

Introduction to Databases

How Do RDBMS Work? Managing DBs Using IDEs

SoftUni Team
Technical Trainers



SoftUni



Software University
<https://softuni.bg>

Table of Contents

1. Data Management
2. Database Engines
3. Data Types in SQL Server
4. Database Modeling
5. Basic SQL Queries



sli.do

#csharp-db



Data Management

When Do We Need a Database?

Storage vs. Management (1)

- Conventional Data Storage

- Notes
 - Receipts



Storage vs. Management (2)

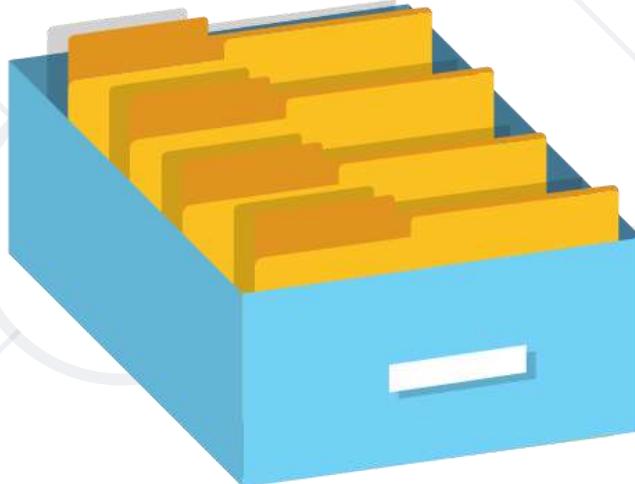
- We can group related pieces of data into separate columns



Order#	Date	Customer	Product	S/N	Unit Price	Qty	Total
315	07/16/2016	David Rivers	Oil Pump	OP147-0623	69.90	1	69.90

Storage vs. Management (3)

- Storing data is **not** the primary reason to use a Database
- Flat storage **eventually** runs into **issues** with
 - Size
 - Ease of updating
 - Searching
 - Concurrency
 - Security
 - Consistency



Databases and RDBMS

- A database is an **organized** collection of information
 - It imposes **rules** on the contained data
 - Relational storage first proposed by **Edgar Codd** in 1970
- A **Relational Data Base Management System** provides tools to **manage** the database
 - It **parses requests** from the user and takes the **appropriate** action
 - The user doesn't have direct access to the stored data

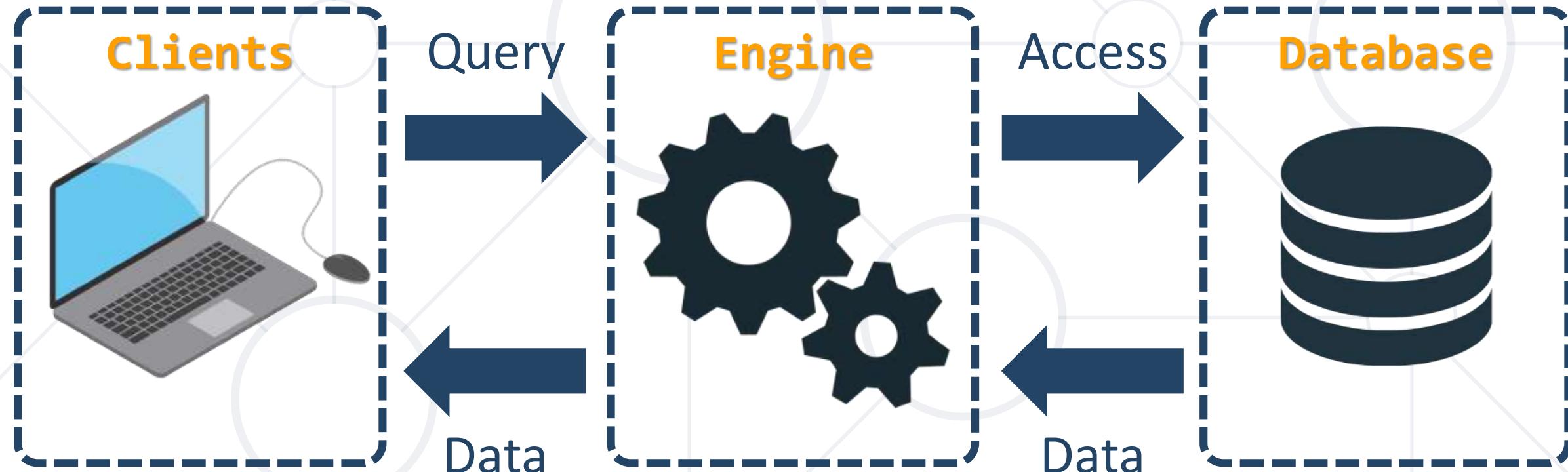




Database Engines

Database Engine Flow

- SQL Server uses the Client-Server Model



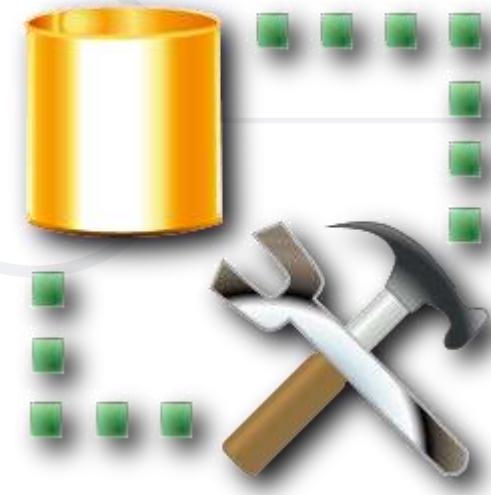
Download Clients & Servers

- Download **SQL Server Express** Edition from Microsoft

<https://go.microsoft.com/fwlink/?LinkId=866662>

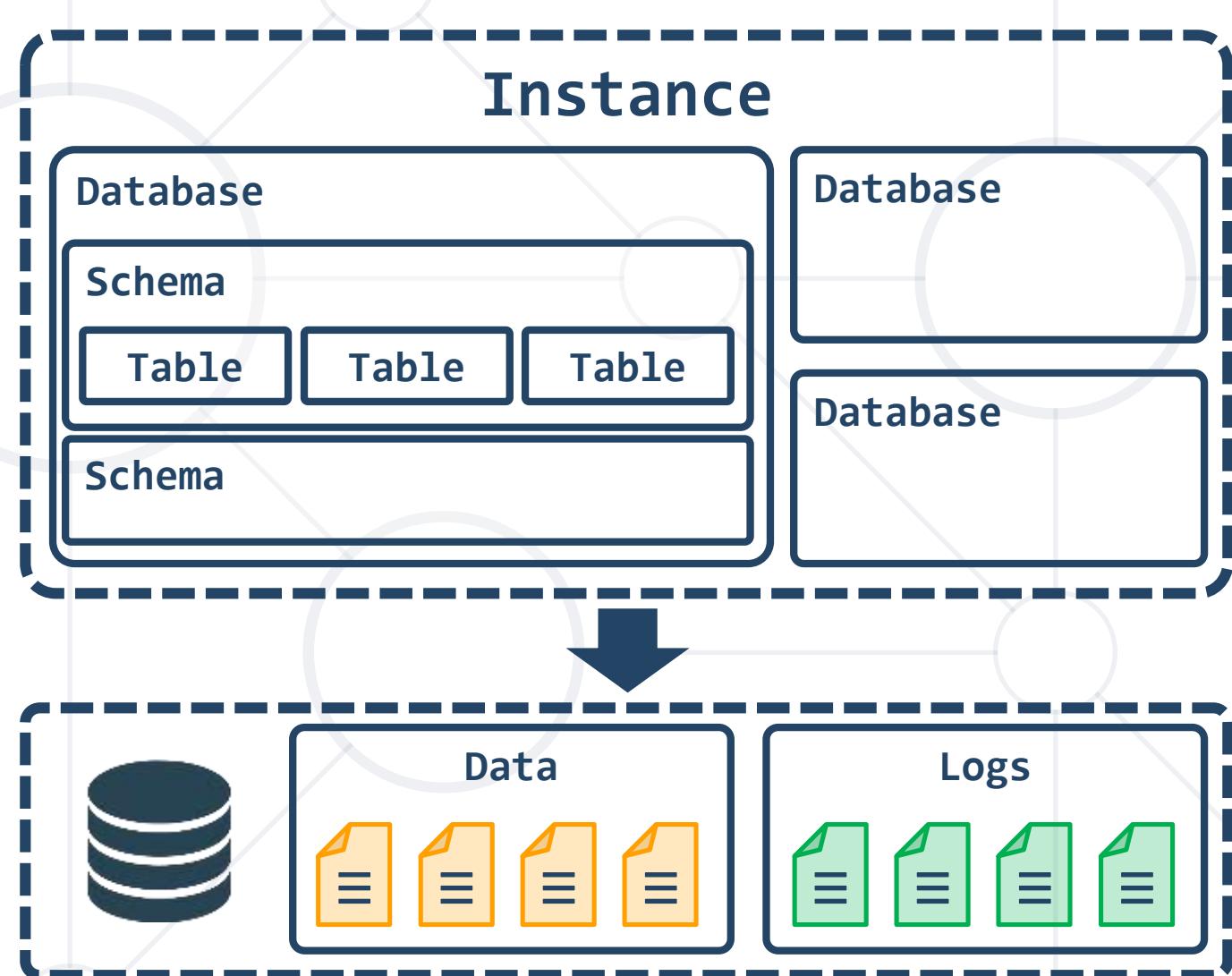
- Download SQL Server **Management Studio** separately

<https://aka.ms/ssmsfullsetup>



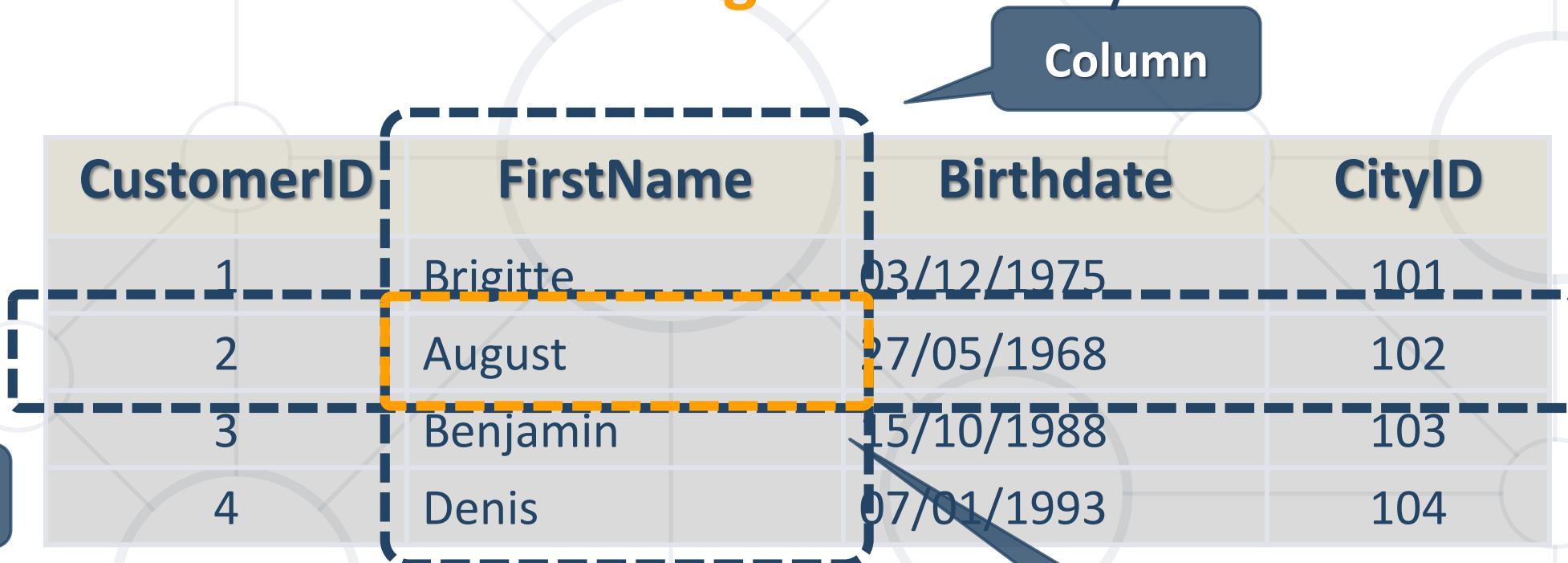
SQL Server Architecture

- Logical Storage
 - Instance
 - Database
 - Schema
 - Table
- Physical Storage
 - Data Files and Log files
 - Data Pages



Database Table Elements

- The table is the main **building block** of any database



CustomerID	FirstName	Birthdate	CityID
1	Brigitte	03/12/1975	101
2	August	27/05/1968	102
3	Benjamin	15/10/1988	103
4	Denis	07/01/1993	104

- Each **row** is called a **record** or **entity**
- Columns (**fields**) define the **type** of data they contain

Structured Query Language

- To communicate with the Engine we use **SQL**
 - **Declarative** language
- Logically divided in four sections
 - **Data Definition** – describe the structure of our data
 - **Data Manipulation** – store and retrieve data
 - **Data Control** – define who can access the data
 - **Transaction Control** – bundle operations and allow rollback



Data Types in SQL Server

Data Types in SQL Server (1)

- Numeric
 - **BIT** (1-bit), **TINYINT** (8-bit), **SMALLINT** (16-bit)
 - **INT** (32-bit), **BIGINT** (64-bit)
 - **FLOAT**, **REAL**, **DECIMAL(precision, scale)**
- Textual
 - **CHAR(size)** – fixed size string
 - **VARCHAR(size)** – variable size string
 - **NCHAR(size)** – Unicode fixed size string
 - **NVARCHAR(size)** – Unicode variable size string

Size of Textual Characters

```
DECLARE @VarcharVar VARCHAR(5) = 'Test';
DECLARE @NvarcharVar NVARCHAR(5) = 'Test';
DECLARE @CharVar CHAR(5) = 'Test';
DECLARE @NCharVar NCHAR(5) = 'Test';

SELECT DATALENGTH(@VarcharVar),
       DATALENGTH(@NvarcharVar),
       DATALENGTH(@CharVar),
       DATALENGTH(@NCharVar)
```

Data Types in SQL Server (2)

- Binary data
 - **BINARY(size)** – fixed length sequence of bits
 - **VARBINARY(size)** – a sequence of bits, 1-8000 bytes or **MAX** (2GB)
- Date and time
 - **DATE** – date in range 0001-01-01 through 9999-12-31
 - **DATETIME** – date and time with precision of 1/300 sec
 - **DATETIME2** – type that has a larger date range
 - **SMALLDATETIME** – date and time (1 minute precision)
 - **TIME** – defines a time of a day (no time zone)
 - **DATETIMEOFFSET** – date and time that has time zone

Date and Time in SQL Server

DATA TYPE	① RANGE OF VALUES	② ACCURACY	③ STORAGE SPACE
SMALLDATETIME	01/01/1900 to 06/06/2079	1 minute	4 bytes
DATETIME	01/01/1753 to 12/31/9999	0.00333 seconds	8 bytes
DATETIME2	01/01/0001 to 12/31/9999	100 nanoseconds	6 to 8 bytes
DATETIMEOFFSET	01/01/0001 to 12/31/9999	100 nanoseconds	8 to 10 bytes
DATE	01/01/0001 to 12/31/9999	1 day	3 bytes
TIME	00:00:00.0000000 to 23:59:59.9999999	100 nanoseconds	3 to 5 bytes

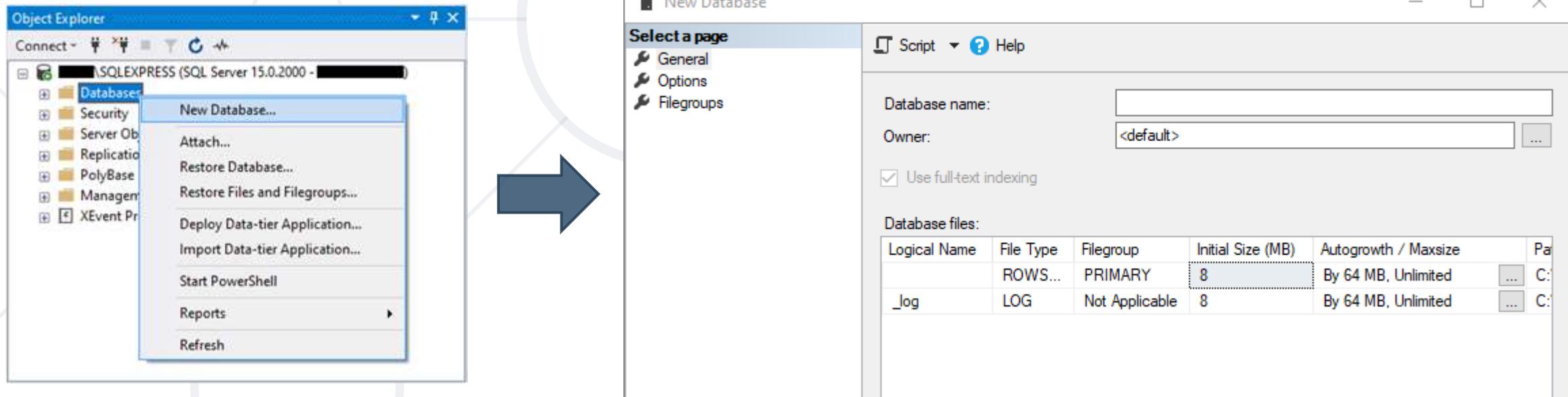


Database Modelling

Data Definition Using SSMS

Creating a New Database

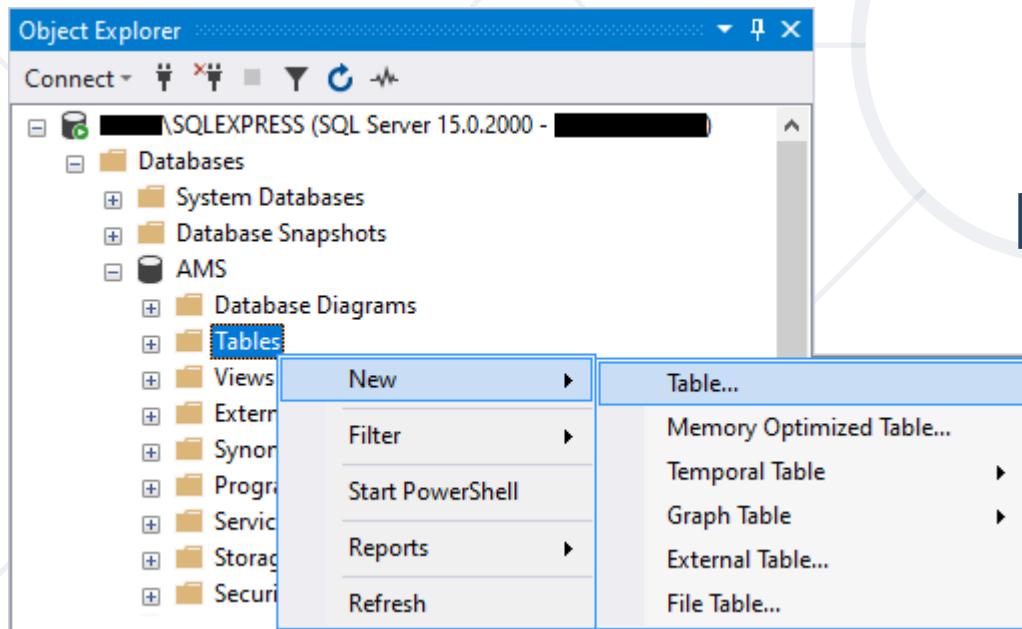
- Select **New Database** from the **context menu** under "Databases"



- You may need to **Refresh [F5]** to see the results

Creating Tables (1)

- Right-click from the **context menu** under "New" inside the desired database → "**Table**"

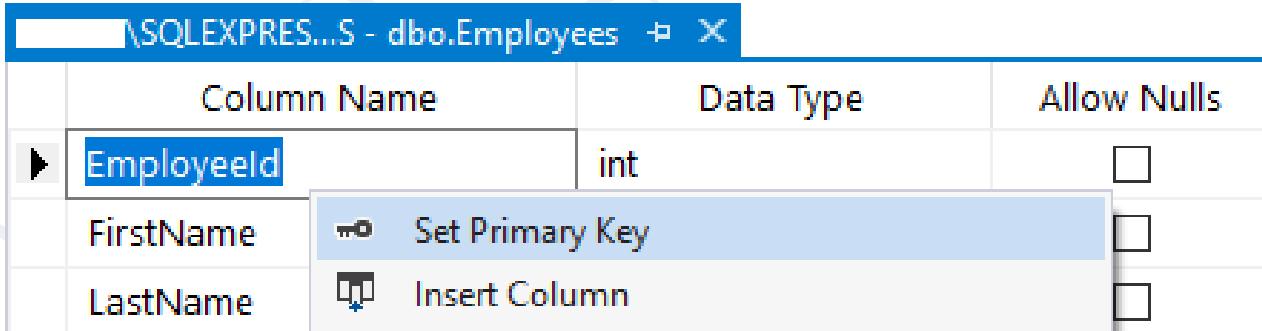


Column Name	Data Type	Allow Nulls
EmployeeId	int	<input type="checkbox"/>
FirstName	nvarchar(50)	<input type="checkbox"/>
LastName	nvarchar(50)	<input type="checkbox"/>

- Table name can be set from its **Properties [F4]** or when it is **saved**

Creating Tables (2)

- A **Primary Key** is used to uniquely identify and index records
- Setting **primary key** on a column:



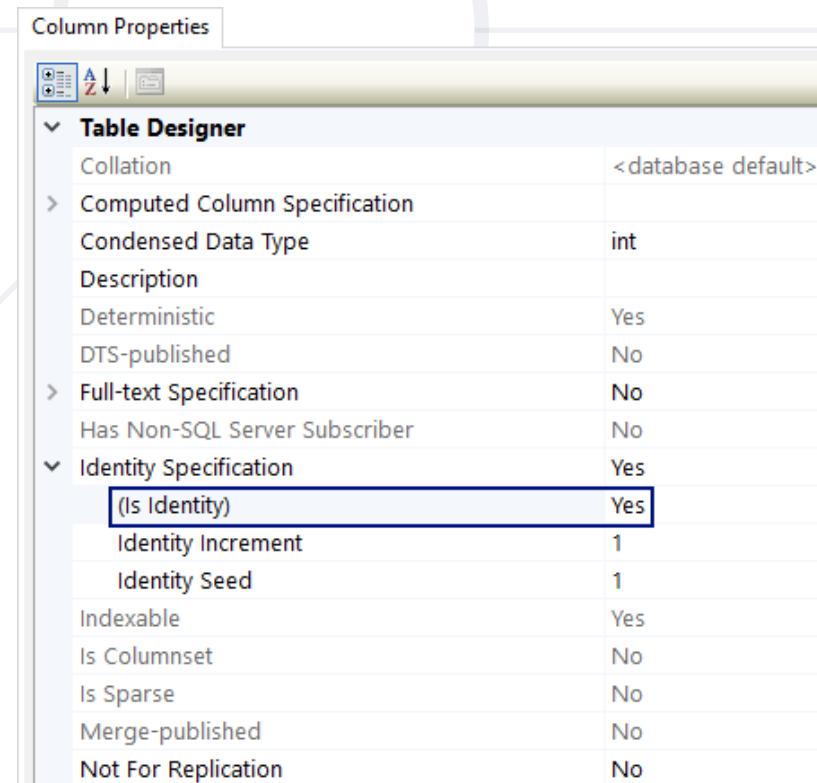
Column Name	Data Type	Allow Nulls
EmployeeId	int	<input type="checkbox"/>
FirstName		<input type="checkbox"/>
LastName		<input type="checkbox"/>

Creating Tables (3)

- **Identity** – The value in the column is automatically incremented when a new record is added
 - These values cannot be assigned manually
 - **Identity Seed** – the initial number (1 by default)
 - **Identity Increment** – how much each consecutive value is incremented

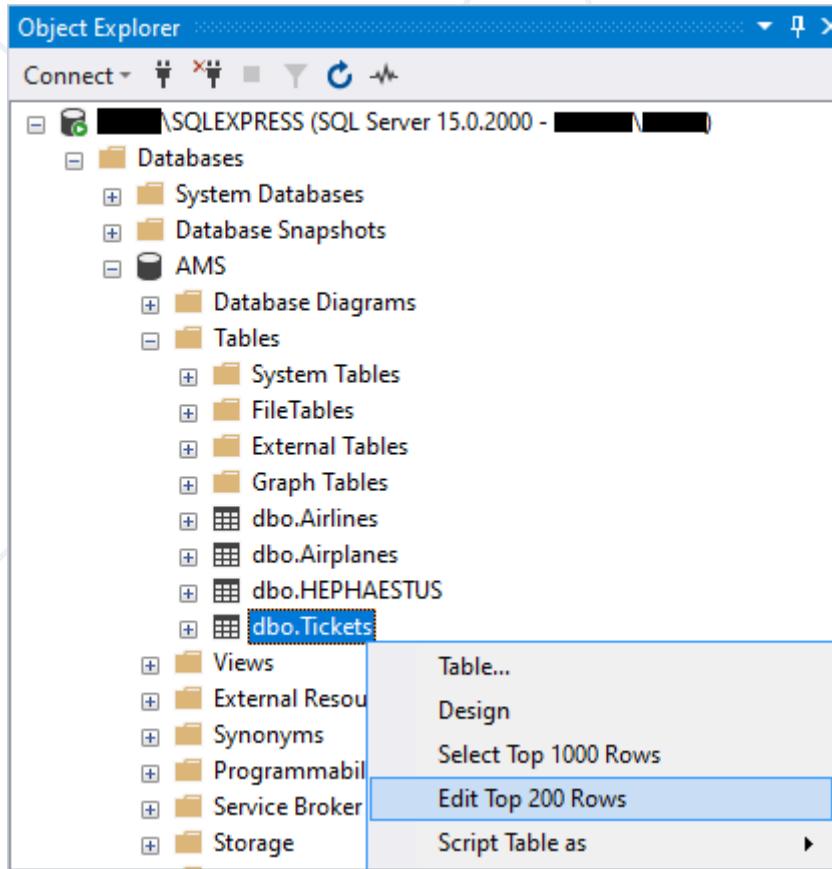
Creating Tables (4)

- Setting an identity through the "**Column Properties**" window:



Storing and Retrieving Data (1)

- We can **add**, **modify** and **read** records with Management Studio
- To insert or edit a record, click **Edit** from the context menu

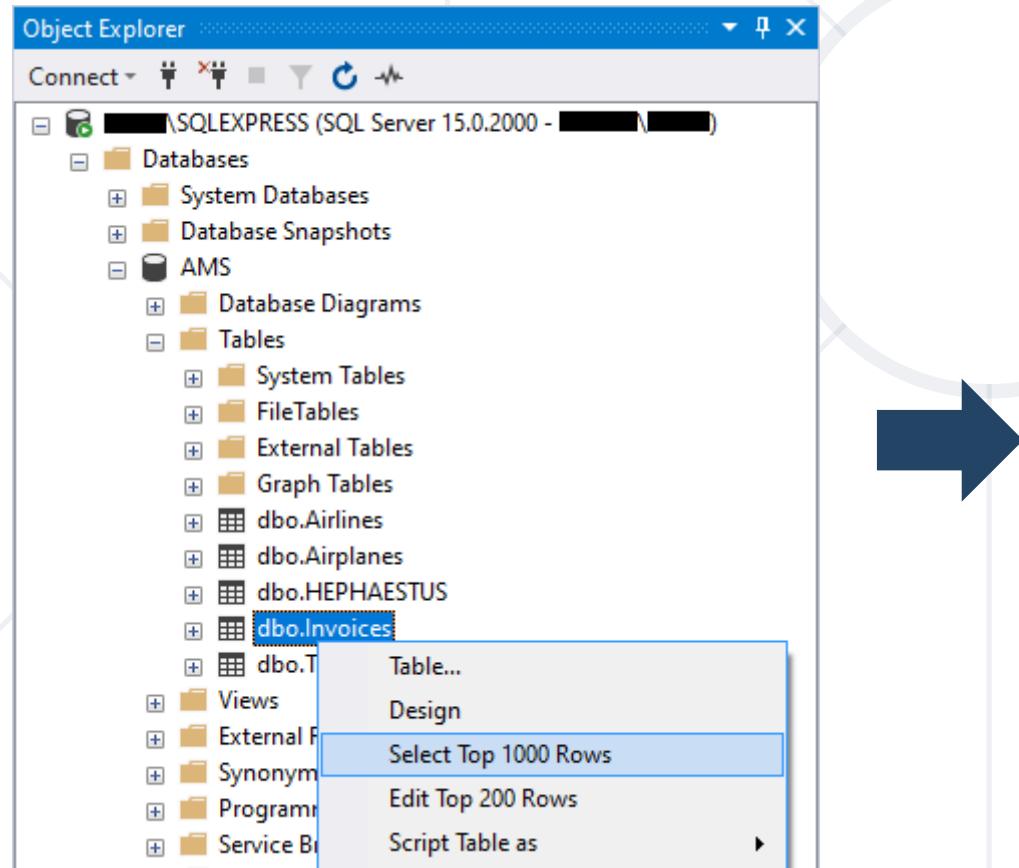


	TicketId	Price	Class	Seat	CustomerId
1	4500.00	First	233-A	3	
2	2669.85	Second	123-D	1	
3	1800.75	Second	12-Z	2	
4	616.02	Third	45-Q	2	
5	840.00	Third	201-R	4	
6	3150	Second	13-T	1	
7	5500.00	First	98-O	2	
8	100.00	First	1	1	
*	NULL	NULL	NULL	NULL	NULL

Enter data at the end to add a new row

Storing and Retrieving Data (2)

- To retrieve records, click **Select** from the context menu

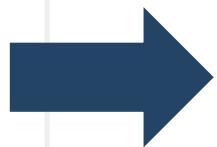
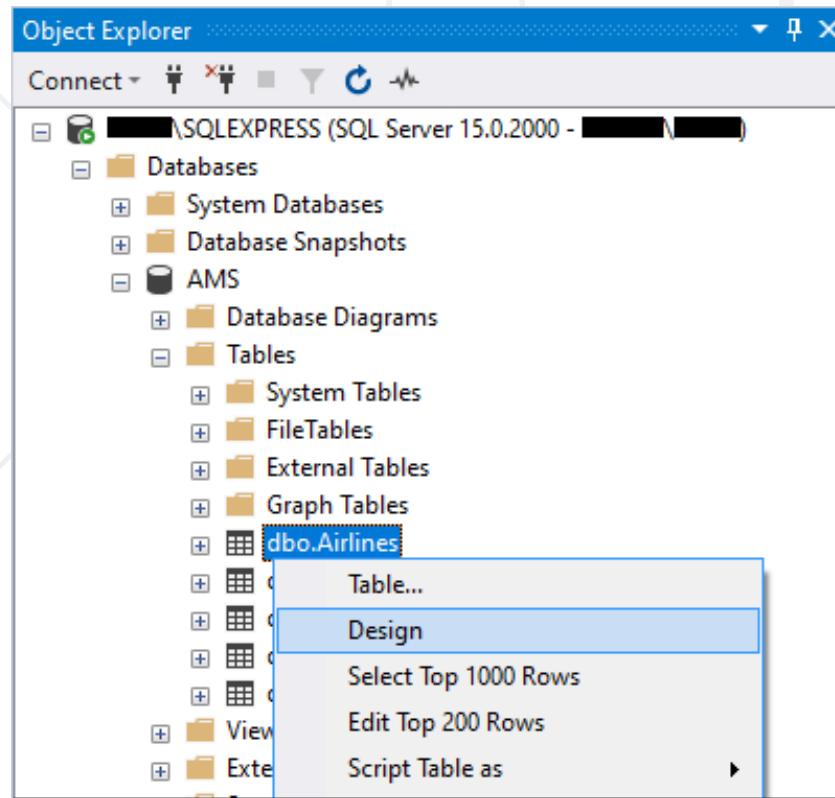


Invoiceld	CustomerId	InvoiceDate	BillingAddress	BillingCountry
1	1	2022-12-01	Theodor Heuss Strasse 34	Germany
2	4	2022-12-15	Ullevalsveien 14	Norway
3	8	2023-01-03	Gretrystraat 63	Belgium
4	23	2023-01-11	69 Salem Street	United States
5	14	2023-01-06	8210 111 ST NW	United States
6	37	2023-01-19	Berger Strasse 10	Germany
7	38	2023-02-01	Barbarossastrasse 19	Germany
8	40	2023-02-01	8, Rue Hanovre	France
9	42	2023-02-02	9, Place Louis Barthou	France
10	46	2023-02-03	3 Chatham Street	Ireland
11	52	2023-02-06	202 Hoxton Street	United Kingdom
12	1	2023-02-11	Theodor Heuss Strasse 34	Germany

- The received information can be customized with **SQL queries**

Altering Tables

- You can change the properties of a table after its creation
- Select **Design** from the table's context menu



Column Name	Data Type	Allow Nulls
AirlineId	int	<input type="checkbox"/>
AirlineName	nvarchar(30)	<input type="checkbox"/>
Nationality	nvarchar(3)	<input type="checkbox"/>
Rating	int	<input checked="" type="checkbox"/>

Changes cannot conflict with existing rules!



