Transaction and Concurrency Control

A. Transaction

- A **transaction** is a collection of operations that form a single logical unit of work.
  Example: As a customer, you will make a payment from the cashier to purchase two (2) pairs of Nike shoes. From a customer's standpoint, this can be considered as a single operation. In the database, however, it consists of several operations, such as writing a new customer's invoice, reducing the quantity of on-hand products in inventory, and updating the sales. If one operation is not successful, all other operations will be undone.
- A **database request** is the equivalent of a single SQL statement in an application program or transaction.
  - If a transaction has three (3) *update* statements and one (1) *insert* statement, the transaction uses four (4) database requests.
- **Database consistent state –** satisfies the constraints specified in the schema.
  - Assume that a specific table contains a gender column where it only accepts an entry having values of "Male" and "Female". If a user attempts to enter a value of 'Person', then the database will disallow the entry of such a value.
- **Transaction log –** A DBMS uses this to keep track of all the transactions that update the database.
  - In the case of system failure, this log helps bring the database back to a consistent state.

Properties of transactions

- **Atomicity –** requires that all operations (SQL requests) of a transaction should be completed.
  Example: Transaction T1 has four (4) SQL requests that must be successfully completed. Otherwise, the entire transaction is aborted.
- **Consistency –** ensures that only valid data following all rules and constraints will be written in the database. When a transaction results in invalid data, the database reverts to its previous state
- **Isolation**: The data used during the execution of a current transaction cannot be used by another transaction until the first one is completed.
  Example: If two (2) people use the same ATM card and make a transaction at the same time, the first one to be connected will have its first transaction. Therefore, other accesses from that account would be locked until the existing session is over.
- **Durability –** ensures that once transaction changes are done and committed, they cannot be undone or lost.

Real-life transaction: Fund transfer

A transaction to transfer ₱1000 from the bank account of Aldous to the account of Brendon:
1. read Aldous's account
2. Aldous's account = Aldous's account – ₱1000
3. Write changes in Aldous's account
4. Read Brendon's account
5. Brendon's account = Brendon's account + ₱1000
6. Write changes in Brendon's account

- **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, an inconsistency will result.
- **Consistency requirement**: The sum of Aldous' account and Brendon's account is unchanged by the execution of the transaction. If between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database.
- **Isolation requirement**: Transactions should be run sequentially. Therefore, no other transaction will happen on both accounts until the current transaction is completed.
- **Durability requirement**: Once the user has been notified that the transaction has completed (i.e., the transfer of the P1000 has taken place), the updates to the database by the transaction must persist despite failures.

SQL Transactional Commands

- **BEGIN TRANSACTION** - This marks the beginning of transaction execution.

- **COMMIT** - this signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **ROLLBACK TRANSACTION** - This signals that the transaction has ended unsuccessfully so that any changes or effects that the transaction may have applied to the database must be undone.
- **SAVE TRANSACTION** - is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.
- **@@TRANCOUNT** - returns the number of BEGIN TRANSACTION statements that have occurred on the current connection.

Transaction Execution States

- **Active state** - in this state, a transaction stays in this state to perform READ and WRITE operations.
- **Partially committed state** - where the final statement in queries has been executed
- **Committed state** - after all the operation has been completed
- **Failed State** - if one of the operations cannot be done or proceed
- **Terminated** - corresponds to the transaction leaving the system and if it is either committed or aborted

