 databricks cs105_lab1a_spark_tutorial



(<http://creativecommons.org/licenses/by-nc-nd/4.0/>)

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. (<http://creativecommons.org/licenses/by-nc-nd/4.0/>)



Spark Tutorial: Learning Apache Spark

This tutorial will teach you how to use Apache Spark (<http://spark.apache.org/>), a framework for large-scale data processing, within a notebook. Many traditional frameworks were designed to be run on a single computer. However, many datasets today are too large to be stored on a single computer, and even when a dataset can be stored on one computer (such as the datasets in this tutorial), the dataset can often be processed much more quickly using multiple computers.

Spark has efficient implementations of a number of transformations and actions that can be composed together to perform data processing and analysis. Spark excels at distributing these operations across a cluster while abstracting away many of the underlying implementation details. Spark has been designed with a focus on scalability and efficiency. With Spark you can begin developing your solution on your laptop, using a small dataset, and then use that same code to process terabytes or even petabytes across a distributed cluster.

During this tutorial we will cover:

- *Part 1:* Basic notebook usage and Python (<https://docs.python.org/2/>) integration
- *Part 2:* An introduction to using Apache Spark (<https://spark.apache.org/>) with the PySpark SQL API (<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark-sql-module>) running in a notebook
- *Part 3:* Using DataFrames and chaining together transformations and actions
- *Part 4:* Python Lambda functions and User Defined Functions
- *Part 5:* Additional DataFrame actions
- *Part 6:* Additional DataFrame transformations
- *Part 7:* Caching DataFrames and storage options
- *Part 8:* Debugging Spark applications and lazy evaluation

The following transformations will be covered:

- `select()` , `filter()` , `distinct()` , `dropDuplicates()` , `orderBy()` , `groupBy()`

The following actions will be covered:

- `first()` , `take()` , `count()` , `collect()` , `show()`

Also covered:

- `cache()` , `unpersist()`

Note that, for reference, you can look up the details of these methods in the Spark's PySpark SQL API (<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark-sql-module>)

Part 1: Basic notebook usage and Python (<https://docs.python.org/2/>) integration

(1a) Notebook usage

A notebook is comprised of a linear sequence of cells. These cells can contain either markdown or code, but we won't mix both in one cell. When a markdown cell is executed it renders formatted text, images, and links just like HTML in a normal webpage. The text you are reading right now is part of a markdown cell. Python code cells allow you to execute arbitrary Python commands just like in any Python shell. Place your cursor inside the cell below, and press "Shift" + "Enter" to execute the code and advance to the next cell. You can also press "Ctrl" + "Enter" to execute the code and remain in the cell. These commands work the same in both markdown and code cells.

```
# This is a Python cell. You can run normal Python code here...
print 'The sum of 1 and 1 is {}'.format(1+1)
```

The sum of 1 and 1 is 2

```
# Here is another Python cell, this time with a variable (x) declaration and an if
statement:
x = 42
if x > 40:
    print 'The sum of 1 and 2 is {}'.format(1+2)
```

The sum of 1 and 2 is 3

(1b) Notebook state

As you work through a notebook it is important that you run all of the code cells. The notebook is stateful, which means that variables and their values are retained until the notebook is detached (in Databricks) or the kernel is restarted (in Jupyter notebooks). If you do not run all of the code cells as you proceed through the notebook, your variables will not be properly initialized and later code might fail. You will also need to rerun any cells that you have modified in order for the changes to be available to other cells.

```
# This cell relies on x being defined already.
# If we didn't run the cells from part (1a) this code would fail.
print x * 2
```

84

(1c) Library imports

We can import standard Python libraries (modules (<https://docs.python.org/2/tutorial/modules.html>)) the usual way. An `import` statement will import the specified module. In this tutorial and future labs, we will provide any imports that are necessary.

```
# Import the regular expression library
import re
m = re.search('(?<=abc)def', 'abcdef')
m.group(0)
```

Out[8]: 'def'

```
# Import the datetime library
import datetime
print 'This was last run on: {}'.format(datetime.datetime.now())
```

This was last run on: 2018-10-10 17:15:12.209644

Part 2: An introduction to using Apache Spark (<https://spark.apache.org/>) with the PySpark SQL API (http://spark.apache.org/docs/latest/api/python/pyspark_sql-module) running in a notebook

Spark Context

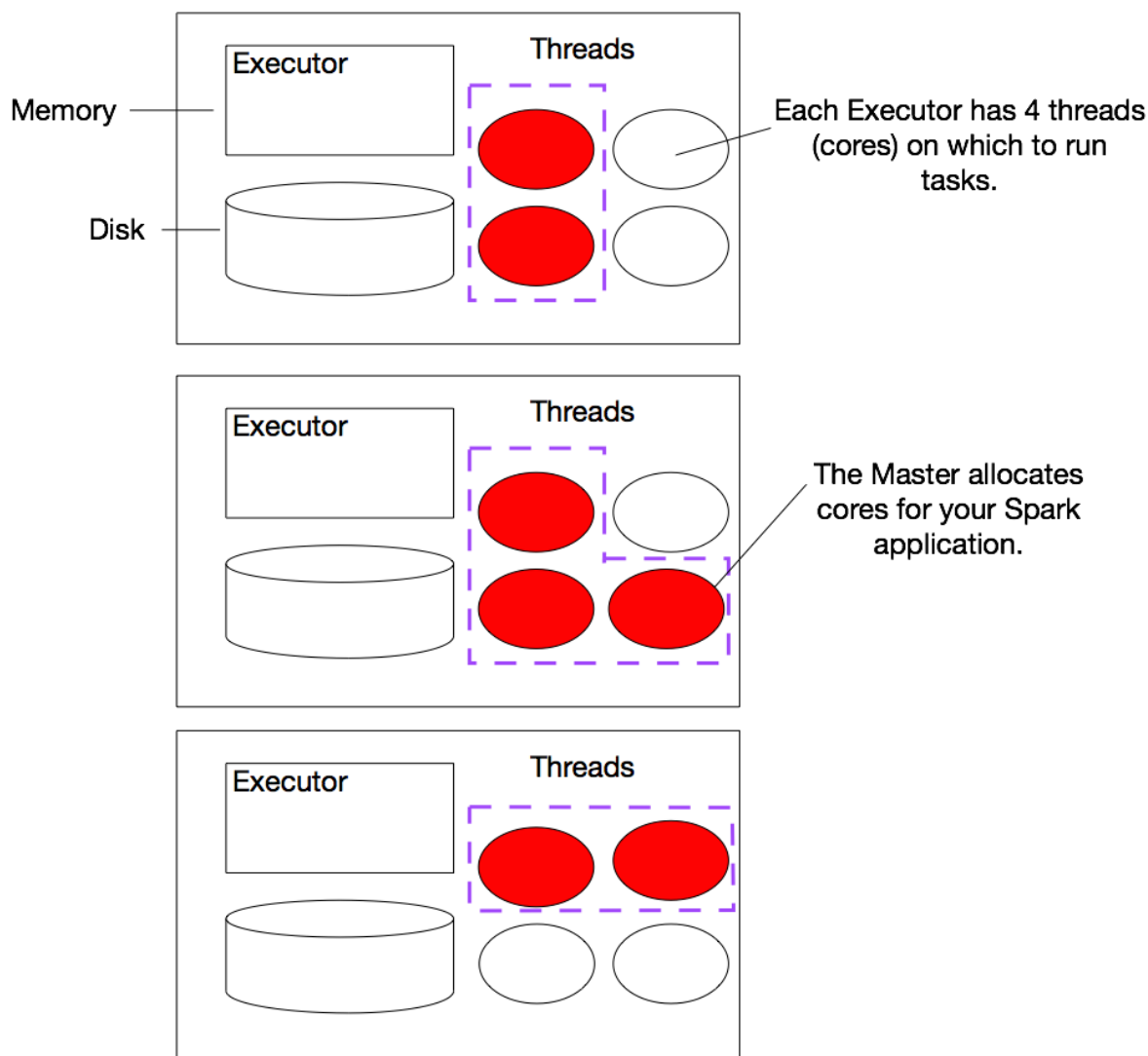
In Spark, communication occurs between a driver and executors. The driver has Spark jobs that it needs to run and these jobs are split into tasks that are submitted to the executors for completion. The results from these tasks are delivered back to the driver.

