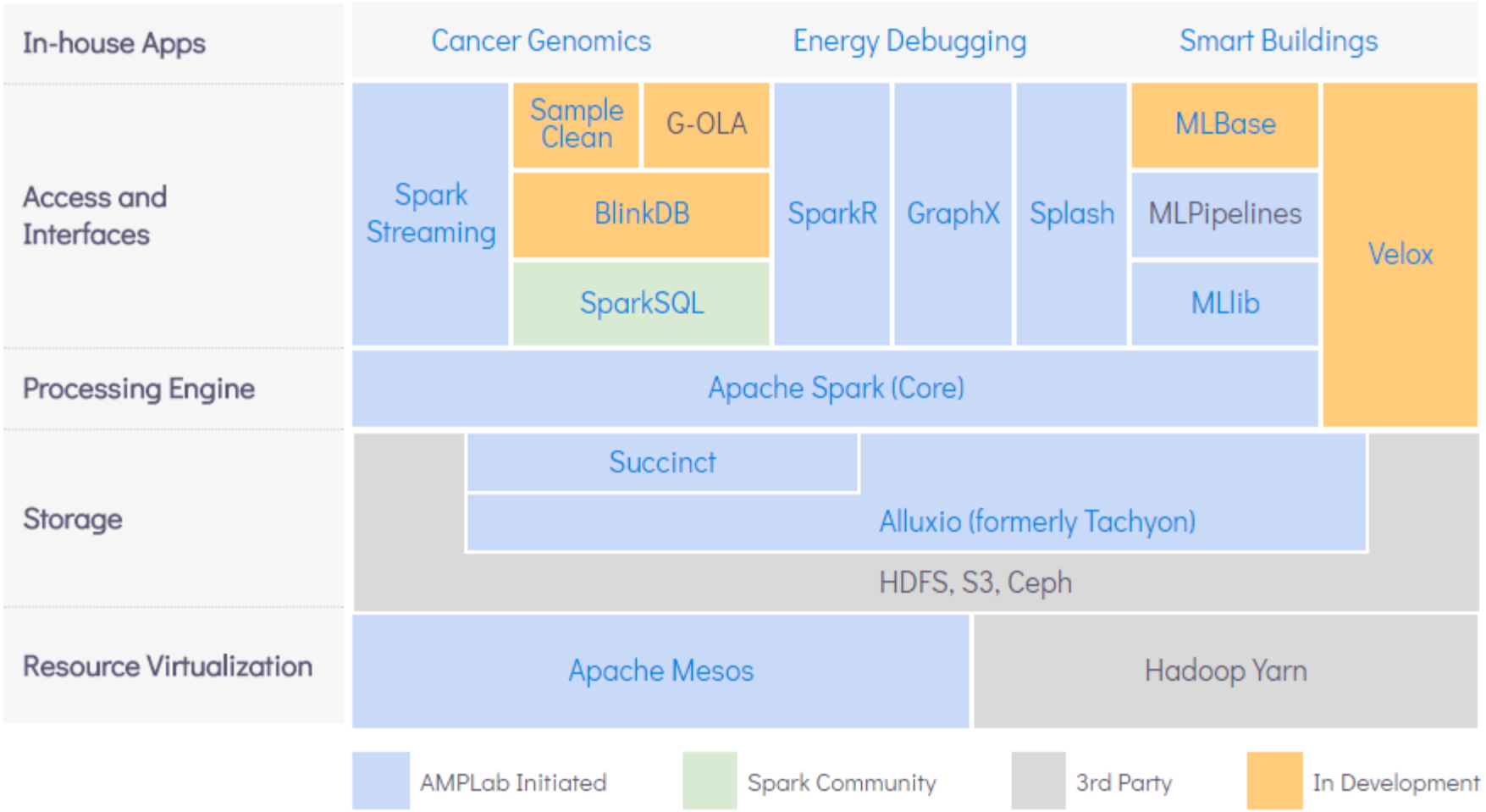
APACHE SPARK FOR DATA SCIENTISTS

Session 1: Introduction to Apache Spark

UC BERKELEY Research Projects – AMPlab (Ended)

BDAS (the Berkeley Data Analytics Stack)



Ref: https://amplab.cs.berkeley.edu/software/

UC BERKELEY Research Projects – RISELab (In Progress)

- Launched in January 2017: https://rise.cs.berkeley.edu/

- Focus on creating open-source platforms for Real-time, Intelligent, Secure, Explainable, Systems:

RISELab Bootcamp will be on 11/12th of October 2018.

Some Interesting RISELab Projects:

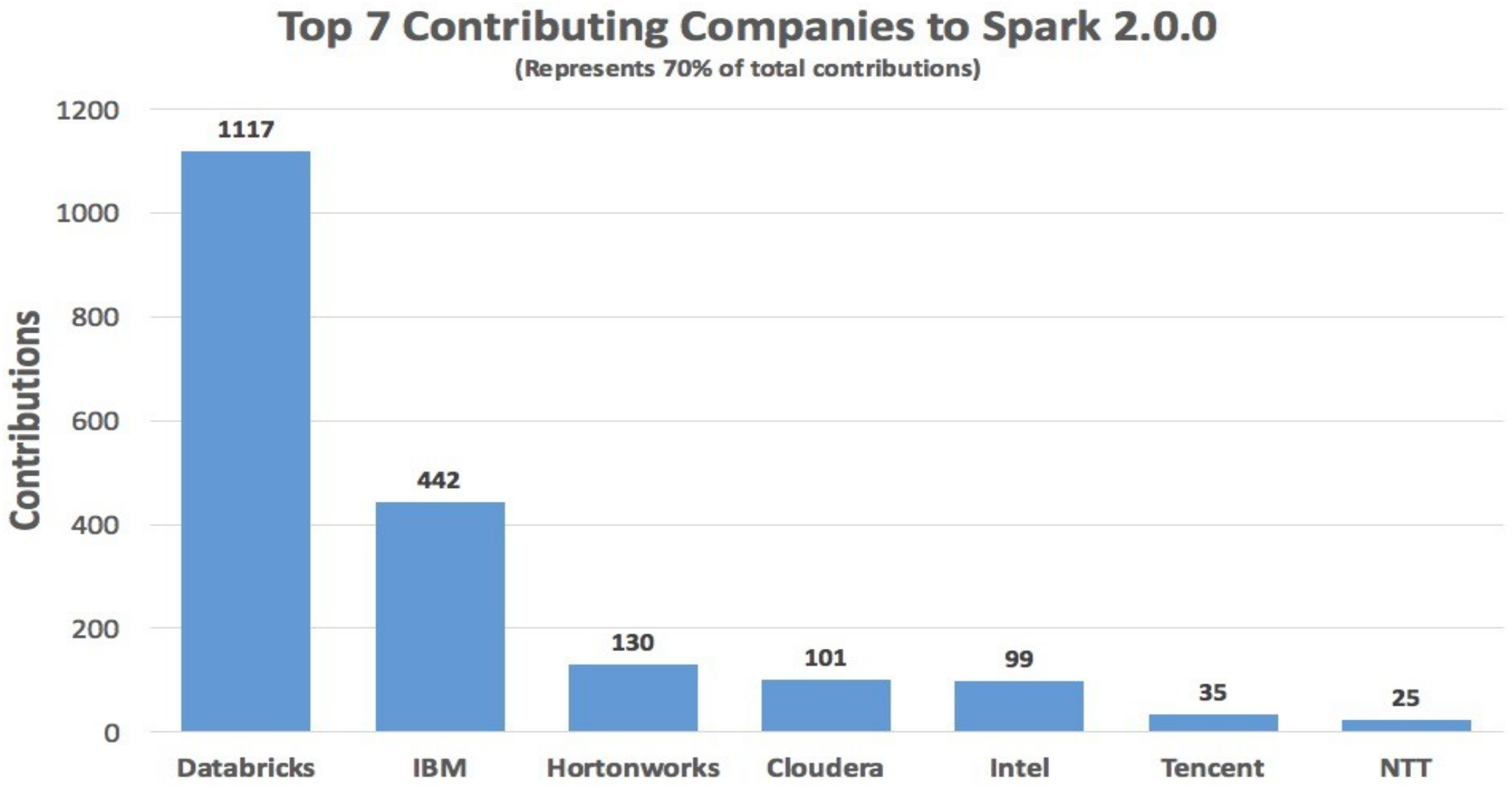
Clipper is a general-purpose low-latency model serving system which can integrate with many ML frameworks including Spark MLlib.

Ray is a Python based distributed execution framework aims to facilitate to parallelize existing codebases.

Opaque aims to enable analytics on sensitive data in an untrusted cloud by encrypting Spark SQL Datasets.

Drizzle is a low latency execution engine for Apache, which unifies the record-at-a-time streaming and micro-batch (legacy API) streaming models in a hybrid streaming system.

Corporate Contribution to Apache Spark



March 2017 Ref:
https://www.slideshare.net/JohanPicard/a-short-introduction-to-spark-and-its-benefits

Contribution to Apache Spark

https://databricks.com/

https://databricks.com/sparkaisummit

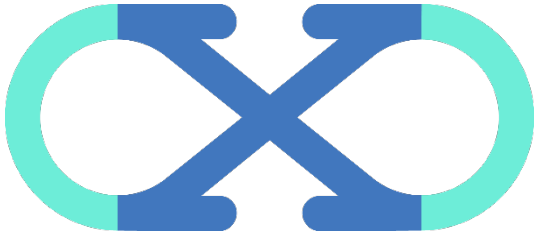
Databricks Community Cloud:
https://community.cloud.databricks.com/

