The Mass Insertion of SQL Server Records Algorithm

Mr Ashlin Darius Govindasamy
University of South Africa

November 9, 2022

Abstract

This paper presents a new algorithm for the mass insertion of records into a SQL Server database and it can handle any file extension which pandas supports. There is a problem with the current method of inserting records into a SQL Server database. We are only limited to .csv, .txt files in the SQL Server Management Studio Import Flat File Wizard. The algorithm is written in Python and it can handle any file extension which pandas supports. It is proven that the SQL Server Management Studio Import Flat File Wizard is 2.18 times faster than the algorithm but that is only for one file. When having n number of files and automation is required then my algorithm is the way to go. I own the *sqlserver* module on PyPI. It simplifies *PyODBC* module and it is a wrapper around the *PyODBC* module. In this paper I will show how to use the *sqlserver* module to insert records into a SQL Server database and also explain the algorithm to commit the records into the database.

Contents

Chapter 1

Approach

1.0.1 Algorithm

We have a file that contains our dataset in the file extension $fileExt$.

Our $fileExt$ can be any file extension that pandas supports.

Some examples of $fileExt$ are:

$$fileExt \subseteq \{csv, txt, xls, xlsx, json, xml\} \tag{1.1}$$

We convert our $fileExt$ to a *.csv* dataframe using the pandas library.

Now we got a dataframe which has n rows and m columns.

We define a $queryBuffer$ which is a *string* and we initialize it to an empty string.

The function $CreateCSVTable$ creates a table in the database with the name $tableName$ and the columns are the m columns names in the dataframe.

$$CreateCSVTable(csvfile) \tag{1.2}$$

The function $InsertCSVRecords$ inserts the records from the dataframe into the table $tableName$.

$$InsertCSVRecords(csvfile) \tag{1.3}$$

Let us elborate on the function $CreateCSVTable$.

We iterate through the columns in the dataframe and we append the column name and the data type to a tuple called *payload*.

Then we inject the tuple into the $queryBuffer$ and we append a comma to the $queryBuffer$ else if the column is the last column in the dataframe we append a closing bracket to the $queryBuffer$.

$$payload = \sum_{i=1}^{m} \begin{cases} ([col_i]varchar(max)), & \text{if } i \neq m \\ ([col_i]varchar(max)) & \text{if } i = m \end{cases} \tag{1.4}$$

First we check if the table exists in the database. If the table exists we drop the table.

Then we create the table with the name $tableName$ and the columns are the m columns names in the dataframe.

```
queryBuffer = IF EXISTS(SELECT * FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_NAME = '{tablename}' AND TABLE_SCHEMA = 'dbo')
BEGIN
    DROP TABLE [dbo].[{tablename}]
    CREATE TABLE [dbo].[{tablename}]
    (
        {payload}
    );
END
ELSE
    CREATE TABLE [dbo].[{tablename}]
    (
        {payload}
    );
END
```

Then we execute the $queryBuffer$ using the $executeQuery$ function.

$$executeQuery(queryBuffer) \tag{1.5}$$

It commits the changes to the database.

The only issue with this approach the SQL batch has to fit in the batch size limit.

$$65536bytes * NetworkPacketSize \tag{1.6}$$

Where $NetworkPacketSize$ is the maximum packet size of the network.

The default value of $NetworkPacketSize$ is 4096.

Therefore the maximum size of the SQL batch is $65536 * 4096 = 268,435,456bytes$.

$$268,435,456bytes = 256MB \tag{1.7}$$

I will doubt someone will have so many columns in a dataframe that may exceed the batch size limit.
But i wouldnt put it past someone like Meta to have a database table with thousands of columns in one table.

Let us elborate on the function $InsertCSVRecords$.

We iterate through the rows in the dataframe and we append the values in the row to a tuple called *payload*.

Then we inject the tuple into the $queryBuffer$ and we append a comma to the $queryBuffer$ else if the row is the last row in the dataframe we append a closing bracket to the $queryBuffer$.
But now we have to implement constraints into our algorithm.

