TONGJI UNIVERSITY

ADMIS Lab

先进数据与机器智能系统实验室

# 恢复系统
# Recovery System

李文根/Wengen Li

Email: lwengen@tongji.edu.cn

先进数据与机器智能系统实验室（ADMIS）

https://admis.tongji.edu.cn/main.htm

同济大学 TONGJI UNIVERSITY 电子与信息工程学院 College of Electronics and Information Engineering 计算机科学与技术系 Department of Computer Science and Technology

## 课程概要

- **Part 0: Overview**
  - Ch1: Introduction
- **Part 1 Relational Databases**
  - Ch2: Relational model
  - Ch3: Introduction to SQL
  - Ch4: Intermediate SQL
  - Ch5: Advanced SQL
- **Part 2 Database Design**
  - Ch6: Database design based on E-R model
  - Ch7: Relational database design
- **Part 3 Application Design & Development**
  - Ch8: Complex data types
  - Ch9: Application development
- **Part 4 Big data analytics**
  - Ch10: Big data
  - Ch11: Data analytics
- **Part 5 Data Storage & Indexing**
  - Ch12: Physical storage system
  - Ch13: Data storage structure
  - Ch14: Indexing
- **Part 6 Query Processing & Optimization**
  - Ch15: Query processing
  - Ch16: Query optimization
- **Part 7 Transaction Management**
  - Ch17: Transactions
  - Ch18: Concurrency control
  - **Ch19: Recovery system**
- **Part 8 Parallel & Distributed Database**
  - Ch20: Database system architecture
  - Ch21-23: Parallel & distributed storage, query processing & transaction processing
- **Part 9**
  - DB Platform: **OceanBase**, MongoDB, Neo4J

## 目录

## 故障分类

- **Transaction failure (事务故障)**
  - Logical errors, e.g., illegal inputs
  - System errors, e.g., dead locks
- **System crash (系统崩溃)**
  - A power failure, or other hardware and software failures cause the system to crash
- **Disk failure (磁盘故障)**
  - A head crash or similar disk failures destroy all or part of disk storage

## 恢复算法

- Techniques to ensure database consistency and transaction atomicity **despite failures**

- Recovery algorithms have two parts
  - Actions taken during normal transaction processing
    - 保证有足够的信息用于故障恢复
  - Actions taken after a failure
    - 恢复数据库到某个一致性状态

## 目录

## 存储器分类

- **Volatile storage (易失性存储器)**
  - does not survive system crashes
  - e.g., main memory, cache memory
- **Non-volatile storage (非易失性存储器)**
  - survives system crashes
  - e.g., disk, tape, flash memory
- **Stable storage (稳定存储器)**
  - a mythical form of storage that survives all failures
  - achieved by maintaining multiple copies on distinct non-volatile media

