# CONTENTS

# BIG DATA ANALYTICS USING SPARK

Yoav Freund, Professor Computer Science & Engineering UC San Diego

LINK

# 1 WELCOME

Welcome to Big Data Analytics using Spark! This course teaches you how to perform statistical analysis of very large datasets that do not fit on a single computer. You will learn some of the most popular tools for performing this type of analysis: apache spark, XGBoost and TensorFlow. You will learn how to use these tools through Jupyter Notebooks and experience the power of combining narrative, code and graphics to create convincing analytical documents.

- Memory Hierarchy, Latency vs. Throughput
- Spark Basics
- Dataframes and SQL
- PCA and Weather Analysis
- K-Means and Intrinsic Dimensions
- Neural Networks and TensorFlow.

## SOFTWARE INSTALLATION DIRECTIONS

### Installation

- This class will use python3
- The easiest way to install everything is to use the Docker instance that we provide. If for whatever reason, you cannot install Docker then we also provide instructions to download and install the required Python implementation and associated libaries.
- If you're a verified learner, then it's up to you to decide if you want to have a local development environment. We will provide you with a cloud hosted jupyter notebook interface for some of the Programming Assignments where you can code directly. We still recommend verified learners to setup the local environment anyhow.
- If you're auditing the course, then you should follow the instructions below. We highly recommend setting up using docker.

### Setup Using Docker

The purpose of this part is to ensure you have a working and compatible Python and PySpark installation. In order to avoid potential compatibility issues generated from students using different versions than the expected, we provide a Docker image with Ubuntu 16.04 and a clean Anaconda 4.3 with python 3.6, jupyter 5.4, spark 2.2 installation. We also provide a script to run the docker image and get Jupyter running on it so that you can program on it directly. In this guide, we provide instructions on how to install Docker and pull the Docker image. In case you are not able to use Docker, you will have to install Python and Pyspark manually.

- Using the provided Docker container requires installing Docker, 6-7 GB of free space and root access on the host machine (admin rights for windows).
- If you are not able to use the provided container, you can install Python and Pyspark on your own. Make sure you follow the instructions given below. We will expect your results to match ours.
- Docker containers are not intended to store data. We highly recommend you develop your solutions locally and only use docker to compile and run. The following guides show you how to do that. Obtained results should be stored locally as well. If you develop within the container you are at risk of losing your work. You have been warned.
- When you work within the provided container (interactively or not) you are automatically logged in as a user named ucsddse230. Your homework notebook is mounted in the directory /home/ucsddse230/work. If you delete the mounted directory containing your work it will be deleted from the host system. Make sure your work is secure at all times. We recommend you use some sort of version control such as `git`.

### Installing Docker

Installing Docker should be straightforward for Windows and Mac OS users. Mac and Windows users can download it from from the Docker website. Linux users will have to use this guide. The linux guides essentially try to upgrade your system to a compatible version (for example upgrading to Ubuntu 16.04). Be careful not to break your current system. If you are working with linux, having a Ubuntu 16.04 system should result in an easier docker installation. For Windows users, Docker will require you enable Hyper-V and restart your computer. Some Windows 10 versions do not have Hyper-V. If you face any issues with installing Docker on Windows, installing Docker Toolbox instead of Docker should be the easiest way out.

### Pulling the Docker image

After successful installation of Docker, open a command prompt or shell and execute the following command (Windows users should skip the "sudo" part): Linux/Mac:

```
$ sudo docker pull ucsddse230/cse255-dse230
```

Windows (Powershell prefered):

```
$ docker pull ucsddse230/cse255-dse230
```

Docker should automatically start downloading and extracting the provided image. If you skip this step the image will automatically be downloaded the first time you attempt to start it. Once finished you can verify you have it by typing Linux/Mac:

```
$ sudo docker images
```

Windows:

```
$ docker images
```

Students using Docker Toolbox for their Windows OS that does not support Hyper-V would now need to execute an additional command to identify their Docker IP address.

```
$ docker-machine ip
```

Note down the IP address that is returned as the output. You will need to use this in the next section.

### Running Docker Images

Next, download the required content (for example, Programming assignment or Section Notebooks) files from EdX directory to some location on your computer. ex: /local/path/to/pa1. An an illustration, let's use Programming Assignment 1. You may open the Programming Assignment 1 section on EdX which has the link to the necessary started code. Let's say after unpacking the files pa1 files are present in `/local/path/to/pa1`

NOTE: This path should be the absolute path.

Then run the following following line of code in your terminal (first time might take a while).

```
$ docker run -it -p 8889:8888 -v /local/path/to/pa1:/home/ucsddse230/work ucsddse230/cse255-dse230 /bin/bash
```

This command will:

1. Start the docker container

2. mount the local directory "`/local/path/to/pa1`" inside the container at the location "`/home/ucsddse230/work`"

3. Forward requests to port 8889 on the local system from port 8888 inside the docker container.

Notice the terminal has changed, you are now inside a virtual machine. Run the following commands to start jupyter at http://localhost:8889 by issuing the command

```
$ jupyter notebook
```

This will start jupyter at port 8888 inside the docker container, which will be accessible outside the docker at port 8889.

Now you can view notebooks and work on homework at the localhost:8889 port. Go ahead open your web browser and put "localhost:8889" in the address bar. You should now be able to see the Jupyter Notebook webpage.

Students using Docker Toolbox can access the Jupyter notebook running in their Docker container at the DockerIP:8888 port, where DockerIP is the IP address returned in the previous section.

Whatever changes you make will also happen to /local/path/to/pa1.

### Setup From Scratch

First install the Python 3.6 version using the anaconda distribution.

#### *Install jupyter*

If you install Anaconda, jupyter and almost all the necessary packages are installed for you.

#### *Install notebook extensions*

This step is not required, but extensions can make your work on notebooks significantly easier. To install a bunch of useful extensions, together with a configurator for managing thses extensions, follow the directions on:

https://github.com/Jupyter-contrib/jupyter_nbextensions_configurator

#### *Install python packages*

Make sure to install the python package findspark. The typing the following command in the terminal installs the package:

Anaconda: `conda install -c conda-forge findspark=1.0.0`[1]

pip: `sudo pip install findspark`

If you are using pip instead of anaconda, you also must install the following packages:

- numpy
- matplotlib
- pandas

Some notebooks require additional packages, or packages of a later version. If an `import` command in a notebook fails, use `pip` or `conda` to install the missing package. Install Spark on your computer

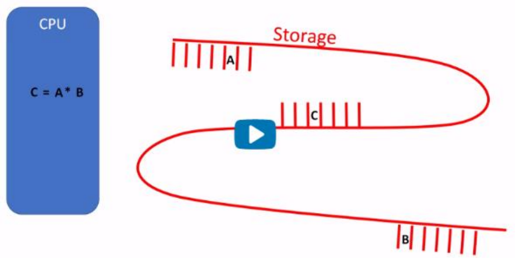
- Install on Linux or Mac OS X
- Install on Windows

[1] Find pyspark to make it importable.

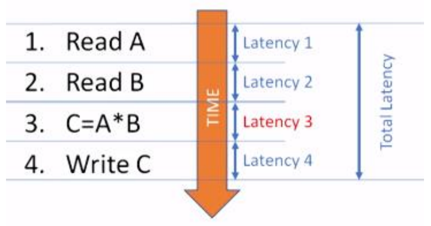
## 2 Map-Reduce and Spark

### 2.1 The Memory Hierarchy

#### 2.1.1 Memory Latency



##### Latencies



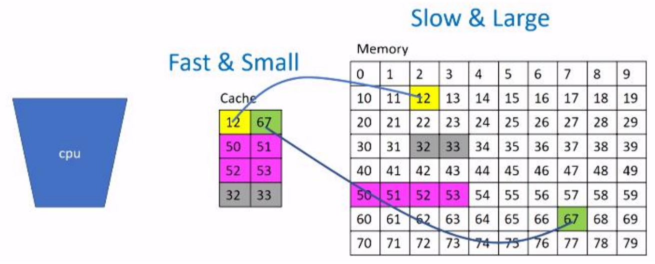
1. Read A
2. Read B
3. C=A*B
4. Write C

With big data, most of the latency is memory latency (1,2,4), not computation (3)

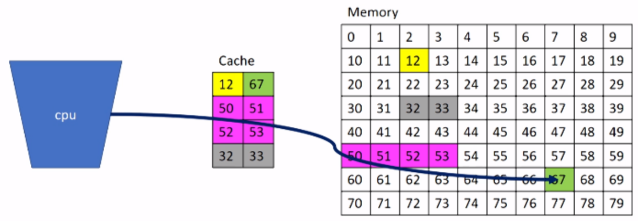
- The major source of latency in data analysis is reading and writing to storage
- Different types of storage offer different latency, capacity and price.
- Big data analytics revolves around methods for organizing storage and computation in ways that maximize speed while minimizing cost.
- Next, memory locality.

#### 2.1.2 Cache

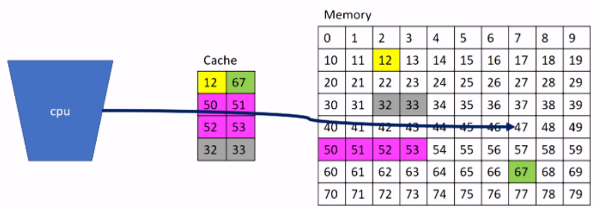
##### Cache: The basic idea



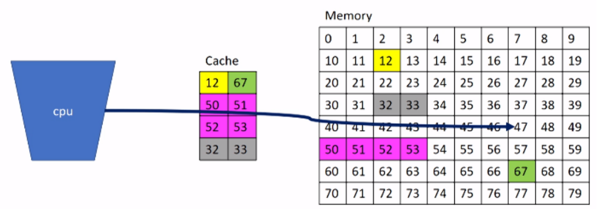
##### Cache Hit



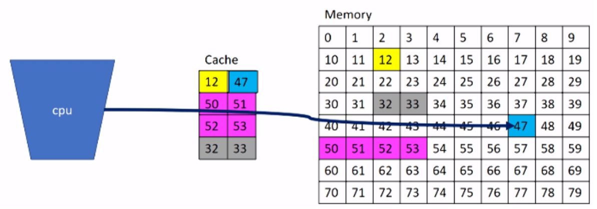
##### Cache Miss



##### Cache Miss Service: 1) Choose byte to drop



##### Cache Miss Service: 3) Read In



##### Access Locality

- The cache is effective If most accesses are hits.
  - Cache Hit Rate is high.
- **Temporal Locality**: Multiple accesses to **same** address within a short time period

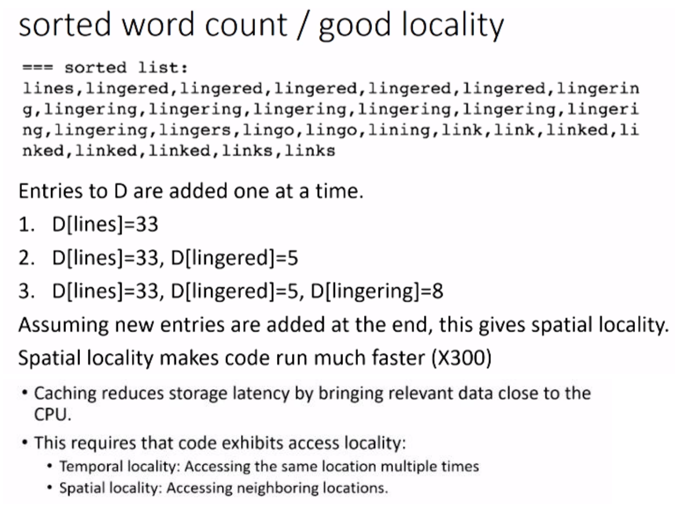
##### Spatial locality

- **Spatial Locality**: Multiple accesses to close-together addresses in short time period.
  - The difference between two sums.
  - Counting words by sorting
- Benefiting from spatial locality
  - Memory is partitioned into **Blocks/Lines** rather than single bytes.
  - Moving a block of memory takes much less time than moving each byte individually.
  - Memory locations that are close to each other are likely to fall in the same block.
  - Resulting in more cache hits.

##### Unsorted word count / poor locality

```
=== unsorted list:
the,vernacular,but,as,for,you,ye,carrion,rogues,turning,to,
```

- Consider the memory access to the dictionary D:
- Count without sort:
  D[the]=12332,…,D[but]=943,………,D[vernacular]=10,…………..,D[for]=..
- Temporal locality for very common words like “the”
- No spatial locality



## sorted word count / good locality

```
=== sorted list:
lines,lingered,lingered,lingered,lingered,lingered,lingerin
g,lingering,lingering,lingering,lingering,lingering,lingeri
ng,lingering,lingers,lingo,lingo,lining,link,link,linked,li
nked,linked,linked,links,links
```

Entries to D are added one at a time.

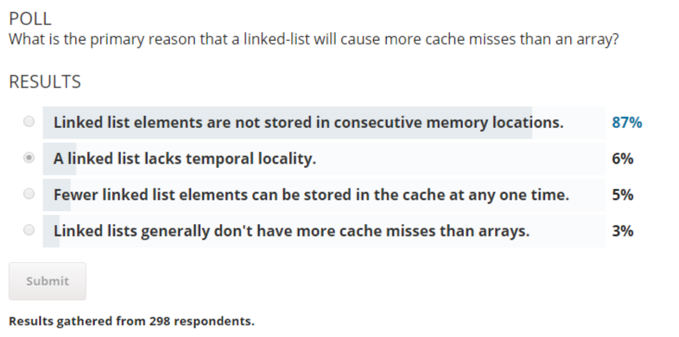
1. D[lines]=33
2. D[lines]=33, D[lingered]=5
3. D[lines]=33, D[lingered]=5, D[lingering]=8

Assuming new entries are added at the end, this gives spatial locality.

Spatial locality makes code run much faster (X300)

- Caching reduces storage latency by bringing relevant data close to the CPU.
- This requires that code exhibits access locality:
  - Temporal locality: Accessing the same location multiple times
  - Spatial locality: Accessing neighboring locations.

## 2.1.3 Memory Access locality



POLL

What is the primary reason that a linked-list will cause more cache misses than an array?

RESULTS

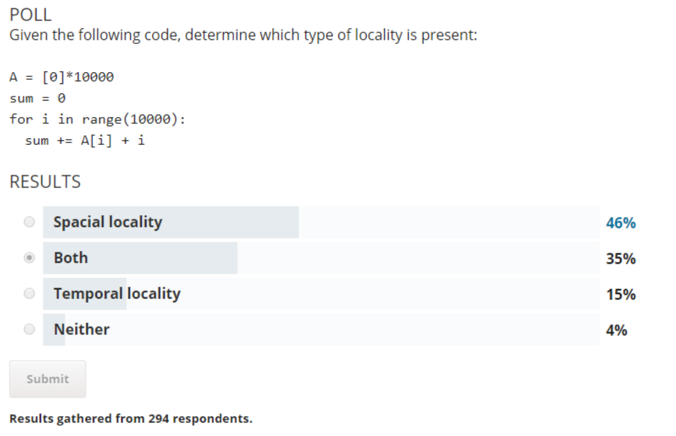
| | |
|---|---|
| Linked list elements are not stored in consecutive memory locations. | 87% |
| [✓ A linked list lacks temporal locality.] | 6% |
| Fewer linked list elements can be stored in the cache at any one time. | 5% |
| Linked lists generally don't have more cache misses than arrays. | 3% |

Submit

Results gathered from 298 respondents.

FEEDBACK

Linked list elements are not stored in consecutive memory locations.



POLL

Given the following code, determine which type of locality is present:

```
A = [0]*10000
sum = 0
for i in range(10000):
  sum += A[i] + i
```

RESULTS

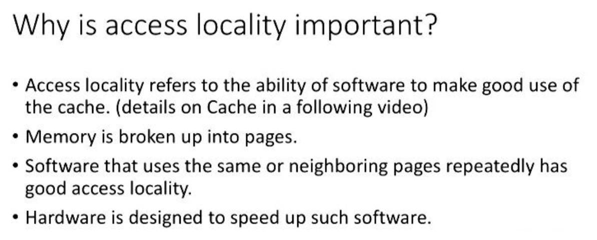
| | |
|---|---|
| Spacial locality | 46% |
| [✓ Both] | 35% |
| Temporal locality | 15% |
| Neither | 4% |

Submit

Results gathered from 294 respondents.

