第十九讲恢复系统

(第十九章)

关继宏教授

电子邮件:jhguan@tongji.edu.cn

计算机科学与技术系

同济大学

课程大纲

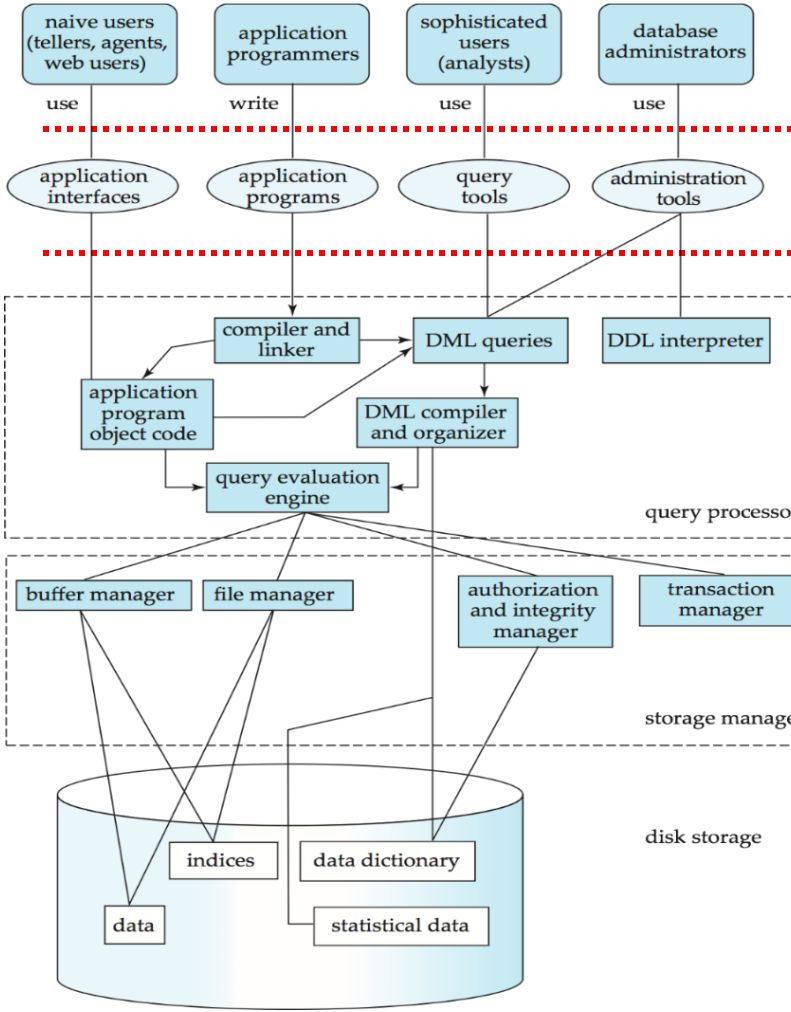
- **第0部分:概述**
  - Ch1:介绍
- **Part 1关系数据库**
  - Ch2:关系模型(数据模型，关系代数)
  - Ch3&4: SQL(结构化查询语言)
  - Ch5:高级SQL
- **第二部分数据库设计**
  - Ch6:基于E-R模型的数据库设计
  - Ch7:关系型数据库设计
- **第三部分:应用程序设计与开发**
  - Ch8:复杂数据类型
  - Ch9:应用开发
- **Part 4大数据分析**
  - Ch10:大数据
  - Ch11:数据分析
- **第5部分:数据存储和索引**
  - Ch12:物理存储系统
  - Ch13:数据存储结构
  - Ch14:索引
- **第6部分:查询处理与优化**
  - Ch15:查询处理
  - Ch16:查询优化

第7部分:事务管理

- Ch17:交易
- Ch18:并发控制
- **Ch19:恢复系统**

- **第8部分:并行和分布式数据库**
  - Ch20:数据库系统架构
  - Ch21-23:并行和分布式存储，查询处理和事务处理
- **第9部分**
  - **DB平台:OceanBase、MongoDB、Neo4J**