Basic SQL Queries

Data Definition Using T-SQL

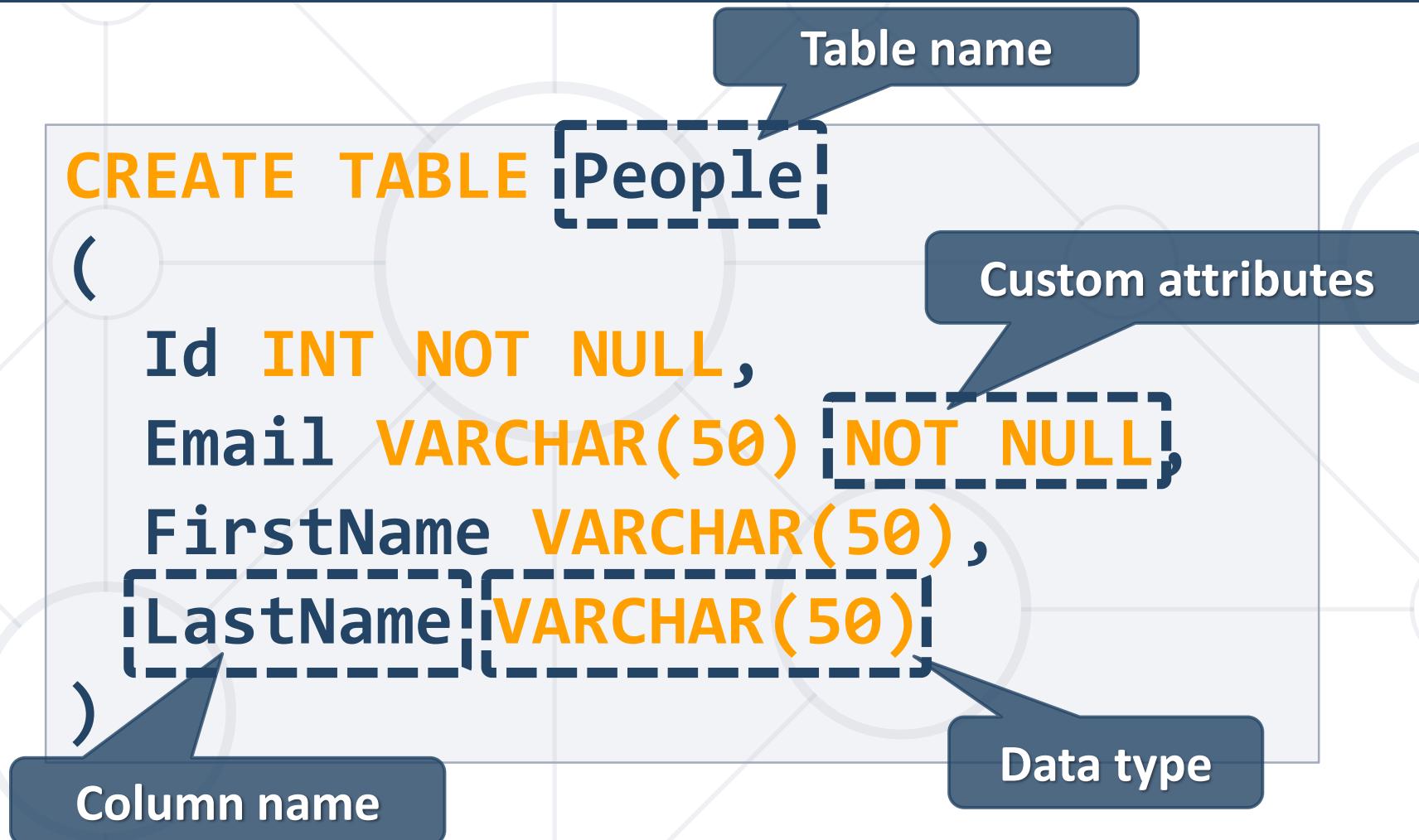
- We can communicate with the database engine using SQL
- Queries provide greater **control** and **flexibility**
- To create a database using SQL:

```
CREATE DATABASE Employees
```

Database name

- SQL keywords are traditionally **capitalized**

Table Creation in SQL



Retrieve Records in SQL

- To get all records from a table

```
SELECT * FROM Employees
```

- You can limit the number of rows and number of columns

Number of records

List of columns

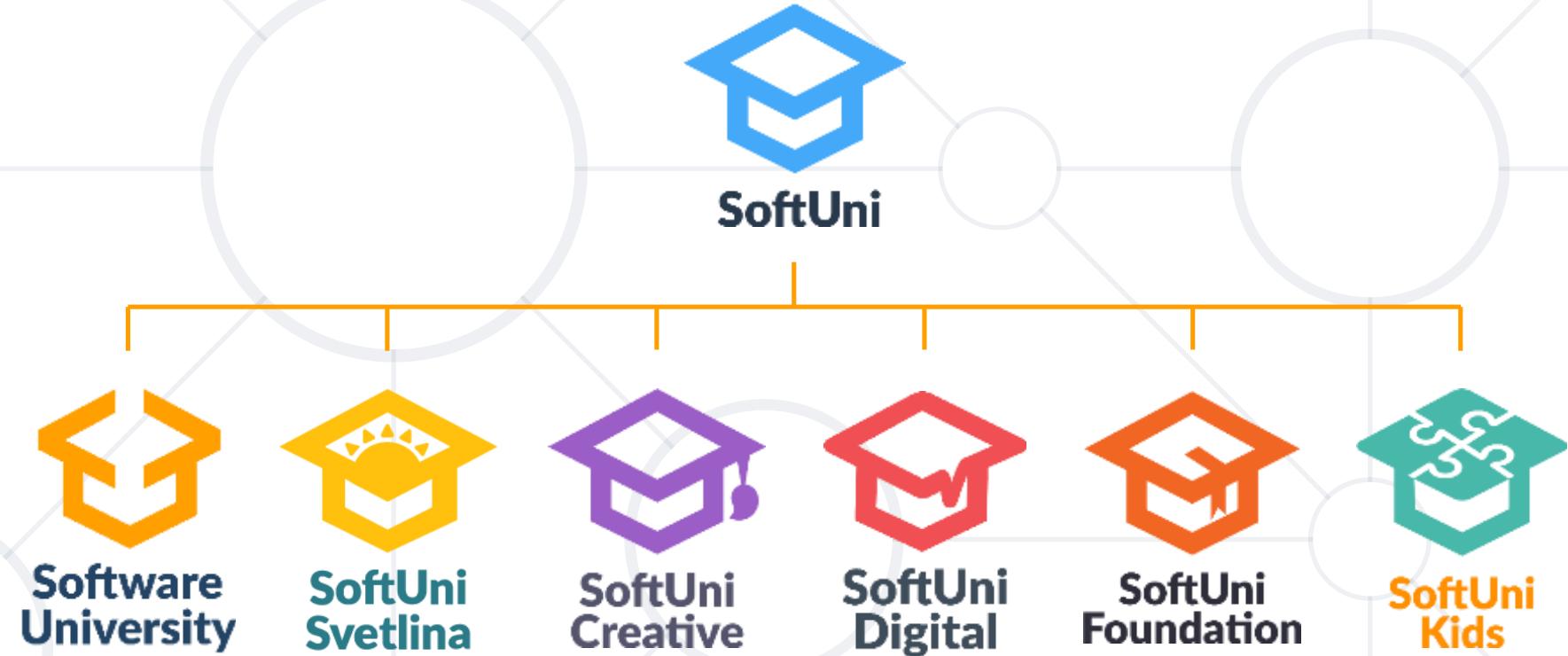
```
SELECT TOP (5) FirstName, LastName  
FROM Employees
```

Summary

- RDBMS stores and manages data
- Table relations reduce repetition and complexity
- Table columns have **fixed types**
- We can use Management Studio to **create** and **customize** tables
- SQL provides **greater control** over actions



Questions?



SoftUni Diamond Partners



SCHWARZ



Coca-Cola HBC
Bulgaria



Postbank

Решения за твоето упре



**SOFTWARE
GROUP**

Bosch.**.IO**



SmartIT



**PHAR
VISION**



CAREERS



INDEAVR
Serving the high achievers



**SUPER
HOSTING
.BG**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Basic CRUD in SQL Server

Create, Read, Update, Delete
using SQL Queries



SoftUni



Table of Contents

1. Query Basics
2. Retrieving Data
 - SELECT
 - Views
3. Writing Data
 - INSERT
4. Modifying Existing Records
 - UPDATE and DELETE



sli.do

#csharp-db



Query Basics

SQL and T-SQL Introduction

What Are SQL and T-SQL?

- **Structured Query Language**

- Declarative language
- Close to regular English

```
SELECT FirstName, LastName, JobTitle FROM Employees
```

- Supports definition, manipulation and access control of records
- **Transact-SQL (T-SQL)** – SQL Server's version of SQL
 - Supports control flow (**if-statements, loops**)
 - Designed for writing **logic** inside the database

SQL – Examples

```
SELECT FirstName, LastName, JobTitle FROM Employees
```

```
SELECT * FROM Projects WHERE StartDate = '1/1/2006'
```

```
INSERT INTO Projects(Name, StartDate)  
VALUES ('Introduction to SQL Course', '1/1/2006')
```

```
UPDATE Projects  
SET EndDate = '8/31/2006'  
WHERE StartDate = '1/1/2006'
```

```
DELETE FROM Projects  
WHERE StartDate = '1/1/2006'
```



Retrieving Data

Using SQL SELECT

Capabilities of SQL SELECT

Projection

Take a subset of the columns

Selection

Take a subset of the rows

Join

Combine tables by some column

Table 1

Table 2

SELECT – Example

- Selecting **all** columns from the "Departments" table

```
SELECT * FROM Departments
```

DepartmentID	Name	ManagerID
1	Engineering	12
2	Tool design	4
3	Sales	273
...

- Selecting **specific** columns

```
SELECT DepartmentId, Name  
FROM Departments
```



DepartmentID	Name
1	Engineering
2	Tool design
3	Sales
...	...

Column Aliases

- **Aliases** rename a table or a column heading

Display Name

```
SELECT EmployeeID AS ID,  
       FirstName,  
       LastName  
  FROM Employees
```



ID	FirstName	LastName
1	Guy	Gilbert
2	Kevin	Brown
...

- You can shorten fields or clarify abbreviations

```
SELECT c.Duration,  
       c.ACG AS 'Access Control Gateway'  
  FROM Calls AS c
```

Concatenation Operator

- You can **concatenate** column names using the **+** operator
 - String literals** are enclosed in **single quotes**
 - Column names containing **special symbols** use **brackets**

```
SELECT FirstName + ' ' + LastName AS [Full Name],  
      EmployeeID AS [No.]  
FROM Employees
```

Full Name	No.
Guy Gilbert	1
Kevin Brown	2
...	...

Problem: Employee Summary

- Find information about all employees, listing their **full name**, **job title** and **salary**
 - Use **concatenation** to display first and last names as **one field**

	Full Name	JobTitle	Salary
1	Guy Gilbert	Production Technician	12500.00
2	Kevin Brown	Marketing Assistant	13500.00
3	Roberto Tamburello	Engineering Manager	43300.00
4	Rob Walters	Senior Tool Designer	29800.00
5	Thierry D'Hers	Tool Designer	25000.00
6	David Bradley	Marketing Manager	37500.00
7	JoLynn Dobney	Production Supervisor	25000.00
8	Ruth Ellerbrock	Production Technician	13500.00
9	Gail Erickson	Design Engineer	32700.00

- Note: Query **SoftUni** database

Solution: Employee Summary

```
SELECT FirstName + ' ' + LastName  
      AS [Full Name],  
      JobTitle,  
      Salary  
  FROM Employees
```

Concatenation

Column Alias

Filtering the Selected Rows

- Use **DISTINCT** to eliminate **duplicate** results

```
SELECT DISTINCT DepartmentID  
FROM Employees
```

- Filter rows by specific **conditions** using the **WHERE** clause

```
SELECT LastName, DepartmentID  
FROM Employees  
WHERE DepartmentID = 1
```

- Other **logical operators** can be used for greater control

```
SELECT LastName, Salary FROM Employees  
WHERE Salary <= 20000
```

Other Comparison Conditions

- Combine conditions using **NOT**, **OR**, **AND** and **brackets**

```
SELECT LastName FROM Employees  
WHERE NOT (ManagerID = 3 OR ManagerID = 4)
```

- Using **BETWEEN** operator to **specify a range**

```
SELECT LastName, Salary FROM Employees  
WHERE Salary BETWEEN 20000 AND 22000
```

- Using **IN / NOT IN** to **specify a set of values**

```
SELECT FirstName, LastName, ManagerID  
FROM Employees  
WHERE ManagerID IN (109, 3, 16)
```

Comparing with NULL

- **NULL** is a special value that means missing value
 - Not the same as **0** or a **blank space**
- Checking for **NULL** values

```
SELECT LastName, ManagerId FROM Employees  
WHERE ManagerId = NULL
```

This is always **false!**

```
SELECT LastName, ManagerId FROM Employees  
WHERE ManagerId IS NULL
```

```
SELECT LastName, ManagerId FROM Employees  
WHERE ManagerId IS NOT NULL
```



Sorting Result Sets

- Sort rows with the **ORDER BY** clause

- ASC**: ascending order, default
- DESC**: descending order

```
SELECT LastName, HireDate  
      FROM Employees  
 ORDER BY HireDate
```

```
SELECT LastName, HireDate  
      FROM Employees  
 ORDER BY HireDate DESC
```



LastName	HireDate
Gilbert	1998-07-31
Brown	1999-02-26
Tamburello	1999-12-12
...	...

LastName	HireDate
Valdez	2005-07-01
Tsoflias	2005-07-01
Abbas	2005-04-15
...	...

Views

- Views are **named (saved) queries**
 - **Simplify** complex queries
 - **Limit access** to data for certain users
- Example: Get employee **names** and **salaries**, by department



```
CREATE VIEW v_EmployeesByDepartment AS
```

```
SELECT FirstName + ' ' + LastName AS [Full Name],  
      Salary  
  FROM Employees
```

Executes query

```
SELECT * FROM v_EmployeesByDepartment
```

Problem: Highest Peak

- Create a **view** that selects all information about the **highest peak**
 - Name the view **v_HighestPeak**

```
SELECT * FROM v_HighestPeak
```



	Id	PeakName	Elevation	MountainId
1	68	Everest	8848	9

- Note: Query **Geography** database

Solution: Highest Peak

- **TOP(x)** selects the first x values

```
CREATE VIEW v_HighestPeak  
AS  
SELECT TOP (1) *  
FROM Peaks  
ORDER BY Elevation DESC
```

Sorting column

Greatest value first





Writing Data in Tables

Using SQL INSERT

Inserting Data

- The SQL **INSERT** command

```
INSERT INTO Towns VALUES (33, 'Paris')
```

```
INSERT INTO Projects (Name, StartDate)  
VALUES ('Reflective Jacket', GETDATE())
```

- Bulk data** can be recorded in a single query, separated by comma

```
INSERT INTO EmployeesProjects  
VALUES (229, 1),  
(229, 2),  
(229, 3), ...
```



Inserting Data (2)

- Inserting rows into existing table:

```
INSERT INTO Projects (Name, StartDate)
    SELECT Name + ' Restructuring', GETDATE()
        FROM Departments
```

List of columns

- Using existing records to create a **new table**:

```
SELECT CustomerID, FirstName, Email, Phone
    INTO CustomerContacts
        FROM Customers
```

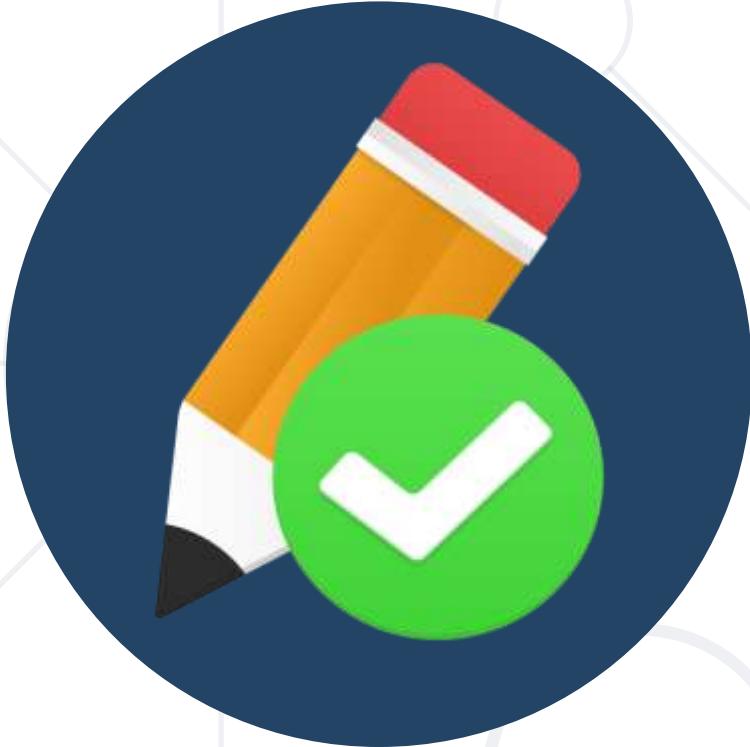
New table name

Existing source

- Sequences are special object in SQL Server
 - Similar to IDENTITY fields
 - Returns an incrementing value every time it's used

```
CREATE SEQUENCE seq_Customers_CustomerID  
    AS INT  
    START WITH 1  
    INCREMENT BY 1
```

```
SELECT NEXT VALUE FOR seq_Customers_CustomerID
```



Modifying Existing Records

Using SQL UPDATE and DELETE

Deleting Data

- Deleting specific rows from a table

```
DELETE FROM Employees WHERE EmployeeID = 1
```

- Note: Don't forget the **WHERE** clause!

Condition

- Delete all rows from a table (works faster than **DELETE**):

```
TRUNCATE TABLE Users
```



Updating Data

- The SQL **UPDATE** command

```
UPDATE Employees  
SET LastName = 'Brown'  
WHERE EmployeeID = 1
```

New values



```
UPDATE Employees  
SET Salary = Salary * 1.10,  
JobTitle = 'Senior' + JobTitle  
WHERE DepartmentID = 3
```

- Note: Don't forget the **WHERE** clause!



Problem: Update Projects

- Mark **all unfinished** Projects as being **completed today**
 - Hint: Unfinished projects have their **EndDate** set to **NULL**

Name	EndDate
Classic Vest	NULL
HL Touring Frame	NULL
LL Touring Frame	NULL
...	...



Name	EndDate
Classic Vest	2017-01-23
HL Touring Frame	2017-01-23
LL Touring Frame	2017-01-23
...	...

- Note: Query **SoftUni** database

Solution: Update Projects

```
UPDATE Projects  
SET EndDate = GETDATE()  
WHERE EndDate IS NULL
```



Filter only records
with no value

Summary

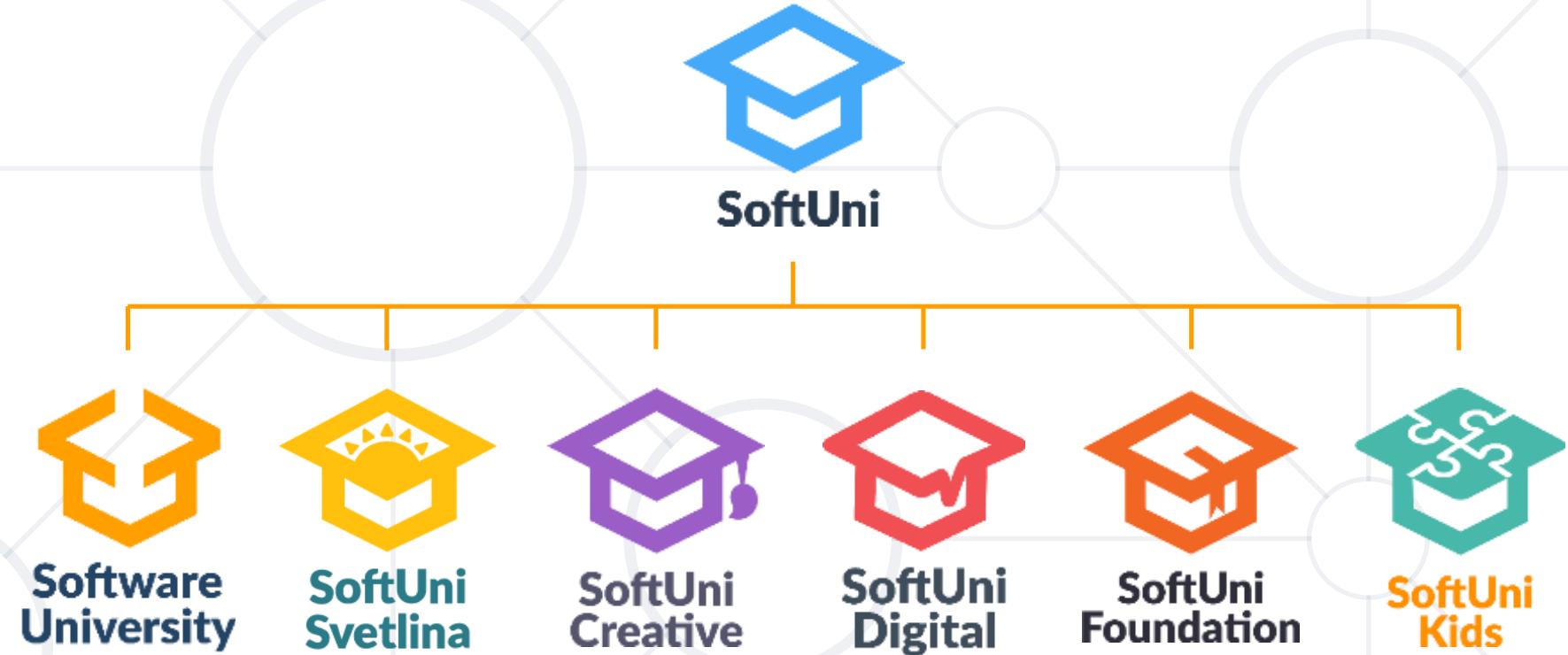
- T-SQL is the language of SQL Server

```
SELECT *
  FROM Projects
 WHERE StartDate = '1/1/2006'
```

- Queries provide a **flexible** and **powerful** **method** to **manipulate records**
- **Views** allow us to **store queries** for easier use



Questions?



SoftUni Diamond Partners



SCHWARZ



Coca-Cola HBC
Bulgaria



Postbank

Решения за твоето упре



**SOFTWARE
GROUP**

Bosch.**.IO**



SmartIT

POKERSTARS

CAREERS

AMBITIONED

INDEAVR
Serving the high achievers

createX

**DRAFT
KINGS**

**SUPER
HOSTING
.BG**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



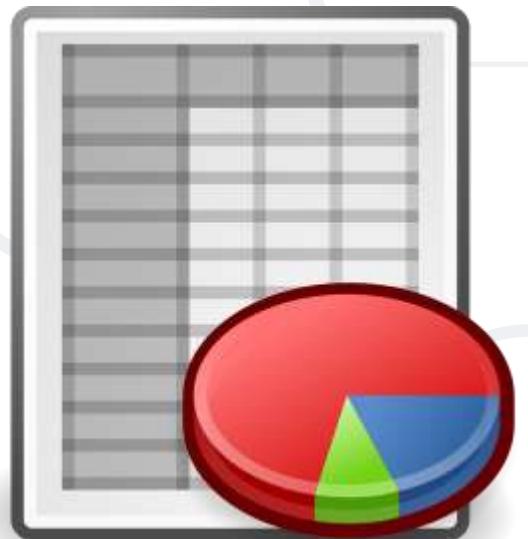
Table Relations

Database Design and Rules

SoftUni Team
Technical Trainers



SoftUni



Software University
<https://softuni.bg>

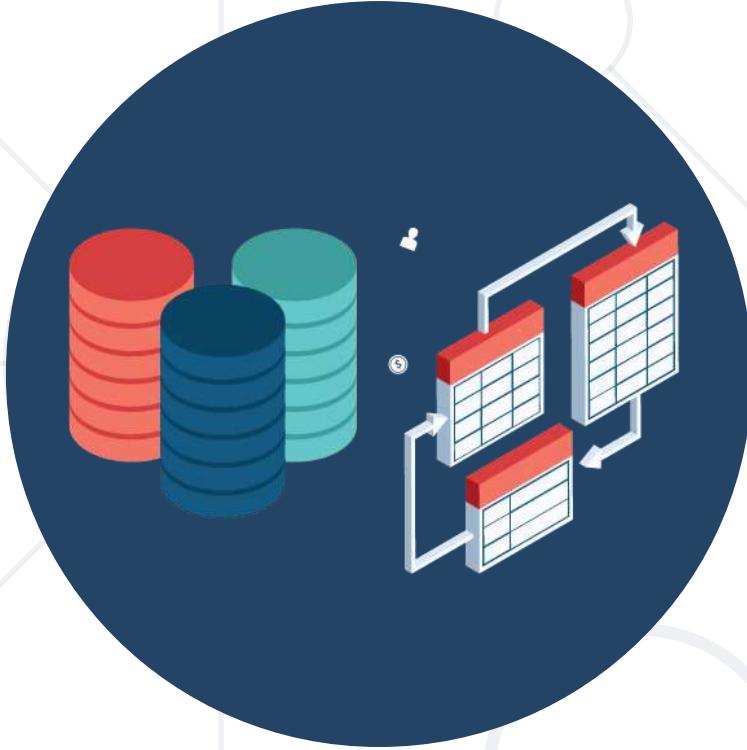
Table of Contents

1. Database Design
2. Database Normalization
3. Table Relations
4. Cascade Operations
5. E/R Diagrams



sli.do

#csharp-db



Database Design

Fundamental Concepts

Steps in Database Design

- Steps in the database design process:
 - Identify entities
 - Identify table columns
 - Define a primary key for each table
 - Identify and model relationships
 - Define other constraints
 - Fill tables with test data



DB Design: Identify Entities

- Entity tables represent objects from the real world
 - Most often they are nouns in the specification
 - For example:

We need to develop a system that stores information about students which are trained in various courses. The courses are held in different towns. When registering a new student the following information is entered: name, faculty number, photo and date.

- Entities: **Student, Course, Town**

DB Design: Identify Table Columns

- Columns are clarifications for the entities in the text of the specification, for example:

We need to develop a system that stores information about **students**, which are trained in various **courses**. The courses are held in different **towns**. When registering a new student the following information is entered:
[name] [faculty number] [photo] and [date]

- Students have the following characteristics:
 - **Name, faculty number, photo, date of enlistment** and a **list of courses** they visit

How to Choose a Primary Key?

- Always define an **additional column** for the primary key
 - Don't use an existing column (for example SSN)
 - Must be an **integer** number
 - Must be **declared** as a primary key
 - Use **IDENTITY** to implement auto-increment
 - Put the **primary key** as a **first column**
- Exceptions
 - Entities that have **well known ID**, e.g. **countries** (BG, DE, US) and **currencies** (USD, EUR, BGN)

DB Design: Identify Entity Relationships

- Relationships are **dependencies** between the entities:

We need to develop a system that stores information about **students**, which are trained in various **courses**. The **courses** are held in different **towns**. When registering a new student, the following information is entered: name, faculty number, photo and date.

