|  |  |
| --- | --- |
| -  Sindy Saintclair  Tuesday, January 18, 2022  Lesson 4 – Creating Tables | |
| **Learning Objectives and Questions** | **Notes and Answers** |
| **Tables** | Now that you have worked somewhat with tables, you will explore more in-depth what tables are and what you can do with them.  WORKSHOP  ALL KEYED-UP KEYS AND CONSTRAINTS IN SQL  Keys – an identifier that helps track and relate your data  *Natural vs Surrogate Keys*   * Natural use an existing column and can only be done if it follows the constraints of being unique and filled. For instance, a driver’s license, serial numbers. * Surrogate key are keys you create and are often made in auto-increment. They can take up less storage space. For instance, Patient 1, 2, 3…   *Primary Key*   * Column(s) that uniquely identify each row in a table * Constraints include uniqueness and no missing values   *Composite Primary key*   * use more than one column’s values together to make a unique key   *Generating Surrogate Keys*   * auto\_increment * serial/bigserial   *Foreign Key*   * connects values in tables so that related tables are properly pieced together * values in another table’s primary key * constraints: must be a value that already exists in the primary key * can be empty * can be duplicated (not unique)   Dealing with NULL data  *Null values*   * denotes no data present * user did not enter it, or it is unknown * queries don’t come back empty; they return NULL   *IS NULL vs. IS NOT NULL*   * Is Null is used as a criterion in a WHERE clause to find columns without data * Is Not Null is used as a criterion in a WHERE clause to find columns with data   What is a Constraint?   * rules placed upon data entry for a table   NOT NULL   * does not allow NULL values in a column   UNIQUE   * values in a column have no overlap * but does allow multiple NULL values * isn’t used for reference like a primary key   CHECK   * will determine whether data falls in the values you want * useful if only certain ranges are valid * uses Boolean operators * For instance, weight – can eliminate typos if you put minimum / maximum caps on values and this saves time in data cleaning   ON DELETE CASCADE - when you delete a row in one table, it will automatically delete rows in other tables that were based on the information  Indexing  Create Index   * Improve query speeds when used sparingly * What to index * Foreign keys * Columns you use frequently in where statements |
| **Columns** | Defining the structure of your table is essential to storing your data in an efficient and meaningful way. *Columns* are the most essential part of the table; this is where the actual data will live. You will define these columns (also called fields), with a datatype that you deem appropriate for the data you wish to store. A good choice for a name, for instance, would be a VARCHAR(50), with the number in the parentheses indicating the maximum number of characters that any given name may have. A good choice for storing a date of birth might be DATE, while INTEGER may be the best choice for the model year of a vehicle. Be sure to choose appropriately here, as changing any of these types could be a cumbersome undertaking if a significant amount of data has been stored in the table. |
| **Nullability** | To review what you have learned so far, Null is a value that is undefined. Because of this, attempts to compare values to afield that contains a Null value will fail. The only operation that works on this value is the binary evaluation of IS NULL or IS NOT NULL. As applied to tables, this is whether a column in your table can contain the value Null or not. The keywords are either Null , or NOT NULL appended to the datatype that follows the column name when you create the table. For instance: FirstName varchar(50) NOT NULL would indicate that the FirstName column should never be undefined. A practical application of using NULLs would be something like a voluntary survey – perhaps the applicant chose not to answer ethnicity. So you could store Null in that column when you save the data to indicate that there was no selection.  The structure of a table is critical. You will be working mainly with the rows and columns of the tables in a database. Tables can be related to Excel spreadsheets. The columns define the data needed, and the rows list out the data for each object in the database. Below is a MS Excel spreadsheet with information about specific animals in a pet store. Tables in a SQL database will be structured very similarly to the below MS Excel example:  Excel table. Table with the headings name, species, breed, and age. Under name the entries are Henry, Rupert, Churchill, and Pepper. Under species the entries are dog, pig, cat, and dog. Under breed the entries are bulldog, mini, redpoint Siamese, mix, and under age the entries are 2, 1, 3 and 8.  Figure 4-1: Excel Table  The columns define the data needed for each animal, and the rows provide that data for each animal. Great! Now that you understand the basic structure of a SQL table, you will learn how to create them!  