How to Create a Simple ETL Job Locally With Spark, Python, and MySQL

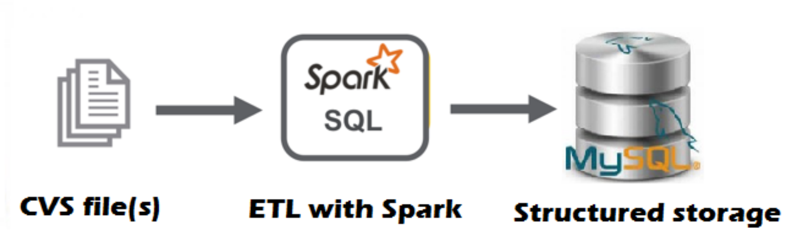
This article demonstrates how Apache Spark can be writing powerful ETL jobs using PySpark.

by Gavaskar Rathnam · May. 27, 20 · Big Data Zone · Tutorial

Introduction

This article demonstrates how Apache Spark can be writing powerful ETL jobs using PySpark. PySpark helps you to create more scalable processing and analysis of (big) data.

In our case, we will work with a dataset that contains information from over 370,000 used cars (*data hosted on Kaggle)* besides. It's important to note that the content of the data is in German.



What Is Apache Spark

Apache Spark is one of the most popular engines for large-scale data processing. It's an open-source system with an API supporting polyglot programming languages. Processing of data is done in memory, hence it’s 100 times faster than MapReduce. Spark comes with libraries that supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. It’s able to run from your local computer, but also can be scaled up to a cluster of hundreds of nodes.

What Is ETL?

ETL (**E**xtract, **T**ransform, and **L**oad) is the procedure of migrating data from one system to another. Data extraction is the process of retrieving data out of homogeneous or heterogeneous sources for further data processing and data storage. During data processing, the data is being cleaned and incorrect or inaccurate records are being modified or deleted. Finally, the processed data is loaded (e.g. stored) into a target system such as a data warehouse or data lake or NoSQL or RDBMS.

Extract

The starting point of every Spark application is the creation of a SparkSession. This is a driver process that maintains all relevant information about your Spark Application, and it is also responsible for distributing and scheduling your application across all executors. We can simply create a SparkSession in the following way:

Python

```python
def initialize_spark ():
    spark = SparkSession.builder \
        .master("local[*]") \
        .appName("simple etl job") \
        .getOrCreate()
    return spark
```

The getOrCreate() method will try to get a SparkSession if one is already created, otherwise, it will create a new one. With the master option, it is possible to specify the master URL that is being connected. However, because we’re running our job locally, we will specify the *local[*]* argument. This means that Spark will use as many worker threads as logical cores on your machine. We set the specify the local[*] argument. This means that Spark will use as many worker threads as logical cores on your machine. We set the application name with the appName option, this name will appear in the Spark UI and log data.

Our next step is to read the CSV file. Reading in a CSV can be done with a DataFrameReader that is associated with our SparkSession.

To choose for schema inference or manually defining a schema depends heavily on the use case, in case of writing an ETL job for a production environment, it is strongly recommended to define a schema in order to prevent inaccurate data representation. Another constraint of schema inference is that it tends to make your Spark application slower, especially when working with CSV or JSON. The example below, how to read in data with a prior defined schema:

SUPP
We res
adverts
the inte
view a

Python

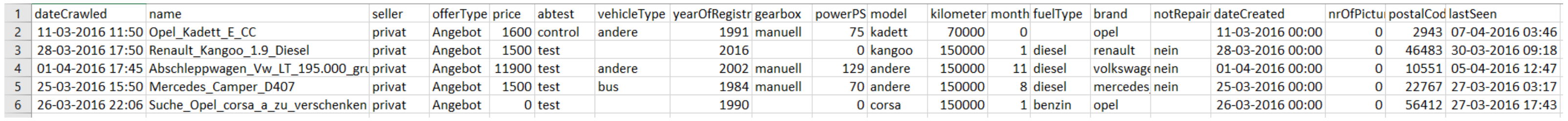
```python
def load_df_with_schema(spark):
    schema = StructType([
        StructField("dateCrawled", TimestampType(), True),
        StructField("name", StringType(), True),
        StructField("seller", StringType(), False),
        StructField("offerType", StringType(), True),
        StructField("price", LongType(), True),
        StructField("abtest", StringType(), True),
        StructField("vehicleType", StringType(), True),
        StructField("yearOfRegistration", StringType(), True),
        StructField("gearbox", StringType(), True),
        StructField("powerPS", ShortType(), True),
        StructField("model", StringType(), True),
        StructField("kilometer", LongType(), True),
        StructField("monthOfRegistration", StringType(), True),
        StructField("fuelType", StringType(), True),
        StructField("brand", StringType(), True),
        StructField("notRepairedDamage", StringType(), True),
        StructField("dateCreated", DateType(), True),
        StructField("nrOfPictures", ShortType(), True),
        StructField("postalCode", StringType(), True),
        StructField("lastSeen", TimestampType(), True)
    ])

    df = spark \
        .read \
        .format("csv") \
        .schema(schema)         \
        .option("header", "true") \
        .load(environ["HOME"] + "/data/autos.csv")

    print("Data loaded into PySpark", "\n")
    return df
```

We res
adverts
the inte
view a
can lea
Thanks

SUPPC

Transform

We have a closer look at our data and start to do more interesting stuff:



1	dateCrawled	name	seller	offerType	price	abtest	vehicleType	yearOfRegistr	gearbox	powerPS	model	kilometer	month	fuelType	brand	notRepair	dateCreated	nrOfPictu	postalCod	lastSeen
2	11-03-2016 11:50	Opel_Kadett_E_CC	privat	Angebot	1600	control	andere	1991	manuell	75	kadett	70000	0		opel		11-03-2016 00:00	0	2943	07-04-2016 03:46
3	28-03-2016 17:50	Renault_Kangoo_1.9_Diesel	privat	Angebot	1500	test		2016		0	kangoo	150000	1	diesel	renault	nein	28-03-2016 00:00	0	46483	30-03-2016 09:18
4	01-04-2016 17:45	Abschleppwagen_Vw_LT_195.000_gru	privat	Angebot	11900	test	andere	2002	manuell	129	andere	150000	11	diesel	volkswage	nein	01-04-2016 00:00	0	10551	05-04-2016 12:47
5	25-03-2016 15:50	Mercedes_Camper_D407	privat	Angebot	1500	test	bus	1984	manuell	70	andere	150000	8	diesel	mercedes	nein	25-03-2016 00:00	0	22767	27-03-2016 03:17
6	26-03-2016 22:06	Suche_Opel_corsa_a_zu_verschenken	privat	Angebot	0	test		1990		0	corsa	150000	1	benzin	opel		26-03-2016 00:00	0	56412	27-03-2016 17:43

Sample five rows of the car dataset

As you can see, there are multiple columns containing null values. We can handle missing data with a wide variety of options. However, discussing this is out of the scope of this article. As a result, we choose to leave the missing values as null. However, there are more strange values and columns in this dataset, so some basic transformations are needed:

The rationale for this cleaning is based on the following: the columns “dateCrawled” and “lastSeen” doesn’t seem to be useful for any future analysis. All the values in the column “nrOfPictures” were equal to 0, hence we decided to drop this column.

