**How to create a simple ETL Job locally with PySpark, PostgreSQL and Docker**

[Anis Boudih](https://itnext.io/@anisboudih?source=post_page-----ea53cd43311d----------------------)

[Anis Boudih](https://itnext.io/@anisboudih?source=post_page-----ea53cd43311d----------------------)

[Follow](https://medium.com/m/signin?operation=register&redirect=https%3A%2F%2Fitnext.io%2Fhow-to-create-a-simple-etl-job-locally-with-pyspark-postgresql-and-docker-ea53cd43311d&source=post_page-10bdd399e06b----ea53cd43311d----------------------)

[Sep 6](https://itnext.io/how-to-create-a-simple-etl-job-locally-with-pyspark-postgresql-and-docker-ea53cd43311d?source=post_page-----ea53cd43311d----------------------) · 6 min read



**Introduction**

In this article, I’m going to demonstrate how Apache Spark can be utilised for writing powerful ETL jobs in Python. If you’re already familiar with Python and working with data from day to day, then PySpark is going to help you to create more scalable processing and analysis of (big) data.

The data that I’ll use is scraped from Ebay-Kleinanzeigen, which is the German branch of Ebay where people can advertise their properties. In our case, we will work with a dataset that contains information from over 370000 used cars; besides, it’s important to note that the content of the data is in German.

[link to data hosted on Kaggle](https://www.kaggle.com/orgesleka/used-cars-database)

**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 multiple programming languages. Processing of data is done in memory, hence it’s several times faster than for example MapReduce. Spark comes with libraries supporting a wide range of tasks, such as streaming, machine learning and SQL. It’s able to run from your local computer, but also can be scaled up to a cluster of hundreds of servers.

**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 database such as a data warehouse or data lake.

**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:

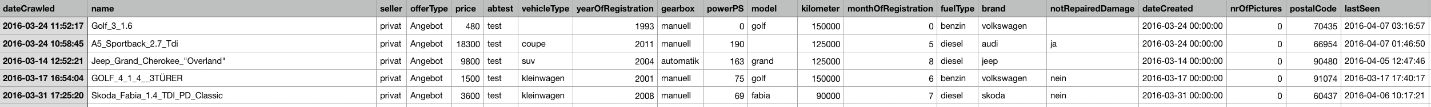
The getOrCreate method will try to get a SparkSession if one is already created, otherwise it will create a new one. With the master option it is possible to specify the master URL that is being connected. However, because we’re running our job locally, we will specify the *local[\*]* argument. This means that Spark will use as many worker threads as logical cores on your machine. We set the 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. In doing so, Spark allows us to specify whether schema inference is being used as well as some other options:

Whether 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. Therefore, I’m also showing how to read in data with a prior defined schema:

**Transform**

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



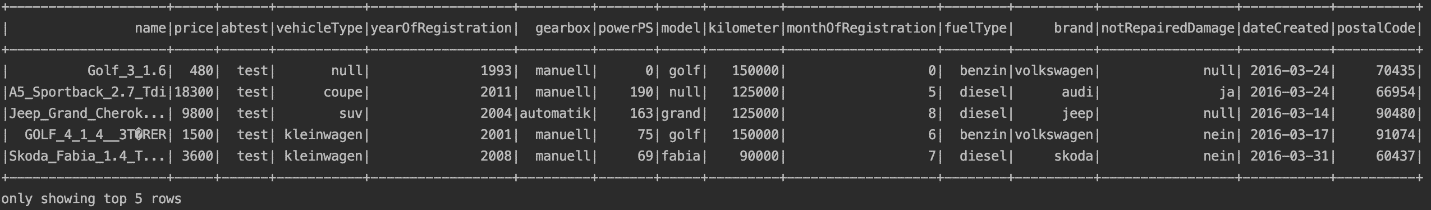
First five rows of the car dataset

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  
privat 371525**offerType**  
Angebot 371513  
Gesuch 12

Inspecting the columns seller and offerType resulted in the following numbers. As a result, we can remove the three rows containing “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 the sake of example, we leave the dataset like this:



First five rows of the ‘cleaned’ car dataset

**Load**

We have translated our raw data into analysis-ready data, hence we’re ready to load our data into our locally running PostgreSQL database for further analysis in the nearby future. We have spun up a PostgreSQL database with pgAdmin with this basic [docker-compose file](https://github.com/klimpie94/pyspark-etl-analytics/blob/master/docker-compose.yml). This docker-compose configuration file defines all containers in our current setup with their corresponding settings. For example, we initialised a PostgreSQL database with namecars, username admin and password admin.

**Psycopg2** is the most popular PostgreSQL database driver for Python. It provides a simple and concise way for interacting with a PostgreSQL instance. First, we establish a connection to thecars database:

In doing so, we have to provide some general parameters to the connect function. There is also a possibility to specify the port, however, in our case there is no need to specify the port, since we’re running the PostgreSQL instance on the default port 5432. So, we have our session started and we’re connected to Postgres. After having a connection, it’s possible to write our commands (e.g. inserts, updates) and **Psycopg2** allows us to do this with cursors. A cursor is created out of a connection and it will allow you to communicate with PostgreSQL.

