**Exercise 10**

*Get started with Cassandra and import data via Spark*

**Prior Knowledge**

Unix Command Line Shell

HDFS

Simple Python

Spark Python  
Simple SQL syntax

**Learning Objectives**

Understand Cassandra’s CQL shell

Integrate Python, Cassandra and Spark

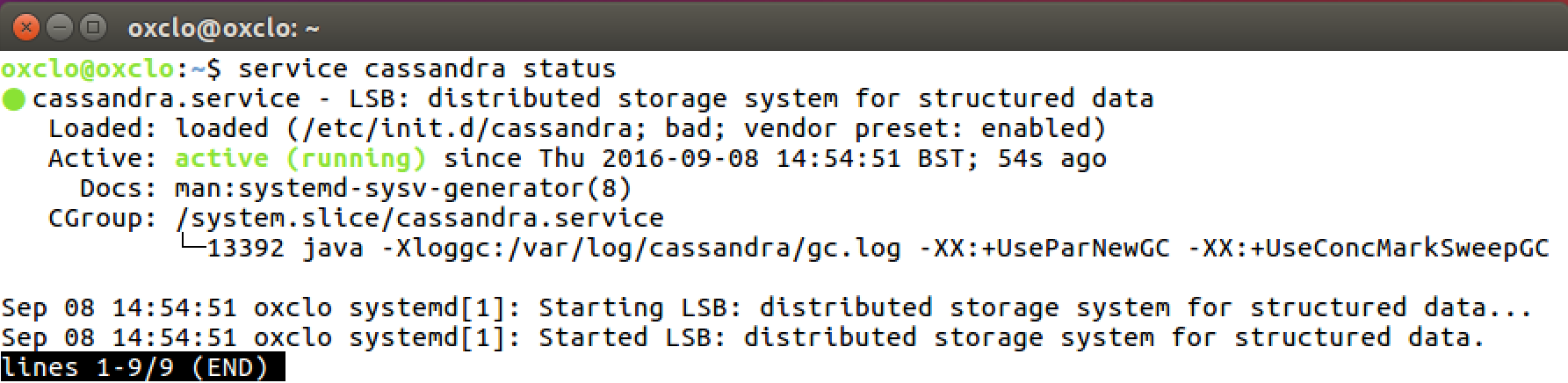
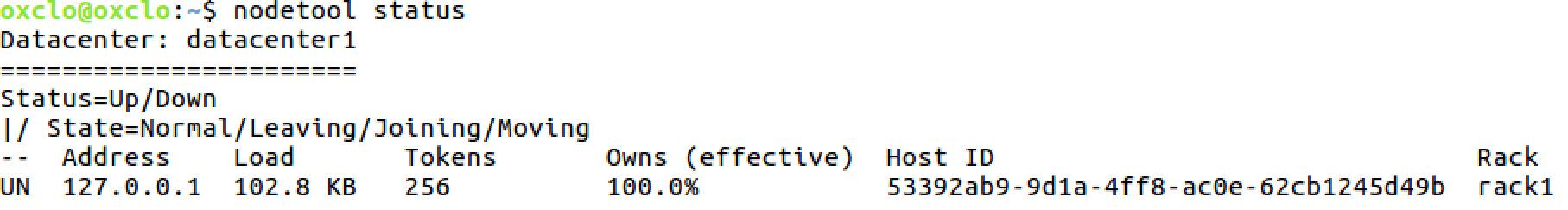
Load data from CSV into Cassandra using Spark Python