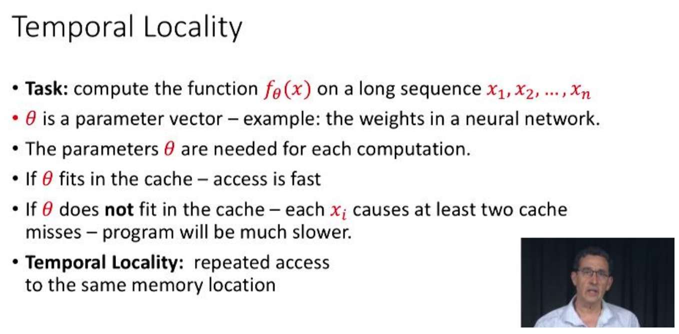
FEEDBACK

Spacial locality



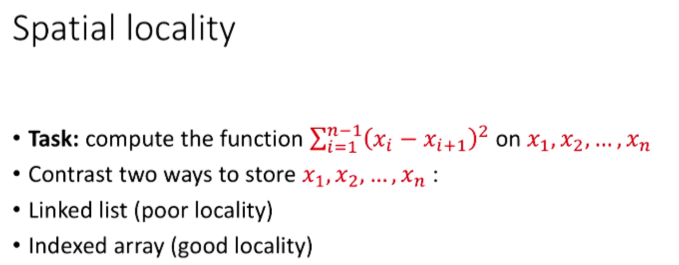
## Why is access locality important?

- Access locality refers to the ability of software to make good use of the cache. (details on Cache in a following video)
- Memory is broken up into pages.
- Software that uses the same or neighboring pages repeatedly has good access locality.
- Hardware is designed to speed up such software.



## Temporal Locality

- **Task:** compute the function $f_\theta(x)$ on a long sequence $x_1, x_2, \ldots, x_n$
- $\theta$ is a parameter vector – example: the weights in a neural network.
- The parameters $\theta$ are needed for each computation.
- If $\theta$ fits in the cache – access is fast
- If $\theta$ does **not** fit in the cache – each $x_i$ causes at least two cache misses – program will be much slower.
- **Temporal Locality:** repeated access to the same memory location

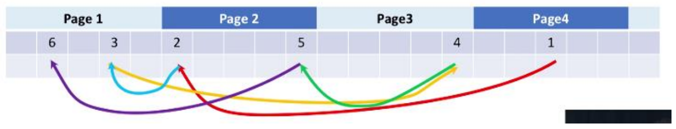


## Spatial locality

- **Task:** compute the function $\sum_{i=1}^{n-1}(x_i - x_{i+1})^2$ on $x_1, x_2, \ldots, x_n$
- Contrast two ways to store $x_1, x_2, \ldots, x_n$ :
- Linked list (poor locality)
- Indexed array (good locality)

## Linked List

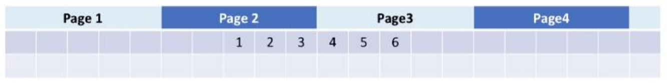
Let $x_1, x_2, \ldots, x_n$ be 1,2,3,4,5,6



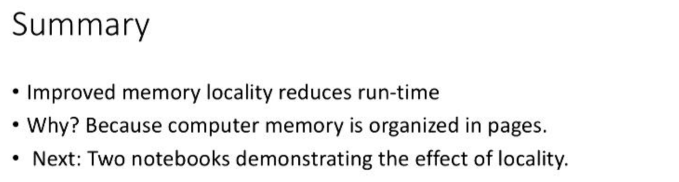
Traversal of 6 elements touches 4 pages

## array

Let $x_1, x_2, \ldots, x_n$ be 1,2,3,4,5,6



Traversal of 6 elements touches 2 pages



## Summary

- Improved memory locality reduces run-time
- Why? Because computer memory is organized in pages.
- Next: Two notebooks demonstrating the effect of locality.

## 2.1.4 Row-wise vs Column-wise Scanning

### *Memory locality, Rows vs. Columns*

Effect of row vs column major layout: The way you traverse a 2D array effects speed.

- numpy arrays are, by default, organized in a row-major order.

```
a=array([range(1,31)]).reshape([3,10])
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |

- `a[i,j]` and `a[i,j+1]` are placed in consecutive places in memory.
- `a[i,j]` and `a[i+1,j]` are 10 memory locations apart.
- This implies that scanning the array row by row is more local than scanning column by column.
- locality implies speed.

```
In [1]:
%pylab inline
```

```
from time import time

# create an n by n array
n=1000
a=ones([n,n])
Populating the interactive namespace from numpy and matplotlib
In [2]:
%%time
# Scan column by column
s=0;
for i in range(n): s+=sum(a[:,i])
CPU times: user 12.7 ms, sys: 1.66 ms, total: 14.3 ms
Wall time: 12.8 ms
In [3]:
%%time
## Scan row by row
s=0;
for i in range(n): s+=sum(a[i,:])
CPU times: user 7.53 ms, sys: 4 ms, total: 11.5 ms
Wall time: 7.89 ms
```

## Some experiments with row vs column scanning

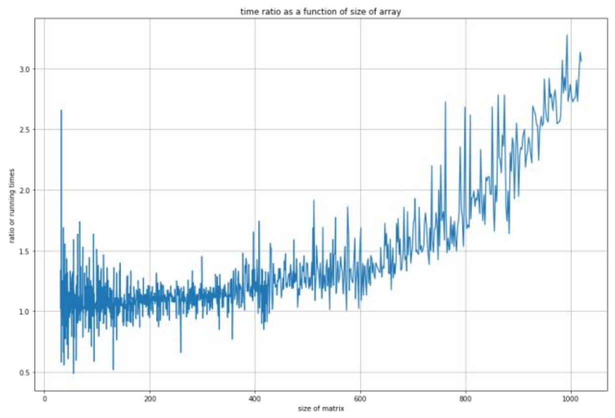
We want to see how the run time of these two code snippets varies as `n`, the size of the array, is changed.

```
In [4]:
def sample_run_times(T,k=10):
    """ compare time to sum array row by row vs column by column
        T: the sizes of the matrix, [10**e for e in T]
        k: the number of repetitions of each experiment
    """
    all_times=[]
    for e in T:
        n=int(10**e)
        #print('\r',n)
        a=np.ones([n,n])
        times=[]

        for i in range(k):
            t0=time()
            s=0;
            for i in range(n):
                s+=sum(a[:,i])
            t1=time()
            s=0;
            for i in range(n):
                s+=sum(a[i,:])
            t2=time()
            times.append({'row minor':t1-t0,'row major':t2-t1})
        all_times.append({'n':n,'times':times})
    return all_times
In [5]:
#example run
sample_run_times([1,2],k=1)
Out[5]:
[{'n': 10,
  'times': [{'row major': 3.1948089599609375e-05,
    'row minor': 5.698204040527344e-05}]},
 {'n': 100,
  'times': [{'row major': 0.0005230903625488281,
    'row minor': 0.0003888607025146484}]}]
```

## Plot the ratio between run times as function of n

Here we have small steps between consecutive values of `n` and only one measurement for each (`k=1`)

```
In [6]:
all_times=sample_run_times(np.arange(1.5,3.01,0.001),k=1)

n_list=[a['n'] for a in all_times]
ratios=[a['times'][0]['row minor']/a['times'][0]['row major'] for
a in all_times]

figure(figsize=(15,10))
plot(n_list,ratios)
grid()
xlabel('size of matrix')
ylabel('ratio or running times')
title('time ratio as a function of size of array');
```



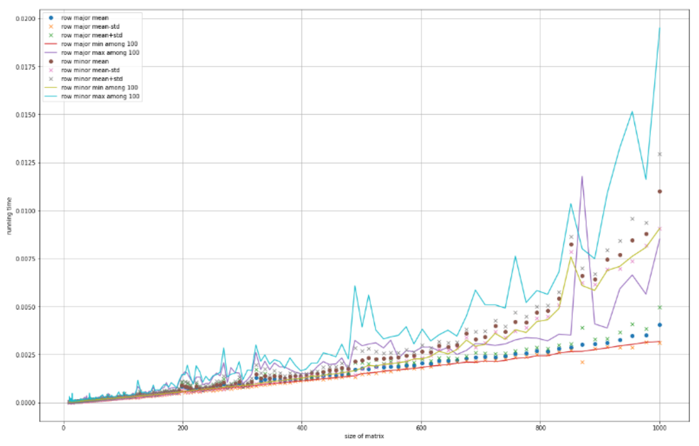
## Conclusions

- Traversing a numpy array column by column takes more than row by row.
- The effect increasese proportionally to the number of elements in the array (square of the number of rows or columns).
- Run time has large fluctuations.
- See you next time.

## Next, we want to quantify the random fluctuations

and see what is their source

```
In [7]:
k=100
all_times=sample_run_times(np.arange(1,3.001,0.01),k=k)
_n=[]
_row_major_mean=[]
_row_major_std=[]
_row_major_std=[]
_row_minor_mean=[]
_row_minor_std=[]
_row_minor_min=[]
_row_minor_max=[]
_row_major_min=[]
_row_major_max=[]

for times in all_times:
    _n.append(times['n'])
    row_major=[a['row major'] for a in times['times']]
    row_minor=[a['row minor'] for a in times['times']]
    _row_major_mean.append(np.mean(row_major))
    _row_major_std.append(np.std(row_major))
    _row_major_min.append(np.min(row_major))
    _row_major_max.append(np.max(row_major))

    _row_minor_mean.append(np.mean(row_minor))
    _row_minor_std.append(np.std(row_minor))
    _row_minor_min.append(np.min(row_minor))
    _row_minor_max.append(np.max(row_minor))

