DAY 6

SQL, Database Constraints, Rules, Create DB (Day 6)

In SQL Server, the concept of database storage is divided into **physical** and **logical** representations.

1. Physical Representation

The physical representation refers to how data is stored on the disk in files that make up the database. There are two primary types of files:

- **MDF (Primary Data File)**:
  - Contains the main data of the database (tables, indexes, etc.).
  - There is only one primary data file for each database.
  - Default extension is `.mdf`.
- **NDF (Secondary Data File)**:
  - Optional files used if you want to split the data across multiple files or disks.
  - The database can have multiple NDF files.
  - Default extension is `.ndf`.

- **LDF (Transaction Log File)**:
  - Stores a record of all transactions and modifications made to the database.
  - This is essential for database recovery in case of failure.
  - Default extension is `.ldf`.

2. Logical Representation

The logical structure of a database refers to how the data is organized internally in the database, regardless of how it's physically stored on disk.

- **Filegroup**:
  - A filegroup is a logical container for database files. It is a grouping of one or more data files (MDF and NDF) that stores data in the database.
  - Every database has a **primary filegroup**, which includes the primary data file (MDF) by default.
  - You can create additional filegroups to organize data into different locations (e.g., on different disks) for performance reasons or data separation.
  - When creating tables or indexes, you can specify which filegroup they should be stored in.
- **Primary Filegroup**:
  - Automatically created when the database is created.
  - Contains the main database objects and by default includes the primary data file (MDF).
  - If no other filegroup is specified, all objects are created in the primary filegroup.

Example of a Logical to Physical Mapping:

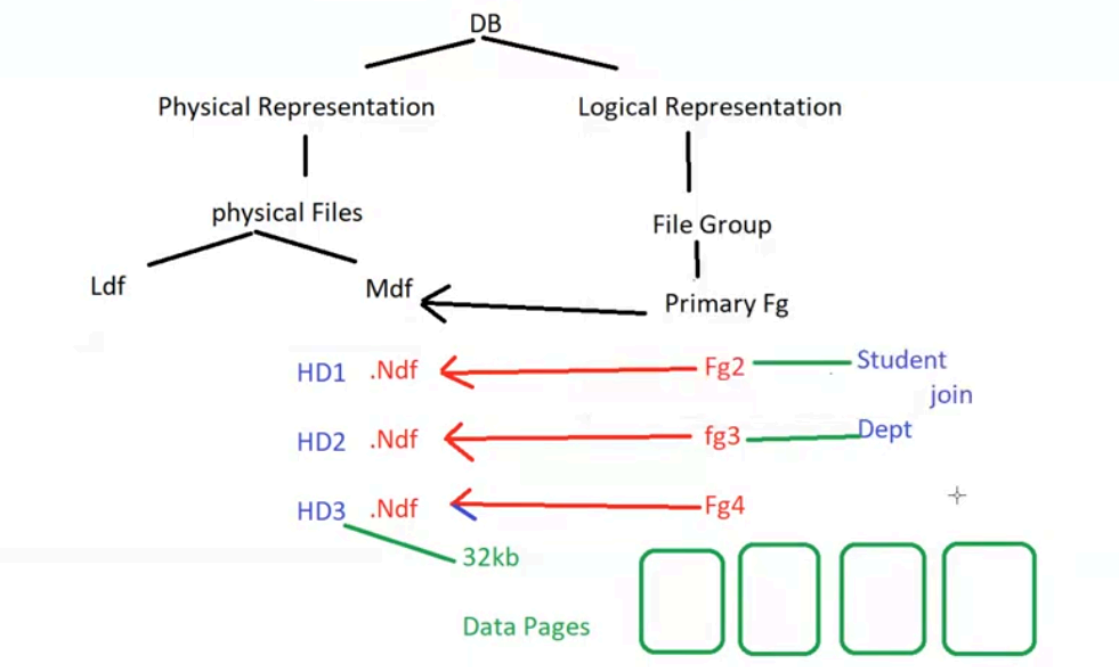
1. **Logical Representation**: A table is created in the database, and the data is logically organized within a specific filegroup.
2. **Physical Representation**: The data for that table is then written to one or more physical data files (MDF or NDF) on the disk, depending on the filegroup configuration.

SQL Server File Structure Example:

```
-- Example of specifying a table to use a different filegroup
CREATE DATABASE MyDatabase
ON PRIMARY
( NAME = MyPrimaryFile,
  FILENAME = 'C:\\MyData\\MyPrimaryFile.mdf',
  SIZE = 5MB,
  MAXSIZE = 100MB,
  FILEGROWTH = 5MB )
LOG ON
( NAME = MyLogFile,
  FILENAME = 'C:\\MyData\\MyLogFile.ldf',
  SIZE = 1MB,
  MAXSIZE = 50MB,
  FILEGROWTH = 1MB );
```

Data pages

- **Data pages** are 8 KB blocks that store the actual data for tables and indexes.
- SQL Server uses a variety of page types to manage different kinds of data, including data pages, index pages, and pages for large objects.
- Pages are grouped into **extents**, and SQL Server uses allocation maps to track which extents and pages are in use or available.
- Pages are central to how SQL Server reads, writes, and manages data storage.



