# 第12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

#### 2020年春

邹兆年 (CS@HIT)

12章· 故障恢复

2020年春

<ロ > < 部 > < き > < き > し > を の く で

1 / 69

# 教学内容1

- Overview
- 2 Failures
- Buffer Pool Policies
- Write-Ahead Logging (WAL)
  - Undo Logging
  - Redo Logging
  - Redo+Undo Logging
- 6 Checkpoints

1课件更新于2020年4月19日

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

#### Overview

邹兆年 (CS@HIT)

512章: 故障恢复

2020年春

3 / 69

# 事务(Transactions)

事务(transaction)是在数据库上执行的一个或多个操作构成的序列,用来完成数据库系统的高级功能

• 事务的操作要么全部执行,要么一个也不执行

## Example (转账事务)

账户A给账户B转账100元

- ❶ 检查账户A余额是否够100元
- ② 从账户A中减去100元
- ③ 在账户B中增加100元

郭兆年 (CS@HIT)第12章

2020年春

# SQL事务语句(Transactional Statements)

#### 事务启动(start): BEGIN;

#### 事务提交(commit): COMMIT;

• 将事务对数据库的修改持久地写到数据库中

#### 事务中止(abort): ROLLBACK;

- 将事务对数据库的修改全部撤销(undo),就像事务从未执行过
- 事务可以中止自己,也可以被DBMS中止

◆□▶ ◆□▶ ◆ ■ ▶ ● ● りへ○

邹兆年 (CS@HIT)

512章: 故障恢复

2020年春

5 / 69

# 事务的ACID性质(The ACID Properties)

#### 原子性(Atomicity): "all or nothing"

• 事务的操作要么全部执行,要么一个也不执行

#### 一致性(Consistency): "it looks correct to me"

如果事务的程序正确,并且事务启动时数据库处于一致状态(consistent state),则事务结束时数据库仍处于一致状态

#### 隔离性(Isolation): "as if alone"

• 一个事务的执行不受其他事务的干扰

#### 持久性(Durability): "survive failures"

• 事务一旦提交,它对数据库的修改一定全部持久地写到数据库中

# 一致性(Consistency)

#### "It Looks Correct to Me"

用户(user)保证事务的一致性

• 别写错程序

◆□▶ ◆□▶ ◆ ≧ ▶ ◆ ≧ ▶ ◆ ○

邹兆年 (CS@HIT)

512章: 故障恢复

2020年春

7 / 69

# 隔离性(Isolation)

#### "As If Alone"

多个事务的执行有2种方式

- 串行执行(serial execution) ⇒ 不破坏隔离性
- 交叉执行(interleaving execution) ⇒ 可能破坏隔离性

DBMS保证事务的隔离性

• 并发控制(concurrency control): 确定多个事务的操作的正确交叉执 行顺序

## 第11章: 并发控制(Concurrency Control)

邹兆年 (CS@HIT)

第12音, お陪恢复

2020年春

# 原子性(Atomicity)

#### "All or Nothing"

事务的执行只有两种结局

- 执行完所有操作后提交⇒ 不破坏原子性
- 执行了部分操作后中止⇒ 破坏原子性

DBMS保证事务的原子性

• 中止事务(aborted txn)执行过的操作必须撤销(undo)

第12章: 故障恢复(Failure Recovery)

< ロ ト ◆ 個 ト ◆ 重 ト ◆ 重 ・ りへ ()

邹兆年 (CS@HIT)

512章·故障恢复

2020年春

9 / 69

# 持久性(Durability)

#### "Survive Failures"

故障(failure)导致事务对数据库的修改有4种结果

- 提交事务所做的修改已全部写入磁盘⇒ 不破坏持久性
- 提交事务所做的修改仅部分写入磁盘 ⇒ 破坏持久性
- 申止事务所做的修改有些已写入磁盘⇒→破坏持久性
- 中止事务所做的修改未写入磁盘⇒ 不破坏持久性

