EDAN95

Applied Machine Learning

http://cs.lth.se/edan95/

Lecture 10: Spark / GPU

Marcus Klang

With contributions from:
Peter Exner

Lund University

Marcus.Klang@cs.lth.se
http://cs.lth.se/marcus_klang/

Overview

- Big Data frameworks:
  Hadoop, Apache Spark, Apache Flink, Dask
- (Py)Spark: RDD, SQL
- GPU computation in Tensorflow/Keras

The structure spectrum

Structured	**Semi-structured**	**Unstructured**
Relational Databases	HTML	Plain text
Parquet	XML	Generic media
Formatted Messages	JSON	

Assignment 5

- Use tools for reading one large corpus: Wikipedia
- Process the corpus:
  Transform, Extract, Count
- Cluster top 10,000 words from Wikipedia into 100 clusters.
- Use Glove 6B embeddings to represent and transform the words into 2D space for visualization.

Big Data

4,087,467,519
Internet Users in the world

1,935,671,223
Total number of Websites

204,424,528,207
Emails sent today

5,078,772,978
Google searches today

4,807,522
Blog posts written today

592,354,253
Tweets sent today

5,470,544,674
Videos viewed today on YouTube

63,131,725
Photos uploaded today on Instagram

104,349,544
Tumblr posts today

http://www.internetlivestats.com/ (2018-12-03)

Big Data

4,087,467,519
Internet Users in the world

1,935,671,223
Total number of Websites

204,424,528,207
Emails sent today

How do you store big data?

5,078,772,978
Google searches today

4,807,522
Blog posts written today

592,354,253
Tweets sent today

How do you compute big data?

5,470,544,674
Videos viewed today on YouTube

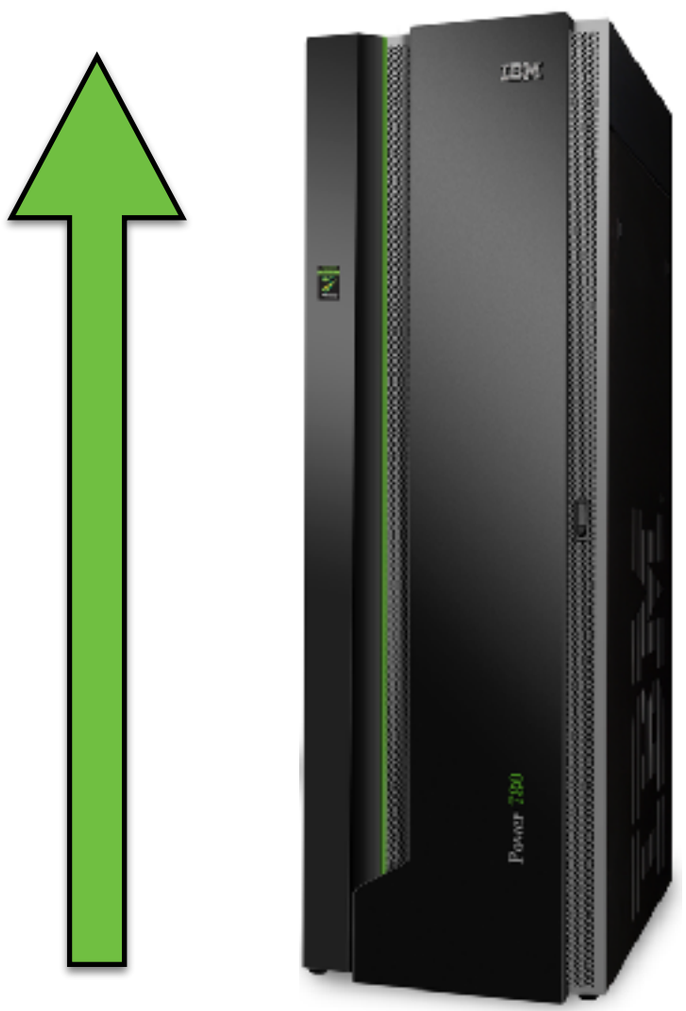
63,131,725
Photos uploaded today on Instagram

104,349,544
Tumblr posts today

http://www.internetlivestats.com/ (2018-12-03)

The Big Data solution

Scale up
(fewer, larger servers)



VS

Scale out
(more, smaller servers)

Frameworks

- Apache Hadoop
- Apache Spark
- Apache Flink
- Dask
- Many more…

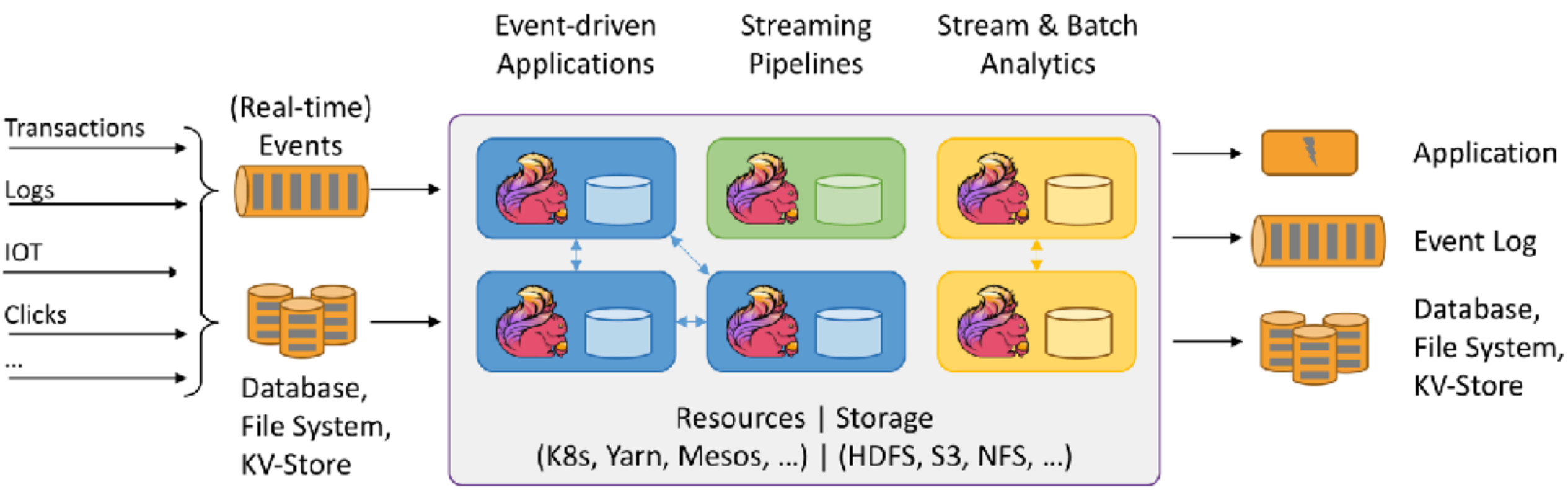
Flink

. . .

Hadoop

- Backbone of all Hadoop ecosystem projects
- Infrastructure for data storage
  `Hadoop Distributed Filesystem (HDFS)`
- Infrastructure for serialization (Writable)
  `Hadoop Common`
- MapReduce Implementation
  `Hadoop MapReduce`
- YARN - System for managing distributed application
  (Spark can run on YARN)

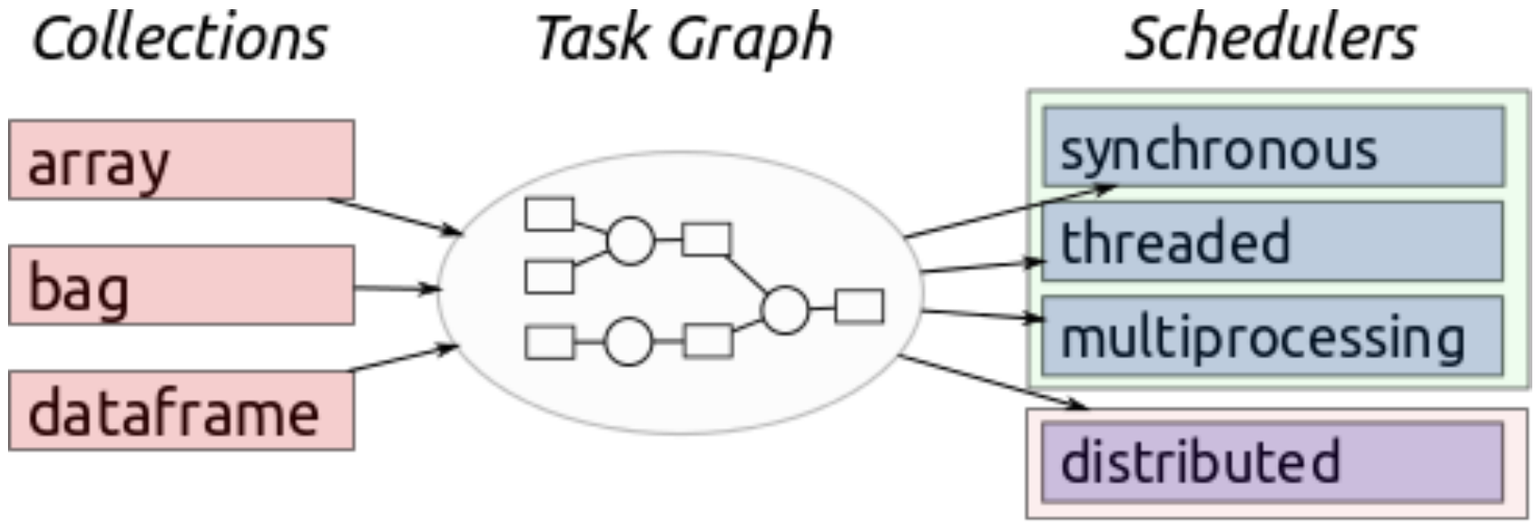
Apache Flink



Source: https://flink.apache.org/

- Focused on streaming (Java/Scala)