- "**Students** are trained in courses" → many-to-many relationship
- "**Courses** are held in towns" → many-to-one (or many-to-many) relationship



Database Normalization

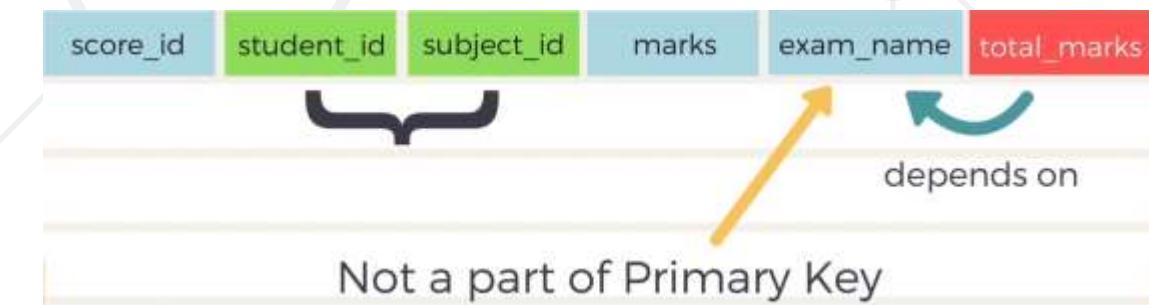
Database Normalization

- It is a technique of organizing the data in the database
- **Normalization** is a systematic approach of **decomposing** tables to eliminate data redundancy (repetition) and undesirable characteristics like insertion, update and deletion **anomalies**
- It is a multi-step process that puts data into **tabular form** removing duplicated data from the relation tables



Normal Forms

- **First Normal Form (1NF)**
 - Table should only have single (atomic) valued attributes/columns
 - Values stored in a column should be of the same domain (same type)
 - All the columns in a table should have unique names
 - The order in which data is stored should not matter
- **Second Normal Form (2NF)**
 - The table should be in the **First Normal form**
 - It shouldn't have **Partial Dependency** (dependency on part of the primary key)
- **Third Normal Form (3NF)**
 - The table is in the **Second Normal form**
 - It doesn't have **Transitive Dependency**



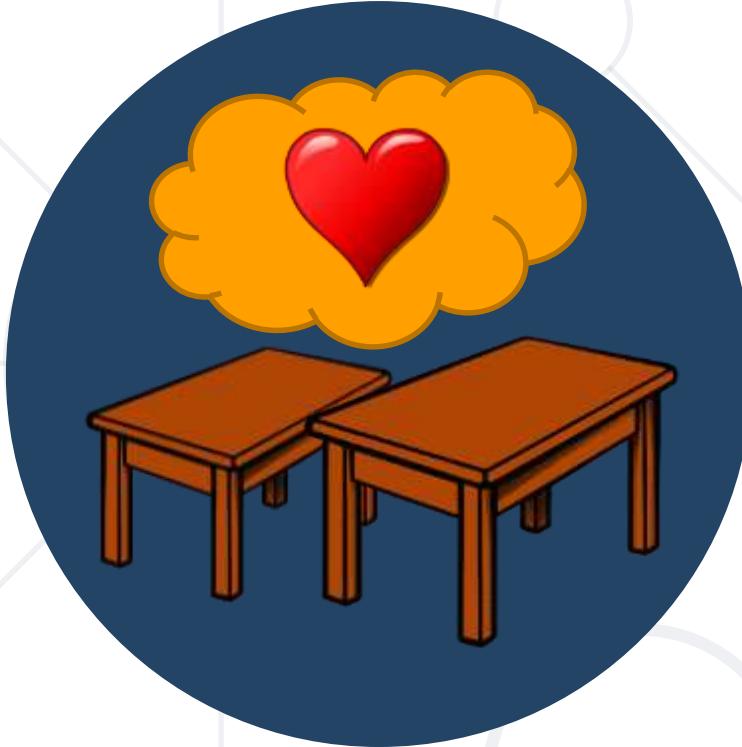


Table Relations

Relational Database Model in Action

Table Relations

- Relationships between tables are based on interconnections:
primary key → foreign key

Primary key

Towns

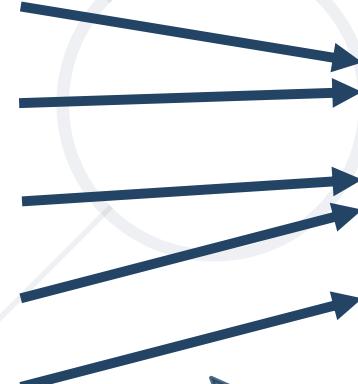
Foreign key

Primary key

Countries

	Id	Name	CountryId
1	Sofia		1
2	Varna		1
3	Munich		2
4	Berlin		2
5	Moscow		3

Relationship



	Id	Name
1	Bulgaria	
2	Germany	
3	Russia	

Custom Column Properties

- Primary Key

```
Id INT NOT NULL PRIMARY KEY
```

- Identity (auto-increment)

```
Id INT PRIMARY KEY IDENTITY
```

- Unique constraint – no repeating values in entire table

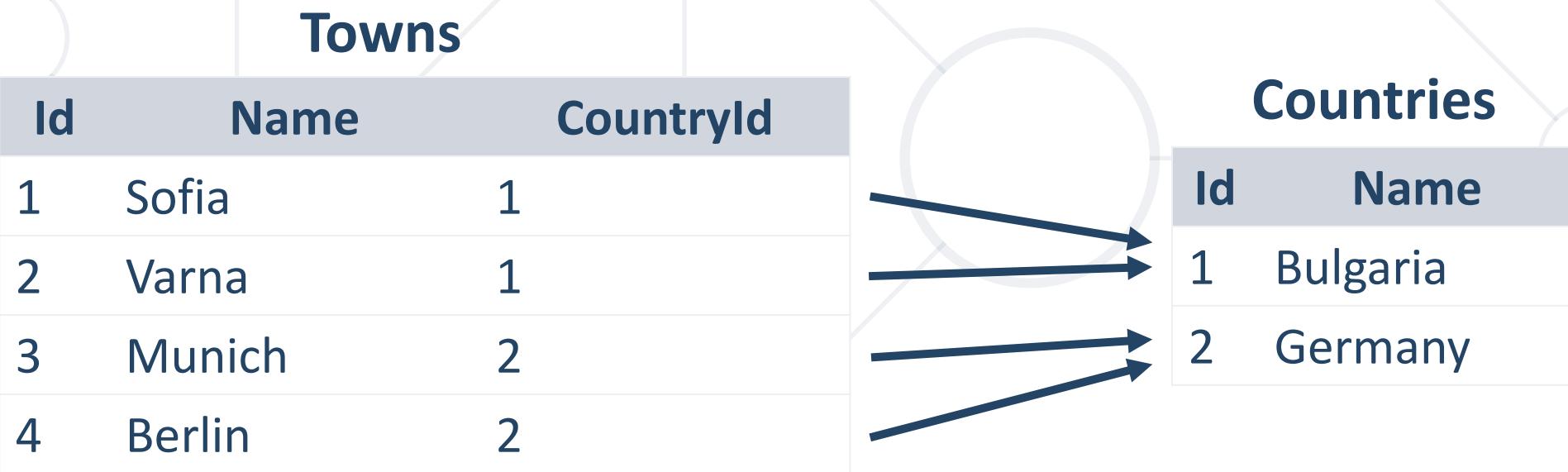
```
Email VARCHAR(50) UNIQUE
```

Table Relations: Foreign Key

- The **foreign key** is an **identifier** of a record located in **another table** (usually a primary key)
- Using relationships, we **refer** to data instead of **repeating** data
 - Country name is **not repeated**, it is **referred** to by its **primary key**

Towns

Id	Name	CountryId
1	Sofia	1
2	Varna	1
3	Munich	2
4	Berlin	2



```

graph LR
    T[Towns] --> C[Countries]
    T --> C
    T --> C
    T --> C
  
```

Countries

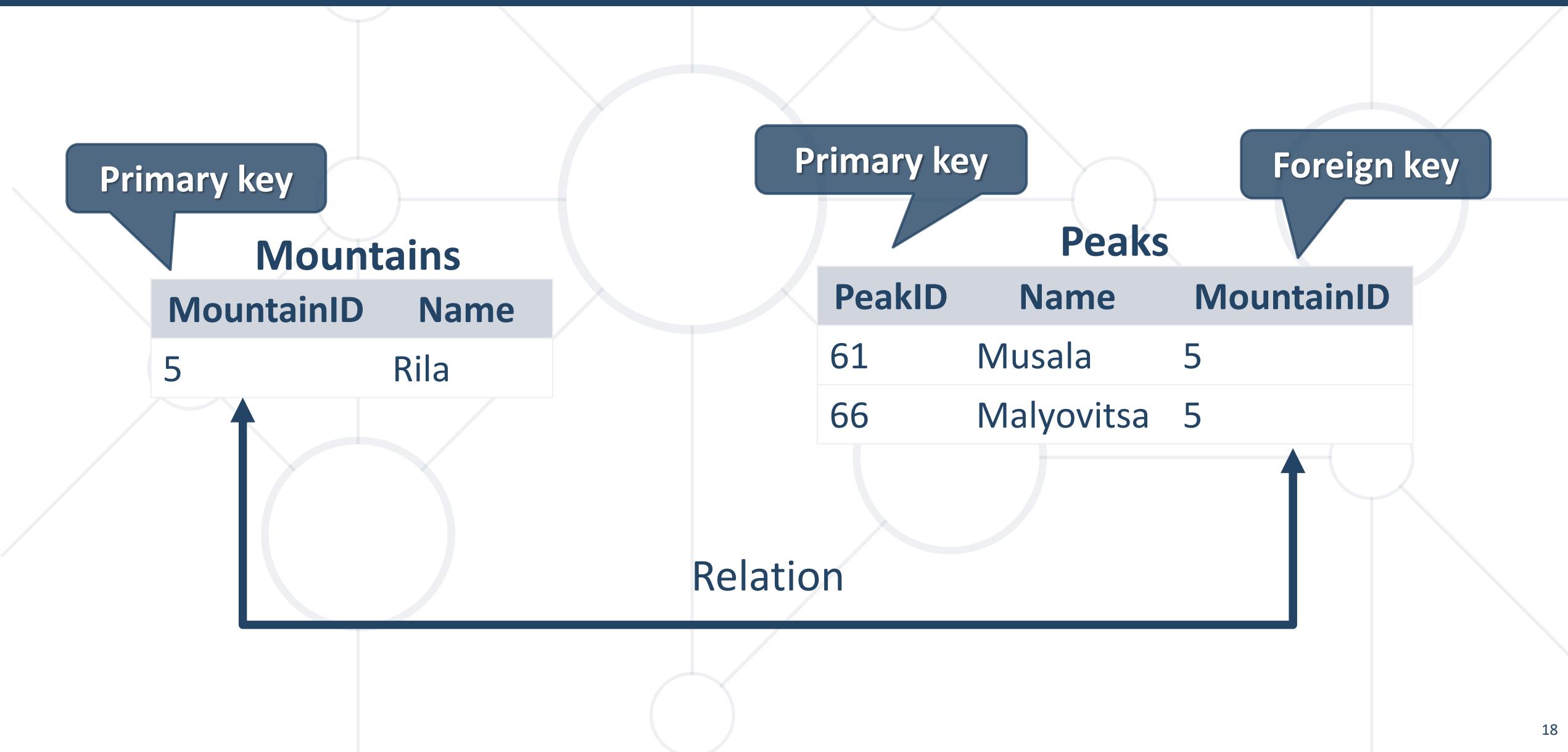
Id	Name
1	Bulgaria
2	Germany

Table Relations: Multiplicity

- **One-to-many** – e.g. country / towns
 - One country has many towns
- **Many-to-many** – e.g. student / course
 - One student has many courses
 - One course has many students
- **One-to-one** – e.g. example driver / car
 - One driver has only one car
 - Rarely used



One-to-Many/Many-to-One



One-to-Many: Tables

```
CREATE TABLE Mountains(  
    MountainID INT PRIMARY KEY,  
    MountainName VARCHAR(50)  
)
```

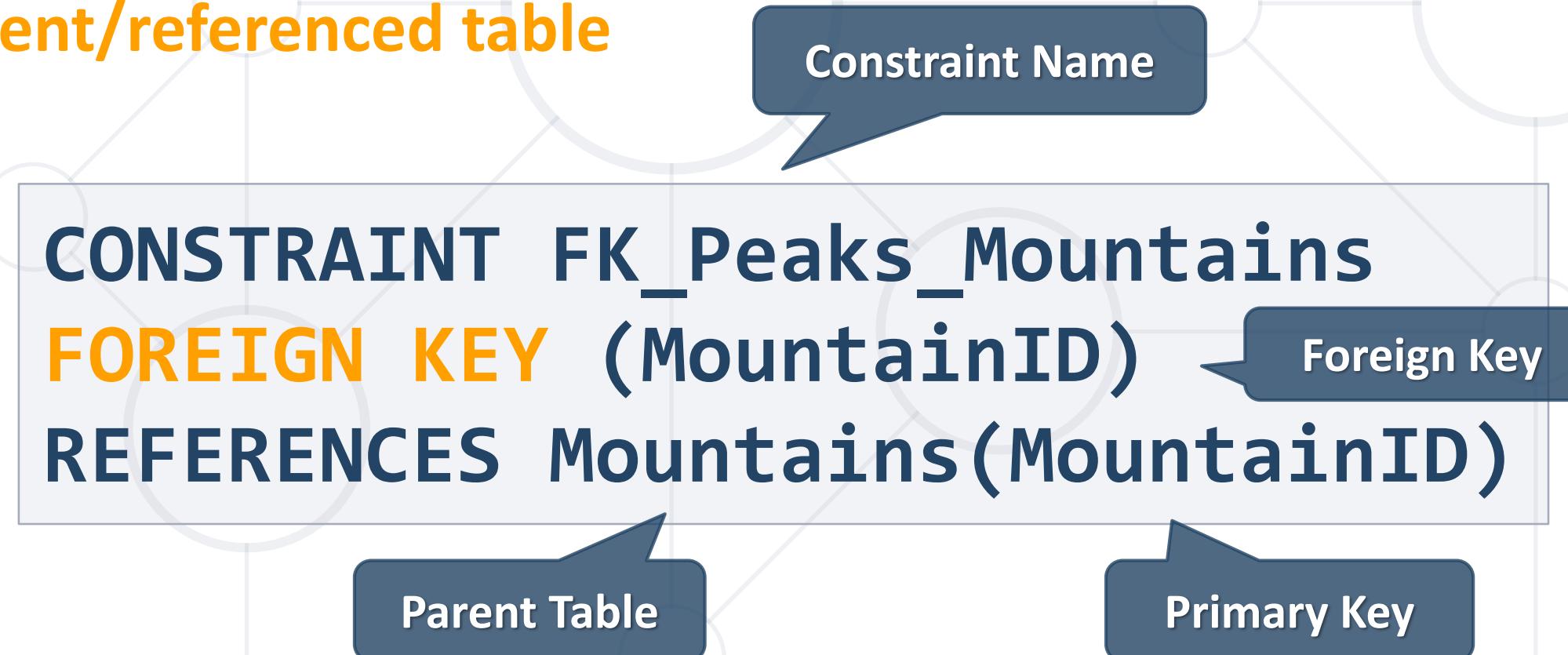
Primary key

```
CREATE TABLE Peaks(  
    PeakId INT PRIMARY KEY,  
    MountainID INT,  
    CONSTRAINT FK_Peaks_Mountains  
    FOREIGN KEY (MountainID)  
    REFERENCES Mountains(MountainID)  
)
```

Foreign Key

One-to-Many: Foreign Key

- The table holding the **foreign key** is the **child table**
- The table holding the **referenced primary key** is the **parent/referenced table**



Many-to-Many

- Many-to-many relations use a mapping/join table

Primary key

Employees

EmployeeID	EmployeeName
------------	--------------

1
...
40
...

Mapping table

Primary key

Projects

ProjectID	ProjectName
-----------	-------------

4
...
24
...

EmployeesProjects

EmployeeID	ProjectID
------------	-----------

1	4
1	24
40	24

Many-to-Many: Tables

```
CREATE TABLE Employees(  
    EmployeeID INT PRIMARY KEY,  
    EmployeeName VARCHAR(50)  
)
```

```
CREATE TABLE Projects(  
    ProjectID INT PRIMARY KEY,  
    ProjectName VARCHAR(50)  
)
```

Many-to-Many: Mapping Table

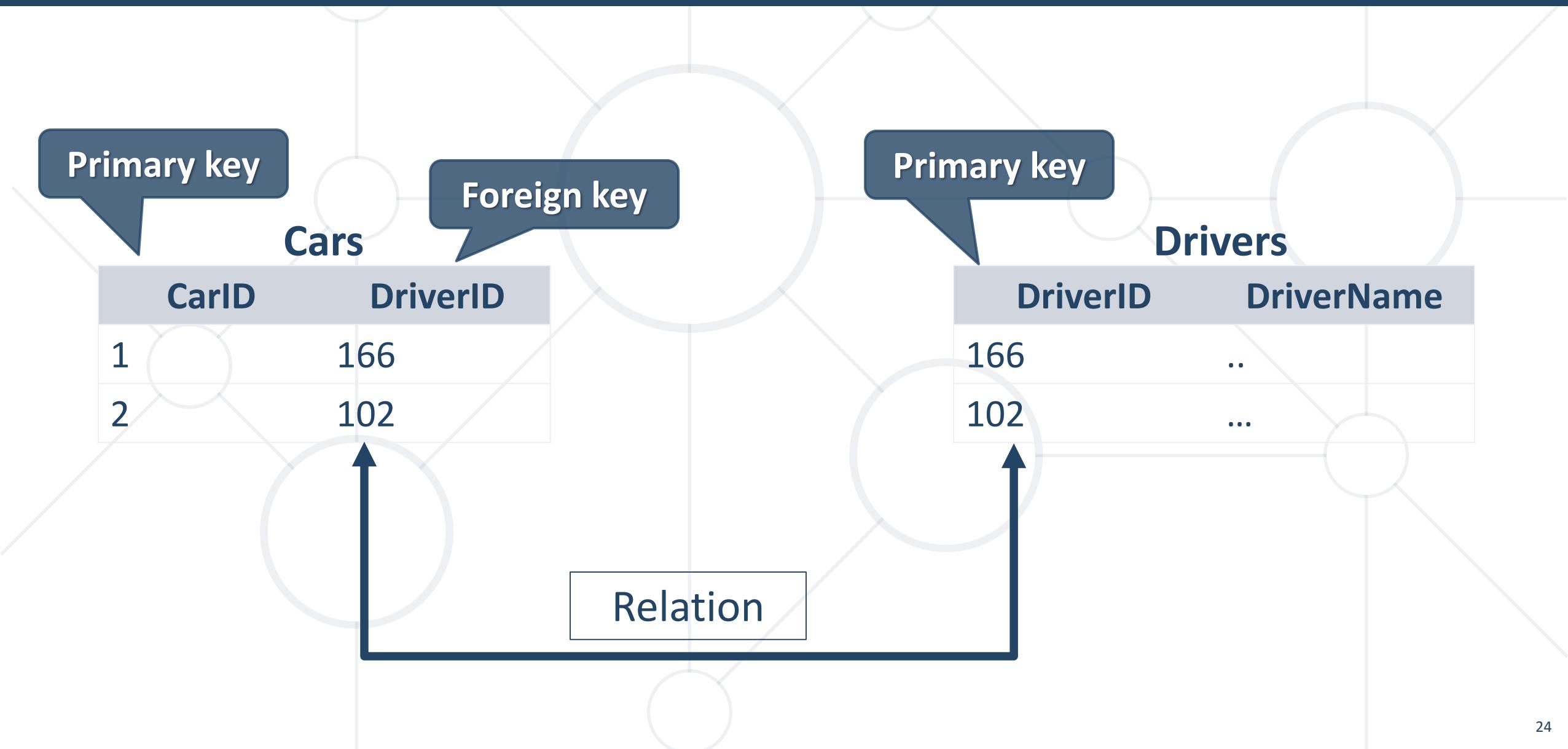
```
CREATE TABLE EmployeesProjects(  
    EmployeeID INT,  
    ProjectID INT,  
    CONSTRAINT PK_EmployeesProjects  
        PRIMARY KEY(EmployeeID, ProjectID),  
    CONSTRAINT FK_EmployeesProjects_Employees  
        FOREIGN KEY(EmployeeID)  
        REFERENCES Employees(EmployeeID),  
    CONSTRAINT FK_EmployeesProjects_Projects  
        FOREIGN KEY(ProjectID)  
        REFERENCES Projects(ProjectID)  
)
```

Composite Primary Key

Foreign Key to Employees

Foreign Key to Projects

One-to-One



```
CREATE TABLE Drivers(  
    DriverID INT PRIMARY KEY,  
    DriverName VARCHAR(50)  
)
```

Primary key

```
CREATE TABLE Cars(  
    CarID INT PRIMARY KEY,  
    DriverID INT UNIQUE,  
    CONSTRAINT FK_Cars_Drivers FOREIGN KEY  
        (DriverID) REFERENCES Drivers(DriverID)  
)
```

One driver
per car

Foreign Key

One-to-One: Foreign Key

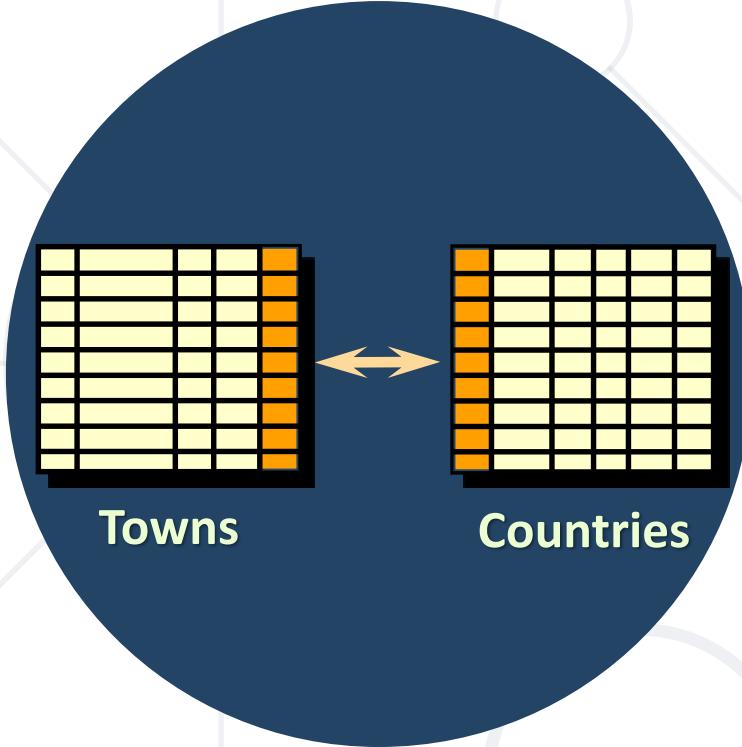
```
CONSTRAINT FK_Cars_Drivers  
FOREIGN KEY (DriverID) REFERENCES Drivers(DriverID)
```

Constraint Name

Foreign Key

Referenced Table

Primary Key



Retrieving Related Data

Using Simple JOIN Statements

JOIN Statements

- With a **JOIN** statement, we can get data from two tables **simultaneously**
 - JOINS** require at least two tables and a "**join condition**"

```
SELECT * FROM Towns
JOIN Countries ON
    Countries.Id = Towns.CountryId
```

Join Condition

Problem: Peaks in Rila

- Use database "Geography". Report all peaks for "Rila" mountain.
 - Report includes mountain's name, peak's name and also peak's elevation
 - Peaks should be **sorted** by elevation descending

	MountainRange	PeakName	Elevation
1	Rila	Musala	2925
2	Rila	Malka Musala	2902
3	Rila	Malyovitsa	2729
4	Rila	Orlovets	2685

Check your solution here: <https://judge.softuni.org/Contests/Compete/Index/292#6>

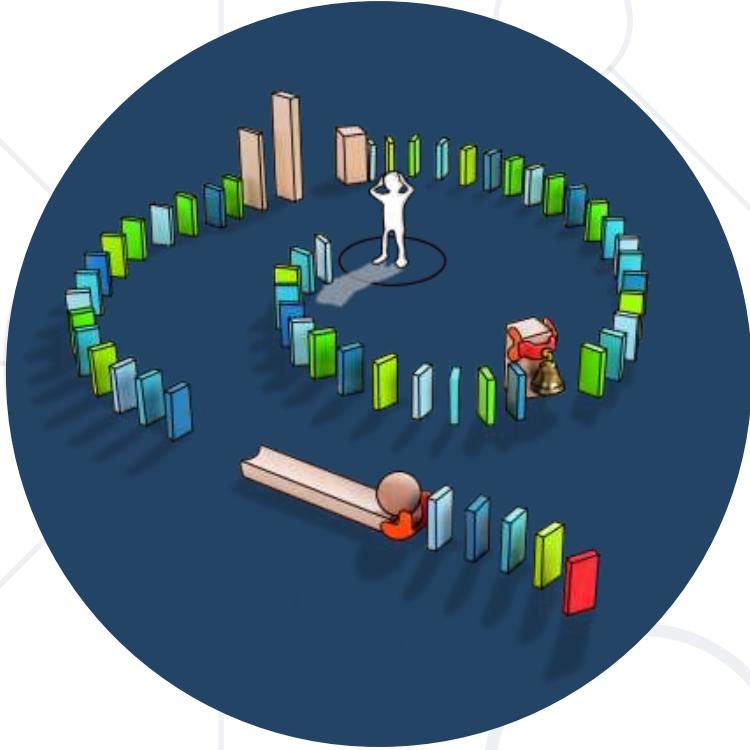
Solution: Peaks in Rila

Cross Table Selection

```
SELECT m.MountainRange, p.PeakName, p.Elevation  
FROM Mountains AS m  
JOIN Peaks As p ON p.MountainId = m.Id  
WHERE m.MountainRange = 'Rila'  
ORDER BY p.Elevation DESC
```

Join Condition

Check your solution here: <https://judge.softuni.org/Contests/Compete/Index/292#6>

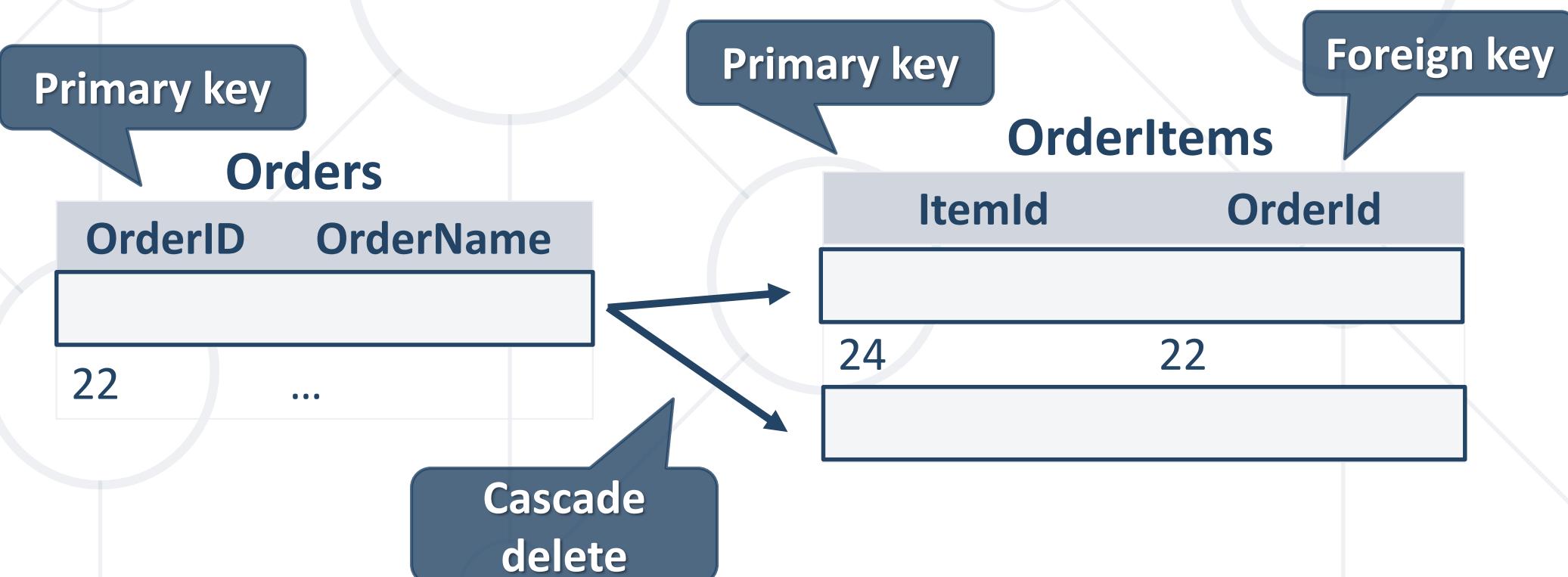


Cascade Operations

Cascade Delete/Update

Definition

- Cascading allows when a change is made to certain entity, this change to apply to all related entities



Cascade Delete

- Cascade can be either Delete or Update
- Use Cascade Delete when:
 - The related entities are meaningless without the "main" one
- Do not use Cascade Delete when:
 - You perform a "logical delete"
 - Entities are marked as deleted (but not actually deleted)
 - In more complicated relations, cascade delete won't work with circular references

Cascade Update

- Use **Cascade Update** when:
 - The primary key is not identity (not auto-increment) and therefore it **can** be changed
 - Best used with unique constraint
- Do **not** use **Cascade Update** when:
 - The primary is identity (auto-increment)
 - Cascading can be avoided using **triggers** or **procedures**

Cascade Delete: Example

```
CREATE TABLE Drivers(  
    DriverID INT PRIMARY KEY,  
    DriverName VARCHAR(50)  
)
```

```
CREATE TABLE Cars(  
    CarID INT PRIMARY KEY,  
    DriverID INT,  
    CONSTRAINT FK_Car_Driver FOREIGN KEY(DriverID)  
    REFERENCES Drivers(DriverID) ON DELETE CASCADE  
)
```

Foreign Key

Cascade

Cascade Update: Example

```
CREATE TABLE Products(  
    BarcodeId INT PRIMARY KEY,  
    Name VARCHAR(50)  
)
```

```
CREATE TABLE Stock(  
    Id INT PRIMARY KEY,  
    Barcode INT,  
    CONSTRAINT FK_Stock_Products FOREIGN KEY(BarcodeId)  
    REFERENCES Products(BarcodeId) ON UPDATE CASCADE  
)
```

