# 数 据 库 原 理

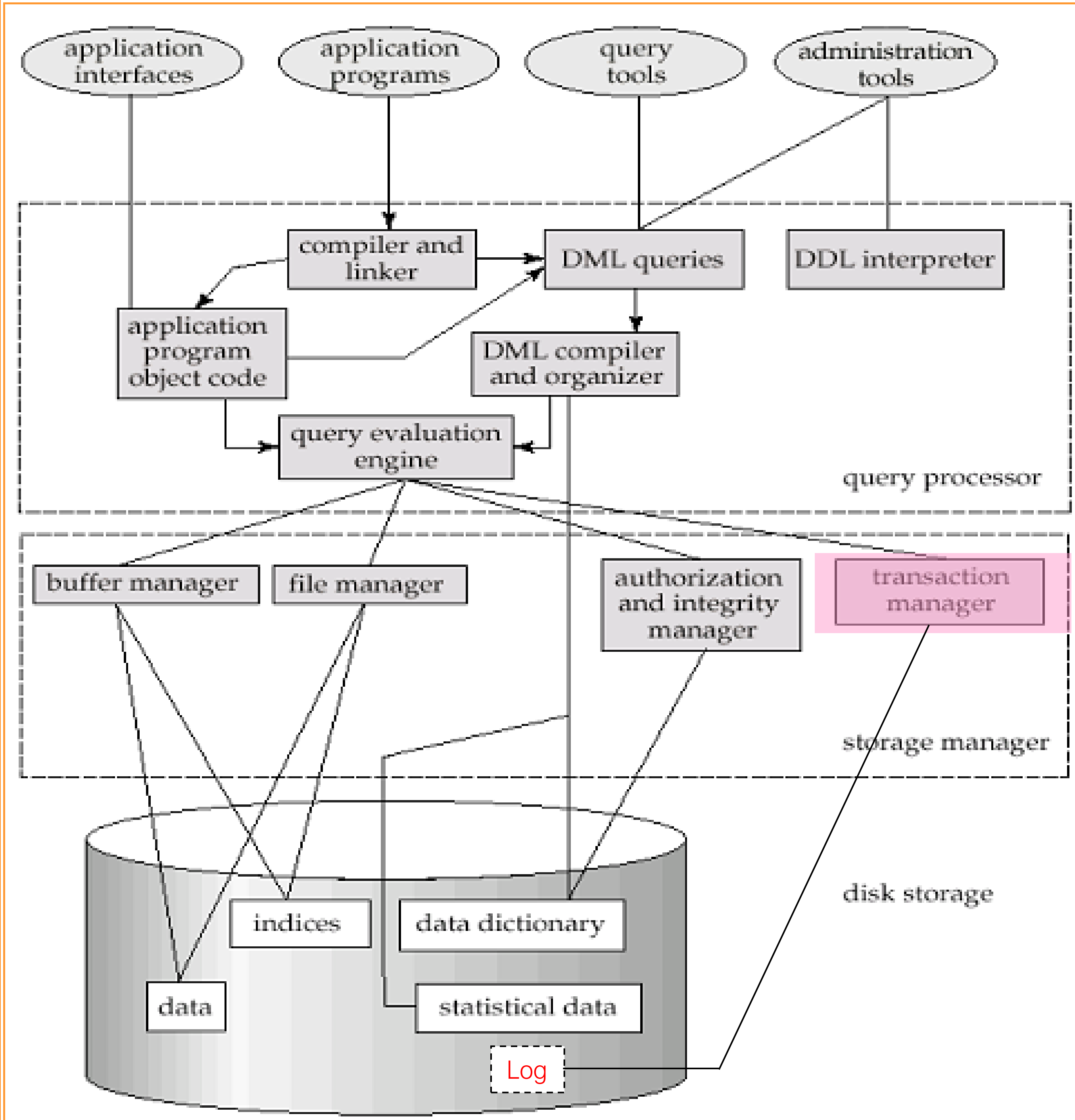
# Chp. 19
# Recovery System

王 朝 坤
清 华 大 学 软 件 学 院
2025年/春

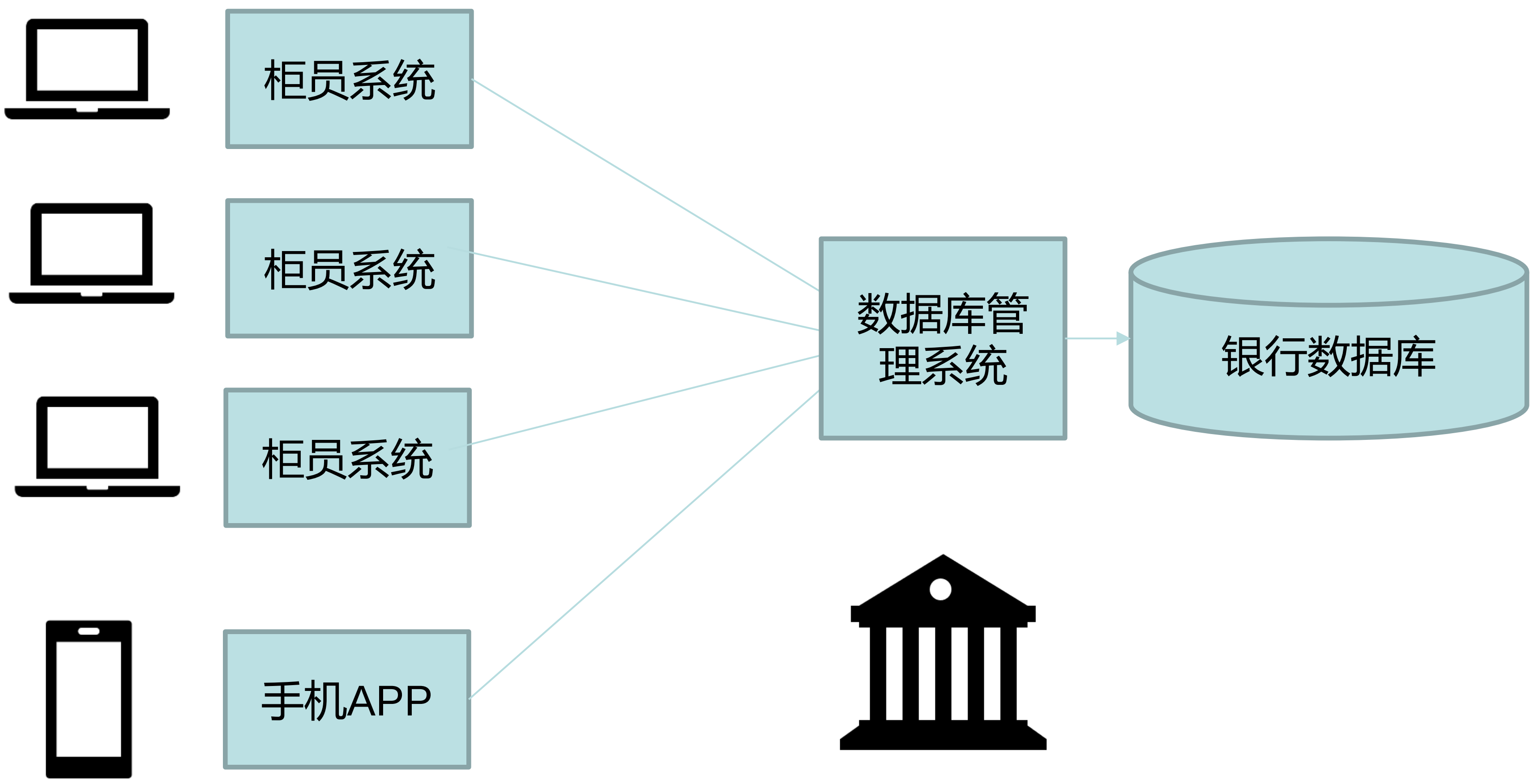
# 本节课的内容

| 内容 | 核心知识点 | 对应章节 |
|---|---|---|
| DBMS理论 | 关系数据模型（以及关系演算)、SQL语言 | CHP. 1、2、3 |
| DBMS设计 | 存贮、索引、查询、优化、事务、并发、**恢复** | CHP. 12、13、14、15、16、17、18、**19** |
| DBMS实现 | 大作业 | 小班辅导 |
| DBMS应用 | 实体-联系图、关系范式、DDL、JDBC | CHP. 4、5、6、7 |

# DBMS Architecture



# 银行数据库系统



# 问题1：脏读

| 事务1 | 事务2 |
| --- | --- |
| **1.Start Trans I-L R-U** | |
| | **1.Start Trans** |
| | **2.write**(*A:50*) |
| 2.Read（A：50） | |
| | 3.rollback |
| 3.Read（A：？） | |

# 问题2：不可重复读

| 事务1 | 事务2 |
| --- | --- |
| **1.Start Trans I-L R-C** | |
| **2.read**(*A:100*) | |
| | **1.Start Trans** |
| | **2.write**(*A:50*) |
| | 3.commit |
| **3.read**(*A:50*) | |

# 问题3：幻读

| 事务1 | 事务2 |
| --- | --- |
