Dataframes

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

In [1]:

```python
import findspark
findspark.init()
from pyspark import SparkContext
sc = SparkContext(master="local[4]")
sc.version
```

Out[1]:

```
u'2.1.0'
```

In [3]:

```python
# Just like using Spark requires having a SparkContext, using SQL requires an SQLContext
sqlContext = SQLContext(sc)
sqlContext
```

Out[3]:

```
<pyspark.sql.context.SQLContext at 0x10b12b0d0>
```

Constructing a DataFrame from an RDD of Rows

Each Row defines it's own fields, the schema is *inferred*.

In [4]:

```python
# 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[4]:

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

In [5]:

```
# 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 [6]:

```python
# A dataframe is an RDD of rows plus information on the schema.
# performing **collect()* on either the RDD or the DataFrame gives the same result.
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')]
```

Defining the Schema explicitly

The advantage of creating a DataFrame using a pre-defined schema allows the content of the RDD to be simple tuples, rather than rows.

In [7]:

```
# 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 disk

There are many maethods to load DataFrames from Disk. Here we will discuss three of these methods

1. JSON
2. CSV
3. Parquet

In addition, there are API's for connecting Spark to an external database. We will not discuss this type of connection in this class.

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 [8]:

```
# 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 [35]:

```python
# 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.
people.show()
```

```
people is a <class 'pyspark.sql.dataframe.DataFrame'>
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

In [37]:
```
people.printSchema()
```

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

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 [38]:

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

```
../../Data/users.parquet
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
+------+--------------+----------------+
```

In [12]:

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

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

In [40]:

```
outfilename="namesAndFavColors.parquet"
!rm -rf $dir/$outfilename
df2.write.save(dir+"/"+outfilename)
!ls -ld $dir/$outfilename
```

```
drwxr-xr-x  12 yoavfreund  staff  408 Apr 18 09:04 ../../Data/namesAndFavColors.parquet
```

Loading a dataframe from a pickle file

Here we are loading a dataframe from a pickle file stored on S3. The pickle file contains meterological data that we will work on in future classes.

In [44]:

```
#List is a list of Rows. Stored as a pickle file.
df=sqlContext.createDataFrame(List)
print df.count()
df.show(1)
```

```
12373
+---------+--------+---------+-----------+-----------+------+--------------------+------+--------+
|elevation|latitude|longitude|measurement|    station|undefs|              vector|  year|   label|
+---------+--------+---------+-----------+-----------+------+--------------------+------+--------+
|    181.4| 41.0092| -87.8242|       PRCP|USC00111458|     8|[00 00 00 00 00 0...|1991.0|BBSSBBSS|
+---------+--------+---------+-----------+-----------+------+--------------------+------+--------+
only showing top 1 row
```

In [45]:

```
#selecting a subset of the rows so it fits in slide.
df.select('station','year','measurement').show(5)
```

```
+-----------+------+-----------+
|    station|  year|measurement|
+-----------+------+-----------+
|USC00111458|1991.0|       PRCP|
|USC00111458|1994.0|       PRCP|
|USC00111458|1995.0|       PRCP|
|USC00111458|1996.0|       PRCP|
|USC00111458|1997.0|       PRCP|
+-----------+------+-----------+
only showing top 5 rows
```

In [18]:

```
### Save dataframe as Parquet repository
filename='%s/US_Weather_%s.parquet'%(data_dir,file_index)
!rm -rf $filename
df.write.save(filename)
```

* Parquet repositories are usually directories with many files.
* Parquet uses its column based format to compress the data.

In [20]:

```
!du -sh $filename
!du -sh $data_dir/$zip_file
```

```
4.2M	/Users/yoavfreund/projects/edX-Micro-Master-in-Data-Science/big-data-analytics-using-spark/Data/US_Weather_BBSSBBSS.parquet
3.3M	/Users/yoavfreund/projects/edX-Micro-Master-in-Data-Science/big-data-analytics-using-spark/Data/US_Weather_BBSSBBSS.csv.gz
```

Dataframe operations

Spark DataFrames allow operations similar to pandas Dataframes. We demonstrate some of those. For more, see this article

In [43]:

```
df.describe().select('station','elevation','measurement').show()
```

```
+-----------+------------------+-----------+
|    station|         elevation|measurement|
+-----------+------------------+-----------+
|      12373|             12373|      12373|
|       null|205.64884021660063|       null|
|       null|170.84234175167742|       null|
|US1ILCK0069|            -999.9|      PRCP|
|USW00014829|             305.1|      TOBS|
+-----------+------------------+-----------+
```

In [22]:

```
df.groupby('measurement').agg({'year': 'min', 'station':'count'}).show()
```

```
+-----------+---------+--------------+
|measurement|min(year)|count(station)|
+-----------+---------+--------------+
|       TMIN|   1893.0|          1859|
|       TOBS|   1901.0|          1623|
|       TMAX|   1893.0|          1857|
|       SNOW|   1895.0|          2178|
|       SNWD|   1902.0|          1858|
|       PRCP|   1893.0|          2998|
+-----------+---------+--------------+
```

Using SQL queries on DataFrames

There are two main ways to manipulate DataFrames:

Imperative manipulation

Using python methods such as .select and .groupby.

- Advantage: order of operations is specified.
- Disrdavantage : You need to describe both **what** is the result you want and **how** to get it.

Declarative Manipulation (SQL)

- Advantage: You need to describe only **what** is the result you want.
- Disadvantage: SQL does not have primitives for common analysis operations such as **covariance**

In [23]:
```
people.show()
```

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

In [24]:

```
# 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")
for each in teenagers.collect():
    print(each[0])
```

```
Justin
```

Counting the number of occurances of each measurement, imparatively

In [27]:

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

Out[27]:

```
[(u'TOBS', 1623),
 (u'TMAX', 1857),
 (u'SNWD', 1858),
 (u'TMIN', 1859),
 (u'SNOW', 2178),
 (u'PRCP', 2998)]
```

Counting the number of occurances of each measurement, declaratively.

In [28]:

```
sqlContext.registerDataFrameAsTable(df,'weather') #using older sqlContext instead of n
ewer (V2.0) sparkSession
query='SELECT measurement,COUNT(measurement) AS count FROM weather GROU
P BY measurement ORDER BY count'
print query
sqlContext.sql(query).show()
```

```
SELECT measurement,COUNT(measurement) AS count FROM weather GROUP BY m
easurement ORDER BY count
+-----------+-----+
|measurement|count|
+-----------+-----+
|       TOBS| 1623|
|       TMAX| 1857|
|       SNWD| 1858|
|       TMIN| 1859|
|       SNOW| 2178|
|       PRCP| 2998|
+-----------+-----+
```

Performing a map command

In order to perform map, you need to first transform the dataframe into an RDD.

In [29]:
```
df.rdd.map(lambda row:(row.longitude,row.latitude)).take(5)
```

Out[29]:
```
[(-87.8242, 41.0092),
 (-87.8242, 41.0092),
 (-87.8242, 41.0092),
 (-87.8242, 41.0092),
 (-87.8242, 41.0092)]
```

Approximately counting the number of distinct elements in column

In [30]:

```
import pyspark.sql.functions as F
F.approx_count_distinct?
df.agg({'station':'approx_count_distinct','year':'min'}).show()
```

```
+---------+----------------------------+
|min(year)|approx_count_distinct(station)|
+---------+----------------------------+
|   1893.0|                         213|
+---------+----------------------------+
```

Approximate Quantile

The method .approxQuantile computes the approximate quantiles.

Recall that this is how we computed the pivots for the distributed bucket sort.

In [31]:

```
print 'with accuracy 0.1: ',df.approxQuantile('year', [0.1*i for i in range(1,10)], 0.1)
print 'with accuracy 0.01: ',df.approxQuantile('year', [0.1*i for i in range(1,10)], 0.01)
```

```
with accuracy 0.1:  [1893.0, 1951.0, 1951.0, 1962.0, 1971.0, 1987.0, 1995.0, 1995.0, 20
12.0]
with accuracy 0.01:  [1929.0, 1946.0, 1956.0, 1965.0, 1974.0, 1984.0, 1993.0, 2000.0, 2
007.0]
```

Lets collect the exact number of rows for each year

This will take much longer than ApproxQuantile on a large file

In [32]:

```
# Lets collect the exact number of rows for each year ()
query='SELECT year,COUNT(year) AS count FROM weather GROUP BY year ORDER BY year'
print query
counts=sqlContext.sql(query)
A=counts.toPandas()
A.head()
```

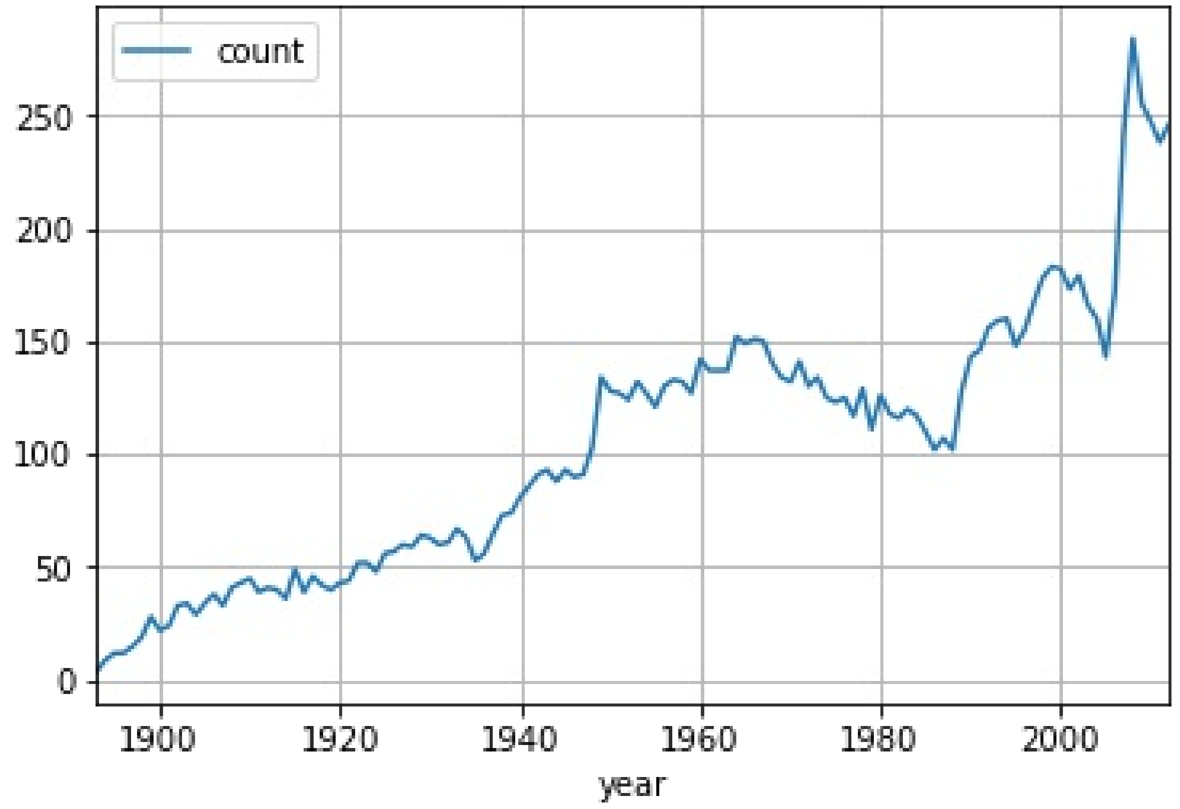
```
SELECT year,COUNT(year) AS count FROM weather GROUP BY year ORDER BY yea
r
```

Out[32]:

	year	count
0	1893.0	4
1	1894.0	9
2	1895.0	12
3	1896.0	12
4	1897.0	15

In [33]:

```
import pandas as pd
A.plot.line('year','count')
grid()
```



Reading rows selectively

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 [34]:

```
query='SELECT station,measurement,year FROM weather WHERE measurement="SNOW"'
print query
df2 = sqlContext.sql(query)
print df2.count(),df2.columns
df2.show(5)
```

```
SELECT station,measurement,year FROM weather WHERE measurement="SNOW"
2178 ['station', 'measurement', 'year']
+-----------+-----------+------+
|    station|measurement|  year|
+-----------+-----------+------+
|USC00111458|       SNOW|1991.0|
|USC00111458|       SNOW|1994.0|
|USC00111458|       SNOW|1995.0|
|USC00111458|       SNOW|1996.0|
|USC00111458|       SNOW|1997.0|
+-----------+-----------+------+
only showing top 5 rows
```