Dask



Source: https://docs.dask.org/en/latest/

- Native Python
- Numpy/Pandas integration (faster for specific workloads)
- Flexible, rich and simple API: huge amount of features in short time.
- During testing for this course: not yet ready for prime time.
  (1.0 released 28 november 2018), maybe next time.

- Answer to Hadoops MapReduce
- Prioritizes memory over disk for performance.
- Functional API which constructs a DAG (Directed Acyclic Graph) from a pipeline description.
  - Maximizes parallelism when possible by fusing multiple stages together which can be run in sequence
- Available in common cluster solutions such as:
  Cloudera, Hortonworks, Amazon EMR, Google Cloud: Cloud Dataproc
- Supports many languages: Python/Java/Scala/R

Development Environment

- Jupyter Notebook / Lab: https://jupyterlab.readthedocs.io/en/stable/

  Python
  Scala with Spark is available via
  Apache Toree https://toree.apache.org/

- Apache Zeppelin: https://zeppelin.apache.org/

  A full solution with Spark, with integration between many languages such as Python, Scala, R, SQL, and more.

Programming with Python Spark (pySpark)

- We will use Python's interface to Spark called **pySpark**

- A **driver program** accesses the Spark environment through a **SparkContext** object.

- The key concept in Spark are datasets called **RDD**s

- We load our data into RDDs and perform some **operations**

Functional Primer

- Defintions:

```
data    = [1,3,5,7,9]
f(x)    = x*x
p(x)    = x > 5
g(x,y) = x+y
(commutative: g(x,y) == g(y,x) and g(g(z,y),x) == g(x,g(y,z))
```

- **map**(f, data) = [f(1), f(3), f(5), f(7), f(9)] = [1, 9, 25, 49, 81]
- **filter**(p, data) = [for all x where p(x) is True] = [7,9]
- **reduce**(g, data) = g(1, g(9, g(g(3, 5),7))
  = 1+(9+((3+5)+7))
  = 25

First Program!

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

lines = sc.textFile("README.md", 4)

lines.count()

pythonLines = lines.filter(lambda line : "Python"
in line)

pythonLines.first()
```

RDD

- **R**esilient **D**istributed **D**ateset (**RDD**)
- Contains **distributed data**, spread across **partitions**
- Enables **operations** to be **performed in parallel**
- Are **immutable**
- **Recomputes data** in case of loss

Creating RDDs

Three ways:

- Loading an external dataset

```
>>> lines = sc.textFile("README.md", 4)
```

- Distributing a collection of objects, e.g. a list

```
>>> lines = sc.parallelize([1, 2, 3])
```

- Transforming an existing RDD

```
>>> pyLines = lines.filter(lambda line : "Python" in
line)
```

Operations on RDDs

- Transformations
  - creates a new RDD from a previous one
  - E.g. map()
- Actions
  - computes a result based on an existing RDD
  - E.g. count()

Functional programming with Python

- Many transformations and some actions expect a function

- These can be defined functions for complex operations

- For simple functions, lambda expressions are convenient

```
>>> lambda line: "Python" in line
```

map()

- Reads one element at a time
- Takes one value, creates a new value

```
>>> rdd = sc.parallelize([1, 2, 3, 4])

>>> rdd.map(lambda x: x * 2)

Out[1]:  [2, 4, 6, 8]
```

filter()

- Reads one element at a time
- Evaluates each element
- Returns the elements that pass the filter()

```
>>> rdd = sc.parallelize([1, 2, 3, 4])

>>> rdd.filter(lambda x: x % 2 == 0)

Out[1]:  [2, 4]
```

flatMap()

- Produce multiple elements for each input element

```
>>> rdd = sc.parallelize([1, 2, 3])
```

```
>>> rdd.map(lambda x: [x, x * 2])
Out[1]:  [[1, 2], [2, 4], [3, 6]]
```

```
>>> rdd.flatMap(lambda x: [x, x * 2])
Out[2]:  [1, 2, 2, 4, 3, 6]
```

Transformations are lazy!

- A transformed RDD is only executed when actions run on it

```
>>> pyLines = lines.filter(lambda line: "Python" in
line)

>>> pyLines.first()
```

- No need for Spark to load all the lines containing “Python” into memory!

Actions

- Actions cause transformations to be executed on RDDs
- Actions return results to either the driver program or to an external storage
- RDDs are recomputed for every action
- RDDs can be cached for reuse, **`rdd.persist()`**

count()

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.count()
Out[1]:  4
```

collect()

- `collect()` retrieves the entire RDD
- Useful for inspecting small datasets locally and for unit tests

Results must fit in memory on the local machine!

```
>>> rdd = sc.parallelize([1, 2, 3])

>>> rdd.collect()

[1, 2, 3]
```

take(), first(), top(), takeSample()

- **`take(n)`** returns n elements from an RDD
- **`take(n)`** may be biased! Suitable for testing, debugging
- **`takeSample()`** - more suitable for taking a sample
- Use **`takeOrdered(), top(n)`** for ordered return

takeOrdered()

```
>>> rdd = sc.parallelize([5, 1, 3, 2])

>>> rdd.takeOrdered(4)

Out[1]:  [1, 2, 3, 5]

>>> rdd.takeOrdered(4, lambda n: -n)

Out[2]:  [5, 3, 2, 1]
```

reduce()

- Takes two elements of the same type and returns one new element

```
>>> rdd = sc.parallelize([1, 2, 3])

>>> rdd.reduce(lambda x, y: x * y)

Out[1]:  6
```

Building a pipeline of RDD operations

```
>>> lines = sc.textFile("README.md")

>>> lines.map(…).filter(…).count(…)
```

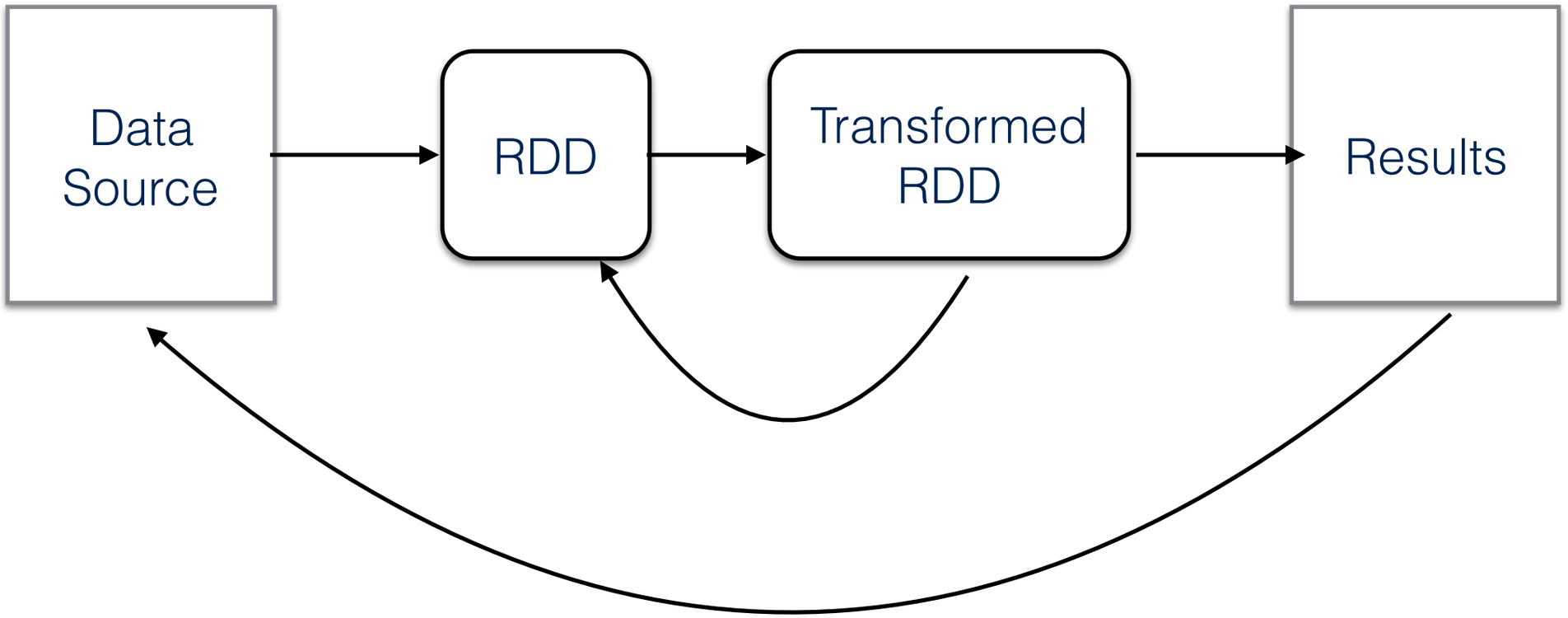
```
>>> (lines
      .map(…)
      .filter(…)
      .count(…))
```