In part 1, we saw that normal Python code can be executed via cells. **When using Databricks this code gets executed in the Spark driver's Java Virtual Machine (JVM) and not in an executor's JVM, and when using an Jupyter notebook it is executed within the kernel associated with the notebook. Since no Spark functionality is actually being used, no tasks are launched on the executors.**

In order to use Spark and its DataFrame API we will need to use a `SQLContext`. When running Spark, you start a new Spark application by creating a `SparkContext` (<http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext>). You can then create a `SQLContext` (<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.SQLContext>) from the `SparkContext`. When the `SparkContext` is created, it asks the master for some cores to use to do work. The master sets these cores aside just for you; they won't be used for other applications. **When using Databricks, both a `SparkContext` and a `SQLContext` are created for you automatically. `sc` is your `SparkContext`, and `sqlContext` is your `SQLContext`.**

(2a) Example Cluster

The diagram shows an example cluster, where the slots allocated for an application are outlined in purple. (Note: We're using the term *slots* here to indicate threads available to perform parallel work for Spark. Spark documentation often refers to these threads as *cores*, which is a confusing term, as the number of slots available on a particular machine does not necessarily have any relationship to the number of physical CPU cores on that machine.)



You can view the details of your Spark application in the Spark web UI. The web UI is accessible in Databricks by going to "Clusters" and then clicking on the "Spark UI" link for your cluster. In the web UI, under the "Jobs" tab, you can see a list of jobs that have been scheduled or run. It's likely there isn't anything interesting here yet because we haven't run any jobs, but we'll return to this page later.

At a high level, every Spark application consists of a driver program that launches various parallel operations on executor Java Virtual Machines (JVMs) running either in a cluster or locally on the same machine. In Databricks, "Databricks Shell" is the driver program. When running locally, `pyspark` is the driver program. In all cases, this driver program contains the main loop for the program and creates distributed datasets on the cluster, then applies operations (transformations & actions) to those datasets. Driver programs access Spark through a

SparkContext object, which represents a connection to a computing cluster. **A Spark SQL context object (`sqlContext`) is the main entry point for Spark DataFrame and SQL functionality.** A `SQLContext` can be used to create DataFrames, which allows you to direct the operations on your data.

Try printing out `sqlContext` to see its type.

```
# Display the type of the Spark sqlContext
type(sqlContext)
```

```
Out[10]: pyspark.sql.context.HiveContext
```

Note that the type is `HiveContext`. This means we're working with **a version of Spark that has Hive support**. Compiling Spark with Hive support is a good idea, even if you don't have a Hive metastore. As the Spark Programming Guide (<http://spark.apache.org/docs/latest/sql-programming-guide.html#starting-point-sqlcontext>) states, a `HiveContext` "provides a superset of the functionality provided by the basic `SQLContext`. Additional features include the ability to write queries using the more complete HiveQL parser, access to Hive UDFs [user-defined functions], and the ability to read data from Hive tables. To use a `HiveContext`, you do not need to have an existing Hive setup, and all of the data sources available to a `SQLContext` are still available."

(2b) SparkContext attributes

You can use Python's `dir()` (<https://docs.python.org/2/library/functions.html?highlight=dir#dir>) function to get a list of all the attributes (including methods) accessible through the `sqlContext` object.

```
# List sqlContext's attributes
dir(sqlContext)
```

```
Out[11]:
['__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__']
```

```
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'_createForTesting',
'_inferSchema'.
```

(2c) Getting help

Alternatively, you can use Python's `help()` (<https://docs.python.org/2/library/functions.html?highlight=help#help>) function to get an easier to read list of all the attributes, including examples, that the `sqlContext` object has.

```
# Use help to obtain more detailed information
help(sqlContext)
```

Help on HiveContext in module pyspark.sql.context object:

```
class HiveContext(SQLContext)
|   A variant of Spark SQL that integrates with data stored in Hive.
|
|   Configuration for Hive is read from ``hive-site.xml`` on the classpath.
|   It supports running both SQL and HiveQL commands.
|
|   :param sparkContext: The SparkContext to wrap.
|   :param jhiveContext: An optional JVM Scala HiveContext. If set, we do not instantiat
e a new
|       :class:`HiveContext` in the JVM, instead we make all calls to this object.
|
|   .. note:: Deprecated in 2.0.0. Use SparkSession.builder.enableHiveSupport().getOrCre
ate().
|
|   Method resolution order:
|       HiveContext
|       SQLContext
|       __builtin__.object
```

Outside of `pyspark` or a notebook, `SQLContext` is created from the lower-level `SparkContext`, which is usually used to create Resilient Distributed Datasets (RDDs). **An RDD is the way Spark actually represents data internally;** DataFrames are actually implemented in terms of RDDs.

While you can interact directly with RDDs, DataFrames are preferred. They're generally faster, and they perform the same no matter what language (Python, R, Scala or Java) you use with Spark.

In this course, we'll be using DataFrames, so we won't be interacting directly with the Spark Context object very much. However, it's worth knowing that **inside pyspark or a notebook, you already have an existing sparkContext in the sc variable**. One simple thing we can do with `sc` is check the version of Spark we're using:

```
# After reading the help we've decided we want to use sc.version to see what version of
Spark we are running.
# Note that the lower case u in front of a string means it's an unicode string. It's only
the encoding, and therefore is no harm at all.
sc.version
```

```
Out[14]: u'2.3.1'
```

```
# Help can be used on any Python object
help(map)
```

```
Help on built-in function map in module __builtin__:
```

```
map(...)
map(function, sequence[, sequence, ...]) -> list
```

```
Return a list of the results of applying the function to the items of
the argument sequence(s). If more than one sequence is given, the
function is called with an argument list consisting of the corresponding
item of each sequence, substituting None for missing values when not all
sequences have the same length. If the function is None, return a list of
the items of the sequence (or a list of tuples if more than one sequence).
```

Part 3: Using DataFrames and chaining together transformations and actions

Working with your first DataFrames

In Spark, we first create a base DataFrame (<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame>). We can then apply one or more transformations to that base DataFrame. **A DataFrame is immutable, so once it is created, it cannot be changed.** As a result, each transformation creates a new DataFrame. Finally, we can apply one or more actions to the DataFrames.

Note that Spark uses lazy evaluation, so transformations are not actually executed until an action occurs.

We will perform several exercises to obtain a better understanding of DataFrames:

- Create a Python collection of 10,000 integers
- Create a Spark DataFrame from that collection
- Subtract one from each value using `map`
- Perform action `collect` to view results
- Perform action `count` to view counts
- Apply transformation `filter` and view results with `collect`
- Learn about lambda functions
- Explore how lazy evaluation works and the debugging challenges that it introduces

A DataFrame consists of a series of `Row` objects; each `Row` object has a set of named columns. You can think of a DataFrame as modeling a table, though the data source being processed does not have to be a table.

More formally, a DataFrame must have a *schema*, which means it must consist of columns, each of which has a *name* and a *type*. Some data sources have schemas built into them. Examples include RDBMS databases, Parquet files, and NoSQL databases like Cassandra. Other data sources don't have computer-readable schemas, but you can often apply a schema programmatically.

(3a) Create a Python collection of 10,000 people

We will use a third-party Python testing library called fake-factory (<https://pypi.python.org/pypi/fake-factory/0.5.3>) to create a collection of fake person records.

```
from faker import Faker
# solution to the error: https://forums.databricks.com/questions/10790/i-am-tring-to-use-faker-python-lib-but-i-am-keep-g.html
fake = Faker()
fake.seed(4321)
```

We're going to use this factory to create a collection of randomly generated people records. In the next section, we'll turn that collection into a DataFrame. We'll use a Python tuple to help us define the Spark DataFrame schema. There are other ways to define schemas, though; see the Spark Programming Guide's discussion of schema inference

(<http://spark.apache.org/docs/latest/sql-programming-guide.html#inferring-the-schema-using-reflection>) for more information. (For instance, we could also use a Python `namedtuple` or a Spark `Row` object.)

```
# Each entry consists of last_name, first_name, ssn, job, and age (at least 1)
from pyspark.sql import Row
def fake_entry():
    name = fake.name().split()
    return (name[1], name[0], fake.ssn(), fake.job(), abs(2016 - fake.date_time().year) +
1)

# Create a helper function to call a function repeatedly
def repeat(times, func, *args, **kwargs):
    for _ in xrange(times):
        yield func(*args, **kwargs)

data = list(repeat(10000, fake_entry))
```

`data` is just a normal Python list, containing Python tuples objects. Let's look at the first item in the list:

```
data[0]
```

```
Out[32]: (u'Brown', u'Jason', u'160-37-9051', 'Aid worker', 38)
```

