

POLUDO INSTITUTE OF TECHNOLOGY & MEDIA



SQL TRAINING

**Module 3**

Creating and maintaining tables

Jesse Teixeira – [jesseteixeira@poludo.ca](mailto:jesseteixeira@poludo.ca)

2018

# Module 3: Creating and maintaining tables

The students will learn how to create and maintain database tables and how to work with table relationships.

## Lessons

- Modelling the database
- Creating tables
- Applying the constraints
- Creating table relationships

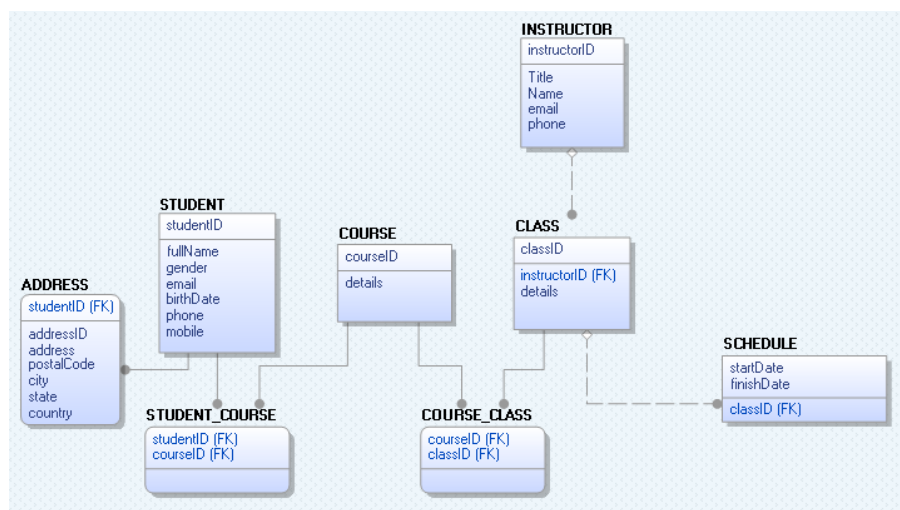
After completing this module, students will be able to:

- Create tables using the design tools
- Understand how the SQL constraints works
- Create tables relationships

## MODELING THE DATABASE

Before creating tables, it is important to always create a model of your database. By doing this you can have a better perception of your table roles and avoid some errors, like wrong relationships or invalid fields. There are several tools that can help you modeling your database, like the SQL Power Architect®, the ERWIN® and so forth. The image below shows us a database model created with the ERWin® software with the tables we will need to create for our school administration. In the references session you will find links to learn how to use the ERWin®. Some things to consider in this model design are:

- One Student can be enrolled in zero or more courses
- One course has zero or more students enrolled.
- A course is composed by one or more classes.
- A class can belong to one or more courses.
- A class can have only one instructor.
- A instructor can teach in more than one class.
- A class can have more than one schedule.
- One schedule belongs only to one class.
- One address belongs only to one student (does it?)



## CREATE TABLE SYNTAX

To create a table, use the create table statement. This statement is universal, having only some minor changes from one relational database management system (RDBMS) to another.

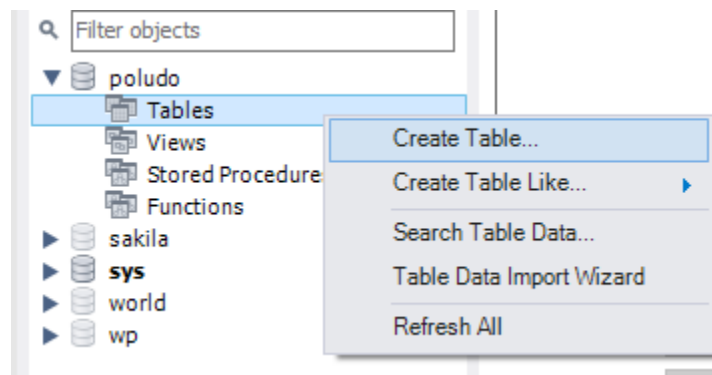
```
CREATE TABLE TableName
(
    columnName1 data_type(size) constraintName,
    columnName2 data_type(size) constraintName,
    columnName3 data_type(size) constraintName,
    ...
);
```