SparkHub:
https://sparkhub.databricks.com/

https://www.ibm.com/analytics/us/en/technology/spark/

IBM Spark Technology Center:
http://www.spark.tc/

IBM Data Science Experience:
https://datascience.ibm.com/



I. Course Methodology & Logistics Overview

II. Spark: Background & Position in Big Data Analytics

III. Core Concepts & Challenges of Distributed Computing

IV. Overview of Spark Components

V. Conceptual Introduction to Spark Application

VI. Appendix

Purpose of Big Data Analytics

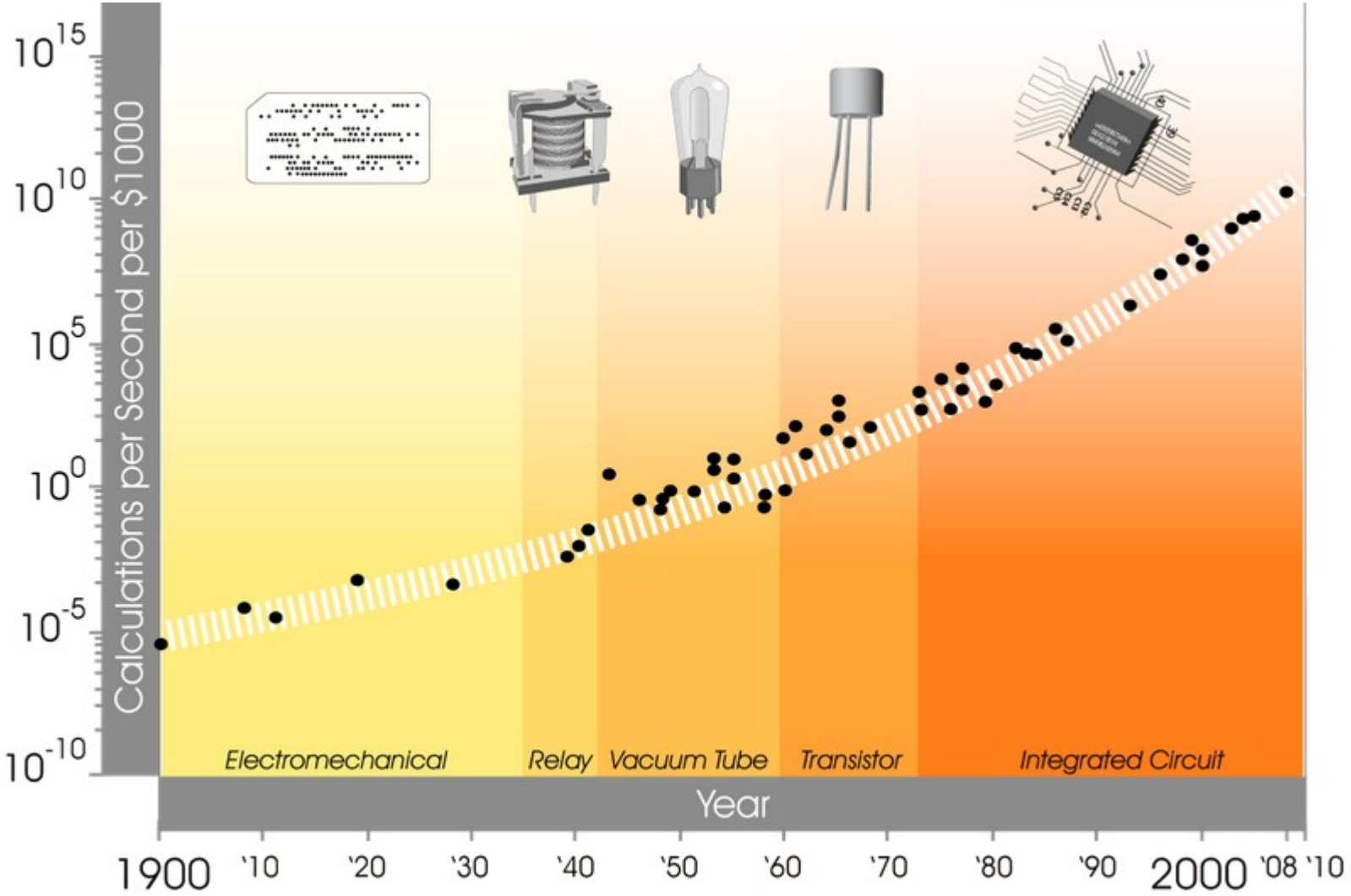
1- Hindsight Metadata patterns emerging from historical data

2- Insight Deep understanding of issues or problems

3- Foresight Accurate prediction in near future in a cost-effective manner

Ref:
Big Data: Principles and Paradigms, Chapter 1: Big Data Analytics = Machine Learning + Cloud Computing

Moore's Law (1/3)

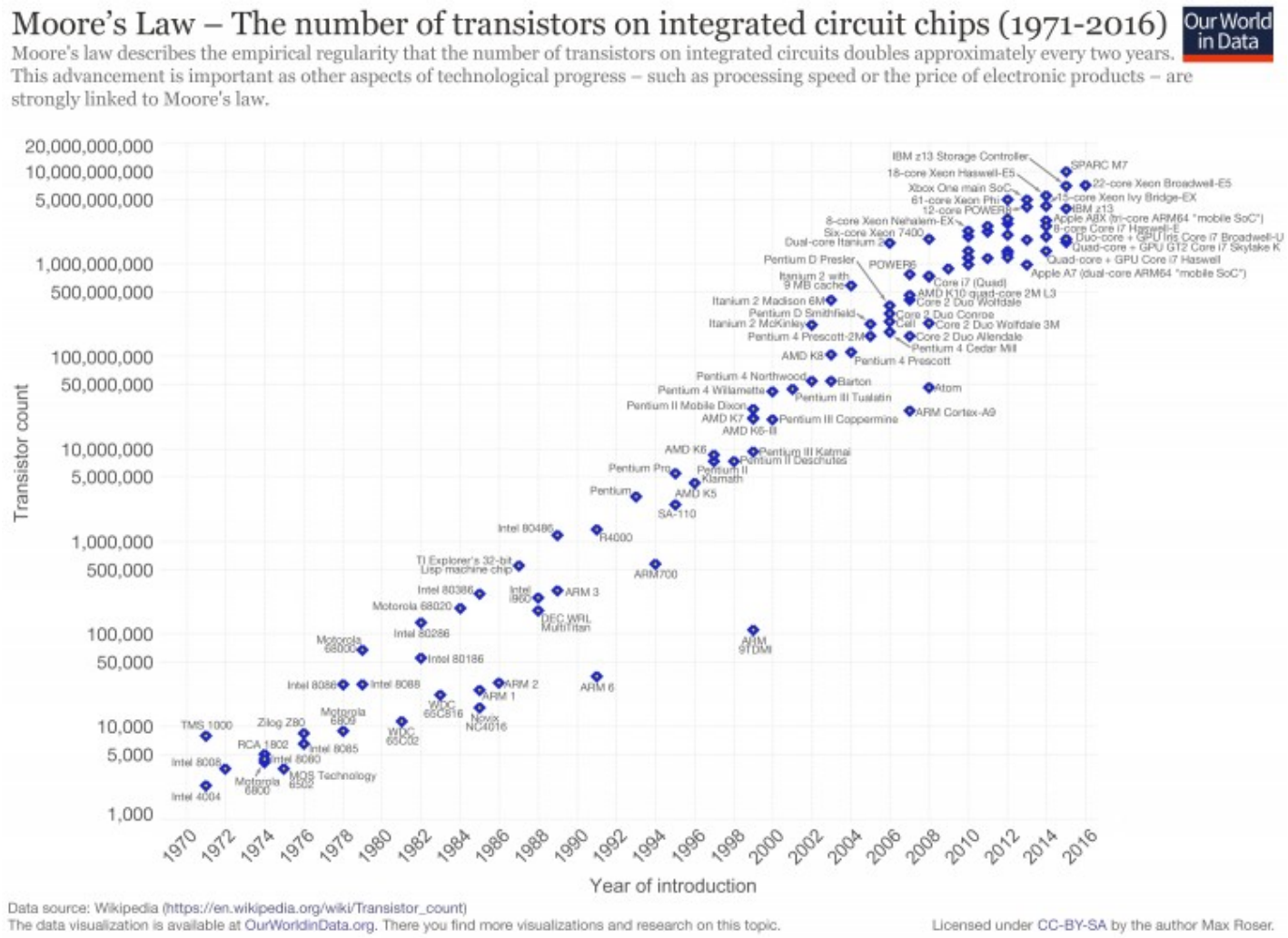


Engineers at a processor fabrication plant assessing a wafer

Processor fabrication is an expensive nanometer scale process, which exploits cutting-edge material science, electromagnetics & optical technologies.

Ref: https://ourworldindata.org/technological-progress/

Moore's Law (2/3)



According to Moore's Law, there will be an exponential growth with transistor count doubling every two years.

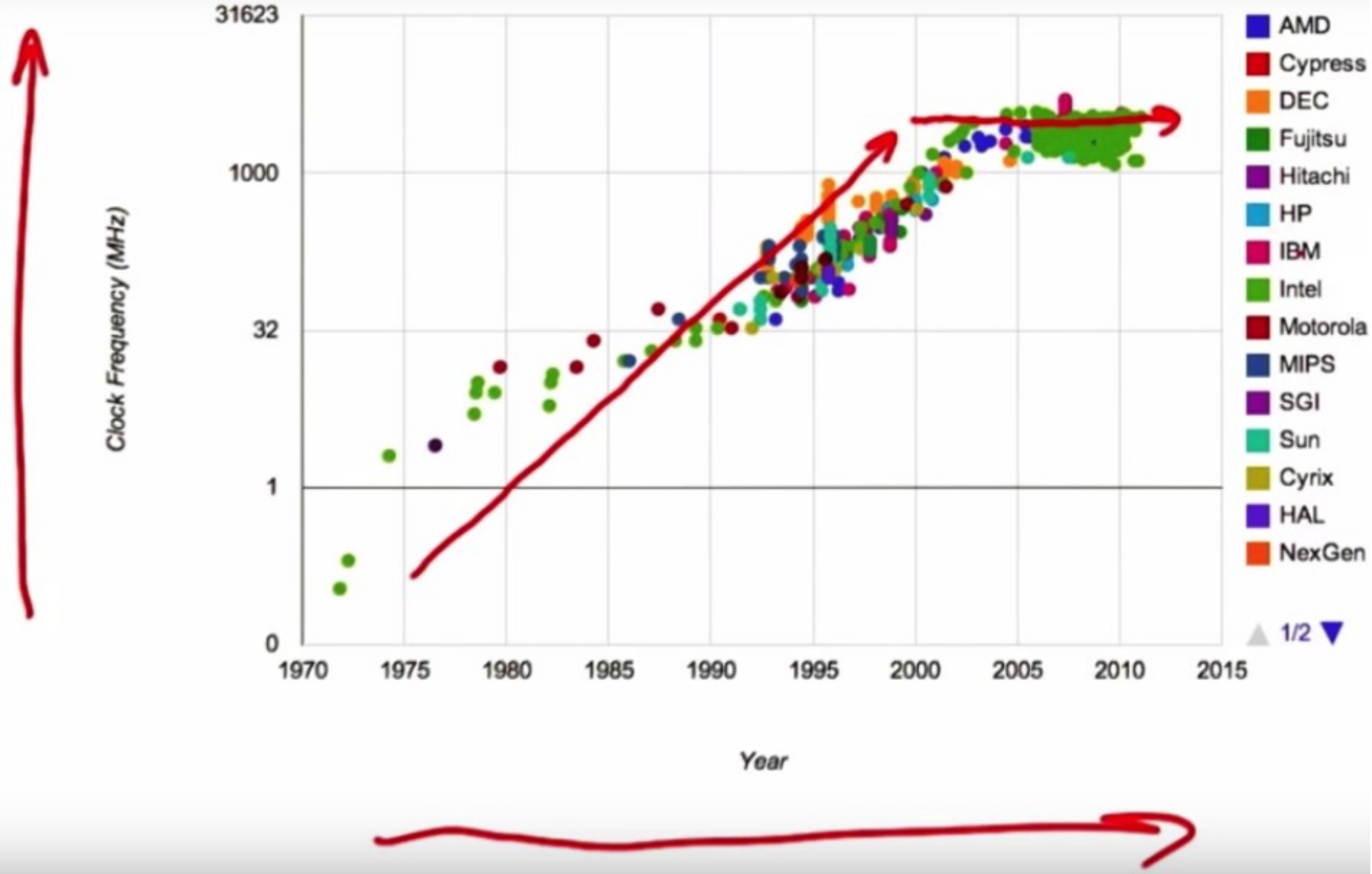
Nowadays, there is a serious debate on validity of Moore's Law between hardware vendors and other stakeholders.

Doubling the transistor density (thereby CPU Clock Speed) each two years is getting more technically complicated and costly.

Ref: https://ourworldindata.org/technological-progress/

Note: Logarithmic scale vertical axis chosen to show the linearity of the growth rate.

Moore’s Law (3/3)



Increase in processor clock rates has been the primary driver of processor performance, but over the last decade clock speeds remained constant. (hardly reached to 5GHz in 2018)

Increase in clock rates is not the only criterion of processing power per dollar.

