第9章: 故障恢复

Failure Recovery

邹兆年

哈尔滨工业大学
计算机科学与技术学院
海量数据计算研究中心
电子邮件: znzou@hit.edu.cn

2023年秋

教学内容[1]

[1]课件更新于2023年11月30日

Failures

故障(Failure)

故障可能会破坏数据库的一致性

已提交的(committed)事务对数据库的修改未全部持久化到磁盘

- 破坏事务的持久性

已中止的(aborted)事务对数据库的修改已部分持久化到磁盘

- 破坏事务的原子性

故障的类型

- 事务故障(transaction failures)
- 系统故障(system failures)
- 存储介质故障(storage media failures)

事务故障(Transaction Failure)

逻辑错误(logical error)

- 事务由于内部错误(internal error)而无法完成，如违反完整性约束

内部状态错误(internal state error)

- DBMS由于内部状态错误(如死锁)而必须中止活跃(active)事务

系统故障(System Failure)

软件故障(software failure)

- DBMS实现的bug所导致的故障

硬件故障(hardware failures)

- 运行DBMS的计算机发生崩溃(crash)，如断电
- 假设系统崩溃不会损坏非易失存储器中的数据

存储介质故障(Storage Media Failures)

存储介质故障

- 非易失存储器发生故障，损坏了存储的数据
- 假设数据损坏可以被检测，如使用校验和(checksum)
- 任何DBMS都无法从这种故障中恢复，必须从备份(archive)中还原(restore)

Buffer Pool Policies

Undo? Redo?

DBMS在进行故障恢复时会执行两种操作

撤销(Undo)

- 撤销(Undo)未完成事务(incomplete txn)对数据库的修改

重做(Redo)

- 重做(Redo)已提交事务(committed txn)对数据库的修改

DBMS如何运用undo和redo取决于DBMS如何管理缓冲池(buffer pool)

- 只需undo?
- 只需redo?
- 既要undo，又要redo?
- 既不用undo，也不用redo?

缓冲池(Buffer Pool)

事务调度

T_1	T_2
BEGIN	
r(A)	
A := A + 1	
w(A)	BEGIN
	r(B)
	B := B * 2
	w(B)
	COMMIT
ABORT	

缓冲池

A=2	B=4	C=3

磁盘

A=1	B=2	C=3

- 是否强制(force)在T_2提交时将B写回磁盘? ⟵ FORCE策略
- 是否允许在T_1提交前覆写磁盘上A的值? ⟵ STEAL策略

STEAL/NO-STEAL策略

DBMS是否允许将未提交事务所做的修改写到磁盘并覆盖现有数据?

- **STEAL**: 允许
- **NO-STEAL**: 不允许

FORCE/NO-FORCE策略

DBMS是否强制事务在提交前必须将其所做的修改全部写回磁盘?

- **FORCE**: 强制
- **NO-FORCE**: 不强制

缓冲池策略(Buffer Pool Policies)

缓冲池效率高	STEAL + FORCE	STEAL + NO-FORCE
缓冲池效率低	NO-STEAL + FORCE	NO-STEAL + NO-FORCE
	I/O效率低	I/O效率高

NO-STEAL + FORCE

- NO-STEAL $\Longrightarrow$ 未提交事务不可能将其修改写回磁盘$\Longrightarrow$ 无需undo
- FORCE $\Longrightarrow$ 已提交事务已将其修改全部写回磁盘$\Longrightarrow$ 无需redo

事务调度

T_1	T_2
BEGIN	
r(A)	
A := A + 1	
w(A)	BEGIN
	r(B)
	B := B * 2
	w(B)
	COMMIT
ABORT	

缓冲池

A=2	B=4	C=3

磁盘

A=1	B=4	C=3

- 优点: 实现简单
- 缺点: 缓冲池得能存得下所有未提交事务所做的修改

Write-Ahead Logging (WAL)

预写式日志(Write-Ahead Log, WAL)

