Microsoft

SQL Server Relational Model

Relational Model in Database Systems

- The **Relational Model** is a foundational concept in database systems
  - It provides a logical structure for organizing and manipulating data using relations (commonly known as **tables**).
- **Relation (Table)**
  A relation is a two-dimensional table consisting of:
  - **Rows (Tuples)**: Each row represents a single record.
  - **Columns (Attributes)**: Each column represents a data field.

Customer		
CustomerID	FirstName	LastName
1	Dan	Drayton
2	Aisha	Witt
3	Rosie	Reeves

Attributes, or columns, can be of various different data types supported by the database server

In relational database concepts, a table is also called a relation. This terminology comes from the mathematical foundation of the relational model, which was developed by Edgar F. Codd.
In this model, a database is seen as a collection of relations.

Key Terminology
The relational model uses specific, set-theory-based terms that correspond to the more common terms used in SQL and database management systems (DBMS).

Relation: The formal term for a table. It's a set of tuples that share the same attributes.
Tuple: The formal term for a row or record. It represents a single entity in the table.
Attribute: The formal term for a column or field. It represents a specific characteristic or property of the entity.

The term relation emphasizes the mathematical theory behind the model, which treats tables as a set of unique tuples (rows), with no duplicates, and where the order of rows and columns doesn't matter. In practice, when you're working with a database, you'll most often hear the term table.

Column Data Types

- In **SQL Server**, **column types** (also called **data types**) define the kind of data that can be stored in each column of a table.
- Choosing the right data type is essential for performance, storage efficiency, and data integrity.

Numeric Types

Used for numbers:

- `INT`: Whole numbers (e.g., 1, 100)
- `BIGINT`: Very large integers
- `DECIMAL(p,s)` / `NUMERIC(p,s)`: Fixed precision numbers (e.g., 123.45)
- `FLOAT` / `REAL`: Approximate decimal numbers

String Types

Used for text:

- `CHAR(n)`: Fixed-length text
- `VARCHAR(n)`: Variable-length text
- `TEXT`: Large text (deprecated)

In **SQL Server**, both CHAR and VARCHAR are used to store **text data**, but they behave differently in terms of **storage and performance**.

CHAR (Fixed-Length)
Definition: Stores a fixed number of characters.
Padding: If the data is shorter than the defined length, SQL Server **pads** it with spaces.
Use Case: Best for fields with **consistent length**, like country codes (PK, US, UK) or gender (M, F).

```
CREATE TABLE Countries (
   CountryCode CHAR(2)
);
```

If you insert 'M', it stores 'M ' (with a space).

VARCHAR (Variable-Length)
Definition: Stores a variable number of characters up to a defined limit.
No Padding: Only uses space for the actual characters.
Use Case: Best for fields with **varying lengths**, like names, addresses, or descriptions.

If you insert 'Ali', it stores exactly 'Ali'—no extra spaces.

Column Data Types (cont…)

Unicode String Types

Used for multilingual text:

- `NCHAR(n)`, `NVARCHAR(n)`: Unicode versions of `CHAR` and `VARCHAR`

Other Types

- `BIT`: Boolean (0 or 1)
- `UNIQUEIDENTIFIER`: Globally unique ID (GUID)
- `XML`: XML data
- JSON: Stored as `NVARCHAR`

Date and Time Types

Used for storing dates and times:

- `DATE`: Only date
- `TIME`: Only time
- `DATETIME`, `SMALLDATETIME`: Date + time
- `DATETIME2`: More precise datetime
- `DATETIMEOFFSET`: Includes time zone

Binary Types

Used for files or binary data:

- `BINARY(n)`, `VARBINARY(n)`: Binary data
- `VARBINARY(MAX)`: Large binary (e.g., images)

Relationships in SQL Server

A **relationship** in SQL Server defines how **two or more tables** are connected through **keys**. These relationships help maintain **data integrity** and allow you to retrieve related data efficiently.

1. One-to-One (1:1)

- One row in Table A relates to **one** row in Table B.
- Rare in practice.
- Example: Each employee has one unique parking spot.

2. One-to-Many (1:N)

- One row in Table A relates to **many** rows in Table B.
- Most common relationship.
- Example: One `Department` has many `Employees`.

🔗 What Is a Many-to-Many Relationship?

A **many-to-many relationship** means:

- One row in **Table A** can relate to **many** rows in **Table B**.
- And one row in **Table B** can relate to **many** rows in **Table A**.

📘 Example Scenario: Students and Courses

- A student can enroll in **many courses**.
- A course can have **many students**.

Primary Key

- A table typically has a column or combination of columns that contain values that uniquely identify each row in the table.
  - This column, or columns, is called the **primary key (PK)** of the table and enforces the entity integrity of the table.
- If a primary key is defined on more than one column, values can be duplicated within one column, but each combination of values from all the columns in the primary key definition must be unique.