_row_major_mean=np.array(_row_major_mean)
_row_major_std=np.array(_row_major_std)
_row_minor_mean=np.array(_row_minor_mean)
_row_minor_std=np.array(_row_minor_std)
In [8]:
figure(figsize=(20,13))
plot(_n,_row_major_mean,'o',label='row major mean')
plot(_n,_row_major_mean-_row_major_std,'x',label='row major mean-
std')
plot(_n,_row_major_mean+_row_major_std,'x',label='row major
mean+std')
plot(_n,_row_major_min,label='row major min among %d'%k)
plot(_n,_row_major_max,label='row major max among %d'%k)
plot(_n,_row_minor_mean,'o',label='row minor mean')
plot(_n,_row_minor_mean-_row_minor_std,'x',label='row minor mean-
std')
plot(_n,_row_minor_mean+_row_minor_std,'x',label='row minor
mean+std')
plot(_n,_row_minor_min,label='row minor min among %d'%k)
plot(_n,_row_minor_max,label='row minor max among %d'%k)
xlabel('size of matrix')
ylabel('running time')
legend()
grid()
```



***Summary***

1. Scan by column slower than scan by row and the difference increases with the size.
2. Scan by row increases linearly and has very little random fluctuations.
3. Scan by column increases linearly with one constant until about `n=430` and then increases with a higher constant.
4. Scan by column has large fluctuations around the mean

## 2.1.5 Measuring Latency

### 1 measuring memory latency

In this notebook we will investigate the distribution of latency times for different size arrays.

1. **Goal 1:** Measure the effects of caching **in the wild**
2. **Goal 2:** Undestand how to study long-tail distributions.

[redacted]

### 1.2 defining memory latency

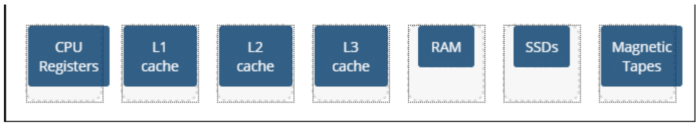
Latency is the time difference between the time at which the CPU is issuing a read or write command and, the time the command is complete.

- This time is very short if the element is already in the L1 Cache,
- and is very long if the element is in external memory (disk or SSD).

### 1.3 setting parameters

- We test access to elements arrays whose sizes are:
  - `m_legend=['Zero','1KB','1MB','1GB','10GB']`
- Arrays are stored **in memory** or **on disk**
- We perform 100,000 read/write ops to random locations in the array.
- We analyze the **distribution** of the latencies as a function of the size of the array.

## 2.1.6 Memory Hierarchy

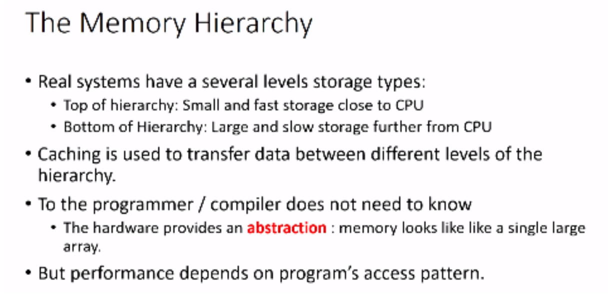


Why do we use caching?

- to transfer data between the different levels of the memory hierarchy 62%
- to ensure we stay in the top of the memory hierarchy 35%
- to ensure the bottom of the memory hierarchy does not lose data 2%

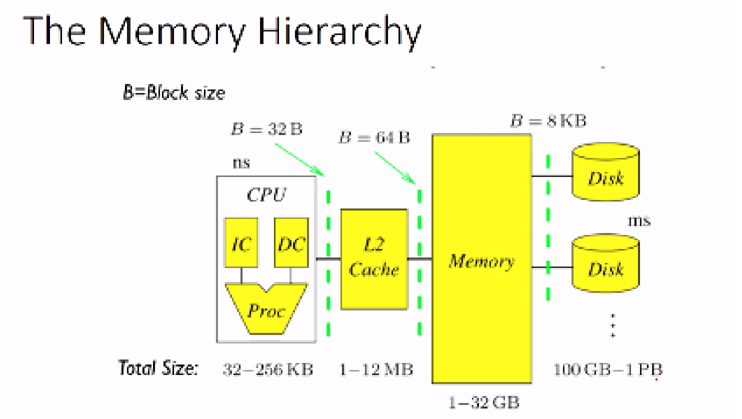
How is memory abstracted to the programmer?

- as a large, single array 72%
- memory is not abstracted to the programmer 15%
- as a high-dimensional matrix 13%

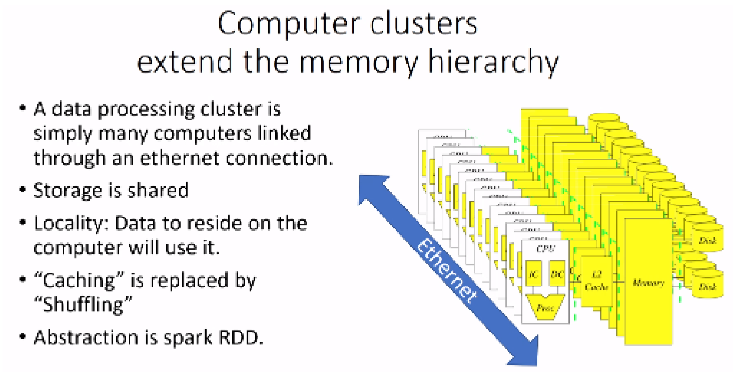


The Memory Hierarchy

- Real systems have a several levels storage types:
  - Top of hierarchy: Small and fast storage close to CPU
  - Bottom of Hierarchy: Large and slow storage further from CPU
- Caching is used to transfer data between different levels of the hierarchy.
- To the programmer / compiler does not need to know
  - The hardware provides an **abstraction** : memory looks like like a single large array.
- But performance depends on program's access pattern.

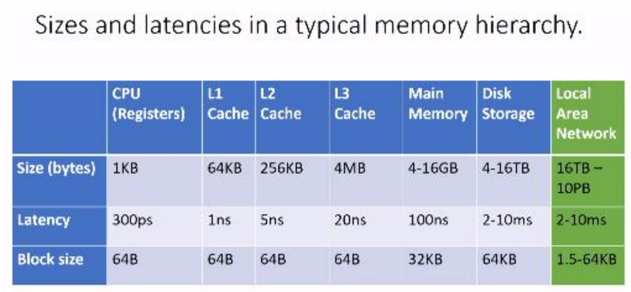


4 levels of storage: trade-off speed vs size



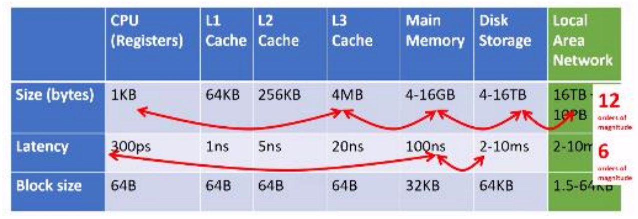
Computer clusters extend the memory hierarchy

- A data processing cluster is simply many computers linked through an ethernet connection.
- Storage is shared
- Locality: Data to reside on the computer will use it.
- "Caching" is replaced by "Shuffling"
- Abstraction is spark RDD.



Sizes and latencies in a typical memory hierarchy.

| | CPU (Registers) | L1 Cache | L2 Cache | L3 Cache | Main Memory | Disk Storage | Local Area Network |
|---|---|---|---|---|---|---|---|
| Size (bytes) | 1KB | 64KB | 256KB | 4MB | 4-16GB | 4-16TB | 16TB – 10PB |
| Latency | 300ps | 1ns | 5ns | 20ns | 100ns | 2-10ms | 2-10ms |
| Block size | 64B | 64B | 64B | 64B | 32KB | 64KB | 1.5-64KB |



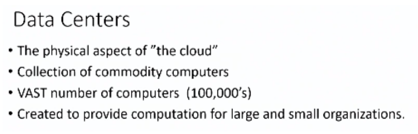
| | CPU (Registers) | L1 Cache | L2 Cache | L3 Cache | Main Memory | Disk Storage | Local Area Network |
|---|---|---|---|---|---|---|---|
| Size (bytes) | 1KB | 64KB | 256KB | 4MB | 4-16GB | 4-16TB | 16TB - 10PB |
| Latency | 300ps | 1ns | 5ns | 20ns | 100ns | 2-10ms | 2-10ms |
| Block size | 64B | 64B | 64B | 64B | 32KB | 64KB | 1.5-64KB |

12
orders of magnitude
6
orders of magnitude

## 2.1.7 History of Large Scale Computing

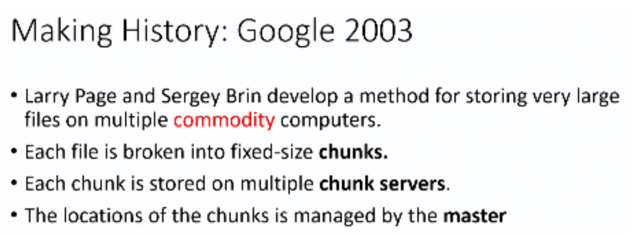
Super computers

- Cray, Deep Blue, Blue Gene ...
- Specialized hardware
- Extremely expensive
- created to solve specialized important problems



Data Centers

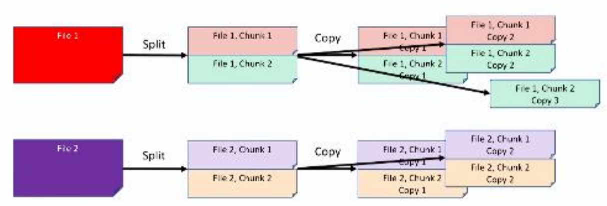
- The physical aspect of "the cloud"
- Collection of commodity computers
- VAST number of computers (100,000's)
- Created to provide computation for large and small organizations.



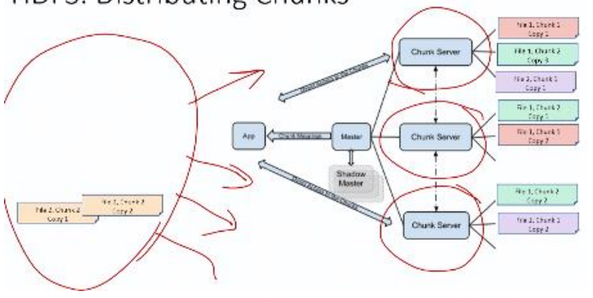
## Making History: Google 2003

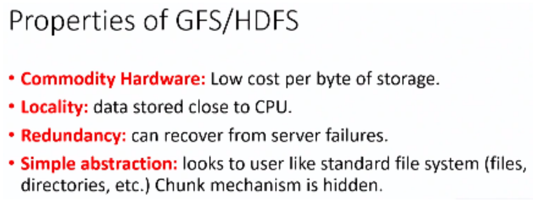
- Larry Page and Sergey Brin develop a method for storing very large files on multiple commodity computers.
- Each file is broken into fixed-size **chunks.**
- Each chunk is stored on multiple **chunk servers**.
- The locations of the chunks is managed by the **master**

## HDFS: Chunking files



## HDFS: Distributing Chunks

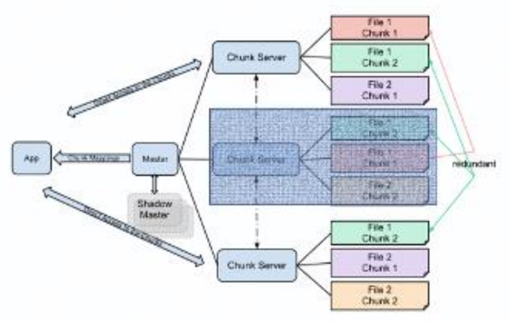




## Properties of GFS/HDFS

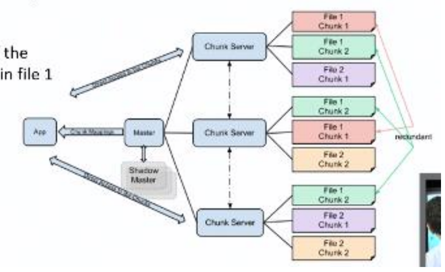
- **Commodity Hardware:** Low cost per byte of storage.
- **Locality:** data stored close to CPU.
- **Redundancy:** can recover from server failures.
- **Simple abstraction:** looks to user like standard file system (files, directories, etc.) Chunk mechanism is hidden.

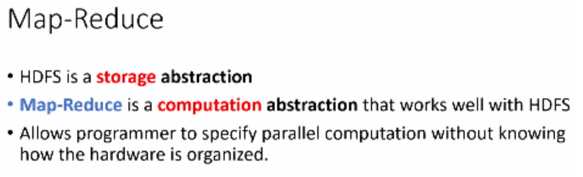
## Redundancy



## Locality

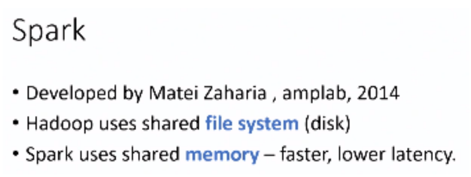
Task:
Sum all of the elements in file 1





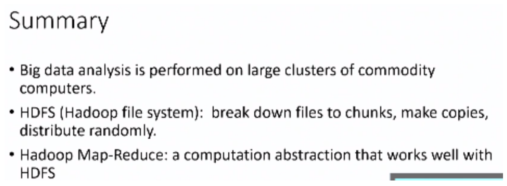
## Map-Reduce

- HDFS is a **storage abstraction**
- **Map-Reduce** is a **computation abstraction** that works well with HDFS
- Allows programmer to specify parallel computation without knowing how the hardware is organized.



## Spark

- Developed by Matei Zaharia , amplab, 2014
- Hadoop uses shared **file system** (disk)
- Spark uses shared **memory** – faster, lower latency.



## Summary

- Big data analysis is performed on large clusters of commodity computers.
- HDFS (Hadoop file system): break down files to chunks, make copies, distribute randomly.
- Hadoop Map-Reduce: a computation abstraction that works well with HDFS

You are given a cluster of 5 servers, each with 500GB of storage. You run HDFS on these 5 servers with a redundancy factor of 4. What is the effective amount of total storage space available on the HDFS cluster?

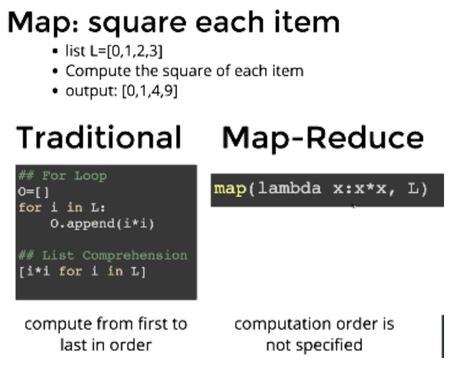
- 625GB 55%
- 2500GB 16%
- 125GB 15%
- 500GB 14%

Which of the following are NOT properties of HDFS?

- Provides computational abstraction 67%
- Locality of data to CPU 16%
- Abstracts filesystem from user 13%
- Redundancy 2%
- Files divided into chunks 2%

## 2.2 SPARK BASICS

### 2.2.1 Map Reduce



## Map: square each item

- list L=[0,1,2,3]
- Compute the square of each item
- output: [0,1,4,9]

**Traditional**

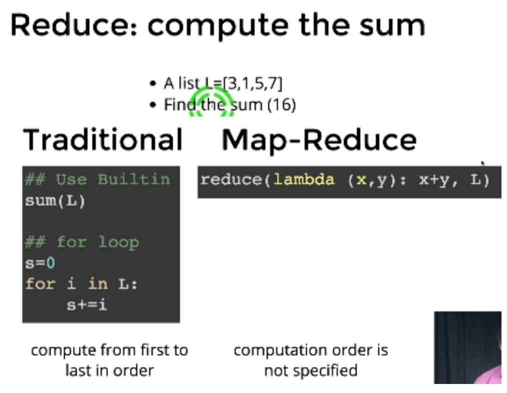
```
## For Loop
O=[]
for i in L:
    O.append(i*i)

## List Comprehension
[i*i for i in L]
```

compute from first to last in order

**Map-Reduce**

```
map(lambda x:x*x, L)
```

computation order is not specified



## Reduce: compute the sum

- A list L=[3,1,5,7]
- Find the sum (16)

**Traditional**

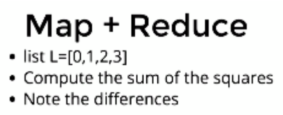
```
## Use Builtin
sum(L)

## for loop
s=0
for i in L:
    s+=i
```

compute from first to last in order

**Map-Reduce**

```
reduce(lambda (x,y): x+y, L)
```

computation order is not specified



## Map + Reduce

- list L=[0,1,2,3]
- Compute the sum of the squares
- Note the differences

## Traditional

```
## For Loop
s=0
for i in L:
    s+= i*i
## List comprehension
sum([i*i for i in L])
```

compute from first to last in order

Immediate execution

## Map-Reduce

```
reduce(lambda x,y:x+y, \\
      map(lambda i:i*i,L))
```

computation order is not specified

Execution **plan**

## The Wrong way

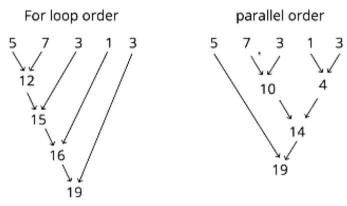
```
reduce(lambda x,y:x+y*y)
```

- Map, Reduce operations should not depend on:
  - Order of items in the list (commutativity)
  - Order of operations (Associativity)

### Order independence

- The result of map or reduce does not depend on the order

### computation order of a sum



## Why Order Independence?

- Computation order can be chosen by compiler/optimizer.
- Allows for **parallel computation** of sums of subsets.
  - Modern hardware calls for parallel computation but parallel computation is very hard to program.
- Using map-reduce programmer **exposes** to the compiler opportunities for parallel computation.

### 2.2.2 Short History of Map Reduce

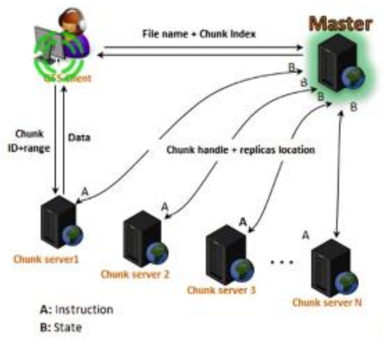
Which of the following is not a characteristic of the collect() function of an RDD?

- It returns a subset of the distributed data  66%
- Parallelism is lost after collect() is performed 15%
- It brings all the distributed data to a single node 10%
- It is too slow for large data 9%

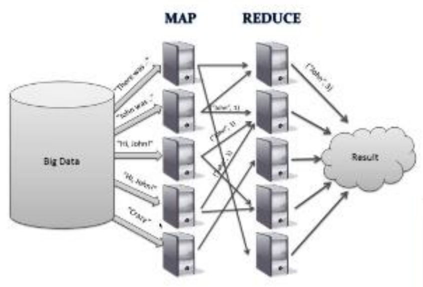
Which of the following operations on an RDD also returns an RDD? Check all that apply

- map()
- reduce()
- collect()
- sample()
- take()
- first()

## Google File System 2003



## Google MapReduce 2004



## Apache Hadoop, 2006 ---

- An open-source implementation of GFS+MapReduce
- File System: GFS -> HDFS
- Compute system: Google MapReduce -> Hadoop MapReduce
- Large eco-system: Apache Pig, Apache Hive, Apache HBase, Apache Phoenix, Apache Spark, Apache ZooKeeper, Cloudera Impala, Apache Flume, Apache Sqoop, Apache Oozie, Apache Storm.

## Apache Spark 2014

- Matei Zaharia, MPLab, Berkeley (Now in MIT)
- Main difference from Hadoop: distributed memory instead of distributed files.

## Spark, java, scala & python

- The native language of the Hadoop eco-system is Java
- Spark can be programmed in java, but code tends to be long.
- **Scala** allows the parallel programming to be abstracted. It is the core language for Spark.
  - The main problem is that it has a small user base.
  - You will want to learn scala if you want to extend spark.
- **PySpark** is a Python library for programming.
  - Does not always achieve the same efficiencies, bu easier to learn.
  - We will use pyspark

## Spark Architecture

SC and RDD

## Spark Context

- The pyspark program runs on the main node.
- Control of other nodes is achieved through a special object called the **SparkContext** (usually named **sc**).
- A notebook can have only one **SparkContext** object.
- Initialization: **sc=SparkContext()**, use parameters for non-default configuration.

### Resilient Distributed Dataset (RDD)

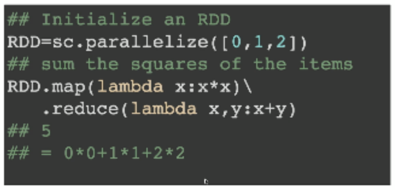
- A list whose elements are distributed over several computers.
- The main data structure in Spark.
- When in RDD form, the elements of the list can be manipulated only through RDD specific methods.
- RDDs can are created from a list on the master node or from a file.

RDDs can be translated back into a local list using the command **collect**.

## Pyspark

Some basic examples

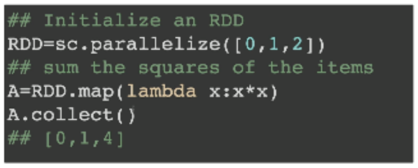
## Basic example



```
## Initialize an RDD
RDD=sc.parallelize([0,1,2])
## sum the squares of the items
RDD.map(lambda x:x*x)\
   .reduce(lambda x,y:x+y)
## 5
## = 0*0+1*1+2*2
```

Reduce generates a single item on the master node

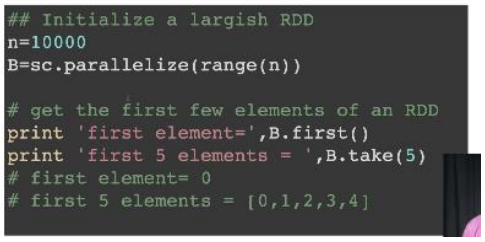
## RDD to RDD



