Database Programmability

Functions, Stored Procedures, Triggers and Transactions

Table of Contents

1. Transact-SQL Programming
2. Functions
3. Stored Procedures
4. Transactions
5. ACID Model
6. Triggers
7. Database Security

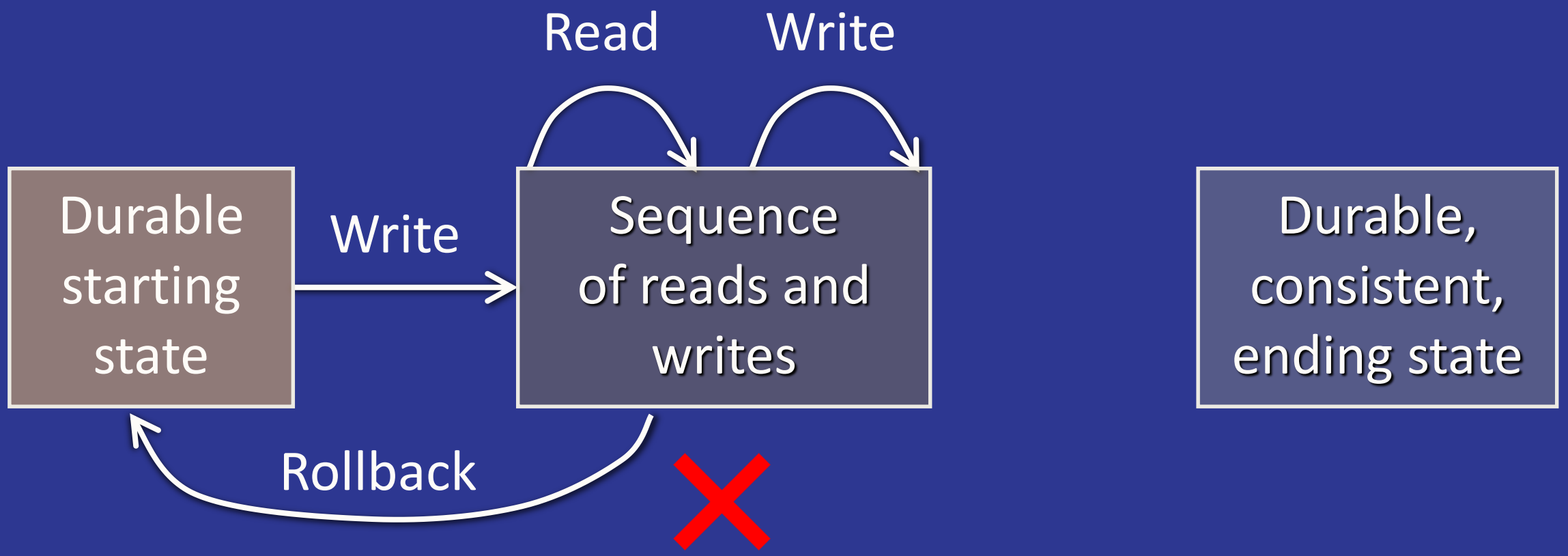
Transactions

Definition, Usage, ACID Model

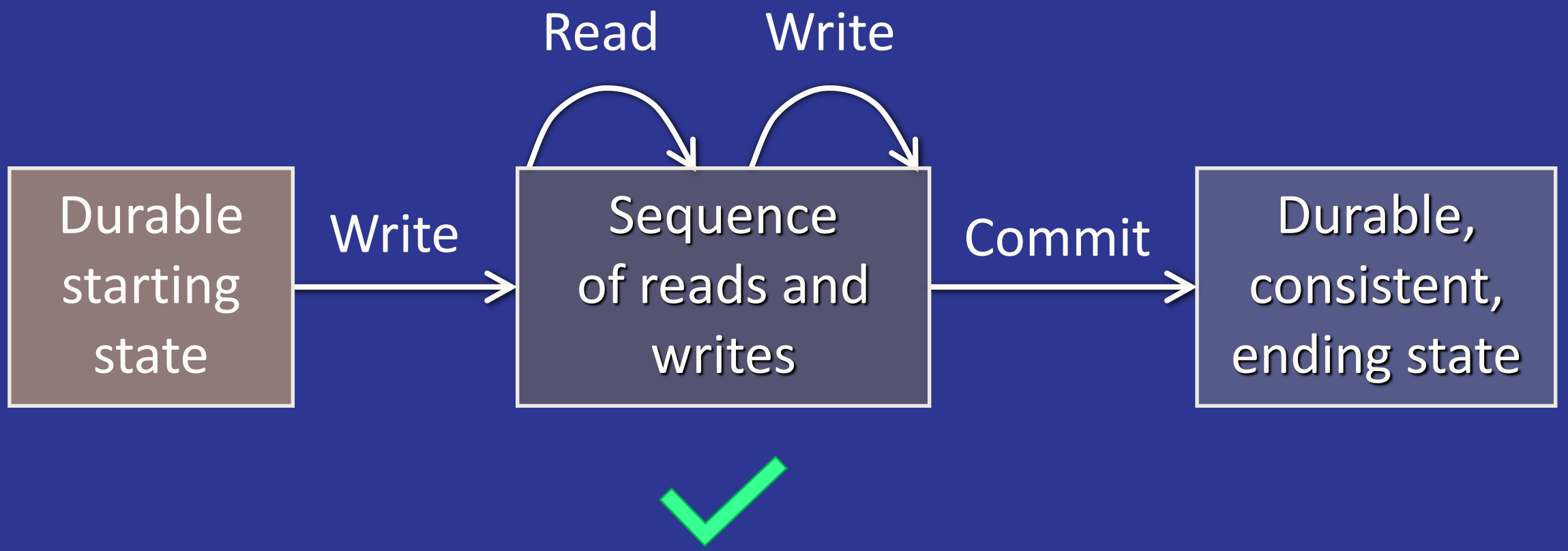
Transactions

- A **Transaction** is a sequence of actions (database operations) executed as a whole:
  - Either all of them complete successfully or none of them do
- Examples:
  - A bank transfer from one account into another (withdrawal + deposit)
  - If either the withdrawal or the deposit fails the whole operation is cancelled

