Learn SQL Server Intuitively
Transact-SQL: The Solid Basics

Author
Peter Lalovsky

Technical Editor
Marleen Banton

Book Layout
George Yordanov

Cover Design
Svetlana Safonova

Series Concept
Peter Lalovsky

Proofreader
Marleen Banton

Paperback ISBN
ISBN-13: 978-0995245105

Contacts and Source Code
LearnIntuitively.Lalovsky.com/SQLServer

Audience
Beginner/Intermediate

Table of Contents

Table of Contents

Conventions Used in This Book

The following typographical conventions are used in this book:


Good practice


Bad practice


Pay attention


Hint


Reference in the book


Comment, clarification and explanation


Result of the execution of the code

Linking tables

Sequence

Intro

Why learn "intuitively"?

Looking at colors and forms and making connections between them creates associations in the brain.
The brain links the colors and forms by using its natural attribute – the intuition.

The following example will give you an exact explanation of the above statement.

Blue
Red
Green

This book Next you can I love add more and starts from more SQL Server - scratch and and more knowledge one of the to the solid gives you most powerful background, databases in a solid base built by the world. in SQL Server. this book.

You know what to do and you did it. Your intuition helped you with this simple exercise.

Metadata is data about the data. It explains the data.

We can do this:
This book Next you can I love add more and starts from more SQL Server - scratch and and more knowledge one of the to the solid gives you most powerful background, databases in a solid base built by the world. in SQL Server. this book.

A collection of information, stored on paper or in electronic format.
A database, created by Microsoft Corporation® and widely used worldwide.

Metadata

or this:

The database (DB) contains tables that store data. One of the usage of the DB is to serve webpages.

Here we declare an abbreviation

Here we use the abbreviation

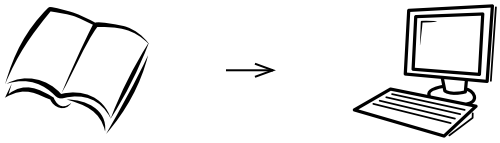
This book will help your intuition to teach you in Transact-SQL programing.

Database (DB)

The Database (DB) is an electronic warehouse where we store data. The data in the DB is a representation of information that exists in the real world and it is strictly organized. Our emails, social networks, bank accounts, medical or school records are stored in DBs. We can store literally any information in a DB.

The following examples will help you draw the big picture.
Let's move our contacts from the old notebook (paper) to a file (electronic).
First we create a text file like this:

```
Name: Anna Laurier
Email: anna.laurier@customer2.net
Phone: +1 555 2222

Name: John Smith
Email: john.smith@customer1.com
Phone: +1 555 1111
```



After we add a few contacts, we notice that repeating **Name**, **Email** and **Phone** for every single contact is time and space consuming. We convert the format of the text file to this:

```
Name           Email                        Phone
Anna Laurier   anna.laurier@customer2.net   +1 555 2222
John Smith     john.smith@customer1.com     +1 555 1111
```

Our next step is moving the data from the text file to a spreadsheet like **Microsoft Office Excel** or **LibreOffice Calc** - the simplest database.
The spreadsheet allows us to handle the information much easier than the text file - insert every slice of data in its own cell, quick search, add filters on the columns, order the data etc.

Name	Email	Phone
Anna Laurier	anna.laurier@customer2.net	+1 555 2222
John Smith	john.smith@customer1.com	+1 555 1111

After we start using the spreadsheet (actually this is our first DB), we add multiple emails and phones for a single contact:

Name	Email	Phone
Anna Laurier	anna.laurier@customer2.net, al@customer2.net	+1 555 2222, +1 555 2223
John Smith	john.smith@customer1.com, js@customer1.com	+1 555 1111, +1 555 1112, +1 555 1113

Oups! We discovered our first data issue: we used a comma (,) to split the data, but the comma itself is data. We fix this issue by splitting the data in separate cells:

Name	Email 1	Email 2	Phone 1	Phone 2	Phone 3
Anna Laurier	anna.laurier@customer2.net	al@customer2.net	+1 555 2222	+1 555 2223	
John Smith	john.smith@customer1.com	js@customer1.com	+1 555 1111	+1 555 1112	+1 555 1113

Later, we search for a contact and we are confused: Column **Phone 1** is a mobile or business phone? The same question stands for column **Email**. We add additional columns to define the type on the phones and emails:

Name	Email 1	Email 1 Type	Email 2	Email 2 Type	Phone 1	Phone 1 Type	Phone 2	Phone 2 Type	Phone 3	Phone 3 Type
Anna Laurier	anna.laurier@ customer2.net	Personal	al@custom-er2.net	Business	+1 555 2222	Home	+1 555 2223	Mobile		
John Smith	john.smith@ customer1.com	Business	js@custom-er1.com	Business	+1 555 1111	Mobile	+1 555 1112	Mobile	+1 555 1113	Business

After one year of using the spreadsheet, we notice that a person has 50 phones and we added 100 columns (Phone and Phone Type pairs), but... what if we split the attributes (Email and Phone) into separate tables and one phone for one contact is one row?
The next step is to convert our contacts DB from one table that stores all the data to multiple related tables:

Emails

ContactName	Email	Type
Anna Laurier	anna.laurier@customer2.net	Personal
Anna Laurier	al@customer2.net	Business
John Smith	john.smith@customer1.com	Business
John Smith	js@customer1.com	Business

One contact, many Emails

Contacts

ContactName
Anna Laurier
John Smith

One contact, many Phones

Phones

ContactName	Phone	Type
Anna Laurier	+1 555 2222	Home
Anna Laurier	+1 555 2223	Mobile
John Smith	+1 555 1111	Mobile
John Smith	+1 555 1112	Mobile
John Smith	+1 555 1113	Business

We just built our first **One-to-Many** relation – One Contact may have multiple phones and emails.

In few months we decide to add description to the phone and email types. We find that it will be easier to manage separate tables for email and phone types and link them to the tables that store email and phone. We also decide to add pictures for our contacts. As we decide to show only one picture per contact, we add column **Picture** in table **Contacts**.

The other significant change is the replacement of the long column headers **ContactName**, **EmailType** and **PhoneType** with short numbers, used to link the tables.

EmailTypes

EmailTypeID	EmailTypeName	EmailTypeDescription
1	Personal	Send jokes, pictures and any other personal data.
2	Business	Related to business conversations. Send contacts and invoices.

Emails

ContactID	Email	EmailTypeID
1	anna.laurier@customer2.net	1
1	al@customer2.net	2
2	john.smith@customer1.com	2
2	js@customer1.com	2

Contacts

ContactID	ContactName	Picture
1	Anna Laurier	1-1.png
2	John Smith	2-3.png

`Link on ContactID`

Phones

ContactID	Phone	PhoneTypeID
1	+1 555 2222	1
1	+1 555 2223	2
2	+1 555 1111	2
2	+1 555 1112	2
2	+1 555 1113	3

PhoneTypes

PhoneTypeID	PhoneTypeName	PhoneTypeDescription
1	Home	Stationary at home. Used by all family members.
2	Mobile	Available all the time.
3	Business	In business hours only. Located in the office.

The last step in our example is to transform the data in the DB to “human readable” reports like this

Anna Laurier

Emails

anna.laurier@customer2.net (Personal)
al@customer2.net (Business)

Phones

+1 555 2222 (Home)
+1 555 2223 (Mobile)

John Smith

Emails

john.smith@customer1.com (Business)
js@customer1.com (Business)

Phones

+1 555 1111 (Mobile)
+1 555 1112 (Mobile)
+1 555 1113 (Business)

or parameterized reports that will give us the option to search the contacts by City, Company, Job Title and any other way that we can imagine, based on the available data.

Already confused? It’s OK. The best thing about the database is that it is a never ending learning adventure! Let’s stop complicating the example for now. We learned that:

- The DB stores data in electronic format
- The data in the DB is strictly organized
- The data in the DB relates to each other

Software and Vendors

We use software to manipulate the data in the DBs. This software is an interface (the point where two systems cross) between the DBs and the DB professionals. It is known as RDBMS (Relational Database Management System). Some of the most popular RDBMS are:

Vendor Name	Software Name
Microsoft Corporation	SQL Server
Oracle Corporation	Oracle
IBM	DB2
SAP	Sybase
MySQL AB	MySQL

Download and Install SQL Server 2016 Express

This book is for Microsoft SQL Server© – one of the most powerful RDBMS. SQL Server is released in different editions that serve different needs - Enterprise, Standard, Web, Developer and Express.
The **Express** edition is **free**. It doesn't support all the functionalities of the other editions, but it includes all that we need in this book. The another version - **Developer** edition - is also **free**. It includes all the functionalities of the Enterprise edition, but serves only development and testing. It can not be used on a production server.

The **Express** edition has different components. We need to download and install the DB engine (DBE) and the main tool that we use to manage the data and the objects in SQL Server – SSMS (SQL Server Management Studio).

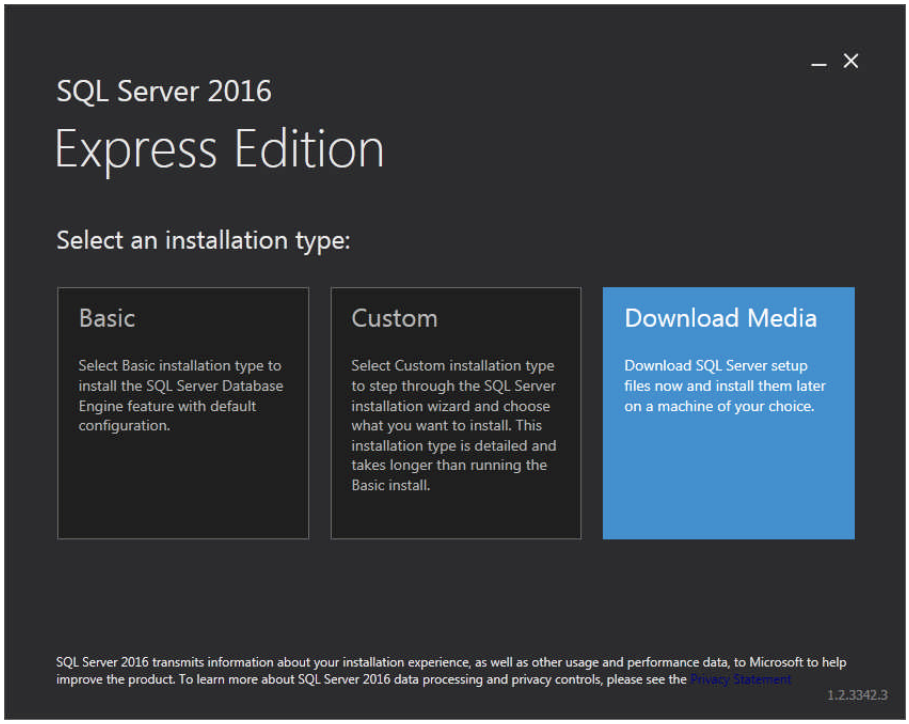
Before starting the installation, we need to verify that our computer meets the minimum requirements.

Hardware:

- Processor: x64 only, 1.4 GHz
- Memory: 512 MB
- Storage: 6 GB free disk space, NTFS file format
- Internet connection

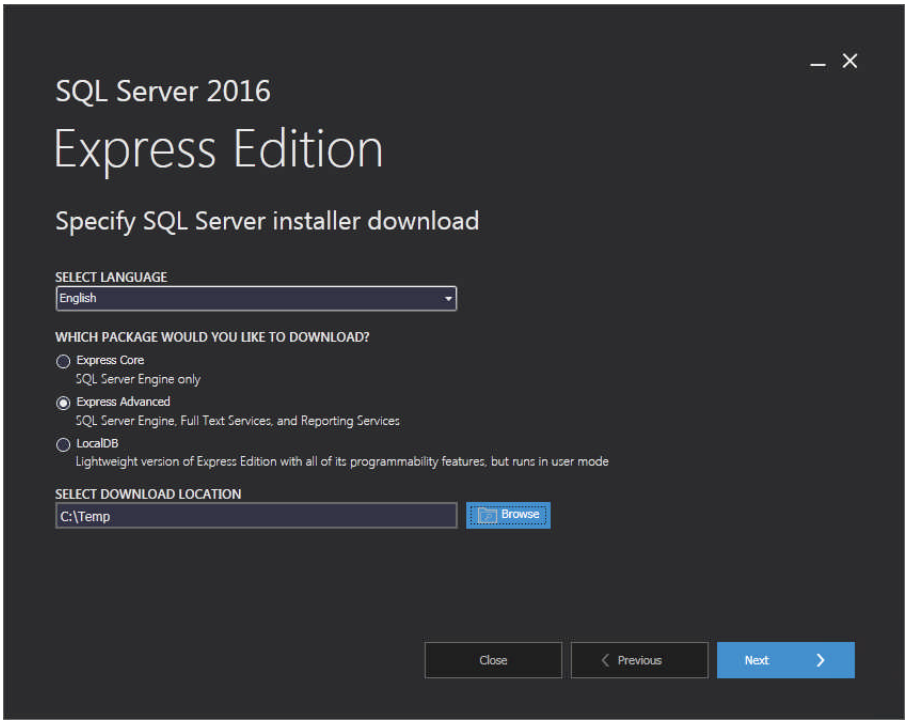
Software:

- Operating system - Windows 8 or Windows Server 2012



From the Microsoft's website (https://www.microsoft.com/en-ca/server-cloud/products/sql-server-editions/sql-server-express.aspx) select **Try SQL Server 2016 Express for free**. Select your preferred language and click Download. Save the installer **SQLServer2016-SSEI-Expr.exe** on your hard drive. To start the installation, Double click on this file.

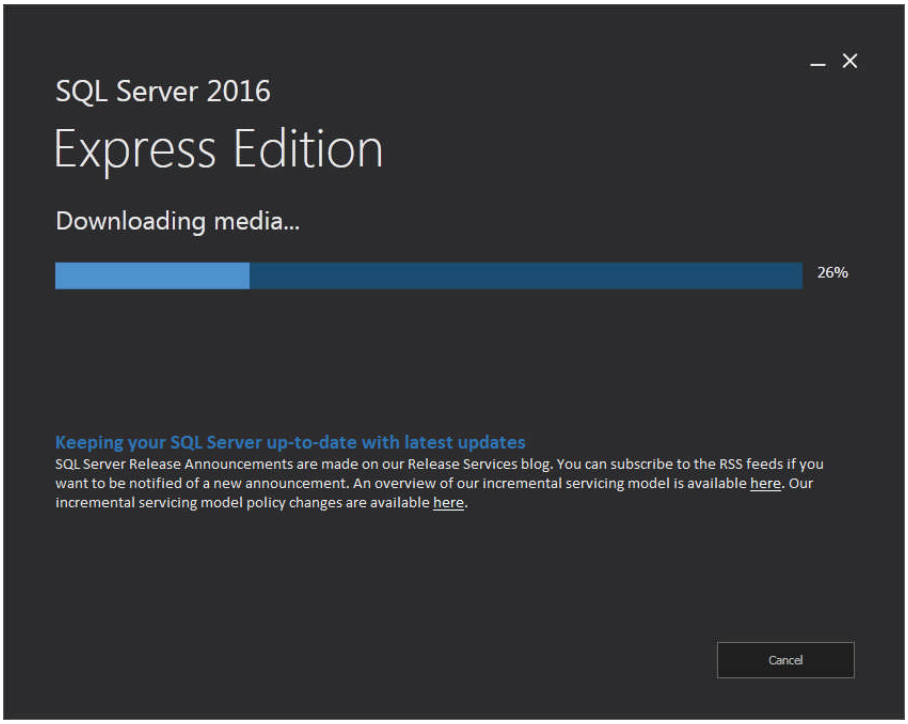
Click **Download Media**.



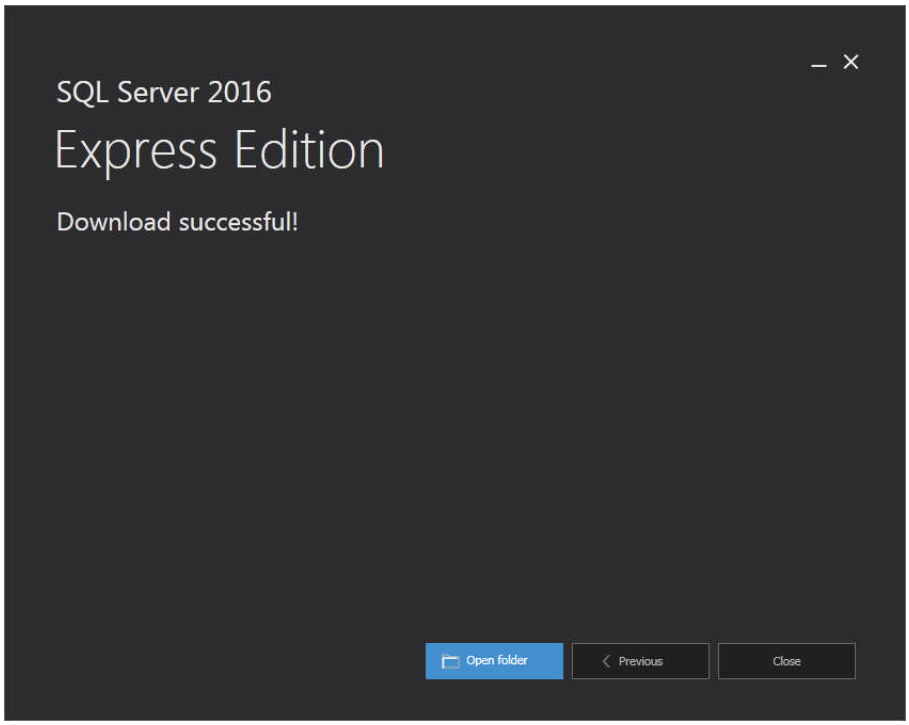
Click:

- SELECT LANGUAGE: English
- WHICH PACKAGE WOULD YOU LIKE TO DOWNLOAD?: Express Advanced
- SELECT DOWNLOAD LOCATION: *Location of your choice*
- Next

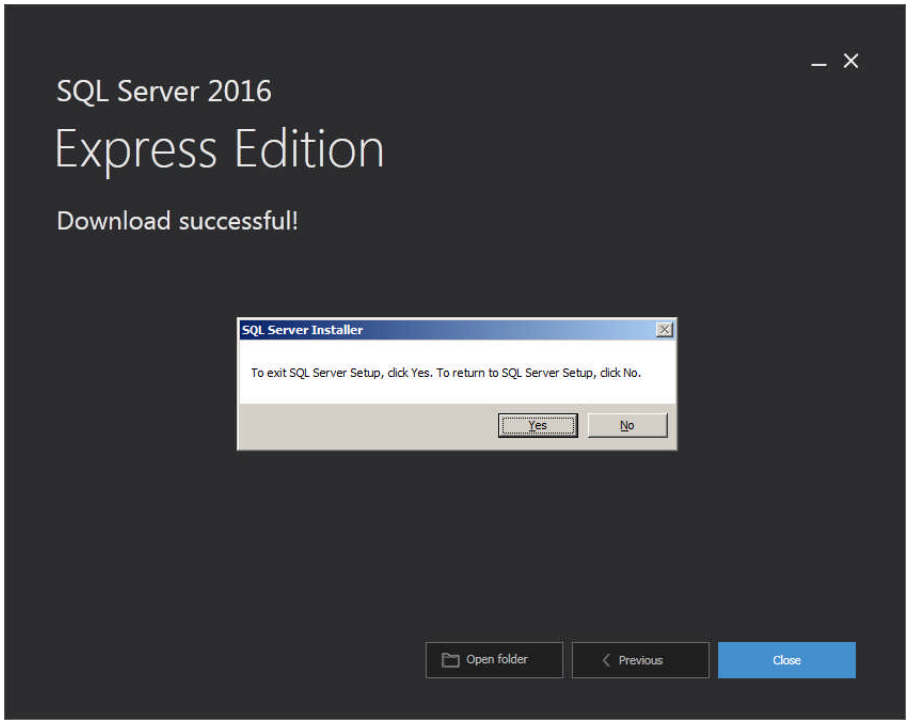
In my installation I use **C:\Temp** as my working directory. This is **not mandatory** and you can create your own working strategy.



Download the SQL Server 2016 Express Database Engine (DBE).



Click **Open Folder**.

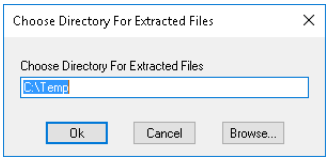


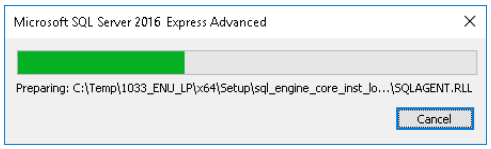
Click **Yes** and **Close**.

In the folder of our choice (C:\Temp) the installer **SQLEXPRADV_x64_ENU.exe** is downloaded.

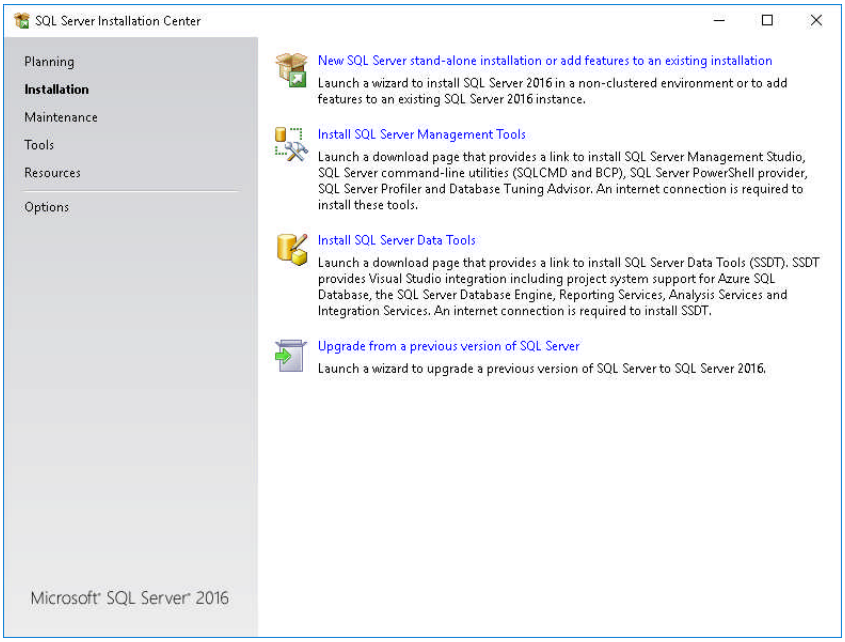
I move the file to another folder, so C:\Temp is empty for the next steps and double click it.

Installation



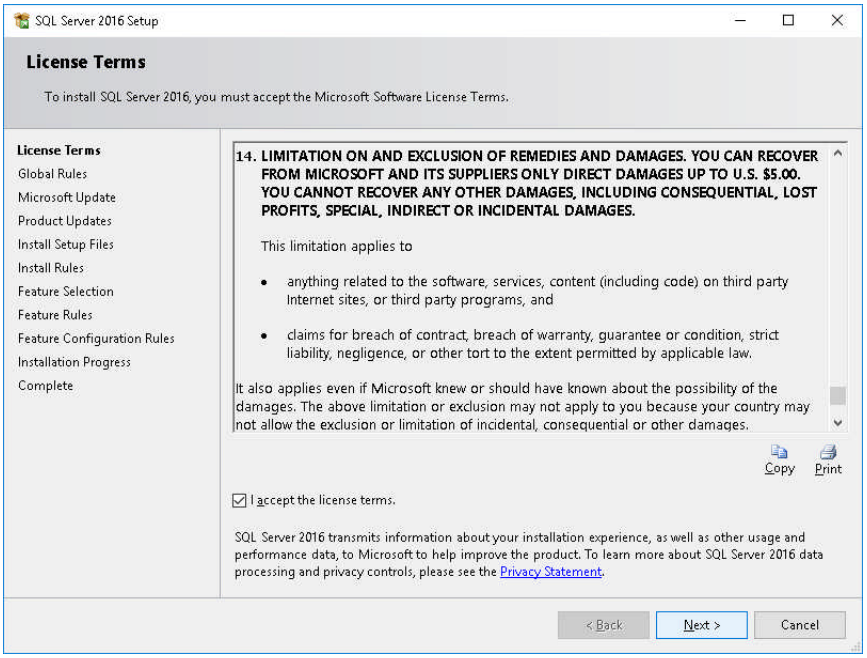


Unzip the installation files in a directory of your choice.

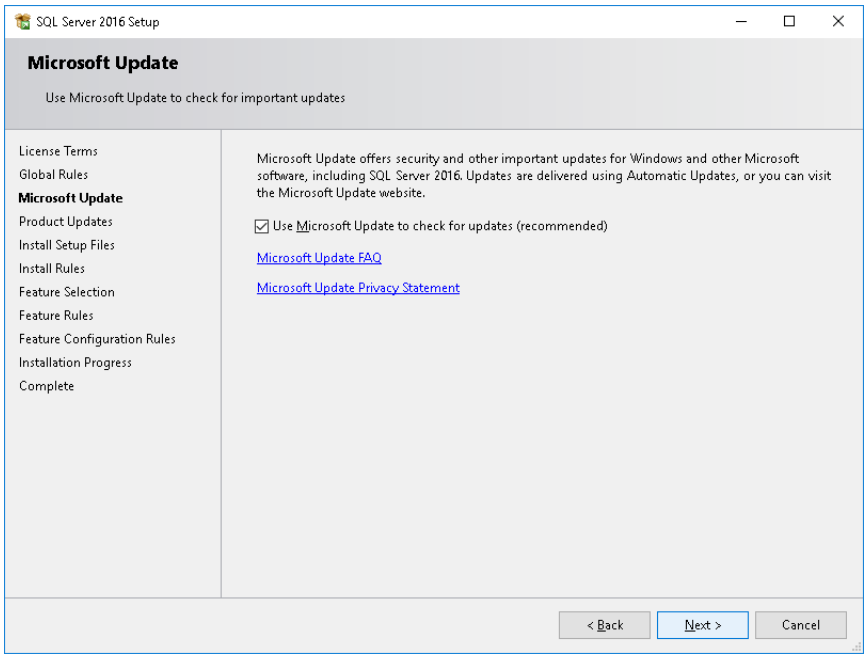


To download the DBE, click on **New SQL Server stand-alone installation or add features to an existing installation**.

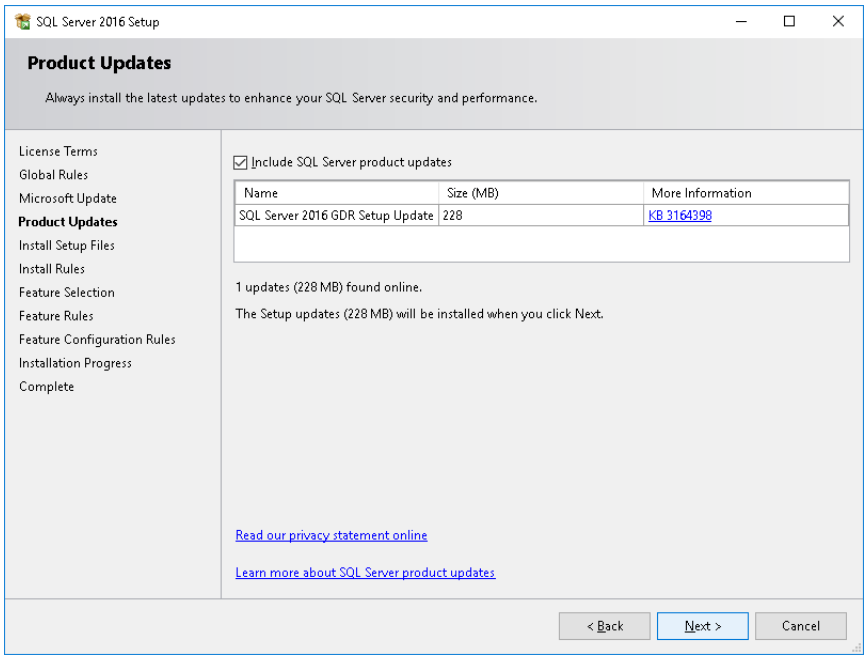
After we finish the installation of the DBE, we'll come back to this window to download SSMS ⟶ Install SQL Server Management Tools.



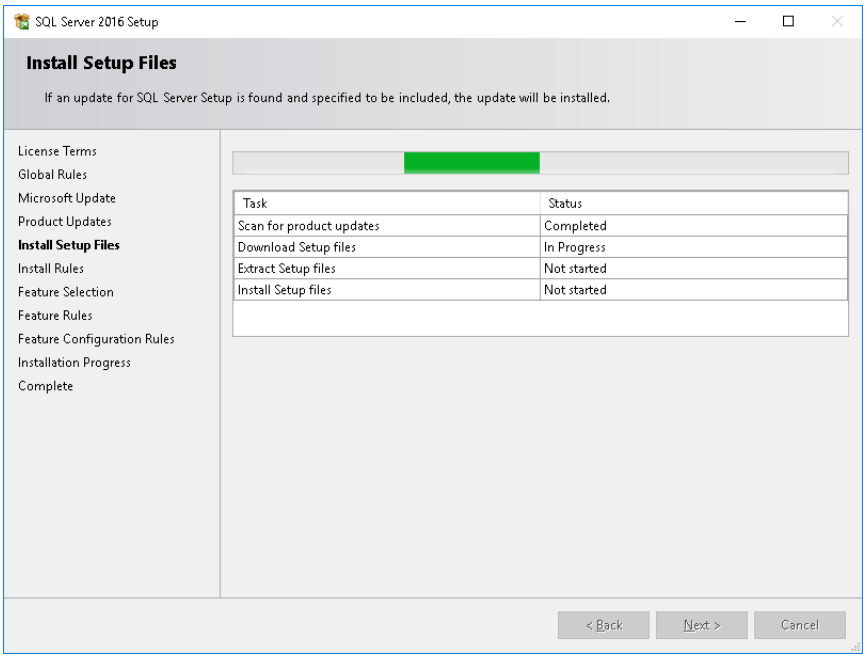
Read the **License Terms** and check **I accept the license terms** to continue the installation.



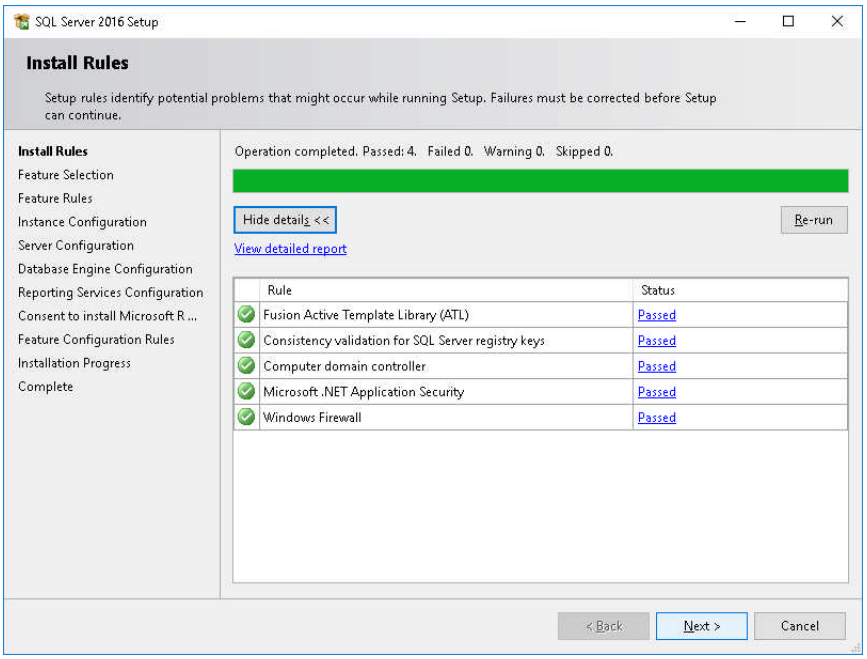
Click **Use Microsoft Update to check for updates (recommended)**.



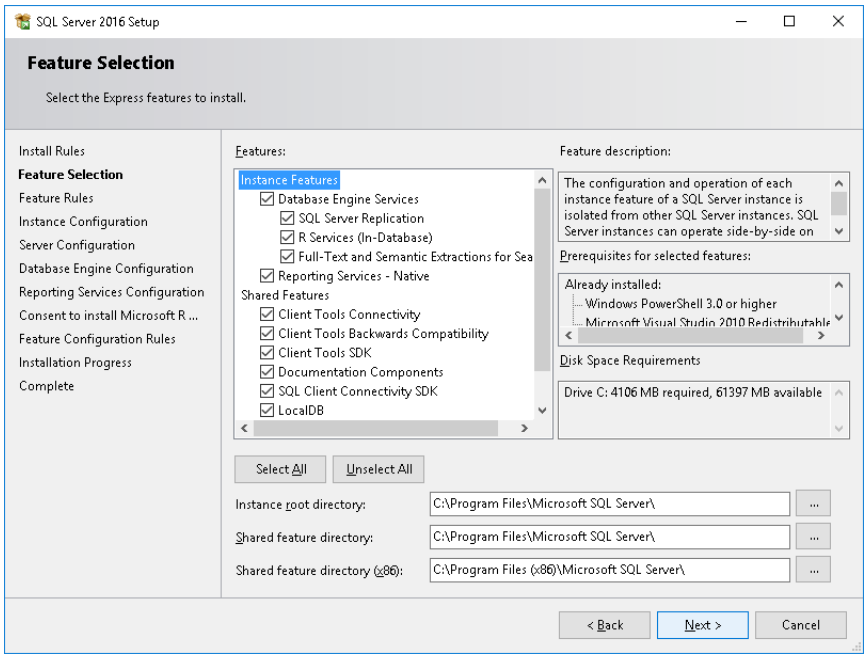
If your computer needs any additional update(s), they are listed here.



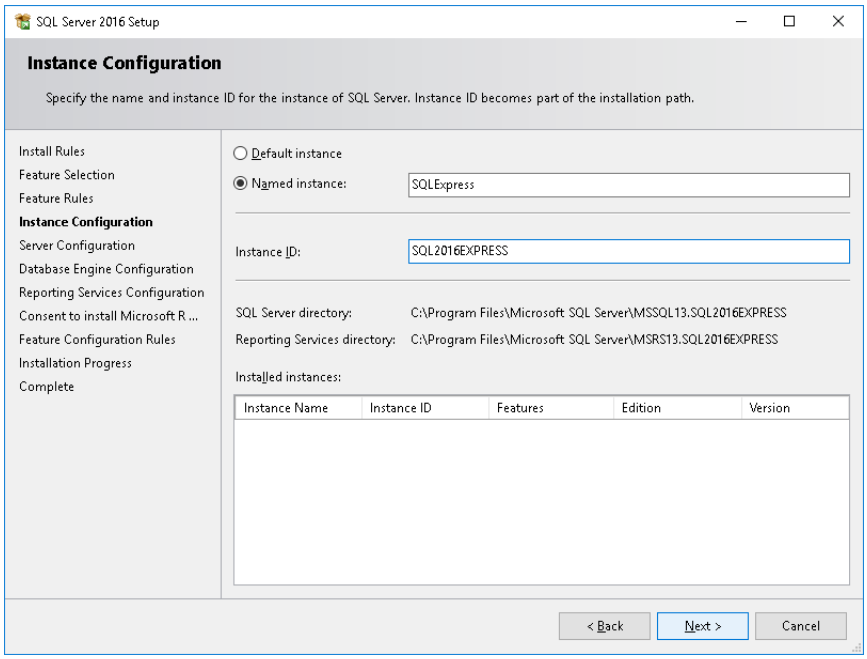
The next window **verifies the necessary updates**, downloads and installs them.



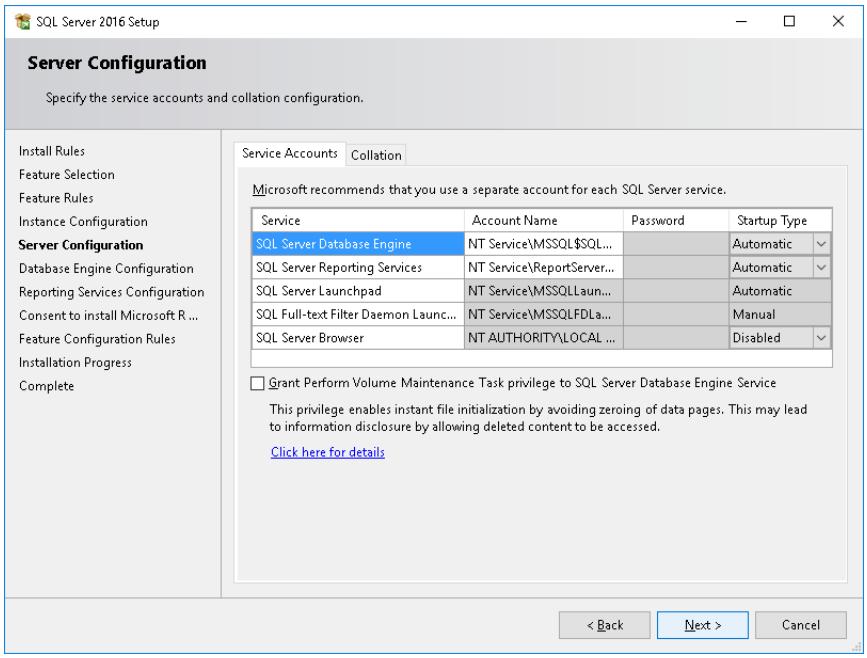
On the next screen all the checks **have to be green**. If any of them is not, we need to make the required fix(es) (for example: edit the firewall settings).



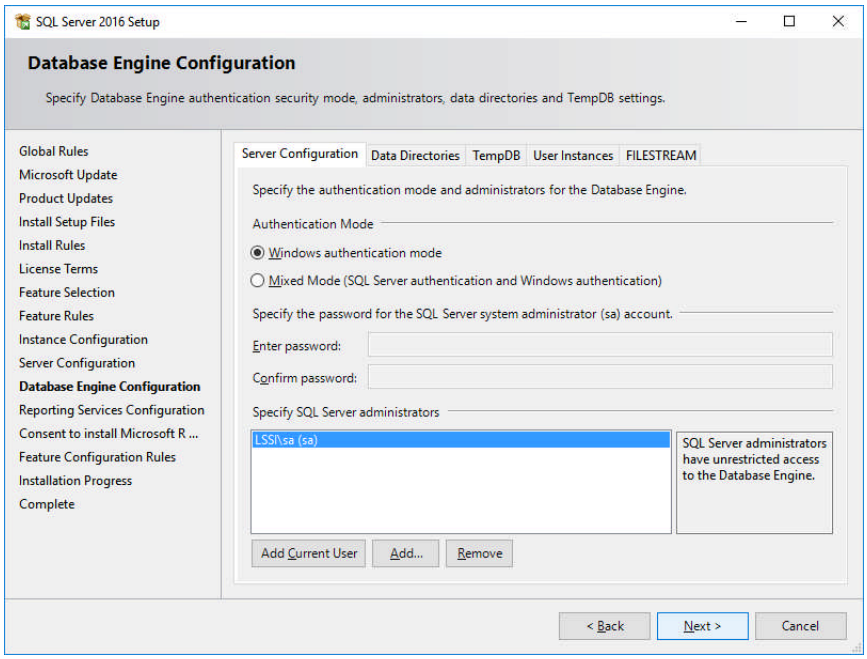
Next we choose **the features that we want to install**. If we install features that we don't need now, but will need in the future, when we need them, they will be there and ready to use. A good example is **R Services** or **Reporting Services**.



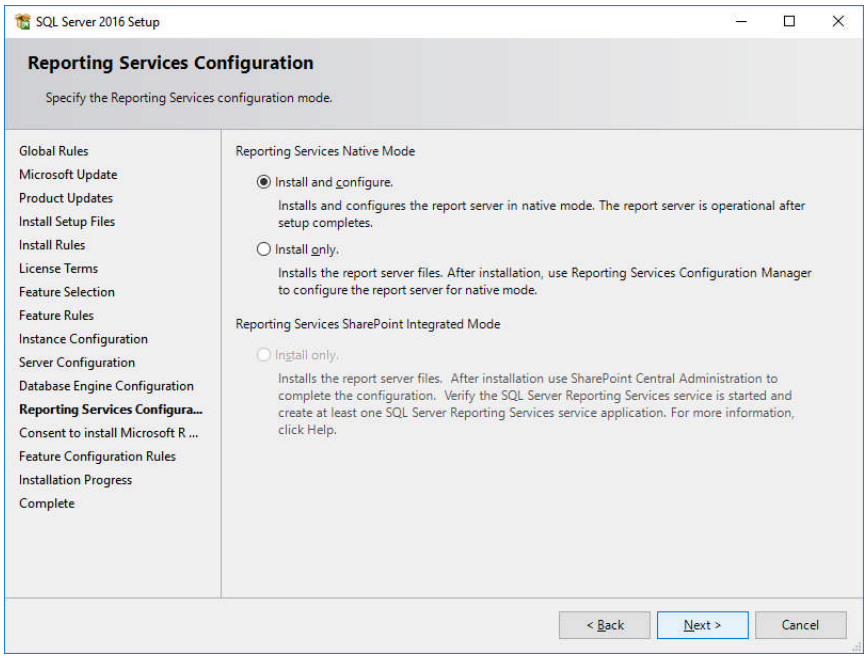
One computer (server) can run **multiple SQL Server instances**. In this step we define a name for our instance - **SQL2016EXPRESS**.



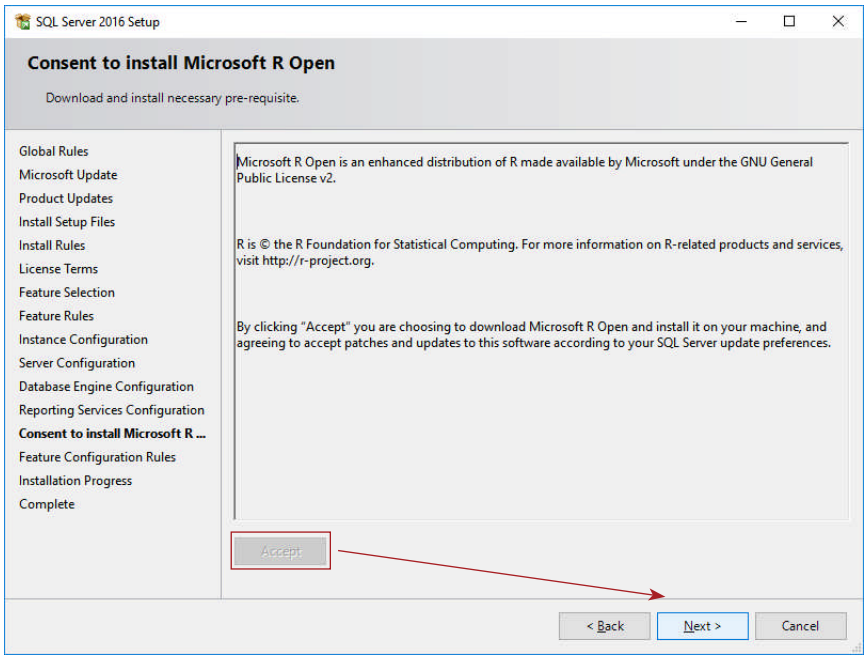
On Server Configuration step, we define how to **start up** the SQL Server services and which **account** to use for each service.



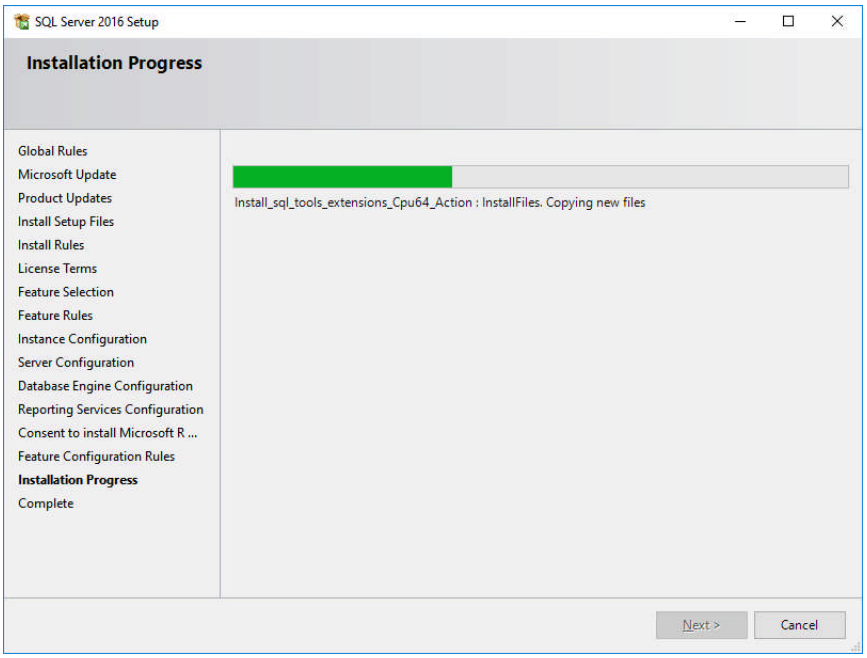
In the next step we configure the **authentication mode** - **windows** (the account, with which we are logged in Windows) or **mixed** (windows accounts and users, created by us).



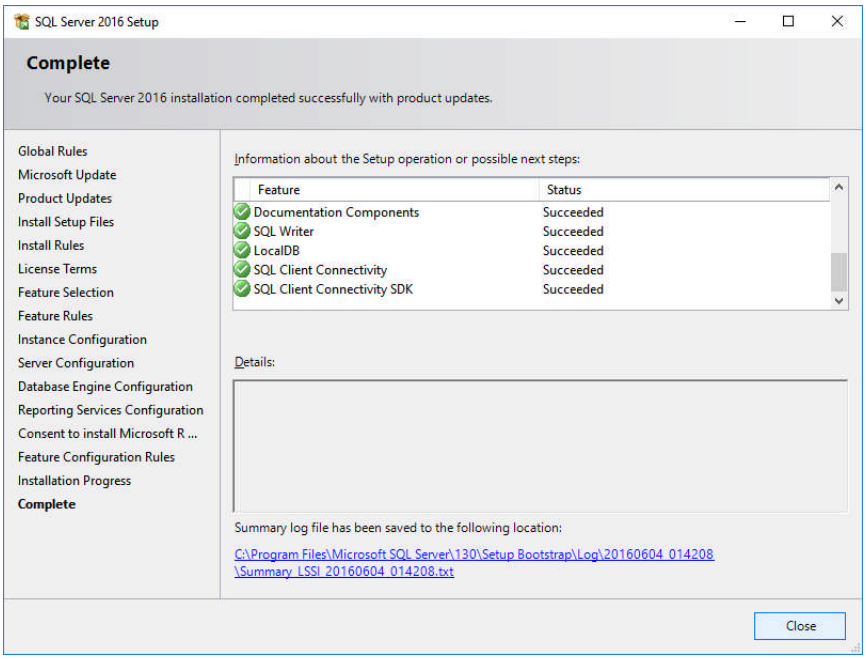
This step takes care of the configuration of SQL Server Reporting Services (**SSRS**).



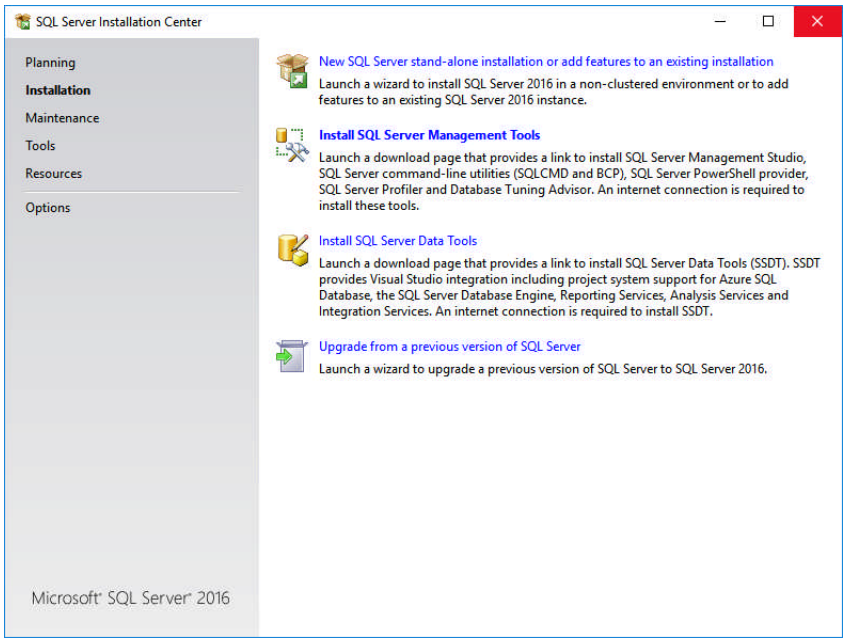
In this step we accept to download and install **Microsoft R Open**.
Click **Accept** and **Next**.



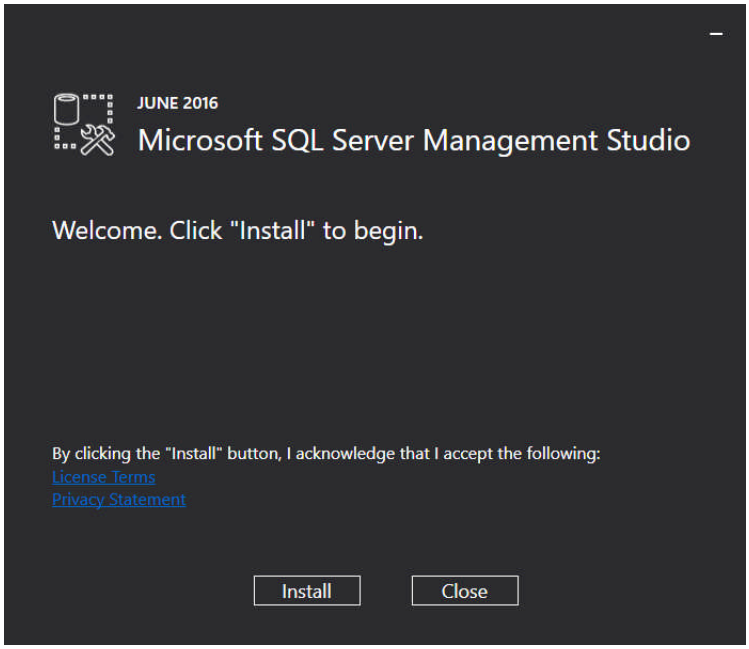
The installation starts.



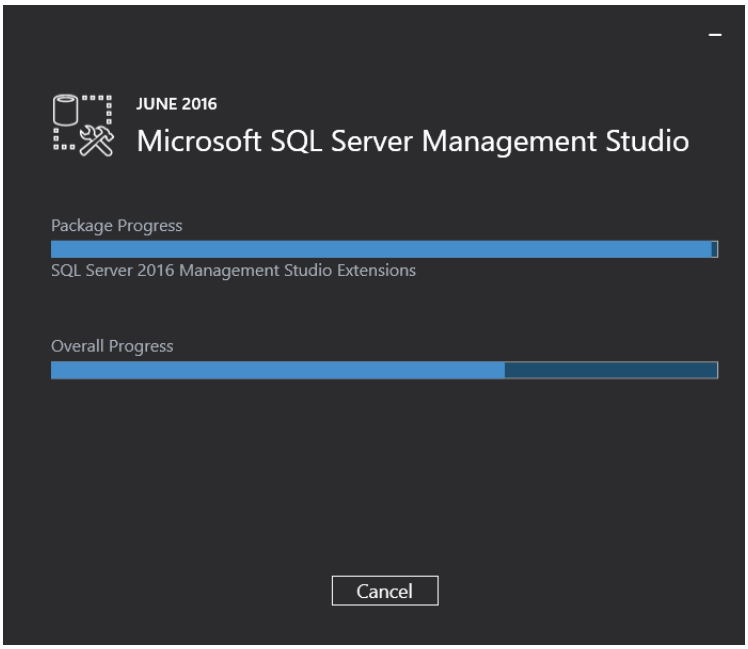
Scroll down and make sure that **all the checks are green**.



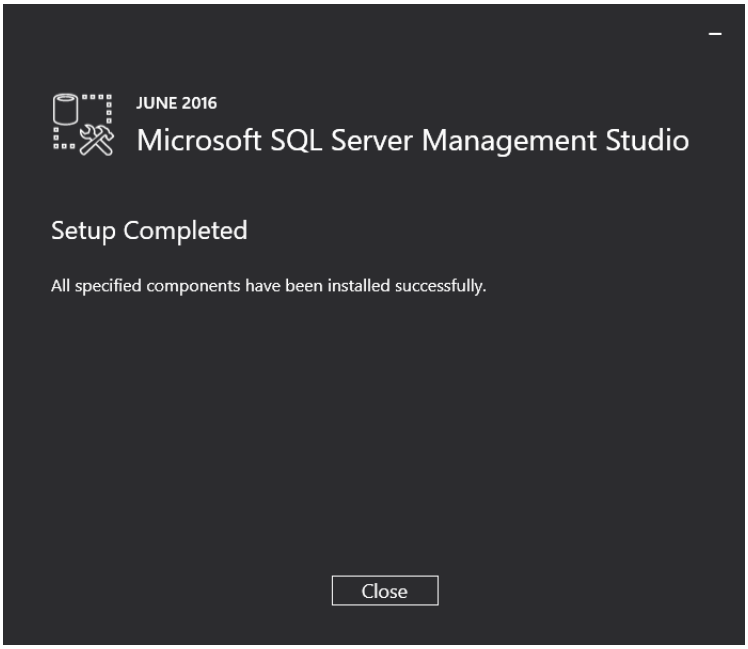
Delete all the files in the working directory (C:\Temp) and click on **Install SQL Server Management Tools** to download SSMS.
In the Microsoft's web page click on **Download SQL Server Management Studio** and save the installer **SSMS-Setup-ENU.exe** in a location of your preference (in my installation C:\Temp).
Double click on **SSMS-Setup-ENU.exe**.



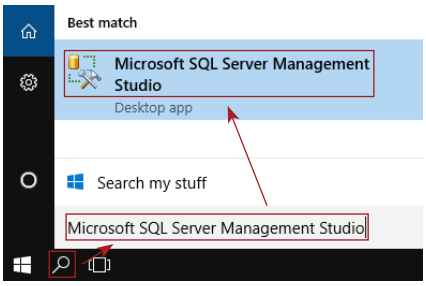
Click **Install**.



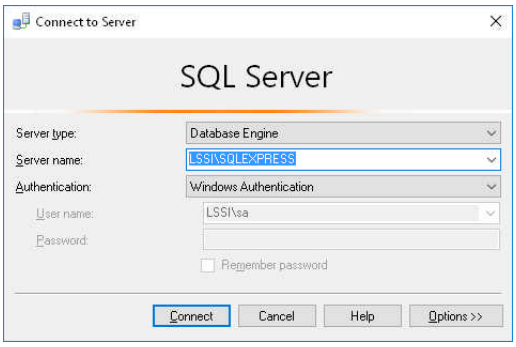
The installation is in progress.



Finally click **Close**.



After the installation is complete, browse **Microsoft SQL Server Management Studio** in the **Search Windows**. This is our interface to the DB. Here we compose and run our code in the programming language that communicates with the DBE - SQL.



Click **Connect** to open SSMS.

SQL Server Management Studio (SSMS) on page 75

Working with DB (DB Professional Roles)

DB Professional Role

The DB needs to be:

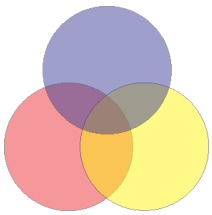
- **Created**:
  - Install the Database Engine (DBE) and the necessary components (SSMS)
  - Create data models and DB objects for storing and manipulating the data
- **Queried**:
  - Select, add, modify and delete data
  - Create, edit and delete DB objects
- **Maintained**:
  - Keep the DB up and running 24/7
  - Create a redundancy copy of the data (backup)
  - Optimize the performance of the DB and the load of the DB server

These tasks are done by DB professionals and are divided in 3 main roles:

- **Database Administrator (DBA)**:
  - Installs and configures the RDBMS
  - Schedules backups and restores DBs and automations
  - Replicates data between multiple DBs
  - Monitors and optimizes the load of the DB server
  - Manages the DB security (groups and users)
- **Database Architect**:
  - Designs the conceptual, logical and physical data models
  - Creates DB objects, entities and relations
- **Database Developer** – Creates the Transact-SQL code which:
  - Connects the DB with the interface that manipulates the data:

Interface	DB
Bank cash machine (ATM)	Your bank account
Your browser	Your email

  - Manipulates data internally in the DB, between different DBs and the exchange of data with another data sources (API to another DBs, flat files on FTP, etc.).
  - Extracts data from different sources, transforms data and loads it into destination tables.
  - Serves internal automated processes such as to send emails from the email queue every 2 minutes and move data to an archive table.



These roles cross each other.

- The DBA can suggest an optimization of code to the developer
- The developer can create a DB architecture
- The architect can build a model that allows the developer to create the most efficient code

Operational Systems - OLTP (On-line Transaction Processing)

Data in the DB is multifarious. It is generated by multiple IT systems:

- Customer Relationship Management (CRM)
- Enterprise Resource Planning (ERP)
- Warehouse Management System (WMS)
- Standalone or web application like ATM or web application that manages the move of goods inside the warehouse
- Custom built web applications
- Files exchanged with business partners
- Any other source of data that you can imagine

These systems are grouped in the **OLTP** (On-line Transaction Processing). They execute a lot of on-line transactions to generate data.

The following example shows the main elements in the process of data generation in OLTP.

Customer 1: Online sale

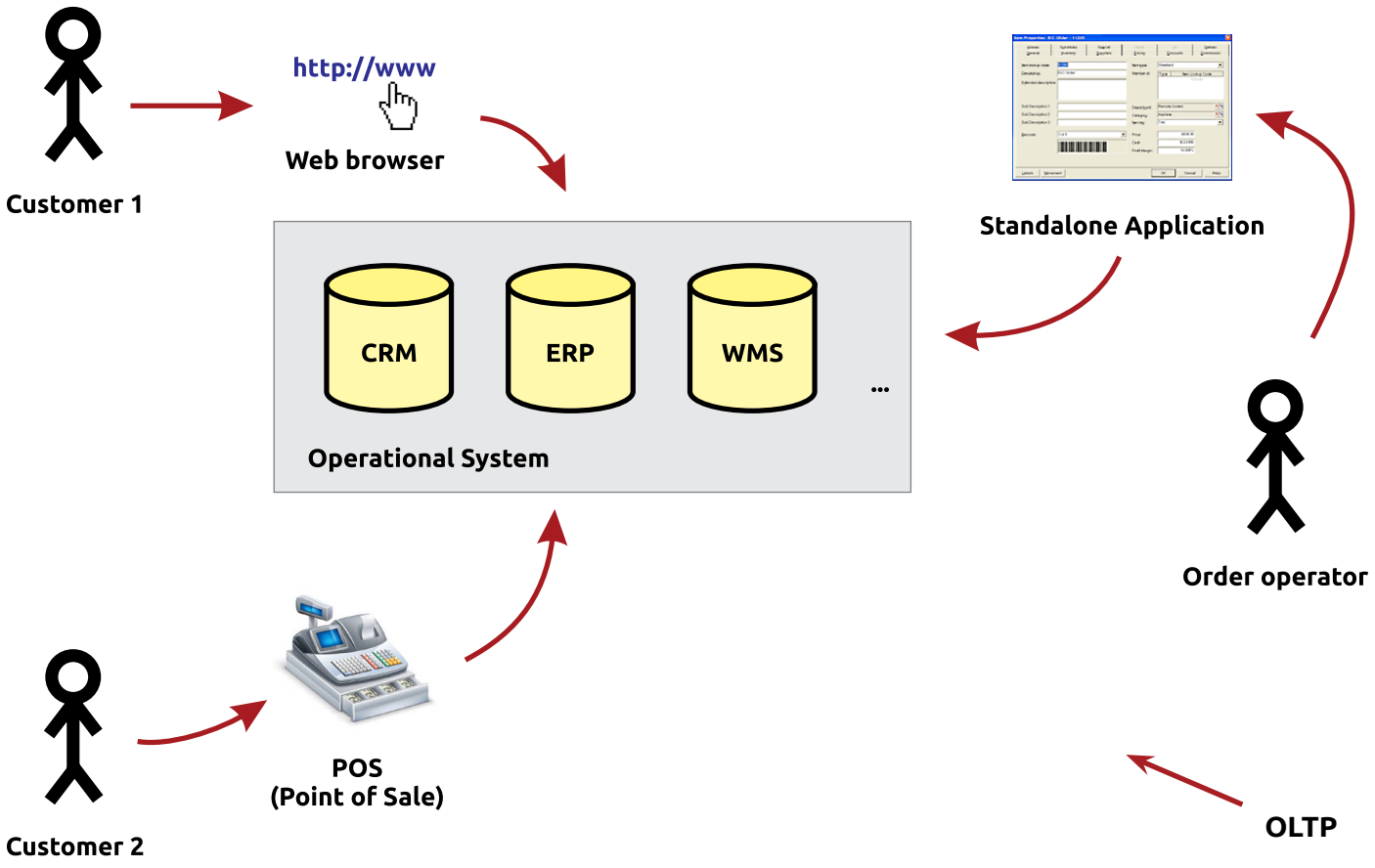
1. DB Interface: Web browser
2. Transactions:
    a. Read the customer data from CRM
    b. Create order in ERP
    c. Manipulate the ordered item(s) in WMS
    d. Send an email to the customer

Customer 2: Sale in our store

1. DB Interface: POS Terminal (cash register)
2. Transactions:
    a. Create order in ERP
    b. Manipulate the ordered item(s) in WMS
    c. Select the data, related to the order to print a bill

Order operator: internal order processing

1. DB interface: Standalone application
2. Transactions:
    a. Process orders in ERP
    b. Update quantities and locations on item(s) in WMS
    c. Insert data in ERP to bill the customer



ETL (Extract, Transform, Load)

To make business decisions and analyses based on data created in OLTP, you need to cleanse, calculate and aggregate the data... in one word **transform** it.

