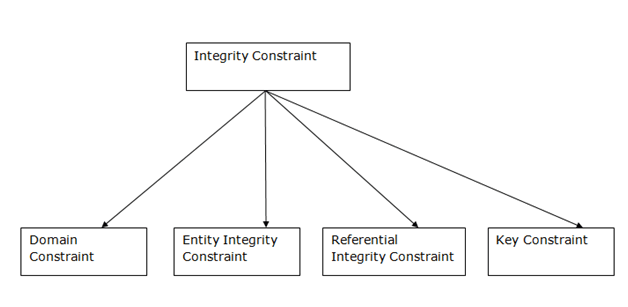
# Integrity Constraints

* Integrity constraints are a set of rules. It is used to maintain the quality of information.
* Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
* Thus, integrity constraint is used to guard against accidental damage to the database.

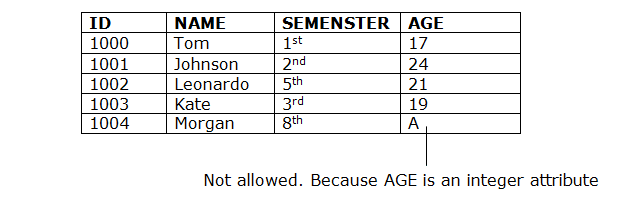
## Types of Integrity Constraint



### 1. Domain constraints

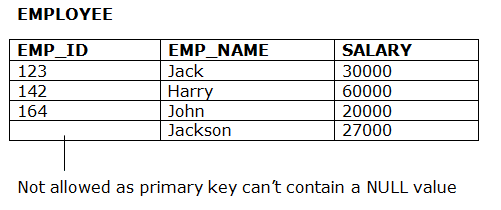
* Domain constraints can be defined as the definition of a valid set of values for an attribute.
* The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

**Example:**

  
**2. Entity integrity constraints**

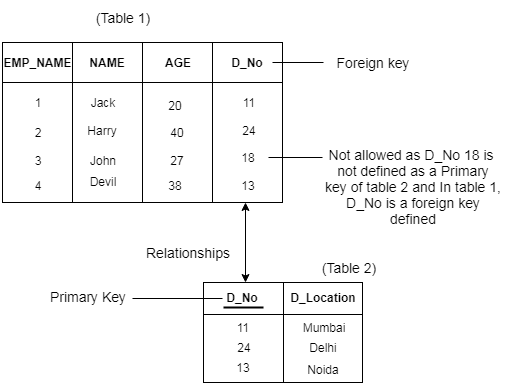
* The entity integrity constraint states that primary key value can't be null.
* This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
* A table can contain a null value other than the primary key field.

**Example:**



### 3. Referential Integrity Constraints

* A referential integrity constraint is specified between two tables.
* In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

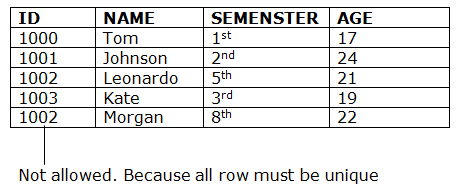
**Example:**  


### 4. Key constraints

* Keys are the entity set that is used to identify an entity within its entity set uniquely.
* An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

**Example:**

Keep Watching



# Constraints in SQL Server: SQL NOT NULL, UNIQUE and SQL PRIMARY KEY

Constraints in SQL Server are predefined rules and restrictions that are enforced in a single column or multiple columns, regarding the values allowed in the columns, to maintain the integrity, accuracy, and reliability of that column’s data. In other words, if the inserted data meets the constraint rule, it will be inserted successfully. If the inserted data violates the defined constraint, the insert operation will be aborted.

Constraints in SQL Server can be defined at the column level, where it is specified as part of the column definition and will be applied to that column only, or declared independently at the table level. In this case, the constraint rules will be applied to more than one column in the specified table. The constraint can be created within the CREATE TABLE T-SQL command while creating the table or added using ALTER TABLE T-SQL command after creating the table. Adding the constraint after creating the table, the existing data will be checked for the constraint rule before creating that constraint.

There are six main constraints that are commonly used in SQL Server that we will describe deeply with examples within this article and the next one. These constraints are:

* SQL NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK
* DEFAULT

In this article, we will go through the first three constraints; SQL NOT NULL, UNIQUE and SQL PRIMARY KEY, and we will complete the rest three constraints in the next article. Let us start discussing each one of these SQL Server constraints with a brief description and practical demo.

NOT NULL Constraint in SQL

By default, the columns are able to hold NULL values. A NOT NULL constraint in SQL is used to prevent inserting NULL values into the specified column, considering it as a not accepted value for that column. This means that you should provide a valid SQL NOT NULL value to that column in the INSERT or UPDATE statements, as the column will always contain data.

Assume that we have the below simple CREATE TABLE statement that is used to define the ConstraintDemo1 table. This table contains only two columns, ID and Name. In the ID column definition statement, the SQL NOT NULL column-level constraint is enforced, considering the ID column as a mandatory column that should be provided with a valid SQL NOT NULL value. The case is different for the Name column that can be ignored in the INSERT statement, with the ability to provide it with NULL value. If the null-ability is not specified while defining the column, it will accept the NULL value by default:

USE Test\_database

GO

CREATE TABLE ConstraintDemo1

(

ID INT NOT NULL,

Name VARCHAR(50) NULL

)

If we try to perform the below three insert operations:

INSERT INTO ConstraintDemo1 ([ID],[NAME]) VALUES (1,'Ali')

GO

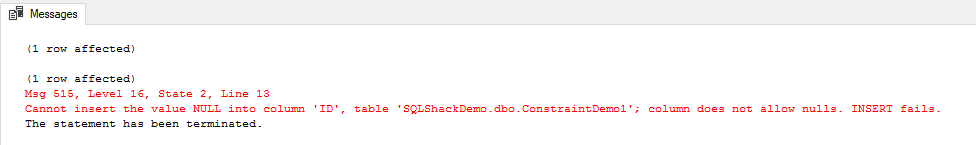
INSERT INTO ConstraintDemo1 ([ID]) VALUES (2)

GO

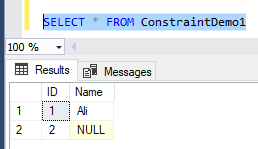
INSERT INTO ConstraintDemo1 ([NAME]) VALUES ('Fadi')

GO

You will see that the first record will be inserted successfully, as both the ID and Name column’s values are provided in the INSERT statement. Providing the ID only in the second INSERT statement will not prevent the insertion process from being completed successfully, due to the fact that the Name column is not mandatory and accepts NULL values. The last insert operation will fail, as we only provide the INSERT statement with a value for the Name column, without providing value for the ID column that is mandatory and cannot be assigned NULL value, as shown in the error message below:



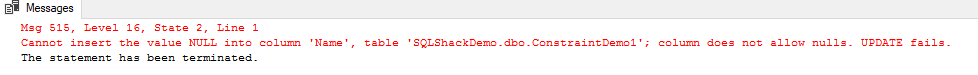
Checking the inserted data, you will see that only two records are inserted and the missing value for the Name column in the second INSERT statement will be NULL, which is the default value, as shown in the result below:



Assume that we need to prevent the Name column on the previous table from accepting NULL values after creating the table, using the ALTER TABLE T-SQL statement below:

**ALTER TABLE ConstraintDemo1 ALTER COLUMN [Name] VARCHAR(50) NOT NULL;**

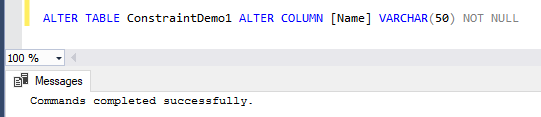
You will see that the command will fail, as it will check the existing values of the Name column for NULL values first before creating the constraint, as shown in the error message below:



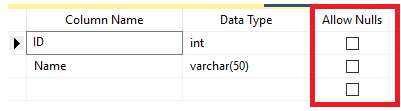
To enforce the NOT NULL Constraints in SQL, we should remove all NULL values of the Name column from the table, using the UPDATE statement below, that replaces the NULL values with empty string:

**UPDATE ConstraintDemo1 SET [Name]='' WHERE [Name] IS NULL;**

If you try to create the Constraints in SQL again, it will be created successfully as shown below:



The SQL NOT NULL constraint can be also created using the SQL Server Management Studio, by right-clicking on the needed table and select the Design option. Beside each column, you will find a small checkbox that you can use to specify the null-ability of that column. Unchecking the checkbox beside the column, a SQL NOT NULL constraint will be created automatically, preventing any NULL value from being inserted to that column, as shown below:



UNIQUE Constraints in SQL

The UNIQUE constraint in SQL is used to ensure that no duplicate values will be inserted into a specific column or combination of columns that are participating in the UNIQUE constraint and not part of the PRIMARY KEY. In other words, the index that is automatically created when you define a UNIQUE constraint will guarantee that no two rows in that table can have the same value for the columns participating in that index, with the ability to insert only one unique NULL value to these columns, if the column allows NULL.

