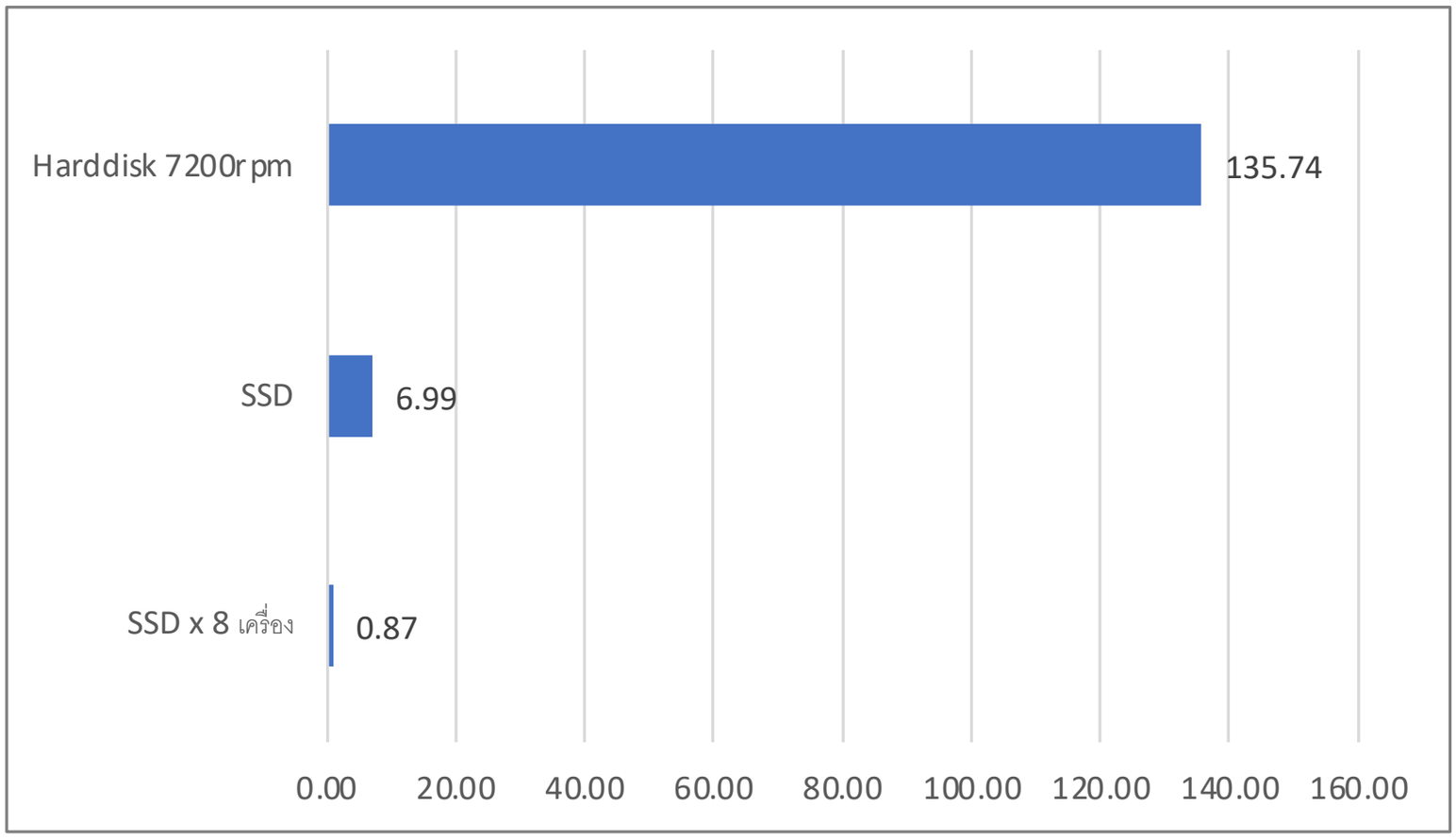
2110446 - Data Science and Data Engineering

Big Data Processing with Spark

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

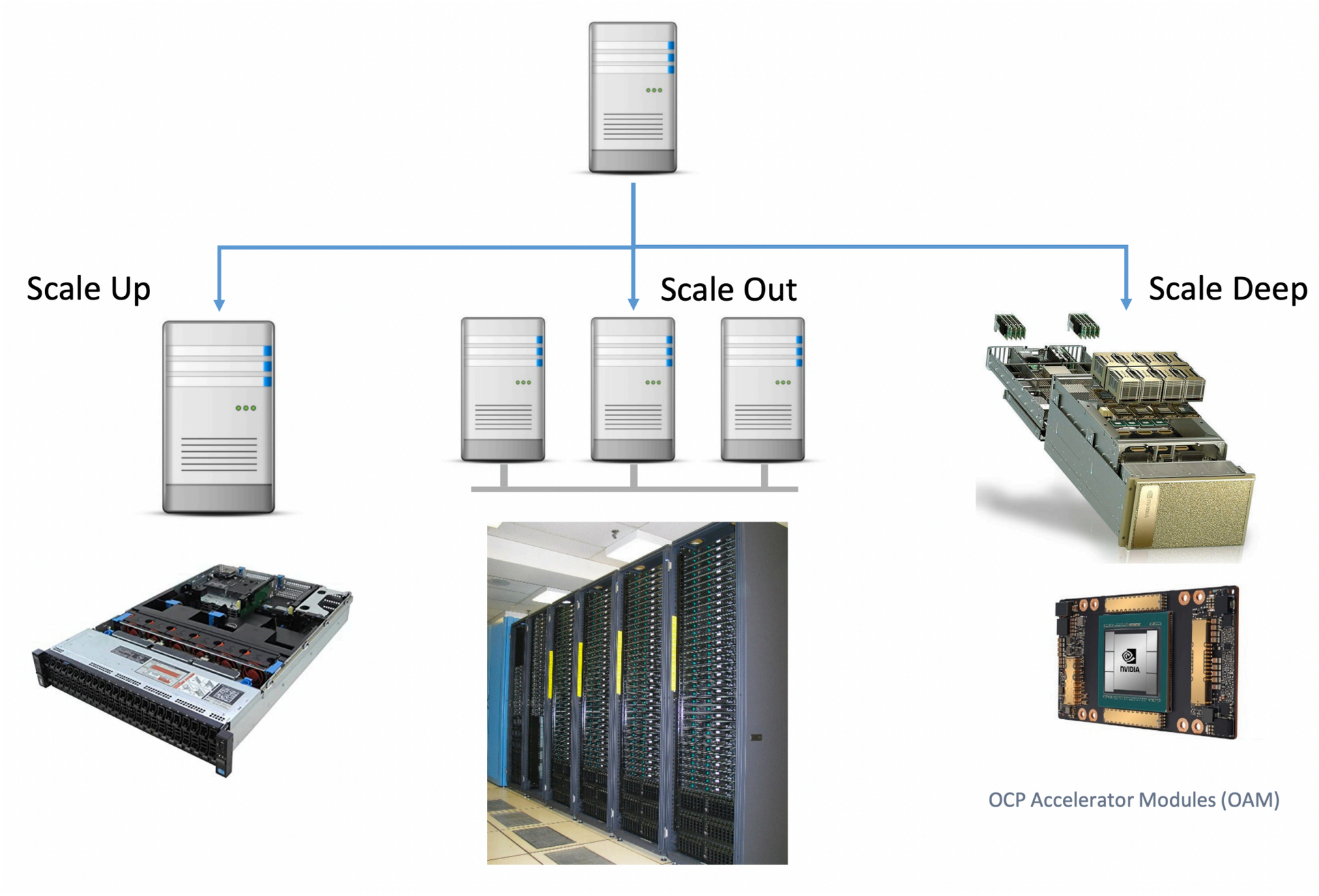
CHULA ENGINEERING
Foundation toward Innovation

Limitations of Disk Data Transfer Bandwidth

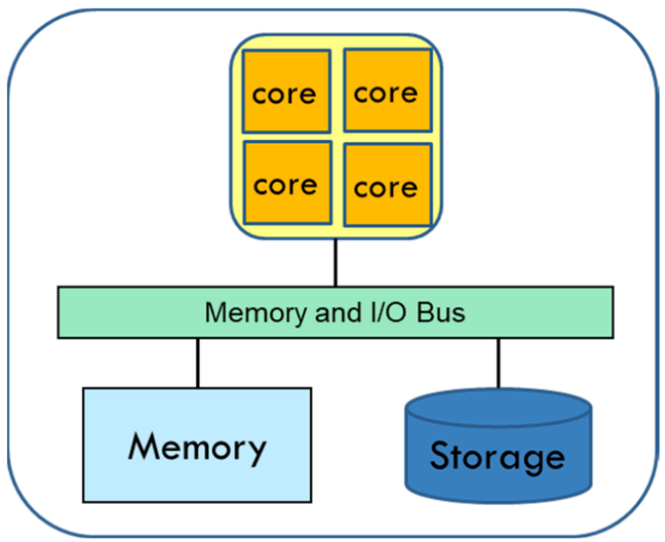


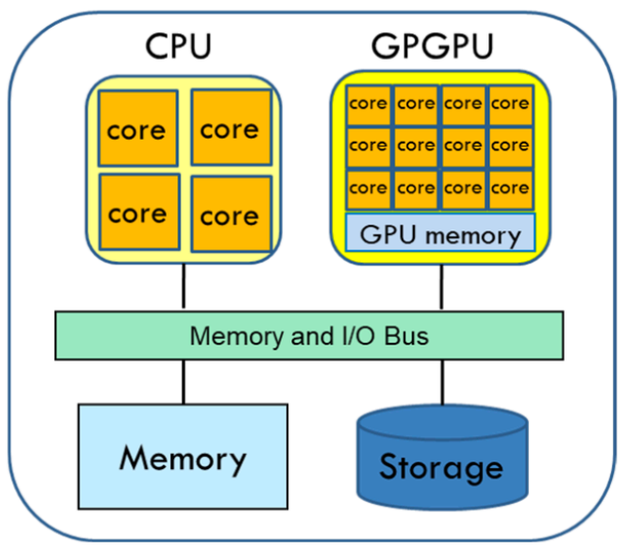
Theoretical time (in minutes) to read 1-TB data from disks

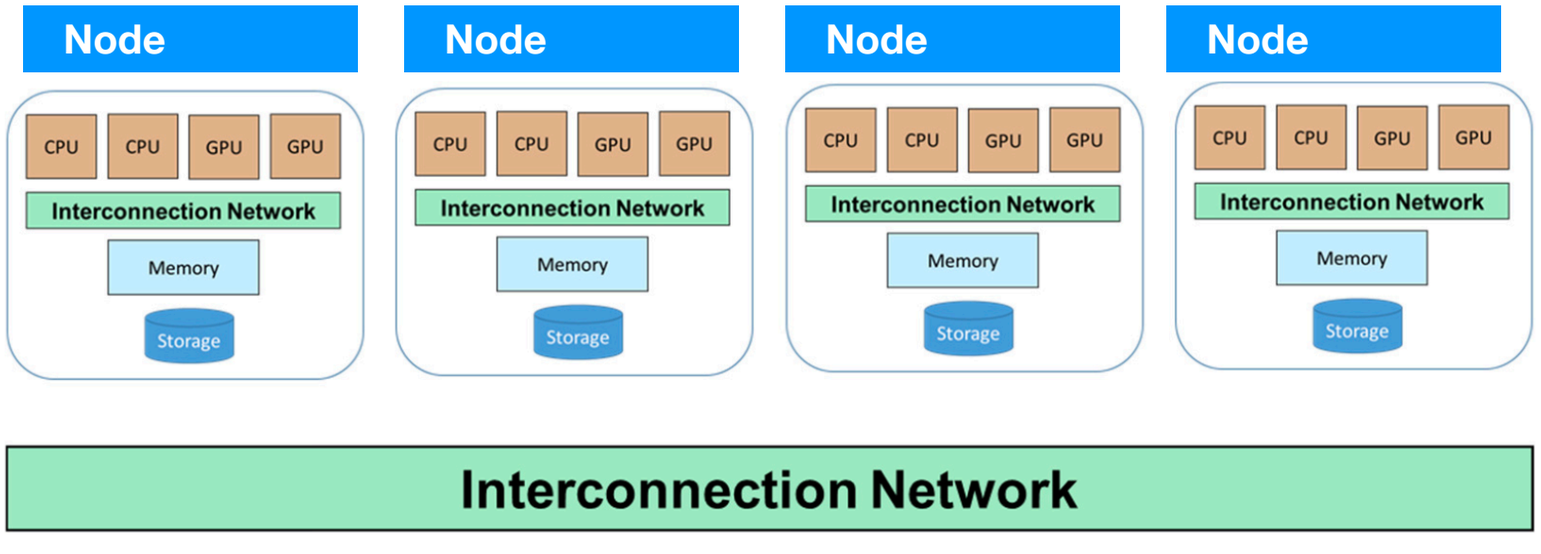
Scale Up / Scale Out / Scale Deep



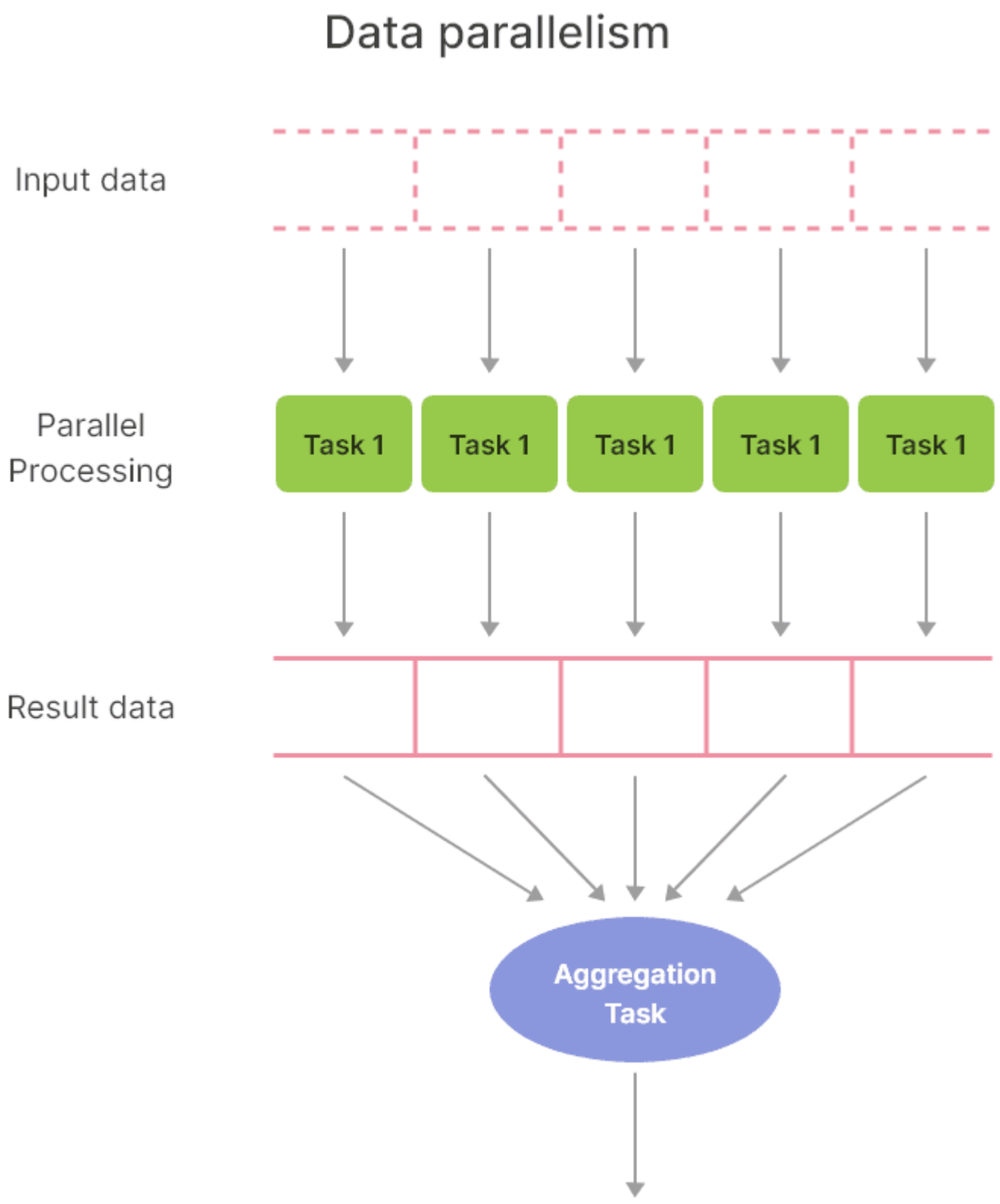
Cluster Computing







Big Question: How to utilize cluster of computers for Big Data processing?



