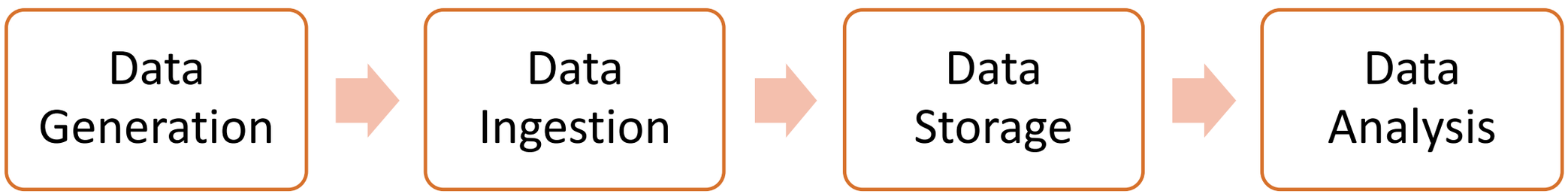
2110403 - Introduction to Data Science and Data Engineering

Big Data Processing with Apache Spark

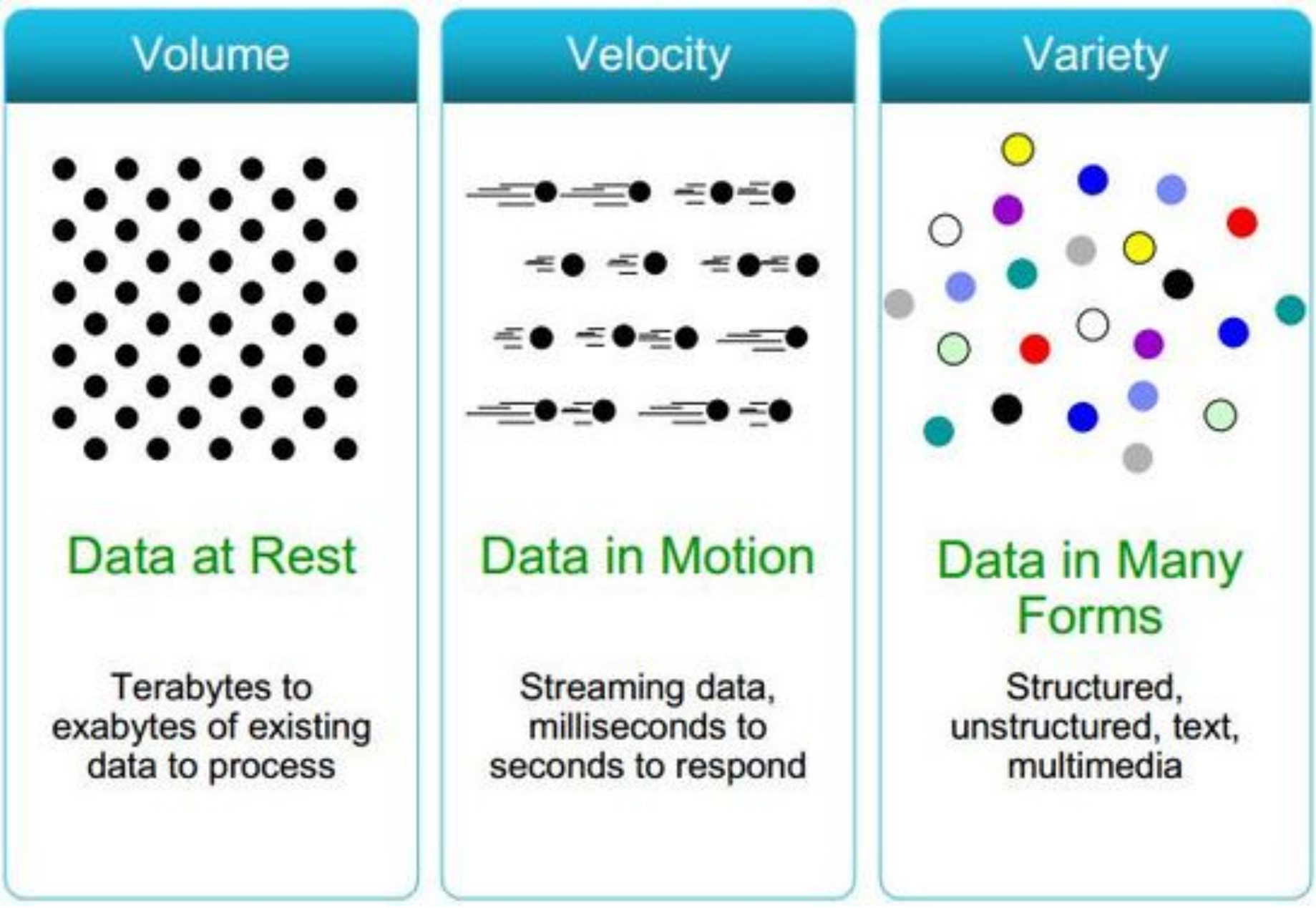
Asst.Prof. Natawut Nupairoj, Ph.D.
Department of Computer Engineering
Chulalongkorn University
natawut.n@chula.ac.th

CHULA ENGINEERING
Foundation toward Innovation

Data Lifecycle



Data Characteristics - 3V



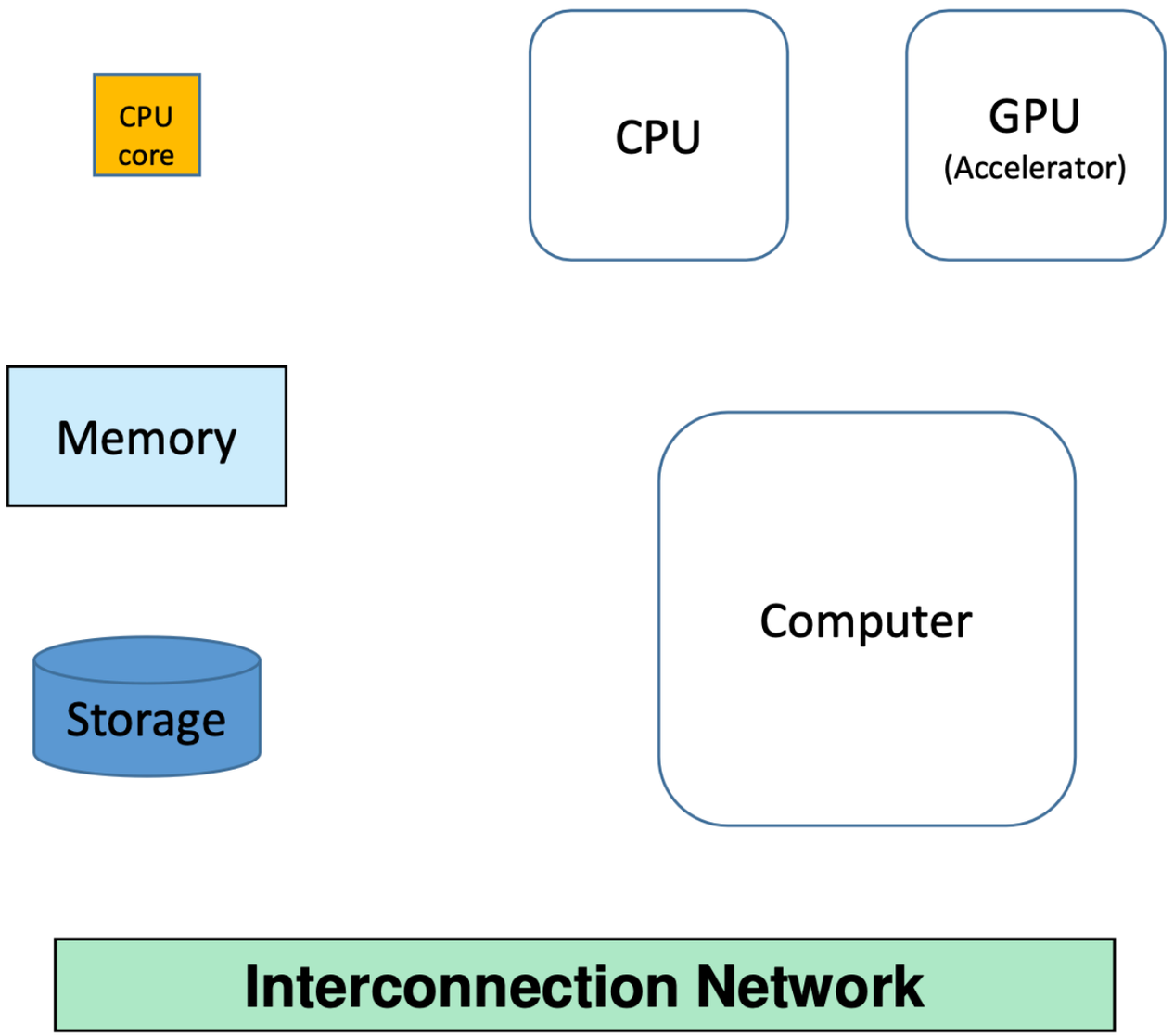
Source: IBM

Needs to Understand Processing Architecture

- 3Vs put lots of constraints in processing architecture
  - How we can process large volume of data
  - How we can process stream data in real-time
  - How we can process unstructured e.g. multimedia data
- To solve these problems effectively, we will need to understand and effectively utilize the underlying architecture

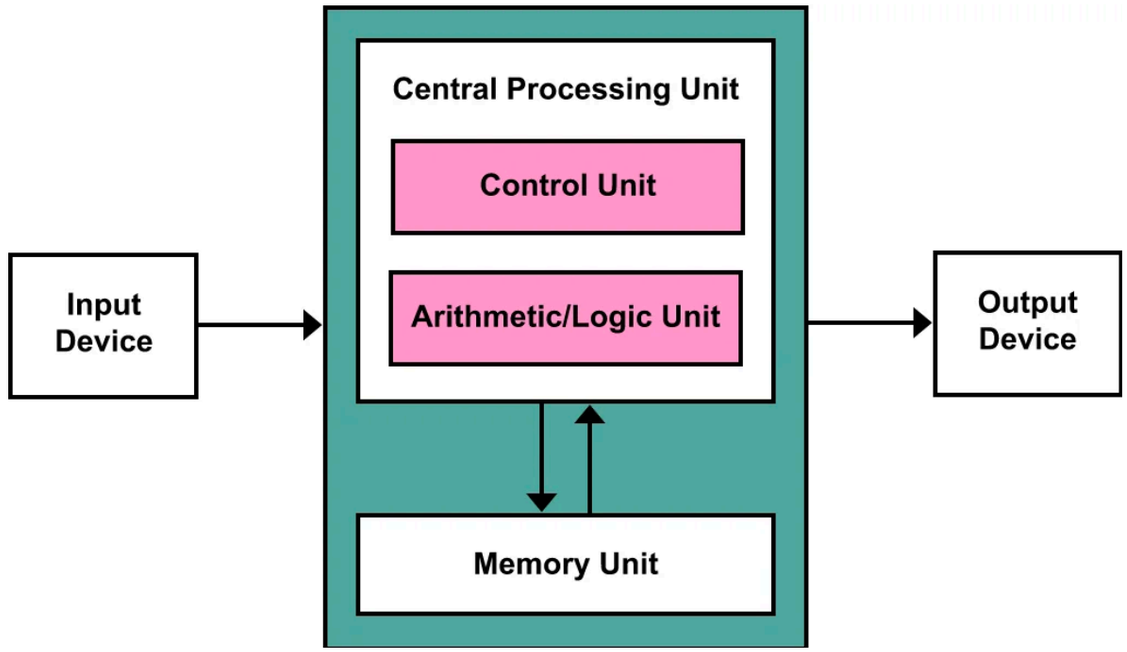
Basic Building Blocks

Building Block of a Computer System

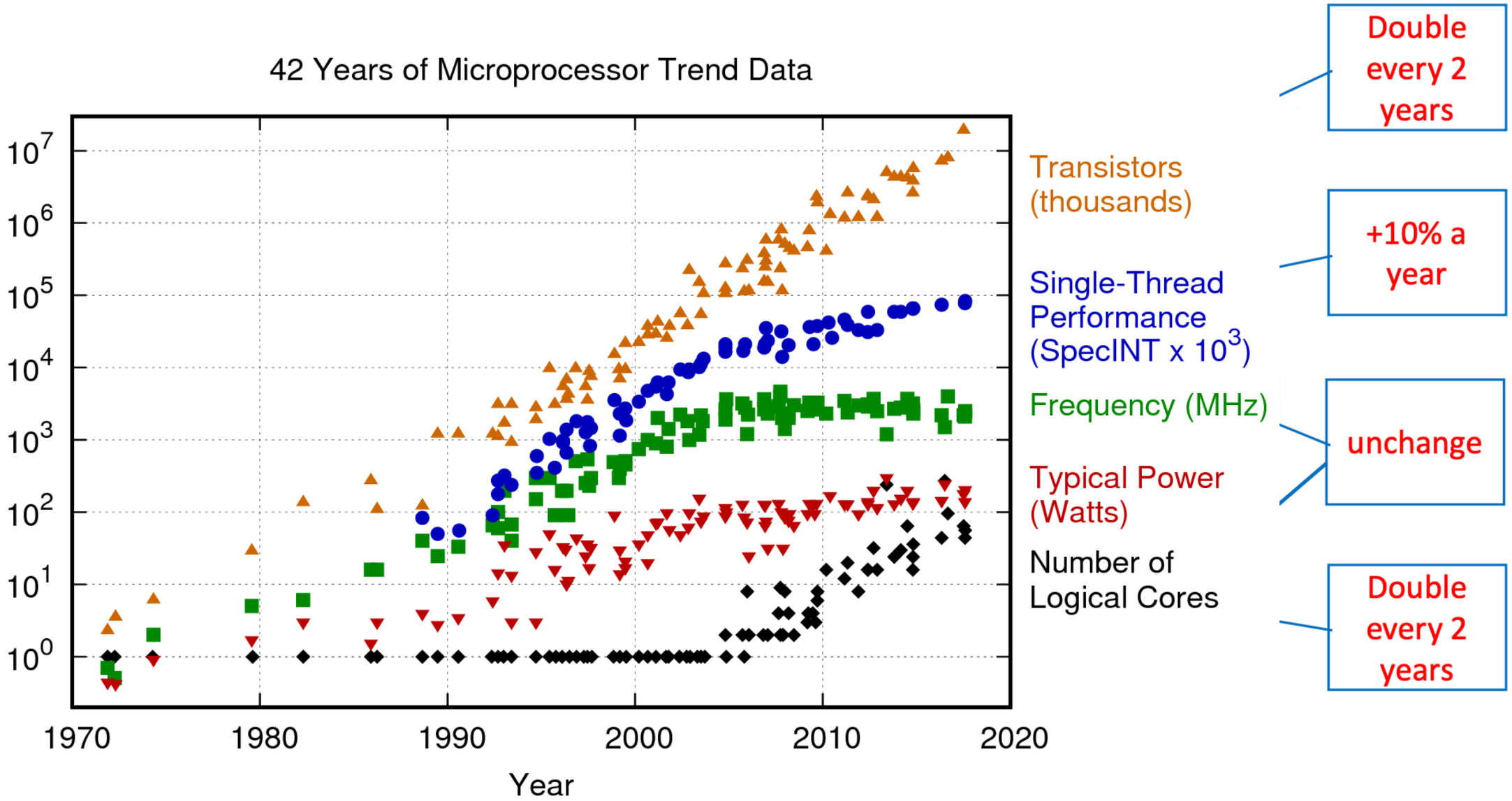


Von Neumann Architecture

- Also known as "stored-program computer”
- both program instructions and data are kept in (same) memory
- Have been the foundation of all CPUs
- Fetching an instruction and performing a data operation cannot occur at the same time because they share a common bus
  “Von Neumann Bottleneck”