Transactions: Lifecycle (Rollback)



Transactions: Lifecycle (Commit)



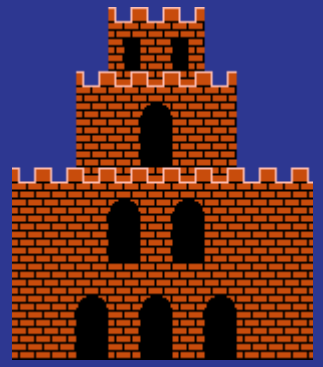
Transactions Behavior

- Transactions guarantee the consistency and the integrity of the database
  - All changes in a transaction are temporary
  - Changes are persisted when **`COMMIT`** is executed
  - At any time, all changes can be canceled by **`ROLLBACK`**
- All changes are persisted at once
  - As long as **`COMMIT`** is called

Transactions: What Can Go Wrong?

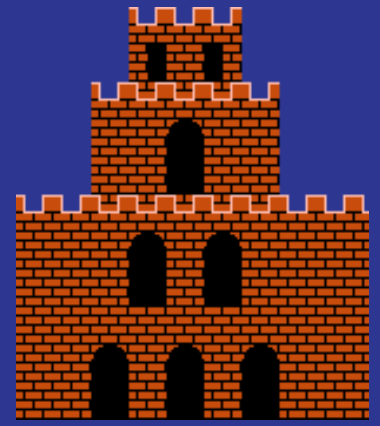
- Some actions fail to complete
  - The application software or database server crashes
  - The user cancels the action while it's in progress
- Interference from another transaction.
  - What happens if several transfers run for the same account at the same time?

Checkpoints in games



Castle 1-1

DIE

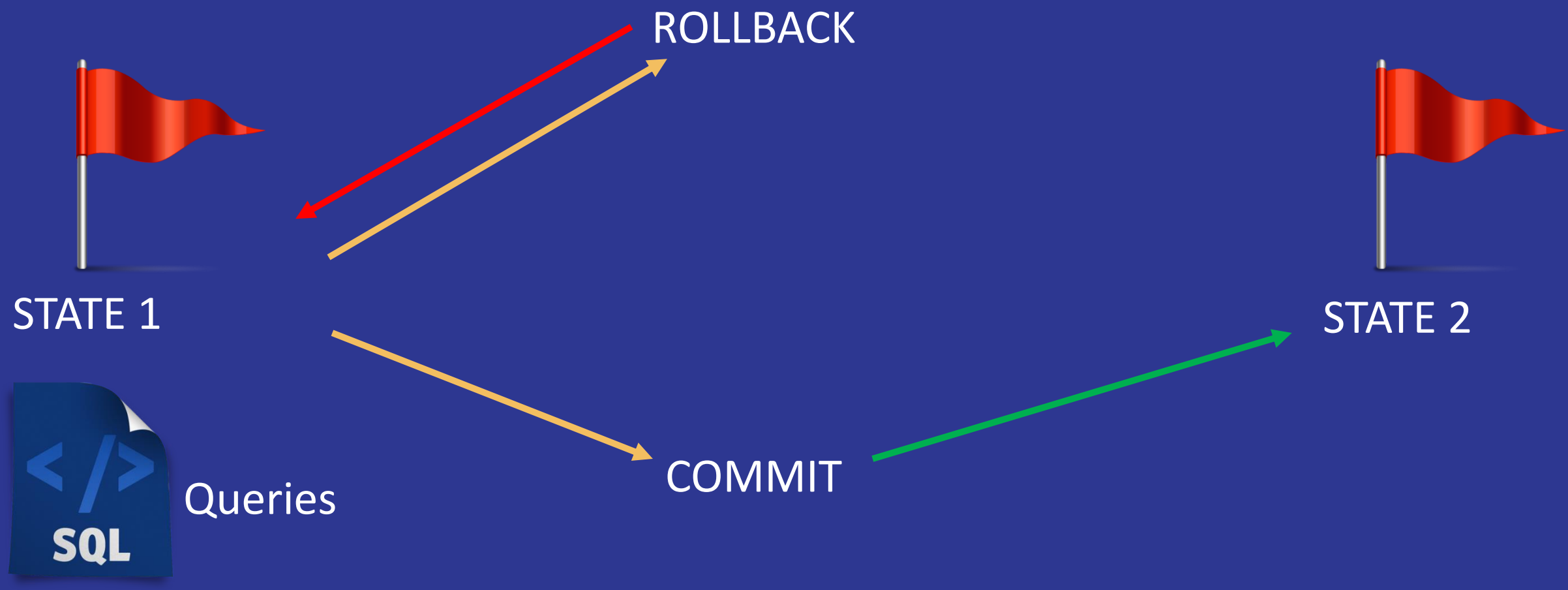


Castle 1-2

Mario

SURVIVE

What are Transactions?