DBMS保证事务的持久性

- 重做(redo)提交事务对数据库的修改
- 撤销(undo)中止事务对数据库的修改

第12章: 故障恢复(Failure Recovery)

# 故障恢复(Failure Recovery)

故障可能会破坏数据库的一致性(consistency)

## 故障恢复(failure recovery)

故障发生后,DBMS将数据库恢复到最新的一致性状态

演示:

```
CREATE TABLE t (
  id INT PRIMARY KEY,
  val INT NOT NULL
);
SET autocommit=OFF;
```

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

11 / 69



▶ ◀ 臺 ▶ ◀ 臺 ▶ ■ ● の Q (

邹兆年 (CS@HIT)

第12音, 按 障 恢 复

2020年春

**Failures** 

◆□ > ◆□ > ◆■ > ◆■ > ■ からの

邹兆年 (CS@HIT)

[12章: 故障恢复

2020年春

13 / 69

# 故障(Failure)的类型

- 事务故障(Transaction Failures)
- 系统故障(System Failures)
- 存储介质故障(Storage Media Failures)

邹兆年 (CS@HIT) 第12章: 故障恢复

2020年春

# 事务故障(Transaction Failures)

#### 逻辑错误(Logical Errors)

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

#### 内部状态错误(Internal State Errors)

• DBMS由于内部状态错误(如死锁)而必须中止活跃事务(active txn)

邹兆年 (CS@HIT)

512章: 故障恢复

2020年春

15 / 69

# 系统故障(System Failures)

#### 软件故障(Software Failures)

• 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(Storage Media Failures)

- 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,如使用校验码(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)

邹兆年 (CS@HIT)

12章: 故障恢复

2020年春

17 / 69

**Buffer Pool Policies** 

**◆□▶ ◆□▶ ◆■▶ ◆■▶ ■ か**990

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

#### Undo? Redo?

DBMS在故障恢复时会做2种操作

#### 撤销(Undo)

• Undo未完成事务(incomplete txn)对数据库的修改

#### 重做(Redo)

• Redo已提交事务(committed txn)对数据库的修改

DBMS如何支持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?

邹兆年 (CS@HIT)

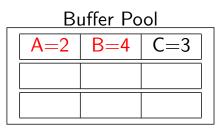
512章: 故障恢复

2020年春

19 / 69

# 缓冲池(Buffer Pool)

Transa	ctions
$T_1$	$T_2$
BEGIN	
r(A)	
A := A + 1	
w(A)	BEGIN
	r(B)
	B := B * 2
	w(B)
	CÒMMIT
ABORT	



	Disk		
A=1	B=2	C=3	
	l		

- 是否强制(force)在T2提交时将B写回磁盘? ← FORCE策略
- 是否允许在T<sub>1</sub>提交前覆写磁盘上A的值? ← STEAL策略

# 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

◆ロ ト ◆ 個 ト ◆ 重 ト ● ■ り へ で

邹兆年 (CS@HIT)

2020年春 21 / 69

# FORCE策略

DBMS是否要求事务在提交前必须将其所做的修改全部写回磁盘?

• FORCE: 要求

• NO-FORCE: 不要求

# 缓冲池策略(Buffer Pool Policies)

邹兆年 (CS@HIT)

312章:故障恢复

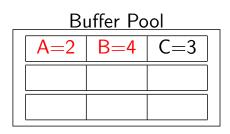
2020年春

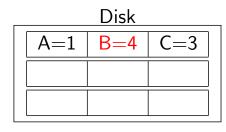
23 / 69

#### NO-STEAL + FORCE

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

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





• 优点: 实现简单

邹兆年 (CS@HIT)

• 缺点: 缓冲池得能存得下所有未提交事务所做的修改

第12章: 故障恢复

2020年春 24 / 69

Write-Ahead Logging (WAL)

4□ > 4□ > 4 = > 4 = > = 90

邹兆年 (CS@HIT)

