# Exercises: Introduction to SQL

This document defines the class assignments from the ["Databases" Course @ Software University](https://softuni.bg/trainings/1168/Databases-Jun-2015).

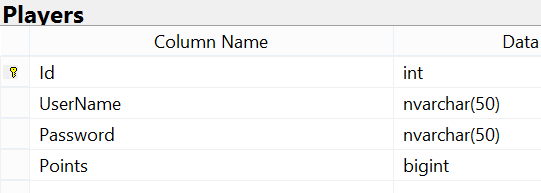
## Create initial architecture for massive multiplayer role-play game

The interview has gone well and you are now in charge for the back-end architecture of a massive multiplayer role-play game. Your first task is to set the initial database structure using the knowledge so far. The game has players which consist of username, email, password and points. Each of them has several planets. A planet is placed in a 3D-Coordinate system (x, y, z) and have resources from predefined types (crystal and metal). There are some buildings in-game. A metal mine, mineral mine and a fusion reactor. Each building can have up to 10 levels that cost some of a predefined resource (crystal and metal). The first level of every building cost 1000 metal and 500 crystal. The metal mine multiplier per level is 1.2, the crystal mine is 1.5 and the fusion reactor’s – 2.0. Each player’s planet has initially all buildings on level 0.

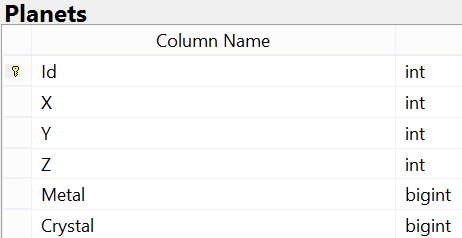
### Task 1. Design main entities

Given the schema above it’s getting clear that there are three main entity types. A **player**, a **planet** and a **building.**

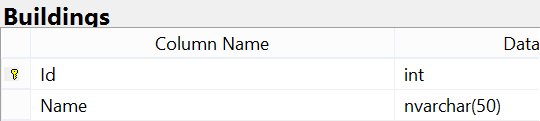
* Design basic schema without any relations
  + **A player** has some identity (**Id**) column, **username** which is textfield, so we will use nvarchar to avoid enconding problems and **password** which is also textfield. It might be a hash, but it does depend on your application and not on your database layer. Finally it has **points** which should be some integer type. As there is not stated a maximum of points, we will consider using BIGINT.



* + **A planet** also has identity (**Id**) column. It also belongs to a 3D Coordinate system. The easiest way to declare this is to **have 3 separate integer columns for x, y and z.** It also has **metal and crystal.** Having in mind there is no maximum declared for planet to store we will use a **bigint** again.



* + **Finally a building**. It does define only one thing – its type/name. Some string (metal mine, crystal mine etc…)

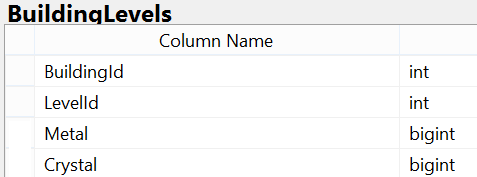


### Task 2. Define basic relations

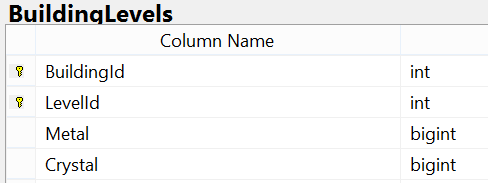
* The simplest relation here is the **player -> planets relation**. One player **can have one or more planets**. Respectively a **planet is owned by only one player**. This means you should add one more column in Planets table. **A PlayerId**, which should be of same datatype (**int**) as Player’s Identity column. This is required, **because in order a relationship to happen, a foreign key constraint should be made** – Planets.PlayerId pointing to Players.Id. It only can happen **if both columns are of same type**