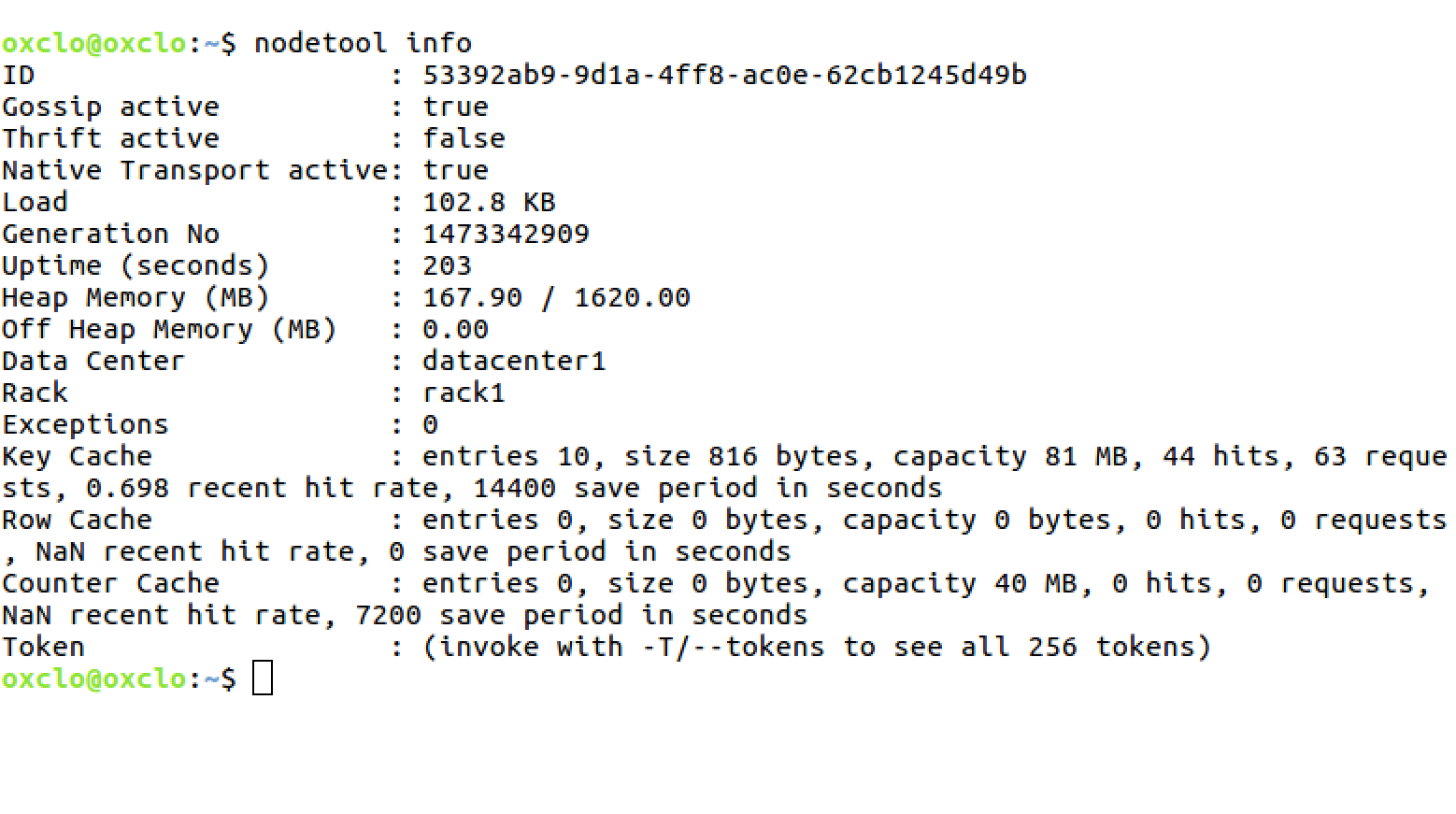
**Software Requirements**

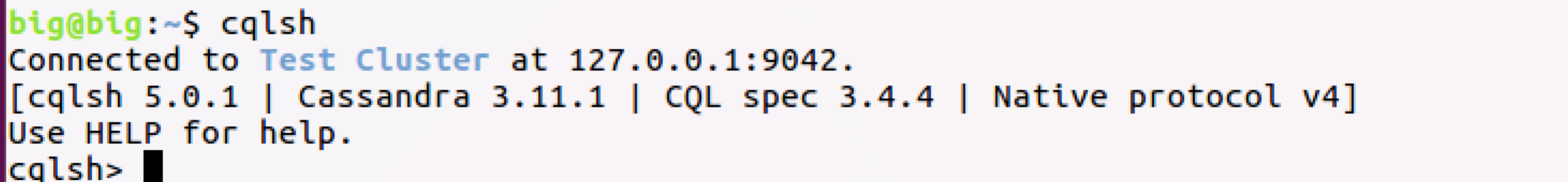
(see separate document for installation of these)

* Apache Spark 2.2.0
* Python 2.7.12
* Apache Cassandra 3.11.1
* Nano text editor or other text editor

**Part A**

1. Make sure Cassandra is running
   1. In a Terminal window (Crtl-Alt-T) type:  
      service cassandra status
   2. You should see  
      
   3. Type q to get back to the command line
   4. If not, try   
      sudo service cassandra start  
      and then check the status again
2. Now you can ask Cassandra about its own situation:  
   nodetool status  
     
   You should see something like:  
   

1. You can also try:  
   nodetool info  
   You should see something like:  
   
2. Now you can start the Cassandra Shell:  
   Type:

cqlsh  
  
You should see:  
  
Let’s create a new database (Keyspace):

* 1. Type (all on a single line)  
       
     CREATE KEYSPACE TEST WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication\_factor' : 1 };
  2. Check it worked:  
     Type:  
       
     desc keyspace test;
  3. You should see:  
       