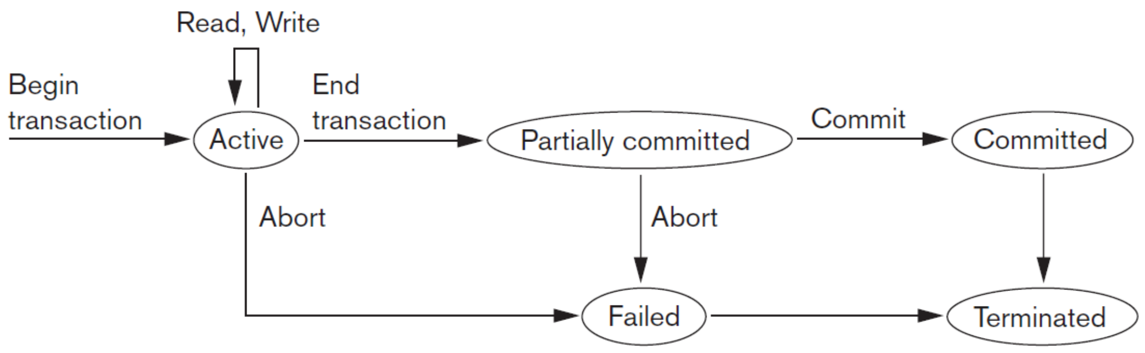


Figure 1. *Transaction execution states*

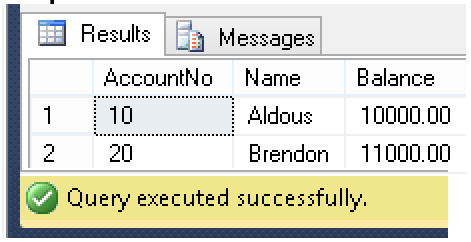
Assume that we have a table named *BankAccount* with the following columns and values:

AccountNo	Name	Balance
10	Aldous	11,000
20	Brendon	10,000

```
BEGIN
BEGIN TRANSACTION

  --Deduct the amount needed for fund transfer
  UPDATE BankAccount
    SET Balance = Balance - 1000
    WHERE AccountNo = 10

  --Add the amount to complete the fund transfer
  UPDATE BankAccount
    SET Balance = Balance + 1000
    WHERE AccountNo = 20
    SAVE TRANSACTION FundTransfer

  --Aldous current balance is 10,000 and he tries to
withdraw an amount of 11,000.
  UPDATE BankAccount
    SET Balance = Balance - 11000
    WHERE AccountNo = 10
  DECLARE @balance INT
  SET @balance = (SELECT Balance
FROM BankAccount WHERE AccountNo = 10)

  --Check the balance if sufficient to the amount being
withdrawn
  IF @balance >= 0
    BEGIN
      SAVE TRANSACTION FundWithdraw
    END
  ELSE IF @balance < 0
    BEGIN
      ROLLBACK TRANSACTION FundTransfer
    END
    COMMIT
END
```

The following code is an example of transactional commands based on the real-life transaction: fund transfer.

Output:



```
Results   Messages
   AccountNo  Name     Balance
1  10         Aldous   10000.00
2  20         Brendon  11000.00
Query executed successfully.
```

Explanation:

- Before starting the first query, we declare a command that the following statements will be classified as a transaction using the "BEGIN TRANSACTION"
- The first query deducts the amount from Aldous's account that will be needed for fund transfer.
- The second query adds the amount to Brendon's account to complete the fund transfer. After completing, we saved the completed transaction in order to prevent from rolling back the entire transaction in case of having an issue in the next transaction.
- Since Aldous's current balance is 10,000 after the transaction, he tries to withdraw an amount of 11,000 to his account. The third query updates the balance of Aldous by deducting the amount he requests to be withdrawn to his balance.
- The third query will only be executed if the balance of Aldous is sufficient for the amount being withdrawn. To achieve this, we create an if-else statement and else if statement.
- In the if-else statement, we set a condition that if the balance of Aldous's account is greater than or equal to 0 after the third query, the transaction will be saved and applied these changes to the database. Afterwards, Aldous will successfully withdraw the amount he requested.
- In the else-if statement, we set a condition that if the balance is less than to 0 after the third query, the transaction in the third query must be undone as it violates the constraints specified in the schema. Aldous won't be able to withdraw the amount he requested.

B. Concurrency Control

- **Concurrency Control**: When several transactions execute simultaneously in the database, there will be a chance that the consistency and integrity of data may no longer be preserved. The system must control the interaction among the concurrent transactions, and this control is achieved through concurrency control techniques.

The main problems in concurrent transactions

- **Lost Update** – occurs when two concurrent transactions, T1 and T2, are updating the same data element, and one of the updates is lost (overwritten by the other transaction).
- **Uncommitted data** – occurs when two transactions, T1 and T2, are executed concurrently, and the first transaction (T1) is rolled back after the second transaction (T2) has already accessed the uncommitted data.
- **Inconsistent retrievals** - occur when a transaction accesses data before and after one or more other transactions finish working with the same data.

Concurrency Control with Locking Methods

- A **lock** guarantees exclusive use of data item to a current transaction.
    - o Transaction T2 does not have access to a data item that is currently being used by transaction T1.
    - o The lock is released (unlocked) when the transaction is completed.
- The **database-level lock:**
    - o Locks the entire database
    - o prevents the use of any tables in the database to transaction T2 while transaction T1 is being executed
    - o Good for batch processes, but not suitable for multiuser DBMS as it would slow down the data access if thousands of transactions had to wait for the current transaction to be completed
- The **table-level** lock:
    - o The entire table is locked.