RDD workflow

`parallelize()`
`textFile()`
…

`map()`
`filter()`
…

`collect()`
`count()`
…



AHA! moment of insight!

Working with key/value pairs

- Many entities can naturally be represented as keys
  - e.g. event time, customer id etc…
- In Python, tuples are used to form key/value pairs
  - E.g. `("fox", 1), ("bear", 3)`

Pair RDDs

- RDDs containing key/value pairs are called Pair RDDs and are composed of tuples:

```
>>> pairs = sc.parallelize([("a", 2), ("b", 6)])
Equivalent to:
>>> pairs = sc.parallelize(list({"a":2, "b":6}.items()))
```

- Spark offers special operations on Pair RDDs
  - examples: reduceByKey, sortByKey, joins
  - Require passing of functions that operate on tuples
- Pair RDDs support same functions as regular RDDs

reduceByKey()

- Runs several parallel reduce operations - one for each key in the dataset
- Each reduce combines values having the same key
- reduceByKey() is not an action like reduce()
- Returns a new RDD, not a value!

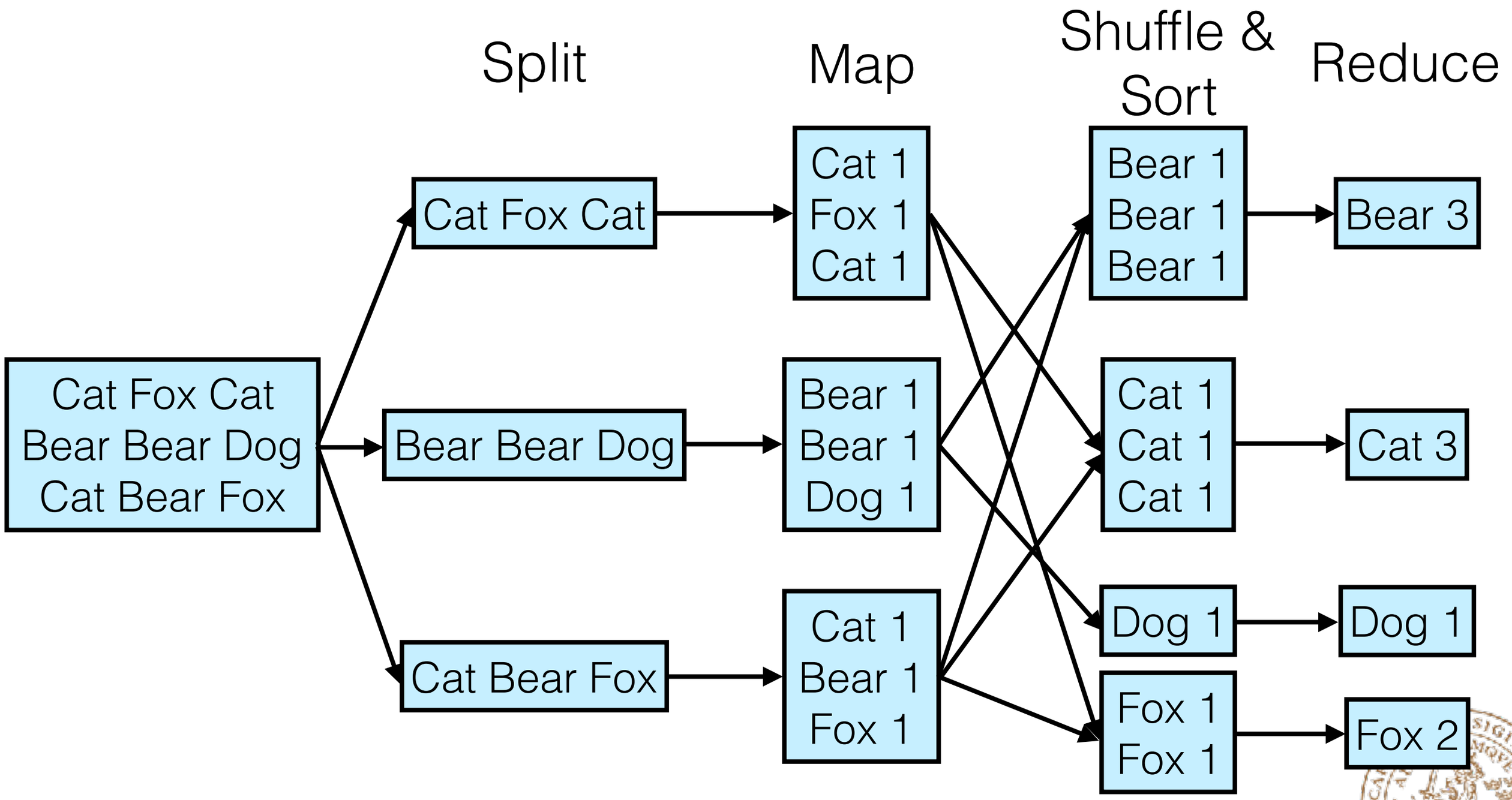
reduceByKey()

```
>>> rdd = sc.parallelize([("a", 2), ("b", 4), ("b", 6)])
```

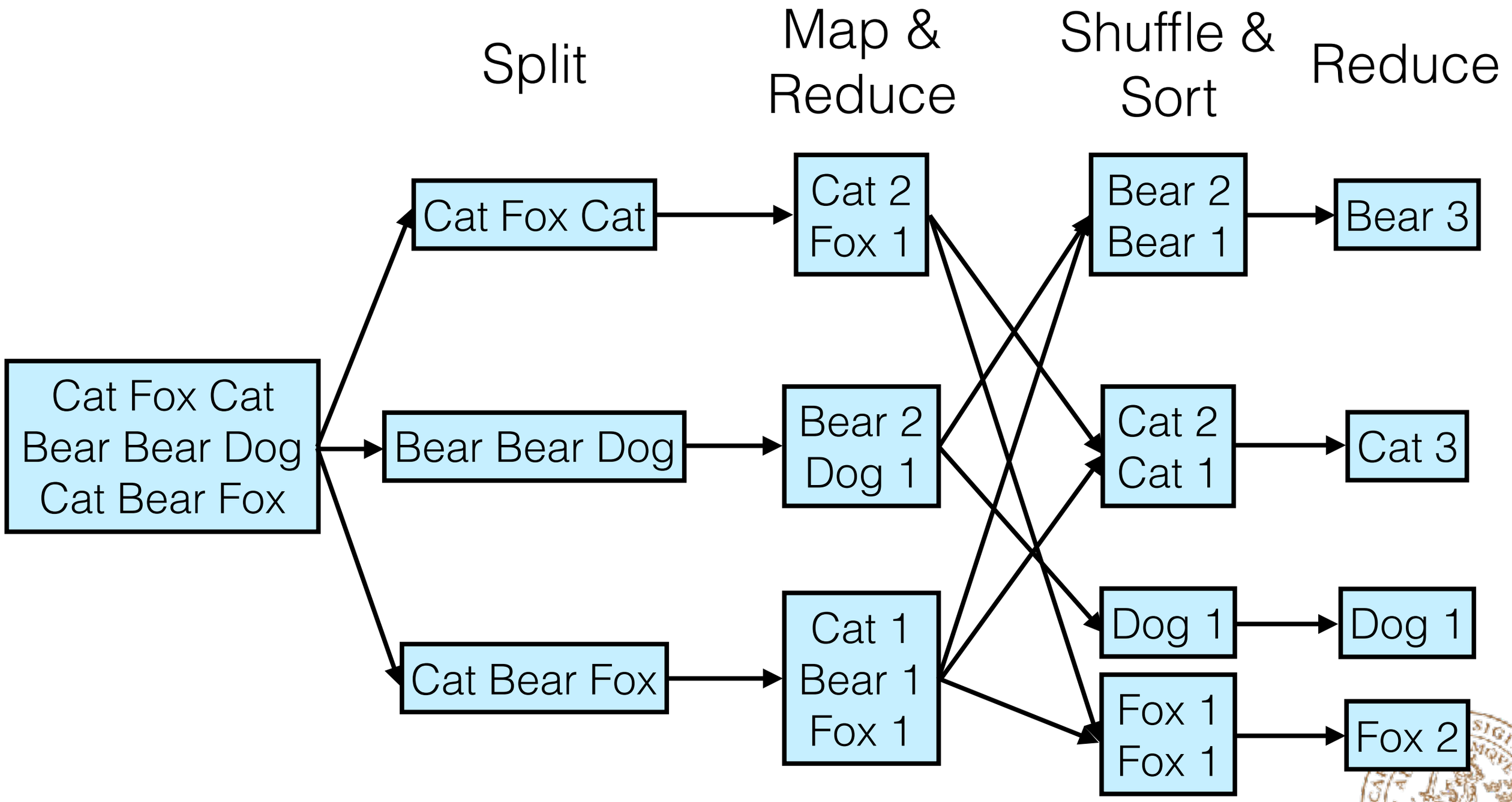
```
>>> rdd.reduceByKey(lambda x, y: x + y)
```

```
Out[1]:  [("a", 2), ("b", 10)]
```

MapReduce Word Count



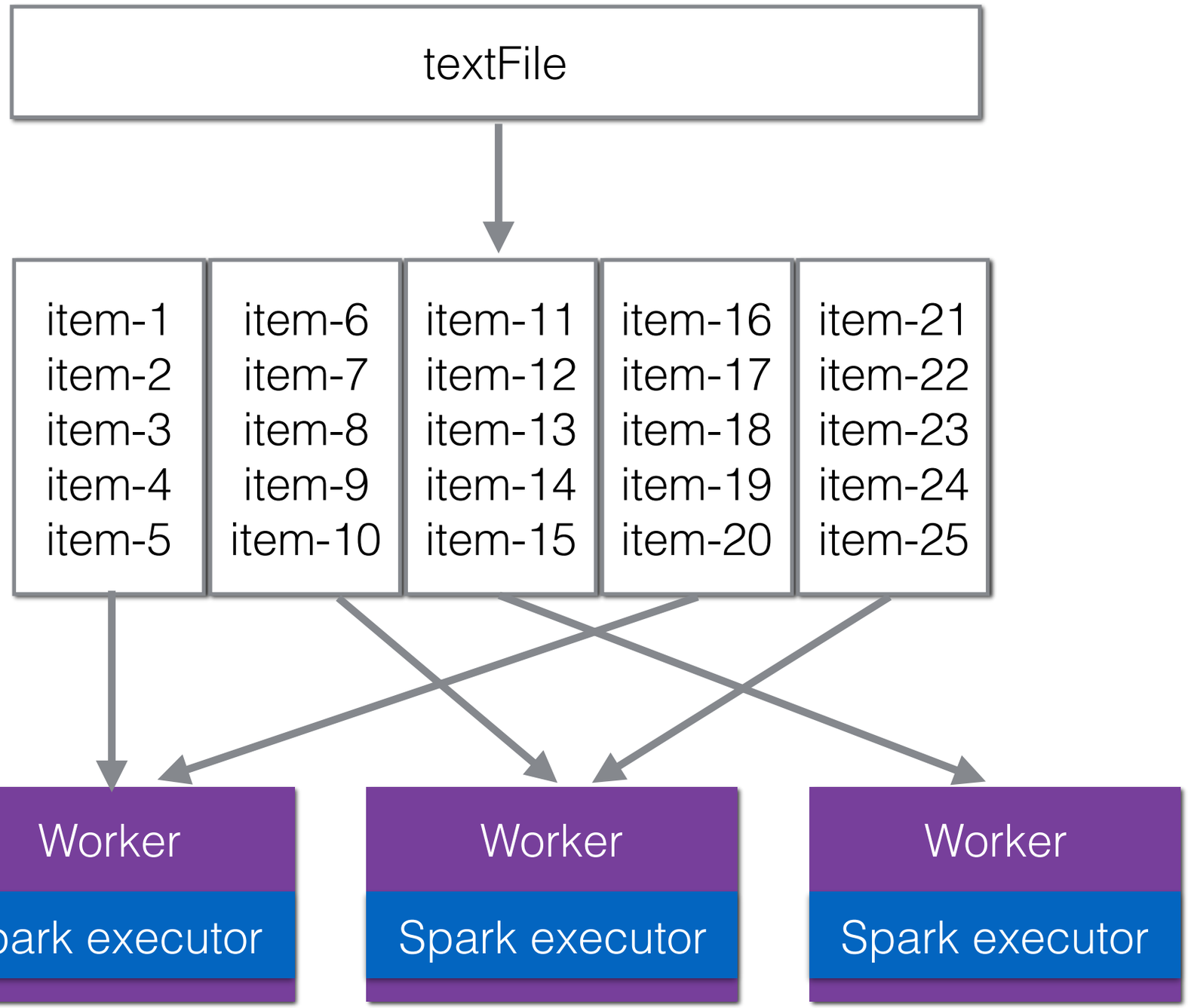
MapReduce Word Count



Partitions

- Every RDD is split up into a number of partitions
- Parallelism is determined by the number of partitions
- rdd.getNumberOfPartitions()

RDD partitions



Setting the number of partitions

- Some operators accept a second parameter

```
>>> sc.textFile(path, 8)
```

- repartition() may be used to create a new set of partitions

- useful when you the expected key distribution is known before-hand

Data partitioning

- use partitionBy on large datasets to partition by hash function

- keys with same hash value will be placed on the same node

- Don't forget to persist the partitioned RDD!

Partitions

- change the number of partitions of records via a shuffle

```
>>> rdd.repartition(numPartitions)
```

- change the number of partitions with optional shuffling

```
>>> rdd.coalesce(numPartitions,
shuffle=False)
```

Partitions

- Small amount of partitions:
  (low overhead, high throughput per core, low parallelism)

- Huge numbers of partitions:
  (high overhead, low throughput per core, high parallelism)

- Optimal number of partitions:
  (low overhead, high throughput per core, optimal parallelism)

Spark: mapPartitions

- Apply a single function over the entire partition and output the results

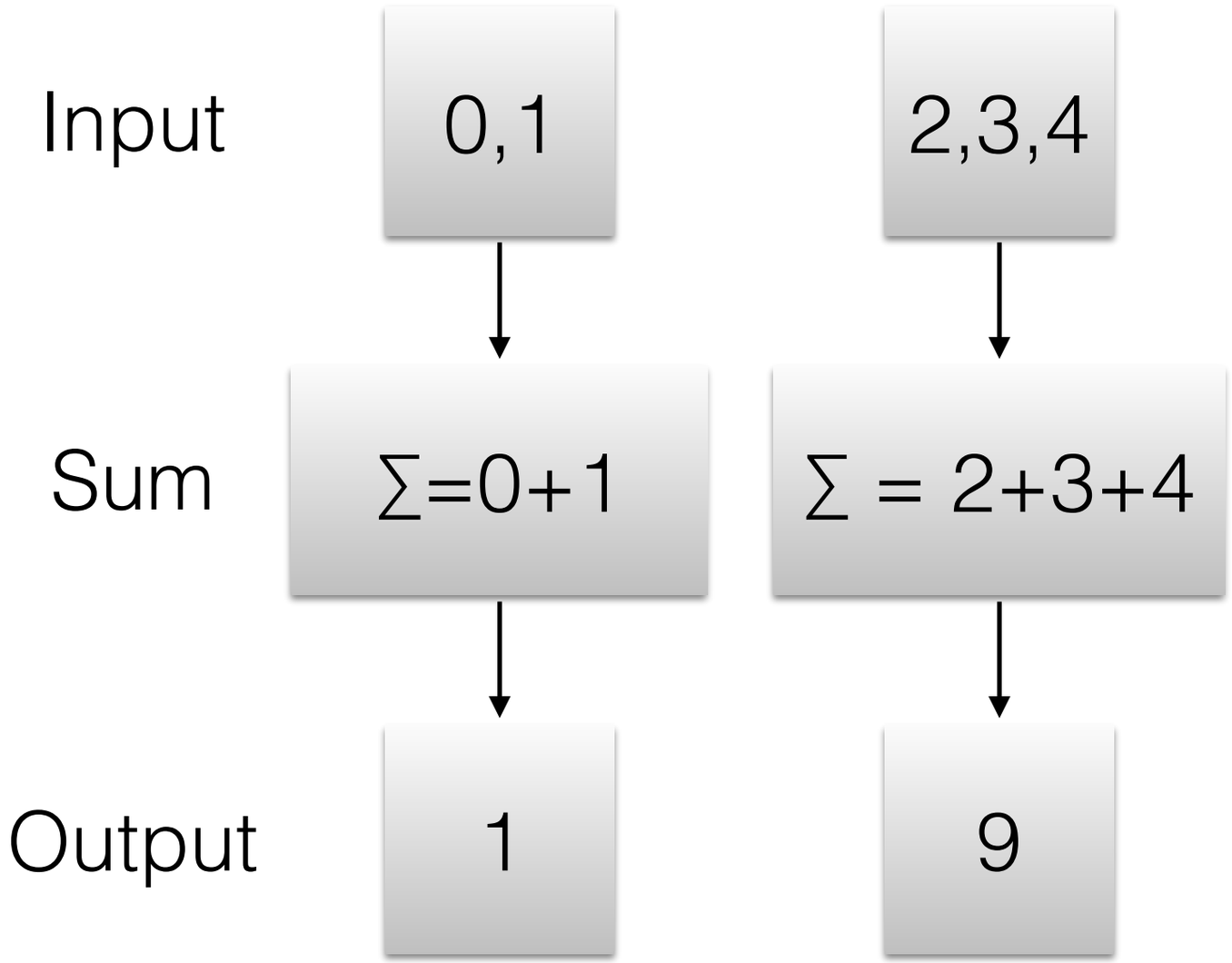
- ```
  >>> rdd = sc.parallelize([0,1,2,3,4],2)

  >>> rdd.mapPartitions(lambda partition: [sum(partition)]
  ).collect()

  [1,9]
  ```


  ```

Spark: mapPartitions



Spark: zipWithIndex

- Append a record index, starting from the first record in the partition to the last record in the last partition

- ```
  >>> rdd = sc.parallelize([4,3,2,1,0],2)
  ```

- ```
  >>> rdd.zipWithIndex().collect()
  [(4, 0), (3, 1), (2, 2), (1, 3), (0, 4)]
  ```

Spark: fold

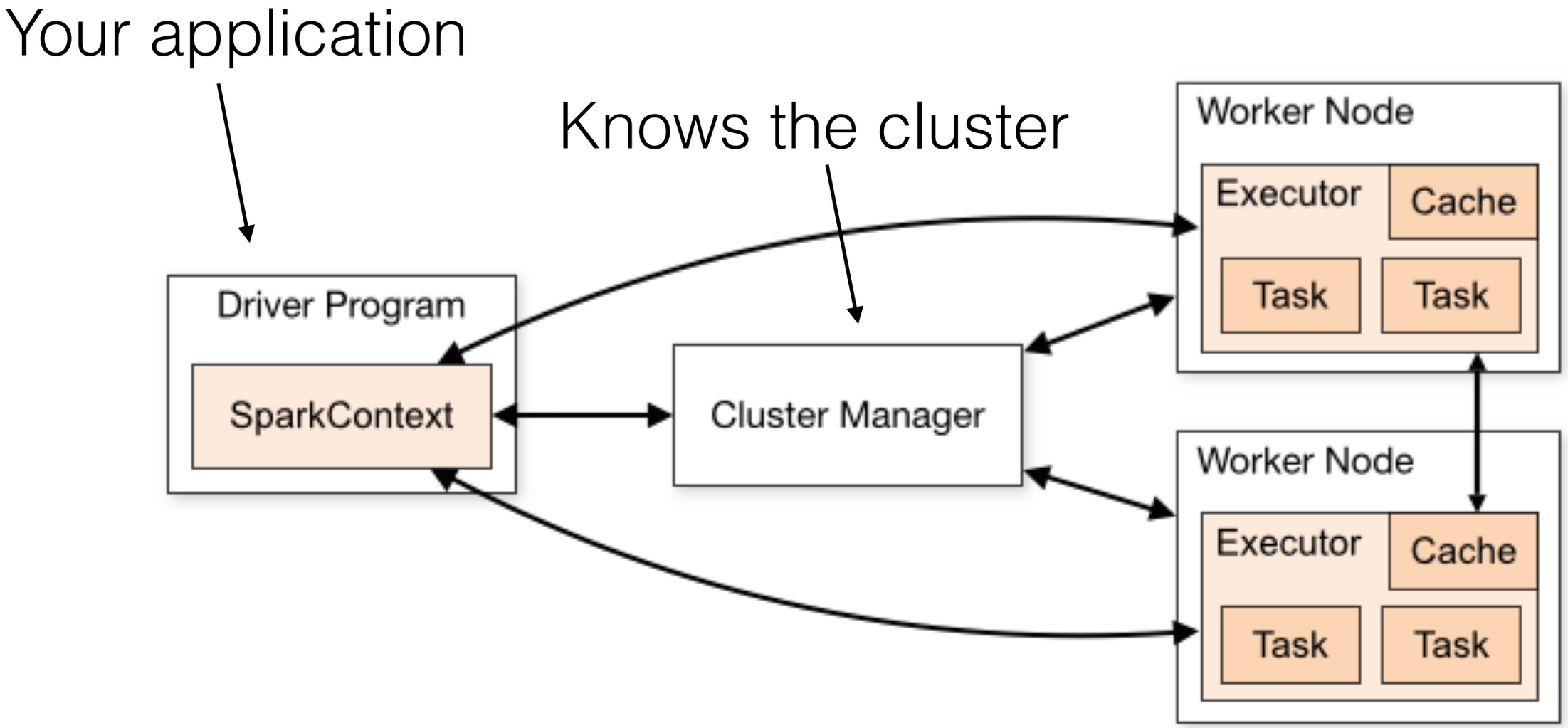
- Combine many values into a single one
  `fold(initial, op)`

- `>>> rdd = sc.parallelize([1, 2, 3, 4, 5])`