     CREATE KEYSPACE test WITH replication = {'class': 'SimpleStrategy', 'replication\_factor': '1'} AND durable\_writes = true;

1. Now we need to select to use that keyspace:  
   use test;
2. The command prompt will change to:  
   cqlsh:test>
3. Let’s create a simple (key, value) table
   1. Type:  
      create table kv ( key text, value text, primary key (key));
   2. Now type  
      desc kv;
   3. You should see:

cqlsh:test> desc kv;

CREATE TABLE test.kv (

key text PRIMARY KEY,

value text

) WITH bloom\_filter\_fp\_chance = 0.01

AND caching = '{"keys":"ALL", "rows\_per\_partition":"NONE"}'

AND comment = ''

AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy'}

AND compression = {'sstable\_compression': 'org.apache.cassandra.io.compress.LZ4Compressor'}

AND dclocal\_read\_repair\_chance = 0.1

AND default\_time\_to\_live = 0

AND gc\_grace\_seconds = 864000

AND max\_index\_interval = 2048

AND memtable\_flush\_period\_in\_ms = 0

AND min\_index\_interval = 128

AND read\_repair\_chance = 0.0

AND speculative\_retry = '99.0PERCENTILE';

* 1. Add some simple values:  
     insert into kv (key, value) values ('a','1');

insert into kv (key, value) values ('b','2');

insert into kv (key, value) values ('c','3');

* 1. Now type:  
     select \* from kv;  
       
     You should see:  
     key | value

-----+-------

a | 1

c | 3

b | 2

(3 rows)

1. You can also do other simple SQL of course

cqlsh:test> select \* from kv where key='a' ;

key | value

-----+-------

a | 1

(1 rows)

**PART B – Loading data from CSV files into Cassandra**

1. Firstly, we need to create a database and a table in which to store our data. Start up the **cqlsh** again and type the following commands (available here: <https://freo.me/winddata-ddl> )

CREATE KEYSPACE wind   
WITH replication = {'class': 'SimpleStrategy', 'replication\_factor': '1'};  
  
USE wind;  
  
CREATE TABLE winddata (

stationid text,

time timestamp,

direction float,

temp float,

velocity float,

PRIMARY KEY (stationid, time)

);

1. Type **exit** to leave the cqlsh command line.
2. In order to load the CSV files into Cassandra, we are going to use a Spark packages to help us: the Cassandra plugin for Spark.   
     
   *Please note, there are lots of ways of loading CSV data into Cassandra, including a built-in Cassandra utility, which might be easier to use for small datasets.  
     
   This exercise is designed to demonstrate how to integrate Cassandra with Spark. For a really large dataset, if this was loaded from HDFS into Cassandra, this Spark-based approach would have the major benefit of parallelizing the operation.*
3. To use these, we need to start pyspark with the correct command line. Since we are starting pyspark via Jupyter, we need to pass this via an environment variable.
4. Start a terminal window and type (all on one line)  
     
   export PYSPARK\_SUBMIT\_ARGS=”--packages datastax:spark-cassandra-connector:2.0.3-s\_2.11 pyspark-shell”

It is important that you start jupyter from this window now. If you close the window, this environment will be lost.

1. Start Jupyter as before. Create a new Python2 notebook and copy code from   
   https://freo.me/big-import  
   to the notebook.  
     
   Let’s look at it line by line.