Let us create a small table with two columns, ID and Name. The ID column cannot hold duplicate values due to the UNIQUE constraint specified with the column definition. No restriction defined on the Name column, as in the CREATE TABLE T-SQL statement below:

USE Test\_database

GO

CREATE TABLE ConstraintDemo2

(

ID INT UNIQUE,

Name VARCHAR(50) NULL

)

INSERT INTO ConstraintDemo2 ([ID],[NAME]) VALUES (1,'Ali')

GO

INSERT INTO ConstraintDemo2 ([ID],[NAME]) VALUES (2,'Ali')

GO

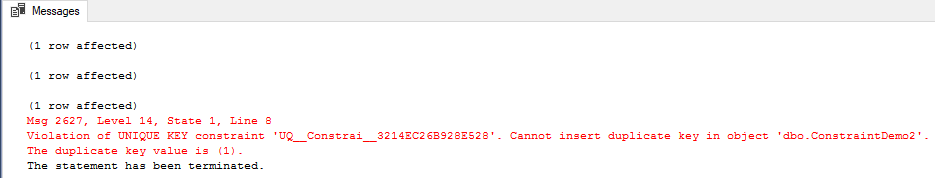
INSERT INTO ConstraintDemo2 ([ID],[NAME]) VALUES (NULL,'Adel')

GO

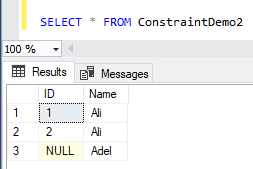
INSERT INTO ConstraintDemo2 ([ID],[NAME]) VALUES (1,'Faris')

GO

The first two records will be inserted successfully, with no constraint preventing duplicate values of the Name column. The third record will be inserted successfully too, as the unique ID column allows only one NULL value. The last INSERT statement will fail, as the ID column doesn’t allow duplicate values and the provided ID value is already inserted to that column, as shown in the error message below:



The three inserted rows will be as shown below:



The **INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS** system object can be easily used to retrieve information about all defined constraints in a specific table using the T-SQL script below:

SELECT CONSTRAINT\_NAME,

TABLE\_SCHEMA ,

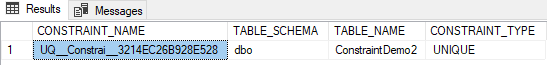
TABLE\_NAME,

CONSTRAINT\_TYPE

FROM INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS

WHERE TABLE\_NAME='ConstraintDemo2';

The previous query result will show the defined UNIQUE constraint in SQL in the provided table, which will be like:



SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint consists of one column or multiple columns with values that uniquely identify each row in the table.

The SQL PRIMARY KEY constraint combines between the UNIQUE and SQL NOT NULL constraints, where the column or set of columns that are participating in the PRIMARY KEY cannot accept a NULL value. If the PRIMARY KEY is defined in multiple columns, you can insert duplicate values on each column individually, but the combination values of all PRIMARY KEY columns must be unique. Take into consideration that you can define only one PRIMARY KEY per each table, and it is recommended to use small or INT columns in the PRIMARY KEY.

In addition to providing fast access to the table data, the index that is automatically created, when defining the SQL PRIMARY KEY, will enforce the data uniqueness. The PRIMARY KEY is used mainly to enforce the entity integrity of the table. Entity integrity ensures that each row in the table is a uniquely identifiable entity.

PRIMARY KEY constraint differs from the UNIQUE constraint in that; you can create multiple UNIQUE constraints in a table, with the ability to define only one SQL PRIMARY KEY per each table. Another difference is that the UNIQUE constraint allows for one NULL value, but the PRIMARY KEY does not allow NULL values.

Assume that we have the below simple table with two columns; the ID and Name. The ID column is defined as a PRIMARY KEY for that table, that is used to identify each row on that table by ensuring that no NULL or duplicate values will be inserted to that ID column. The table is defined using the CREATE TABLE T-SQL script below:

USE Test\_database

GO

CREATE TABLE ConstraintDemo3

(

ID INT PRIMARY KEY,

Name VARCHAR(50) NULL

)

If you try to run the three INSERT statements below:

INSERT INTO ConstraintDemo3 ([ID],[NAME]) VALUES (1,'John')

GO

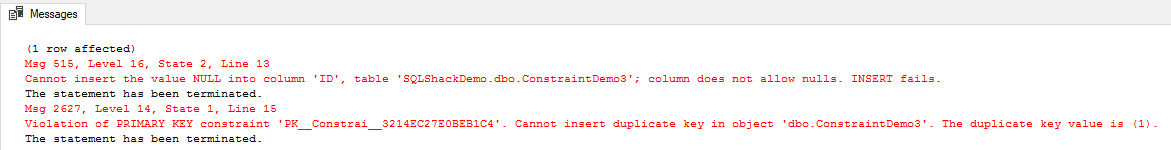
INSERT INTO ConstraintDemo3 ([NAME]) VALUES ('Fadi')

GO

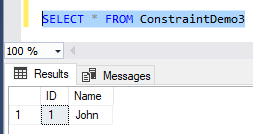
INSERT INTO ConstraintDemo3 ([ID],[NAME]) VALUES (1,'Saeed')

GO

You will see that the first record will be inserted successfully as both the ID and Name values are valid. The second insert operation will fail, as the ID column is mandatory and cannot be NULL, as the ID column is the SQL PRIMARY KEY. The last INSERT statement will fail too as the provided ID value already exists and the duplicate values are not allowed in the PRIMARY KEY, as shown in the error message below:



Checking the inserted values, you will see that only the first record is inserted successfully as below:



If you do not provide the SQL PRIMARY KEY constraint with a name during the table definition, the SQL Server Engine will provide it with a unique name as you can see from querying the INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS system object below:

SELECT CONSTRAINT\_NAME,

TABLE\_SCHEMA ,

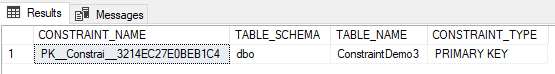
TABLE\_NAME,

CONSTRAINT\_TYPE

FROM INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS

WHERE TABLE\_NAME='ConstraintDemo3'

With the below result in our example:



## FOREIGN KEY Constraint

A Foreign Key is a database key that is used to link two tables together. The FOREIGN KEY constraint identifies the relationships between the database tables by referencing a column, or set of columns, in the **Child** table that contains the foreign key, to the PRIMARY KEY column or set of columns, in the **Parent**table.

The relationship between the child and the parent tables is maintained by checking the existence of the child table FOREIGN KEY values in the referenced parent table’s PRIMARY KEY before inserting these values into the child table. In this way, the FOREIGN KEY constraint, in the child table that references the PRIMARY KEY in the parent table, will enforce database referential integrity. Referential integrity ensures that the relationship between the database tables is preserved during the data insertion process. Recall that the PRIMARY KEY constraint guarantees that no NULL or duplicate values for the selected column or columns will be inserted into that table, enforcing the entity integrity for that table. The entity integrity enforced by the PRIMARY KEY and the referential integrity enforced by the FOREIGN KEY together form the key integrity.

The FOREIGN KEY constraint differs from the PRIMARY KEY constraint in that, you can create only one PRIMARY KEY per each table, with the ability to create multiple FOREIGN KEY constraints in each table by referencing multiple parent table. Another difference is that the FOREIGN KEY allows inserting NULL values if there is no NOT NULL constraint defined on this key, but the PRIMARY KEY does not accept NULLs.

The FOREIGN KEY constraint provides you also with the ability to control what action will be taken when the referenced value in the parent table is updated or deleted, using the ON UPDATE and ON DELETE clauses. The supported actions that can be taken when deleting or updating the parent table’s values include:

* **NO ACTION**: When the ON UPDATE or ON DELETE clauses are set to NO ACTION, the performed update or delete operation in the parent table will fail with an error.
* **CASCADE**: Setting the ON UPDATE or ON DELETE clauses to CASCADE, the same action performed on the referenced values of the parent table will be reflected to the related values in the child table. For example, if the referenced value is deleted in the parent table, all related rows in the child table are also deleted.
* **SET NULL**: With this ON UPDATE and ON DELETE clauses option, if the referenced values in the parent table are deleted or modified, all related values in the child table are set to NULL value.
* **SET DEFAULT**: Using the SET DEFAULT option of the ON UPDATE and ON DELETE clauses specifies that, if the referenced values in the parent table are updated or deleted, the related values in the child table with FOREIGN KEY columns will be set to its default value.

You can add the FOREIGN KEY constraint while defining the column using the CREATE TABLE T-SQL statement, or add it after the table creation using the ALTER TABLE T-SQL statement. We will create two new tables to understand the FOREIGN KEY constraint functionality. The first table will act as the parent table with the ID column defined as a PRIMARY KEY column. The second table will act as the child table, with the ID column defined as the FOREIGN KEY column that references the ID column on the parent table. This can be achieved using the T-SQL script below:

USE Test\_database

GO

CREATE TABLE ConstraintDemoParent

(

ID INT PRIMARY KEY,

Name VARCHAR(50) NULL

)

GO

CREATE TABLE ConstraintDemoChild

(

CID INT PRIMARY KEY,

ID INT FOREIGN KEY REFERENCES ConstraintDemoParent(ID)

)

After creating the two tables, we will insert three records to the parent table, and two records to the child table, using the following INSERT statements:

INSERT INTO ConstraintDemoParent ([ID],[NAME]) VALUES (1,'John'),(2,'Mika'),(3,'Sanya')

GO

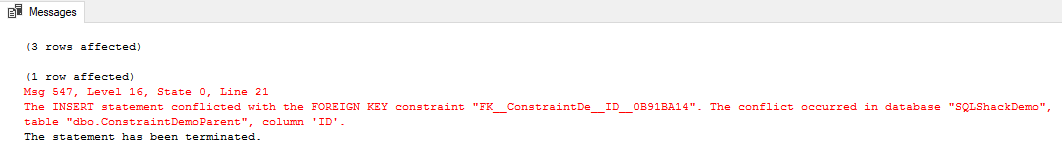
INSERT INTO ConstraintDemoChild (CID,ID) VALUES (1,1)

GO

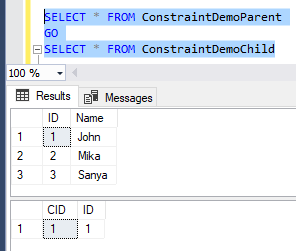
INSERT INTO ConstraintDemoChild (CID,ID) VALUES (2,4)

GO

The result will show you that the three records are successfully inserted into the parent table. The first record that we tried to insert into the child table is inserted without any error as the ID value of 1 already exists in the parent table. Trying to insert the second record into the child table will fail because the ID value of 4 doesn’t exist in the parent table, and due to the FOREIGN KEY constraint, you will not be able to insert an ID value to the child table that doesn’t exist in the parent table:



Checking the parent and child tables content, you will see that only one record is inserted into the child table, as you can see below:



As we did not mention the FOREIGN KEY constraint name while creating the child table, SQL Server will assign it a unique name that we can retrieve from the INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS system view using the following query:

SELECT CONSTRAINT\_NAME,

TABLE\_SCHEMA ,

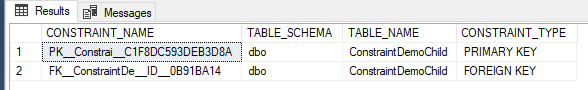
TABLE\_NAME,

CONSTRAINT\_TYPE

FROM INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS

WHERE TABLE\_NAME='ConstraintDemoChild'

The result in our case for the child table is as below:



## CHECK Constraint

A CHECK constraint is defined on a column or set of columns to limit the range of values, that can be inserted into these columns, using a predefined condition. The CHECK constraint comes into action to evaluate the inserted or modified values, where the value that satisfies the condition will be inserted into the table, otherwise, the insert operation will be discarded. It is allowed to specify multiple CHECK constraints for the same column.