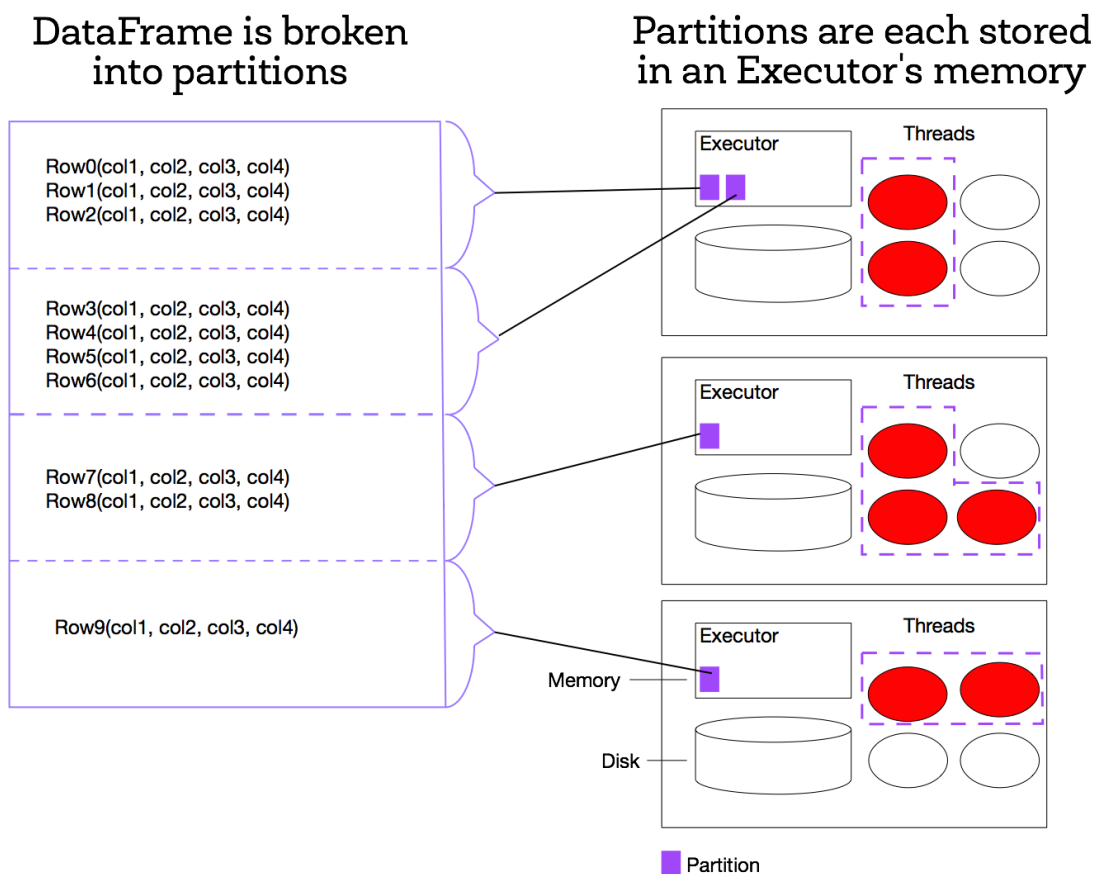
We can check the size of the list using the Python `len()` function.

```
len(data)
```

```
Out[33]: 10000
```

(3b) Distributed data and using a collection to create a DataFrame

In Spark, datasets are represented as a list of entries, where the list is broken up into many different partitions that are each stored on a different machine. Each partition holds a unique subset of the entries in the list. Spark calls datasets that it stores "Resilient Distributed Datasets" (RDDs). Even **DataFrames are ultimately represented as RDDs**, with additional meta-data.



One of the defining features of Spark, compared to other data analytics frameworks (e.g., Hadoop), is that it stores data in memory rather than on disk. This allows Spark applications to run much more quickly, because they are not slowed down by needing to read data from disk. The figure to the right illustrates how Spark breaks a list of data entries into partitions that are each stored in memory on a worker.

To create the `DataFrame`, we'll use `sqlContext.createDataFrame()`, and we'll pass our array of data in as an argument to that function. Spark will create a new set of input data based on data that is passed in. **A `DataFrame` requires a *schema*, which is a list of columns, where each column has a name and a type.** Our list of data has elements with types (mostly strings, but one integer). We'll supply the rest of the schema and the column names as the second argument to `createDataFrame()`.

Let's view the help for `createDataFrame()`.

```
help(sqlContext.createDataFrame)
```

Help on method createDataFrame in module pyspark.sql.context:

createDataFrame(self, data, schema=None, samplingRatio=None, verifySchema=True) method of pyspark.sql.context.HiveContext instance

Creates a :class:`DataFrame` from an :class:`RDD`, a list or a :class:`pandas.DataFrame`.

When ``schema`` is a list of column names, the type of each column will be inferred from ``data``.

When ``schema`` is ``None``, it will try to infer the schema (column names and types) from ``data``, which should be an RDD of :class:`Row`, or :class:`namedtuple`, or :class:`dict`.

When ``schema`` is :class:`pyspark.sql.types.DataType` or a datatype string it must match the real data, or an exception will be thrown at runtime. If the given schema is not :class:`pyspark.sql.types.StructType`, it will be wrapped into a :class:`pyspark.sql.types.StructType` as its only field, and the field name will be "value",

```
dataDF = sqlContext.createDataFrame(data, ('last_name', 'first_name', 'ssn',
'occupation', 'age'))
```

Let's see what type `sqlContext.createDataFrame()` returned.

```
print 'type of dataDF: {0}'.format(type(dataDF))
```

```
type of dataDF: <class 'pyspark.sql.dataframe.DataFrame'>
```

Let's take a look at the DataFrame's schema and some of its rows.

```
dataDF.printSchema()
```

```
root
|-- last_name: string (nullable = true)
|-- first_name: string (nullable = true)
|-- ssn: string (nullable = true)
|-- occupation: string (nullable = true)
|-- age: long (nullable = true)
```

We can register the newly created DataFrame as a named table, using the `registerDataFrameAsTable()` method.

```
sqlContext.registerDataFrameAsTable(dataDF, 'dataframe')
```

What methods can we call on this DataFrame?

```
help(dataDF)
```

```
Help on DataFrame in module pyspark.sql.dataframe object:

class DataFrame(__builtin__.object)
|   A distributed collection of data grouped into named columns.
|
|   A :class:`DataFrame` is equivalent to a relational table in Spark SQL,
|   and can be created using various functions in :class:`SparkSession`:
|
|       people = spark.read.parquet("...")
|
|   Once created, it can be manipulated using the various domain-specific-language
|   (DSL) functions defined in: :class:`DataFrame`, :class:`Column`.
|
|   To select a column from the data frame, use the apply method::
|
|       ageCol = people.age
|
|   A more concrete example::
|
|       # To create DataFrame using SparkSession
|       people = spark.read.parquet("...")
```

How many partitions will the DataFrame be split into?

```
dataDF.rdd.getNumPartitions()
```

```
Out[40]: 8
```

A note about DataFrames and queries

When you use DataFrames or Spark SQL, you are building up a *query plan*. **Each transformation you apply to a DataFrame adds some information to the query plan. When you finally call an action, which triggers execution of your Spark job, several things happen:**

1. Spark's Catalyst optimizer analyzes the query plan (called an *unoptimized logical query plan*) and attempts to optimize it. Optimizations include (but aren't limited to) rearranging and combining `filter()` operations for efficiency, converting `Decimal` operations to more efficient long integer operations, and pushing some operations down into the data source

(e.g., a `filter()` operation might be translated to a SQL `WHERE` clause, if the data source is a traditional SQL RDBMS). The result of this optimization phase is an *optimized logical plan*.

2. Once Catalyst has an optimized logical plan, it then constructs multiple *physical* plans from it. Specifically, it implements the query in terms of lower level Spark RDD operations.
3. Catalyst chooses which physical plan to use via *cost optimization*. That is, it determines which physical plan is the most efficient (or least expensive), and uses that one.
4. Finally, once the physical RDD execution plan is established, Spark actually executes the job.

You can examine the query plan using the `explain()` function on a `DataFrame`. By default, `explain()` only shows you the final physical plan; however, if you pass it an argument of `True`, it will show you all phases.

(If you want to take a deeper dive into how Catalyst optimizes `DataFrame` queries, this blog post, while a little old, is an excellent overview: [Deep Dive into Spark SQL's Catalyst Optimizer](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html) (<https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>).