| **1.Start Trans I-L R-R** | |
| **2.select**(*B*<*100*) | |
| -(01,’BJ’, 90) | **1.Start Trans** |
| | **2.Insert** |
| | **-**(*03,’BJ’,50*) |
| | **3.Commit** |
| **3.select**(*B*<*100*) | |
| -(01,’BJ’, 90) | |
| **-**(*03,’BJ’,50*) | |

# 事务隔离级别（§17.8)

| 隔离级别 \ 问题 | | 脏读 | 不可重复读 | 幻读 |
|---|---|---|---|---|
| 1 | **Read Uncommitted**<br>读未提交 | V | V | V |
| 2 | **Read Committed**<br>读提交 | X | V | V |
| 3 | **Repeatable Read**<br>可重复读 | X | X | V |
| 4 | **Serializable**<br>可串行化 | X | X | X |

# 系统出现故障怎么办?

转账事务

**1.read**(*A:100*)

*2.A* := *A* – 50

**3.write**(*A:50*)

**4.read**(*B:50*)

*5.B* := *B* + 50

**6.write**(*B:100*)

# Main Contents

**1.Failure Classification(19.1)**

2.Storage (19.2)

3.Recovery and Atomicity(19.3)

4.Log-Based Recovery Algorithm(19.4)

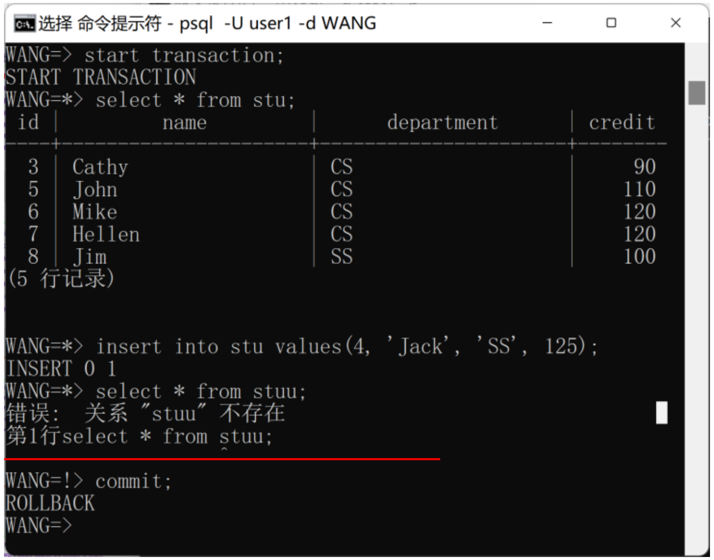
5.Buffer Management(19.5)