seller

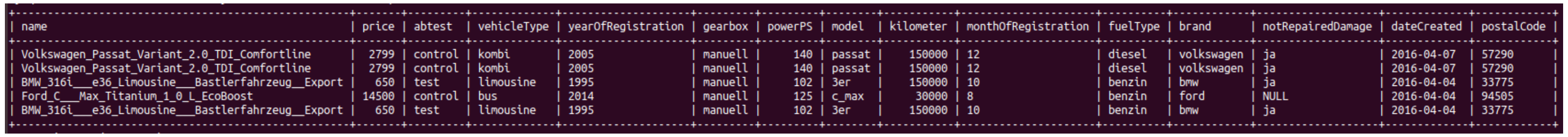
gewerblich	3
private	371525

offerType

Angebot	371513
Gesuch	12

Gesuch 12

Inspecting the columns “seller” and “offerType” resulted in the following numbers. As a result, we can remove the three rows containing value “gewerblich” and then drop the column “seller”. The same logic applies also for the column “offerType”, consequently, we’re left with a more balanced dataset. For example, we leave the dataset like this:



```
+----------------------------------------------------+-------+---------+-------------+--------------------+---------+---------+--------+-----------+---------------------+----------+------------+-------------------+-------------+------------+
| name                                               | price | abtest  | vehicleType | yearOfRegistration | gearbox | powerPS | model  | kilometer | monthOfRegistration | fuelType | brand      | notRepairedDamage | dateCreated | postalCode |
+----------------------------------------------------+-------+---------+-------------+--------------------+---------+---------+--------+-----------+---------------------+----------+------------+-------------------+-------------+------------+
| Volkswagen_Passat_Variant_2.0_TDI_Comfortline      |  2799 | control | kombi       | 2005               | manuell |     140 | passat |    150000 | 12                  | diesel   | volkswagen | ja                | 2016-04-07  | 57290      |
| Volkswagen_Passat_Variant_2.0_TDI_Comfortline      |  2799 | control | kombi       | 2005               | manuell |     140 | passat |    150000 | 12                  | diesel   | volkswagen | ja                | 2016-04-07  | 57290      |
| BMW_316i___e36_Limousine___Bastlerfahrzeug__Export |   650 | test    | limousine   | 1995               | manuell |     102 | 3er    |    150000 | 10                  | benzin   | bmw        | ja                | 2016-04-04  | 33775      |
| Ford_C___Max_Titanium_1_0_L_EcoBoost               | 14500 | control | bus         | 2014               | manuell |     125 | c_max  |     30000 | 8                   | benzin   | ford       | NULL              | 2016-04-04  | 94505      |
| BMW_316i___e36_Limousine___Bastlerfahrzeug__Export |   650 | test    | limousine   | 1995               | manuell |     102 | 3er    |    150000 | 10                  | benzin   | bmw        | ja                | 2016-04-04  | 33775      |
+----------------------------------------------------+-------+---------+-------------+--------------------+---------+---------+--------+-----------+---------------------+----------+------------+-------------------+-------------+------------+
```

Last five rows of the ‘cleaned’ car dataset

SUPP

We res
adverts
the inte
view a
can lea
Thanks

SUPPC

Load

We have translated our raw data into analysis-ready data, hence we’re ready to load our data into our locally running MySQL database for further analysis. For example, we initialized a MySQL database with “autos” and a table with “cars”.

Steps to connect MySQL database in Python using MySQL Connector Python

1. Install MySQL Connector Python using pip.
2. Use mysql.connector.connect() method of MySQL Connector Python with required parameters to connect MySQL.
3. Use the connection object returned by a connect() method to create a cursor object to perform Database Operations.
4. The cursor.execute() to execute SQL queries from Python.
5. Close the Cursor object using a cursor.close() and MySQL database connection using connection.close() after your work completes.
6. Catch Exception if any that may occur during this process.

Now that we have created a cursor, we are able to create a table named “cars” in our “autos” database:

After creating the table, it's now ready to be populated with our dataset. We can insert our data, by providing our data as a list of tuples (where each record is one tuple) to our INSERT statement:

As a result, it is now possible to execute this command with our previously defined cursor:

Python

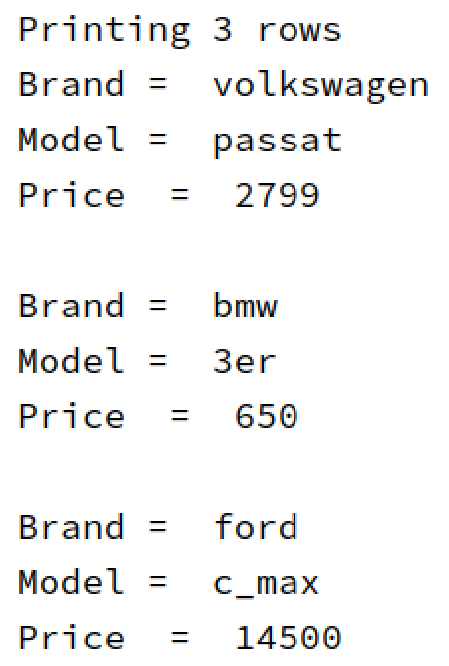
```python
cur.executemany(insert_query, cars_seq) # we are inserting multiple rows (from a List) into the table.
```

adverts
the inte
view a
can lea
Thanks

SUPPC

Which gives us the following output in our terminal:

Printing 3 rows



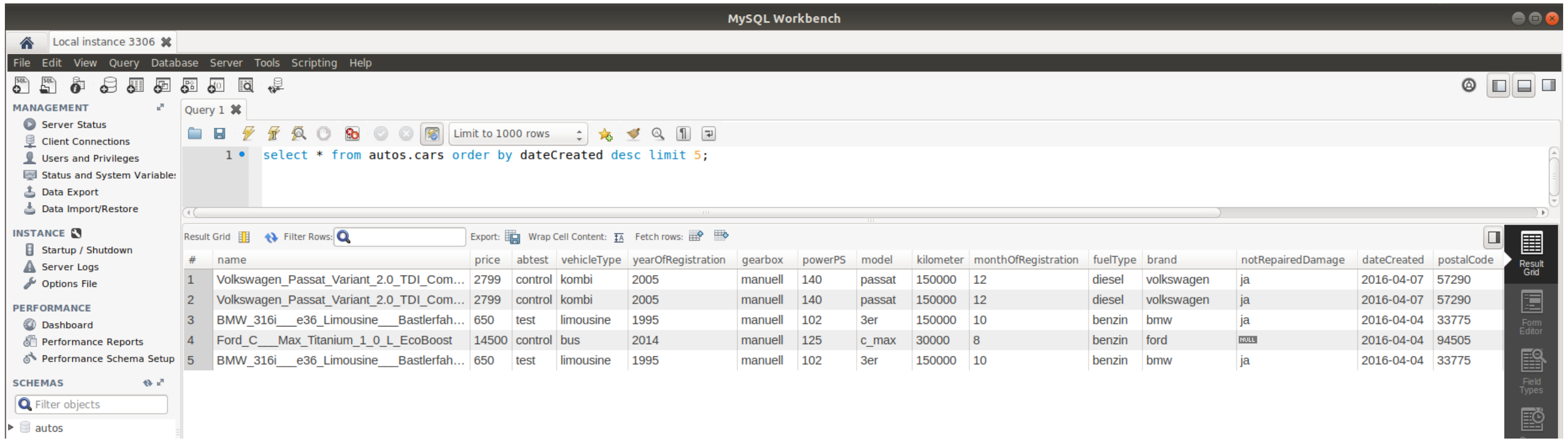
```
Printing 3 rows
Brand =  volkswagen
Model =  passat
Price  =  2799

Brand =  bmw
Model =  3er
Price  =  650

Brand =  ford
Model =  c_max
Price  =  14500
```

We need to call the following code to commit our transaction to MySQL:

Python

```
conn.commit()
```

And to make sure, we can check in MySQL workbench if the dataset is loaded correctly in MySQL:



```sql
select * from autos.cars order by dateCreated desc limit 5;
```

#	name	price	abtest	vehicleType	yearOfRegistration	gearbox	powerPS	model	kilometer	monthOfRegistration	fuelType	brand	notRepairedDamage	dateCreated	postalCode
1	Volkswagen_Passat_Variant_2.0_TDI_Com...	2799	control	kombi	2005	manuell	140	passat	150000	12	diesel	volkswagen	ja	2016-04-07	57290
2	Volkswagen_Passat_Variant_2.0_TDI_Com...	2799	control	kombi	2005	manuell	140	passat	150000	12	diesel	volkswagen	ja	2016-04-07	57290
3	BMW_316i___e36_Limousine___Bastlerfah...	650	test	limousine	1995	manuell	102	3er	150000	10	benzin	bmw	ja	2016-04-04	33775
4	Ford_C___Max_Titanium_1_0_L_EcoBoost	14500	control	bus	2014	manuell	125	c_max	30000	8	benzin	ford	NULL	2016-04-04	94505
5	BMW_316i___e36_Limousine___Bastlerfah...	650	test	limousine	1995	manuell	102	3er	150000	10	benzin	bmw	ja	2016-04-04	33775

Final Remarks

PySpark is a very powerful and useful (big) data tool for any Data Engineer or Data Scientist who is trying to build scalable data applications. The code of this article can be found on GitHub. Since this was my first article on a platform. Please feel free to provide me with your feedback or comments.

References

- https://spark.apache.org/docs/latest/sql-programming-guide.html
- https://dev.mysql.com/doc/connector-python/en/connector-python-examples.html
- https://www.kaggle.com/orgesleka/used-cars-database

Like This Article? Read More From DZone