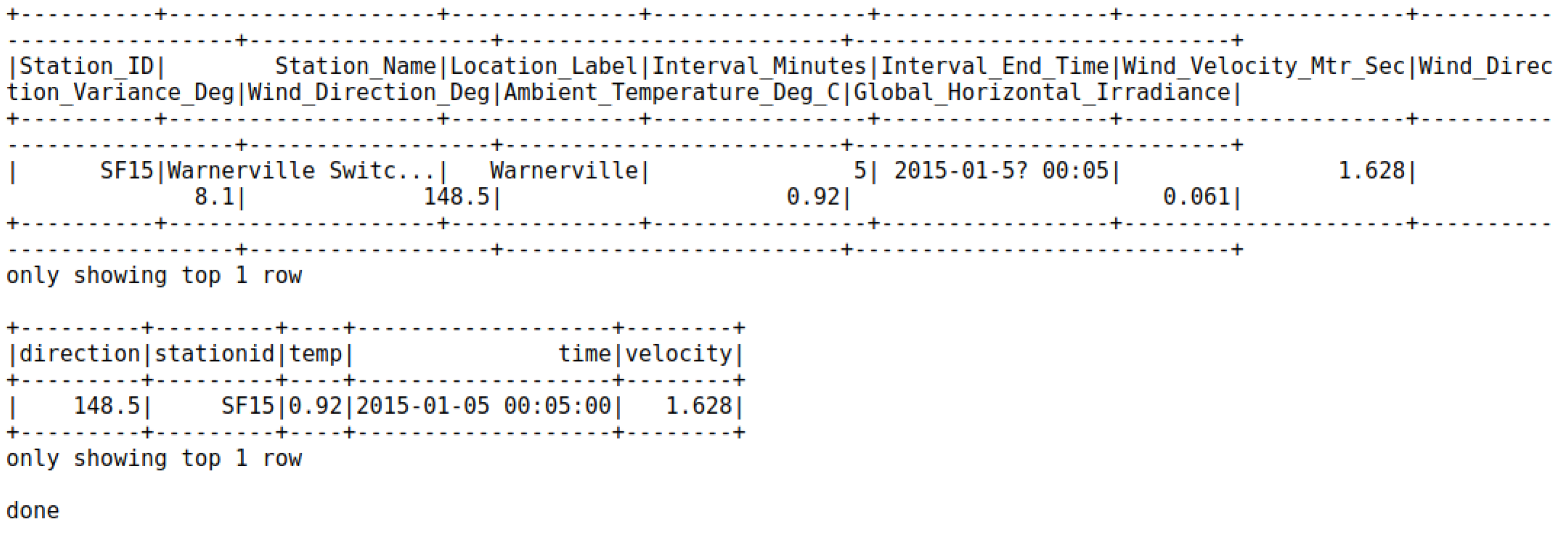
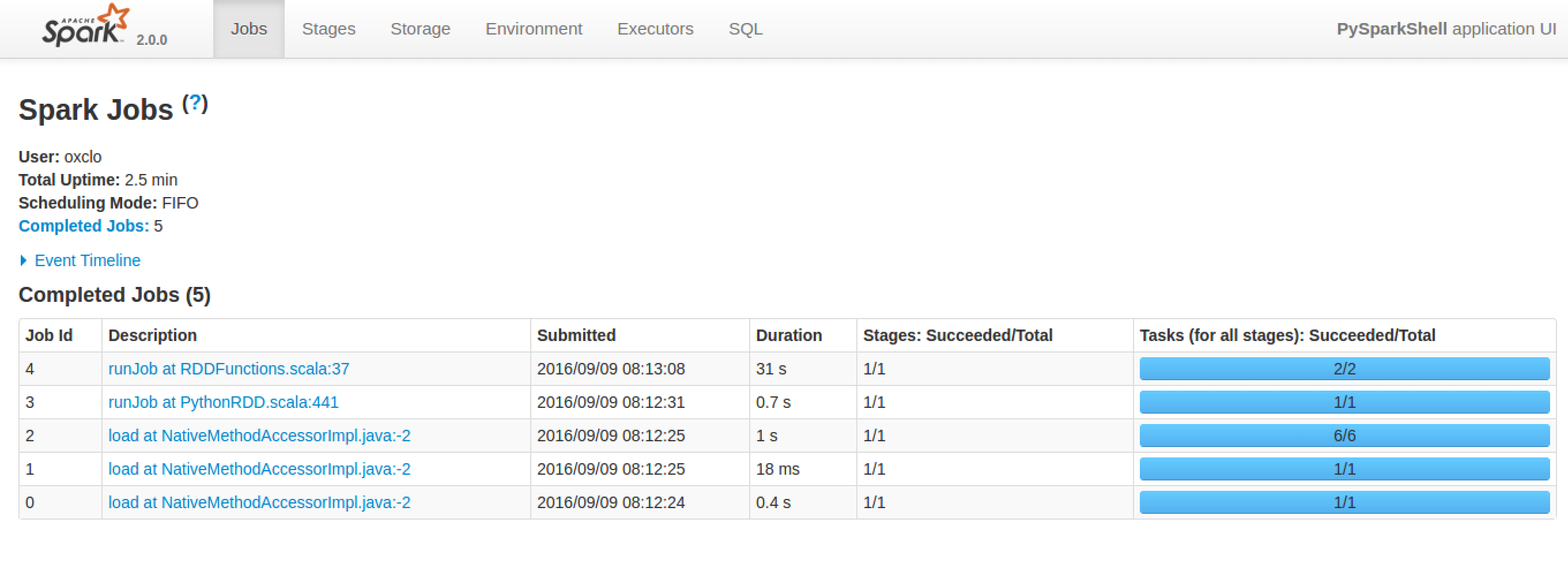