- Prevents access to any row by transaction T2 while transaction T1 is using the table.
- If a transaction requires access to several tables, each table may be locked.
- Two (2) transactions can access the same database as long as they access different tables.
- Transactions T1 and T2 cannot access the same table even when they try to use different rows.

- A **row-level** lock:
  - Less restrictive
  - Allows concurrent transactions to access different rows of the same table even when the rows are located on the same page
  - Improves the availability of data but requires high usage of space because a lock exists for each row in a table of the database

Lock types

- A **binary lock**:
  - Has only two states: locked (1) and unlocked (0).
  - If an object such as a database, table, or row is locked by a transaction, no other transaction can use that object.
  - As a rule, the transaction must unlock the object after its termination.
  - These options are automatically managed by the DBMS.
  - User does not require to manually lock or unlock data items.

- A **shared/exclusive** lock:
  - An exclusive lock exists when access is reserved specifically for the transaction that locked the object.
  - A shared lock exists when a transaction wants to read data from the database, and no exclusive lock has held that data item.
  - Using the shared/exclusive locking concept, a lock can have three states: unlocked, shared (read), and exclusive (write).

Deadlocks

- A **deadlock** occurs when two (2) transactions wait indefinitely for each other to unlock data.

Example:

Concurrent transaction:
T1 **= access data items X and Y**
T2 **= access data items Y and X**

- If T1 has not unlocked data item Y, T2 cannot begin.
- If T2 has not unlocked data item X, T1 cannot continue.
- Consequently, T1 and T2 wait for each other to unlock the required data item.

- The three basic techniques to control deadlocks are:
  - Deadlock prevention – A transaction requesting a new lock is aborted when there is the possibility that a deadlock can occur. If the transaction is aborted, all changes made by this transaction are rolled back, and all locks obtained by the transaction are released.
  - Deadlock detection – The DBMS periodically tests the database for deadlocks. If a deadlock is found, the “victim” transaction is aborted (rolled back and restarted), and another transaction continues.
  - Deadlock avoidance – The transaction must obtain all the locks it needs before it can be executed. However, the serial lock assignment required in deadlock avoidance increases action response time.

Transaction Isolation Level

- Transaction isolation levels are described by the type of “reads” that a transaction allows or not. The types of *read* operation are:

- **Dirty Read** - a transaction can read data that is not committed yet.
  Example:
  1. The original salary of Mary was ₱15,000. The treasurer changed Mary's salary to ₱20,000 but did not commit the transaction.
  2. Mary read her wages and found an increase in her wages by ₱5,000.
  3. The treasurer found out that they had a mistake in the corresponding transaction. The transaction had been rolled back, and Mary's salary is back to ₱15,000. The ₱20,000 salary is now considered *dirty data*.
- **Non-repeatable read** - a transaction reads a given row at time T1, and then it reads the same row at time T2, yielding different results. The original row may have been updated or deleted.
  Example:

1. In Transaction 1, Mary read her own salary of ₱15,000, and the operation was not completed.
2. In Transaction 2, the Treasurer modified Mary's salary to ₱20,000 and submitted the transaction.
3. In Transaction 1, when Mary read her salary again, her salary changed to ₱20,000.

- **Phantom Read** - a transaction executes a query at time T1, and then it runs the same query at time T2, yielding additional rows that satisfy the query.
- Example:
  1. Transaction 1, read all employees with a salary of ₱1000 and return 10 rows.
  2. At this point, Transaction 2 inserts an employee record into the employee table with a salary of 1000
  3. Transaction 1 read all employees with a salary of ₱1000 again and return 11 rows.

Four (4) Transaction Isolation levels:

1. A **READ UNCOMMITTED**
- o is the least restrictive isolation level
- o it ignores locks placed by other transactions.
- o Can read modified data values that have not yet been committed by other transactions; these are called "dirty" reads.