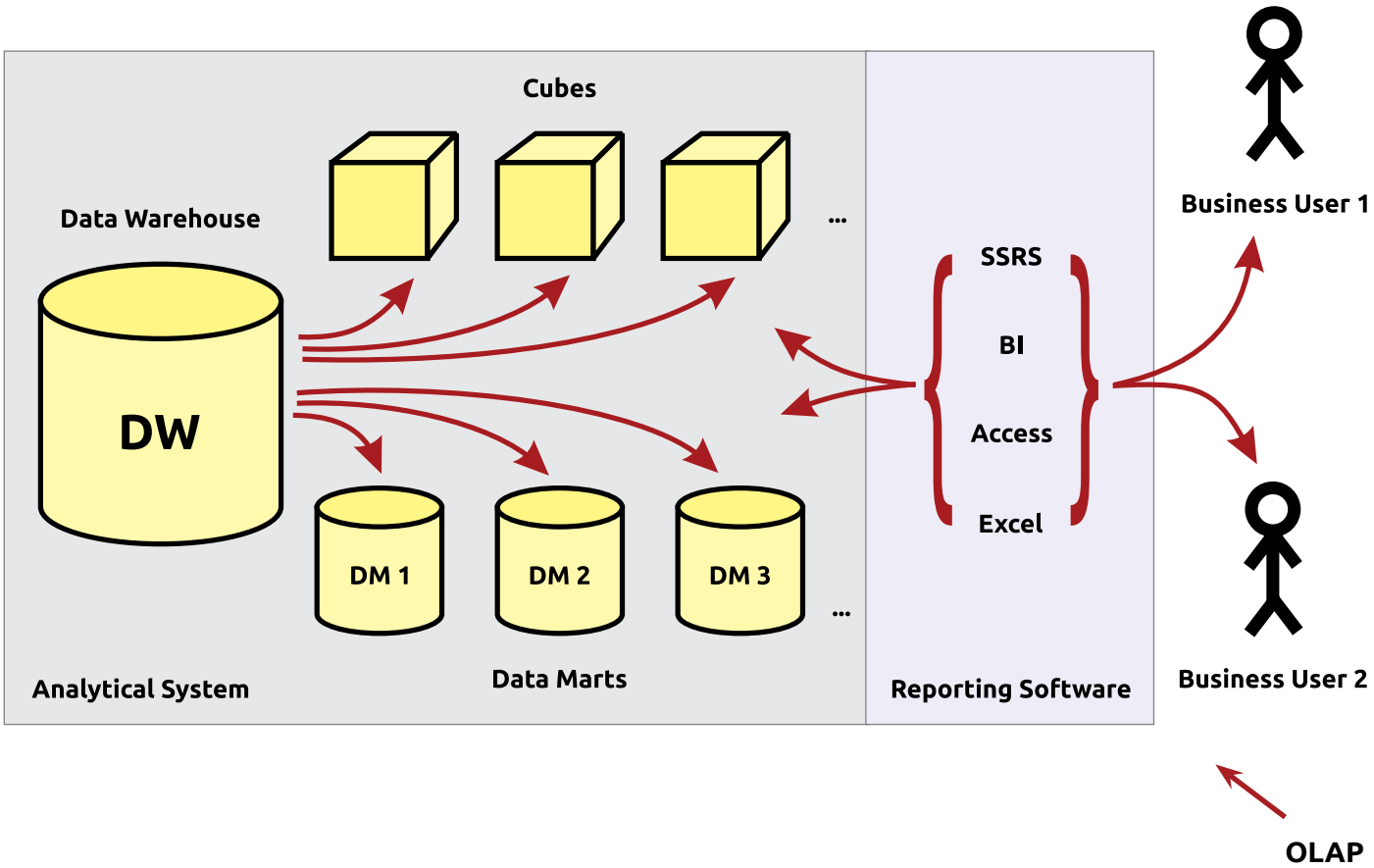
- How much did our sales people sell last month?
- Which 5 marketers signed the most new contracts last year?

These are questions that lead to business decisions.

The process that **E**xtract, **T**ransform and **L**oad data from the OLTP systems to the destination system is called **ETL** – Extract, Transform, Load. This process is necessary because the OLTP system generates a lot of data that is not useful for the analyses (corrected lines in orders, internal moves of items, etc.), unclean data (data for non existing items, garbage from bad transactions, etc.) or data from different sources, that needs to be joined for the analyses.

Analytical Systems - OLAP (On-line Analytical Processing)

The ETL loads data into the Data Warehouse or **OLAP** (On-line Analytical Processing) system. The data in the OLAP is organized into the **Data Warehouses** and **Data Marts**. It can be sliced and dice in any possible way. We query the OLAP system to build reports that help the business to make decisions.



The dataflow between the DB systems:

How the DBE is Structured

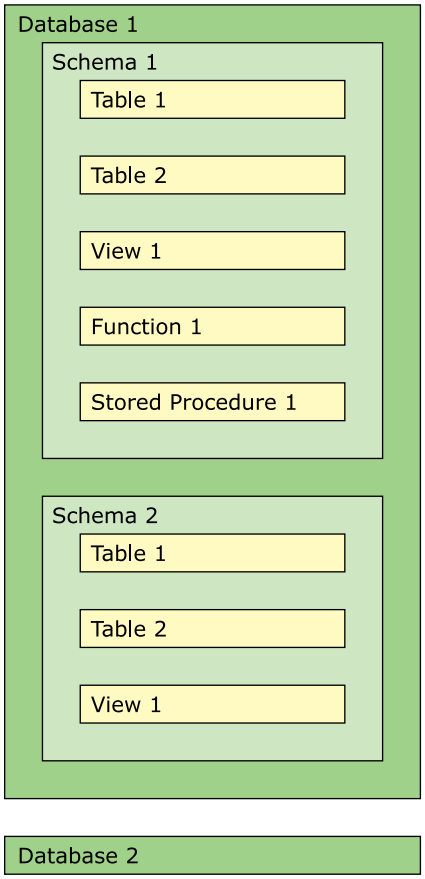
Microsoft SQL Server is a RDBMS that includes several components:

- Database Engine (DBE) - The DB
- SSRS (SQL Server Reporting Services) - Creates, runs and automates reports
- SSIS (SQL Server Integration Services) - Connects the DB to the world by import, export, transformation and load of data and automations
- SSAS (SQL Server Analysis Services) - Creates OLAP systems

and tools:

- SSMS (SQL Server Management Studio) - Visual tool for writing and executing SQL code and managing DB objects
- Profiler - Tracing and monitoring
- Tuning Advisor - Helps us to improve the performance of the DB, by suggesting sets of indexes

In this book we'll need only the DBE and SSMS.

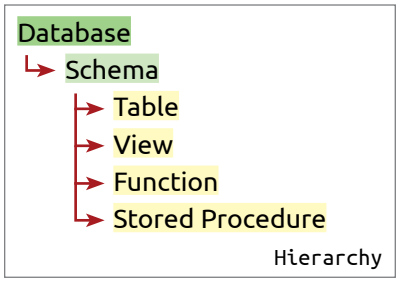


The DBE is built by different components which serves different needs:

- Database - Collection of structured and correlated objects that store data and SQL code

- Schema - Container that isolates sets of DB objects
- Table - Stores data in rows and columns
- View - Shows sliced and diced data from the tables
- Function - Reusable code that is doing a specific task
- Stored Procedure - Reusable multi-step and complex code

DB objects (DBO)

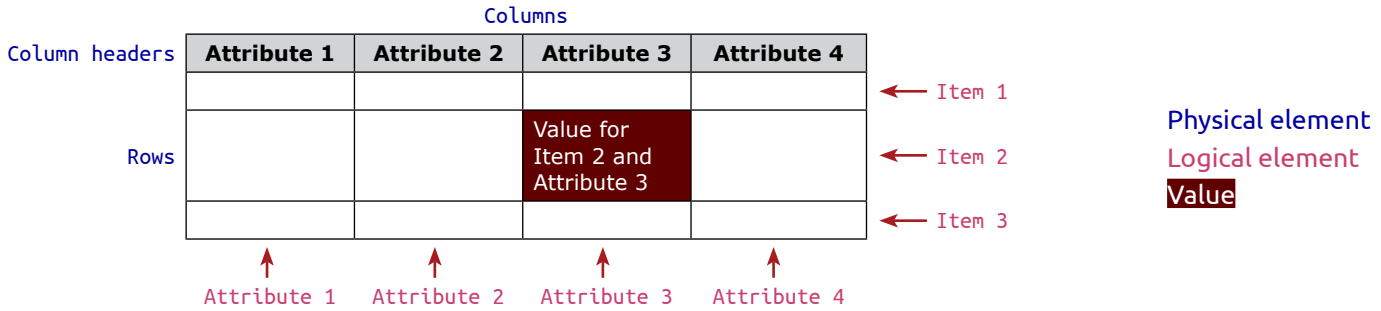


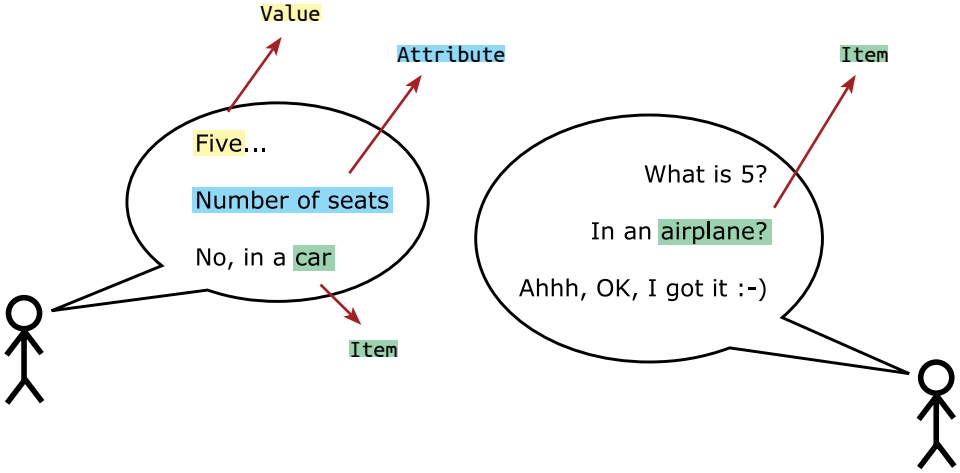
The DB objects are **System** and **User-Defined**:
System – created and used by the DB for internal purposes, a.k.a. **Object Catalog**. For example we can query system objects to list all table names in a DB
User-Defined – created and used by the DB professionals. These are the objects, where we store our data and code

Schemas
Namespace to store separately logical sets of objects related to the data for Sales, People, Human Resources, etc. The Schemas facilitate the management of the DB security. The default schema is "dbo" (DB object).

Tables
The tables are composed of:

- Rows (records) – Items
- Columns – Attributes (quality or characteristic belonging to the data) of the items
- Cells – Joins an attribute to an item
- Values in cells – The value of the data stored in a cell





Vechile	NumberOfSeats
Airplane	312
Car	5

How the DBE is Structured

```
Item
└→ Attribute
    └→ Value

Example
Car                     Item
└→ Color                Attribute
    ├→ Black            Value
    ├→ Blue             Value
    └→ Red              Value
   Number of seats      Attribute
    ├→ 4                Value
    └→ 5                Value
   Type                 Attribute
    ├→ Sedan            Value
    └→ Convertible      Value

Airplane                Item
└→ Color                Attribute
    ├→ White            Value
    └→ Pink             Value
   Number of seats      Attribute
    ├→ 4                Value
    └→ 223              Value
   Type                 Attribute
    ├→ Cargo            Value
    └→ Passenger        Value
```

Item – Attribute – Value is a hierarchical structure. Every value belongs to a attribute and every attribute belongs to an item.

This little table will help us put the pieces together:

The table name is **Customers**. The names of the DB objects are called **identifiers**. In our SQL code, we use the names to identify which objects we manipulate.

Customers

FirstName	LastName	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	*NULL*	*NULL*
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org
Melanie	Larson	melanie.larson@customer4.biz
Xavier	Jameson	xavier.garcía@customer6.com
Zak	Smith	*NULL*

The headers identify the names of the attributes (the columns).

In the rows we store the items (the customers). One row in the table represents one customer.

In the cells, we store the values for the attributes belonging to the items.

The attributes that we store for each customer, are `FirstName`, `LastName` and `Email`.

The table contains:

- 8 rows for 8 items (customers)
- 3 columns for the attributes `FirstName`, `LastName` and `Email`.

The value for the attribute `Email` and Item in row 4 is **john.smith@customer1.com**.
The value for the attribute `LastName`, which belongs to item **Melanie**, is **Larson**.

The View (**VW**):

- Is a virtual recordset that selects data from tables or another views
- Is defined by SQL code and doesn't store the data that it produces
- Directly selects and transforms data from underlying tables and views. Transformation or direct select
- Restricts sensitive data to specified roles and users (security)

Views on page 220

We can create a view to select all the columns from the **`HumanResources`** table, excluding the **`Salary`** column and make it visible to everyone.
Another view is the column **`Salary`** which is visible only to the HR director.

The function (**FN**):

- Is SQL code that performs specific task
- Is stored in the DBE, centralized and reusable
- Is called with or without input parameters
- Is **scalar-valued** when it returns a single value or **table-valued** when it returns a recordset

When the business logic needs to change, we edit the logic in the function only. Objects dependent on the function don't need to be edited.

We can think of the function as a workshop in a factory.
Every workshop is doing a specific task.
In a guitar factory, there are multiple workshops to cut the bodies and necks; paint; and assemble the guitars and more.
If we call the function **`PaintBody`** with a parameter called **`Color`** with value **Blue**, the body of the guitar is painted in blue.
Otherwise it will be painted with the default color defined in the workshop (call the FN without specifying value for the parameter).

An example of not parameterized scalar FN is: **Select the last day of the current month**. The result that the FN returns is based on today's date.

The parameterized scalar FN **`udf_UnitConvertor`** accepts the parameters **`Type`**, **`FromMetric`**, **`ToMetric`** and **`Value`** and returns the value in the desired metric.

Input				Output
Type	FromMetric	ToMetric	Value	Returned Value
Length	Inch	Centimeter	1	2.54
Volume	Liter	US Gallon	10	2.64172

The non-parameterized table-valued function **`udf_CustomersEligibleForDiscount`** returns a list of the customers, who have sales of $1,000 or more in the last fiscal month.

udf_CustomersEligibleForDiscount

CustomerID	Sales
5174	2356.15
7652	1005.14

Functions on page 227

The Stored Procedure (**SP**):

- Is complex SQL code that executes multiple consecutive steps
- Is stored in the DBE
- Can be executed input parameters
- Returns a result (output parameter or recordset)
- Is the most powerful SQL code execution object

We can now compare a SP to a warehouse, factory and related business processes together:

1. We store the wood (warehouse) for the bodies of the guitars
2. We create guitars (production)
3. We store and ship (distribution) the guitars

All these steps in the process are dependent on each other and are executed in order.

An example of a SP that executes an ETL:

- If it does not exist, create (temporary) tables for the ETL process
- Select the sales data from the last month into the tables
- Aggregate the data and add additional columns to the tables
- Load the data by weeks into destination tables
- Delete/truncate the (temp) tables at the end of the ETL process
- Manipulate the log created during the ETL process

Stored Procedures on page 240

RDBMS

Data in the DB refers to each other. We can't create a sales order for a customer that doesn't exist. **Sales** and **Customers** data are related. This is why databases like SQL Server are called **RDBMS** (Relational database management system).

Structure (Dim) and Data (Fact) Tables

The tables store different attributes of data. Based on the stored data, the tables are logically defined as:

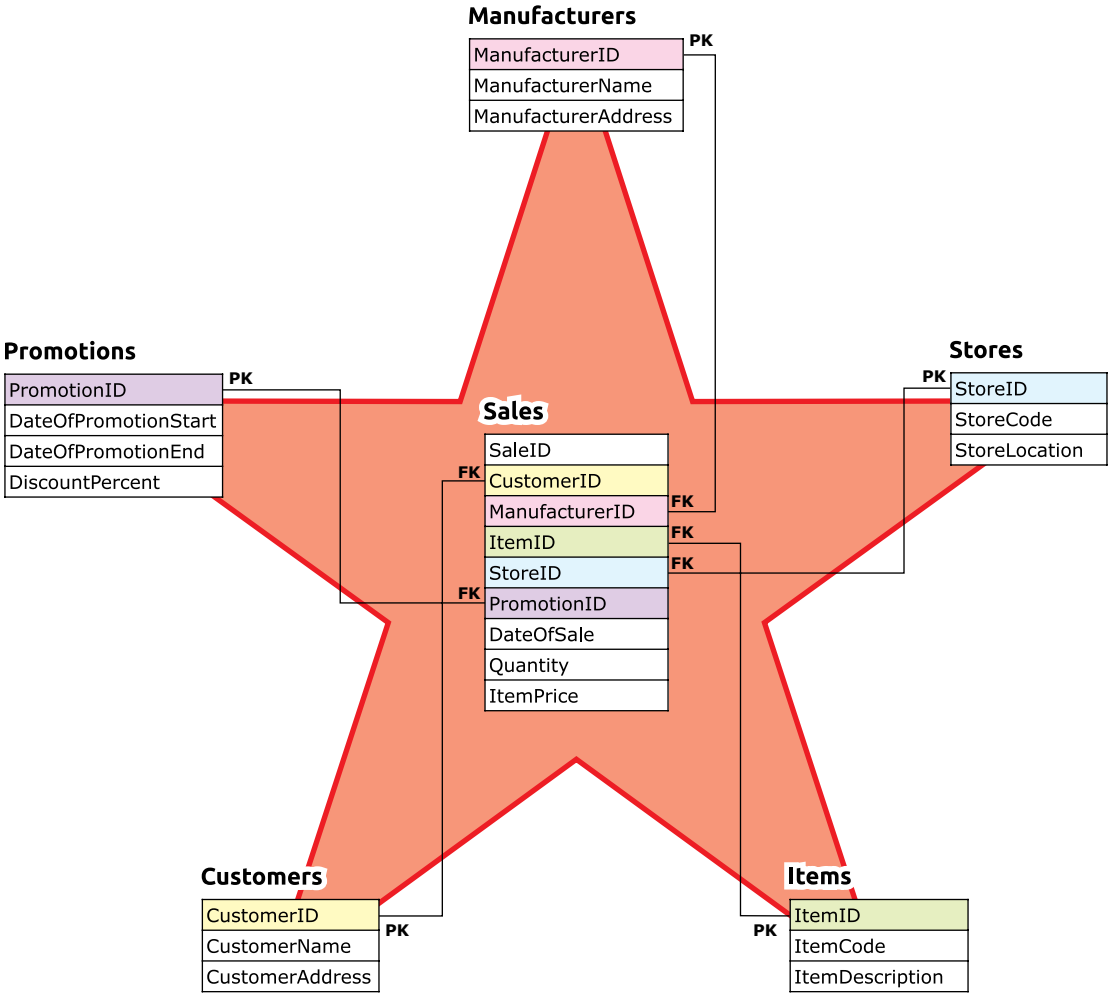
- **Structure** (**Dim** or **Dimension** in OLAP) tables – stores the structure of data like customers, items and manufacturers
- **Data** (**Fact** in OLAP) tables – stores the values (**Metrics**, **Measures**), related to the **structure** tables such as quantity and price of purchased item, made by manufacturer and total value of sales for a customer, etc

The structure and data tables are organized into logical schema. We can define two main schemas:

- **Star** - Represents a **data** table in the middle and the **structure** tables on the rays of the star
- **Snowflake** - an extension of the Star schema. Represents a **data** table in the middle and **structure** table in the rays and related **structure** tables on the sub rays

Star Schema

Built by a **data** table, linked to one or multiple **structure** table(s).



The **`Customers`** table stores the structure of the customers. One row is one customer. The column **`CustomerID`** identifies the row and can store only unique numeric values.
This column is called **Primary Key (PK)**. In table **`Customers`** we store all the attributes of the customer like names, addresses, phone numbers, emails, etc. The same logic applies to all the **structure** tables.

The **data** table combines the **structure** tables with the values (metrics) that the table represents. In the **`Sales`** table, these values are **`DateOfSale`**, **`Quantity`** and **`ItemPrice`**.

The column **`SaleID`** is the **PK**. Columns **`CustomerID`**, **`ItemID`** and **`ManufacturerID`** are linked to the **structure** tables (**Customers**, **Items** and **Manufacturers**), they are not unique (they may store duplicates) because one customer can order the same item by the same manufacturer multiple times. These columns are called **Foreign Key (FK)**.

There can only be one **PK** column per table and its type is:

- Natural (real data) - created by attribute(s) in the table
  We can use the column `Email` (unique values) as **natural PK**
- Surrogate (not meaningful data) - consecutive number (columns `CustomerID`, `ManufactiurerID`, etc)

The tables that make up the Star schema may look like this:

Customers (structure)

CustomerID	CustomerFirstName	CustomerLastName	CustomerAddress
1	John	Smith	48 Mountain Str.
2	Adrian	Jameson	9423 Highway Road
3	Michael	Olson	8 av. Riverside

1

a.k.a. **reference value** or **look-up value**

a.k.a. **key value**

Manucaturers (structure)

ManufacturerID	ManufacturerName	ManufactureRAddress
1	Toys For Boys Ltd.	164 Blue Str.
2	Little Princess Fun Inc.	1534 av. Lake Shore
3	Toys and Nature Co.	74 Green Forest Blvd.

1

Items (structure)

ItemID	ItemCode	ItemDescription
1	P123	Plastic Airplane Blue
2	F12S	Rag Doll 33 cm.
3	1234	Wooden Horse (Age: 3-6 years)

1

Sales (data)

∞ ∞ ∞

SaleID	CustomerID	ManufacturerID	ItemID	StoreID	PromotionID	DateOfSale	Quantity	ItemPrice
1	1	3	2	1	*NULL*	1967-04-15	4	9.3582
2	3	3	2	1	3	1964-12-13	12	1.5237
3	3	2	1	2	*NULL*	1965-03-17	5	22.3452
4	1	1	3	3	*NULL*	1966-01-27	75	0.8374
5	2	2	1	2	1	1966-04-05	6	45.5384

1 ∞

∞ 1

Stores (structure)

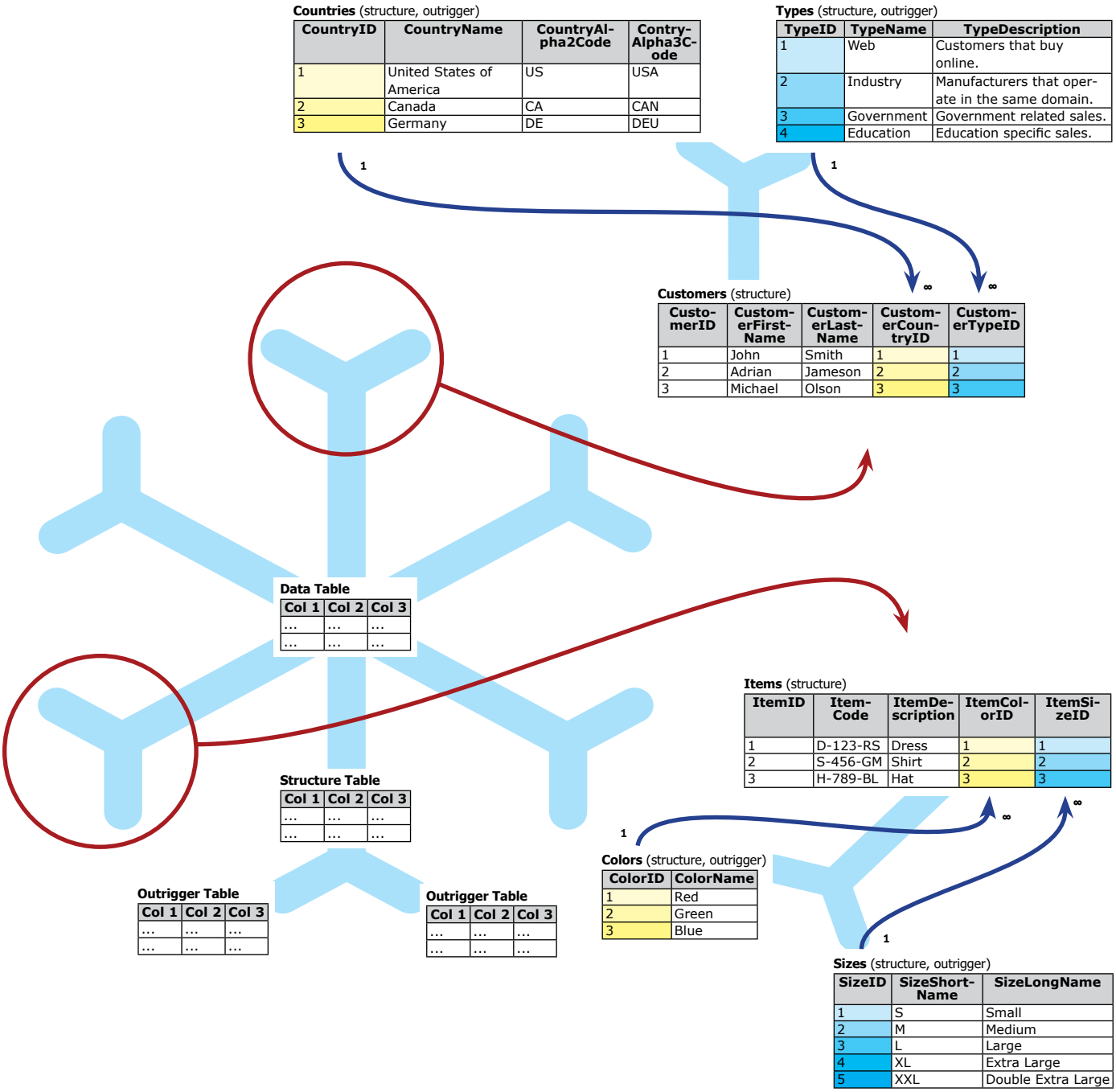
StoreID	StoreName	StoreAddress
1	Magic Toys	15 1st Avenue
2	Happy Kids	174 Rockland Str.
3	My Childhood	90 Blueway Blvd. #303

Promotions (structure)

PromotionID	PromotionDateStart	PromotionDateEnd	PromotionPercent
1	1966-03-08	1966-04-08	0.21
2	1963-07-12	1963-09-12	0.07
3	1964-12-10	1964-12-17	0.35

Snowflake Schema

An extension of the Star Schema. The **structure** table refers to other **structure** tables, called **outriggers**.



Entity

Entity is a logical group of tables that store related data on one subject - Time, Items, Manufacturers, Vendors and Customers. The entities give us the option to extend the flexibility of the data. We can store multiple variants of one item in separate table.

If we manufacture or sell clothes, one item is the **dress** and the multiple variants are **size**, **color**, **fabric** and other.

We can store multiple phones for one customer in **`CustomersPhones`** table. The uniqueness (one row represents one customer) of table **`Customers`** is kept, and the requirement to store multiple phones for one customer is met.

Entity	**Tables in the entity**
Time	Years, Quarters, Months, Weeks, Days
Product	Items, Variants, Colors, Sizes, UnitOfMeasures
Producers	Manufacturers, Industries, Geography
Suppliers	Vendors, Categories, Classes, Geography
Clients	Customers, Addresses, Phones, Geography
Geography	Countries, Cities, PostalCodees

The entities above are used in all subject areas (Purchases , Sales , Marketing Campaigns) of the business.

Sales

Time

Days

Day	Day-Of-Week	Day-OfMonth	Day-Of-Year	Quar-terID
1969-03-30	1	30	89	1969-01
1969-03-31	2	31	90	1969-01
1969-04-01	3	1	91	1969-02

Quarters

QuarterID	DaysIn-Quarter	WeeksIn-Quarter
1969-01	90	14
1969-02	91	14
1969-03	92	14

Entity

Product

Items

ItemID	Item-Code	ItemDescription	Col-orID
1	P123	Plastic Airplane Blue	1
2	F125	Rag Doll 33 cm.	2
3	1234	Wooden Horse (Age: 3-6 years)	1

Colors

Color-ID	Color-Name
1	Red
2	Green
3	Blue

Entity

Suppliers

Vendors

Ven-dorID	Ven-dorCode	VndorName	Vendor-ClassID
1	VC12	Toys Wholesales	3
2	VA56	Toys and Games	1
3	CU05	Infinite Toys	3

Classes

Clas-sID	ClassName
1	Critical
2	Digital
3	Tactical

Entity

Sales

SaleID	Ven-dorID	ItemID	DateOf-Sale	Quan-tity	Item-Price
1	1	2	1969-03-31	536	3.14
2	3	3	1969-03-30	78	22.75
3	2	1	1969-04-01	13	673.24

Entity

The presented structures (star, snowflake and entity) are usually built by a DB architect, so as mentioned, the DB professional's roles can overlap and a DBA or developer can also be involved.

Relations

The tables are linked by the following relations:

- **One-to-One** - one row corresponds to one row in another table. Extends the number of the attributes, by spliting them in two tables

Customers

CustomerID	First-Name	Last-Name	Email
1	John	Smith	js@customer1.net
2	Anna	Laurier	lauriera@customer2.org
3	Kimberly	Nelson	kim_nel@customer3.com

CustomersMarketingDetails

CustomerID	MarketingEmails	MarketingCalls
1	1	1
2	*NULL*	1
3	0	*NULL*

1 1

- **One-to-Many** - one row corresponds to many rows in another table. One customer may have multiple phone numbers

Customers

CustomerID	First-Name	Last-Name	Email
1	John	Smith	js@customer1.net
2	Anna	Laurier	lauriera@customer2.org
3	Kimberly	Nelson	kim_nel@customer3.com

CustomersPhones

CustomerID	Phone	Type
1	+1 555 2222	Home
1	+1 555 2223	Mobile
1	+1 555 2224	Business
2	+1 555 1111	Home
2	+1 555 1112	Mobile

1 ∞

- **Many-to-Many** - many rows corresponds to many rows in another table. This relation is built by two One-to-Many relations. One customer can rent multiple cars and one car can be rented to multiple customers

Customers

CustomerID	First-Name	Last-Name	Email
1	John	Smith	js@customer1.net
2	Anna	Laurier	lauriera@customer2.org
3	Kimberly	Nelson	kim_nel@customer3.com

Rents

RentID	CustomerID	CarID
1	3	2
2	2	2
3	3	1
4	1	1
5	2	3

Cars

CarID	Make
1	Toyota
2	Ford
3	Mazda

1 ∞ ∞ 1

Primary Key – Foreign Key (PK - FK) Relations

To manage these relations, we create a **PRIMARY KEY (PK)** and a **FOREIGN KEY (FK)** constraints on the columns that define the relations. We are flagged to insert a **FK** when a corresponding **PK** doesn't exist.

PRIMARY KEY is created on:

- One column with unique values that identify the rows in the table. One of the most commonly used PKs is an auto incremental identity column (surrogate PK)
- Multiple columns - composite (or compound) PK. The combination of the columns in the key is unique

FOREIGN KEY is a column in the **data** table that is linked to the **PK** in the **structure** table. The values in the FK column can be duplicated.

Customers

CustomerID	First-Name	Last-Name	Email
1	John	Smith	john.smith@customer1.com
2	Anna	Laurier	anna.laurier@customer2.net
3	Melanie	Larson	melanie.larson@customer4.biz

PRIMARY KEY

Items

ItemID	Item-Code	ItemDescription
1	P123	Plastic Airplane Blue
2	F12S	Rag Doll 33 cm.
3	1234	Wooden Horse (Age: 3-6 years)

PRIMARY KEY

1 ∞ 1 ∞

Sales

SaleID	CustomerID	ItemID	DateOfSale	Quantity	Price
1	2	3	1967-12-14	73	0.2563
2	1	3	1966-05-17	12	5.2341
3	1	2	1968-10-27	52	62.3251

PRIMARY KEY FOREIGN KEY FOREIGN KEY

Logical and physical PK – FK relations:

- Logical relations – constraints are not created on the columns. Managed by a logic in the SQL code or outside the DB (another layer of the software architecture – web or standalone application)
- Physical – PK and FK are real constraints in the DB and they are respected when manipulating the data

When a **physical PK-FK** relation is created, **we can't insert FK with no corresponding PK**. That means that we need a customer to exists in the **Customers** table (**PK**) to sell to this customer (**FK** in **Sales** table).

Relation	Keys
One-to-One	PK to PK
One-to-Many	PK to FK
Many-to-Many	PK to FK and FK to PK (junction table)

Add **PK** to all the tables as:

- Surrogate - Integer identity column *or*
- Natural - One column or composite PK

Naming Conventions

The DB objects, the identifiers and the variables need to have names. The goal is to build a easy to understand DB structure and programming code. To construct the names, we follow rules called **Naming Conventions**. The rules are defined by the DB professionals and are not mandatory. We can group the naming conventions as:

Descriptive Names

- Table names explain the data that the table stores:
  `Customers` `SalesHeader` `SalesDetails` `Items`
- What is the task performed by the function:
  `usf_CalculateRebateforCustomer`
  `udf_IsFirstDayofFiscalMonth`
  `utf_SalesForCustomerIDByPeriod`
- The name of the Stored Procedure explains what the procedure executes:
  `usp__ProjectID_1234__ETL` `uspDailyFullBackup`

Special characters, start with numbers and keywords

When the name:

- Contains a special character (every character that is not a letter or number) `[This is My First Table!]`
- Starts with a number `[123ThisisMyFirstTable]`
- Is T-SQL keyword `SELECT`, `FROM`, `TABLE`, `CREATE`, `AND` etc

we make it valid by surrounding it with square brackets.

Examples: `[12 Months Sales $]` `[Customers-Stats]` `[Split String]`

Case

The different options of combining the upper and lower case define:

- PascalCase – every word starts with upper case `MyFirstTable`
- camelCase – first word starts with lower case and every next word starts with upper case `myFirstTable`
- All UPPER, all lower case `MYFIRSTTABLE`, `myfirsttable`

Delimited

We can use a character to delimit parts of the name:

- An underscore (_) can replace the spaces:

Before	After
`[My First Table]`	`My_First_Table`

- Two underscores (__) can be a splitter between the prefix, the name and the suffix:

Before	After
`uspSalesData_Archive`	`usp__Sales_Data__Archive`

Prefix

We can add a prefix to include the object type in the name:

`tbl_MyFirstTable` ← table `vw_MyFirstView` ← view

`udf_MyFirstFunction` ← function `usp_MyFirstStoredProcedure` ← stored procedure

The prefix gives us the flexibility to give the same name to different types of objects and facilitates the DB professionals by giving them a hint of what type of object they are working with.

We can add custom clarification in the prefix to identify the action that the object performs or to which project the object is assigned:

- `usp_Monthly_ETL__ProjectID_1234` is the name of a stored procedure, assigned to ProjectID 1234, that runs an ETL once per month.
- `usp_pid1234__Monthly_ETL` and `ups_pid1234__ReportByClient` are assigned to the same projectID 1234.

We can use some of the suggestions below or define our custom prefixes.

Prefix	Object Type	Comment
`t, t_,` `tbl, tbl_`	Table	t, tbl - table
`v, v_,` `vw, vw_`	View	v, vw - view

`fn, fn_,`
`ufn, ufn_,`
`udf, udf_,`
`usf, usf_`
`utf, utf_`
	Function	fn - function
		ufn - user function
		udf - user-defined function
		usf - user-defined scalar function
		utf - user-defined table-valued funcion

`sp`
`usp_`
~~`sp_`~~
	Stored Procedure	sp - stored procedure
		usp - user-defined stored procedure
		sp_ is a system prefix and using it as user-defined prefix is not a good practice

Suffix

- The suffix can be used to give additional description or clarity:
- The variation of the data in table or view:

  `tbl__Sales_Current` `tbl__Sales_LastYear` `tbl__Sales_Archive`
- The action, performed by a function or SP

  `usp_SalesCurrentMonth_ETL` `usp__Sales_Current_Month__Report`

Prefix, Suffix and delimiter

By adding both prefix and suffix, we benefit from all the advantages that they offer.

When we delimit the name with underscores, we can delimit the name from the prefix and the suffix with two delimiters (__) to distinct the delimiter between:

- Prefix and suffix
- Name and the delimiter inside the name

By adding:

- Delimiter in the name — underscore (_)
- Delimiter after prefix and before suffix — two underscores (__)

we split the elements clearly and make the name much more intuitive `Prefix__This_Is_Object_Name__Suffixfx`

Order of the Words

`DateStart` or `StartDate`	`FromDate` or `DateFrom`	`InvoiceDate` or `DateInvoiced`
`DateEnd` or `EndDate`	`ToDate` or `DateTo`	`DateModified` or `MofifyDate`

To facilitate the read and use of the (object, column, variable) name, we start with the common word of a group:

Group	**Name**
Date	`DateInvoiced`, `DateOrdered`, `@DateStart`, `@DateEnd`
Name	`NameFirst`, `NameLast`, `NameDisplayed`
Status	`StatusOrder`, `StatusRejection`, `StatusActive` etc

By combining the rules, we may end with names like:

Rule	Variant	Name
Special Characters	Space	`[My First Table]`
	Exclamation mark	`[MyFirstTable!]`
	Space and quote sign	`[John's Column in His First Table]`
	Space and dash	`[Sales – 1967]`
	Start with number	`[123 Store Inc. Sales]`
	Keyword	`[Table]`
Case	camelCase	`myFirstTable`
	PascalCase	`MyFirstTable`
	All lower	`myfirsttable`
	ALL UPPER	`MYFIRSTTABLE`
Delimited	Underscore and camelCase	`my_first_table`
	Underscore and PascalCase	`My_First_Table`
	Underscore all lower	`my_first_table`
	Underscore all upper	`MY_FIRST_TABLE`
Prefix	Not delimited and camelCase	`tblMyFirstTable`
	Delimited and camelCase	`tbl_My_Table`
	Double delimiter and PascalCase	`Tbl__My_First_Table`

Suffix	camelCase	`tblMyFirstTableArchive`
	camelCase and delimiter	`tblMyFirstTable_Archive`
	PascalCase and double delimiter	`Tbl__My_First_Table__Archive`
Prefix and Suffix	Single delimiter	`tbl_MyFirstTable_Archive`
	Double and single delimiter	`usp_ETL__Project_1234__Daily`

SDSO, DSO and SO

When pointing to an object, we can use:

- Relative path - only **ObjectName** *or*
- Full path - start from a higher level and point to the object:
  ServerName ⟶ **D**atabaseName ⟶ **S**chemaName ⟶ **O**bjectName

As the object name is an identifier, **two objects** of the **same type** (table, view, function, stored procedure) and with the **same name** can't exist on one level. They can exist in different schemas.
Two schemas with the same name can't exist in one DB, but can exist in different DB.s
Two DBs with the same name can't exist on one DBE Instance, but can exist in different instances.

To point to the exact DB object, we use the full path to the object.

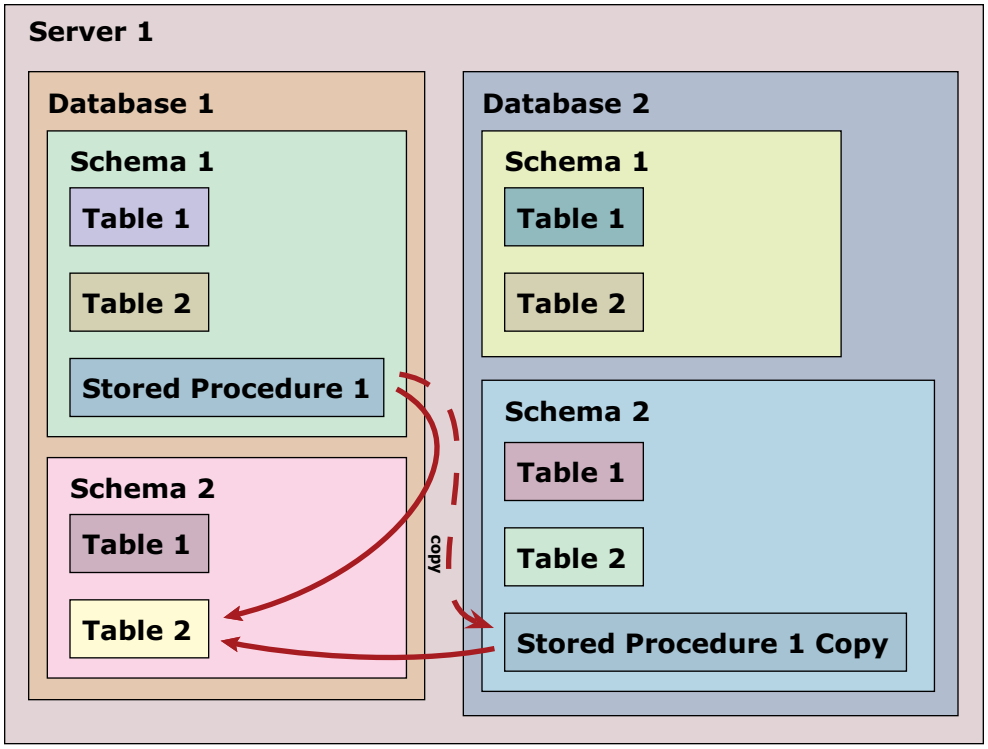
	Level 1	Level 2	Level 3	Level 4
SDSO	`ServerName.`	`DatabaseName.`	`SchemaName.`	`ObjectName`
DSO		`DatabaseName.`	`SchemaName.`	`ObjectName`
SO			`SchemaName.`	`ObjectName`

SDSO Level	SDSO Syntax	FROM Clause Example

Object	[ObjectName]	`FROM [MyTable]`
		`FROM [vw_MyView]`
		`FROM [udf_MyFunction]`

Schema	[SchemaName].[ObjectName]	`FROM [dbo].[MyTable]`
Database	[DatabaseName].[SchemaName].[ObjectName]	`FROM [MyDatabase].[mySchema].[vw_MyView]`
Server	[ServerName].[DatabaseName].[SchemaName].[ObjectName]	`FROM [SQLServer1].[Client173].[dbo].[usp_ETL_Daily]`

Using the `DatabaseName.SchemaName.ObjectName` structure is making our code more robust and able to be executed in different databases and schemas on the same server.



When we use the **DSO structure**, we can copy `[Database 1].[Schema 1].[Stored Procedure 1]` that queries `[Database 1].[Schema 2].[Table 2]` to `[Database 2].[Schema 2].[Stored Procedure 1 Copy]` and the copied stored procedure will continue to query the same object - `[Database 1].[Schema 2].[Table 2].`

SQL and T-SQL (Transact-SQL)

SQL is an abbreviation of **Structured Query Language**. This is the programming language that communicates with the DB.

SQL is **Structured** because it is built by batches that hold statements which hold clauses in an organized structure.

Code, executed on one run →	One step of the code →	Command that builds the step	T-SQL keyword
Batch	Statement (**Select** query)	Clause	`SELECT`
		Clause	`FROM`
		Clause	`WHERE`
	Statement (**Modify** query)	Clause	`UPDATE`
		Clause	`SET`
		Clause	`WHERE`

s**Q**L runs **Queries** to retrieve data from the DB (**Select** queries). The select queries can be understood as **questions**. We ask the DB for specific data and it gives us back an answer (the data, organized in **recordset**). We may also ask (query) the DB to add, modify or delete existing data (**Modify** queries).

One set of queries creates, reads, updates and deletes **data**.

Manipulates what	T-SQL clause	Query type
Existing data	`SELECT`	**Select** queries
	`UPDATE`	**Modify** queries
	`DELETE`	
New data	`INSERT`	

Another set of queries creates, edits and deletes DB **objects**.

Manipulates what	T-SQL clause	Query type
New object	`CREATE`	**Create** queries
Existing object	`ALTER`	**Modify** queries
	`TRUNCATE`	
	`DROP`	

s**QL** is **Language**, because it is a programming language.

SQL is standardized by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). The standards are revised continuously to cover new technologies and improve the functionality of the databases. The revisions have names like SQL-92, SQL:2003 or SQL:2008.

As different RDBMSs are created by different companies, the SQL has specific extensions that serve the specific software. The extended SQL respects the standards defined by ANSI and ISO (ANSI SQL). The extended SQL in Microsoft SQL Server© is called **T-SQL** or **Transact-SQL**.

SQL and T-SQL (Transact-SQL)

Some of the most popular database software and the extended SQL:

Vendor	DB Software	Extended SQL
Microsoft Corporation	SQL Server	T-SQL (Transact SQL)
Oracle Corporation	Oracle	PL/SQL (Procedural Language/SQL)
IBM	DB2	SQL PL (SQL Procedural Language)
SAP	Sybase	Watcom-SQL
MySQL AB	MySQL	SQL/PSM (SQL/Persistent Stored Module)

This book is dedicated to Microsoft SQL Server and T-SQL.

Code Elements

The structure of T-SQL code is hierarchical and contains different elements:

1. Batch
2. Statement (a.k.a. Query, Command)
3. Clause

```
SELECT
  FirstName
  , LastName
FROM Customers
WHERE FirstName = 'Beverly';

UPDATE Customers
SET Email = 'b@BeverlyWebsite.com'
WHERE
  FirstName = 'Beverly'
  AND LastName IS NULL;
GO
```

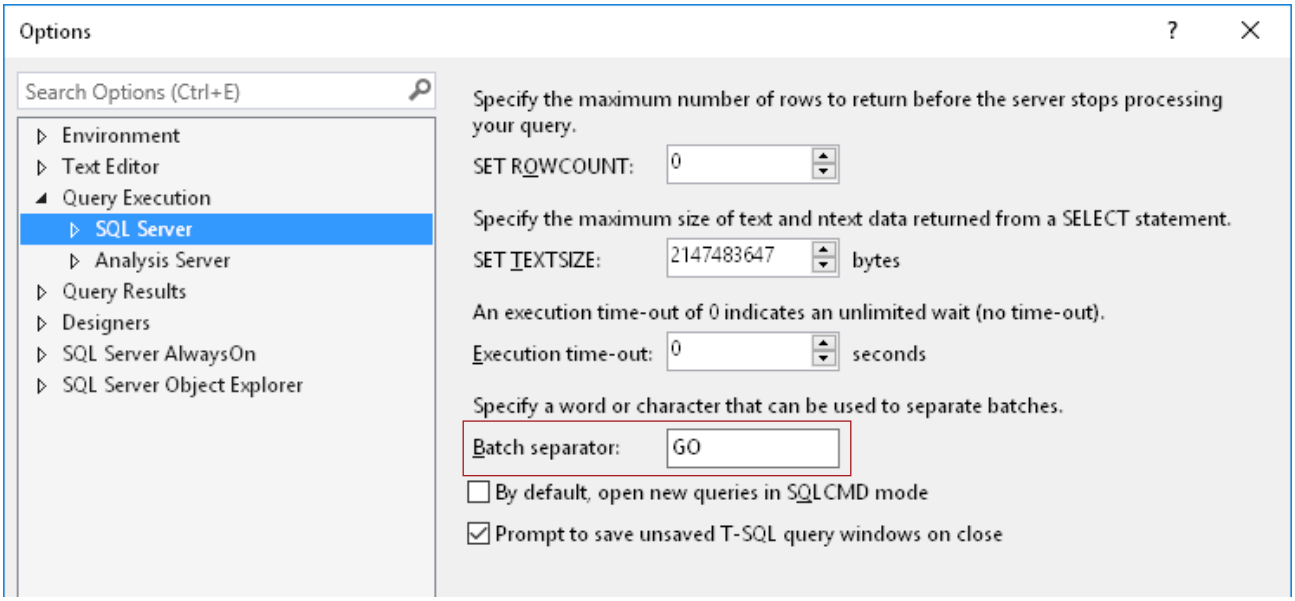
Batch

Statement

Clause

Batch

SQL code, grouping one or multiple statements (commands) and executing on one run.
In SSMS we use the keyword `GO` to separate the batches, customizable in **Tools** ⟶ **Options** ⟶ **Query Execution** ⟶ **SQL Server** ⟶ **General**:



`GO` is not a T-SQL command. It is not relevant outside SSMS.

The `GO` batch separator sends the batch to the DBE for execution.
`GO 50` will send the batch 50 times to the DBE (will be executed 50 times).

Statement

SQL Code that executes a command to:

- Create, alter or delete DB object (Data Definition Language (DDL))
- Manipulate data (Data Manipulation Language (DML)

The statements in a batch are executed consecutively and synchronously - the execution of a second statement will start after the execution of the first statement executed has finished.

Execution Step	Statement 1		Statement 2		Statement 3	
Time	Start	End	Start	End	Start	End
	10:00:02 AM	10:02:14 AM	10:02:14 AM	10:05:56 AM	10:05:56 AM	10:06:06 AM

Every statement starts with a keyword (`CREATE`, `ALTER`, `DROP`, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc.) and terminates with a semicolon (`;`).

Clause

Component that constructs the statement, structured in exact order. There can be one or multiple in a statement and define the type of the statement (DDL or DML).

Keyword

Words, existing in the English language, used in T-SQL to construct the clauses and statements. They are defined by ANSI and ISO and modified in T-SQL to extend the functionality of the SQL Server. Keywords are reserved in T-SQL, so it is not good practice to use them as identifiers. When we use a keyword as an identifier, we surround it in square brackets ([]).

Keywords are not case sensitive. `CREATE TABLE` and `create table` will be executed the same way, but we write the keywords in upper case to make them stand out to distinguish them from the identifiers (names of DB objects, column names in table etc.) and the values.

They are self explanatory and:

- `USE` selects the DB to be used
- `CREATE` creates a new object
- `ALTER` alters the structure of an object
- `DROP` drops existing object

- SELECT selects data from an object
- FROM points to the object that we query from, etc

The keywords constitute the T-SQL grammar.

Identifier

```
SELECT
  FirstName          <------ Identifier (column name)
  , LastName
FROM Customers       <------ Identifier (table name)
WHERE FirstName = 'Beverly';

UPDATE Customers
SET Email = 'b@BeverlyWebsite.com'
WHERE
  FirstName = 'Beverly'
  AND LastName IS NULL;
GO
```

The names of:
- DB
- DB objects
- Table or view columns

are their identifiers in SQL.

They are:
- Required (created by the DB architect) *or*
- Optional (automatically generated by SQL Server)

Identifiers are collected in the Object Catalog and can be queried to perform a task such as to **bulk update some value in all string columns in multiple tables**.

The case sensitivity of the identifiers is defined by the SQL Server Collation.

Collation on page 66

Data Type

When we create or alter DB object, we specify the data type that table column, variable, function parameter or stored procedure parameter can store. One table column can store strings, whilst another can store numbers, dates and times or booleans.

```
CREATE TABLE Customers
  (
    FirstName NVARCHAR(32)
  , LastName NVARCHAR(64)
  , Email VARCHAR(128)
  );
GO
```

```
ALTER TABLE Customers
ADD Phone VARCHAR(16);
GO
```

Data Type

```
DECLARE @MyVariable INT;
GO
```

CREATE and ALTER DDL statements specify the data type that table column can store or the data type of SP or FN parameters. When we create (DECLARE) a variable, we also specify the data type that the variable can store.

```
SELECT ('MyString' + 3) AS Expression;
GO
```

String

Integer

The data type of the elements has to be the same in order the expression to be evaluated correctly.

```
Conversion failed when converting the varchar value
'MyString' to data type int.
```

Expression

Data Types and Conversions on page 56

Combined values and operators that evaluates to a single value.

```
SELECT ('John' + ' ' + 'Smith');
SELECT (TotalSales * 2);
```

Value

Operator

Expresion

```
Evaluates to John Smith
If TotalSales = 1234, evaluates to 2468 (1234 * 2)
```

The data type of the expression is the same as the data types of its components.

Operator

Keywords or character that perform comparison, arithmetic, logical or negate operation.

```
SELECT
  CustomerID
  , ItemID
  , (TotalSales * 1.3) AS ExpandedSales
FROM Sales
WHERE
  ItemID = 2
  AND DateOfSale IN ('1967-04-15', '1967-05-17')
  AND NOT IsCorrection = 0;
GO
```

Arithmetic

Comparison

Logical

Negate

Value

Value is the amount of the data.

- In expressions:
  - (1.3 + 17)
  - ('John' + ' ' + 'Smith')
- In operators: FirstName = 1.3
- In identifier or variable, transformed to value during the execution:
  - In expression that contains identifier: (SalesTotal * 1.3)
  - In expression that contains identifier and variable: (SalesTotal * @Multiplier)

The value, stored in the variable

The value, stored in the column

```
SELECT
  CustomerID
  , ItemID
  , (TotalSales * 1.3) AS ExpandedSales
FROM Sales
WHERE
  ItemID = 2
  AND DateOfSale IN ('1967-04-15', '1967-05-17')
  AND NOT IsCorrection = 0;
GO
```

Value

Alias

Name, defined in the code that replaces the name of identifier. Customizes and facilitates the usage of the identifiers.

Aliases on page 138

```
SELECT
  CONCAT(C.FirstName, ' ', C.LastName) AS CustomerName
  , I.ItemCode
  , (S.TotalSales * 1.3) AS [Expanded Sales]
FROM
  Sales AS S
  JOIN Customers AS C
   ON S.CustomerID = C.CustomerID
  JOIN Items AS I
   ON S.ItemID = I.ItemID
WHERE
  I.ItemID = 2
  AND S.DateOfSale IN ('1967-04-15', '1967-05-17')
  AND NOT S.IsCorrection = 0;
GO
```

Assign Alias (AS)

Bind Alias (.)

Square Bracket

We surround the identifier that holds a special characters (not letter or number) in square brackets. Else they are not valid in T-SQL and we can't execute the batch.

```
SELECT [First!Name] AS [First Name]
FROM [Approved?Customers];
GO
```

Special Character

Literal value

Literal is exact **constant** value.

```
SELECT
  CustomerID
  , ItemID
  , (TotalSales * 1.3) AS ExpandedSales
FROM Sales
WHERE
  ItemID = 2
  AND DateOfSale IN ('1967-04-15', '1967-05-17')
  AND NOT IsCorrection = 0;
GO
```

Literal decimal value

Literal string value

Literal integer value

In T-SQL the letter N before the left single quote of a string, defines the string as Unicode.

```
'This is My String'
N'これは私の文字列であります'
```

Character string

Unicode character string

Variable

The variable is place in the memory of the computer where we store value that can vary.

```
DECLARE
  @ItemID NVARCHAR(64) = 2
  , @DateStart DATE = '1967-04-15'
  , @DateEnd DATE = '1967-05-17';

SELECT
  CustomerID
  , ItemID
  , (TotalSales * 1.3) AS ExpandedSales
FROM Sales
WHERE
  ItemID = @ItemID
  AND DateOfSale IN (@DateStart, @DateEnd)
  AND NOT IsCorrection = 0;
```

Variable

Create variable and assign a value to it

The value of @DateEnd is '1967-05-17'

Use the variable

```
SET @ItemID = 3;

SELECT @ItemID AS Result;
GO
```

Change the value of the variable

Now the value of @ItemID is 3

Delimiter

The delimiter:

- Shows where one portion of the code ends
- List identifiers

To delimit	We use	Glyph
Identifiers	Comma	,
Statements	Semicolon	;
Batches	Keyword	GO

```
SELECT
  FirstName
, LastName
, Email
FROM Customers
WHERE FirstName = 'Beverly';

UPDATE Customers
SET Email = 'b@BeverlyWebsite.com'
WHERE
  FirstName = 'Beverly'
  AND LastName IS NULL;
GO
```

Identifier delimiter

Statement delimiter

Batch delimiter

Syntax

Every programming language is following rules, called syntax. The above rules define the SQL syntax.

DDL and DML Statements

SQL statements work with:

- DB objects
- Data

The statements that create, alter or delete DB objects are grouped as **Data Definition Language (DDL)**. The statements that query or manipulate the data are called **Data Manipulation language (DML)**.

DDL

We can recognize the DDL Statements by their first clause - `CREATE`, `ALTER`, `TRUNCATE`, `DROP`.

Clause	Apply On	Description
`CREATE`	Database, schema, table, view, function, stored procedure	Create new DB or DB object
`ALTER`	Database, schema, table, view, function, stored procedure	Edit existing DB or DB object
`TRUNCATE`	Table	Delete all the data in a table and resets the identity column
`DROP`	Database, schema, table, view, function, stored procedure	Delete existing DB or DB object

DML

The DML statements manipulate the data in the DB and their first clause are `INSERT`, `SELECT`, `UPDATE`, `DELETE`

Clause	Apply On	Description
`INSERT`	Table, view	Insert new row
`SELECT`	Table, view, table-valued function	Select existing data
`UPDATE`	Table, view	Update existing data
`DELETE`	Table, view	Delete existing data

CRUD is an abbreviation of the four basic actions that we perform in the DB - **Create**, **Read**, **Update** and **Delete**. We apply CRUD actions on DDL and DML as follows:

	DDL		DML	
Abbreviation	**Clause**	**Action**	**Clause**	**Action**
Create	`CREATE`, `TRUNCATE`	Create object	`INSERT`	Create row in a table
Read			`SELECT`	Select data from a table
Update	`ALTER`	Change the definition of an object	`UPDATE`	Update data in a table
Delete	`DROP`, `TRUNCATE`	Delete object	`DELETE`	Delete row from a table

The DDL `TRUNCATE` statement is logically a DML statement, because it deletes all the data in a table, but not the table itself. It is in the DDL group, because it changes the object (resets the identity column). It is a variation of `DROP` and `CREATE` statements together.

Data Types and Conversions

The values of:

- Columns in the DB tables
- Variables
- Parameters in functions or stored procedures
- Elements of the expressions

can be a specified data type.

Table columns

Customers

CustomerID	FirstName	LastName	Email	YTDSales	DateRegistered	TimeRegistered
1	John	Smith	jsmith@js1.org	523.43	1967-03-25	17:36:55.2030000
2	Anna	Larson	larsona@al2.com	8542.12	1968-12-14	03:21:28.7470000
3	Michelle	Boyer	mb@customer3.net	85.59	1965-07-18	13:47:00.8130000

Column	Data type	Can store only
CustomerID	INT	Whole numbers (interegs)
FirstName	NVARCHAR(32)	Strings of maximum 32, characters
LastName	NVARCHAR(64)	Strings of maximum 64 characters
Email	VARCHAR(128)	Strings of maximum 128 characters
YTDSales	MONEY	Numbers (money)
DateRegistered	DATE	Dates (year, month, day)
TimeRegistered	TIME	Time (hour, minute, second, millisecond)

Variable

```
DECLARE @MyIntVariable INT;
SET @MyIntVariable = 123;
SET @MyIntVariable = 'ABC';
```

@MyIntVariable can store only whole numbers

We can't assign the string value 'ABC'

Conversion failed when converting the varchar value 'ABC' to data type int.