Transactions Syntax

Start Transaction

Withdraw Money

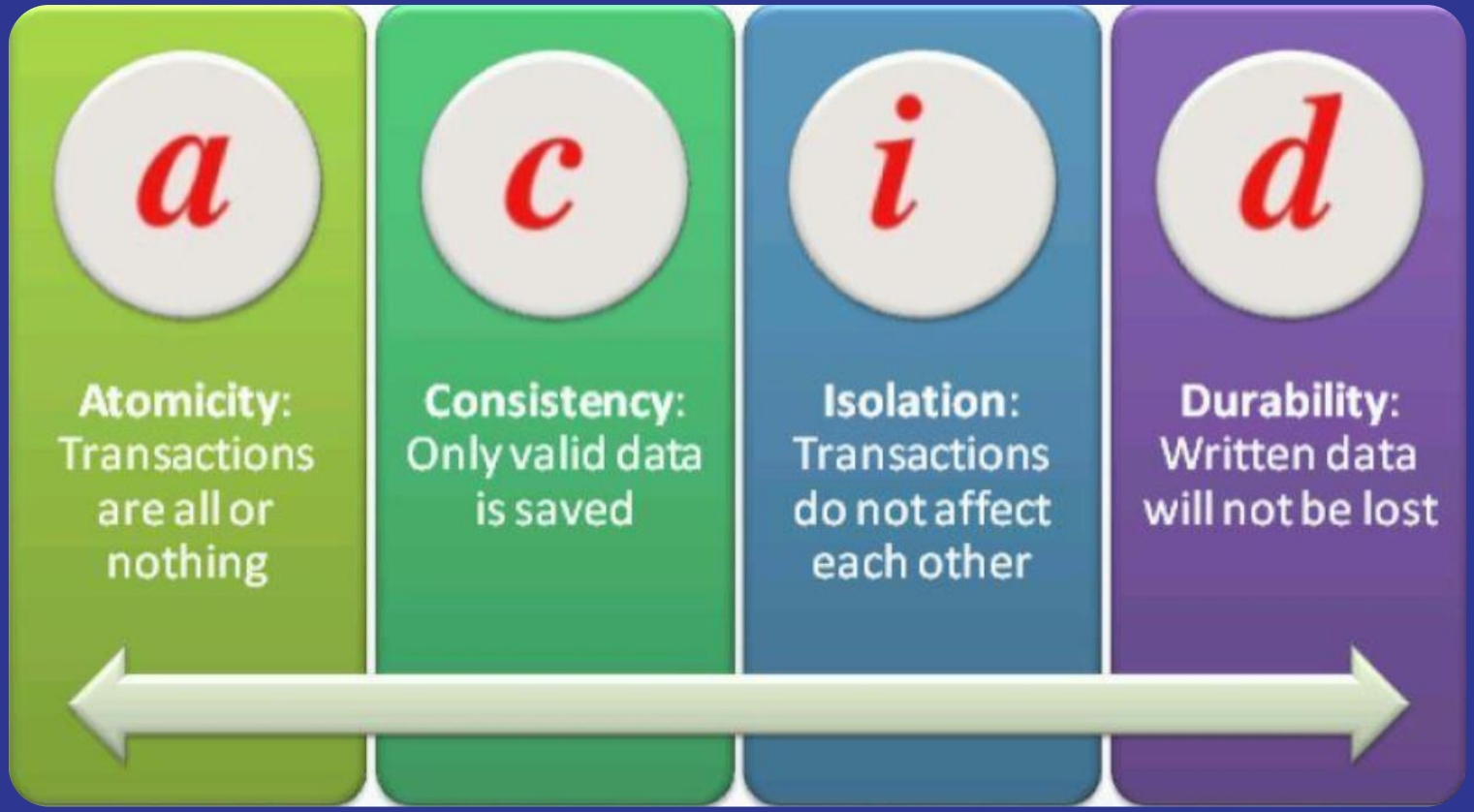
```
BEGIN TRANSACTION

UPDATE Accounts SET Balance = Balance - @withdrawAmount
WHERE Id = @accountId
IF @@ROWCOUNT <> 1 – Didn't affect exactly one row
BEGIN
   ROLLBACK
   RAISERROR('Invalid account!', 16, 1)
   RETURN
END

COMMIT
```

Undo Changes

Save Changes



ACID Model

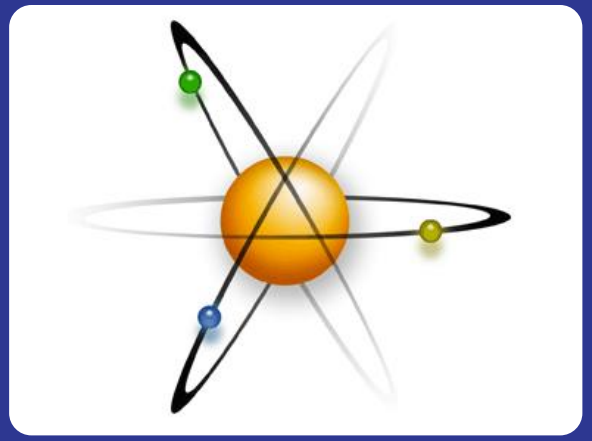
Solving problems before they arise

Transaction Properties

- Modern DBMS servers have built-in transaction support
  - Implement "ACID" transactions.
  - E.g. MS SQL Server, Oracle, MySQL, …
- ACID means:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

Atomicity

- Atomicity means that:
  - Transactions execute as a whole
  - DBMS guarantees that either all of the operations are performed or none of them



- Example: Transferring funds between bank accounts
  - Either withdraw + deposit both succeed, or none of them do
  - In case of failure, the database stays unchanged

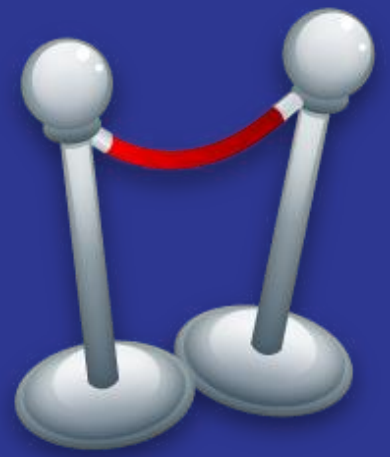
Consistency

- Consistency means that:
  - The database has a legal state in both the transaction's beginning and end
  - Only valid data will be written to the DB
  - Transaction cannot break the rules of the database
    - Primary keys, foreign keys, check constraints…

- Consistency example:
  - Transaction cannot end with a duplicate primary key in a table

Isolation

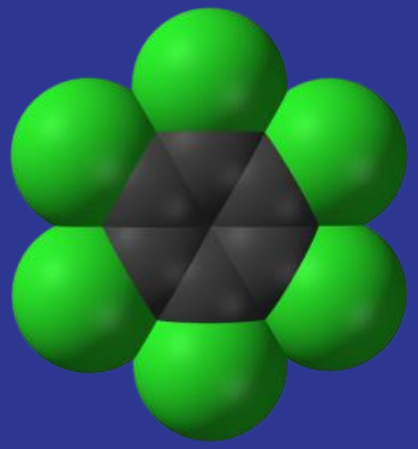
- Isolation means that:
  - Multiple transactions running at the same time do not impact each other's execution
  - Transactions don't see other transactions' uncommitted changes
  - Isolation level defines how deep transactions isolate from one another



- Isolation example:
  - If two or more people try to buy the last copy of a product, only one of them will succeed

Durability

- Durability means that:
  - If a transaction is committed it becomes persistent
    - Cannot be lost or undone
  - Ensured by use of database transaction logs



- Durability example:
  - After funds are transferred and committed the power supply at the DB server is lost
  - Transaction stays persistent (no data is lost)

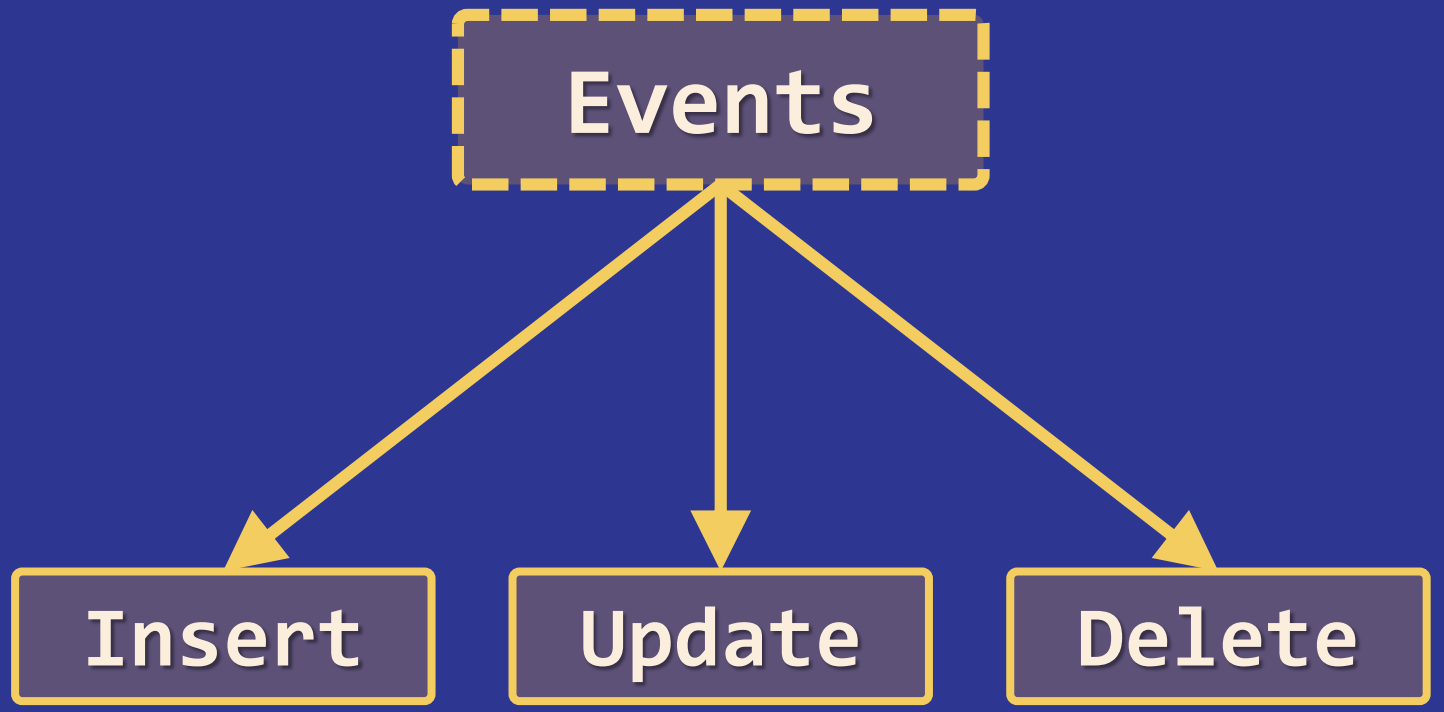
Triggers

What Are Triggers?

- Triggers are very much like stored procedures.
  - Called in case of specific event
- We do not call triggers explicitly
  - Triggers are attached to a table.
  - Triggers are fired when a certain SQL statement is executed against the contents of the table.
  - Syntax:
    - **AFTER INSERT/UPDATE/DELETE**
    - **INSTEAD OF INSERT/UPDATE/DELETE**

Events

- There are three different events that can be applied within a trigger:



Creating DML Triggers

- DML triggers are executed when DML events occur in tables or views.
- These DML events include the `INSERT`, `UPDATE`, and `DELETE` statements.
- DML triggers enforce referential integrity by cascading changes to related tables when a row is modified
- DML triggers are of three main types namely, `INSERT` trigger, `UPDATE` trigger, and `DELETE` trigger

Introduction to Inserted and Deleted Tables

- **Inserted Table**
  - Contains copies of records that are modified with the `INSERT` and `UPDATE` operations on the trigger table.
  - The `INSERT` and `UPDATE` operations insert new records into the Inserted and Trigger tables.
- **Deleted Table**
  - Contains copies of records that are modified with the `DELETE` and `UPDATE` operations on the trigger table.
  - These operations delete the records from the trigger table and insert them in the Deleted table

Insert Triggers

- Are executed when a new record is inserted in a table
- Ensure that the value being entered conforms to the constraints defined on that table.
- Save a copy of that record in the Inserted table and checks whether the new value in the Inserted table conforms to the specified constraints.
- Insert the row in the trigger table if the record is valid otherwise, it displays an error message.

Insert Triggers

Syntax:

```
CREATE TRIGGER [schema_name.] trigger_name
ON [schema_name.] table_name [WITH ENCRYPTION]
{FOR INSERT} AS
[IF UPDATE (column_name)...]
[{AND | OR} UPDATE (column_name)...]
<sql_statements>
```

where,

schema_name: specifies the name of the schema to which the table/trigger belongs.
trigger_name: specifies the name of the trigger.
table_name: specifies the table on which the DML trigger is created.
WITH ENCRYPTION: encrypts the text of the CREATE TRIGGER statement.
FOR: specifies that the DML trigger executes after the modification operations are complete.
INSERT: specifies that this DML trigger will be invoked after the update operations.
UPDATE: Returns a Boolean value that indicates whether an INSERT or UPDATE attempt was made on a specified column.

Insert Triggers

Example:

```
CREATE TRIGGER CheckWithdrawal_Amount
ON Account_Transactions
FOR INSERT
AS
   IF (SELECT Withdrawal From inserted) > 80000
   BEGIN
        PRINT 'Withdrawal amount cannot exceed 80000'
        ROLLBACK TRANSACTION
   END
```

```
INSERT INTO Account_Transactions
(TransactionID, EmployeeID, CustomerID,TransactionTypeID,TransactionDate,
TransactionNumber,Deposit,Withdrawal)
VALUES
(1008,'E08','C08','T08','05/02/12','TN08', 300000,90000)
```

Withdrawal amount cannot exceed 80000.

Update Triggers

- Are executed when a record is updated in a table
- Copies the original record in the Deleted table and the new record into the Inserted table.
- Evaluates the new record to determine if the values conform to the constraints specified in the trigger table.
- Copies the record from the Inserted table to the trigger table provided the record is valid
- Is created using the `UPDATE` keyword in the `CREATE TRIGGER` and `ALTER TRIGGER` statements.

Update Triggers

Syntax:

```
CREATE TRIGGER [schema_name.] trigger_name
ON [schema_name.] table_name [WITH ENCRYPTION]
{FOR UPDATE} AS
[IF UPDATE (column_name)...]
[{AND | OR} UPDATE (column_name)...]
<sql_statements>
```

where,

schema_name: specifies the name of the schema to which the table/trigger belongs.
trigger_name: specifies the name of the trigger.
table_name: specifies the table on which the DML trigger is created.
WITH ENCRYPTION: encrypts the text of the CREATE TRIGGER statement.
FOR: specifies that the DML trigger executes after the modification operations are complete.
INSERT: specifies that this DML trigger will be invoked after the update operations.
UPDATE: Returns a Boolean value that indicates whether an INSERT or UPDATE attempt was made on a specified column.

UpdateTriggers

Example:

```
CREATE TRIGGER CheckBirthDate
ON EmployeeDetails
FOR UPDATE
AS
   IF (SELECT BirthDate From inserted) > getDate()
   BEGIN
      PRINT 'Date of birth cannot be greater than today's date'
      ROLLBACK TRAN
   END
```

```
UPDATE EmployeeDetails
SET BirthDate='2015/06/02'
WHERE EmployeeID='E06')
```

```
Date of birth cannot be greater than today's date.
```

Delete Triggers

- The record is deleted from the trigger table and inserted in the Deleted table
- It is checked for constraints against deletion.
- If there is a constraint on the record to prevent deletion, the DELETE trigger displays an error message.
- The deleted record stored in the Deleted table is copied back to the trigger table
- Is created using the `DELETE` keyword in the `CREATE TRIGGER` statement.

Delete Triggers

Syntax:

```
CREATE TRIGGER <trigger_name>
ON <table_name>
[WITH ENCRYPTION]
FOR DELETE
AS <sql_statement>
```

where,

DELETE: specifies that this DML trigger will be invoked by delete operations.

Delete Triggers

Example:

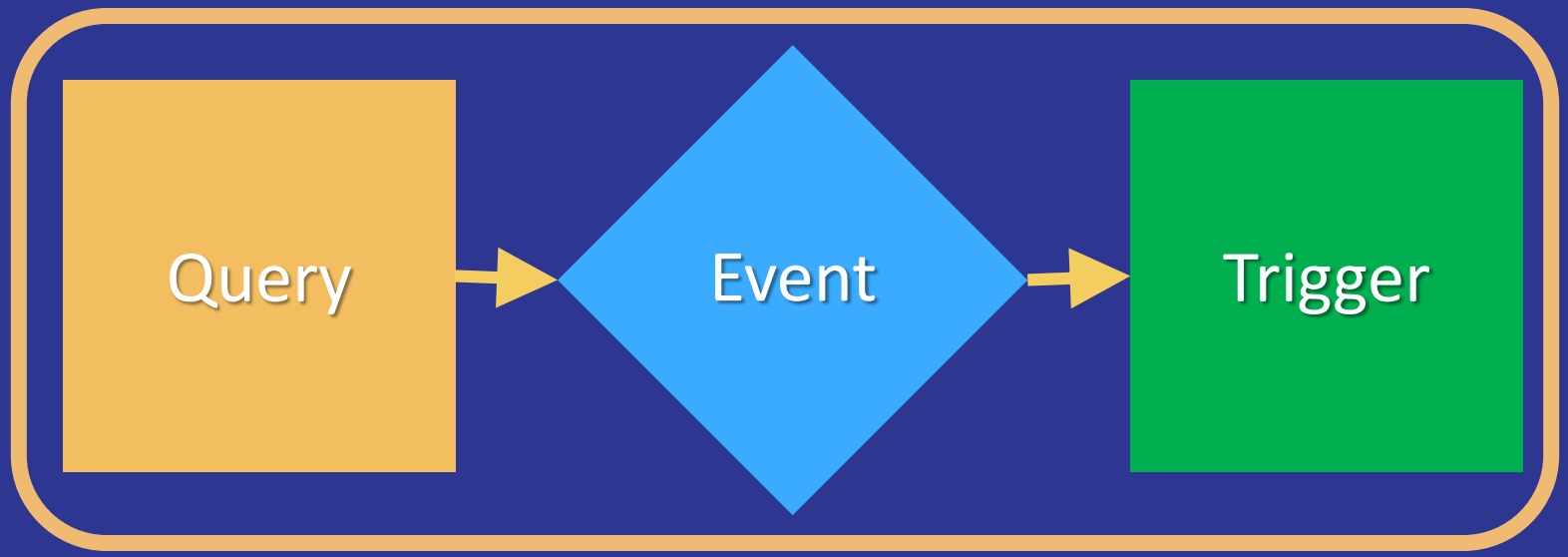
```
CREATE TRIGGER CheckTransactions
ON Account_Transactions
FOR DELETE
AS
IF 'T01' IN (SELECT TransactionID FROM deleted)
BEGIN
PRINT 'Users cannot delete the transactions.'
ROLLBACK TRANSACTION
END
```

```
DELETE FROM Account_Transactions
WHERE Deposit= 50000
```

```
Users cannot delete the transactions.
```

After Trigger

- **AFTER** Trigger is executed right after an event is fired



After Triggers

- Defined by the keyword **AFTER**

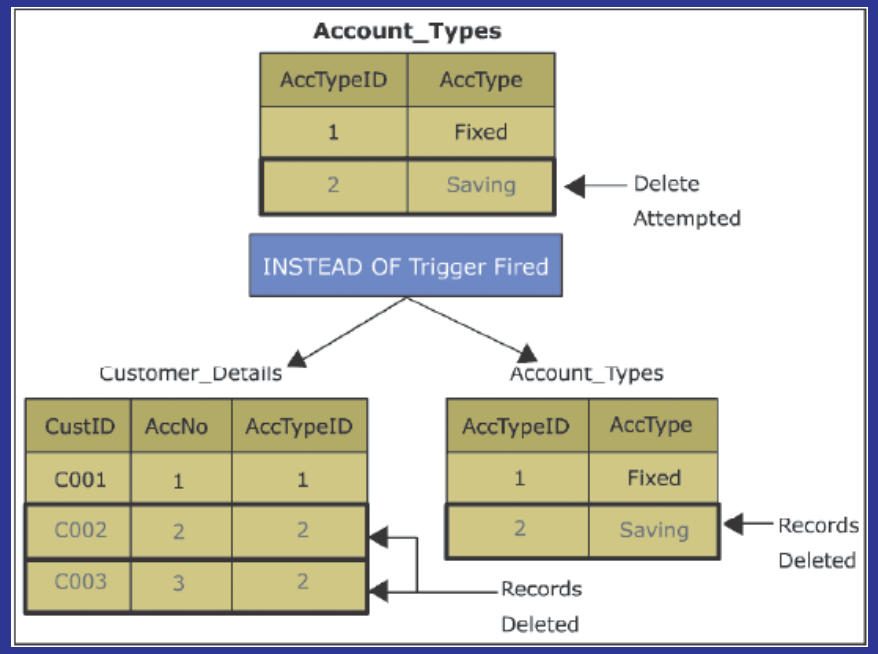
```
CREATE TRIGGER tr_TownsUpdate ON Towns AFTER UPDATE
AS
   IF (EXISTS(
         SELECT * FROM inserted
         WHERE Name IS NULL OR LEN(Name) = 0))
   BEGIN
     RAISERROR('Town name cannot be empty.', 16, 1)
     ROLLBACK
     RETURN
   END

UPDATE Towns SET Name='' WHERE TownId=1
```

Causes an error

Instead of Trigger

- Is executed in place of the `INSERT`, `UPDATE`, or `DELETE` operations
- Can be created on tables as well as views and there can be only one `INSTEAD OF` trigger defined for each `INSERT`, `UPDATE`, and `DELETE` operation



Instead of Trigger

Syntax:

```
CREATE TRIGGER <trigger_name>
ON { <table_name> | <view_name> }
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS <sql_statement>
```

where,

view_name: specifies the view on which the DML trigger is created.
INSTEAD OF: specifies that the DML trigger executes in place of the modification operations. These triggers are not defined on updatable views using WITH CHECK OPTION.

Instead of Trigger

Example:

```
CREATE TRIGGER Delete_AccType
ON Account_Transactions
INSTEAD OF DELETE
AS
BEGIN
    DELETE FROM EmployeeDetails WHERE EmployeeID IN
    (SELECT TransactionTypeID FROM deleted)
    DELETE FROM Account_Transactions WHERE TransactionTypeID
   IN (SELECT TransactionTypeID FROM deleted)
END
```

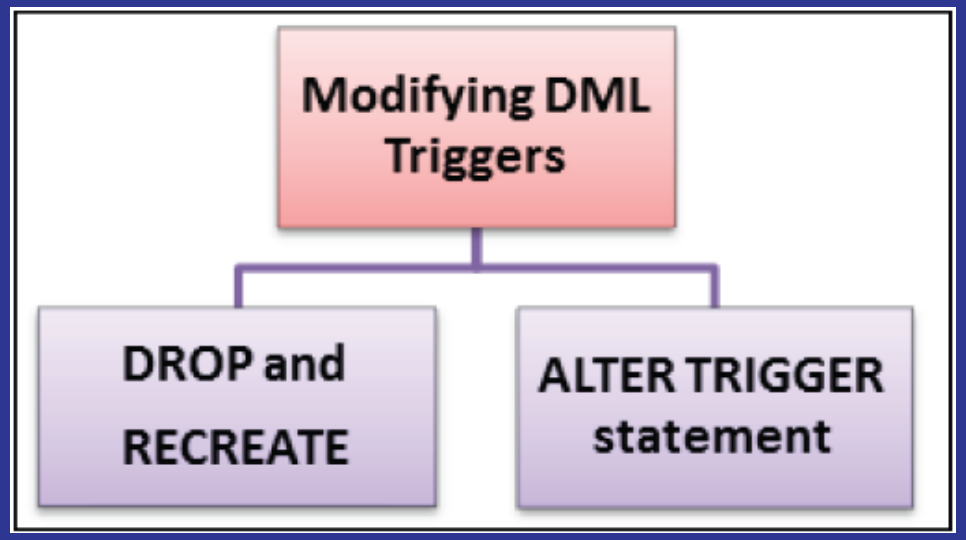
Instead Of Triggers

- Defined by using **INSTEAD OF**

```
CREATE TABLE Accounts(
  Username varchar(10) NOT NULL PRIMARY KEY,
  [Password] varchar(20) NOT NULL,
  Active char(1) NOT NULL DEFAULT 'Y’
)
CREATE TRIGGER tr_AccountsDelete ON Accounts
INSTEAD OF DELETE
AS
UPDATE a SET Active = 'N'
  FROM Accounts AS a JOIN DELETED d
    ON d.Username = a.Username
 WHERE a.Active = 'Y'
```

