Python: Insert large CSV files to SQL Server

Owen Price - linkedin.com/in/owenhprice

1. Get the list of files

os to get the list of csv files

pandas to read and prepare the csv files

pyodbc to interact with SQL Server

```python
import os
import pandas as pd
import pyodbc

# Set the folder path for the CSV files
folder_path = 'data/'

# Get a list of all CSV files in the folder
csv_files = [os.path.join(folder_path, f) \
             for f in os.listdir(folder_path) \
             if f.endswith('.csv’)]
```

This is known as a list comprehension.

It takes the form [action for item in list condition]

os.path.join(folder_path, f) joins the folder_path string to f as a path, where f is an item in the list of files returned by os.listdir(folder_path) that satisfy the condition f.endswith(‘.csv’)

2. Prepare a connection string

Define the various components of the connection string as separate variables

Your driver string may differ

```
# variables to configure the connection string
driver='{ODBC Driver 17 for SQL Server}'
server=‘server'
database=‘database'
trusted_connection='yes'

# pyodbc connection string
connection_string = f'DRIVER={driver};SERVER={server};'
connection_string += f'DATABASE={database};'
connection_string += f'TRUSTED_CONNECTION={trusted_connection}’
```

We can insert the variables into the structure of the connection string using f-strings. We simply wrap the variable name in curly-braces inside the string. The string must be immediately preceded by f.

3. Pre-define an INSERT

Defining the INSERT in this way is just one way to do this.

It could also be built dynamically if the file structures differed from file to file.

```
insert_sql = 'INSERT INTO [dbo].[tripdata]\
           ([Trip Duration]\
           ,[Start Time]\
           ,[Stop Time]\
           ,[Start Station ID]\
           ,[Start Station Name]\
           ,[Start Station Latitude]\
           ,[Start Station Longitude]\
           ,[End Station ID]\
           ,[End Station Name]\
           ,[End Station Latitude]\
           ,[End Station Longitude]\
           ,[Bike ID]\
           ,[User Type]\
           ,[Birth Year]\
           ,[Gender])\
     VALUES\
           (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)'
```

Each ? is a placeholder for a value that will be read from a column in the csv file

4. Connect and loop over files

We create a connection object, from which we can create a cursor object to interact with the database

```
conn = pyodbc.connect(connection_string)
cursor = conn.cursor()
cursor.fast_executemany = True

for csv_file in csv_files:

    # code to do something with each file

    conn.commit()

conn.close()
```

Set this to True for faster INSERTs

Now we're ready to loop through the files and load them into the SQL database

We'll commit the new data to the database at the end of processing each file

5. INSERT chunks to the database

This ensures empty values in the csv are properly inserted to the database as NULL

Instead of reading the entire file to memory, we read it in ‘chunks’

```
for csv_file in csv_files:

    for chunk in pd.read_csv(csv_file, chunksize=100000):

        chunk.replace({np.nan: None},inplace=True)

        data = [tuple(x) for x in chunk.values]

        try:
            cursor.executemany(insert_sql, data)

        except pyodbc.Error as e:
            # do something with the data that caused the error
            break

    conn.commit()

conn.close()
```

This list comprehension builds a list of tuples – rows - to insert to the table

cursor.executemany substitutes the values from the list of tuples into the ? placeholders in the insert statement, then issues batches of VALUES inserts to the server in fewer server round-trips