# Database Design & Applications

## Transaction Management

# Objectives

- What is transaction
- Transaction State
- Schedules
- Serializability
- Recoverability
- Transactions in SQL Server
- Defining Explicit Transaction

# What is a Transaction?

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

# Transaction Properties

To preserve integrity of data, the database system must ensure:

- **Atomicity:** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency:** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$, finished execution before $T_i$ started, or $T_j$ started execution after $T_i$ finished.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

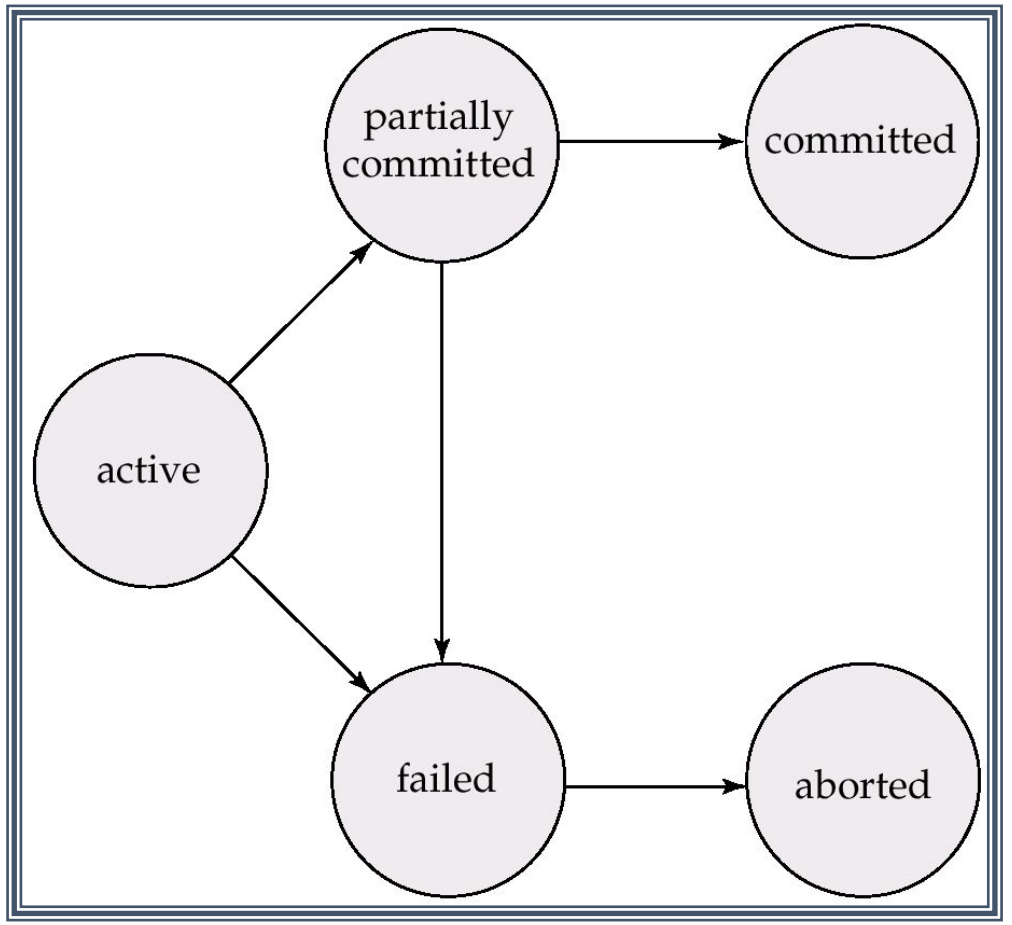
# Example of Transaction

- Transaction to transfer $50 from account A to account B:
  1. read(A)
  2. A := A – 50
  3. write(A)
  4. read(B)
  5. B := B + 50
  6. write(B)
- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.
- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

# Example of Transaction (Continue…)

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database. (the sum A + B will be less than it should be).