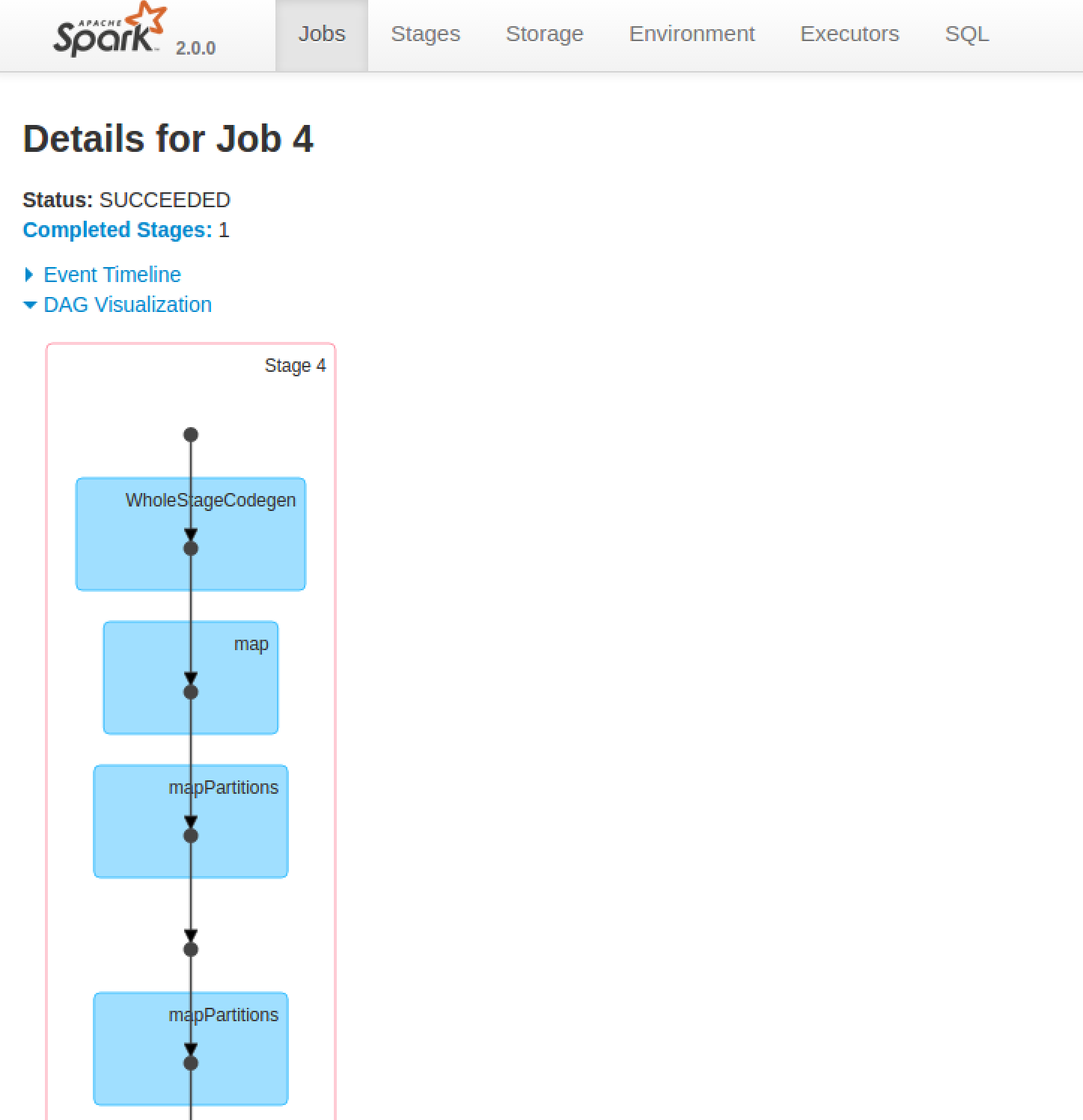
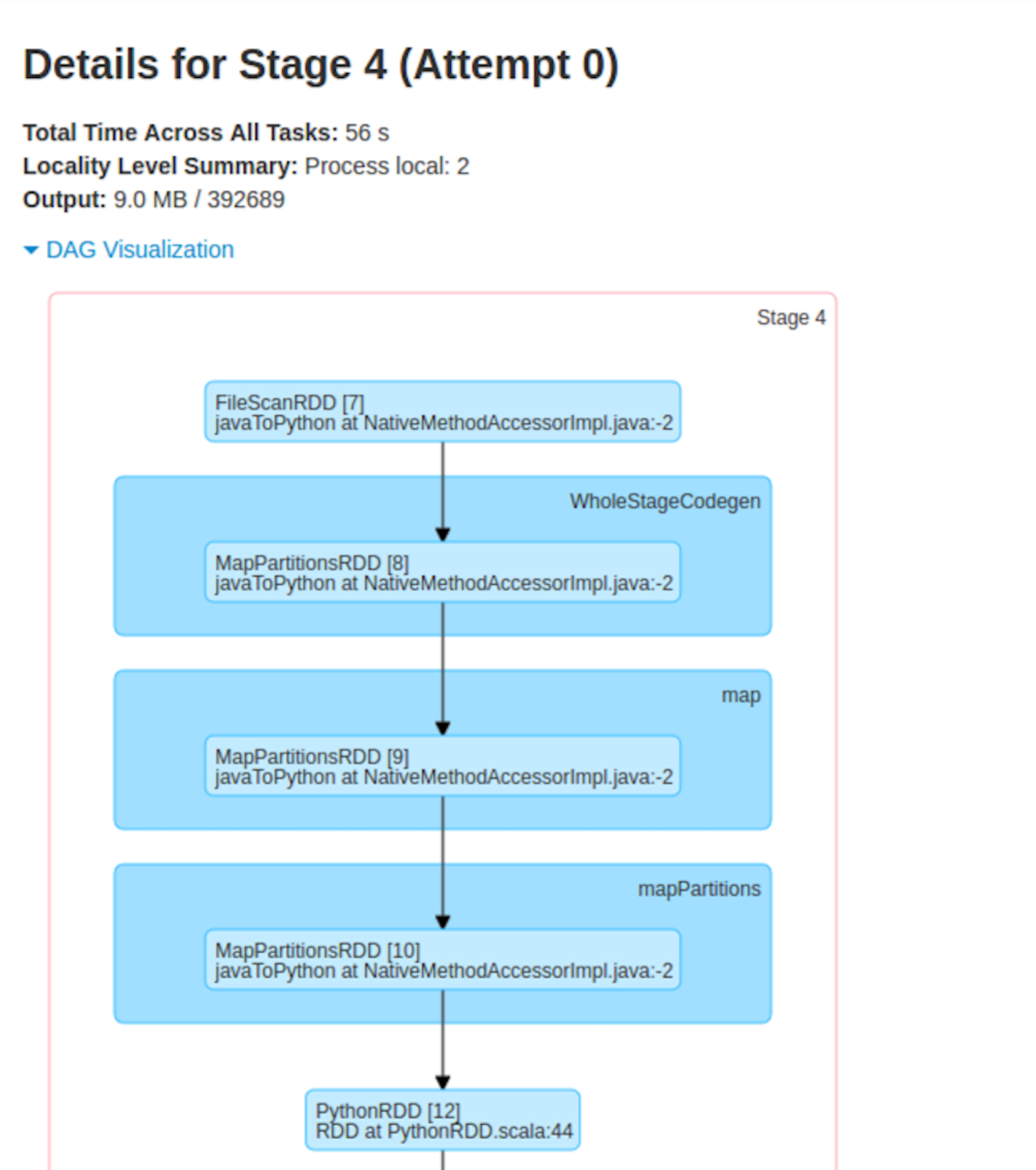