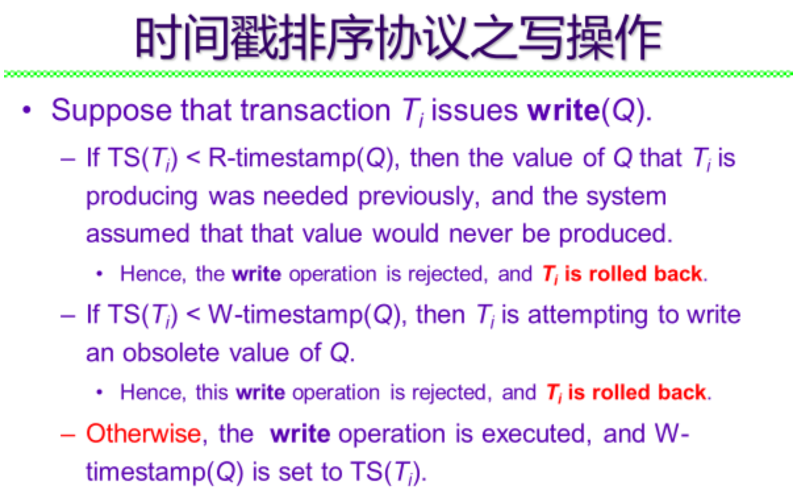
6.Failure with Loss of Nonvolatile Storage(19.6)

7.Remote Backup Systems(19.7)

# 系统故障分类1 - 事务故障

- **Logical errors**: transaction cannot complete due to some internal error condition, such as bad input, data not found, overflow, or resource limit exceeded.
- **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock).

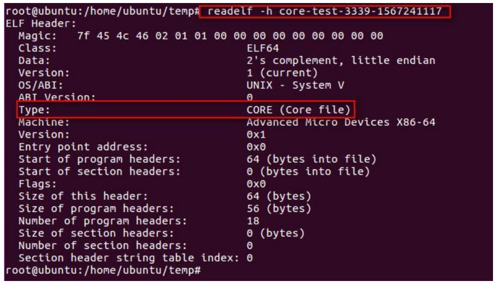


# 系统故障分类2 - 系统崩溃

- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash /* 硬盘完好
    - Database systems have numerous integrity checks to prevent corruption of disk data

图片来至百度




Unix Core Dump了


Windows 蓝屏了

# 系统故障分类3 – 磁盘故障

- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage.
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures


硬盘短路起火

硬盘磁头损坏

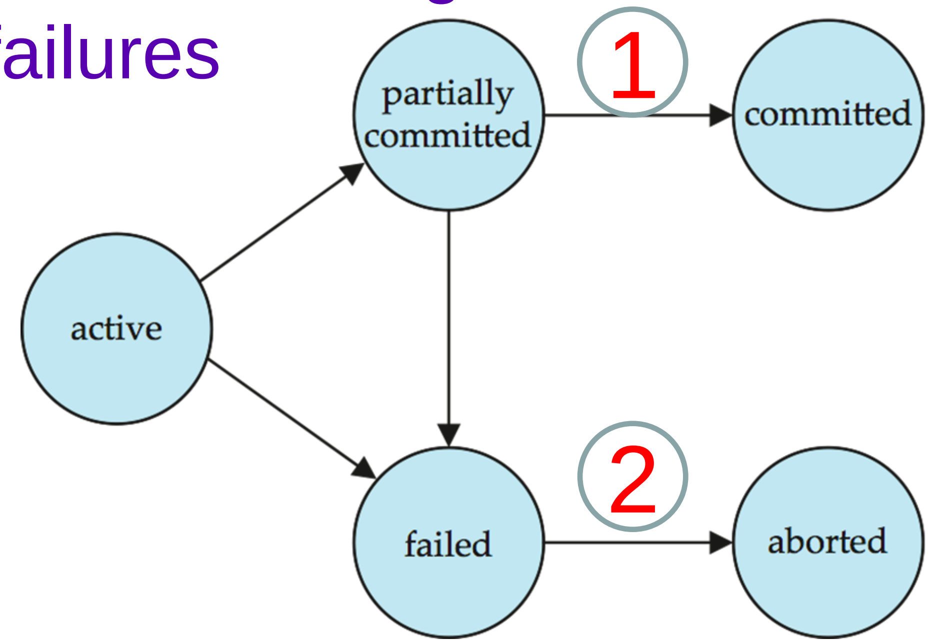
图片来至百度

# 三类系统故障带来的问题

- 原子性问题
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - ***Modifying the database partially*** without ensuring that the transaction will commit may leave the database in an *inconsistent state*
- 持久性问题
  - ***Not modifying*** the database may *result in* ***lost updates*** if failure occurs just after transaction commits.