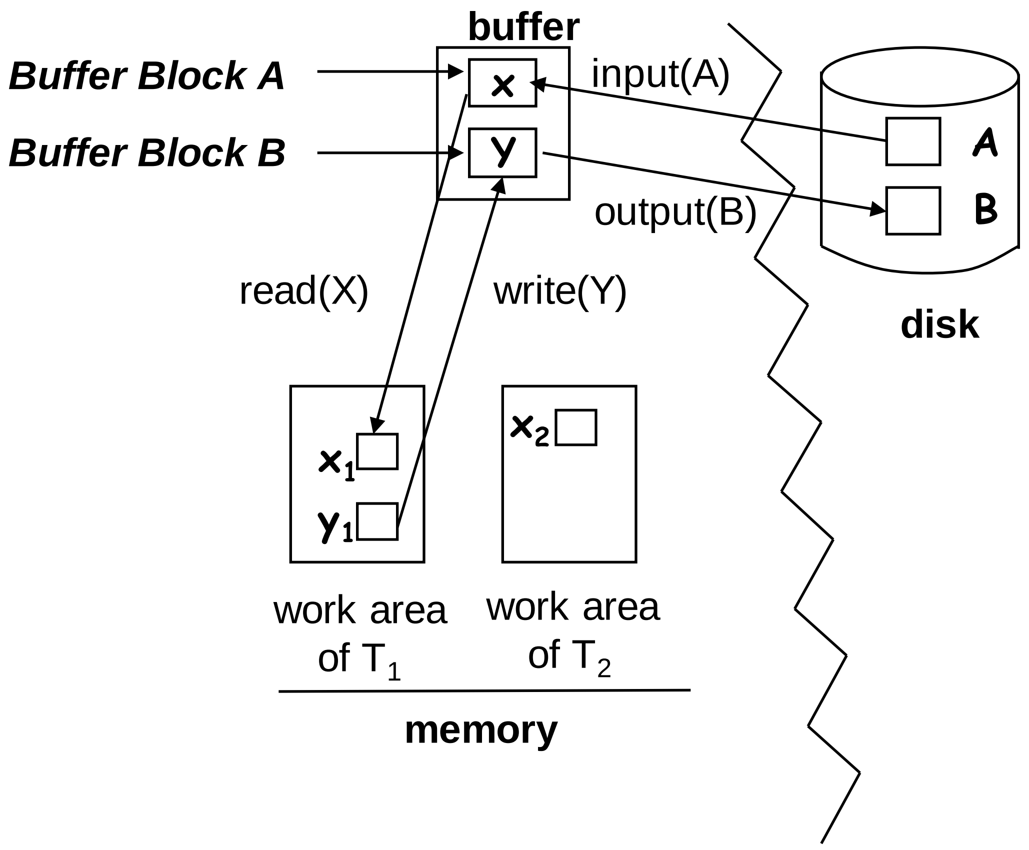
## 数据访问

- **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 (buffer)
  - **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 denoted by $x_i$

## 数据访问（续）

- Transaction transfers data items between system buffer blocks and its private work-area using
  - read(X)
  - write(X)
- **Transactions**
  - Perform read(X) while accessing X for the first time
  - All subsequent accesses are to the local copy
  - After the last access, transaction executes write(X)
- **output($B_X$)** does not need to immediately follow write(X)
  - System can perform the output operation when it deems fit

## 数据访问（续）


buffer
Buffer Block A
Buffer Block B
x
y
input(A)
output(B)
A
B
disk
read(X)
write(Y)
$x_1$
$y_1$
$x_2$
work area of $T_1$
work area of $T_2$
memory

## 目录

## 恢复与原子性

- 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
- **Two approaches**
  - **Log-based recovery (基于日志的恢复)**
  - Shadow-paging (影子页)

## 基于日志的恢复

- 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 the 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 < $T_i$, $X$, $V_1$, $V_2$> being written, where $V_2$ 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

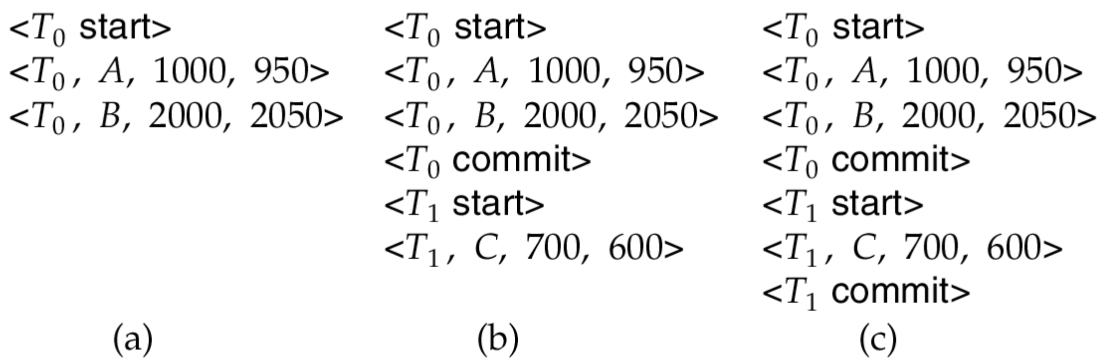
## 延迟数据库修改（续）

- **Recovery after a crash**
  - A transaction needs to be redone iff both <$T_i$ start> and < $T_i$ commit> are 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

```
T0: read (A)          T1 : read (C)
A: = A - 50                C:= C- 100
write (A)                  write (C)
read (B)
B:=  B + 50
write (B)
```

