Dataframes

Dataframes are a special type of RDDs. They are similar to, but not the same as, pandas dataframes. They are used to store two dimensional data, similar to the type of data stored in a spreadsheet. Each column in a dataframe can have a different type and each row contains a record.

Spark DataFrames are similar to pandas DataFrames. With the important difference that spark DataFrames are **distributed** data structures, based on RDDs.

In [20]:
```
from pyspark import SparkContext
sc = SparkContext(master="local[4]")
```

In [22]:
```
# Just like using Spark requires having a SparkContext, using SQL requires an SQLCon
text
sqlContext = SQLContext(sc)
sqlContext
```

Out[22]:
```
<pyspark.sql.context.SQLContext at 0x7f2312a51bd0>
```

In [23]:
```
# One way to create a DataFrame is to first define an RDD from a list of rows
some_rdd = sc.parallelize([Row(name=u"John", age=19),
                           Row(name=u"Smith", age=23),
                           Row(name=u"Sarah", age=18)])
some_rdd.collect()
```

Out[23]:
```
[Row(age=19, name=u'John'),
 Row(age=23, name=u'Smith'),
 Row(age=18, name=u'Sarah')]
```

In [24]:
```
# The DataFrame is created from the RDD or Rows
# Infer schema from the first row, create a DataFrame and print the schema
some_df = sqlContext.createDataFrame(some_rdd)
some_df.printSchema()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

In [25]:

```
# A dataframe is an RDD of rows plus information on the schema.
# In our case, the content of the RDD is the same as the content of the dataframe.
print type(some_rdd),type(some_df)
print 'some_df =',some_df.collect()
print 'some_rdd=',some_rdd.collect()
```

```
<class 'pyspark.rdd.RDD'> <class 'pyspark.sql.dataframe.DataFrame'>
some_df = [Row(age=19, name=u'John'), Row(age=23, name=u'Smith'), Row(age=18, n
ame=u'Sarah')]
some_rdd= [Row(age=19, name=u'John'), Row(age=23, name=u'Smith'), Row(age=18,
name=u'Sarah')]
```

Example of using an RDD

In [26]:
```
# In this case we create the dataframe from an RDD of tuples (rather than Rows) and pr
ovide the schema explicitly
another_rdd = sc.parallelize([("John", 19), ("Smith", 23), ("Sarah", 18)])
# Schema with two fields - person_name and person_age
schema = StructType([StructField("person_name", StringType(), False),
                     StructField("person_age", IntegerType(), False)])

# Create a DataFrame by applying the schema to the RDD and print the schema
another_df = sqlContext.createDataFrame(another_rdd, schema)
another_df.printSchema()
# root
#  |-- age: binteger (nullable = true)
#  |-- name: string (nullable = true)
```

```
root
 |-- person_name: string (nullable = false)
 |-- person_age: integer (nullable = false)
```

Loading dataframes from JSON files

JSON is a very popular readable file format for storing structured data. Among it's many uses are **twitter**, javascript communication packets, and many others. In fact this notebook file (with the extension .ipynb is in json format. JSON can also be used to store tabular data and can be easily loaded into a dataframe.

In [27]:
```
# when loading json files you can specify either a single file or a directory containing ma
ny json files.
path = "../../Data/people.json"
!cat $path
```

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

In [28]:

```
# Create a DataFrame from the file(s) pointed to by path
people = sqlContext.read.json(path)
print 'people is a',type(people)
# The inferred schema can be visualized using the printSchema() method.
print
people.printSchema()
# root
#  |-- age: IntegerType
#  |-- name: StringType
```

```
people is a <class 'pyspark.sql.dataframe.DataFrame'>

root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

What formats does spark-sql support?

According to this post, spark supports many formats. However, I could not find definite documentation on which formats are supported.

In terms of syntax, you can use the format

```
sqlContext.read.format('json').load('python/test_support/sql/people.json')
```

Where instead of json you can use parquet,text and suppposedly other formats, but I could not find an authoritative list of formats. It seems that csv is not supported at this time.

Using HiveSQL queries on DataFrames

You can use Hive select syntax to select a subset of the rows in a dataframe. Spark supports a subset of the Hive SQL query language

To use sql on a dataframe you need to first register it as a TempTable.