```
## Initialize an RDD
RDD=sc.parallelize([0,1,2])
## sum the squares of the items
A=RDD.map(lambda x:x*x)
A.collect()
## [0,1,4]
```

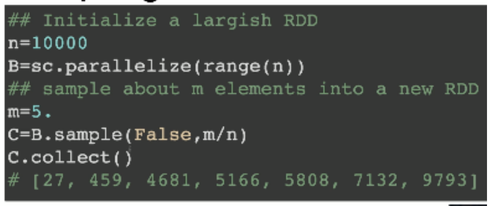
- **collect()** Collects all of the items in the RDD into a the master.
- If the RDD is large, this can take a long time.

## Checking the start of an RDD



```
## Initialize a largish RDD
n=10000
B=sc.parallelize(range(n))

# get the first few elements of an RDD
print 'first element=',B.first()
print 'first 5 elements = ',B.take(5)
# first element= 0
# first 5 elements = [0,1,2,3,4]
```

## Sampling an RDD

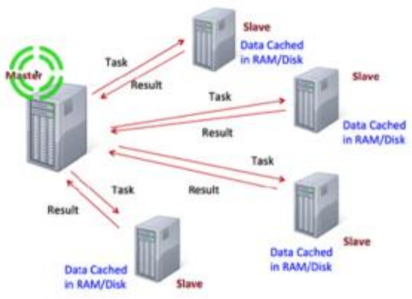


```
## Initialize a largish RDD
n=10000
B=sc.parallelize(range(n))
## sample about m elements into a new RDD
m=5.
C=B.sample(False,m/n)
C.collect()
# [27, 459, 4681, 5166, 5808, 7132, 9793]
```

- Each run results in a different sample.
- Sample size varies, expected size is 5.
- Result is an RDD, need to collect to list.
- Sampling very useful for machine learning.

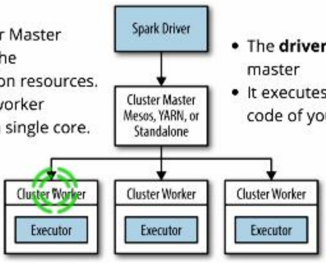
### 2.2.3 Spark Architecture

## Hardware organization



In local installation, cores serve as master & slaves

## spatial software organization

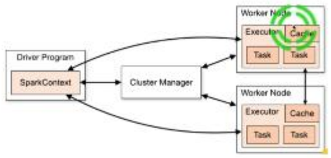


- The Cluster Master manages the computation resources.
- The each worker manages a single core.

- The **driver** runs on the master
- It executes the "main()" code of your program.

- Each RDD is **partitioned** among the workers,
- Workers manage **partitions** and **Executors**
- Executors execute **tasks** on their partition, are myopic.

## spatial organization (more detail)

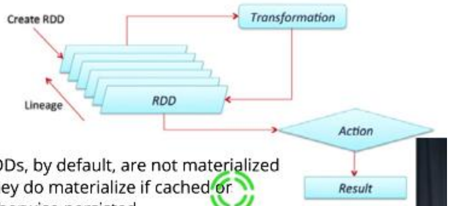


- SparkContext (sc) is the abstraction that encapsulates the cluster for the driver node (and the programmer).
- Worker nodes manage resources in a single slave machine.
- Worker nodes communicate with the cluster manager.
- Executors are the processes that can perform **tasks**.
- Cache refers to the local memory on the slave machine.

## materialization

- Consider RDD1
  -> Map (x: x*x) -> RDD2
  ->Reduce (x,y:x+y)-> float (in head node)
- RDD1 -> RDD2 is a lineage
- RDD2 can be consumed as it is being generated.
- Does not have to be **materialized** = stored in memory

## RDD Processing

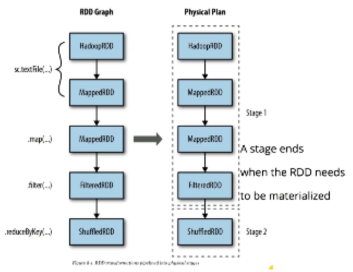


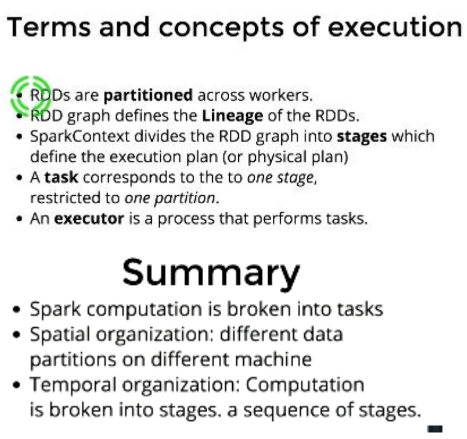
- RDDs, by default, are not materialized
- They do materialize if cached or otherwise persisted.

Temporal organization

RDD Graph and Physical plan

Recall Spatial organization




### Terms and concepts of execution

- RDDs are **partitioned** across workers.
- RDD graph defines the **Lineage** of the RDDs.
- SparkContext divides the RDD graph into **stages** which define the execution plan (or physical plan)
- A **task** corresponds to the to *one stage*, restricted to *one partition*.
- An **executor** is a process that performs tasks.

### Summary

- Spark computation is broken into tasks
- Spatial organization: different data partitions on different machine
- Temporal organization: Computation is broken into stages. a sequence of stages.

RDDs, by default, are stored in memory. (True/False)

- False 53%
- True 47%

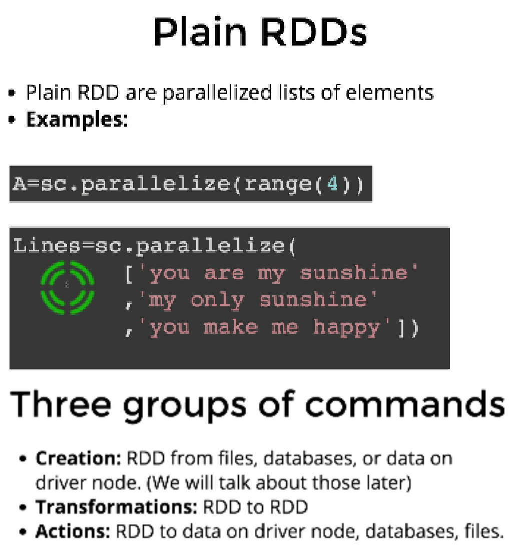
A stage can be defined as...

- A set of operations that can be done without materialization 83%
- A set of operations that require materialization 14%
- A set of operations that are carried out after materialization 2%

## 2.2.4 Manipulating Plain RDD

If rdd is a resilient distributed dataset, rdd.collect() returns a

- list 93%
- dataframe  6%
- dictionary  1%


### Plain RDDs

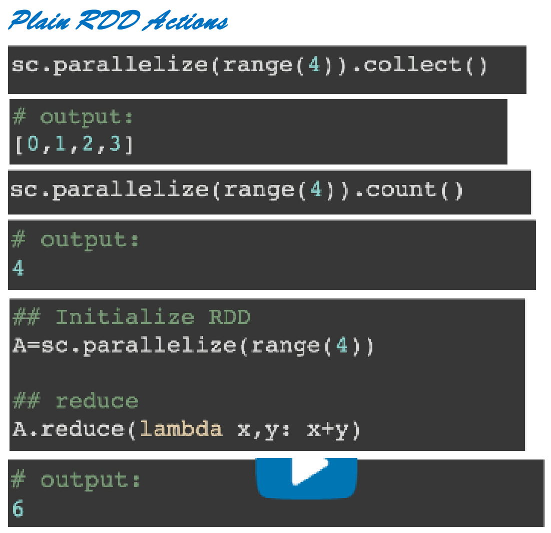
- Plain RDD are parallelized lists of elements
- **Examples:**

```
A=sc.parallelize(range(4))
```

```
Lines=sc.parallelize(
        ['you are my sunshine'
        ,'my only sunshine'
        ,'you make me happy'])
```

### Three groups of commands

- **Creation:** RDD from files, databases, or data on driver node. (We will talk about those later)
- **Transformations:** RDD to RDD
- **Actions:** RDD to data on driver node, databases, files.


***Plain RDD Actions***

```
sc.parallelize(range(4)).collect()
```

```
# output:
[0,1,2,3]
```

```
sc.parallelize(range(4)).count()
```

```
# output:
4
```

```
## Initialize RDD
A=sc.parallelize(range(4))

## reduce
A.reduce(lambda x,y: x+y)
```

```
# output:
6
```

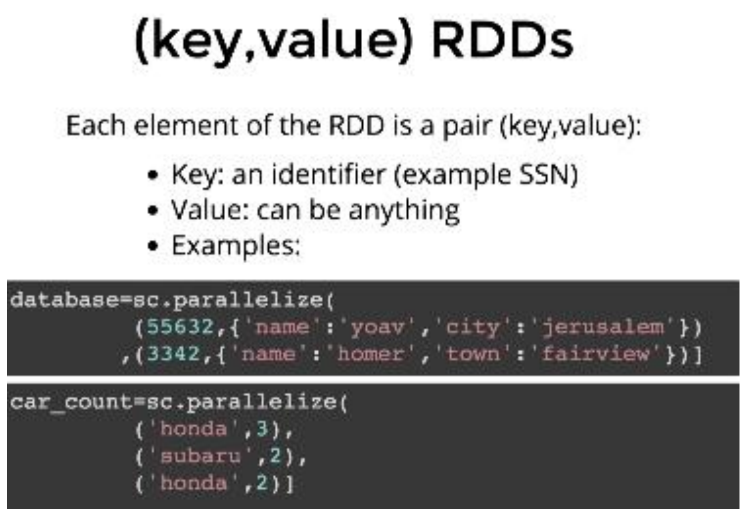
## 2.2.5 Manipulating KeyVal RDD

The key in an RDD has to be unique (True or False).

- False 88%
- True 12%

countByKey returns...

- a dictionary 70%
- A list of tuples 26%
- a list of lists 4%

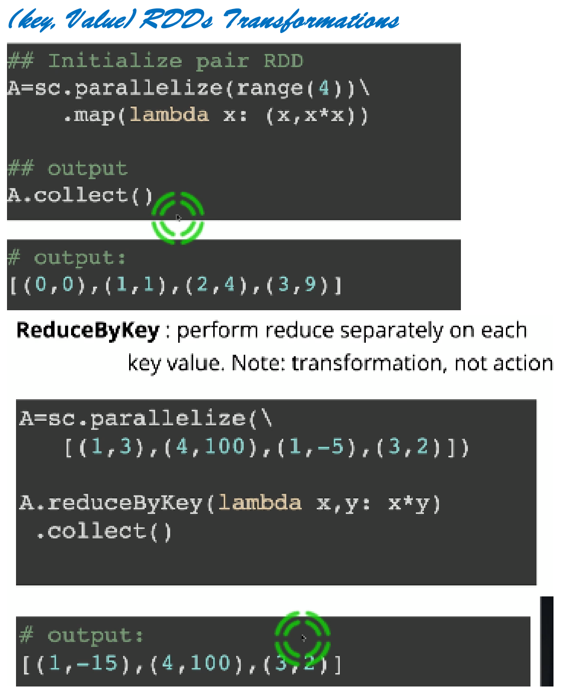

### (key,value) RDDs

Each element of the RDD is a pair (key,value):

- Key: an identifier (example SSN)
- Value: can be anything
- Examples:

```
database=sc.parallelize(
         (55632,{'name':'yoav','city':'jerusalem'})
        ,(3342,{'name':'homer','town':'fairview'})]
```

```
car_count=sc.parallelize(
         ('honda',3),
         ('subaru',2),
         ('honda',2)]
```


***(key, Value) RDDs Transformations***

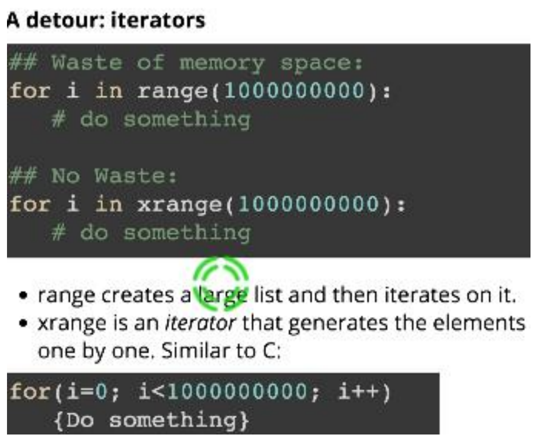
```
## Initialize pair RDD
A=sc.parallelize(range(4))\
    .map(lambda x: (x,x*x))

## output
A.collect()
```

```
# output:
[(0,0),(1,1),(2,4),(3,9)]
```

**ReduceByKey** : perform reduce separately on each key value. Note: transformation, not action

```
A=sc.parallelize(\
   [(1,3),(4,100),(1,-5),(3,2)])

A.reduceByKey(lambda x,y: x*y)
 .collect()
```

```
# output:
[(1,-15),(4,100),(3,2)]
```

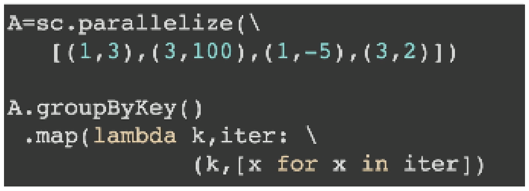

**A detour: iterators**

```
## Waste of memory space:
for i in range(1000000000):
   # do something

## No Waste:
for i in xrange(1000000000):
   # do something
```

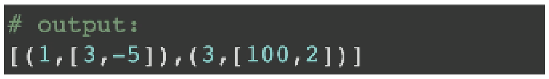
- range creates a large list and then iterates on it.
- xrange is an *iterator* that generates the elements one by one. Similar to C:

```
for(i=0; i<1000000000; i++)
   {Do something}
```

**groupByKey** : returns a (key,<iterator>) pair for each key value. The iterator iterates over the values corresponding to the key.



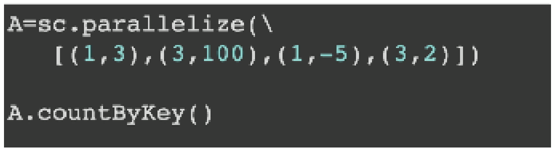
```
A=sc.parallelize(\
   [(1,3),(3,100),(1,-5),(3,2)])

A.groupByKey()
 .map(lambda k,iter: \
            (k,[x for x in iter])
```



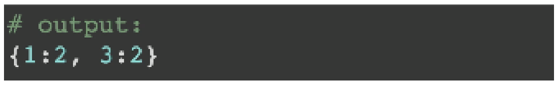
```
# output:
[(1,[3,-5]),(3,[100,2])]
```

### *(key, Value) RDDs Actions*

**countByKey** : returns a python Dictionary with the number of pairs for each key.

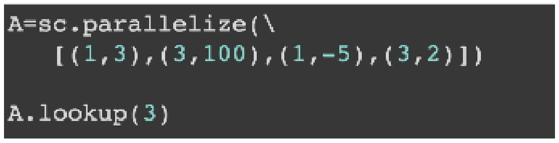


```
A=sc.parallelize(\
   [(1,3),(3,100),(1,-5),(3,2)])

A.countByKey()
```

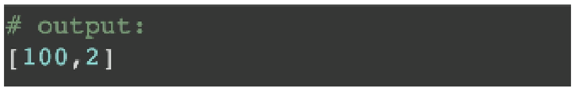


```
# output:
{1:2,  3:2}
```

**lookup(key)** : returns the list of all of the values associated with **key**

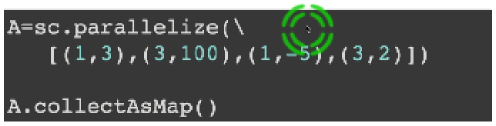


```
A=sc.parallelize(\
   [(1,3),(3,100),(1,-5),(3,2)])

A.lookup(3)
```

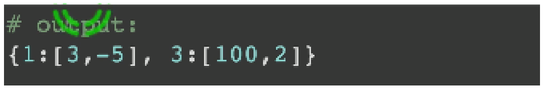


```
# output:
[100,2]
```

**collectAsMap()** : like collect() but instead of returning a list of tuples it returns a Map (=Dictionary)



```
A=sc.parallelize(\
   [(1,3),(3,100),(1,-5),(3,2)])

