Exercise 8

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
   a. In a Terminal window (Crtl-Alt-T) type:
   `service cassandra status`
   b. You should see

   ```
   big@big:~$ service cassandra status
   ● cassandra.service - LSB: distributed storage system for structured da
      Loaded: loaded (/etc/init.d/cassandra; bad; vendor preset: enabled)
      Active: active (running) since Fri 2019-03-01 14:31:59 GMT; 2h 12min
        Docs: man:systemd-sysv-generator(8)
     Process: 1800 ExecStart=/etc/init.d/cassandra start (code=exited, sta
      CGroup: /system.slice/cassandra.service
              └─2311 java -Xloggc:/var/log/cassandra/gc.log -ea -XX:+UseTh

   Mar 01 14:31:59 big systemd[1]: Starting LSB: distributed storage syste
   Mar 01 14:31:59 big systemd[1]: Started LSB: distributed storage system
   ```

   c. Type q to get back to the command line
   d. 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:

   ```
   big@big:~$ nodetool status
   Datacenter: datacenter1
   =======================
   Status=Up/Down
   |/ State=Normal/Leaving/Joining/Moving
   --  Address    Load       Tokens       Owns (effective)  Host ID
                              Rack
   UN  127.0.0.1  227.52 KiB  256          100.0%            405b1e36-7393
   -49d9-9d3c-252d5f516829   rack1
   ```

3. You can also try:
   `nodetool info`
   You should see something like:

   ```
   big@big:~$ nodetool info
   ID                     : 405b1e36-7393-49d9-9d3c-252d5f516829
   Gossip active          : true
   Thrift active          : false
   Native Transport active: true
   Load                   : 227.52 KiB
   Generation No          : 1551450738
   Uptime (seconds)       : 8062
   Heap Memory (MB)       : 72.08 / 1976.00
   Off Heap Memory (MB)   : 0.00
   Data Center            : datacenter1
   Rack                   : rack1
   Exceptions             : 0
   Key Cache              : entries 15, size 1.26 KiB, capacity 98 MiB, 65
    hits, 93 requests, 0.699 recent hit rate, 14400 save period in seconds
   Row Cache              : entries 0, size 0 bytes, capacity 0 bytes, 0 h
   its, 0 requests, NaN recent hit rate, 0 save period in seconds
   Counter Cache          : entries 0, size 0 bytes, capacity 49 MiB, 0 hi
   ts, 0 requests, NaN recent hit rate, 7200 save period in seconds
   Chunk Cache            : entries 1, size 64 KiB, capacity 462 MiB, 78 m
   isses, 213 requests, 0.634 recent hit rate, NaN microseconds miss laten
   cy
   Percent Repaired       : 100.0%
   Token                  : (invoke with -T/--tokens to see all 256 tokens
   )
   ```

4. Now you can start the Cassandra Shell:
   Type:
   `cqlsh`

   You should see:

   ```
   big@big:~$ cqlsh
   Connected to Test Cluster at 127.0.0.1:9042.
   [cqlsh 5.0.1 | Cassandra 3.11.1 | CQL spec 3.4.4 | Native protocol v4]
   Use HELP for help.
   cqlsh>
   ```

   Let's create a new database (Keyspace):
   a. Type (all on a single line)

   ```
   CREATE KEYSPACE TEST WITH REPLICATION = { 'class' :
   'SimpleStrategy', 'replication_factor' : 1 };
   ```

    b. Check it worked:
        Type:

```
desc keyspace test;
```

    c. You should see:

```
CREATE KEYSPACE test WITH replication = {'class':
'SimpleStrategy', 'replication_factor': '1'}  AND
durable_writes = true;
```

5. Now we need to select to use that keyspace:
```
use test;
```

6. The command prompt will change to:
```
cqlsh:test>
```

7. Let's create a simple (key, value) table
    a. Type:
```
create table kv ( key text, value text, primary key (key));
```

    b. Now type
```
desc kv;
```

    c. 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.SizeTieredCompactionStra
tegy'}
    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';
```

    d. 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');
```

    e. Now type:
```
select * from kv;
```

You should see:
```
key | value
-----+-------
   a |     1
   c |     3
   b |     2

(3 rows)
```

8. You can also do other simple SQL of course

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

 key | value
-----+-------
   a |     1

(1 rows)
```

9. Type **exit** to leave the cqlsh command line.

PART B – Loading data from CSV files into Cassandra

10. 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

```
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)
);
```

11. 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.

12. 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.

13. 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.

Start Jupyter as before. Create a new Python2 notebook and copy code from
`~/BigData/code_jw/starters/big-import.py`
to the notebook.

Let's look at it line by line.