Parameter in Function or Stored Procedure

```
CREATE FUNCTION udf_MyFunction
  (
    @MyFunctionParameter1 INT
    , @MyFunctionParameter2 DATETIME2
  )...
```

Accepts only whole numbers

Accepts only date and time

Expression

```
SELECT 'My String ' + CAST(3 AS CHAR(1)) AS MyExpression;
```

String

Integer, converted to string

MyExpression
My String 3

The data types in SQL Server are grouped as follows:

Numerics
- **Exact**
  - Integer - TINYINT, SMALLINT, INT, BIGINT
  - Fraction - NUMERIC, DECIMAL (DEC)
  - Monetary, currency - MONEY, SMALLMONEY
  - Boolean - BIT
- **Approximate**
  - Floating point - FLOAT, REAL

Strings
- **Character Non-Unicode**
  - CHAR
  - VARCHAR
  - ~~TEXT~~ - **Obsolete**. Replaced with VARCHAR(MAX)
- **Character Unicode**
  - NCHAR
  - NVARCHAR
  - ~~NTEXT~~ - **Obsolete**. Replaced with NVARCHAR(-MAX)

Binary
- BINARY
- VARBINARY
- ~~IMAGE~~ - **Obsolete**. Replaced with BINARY(MAX)

Date and time
- SMALLDATETIME, DATETIME, DATETIME2
- DATE
- TIME
- DATETIMEOFFSET

Other
- UNIQUEIDENTIFIER
- ~~TIMESTAMP~~ - **Obsolete**. Replaced with ROWVERSION.
- XML
- SQL_VARIANT
- TABLE
- HIYERERARCHYID
- CURSOR
- **Spacial**
  - GEOGRAPHY
  - GEOMETRY

Numerics - Exact

Integer (whole number)

Name	Storage Size (bytes)	Range (From)		Range (To)	
TINYINT	1		0		255
SMALLINT	2	-2^{15}	-32,768	$2^{15} - 1$	32,767
INT	4	-2^{31}	-2,147,483,648	$2^{31} - 1$	2,147,483,647
BIGINT	8	-2^{63}	-9,223,372,036,854,775,808	$2^{63} - 1$	9,223,372,036,854,775,807

Fraction

Name	Precision: Storage Size (bytes)	Range (From)	Range (To)
DECIMAL(P, S) DEC	1 to 9: 5 10 to 19: 9 20 to 28: 13 29 to 38: 17	$-10^{38} + 1$	$10^{38} - 1$
NUMERIC(P, S)			

P - Precision: Total digits (including the decimal point) - 1 to 38; Default is 18
S - Scale: Digits after the decimal point - 0 to 37; Default is 0

The Position of the digits and the decimal point

Millions	Hundred Thousands	Tens Thousands	Thousands	Hundreds	Tens	Ones		Tens	Hundreds	Thousands	Tens Thousands	Hundred Thousands	Millions	Tens Millions
1,000,000	100,000	10,000	1,000	100	10	1		10	100	1,000	10,000	100,000	1,000,000	10,000,000
7	**6**	**5**	**4**	**3**	**2**	**1**	**.**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Whole Number							Decimal Point	Decimal Fraction						

Position

Monetary, Currency

Name	Storage Size (bytes)	Range (From)	Range (To)
MONEY	8	-922,337,203,685,477.5808	922,337,203,685,477.5807
SMALLMONEY	4	-214,748.3648	214,748.3647

Boolean

Name	BIT columns per table: Storage Size (bytes)	Range (From)	Range (To)
BIT	1 to 8: 1 9 to 16: 2 17 to 24: 3 25 to 32: 4 ...	0 False No	1 True Yes

Numerics - Approximate

Floating Point Number

Name	N (value)	Storage Size (bytes)	Precision (digits)	Range 1 (From)	Range 1 (To)	Zero	Range 2 (From)	Range 2 (To)
FLOAT(N)	1 to 24 N is 1 to 24: 24	4	7	-3.40e38	-1.18e-38	0	1.18e-38	3.40e38
	25 to 53 N is 25 to 53: 53 Default: 53	8	15	-1.79e308	-2.23e-308	0	2.23e-308	1.79e308
REAL *		4	7	-3.40e38	-1.18e-38	0	1.18e-38	3.40e38

* REAL is an equivalent of FLOAT(24)

Strings

Character Non-Unicode

Name	Storage Size	Number of Characters (From)	Number of Characters (To)
`CHAR(N)`	1 byte per N	1	8,000
`VARCHAR(N)`	N is 1 to 8000: 1 byte per N + 2 bytes N is `MAX`: up to 2^{31} - 1 bytes (2,147,483,647 bytes or 2 GB)	1	8,000

Character Unicode

Name	Storage Size	Number of Characters (From)	Number of Characters (To)
`NCHAR(N)`	2 bytes per N	1	4,000
`NVARCHAR(N)`	N is 1 to 4000: 2 byte per N + 2 bytes N is `MAX`: up to 2^{31} - 1 bytes (2,147,483,647 bytes or 2 GB)	1	4,000

Binary

Name	Storage Size (bytes)	Number of Characters (From)	Number of Characters (To)
`BINARY(N)`	1 byte per N	1	8,000
`VARBINARY(N)`	N is 1 to 8000: 1 byte per N + 2 bytes N is `MAX`: up to 2^{31} - 1 bytes (2,147,483,647 bytes or 2 GB)	1	8,000

N - number of characters

Date and Time

Date and time in Gregorian calendar

Name	Precision	Rande (From)	Rande (To)	N: Storage Size (bytes)	Format	Note
DATE	1 day	0001-01-01	9999-12-31		YYYY-MM-DD	
TIME(N)	100 ns	00:00:00.0000000	23:59:59.9999999	0 to 2: 3 3 to 4: 4 5 to 7: 5 Not specified: 5	HH:MI:SS.fffffff	
DATETIME	3.33 ms	1753-01-01 00:00:00.000	9999-12-31 23:59:59.997	8	YYYY-MM-DD HH:MI:SS.fff	The last decimal fraction digit is rounded to 0, 3, 7
DATETIME2(N)	100 ns	0001-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999	0 to 2: 6 3 to 4: 7 5 to 7: 8 Not specified: 8	YYYY-MM-DD HH:MI:SS.fffffff	
SMALLDATETIME	1 min	1900-01-01 00:00:00	2079-06-06 23:59:59:00	4	YYYY-MM-DD HH:MI:SS	SS is always 00 29 seconds - rounded down 30 seconds - ronded up
DATETIMEOFFSET	100 ns	0001-01-01 00:00:00.0000000	9999-12-31 23:59:59.9999999	0 to 2: 8 3 to 4: 9 5 to 7: 10 Not specified: 10	YYYY-MM-DD HH:MI:SS.fffffff (- or +) HH:MI	Time zone offset Range: -14:00 - +14:00

N - number of digits to present the fraction of the second (0-7)

```
YYYY - 4-digits year
MM - 2-digits month
DD - 2-digits day
HH - 2-digits hour
MI - 2-digits minute
SS - 2-digits second
f - decimal fraction for seconds
```

Precision examples

TIME	TIME(0)	TIME(1)	TIME(2)	TIME(3)	TIME(4)	TIME(5)
14:10:36.7718695	14:10:37	14:10:36.8	14:10:36.77	14:10:36.772	14:10:36.7719	14:10:36.77187

TIME(6)	TIME(7)
14:10:36.771870	14:10:36.7718695

DATETIME2	DATETIME2(0)	DATETIME2(1)	DATETIME2(2)	DATETIME2(3)	DATETIME2(4)	DATETIME2(5)
1967-05-12 16:19:53.4739587	1967-05-12 16:19:53	1967-05-12 16:19:53.5	1967-05-12 16:19:53.47	1967-05-12 16:19:53.474	1967-05-12 16:19:53.4740	1967-05-12 16:19:53.47396

DATETIME2(6)	DATETIME2(7)
1967-05-12 16:19:53.473959	1967-05-12 16:19:53.4739587

DATETIMEOFFSET	DATETIMEOFFSET(0)	DATETIMEOFFSET(1)	DATETIMEOFFSET(2)	DATETIMEOFFSET(3)	DATETIMEOFFSET(4)
1967-05-12 16:35:10.8617596 +02:00	1967-05-12 16:35:11 +02:00	1967-05-12 16:35:10.9 +02:00	1967-05-12 16:35:10.86 +02:00	1967-05-12 16:35:10.862 +02:00	1967-05-12 16:35:10.8618 +02:00

DATETIMEOFFSET(5)	DATETIMEOFFSET(6)	DATETIMEOFFSET(7)
1967-05-12 16:35:10.86176 +02:00	1967-05-12 16:35:10.861760 +02:00	1967-05-12 16:35:10.8617596 +02:00

Conversion - Implicit and explicit

When we build expression or compare values, we need to have the same data type for the elements of the expression and the compared values.

Implicit conversion is the conversion that the SQL Server is doing automatically in the background without notifying that it is being made. It converts compatible (String and convertible to String, Numeric and convertible to Numeric and so on) data types.

Explicit conversion is what we do with built-in functions:

- CAST() or TRY_CAST()
- CONVERT() or TRY_CONVERT()
- PARSE() or TRY_PARSE()
- STR()

Data Types and Conversions

```
SELECT CAST('03.4.1965a' AS DATE)
SELECT TRY_CAST('03.4.1965a' AS DATE)

SELECT CONVERT(DATE, '03.4.1965a')
SELECT TRY_CONVERT(DATE, '03.4.1965a')

SELECT PARSE('123a' AS INT)
SELECT TRY_PARSE('123a' AS INT)

SELECT '>' + STR(123) + '<'
SELECT STR(123.45, 6, 1);
SELECT STR(123.45, 2, 2);
SELECT STR (FLOOR(123.45), 8, 3);
```

Error message on fail

NULL on fail

(No column name)
> 123<

(No column name)
123.5

(No column name)
**

(No column name)
123.000

Expression

Implicit conversion

```
SELECT '3' + 3 AS Expression;
GO
```

String

Number

Expression
6

Explicit conversion

```
SELECT 'My String ' + CONVERT(CHAR(1), 3) AS Expression;
GO
```

String

Numeric, converted to String

Expression
My String 3

Comparison

Implicit conversion

```
SELECT *
FROM Sales
WHERE DateOfSale = '1968-09-15';
GO
```

Date

String

Explicit conversion

```
SELECT *
FROM Sales
WHERE (MonthLiteral + ' ' + CAST(YearFiscal AS VARCHAR(9))) = 'September 1968';
GO
```

Numeric, converted to String

String

String

The VALUES clause creates a VR. This VR respects the **Data Type Precedence** (on page 64)

Clarification:

```
INSERT DatabaseName.SchemaName.TableName
  (
    ColumnString
    , ColumnNumeric
  )
VALUES
  (1, 2)                          ← Numeric, Numeric
  , ('3', 4)                      ← String, Numeric
  , ('D', '5');                   ← String, String
GO
```

In the first row 1 is Numeric. The data type of the first column will be defined as INT.
In the second row, the string '3' is implicitly converted to INT.
As the data type of the first column is defined as INT, the string 'D' can't be implicitly converted to INT.

The same is valid for any VR.

```
INSERT DatabaseName.SchemaName.TableName
  (
    ColumnString
    , ColumnNumeric
  )
SELECT 1, 2                       ← Numeric, Numeric
UNION ALL SELECT '3', 4           ← String, Numeric
UNION ALL SELECT 'D', '5';        ← String, String
GO
```

It is good practice to always use **ISO 8601** format for dates:

Date *ISO 8601*

YYYY-MM-DD

YYYY – Year (4 digits)
MM – Month (4 digits)
DD - Day (4 digits)

```
SELECT CAST('08/31/1965 14:11:56.3506429' AS DATE) AS [Date];
GO
```

→

Date
1965-08-31

Time *ISO 8601*

HH:mm:ss.fffffff

HH – hours (24-hour clock)
m - minutes (2 digits)
s – seconds (2 digits)
f – fraction of the second (3 to 7)

```
SELECT CAST('08/31/1965 14:11:56.3506429' AS TIME) AS [Time];
GO
```

Time
14:11:56.3506429

Date and Time *SQL Server*

YYYY-MM-DD HH:mm:ss.fffffff

```
SELECT CAST('08/31/1965 14:11:56.350' AS DATETIME) AS [Time];
GO

SELECT CAST('08/31/1965 14:11:56.3506429' AS DATETIME2) AS [Time];
GO
```

Time
1965-08-31 14:11:56.350

Time
1965-08-31 14:11:56.3506429

Data Type Precedence

When the DB converts implicitly, it converts data with a lower precedence data type to one with higher precedence.

The hierarchy is:

User-defined Data Types

Other

- SQL_VARIANT
- XML

Date and Time

- DATETIMEOFFSET
- DATETIME2
- DATETIME
- SMALLDATETIME
- DATE
- TIME

Numeric

Approximate Numeric

- FLOAT
- REAL

Exact Numeric

- NUMERIC
- DECIMAL
- MONEY
- SMALLMONEY
- BIGINT
- INT

SMALLINT
TINYINT
BIT

Other

TIMESTAMP
UNIQUEIDENTIFIER

String

Unicode

NVARCHAR
NCHAR

Character

VARCHAR
CHAR

Binary

VARBINARY
BINARY

Numeric and String - can be implicitly converted to Numeric
When we concatenate INT and NVARCHAR (can be implicitly converted to INT), the result is INT

```
SELECT 12 + '345' AS Result;
SELECT 12 + 'CDE' AS Result;
```

357 (12 + 345)

Conversion failed when converting the CHAR value 'CDE' to data type INT

Numeric and Date and Time (Implicitly converted to Date and Time):

```
SELECT 12 + GETDATE();
```

The built-in function GETDATE() returns **now** (**now** is 1968-08-13 15:10:29.017)
The result is 1968-08-25 15:10:29.017 (12 days added)

Collations

Collation

The way the:

- Server
- DB
- Column (table or view)

stores and represents **string** data type is specific for language (Japanese or Russian) or alphabet (Latin, Cyrillic or Indic). Collation is a set (code page) of characters, used in a language or alphabet.

SQL Server collation can be applied to:

- Server
- DB
- Table or view column

levels.

We define the collation in DDL (create or alter object) and in DML statements (select, join or filter the data).

Server Level

```
SELECT *
FROM sys.fn_helpcollations();
GO
```

Check all the collations on the server

name	
Albanian_BIN	Albanian, binary sort
Albanian_BIN2	Albanian, binary code point comparison sort
...	...

```
SELECT SERVERPROPERTY('Collation') AS ServerCollation;
GO
```

Check the current collation on the server

ServerCollation
SQL_Latin1_General_CP1_CI_AS

DB level

Create DB with specified collation

```
CREATE DATABASE Collations
  COLLATE Korean_Wansung_CS_AI_WS;
GO

SELECT name, collation_name
FROM Collations.sys.databases
WHERE name = N'Collations';
GO
```

name	collation_name
Collations	Korean_Wansung_CS_AI_WS

Check the collation of a specified database

```
USE Collations;
SELECT DATABASEPROPERTYEX('Collations', 'Collation') AS Collation;
GO
```

Column level

```sql
USE Collations;
GO

CREATE TABLE CollationsTable
(
  FirstName NVARCHAR(32) COLLATE Japanese_CS_AI_WS
  , Email VARCHAR(128)
);
GO

SELECT
  [name] AS ColumnName
  , [collation_name] AS ColumnCollation
FROM Collations.sys.columns
WHERE
  OBJECT_NAME([object_id]) = 'CollationsTable'
  --AND [name] = N'Email';
GO
```

Specified collation for column **FirstName**

Column **Email** uses the default collation of the DB (**Korean_Wansung_CS_AI_WS**)

Check column collation

Object name

Uncomment to filter exact column

ColumnName	ColumnCollation
FirstName	Japanese_CS_AI_WS
Email	Korean_Wansung_CS_AI_WS

Overwrite the collation

The keyword COLLATE, added to column identifier, switches the collation of the column:

In the SELECT clause

```sql
SELECT ColumnName COLLATE Latin1_General_CI_AI
FROM MyTable;
GO
```

In the FROM clause

```sql
SELECT
  T1.Column1
  , T2.Column2
FROM
  Table1 AS T1
  JOIN Table2 AS T2
    ON T1.Column1 = T2.Column1 COLLATE Latin1_General_CI_AI
GO
```

T2.Column1 is not **Latin1_General_CI_AI** and we need to equalize the collations in order to join the objects

T1.Column1 is **Latin1_General_CI_AI**

In the WHERE clause

```
SELECT
       T1.Column1
       , T1.Column2
FROM
       Table1 AS T1
       JOIN Table2 AS T2
              ON T1.Column1 = T2.Column1
WHERE T1.Column1 = T2.Column1 COLLATE Latin1_General_CI_AI;
GO
```

Equalize the collations to apply to filter condition

When we UNION recordsets, the collation of the matching columns have to be the same

UNION on page 172

The flags in the collation name explain the properties of the collation:

Case Sensitivity
- CI - Case Insensitive
- CS - Case Sensitive

Accent Sensitivity
- AI - Accent Insensitive
- AS - Accent Sensitive

Kana (Japanese kana characters)
- KI - Kana Insensitive
- KS - Kana Sensitive

Width
- WI - Width Insensitive
- WS - Width Sensitive

Code Page CP(X) (X is between1 and 4)
- CP1 - Code page 1252 (Latin1 (ANSI))
- CP1251 - Code Page 1251 (Cyrillic)
- CP949 - Code page 949 (Korean)

Binary
- BIN - older BIN collations
- BIN2 - newer BIN2 collations

Unicode
- CS - Supplementary Characters

Collation with name **SQL_Latin1_General_CP1_CI_AS** is:
- CP1 - Code Page 1252
- CI - Case Insensitive
- AS - Accent sensitive

NULL and 3VL (Three-valued Logic)

Based on the standard ISO 9075-1: 2011, `NULL` is a "Special value that is used to indicate the absence of any data value".
When we add new nullable column to an existing table, it stores `NULL`, before we insert values.
When we create (`DECLARE`) a new variable, its value is `NULL`, i.e. a value is missing.
`NULL` is a marker that shows an absence of data.
`NULL` is not an empty string, space, multiple spaces or 0 (zero), all of which are actually values. It is an absence of value - **unknown** or **nothing**.

The 3VL (Three-value Logic) explains that the `NULL` marker or placeholder can't result in `True` or `False`.
The result of comparison with `NULL` is **unknown**.
This is why we compare `NULL` in a table column or variable with the special keyword `IS` (`IS NULL`, `IS NOT NULL`) and manipulate `NULL` separately from the other values.
The question should we use or not use `NULL` in the DB architecture and development is not explicit. Both options have their pros and cons.

The following example explains 3VL

Customers

FirstName	LastName
John	Smith
Anna	*NULL*
Deborah	Smilesh
Melanie	Johnson

If we need to select the rows, where the last name doesn't start with **Smi**, the returned recordset will be:

```sql
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE LastName NOT LIKE 'Smi%';
GO
```

FirstName	LastName
Melanie	Johnson

The result above is correct, because when we compare column **LastName** to **Smi***, the DBE asks the question "Column LastName on this row begins with **Smi**?" and returns the rows, corresponding to answer **Yes** (True):

FirstName	LastName
John	Smith
Anna	*NULL*
Deborah	Smilesh
Melanie	Johnson

LastName begins with 'Smi'?	
False	Question
Unknown	
False	Answer
True	

To select the rows that we need, we need to select the rows where "**LastName** doesn't start with **Smi** or **LastName** is **unknown**?":

```sql
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE
  (
    LastName NOT LIKE 'Smi%'
    OR LastName IS NULL
  );
GO
```

FirstName	LastName
Anna	*NULL*
Melanie	Johnson

NULL and 3VL (Three-valued Logic)

We use built-in functions to replace `NULL` with a real value and vice versa:

- `ISNULL(`*`ValueToCheck, SpecifiedValue`*`)` - When *`ValueToCheck`* `IS NULL` *`SpecifiedValue`* is returned
- `NULLIF(`*`ValueToCheck, SpecifiedValue`*`)` - When *`ValueToCheck`* is equal to *`SpecifiedValue`*, `NULL` is returned

The built-in function `COALESCE` returns the first not `NULL` value from the specified values:

- `COALESCE(`*`ValueToCheck1, ValueToCheck2, ValueToCheck3,`* `'N/A')` - returns the first not `NULL` value and **N/A** if all the precedent values are `NULL`. If all the specified values are `NULL`, `NULL` is returned.

`NULL` can't be used to join objects (table, view, table-valued function).

`JOIN` on page 128

Special attention has to be paid to `NULL`, because knowing its logic helps us to create more robust and stable code.

Operators

Characters or reserved words acting on operands.

Left Operand	Operator	Right Operand	Operator Category	Left Operand Type	Right Operand Type
19	+	8	Arithmetic	Value	Value
Age	>	18	Comparison	Table cell	Value
@Age	=	18	Assignment	Variable	Value
(19 > 8)	AND	(Age > 18)	Logical	Expression (Value, Value)	Expression (Table cell, Value)
@FirstName	+	'Smith'	Concatenation	Variable	String

The operators are:

Arithmetic

Does mathematical operation. Both operands have to be a numeric data type.

Operator	Name	Example	Description	Result
+	Addition	19 + 8	Adds 8 to 19	27
-	Subtraction	19 - 8	Subtracts 8 from 19	11
*	Multiplication	19 * 8	Multiplies 19 by 8	152
/	Division	19 / 8	Divides 19 by 8	2.375
%	Modulo	19 % 8	Divides 19 by 8 and returns the remainder	3

8 - operand

19 - operand

Comparison (Relational)

Compares the values of the operands and returns true, false or unknown (boolean operator). The boolean operators test the truth or falsity of the condition.
When the values of the operands can't be implicitly converted, we convert them with the built-in functions CAST(), TRY_CAST(), CONVERT(), TRY_CONVERT(), PARSE(), TRY_PARSE(), STR().

Operator	True If	Compares if the left operand	Result
=	19 = 8	equals	False
<> !=	19 <> 8 19 != 8	doesn't equal	True
>	19 > 8	is greater than	True
<	19 < 8	is less than	False
>= !<	19 >= 8 19 !< 8	is greater than or equal to isn't less than	True
<= !>	19 <= 8 19 !> 8	is less than or equal to isn't greater than	False
		the right operand	

Data Types and Conversions on page 56

! reverses the meaning of the operator.
Not ISO standard.

Operators

Assignment

Sets a value to the table cell or variable.

Operator	Name	Description	Example	Current Value	New Value
=	Assignment	Overwrites the current value.	`UPDATE Column1 = 8;` `SELECT @Variable1 = 8;`	123456	8
+=	Addition	Adds 8 to the current value and assigns the result	`UPDATE Column1 += 8;` `SELECT @Variable1 += 8;`	19	27
-=	Subtraction	Substracts 8 from the current value and assigns the result	`UPDATE Column1 -= 8;` `SELECT @Variable1 -= 8;`	19	11
*=	Multiplication	Multiplies the current value by 8 and assigns the result	`UPDATE Column1 *= 8;` `SELECT @Variable1 *= 8;`	19	152
/=	Division	Divides the current value by 8 and assigns the result	`UPDATE Column1 /= 8;` `SELECT @Variable1 /= 8;`	19	2.375
%=	Modulo	Divides the current value by 8 and assigns the remainder	`UPDATE Column1 %= 8;` `SELECT @Variable1 %= 8;`	19	1.5

Logical

Tests the truth of a condition, applied on the operands (boolean operator).

Operator	Example	Result	NOT (reverses the meaning of the operator)
AND	`(Age >= 18) AND (DateRegistered = '1968-05-18')`	True if **both** operands (expressions) return True	AND NOT
OR	`(Age >= 18) OR (DateRegistered = '1968-05-18')`	True if **one** of the operands (expressions) return True	OR NOT
EXISTS	`EXISTS (Subuery)`	True if the subquery returns at least one row	NOT EXISTS
BETWEEN	`Age BETWEEN 18 AND 32`	True if **Age** is in the range from 18 to 32 (including)	NOT BETWEEN
IN	`Age IN (18, 22)`	True if **Age** matches any value in the list 18, 22	NOT IN
IS NULL	`Age IS NULL`	True if **Age** is NULL	IS NOT NULL
LIKE	`FirstName LIKE 'Smi%'`	True if **FirstName** starts with 'Smi'	NOT LIKE

		Operator	
Left Operand	**Right Operand**	**AND**	**OR**
(8 = 8)	(12 = 12)	True	True
(8 = 8)	(12 = 13)	False	True
(8 = 9)	(12 = 13)	False	False

True AND True = True
True AND False = False
False AND False = False

True OR True = True
True OR False = True
False OR False = False

Concatenation

Joins **String** data type operands.

Operator	Name	Example	Description	Result
`+`	Concatenation	`'John' + ' ' + 'Smith'`	Joins **John** to **space** and to **Smith**	John Smith

Operator Precedence

The order in which the operators are executed:

1. Multiplication, Division, Modulo
2. Addition, Subtraction, Concatenation
3. Comparison
4. `NOT`
5. `AND`
6. `BETWEEN`, `IN`, `LIKE`, `OR`
7. Assignment

To force the precedence, we use round brackets (`()`).

The execution order of the operations are:

Order	Operation	Direction of the execution	T-SQL	Result	Explanation
1	Parentheses	Left to right	`SELECT 7 + 6 * 5;`	37	7 + 30
			`SELECT (7 + 6) * 5;`	65	13 * 5
2	Exponents (power and root)	Right to left	`SELECT POWER(POWER(2, 3), 2);`	64	$2^{2^3} = 2^8$
			`SELECT 7 + 6 * POWER(2, 3);`	55	7 + 6 * 8 = 7 + 48
			`SELECT - 4 + POWER(2, 3);`	4	- 4 + 8 = 8 - 4
3	Multiplication and division	Left to right	`SELECT 7. / 6. * 5.;`	5.833330	1.166666 * 5
			`SELECT 5. * 6. / 7.;`	4.285714	30 / 7
4	Addition and subtraction	Left to right	`SELECT 7 + 6 - 5;`	8	13 - 5
			`SELECT 5 - 6 + 7;`	6	- 1 + 7 = 7 - 1

Examples:

Arithmetical Operators

DB defined precedence:

2 + 4 * 6 = 26

4 * 6 = 24 + 2 = 26

Not 2 + 4 = 6 * 6 = 36

Parentheses to force the precedence:

(2 + 4) * 6 = 36

2 + 4 = 6 * 6 = 36

Logical Operators

DB defined precedence:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE
  Age >= 18
  OR IsEligible = 1
  AND DateRegistered = '1969-12-17';
GO
```

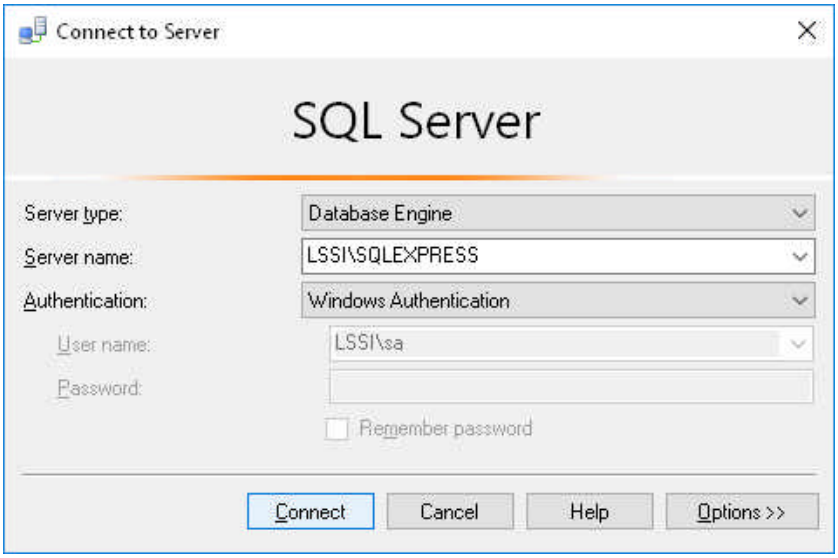
Parentheses to force the precedence:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE
  (
    Age >= 18
    OR IsEligible = 1
  )
  AND DateRegistered = '1969-12-17';
GO
```

Until now we have defined the theory and the basics. The fun starts here :-) From now on the main tool to learn T-SQL is **SSMS** (SQL Server Management Studio).

Log in SSMS

To start **SQL Server Management Studio**, click the Start menu and Type **SQL Server 2016 Management Studio** in the search box, then click on **SQL Server 2016 Management Studio** in the top section.



Server type: Database Engine
Server name: Computer name of the server, where the SQL Server is installed\Instance name
Authentication:

- **Windows Authentication** - uses windows credentials, i.e. the user, logged in Windows (used in the book)
- **SQL Server Authentication** - uses DB logins, defined in the SQL Server
- **Active Directory Password Authentication** and
- **Active Directory Integrated Authentication** - use the Active Directory Authentication Library (ADAL) authentication

When login in with **Windows Authentication**, the text fields **Login**, **Password** and the checkbox **Remember password** are not active.

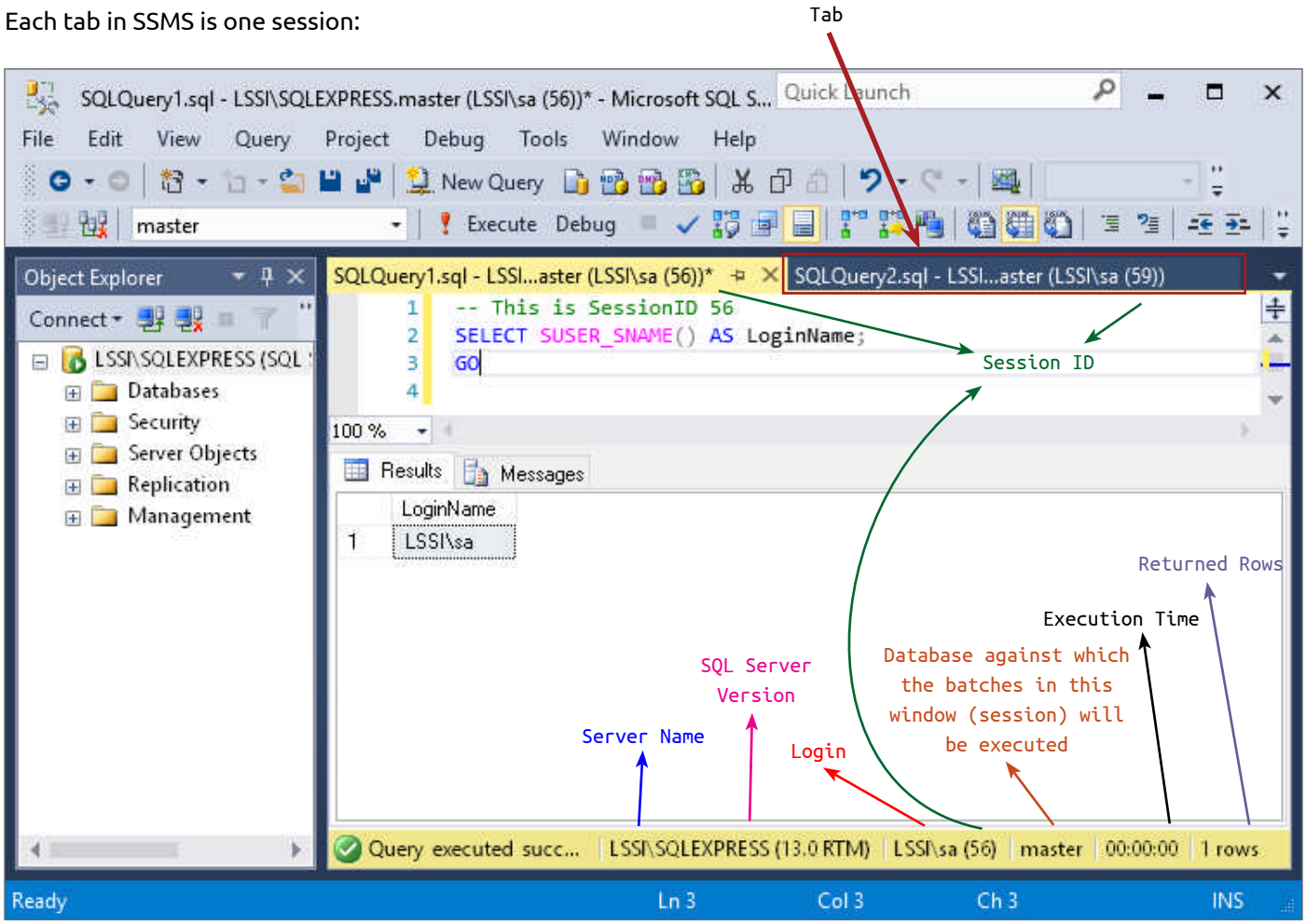
When we login with SQL Server Authentication, we specify:

- User name - login name, existing in SQL Server
- Password - password, assigned to the login name
- Remember password (checked) - SQL Server remembers the password for this login name and we don't need to type it in future logins

We can use the built-in function SUSER_SNAME() to verify our login name:

```
SELECT SUSER_SNAME() AS LoginName;
GO
```

Each tab in SSMS is one session:

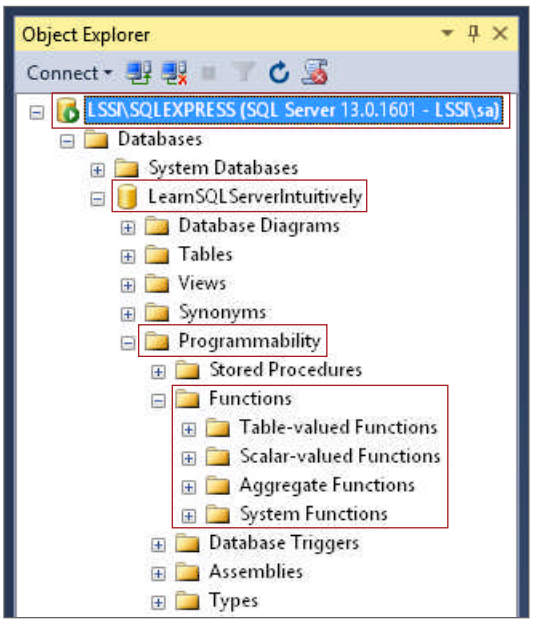


On the left-hand side is the Object Explorer (DBs, Tables, Views, Functions, Stored Procedures,etc.).

The right part of the window is the area where we write our T-SQL code.

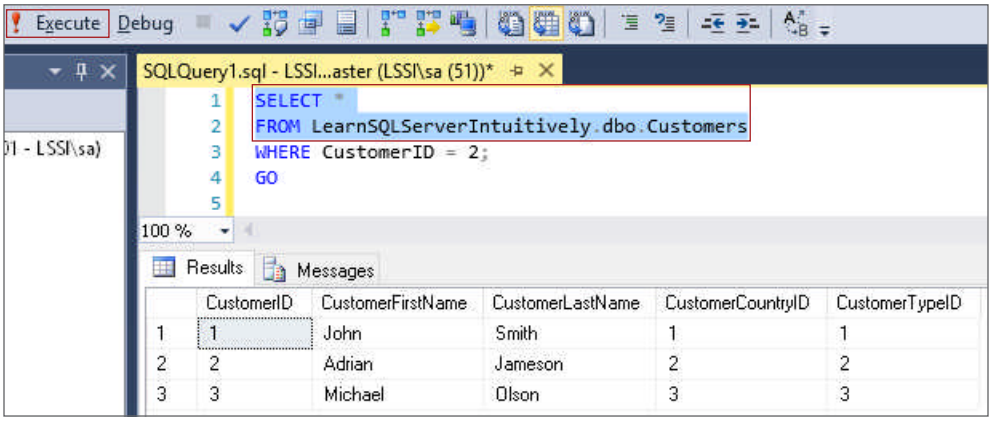
The bottom-right area is where we examine the result of the batch and the error messages (if any).

In SSMS the functions are in the database's Programmability section:



Tables, views and stored procedures are in the database's Programmability section:

By selecting a portion of the code and click **Execute** (F5), we execute only the selected portion of the code:



By selecting a keyword and pressing **Shift + F1**, we search **Books Online** for this keyword.

Constraints

As the name suggests, constraints constrain data. These are rules that belong to the table columns that limit the values that we can stored in the columns.

The constraints are:

PRIMARY KEY (PK)

- A column with a PK constraint can store only unique values and can't store `NULL`
- PK uniquely identifies the rows in the table
- Only one PK can exist in a table

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org

PK
`NOT NULL`
`UNIQUE`

```
Insert NULL fail
Insert 3 fail (duplicate)
```

Composite (compound) PK

- The PK constraint is defined in multiple columns
- Neither of the columns can store `NULL`
- The combination of the column values define the unique value for the PK

Customers

FirstName	LastName	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
John	Smith	john.smith@customer1.com
Melanie	Larson	melanie.larson@customer4.biz
Zak	Smith	NULL

PK
`NOT NULL` `NOT NULL`
`UNIQUE`

```
Insert NULL fail      Insert NULL fail
Insert 'John'         Insert 'Henderson' OK
Insert 'John'         Insert 'Smith'     fail (duplicate)
```

It is good practice to define a PK constraint in all the tables in the DB to enforce the referential (PK - FK) integrity of the database.

To view the **PK** in the **Object Catalog**:

```
USE LearnSQLServerIntuitively;
GO

SELECT
  OBJECT_NAME([parent_object_id]) AS TableName
  , [name] AS PKName
FROM sys.key_constraints
WHERE [type] = 'PK';
GO
```

FOREIGN KEY (FK)

- Constraints the column to store only values that exist in the referencing PK column in another table
- Ensures the referential integrity of the data

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
PK			

Reference

Sales

SaleID	CustomerID	ItemID	Quantity	Prce	DateOrdered
1	3	3	3	1.24	1967-10-17
2	3	5	5	2.43	1968-05-08
3	4	2	2	12.44	1965-09-04
4	3	3	7	5.88	1968-12-17
5	5	1	1	28.19	1968-03-04
PK	FK				

Only values, existing in the PK reference column

```
Insert 6 - fail (doesn't exist)
Insert 3 - OK (exist)
Insert NULL - OK
```

FK Referential Actions

When we update or delete a value in a PK column, the FK can be automatically actioned with the following rules:

- Delete - `ON DELETE`
- Update - `ON UPDATE`

The referential actions, applied on the rules are:

- `NO ACTION` - Used when no action is specified (default)
- `CASCADE` - FK is automatically updated with the new PK value or deleted
- `SET NULL` - the FK is set to `NULL`
- `SET DEFAULT` - the FK is set to the `DEFAULT` value of the column

	Rules	
Referential Action	**Update PK**	**Delete PK**
`NO ACTION`	Not Allowed	Not Allowed
`CASCADE`	Updates FK with PK	Deletes the FK
`SET NULL`	Updates FK to `NULL`	Updates FK to `NULL`
`SET DEFAULT`	Updates FK with the `DEFAULT` value	Updates FK with the `DEFAULT` value

CASCADE (UPDATE)

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
PK			

Update **CustomerID** to 6

Reference

Sales

SaleID	CustomerID	ItemID	Quantity	Prce	DateOrdered
1	3	1004	3	1.24	1967-10-17
2	1	5273	5	2.43	1968-05-08
3	4	5432	2	12.44	1965-09-04
4	3	8541	7	5.88	1968-12-17
5	5	9514	1	28.19	1968-03-04
PK	FK				

Updates FK (**CustomerID**) to 6

CASCADE (DELETE)

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
PK			

DELETE **CustomerID** = 3

Reference

Sales

SaleID	CustomerID	ItemID	Quantity	Prce	DateOrdered
1	3	1004	3	1.24	1967-10-17
2	1	5273	5	2.43	1968-05-08
3	4	5432	2	12.44	1965-09-04
4	3	8541	7	5.88	1968-12-17
5	5	9514	1	28.19	1968-03-04
PK	FK				

Deletes row where FK (**CustomerID**) = 3

SET NULL (UPDATE, DELETE)

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
PK			

Update **CustomerID** to 6 or delete

Reference

Sales

SaleID	CustomerID	ItemID	Quantity	Prce	DateOrdered
1	3	1004	3	1.24	1967-10-17
2	1	5273	5	2.43	1968-05-08
3	4	5432	2	12.44	1965-09-04
4	3	8541	7	5.88	1968-12-17
5	5	9514	1	28.19	1968-03-04
PK	FK				

Updates FK (**CustomerID**) to NULL

SET DEFAULT (UPDATE, DELETE)

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
PK			

Update **CustomerID** to 6 or delete

Reference

Sales

SaleID	CustomerID	ItemID	Quantity	Prce	DateOrdered
1	3	1004	3	1.24	1967-10-17
2	1	5273	5	2.43	1968-05-08
3	4	5432	2	12.44	1965-09-04
4	3	8541	7	5.88	1968-12-17
5	5	9514	1	28.19	1968-03-04
PK	FK				

Updates FK (**CustomerID**) to the DEFAULT value of the column

To view the **FK** in the **Object Catalog**:

```
SELECT
  OBJECT_NAME([parent_object_id]) AS TableName
  , [name] AS FKName
FROM sys.foreign_keys;
GO
```

To view the **Referential Actions** in the **Object Catalog**:

```
SELECT *
FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS;
GO
```

DEFAULT (DF)

When a row is inserted and no value is specified for a column with a DF value, the default value is inserted. DF is a replacement for `NULL`.

Customers

CustomerID	FirstName	LastName	Email	DateRegistered
1	Anabel	Larson	anabel.larson@customer5.info	1967-12-04
2	Anna	Laurier	anna.laurier@customer2.net	1966-05-17
3	Beverly	NULL	NULL	1964-08-11
4	John	Smith	john.smith@customer1.com	1966-11-07
5	John	Smith	john.smith@customer3.org	1965-08-11
				DF DateRegistered = today
	Adam	Hasen	ah@customer17.com	
6	Adam	Hasen	ah@customer17.com	1969-12-08

DF Definition

Insert new line without specifying **DateRegistered**

New line with default **DateRegistered** (today)

Today is 1969-12-08

To list the **DF** in the **Object Catalog**:

```
SELECT
  OBJECT_NAME([parent_object_id]) AS TableName
  , [name] AS DFName
  , [definition] AS DFDefinition
FROM sys.default_constraints
GO
```

CHECK (CK)

Verifies if the value in a CK column is valid against the CK definition.

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
			CK Column **Email** contains @ sign
6	Adam	Hasen	ah@customer17.com
7	Adam	Hasen	ah#customer17.com
4	John	Smith	john.smith&customer1.com

CK Definition

Insert valid **Email** - **OK**

Insert not valid **Email** - **fail**

Update **CustomerID** = 4 with not valid **Email** - **fail**

To view the **CK** in the **Object Catalog**:

```
SELECT
  OBJECT_NAME([parent_object_id]) AS TableName
  , [name] AS CKName
  , [definition] AS CKDefinition
FROM sys.check_constraints
GO
```

UNIQUE (UQ)

Constraints the values in a column to be unique.

Customers

Custome-rID	First-Name	Last-Name	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
			UQ
6	Adam	Hasen	ah@customer17.com
7	Adam	Hasen	ah@customer17.com
4	John	Smith	ah@customer17.com

Insert not existing value in **Email** - **OK**

Insert duplicate in **Email** - **fail**

Update **CustomerID** = 4 with duplicate **Email** - **fail**

To view the **UQ** in the **Object Catalog**:

```
SELECT
  OBJECT_NAME(C.[object_id]) AS TableName
  , C.[name] AS UQColumn
  , I.[name] AS UQName
FROM
  sys.indexes AS I
  JOIN sys.index_columns AS IC
    ON I.[object_id] = IC.[object_id]
    AND I.[index_id] = IC.[index_id]
  JOIN sys.columns AS C
    ON IC.[object_id] = C.[object_id]
    AND IC.[column_id] = C.[column_id]
WHERE I.[is_unique_constraint] = 1;
GO
```

NOT NULL (NN)

Constrains a column not to store `NULL`. We can't insert or update a row, without specifying a value for the NN column.

Customers

CustomerID	FirstName	LastName	Email	
1	Anabel	Larson	anabel.larson@customer5.info	
2	Anna	Laurier	anna.laurier@customer2.net	
3	Beverly	NULL	bev@customer7.pl	
4	John	Smith	john.smith@customer1.com	
5	John	Smith	john.smith@customer3.org	
			NOT NULL	
6	Adam	Hasen	ah@customer17.com	← Insert **Email** - **OK**
7	Adam	Hasen		← Insert row and skip **Email** - **fail**
4	John	Smith	NULL	← Update **CustomerID** = 4 with **Email** = NULL - **fail**

Data Type

The data type that a column in a table can store is also a constraint. We can't insert **String** into a column with a **Numeric** data type.

Constraints are created, altered and deleted with **DDL statements** (on page 86)

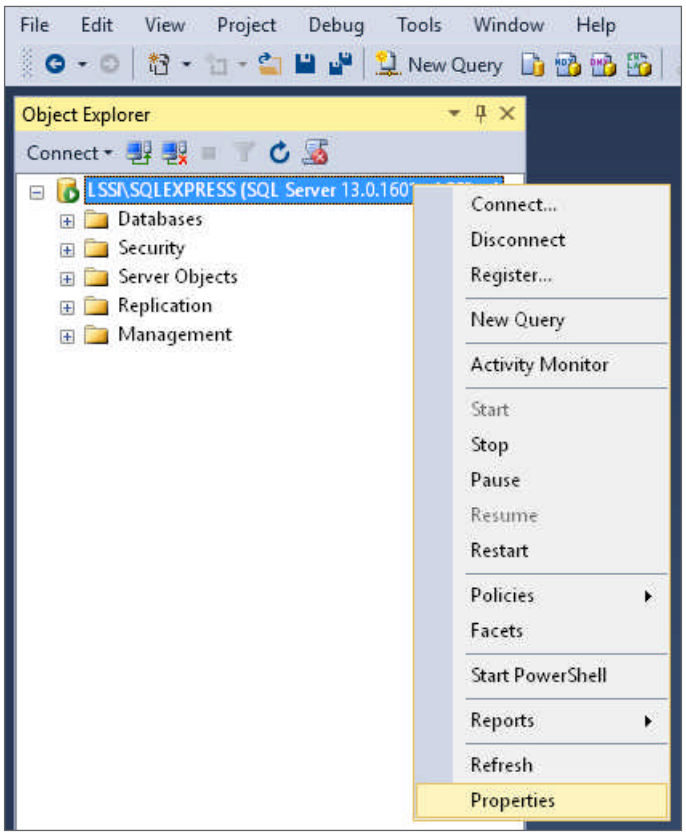
Naming Conventions on page 40

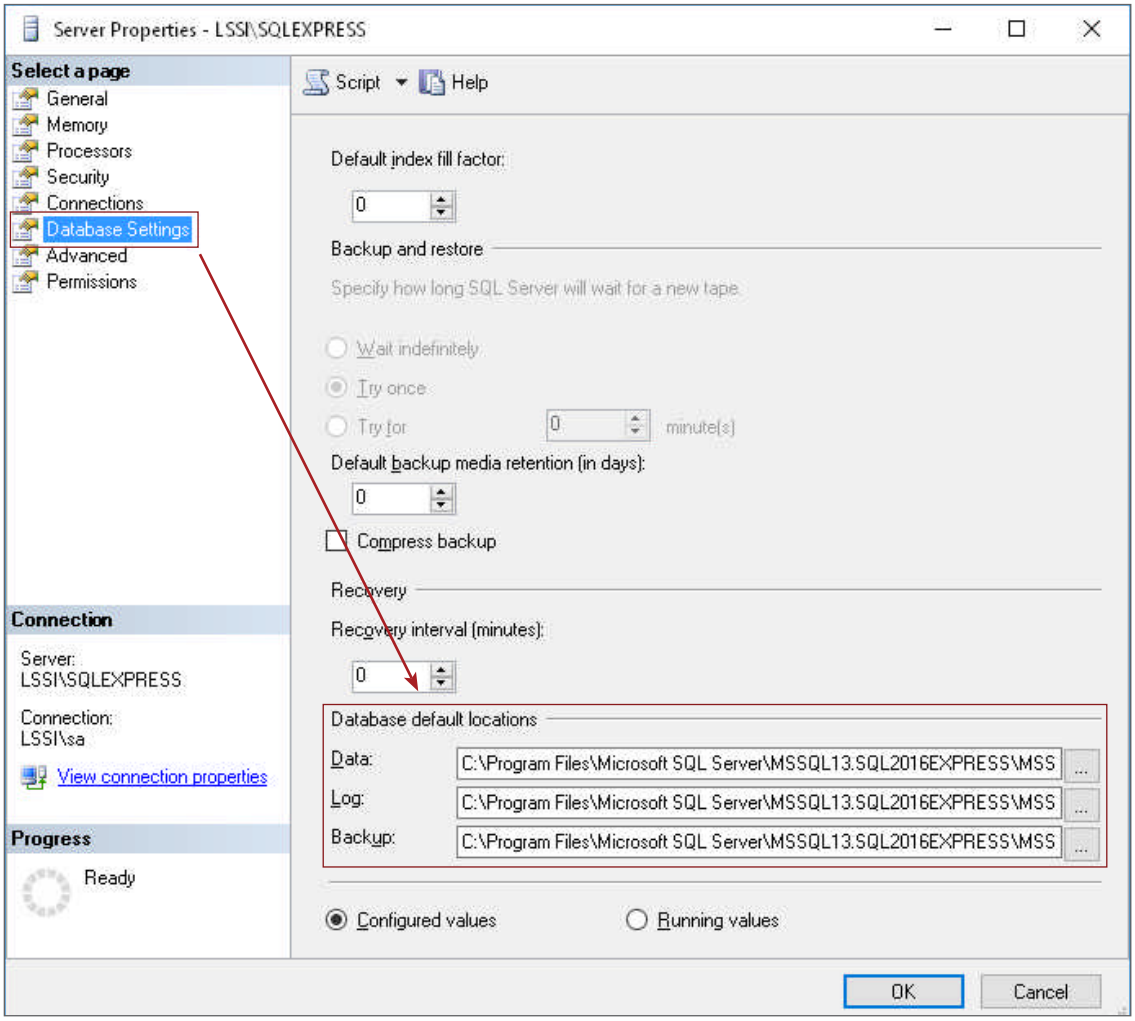
CREATE Statement

Creates new DB objects. It is good practice to use the DSO naming convention to point to the database and the schema, where we need to create the object.
The keyword CREATE is the first clause in a DDL statement, followed by the type of the object (DATABASE, SCHEMA, TABLE, VIEW, etc.) and the name of the object (identifier).

CREATE DATABASE

Creates a new database by using the default settings for DB location and DB size. We can see where the default DB location is:
In SSMS **Object Explorer** (**F8**) ⟶ (right click on the server name) ⟶ **Properties** ⟶ **Database Settings** ⟶ **Database default locations**





To create a DB, we execute the following statement:

```
CREATE DATABASE LearnSQLServerIntuitively;
GO
```

SQL Server creates two files, used by the DBE to store and manipulate the data:

1. LearnSQLServerIntuitively.mdf – stores the DB objects and the data
2. LearnSQLServerIntuitively_log.ldf – used for transactional purposes on data manipulation

To query the **Object Catalog** for information about the **DB**:

```
SELECT *
FROM LearnSQLServerIntuitively.sys.databases
WHERE [name] = 'LearnSQLServerIntuitively';      ← Comment this line to select all the DBs
GO
```

The **DB files**:

```
SELECT *
FROM LearnSQLServerIntuitively.sys.database_files;
GO
```

To point to a specific database, we use the keyword `USE` in the beginning of the batch:

```
USE LearnSQLServerIntuitively;
GO
```

After we change the DB with `USE`, all the following statements in the batch are be executed against DB **LearnSQLServerIntuitively**.

`CREATE DATABASE` limitations:

- Storage size: 524,272 terabytes
- Number of databases: 32,767

CREATE SCHEMA

After the DB exists, we create the next level DB object - **the schema**. To create a schema:

```
CREATE SCHEMA Sales;
GO
```

To view the schemas in the DB, we query the **Object Catalog**:

```
SELECT *
FROM LearnSQLServerIntuitively.sys.schemas;
GO
```

We use the default schema **dbo** in the book.

CREATE TABLE

After the keyword `CREATE` and the identifier for the name of the table, we list (in brackets) the column names and their data types:

```
CREATE TABLE [DatabaseName].[SchemaName].[Customers]
  (
   FirstName NVARCHAR(32)
   , LastName NVARCHAR(64)
```

Identifier

Between the braces, we list the column definition (the properties of the column): name (identifier), data type, identity specification, collation, nullability and constraint

```
    , [Email Address] VARCHAR(128)
  );
GO
```

Data type

Square brackets for identifiers that contain special characters or reserved words

We can **create a table** and the **constraints** belonging to the table in one DDL CREATE statement:

PRIMARY KEY Constraint (PK)

To create the PK and let the DBE generate the name for the PK:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    CustomerID INT IDENTITY(1, 1) PRIMARY KEY
    , FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
  );
GO
```

PK with **automatically generated name**

To specify a name for the PK right after column name and data type:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    CustomerID INT IDENTITY(1, 1)
        CONSTRAINT PK__Customers
        PRIMARY KEY
    , FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
  );
GO
```

PK with **specified name**

or at the end of the column list:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    CustomerID INT IDENTITY(1, 1)
    , FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
    , CONSTRAINT PK__Customers
      PRIMARY KEY (CustomerID)
```

PK with **specified name** on column **CustomerID**

```
  );
GO
```

Composite PK with specified name at the end of the column list:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
    , CONSTRAINT PK__Customers__FirstName_LastName
       PRIMARY KEY (FirstName, LastName)
  );
GO
```

Composite PK on columns **First-Name, LastName** with **specified name**

FOREIGN KEY Constraint (FK)

Both referenced columns (PK - FK) have to be the same data type.

The reference table must exist with a PK.

We can add a FK to column and let the DBE generate the FK name:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Sales
  (
    SaleID INT
    , CustomerID INT
      FOREIGN KEY REFERENCES LearnSQLServerIntuitively.dbo.Customers (CustomerID)
    , ItemID INT
    , Quantity INT
    , Price MONEY
    , DateOrdered DATETIME
    , CONSTRAINT PK__Sales PRIMARY KEY (SaleID)
  );
GO
```

FK with automatically generated name to table **Customers**, column **CustomerID**

PK with **specified name** on column **SaleID**

We can specify the FK name:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Sales
  (
    SaleID INT
    , CustomerID INT
      CONSTRAINT FK__Sales__CustomerID
        FOREIGN KEY REFERENCES LearnSQLServerIntuitively.dbo.Customers (CustomerID)
```

FK with specified name to table **Customers**, column **CustomerID**

```
    , ItemID INT
    , Quantity INT
    , Price MONEY
    , DateOrdered DATETIME
    , CONSTRAINT PK__Sales
      PRIMARY KEY (SaleID)
  );
GO
```

PK with specified name on column SaleID

or

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Sales
  (
    SaleID INT
    , CustomerID INT
    , ItemID INT
    , Quantity INT
    , Price MONEY
    , DateOrdered DATETIME
    , CONSTRAINT PK__Sales
      PRIMARY KEY (SaleID)
    , CONSTRAINT FK__Sales__CustomerID
      FOREIGN KEY (CustomerID)
      REFERENCES LearnSQLServerIntuitively.dbo.Customers (CustomerID)
  );
GO
```

PK on column SaleID

FK on column CustomerID to table Customers, column CustomerID

To add the referential actions to the FK, we list them after the FK definition:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Sales
  (
    SaleID INT
    , CustomerID INT
    , ItemID INT
    , Quantity INT
    , Price MONEY
    , DateOrdered DATETIME
    , CONSTRAINT PK__Sales
      PRIMARY KEY (SaleID)
    , CONSTRAINT FK__Sales__CustomerID
      FOREIGN KEY (CustomerID)
```

PK on column SaleID

FK on column CustomerID to table Customers, column CustomerID

```
      REFERENCES LearnSQLServerIntuitively.dbo.Customers (CustomerID)
        ON DELETE CASCADE                                          ← Delete rule
        ON UPDATE CASCADE                                          ← Update rule
  );
GO
```

Referential Actions

DEFAULT Constraint (DF)

To create a DEFAULT constraint, we add the keyword DEFAULT and the constraint definition after the column name and data type:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
    , [Date Registered] DATETIME
      DEFAULT (GETDATE())                                          ← DF with automatically generated name
  );
GO
```

To specify a name for the constraint, we add the keyword CONSTRAINT, followed by the constraint name and definition:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
    , [Date Registered] DATETIME
      CONSTRAINT DF__Customers_DateRegistered__GetDate             ← DF with specified name
        DEFAULT (GETDATE())
  );
GO
```

CHECK Constraint (CK)

To create the CK with an automatically generated name, we add the keyword CHECK next to the column name and data type, followed by the constraint definition:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
```

```
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
      CHECK (CHARINDEX('@', [Email Address]) > 0)
    , [Date Registered] DATETIME
  );
GO
```

CK with automatically generated name

To specify name, we use the keyword CONSTRAINT after the column name and data type:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
      CONSTRAINT CK__Email__AtSign
        CHECK (CHARINDEX('@', [Email Address]) > 0)
    , [Date Registered] DATETIME
  );
GO
```

CK with specified name

UNIQUE Constraint (UQ)

To create UNIQUE constraint with an automatically generated name, we add the keyword UNIQUE after the column name and data type:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
      UNIQUE
  );
GO
```

UQ with automatically generated name

To specify the constraint name, we use the keyword CONSTRAINT after the column name and data type:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
```

```
    , [Email Address] VARCHAR(128)
      CONSTRAINT UQ__Customers__Email
        UNIQUE
  );
GO
```

UQ with specified name

or we add the keyword CONSTRAINT after the column list

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128)
    , CONSTRAINT UQ__Customers__Email
      UNIQUE ([Email Address])
  );
GO
```

UQ with specified name

NOT NULL **Constraint (NN)**

To create **NN** constraint for a column, add NOT NULL after the column name and data type:

```
CREATE TABLE LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , [Email Address] VARCHAR(128) NOT NULL
  );
GO
```

NOT NULL Constraint

CREATE TABLE limitations:

- Maximum number of columns: 1,024
- Storage size per row: 8,060 bytes

Tables on page 194

CREATE VIEW

The identifier after CREATE VIEW doesn't allow us to specify a DatabaseName. We use **SO** ([Schema],[Object]) naming convention as the identifier.
After the keyword AS, we write the SELECT statement that creates the recordset, returned by the view.

To create a view:

```sql
USE LearnSQLServerIntuitively;
GO

CREATE VIEW dbo.vw_Sales
AS
  SELECT
    C.FirstName
    , C.LastName
    , S.DateOfSale
    , S.Quantity
    , S.Price
    , (S.Quantity * S.Price) AS LineTotal
  FROM
    LearnSQLServerIntuitively.dbo.Customers AS C
    JOIN LearnSQLServerIntuitively.dbo.Sales AS S
      ON C.CustomerID = S.CustomerID;
GO
```

View definition

Views on page 220

`CREATE VIEW` limitations:

- `ORDER BY` clause is not allowed in the view definition
- **DSO** ([Database].[Schema].[Object]) is not allowed, we use **SO** ([Schema].[Object]) identifier structure
- `CREATE VIEW` has to be the first statement in the batch

CREATE PROC (CREATE PROCEDURE)

`CREATE PROC` creates a new Stored Procedure. After the SP name, we specify the parameter(s) list and the SP definition:

```sql
CREATE PROC dbo.usp_EligibleCustomers
  (
    @Paramater1 INT
    , @Parameter2 DECIMAL(10, 2)
    , @Parameter3 MONEY
  )
AS
BEGIN
  ... Statement 1...
  ... Statement 2...
  ...
  ... Statement n...;
END
GO
```

SO (not **DSO**) naming convention

Stored Procedure **Parameters List**

Stored Procedure **Definition**

`CREATE PROC` limitations:

- Number of parameters: 2,100

Stored Procedures on page 240

CREATE FUNCTION

After the CREATE FUNCTION, we specify the function name, the parameter(s) list and the function definition.

Scalar-valued:

```
CREATE FUNCTION dbo.udf_CustomerLastSaleValue
  (
    @CustomerID INT
    , @Quantity INT = NULL
  )
RETURNS MONEY
AS
BEGIN
  DECLARE @LastSaleValue MONEY;

  SELECT @LastSaleValue = S.LineTotal
  FROM
  (
    SELECT
      (Quantity * Price) AS LineTotal
      , ROW_NUMBER() OVER (ORDER BY DateOfSale DESC) AS [Rank]
    FROM LearnSQLServerIntuitively.dbo.Sales
    WHERE
      CustomerID = @CustomerID
      AND Quantity >= ISNULL(@Quantity, Quantity)
  ) AS S
  WHERE S.[Rank] = 1;

  RETURN @LastSaleValue;
END
GO
```

SO (not **DSO**) naming convention

Function **Parameter(s) List**

NULL is the **default** value

The **scalar-valued function** returns a **single value** of a **specified data type**. Same as the data type of the **return variable**

The first statement declares the **return variable** that the **function returns**

One or multiple **statements** to **assign value** to the **return variable**

The function definition is between BEGIN and END

The last statement **returns** the **return variable**

Table-valued (Inline):

```
CREATE FUNCTION dbo.udf_SalesOverValueInline (@SaleValue MONEY)
RETURNS TABLE
AS
RETURN
(
```

Parameter(s) List

```
  WITH CTE_SumLineTotal (CustomerID) AS
    (
      SELECT CustomerID
      FROM LearnSQLServerIntuitively.dbo.Sales
      GROUP BY CustomerID
      HAVING SUM(Quantity * Price) >= @SaleValue
    )
  SELECT
    C.FirstName
    , C.LastName
    , S.SaleDate
    , S.Quantity
    , S.Price
    , (S.Quantity * S.Price) AS LineTotal
  FROM
    LearnSQLServerIntuitively.dbo.Customers AS C
    JOIN LearnSQLServerIntuitively.dbo.Sales AS S
      ON C.CustomerID = S.CustomerID
    JOIN CTE_SumLineTotal AS X
      ON C.CustomerID = X.CustomerID
);
GO
```

Between the brackets (the function definition) we add a **single** (SELECT) **statement** that returns a recordset

Table-valued (Multi-Statement):

```
CREATE FUNCTION dbo.udf_SalesOverValueMultiStatement (@SaleValue MONEY)
RETURNS @SalesOverValue TABLE
  (
    FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , SaleDate DATETIME
    , Quantity INT
    , Price MONEY
    , LineTotal MONEY
  )
AS
BEGIN
  DECLARE @FilterCustomerID TABLE (CustomerID INT);

  INSERT @FilterCustomerID (CustomerID)
  SELECT CustomerID
```

Table variable (recordset), returned by the function

Column definition of the table variable

Statement No. 1

Statement No. 2

```
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY CustomerID
HAVING SUM(Quantity * Price) >= @SaleValue;

INSERT @SalesOverValue (FirstName, LastName, SaleDate, Quantity, Price, LineTotal)
SELECT
  C.FirstName
  , C.LastName
  , S.DateOfSale
  , S.Quantity
  , S.Price
  , (S.Quantity * S.Price) AS LineTotal
FROM
LearnSQLServerIntuitively.dbo.Customers AS C
JOIN LearnSQLServerIntuitively.dbo.Sales AS S
ON C.CustomerID = S.CustomerID
WHERE C.CustomerID IN (SELECT CustomerID FROM @FilterCustomerID);

RETURN;
END
GO
```

Statement No. 3

Populate the **return table variable**

Statement No. 4

Function definition (multiple statements to populate the return table)

The code (View, Function, Stored Procedure) deployed in production is **repeatable** - it is executed multiple times.

To avoid the error `Cannot drop the {ObjectType} '{ObjectName}', because it does not exist or you do not have permission.`, we add a cleanup statement (delete if exists) before the `CRAETE` statement.

Functions on page 227

`CREATE FUNCTION` limitations:

- Number of parameters: 2,100

Before creating an object, we delete an existing object with the same name (if any).

DROP on page 104

ALTER Statement

Alters an existing DB object.

ALTER DATABASE

Modify the DB name or the DB files (.mdf, .ldf):

```
ALTER DATABASE LearnSQLServerIntuitively
MODIFY Name = LearnSQLServerIntuitively_1;
GO
```

ALTER SCHEMA

Transfer DB objects from one schema to another.