### Task 3. Establish dependencies and advanced relations.

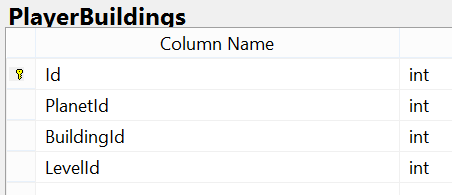
* The task is set and it’s clear. Your buildings should be expanded on levels. Each building can have up to TEN levels. You already have Buildings table but **you cannot expand it in order to achieve it, because each row is different building**. It also will **be totally wrong if you make column for each level**. So you have one option to **extend** the table with one more table which will keep track of each building’s level’s cost. Let’s say **BuildingLevels**. It will **relate** to a row in the buildings table (**BuildingId** /int/), will declare **Level** (int) and **declare cost in crystal and metal**. Since there’s an **exponential** progression, if levels continue expanding you **might hit the integer’s maxvalue**. Declare them as **BIGINT**.



* Without any **constraints** the table has some leaks. One can add **duplicate** rows for a pair BuildingId – LevelId, which will result **ambiguity** in your application. Which cost is the relevant one. E.g. the set [1, 2, 1200, 800] or the set [1, 2, 1300, 1000]? We can select both BuildingId and LevelId and declare a **composite primary key**