You should recognize lines 1 and 2.  
Line 4: dateutil.parser is a useful utility that can read most common date formats.

Lines 6-8: these are just as in the SQL exercise.  
  
Line 10: This will print out one line of the data we’ve read in so we can see the format.  
  
Line 12. This is a function that will parse the date time string into a Python datetime object. Unfortunately the input format is not one recognized by dateutil.parser, but we can easily fix that by removing the ‘?’.

Line 14: this function replaces one Row with another. In the new Row, the names are simpler (and match those we used to create the keyspace). Also the date is formatted using our **convertTime** function.

Line 20: This converts from a dataframe to an RDD, uses map to apply our clean function, and then converts back to a dataframe.  
  
Line 22: This shows our more beautiful dataframe layout.  
  
Line 24: This exports the dataframe to Cassandra, specifying the database and table to use.

1. Run the cell. This will take a bit of time.
2. You should see the following:  
   
3. Browse to <http://localhost:4040>   
   It will look similar to:  
   
4. Click on the most recent job:   
   
5. You can also get more details by clicking on a stage in the DAG (Directed Acyclic Graph) picture:  
   
6. Check that the data has loaded. Start another terminal window and restart **cqlsh**.
7. In your **cqlsh** window type:  
     
   select \* from wind.winddata limit 15;
8. You should see something like:

stationid | time | direction | temp | velocity

-----------+--------------------------+-----------+-------+----------

SF36 | 2015-01-01 00:00:00+0000 | 116.9 | 11.33 | 2.727

SF36 | 2015-01-01 00:05:00+0000 | 108.5 | 11.25 | 1.814

SF36 | 2015-01-01 00:10:00+0000 | 113.7 | 11.2 | 2.621

SF36 | 2015-01-01 00:15:00+0000 | 117.8 | 11.11 | 3.678

SF36 | 2015-01-01 00:20:00+0000 | 117.3 | 11.07 | 2.842

SF36 | 2015-01-01 00:25:00+0000 | 117.3 | 11.07 | 2.629

SF36 | 2015-01-01 00:30:00+0000 | 117.3 | 11.09 | 2.235

SF36 | 2015-01-01 00:35:00+0000 | 117.2 | 11.09 | 2.043

SF36 | 2015-01-01 00:40:00+0000 | 117.2 | 11.05 | 1.635

SF36 | 2015-01-01 00:45:00+0000 | 117.3 | 10.93 | 2.224

SF36 | 2015-01-01 00:50:00+0000 | 112.5 | 10.86 | 1.822

SF36 | 2015-01-01 00:55:00+0000 | 108.7 | 10.8 | 0.866

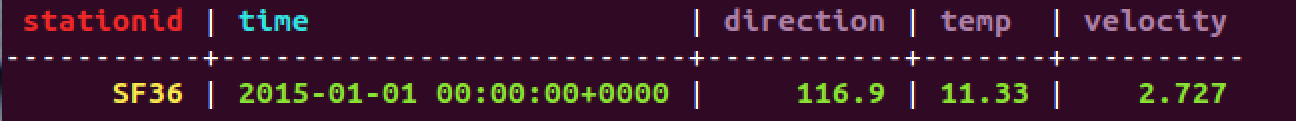
SF36 | 2015-01-01 01:00:00+0000 | 108.7 | 10.67 | 1.068

SF36 | 2015-01-01 01:05:00+0000 | 108.6 | 10.54 | 1.393

SF36 | 2015-01-01 01:10:00+0000 | 108.7 | 10.44 | 1.468

(15 rows)

1. We can do some more queries on the data.

use wind;  
select \* from winddata where time = '2015-01-01' and stationid = 'SF36';  
  
You should see:  


1. Now try  
   select \* from winddata where time <= '2015-01-02' and stationid = 'SF36' limit 20;  
     
   All normal:

stationid | time | direction | temp | velocity

-----------+--------------------------+-----------+-------+----------

SF36 | 2015-01-01 00:00:00+0000 | 116.9 | 11.33 | 2.727

SF36 | 2015-01-01 00:05:00+0000 | 108.5 | 11.25 | 1.814

SF36 | 2015-01-01 00:10:00+0000 | 113.7 | 11.2 | 2.621

SF36 | 2015-01-01 00:15:00+0000 | 117.8 | 11.11 | 3.678

SF36 | 2015-01-01 00:20:00+0000 | 117.3 | 11.07 | 2.842

SF36 | 2015-01-01 00:25:00+0000 | 117.3 | 11.07 | 2.629

SF36 | 2015-01-01 00:30:00+0000 | 117.3 | 11.09 | 2.235

SF36 | 2015-01-01 00:35:00+0000 | 117.2 | 11.09 | 2.043

SF36 | 2015-01-01 00:40:00+0000 | 117.2 | 11.05 | 1.635

SF36 | 2015-01-01 00:45:00+0000 | 117.3 | 10.93 | 2.224

SF36 | 2015-01-01 00:50:00+0000 | 112.5 | 10.86 | 1.822

SF36 | 2015-01-01 00:55:00+0000 | 108.7 | 10.8 | 0.866

SF36 | 2015-01-01 01:00:00+0000 | 108.7 | 10.67 | 1.068

SF36 | 2015-01-01 01:05:00+0000 | 108.6 | 10.54 | 1.393

SF36 | 2015-01-01 01:10:00+0000 | 108.7 | 10.44 | 1.468

SF36 | 2015-01-01 01:15:00+0000 | 108.9 | 10.37 | 1.859

SF36 | 2015-01-01 01:20:00+0000 | 108.6 | 10.29 | 1.67

SF36 | 2015-01-01 01:25:00+0000 | 108.6 | 10.25 | 1.241

SF36 | 2015-01-01 01:30:00+0000 | 108.5 | 10.21 | 0.675

SF36 | 2015-01-01 01:35:00+0000 | 108.4 | 10.26 | 0.623

(20 rows)