# 解决方案：故障恢复机制/算法

① **Actions** taken during normal transaction processing to ensure enough information exists to recover from failures




② **Actions** taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

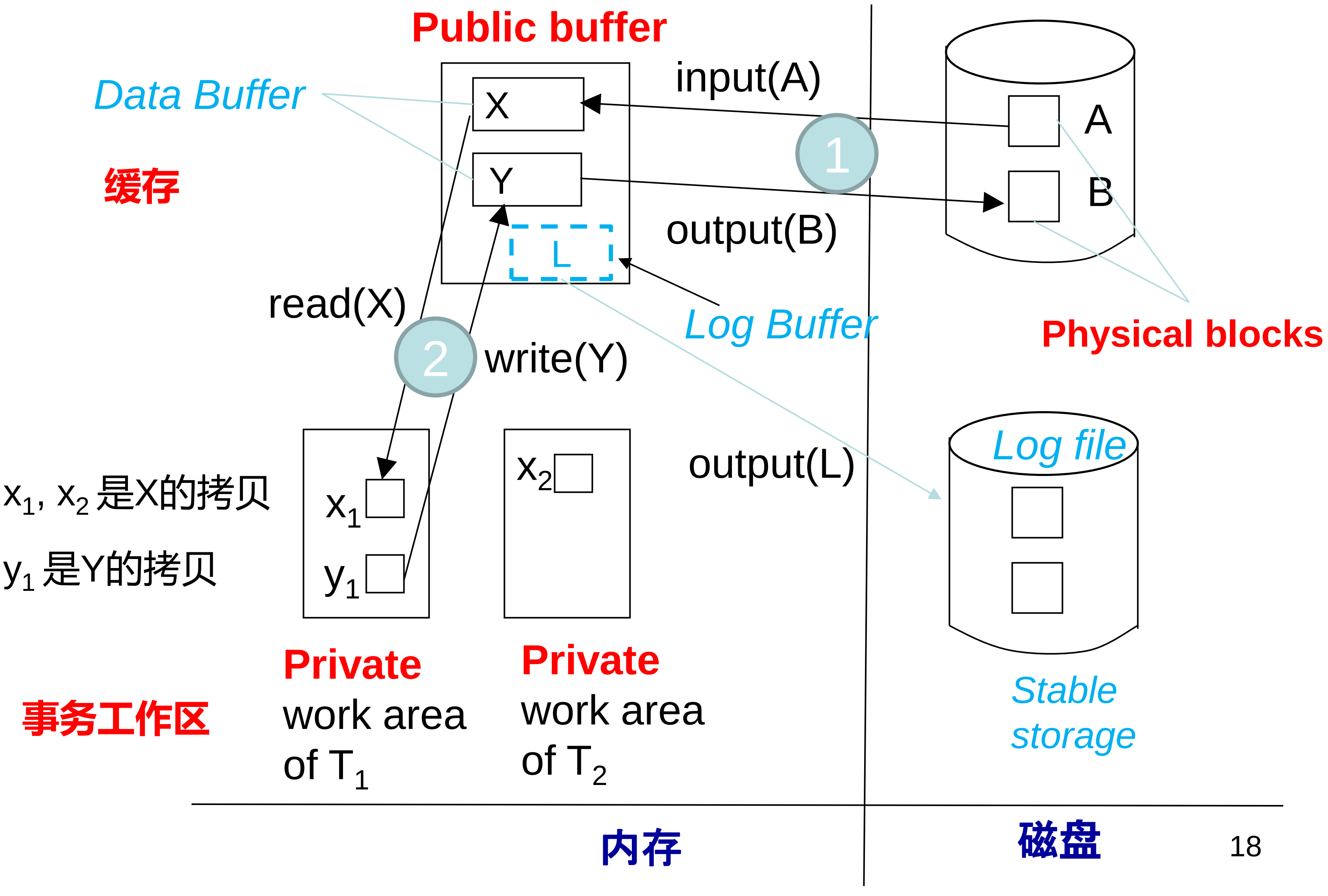
# Main Contents

1. Failure Classification
2. **Storage (19.2)**
3. Recovery and Atomicity
4. Log-Based Recovery Algorithm
5. Buffer Management
6. Failure with Loss of Nonvolatile Storage
7. Remote Backup Systems

# 可靠存储 – 存放日志

- **易失存储:volatile storage**
  - loses contents when power is switched off
- **非易失存储:non-volatile storage**
  - Contents persist even when power is switched off.
  - Includes secondary and tertiary storage, as well as battery-backed up main-memory.
- **可靠存储:Stable storage**
  - Information residing in stable storage is *never* lost
  - To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with independent failure modes
  - **Store the log file to keep track of the old values of any data on which a transaction performs a write**

# 事务访问数据的基本流程



# 磁盘与缓存之间的数据传输

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.

- **input(*B*)** transfers **the physical block B** to main memory.
- **output(*B*)** transfers **the buffer block B** to the disk, and replaces the appropriate physical block there.

- **For simplicity, we assume that each data item fits in a single block.**

# 事务工作区与缓存之间的数据传输

- **read(*X*)** assigns the value of data item *X* to the local variable $x_i$.
- **write(X)** assigns the value of local variable $x_i$ to data item *X* in the buffer block.
  - **Note: output**($B_X$) , **also called as *force-output* of buffer B**, need not immediately follow **write**(*X*). System can perform the **output** operation when **it deems fit**.

- Transactions
  - Must perform **read**(*X*) before accessing *X* for the first time (subsequent reads can be from local copy)
  - **write**(*X*) can be executed **at any time** before the transaction commits

# Main Contents

1.Failure Classification

2.Storage Structure

**3.Recovery and Atomicity (19.3)**

4.Log-Based Recovery Algorithm

5.Buffer Management

6.Failure with Loss of Nonvolatile Storage

7.Remote Backup Systems

# 基于日志的恢复机制

- To ensure atomicity despite failures, we first **output** information describing the modifications to ***stable storage*** without modifying the database itself.

- **log-based recovery mechanisms :** The log is a sequence of **log records**, and maintains a record of **update** activities on the database.

# 日志生成机制/时机

- A **log** is kept on **stable storage**. **日志写在可靠存储器中**

1. When transaction $T_i$ starts, it registers itself by writing a

   <$T_i$ **start**> log record

2. *Before* $T_i$ executes **write**($X$), a log record /* 只管写，不管读

   <$T_i$, $X$, $V_1$, $V_2$>

   is written, where $V_1$ is the value of $X$ before the write (the **old value**), and $V_2$ is the value to be written to $X$ (the **new value**).