DBMS在数据文件之外维护一个日志文件(log file)，用于记录事务对数据库的修改

- 假定日志文件存储在稳定存储器(stable stroage)中
- 日志记录(log record)包含undo或redo时所需的信息

DBMS在将修改过的对象写到磁盘之前，必须先将修改此对象的日志记录刷写到磁盘

WAL协议(WAL Protocol)

当事务T_i启动时，向日志中写入记录<tid, BEGIN>

- tid: T_i的ID (txn ID)

当T_i提交时，向日志中写入记录<tid, COMMIT>

- 在DBMS向应用程序返回确认消息之前，必须保证T_i的所有日志记录都已刷写到磁盘
- <tid, COMMIT>写入磁盘才代表事务提交

当T_i修改对象A时，向日志中写入记录<tid, oid, before, after>

- oid: A的ID (object ID)
- before: A修改前的值(undo时用)
- after: A修改后的值(redo时用)

基于WAL的故障恢复

第1部分: 事务正常执行时的行为

- 记录日志
- 按照缓冲池策略将修改过的对象写到磁盘

第2部分: 故障恢复时的行为

- 根据日志和缓冲池策略，对事务进行undo或redo

事务的分类

根据日志将事务分为3类

已提交事务(committed txn)

- 既有<T, BEGIN>，又有<T, COMMIT>

不完整事务(incomplete txn)

- 只有<T, BEGIN>，而没有<T, COMMIT>

已中止事务(aborted txn)

- 既有<T, BEGIN>，又有<T, ABORT>
- 在事务正常执行和故障恢复过程中，如果T所做的修改已全部撤销，则将日志记录<T, ABORT>写到日志
- 已中止事务相当于从未执行过，故不需要undo，更不需要redo

故障恢复时的行为

已提交事务

- 如果一个已提交事务的修改已全部写到磁盘，则无需redo
- 否则，需要redo

不完整事务

- 如果一个不完整事务的任何修改都未写到磁盘，则无需undo
- 否则，需要undo

缓冲池策略决定了上述行为

WAL协议的分类

根据缓冲池策略的不同，可以实现三类WAL协议

- Undo Logging: WAL + STEAL + FORCE
- Redo Logging: WAL + NO-STEAL + NO-FORCE
- Redo+Undo Logging: WAL + STEAL + NO-FORCE