Defining the CHECK constraint condition is somehow similar to writing the WHERE clause of a query, using the different comparison operators, such as AND, OR, BETWEEN, IN, LIKE and IS NULL to write its Boolean expression that will return TRUE, FALSE or UNKNOWN. The CHECK constraint will return UNKNOWN value when a NULL value is present in the condition. The CHECK constraint is used mainly to enforce the domain integrity by limiting the inserted values to the ones that follow the defined values, range or format rules.

Let us create a new simple table that has three columns; the ID column that is considered as the PRIMARY KEY of that table, Name, and Salary. A CHECK constraint is defined on the Salary column to make sure that no zero or negative values are inserted into that column. The CHECK constraint is defined within CREATE TABLE T-SQL statement below:

CREATE TABLE ConstraintDemo4

(

ID INT PRIMARY KEY,

Name VARCHAR(50) NULL,

Salary INT CHECK (Salary>0)

)

GO

If you execute the below three INSERT statements:

INSERT INTO ConstraintDemo4 ([ID],[NAME],Salary) VALUES (1,'John',350)

GO

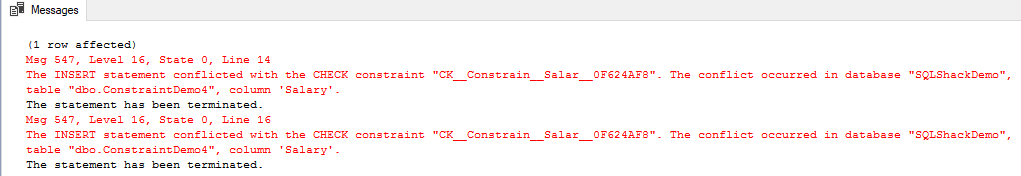
INSERT INTO ConstraintDemo4 ([ID],[NAME],Salary) VALUES (2,'Mike',0)

GO

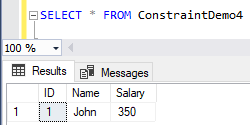
INSERT INTO ConstraintDemo4 ([ID],[NAME],Salary) VALUES (3,'Nikola',-72)

GO

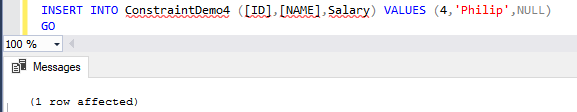
It is clear from the generated result that the first record is inserted with no error as the provided Salary value meets the checking condition. The second and third INSERT statements will fail, as the provided Salary values do not meet the CHECK constraint condition due to inserting zero and negative Salary values, as you can see in the following error message:



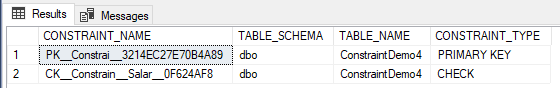
Checking the table’s data shows you that only the first row that passed the CHECK constraint condition will be inserted to the table as below:



We mentioned earlier in this article that the CHECK constraint allows inserting NULL values if the participated columns allow NULL, and the NULL values will be evaluated as UNKNOWN without throwing any error. This is clear from the below record that is inserted successfully, although the provided value of the Salary column is NULL:



If you review the CHECK constraint definition in the previous CREATE TABLE statement, you will see that we have not mentioned the name of the defined constraint. In this case, SQL Server will assign a unique name for that constraint that can be shown by querying the INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS system view for the ConstraintDemo4 table. The result will be as shown below:

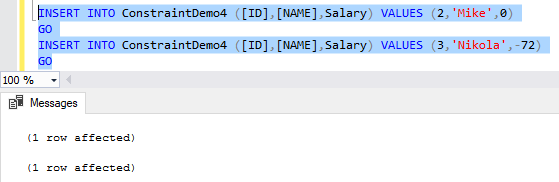


The CHECK constraint can be dropped using the ALTER TABLE T-SQL statement. Using the CHCEK constraint name we got previously, the below command can be used to drop the CHECK constraint on the ConstraintDemo4 table:

ALTER TABLE ConstraintDemo4

DROP CONSTRAINT CK\_\_Constrain\_\_Salar\_\_0F624AF8;

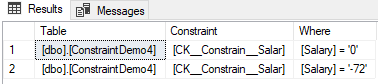
Once the CHECK constraint is dropped, the failed two INSERT statement will be executed successfully without any error, as there is no restriction on the inserted Salary values, as follows:



Let us use the DBCC CHECKCONSTRAINTS command to check the data that does not meet the condition of the ConstraintDemo4 table’s constraint

DBCC CHECKCONSTRAINTS(ConstraintDemo4)

The result will show you the two Salary values that break the CHECK constraint condition, as shown below:



## DEFAULT Constraint

A DEFAULT constraint is used to provide a default column value for the inserted rows if no value is specified for that column in the INSERT statement. The Default constraint helps in maintaining the domain integrity by providing proper values for the column, in case the user does not provide a value for it. The default value can be a constant value, a system function value or NULL.

In the below CREATE TABLE statement for a simple table with three columns, a DEFAULT constraint is defined on the EmployeeDate column, that assigns the GETDATE() system function value for that column in case we miss specifying it in the INSERT statement:

CREATE TABLE ConstraintDemo5

(

ID INT PRIMARY KEY,

Name VARCHAR(50) NULL,

EmployeeDate DATETIME NOT NULL DEFAULT GETDATE()

)

GO

If we execute the two INSERT statements below:

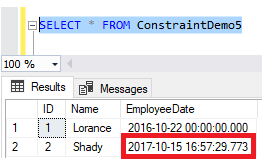
INSERT INTO ConstraintDemo5 ([ID],[NAME],EmployeeDate) VALUES (1,'Lorance','2016/10/22')

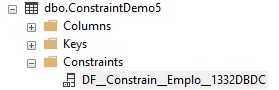
GO

INSERT INTO ConstraintDemo5 ([ID],[NAME]) VALUES (2,'Shady')

GO

And check the inserted records, you will see that the EmployeeDate column value for the second record, that we did not mention in the INSERT statement, is assigned to the current date and time value as shown below:

Expanding the Constraints node under the current table will show us the created DEFAULT constraint name, recalling that SQL Server will assign a unique name for it if we do not provide a name for it, as shown below:



SQL Create Table statement

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TABLE table\_name (      column\_name column\_type,      column\_name column\_type,      ); |

SQL Create Table example

Let’s take a look at the definition of our two tables:

First, we’ll define the **city** table.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | -- Table: city  CREATE TABLE city (      id int  NOT NULL IDENTITY(1, 1),      city\_name char(128)  NOT NULL,      lat decimal(9,6)  NOT NULL,      long decimal(9,6)  NOT NULL,      country\_id int  NOT NULL,      CONSTRAINT city\_pk PRIMARY KEY  (id)  ); |

Please notice a few things:

* NOT NULL -> This is a property telling us that this column can’t be empty (must be defined)
* IDENTITY(1, 1) -> is also a property of the column telling us that this value shall be generated automatically, starting from 1 and increasing by 1 for each new record
* CONSTRAINT city\_pk PRIMARY KEY (id) -> This is not a column, but the rule, telling us that column id shall contain only UNIQUE values. So only 1 city can have id =5

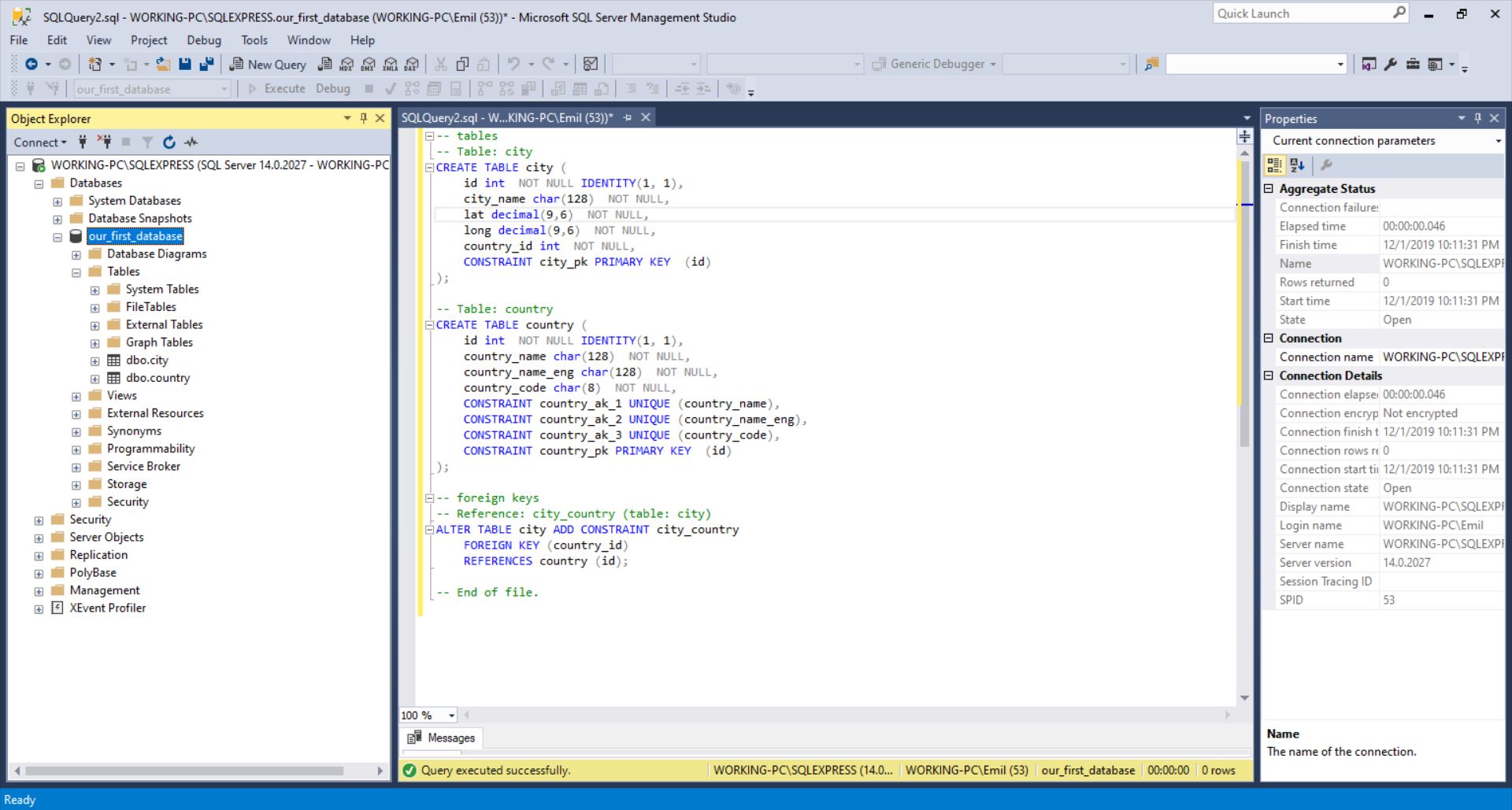
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | -- Table: country  CREATE TABLE country (      id int  NOT NULL IDENTITY(1, 1),      country\_name char(128)  NOT NULL,      country\_name\_eng char(128)  NOT NULL,      country\_code char(8)  NOT NULL,      CONSTRAINT country\_ak\_1 UNIQUE (country\_name),      CONSTRAINT country\_ak\_2 UNIQUE (country\_name\_eng),      CONSTRAINT country\_ak\_3 UNIQUE (country\_code),      CONSTRAINT country\_pk PRIMARY KEY  (id)  ); |