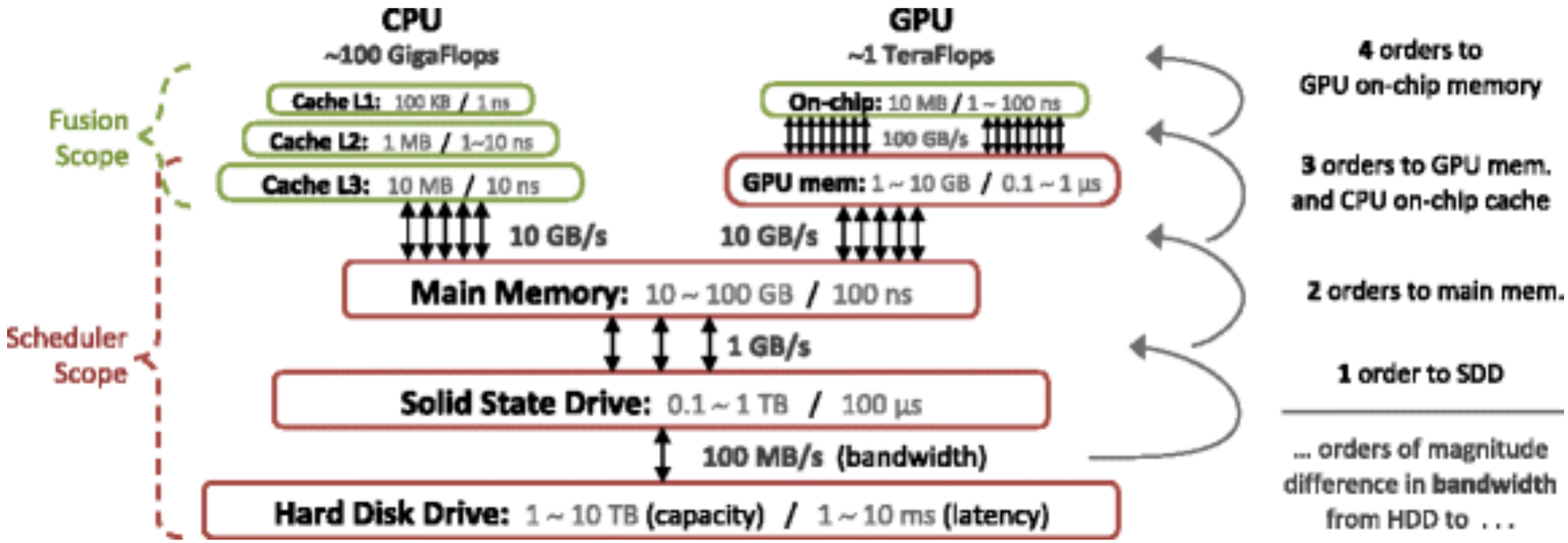


Microprocessor Trends



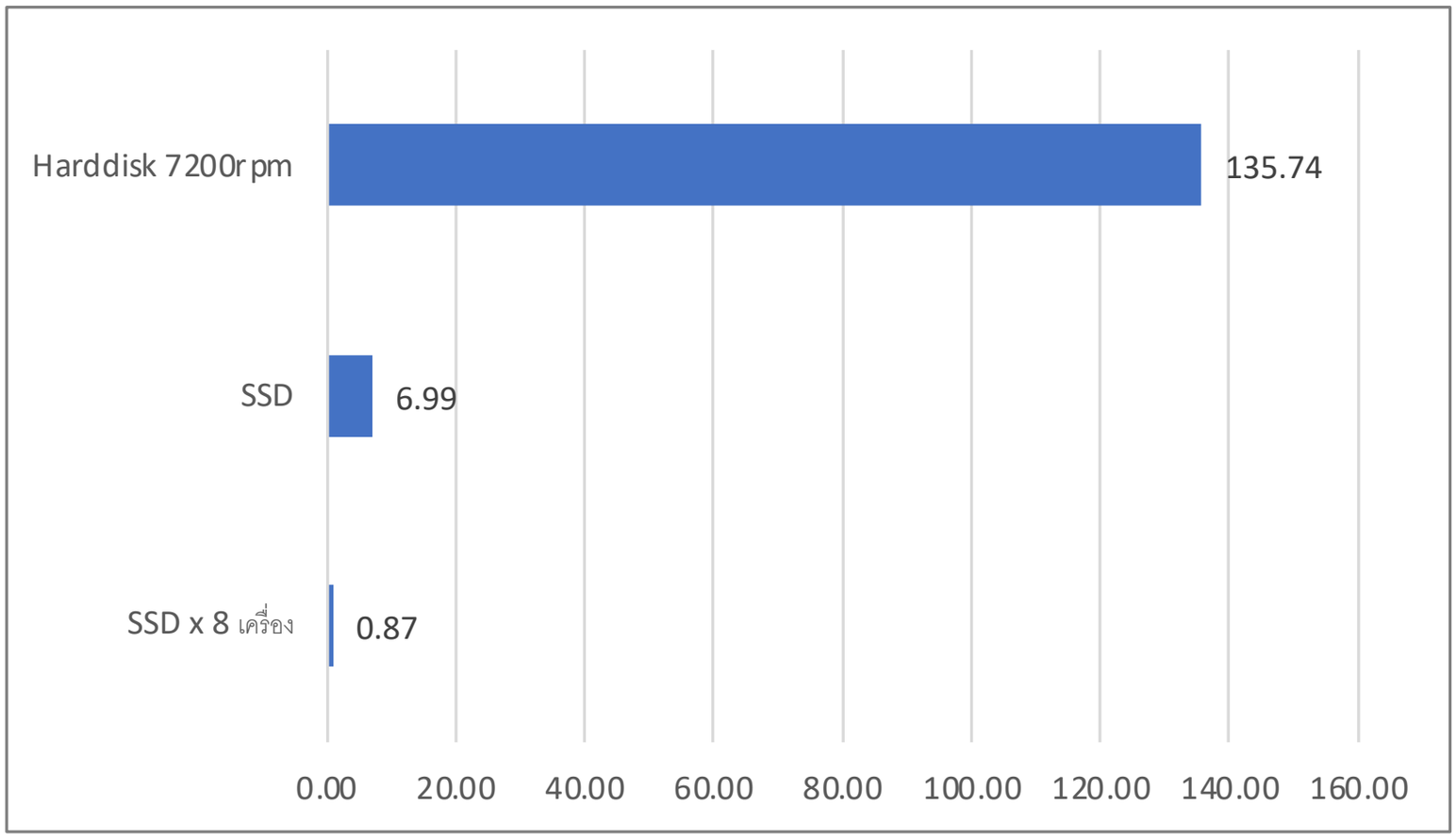
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Memory Hierarchy and Bandwidth



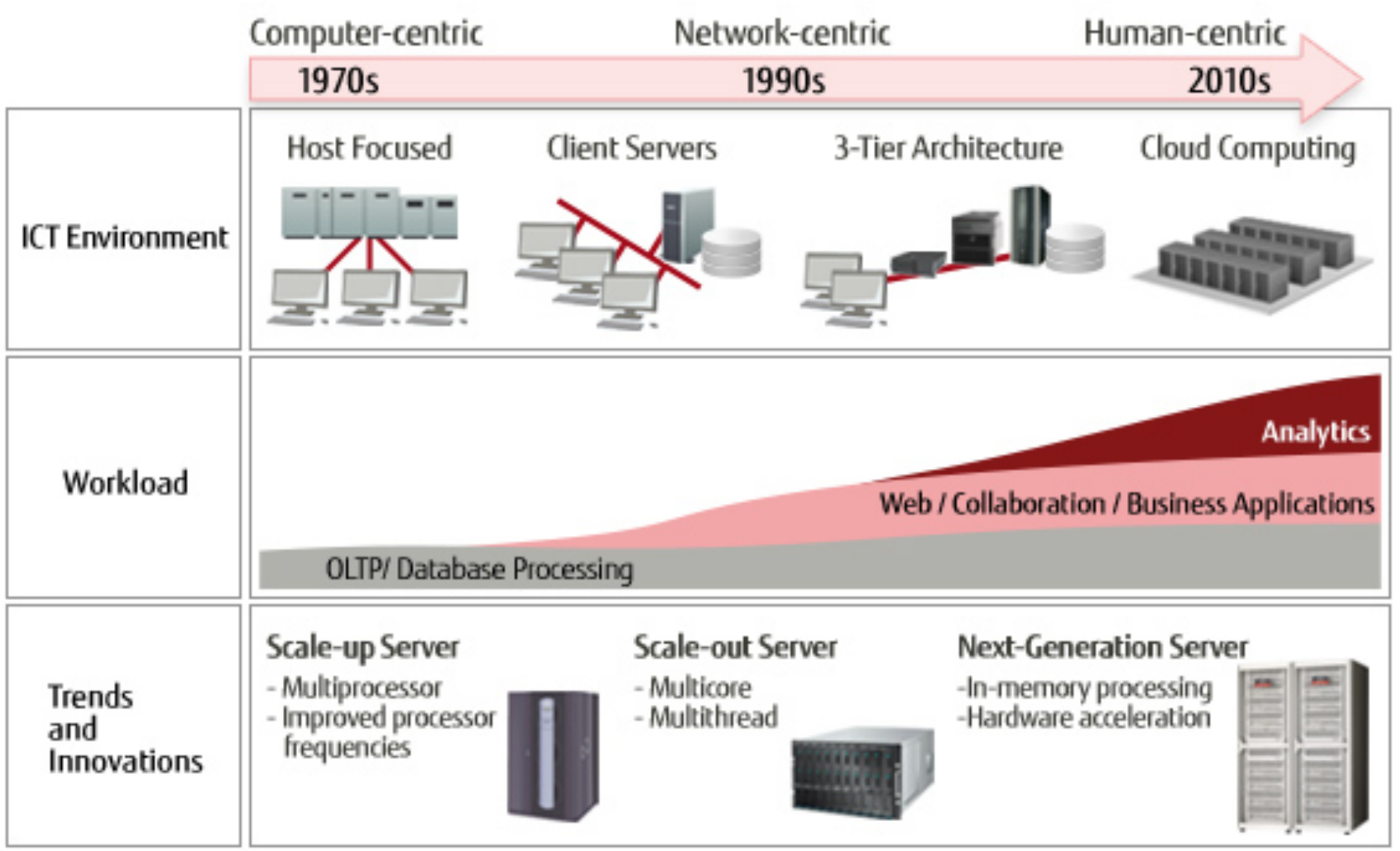
https://www.researchgate.net/publication/321150603_A_compiler_approach_to_map_algebra_automatic_parallelization_locality_optimization_and_GPU_acceleration_of_raster_spatial_analysis

Limitations of Disk Data Transfer Bandwidth



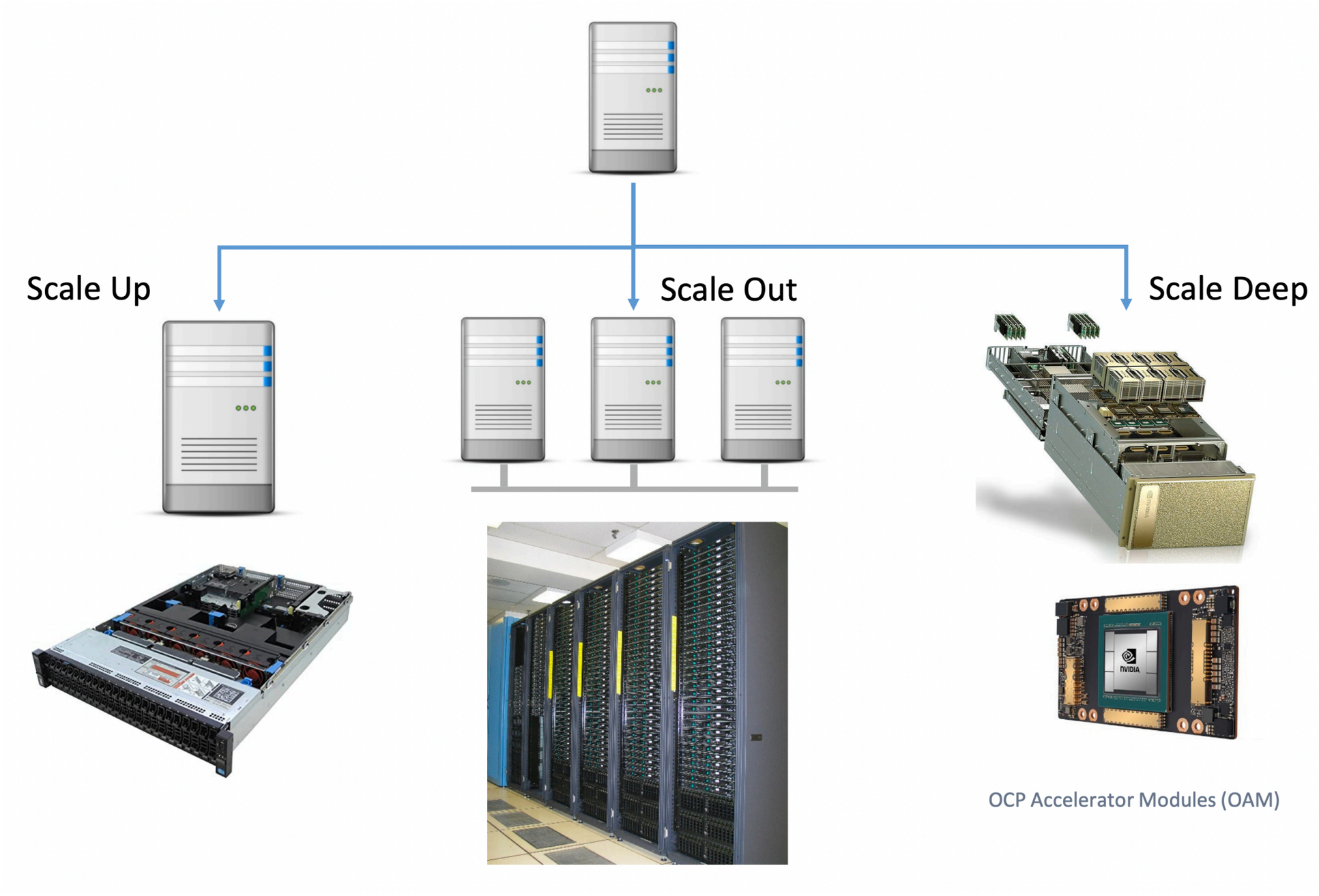
Theoretical time (in minutes) to read 1-TB data from disks

Apple M2 Ultra Memory Bandwidth = 800 GB/s
Time to transfer 1-TB data = 0.02 minutes

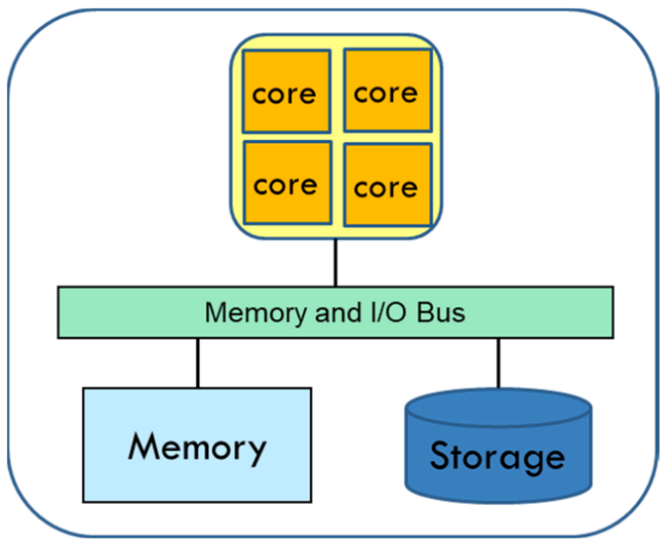


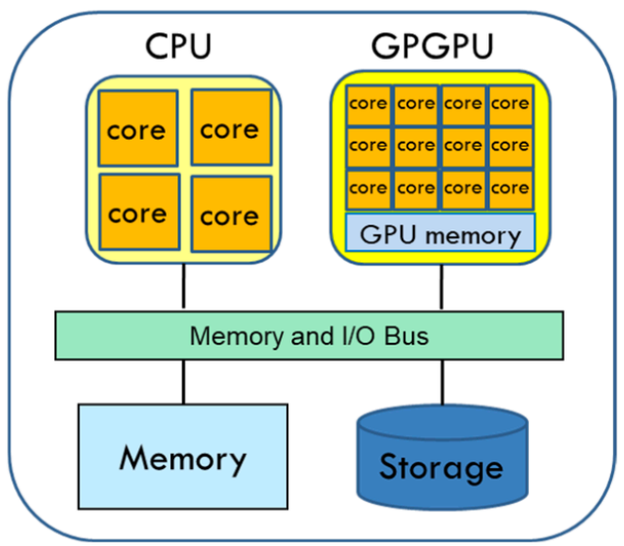
Source: https://www.fujitsu.com/global/products/computing/servers/unix/sparc/key-reports/featurestory/fs-20140813.html