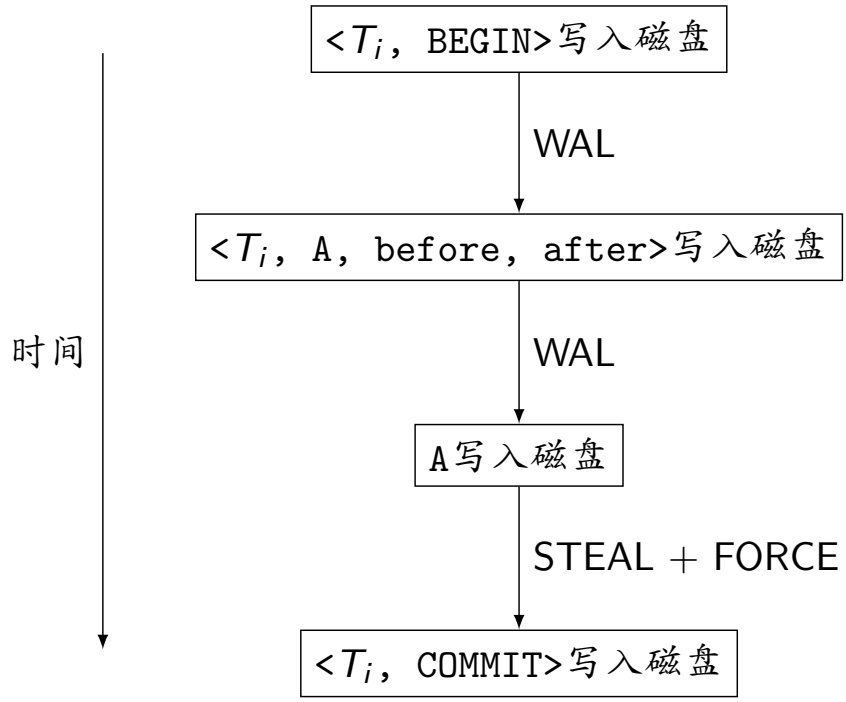
Write-Ahead Logging (WAL)
Undo Logging

Undo Logging

Undo Logging = WAL + STEAL + FORCE

缓冲池效率高	STEAL + FORCE	STEAL + NO-FORCE
缓冲池效率低	NO-STEAL + FORCE	NO-STEAL + NO-FORCE
	I/O效率低	I/O效率高

基于Undo Logging的事务正常执行时的行为



基于Undo Logging的事务正常执行时的行为

Example (Undo Logging)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11	COMMIT						<T, COMMIT>
12	FLUSH LOG						

基于Undo Logging的故障恢复

已提交事务(Committed Txn): 不需要恢复

- FORCE $\Longrightarrow$ 已提交事务所做的修改已全部写入磁盘

不完整事务(Incomplete Txn): 全部undo

- STEAL $\Longrightarrow$ 不完整事务所做的一部分修改可能已经写入磁盘

基于Undo Logging的故障恢复方法

从后(最后一条记录)向前(第一条记录)扫描日志

根据每条日志记录的类型执行相应的动作

- <T, COMMIT>: 将T记录为已提交事务(无需redo)
- <T, ABORT>: 将T记录为已中止事务(无需undo)
- <T, A, before, after>: 如果T是不完整事务, 则将磁盘上A的值恢复为before
- <T, BEGIN>: T恢复完毕; 如果T是不完整事务, 则向日志中写入<T, ABORT> (今后故障恢复时无需再undo)

基于Undo Logging的故障恢复

Example (基于Undo Logging的故障恢复)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11	COMMIT						<T, COMMIT>
12	FLUSH LOG						
	Crash!						

基于Undo Logging的故障恢复

Example (基于Undo Logging的故障恢复)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
	Crash!						

邹兆年 (CS@HIT) 第9章: 故障恢复 2023年秋 29 / 59

基于Undo Logging的故障恢复

Example (基于Undo Logging的故障恢复)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
	Crash!						