Example: Assume that we have a table named *Stocks* and have the values of the following:

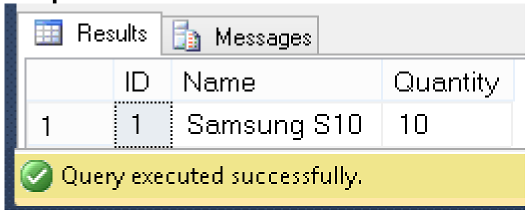
ID	Name	Quantity
1	Samsung S10	10

Table 1. *Stocks*

```
-- Transaction 1
BEGIN TRANSACTION
UPDATE Stocks set Quantity = Quantity - 1
   WHERE ID = 1
--BILLING THE CUSTOMER
WAITFOR DELAY '00:00:15'
ROLLBACK TRANSACTION

SELECT * FROM Stocks
```

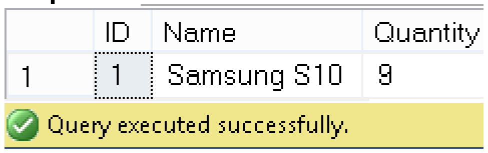
```
--TRANSACTION 2
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT * FROM Stocks WHERE ID = 1
```

Output for the Transaction 1:



```
   ID  Name         Quantity
1  1   Samsung S10  10
Query executed successfully.
```

Output for the Transaction 2:



```
   ID  Name         Quantity
1  1   Samsung S10  9
Query executed successfully.
```

Explanation:
- In Transaction 1, we set the following statements to be a transaction using the BEGIN TRANSACTION command.
- The first query reduced the quantity of Samsung S10 by 1.
- The next statement used a WAITFOR DELAY function in order to delay the execution of the previous query (15 seconds).
- We used the ROLLBACK TRANSACTION command in order to undo and end the existing transaction.
- Before writing the query in Transaction 2, we used the READ UNCOMMITTED command in order to return the rows even if it's not committed yet.
- In order to run this concurrent transaction, use two (2) windows for writing the queries/codes and execute Transaction 1 first. While Transaction 1 is waiting for its execution through the WAITFOR DELAY function, execute the second window or Transaction 2.

2. A READ COMMITTED
- o Default isolation level for SQL Server

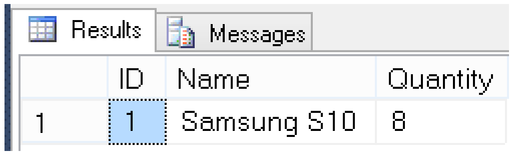
- Prevents dirty reads by specifying that statements cannot read data values that have been modified but not yet committed by other transactions
- Other transactions can still modify, insert, or delete data between executions of individual statements within the current transaction, resulting in non-repeatable reads, or "phantom" data.

Example: Using the *Stocks* (Table 1).

```
--Transaction 1 READ COMMITTED
BEGIN TRANSACTION
UPDATE Stocks
SET Quantity = Quantity - 2
WHERE ID = 1
WAITFOR DELAY '00:00:010'
COMMIT
SELECT * FROM Stocks
```

```
--Transaction 2
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
SELECT * FROM Stocks
COMMIT
```

Output of the two (2) transactions:



	ID	Name	Quantity
1	1	Samsung S10	8

Explanation:

- If you try to execute these two (2) transactions concurrently, both results will display an "executing query" loading. (See figure 1.)
- Since we used the READ COMMITTED command in transaction 2, transaction 2 will only return the rows from the requested query until the transaction 1 is committed.
- This is because Transaction 2 was trying to access the same data that is being used by transaction 1 and not been committed yet.

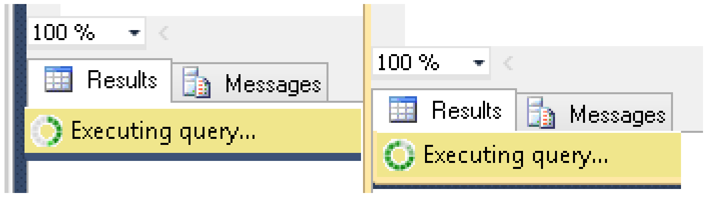


***Figure 1.** "Executing query" loading*

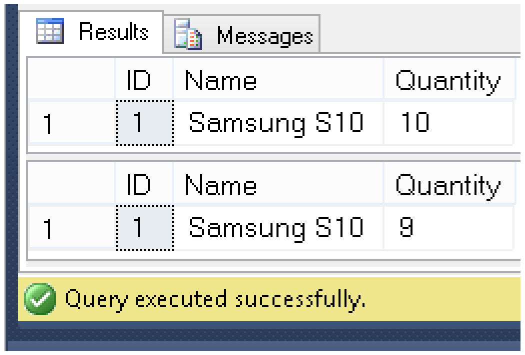
3. A **REPEATABLE READ**
   - A more restrictive isolation level than READ COMMITTED
   - Encompasses READ COMMITTED
   - Ensures that no other transactions can modify or delete data that has been read by the current transaction until the current transaction commits
   - Does not prevent other transactions from inserting new rows into the tables which have been using in the existing transaction, resulting in "phantom reads"