Source: https://bytewax.io/blog/data-parallel-task-parallel-and-agent-actor-architectures

Early Days of Big Data - Apache Hadoop

- Opensource software framework inspired by Google Search Engine Architecture

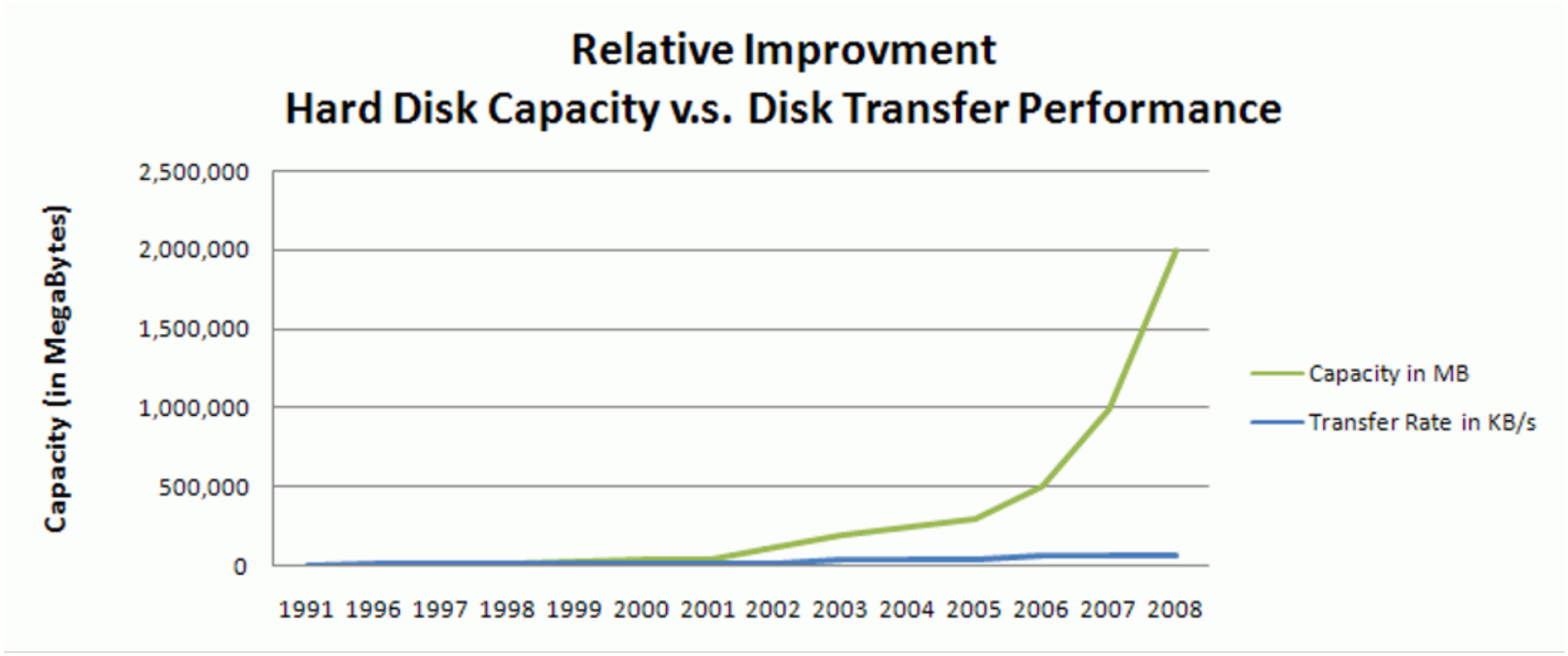
- Provide easy-to-program scale-out foundation for data-intensive applications on large clusters of commodity hardware

- 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



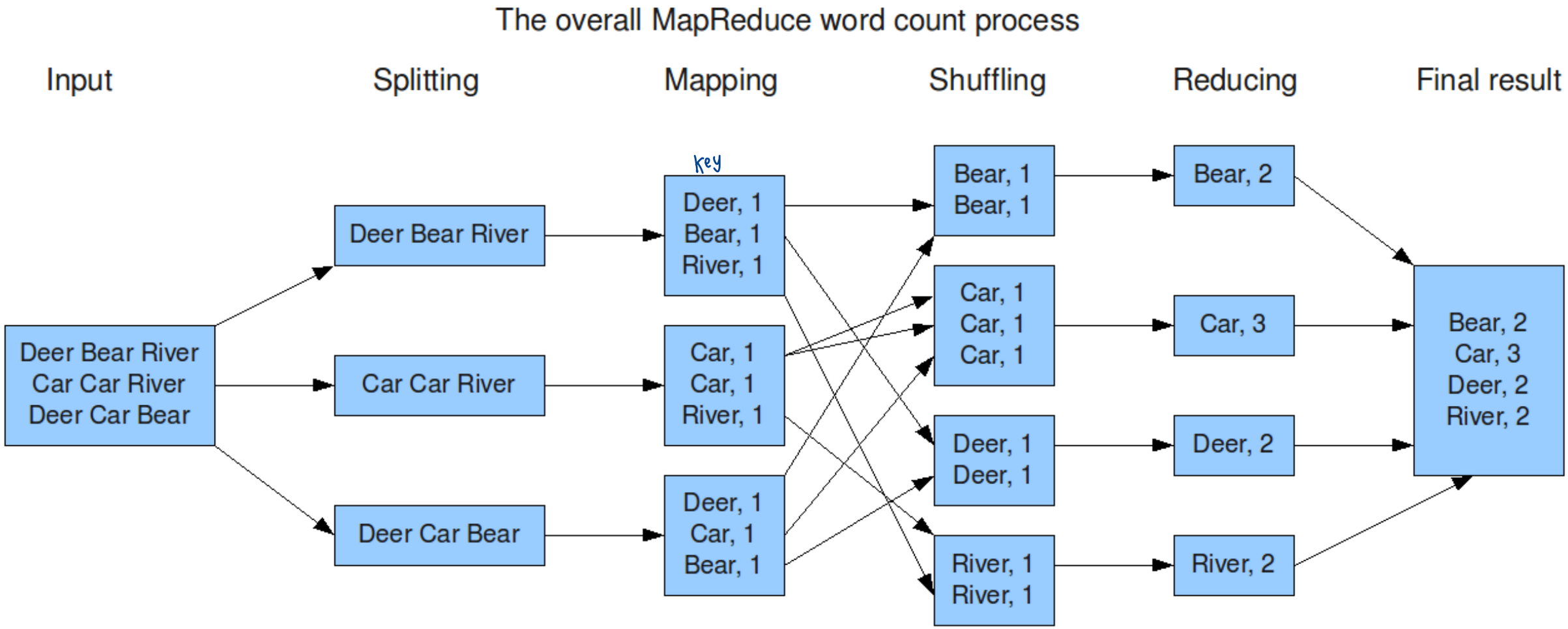
Source: http://wiki.r1soft.com/pages/viewpage.action?pageId=3016608

- Disk data transfer improves very slowly
- Solution: scale-out with cluster – processing data on multiple disks in parallel

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

Map-Reduce focuses on data processing pattern



Source: https://www.todaysoftmag.com/article/1358/hadoop-mapreduce-deep-diving-and-tuning

Hadoop Limitations

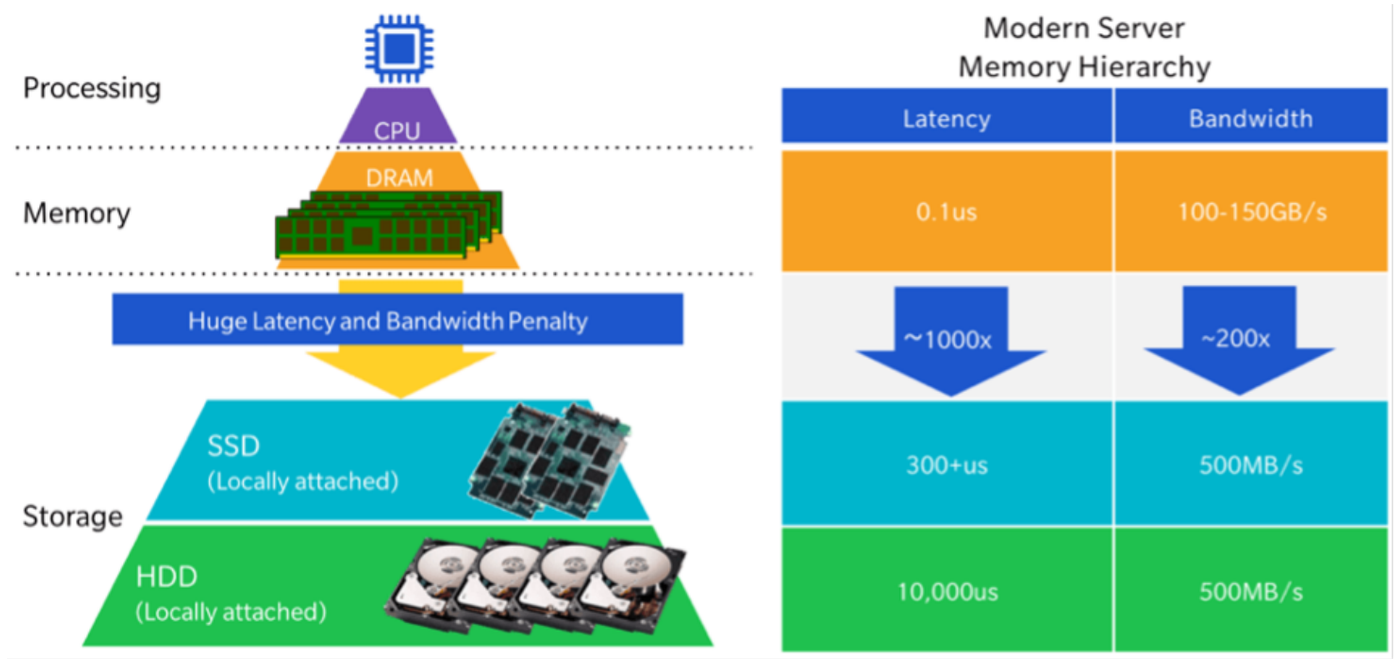
- Hadoop is disk-oriented เขียนลง disk
- Suitable for Batch System; No real-time, repetitive queries
- Data Scientist = Analyze, Discover, Investigate

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)

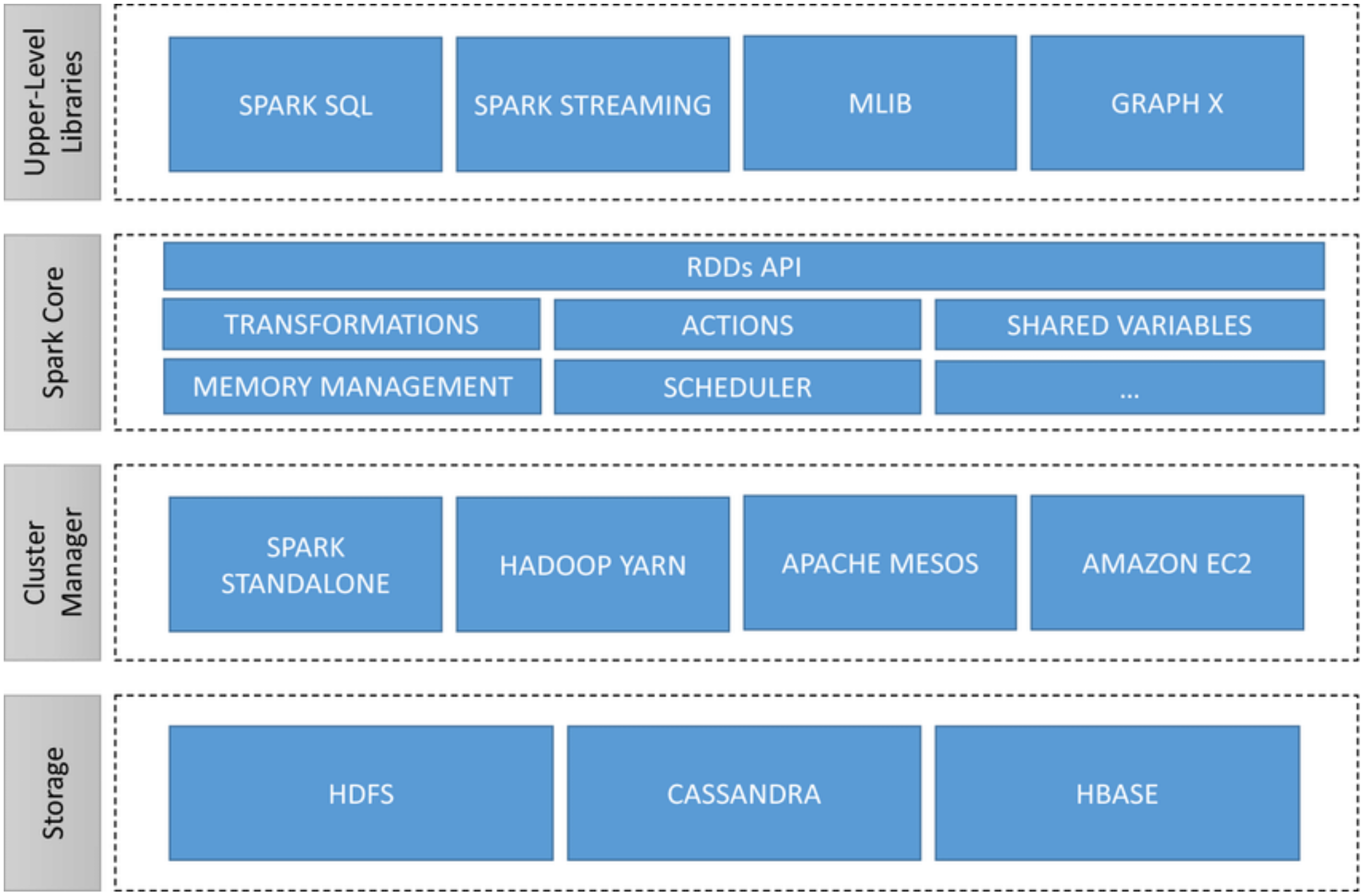
Spark's Principles

- 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



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**



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

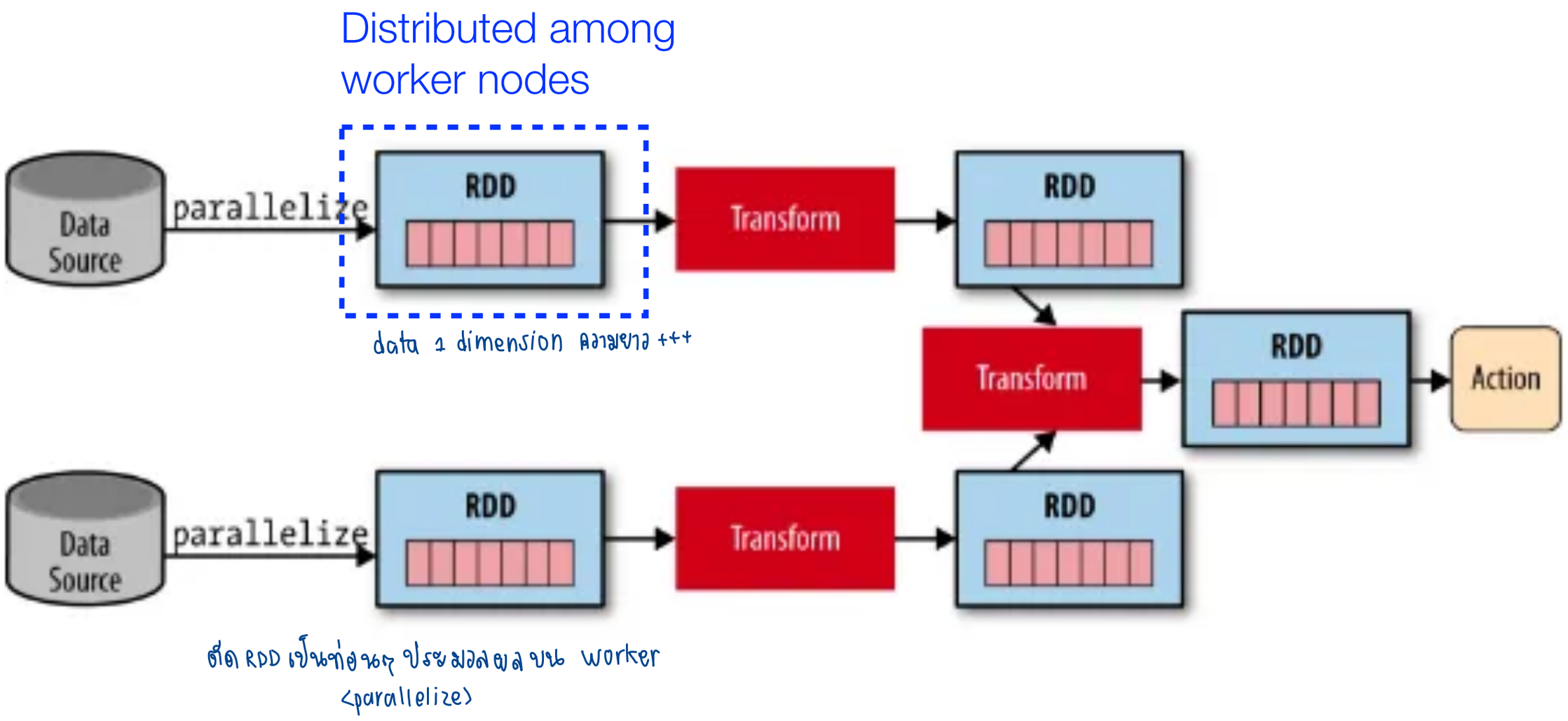
SQL analytics and BI

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™

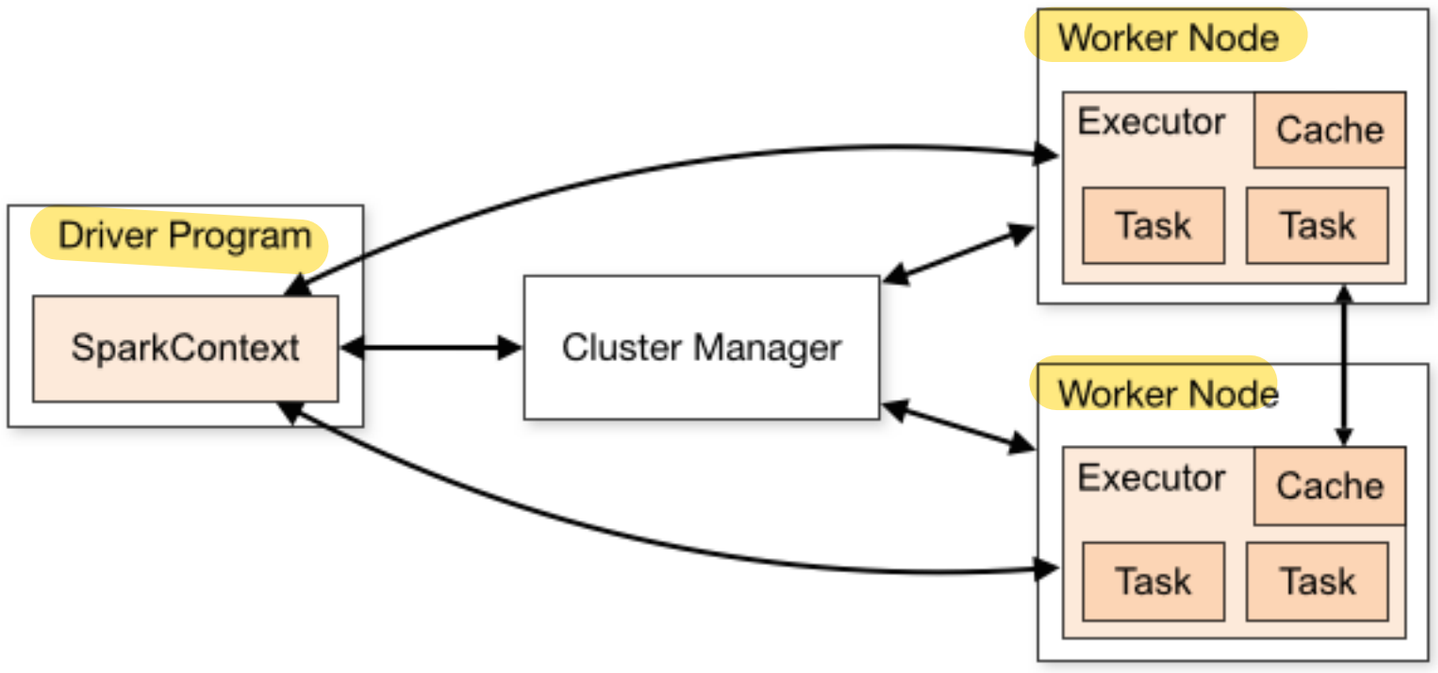
Spark focuses on data abstraction with flexible processing patterns



Source: https://medium.com/@gkadam2011/beneath-rdd-resilient-distributed-dataset-in-apache-spark-260c0b7250c6

Spark 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



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 in **standalone mode**

- Running pyspark

  pyspark

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

- References
  - Linux and mac

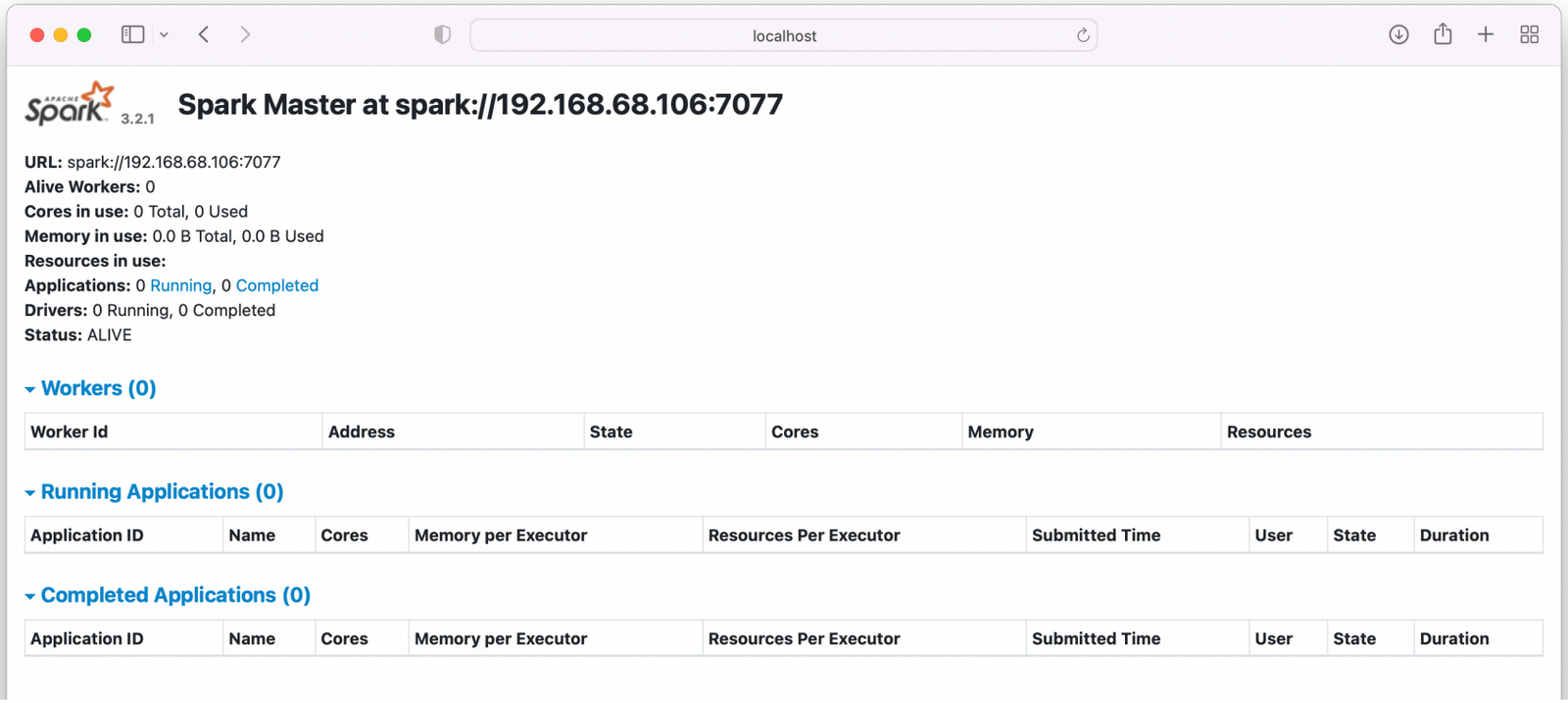
    https://medium.com/@GalarnykMichael/install-spark-on-ubuntu-pyspark-231c45677de0
  - Windows
    - https://medium.com/@GalarnykMichael/install-spark-on-windows-pyspark-4498a5d8d66c
  - Using pip (Linux, mac, and windows)
    - http://sigdelta.com/blog/how-to-install-pyspark-locally/

Running pyspark in **master-worker mode**

- 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
```

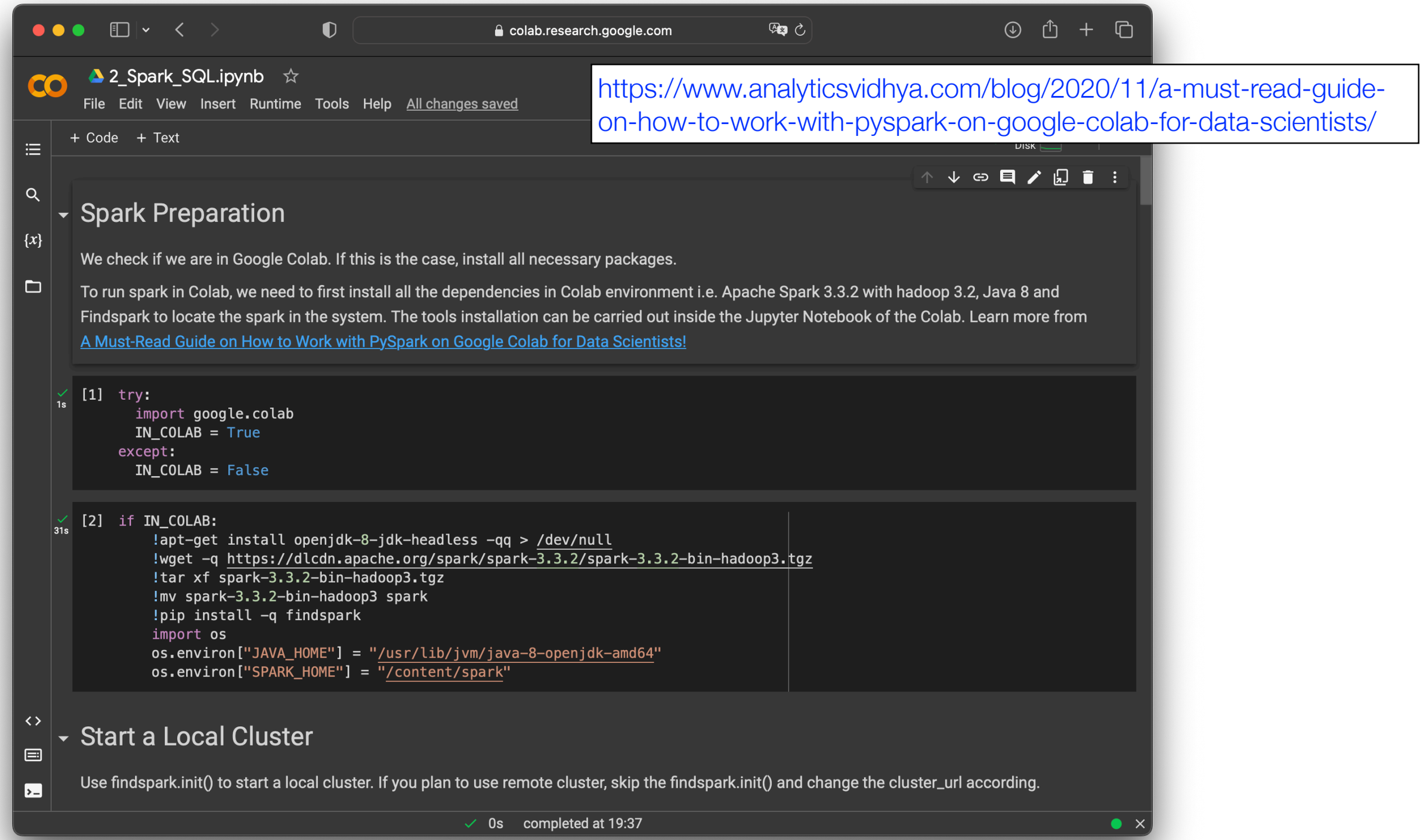
Out[4]:
```
<pyspark.sql.context.SQLContext at 0x7f87b0aebdf0>
```

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

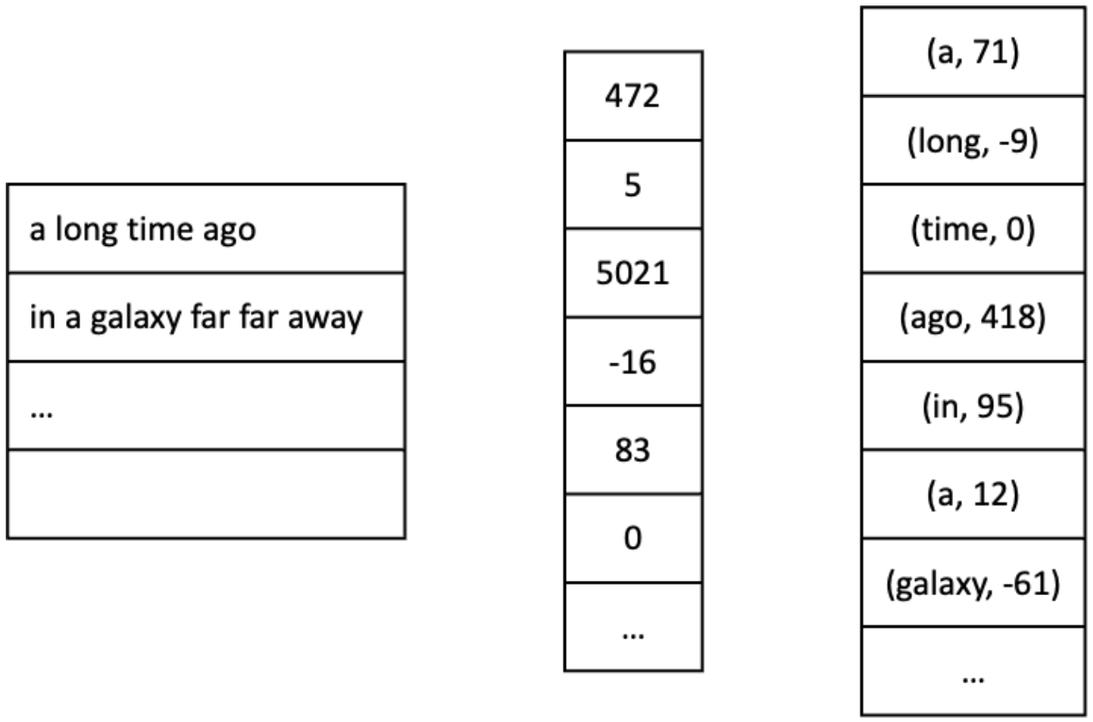
In [ ]:

Using findspark to start a Spark local cluster in Google Colab

---



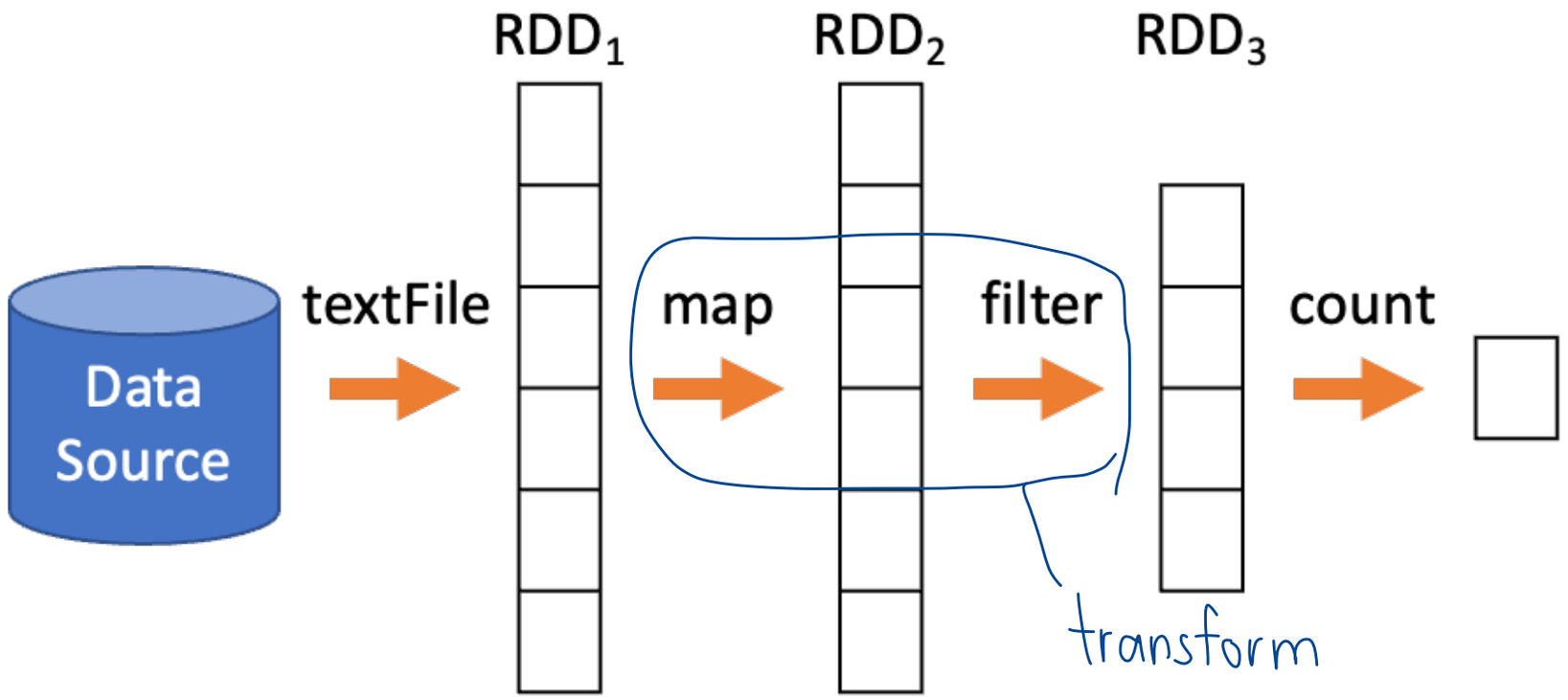
Key Concept: RDD



- 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) — transform
  create an RDD from data
- rdd.count() — action
  count number of elements in an odd
- rdd.filter(func) — transform
  create a new rdd from existing rdd and
  keep only those elements that func is true
- rdd.first() — action
  get the frist element in the rdd
- rdd.collect() — action
  gather all elements in the rdd into a
  python list
- rdd.take(n) — action
  gather first n-th elements in the rdd into a
  python list