```
1   import findspark
2   findspark.init()
3
4   import time
5   from dateutil import parser
6
7   from pyspark.sql import Row, SparkSession
8   spark = SparkSession.builder.getOrCreate()
9   df = spark.read.csv('/home/big/sql/*.csv', header=True)
10
11  df.show(1)
12
13  convertTime = lambda t: parser.parse(t.replace('?',' '))
14
15  clean = lambda s: Row(stationid=s.Station_ID,
16      time=convertTime(s.Interval_End_Time),
17      direction=s.Wind_Direction_Deg,
18      temp=s.Ambient_Temperature_Deg_C,
19      velocity=s.Wind_Velocity_Mtr_Sec)
20
21  newDF = df.rdd.map(clean).toDF()
22
23  newDF.show(1)
24
25  newDF.write.format("org.apache.spark.sql.cassandra").mode('append').options(table="winddata", keyspace="wind").save()
26
27  print('done')
```

You should recognize lines 1 and 2.
Line 4-5: dateutil.parser is a useful utility that can read most common date formats.
Lines 7-9: these are just as in the SQL exercise.

Line 11: This will print out one line of the data we've read in so we can see the format.

Line 13. 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 15: 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 21: This converts from a dataframe to an RDD, uses map to apply our clean function, and then converts back to a dataframe.

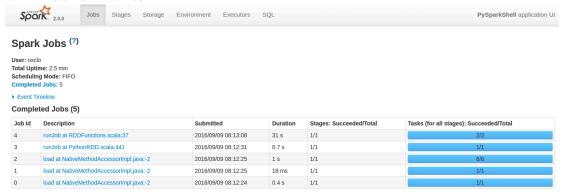Line 23: This shows our more beautiful dataframe layout.

Line 25: This exports the dataframe to Cassandra, specifying the database and table to use.

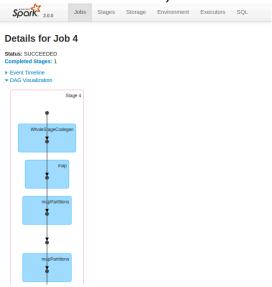14. Run the cell. This will take a bit of time.

15. You should see the following:

```
+---------+------------------+--------------+---------------+----------------+---------------------+----------
-----------------+------------------+------------------------+----------------------------+
|Station_ID|      Station_Name|Location_Label|Interval_Minutes|Interval_End_Time|Wind_Velocity_Mtr_Sec|Wind_Direc
tion_Variance_Deg|Wind_Direction_Deg|Ambient_Temperature_Deg_C|Global_Horizontal_Irradiance|
+---------+------------------+--------------+---------------+----------------+---------------------+----------
-----------------+------------------+------------------------+----------------------------+
|     SF15|Warnerville Switc...|   Warnerville|              5| 2015-01-5? 00:05|               1.628|
       8.1|             148.5|                    0.92|                       0.061|
+---------+------------------+--------------+---------------+----------------+---------------------+----------
-----------------+------------------+------------------------+----------------------------+
only showing top 1 row

+---------+---------+----+-------------------+--------+
|direction|stationid|temp|               time|velocity|
+---------+---------+----+-------------------+--------+
|    148.5|     SF15|0.92|2015-01-05 00:05:00|   1.628|
+---------+---------+----+-------------------+--------+
only showing top 1 row

done
```

16. Browse to http://localhost:4040
It will look similar to:

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	runJob at RDDFunctions.scala:37	2016/09/09 08:13:08	31 s	1/1	2/2
3	runJob at PythonRDD.scala:441	2016/09/09 08:12:31	0.7 s	1/1	1/1
2	load at NativeMethodAccessorImpl.java:-2	2016/09/09 08:12:25	1 s	1/1	6/6
1	load at NativeMethodAccessorImpl.java:-2	2016/09/09 08:12:25	18 ms	1/1	1/1
0	load at NativeMethodAccessorImpl.java:-2	2016/09/09 08:12:24	0.4 s	1/1	1/1

Spark Jobs (?)

User: oxclo
Total Uptime: 2.5 min
Scheduling Mode: FIFO
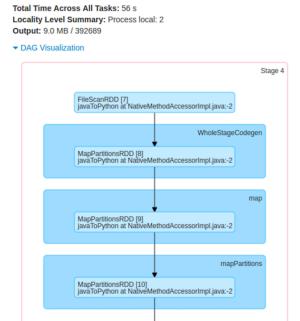**Completed Jobs:** 5
▸ Event Timeline
Completed Jobs (5)

17. Click on the most recent job:

18. You can also get more details by clicking on a stage in the DAG (Directed Acyclic Graph) picture:

19. Check that the data has loaded. Start another terminal window and restart **cqlsh**.

20. In your **cqlsh** window type:

```
select * from wind.winddata limit 15;
```

21. 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)

22. 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:

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)

23. 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)
    ```

24. 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!

25. Try again, but this time explicitly enabling this query.
    ```
    select * from winddata where time <= '2015-01-01 01:00:00' allow
    filtering;
    ```

26. 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.

```
CREATE KEYSPACE jsontest WITH REPLICATION =
  { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };

use jsontest;
create table users  (id text primary key, name text, age int , job text);
insert into users (id, name, age, job) values ('1', 'Paul', 46, 'Student');
select json * from users;
```

You should see:
```
[json]
---------------------------------------------------------
 {"id": "1", "age": 46, "job": "Student", "name": "Paul"}

(1 rows)
```

If not, close down your cqlsh window (exit) and your Jupyter notebook and session as before.

Extension

First let's try some JSON support. Try the following:

27. Now let's insert data using JSON.

    ```
    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)
    ```

    Notice how we can use either JSON or not.

28. 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;
    ```

29. 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;
    ```

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

List example:

```
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?
```

Congratulations - you've completed this lab and extensions.