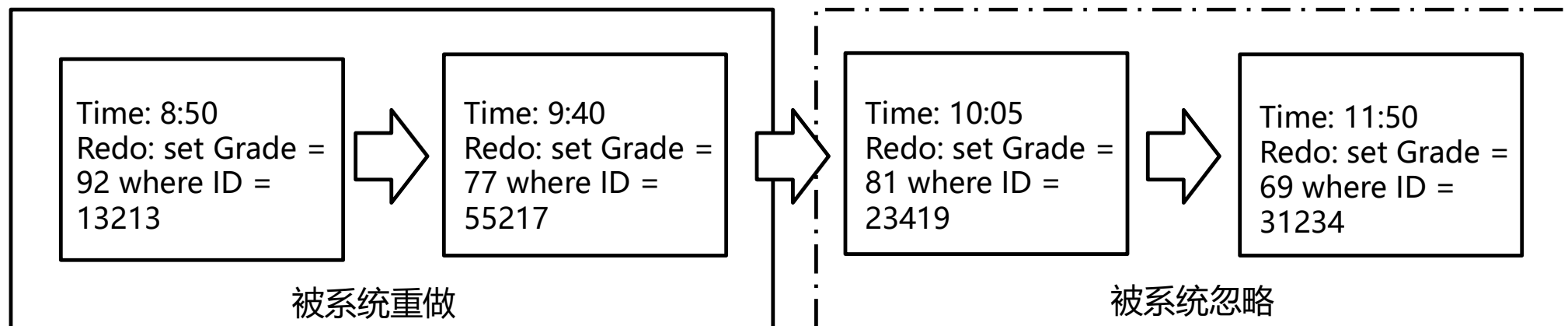
恢复到10:00
时刻的内容



恢复后的表

Name	ID	Grade
Mike	13213	92
James	23419	80
Bob	55217	77

重做日志相关内容



数据库闪回

□轻量级的数据库回滚技术

□借助undo回滚日志、回收站来实现不同粒度的闪回

- (1) 闪回查询：查询过去某个时刻的数据。
- (2) 闪回数据归档：对于undo回滚数据进行归档，使闪回功能支持的时间延长。
- (3) 闪回事务查询：查询过去执行过的某一事务相关的数据。
- (4) 闪回版本查询：查询某个版本范围内的数据。
- (5) 闪回表：将一张数据表的内容回退到一个指定时刻的状态。
- (6) 闪回删除：恢复一张已经被删除的表。
- (7) 闪回数据库：将数据库的所有内容回退到一个指定的时刻。

闪回技术实现方法

□ 依靠回滚段（UNDO SEGMENT）实现，其中保存了撤销数据

- 回滚数据是反转DML语句结果所需的信息，只要某个事务修改了数据，那么更新前的原有数据就会被写入一个撤销段
- 回滚段的数据保存在磁盘特定目录中，可以按需求配置撤销段存储空间大小和保存时间

□ 闪回日志（Flashback Log）

- 闪回日志注重捕获和保留特定表的历史行版本和变更过程
- 撤销日志负责记录事务级别的更改并支持事务一致性
- 闪回日志是页面粒度

数据库闪回实现方法

- 闪回查询：通过时间戳查询过去某个时间点的数据，可以找回由于意外删除或更改而丢失的数据，一般通过**回滚段实现**。
- 闪回版本查询：查询某个时间段内某个数据所有的版本信息，提供了查询一个数据随时间变化的方法，一般通过**回滚段实现**。
- 闪回事务：回滚一个指定的事务，一般通过**回滚段实现**。
- 闪回事务查询：查询过去执行过的某一事务相关执行信息，用于追溯某个可疑事务，一般通过**回滚段实现**。
- 闪回表：将一张数据表的内容回退到一个指定时刻的状态，一般通过**回滚段实现**。

数据库闪回实现方法

- ❑ 闪回删除：恢复一张已经被（误）删除的表，一般通过**回收站实现**（删除表时不是真正的物理删除，而是逻辑删除，放到回收站）。
- ❑ 闪回数据库：将数据库的所有内容回退到一个指定时刻。开启数据库闪回时，系统会记录**闪回日志**，它将数据库所有修改信息保存到快速恢复区（不同于undo日志根据记录来组织，闪回日志按照页面粒度，将所有修改保存到一起便于快速闪回）。闪回日志有大小和时间限制，所以闪回也有时间限制。
- ❑ 闪回数据归档：将数据库恢复到某一时刻的状态。由于undo有保留时间限制，定期会被删除，时间久远的数据无法通过undo闪回。因此一般通过对**undo回滚数据进行归档**，延长闪回功能支持的时间。

目录

1. 事务原子性和持久性的实现
2. 数据库故障恢复机制概述
3. 单机系统崩溃恢复方法
4. ARIES恢复算法
5. 数据库备份技术
- 6. 数据库多机恢复**