```
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

ต้องเป็นค่าสิ่งที่สลับที่ได้

÷ ไม่ได้

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

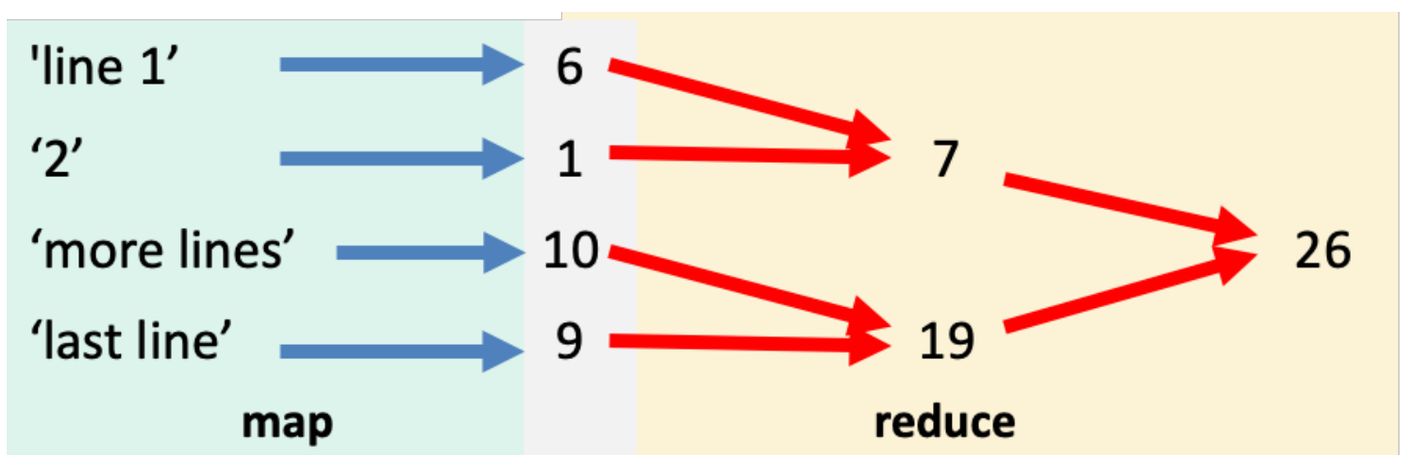
```
['line 1', '2', 'more lines', 'last line']
```

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

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

```
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
```

RDD Aggregation

- Aggregate is an action operation

  `rdd.aggregate(zeroValue, seqOp, combOp)`

  - performs *seqOp* to *zeroValue* and all RDD elements (this basically transforms all elements in RDD into the type of output value)
  - and then aggregates the transformed RDD elements using *combOp*

- Note that reduce is a simple form of aggreate operation.

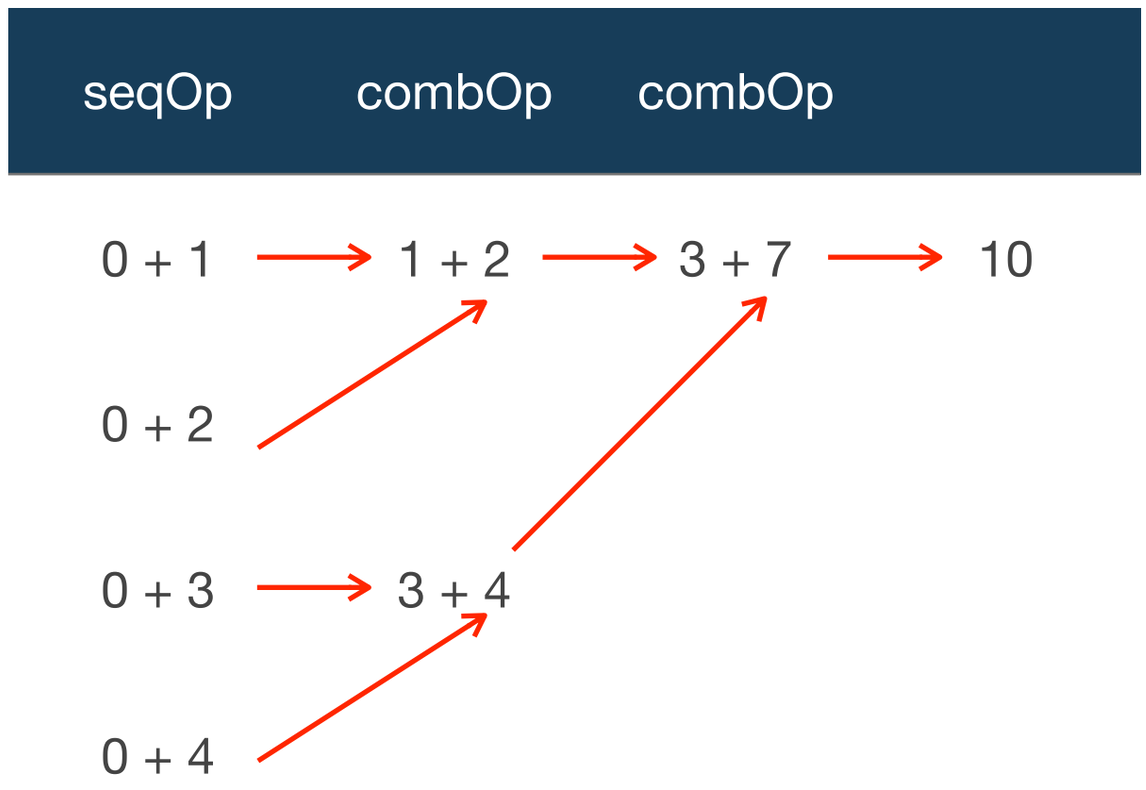
```
rdd.collect()
```

```
[1, 2, 3, 4]
```

The following aggregate operation is basically a *rdd.reduce(lambda a, b: a+b)* as the type output value is an integer which is the same as the RDD elements

```
rdd.aggregate(0,
              lambda zero, e: zero+e,
              lambda a, b: a+b)
```

```
10
```



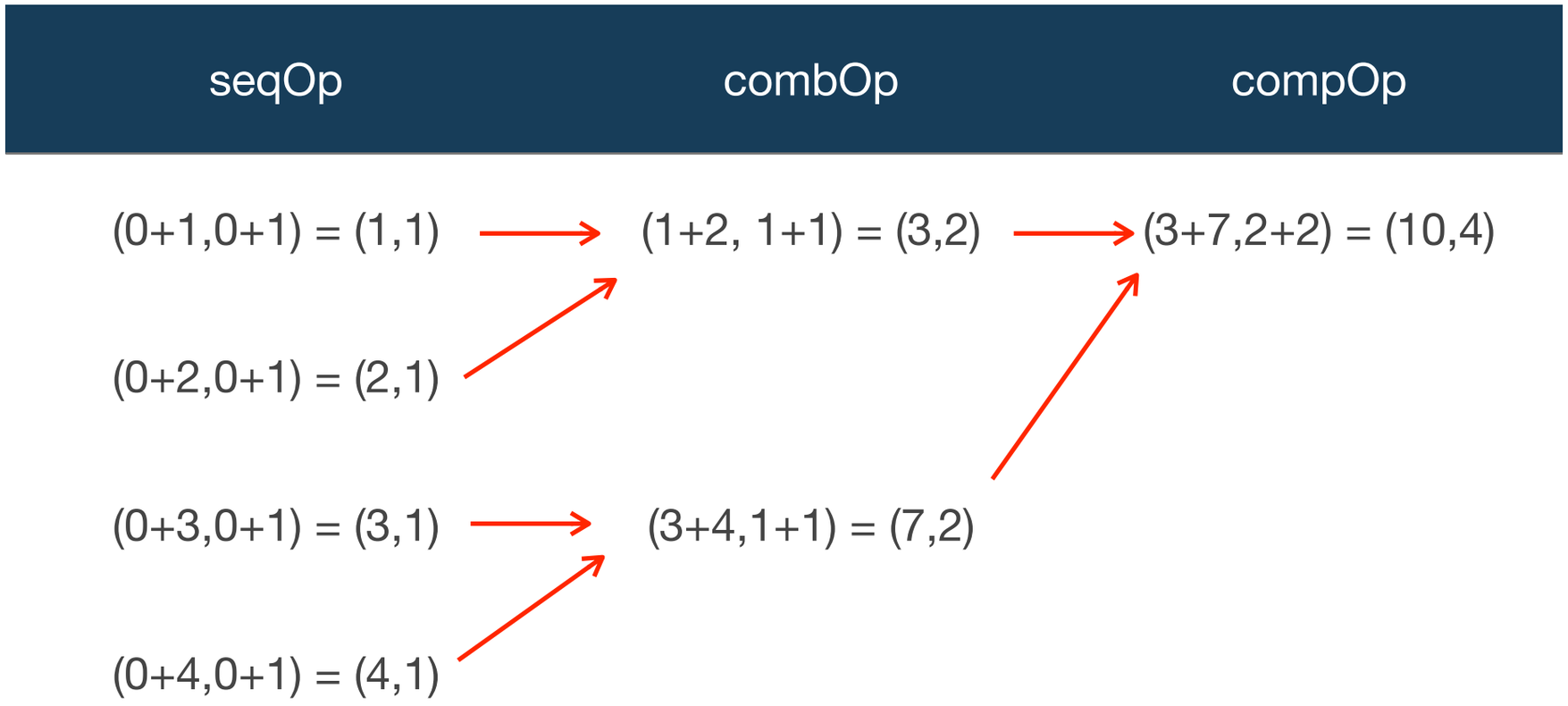
RDD Aggregation of Pairwise RDD

The following aggregate operation returns an order pairs of (x, y) where

- x is the sum of all elements in RDD
- y is the count of all elements in RDD

```
rdd.aggregate((0, 0),
              lambda zero, e: (zero[0]+e, zero[1]+1),
              lambda a, b: (a[0]+b[0], a[1]+b[1]))
```

```
(10, 4)
```