Here we have 1 new CONSTRAINT and that is the UNIQUE constraining. This one tells us that this value must be UNIQUE within this table. E.g. *CONSTRAINT country\_ak\_1 UNIQUE (country\_name)* defines that we can’t store 2 countries with the same name.

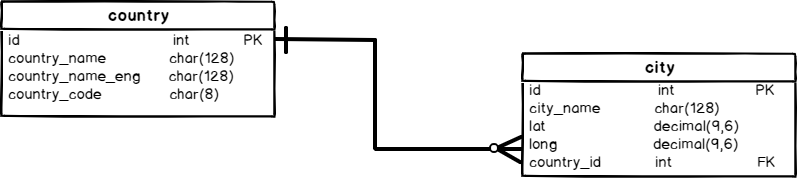
The last part of the script is the definition of foreign keys. We have only 1 such key and it relates city and country table (*city.county\_id = country.id*).

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | -- foreign keys  -- Reference: city\_country (table: city)  ALTER TABLE city ADD CONSTRAINT city\_country      FOREIGN KEY (country\_id)      REFERENCES country (id); |

Keys (primary and foreign) are too complex and shall be covered in a separate article. After executing these commands, the status of our database is as in the picture below:



INSERT INTO TABLE example



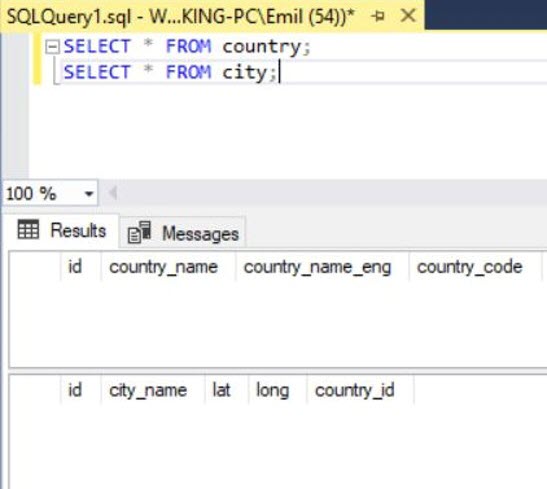
In order to check the contents of these two tables, we’ll use two simple SELECT statements:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM country;  SELECT \* FROM city; |

While SELECT is not the topic of this article, it should be mentioned that its’ basic syntax is:

SELECT 1 or more attributes FROM table;

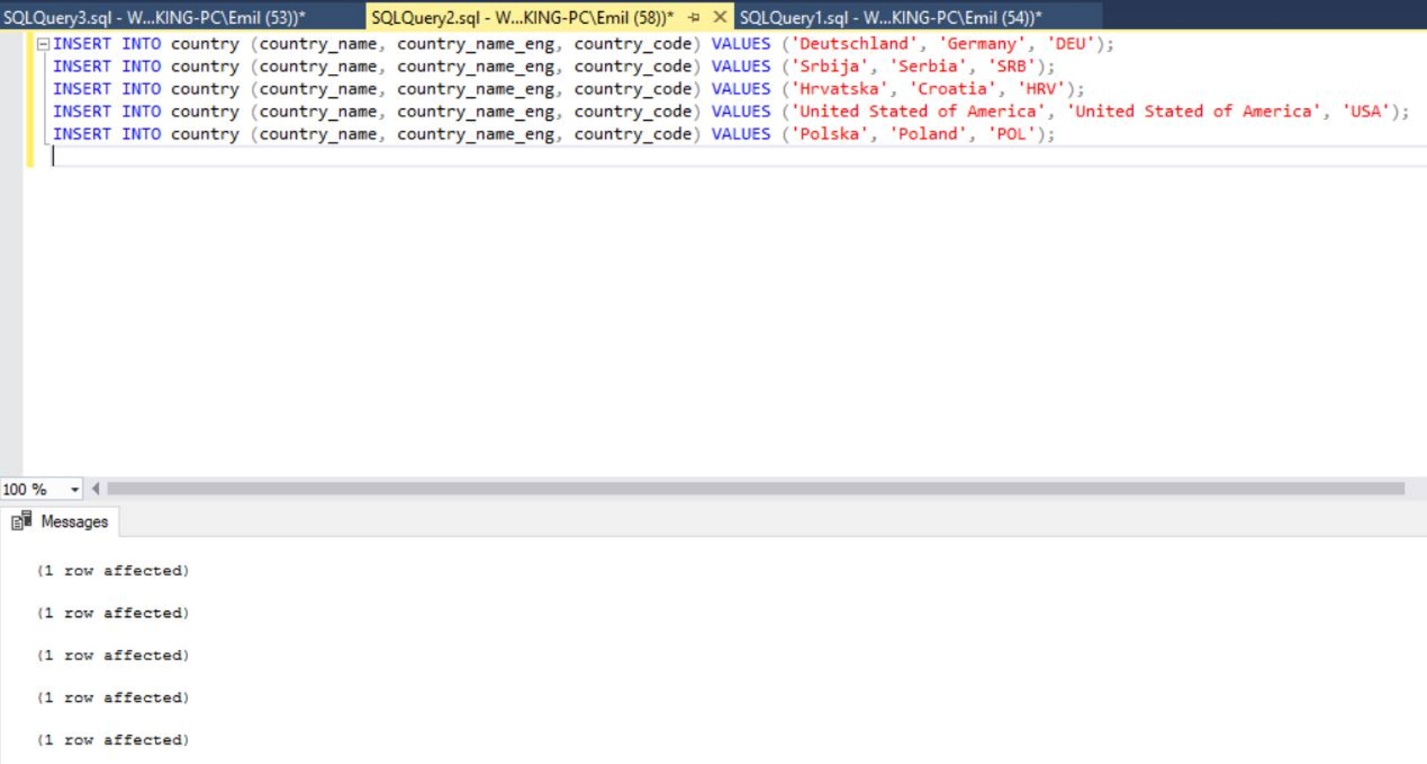
The star (\*) after SELECT represents that we want to show the values of all attributes/columns from that table in the query result.



Now, we’ll need to change that.

First, we’ll populate the **country** table using the following INSERT INTO TABLE statements:

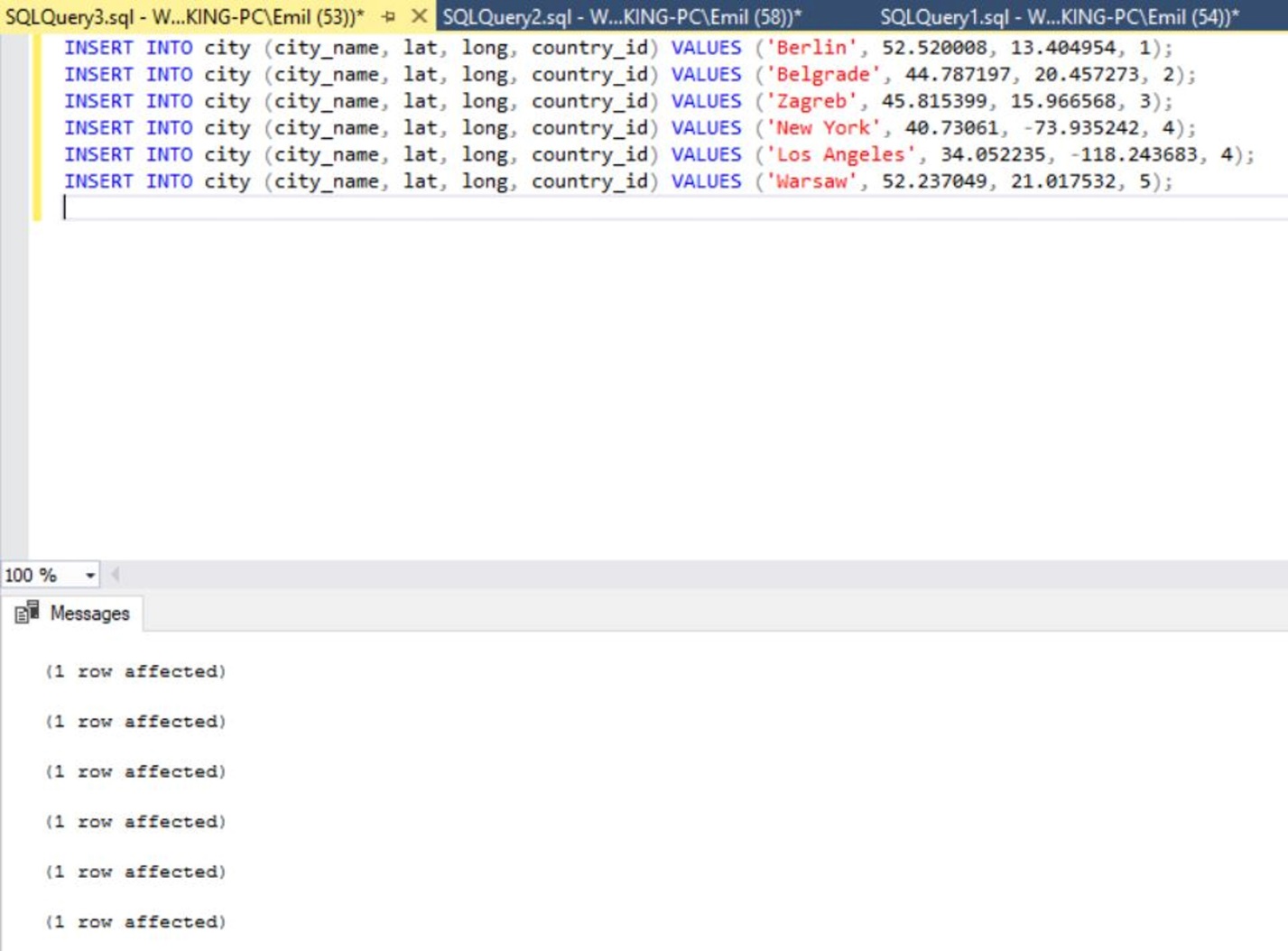
|  |  |
| --- | --- |
| 1  2  3  4  5 | INSERT INTO country (country\_name, country\_name\_eng, country\_code) VALUES ('Deutschland', 'Germany', 'DEU');  INSERT INTO country (country\_name, country\_name\_eng, country\_code) VALUES ('Srbija', 'Serbia', 'SRB');  INSERT INTO country (country\_name, country\_name\_eng, country\_code) VALUES ('Hrvatska', 'Croatia', 'HRV');  INSERT INTO country (country\_name, country\_name\_eng, country\_code) VALUES ('United Stated of America', 'United Stated of America', 'USA');  INSERT INTO country (country\_name, country\_name\_eng, country\_code) VALUES ('Polska', 'Poland', 'POL'); |



The next thing we need to do is to populate the **city** table. We’ll do that using the following statements:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | INSERT INTO city (city\_name, lat, long, country\_id) VALUES ('Berlin', 52.520008, 13.404954, 1);  INSERT INTO city (city\_name, lat, long, country\_id) VALUES ('Belgrade', 44.787197, 20.457273, 2);  INSERT INTO city (city\_name, lat, long, country\_id) VALUES ('Zagreb', 45.815399, 15.966568, 3);  INSERT INTO city (city\_name, lat, long, country\_id) VALUES ('New York', 40.73061, -73.935242, 4);  INSERT INTO city (city\_name, lat, long, country\_id) VALUES ('Los Angeles', 34.052235, -118.243683, 4);  INSERT INTO city (city\_name, lat, long, country\_id) VALUES ('Warsaw', 52.237049, 21.017532, 5); |

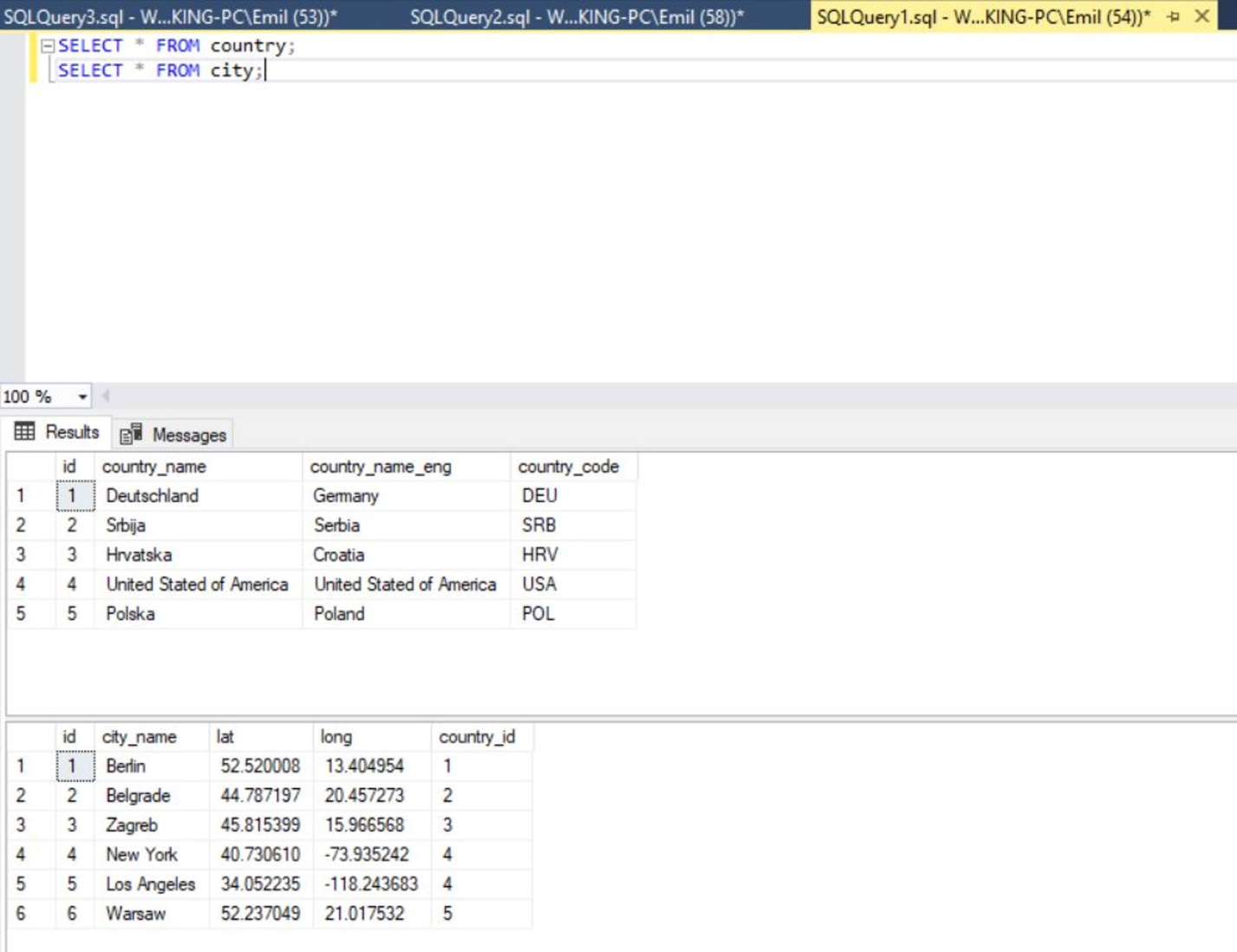
After executing these statements, this was the result. As expected, 6 rows were added. And once more we have 1 message for each insert in the **Messages** section:



SELECT – Check what was inserted

Now we’ll once more check what is stored in our tables. We’ll use the same two SELECT statements we have used previously:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM country;  SELECT \* FROM city; |



INFORMATION\_SCHEMA Database?

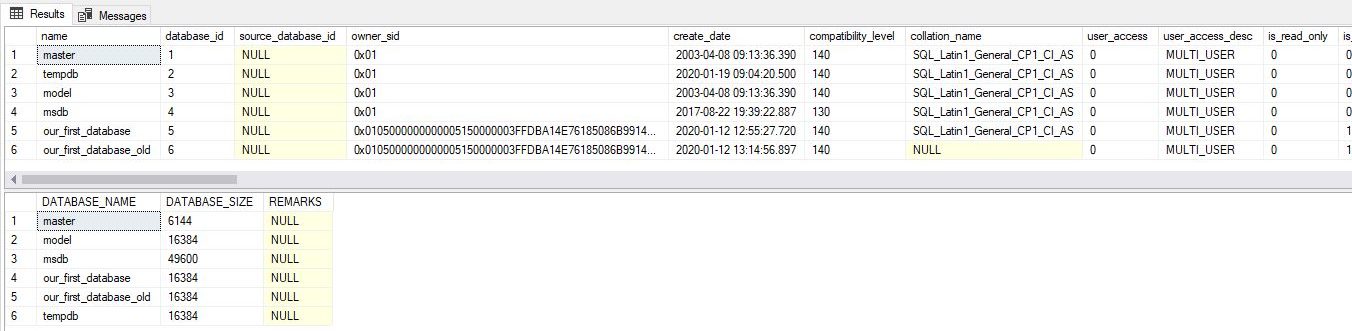
The INFORMATION\_SCHEMA database is an ANSI standard set of views we can find in SQL Server, but also MySQL. Other database systems also have either exactly such or similar database implemented. It provides the read-only access to details related to databases and their objects (tables, constraints, procedures, views…) stored on the server.

