Unit 12
Database Recovery
Contents

- 12.1 Introduction
- 12.2 Transactions
- 12.3 Transaction Failures and Recovery
- 12.4 System Failures and Recovery
- 12.5 Media Failures and Recovery
12.1 Introduction
Database Recovery: Introduction

- The Problem of Database Recovery
  - To restore the database to a **state** that is known to be **correct** after some failures.

- Possible Failures
  - programming errors, e.g. divide by 0, QTY < 0
  - hardware errors, e.g. disk crashed
  - operator errors, e.g. mounting a wrong tape
  - power supply, fire, ...

- Principle of Recovery:
  Backup is necessary
Database Recovery (cont.)

- **Basic approach**
  1. Dump database periodically.
  2. Write a log record for every change.
     e.g. E#, old_value, new_value, ...
  3. If a failure occurs:
     - CASE1: DB is damaged
       ==> archive copy + redo log = current DB.
     - CASE2: DB is not damaged but contents unreliable
       ==> undo some log.
12.2 Transactions

- unit of Work
- unit of Recovery
- unit of Concurrency (Unit 13)
Transactions: Concepts

- A logical unit of work.
- Atomic from the point of view of the end-user.
- An all-or-nothing proposition.

```plaintext
<e.g.> TRANSFER : PROC; /* transfer account */
    GET (FROM, TO, AMOUNT);
    FIND UNIQUE (ACCOUNT WHERE ACC#=FROM);
    ASSIGN (BALANCE - AMOUNT) TO BALANCE;
    IF BALANCE < 0
        THEN
            DO;
            PUT ('INSUFFICIENCY FUNDS');
            ROLLBACK;
            END;
        ELSE
            DO;
            FIND UNIQUE (ACCOUNT WHERE ACC# = TO);
            ASSIGN (BALANCE + AMOUNT) TO BALANCE;
            PUT ('TRANSFER COMPLETE');
            COMMIT;
            END;
    END;
```
Transactions: Example

\(<\text{e.g.}>\) [CASCADE CHANGE ON S.S\#  TO SP.S\#]

\text{CHANGE: PROC OPTIONS (MAIN)}
\begin{verbatim}
   EXEC SQL WHENEVER SQLERROR GOTO UNDO;
   GET LIST (SX, SY);
   (i)  EXEC SQL UPDATE S
       SET S# =: SY;
       WHERE S# =: SX;
   (ii) EXEC SQL UPDATE SP
       SET S# =: SY;
       WHERE S# =: SX;
   EXEC SQL COMMIT;
   GO TO FINISH;
UNDO: EXEC SQL ROLLBACK;
FINISH: RETURN;
END
\end{verbatim}
Transactions: Structure

- **Structure of a Transaction**
  
  - BEGIN TRANSACTION;
  
  ```
  /* application specified sequence of operations*/
  ...
  ```
  
  - COMMIT; /* signal successful termination */
  - (or ROLLBACK; /* signal unsuccessful termination*/) 

- **Implicit**
  
  BEGIN TRANSACTION, COMMIT, ROLLBACK may be implicit:
  
  - Program initiation: BEGIN TRANSACTION
  - Normal termination: COMMIT
  - Abnormal termination: ROLLBACK

- **Program and Transaction:**
  
  one program may contain several transactions.
Transactions: Manager

- **Transaction cannot be nested:**

  ![Diagram showing transactions](image)

  - Does Tx2 need to be rolled back?

- **Transaction Manager:**

  Transaction should not be lost, or partially done, or done more than once

  <e.g.> Consider the CASCADE example,

  if the system crashed between two updates

  ==> the first update must be **undone**!
Transactions: Commit and Rollback

- **COMMITH**:  
  - signal successful end-of-transaction.  
  - all updates made by that transaction can now be made permanent. (e.g. buffer to disk)

- **ROLLBACK**:  
  - signal unsuccessful end-of-transaction.  
  - the database may be in an inconsistent state.  
  - all update made by that transaction so far must be 'rolled back or undone'

- How to undone an update?  
  - system maintain a log or journal on tape or disk on which details of all update are recorded.
Transactions: Synchronization Point (SynchPoint)

- Represents the boundary between two consecutive transactions.
- Corresponds to the end of logical unit of work.
- A point at which the database is in a **state of consistency**.
- Established by COMMIT, ROLLBACK, and program initiation.

- When a **synchpoint** is established:
  - All updates since the previous **synchpoint** are committed (COMMIT) or undone (ROLLBACK)
  - All database positioning is lost. (e.g. cursor).
  - All record locks are released.
Types of Transaction Failure

- **Type 1 Transaction Failures:**
  - detected by the application program itself.
    - e.g. Insufficient Funds (balance < 0)
  - How to handle?
    - Issue the ROLLBACK command after the detection. (ref. p.12-7)

- **Type 2 Transaction Failures:**
  - not explicitly handled by the application
    - e.g. divide by zero, arithmetic overflow, ...

- **System Failures (Soft crash):**
  - affect all transactions currently in progress,
  - but do not damage the database. e.g. CPU failure.

- **Media Failures (Hard crash):**
  - damage the database.
  - affect all transactions currently using that portion.
    - e.g. disk head crash.
12.3 Type 2 Transaction Failures and Recovery
Transaction Failures and Recovery

- **Transaction Failures:**
  failures caused by unplanned, abnormal program termination.
  <e.g.> arithmetic overflow
  divided by zero
  storage protection violation
  log overflow...