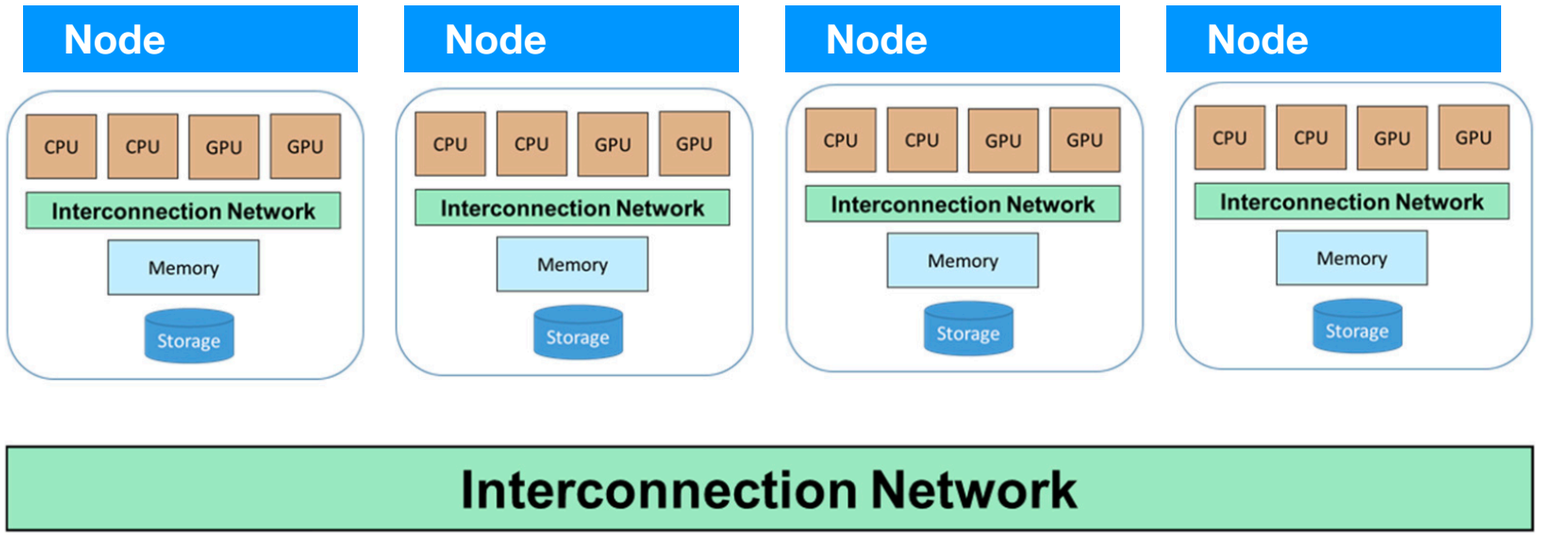
Scale Up / Scale Out / Scale Deep



Cluster Computing

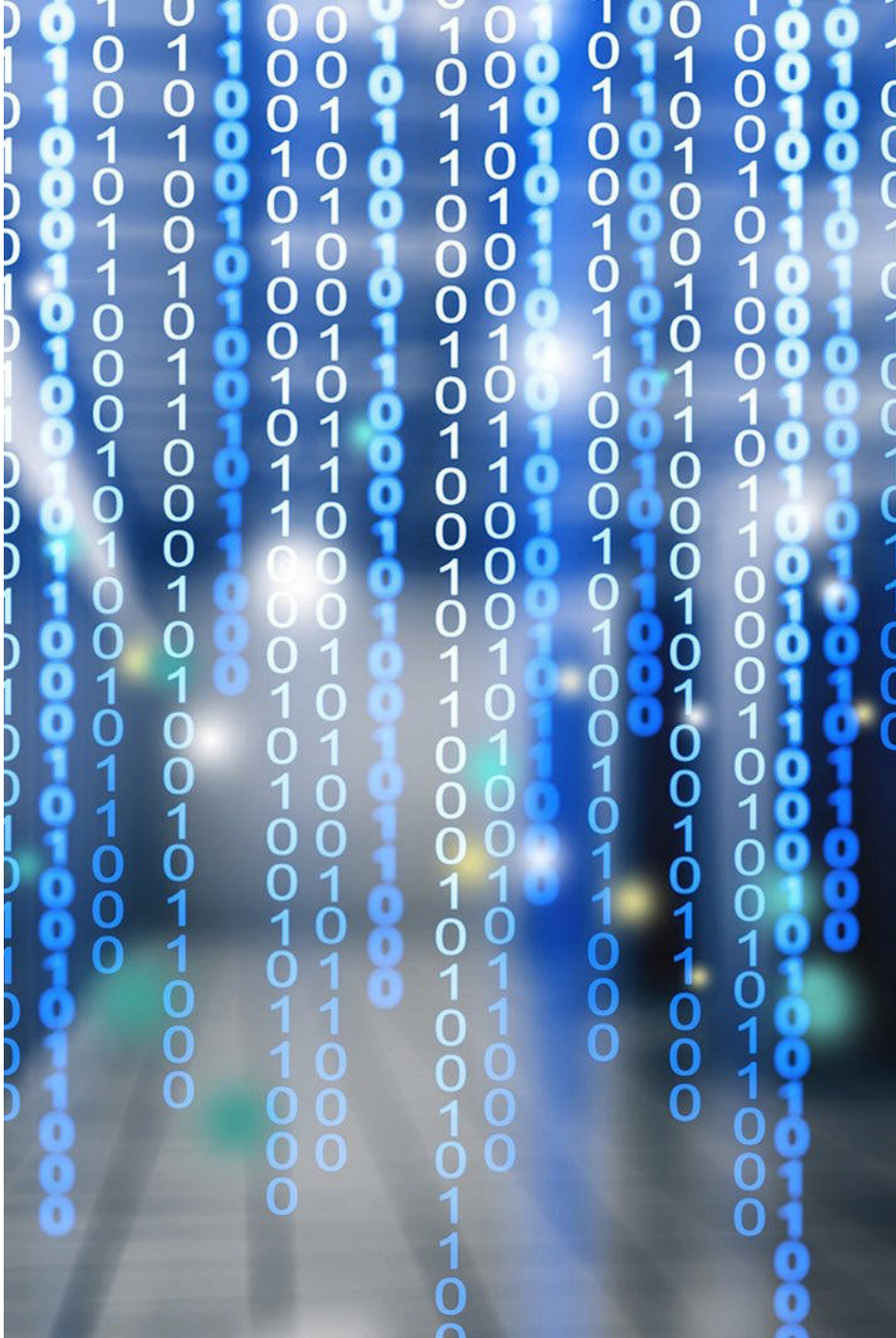






Big Data - Early Days

Parallelism and Hadoop



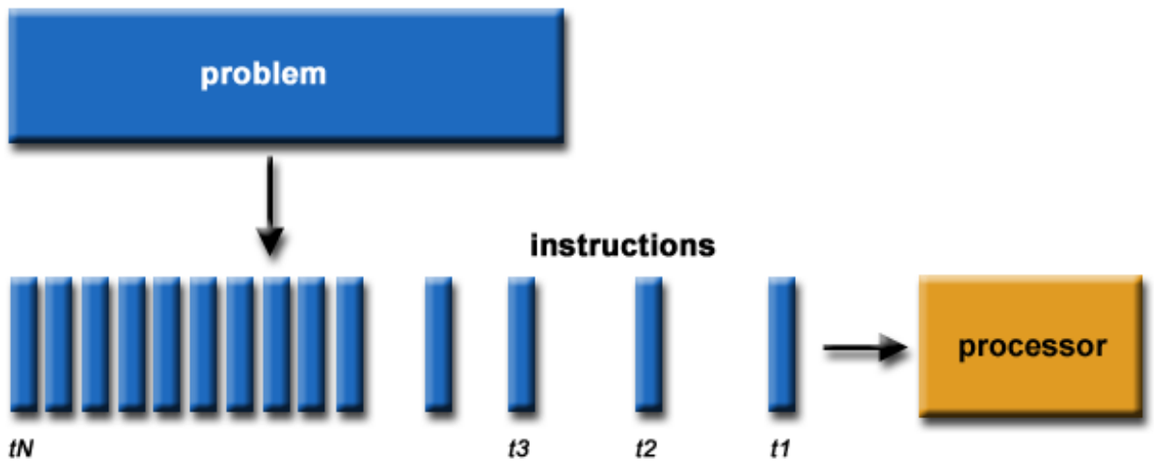
Parallelism

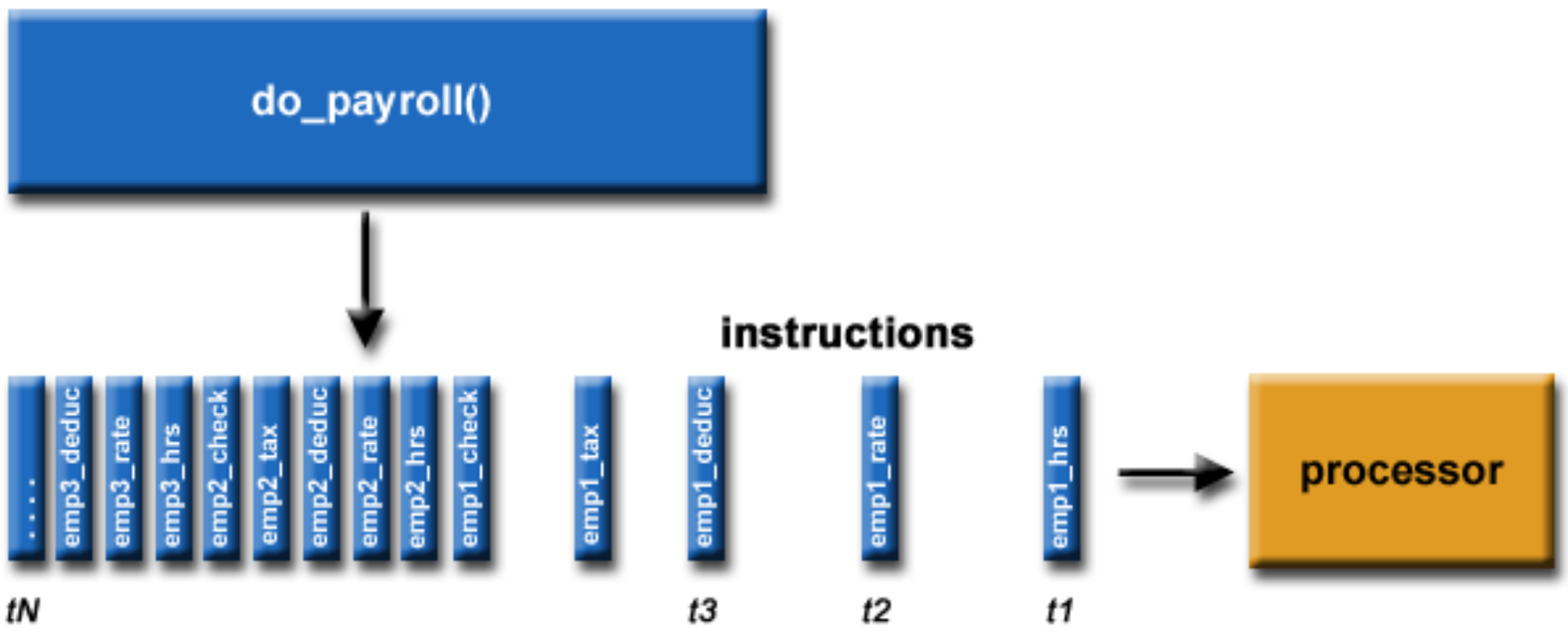
- Big Data relies on the concept of Parallelism on Cluster Computing

- Parallelism is the potential to divide a task into sub-tasks and perform them in parallel

- Parallel computing involves how to exploit parallelism in software with parallelism in hardware (parallel architecture)

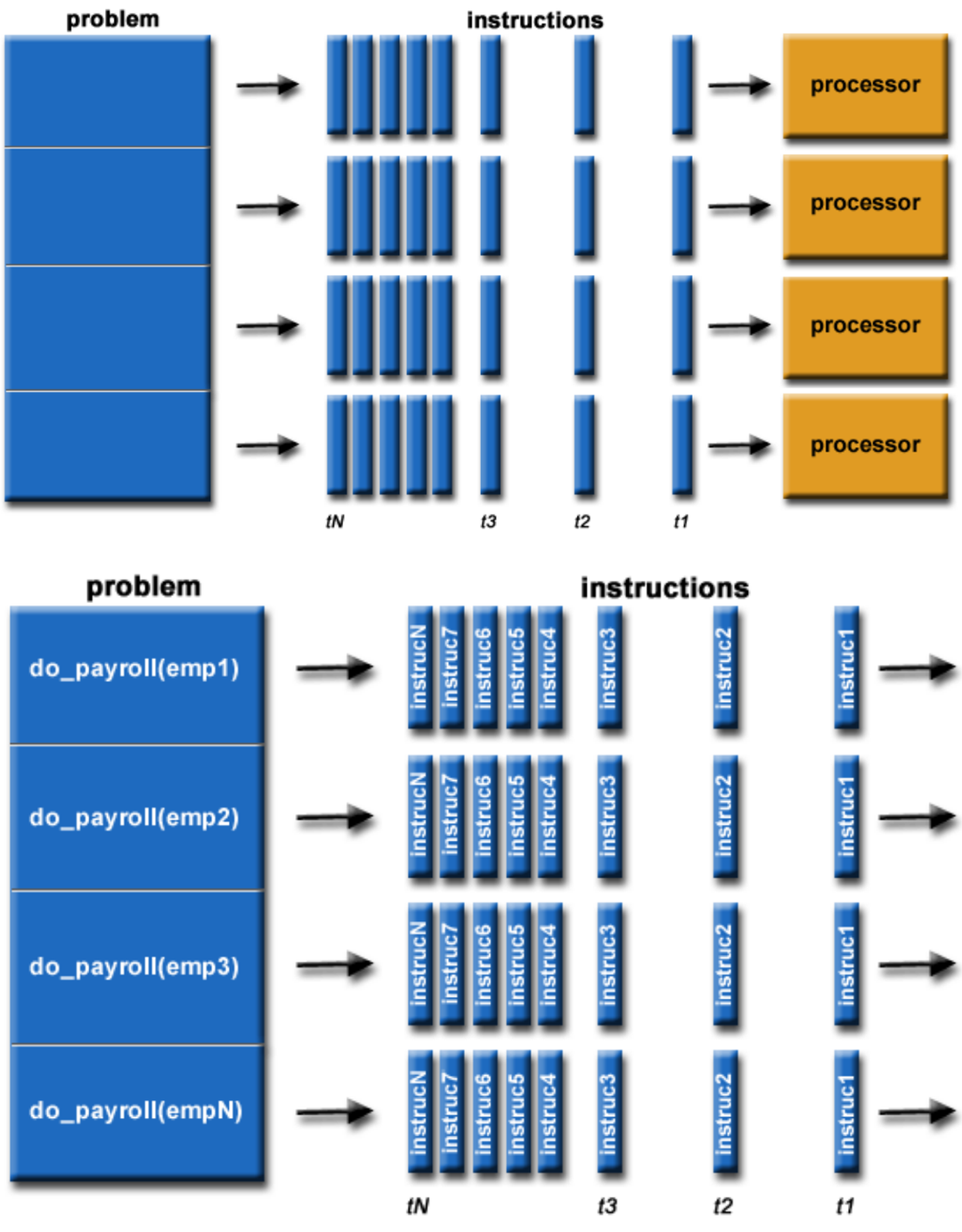
Serial Computation





https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Parallel Computation



https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Difficulties of Parallelism and Big Data

- Scalability of a parallel system is relative to the ability to exploit parallelism on large-scale clusters (100s-1000s nodes)

- Develop a program for a cluster with 100 nodes can be very difficult
  - How to write a program for each node
  - How to exchange data between nodes
  - How to manage nodes (program distribution, node availability, node coordination)
  - How recovery from node failures

Early Days of Big Data - Apache Hadoop

- Opensource software framework inspired by Google Search Engine Architecture

- Provide easy-to-program scale-out foundation for data-intensive applications on large clusters of commodity hardware

- Three key components: cheap cluster, MapReduce, HDFS

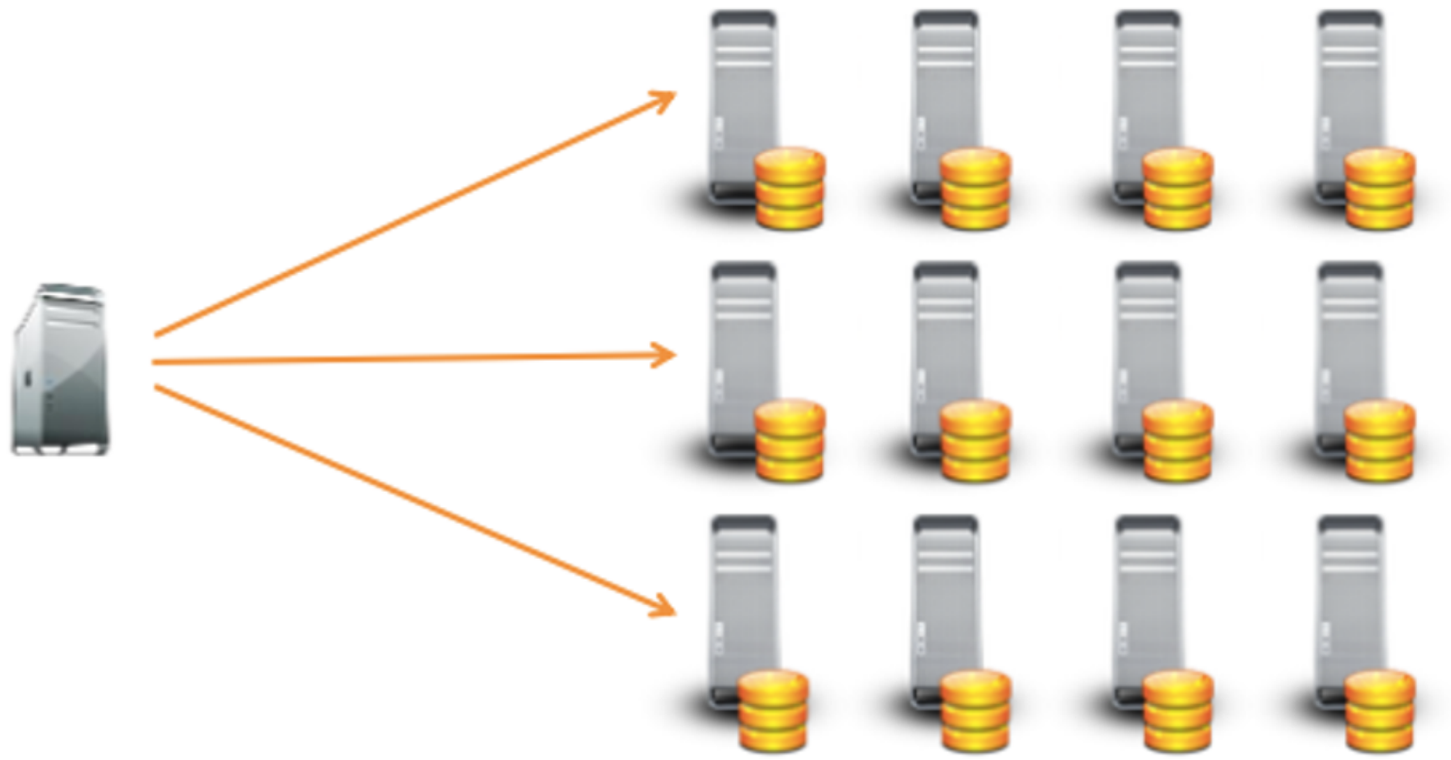
- Hadoop File System (HDFS) has been widely used

- Users: Yahoo!, Facebook, Amazon, eBay, American Airline, Apple, Google, HP, IBM, Microsoft, Netflix, New York Times, etc.

Understand Hadoop in Principles

- Hadoop relies on processing data from multiple disks
  - High throughput
  - Parallel programming is required
    - MapReduce simplifies parallel programming on multiple-disk cluster
- Cheap cluster can suffer from failure very frequently
- Failure recovery and data replication mechanisms

Principles of MapReduce



1. Data is distributed across machine
2. MAP – all nodes process data
3. REDUCE – gather results back

Apache Hadoop 3.X

Hadoop Common

Hadoop MapReduce

Other Data Processing Engines

Hadoop Yarn

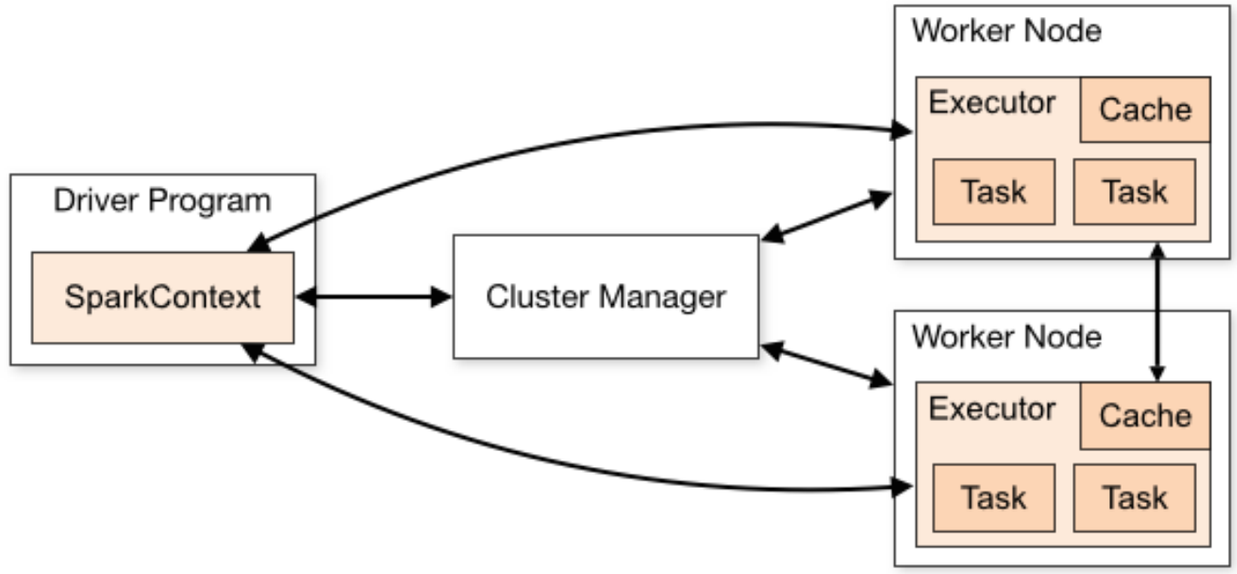
Hadoop Distributed File System

Hadoop Limitations

- Hadoop is disk-oriented
- Suitable for Batch System; No real-time, repetitive queries
- Data Scientist = Analyze, Discover, Investigate