A.collectAsMap()
```



```
# output:
{1:[3,-5], 3:[100,2]}
```

## 2.3 QUIZ 1

## 2.4 ASSIGNMENT 1: COLLINEAR POINTS

In this programming assignment, you will write python3 code using pyspark to find sets of collinear points given arbitrary number of 2D points.
To download the tar file for PA1 click here. After Downloading the file, "cd" to the download directory and unpack it using the command: "tar -xzf pa1.tgz". You should get a directory named **pa1** which contains the jupyter notebook and the data files for PA1.
Depending on your installation setup, you could either mount the directory on docker (if you are using docker) OR just run jupyter locally and start working on the assignment.
We do not currently support submission and grading of assignments for audit learners. However, there are lots of visible test cases in the assignment notebook that you can use to verify the correctness of your code.

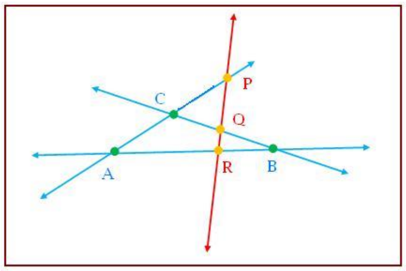
### Programming Assignment 1: Collinear Points

For this programming assignment, we'll be using a Jupyter notebook.

### *Background*

#### *Collinear points*

Definition of collinearity[1]: In geometry, collinearity of a set of points is the property of their lying on a single line. A set of points with this property is said to be collinear.



Here, points P,Q,R and A,R,B are collinear. However, points A,B,C are non-collinear. For more, refer [2].
1. https://en.wikipedia.org/wiki/Collinearity
2. http://www.mathcaptain.com/geometry/collinear-points.html

#### *Parameterizing lines*

In order to determine whether a set of points all lie on the same line we need a standard way to define (or parametrize) a line.

- One way of defining a line is as the set of points $(x,y)$(x,y) such that $y=ax+b$y=ax+b for some fixed real values $a,b$a,b.
- We call $a$a the **slope** of the line and $b$b is the $y$y-intercept which is defined as the value of $y$y when $x=0$x=0.
- This parameterization works for *almost* all lines. It does not work for vertical lines. For those lines we define $a$a to be **infinity** and $b$b to be the $x$x intercept of the line (the line is parallel to the $y$y axis so it does not intercept the $y$y axis (other than if it is the vertical line going through the origin).

To summarize, given two different points $(x_1,y_1)\neq(x_2,y_2)$(x1,y1)≠(x2,y2), we define the parameterization $(a,b)$(a,b) as:

- **if** $x_1=x_2$x1=x2**:** $(\text{Inf},x_1)$(Inf,x1)
- **Else:** $(a,b)$(a,b) such that $y_1=ax_1+b$y1=ax1+b and $y_2=ax_2+b$y2=ax2+b.

### *Task*

Given an input file with an arbitrary set of co-ordinates, your task is to use pyspark library functions and write a program in python3 to find if three or more points are collinear. For instance, if given these points: {(1,1), (0,1), (2,2), (3,3), (0,5), (3,4), (5,6), (0,-3), (-2,-2)} Sets of collinear points are: {((-2,-2), (1,1), (2,2), (3,3)), ((0,1), (3,4), (5,6)), ((0,-3), (0,1), (0,5))}. Note that the ordering of the points in a set or the order of the sets does not matter. Note:

- Every set of collinear points has to have **at least three points** (any pair of points lie on a line).
- There are two types of test cases:
  - **Visible Test cases**: Test cases given to you as a part of the notebook. These tests will help you validate your program and figure out bugs in it .
  - **Hidden Test cases**: Test cases that are not given as a part of the notebook, but will be used for grading.
    Cells in this notebook that have "##*Hidden test cases here*" are read-only cells containing hidden tests.
- Any cell that does not require you to submit code cannot be modified. For example: Assert statement unit test cells. Cells that have "**# YOUR CODE HERE**" are the ONLY ones you will need to alter.
- DO NOT change the names of functions.
- Remove the "Raise NotImplementedError()" line when you write the definition of your function.

### *Description of the Approach*

The goal of this assignment is to make you familiar with programming using pyspark. There are many ways to find sets of collinear points from a list of points. For the purposes of this assignment, we shall stick with the below approach:

1. List all pairs of points. You can do that efficiently in spark by computing cartesian product of the list of points with itself. For example, given three points [(1,0),(2,0),(3,0)], we construct a list of nine pairs
   [((1,0),(1,0)),((1,0),(2,0)),((1,0),(3,0))
   ((2,0),(1,0)),((2,0),(2,0)),((2,0),(3,0))
   ((3,0),(1,0)),((3,0),(2,0)),((3,0),(3,0))]
2. Remove the pairs in which the same point appears twice such as ((2,0),(2,0)). After these elimination you end up (for this example) with a list of just six pairs:
   [((1,0),(2,0)),((1,0),(3,0)),((2,0),(1,0)),((2,0),(3,0)),((3,0),(1,0)),((3,0),(2,0))]

3. For each pair of points, find the parameterization $(a,b)$ of the line connecting them as described above.
4. Group the pairs according to their parameters. Clearly, if two pairs have the same $(a,b)$ values, all points in the two pairs lie on the same line.
5. Eliminate groups that contain only one pair (any pair of points defines a line).
6. In each of the remaining groups, unpack the point-pairs to identify the individual points. Note that if a set of points $(x_1,y_1),\ldots,(x_k,y_k)$ lie on the same line then each point will appear $k-1$ times in the list of point-pairs. You therefore need to transform the list of points into sets to remove duplicates.
7. Output the sets of 3 or more colinear points.

Your task is to implement the described algorithm in Spark. You should use RDD's all the way through and collect the results into the driver only at the end.

## Notebook Setup

```
from pyspark import SparkContext, SparkConf

#Create SparkConf() object, use it to initialize spark context
conf = SparkConf().setAppName("Collinear
  Points").setMaster("local[4]") #Initialize spark context using
  4 local cores as workers
sc = SparkContext(conf=conf)

from pyspark.rdd import RDD
```

## Helper Functions

Here are some helper functions that you are encouraged to use in your implementations. Do not change these functions. The function format_result takes an element of the form shown below in the example. It outputs a tuple of all points that are collinear (shown below). Input: ((A,slope), [C1,..., Ck]) where each of A, C1, ..., Ck is a point of form (Ax, Ay) and slope is of type float.

**Example Code**

```
      my_input = (((2, 1), 0.5), [(4, 2), (6, 3)])
      format_result(my_input)
```

Output: (C1,..., Ck, A) each of A,C1,...,Ck is a point of form (Ax, Ay)

**Example Output**

```
      ((4, 2), (6, 3), (2, 1))
```

**Hint : ** The above example is given just to provide the input and output format. This function is called a different way in the spark exercise.

```
def format_result(x):
    x[1].append(x[0][0])
    return tuple(x[1])
def to_sorted_points(x):
    """
    Sorts and returns a tuple of points for further processing.
    """
    return tuple(sorted(x))
```

## Exercises

Here are some functions that you will implement. You should follow the function definitions, and use them appropriately elsewhere in the notebook.

### Exercise 1: to_tuple

**Example**

The function to_tuple converts each point of form 'Ax Ay' into a point of form (Ax, Ay) for further processing.

**Example Code**

```
      my_input = '2 3'
      to_tuple(my_input)
```

**Example Output**

```
      (2, 3)
```

**Hint : ** The above example is given just to provide the input and output format. This function is called a different way in the spark exercise.

**Definition**

```
## Insert answer in this cell. DON'T CHANGE NAME OF FUNCTION.
def to_tuple(x):
    # YOUR CODE HERE
    raise NotImplementedError()
```

**Unit Tests**

```
assert type(to_tuple('1 1')) == tuple, "Incorrect type: Element
  returned is not a tuple"
assert type(to_tuple('1 1')[0])==int and type(to_tuple('1
  1')[1])==int, "Incorrect element type: Element returned is not
  an integer"
assert to_tuple('1 1') == (1,1), "Incorrect Return Value: Value
  obtained does not match"
```

### Exercise 2: non_duplicates

**Example**

The function non_duplicates checks if a set of points contains duplicates or not. Input: Pair (A,B) where A and B are of form (Ax, Ay) and (Bx, By) respectively.

**Example Code**

```
      my_input = ((0,0),(1,2))
      non_duplicates(my_input)
```

Output: Returns True if A != B, False otherwise.

**Example Output**

```
      True
```

**Hint : ** The above example is given just to provide the input and output format. This function is called a different way in the spark exercise.

**Definition**

```
## Insert your answer in this cell. DO NOT CHANGE THE NAME OF THE
FUNCTION.
def non_duplicates(x):
    """
    Use this function inside the get_cartesian() function to
'filter' out pairs with duplicate points
    """
    # YOUR CODE HERE
    raise NotImplementedError()
```

**Unit Tests**

```
assert type(non_duplicates(((0,0),(1,2)))) == bool, "Incorrect
  Return type: Function should return a boolean value"
assert non_duplicates(((0,0),(1,2))) == True, "No duplicates are
  present"
assert non_duplicates(((0,0),(0,0))) == False, "Duplicates exist:
  (0,0)"
```

### Exercise 3: get_cartesian

**Example**

The function get_cartesian does a cartesian product of an RDD with itself and returns an RDD with **DISTINCT** pairs of points. Input: RDD containing the given list of points. Output: RDD containing The cartesian product of the RDD with itself

**Example Code**

```
      test_rdd = sc.parallelize([(1,0), (2,0), (3,0)])
      get_cartesian(test_rdd).collect()
```

**Example Output**

```
   [((1, 0), (2, 0)), ((1, 0), (3, 0)), ((2, 0), (1, 0)),
   ((2, 0), (3, 0)), ((3, 0), (1, 0)), ((3, 0), (2, 0))]
```

Refer: http://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=cartesian#pyspark.RDD.cartesian

**Definition**

```
## Insert your answer in this cell. DO NOT CHANGE THE NAME OF THE
FUNCTION.
def get_cartesian(rdd):
    # YOUR CODE HERE
    raise NotImplementedError()
```

**Unit Tests**

```
test_rdd = sc.parallelize([(1,0), (2,0), (3,0)])

l = [((1, 0), (2, 0)), ((1, 0), (3, 0)), ((2, 0), (1, 0)), ((2,
0), (3, 0)), ((3, 0), (1, 0)), ((3, 0), (2, 0))]

assert isinstance(get_cartesian(test_rdd), RDD) == True,
  "Incorrect Return type: Function should return an RDD"
assert set(get_cartesian(test_rdd).collect()) == set(l),
  "Incorrect Return Value: Value obtained does not match"
##Hidden test cases here
##Hidden test cases here
```

### Exercise 4: find_slope

**Example**

The function find_slope computes slope between points A and B and returns it in the format specified below. Input: Pair (A,B) where A and B are of form (Ax, Ay) and (Bx, By) respectively.

**Example Code**

```
      my_input = ((1,2),(3,4))
      find_slope(my_input)
```

Output: Pair ((A,slope), B) where A and B have the same definition as input and slope refers to the slope of the line segment connecting point A and B.

**Example Output**

```
      (((1, 2), 1.0), (3, 4))
```

**Note: ** If Ax == Bx, use slope as "inf".

**Hint : ** The above example is given just to provide the input and output format. This function is called a different way in the spark exercise.

**Definition**

```
## Insert your answer in this cell

def find_slope(x):
    # YOUR CODE HERE
    raise NotImplementedError()
```

**Unit Tests**

```
assert type(find_slope(((1,2),(3,4)))) == tuple, "Function must
  return a tuple"
assert find_slope(((1,2),(-7,-2)))[0][1] == 0.5, "Slope value
  should be 0.5"
assert find_slope(((1,2),(3,4))) == (((1,2),1),(3,4)), "Incorrect
  return value: Value obtained does not match"
assert find_slope(((1,2),(1,5))) == (((1,2),"inf"),(1,5)),
  "Incorrect return value: Value obtained must have slope 'inf'"
assert find_slope(((1,2),(2,5))) == (((1,2),3),(2,5)), "Incorrect
  return value: Value obtained does not match"
```

```
##Hidden test cases here
##Hidden test cases here
##Hidden test cases here
```

### Exercise 5: find_collinear

**Example**

The function find_collinear finds the set of collinear points. Input: An RDD (which is the output of the get_cartesian() function. Output: An RDD containing the list of collinear points formatted according to the format_result function.

Approach:

1. Find slope of the line between all pairs of points A = (Ax, Ay) and B = (Bx, By).
2. For each (A, B), find all points C = ((C1x, C1y), (C2x, C2y), ... (Cnx, Cny)) where slope of (A,B) = slope of (A, Ci).
3. Return (A, B, Ck) where Ck = all points of C which satisfy the condition 1.

The assert statement unit tests for this function will help you with this. **Hint : ** You should use the above helper functions in conjunction with Spark RDD API (refer http://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=rdd#pyspark.RDD) Finally, use helper function format_result() appropriately from inside this function after you have implemented the above operations.

**Definition**

```
def find_collinear(rdd):
    # YOUR CODE HERE
    raise NotImplementedError()
```

**Unit Tests**

```
def verify_collinear_sets(collinearpointsRDD, testlist):
    collinearpoints = [tuple(sorted(x)) for x in
      list(set(collinearpointsRDD.collect()))]
    testlist = [tuple(sorted(x)) for x in list(set(testlist))]
    return set(collinearpoints) == set(testlist)

test_rdd = sc.parallelize([((4, 2), (2, 1)), ((4, 2), (-3, 4)),
((4, 2), (6, 3)), ((2, 1), (4, 2)), ((2, 1), (-3, 4)), ((2, 1),
(6, 3)), ((-3, 4), (4, 2)), ((-3, 4), (2, 1)), ((-3, 4), (6, 3)),
((6, 3), (4, 2)), ((6, 3), (2, 1)), ((6, 3), (-3, 4))])
assert isinstance(find_collinear(test_rdd), RDD) == True,
  "Incorrect return type: Function must return RDD"

assert verify_collinear_sets(find_collinear(test_rdd), [((2, 1),
  (4, 2), (6, 3))]), "Incorrect return value: Value obtained does
  not match"
##Hidden test cases here
```

**Unit Tests II : Using the output of get_cartesian(rdd)**

```
test_rdd = sc.parallelize([(4, -2), (2, -1), (-3,4), (6,3), (-
  9,4), (6, -3), (8,-4), (6,9)])
test_rdd = get_cartesian(test_rdd)
assert verify_collinear_sets(find_collinear(test_rdd), [((6, -3),
  (6, 3), (6, 9)), ((2, -1), (4, -2), (6, -3), (8, -4))]),
  "Incorrect return value: You have not implemented the
  find_collinear function in Python"
##Hidden test cases here
```

### Exercise 6: The build_collinear_set function

**Example**

Using the above functions that you have written along with pyspark functions, write the **build_collinear_set** function and returns an RDD containing the set of collinear points. Input: RDD containing the given set of points Output: RDD containing the set of collinear points

**Hint : ** Remember that the input RDD consists of a set of strings. Remember to pre-process them using the to_tuple function before performing other operations.

**Definition**

```
def build_collinear_set(rdd):

# YOUR CODE HERE
    raise NotImplementedError()

# Sorting each of your returned sets of collinear points. This is
for grading purposes.
    # YOU MUST NOT CHANGE THIS.
    rdd = rdd.map(to_sorted_points)

    return rdd
```

**Unit Tests**

```
test_rdd = sc.parallelize(['4 -2', '2 -1', '-3 4', '6 3', '-9 4',
  '6 -3', '8 -4', '6 9'])
assert isinstance(build_collinear_set(test_rdd), RDD) == True,
  "build_collinear_set should return an RDD."
```

### The process function

**Definition**

```
def process(filename):
    """
This is the process function used for finding collinear points
using inputs from different files
Input: Name of the test file
Output: Set of collinear points
    """
    # Load the data file into an RDD
    rdd = sc.textFile(filename)

    rdd = build_collinear_set(rdd)

# Collecting the collinear points RDD in a set to remove
duplicate sets of collinear points. This is for grading purposes.
You may ignore this.
    res = set(rdd.collect())

    return res
```

**Unit Tests: Testing the build_collinear_set function using the process function**

NOTE: You may assume that input files do not have duplicate points.