3. When $T_i$ finishes its last statement, the log record <$T_i$ **commit**> is written.

# 数据库修改的两种策略

- Two approaches using logs
  - Immediate database modification /* 立即修改
  - Deferred database modification /* 延迟修改

# 立即修改

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, **before the transaction commits**
  - Update log record must be written *before* database item is written
    - We assume that the log record is output directly to stable storage
    - (Will see later that how to postpone log record output to some extent)
  - Output of updated blocks to disk 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.

# 延迟修改

- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy

# 事务提交的标志

- **A transaction is said to have committed when its *commit log record* is output to stable storage**
  - all previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

# 串行事务/立即修改的日志示例

| **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**> | | |
| | | $B_A$ |

$B_C$ output before $T_1$ commits

$B_A$ output after $T_0$ commits

- Note: $B_X$ denotes block containing $X$.

Memory

Disk

# 并发事务的日志格式

- With concurrent transactions, **all transactions share a single disk buffer and a single log**
  - A buffer block can have data items updated by one or more transactions /*缓存块已被多个事务修改
- Log records of different transactions may be **interspersed** in the log

# 调度假设 – 杜绝写丢失

- *if a transaction $T_i$ has modified an item, no other transaction can modify the same item until $T_i$ has committed or aborted*
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort? ***that is not recoverable***.

- Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking) /* 严格两阶段加锁协议来保证

# 事务的撤销与重做

- ***Undo of Transactions /* 自后向前做***
  - **Undo** of a log record <$T_i$, $X$, $V_1$, $V_2$> writes the **old** value $V_1$ to $X$
  - **undo**($T_i$) restores the value of all data items updated by $T_i$ to their old values, going backwards from the last log record for $T_i$
    - each time a data item X is restored to **its old value** V a special log record <$T_i$ , $X$, $V$> is written out
    - when undo of a transaction is complete, a log record <$T_i$ **abort**> is written out.
- ***Redo of Transactions /* 自前向后做***
  - **Redo** of a log record <$T_i$, $X$, $V_1$, $V_2$> writes the **new** value $V_2$ to $X$
  - **redo**($T_i$) sets the value of all data items updated by $T_i$ to the new values, going forward from the first log record for $T_i$
    - No logging is done in this case

# 基于日志的事务恢复逻辑

- Transaction $T_i$ needs to be undone if the log
  - contains the record <$T_i$ **start**>,
  - but does not contain either the record <$T_i$ **commit**> *or* <$T_i$ **abort**>.
- Transaction $T_i$ needs to be redone if the log
  - contains the records <$T_i$ **start**>
  - and contains the record <$T_i$ **commit**> *or* <$T_i$ **abort**>

# 对于撤销日志的重做

- Note that If transaction $T_i$ was undone earlier and the <$T_i$ **abort**> record written to the log, and then a failure occurs, on recovery from failure, $T_i$ is redone
  - **such a redo redoes all the original actions *including the steps that restored old values***
  - Known as **repeating history**
  - **Seems wasteful, but simplifies recovery greatly**

# 立即更新模式的事务恢复示例

Below we show the log as it appears at three instances of time.

| (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> |
| <T0, B, 2000> | <$T_0$ commit> | <$T_0$ commit> |
| <T0, A, 1000 > | <$T_1$ start> | <$T_1$ start> |
| <T0, Abort> | <$T_1$, $C$, 700, 600> | <$T_1$, $C$, 700, 600> |
| | <T1, C, 700 > | <$T_1$ commit> |
| | <T1, abort> | |

(a) undo ($T_0$):

B is restored to 2000 and A to 1000, and log records <$T_0$, B, 2000>, <$T_0$, A, 1000>, <$T_0$, **abort**> are written out

(b) redo ($T_0$) and undo ($T_1$):

*A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <$T_1$, C, 700>, <$T_1$, **abort**> are written out.

(c) redo ($T_0$) and redo ($T_1$):

A and B are set to 950 and 2050 respectively. Then *C* is set to 600

# 撤销与重做的代价

- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time
  2. we might unnecessarily redo transactions which have already output their updates to the database.

# 检查点机制

- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all **log** records currently residing in main memory onto stable storage.
  2. Output all **modified** buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is **a list of all transactions active** at the time of checkpoint. ***/* 记下当前活动的事务***
  - All updates are stopped while doing checkpointing