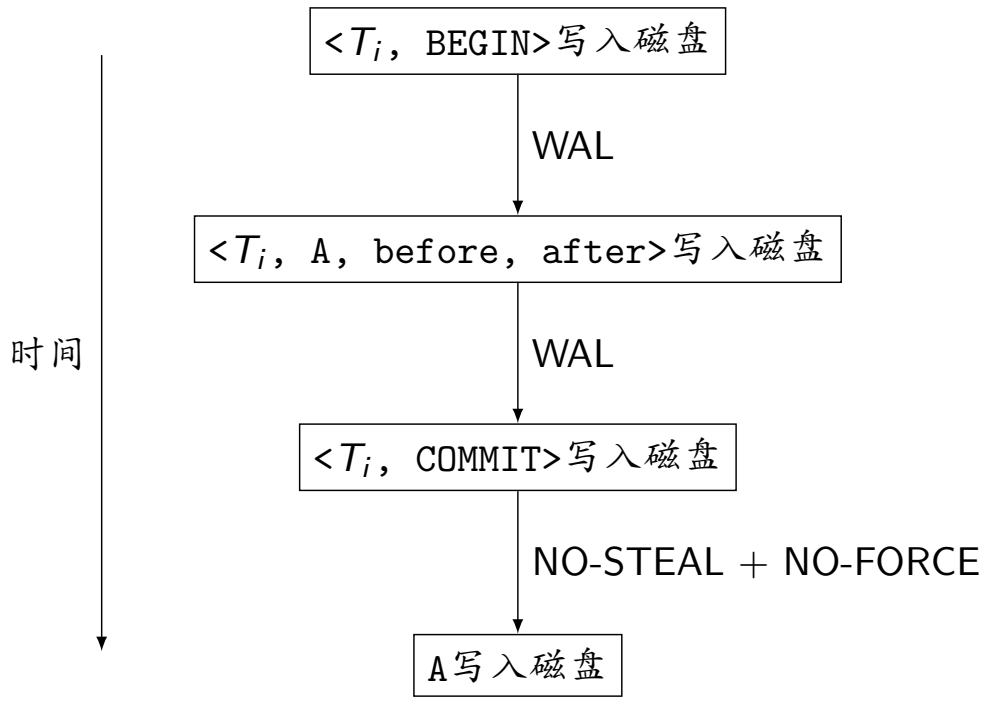
Write-Ahead Logging (WAL)
Redo Logging

Redo Logging

Redo Logging = WAL + NO-STEAL + NO-FORCE

缓冲池效率高	STEAL + FORCE	STEAL + NO-FORCE
缓冲池效率低	NO-STEAL + FORCE	NO-STEAL + NO-FORCE
	I/O效率低	I/O效率高

事务正常执行时的行为



基于Redo Logging的事务正常执行时的行为

Example (Redo Logging)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	COMMIT						<T, COMMIT>
9	FLUSH LOG						
10	OUTPUT(A)	16	16	16	16	8	
11	OUTPUT(B)	16	16	16	16	16	

基于Redo Logging的故障恢复

已提交事务(Committed Txn): 全部redo

- NO-FORCE $\Longrightarrow$ 已提交事务所做的修改可能尚未全部写入磁盘

不完整事务(Incomplete Txn): 不需要恢复

- NO-STEAL $\Longrightarrow$ 不完整事务所做的任何修改都未写入磁盘

基于Redo Logging的故障恢复方法

从前(第一条记录)向后(最后一条记录)扫描日志两遍

第1遍扫描: 记录已提交事务和已中止事务

- <T, COMMIT>: 将T记录为已提交事务(需要redo)
- <T, ABORT>: 将T记录为已中止事务(无需undo)

第2遍扫描: 根据每条日志记录的类型执行相应的动作

- <T, A, before, after>: 如果T是已提交事务，则将磁盘上A的值覆写为after
- <T, BEGIN>: 如果T是不完整事务，则向日志中写入<T, ABORT>

基于Redo Logging的故障恢复

Example (基于Redo Logging的故障恢复)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	COMMIT						<T, COMMIT>
9	FLUSH LOG						
10	OUTPUT(A)	16	16	16	16	8	
11	OUTPUT(B)	16	16	16	16	16	
	Crash!						

基于Redo Logging的故障恢复

Example (基于Redo Logging的故障恢复)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	COMMIT						<T, COMMIT>
9	FLUSH LOG						
	Crash!						

基于Redo Logging的故障恢复

Example (基于Redo Logging的故障恢复)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
	Crash!						

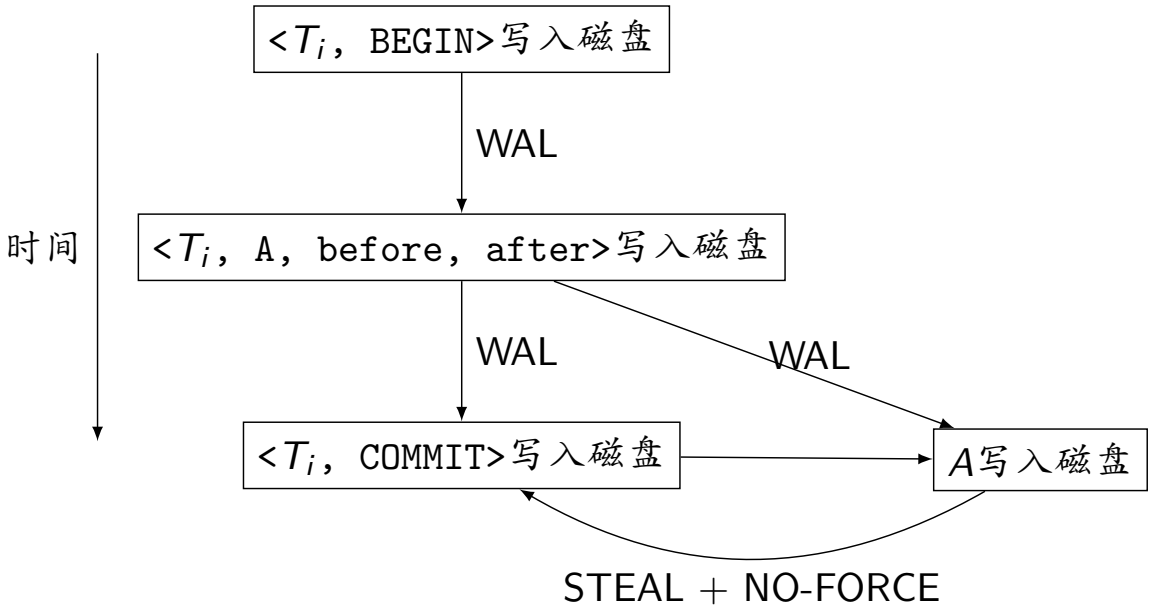
Write-Ahead Logging (WAL)
Redo+Undo Logging