Apache Spark

Spark Core and RDD

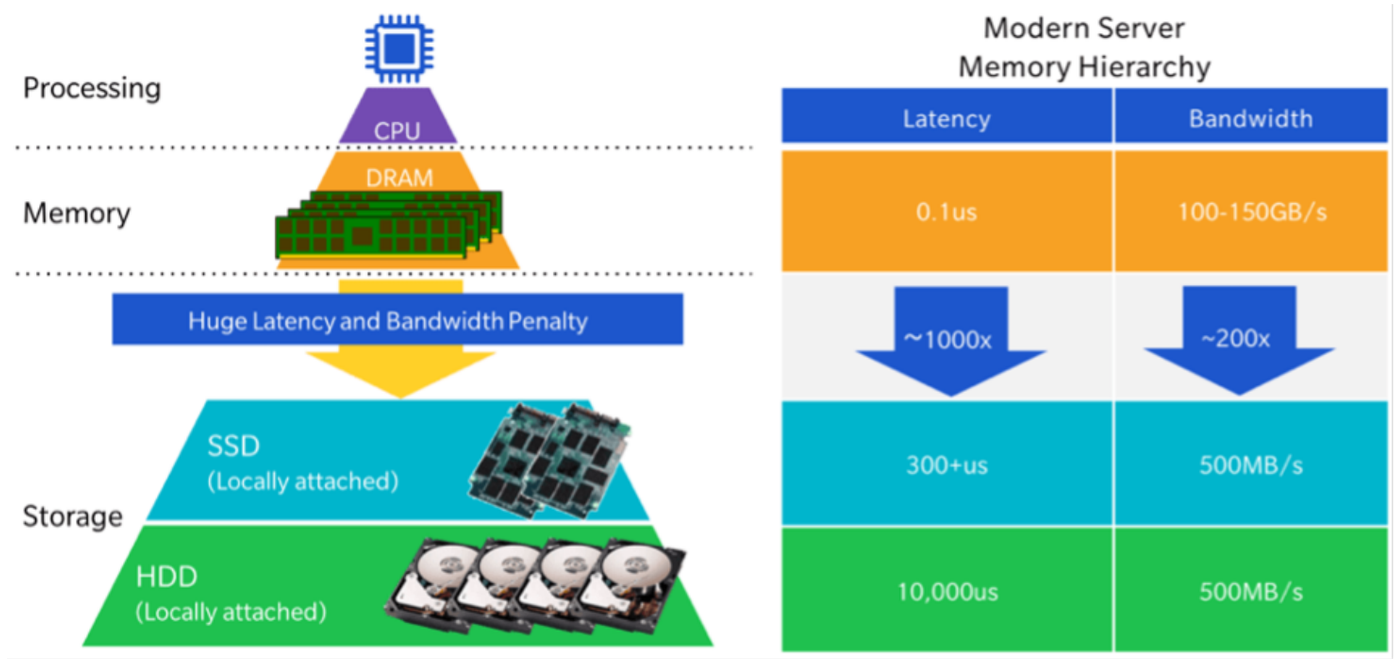


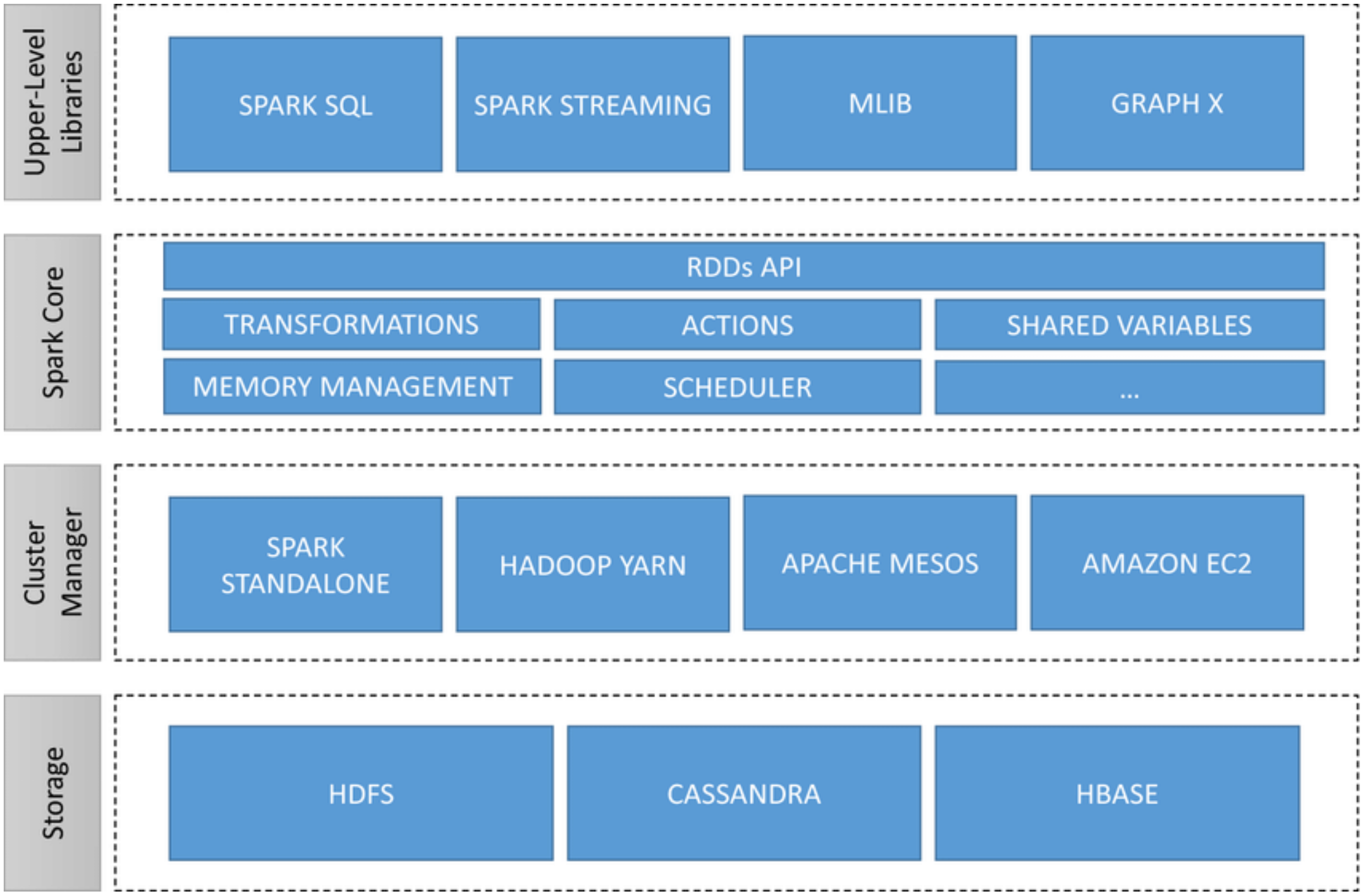
Apache Spark

- Spark = In-Memory Data Processing
  - Extend MapReduce model to support interactive queries and stream processing
  - Keep data in memory as long as possible
    - Spark supports interactive queries with Spark Shell (Scala, Python, and R)
- Data Scientist Friendly
  - Low latency (interactive) queries on historical data
  - Low latency queries on live data (streaming)
  - Compatibility with popular systems (Hadoop)

Spark's Principles

- Aggressive use of memory
- Memory transfer rates >> disk or even SSDs
- Many datasets already fit into memory
  - E.g., 1TB = 1 billion records @ 1 KB each
- Memory density (still) grows with Moore's law - RAM/SSD hybrid memories at horizon





N. Fikri, et al. “An adaptive and real-time based architecture for financial data integration”, Journal of Big Data, Nov. 2019

Spark Ecosystems

Data science and Machine learning

scikit learn · pandas · TensorFlow · PyTorch · mlflow · R

SQL analytics and BI

Apache Superset · Power BI · Looker · re'dash · +ableau SOFTWARE · dbt

Storage and Infrastructure

elasticsearch · mongoDB. · APACHE kafka. · DELTA LAKE · kubernetes · Apache Airflow · Parquet · Microsoft SQL Server · cassandra · Apache orc™

Spark Performance

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Running pyspark

- Install anaconda
- Download latest apache spark from spark.apache.org
- Setup proper environment variables

```
# spark environment
export SPARK_HOME=/Users/natawut/spark
export PATH="$HOME/bin:$SPARK_HOME/bin:$PATH"
export PYTHONPATH="$SPARK_HOME/python:$PYTHONPATH”

# jupyter and pyspark integration
export ANACONDA_HOME="/Users/natawut/anaconda3"
export PYSPARK_PYTHON=$ANACONDA_HOME/bin/python
export PYSPARK_DRIVER_PYTHON=$ANACONDA_HOME/bin/jupyter
export PYSPARK_DRIVER_PYTHON_OPTS="notebook"
```

Running pyspark: direct from command line

- Running pyspark

  pyspark

- This method starts Spark standalone and connect Jupiter notebook (or spark shell) to Spark

- References
  - Linux and mac

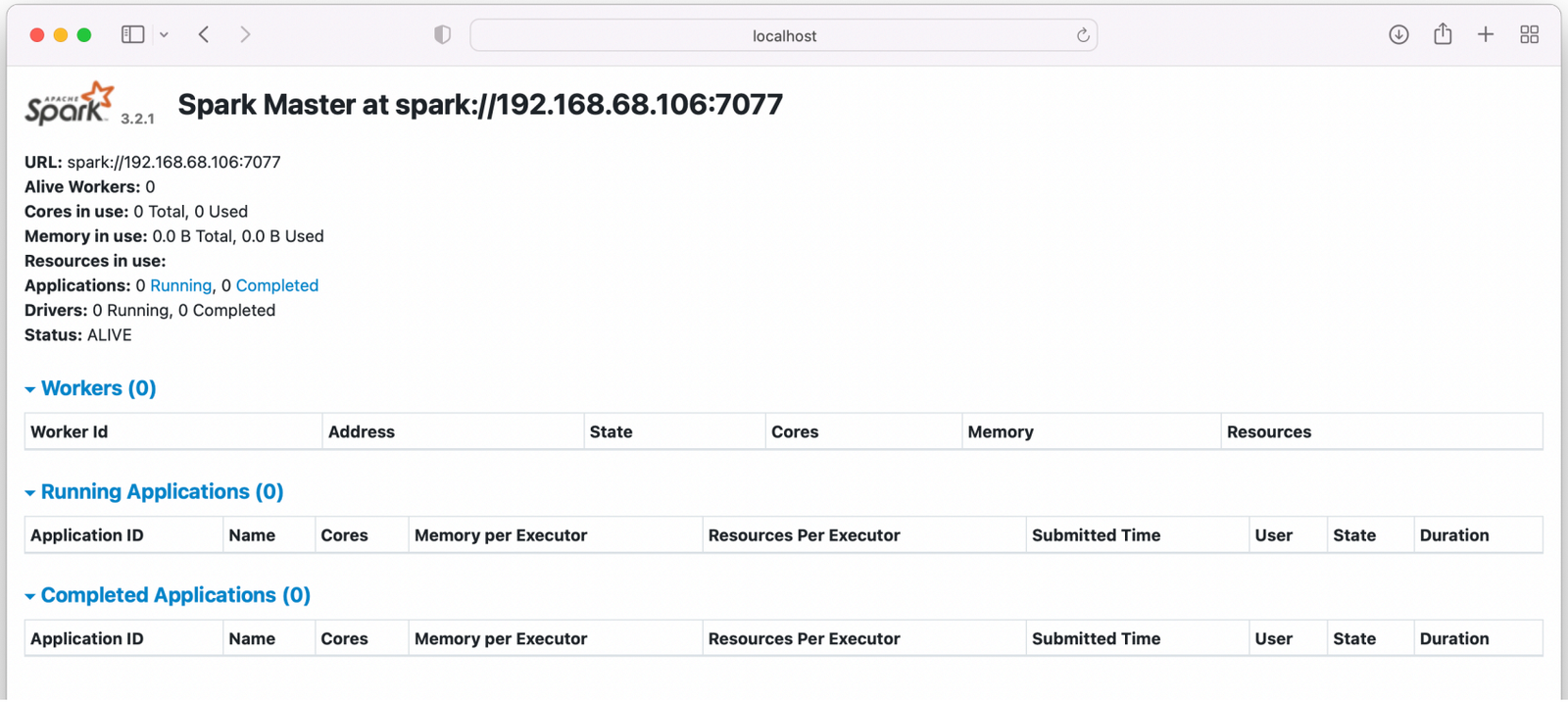
    https://medium.com/@GalarnykMichael/install-spark-on-ubuntu-pyspark-231c45677de0
  - Windows
    - https://medium.com/@GalarnykMichael/install-spark-on-windows-pyspark-4498a5d8d66c
  - Using pip (Linux, mac, and windows)
    - http://sigdelta.com/blog/how-to-install-pyspark-locally/

Running pyspark: import packages

- Start spark cluster separately

  start-master.sh

- Import packages at the beginning and get spark context
- This is suitable for environment with existing spark cluster e.g. production



http://localhost:8080

In [1]:
```
import os
import pyspark
from pyspark.sql import SQLContext, SparkSession
```

In [2]:
```
spark = SparkSession \
        .builder \
        .master('spark://192.168.68.106:7077') \
        .appName("sparkFromJupyter") \
        .getOrCreate()

sc = spark.sparkContext
sqlContext = SQLContext(sparkContext=sc, sparkSession=spark)
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/opt/spark/jars/spark-unsafe_2.12-3.2.
1.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
22/02/02 11:46:38 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-ja
va classes where applicable
```

In [3]:
```
sc
```

Out[3]:

SparkContext

Spark UI

Version
`v3.2.1`
Master
`spark://192.168.68.106:7077`
AppName
`sparkFromJupyter`

In [4]:
```
sqlContext
```

Out[4]:
```
<pyspark.sql.context.SQLContext at 0x7f87b0aebdf0>
```

In [5]:
```
spark.stop()
```

In [ ]:

Using findspark to start a Spark local cluster in Google Colab

---

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

https://www.analyticsvidhya.com/blog/2020/11/a-must-read-guide-on-how-to-work-with-pyspark-on-google-colab-for-data-scientists/

Spark Preparation

We check if we are in Google Colab. If this is the case, install all necessary packages.

To run spark in Colab, we need to first install all the dependencies in Colab environment i.e. Apache Spark 3.3.2 with hadoop 3.2, Java 8 and Findspark to locate the spark in the system. The tools installation can be carried out inside the Jupyter Notebook of the Colab. Learn more from A Must-Read Guide on How to Work with PySpark on Google Colab for Data Scientists!

In [1]:
```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False
```

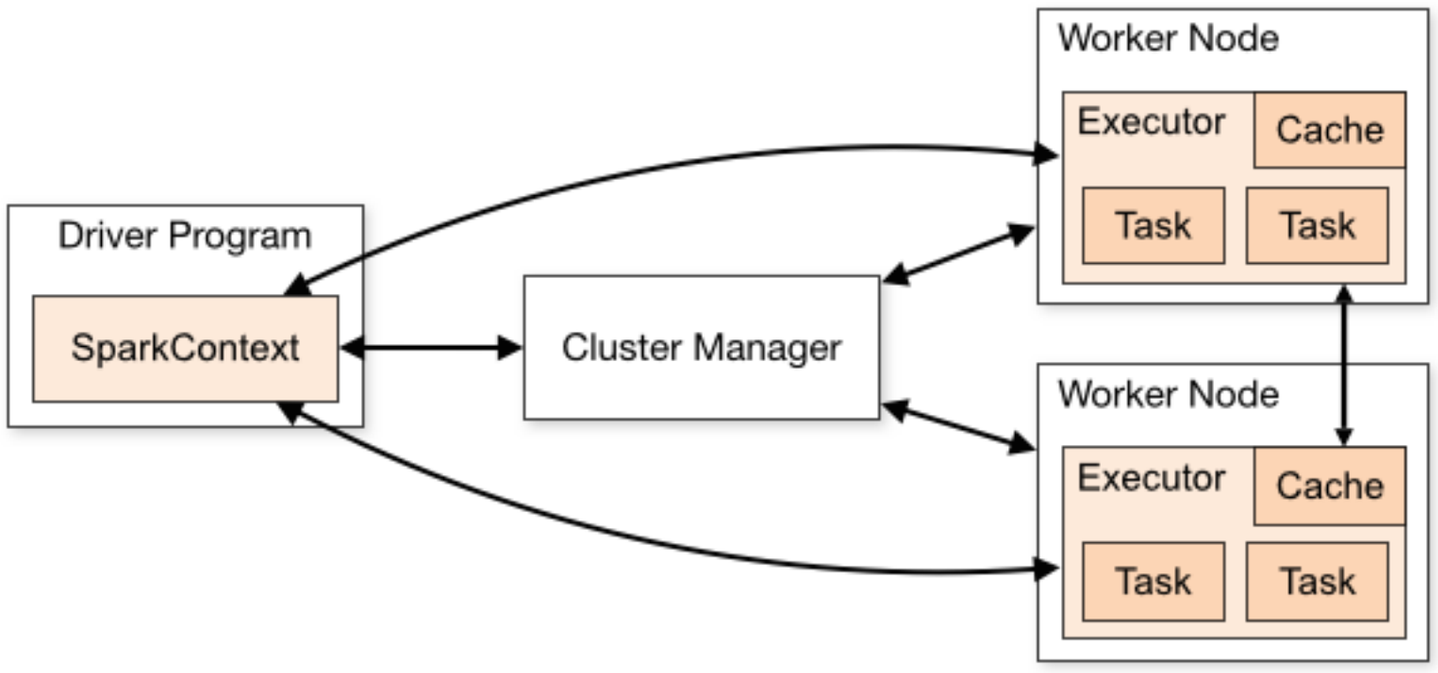
In [2]:
```
if IN_COLAB:
    !apt-get install openjdk-8-jdk-headless -qq > /dev/null
    !wget -q https://dlcdn.apache.org/spark/spark-3.3.2/spark-3.3.2-bin-hadoop3.tgz
    !tar xf spark-3.3.2-bin-hadoop3.tgz
    !mv spark-3.3.2-bin-hadoop3 spark
    !pip install -q findspark
    import os
    os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
    os.environ["SPARK_HOME"] = "/content/spark"
```

Start a Local Cluster

Use findspark.init() to start a local cluster. If you plan to use remote cluster, skip the findspark.init() and change the cluster_url according.