- **How to recover transaction failures?**
  - System force a *rollback*.
  - the rollback is coordinated by **Recovery Manager**.
  - working *backward* through the *log*
    - to undo changes (replace new value by old value)
    - until the “**BEGIN TRANSACTION**” is encountered.
UNDO Logic and REDO Logic

- **UNDO Logic**

  UNDO Logic => cause the rollback procedure to be restarted from the beginning.

  • **Idempotent Property**: [Gray '78]

    \[ \text{UNDO} \left( \text{UNDO} \left( \text{UNDO} \left( \ldots \left( \text{UNDO} \left( x \right) \right) \right) \right) \right) = \text{UNDO} \left( x \right) \]

    i.e. undoing a given change any number of times is the same as undoing it exactly once.

- **REDO Logic**

  \[ \text{REDO} \left( \text{REDO} \left( \text{REDO} \left( \ldots(x) \right) \right) \right) = \text{REDO} \left( x \right) \]

  for all x.
Log

- **On-line log (active log) v.s. Off-line log (archive log):**
  - log data: 200 million byte/day ==> infeasible to be stored entirely on-line
  - active log: stored on disk if full ==> dump to tape ==> archive log.

- **Log Compression**
  - Archive log can be compressed
    - => reduce storage, and then increasing efficiency
  - How to compress archive log?
    - log records for transactions that failed to commit can be deleted (since they have been rolled back).
    - old values are no longer needed for the transactions that did **commit** (since they will never have to be undone).
    - changes can be consolidated (only the final value is kept)

![Diagram of transaction log](image)
Long Transaction

- Transaction is unit of work, and unit of recovery.
  - Transaction should be short.
    => reduce the amount that has to be undone.
- long transaction => subdivided into multiple transactions.
  <e.g.> $T_1$: Update all supplier records, $S$.
     \[\downarrow\]
     $T_{11}$: Update all supplier records for supplier name is 'A%'.
     $T_{12}$: Update all supplier records for supplier name is 'B%'.
    .
    .
     $T_{1,26}$: Update all supplier records for supplier name is 'Z%'.

12.4 System Failures and Recovery
System Failures and Recovery

- Critical point: contents of main storage are lost, in particular, the database buffers are lost. *e.g.* CPU failure.

- How to recover?
  1. UNDO the transactions in progress at the time of failure. *e.g.* $T_3, T_5$
  2. REDO the transactions that successfully complete but did not write to the physical disk.

- *<e.g.>*

```
Time
T1  T2  T3  T4  T5
```

- $T_1$: no need to be undone or redone
- $T_2, T_4$: must be redone
- $T_3, T_5$: must be undone
System Failures and Recovery

- How does the system know: which transaction to redo and which to undo?
  - Taking a checkpoint:
    - at certain prescribed intervals
    - involves:
      (1) writing the contents of the database buffers out to the physical database. e.g. disk
      (2) writing a special checkpoint record (contains a list of transactions which are in progress) e.g. \{T2, T3\} in progress

<e.g.>

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1: no need to be undone or redone
T2, T4: must be redone
T3, T5: must be undone
Decide undo and redo list

Decide the **undo list** and **redo list** by the following procedure:

**STEP 1:**

UNDO-list = list of transactions given in the checkpoint record = \{T2, T3\}

REDO-list = \{ \}

**STEP 2:**

Search **forward** through the log, starting from the checkpoint, to the end of log:

- if a 'BEGIN TRANSACTION' is found => add to UNDO-list \{T2, T3, T4, T5\}
- if a 'COMMIT' is found => remove from UNDO-list to REDO-list

UNDO-list = \{T3, T5\} 　做一半的，要undo
REDO-list = \{T2, T4\} 　應該已做完，不確定有無 write to disk

**<3> Undo:** System works **backward** through the log, **undoing** the UNDO-List.

**<4> Redo:** System then works **forward** through the log, **redoing** the REDO-List
Write-Ahead Log Protocol

- **Write-Ahead Log Protocol** (i.e. Log first protocol)
  
  **Note:** 'write a change to database' and 'write the log record to log' are two distinct operations
  
  => failure may occur between them!

- Before writing a record to physical database, the log record must first be written to physical log.
- Before committing a transaction, all log records must first be written to physical log.
Write-Ahead Log Protocol (cont.)

- Why log need to write ahead? (Think!)

<table>
<thead>
<tr>
<th>Time</th>
<th>tc</th>
<th>tf</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Check point
- System failure

**Case 1**
- Write log
- Write db
- Time
- Write log
- Write db

**Case 2**
- Write log
- Write Commit log
- Commit

T1: no need to be undone or redone
T2, T4: must be redone
T3, T5: must be undone
12.5 Media Failures and Recovery
Types of Transaction Failure

- **Type 1 Transaction Failures:**
  - detected by the application program itself.
    - e.g. Insufficient Funds (balance < 0)
  - How to handle?
    - Issue the ROLLBACK command after the detection. (ref. p.12-7)

- **Type 2 Transaction Failures:**
  - not explicitly handled by the application
    - e.g. divide by zero, arithmetic overflow, ...

- **System Failures (Soft crash):**
  - affect all transactions currently in progress,
  - but do not damage the database. e.g. CPU failure.

- **Media Failures (Hard crash):**
  - damage the database.
  - affect all transactions currently using that portion.
    - e.g. disk head crash.
Media Failures and Recovery

- **Critical point:**
  Some portion of the secondary storage is damaged.

- **How to recover?**
  1. Load the database to new device from the most recent archive copy (old DB.)
  2. Use the log (both active and archive) to redo all the transactions that are completed since that dump was taken.

  **Note:** Assume log does not fail. (Duplex log to avoid log failure.)
end of unit 12