```
assert process("data.txt") == {((-2, -2), (1, 1), (2, 2), (3,
  3)), ((0, 1), (3, 4), (5, 6)), ((0, -3), (0, 1), (0, 5))},
  "Your implementation of build_collinear_set is not correct."
assert process("data50.txt") ==
{((3, 6), (7, 4), (9, 3)), ((1, 6), (3, 6), (4, 6), (7, 6)),
 ((0, 2), (3, 1), (6, 0)), ((1, 0), (2, 0), (5, 0), (6, 0)),
 ((1, 3), (3, 6), (5, 9)), ((0, 8), (4, 6), (6, 5)),
 ((6, 0), (6, 1), (6, 5), (6, 9)),
 ((7, 2), (7, 3), (7, 4), (7, 6), (7, 8)), ((3, 1), (3, 3), (3, 6)),
 ((0, 2), (1, 2), (5, 2), (7, 2)), ((0, 3), (2, 5), (3, 6), (6, 9)),
 ((0, 2), (1, 3), (2, 4), (4, 6), (5, 7)), ((1, 2), (4, 3), (7, 4)),
 ((0, 3), (4, 6), (8, 9)), ((9, 3), (9, 4), (9, 5)), ((2, 5), (5, 7), (8, 9)),
 ((0, 5), (2, 4), (4, 3), (8, 1)), ((0, 8), (1, 6), (2, 4)),
  ((3, 6), (5, 2), (6, 0)), ((5, 9), (6, 9), (8, 9)),
  ((0, 8), (1, 8), (7, 8)), ((0, 4), (1, 3), (3, 1)), ((5, 9), (7, 6), (9, 3)),
  ((1, 2), (2, 4), (3, 6)), ((0, 7), (1, 5), (3, 1)),
  ((1, 5), (2, 4), (3, 3), (6, 0)), ((0, 2), (3, 3), (9, 5)),
  ((0, 7), (1, 6), (2, 5), (4, 3), (5, 2), (6, 1)),
 ((0, 4), (1, 5), (5, 9)), ((1, 5), (3, 6), (5, 7), (7, 8)),
  ((1, 6), (3, 3), (5, 0)), ((3, 6), (4, 3), (5, 0)),
  ((1, 2), (4, 5), (7, 8), (8, 9)), ((0, 2), (1, 1), (2, 0)),
  ((3, 3), (4, 5), (5, 7), (6, 9)), ((0, 2), (0, 3), (0, 4), (0, 5), (0, 7), (0, 8)),
 ((2, 0), (4, 3), (8, 9)), ((5, 7), (6, 5), (7, 3), (8, 1)), ((5, 0), (7, 6), (8, 9)),
 ((5, 0), (6, 1), (7, 2), (9, 4)), ((0, 4), (1, 2), (2, 0)),
  ((1, 1), (3, 1), (6, 1), (8, 1)), ((5, 7), (7, 6), (9, 5)), ((1, 1), (7, 4), (9, 5)),
 ((0, 4), (2, 4), (7, 4), (9, 4)), ((1, 0), (3, 1), (5, 2), (7, 3), (9, 4)),
 ((2, 0), (3, 3), (4, 6), (5, 9)), ((4, 3), (4, 5), (4, 6)),
  ((1, 0), (4, 3), (6, 5), (7, 6)), ((0, 3), (2, 4), (4, 5)),
  ((1, 6), (4, 5), (7, 4)), ((1, 0), (1, 1), (1, 2), (1, 3), (1, 5), (1, 6), (1, 8)),
  ((0, 3), (1, 3), (3, 3), (4, 3), (7, 3), (9, 3)), ((0, 4), (2, 5), (4, 6)),
  ((0, 7), (3, 6), (6, 5), (9, 4)), ((1, 8), (4, 6), (7, 4)),
   ((0, 5), (3, 3), (6, 1)), ((1, 8), (3, 6), (4, 5), (7, 2), (8, 1)),
  ((1, 2), (3, 1), (5, 0)), ((1, 1), (5, 2), (9, 3)),
  ((5, 0), (5, 2), (5, 7), (5, 9)), ((0, 5), (1, 5), (2, 5), (4, 5), (6, 5), (9, 5)),
  ((3, 1), (4, 5), (5, 9)), ((2, 0), (2, 4), (2, 5)), ((5, 2), (6, 5), (7, 8))}, "Your
  implementation of build_collinear_set is not correct."
##Hidden test cases here
##Hidden test cases here
##Hidden test cases here
##Hidden test cases here
```

## 2.5 PYSPARK AND RDDS

### 2.5.1 Spark Notebook Basics pt. 1

Which library do we need to import to create a SparkContext?

- pyspark 98%
- matplotlib 2%
- pandas 0%
- scipy 0%

Which of the following stops the SparkContext kernel?

- sc.stop() 92%
- Kernels are not required 5%
- kern.exit() 2%
- exit 1%

What did Professor Freund say is the simplest way to create an RDD?

- A = sc.parallelize(L) where L is a list 97%
- sparkContext(A) where A is an RDD object 2%
- A = sc.create(L) where L is a list 1%

This notebook introduces two fundamental objects in Spark:

- The Spark Context
- The Resilient Distributed DataSet or RDD

### Spark Context

We start by creating a **SparkContext** object named **sc**. In this case we create a spark context that uses 4 *executors* (one per core)

```
#start the SparkContext
import findspark
findspark.init()

from pyspark import SparkContext
sc = SparkContext(master="local[4]")
print(sc)
<SparkContext master=local[4] appName=pyspark-shell>
```

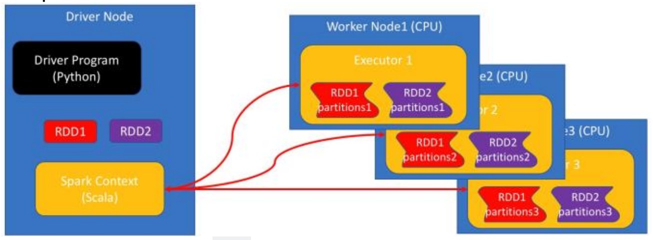
#### Only one sparkContext at a time!

- Spark is designed for single user
- Only one sparkContext per program/notebook.
- Before starting a new sparkContext. Stop the one currently running

```
# sc.stop() #commented out so that you don't stop your
context by mistake
```

## RDDs

RDD (or Resilient Distributed DataSet) is the main novel data structure in Spark. You can think of it as a list whose elements are stored on several computers.



The elements of each `RDD` are distributed across the **worker nodes** which are the nodes that perform the actual computations. This notebook, however, is running on the **Driver node**. As the RDD is not stored on the driver-node you cannot access it directly. The variable name `RDD` is really just a pointer to a python object which holds the information regardnig the actual location of the elements.

### Some basic RDD commands

#### Parallelize

- Simplest way to create an RDD.
- The method `A=sc.parallelize(L)`, creates an RDD named `A` from list `L`.
- `A` is an RDD of type `PythonRDD`.

```
A=sc.parallelize(range(3))
A
PythonRDD[1] at RDD at PythonRDD.scala:48
```

#### Collect

- RDD content is distributed among all executors.
- `collect()` is the inverse of `parallelize()'
- collects the elements of the RDD
- Returns a list

```
L=A.collec t()
print(type(L))
print(L)
<class 'list'>
[0, 1, 2]
```

**Using `.collect()` eliminates the benefits of parallelism**

It is often tempting to `.collect()` and RDD, make it into a list, and then process the list using standard python. However, note that this means that you are using only the head node to perform the computation which means that you are not getting any benefit from spark.
Using RDD operations, as described below, **will** make use of all of the computers at your disposal.

#### Map

- applies a given operation to each element of an RDD
- parameter is the function defining the operation.
- returns a new RDD.
- Operation performed in parallel on all executors.
- Each executor operates on the data local to it.

```
A.map(lambda x: x*x).collect()
 [0, 1, 4]
```

**Note:** Here we are using **lambda** functions, later we will see that regular functions can also be used.
For more on lambda function see here

#### Reduce

- Takes RDD as input, returns a single value.
- **Reduce operator** takes **two** elements as input returns **one** as output.
- Repeatedly applies a **reduce operator**
- Each executor reduces the data local to it.
- The results from all executors are combined.

The simplest example of a 2-to-1 operation is the sum:

```
A.reduce(lambda x,y:x+y)
3
```

Here is an example of a reduce operation that finds the shortest string in an RDD of strings.

```
words=['this','is','the','best','mac','ever']
wordRDD=sc.parallelize(words)
wordRDD.reduce(lambda w,v: w if len(w)<len(v) else v)
'is'
```

**Properties of reduce operations**

- Reduce operations **must not depend on the order**
  - Order of operands should not matter
  - Order of application of reduce operator should not matter
- Multiplication and summation are good:

```
                    1 + 3 + 5 + 2
    5 + 3 + 1 + 2
```

- Division and subtraction are bad:

```
                    1 - 3 - 5 - 2
    1 - 3 - 5 - 2
```

**Why must reordering not change the result?**

You can think about the reduce operation as a binary tree where the leaves are the elements of the list and the root is the final result. Each triplet of the form (parent, child1, child2) corresponds to a single application of the reduce function. The order in which the reduce operation is applied is **determined at run time** and depends on how the RDD is partitioned across the cluster. There are many different orders to apply the reduce operation. If we want the input RDD to uniquely determine the reduced value **all evaluation orders must must yield the same final result**. In addition, the order of the elements in the list must not change the result. In particular, reversing the order of the operands in a reduce function must not change the outcome.
For example the arithmetic operations multiply `*` and add `+` can be used in a reduce, but the operations subtract `-` and divide `/` should not.
Doing so will not raise an error, but the result is unpredictable.

```
B=sc.parallelize([1,3,5,2])
B.reduce(lambda x,y: x-y)
-9
```

Which of these the following orders was executed?

- $((1-3)-5)-2((1-3)-5)-2$

  or

- $(1-3)-(5-2)(1-3)-(5-2)$

**Using regular functions instead of lambda functions**

- lambda function are short and sweet.
- but sometimes it's hard to use just one line.
- We can use full-fledged functions instead.

```
A.reduce(lambda x,y: x+y)
3
```

Suppose we want to find the

- last word in a lexicographical order
- among
- the longest words in the list.

We could achieve that as follows

```
def largerThan(x,y):
    if len(x)>len(y): return x
    elif len(y)>len(x): return y
    else:  #lengths are equal, compare lexicographically
        if x>y:
            return x
        else:
            return y

wordRDD.reduce(largerThan)
'this'
```

**Summary**

We saw how to:

- Start a SparkContext
- Create an RDD
- Perform Map and Reduce operations on an RDD
- Collect the final results back to head node.

## 2.5.2 Changing Number of Workers

**The effect of changing the number of workers**

- When you initialize SparkContext, you can specify the number of workers.
- Usually the recommendation is for one worker per core.
- But the number of workers can be smaller or larger than the number of cores

```
from time import time
from pyspark import SparkContext
for j in range(1,10):
    sc = SparkContext(master="local[%d]"%(j))
    t0=time()
    for i in range(10):
        sc.parallelize([1,2]*1000000).reduce(lambda x,y:x+y)
    print("%2d executors, time=%4.3f"%(j,time()-t0))
    sc.stop()
 1 executors, time=17.836
 2 executors, time=14.356
 3 executors, time=14.686
 4 executors, time=14.881
 5 executors, time=15.043
 6 executors, time=15.079
 7 executors, time=15.422
 8 executors, time=15.360
 9 executors, time=15.856
```

**Summary**

- This machine has 4 cores
- Increasing the number of executors from 1 to 3 speeds up the computation
- From 3 and up you have fluctuations in performance.
- More than one worker per core is usually unhelpful

## 2.5.3 Execution Plans Lazy Eval and Caching

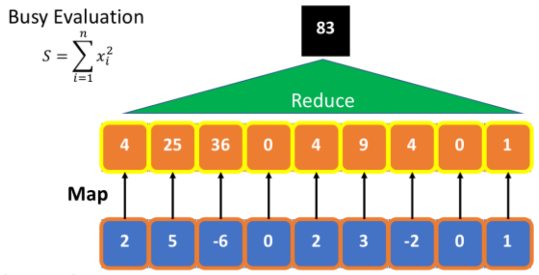
Task: calculate the sum of squares :

$$\sum_{i=1}^{n} x_i^2$$

The standard (or **busy**) way to do this is
1. Calculate the square of each element.
2. Sum the squares.

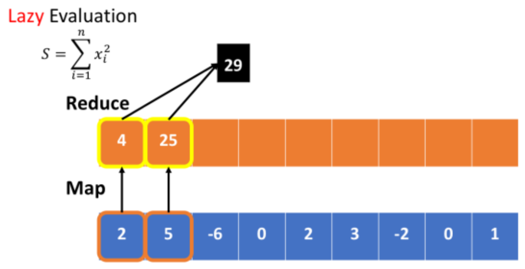
This requires **storing** all intermediate results.

### Busy Evaluation



**lazy evaluation:**

- **postpone** computing the square until result is needed.
- No need to store intermediate results.
- Scan through the data once, rather than twice.



### Lazy Evaluation

Unlike a regular python program, map/reduce commands do not always perform any computation when they are executed. Instead, they construct something called an **execution plan**. Only when a result is needed does the computation start. This approach is also called **lazy execution**.
The benefit from lazy execution is in minimizing the the number of memory accesses. Consider for example the following map/reduce commands:

```
        A=RDD.map(lambda x:x*x).filter(lambda x: x%2==0)
        A.reduce(lambda x,y:x+y)
```

The commands defines the following plan. For each number `x` in the RDD:
1. Compute the square of `x`
2. Filter out `x*x` whose value is odd.
3. Sum the elements that were not filtered out.

A naive execution plan is to square all items in the RDD, store the results in a new RDD, then perform a filtering pass, generating a second RDD, and finally perform the summation. Doing this will require iterating through the RDD three times, and creating 2 interim RDDs. As memory access is the bottleneck in this type of computation, the execution plan is slow.
A better execution plan is to perform all three operations on each element of the RDD in sequence, and then move to the next element. This plan is faster because we iterate through the elements of the RDD only once, and because we don't need to save the intermediate results. We need to maintain only one variable: the partial sum, and as that is a single variable, we can use a CPU register.
For more on RDDs and lazy evaluation see here in the spark manual

### Experimenting with Lazy Evaluation

#### The `%%time` magic

The `%%time` command is a *cell magic* which measures the execution time of the cell. We will mostly be interested in the wall time, which includes the time it takes to move data in the memory hierarchy.
For more on jupyter magics See here

#### Preparations

In the following cells we create an RDD and define a function which wastes some time and then returns `cos(i)`. We want the function to waste some time so that the time it takes to compute the `map` operation is significant.

```
from pyspark import SparkContext
sc = SparkContext(master="local[4]")   #note that we set the
number of workers to 3
```

We create an RDD with one million elements to amplify the effects of lazy evaluation and caching.

```
%%time
RDD=sc.parallelize(range(1000000))
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 604 ms
```

It takes about 01.-0.5 sec. to create the RDD.

```
print(RDD.toDebugString().decode())
(4) PythonRDD[1] at RDD at PythonRDD.scala:48 []
 |  ParallelCollectionRDD[0] at parallelize at
PythonRDD.scala:175 []
```

#### Define a computation

The role of the function `taketime` is to consume CPU cycles.

```
from math import cos
def taketime(i):
    [cos(j) for j in range(100)]
    return cos(i)
%%time
taketime(1)
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 52 µs
0.5403023058681398
```

#### Time units

- 1 second = 1000 Milli-second (msms)
- 1 Millisecond = 1000 Micro-second (µsµs)
- 1 Microsecond = 1000 Nano-second (nsns)

#### Clock Rate

One cycle of a 3GHz cpu takes 13ns13ns
`taketime(1000)` takes about 25 µsµs = 75,000 clock cycles.

#### The `map` operation.