You could easily use this data to:

* Check what’s on the server and/or in the database
* Check if everything is as expected (e.g. compared to the last time you’ve performed this check)
* Automate processes and build some complex code (e.g. code generators – we’ll talk about this later)

Listing All Databases

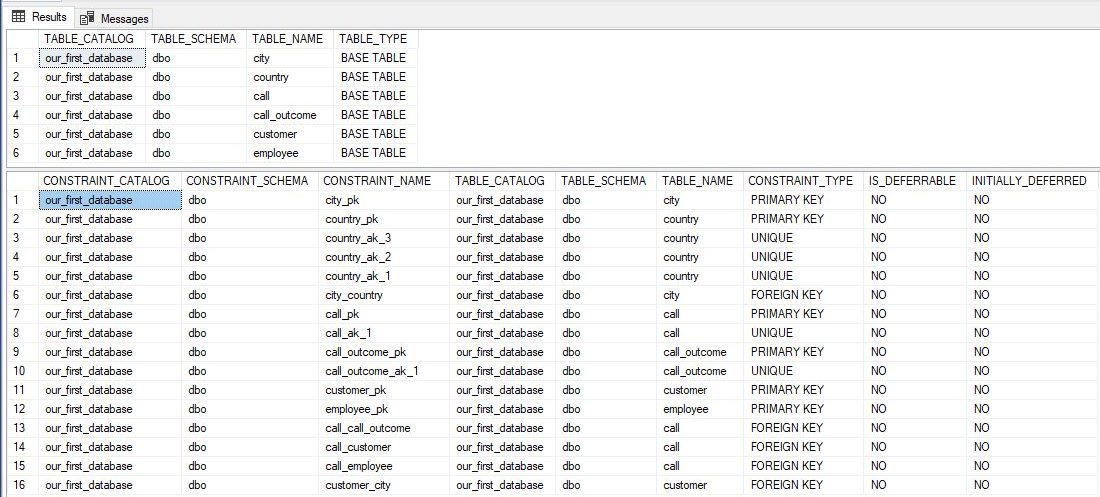
|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM sys.databases;  EXEC sp\_databases; |



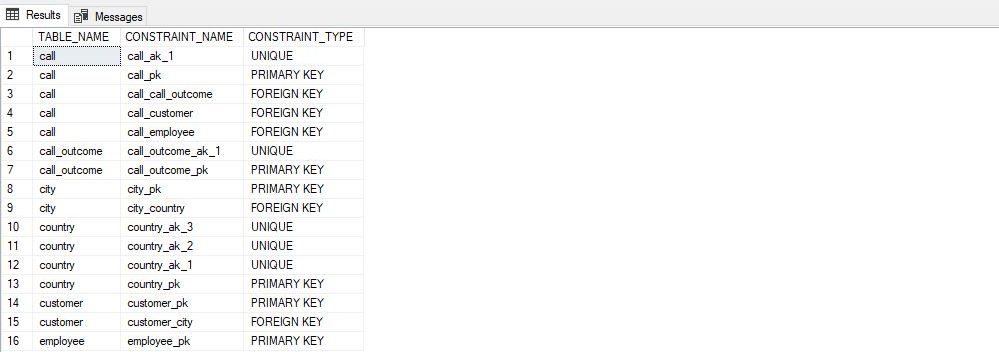
Using INFORMATION\_SCHEMA to Access Tables Data

Since this database is an ANSI standard, the following queries should work in other DBMS systems as well. We’ll list all tables in the database we’ve selected and also all constraints. To do that, we’ll use the following queries:

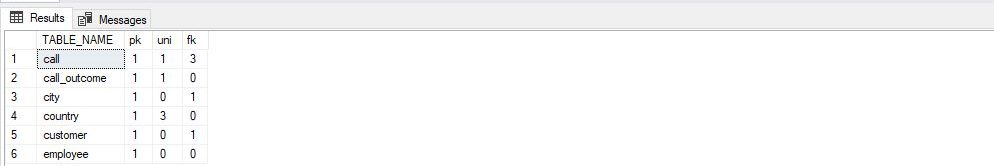
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | USE test\_database;    -- list of all tables in the selected database  SELECT \* FROM INFORMATION\_SCHEMA.TABLES;    -- list of all constraints in the selected database  SELECT \* FROM INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS; |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | USE test\_database;    -- join tables and constraints data  SELECT      INFORMATION\_SCHEMA.TABLES.TABLE\_NAME,      INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS.CONSTRAINT\_NAME,      INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS.CONSTRAINT\_TYPE  FROM INFORMATION\_SCHEMA.TABLES  INNER JOIN INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS ON INFORMATION\_SCHEMA.TABLES.TABLE\_NAME = INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS.TABLE\_NAME  ORDER BY      INFORMATION\_SCHEMA.TABLES.TABLE\_NAME ASC,      INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS.CONSTRAINT\_TYPE DESC; |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | USE test\_database;    -- join tables and constraints data  SELECT      INFORMATION\_SCHEMA.TABLES.TABLE\_NAME,      SUM(CASE WHEN INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS.CONSTRAINT\_TYPE = 'PRIMARY KEY' THEN 1 ELSE 0 END) AS pk,      SUM(CASE WHEN INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS.CONSTRAINT\_TYPE = 'UNIQUE' THEN 1 ELSE 0 END) AS uni,      SUM(CASE WHEN INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS.CONSTRAINT\_TYPE = 'FOREIGN KEY' THEN 1 ELSE 0 END) AS fk  FROM INFORMATION\_SCHEMA.TABLES  LEFT JOIN INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS ON INFORMATION\_SCHEMA.TABLES.TABLE\_NAME = INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS.TABLE\_NAME  GROUP BY      INFORMATION\_SCHEMA.TABLES.TABLE\_NAME  ORDER BY      INFORMATION\_SCHEMA.TABLES.TABLE\_NAME ASC; |



The INFORMATION\_SCHEMA Tables

It would be hard to try out every single table and show what it returns. At least, that would be hard to put into one readable article. I strongly encourage you to play with the INFORMATION\_SCHEMA database and explore what is where. The only thing I’ll do here is to list all the tables (views) you have at disposal. They are:

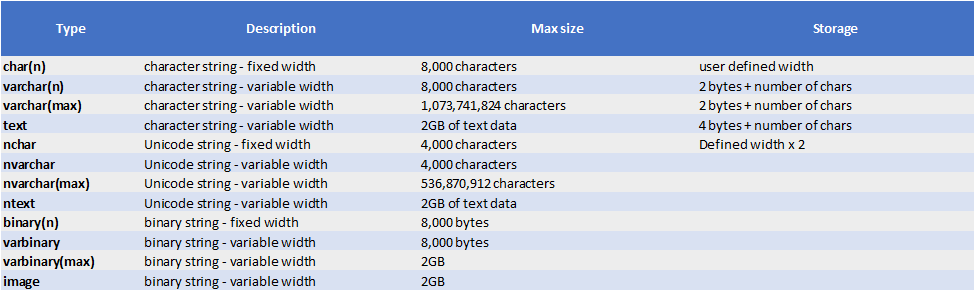
* CHECK\_CONSTRAINTS – details related to each CHECK constraint
* COLUMN\_DOMAIN\_USAGE – details related to columns that have an alias data type
* COLUMN\_PRIVILEGES – columns privileges granted to or granted by the current user
* COLUMNS – columns from the current database
* CONSTRAINT\_COLUMN\_USAGE – details about column-related constraints
* CONSTRAINT\_TABLE\_USAGE – details about table-related constraints
* DOMAIN\_CONSTRAINTS – details related to alias data types and rules related to them (accessible by this user)
* DOMAINS – alias data type details (accessible by this user)
* KEY\_COLUMN\_USAGE – details returned if the column is related with keys or not
* PARAMETERS – details related to each parameter related to user-defined functions and procedures accessible by this user
* REFERENTIAL\_CONSTRAINTS – details about foreign keys
* ROUTINES –details related to routines (functions & procedures) stored in the database
* ROUTINE\_COLUMNS – one row for each column returned by the table-valued function
* SCHEMATA – details related to schemas in the current database
* TABLE\_CONSTRAINTS – details related to table constraints in the current database
* TABLE\_PRIVILEGES –table privileges granted to or granted by the current user
* TABLES –details related to tables stored in the database
* VIEW\_COLUMN\_USAGE – details about columns used in the view definition
* VIEW\_TABLE\_USAGE – details about the tables used in the view definition
* VIEWS – details related to views stored in the database

**SQL data types:**

Textual Data Types

As you could see in our model, there are 3 most common types of real-world data we need to store: texts, numbers, and dates. Therefore, most DBMS (and programming languages) will support these 3 main groups. In each of these groups, you’ll find more or less standardized SQL data types.

So, let’s quickly review textual data types first. In the table below, you’ll find Transact-SQL string/textual data types with short descriptions.

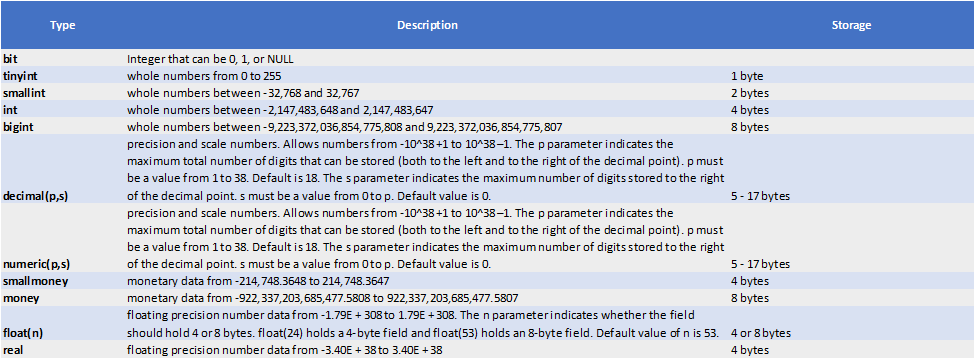


The most important are:

* char(n) – where n is replaced by the maximum number of characters we expect in any string. If that number is exceeded, we won’t be able to store the complete string. E.g. char(8) means that we’ll store 8 characters. If the string has less than that, the remaining characters shall be filled with blanks. Therefore, char(n), always has the same length, uses static memory allocation, and works 50% faster than varchar(n) (in an ideal scenario). It’s wise to use char(n) when you’re completely sure about the length of the data you’ll store, e.g. postal codes, account numbers, phone numbers, bank codes, etc.
* varchar(n) – serves the same purpose as char(n), but the difference here is that the string ends after the last character – we don’t add blanks to use the whole string. E.g. if you use varchar(255) and fill on 15 characters, then only 15 characters are stored. The problem here is that we don’t know the exact length of each string, so everything works slower (still, nothing critical). So, we’ve traded better performance in order to save some disk space
* text – well, if you want to store an immense text, you should use this one

Numeric Data Types

The next big group of SQL data types is definitely the one containing numeric data types. Once more, we have the overview given in the table below.