Apache Spark High-Level Architecture

- A Spark program is two programs
  - A driver program and a workers program
- Worker programs run on cluster nodes or in local threads
- RDDs are distributed across workers



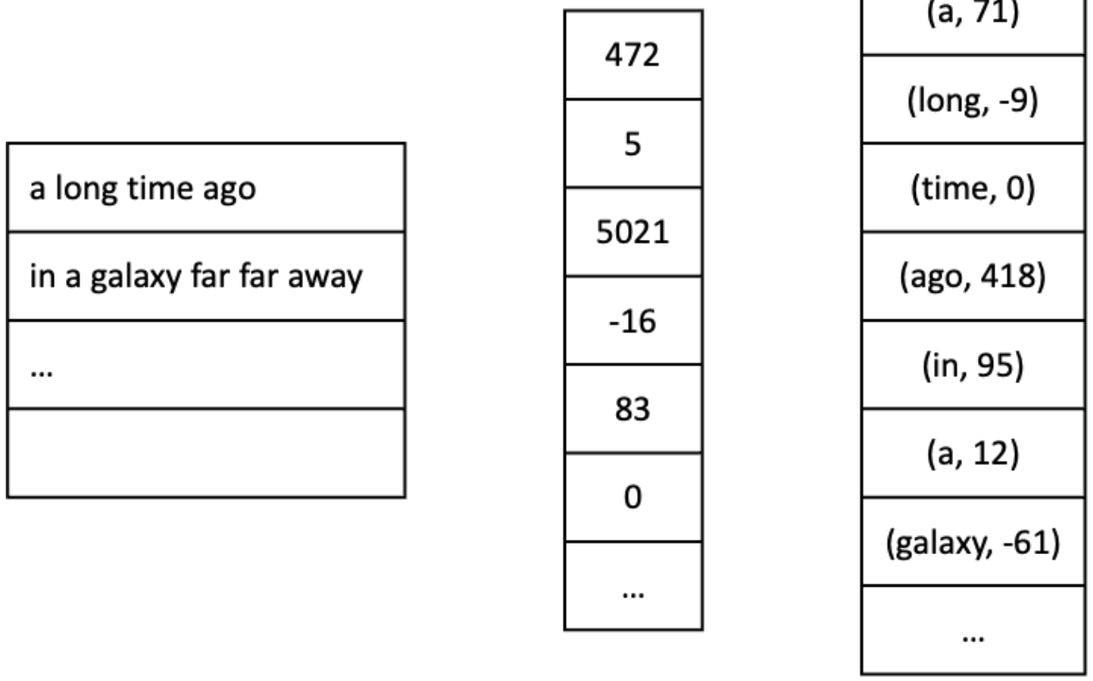
SparkSession / SparkContext / SqlContext

- To use Spark, we will need a ‘SparkSession’ which is an entry point to SparkContext and SqlContext

- SparkContext tells Spark how and where to access a cluster (for client and cluster mode)

- We can also create RDD using method in SparkContext

- SqlContext is similar to SparkContext, but for Spark SQL

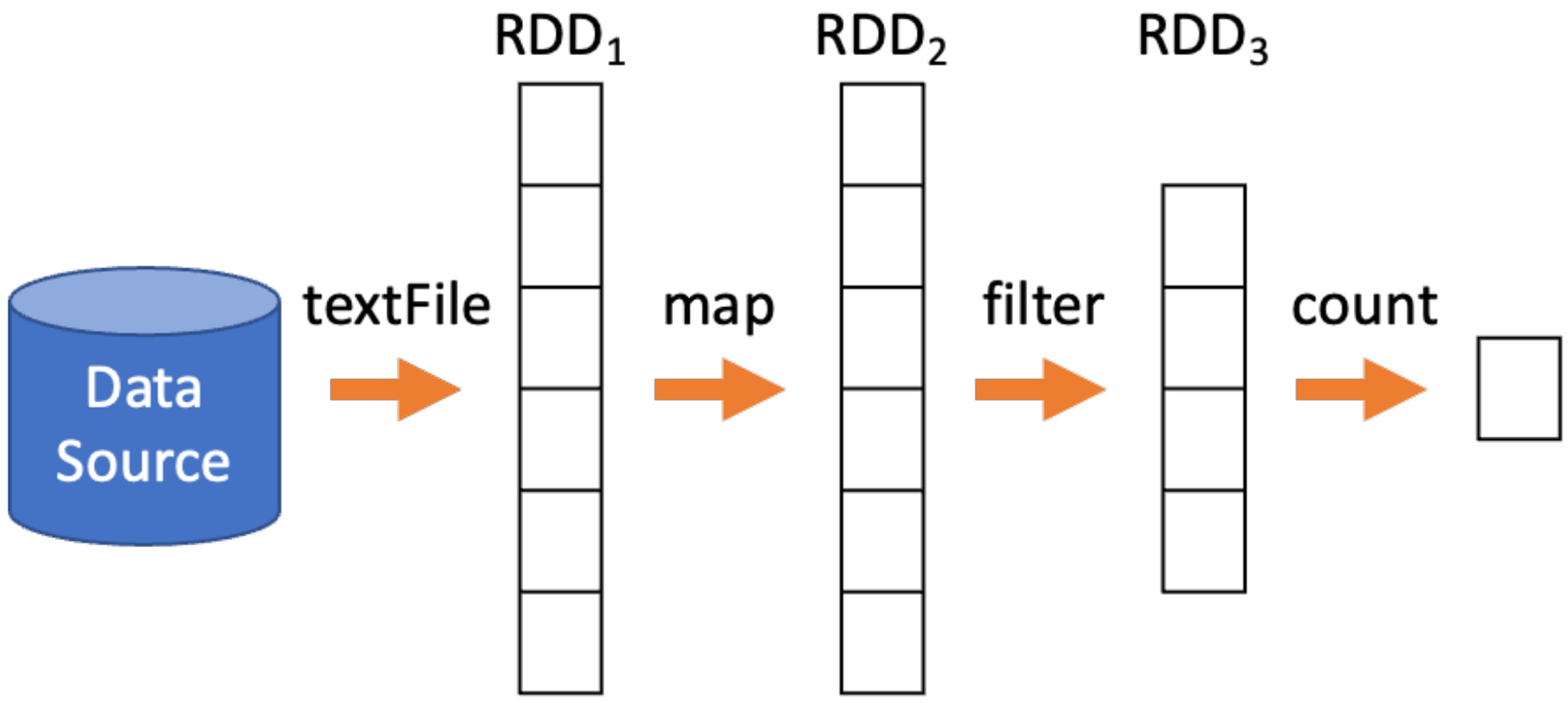


Key Concept: RDD

- Immutable Distributed Collection of Objects
- Can contain any type of Python, Java, or Scala objects
- Two types of operations: transformations and actions
  - Transformations are lazy (not computed immediately)
  - Transformed RDD is executed when action runs on it

Typical Spark Program

- Created from external dataset (e.g. file from HDFS) or existing collection of objects (List, Set)
- Transformation (Create a new RDD from existing RDDs)
- Action (Compute result from an RDD)
- (Maybe) persist (cache) RDDs to disk or memory



Simple RDD Operations

- sc.parallelize(data)
  create an RDD from data

- rdd.count()
  count number of elements in an odd

- rdd.filter(func)
  create a new rdd from existing rdd and keep only those elements that func is true

```
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
n = rdd.count()
print('count = {0}'.format(n))
l = rdd.collect()
print(l)
```

```
count = 5
[1, 2, 3, 4, 5]
```

```
l = rdd.take(3)
print(l)
```

```
[1, 2, 3]
```

```
f_rdd = rdd.filter(lambda d: d > 2)
for d in f_rdd.collect():
    print(d)
print('filter count = {0}'.format(f_rdd.count()))
```

```
3
4
5
filter count = 3
```

RDD Operations
map and reduce

- rdd.map(func)
  create a new rdd by performing function func on each element in an rdd

- rdd.reduce(func)
  aggregate all elements in an rdd using function func

A

```
data = ['line 1', '2', 'more lines', 'last line']
lines = sc.parallelize(data)
print(lines.collect())
```

```
['line 1', '2', 'more lines', 'last line']
```

A

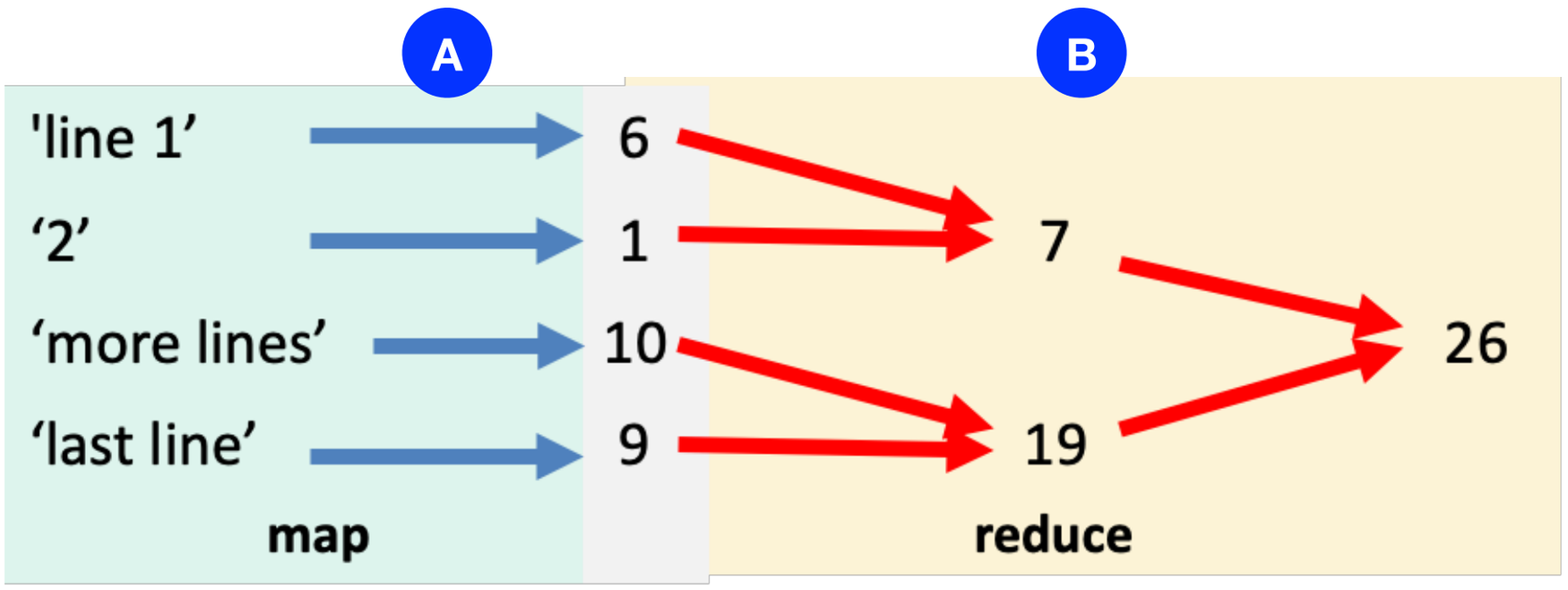
```
lineLengths = lines.map(lambda line: len(line))
print(lineLengths.collect())
```

```
[6, 1, 10, 9]
```

B

```
totalLength = lineLengths.reduce(lambda a, b: a + b)
print(totalLength)
```

```
26
```



RDD Operations: more map and reduce

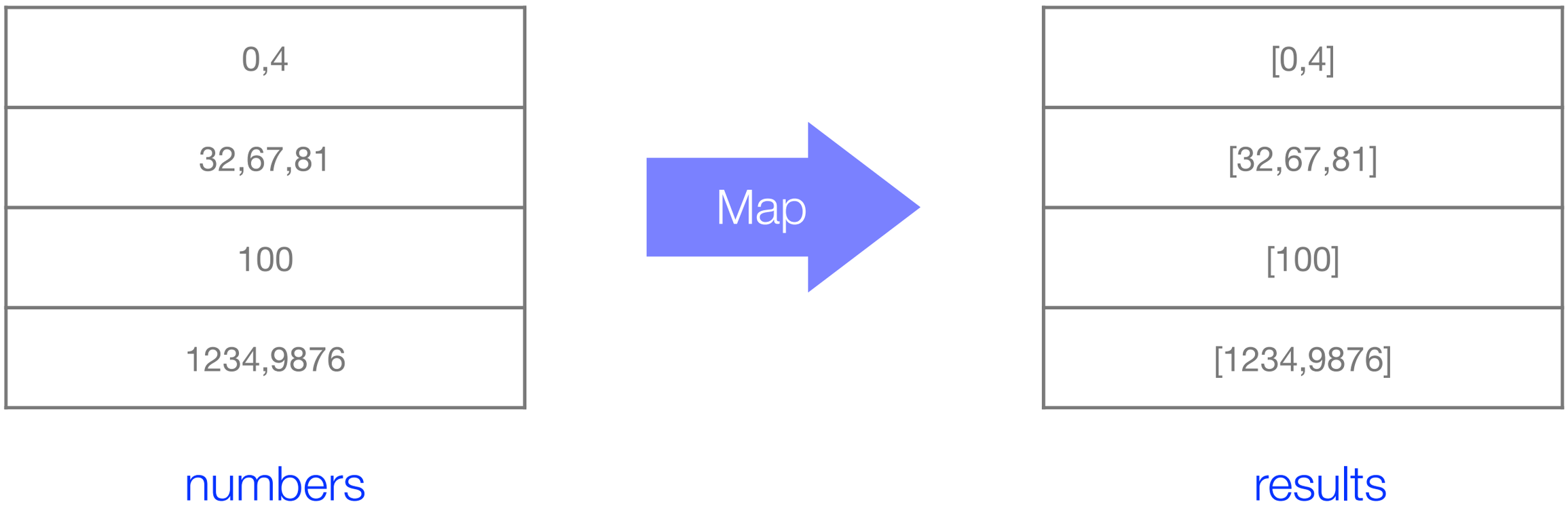
```
data = (1,2,3,4)
rdd = sc.parallelize(data)
rdd2 = rdd.map(lambda x: x*2)
print(rdd2.collect())
sum_val= rdd2.reduce(lambda a, b: a + b)
print('sum = {0}'.format(sum_val))
mul_val = rdd2.reduce(lambda a, b: a * b)
print('mul = {0}'.format(mul_val))
```

```
[2, 4, 6, 8]
sum = 20
mul = 384
```

Flatmap

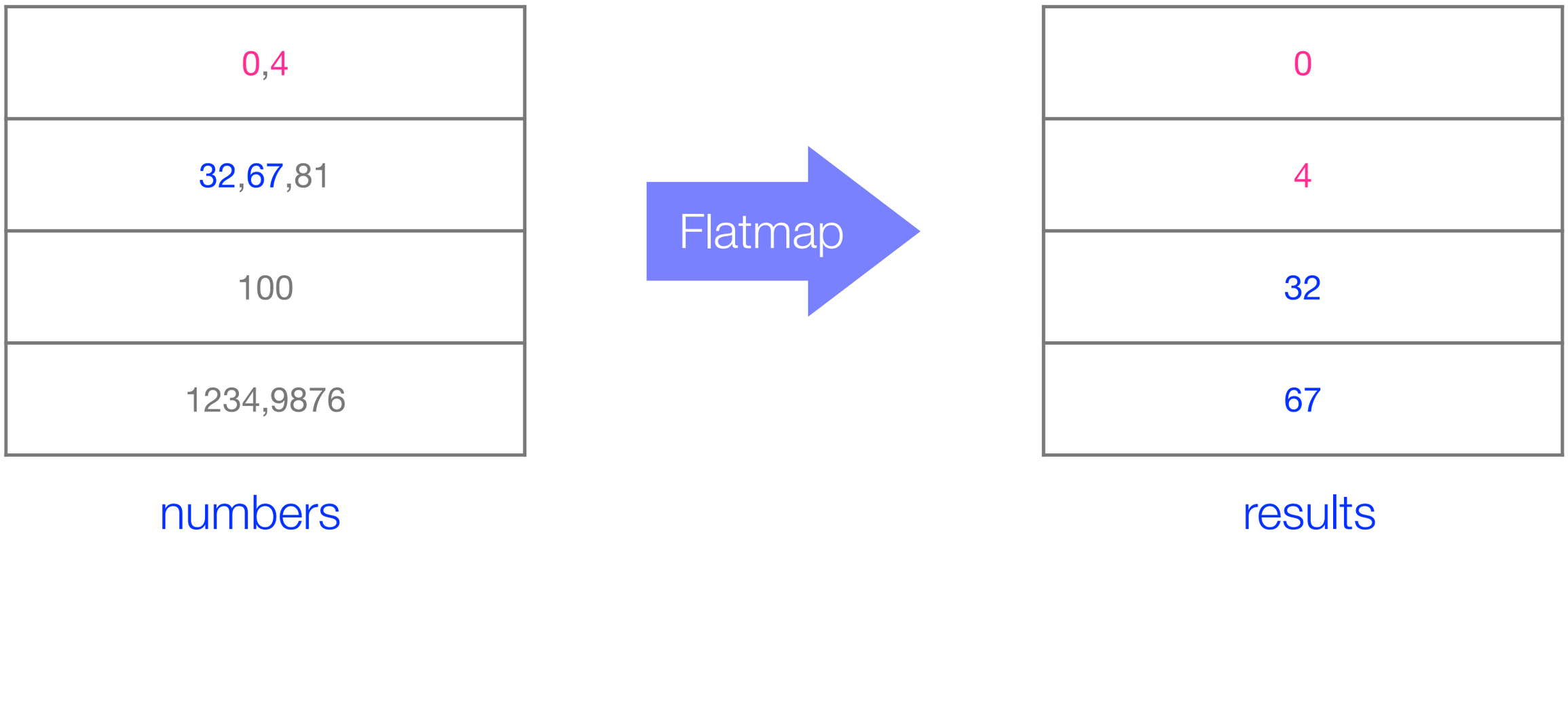
- Flatmap is similar to map; create a new rdd by performing function func on each element in an rdd
- Difference is how results are handled

```
results = numbers.map(lambda line: line.split(","))
```



Flatmap

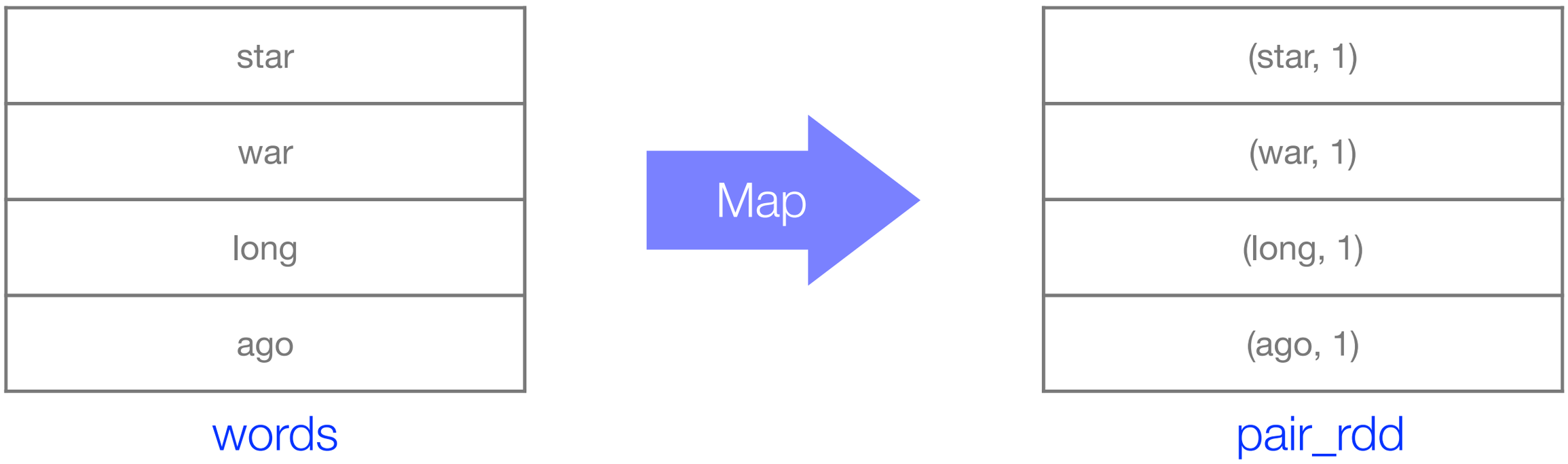
```
results = numbers.flatmap(lambda line: line.split(","))
```