数据库系统结构



用户

应用程序/工具

数据库管理系统

数据库

大纲

<s:1>故障分类

- 存储
- 恢复和原子性
- 恢复算法
- 缓冲区管理

故障分类

- **事务失败(英文)**
  - 逻辑错误，例如非法输入
  - 系统错误，如死锁
- **系统崩溃**
  - 电源故障或其他硬件和软件故障导致系统崩溃
- **磁盘故障(cu盘)**
  - 磁头崩溃或类似的磁盘故障会破坏全部或部分磁盘存储

恢复算法

- 在失败的情况下确保数据库一致性和事务原子性的技术
- **恢复算法有两部分**
  - **在正常事务处理过程中采取的动作**
    - 保证有足够的信息用于故障恢复
  - **失败后采取的行动**
    - 恢复数据库到某个一致性状态

大纲

- 故障分类

☞**存储**

- 恢复和原子性
- 恢复算法
- 缓冲区管理

存储结构

- **易失性存储器**
  - 系统崩溃后不能保存
  - 例如，主存储器，缓存存储器
- **非易失性存储器**
  - 在系统崩溃时仍然存在
  - 例如，磁盘、磁带、闪存
- **稳定存储()**
  - 一种神话般的存储形式，可以在所有故障中幸存下来
  - 近似于在不同的非易失性介质上保持多个副本