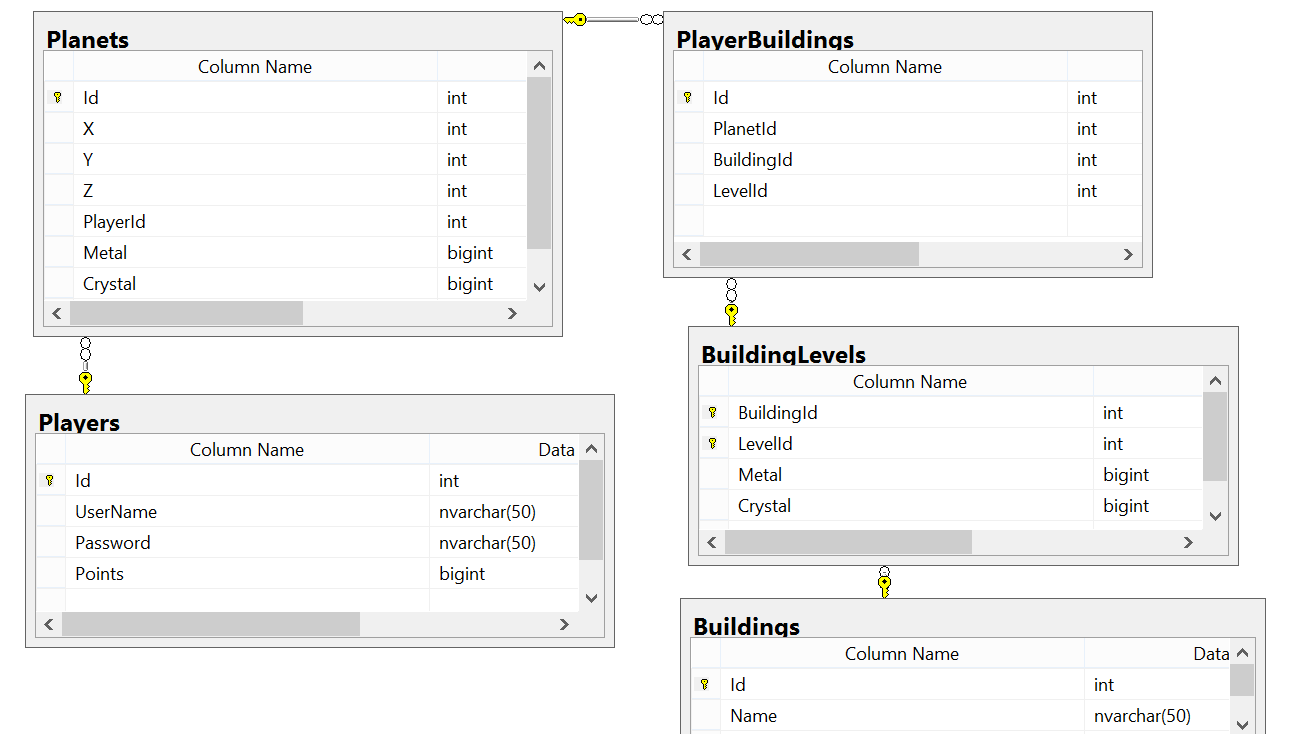
****

* One more step left before we reach **the final touches** of modeling our very database. On each player’s planet there can be each building on certain level. Let’s say **on planet\_x we have metal mine on level 7, crystal mine on level 3 and fusion reactor on level 10**. As we already have table for planets, we store planet\_x as a single row with identity (Id). Same rules apply for the building. And after the last changes (BuildingLevels) we have a declaration for **LevelId** too. So we need to create a table only of relational columns. Let’s say **PlayerBuildings** (or **PlanetBuildings**). **PlanetId** (int) referring to Planets.Id and a composite **foreign key (BuildingId, LevelId)** referring the **composite primary key in BuildingLevels (BuildingId, LevelId)**

****

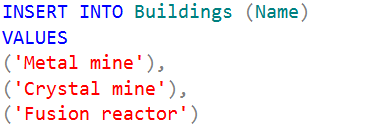
### Task 4. Create database diagram

* Finally we need to see an overall diagram. It should be something like this

****

### Task 5. Populate buildings

* It’s time to seed the predefined building types. On Id 1 place Metal mine, on Id 2 – Crystal mine and on Id 3 – fusion reactor

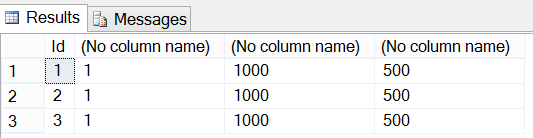


### Task 6. Populate levels

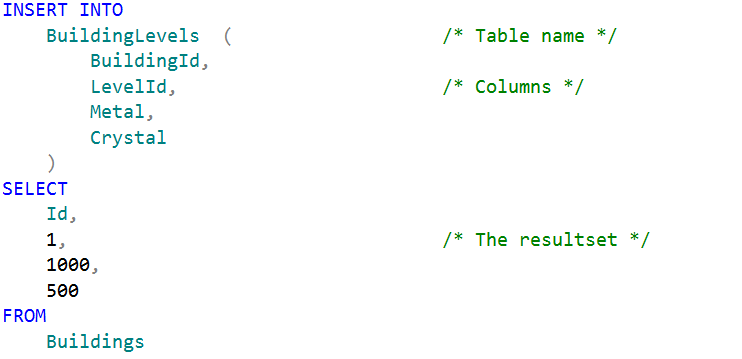
* After we have buildings with the appropriate Id’s, now we need to seed the predefined ten levels per building. It’s said that each buildings first level costs the same, so we can start from there. We will use an insert with a result of a datagrid from a select statement. Executing the following extraction



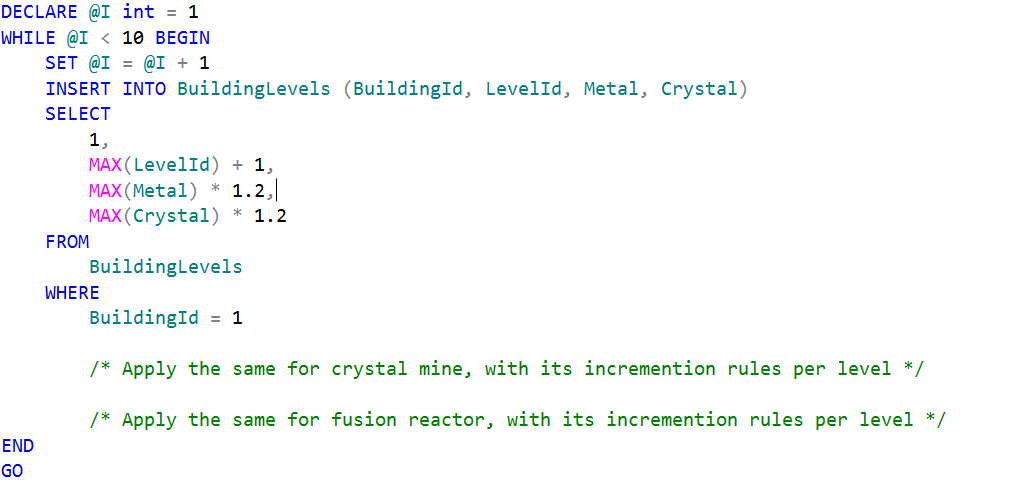
* Results into a resultset of rows as many as the building types are, returning each of their Id’s because the Id column is selected explicitly in the query, and the other columns are constants (the level, the metal cost and the crystal cost). Since we have only two buildings the grid is as following



