# **COMP-1701 - Transferring Data to Databases**

## **Data Science & Machine Learning (DSML) - RRC Polytech**

Module D.1 - Database Build - CREATE TABLE with 2 COLUMNs

Now that you've learnt how to CREATE a DATABASE, and install a GUI, let's now learn how to run a CREATE TABLE and add in one COLUMN to the DATABASE.

From the GUI, click on your ***FL*\_*STUDENTID*\_boxstore.sql** database and open your previous working SQL script, as well as your ***FL\_STUDENTID*\_cheatsheet.sql**

**CREATE TABLE syntax (w/ DROP TABLE)**

When developing a TABLE schema, you will eventually end up with a TABLE structure and data INSERTed into the TABLE.

Typically, these table sections start with a DROP TABLE and thereafter a CREATE TABLE, so everytime you run the script, you are left with a table that is ready for data.

DROP TABLE IF EXISTS table\_name;  
CREATE TABLE IF NOT EXISTS table\_name (  
 colPK INTtype AUTO\_INCREMENT   
 , col2 DATAtype(size) NULL DEFAULT value  
 ...  
 , colN DATAtype(size) NULL  
 , colFKs INTtype NULL  
 , p\_id\_user INTtype -- logged in user p\_id  
 , date\_mod DATETIME DEFAULT NOW()  
 , active BIT DEFAULT 1 -- 0 hide  
   
 , CONSTRAINT table\_name\_\_\_PK PRIMARY KEY(colPK)  
 , CONSTRAINT table\_name\_\_\_UK UNIQUE(col2,...colN)  
  
);

explained:

* **table­\_name** should be singular, so they don't have an 's' at the end of it, unnecessary, you will see many databases using plural forms
* tables must have at the very least one column, however relational database tables have at least 2 columns
* here are the attributes to the main columns in a TABLE:
* **PRIMARY KEY** (**PK**) column, **INT**EGER type, that **AUTO\_INCREMENT**s, where the first row entered will auto-number itself starting at 1, then next row a 2, and so on
* a specific **PRIMARY KEY** **CONSTRAINT** line is defined after all of the columns and their attributes (datatype, nullability, default, comment) have been entered
* all tables will have only 1 column as the **PRIMARY KEY** and is typically a *SURROGATE* key (which means it is not a human derived data column, just an **AUTO\_INCREMENT**ing column where its values would later be added to other tables as a FOREIGN KEY)
* An older option, was to create a *COMPOSITE* key, which allowed for multiple columns to be combined as a **PRIMARY KEY** for the **TABLE**
* This method has been deprecated, mainly due to its lack of expansion capabilities, far easier to send 1 PK column to another table as a FK, rather than sending 2 or more PKs to that other table
* The reason they used this, was to ensure the values for those PK combinations where UNIQUE…however, we can develop UNIQUE CONSTRAINTs to handle this need, and keep a simple INT value as the PK
* **UNIQUE** key (UK) column(s), of any type, that makes the row UNIQUE from other rows in the table
* it could be 2 or more UNIQUE columns that make the row UNIQUE to other rows in the TABLE
* EVERY TABLE SHOULD HAVE A:
* SURROGATE AUTO\_INCREMENTing PRIMARY KEY
* UNIQUE key (UK) which is the data in the table that should never duplicate, making the justification for why the row is created in the first place
* *additional columns*, that are supporting attributes for that **UNIQUE** column or columns
* *datatype* and *size* define the exact SQL Data Type being used by the field, we mostly deal in 'strings', numbers, 'date', boolean/bit (TRUE or FALSE)
* **NULL** simply means that no value exists for the column's row
* **DEFAULT** is another attribute that can be added after the NULL, if you don't want it to be NULL when inserting a row. **DEFAULT ''** where '' (2 single quotes [**'**] ) means an empty string, which to note is different from the value **NULL**
* **NULL** does not equal **''** (an empty string)or in **SQL: NULL<>'' or NULL!=''** …it can only be applied to a field and can be WHERE filtered with **column\_name IS NULL** or **column\_name IS NOT NULL**

Regarding the **NULL** attribute, *during build phase*, best to **allow NULLs** in the CREATE TABLE columns, even if the column requires a value. When adding columns to an existing table with existing data(rows), the CREATE TABLE only adds columns to the table structure, as it has no row values yet. Specifying **NOT NULL** when on a column will throw an error, as the column will require data within. The CREATE TABLE default is set to NULL.