Let us denote the constraints as C.
Where C is determined by the user to insert k records at a time in one query.

$$record ='' \tag{1.8}$$

We set record = ” to an empty string.

$$\sum_{i=1}^{n}\sum_{j=1}^{m} record+ = "'" + str(row[column]).replace("'","") + "'"+',' \tag{1.9}$$

We iterate through the rows and columns in the dataframe and we append the values in the row to a string called *record* but replacing the single quotes with a space since Python uses single quotes to denote a string.

$$record = record[:-1] \tag{1.10}$$

We remove the last comma in the *record* string.

$$payload = \sum_{i=1}^{n} \begin{cases} (record), & \text{if } i \neq n \\ (record) & \text{if } i = n \end{cases} \tag{1.11}$$

We append the *record* string to the *payload* tuple and we append a comma to the *payload* tuple else if the row is the last row in the dataframe we append a closing bracket to the *payload* tuple.

$$\text{iff } i \neq n \text{ and } i \mod C = 0 \tag{1.12}$$

We check if the row is not the last row in the dataframe and if the row is divisible by C.
If the condition is true we append a closing bracket to the *payload* tuple.

```
query = f'''
INSERT INTO [{tablename}]
VALUES
{payload[:-2]}
'''
executeQuery(query)
payload = ''
```

We inject the *payload* tuple into the $queryBuffer$ and we append a closing bracket to the $queryBuffer$.
Then we execute the $queryBuffer$ using the *executeQuery* function.
It commits the changes to the database.
We set *payload* = " to an empty string.

$$\text{iff len(payload)} > 0 \tag{1.13}$$

```
query = f'''
INSERT INTO [{tablename}]
VALUES
{payload[:-2]}
'''
executeQuery(query)
```

We check if the length of the *payload* tuple is greater than 0.
If the condition is true we inject the *payload* tuple into the $queryBuffer$ and we append a closing bracket to the $queryBuffer$.
Then we execute the $queryBuffer$ using the *executeQuery* function.
It commits the changes to the database.

That is the algorithm for the function $InsertCSVRecords$.

It is unsure what pyodbc query size limit is.
But i would assume it is the same as the SQL batch size limit.
In the sql server module i have created i have set the default value of C to 1000.

$$C = 1000 \tag{1.14}$$

I have injected millions of records into a database table using this algorithm.
Quite fast too.
Uncertain if it failed to insert any records.
As the safety i have implement is quite good.
In any case if it fails the algorithm will safely continue to insert the records.
But skips the records that failed the query limit.
And a query error will be in the stdout.

If i reach the query limit someday i will re-engineer the algorithm to insert the records in batches.
As per by finding the query limit by bruteforce.

Chapter 2

Usage

2.0.1 Using the module

Since the algorithm is written in Python, we can use it in any Python project.

We start by downloading the module from PyPI.

```
pip install sqlserver
```

We import the module into our project.

```
import sqlserver
```

Below i will show you the guide for version 1.0.16. In the future i will update this guide to the latest version so use the guide from the PyPI page or look at the link below.

GitHub PyPI

2.0.2 Installation for linux guys

1. Install a driver for your linux machine!
2. I recommend `FreeTDS`

2.0.2.1 Installing `FreeTDS`

1. Install pre-requisite packages

```
sudo apt-get install unixodbc unixodbc-dev freetds-dev freetds-bin tdsodbc
```

2. Point `odbcinst.ini` to the driver in `/etc/odbcinst.ini`

```
[FreeTDS]
Description = v0.91 with protocol v7.2
Driver = MYDRIVERPATH
```

where `MYDRIVERPATH` is the path of the `libtdsodbc.so` file
Hint! Look in the `/usr/lib/mylinuxdistro/odbc` folder! Will implement script in the future to install/automate this for linux solutions.

```
pip install sqlserver
```