- **ColumnName**: specify the names of the table columns. Each column name must be unique inside the same table.
- **Data\_type**: specifies the type of data the column can hold (e.g. varchar, integer, decimal, date, etc.).
- **Size**: define the exact or maximum length of the column.
- **ConstraintName**: As described in the module 2, a constraint defines some attributes for a given column:
  - ✓ NOT NULL - Indicates that a column cannot store NULL values
  - ✓ UNIQUE - Ensures that each row for a column must have a unique value
  - ✓ PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have a unique identity which helps to find a particular record in a table more easily and quickly
  - ✓ FOREIGN KEY - Ensure the referential integrity of the data in one table to match values in another table
  - ✓ CHECK - Ensures that the value in a column meets a specific condition
  - ✓ DEFAULT - Specifies a default value for a column

## CREATING OUR TABLES

With the database model done, we can start to create our tables. We can create a table using the SQL language on the query editor or using the design option. We will make use of the two approaches to design our tables, because as said before, we can reuse these commands in others database managers with only a few or no changes at all.

- In the object explorer, select your database and click on create table



- In the query window we will add our columns one by one, defining its data type and some more useful information, as constraints.

[illegible]

Sometimes you will have to change your design for one or more reasons, but every time you do it make sure that you are not going against the data integrity and normalization rules and the main purpose of your database. If everything is ok, always update your model as well.

In our model, we defined the *studentID* as the primary Key for our table. Notice that we also set the studentID to be auto generated (auto increment), which means that the ID will be generated automatically. We also defined that all the columns are set to not accept null values.

Save the table with the name **student**.

- If you want to create the same table using the SQL language, you should write:

```
CREATE TABLE `poludo`.`student` (  
  `studentID` INT NOT NULL AUTO_INCREMENT,  
  `fullName` VARCHAR(45) NOT NULL,  
  `gender` CHAR(1) NOT NULL,  
  `email` VARCHAR(45) NOT NULL,  
  `birthDate` DATE NOT NULL,  
  `phone` VARCHAR(45) NOT NULL,  
  `mobile` VARCHAR(45) NULL,  
  PRIMARY KEY (`studentID`))  
COMMENT = 'This is the student table, containing only Student related information';
```

To test our newly created table, let's try to insert a student and analyze the result. Run the SQL statement below and discuss the result with the instructor

```
insert into poludo.student (fullName, gender,email,birthDate,phone)  
values ('John Doe','m','johnDoe@poludo.ca','1983-05-04','604333525')
```

## CREATING THE RELATIONSHIPS

A relationship is a connection that you create between two tables and defines how the data in them should be related. For example, a Student table and Course table can be related in order to show the student name that is associated with each course. There are three types of tables' relationships:

- **One-to-one relationships**

Each row in one table is linked to one and only one row in another table. In a one-to-one relationship between Table A and Table B, each row in Table A is linked to another row in Table B.

- **One-to-many relationships**

Each row in one table can be related to many rows in the relating table. This effectively save storage as the related record does not need to be stored multiple times in the relating table.

- **Many-to-many relationships**

One or more rows in a table can be related to 0, 1 or many rows in another table. In this kind of relationship, we need to use junction tables, also known as mapping tables.

## FOREIGN KEY, UPDATE AND DELETE RULES

When we create a relationship between tables, it means that we are working, directly or not, with foreign keys that can be placed inside one of the related tables or in an auxiliary junction table.

When applying update or delete operations on parent tables, there may be different requirements about the effect on associated values in the child tables (the ones that carries the foreign key). There are four available options to handle this situation.

SPECIFICATION	UPDATE ON PARENT	DELETE ON PARENT
<b>NO ACTION</b>	Not allowed. Error message would be generated.	
<b>CASCADE</b>	Associated values in child table would also be updated or deleted	
<b>SET NULL</b>	Associated values in child table would be set to NULL. Foreign key column should allow NULL values to specify this rule.	
<b>SET DEFAULT</b>	Associated values in child table would be set to default value specified in column definition. Also default value should be present in primary key column. Otherwise basic requirement of FK relation would fail and update/delete operation would not be successful. If no default value is provided in foreign key column this rule could not be implemented.	

■ If nothing is specified then the default rule is No Action.

We will learn how to construct the correct relationship between the tables and which is the best approach for our child tables based on the



parent table changes. Let's create the tables course, address, instructor, class and schedule. We can create the tables using the object explorer or coding the query in the query editor.

- Course

This table will have nothing more than the course id and details.