```
ALTER SCHEMA DestinationSchemaName
TRANSFER SourceSchemaName.SourceObjectName;
GO
```

Destination schema

Source schema and DB object to transfer

ALTER TABLE

`ALTER TABLE` modifies a table design. The keyword that follows is:

- `ADD` - adds one or multiple new columns to an existing table
- `ALTER COLUMN` - changes the column definition of an existing column
- `DROP COLUMN` - deletes one or multiple existing columns

Columns

- `ADD CONSTRAINT` - adds new constraint
- `DROP` - deletes existing constraint

Constraints

ADD

```
ALTER TABLE LearnSQLServerIntuitively.dbo.Customers
ADD
  FirstName NVARCHAR(32)
  , LastName NVARCHAR(64)
  , Email VARCHAR(128);
GO
```

Customers

Custo-merID
1
2
3

Customers

Custo-merID	First-Name	Last-Name	Email
1	NULL	NULL	NULL
2	NULL	NULL	NULL
3	NULL	NULL	NULL

ALTER COLUMN

Sales

Price
5.3542
52.6234
203.2416

MONEY
NULL

```sql
ALTER TABLE LearnSQLServerIntuitively.dbo.Sales
ALTER COLUMN Price DECIMAL(18, 6) NOT NULL;
GO
```

Customers

Price
5.354200
52.623400
203.241600

DECIMAL(18, 6)
NOT NULL

DROP COLUMN

```sql
ALTER TABLE LearnSQLServerIntuitively.dbo.Customers
DROP COLUMN
  LastName
  , Email;
GO
```

Customers

Custo-merID	First-Name	Last-Name	Email
1	Anabel	Larson	anabel.larson@ustomer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL

Customers

Custo-merID	First-Name
1	Anabel
2	Anna
3	Beverly

ADD CONSTRAINT

Adds constraint on a column for values that will be inserted in the future. Doesn't affect current values that don't violate the constraint. If the column already contains values, these values are then validated against the constraint to be created and in case of violation, the constraint can't be created.

Customers

First-Name	Last-Name	Email	DateReg-istered
Anabel	Larson	anabel.larson@customer5.info	1967-12-04
Anna	Laurier	anna.laurier@customer2.net	1966-05-17
Beverly	NULL	NULL	NULL

```
ALTER TABLE LearnSQLServerIntuitively.dbo.Customers
ADD CONSTRAINT DF__DateRegistered__GetDate
  DEFAULT (GETDATE())
  FOR DateRegistered;
GO
```

Insert new row, by not specifying value for **DateRegistered**

Customers

First-Name	Last-Name	Email	DateReg-istered
Anabel	Larson	anabel.larson@customer5.info	1967-12-04
Anna	Laurier	anna.laurier@customer2.net	1966-05-17
Beverly	NULL	NULL	NULL
John	Smith	john.smith@customer1.com	1969-11-07

Not affected by **DF**

The **date** when the row is **inserted**, generated by the **DF**

DROP (DROP CONSTRAINT)

Deletes an existing constraint.

Customers

First-Name	Last-Name	Email	DateReg-istered
Anabel	Larson	anabel.larson@customer5.info	1967-12-04
Anna	Laurier	anna.laurier@customer2.net	1966-05-17
Beverly	NULL	NULL	NULL
John	Smith	john.smith@customer1.com	1969-11-07

```
ALTER TABLE LearnSQLServerIntuitively.dbo.Customers
DROP DF__DateRegistered__GetDate;
GO
```

The keyword CONSTRAINT can be omitted

or

```
ALTER TABLE LearnSQLServerIntuitively.dbo.Customers
DROP CONSTRAINT DF__DateRegistered__GetDate;
GO
```

Insert new row, by not specifying value for **DateRegistered**

Customers

First-Name	Last-Name	Email	DateReg-istered
Anabel	Larson	anabel.larson@customer5.info	1967-12-04
Anna	Laurier	anna.laurier@customer2.net	1966-05-17
Beverly	NULL	NULL	NULL
John	Smith	john.smith@customer1.com	1969-11-07
Barnie	Jason	bj@customer6.br	NULL

No **DF**

ALTER VIEW

Changes the definition (the code) of an existing view. As the `ALTER VIEW` statement overwrites the definition of the view, we need first to create a backup of the view.

```
ALTER VIEW [SchemaName].[ViewName]
AS
  ...SELECT statement that will overwrite the existing definition of the view...;
GO
```

When we change the design of an underlying object, we need to refresh the view:

```
EXEC sp_refreshview '[SchemaName].[ViewName]';
```

ALTER PROC (ALTER PROCEDURE)

Edits the definition (and the parameters) of an existing Stored Procedure. First we back up the Stored Procedure.

```
ALTER PROC [SchemaName].[StoredProcedureName]
  (
    @Paramater1 INT                    <------ Edit the parameters list
    , @Parameter2 DECIMAL(10, 2)
    , @Parameter3 MONEY
  )
AS
BEGIN
  ... Statement 1...
  ... Statement 2...
  ...                                  <------ Edit the definition
  ... Statement n...;
END
GO
```

ALTER FUNCTION

Edits the definition (and the parameters) of an existing Function. We back up the function before altering it.

Scalar-valued

```
ALTER FUNCTION [SchemaName].[FunctionName]
  (
    @Parameter1 NVARCHAR(16)           <------ Edit the parameters list
    , @Parameter2 SMALLINT = NULL      <------ NULL is the default value
  )
```

```
RETURNS DATETIME
AS
BEGIN
  DECLARE @ReturnVariable DATETIME;

  SELECT @ReturnVariable = ...Statement that populates @ReturnVariable...;

  RETURN @ReturnVariable;
END
```

Edit the **definition**

Table-valued Inline

```
ALTER FUNCTION [SchemaName].[FunctionName] (@Parameter1 MONEY)
RETURNS TABLE
AS
RETURN
(
  SELECT Statement...
);
GO
```

Edit the **parameters** list

Edit the **definition**

Table-valued Multi-Statement

```
ALTER FUNCTION [SchemaName].[FunctionName]
  (
    @Parameter1 INT
    , @Parameter2 MONEY = NULL
  )
RETURNS @ReturnTable TABLE
  (
    Column1 NVARCHAR(32)
    , Column2 NVARCHAR(64)
  )
AS
BEGIN
  ...Statement 1...
  ...Statement 2...

  INSERT @ReturnTable ...Statement 3...

  RETURN (0);
END
GO
```

Edit the **parameters** list

NULL is the **default** value

Edit the **returned table**

Edit the **definition**

DROP Statement

Deletes an existing DB object.

DROP DATABASE

Deletes an existing database.

```
USE [master];
GO

DROP DATABASE LearnSQLServerIntuitively;
GO
```

If we execute the statement above and the DB doesn't exist, an error message "`Cannot drop the database 'LearnSQLServerIntuitively', because it does not exist or you do not have permission.`" is returned and the execution of the batch stops. To avoid this, we wrap the execution of the `DROP` with the `IF` condition:

```
USE [master];
GO

DROP DATABASE IF EXISTS LearnSQLServerIntuitively;
GO
```

From SQL Server 2016

```
USE [master];
GO

IF (DB_ID('LearnSQLServerIntuitively') IS NOT NULL)
  BEGIN
    ALTER DATABASE LearnSQLServerIntuitively
      SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
    DROP DATABASE LearnSQLServerIntuitively;
  END
GO
```

Validate if **DB exists** with the built-in function `DB_ID()`

Before SQL Server 2016

or

```
IF EXISTS
(
  SELECT [name]
  FROM [master].sys.databases
```

Validate if **DB exists** with `IF EXISTS` in **Object Catalog**

```
  WHERE [name] = 'LearnSQLServerIntuitively'
)
  BEGIN
    ALTER DATABASE LearnSQLServerIntuitively SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
    DROP DATABASE LearnSQLServerIntuitively;
  END
GO
```

Before deleting a DB, we delete the backup history for the DB:

```
EXEC msdb.dbo.sp_delete_database_backuphistory
  @database_name = N'LearnSQLServerIntuitively';
GO
```

DROP SCHEMA

Deletes an existing schema.

```
USE LearnSQLServerIntuitively;
GO

DROP SCHEMA Sales;
GO
```

Execute DROP SCHEMA with the IF condition to prevent the code termination in case the schema doesn't exist:

```
DROP SCHEMA IF EXISTS Sales;
GO
```

From SQL Server 2016

```
USE LearnSQLServerIntuitively;
GO

IF (SCHEMA_ID('Sales') IS NOT NULL)
  BEGIN DROP SCHEMA Sales; END
GO
```

Validate if **schema exists** with the built-in function **SCHEMA_ID()**

Before SQL Server 2016

or

```
IF EXISTS
(
  SELECT [name]
  FROM LearnSQLServerIntuitively.sys.schemas
  WHERE [name] = 'Sales'
)
  BEGIN DROP SCHEMA Sales; END
GO
```

Validate if **schema exists** with IF EXISTS in **Object Catalog**

We can't delete schema that contains DB object(s). The error message "`Cannot drop schema 'Sales' because it is being referenced by object 'SalesDetails'.`" is returned.

DROP TABLE

Deletes an existing table.

```
DROP TABLE IF EXISTS LearnSQLServerIntuitively.dbo.Customers;
GO

IF (OBJECT_ID('LearnSQLServerIntuitively.dbo.Customers', 'U') IS NOT NULL)
  BEGIN DROP TABLE LearnSQLServerIntuitively.dbo.Customers; END
GO
```

From SQL Server 2016

Validate if the **table exists** with the built-in function **OBJECT_ID()**

or

Before SQL Server 2016

```
IF EXISTS
(
  SELECT [name]
  FROM LearnSQLServerIntuitively.sys.tables
  WHERE [name] = 'Customers'
)
  BEGIN DROP TABLE LearnSQLServerIntuitively.dbo.Customers; END
GO
```

Validate if the **table exists** with IF EXISTS in **Object Catalog**

DROP CONSTRAINT

Deletes an existing constraint.

```
ALTER TABLE LearnSQLServerIntuitively.dbo.Sales
DROP CONSTRAINT IF EXISTS FK__Sales__CustomerID;
GO

IF EXISTS
(
  SELECT 1
  FROM LearnSQLServerIntuitively.sys.foreign_keys
  WHERE
    OBJECT_NAME([parent_object_id]) = 'Sales'
    AND [name] = 'FK__Sales__CustomerID'
)
  BEGIN
    ALTER TABLE LearnSQLServerIntuitively.dbo.Sales
```

From SQL Server 2016

Validate if **table exists** with IF EXISTS in **Object Catalog**

```
    DROP CONSTRAINT FK__Sales__CustomerID;
  END
GO
```

Constraints on page 78

DROP VIEW

Deletes an existing view.

```
USE LearnSQLServerIntuitively;
GO

DROP VIEW IF EXISTS dbo.vw_Sales;
GO
```

From SQL Server 2016

Validate if the **view exists** with the built-in function **OBJECT_ID()**

```
IF (OBJECT_ID('dbo.vw_Sales', 'V') IS NOT NULL)
  BEGIN DROP VIEW dbo.vw_Sales; END
GO
```

Before SQL Server 2016

or

```
IF EXISTS
(
  SELECT [name]
  FROM LearnSQLServerIntuitively.sys.views
  WHERE
    SCHEMA_NAME([schema_id]) = 'dbo'
    AND [name] = 'vw_Sales'
)
BEGIN DROP VIEW dbo.vw_Sales; END
GO
```

Validate if the **view exists** with IF EXISTS in **Object Catalog**

DROP PROC (DROP PROCEDURE)

Deletes an existing stored procedure.

```
USE LearnSQLServerIntuitively;
GO

DROP PROC IF EXISTS dbo.usp_Customers_CRUD;
GO
```

From SQL Server 2016

```sql
IF (OBJECT_ID('dbo.usp_Customers_CRUD', 'P') IS NOT NULL)
  BEGIN DROP PROC dbo.usp_Customers_CRUD; END
GO
```

Validate if the **stored procedure exists** with the built-in function OBJECT_ID()

or

Before SQL Server 2016

Validate if the **stored procedure exists** with IF EXISTS in **Object Catalog**

```sql
IF EXISTS
(
  SELECT [name]
  FROM LearnSQLServerIntuitively.sys.procedures
  WHERE
    SCHEMA_NAME([schema_id]) = 'dbo'
    AND [name] = 'usp_Customers_CRUD'
)
BEGIN DROP PROC dbo.usp_Customers_CRUD; END
GO
```

DROP FUNCTION

Deletes an existing function.

```sql
USE LearnSQLServerIntuitively;
GO

DROP FUNCTION IF EXISTS dbo.udf_DateNoTime;
GO
```

From SQL Server 2016

```sql
IF (OBJECT_ID('dbo.udf_DateNoTime', 'FN') IS NOT NULL)
  BEGIN DROP FUNCTION dbo.udf_DateNoTime; END
GO
```

Validate if the **function exists** with the built-in function OBJECT_ID()

or

Before SQL Server 2016

Validate if the **function exists** with IF EXISTS in **Object Catalog**

```sql
IF EXISTS
(
  SELECT [name]
  FROM LearnSQLServerIntuitively.sys.objects
  WHERE
    [type] IN ('IF', 'FN', 'TF')
    AND SCHEMA_NAME([schema_id]) = 'dbo'
    AND [name] = 'udf_DateNoTime'
```

IF = SQL_INLINE_TABLE_VALUED_FUNCTION
FN = SQL_SCALAR_FUNCTION
TF = SQL_TABLE_VALUED_FUNCTION

```
)
BEGIN DROP FUNCTION dbo.udf_DateNoTime; END
GO
```

Object Type

As you noticed, the second parameter in the built-in function `OBJECT_ID()` is **object type**. We specify it to limit the function to search only for specific object type.

To view the object types in the **Object Catalog**:

```
SELECT DISTINCT
  [type] AS ObjectType
  , [type_desc] AS ObjectTypeDescription
FROM LearnSQLServerIntuitively.sys.all_objects
ORDER BY type_desc;
GO
```

DDL Statements
Script Objects in SSMS

Log in SQL Server Management Studio (SSMS).

SQL Server Management Studio (SSMS) on page 75

We execute **DDL** (**Data Definition Language**) **statements** to:

- CREATE
- ALTER
- DROP

DB objects.

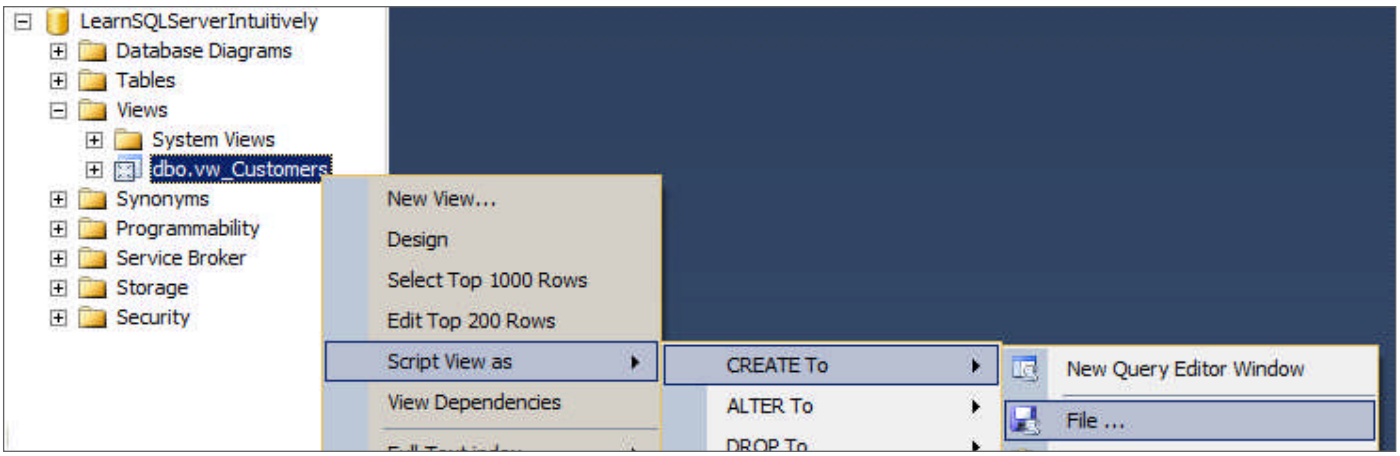
The DB objects, covered by this book are database, schema, table, view, stored procedure and function.

Scripting DB objects in SSMS

Create a backup of every DB object that you modify or delete.

View

In SSMS: Expand the Object Explorer to the View ⟶ (right click) **Script View as** ⟶ **CREATE To** (**ALTER To**) ⟶ **File...** (**New Query Editor Window**)



Stored Procedure

In SSMS: Expand the Object Explorer to the Stored Procedure (**Database** ⟶ **Programability** ⟶ **Stored Procedures**) ⟶ (right click) **Script Stored Procedure as** ⟶ **CREATE To** (**ALTER To**) ⟶ **File...** (**New Query Editor Window**)

In the same way we script the other objects.

DML (Data Manipulation Language) Statements

As the **DDL** statements are the ones that manage the DB objects, the **DML** statements are the ones that manipulate the data in the DB. They are represented by **CRUD** as follows:

- `INSERT` for **C** (Create)
- `SELECT` for **R** (Read)
- `UPDATE` for **U** (Update)
- `DELETE` for **D** (Delete)

The source of the data is/are the table(s), view(s), table-valued function(s).
T-SQL extracts data from these objects into virtual recordsets (virtual tables, created in the background, used to store data the T-SQL needs to manipulate) or tables (temporary or real) and manipulates it.
After all the manipulations are done, the resulting recordset is returned to the requester (SSMS or the application that executed the T-SQL code).

To create a report that shows the sales persons, eligible for commission (Threshold >= 10) for October 1968, ordered by name, we extract the data from multiple sources:

Sales (table)

SalesPer-sonID	Period	Sales-Value
1	1968-10	463.27
3	1967-01	196.52
3	1968-10	16.23
4	1969-02	113.2
3	1968-10	5324.87

SalesPersons (view)

SalesPer-sonID	First-Name	Last-Name
1	John	Smith
2	Ashley	Larson
3	Brandon	Fuston
4	Jude	Audrey
5	Michael	Stolknom
6	Peter	Bernier

CommissionsTreshholds (table-valued function)

SalesPer-sonID	Threshold-Value
1	11.35
2	1.52
3	56.12
4	5.2
5	5.34
6	452.34
7	753.2
8	18
9	0.563

`Data sources`

into virtual recordset (VR):

SalesPer-sonID	Period	Sales-Value	First-Name	Last-Name	Threshold-Value
1	1968-10	463.27	John	Smith	11.350
3	1967-01	196.52	Brandon	Fuston	56.120
3	1968-10	16.23	Brandon	Fuston	56.120
3	1968-10	5324.87	Brandon	Fuston	56.120
4	1969-02	113.20	Jude	Audrey	5.200

`Virtual recordset (result of JOINed sources)`

and manipulate the virtual recordset to build the resulting recordset, used for the report:

Sales-Value	First-Name	Last-Name	Threshold-Value
463.27	John	Smith	11.350
16.23	Brandon	Fuston	56.120
5324.87	Brandon	Fuston	56.120

`Transform the dataset to report`

Commission Program Sales for October 1968

SalesPerson	Sales (USD)
Brandon Fuston	5341.10
John Smith	463.27

The recordsets are created and used in background. After all the manipulations are done and the data is returned to the requester, the virtual recordsets are deleted.

The example above summarizes the material that we'll learn in **DML Statements**.

INSERT

Inserts a new row or recordset (multiple rows) in an existing table.

The syntax is:

```
INSERT INTO Identifier
...Statement that creates the recordset to be inserted...;
GO
```

Insert values in all the columns

or

```
INSERT INTO LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName
    , LastName
    , Email
    ,...
  )
...Statement that creates the recordset to be inserted...;
GO
```

Insert values in the listed columns

The keyword INTO can be omitted:

```
INSERT LearnSQLServerIntuitively.dbo.Customers
  (
    FirstName
    , LastName
    , Email
    , ...
)
...Statement that creates the recordset to be inserted...;
GO
```

Customers

FirstName	LastName	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	NULL	NULL
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org

Existing table into which we insert rows

INSERT… SELECT

This example Inserts one row. The column list is not specified so we provide values for all the columns in the table in the same order as in the table structure:

```
INSERT LearnSQLServerIntuitively.dbo.Customers
SELECT
  'Melanie'                                   ← Insert in the first column
  , 'Larson'                                  ← Insert in the second column
  , 'melanie.larson@customer4.biz';           ← Insert in the third column
GO
```

The next example Insert multiple rows and specifies the column list. The UNION clause unions two recordsets into one:

```
INSERT LearnSQLServerIntuitively.dbo.Customers (LastName, Email, FirstName)   ← Row 1
SELECT 'Larson', 'melanie.larson@customer4.biz', 'Melanie'                    ← Row 2
UNION ALL SELECT 'Dernier', 'gdernier@customer412.be', 'George';
GO
```

The order of the columns in the column list must match the order of the columns in the inserted recordset

INSERT… SELECT… FROM

We can SELECT rows from one data source and INSERT them into existing table:

```
INSERT LearnSQLServerIntuitively.dbo.Customers
  (
    LastName
    , FirstName
    , Email
  )
SELECT
  LastName
  , FirstName
  , Email
FROM [Another Database].[Another Schema].Customers
GO
```

The number of the inserted rows has to match the number of the rows in the recordset, produced by the SELECT statement

SELECT... FROM creates recordset from rows in existing object (table, view or table-valued function)

INSERT… VALUES

The VALUES clause creates a recordset with one row:

```
INSERT LearnSQLServerIntuitively.dbo.Customers
VALUES ('Jonathan', 'Grenier', 'jg@jgcompanyandassocies.ca');
GO
```

To create recordset with multiple rows, we delimit them with comma (,):

```
INSERT LearnSQLServerIntuitively.dbo.Customers (Email, LastName, FirstName)
VALUES
  ('Michaels.Zoe@customer18.se', 'Michaels', 'Zoe')
  , ('t_robertson@customer19.at', 'Robertson', 'Thomas')
  , ('af@customer20.fr', 'Frances', 'Albert');
GO
```

INSERT… EXEC

Inserts a recordset, created by Stored Procedure.

```
INSERT LearnSQLServerIntuitively.dbo.Customers (FirstName, LastName, Email)
EXEC LearnSQLServerIntuitively.dbo.usp_InsertCustomer;
GO
```

Stored Procedures on page 240

The columns, not specified in the column list of the INSERT clause will be populated with NULL or their default values (if default constraint is specified for the columns).
The constraints are checked on INSERT and if there is a violation in a constraint, the row is not inserted.

We specify the names of the columns that we insert into. If we don't, a change of the design of the object may break the code. For example if we insert into a table with 4 columns and we didn't specify the column list, adding a 5-th column to the table will cause the INSERT statement to return an error 'Column name or number of supplied values does not match table definition'.

INSERT limitations

- Number of columns: 4,096

SELECT... INTO

SELECT... INTO combines **DDL** and **DML** clauses into one statement. It SELECTs data from an existing object (table, view or table-valued function) and inserts the resulting recordset into a new table, created during the execution of the SELECT... INTO statement.

The syntax is:

```
SELECT Column1, Column2, Column3...
INTO [DatabaseName].[SchemaName].[DestinationTableName]
FROM [DatabaseName].[SchemaName].[SourceTableName];
GO
```

If the new table, specified in the INTO clause, already exists, the execution of the statement will return an error "There is already an object named 'DestinationTableName' in the database." and will be terminated. To avoid this, we run a cleanup statement first (If DestinationTableName exists, delete it).

```
DROP TABLE IF EXISTS LearnSQLServerIntuitively.dbo.Customers_1;
GO
```

Clean up

```
SELECT FirstName, LastName, Email
INTO LearnSQLServerIntuitively.dbo.Customers_1
FROM LearnSQLServerIntuitively.dbo.Customers
GO
```

Table Customers_1 is created during the execution of the statement

The columns in the destination table inherits the data types of the columns in the recordset.

Customers

First-Name	Last-Name	Email
John	Smith	john.smith@customer3.org
Melanie	Larson	melanie.larson@customer4.biz
Xavier	Jameson	xavier.j@customer6.com

NVARCHAR(32) NVARCHAR(64) VARCHAR(128)

NVARCHAR(97)
32 + space + 64 = 97

Customers_1

Name	Email
John Smith	john.smith@customer3.org
Melanie Larson	melanie.larson@customer4.biz
Xavier Jameson	xavier.garcía@customer6.com

```
SELECT
  CONCAT(FirstName, ' ', LastName) AS [Name]
  , Email
INTO LearnSQLServerIntuitively.dbo.Customers_1
FROM LearnSQLServerIntuitively.dbo.Customers;
GO
```

In the example below, the data type of the **TotalValue** column in the source table is TINYINT (Value: 0 to 255). The query sums the column **TotalValue** and the summed value for CustomerID = 1 (280) extends the range of TINYINT (255). The data type of **TotalValue** in the destination table is converted to INT.

Sales

CustomerID	ItemID	TotalValue
1	3	130
2	6	37
1	3	150
INT	INT	TINYINT

```
SELECT
  CustomerID
  , ItemID
  , SUM(TotalValue) AS Sum_TotalValue
INTO LearnSQLServerIntuitively.dbo.Sales_1
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY
  CustomerID
  , ItemID;
GO
```

Sales

CustomerID	ItemID	TotalValue
1	3	280
2	6	37
INT	INT	INT

If a column in the SELECT clause has no name (if we omit the alias AS Sum_TotalValue in the example above), the DBE will return an error "An object or column name is missing or empty. For SELECT INTO statements, verify each column has a name. For other statements, look for empty alias names. Aliases defined as "" or [] are not allowed. Change the alias to a valid name.".

DML Statements (Modify)
UPDATE

Updates the value(s) of one or multiple cells in a table.

The syntax is:

```
UPDATE DatabaseName.SchemaName.UpdatedObjectName
SET UpdatedColumnName = NewValue;
GO
```

Table, view or table-valued function

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
6	Melanie	Larson	larson_mel@customer4.biz
7	Xavier	Jameson	xj@customer6.com
8	Zak	Smith	NULL

UPDATE... SET

Updates all the rows of the updated column.

Updates all the rows in column **Email**

```
UPDATE LearnSQLServerIntuitively.dbo.Customers
SET Email = 'customer84186@customer84186.pl';
GO
```

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	customer84186@customer84186.pl
2	Anna	Laurier	customer84186@customer84186.pl
3	Beverly	NULL	customer84186@customer84186.pl
4	John	Smith	customer84186@customer84186.pl
5	John	Smith	customer84186@customer84186.pl
6	Melanie	Larson	customer84186@customer84186.pl
7	Xavier	Jameson	customer84186@customer84186.pl
8	Zak	Smith	customer84186@customer84186.pl

UPDATE... SET... WHERE

Affects only the rows that correspond to the filter condition(s) in the `WHERE` clause.

```
UPDATE LearnSQLServerIntuitively.dbo.Customers
SET Email = 'customer84186@customer84186.pl'
WHERE
  FirstName = 'Anabel'
  AND LastName = 'Larson';
GO
```

Customers

Custo-merID	First-Name	Last-Name	Email	FirstName = 'Anabel'?		LastName = 'Larson'?
1	Anabel	Larson	anabel.larson@customer5.info	Yes	AND	Yes
2	Anna	Laurier	anna.laurier@customer2.net	No	AND	No
3	Beverly	NULL	NULL	No	AND	No
4	John	Smith	john.smith@customer1.com	No	AND	No
5	John	Smith	john.smith@customer3.org	No	AND	No
6	Melanie	Larson	larson_mel@customer4.biz	No	AND	Yes
7	Xavier	Jameson	xj@customer6.com	No	AND	No
8	Zak	Smith	NULL	No	AND	No

Logical Operator

Email
customer84186@customer84186.pl

New value

UPDATE... SET... FROM... WHERE

Updates cell values, based on a resultset, created from multiple DB objects.

The following tables are used to demonstrate an UPDATE... SET... FROM... WHERE example:

Customer

CustomerID	First-Name	Last-Name	Email	IsEligibleFor-Discount	IsLowSales-Value	DateLastUpdated
1	Anabel	Larson	anabel.larson@customer5.info	NULL	NULL	NULL
2	Anna	Laurier	anna.laurier@customer2.net	NULL	NULL	NULL
3	Beverly	NULL	NULL	NULL	NULL	NULL
4	John	Smith	john.smith@customer1.com	NULL	NULL	NULL
5	John	Smith	john.smith@customer3.org	NULL	NULL	NULL
6	Melanie	Larson	larson_mel@customer4.biz	NULL	NULL	NULL
7	Xavier	Jameson	xj@customer6.com	NULL	NULL	NULL
8	Zak	Smith	NULL	NULL	NULL	NULL

PK

Sales

SaleID	CustomerID	ItemID	DateOfSale	Quantity	Price	LineTotal
1	7	345	1967-05-13	512	5.26	2693.12
2	1	58	1965-01-12	23	5.22	120.06
3	2	173	1965-12-14	1	5.231	5.231
4	5	1542	1968-08-15	55	66.231	3642.705
5	3	58	1967-11-28	276	5.24	1446.24
6	4	234	1964-04-07	32	126.24	4039.68
7	2	6562	1968-10-01	27	4638.02	125226.54
8	5	44	1968-03-28	127	1.24	157.48
9	2	58	1967-12-03	65	5.24	340.6
10	5	727	1965-08-17	52	5.21	270.92

FK Today is December 18, 1967

ItemID = 58

DateOfSale in the last 30 days (including today)

This example UPDATEs the value of **IsEligibleForDiscount** for the customers that bought **ItemID** = 58 in the last 30 days (including today):

```
UPDATE C
SET
  IsEligibleForDiscount = 1
  , DateLastUpdated = GETUTCDATE()
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
WHERE
  S.ItemID = 58
  AND S.DateOfSale BETWEEN DATEADD(DD, -30, CAST(GETDATE() AS DATE)) AND CAST(GETDATE() AS
DATE);
GO
```

Alias

Customers table after the execution on December 18, 1967:

CustomerID	First-Name	Last-Name	Email	IsEligibleFor-Discount	IsLowSales-Value	DateLastUpdated
1	Anabel	Larson	anabel.larson@customer5.info	NULL	NULL	NULL
2	Anna	Laurier	anna.laurier@customer2.net	1	NULL	1967-12-18
3	Beverly	NULL	NULL	1	NULL	1967-12-18
4	John	Smith	john.smith@customer1.com	NULL	NULL	NULL
5	John	Smith	john.smith@customer3.org	NULL	NULL	NULL
6	Melanie	Larson	larson_mel@customer4.biz	NULL	NULL	NULL
7	Xavier	Jameson	xj@customer6.com	NULL	NULL	NULL
8	Zak	Smith	NULL	NULL	NULL	NULL

UPDATE... SET... FROM (Subquery)... WHERE

We can't use an aggregate built-in function in the SET clause.

The example below UPDATEs columns **IsLowSalesValue** and **DateLastUpdated** (in table **Customers**) with the values from the returned by the subquery (**SQ**) recordset (**ILSV** and **DEFDU**), on the rows where the value of **CustomerID** column equals the value **CustID** in the subquery (**SQ**):

```
UPDATE LearnSQLServerIntuitively.dbo.Customers
SET
  IsLowSalesValue = SQ.ILSV
  , DateLastUpdated = SQ.DEFDU
FROM
  (
    SELECT
      C.CustomerID AS CustID
      , 1 AS ILSV
      , GETUTCDATE() AS DEFDU
    FROM
      LearnSQLServerIntuitively.dbo.Customers AS C
      JOIN LearnSQLServerIntuitively.dbo.Sales AS S
        ON C.CustomerID = S.CustomerID
    WHERE S.ItemID = 58
    GROUP BY C.CustomerID
    HAVING SUM(S.LineTotal) < 200
  ) AS SQ
WHERE  CustomerID = SQ.CustID;
GO
```

The subquery (between the brackets) returns this recordset

CustID	ILSV	DEFDU
1	1	1967-12-18

Customers table after the execution (`CustomerID` = 1 is `UPDATE`d):

CustomerID	First-Name	Last-Name	Email	IsEligibleFor-Discount	IsLowSales-Value	DateLastUpdated
1	Anabel	Larson	anabel.larson@customer5.info	NULL	1	1967-12-18
2	Anna	Laurier	anna.laurier@customer2.net	1	NULL	1967-12-18
3	Beverly	NULL	NULL	1	NULL	1967-12-18
4	John	Smith	john.smith@customer1.com	NULL	NULL	NULL
5	John	Smith	john.smith@customer3.org	NULL	NULL	NULL
6	Melanie	Larson	larson_mel@customer4.biz	NULL	NULL	NULL
7	Xavier	Jameson	xj@customer6.com	NULL	NULL	NULL
8	Zak	Smith	NULL	NULL	NULL	NULL

`SELECT` on page 125
`FROM` on page 127
`WHERE` on page 142
`GROUP BY` on page 151
`HAVING` on page 154

DELETE FROM... (DELETE)

Deletes one or multiple rows from a table.

```
DELETE FROM TableName;
GO
```

The keyword FROM is optional and can be omitted

DELETE statement without WHERE clause deletes all the rows in the table.

```
DELETE [DatabaseName].[SchemaName].[TableName]
WHERE ColumnName Operator Value;
GO
```

Deletes the rows that correspond to the filter condition(s)

We need to pay special attention to the DELETE statements and:

- Backup the data before executing the DELETE statement
- Test the DELETE statement(s) on objects located on the development server and deploy on the production server, after being sure that we deleted exactly the data that we need

DELETE FROM... FROM...

When we join tables and delete rows from one of the joined tables, we specify from which table we delete rows.

We can identify the rows that we need to delete by writing a query that JOINs multiple DB objects. In the example below, we JOIN the tables **Customers** and **Consolas** to pick the right rows to delete from table **Sales**.

Customer

CustomerID	First-Name	Last-Name	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL

FROM can be omitted

```
DELETE FROM C
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
WHERE S.DateOfSale <= '1967-01-01';
GO
```

Alias

PK

FK

The rows of **CustomerID** 1 and 2 are deleted from table **Customers**

Sales

SaleID	CustomerID	ItemID	DateOfSale	Quantity	Price	LineTotal
1	2	345	1967-05-13	512	5.26	2693.12
2	1	58	1965-01-12	23	5.22	120.06
3	2	173	1965-12-14	1	5.231	5.231

Translation: Create a recordset by joining both tables and delete from **Customers** the customers which sales are older than a specified date.

DateOfSale older than January 01, 1967

DML Statements (Modify)
DELETE

Before we execute the DELETE or UPDATE statements, we execute the same statement with the SELECT clause, instead of the DELETE or UPDATE clause to verify the rows to be deleted/updated.

Execute first:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE FirstName = 'John';
GO
```

After you confirm that the statement affects the exact rows, execute:

```
DELETE LearnSQLServerIntuitively.dbo.Customers
WHERE FirstName =  'John';
GO

UPDATE LearnSQLServerIntuitively.dbo.Customers
SET Email = 'j@customer2.net'
WHERE FirstName = 'John';
GO
```

SELECT... **...or create a recordset**

Creates a recordset and returns it to the requester (SSMS, external application that queries the DB etc.).
The SELECT clause can be constructed from a single SELECT clause or multiple clauses - SELECT... FROM... WHERE...

Manipulates What	T-SQL	Data Type	Returned Value or Recordset
Value	`SELECT 'John';`	String (in single quotes)	John
	`SELECT 123;`	Numeric	123
Expression	`SELECT ('John' + ' ' + 'Smith');`	String and String	John Smith
	`SELECT (123 + 456);`	Numeric and Numeric	579
	`SELECT ('John' + ' ' + CAST(123 AS VARCHAR));`	String and Numeric, explicitly converted to String	John 123
	`SELECT (Price * 1.3)` `FROM DBObjectName`	Column value and Numeric	3.9 *(Price = 3)*
	`SELECT SellPrice - Expences` `FROM DBObjectName`	Column value and column value	7.18 *(SellPrice = 9.86* *Expences = 2.68)*
Built-in function	`SELECT GETDATE();`	Output: Date and Time	1965-06-05 13:33:23.390 *(now)*
	`SELECT LEN('John');`	Input: String Output: Numeric	4
User-defined function	`SELECT dbo.udf_GetFirstDayOfCurrentFiscalMonth();`	Function output data type	1968-04-13 00:00:00.000

Value, expression, user-defined function and retutns recordset (rows and columns)	`SELECT`
	`  'John'' AS FirstName`
	`  , 'Smith' AS LastName;`
		Both columns: String	FirstName: John
			LastName: Smith

	`SELECT`
	`  'John' AS FirstName`
	`  , (100 + 23) AS NumericValue;`
		First column: String
		Second column: Numeric
			FirstName: John
			NumericValue: 123

	`SELECT`
	`  'John' AS FirstName`
	`  , 123 AS NumericValue`
	`  , dbo.udf_GetFirstDayOfCurrentFiscalMonth() AS UDFResult;`
		First column: String
		Second column: Numeric
		Third column: Function output data type
			FirstName: John
			NumericValue: 123
			UDFResult: 1968-04-13 00:00:00.000

Variable *	`SELECT @FirstName AS VariableValue;`	The data type of the variable	John *(@FirstName = John)*

* Variables on page 199

SELECT DISTINCT...

The keyword DISTINCT, added to the SELECT statement, changes the logic to select only distinct (unique) rows.

The rows in table **Customers** are unique (the combination of all the columns are unique).

Customers

First-Name	Last-Name	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	NULL	NULL
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org

When we select columns **FirstName** and **LastName**, the SELECT statement generates a recordset, that is not unique (John Smith is a duplicate).

```
SELECT
  FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.Customers;
GO
```

First-Name	Last-Name
Anabel	Larson
Anna	Laurier
Beverly	NULL
John	Smith
John	Smith

To generate a recordset with unique rows across the **FirstName** and **LastName** rows, we add the keyword DISTINCT to the SELECT clause:

```
SELECT DISTINCT
  FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.Customers;
GO
```

First-Name	Last-Name
Anabel	Larson
Anna	Laurier
Beverly	NULL
John	Smith

SELECT limitations

- Number of columns in the SELECT statement: 4,096

SELECT... FROM

...or point to the source of the data

The FROM clause points to the DB object (table - real or temporary, view, table-values function) or subquery which is a data source for the SELECT clause.

The syntax is:

```
SELECT [Column1], [Column2], [Column3]...
FROM [DatabaseName].[SchemaName].[ObjectName];
GO
```

Column list, delimited with comma (,). Square brackets if the column header contains special character(s).

Table, view or table-valued function.

When we select all the columns, we use SELECT * (star). If we need specific columns, we SELECT a list of columns, delimited with comma (,).

Customer

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL

```
SELECT FirstName, LastName
FROM Customers;
GO
```

* means **all the columns**

```
SELECT *
FROM Customers;
GO
```

Table **Customers** is the source of the data, manipulated by the statement

Subquery

The subquery is a nested SELECT statement (inner) into another SELECT statement (outer).
When we have a subquery in the FROM clause the inner SELECT statement creates a recorsdet and the outer SELECT statement selects from this recordset.

Subqueries on page 165

1. List the columns in the SELECT clause instead of using * (star), because:
   1.1. After a change of the design of the data source the query may break.
   1.2. * Selects columns that we don't need and creates an overload of the server.
2. Use the **DSO** ([DatabaseName].[SchemaName].[ObjectName]) naming convention as identifier in the FROM clause. This way we can move the SQL code to another database in the same instance on the SQL Server without editing it in order to point to the exact object.

SELECT... FROM... JOIN

...or create a recordset from multiple data sources

The data in the **RDBMS** are related and it is in multiple tables split by subject area. Joining related tables is one of the most important and powerful tasks in SQL.
When we select data from multiple data sources (DS) - table, view, table-valued function or subquery - we JOIN them.
The joined objects create a **virtual recordset** (**VR**) in the background that holds all data from the data sources and we query this **VR**.
We JOIN the objects ON one or multiple columns.

The different types of JOIN, create different **virtual recordsets**:

JOIN **type**	**creates a VR that contains**	**and**	**based on the JOIN conditions in the FROM clause**
INNER JOIN	the **matching** rows in **both** DS	→	Yes
LEFT OUTER JOIN	**all** the rows from the **left** DS	the **matching** rows from the **right** DS	Yes
RIGHT OUTER JOIN	**all** the rows from the **right** DS	the **matching** rows from the **left** DS	Yes
FULL OUTER JOIN	**all** the rows from **both** DS	→	Yes
CROSS JOIN	**all** the **combinations** of the rows in **both** DS	→	No ON clause

DS - data source

Aliases on page 138

The syntax is:

```
SELECT
  L.Col2
  , R.Col3
  , ...
FROM
  LeftDataSource AS L
  (INNER, LEFT, RIGHT, FULL) JOIN RightDataSource AS R
    ON L.Column1 = R.Column1
    AND
    (
      L.Column2 = R.Column2
      OR L.Column3 >= R.Column3
    )
    AND R.Column4 > 123
    AND ...;
GO
```

Alias in the SELECT clause (bind alias)

Keyword (**type of joining**), JOIN, **right data source** and **alias**

Left data source and **alias** in the FROM clause (assign alias)

After the ON clause, we list the JOIN **condition(s)** (link column(s))

When we use both AND and OR logical operators, we separate them with **brackets**

Comparison operators

Filter the **VR** with **value** or **expression**

Multiple JOIN conditions joined with logical operators (AND, OR)

Let's join the two sources below. For the simplicity of the example, the test data is:

- **Col1** is used to join the tables and values that exist in both tables such as **Col1** (3,4 and 5)
- The values in **Col2** and **Col3** are: first character is the column index and the second character is the value in **Col1**

Left data source

Col1	Col2	Col3
1	21	31
2	22	32
3	23	33
4	24	34
5	25	35

Right data source

Col1	Col2	Col3
3	23	33
4	24	34
5	25	35
6	26	36
7	27	37

Joining the sources on column **Col1**, creates the following **VR**:

JOINed data sources					
Left data source			Right data source		
Col1	Col2	Col3	Col1	Col2	Col3
1	21	31			
2	22	32			
3	23	33	3	23	33
4	24	34	4	24	34
5	25	35	5	25	35
			6	26	36
			7	27	37

```
ON L.Col1 = R.Col1
```

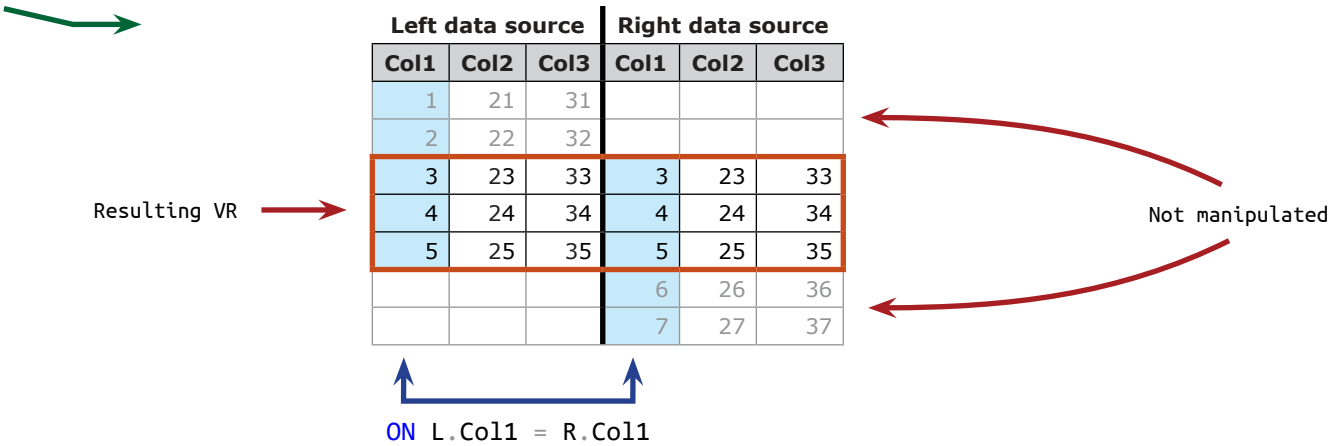
The statement that manipulates the joined tables is actually manipulating the recordset, built with the FROM... JOIN clause.

JOIN (INNER JOIN)

INNER JOIN is inner, because it creates a **VR** that contains **only the rows that meet the join criteria(s)**.

```
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON L.Col1 = R.Col1;
GO
```

The keyword INNER is optional and can be omitted

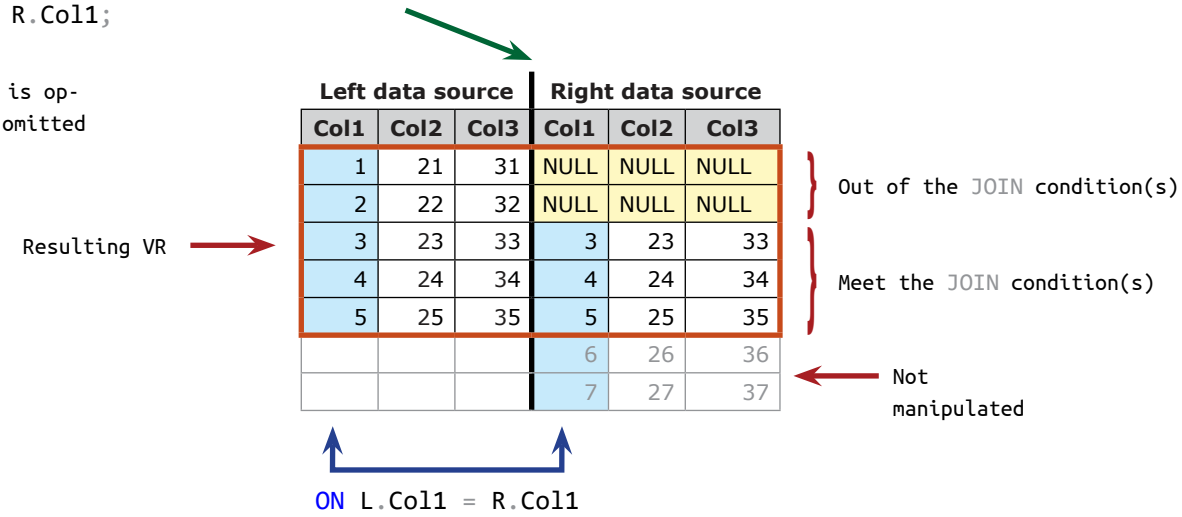


LEFT JOIN (LEFT OUTER JOIN)

OUTER JOIN is outer, because it creates VR, that contains more rows than the rows that meet the JOIN condition(s) in the ON clause.
LEFT JOIN creates VR with all the rows from the left data source and only the rows from the right data source that meet the JOIN condition(s).
The **VR** fills the rows in the right table that don't have a match on the JOIN condition(s) with NULL.

```
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  LEFT JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON L.Col1 = R.Col1;
GO
```

The keyword OUTER is optional and can be omitted



When we use the LEFT or RIGHT JOIN and filter in the WHERE clause, the JOIN type is converted from LEFT (RIGHT) to INNER.
To avoid this and filter only the LEFT (RIGHT) data source, we filter in the ON clause.

```
SELECT
  T1.A
  , T1.B
  , T2.C
FROM
  Table1 AS T1
  LEFT JOIN Table2 AS T2
    ON T1. Column1 = T2.Column1
    AND T2.Column 2 = @Parameter1
WHERE T2.Column 2 = @Parameter1;
GO
```

Filter here

If we filter on the WHERE clause, the JOIN is converted from LEFT (RIGHT) to INNER

RIGHT JOIN (RIGHT OUTER JOIN)

The same logic as LEFT JOIN, but reversed - the **VR** contains **all the rows from the right data source** and **the matching rows from the left data source**.

```
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  RIGHT JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON L.Col1 = R.Col1;
GO
```

Not manipulated

Left data source			Right data source		
Col1	Col2	Col3	Col1	Col2	Col3
1	21	31			
2	22	32			
3	23	33	3	23	33
4	24	34	4	24	34
5	25	35	5	25	35
NULL	NULL	NULL	6	26	36
NULL	NULL	NULL	7	27	37

Resulting VR

ON L.Col1 = R.Col1

FULL JOIN (FULL OUTER JOIN)

Creates a **VR** with **all the rows from both data sources**. The unmatched values in the left and the right source are represented with NULL. It is a combined LEFT JOIN and RIGHT JOIN.

```
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  FULL JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON L.Col1 = R.Col1;
GO
```

Resulting VR

Left data source			Right data source		
Col1	Col2	Col3	Col1	Col2	Col3
1	21	31	NULL	NULL	NULL
2	22	32	NULL	NULL	NULL
3	23	33	3	23	33
4	24	34	4	24	34
5	25	35	5	25	35
NULL	NULL	NULL	6	26	36
NULL	NULL	NULL	7	27	37

ON L.Col1 = R.Col1

Let's combine (LEFT, RIGHT) JOIN and WHERE clauses to select:

- Rows from the **left data source**, that **don't have a match in the right data source** (LEFT JOIN WHERE *RightRecordset* IS NULL)

WHERE on page 142

```
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  LEFT JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON L.Col1 = R.Col1
WHERE R.Col1 IS NULL;
GO
```

Resulting VR

Left data source			Right data source		
Col1	Col2	Col3	Col1	Col2	Col3
1	21	31	NULL	NULL	NULL
2	22	32	NULL	NULL	NULL
3	23	33	3	23	33
4	24	34	4	24	34
5	25	35	5	25	35
			6	26	36
			7	27	37

Not manipulated

ON L.Col1 = R.Col1

- Rows from the **right data source**, that **don't have a match in the left data source** (`RIGHT JOIN` `WHERE` *LeftRecordset* `IS NULL`)

```
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  RIGHT JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON L.Col1 = R.Col1
WHERE L.Col1 IS NULL;
GO
```

Not manipulated

Left data source			Right data source		
Col1	Col2	Col3	Col1	Col2	Col3
1	21	31			
2	22	32			
3	23	33	3	23	33
4	24	34	4	24	34
5	25	35	5	25	35
NULL	NULL	NULL	6	26	36
NULL	NULL	NULL	7	27	37

Resulting VR

ON L.Col1 = R.Col1

- Rows from **both data sources**, that **don't have a match** (`FULL JOIN` `WHERE` *LeftRecordset* `OR` *RightRecordset* `IS NULL`). `FULL JOIN` except `INNER JOIN`.

```sql
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  FULL JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON L.Col1 = R.Col1
WHERE
  L.Col1 IS NULL
  OR R.Col1 IS NULL;
GO
```

Resulting VR

Left data source			Right data source		
Col1	Col2	Col3	Col1	Col2	Col3
1	21	31	NULL	NULL	NULL
2	22	32	NULL	NULL	NULL
3	23	33	3	23	33
4	24	34	4	24	34
5	25	35	5	25	35
NULL	NULL	NULL	6	26	36
NULL	NULL	NULL	7	27	37

ON L.Col1 = R.Col1

CROSS JOIN

Creates a VR, containing all the combinations of the rows in both data sources, (a.k.a. Cartesian Product).

```sql
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  CROSS JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R;
GO
```

No ON clausev

All the rows from the left data source

Col1	Col2	Col3	Col1	Col2	Col3
1	21	31	3	23	33
2	22	32	3	23	33
3	23	33	3	23	33
4	24	34	3	23	33
5	25	35	3	23	33
1	21	31	4	24	34
2	22	32	4	24	34
3	23	33	4	24	34
4	24	34	4	24	34
5	25	35	4	24	34
10 more rows here					
1	21	31	7	27	37
2	22	32	7	27	37
3	23	33	7	27	37
4	24	34	7	27	37
5	25	35	7	27	37

Row No. 1 from the right data source

Row No. 2 from the right data source

Row No. 5 (last) from the right data source

The **number of the combinations** of the rows is **25** (**5 rows** in the **left data source**, **5 rows** in the **right data source**).

JOIN and NULL in the ON clause (NULL in link column(s))

When we join on a column, that stores NULL, the rows with NULL can't be joined like NULL = NULL (L.Col1 = R.Col1).

Let's insert NULL to the data sources:

INSERT on page 113

```
INSERT LearnSQLServerIntuitively.dbo.LeftDataSource
  (
    Col1
    , Col2
    , Col3
  )
SELECT
  NULL
  , '2NULL'
  , '3NULL';
GO
```

The same INSERT statement to the right data source

Left data source

Col1	Col2	Col3
1	21	31
2	22	32
3	23	33
4	24	34
5	25	35
NULL	2NULL	3NULL

Right data source

Col1	Col2	Col3
3	23	33
4	24	34
5	25	35
6	26	36
7	27	37
NULL	2NULL	3NULL

```sql
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON L.Col1 = R.Col1;
GO
```

We can’t link NULL to NULL

Resulting VR

Left data source			Right data source		
Col1	Col2	Col3	Col1	Col2	Col3
1	21	31			
2	22	32			
3	23	33	3	23	33
4	24	34	4	24	34
5	25	35	5	25	35
NULL	2NULL	3NULL	6	26	36
			7	27	37
			NULL	2NULL	3NULL

Not manipulated

ON L.Col1 = R.Col1

To return the linked NULL to NULL rows, we use the built-in function ISNULL() to replace NULL (unknown) with known value and use this value to join.

```sql
SELECT
  L.*
  , R.*
FROM
  LearnSQLServerIntuitively.dbo.LeftDataSource AS L
  JOIN LearnSQLServerIntuitively.dbo.RightDataSource AS R
    ON ISNULL(L.Col1, '') = ISNULL(R.Col1, '');
GO
```

Link **empty string** to **empty string**

Resulting VR

Left data source			Right data source		
Col1	Col2	Col3	Col1	Col2	Col3
1	21	31			
2	22	32			
3	23	33	3	23	33
4	24	34	4	24	34
5	25	35	5	25	35
NULL	2NULL	3NULL	NULL	2NULL	3NULL
			6	26	36
			7	27	37

Not manipulated

```
ON L.Col1 = R.Col1
```

Replacing NULL with a real value can create incorrect results, because NULL (unknown) is not an actual value and we are joining **unknown** to **unknown**.

Aliases

The alias gives a pseudo name to a DB object, column or expression and is used to:

- point to the exact data source and the exact column when JOINing data sources
- makes the code neat and easy to understand

Customers

CustomerID	FirstName	LastName	Email	DateModified
1	Anabel	Larson	anabel.larson@ustomer5.info	1966-12-11
2	Anna	Laurier	anna.laurier@customer2.net	1964-05-13
3	Beverly	NULL	NULL	1964-08-15
4	John	Smith	john.smith@customer1.com	1963-04-25
5	John	Smith	john.smith@customer3.org	1968-05-13
6	Melanie	Larson	melanie.larson@customer4.biz	1963-12-31
7	Xavier	Jameson	xavier.garcía@customer6.com	1968-04-28
8	Zak	Smith	NULL	1963-12-11

Sales

SaleID	CustomerID	ItemID	DateOfSale	Quantity	Price	DateModified
1	3	2043	1967-01-24	4	1.27	1967-01-24
2	4	5164	1967-05-15	2	1.56	1967-05-15
3	3	5293	1966-12-11	77	55.12	1967-01-14
4	5	6223	1969-01-17	3	4.89	1969-01-17
5	7	1352	1965-10-14	89	0.84	1965-10-16
6	2	1953	1966-03-05	53	2.84	1966-03-05
7	3	8613	1963-04-13	22	23.41	1963-04-13
8	2	5523	1967-09-18	55	127.19	1967-09-18
9	1	8534	1962-10-13	1	45.88	1963-04-15
10	2	9375	1965-10-06	6	0.43	1966-04-13

CustomerID and **DateModified** exist in both tables. When we JOIN the data sources, we need to create a separate identification of the columns with the same name:

- CustomerID from Customers
- CustomerID from Sales
- DateModified from Customers
- DateModified from Sales

```
SELECT
  FirstName
  , LastName
  , DateModified
FROM
  LearnSQLServerIntuitively.dbo.Customers
  JOIN LearnSQLServerIntuitively.dbo.Sales
    ON LearnSQLServerIntuitively.dbo.Customers.CustomerID
      = LearnSQLServerIntuitively.dbo.Sales.CustomerID
WHERE CustomerID = 2;
GO
```

Exists only in Customers

Exists in more than one data source

Exists in more than one data source

Error messages are returned:

```
Ambiguous column name 'CustomerID'.
Ambiguous column name 'DateModified'.
```

To point to the exact columns in the exact objects, we add the identifiers before the column name as described in **Naming Conventions - DSO**:

Naming Conventions on page 40

```
SELECT
  FirstName
  , LastName
  , LearnSQLServerIntuitively.dbo.Customers.DateModified
  , LearnSQLServerIntuitively.dbo.Sales.DateModified
  , (Quantity * Price) AS LineTotal
FROM
  LearnSQLServerIntuitively.dbo.Customers
  JOIN LearnSQLServerIntuitively.dbo.Sales
    ON LearnSQLServerIntuitively.dbo.Customers.CustomerID
      = LearnSQLServerIntuitively.dbo.Sales.CustomerID
WHERE (Quantity * Price) >= 500
ORDER BY LineTotal DESC;
GO
```

Bind to the exact object and column

Assign Alias for the column name

[DatabaseName].[SchemaName].
[ObjectName].[ColumnName]

Alias, assigned in the SELECT clause

First-Name	Last-Name	DateModi-fied	DateModi-fied	LineTotal
Anna	Laurier	1964-05-13	1967-09-18	6995.45
Beverly	NULL	1964-08-15	1967-01-14	4244.24
Beverly	NULL	1964-08-15	1963-04-13	515.02

The **DSO identification** is too long to be added in all the clauses where needed which makes the code hard to read. The next step is to r**eplace the DSO identifiers** with **Aliases**:

```
SELECT
  FirstName
  , C.LastName
  , C.DateModified AS Cust_DateModified
  , S.DateModified AS Sale_DateModified
  , (S.Quantity * S.Price) AS LineTotal
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
WHERE (S.Quantity * S.Price) >= 500
ORDER BY LineTotal DESC;
GO
```

No Alias. The statement will work until we add another column **FirstName**, selected from another source

1. Bind to alias, assigned in the FROM clause (C, S)
2. Assign the alias in the SELECT clause (column name)

Assign alias

Can't use aliases, assigned in the SELECT clause in the WHERE clause

Aliases

With the keyword AS we assign an alias. The keyword AS is optional and can be omitted.
With . (dot) we bind to an already assigned alias.

Don't skip the keyword AS. It is very useful for a quick search of aliases in the development or debugging processes.

Search for	Find
AS	Assign
. *(dot)*	Bind

When we JOIN data sources, we always add aliases to always point to the exact column in the exact source.

```
If
  (
    We don't use aliases
    and
    Modify a data source, used by the statement (add column with the same name as a col-
umn in another data source, used by the statement)
  )
{ the execution stops with the error message "Ambiguous column name". };
```

Assign TableName and ColumnName(s) in one alias

We can assign **column names** of an alias object by adding the list next to the table alias in round braces:

```
SELECT
  C.FN
  , C.LN
FROM
  (
    SELECT TOP 2
      FirstName
      , LastName
    FROM LearnSQLServerIntuitively.dbo.Customers
  ) AS C (FN, LN);
GO
```

FN	LN
Anabel	Larson
Anna	Laurier

C

FirstName	LastName
Anabel	Larson
Anna	Laurier

Assign **table alias** and **column aliases** in braces

Alias in the SELECT clause

In the SELECT clause we can use **assign** (creates a pseudo name for the column) and **bind** (binds to the data source in the FROM clause) aliases.

If we skip the alias, the column header is **(No column name)**:

```
SELECT TOP 3 CONCAT(FirstName, ' ', LastName)
FROM LearnSQLServerIntuitively.dbo.Customers
GO
```

(No column name)
Anabel Larson
Anna Laurier
Beverly

Assign alias (column name)

```
SELECT TOP 3 CONCAT(FirstName, ' ', LastName) AS Name
SELECT TOP 3 CONCAT(C.FirstName, ' ', C.LastName)
SELECT TOP 3 CONCAT(FirstName, ' ', LastName) AS [Customer Name]
```

Bind Alias (to ON in the FROM clause)

Alias, including special character is wrapped in **square brackets** ([])

Create aliases without special characters, to facilitate the usage of the column names in the requesting application.

Alias with equal sign (=)

We can assign an alias with the equal sign:

Alias

Column Identifier

```
SELECT FN = FirstName
FROM LearnSQLServerIntuitively.dbo.Customers;
GO
```

- Can be used only in the SELECT clause
- The syntax that assigns value to variable is similar:

```
DECLARE @FN VARCHAR(128);
SELECT @FN = FirstName
FROM LearnSQLServerIntuitively.dbo.Customers;
GO
```

will assign a value to the variable @FN.

Variables on page 199

Alias in the FROM clause

The **assign** alias in the FROM clause creates a pseudo name for the DB object.
Bind alias in ON clause, pointing to the assign alias in the FROM clause

Alias in the WHERE, GROUP BY, HAVING and ORDER BY clauses

Only **bind** aliases.

SELECT... (FROM...) WHERE

...or filter the recordset

After a virtual recordeset (VR) is built with the SELECT and FROM (if FROM is used) clauses, we can filter the rows with the WHERE clause.

How it works?

1. The SELECT (and FROM) clause(s) build the source virtual recordset (SVR)
2. The filter condition(s) in the WHERE clause is/are verified on every row of the SVR
3. If all the filtered conditions return **True**, the row is selected in the resulting recordert (RR)

The source of the data (table **Customers**)

Customers

First-Name	Last-Name	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	NULL	NULL
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org

Source virtual record-set (source of the data to be filtered with the WHERE clause)

First-Name	Last-Name
Anabel	Larson
Anna	Laurier
Beverly	NULL
John	Smith
John	Smith

Resulting recordert

First-Name	Last-Name
Anabel	Larson
Anna	Laurier

```
SELECT
  FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE LastName LIKE 'La%';
GO
```

The basic syntax is:

```
SELECT...
FROM...
WHERE  {Left Operand} {Operator} {Right Operand};
GO
```

In the statement:

```
SELECT 'ABC' AS ColumnName1
WHERE 17 = 45;
GO
```

- The SELECT clause builds the SVR (one row, one column and value 'ABC')
- The WHERE clause is validated once (one row in the SVR) and the result of the validation is False (17 doesn't equal 45)
- The RR is blank (nothing is selected)

The statement:

```
SELECT 'ABC' AS ColumnName1
WHERE 17 = 17;
GO
```

returns a RR with one row, one column and value 'ABC', because the filter condition in the where clause is validated to True (17 equals 17).

The full syntax is:

```
SELECT
  T1.ColumnName2
  , T2.ColumnName3
  , ...
FROM
  [DatabaseName].[SchemaName].[Table1Name] AS T1
  JOIN [DatabaseName].[SchemaName].[Table2Name] AS T2
    ON T1.ColumnName1 = T2.ColumnName1
WHERE
  {NOT logical operator} [Bind alias].[Filtered column name] {comparison or logical operator} {compared value, expression or VR}
  {AND, OR logical operator} {NOT logical operator} [Bind Alias].[Filtered column name] {comparison or logical operator} {compared value, expression or VR}
  AND ...;
GO
```

Left Operand
Operator
Right Operand

Filter condition 1
Filter condition 2
Filter condition 3

The filtering logic is:

- The condition(s) is/are validated on every row of the SVR
- The validation returns Boolean value - **True**, **False** or **Unknown** (NULL)
- The rows from the SVR validated as True are selected

NULL **and (Three-valued Logic)** on page 69

NOT **(logical operator)**
Reverses the verification result of the filter condition.

[Bind alias].[Filtered column name] ← Left operand of the comparison
Column in the SVR on which the filter condition is applied.

Operator (comparison or logical) ← Operator
Determines how to compare the values in the filter condition.

Operators on page 71

Compared value, expression or recordset (VR) ← Right operand of the comparison
Value, expression or VR that is compared with the filtered column.