The process you are describing involves creating a SQL Server database with multiple **filegroups** and distributing the data files across different disks. This method can improve performance by allowing parallel disk I/O operations and better management of large data sets.

1. Create the Database with Multiple Filegroups

- **Step 1**: In SQL Server Management Studio (SSMS), right-click on the **Databases** node and select **New Database**.
- **Step 2**: In the **New Database** window, enter the name of the database.
- **Step 3**: On the left side, select **Filegroups**.
- **Step 4**: Click **Add** to create new filegroups. You can add multiple filegroups depending on your needs.
    - Each **filegroup** represents a logical unit for grouping data files.

2. Assign Files to Filegroups

- **Step 5**: Go back to the **General** tab on the left side.
- **Step 6**: In the **Database files** section, click **Add** to add multiple data files.
    - Each data file can be assigned to a different filegroup by changing the **Filegroup** column.

    - For example, you could create a file for `PRIMARY`, another for `SECONDARY`, etc.
- **Step 7**: To optimize performance, specify a different **path** for each data file on separate physical drives (if available). This allows SQL Server to perform parallel I/O operations on different disks.

3. Benefit of Parallel Disk I/O

- By placing data files on different physical disks, you increase the **parallelism** in reading and writing data. SQL Server can access multiple files in parallel, which improves performance, especially during **intensive operations** like joins, large queries, and index rebuilds.
- This setup is especially useful for large databases with heavy read/write operations.

4. Create Tables and Assign to Filegroups

- **Step 8**: After the database is created with multiple filegroups, you can assign tables to specific filegroups.
- **Step 9**: Right-click the database you just created and select **New Table**.
- **Step 10**: Define the columns of the table.
- **Step 11**: In the **Properties** window (usually on the right), scroll down to the **Storage** section.
- **Step 12**: For the **FileGroup or Partition Scheme** (Regular Data Space Specification), choose the filegroup where you want the table to be stored from the **Filegroup** dropdown.
    - This will ensure that the table's data is stored in the specified filegroup, which can be beneficial for performance.

5. Performance Benefits of Filegroups

- **Parallel Disk Access**: Storing data across different disks allows SQL Server to perform **parallel reads and writes**, speeding up queries and operations.
- **Improved Joins and Queries**: When performing joins on tables that are stored in different filegroups (on separate physical drives), the server can retrieve data faster.

- **Scalability**: You can manage large databases more easily by distributing the load across different filegroups and drives.
- **Backup and Restore Efficiency**: You can back up and restore individual filegroups rather than the entire database, making these processes faster for large databases.

By following this strategy, you create a more efficient database infrastructure, especially for environments with high data traffic and performance requirements.

To make a column **driven (calculated) and stored in the hard disk** in SQL Server, you need to use a **computed column** with the property **persisted**.

1. **Create a Computed Column**:
   - When defining your table, add a computed column that contains an expression based on other columns. For example, if you have `Salary` and `Overtime`, you can create a `NetSalary` column as `Salary + Overtime`.
2. **Make it Persisted**:
   - To store the computed value on disk (which improves performance by avoiding recalculations), mark the column as **persisted**. This ensures that the computed value is stored physically in the table, not just calculated on the fly during queries.

Example SQL Query:

```
CREATE TABLE Employees (
EmployeeID INT PRIMARY KEY,
Salary DECIMAL(10, 2),
Overtime DECIMAL(10, 2),
NetSalary AS (ISNULL(Salary, 0) + ISNULL(Overtime, 0)) PERSISTED );
-- Computed column with ISNULL to handle nulls
-- Computed column stored in the database
```

- In this example, the `NetSalary` column is automatically calculated as `Salary + Overtime`, and because it's marked as **PERSISTED**, the result is stored on the disk, reducing the need for recalculations each time it's queried.
- The `ISNULL()` function is commonly used in SQL to handle null values in a formula. If any value in your formula is `NULL`, the result of the formula could

also be `NULL`, which is often undesirable. Using `ISNULL()`, you can replace the `NULL` value with a default value to ensure that the formula works as expected.

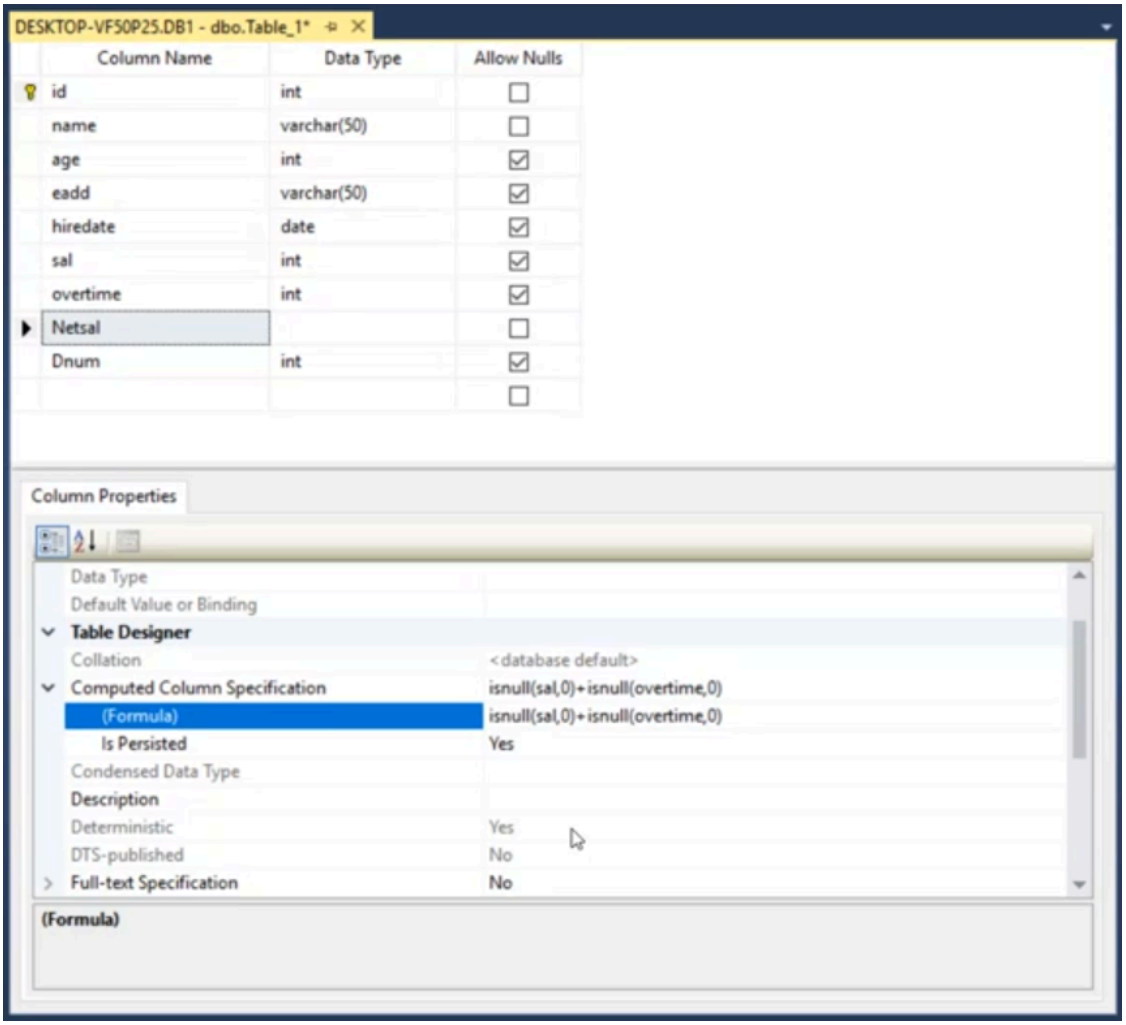
- `ISNULL(Salary, 0)`: If `Salary` is `NULL`, it will be replaced with `0`.
- `ISNULL(Overtime, 0)`: If `Overtime` is `NULL`, it will be replaced with `0`.

Steps in Table Designer (SQL Server Management Studio):

1. Open the **Column Properties** in the **Table Designer**.
2. Under **Computed Column Specification**, enter the formula for the computed column (e.g., `Salary + Overtime`).
3. Set the **Is Persisted** option to **Yes** to store the calculated value in the database.

Sparse :

- **Sparse columns** are designed for scenarios where most rows contain `NULL` values in the column.
- When a column is marked as **Sparse**, `NULL` values do not take up any physical storage.
- **Non-NULL** values in a sparse column may require more storage than they would in a non-sparse column, so it's important to use this feature in situations where there are many `NULL` values.
- **Sparse columns** help reduce disk space usage when dealing with a large number of `NULL` values.
- If most rows contain non-`NULL` values, the sparse column may actually increase storage requirements for those non-`NULL` values, as sparse columns require more metadata to handle them efficiently.



Managing Relationships in SQL Server:

1. **Creating Database Relationships:**
   - In a database diagram, when you establish relationships between tables, you can specify how actions on the primary key affect the foreign key.
   - Right-click on the relationship line connecting the tables and select **Properties**.
2. **Delete Rule Options:**
   - **Set Null:** When you delete a row in the primary key table, the corresponding foreign key values in the referencing table are set to `NULL`.
   - **Cascade:** When you delete a row in the primary key table, all related rows in the foreign key table are also deleted automatically.

3. **Update Rule Options:**
    - **Cascade:** If you update the value of a primary key, the same value is updated in all related foreign keys.

Example:

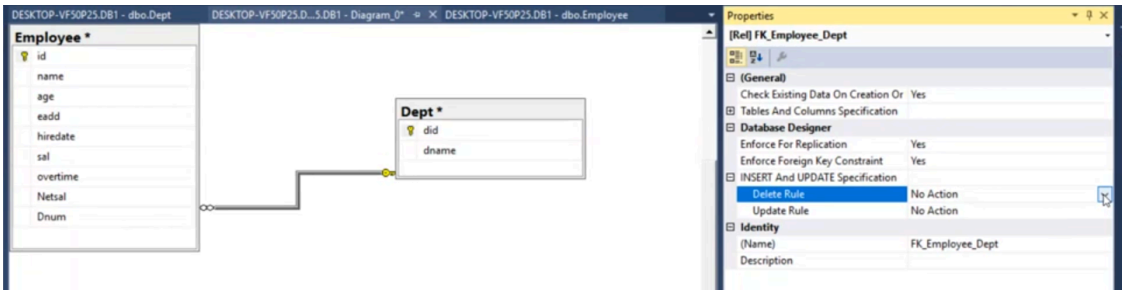
- Suppose you have two tables: `Orders` (with `OrderID` as PK) and `OrderDetails` (with `OrderID` as FK referencing `Orders`).

Setting Delete and Update Rules:

- To set up these rules:
    - **Delete Rule:** You can choose either **Set Null** or **Cascade**.
        - If you choose **Set Null**, deleting an order will set the `OrderID` in `OrderDetails` to `NULL` for the related entries.
        - If you choose **Cascade**, deleting an order will remove all associated entries in `OrderDetails`.
    - **Update Rule:** Choosing **Cascade** will ensure that if you change the `OrderID` in `Orders`, it will automatically update the `OrderID` in `OrderDetails`.

Additional Options:

- **Set Default:** This option allows you to set a default value for the foreign key attribute if the corresponding primary key is deleted. The default value should be valid and exist in the primary key table.



Schema in Database Management

In database management systems, a **schema** is a logical structure that helps organize and manage database objects. It consists of a collection of related

objects such as tables, views, procedures, and more. The schema is identified by a name and is prefixed to the object name, typically following the format:

SchemaName.ObjectName (e.g., `dbo.Employee`).

Reasons for Using Schemas

1. **Unique Object Names:**
   - Schemas allow for the creation of multiple objects with the same name as long as they belong to different schemas. For instance, you can have `dbo.Employee` and `hr.Employee` in the same database without conflict.
2. **Logical Grouping:**
   - Schemas enable logical grouping of related objects. This organization helps manage and categorize tables and other objects that serve similar purposes or belong to the same business area, making it easier to maintain the database structure.
3. **Permission Management:**
   - Managing permissions at the schema level simplifies security administration. Instead of granting permissions on each individual table or object, you can grant permissions to the entire schema. This allows users to access all objects within a schema without needing separate permissions for each table.

Example of Schema Usage

- To select all records from the `Employee` table in the `dbo` schema, you would use the following SQL command:

```
SELECT * FROM dbo.Employee;
```

This command specifies the schema (`dbo`) along with the table name (`Employee`), ensuring clarity about which object is being referenced and facilitating effective management within the database.

Schemas in SQL Server

1. **Create a Schema:**
   - To create a schema named `HR` in your current database:

- Open your SQL Server Management Studio (SSMS).
- Navigate to the database you are currently using.
- Expand the **Security** folder.
- Right-click on **Schemas** and select **New Schema** to create a new schema.

2. **Transfer a Table to the New Schema:**
   - To transfer an existing table (e.g., `table2`) from its current schema to the `HR` schema, use the following SQL command:

```
ALTER SCHEMA HR TRANSFER dbo.table2;
```

   - Replace `dbo.table2` with the correct schema and table name if it's different.

3. **Creating a Table with the Same Name:**
   - After transferring `table2` to the `HR` schema, you can create another table with the same name in the default schema (e.g., `dbo`):

```
CREATE TABLE dbo.table2 (
    id INT,
    name VARCHAR(50)
);
```

   - This allows you to have two tables with the same name but in different schemas.

4. **Accessing the Table:**
   - To access the transferred `table2` in the `HR` schema, you must use the fully qualified name:

```
SELECT * FROM HR.table2;
```

   - You cannot simply use `SELECT * FROM table2` because this will refer to the table in the default schema (e.g., `dbo`).

5. **Additional Notes:**

- The schema can contain various objects, including tables, views, and functions.
- Transferring objects between schemas can help organize your database structure and manage permissions more effectively.

User Authentication and Permissions in SQL Server

1. Set Authentication Mode

- Right-click on the SQL Server instance in Object Explorer and select **Properties**.
- Navigate to the **Security** page.
- Choose **SQL Server and Windows Authentication mode**.
- Click **OK** to save your changes.

2. Restart the Server

- After changing the authentication mode, restart the SQL Server instance for the changes to take effect.