# Transaction State



# Transaction State (Cont.)

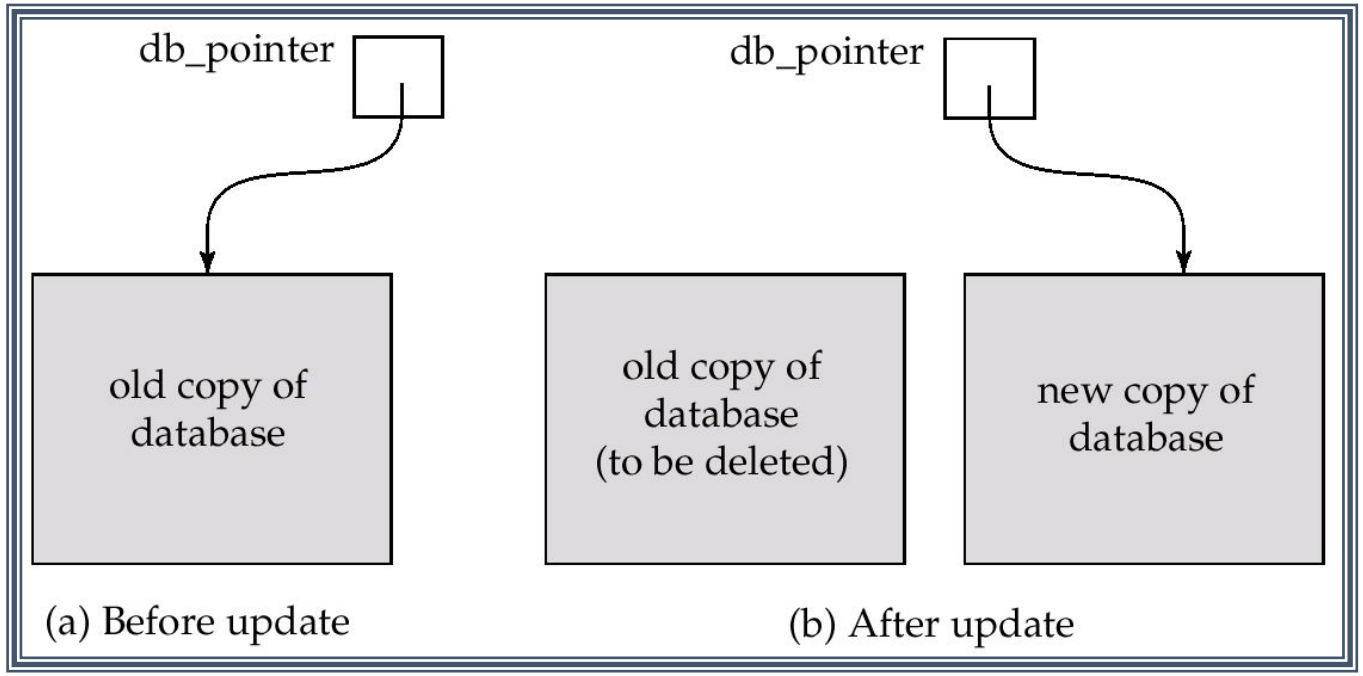
- **Active, the initial state;** the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction – only if no internal logical error
  - kill the transaction
- **Committed**, after successful completion.

# Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.
- **The shadow-database scheme:**
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a shadow copy of the database, and db_pointer is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by db_pointer can be used, and the shadow copy can be deleted.

# Implementation of Atomicity and Durability (Cont.)

**The shadow-database scheme:**




- Assumes disks to not fail
- Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying the entire database.

# Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
  - reduced average response time for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

# Schedules

- **Schedules** – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

# Example Schedules

- Let $\mathbf{T_1}$ transfer \$50 from A to B, and $\mathbf{T_2}$ transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which $T_1$ is followed by $T_2$.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write ($A$) | |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| | read($B$) |
| | $B := B + temp$ |
| | write($B$) |

# Example Schedule (Cont.)

- Let $T_1$ and $T_2$ be the transactions defined previously. The following schedule (Schedule 3 in the text) is not a serial schedule, but it is equivalent to Schedule 1.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write($A$) | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($B$) |
| | $B := B + temp$ |
| | write($B$) |

In both Schedule 1 and 3, the sum A + B is preserved.

# Example Schedules (Cont.)

- The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the the sum A + B.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| | read($B$) |
| write($A$) | |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | $B := B + temp$ |
| | write($B$) |

# Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

# Recoverability

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction $T_j$ reads a data items previously written by a transaction $T_i$ , the commit operation of $T_i$ appears before the commit operation of $T_j$.
- The following schedule is not recoverable if $T_9$ commits immediately after the read.

| $T_8$ | $T_9$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| read($B$) | |

- If T8 should abort, T9 would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable.

# Recoverability (Cont.)

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

| $T_{10}$ | $T_{11}$ | $T_{12}$ |
|---|---|---|
| read($A$) | | |
| read($B$) | | |
| write($A$) | | |
| | read($A$) | |
| | write($A$) | |
| | | read($A$) |

If $T_{10}$ fails, $T_{11}$ and $T_{12}$ must also be rolled back.

