Running PySpark with Cassandra in Jupyter

Amir Ziai
Apr 1, 2016 · 4 min read

Apache Cassandra is an open-source distributed database system. Interfacing with Cassandra using Python is made possible using the Python client driver which you can pip install:

```
pip install cassandra-driver
```

In Python you can simply execute Cassandra Query Language (CQL) queries and receive the results in an iterator. CQL is syntactically very similar to SQL, however it misses join, group by and some other operations.

```
from cassandra.cluster import Cluster
cluster = Cluster(['10.0.0.0'])  # provide contact points and port
session = cluster.connect('keyspace')
rows = session.execute('select * from table limit 5;')

for row in rows:
    print row.id
```

```
import pandas as pd

rows = session.execute('select * from table;')
df = pd.DataFrame(list(rows))
```

Which reads the whole table into memory.

If your data does not fit into memory you can use Apache Spark. Spark is an open source cluster computing framework which comes with a Python API called PySpark.

A simple way to get started with PySpark is to download a pre-built version for Hadoop. Here is an example for Spark 1.6.1 with Hadoop 2.6 and later:

```
$ wget http://www.apache.org/dyn/closer.lua/spark/spark-1.6.1/spark-
1.6.1-bin-hadoop2.6.tgz
```

Then add the following environment variable (SPARK_HOME) to your bash profile (~/.bash_profile or ~/.bashrc):

```
export SPARK_HOME=/path/to/download/spark/folder
```

and run the following:

```
. ~/.bashrc (or . ~/.bash_profile)
```

to make the variable available.

Now you have Spark! We still need to take a few more steps to get PySpark and Cassandra to work together. One way is to use pyspark-cassandra which uses the Cassandra Spark Connector.

First step is to clone the repository:

Then compile the repo (make sure you have sbt installed first):

```
$cd pyspark-cassandra
$sbt compile
```

Create a bash script:

```
sudo vi start_jupyter_cassandra.sh.
```

with the following content:

```
#!/bin/bash

cd $SPARK_HOME/python/pyspark/
export IPYTHON_OPTS="notebook"
pyspark --jars /home/ec2-user/pyspark-cassandra/target/scala-
2.10/pyspark-cassandra-assembly-0.2.7.jar --driver-class-path
/home/ec2-user/pyspark-cassandra/target/scala-2.10/pyspark-
cassandra-assembly-0.2.7.jar --conf
spark.cassandra.connection.host=10.0.0.0 &> /dev/null &
```

The script is simply calling pyspark with the IPYTHON_OPTS=”notebook” flag. We also need to pass the JARs that we built when we compiled pyspark-cassandra (make sure you replace the path with your own). The last step is to provide the host address.

Make the script executable:

```
chmod a+x start_jupyter_cassandra.sh
```

And then execute the script:

```
./start_jupyter_cassandra.sh
```

```
sc
```

which should return something like this:
<pyspark.context.SparkContext at 0x7fbb28107890>

SparkContext (sc) represents the connection to your Spark cluster. Once you import pyspar-cassandra your SparkContext changes:

```
import pyspark_cassandra
```

Executing sc returns:

<pyspark_cassandra.context.CassandraSparkContext at 0x7fbb28107890>

Your SparkContext now has an extra method cassandraTable which return a Spark RDD. To get a table you need to specify the keyspace as well as the table name:

```
def get_table(key_space, table):
    return sc.cassandraTable(key_space, table)

table = get_table('key_space', 'table')
```

Even though table is now a Spark RDD (technically a MapPartitionsRDD), the pyspark-cassandra API exposes some methods that delegate to Cassandra. For example:

```
print table.cassandraCount()
```

Which never loads the data into Spark and is different from .count() in that it prevents IO.

At this point you can start applying transformations and actions on the RDD. For instance you can only get a subset of the columns using select:

Here's what the output looks like:

```
results.take(5)

[Row(bmi=u'20.89', height=u'5 ft 3 in'),
 Row(bmi=u'25.28', height=u'5 ft 8 in'),
 Row(bmi=None, height=None),
 Row(bmi=u'18.41', height=u'5 ft 6 in'),
 Row(bmi=u'27.04', height=u'5 ft 2.5 in')]
```

Refer to the pyspark-cassandra API documentation for more methods such as "where", which are defined as a part of Cassandra Query Language (CQL).

. . .

For complicated queries and aggregation we can rely on Spark. As an example let's do a filter on results:

```
filtered = results.filter(lambda x: check_bmi(x['bmi'], 35) and
check_height(x['height'], 5))
```

Since "bmi" and "height" are strings, it would be hard to manipulate them as numbers using CQL and "where". Therefore I'm resorting to lambda functions and Spark.

```
def check_bmi(x, threshold):
    try:
        return float(x) >= threshold
    except:
        return False
```

For every record of result, the "bmi" column is passed to the check_bmi function, along with a threshold. check_bmi returns "true" only if "bmi" can be converted to a float and is bigger than the provided threshold (35 in this case). Otherwise we get "false", which effectively filters out the record.

```
def check_height(x, threshold):
    try:
        return float(x.split()[0]) >= threshold
    except:
        return False
```

Now let's count the number of occurrences of height in feet. This is equivalent to performing a "group by" operation in SQL. To accomplish this we can follow these steps:

1- Get height in feet for each filtered row
2- Map (height, 1) tuples to get ready for the reducers to count
3- Sum the mapped tuples by key (height)
4- Collect the reduced results

```
def get_height(x):
    try:
        return float(x.split()[0])
    except:
        return 0

filtered.map(lambda x: (get_height(x['height']), 1)) \
.reduceByKey(lambda a, b: a + b) \
.collect()
```

The result is an array of (height, count) combinations:
[(6.0, 1552), (5.0, 13392), (7.0, 3)]

Alternatively you can use a dataframe:

```
df = results.toDF()
df.groupby('pulse').count().collect()
```

which returns:
[Row(pulse=u'160 bpm ', count=124),
Row(pulse=u'45 bpm ', count=37),
Row(pulse=u'18 bpm', count=17), …]

Some good references:

http://www.slideshare.net/JonHaddad/intro-to-py-spark-and-cassandra

http://rustyrazorblade.com/2015/07/cassandra-pyspark-dataframes-revisted/

Spark Python Cassandra