Now that you understand the basic structure of a table, you will build one! You are going to be using the CREATE TABLE keywords to create a table in your existing database. Below is the syntax:  CREATE TABLE table\_name (  column1\_name datatype,  column2\_name datatype,  column3\_name datatype,  column4\_name datatype,  ...  )  The column parameters will define the column names in the table. The datatype will specify the type of data each column can hold. Below is a list of common data types used:   * CHAR(size) * VARCHAR(size) * NVARCHAR(size) * INTEGER * DATETIME * NUMERIC   Before you create your table in MySQL, take a look at a basic CREATE TABLE query:  CREATE TABLE AppUsers (  AppUserID INTEGER,  FirstName VARCHAR(50),  LastName VARCHAR(50),  SignUpDate DATETIME  )  The above query will create a table with four columns, but with no data, as you can see below:  Basic table. File directory in Sakila. The folder Sakila has a folder called tables within it. Inside that folder are others titled actor, address, appusers,  Figure 4-2: Basic Table  The above table looks great, but a few things are missing that are used very frequently when creating a table. You will explore something called *Constraints* and how they can help you define your data. |
| **Constraints** | Constraints are used to specify rules for data in a table that will limit the type of data that can go within a table or column. This will make sure that the data is reliable and accurate and if any of the constraints are violated, the action or query will be aborted. You can apply a constraint to a column or a table. Read on to explore commonly used constraints.  *NOT NULL*  NOT NULL will force that column not to accept null values. By default, columns can hold NULL values. NOT NULL will prevent that column from having a null value. This is useful when you are gathering data from a user (like signing up for a website), and you need every input field to contain some data.  Below is how it will look in your query:  CREATE TABLE AppUsers1 (  AppUserID INTEGER,  FirstName VARCHAR(50) NOT NULL,  LastName VARCHAR(50) NOT NULL,  SignUpDate DATETIME NOT NULL)  Above, you are adding the NOT NULL constraint to the FirstName, LastName and SignUpDate columns. This will ensure that each of those rows in the AppUsers table will have data.  UNIQUE  *Unique* ensures that some column or combination of columns is unique to each row in the table. It’s similar to a primary key, but a table may only have on Primary Key, whereas it might have several Unique Keys. An example here might be a column containing a social security number. The primary key might be id, but you might also want to monitor that the social security numbers that are being inputted are also unique to a particular customer. Since social security numbers SHOULD be unique, a failure to insert because of a violation of such a unique key could be a way to sound an alert.  Consider below:  CREATE TABLE AppUsers2 (  AppUserID INTEGER UNIQUE,  FirstName VARCHAR(50) NOT NULL,  LastName VARCHAR(50) NOT NULL,  SignUpDate DATETIME NOT NULL)  What are you doing above is making sure that the AppUserID is going to be unique for each user. You would not want to have two users with the same id. That could prove to be confusing and could potentially mess up your data. |
| **PRIMARY KEY** | PRIMARY KEY is a combination of NOT NULL and UNIQUE, so that every row in the table is unique and not null. This will enabled you, when working in other tables, to identify that the data is related to the original table. For example, a customer table may have an id column that is a primary key. You could then use that customer id in a table that lists purchases, to relate a particular purchase to an individual customer account. **Almost all tables should have a primary key.** Without a primary key, there is no way to identify each row uniquely.  All in all, Primary Key is a constraint that defines a column in a row so that it is not null and unique for every individual row. The SQL syntax for Primary Key is shown below:  Consider below:  CREATE TABLE AppUsers3 (  AppUserID INTEGER PRIMARY KEY,  FirstName VARCHAR(50) NOT NULL,  LastName VARCHAR(50) NOT NULL,  SignUpDate DATETIME NOT NULL)  What you have done above is replace UNIQUE with PRIMARY KEY. This will now ensure that the AppUserID is unique to every row and also is required to have a value.  *Autoincrement*  AUTOINCREMENT will automatically generate a unique number when a new record is inserted into a table. Often, the PRIMARY KEY field will also have AUTOINCREMENT since the primary key is the unique id for each row. It is handy to be able to generate the unique id for each row automatically.  CREATE TABLE AppUsers4 (  AppUserID INTEGER PRIMARY KEY AUTO\_INCREMENT,  FirstName VARCHAR(50) NOT NULL,  LastName VARCHAR(50) NOT NULL,  SignUpDate DATETIME NOT NULL)  Above, your query will now automatically produce an AppUserID when a row is generated.  *Types of Primary Keys*  When using Primary keys, there are two different types to keep in mind. Those types of *Natural* and *Surrogate*. Natural is when the Primary Key is assigned to a piece of data that already exists in the record and will always be unique for each record. an example of this could be a Social Security Number. If you had employees in your database, every employee would have a unique SSN. A Surrogate is a Primary Key that doesn’t have anything to do with the actual data. Up until this point, you have been using a Surrogate Primary Key by assigning an ID number to your customers or users. Surrogate Primary Key by assigning an ID number to your customers or users. Surrogate Primary Keys are becoming more popular these days mainly because the cost of having a database with a significant amount of data is not nearly as expensive as it used to be. In the past, companies would save money by using a Natural key. Problems would arise when that Social Security Number of an employee changes, because that can and has happened. That could potentially break anything that is linked to that employee by his/her SSN. To wrap up, Surrogate keys are used more and more because that data has nothing to do with the actual record except to give it a unique piece of data. |
| **FOREIGN KEY** | FOREIGN KEY is a relation between tables that will restrict the insertion of data in one table, depending on the contents of a second table. The FOREIGN KEY refers to the PRIMARY KEY in a separate table.  Say you have a Customer Table that has a column for a state of residence. You might have Foreign Key to a table with all of the states and their abbreviations in it, to ensure that any entry into the Customer Table has a valid state entry. Consider the following tables:  Below, you have a snippet of a table called “states” that has the state name and their abbreviation:   |  |  | | --- | --- | | **State** | **Abbreviation** | | Alabama | AL | | Alaska | AK | | Arizona | AZ | | Arkansas | AR | | California | CA | | Colorado | CO | | Connecticut | CT |   And below, you have a snippet of a table called “customers” that contains their info including the state in which they live.   |  |  |  | | --- | --- | --- | | **FirstName** | **LastName** | **State** | | Alex | Smith | MA | | Ryan | Williams | AL | | Bob | Jones | RI |   Below is how you would use FOREIGN KEY when you are creating the customers table:  CREATE TABLE customerExample(  FirstName NVARCHAR(40),  LastName NVARCHAR(30),  State NVARCHAR(2),  FOREIGN KEY(State) REFERENCES states(Abbreviation)  );  Above, you are using FOREIGN KEY to call out the column “State” in your customers table, so it references the Abbreviation column in the states table. |
| **CHECK** | CHECK is an expression that you create that will check each entry into the table for validation. If the entry fails the constraint, then the update or insert will fail. An example here would be a check constraint that would validate a phone or zip has the right number of digits, and that they were in fact digits instead of text. Here is an example:  CREATE TABLE customerExample(  FirstName NVARCHAR(40),  LastName NVARCHAR(30),  State NVARCHAR(10),  PostalCode INTEGER,  FOREIGN KEY(State) REFERENCES states(Abbreviation),  CHECK (length(PostalCode) = 5)  );  Above, you see that CHECK is used to ensure that PostalCode is equal to 5. To do this, you needed to use length(), a function that will return the length of whatever is inside the parenthesis. Say you are inserting a new customer into this table and forget a number in the postal code. You will get the below error:  Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails  *Default*  DEFAULT defines default values that you can assign to a column, in the case where an insert statement does not include a value for that column. A common use for this scenario is an Inserted column that you might have in your table, to indicate when the record was written. You might set the default constraint to the datetime() function, which would populate the date and time (local time, not UTC) of when the record was created in the table. An example would look like this:  CREATE TABLE customerExample(  FirstName NVARCHAR(40),  LastName NVARCHAR(30),  State NVARCHAR(10),  PostalCode INTEGER,  SignUpDate datetime DEFAULT current\_timestamp,  FOREIGN KEY(State) REFERENCES states(Abbreviation),  CHECK (length(PostalCode) = 5)  );  *Triggers*  Triggers are objects that you can append to your table that will run code based on the operation that is performed on a table. A common usage would be an ‘after’ trigger, which would be executed ‘after’ an insert, update, or delete. You could use a trigger of this sort to capture the values changing in the table and send them to a log table for historical usage, to see how the values in the table have changed in the past. Triggers are usually found on smaller databases and not so much on larger ones, as the overhead and resource contention caused by triggers fired off continually can adversely impact performance to an extreme degree. |
| **CHECK** | CHECK is an expression that you create that will check each entry into the table for validation. If the entry fails the constraint, then the update or insert will fail. An example here would be a check constraint that would validate a phone or zip has the right number of digits, and that they were in fact digits instead of text. Here is an example:  CREATE TABLE customerExample(  FirstName NVARCHAR(40),  LastName NVARCHAR(30),  State NVARCHAR(10),  PostalCode INTEGER,  FOREIGN KEY(State) REFERENCES states(Abbreviation),  CHECK (length(PostalCode) = 5)  );  Above, you see that CHECK is used to ensure that PostalCode is equal to 5. To do this, you needed to use length(), a function that will return the length of whatever is inside the parenthesis. Say you are inserting a new customer into this table and forget a number in the postal code. You will get the below error:  Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails |
| **INSERT** | Now that you understand how to create a table, you can use INSERT, which you learned previously in this module.  Run the below query:  create table actorExample(  actor\_id smallint(5) unsigned,  first\_name varchar(45),  last\_name varchar(45),  last\_update timestamp)  Now you can insert data into this table:  insert into actorExample  values (200, "Jamie", "Thomas", "2020-01-23 12:16:34")  And if you look at the data in the actorExample table, you should see the below output:  200 Jamie Thomas 2020-01-23 12:16:34  Awesome! But what if you want to insert data into this table that already exists in your database? You can do that!  *Insert All Columns*  If you wanted to insert all columns from one table to another, you would do this in general terms:  INSERT INTO table2  SELECT \* FROM table1  WHERE condition;  *Insert Some Columns*  If you wanted to insert only some columns, you would do this in general terms:  INSERT INTO table2 (column1, column2, column3, ...)  SELECT column1, column2, column3, ...  FROM table1;  You could also add a where statement to this:  INSERT INTO table2 (column1, column2, column3, ...)  SELECT column1, column2, column3, ...  FROM table1  WHERE condition;  Notice the difference from the INSERT queries you have run in the past. Instead of having the keyword VALUES, you are now using SELECT to select which columns should be inserted into your table.  *Try it Yourself!*  Try inserting the data from the actor table into the actorExample table. Give it a try first, and if you get stuck, the answer is located below. If you get the following message with a green check mark next to it, then you did it correctly!  3 40 12:23:25 INSERT INTO actorExample (actor\_id, first\_name, last\_name, last\_update)  SELECT actor\_id, first\_name, last\_name, last\_update  FROM actor 204 row(s) affected  Records: 204 Duplicates: 0 Warnings: 0 0.063 sec  If you do not see the same as above, the correct code is located below:  INSERT INTO actorExample (actor\_id, first\_name, last\_name, last\_update)  SELECT actor\_id, first\_name, last\_name, last\_update  FROM actor; |
| **DROPPING A TABLE** | Now that you know how to create a table and insert data into it, you will learn how to delete the table. The syntax is relatively straightforward:  DROP TABLE table\_name;  Try out this syntax by deleting the table actorExample  you created earlier:  drop table actorExample  If you, did it correctly, you should see that wonderful green checkmark.  Great work! Let’s keep moving! |
| **Views** | A View is a virtual table based on the result of a SQL statement. It contains rows and columns, just like a real table, but when a view is created, a table is NOT created; you are just showing the output of the selected data. The fields in the View are from one or more real tables that exist in the database. You can use SQL functions, WHERE, and JOIN statements in a view to be able to present the data as if it were coming from one table.  *CREATE VIEW Syntax*  Here is the generalized syntax for creating views:  CREATE VIEW view\_name AS  SELECT column1, column2, ...  FROM table\_name  WHERE condition;  And now you can try actually creating a view:  create view CurrentCustomers as  select customer\_id, first\_name, last\_name  from customer  where active = 1  Above, you are creating a view to see all customer IDs, first name and last name who are active. When the above query is executed, the green checkmark will appear, so you know the view was created correctly. Next, to see the view, run the following query:  SELECT \* FROM CurrentCustomers;  You will now see the CustomerId, FirstName, and LastName of customers that are active. Creating views are useful because it means you don’t need to run complicated queries many times. You can run a complicated query and set it to a view that you can then select easily.  *Create View with Join*  Creating a View using a Join will look very similar to what you have done in the past with Joins. Below is a Join query you ran in previous lessons:  SELECT first\_name, last\_name, film\_id  FROM sakila.actor  INNER JOIN sakila.film\_actor  ON sakila.actor.actor\_id = sakila.film\_actor.actor\_id  All you would need to do if you wanted to create a view with the above query is add the Create View statement on the first line:  SELECT first\_name, last\_name, film\_id  FROM sakila.actor  INNER JOIN sakila.film\_actor  ON sakila.actor.actor\_id = sakila.film\_actor.actor\_id  And if you run:  SELECT \* FROM ActorFilms;  You will easily be able to see the three columns selected in the CREATE VIEW query.  *Drop View*  Now, if you didn’t need your view anymore, you would use the following generalized syntax:  DROP VIEW [view\_name];  Practice by dropping the CurrentCustomers view you created:  drop view CurrentCustomers |