Spark PairRDD

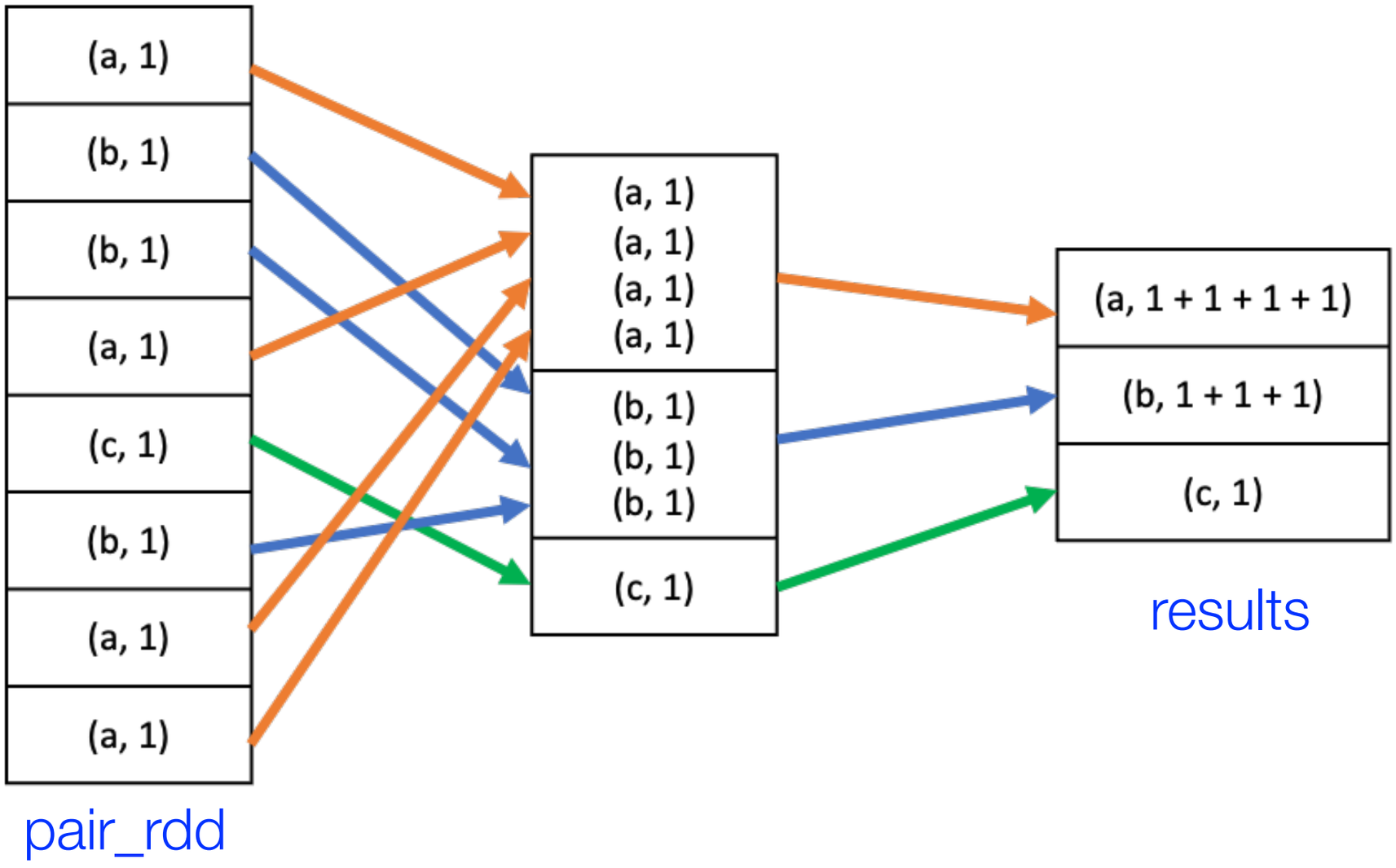
- Each element in PairRDD is a tuple (key, value) where key and value can be of any data type
- Commonly used for operations that require data to be grouped or aggregated by keys such as reduceByKey, groupByKey, combineByKey, and foldByKey to summarize data based on keys

```
pair_rdd = words.map(lambda word:(word, 1))
```



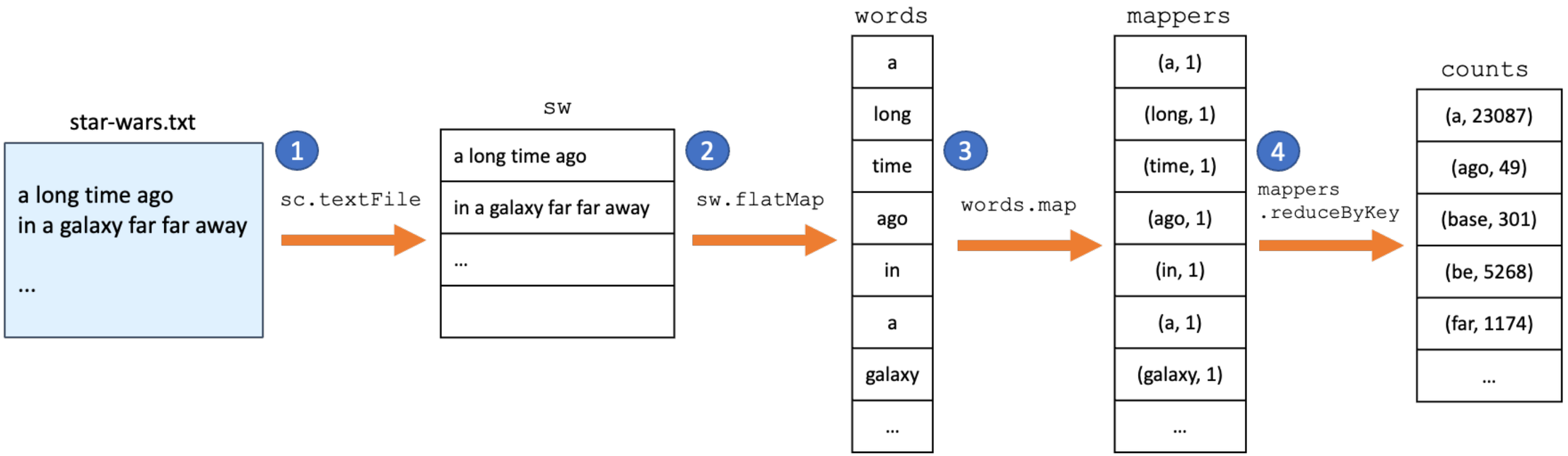
ReduceByKey

```
results = pair_rdd.reduceByKey(lambda x, y: x+y)
```



Example: Word Count

```
sw = sc.textFile('star-wars.txt')
words = sw.flatMap(lambda line: line.split())
mappers = words.map(lambda word: (word, 1))
counts = mappers.reduceByKey(lambda x, y: x+y)
```



In [1]:
```
import os
import pyspark
from pyspark.sql import SQLContext, SparkSession
```

In [2]:
```
spark = SparkSession \
        .builder \
        .master('spark://192.168.68.106:7077') \
        .appName("sparkFromJupyter") \
        .getOrCreate()

sc = spark.sparkContext
sqlContext = SQLContext(sparkContext=sc, sparkSession=spark)
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/opt/spark/jars/spark-unsafe_2.12-3.2.
1.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
22/02/02 11:58:57 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-ja
va classes where applicable
```

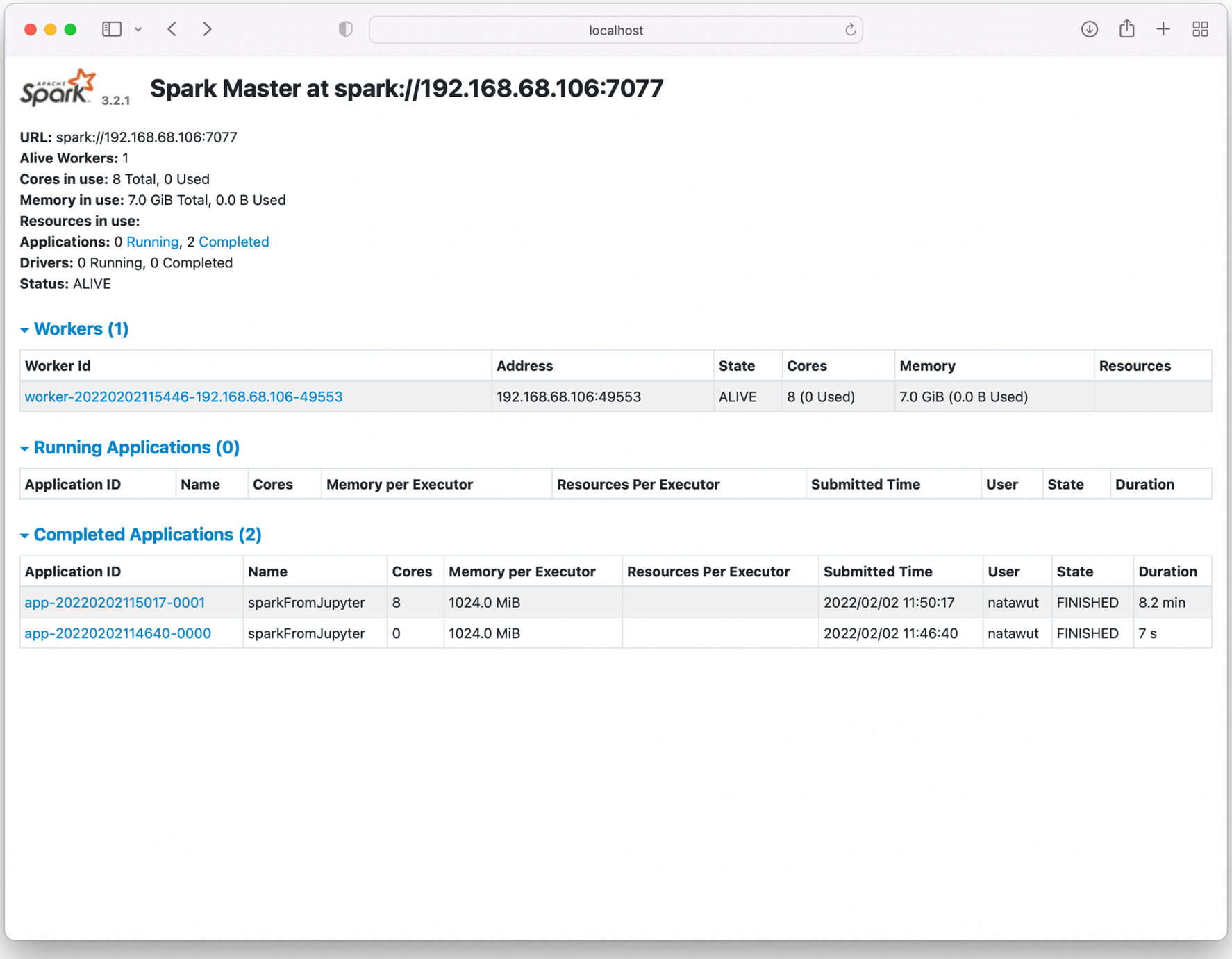
In [3]:
```
sw = sc.textFile('star-wars.txt')
words = sw.flatMap(lambda line: line.split())
mappers = words.map(lambda word: (word, 1))
counts = mappers.reduceByKey(lambda x, y: x+y)
```

In [4]:
```
results = counts.collect()
for row in results:
    print(row[0] + "," +str(row[1]))
```

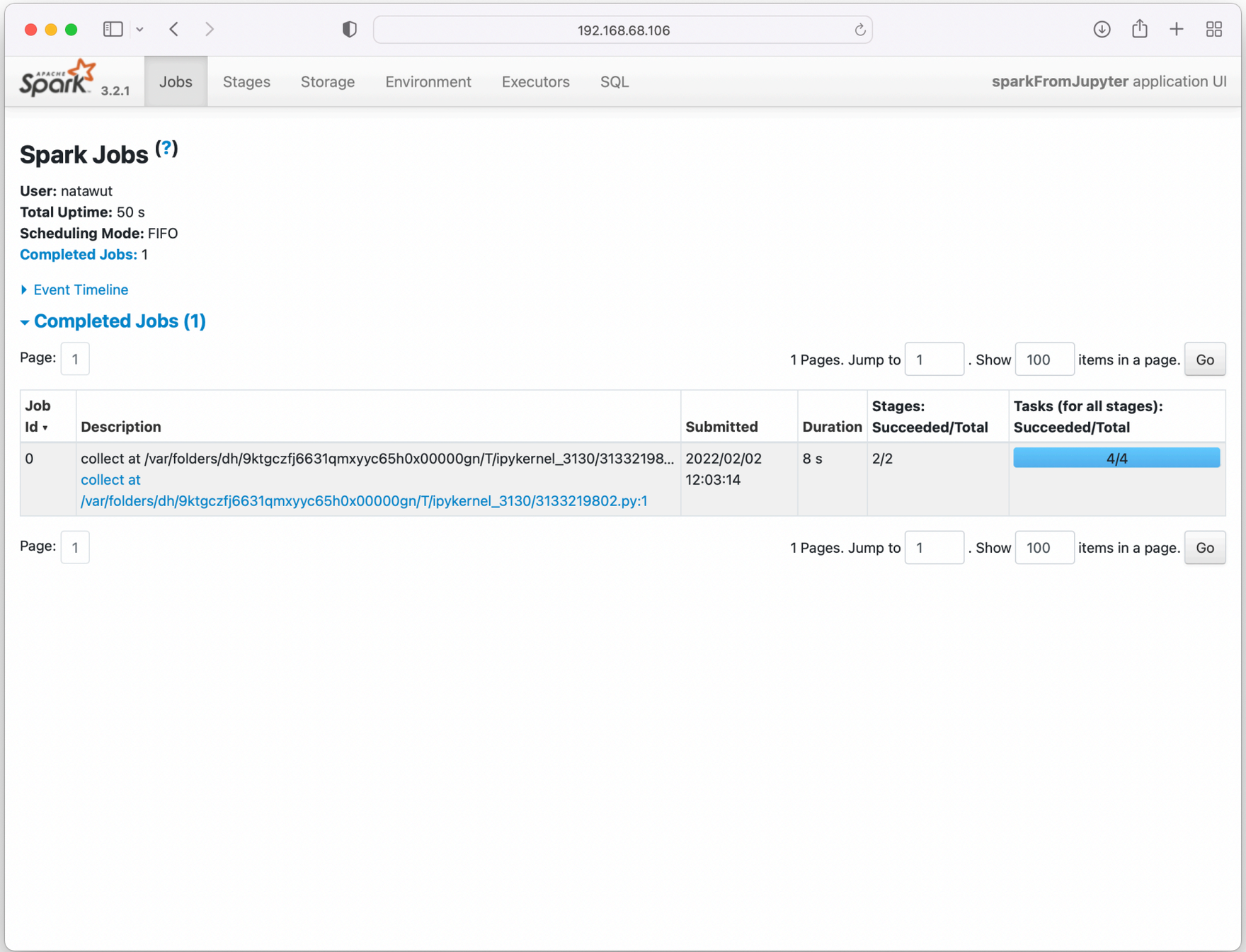
```
STAR,150
!!,2
PUBLIC,1
VERSION,1
A,1
long,31
ago,,1
in,379
far,,1
far,18
sea,2
of,579
as,211
drums,1
echo,1
heavens,2
crawls,1
into,147
```

In [5]:
```
spark.stop()
```

Spark Web UI (localhost:8080)



Spark Application Details (localhost:4040)



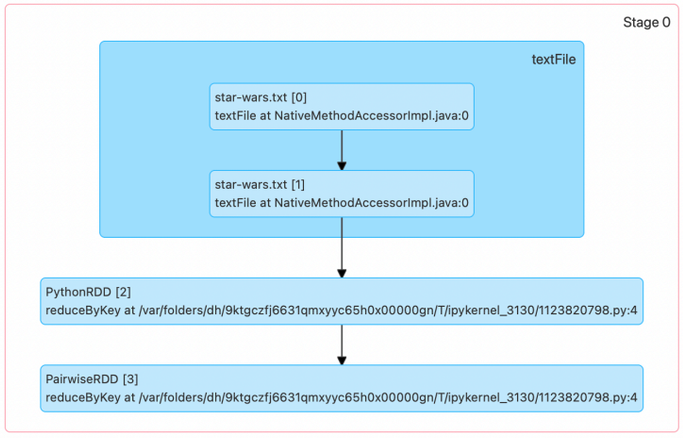
192.168.68.106

Spark 3.2.1 | Jobs | Stages | Storage | Environment | Executors | SQL | **sparkFromJupyter** application UI

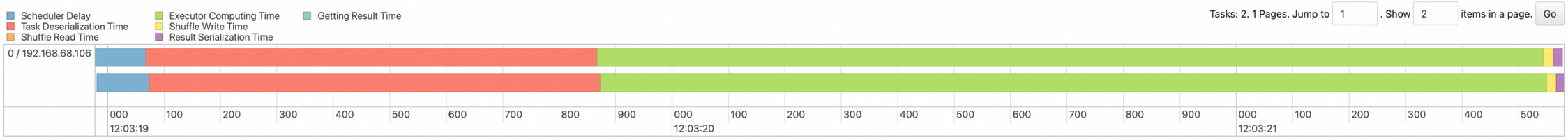
Details for Stage 0 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 3 s
Locality Level Summary: Process local: 2
Input Size / Records: 308.7 KiB / 7518
Shuffle Write Size / Records: 66.5 KiB / 48
Associated Job Ids: 0

▾ DAG Visualization



▸ Show Additional Metrics
▾ Event Timeline
☐ Enable zooming



Tasks: 2. 1 Pages. Jump to 1 . Show 2 items in a page. Go

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2 s	2 s	2 s	2 s	2 s
GC Time	10.0 ms	10.0 ms	10.0 ms	10.0 ms	10.0 ms
Input Size / Records	116.7 KiB / 3428	116.7 KiB / 3428	192 KiB / 4090	192 KiB / 4090	192 KiB / 4090
Shuffle Write Size / Records	30.6 KiB / 24	30.6 KiB / 24	35.9 KiB / 24	35.9 KiB / 24	35.9 KiB / 24

Internals of Job Execution In Spark

TechVidvan

Submit the code (jar files) and configured dependencies to Executors for further execution

Driver Program

```
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
```

Action

Request for worker nodes/Executers in the cluster

Cluster Manager

Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.