512章: 故障恢复

2020年春

25 / 69

# 预写式日志(Write-Ahead Log, WAL)

DBMS在数据文件(data file)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

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春 26 / 69

# WAL协议(WAL Protocol)

当事务T;启动时,向日志中写入记录<tid, BEGIN>

• tid: Ti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;的所有日志记录都已刷写到磁盘

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)

邹兆年 (CS@HIT)

512章·故障恢复

2020年春

27 / 69

# 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

邹兆年 (CS@HIT)

第12音, 按隨恢复

2020年春 28 / 6

## 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

邹兆年 (CS@HIT)

512章·故障恢复

2020年表 20

# 故障恢复时的行为

#### 已提交事务(Committed Txn)

- 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, redo

# 不完整事务(Incomplete Txn)

- 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, 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3类WAL协议

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

邹兆年 (CS@HIT)

12章: 故障恢复

2020年春

31 / 69

Write-Ahead Logging (WAL)
Undo Logging

邹兆年 (CS@HIT)
第12章: □

2020年春

# **Undo Logging**

#### Undo Logging = WAL + STEAL + FORCE

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

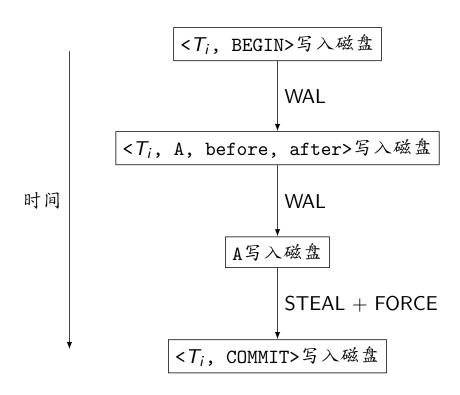
邹兆年 (CS@HIT)

812章: 故障恢复

2020年春

33 / 69

# 基于Undo Logging的事务正常执行时的行为



邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 基于Undo Logging的事务正常执行时的行为

Examp	le (Undo Loggi	ng)					
Step	Action	t	$M_A$	$M_B$	$D_A$	$D_B$	Log
1							<t, begin=""></t,>
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, 16="" 8,="" a,=""></t,>
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, 16="" 8,="" b,=""></t,>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11	COMMIT						<t, commit=""></t,>
12	FLUSH LOG						

 第12章: 故障恢复
 2020年春
 35 / 69

# 基于Undo Logging的故障恢复

#### 已提交事务(Committed Txn): 不需要恢复

● FORCE ⇒ 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 ⇒ 不完整事务所做的一部分修改可能已经写入磁盘

**◆□▶◆□▶◆■▶◆■▼ 990** 

# 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)

邹兆年 (CS@HIT)

812章: 故障恢复

2020年春

37 / 69

# 基于Undo Logging的故障恢复

#### Example (基于Undo Logging的故障恢复) Action $M_A$ $M_B$ $D_A$ Step $D_{B}$ Log t 1 <T, BEGIN> 2 READ(A, t) 8 8 8 8 3 t := t \* 216 8 8 8 WRITE(A, t) 16 8 4 16 8 <T, A, 8, 16> 5 READ(B, t) 8 16 8 8 8 6 t := t \* 216 16 8 8 8 <T, B, 8, 16> 7 16 16 16 8 WRITE(B, t) 8 8 FLUSH LOG 9 OUTPUT(A) 16 16 16 16 8 10 16 16 16 16 16 OUTPUT(B) 11 COMMIT <T, COMMIT> 12 FLUSH LOG Crash!

◆ロ → ◆ □ → ◆ き → ◆ き → り へ ()

邹兆年 (CS@HIT) \_\_\_\_\_

第12章: 故障恢复

2020年春 38 / 69

# 基于Undo Logging的故障恢复

Examp	le (基于Undo L	_oggi	ng的t	故障协	反复)		
Step	Action	t	$M_A$	$M_B$	$D_A$	$D_B$	Log
1							<t, begin=""></t,>
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, 16="" 8,="" a,=""></t,>
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, 16="" 8,="" b,=""></t,>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
	Crash!						