数据库多机恢复概述

- ❑ 数据库备份可以处理磁盘故障，但是备份数据在地理上非常接近
- ❑ 一旦出现火灾、洪水等灾害，距离接近的数据很有可能同时损坏
- ❑ 数据备份在恢复过程中无法对外提供服务，影响高可用性
- ❑ 解决方案：数据库多机恢复

主备模式架构

□主站点 (primary site)

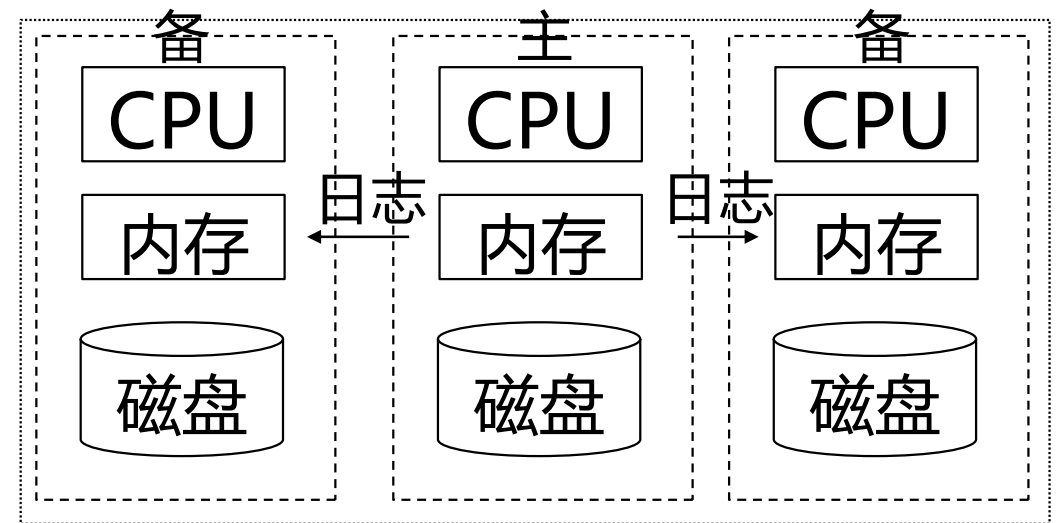
用来处理用户发出的请,

□备份站点 (backup site)