Foreign Key

Cascade



E/R Diagrams

Entity / Relationship Diagrams

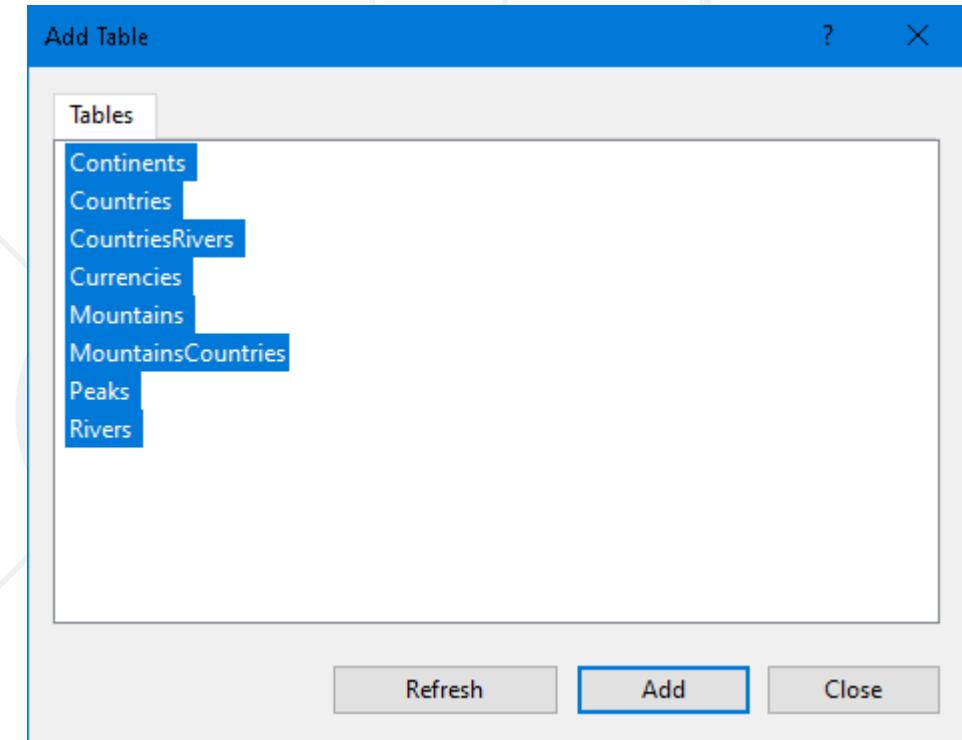
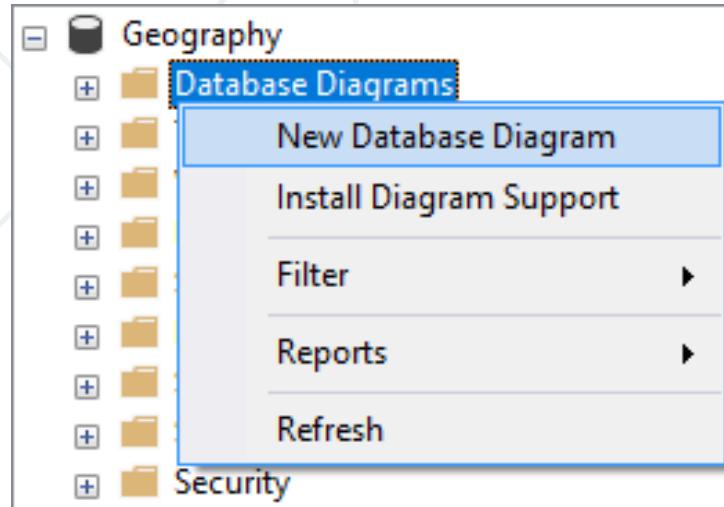
Relational Schema

- **Relational schema** of a DB is the collection of:
 - The schemas of all tables
 - Relationships between the tables
 - Any other database objects (e.g. constraints)
- The relational schema describes the structure of the database
 - Doesn't contain data, but metadata
- Relational schemas are graphically displayed in Entity / Relationship diagrams (**E/R Diagrams**)

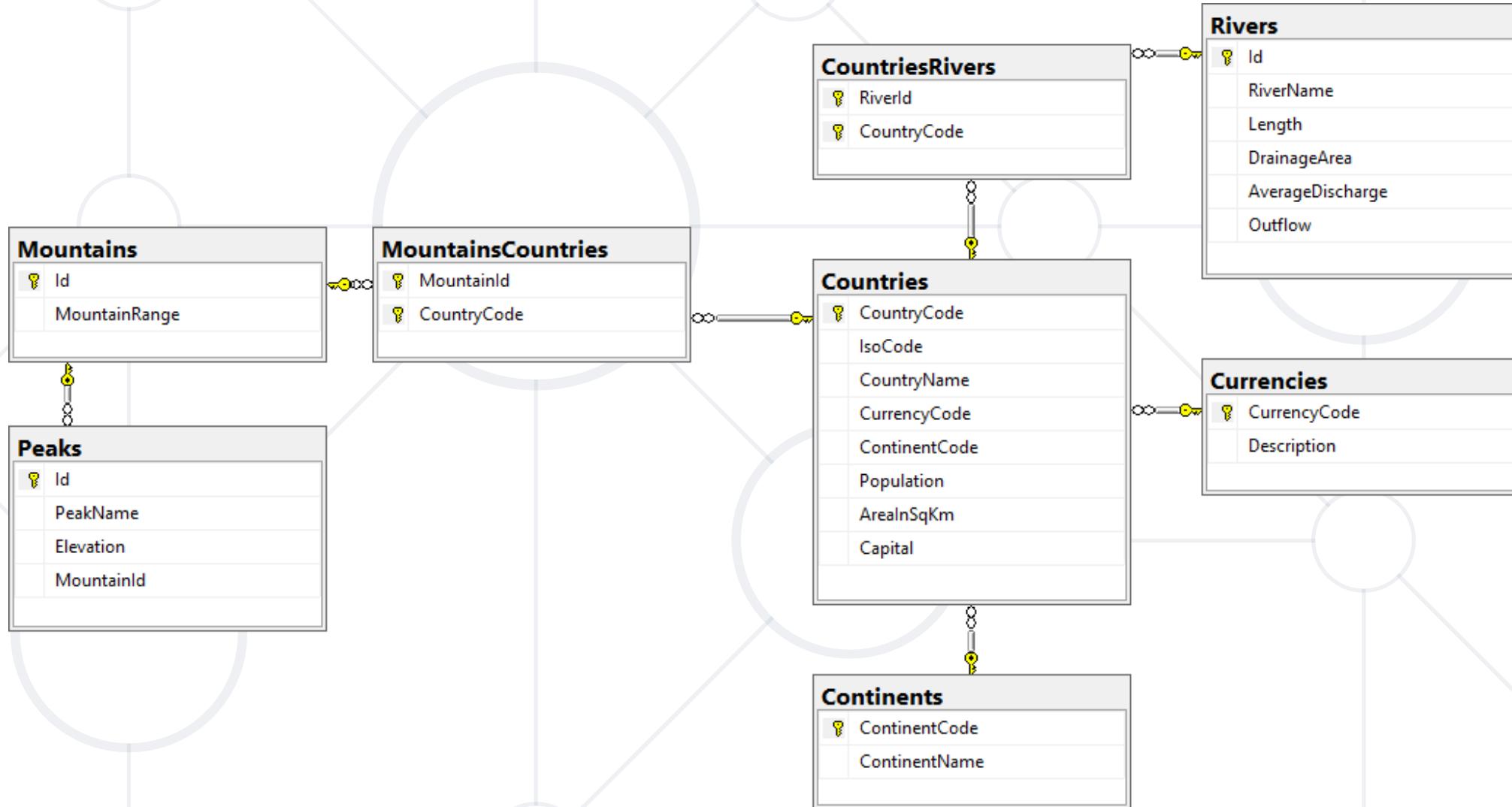


SSMS E/R Diagram: Usage

- Expand a database in **Object Explorer**
 - Right click "Database Diagrams" then select "**New Database Diagram**"



SSMS E/R Diagram

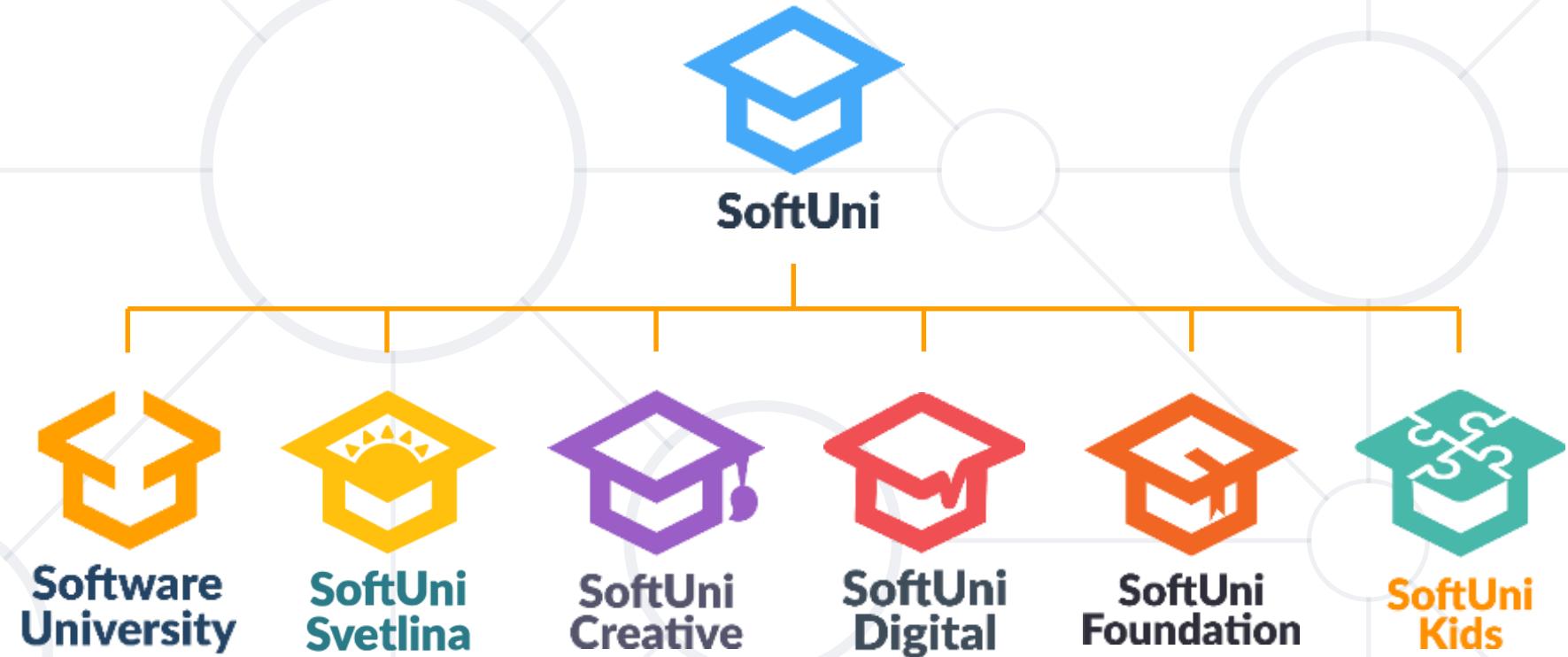


Summary

- Design **multiple** tables with related data
- Types of table relations
- **Cascading** – Pros and Cons
- Entity / Relationship **Diagrams**



Questions?



SoftUni Diamond Partners



SCHWARZ



Coca-Cola HBC
Bulgaria



Postbank

Решения за твоето упре



Bosch..IO****



INDEAVR
Serving the high achievers



**SUPER
HOSTING
.BG**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



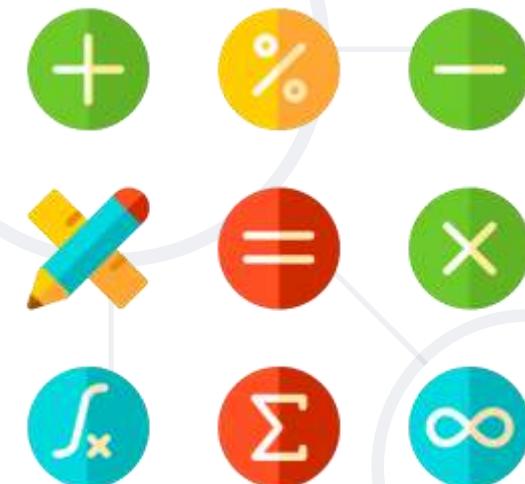
Built-in Functions

Functions and Wildcards in SQL Server

SoftUni Team
Technical Trainers



SoftUni



Software University
<https://softuni.bg>

Table of Contents

1. Function Overview
2. String Functions
3. Math Functions
4. Date Functions
5. Other Useful Functions
6. Wildcards



sli.do

#csharp-db



Functions in SQL Server

Overview

SQL Functions

■ Aggregate functions

- Perform a calculation on a set of values and return a single value
- Examples: **AVG, COUNT, MIN, MAX, SUM**

■ Analytic functions

- Compute an aggregate value based on a group of rows
- Unlike aggregate functions, analytic functions can return multiple rows for each group

```
PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY Salary DESC)  
OVER (PARTITION BY DepartmentId) AS MedianCont
```



SQL Functions

- **Ranking functions**

- Return a ranking value for each row in a partition
- **RANK, ROW_NUMBER, DENSE_RANK, NTILE (OVER)**

- **Rowset functions**

- Return an object that can be used like table references in a statement
- **OPENDATASOURCE, OPENJSON, OPENXML, OPENROWSET**

- **Scalar functions**

- Operate on a single value and then return a single value
- Scalar functions can be used wherever an expression is valid





String Functions

String Functions (1)

- **Concatenation** – combines strings

```
SELECT FirstName + ' ' + LastName  
      AS [Full Name]  
  FROM Employee
```

```
SELECT CONCAT(FirstName, ' ', LastName)  
      AS [Full Name]  
  FROM Employee
```

- **CONCAT** replaces **NULL** values with **empty string**
- **CONCAT_WS** combines strings with separator

String Functions (2)

- **SUBSTRING** – extracts a part of a string

```
SUBSTRING(String, StartIndex, Length)
```

```
SUBSTRING('SoftUni', 5, 3)
```



Uni

- Example: get short **summary** of an article

```
SELECT ArticleId, Author, Content,  
       SUBSTRING(Content, 1, 200) + '...' AS Summary  
FROM Articles
```

String Functions (3)

- **REPLACE** – replaces a specific string with another

```
REPLACE(String, Pattern, Replacement)
```

```
REPLACE('SoftUni', 'Soft', 'Hard')
```

HardUni

- Example: **censor** the word blood from album names

```
SELECT REPLACE(Title, 'blood', '*****')  
AS Title  
FROM Album
```

String Functions (4)

- **LTRIM & RTRIM** – remove spaces from either side of string

LTRIM(String)

RTRIM(String)

- **LEN** – counts the number of characters

LEN(String)

- **DATALENGTH** – gets the number of used bytes

DATALENGTH(String)

String Functions (5)

- **LEFT & RIGHT** – get characters from the beginning or the end of a string

LEFT(String, Count)

RIGHT(String, Count)

- Example: name **shortened** (first 3 letters)

```
SELECT Id, Start,  
       LEFT(Name, 3) AS Shortened  
  FROM Games
```

String Functions (6)

- **LOWER & UPPER** – change letter casing

LOWER(String)

UPPER(String)

- **REVERSE** – reverses order of all characters in a string

REVERSE(String)

- **REPLICATE** – repeats a string

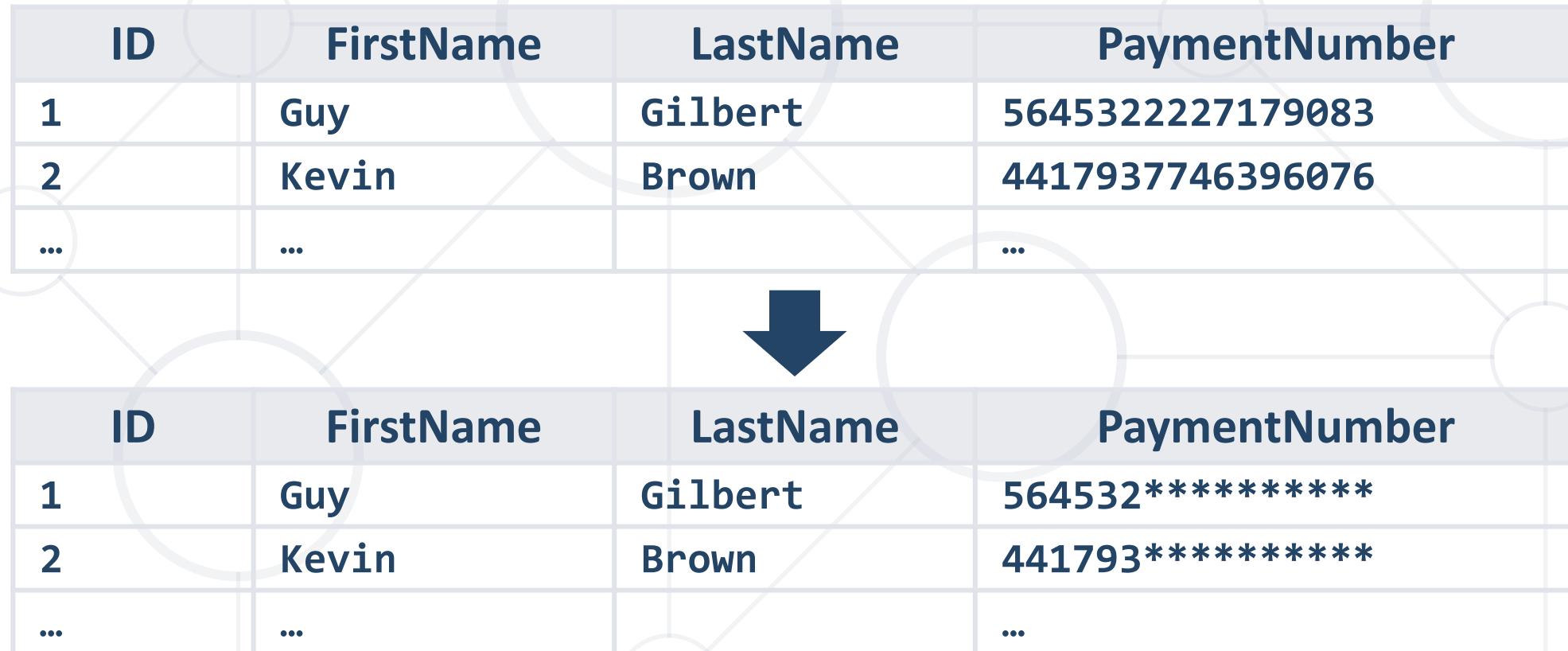
REPLICATE(String, Count)

- **FORMAT** – format a value with a valid .NET format string

FORMAT(SomeDate, 'yyyy-MMMM-dd', 'bg-BG')

Problem: Obfuscate CC Numbers

- The database contains credit card details for customers
- Provide a summary without revealing the serial numbers



The diagram illustrates a data transformation process. At the top, there is a full database table with columns: ID, FirstName, LastName, and PaymentNumber. The data includes rows for customers with IDs 1 and 2, and an ellipsis row. The PaymentNumber column contains full credit card numbers: 5645322227179083 and 4417937746396076. A large blue arrow points downwards, indicating a transformation or summary step. Below the arrow is a second table, which is a summary version of the first. It contains the same four columns: ID, FirstName, LastName, and PaymentNumber. The data is identical to the first table, but the PaymentNumber values have been partially obscured by asterisks (***) to represent obfuscation. Ellipsis rows are present in both tables.

ID	FirstName	LastName	PaymentNumber
1	Guy	Gilbert	5645322227179083
2	Kevin	Brown	4417937746396076
...

ID	FirstName	LastName	PaymentNumber
1	Guy	Gilbert	564532*****
2	Kevin	Brown	441793*****
...

Solution : Obfuscate CC Numbers

- Reveal the first 6 digits and obfuscate the rest

```
SELECT CustomerID,  
       FirstName,  
       LastName,  
       LEFT(PaymentNumber, 6) + '*****'  
FROM Customers
```

- Bonus – create a View for the use of clients

```
CREATE VIEW v_PublicPaymentInfo AS
```

```
...
```

String Functions (7)

- **CHARINDEX** – locates a specific pattern (substring) in a string

Optional, begins at 1

CHARINDEX(Pattern, String, [StartIndex])

- **STUFF** – inserts a substring at a specific position

STUFF(String, StartIndex, Length, Substring)

Number of chars
to delete



Math Functions

Arithmetic, PI, ABS, ROUND, Etc.

Math Functions (1)

- SQL Server supports **basic arithmetic operations**
- Example: find the area of triangles by the given side and height

Id	A	H
1	2	4
2	1	18
3	4.5	3
4	8	12
5	3	5



Id	Area
1	4
2	9
3	6.75
4	48
5	7.5

```
SELECT Id,  
       (A*H)/2 AS Area  
  FROM Triangles2
```



Math Functions (2)

- **PI** – gets the value of Pi as a float (15 –digit precision)

```
SELECT PI() --3.14159265358979
```

- **ABS** – absolute value

```
ABS(Value)
```

- **SQRT** – square root (the result will be float)

```
SQRT(Value)
```

- **SQUARE** – raise to power of two

```
SQUARE(Value)
```

Example: Line Length

- Find the length of a line by given coordinates of the end points

Id	X1	Y1	X2	Y2
1	0	0	10	0
2	0	0	5	3
4	-1	5	8	-3
5	18	23	8882	134



Id	Length
1	10
2	5.8309518948453
4	12.0415945787923
5	8864.69497501183

```
SELECT Id,  
       SQRT(SQUARE(X1-X2) + SQUARE(Y1-Y2))  
  AS Length  
FROM Lines
```

Math Functions (3)

- **POWER** – raises value to the desired exponent

POWER(Value, Exponent)

- **ROUND** – obtains the desired precision

- Negative precision rounds characters before the decimal point

ROUND(Value, Precision)

- **FLOOR & CEILING** – return the nearest integer

FLOOR(Value)

CEILING(Value)

Problem: Pallets

- Calculate the required number of pallets to ship each item
 - BoxCapacity** specifies how many items can fit in one box
 - PalletCapacity** specifies how many boxes can fit in a pallet

Id	Name	Quantity	BoxCapacity	PalletCapacity
1	Perlenbacher 500ml	108	6	18
2	Perlenbacher 500ml	10	6	18
3	Chocolate Chips	350	24	3
4	Oil Pump	100	1	12
5	OLED TV 50-Inch	13	1	5
6	Penny	1	2239488	1



Number of pallets
1
1
5
9
3
1

Solution: Pallets

- Since we can't use half a box or half a pallet, we need to round up to the nearest integer value

```
SELECT  
    CEILING(  
        CEILING(  
            CAST(Quantity AS float) /  
            BoxCapacity) / PalletCapacity)  
    AS [Number of pallets]  
FROM Products
```

- **SIGN** – returns 1, -1 or 0, depending on the value of the sign

SIGN(Value)

- **RAND** – gets a random float value in the range [0, 1]

- If Seed is not specified, it will be assigned randomly

RAND()

RAND(Seed)



Date Functions

GETDATE, DATEDIFF, DATEPART, Etc.

Date Functions (1)

- **DATEPART** – extract a segment from a date as an integer
 - Part can be any part and format of date or time

DATEPART(Part, Date)

year, yyyy, yy

month, mm, m

day, dd, d

YEAR(Date)

MONTH(Date)

DAY(Date)

- For a full list, take a look at the [official documentation](#)

Problem: Quarterly Report

- Prepare sales data for aggregation by displaying yearly quarter, month, year and day of sale

Invoiceld	InvoiceDate	Total
1	2023-01-01	1.98
2	2023-01-02	3.96
3	2023-01-03	5.94
4	2023-01-06	8.91



Invoiceld	Total	Quarter	Month	Year	Day
1	1.98	1	1	2023	1
2	3.96	1	1	2023	2
3	5.94	1	1	2023	3
4	8.91	1	1	2023	6

Solution: Quarterly Report

- Use **DATEPART** to get the relevant parts of the date

```
SELECT InvoiceId, Total,  
       DATEPART(QUARTER, InvoiceDate) AS Quarter,  
       DATEPART(MONTH, InvoiceDate) AS Month,  
       DATEPART(YEAR, InvoiceDate) AS Year,  
       DATEPART(DAY, InvoiceDate) AS Day  
FROM Invoice
```

- This statement might be useful as a View

Date Functions (2)

- **DATEDIFF** – finds the difference between two dates
 - Part can be **any part** and **format** of date or time

```
DATEDIFF(Part, FirstDate, SecondDate)
```

- Example: Show employee experience

```
SELECT ID, FirstName, LastName,  
       DATEDIFF(YEAR, HireDate, '2017/01/25')  
     AS [Years In Service]  
   FROM Employees
```

Date Functions (3)

- **DATENAME** – gets a string representation of a date's part

```
DATENAME(Part, Date)
```

```
SELECT DATENAME(weekday, '2017/01/27')
```

- **DATEADD** – performs date arithmetic

- Part can be **any part** and **format** of date or time

```
DATEADD(Part, Number, Date)
```

- **GETDATE** – obtains the current date and time

```
SELECT GETDATE()
```

- **EOMONTH** – returns the last day of the month



Other Functions

CAST, CONVERT, OFFSET, FETCH

Other Functions (1)

- **CAST & CONVERT** – conversion between data types

```
CAST(Data AS NewType)
```

```
CONVERT(NewType, Data)
```

- **ISNULL** – swaps **NULL** values with a specified **default value**

```
ISNULL(Data, DefaultValue)
```

- Example: Display "Not Finished" for projects with **no EndDate**

```
SELECT ProjectID, Name,  
       ISNULL(CAST(EndDate AS varchar), 'Not Finished')  
FROM Projects
```

Other Functions (2)

- **COALESCE** – evaluates the arguments in order and returns the current value of the first expression that initially does not evaluate to **NULL**

```
SELECT COALESCE(NULL, NULL, 'third_value',
'fourth_value');

// third_value
```

Other Functions (3)

- **OFFSET & FETCH** – get only specific rows from the result set
 - Used in combination with **ORDER BY** for pagination

```
SELECT ID, FirstName, LastName  
      FROM Employees  
  ORDER BY ID  
  OFFSET 10 ROWS  
  FETCH NEXT 5 ROWS ONLY
```

Rows to skip

Rows to include

Ranking Functions

- **ROW_NUMBER** – always generate unique values without any gaps, even if there are ties
- **RANK** – can have gaps in its sequence and when values are the same, they get the same rank
- **DENSE_RANK** – returns the same rank for ties, but it doesn't have any gaps in the sequence
- **NTILE** – Distributes the rows in an ordered partition into a specified number of groups



Wildcards

Selecting Results by Partial Match

Using WHERE ... LIKE

- Wildcards are used with WHERE for partial filtration
- Similar to Regular Expressions, but less capable
- Example: Find all employees who's first name starts with "Ro"

```
SELECT ID, FirstName, LastName  
FROM Employees  
WHERE FirstName LIKE 'Ro%'
```

Wildcard symbol

Wildcard Characters

- Supported characters include:

%	-- any string, including zero-length
_	-- any single character
[...]	-- any character within range
[^...]	-- any character not in the range

- **ESCAPE** – specify a prefix to treat special characters as normal

```
SELECT ID, Name
  FROM Tracks
 WHERE Name LIKE '%max!%' ESCAPE '!'
```

Summary

- Various **built-in functions**
- String functions - **CONCAT, LEFT/RIGHT, REPLACE**, etc.
- Math functions - **PI, ABS, POWER, ROUND**, etc.
- Date functions - **DATEPART, DATEDIFF, GETDATE**, etc.
- Using **Wildcards**, we can obtain results by partial string matches



SoftUni Diamond Partners



SCHWARZ



Coca-Cola HBC
Bulgaria



Postbank

Решения за твоето упре



**SOFTWARE
GROUP**

Bosch.**.IO**



SmartIT

POKERSTARS

CAREERS

AMBITIONED

INDEAVR
Serving the high achievers

createX

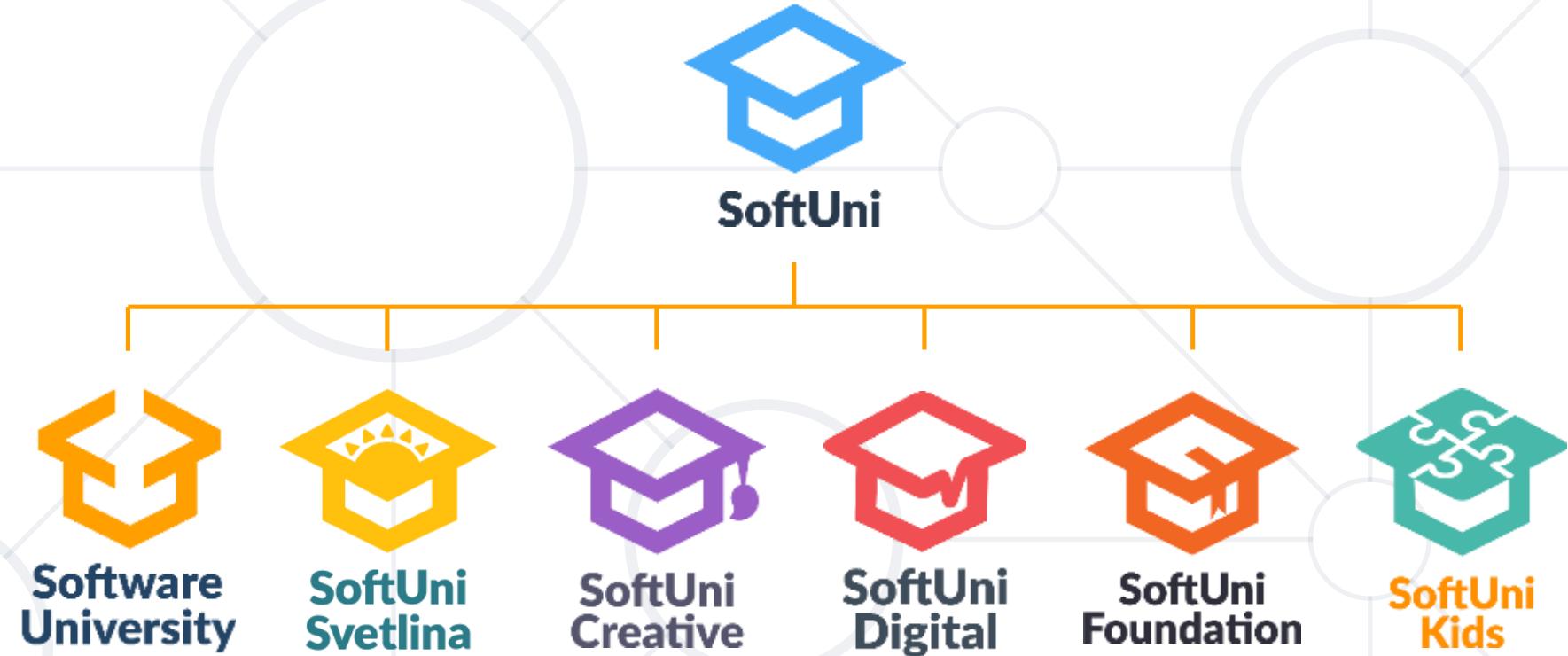
**DRAFT
KINGS**

**SUPER
HOSTING
.BG**

Educational Partners



Questions?