Let's add a couple transformations to our `DataFrame` and look at the query plan on the resulting transformed `DataFrame`. Don't be too concerned if it looks like gibberish. As you gain more experience with Apache Spark, you'll begin to be able to use `explain()` to help you understand more about your `DataFrame` operations.

```
newDF = dataDF.distinct().select('*')
newDF.explain(True)
```

```
== Parsed Logical Plan ==
'Project [*]
+- AnalysisBarrier
    +- Deduplicate [first_name#58, age#61L, last_name#57, occupation#60, ssn#59]
      +- LogicalRDD [last_name#57, first_name#58, ssn#59, occupation#60, age#61L], false
lse

== Analyzed Logical Plan ==
last_name: string, first_name: string, ssn: string, occupation: string, age: bigint
Project [last_name#57, first_name#58, ssn#59, occupation#60, age#61L]
+- Deduplicate [first_name#58, age#61L, last_name#57, occupation#60, ssn#59]
  +- LogicalRDD [last_name#57, first_name#58, ssn#59, occupation#60, age#61L], false

== Optimized Logical Plan ==
Aggregate [first_name#58, age#61L, last_name#57, occupation#60, ssn#59], [last_name#57,
first_name#58, ssn#59, occupation#60, age#61L]
+- LogicalRDD [last_name#57, first_name#58, ssn#59, occupation#60, age#61L], false

== Physical Plan ==
*(2) HashAggregate(keys=[first_name#58, age#61L, last_name#57, occupation#60, ssn#59], f
unctions=[], output=[last_name#57, first_name#58, ssn#59, occupation#60, age#61L])
```

(3c): Subtract one from each value using *select*

So far, we've created a distributed DataFrame that is split into many partitions, where each partition is stored on a single machine in our cluster. Let's look at what happens when we do a basic operation on the dataset. Many useful data analysis operations can be specified as "do something to each item in the dataset". These data-parallel operations are convenient because each item in the dataset can be processed individually: the operation on one entry doesn't effect the operations on any of the other entries. Therefore, Spark can parallelize the operation.

One of the most common DataFrame operations is `select()`, and it works more or less like a SQL `SELECT` statement: You can select specific columns from the DataFrame, and you can even use `select()` to create *new* columns with values that are derived from existing column values. We can use `select()` to create a new column that decrements the value of the existing `age` column.

`select()` is a *transformation*. It returns a new DataFrame that captures both the previous DataFrame and the operation to add to the query (`select`, in this case). But it does *not* actually execute anything on the cluster. When transforming DataFrames, we are building up a *query plan*. That query plan will be optimized, implemented (in terms of RDDs), and executed by Spark *only* when we call an action.

```
# Transform dataDF through a select transformation and rename the newly created '(age
-1)' column to 'age'
# Because select is a transformation and Spark uses lazy evaluation, no jobs, stages,
# or tasks will be launched when we run this code.
subDF = dataDF.select('last_name', 'first_name', 'ssn', 'occupation', (dataDF.age -
1).alias('age'))
```

Let's take a look at the query plan.

```
subDF.explain(True)

== Parsed Logical Plan ==
'Project [unresolvedalias('last_name', None), unresolvedalias('first_name', None), unresol
vedalias('ssn', None), unresolvedalias('occupation', None), (age#61L - 1) AS age#112]
+- AnalysisBarrier
   +- LogicalRDD [last_name#57, first_name#58, ssn#59, occupation#60, age#61L], false

== Analyzed Logical Plan ==
last_name: string, first_name: string, ssn: string, occupation: string, age: bigint
Project [last_name#57, first_name#58, ssn#59, occupation#60, (age#61L - cast(1 as bigin
t)) AS age#112L]
+- LogicalRDD [last_name#57, first_name#58, ssn#59, occupation#60, age#61L], false
```

```

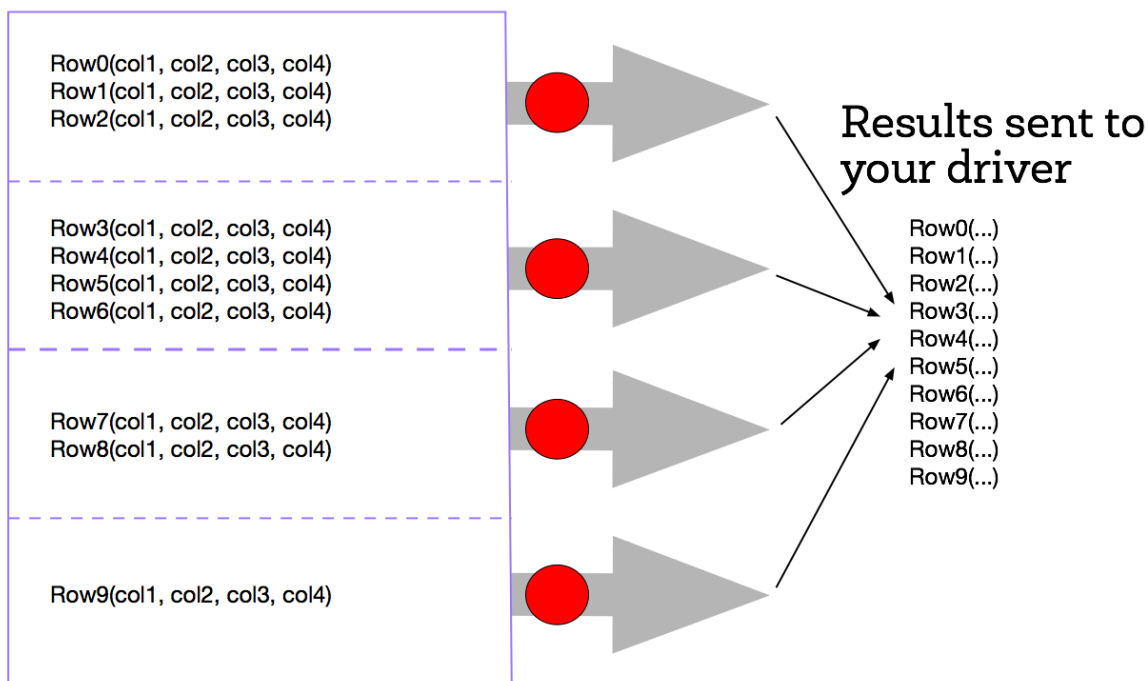
== Optimized Logical Plan ==
Project [last_name#57, first_name#58, ssn#59, occupation#60, (age#61L - 1) AS age#112L]
+- LogicalRDD [last_name#57, first_name#58, ssn#59, occupation#60, age#61L], false

== Physical Plan ==
*(1) Project [last_name#57, first_name#58, ssn#59, occupation#60, (age#61L - 1) AS age#112L]
+- *(1) Scan ExistingRDD[last_name#57,first_name#58,ssn#59,occupation#60,age#61L]

```

(3d) Use *collect* to view results

`collect()`: Gathers the entries from all partitions into the driver



To see a list of elements decremented by one, we need to create a new list on the driver from the the data distributed in the executor nodes. To do this we can call the `collect()` method on our `DataFrame`. `collect()` is often used after transformations to ensure that we are only returning a *small* amount of data to the driver. This is done because the data returned to the driver must fit into the driver's available memory. If not, the driver will crash.

The `collect()` method is the first action operation that we have encountered. Action operations cause Spark to perform the (lazy) transformation operations that are required to compute the values returned by the action. In our example, this means that tasks will now be launched to perform the `createDataFrame`, `select`, and `collect` operations.

In the diagram, the dataset is broken into four partitions, so four `collect()` tasks are launched. Each task collects the entries in its partition and sends the result to the driver, which creates a list of the values, as shown in the figure below.

Now let's run `collect()` on `subDF`.

```
# Let's collect the data
results = subDF.collect()
print results
```

```
[Row(last_name=u'Brown', first_name=u'Jason', ssn=u'160-37-9051', occupation=u'Aid worke
r', age=37), Row(last_name=u'Morrison', first_name=u'Earl', ssn=u'361-94-4342', occupati
on=u'Teacher, music', age=25), Row(last_name=u'Young', first_name=u'Christian', ssn=u'76
9-27-5887', occupation=u'Sales professional, IT', age=19), Row(last_name=u'George', firs
t_name=u'Daniel', ssn=u'175-24-7915', occupation=u'Geochemist', age=41), Row(last_name=
u'Lee', first_name=u'Dawn', ssn=u'310-69-7326', occupation=u'Fine artist', age=24), Row
(last_name=u'Hamilton', first_name=u'Matthew', ssn=u'099-90-9730', occupation=u'Best bo
y', age=42), Row(last_name=u'Amanda', first_name=u'Miss', ssn=u'476-06-5497', occupation
=u'Engineer, production', age=41), Row(last_name=u'Fernandez', first_name=u'Thomas', ssn
=u'722-09-8354', occupation=u'Psychologist, counselling', age=4), Row(last_name=u'Kirk',
first_name=u'Jennifer', ssn=u'715-56-1708', occupation=u'Runner, broadcasting/film/vide
o', age=3), Row(last_name=u'Young', first_name=u'Todd', ssn=u'123-48-8354', occupation=
u'Emergency planning/management officer', age=15), Row(last_name=u'Cook', first_name=u'A
my', ssn=u'293-22-0265', occupation=u'Scientist, forensic', age=27), Row(last_name=u'Mar
tinez', first_name=u'Mark', ssn=u'041-23-3263', occupation=u'Broadcast presenter', age=2
2), Row(last_name=u'Walker', first_name=u'Aaron', ssn=u'725-61-1132', occupation=u'Arts
administrator', age=41), Row(last_name=u'Dennis', first_name=u'Edward', ssn=u'268-79-433
0', occupation=u'Chief Marketing Officer', age=12), Row(last_name=u'Hebert', first_name=
u'Meredith', ssn=u'077-96-8349', occupation=u'Surveyor, land/geomatics', age=13), Row(la
st_name=u'Hess', first_name=u'Phyllis', ssn=u'061-88-1648', occupation=u'Product manage
r', age=31), Row(last_name=u'Reed', first_name=u'Kayla', ssn=u'582-28-0099', occupation=
```

A better way to visualize the data is to use the `show()` method. If you don't tell `show()` how many rows to display, it displays 20 rows.

```
subDF.show()
```

```
+-----+-----+-----+-----+
|last_name|first_name|      ssn|      occupation|age|
+-----+-----+-----+-----+
|   Brown|   Jason|160-37-9051|   Aid worker| 37|
|Morrison|   Earl|361-94-4342|Teacher, music| 25|
|   Young|Christian|769-27-5887|Sales professiona...| 19|
```

```
| George| Daniel|175-24-7915| Geochemist| 41|
| Lee| Dawn|310-69-7326| Fine artist| 24|
| Hamilton| Matthew|099-90-9730| Best boy| 42|
| Amanda| Miss|476-06-5497|Engineer, production| 41|
| Fernandez| Thomas|722-09-8354|Psychologist, cou...| 4|
| Kirk| Jennifer|715-56-1708|Runner, broadcast...| 3|
| Young| Todd|123-48-8354|Emergency plannin...| 15|
| Cook| Amy|293-22-0265| Scientist, forensic| 27|
| Martinez| Mark|041-23-3263| Broadcast presenter| 22|
| Walker| Aaron|725-61-1132| Arts administrator| 41|
| Dennis| Edward|268-79-4330|Chief Marketing O...| 12|
| Hebert| Meredith|077-96-8349|Surveyor, land/ge...| 13|
| Hess| Phyllis|061-88-1648| Product manager| 31|
| Reed| Kayla|582-28-0099|Maintenance engineer| 10|
| Miller| Dwayne|386-07-6013| Set designer| 4|
```

If you'd prefer that `show()` not truncate the data, you can tell it not to:

```
subDF.show(n=30, truncate=False)
```

```
+-----+-----+-----+-----+-----+
|last_name|first_name|ssn      |occupation                               |age|
+-----+-----+-----+-----+-----+
|Brown    |Jason     |160-37-9051|Aid worker                               |37|
|Morrison |Earl      |361-94-4342|Teacher, music                           |25|
|Young    |Christian |769-27-5887|Sales professional, IT                   |19|
|George   |Daniel    |175-24-7915|Geochemist                               |41|
|Lee      |Dawn      |310-69-7326|Fine artist                              |24|
|Hamilton |Matthew   |099-90-9730|Best boy                                 |42|
|Amanda   |Miss      |476-06-5497|Engineer, production                     |41|
|Fernandez|Thomas    |722-09-8354|Psychologist, counselling                |4|
|Kirk     |Jennifer  |715-56-1708|Runner, broadcasting/film/video          |3|
|Young    |Todd      |123-48-8354|Emergency planning/management officer    |15|
|Cook     |Amy       |293-22-0265|Scientist, forensic                      |27|
|Martinez |Mark      |041-23-3263|Broadcast presenter                      |22|
|Walker   |Aaron     |725-61-1132|Arts administrator                       |41|
|Dennis   |Edward    |268-79-4330|Chief Marketing Officer                  |12|
|Hebert   |Meredith  |077-96-8349|Surveyor, land/geomatics                 |13|
|Hess     |Phyllis   |061-88-1648|Product manager                          |31|
|Reed     |Kayla     |582-28-0099|Maintenance engineer                     |10|
|Miller   |Dwayne    |386-07-6013|Set designer                             |4|
```

In Databricks, there's an even nicer way to look at the values in a DataFrame: The `display()` helper function.

```
display(subDF)
```

last_name	first_name	ssn	occupation

Brown	Jason	160-37-9051	Aid worker
Morrison	Earl	361-94-4342	Teacher, music
Young	Christian	769-27-5887	Sales professional, IT
George	Daniel	175-24-7915	Geochemist
Lee	Dawn	310-69-7326	Fine artist
Hamilton	Matthew	099-90-9730	Best boy
Amanda	Miss	476-06-5497	Engineer, production

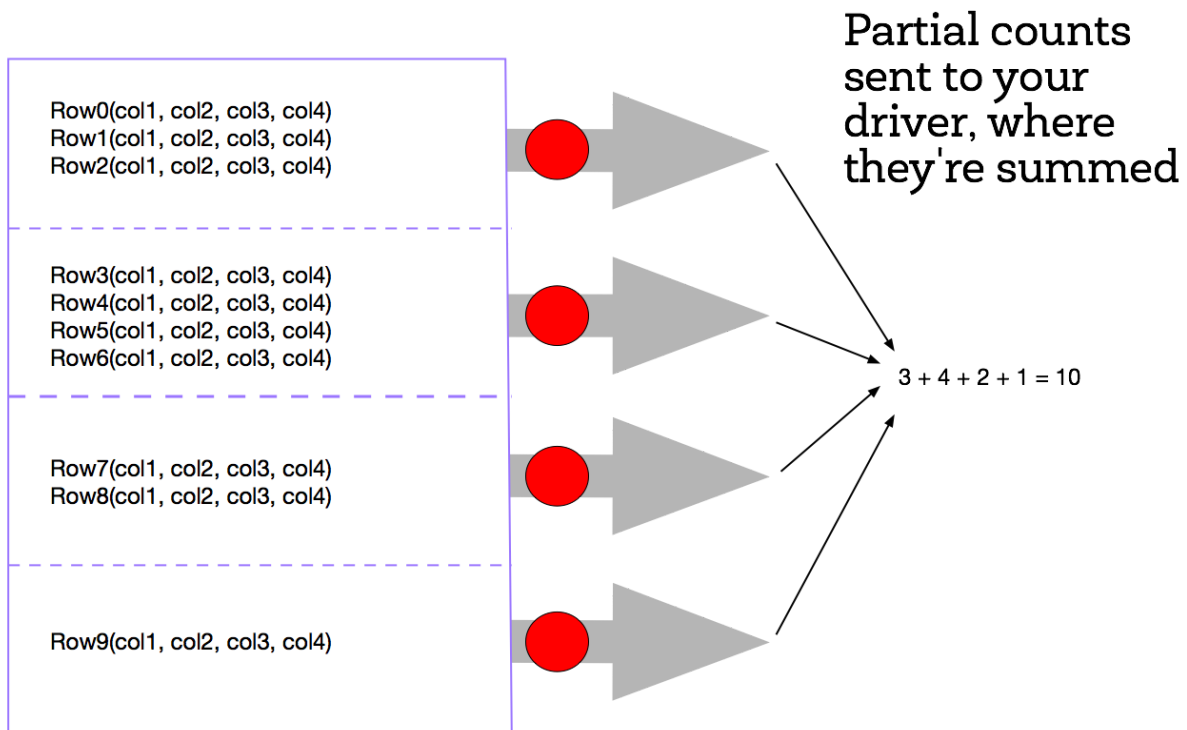
Showing the first 1000 rows.



(3e) Use *count* to get total

One of the most basic jobs that we can run is the `count()` job which will count the number of elements in a `DataFrame`, using the `count()` action. Since `select()` creates a new `DataFrame` with the same number of elements as the starting `DataFrame`, we expect that applying `count()` to each `DataFrame` will return the same result.

count () : Each task counts the rows in one partition



Note that because `count()` is an action operation, if we had not already performed an action with `collect()`, then Spark would now perform the transformation operations when we executed `count()`.

Each task counts the entries in its partition and sends the result to your SparkContext, which adds up all of the counts. The figure on the right shows what would happen if we ran `count()` on a small example dataset with just four partitions.

```
print dataDF.count()
print subDF.count()
```

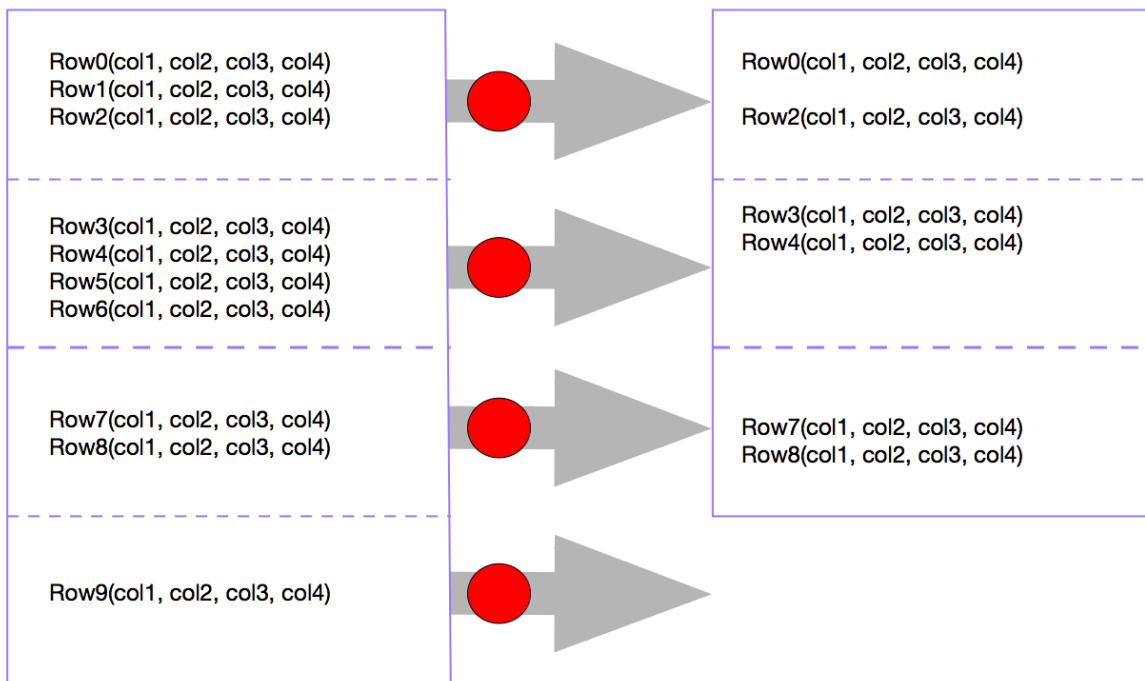
```
10000
10000
```

(3f) Apply transformation *filter* and view results with *collect*

Next, we'll create a new DataFrame that only contains the people whose ages are less than 10. To do this, we'll use the `filter()` transformation. (You can also use `where()`, an alias for `filter()`, if you prefer something more SQL-like). The `filter()` method is a transformation operation that creates a new DataFrame from the input DataFrame, keeping only values that match the filter expression.

The figure shows how this might work on the small four-partition dataset.

`filter()`: Each task makes a new partition with entries from the original partition that have an "age" column value less than 10.



To view the filtered list of elements less than 10, we need to create a new list on the driver from the distributed data on the executor nodes. We use the `collect()` method to return a list that contains all of the elements in this filtered DataFrame to the driver program.

```
filteredDF = subDF.filter(subDF.age < 10)
filteredDF.show(truncate=False)
filteredDF.count()
```

```
+-----+-----+-----+-----+-----+
+
```

last_name	first_name	ssn	occupation	age
Fernandez	Thomas	722-09-8354	Psychologist, counselling	4
Kirk	Jennifer	715-56-1708	Runner, broadcasting/film/video	3
Miller	Dwayne	386-07-6013	Set designer	4
Williams	Cassie	386-39-5490	Historic buildings inspector/conservation officer	9
Rose	Daniel	737-44-0894	Diplomatic Services operational officer	5
Lewis	David	695-60-0033	Field seismologist	7
Rodgers	Beverly	819-57-7194	Records manager	2
Rubio	Christopher	750-41-6999	Surveyor, planning and development	19

(These are some *seriously* precocious children...)

Part 4: Python Lambda functions and User Defined Functions

Python supports the use of small one-line anonymous functions that are not bound to a name at runtime.

`lambda` functions, borrowed from LISP, can be used wherever function objects are required. They are syntactically restricted to a single expression. Remember that `lambda` functions are a matter of style and using them is never required - semantically, they are just syntactic sugar for a normal function definition. You can always define a separate normal function instead, but using a `lambda` function is an equivalent and more compact form of coding. Ideally you should consider using `lambda` functions where you want to encapsulate non-reusable code without littering your code with one-line functions.

Here, instead of defining a separate function for the `filter()` transformation, we will use an inline `lambda()` function and we will register that `lambda` as a Spark *User Defined Function* (UDF). **A UDF is a special wrapper around a function, allowing the function to be used in a `DataFrame` query.**

```

from pyspark.sql.types import BooleanType
less_ten = udf(lambda s: s < 10, BooleanType()) # Datatype needs to be defined in lambda
lambdaDF = subDF.filter(less_ten(subDF.age))
lambdaDF.show()
lambdaDF.count()

```

last_name	first_name	ssn	occupation	age
Fernandez	Thomas	722-09-8354	Psychologist, cou...	4
Kirk	Jennifer	715-56-1708	Runner, broadcast...	3
Miller	Dwayne	386-07-6013	Set designer	4
Williams	Cassie	386-39-5490	Historic building...	9
Rose	Daniel	737-44-0894	Diplomatic Servic...	5
Lewis	David	695-60-0033	Field seismologist	7
Rodgers	Beverly	819-57-7194	Records manager	2
Rubio	Christopher	750-41-6999	Surveyor, plannin...	9
Johnson	Amanda	494-58-8536	Higher education ...	2
Johnson	Nicholas	157-13-2050	IT consultant	2
Johnson	Candice	639-80-6841	Electrical engineer	4
Arias	Mark	501-67-7559	Management consul...	2
Jones	Jeremy	648-72-8898	Product manager	6
Delgado	Sonia	283-58-4951	Solicitor	5
Bishop	Jonathan	714-87-5335	Medical physicist	9
Pineda	Connor	041-63-6459	Dentist	9
Robinson	Adam	279-56-1630	Financial manager	1
Holloway	Julia	115-33-9587	Animator	1

```

# Let's collect the even values less than 10
even = udf(lambda s: s % 2 == 0, BooleanType())
evenDF = lambdaDF.filter(even(lambdaDF.age))
evenDF.show()
evenDF.count()

```

last_name	first_name	ssn	occupation	age
Fernandez	Thomas	722-09-8354	Psychologist, cou...	4
Miller	Dwayne	386-07-6013	Set designer	4
Rodgers	Beverly	819-57-7194	Records manager	2
Johnson	Amanda	494-58-8536	Higher education ...	2
Johnson	Nicholas	157-13-2050	IT consultant	2
Johnson	Candice	639-80-6841	Electrical engineer	4
Arias	Mark	501-67-7559	Management consul...	2
Jones	Jeremy	648-72-8898	Product manager	6
Fleming	Jodi	833-40-8181	Agricultural engi...	2
Powell	Kimberly	102-37-5719	Contracting civil...	4
Griffith	Valerie	182-03-1388	Exhibition designer	4
Peck	Douglas	133-19-9920	Secretary, company	0
Gallegos	Michael	203-15-1520	Water engineer	2
King	Jacqueline	506-33-4081	Risk manager	0
Hughes	Amy	255-17-3490	Chemical engineer	8

Williams	Nicole 574-72-0358	Advice worker	2
Mack	Bradley 662-80-6804	Market researcher	2
Howard	Nicholas 188-57-7205	Astronomer	6

Part 5: Additional DataFrame actions

Let's investigate some additional actions:

- `first()`
(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.first>)
- `take()`
(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.take>)

One useful thing to do when we have a new dataset is to look at the first few entries to obtain a rough idea of what information is available. In Spark, we can do that using actions like `first()`, `take()`, and `show()`. Note that for the `first()` and `take()` actions, the elements that are returned depend on how the DataFrame is *partitioned*.