Operator that joins filter conditions (logical operator)

- AND - the row in the SVR is selected if the left AND the right filter condition returns True
- OR - the row in the SVR is selected if one of the left OR the right filter condition returns True

Example with one filter condition

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE FirstName = 'John';
GO
```

Left operand — Operator — Right operand

Filter condition

Customers

First-Name	Last-Name	Email	FirstName = 'John'?	Row is returned?
Anabel	Larson	anabel.larson@customer5.info	False	No
Anna	Laurier	anna.laurier@customer2.net	False	No
Beverly	NULL	NULL	False	No
John	Smith	john.smith@customer1.com	True	Yes
John	Smith	john.smith@customer3.org	True	Yes
Melanie	Larson	melanie.larson@customer4.biz	False	No
Xavier	Jameson	xavier.garcía@customer6.com	False	No
Zak	Smith	NULL	False	No

Example with more than one filter conditions (AND, OR logical operators)

The logical operator AND:

- Additionally narrows down the selected result, because all conditions joined with AND have to be valid (return True)
- Marks the rows to be returned where the left AND the right conditions evaluate to True

```sql
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE
  FirstName = 'Anabel'          ← Left filter condition
  AND LastName = 'Larson';      ← Right filter condition
GO
```

Logical operator AND which joins the filter conditions

Customers

First-Name	Last-Name	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	NULL	NULL
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org
Melanie	Larson	melanie.larson@customer4.biz
Xavier	Jameson	xavier.garcía@customer6.com
Zak	Smith	NULL

FirstName = 'Anabel'?	Logical Join Operator	LastName = 'Larson'?	Row is returned?
True	AND	True	Yes
False	AND	False	No
False	AND	False	No
False	AND	False	No
False	AND	False	No
False	AND	True	No
False	AND	False	No
False	AND	False	No

Filter condition

The operator OR:

- returns the lines where **either left** OR **right** condition evaluates to True

```sql
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE
  FirstName = 'Anabel'          ← Left filter condition
  OR LastName = 'Larson';       ← Right filter condition
GO
```

Logical operator AND that joins the filter conditions

Customers

First-Name	Last-Name	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	NULL	NULL
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org
Melanie	Larson	melanie.larson@customer4.biz
Xavier	Jameson	xavier.garcía@customer6.com
Zak	Smith	NULL

FirstName = 'Anabel'?	Logical Join Operator	LastName = 'Larson'?	Row is returned?
True	OR	True	Yes
False	OR	False	No
False	OR	False	No
False	OR	False	No
False	OR	False	No
False	OR	True	Yes
False	OR	False	No
False	OR	False	No

More than one filter conditions, joined with both logical operator AND and OR:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE
  (
    FirstName = 'Anabel'        ← Left filter condition
    OR LastName = 'Larson'      ← Right filter condition
  )                             ← Left filter condition (between the brackets)
  AND Email LIKE '%.biz';       ← Right filter condition
GO
```

Separate (in brackets) the OR operator(s) from the AND operator(s).

Customers

First-Name	Last-Name	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	NULL	NULL
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org
Melanie	Larson	melanie.larson@customer4.biz
Xavier	Jameson	xavier.garcía@customer6.com
Zak	Smith	NULL

FirstName = 'Anabel'?	Logical Join Operator	LastName = 'Larson'?	OR Evaluation	Logical Join Operator	Email LIKE '%.biz'?	AND Evaluation	Row is returned?
True	OR	True	True	AND	False	False	No
False	OR	False	False	AND	False	False	No
False	OR	False	False	AND	False	False	No
False	OR	False	False	AND	False	False	No
False	OR	False	False	AND	False	False	No
False	OR	True	True	AND	True	True	Yes
False	OR	False	False	AND	False	False	No
False	OR	False	False	AND	False	False	No

The WHERE clause is used to filter not only SELECT statements; it is also used in other DML statements (UPDATE, DELETE) that manipulate SVR.

UPDATE on page 118

DELETE on page 123

The ORDER BY clause orders the resulting recordset (RR).

We can order on:

- Identifier - the name of the column in a table, view or table-valued function
- Bind alias - a substitute of the column name in the SELECT clause
- Column Index - the position of a column in the RR

Aliases on page 138

Customers

First-Name	Last-Name	Email	DateOf-Birth
Anabel	Larson	anabel.larson@customer5.info	1953-03-15
Anna	Laurier	anna.laurier@customer2.net	1948-07-17
Beverly	NULL	NULL	1948-07-17
John	Smith	john.smith@customer1.com	1955-08-16
John	Smith	john.smith@customer3.org	1950-05-18
Melanie	Larson	melanie.larson@customer4.biz	1951-12-18
Xavier	Jameson	xavier.garcía@customer6.com	1941-08-23
Zak	Smith	NULL	1936-11-08

```
SELECT
  FirstName AS FN
  , LastName
  , DateOfBirth
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE DateOfBirth >= '1950-01-01'
ORDER BY
  LastName
  , FN ASC
  , 3 DESC;
GO
```

Assign alias

Column index is 3 (the third column in the RR)

Column name

Bind alias

The column with index 3 is DateOfBirth

First-Name	Last-Name	DateOf-Birth
Anabel	Larson	1953-03-15
Melanie	Larson	1951.12-18
John	Smith	1955-08-16
John	Smith	1950-05-18

ASC - Ascending. Default, can be omitted
DESC - Descending

The RR is ordered by:

1. **LastName** - ascending (L in Larson is before S in Smith in the alphabet)
2. **FirstName** - ascending (A in Anabel is before M in Melanie in the alphabet)
3. **DateOfBirth** - descending (for both John Smith year 1955 is after 1950 in the calendar)

We can ORDER BY column(s), not included in the RR (not in the SELECT clause).

ORDER BY is expensive operation. It is recommended to be used only when we need ordered resulting recordset.

SELECT TOP... ORDER BY...

When we extend the SELECT clause with the TOP keyword, we select a recordset with *Number* or *Percent* rows.

In order to return the exact result, TOP always has to be used together with ORDER BY clause.

The syntax is:

```
SELECT TOP Number PERCENT ColumnName(s) delimited with comma (,) or *
FROM ObjectName - table, view, table-valued function
ORDER BY ColumnName (ASC, DESC);
GO
```

ASC is default and can be omitted

Customers

First-Name	Last-Name	Email	DateOf-Birth
Anabel	Larson	anabel.larson@customer5.info	1967-05-11
Anna	Laurier	anna.laurier@customer2.net	1967-05-11
Beverly	NULL	NULL	1965-11-09
John	Smith	john.smith@customer1.com	1963-07-19
John	Smith	john.smith@customer3.org	1965-11-09
Melanie	Larson	melanie.larson@customer4.biz	1965-09-08
Xavier	Jameson	xavier.garcía@customer6.com	1965-11-09
Zak	Smith	NULL	1963-12-11

```
SELECT TOP (3)
  FirstName
  , LastName
  , DateOfBirth
FROM LearnSQLServerIntuitively.dbo.Customers
ORDER BY
  DateOfBirth DESC
  , LastName
  , FirstName;
GO
```

Number of rows

First-Name	Last-Name	DateOf-Birth
Anabel	Larson	1967-05-11
Anna	Laurier	1967-05-11
Beverly	NULL	1965-11-09

SELECT TOP... WITH TIES

WITH TIES adds **ties** to the resulting recordset.

We select TOP (3), but **DateOfBirth = 1965-11-09** exists also for John Smith and Xavier Jameson (5 rows selected - ties)

```
SELECT TOP (3) WITH TIES *
FROM LearnSQLServerIntuitively.dbo.Customers
ORDER BY DateOfBirth DESC;
GO
```

First-Name	Last-Name	DateOf-Birth
Anabel	Larson	1967-05-11
Anna	Laurier	1967-05-11
Beverly	NULL	1965-11-09
John	Smith	1965-11-09
Xavier	Jameson	1965-11-09

SELECT TOP… PERCENT

Select specified percent of all the rows in the source recordset.

```
SELECT TOP (35) PERCENT *
FROM LearnSQLServerIntuitively.dbo.Customers
ORDER BY
  DateOfBirth DESC
  , LastName
  , FirstName;
GO
```

First-Name	Last-Name	DateOf-Birth
Anabel	Larson	1967-05-11
Anna	Laurier	1967-05-11
Beverly	NULL	1965-11-09

SELECT TOP... ORDER BY... OFFSET... FETCH

Starting from any row but the first, we can select rows from ordered recordsets. To do this, we use the OFFSET clause, added to the ORDER BY clause.

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
6	Melanie	Larson	melanie.larson@customer4.biz
7	Xavier	Jameson	xavier.garcía@customer6.com
8	Zak	Smith	NULL

Customers

CustomerID from 1 to 3 are skipped

CustomerID	FirstName	LastName	Email
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
6	Melanie	Larson	melanie.larson@customer4.biz
7	Xavier	Jameson	xavier.garcía@customer6.com
8	Zak	Smith	NULL

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
ORDER BY
  FirstName
  , LastName
  OFFSET 3 ROWS;
GO
```

OFFSET 1 ROW, OFFSET **X** ROWS

By adding the FETCH clause to OFFSET, we can pick a specified number of rows, starting from the row specified in OFFSET:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
ORDER BY
  FirstName
  , LastName
  OFFSET 3 ROWS
  FETCH NEXT 3 ROWS ONLY;
GO
```

1 ROW ONLY, **X** ROWS ONLY

FETCH FIRST, FETCH NEXT

CustomerID	FirstName	LastName	Email
5	John	Smith	john.smith@customer3.org
4	John	Smith	john.smith@customer1.com
6	Melanie	Larson	melanie.larson@customer4.biz

CustomerID from 1 to 3 are skipped and the next 3 rows are selected

OFFSET and FETCH are very useful when we need to **paginate** the result.

By adding a GROUP BY clause, we create groups from columns and calculate values for other columns.

- Grouped columns (a.k.a. Dimensions) – the columns that define the scope for aggregation (the groups)
- Aggregated columns (a.k.a. Facts, Measures or Properties) – summarized data for the groups

The syntax is:

```
SELECT
  Col1
  , Col2
  , Aggregation Function(Col3)
  , Aggregation Function(Col4)
FROM [DatabaseName].[SchemaName].[ObjectName]
GROUP BY
  Col1
  , Col2;
GO
```

Grouped column

Aggregated column

Grouped column

Source of the data

LearnSQLServer-Intuitively.dbo. GroupBy1

	Grouped columns			Not selected columns		Aggregated columns				
	Col1	**Col2**	**Col3**	**Col4**	**Col5**	**Col6**	**Col7**	**Col8**	**Col9**	**Col10**
Group 1	1	2	3	NULL		4	3	3	6	8
	1	2	3	A	B	4	4	1	6	2
	1	2	3	D	C	NULL	5	8	7	2
Group 2	2	1	3	15	23	4	1	12	23	2
	2	1	3	6	C	8	7	4	78.5	3
	2	1	3	8	7	8	4	-56	5	1
	2	1	3	FA	S	5	NULL	6	5	4
Group 3	3	2	1	NULL	SD	2	6	BCD	BCD	ABE
	3	2	1	16	R	8	23	ABD	ABD	ABC
	3	2	1	D	2	17	5	ABC	ABC	ABC
	3	2	1	RH	42	45	9	AAB	AAB	ABD

First we build a virtual recordset:

```
SELECT
  Col1, Col2, Col3
  , Col6, Col7, Col8
  , Col9, Col10
FROM LearnSQLServer
Intuitively.dbo.
GroupBy1
GO
```

Next we modify the query to create groups on **Col1**, **Col2** and **Col3** and aggregate the data in columns **Col6** to **Col10**:

```
SELECT
  Col1 AS [Column 1]
  , Col2
  , Col3
  , SUM(Col6) AS [Sum Col6]
  , AVG(Col7) AS Avg_Col7
  , MIN(Col8) AS Min_Col8
  , MAX(Col9) AS Max_Col9
  , COUNT(Col10) AS Count_Col10
  , COUNT(DISTINCT Col10) AS CountDistinct_Col10
FROM LearnSQLServerIntuitively.dbo.GroupBy1
```

Grouped columns (Source is **structure tables** or **Dimensions** in OLAP)

Aggregated columns (Source is **data tables** or **Facts** in OLAP. A.k.a. Measures in the reporting slang)

```
GROUP BY
  Col1
  , Col2
  , Col3;
GO
```

Grouped columns. We can't use the alias **Column 1** here

Resulting recordset

LearnSQLServerIntuitively.dbo.GroupBy1

				SUM()	AVG()	MIN()	MAX()	COUNT()	COUNT(DISTINCT)
	Column 1	**Col2**	**Col3**	**Sum Col6**	**Avg_Col7**	**Min_Col8**	**Max_Col9**	**Count_Col10**	**CountDistinct_Col10**
Group 1	1	2	3	8.000	4.000000	1	7	3	2
Group 2	2	1	3	25.000	4.000000	-56	78.5	4	4
Group 3	3	2	1	72.000	10.750000	AAB	BCD	4	3
	Grouped columns			*Aggregated columns*					

Numbers ordered **first**

Characters ordered **last**

The grouped columns in the resultset are unique (no duplicates for **Col 1**, **Col2** and **Col3**).
The aggregate functions summarize the values for the aggregated columns.

- NULL is excluded from the aggregations
  In **Col7**, *Group 2*, the values are **1**, **7**, **4** and NULL
  AVG() returns **4** ((**1** + **7** + **4**) / **3**) and not **3** ((**1** + **7** + **4** + NULL) / **4**)

- GROUP BY can't use the bound aliases, assigned in the SELECT clause, because GROUP BY is before SELECT (where the aliases are assigned) in the execution order

Execution Logic on page 163

Aggregate Function	Returns
SUM(ColumnName)	Sums the values in the column
MIN(ColumnName)	Selects the minimum value in the column
MAX(ColumnName)	Selects the maximum value in the column
AVG(ColumnName)	Selects the average of the values in the column
COUNT(ColumnName)	Counts the values in the column
COUNT(DISTINCT ColumnName)	Count unique values in the column

Aggregate Function, covered by this book

GROUP BY is like SELECT DISTINCT

When we SELECT columns and GROUP BY the same columns, without aggregating other columns, the resultset is the same as if we SELECT DISTINCT these columns.

```sql
SELECT
  Col1
  , Col2
  , Col3
  , Col6
FROM LearnSQLServerIntuitively.dbo.GroupBy1
GROUP BY
  Col1
  , Col2
  , Col3
  , Col6
GO
```

No aggregated columns

The same columns as in the SELECT clause

and

```sql
SELECT DISTINCT
  Col1
  , Col2
  , Col3
  , Col6
FROM LearnSQLServerIntuitively.dbo.GroupBy1;
GO
```

create the same resultset:

Col1	Col2	Col3	Col6
1	2	3	NULL
1	2	3	4
2	1	3	4
2	1	3	5
2	1	3	8
3	2	1	2
3	2	1	8
3	2	1	17
3	2	1	45

HAVING

The HAVING clause filters already aggregated data. It can't be used without the GROUP BY clause.

The data flow is:

1. Create a recordset - SELECT... FROM... JOIN
2. Filter the recordset - WHERE
3. Aggregate the recordset - GROUP BY
4. Filter the aggregated recordset - HAVING

The step by step example below selects the customers with total sales more or equal to 150$ for the period from 1965-10-06 to 1967-05-15.

Customers

CustomerID	FirstName	LastName	Email	DateModified
1	Anabel	Larson	anabel.larson@customer5.info	1966-12-11
2	Anna	Laurier	anna.laurier@customer2.net	1964-05-13
3	Beverly	NULL	NULL	1964-08-15
4	John	Smith	john.smith@customer1.com	1963-04-25
5	John	Smith	john.smith@customer3.org	1968-05-13
6	Melanie	Larson	melanie.larson@customer4.biz	1963-12-31
7	Xavier	Jameson	xavier.garcía@customer6.com	1968-04-28
8	Zak	Smith	NULL	1963-12-11

1 ∞

Sales

The structure table **Items** is not shown

SaleID	CustomerID	ItemID	DateOfSale	Quantity	Price	DateModified
1	3	2043	1967-01-24	4	1.27	1967-01-24
2	4	5164	1967-05-15	2	1.56	1967-05-15
3	3	5293	1966-12-11	77	55.12	1967-01-14
4	5	6223	1969-01-17	3	4.89	1969-01-17
5	7	1352	1965-10-14	89	0.84	1965-10-16
6	2	1953	1966-03-05	53	2.84	1966-03-05
7	3	8613	1963-04-13	22	23.41	1963-04-13
8	2	5523	1967-09-18	55	127.19	1967-09-18
9	1	8534	1962-10-13	1	45.88	1963-04-15
10	2	9375	1965-10-06	6	0.43	1966-04-13

1. Create a recordset (JOIN tables **Customers** and **Sales**)

```
SELECT
  S.DateOfSale
  , C.FirstName
  , C.LastName
  , S.ItemID
  , (S.Quantity * S.Price) AS LineTotal
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID;
GO
```

DateOfSale	FirstName	LastName	ItemID	LineTotal
1967-01-24	Beverly	NULL	2043	5.08
1967-05-15	John	Smith	5164	3.12
1966-12-11	Beverly	NULL	5293	4244.24
1969-01-17	John	Smith	6223	14.67
1965-10-14	Xavier	Jameson	1352	74.76
1966-03-05	Anna	Laurier	1953	150.52
1963-04-13	Beverly	NULL	8613	515.02
1967-09-18	Anna	Laurier	5523	6995.45
1962-10-13	Anabel	Larson	8534	45.88
1965-10-06	Anna	Laurier	9375	2.58

2. Filter the recordset (sales WHERE the period is BETWEEN **1965-10-06** to **1967-05-15**)

```
SELECT
  S.DateOfSale
  , C.FirstName
  , C.LastName
  , S.ItemID
  , (S.Quantity * S.Price) AS LineTotal
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
WHERE S.DateOfSale BETWEEN '1965-10-06' AND '1967-05-15';
GO
```

DateOf-Sale	First-Name	Last-Name	ItemID	LineTo-tal
1966-03-05	Anna	Laurier	1953	150.52
1965-10-06	Anna	Laurier	9375	2.58
1967-01-24	Beverly	NULL	2043	5.08
1966-12-11	Beverly	NULL	5293	4244.24
1967-05-15	John	Smith	5164	3.12
1965-10-14	Xavier	Jameson	1352	74.76

3. Aggregate the recordset (SUM() the sales for each customer)

```
SELECT
       C.FirstName
       , C.LastName
       , SUM(S.Quantity * S.Price) AS LineTotal
FROM
       LearnSQLServerIntuitively.dbo.Customers AS C
       JOIN LearnSQLServerIntuitively.dbo.Sales AS S
              ON C.CustomerID = S.CustomerID
WHERE S.DateOfSale BETWEEN '1965-10-06' AND '1967-05-15'
GROUP BY
       C.FirstName
       , C.LastName;
GO
```

First-Name	Last-Name	LineTo-tal
Anna	Laurier	153.10
Beverly	NULL	4249.32
John	Smith	3.12
Xavier	Jameson	74.76

150.52 + 2.58

4244.24 + 5.08

4. Filter the aggregated recordset (customers HAVING total sales more or equal to 150$)

```
SELECT
       C.FirstName
       , C.LastName
       , SUM(S.Quantity * S.Price) AS LineTotal
FROM
       LearnSQLServerIntuitively.dbo.Customers AS C
       JOIN LearnSQLServerIntuitively.dbo.Sales AS S
              ON C.CustomerID = S.CustomerID
WHERE S.DateOfSale BETWEEN '1965-10-06' AND '1967-05-15'
GROUP BY
```

First-Name	Last-Name	LineTo-tal
Anna	Laurier	153.10
Beverly	NULL	4249.32

>= 150

```
        C.FirstName
        , C.LastName
HAVING SUM(S.Quantity * S.Price) >= 150;
GO
```

HAVING can’t bind the aliases, assigned in the SELECT statement

Combines multiple GROUP BY statements into one.

Sales

Customer	Store	Item	SaleValue
Customer 1	Store 1	Item 3	303.6996
Customer 1	Store 5	Item 1	368.5147
Customer 2	Store 2	Item 1	366.3898
Customer 2	Store 2	Item 4	109.3525
Customer 2	Store 5	Item 6	949.0834
Customer 3	Store 1	Item 6	615.9823
Customer 4	Store 1	Item 3	911.5682
Customer 4	Store 4	Item 7	93.2123
Customer 4	Store 5	Item 1	95.5645
Customer 5	Store 2	Item 3	550.4955
Customer 5	Store 4	Item 4	914.9231
Customer 5	Store 4	Item 4	679.8012
Customer 6	Store 5	Item 3	758.6422
Customer 6	Store 5	Item 3	951.0204
Customer 7	Store 2	Item 2	923.3026
Customer 8	Store 4	Item 5	813.9188

The statement below returns aggregated data for groups:

- Total
- Customer
- Store
- Item

```
SELECT
  Customer
  , Store
  , Item
  , SUM(SaleValue) AS Sum_SaleValue
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY GROUPING SETS
  (
    Customer
    , Store
    , Item
    , ()                    ← Total
  )
ORDER BY
  Customer
  , Store
  , Item
GO
```

Customer	Store	Item	Sum_SaleValue	
NULL	NULL	NULL	9405.4711	Total
NULL	NULL	Item 1	830.469	Item
NULL	NULL	Item 2	923.3026	
NULL	NULL	Item 3	3475.4259	
NULL	NULL	Item 4	1704.0768	
NULL	NULL	Item 5	813.9188	
NULL	NULL	Item 6	1565.0657	
NULL	NULL	Item 7	93.2123	
NULL	Store 1	NULL	1831.2501	Store
NULL	Store 2	NULL	1949.5404	
NULL	Store 4	NULL	2501.8554	
NULL	Store 5	NULL	3122.8252	
Customer 1	NULL	NULL	672.2143	
Customer 2	NULL	NULL	1424.8257	
Customer 3	NULL	NULL	615.9823	
Customer 4	NULL	NULL	1100.345	
Customer 5	NULL	NULL	2145.2198	
Customer 6	NULL	NULL	1709.6626	
Customer 7	NULL	NULL	923.3026	
Customer 8	NULL	NULL	813.9188	

Σ = 9405.4711

GROUPING SETS

The statement below returns aggregated data for groups:

- Total
- Store and Item
- Customer and Item

```
SELECT
  Customer
  , Store
  , Item
  , SUM(SaleValue) AS Sum_SaleValue
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY GROUPING SETS
  (
    (Customer, Item)
    , (Store, Item)
    , ()
  )
ORDER BY
  Customer
  , Store
  , Item
GO
```

Customer and Item

Store and Item

Total

Customer	Store	Item	Sum_SaleValue
NULL	NULL	NULL	9405.4711
NULL	Store 1	Item 3	1215.2678
NULL	Store 1	Item 6	615.9823
NULL	Store 2	Item 1	366.3898
NULL	Store 2	Item 2	923.3026
NULL	Store 2	Item 3	550.4955
NULL	Store 2	Item 4	109.3525
NULL	Store 4	Item 4	1594.7243
NULL	Store 4	Item 5	813.9188
NULL	Store 4	Item 7	93.2123
NULL	Store 5	Item 1	464.0792
NULL	Store 5	Item 3	1709.6626
NULL	Store 5	Item 6	949.0834
Customer 1	NULL	Item 1	368.5147
Customer 1	NULL	Item 3	303.6996
Customer 2	NULL	Item 1	366.3898
Customer 2	NULL	Item 4	109.3525
Customer 2	NULL	Item 6	949.0834
Customer 3	NULL	Item 6	615.9823
Customer 4	NULL	Item 1	95.5645
Customer 4	NULL	Item 3	911.5682
Customer 4	NULL	Item 7	93.2123
Customer 5	NULL	Item 3	550.4955
Customer 5	NULL	Item 4	1594.7243
Customer 6	NULL	Item 3	1709.6626
Customer 7	NULL	Item 2	923.3026
Customer 8	NULL	Item 5	813.9188

Σ = 9405.4711

The same result will be created by multiple UNIONed GROUP BY statements:

```
SELECT A.*
FROM
  (
    SELECT
      NULL AS Customer
      , NULL AS Store
      , NULL AS Item
      , SUM(SaleValue) AS Sum_SaleValue
    FROM LearnSQLServerIntuitively.dbo.Sales

    UNION ALL
```

Total

```
    SELECT
      Customer
      , NULL AS Store
      , Item
      , SUM(SaleValue) AS Sum_SaleValue
    FROM LearnSQLServerIntuitively.dbo.Sales
    GROUP BY
      Customer
      , Item
```

Customer and Item

```
    UNION ALL
```

```
    SELECT
      NULL AS Customer
      , Store
      , Item
      , SUM(SaleValue) AS Sum_SaleValue
    FROM LearnSQLServerIntuitively.dbo.Sales
    GROUP BY
      Store
      , Item
  ) AS A
ORDER BY
  A.Customer
  , A.Store
  , A.Item
GO
```

Store and Item

With GROUPING SETS we can present different slice and dices of the data in one statement.

DML Statements (Query)
ROLLUP and CUBE

When we aggregate data, we add subtotals and a grand total to the resultset with ROLLUP and CUBE.

Sales

SaleID	CustomerName	ItemDescription	DateOfSale	Quantity	Price
1	Xavier Jameson	Plastic Plain Blue	1967-01-24	4	1.27
2	John Smith	Plastic Plain Blue	1967-05-15	2	1.56
3	Xavier Jameson	Rag Doll 33 cm.	1966-12-11	77	55.12
4	Anabel Larson	Plastic Plain Blue	1969-01-17	3	4.89
5	Anabel Larson	Wooden Horse	1965-10-14	89	0.84
6	John Smith	Rag Doll 33 cm.	1966-03-05	53	2.84
7	Xavier Jameson	Wooden Horse	1963-04-13	22	23.41
8	John Smith	Rag Doll 33 cm.	1967-09-18	55	127.19
9	Anabel Larson	Wooden Horse	1962-10-13	1	45.88
10	John Smith	Plastic Plain Blue	1965-10-06	6	0.43

Add ROLLUP to the GROUP BY clause to calculate subtotals and the grand total for the grouped columns:

```
SELECT
  CustomerName
  , ItemDescription
  , DateOfSale
  , SUM(Quantity * Price) AS SalesValue
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY ROLLUP
  (
    CustomerName
    , ItemDescription
    , DateOfSale
  )
ORDER BY
  CustomerName
  , ItemDescription
  , DateOfSale;
GO
```

CustomerName	ItemDescription	DateOf-Sale	SalesVal-ue
NULL	**NULL**	**NULL**	**12051.32**
Anabel Larson	**NULL**	**NULL**	**135.31**
Anabel Larson	**Plastic Plain Blue**	**NULL**	**14.67**
Anabel Larson	Plastic Plain Blue	1969-01-17	14.67
Anabel Larson	**Wooden Horse**	**NULL**	**120.64**
Anabel Larson	Wooden Horse	1962-10-13	45.88
Anabel Larson	Wooden Horse	1965-10-14	74.76
John Smith	**NULL**	**NULL**	**7151.67**
John Smith	**Plastic Plain Blue**	**NULL**	**5.70**
John Smith	Plastic Plain Blue	1965-10-06	2.58
John Smith	Plastic Plain Blue	1967-05-15	3.12
John Smith	**Rag Doll 33 cm.**	**NULL**	**7145.97**
John Smith	Rag Doll 33 cm.	1966-03-05	150.52
John Smith	Rag Doll 33 cm.	1967-09-18	6995.45
Xavier Jameson	**NULL**	**NULL**	**4764.34**
Xavier Jameson	**Plastic Plain Blue**	**NULL**	**5.08**
Xavier Jameson	Plastic Plain Blue	1967-01-24	5.08
Xavier Jameson	**Rag Doll 33 cm.**	**NULL**	**4244.24**
Xavier Jameson	Rag Doll 33 cm.	1966-12-11	4244.24
Xavier Jameson	**Wooden Horse**	**NULL**	**515.02**
Xavier Jameson	Wooden Horse	1963-04-13	515.02

Level 1 (GROUP BY CustomerName, ItemDescription, SaleDate)

Level 2 (GROUP BY Customer-Name, ItemDescription)

Subtotal

Level 3 (GROUP BY CustomerName)

Level 4 (No GROUP BY)

Grand Total

All the rows with NULL in a column, belonging to a group (not an aggregated column) are added by ROLLUP operator.

We can use the GROUPING() function to transform NULLs to human readable values:

```
SELECT
  CASE
    WHEN GROUPING(CustomerName) = 1
    THEN 'Total: Customer'
    ELSE CustomerName
  END AS Customer
  , CASE
    WHEN GROUPING(ItemDescription) = 1
    THEN 'Total: Item'
    ELSE ItemDescription
  END AS Item
  , CASE
    WHEN GROUPING(DateOfSale) = 1
    THEN 'Total: Date of Sale'
    ELSE CONVERT(VARCHAR(17), DateOfSale)
  END AS DateOfSale
  , SUM(Quantity * Price) AS SalesValue
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY ROLLUP
  (
    CustomerName
    , ItemDescription
    , DateOfSale
  )
ORDER BY
  CustomerName
  , ItemDescription
  , DateOfSale;
GO
```

CASE on page 186

Customer	Item	Date	SalesValue
Total: Customer	**Total: Item**	**Total: Date**	**12051.32**
Anabel Larson	**Total: Item**	**Total: Date**	**135.31**
Anabel Larson	**Plastic Plain Blue**	**Total: Date**	**14.67**
Anabel Larson	Plastic Plain Blue	1969-01-17	14.67
Anabel Larson	**Wooden Horse**	**Total: Date**	**120.64**
Anabel Larson	Wooden Horse	1962-10-13	45.88
Anabel Larson	Wooden Horse	1965-10-14	74.76
John Smith	**Total: Item**	**Total: Date**	**7151.67**
John Smith	**Plastic Plain Blue**	**Total: Date**	**5.70**
John Smith	Plastic Plain Blue	1965-10-06	2.58
John Smith	Plastic Plain Blue	1967-05-15	3.12
John Smith	**Rag Doll 33 cm.**	**Total: Date**	**7145.97**
John Smith	Rag Doll 33 cm.	1966-03-05	150.52
John Smith	Rag Doll 33 cm.	1967-09-18	6995.45
Xavier Jameson	**Total: Item**	**Total: Date**	**4764.34**
Xavier Jameson	**Plastic Plain Blue**	**Total: Date**	**5.08**
Xavier Jameson	Plastic Plain Blue	1967-01-24	5.08
Xavier Jameson	Rag Doll 33 cm.	Total: Date	4244.24
Xavier Jameson	Rag Doll 33 cm.	1966-12-11	4244.24
Xavier Jameson	Wooden Horse	Total: Date	515.02
Xavier Jameson	Wooden Horse	1963-04-13	515.02

CUBE

When we add CUBE to the GROUP BY clause, we calculate subtotals and grand totals for all the combinations in the grouped columns:

Sales

SaleID	CustomerName	ItemDescription	SaleDate	Quantity	Price
1	Xavier Jameson	Plastic Plain Blue	1967-01-24	4	1.27
2	Anabel Larson	Wooden Horse	1962-10-13	1	45.88
3	Anabel Larson	Wooden Horse	1964-11-07	1	41.32

DML Statements (Query)
ROLLUP and CUBE

```
SELECT
  CASE
    WHEN GROUPING(CustomerName) = 1
      THEN 'Total: Customer'
    ELSE CustomerName
  END AS Customer
  , CASE
    WHEN GROUPING(ItemDescription) = 1
      THEN 'Total: Item'
    ELSE ItemDescription
  END AS Item
  , CASE
    WHEN GROUPING(DateOfSale) = 1
      THEN 'Total: Date of Sale'
    ELSE CONVERT(VARCHAR(17), DateOfSale)
  END AS [Date]
  , SUM(Quantity * Price) AS SalesValue
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY CUBE
  (
    CustomerName
    , ItemDescription
    , DateOfSale
  )
ORDER BY
  CustomerName
  , ItemDescription
  , DateOfSale;
GO
```

Customer	Item	Date	Sales-Value
Total: Customer	**Total: Item**	**Total: Date**	**92.28**
Total: Customer	**Total: Item**	**1962-10-13**	**45.88**
Total: Customer	**Total: Item**	**1964-11-07**	**41.32**
Total: Customer	**Total: Item**	**1967-01-24**	**5.08**
Total: Customer	**Plastic Plain Blue**	**Total: Date**	**5.08**
Total: Customer	**Plastic Plain Blue**	**1967-01-24**	**5.08**
Total: Customer	**Wooden Horse**	**Total: Date**	**87.20**
Total: Customer	**Wooden Horse**	**1962-10-13**	**45.88**
Total: Customer	**Wooden Horse**	**1964-11-07**	**41.32**
Anabel Larson	**Total: Item**	**Total: Date**	**87.20**
Anabel Larson	**Total: Item**	**1962-10-13**	**45.88**
Anabel Larson	**Total: Item**	**1964-11-07**	**41.32**
Anabel Larson	**Wooden Horse**	**Total: Date**	**87.20**
Anabel Larson	Wooden Horse	1962-10-13	45.88
Anabel Larson	Wooden Horse	1964-11-07	41.32
Xavier Jameson	**Total: Item**	**Total: Date**	**5.08**
Xavier Jameson	**Total: Item**	**1967-01-24**	**5.08**
Xavier Jameson	**Plastic Plain Blue**	**Total: Date**	**5.08**
Xavier Jameson	Plastic Plain Blue	1967-01-24	5.08

In this example CUBE expanded 3 rows to 19 rows:

- 3 rows details
- 15 rows subtotals
- 1 row grand total

GROUP BY:

No GROUP BY
Date
Item
Item, Date
Customer
Customer, Date
Customer, Item
Customer, Item, Date

DML Statements (Query) Execution Logic

The clauses in the DML query statement are **ordered** as follows:

1. `SELECT`
2. `FROM... JOIN`
3. `WHERE`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

Behind the scene SQL Server is **executing** the DML query statement in a **different order**:

Step 1: `FROM... JOIN` - build the virtual recordset (VR) from one or multiple data sources (DS)
Step 2: `WHERE` - filter the VR
Step 3: `GROUP BY... HAVING` - group, aggregate and filter the grouped VR
Step 4: `SELECT` – build the resulting VR and create aliases
Step 5: `ORDER BY` – order the resulting VR and use the aliases, created in the `SELECT` statement

Step 1: `FROM Sales AS S JOIN Customers AS C ON...`

Virtual recordset

Customers		Sales	
CustomerID	CustomerName	CustomerID	LineTotal
5	Melanie Larson	5	21.35
3	Beverly	3	5.29
1	Anabel Larson	1	53.29
2	Anna Laurier	2	96.31
3	Beverly	3	18.16
4	John Smith	4	7.65
5	Melanie Larson	5	3.14

Resultset from `FROM`

Datasource for `WHERE`

Step 2: `WHERE C.CustomerName != 'Beverly'`

Virtual recordset

Customers		Sales	
CustomerID	CustomerName	CustomerID	LineTotal
5	Melanie Larson	5	21.35
1	Anabel Larson	1	53.29
2	Anna Laurier	2	96.31
4	John Smith	4	7.65
5	Melanie Larson	5	3.14

`CustomerName` `'Beverly'` is excluded

Resultset from `WHERE`

Datasource for `GROUP BY`

Step 3: `GROUP BY C.CustomerID...`

Virtual recordset

Customers		Sales	
CustomerID	CustomerName	CustomerID	LineTotal
1	Anabel Larson	1	53.29
2	Anna Laurier	2	96.31
4	John Smith	4	7.65
5	Melanie Larson	5	**24.49**

`LineTotal` `for CustomerID = 5 is summed`

Resultset from `GROUP BY`

Datasource for `HAVING`

Step 4: `HAVING SUM(S.LineTotal) >= 20`

Virtual recordset

Customers		Sales	
CustomerID	CustomerName	CustomerID	LineTotal
1	Anabel Larson	1	53.29
2	Anna Laurier	2	96.31
5	Melanie Larson	5	24.49

`CustomerID = 4 is excluded`

Resultset from `HAVING`

DML Statements (Query) Execution Logic

Resultset from HAVING

Step 5: SELECT CustomerName, LineTotal AS LT

Resulting recordset

Customers	Sales
Customer-Name	LT
Anabel Larson	53.29
Anna Laurier	96.31
Melanie Larson	24.49

Resultset from SELECT

Datasource for ORDER BY

Step 6: ORDER BY LT DESC

Resulting recordset

Customers	Sales
Customer-Name	LT
Anna Laurier	96.31
Anabel Larson	53.29
Melanie Larson	24.49

Subquery

Inner Outer

Subquery is a **query** (statement), **nested in another query** (statement).
During the execution, the subquery creates a virtual recordset (VR) a.k.a. derived (or virtual) table.
This recordset is the data source for the outer query (statement).

Scope of the subquery

The VR, created by the subquery is **alive during the execution** of the outer statement.

The subquery is **executed on every row** of the virtual recordset, created by the `FROM` clause (the outer query) and this can lead to **bad performance**.

We can use **subquery** in **`SELECT`**, **`FROM`**, **`WHERE`** and **`HAVING`** **clauses**.

Data sources for the examples below:

Customers

CustomerID	First-Name	Last-Name	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org
6	Melanie	Larson	melanie.larson@customer4.biz
7	Xavier	Jameson	xavier.garcía@customer6.com
8	Zak	Smith	NULL

Sales

SaleID	CustomerID	ItemID	DateOf-Sale	Quan-tity	Price
1	2	1953	1966-03-05	53	2.84
2	2	5523	1967-09-18	55	127.19
3	2	9375	1965-10-06	6	0.43
4	3	2043	1967-01-24	4	1.27
5	3	5293	1966-12-11	77	55.12
6	3	8613	1963-04-13	22	23.41
7	4	5164	1967-05-15	2	1.56
8	6	8534	1962-10-13	1	45.88
9	7	1352	1965-10-14	89	0.84
10	7	6223	1969-01-17	3	4.89

Subquery in the `SELECT` clause

The subquery in the `SELECT` clause has to return one value (one row and one column). If it returns more than one value, an error message "Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression." is returned:

```
SELECT
  (
   SELECT CONCAT(FirstName, ' ', LastName) AS CustomerName
   FROM LearnSQLServerIntuitively.dbo.Customers
   WHERE Email IS NULL
  ) AS CustomerName
  , (S.Quantity * S.Price) AS LineTotal
```

This subquery returns more than 1 value (two rows and one column)

CustomerName
Beverly
Zak Smith

```
FROM LearnSQLServerIntuitively.dbo.Sales AS S
GO
```

The statement below uses a subquery in the `SELECT` clause to show the **average sales for all the customers** next to the individual sales:

```
• SELECT
    C.FirstName
    , C.LastName
    , SUM(S.Quantity * S.Price) AS Sum_Sales
    , (
    • SELECT AVG(Quantity * Price)
    • FROM LearnSQLServerIntuitively.dbo.Sales
    ) AS Avg_SalesForAllCustomers
• FROM
    LearnSQLServerIntuitively.dbo.Customers AS C
    JOIN LearnSQLServerIntuitively.dbo.Sales AS S
      ON C.CustomerID = S.CustomerID
• GROUP BY
    C.FirstName
    , C.LastName
• ORDER BY
    C.FirstName
    , C.LastName;
  GO
```

• Outer statement

• Inner statement (subquery) Returns 1 value (one row and one column)

(No column name)
1205.132

FirstName	LastName	Sum_ Sales	Avg_SalesForAll-Customers
Anna	Laurier	7148.55	1205.132
Beverly	NULL	4764.34	1205.132
John	Smith	3.12	1205.132
Melanie	Larson	45.88	1205.132
Xavier	Jameson	89.43	1205.132

Subquery in the `FROM` clause

The subquery in the `FROM` clause allows us to extract the portion of the data that we need (from multiple data sources, aggregated, etc) and `JOIN` it to the other data sources in the `FROM` clause.

The statement below returns the customers **without `Email`** where their **sales (if any) are on and after January 01, 1965**:

```
SELECT
  C.FirstName
  , C.LastName
  , SUM(S.Quantity * S.Price) AS Sum_Sales
FROM
  (
    SELECT
      CustomerID
```

Bind alias

This subquery creates a recordset with the customers without **`Email`**

CustomerID	FirstName	LastName
3	Beverly	NULL
8	Zak	Smith

```
    , FirstName
    , LastName
  FROM LearnSQLServerIntuitively.dbo.Customers
  WHERE Email IS NULL
) AS C
LEFT JOIN LearnSQLServerIntuitively.dbo.Sales AS S
  ON C.CustomerID = S.CustomerID
  AND S.DateOfSale >= '1965-01-01'
GROUP BY
  C.FirstName
  , C.LastName;
GO
```

Data source for the subquery

Assign alias

Filter in the ON clause (not in the WHERE clause) to avoid the conversion of the JOIN from LEFT to INNER

FirstName	LastName	Sum_Sales
Beverly	NULL	4249.32
Zak	Smith	NULL

The subquery is LEFT JOINed to the **Sales** table to SELECT the customers with no sales.

When we LEFT JOIN, we filter the right data source in the ON clause. If we apply the filter in the WHERE clause, the JOIN is transformed to an INNER JOIN.

LEFT JOIN on page 130

Subquery in the WHERE clause

To select the **3 customers with highest sales**:

```
SELECT
  FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE
  CustomerID IN
  (
    SELECT TOP 3 CustomerID
    FROM LearnSQLServerIntuitively.dbo.Sales
    GROUP BY CustomerID
    ORDER BY SUM(Quantity * Price) DESC
  );
GO
```

FirstName	LastName
Anna	Laurier
Beverly	NULL
Xavier	Jameson

The subquery creates a list of **CustomerID** to be filtered

CustomerID
2
3
7

Subquery in the HAVING clause

```sql
SELECT
  C.FirstName
  , C.LastName
  , SUM(Quantity * Price) AS Sum_Sales
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
GROUP BY
  C.FirstName
  , C.LastName
HAVING
  SUM(Quantity * Price) >
  (
    SELECT AVG(Quantity * Price)
    FROM LearnSQLServerIntuitively.dbo.Sales
  );
GO
```

Select the customers where **total** sales is **greater than** the **average** sales:

FirstName	LastName	Sum_Sales
Beverly	NULL	4764.34
Anna	Laurier	7148.55

The subquery SELECTs the average sales for all the customers

(No column name)
1205.132

Correlated subquery

The correlated subquery is linked to the outer query. It correlates with objects in the FROM clause.
We can use **correlated subquery** in the **SELECT**, **WHERE** and **HAVING clauses**.

Let's add one more table that stores the results of the marketing calls to the data sources, for the example:

Marketing-CallID	Customer-ID	CallDateTime	CallDuration-InMin	Notes
1	1	1966-12-18 07:42:15.503	2	On vacation. Call on Jan 08
2	1	1967-01-08 17:27:18.140	18	Not interested
3	5	1966-12-10 22:31:00.983	4	Needs red sweater. Will be in warehouse on Dec 23
4	5	1966-12-23 01:15:08.790	7	Will take a look in our store on Queen street
5	5	1967-04-28 13:02:17.363	1	Spent the Christmas budget. Already bought red sweater
6	3	1967-01-24 10:12:48.983	23	Will take a look at the golden rings
7	3	1967-04-14 14:23:08.247	23	Positive feedback for ItemID 8613
8	4	1967-05-12 04:57:16.210	3	Has a dog and needs a new leash
9	7	1969-01-17 09:12:43.260	7	Very happy with the old purchase. Will visit our store on St. Eduard Str.
10	7	1969-01-19 10:08:13.320	7	Very happy with the old purchase. Will visit our store on St. Eduard Str.

Correlated subquery in the SELECT clause

We can query the MarketingCalls table using subqueries in the SELECT clause to SELECT the count and the duration of the marketing calls for each customer:

```
SELECT
  C.FirstName
  , C.LastName
  , (
    SELECT AVG(Quantity * Price)
    FROM LearnSQLServerIntuitively.dbo.Sales
  ) AS Avg_SalesAllCustomers
  , SUM(S.Quantity * S.Price) AS Sum_Sales
  , (
    SELECT COUNT(MarketingCallID)
    FROM LearnSQLServerIntuitively.dbo.MarketingCalls AS MC
    WHERE MC.CustomerID = C.CustomerID
  ) AS Count_MarketingCalls
  , (
    SELECT SUM(CallDurationInMin)
    FROM LearnSQLServerIntuitively.dbo.MarketingCalls AS MC
    WHERE MC.CustomerID = C.CustomerID
  ) AS Sum_CallDurationInMin
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
GROUP BY
  C.FirstName
  , C.LastName
  , C.CustomerID
ORDER BY
  C.FirstName
  , C.LastName;
GO
```

Not correlated subquery

Correlation (link to an object in the FROM clause)

First-Name	Last-Name	Avg_SalesAll-Customers	Sum_Sales	Count_Mar-ketingCalls	Sum_CallDu-rationInMin
Anna	Laurier	1205.132	7148.55	0	NULL
Beverly	NULL	1205.132	4764.34	2	46
John	Smith	1205.132	3.12	1	3
Melanie	Larson	1205.132	45.88	0	NULL
Xavier	Jameson	1205.132	89.43	2	14

Correlated subquery in the WHERE clause

To SELECT customers with total sales greater than 2000:

Subquery with WHERE EXISTS

```
SELECT
  C.FirstName
  , C.LastName
FROM LearnSQLServerIntuitively.dbo.Customers AS C
WHERE
  EXISTS
  (
    SELECT 1
    FROM LearnSQLServerIntuitively.dbo.Sales AS S
    WHERE C.CustomerID = S.CustomerID
    GROUP BY CustomerID
    HAVING SUM(S.Quantity * S.Price) >= 2000
  );
GO
```

On every row of the execution of the outer statement, **CustomerID** is linked and if the filter condition is met (total sales equal to or is greater than 2000), the customer is selected (it EXISTS)

FirstName	LastName
Anna	Laurier
Beverly	NULL

EXISTS returns True if the subquery contains one or multiple rows.
We actually don't need to SELECT any value from the subquery in the WHERE EXISTS filtering condition. We just need to know if the subquery contains a row or not.

```
SELECT
  C.FirstName
  , C.LastName
FROM LearnSQLServerIntuitively.dbo.Customers AS C
WHERE
  CustomerID IN
  (
    SELECT CustomerID
    FROM LearnSQLServerIntuitively.dbo.Sales AS S
    WHERE C.CustomerID = S.CustomerID
    GROUP BY CustomerID
    HAVING SUM(S.Quantity * S.Price) >= 2000
  );
GO
```

Subquery with WHERE ColumnName IN

Filter

FirstName	LastName
Anna	Laurier
Beverly	NULL

Correlated subquery in the HAVING clause

To SELECT the **total sales** for the customers that have **single sale equal to** or **greater than 150**:

```
SELECT
  C.FirstName
  , C.LastName
  , SUM(S1.Quantity * S1.Price) AS Sum_Sales
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S1
    ON C.CustomerID = S1.CustomerID
GROUP BY
  C.CustomerID
  , S1.CustomerID
  , C.FirstName
  , C.LastName
HAVING
  S1.CustomerID IN
  (
    SELECT CustomerID
    FROM LearnSQLServerIntuitively.dbo.Sales AS S2
    WHERE
      C.CustomerID = S2.CustomerID
      AND (S2.Quantity * S2.Price) >= 150
  )
GO
```

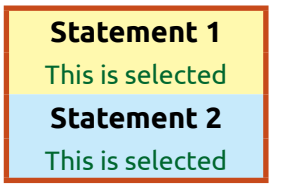
FirstName	LastName	Sum_Sales
Anna	Laurier	7148.55
Beverly	NULL	4764.34

UNION

UNION creates a recordset by appending the result from one statement to the result of another statement.

Both statements have to:

- Have the same number of columns
- Have the same collation and the same or implicitly convertible data type of the matching columns. If the data type can't be implicitly converted we convert it manually (explicit conversion)




```
SELECT Column1, Column2, Column3
FROM TableName1

UNION

SELECT Column1, Column2, Column3
FROM TableName2;
GO
```

Statement 1

Statement 2

Customers

First-Name	Last-Name	Email	CustomerCode
Anabel	Larson	anabel.larson@customer5.info	34
Anna	Laurier	anna.laurier@customer2.net	863
Beverly	NULL	NULL	23
John	Smith	john.smith@customer1.com	943
John	Smith	john.smith@customer3.org	72
Melanie	Larson	melanie.larson@customer4.biz	7565
Xavier	Jameson	xavier.garcía@customer6.com	4
Zak	Smith	NULL	425

Employees

First-Name	Last-Name	Email	EmployeeCode
Anabel	Larson	anabel.larson@customer5.info	E1-465
Anna	Laurier	anna.laurier@customer2.net	67 C22
John	Smith	john.smith@customer1.com	Fa 3 58B

Let's say we have external and internal (employees) customers. We may need to UNION both tables and output the result to a report.

```
SELECT
  FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.Customers

UNION

SELECT
  FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.Employees;
GO
```

FirstName	LastName
Anabel	Larson
Anna	Laurier
Beverly	NULL
John	Smith
Melanie	Larson
Xavier	Jameson
Zak	Smith

UNION acts like SELECT DISTINCT - no duplicates in the resulting recordset.

UNION ALL

UNION ALL returns duplicates:

Can't be implicitly converted to Character and we **convert it manually**.

```
SELECT
  FirstName
  , LastName
  , Email
  , CONVERT(VARCHAR(10), CustomerCode) AS Code
FROM LearnSQLServerIntuitively.dbo.Customers

UNION ALL

SELECT
  FirstName
  , LastName
  , Email
  , EmployeeCode AS Code
FROM LearnSQLServerIntuitively.dbo.Employees;
GO
```

First-Name	Last-Name	Email	Code
Anabel	Larson	anabel.larson@customer5.info	34
Anna	Laurier	anna.laurier@customer2.net	863
Beverly	NULL	NULL	23
John	Smith	john.smith@customer1.com	943
John	Smith	john.smith@customer3.org	72
Melanie	Larson	melanie.larson@customer4.biz	7565
Xavier	Jameson	xavier.garcía@customer6.com	4
Zak	Smith	NULL	425
Anabel	Larson	anabel.larson@customer5.info	E1-465
Anna	Laurier	anna.laurier@customer2.net	67 C22
John	Smith	john.smith@customer1.com	Fa 3 58B

An **alias** is **not needed in the bottom statement**. We add it to be able to execute the bottom statement separately at the time of the development and verify the column names.

The performance of UNION is slower, compared to UNION ALL, because it cleans duplicates (additional operation).

UNION and ORDER BY

```
SELECT
  'Top Table' AS [Type]
  , FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE
  LEFT(FirstName, 1) BETWEEN 'J' AND 'M'
  OR FirstName IN ('Anna', 'Anabel')

UNION ALL

SELECT
  'Bottom Table' AS [Type]
  , FirstName
  , LastName
```

Add **identification column** to mark the source of the data and identify it in future statement(s)

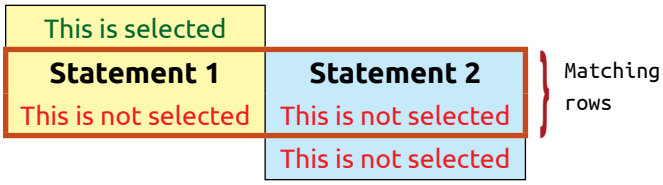
Type	FirstName	LastName
Top Table	Anabel	Larson
Bottom Table	Anabel	Larson
Bottom Table	Anna	Laurier
Top Table	Anna	Laurier
Top Table	John	Smith
Top Table	John	Smith
Bottom Table	John	Smith
Top Table	Melanie	Larson

```
FROM LearnSQLServerIntuitively.dbo.Employees
ORDER BY
  FirstName
  , LastName;
GO
```

ORDER BY is at the end of the bottom statement and orders the UNIONed resulting recordset (not the last statement).

EXCEPT

We can create one statement that selects data and another statement to exclude the matching data from the first statement. The combining keyword in this case is EXCEPT. Nothing from the bottom statement is selected. It just determines which rows from the top statement are excluded.




```
SELECT
  FirstName
  , LastName
  , Email
FROM LearnSQLServerIntuitively.dbo.Customers

EXCEPT

SELECT
  FirstName
  , LastName
  , Email
FROM LearnSQLServerIntuitively.dbo.
CustomersWithExceptionalities;
GO
```

● Match and excluded

Customers

FirstName	LastName	Email
● Anabel	Larson	anabel.larson@customer5.info
● Anna	Laurier	anna.laurier@customer2.net
● Beverly	NULL	NULL
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org
Melanie	Larson	melanie.larson@customer4.biz
Xavier	Jameson	xavier.garcía@customer6.com
Zak	Smith	NULL

CustomersWithExceptionalities

FirstName	LastName	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
John	Smith	john.smith@customer1.com

FirstName	LastName	Email
Beverly	NULL	NULL
John	Smith	john.smith@customer3.org
Melanie	Larson	melanie.larson@customer4.biz
Xavier	Jameson	xavier.garcía@customer6.com
Zak	Smith	NULL

EXCEPT excludes the data that matches in the virtual recordsets.

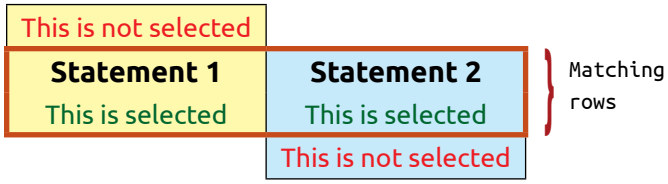
If we don't select column **Email**, the rows for John Smith are not unique anymore and are excluded:

```sql
SELECT
  FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.Customers

EXCEPT

SELECT
  FirstName
  , LastName
FROM LearnSQLServerIntuitively.dbo.
CustomersWithExceptionalities;
GO
```

FirstName	LastName
Beverly	NULL
Melanie	Larson
Xavier	Jameson
Zak	Smith

INTERSECT

To select the rows that exist in the both statements, we use `INTERSECT`.




```sql
SELECT
  FirstName
  , Email
FROM LearnSQLServerIntuitively.dbo.Customers

INTERSECT

SELECT
  FirstName
  , Email
FROM LearnSQLServerIntuitively.dbo.
CustomersWithExceptionalities;
GO
```

Customers

FirstName	LastName	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	NULL	NULL
John	Smith	john.smith@customer1.com
John	Smith	john.smith@customer3.org
Melanie	Larson	melanie.larson@customer4.biz
Xavier	Jameson	xavier.garcía@customer6.com
Zak	Smith	NULL

CustomersWithExceptionalities

FirstName	LastName	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
John	Smith	john.smith@customer1.com

Rows, matching in both recordsets

FirstName	Email
Anabel	anabel.larson@customer5.info
Anna	anna.laurier@customer2.net
John	john.smith@customer1.com

PIVOT

Pivot is the central point of rotation. When we pivot data, we rotate, reorganize and aggregate it.
When we pivot a table, we create a matrix from the table. The matrix allows us to show a value, belonging to two groups of attributes.
The pivot table is also known as Tablix or Cross-Tab.
The pivoting is generally used for presentation of data, related to reporting and analysis purposes.

Sales

SaleID	CustomerID	ItemID	Period	SalesValue
3	1	1	1966-Q1	8.99
16	1	1	1966-Q1	53.24
10	1	1	1966-Q2	81.09
12	1	1	1966-Q3	81.82
15	2	1	1966-Q1	39.37
2	2	1	1966-Q2	86.92
14	2	2	1966-Q1	93.12
4	2	2	1966-Q3	40.55
8	2	3	1966-Q2	18.37
11	3	1	1966-Q1	12.36
7	3	1	1966-Q3	31.22
1	3	1	1966-Q3	21.44
6	3	2	1966-Q2	13.33
9	3	2	1966-Q3	86.42
5	3	2	1966-Q3	68.46
13	3	3	1966-Q2	29.39
17	3	3	1966-Q2	84.72

Attributes *Pivot column* *Values*

This **groups** (CustomerID, ItemID, Period) is be **aggregated** with SUM(SalesValue)

Martix

	Columns
Rows	Values

Columns **CustomerID** and **ItemID** are presented on the **rows**.

Column **Period** is the pivot - transforms the data from rows to **columns**.

SalesValue column is the values, aggregated in CustomerID, ItemID (rows), Period (columns) groups.

```
SELECT
  P.CustomerID, P.ItemID
  , P.[1966-Q1], P.[1966-Q2], P.[1966-Q3]
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  PIVOT
  (
    SUM(SalesValue)
    FOR Period IN ([1966-Q1], [1966-Q2], [1966-Q3])
  ) AS P
ORDER BY CustomerID, ItemID;
GO
```

To pivot a table, we add the PIVOT relational operator in the FROM clause.

Column **CustomerID** is not selected, but as we pivot the table directly, it is in the virtual recordset that is pivoted

CustomerID	ItemID	1966-Q1	1966-Q2	1966-Q3
1	1	8.99	NULL	NULL
1	1	NULL	81.09	NULL
1	1	NULL	NULL	81.82
1	1	53.24	NULL	NULL
2	1	NULL	86.92	NULL
2	1	39.37	NULL	NULL
2	2	NULL	NULL	40.55
2	2	93.12	NULL	NULL
2	3	NULL	18.37	NULL
3	1	NULL	NULL	21.44
3	1	NULL	NULL	31.22
3	1	12.36	NULL	NULL
3	2	NULL	NULL	68.46
3	2	NULL	13.33	NULL
3	2	NULL	NULL	86.42
3	3	NULL	29.39	NULL
3	3	NULL	84.72	NULL

The result is not exactly as we expect. It is **not aggregated**. What this means is that the recordset pivoted (table **Sales**) includes a column that is not needed in the pivoting (column **SaleID**). To get the result we need, we have to select only the data that we need to PIVOT in the **subquery**:

Column	PIVOT **as**
CustomerID, ItemID	Rows
Period	Columns
SalesValue	Values

```
SELECT
  P.CustomerID, P.ItemID
  , P.[1966-Q1], P.[1966-Q2], P.[1966-Q3]
FROM
  (
    SELECT DISTINCT CustomerID, ItemID, Period, SalesValue
    FROM LearnSQLServerIntuitively.dbo.Sales
  ) AS S
  PIVOT
  (
    SUM(SalesValue)
    FOR Period IN ([1966-Q1], [1966-Q2], [1966-Q3])
  ) AS P
ORDER BY CustomerID, ItemID;
GO
```

Rows

Columns

In a **subquery** select only the columns for **rows** (CustomerID, ItemID), **columns** (Period) and **values** (SalesValue)

Pivoting the above data source (the subquery, aliased with S)

Values (Aggregated)

Only these values are transformed into columns (PIVOTed)

* Aggregated

CustomerID	ItemID	1966-Q1	1966-Q2	1966-Q3
1	1	62.23 *	81.09	81.82
2	1	39.37	86.92	NULL
2	2	93.12	NULL	40.55
2	3	NULL	18.37	NULL
3	1	12.36	NULL	52.66 *
3	2	NULL	13.33	154.88 *
3	3	NULL	114.11	NULL

After we PIVOT the table, we can easily compare the sales in the each period side by side.

UNPIVOT

UNPIVOT is the opposite of PIVOT. It transforms the data from columns to rows. In the example below, we use the table that we just PIVOTed.

```
SELECT
  CustomerID
  , ItemID
  , Period
  , SaleValue
FROM
  (
    SELECT
      CustomerID, ItemID
      , [1966-Q1], [1966-Q2], [1966-Q3]
    FROM LearnSQLServerIntuitively.dbo.Sales
  ) AS P
  UNPIVOT
  (
    SaleValue
    FOR Period IN ([1966-Q1], [1966-Q2], [1966-Q3])
  ) AS U
ORDER BY
  CustomerID
  , ItemID
  , Period;
GO
```

Subquery to select only the data that we need to UNPIVOT

Combine multiple columns into a **new column** (**Period**)

The UNPIVOTed result is:

Custo-merID	ItemID	Period	Sales-Value
1	1	1966-Q1	62.23 *
1	1	1966-Q2	81.09
1	1	1966-Q3	81.82
2	1	1966-Q1	39.37
2	1	1966-Q2	86.92
2	2	1966-Q1	93.12
2	2	1966-Q3	40.55
2	3	1966-Q2	18.37
3	1	1966-Q1	12.36
3	1	1966-Q3	52.66 *
3	2	1966-Q2	13.33
3	2	1966-Q3	154.88 *
3	3	1966-Q2	114.11

* Can’t undo the aggregation

UNPIVOT is not able to fully reverse the data as it was before the PIVOTing. We can’t undo the aggregation.

IF... ELSE are keywords of the **Control-of-Flow Language** which manipulates the execution of the code. **T-SQL** is executed **from top to bottom**. Based on a specified condition, we can skip the execution of a block of code.

Execution steps:

1. The IF condition is **verified**
2. If the verification returns **True**, the **block of code** between BEGIN and END keywords that **follows** is **executed**
3. ELSE keyword adds **another block of code** that is **executed** when the IF condition returns **False**. The ELSE statement is optional

The syntax is:

```
IF (Condition)
  BEGIN
    ... Statement 1 ...;
  END
ELSE
  BEGIN
    ... Statement 1 ...;
    ... Statement 2 ...;
  END
GO
```

Condition that is **verified** and returns **boolean expression** (**True** or **False**)

Block of code, executed if the **verification** of the condition returns **True**. One or multiple statements.

Block of code, executed if the verification of the condition returns **False** or NULL (unknown). One or multiple statements.

If the block contains **one statement**, BEGIN and END can be **omitted**.

Always use BEGIN and END, regardless if the block contains one or more than one statements.

```
IF (7 > 17)
  BEGIN
    PRINT 'This BEGIN... END block is executed when condition is True';
    PRINT '7 is greater that 17';
  END
GO
```

The **condition** is not met (**False**) and the block of T-SQL that follows is **not executed**

IF... ELSE

The **block of code**, defined by ELSE is **executed** if the verification of the condition returns **False**.

```
IF (DATENAME(DW, GETDATE()) = 'Saturday')
  BEGIN PRINT 'Today is Saturday'; END
ELSE
```

GETDATE returns the **date and time** now
DATENAME returns the **name of the day in the week**

If **one statement** then BEGIN and END **on the same line**

Formating the code

```
  BEGIN
    PRINT 'Block of code when the condition is FALSE';
    PRINT 'Today is not Saturday';
  END
GO
```

Two statements

The statement is executed **not** on **Saturday** and returns this:

```
Block of code when the condition is FALSE
Today is not Saturday
```

Nested IF

We can verify one condition in case when another condition is met by nesting one IF condition into another.

```
IF (DATENAME(DW, GETDATE()) = 'Tuesday')
  BEGIN
    IF ((SELECT DATEPART(HOUR, GETDATE())) = 15)
      BEGIN PRINT 'Today is Tuesday and the hour is 15'; END
    ELSE
      BEGIN PRINT 'Today is Tuesday and the hour is not 15'; END
  END
ELSE
  BEGIN PRINT 'Today is not Tuesday'; END
GO
```

DATEPART with first parameter = HOUR extracts the **hour** from the time

Executed on Tuesday at 11h 27 min and results to:

```
Today is Tuesday and the hour is not 15
```

In this example if the condition is verified to False, ELSE, including nested IF is executed:

```
IF (7 > 17)
  BEGIN PRINT '(7 > 17) is True'; END
ELSE
  BEGIN
    IF (8 > 17)
      BEGIN PRINT '(8 > 17) is True'; END
    ELSE
      BEGIN PRINT '(8 > 17) is False'; END
  END
GO
```

Independent IFs

```
IF (7 > 17)
  BEGIN PRINT 'First IF --> True'; END
ELSE
  BEGIN PRINT 'First IF --> False'; END
```

```
IF (7 < 17)
  BEGIN PRINT 'Second IF --> True'; END
ELSE
  BEGIN PRINT 'Second IF --> False'; END
GO
```

```
First IF --> False
Second IF --> True
```

As they are not nested, both conditions are executed consecutively and independently.