## 延迟数据库修改（续）

- Recovery actions in each failure case below:
  - (a) no redo actions need to be taken
  - (b) redo($T_0$) must be performed
  - (c) redo($T_0$) must be performed followed by redo($T_1$)



| | | |
|---|---|---|
| <$T_0$ 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> |
| (a) | (b) | (c) |

## 即刻数据库修改

- Allows database updates of an uncommitted transaction
  - Update log records must be written **before** database items are written
  - The output of updated blocks can take place at any time before or after transaction commit
  - Order in which blocks are output can be different from the order in which they are written

## 即刻数据库修改（续）

| **Log** | **Write** | **Output** |
|---|---|---|
| <$T_0$ 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 | |
| | | $B_B$, $B_C$ |
| <$T_1$ commit> | | |
| | | $B_A$ |

**Note: $B_X$ denotes the block containing $X$**

# 即刻数据库修改（续）

- Recovery procedure has two operations
  - undo($T_i$): restore the values of all the data items updated by transaction $T_i$ to the old values
  - redo($T_i$): set the values 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

## 即刻数据库修改（续）

<$T_0$ start>
<$T_0$, $A$, 1000, 950>
<$T_0$, $B$, 2000, 2050>

(a)

<$T_0$ start>
<$T_0$, $A$, 1000, 950>
<$T_0$, $B$, 2000, 2050>
<$T_0$ commit>
<$T_1$ start>
<$T_1$, $C$, 700, 600>

(b)

<$T_0$ start>
<$T_0$, $A$, 1000, 950>
<$T_0$, $B$, 2000, 2050>
<$T_0$ commit>
<$T_1$ start>
<$T_1$, $C$, 700, 600>
<$T_1$ commit>

(c)

- Recovery actions in each case above are:
  - (a) undo($T_0$)
  - (b) undo($T_1$) and redo($T_0$)
  - (c) redo($T_0$) and redo($T_1$)

## 检查点（Checkpoint）

TONGJI UNIVERSITY

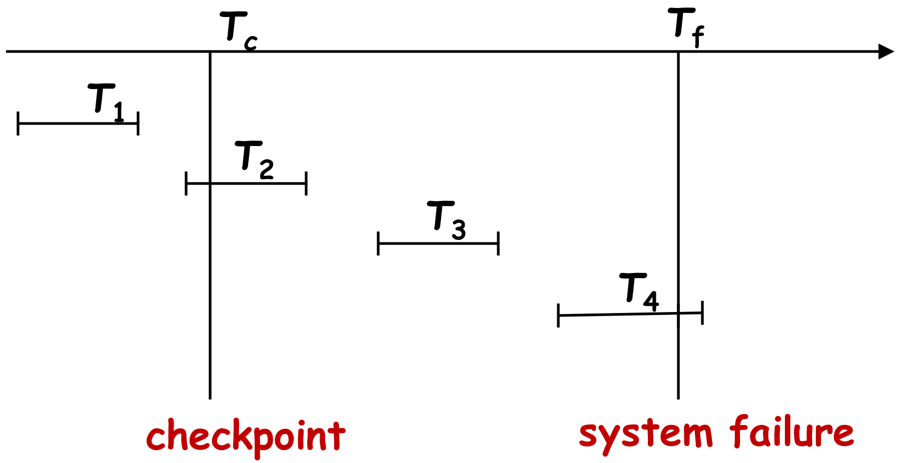
- **Problems in the recovery procedure**
  - Searching the entire log is time-consuming
  - Might unnecessarily redo transactions which have already output their updates to the database

- **Recovery procedure sets checkpoints periodically**
  - Output all the **log records** currently residing in main memory to stable storage
  - Output all the **modified buffer blocks** to the disk
  - Write a log record <checkpoint> to stable storage

## 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 <$T_j$ commit>, execute undo($T_j$).
    - Scanning forward in the log, for all transactions starting from $T_i$ or later with a <$T_j$ commit>, execute redo($T_j$).