Worker Node

Executer

Task Task

Worker Node

Executer

Task Task

Worker Node

Executer

Task Task

SparkContext

DAG Scheduler

As the Action encounters, it will create the job

Job

RDD1 RDD2

RDD

RDD

Task Task Task Task

Task Scheduler

Tasks

Launch tasks via cluster manager

Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler

Submit the code (jar files) and configured dependencies to Executors for further execution

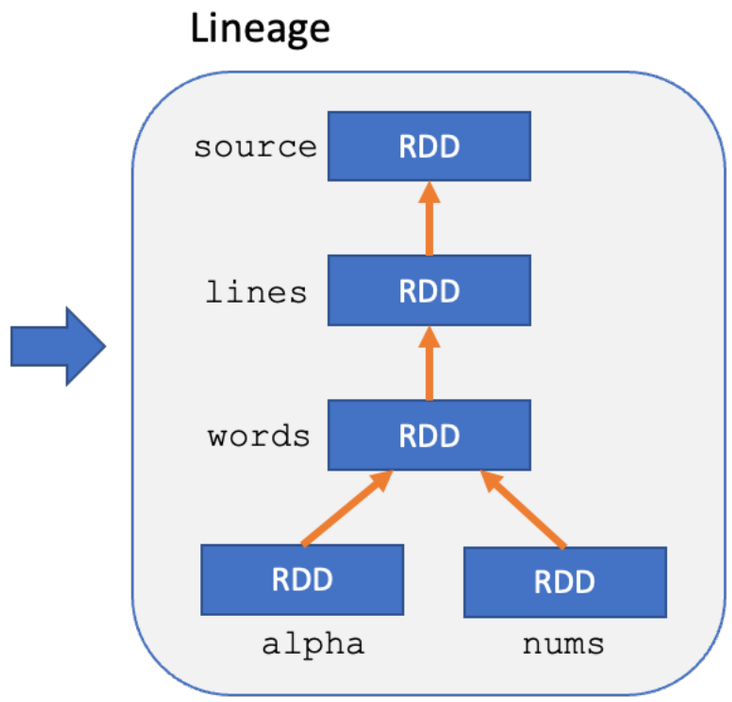
RDD Lineage

Transformations

```
source = sc.textFile("datafile.txt")
lines = source.map(lambda line: line.lower())
words = lines.flatMap(lambda line: line.split())
alpha = words.filter(lambda word: re.match(r'[a-z]+', word))
nums = words.filter(lambda word: re.match(r'[0-9]+', word))
```

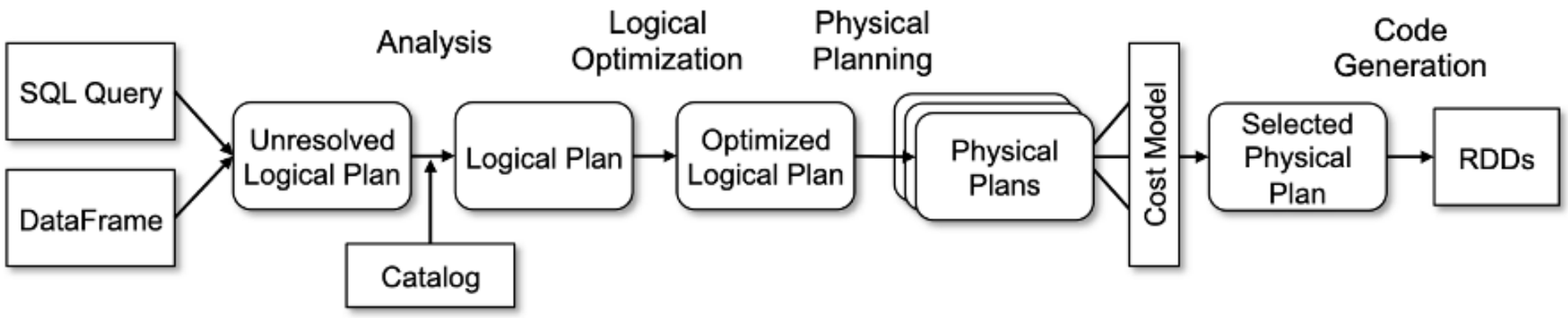
```
acount = alpha.count()
ncount = nums.count()
```

Actions



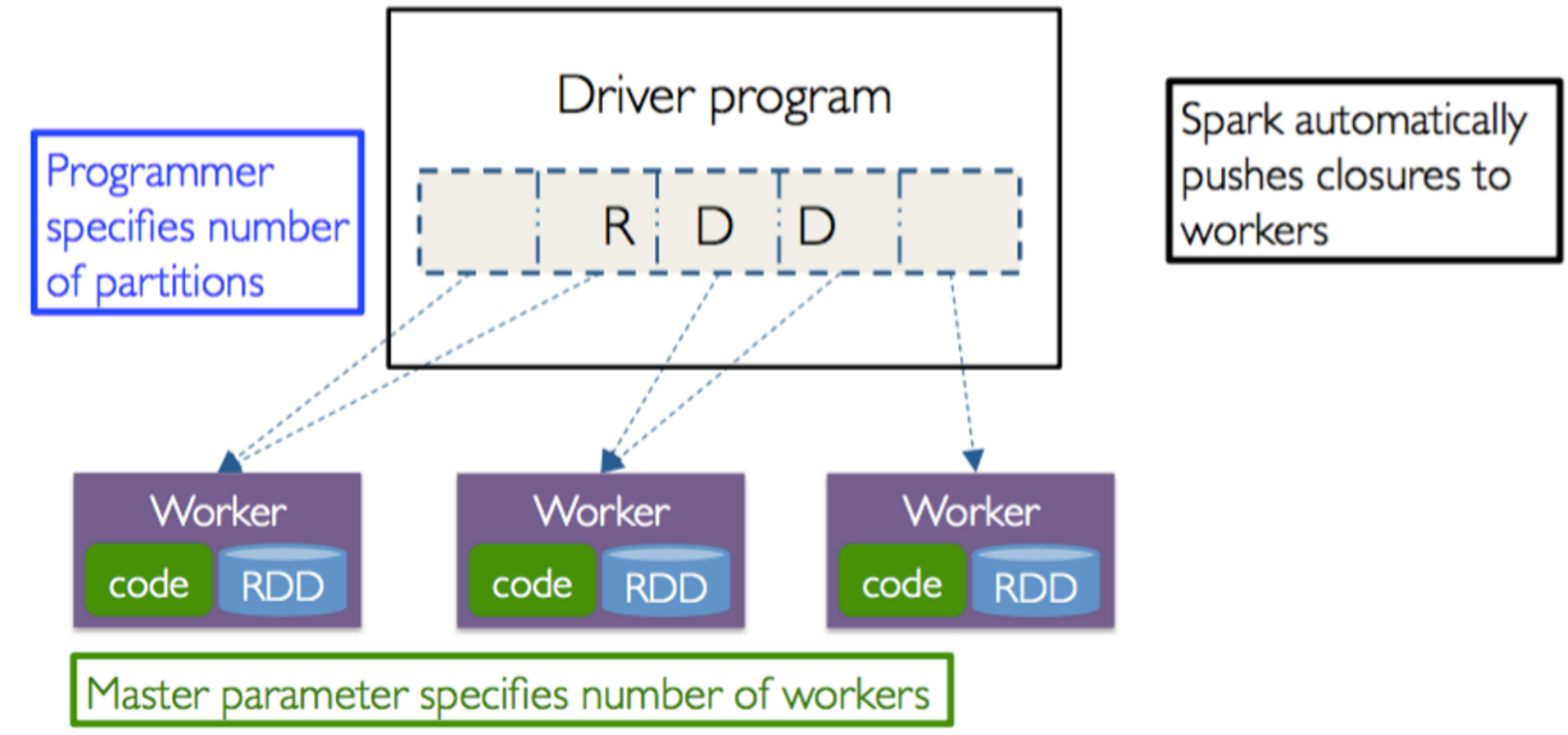
- RDD is “read-only”
- Spark is “lazy execution” = aggregate several commands and execution in batch

Execution Planning



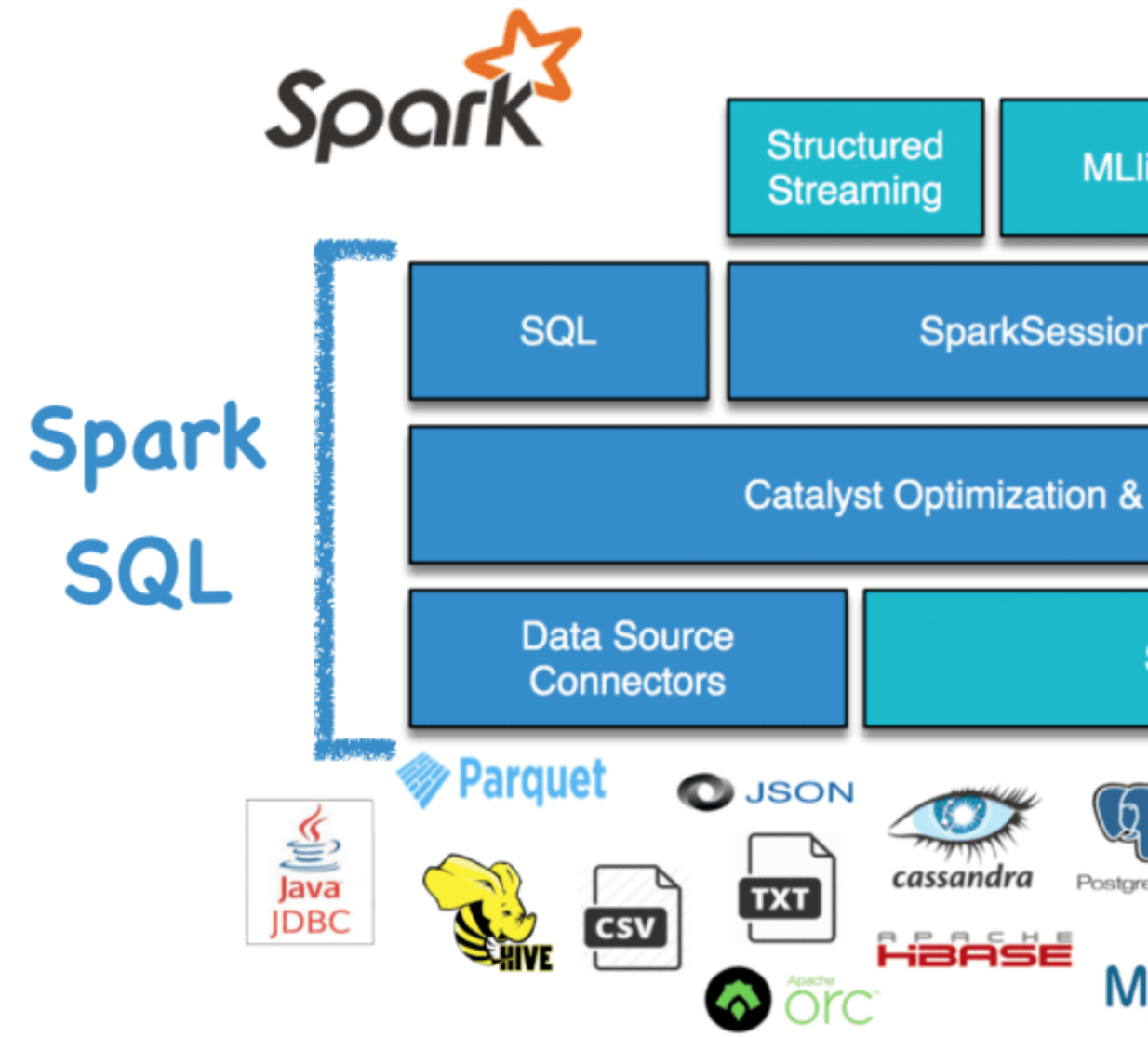
- Lazy execution allows Spark to perform “execution planning”
- Similar to database query execution
- Eliminate those wasteful execution e.g. no need to execute some commands for the entire data after “filter” command

RDD and Workers



Spark SQL and Pandas API

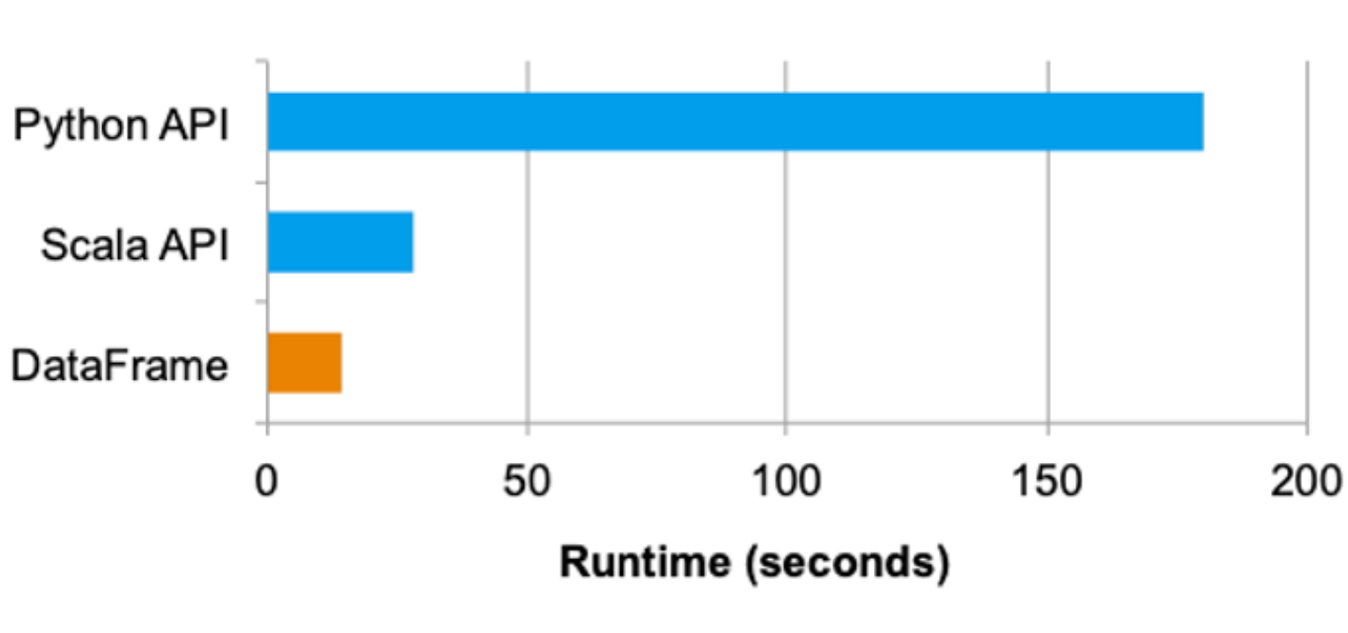
EDA the Spark Ways



https://rharshad.com/spark-sql-dataframes-datasets/

Spark SQL

- Spark SQL provides “SQL-Like” query capability on “Pandas-Like” Dataframe
- Provide Spark with more information about the structure of both the data and the computation being performed
- Allow better optimization, lead to better performance



M. Armbrust et al., “Spark sql: Relational data processing in spark,” in Proceedings of the 2015 ACM SIGMOD international conference on management of data, 2015, pp. 1383–1394

Dataframe - Schema RDD

Title	Genre	Premiere	Runtime	IMDB Score	Language
Enter the Anime	Documentary	August 5, 2019	58	2.5	English/Japanese
Dark Forces	Thriller	August 21, 2020	81	2.6	Spanish
The App	Science fiction/D...	December 26, 2019	79	2.6	Italian
The Open House	Horror thriller	January 19, 2018	94	3.2	English
Kaali Khuhi	Mystery	October 30, 2020	90	3.4	Hindi
Drive	Action	November 1, 2019	147	3.5	Hindi
Leyla Everlasting	Comedy	December 4, 2020	112	3.7	Turkish
The Last Days of ...	Heist film/Thriller	June 5, 2020	149	3.7	English
Paradox	Musical/Western/F...	March 23, 2018	73	3.9	English
Sardar Ka Grandson	Comedy	May 18, 2021	139	4.1	Hindi

https://www.kaggle.com/luiscorter/netflix-original-films-imdb-scores

- Two dimensional data structure
- Each column is similar to basic RDD and can have only one datatype
- Spark SQL provides sql-like operations on dataframe

Avocado Prices Dataset

- This dataset is a historical data on avocado prices and sales volume in multiple US markets, available at haggle.com (https://www.kaggle.com/datasets/neuromusic/avocado-prices)
  - Date - The date of the observation
  - AveragePrice - the average price of a single avocado
  - type - conventional or organic
  - year - the year
  - Region - the city or region of the observation
  - Total Volume - Total number of avocados sold
  - 4046 - Total number of avocados with PLU 4046 sold
  - 4225 - Total number of avocados with PLU 4225 sold
  - 4770 - Total number of avocados with PLU 4770 sold

Basic Spark SQL Commands

We can select some columns using **'select'** and select some rows using **'filter'**. Note that we can perform basic math to columns.

In [13]:
```
df.select(df['Date'], df['AveragePrice'], df['Total_Bags'], df['year'], df['region']).show(5)
```

```
+----------+-----------+----------+----+------+
|      Date|AveragePrice|Total_Bags|year|region|
+----------+-----------+----------+----+------+
|2015-12-27|       1.33|   8696.87|2015|Albany|
|2015-12-20|       1.35|   9505.56|2015|Albany|
|2015-12-13|       0.93|   8145.35|2015|Albany|
|2015-12-06|       1.08|   5811.16|2015|Albany|
|2015-11-29|       1.28|   6183.95|2015|Albany|
+----------+-----------+----------+----+------+
only showing top 5 rows
```

In [14]:
```
df.select(df['Date'], df['Small_Bags']+df['Large_Bags']+df['XLarge_Bags'],
          df['Total_Bags']).show(5)
```

```
+----------+----------------------------------------+----------+
|      Date|((Small_Bags + Large_Bags) + XLarge_Bags)|Total_Bags|
+----------+----------------------------------------+----------+
|2015-12-27|                                 8696.87|   8696.87|
|2015-12-20|                                 9505.56|   9505.56|
|2015-12-13|                                 8145.35|   8145.35|
|2015-12-06|                                 5811.16|   5811.16|
|2015-11-29|                                 6183.95|   6183.95|
+----------+----------------------------------------+----------+
only showing top 5 rows
```

In [15]:
```
df.select(df['Date'], df['Total_Bags'], df['Total_Volume'],
          df['Total_Volume']/df['Total_Bags']).show(5)
```

```
+----------+----------+------------+--------------------------+
|      Date|Total_Bags|Total_Volume|(Total_Volume / Total_Bags)|
+----------+----------+------------+--------------------------+
|2015-12-27|   8696.87|    64236.62|          7.386176865929926|
```

Select and Filter Command

- Select is similar to SQL select command
  - select a single or multiple columns
  - select from list of column names and by column index
- Filter (or where) is very similar to SQL where condition
  - filter with column condition and multiple columns (similar to pandas)
  - filter with SQL expression
  - ilter with string expression e.g. startswith, endswith, contains, etc.
- See https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/ and https://sparkbyexamples.com/pyspark/pyspark-where-filter/ for more details

Basic Spark SQL Commands

We can select some columns using **'select'** and select some rows using **'filter'**. Note that we can perform basic math to columns.

In [13]:
```
df.select(df['Date'], df['AveragePrice'], df['Total_Bags'], df['year'], df['region']).show(5)
```

```
+----------+-----------+----------+----+------+
|      Date|AveragePrice|Total_Bags|year|region|
+----------+-----------+----------+----+------+
|2015-12-27|       1.33|   8696.87|2015|Albany|
|2015-12-20|       1.35|   9505.56|2015|Albany|
|2015-12-13|       0.93|   8145.35|2015|Albany|
|2015-12-06|       1.08|   5811.16|2015|Albany|
|2015-11-29|       1.28|   6183.95|2015|Albany|
+----------+-----------+----------+----+------+
only showing top 5 rows
```

In [14]:
```
df.select(df['Date'], df['Small_Bags']+df['Large_Bags']+df['XLarge_Bags'],
          df['Total_Bags']).show(5)
```

```
+----------+-----------------------------------------+----------+
|      Date|((Small_Bags + Large_Bags) + XLarge_Bags)|Total_Bags|
+----------+-----------------------------------------+----------+
|2015-12-27|                                  8696.87|   8696.87|
|2015-12-20|                                  9505.56|   9505.56|
|2015-12-13|                                  8145.35|   8145.35|
|2015-12-06|                                  5811.16|   5811.16|
|2015-11-29|                                  6183.95|   6183.95|
+----------+-----------------------------------------+----------+
only showing top 5 rows
```

In [15]:
```
df.select(df['Date'], df['Total_Bags'], df['Total_Volume'],
          df['Total_Volume']/df['Total_Bags']).show(5)
```

```
+----------+----------+------------+--------------------------+
|      Date|Total_Bags|Total_Volume|(Total_Volume / Total_Bags)|
+----------+----------+------------+--------------------------+
|2015-12-27|   8696.87|    64236.62|          7.386176865929926|
```

Spark SQL Aggregate, Groupby, and User-Defined Functions

- Spark SQL has several built-in aggregate functions e.g. min, max, avg, std, sum, sumDistinct, count, count_if, first, last, etc.