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



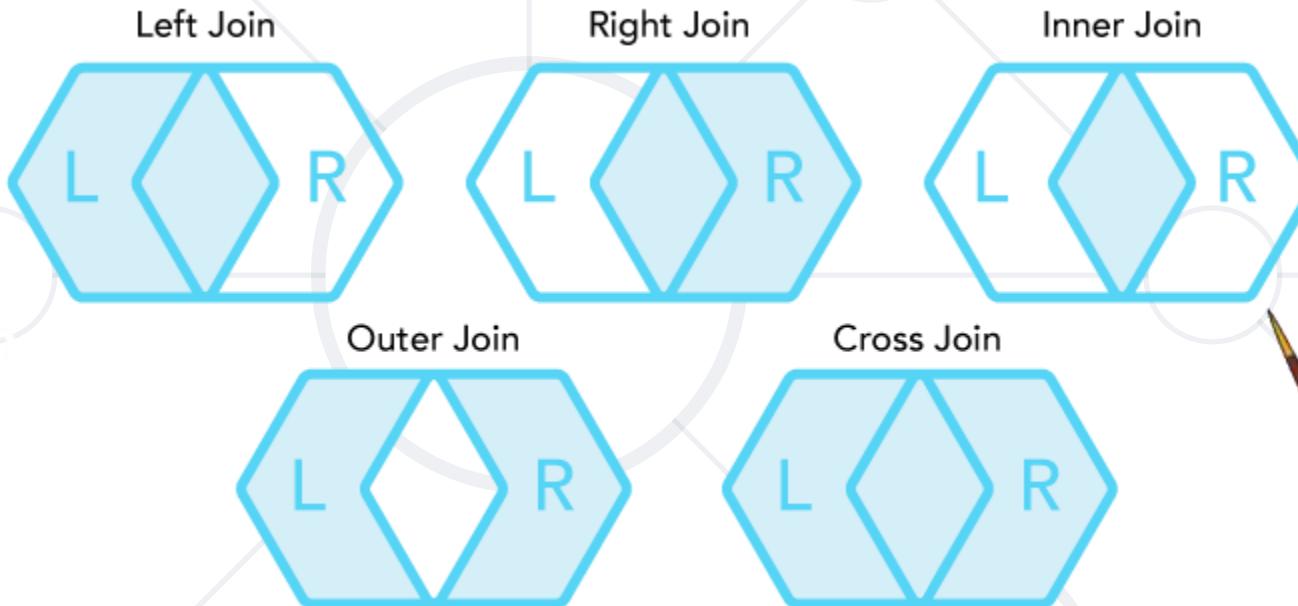
Software
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Joins, Subqueries, CTEs



SoftUni Team

Technical Trainers



SoftUni



Software University
<https://softuni.bg>

Table of Contents

1. Joins
2. Subqueries
3. Common Table Expressions (CTE)
4. Temporary Tables



sli.do

#csharp-db



JOINS

Gathering Data from Multiple Tables

Data from Multiple Tables

- Sometimes you need data from **several tables**



Employees

EmployeeName	DepartmentID
Edward	3
John	NULL

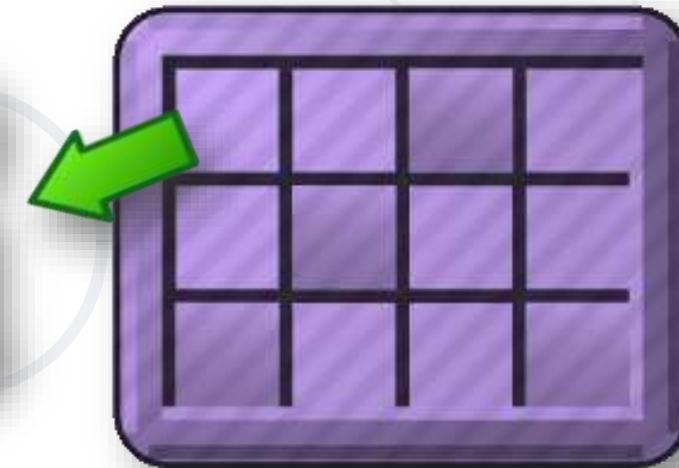
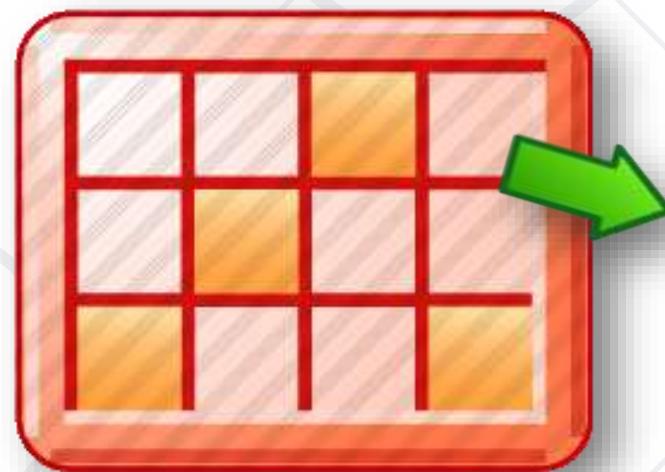
Departments

DepartmentID	DepartmentName
3	Sales
4	Marketing
5	Purchasing

EmployeeName	DepartmentID	DepartmentName
Edward	3	Sales

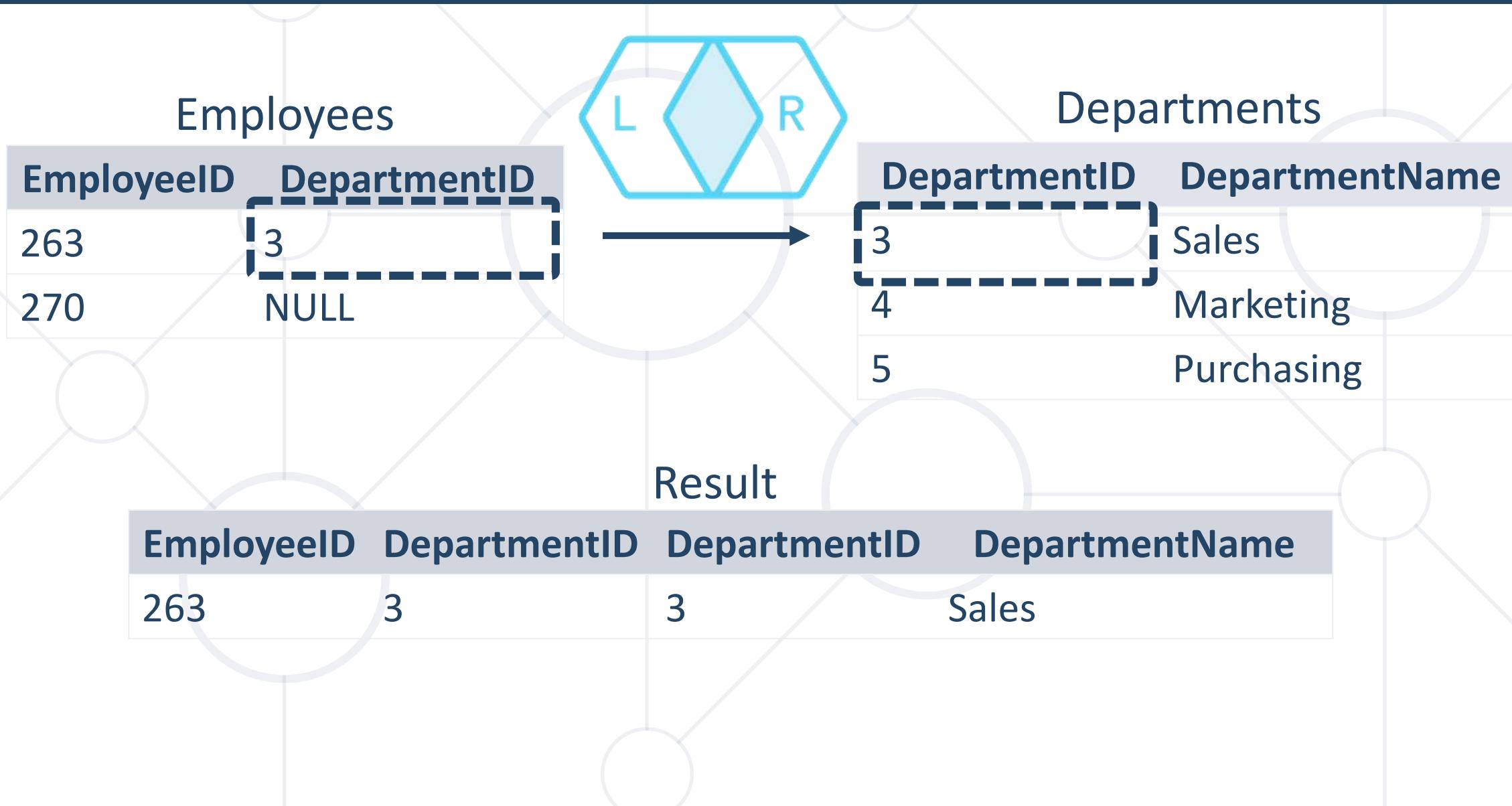
Types of Joins

- Inner joins
- Left, right and full outer joins
- Cross joins



- **Inner join**
 - Join of two tables returning **only rows matching** the join **condition**
- **Left (or right) outer join**
 - Returns the results of the inner join as well as unmatched rows from the left (or right) table
- **Full outer join**
 - Returns the results of an **inner join** along with all **unmatched rows**

Inner Join



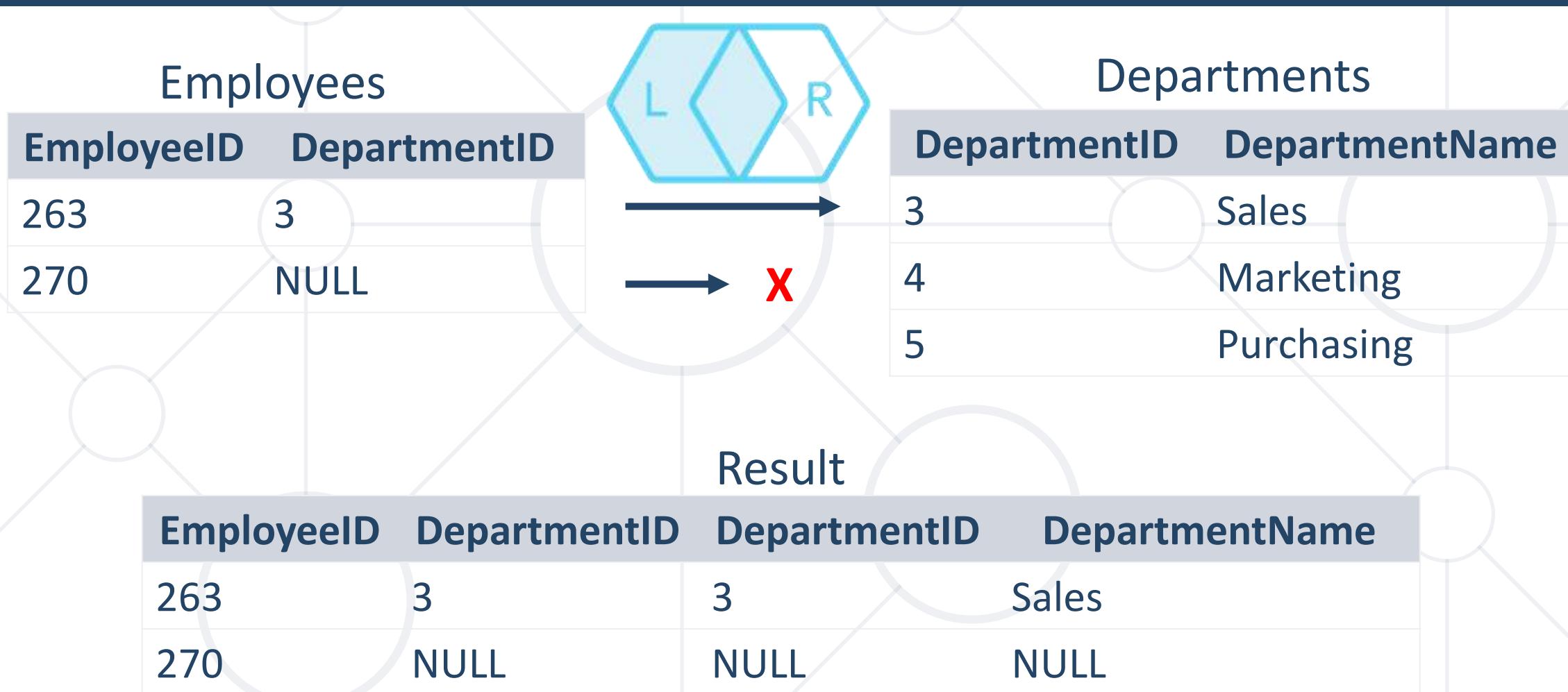
Inner Join Syntax

```
SELECT * FROM Employees AS e  
INNER JOIN Departments AS d  
ON e.DepartmentID = d.DepartmentID
```

Departments Table

Join Condition

Left Outer Join



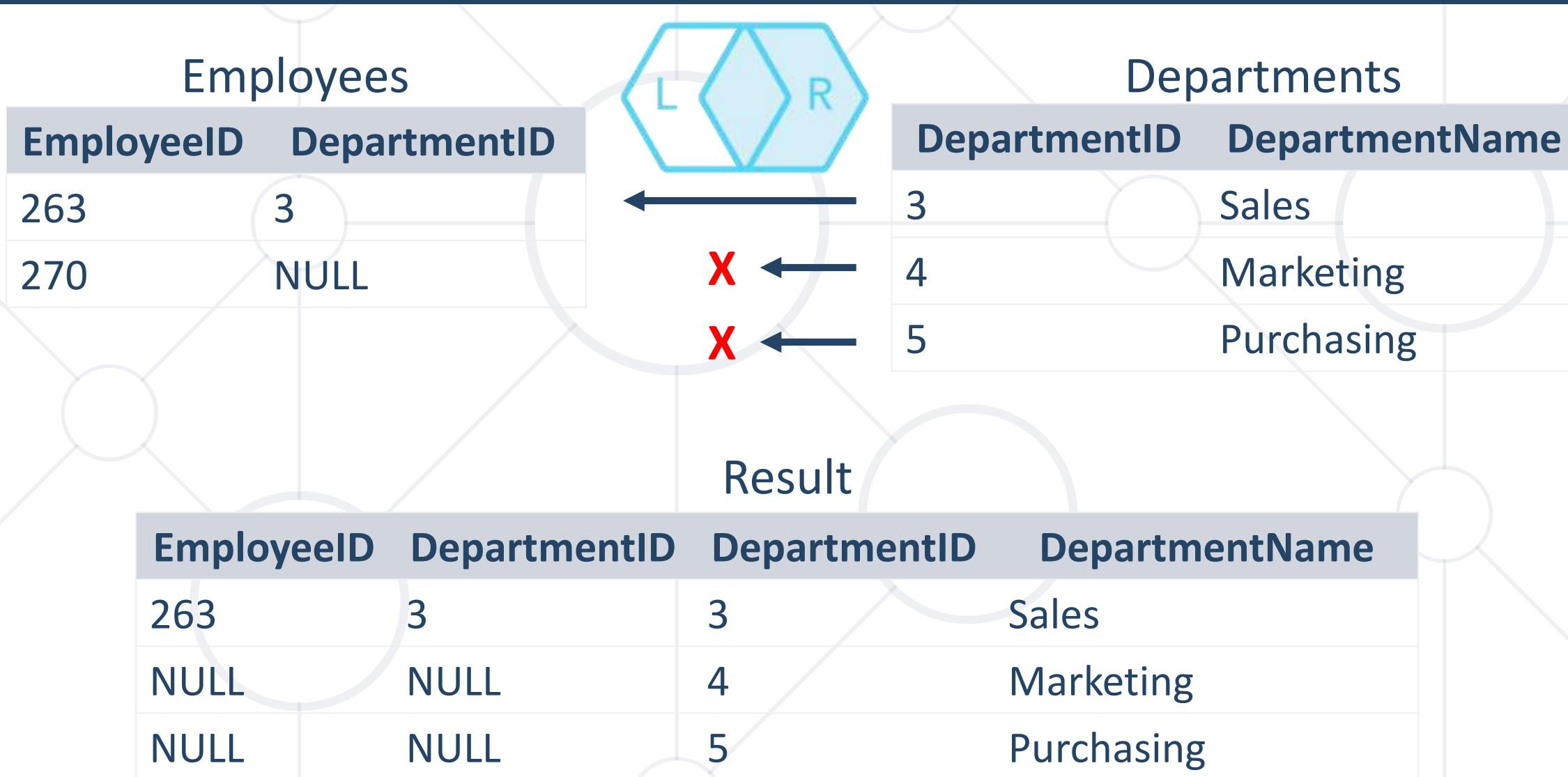
Left Outer Join Syntax

```
SELECT * FROM Employees AS e  
LEFT OUTER JOIN Departments AS d  
ON e.DepartmentID = d.DepartmentID
```

Table Departments

Join Condition

Right Outer Join



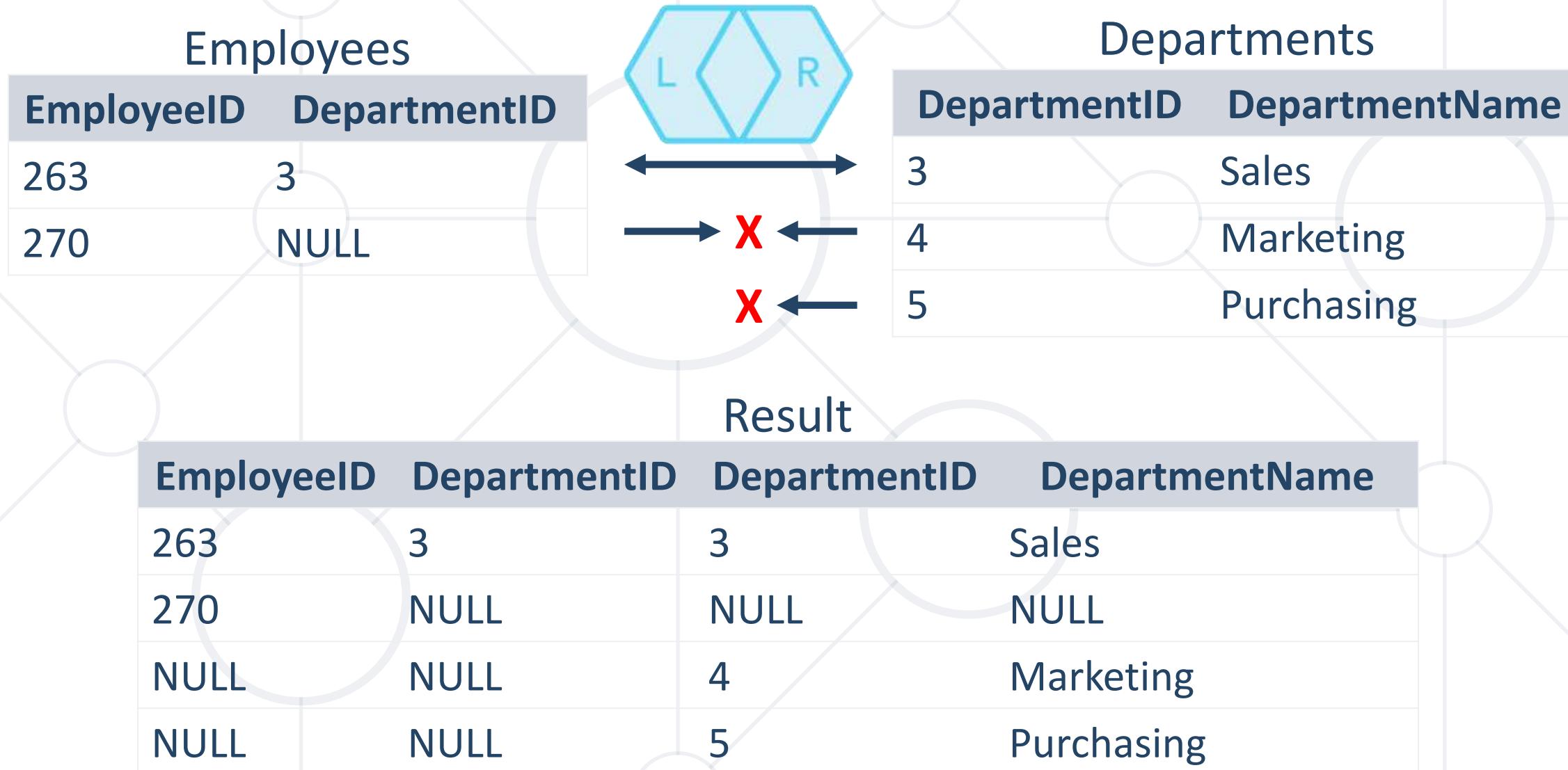
Right Outer Join Syntax

```
SELECT * FROM Employees AS e  
RIGHT OUTER JOIN Departments AS d  
ON e.DepartmentID = d.DepartmentID
```

Departments Table

Join Condition

Full Join



Full Join Syntax

```
SELECT * FROM Employees AS e  
FULL JOIN Departments AS d  
ON e.DepartmentID = d.DepartmentID
```

Departments Table

Join Condition

Cartesian Product (1)

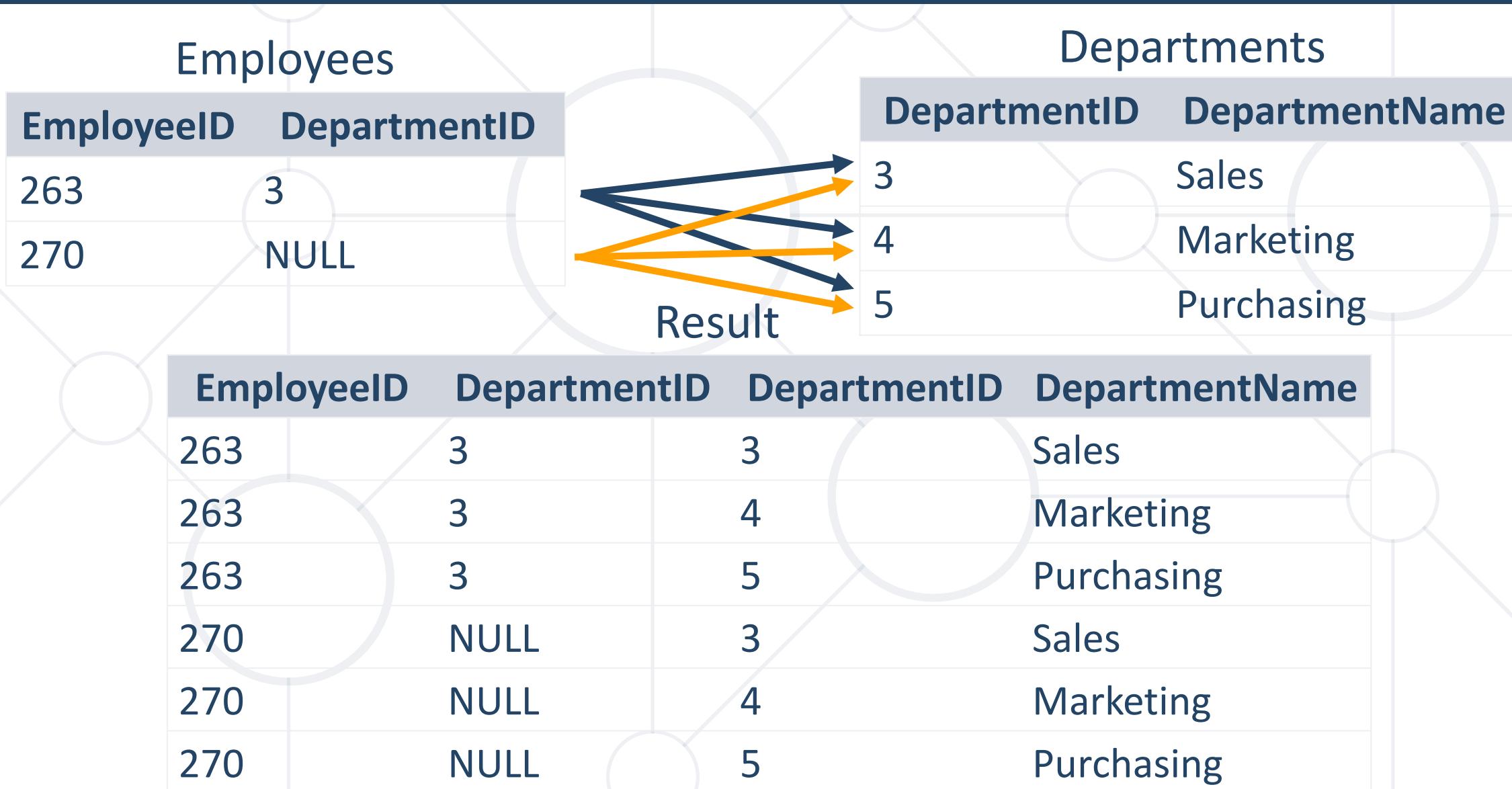
- This will produce a Cartesian product:

```
SELECT LastName, Name AS  
      DepartmentName  
FROM Employees, Departments
```

- The result:

LastName	DepartmentName
Gilbert	Engineering
Brown	Engineering
...	...
Gilbert	Sales
Brown	Sales

Cross Join



Cross Join Syntax

```
SELECT * FROM Employees AS e  
CROSS JOIN Departments AS d
```

Departments Table

No Join Conditions

Join Overview



Sally	13
John	10
Michael	22
Bob	11
Robin	7
Jessica	15

18	Accounting
10	Marketing
12	HR
22	Engineering
8	Sales
7	Executive

Relation

Join Overview (2)

- Inner Join



Sally	13
John	10
Michael	22
Bob	11
Robin	7
Jessica	15

18	Accounting
----	------------

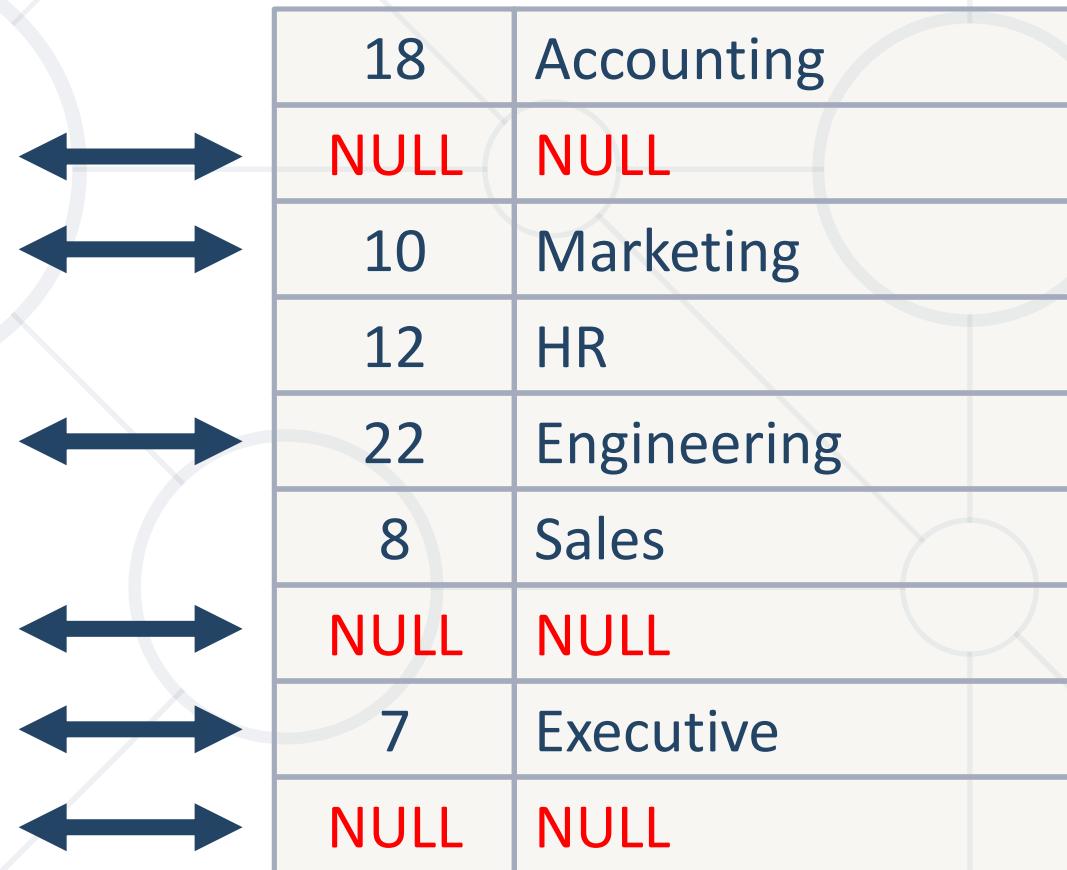
10	Marketing
12	HR
22	Engineering
8	Sales

7	Executive
---	-----------

Join Overview (3)

Left Outer Join

Sally	13
John	10
Michael	22
Bob	11
Robin	7
Jessica	15



18	Accounting
NULL	NULL
10	Marketing
12	HR
22	Engineering
8	Sales
NULL	NULL
7	Executive
NULL	NULL



Join Overview (4)

Right Outer Join



NULL	NULL
Sally	13
John	10
NULL	NULL
Michael	22
NULL	NULL
Bob	11
Robin	7
Jessica	15



Join Overview (5)

Full Outer Join



NULL	NULL
Sally	13
John	10
NULL	NULL
Michael	22
NULL	NULL
Bob	11
Robin	7
Jessica	15



18	Accounting
NULL	NULL
10	Marketing
12	HR
22	Engineering
8	Sales
NULL	NULL
7	Executive
NULL	NULL

Problem: Addresses with Towns

- Display **address information** of all employees in "SoftUni" **database**. Select **first 50 employees**.
 - The exact format of data is shown below
 - Order them by FirstName, then by LastName (ascending)
 - Hint: **Use three-way join**

	FirstName	LastName	Town	AddressText
1	A. Scott	Wright	Newport Hills	1400 Gate Drive
2	Alan	Brewer	Kenmore	8192 Seagull Court
3	Alejandro	McGuel	Seattle	7842 Ygnacio Valley Road
4	Alex	Nayberg	Newport Hills	4350 Minute Dr.