Example: Using the *Stocks* (table 1).

```
--Transaction 1 REPEATABLE READ
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
WAITFOR DELAY '00:00:10'
SELECT * FROM Stocks
COMMIT
SELECT * FROM Stocks
```

```
--Transaction 2 REPEATABLE READ
BEGIN TRANSACTION
UPDATE Stocks
SET Quantity = Quantity - 1 WHERE ID = 1
WAITFOR DELAY '00:00:01'
COMMIT
```

Output of the Transaction 1:



	ID	Name	Quantity
1	1	Samsung S10	10

	ID	Name	Quantity
1	1	Samsung S10	9

Query executed successfully.

Explanation:
- In Transaction 1, we want to query the data from *Stocks* with a TRANSACTION ISOLATION LEVEL REPEATABLE READ.
- In Transaction 2, we want to modify the quantity of Samsung S10 in *Stocks* by reducing its quantity by 1.
- Since Transaction 1 is the first transaction that has been executed, this will prevent other transaction (Transaction 2) to modify the data that was being used by the first transaction (Transaction 1).
- Note that this isolation level does not prevent you from inserting a new entry even if the object was being locked.

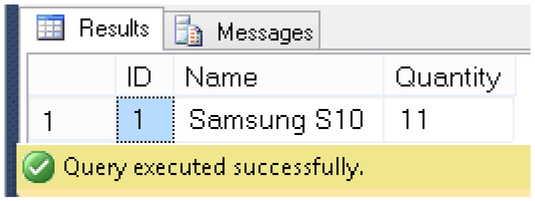
4. A **SERIALIZABLE isolation**
   - o Ensures that the data that one transaction has read, will be prevented from being updated or deleted by any other transaction.
   - o Most restrictive level and gives solution to the phantom read. problem.

     Example: **U**sing the *Stocks* (table 1)

```
--Transaction 1 SERIALIZABLE
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION
UPDATE Stocks
SET Quantity = Quantity + 1 WHERE ID = 1
SELECT * FROM Stocks
WAITFOR DELAY '00:00:010'
COMMIT
```

```
--Transaction 2 SERIALIZABLE
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION
INSERT INTO Stocks VALUES (3, 'Iphone 11', 3)
SELECT * FROM Stocks
WAITFOR DELAY '00:00:01'
COMMIT
```

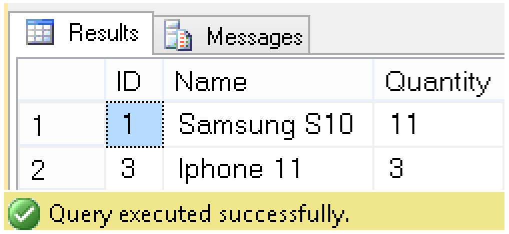
Output for Transaction 1:



	ID	Name	Quantity
1	1	Samsung S10	11

Query executed successfully.

Output for Transaction 2:



	ID	Name	Quantity
1	1	Samsung S10	11
2	3	Iphone 11	3

Query executed successfully.

Explanation:

- In Transaction 1, we modify the quantity of Samsung S10 by adding 1.
- In Transaction 2, we execute a concurrent transaction that will insert a new entry to table stocks.
- Upon using SERIALIZABLE isolation, Transaction 1 does not allow Transaction 2 to insert any entry or rows as it protects the transaction from all three read problems (dirty read, non-repeatable read, and phantom read).

REFERENCES

Coronel, C. and Morris, S. (2018). *Database systems design, implementation, & management (13th ed.)*. Cengage Learning.

Elmasri, R. & Navathe, S. (2016). *Fundamentals of Database systems (7th ed.)*. Pearson Higher Education.

Kroenke, D. & Auer, D. *Database processing: Fundamentals, design, and implementation (12th ed.)*. Pearson Higher Education.

Silberschatz A., Korth H.F., & Sudarshan, S. (2019). *Database system concepts (7th ed.).* McGraw-Hill Education.

Microsoft. (2017). *Transactions*. Retrieved from https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql?view=sql-server-ver15

Microsoft (2018). *SET TRANSACTION ISOLATION LEVEL (Transact-SQL)*. https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver15