

# Instructions for Downloading Hands On Datasets

---

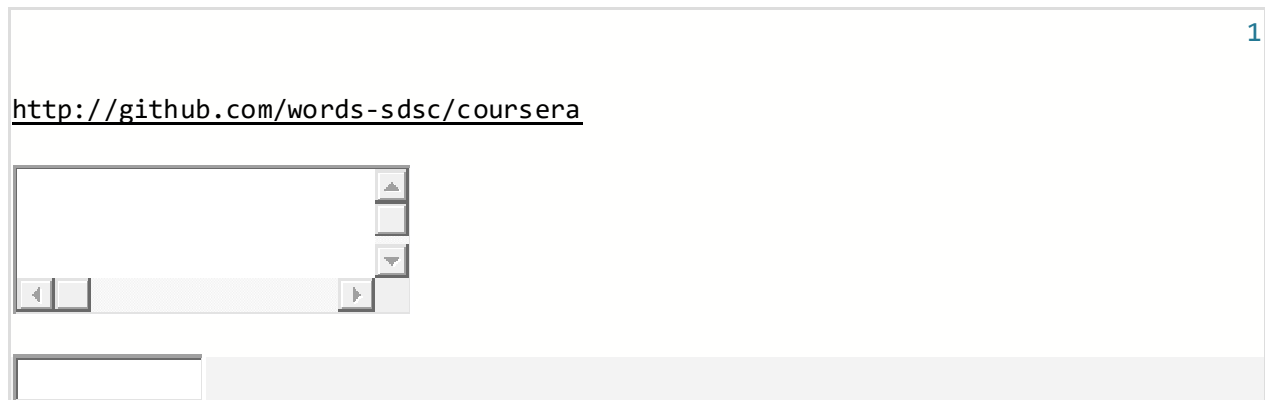
This Reading describes the steps to download the datasets and scripts necessary for the Hands On exercises in this course. Before proceeding, make sure you have downloaded and installed VirtualBox and the Cloudera virtual machine. Instructions for this process can be found in the Reading called *Downloading and Installing the Cloudera VM*.

Step 1. **Start the Cloudera VM.** Most of the Hands On exercises in this course use the Cloudera Virtual Machine, so we will download the datasets onto the VM. Start the VM in VirtualBox and perform the remaining steps in the VM.

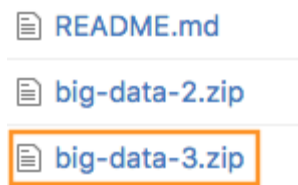
Step 2. **Open a web browser.** Open a web browser by clicking on the web browser icon in the top toolbar.



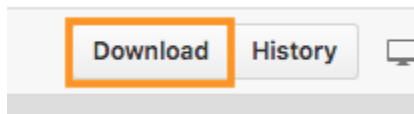
In the web browser, enter the following for the URL:



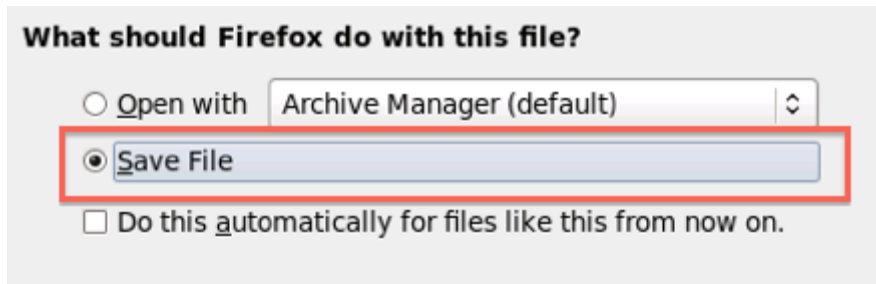
Step 3. **Download the datasets.** Click on *big-data-3.zip*:



Click on the *Download* button:



In the dialog, select *Save File*:



Click *OK*, and the file *big-data-3.zip* file will be downloaded to the Downloads directory.

Step 4. **Uncompress the datasets.** Open a terminal shell by clicking on the terminal shell icon in the top toolbar.



In the terminal, run:

```
1  
2  
  
cd Downloads  
  
unzip -o big-data-3.zip  
  
[Terminal window showing a file explorer view]
```

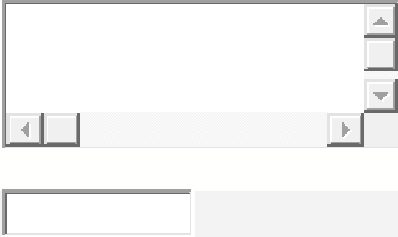
Step 5. **Install tools.** Change directories to *big-data-3* and run *setup.sh* to install tools and libraries.

1

2

```
cd big-data-3

./setup.sh
```



During the setup process, Anaconda will give you a series of prompts. First, press *enter* to continue the installation:

```
Welcome to Anaconda3 4.0.0 (by Continuum Analytics, Inc.)

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>> █
```

Next, read and accept the license:

```
Do you approve the license terms? [yes|no]
>>> █
```

Next, press *enter* to accept the default installation location:

```
Anaconda3 will now be installed into this location:
/home/cloudera/anaconda3

- Press ENTER to confirm the location
- Press CTRL-C to abort the installation
- Or specify a different location below

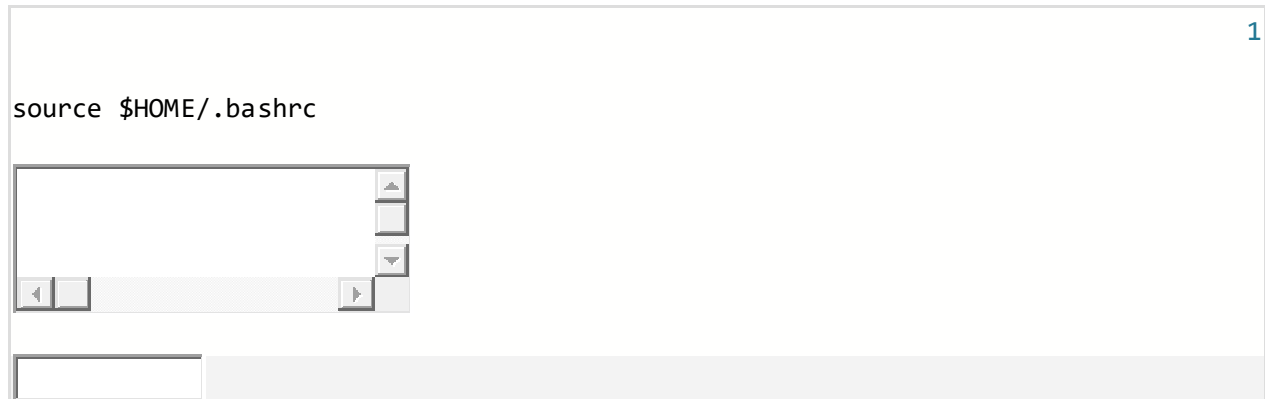
[/home/cloudera/anaconda3] >>> █
```

Next, enter *yes* when it asks if you want to prepend the install location to PATH:

```
installation finished.  
Do you wish the installer to prepend the Anaconda3 install location  
to PATH in your /home/cloudera/.bashrc ? [yes|no]  
[no] >>> yes
```

The setup of tools and datasets should continue.

Finally, source \$HOME/.bashrc:



## Querying Relational Data with Postgres

---

By this end of this activity, you will be able to:

1. View table and column definitions, and perform SQL queries in the Postgres shell
2. Query the contents of SQL tables
3. Filter table rows and columns
4. Combine two tables by joining on a column

Step 1. **Open a terminal window and start Postgres shell.** Open a terminal window by clicking on the square black box on the top left of the screen.



Next, start the Postgres shell by running *psql*:

```
[cloudera@quickstart big-data-3]$ psql
psql (8.4.20)
Type "help" for help.
```

```
cloudera=#
```

Step 2. **View table and column definitions.** We can list the tables in the database with the *\d* command:

```
cloudera=# \d
List of relations
Schema |      Name      | Type  | Owner
-----+-----+-----+-----
public | adclicks       | table | cloudera
public | buyclicks      | table | cloudera
public | gameclicks     | table | cloudera
(3 rows)
```

The database contains three tables: *adclicks*, *buyclicks*, and *gameclicks*. We can see the column definitions of the *buyclicks* table by running *\d buyclicks*:

```
cloudera=# \d buyclicks
Table "public.buyclicks"
Column      |      Type      | Modifiers
-----+-----+-----
timestamp   | timestamp without time zone | not null
txid         | integer         | not null
usersessionid | integer         | not null
team         | integer         | not null
userid       | integer         | not null
buyid        | integer         | not null
price        | double precision | not null
```

This shows that the *buyclicks* table has seven columns, and what each column name and data type is.

Step 3. **Query table.** We can run the following command to view the contents of the *buyclicks* table:

```
select * from buyclicks;
```

The *select \** means we want to query all the columns, and *from buyclicks* denotes which table to query. Note that all query commands in the Postgres shell must end with a semi-colon.

The result of the query is:

timestamp	txid	usersessionid	team	userid	buyid	price
2016-05-26 15:36:54	6004	5820	9	1300	2	3
2016-05-26 15:36:54	6005	5775	35	868	4	10
2016-05-26 15:36:54	6006	5679	97	819	5	20
2016-05-26 16:36:54	6067	5665	18	121	2	3
2016-05-26 17:06:54	6093	5709	11	2222	5	20
2016-05-26 17:06:54	6094	5798	77	1304	5	20
2016-05-26 18:06:54	6155	5920	9	1027	5	20
2016-05-26 18:06:54	6156	5697	35	2199	2	3
2016-05-26 18:36:54	6183	5893	64	1544	5	20
2016-05-26 18:36:54	6184	5697	35	2199	1	2
2016-05-26 19:36:54	6243	5659	13	1623	4	10

You can hit *<space>* to scroll down, and *q* to quit.

Step 4. **Filter rows and columns.** We can query only the *price* and *userid* columns with the following command:

1

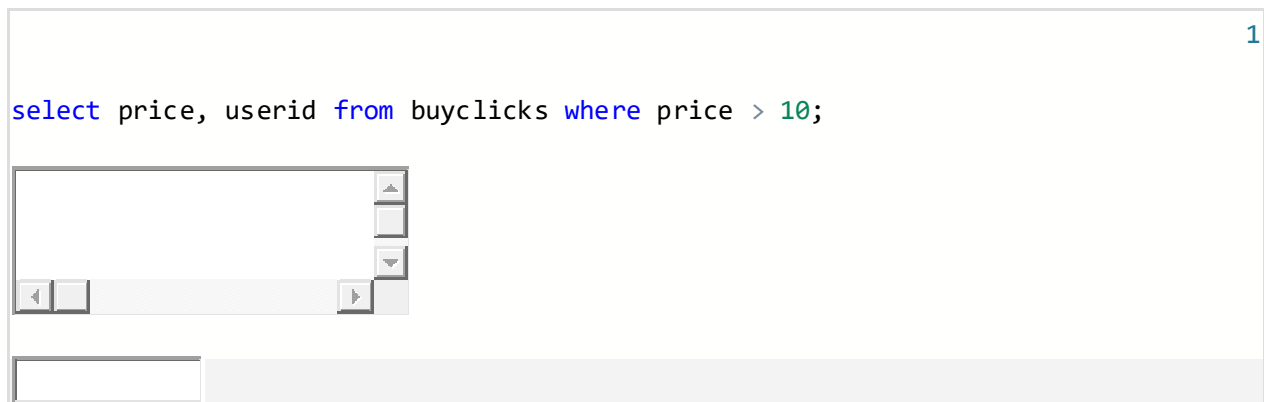
```
select price, userid from buyclicks;
```

The result of this query is:

price	userid
3	1300
10	868
20	819
3	121
20	2222
20	1304
20	1027
3	2199
20	1544

We can also query rows that match a specific criteria. For example, the following command queries only rows with a price greater than 10:

```
1
select price, userid from buyclicks where price > 10;
```



The result is:

price	userid
20	819
20	2222
20	1304
20	1027
20	1544
20	1065
20	2221

Step 5. **Perform aggregate operations.** The SQL language provides many aggregate operations. We can calculate the average price:

```
cloudera=# select avg(price) from buyclicks;
avg
-----
7.26399728537496
(1 row)
```

We can also calculate the total price:

```
cloudera=# select sum(price) from buyclicks;
sum
-----
21407
(1 row)
```

The complete list of aggregate functions for Postgres 8.4 (the version installed on the Cloudera VM) can be found here: <https://www.postgresql.org/docs/8.4/static/functions-aggregate.html>

Step 6. **Combine two tables.** We combine the contents of two tables by matching or joining on a single column. If we look at the definition of the *adclicks* table:

```
cloudera=# \d adclicks
Table "public.adclicks"
  Column          |          Type          | Modifiers
-----+-----+-----
timestamp        | timestamp without time zone | not null
txid              | integer                | not null
usersessionid    | integer                | not null
teamid           | integer                | not null
userid           | integer                | not null
adid             | integer                | not null
adcategory       | character varying(11)   | not null
```

We see that *adclicks* also has a column named *userid*. The following query combines the *adclicks* and *buyclicks* tables on the *userid* column in both tables:

1

2

```
select adid, buyid, adclicks.userid
from adclicks join buyclicks on adclicks.userid = buyclicks.userid;
```

This query shows the columns *adid* and *userid* from the *adclicks* table, and the *buyid* column from the *buyclicks* table. The *from adclicks join buyclicks* denotes that we want to combine these two tables, and *on adclicks.userid = buyclicks.userid* denotes which two columns to use when the tables are combined.



The result of the query is:

adid	buyid	userid
2	5	611
2	4	611
2	4	611
2	5	611
2	4	611
2	1	611
21	1	1874
21	1	1874
21	3	1874
21	1	1874
21	2	1874

Enter `lq` to quit the Postgres shell.

Complete

## Querying Documents in MongoDB

By the end of this activity, you will be able to:

1. Find documents in MongoDB with specific field values.
2. Filter the results returned by MongoDB queries.
3. Count documents in a MongoDB collection and returned by queries.

Step 1. **Start MongoDB server and MongoDB shell.** Open a terminal window by clicking on the square black box on the top left of the screen.



Next, change to the *mongodb* directory, and start the server:

```
1  
2  
cd Downloads/big-data-3/mongodb
```

```
./mongodb/bin/mongod --dbpath db
```



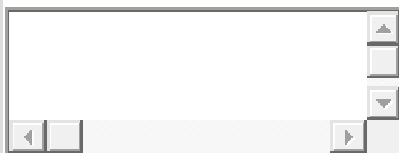
The arguments `--dbpath db` specify that the directory `db` should be used for the MongoDB directory for datafiles. After starting the MongoDB server, you will see the following lines indicating that the server is running:

```
I FTDC      [initandlisten] Initializing full-time diagnostic data capture with directory 'db/diagnostic.data'
I NETWORK   [HostnameCanonicalizationWorker] Starting hostname canonicalization worker
I NETWORK   [initandlisten] waiting for connections on port 27017
```

Next, let's run the MongoDB shell so that we can query the server. Open a new terminal shell window, change to the `mongodb` directory, and start the shell:

```
cd Downloads/big-data-3/mongodb
```

```
./mongodb/bin/mongo
```



1  
2

Step 2. **Show Databases and Collections.** Run the *show dbs* command to see the databases:

```
> show dbs
journaldev  0.000GB
local       0.000GB
sample      0.004GB
test        0.000GB
```

The database named *sample* has been created and loaded with Twitter JSON data. Let's switch to that database by running the *use* command:

```
> use sample
switched to db sample
```

We can see the collections in the *sample* database by running *show collections*:

```
> show collections
collection
users
```

The Twitter data is stored in the *users* collection. We can run *db.users.count()* to count the number of documents:

```
> db.users.count()
11188
```

Step 3. **Look at document and find distinct values.** We can examine the contents of one of the documents by running *db.users.findOne()*:

```
> db.users.findOne()
{
  "_id" : ObjectId("578ffa8e7eb9513f4f55a935"),
  "user_name" : "koterass",
  "retweet count" : 0,
  "tweet_followers count" : 461,
  "source" : "<a href='\"http://twitter.com/download/iphone\"' rel='\"nofollow\"'>Twitter for iPhone</a>",
  "coordinates" : null,
  "tweet_mentioned count" : 1,
  "tweet_ID" : "755891629932675072",
  "tweet_text" : "RT @ochocinco: I beat them all for 10 straight hours #FIFA16KING https://t.co/BFnV6jfkBL",
  "user" : {
    "CreatedAt" : ISODate("2011-12-27T09:04:01Z"),
    "FavouritesCount" : 5223,
    "FollowersCount" : 461,
    "FriendsCount" : 619,
    "UserId" : 447818090,
    "Location" : "501"
  }
}
```

The document has several fields, e.g., *user\_name*, *retweet\_count*, *tweet\_ID*, etc., and nested fields under *user*, e.g., *CreatedAt*, *UserId*, *Location*, etc.

We can find the distinct values for a specific field by using the *distinct()* command. For example, let's find the distinct values for *user\_name*:

```
> db.users.distinct("user_name")
[
  "koteras",
  "AllieLovesR5_1D",
  "Tonkatol",
  "Gaslet",
  "Syaxmii ",
  "CamSteele_96",
```

Step 4. **Search for specific field value.** We can search for fields with a specific value using the *find()* command. For example, let's search for *user\_name* with the value *ActionSportsJax*:

```
> db.users.find({user_name : "ActionSportsJax"})
{ "_id" : ObjectId("579670bfc38159226b4c8e47"), "user_name" : "ActionSportsJax", "retweet
rce" : "<a href=\"http://twitter.com/download/iphone\" rel=\"nofollow\">Twitter for iPhon
unt" : 2, "tweet_ID" : "757667800521531393", "tweet_text" : "RT @wwbrown19: I'm watching
Augustine football and asked myself \"How on earth did we stop...", "user" : { "CreatedAt"
nt" : 120, "FollowersCount" : 3539, "FriendsCount" : 476, "UserId" : 35857042, "Location"
```

By appending *.pretty()* to the end of the find command, the results will be formatted:

```
> db.users.find({user_name : "ActionSportsJax"}).pretty()
{
  "_id" : ObjectId("579670bfc38159226b4c8e47"),
  "user_name" : "ActionSportsJax",
  "retweet_count" : 0,
  "tweet_followers_count" : 3539,
  "source" : "<a href=\"http://twitter.com/download/i
  \"coordinates\" : null,
  "tweet_mentioned_count" : 2,
  "tweet_ID" : "757667800521531393",
```

Step 5. **Filter fields returned by query.** We can specify a second argument to the *find()* command to only show specific field(s) in the result. Let's repeat the previous search, but only show the *tweet\_ID* field:

```
> db.users.find({user_name: "ActionSportsJax"}, {tweet_ID: 1})
{ "_id" : ObjectId("579670bfc38159226b4c8e47"), "tweet_ID" : "757667800521531393" }
```

The *\_id* field is primary key for every document, and we can remove it from the results with the following filter:

```
> db.users.find({user_name: "ActionSportsJax"}, {tweet_ID: 1, _id: 0})
{ "tweet_ID" : "757667800521531393" }
```

Step 6. **Perform regular expression search.** MongoDB also supports searching documents with regular expressions. If we search for the value *FIFA* in the *tweet\_text* field, there are no results:

```
> db.users.find({tweet_text: "FIFA"})
> █
```

However, if we search using a regular expression, there are many results:

```
> db.users.find({tweet_text: /FIFA/})
{ "_id" : ObjectId("578ffa8e7eb9513f4f55a935"), "user_name" : "koterass", "retweeted" : false, "tweet_text" : "RT @GameSeek: Follow & Retweet for your chance to win a iPhone 4S (your choice) in our #giveaway! https://...", "user" : { "CreatedAt" : ISODate("2012-12-27T09:04:01Z"), "FavouritesCount" : 447818090, "Location" : "501" } }
{ "_id" : ObjectId("578ffa917eb9513f4f55a939"), "user_name" : "Tonkatol", "retweeted" : false, "tweet_text" : "RT @GameSeek: Follow & Retweet for your chance to win a iPhone 4S (your choice) in our #giveaway! https://...", "user" : { "CreatedAt" : ISODate("2012-12-27T09:04:01Z"), "FavouritesCount" : 616, "FriendsCount" : 2675, "UserId" : 722815650, "Location" : null } }
```

The difference between the queries is that the first searched for where the *tweet\_text* field value was exactly equal to *FIFA*, and the second searched for where the field value contained *FIFA*.

We can append *.count()* to the command to count the number of results:

```
> db.users.find({tweet_text: /FIFA/}).count()
3697
```

Step 7. **Search using text index.** A text index can be created to speed up searches and allows advanced searches with *\$text*. Let's create the index using *createIndex()*:

```
> db.users.createIndex({"tweet_text" : "text"})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 2,
  "numIndexesAfter" : 2,
  "note" : "all indexes already exist",
  "ok" : 1
}
```

The argument *tweet\_text* specifies the field on which to create the index.

Next, we can use the *\$text* operator to search the collection. We can perform the previous query to find the documents containing *FIFA*:

```
> db.users.find({$text : {$search : "FIFA"}}).count()
4031
```

We can also search for documents not containing a specific value. For example, let's search for documents containing *FIFA*, but not *Texas*:

```
> db.users.find({$text : {$search : "FIFA -Texas"}}).count()
4022
```

Step 8. **Search using operators.** MongoDB can also search for field values matching a specific criteria. For example, we can find where the *tweet\_mentioned\_count* is greater than six:

```
> db.users.find({tweet_mentioned_count: {$gt : 6}})
{ "_id" : ObjectId("57966e26c3815920e1131b03"), "user_name" : "marshallrupe",
  : "<a href=\"http://twitter.com/download/iphone\" rel=\"nofollow\">Twitter f
  : 7, "tweet_ID" : "757665013817409536", "tweet_text" : "RT @BrianBorrison: h
  lrupe @da_carlos30 @TrissyTim @BrockHiltonb @ZachZippe @ro...", "user" : { "Cre
  : 326, "FollowersCount" : 215, "FriendsCount" : 165, "UserId" : -2073286695,
  { "_id" : ObjectId("57966ecfc3815920e1132053"), "user_name" : "marvheys", "re
  "<a href=\"http://twitter.com/download/android\" rel=\"nofollow\">Twitter for
  : 7, "tweet_ID" : "757665717948977152", "tweet_text" : "RT @philhayton11: Ple
  1882 @marvheys @Matlalar https://t.co/y0s8MxjA5d", "user" : { "CreatedAt" :
  ollowersCount" : 1215, "FriendsCount" : 850, "UserId" : 477884041, "Location"
  { "_id" : ObjectId("5796706bc38159226b4c8af1"), "user_name" : "HearAllAbtIt",
  "<a href=\"http://twitter.com\" rel=\"nofollow\">Twitter Web Client</a>", "c
```

The *\$gt* operator search for values greater than a specific value. We can use the *\$where* command to compare between fields in the same document. For example, the following searches for *tweet\_mentioned\_count* is greater than *tweet\_followers\_count*:

```
> db.users.find({$where : "this.tweet_mentioned_count > this.tweet_followers_count"}).count()
18
```

Note that the field names for *\$where* are required to be prefixed with *this*, which represent the document.

We can combine multiple searches by using *\$and*. For example, let's search for *tweet\_text* containing *FIFA* and *tweet\_mentioned\_count* greater than four:

```
> db.users.find({$and : [ {tweet_text : /FIFA/}, {tweet_mentioned_count: {$gt : 4}}]}).count()
1
```

When you are done querying MongoDB, run *exit* in the MongoDB shell, and *Control-C* in the terminal window running the server.

# Exploring Pandas DataFrames

---

By the end of this activity, you will be able to:

1. Read a CSV file into a Pandas DataFrame
2. View the contents and shape of a DataFrame
3. Filter rows and columns of a DataFrame
4. Calculate the average and sum of a column in a DataFrame
5. Combine two DataFrames by joining on a single column

This activity consists of programming in a Jupyter Python Notebook. If you have not already started the Jupyter server, follow the instructions in the Reading entitled *Starting Jupyter for Python Notebooks*.

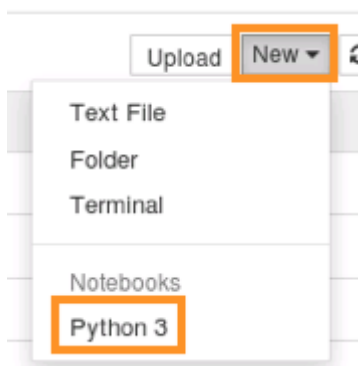
Step 1. **Open a web browser and create a new Jupyter Python Notebook.** Open a web browser by clicking on the web browser icon at the top of the toolbar:



Navigate to `localhost:8889/tree/Downloads/big-data-3`.



Create a new Python Notebook by clicking on *New*, and then click on *Python 3*:



Step 2. **Load Pandas and Read a CSV file into a DataFrame.** We first load the Pandas library:

```
In [1]: import pandas
```

Note that to execute commands in Jupyter Notebooks, hold the *<shift>* key and press *<enter>*.

We can load the file *buy-clicks.csv* into a Pandas DataFrame:

```
In [2]: buyclicksDF = pandas.read_csv('buy-clicks.csv')
```

This command assigns the DataFrame to a new variable named *buyclicksDF*, and reads the CSV using *pandas.read\_csv()*.

Step 3. **View the contents and shape of a DataFrame.** We can view the contents of the DataFrame by executing the variable:

```
In [3]: buyclicksDF
```

```
Out[3]:
```

	timestamp	txId	userSessionId	team	userId	buyId	price
0	2016-05-26 15:36:54	6004	5820	9	1300	2	3.0
1	2016-05-26 15:36:54	6005	5775	35	868	4	10.0
2	2016-05-26 15:36:54	6006	5679	97	819	5	20.0
3	2016-05-26 16:36:54	6067	5665	18	121	2	3.0



Note that the Notebook does not display all the rows and displays the missing ones as .... :

28	2016-05-27 02:06:54	6567	5860	57	2221	2	3.0
29	2016-05-27 03:36:54	6651	5955	64	2009	3	5.0
...	...	...	...	...	...	...	...
2917	2016-06-16 08:06:54	39557	34632	72	1294	0	1.0
2918	2016-06-16 08:06:54	39558	34498	59	2029	3	5.0

We can view the first five rows by using the `head(5)` command:

```
In [4]: buyclicksDF.head(5)
```

```
Out[4]:
```

	timestamp	txId	userSessionId	team	userId	buyId	price
0	2016-05-26 15:36:54	6004	5820	9	1300	2	3.0
1	2016-05-26 15:36:54	6005	5775	35	868	4	10.0
2	2016-05-26 15:36:54	6006	5679	97	819	5	20.0
3	2016-05-26 16:36:54	6067	5665	18	121	2	3.0
4	2016-05-26 17:06:54	6093	5709	11	2222	5	20.0

We can see how many rows and columns are in the DataFrame by looking at its shape:

```
In [5]: buyclicksDF.shape
```

```
Out[5]: (2947, 7)
```

The result says that there are 2947 rows and 7 columns.

Step 4. **Filter rows and columns of a DataFrame.** We can view only the *price* and *userId* columns of the DataFrame:

```
In [6]: buyclicksDF[['price', 'userId']].head(5)
```

Out[6]:

	price	userId
0	3.0	1300
1	10.0	868
2	20.0	819
3	3.0	121
4	20.0	2222

The `[[ ]]` creates a copy of the DataFrame with only the specified columns.

We can also filter rows based on a criteria. The following selects rows with a price less than 3:

```
In [7]: buyclicksDF[buyclicksDF['price'] < 3].head(5)
```

Out[7]:

	timestamp	txId	userSessionId	team	userId	buyId	price
9	2016-05-26 18:36:54	6184	5697	35	2199	1	2.0
14	2016-05-26 20:06:54	6271	5706	9	1652	0	1.0
15	2016-05-26 20:36:54	6292	5921	2	518	0	1.0
18	2016-05-26 22:06:54	6395	5880	35	2146	1	2.0
19	2016-05-26 22:36:54	6411	6230	77	1457	0	1.0

Step 5. **Calculate sum and average of a column.** Pandas DataFrames provide many aggregation operations. We can calculate the total price:

```
In [8]: buyclicksDF['price'].sum()
```

Out[8]: 21407.0

We can also calculate the average price:

```
In [9]: buyclicksDF['price'].mean()  
Out[9]: 7.263997285374957
```

A complete list of statistical aggregation operations for Pandas DataFrames is at <http://pandas.pydata.org/pandas-docs/stable/api.html#computations-descriptive-stats>

Step 6. **Combine two DataFrames.** We can combine two DataFrames on a single column. First, we will load *ad-clicks.csv* into a new DataFrame:

```
In [10]: adclicksDF = pandas.read_csv('ad-clicks.csv')
```

If we look at the contents, we see that *adclicksDF* also has a column named *userId*:

```
In [11]: adclicksDF.head(5)
```

```
Out[11]:
```

	timestamp	txId	userSessionId	teamId	userId	adId	adCategory
0	2016-05-26 15:13:22	5974	5809	27	611	2	electronics
1	2016-05-26 15:17:24	5976	5705	18	1874	21	movies
2	2016-05-26 15:22:52	5978	5791	53	2139	25	computers
3	2016-05-26 15:22:57	5973	5756	63	212	10	fashion
4	2016-05-26 15:22:58	5980	5920	9	1027	20	clothing

We can create a combine *buyclicksDF* and *adclicksDF* on the *userId* column with the following command:

```
In [12]: mergedDF = adclicksDF.merge(buyclicksDF, on='userId')
```

The combined DataFrame is assigned to a new variable named *mergedDF*. The command *adclicks.merge()* combines *adclicksDF* with the first argument *buyclicksDF*, and *on='userId'* denotes which column to join on.

## Exploring Splunk Queries

---

By the end of this activity, you will be able to:

- Import CSV files into Splunk.
- Query, filter, and plot data.
- Perform statistical calculations.

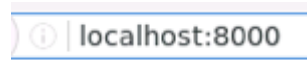
**NOTE: Steps 4 and 5 below contain examples using the 'sort' command which are not covered in the video lecture but which will be covered in the accompanying quiz.**

The Census CSV data used in this activity can be downloaded here:

[census.zip](#)

After downloading, unzip the file.

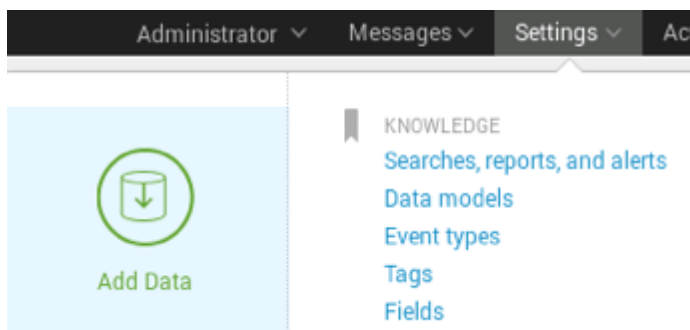
Step 1. **Login to Splunk.** Open a web browser and navigate to *localhost:8000*:



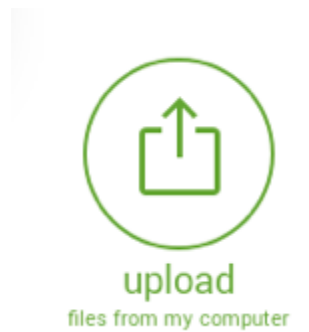
Next, login to Splunk by enter *admin* and the default password *changeme*:



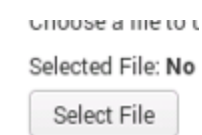
Step 2. **Import census data.** Let's import the census data CSV file to Splunk. First, click on *Settings* in the top right, then click on *Add Data*:



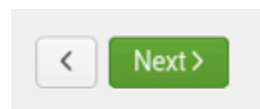
Next, click on *Upload*:



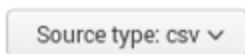
Click on *Select File*:



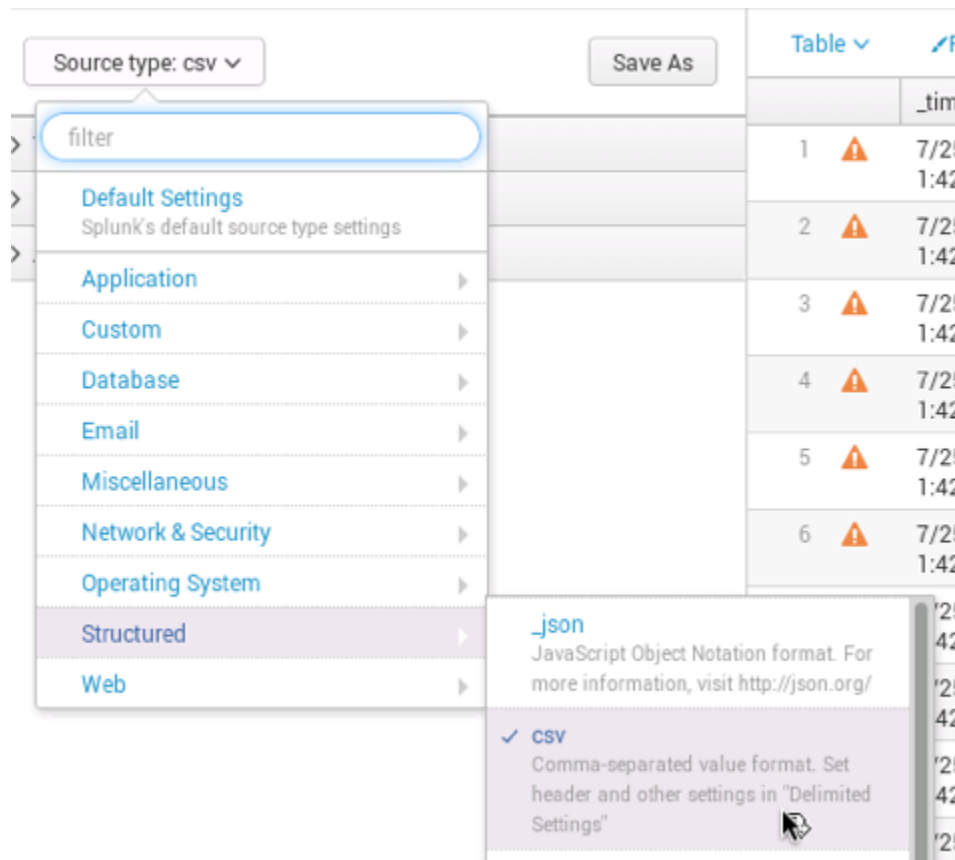
Navigate to *census.csv*, and select it. Then click *Next>*:



On the left, make sure the *Source type* is *csv*.



If the *Source type* is not *csv*, click on *Source type*, go down to *Structured*, and select *csv*.



The table on the right is a preview of the CSV data. It shows the values for different fields:

		_time	BIRTHS2010	BIRTHS2011	BIRTHS2012	BIRTHS2013	BIRTHS2014	BIRTHS2015	CENSUS2010POP
1	⚠	7/25/16 1:42:30.000 PM	14226	59689	59062	57938	58334	58305	4779736
2	⚠	7/25/16 1:42:30.000 PM	151	636	615	574	623	600	54571
3	⚠	7/25/16 1:42:30.000 PM	517	2187	2092	2160	2186	2240	182265

Next, click on *Next*, click on *Review*, and then click *Submit*. Splunk will say the file is successfully uploaded:

✓ File has been uploaded successfully.

Step 3. **View data**. Click on *Start Searching*:

Start Searching

Splunk will enter a default query in the search box to show the data we just imported:

```
source="census.csv" sourcetype="csv"
```

✓ 3,193 events (before 7/25/16 1:51:42.000 PM)

This query shows all the data from the *census.csv* file and whose data type is CSV. In general, we can use *source=* to query from different file names, and *sourcetype=* to query from different formats.

The table shows the results matching this query:

List ▾ Format ▾ 20 Per Page ▾

i	Time	Event
>	7/25/16 1:48:34.000 PM	050,4,8,56,045,Wyoming,Weston County,7208,7208,7181,7114,7065,7160,7185,7234,-27,-67,-49,6,7,2,1,-2,0,0,0,0,-41,-84,-57,88,11,50,-40,-86,-57,88,11,50,-4,9,1,-9,7,-3,313,313,313,3,10.735447891,10.957764061,9.9335431969,9.4505959518,10.826010545,9.759498083,10.68035231,0.2774117484,-0.279818118,0,0,0,0,-11.75236097,-8.040059243,12.37258348,1.533635413,6.9335413,6.9352937097 host = florian   source = census.csv   sourcetype = csv
>	7/25/16 1:48:34.000 PM	050,4,8,56,043,Wyoming,Washakie County,8533,8533,8545,8469,8443,8443,8316,8328,12,-76,-26,9,-15,18,26,11,1,-3,-3,-2,-2,-2,20,-99,-7,-17,-149,14,21,-102,-10,-19,-151,12,-1,-3,-1,1,333018,11.251924671,11.456530819,10.814708003,9.2864699659,12.417218543,9.1199810494,8.35436219,3.1028104302,1.3217976448,-0.352650758,-0.354777673,-0.236882625,-0.238677725,-0.249054,1.6822879116,-11.99012578,-1.182592242,-2.250384934,-18.02016827,1.4419610671 host = florian   source = census.csv   sourcetype = csv

Step 4. **Filtering for specific values.** We can filter the results by looking for a field with a specific value. For example, we can find the entries where the state is California:

```
STNAME="California"
```

The field *STNAME* contains the name of the state, and the above query only shows the results where the state is California. We can use an *OR* to search for multiple values on the same field:

```
STNAME="California" OR STNAME="Alaska"
```

We can search multiple fields with specific values by adding them to query. For example, let's search for state name California and 2010 population greater than one million people:

```
STNAME="California" CENSUS2010POP > 1000000
```

We can filter our results to just show a single column. For example, let's just show the county names of the previous query:

```
STNAME="California" CENSUS2010POP > 1000000 | table CTYNAME
```

The | (pipe) is the syntax for sending the results from one query to the next, and the *table* command creates a table using only the specified column name(s).

We can sort the results based on any of the fields, such as population, and order them in either ascending or descending order. The image below shows an example of a search for all items with a population greater than 100000, sorts the results in descending order, and creates a table containing the population and state name. [To sort in ascending order you would replace "desc" with "asc"].

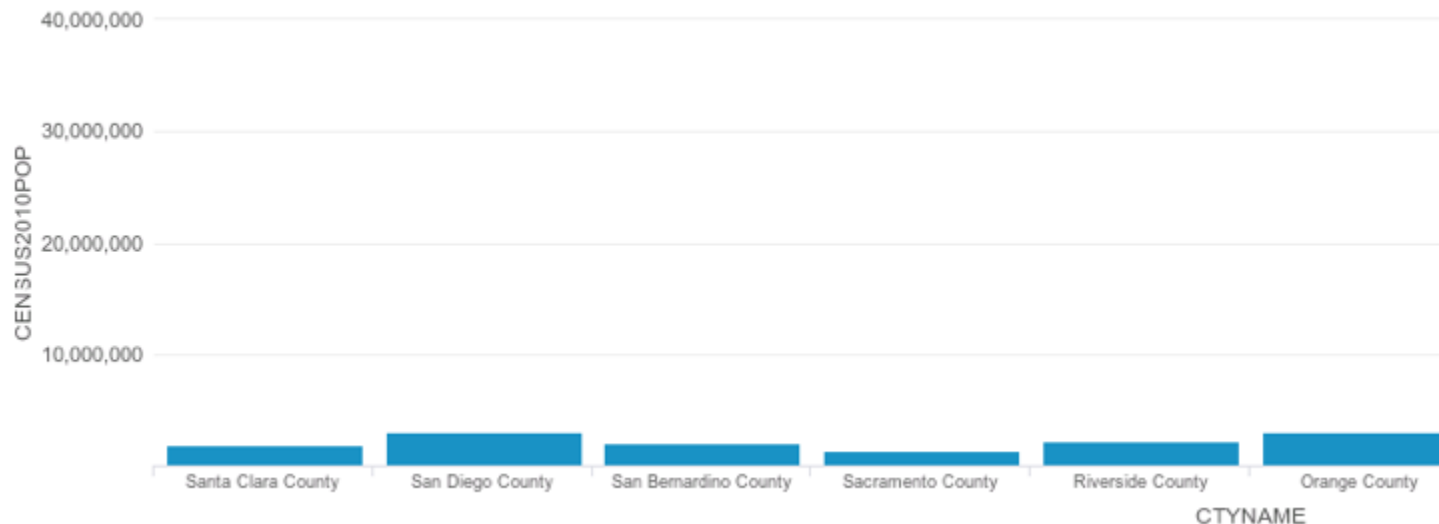
```
CENSUS2010POP > 100000 | sort CENSUS2010POP desc | table CENSUS2010POP,STNAME
```

Instead of using "desc" you can use a dash before the sorting field, e.g. "...| sort -CENSUS2010POP | table ..." for the above query.

We can view plots of search results by clicking on the *Visualization* tab. For example, if we use our last query and add the 2010 population value to the table:

```
STNAME="California" CENSUS2010POP > 1000000 | table CTYNAME, CENSUS2010POP
```

We can click on the *Visualization* tab to see a chart of the results:



Step 5. **Perform statistical calculations.** The Splunk *stats* command is used to perform statistical calculations on the data. Let's count the results where the state is California:



STNAME="California"   stats count
✓ 59 events (before 7/25/16 2:18:39.000 PM)
Events (59) Patterns Statistics
20 Per Page ✓ Format ✓ Preview ✓
count ↕
59

We can sort based on the count by adding "| sort count" to the above query. This would sort in ascending order. if we want to sort in descending order we would use "| sort -count".

Next, let's compute the total 2010 population for California:

STNAME="California"   stats sum(CENSUS2010POP)
✓ 59 events (before 7/25/16 2:26:16.000 PM) No Event Sam
Events (59) Patterns Statistics (1) Vis
20 Per Page ✓ Format ✓ Preview ✓
sum(CENSUS2010POP) ↕
74507912

Finally, let's compute the average 2010 population for California:

STNAME="California"   stats mean(CENSUS2010POP)
✓ 59 events (before 7/25/16 2:20:34.000 PM) No Event Sam
Events (59) Patterns Statistics (1) Vis
20 Per Page ✓ Format ✓ Preview ✓
mean(CENSUS2010POP) ↕
1262845.966102

We can see that the combined DataFrame contains the columns from both *adclicksDF* and *buyclicksDF*:

```
In [13]: mergeDF.head(5)
```

```
Out[13]:
```

	timestamp_x	txld_x	userSessionId_x	teamId	userId	adId	adCategory	timestamp_y	txld_y	userSessionId_y	team	buyId	price
0	2016-05-26 15:13:22	5974	5809	27	611	2	electronics	2016-05-30 13:06:54	11058	9769	27	1	2.0
1	2016-05-26 15:13:22	5974	5809	27	611	2	electronics	2016-06-03 18:36:54	17005	15910	27	4	10.0
2	2016-05-26 15:13:22	5974	5809	27	611	2	electronics	2016-06-07 12:06:54	22930	20644	27	5	20.0
3	2016-05-26 15:13:22	5974	5809	27	611	2	electronics	2016-06-11 02:06:54	29101	26524	27	4	10.0
4	2016-05-26 15:13:22	5974	5809	27	611	2	electronics	2016-06-13 02:36:54	32796	26524	27	4	10.0

## WordCount in Spark

---

By the end of this activity, you will be able to:

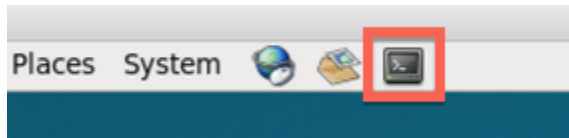
1. Read and write text files to HDFS with Spark
2. Perform WordCount with Spark Python

This activity requires programming in a Jupyter Notebook. If you have not already started the Jupyter Server, do that now. The instructions for starting the server can be found in the Reading [Instructions for Starting Jupyter Server](#) at the end of Week 1.

[NOTE: if you would like to copy and paste commands into your terminal window, you can access the completed Python Notebook for this reading at <https://github.com/words-sdsc/coursera/blob/master/big-data-3/notebooks/Spark%20WordCount.ipynb>]

**Step 1. Check if Shakespeare text is in HDFS.** In this activity, we will count the words in the complete works of Shakespeare. In Course 1: Intro to Big Data, we downloaded and copied a text file containing these works to HDFS. Let's check to see if the file is still in HDFS. If the file is not there (or you did not take Course 1), there are instructions below for copying the file into HDFS.

Open a terminal window by clicking on the square black box on the top left of the screen.



We can see the contents of HDFS by running `hadoop fs -ls`:

```
[cloudera@quickstart Downloads]$ hadoop fs -ls
Found 1 items
-rw-r--r--    1 cloudera cloudera    5458199 2016-02-12 15:14 words.txt
[cloudera@quickstart Downloads]$ █
```

In Course 1, we saved the complete works of Shakespeare as `words.txt`. If you see this file in HDFS, proceed to Step 2. If this file is not in HDFS, let's copy it. Change directory to `Downloads/big-data-3/spark-wordcount`:

Copy the file to HDFS:

```
hadoop fs -put words.txt
```

If you re-run `hadoop fs ls`, you should see `words.txt`.

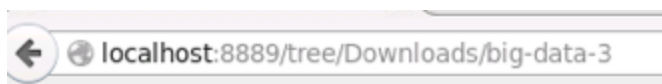
TO execute a Line of code in the Jupyter notebook, either:

- a). Click on the play next button as you would on a DVD player
- b). Hold down Shift and press Enter

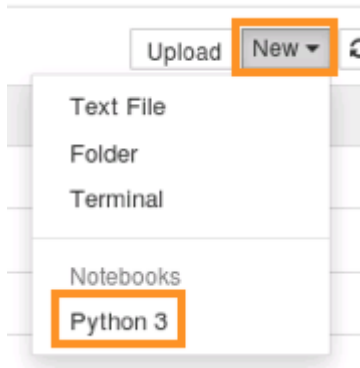
**Step 2. Open a web browser and create a new Jupyter Python Notebook.** Open a web browser by clicking on the web browser icon at the top of the toolbar:



Navigate to `localhost:8889/tree/Downloads/big-data-3`:



Create a new Python Notebook by clicking on *New*, and then click on *Python 3*:



Step 3. **Read Shakespeare text from HDFS.** We can read the text in HDFS into a new variable called *lines*:

```
In [1]: lines = sc.textFile("hdfs://user/cloudera/words.txt")
```

The `SparkContext`, `sc`, is the main entry point for accessing Spark in Python. The `textFile()` method reads the file into a Resilient Distributed Dataset (RDD) with each line in the file being an element in the RDD collection. The URL `hdfs://user/cloudera/words.txt` specifies the location of the file in HDFS.

We can verify the file was successfully loaded by calling the `count()` method, which prints the number of elements in the RDD:

```
In [2]: lines.count()
```

```
Out[2]: 124456
```

Step 4. **Split each line into words.** Next, we will split each line into a set of words. To split each line into words and store them in an RDD called *words*, run:

```
In [3]: words = lines.flatMap(lambda line : line.split(" "))
```

The `flatMap()` method iterates over every line in the RDD, and `lambda line : line.split(" ")` is executed on each line. The `lambda` notation is an anonymous function in Python, i.e., a function defined without using a name. In this case, the anonymous function takes a single argument, `line`, and calls `split(" ")` which splits the line into an array `words`.

## Exploring SparkSQL and Spark DataFrames

---

By the end of this activity, you will be able to:

1. Access Postgres database tables with SparkSQL
2. Filter rows and columns of a Spark DataFrame
3. Group and perform aggregate functions on columns in a Spark DataFrame
4. Join two SparkDataframes on a single column

Step 1. **Open Jupyter Python Notebook for SparkSQL.** First Open the Terminal and enter the command "pyspark" to setup the server. Next, open a web browser by clicking on the web browser icon at the top of the toolbar:



Navigate to `localhost:8889/tree/Downloads/big-data-3/spark-sql/`.



Open the SparkSQL Notebook by clicking on `SparkSQL.ipynb`:



Step 2. **Connect to Postgres Table.** This notebook already contains three lines of code so you do not have to enter them. Run these three lines. The first line imports the `SQLContext` module, which is needed access SQL databases in Spark:

```
In [1]: from pyspark.sql import SQLContext
```

The second line creates a new `SQLContext` from the `SparkContext` `sc`:

```
In [2]: sqlsc = SQLContext(sc)
```

The third line creates a new Spark DataFrame in the variable `df` for the Postgres table `gameclicks`:

```
In [3]: df = sqlsc.read.format("jdbc") \
        .option("url", "jdbc:postgresql://localhost/cloudera?user=cloudera") \
        .option("dbtable", "gameclicks") \
        .load()
```

The `format("jdbc")` says that the source of the DataFrame will be using a Java database connection, the `url` option is the URL connection string to access the Postgres database, and the `dbtable` option specifies the `gameclicks` table.

Step 3. **View Spark DataFrame schema and count rows.** We can call the `printSchema()` method to view the schema of the DataFrame:

```
In [4]: df.printSchema()

root
|-- timestamp: timestamp (nullable = false)
|-- clickid: integer (nullable = false)
|-- userid: integer (nullable = false)
|-- usersessionid: integer (nullable = false)
|-- ishit: integer (nullable = false)
|-- teamid: integer (nullable = false)
|-- teamlevel: integer (nullable = false)
```

The description lists the name and data type of each column.

We can also call the `count()` method to count the number of rows in the DataFrame:

```
In [5]: df.count()
```

```
Out[5]: 755806
```

Step 4. **View contents of DataFrame.** We can call the `show()` method to view the contents of the DataFrame. The argument specifies how many rows to display:

```
In [6]: df.show(5)
```

```
+-----+-----+-----+-----+-----+-----+-----+
|          timestamp|clickid|userid|usersessionid|ishit|teamid|teamlevel|
+-----+-----+-----+-----+-----+-----+-----+
|2016-05-26 15:06:...|    105|   1038|         5916|    0|    25|         1|
|2016-05-26 15:07:...|    154|   1099|         5898|    0|    44|         1|
|2016-05-26 15:07:...|    229|    899|         5757|    0|    71|         1|
|2016-05-26 15:07:...|    322|   2197|         5854|    0|    99|         1|
|2016-05-26 15:07:...|     22|   1362|         5739|    0|    13|         1|
+-----+-----+-----+-----+-----+-----+-----+
only showing top 5 rows
```

Step 5. **Filter columns in DataFrame.** We can filter for one or more columns by calling the `select()` method:

```
In [7]: df.select("userid", "teamlevel").show(5)
```

```
+-----+-----+
|userid|teamlevel|
+-----+-----+
|  1038|         1|
|  1099|         1|
|   899|         1|
|  2197|         1|
|  1362|         1|
+-----+-----+
only showing top 5 rows
```

Step 6. **Filter rows based on criteria**. We can also filter for rows that match a specific criteria using `filter()`:

```
In [8]: df.filter(df["teamlevel"] > 1).select("userid", "teamlevel").show(5)
```

```
+-----+-----+
|userid|teamlevel|
+-----+-----+
|  1513|         2|
|   868|         2|
|  1453|         2|
|  1282|         2|
|  1473|         2|
+-----+-----+
only showing top 5 rows
```

The arguments to `filter()` are a Column, in this case specified as `df["teamlevel"]`, and the condition, which is greater than 1. The remainder of the command selects only the `userid` and `teamlevel` columns and shows the first five rows.

Step 7. **Group by a column and count**. The `groupBy()` method groups the values of column(s). The `ishit` column only has values 0 and 1. We can calculate how many times each occurs by grouping the `ishit` column and counting the result:

```
In [9]: df.groupBy("ishit").count().show()
```

```
+-----+-----+
|ishit| count|
+-----+-----+
|     0|672423|
|     1| 83383|
+-----+-----+
```

Step 8. **Calculate average and sum**. Aggregate operations can be performed on columns of DataFrames. First, let's import the Python libraries for the aggregate operations. Next, we can calculate the average and total values by calling the `mean()` and `sum()` methods, respectively:

```
In [10]: from pyspark.sql.functions import *
df.select(mean('ishit'), sum('ishit')).show()
```

```
+-----+-----+
|      avg(ishit)|sum(ishit)|
+-----+-----+
|0.1103232840173272|      83383|
+-----+-----+
```

Step 9. **Join two DataFrames.** We can merge or join two Dataframes on a single column. First, let's create a DataFrame for the *adclicks* table in the Postgres database by copying the third cell in this notebook and changing *gameclicks* to *adclicks* and storing the result in a new variable *df2*:

```
In [11]: df2 = sqlsc.read.format("jdbc") \
    .option("url", "jdbc:postgresql://localhost/cloudera?user=cloudera") \
    .option("dbtable", "adclicks") \
    .load()
```

Let's view the columns in *df2* by calling *printSchema()*:

```
In [12]: df2.printSchema()

root
 |-- timestamp: timestamp (nullable = false)
 |-- txid: integer (nullable = false)
 |-- usersessionid: integer (nullable = false)
 |-- teamid: integer (nullable = false)
 |-- userid: integer (nullable = false)
 |-- adid: integer (nullable = false)
 |-- adcategory: string (nullable = false)
```

We can see that the *adclicks* *df2* DataFrame also has a column called *userid*. Next, we will combine the *gameclicks* and *adclicks* DataFrames by calling the *join()* method and saving the resulting DataFrame in a variable called *merge*:

```
In [13]: merge = df.join(df2, 'userid')
```

We are calling the *join()* method on the *gameclicks* DataFrame; the first argument is the DataFrame to join with, i.e., the *adclicks* DataFrame, and the second argument is the column name in both DataFrames to join on.

Let's view the schema of *merge*:



```
In [14]: merge.printSchema()
```

```
root
 |-- userid: integer (nullable = false)
 |-- timestamp: timestamp (nullable = false)
 |-- clickid: integer (nullable = false)
 |-- usersessionid: integer (nullable = false)
 |-- ishit: integer (nullable = false)
 |-- teamid: integer (nullable = false)
 |-- teamlevel: integer (nullable = false)
 |-- timestamp: timestamp (nullable = false)
 |-- txid: integer (nullable = false)
 |-- usersessionid: integer (nullable = false)
 |-- teamid: integer (nullable = false)
 |-- adid: integer (nullable = false)
 |-- adcategory: string (nullable = false)
```

We can see that the merged DataFrame has all the columns of both *gameclicks* and *adclicks*.

Finally, let's look at the contents of *merge*:

```
In [15]: merge.show(5)
```

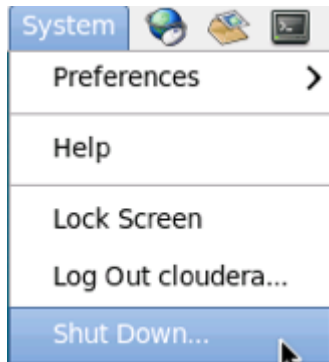
```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
|userid|      timestamp|clickid|usersessionid|ishit|teamid|teamlevel|      timestamp| txid|usersessionid|teamid|
|adid|adcategory|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| 231|2016-06-08 00:45:...| 376796|      23626|  0|  142|      4|2016-06-08 01:40:...|23669|      23626|  142|
| 27|  games|
| 231|2016-06-08 00:45:...| 376796|      23626|  0|  142|      4|2016-06-08 09:24:...|24122|      23626|  142|
|  4|  games|
| 231|2016-06-08 00:45:...| 376796|      23626|  0|  142|      4|2016-06-08 17:21:...|24659|      23626|  142|
| 22| computers|
| 231|2016-06-08 00:45:...| 376796|      23626|  0|  142|      4|2016-06-08 23:34:...|25076|      23626|  142|
| 21|  movies|
| 231|2016-06-08 00:45:...| 376796|      23626|  0|  142|      4|2016-06-09 16:32:...|26220|      23626|  142|
| 16| clothing|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
only showing top 5 rows
```

## Instructions for Configuring VirtualBox for Spark Streaming

---

Spark Streaming requires more than one executor. This Reading describes how to configure VirtualBox so that the Cloudera VM has more than one virtual processor.