As higher transistor density means:

- Higher temperature & less reliability. expensive chips and cooling systems
- More power consumption increases costs, sometimes we need low energy processors (IoT)

Variations of Hardware for Parallelism

Parallel Data Processing Hardware:

- Multi-core Processors (CPU)
- Symmetric Multiprocessors (CPU)
- General Purpose Graphics Processing Unit (GPU)
- Field-Programmable Gate Arrays (FPGA)
- Tensor Processing Unit (TPU/ ASICs)

Storage Hardware for Data Parallelism:

- Random Access Memory
- SSD Disks
- SATA Disks
- Network Attached Storage
- Storage Area Network

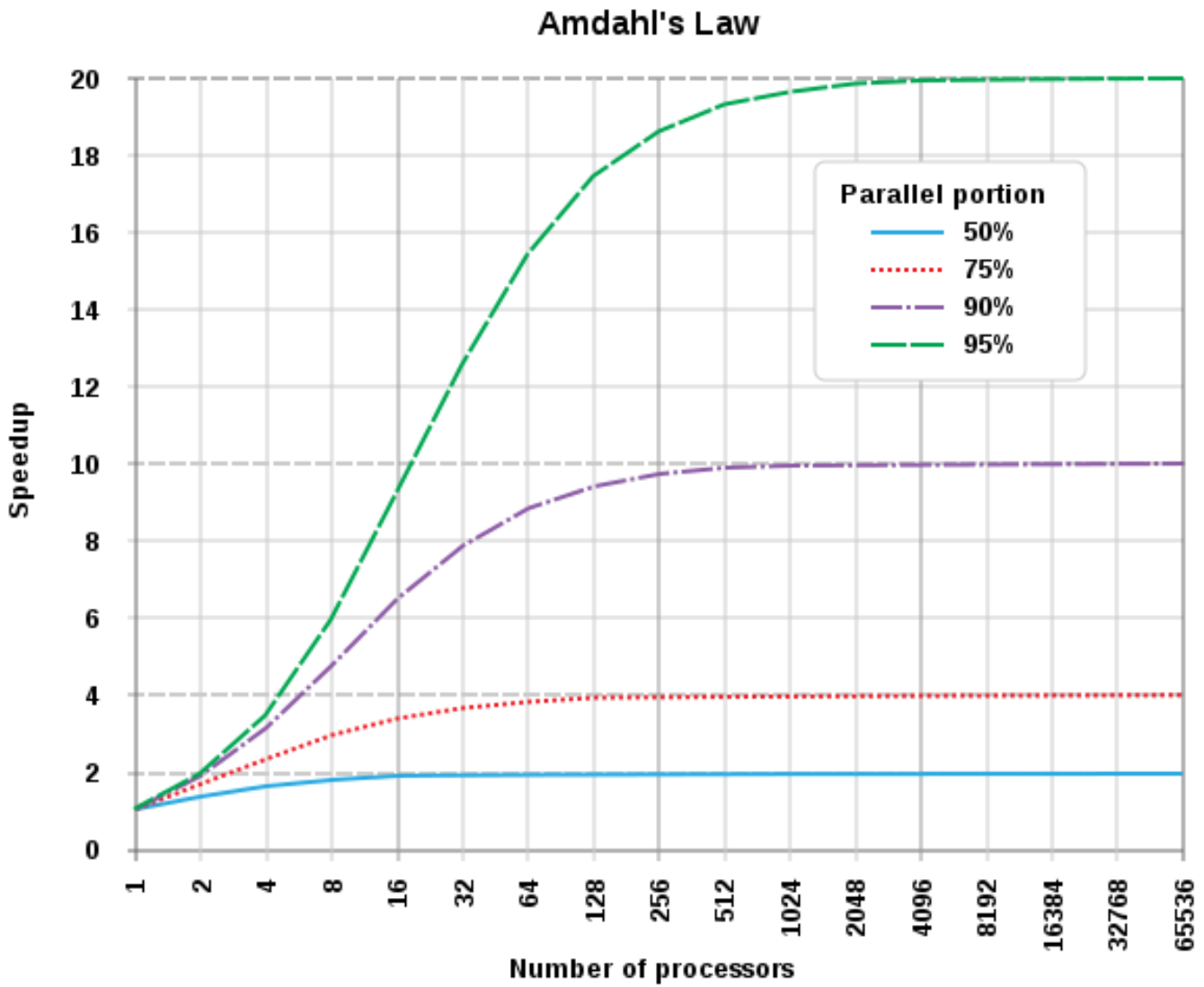
- Computer Clusters with many hosts/nodes

Amdahl's Law

The "theoretical speedup" of the execution of the whole task

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

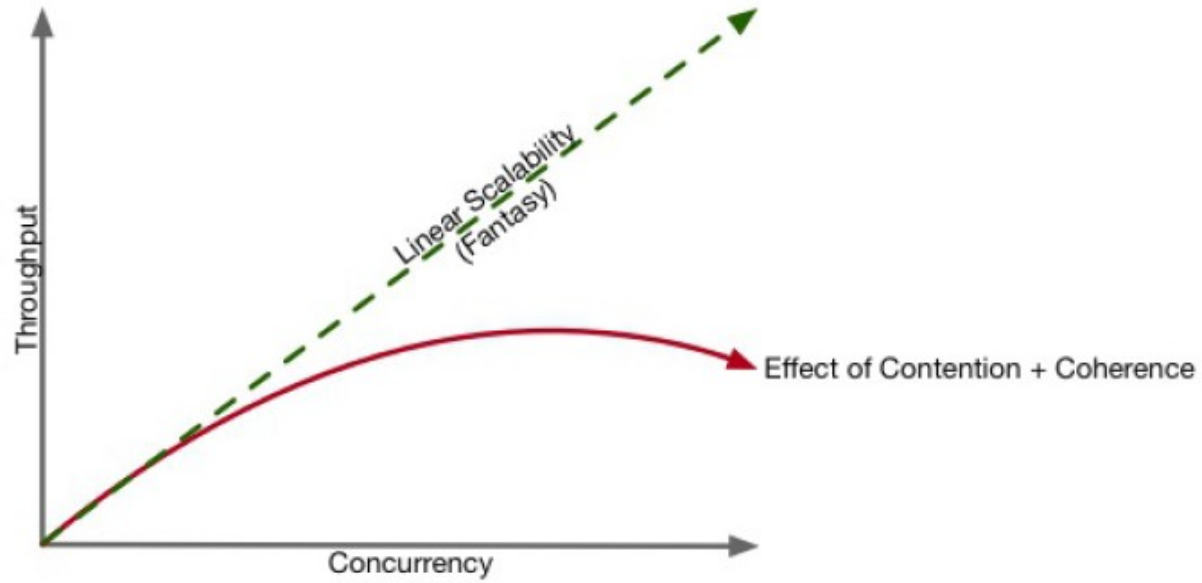
$$\begin{cases} S_{\text{latency}}(s) \leq \dfrac{1}{1-p} \\ \lim\limits_{s \to \infty} S_{\text{latency}}(s) = \dfrac{1}{1-p}. \end{cases}$$



The theoretical speedup by parallelism is always limited by the part of the task that cannot benefit from the improvement !

Ref: https://en.wikipedia.org/wiki/Amdahl%27s_law

Gunther’s Universal Scalability Law



- Gunther's Universal Scalability Law builds on Amdahl's Law.
- In addition to contention, it accounts for Coherency Delay.
- Coherency Delay results in negative returns.
- As the system scales up, the cost to coordinate between nodes exceeds any benefits.

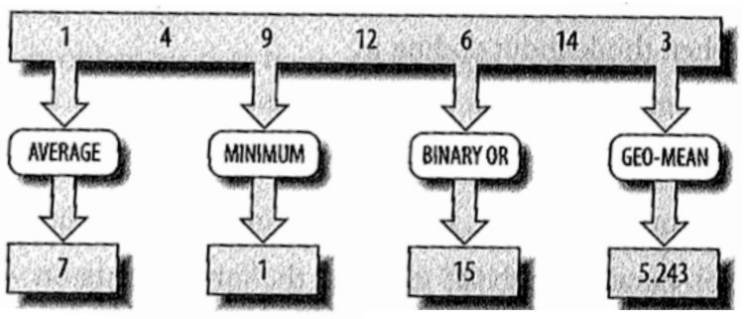
Women have always pioneered Computer Science!

Types of Parallelism

Task Parallelism:

A form of parallelization that distributes different independent tasks **on the same data** *across computing nodes/hosts.*

Example: Several functions on the same data

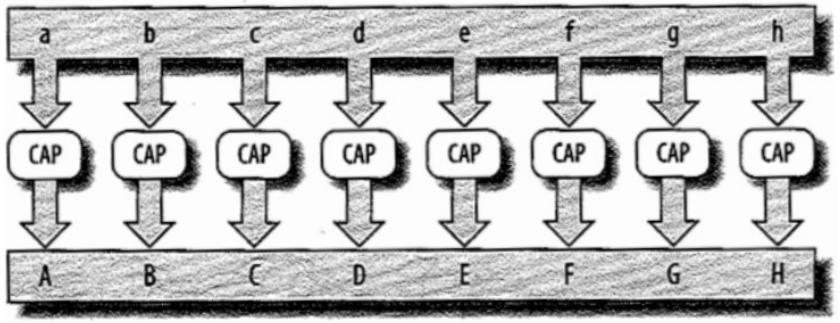


Platforms:
Apache Storm

Data Parallelism:

A form of parallelization that distributes data across computing nodes/hosts & executes the same task via independent invocations over subsets/blocks of distributed data.

Example: convert all characters in an array to upper-case



Platforms:
Apache Spark (initially)
Apache MapReduce

Hybrid Data & Task Parallelism:

A form of parallelization where a parallel pipeline of tasks execute over the ***same subsets of distributed data****.*

Example: A ML algorithm may employ vertical data parallelism in the beginning of the algorithm and then it changes the parallelism type to task parallelism in the latter stage of the algorithm.

Platforms:
Apache YARN
Apache Spark (recently)

Scaling Up versus Scaling Out

Large datasets can be analyzed and interpreted in two ways:

- Distributed Processing – use many separate commodity nodes, where each analyze a portion of the data.

  This method is sometimes called as scaling-out or horizontal scaling.

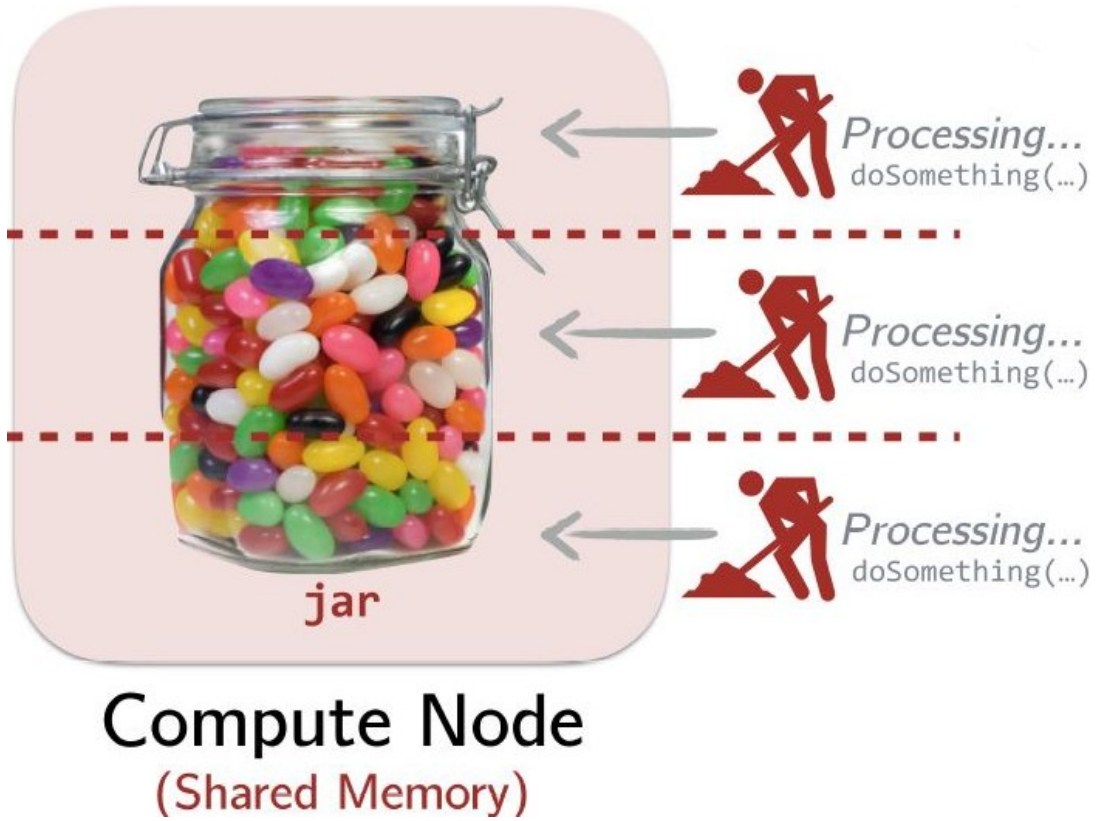
  Potentially *Low Infrastructure cost (CAPEX), High Maintenance cost (OPEX)*
- Shared Memory Processing – use large systems with enough resources (sometimes even specialized hardware) to analyze huge amounts of the data.

  This method is sometimes called as scaling-up or vertical scaling.

  Potentially *High Infrastructure cost (CAPEX), Low Maintenance cost (OPEX)*

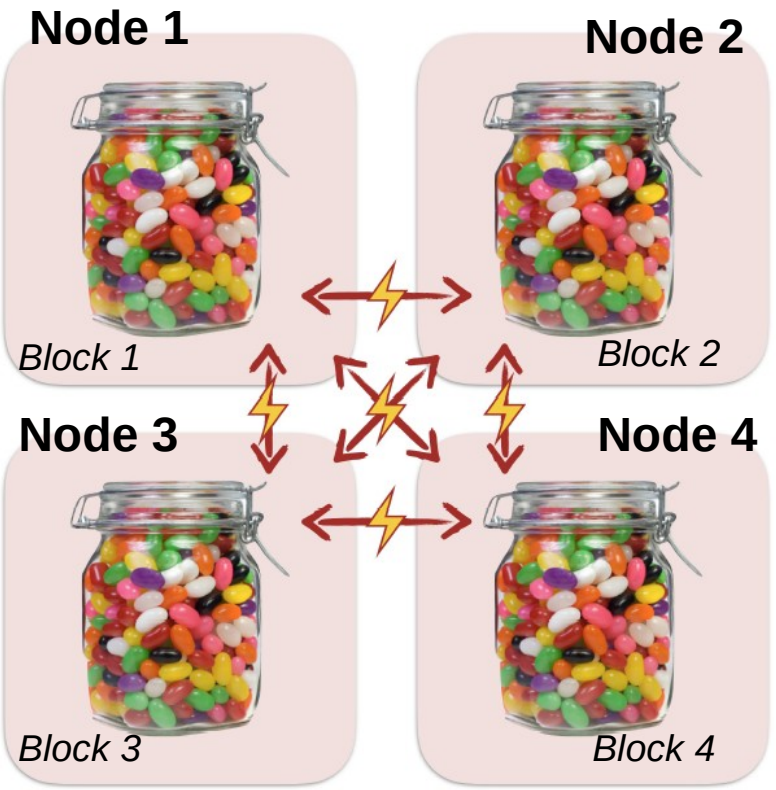
Data Parallelism

Shared Memory Data Parallelism



Data parallelism in a multi-core/multi-processor single machine

Distributed Data Parallelism



Data parallelism in a cluster with 2 or more machines

Data Parallelism Stored in Memory

Big Data

Word	Index	Count
I	0	I
am	2	I
Sam	5	I
I	9	I
am	11	I
Sam	14	I

I	0	I
am	2	I

Partition 1

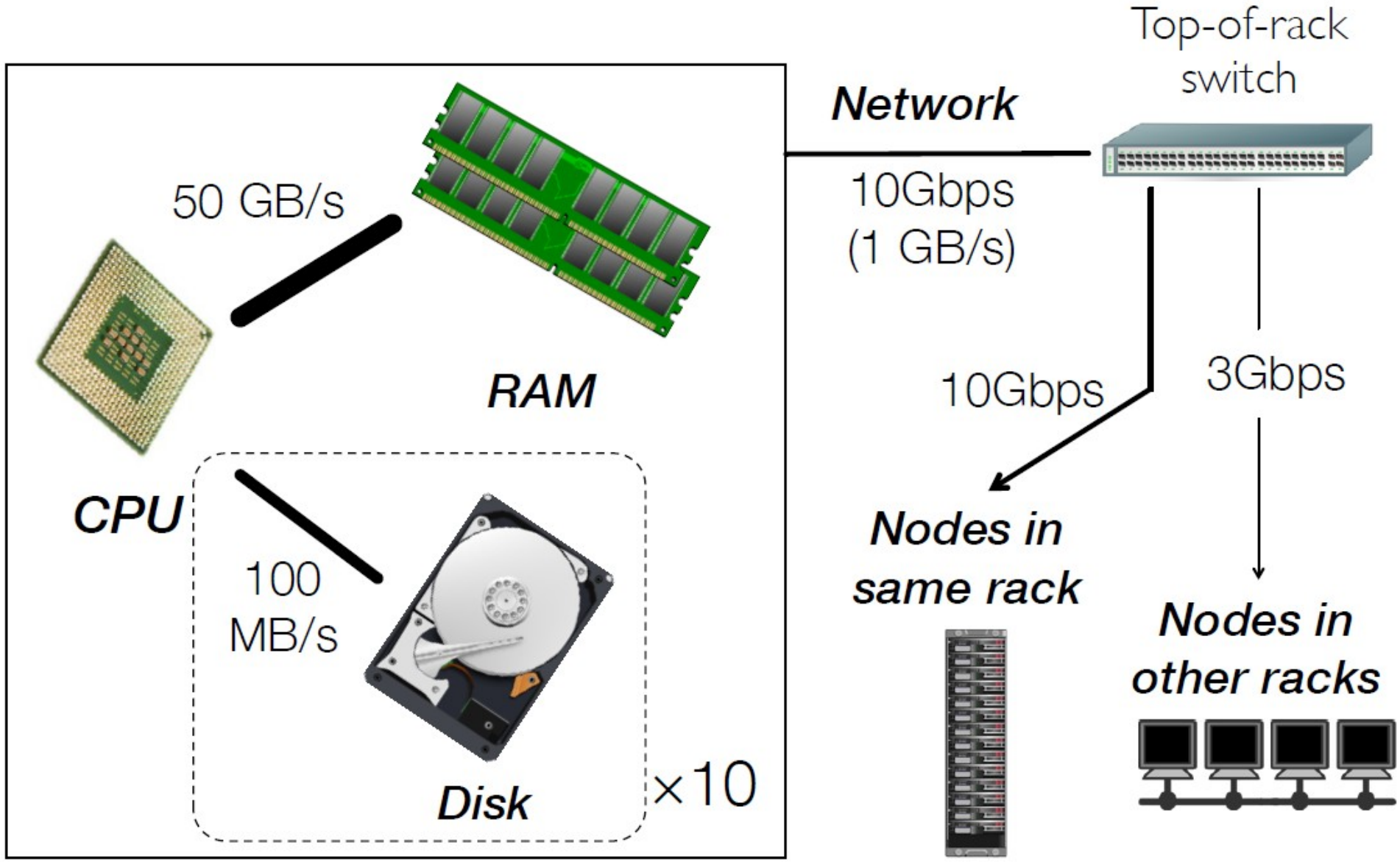
Sam	5	I
I	9	I

Partition 2

am	11	I
Sam	14	I

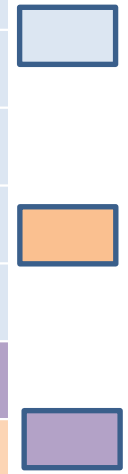
Partition 3

Communication Hierarchy in a Typical Data Center



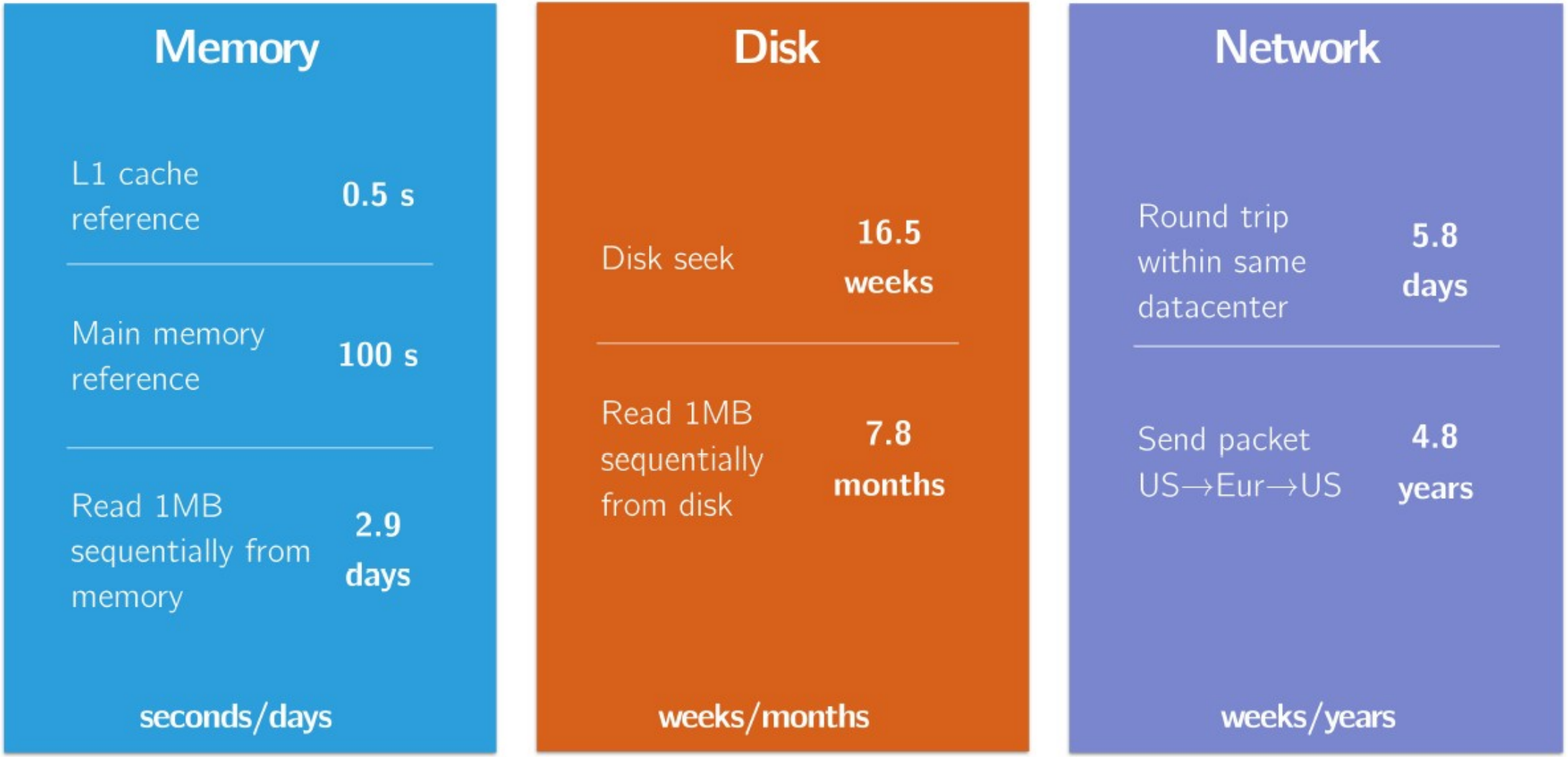
Example Latency Results for Different Operations

Reference Data Tasks	Latency	Multiplied by 1 billion
L1 CPU cache reference	0.5 ns	0.5 s
Branch mispredict	5 ns	5 s
L2 CPU cache reference	7 ns	7 s
Mutex lock/unlock	25 ns	25 s
Main memory reference	100 ns	100 s
Send 2KBs over 1 Gbps network	20,000 ns= 20 µs	5.5 hr
SSD (Fast Disk) Random Read	150,000 ns= 150 µs	1.7 days
Read 1 MB sequentially from memory	250,000 ns= 250 µs	2.9 days
Rountrip within the same datacenter network	500,000 ns= 0.5 ms	5.8 days
Read 1 MB sequentially from SSD (Fast Disk)	1,000,000 ns= 1 ms	11.6 days
SATA Disk Seek	10,000,000 ns= 10 ms	16.5 weeks
Read 1 MB sequentially from SATA Disk	20,000,000 ns= 20 ms	7.8 months
Send packet US -> Europe -> US	150,000,000 ns= 150 ms	4.8 years

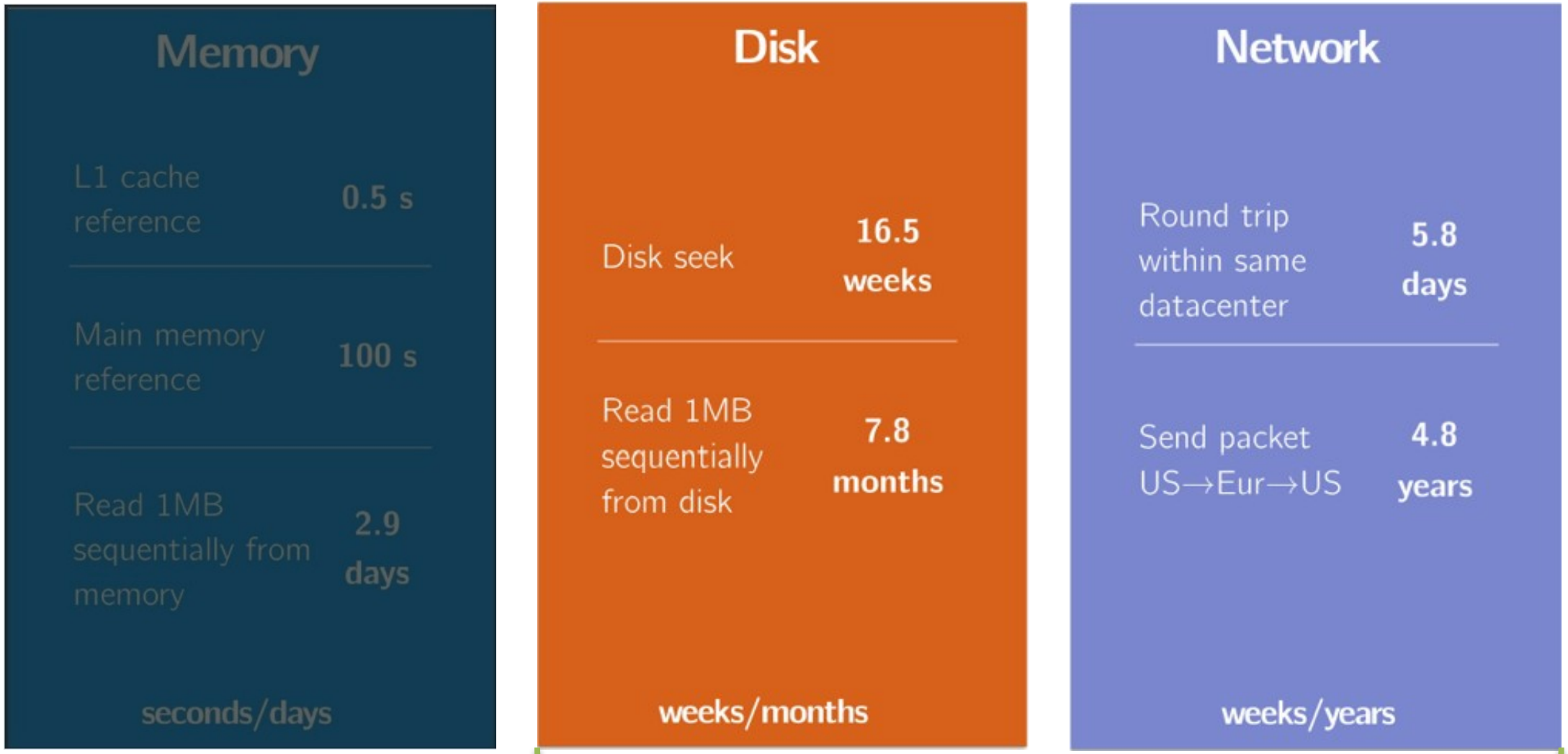


Original compilation by Jeff Dean & Peter Novig

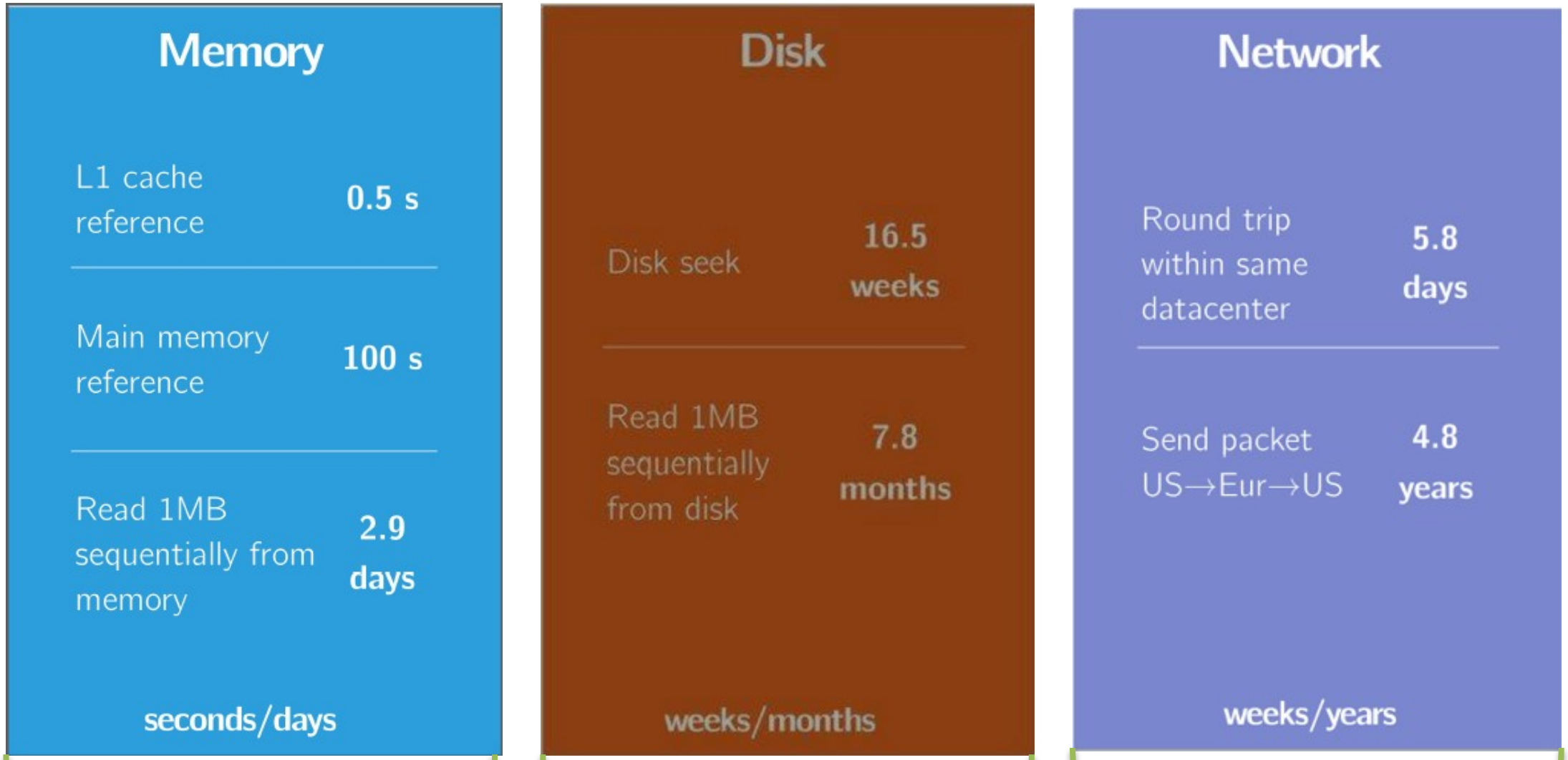
Latency in a real life big data scenario (Multiplied with 1 Billion)



Latency in Hadoop MapReduce (Multiplied with 1 Billion)



Latency in Spark (Multiplied with 1 Billion)



Aggresive usage of memory by Spark Datasets!

Optional usage of disk if needed!

Aggresively minimized data shuffling over the network !

Challenges of Distributed Systems (1/2)

- Latency

  Data storage, reading & transfer operations might have significant difference based on the involvement of different computer hardware

- Distributed Data Abstraction & Handling:

  *We must consider network, data locality**

  Transfering data between nodes or processes may be very time consuming depending on your network bandwith

Note:* Data locality became less of your concern with Terabit/Petabit per second Network Bandwith in Cloud Platforms

Challenges of Distributed Systems (2/2)

- Fault Tolerance:

  Process & Node Failures:

  Likelihood of singe machine's failure is very high for commodity hardware.

  Note: Imagine 1 server fails every 3 years with 10,000 nodes you would observe 10 faults/day

  It is also necessary to deal with the nodes with poor performance (in terms of speed, space etc.)