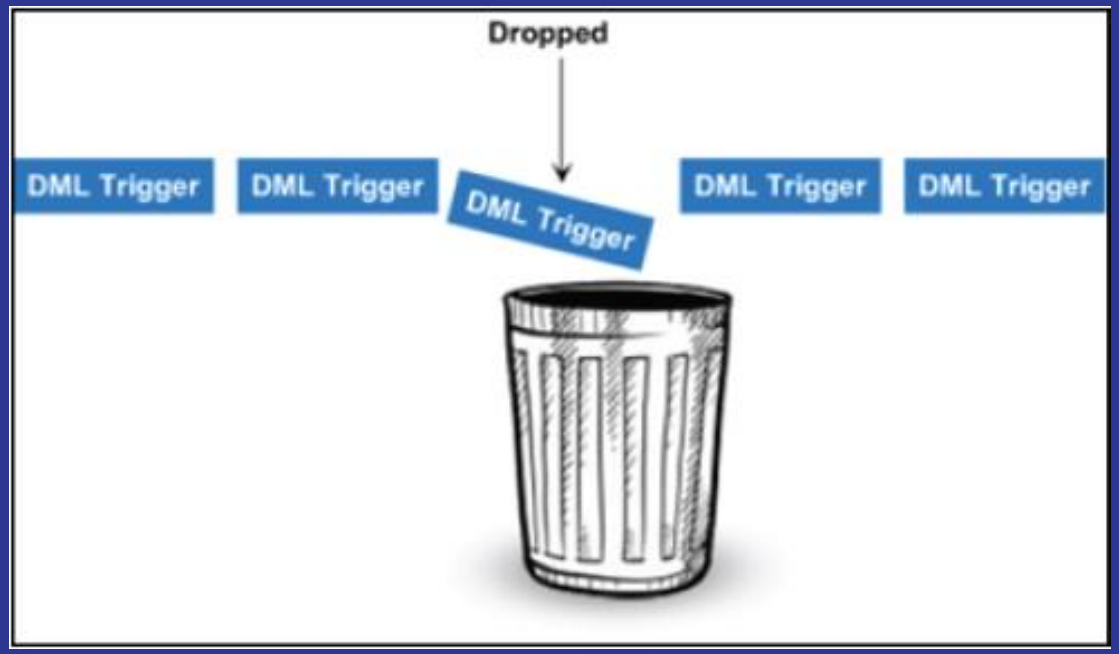
Modify Triggers

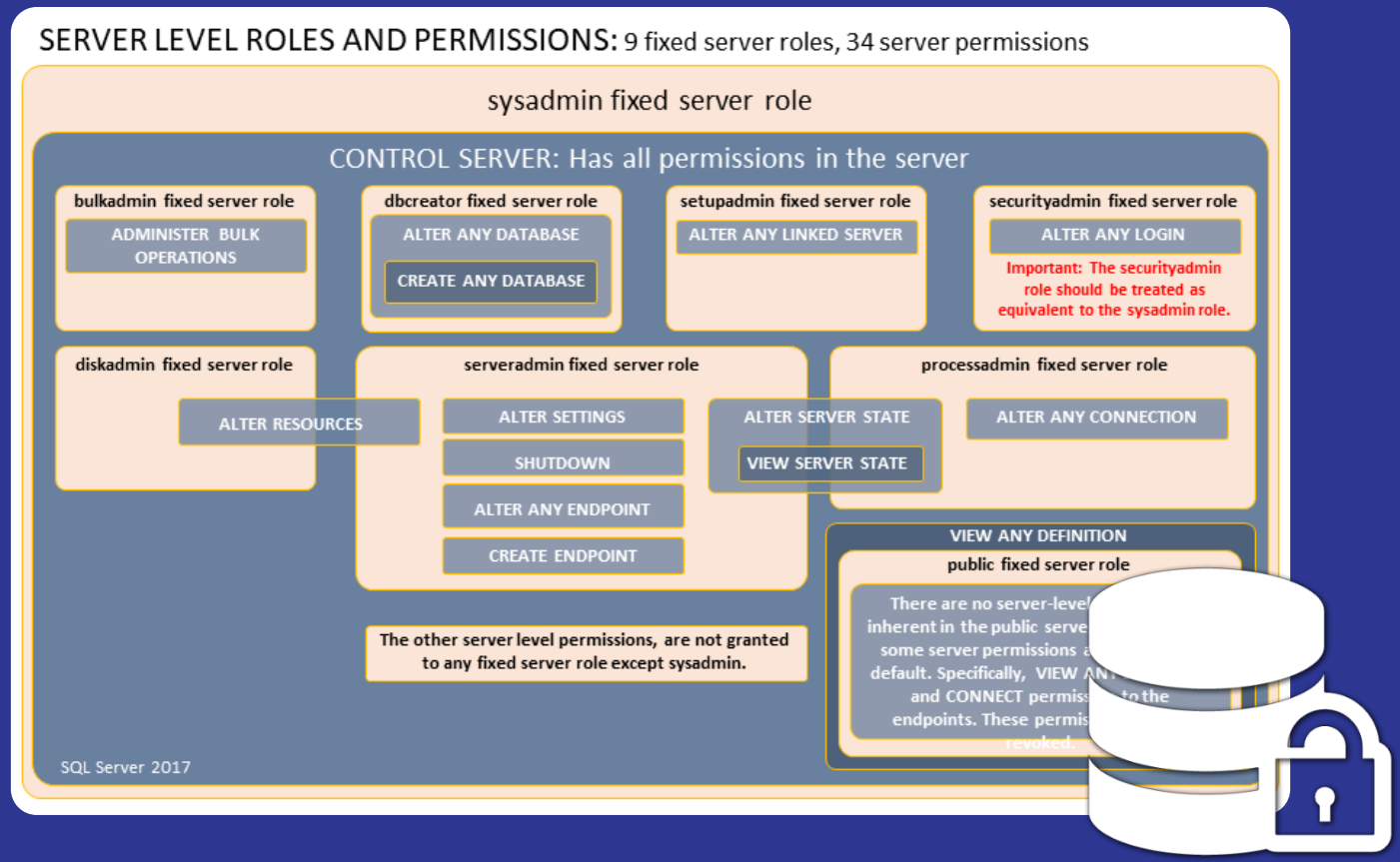
- User can modify any of these parameters for a DML trigger in any one of two ways
  - Drop and re-create the trigger with the new parameters.
  - Change the parameters using the ALTER TRIGGER statement.