```
>>> from operator import add
>>> rdd.fold(0, add)
15
```

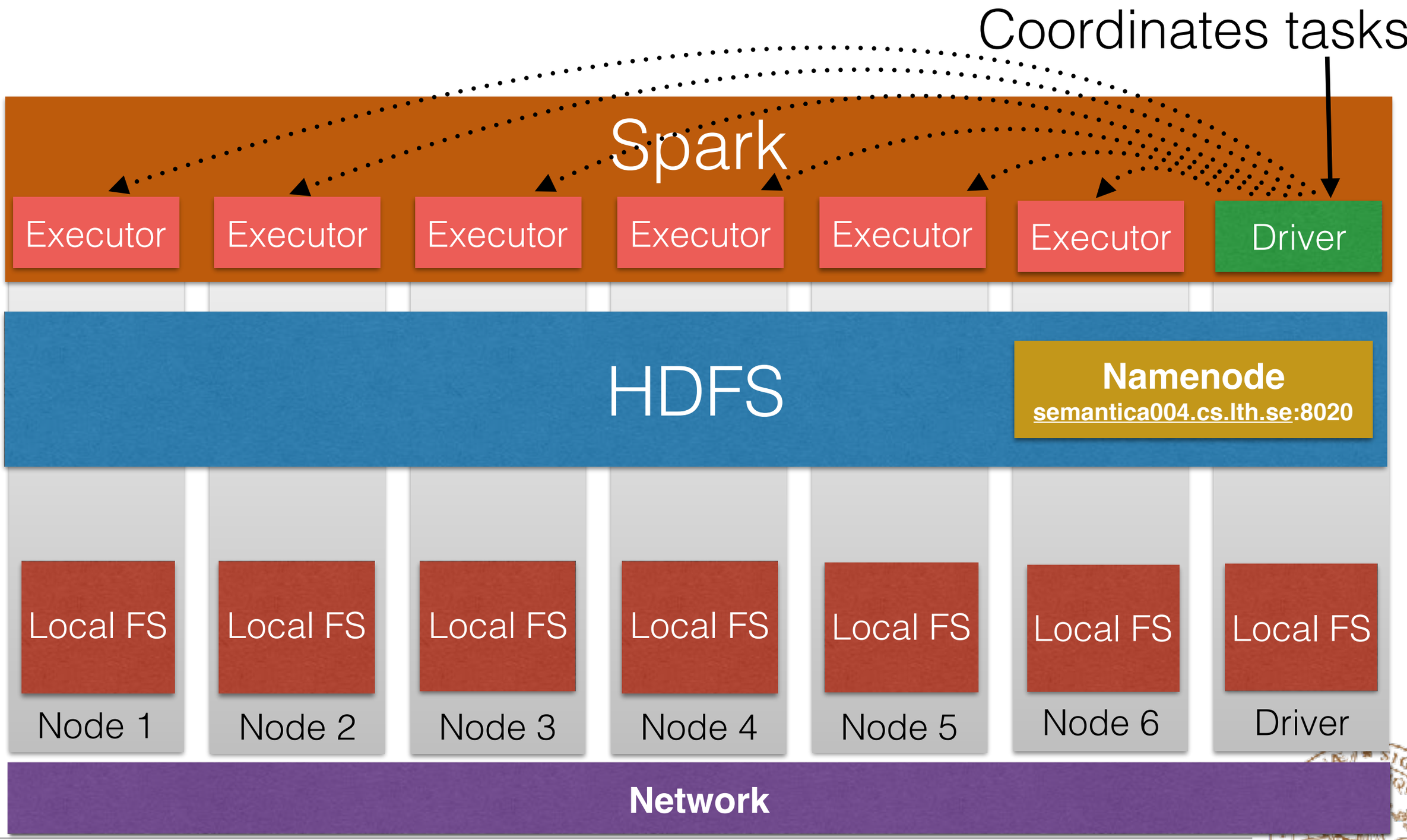
- **Ordering is not guaranteed**, assumes that the function is associative and commutative i.e f(x,y) = f(y,x) **and** f(f(x,y),z) = f(f(x,z),y)

Spark

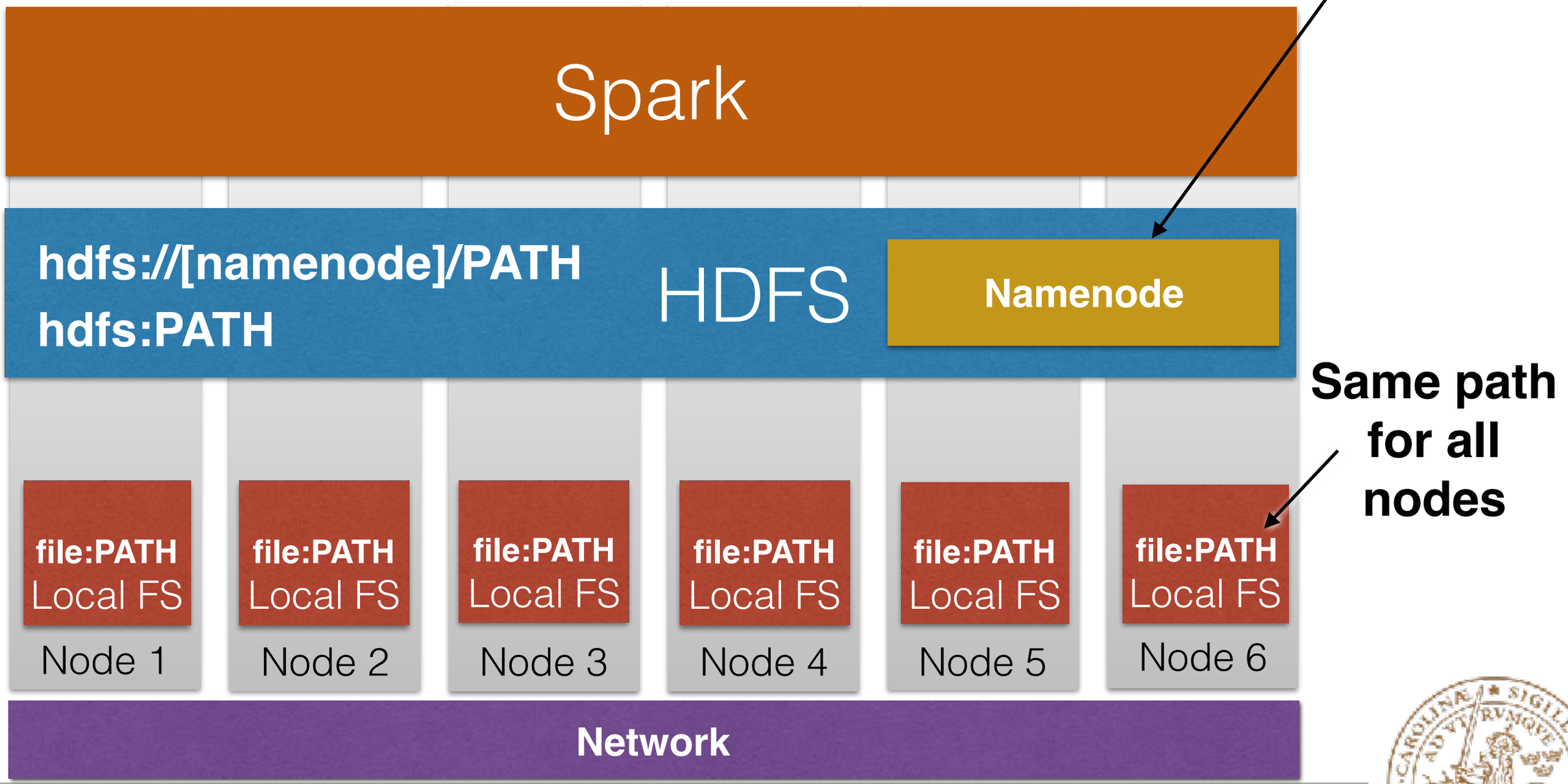


Source: https://spark.apache.org/docs/latest/cluster-overview.html

Spark: Cluster

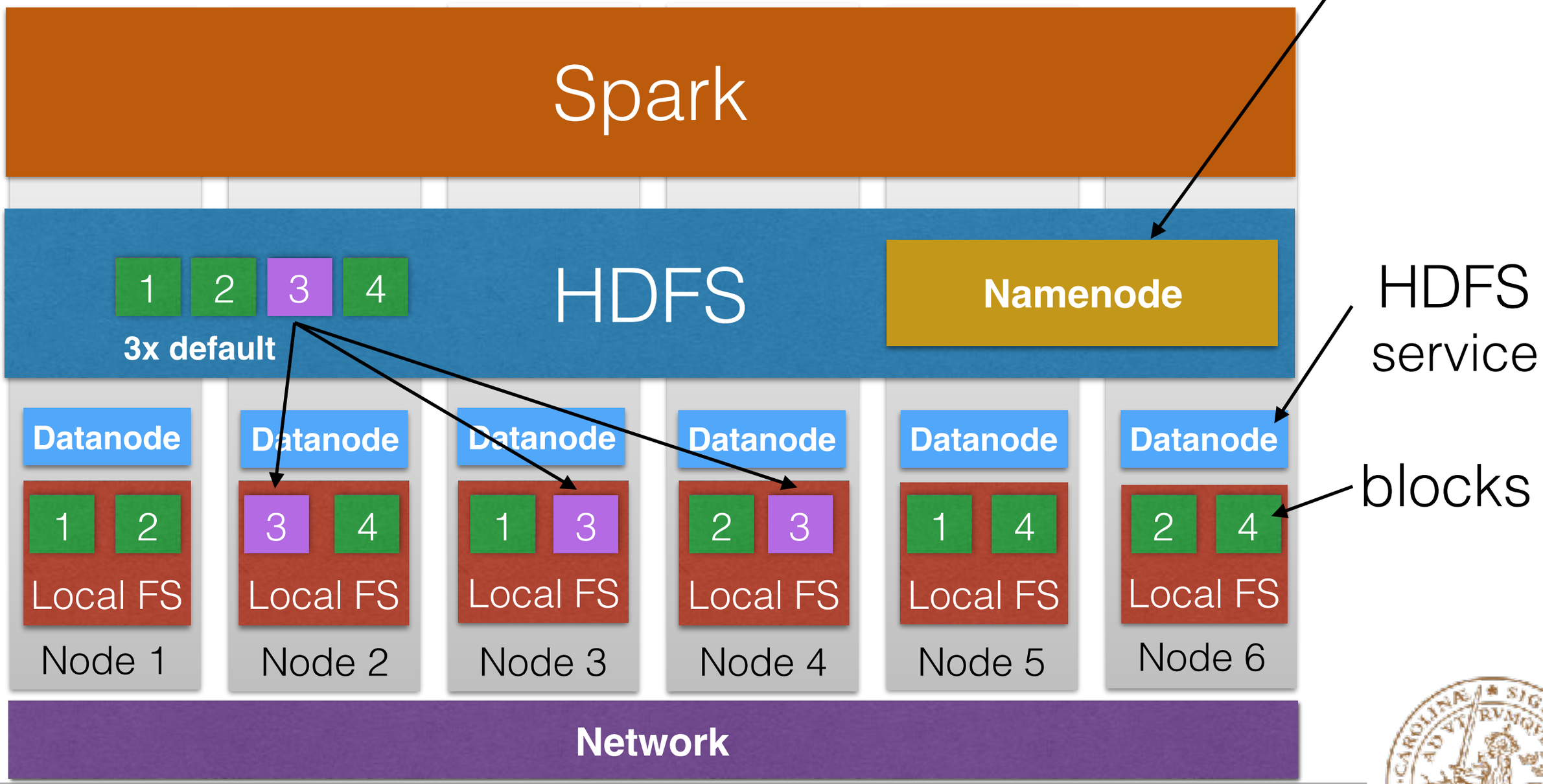


Spark: Cluster



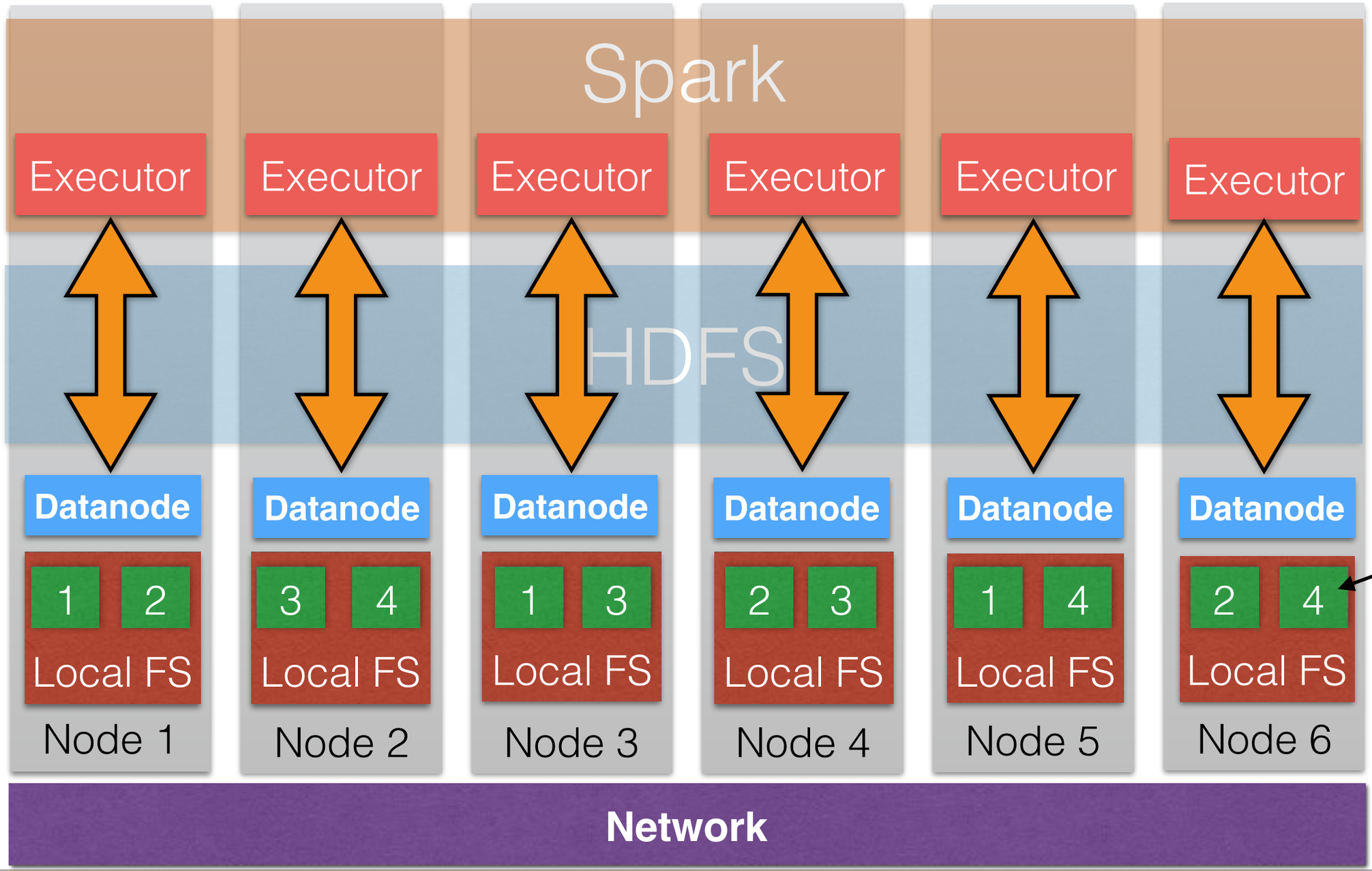
Spark: Cluster

Id Large binary block of data, might be partial files.

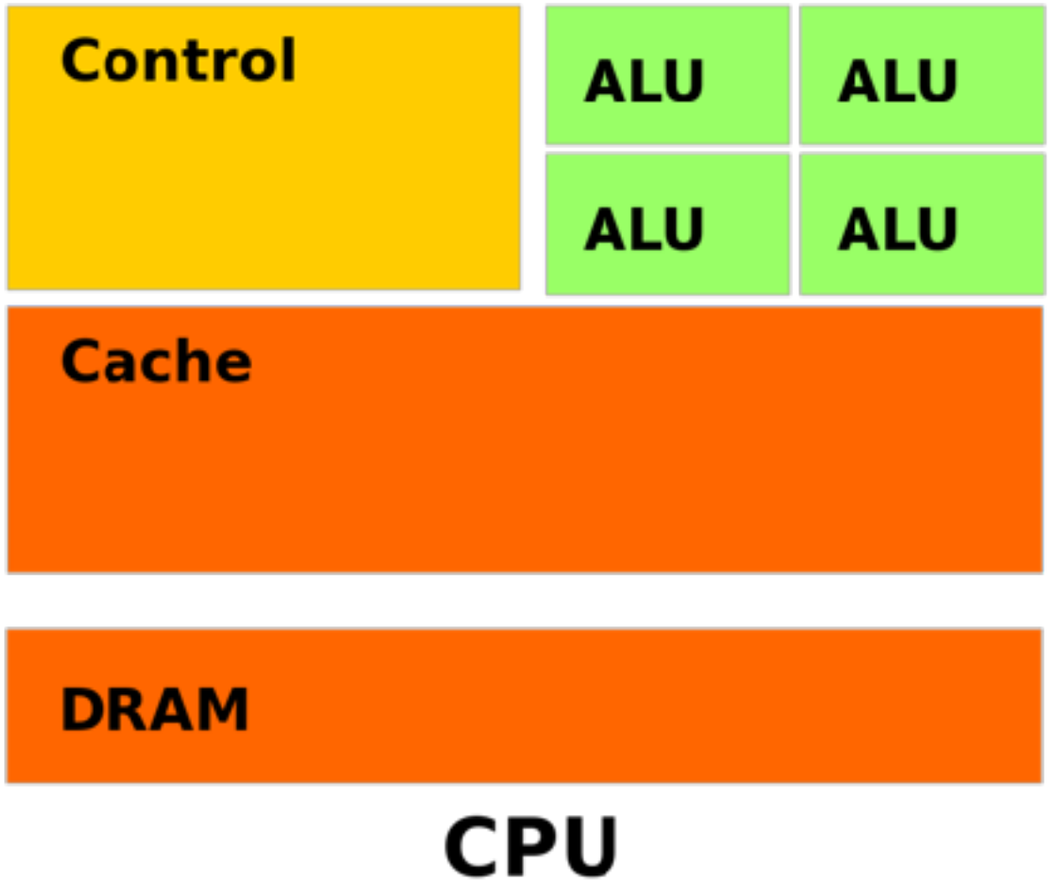


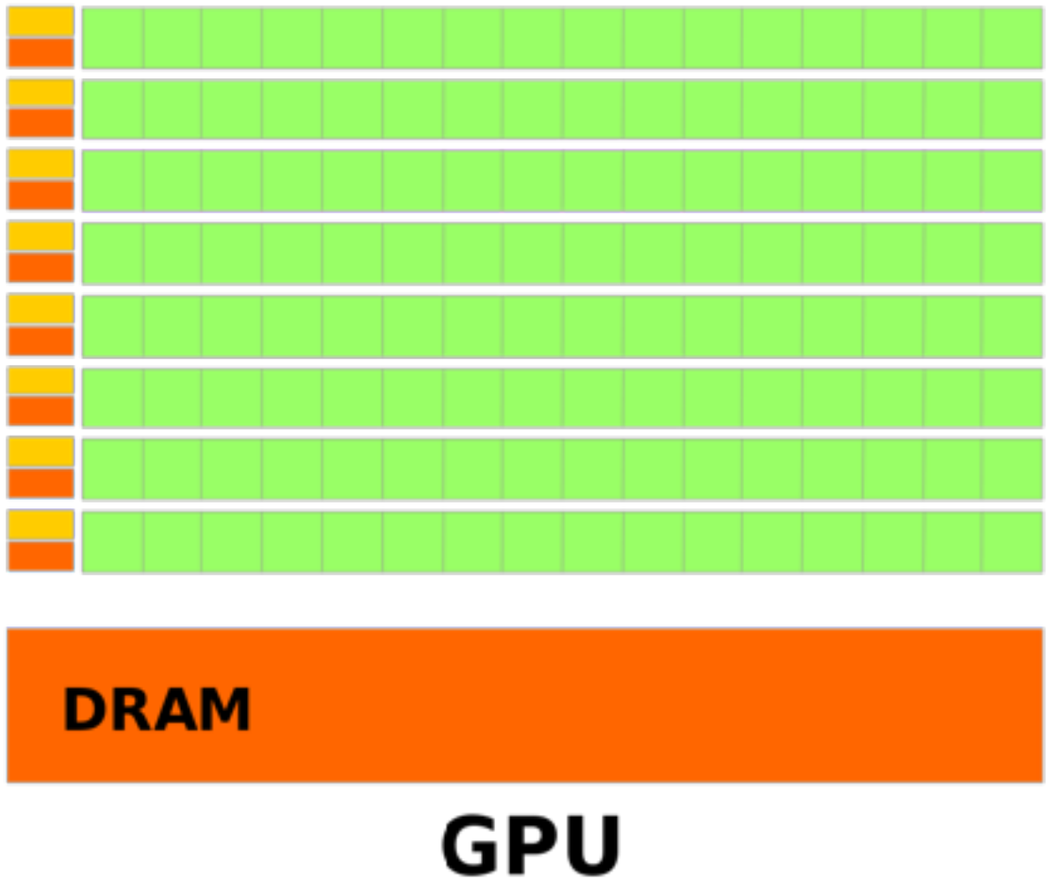
Spark: Cluster

Data is read locally if possible.



CPU vs GPU





Source: https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg

CPU vs GPU

- CPU: Quick, but does not scale.