BOXSTORE: Your course project

As mentioned, the concept for this course, is to build a database, based on your own big box store, like a mix between a Memory Express and Best Buy. Your instructor will be the owner, and you will be the employee. You will also need customers, so as many employees working for such a store will be customers as well, we will just create one TABLE called **people** with one field called **full\_name**.

If we have a **boxstore**, we will need **people** to work there and the same but mostly other **people** to buy things.

So we will need a **TABLE** to store this information. ***full\_name*** will just use a **VARCHAR**(**100**) field to start, which is a variable length string, with **100** characters in length. And the ***full\_name*** field must be entered, so it can't be nothing, and later we may alter the table to be **NOT NULL**.

We are also going to add an INTEGER with this ROW. It will AUTO\_INCREMENT, so when adding data, it will on its own start at 1, the next row will be 2, and so on. This is our table's PRIMARY KEY.

-- syntax for the people table  
--  
-- DROP TABLE *IF EXISTS* people;  
CREATE TABLE *IF NOT EXISTS* people (  
 p\_id INT(11) AUTO\_INCREMENT  
 , full\_name VARCHAR(100) NULL  
 ­­­, CONSTRAINT people\_\_\_PK PRIMARY KEY(p\_id)  
);

-- CONSTRAINT for UNIQUE, not yet defined, as  
-- people can have the same names, other columns  
-- will be needed to ensure the same person is   
-- not being duplicated  
  
-- view your people table structure mods  
DESCRIBE people;  
  
-- verify your table query works  
SELECT p\_id, full\_name   
FROM people   
WHERE *1=1*; -- returns 0 rows, contains no data yet

**INSERT INTO … VALUES syntax**

We need to add data into a table, so we can perform this operation with an INSERT INTO/VALUES statement:

-- insert syntax  
INSERT INTO table\_name (column2 ... , columnN)  
VALUES (value2 ... , valueN);

explained:

* number of columns you are updating, must have a corresponding number of values in the VALUES (…) portion of the command
* note, I did not enter a column1, the column one will be an AUTO\_INCREMENT key, that does not need to be managed by the human touch. It is the PRIMARY KEY for our table, and will always be unique for each row, starting at 1, then 2, … and so on.

Insert the first 2 rows into our people table

Here we will manually insert the first 2 records into our people table, this will require 2 insert statements, you will insert the owner, which is your 'Instructors Name' and 'Your Name', and remember the PRIMARY KEY(p\_id) takes care of itself:

-- replace Instructor Name and Your Name with   
-- respective values  
  
INSERT INTO people (full\_name) VALUES ('Instructor Name');  
INSERT INTO people (full\_name) VALUES ('Your Name');  
  
  
-- running it as a BULK INSERT  
  
INSERT INTO people (full\_name) VALUES ('Instructor Name')   
 , ('Your Name');

Run a SELECT to verify the data rows were inserted.