# cursor  
cur = conn.cursor()

Now that we have created a cursor, we are able to create a table named cars\_table in our cars database:

After creating the table, it’s now ready to be populated with our dataset. We can insert our data row by row, 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:

cur.execute(insert\_query, cars\_seq)

As you can see, thanks to **Psycopg2**, it is really easy to transfer your data from your application to a backend database such as PostgreSQL. Lastly, we’re going to query our recently populated table with **Psycopg2**:

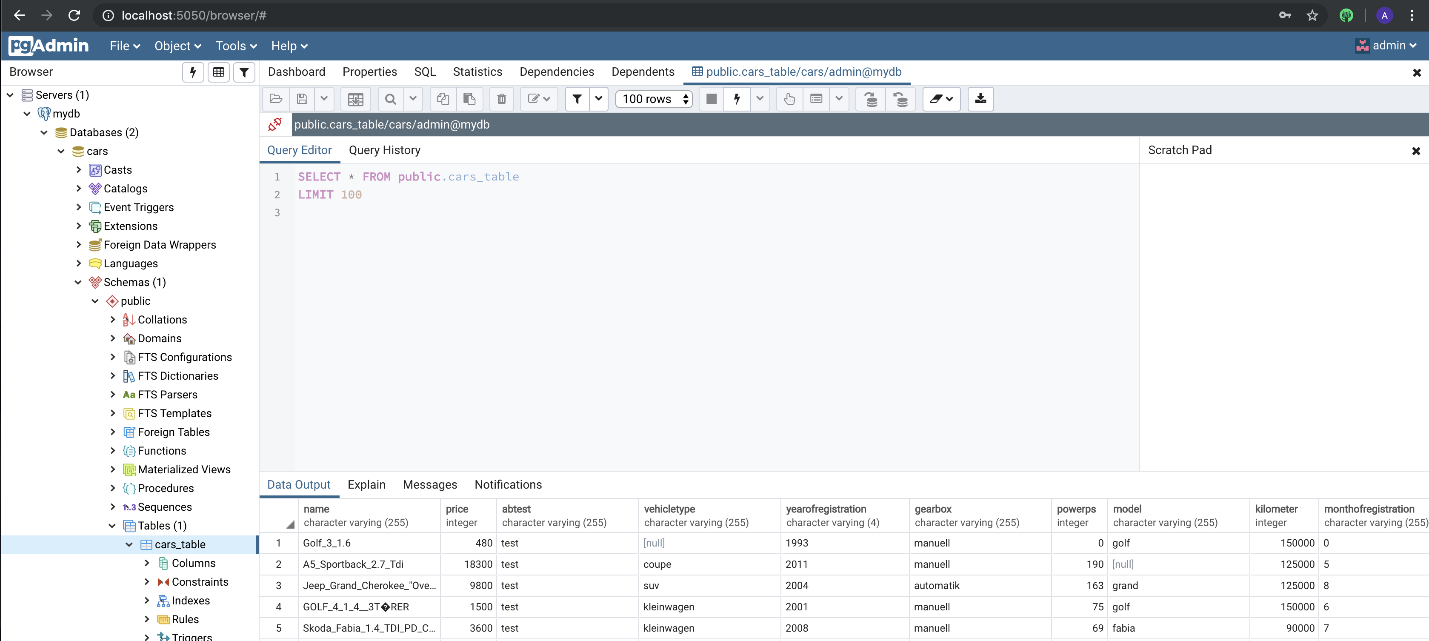
Which gives us the following output in our terminal:

Printing 2 rows  
Brand = volkswagen  
Model = golf  
Price = 480Brand = audi  
Model = None  
Price = 18300

However, we’re still *missing one important* piece of code: **Psycopg**is Python DB-API compliant, so the auto-commit feature is off by default. As a result, we need to call the following code to commit our transaction to PostgreSQL:

conn.commit()

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



Cars dataset loaded in pgAdmin

**Final remarks**

**Pyspark** is a powerful and useful (big) data tool for any Data Engineer or Data Scientist who is trying to build scalable data applications. I can definitely recommend everyone to have a serious look at it and try to incorporate it in one of your future projects. Thanks to **Docker**, we were able to spin up a local **PostgreSQL** database without installing anything! The code of this article can be found on [Github](https://github.com/klimpie94/pyspark-etl-analytics" \t "_blank). Please feel free to provide me with any feedback or comments, since this was my first article on a platform.

Also, feel free to contact me if you’re interested in attending a [Data Engineering Bootcamp](https://www.linkit.nl/vacatures/data-engineering/0007529/data-engineering-apprenticeship-starts-feb-1st-2020) or if you’re just curious about how I experienced these two ‘intense’ months:)

**References**

* [Official Apache Spark website](https://spark.apache.org/)
* “Spark: The Definitive Guide by Bill Chambers and Matei Zaharia (O’Reilly). Copyright 2018 Databricks, Inc., 978–1–491–91221–8.”
* [What is ETL?](https://en.wikipedia.org/wiki/Extract,_transform,_load)
* [Psycopg2 documentation](http://initd.org/psycopg/docs/)

**[ITNEXT](https://itnext.io/?source=post_sidebar--------------------------post_sidebar-)**

ITNEXT is a platform for IT developers & software engineers to share knowledge, connect, collaborate, learn and experience next-gen technologies.

Thanks to Nils Nieuwenkamp and Kiarash Irandoust.