In [49]:
```
people.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

In [54]:
```
# Register this DataFrame as a table.
people.registerTempTable("people")

# SQL statements can be run by using the sql methods provided by sqlContext
teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age
 <= 19")
```

Using Hive and other RDBMS

Spark can interact efficiently with the Hive column-oriented RDBMS. That adds both the flexibility of using SQL and the benefit of performing SQL operations on disk, without necessarily loading all of the data to memory.

We will not use Hive in this course. However we will use the Parquet file format which allows performing select operations before loading rows into memory. This is often a much faster approach than reading the whole file and then performing a filter() operation.

Parquet files

Parquet is a columnar format that is supported by many other data processing systems. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data.

In [32]:

```
dir='../../Data'
parquet_file=dir+"/users.parquet"
!ls -ld $dir/*.parquet
!rm -rf ../../Data/namesAndFavColors.parquet
```

```
drwxr-xr-x  192 yoavfreund  staff  6528 Apr  1 14:59 ../../Data/Weather_sampled.parquet
-rw-r--r--@   1 yoavfreund  staff   615 Mar 31 13:47 ../../Data/users.parquet
```

In [33]:

```
#load a Parquet file
df = sqlContext.read.load(parquet_file)
df.show()
```

```
+------+-------------+---------------+
|  name|favorite_color|favorite_numbers|
+------+-------------+---------------+
|Alyssa|         null|  [3, 9, 15, 20]|
|   Ben|          red|             []|
+------+-------------+---------------+
```

In [34]:

```
df2=df.select("name", "favorite_color")
df2.show()
```

```
+------+--------------+
|  name|favorite_color|
+------+--------------+
|Alyssa|          null|
|   Ben|           red|
+------+--------------+
```

In [35]:

```
df2.write.save(dir+"/namesAndFavColors.parquet")
!ls -ld $dir/*.parquet
```

```
drwxr-xr-x  192 yoavfreund  staff  6528 Apr  1 14:59 ../../Data/Weather_sampled.parquet
drwxr-xr-x   10 yoavfreund  staff   340 Apr  1 22:18 ../../Data/namesAndFavColors.parquet
-rw-r--r--@   1 yoavfreund  staff   615 Mar 31 13:47 ../../Data/users.parquet
```

A somewhat bigger example

This file contains 1/1000 of the original file, which contains about 9 million lines and whose size is about 7GB.

In [13]:

```
%cd ../../Data
import urllib
# If these commands don't work, you can download the files using your rowser
f=urllib.urlretrieve("https://drive.google.com/open?id=0B8IGBNGc5gVANWtlUElzd0JxV1E")
!curl  'https://drive.google.com/open?id=0B8IGBNGc5gVANWtlUElzd0JxV1E' > test.tgz
!ls -lrt
#!tar xzvf  weather.tgz
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   261    0   261    0     0    951      0 --:--:-- --:--:-- --:--:--   952
total 2528
-rw-r--r--@  1 yoavfreund  503    1257260 Jan  3  2013 Moby-Dick.txt
-rw-r--r--   1 yoavfreund  staff      679 Sep  8  2015 states.txt
-rw-r--r--@  1 yoavfreund  staff      615 Mar 31 13:47 users.parquet
-rw-r--r--@  1 yoavfreund  staff       73 Mar 31 17:19 people.json
-rw-r--r--   1 yoavfreund  staff       43 Apr  1 21:56 example.csv
drwxr-xr-x  10 yoavfreund  staff      340 Apr  1 22:18 namesAndFavColors.parquet
-rw-r--r--   1 yoavfreund  staff     7793 Apr  2 00:35 weather.tgz
-rw-r--r--   1 yoavfreund  staff     7793 Apr  2 00:35 weather2.tgz
-rw-r--r--   1 yoavfreund  staff      261 Apr  2 00:41 test.tgz
```

In [67]:

```
parquet_file=dir+"/Weather_sampled.parquet"
df = sqlContext.read.load(parquet_file)
print 'count=',df.count()
print 'columns=',df.columns[:5]
```

```
count= 9562
columns= ['station', 'measurement', 'year', '1', '2']
```

Count the number of occurances of each measurement

In [65]:
```
L=df.groupBy('measurement').count().collect()
D=[(e.measurement,e['count']) for e in L]
sorted(D,key=lambda x:x[1], reverse=True)[:6]
```

Out[65]:
```
[(u'PRCP', 2618),
 (u'TMAX', 1003),
 (u'TMIN', 949),
 (u'SNWD', 898),
 (u'SNOW', 870),
 (u'TOBS', 482)]
```

Reading only select lines

Suppose we are only interested in snow measurements. We can apply an SQL query directly to the parquet files. As the data is organized in columnar structure, we can do the selection efficiently without loading the whole file to memory.

Here the file is small, but in real applications it can consist of hundreds of millions of records. In such cases loading the data first to memory and then filtering it is very wasteful.

In [66]:

```
query='SELECT station,measurement,year FROM parquet.`%s` WHERE measurement
="SNOW"'%parquet_file
print query
df2 = sqlContext.sql(query)
print df2.count(),df2.columns
```

```
SELECT station,measurement,year FROM parquet.`../../Data/Weather_sampled.parquet
` WHERE measurement="SNOW"
870 ['station', 'measurement', 'year']
```

For more on DataFrames see:

- API for the DataFrame class

For more on Spark SQL see:

- API for the pyspark.sql module