Small Data

CMPT 732, Fall 2017

Lots of data sets aren't big. In fact, most aren't.

Modern phones have 4 GB of memory: if you have less than that, it must be “small”. Why use Spark for everything?

Even running locally, Spark has a ≈10 s startup time: any work that takes less than that makes **no** sense in Spark.

Good reasons to use Spark: [editorial content]

- You actually have big data.
- You think your data might be big in the future, and need to be ready.
- You have “medium” data and the startup time pays off when running locally on multiple cores.

Spark for ETL

A very good use-case for Spark: ETL work that makes big data small.

Use Spark to extract/aggregate the data you really want to work with. Realize that made your data “small”. Move to some small data tools…

Python Data Tools

Python is one of the most common choices for data science work. (The other is R.)

As a result, there are many very mature data manipulation tools in Python. You should know they exist.

NumPy

Python's built-in data structures are not very memory-efficient: Python object overhead, references cause bad memory locality, etc.

Data you have will often have fixed types and sizes: exactly what C-style arrays are good at. NumPy provides efficient, typed arrays for Python.

```
import numpy as np
a = np.zeros((10000, 5), dtype=np.float32)
print(a)
print((a + 6).sum())
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 ...,
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
300000.0
```

NumPy can do lots of manipulation on arrays (at C-implemention speeds). e.g.

- basic arithmetic
- datetime manipulation
- matrix/linear algebra operations
- sorting, searching

Pandas

[Pandas](#) provides a DataFrame class for in-memory data manipulation. Pandas DataFrame ≠ Spark DataFrame, but concepts are similar.

```
import pandas as pd
cities = pd.read_csv('cities.csv')
print(cities)
```

```
        city  population     area
0  Vancouver     2463431  2878.52
1    Calgary     1392609  5110.21
2    Toronto     5928040  5905.71
3   Montreal     4098927  4604.26
4    Halifax      403390  5496.31
```

Similar operate-on-whole-DataFrame API. Slightly different operations. Not lazily evaluated.

```
cities['area_m2'] = cities['area'] * 1e6
print(cities)
```

```
        city  population     area       area_m2
0  Vancouver     2463431  2878.52  2.878520e+09
1    Calgary     1392609  5110.21  5.110210e+09
2    Toronto     5928040  5905.71  5.905710e+09
3   Montreal     4098927  4604.26  4.604260e+09
4    Halifax      403390  5496.31  5.496310e+09
```

Pandas Series (==columns) are stored as NumPy arrays, so you can use NumPy functions if you need to.

```
print(type(cities['population'].values))
print(cities['population'].values.dtype)
```

```
<class 'numpy.ndarray'>
int64
```

A Pandas DataFrame can be converted to a Spark DataFrame:

```
cities_pd = pd.read_csv('cities.csv')
cities_spark = spark.createDataFrame(cities_pd)
cities_spark.show()
```

```
+---------+----------+-------+
|     city|population|   area|
+---------+----------+-------+
|Vancouver|   2463431|2878.52|
|  Calgary|   1392609|5110.21|
|  Toronto|   5928040|5905.71|
| Montreal|   4098927|4604.26|
|  Halifax|    403390|5496.31|
+---------+----------+-------+
```

... and a Spark DataFrame to Pandas **if** it will fit in memory in the driver:

```
cities_pd2 = cities_spark.toPandas()
print(cities_pd2)
```

```
        city  population     area
0  Vancouver     2463431  2878.52
1    Calgary     1392609  5110.21
2    Toronto     5928040  5905.71
3   Montreal     4098927  4604.26
4    Halifax      403390  5496.31
```

This is faster in Spark ≥2.3 if you use [the Apache Arrow option](#).

With NumPy and Pandas, you can do a lot of basic data manipulation operations.

They will likely be faster on small (and medium?) data: no overhead of managing executors or distributing data, but single-threaded.

SciPy

The [SciPy](#) libraries include many useful tools to analyze data. Some examples:

- NumPy and Pandas
- Fourier Transforms (`scipy.fftpack`)
- Signal Processing (`scipy.signal`)
- Linear Algebra (`scipy.linalg`)
- Statistics (`scipy.stats`)
- Image processing (`scipy.ndimage`)
- Plots (`matplotlib`)

If those aren't enough, there are [SciKits](#) containing much more. e.g.

- Image processing (scikit-image)
- Video processing (scikit-video)
- Bioinformatics (scikit-bio)

SciKit-Learn

[Scikit-learn](#) is probably going to be useful to you some time: implementations of many machine learning algorithms for Python (and NumPy-shaped data).

Compared to `pyspark.ml`: older and more battle-tested; [includes algorithms](#) that don't distribute well; doesn't do distributed computation.

Python Libraries

Maybe the biggest pro-Python argument: it's used for data science and many other things, so libraries you need are implemented in Python.

[PyPI](#) is the package repository for Python. You can install packages with the `pip` command.

```
pip3 install --user scikit-learn scipy pandas
```