# 基于Undo Logging的故障恢复

邹兆年 (CS@HIT)

Example (基于Undo Logging的故障恢复)										
Step	Action	t	$M_A$	$M_B$	$D_{\mathcal{A}}$	$D_B$	Log			
1							<t, begin=""></t,>			
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, 16="" 8,="" a,=""></t,>			
5	READ(B, t)	8	16	8	8	8				
	Crash!									

990

38 / 69

2020年春

# Write-Ahead Logging (WAL) Redo Logging

邹兆年 (CS@HIT)

12章: 故障恢复

2020年春

39 / 69

# 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

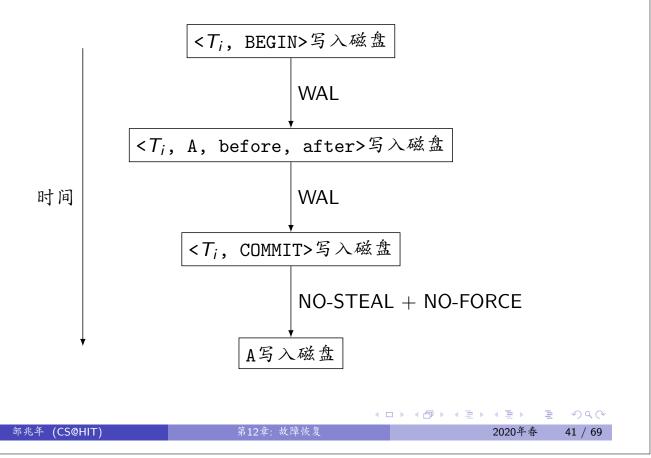
□ > <□ > <□ > <□ > <□ ≥ </li>○

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 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

Examp	le (Redo Loggii	ng)					
Step	Action	t	$M_A$	$M_B$	$D_{\mathcal{A}}$	$D_B$	Log
1							<t, begin=""></t,>
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, 16="" 8,="" a,=""></t,>
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, 16="" 8,="" b,=""></t,>
8	COMMIT						<t, commit=""></t,>
9	FLUSH LOG						
10	OUTPUT(A)	16	16	16	16	8	
11	OUTPUT(B)	16	16	16	16	16	

990

第12章: 故障恢复 邹兆年 (CS@HIT)

2020年春

# 基于Redo Logging的故障恢复

#### 已提交事务(Committed Txn): 全部redo

● NO-FORCE ⇒ 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 ⇒ 不完整事务所做的任何修改都未写入磁盘

邹兆年 (CS@HIT)

512章·故障恢复

2020年春

43 / 69

# 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2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>

**◆□▶ ◆□▶ ◆■▶ ◆■ り**९@

# 基于Redo Logging的故障恢复

Examp	le (基于Redo L	.oggi	ng的古	文障恢	复)		
Step	Action	t	$M_A$	$M_B$	$D_{\mathcal{A}}$	$D_B$	Log
1							<t, begin=""></t,>
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, 16="" 8,="" a,=""></t,>
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, 16="" 8,="" b,=""></t,>
8	COMMIT						<t, commit=""></t,>
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

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

# 基于Redo Logging的故障恢复

Examp	le (基于Redo L	.oggi	ng的t	文障恢	复)		
Step	Action	t	$M_A$	$M_B$	$D_{\mathcal{A}}$	$D_B$	Log
1							<t, begin=""></t,>
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, 16="" 8,="" a,=""></t,>
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, 16="" 8,="" b,=""></t,>
	Crash!						