3. Create a New Login

- Expand the database folder in your SQL Server instance.
- Go to **Security**, right-click on **Logins**, and select **New Login**.
- Enter the **Login name ex: Amr, Password: amr123**.
- Choose **SQL Server authentication** and set a password.
- Uncheck the box for **Enforce password policy** if you want to disable password complexity requirements.

4. User Creation in Database

- Open the database you want to assign the user to (e.g., `CompanyDB`).
- Go to **Security**, right-click on **Users**, and select **New User**.

5. User Connection to Database

- To connect to the database, the user will need to:
  - Enter the server name where `Company DB` is located.

- Choose **SQL Server Authentication** and provide their login name and password.

6. Schema Permissions

- To connect the user to a specific schema (which contains tables):
  - Open the schema you want to grant access to within the main server.
  - Double-click on the schema (e.g., `HR`) to open its properties.
  - Go to the **Permissions** page, click **Search**, and find the newly created user. Click **OK**.
  - At the bottom, you can grant or deny permissions (like `DELETE`, `UPDATE`, etc.) to this user.

7. Granting Permissions

- By default, users cannot access anything in the schema. You can grant permissions by selecting the **Grant** column. This allows you to specify what actions the user can perform.
- If you want to deny certain permissions, you can do so from the **Deny** column.

Synonyms in SQL Server

A synonym is an alternative name for a database object, such as a table, view, or schema. Using synonyms can simplify your queries and provide a level of abstraction, making it easier to manage database changes.

Creating and Using Synonyms

1. **Creating a Synonym**:
   After creating a schema (e.g., `HR`), you can create a synonym for a table within that schema to simplify its reference. For example, to create a synonym for `HR.table2`, you can use the following SQL command:

   ```
   CREATE SYNONYM HRT FOR HR.table2; --HRT = HR Table
   ```

2. **Selecting Data from a Synonym**:
   Once the synonym is created, you can easily select data from the table

using the synonym:

```
SELECT * FROM HRT;
```

3. **Benefits of Using Synonyms**:
   - If you need to rename the original table (e.g., `HR.table2`), you won't have to update all the views or queries that reference the original table. Instead, you can simply alter the synonym to point to the new table name.
   - This helps maintain consistency and reduces the effort required to make changes across multiple database objects.

Example

If you decide to change the name of `HR.table2` to `HR.NewTable`, you can update the synonym using:

```
ALTER SYNONYM HRT FOR HR.NewTable;
```

This way, all existing queries using the synonym `HRT` will automatically reference the new table without needing further modifications.

Deleting Data in Databases or Tables

1. **Dropping a Table**:
   - **Command**: `DROP TABLE Employee;`
   - This command deletes both the data and metadata associated with the `Employee` table. Once executed, the entire table will disappear from the database.
2. **Deleting Rows**:
   - **Command**: `DELETE FROM Employee;` (with optional `WHERE` clause)
   - This command removes rows from the `Employee` table. You can specify conditions using a `WHERE` clause to delete specific records.
   - The `DELETE` operation is slower compared to `TRUNCATE`, and it allows for rollback if needed. If you delete rows with an identity attribute (e.g., IDs

like 1, 2, 3, 4, 5), the next inserted ID will continue from the last used value (e.g., 6, 7, 8, etc.).

3. **Truncating a Table**:
   - **Command**: `TRUNCATE TABLE Employee;`
   - This command quickly removes all rows from the `Employee` table without logging individual row deletions, making it faster than `DELETE`.
   - When you truncate a table with an identity attribute, the identity counter resets, starting again from the beginning (e.g., the next ID will be 1).

Clearing Data from a Specific Column

To clear data from a specific column without altering its structure, you can use the `UPDATE` statement:

- **Command**:

```
UPDATE Employee SET ColumnName = NULL;
```

- This command sets all values in the specified column (e.g., `ColumnName`) to `NULL`, effectively removing the data while keeping the column structure intact.
- Avoid using the `ALTER` statement for this purpose, as it modifies the column's structure (e.g., changing data types, adding constraints).

Summary of Differences

Feature	DROP	DELETE	TRUNCATE
Purpose	Remove table and data	Remove rows from table	Remove all rows from table
Effect on Structure	Deletes the entire table	Retains table structure	Retains table structure
Data Removal	Yes, all data	Yes, based on condition or all	Yes, all data
Rollback Capability	No (unless in a transaction)	Yes	No (unless in a transaction)
Speed	Fast (removes metadata)	Slower (logs each row deletion)	Fast (does not log individual rows)

Identity Reset	N/A	No	Yes
Use of WHERE clause	N/A	Yes	No
Referential Integrity	N/A	Maintained	Not maintained if FK constraints exist
Logging Behavior	Minimal logging	Logs each row deletion	Logs page deallocations
PK and FK Considerations	PK and FK removed	Maintains PK and FK relationships	Maintains PK and FK relationships but can't truncate if FK exists

🧹How to delete column without deleting its structure using sql?