RDD Aggregation of Pairwise RDD

```
lines.collect()
```

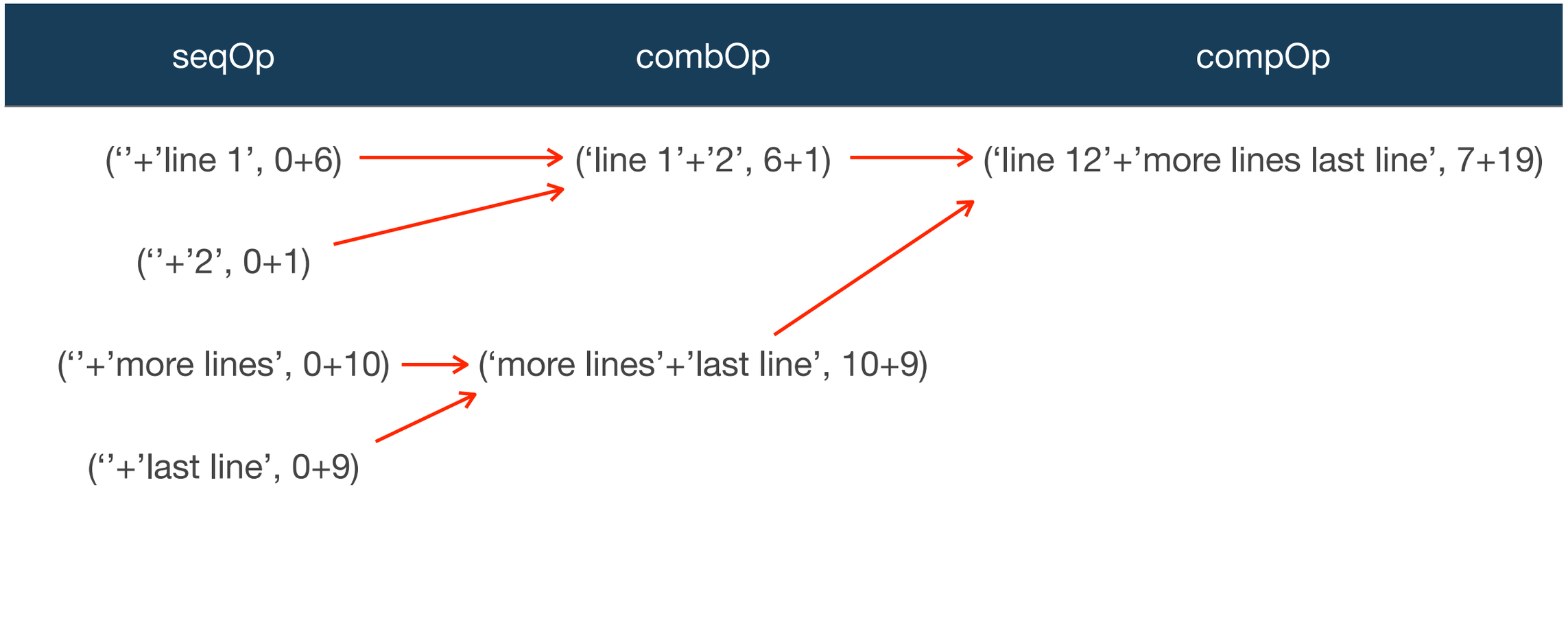
```
['line 1', '2', 'more lines', 'last line']
```

The following aggregate operation returns an order pairs of (x, y) where

- x is the concatenation of all elements in RDD
- y is the sum of the length of all elements in RDD

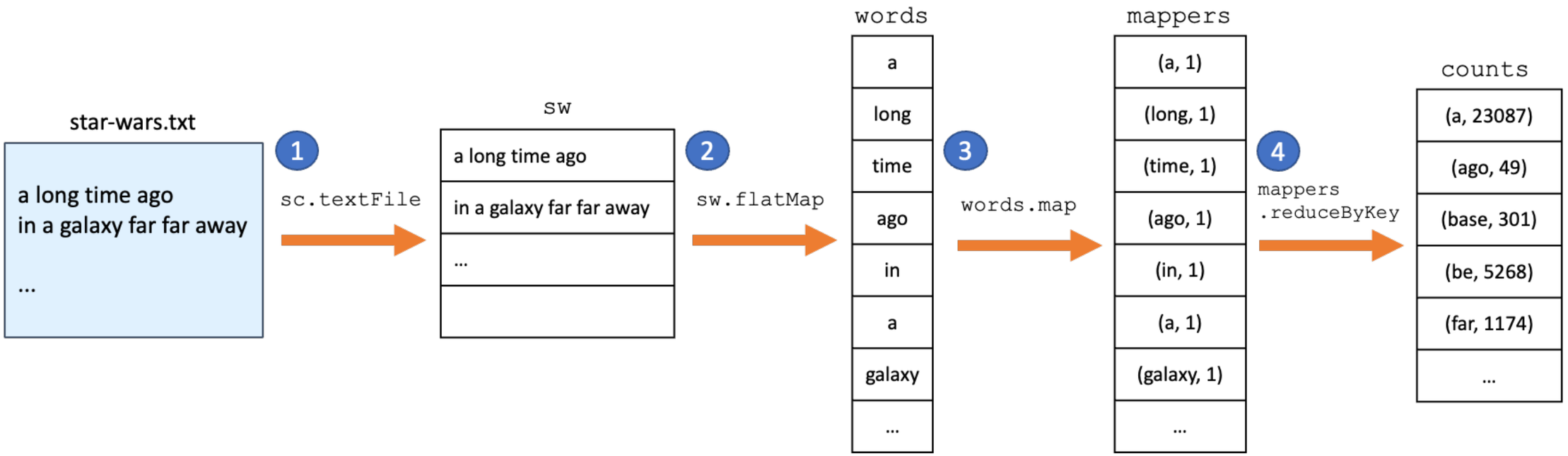
```
lines.aggregate(("", 0),
                lambda zero, e: (zero[0]+e, zero[1]+len(e)),
                lambda a, b: (a[0]+b[0], a[1]+b[1]))
```

```
('line 12more lineslast line', 26)
```

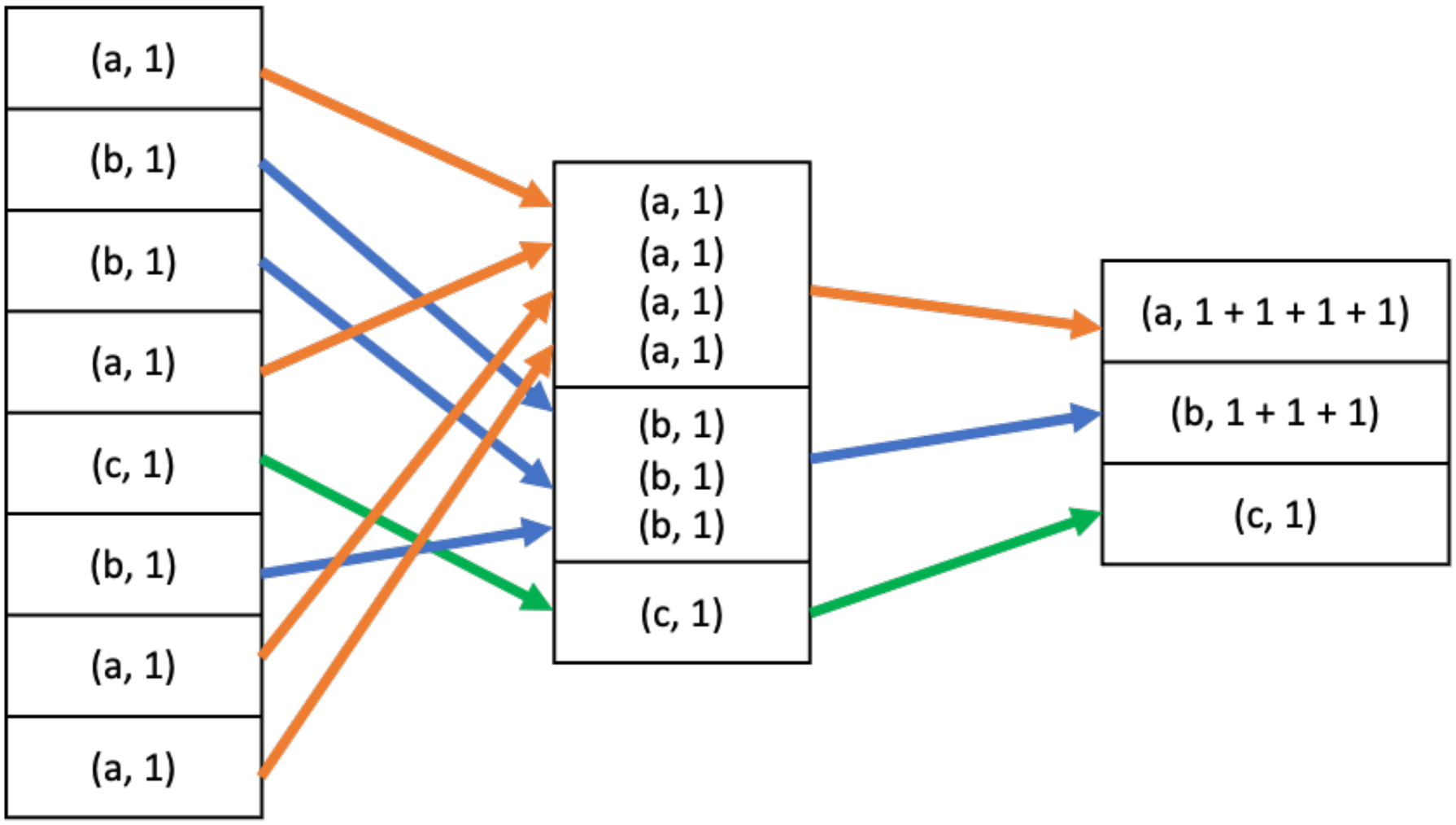


Example: Word Count

```
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)
```



ReduceByKey



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
```

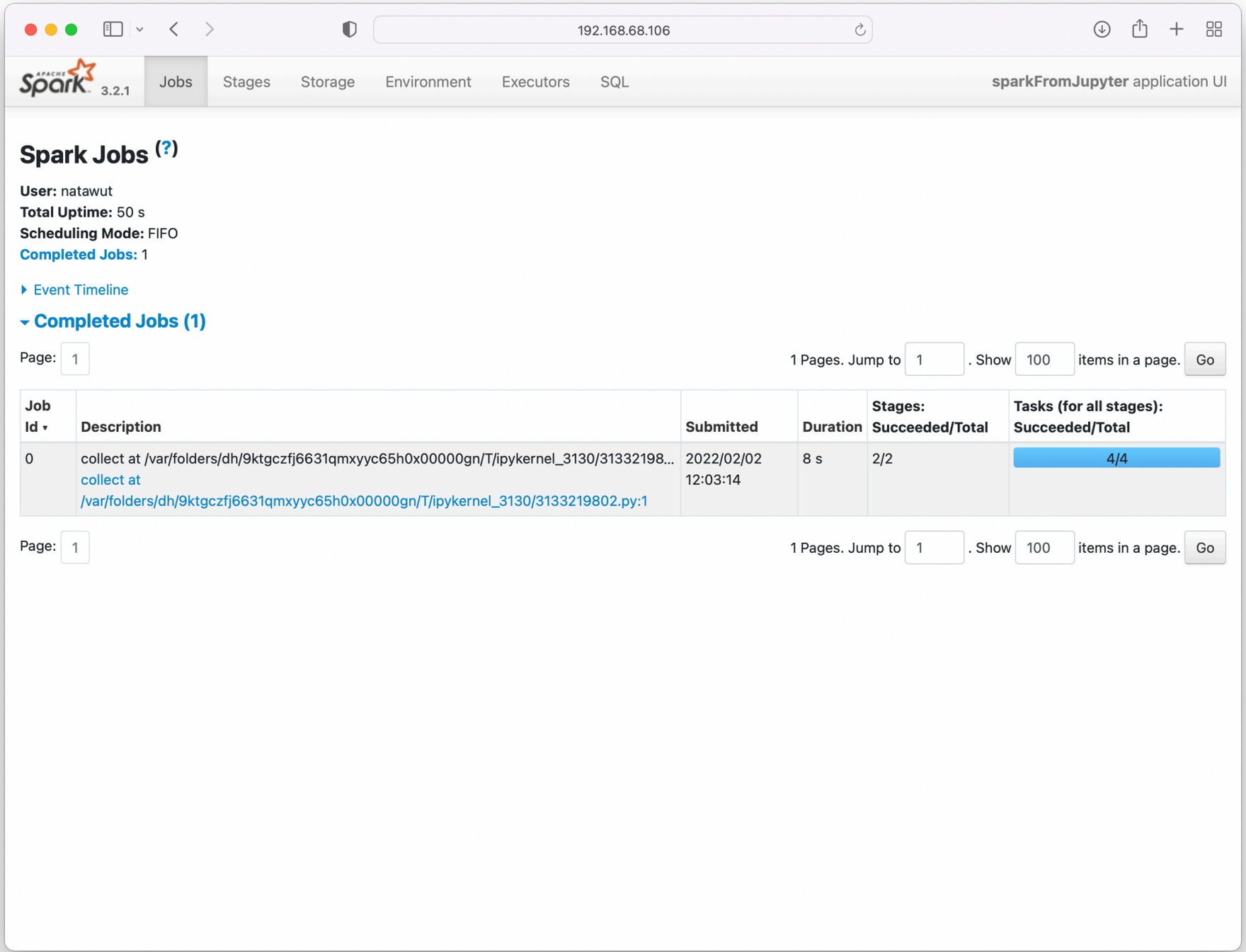
In [3]:
```
sw = sc.textFile('star-wars.txt')
words = sw.flatMap(lambda line: line.split())
mappers = words.map(lambda word: (word, 1))
counts = mappers.reduceByKey(lambda x, y: x+y)
```

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 Jobs (localhost:4040)



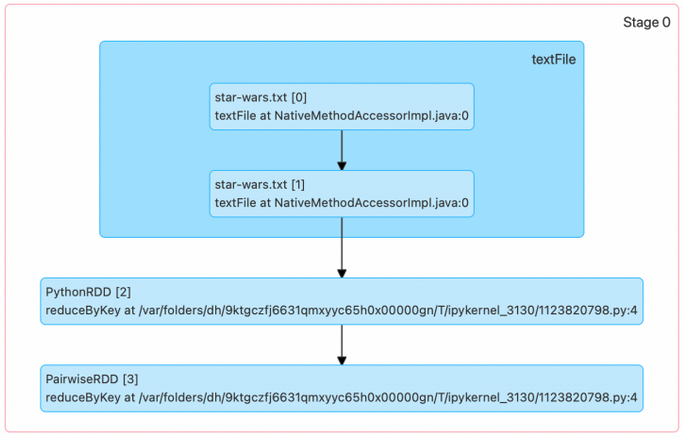
192.168.68.106

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

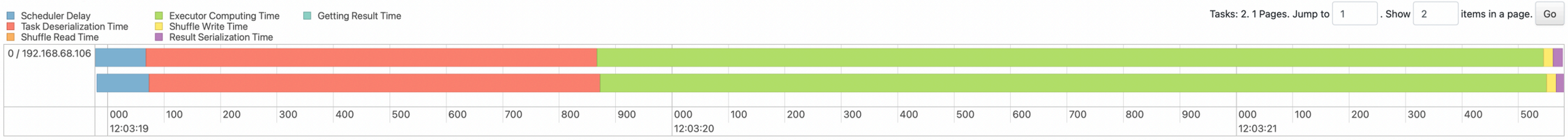
Details for Stage 0 (Attempt 0)

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

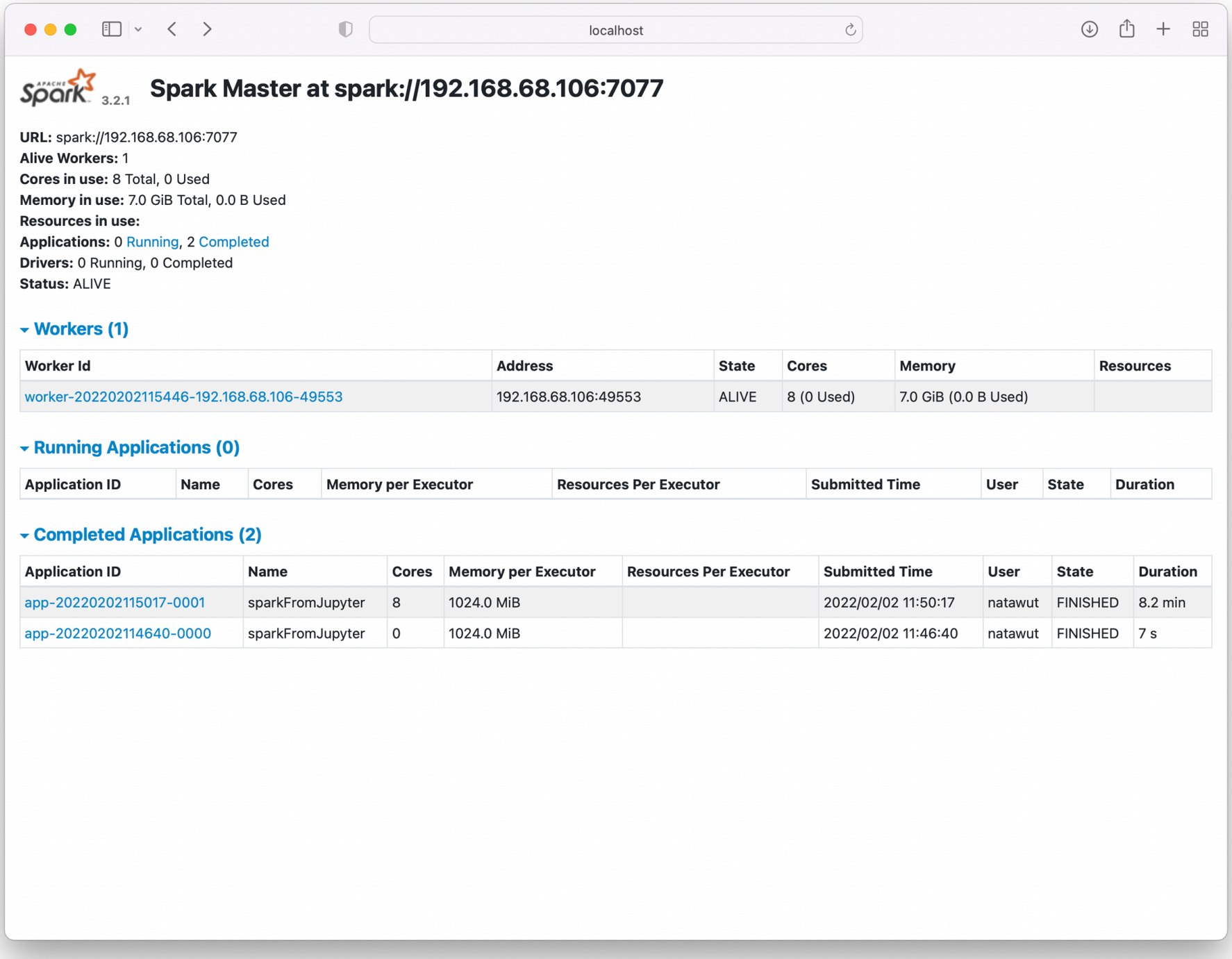


Tasks: 2. 1 Pages. Jump to 1 . Show 2 items in a page. Go

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

Spark Web UI (localhost:8080)



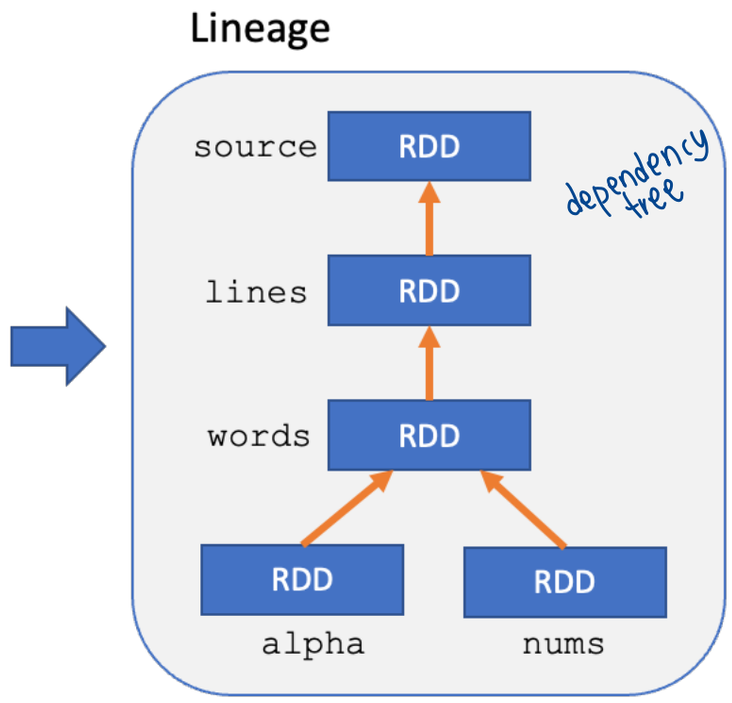
RDD Lineage

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))
```