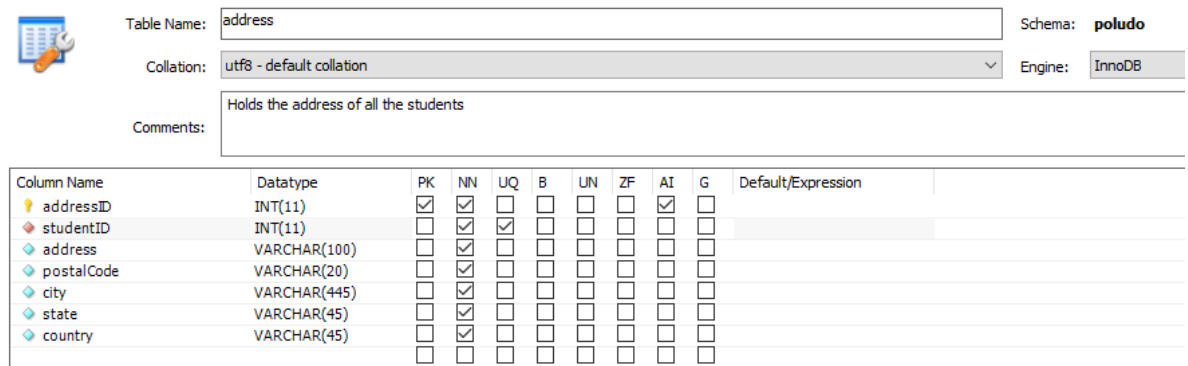
- Instructor

Our table representing the instructors must have the basic information about the teacher.

- Address

As shows the design, one address belongs to one student and one student can have only one address. As a result we need to create a table that contains a foreign key to the table student, but if we just do that we will be allowing a student to have more than one address, right?

To avoid this situation, we will use another constraint that we already know, the UNIQUE constraint, which avoids any duplicated values in our columns, what means that if we try to add more than one address to the same student we will have a problem. For our address table we will define the primary key to be an auto increment value the same way we did with our student table.



Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
addressID	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
studentID	INT(11)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
address	VARCHAR(100)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
postalCode	VARCHAR(20)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
city	VARCHAR(445)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
state	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
country	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Notice that we have created the studentID column as unique but still did not defined it as a foreign key of the student table. Click on the

“foreign keys” tabs and create the appropriate relationship. Set the onDelete option to “CASCADE”

Foreign Key Name	Referenced Table	Column	Referenced Column	Foreign Key Options
student_FK	poludo`.`student`	<input type="checkbox"/> addressID		On Update: NO ACTION
		<input checked="" type="checkbox"/> studentID	studentID	On Delete: CASCADE
		<input type="checkbox"/> address		
		<input type="checkbox"/> postalCode		
		<input type="checkbox"/> city		
		<input type="checkbox"/> state		
		<input type="checkbox"/> country		

☐ Skip in SQL generation

Create your table (below is the SQL code for the creation)

```
CREATE TABLE `address` (  
  `addressID` int(11) NOT NULL AUTO_INCREMENT,  
  `studentID` int(11) NOT NULL,  
  `address` varchar(100) NOT NULL,  
  `postalCode` varchar(20) NOT NULL,  
  `city` varchar(445) NOT NULL,  
  `state` varchar(45) NOT NULL,  
  `country` varchar(45) NOT NULL,  
  PRIMARY KEY (`addressID`),  
  UNIQUE KEY `studentID_UNIQUE` (`studentID`),  
  KEY `student_FK_idx` (`studentID`),  
  CONSTRAINT `student_FK` FOREIGN KEY (`studentID`) REFERENCES `student` (`studentID`) ON DELETE CASCADE ON UPDATE NO ACTION  
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8 COMMENT='Holds the address of all the students';
```

Now, if you try to insert two addresses with the same student ID:

```
insert into poludo.address (studentid,address,postalcode,city,state,country)  
values (1,'000 some street','ABC123','Vancouver','British Columbia','Canada');
```

you will have the following error (1 is the student id):

```
Error Code: 1062. Duplicate entry '1' for key 'studentID_UNIQUE'
```

We defined that if a Student is deleted, its address will also be deleted, because there is no meaning having an address of a non-existing student.

*\*Now that we created our address and student tables, we realized that maybe is not a good idea to make the address have a FK of the student table, because we can have more than one student, for example, living in the same address, so should we repeat all the info just because of that? And how about other people, like the instructors? Can't they make use of the same address table? This is a very clear situation where we must change our tables and also our design. Let's do it before we go on. Follow the instructor lead to fix these issues in classroom.*

After fixing our student and address tables, let's continue creating the remaining tables, but always making sure we are really following the best approach for our columns and relationships :)

- Instructor

The table that represents the instructor will also differ from the original design, as it will now hold a foreign key for the address table. All the other columns can remain the same.