Check your solution here: <https://judge.softuni.org/Contests/Compete/Index/393#0>

Solution: Addresses with Towns

```
SELECT TOP 50 e.FirstName, e.LastName,  
t.Name as Town, a.AddressText  
FROM Employees e  
JOIN Addresses a ON e.AddressID = a.AddressID  
JOIN Towns t ON a.TownID = t.TownID  
ORDER BY e.FirstName, e.LastName
```

Check your solution here: <https://judge.softuni.org/Contests/Compete/Index/393#0>

Problem: Sales Employees

- Find **all employees** that are in the "**Sales**" department. Use "**SoftUni**" database.
- Follow the specified format:

	EmployeeID	FirstName	LastName	DepartmentName
1	268	Stephen	Jiang	Sales
2	273	Brian	Welcker	Sales
3	275	Michael	Blythe	Sales
4	276	Linda	Mitchell	Sales
5	277	Jillian	Carson	Sales

- Order them by **EmployeeID**

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/393#2>

Solution: Sales Employees

```
SELECT e.EmployeeID, e.FirstName, e.LastName,  
      d.Name AS DepartmentName  
  FROM Employees e  
    INNER JOIN Departments d  
      ON e.DepartmentID = d.DepartmentID  
 WHERE d.Name = 'Sales'  
 ORDER BY e.EmployeeID
```

Departments Table

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/393#2>

Problem: Employees Hired After

- Show **all employees** that:
 - Are **hired after 1/1/1999**
 - Are either **in "Sales" or "Finance" department**

	FirstName	LastName	HireDate	DeptName
1	Deborah	Poe	2001-01-19 00:00:00	Finance
2	Wendy	Kahn	2001-01-26 00:00:00	Finance
3	Candy	Spoon	2001-02-07 00:00:00	Finance
4	David	Barber	2001-02-13 00:00:00	Finance

- Sorted by **HireDate (ascending)**

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/393#5>

Solution: Employees Hired After

```
SELECT e.FirstName, e.LastName, e.HireDate,  
      d.Name as DeptName  
FROM Employees e  
    INNER JOIN Departments d  
      ON (e.DepartmentId = d.DepartmentId  
        AND e.HireDate > '1/1/1999'  
        AND d.Name IN ('Sales', 'Finance'))  
ORDER BY e.HireDate ASC
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/393#5>

Problem: Employee Summary

- Display information about **employee's manager** and **employee's department**
 - Show only **the first 50** employees
 - The exact format is shown below:

	EmployeeID	EmployeeName	ManagerName	DepartmentName
1	1	Guy Gilbert	Jo Brown	Production
2	2	Kevin Brown	David Bradley	Marketing
3	3	Roberto Tamburello	Terri Duffy	Engineering
4	4	Rob Walters	Roberto Tamburello	Tool Design
5	5	Thierry D'Hers	Ovidiu Craciun	Tool Design

- Sort by **EmployeeID (ascending)**

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/393#9>

Solution: Employee Summary

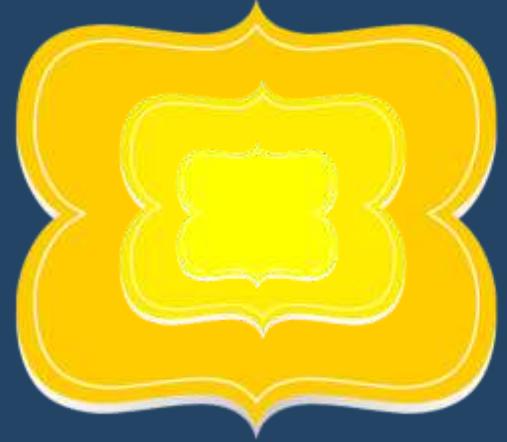
```
SELECT TOP 50
    e.EmployeeID,
    e.FirstName + ' ' + e.LastName AS EmployeeName,
    m.FirstName + ' ' + m.LastName AS ManagerName,
    d.Name AS DepartmentName
FROM Employees AS e
    LEFT JOIN Employees AS m ON m.EmployeeID =
        e.ManagerID
    LEFT JOIN Departments AS d ON d.DepartmentID =
        e.DepartmentID
ORDER BY e.EmployeeID ASC
```

Cross Table Selection

Self-join

Table Departments

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/393#9>



Subqueries

Query Manipulation On Multiple Levels

Subqueries

- Use a query's result as data for another query



Employees

EmployeeID	Salary
------------	--------

59	19,000
71	43,300
...	...

Query

Subquery

WHERE DepartmentID IN

DepartmentID	Name
10	Finance

Subquery Syntax

```
SELECT FROM Employees AS e  
WHERE e.DepartmentID IN  
(  
    SELECT d.DepartmentID  
    FROM Departments AS d  
    WHERE d.Name = 'Finance'  
)
```

Subquery

Table Departments

Problem: Min Average Salary

- Display **lowest average salary of all departments.**
 - Calculate average salary for each department.
 - Then show the value of smallest one.

MinAverageSalary	
1	10866.6666

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/393#10>

Solution: Min Average Salary

```
SELECT  
    MIN(a.AverageSalary) AS MinAverageSalary  
FROM  
(  
    SELECT e.DepartmentID,  
        AVG(e.Salary) AS AverageSalary  
    FROM Employees AS e  
    GROUP BY e.DepartmentID  
) AS a
```

Subquery

Table Employees

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/393#10>



Common Table Expressions

Reusable Subqueries

Common Table Expressions

- Common Table Expressions (CTE) can be considered as "named subqueries"
- They could be used to improve code **readability** and code **reuse**
- Usually, they are positioned in the beginning of the query

```
WITH CTE_Name (ColumnA, ColumnB...)  
AS  
(  
    -- Insert subquery here.  
)
```

Common Table Expressions Syntax

```
WITH Employees_CTE
    (FirstName, LastName, DepartmentName)
AS
(
    SELECT e.FirstName, e.LastName, d.Name
    FROM Employees AS e
    LEFT JOIN Departments AS d ON
        d.DepartmentID = e.DepartmentID
)
SELECT FirstName, LastName, DepartmentName
FROM Employees_CTE
```





Temporary Tables

Temporary Tables

- **Temporary tables** are stored in **tempdb**
- Automatically deleted when they are **no longer used**

```
CREATE TABLE #TempTable
```

```
(
```

-- Add columns here.

```
)
```

```
SELECT * FROM #TempTable
```

Temporary Table Syntax

```
CREATE TABLE #Employees
(
    Id INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50),
    Address VARCHAR(50)
)
SELECT * FROM #Employees
```



Types of Temporary Tables

- **Table variables** (DECLARE @t TABLE)
 - Visible only to the connection that creates it
- **Local temporary tables** (CREATE TABLE #t)
 - Visible only to the connection that creates it
- **Global temporary tables** (CREATE TABLE ##t)
 - Visible to everyone
 - Deleted when all connections that have referenced them, have closed
- **Tempdb permanent tables** (USE tempdb CREATE TABLE t)
 - Visible to everyone. Deleted when the server is restarted

Summary

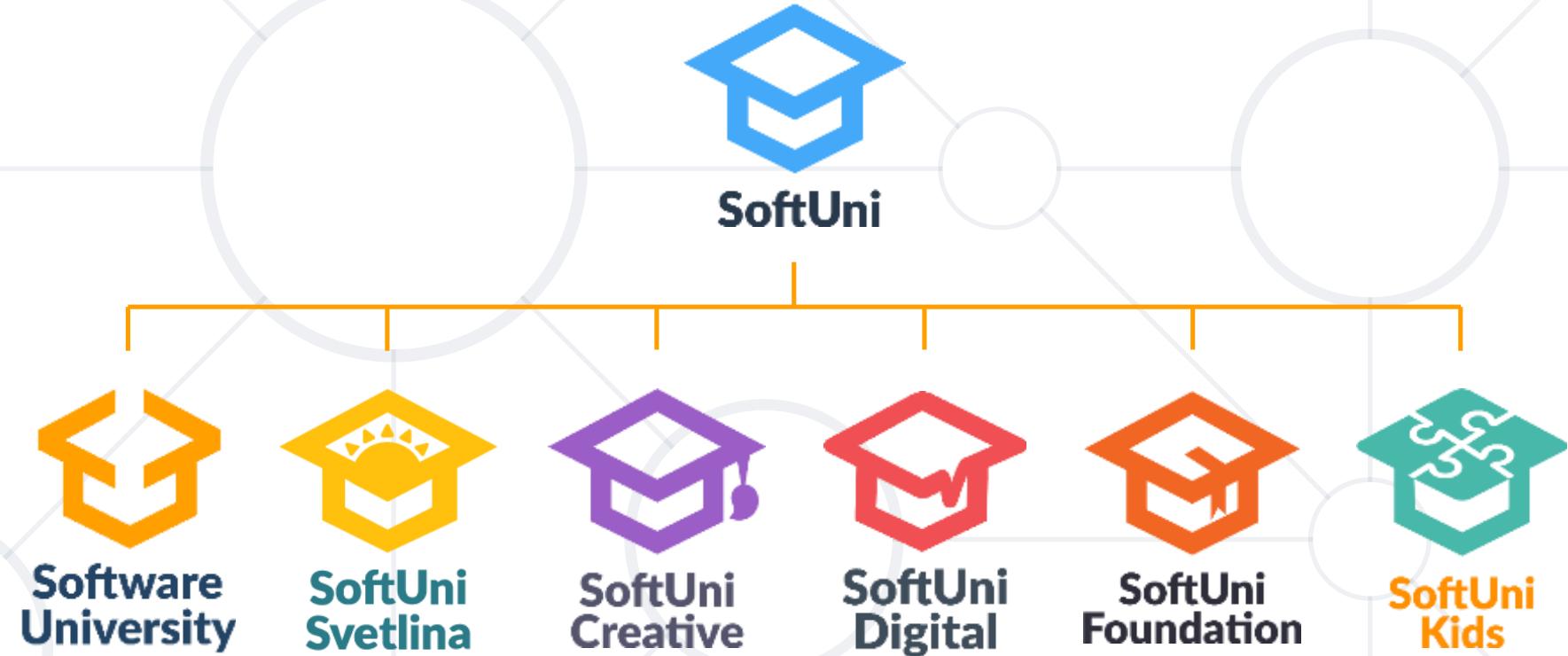
- **Joins**

```
SELECT * FROM Employees AS e
JOIN Departments AS d ON
d.DepartmentId = e.DepartmentID
```

- **Subqueries** are used to nest queries
- **CTE's** improve code reuse and readability
- **Indices** improve SQL search performance if used properly



Questions?



SoftUni Diamond Partners



SCHWARZ



Coca-Cola HBC
Bulgaria



Postbank

Решения за твоето упре



**SOFTWARE
GROUP**

Bosch.**.IO**



SmartIT



**PHAR
VISION**



CAREERS



INDEAVR
Serving the high achievers

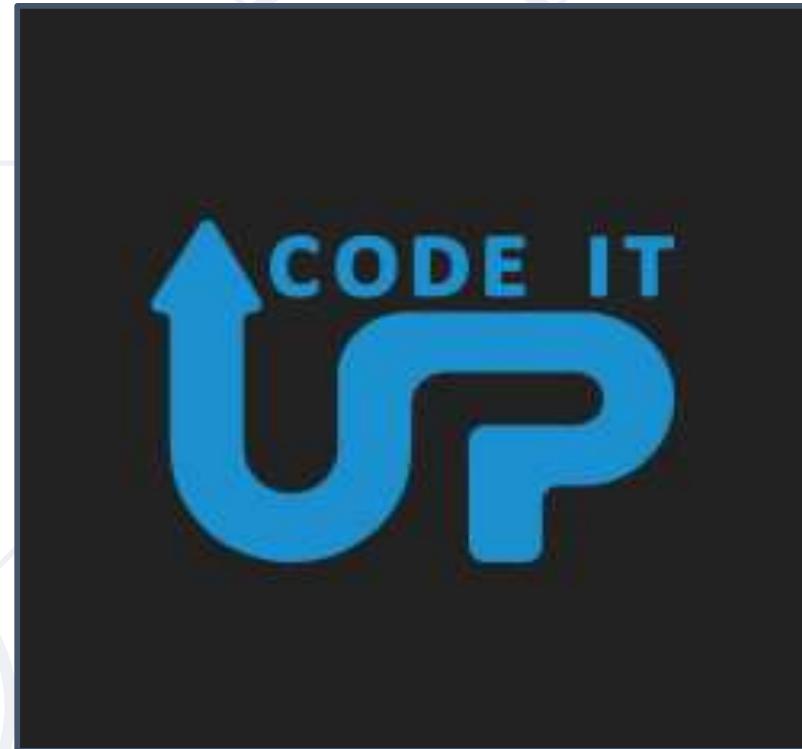


**SUPER
HOSTING
.BG**



TECHNOLOGY

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University

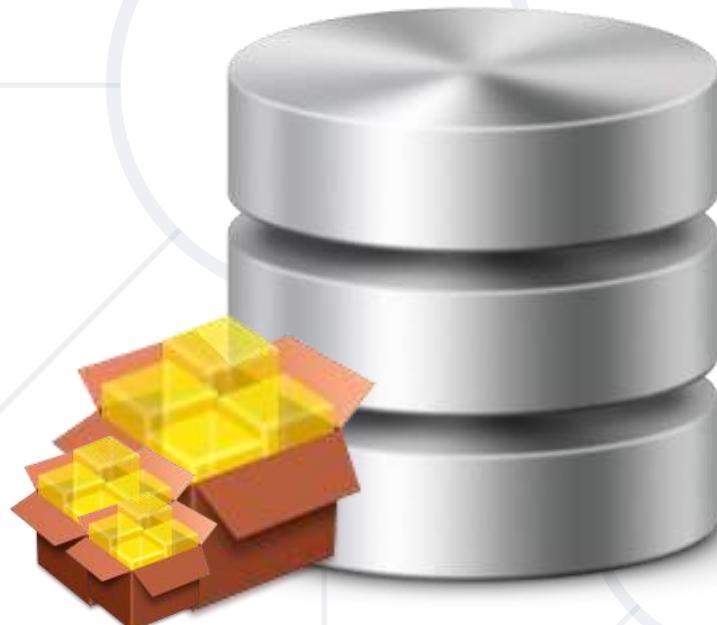


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Indices and Data Aggregation

How to Get Data Insights?



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Table of Contents

1. Indices
2. Grouping
3. Aggregate Functions
4. Having Clause



sli.do

#csharp-db



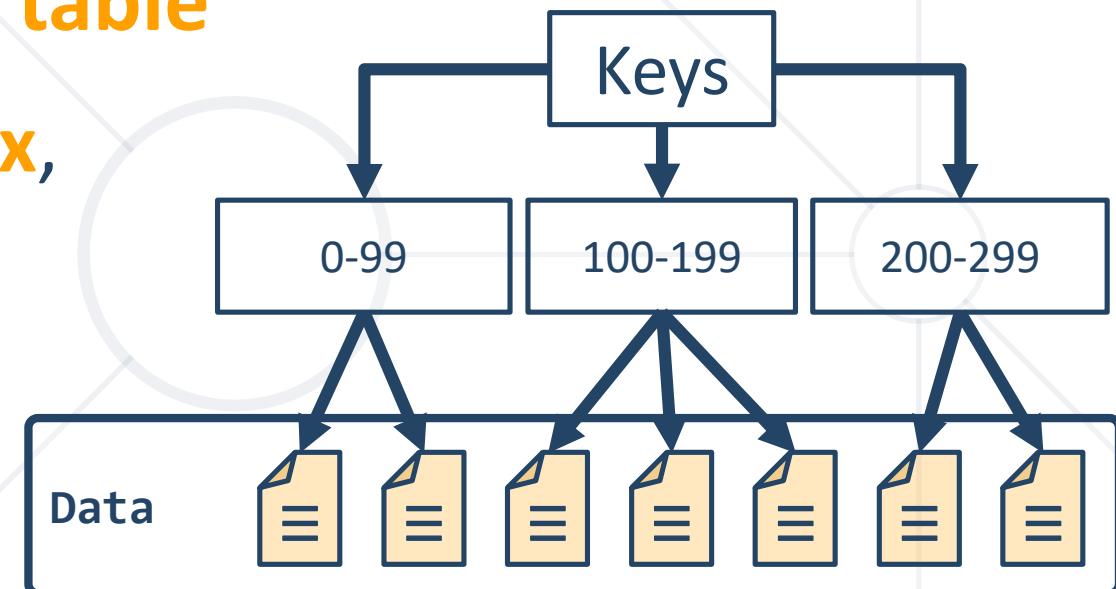
Indices

Clustered and Non-Clustered Indexes

- Indices speed up the searching of values in a certain column or group of columns
 - Usually implemented as B-trees
- Indices can be built-in the table (clustered) or stored externally (non-clustered)
- Adding and deleting records in indexed tables is slower!
- Indices should be used for big tables only (e.g. 500 000 rows).

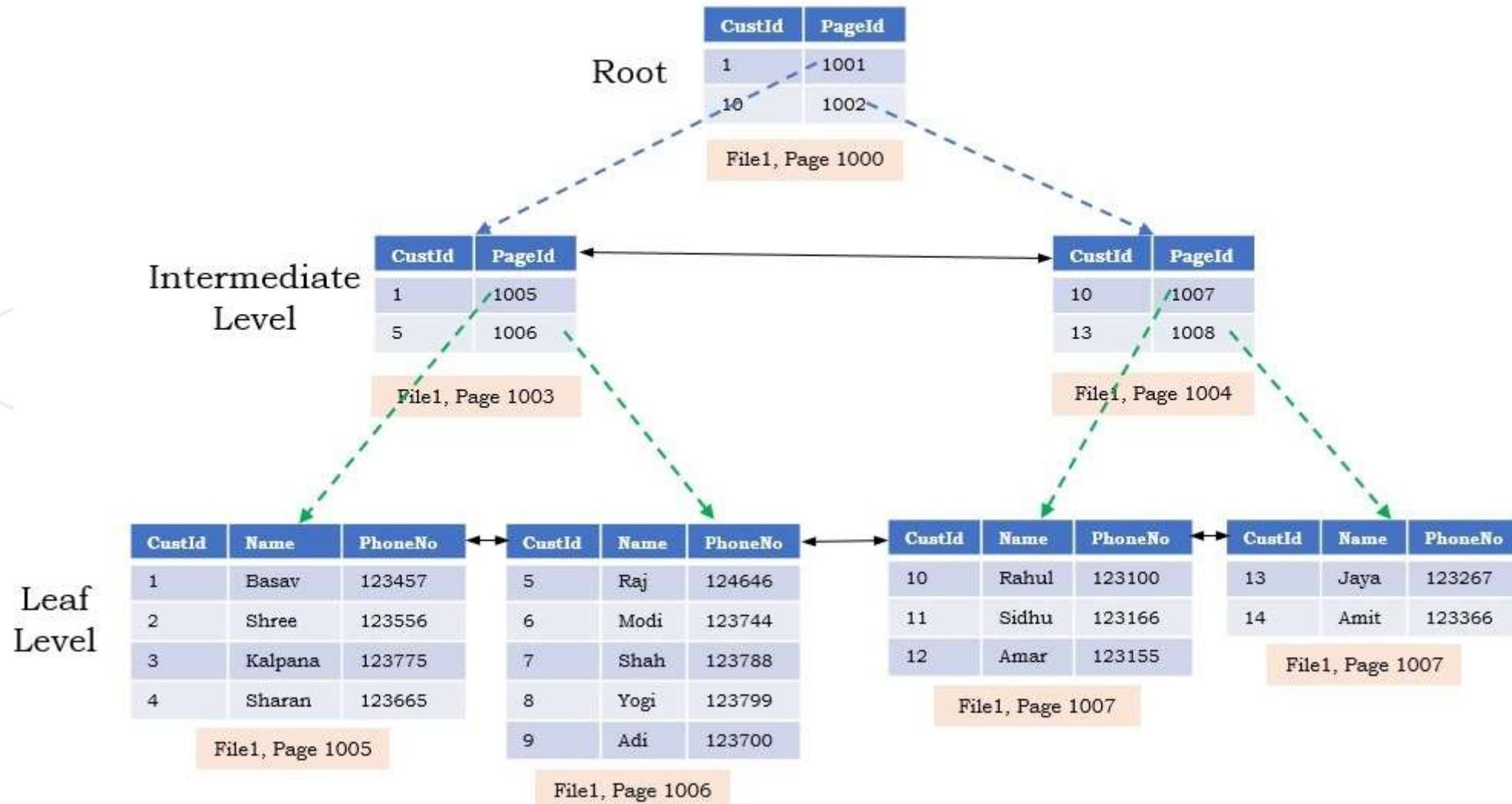
Clustered Indices

- Clustered index is actually **the data itself**
 - Very useful for **fast execution** of WHERE, ORDER BY and GROUP BY clauses
- Maximum **1** clustered index per table
 - If a table **has no clustered index**, its **data rows are stored in an unordered structure (heap)**.



Clustered Indexes (2)

B+ Tree Structure of a Clustered Index

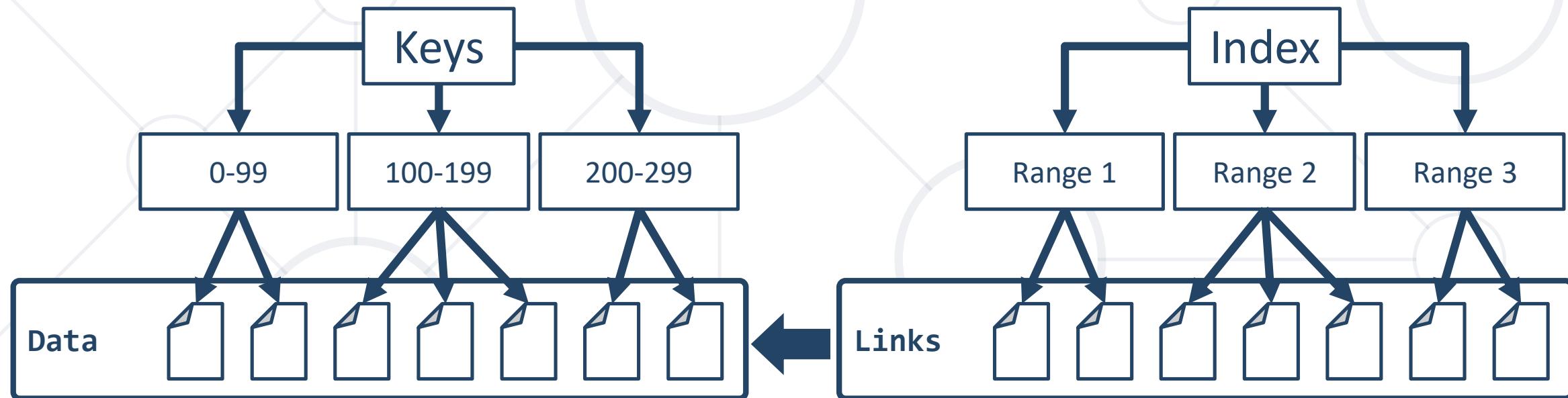


Non-Clustered Indeces

- Useful for **fast retrieving** of a range of records
- Maintained in a **separate structure** in the DB
- Tend to be **much narrower** than the base table
 - Can **locate the exact record(s)** with **less I/O**
- Has **at least one more intermediate level** than the clustered index
 - Much **less valuable** if a table **doesn't have a clustered index**

Non-Clustered Indexes (2)

- A non-clustered index **has pointers** to the **actual data rows** (pointers to the clustered index if there is one).



Indices Syntax

```
CREATE NONCLUSTERED INDEX  
IX_Employees_FirstName_LastName  
ON Employees(FirstName, LastName)
```

Index Type

Table Name

Columns



Demo: Index Performance

Live Demo



Grouping

Consolidating Data Based On Criteria

Grouping (1)

- **Grouping** allows receiving data into separate groups based on a common property



Single row

Grouping column

Employee	DepartmentName	Salary
Adam	Database Support	5,000
John	Database Support	15,000
Jane	Application Support	10,000
George	Application Support	15,000
Lila	Application Support	5,000
Fred	Software Support	15,000

Can be aggregated

Grouping (2)

- **GROUP BY** allows you to get each **separate group** and use an "**aggregate**" function over it (like **Average**, **Min** or **Max**):

```
SELECT e.DepartmentID  
      FROM Employees AS e  
GROUP BY e.DepartmentID
```

Group Columns

- **DISTINCT** allows you to get **all unique** values:

```
SELECT DISTINCT e.DepartmentID  
      FROM Employees AS e
```

Unique
Values

Problem: Departments Total Salaries

- Use "SoftUni" database to create a query which prints the total sum of salaries for each department
 - Order them by **DepartmentID** (ascending)

Employee	DepartmentID	Salary
Adam	1	5,000
John	1	15,000
Jane	2	10,000
George	2	15,000
Lila	2	5,000
Fred	3	15,000



DepartmentID	TotalSalary
1	20,000
2	30,000
3	15,000

Solution: Departments Total Salaries

- After **grouping** every employee **by** its **department**, we can use an **aggregate function** to calculate the total amount of money per group

```
SELECT e.DepartmentID,  
       SUM(e.Salary) AS TotalSalary  
  FROM Employees AS e  
 GROUP BY e.DepartmentID  
 ORDER BY e.DepartmentID
```

Column Alias

Table Alias

Group Columns

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/291#12>



Aggregate Functions

COUNT, SUM, MAX, MIN, AVG...

Aggregate Functions

- Operate over (non-empty) groups
- Perform data analysis on each one
 - MIN, MAX, AVG, COUNT, etc.

```
SELECT e.DepartmentID,  
       MIN(e.Salary) AS MinSalary  
  FROM Employees AS e  
 GROUP BY e.DepartmentID
```



DepartmentID	MinSalary
1	32700.00
2	25000.00
3	23100.00
4	13500.00
5	12800.00
6	40900.00
7	9500.00

- Aggregate functions usually ignore NULL values

Aggregate Functions: COUNT

- **COUNT** - counts the **values** in one or more **grouped columns**
 - Ignores **NULL** values

Employee	DepartmentName	Salary
Adam	Database Support	5,000
John	Database Support	15,000
Jane	Application Support	10,000
George	Application Support	15,000
Lila	Application Support	5,000
Fred	Software Support	15,000

DepartmentName	SalaryCount
Database Support	2
Application Support	3
Software Support	1

COUNT Syntax

- **COUNT(Column Name)**

```
SELECT e.DepartmentID,  
       COUNT(e.Salary) AS SalaryCount  
  FROM Employees AS e  
 GROUP BY e.DepartmentID
```

New Column Alias

Group Columns

- Note: **COUNT ignores** any employee with **NULL** salary

Aggregate Functions: SUM

- **SUM** - sums the values in a column

Employee	DepartmentName	Salary
Adam	Database Support	5,000
John	Database Support	15,000
Jane	Application Support	10,000
George	Application Support	15,000
Lila	Application Support	5,000
Fred	Software Support	15,000



DepartmentName	TotalSalary
Database Support	20,000
Application Support	30,000
Software Support	15,000

SUM Syntax

- If any department **has no salaries**, it **returns NULL**

```
SELECT e.DepartmentID,  
       SUM(e.Salary) AS TotalSalary  
  FROM Employees AS e  
 GROUP BY e.DepartmentID
```

Grouping Column

New Column Alias

Aggregate Functions: MAX

- **MAX** - takes **the largest value** in a column

Employee	DepartmentName	Salary
Adam	Database Support	5,000
John	Database Support	15,000
Jane	Application Support	10,000
George	Application Support	15,000
Lila	Application Support	5,000
Fred	Software Support	15,000



DepartmentName	MaxSalary
Database Support	15,000
Application Support	15,000
Software Support	15,000

MAX Syntax

```
SELECT e.DepartmentID,  
       MAX(e.Salary) AS MaxSalary  
  FROM Employees AS e  
GROUP BY e.DepartmentID
```

Grouping Column

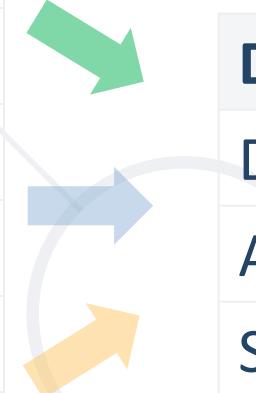
New Column Alias

Group Columns

Aggregate Functions: MIN

- **MIN** - takes **the smallest value** in a column

Employee	DepartmentName	Salary
Adam	Database Support	5,000
John	Database Support	15,000
Jane	Application Support	10,000
George	Application Support	15,000
Lila	Application Support	5,000
Fred	Software Support	15,000



DepartmentName	MinSalary
Database Support	5,000
Application Support	5,000
Software Support	15,000

MIN Syntax

```
SELECT e.DepartmentID,  
       MIN(e.Salary) AS MinSalary  
  FROM Employees AS e  
 GROUP BY e.DepartmentID
```

New Column Alias

Group Columns

Aggregate Functions: AVG

- **AVG** - calculates the **average value** in a column

Employee	DepartmentName	Salary
Adam	Database Support	5,000
John	Database Support	15,000
Jane	Application Support	10,000
George	Application Support	15,000
Lila	Application Support	5,000
Fred	Software Support	15,000



DepartmentName	AvgSalary
Database Support	10,000
Application Support	10,000
Software Support	15,000

AVG Syntax

```
SELECT e.DepartmentID,  
       AVG(e.Salary) AS AvgSalary  
  FROM Employees AS e  
 GROUP BY e.DepartmentID
```

New Column Alias

Group Columns

Aggregate Functions: STRING_AGG

- **STRING_AGG** - Concatenates the values of string expressions and places separator values between them. The separator is not added at the end of string

Expressions are converted to **NVARCHAR** or **VARCHAR** types during concatenation. Non-string types are converted to **NVARCHAR** type