Instead of using the `collect()` action, we can use the `take(n)` action to return the first n elements of the DataFrame. The `first()` action returns the first element of a DataFrame, and is equivalent to `take(1)[0]`.

```
print "first: {0}\n".format(filteredDF.first())
```

```
print "Four of them: {0}\n".format(filteredDF.take(4))
```

```
first: Row(last_name=u'Fernandez', first_name=u'Thomas', ssn=u'722-09-8354', occupation=
u'Psychologist, counselling', age=4)
```


```
Four of them: [Row(last_name=u'Fernandez', first_name=u'Thomas', ssn=u'722-09-8354', occ
upation=u'Psychologist, counselling', age=4), Row(last_name=u'Kirk', first_name=u'Jennif
er', ssn=u'715-56-1708', occupation=u'Runner, broadcasting/film/video', age=3), Row(last
_name=u'Miller', first_name=u'Dwayne', ssn=u'386-07-6013', occupation=u'Set designer', a
ge=4), Row(last_name=u'Williams', first_name=u'Cassie', ssn=u'386-39-5490', occupation=
u'Historic buildings inspector/conservation officer', age=9)]
```

This looks better:

```
display(filteredDF.take(4))
```

last_name ▼	first_name ▼	ssn ▼	occupation
Fernandez	Thomas	722-09-8354	Psychologist, counselling
Kirk	Jennifer	715-56-1708	Runner, broadcasting/fi
Miller	Dwayne	386-07-6013	Set designer

Williams	Cassie	386-39-5490	Historic buildings inspe
----------	--------	-------------	--------------------------



Part 6: Additional DataFrame transformations

(6a) *orderBy*

`orderBy()`

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.distinct>) allows you to sort a DataFrame by one or more columns, producing a new DataFrame.

For example, let's get the first five oldest people in the original (unfiltered) DataFrame. We can use the `orderBy()` transformation. `orderBy` takes one or more columns, either as *names* (strings) or as `Column` objects. To get a `Column` object, we use one of two notations on the DataFrame:

- Pandas-style notation: `filteredDF.age`
- Subscript notation: `filteredDF['age']`


Both of those syntaxes return a `Column`, which has additional methods like `desc()` (for sorting in descending order) or `asc()` (for sorting in ascending order, which is the default).

Here are some examples:

```
dataDF.orderBy(dataDF['age']) # sort by age in ascending order; returns a new DataFrame
dataDF.orderBy(dataDF.last_name.desc()) # sort by last name in descending order
```

```
# Get the five oldest people in the list. To do that, sort by age in descending order.
display(dataDF.orderBy(dataDF.age.desc()).take(5))
```

last_name ▼	first_name ▼	ssn ▼
Tate	Marvin	533-78-7142
Nicholson	James	899-28-6159
Welch	Jeffrey	366-71-8910
Lawson	James	082-05-2881
Miller	Daniel	384-37-2604



Let's reverse the sort order. Since ascending sort is the default, we can actually use a `Column` object expression or a simple string, in this case. **The `desc()` and `asc()` methods are only defined on `Column`. Something like `orderBy('age'.desc())` would not work**, because there's no `desc()` method on Python string objects. That's why we needed the column expression. But if we're just using the defaults, we can pass a string column name into `orderBy()`. This is sometimes easier to read.

```
display(dataDF.orderBy('age').take(5))
```

last_name ▼	first_name ▼	ssn ▼
Myers	Donna	844-84-7394
Munoz	Laura	776-13-4126
Rivera	Katelyn	445-27-3293
Schmidt	Maria	760-26-2009
Singleton	Jennifer	652-23-8827



(6b) *distinct* and *dropDuplicates*

```
distinct()
```

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.distinct>) filters out duplicate rows, and it considers all columns. Since our data is completely randomly generated (by `fake-factory`), it's extremely unlikely that there are any duplicate rows:

```
print dataDF.count()
print dataDF.distinct().count()
```

```
10000
10000
```

To demonstrate `distinct()`, let's create a quick throwaway dataset.

```
tempDF = sqlContext.createDataFrame([("Joe", 1), ("Joe", 1), ("Anna", 15), ("Anna", 12),
("Ravi", 5)], ('name', 'score'))
```

```
tempDF.show()
```

```
+-----+-----+
|name|score|
```

```
+----+-----+
| Joe|    1|
| Joe|    1|
| Anna|   15|
| Anna|   12|
| Ravi|    5|
+----+-----+
```

```
tempDF.distinct().show()
```

```
+----+-----+
| name|score|
+----+-----+
| Joe|    1|
| Ravi|    5|
| Anna|   12|
| Anna|   15|
+----+-----+
```

Note that one of the ("Joe", 1) rows was deleted, but both rows with name "Anna" were kept, because all columns in a row must match another row for it to be considered a duplicate.

```
dropDuplicates()
```

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.dropDup>) is like `distinct()`, except that `dropDuplicates()` allows us to **specify the columns to compare**. For instance, we can use it to drop all rows where the first name and last name duplicates (ignoring the occupation and age columns).

```
print dataDF.count()
print dataDF.dropDuplicates(['first_name', 'last_name']).count()
```

```
10000
9347
```

(6c) drop

```
drop()
```

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.drop>) is like the opposite of `select()`: Instead of selecting specific columns from a DataFrame, it drops a specified column from a DataFrame.

Here's a simple use case: Suppose you're reading from a 1,000-column CSV file, and you have to get rid of five of the columns. Instead of selecting 995 of the columns, it's easier just to drop the five you don't want.

```
dataDF.drop('occupation').drop('age').show()
```

```
+-----+-----+-----+
|last_name|first_name|      ssn|
+-----+-----+-----+
|   Brown|   Jason|160-37-9051|
|Morrison|   Earl|361-94-4342|
|   Young|Christian|769-27-5887|
|   George|Daniel|175-24-7915|
|   Lee|   Dawn|310-69-7326|
|Hamilton|Matthew|099-90-9730|
|  Amanda|   Miss|476-06-5497|
|Fernandez|Thomas|722-09-8354|
|   Kirk|Jennifer|715-56-1708|
|   Young|   Todd|123-48-8354|
|   Cook|   Amy|293-22-0265|
|Martinez|   Mark|041-23-3263|
|   Walker|Aaron|725-61-1132|
|   Dennis|Edward|268-79-4330|
|Hebert|Meredith|077-96-8349|
|   Hess|Phyllis|061-88-1648|
|   Reed|Kayla|582-28-0099|
|   Miller|Dwayne|386-07-6013|
```

(6d) *groupBy*

```
groupBy()
```

((<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.groupBy>) is **one of the most powerful transformations**. It allows you to **perform aggregations on a DataFrame**.

Unlike other DataFrame transformations, `groupBy()` does *not* return a DataFrame. Instead, it returns a special `GroupedData`

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData>) object that contains various aggregation functions.

The most commonly used aggregation function is [count()]

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.count>)

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.count>)

but there are others (like `sum()`)

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.sum>),

max()

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.max>),
and avg()

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData.avg>).

These aggregation functions typically create a new column and return a new DataFrame.

```
dataDF.groupBy('occupation').count().show(truncate=False)
```

```
+-----+-----+
|occupation|count|
+-----+-----+
|Diplomatic Services operational officer|16|
|Librarian, academic|18|
|Catering manager|23|
|Retail merchandiser|14|
|Designer, ceramics/pottery|18|
|Engineer, aeronautical|11|
|English as a second language teacher|20|
|Early years teacher|17|
|Occupational hygienist|16|
|Patent examiner|12|
|Primary school teacher|14|
|Clinical molecular geneticist|17|
|Control and instrumentation engineer|18|
|Estate agent|19|
|Art therapist|13|
|Transport planner|21|
|Applications developer|19|
|Petroleum engineer|23|
```

```
dataDF.groupBy().avg('age').show(truncate=False)
```

```
+-----+
|avg(age)|
+-----+
|23.4697|
+-----+
```

We can also use `groupBy()` to do another useful aggregations:

```
print "Maximum age: {}".format(dataDF.groupBy().max('age').first()[0])
print "Minimum age: {}".format(dataDF.groupBy().min('age').first()[0])
```

Maximum age: 47

Minimum age: 1

Show the schema of dataDF

```
dataDF.schema
```

```
Out[58]: StructType(List(StructField(last_name,StringType,true),StructField(first_name,StringType,true),StructField(ssn,StringType,true),StructField(occupation,StringType,true),StructField(age,LongType,true)))
```

(6e) *sample* (optional)

When analyzing data, the `sample()`

(<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.sample>)

transformation is often quite useful. It returns a new `DataFrame` with a random sample of elements from the dataset. It takes in a `withReplacement` argument, which specifies whether it is okay to randomly pick the same item multiple times from the parent `DataFrame` (so when `withReplacement=True`, you can get the same item back multiple times). It takes in a `fraction` parameter, which specifies the fraction elements in the dataset you want to return.