 4 □ ト 4 □ ト 4 豆 ト 豆 り Q ○

 郵兆年 (CS@HIT)
 第12章: 故障恢复
 2020年春 45 / 69

Write-Ahead Logging (WAL) Redo+Undo Logging

4 D > 4 B > 4 B > B 9 Q @

邹兆年 (CS@HIT) 第12章: 故障恢复

2020年春 46 / 69

# Redo+Undo Logging

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

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

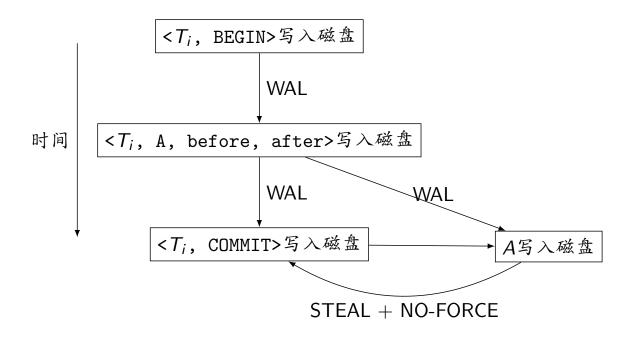
邹兆年 (CS@HIT)

812章: 故障恢复

2020年春

47 / 69

# 基于Redo+Undo Logging的事务正常执行时的行为



邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春 4

# 基于Redo+Undo Logging的事务正常执行时的行为

Examp	le (Redo+Undo	Log	ging)				
Step	Action	t	$M_A$	$M_B$	$D_{\mathcal{A}}$	$D_B$	Log
1							<t, begin=""></t,>
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, 16="" 8,="" a,=""></t,>
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, 16="" 8,="" b,=""></t,>
8	OUTPUT(A)	16	16	16	16	8	
9	COMMIT						<t, commit=""></t,>
10	FLUSH LOG						
11	OUTPUT(B)	16	16	16	16	16	

 邹兆年 (CS@HIT)
 第12章: 故障恢复
 2020年春
 49 / 69

# 基于Redo+Undo Logging的故障恢复

#### 已提交事务(Committed Txn): 全部redo

● NO-FORCE ⇒ 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 ⇒ 不完整事务所做的一部分修改可能已经写入磁盘

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春 50 / 69

# 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

邹兆年 (CS@HIT)

12章: 故障恢复

2020年春

51 / 69

# 基于Redo+Undo Logging的故障恢复

#### Example (基于Redo+Undo Logging的故障恢复) Step $M_A$ $M_B$ $D_A$ $D_{B}$ Action Log 1 <T, BEGIN> 2 READ(A, t) 8 8 8 8 3 t := t \* 28 8 8 16 <T, A, 8, 16> WRITE(A, t) 16 16 4 8 8 5 READ(B, t) 8 16 8 8 8 8 6 t := t \* 2**16** 16 8 8 7 WRITE(B, t) 16 16 16 8 8 <T, B, 8, 16> 8 OUTPUT(A) 16 16 16 16 8 9 <T, COMMIT> 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=""></t,>
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, 16="" 8,="" a,=""></t,>
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, 16="" 8,="" b,=""></t,>
8	OUTPUT(A)	16	16	16	16	8	
	Crash!						

邹兆年 (CS@HIT)

[12章: 故障恢复

2020年春

52 / 69

# 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

**◆□▶ ◆□▶ ◆■▶ ◆■▶ ■ か**990

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 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

邹兆年 (CS@HIT)

[12章: 故障恢复

2020年春

54 / 69

Checkpoints

邹兆年 (CS@HIT)

第10音, 按隨恢复

2020年春

## 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>

 $\langle T_2, \text{ BEGIN} \rangle$ 

 $\langle T_2, B, 10, 20 \rangle$ 

 $\langle T_2, C, 15, 25 \rangle$ 

 $< T_2$ , COMMIT>

 $< T_3$ , BEGIN>

如果使用Undo Logging,则扫描到这里即可

 $\langle T_1, D, 20, 30 \rangle$ 

 $< T_3$ , E, 25, 35>

 $< T_1$ , COMMIT>

 $< T_3$ , F, 30, 40>

◆□ ト ◆□ ト ◆ ≧ ト ◆ ≧ ・ りゅう

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

56 / 69

# 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<sub>1</sub>,T<sub>2</sub>,...,T<sub>n</sub>全部提交后,才能结束检查点

4 D > 4 B > 4 E > 4 E > 9 Q @

邹兆年 (CS@HIT)