Primary Key

ProductID	VendorID	AverageLeadTime	StandardPrice	LastReceiptCost
1	1	17	47.8700	50.2635
2	104	19	39.9200	41.9160
7	4	17	54.3100	57.0255
609	7	17	25.7700	27.0585
609	100	19	28.1700	29.5785

ProductVendor table

In database terminology, there is a general key concept called Candidate Key. A **candidate key** is an attribute or a set of attributes that can uniquely identify each record (row) in a database table. In simpler terms, it's a "candidate" to become the primary key of the table.

A table can have one or more candidate keys. Out of all the candidate keys, one is chosen by the database designer to be the **primary key**. The primary key is the main, official unique identifier for the table. The other candidate keys that are not chosen as the primary key are called **alternate keys**.

Each value in the primary key column(s) must be **unique**.
A primary key **cannot contain NULL** values.
Primary key values ideally should **not change** over time.

Foreign Key

A **foreign key** is a field (or collection of fields) in one table that **uniquely identifies a row of another table**.

✅ 1. Reference to a Primary Key

- The foreign key must reference a **primary key** (or a unique key) in another table.

✅ 2. Matching Data Types

- The data type of the foreign key column must match the data type of the referenced primary key column.

✅ 3. Referential Integrity

- The foreign key ensures that the value in the child table **must exist** in the parent table, unless `NULL` is allowed.

From Adventureworks Database:

The Sales.SalesOrderHeader table has a foreign key link to the Sales.SalesPerson table because there's a logical relationship between sales orders and salespeople.

The SalesPersonID column in the SalesOrderHeader table matches the primary key column of the SalesPerson table.

The SalesPersonID column in the SalesOrderHeader table is the foreign key to the SalesPerson table.

By creating this foreign key relationship, a value for SalesPersonID can't be inserted into the SalesOrderHeader table if it doesn't already exist in the SalesPerson table.

Primary Key & Foreign Key Example

```sql
-- Parent table
CREATE TABLE Departments (
    DeptID INT PRIMARY KEY,
    DeptName VARCHAR(100)
);
```

```sql
-- Child table with a clearly defined foreign key
CREATE TABLE Employees (
    EmpID INT PRIMARY KEY,
    EmpName VARCHAR(100),
    DeptID INT,
    FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)
);
```


In this example:

DeptID in **Employees** is a **foreign key**.
It references **DeptID** in Departments, which is a **primary key**.

This ensures that every employee is assigned to a valid department.

🧩 Types of Keys in SQL Server

1. Primary Key

- Uniquely identifies each row in a table.
- Cannot contain `NULL` values.
- Each table can have **only one** primary key.
- Often used in relationships as a reference.

Example:

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100)
);
```

2. Foreign Key

- Refers to the **primary key** in another table.
- Creates a **relationship** between tables.
- Ensures **referential integrity** (i.e., the referenced value must exist).

Example:

```
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

🧩 Types of Keys in SQL Server (cont…)

3. Unique Key

- Ensures all values in a column are **unique**.
- Can contain `NULL` values.
- A table can have **multiple** unique keys.

Example:

```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE
);

```

4. Composite Key

- A key made up of **two or more columns**.
- Used when a single column is not enough to uniquely identify a row.

Example:

```
CREATE TABLE CourseRegistrations (
    StudentID INT,
    CourseID INT,
    PRIMARY KEY (StudentID, CourseID)

```

Constraints

- In **Microsoft SQL Server**, a **constraint** is a rule applied to a **column** or a **table** to **enforce data integrity**.
  - Constraints ensure that the data entered into the database is **accurate, consistent, and reliable**.
  - **Primary Keys**, **Foreign Keys**, and **Unique Keys** all are various types of constraints

✅ Why Constraints?

- Prevent invalid data entry.
- Maintain relationships between tables.
- Enforce business rules at the database level (not just in the application).

Constraint	Purpose
PRIMARY KEY	Uniquely identifies each row
FOREIGN KEY	Maintains referential integrity
UNIQUE	Ensures unique values in a column
CHECK	Validates data against a condition
DEFAULT	Provides a default value
NOT NULL	Disallows NULL values

Constraints (cont…)

4. CHECK

- Validates that values in a column meet a specific condition.

Example:

```
CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    Credits INT CHECK (Credits BETWEEN 1 AND 10)
);
```

5. DEFAULT

- Assigns a default value when no value is provided.

Example:

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name NVARCHAR(100),
    IsActive BIT DEFAULT 1
);
```


Explore constraints in SSMS Diagrams using AdventureWorks database

Constraints (cont…)

6. NOT NULL

- Ensures a column cannot store NULL values.

Example:

```
CREATE TABLE Departments (
    DeptID INT PRIMARY KEY,
    DeptName NVARCHAR(100) NOT NULL
);
```


Explore constraints in SSMS Diagrams using AdventureWorks database

Constraints (Complete Example)