To delete the data in a column without deleting its structure in SQL, you can use the `UPDATE` statement to set all the values in that column to `NULL` (or a specific default value). This approach retains the column itself in the table structure but removes all the data contained within it.

```
UPDATE table_name
SET column_name = NULL;

UPDATE Employees
SET Address = NULL;

UPDATE Employees
SET Address = 'N/A';
--If you want to set the column to a specific value instead of NULL, you can re
```

- **Update Statement**: This SQL command updates the specified column in the table.
- **Setting to NULL**: By setting the column to `NULL`, you effectively remove the existing data while keeping the column structure intact.
- **Maintaining Structure**: The column remains part of the table and can still be used in future queries.

Database Integrity (Constraints):

Database integrity ensures the accuracy, consistency, and correctness of the data stored in a database. It is maintained through constraints and rules on database objects like tables and views, and it's categorized into three main types:

1. Domain Integrity (Range of Values):

Domain integrity ensures that the data entered into a column falls within a valid range or adheres to certain rules.

- **Constraints**:
  - **Data Types**: Ensures that the data type (e.g., `int`, `varchar`, `date`) is respected.
  - **Default Values**: Provides a default value if none is specified (e.g., `Default('Cairo')`).
  - **Allow Null**: Specifies whether null values are permitted.
  - **Not Null**: Ensures that null values are not allowed.
  - **Check Constraints**: Defines specific conditions that data must meet (e.g., `CHECK (Age >= 18)`).
- **Database Objects**:
  - **Rules**: Used to enforce business rules.
  - **Triggers**: Used to automatically enforce rules or actions based on data changes.

2. Entity Integrity (Uniqueness):

Entity integrity ensures that each row in a table is uniquely identifiable, typically through a primary key.

- **Constraints**:
  - **Primary Key (PK) Constraints**: Ensures uniqueness and disallows null values. It uniquely identifies each row in a table.
  - **Unique Constraints**: Ensures that all values in a column or a group of columns are unique but allows one `NULL` value.
- **Database Objects**:

- **Index**: Helps maintain the uniqueness of data and improves query performance.
- **Triggers**: Can be used to ensure the uniqueness of rows beyond primary key and unique constraints.

3. Referential Integrity (Relationships):

Referential integrity ensures that relationships between tables remain consistent, typically enforced through foreign keys.

- **Constraints**:
  - **Foreign Key (FK) Constraints**: Ensures that a value in one table matches a value in another table (enforcing relationships).
- **Database Objects**:
  - **Triggers**: Used to maintain referential integrity by performing actions when a referenced table's data is modified.

Aspect	Domain Integrity (Range of Values)	Entity Integrity (Uniqueness)	Referential Integrity (Relationships)
Definition	Ensures that the values entered in a column are valid and within a specified range.	Ensures that each row in a table is uniquely identifiable.	Ensures that relationships between tables are valid and consistent.
Purpose	Validates data at the column level.	Guarantees that no duplicate rows exist in a table.	Enforces valid relationships between a primary key in one table and a foreign key in another.
Example	Enforcing that `Age` must be between 0 and 100, or `Salary` must be greater than 0.	Ensuring that each `EmployeeID` is unique in the `Employee` table.	Ensuring that an `Order` can only reference a valid `CustomerID` from the `Customer` table.
DB Constraints	- Data Types (e.g., `int`, `varchar`) - Default (`DEFAULT`) - Check	- Primary Key (`PRIMARY KEY`) - Unique (`UNIQUE`)	- Foreign Key (`FOREIGN KEY`)

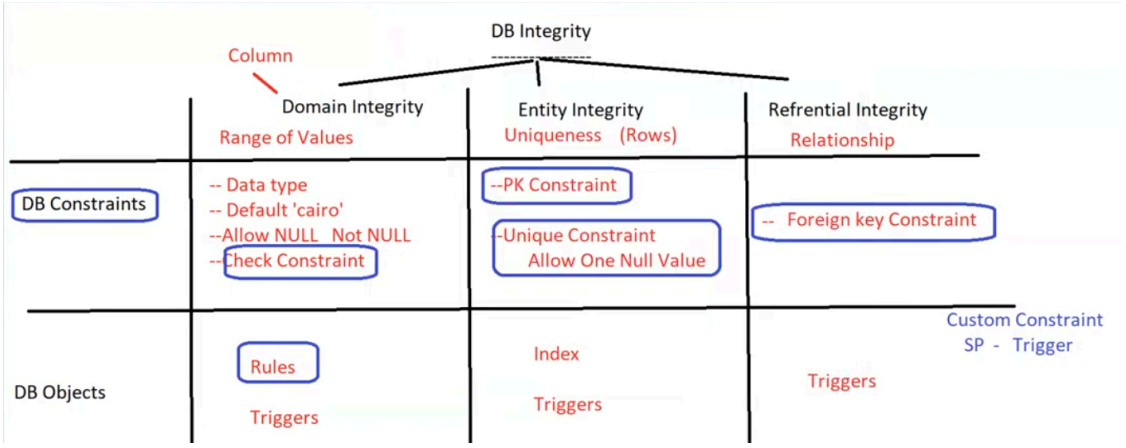
	(`CHECK`) - Not Null (`NOT NULL`)		
DB Objects	- Rules - Triggers	- Indexes - Triggers	- Triggers