1. Now another:   
   select \* from winddata where time <= '2015-01-01 01:00:00' and stationid in ('SF37', 'SF36');

stationid | time | direction | temp | velocity

-----------+--------------------------+-----------+-------+----------

SF36 | 2015-01-01 00:00:00+0000 | 116.9 | 11.33 | 2.727

SF36 | 2015-01-01 00:05:00+0000 | 108.5 | 11.25 | 1.814

SF36 | 2015-01-01 00:10:00+0000 | 113.7 | 11.2 | 2.621

SF36 | 2015-01-01 00:15:00+0000 | 117.8 | 11.11 | 3.678

SF36 | 2015-01-01 00:20:00+0000 | 117.3 | 11.07 | 2.842

SF36 | 2015-01-01 00:25:00+0000 | 117.3 | 11.07 | 2.629

SF36 | 2015-01-01 00:30:00+0000 | 117.3 | 11.09 | 2.235

SF36 | 2015-01-01 00:35:00+0000 | 117.2 | 11.09 | 2.043

SF36 | 2015-01-01 00:40:00+0000 | 117.2 | 11.05 | 1.635

SF36 | 2015-01-01 00:45:00+0000 | 117.3 | 10.93 | 2.224

SF36 | 2015-01-01 00:50:00+0000 | 112.5 | 10.86 | 1.822

SF36 | 2015-01-01 00:55:00+0000 | 108.7 | 10.8 | 0.866

SF36 | 2015-01-01 01:00:00+0000 | 108.7 | 10.67 | 1.068

SF37 | 2015-01-01 00:00:00+0000 | 252.3 | 11.11 | 3.774

SF37 | 2015-01-01 00:05:00+0000 | 273.89999 | 10.75 | 2.69

SF37 | 2015-01-01 00:10:00+0000 | 299.79999 | 11.1 | 1.747

SF37 | 2015-01-01 00:15:00+0000 | 303.5 | 11.65 | 1.534

SF37 | 2015-01-01 00:20:00+0000 | 282.79999 | 10.27 | 2.269

SF37 | 2015-01-01 00:25:00+0000 | 281.70001 | 9.72 | 2.141

SF37 | 2015-01-01 00:30:00+0000 | 292.70001 | 9.78 | 1.054

SF37 | 2015-01-01 00:35:00+0000 | 280.39999 | 9.53 | 2.36

SF37 | 2015-01-01 00:40:00+0000 | 280.29999 | 9.3 | 2.155

SF37 | 2015-01-01 00:45:00+0000 | 266.10001 | 9.37 | 3.1

SF37 | 2015-01-01 00:50:00+0000 | 272 | 9.46 | 2.703

SF37 | 2015-01-01 00:55:00+0000 | 265.39999 | 9.54 | 3.026

SF37 | 2015-01-01 01:00:00+0000 | 291.60001 | 9.7 | 1.508

(26 rows)

1. So we can query normally can we? Let’s try something else:

select \* from winddata where time <= '2015-01-01 01:00:00';  
  
Uh oh!  
  
InvalidRequest: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"  
  
Basically, Cassandra will not do unbounded time queries, unless you force it to!

1. Try again, but this time explicitly enabling this query.   
   select \* from winddata where time <= '2015-01-01 01:00:00' allow filtering;
2. Now let’s try another query:  
     
   select \* from winddata where time <= '2015-01-01 01:00:00' and temp < 10 ;  
     
   Again this fails. Unlike a normal SQL database, you cannot do arbitrary queries on Cassandra. You must limit your queries to those that can be done based on the primary key. There are ways of creating secondary indices, but these basically create a whole new table under the covers to allow efficient searching.  
     
   That is all, unless you want to explore some advanced features of Cassandra.  
     
   If not, close down your cqlsh window (exit) and your Jupyter notebook and session as before.  
     
   **Extension**
3. First let’s try some JSON support. Try the following:
4. Now let’s insert data using JSON.  
   Notice how we can use either JSON or not.

insert into users json ' {"id": "2", "age": 43, "job": "Teacher", "name": "Henry"} ';

select \* from users;

id | age | job | name

----+-----+---------+-------

2 | 43 | Teacher | Henry

1 | 46 | Student | Paul

(2 rows)

1. Of course JSON supports complex types including lists, maps, sets and other data. Luckily Cassandra does too. Try out the map type with the following commands:

create table demomap ( id int primary key, mapdata map<text,text>);

insert into demomap json   
'{"id":1, "mapdata":{ "key1": "value1","key2":"value2"}}';

select \* from demomap;

select json \* from demomap;

1. Now let’s try out the **set** type.

create table demoset (id int primary key, myset set<text>);

-- insert as json

insert into demoset json ' { "id":1, "myset":["a","b","c"]}';

-- insert in traditional sql style   
insert into demoset (id, myset) values (2, {'hello','paul'});

select \* from demoset;  
select json \* from demoset;

1. CQL also supports a list type. See if you can figure it out. If not, there is an example over the page.
2. List example:

Congratulations - you’ve completed.

create table demolist (id int primary key, list list<text>);

insert into demolist (id, list) values (1,['a1','b2','c3']);

select \* from demolist;

id | list

----+--------------------

1 | ['a1', 'b2', 'c3']

(1 rows)

update demolist set list = ['z1'] + list where id = 1;  
select \* from demolist;

-- what do you expect here?