2.0.3 Installation for windows guys

```
pip install sqlserver
```

2.0.3.0.1 Example of ConnectionString

```
DRIVERS = db.ReturnDrivers()
# output of drivers
['SQL Server', 'ODBC Driver 17 for SQL Server',
'SQL Server Native Client 11.0', 'SQL Server Native Client RDA 11.0'
, 'Microsoft Access Driver (*.mdb, *.accdb)',
'Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)',
 'Microsoft Access Text Driver (*.txt, *.csv)']

# We can use a SQL ODBC Driver or FreeTDS
DRIVER={ODBC Driver 17 for SQL Server};SERVER=SERVERNAME,PORT;DATABASE=DB;
UID=USERNAME;PWD=PASSWORD
```

2.0.4 Usage

2.0.4.1 Initialization

```
import sqlserver
db = sqlserver.adgsqlserver('yourconnectionstring')
```

2.0.4.2 Commands

2.0.4.2.1 Execute Query `parms`: ExecuteQuery(query:str)

This enables you to execute any query without any `stdout` but returns a `bool True` or `False` if query passes and logs `exception` in terminal as `stdout`.

```
query = 'somequery'
db.ExecuteQuery()
```

2.0.4.2.2 Return Query as Dictionary `parms`: GetRecordsAsDict(query:str)

We use this for `select` statements or any other query that returns a `table` as a result.

```
query = "SELECT 'Connection Passed' AS Result"
db.GetRecordsAsDict(query)
```

`stdout`

```
{'results': [{'Result': 'Connection Passed'}]}
```

2.0.4.2.3 Return Query as Column List `parms`: GetRecordsOfColumn(query:str,ColumnName:str)

We use this for `select` statements or any other query that returns a `table` as a result.

```
db.GetRecordsOfColumn("SELECT 'Connection Passed' AS Result", "Result")
```

`stdout`

```
['Connection Passed']
```

2.0.4.2.4 Create CSV Table Schema `parms`: CreateCSVTable(csvfile:str)

Creates a SQL Table with varchar(max) columns such that it can be ready to be inserted to based on the .csv column names

Assumption: `somefile.csv`

```
name,surname,phonenumber
test,testor,01234567810
```

```
path = 'C:/somefile.csv'
db.CreateCSVTable(path)
```

In SQL Table `somefile.dbo`

```
|name|surname|phonenumber|
```

2.0.4.2.5 Insert CSV Table `parms`: InsertCSVTable(csvfile:str)

Assumption: `somefile.csv`

```
name,surname,phonenumber
test,testor,01234567810
```

```
path = 'C:/somefile.csv'
db.InsertCSVTable(path)
```

In SQL Table `somefile.dbo`

```
--------------------------
|name|surname|phonenumber|
|----|-------|-----------|
|test|testor |01234567810|
--------------------------
```

2.0.4.2.6 Insert XML `parms`: InsertXMLSQLTable(xmlfilepath:str)

```
xmlfilepath = 'C:/somexml.xml'
db.InsertXMLSQLTable(xmlfilepath)
```

Chapter 3

Performance

3.0.1 Testing the performance of the bulk .csv insertion commit

We will use a dataset called *baby − names.csv* which contains the names of babies born in the United States from 1880 to 2014. The dataset is available at https://raw.githubusercontent.com/hadley/data-baby-names/master/baby-names.csv

It consists of n rows and m columns. The columns are *year*, *name*, *percent* and *sex*.

And n is the number of rows in the dataset which is 258000.

In this case we will use the n rows to test the performance of the bulk .csv insertion commit.

We will run two tests:

1. Native SQL Server Management Studio and the bulk .csv insertion commit.
2. Our Python script and the bulk .csv insertion commit using *sqlserver* module.

I will be using my laptop to run the tests.

My system configuration is as follows:

- Processor Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz, 2496 Mhz, 4 Core(s), 8 Logical Processor(s)
- Installed Physical Memory (RAM) 8.0 GB
- 512GB NVMe SSD

The database will be hosted locally on my laptop.
The database will be hosted on the same machine as the Python script.
I will be using SQL Server 2019 Developer Edition.

Using the *app.py* i created using my Python script, i will be inserting the n rows into the database.