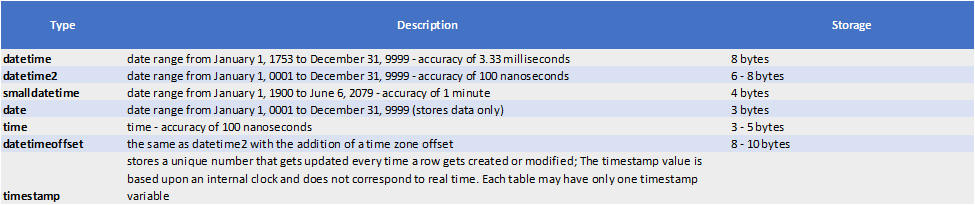


Still, I want to point to some of the most important types from this group:

* int – Whole numbers are the first numbers we learn. We count sheep to fall asleep – 1 sheep, 2 sheep… I want to say that everything starts with them. In most cases, you’ll use int – for primary and foreign keys these shall be unsigned. In other places, you’ll use int when you need to store whole numbers. In case, you want to spare some disc space, you could use tiny, or small int if you’re sure this will do. bigint should really be able to cover all your needs regarding whole numbers
* decimal(p,s) / float(n) / real – Are used to store decimal numbers. We can expect that most numerical values we want to store are actually decimal values – percentage, graphical coordinates, sports results etc.
* bit – Uses only 1 bit to store value 0 or 1 (NULL if not defined). It serves as a flag telling us if something stands or not. One of the most common usages of this type is to store info if the data in the related record is still valid/active/not deleted

Date & Time

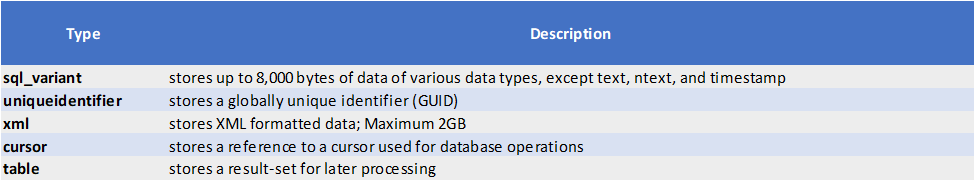
The last SQL data types group we’ll analyze are date & time types. The overview is given in the table below.



I would like to emphasize the following ones:

* date – In most cases, you’ll need to store date only, e.g. birth date, registration date, etc.
* datetime – Stores date and time in one attribute
* timestamp – Is not SQL Server standard, but you’ll probably meet it in other DBMSs. It serves the same purpose as datetime, but is UTC-time related

The list of the remaining interesting MS SQL data types is given in the table below.



## What is a Primary Key (PK)?

“Primary key is a value which is unique for every record in the table.”

And a rule – “Each table in the database should have a PK defined.”

“Primary key is a value, or a combination of few values from the table, uniquely defining each record in this table. If we know this value/combination, we can easily find the related record and access all remaining values from that record.”

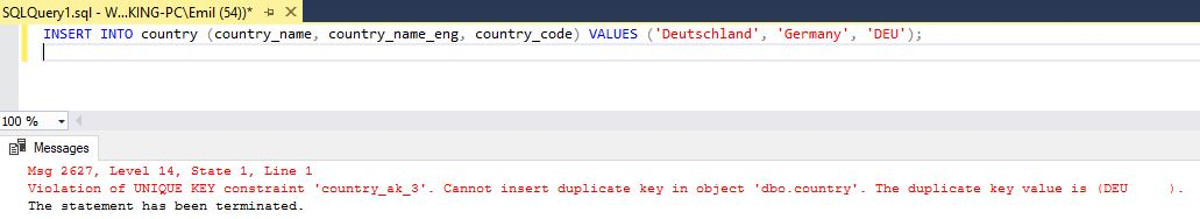
For instance, if you know the name of the country, you’ll easily find the row with that name and have access to remaining values related to that country – population, statistical data, etc.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | -- Table: city  CREATE TABLE city (      id int  NOT NULL IDENTITY(1, 1),      city\_name char(128)  NOT NULL,      lat decimal(9,6)  NOT NULL,      long decimal(9,6)  NOT NULL,      country\_id int  NOT NULL,      CONSTRAINT city\_pk PRIMARY KEY  (id)  );  -- Table: country  CREATE TABLE country (      id int  NOT NULL IDENTITY(1, 1),      country\_name char(128)  NOT NULL,      country\_name\_eng char(128)  NOT NULL,      country\_code char(8)  NOT NULL,      CONSTRAINT country\_ak\_1 UNIQUE (country\_name),      CONSTRAINT country\_ak\_2 UNIQUE (country\_name\_eng),      CONSTRAINT country\_ak\_3 UNIQUE (country\_code),      CONSTRAINT country\_pk PRIMARY KEY  (id)  ); |

## Primary key VS. UNIQUE (alternate keys)

When I’m designing a database, I always stick to a few rules regarding PKs and unique values. This is the best I’ve learned (so far) and besides technical reasons, it’s also worth mentioning that using this approach you’ll stay consistent throughout the whole database model. So, my rules are:

* **Each table** in the database **should have the PK defined**. This will not only improve the overall database performance but is also essential in order that data are related and consistent
* In each table, I’ll add the **column named id**. It shall be used as a PK column, the type is unsigned integer, **with IDENTITY set to (1,1)**. This way, the DBMS will automatically generate PK values as we add rows. Using integers as PKs also significantly improves the performance (index is created over that attribute automatically! – indexes shall be covered in a separate article)
* **All attributes**, besides PKs, **which contain unique values, should be defined as UNIQUE** (alternate keys). This property could be defined on a single attribute, or on the combination of few attributes. This will prevent inserting unwanted duplicated values
  + E.g. if we don’t have a UNIQUE defined on a country\_name and we insert country with the same name twice, we’ll have 2 records with different ids with the same country\_name. The DBMS would treat these as 2 different countries. Having the alternate key/UNIQUE defined prevents this from happening. Let’s take a look at an example. We already have the country with the name “Deutschland” in our database

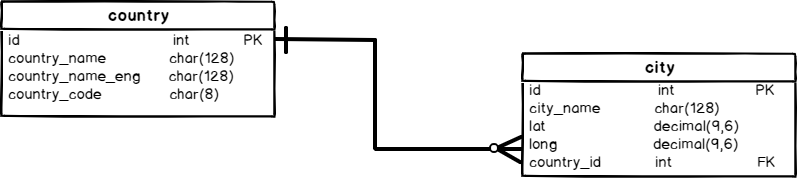


Since we’ve defined the UNIQUE rules on 3 columns, all 3 of them will prevent this Insert statement and this is the desired behavior. We’ve defined rules and the system takes care of whether the data we want to insert follow these rules or not.

Foreign key graphical representation

Before we define what FK is, we’ll once more remind ourselves of what is the primary key:

*“Primary key is a value, or a combination of few values from the table, uniquely defining each record in this table. If we know this value/combination, we can easily find the related record and access all remaining values from that record.”*



In our model, both tables, **country**, and **city** have 1 attribute which is used as a primary key and that is the **id** attribute. The value of this attribute is defined to start from 1 and increment by 1 for each new row added. That way, the system automatically ensures that we don’t have duplicate values.

The one thing which is pretty obvious is that we have a line connecting our tables. This line goes from **country.id** to **city.country\_id** and there is a good reason for that. This is how we graphically represent a FK.

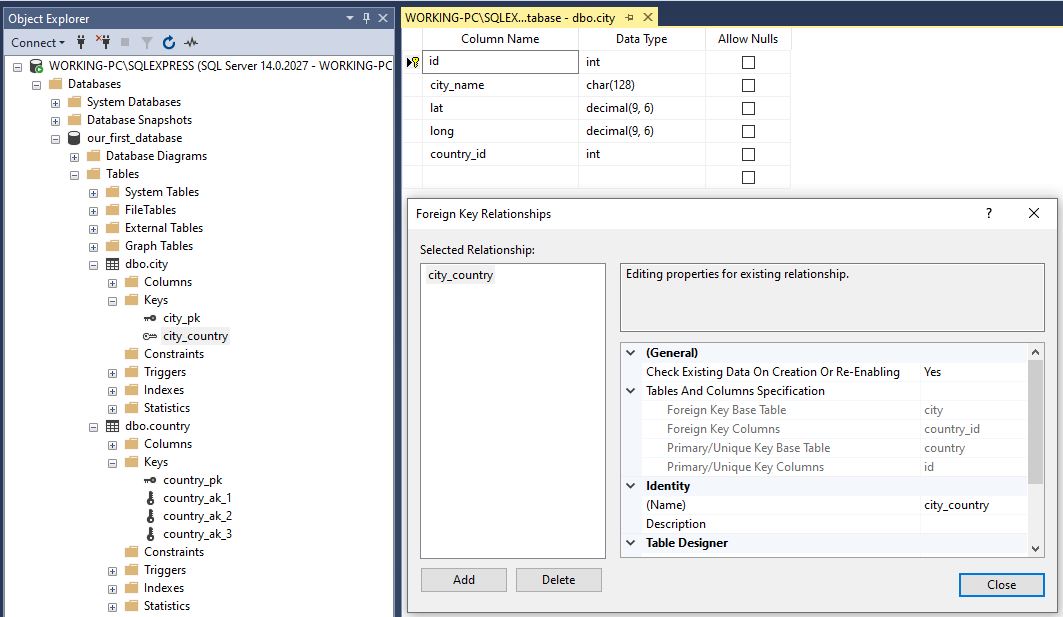
What is a foreign key?