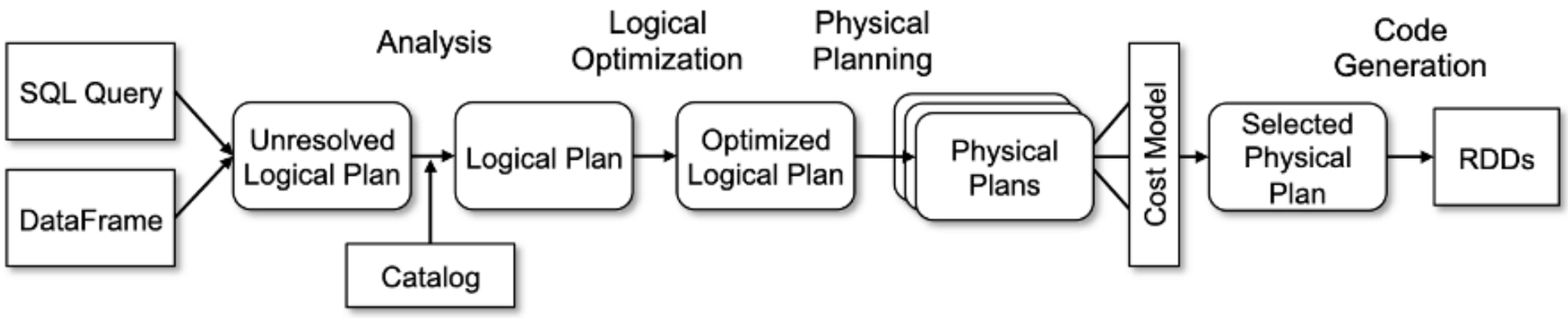
```
acount = alpha.count()
ncount = nums.count()
```

Actions



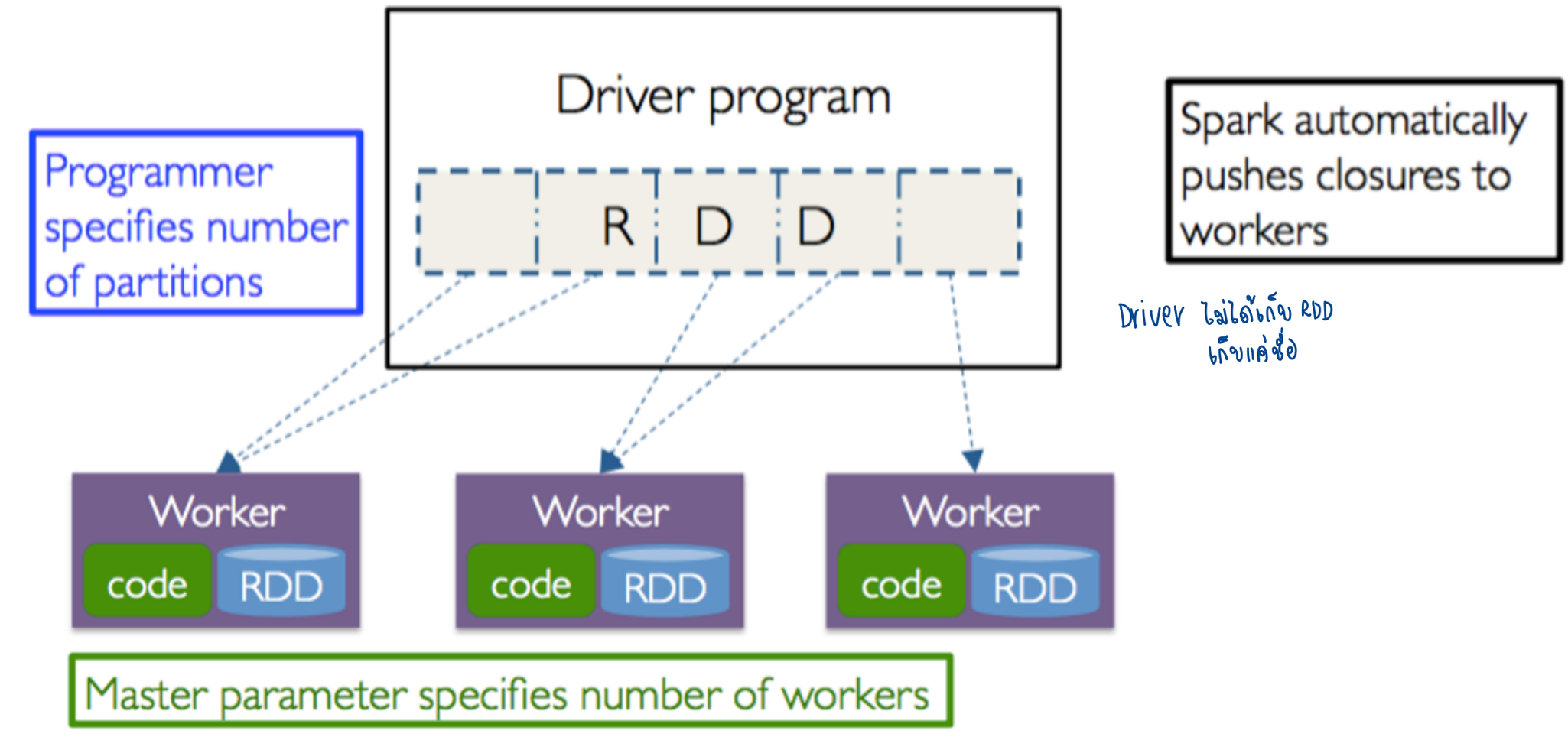
- RDD is "read-only"
- Spark is "lazy execution" = aggregate several commands and execution in batch

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



Driver ไม่ได้เก็บ RDD
เก็บแค่ชื่อ

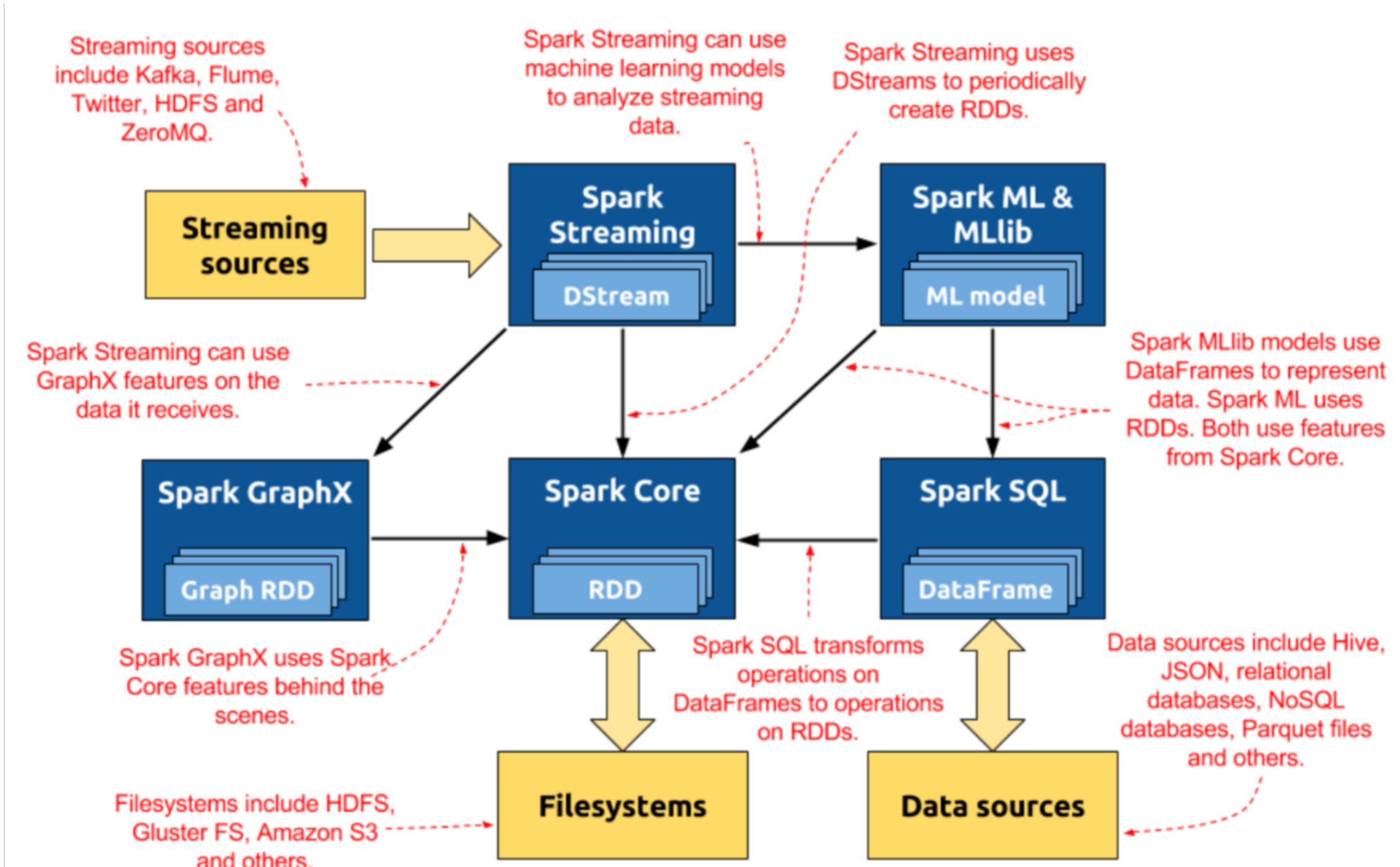


Figure 1.1: Main Spark components and various runtime environments and storage options

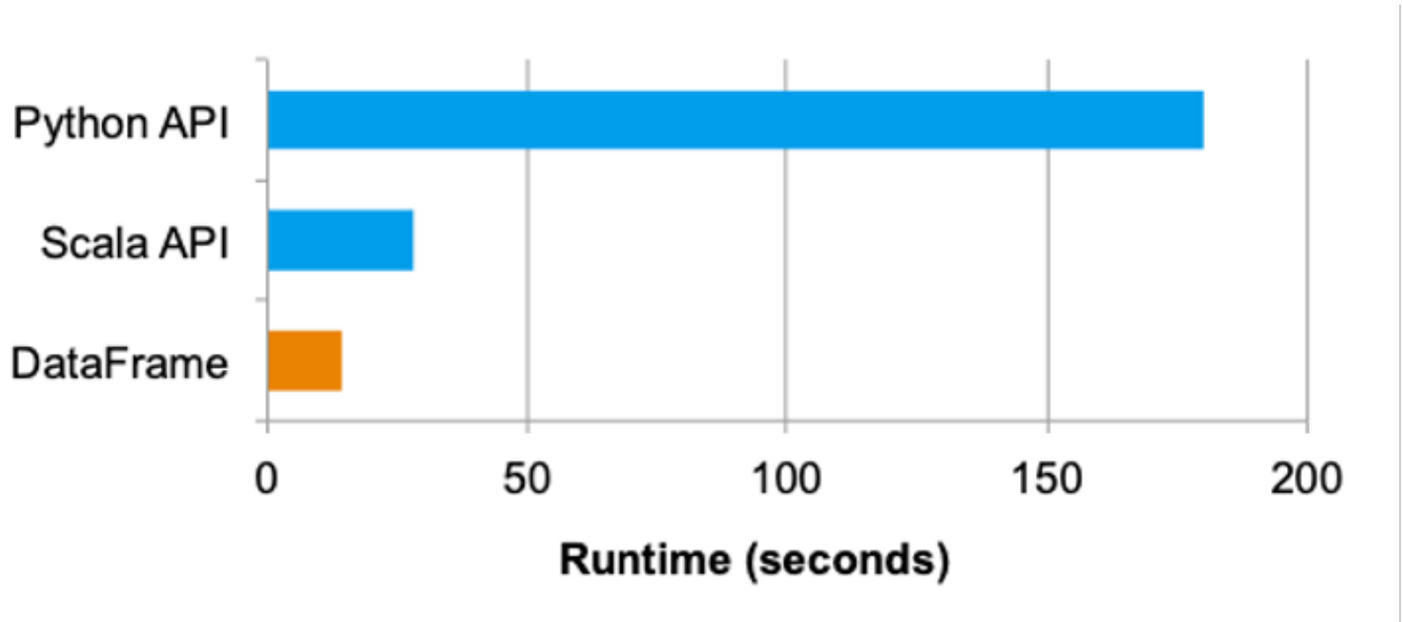
Source: P.Zecevic and M.Bonaci, Spark in Action, Manning Publications, Summer 2016 (est.)