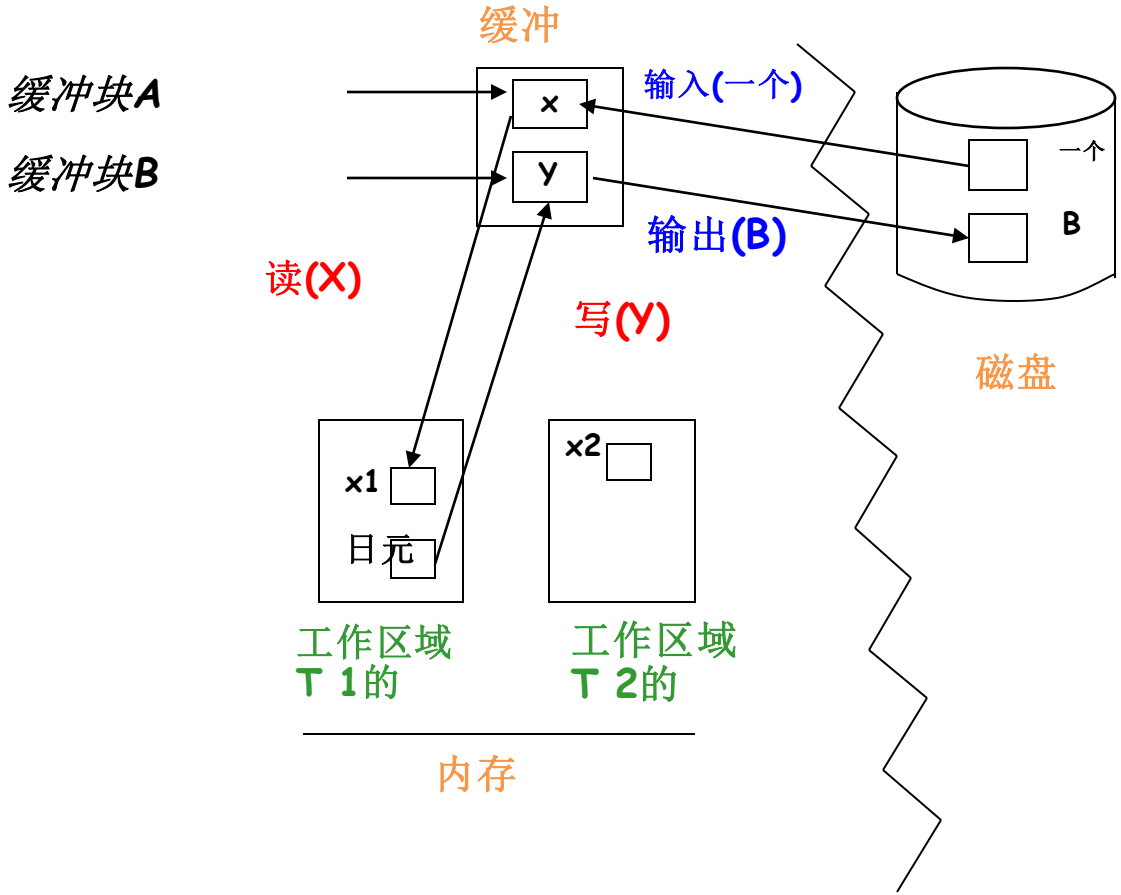
数据访问

- **Physical blocks** (物理块)
  - the blocks residing on the disk
- **Buffer blocks** (缓冲块)
  - the blocks residing temporarily in main memory
- Block movements between disk and main memory
  - **input(B): physical block -> memory**
  - **output(B): buffer block -> disk**
- Each transaction T_i has its **private work-area** (私有工作区)
  - T_i's local copy of a data item X is called x_i

数据访问(续)

- 事务在系统缓冲块和它的私有工作区域之间传输数据项
  - **读(X)**
  - **写(X)**
- 交易
  - 第一次访问X时执行read(X)
  - 所有后续的访问都是对本地副本的访问
  - 在最后一次访问之后，事务执行write(X)
- **output(B X)不需要立即跟随write(X)**
  - 系统可以在它认为合适的时候执行输出操作

数据访问示例


缓冲
缓冲块A
缓冲块B
x
y
输入(一个)
输出(B)
读(X)
写(Y)
一个
B
磁盘
x1
日元
x2
工作区域
T 1的
工作区域
T 2的
内存

大纲

- 故障分类
- 存储

恢复和原子性

- 恢复算法
- 缓冲区管理

恢复和原子性

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Consider transaction T_i that transfers \$50 from account **A** to account **B**
  - Several output operations may be required for T_i to output **A** and **B**
  - A failure may occur after one of these modifications have been made but before all of them are made
- **To ensure atomicity despite failures**, we **first output information describing the modifications to stable storage without modifying the database itself**
- **Two approaches**
  - **log-based recovery (基于日志的恢复)**
  - **shadow-paging (影子页)**

基于日志恢复

- A log is kept on stable storage
  - The log is a sequence of log records
- When transaction T_i starts, it registers itself by writing a $<T_i$ start$>$ log record
  - Before T_i executes write(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the old value and V_2 is the new value
  - When T_i finishes it's last statement, the log record $<T_i$, commit$>$ is written
- Two approaches using logs
  - Deferred database modification (延迟数据库修改)
  - Immediate database modification (即刻数据库修改)

延迟数据库修改

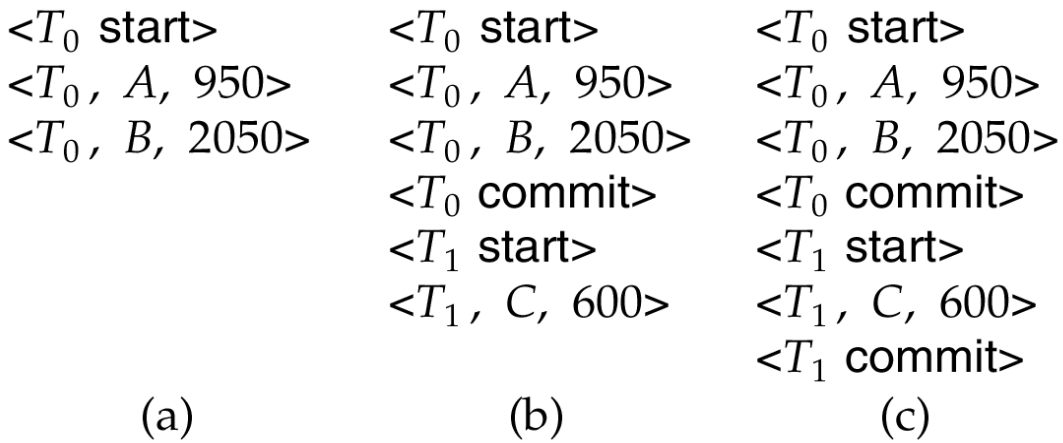
- Record all modifications to the log, but **defer all the writes to after partial commit**
  - Transaction starts by writing **<T_i start>** record to log
  - A **write(X)** operation results in a log record **<T_i, X, V>** being written, where V is the **new value**.
  - The write is not performed on X at this time, but is **deferred**
  - When T_i partially commits, **< T_i commit>** is written to the log
  - Finally, the log records are read and used to **actually execute the previously deferred writes**

