The Mass Insertion of SQL Server Records Algorithm

Mr Ashlin Darius Govindasamy
University of South Africa

November 8, 2022

Abstract

This paper presents a new algorithm for the mass insertion of records into a SQL Server database and it can handle any dataframe which pandas supports. Instead of using the traditional method of inserting records one by one, this algorithm uses the bulk insert method which is much faster. The algorithm is written in Python and it is tested on a dataset of 1 million records.

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 $queryBuffer$.

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

```
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.