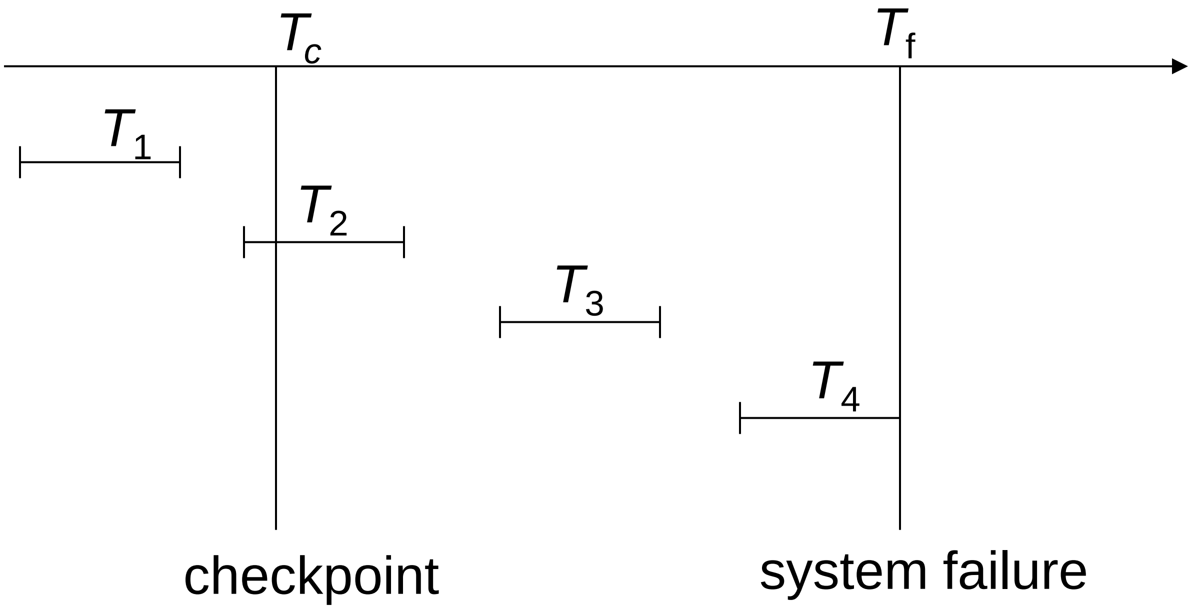
# 检查点的恢复机制

- During recovery we need to consider only the most recent transaction $T_i$ that started before the checkpoint, and transactions that started after $T_i$.
  - Scan backwards from end of log to find the most recent <checkpoint L> record ***/* 先找到检查点***
  - Only transactions that are in *L* **or** started after the checkpoint need to be redone or undone
  - Transactions that committed **or** aborted before the checkpoint already have all their updates output to stable storage.

# 日志记录的删除

- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <$T_i$ **start**> is found for every transaction $T_i$ in $L$.
  - Parts of log prior to earliest <$T_i$ **start**> record above are not needed for recovery, **and can be erased whenever desired**. ***/* 避免日志文件无限膨胀***

# 检查点示例




- $T_1$ can be ignored (updates already output to disk due to checkpoint)
- $T_2$ and $T_3$ redone.
- $T_4$ undone

# Main Contents

1.Failure Classification

2.Storage Structure

3.Recovery and Atomicity

**4.Log-Based Recovery Algorithm (19.4)**

5.Buffer Management

6.Failure with Loss of Nonvolatile Storage

7.Remote Backup Systems

# 故障类型1：事务回滚操作

- **Logging** (during normal operation):
  - <$T_i$ **start**> at transaction start
  - <$T_i$, $X_j$, $V_1$, $V_2$> for each update, and
  - <$T_i$ **commit**> at transaction end
- **Transaction rollback**
  - Let $T_i$ be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of $T_i$ of the form <$T_i$, $X_j$, $V_1$, $V_2$>
    - perform the undo by writing $V_1$ to $X_j$,
    - write a log record <$T_i$ , $X_j$, $V_1$>
      - such log records are called ***compensation log records***
  - Once the record <$T_i$ **start**> is found stop the scan and write the log record <$T_i$ **abort**>

# 故障类型2：系统恢复的两个阶段

- **Redo phase**:  replay updates of **all** transactions, whether they committed, aborted, or *are incomplete*
- **Undo phase**: undo all incomplete transactions

# 重做阶段-Redo phase

- Find last <**checkpoint** *L*> record, and set **undo-list** to *L*.
- Scan forward from above <**checkpoint** *L*> record
  - Whenever a record <$T_i$, $X_j$, $V_1$, $V_2$>, or a redo-only log record of the form <$T_i$, $X_j$, $V_2$>, is found, redo it by writing $V_2$ to $X_j$
  - Whenever a log record <$T_i$ **start**> is found, add $T_i$ to undo-list
  - Whenever a log record <$T_i$ **commit**> *or* <$T_i$ **abort**> is found, remove $T_i$ from undo-list

# 撤销阶段 - Undo phase

Scan log backwards from end

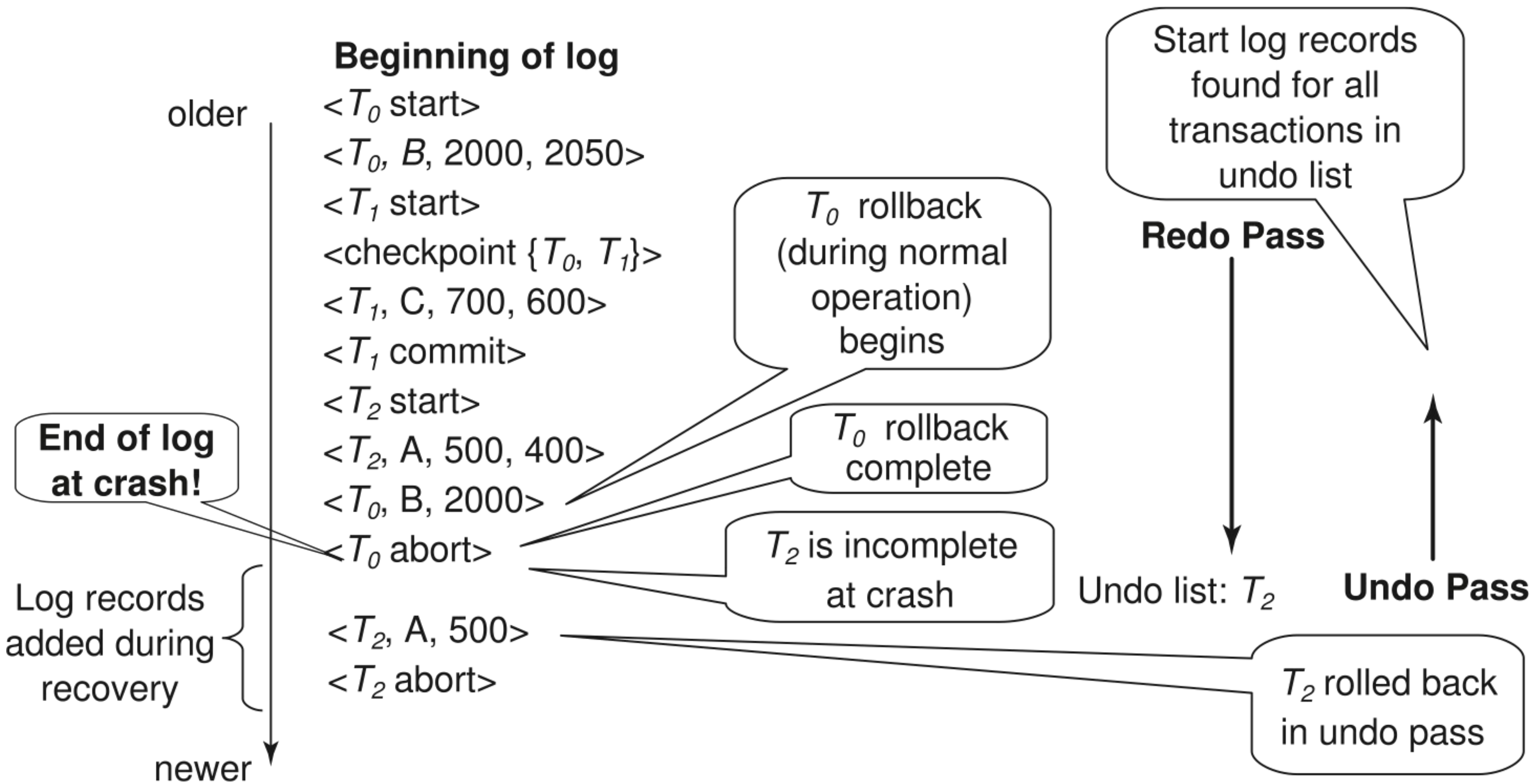
– Whenever a log record <$T_i$, $X_j$, $V_1$, $V_2$> is found where $T_i$ is in ***undo-list*** perform same actions as for transaction rollback:
  - perform undo by writing $V_1$ to $X_j$.
  - write a log record <$T_i$ , $X_j$, $V_1$>