- Can lead to the undoing of a significant amount of work

# When to Use Transactions

- The following are some frequent scenarios where use of transactions is recommended:
  - In batch processing, where multiple rows must be inserted, updated, or deleted as a single unit
  - Whenever a change to one table requires that other tables be kept consistent
  - When modifying data in two or more databases concurrently
  - In distributed transactions, where data is manipulated in databases on various servers
- When you use transactions, you put locks on data that is pending for permanent change to the database.
- No other operations can take place on locked data until the acquired lock is released.

# Specifying Transaction Boundaries

- SQL Server transaction boundaries help you to identify when a SQL Server transaction starts and ends by using API functions and methods as in the following:
- **Transact-SQL statements:**
  - BEGIN TRANSACTION
  - COMMIT TRANSACTION
  - COMMIT WORK
  - ROLLBACK TRANSACTION
  - ROLLBACK WORK
  - SET IMPLICIT_TRANSACTIONS
- **API functions and methods:**
  - Database APIs such as ODBC, OLE DB, ADO, and the .NET Framework SqlClient namespace contain functions or methods used to delineate transactions.

# Transactions in SQL Server

- All database engines are supposed to provide built-in support for transactions.
- Transactions that are restricted to only a single resource or database are known as local transactions.
- There are following transaction modes :
  - **Implicit Transaction:** When you connect to a database using SQL Server Management Studio and execute a DML query, the changes are automatically saved.
  - **Explicit Transaction:** Explicit transactions are those in which you explicitly control when the transaction begins and when it ends.

# How to define an Explicit Transaction in SQL Server

- In order to define an explicit transaction, we start to use the
- **BEGIN TRANSACTION** command
- **BEGIN TRANSACTION** [transaction_name]
  - transaction_name option is used to assign a specific name to transactions
- After defining **BEGIN TRANSACTION** command, the related resources acquired a lock depending on the isolation level of the transaction
- The **COMMIT TRANSACTION** statement applies the data changes to the database and the changed data will become permanent.
- **ROLLBACK TRANSACTION** statement helps in undoing all data modifications that are applied by the transaction

# Explicit Transaction Example

- BEGIN TRANSACTION
- UPDATE Countries
- SET City = 'Mumbai' where City= 'Bombay'
- COMMIT TRANSACTION

- BEGIN TRAN
- UPDATE Countries
- SET City = 'Mumbai' where City= 'Bombay'
- ROLLBACK TRAN

# Save Points in Transactions

- Savepoints can be used to rollback any particular part of the transaction rather than the entire transaction.
- To define a save point in a transaction we use the syntax:
- **SAVE TRANSACTION** transaction_name

```
BEGIN TRANSACTION
INSERT INTO Person
VALUES('Mouse', 'Micky','500 South Buena Vista Street, Burbank','California',43)
SAVE TRANSACTION InsertStatement
DELETE Person WHERE PersonID=3
SELECT * FROM Person
ROLLBACK TRANSACTION InsertStatement
COMMIT
```

# Transaction ACID Test

- A transaction is a group of database commands that are treated as a single unit. A successful transaction must pass the 'ACID' test:
- **ATOMIC:** All statements in the transaction either completed successfully or they were all rolled back. The task that the set of operations represents is either accomplished or not, but in any case not left half-done.

# Transaction ACID Test

- **CONSISTENT:** All data touched by the transaction is left in a logically consistent state.

```
CREATE PROCEDURE update_job
@p_empid int,
@p_jobid varchar(10),
@p_deptid int,
@p_startdate date
as
BEGIN
    declare @v_job_id varchar(10)
    declare @v_dept_id int
    declare @v_enddate date
    BEGIN TRY
        BEGIN TRANSACTION
            select @v_job_id =job_id, @v_dept_id =department_id , @v_enddate=hire_date
            from employees where employee_id =@p_empid
            update employees
            set job_id =@p_jobid, department_id =@p_deptid, hire_date=@p_startdate
            where employee_id =@p_empid
            insert into job_history(employee_id, start_date, end_date, job_id, department_id)
            values (@p_empid, @p_startdate, @v_enddate,@v_job_id,@v_dept_id)
        COMMIT TRANSACTION
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION
    END CATCH
END
```

# Transaction ACID Test

- **Isolation :** The transaction must affect data without interfering with other concurrent transactions, or being interfered with by them. This prevents transactions from making changes to data based on uncommitted information, for example changes to a record that are subsequently rolled back. Most databases use locking to maintain transaction isolation.
- **Durable:** Once a change made, it is permanent. If a system error or power failure occurs before a set of commands is complete, those commands are undone and the data is restored to its original state once the system begins running again.


# THANK YOU!