  Network Failures:

  Dropped messages

  Lost Partitions

  Network Congestions

  Routing Errors

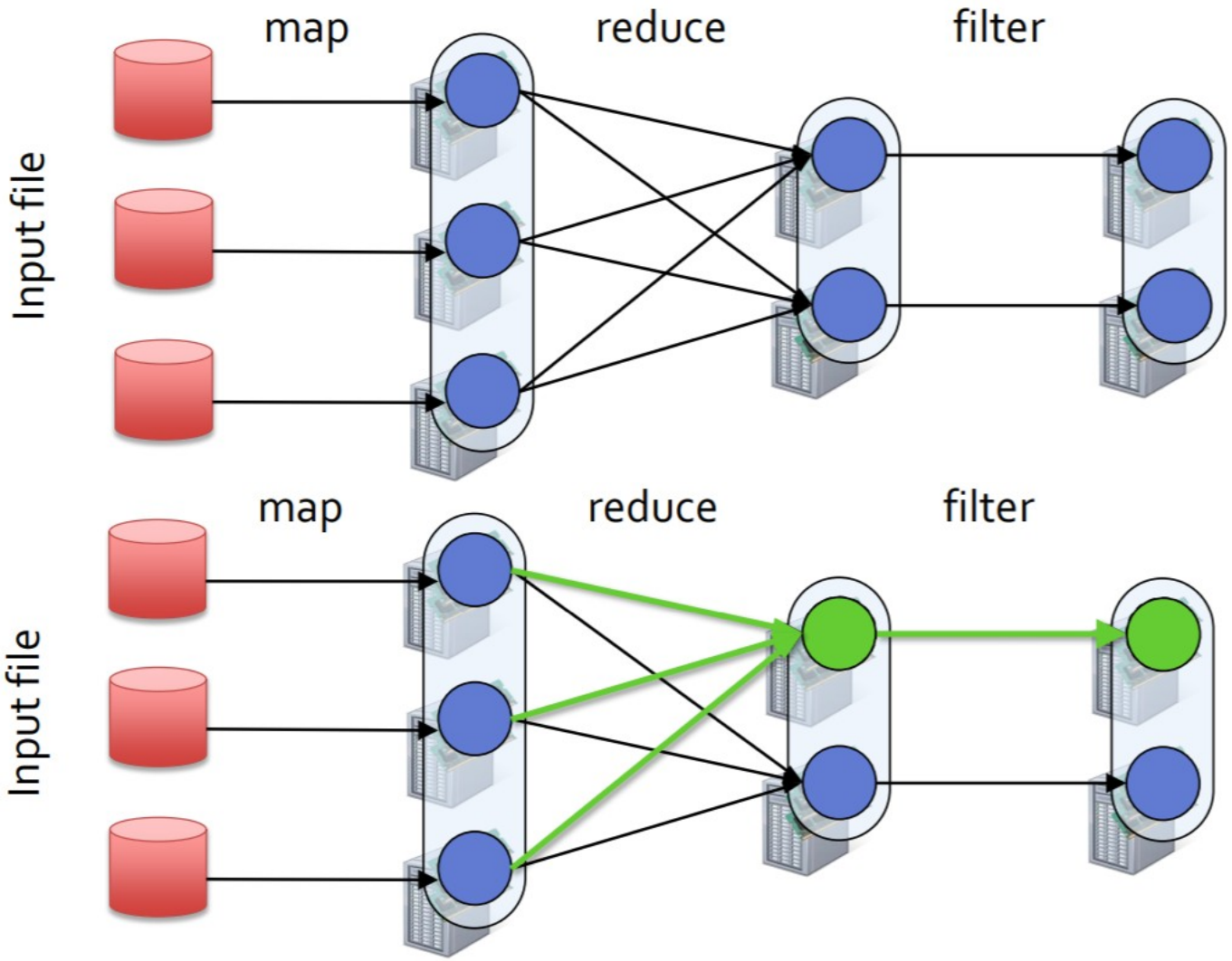
Fault Tolerance in Hadoop versus Spark

Fault tolerance in Hadoop MapReduce framework has an additional cost. Between each map & reduce step, as a safeguard against potential failures, intermediary data outputs have to be written / serialized to disk.

Spark keeps all data in ***immutable, in-memory**** and ***lazily evaluated*** variables / collections called RDDs.

Spark keeps the ***data lineage (récipe)*** of these in memory datasets across all pipeline, so that in case of a failure during processing, it replays all this récipe (RDD Lineage) up to the immutable source dataset**.

Fault Tolerance in Spark



Each Spark Dataset (RDD) keeps the lineage informartion which can be used to reconstruct the dataset in case of failure.

This example shows a mapped, reduced & filtered dataset block reconstructed using Spark dataset (RDD) lineage.

What Apache Spark is not? (1/3)

Not a Data Store:

Spark connects to other data stores but it does not provide its own database or filesystem.*

Spark can access & store data in HDFS, AWS S3, Google Cloud Storage, DBFS, Hbase, Hive, Alluxio, external SQL, NoSQL databases or any Hadoop data source

Not a Programming Language:

It is a unified Big Data Framework which provides an Application Programming Interface (API) for Scala, Java, Python, R and SQL. Nevertheless, it is built in Scala Programming Language.

Note:* We will learn the "Spark Storage" functionality and setting, it still doesn't mean that Spark can be classified as a Data Store.

What Apache Spark is not? (2/3)

Not only for Big Data SQL:

It is also used for Machine Learning, Streaming and Graph Analytics at scale.

Not *a substitute* for all Hadoop Big Data Ecosystem:

It is correct that is a substitute for Hadoop MapReduce, Pig, Sqoop, Hive and Tez, Mahout, but it is also common to integrate it with HDFS, Hive Metastore, Oozie, Kafka, Solr, Hbase, NIFI etc.

What Apache Spark is not? (3/3)

Not *only* for Hadoop :

Spark can work with Hadoop & Yarn, but Spark is a standalone system

Spark Application Code can also be executed on Mesos (DC/OS), Kubernetes, Databricks Runtime, AWS Glue as an alternative to any Hadoop Distribution.

Not *only* for Streaming & Real-time Analytics:

Apache Spark is oftenly used for In-Memory Batch Processing, apart from being an alternative to Storm, Flink, Beam, Samza, Heron, Ignite (which are streaming tools).

Programming Models for Big Data

A **Programming Model** is an abstraction or existing machinery or infrastructure.

It is a set of abstract *runtime libraries* and *programming APIs* that form a model of computation.

MapReduce and Spark are **Big Data Programming Models**.

Traits of a Big Data Programming Model:

I. Splitting large volumes of data

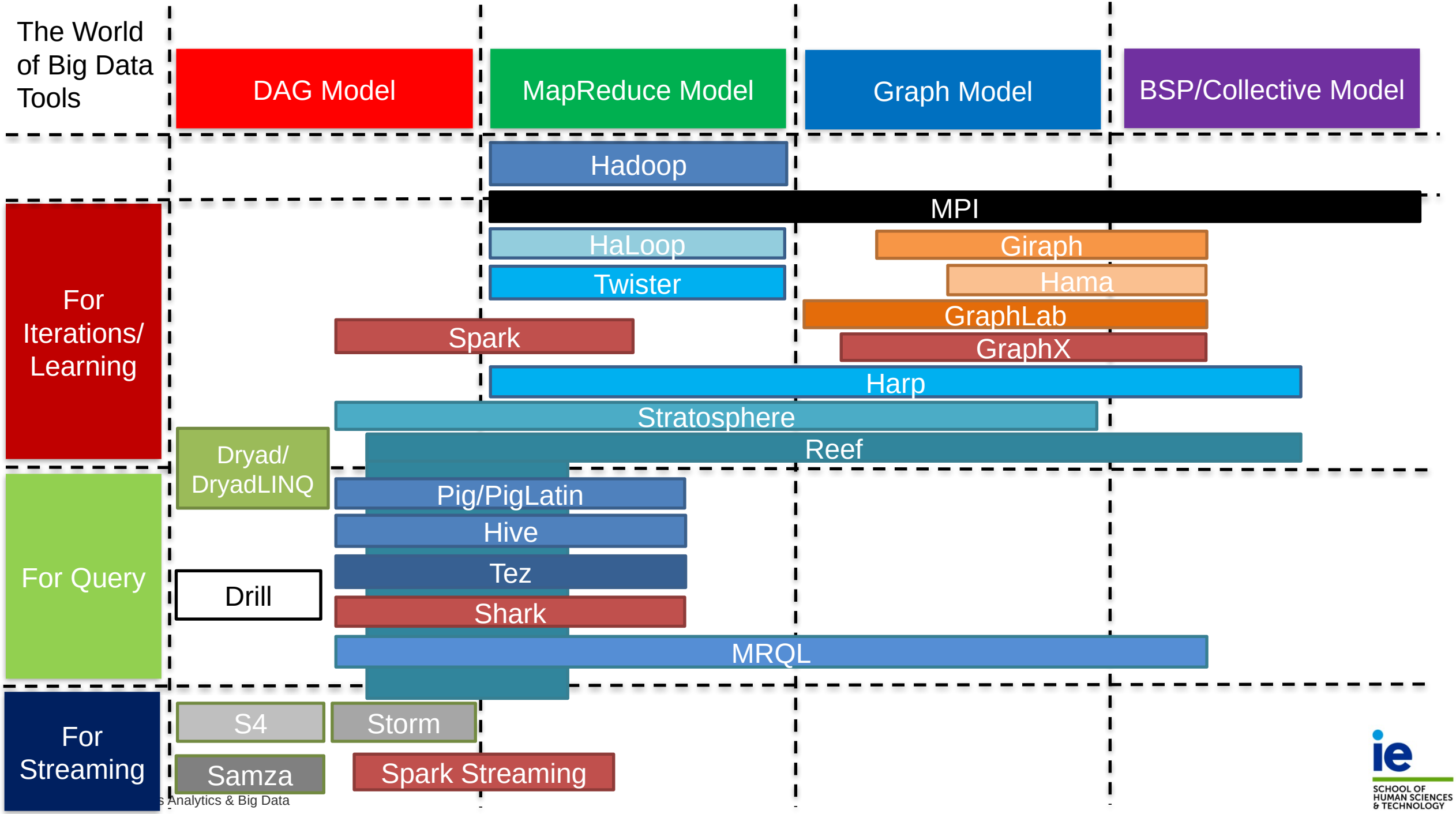
II. Fast access to data

III. Scalable & fast distribution to nodes

IV. Enabling reliability of the computing and full tolerance from failures

V. Allowing to add new resources

VI. Enable operations over a particular set of these types

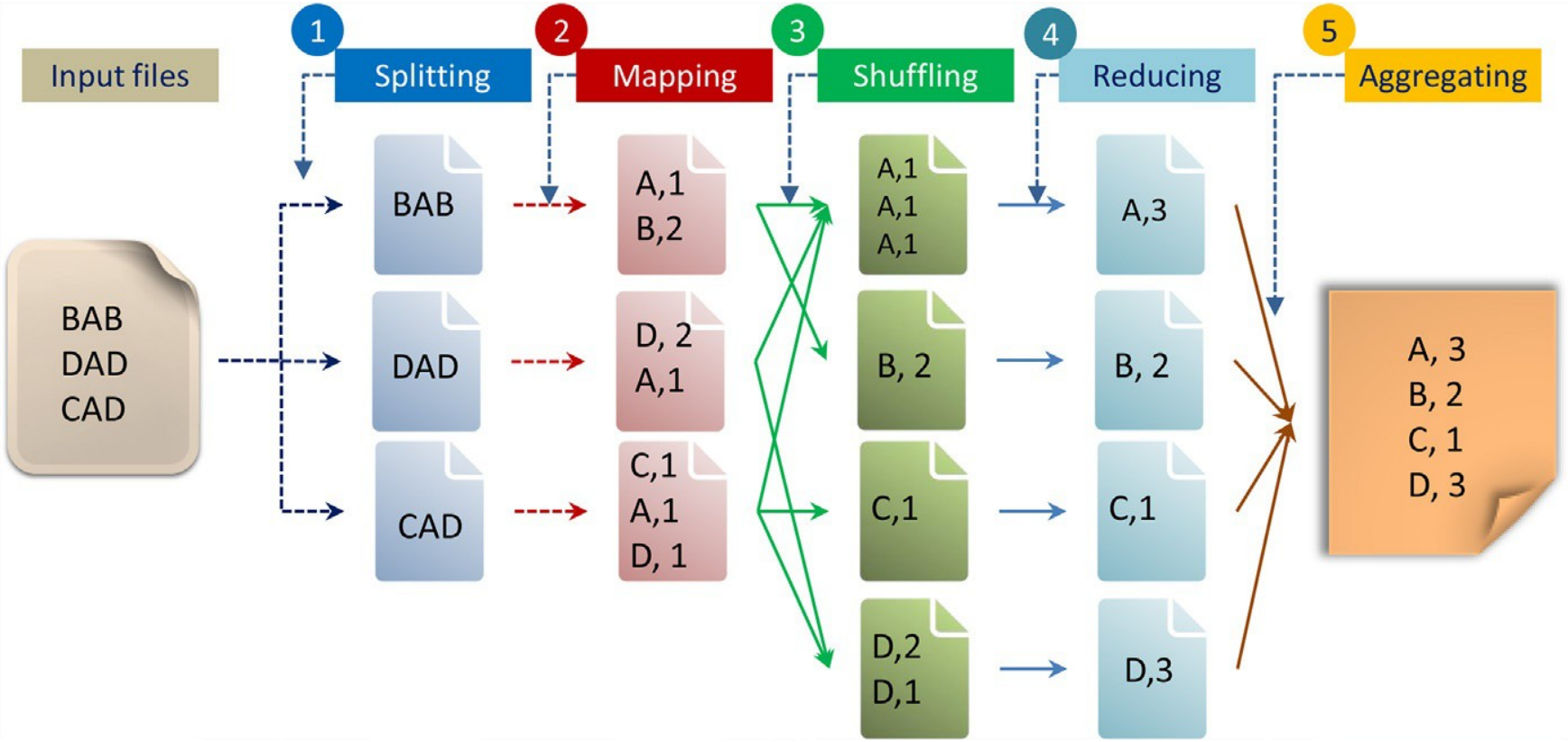


MapReduce as a Programming Model

MapReduce is a *programming model* used to process large dataset workloads in parallel.

The basic strategy of MapReduce is to divide and conquer.

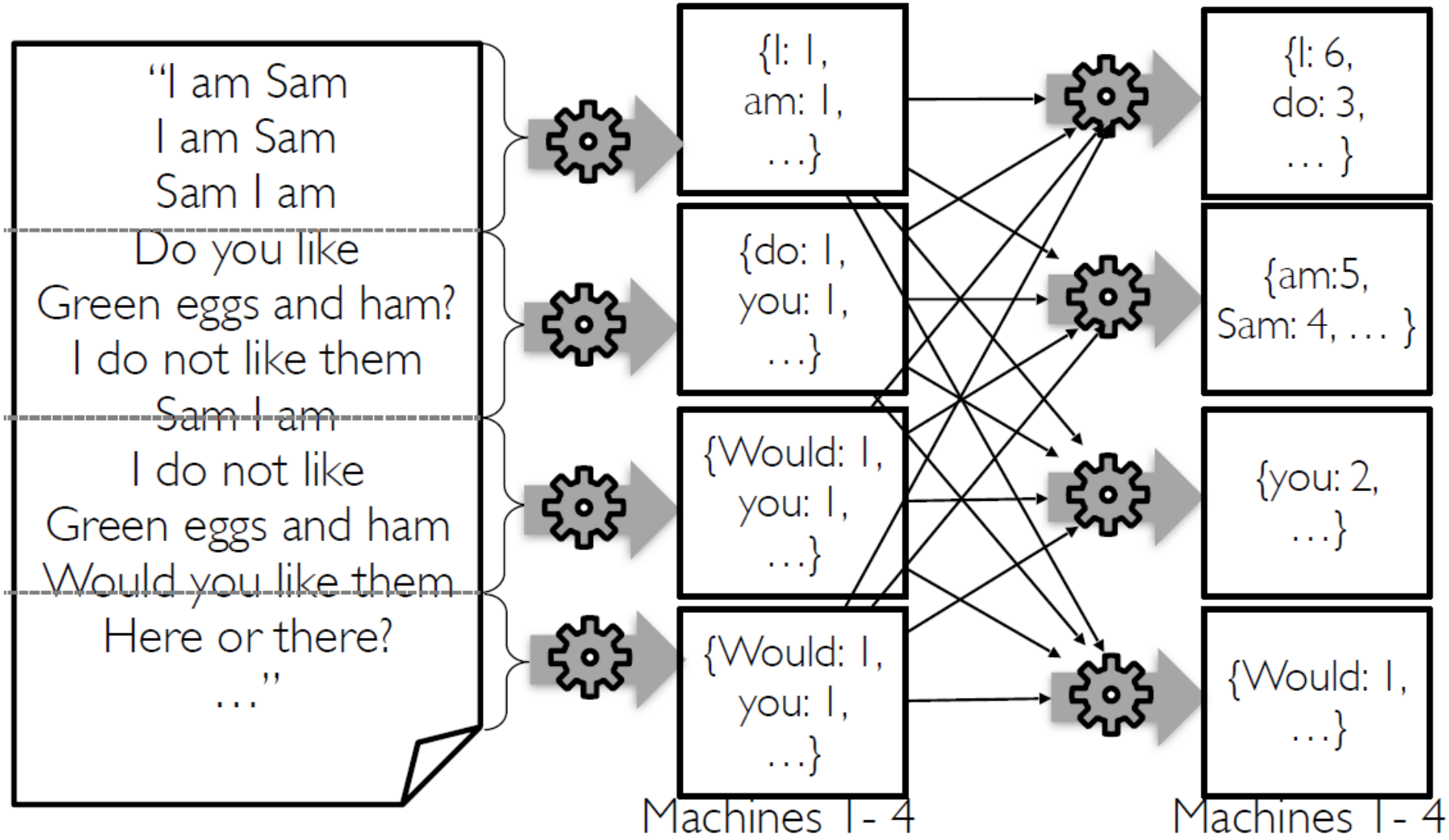
MapReduce treats computation as the evaluation of mathematic functions. In essence, Functional Programming in MapReduce can avoid In-progress state and just list in-and-out states.



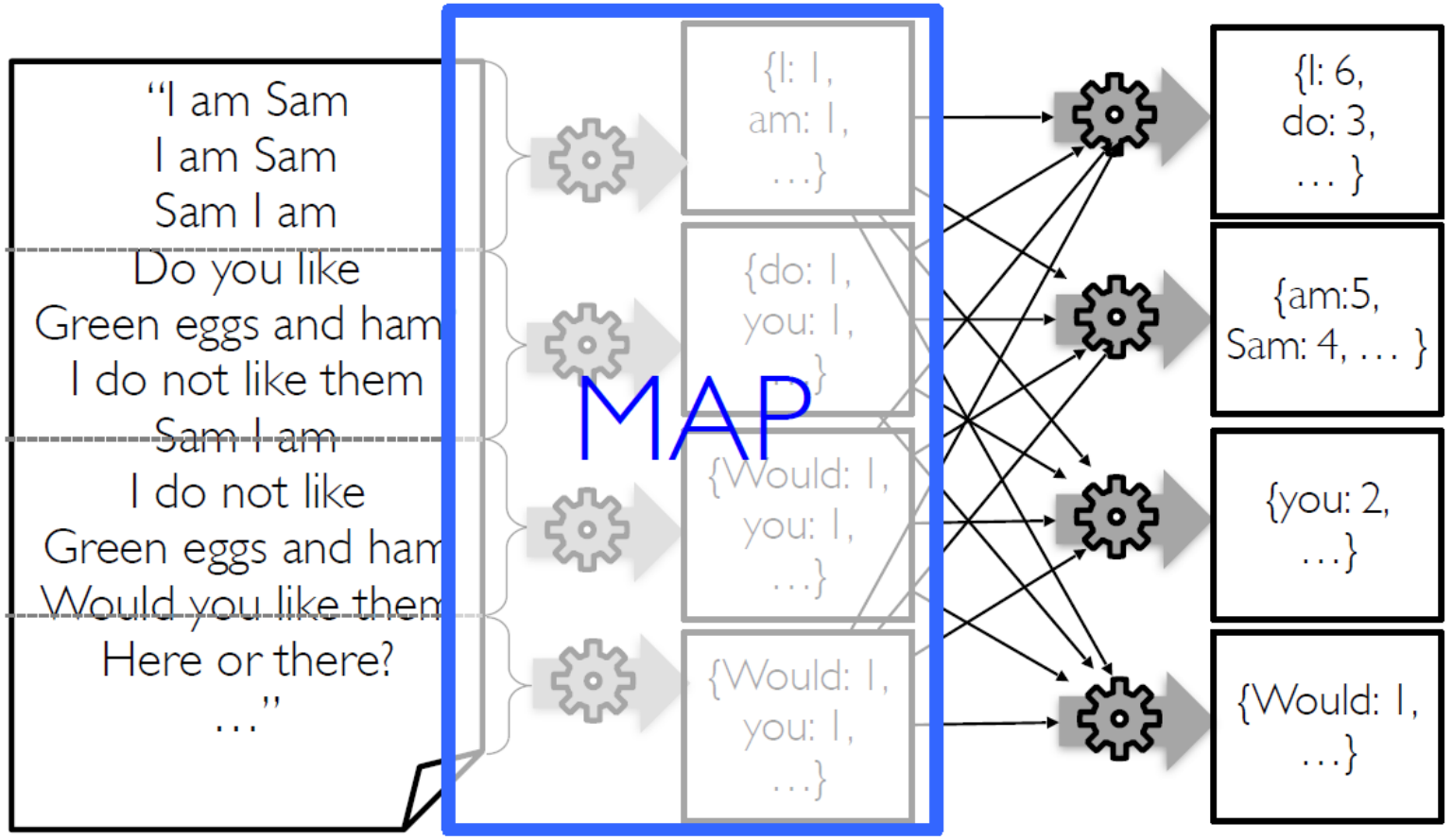
Ref:
Big Data: Principles and Paradigms, Chapter 1: Big Data Analytics = Machine Learn + Cloud Computing

Actually, MapReduce Paradigm has some variations like *Iterative MapReduce* & *Twister* paradigms.