– Whenever a log record <$T_i$ **start**> is found where $T_i$ is in undo-list,
  - Write a log record <$T_i$ **abort**>
  - Remove $T_i$ from undo-list

– Stop when undo-list is empty

i.e. <$T_i$ **start**> has been found for every transaction in undo-list

# 带有检查点的恢复过程



Beginning of log
older
<$T_0$ start>
<$T_0$, $B$, 2000, 2050>
<$T_1$ start>
<checkpoint {$T_0$, $T_1$}>
<$T_1$, C, 700, 600>
<$T_1$ commit>
<$T_2$ start>
<$T_2$, A, 500, 400>
<$T_0$, B, 2000>
<$T_0$ abort>
<$T_2$, A, 500>
<$T_2$ abort>
newer
End of log at crash!
Log records added during recovery
$T_0$ rollback (during normal operation) begins
$T_0$ rollback complete
$T_2$ is incomplete at crash
$T_2$ rolled back in undo pass
Start log records found for all transactions in undo list
Redo Pass
Undo list: $T_2$
Undo Pass

# Main Contents

1.Failure Classification

2.Storage Structure

3.Recovery and Atomicity

4.Log-Based Recovery Algorithm

**5.Buffer Management (19.5)**

6.Failure with Loss of Nonvolatile Storage

7.Remote Backup Systems

# 日志记录缓存

- **Log record buffering**: log records are buffered in main memory, instead of 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 thus be output using a single output operation, reducing the I/O cost.

# 先写日志规则 (WAL)

- The rules below must be followed if log records are buffered:
  1. Log records are output to stable storage in the order in which they are created.
  2. Transaction $T_i$ enters the commit state only when the log record <$T_i$ **commit**> has been output to stable storage.
  3. Before a block of data in main memory is output to the disk, all log records pertaining to data in that block must have been output to stable storage.
     - This rule is called the **write-ahead logging** or **WAL** rule
       – Strictly speaking WAL only requires undo information to be output
       – 新的数据块会覆盖旧的值

# 数据库缓存（参见13.5节）

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

# 非强制策略与窃取策略

- **force policy**: requires updated blocks to be written at commit
- **no-force policy**: updated blocks need not be written to disk when transaction commits
- **steal policy** : blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

# 数据库缓存输出到硬盘

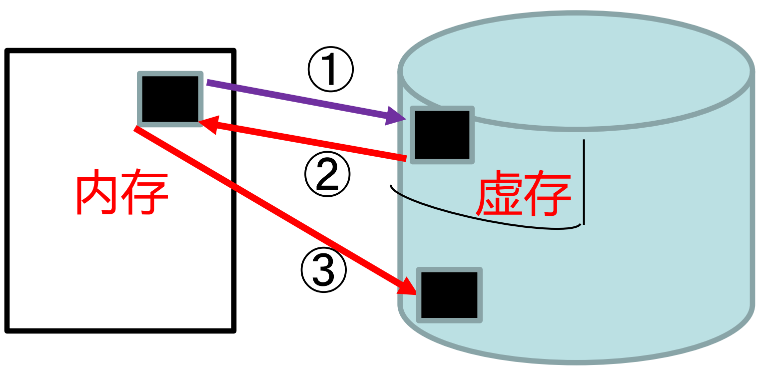
- No updates should be in progress on a block when it is **output** to disk.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     1. Ensures no **write** can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch on the block

# 操作系统对缓存管理的支持

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or in virtual memory
- reserving database buffer in main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and non-database applications, limiting flexibility.
  - 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.