用来做主站点数据的备份,
包含了和主站点完全相同
的数据。

□所有主站点发生的更新会

以日志的形式经由网络发
送到备份站点上。

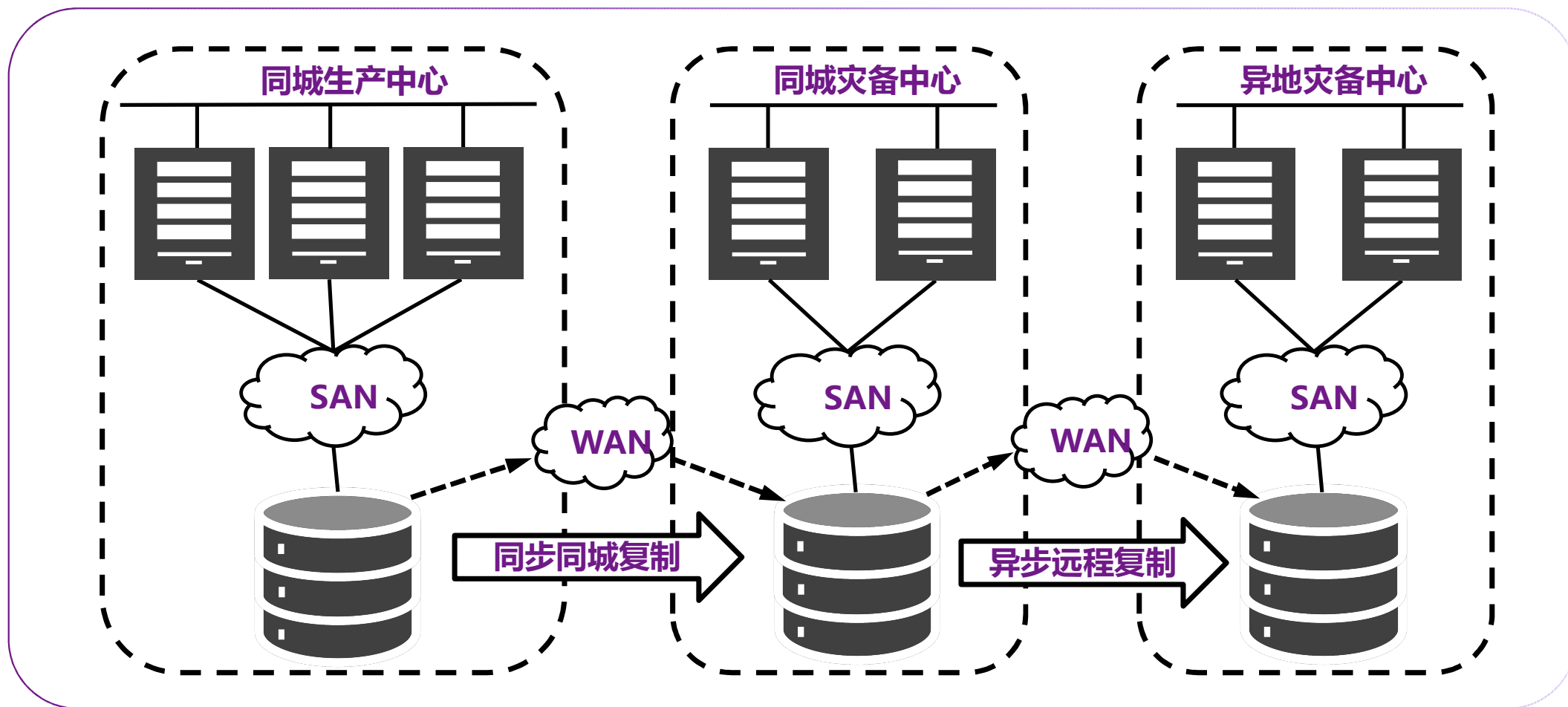


一主多备

两地三中心

- 在同城、异地建立双数据备份中心，比主备模式更高的安全性
- 更大的执行代价，需要维护三台以上主机的数据
- 同城数据传输和异地数据传输代价
 - 同城之间采用同步传输，所有数据不会丢失
 - 异地之间采用异步传输，少量数据可能会丢失

两地三中心



异地多活

□ 一种多机容灾架构

□ 支持多节点读写

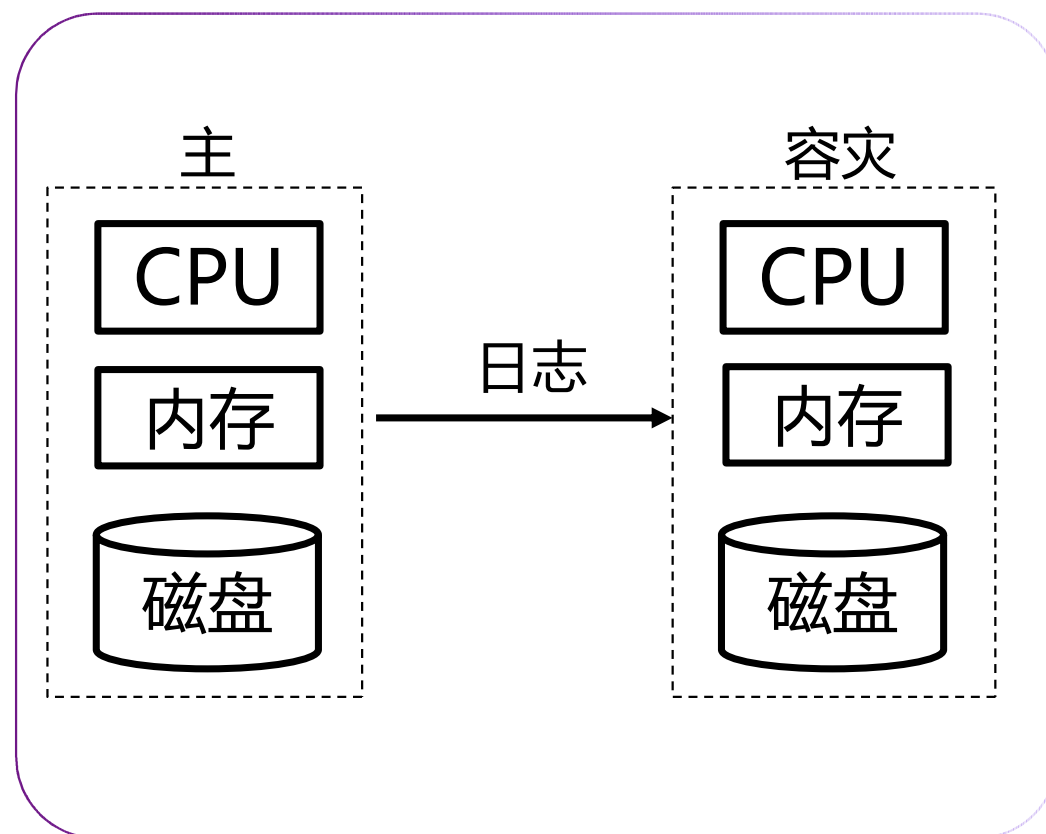
□ 挑战：

- 数据同步
- 多节点带来数据一致性的问题

□ 解决方案：

– 双集群，逻辑日志复制

- 同城：强同步， $RPO=0$
- 异地：异步， $RPO>0$ (物理距离限制)



小结

- 本章介绍了事务原子性和持久性在各种场景下的实现方法
- 本章介绍了针对系统崩溃的四种不同故障恢复算法
- 本章介绍了数据库备份各种技术，包括热备份、冷备份、全量备份、增量备份、差异备份
- 本章介绍了数据库容灾的架构，包括主备模式、两地三中心、异地多活