The code for the *app.py* is as follows:

```
import sqlserver
import time
connectionstring = 'Driver={ODBC Driver 17 for SQL Server};Server=(local);Database=ADGSTUDIOS;
+Trusted_Connection=yes;'
db = sqlserver.adgsqlserver(connectionstring)

path = r'C:\Users\adgru\Code\datasets\babynames.csv'
```

```
start = time.time()
print('Creating table')
db.CreateCSVTable(path)
print('Table created')
print('Time taken to create table: ', time.time() - start)
print('inserting data')
db.InsertCSVData(path)
print('Data inserted')
print('Time taken to insert data: ', time.time() - start)
end = time.time()
# print time in seconds
print('Total time taken: ', end - start)
```

When running the *app.py* i get the following stdoutput:

```
Table created
Time taken to create table:  0.1863565444946289
inserting data
Data inserted
Time taken to insert data:  63.31948184967041
Total time taken:  63.31948184967041
```

Quite interestingly, the time taken to create the table is 0.1863565444946289 seconds.
The time taken to insert the data is 63.31948184967041 seconds.
The total time taken is 63.31948184967041 seconds.

It is amazingly to see that thte total time taken = time taken to insert data.
This is because the time taken to create the table is negligible.
But in some cases the time taken to create the table can be significant.
As there could be a delay in transaction query processing.

Now we will test the performance of the bulk .csv insertion commit using SQL Server Management Studio.

The steps to test the performance of the bulk .csv insertion commit using SQL Server Management Studio are as follows:

1. Open SQL Server Management Studio.
2. Connect to the database.
3. Import Flat File Wizard.
4. Select the $baby-names.csv$ file.
5. Select the *dbo* schema.
6. Create a new table.
7. Process the file and import the data.

In this experiment we only calculate the processing time. Using the GUI to input settings is not included in the processing time.
It is important to note that the processing time is the time taken to import the data and commit the transaction.
By using a stop watch.
There might be human error in the stop watch.
But the difference in the time taken will be negligible.

Note that sql set the following scheme for the table:

```
year:smallint
name:nvarchar(50)
percent:float
sex:nvarchar(50)
```

But wait in few seconds i run the job. It crashes. The input string is not in the correct format.

I set all the columns to $nvarchar(max)$.

```
year:nvarchar(max)
name:nvarchar(max)
percent:nvarchar(max)
sex:nvarchar(max)
```

The job runs successfully and the t taken to process the file was 29.17 seconds.
Thats cool! Well done SQL Server Management Studio.

Lets compare the performance of the bulk .csv insertion commit using SQL Server Management Studio and the bulk .csv insertion commit using *sqlserver* module.

Platform	t
SQL Server Management Studio Import Flat File	29.17 seconds
Python *sqlserver* module	63.32 seconds

Table 3.1: Performance of the bulk .csv insertion commit using SQL Server Management Studio and the bulk .csv insertion commit using *sqlserver* module

The SQL Server Management Studio is 2.18 times faster than the *sqlserver* module.
Quite interestingly, the *sqlserver* module is 2.18 times slower than the SQL Server Management Studio.

But wait, surely in the future we can beat the performance of SQL Server Management Studio. Will just have to engineer the *sqlserver* module more efficiently.

For now we take the win, because we dont have to go to the GUI multiple times to set the settings. And we can automate the process for n number of files.

Imagine using the SQL Server Management Studio to import n number of files.
That will be super tedious.

Sometimes the SQL Server Management Studio Import Flat File Wizard crashes when importing a large file or fails.

If the invalid datatypes are set in the table. You will have to manually change the datatypes for each column m.
This is not efficient the time spending on the GUI will be significant.

In overall if you got a large dataset and you want to import it into a database and its only one file. Then the SQL Server Management Studio Import Flat File Wizard is the way to go and make sure you set the correct datatypes.
But if you got datasets and you want to import it into a database then the *sqlserver* module is the way to go.
You can automate the process for n number of files.

My final point is that the SQL Server Management Studio only allows the
$fileExt \in [.csv, .txt]$ where as my *sqlserver* module takes any file type pandas supports.

Was a cool experiment. I am glad with the results.

Feel free to use the *sqlserver* module and as well contribute to the module on GitHub. I am open to any suggestions and feedback.