* These 4 columns perfectly match the requirements of an insert statement in BuildingLevels table. A buildingId, a levelId, metal cost and crystal cost. So we just need to insert the result of the extraction by doing



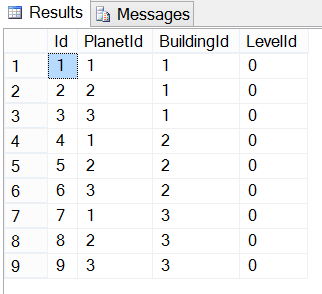
* Now we need the other 9 levels per building. There are many ways to achieve it. We will try to automate it with some straightforward way. We need to extract the maximum level so far (now it’s 1), the maximum metal so far (1000) and the maximum crystal so far (500) per building. Then to increase the level by 1, to increase the metal and crystal 1.2 times and repeat the operation. After we insert this and repeat the operation, the maximum level will be 2, the maximum metal will be 1200 and the maximum crystal will be 600. We can perform this over and over until reaching level 10. Fortunately the RDBMS supports loops. So we can loop that 9 times with a while loop. First we declare variable with value 1. Then we loop on condition this variable to be less than 10. Increment our variable and perform our insert.



* HINT: This could be simplified by declaring the modifier in relation with the building so only one select statement to be made

### Task 7. Populate player’s buildings with initial level 0

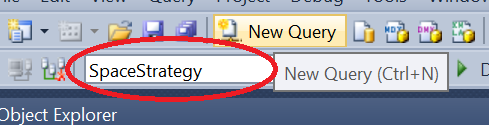
* Consider using Cartesian product in order to populate the relational table with only one INSERT query. The final result after populating should be something like the screenshot below



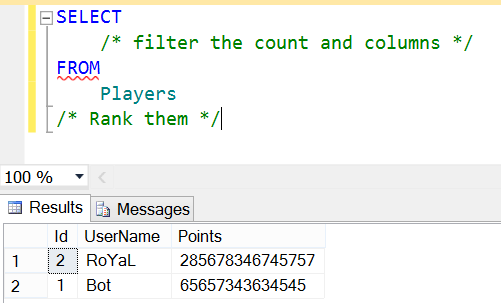
## Show game information

### Task 1. Show player ranking (first 100 players) without showing any sensitive information

* Query the players table

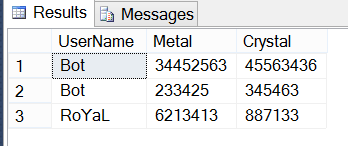


* Make a projection



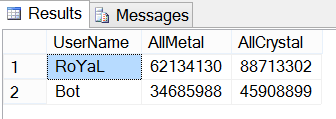
### Task 2. Show planets resource information for first three players

* Consider using **JOINS and/or NESTED SELECTS**



### Task 3. Rank players by all their resources and show them

* Consider using:
  + Aggregate functions
  + Aliases
  + Group condition



### Task 4. Extract information for each player’s building levels

* Consider using **multiple joins** and aliases
* See the example resultset below