```
STRING_AGG ( expression, separator )
[WITHIN GROUP ( ORDER BY expression [ ASC | DESC ] )]
```



Having
Using Predicates While Grouping

HAVING Clause

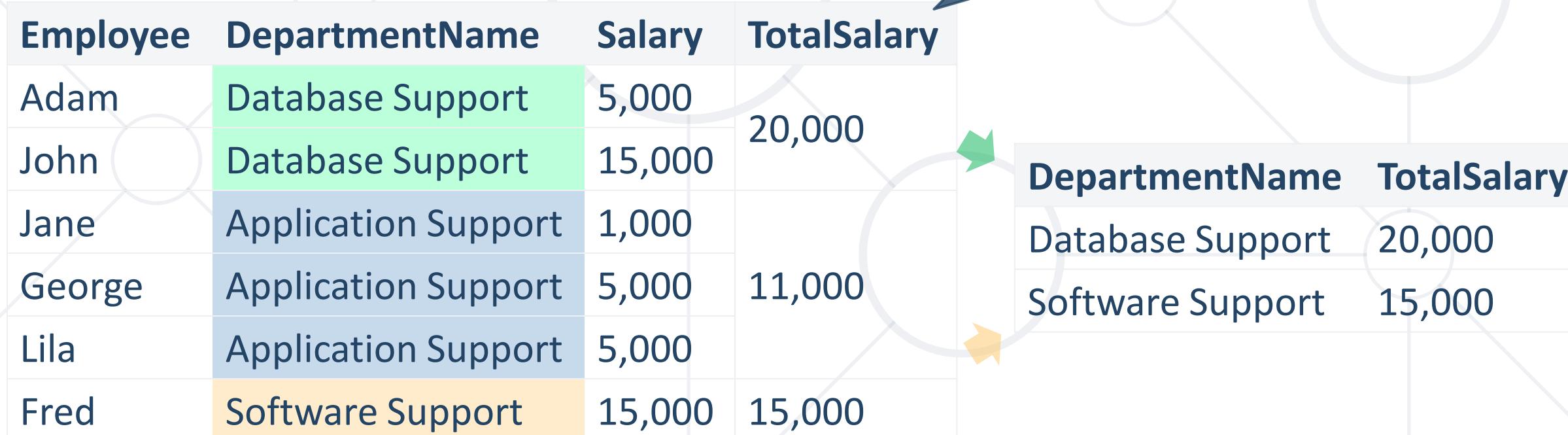
- The **HAVING clause** is used to **filter data** based on **aggregate values**
 - We **cannot** use it **without grouping** first
- **Aggregate functions** (MIN, MAX, SUM etc.) are **executed only once**
 - Unlike HAVING, **WHERE filters rows before aggregation**



HAVING Clause: Example

- Filter departments having total salary more than or equal to 15,000

Aggregated value



The diagram illustrates the aggregation process. On the left, a detailed table shows individual employee records with columns: Employee, DepartmentName, Salary, and TotalSalary. The rows show employees Adam, John, Jane, George, Lila, and Fred, each assigned to a specific department. The 'TotalSalary' column is explicitly shown for each row. On the right, a summary table shows aggregated data by department, with columns: DepartmentName and TotalSalary. It lists two departments: Database Support (TotalSalary 20,000) and Software Support (TotalSalary 15,000). Arrows point from the 'TotalSalary' column in the detailed table to the corresponding values in the summary table, indicating the aggregation function.

Employee	DepartmentName	Salary	TotalSalary
Adam	Database Support	5,000	20,000
John	Database Support	15,000	
Jane	Application Support	1,000	11,000
George	Application Support	5,000	
Lila	Application Support	5,000	
Fred	Software Support	15,000	15,000

DepartmentName	TotalSalary
Database Support	20,000
Software Support	15,000

HAVING Syntax

```
SELECT e.DepartmentID,  
       SUM(e.Salary) AS TotalSalary  
  FROM Employees AS e  
 GROUP BY e.DepartmentID  
 HAVING SUM(e.Salary) >= 15000
```

Aggregate Function

Column Alias

Grouping Columns

Having Predicate

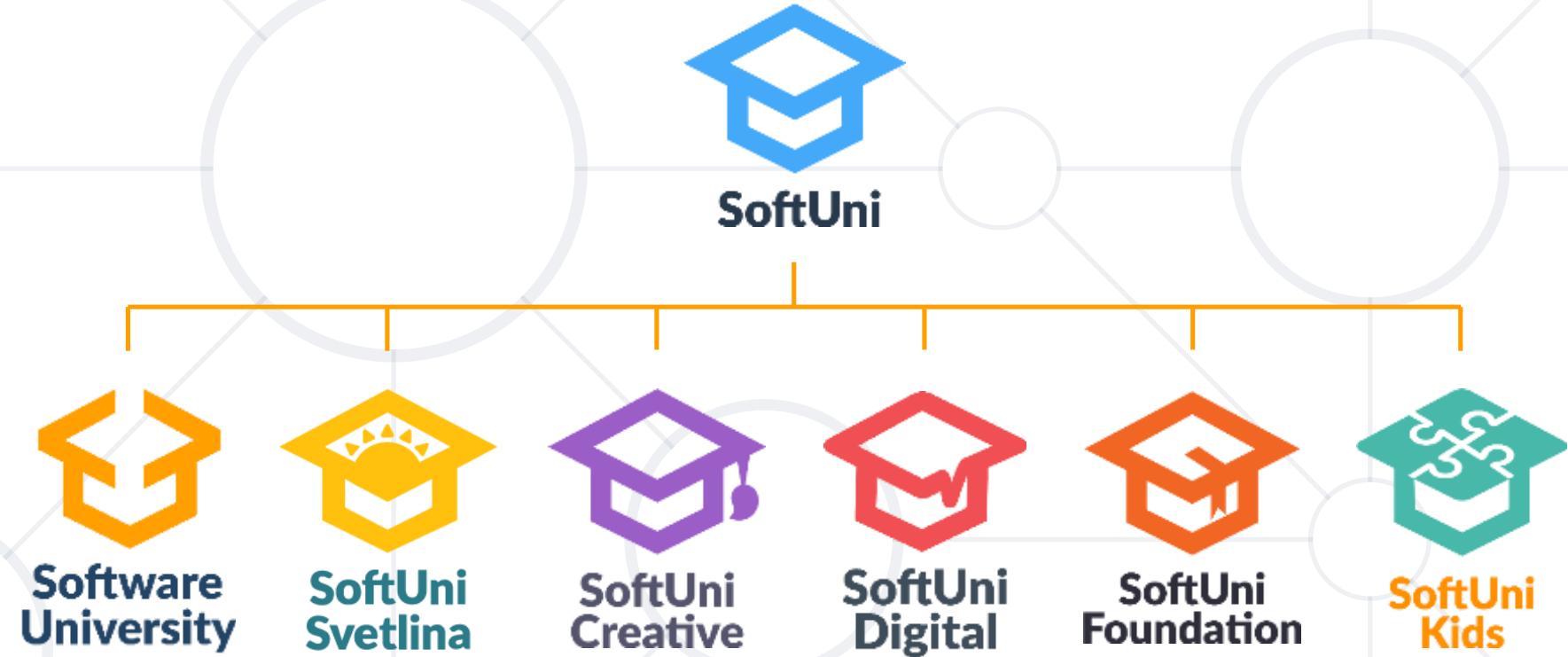
Summary

- Grouping by Shared Properties
- Aggregate Functions
- Having Clause

```
SELECT
    SUM(e.Salary) AS 'TotalSalary'
FROM Employees AS e
GROUP BY e.DepartmentID
HAVING SUM(e.Salary) >= 15_000
```



Questions?



SoftUni Diamond Partners



SCHWARZ



Educational Partners



VIRTUAL RACING SCHOOL



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Functions and Stored Procedures

Database Programmability



SoftUni



Software University

<https://about.softuni.bg/>

SoftUni Team

Technical Trainers



Table of Contents

1. User-Defined Functions
2. Stored Procedures
3. Stored Procedures with Parameters
4. Error Handling



sli.do

#csharp-db



User-Defined Functions

Definition, Usage, Syntax

Functions: Basic Definition

- At its core, a function **receives an input** and **produces an output**



Types of User-Defined Functions

- 
- **Scalar functions**
 - Similar to the **built-in functions**
 - Returns a **single value**
 - **Table-valued functions**
 - Similar to **a view with parameters**
 - **Returns a table** as a result of a single **SELECT** statement
 - **Inline table-valued function (TVF)**
 - **Multi-statement table-valued function (MSTVF)**

Functions: Limitations

- User-defined functions **cannot** be used to perform actions that **modify** the database state
- User-defined functions **cannot** return **multiple** result sets
- User-defined functions cannot make use of **dynamic SQL** or **temp tables**. Table variables are allowed.
- User-defined functions can be nested up to 32 levels
- Error handling is restricted in a user-defined function
UDF does not support **TRY...CATCH**, **@ERROR** or **RAISERROR**



Create Functions (Scalar)

```
CREATE FUNCTION udf_ProjectDurationWeeks (@StartDate DATETIME,  
@EndDate DATETIME)  
RETURNS INT  
AS  
BEGIN  
    DECLARE @projectWeeks INT;  
    IF(@EndDate IS NULL)  
        BEGIN  
            SET @EndDate = GETDATE()  
        END  
    SET @projectWeeks = DATEDIFF(WEEK, @StartDate, @EndDate)  
    RETURN @projectWeeks;  
END
```

Diagram illustrating the components of the SQL function code:

- Function Name:** udf_ProjectDurationWeeks
- Parameters:** @StartDate DATETIME, @EndDate DATETIME
- Return Type:** INT
- Variable:** @projectWeeks
- IF Statement:** IF(@EndDate IS NULL)
- Return Value:** @projectWeeks

Create Functions (Table-Valued Function)

```
CREATE FUNCTION udf_AverageSalaryByDepartment()
RETURNS TABLE AS
RETURN
(
    SELECT d.[Name] AS Department, AVG(e.Salary) AS AverageSalary
    FROM Departments AS d
    JOIN Employees AS e ON d.DepartmentID = e.DepartmentID
    GROUP BY d.DepartmentID, d.[Name]
```

Function Name

No Parameters

Return Type

Return Value

Create Functions (Multi-statement TVF)

```
CREATE FUNCTION udf_EmployeeListByDepartment(@DepName nvarchar(20))
RETURNS @result TABLE(
    FirstName nvarchar(50) NOT NULL,
    LastName nvarchar(50) NOT NULL,
    DepartmentName nvarchar(20) NOT NULL) AS
BEGIN
    WITH Employees_CTE (FirstName, LastName, DepartmentName)
    AS(
        SELECT e.FirstName, e.LastName, d.[Name]
        FROM Employees AS e
        LEFT JOIN Departments AS d ON d.DepartmentID = e.DepartmentID)
    INSERT INTO @result SELECT FirstName, LastName, DepartmentName
    FROM Employees_CTE WHERE DepartmentName = @DepName
    RETURN
END
```

Execute Functions

- Functions are called using **schemaName.functionName**

```
SELECT [ProjectID],  
       [StartDate],  
       [EndDate],  
       dbo.udf_ProjectDurationWeeks([StartDate],[EndDate])  
AS ProjectWeeks  
FROM [SoftUni].[dbo].[Projects]
```

Call the Function

ProjectID	StartDate	EndDate	ProjectWeeks
1	2016-09-01	2016-10-07	5
2	2016-10-01	2016-10-07	1
3	2015-10-07	NULL	52

Problem: Salary Level Function

- Write a function **ufn_GetSalaryLevel(@Salary MONEY)** that receives salary of an employee and returns the level of the salary
 - If salary is < 30000 return "Low"
 - If salary is between 30000 and 50000 (inclusive) returns "Average"
 - If salary is > 50000 return "High"

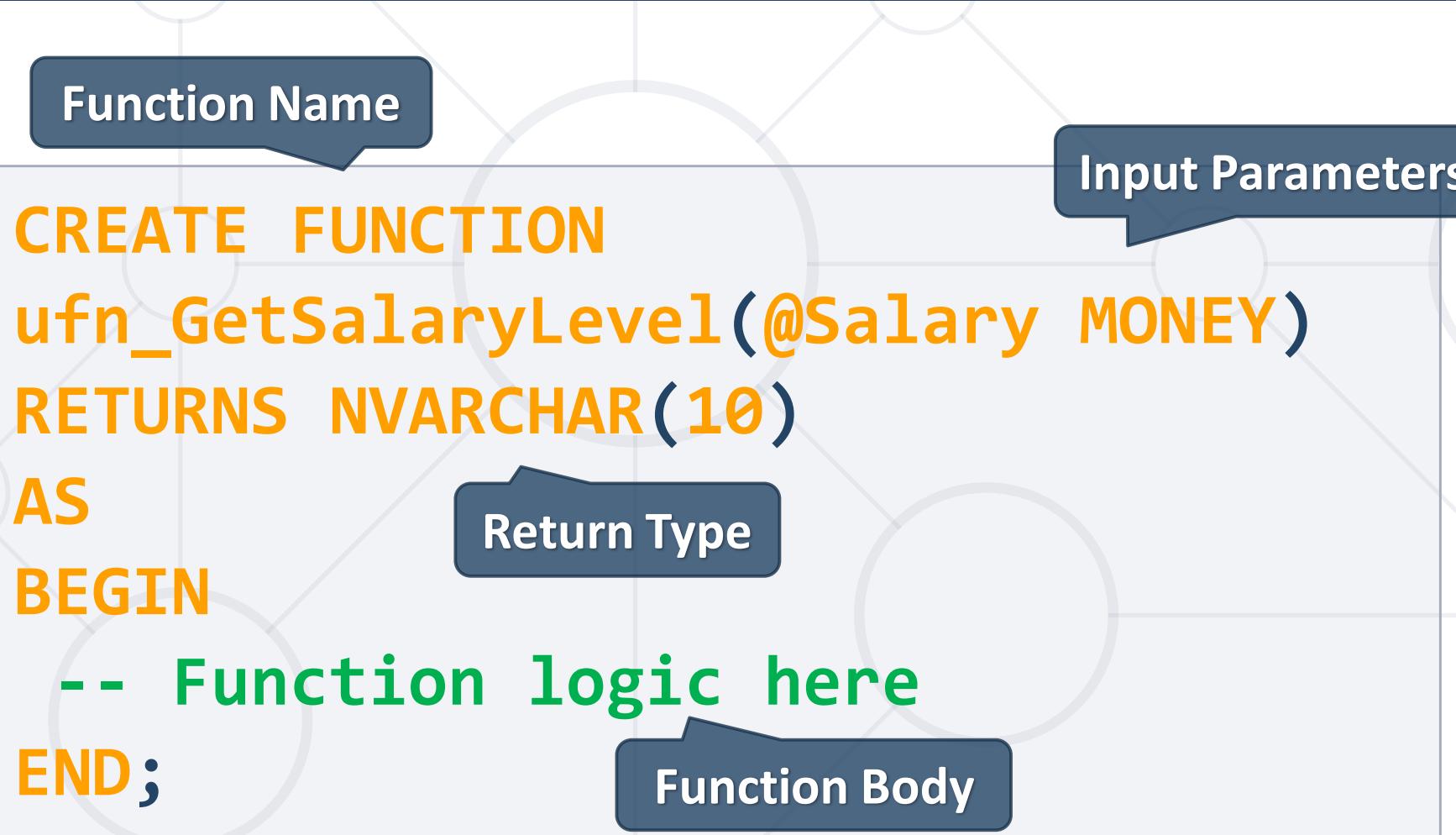
	FirstName	LastName	Salary	SalaryLevel
1	Guy	Gilbert	12500.00	Low
2	Kevin	Brown	13500.00	Low
3	Roberto	Tamburello	43300.00	Average
4	Rob	Walters	29800.00	Low
5	Thierry	D'Hers	25000.00	Low



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1025#4>

Solution: Salary Level Function (1)

```
CREATE FUNCTION  
ufn_GetSalaryLevel(@Salary MONEY)  
RETURNS NVARCHAR(10)  
AS  
BEGIN  
    -- Function logic here  
END;
```



The diagram illustrates the structure of the function code. It features a central light gray rectangular box containing the T-SQL code. Four callout boxes with rounded corners point to specific parts of the code:

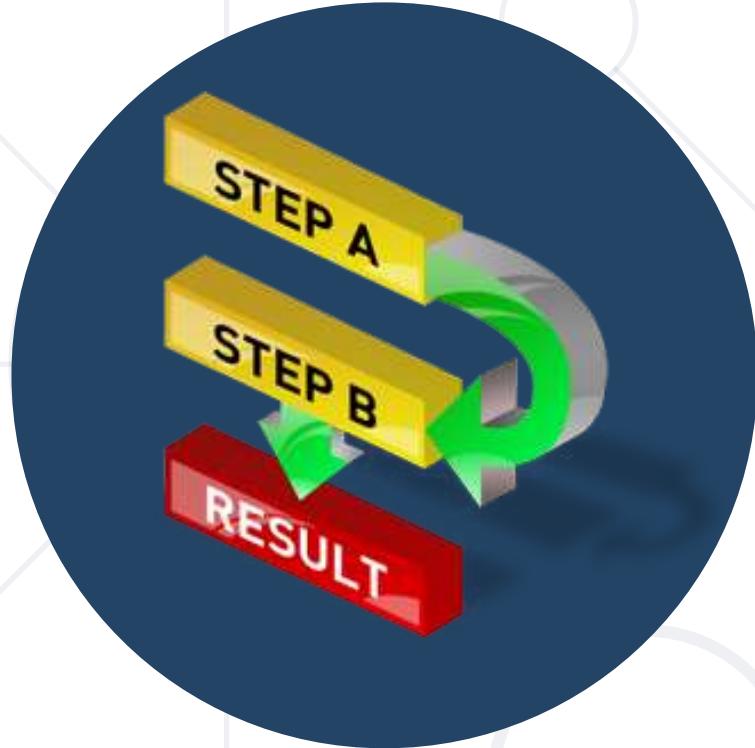
- Function Name:** Points to the first line, `CREATE FUNCTION`.
- Input Parameters:** Points to the parameter declaration, `ufn_GetSalaryLevel(@Salary MONEY)`.
- Return Type:** Points to the `RETURNS NVARCHAR(10)` clause.
- Function Body:** Points to the `BEGIN` block and the placeholder comment `-- Function logic here`.

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1025#4>

Solution: Salary Level Function (2)

```
DECLARE @salaryLevel VARCHAR(10)           Variable  
IF (@Salary < 30000)                      IF Statement  
    SET @salaryLevel = 'Low'  
ELSE IF(@Salary <= 50000)  
    SET @salaryLevel = 'Average'  
ELSE  
    SET @salaryLevel = 'High'  
RETURN @salaryLevel                         Return Result
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1025#4>



Stored Procedures

What Are Stored Procedures?

- **Stored procedures** are **named sequences** of **T-SQL statements**
 - **Encapsulate** repetitive program **logic**
 - **Can accept input parameters**
 - **Can return output results**
- **Benefits** of stored procedures
 - **Share** application logic
 - **Improved performance**
 - **Reduced network traffic**
 - They can be used as a **security** mechanism

Types of Stored Procedures

- **User-defined**
 - Can be created in a **user-defined database** or in all system databases except the **Resource** database
 - Can be developed in either **Transact-SQL** or as a reference to a **Microsoft .NET Framework** method
- **Temporary**
 - A form of user-defined procedures stored in **tempdb**

Creating Stored Procedures

- Syntax: **CREATE PROCEDURE ... AS ...**

- Example:

```
USE SoftUni
GO

CREATE PROC dbo.usp_SelectEmployeesBySeniority
AS
    SELECT *
    FROM Employees
    WHERE DATEDIFF(Year, HireDate, GETDATE()) > 20
GO
```



Executing Stored Procedures

- Executing a stored procedure by **EXEC**

```
EXEC usp_SelectEmployeesBySeniority
```

- Executing a stored procedure within an **INSERT** statement

```
INSERT INTO Customers  
EXEC usp_SelectEmployeesBySeniority
```

Altering Stored Procedures

- Use the **ALTER PROCEDURE** statement

```
USE SoftUni
GO
ALTER PROC usp_SelectEmployeesBySeniority
AS
    SELECT FirstName, LastName, HireDate,
           DATEDIFF(Year, HireDate, GETDATE()) as Years
    FROM Employees
    WHERE DATEDIFF(Year, HireDate, GETDATE()) > 20
    ORDER BY HireDate
GO
```

Procedure Name

Dropping Stored Procedures

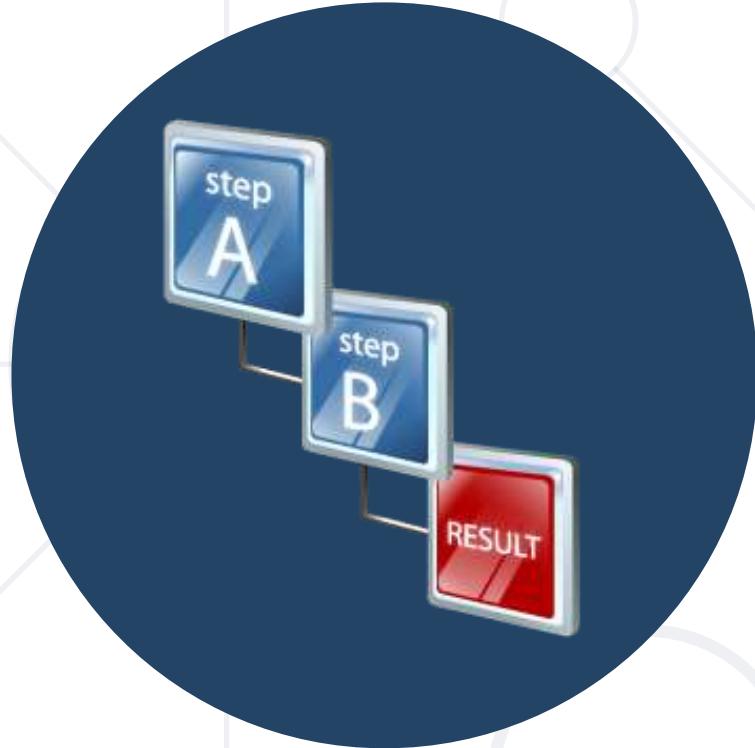
- **DROP PROCEDURE**

```
DROP PROC usp_SelectEmployeesBySeniority
```

- You could **check** if any objects **depend** on the stored procedure by executing **the system stored procedure sp_depends**

```
EXEC sp_depends 'usp_SelectEmployeesBySeniority'
```

Stored Procedures with Parameters



Defining Parameterized Procedures

- To define a **parameterized procedure**, use the syntax:

```
CREATE PROCEDURE usp_ProcedureName  
(@parameter1Name parameterType,  
 @parameter2Name parameterType,...) AS
```

- Choose the parameter types carefully and provide **appropriate default values**

```
CREATE PROC  
usp_SelectEmployeesBySeniority(  
@minYearsAtWork int = 5) AS ...
```

Parameterized Stored Procedures - Example



```
CREATE PROC usp_SelectEmployeesBySeniority  
    (@minYearsAtWork int = 5)  
AS  
    SELECT FirstName, LastName, HireDate,  
          DATEDIFF(Year, HireDate, GETDATE()) as Years  
    FROM Employees  
    WHERE DATEDIFF(Year, HireDate, GETDATE()) > @minYearsAtWork  
    ORDER BY HireDate  
GO  
  
EXEC usp_SelectEmployeesBySeniority 10
```

Procedure Name

Procedure Logic

Usage

Passing Parameter Values

- Passing values by **parameter name**

```
EXEC usp_AddCustomer  
    @customerID = 'ALFKI',  
    @companyName = 'Alfreds Futterkiste',  
    @address = 'Obere Str. 57',  
    @city = 'Berlin',  
    @phone = '030-0074321'
```

- Passing values by **position**

```
EXEC usp_AddCustomer 'ALFKI2', 'Alfreds  
Futterkiste', 'Obere Str. 57', 'Berlin',  
'030-0074321'
```

Returning Values Using OUTPUT Parameters

```
CREATE PROCEDURE dbo.usp_AddNumbers  
    @firstNumber SMALLINT,  
    @secondNumber SMALLINT,  
    @result INT OUTPUT  
AS  
    SET @result = @firstNumber + @secondNumber  
GO  
  
DECLARE @answer smallint  
EXECUTE usp_AddNumbers 5, 6, @answer OUTPUT  
SELECT 'The result is: ', @answer  
  
-- The result is: 11
```

Creating procedure

Executing procedure

Display results

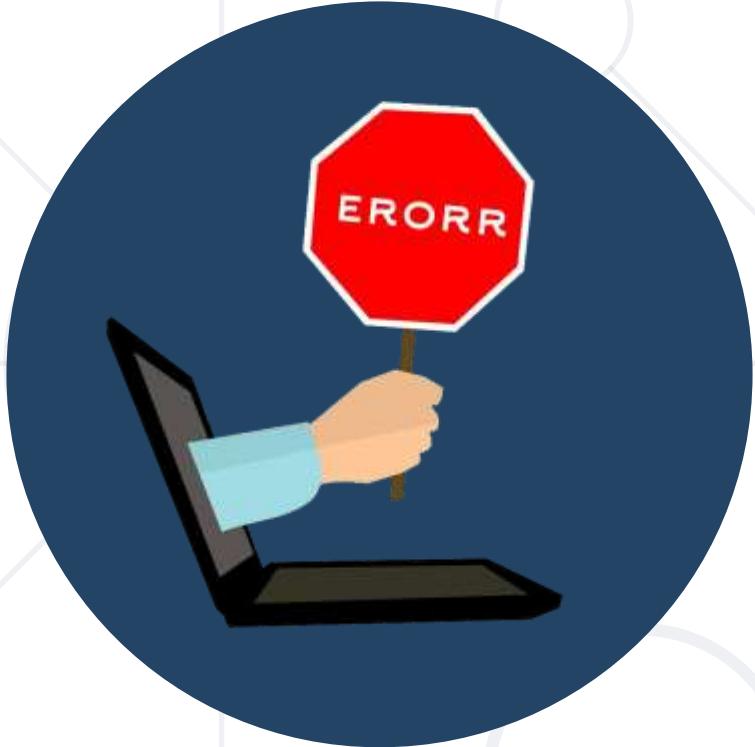
Returning Multiple Results

Checks if procedure exists
and then Creates or Alters it

```
CREATE OR ALTER PROC usp_MultipleResults
AS
SELECT FirstName, LastName FROM Employees
SELECT FirstName, LastName, d.[Name] AS Department
FROM Employees AS e
JOIN Departments AS d ON e.DepartmentID = d.DepartmentID;
GO

EXEC usp_MultipleResults
```

Multiple SELECT
statements



Error Handling

- **THROW**

- Raises an exception and transfers execution to a **CATCH** block
- Arguments:
 - error_number - **INT** (between **50000** and **2147483647**)
 - message - **NVARCHAR(2048)**
 - state - **TINYINT** (between **0** and **255**)

```
IF(@candidateAge < @minimalCandidateAge)
BEGIN
    THROW 50001, 'The candidate is too young!', 1;
END
```

- **TRY...CATCH**

- SQL Statements can be enclosed in a **TRY** block
- If an error occurs in the **TRY** block, control is passed to another group of statements that is enclosed in a **CATCH** block

```
BEGIN TRY  
    { sql_statement | statement_block }  
END TRY  
BEGIN CATCH  
    [ { sql_statement | statement_block } ]  
END CATCH  
[ ; ]
```

Error Handling

```
BEGIN TRY
    -- Generate a divide-by-zero error.
    SELECT 1/0
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber
        ,ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_STATE() AS ErrorState
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_LINE() AS ErrorLine
        ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
GO
```

- **@@ERROR**
 - Returns 0 if the previous Transact-SQL statement encountered no errors
 - Returns an error number if the previous statement encountered an error
 - **@@ERROR** is cleared and reset on each statement executed, check it immediately following the statement being verified, or save it to a local variable that can be checked later

Summary

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

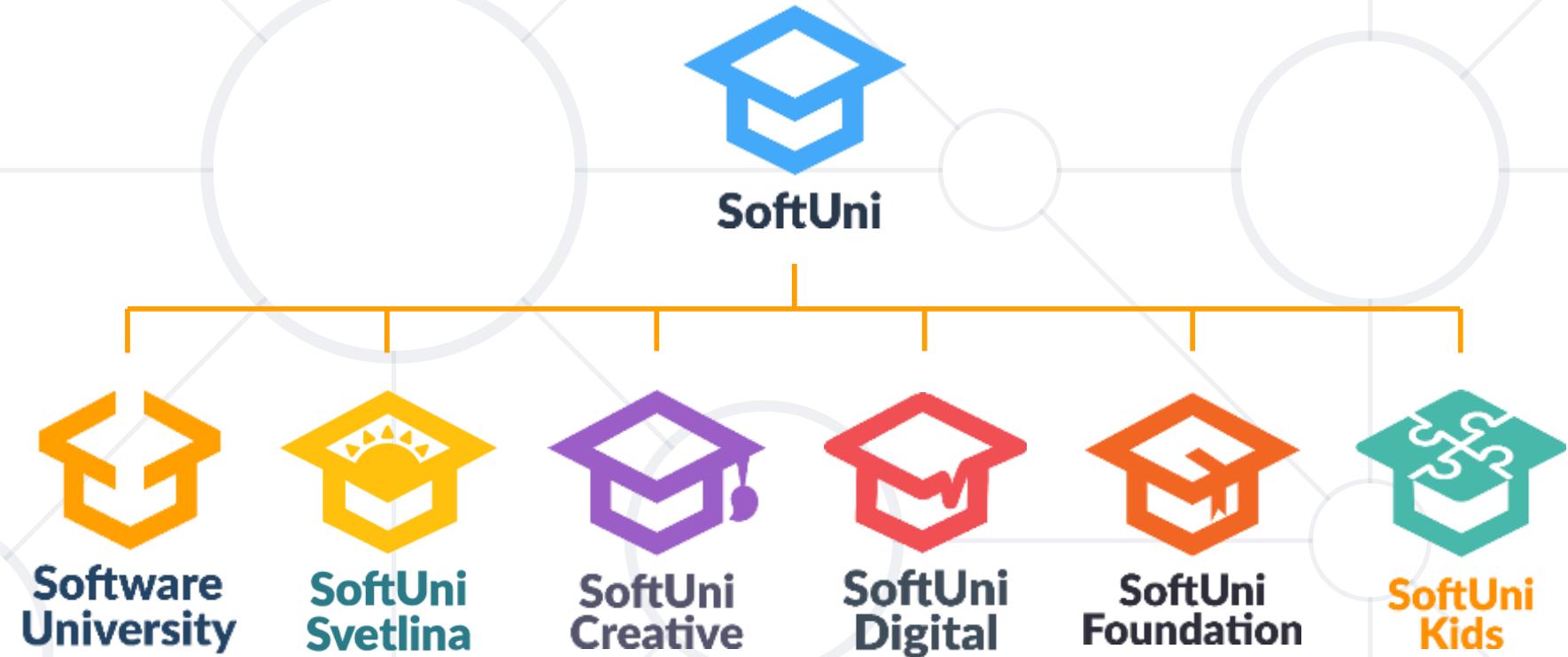
```
CREATE FUNCTION f_ProcedureName  
RETURNS ...  
AS  
...
```

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

```
CREATE PROC usp_ProcedureName  
AS ...
```



Questions?