512章·故障恢复

2020年春

58 / 69

# 涉及检查点的故障恢复

缓冲池策略: STEAL + NO-FORCE

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<Ti, BEGIN>开始扫描

◆□▶ ◆□▶ ◆■▶ ◆■▶ ■ 990

邹兆年 (CS@HIT)

512章·故障恢复

2020年春

60 / 69

# 证明 |

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,>	
< T, COMMIT>	
<begin <math="" checkpoint="">(T_1, T_2, \ldots, T_n)&gt;</begin>	$ T \not\in \{T_1, T_2, \ldots, T_n\} $
<end checkpoint=""></end>	T所做的修改已全部写 到磁盘,无需redo
	到磁盘,无需redo

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 证明 ||

Log	Fact
<t, begin=""></t,>	
<pre><begin <math="" checkpoint="">(T_1, T_2, \ldots, T_n)&gt;</begin></pre>	$T \in \{T_1, T_2, \dots, T_n\}$
··· <pre> </pre> <pre> <pre> </pre> <pre> <pre> </pre> <pre> <pre> </pre> <pre> <pre> </pre> <pre> </pre> <pre> </pre> <pre>  <pre> <td></td></pre></pre></pre></pre></pre></pre>	
<pre><end checkpoint=""></end></pre>	T所做的修改已全部写 到磁盘,无需redo

4 D > 4 B > 4 B > B 9 Q @

邹兆年 (CS@HIT)

512章: 故障恢复

2020年春

62 / 69

# 证明 |||

Log	Fact
<t, begin=""></t,>	
  <	$T \in \{T_1, T_2, \ldots, T_n\}$
<pre> <end checkpoint=""> </end></pre>	
<t, commit=""></t,>	T所做的修改未必全部 写到磁盘,必须redo

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 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

**◆□▶ ◆□▶ ◆≣▶ ■ り**900

邹兆年 (CS@HIT)

812章: 故障恢复

2020年春

54 / 69

## 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=""></t,>	
  <	$T \in \{T_1, T_2, \ldots, T_n\}$
<pre><end checkpoint=""></end></pre>	T所做的部分修改可能 已写到磁盘,必须undo

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 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<sub>3, B, 10, 20&gt;</t<sub>	
<pre><begin <math="" checkpoint="">(T_2, T_3)&gt;</begin></pre>	从这里开始redo
<t<sub>2, C, 15, 25&gt;</t<sub>	
<t<sub>3, D, 20, 30&gt;</t<sub>	
<end checkpoint=""></end>	
$< T_2$ , COMMIT>	

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

66 / 69

# 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<sub>3, B, 10, 20&gt;</t<sub>	
<pre><begin <math="" checkpoint="">(T_2, T_3)&gt;</begin></pre>	从这里开始redo
<t<sub>2, C, 15, 25&gt;</t<sub>	C ← 25
<t<sub>3, D, 20, 30&gt;</t<sub>	
<end checkpoint=""></end>	
$< T_2$ , COMMIT>	

**◆□▶ ◆□▶ ◆■▶ ◆■▶ ■ か**990

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 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<sub>3, B, 10, 20&gt;</t<sub>	
<begin <math="" checkpoint="">(T_2, T_3)&gt;</begin>	从这里开始redo
<t<sub>2, C, 15, 25&gt;</t<sub>	C ← 25
<t<sub>3, D, 20, 30&gt;</t<sub>	T <sub>3</sub> 无需redo
<end checkpoint=""></end>	
$< T_2$ , COMMIT>	

4□ > 4□ > 4 = > 4 = > =

邹兆年 (CS@HIT)