- GPU: Scales, is optimized for heavy parallelism which yields many factors greater performance over CPU under optimal loads
- No magic: Performance is ultimately limited by communication and/or available computation units.
- Parallelism: Limited by overhead and/or the possibility for independent fused operations (many operations in sequence only dependent on input data)

Tensorflow: GPU

- Generally, install suitable drivers for your GPU
- Install the Tensorflow GPU version matching your drivers
- GPU support is transparent and will accelerate computations where possible.
- CPU is freed for other tasks when the GPU computes, such as data prefetching.

Tensorflow Code

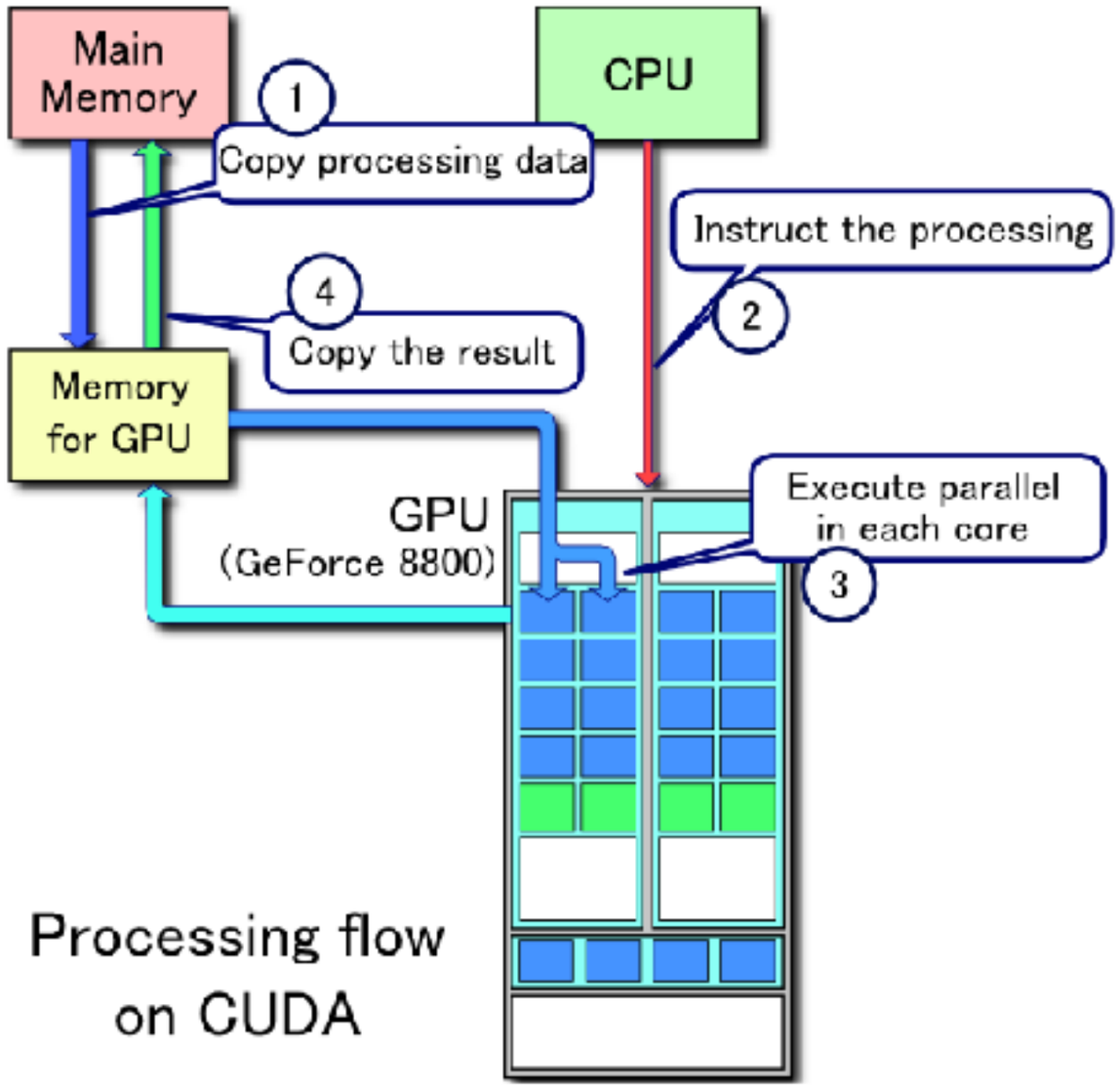
```
n = 8000
dtype = tf.float32
with tf.device(”/CPU:0"):
    matrix1 = tf.Variable(tf.ones((n, n), dtype=dtype))
    matrix2 = tf.Variable(tf.ones((n, n), dtype=dtype))
    product = tf.matmul(matrix1, matrix2)
```