Redo+Undo Logging

Redo+Undo Logging = WAL + STEAL + NO-FORCE

缓冲池效率高	STEAL + FORCE	STEAL + NO-FORCE
缓冲池效率低	NO-STEAL + FORCE	NO-STEAL + NO-FORCE
	I/O效率低	I/O效率高

基于Redo+Undo Logging的事务正常执行时的行为



基于Redo+Undo Logging的事务正常执行时的行为

Example (Redo+Undo Logging)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	OUTPUT(A)	16	16	16	16	8	
9	COMMIT						<T, COMMIT>
10	FLUSH LOG						
11	OUTPUT(B)	16	16	16	16	16	

基于Redo+Undo Logging的故障恢复

已提交事务(Committed Txn): 全部redo

- NO-FORCE $\Longrightarrow$ 已提交事务所做的修改可能尚未全部写入磁盘

不完整事务(Incomplete Txn): 全部undo

- STEAL $\Longrightarrow$ 不完整事务所做的一部分修改可能已经写入磁盘

基于Redo+Undo Logging的故障恢复方法

Redo阶段: redo已提交事务

- 与基于Redo Logging的故障恢复方法相同

Undo阶段: undo不完整事务

- 与基于Undo Logging的故障恢复方法相同

基于Redo+Undo Logging的故障恢复

Example (基于Redo+Undo Logging的故障恢复)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	OUTPUT(A)	16	16	16	16	8	
9	COMMIT						<T, COMMIT>
10	FLUSH LOG						
11	OUTPUT(B)	16	16	16	16	16	
	Crash!						

基于Redo+Undo Logging的故障恢复

Example (基于Redo+Undo Logging的故障恢复)

Step	Action	t	M_A	M_B	D_A	D_B	Log
1							<T, BEGIN>
2	READ(A, t)	8	8		8	8	
3	t := t * 2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8, 16>
5	READ(B, t)	8	16	8	8	8	
6	t := t * 2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8, 16>
8	OUTPUT(A)	16	16	16	16	8	
	Crash!						

缓冲池策略的比较

运行时效率

	FORCE	NO-FORCE
STEAL	–	Fastest
NO-STEAL	Slowest	–

故障恢复效率

	FORCE	NO-FORCE
STEAL	–	Slowest
NO-STEAL	Fastest	–

几乎所有DBMS都采用STEAL + NO-FORCE

组提交(Group Commit)

每条日志记录单独刷写(flush)到磁盘的I/O开销太大

在内存中设置日志缓冲区(log buffer)，将日志记录写到日志缓冲区，然后成批刷写到日志文件

- 日志缓冲区满时刷写
- 定时刷写

Checkpoints

WAL的问题

- 日志永远在变大
- 故障恢复时需要扫描日志，恢复时间越来越长

Example (WAL)