(So a `fraction` value of `0.20` returns 20% of the elements in the `DataFrame`.) It also takes an optional `seed` parameter that allows you to specify a seed value for the random number generator, so that reproducible results can be obtained.

```
sampldDF = dataDF.sample(withReplacement=False, fraction=0.10)
print sampldDF.count()
sampldDF.show()
# The result is not exactly 10000*0.1 = 1000, but 943 (1st time), 976 (2nd time), 1011 (3rd time) ...
```

```
1011
+-----+-----+-----+-----+-----+
|last_name|first_name|      ssn|      occupation|age|
+-----+-----+-----+-----+-----+
|   George|   Daniel|175-24-7915|      Geochemist| 42|
|    Ford|  Joshua|623-03-3910|  Drilling engineer| 29|
|  Jenkins|  Robert|828-24-7280|   Data scientist| 27|
|   Arias|    Mark|501-67-7559|Management consul...| 3|
|   Jones|  Jeremy|648-72-8898|   Product manager| 7|
|  Larson|  Joseph|076-36-8031|Embryologist, cli...| 26|
|Valencia|  Alicia|274-89-5763|Lecturer, further...| 4|
|   Price|  Ronald|694-58-1212|Runner, broadcast...| 25|
|  Morrow|   Alex|533-54-4033|   Town planner| 2|
| Jackson| Cynthia|699-10-4819|Fitness centre ma...| 42|
|   Miller|   Jacob|691-69-2880|   Chief of Staff| 29|
|Roberts|    Luis|562-33-8154|Psychologist, cli...| 11|
|   George| Kathryn|068-01-3503|   Designer, graphic| 35|
|   Barron| Courtney|519-80-0824|Engineer, broadca...| 10|
```

Williams	Jerry	062-08-0637	Engineer, water	31
Newman	Rebecca	050-93-5616	Barista	13

```
print dataDF.sample(withReplacement=False, fraction=0.05).count()
```

474

Part 7: Caching DataFrames and storage options

(7a) Caching DataFrames

For efficiency Spark keeps your DataFrames in memory. (More formally, it keeps the *RDDs* that implement your DataFrames in memory.) By keeping the contents in memory, Spark can quickly access the data. **However, memory is limited, so if you try to keep too many partitions in memory, Spark will automatically delete partitions from memory to make space for new ones.** If you later refer to one of the deleted partitions, Spark will automatically recreate it for you, but that takes time.

So, **if you plan to use a DataFrame more than once, then you should tell Spark to cache it.** You can use the `cache()` operation to keep the DataFrame in memory. However, **you must still trigger an action on the DataFrame, such as `collect()` or `count()` before the caching will occur.** In other words, `cache()` is lazy: It merely tells Spark that the DataFrame should be cached *when the data is materialized*. You have to run an action to materialize the data; the DataFrame will be cached as a side effect. The next time you use the DataFrame, Spark will use the cached data, rather than recomputing the DataFrame from the original data.

You can see your cached DataFrame in the "Storage" section of the Spark web UI. If you click on the name value, you can see more information about where the DataFrame is stored.

```
# Cache the DataFrame
filteredDF.cache()
# Trigger an action
print filteredDF.count()
# Check if it is cached
print filteredDF.is_cached
```

2286

True

(7b) Unpersist and storage options

Spark automatically manages the partitions cached in memory. If it has more partitions than available memory, by default, it will evict older partitions to make room for new ones. For efficiency, once you are finished using cached DataFrame, you can optionally tell Spark to stop caching it in memory by using the DataFrame's `unpersist()` method to inform Spark that you no longer need the cached data.

Advanced: Spark provides many more options for managing how DataFrames are cached. For instance, you can tell Spark to spill cached partitions to disk when it runs out of memory, instead of simply throwing old ones away. You can explore the API for DataFrame's `persist()` (<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.persist>) operation using Python's `help()` (<https://docs.python.org/2/library/functions.html?highlight=help#help>) command. The `persist()` operation, optionally, takes a `pySpark StorageLevel` (<http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.StorageLevel>) object.

```
# If we are done with the DataFrame we can unpersist it so that its memory can be
reclaimed
filteredDF.unpersist()
# Check if it is cached
print filteredDF.is_cached

False
```

Part 8: Debugging Spark applications and lazy evaluation

How Python is Executed in Spark

Internally, Spark executes using a Java Virtual Machine (JVM). **pySpark runs Python code in a JVM using [Py4J]**(<http://py4j.sourceforge.net> (<http://py4j.sourceforge.net>)). Py4J enables Python programs running in a Python interpreter to dynamically access Java objects in a Java Virtual Machine. Methods are called as if the Java objects resided in the Python interpreter and Java collections can be accessed through standard Python collection methods. Py4J also enables Java programs to call back Python objects.

Because pySpark uses Py4J, coding errors often result in a complicated, confusing stack trace that can be difficult to understand. In the following section, we'll explore how to understand stack traces.

(8a) Challenges with lazy evaluation using transformations and actions

Spark's use of lazy evaluation can make debugging more difficult because code is not always executed immediately. To see an example of how this can happen, let's first define a broken filter function. Next we perform a `filter()` operation using the broken filtering function. No error will occur at this point due to Spark's use of lazy evaluation.

The `filter()` method will not be executed *until* an action operation is invoked on the `DataFrame`. We will perform an action by using the `count()` method to return a list that contains all of the elements in this `DataFrame`.

```
def brokenTen(value):
    """Incorrect implementation of the ten function.

    Note:
        The `if` statement checks an undefined variable `val` instead of `value`.

    Args:
        value (int): A number.

    Returns:
        bool: Whether `value` is less than ten.

    Raises:
        NameError: The function references `val`, which is not available in the local or
global
        namespace, so a `NameError` is raised.
    """
    if (val < 10):
        return True
    else:
        return False

btUDF = udf(brokenTen)
brokenDF = subDF.filter(btUDF(subDF.age) == True)

# Now we'll see the error
# Click on the `+` button to expand the error and scroll through the message.
brokenDF.count()
```

```
org.apache.spark.SparkException: Job aborted due to stage failure: Task 6 in stage 95.
0 failed 1 times, most recent failure: Lost task 6.0 in stage 95.0 (TID 904, localhos
t, executor driver): org.apache.spark.api.python.PythonException: Traceback (most rece
nt call last):
```

(8b) Finding the bug

When the `filter()` method is executed, Spark calls the UDF. Since our UDF has an error in the underlying filtering function `brokenTen()`, an error occurs.

Scroll through the output "Py4JJavaError Traceback (most recent call last)" part of the cell and first you will see that the line that generated the error is the `count()` method line. There is *nothing wrong with this line*. However, it is an action and that caused other methods to be executed. Continue scrolling through the Traceback and you will see the following error line:

```
NameError: global name 'val' is not defined
```

Looking at this error line, we can see that we used the wrong variable name in our filtering function `brokenTen()`.

(8c) Moving toward expert style

As you are learning Spark, I recommend that you write your code in the form:

```
df2 = df1.transformation1()
df2.action1()
df3 = df2.transformation2()
df3.action2()
```

Using this style will make debugging your code much easier as it makes errors easier to localize - errors in your transformations will occur when the next action is executed.

Once you become more experienced with Spark, you can write your code with the form:

```
df.transformation1().transformation2().action()
```

We can also use `lambda()` functions instead of separately defined functions when their use improves readability and conciseness.

```
# Cleaner code through lambda use
myUDF = udf(lambda v: v < 10)
subDF.filter(myUDF(subDF.age) == True)
```

```
Out[79]: DataFrame[last_name: string, first_name: string, ssn: string, occupation: string, age: bigint]
```

(8d) Readability and code style

To make the expert coding style more readable, enclose the statement in parentheses and **put each method, transformation, or action on a separate line.**

```
+-----+-----+
|concat(first_name,  , last_name)|occupation|
+-----+-----+
|Jason Brown                    |Aid worker|
|Earl Morrison                  |Teacher, music|
|Daniel George                  |Geochemist|
|Dawn Lee                       |Fine artist|
|Matthew Hamilton               |Best boy|
|Miss Amanda                    |Engineer, production|
|Amy Cook                       |Scientist, forensic|
|Mark Martinez                  |Broadcast presenter|
|Aaron Walker                   |Arts administrator|
|Phyllis Hess                   |Product manager|
|Sean Weeks                     |Administrator, education|
|Taylor Dawson                  |Psychologist, occupational|
|Steven Fisher                  |Therapist, drama|
|Stephanie Johnson              |Presenter, broadcasting|
|Tonya Davis                    |Communications engineer|
|David Shaw                     |Production manager|
|Victoria Paul                  |Environmental manager|
|Jeff Hays                      |Music tutor|
```