Drop Triggers

- Trigger can be dropped using the `DROP TRIGGER` statement
- Multiple triggers can also be dropped using a single drop trigger statement.





Database Security

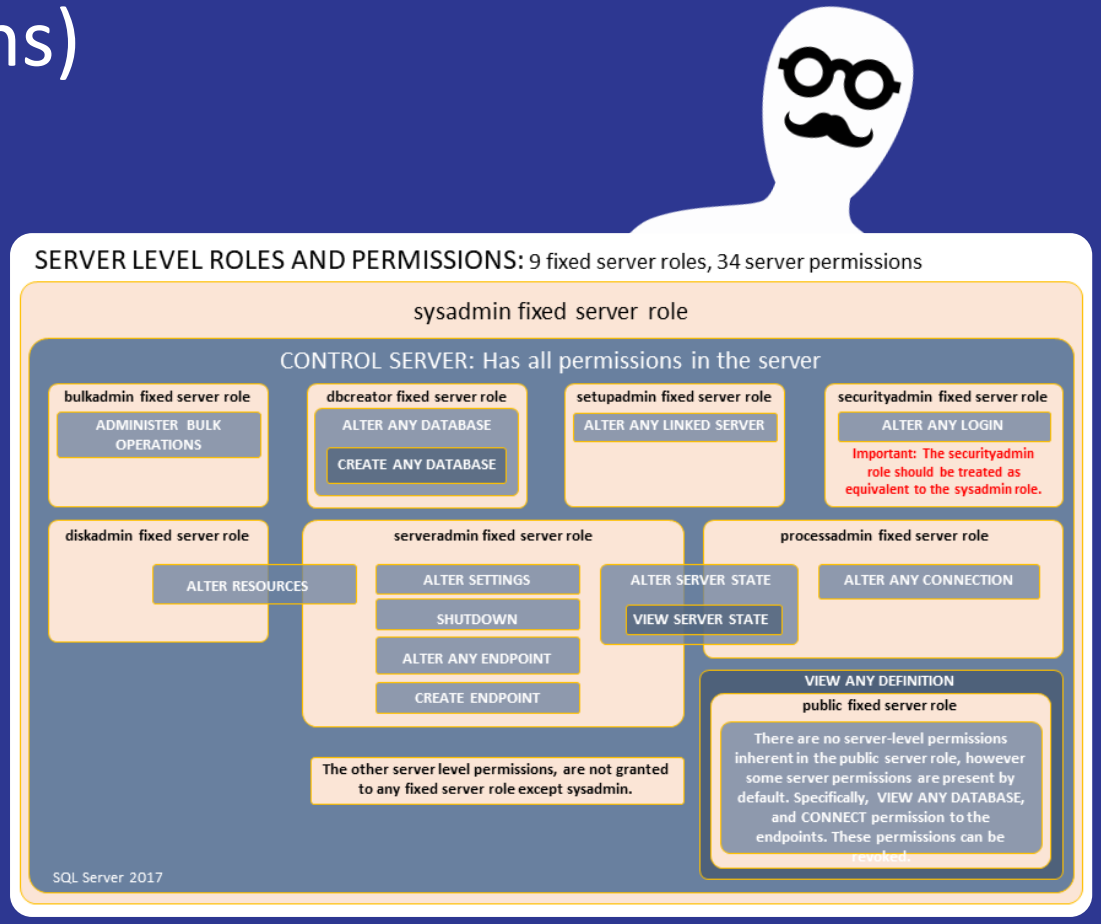
Fixed Server Roles, Fixed Database Roles

Database Security: SQL Server

- SQL Server has two layers of database security:
  - Fixed Server Roles
    - **sysadmin**, **bulkadmin**, **dbcreator**, **securityadmin**
  - Fixed Database Roles
    - **db_owner**, **db_securityadmin**, **db_accessadmin**
    - **db_backupoperator**, **db_ddladmin**
    - **db_datareader/db_datawriter**

Custom Roles

- SQL Server lets us create custom roles
  - Collection of privileges (permissions)
- Fine control over permissions
  - Can use one role for multiple users (groups)
- Makes auditing operations easier



Custom Role Permissions

- **CONTROL SERVER/DATABASE**
  - Gives all permissions on the server/database
- **TAKE OWNERSHIP**
  - Enables the grantee to take ownership of a securable
- **VIEW CHANGE TRACKING**
  - Manage change tracking on schemas and tables
- **VIEW DEFINITION**
  - Enables the grantee to access metadata
- **EXECUTE**
  - To run procedures, scalar and aggregate functions

Custom Role Permissions (2)

- **ALTER**
  - Lets a role create, alter, and drop objects from the schema
- **REFERENCES**
  - Lets the role create FOREIGN KEY constraints
- **SELECT/DELETE/INSERT/UPDATE**
  - Grant access to CRUD Operations
  - Can be granted on database, schema and object level
    - Individual tables, views and columns can be targeted

Summary

- Functions allow for complex calculations
  - Usually return a scalar value

```
CREATE FUNCTION f_ProcedureName
RETURNS …
AS
…
```

- Stored Procedures allow us to save time by
  - Shortening code
  - Simplifying complex tasks

```
CREATE PROC usp_ProcedureName AS …
```

Summary

- Transactions give our operations stability
  - Operation Integrity
  - Solving the concurrent operation problem
  - The ACID model is implemented in most RDBMS
- Triggers apply a given behavior when a condition is hit
  - Gives us temporary **`INSERTED`** and **`DELETED`** tables
- Security in SQL Server can be finely controlled
  - Using fixed server roles and fixed database roles
- Custom roles control permissions even more finely