IF EXISTS

The statement in the condition creates a recordset. If a row exists in the recordset, the verification of the condition returns True.

Customers

FirstName	LastName	Email
Anabel	Larson	anabel.larson@customer5.info
Anna	Laurier	anna.laurier@customer2.net
Beverly	NULL	NULL

If a customer with a specified **Email** exists, a message is printed:

```
IF EXISTS
(
  SELECT 1
  FROM LearnSQLServerIntuitively.dbo.Customers
  WHERE Email = 'anna.laurier@customer3.net'
)
  BEGIN SELECT 'anna.laurier@customer3.net EXISTS' AS Result; END
ELSE
  BEGIN SELECT 'anna.laurier@customer3.net does not EXISTS' AS Result; END
GO
```

The reason to select 1 is because We don't need any actual value. We just need to know if a row exists in the recordset

Result
anna.laurier@customer3.net does not EXISTS

RETURN

Terminates the execution of the code (the rest of the code after RETURN is not executed).

```
IF (7 > 17)
  BEGIN SELECT 'True' AS Result; END
ELSE
  BEGIN
    SELECT 'False' AS Result;
    RETURN;
  END

SELECT 'This is not executed, because RETURN is reached';
GO
```

The execution of the code is **terminated here**

The code **after** RETURN is **not executed**

Result
False

Conditional Execution
IIF

Let's pick the following tables for the IIF example:

Sales

SaleID	Custo-merID	ItemID	IsItemDis-counted	Quan-tity
1	1	11	1	52
2	2	22	1	2
3	3	11	1	62
4	1	22	0	552
5	1	11	0	1

ItemsStandardDiscounted

ItemID	Discount-edItemID
11	111
22	222

Items

ItemID	Price
11	5.23
22	15.88
111	4.99
222	14.26

1. Table **Sales** contains the sales by customer and item. Column **IsItemDiscounted** determines how we manipulate the data and how we JOIN the tables
2. Table **ItemsStandarsDiscounted** is intermediate and links the standard item (the ones that we usually sell) to the discounted item (the same item with reduced price for one or another reason)
3. Table **Items** contains the attributes for the items. For the purposes of this example we need only Price

In the following scenarios we JOIN:

- Sales to ItemsStandarsDiscounted when we need the special routing
- Sales to Items for the cases when we manipulates the discounted items differently

We can't implement the IF condition in DML statement's clause (SELECT, FROM, WHERE, GROUP BY, ORDER BY) directly:

```
SELECT
  S.CustomerID
  , S.ItemID
  , S.IsItemDiscounted
  , S.Quantity
  , I.Price
  , IF (S.IsItemDiscounted = 0)
      BEGIN (S.Quantity * I.Price) END
    ELSE
      BEGIN (S.Quantity * (I.Price * 0.85)) END
  AS SalesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.Items AS I
    ON S.ItemID = I.ItemID;
GO
```

IF condition

```
Incorrect syntax near the keyword 'IF'.
Incorrect syntax near 'S'.
Incorrect syntax near 'S'.
```

To accomplish this task, we need to use the built-in function IIF
The syntax is:

IIF(*Condition*, *WhenTrue*, *WhenFalse*);

- *Condition* - validation that returns True or False
- *WhenTrue* - value or expression, that the function returns when the condition returns True
- *WhenFalse* - value or expression, that the function returns when the condition returns True

IIF in the SELECT clause

The IIF condition verifies if the item is discounted and calculates the standard or the discounted (85%) sales value:

```
SELECT
  S.CustomerID
  , S.ItemID
  , S.IsItemDiscounted
  , S.Quantity
  , I.Price
  , IIF((S.IsItemDiscounted = 0), (S.Quantity * I.Price), (S.Quantity * (I.Price * 0.85))) AS
SalesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.Items AS I
    ON S.ItemID = I.ItemID;
GO
```

CustomerID	ItemID	IsItemDiscounted	Quantity	Price	SalesValue	
1	11	1	52	5.23	231.166000	
2	22	1	2	15.88	26.996000	← (Quantity * (Price * 0.85))
3	11	1	62	5.23	275.621000	
1	22	0	552	15.88	8765.760000	← (Quantity * Price)
1	11	0	1	5.23	5.230000	

SalesValue is calculated differently, based on the flag in **IsItemDiscounted** column.

IIF in the FROM clause

The conditional executions in the FROM... JOIN (ON) clause gives us the chance to create a **zig-zag** JOIN.
This means that, based on **condition**, we link to **one or another** columns.

```
SELECT
  S.CustomerID
  , S.ItemID
  , S.IsItemDiscounted
```

```
  , S.Quantity
  , ISD.ItemID
  , ISD.DiscountedItemID
  , I.ItemID
  , I.Price
  , (S.Quantity * I.Price) AS SavesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.ItemsStandardDiscounted AS ISD
    ON S.ItemID = ISD.ItemID
  JOIN LearnSQLServerIntuitively.dbo.Items AS I
    ON IIF((S.IsItemDiscounted = 0), ISD.ItemID, ISD.DiscountedItemID) = I.ItemID;
GO
```

Zig-zag JOINing

Sales				ItemsStandardDiscounted		Items		IIF()
Custome-rID	**ItemID**	**IsItemDis-counted**	**Quan-tity**	**ItemID**	**Discounte-dItemID**	**ItemID**	**Price**	**Saves-Value**
1	11	1	52	11	111	111	4.99	259.48
2	22	1	2	22	222	222	14.26	28.52
3	11	1	62	11	111	111	4.99	309.38
1	22	0	552	22	222	22	15.88	8765.76
1	11	0	1	11	111	11	5.23	5.23

In the first row `ItemID` = 11 is linked to `ItemID` = 11. Hense **`IsItemDiscounted`** is 1, `DiscountedItemID` = 111 is linked to `ItemID` = 111 and the price of `ItemID` = 111 (4.99) is used in the calculation (52 * 4.99 = 259.48)

In the last row `ItemID` = 11 is linked to `ItemID` = 11. Hense **`IsItemDiscounted`** is 0, `ItemID` = 11 is linked to `ItemID` = 11 and the price of `ItemID` = 11 (5.23) is used in the calculation (1 * 5.23 = 5.23)

`IIF` in the `WHERE` clause

We can filter on the calculated column **`SalesValue`** and select the rows where **`SalesValue`** is **greater than 200**:

```
SELECT
  S.CustomerID
  , S.ItemID
  , S.IsItemDiscounted
  , S.Quantity
  , I.Price
  , IIF((S.IsItemDiscounted = 0), (S.Quantity
* I.Price), (S.Quantity * (I.Price * 0.85)))
AS SalesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.Items AS I
    ON S.ItemID = I.ItemID
WHERE IIF((S.IsItemDiscounted = 0), (S.Quantity *
I.Price), (S.Quantity * (I.Price * 0.85))) > 200;
GO
```

CustomerID	ItemID	IsItemDiscounted	Quantity	Price	SalesValue
1	11	1	52	5.23	231.166000
3	11	1	62	5.23	275.621000
1	22	0	552	15.88	8765.760000

(52 * 5.23 * 0.85) = 231.166

(552 * 15.88) = 8765.76

IIF in the GROUP BY clause

We can create a column, based on a condition and use it in a group.
COUNT the customers for item:

```
SELECT
  IIF((S.IsItemDiscounted = 0), ISD.ItemID, ISD.DiscountedItemID) AS ItemID
  , COUNT(DISTINCT CustomerID) AS Count_Customers
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.ItemsStandardDiscounted AS ISD
    ON S.ItemID = ISD.ItemID
GROUP BY IIF((S.IsItemDiscounted = 0), ISD.ItemID, ISD.DiscountedItemID);
GO
```

The same as in the SELECT clause

ItemID	Count_Customers
11	1
22	1
111	2
222	1

IIF in the HAVING clause

Filter the aggregated recordset where **SalesValue** is **greater than 250**:

```
SELECT
  S.CustomerID
  , S.ItemID
  , SUM(IIF((S.IsItemDiscounted = 0), (S.Quantity *
I.Price), (S.Quantity * (I.Price * 0.85)))) AS SalesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.Items AS I
    ON S.ItemID = I.ItemID
GROUP BY
  S.CustomerID
  , S.ItemID
HAVING SUM(IIF((S.IsItemDiscounted = 0), (S.Quantity *
I.Price), (S.Quantity * (I.Price * 0.85)))) > 250;
GO
```

CustomerID	ItemID	SalesValue
1	22	8765.760000
3	11	275.621000

Conditional Execution
CASE

The CASE expression is an extension of the logic of the IF condition. CASE combines multiple conditions into one.

Structure

- CASE and END distinct the CASE expression from the rest of the code
- WHEN defines the conditions to be verified
- THEN defines the code that is executed if the WHEN condition evaluates to True
- ELSE defines the code that is executed if none of the WHEN conditions evaluates to True

Execution

1. The WHEN conditions are verified in the order that they appear
2. When the condition verification returns False, the execution jumps to the next condition
3. When the condition's verification is True, the code defined with the keyword THEN is executed and the execution of the CASE terminates
4. If none of the conditions validates to True, the code defined with the keyword ELSE is executed

Sales

SaleID	CustomerName	ItemDescription	HasDis-count	SaleDate	Quan-tity	Price
1	Anabel Larson	Plastic Plain Blue	0	1969-01-17	3	4.89
2	Anabel Larson	Wooden Horse	1	1965-10-14	89	43.12
3	John Smith	Rag Doll 33 cm.	0	1966-03-05	53	7.52
4	John Smith	Plastic Plain Blue	0	1965-10-06	6	3.43
5	Xavier Jameson	Plastic Plain Blue	1	1968-01-24	4	1.27
6	Xavier Jameson	Rag Doll 33 cm.	1	1966-12-11	77	55.12
7	Xavier Jameson	Wooden Horse	1	1968-04-13	22	23.41

We can write the CASE expression in two variations: **simple** and **complex**.

We can calculate the discount, based on the boolean value in **HasDiscount** column with **simple** CASE expression:

```
SELECT
  CustomerName
  , ItemDescription
  , HasDiscount
  , SUM(Quantity * Price) AS SalesValue
  , CASE HasDiscount
    WHEN 1 THEN (SUM(Quantity * Price) * 0.85)
    ELSE SUM(Quantity * Price)
  END AS Case_Discount
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY
  CustomerName
```

Left operand
The operator is **equals to** (=)
Right operand

ColumnName, which's values are validated

If the value in **HasDiscount** equals to 1 (True) then multiply **SalesValue** by 0.85

If **all** the **conditions above** evaluate to False

```
  , ItemDescription
  , HasDiscount
ORDER BY
  CustomerName
  , ItemDescription;
GO
```

Customer-Name	ItemDescrip-tion	HasDis-count	Sales-Value	Case_Discount	
Anabel Larson	Plastic Plain Blue	0	14.67	14.670000	ELSE
Anabel Larson	Wooden Horse	1	3837.68	3262.028000	WHEN
John Smith	Plastic Plain Blue	0	20.58	20.580000	
John Smith	Rag Doll 33 cm.	0	398.56	398.560000	
Xavier Jameson	Plastic Plain Blue	1	5.08	4.318000	
Xavier Jameson	Rag Doll 33 cm.	1	4244.24	3607.604000	
Xavier Jameson	Wooden Horse	1	515.02	437.767000	

When we use the **complex** variation of the CASE expression, we can involve one or more condition like:
`WHEN ('A' = 'A' AND (13 > 7 OR ColumnName = 'Y')) THEN...`

To create groups by verifying the **year** in **SaleDate**:

```
SELECT
  CustomerName
  , ItemDescription
  , YEAR(SaleDate) AS Year_SaleDate
  , CASE
    WHEN YEAR(SaleDate) = @CurrentYear THEN 'Current Year'
    WHEN YEAR(SaleDate) = (@CurrentYear - 1) THEN 'Last Year'
    ELSE 'Older than last year'
  END AS Case_Year_SaleDate
  , SUM(Quantity * Price) AS SalesValue
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY
  CustomerName
  , ItemDescription
  , YEAR(SaleDate)
  , CASE
    WHEN YEAR(SaleDate) = @CurrentYear THEN 'Current Year'
    WHEN YEAR(SaleDate) = (@CurrentYear - 1) THEN 'Last Year'
    ELSE 'Older than last year'
  END
ORDER BY
  CustomerName
  , ItemDescription;
GO
```

Both operands and the operator are after WHEN

@CurrentYear = 1969

The same as in the SELECT clause

Customer-Name	ItemDescrip-tion	Year_ SaleDate	Case_Year_Sale-Date	Sales-Value
Anabel Larson	Plastic Plain Blue	1969	Cusrrent Year	14.67
Anabel Larson	Wooden Horse	1965	Older than last year	3837.68
John Smith	Plastic Plain Blue	1965	Older than last year	20.58
John Smith	Rag Doll 33 cm.	1966	Older than last year	398.56
Xavier Jameson	Plastic Plain Blue	1968	Last Year	5.08
Xavier Jameson	Rag Doll 33 cm.	1966	Older than last year	4244.24
Xavier Jameson	Wooden Horse	1968	Last Year	515.02

CASE in the FROM clause

We can use CASE in the FROM clause when we need to equalize values in the ON clause.
A simple example is joining two data sources where values in **Table1** are 0 and 1, but the corresponding values in **Table2** are 'Y' and 'N':

```
...
FROM
  Table1 AS T1
  JOIN Table1 AS T2
    ON CASE
      WHEN T1.HasDiscount = 1 THEN 'Y'
      ELSE 'N'
    END = T2.HasDiscount
...
```

CASE in the WHERE clause

Let's add one more table that contains the thresholds, related to the marketing strategies:

ItemDiscounts

ItemDis-countID	ItemID	Current-Year	Last-Year	OlderThan-LastYear
1	1	10.00	20.00	30.00
2	2	300.00	400.00	500.00
3	3	50.00	60.00	70.00

The **items** are on the rows and **three groups** are in the columns. We generated these groups in the last example.
Let's move the last example into a **subquery** and join the new table with the subquery:

```
SELECT
  S.CustomerName
  , S.ItemDescription
```

```
  , S.Year_SaleDate
  , S.Case_Year_SaleDate
  , S.SalesValue
  , ID.CurrentYear
  , ID.LastYear
  , ID.OlderThanLastYear
FROM
  (
```

Subquery (from the previous example) to create the **sales data**

CustomerName	ItemID	ItemDescription	Year_SaleDate	Case_Year_SaleDate	SalesValue
Anabel Larson	1	Plastic Plain Blue	1969	Current Year	14.67
Anabel Larson	3	Wooden Horse	1965	Older than last year	3837.68
John Smith	1	Plastic Plain Blue	1965	Older than last year	20.58
John Smith	2	Rag Doll 33 cm.	1966	Older than last year	398.56
Xavier Jameson	1	Plastic Plain Blue	1968	Last Year	5.08
Xavier Jameson	2	Rag Doll 33 cm.	1966	Older than last year	4244.24
Xavier Jameson	3	Wooden Horse	1968	Last Year	515.02

```
) AS S
  JOIN LearnSQLServerIntuitively.dbo.ItemDiscounts AS ID
    ON S.ItemID = ID.ItemID
WHERE
  S.SalesValue >=
  CASE
    WHEN S.Case_Year_SaleDate = 'Current Year' THEN ID.CurrentYear
    WHEN S.Case_Year_SaleDate = 'Last Year' THEN ID.LastYear
    ELSE ID.OlderThanLastYear
  END
ORDER BY
  S.CustomerName
  , S.ItemDescription;
GO
```

Zig-zag filtering

Customer-Name	ItemDescrip-tion	Year_ SaleDate	Case_Year_Sale-Date	Sales-Value	Current-Year	Last-Year	OlderThan-LastYear
Anabel Larson	Plastic Plain Blue	1969	Current Year	14.67	10.00	20.00	30.00
Anabel Larson	Wooden Horse	1965	Older than last year	3837.68	50.00	60.00	70.00
Xavier Jameson	Rag Doll 33 cm.	1966	Older than last year	4244.24	300.00	400.00	500.00
Xavier Jameson	Wooden Horse	1968	Last Year	515.02	50.00	60.00	70.00

We can use the same zig-zag logic in the `FROM` (`ON`) clause to `JOIN` on different columns, based on a condition.

CASE in the GROUP BY clause

When we use CASE in the GROUP BY clause, it has to be the same as the corresponding CASE in the SELECT clause. This is not mandatory. We may have a logic that SELECTs and GROUPs BY differently.

CASE in the HAVING clause

In the below example, we used CASE to add a **calculated column** (**Case_Discount**) to explain the simple CASE expression.
We can filter the **aggregated virtual recordset** on this column by adding CASE expression in the HAVING clause:

```
SELECT
  CustomerName
  , ItemDescription
  , HasDiscount
  , SUM(Quantity * Price) AS SalesValue
  , CASE HasDiscount
    WHEN 1 THEN (SUM(Quantity * Price) * 0.85)
    ELSE SUM(Quantity * Price)
  END AS Case_Discount
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY
  CustomerName
  , ItemDescription
  , HasDiscount
HAVING
  CASE HasDiscount
    WHEN 1 THEN (SUM(Quantity * Price) * 0.85)
    ELSE SUM(Quantity * Price)
  END <= 500
ORDER BY
  CustomerName
  , ItemDescription;
GO
```

CustomerName	ItemDescription	HasDiscount	SalesValue	Case_Discount
Anabel Larson	Plastic Plain Blue	0	14.67	14.670000
John Smith	Plastic Plain Blue	0	20.58	20.580000
John Smith	Rag Doll 33 cm.	0	398.56	398.560000
Xavier Jameson	Plastic Plain Blue	1	5.08	4.318000
Xavier Jameson	Wooden Horse	1	515.02	437.767000

CASE in the ORDER BY clause

We can order the resultset by moving the **rows with discount** at the **top**:

```
SELECT
  CustomerName
  , ItemDescription
  , HasDiscount
  , SUM(Quantity * Price) AS SalesValue
  , CASE HasDiscount
    WHEN 1 THEN (SUM(Quantity * Price) * 0.85)
    ELSE SUM(Quantity * Price)
  END AS Case_Discount
FROM LearnSQLServerIntuitively.dbo.Sales
GROUP BY
  CustomerName
```

```
  , ItemDescription
  , HasDiscount
ORDER BY
  CASE HasDiscount
    WHEN 1 THEN 0
    ELSE 1
  END
  , CustomerName
  , ItemDescription;
GO
```

The sales that have discount are marked with 0

All the other rows are marked with 1 and are ordered after the discounted sales

Customer-Name	ItemDescrip-tion	HasDiscount	SalesValue	Case_Discount
Anabel Larson	Wooden Horse	1	3837.68	3262.028000
Xavier Jameson	Plastic Plain Blue	1	5.08	4.318000
Xavier Jameson	Rag Doll 33 cm.	1	4244.24	3607.604000
Xavier Jameson	Wooden Horse	1	515.02	437.767000
Anabel Larson	Plastic Plain Blue	0	14.67	14.670000
John Smith	Plastic Plain Blue	0	20.58	20.580000
John Smith	Rag Doll 33 cm.	0	398.56	398.560000

HasDiscount = 1

HasDiscount = 0

Sessions

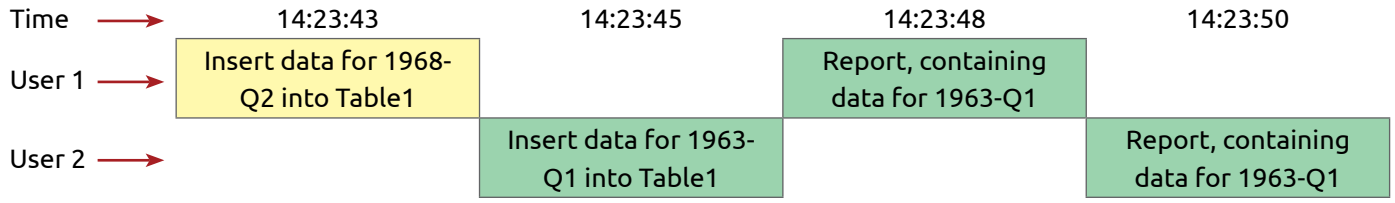
In a real life scenario we may have a parametrized stored procedure (SP) that creates the data for a report.
The steps in the SP are:

1. Extract the data for the chosen fiscal period (parameter)
2. Insert the data into a new table
3. Manipulate the new table
4. Extract the final data for the report from the new table
5. Delete the new table

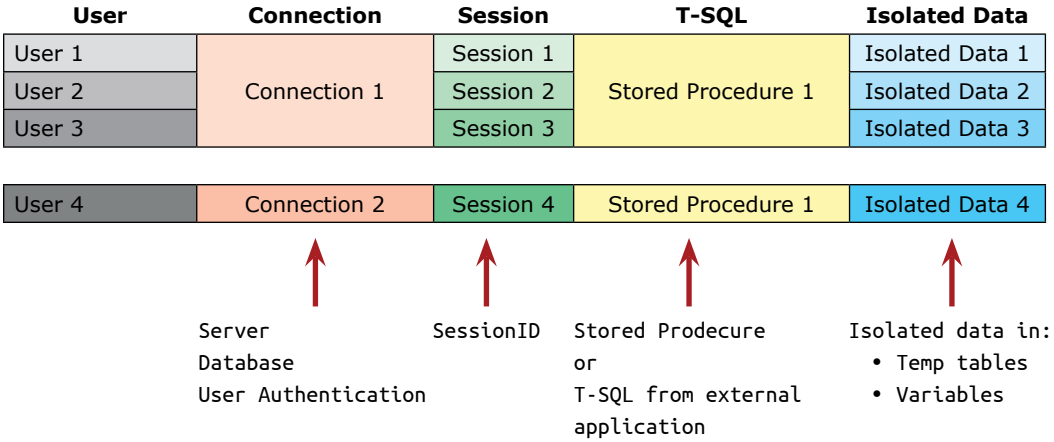
If multiple users execute the same SP simultaneously, there is a risk that they manipulate each others data.

The simultaneous execution steps may be:

1. User 1 runs a report for 1968-Q2, the SP truncates and loads the data for 1968-Q2 into `Table1`
2. User 2 runs a report for 1963-Q1, the SP truncates and loads the data for 1963-Q1 into `Table1` (the same table that serves User 1)
3. User 2 overwrites the data of User 1 in `Table1`
4. User 1 reads data, not related to the fiscal period he picked



To **isolate** the data for each user, SQL Server associates a **session** for each user's execution.



The dataflow is:

- The User is using specified Connection and sends a command - T-SQL code
- The DBE associates a unique Session for each command

- All the commands that run the same Stored Procedure, create their own Isolated Data (real or temp table, variable, etc.)
- Unique resultset is returned to each User

In the table above:

- User - the application that is connecting to the database and executes T-SQL
- Connection - the credentials that the User sends to the DBE:
  - Server name
  - DB name
  - User name
  - Password
- Session – the unique session assigned to each command (Combination of user and T-SQL code)
- Stored Procedure – the T-SQL that may cause an overlapping of the data between the users
- Isolated Data – the unique data for the unique session

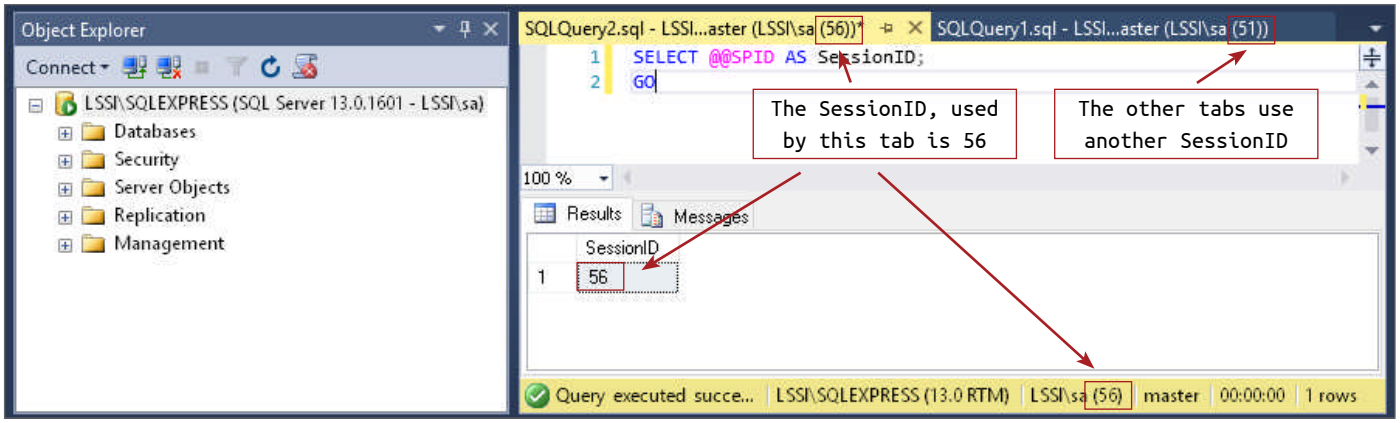
The built-in function `@@SPID` returns the **SessionID** of the **current session**:

```
SELECT @@SPID As SessionID;
GO
```

SessionID
53

Session in SSMS

We can test different statements in different sessions by executing them in different tabs in SSMS:



Tables

The table is a DB object where the data is stored.

In SQL Server the tables are:

- Permanent
- Temporary (temp)
  - Local
  - Global
  - Variable

Permanent Table

The permanent table is stored on the hard drive(s) of the server and exists permanently until it is dropped with the **DDL** `DROP TABLE` **statement**.

Query the **Object Catalog** and list the permanent tables in a **database**:

```sql
USE LearnSQLServerIntuitively;
GO

SELECT
  SCHEMA_NAME([schema_id]) AS SchemaName
  , name AS TableName
  , create_date AS DateCreated
FROM sys.tables
ORDER BY
  SchemaName
  , TableName;
GO
```

SchemaName	TableName	DateCreated
dbo	Table1	1965-07-15 17:35:02.160
dbo	Table2	1966-11-01 02:37:12.747
etl	Table1	1967-08-18 08:12:45.560
etl	Table2	1967-04-05 12:54:37.783

```sql
SELECT DISTINCT *
FROM LearnSQLServerIntuitively.sys.objects
WHERE [type] = 'U'
GO
```

Temporary (temp) Table

The temp table exists **temporarily**.
It is created by T-SQL and **isolated** to:

- the **batch** (the **session**) that executes the T-SQL
- **all** the **sessions**

The **temporary existence** and the **isolation** are the reasons why we use temp tables.

The different **types** of temp tables are:

- Local
- Global
- Variable

	Local temp table	**Global** temp table	Table **Variable**
The **identifier** (table name) **starts with**	# `#TempTable1`	## `##TempTable1`	@ `@TempTable1`
Exists until	the **session** that **creates** the table is **open**	the **session** that **creates** the table is **open**	the **batch** that `DECLARE`**s** the table variable is executed
Stores the **data** in	the system database **tempdb**	the system database **tempdb**	the **memory** of the server
Accessible by	**Only** the **session** that **creates** the table	**All** the **sessions**	Only the **batch** that creates the table variable
Manipulated as a permanent table with	**DDL** and **DML** statements	**DDL** and **DML** statements	**DML** statements

Create and use temp tables

Before we create **local** or **global** temp tables, we **clean up**:

```
DROP TABLE IF EXISTS #LocalTempTable;
GO
```

Create temp table with `CREATE TABLE` or `SELECT... INTO` statement:

```
CREATE TABLE #LocalTempTable1
  (
    CustomerID INT IDENTITY(1, 1) PRIMARY KEY
    , FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
    , Email VARCHAR(128)
  );
GO
```

After the **local** or **global** temp table is created, we can manipulate it with the **DDL** and **DML statements** that we learned so far.

Once we **no longer need** the temp table, we run the clean up code to **release the disk space** that was used by the temp table.

As the **table variable** is created in the **memory** and is **alive** at the time of the **execution of the batch**, it is **not necessary** to handle the **clean up manually**.

We can't use the `SELECT... INTO` statement to load data into table variable.

```sql
DECLARE @TableVariable1 TABLE
  (
    CustomerID INT IDENTITY(1, 1) PRIMARY KEY
    , FirstName NVARCHAR(32)
    , LastName NVARCHAR(64)
  );

INSERT @TableVariable1 (FirstName, LastName)
SELECT 'Anna', 'Laurier'
UNION ALL SELECT 'Anabel', 'Larson';

SELECT *
FROM @TableVariable1;
GO
```

Create the table variable

Load data

Select the data from the table variable

After the execution of the batch, the variable doesn't exist

CustomerID	FirstName	LastName
1	Anna	Laurier
2	Anabel	Larson

We **can't** manipulate the **table variable** with **DDL statements**.

Query the **Object Catalog** and list the tables in **tempdb**:

```sql
SELECT
  SCHEMA_NAME([schema_id]) AS SchemaName
  , name AS TableName
  , create_date AS DateCreated
FROM tempdb.sys.tables
ORDER BY
  SchemaName
  , TableName;
GO
```

Schema-Name	TableName	DateCreated
dbo	##GlobalTempTable1	1966-04-18 11:13:41.870
dbo	#LocalTempTable1__...____000000000002	1966-04-18 12:05:05.097

Isolation of the temp table

The local temp table is **local**, because it is accessible only by the session that creates it.

Local temp table
The **local** temp table, created in the **first tab** in SSMS:

```sql
DROP TABLE IF EXISTS #LocalTempTable;

CREATE TABLE #LocalTempTable (TableType VARCHAR(6));
```

Step 1: Execute in the **first tab** in SSMS

```sql
INSERT #LocalTempTable (TableType)
SELECT 'Local';

SELECT *
FROM #LocalTempTable;
GO
```

TableType
Local

is **not accessible** in the **second tab** in SSMS (**another session**):

```sql
SELECT *
FROM #LocalTempTable;
GO
```

Step 2: Execute in the **second tab** in SSMS

Invalid object name '#LocalTempTable'.

Global temp table

The **global** temp table, created in the **first tab** in SSMS:

```sql
DROP TABLE IF EXISTS ##GlobalTempTable;

CREATE TABLE ##GlobalTempTable (TableType VARCHAR(6));

INSERT ##GlobalTempTable (TableType)
SELECT 'Global';

SELECT *
FROM ##GlobalTempTable;
GO
```

Step 1: Execute in the **first tab** in SSMS

TableType
Global

is **accessible** by **all** the **sessions**:

```sql
SELECT *
FROM ##GlobalTempTable;
GO
```

Step 2: Execute in any **tab** other than the **first** in SSMS

TableType
Global

until the **session that creates** the table is **open**:

Step 3: **Close** the **first tab** (close the session)

```sql
SELECT *
FROM ##GlobalTempTable;
GO
```

Step 2: Execute in any **tab** other than the **first** in SSMS

Invalid object name '##GlobalTempTable'.

Table variable

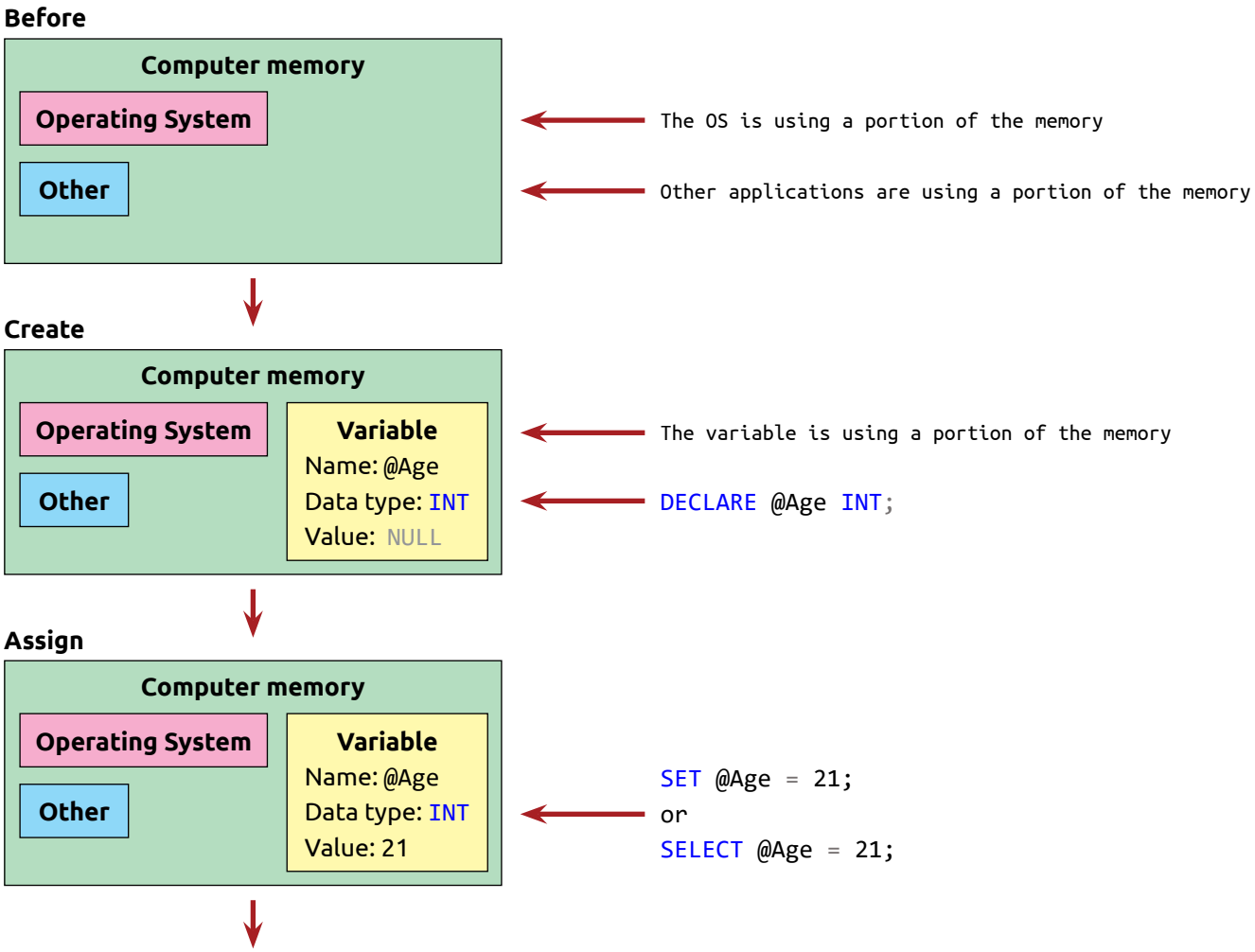
```
DECLARE @TableVariable1…
INSERT @TableVariable1…
GO                                  GO ends the batch and the table variable doesn't exist anymore

SELECT *
FROM @TableVariable1;               The table variable doesn't exist anymore
GO
                                    Must declare the table variable "@TableVariable1".
```

The variable:

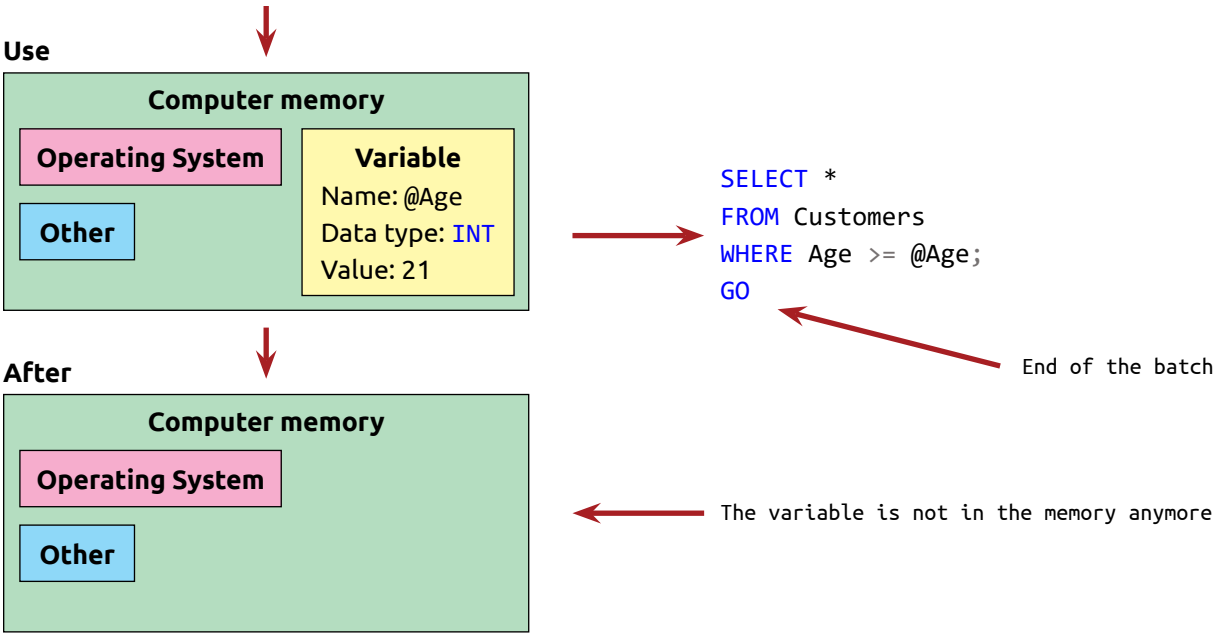
- Stored **in the memory** of the server is:
  - **Single value** of specific data type
  - **Multiple values** of multiple data types (table variable)
- **Exists** at the time of the **execution of the batch** where it is created
- Is accessible only inside the batch where it is created
- Has unique name, starting with the **at** symbol (**@**)

Variable life cycle:

1. **Create** - declare the variable and reserve space in the memory of the server
2. **Assign value** - attach a value(s) to the variable
3. **Use** - read the value of the variable from the computers memory and implement it in the code
4. **Clean up** - after the execution of the batch, the variable doesn't exist anymore



Variables



Create variable (DECLARE)

```
DECLARE @VariableNumeric INT;
```

DECLARE one variable.

```
DECLARE
  @VariableNumeric INT
  , @VariableString VARCHAR(32);
```

DECLARE more than one variable, delimited with a comma (,).

After the variable is created its value is NULL.

Variable name

Starts with the **at** symbol (**@**).
@FirstName, @firstName, @_firstName, @Par_DateInvoiced are valid variable names.

The following characters **can not** be used in the variable name:

Character	Variable Name
(space)	@Variable Name
!	@Variable!Name
"	@Variable"Name
%	@Variable%Name
&	@Variable&Name

`	@Variable'Name
(	@Variable(Name
)	@Variable)Name
*	@Variable*Name
+	@Variable+Name
,	@Variable,Name

-	@Variable-Name
.	@Variable.Name
/	@Variable/Name
:	@Variable:Name
;	@Variable;Name
<	@Variable<Name

=	@Variable=Name
>	@Variable>Name
?	@Variable?Name
[	@Variable[Name
\	@Variable\Name
]	@Variable]Name
^	@Variable^Name
`	@Variable`Name
{	@Variable{Name
|	@Variable|Name
}	@Variable}Name
~	@Variable~Name
€	@Variable€Name
‚	@Variable‚Name
„	@Variable„Name
…	@Variable…Name
†	@Variable†Name
‡	@Variable‡Name
ˆ	@Variable ˆName
‰	@Variable‰Name

‹	@Variable‹Name
‘	@Variable‘Name
’	@Variable’Name
“	@Variable“Name
”	@Variable”Name
•	@Variable•Name
–	@Variable–Name
—	@Variable—Name
˜	@Variable˜Name
™	@Variable™Name
›	@Variable›Name
¡	@Variable¡Name
¢	@Variable¢Name
£	@Variable£Name
¤	@Variable¤Name
¥	@Variable¥Name
¦	@Variable¦Name
§	@Variable§Name
¨	@Variable¨Name
©	@Variable©Name

«	@Variable«Name
¬	@Variable¬Name
®	@Variable®Name
¯	@Variable¯Name
°	@Variable°Name
±	@Variable±Name
²	@Variable²Name
³	@Variable³Name
´	@Variable´Name
¶	@Variable¶Name
·	@Variable·Name
¸	@Variable¸Name
¹	@Variable¹Name
»	@Variable»Name
¼	@Variable¼Name
½	@Variable½Name
¾	@Variable¾Name
¿	@Variable¿Name
×	@Variable×Name
÷	@Variable÷Name

We are not allowed to wrap the variable name in square brackets ([ ]) in order to use the characters above.

We can extend the naming conventions rules with:

- Include the data type as a prefix in the name

```
DECLARE @strVariable NVARCHAR(32)
DECLARE @intVariable INT;
```

Variable Data type

The data type that a variable can hold which is any SQL Server data type, except TEXT, NTEXT and IMAGE (these data types are obsolete and not suggested to be used).

TEXT, NTEXT and IMAGE data types are obsolete and suggested not to be used.

Assign value to variable (SET, SELECT)

Data type conversion

- When the variable and the assigned value are of **the same data** type or **can be implicitly converted**, we assign with an **assignment operator**:

- =
- +=
- -=
- *=
- /=
- %=

- Else, we **manually convert** the assigned value to the data type of the variable

Assign value

```
DECLARE @VariableNumeric INT;

SET @VariableNumeric = 123;                          No conversion
SET @VariableNumeric = '123';                        Implicit conversion
SET @VariableNumeric = CAST(GETDATE() AS INT);       Manual conversion
SET @VariableNumeric = (120 + 3);                    Expression. No conversion

SELECT @VariableNumeric = 123;
```

The SELECT statement can assign values to more than one variable:

```
SELECT
  @VariableNumeric = 456
  , @VariableString = 'DEF';
```

Assign value from recordset

Variables

ColumnNumeric	ColumnString
1	A
2	B
3	C
4	D
5	E
6	F
7	G

As the variable accepts a single value, when we use the result of a recordset, it must contain a single value (1 row and 1 column).

```
DECLARE @VariableNumeric INT;
SELECT @VariableNumeric =
  (
  SELECT
    1 AS ColumnNumeric
    , 'A' AS ColumnString
  );
GO
```

More than one column

Only one expression can be specified in the select list when the subquery is not introduced with EXISTS.

```
SET @VariableNumeric =
  (
    SELECT MAX(ColumnNumeric)
    FROM LearnSQLServerIntuitively.dbo.Variables
  );
GO
```

Assign value 7 to @VariableNumeric

One row, one column

When we assign a new value, it overwrites the old one:

```
SELECT @VariableNumeric = 123;          ← The value is 123
SELECT @VariableNumeric = 456;          ← The new value 456 overwrites the old value 123
SELECT @VariableNumeric += 1;           ← The new value is 456 + 1 = 457
```

When we assign a value from the recordset, we'll assign it as many times as the number of the rows in the recordset.
The last assigned value is the last queried value of the recordset.

Assign:

- Constant value 123 to `@VariableNumeric`
- The value of **`ColumnString`** from **`Variables`** to `@VariableString`:

```
SELECT
  @VariableNumeric = 123                          ← Constant value
  , @VariableString = ColumnString                ← Value, obtained from a recordset (Data-
FROM LearnSQLServerIntuitively.dbo.Variables;       baseName.SchemaName.TableName)
GO
```

Value 123 will be assigned 7 times to `@VariableNumeric`, because there are 7 rows in table **`Variables`**.
On every row, returned by the `SELECT` statement, the value of the column **`ColumnString`** is assigned to `@VariableString`.
The last value of column **`ColumnString`** `'G'` is assigned.

To avoid any additional load of the server, we need to make sure that the recordset returns one row:

```
SELECT
  @VariableNumeric = MIN(ColumnNumeric)
  , @VariableString = MAX(ColumnString)
FROM LearnSQLServerIntuitively.dbo.Variables;
GO
```

Now the recordset contains 1 (the **minimum** value in column **`ColumnNumeric`**) and `'G'` (the **maximum** value in column **`ColumnString`**).

Assigning a value from the multi-row recordset with the `SET` statement returns an error:

```
DECLARE @VariableNumeric INT;
SET @VariableNumeric =
  (
```

```
    SELECT ColumnNumeric
    FROM LearnSQLServerIntuitively.dbo.Variables
  );
GO
```

Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.

We can create a variable and assign a value to it in one statement:

```
DECLARE @VariableNumeric INT = 123;
```

Use the variable

```
DECLARE
  @VariableNumeric INT = 11
  , @VariableString VARCHAR(2) = '22';
```

Create and assign a value

11 + 22

Result
33

```
SELECT (@VariableNumeric + @VariableString) AS Result;
GO
```

Use

After the batch is executed, the variable doesn't exist anymore

or

Create variable(s)

```
DECLARE
  @CommissionPercentage DECIMAL(5, 3)
  , @Today DATE = CONVERT(CHAR(10), GETDATE(), 120);
```

Remove the time from the DATETIME data type, returned by the built-in function GETDATE()

```
SELECT @CommissionPercentage = CommisionPercentage
FROM LearnSQLServerIntuitively.dbo.CommissionPercentage
WHERE DateFiscal = @Today;
```

Assign a value to the variable(s)

```
SELECT
  CONCAT(SP.FirstName, SP.LastName) AS CustomerName
  , FORMAT(@Today, 'MMM dd, yyy') AS CommissionDate
  , SUM(S.Quantity * S.Price) AS LineTotal
  , @CommissionPercentage AS CommissionPercentage
  , (SUM(S.Quantity * S.Price) * @CommissionPercentage) AS Commission
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.SalesPersons AS SP
    ON S.SalesPersonID = SP.SalesPersonID
WHERE S.SaleDate = @Today
GROUP BY
  SP.FirstName
  , SP.LastName;
GO
```

Use the variables

Variable in the SELECT clause

Variable in the WHERE clause

Table variable

The table variable:

- Stores multiple values in rows and columns
- Is queried with SELECT DML statements like any other table

In the example below, we (INNER) JOIN table variable to filter the recordset.

TableName

CustomerID	IsCommisionable	FiscalYear	FiscalMonth	SalesValue
1	1	1967	9	927.288
1	1	1967	10	476.5055
1	1	1967	11	965.9223
1	1	1967	12	92.0612
2	1	1967	5	820.0465
2	1	1968	1	906.2333
2	0	1968	1	518.9526
2	1	1968	2	408.4823
2	1	1968	2	295.8332
3	1	1967	4	946.367
3	1	1967	11	296.7948
3	1	1967	11	762.0847
3	0	1968	1	486.4593

```
DECLARE @IsCommisionable BIT = 1;

DECLARE @TableVariable TABLE
  (
    FiscalYear SMALLINT
    , FiscalMonth TINYINT
  );

INSERT @TableVariable
  (
    FiscalYear
    , FiscalMonth
  )
SELECT 1967, 11
UNION ALL SELECT 1967, 12
```

Create variable

Create Table Variable

Assign values (INSERT)

FiscalYear	FiscalMonth
1967	11
1967	12
1968	1

```sql
UNION ALL SELECT 1968, 1;
SELECT
  CustomerID
  , CAST(S.FiscalYear AS CHAR(4))
    + '-' + RIGHT('0' + CAST(S.FiscalMonth AS VARCHAR(2)), 2) AS Period
  , SUM(S.SalesValue) AS Sum_SalesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN @TableVariable AS TV
    ON S.FiscalYear = TV.FiscalYear
    AND S.FiscalMonth = TV.FiscalMonth
    AND S.IsCommisionable = @IsCommisionable
GROUP BY
  CustomerID
  , CAST(S.FiscalYear AS CHAR(4))
    + '-' + RIGHT('0' + CAST(S.FiscalMonth AS VARCHAR(2)), 2)
ORDER BY
  CustomerID
  , Period;
GO
```

Add leading zero

Use the Table Variable in the FROM clause

Variable in the ON clause

CustomerID	Period	Sum_SalesValue
1	1967-11	965.9223
1	1967-12	92.0612
2	1968-01	906.2333
3	1967-11	1058.8795

Scope of the variable

The variable exists in the batch, in which it is created.

```sql
DECLARE @VariableNumeric INT = 1;
SELECT @VariableNumeric;
GO

SELECT @VariableNumeric;
GO
```

End of the batch
@VariableNumeric is not alive anymore

Next batch

Must declare the scalar variable “@VariableNumeric”.

@VariableNumeric is created in the first batch and can’t be used in the next batch.

There are system functions, with names starting with 2 **at** symbols (@@). They are not variables.

```sql
SELECT
  @@VERSION AS [Database Version]
  , @@SPID AS [Session ID]
  , @@SERVERNAME AS [Server Name]
  , @@LANGUAGE AS [Language];
GO
```

WHILE Loops

The WHILE statement is a loop i.e. a block of statement(s) in which the execution is repeated as long as the verification of condition returns True.

The syntax is:

```
1. WHILE (Condition)
2. BEGIN
3.   ...Statement1...;
4.   ...Statement2...;
5. END
6.
7. Statement3...;
```

The execution steps are:

1. The Condition is verified (line 1)
2. If the Condition is met (True), the execution continues inside the BEGIN... END block (lines 3, 4), else the BEGIN... END block is skipped (continue to line 7)
3. The statement(s) inside the BEGIN... END block (lines 3, 4)) is/are executed
4. The Condition is verified again (line 1)
5. Repeat (loop) the statement(s) inside the BEGIN... END block (lines 3, 4) until the Condition (line 1) is met (True) and exit the BEGIN... END block (continue to line 7)
6. If the Condition (line 1) is never met (False) after the execution is inside the BEGIN... END block (lines 3, 4)), the loop repeats infinitely (Infinite Loop)

Example: The condition is not met on the first verification

```
DECLARE @LoopedValue INT = 1;

WHILE (@LoopedValue >= 5)
BEGIN
  PRINT @LoopedValue;
  SELECT @LoopedValue += 1;
END
GO
```

The condition is not met (the value of @LoopedValue is 1) and the BEGIN... END block is skipped

Example: Count to 5

To loop the numbers from 1 to 5, we need one variable that stores and prints the current value and another variable that counts the loops:

```
DECLARE
  @Counter TINYINT = 1
  , @LoopedValue INT = 1;

WHILE (@Counter <= 5)
BEGIN
  PRINT @LoopedValue;
  SELECT
    @LoopedValue += 1
    , @Counter += 1;
END
GO
```

Variables

Counts the loops

Stores and prints in the looped value

Condition

Print the current value of @LoopedValue

Increase the value of @LoopedValue by 1

Increase the value of @Counter by 1

When the condition is verified for the first time, the value of both variables is 1 and the verification of the condition returns True (1 is less or equal to 5).
Inside the BEGIN... END block the value of @LoopedValue (1) is printed and the values of both variables are increased by 1. Now the value of the both variables is 2.
The condition is verified again and returns True (2 is less or equal to 5)
The value of @LoopedValue (2) is printed and the values of both variables are increased by 1. Now the value of the both variables is 3.
...
The condition is verified again and returns False (6 is not less or equal to 5)
The execution of the WHILE loop is terminated.

Example: Print the capital characters of the alphabet

To print the alphabet, we use the built-in function CHAR() that converts an integer value to its corresponding ASCII code.

The same variable @LoopedValue is used for data manipulation and counting purposes:

```
DECLARE @LoopedValue INT = 65;        ←——— SELECT CHAR(65); ——> Capital A

WHILE (@LoopedValue <= 90)            ←——— SELECT CHAR(90); ——> Capital Z
BEGIN
  PRINT CHAR(@LoopedValue);

  SET @LoopedValue += 1;
END
GO
```

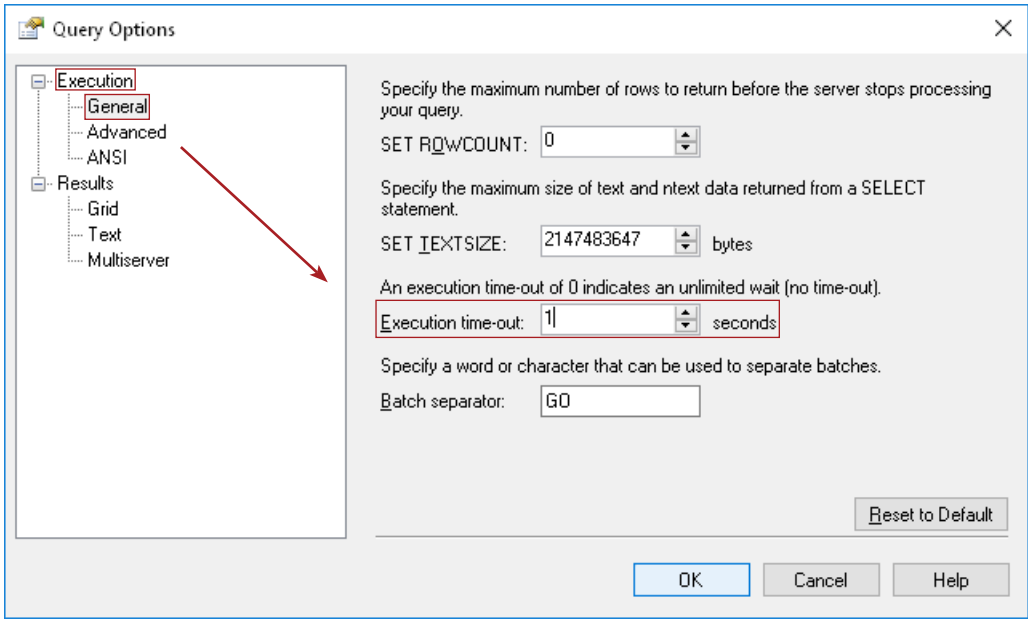
Example: The condition can not be met after the execution is inside the BEGIN... END block (Infinite Loop)

```
DECLARE @Counter INT = 1

WHILE (@Counter <= 50)
BEGIN
  PRINT @Counter;

  SELECT @Counter -= 1;
END
GO
```

Starting from 1 and decreasing @Counter by 1 (1, 0, -1, -2...)), the condition is always True. (the value of @Counter is always less than 50)

The condition will never be met and the WHILE loop will be repeated until the execution time-out is reached.
To change the time-out in SSMS only for the current tab, right-click in the Query Window ——> Query Options… ——> Execution ——> General ——> Execution time-out:

As this statement is an infinite loop, cancel it as soon as possible.



When we change the query time-out and execute again with the same `WHILE` statement, after 1 second an error message is returned:

```
Timeout expired.  The timeout period elapsed prior to completion of the operation or the server is not
responding.
```

BREAK... CONTINUE or CONTINUE... BREAK

`CONTINUE` and `BREAK` control the execution inside the `BEGIN`... `END` block. They are used together with another condition.

To stop the looping when a certain condition is met, we can use the `IF` statement inside the `BEGIN`... `END` block:

```
DROP TABLE IF EXISTS #WHILEResult1;
GO

CREATE TABLE #WHILEResult1 (Result VARCHAR(16));

DECLARE
  @MaxID INT
  , @ID INT = 1;

SELECT @MaxID = MAX(ID)
FROM LearnSQLServerIntuitively.dbo.WHILE1;
```

Select the maximum value of column ID (4) into variable

WHILE1

ID	Column1	Column2
1	100	A
2	101	B
3	102	C
4	103	D

```sql
WHILE (@ID <= @MaxID)
BEGIN
  IF (@ID <= 2)
    BEGIN
      INSERT #WHILEResult1 (Result)
      SELECT CONCAT(CAST(Column1 AS VARCHAR), ' - ', Column2)
      FROM LearnSQLServerIntuitively.dbo.WHILE1
      WHERE ID = @ID;

      SELECT @ID += 1;

      CONTINUE;
    END
  ELSE
    BEGIN BREAK; END
END

SELECT *
FROM #WHILEResult1;
GO

DROP TABLE IF EXISTS #WHILEResult1;
GO
```

If @ID is less or equal to 2, insert a row in #WHILEResult1 and CONTINUE the execution (Verify the WHILE condition again)

Transfer the data for the row where the ID matches the current @ID into the temp table

If @ID is not less or equal to 2, BREAK the execution

SELECT the data from the temp table

#WHILEResult1

Result
100 - A
101 - B

Clean up the temp table

Common Table Expressions (CTE)

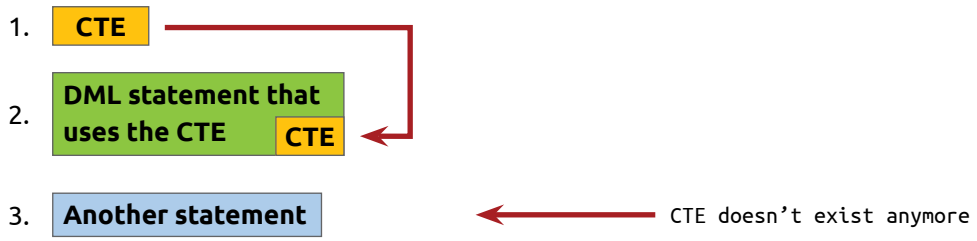
Common Table Expression (CTE) is a virtual recordset (VR), used in the subsequent DML statement.

The syntax is:

```
WITH CTE_Name (Col1, Col2) AS
  (
    SELECT                                  <------ CTE definition
      Column1
      , Column2
    FROM DatabaseName.SchemaName.TableName
  )
SELECT                                      <------ Subsequent DML statement
  Col1
  , Col2
FROM CTE_Name;
GO
```

The life cycle of CTE is:

1. The `WITH` clause creates the VR that is used in the next DML statement
2. The next DML statement is using the VR
3. After the execution of the DML statement, the CTE doesn't exist anymore



Data sources for the examples below:

Customers

CustomerID	FirstName	LastName	Email	DateTimeRegistered	IsActive
1	Anabel	Larson	anabel.larson@customer5.info	1968-07-02 03:22:20.133	1
2	Anna	Laurier	anna.laurier@customer2.net	1969-12-14 19:24:49.527	NULL
3	Beverly	NULL	NULL	1969-06-02 22:09:48.943	NULL
4	John	Smith	john.smith@customer1.com	1968-04-12 06:16:25.337	1
5	John	Smith	john.smith@customer3.org	1963-04-11 01:55:38.420	NULL
6	Melanie	Larson	melanie.larson@customer4.biz	1965-11-18 16:34:48.753	1
7	Xavier	Jameson	xavier.garcía@customer6.com	1961-12-11 01:49:52.020	1
8	Zak	Smith	NULL	1967-12-03 06:15:50.117	NULL

Sales

SaleID	CustomerID	ItemID	DateTimeOfSale	Quantity	Price
1	3	2043	1969-09-05 12:47:05.467	4	1.27
2	4	5164	1969-12-18 02:15:36.470	2	1.56
3	3	5293	1969-11-10 20:18:49.610	77	55.12
4	5	6223	1964-01-10 21:13:18.150	3	4.89
5	7	1352	1961-12-12 02:28:34.363	89	0.84
6	2	1953	1969-12-22 07:39:41.243	53	2.84
7	3	8613	1969-12-24 18:34:13.777	22	23.41
8	2	5523	1969-12-27 16:43:01.750	55	127.19
9	1	8534	1969-12-25 08:33:58.513	1	45.88
10	2	9375	1969-12-15 09:39:11.403	6	0.43

CTE in the SELECT clause

As the CTE is a recordset, when we use it in the SELECT clause, we put it in a subquery, that has to return a single value (one row and one column):

```
WITH CTE_SELECT (CID, IA) AS
  (
    SELECT
      CustomerID
      , 'Yes'
    FROM LearnSQLServerIntuitively.dbo.Customers
    WHERE IsActive = 1
  )
SELECT
  S.CustomerID
  , C.FirstName
  , C.LastName
  , (SELECT IA FROM CTE_SELECT WHERE CID = S.CustomerID) AS IsActive
  , MAX(S.DateTimeOfSale) AS Max_DateOfSale
  , SUM(S.Quantity * S.Price) AS SalesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.Customers AS C
    ON S.CustomerID = C.CustomerID
GROUP BY
  S.CustomerID
  , C.FirstName
  , C.LastName;
GO
```

CTE name and aliases (CID, IA), created explicitly and in the same order as the columns in the CTE definition

CTE definition

CTE_SELECT

CID	IA
1	Yes
4	Yes
6	Yes
7	Yes

Subquery to add the status of the Customers

CustomerID	FirstName	LastName	IsActive	Max_DateOfSale	SalesValue
1	Anabel	Larson	Yes	1969-12-25 08:33:58.513	45.88
2	Anna	Laurier	NULL	1969-12-27 16:43:01.750	7148.55
3	Beverly	NULL	NULL	1969-12-24 18:34:13.777	4764.34
4	John	Smith	Yes	1969-12-18 02:15:36.470	3.12
5	John	Smith	NULL	1964-01-10 21:13:18.150	14.67
7	Xavier	Jameson	Yes	1961-12-12 02:28:34.363	74.76

CTE in the FROM clause

We can use a CTE in the FROM clause the same way we use any other recordset (data source).