Step 1. **Stop the Cloudera VM.** Before we can change the settings for the Cloudera VM, the VM needs to be powered off. If the VM is running, click on *System* in the top toolbar, and then click on *Shutdown*:



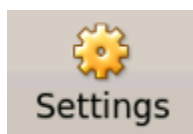
Next, click on *Shut down*:



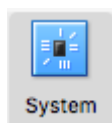
Step 2. **Change number of processors.** Once the Cloudera VM is powered off, select the Cloudera VM in the list of virtual machines in the VirtualBox Manager:



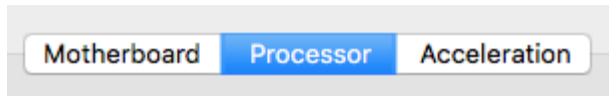
Next, click on *Settings*:



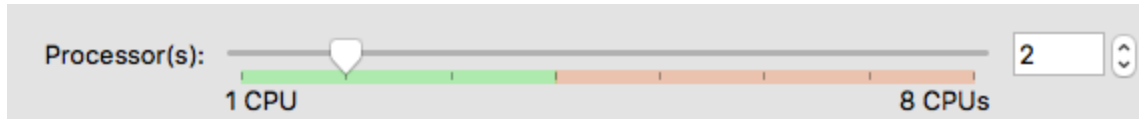
Next, click on *System*:



Next, click on *Processor*:



The default number of processors is one. Change this to two or more:



Finally, click on *OK* and start the Cloudera VM.

## Analyzing Sensor Data with Spark Streaming

---

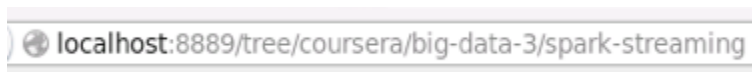
By the end of this activity, you will be able to:

1. Read streaming data into Spark
2. Create and apply computations over a sliding window of data

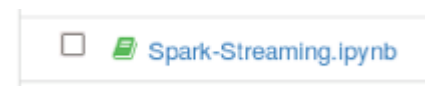
Step 1. **Open Jupyter Python Notebook for Spark Streaming.** Open a web browser by clicking on the web browser icon at the top of the toolbar:



Navigate to *localhost:8889/tree/Downloads/big-data-3/spark-streaming*:



Open the Spark Streaming Notebook by clicking on *Spark-Streaming.ipynb*:



Step 2. **Look at sensor format and measurement types.** The first cell in the notebook gives an example of the streaming measurements coming from the weather station:

```
In [ ]: # Example weather station data
#
# 1419408015 0R1,Dn=059D,Dm=066D,Dx=080D,Sn=8.5M,Sm=9.5M,Sx=10.3M
# 1419408016 0R1,Dn=059D,Dm=065D,Dx=078D,Sn=8.5M,Sm=9.5M,Sx=10.3M
# 1419408016 0R2,Ta=13.9C,Ua=28.5P,Pa=889.9H
# 1419408017 0R1,Dn=059D,Dm=064D,Dx=075D,Sn=8.7M,Sm=9.6M,Sx=10.3M
# 1419408018 0R1,Dn=059D,Dm=064D,Dx=075D,Sn=8.9M,Sm=9.6M,Sx=10.3M
# 1419408019 0R1,Dn=059D,Dm=065D,Dx=075D,Sn=8.8M,Sm=9.5M,Sx=10.3M
```

Each line contains a timestamp and a set of measurements. Each measurement has an abbreviation, and for this exercise, we are interested in the average wind direction, which is *Dm*. The next cell lists the abbreviations used for each type of measurement:

```
In [ ]: # Key for measurements:
#
# Sn      Wind speed minimum m/s, km/h, mph, knots #,M, K, S, N
# Sm      Wind speed average m/s, km/h, mph, knots #,M, K, S, N
# Sx      Wind speed maximum m/s, km/h, mph, knots #,M, K, S, N
# Dn      Wind direction minimum deg #, D
# Dm      Wind direction average deg #, D
# Dx      Wind direction maximum deg #, D
# Pa      Air pressure hPa, Pa, bar, mmHg, inHg #, H, P, B, M, I
# Ta      Air temperature °C, °F #, C, F
# Tp      Internal temperature °C, °F #, C, F
# Ua      Relative humidity %RH #, P
# Rc      Rain accumulation mm, in #, M, I
# Rd      Rain duration s #, S
# Ri      Rain intensity mm/h, in/h #, M, I
# Rp      Rain peak intensity mm/h, in/h #, M, I
# Hc      Hail accumulation hits/cm2, hits/in2, hits #, M, I, H
# Hd      Hail duration s #, S
# Hi      Hail intensity hits/cm2h, hits/in2h, hits/ h #, M, I, H
# Hp      Hail peak intensity hits/cm2h, hits/in2h, hits/ h #, M, I, H
# Th      Heating temperature °C, °F #, C, F
# Vh      Heating voltage V #, N, V, W, F2
# Vs      Supply voltage V V
# Vr      3.5 V ref. voltage V V
```

The third cell defines a function that parses each line and returns the average wind direction (*Dm*). Run this cell:

```
In [1]: # Parse a line of weather station data, returning the average wind direction measurement
#
import re
def parse(line):
    match = re.search("Dm=(\d+)", line)
    if match:
        val = match.group(1)
        return [int(val)]
    return []
```

Step 3. **Import and create streaming context.** Next, we will import and create a new instance of Spark's `StreamingContext`:

```
In [2]: from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 1)
```

Similar to the `SparkContext`, the `StreamingContext` provides an interface to Spark's streaming capabilities. The argument `sc` is the `SparkContext`, and `1` specifies a batch interval of one second.

Step 4. **Create DStream of weather data.** Let's open a connection to the streaming weather data:

```
In [3]: lines = ssc.socketTextStream("rtd.hpwren.ucsd.edu", 12028)
```

Instead of 12028, you may find that port 12020 works instead. This creates a new variable `lines` to be a Spark `DStream` that streams the lines of output from the weather station.

Step 5. **Read measurement.** Next, let's read the average wind speed from each line and store it in a new `DStream` `vals`:

```
In [4]: vals = lines.flatMap(parse)
```

This line uses `flatMap()` to iterate over the `lines` `DStream`, and calls the `parse()` function we defined above to get the average wind speed.

Step 6. **Create sliding window of data.** We can create a sliding window over the measurements by calling the `window()` method:

```
In [5]: window = vals.window(10, 5)
```

This creates a new `DStream` called `window` that combines the ten seconds worth of data and moves by five seconds.

Step 7. **Define and call analysis function.** We would like to find the minimum and maximum values in our window. Let's define a function that prints these values for an `RDD`:

```
In [6]: def stats(rdd):
        print(rdd.collect())
        if rdd.count() > 0:
            print("max = {}, min = {}".format(rdd.max(), rdd.min()))
```

This function first prints the entire contents of the RDD by calling the `collect()` method. This is done to demonstrate the sliding window and would not be practical if the RDD was containing a large amount of data. Next, we check if the size of the RDD is greater than zero before printing the maximum and minimum values.

Next, we call the `stats()` function for each RDD in our sliding window:

```
In [7]: window.foreachRDD(lambda rdd: stats(rdd))
```

This line calls the `stats()` function defined above for each RDD in the DStream window.

Step 8. **Start the stream processing.** We call `start()` on the `StreamingContext` to begin the processing:

```
In [8]: ssc.start()
```

Window 1 → [346, 333, 332]  
           max = 346, min = 332

Window 2 → [346, 333, 332, 313, 314, 316, 317, 317]  
           max = 346, min = 313

Window 3 → [313, 314, 316, 317, 317, 316, 316, 316, 316, 317]  
           max = 317, min = 313

Window 4 → [316, 316, 316, 316, 317, 316, 316, 316, 316, 317]  
           max = 317, min = 316

Window 5 → [316, 316, 316, 316, 317, 320, 321, 323, 325, 326]  
           max = 326, min = 316

Window 6 → [320, 321, 323, 325, 326, 309, 309, 308, 308, 307]  
           max = 326, min = 307

Window 7 → [309, 309, 308, 308, 307, 307, 307, 307, 307, 307]  
           max = 309, min = 307

The sliding window contains ten seconds worth of data and slides every five seconds. In the beginning, the number of values in the windows are increasing as the data accumulates, and after Window 3, the size stays (approximately) the same. Since the window slides half as often as the size of the window, the second half of a window becomes the first half of the next window. For example, the second half of Window 5 is 310, 321, 323, 325, 326, which becomes the first half of Window 6.

When we are done, call `stop()` on the `StreamingContext`:

```
In [9]: ssc.stop()
```