SoftUni Diamond Partners



SCHWARZ



Coca-Cola HBC
Bulgaria



Postbank

Решения за твоето упре



Bosch..IO****



SmartIT

POKERSTARS

CAREERS

AMBITIONED

INDEAVR
Serving the high achievers

DRAFT
KINGS

create**X**

**SUPER
HOSTING
.BG**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



Triggers and Transactions

Database Programmability



SoftUni Team

Technical Trainers



SoftUni

Software University

<https://about.softuni.bg/>

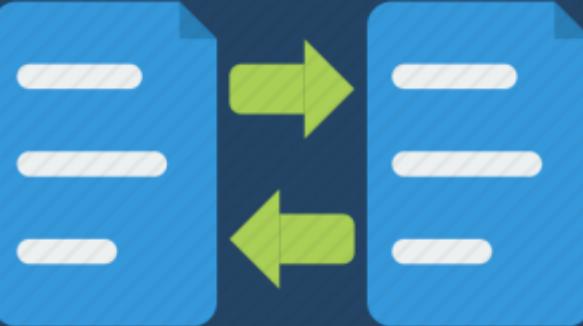
Table of Contents

1. Transactions
2. ACID Model
3. Triggers
4. Database Security



sli.do

#csharp-db

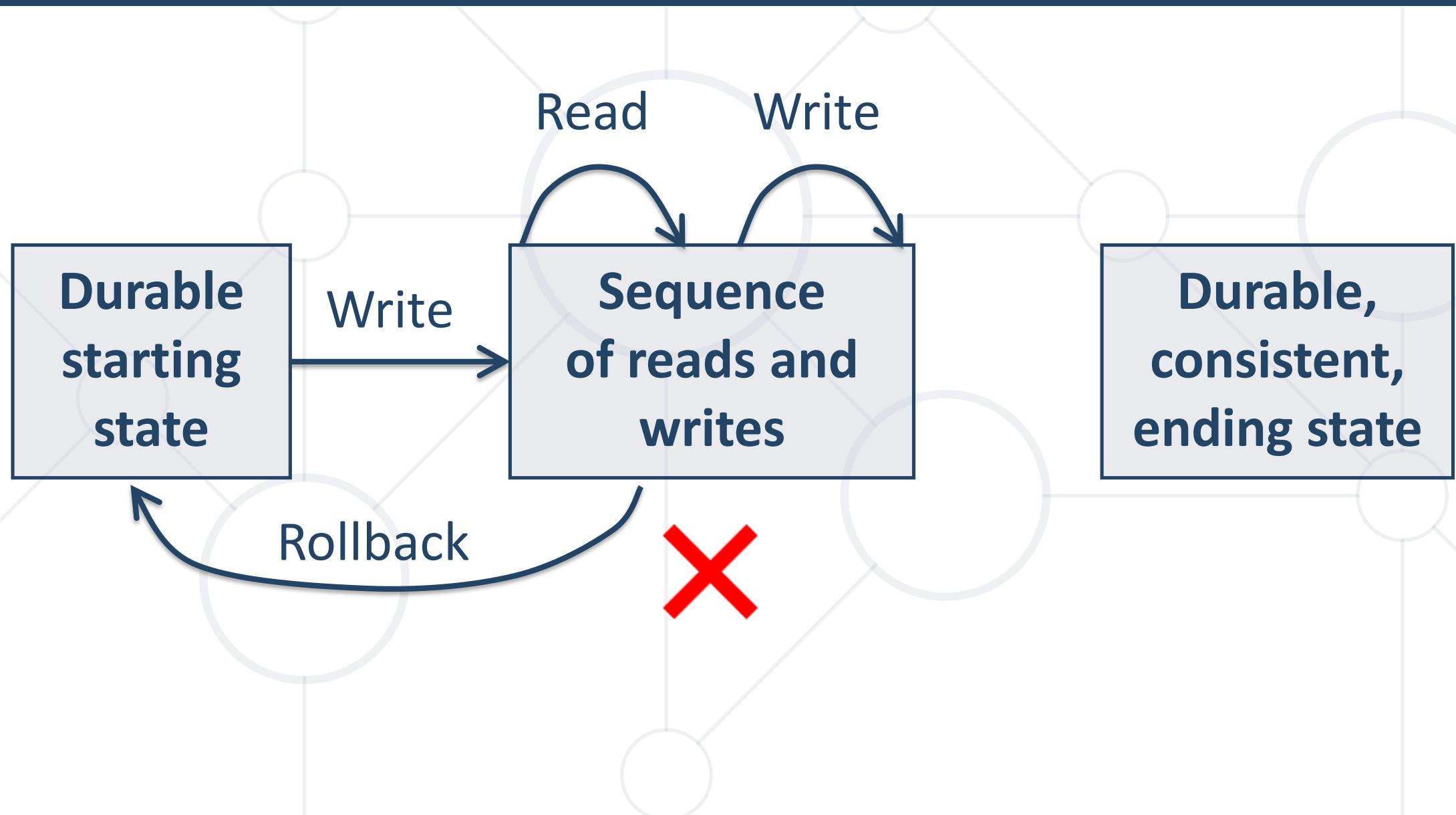


Transactions

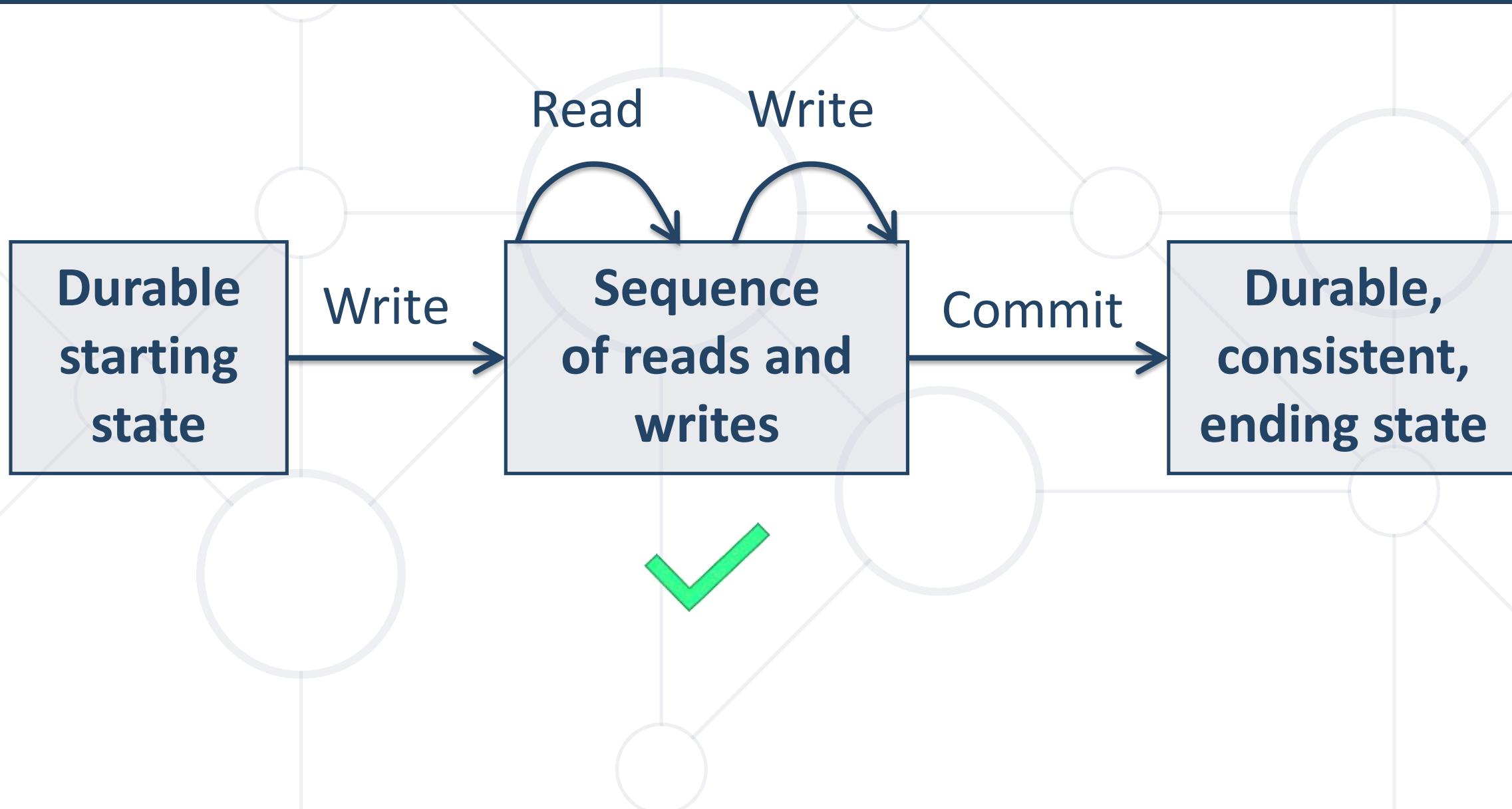
Definition, Usage, ACID Model

- 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)



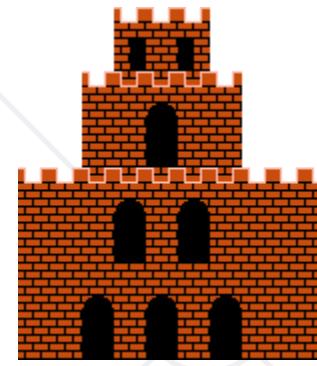
Transactions Behavior

- **Transactions** guarantee the **consistency** and the **integrity** of the **database**
 - All **changes** in a transaction **are temporary**
 - Changes are **persisted** when a **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



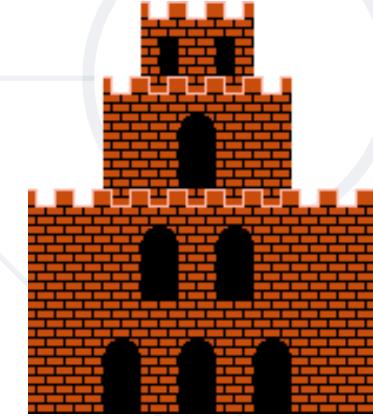
Castle 1-2



Mario

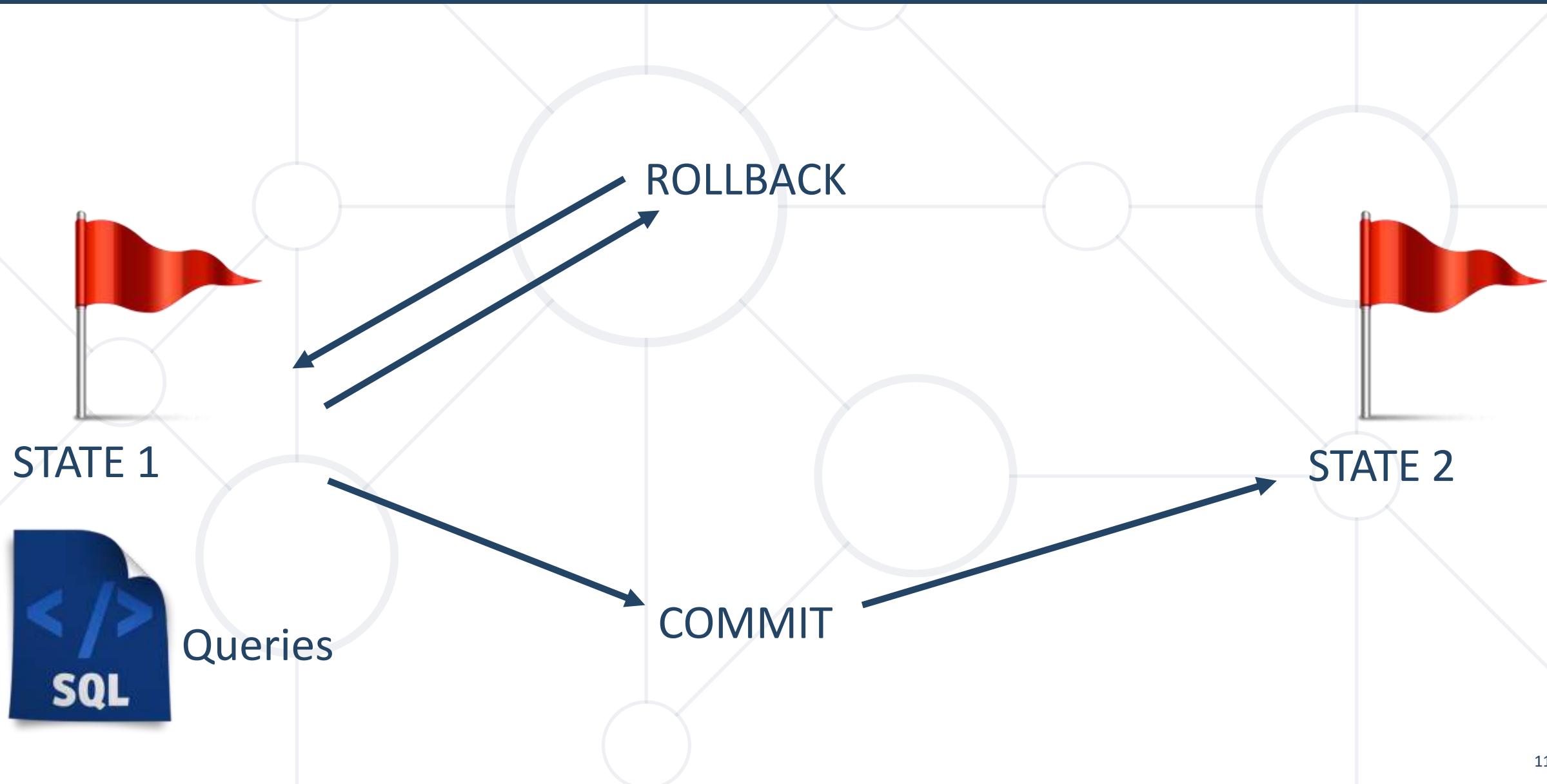
SURVIVE

DIE



Castle 1-3

What Are Transactions?



Transactions Syntax

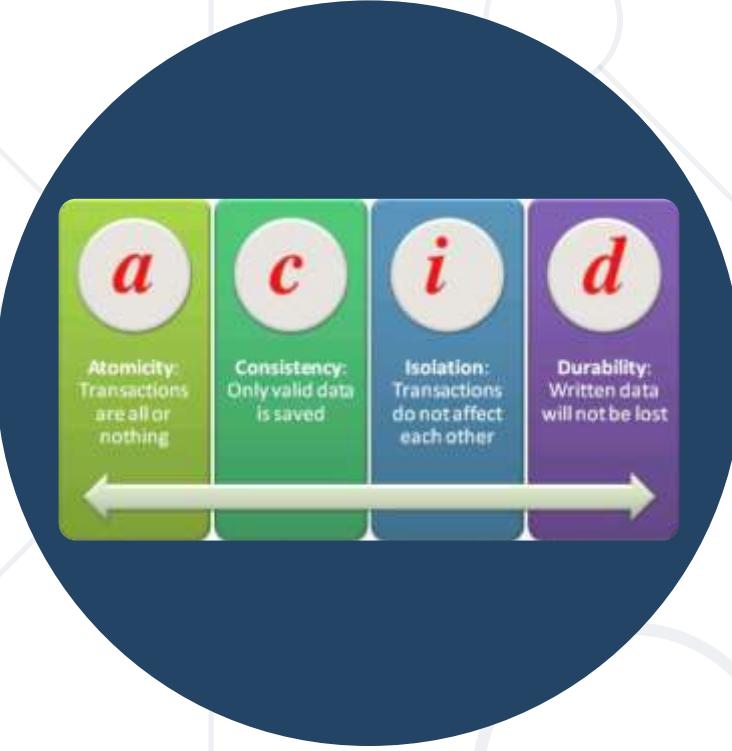
```
CREATE PROC usp_Withdraw (@withdrawAmount DECIMAL(18,2), @accountId INT)
AS
BEGIN TRANSACTION
    UPDATE Accounts SET Balance = Balance - @withdrawAmount
    WHERE Id = @accountId
    IF @@ROWCOUNT <> 1 -- Didn't affect exactly one row
        BEGIN
            ROLLBACK
            THROW 50001, 'Invalid account!', 1
        RETURN
    END
    COMMIT
```

Start Transaction

Withdraw Money

Undo Changes

Save Changes



ACID Models

Solving Problems Before They Arise

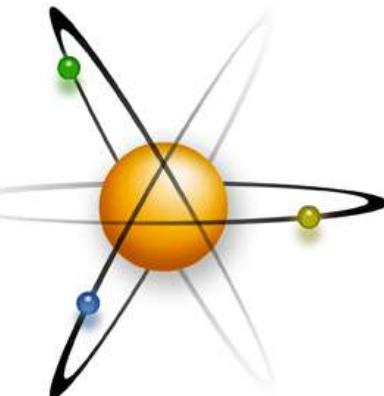
Transaction Properties

- Modern DBMS servers have **built-in transaction support**
 - Implement "**ACID**" transactions
 - MS SQL Server, Oracle, MySQL, PostgreSQL, etc.
- **ACID** means:
 - **Atomicity**
 - **Consistency**
 - **Isolation**
 - **Durability**



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 **its end**
 - Only **valid data** will be **written** to the DB
 - Transaction **cannot break the rules** of the database
 - Primary keys, foreign keys, check constraints, data types...
- Consistency example:
 - Transaction **cannot end with a duplicate primary key** in a table

- **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 means that:
 - If a transaction is **committed** it becomes **persistent**
 - **Cannot be lost or undone**
 - Ensured by the 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**)





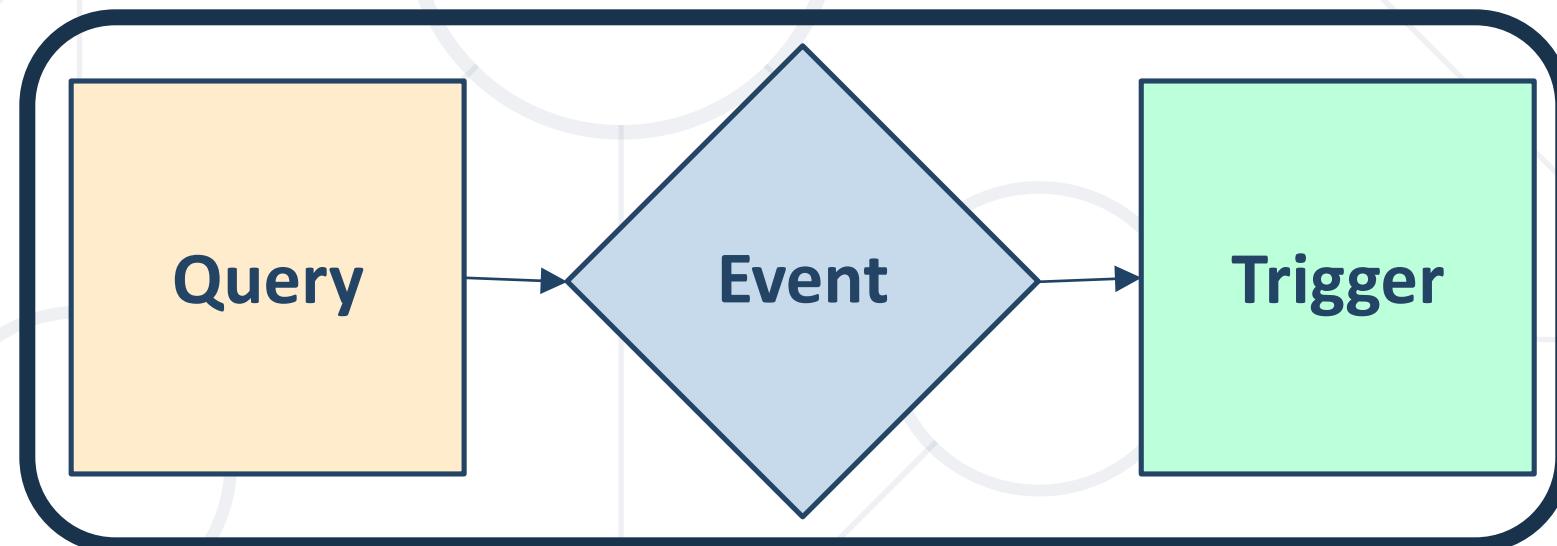
Triggers

What Are Triggers?

- Triggers are very much like stored procedures
 - Called in case of a 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

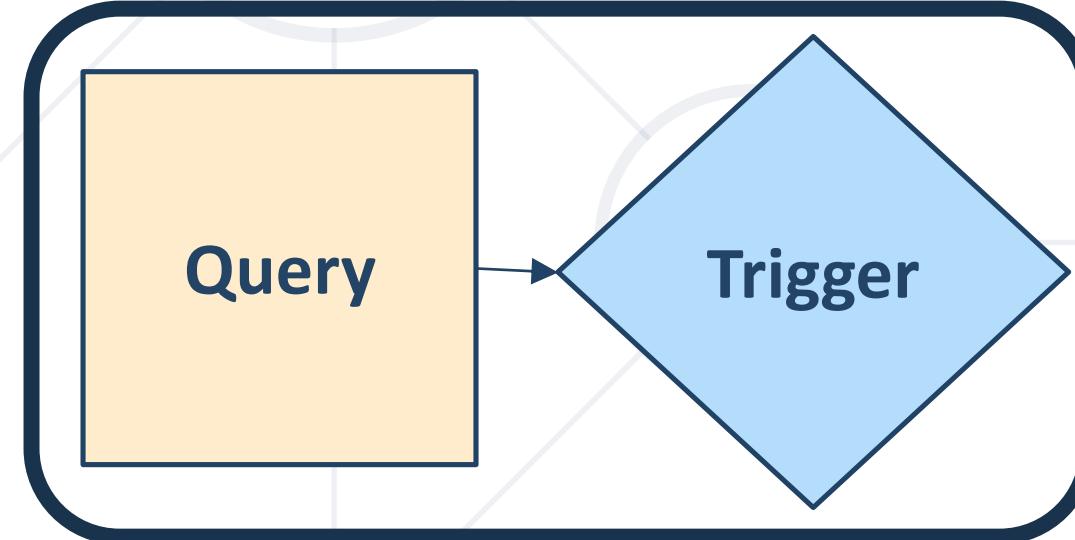
AFTER Trigger

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

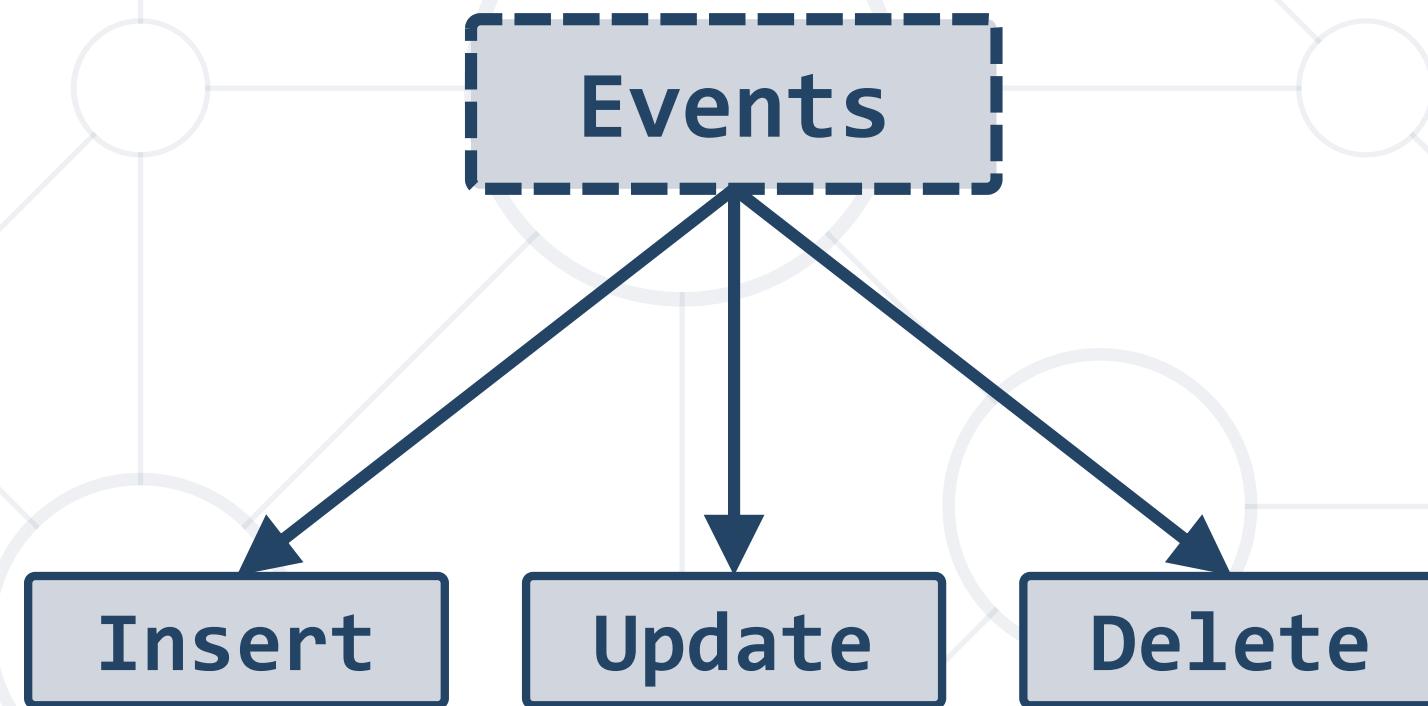


INSTEAD OF Trigger

- **INSTEAD OF** Trigger completely replaces an event action from happening
 - You can apply totally different logic



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



AFTER Triggers

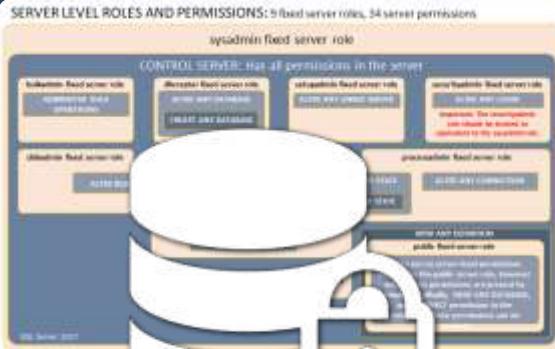
- Defined by the keyword **FOR**

```
CREATE TRIGGER tr_AddToLogsOnAccountUpdate
ON Accounts FOR UPDATE
AS
    INSERT INTO Logs(AccountId, OldAmount, NewAmount, UpdatedOn)
    SELECT i.Id, d.Balance, i.Balance, GETDATE()
    FROM inserted AS i
    JOIN deleted AS d ON i.Id = d.Id
    WHERE i.Balance != d.Balance
GO
```

INSTEAD OF Triggers

- Defined by using **INSTEAD OF**

```
CREATE OR ALTER TRIGGER tr_SetIsDeletedonDelete
ON AccountHolders
INSTEAD OF DELETE
AS
    UPDATE AccountHolders SET IsDeleted = 1
    WHERE Id IN (SELECT Id FROM deleted)
GO
```



Database Security

Fixed Server Roles, Fixed Database Roles

- 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

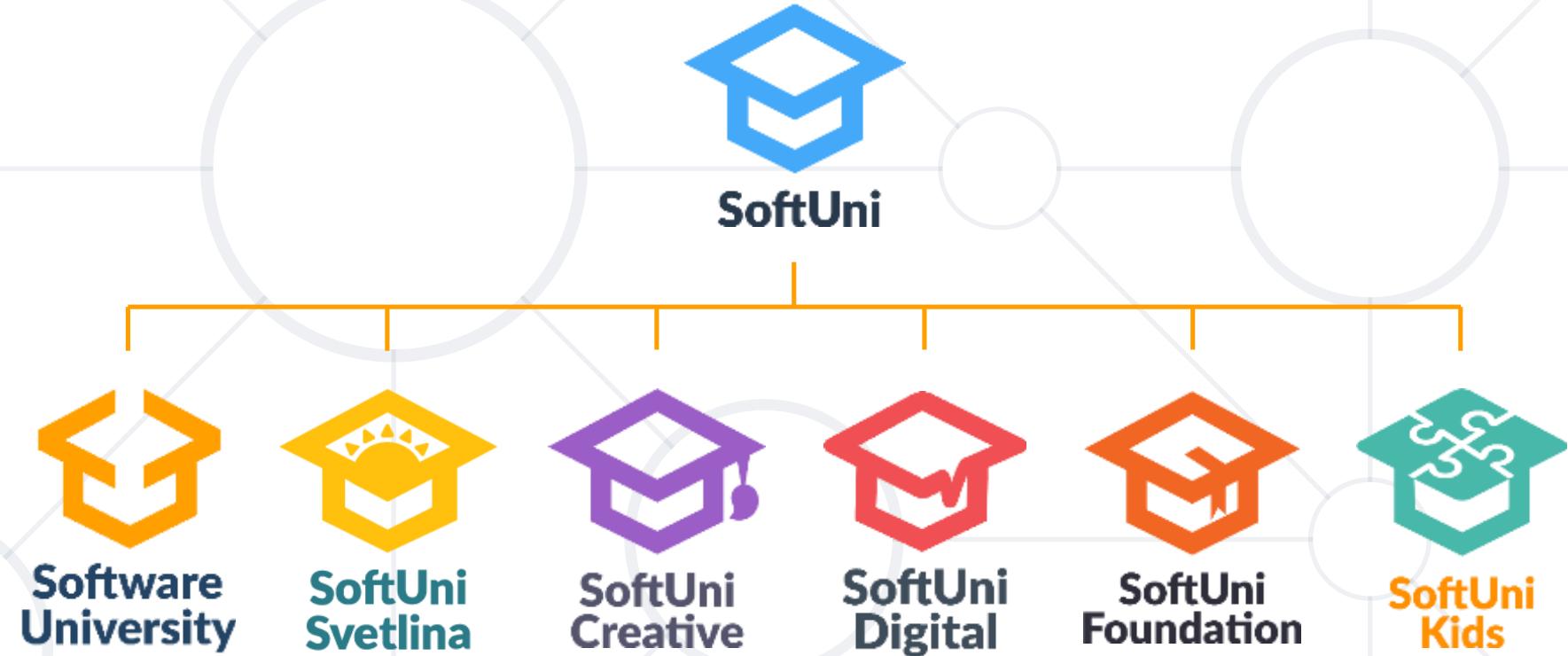
- SQL Server lets us create **custom roles**
 - **Collection of privileges** (permissions)
- Fine **control over permissions**
 - Can use **one role for multiple users** (groups)
- Makes **auditing operations easier**



- **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



Questions?



SoftUni Diamond Partners



SCHWARZ



Coca-Cola HBC
Bulgaria



Postbank

Решения за твоето упре



**SOFTWARE
GROUP**

Bosch.**.IO**



SmartIT



**PHAR
VISION**



CAREERS



INDEAVR
Serving the high achievers



**SUPER
HOSTING
.BG**

Educational Partners



Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg, about.softuni.bg
- Software University Foundation
 - softuni.foundation
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



Software
University



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