<T_1, BEGIN>
<T_1, A, 5, 15>
<T_2, BEGIN>
<T_2, B, 10, 20>
<T_2, C, 15, 25>
<T_2, COMMIT>
<T_3, BEGIN>
<T_1, D, 20, 30>
<T_3, E, 25, 35>
<T_1, COMMIT>
<T_3, F, 30, 40>

如果使用Undo Logging，则扫描到这里即可

检查点(Checkpoints)

DBMS定期设置检查点(checkpoint)

- 将日志刷写到磁盘
- 根据缓冲池策略，将脏页(dirty page)写到磁盘
- 故障恢复时只需扫描到最新的检查点

模糊检查点(Fuzzy Checkpoints)

检查点开始: 向日志中写入<BEGIN CHECKPOINT ($T_1, T_2, \ldots, T_n$)>

- $T_1, T_2, \ldots, T_n$是检查点开始时的活跃事务(active txn)
- 活跃事务是尚未提交或中止的事务

检查点中间: 根据缓冲池策略，将脏页(dirty page)写到磁盘

- 如果采用STEAL，则将全部脏页写到磁盘
- 否则，只将已提交事务所做的修改写到磁盘

检查点结束: 向日志中写入<END CHECKPOINT>，并将日志刷写到磁盘

- 如果采用NO-FORCE，则写完全部脏页后即可结束检查点
- 否则，在$T_1, T_2, \ldots, T_n$全部提交后，才能结束检查点

涉及检查点的故障恢复

缓冲池策略: STEAL + NO-FORCE (自行思考其它策略下的恢复方法)

Redo阶段: redo已提交事务

- 从前向后扫描日志
- 从哪条日志记录开始?

Undo阶段: undo不完整事务

- 从后向前扫描日志
- 到哪条日志记录为止?

Redo阶段

日志中最新的完整检查点

<BEGIN CHECKPOINT ($T_1, T_2, \ldots, T_n$)>
...
<END CHECKPOINT>

需要redo的最早的事务一定属于$\{T_1, T_2, \ldots, T_n\}$

从日志记录<BEGIN CHECKPOINT ($T_1, T_2, \ldots, T_n$)>开始向后扫描日志

- 不需要从最早的<T_i, BEGIN>开始扫描

证明 I

需要redo的最早的事务T一定属于$\{T_1, T_2, \ldots, T_n\}$

Log	Fact
<T, BEGIN>	
...	
<T, COMMIT>	
...	
<BEGIN CHECKPOINT ($T_1, T_2, \ldots, T_n$)>	$T \notin \{T_1, T_2, \ldots, T_n\}$
...	
<END CHECKPOINT>	T所做的修改已全部写到磁盘，无需redo

证明 II

Log	Fact
<T, BEGIN>	
...	
<BEGIN CHECKPOINT ($T_1, T_2, \ldots, T_n$)>	$T \in \{T_1, T_2, \ldots, T_n\}$
...	
<T, COMMIT>	
...	
<END CHECKPOINT>	T所做的修改已全部写到磁盘，无需redo

证明 III

Log	Fact
<T, BEGIN>	
...	
<BEGIN CHECKPOINT ($T_1, T_2, \ldots, T_n$)>	$T \in \{T_1, T_2, \ldots, T_n\}$
...	
<END CHECKPOINT>	
...	
<T, COMMIT>	T所做的修改未必全部写到磁盘，必须redo

Undo阶段

日志中最新的完整检查点

<BEGIN CHECKPOINT ($T_1, T_2, \ldots, T_n$)>
...
<END CHECKPOINT>

需要undo的最早的事务一定属于$\{T_1, T_2, \ldots, T_n\}$

扫描到$T_1, T_2, \ldots, T_n$中最早的事务T_i的日志记录<T_i, BEGIN>为止

证明

需要undo的最早的事务T一定属于$\{T_1, T_2, \ldots, T_n\}$

Log	Fact
<T, BEGIN>	
...	
<BEGIN CHECKPOINT ($T_1, T_2, \ldots, T_n$)>	$T \in \{T_1, T_2, \ldots, T_n\}$
...	
<END CHECKPOINT>	T所做的部分修改可能已写到磁盘，必须undo