SparkSession / SparkContext / SqlContext

- Spark provides ‘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)
    - Spark shell automatically create the sc variable
    - Spark Submit and other programs must use a constructor to create a new SparkContext
    - Use SparkContext to create RDDs
    - SqlContext is similar to SparkContext, but for Spark SQL

Spark SQL

- Spark SQL provides “SQL-Like” query capability
- 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 row is similar to basic RDD
- Each column can have one datatype
- Spark SQL provides sql-like operations on dataframe

Bank Marketing Dataset

- https://archive.ics.uci.edu/ml/datasets/Bank+Marketing

- The data is related with direct marketing campaigns (phone calls) of a Portuguese banking institution

- 41188 records with 20 inputs and 1 predicted variable

- The goal is to predict if the client will subscribe a term deposit (variable y) — classification problem

- In real-life, this is a “propensity to buy” which indicates the customers who we should give higher priority that others

```
df = spark.read.option("delimiter", ";").option("header", True).csv(path)
```

```
df.columns
```

```
['age',
 'job',
 'marital',
 'education',
 'default',
 'housing',
 'loan',
 'contact',
 'month',
 'day_of_week',
 'duration',
 'campaign',
 'pdays',
 'previous',
 'poutcome',
 'emp.var.rate',
 'cons.price.idx',
 'cons.conf.idx',
 'euribor3m',
 'nr.employed',
 'y']
```

```
cols = [c.replace('.', '_') for c in df.columns]
df = df.toDF(*cols)
```

```
df.show(5)
```

```
+---+--------+-------+----------+-------+-------+----+---------+-----+----------+--------+--------+-----+---
-----+----------+-----------+-------------+-------------+---------+-----------+---+
|age|     job|marital|  education|default|housing|loan|  contact|month|day_of_week|duration|campaign|pdays|pre
vious|   poutcome|emp_var_rate|cons_price_idx|cons_conf_idx|euribor3m|nr_employed|  y|
+---+--------+-------+----------+-------+-------+----+---------+-----+----------+--------+--------+-----+---
-----+----------+-----------+-------------+-------------+---------+-----------+---+
| 56|housemaid|married|   basic.4y|     no|     no|  no|telephone|  may|        mon|     261|       1|  999|
0|nonexistent|         1.1|        93.994|         -36.4|    4.857|       5191| no|
| 57| services|married|high.school|unknown|     no|  no|telephone|  may|        mon|     149|       1|  999|
0|nonexistent|         1.1|        93.994|         -36.4|    4.857|       5191| no|
| 37| services|married|high.school|     no|    yes|  no|telephone|  may|        mon|     226|       1|  999|
0|nonexistent|         1.1|        93.994|         -36.4|    4.857|       5191| no|
| 40|   admin.|married|   basic.6y|     no|     no|  no|telephone|  may|        mon|     151|       1|  999|
0|nonexistent|         1.1|        93.994|         -36.4|    4.857|       5191| no|
| 56| services|married|high.school|     no|     no| yes|telephone|  may|        mon|     307|       1|  999|
0|nonexistent|         1.1|        93.994|         -36.4|    4.857|       5191| no|
+---+--------+-------+----------+-------+-------+----+---------+-----+----------+--------+--------+-----+---
-----+----------+-----------+-------------+-------------+---------+-----------+---+
only showing top 5 rows
```

```
df.printSchema()
```

```
root
 |-- age: string (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- month: string (nullable = true)
 |-- day_of_week: string (nullable = true)
 |-- duration: string (nullable = true)
 |-- campaign: string (nullable = true)
 |-- pdays: string (nullable = true)
 |-- previous: string (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- emp_var_rate: string (nullable = true)
 |-- cons_price_idx: string (nullable = true)
```

Basic Spark SQL Commands

- When Spark reads input, it tries to guess the type of each column and uses the headers of the dataset as column names
    - To change the column name, use *withColumnRenamed* function or use *toDF* to create a new DataFrame with new column names
    - To change data type of a column, use *withColumn* and *cast*
- For data exploration, several commands can be applied e.g. select, filter, groupby, etc.

```
df = df.withColumn('age', df.age.cast('int'))
```

```
from pyspark.sql.functions import col
```

```
cols = ['age', 'duration', 'campaign', 'pdays', 'previous', 'nr_employed']
for c in cols:
    df = df.withColumn(c, col(c).cast('int'))
```

```
cols = ['emp_var_rate', 'cons_price_idx', 'cons_conf_idx', 'euribor3m']
for c in cols:
    df = df.withColumn(c, col(c).cast('double'))
```

Cast and also rename the column y to label

```
df = df.withColumn('label', df.y.cast('boolean'))
```

```
df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- month: string (nullable = true)
 |-- day_of_week: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- emp_var_rate: double (nullable = true)
 |-- cons_price_idx: double (nullable = true)
```

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
  - filter 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

```
df.select(df['job'], df['education'], df['housing']).show(5)
```

```
+---------+-----------+-------+
|      job|  education|housing|
+---------+-----------+-------+
|housemaid|   basic.4y|     no|
| services|high.school|     no|
| services|high.school|    yes|
|   admin.|   basic.6y|     no|
| services|high.school|     no|
+---------+-----------+-------+
only showing top 5 rows
```

```
df.select(df['age'], df['duration'], df['pdays'], df['age']*2, df['duration']+df['pdays']).show(5)
```

```
+---+--------+-----+---------+------------------+
|age|duration|pdays|(age * 2)|(duration + pdays)|
+---+--------+-----+---------+------------------+
| 56|     261|  999|      112|              1260|
| 57|     149|  999|      114|              1148|
| 37|     226|  999|       74|              1225|
| 40|     151|  999|       80|              1150|
| 56|     307|  999|      112|              1306|
+---+--------+-----+---------+------------------+
only showing top 5 rows
```

```
df.filter(df.duration < 100).show(3)
```

```
+---+----------+-------+----------------+-------+-------+----+---------+-----+----------+-------+--------+-----+-------
+--------+-----------+-----------+-------------+-------------+---------+-----------+---+-----+
|age|       job|marital|       education|default|housing|loan|  contact|month|day_of_week|duration|campaign|pdays
|previous|   poutcome|emp_var_rate|cons_price_idx|cons_conf_idx|euribor3m|nr_employed|  y|label|
+---+----------+-------+----------------+-------+-------+----+---------+-----+----------+-------+--------+-----+-------
+--------+-----------+-----------+-------------+-------------+---------+-----------+---+-----+
| 25|  services| single|     high.school|     no|    yes|  no|telephone|  may|       mon|      50|       1|  999
|       0|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
| 41|blue-collar|married|         unknown|unknown|     no|  no|telephone|  may|       mon|      55|       1|  999
|       0|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
| 30| unemployed|married|     high.school|     no|     no|  no|telephone|  may|       mon|      38|       1|  999
|       0|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
| 35| technician|married|university.degree|     no|     no| yes|telephone|  may|       mon|      99|       1|  999
|       0|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
| 59| technician|married|         unknown|     no|    yes|  no|telephone|  may|       mon|      93|       1|  999
|       0|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
+---+----------+-------+----------------+-------+-------+----+---------+-----+----------+-------+--------+-----+-------
+--------+-----------+-----------+-------------+-------------+---------+-----------+---+-----+
only showing top 5 rows
```

```
df.filter(df['job'] == 'housemaid').show(5)
```

```
+---+---------+-------+---------+-------+-------+----+---------+-----+----------+-------+--------+-----+--------
+----------+-----------+-------------+-------------+---------+-----------+---+-----+
|age|      job| marital|education|default|housing|loan|  contact|month|day_of_week|duration|campaign|pdays|previous
|   poutcome|emp_var_rate|cons_price_idx|cons_conf_idx|euribor3m|nr_employed|  y|label|
+---+---------+-------+---------+-------+-------+----+---------+-----+----------+-------+--------+-----+--------
+----------+-----------+-------------+-------------+---------+-----------+---+-----+
| 56|housemaid| married| basic.4y|     no|     no|  no|telephone|  may|       mon|     261|       1|  999|       0
|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
| 57|housemaid|divorced| basic.4y|     no|    yes|  no|telephone|  may|       mon|     293|       1|  999|       0
|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
| 39|housemaid| married| basic.4y|     no|     no| yes|telephone|  may|       mon|     266|       1|  999|       0
|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
| 34|housemaid| married| basic.6y|     no|    yes|  no|telephone|  may|       mon|     300|       2|  999|       0
|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
| 39|housemaid| married| basic.4y|unknown|     no|  no|telephone|  may|       mon|      69|       2|  999|       0
|nonexistent|        1.1|       93.994|         -36.4|    4.857|       5191| no|false|
+---+---------+-------+---------+-------+-------+----+---------+-----+----------+-------+--------+-----+--------
+----------+-----------+-------------+-------------+---------+-----------+---+-----+
only showing top 5 rows
```

```
df.filter((df['age'] > 60) & (df.age <= 65)).select('age', 'marital').show(5)
```

```
+---+--------+
|age| marital|
+---+--------+
| 61| married|
| 61| married|
| 61| married|
| 63|divorced|
| 62| married|
+---+--------+
only showing top 5 rows
```

```
df.filter("marital == 'married'").select('job', 'marital').show(5)
```

```
+---------+-------+
|      job|marital|
+---------+-------+
|housemaid|married|
| services|married|
| services|married|
|   admin.|married|
| services|married|
+---------+-------+
only showing top 5 rows
```

```
df.filter('age < 40 and duration > 200').select('age', 'duration', 'marital').show(5)
```

```
+---+--------+-------+
|age|duration|marital|
+---+--------+-------+
| 37|     226|married|
| 24|     380| single|
| 25|     222| single|
| 35|     312|married|
| 39|     233|married|
+---+--------+-------+
only showing top 5 rows
```

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

```
from pyspark.sql.functions import avg, min, max, countDistinct
```

```
df.select(avg('age'), min('age'), max('duration')).show()
```

```
[Stage 6:>                                                          (0 + 1) / 1]
```

```
+-----------------+--------+-------------+
|         avg(age)|min(age)|max(duration)|
+-----------------+--------+-------------+
|40.02406040594348|      17|         4918|
+-----------------+--------+-------------+
```

Groupby function allows us to work data in groups.

```
df.groupby('marital').count().show()
```

```
[Stage 9:>                                                          (0 + 1) / 1]
```

```
+--------+-----+
| marital|count|
+--------+-----+
| unknown|   80|
|divorced| 4612|
| married|24928|
|  single|11568|
+--------+-----+
```

```
df.groupby('marital', 'education').agg({'age': 'min'}).show()
```

```
+--------+------------------+--------+
| marital|         education|min(age)|
+--------+------------------+--------+
|divorced|       high.school|      24|
|divorced|           unknown|      26|
| unknown| university.degree|      25|
| unknown|           unknown|      31|
| married|professional.course|      22|
|  single|          basic.9y|      17|
| married|          basic.4y|      20|
```

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
```

```
def agegroup_mapping(age):
    if age < 25:
        return 'young'
    if age < 55:
        return 'adult'
    return 'senior'

to_agegroup = udf(agegroup_mapping, StringType())
```

```
df.select('age', to_agegroup('age')).show(5)
```

```
[Stage 15:>                                                         (0 + 1) / 1]
```

```
+---+--------------------+
|age|agegroup_mapping(age)|
+---+--------------------+
| 56|              senior|
| 57|              senior|
| 37|               adult|
| 40|               adult|
| 56|              senior|
+---+--------------------+
only showing top 5 rows
```

```
new_df = df.withColumn('agegroup', to_agegroup(df.age))
new_df.select(new_df['age'], new_df['agegroup']).show(10)
```

```
+---+--------+
|age|agegroup|
+---+--------+
| 56|  senior|
| 57|  senior|
| 37|   adult|
| 40|   adult|
| 56|  senior|
```

Spark ML

- Spark provides two machine learning packages, Spark MLlib and Spark ML

- Spark MLlib (spark.mllib package) is an RDD-based API, which is in maintenance mode

- Spark ML (spark.ml) is a DataFrame-based API, which is more user-friendly and can utilize Spark SQL capabilities

Spark ML Pipeline Concepts

- DataFrame — the base of ML API to store feature vectors, true labels, and predictions

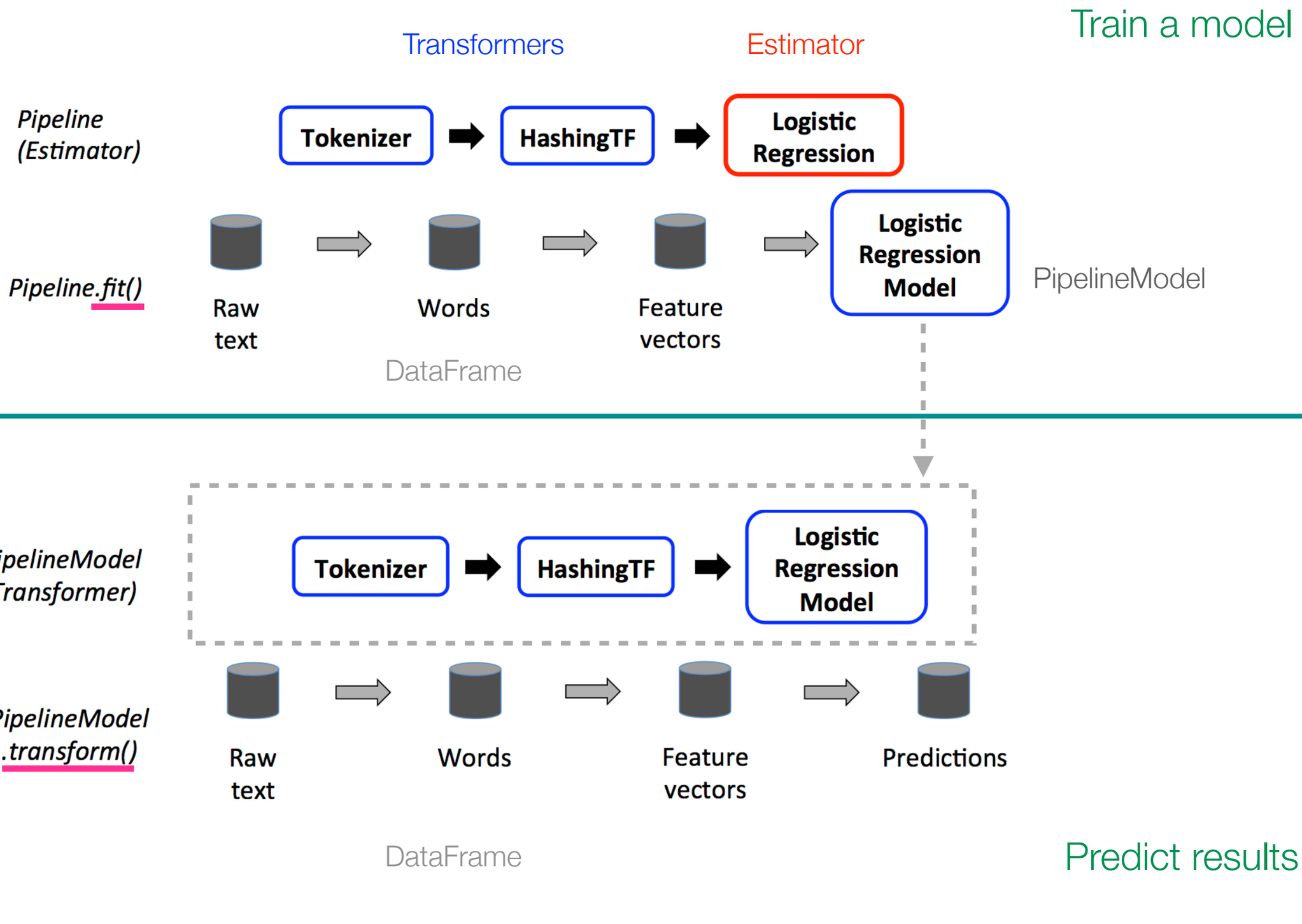
- Transformer — transform DataFrame to DataFrame e.g. ML model is a Transformer — from features to predictions

- 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

- Pipeline — chains multiple Transformers and Estimators into stages to specify an ML workflow

- Parameter — all Transformers and Estimators share a common API for specifying parameters

Example Pipelines



Back to Our Bank Marketing Dataset

- After we prepare the data, we split data into training data (80%) and testing data (20%)
- We create 2 pipelines, feature extraction pipeline and ML prediction pipeline
  - Fit featurePipeline with training dataset to create featureOnlyModel (feature extraction pipeline)
  - Transform training and testing datasets using featureOnlyModel to create DataFrames with features column (trainingFeaturesDf, testingFeaturesDf)
  - Fit ML Pipeline (an estimator = ML algo) with training dataset to create PipelineModel (ML prediction pipeline)
  - Transform PipelineModel with testing dataset to predict the results