```
%%time
Interm=RDD.map(lambda x: taketime(x))
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 33.4 µs
```

#### How come so fast?

- We expect this map operation to take 1,000,000 * 25 µs = 25 Seconds.
- **Why** did the previous cell take just 29 µs?
- Because **no** computation **was done**
- The cell defined an execution **plan**, but did not execute it yet.

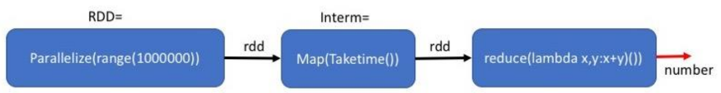
**Lazy Execution** refers to this type of behaviour. The system delays actual computation until the latest possible moment. Instead of computing the content of the RDD, it adds the RDD to the **execution plan**.
Using Lazy evaluation of a plan has two main advantages relative to immediate execution of each step:
1. A single pass over the data, rather than multiple passes.
2. Smaller memory footprint becase no intermediate results are saved.

#### Execution Plans

At this point the variable `Interm` does not point to an actual data structure. Instead, it points to an execution plan expressed as a **dependence graph**. The dependence graph defines how the RDDs are computed from each other.
The dependence graph associated with an RDD can be printed out using the method `toDebugString()`.

```
print(Interm.toDebugString().decode())
(4) PythonRDD[2] at RDD at PythonRDD.scala:48 []
 |  ParallelCollectionRDD[0] at parallelize at
PythonRDD.scala:175 []
```

**Interm=** `(4) PythonRDD[2] at RDD at PythonRDD.scala:48 []`
______ `(4)` corresponds to the number of partitions
**RDD** = `| ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:489 []`
At this point only the two left blocks of the plan have been declared.



### Actual execution

The `reduce` command needs to output an actual output, **spark** therefor has to actually execute the `map` and the `reduce`. Some real computation needs to be done, which takes about 1 - 3 seconds (Wall time) depending on the machine used and on it's load.

```
%%time
print('out=',Interm.reduce(lambda x,y:x+y))
out= -0.2887054679684464
CPU times: user 10 ms, sys: 10 ms, total: 20 ms
Wall time: 25 s
```

### How come so fast? (take 2)

- We expect this map operation to take 1,000,000 * 25 μsμs = 25 Seconds.
- Map+reduce takes only ~4 second.
- Why?
- Because we have 4 workers, rather than one.
- Because the measurement of a single iteration of `taketime` is an overestimate.

### Executing a different calculation based on the same plan.

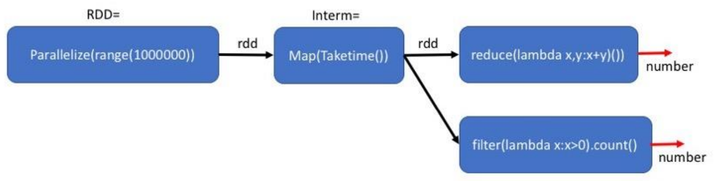
The plan defined by `Interm` might need to be executed more than once.
**Example:** compute the number of map outputs that are larger than zero.

```
%%time
print('out=',Interm.filter(lambda x:x>0).count())
out= 500000
CPU times: user 20 ms, sys: 0 ns, total: 20 ms
Wall time: 22.7 s
```

### The price of not materializing

- The run-time (3.4 sec) is similar to that of the reduce (4.4 sec).
- Because the intermediate results in `Interm` have not been saved in memory (materialized)
- They need to be recomputed.

The middle block: `Map(Taketime)` is executed twice. Once for each final step.

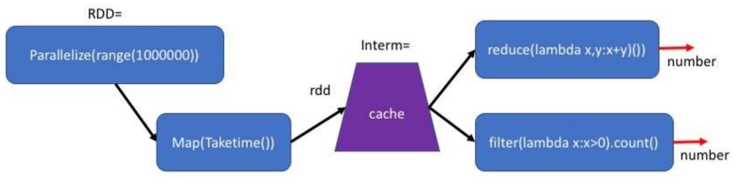


### Caching intermediate results

- We sometimes want to keep the intermediate results in memory so that we can reuse them later without recalculating. * This will reduce the running time, at the cost of requiring more memory.
- The method `cache()` indicates that the RDD generates in this plan should be stored in memory. Note that this is a **plan to cache**. The actual caching will be done only when the final result is needed.

```
%%time
Interm=RDD.map(lambda x: taketime(x)).cache()
CPU times: user 10 ms, sys: 0 ns, total: 10 ms
Wall time: 47.1 ms
```

By adding the Cache after `Map(Taketime)`, we save the results of the map for the second computation.



### Plan to cache

The definition of `Interm` is almost the same as before. However, the *plan* corresponding to `Interm` is more elaborate and contains information about how the intermediate results will becached and replicated.
Note that `PythonRDD[4]` is now [Memory Serialized 1x Replicated]

```
print(Interm.toDebugString().decode())
(4) PythonRDD[5] at RDD at PythonRDD.scala:48 [Memory Serialized
1x Replicated]
 |  ParallelCollectionRDD[0] at parallelize at
PythonRDD.scala:489 [Memory Serialized 1x Replicated]
```

### Comparing plans with and without cache

Plan with Cache

```
        (4) PythonRDD[33] at RDD at PythonRDD.scala:48
        [Memory Serialized 1x Replicated]
         |  ParallelCollectionRDD[0] at parallelize at
        PythonRDD.scala:489 [Memory Serialized 1x Replicated]
```

Plan without Cache

```
        (4) PythonRDD[2] at RDD at PythonRDD.scala:48 []
         |  ParallelCollectionRDD[0] at parallelize at
        PythonRDD.scala:489 []
```

The difference is that the plan for both RDDs includes **[Memory Serialized 1x Replicated]** which is the plan to materialize both RDDs when they are computed.

### Creating the cache

The following command executes the first map-reduce command **and** caches the result of the `map`command in memory.

```
%%time
print('out=',Interm.reduce(lambda x,y:x+y))
out= -0.2887054679684655
CPU times: user 5.43 ms, sys: 3.79 ms, total: 9.22 ms
Wall time: 3.59 s
```

### Using the cache

This time `Interm` is cached. Therefor the second use of `Interm` is much faster than when we did not use `cache`: 0.25 second instead of 1.9 second. (your milage may vary depending on the computer you are running this on).

```
%%time
print('out=',Interm.filter(lambda x:x>0).count())
out= 500000
CPU times: user 5.31 ms, sys: 2.97 ms, total: 8.28 ms
Wall time: 121 ms
```

### Summary

- Spark uses **Lazy Evaluation** to save time and space.
- When the same RDD is needed as input for several computations, it can be better to keep it in memory, also called `cache()`.
- Next Video, Partitioning and Gloming

### Partitioning and Gloming

- When an RDD is created, you can specify the number of partitions.
- The default is the number of workers defined when you set up `SparkContext`

```
A=sc.parallelize(range(1000000))
print(A.getNumPartitions())
4
```

We can repartition `A` into a different number of partitions.

```
D=A.repartition(10)
print(D.getNumPartitions())
10
```

We can also define the number of partitions when creating the RDD.

```
A=sc.parallelize(range(1000000),numSlices=10)
print(A.getNumPartitions())
10
```

### Why is the #Partitions important?

- They define the unit the executor works on.
- You should have at least as pany partitions as workers.
- Smaller partitions can allow more parallelization.

### Repartitioning for Load Balancing

Suppose we start with 10 partitions, all with exactly the same number of elements

```
A=sc.parallelize(range(1000000))\
    .map(lambda x:(x,x)).partitionBy(10)
print(A.glom().map(len).collect())
[100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000,
100000, 100000]
```

- Suppose we want to use `filter()` to select some of the elements in `A`.
- Some partitions might have more elements remaining than others.

```
#select 10% of the entries
B=A.filter(lambda pair: pair[0]%5==0)
# get no. of partitions
print(B.glom().map(len).collect())
[100000, 0, 0, 0, 0, 100000, 0, 0, 0, 0]
```

- Future operations on B will use only two workers.
- The other workers will do nothing,
  because their partitions are empty.
- To fix the situation we need to repartition the RDD.
- One way to do that is to repartition using a new key.
- The method **`.partitionBy(k)`** expects to get a **`(key,value)`** RDD where keys are integers.
- Partitions the RDD into **`k`** partitions.
- The element **`(key,value)`** is placed into partition no. **`key % k`**

```
C=B.map(lambda pair:(pair[1]/10,pair[1])).partitionBy(10)
print(C.glom().map(len).collect())
[20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000,
20000]
```

Another approach is to use random partitioning using **`repartition(k)`**

- An **advantage** of random partitioning is that it does not require defining a key.
- A **disadvantage** of random partitioning is that you have no control on the partitioning.

```
C=B.repartition(10)
print(C.glom().map(len).collect())
```

```
[20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000,
20000]
```

#### *Glom()*

- In general, spark does not allow the worker to refer to specific elements of the RDD.
- Keeps the language clean, but can be a major limitation.
- **glom()** transforms each partition into a tuple (immutabe list) of elements.
- Creates an RDD of tules. One tuple per partition.
- workers can refer to elements of the partition by index.
- but you cannot assign values to the elements, the RDD is still immutable.
- Now we can understand the command used above to count the number of elements in each partition.
- We use `glom()` to make each partition into a tuple.
- We use `len` on each partition to get the length of the tuple - size of the partition.
- We `collect` the results to print them out.

```
print(C.glom().map(len).collect())
[20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000, 20000,
20000]
```

#### *A more elaborate example*

There are many things that you can do using `glom()`.
Below is an example, can you figure out what it does?

```
def getPartitionInfo(G):
    d=0
    if len(G)>1:
        for i in range(len(G)-1):
            d+=abs(G[i+1][1]-G[i][1]) # access the glomed RDD
that is now a  list
        return (G[0][0],len(G),d)
    else:
        return(None)

output=B.glom().map(lambda B: getPartitionInfo(B)).collect()
print(output)
[(0, 100000, 999990), None, None, None, None, (5, 100000,
999990), None, None, None, None]
```

#### *Summary*

- We learned why partitions are important and how to control them.
- We Learned how `glom()` can be used to allow workers to access their partitions as lists.

### 2.5.4 Partition and Gloming

### 2.5.5 Spark Basics pt. 2

### 2.5.6 Word Count

### 2.5.7 Finding Most Common Words

### 2.5.8 Operations on KeyVal RDDs

## 2.6 SPARK SQL AND DATAFRAMES

### 2.6.1 Dataframes

### 2.6.2 Dataframe Operations

### 2.6.3 Loading and Using Parquet

## 2.7 PREPARING FOR DATA ANALYSIS

### 2.7.1 Where is the Data Dense

### 2.7.2 Moving and Deserialization

### 2.7.3 Deserialization

### 2.7.4 Transform RDD into Spark

## 2.8 QUIZ 2

## 2.9 PROGRAMMING ASSIGNMENT 2

# 3 PCA AND WEATHER ANALYSIS

## 3.1 COVARIANCE AND PCA

## 3.2 VISUALIZING PCA COEFFICIENTS

## 3.3 QUIZ 3

## 3.4 PROGRAMMING ASSIGNMENT 3

## 3.5 VIZUALIZING PCA RESIDUALS I

## 3.6 VIZUALIZING PCA RESIDUALS II

## 3.7 QUIZ 4

## 3.8 PROGRAMMING ASSIGNMENT 4

# 4 K-MEANS AND INTRINSIC DIMENSIONS

## 4.1 K-MEANS CLUSTERING

## 4.2 QUIZ 5

## 4.3 INTRINSIC DIMENSION

## 4.4 QUIZ 6

## 4.5 PROGRAMMING ASSIGNMENT 5

## 4.6 Decision Trees, Random Forests and Boosting

## 4.7 Decision Trees

## 4.8 Quiz 7

## 4.9 Programming Assignmnet 6

## 4.10 Boosting

## 4.11 Ensembles

## 4.12 Real-world Application for PCA and Boosting

## 4.13 Quiz 8

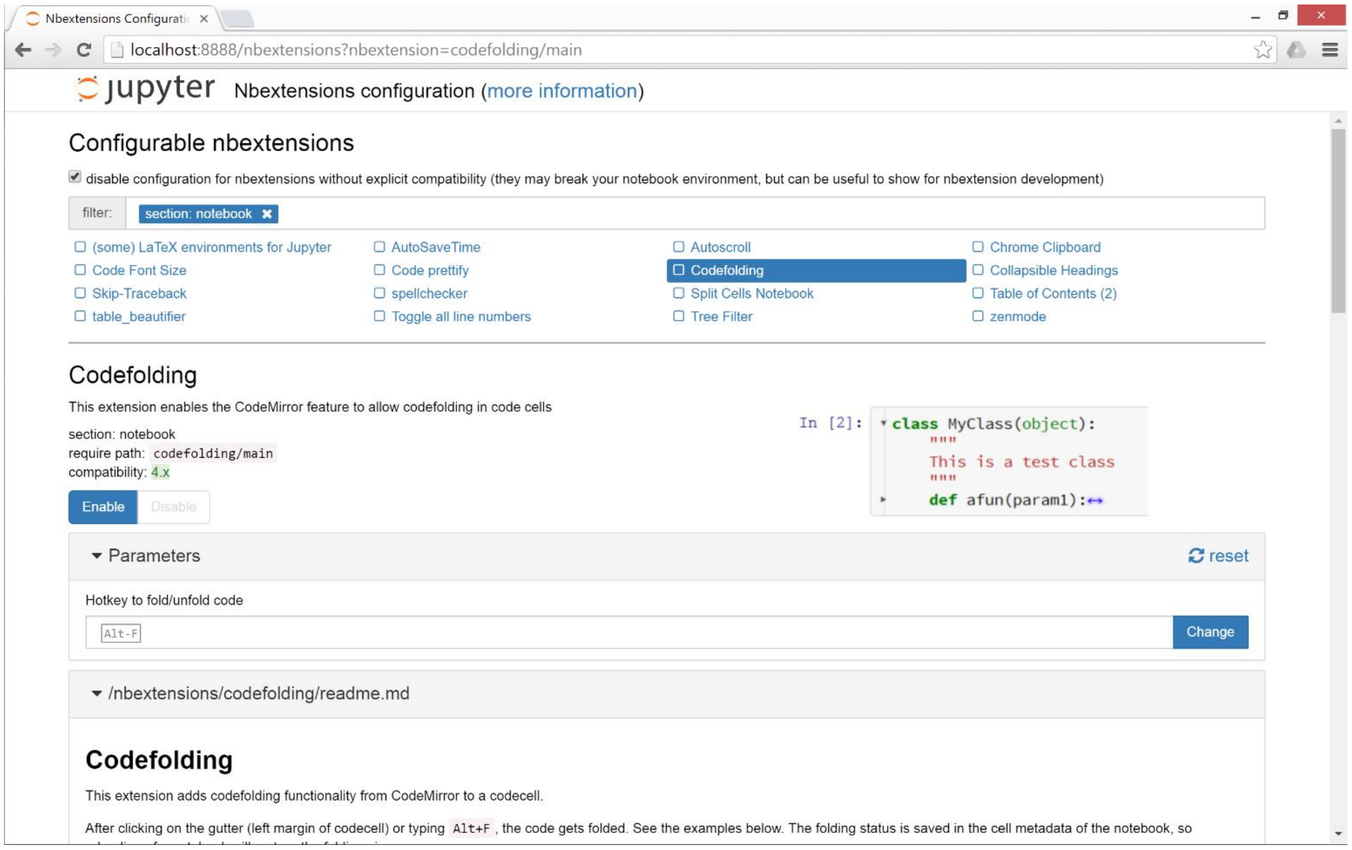
# APPENDIX 1 JUPYTER NBEXTENSIONS CONFIGURATOR

**LINK**

A server extension for jupyter notebook which provides configuration interfaces for notebook extensions (nbextensions). The jupyter_nbextensions_configurator jupyter server extension provides graphical user interfaces for configuring which nbextensions are enabled (load automatically for every notebook). In addition, for nbextensions which include an appropriate yaml descriptor file (see below), the interface also renders their markdown readme files, and provides controls to configure the nbextensions' options.

## USAGE

Once jupyter_nbextensions_configurator is installed and enabled, and your notebook server has been restarted, you should be able to find the nbextensions configuration interface at the url <base_url>nbextensions, where <base_url> is described below (for simple installs, it's usually just /, so the UI is at /nbextensions).



Nbextensions Configuratio ×

localhost:8888/nbextensions?nbextension=codefolding/main

jupyter Nbextensions configuration (more information)

Configurable nbextensions

☑ disable configuration for nbextensions without explicit compatibility (they may break your notebook environment, but can be useful to show for nbextension development)

filter: section: notebook ✖

| | | | |
|---|---|---|---|
| ☐ (some) LaTeX environments for Jupyter | ☐ AutoSaveTime | ☐ Autoscroll | ☐ Chrome Clipboard |
| ☐ Code Font Size | ☐ Code prettify | ☐ Codefolding | ☐ Collapsible Headings |
| ☐ Skip-Traceback | ☐ spellchecker | ☐ Split Cells Notebook | ☐ Table of Contents (2) |
| ☐ table_beautifier | ☐ Toggle all line numbers | ☐ Tree Filter | ☐ zenmode |

Codefolding

This extension enables the CodeMirror feature to allow codefolding in code cells

section: notebook
require path: `codefolding/main`
compatibility: 4.x

Enable Disable