The CTE creates a VR with data for the last 3 registered customers. The next statement is using the CTE to SELECT the sales for these customers:

CTE name. Aliases are not specified here and the column names as they are defined in the CTE definition are used

```
WITH CTE_FROM AS
  (
    SELECT TOP 3 CustomerID, FirstName, LastName, DateTimeRegistered AS DTR
    FROM LearnSQLServerIntuitively.dbo.Customers
    ORDER BY DateTimeRegistered DESC
  )
SELECT
  C.CustomerID
  , C.FirstName
  , C.LastName
  , C.DTR AS DateTimeRegistered
  , MAX(S.DateTimeOfSale) AS Max_DateTimeOfSale
  , SUM(S.Quantity * S.Price) AS SalesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN CTE_FROM AS C
    ON S.CustomerID = C.CustomerID
GROUP BY
  C.CustomerID
  , C.FirstName
  , C.LastName
  , C.DTR
ORDER BY
  MAX(S.DateTimeOfSale) DESC
  , SUM(S.Quantity * S.Price) DESC;
GO
```

CustomerID	FirstName	LastName	DTR
2	Anna	Laurier	1969-12-14 19:24:49.527
3	Beverly	NULL	1969-06-02 22:09:48.943
1	Anabel	Larson	1968-07-02 03:22:20.133

If more than one customer bought in the same day, sort by **SalesValue** descending

CustomerID	FirstName	LastName	DateTimeRegistered	Max_DateTimeOfSale	SalesValue
2	Anna	Laurier	1969-12-14 19:24:49.527	1969-12-27 16:43:01.750	7148.55
1	Anabel	Larson	1968-07-02 03:22:20.133	1969-12-25 08:33:58.513	45.88
3	Beverly	NULL	1969-06-02 22:09:48.943	1969-12-24 18:34:13.777	4764.34

CTE in the WHERE clause

```sql
WITH CTE_WHERE AS
  (
    SELECT TOP 3 CustomerID
    FROM LearnSQLServerIntuitively.dbo.Customers
    ORDER BY DateTimeRegistered DESC
  )
SELECT
  C.CustomerID
  , C.FirstName
  , C.LastName
  , MAX(S.DateTimeOfSale) AS Max_DateTimeOfSale
  , SUM(S.Quantity * S.Price) AS SalesValue
FROM
  LearnSQLServerIntuitively.dbo.Sales AS S
  JOIN LearnSQLServerIntuitively.dbo.Customers AS C
    ON S.CustomerID = C.CustomerID
WHERE
  C.CustomerID IN
  (
    SELECT CustomerID
    FROM CTE_WHERE
  )
GROUP BY
  C.CustomerID
  , C.FirstName
  , C.LastName
ORDER BY
  MAX(S.DateTimeOfSale) DESC
  , SUM(S.Quantity * S.Price) DESC;
GO
```

CTE_WHERE

CustomerID
2
3
1

Custo-merID	First-Name	LastName	Max_DateTimeOfSale	Sales-Value
2	Anna	Laurier	1969-12-27 16:43:01.750	7148.55
1	Anabel	Larson	1969-12-25 08:33:58.513	45.88
3	Beverly	NULL	1969-12-24 18:34:13.777	4764.34

Multiple CTEs in one statement

We can split multiple CTEs with comma (,) and use them in the subsequent DML statement.

```sql
WITH CTE_CountItemID (CustomerID, Count_ItemID) AS
  (
    SELECT
      CustomerID AS FakeAlias
      , COUNT(DISTINCT ItemID)
    FROM LearnSQLServerIntuitively.dbo.Sales
    GROUP BY CustomerID
  )
  , CTE_Sum_SalesValue AS
  (
    SELECT
      CustomerID
      , SUM(Quantity * Price) AS Sum_SalesVallue
    FROM LearnSQLServerIntuitively.dbo.Sales
    GROUP BY CustomerID
    HAVING SUM(Quantity * Price) > 20
  )
SELECT
  CI.CustomerID
  , CONCAT(C.FirstName, ' ', C.LastName) AS CustomerName
  , CI.Count_ItemID
  , SSV.Sum_SalesVallue
  , CASE
    WHEN CI.Count_ItemID != 0 THEN (SSV.Sum_SalesVallue / CI.Count_ItemID)
  END AS Avg_SalesValuePerItemID
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN CTE_CountItemID AS CI
    ON C.CustomerID = CI.CustomerID
  JOIN CTE_Sum_SalesValue AS SSV
    ON CI.CustomerID = SSV.CustomerID;
GO
```

Fake Alias is not used, because a column list is provided

Count the sold items by customer. No alias. The column names are defined in the column list

Alias list is not provided next to the CTE name and the column names, defined in the CTE definition are used

Filter customers with sales greater than 20

CTE_CountItemID

CustomerID	Count_ItemID
1	1
2	3
3	3
4	1
5	1
7	1

CTE_Sum_SalesValue

CustomerID	Sum_SalesVallue
1	45.88
2	7148.55
3	4764.34
7	74.76

CustomerID	CustomerName	Count_ItemID	Sum_SalesVallue	Avg_SalesValuePerItemID
1	Anabel Larson	1	45.88	45.88
2	Anna Laurier	3	7148.55	2382.85
3	Beverly	3	4764.34	1588.1133
7	Xavier Jameson	1	74.76	74.76

Recursive CTE

Recursive CTE is one that calls itself. In the example below we build a parent-child relation with recursive CTE.

GroupsParentChild

ParentID	ChildID	ParentToChild	ChildToParent
NULL	3	Parent None - Child 3	Child 3 - Parent None
NULL	7	Parent None - Child 7	Child 7 - Parent None
1	8	Parent 1 - Child 8	Child 8 - Parent 1
2	4	Parent 2 - Child 4	Child 4 - Parent 2
3	1	Parent 3 - Child 1	Child 1 - Parent 3
3	2	Parent 3 - Child 2	Child 2 - Parent 3
6	9	Parent 6 - Child 9	Child 9 - Parent 6
7	5	Parent 7 - Child 5	Child 5 - Parent 7
7	6	Parent 7 - Child 6	Child 6 - Parent 7
8	10	Parent 8 - Child 10	Child 10 - Parent 8

Every child belongs to a parent. The child which parent is `NULL` is the top level. We can show the dependencies in 2 directions:

- Parent to child
- Child to parent

Parent to child (only `ChildID` = 3 is shown):

Level 0 (Anchor)	Level 1 (Recursive)	Level 2 (Recursive)	Level 3 (Recursive)
Parent None - Child 3	Parent 3 - Child 1	Parent 1 - Child 8	Parent 8 - Child 10
Parent None - Child 3	Parent 3 - Child 2	Parent 2 - Child 4	

As we don't know how many levels we have, we need to build the dependency list of the parent-child hierarchy dynamically:

```
WITH CTE_Recursive AS
  (
    SELECT
      'A' AS SourceStatement
    , 0 AS [Level]
    , ParentID
    , ChildID
    , ParentToChild
    FROM LearnSQLServerIntuitively.dbo.GroupsParentChild
    WHERE ParentID IS NULL
```

Anchor statement

NULL is the top level

SourceStatement	Level	ParentID	ChildID	ParentToChild
A	0	NULL	3	Parent None - Child 3
A	0	NULL	7	Parent None - Child 7

```
  UNION ALL                                   ← UNION ALL combines the anchor and the recursive statement(s)

  SELECT                                      ← Recursive statement
    'R' AS SourceStatement
    , (C.[Level] + 1) AS [Level]
    , GPC.ParentID
    , GPC.ChildID
    , GPC.ParentToChild
  FROM LearnSQLServerIntuitively.dbo.GroupsParentChild AS GPC
    JOIN CTE_Recursive AS C                   ← This CTE
      ON GPC.ParentID = C.ChildID
 )                                            ← JOIN parent to child
SELECT *
FROM CTE_Recursive
ORDER BY
  [Level]
  , ParentID;
GO
```

Recursions (only `ChildID` = 3 is shown):

Step 1: Execute the **anchor statement**

SourceStatement	Level	ParentID	ChildID	ParentToChild
A	0	NULL	3	Parent None - Child 3

Step 2: **Recursion** 1 (anchor statement `UNION ALL` Level 1)

SourceStatement	Level	ParentID	ChildID	ParentToChild
R	1	3	1	Parent 3 - Child 1
R	1	3	2	Parent 3 - Child 2

Step 3: **Recursion** 2 (`CTE_Recursive` `UNION ALL` Level 2)

SourceStatement	Level	ParentID	ChildID	ParentToChild
R	2	1	8	Parent 1 - Child 8
R	2	2	4	Parent 2 - Child 4

Step 4: **Recursion** 3 (`CTE_Recursive` `UNION ALL` Level 3)

SourceStatement	Level	ParentID	ChildID	ParentToChild
R	3	8	10	Parent 8 - Child 10

Values 10 and 4 don't exist in column **`ParentID`** and the recursions are terminated. The deepest level (3) is reached and the execution is finished.

The result of the execution of the above statement is:

SourceS-tatement	Level	ParentID	ChildID	ParentToChild
A	0	NULL	3	Parent None - Child 3
A	0	NULL	7	Parent None - Child 7
R	1	3	1	Parent 3 - Child 1
R	1	3	2	Parent 3 - Child 2
R	1	7	5	Parent 7 - Child 5
R	1	7	6	Parent 7 - Child 6
R	2	1	8	Parent 1 - Child 8
R	2	2	4	Parent 2 - Child 4
R	2	6	9	Parent 6 - Child 9
R	3	8	10	Parent 8 - Child 10

A - Anchor
R - Recursive

Child to parent

We can build the parent-child relation in the opposite direction – from child to parent:

```
DECLARE @ChildID INT = 10;

WITH CTE_Recursive AS
  (
    SELECT
      'A' AS SourceStatement
      , 0 AS [Level]
      , ChildToParent
      , ChildID
      , ParentID
    FROM LearnSQLServerIntuitively.dbo.GroupsParentChild
    WHERE ChildID = @ChildID                                  ← Start from the child

    UNION ALL

    SELECT
      'R' AS SourceStatement
      , (C.[Level] + 1) AS [Level]
      , GPC.ChildToParent
      , GPC.ChildID
      , GPC.ParentID
    FROM LearnSQLServerIntuitively.dbo.GroupsParentChild AS GPC
      JOIN CTE_Recursive AS C
        ON C.ParentID = GPC.ChildID                           ← Reverse the link logic
  )
SELECT *
FROM CTE_Recursive;
GO
```

The result is:

SourceStatement	Level	ChildToParent	ChildID	ParentID
A	0	Child 10 - Parent 8	10	8
R	1	Child 8 - Parent 1	8	1
R	2	Child 1 - Parent 3	1	3
R	3	Child 3 - Parent None	3	NULL

The recursion No. 3 reaches the top level (**ParentID** = NULL) and the execution terminates.

Changing the value of the variable @ChildID to 9, selects the relation for ChildID = 9 (9 ⟶ 6 ⟶ 7)

SourceStatement	Level	ChildToParent	ChildID	ParentID
A	0	Child 9 - Parent 6	9	6
R	1	Child 6 - Parent 7	6	7
R	2	Child 7 - Parent None	7	NULL

Limit the recursions

To avoid an infinite loop in recursive CTEs, we limit the number of the recursions by adding the `OPTION` clause (`MAXRECURSION` hint) at the end of the CTE.

```
OPTION (MAXRECURSION X)
```

```
X is between 1 and 32,767.
The default value of X (if OPTION (MAXRECURSION X) is not used) is 100.
```

As the deepest level in the last example is 2, this statement is executed successfully:

```
WITH CTE_Recursive...
SELECT *
FROM CTE_Recursive
OPTION (MAXRECURSION 2);
GO
```

If we change the value of `MAXRECURSION` to 1, an error message "`The statement terminated. The maximum recursion 1 has been exhausted before statement completion.`" is returned.

CTE Scope

The CTE can be used only by the DML statement that is after the WITH clause. If the statement that follows is using the CTE, an error message "`Invalid object name 'CTE_Name'.`" is returned.

The statement preceding the CTE (the `WITH` clause) must end with semicolon (`;`)

Views

The view is **T-SQL code**. It is **not** a DB objects that stores **data**

↳ The T-SQL is a **single SELECT** (SELECT, FROM, WHERE, GROUP BY) **statement**

↳ The **statement** SELECTs data from one or more data source(s) (underlying table(s)) and **creates virtual recordset**

↳ The **recordset** is **manipulated** like a **table**

The view is used to

- Simplify the usage of the data (hides from the user complex calculations, JOINed data sources, etc.)
- Transform the data to the business needs
- Manage the security by permitting the users to access only the views that show data that the users are authorized to use

The view can:

- SELECT specified columns

Customers

Custo-merID	First-Name	Last-Name	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org

vw_Customers

Before creating a view, we delete the view with the same name

```
DROP VIEW IF EXISTS dbo.vw_Customers
GO

CREATE VIEW dbo.vw_Customers
AS
  SELECT
    FirstName AS FN
    , LastName AS LN
  FROM LearnSQLServerIntuitively.dbo.Customers;
GO

SELECT *
FROM LearnSQLServerIntuitively.dbo.vw_Customers
ORDER BY
  FN
  , LN;
GO
```

CREATE VIEW (DDL statement)

View definition

Use the View (DML statement)

vw_Customers

FN	LN
Anabel	Larson
Anna	Laurier
Beverly	NULL
John	Smith
John	Smith

We SELECT the data that the **View** provides like any other data source (table, table variable, table-valued function etc.)

- SELECT specified rows

Customers

Custo-merID	First-Name	Last-Name	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org

vw_Customers

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE CustomerID <= 3;
```

View definition

vw_Customers

Custo-merID	First-Name	Last-Name	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL

- SELECT specified rows and columns

Customers

Custo-merID	First-Name	Last-Name	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org

vw_Customers

View definition

```
SELECT
       FirstName
       , LastName
FROM LearnSQLServerIntuitively.dbo.Customers
WHERE CustomerID <= 3;
```

vw_Customers

First-Name	Last-Name
Anabel	Larson
Anna	Laurier
Beverly	NULL

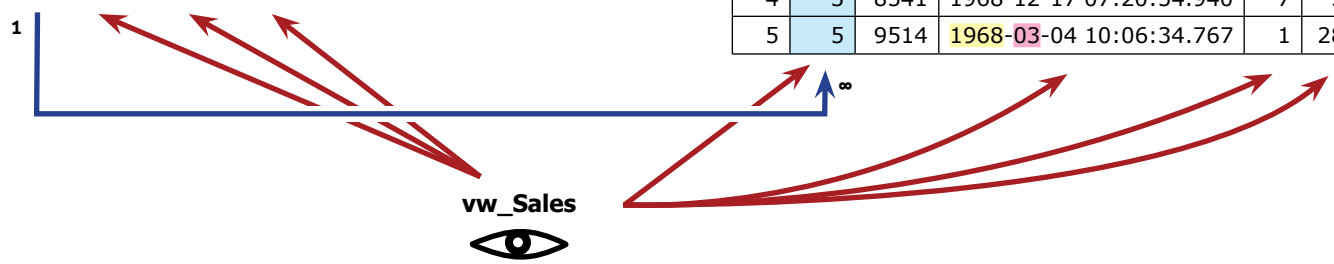
- `JOIN` DB objects

Customers

CustomerID	FirstName	LastName	Email
1	Anabel	Larson	anabel.larson@customer5.info
2	Anna	Laurier	anna.laurier@customer2.net
3	Beverly	NULL	NULL
4	John	Smith	john.smith@customer1.com
5	John	Smith	john.smith@customer3.org

Sales

SaleID	CustomerID	ItemID	DateOfSale	Quantity	Price
1	3	1004	1967-10-17 21:03:40.813	3	1.24
2	1	5273	1968-05-08 04:07:40.647	5	2.43
3	4	5432	1965-09-04 22:51:01.710	2	12.44
4	3	8541	1968-12-17 07:20:54.940	7	5.88
5	5	9514	1968-03-04 10:06:34.767	1	28.19




```sql
SELECT
  C.CustomerID AS CustomerID
  , C.FirstName
  , C.LastName
  , S.DateOfSale
  , S.Quantity
  , S.Price
  , (S.Quantity * S.Price) AS LineTotal
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
WHERE
  YEAR(S.DateOfSale) = 1968
  AND MONTH(S.DateOfSale) = 03;
```

View definition

vw_Sales

CustomerID	FirstName	LastName	DateOfSale	Quantity	Price	LineTotal
5	John	Smith	1968-03-04 10:06:34.767	1	28.19	28.19

- `GROUP BY` and aggregate the data

```sql
SELECT
  C.CustomerID
  , CONCAT(C.FirstName, ' ', C.LastName) AS CustomerName
  , SUM(S.Quantity * S.Price) AS LineTotal
```

View definition

```sql
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
GROUP BY
  C.CustomerID
  ,CONCAT(C.FirstName, ' ', C.LastName);
```

vw_Sales

CustomerID	CustomerName	LineTotal
1	Anabel Larson	12.15
3	Beverly	44.88
4	John Smith	24.88
5	John Smith	28.19

- Calculate the data

The column **`Discount`** in table **`Customers`** presents the discount in percent and it is used in the calculation of the **`LineTotal`**.

Customers

CustomerID	First-Name	Last-Name	Email	Discount
1	Anabel	Larson	anabel.larson@customer5.info	0.111
2	Anna	Laurier	anna.laurier@customer2.net	0.000
3	Beverly	NULL	NULL	0.217
4	John	Smith	john.smith@customer1.com	0.068
5	John	Smith	john.smith@customer3.org	0.000

Sales

SaleID	CustomerID	ItemID	DateOfSale	Quantity	Price
1	3	1004	1967-10-17 21:03:40.813	3	1.24
2	1	5273	1968-05-08 04:07:40.647	5	2.43
3	4	5432	1965-09-04 22:51:01.710	2	12.44
4	3	8541	1968-12-17 07:20:54.940	7	5.88
5	5	9514	1968-03-04 10:06:34.767	1	28.19

1 ∞

vw_SalesWithDiscounts

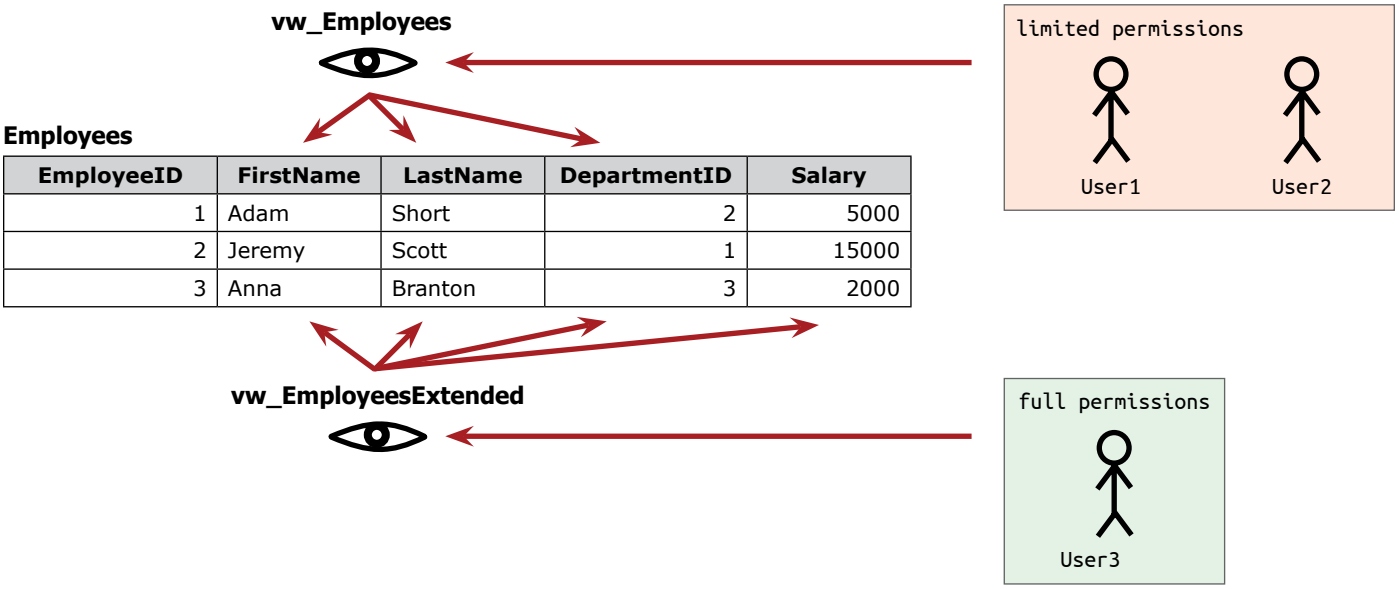
```sql
SELECT
  C.CustomerID AS CustomerID
  , CONCAT(C.FirstName, ' ', C.LastName) AS CustomerName
  , S.DateOfSale
  , S.Quantity
  , S.Price
  , C.Discount
  , (S.Quantity * S.Price) AS LineTotal
  , (S.Quantity * S.Price * (1 - C.Discount)) AS LineTotalDiscount
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID;
```

View definition

vw_SalesWithDiscounts

CustomerID	CustomerName	DateOfSale	Quantity	Price	Discount	LineTotal	LineTotalDiscount
3	Beverly	1967-10-17 21:03:40.813	3	1.24	0.217	3.72	2.9127600
1	Anabel Larson	1968-05-08 04:07:40.647	5	2.43	0.111	12.15	10.8013500
4	John Smith	1965-09-04 22:51:01.710	2	12.44	0.068	24.88	23.1881600
3	Beverly	1968-12-17 07:20:54.940	7	5.88	0.217	41.16	32.2282800
5	John Smith	1968-03-04 10:06:34.767	1	28.19	0.000	28.19	28.1900000

- Be accessible by specified user and not by other (data security)



An error message "`The server principal "User4" is not able to access the database "LearnSQLServerIntuitively" under the current security context.`" is returned to the users that are not permitted to manipulate the View.

Advantages (Pros):

- Facilitate the correlations between the business logic and the data
- Facilitate the DB development by "masking" complex code
- Apply security rules
- Saves space that should be used if we store the result of a view in a table
- We can edit the code of the view without the need to edit the code of the dependent DB object(s)

Disadvantages (Cons):

- When we use the view as a table, we add the complexity of the view to the code. This can lead to bad performance

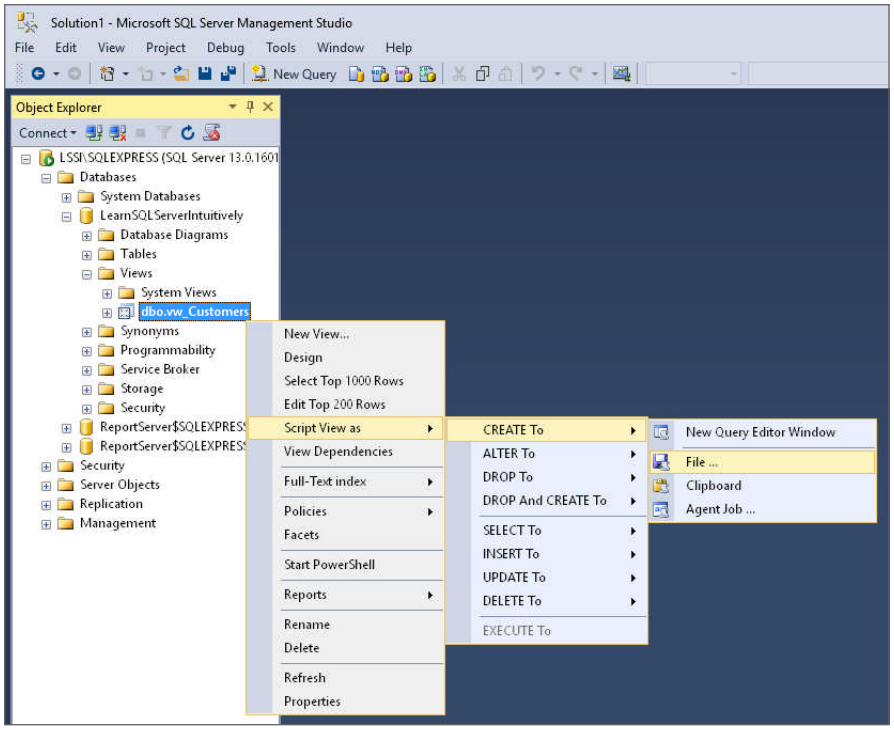
CREATE VIEW

The DDL statement `CREATE VIEW` is wrapping the definition of the View (the `SELECT` statement that builds the virtual recordset that the view `SELECT`s):

```
CREATE VIEW [dbo].[vw_Sales]
AS
  SELECT
    C.CustomerID AS CustomerID
    , C.FirstName
    , C.LastName
    , S.DateOfSale
    , S.Quantity
    , S.Price
    , (S.Quantity * S.Price) AS LineTotal
  FROM
    LearnSQLServerIntuitively.dbo.Customers AS C
    JOIN LearnSQLServerIntuitively.dbo.Sales AS S
      ON C.CustomerID = S.CustomerID;
GO
```

The **definition** of the View

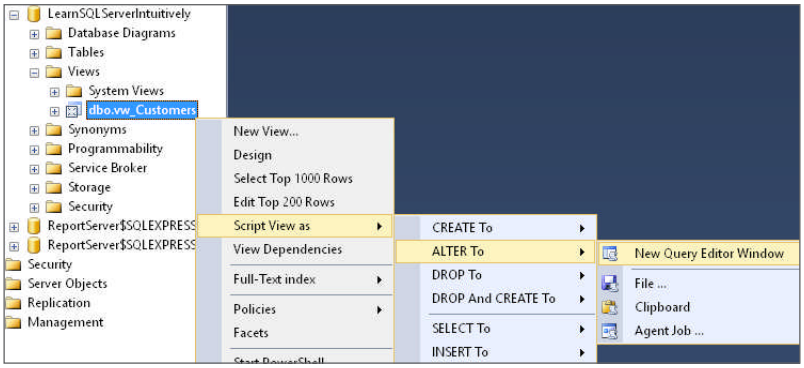
We can export the definition of an existing View in SSMS:



ALTER VIEW

To edit the code behind the view, we need to:

- Script the view in a New Query Editor Window:



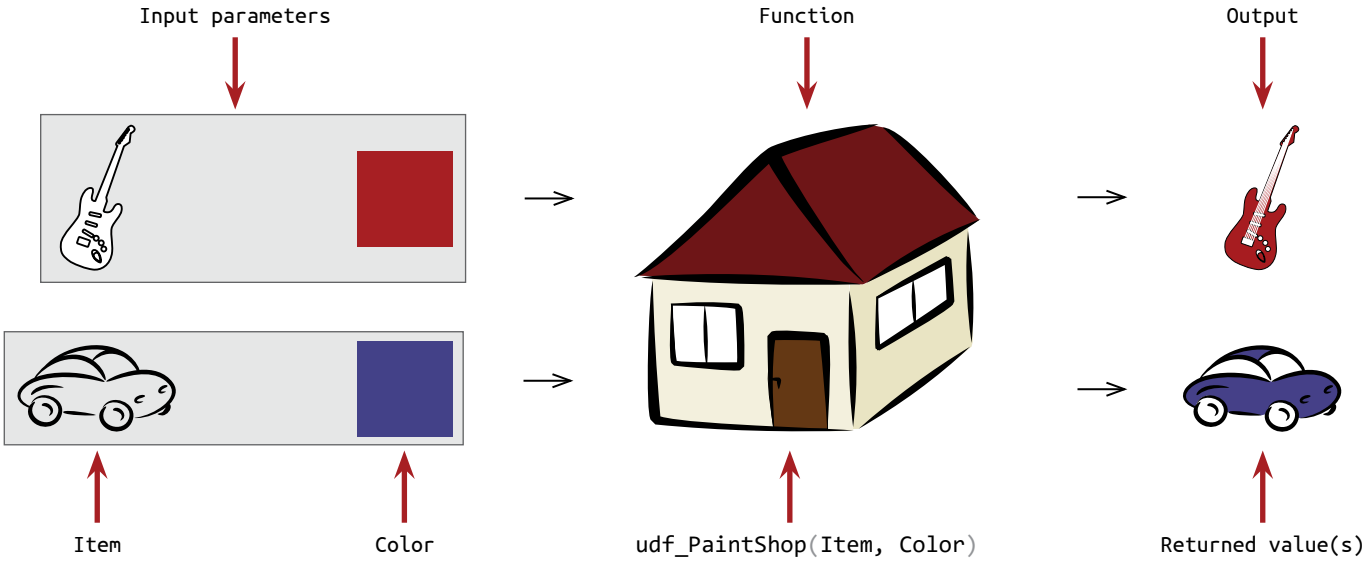
- Edit the code
- Execute the code (press F5)

To browse the **Views** in the **Object Catalog**:

```
SELECT *
FROM LearnSQLServerIntuitively.sys.views;
GO

SELECT *
FROM LearnSQLServerIntuitively.sys.objects
WHERE [type] = 'V';
GO
```

The function:

- Is a DB object, stored in the DBE
- Is encapsulated and reusable code that extends the functionality of T-SQL
- Is created by us (User-defined functions) or installed together with SQL Server (Built-in and System functions)
- Requires or does not require input parameter(s) (parametrized function) with or without default values
- Outputs single value (scalar-valued function) or recordset (table-valued function)
- Returns the same (deterministic function) or different (non deterministic function) value(s) on every execution



The paint shop may act like a function.
We can input the item (car, guitar) and define the printing color (blue, red).
On the output is the painted item.

We can define a calculation that is often used, encapsulate it in a function and call it when needed.

Input	udf__CalculateRebate (Function, data manipulation)	Output
RebatePercent = 0.8 SalesValue = 5324.73	(5324.73 * 0.8)	4,259.784

Input Parameters

When we create a function, we define if:

- It requires input parameter(s) or not
  - A function that removes the time from `DATETIME` data type (1963-04-17 18:23:45.517 to 1963-04-17

00:00:00.000), doesn't need input parameter
    - A function that divides two numbers and avoids divide by zero needs two input parameters – one for the dividend and one for the divisor
- The parameters have DEFAULT values (optional parameters)
    - When we call the function, can omit the values for the parameters that have DEFAULT. In this case the default value is used
    - The parameters without default value can't be omitted

Function data manipulation
The code of the function manipulates the data. Similar to the view and the stored procedure, the function's code is called (function's) definition.
The definition of the function is not visible for the code that calls the function and the complexity of the function's code is hidden.

Output
The function returns a result, which type define the function as:
- Scalar-valued – returns single value
- Inline Table-valued – returns multiple values (table), created with a single statement
- Multi-statement Table-valued – returns multiple values (table), created with multiple statements

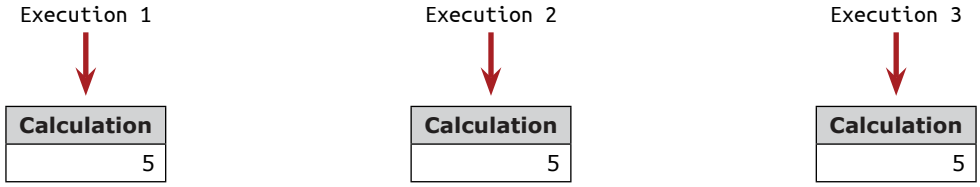
Deterministic and Non Deterministic Functions

The result of a function may be the same or not the same on every execution.

	Deterministic	Non Deterministic	
Time asked	**How much is 2 + 3?**	**What is the time now?**	**Give me a random number**
1	5	1966-04-27 10:53:47.657	0.254343874234445
2	5	1967-07-03 11:08:07.370	0.982723166477615
3	5	1968-11-05 09:23:15.060	0.818569863640294

Deterministic
The result of **How much is 2 + 3?** is always the same (5):

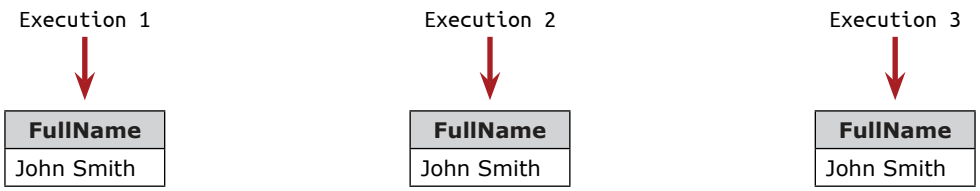
```
SELECT (2 + 3) AS Calculation;
GO
```



The function that calculates 2 + 3 is **deterministic**.

The built-in function that concatenates customer's **`FirstName`** and **`LastName`** is also **deterministic**:

```
SELECT CONCAT('John', ' ', 'Smith') AS FullName;
GO
```



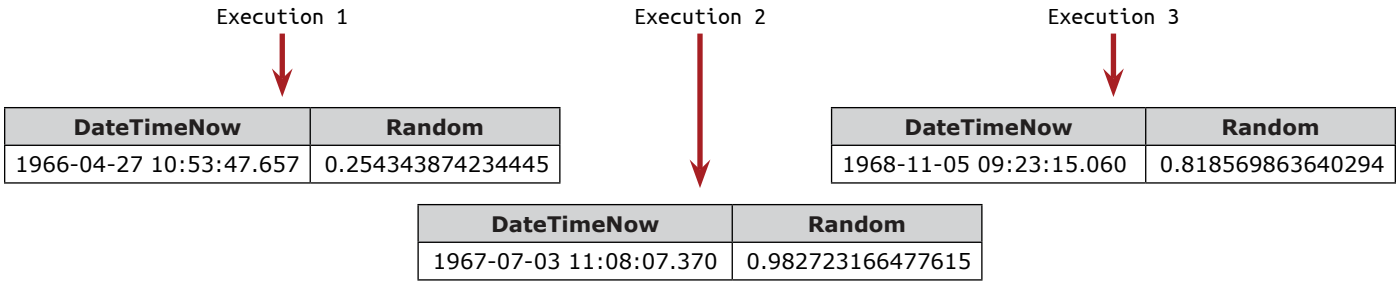
Returns the same result on every execution, unless the data in the source DB object is changed.

Non Deterministic
The results of

- What is the time now?
- Give me a random number

are different on every execution of the functions:

```
SELECT
  GETDATE() AS DateTimeNow
  , RAND() AS Random;
GO
```



`GETDATE()` and `RAND()` are **non deterministic**.

To list the types of the user-defined functions covered in this book in the **Object Catalog**:

```
SELECT DISTINCT [type], type_desc
FROM LearnSQLServerIntuitively.sys.objects
WHERE [type] IN ('FN', 'TF', 'IF');
GO
```

The types of the functions are:

type	type_desc
FN	SQL_SCALAR_FUNCTION
IF	SQL_INLINE_TABLE_VALUED_FUNCTION
TF	SQL_TABLE_VALUED_FUNCTION

Before we create an object with a DDL statement, we clean up

```
DROP FUNCTION IF EXISTS SchemaName.udf_FunctionName;
GO
```

`SchemaName.ObjectName` naming convention

Comparison of the function types, based on the output:

	Scalar-valued	Inline Table-valued	Multi-statement Table-valued
Input parameters	Yes, No	Yes, No	Yes, No
`DEFAULT` values	Yes, No	Yes, No	Yes, No
Output	Single value of a specified data type	Recordset (multiple columns, multiple data types)	Recordset (multiple columns, multiple data types)
Function's definition			
Is wrapped in `BEGIN`... `END` block?	Yes	No	Yes
Temp Tables in the function's definition			
Local and Global (`#LocalTempTable`, `##GlobalTempTable`)	No	No	No
Table Variable (`@TableVariable`)	Yes	No	Yes
Statements in the function's definition			
Is the definition single statement?	No	Yes	No
`SELECT`	Yes	Yes	Yes
`INSERT`, `UPDATE`, `DELETE`	Only manipulate table variable, created in the definition	No	Only to manipulate the table variable that is `INSERT`ed
Execution			
The function can be called in statement			
`SELECT`	Yes	Yes (in subquery)	Yes (in subquery)
`FROM`	Yes (in the `ON` clause)	Yes	Yes
`WHERE`	Yes	Yes (in subquery)	Yes (in subquery)
`GROUP BY`	Yes	No	No
`ORDER BY`	Yes	Yes	Yes
Is executed on every row?	Yes	No	No
Steps	`RETURN` (*Value*) or 1. `DECLARE @ReturnVariable;` 2. `SET @ReturnVariable = ...;` 3. `RETURN @ReturnVariable;`	`RETURN` (*T-SQL to generate the returned recordset*);	1. Create `@TableVariable` 2. Execute multiple statements to populate the `@TableVariable` 3. `RETURN`

Scalar-valued Functions

In mathematics the division by zero is undefined. The code:

```
SELECT (15 / 0) AS Divide;
GO
```

returns error message "Divide by zero error encountered.".

We can create a function udf_Divide() and use it everywhere we need to divide. It accepts two input parameters (@Dividend and @Divisor) and returns a single value - which is the result of the division. When the divisor is NULL or 0, the returned value is NULL.

```
                                   Function parameters
CREATE FUNCTION dbo.udf_Divide(@Dividend DECIMAL(38, 6), @Divisor DECIMAL(38, 6))
RETURNS DECIMAL(38, 6)      Data type of the value that the function returns      Parameter name and data type
AS                                  AS is optional and can be omitted
BEGIN
  RETURN (CASE WHEN ISNULL(@Divisor, 0) != 0 THEN (@Dividend / @Divisor) END);   Function Definition
END
GO
```

The functions in the **Object Catalog**:

Name	type	type_desc
udf_Divide	FN	SQL_SCALAR_FUNCTION

To use a scalar-valued function, we add it in the SELECT, ON (in FROM), WHERE, GROUP BY or ORDER BY clause.

Test divide by zero:

```
SELECT LearnSQLServerIntuitively.dbo.udf_Divide(15, 0) AS Division;
GO
```

Division
NULL

Test divide:

```
SELECT LearnSQLServerIntuitively.dbo.udf_Divide(15, 0.35) AS Division;
GO
```

Division
42.857142

Data sources for the examples below:

Customers

CustomerID	FirstName	LastName	Email	DateTimeRegistered
1	Anabel	Larson	anabel.larson@customer5.info	1966-08-24 05:58:58.983
2	Anna	Laurier	anna.laurier@customer2.net	1967-09-09 16:43:46.510
3	Beverly	NULL	NULL	1967-10-15 16:57:11.237
4	John	Smith	john.smith@customer1.com	1965-07-12 17:20:31.277
5	John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667

Sales

SaleID	CustomerID	ItemID	Quantity	Price	DateTimeOfSale
1	3	1004	3	1.24	1967-10-17 04:37:37.487
2	5	5273	5	2.43	1968-05-08 20:44:01.887
3	4	5432	2	12.44	1965-09-04 22:13:09.130
4	3	8541	7	5.88	1968-12-17 08:46:12.740
5	5	9514	1	28.19	1968-03-04 10:28:11.877

We can create a user-defined scalar-valued function that returns the last sale value for a specified customer:

```
CREATE FUNCTION dbo.udf_LastSaleValueForCustomerID(@CustomerID INT)
RETURNS MONEY
AS
BEGIN
  DECLARE @LastSaleValue MONEY;

  SET @LastSaleValue =
    (
      SELECT TOP 1 (Quantity * Price)
      FROM LearnSQLServerIntuitively.dbo.Sales
      WHERE CustomerID = @CustomerID
      ORDER BY DateTimeOfSale DESC
    )

  RETURN @LastSaleValue;
END
GO
```

The same data type

Creates a variable to store the value that the function returns (the same data type like the returned variable)

Assign value to the variable

Can't use the SELECT statement directly to return a value

The last statement is RETURN to return the value from the variable to the caller of the function

Test the function for `CustomerID = 3`:

```
SELECT LearnSQLServerIntuitively.dbo.udf_LastSaleValueForCustomerID(3);
GO
```

LastSaleValue
41.16

The code below uses `udf_LastSaleValueForCustomerID()` and `udf_Divide()` functions. The result of `udf_LastSaleValueForCustomerID()` is passed as parameter in the `udf_Divide()` function:

```
SELECT
  C.CustomerID
  , C.FirstName
  , C.LastName
  , AVG(Quantity * Price) AS Avg_SalesValue
  , LearnSQLServerIntuitively.dbo.udf_LastSaleValueForCustomerID(C.CustomerID) AS Last_
```

```
SaleValue
  , LearnSQLServerIntuitively.dbo.udf_Divide
    (
      AVG(Quantity * Price)
      , LearnSQLServerIntuitively.dbo.udf_LastSaleValueForCustomerID(C.CustomerID)
    )
  AS Percentage
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
GROUP BY
  C.CustomerID
  , C.FirstName
  , C.LastName
ORDER BY CustomerID;
GO
```

@Divident (Avg_SalesValue)

@Divisor (Last_SaleValue) - Nested functions

(Avg_SalesValue / Last_SaleValue) = Percentage

Custo-merID	First-Name	Last-Name	Avg_Sales-Value	Last_Sa-leValue	Percent-age
3	Beverly	NULL	22.44	41.16	0.545189
4	John	Smith	24.88	24.88	1.000000
5	John	Smith	20.17	12.15	1.660082

To define the DEFAULT value of a parameter, we add it to the parameter name and data type.

We can use NULL as DEFAULT and handle the default value (NULL) in the function's definition:

```
CREATE FUNCTION dbo.udf_DateNoTime(@DateTime DATE = NULL)
RETURNS DATE
AS
BEGIN
  DECLARE @Date DATE;

  IF (@DateTime IS NULL)
  BEGIN SELECT @DateTime = GETDATE(); END

  SELECT @Date = CONVERT(CHAR(10), @DateTime, 120);

  RETURN @Date;
END
GO
```

The default value for @DateTime parameter is NULL

Handle the default parameter - replace NULL with **now**

Call the function with a specified parameter value:

```
SELECT LearnSQLServerIntuitively.dbo.udf_DateNoTime ('1968-10-08 13:45:18.277') AS DateNoTime;
GO
```

DateNoTime
1968-10-08

Calling the function with the DEFAULT value (NULL) returns the date of **now**:

```
SELECT LearnSQLServerIntuitively.dbo.udf_DateNoTime (DEFAULT) AS DateNoTime;
GO
```

Use the default value (today is 1966-04-28).

DateNoTime
1966-04-28

If the keyword DEFAULT is omitted, the DEFAULT value is not passed and an error message "An insufficient number of arguments were supplied for the procedure or function LearnSQLServerIntuitively.dbo.udf_DateNoTime." is returned.

Pass a column value as input parameter:

```
SELECT
  SaleID
  , CustomerID
  , ItemID
  , DateTimeOfSale
  , LearnSQLServerIntuitively.dbo.udf_DateNoTime(DateTimeOfSale) AS DateOfSale
FROM LearnSQLServerIntuitively.dbo.Sales
ORDER BY CustomerID;
GO
```

Date and time

Date only

SaleID	CustomerID	ItemID	DateTimeOfSale	DateOfSale
1	3	1004	1967-10-17 04:37:37.487	1967-10-17
4	3	8541	1968-12-17 08:46:12.740	1968-12-17
3	4	5432	1965-09-04 22:13:09.130	1965-09-04
5	5	9514	1968-03-04 10:28:11.877	1968-03-04
2	5	5273	1968-05-08 20:44:01.887	1968-05-08

The DEFAULT value can not be additionally manipulated in the function's definition:

```
CREATE FUNCTION dbo.udf_Calculation(@Value1 INT, @Value2 INT = 15)
RETURNS INT
AS
BEGIN RETURN (@Value1 + @Value2); END
GO
```

Default = 15

Call with specified values:

```
SELECT LearnSQLServerIntuitively.dbo.udf_Calculation (7, 12) AS Calculation;
GO
```

7 + 12 = 19

Calculation
19

Call with DEFAULT value:

```
SELECT LearnSQLServerIntuitively.dbo.udf_Calculation (7, DEFAULT) AS Calculation;
GO
```

7 + 15 = 22

Calculation
22

Call and pass DEFAULT to parameter that doesn't have specified DEFAULT value:

```
SELECT LearnSQLServerIntuitively.dbo.udf_Calculation (DEFAULT, 12) AS Calculation;
GO
```

Default value for the first parameter

Calculation
NULL

An example of a scalar-valued function that has no parameter and returns today's date (no time):

```
CREATE FUNCTION dbo.udf_DateNow()
RETURNS DATE
AS
BEGIN
  DECLARE @DateNow DATE;

  SELECT @DateNow = GETDATE();

  RETURN @DateNow;
END
GO
```

No parameter

Assigning DATETIME data type to DATE data type cuts the time

Call the function without parameter (brackets only):

```
SELECT LearnSQLServerIntuitively.dbo.udf_DateNow() AS DateToday;
GO
```

Today is 1966-04-28

DateToday
1966-04-28

Table-valued Functions

The table-valued functions return a recordset.

Inline
The inline table-valued function is constructed by one statement.

This function returns the total sales for the customers:

```
CREATE FUNCTION dbo.udf_SumSalesValue()
RETURNS TABLE
AS
  RETURN
  (
    SELECT
      CustomerID
      , SUM(Quantity * Price) AS Sum_SaleValue
    FROM LearnSQLServerIntuitively.dbo.Sales
    GROUP BY CustomerID
  )
GO
```

Single statement

Call the function:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.udf_SumSalesValue();
GO
```

CustomerID	Sum_SalesValue
3	44.88
4	24.88
5	40.34

Collect the totals of **SaleValue** for the customers and calculate the percentage of the current sale of the total sales:

```
SELECT
  C.CustomerID
  , C.FirstName
  , C.LastName
  , LearnSQLServerIntuitively.dbo.udf_DateNoTime(S.DateTimeOfSale) AS DateOfSale
  , (S.Quantity * S.Price) AS SaleValue
  , SSV.Sum_SaleValue
  , LearnSQLServerIntuitively.dbo.udf_Divide
    (
      (S.Quantity * S.Price)
      , SSV.Sum_SaleValue
    )
  AS Percentage
FROM
```

Scalar-valued function

Scalar-valued function

```
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
  JOIN LearnSQLServerIntuitively.dbo.udf_SumSalesValue() AS SSV
    ON C.CustomerID = SSV.CustomerID
ORDER BY CustomerID;
GO
```

Table-valued function

Implement the table-valued function by JOINing it in the FROM clause:

Custo-merID	First-Name	Last-Name	DateOfSale	SaleV-alue	Sum_ SaleV-alue	Percent-age
3	Beverly	NULL	1967-10-17	3.72	44.88	0.082887
3	Beverly	NULL	1968-12-17	41.16	44.88	0.917112
4	John	Smith	1965-09-04	24.88	24.88	1.000000
5	John	Smith	1968-05-08	12.15	40.34	0.301189
5	John	Smith	1968-03-04	28.19	40.34	0.698810

3.72 / 44.88 = 0.082887

Multi-Statement

Creates a recordset by executing multiple statements. Function udf_EligibleCustomers() returns **CustomerID** and **Sum_SalesValue** for customers that have a total **SaleValue** greater then the parametrized threshold value.

```
CREATE FUNCTION dbo.udf_EligibleCustomers (@ThresholdValue MONEY)
RETURNS
  @Avg_SalesValuesLast3Customers TABLE
    (
      CustomerID INT
      , Sum_SalesValue MONEY
    )
AS
BEGIN
  DECLARE @EligibleCustomers TABLE (CustomerID INT);
  INSERT @EligibleCustomers (CustomerID)
  SELECT CustomerID
  FROM LearnSQLServerIntuitively.dbo.Sales
  GROUP BY CustomerID
  HAVING SUM(Quantity * Price) > @ThresholdValue;

  INSERT @Avg_SalesValuesLast3Customers (CustomerID, Sum_SalesValue)
  SELECT CustomerID, SUM(Quantity * Price) AS Sum_SalesValue
  FROM LearnSQLServerIntuitively.dbo.Sales
  WHERE
    CustomerID IN
```

Specifies the structure of the table variable that the function returns

Statement 1 (Table variable in the function's definition)

Statement 2 (Populate the table variable)

Statement 3 (Populates the table variable that the function returns. Filter with the values from the other table variable @EligibleCustomers)

```
    (
      SELECT CustomerID
      FROM @EligibleCustomers
    )
  GROUP BY CustomerID;

  RETURN
END
GO
```

Call the function with `@ThresholdValue = 30`:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.udf_EligibleCustomers(30);
GO
```

CustomerID	Sum_SalesValue
3	44.88
5	40.34

Call the function with `@ThresholdValue = 41`:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.udf_EligibleCustomers(41);
GO
```

CustomerID	Sum_SalesValue
3	44.88

Implement the function **udf_EligibleCustomers** in the FROM clause and calculate the **DiscountedValue** only for the eligible customers (the ones who have total **SalesValue** above the threshold):

```
SELECT
  C.CustomerID
  , C.FirstName
  , C.LastName
  , LearnSQLServerIntuitively.dbo.udf_DateNoTime (S.DateTimeOfSale) AS DateOfSale
  , (S.Quantity * S.Price) AS SalesValue
  , EC.Sum_SalesValue
  , CASE
    WHEN EC.Sum_SalesValue IS NOT NULL THEN ((S.Quantity * S.Price) * 0.9)
  END AS DiscountedValue
FROM
  LearnSQLServerIntuitively.dbo.Customers AS C
  JOIN LearnSQLServerIntuitively.dbo.Sales AS S
    ON C.CustomerID = S.CustomerID
  LEFT JOIN LearnSQLServerIntuitively.dbo.udf_EligibleCustomers(41) AS EC
    ON C.CustomerID = EC.CustomerID
ORDER BY CustomerID;
GO
```

The discounted value is 90%

Only the eligible customers

Table-valued function. Parameter to filter the data.

Custo-merID	First-Name	Last-Name	DateOfSale	Sales-Value	Sum_ Sales-Value	Discount-edValue
3	Beverly	NULL	1967-10-17	3.72	44.88	3.34800
3	Beverly	NULL	1968-12-17	41.16	44.88	37.04400
4	John	Smith	1965-09-04	24.88	NULL	NULL
5	John	Smith	1968-03-04	28.19	NULL	NULL
5	John	Smith	1968-05-08	12.15	NULL	NULL

Functions Nesting

Nest one function inside another. The inner function is executed first and its result is used as a parameter for the outer function.

```
SELECT RIGHT(LEFT('This is My String', 14), 6) AS Result;
GO
```

Result
My Str

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
String	T	h	i	s		i	s		M	y		S	t	r	i	n	g
LEFT(*String*, 14)	Left 14 characters																
RIGHT(*String*, 6)									Right 6 characters								

The result of `LEFT()` (`'This is My Str'`) is used as parameter for `RIGHT()`.

Limitations:

- Nesting maximum 32 levels
- User-fefined functions: Maximum 2100 input parameter
- No special characters in the name

Stored Procedures

The Stored Procedure (SP):

- Is the most powerful programmable object in SQL Server
- Encapsulates reusable code, stored in the DBE
- Uses the full capacity of T-SQL. All the statements we learned so far can be used in the SP definition, except:
  - `USE`
  - `CREATE SCHEMA`
  - `CREATE`, `ALTER VIEW`
  - `CREATE`, `ALTER FUNCTION`
  - `CREATE`, `ALTER PROC`
- Can call another SP
- Can be parametrized:
  - With input parameters (`DEFAULT`ed or not)
  - With output parameters
- Returns status value (success or error message)

The SPs in the **Object catalog**:

```
SELECT DISTINCT name, [type], type_desc
FROM LearnSQLServerIntuitively.sys.objects
WHERE [type] IN ('P', 'X');
GO
```

```
SELECT DISTINCT *
FROM LearnSQLServerIntuitively.sys.procedures
GO
```

P - SQL Stored Procedure
X - Extended Stored Procedure

Naming Conventions for SPst

We can add the following rules, that are specific for SPs to the Naming Convention rules:

- SP (and every other DB object) name is a maximum of 128 characters
- Avoid sp_ prefix. It is used by system SPs
- Add the group where the SP belongs as prefix:
  - main_ - general use
  - rep_ - reporting
  - etl_ - ETL
  - task_ - called by SQL Job Agent, Windows scheduled job etc
- Add the action (what the SP is doing) as suffix:
  - _Select - `SELECT`s data
  - _Insert - `INSERT`s data
  - _Update - `UPDATE`s data
  - _Delete - `UPDATE`s data

Tables, used as data source in the examples:

Customers

CustomerID	FirstName	LastName	Email	DateTimeRegistered
1	Anabel	Larson	anabel.larson@customer5.info	1966-08-24 05:58:58.983
2	Anna	Laurier	anna.laurier@customer2.net	1967-09-09 16:43:46.510
3	Beverly	NULL	NULL	1967-10-15 16:57:11.237
4	John	Smith	john.smith@customer1.com	1966-07-12 17:20:31.277
5	John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667

Items

ItemID	ItemCode	ItemDescription	UnitOfMeasure	Price
1	ID52-P1	Interactive Doll - Pink	Pcs	12.42
2	EG5-WW58	Word Whammer	Kg	49.38
3	BT-D521	Bath Toy Duck	Box	0.58
4	BT-S512	Bath Toy Forms (set)	Pcs	5.68
5	B-P12_G	Bike for Girls (Purple)	Pcs	124.86

Sales

SaleID	CustomerID	ItemID	Quantity	Price	DateTimeOfSale
1	3	1	3	1.24	1967-10-17 04:37:37.487
2	5	3	5	2.43	1968-05-08 20:44:01.887
3	4	2	2	12.44	1965-09-04 22:13:09.130
4	3	4	7	5.88	1968-12-17 08:46:12.740
5	5	5	1	28.19	1968-03-04 10:28:11.877
6	2	2	12	12.28	1966-08-05 15:23:47.233
7	1	3	5	2.28	1969-11-14 07:18:52.667

CREATE PROC (PROCEDURE).

Creates a new SP.

Has to be the first statement in the batch.

Before creating a new SP, delete the SP with the same name (if it exists):

```
DROP PROC IF EXISTS SchemaName.udf_StoredProcedureName;
GO
```

SchemaName.ObjectName naming convention

The syntax is:

```
CREATE PROC SchemaName.StoredProcedureName
  (
    @Parameter1 INT
    , @Parameter2 DECIMAL(18, 6) OUT
  )
AS
BEGIN
  SET NOCOUNT ON;

  ... Statement 2 ...;
  ... Statement 3 ...;
  ... Statement 4 ...;
END
GO
```

SchemaName.ObjectName naming convention

Parameter(s) list

OUT or OUTPUT(output parameter)

SP definition

NOCOUNT Turns off the messages that informs how many rows are manipulated in a DML statements and avoid unnecessary network traffic

CREATE PROC is followed by the parameter(s) list, if the SP is parametrized. In the BEGIN… END block is the SP definition – the code that the SP executes. The RETURN keyword inside the SP definition terminates the execution of the SP.

To create a SP that has no input parameters and selects all customers, ordered from the last to the first registered:

```
CREATE PROC dbo.usp_Customers_Select
AS
BEGIN
  SET NOCOUNT ON;

  SELECT
    FirstName
    , LastName
    , Email
    , DateTimeRegistered
  FROM LearnSQLServerIntuitively.dbo.Customers
  ORDER BY DateTimeRegistered DESC;
END
GO
```

DSO naming convention is allowed in the stored procedure's definition

After the SP is created, we can EXECute it:

```
EXEC LearnSQLServerIntuitively.dbo.usp_Customers_Select;
GO
```

First-Name	Last-Name	Email	DateTimeRegistered
Beverly	NULL	NULL	1967-10-15 16:57:11.237
Anna	Laurier	anna.laurier@customer2.net	1967-09-09 16:43:46.510
Anabel	Larson	anabel.larson@customer5.info	1966-08-24 05:58:58.983
John	Smith	john.smith@customer1.com	1966-07-12 17:20:31.277
John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667

Parametrized SP

The parameters are variables that give us the option to pass input values into the SP definition.
The parameters add flexibility to the logic of the SP.
To add parameters to a SP, we add the parameter(s) list after the `CREATE PROC` DDL statement and we define their names and data types.

To create a SP that apply the CRUD DML statements on the **Customers** table, we execute the `CREATE PROC` DDL statement:

Prefix ⟶ type of the DB object
SPName - Name of the manipulated object
Suffix ⟶ Action

No `DEFAULT`. Value has to be provided on execution (mandatory parameter)

```
1. CREATE PROC dbo.usp_Customers_CRUD
2.  (
3.    @Action CHAR(1) -- C = INSERT, R = SELECT, U = UPDATE, D = DELETE
4.    , @CustomerID INT = NULL
5.    , @FirstName NVARCHAR(32) = NULL
6.    , @LastName NVARCHAR(64) = NULL
7.    , @Email VARCHAR(128) = NULL
8.  )
9. AS
10. BEGIN -- usp_Customers_CRUD
11. SET NOCOUNT ON;
12.
13. -- Error for mandatory parameter and exit
14. IF (@Action IN ('R', 'U', 'D') AND @CustomerID IS NULL)
15. BEGIN
16.   RAISERROR('Parameter @CustomerID is NULL', 11, 1);
17.   RETURN;
18. END
19.
20. IF (@Action = 'C')
21.   BEGIN
```

Parameters list (line 3 to 7)

`NULL` is the `DEFAULT` value. The SP can be executed without providing a value for these parameters

The `data type` matches the destination data type (columns **FirstName**, **LastName** and **Email** in table **Customers**)

Add comments to facilitate the future editing of the SP

Handle the parameters and their default values in the beginning of the SP definition (lines 14 to 18)

`RETURN` terminates the execution

```
22.     INSERT LearnSQLServerIntuitively.dbo.Customers
23.       (
24.         FirstName
25.         , LastName
26.         , Email
27.         , DateTimeRegistered
28.       )
29.     SELECT
30.       @FirstName
31.       , @LastName
32.       , @Email
33.       , GETDATE();
34.   END
35.
36. IF (@Action = 'R')
37.   BEGIN -- @Action = 'R'
38.     SELECT
39.       FirstName
40.       , LastName
41.       , Email
42.       , DateTimeRegistered
43.     FROM LearnSQLServerIntuitively.dbo.Customers
44.     WHERE CustomerID = ISNULL(@CustomerID, CustomerID);
45.   END -- @Action = 'R'
46.
47. IF (@Action = 'U')
48.   BEGIN
49.     UPDATE LearnSQLServerIntuitively.dbo.Customers
50.     SET
51.       FirstName = @FirstName
52.       , LastName = @LastName
53.       , Email = @Email
54.     WHERE CustomerID = @CustomerID;
55.   END
56.
57. IF (@Action = 'D')
58.   BEGIN
59.     DELETE LearnSQLServerIntuitively.dbo.Sales
60.     WHERE CustomerID = @CustomerID;
```

If the value of the parameter @Action is:

- 'C' (Create), an INSERT
- 'R' (Read), an SELECT
- 'U' (Update), an UPDATE
- 'D' (Delete), an DELETE

DML statement is executed

Mark with comments BEGIN and the matching END (line 37 and 45)

NULL = All Customers

CustomerID is the **FK** in table **Sales**. The **FK**s have to be **deleted before** deleting the **PK**

```
61.
62.     DELETE LearnSQLServerIntuitively.dbo.Customers
63.     WHERE CustomerID = @CustomerID;
64.   END
65. END -- usp_Customers_CRUD
66. GO
```

After the **FK** rows are **deleted** from **all the dependent** tables, the **PK can be deleted**

Add comments to the parameter(s) list and SP definition to facilitate the future editing and understanding of the SP.

Looks scary, but it is not!

- The parameter(s) list defines the parameters (variables)
- To assign a `DEFAULT` value to a parameter, we add the **equal to** sign and the `DEFAULT` **value**
- The parameters that have a `DEFAULT` value can be omitted when executing the SP
- The parameters that don't have a `DEFAULT` value are **mandatory** and can't be omitted when `EXEC`uting the SP
- The `DEFAULT` value can be `NULL` or any value, compatible with the data type of the variable (the parameter)

First we check the parameters. In `usp_Customers_CRUD` the only check is if a value for the parameter `@CustomerID` is passed for Read, Update or Delete actions (line 14). If the result it True (no value):

- `RAISERROR()` - returns custom error message
- `RETURN` - terminates the execution of the SP

In this simple SP we have just a few consecutive statements and we execute only three of them:

line 11
and lines 16, 17 (if `@CustomerID` is omitted for Read, Update, Delete actions)
and line 22 (on Create action)
or line 38 (on Read action)
or line 49 (on Update action)
or line 59, 62 (on Delete action)

With the `IF` condition we decide which portion of the code is to be executed. The `@Action` parameter determines the CRUD (Create, Read, Update or Delete a row in a table) cases:

- To **Create** a new customer, we need values for the parameters `@FirstName`, `@LastName` and `@Email`. `@CustomerID` is not needed, it is created automatically (Column **`CustomerID`** in **`Customers`** table is `IDENTITY(1, 1)` i.e. auto incremental). In column **`DateOfRegistration`** we insert **now**
- The **Read** action `SELECT`s the row where `CustomerID = @CustomerID` from table **`Customers`**. When `@CustomerID` is `NULL`, the right operand in the filtering condition is replaced with `CustomerID` (`CustomerID = CustomerID`) and all the customers are selected.
- The **Update** action updates the attributes (the columns **`FirstName`**, **`LastName`**, **`Email`**) for `@CustomerID`
- The **Delete** action executes two statements:
  - The first one deletes the rows for `@CustomerID` from the table **`Sales`**. This is needed, because we can't delete a **Primary Key** if a F**oreign Key exists**.
  - The second statement deletes `@CustomerID` from table **`Customers`**.

EXEC PROC (EXECUTE PROCEDURE)

To execute a SP, we use EXEC PROC (EXECUTE PROCEDURE)

If we EXECute a SP without specifying value(s) for the parameters which don't have DEFAULT values:

```
EXEC LearnSQLServerIntuitively.dbo.usp_Customers_CRUD;
GO
```

the error message "Procedure or function 'usp_Customers_CRUD' expects parameter '@Action', which was not supplied." is returned.

We can EXECute parametrized SP in two ways:

1. Pass the parameter(s) value(s) in the exact order as they are defined in the parameter(s) list:

```
EXEC LearnSQLServerIntuitively.dbo.usp_Customers_CRUD
  'R'                                          The value for Parameter 1 is in Position 1
  , 5                                          The value for Parameter 2 is in Position 1
  , DEFAULT
  , DEFAULT
  , DEFAULT;                                   The value for Parameter 5 is in Position 1
GO
```

FirstName	LastName	Email	DateTimeRegistered
John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667

If we use the DEFAULT value for the last parameter, we can skip the DEFAULT keyword:

```
EXEC LearnSQLServerIntuitively.dbo.usp_Customers_CRUD
  'R'                                          Parameter 1 in Position 1
  , 5;                                         Parameter 2 in Position 2
GO
```

We can execute a SP without the keyword EXEC, but this is not a good practice, because:

- The code is not easy to understand
- We can't search the batch for a SP by the keyword EXEC

```
LearnSQLServerIntuitively.dbo.usp_Customers_CRUD
  'R'
  , 5;
GO
```

For any other than the last parameter, we can't skip the keyword DEFAULT, because the next parameter takes the previous position:

```
EXEC LearnSQLServerIntuitively.dbo.usp_Customers_CRUD
  'C'              ← @Action
  , DEFAULT        ← @CustomerID   ← If we skip this DEFAULT, the next parameter is used in position 2
  , 'Antoine'      ← @FirstName      (the value 'Antoine' is passed for parameter @CustomerID
  , 'Kern';        ← @LastName
GO                                   ← The last parameter @Email can be omitted

SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers;
GO
```

CustomerID	FirstName	LastName	Email	DateTimeRegistered
1	Anabel	Larson	anabel.larson@customer5.info	1966-08-24 05:58:58.983
2	Anna	Laurier	anna.laurier@customer2.net	1967-09-09 16:43:46.510
3	Beverly	NULL	NULL	1967-10-15 16:57:11.237
4	John	Smith	john.smith@customer1.com	1966-07-12 17:20:31.277
5	John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667
6	Antoine	Kern	NULL	1969-05-13 21:19:01.747

2. Pass the parameter(s) value and the parameter(s) name:

When we EXECute a SP with parameter(s) name and value(s), we do not have to list them in order:

```
EXEC LearnSQLServerIntuitively.dbo.usp_Customers_CRUD
  @CustomerID = 5
  , @Action = 'R';
GO
```

FirstName	LastName	Email	DateTimeRegistered
John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667

To test the custom error, we EXECute the SP without passing a value for @CustomerID:

Execute this in a new SSMS window, starting from line 1. This helps the error message to return the correct line number.

```
1. EXEC LearnSQLServerIntuitively.dbo.usp_Customers_CRUD @Action = 'R';
2. GO
```

```
Msg 50000, Level 11, State 1, Procedure usp_Customers_CRUD, Line 17
Parameter @CustomerID is NULL
```

The error message points us to line 17. To debug the SP, we need to examine line 17.