涉及检查点的故障恢复—Redo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Redo阶段)

Log	Redo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	
<T_3, B, 10, 20>	
<BEGIN CHECKPOINT (T_2, T_3)>	从这里开始redo
<T_2, C, 15, 25>	
<T_3, D, 20, 30>	
<END CHECKPOINT>	
<T_2, COMMIT>	

涉及检查点的故障恢复—Redo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Redo阶段)

Log	Redo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	
<T_3, B, 10, 20>	
<BEGIN CHECKPOINT (T_2, T_3)>	从这里开始redo
<T_2, C, 15, 25>	C ← 25
<T_3, D, 20, 30>	
<END CHECKPOINT>	
<T_2, COMMIT>	

涉及检查点的故障恢复—Redo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Redo阶段)

Log	Redo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	
<T_3, B, 10, 20>	
<BEGIN CHECKPOINT (T_2, T_3)>	从这里开始redo
<T_2, C, 15, 25>	C ← 25
<T_3, D, 20, 30>	T_3无需redo
<END CHECKPOINT>	
<T_2, COMMIT>	

涉及检查点的故障恢复—Redo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Redo阶段)

Log	Redo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	
<T_3, B, 10, 20>	
<BEGIN CHECKPOINT (T_2, T_3)>	从这里开始redo
<T_2, C, 15, 25>	C $\leftarrow$ 25
<T_3, D, 20, 30>	T_3无需redo
<END CHECKPOINT>	
<T_2, COMMIT>	T_2 redo完毕

涉及检查点的故障恢复—Undo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Undo阶段)

Log	Undo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	
<T_3, B, 10, 20>	
<BEGIN CHECKPOINT (T_2, T_3)>	
<T_2, C, 15, 25>	
<T_3, D, 20, 30>	
<END CHECKPOINT>	
<T_2, COMMIT>	T_2无需undo

涉及检查点的故障恢复—Undo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Undo阶段)

Log	Undo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	
<T_3, B, 10, 20>	
<BEGIN CHECKPOINT (T_2, T_3)>	
<T_2, C, 15, 25>	
<T_3, D, 20, 30>	D ← 20
<END CHECKPOINT>	
<T_2, COMMIT>	T_2无需undo

涉及检查点的故障恢复—Undo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Undo阶段)

Log	Undo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	
<T_3, B, 10, 20>	
<BEGIN CHECKPOINT (T_2, T_3)>	
<T_2, C, 15, 25>	T_2无需undo
<T_3, D, 20, 30>	D ← 20
<END CHECKPOINT>	
<T_2, COMMIT>	T_2无需undo

涉及检查点的故障恢复—Undo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Undo阶段)

Log	Undo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	
<T_3, B, 10, 20>	B ← 10
<BEGIN CHECKPOINT (T_2, T_3)>	
<T_2, C, 15, 25>	T_2无需undo
<T_3, D, 20, 30>	D ← 20
<END CHECKPOINT>	
<T_2, COMMIT>	T_2无需undo

涉及检查点的故障恢复—Undo阶段

缓冲池策略: STEAL + NO-FORCE

Example (涉及检查点的故障恢复—Undo阶段)

Log	Undo Action
<T_1, BEGIN>	
<T_1, A, 5, 15>	
<T_2, BEGIN>	
<T_1, COMMIT>	
<T_3, BEGIN>	T_3 undo完毕，写<T_3, ABORT>
<T_3, B, 10, 20>	B ← 10
<BEGIN CHECKPOINT (T_2, T_3)>	
<T_2, C, 15, 25>	T_2无需undo
<T_3, D, 20, 30>	D ← 20
<END CHECKPOINT>	
<T_2, COMMIT>	T_2无需undo

总结

1. Failures
2. Buffer Pool Policies
3. Write-Ahead Logging (WAL)
   - Undo Logging
   - Redo Logging
   - Redo+Undo Logging
4. Checkpoints