延迟数据库修改(续)

- **Recovery after a crash**
  - a transaction needs to be **redone iff both <T_i start> and < T_i commit>** are there in the log
  - **Redo(T_i)** sets the value of all data items updated by the transaction **to the new values**
- **Example:**
  - T_0 executes before T_1, and initial: **A=1000, B=2000, C=700**

T_0:		T_1:	
	read (A)		read (C)
	A: = A - 50		C:= C- 100
	write (A)		write (C)
	read (B)		
	B:= B + 50		
	write (B)		

延期数据库修改(Cont.)



- 上述每种情况下的恢复操作是:
  - (a)不需要采取重做操作
  - (b)必须执行重做(t0)
  - (c)重做(t0)后必须执行重做(t1)

立即数据库修改(Immediate Database Modification)

- **允许未提交事务的数据库更新**
  - **更新日志记录必须在写入数据库项之前写入**
  - 更新块的输出可以发生在事务提交之前或之后的任何时间
  - 块的输出顺序可以不同于它们被写入的顺序

即时数据库修改示例

$\langle T_i, X, V_1, V_2 \rangle$,
where V_1 is the **old value**,
and V_2 is the **new value**

日志写输出

<T 0 start>
< t 0, a, 1000,950 >
< t 0, b, 2000, 2050>

A = 950
B = 2050

<T 0 commit>
<T 1 start>
< t 1, c, 700, 600>

C = 600

B, B, c

<T 1 commit>

B一个

注:B X表示包含X的块

即时数据库修改(Cont.)

- Recovery procedure has two operations
  - **undo(T_i)**: restore the value of all data items updated by transaction T_i **to the old values**
  - **redo(T_i)**: set the value of all data items updated by transaction T_i **to the new values**
- When recovering after failure
  - Transaction T_i needs to be **undone** if **the log contains the record <T_i start>, but does not contain <T_i commit>**
  - Transaction T_i needs to be **redone** if **the log contains both the record <T_i start> and <T_i commit>**
- **Undo** operations are performed **first**, then redo operations

数据库立即修改恢复样例

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

- 上述每种情况下的恢复操作是:
  - **(a)撤消(T0)**
  - **(b)撤销(T1)和重做(T0)**
  - **(c)重做(T0)和重做(T1)**

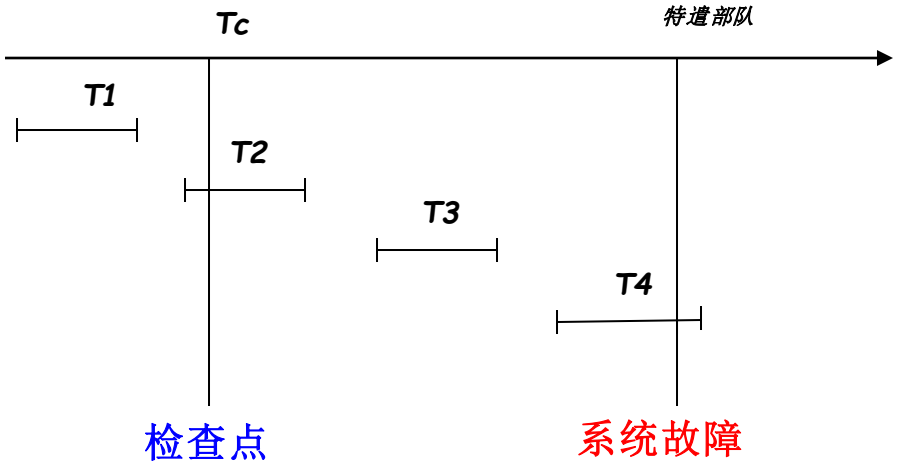
检查点()