We do this by EXECuting the system SP sp_helptext. The parameter that it accepts is **ObjectName**. The returned recordset is the object (Function, Stored Procedure) definition:

```
EXEC LearnSQLServerIntuitively.sys.sp_helptext 'LearnSQLServerIntuitively.dbo.usp_
Customers_CRUD';
GO
```

	Text
16	...
17	RAISERROR('Parameter @CustomerID is NULL', 11, 1);
18	...

To UPDATE a row, we provide values for the parameters that edit the values for the columns - **FirstName**, **LastName** and **Email**:

```
EXEC LearnSQLServerIntuitively.dbo.usp_Customers_CRUD
  @Action = 'U'
  , @FirstName = 'Jonathan'
  , @LastName = 'Gourmand'
  , @Email = 'jonathang@customer52341.org'
  , @CustomerID = 3;
GO
```

Verify table **Customers**:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers;
GO
```

Customers

CustomerID	FirstName	LastName	Email	DateTimeRegistered
1	Anabel	Larson	anabel.larson@customer5.info	1966-08-24 05:58:58.983
2	Anna	Laurier	anna.laurier@customer2.net	1967-09-09 16:43:46.510
3	Jonathan	Gourmand	jonathang@customer52341.org	1967-10-15 16:57:11.237
4	John	Smith	john.smith@customer1.com	1966-07-12 17:20:31.277
5	John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667
6	Antoine	Kern	NULL	2016-05-16 21:32:43.563

The way this SP is designed, if we omit a value for one of the parameters `@FirstName`, `@LastName` or `@Email`, its `DEFAULT` value (`NULL`) is used.

To `DELETE` all the rows for `CustomerID = 3` in table **Sales** and the row in table **Customers**, we `EXEC`:

```
EXEC LearnSQLServerIntuitively.dbo.usp_Customers_CRUD
  @Action = 'D'
  , @CustomerID = 3;
GO
```

Verify table **Sales**:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Sales;
GO
```

Sales

SaleID	CustomerID	ItemID	Quantity	Price	DateTimeOfSale
2	5	3	5	2.43	1968-05-08 20:44:01.887
3	4	2	2	12.44	1965-09-04 22:13:09.130
5	5	5	1	28.19	1968-03-04 10:28:11.877
6	2	2	12	12.28	1966-08-05 15:23:47.233
7	1	3	5	2.28	1969-11-14 07:18:52.667

Verify table **Customers**:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.Customers;
GO
```

`CustomerID = 3` is deleted from tables **`Sales`** and **`Customers`**.

Customers

CustomerID	FirstName	LastName	Email	DateTimeRegistered
1	Anabel	Larson	anabel.larson@customer5.info	1966-08-24 05:58:58.983
2	Anna	Laurier	anna.laurier@customer2.net	1967-09-09 16:43:46.510
4	John	Smith	john.smith@customer1.com	1966-07-12 17:20:31.277
5	John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667
6	Antoine	Kern	NULL	2016-05-16 11:08:46.963

Output Parameter(s)

The OUTPUT parameter(s) return value(s) to the caller (SSMS, external application, etc.) of the SP.

To CREATE PROC with OUTPUT parameter(s), we:

- Add parameter(s) in the parameter(s) list with the OUT (or OUTPUT) keyword
- Assign a value to the variable in the SP definition

To EXECute a SP with an OUTPUT parameter(s):

- Create @Variable that stores the value of the OUTPUT parameter
- EXECute the SP and assign the value of the OUTPUT parameter to the @Variable
- Manipulate the @Variable (the value of the OUTPUT parameter)

The SP **usp_Sales** returns a summarized or detailed recordset for specified customer, item and sale period.
Create usp_Sales:

```
1. CREATE PROC rep.usp_Sales
2.  (
3.    @CustomerID INT = NULL -- Mandatory. Single value. Validated in the SP Definition
4.    , @ItemID INT = NULL -- NULL = All_Values
5.    , @Grain CHAR(1) = NULL -- S = Summary; D = Details
6.    , @DateStart DATE = NULL -- NULL is converted to now
7.    , @DateEnd DATE = NULL -- NULL is converted to tomorrow
8.    , @ReturnParameter_OUT NVARCHAR(64) = NULL OUT -- number of the manipulated rows
9.  )
10. AS
11. BEGIN
12.   SET NOCOUNT ON;
13.
14.   --** Handle Parameters - Start **--
15.   IF (@CustomerID IS NULL)
16.     -- Custom error
17.     BEGIN
```

Schema **rep** serves the reporting

The comments in the parameter(s) list and the object definition facilitate the future understanding and editing of the object

```
18.       RAISERROR('Parameter @CustomerID is NULL', 11, 1);
19.       RETURN;
20.     END
21.
22.   -- DEFAULT - Start
23.   IF (@Grain IS NULL)
24.     BEGIN SET @Grain = 'D'; END
25.
26.   IF (@DateStart IS NULL)
27.     BEGIN
28.       SELECT @DateStart = MIN(DateTimeOfSale)
29.       FROM LearnSQLServerIntuitively.dbo.Sales
30.       WHERE CustomerID = @CustomerID;
31.     END
32.
33.   IF (@DateEnd IS NULL)
34.     BEGIN
35.       SELECT @DateEnd = MAX(DateTimeOfSale)
36.       FROM LearnSQLServerIntuitively.dbo.Sales
37.       WHERE CustomerID = @CustomerID;
38.     END
39.
40.   SET @DateEnd = DATEADD(DD, 1, @DateEnd);
41.   -- DEFAULT - End
42.   --** Handle Parameters - End **--
43.
44.   --** Get the report - Start **--
45.   -- Details
46.   IF (@Grain = 'D')
47.     BEGIN
48.       SELECT
49.         C.FirstName
50.       , C.LastName
51.         , C.Email
52.         , C.DateTimeRegistered
53.         , I.ItemDescription
54.         , S.Quantity
55.         , S.Price
56.         , S.DateTimeOfSale
57.       FROM
58.         LearnSQLServerIntuitively.dbo.Sales AS S
```

```
59.       JOIN LearnSQLServerIntuitively.dbo.Customers AS C
60.         ON C.CustomerID = S.CustomerID
61.       JOIN LearnSQLServerIntuitively.dbo.Items AS I
62.         ON S.ItemID = I.ItemID
63.     WHERE
64.       C.CustomerID = @CustomerID
65.       AND S.ItemID = ISNULL(@ItemID,S. ItemID) -- NULL = All Items
66.       AND S.DateTimeOfSale >= @DateStart
67.       AND S.DateTimeOfSale < @DateEnd
68.     ORDER BY
69.       C.FirstName
70.       , C.LastName
71.       , I.ItemCode
72.       , S.DateTimeOfSale;
73.
74.     SELECT @ReturnParameter_OUT =
          'Grain: '
          + @Grain
          + '; Rows: '
          + CAST(@@ROWCOUNT AS VARCHAR);
75.   END
76.
77. -- Summary
78. IF (@Grain = 'S')
79.   BEGIN
80.     SELECT
81.       C.FirstName + ' ' + C.LastName AS CustomerName
82.       , SUM(S.Quantity * S.Price) As SalesValue
83.     FROM
84.       LearnSQLServerIntuitively.dbo.Sales AS S
85.       JOIN LearnSQLServerIntuitively.dbo.Customers AS C
86.         ON C.CustomerID = S.CustomerID
87.       JOIN LearnSQLServerIntuitively.dbo.Items AS I
88.         ON S.ItemID = I.ItemID
89.     WHERE
90.       C.CustomerID = @CustomerID
91.       AND S.ItemID = ISNULL(@ItemID, S.ItemID) -- NULL = All Items
92.       AND S.DateTimeOfSale >= @DateStart
93.       AND S.DateTimeOfSale < @DateEnd
94.     GROUP BY
95.     C.FirstName
```

```
96.         , C.LastName
97.       ORDER BY
98.         C.FirstName
99.         , C.LastName;
100.
101.      SELECT @ReturnParameter_OUT =
            'Grain: '
            + @Grain
            + '; Rows: '
            + CAST(@@ROWCOUNT AS VARCHAR);
102.    END
103.  --** Get the report - End **--
104.  END
105.  GO
```

On line 8 we add the parameter `@ReturnParameter_OUT` with a `DEFAULT` value of `NULL` (to be able to `EXEC`ute the SP without the `OUTPUT` functionality) and `OUT` keyword that specifies the parameter as `OUTPUT`.
On lines 74 and 101 we assign a value to the `OUTPUT` parameter - the number of rows in the recordset, returned by the SP.

The built-in function `@@ROWCOUNT` returns the number of the rows, manipulated by the last statement.

Additionaly the SP verifies the parameters:

- If `@CustomerID` doesn't have a value, the execution is terminated (lines 15 to 20)
- If `@Grain` is not specified, `'D'` (Details) is set (lines 23 and 24)
- If `@DateStart` and `@DateEnd` are not specified, the first (`MIN`) and last (`MAX`) sale dates for `@CustomerID` are picked (lines 26 to 38). Additional logic can be added in the case when `@CustomerID` has no sales. On line 40 we add one day to `@DateEnd` to avoid the usage of a function that removed the time from `DATETIME` data type in the WHERE `WHERE` clause
- Based on the value of the `@Grain` parameter, we execute different statements that create the recordsets for **Details** (lines 46 to 72) and **Summary** (lines 77 to 98)

We can `EXEC`ute the SP without using the `OUTPUT` functionality:

```
EXEC LearnSQLServerIntuitively.rep.usp_Sales
  @CustomerID = 5;
GO
```

Details (@Action parameter is not specified and the DETAILS value is used)

First-Name	Last-Name	Email	DateTimeRegis-tered	ItemDescription	Quan-tity	Price	DateTimeOfSale
John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667	Bike for Girls (Pur-ple)	1	28.19	1968-03-04 10:28:11.877
John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667	Bath Toy Duck	5	2.43	1968-05-08 20:44:01.887

EXECute the summary for @CustomerID = 4:

```
EXEC LearnSQLServerIntuitively.rep.usp_Sales
  @Grain = 'S'
  , @CustomerID = 4;
GO
```

CustomerName	SalesValue
John Smith	24.88

To handle the value, returned by the OUTPUT parameter(s), we need to follow this 3-steps logic:

Step 1: Create variable (@Out_ReturnParameter) that stores the value of the OUTPUT parameter:

```
DECLARE
  @Out_ReturnParameter NVARCHAR(64)
  , @DateTimeExecuted DATETIME = GETDATE();
```

Variable that stores the value of the OUTPUT parameter

Variable that stores the value of now. Inserted into the log table in Step 3

Step 2: EXECute the SP and populate the variable @Out_ReturnParameter:

```
EXEC LearnSQLServerIntuitively.rep.usp_Sales
  @CustomerID = 5
  , @ReturnParameter_OUT = @Out_ReturnParameter OUT;
```

Assign a value to @Out_ReturnParameter

First-Name	Last-Name	Email	DateTimeRegis-tered	ItemDescription	Quan-tity	Price	DateTimeOfSale
John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667	Bike for Girls (Pur-ple)	1	28.19	1968-03-04 10:28:11.877
John	Smith	john.smith@customer3.org	1964-09-02 17:27:07.667	Bath Toy Duck	5	2.43	1968-05-08 20:44:01.887

Step 3: Manipulate the value of the OUTPUT parameter:

```
INSERT LearnSQLServerIntuitively.dbo.StoredProceduresLogs
  (
    DateTimeExecuted
    , Details
```

Add row in the **Log**

```
  )
SELECT
  GETDATE()
  , @Out_ReturnParameter;
GO
```

Edit **Step 2** - the value of `@CustomerID` to 4, `@Grain` to `'S'` and `EXEC`ute the batch again:

```
EXEC LearnSQLServerIntuitively.rep.usp_Sales
  @Grain = 'S'
  , @CustomerID = 4
  , @ReturnParameter_OUT = @Out_ReturnParameter OUT;
```

CustomerName	SalesValue
John Smith	24.88

To verify the hahdling of the OUTPUT parameter in **Step 3**, we query the **Log** table:

```
SELECT *
FROM LearnSQLServerIntuitively.dbo.StoredProceduresLogs;
GO
```

LogID	DateTimeExecuted	Details
1	1968-05-17 11:15:59.973	Grain: D; Rows: 2
2	1968-06-05 23:48:05.633	Grain: S; Rows: 1

SP that EXECs another SP

SP **usp_SalesAndLog** `EXEC`utes another SP (**usp_Sales**) and inserts the value of the `OUTPUT` parameter (`@ReturnParameter_OUT`) in the log table. It collects and bypasses the parameters, needed to `EXEC`ute the inner SP.

Create usp_SalesAndLog that reproduces the 3-steps logic, explained above. **Step 2** bypasses the parameters from the outer SP to the inner SP:

```
CREATE PROC rep.usp_SalesAndLog
  (
    @CustomerID INT = NULL
    , @ItemID INT = NULL
    , @Grain CHAR(1) = NULL
    , @DateStart DATE = NULL
    , @DateEnd DATE = NULL
  )
AS
BEGIN
```

Bypass the parameters from the outer SP to the inner SP

```sql
SET NOCOUNT ON;

DECLARE
  @Out_ReturnParameter NVARCHAR(64)
  , @DateTimeExecuted DATETIME = GETDATE();

EXEC LearnSQLServerIntuitively.rep.usp_Sales
  @CustomerID = @CustomerID
  , @ItemID = @ItemID
  , @Grain = @Grain
  , @DateStart = @DateStart
  , @DateEnd = @DateEnd
  , @ReturnParameter_OUT = @Out_ReturnParameter OUT;

INSERT LearnSQLServerIntuitively.dbo.StoredProceduresLogs
  (
    DateTimeExecuted
    , Details
  )
SELECT
  GETDATE()
  , @Out_ReturnParameter;
END
GO
```

Step 1

Step 2

Bypass the parameters from the outer SP to the inner SP

Step 3

EXECute the outer SP:

```sql
EXEC LearnSQLServerIntuitively.rep.usp_SalesAndLog
  @CustomerID = 4
  , @Grain = 'S'
GO
```

CustomerName	SalesValue
John Smith	24.88

Verify the handling of the OUTPUT parameter:

```sql
SELECT *
FROM LearnSQLServerIntuitively.dbo.StoredProceduresLogs;
GO
```

LogID	DateTimeExecuted	Details
1	1968-05-17 11:15:59.973	Grain: D; Rows: 2
2	1968-06-05 23:48:05.633	Grain: S; Rows: 1
3	1969-05-17 11:25:34.630	Grain: S; Rows: 1

INSERT… EXEC

We can insert a recordset, created by a SP into a table, by using the INSERT… EXEC statement.
Collect the results of the EXECutions of **usp_SalesAndLog** into table **SPExecResults** in schema **rep**:

```
INSERT LearnSQLServerIntuitively.rep.SPExecResults (CustomerName, SalesValue)
EXEC LearnSQLServerIntuitively.rep.usp_SalesAndLog
  @CustomerID = 4
  , @Grain = 'S'
GO
```

EXECute again for @CustomerID = 2 and verify the content of table **SPExecResults**:

```
SELECT *
FROM LearnSQLServerIntuitively.rep.SPExecResults;
GO
```

CustomerName	SalesValue	DateTimeExecuted
John Smith	24.88	1967-05-17 11:43:45.060
Anna Laurier	147.36	1969-06-09 20:05:45.757

ALTER PROC

To edit a SP:

- Export the current SP definition: *(In SSMS Object Explorer)* ⟶ **Databases** ⟶ *(DatabaseName)* ⟶ **Programmability** ⟶ **Stored Procedures** ⟶ *(right click) (StoredProcedureName)* ⟶ **Script Stored Procedure as** ⟶ **ALTER To** ⟶ **File …**
- Edit the:
  - Parameters list (if we need to add an additional parameter or edit/delete an existing one)
  - The SP definition
- Execute the ALTER PROC statement (F5) to save the changes in the DBE

Before the ALTER or DROP SP, we save a backup, exported as explained above (…**CREATE To**…)

Add parameter @CustomerID to usp_Customers_Select:

```
ALTER PROC dbo.usp_Customers_Select (@CustomerID INT)
AS
BEGIN
  SET NOCOUNT ON;
```

Add parameter

```
  SELECT
    FirstName
    , LastName
    , Email
    , DateTimeRegistered
  FROM LearnSQLServerIntuitively.dbo.Customers
  WHERE CustomerID = @CustomerID;
END
GO
```

Use the parameter

DROP PROC

We delete the Stored Procedure with the `DROP PROC` DDL statement.

Before the `DROP PROC`, check the dependencies:

- DB object(s) that depend on the SP
- DB objects in which the SP depends

```
EXEC sp_depends 'LearnSQLServerIntuitively.rep.usp_Sales';
GO
```

Delete the SP:

```
USE LearnSQLServerIntuitively;
GO

DROP PROC IF EXISTS dbo.usp_Customers_Select;
GO
```

SP Limitations

- Number of parameters per SP: 2,100

Naming Conventions

`camelCase`, `PascalCase`, `[Special Characters]`, `alllowercase`, `ALLUPPERCASE` ← `Case`
`NoDelimiter`, `Delimiter_Underscore`, `[Delimiter Space]` ← `Delimiter`

Prefix

`t`, `t_`, `tbl`, `tbl_` - Table
`v`, `v_`, `vw`, `vw_` - View
`fn`, `fn_`, `ufn`, `ufn_`, `udf`, `udf_`, `usf`, `utf` - Function
`sp`, `usp_` (Do not use `sp_` - built in SP) - Stored Procedure

`__Current`, `__Archive`, `_AddRow`, `_Report`, `_ETL` ← `Sufix`
`DateInvoiced`, `InvoiceDate`, `@DateStart`, `@StartDate` ← `Words Order`

Hierarchy

`[ServerName].[DatabaseName].[SchemaName].[ObjectName]` ← `SDSO`
`[DatabaseName].[SchemaName].[ObjectName]` ← `DSO`
`[SchemaName].[ObjectName]` ← `SO`
`[ObjectName]` ← `O`

SSMS

Script Object

- In Object Explorer expand Databases
- Expand *`DatabaseName`*
- Expand Tables, Views or Programmability (Stored Procedures or Functions (Scalar-valued, Table-valued))
- Right click on the *`ObjectName`*
- Script (Table, View, Function, Stored Procedure) as
- Pick the type of the exported script (DDL or DML)
- New Query Window, File, Clipboard

Shortcuts

F5	Execute	F3	Find next
Alt + Break	Cancel running execution	Shift + F1 (mark *keyword*)	Search in Books Online
Ctrl + F5	Parse	Ctrl + Tab	Move to the next tab
Ctrl + R	Toggle Results and Messages	Ctrl + Shift + Tab (↑, ↓)	Jump to tab
F4	Properties	Ctrl + N	New query tab
Ctrl + M	Include actual execution plan	Ctro + O	Open file
Ctrl + L	Estimated execution plan	Ctrl + F4	Close the current tab
Alt + F1 (mark *`ObjectName`*)	`EXEC sp_help`	Alt + F4	Exit SSMS
Ctrl + F (Ctrl + H)	Find and Replace	Ctrl + S	Save

Ctrl + Shift + S	Save all	Shift + Delete	Delete entire row
Ctrl + X	Cut	Ctrl + Shift + U	Uppercase
Ctrl + C	Copy	Ctrl + Shift + L	Lowercase
Ctrl + V	Paste	Ctrl +Shift + D	Results to grid
Ctrl + Z	Undo	Ctrl + T	Results to text
Ctrl + Y	Redo	Ctrl + Shift + T	Results to file

Comments

```
-- Single line comment
...T-SQL... -- Comment for this line

/*
Multiple lines comment - row 1
Multiple lines comment - row 2
*/
```

Collations

Server Level

```
SELECT *
FROM sys.fn_helpcollations();

SELECT SERVERPROPERTY('Collation') AS [Server Collation];
```

DB level

```
CREATE DATABASE DatabaseName
  COLLATE SQL_Latin1_General_CP1_CI_AS;

SELECT name, collation_name
FROM sys.databases;

SELECT DATABASEPROPERTYEX('DatabaseName', 'Collation') AS [Database Collation];
```

Column level

```
SELECT [collation_name] AS ColumnCollation
FROM sys.columns
WHERE
  OBJECT_NAME([object_id]) = 'TableName'
  AND [name] = 'ColumnName';
```

Overwrite collation

```
SELECT ColumnName COLLATE SQL_Latin1_General_CP1_CI_AS
FROM TableName;

SELECT
  T1.ColumnName2
  , T2.ColumnName2
FROM
  DatabaseName1.SchemaName1.TableName1 AS T1
  JOIN DatabaseName2.SchemaName2.TableName2 AS T2
    ON T1.ColumnName1 COLLATE SQL_Latin1_General_CP1_CI_AS = T2.ColumnName1;
```

DML Query: Object Catalog

Objects

```
SELECT *
FROM DatabaseName.sys.objects;
```

All system and user objects

```
SELECT *
FROM DatabaseName.sys.all_objects;
```

Object Types

Tables	
IT	Internal table
S	System base table
U	Table (user-defined)
Constraints	
C	CHECK constraint
D	DEFAULT (constraint or stand-alone)
F	FOREIGN KEY constraint
PK	PRIMARY KEY constraint
UQ	UNIQUE constraint
Views	
V	View
Functions	
AF	Aggregate function (CLR)
FN	SQL scalar function
FS	Assembly (CLR) scalar-function
FT	Assembly (CLR) table-valued function
IF	SQL inline table-valued function
TF	SQL table-valued-function
Stored Procedures	
P	SQL Stored Procedure
PC	Assembly (CLR) stored-procedure
RF	Replication-filter-procedure
X	Extended stored procedure
Triggers	
TA	Assembly (CLR) DML trigger
TR	SQL DML trigger
Other	
PG	Plan guide
R	Rule (old-style, stand-alone)
SN	Synonym
SO	Sequence object
SQ	Service queue
TT	Table type

Object Definition (View, Function, Stored Procedure)

```
EXEC sp_help 'SchemaName.ObjectName';                                  ← T, V, F, SP
SELECT OBJECT_DEFINITION(OBJECT_ID('SchemaName.ObjectName'));          ← V, F, SP
EXEC sp_helptext 'SchemaName.ObjectName';                              ← V, F, SP
```

Server, Database Properties

```
SELECT SERVERPROPERTY('PropertyName');
```

BuildClrVersion, Collation, CollationID, ComparisonStyle, ComputerNamePhysicalNetBIOS, Edition, EditionID, EngineEdition, FilestreamConfiguredLevel, FilestreamEffectiveLevel, FilestreamShareName, HadrManagerStatus, InstanceDefaultDataPath, InstanceDefaultLogPath, InstanceName, IsAdvancedAnalyticsInstalled, IsClustered, IsFullTextInstalled, IsHadrEnabled, IsIntegratedSecurityOnly, IsLocalDB, IsPolybaseInstalled, IsSingleUser, IsXTPSupported, LCID, LicenseType, MachineName, NumLicenses, ProcessID, ProductBuild, ProductBuildType, ProductLevel, ProductMajorVersion, ProductMinorVersion, ProductUpdateLevel, ProductUpdateReference, ProductVersion, ResourceLastUpdateDateTime, ResourceVersion, ServerName, SqlCharSet, SqlCharSetName, SqlSortOrder, SqlSortOrderName

```
SELECT DATABASEPROPERTY('DatabaseName', 'PropertyName');
```

IsAnsiNullDefault, IsAnsiNullsEnabled, IsAnsiWarningsEnabled, IsAutoClose, IsAutoCreateStatistics, IsAutoShrink, IsAutoUpdateStatistics, IsBulkCopy, IsCloseCursorsOnCommitEnabled, IsDboOnly, IsDetached, IsEmergencyMode, IsFulltextEnabled, IsInLoad, IsInRecovery, IsInStandBy, IsLocalCursorsDefault, IsNotRecovered, IsNullConcat, IsOffline, IsParameterizationForced, IsQuotedIdentifiersEnabled, IsReadOnly, IsRecursiveTriggersEnabled, IsShutDown, IsSingleUser, IsSuspect, IsTruncLog, Version

```
SELECT DATABASEPROPERTYEX('DatabaseName', 'PropertyName');
```

Collation, ComparisonStyle, Edition, IsAnsiNullDefault, IsAnsiNullsEnabled, IsAnsiPaddingEnabled, IsAnsiWarningsEnabled, IsArithmeticAbortEnabled, IsAutoClose, IsAutoCreateStatistics, IsAutoCreateStatisticsIncremental, IsAutoShrink, IsAutoUpdateStatistics, IsCloseCursorsOnCommitEnabled, IsFulltextEnabled, IsInStandBy, IsLocalCursorsDefault, IsMemoryOptimizedElevateToSnapshotEnabled, IsMergePublished, IsNullConcat, IsNumericRoundAbortEnabled, IsParameterizationForced, IsPublished, IsQuotedIdentifiersEnabled, IsRecursiveTriggersEnabled, IsSubscribed, IsSyncWithBackup, IsTornPageDetectionEnabled, IsXTPSupported, LCID, MaxSizeInBytes, Recovery, ServiceObjective, ServiceObjectiveId, SQLSortOrder, Status, Updateability, UserAccess, Version

Databases

DML Query: Object Catalog

```
SELECT *
FROM sys.databases;
```

Databases

```
SELECT
  DB_NAME(database_id) AS [Database Name]
  , name AS [File Name]
  , physical_name AS [File Path]
  , (size * 8) / 1024 AS [Size (MB)]
FROM sys.master_files;

SELECT *
FROM DatabaseName.sys.database_files;
```

Database files

DDL: DROP DATABASE

```
USE [master];
IF (DB_ID('DatabaseName') IS NOT NULL)
  BEGIN
    ALTER DATABASE DatabaseName SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
    DROP DATABASE DatabaseName;                                          ← Before SQL Server 2016
  END

DROP DATABASE IF EXISTS DatabaseName;

EXEC msdb.dbo.sp_delete_database_backuphistory @database_name = N'DatabaseName';
                                                                         ← Delete the backup history
```

DDL: CREATE DATABASE

```
CREATE DATABASE DatabaseName;
```

DDL: ALTER DATABASE

```
ALTER DATABASE DatabaseName
MODIFY Name = DatabaseNameNew;
```

Schemas

DML Query: Object Catalog

```
SELECT *
FROM DatabaseName.sys.schemas;
```

DDL: DROP SCHEMA

```
IF (SCHEMA_ID('SchemaName') IS NOT NULL)
  BEGIN DROP SCHEMA SchemaName; END                                      ← Before SQL Server 2016

DROP SCHEMA IF EXISTS SchemaName;
```

DDL: CREATE SCHEMA

```
CREATE SCHEMA SchemaName;
```

DDL: ALTER SCHEMA

```
ALTER SCHEMA DestinationSchemaName
TRANSFER SourceSchemaName.ObjectToTransferName;
```

Tables

DML Query: Object Catalog

Tables

```
SELECT *
FROM DatabaseName.sys.tables;

SELECT *
FROM DatabaseName.sys.objects
WHERE [type] IN ('IT', 'S', 'U');
```

Change ***DatabaseName*** to **tempdb** for temp tables

Tables, Columns, Data Types

```
SELECT
  T.*
  , '<-- Table | Column -->' AS [<-- Table | Column -->]
  , C.*
  , '<-- Column | Type -->' AS [<-- Column | Type -->]
  , Ty.*
FROM
  DatabaseName.sys.tables AS T
  JOIN DatabaseName.sys.columns AS C
    ON T.[object_id] = C.[object_id]
  JOIN DatabaseName.sys.types AS Ty
    ON C.user_type_id = Ty.user_type_id;
```

DDL: DROP TABLE (Real table)

```
IF (OBJECT_ID('DatabaseName.SchemaName.TableName', 'U') IS NOT NULL)
  BEGIN DROP TABLE DatabaseName.SchemaName.TableName; END

DROP TABLE IF EXISTS DatabaseName.SchemaName.TableName;
```

Before SQL Server 2016

DDL: CREATE TABLE (Real table)

```
CREATE TABLE DatabaseName.SchemaName.TableName
  (
    ColumnName1 INT NOT NULL
    , ColumnName2 DATETIME
    , CONSTRAINT PK__Customers PRIMARY KEY (ColumnName1)
  );
```

NOT NULL constraint, created after the column definition

Create constraint(s) after the column list

DDL: DROP TABLE (Temp table)

```
IF (OBJECT_ID('tempdb.dbo.#TempTableName', 'U') IS NOT NULL)
  BEGIN DROP TABLE #TempTableName; END

DROP TABLE IF EXISTS #TempTableName;
```

Before SQL Server 2016

- Local; ## - Global

DDL: CREATE TABLE (Temp table)

```
CREATE TABLE #TempTableName
  (
    ColumnName1 INT
    , ColumnName2 DATETIME
  );
```

DDL: Create CONSTRAINT(s) in CREATE TABLE (after the column list)

PRIMARY KEY (PK)

```
, CONSTRAINT PK__TableName -- Consraint name
  PRIMARY KEY (ColumnName)
, CONSTRAINT PK__TableName_ColumnName1_ColumnName2
  PRIMARY KEY (ColumnName1, ColumnName2) -- Composite PK
```

FOREIGN KEY (PK)

```
, CONSTRAINT FK__PKTableName_FKColumnName
  FOREIGN KEY (FKColumnName) REFERENCES PKTableName (PKColumnName)
    ON DELETE CASCADE
    ON UPDATE CASCADE
```

NO ACTION (default), CASCADE, SET NULL, SET DEFAULT

DDL: Create CONSTRAINT(s) in CREATE TABLE (after the column definition)

DEFAULT (DF)

```
CONSTRAINT DF__TableName_ColumnName__DFDescription
  DEFAULT (GETDATE())
```

CHECK (CK)

```
CONSTRAINT CK__TableName_ColumnName__CKDescription
  CHECK (ColumnName > 0)
```

UNIQUE (UQ)

```
CONSTRAINT UQ__TableName_ColumnName
  UNIQUE
```

NOT NULL (NN)

```
NOT NULL
```

DDL: ALTER TABLE

Columns

ADD – add columns(s)

```
ALTER TABLE DatabaseName.SchemaName.TableName
ADD
```

```
  ColumnName1 INT
  , ColumnName2 MONEY;
```

ALTER COLUMN – change the column definition

```
ALTER TABLE DatabaseName.SchemaName.TableName
ALTER COLUMN ColumnName MONEY NOT NULL;
```

New data type, NN constraint

DROP COLUMN – delete column(s)

```
ALTER TABLE DatabaseName.SchemaName.TableName
DROP COLUMN
  ColumnName1
  , ColumnName2;
```

Before SQL Server 2016

```
ALTER TABLE DatabaseName.SchemaName.TableName
DROP COLUMN IF EXISTS ColumnName;
```

Constraints

ADD CONSTRAINT – add new constraint

```
ALTER TABLE DatabaseName.SchemaName.TableName
ADD CONSTRAINT DF__TableName_ColumnName__DFDescription
  DEFAULT (GETDATE())
  FOR ColumnName;
```

DROP CONSTRAINT – delete existing constraint

```
ALTER TABLE DatabaseName.SchemaName.TableName
DROP ConstraintName;
```

Before SQL Server 2016

```
ALTER TABLE DatabaseName.SchemaName.TableName
DROP CONSTRAINT IF EXISTS ConstraintName;
```

DDL, DML Modify: TRUNCATE TABLE

```
TRUNCATE TABLE DatabaseName.SchemaName.TableName;
```

DML Modify: INSERT

INSERT… SELECT

```
INSERT DatabaseName1.SchemaName1.TableName1
  (
    ColumnName1
    , ColumnName2
  )
```

```
SELECT
  ColumnName1
  , ColumnName2
FROM DatabaseName2.SchemaName2.TableName2;
```

INSERT… UNION

```
INSERT DatabaseName.SchemaName.TableName
  (
    ColumnName1
    , ColumnName2
  )
SELECT 1, 'Value1'
UNION ALL SELECT 2, 'Value2';
```

INSERT… VALUES

```
INSERT DatabaseName.SchemaName.TableName
  (
    ColumnName1
    , ColumnName2
  )
VALUES
  (1, 'Value1')
  , (2, 'Value2');
```

DML Modify: UPDATE

```
UPDATE DatabaseName.SchemaName.TableName
SET ColumnName1 = NewValue
WHERE ColumnName1 = FilterValue

UPDATE DatabaseName1.SchemaName1.TableName1
SET
  ColumnName3 = A.Sum_Col3
  , ColumnName4 = A.Avg_Col4
FROM
  (
    SELECT
      ColumnName1 AS Col1
      , ColumnName2 AS Col2
      , SUM(ColumnName3) AS Sum_Col3
      , AVG(ColumnName4) AS Avg_Col4
    FROM DatabaseName2.SchemaName2.TableName2
```

```
    GROUP BY
      ColumnName1
      , ColumnName2
  ) AS A
WHERE
  ColumnName1 = A.Col1
  AND ColumnName2 = A.Col2;
```

DML Modify: DELETE

```
DELETE DatabaseName.SchemaName.TableName
WHERE ColumnName = FilterValue;

DELETE T1
FROM
  DatabaseName1.SchemaName1.TableName1 AS T1
  JOIN DatabaseName2.SchemaName2.TableName2 AS T2
    ON T1.ColumnName1 = T2.ColumnName1;
WHERE T2.ColumnName2 = FilterValue;
```

Views

DML Query: Object Catalog

```
SELECT *
FROM DatabaseName.sys.views;

SELECT *
FROM DatabaseName.sys.objects
WHERE [type] IN ('V');
```

DDL: DROP VIEW

```
IF (OBJECT_ID('SchemaName.vw_ViewName', 'V') IS NOT NULL)
  BEGIN DROP VIEW SchemaName.vw_ViewName; END          ← Before SQL Server 2016

DROP VIEW IF EXISTS SchemaName.vw_ViewName;
```

DDL: CREATE VIEW

```
CREATE VIEW SchemaName.vw_ViewName                     ← SO naming convention
AS
  SELECT
    ColumnName1
```

```
  , ColumnName2
FROM DatabaseName.SchemaName.TableName;
```

No ORDER BY in Views

DDL: ALTER VIEW

1. Script the view in SSMS
2. Edit the definition
3. Change the keyword CREATE to ALTER
4. Execute the edited code (F5)

DML Query: SELECT

```
SELECT *
FROM DatabaseName.SchemaName.ViewName;
```

DML Query: Tables, Views, Table-valued functions

SELECT

```
SELECT 123;
SELECT (123 + 456);
SELECT '123 ' + 456;
SELECT '123 ' + CAST(456 AS VARCHAR);
```

FROM

```
SELECT
  ColumnName1
  , ColumnName2
FROM DatabaseName.SchemaName.TableName;
```

JOIN

```
SELECT
  T1.ColumnName2
  , T2.ColumnName2
FROM
  DatabaseName1.SchemaName1.TableName1 AS T1
  JOIN DatabaseName2.SchemaName2.TableName2 AS T2 -- JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN
    ON T1.ColumnName1 = T2.ColumnName1;
```

WHERE

```sql
SELECT 'A'
WHERE @VariableName = FilterValue;

SELECT
  T1.ColumnName2
  , T2.ColumnName2
FROM
  DatabaseName1.SchemaName1.TableName1 AS T1
  JOIN DatabaseName2.SchemaName2.TableName2 AS T2
    ON T1.ColumnName1 = T2.ColumnName1;
WHERE
  T1.ColumnName3 = FilterValue1
  AND T2.ColumnName4 <= FilterValue2
  AND
  (                                                    ← Separate OR from AND in brackets
    T1.ColumnName5 > (@ParameterValue1 + @ParameterValue2)
    OR T1.ColumnName6 BETWEEN FilterValue4 AND FilterValue5
  );

GROUP BY
SELECT
  ColumnName1
  , ColumnName2
  , SUM(ColumnName3) AS Sum_ColumnName3
  , MIN(ColumnName4) AS Min_ColumnName4
  , MAX(ColumnName5) AS Max_ColumnName5
  , AVG(ColumnName6) AS Avg_ColumnName6
  , COUNT(ColumnName7) AS Count_ColumnName7
  , COUNT(DISTINCT ColumnName8) AS CountDistinct_ColumnName8
FROM DatabaseName.SchemaName.TableName
GROUP BY
  ColumnName1
  , ColumnName2;

GROUPING SETS
…
GROUP BY GROUPING SETS
  (
    (ColumnName1, ColumnName2)                         ← Group 1
    , (ColumnName3, ColumnName4, ColumnName5)          ← Group 2
    , ()                                               ← Grand Total
```

```
  )

ROLLUP
…
GROUP BY ROLLUP
  (
    ColumnName1
    , ColumnName2
  )

CUBE
…
GROUP BY CUBE
  (
    ColumnName1
    , ColumnName2
  )

HAVING
SELECT
  ColumnName1
  , ColumnName2
  , MIN(ColumnName3) AS Min_ColumnName3
FROM DatabaseName.SchemaName.TableName
GROUP BY
  ColumnName1
  , ColumnName2
HAVING SUM(ColumnName4) >= FilterValue;

ORDER BY
SELECT
  ColumnName1
  , ColumnName2 AS Col2
  , ColumnName3
FROM DatabaseName.SchemaName.TableName;
ORDER BY
  ColumnName1                    ← Column name
  , Col2 DESC                    ← Alias. Descending
  , 3;                           ← Column index

TOP
```

```sql
SELECT TOP (10) *
FROM DatabaseName.SchemaName.TableName;
ORDER BY ColumnName;
```

Rows

```sql
SELECT TOP (10) WITH TIES
  ColumnName1
  , ColumnName2
FROM DatabaseName.SchemaName.TableName;
ORDER BY ColumnName3;
```

Rows

```sql
SELECT TOP 10 PERCENT *
FROM DatabaseName.SchemaName.TableName;
ORDER BY ColumnName DESC;
```

Statement Execution Order

Order of the clauses in statement:

1. `SELECT`
2. `FROM... JOIN`
3. `WHERE`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

Order of execution:

Step 1: `FROM... JOIN` - build the virtual recordset (VR) from one or multiple data sources (DS)
Step 2: `WHERE` - filter the VR
Step 3: `GROUP BY... HAVING` - group, aggregate and filter the grouped VR
Step 4: `SELECT` – builds the resulting VR and create aliases
Step 5: `ORDER BY` – order the resulting VR and use the aliases created in the `SELECT` statement

Subqueries

Non-Correlated Subquery

```sql
SELECT
  ColumnName1
  , (
    SELECT MIN(ColumnName2)
    FROM DatabaseName1.SchemaName1.TableName1
  ) AS ColumnName2
FROM DatabaseName2.SchemaName2.TableName2;
```

Single value

In the SELECT clause

```
SELECT
  T1.ColumnName2
  , T2.Sum_ColumnName2
FROM
  DatabaseName1.SchemaName1.TableName1 AS T1
  JOIN
  (
    SELECT DISTINCT
      ColumnName1
      , SUM(T2.ColumnName2) AS Sum_ColumnName2
    FROM DatabaseName2.SchemaName2.TableName2
  ) AS T2                                                    ← In the FROM clause
    ON T1.ColumnName1 = T2.ColumnName1;

SELECT
  ColumnName1
  , ColumnName2
FROM DatabaseName1.SchemaName1.TableName1
WHERE
  ColumnName3 IN                                             ← In the WHERE clause (IN)
  (
    SELECT TOP 10 ColumnName1
    FROM DatabaseName2.SchemaName2.TableName2
    ORDER BY ColumnName2 DESC
  );

SELECT
  ColumnName1
  , ColumnName2
FROM DatabaseName1.SchemaName1.TableName1
WHERE
  EXISTS                                                     ← In the WHERE clause (EXISTS)
  (
    SELECT 1                                                 ← Not needed to select data
    FROM DatabaseName2.SchemaName2.TableName2
    WHERE ColumnName1 = @VariableValue
  );
```

Correlated Subquery

```
SELECT
  T2.ColumnName2
```

```sql
  , (
    SELECT MIN(T1.ColumnName2)
    FROM DatabaseName1.SchemaName1.TableName1 AS T1
    WHERE T1.Column1 = T2.Column1
  ) AS Min_ColumnName2                                    ← In the SELECT clause
FROM DatabaseName2.SchemaName2.TableName2 AS T2;

SELECT
  T1.ColumnName2
  , T1.ColumnName3
FROM DatabaseName1.SchemaName1.TableName1 AS T1
WHERE
  T1.ColumnName3 IN                                       ← In the WHERE clause (IN)
  (
    SELECT T2.ColumnName1
    FROM DatabaseName2.SchemaName2.TableName2 AS T2
    WHERE T2.ColumnName1 = T1.ColumnName1                 ← Correlation
    GROUP BY T2.ColumnName1
    HAVING SUM(T2.ColumnName2) >= FilterValue
  );

SELECT
  T1.ColumnName2
  , T1.ColumnName3
FROM DatabaseName1.SchemaName1.TableName1 AS T1
WHERE
  EXISTS                                                  ← In the WHERE clause (EXISTS)
  (
    SELECT 1
    FROM DatabaseName1.SchemaName1.TableName1 AS T2
    WHERE T2.ColumnName1 = T1.ColumnName1                 ← Correlation
    GROUP BY T2.ColumnName1
    HAVING SUM(T2.ColumnName2) >= FilterValue
  );
```

Functions

Object Catalog

```sql
SELECT *
FROM DatabaseName.sys.objects
WHERE [type] IN ('AF', 'FN', 'FS', 'FT', 'IF', 'TF');
```

Scalar-valued

DDL: DROP FUNCTION

```
IF (OBJECT_ID('SchemaName.udf_FunctionName', 'FN') IS NOT NULL)
  BEGIN DROP FUNCTION SchemaName.udf_FunctionName; END
```

Before SQL Server 2016

```
DROP FUNCTION IF EXISTS SchemaName.udf_FunctionName;
```

DDL: CREATE FUNCTION

```
CREATE FUNCTION SchemaName.udf_FunctionName
  (
    @ParameterName1 INT
    , @ParameterName2 MONEY = NULL
  )
RETURNS INT
AS
BEGIN
  IF (@ParameterName2 IS NULL)
    BEGIN
      {Statments to handle the DEFAULT value}
      ...Statement 1;...
      ...Statement 2;...
    END

  DECLARE @ReturnVariableName INT;
  SELECT @ReturnVariableName = {Statement(s) to populate @ReturnVariableName};
  RETURN @ReturnVariableName;
END
```

SO naming convention

DEFAULT

DML Query: Call Scalar-valued function

```
SELECT DatabaseName.SchemaName.udf_FunctionName(ParameterName1Value, DEFAULT);
```

In the SELECT clause, DEFAULT parameter

```
SELECT *
FROM
  DatabaseName1.SchemaName1.TableName1 AS T1
  JOIN DatabaseName2.SchemaName2.TableName2 AS T2
    ON DatabaseName3.SchemaName3.udf_FunctionName(T1.ColumnName1) = T2.ColumnName1;
```

In the FROM clause

```
SELECT *
FROM DatabaseName.SchemaName.TableName
WHERE ColumnName = DatabaseName.SchemaName.udf_FunctionName(@ParameterName1Value, DEFAULT);
```

In the WHERE clause

Table-valued (Inline)

DDL: DROP FUNCTION

```
IF (OBJECT_ID('SchemaName.udf_FunctionName', 'IF') IS NOT NULL)
  BEGIN DROP FUNCTION SchemaName.udf_FunctionName; END          ← Before SQL Server 2016

DROP FUNCTION IF EXISTS SchemaName.udf_FunctionName;
```

DDL: CREATE FUNCTION

```
CREATE FUNCTION SchemaName.udf_FunctionName                     ← SO naming convention
  (
    ParameterName1 INT
    , @ParameterName2 MONEY = NULL                              ← DEFAULT
  )
RETURNS TABLE
AS
RETURN
(
  ...Statement 1;...                                            ← Single statement that returns a recordset
);
```

Table-valued (Multi-statement)

DDL: DROP FUNCTION

```
IF (OBJECT_ID('SchemaName.udf_FunctionName', 'TF') IS NOT NULL)
  BEGIN DROP FUNCTION SchemaName.udf_FunctionName; END          ← Before SQL Server 2016

DROP FUNCTION IF EXISTS SchemaName.udf_FunctionName;
```

DDL: CREATE FUNCTION

```
CREATE FUNCTION SchemaName.udf_FunctionName
  (
    @ParameterName1 INT
    , @ParameterName2 MONEY = NULL                              ← DEFAULT
  )
RETURNS @ReturnTableName TABLE
  (
    ColumnName1 INT
    , ColumnName2 MONEY
  )
AS
BEGIN
  ...Statement 1....                                            ← Multiple statements to populate @ReturnTableName
```

```
  ...Statement 2;...

  INSERT @ReturnTableName                                    ← Statement 3
  SELECT
    ColumnName1
    , ColumnName2
  FROM DatabaseName.SchemaName.ObjectName;

  RETURN;
END
```

DML Query: Call Table-valued (Inline, Multi-statement) function

```
SELECT *
FROM DatabaseName.SchemaName.udf_FunctionName;               ← In the FROM clause
```

DDL: `ALTER FUNCTION`

1. Script the function in SSMS
2. Edit the parameters and/or the definition
3. Change the keyword `CREATE` with `ALTER`
4. Execute the edited code (F5)

Stored Procedures

Object Catalog

```
SELECT *
FROM DatabaseName.sys.procedures;

SELECT *
FROM DatabaseName.sys.objects
WHERE type IN ('P', 'PC', 'RF', 'X');
```

DDL: `DROP PROC`

```
IF (OBJECT_ID('SchemaName.usp_StoredProcedureName', 'P') IS NOT NULL)
  BEGIN DROP PROC SchemaName.usp_StoredProcedureName; END    ← Before SQL Server 2016

DROP PROCEDURE IF EXISTS SchemaName.usp_StoredProcedureName;
```

DDL: `CREATE PROC`

```
CREATE PROC SchemaName.usp_StoredProcedureName               ← SO naming convention
  (
    @ParameterName1 INT
    , @ParameterName2 DATE = NULL                            ← DEFAULT
```

```
  )
AS
BEGIN
  IF (@ParameterName2 IS NULL)
    BEGIN
      ...Statement(s) to sets DEFAULT value to @ParameterName2...;
    END

  ...Statement 2...;
  ...Statement 3...;
END
```

DDL: ALTER PROC

1. Script the stored procedure in SSMS
2. Edit the parameters and/or the definition
3. Change the keyword CREATE to ALTER
4. Execute the edited code (F5)

DML Query: EXEC PROC

```
EXEC PROC DatabaseName.SchemaName.usp_StoredProcedureName
  @ParameterName1 = 123
  , @ParameterName3 OUT = 'ABC';
```

@ParameterName2 has default value and can be omitted

OUTPUT patameter

```
DECLARE @ParameterName1_Out INT;

EXEC PROC DatabaseName.SchemaName.usp_StoredProcedureName
  @ParameterName1 = 123
  , @ParameterName2
  , @ParameterName3 = @ParameterName1_Out OUT;

INSERT DatabaseName.SchemaName.TableName
SELECT @ParameterName1_Out;
```

Step 1: Create a variable to store the value returned by the OUT parameter

Step 2: EXECute and populate the variable

Step 3: Manipulate the value returned by the OUT parameter

Conditional Execution

IF

```
IF (5 > 17)
  BEGIN PRINT 'The condition is True'; END
ELSE
  BEGIN PRINT 'The condition is False'; END
```

Condition (False)

Nested IF

```
IF (5 < 17)                                                    ← Condition 1 (True)
  BEGIN
    IF (6 = 7)                                                 ← Condition 2 (False)
      BEGIN PRINT 'Condition 1 is True, Condition 2 is True'; END
    ELSE
      BEGIN PRINT 'Condition 1 is True, Condition 2 is False'; END
  END
ELSE
  BEGIN PRINT 'Condition 1 is False'; END
```

IIF

```
SELECT IIF(ColumnName1 > 5, 'Greater than 5', 'Not greater than five') AS IIF_Result
FROM DatabaseName.SchemaName.TableName;
```

CASE

```
SELECT
  ColumnName1
  , ColumnName2
  , CASE ColumnName2
    WHEN 5 THEN 'Five'
    WHEN 6 THEN 'Six'
    ELSE 'Not 5 and 6'
  END AS Case_Result
FROM DatabaseName.SchemaName.TableName;

SELECT
  ColumnName1
  , ColumnName2
  , CASE
    WHEN (ColumnName2 BETWEEN 5 AND 6 AND ColumnName1 = 52) THEN 'BETWEEN 5 AND 6 (52)'
    WHEN ColumnName2 BETWEEN 5 AND 6 THEN 'BETWEEN 5 AND 6'
    ELSE 'Ignored'
  END AS Case_Result
FROM DatabaseName.SchemaName.TableName;
```

Common Table Expressions (CTE)

```
WITH CTE_Name (Col1, Col2) AS
  (
    SELECT
```

```
    ColumnName1
    , ColumnName2
  FROM DatabaseName1.SchemaName1.TableName1
)
SELECT
  C.Col1
  , C.Col2
  , T1.ColumnName2
FROM
  CTE AS C
  JOIN DatabaseName2.SchemaName2.TableName2 AS T1
    ON C.Col1 = T1.ColumnName1;
```

Non-recursive, multiple CTEs

```
WITH
  CTE_Name1 (Col1, Col2, Col3) AS
  (
    SELECT
      ColumnName1
      , ColumnName2
      , ColumnName3
    FROM DatabaseName1.SchemaName1.TableName1
  )
  , CTE_Name2 AS
  (
    SELECT
      ColumnName1
      , ColumnName2
    FROM DatabaseName2.SchemaName2.TableName2
  )
SELECT
  C1.Col2 AS ColumnAlias1
  , C2.Col2 AS ColumnAlias2
FROM
  CTE_Name1 AS C1
  JOIN CTE_Name2 AS C2
    ON C1.Col1 = C2.ColumnName1
WHERE C1.Col3 >= FilterValue;
```

INSERT, SELECT, UPDATE, DELETE

Recursive CTE (Parent to child)

```
WITH CTE_Recursive AS
  (
    SELECT                                                          ← Anchor statement
      0 AS [Level]
      , ParentColumnName
      , ChildColumnName
    FROM DatabaseName.SchemaName.TableName
    WHERE ParentColumnName IS NULL                                  ← NULL is the top level

    UNION ALL                   ← UNION ALL combines the anchor and the recursive statement(s)

    SELECT                                                          ← Recursive statement
      (C.[Level] + 1) AS [Level]
      , P.ParentColumnName
      , P.ChildColumnName
    FROM DatabaseName.SchemaName.TableName AS P
      JOIN CTE_Recursive AS C                                       ← This CTE
        ON P.ParentColumnName = C.ChildColumnName                   ← JOIN parent to child
  )
SELECT *
FROM CTE_Recursive
ORDER BY
  [Level]
  , ParentColumnName
OPTION (MAXRECURSION 3);                                            ← Limit the recursions
```

Recursive CTE (Child to parent)

```
WITH CTE_Recursive AS
  (
    SELECT
      0 AS [Level]
      , ChildColumnName
      , ParentColumnName
    FROM DatabaseName.SchemaName.TableName
    WHERE ChildColumnName = FilterValue

    UNION ALL

    SELECT
      (C.[Level] + 1) AS [Level]
```

```
    , P.ChildColumnName
    , P.ParentColumnName
  FROM DatabaseName.SchemaName.TableName AS P
    JOIN CTE_Recursive AS C
      ON C.ParentColumnName = P.ChildColumnName
 )
SELECT *
FROM CTE_Recursive;
```

The statement before `WITH` has to end with semicolon (`;`)

Variables

```
DECLARE @VariableName1 INT = 123;                    ← Create and assign a value to the variable
SELECT *
FROM DatabaseName.SchemaName.TableName
WHERE ColumnName = @VariableName1;                   ← Use the variable

DECLARE @VariableName2 INT                           ← Create variable
SELECT @VariableName2 = MAX(ColumnName)              ← Assign a value to the variable
FROM DatabaseName.SchemaName.TableName;
SELECT @VariableName2;                               ← Use the variable
```

Table variable

```
DECLARE @TableVariable TABLE                         ← Create table variable
      (
            ColumnName1 INT
            , ColumnName2 DATETIME2
      );

INSERT @TableVariable (ColumnName1, ColumnName2)     ← Assign a value to the table variable
SELECT ColumnName1, ColumnName2
FROM DatabaseName1.SchemaName1.TableName1
WHERE ColumnName3 >= FilterValue;

SELECT *
FROM                                                 ← Use the table variable
      DatabaseName2.SchemaName2.TableName2 AS T1
      JOIN @TableVariable AS TV
            ON T1.ColumnName1 = TV.ColumnName1
            AND T1.ColumnName2 = TV.ColumnName2;
```

Loops (WHILE)

Count to 5

```
DECLARE @Counter INT = 1
WHILE (@Counter <= 5)
  BEGIN
    PRINT @Counter;
    SET @Counter += 1;
  END
```

BREAK, CONTINUE

```
DECLARE @Counter INT = 1
WHILE (@Counter <= 10)
  BEGIN
    PRINT @Counter;
    SET @Counter += 1;

    IF (@Counter <= 6)
      BEGIN CONTINUE; END      ← Go to the statement after BEGIN
    ELSE
      BEGIN BREAK; END         ← Go to the statement after END
  END

PRINT 'Finished!';
```

Data Types

Numerics

- **Exact**
  - TINYINT, SMALLINT, INT, BIGINT
  - NUMERIC, DECIMAL (DEC)
  - MONEY, SMALLMONEY
  - BIT
- **Approximate**
  - FLOAT, REAL

Strings

- **Character Non-Unicode**
  - CHAR
  - VARCHAR
  - ~~TEXT~~ - **Obsolete**. Replaced with VARCHAR(MAX)
- **Character Unicode**
  - NCHAR
  - NVARCHAR
  - ~~NTEXT~~ - **Obsolete**. Replaced with NVARCHAR(-MAX)

Binary

- BINARY
- VARBINARY
- ~~IMAGE~~ - **Obsolete**. Replaced with BINARY(MAX)

Date and time

- SMALLDATETIME, DATETIME, DATETIME2
- DATE
- TIME
- DATETIMEOFFSET

Other

- UNIQUEIDENTIFIER
- ~~TIMESTAMP~~ - **Obsolete**. Replaced with ROWVERSION.
- XML
- SQL_VARIANT
- TABLE
- HIYERERARCHYID
- CURSOR
- **Spacial**
  - GEOGRAPHY
  - GEOMETRY

Conversions

Convert one data type to another

```
SELECT
  ColumnName
  , CAST(ColumnName AS VARCHAR)
FROM DatabaseName.SchemaName.TableName;
```

numeric to character

```
SELECT
  ColumnName
  , CONVERT(DATETIME ColumnName)
FROM DatabaseName.SchemaName.TableName;

SELECT PARSE('10/02/1968' AS DATETIME USING 'en-US') AS PARSE_enUS;
SELECT PARSE('10/02/1968' AS DATETIME USING 'ru-RU') AS PARSE_ruRU;
```

TRY_ converts from one data type to another and return NULL instead of an error message

```
SELECT TRY_CAST('ABC' AS INT);
SELECT TRY_CONVERT(INT, 'ABC');
SELECT TRY_PARSE('10/48/1968' AS DATETIME USING 'en-US') AS [TRY_PARSE];

SELECT STR(123.789);
```

Operators

Arithmetic Operators

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `%` Modulo

Compound Operators

- `+=` Add the value of the right operand to the left operand
- `-=` Subtracts the value of the right operand from the left operand
- `*=` Multiplies the operands and sets the result to the left operand
- `/=` Divides the operands and sets the result to the left operand
- `%=` Sets the result of the modulo to the left operand

Assignment Operator

Equal sign (=)

Comparison Operators

- `=` Equals to

`<>` or `!=`	Not equal to *
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`!<`	Not less than *
`!>`	Not greater than *

* !	not ISO standard

Logical Operators

`AND`	True if both operands return True
`OR`	True if one of the operands return True
`IN`	True if the operand is in the list (the other operand)
`LIKE`	True if the operand matches a pattern (the other operand)
`BETWEEN`	True if the operand is in the `BETWEEN` range
`EXISTS`	True if the query returns one or more rows
`NOT`	Reverse the meaning of the operator

Set Operators

`UNION` - combines two recordsets into one

```
SELECT ColumnName1, ColumnName2, ColumnName3
FROM DatabaseName1.SchemaName1.TableName1
UNION ALL -- or UNION to return duplicate rows
SELECT ColumnName1, ColumnName2, ColumnName3
FROM DatabaseName2.SchemaName2.TableName2
```

`EXCEPT` - remove the matching rows from the top recordset

```
SELECT ColumnName1, ColumnName2, ColumnName3
FROM DatabaseName1.SchemaName1.TableName1
EXCEPT
SELECT ColumnName1, ColumnName2, ColumnName3
FROM DatabaseName2.SchemaName2.TableName2
```

`INTERSECT` - return only the matching rows in the both recordsets

```
SELECT ColumnName1, ColumnName2, ColumnName3
FROM DatabaseName1.SchemaName1.TableName1
INTERSECT
SELECT ColumnName1, ColumnName2, ColumnName3
FROM DatabaseName2.SchemaName2.TableName2
```

String Concatenation Operator

+	Concatenate the left and the right operands
+=	Appends value to the left operand

Wildcards in LIKE operator

%	Any character(s)
[ ]	Range of matching character(s)
[^]	Range of not matching character(s)
_	Any matching single character

NULL and 3VL (Three-valued Logic)

```
SELECT ISNULL(ColumnName, ValueToReplaceNULL)
FROM DatabaseName.SchemaName.TableName;

SELECT NULLIF(ColumnName, ValueToBeNULL)
FROM DatabaseName.SchemaName.TableName;

SELECT COALESCE(ColumnName1, ColumnName2, ColumnName3)
FROM DatabaseName.SchemaName.TableName;
```

3VL from

```
...
WHERE ColumnName1 NOT LIKE '%StringToSearch%';
```

3VL to

```
...
WHERE
  (
    ColumnName1 NOT LIKE '%StringToSearch%'
    OR ColumnName1 IS NULL
  );
```

Pivoting

	Columns
Rows	Values

```
PIVOT
SELECT
  P.ColumnNameRows1 AS Row1, P.ColumnNameRows2 AS Row2
  , P.[ValueInColumns1] AS Col1, P.[ValueInColumns2] AS Col2
FROM
```

```
  (
    SELECT
      ColumnNameRows1, ColumnNameRows2
      , ColumnNameColumns
      , ColumnNameValues
    FROM DatabaseName.SchemaName.TableName
  ) AS S                                          ← Subquery for Rows, Columns and Values only
  PIVOT
  (
    SUM(ColumnNameValues)
    FOR ColumnNameColumns IN ([ValueInColumns1], [ValueInColumns2])
  ) AS P
ORDER BY P.ColumnNameRows1, P.ColumnNameRows2;

UNPIVOT
SELECT
  U.ColumnNameRows1, U.ColumnNameRows2
  , U.ColumnNameColumns
  , U.ColumnNameValues
FROM
  (
    SELECT
      ColumnNameRows1, ColumnNameRows2
      , ValueInColumns1, ValueInColumns2
    FROM DatabaseName.SchemaName.TableName
  ) AS P                                          ← Subquery to select only data to be UNPIVOTed
  UNPIVOT
  (
    ColumnNameValues
    FOR ColumnNameColumns IN ([ValueInColumns1], [ValueInColumns2])
  ) AS U
ORDER BY
  U.ColumnNameRows1, U.ColumnNameRows2
  , U.ColumnNameColumns;
```

Learn SQL Server Intuitively

Transact-SQL: The Solid Basics

Learn SQL Server Intuitively is a complete, hands-on and practical guide to everything you will ever need to know about the basics of Microsoft SQL Server relational database management systems.

From the basics, ideal for beginners with their own PC, through to more complex ideas for big business, **Learn SQL Server Intuitively** teaches you one step at a time, in an easy-to-follow and simple format, written in a language that you will understand.

The wide ranging benefits and features of this book include:

- A hands on approach
- Fast and easy
- Practical problem solving
- Saves you time and money
- Self-educating
- Intuitive and visual
- Tutorials and in-depth explanations
- And much more…

Learn SQL Server Intuitively is the only book you will ever need to help you navigate these complex processes. With its unique visual approach you will find it easier than ever to completely understand the database programming and how it can work better for you.

Get your copy of **Learn SQL Server Intuitively** today and get the most out of your home PC or business now.

Peter Lalovsky was born in Sofia, Bulgaria in 1976. In 2002 he obtained a Master's Degree in Economics and since then he has worked as a prepress designer and an IT specialist.

His interest in all things IT has led to Peter to become and Microsoft Certified Professional and prompted him to write his first book – **Learn SQL Server Intuitively** – which is a hands on and practical manual, in which he hopes to help others with his in-depth knowledge of the subject.

Nowadays Peter lives with his wife in Montreal, Canada, having moved there in 2009. In his spare time he plays bass guitar and has performed in several bands. He also enjoys photography, road biking and cooking and he also, quite unusually, has a taste for 18th and 19th century furniture.

Looking towards the future, Peter is hoping to have a home filled with children, perhaps moving to a small village where he can provide for his family and enjoy some peace and quiet away from city life.

ISBN 9780995245105

Source code online: LearnIntuitively.Lalovsky.com/SQLServer

Audience: Beginner/Intermediate