- Spark SQL supports groupby operation, which is similar to groupby in pandas

- User-defined function can be applied in select and withColumn commands

Using group and groupby functions

In [23]:
```
from pyspark.sql.functions import avg, min, max
```

In [24]:
```
df.select(min('AveragePrice'), avg('AveragePrice'), max('AveragePrice')).show()
```

```
+-----------------+------------------+-----------------+
|min(AveragePrice)| avg(AveragePrice)|max(AveragePrice)|
+-----------------+------------------+-----------------+
|             0.44|1.4059784097758825|             3.25|
+-----------------+------------------+-----------------+
```

In [25]:
```
df.filter('region == "SanDiego"').select(avg('AveragePrice')).show()
```

```
+------------------+
| avg(AveragePrice)|
+------------------+
|1.3981656804733738|
+------------------+
```

Groupby function allows us to work data in groups.

In [26]:
```
df.groupby('type').count().show()
```

```
+------------+-----+
|        type|count|
+------------+-----+
|     organic| 9123|
|conventional| 9126|
+------------+-----+
```

In [27]:
```
df.groupby('year', 'type').agg({'AveragePrice': 'avg'}).orderBy('year', 'type').show()
```

```
+----+------------+------------------+
|year|        type| avg(AveragePrice)|
+----+------------+------------------+
```

Spark Pandas API

- PySpark provides Pandas API, which is fully compatible with Python Pandas

- This simplifies EDA process e.g. data manipulation, exploring data distribution, plotting graphs

- Spark SQL (ML Dataframe-Based API) is still needed for ML modeling as it is the only way to do it

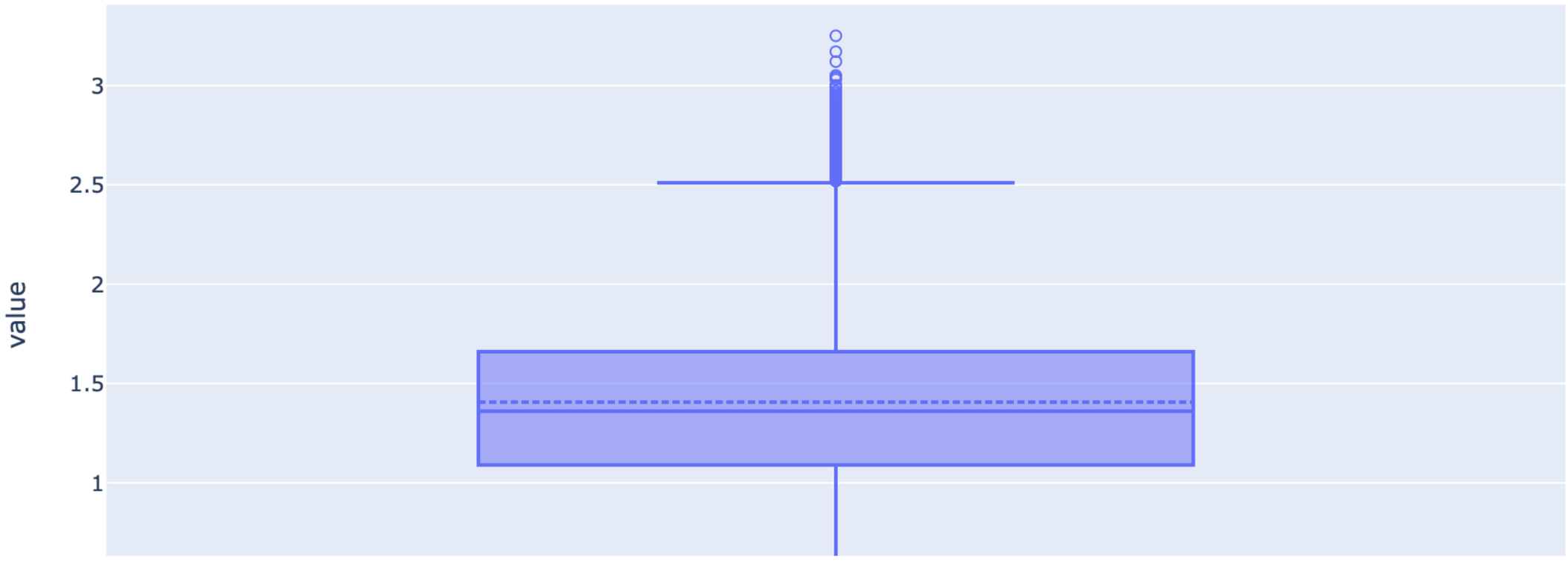
In [33]:
```
pdf = ps.DataFrame(df)
```

In [34]:
```
pdf.head()
```

Out[34]:

	Id	Date	AveragePrice	Total_Volume	4046	4225	4770	Total_Bags	Small_Bags	Large_Bags	XLarge_Bags	type	year	region
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25	0.0	conventional	2015	Albany
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49	0.0	conventional	2015	Albany
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14	0.0	conventional	2015	Albany
3	3	2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76	0.0	conventional	2015	Albany
4	4	2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69	0.0	conventional	2015	Albany

In [35]:
```
pdf.AveragePrice.plot.box()
```



Spark ML

MLlib DataFrame-based API

Machine Learning with Apache Spark

https://pragmaedge.com/beginners-guide-to-machine-learning-with-apache-spark/

Spark ML

- Spark ML is a DataFrame-based API over Spark SQL with several ML tools, including:
  - ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
  - Featurization: feature extraction, transformation, dimensionality reduction, and selection
  - Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
  - Persistence: saving and load algorithms, models, and Pipelines
  - Utilities: linear algebra, statistics, data handling, etc.

Spark ML Concepts

- The base data structure of Spark ML is DataFrame, containing raw data, features, feature vectors, true labels, and predictions

- Spark ML manipulates DataFrames with pipeline, which is a chain of operations with one or more stages

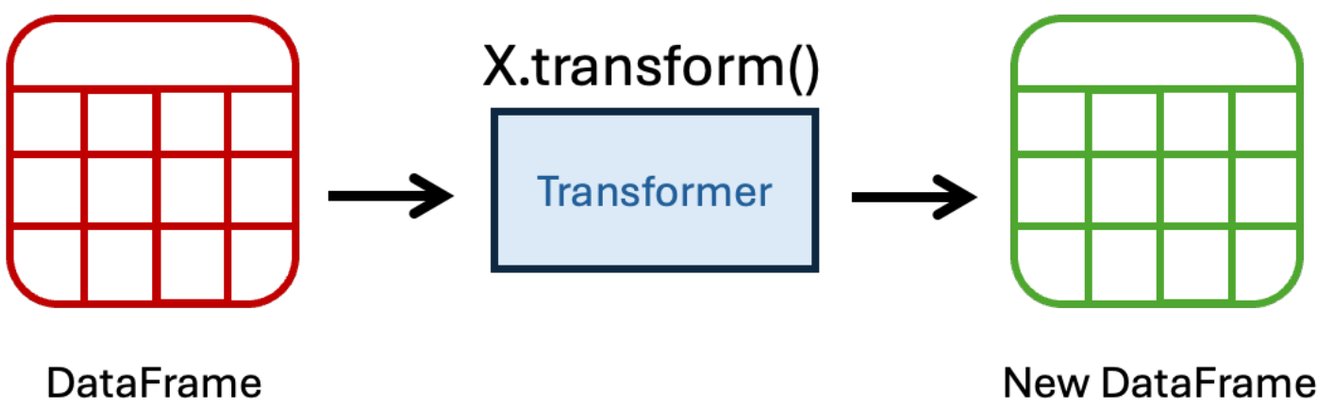
- Each pipeline stage can be either “Transformer” or “Estimator”

  - Transformer — transform DataFrame to DataFrame e.g. ML model is a Transformer — from features (dataframe) to predictions (dataframe)

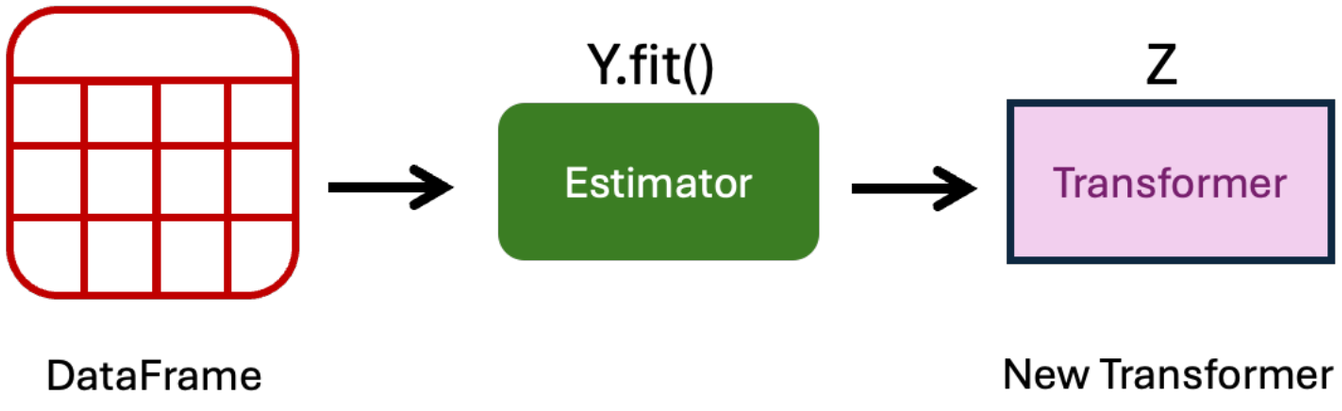
  - Estimator — fit on a DataFrame to produce a Transformer e.g. a learning algorithm is an Estimator — trains on a dataframe to produce a model

  - Estimator learns parameters from training data and create a transformer with those parameters / transformer uses pre-defined values of parameters

Understanding Transformer and Estimator



insert, transform, combine, remove columns

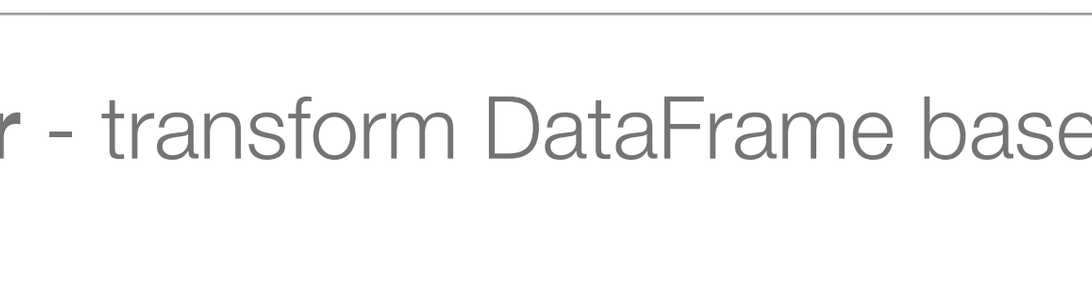


create a new ML model or custom operation

Example: Transformers in Avocado Prices Prediction

SQLTransfomer - transform DataFrame based on given SQL statement

Date	Average Price	Total Volume	Small_Bags	...



AveragePrice	LOG_Small_Bags	month

```
SELECT `AveragePrice`, LOG(`Small_Bags`+1) AS `LOG_Small_Bags`,
MONTH(__THIS__.Date) AS month FROM __THIS__
```

Numerical Feature Transformers

sql_transformer: numeric column selection and log-transform

Create a transformer to select columns and log-transform some numerical columns

In [ ]:
```
cols = ['AveragePrice', 'type']
cols = [f"`{col}`" for col in cols]
cols
```

In [ ]:
```
log_cols =  ['4225', '4770', 'Small_Bags', 'Large_Bags', 'XLarge_Bags']
log_cols = [f"LOG(`{col}`+1) AS `LOG_{col}`" for col in log_cols]
log_cols
```

In [ ]:
```
statement = f"""SELECT{', '.join(cols)}, {', '.join(log_cols)},
    YEAR(__THIS__.Date)-2000 AS year, MONTH(__THIS__.Date) AS month
    FROM __THIS__
    """
statement
```

In [ ]:
```
sql_transformer = SQLTransformer(statement=statement)
```

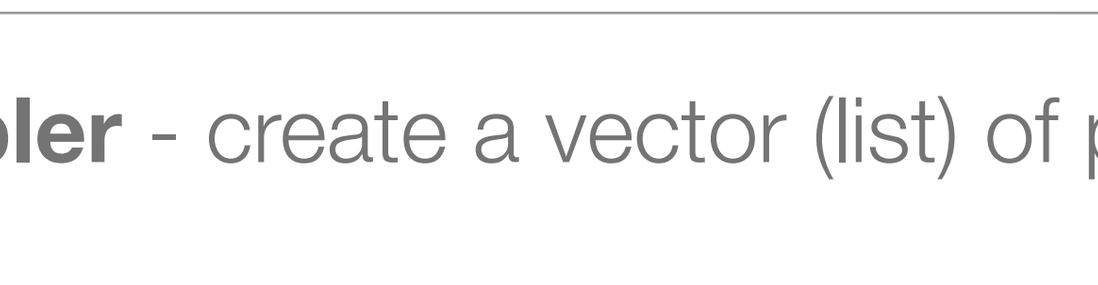
In [ ]:
```
df_avocado_train.show(4)
```

In [ ]:
```
sql_transformer.transform(df_avocado_train).show(4)
```

Example: Transformers in Avocado Prices Prediction

VectorAssembler - create a vector (list) of parameters

AveragePrice	LOG_Small_Bags	month
1.04	0.38	4



Average Price	LOG_Small_Bags	month	features_num
1.04	0.38	4	**[0.38,4]**

```
inputCols=[‘month','LOG_Small_Bags'], outputCol='features_num'
```

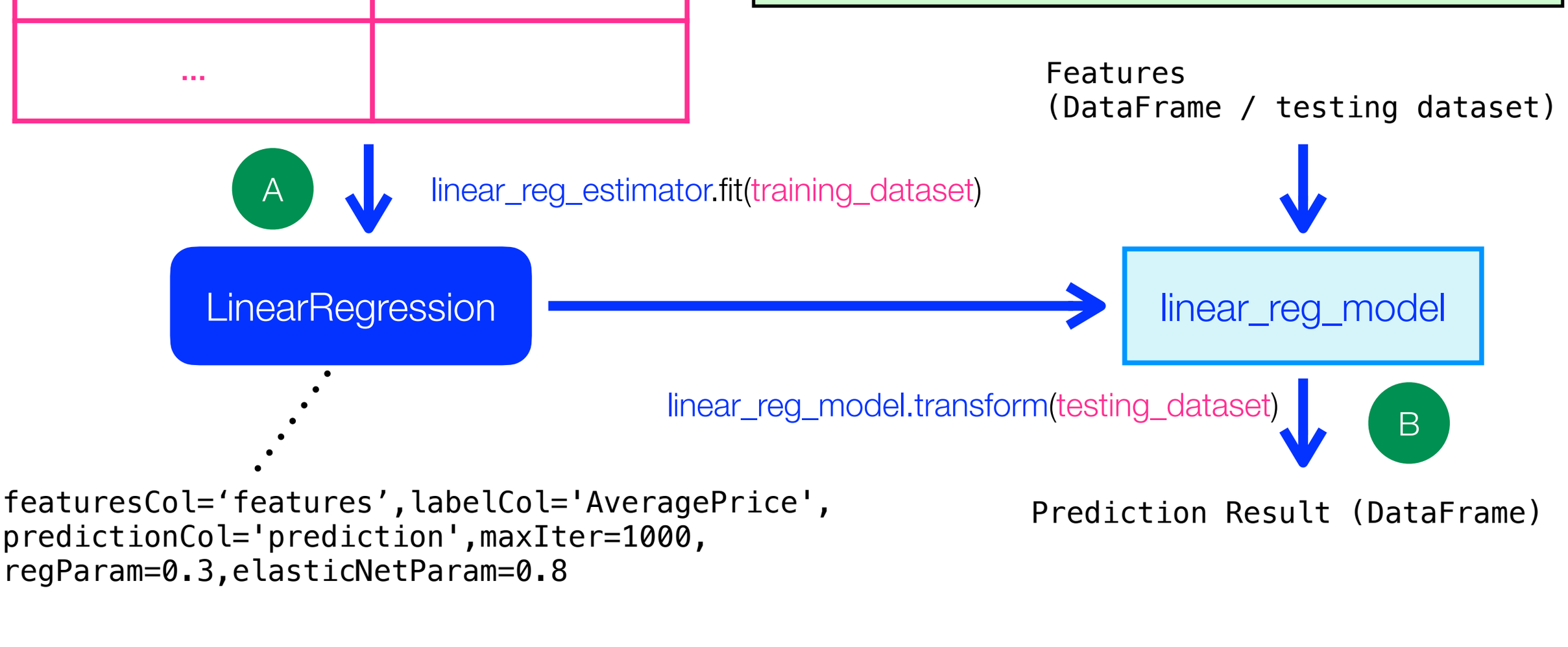
Example: Estimators in Avocado Prices Prediction

LinearRegression - algorithm to train linear regression model from data

training_dataset

features	AveragePrice
[0.38,4]	1.04
...	

A. Fitting LinearRegession estimator with training dataset to create Linear Regression Model (linear_reg_model)
B. linear_reg_model.transform(testing_dataset) to inference the testing dataset



Spark ML Pipeline

- A pipeline is an estimator that handle stages of transformers
- We create a model from training dataset

```
pipeline = Pipeline(stages=[sql_transformer,
                            month_vec_asm_transformer, ...])
pipe_model = pipeline.fit(train_df)
```

- We use model (transformer) to transform testing dataset

```
results_df = pipe_model.transform(test_df)
```

sql_transformer
month_vec_asm_transformer
month_scaler_transformer
numerical_vec_asm_transformer
std_scaler_transformer
type_indexer_transformer
categorical_vec_asm_transformer
all_vec_asm_transformer

Spark ML: Predicting Avocado Prices

This notebook introduces how to train a ML model using Spark ML. This bases on an excellent article in Towards Data Science First Steps in Machine Learning with Apache Spark using Avocado Prices dataset in Kaggle.

The objective of this model is to predict the average price of avocado given datetime, supply amounts, and region.

Spark Cluster Preparation

In [ ]:
```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False
```

In [ ]:
```
if IN_COLAB:
    !apt-get install openjdk-8-jdk-headless -qq > /dev/null
    !wget -q https://dlcdn.apache.org/spark/spark-3.3.2/spark-3.3.2-bin-hadoop3.tgz
    !tar xf spark-3.3.2-bin-hadoop3.tgz
    !mv spark-3.3.2-bin-hadoop3 spark
    !pip install -q findspark
    import os
    os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
    os.environ["SPARK_HOME"] = "/content/spark"
```

In [ ]:
```
import findspark
findspark.init()
```

In [ ]:
```
spark_url = 'local'
```

In [ ]:
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
```

In [ ]:
```
spark = SparkSession.builder\
```

Conclusion

- Spark is a popular platform, especially for data science
- It enables flexible programming structure, not just “map” and “reduce”
- Spark provides in-memory processing along with lazy execution, RDD lineage, and execution planning makes Spark very efficient
- DataFrame in Spark SQL and ML pipeline in Spark ML are the mainstream features as they are more flexible and more user-friendly than RDD in Spark Core

References

- A. Joseph, “Introduction to Big Data with Apache Spark”, https://www.edx.org/course/introduction-big-data-apache-spark-uc-berkeleyx-cs100-1x

- Spark Programming Guide, http://spark.apache.org/docs/latest/programming-guide.html

- M. Zaharia, Learning Spark, O'Reilly Media, February 2015