```
train_df, test_df = df.randomSplit([0.8,0.2])
```

```
train_df.count()
```

```
32889
```

```
test_df.count()
```

```
8299
```

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler
from pyspark.ml import Pipeline
```

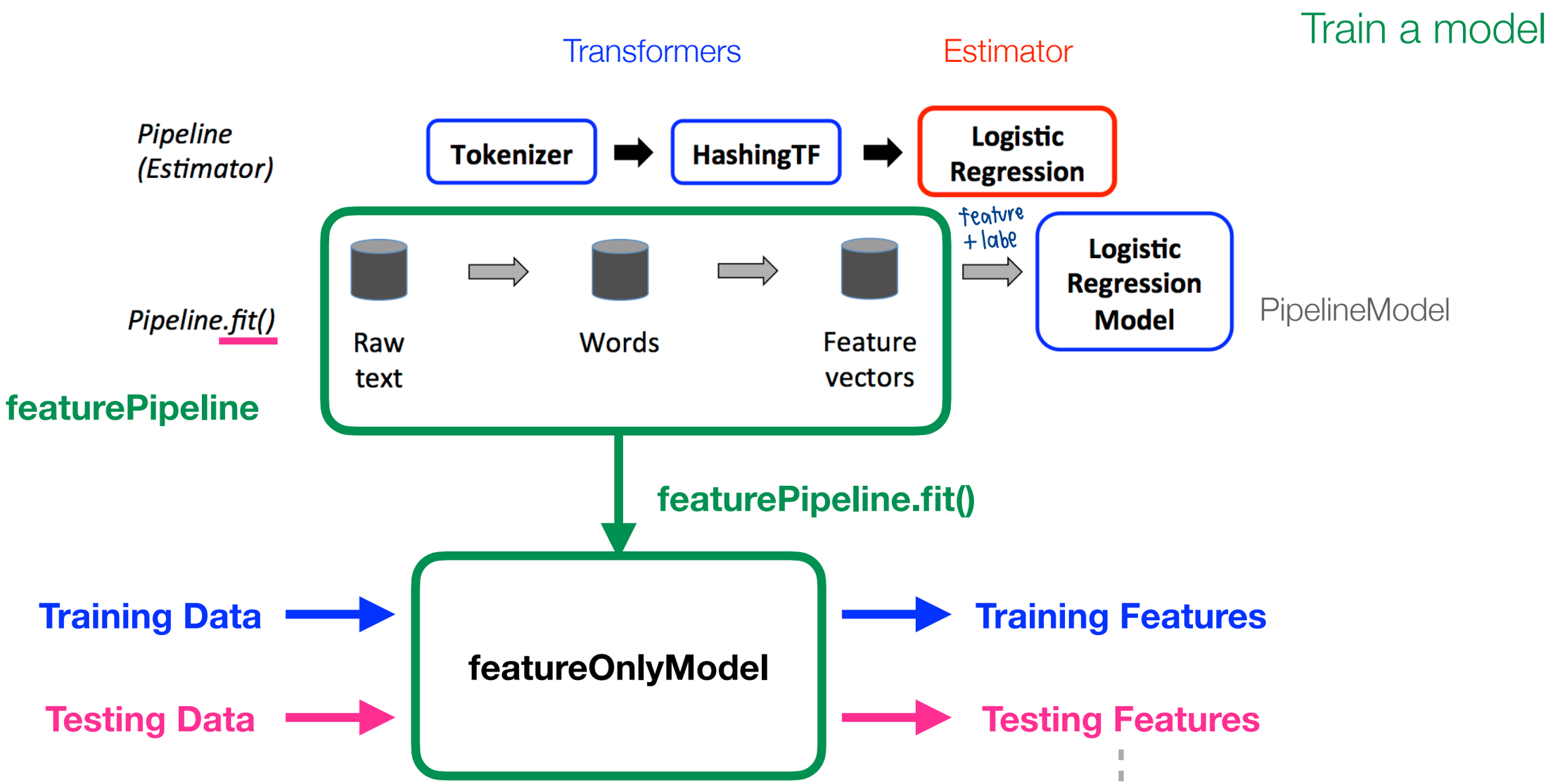
We first setup a pipeline of all data transformation.

```
stages = []
categoricalAttributes = ['job', 'marital', 'education', 'default',
                         'housing', 'loan', 'contact',
                         'month', 'day_of_week', 'poutcome']
for columnName in categoricalAttributes:
    stringIndexer = StringIndexer(inputCol=columnName, outputCol=columnName+ "Index")
    stages.append(stringIndexer)
    oneHotEncoder = OneHotEncoder(inputCol=columnName+ "Index", outputCol=columnName + "Vec")
    stages.append(oneHotEncoder)

categoricalCols = [s + "Vec" for s in categoricalAttributes]
```

```
numericColumns = ['age', 'campaign', 'pdays', 'previous',
          'emp_var_rate', 'cons_price_idx', 'cons_conf_idx',
                'euribor3m', 'nr_employed']
```

```
# Combine all the feature columns into a single column in the dataframe

allFeatureCols =  numericColumns + categoricalCols
vectorAssembler = VectorAssembler(
    inputCols=allFeatureCols,
    outputCol="features")
stages.append(vectorAssembler)
```

Feature Pipeline




```
[Row(features=SparseVector(52, {0: 17.0, 1: 2.0, 2: 999.0, 3: 1.0, 4: -2.9, 5: 92.201, 6: -31.4, 7: 0.869, 8: 5
076.0, 19: 1.0, 21: 1.0, 25: 1.0, 30: 1.0, 36: 1.0, 39: 1.0, 51: 1.0}), label=1),
 Row(features=SparseVector(52, {0: 17.0, 1: 3.0, 2: 4.0, 3: 2.0, 4: -2.9, 5: 92.201, 6: -31.4, 7: 0.869, 8: 507
6.0, 19: 1.0, 21: 1.0, 25: 1.0, 30: 1.0, 32: 1.0, 34: 1.0, 36: 1.0, 39: 1.0}), label=0),
```

Feature Extraction Pipeline ¶

We build 2 pipelines, feature transformation pipeline and ML pipeline. This allows us to reuse the feature transformation pipeline with several ML algorithms. **'fit'** method is called to create a model and we can use **'transform'** to actual transform or infer data

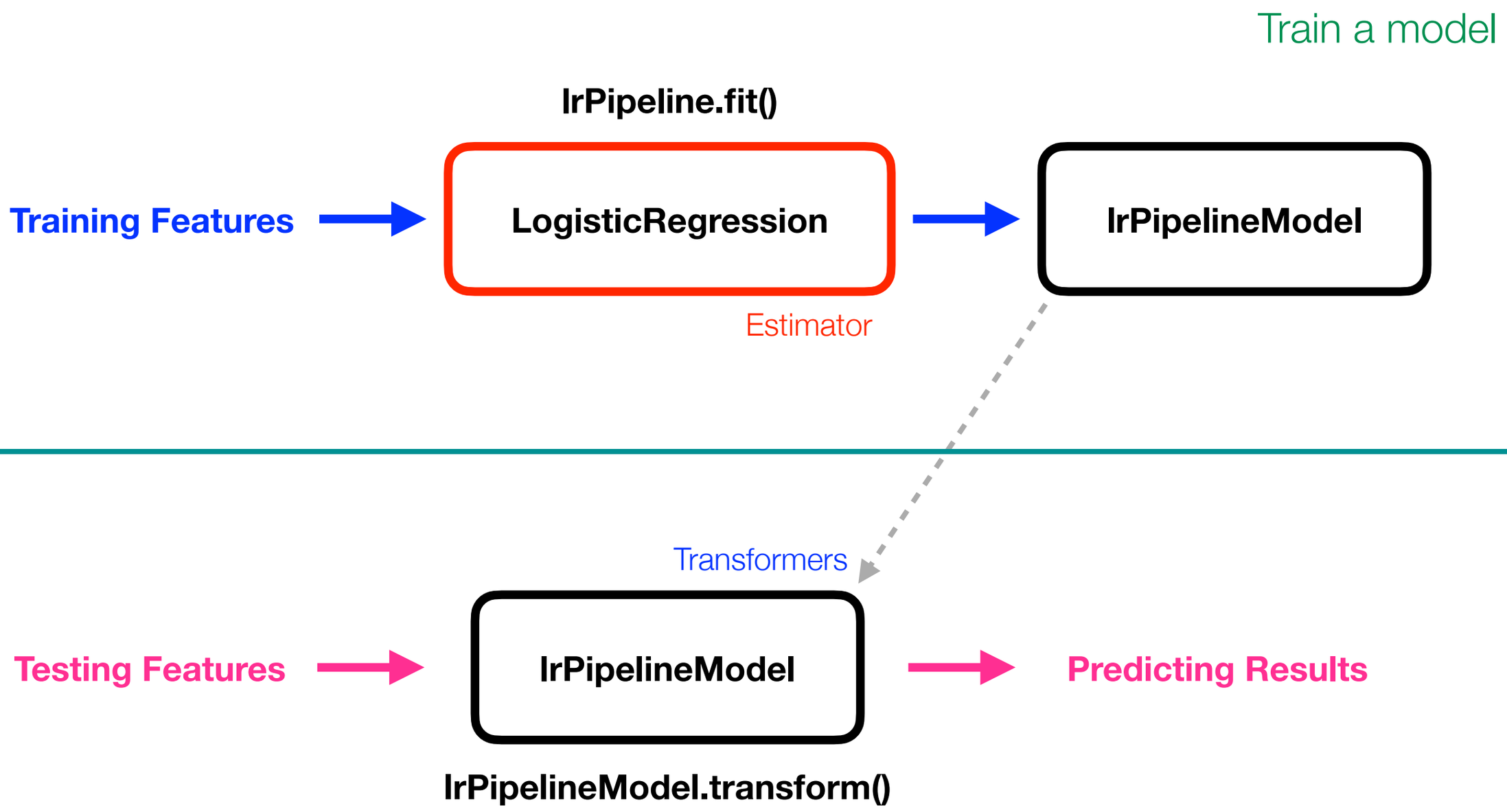
```
# Build pipeline for feature extraction

featurePipeline = Pipeline(stages=stages)
featureOnlyModel = featurePipeline.fit(train_df)
```

```
# Apply to training and testing dataframes to create feature dataframes

trainingFeaturesDf = featureOnlyModel.transform(train_df)
testFeaturesDf = featureOnlyModel.transform(test_df)
```

```
# Peek into training features

trainingFeaturesDf.select("features", "label").rdd.take(5)
```

```
[Row(features=SparseVector(52, {0: 17.0, 1: 2.0, 2: 999.0, 3: 1.0, 4: -2.9, 5: 92.201, 6: -31.4, 7: 0.869, 8: 5076.0, 19: 1.0, 21: 1.0, 25: 1.0, 30: 1.0, 36: 1.0, 39: 1.0, 51: 1.0}), label=1),
 Row(features=SparseVector(52, {0: 17.0, 1: 3.0, 2: 4.0, 3: 2.0, 4: -2.9, 5: 92.201, 6: -31.4, 7: 0.869, 8: 5076.0, 19: 1.0, 21: 1.0, 25: 1.0, 30: 1.0, 32: 1.0, 34: 1.0, 36: 1.0, 39: 1.0}), label=0),
 Row(features=SparseVector(52, {0: 17.0, 1: 2.0, 2: 999.0, 3: 2.0, 4: -2.9, 5: 92.201, 6: -31.4, 7: 0.869, 8: 5076.0, 19: 1.0, 21: 1.0, 25: 1.0, 30: 1.0, 32: 1.0, 34: 1.0, 36: 1.0, 39: 1.0, 51: 1.0}), label=0),
 Row(features=SparseVector(52, {0: 17.0, 1: 1.0, 2: 2.0, 3: 2.0, 4: -3.4, 5: 92.431, 6: -26.9, 7: 0.742, 8: 5017.0, 19: 1.0, 21: 1.0, 29: 1.0, 30: 1.0, 33: 1.0, 35: 1.0, 36: 1.0, 43: 1.0, 49: 1.0}), label=1),
 Row(features=SparseVector(52, {0: 17.0, 1: 3.0, 2: 4.0, 3: 2.0, 4: -2.9, 5: 92.201, 6: -31.4, 7: 0.884, 8: 5076.0, 19: 1.0, 21: 1.0, 29: 1.0, 30: 1.0, 32: 1.0, 34: 1.0, 36: 1.0, 39: 1.0, 48: 1.0}), label=0)]
```

ML Pipeline - Training / Predicting



Logistic Regression Model

Configure an machine learning pipeline, which consists of an estimator (classification) (Logistic regression)

```
lr = LogisticRegression(maxIter=10, regParam=0.01)
lrPipeline = Pipeline(stages=[lr])
```

```
# Fit the pipeline to create a model from the training data

lrPipelineModel = lrPipeline.fit(trainingFeaturesDf)
```

```

```

```
results = lrPipelineModel.transform(testFeaturesDf)
print('LogisticRegression Model test accuracy = ', calculateAccuracy(results))
```

```
LogisticRegression Model test accuracy =  0.896555950484128
```

DecisionTree Model

```
dt = DecisionTreeClassifier(labelCol='label', featuresCol='features')
dtPipeline = Pipeline(stages=[dt])
```

```
dtPipelineModel = dtPipeline.fit(trainingFeaturesDf)
```

```

```

```
results = dtPipelineModel.transform(testFeaturesDf)
print('DecisionTree Model test accuracy = ', calculateAccuracy(results))
```

```
DecisionTree Model test accuracy =  0.8979041549209462
```

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

- https://www.kaggle.com/code/kkhandekar/apache-spark-beginner-tutorial/notebook

- https://www.kaggle.com/code/fatmakursun/pyspark-ml-tutorial-for-beginners

- https://sparkbyexamples.com/