```
n = 8000
dtype = tf.float32
with tf.device(”/GPU:0”):
    matrix1 = tf.Variable(tf.ones((n, n), dtype=dtype))
    matrix2 = tf.Variable(tf.ones((n, n), dtype=dtype))
    product = tf.matmul(matrix1, matrix2)
```

Keras == No difference (in general automatic)

CUDA API

Under the hood



Source: https://commons.wikimedia.org/wiki/File:CUDA_processing_flow_(En).PNG

Read more: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Benchmarking CPU vs GPU (cheaper hardware)

Time Taken	Intel i5 – 4210U 1.7GHz (2014)	GeForce Nvidia 1060 6GB GDDR5 (2016)	Speedup
Matrix Multiplication of 8000×8000	16 seconds at 63.36 G ops/sec	0.29 seconds at 3588.87 G ops/ sec	55,1x
Training of MNIST (60,000 images)	27 minutes 47 seconds	1 min 1 seconds	27.3x

Source: https://www.analyticsindiamag.com/deep-learning-tensorflow-benchmark-intel-i5-4210u-vs-geforce-nvidia-1060-6gb/

Benchmarking CPU vs GPU (expensive hardware)

Time Taken	Intel Xeon E5-2643 3.4 GHz (2014)	Geforce GTX TITAN X 12 GB GDDR5 (2015)	Speedup
Matrix Multiplication of 8000×8000	1.68 seconds at 609.85G ops/sec	0.21 seconds at 4791.65G ops/sec	8x
Training of MNIST (60,000 images)	5 min 4 sec	1 min 15 second	4x

Environment: Ubuntu 16.04, Tensorflow 1.12, CUDA 9.0

Method used:
https://www.analyticsindiamag.com/deep-learning-tensorflow-benchmark-intel-i5-4210u-vs-geforce-nvidia-1060-6gb/