-- verify your query works (returns 2 rows)  
SELECT p\_id, full\_name   
FROM people   
WHERE *1=1*;

…maybe we should talk about the SELECT:

SELECT FROM syntax

The syntax is simple you SELECT comma-delimited columns FROM table, and you may add in a WHERE to filter the rows or a LIMIT to

SELECT *DISTINCT* column1, column2 ... , columnN   
FROM table\_name   
WHERE filter\_condition *AND|OR second\_condition*  
ORDER BY column2 DESC, ...  
LIMIT *rowoffset#*, *rowsperpage#(rpp)*;

explained:

* the order of command parts is specific **SELECT** to **FROM** then **WHERE** then **ORDER BY** then **LIMIT**
* you only select the columns you need or are asked to present
* if you want to show all columns, use a **SELECT** \*, which is typically used during development and analysis only
* columns in the **SELECT** are comma-delimited
* SELECT col1, col2, col3 … , colN
* **WHERE** contains a condition that must evaluate to TRUE, as discussed previously
* In examples you will see the use of 1=1, which of course evaluates to TRUE, it is usually used for a comparison and it just has the WHERE placeholder already in my code to change the condition to something else
* **ORDER BY**, allows you to order by a column in ascending order (by default, ie: alphabetic order and numeric orders, smaller first) or DESCending order (specified by **DESC**, reverse order, bigger numbers/dates first)
* there is an **ASC** for ascending, though it never needs to be specified, just the column\_name and it will sort ascending
* **LIMIT** is used for paging your results, ***if no ORDER BY is specified***, will use the ***current table's default row order***:
* **LIMIT 2;** displays beginning 2 rows
* **LIMIT 0,10;** displays beginning 10 rows
* **LIMIT 10,10;** displays beginning at row 10, 10 rows
* **LIMIT 20,10;** displays beginning at row 20, 10 rows
* *for paging*, to calculate the row offset:   
   (***page#*** x **rpp#**) - **rpp#** = **ro#**  
   ie: Page 4 with 10 rows: (4 x 10)-10=30  
   for LIMIT 30,10
* **DISTINCT** is a way to group data into one of each item, more on this later.

-- get all rows  
--  
SELECT p\_id, full\_name  
FROM people  
WHERE 1=1; -- condition: 1=1 evals to TRUE  
 -- if a filter is being added  
 -- later in the query, just   
 -- put WHERE 1=1 as placeholder  
  
-- get all columns, an asterisk means all   
-- columns, though it is never wise to use   
-- asterisks, most of the time you need only a  
-- few columns to travel from the database  
-- over the network, to the web or app pages  
-- calling it, this fine for quick testing  
-- only!!  
--  
SELECT \*   
FROM people;

-- get first record  
SELECT p\_id, full\_name  
FROM people  
WHERE p\_id=1;  
  
-- get second record  
SELECT p\_id, full\_name  
FROM people  
WHERE p\_id=2;  
  
-- show first row, if 2, shows both rows  
SELECT p\_id, full\_name  
FROM people  
LIMIT 1;  
  
-- would show second record only  
SELECT p\_id, full\_name  
FROM people  
LIMIT 1,1;   
  
  
-- get your name record (replace Your Name)  
SELECT p\_id, full\_name  
FROM people  
WHERE full\_name='Your Name';

-- get the instructor name record (replace   
-- Instructor Name)  
SELECT p\_id, full\_name  
FROM people  
WHERE full\_name='Instructor Name';

Typically, data storage within a database is done, so you can easily call up rows, in this case, it is trickier to call the records by the First Name or Last Name, we would need to do:

-- get your last name (replace only the Name in   
-- the last name here, leave in the % and space)  
  
SELECT p\_id, full\_name  
FROM people  
WHERE full\_name LIKE '% Name';

The **LIKE**'s **%** is a wildcard, that **means zero to many characters**. I added a space, so it will filter out the last name, assuming the format of *'FirstName LastName'* is used throughout. The % in the above example, will pull a record called ' Name' (if it had a space before the Name text). A LIKE query can perform poorly…always pay attention to the execution times.

As mentioned, **full\_name** fields like this, are not typically created in this fashion, they would be split up and stored into 2 columns, making it easier on the WHERE condition filtering.

Next, we will be learning to ALTER a TABLE and Parse (split up) some data.

## 