- **恢复过程中的问题**
  - 搜索整个日志非常耗时
  - 可能会不必要地重做已经将更新输出到数据库的事务
- **通过定期设置检查点来恢复过程**
  - **将当前驻留在主存中的所有日志记录输出到稳定的存储器**
  - **输出所有修改过的缓冲块到磁盘**
  - 将日志记录<checkpoint>写入稳定存储

检查点(续)。

- **During recovery**
  - Scan **backwards** from the end of log to find the most recent **<checkpoint>** record
  - Continue scanning **backwards** till a record **<T_i,start>** is found. We assume that all transactions are executed **serially**.
    - Need only consider the part of log following above start record
    - For all transactions (starting from T_i or later) with **no <T_i,commit>**, execute **undo(T_i)**.
    - Scanning **forward** in the log, for all transactions starting from T_i or later with a **<T_i,commit>**, execute **redo(T_i)**.

检查点示例



- T1可以忽略(根据检查点更新已经输出到磁盘)
- T2和t3重做
- T4撤销

大纲

- 故障分类
- 存储
- 恢复和原子性

恢复算法

- 缓冲区管理

使用并发事务进行恢复

- 我们修改了基于日志的恢复方案，以允许多个事务并发执行
  - 所有事务共享一个磁盘缓冲区和一个日志
  - 一个缓冲块可以有一个或多个事务更新的数据项
- 我们假设并发控制使用严格的两阶段锁定
- 日志记录按照前面描述的方式完成
  - 不同事务的日志记录可能会散布在日志中
- 必须更改检查点技术和恢复时采取的操作

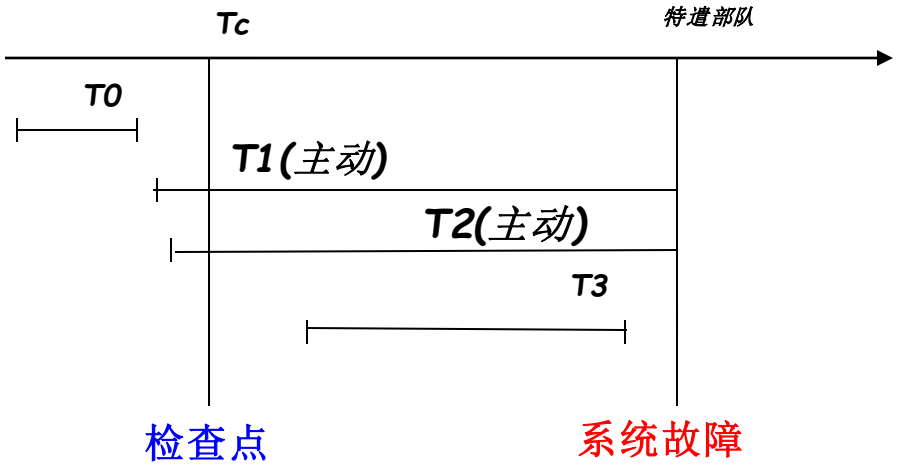
并发事务的恢复(续)

- Checkpoints are performed as before, except that the checkpoint log record is the form **<checkpoint L>**
  - **L** is a list of transactions **active** at the time of the **checkpoint**
  - We assume no update is in progress while the checkpoint is carried out
- When the system **recovers from a crash**
  - Initialize **undo-list** and **redo-list** to empty
  - Scan the log backwards **until a <checkpoint L>** record is found:
    - if the record is **<T_i commit>**, add T_i to **redo-list**
    - if the record is **<T_i start>** and T_i is not in **redo-list**, add T_i to **undo-list**
    - for every T_i in **L**, if T_i is not in **redo-list**, add T_i to **undo-list**

恢复示例

- 请按照下面日志中的恢复算法步骤进行操作

<T 0 * T >
< t 0, a, 0, 10>
<T 0 commit >
<T 1 start >
< t 1, b, 0, 10>
<T 2 start >
< t 2, c, 0, 10>
< t 2, c, 10, 20>
<检查点{t1, t2}>
< t3 start >
< t 3, a, 10, 20>
< t 3, d, 0, 10>
<T 3 commit >