# 检查点（续）



- $T_1$ can be ignored (updates have been already output to disk according to the checkpoint)
- redo $T_2$ and $T_3$
- undo $T_4$

## 目录

## 并发事务的恢复

- Modify the log-based recovery scheme to allow multiple transactions to execute concurrently
  - All transactions share **a single disk buffer** and **a single log**
  - A buffer block can have data items updated by one or more transactions
- Assume that concurrency control uses strict two-phase locking
- Logging is done as described earlier
  - Log records of different transactions may be interspersed（散布）in the log
- The checkpointing technique and actions taken on recovery have to be changed

## 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 active transactions at the time of the checkpoint
  - 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

## 并发事务的恢复（续）

- Go through the steps of the recovery algorithm on the following log

<$T_0$ **start**>
<$T_0$, $A$, 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>
<checkpoint {$T_1$, $T_2$}>
<$T_3$ **start**>
<$T_3$, $A$, 10, 20>
<$T_3$, $D$, 0, 10>
<$T_3$ **commit**>

## 并发事务的恢复（续）

- **Recovery**
  - Scan log backwards from the end of the log
    - During the scan, perform undo for each log record that belongs to a transaction in undo-list
  - Locate the most recent <checkpoint L> record
  - Scan log forwards from the <checkpoint L> record till the end of the log
    - During the scan, perform redo for each log record that belongs to a transaction on redo-list

## 目录

## 日志记录的缓存

- **Log record buffering**
  - Log records are **buffered in main memory**, instead of being output directly to stable storage
  - Log records are output to stable storage when a block of log records in the buffer is full, or a log force operation is executed
  - **Log force** is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage
  - Several log records can be output using a single output operation, thus reducing the I/O cost

## 日志记录的缓存（续）

- **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

## 数据库缓存

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, an existing block should be removed from buffer if the buffer is full
  - If the block chosen for removal has been updated, it must be output to disk

- No update should be in progress on a block when it is output to disk, which is ensured as follows
  - Before a block is output to disk, the system acquires an **exclusive latch (闩锁)** on the block
    - Ensures no update can be in progress on the block

## 缓存管理

- **Database buffer can be implemented either**
  - In an area of real **main-memory** reserved for the database, or in **virtual memory**

- Implementing buffer in reserved main-memory has drawbacks
  - Memory is partitioned before-hand between database buffer and applications, limiting the flexibility of main-memory
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory

## 缓存管理（续）

- Database buffers are generally implemented in **virtual memory** in spite of some drawbacks
  - When OS needs to evict（逐出）a page that has been modified, the page is written to swap space on disk
  - When DB decides to write buffer page to disk, buffer page may be in swap space (交换区), and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/Os
    - Known as dual paging（双分页）problem
  - Ideally when swapping out a database buffer page, operating system should pass control to database, which in turn outputs the page to database instead of to swap space
    - Dual paging can thus be avoided, but common operating systems do not support such functionality

## 非易失存储器故障损失

- **Technique similar to checkpointing is used to deal with loss of non-volatile storage**
  - Periodically dump the entire content of the database to stable storage
  - No transaction may be active during the dump procedure, and a procedure similar to checkpointing must take place
    - output all log records currently residing in main memory onto stable storage
    - output all buffer blocks onto the disk
    - copy the contents of the database to stable storage
    - output a record <dump> to log on stable storage
  - To recover from disk failure
    - restore database from the most recent dump
    - consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump, known as fuzzy dump or online dump