```
CREATE TABLE dbo.Customer
(
    CustomerID INT IDENTITY(1,1)
        CONSTRAINT PK_Customer PRIMARY KEY,  -- Primary Key
    FullName NVARCHAR(100) NOT NULL,         -- NOT NULL constraint
    Email NVARCHAR(320)
        CONSTRAINT UQ_Customer_Email UNIQUE, -- Unique constraint
    Age INT
        CONSTRAINT CK_Customer_Age CHECK (Age >= 18), -- Check constraint
    CityID INT
        CONSTRAINT FK_Customer_City FOREIGN KEY REFERENCES dbo.City(CityID), -- Foreign Key
    IsActive BIT
        CONSTRAINT DF_Customer_IsActive DEFAULT (1), -- Default constraint
    CreatedAt DATETIME2
        CONSTRAINT DF_Customer_CreatedAt DEFAULT (SYSUTCDATETIME()) -- Default constraint
);
```


You can **name** the constraint in **both** forms (CONSTRAINT PK_Name ...).
If you **don't** provide a name, SQL Server generates a system name (e.g., PK__Table__...).
For maintainability, it's best to **name your constraints**.

Table Schema

- We have covered how to create a table, set keys/constraints, relationships, etc.
- In SQL Server, a **table schema** refers to the:
  - **structure and organization, or**
  - **a blueprint** of a table within a database.
  - It is a combination of all the below (and more):
    - The columns (attributes)
    - The data types of each column
    - Constraints (rules) on the data
    - Relationships with other tables

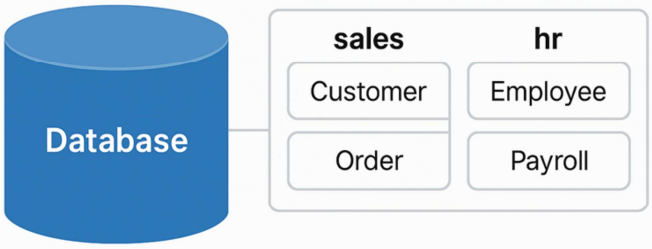
Previous slide provides a complete example of a table schema

Schema Object in SQL Server

- Besides table schema, or a table blueprint, the word **schema** refers to a specific object in SQL Server.
  - In **SQL Server**, a **schema** is a **container** that groups multiple database objects such as tables, and other database objects (which we haven't covered so far, such as *functions*, *views*, *stored procedures*, etc.)
  - It's a way to organize and secure objects within a database.
- Basically, a schema is like a **folder inside a database** that helps organize objects, manage security, and avoid naming conflicts.

What is the purpose of a schema in SQL Server?

- Organization of objects
- Security and permissions
- Avoiding name conflicts
- Ownership and management



sales schema contains Customer and Order Tables

hr schema contains Employee and Payroll Tables

Purpose of a Schema in SQL Server

1. Organization of Objects

- Schemas help group related objects together.
- Example:
  - `sales.Customer`, `sales.Order` → all sales-related tables in the `sales` schema.
  - `hr.Employee`, `hr.Payroll` → HR-related tables in the `hr` schema.

2. Security and Permissions

- Permissions can be granted at the **schema level** instead of individual tables.
- Example:

```
1  GRANT SELECT ON SCHEMA::sales TO SalesUser;
```

This gives `SalesUser` read access to **all objects in the `sales` schema**.

3. Avoiding Name Conflicts

- Two tables with the same name can exist in different schemas.
- Example:

```
1  CREATE TABLE hr.Employee (...);
2  CREATE TABLE sales.Employee (...);
3
```

Both `hr.Employee` and `sales.Employee` can coexist in the same database.

Purpose of a Schema in SQL Server

4. Ownership and Management

- Schemas allow **separation of ownership** from the database owner.
- You can assign different owners to different schemas for better governance.

✅ Default Schema

- If you don't specify a schema, SQL Server uses **dbo** by default.
- Example:

```
1  CREATE TABLE dbo.Customer (...);  -- dbo is the default schema
```

Purpose of a Schema in SQL Server

3. Avoiding Name Conflicts

- Two tables with the same name can exist in different schemas.
- Example:

```
CREATE TABLE hr.Employee (...);
CREATE TABLE sales.Employee (...);

```

Both `hr.Employee` and `sales.Employee` can coexist in the same database.

4. Ownership and Management

- Schemas allow **separation of ownership** from the database owner.
- You can assign different owners to different schemas for better governance.

Purpose of a Schema in SQL Server

- In SQL Server, **dbo** stands for **Database Owner**, and it is the **default schema** for all objects created by users who do not explicitly specify a schema.
- If you create a table without specifying a schema, it goes into **dbo** schema.

Default Schema

- If you don't specify a schema, SQL Server uses **dbo** by default.
- Example:

```
CREATE TABLE dbo.Customer (...);  -- dbo is the default schema
```

```
CREATE TABLE Customer (CustomerID INT PRIMARY KEY, Name NVARCHAR(100));
-- Actually created as dbo.Customer
```

It's a best practice of database design to create separate schemas like sales, hr, edu to organize tables and other related database objects

Schemas (cont…)

Fully Qualified Name

- Always reference objects as `schema.object` for clarity:

```sql
SELECT * FROM dbo.Customer;
```

Example

```sql
-- Table created without schema goes to dbo
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE
);

-- Equivalent to:
CREATE TABLE dbo.Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE
);

```