- Class

The table that represents a class must have a foreign key to instructor, because as shown in the diagram, one class can have only one instructor, but an instructor can teach in zero or more classes, so it is a many-to-one relation (many classes to one instructor).

- Schedule

The table schedule will work with a different kind of approach. This is a **many-to-one** relationship, so work with the instructor to create the proper constraint.

## MANY TO MANY RELATIONSHIPS

So far all the relationships we created are simple, like many to one (classes and instructors) and one to many (classes and schedules). But what should we do with tables that have relationships of the type many to many? We can't just add a foreign key to each table because one column can only reference one value. Maybe you can think about the possibility of putting one more column in the table class, just in case we have two courses using the same class. At first glance it seems to be a good idea, but what if we have 50 courses using the same class? Will you create 50 more columns?

For that situation, the best approach is to create auxiliary tables (junction tables) that will contain nothing more than the relations between the two tables. For example, every class can belong for one or more courses and a course can be formed for one or more classes, we need to create a junction table to hold the information about it:

```
CREATE TABLE `poludo`.`course_class` (  
  `courseID` INT NOT NULL,  
  `classID` INT NOT NULL,  
  PRIMARY KEY (`courseID`, `classID`),  
  INDEX `class_ID_idx` (`classID` ASC),  
  CONSTRAINT `course_FK`  
    FOREIGN KEY (`courseID`)  
      REFERENCES `poludo`.`course` (`courseID`)  
    ON DELETE NO ACTION  
    ON UPDATE NO ACTION,  
  CONSTRAINT `class_ID`  
    FOREIGN KEY (`classID`)  
      REFERENCES `poludo`.`class` (`classID`)  
    ON DELETE NO ACTION  
    ON UPDATE NO ACTION);
```

The script above creates a new junction table that will hold the information about the classes and courses. Doing that we can relate any class with any course with no problems. Because the primary key is a junction of the two fields, we are preventing the user to insert the same relationship twice.

Now use the same approach to create the table *student\_course* that will represent the relationship between all the students and their courses.

## ALTERING TABLE COLUMNS

We use the ALTER table statement to add, delete or modify the columns in an existing table.

- Removing a column

```
ALTER TABLE table_name  
DROP COLUMN column_name;
```

Remember that in some database systems we are not allowed to delete or change columns. In our example, suppose that we do not want the fullname of the student as a column. The command would be:

```
ALTER TABLE Student  
DROP COLUMN fullName;
```

- Adding a new column

```
ALTER TABLE table_name  
ADD column_name datatype;
```

Where the datatype must be one of the types accepted by your database system. For example, now that we removed our student fullName column, we would like to replace it by three new columns: First name, middle name and surname, what will facilitate our future reports. To add those new columns just do:

```
ALTER TABLE Student  
ADD firstName varchar(50) NOT NULL,  
    middleName varchar(50),  
    surName    varchar(50) NOT NULL;
```

If we want to add a column indicating the hiring date of our instructor we should use the command:

```
ALTER TABLE Instructor  
ADD hireDate varchar(50);
```

The alter table command only allows us to insert new columns that can contains null in tables that already contain records.

- Changing a column data type

```
ALTER TABLE table_name  
ALTER COLUMN column_name datatype;
```

In the last example, we defined that the hireDate column of the table Instructor were from the data type varchar, but we know that this is not the best data type to represent date, so we need to change it to the proper datatype: the date.

```
ALTER TABLE Instructor  
ALTER COLUMN hireDate date;
```

When the column value conversion can be done automatically we do not receive any error and the column type is changed with no problems. But there is some datatypes that can't be converted automatically. For example, if we try to change our Instructor hireDate column to an **int** datatype, an error will occur.

## DROPPING A TABLE

The syntax to drop (delete) a table is:

```
DROP TABLE tableName;
```

For example, if we want to remove the table ***instructor***, the command would be:

```
DROP TABLE instructor;
```

We are not allowed to drop a table that is referenced by foreign key constraint. If we try, as example, to remove the table Instructor we will see the following message:

```
Error Code: 1217. Cannot delete or update a parent row: a foreign key constraint fails
```

In this situation we need first drop the foreign constraint and then drop the table.

# REFERENCES

- **Books**

- Fundamentals of database systems – 6<sup>th</sup> edition
- Head First SQL - Lynn Beighley

- **Internet**

- <http://learndatamodeling.com/blog/creating-a-physical-data-model-using-erwin>
- <http://learndatamodeling.com/blog/erwin-tutorial/>
- <http://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561>