```
In [2]: class MyClass(object):
            """
            This is a test class
            """
            def afun(param1):↔
```

Parameters reset

Hotkey to fold/unfold code

Alt-F Change

/nbextensions/codefolding/readme.md

Codefolding

This extension adds codefolding functionality from CodeMirror to a codecell.

After clicking on the gutter (left margin of codecell) or typing `Alt+F`, the code gets folded. See the examples below. The folding status is saved in the cell metadata of the notebook, so

### base_url

For most single-user notebook servers, the dashboard (the file-browser or 'tree' view) is at

```
http://localhost:8888/tree
```

So the `base_url` is the part between the host (`http://localhost:8888`) and `tree`, so in this case it's the default value of just `/`. If you have a non-default base url (such as with JupyterHub), you'll need to prepend it to the url. So, if your dashboard is at

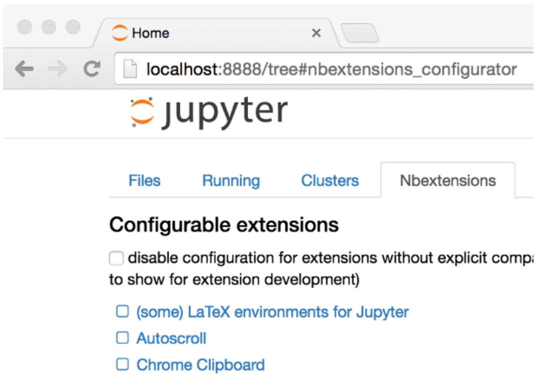
```
http://localhost:8888/custom/base/url/tree
```

then you'll find the configurator UI page at

```
http://localhost:8888/custom/base/url/nbextensions
```

### tree tab

In addition to the main standalone page, the nbextensions configurator interface is also available as a tab on the dashboard:



Home ×

localhost:8888/tree#nbextensions_configurator

jupyter

Files Running Clusters Nbextensions

Configurable extensions

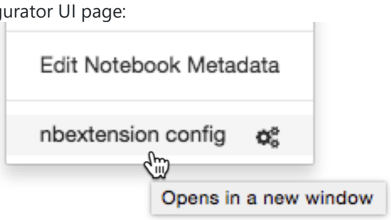
☐ disable configuration for extensions without explicit compa
to show for extension development)

☐ (some) LaTeX environments for Jupyter
☐ Autoscroll
☐ Chrome Clipboard

The dashboard tab is provided via an nbextension called "Nbextensions dashboard tab", with requirejs uri`nbextensions_configurator/tree_tab/main`. Since version 0.2.0, this nbextension is enabled by default on enabling the `jupyter_nbextensions_configurator` serverextension, but it can be disabled as with any other nbextension if you don't want to use it.

### edit menu item

`jupyter_nbextensions_configurator` provides a second small nbextension, which simply adds an item to the notebook-view edit menu, which links to the configurator UI page:



Similarly to the tree tab nbextension detailed above, since version 0.2.0, the edit menu item nbextension is enabled by default when enabling the main `jupyter_nbextensions_configurator` serverextension, but can be disabled at any time in the same way as other nbextensions.

## YAML FILE FORMAT

You don't need to know about the yaml files in order simply to use `jupyter_nbextensions_configurator`. An nbextension is 'found' by the `jupyter_nbextensions_configurator` server extension when a special yaml file describing the nbextension and its options is found in the notebook server's `nbextensions_path`. The yaml file can have any name with the file

extension `.yaml` or `.yml`, and describes the nbextension and its options to `jupyter_nbextensions_configurator`.
The case-sensitive keys in the yaml file are as follows:

- `Type`, (**required**) a case-sensitive identifier, must be `IPython Notebook Extension` or `Jupyter Notebook Extension`
- `Main`, (**required**) the main javascript file that is loaded, typically `main.js`
- `Name`, the name of the nbextension
- `Section`, which view the nbextension should be loaded in (defaults to `notebook`, but can alternatively be `tree`, `edit`, or to load in all views, `common`).
- `Description`, a short explanation of the nbextension
- `Link`, a URL for more documentation. If this is a relative url with a `.md` file extension (recommended!), the markdown readme is rendered in the configurator UI.
- `Icon`, a URL for a small icon for the configurator UI (rendered 120px high, should preferably end up 400px wide. Recall HDPI displays may benefit from a 2x resolution icon).
- `Compatibility`, Jupyter major version compatibility, e.g. `3.x` or `4.x`, `3.x 4.x`, `3.x, 4.x, 5.x`
- `Parameters`, an optional list of configuration parameters. Each item is a dictionary with (some of) the following keys
  - `name`, (**required**) the name used to store the configuration variable in the config json. It follows a json-like structure, so you can use `.` to separate sub-objects e.g. `myextension.buttons_to_add.play`.
  - `description`, a description of the configuration parameter
  - `default`, a default value used to populate the tag in the configurator UI, if no value is found in config. Note that this is more of a hint to the user than anything functional - since it's only set in the yaml file, the javascript implementing the nbextension in question might actually use a different default, depending on the implementation.
  - `input_type`, controls the type of html tag used to render the parameter in the configurator UI. Valid values include `text`, `textarea`, `checkbox`, [html5 input tags such as `number`, `url`, `color`, ...], plus a final type of `list`
  - `list_element`, a dictionary with the same `default` and `input_type` keys as a `Parameters` entry, used to render each element of the list for parameters with input_type `list`
  - finally, extras such as `min`, `step` and `max` may be used by `number` tags for validation
- `tags`, a list of string tags describing the nbextension, to allow for filtering

**Example:**
```
Type: Jupyter Notebook Extension
Name: Limit Output
Section: notebook
Description: This nbextension limits the number of characters
that can be printed below a codecell
tags:
- usability
- limit
- output
Link: readme.md
Icon: icon.png
Main: main.js
Compatibility: 4.x
Parameters:
- name: limit_output
  description: Number of characters to limit output to
  input_type: number
  default: 10000
  step: 1
  min: 0
- name: limit_output_message
  description: Message to append when output is limited
  input_type: text
  default: '**OUTPUT MUTED**'
```

## APPENDIX 2 SPARK INSTALL INSTRUCTIONS - WINDOWS

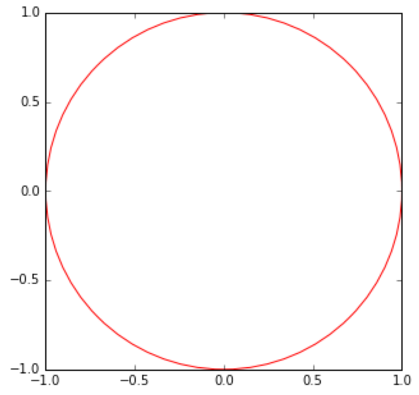
Instructions tested with Windows 10 64-bit. It is highly recommend that you use Mac OS X or Linux for this course, these instructions are only for people who cannot run Mac OS X or Linux on their computer.

### INSTALL AND SETUP

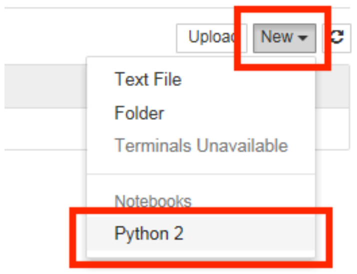
Spark provides APIs in Scala, Java, Python (PySpark) and R. We use PySpark and Jupyter, previously known as IPython Notebook, as the development environment. There are many articles online that talk about Jupyter and what a great tool it is, so we won't introduce it in details here.
This Guide Assumes you already have Anaconda and Gnu On Windows installed. See https://mas-dse.github.io/startup/anaconda-windows-install/
1. Go to http://www.java.com and install Java 7+.
2. Get Spark pre-built package from the downloads page of the Spark project website.
3. Open PowerShell by pressing ⊞ Win-R, typing "powershell" in Run dialog box and clicking "OK". Change your working directory to where you downloaded the Spark package.
4. Type the commands in red to uncompress the Spark download. Alternatively, you can use any other software of your preference to uncompress.
```
> gzip -d spark-2.1.0-bin-hadoop2.7.tgz
> tar xvf spark-2.1.0-bin-hadoop2.7.tar
```
5. Type the commands in red to move Spark to the c:\opt\spark\ directory.
```
> mkdir C:\opt\
> move spark-2.1.0-bin-hadoop2.7 C:\opt\spark\
```
6. Type the commands in red to download winutils.exe for Spark.
```
> cd C:\opt\spark\bin\
> curl -k -L -o winutils.exe
https://github.com/steveloughran/winutils/blob/master/hadoo
p-2.6.0/bin/winutils.exe?raw=true
```
7. Create an environment variable with variable name = SPARK_HOME and variable value = C:/opt/spark. This link provides a good description of how to set environment variable in windows
8. Type the commands in red to create a temporary directory.
```
> mkdir ~/Documents/jupyter-temp/
> cd ~/Documents/jupyter-temp/
```
9. Type the commands in red to install, configure and run Jupyter Notebook. Jupyter Notebook will launch using your default web browser.
```
> conda install jupyter -y
> ipython kernelspec install-self
> jupyter notebook
```

### FIRST SPARK APPLICATION

In our first Spark application, we will run a Monte Carlo experiment to find an estimate for $\pi$.
Here is how we are going to do it. The figure bellow shows a circle with radius $r = 1$ inscribed within a 2×2 square. The ratio between the area of the circle and the area of the square is $\frac{\pi}{4}$. If we sample enough points in the square, we will have approximately $\rho = \frac{\pi}{4}$ of these points that lie inside the circle. So we can estimate $\pi$ as $4 \rho$.



1. Create a new Notebook by selecting **Python 2** from the **New** drop down list at the right of the page.



2. First we will create the Spark Context. Copy and paste the red text into the first cell then click the (run cell) button:
```
import os
import sys

import findspark
findspark.init()

from pyspark import SparkContext
```

```
sc = SparkContext(master="local[4]")
```

3. Next, we draw a sufficient amount of points inside the square. Copy and paste the red text into the next cell then click the  (run cell) button:

```
import numpy as np

TOTAL = 1000000
dots = sc.parallelize([2.0 * np.random.random(2) - 1.0 for
i in range(TOTAL)]).cache()
print("Number of random points:", dots.count())

stats = dots.stats()
print('Mean:', stats.mean())
print('stdev:', stats.stdev())
```

Output:

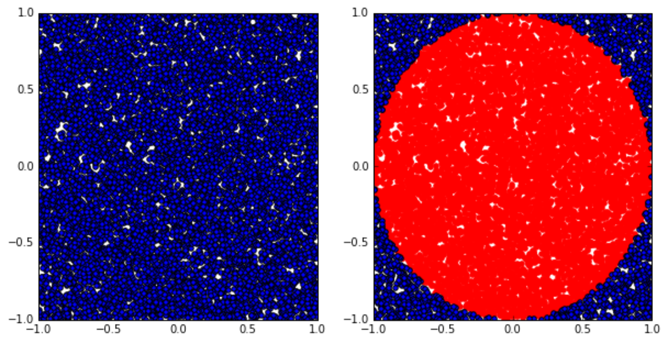
```
('Number of random points:', 1000000)
('Mean:', array([-0.0004401 , 0.00052725]))
('stdev:', array([ 0.57720696, 0.57773085]))
```

4. We can sample a small fraction of these points and visualize them. Copy and paste the red text into the next cell then click the  (run cell) button:

```
%matplotlib inline
from operator import itemgetter
from matplotlib import pyplot as plt

plt.figure(figsize = (10, 5))

# Plot 1
plt.subplot(1, 2, 1)
plt.xlim((-1.0, 1.0))
plt.ylim((-1.0, 1.0))

sample = dots.sample(False, 0.01)
X = sample.map(itemgetter(0)).collect()
Y = sample.map(itemgetter(1)).collect()
plt.scatter(X, Y)

# Plot 2
plt.subplot(1, 2, 2)
plt.xlim((-1.0, 1.0))
plt.ylim((-1.0, 1.0))

inCircle = lambda v: np.linalg.norm(v) <= 1.0
dotsIn = sample.filter(inCircle).cache()
dotsOut = sample.filter(lambda v: not inCircle(v)).cache()

# inside circle
Xin = dotsIn.map(itemgetter(0)).collect()
Yin = dotsIn.map(itemgetter(1)).collect()
plt.scatter(Xin, Yin, color = 'r')

# outside circle
Xout = dotsOut.map(itemgetter(0)).collect()
Yout = dotsOut.map(itemgetter(1)).collect()
plt.scatter(Xout, Yout)
```

Output:

```
<matplotlib.collections.PathCollection at 0x17a78780>
```



5. Finally, let's compute the estimated value of $\pi$. Copy and paste the red text into the next cell then click the  (run cell) button:

```
pi = 4.0 * (dots.filter(inCircle).count() / float(TOTAL))
print("The estimation of \pi is:", pi)
```

Output:

```
('The estimation of \\pi is:', 3.142204)
```

**Next Steps**

- Spark Programming Guide
- Example Spark Programs

**References**

- Spark official documents
- Example Python Spark programs on the Spark Github repository

# APPENDIX 3 GOW - THE LIGHTWEIGHT ALTERNATIVE TO CYGWIN

## INTRODUCTION

Gow (Gnu On Windows) is the lightweight alternative to Cygwin. It uses a convenient NSIS installerthat installs over 100 extremely useful open source UNIX applications compiled as native win32 binaries. It is designed to be as small as possible, about 18 MB, as opposed to Cygwin which can run well over 100 MB depending upon options.
Here are a couple quotes from happy Gow users:
"Gow is one of the few things that makes Windows bearable/usable"
"I use Gow constantly. It's awesome."
"I just wanted to let you know that the GOW Suite is simply great - it is far lighter than the Cygwin tool, and is extremely useful. "

## FEATURES AND BENEFITS

- *Ultra light*: Small, light subset (about 18 MB) of very useful UNIX binaries that do not have decent installers (until now!).
- *Shell window from any directory*: Adds a Windows Explorer shell window (screenshot) so that you can right-click on any directory and open a command (cmd.exe) window from that directory.
- *Simple install/remove*: Easy to install and remove, all files contained in a single directory in a standard C:\Program Files path.
- *Included in PATH*: All binaries are conveniently installed into the Windows PATH so they are accessible from a command-line window.
- *Stable binaries*: All commands are stable and tested.

## WIN32 UTILITIES OVERVIEW

Below are just a few of the 100+ applications found in Gow.

- *Shell scripting*: bash, zsh
- *Compression*: gzip, zip, bzip2, compress
- *SSH*: putty, psftp, pscp, pageant, plink
- *Download/upload*: cURL, wget
- *FTP*: NcFTP
- *Editing*: vim, nano
- *Text search/view*: grep, agrep, less, cat, tail, head
- *File system*: mv, cp, du, ls, pwd, rmdir, whereis
- *Development*: make, diff, diff3, sleep, cvs, dos2unix, unix2dos