向后扫描日志:撤销列表中撤销T1、T2
正向扫描日志:重做列表中重做T3

使用并发事务进行恢复(续)

- 恢复的工作原理如下
  - 从日志末尾向后扫描日志
    - 扫描期间，对撤消列表中属于事务的每条日志记录执行撤消
  - 找到最近的<检查点L>记录
  - 从<checkpoint L>记录开始向前扫描日志，直到日志的末尾
    - 在扫描期间，对重做列表中属于事务的每个日志记录执行重做

大纲

- 故障分类
- 存储
- 恢复和原子性
- 恢复算法

<s:1>缓冲区管理

日志记录缓冲

- **日志记录缓冲**
  - 日志记录缓冲在主内存中，而不是直接输出到稳定的存储中
  - **当缓冲区中的一块日志记录被填满，或者执行日志强制操作时，日志记录才会输出到稳定存储器**
  - **执行Log force是通过强制事务的所有日志记录(包括提交记录)到稳定存储来提交事务**
  - 使用单个输出操作可以输出多个日志记录，从而降低I/O成本

日志记录缓冲(Cont.)

- **Write-ahead logging (WAL) rule for buffering log records**
  - Log records are output to stable storage in the order in which they are created
  - Transaction T_i enters the commit state only when the log record $<T_i$ **commit>** has been output to stable storage
  - **Before** a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage

数据库缓冲

- 数据库维护数据块的内存缓冲区
  - 当需要一个新的块时，如果缓冲区已满，则应从缓冲区中删除一个现有的块
  - 如果选择删除的块已经更新，则必须将其输出到磁盘
- 当一个块被输出到磁盘时，它不应该正在进行更新，这是如下所保证的。
  - 在写入数据项之前，事务在包含该数据项的块上获得排他锁
  - 一旦写操作完成，锁就会被释放。
    - 这种持续时间较短的锁被称为锁存闩
  - 在一个块被输出到磁盘之前，系统在该块上获得一个独占锁存器
    - 确保块上没有正在进行的更新

缓冲区管理(续)

- 数据库缓冲区也可以实现
  - 在为数据库保留的真正的主存区域中，或者
  - 在虚拟内存中
- 在保留主存中实现缓冲区有缺点
  - 内存在数据库缓冲区和应用程序之间预先分区，限制了灵活性
  - 需求可能会改变，尽管操作系统在任何时候都最清楚内存应该如何划分，但它无法改变内存的分区

缓冲区管理(续)

- 数据库缓冲区通常在虚拟内存中实现，尽管存在一些缺点
  - 当操作系统需要驱逐()一个被修改过的页面时，这个页面会被写入到磁盘上交换空间
  - 当DB决定将缓冲页写入磁盘时，缓冲页可能在交换空间中，并且可能需要从磁盘上的交换空间中读取并输出到磁盘上的数据库，从而导致额外的I/O
    - 称为双分页()问题
  - 理想情况下，当交换数据库缓冲区页时，操作系统应该将控制权传递给数据库，从而将页输出到数据库而不是交换空间(确保首先输出日志记录)。
    - 这样就可以避免双分页，但是普通的操作系统并不支持这样的功能。

非易失性存储器丢失导致的故障

- 类似于检查点的技术，用于处理非易失性存储器的丢失
  - 定期将数据库的全部内容转储到稳定的存储中
  - 在转储过程中，没有事务是活动的;必须发生一个类似于检查点的过程
    - 将当前驻留在主存中的所有日志记录输出到稳定存储器
    - 输出所有缓冲块到磁盘上
    - 将数据库的内容复制到稳定的存储中
    - 输出一条记录<dump>以登录稳定存储
  - 从磁盘故障中恢复
    - 从最近的转储恢复数据库。
    - 查阅日志并重做转储后提交的所有事务
- 是否可以扩展到允许事务在转储期间处于活动状态，称为模糊转储或在线转储

第19讲结束