2020年春

# 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<sub>3, B, 10, 20&gt;</t<sub>	
<pre><begin <math="" checkpoint="">(T_2, T_3)&gt;</begin></pre>	从这里开始redo
<t<sub>2, C, 15, 25&gt;</t<sub>	C ← 25
<t<sub>3, D, 20, 30&gt;</t<sub>	T <sub>3</sub> 无需redo
<end checkpoint=""></end>	
$< T_2$ , COMMIT>	T <sub>2</sub> redo完毕

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 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<sub>1, A, 5, 15&gt;</t<sub>	
$< T_2$ , BEGIN>	
$< T_1$ , COMMIT>	
$< T_3$ , BEGIN>	
<t<sub>3, B, 10, 20&gt;</t<sub>	
<pre><begin <math="" checkpoint="">(T_2, T_3)&gt;</begin></pre>	
<t<sub>2, C, 15, 25&gt;</t<sub>	
<t<sub>3, D, 20, 30&gt;</t<sub>	
<end checkpoint=""></end>	
<t<sub>2, COMMIT&gt;</t<sub>	T <sub>2</sub> 无需undo

邹兆年 (CS@HIT)

12章: 故障恢复

2020年春

67 / 69

# 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<sub>3, B, 10, 20&gt;</t<sub>	
<begin <math="" checkpoint="">(T_2, T_3)&gt;</begin>	
<t<sub>2, C, 15, 25&gt;</t<sub>	
<t<sub>3, D, 20, 30&gt;</t<sub>	D ← 20
<end checkpoint=""></end>	
$< T_2$ , COMMIT>	$T_2$ 无需undo

◆□ → ◆□ → ◆ ≥ → ◆ ≥ → ○ へ()

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春

# 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<sub>3, B, 10, 20&gt;</t<sub>	
<pre><begin <math="" checkpoint="">(T_2, T_3)&gt;</begin></pre>	
<t<sub>2, C, 15, 25&gt;</t<sub>	T <sub>2</sub> 无需undo
<t<sub>3, D, 20, 30&gt;</t<sub>	D ← 20
<end checkpoint=""></end>	
<t<sub>2, COMMIT&gt;</t<sub>	T <sub>2</sub> 无需undo

邹兆年 (CS@HIT)

512章: 故障恢复

2020年春

67 / 69

# 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<sub>3, B, 10, 20&gt;</t<sub>	B ← 10
<begin <math="" checkpoint="">(T_2, T_3)&gt;</begin>	
<t<sub>2, C, 15, 25&gt;</t<sub>	$T_2$ 无需undo
<t<sub>3, D, 20, 30&gt;</t<sub>	D ← 20
<pre><end checkpoint=""></end></pre>	
$< T_2$ , COMMIT>	$T_2$ 无需undo

**◆□ ▶ ◆□ ▶ ◆■ ▶ ◆■ ● り**९()

# 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 <sub>3</sub> undo完毕,写 <t<sub>3,ABORT&gt;</t<sub>
<t<sub>3, B, 10, 20&gt;</t<sub>	B ← 10
<pre><begin <math="" checkpoint="">(T_2, T_3)&gt;</begin></pre>	
<t<sub>2, C, 15, 25&gt;</t<sub>	T <sub>2</sub> 无需undo
<t<sub>3, D, 20, 30&gt;</t<sub>	D ← 20
<end checkpoint=""></end>	
$< T_2$ , COMMIT>	$T_2$ 无需undo

邹兆年 (CS@HIT)

12章: 故障恢复

2020年春

67 / 69

# 总结

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

感谢同学们的热情,让我面对屏幕也能愉快地讲课 感谢同学们的配合,让我的首次网络授课顺利完成 感谢同学们的提问,让我对数据库的认识更加深入 感谢同学们的指正,让我的教学能力得到显著提升 祝同学们身体健康, 学业有成, 前程似锦

> 邹兆年 2020年春季疫情期间 于哈尔滨家中

> > COMMIT;

◆ロ ト ◆ 個 ト ◆ 重 ト ● ■ り へ で

邹兆年 (CS@HIT)

第12章: 故障恢复

2020年春 69 / 69