# 虚拟内存的问题

- When operating system needs to evict a page that has been modified, the page is ***written to swap space on disk*** ①
- When database decides to write buffer page to disk, buffer page may be in swap space, and have to be ***read from swap space on disk*** ② and ***output to the database on disk*** ③, resulting in extra I/O!
  - Known as **dual paging** problem



# 解决方案

- Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
  ① **Output the page to database** instead of to swap space (making sure to output log records first), if it is modified
  ② Release the page from the buffer, for the OS to use
- Dual paging can thus be avoided, but common operating systems do not support such functionality

# Main Contents

1. Failure Classification
2. Storage Structure
3. Recovery and Atomicity
4. Log-Based Recovery Algorithm
5. Buffer Management
6. **Failure with Loss of Nonvolatile Storage(19.6)**
7. Remote Backup Systems

# 故障类型3 - 磁盘故障

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage


硬盘短路起火


硬盘磁头损坏

图片来至百度

# 数据转储

- Periodically **dump** the entire content of the database to stable storage
  - Database files + Log files
- No transaction may be active during the dump procedure; 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.

# 数据恢复

- **To recover from disk failure**
  - restore database from  most recent dump.
  - Consult the log and redo all transactions that committed after the dump

# SQL转储

- **Most database systems also support an SQL dump**
  - Write out SQL DDL statements and SQL insert statements to a file, which can then be executed to re-create the database.

# Main Contents

1.Failure Classification

2.Storage Structure

3.Recovery and Atomicity

4.Recovery Algorithm (Log-Based Recovery)

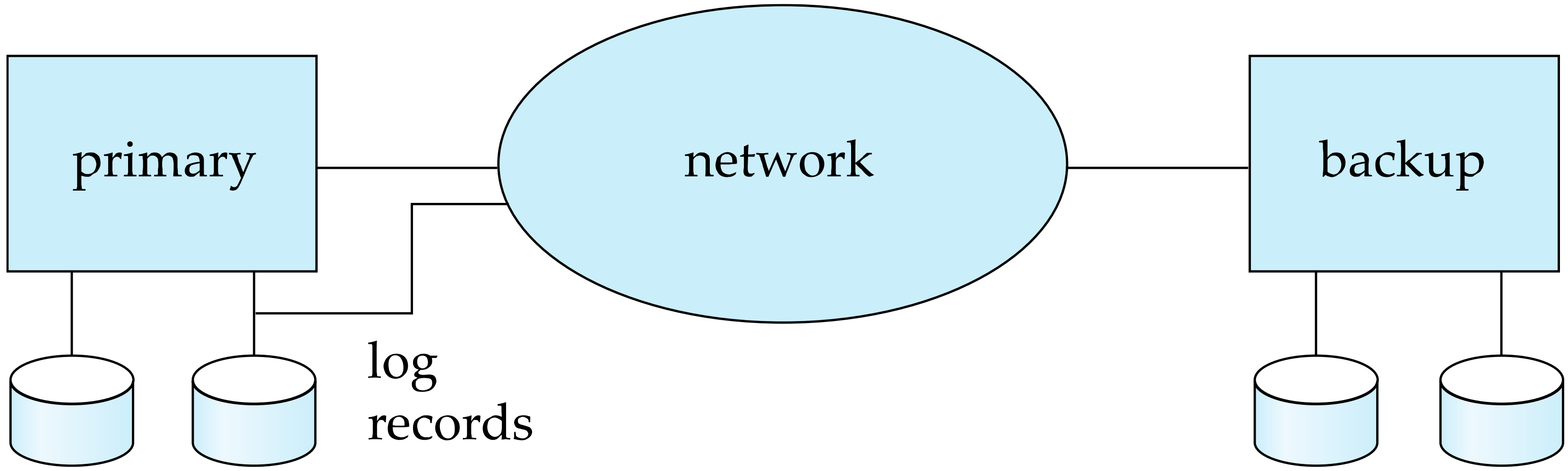
5.Buffer Management

6.Failure with Loss of Nonvolatile Storage

**7.Remote Backup Systems(19.7)**

# 远程备份系统

- Remote backup systems provide **high availability** by allowing transaction processing to continue even if the primary site is destroyed.



# 热备份配置

- **Hot-Spare** configuration permits very fast takeover
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup **rolls back incomplete transactions**, and is ready to process new transactions.

# Conclusions

- 故障恢复
  - 故障分类
    - 系统故障分类
      - 系统故障分类1: 事务故障
      - 系统故障分类2: 系统崩溃
      - 系统故障分类3: 磁盘故障
    - 系统故障带来的问题
      - 原子性问题
      - 持久性问题
    - 解决方案
      - 正常事务处理中的措施
      - 故障发生后的措施
  - 存储
    - 可靠存储
      - 易失存储 (volatile storage)
      - 非易失存储 (non-volatile storage)
      - 可靠存储 (stable storage)
    - 事务访问数据的基本流程
      - 磁盘与缓存之间的数据传输
      - 事务工作区与缓存之间的数据传输
  - 恢复与原子性
    - 基于日志的恢复机制
    - 日志生成机制/时机
    - 数据库修改策略
      - 立即修改
      - 延迟修改
    - 事务提交的标志
    - 并发事务的日志格式
    - 调度假设 – 杜绝写丢失
    - 事务的撤销与重做
    - 基于日志的事务恢复逻辑
    - 对于撤销日志的重做
    - 撤销与重做的代价
    - 检查点
      - 检查点机制
      - 检查点的恢复机制
      - 日志记录的删除
  - 基于日志的恢复算法
    - 故障类型1：事务回滚操作
    - 故障类型2：系统恢复的两个阶段
      - 重做阶段-Redo phase
      - 撤销阶段 - Undo phase
  - 缓存管理
    - 日志记录缓存
    - 先写日志规则
    - 数据库缓存
      - 非强制策略与窃取策略
      - 数据库缓存输出到硬盘
    - 操作系统对缓存管理的支持
      - 虚拟内存的问题
      - 解决方案
  - 带有非易失存储的故障
    - 故障类型3 - 磁盘故障
    - 数据转储
    - 数据恢复
    - SQL转储
  - 远程备份系统
    - 远程备份系统
    - 热备份配置

谢谢