Types of Constraints:

1. **Check Constraints**: Ensures that the value in a column meets a specific condition (e.g., `CHECK (Salary > 0)`).
2. **Primary Key (PK) Constraints**: Enforces uniqueness and not null on a column or combination of columns.
3. **Unique Constraints**: Ensures that all values in a column or a set of columns are unique.
4. **Foreign Key (FK) Constraints**: Enforces referential integrity between tables.
5. **Custom Constraints**: Defined based on custom rules or business logic, often through triggers or stored procedures.

Types DB Objects:

- **Tables** store data.
- **Views** are virtual tables.
- **Indexes** optimize performance.
- **Stored Procedures** and **Functions** encapsulate reusable SQL logic.
- **Triggers** automate actions based on data changes.
- **Schemas** organize database objects.
- **Sequences** generate unique numbers.
- **Synonyms** provide shortcuts.
- **Constraints** enforce data integrity.



⇒ Examples:

SQL Code Example

```
-- Create the Department table with a primary key
CREATE TABLE Department (
    Department_id INT PRIMARY KEY,
    Department_name VARCHAR(20)
);

-- Create the Employee table with various constraints
CREATE TABLE Employee (
    Employee_id INT IDENTITY(1, 2),
     -- Automatically incrementing employee ID
    Employee_name VARCHAR(20),
    Employee_address VARCHAR(20) DEFAULT 'assuit',
     -- Default address value
    hire_date DATE DEFAULT GETDATE(),
     -- Default hire date is the current date
    salary INT,
    overtime INT,
    net_salary AS (ISNULL(salary, 0) + ISNULL(overtime, 0)) PERSISTED, --
Calculated field
    BD DATE, -- Birth date
    age AS (YEAR(GETDATE()) - YEAR(BD)), -- Calculated field for age
    gender VARCHAR(1), -- Gender (M or F)
    hour_rate INT NOT NULL, -- Hourly rate, cannot be NULL
```

```
    department_id INT, -- Foreign key referencing Department

    -- Primary Key Constraint (Composite Key)
    CONSTRAINT PK_Employee PRIMARY KEY (Employee_id, Employee_nam
e),

    -- Unique Constraints
    CONSTRAINT UQ_Salary UNIQUE (salary),
    CONSTRAINT UQ_Overtime UNIQUE (overtime),

    -- Check Constraints
    CONSTRAINT CK_Salary CHECK (salary > 1200),
    CONSTRAINT CK_Address CHECK (Employee_address IN ('alex', 'cairo',
'sohag', 'assuit')),
    CONSTRAINT CK_Gender CHECK (gender IN ('M', 'F')),
    CONSTRAINT CK_Overtime CHECK (overtime BETWEEN 100 AND 500),

    -- Foreign Key Constraint with cascading delete and update rules
    CONSTRAINT FK_Department FOREIGN KEY (department_id)
        REFERENCES Department (Department_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

SQL Table Creation with Constraints

1. **Composite Primary Key**:
    - To create a composite primary key using multiple columns, you can define it with a constraint. The composite primary key ensures that the combination of specified columns is unique across the table.

```
CONSTRAINT PK_Employee PRIMARY KEY (Employee_id, Employee_na
me)
```

2. **Unique Constraints**:
    - To ensure that the values in a specific column are unique, you can use a unique constraint. For example, to ensure that salaries are unique

across the Employee table:

```
CONSTRAINT UQ_Salary UNIQUE (salary)
```

- You can also apply a unique constraint to a combination of columns. For example, ensuring that the combination of salary and overtime is unique:

```
CONSTRAINT UQ_Salary_Overtime UNIQUE (salary, overtime)
```

This means that no two rows can have the same values for both salary and overtime.

3. **Check Constraints**:
   - To enforce specific rules on the values in a column, you can use check constraints. You can add multiple checks to a column:

```
CONSTRAINT CK_Salary CHECK (salary > 1200)
```

   - If you have a default value for a column and also want to apply a check constraint, make sure the default value meets the check criteria:

```
CONSTRAINT CK_Address CHECK (Employee_address IN ('alex', 'cair
o', 'sohag', 'assuit'))
```

4. **Foreign Key Constraints**:
   - To establish a relationship between two tables, you can use a foreign key constraint. This links a column in one table to a primary key in another table. For example:

```
CONSTRAINT FK_Department FOREIGN KEY (department_id)
REFERENCES Department (Department_id)
```

   - You can also specify delete and update rules for foreign keys. For example, if a referenced department is deleted, set the foreign key to NULL, and if the primary key in the Department table is updated, cascade that change to the Employee table:

```
ON DELETE SET NULL
ON UPDATE CASCADE
```

Adding Relationships and Managing Constraints in SQL

1. **Visualizing Relationships with Diagrams**:
   - When you create a diagram that includes the `Employee` and `Department` tables, you will observe that the relationship defined by the foreign key is accurately represented. This visualization helps confirm that the tables are properly linked based on the constraints established.
2. **Adding Constraints After Table Creation**:
   - You can add constraints to existing tables using the `ALTER TABLE` statement. For example, to add a check constraint that ensures the `hour_rate` is greater than 100, you can execute the following SQL command:

```
ALTER TABLE Employee
ADD CONSTRAINT CK_HourRate CHECK (hour_rate > 100);
```

3. **Handling Conflicts with Existing Data**:
   - If you attempt to add a constraint and there are already existing records in the table that violate this constraint, an error will be triggered. This is because the database cannot enforce the constraint due to conflicting data. It's essential to clean or update the data to ensure compliance before adding new constraints.
4. **Dropping Constraints**:
   - If you need to remove a constraint that was previously added, you can do so with the following command:

```
ALTER TABLE Employee
DROP CONSTRAINT CK_HourRate;
```

Managing Rules, Defaults, and Data Types in SQL

1. **Adding Rules to Apply Constraints on New Data**:

- You can create rules in SQL to impose conditions on new data. For example, to ensure that a variable `xule` is greater than 1200, you can define a rule as follows:

```
CREATE RULE rule1 AS @x > 1200;
```

- To bind this rule to a column (e.g., `Dept_id` in the `Department` table), you would use:

```
EXEC sp_bindrule 'rule1', 'Department.Dept_id';
```

- If you need to drop the rule later, first unbind it from all columns:

```
EXEC sp_unbindrule 'Department.Dept_id';
```

- Then drop the rule:

```
DROP RULE rule1;
```

2. **Creating Shared Constraints**:
   - Rules can be shared between multiple tables. By using rules, you can ensure that the same constraint applies across different tables, enhancing consistency in your database.
3. **Creating Default Values**:
   - You can establish default values for a column in a table using a default object. For example, to create a default value of 'it', you would use:

```
CREATE DEFAULT d1 AS 'it';
EXEC sp_bindefault 'd1', 'Department.Dname';
```

- To unbind the default value from the column, use:

```
EXEC sp_unbindefault 'Department.Dname';
```

- Finally, to drop the default:

```
DROP DEFAULT d1;
```

4. **Creating a Data Type with Constraints and Default Values**:
   - You can create a user-defined data type that includes constraints and a default value. For example, to create a complex data type that enforces an integer to be greater than 1000 with a default value of 1200, follow these steps:

```
CREATE DEFAULT d2 AS 1200;
CREATE RULE r2 AS @x > 1000;
EXEC sp_addtype complexDT1, 'int';
EXEC sp_bindrule r2, 'complexDT1';
EXEC sp_bindefault d2, 'complexDT1';
```

   - You can then create a table that uses this complex data type:

```
CREATE TABLE test (
    id INT,
    salary complexDT1
);
```

5. **Managing Rules and Defaults**:
   - To find the rules and defaults you've used, navigate to the database (e.g., `CompanyDB`) and look under **Programmability** > **Rules** and **Defaults**.
6. **Key Considerations**:
   - You can have one rule on the same column.
   - If a column has both a rule and a constraint, the constraint takes precedence.

- **One Column with One Rule**: Each column can have only one associated rule that applies to new data. This rule can enforce a specific condition on the data being entered.
- **One Column with Many Constraints**: A single column can have multiple constraints to enforce different conditions, such as uniqueness, checks for valid values, and not allowing nulls.

Feature	Rules	Constraints
Definition	A rule is a condition applied to new data for a specific column.	A constraint is a rule enforced on data in one or more columns to maintain data integrity.
Applicability	Can be applied to a single column.	Can be applied to one or more columns.
Number of Associations	Only one rule can be associated with a column at a time.	Multiple constraints can be associated with a single column.
Types	Typically defines conditions (e.g., a variable must meet a certain condition).	Includes various types: primary key, foreign key, unique, check, and not null constraints.
Usage	Used primarily for defining conditions on new data entries.	Used to enforce rules on existing data and maintain data integrity throughout the database.
Error Handling	If the data does not meet the rule, it will reject the new entry.	If existing data violates a constraint when it is added or modified, an error will occur.
Creation	Created using the `CREATE RULE` statement and bound to a column using `sp_bindrule`.	Created using the `CREATE TABLE` or `ALTER TABLE` statements with `CONSTRAINT` clauses.
Example	`CREATE RULE r1 AS @x > 1000; EXEC sp_bindrule r1, 'TableName.ColumnName';`	`ALTER TABLE Employee ADD CONSTRAINT CK_Salary CHECK (Salary > 1200);`
Data Type Limitations	Can only apply to variable types, not directly on tables or columns.	Directly applies to table columns and can enforce type-specific rules.