*“The foreign key is an attribute, or more of them, directly related to the primary key of another table. When properly set, this rule shall ensure that we must always set the value of that attribute to exactly 1 value from the referenced table. This is the way how we relate data stored in different tables in our database model.”*

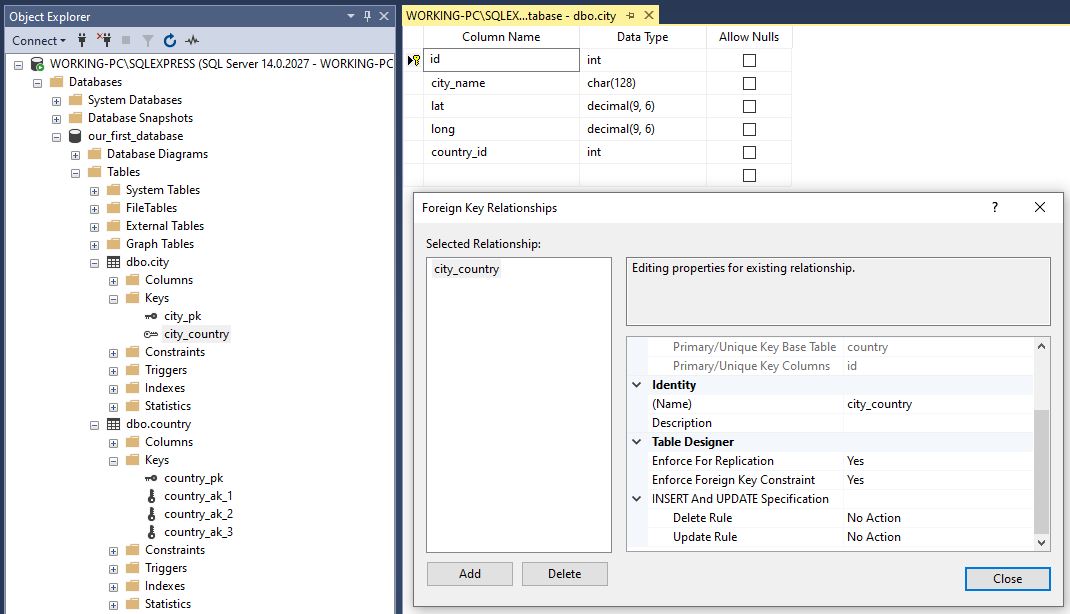
**Note:** If primary keys are always defined as a single column, then each FK shall be exactly 1 attribute. While this might not look so important at first, it actually proves to be a good “rule” to follow when designing a database. We’ll talk more about that in the next article related to indices.

Foreign key – How it’s stored/defined in the database?

After the theory, it’s a good time to take a look at when is actually in the database. We’ll do it using the database we’ve created previously (same diagram as the one used in the first section of this article):



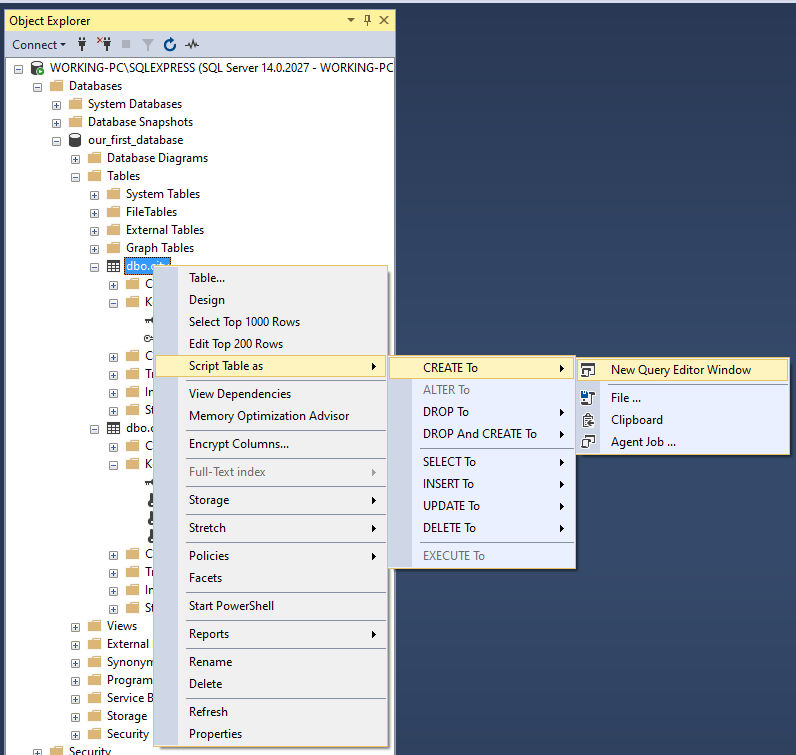
After I clicked on the relationship **city\_country** in Object explorer, the Foreign Key Relationship window popped up. Under ***General****->****Tables And Columns Specification*** you can see the previously mentioned relation between 2 columns in out 2 tables. If we scroll down a little, we can also see the remaining properties:



You can see the following settings:

* **Enforce for replication** – possible values here are “Yes” and “No” – Enables or disables it for replication. Please read more about the replication in SQL Server Replication (Merge) – What gets replicated and what doesn’t article
* **Enforce Foreign Key Constraint** – possible values are “Yes” and “No”. If it’s enabled, the FK constraint works as we want it. We could disable it if we want, but we should have a good reason for that. One could be that we want to manually change keys values and we wouldn’t be able to do that because constraints would “scream” all the time
* **Delete Rule** – possible values are “No Action”, “Cascade”, “Set NULL”, and “Set Default” – defines what will happen with child table rows if we delete the related row from the referenced table
* **Update Rule** – possible values are “No Action”, “Cascade”, “Set NULL”, and “Set Default” – defines what will happen with child table rows if we update the related row in the referenced table

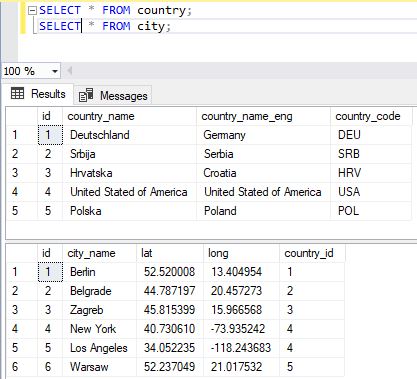
Now we’ll examine the code to see how the FK was defined. Let’s right-click on the table **dbo.city** and then **Script Table as**|**CREATE to**|**New Query Editor Window** just as on the picture below:



The result is the following code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | USE [test\_database]  GO    SET ANSI\_NULLS ON  GO    SET QUOTED\_IDENTIFIER ON  GO    CREATE TABLE [dbo].[city](      [id] [int] IDENTITY(1,1) NOT NULL,      [city\_name] [char](128) NOT NULL,      [lat] [decimal](9, 6) NOT NULL,      [long] [decimal](9, 6) NOT NULL,      [country\_id] [int] NOT NULL,  CONSTRAINT [city\_pk] PRIMARY KEY CLUSTERED  (      [id] ASC  )WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]  ) ON [PRIMARY]  GO    ALTER TABLE [dbo].[city]  WITH CHECK ADD  CONSTRAINT [city\_country] FOREIGN KEY([country\_id])  REFERENCES [dbo].[country] ([id])  GO    ALTER TABLE [dbo].[city] CHECK CONSTRAINT [city\_country]  GO |

Foreign key – What it actually does?



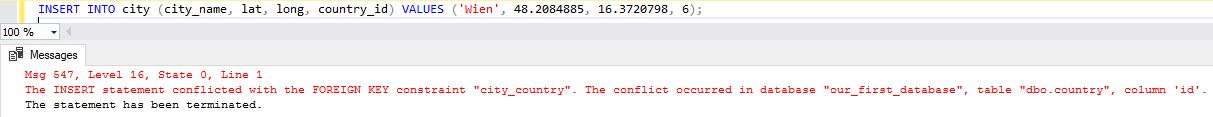
FK will do a few checks, but most important are:

* If you add a row to a table, the attribute which is part of the foreign key **must have the pair in the original table** (1:n). It can be NULL if the type of that relation (FK) is not mandatory (0:n). In any case, it can’t contain a value not existing in the referenced table

In our case, this means, that we can’t add a **city** if the **country\_id** is not in the range [1,5] – set of **id** values from the **country** table. If we try to execute something like this (notice that in this statement **country\_id** = 6 and we don’t have a pair in the **country** table):

|  |  |
| --- | --- |
| 1 | INSERT INTO city (city\_name, lat, long, country\_id) VALUES ('Wien', 48.2084885, 16.3720798, 6); |

This is what happens:



And this is great because the FK prevented us from making a mistake

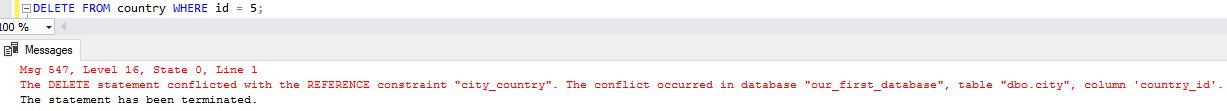
**Note:** In case we could execute this INSERT statement, we would insert a city belonging to a country with id = 6, without actually knowing that country. Later, someone could insert e.g. the Netherlands as the country with id = 6 and we would have inconsistent data

* When we delete data from the referenced table, we won’t be able to delete if there are related records in the child table. This stands when setting *Delete Rule* is set to “No Action” (this should be how you set your foreign key in most cases). “Cascade” would also delete rows from the child table, while “Set NULL” and “Set Default” wouldn’t delete entire rows from the child table, but just set the values of these attributes to NULL or predefined value

Let’s try to delete country with id = 5 (Poland) using the statement:

|  |  |
| --- | --- |
| 1 | DELETE FROM country WHERE id = 5; |

The result is shown below:



Once again, this is really great because the defined rule (FK), prevented us from deleting something we still use

**Note:** If the statement would delete Poland, then Warsaw would be assigned to the non-existing country. While we don’t know which country it belongs to, someone could later insert another country with id = 5 (as for inserts) and we would have a problem with our data.

The FK, when defined properly, instead ofm you, does the job in the background. It takes care of the referential integrity in our database.

SELECT id, country\_name\_eng

FROM country

WHERE id = 2;

SELECT id, country\_name\_eng

FROM country

WHERE id > 2;

SELECT id, country\_name\_eng

FROM country

WHERE id = 6;

SELECT city.id AS city\_id, city.city\_name, country.id AS country\_id, country.country\_name, country.country\_name\_eng, country.country\_code

FROM city

INNER JOIN country ON city.country\_id = country.id

WHERE country.id IN (1,4,5);