Modelling a massive Word Count problem in MapReduce (1/3)



Modelling a massive Word Count problem in MapReduce (2/3)



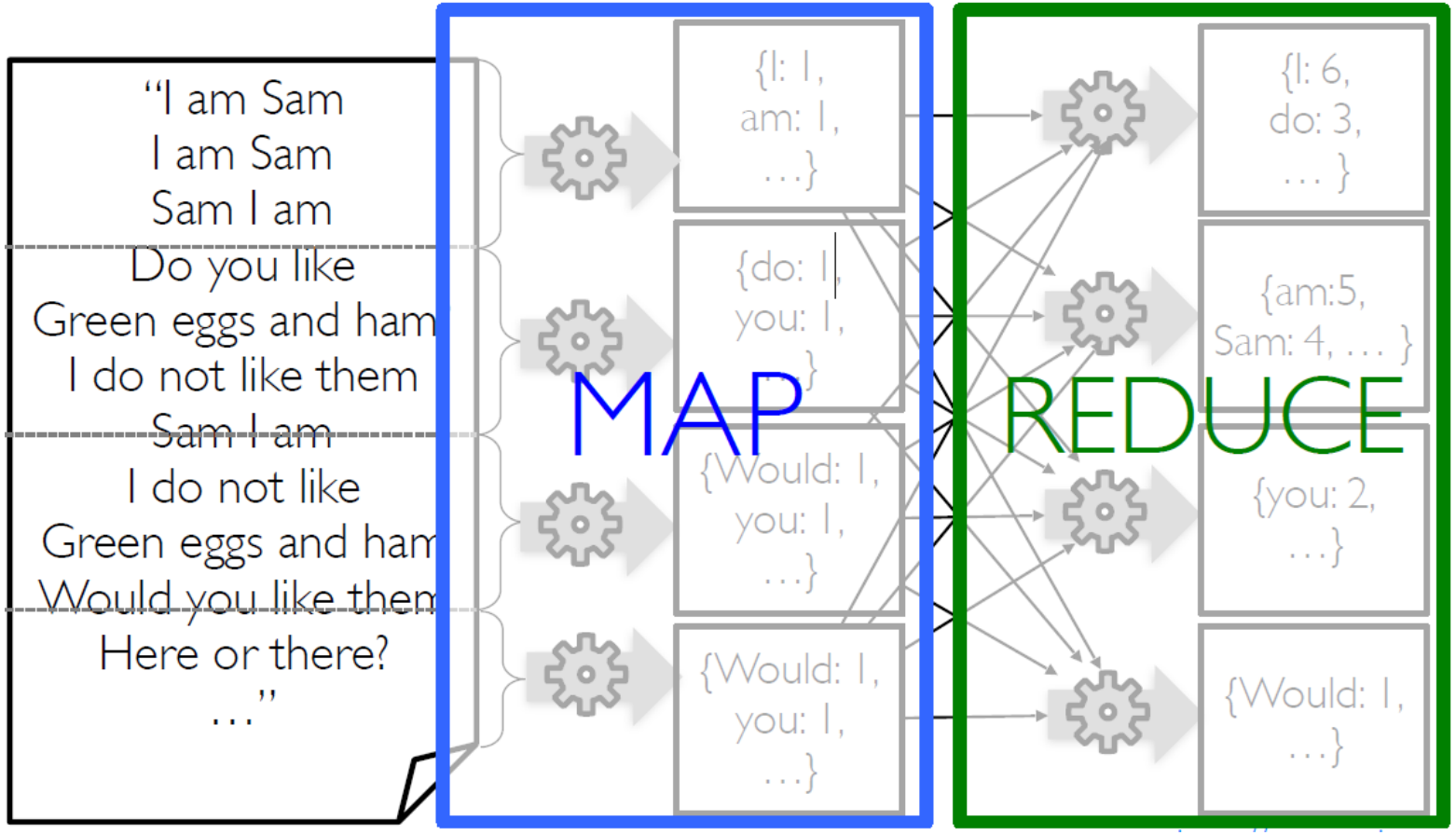
"Mapping" requires splitting for distribution.

- *map()* is *a high-order function* from functional programming perspective which invokes a function/operation over each element of an Iterable/Collection.
- In this example each element of the Iterable/Collection is a sentence; but it could be anything.

A major advantage of MapReduce is its shared-nothing data processing behaviour, which means all mapper tasks can **process their data partitions independently and in parallel**.

This enables a simple program to run across thousands or even millions of unreliable and homogeneous machines in parallel and to complete a large data processing task in an acceptable amount of time.

Modelling a massive Word Count problem in MapReduce (3/3)



"Reducing" might require shuffling between nodes and sorting in parallel.

reduce() is *a high-order function* from functional programming perspective which invokes a mathematically commutative & associative function/operation over a collection of elements grouped based on a criterion.

MapReduce Paper from Google:
https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

Use Cases of MapReduce

MapReduce is potentially capable to handle five types of workloads:

1. Large-scale ML problems (usually via Mahout)

2. Clustering problems for Google News and Google products

3. Data Extraction to produce reports of popular queries (eg, Google Zeitgeist)

4. Feature Extraction of web pages for new experiments and products (eg, extraction of geographical locations from a large corpus of web pages for localized search)

5. Large-scale graph computations (usually via Giraph)

Ref:
Big Data: Principles and Paradigms, Chapter 1: Big Data Analytics = Machine Learning + Cloud Computing

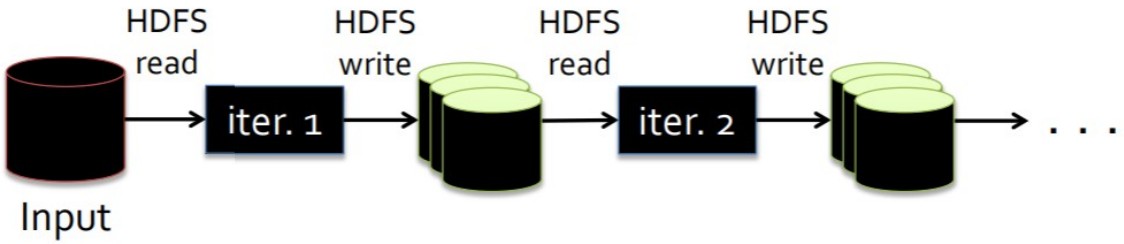
Drawbacks of MapReduce

MapReduce is not very efficient in performing:

I. An iterative and recursive process that is widely utilized for a simulation type of workload in ML

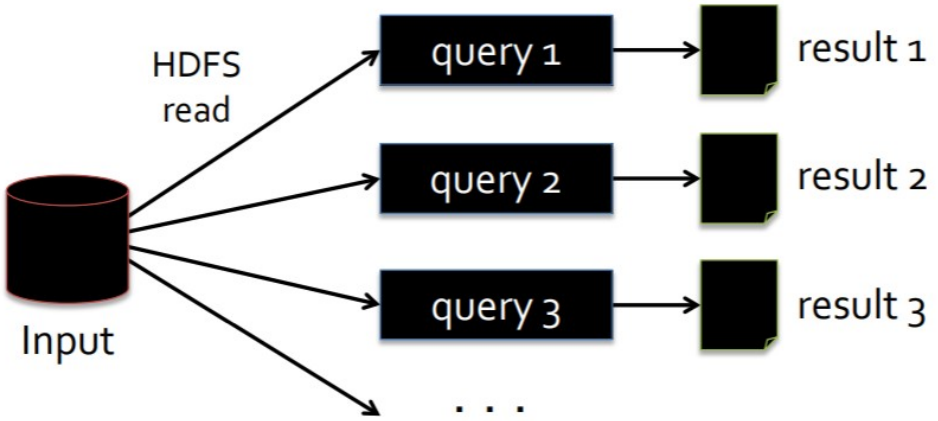
II. An interactive query during Exploratory Data Analysis tasks of Data Scientists.

Iterative Process in MapReduce



Imagine training a Logistic Regression model seeking the best fit recalculating a gradient over a dataset.

Interactive Query in MapReduce

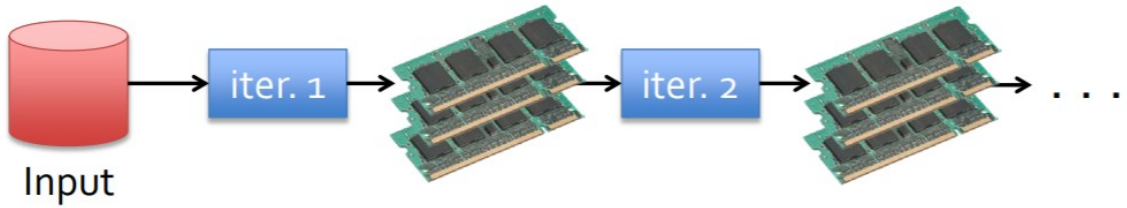


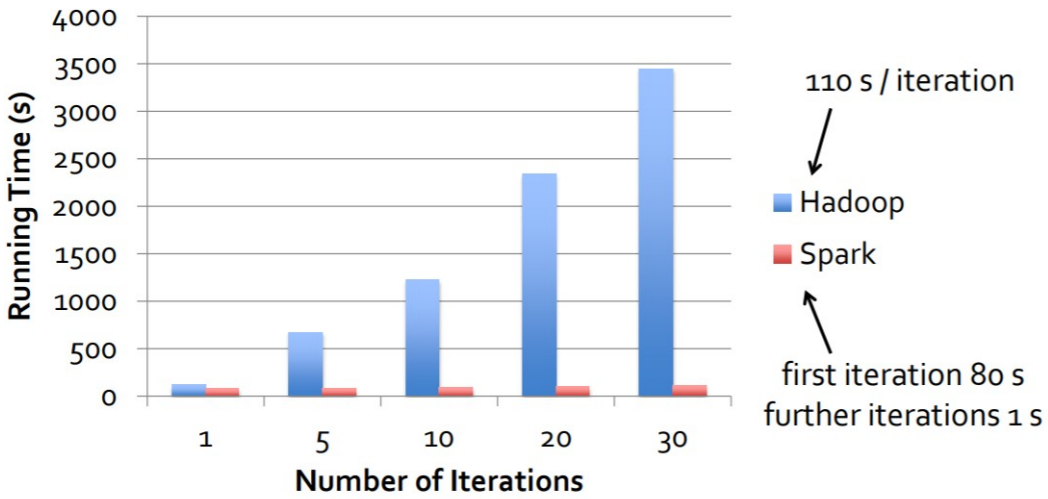
Imagine a Data Scientist exploring the same massive dataset by executing queries on it.

Both of them can be quite cumbersome operations due to replication, serialization, Disk I/O Latency.

Iterative Processes & Interactive Queries in Spark

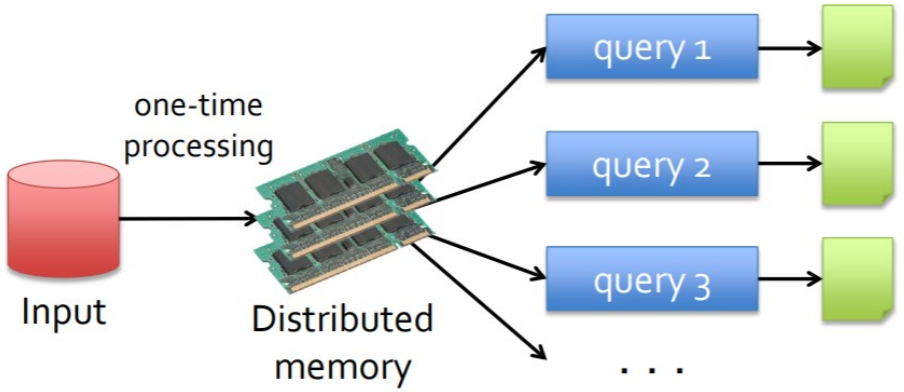
Iterative Processing in Spark





Spark implementation of Logistic Regression have run **100x faster than** previous MapReduce based implementations

Interactive Query in Spark



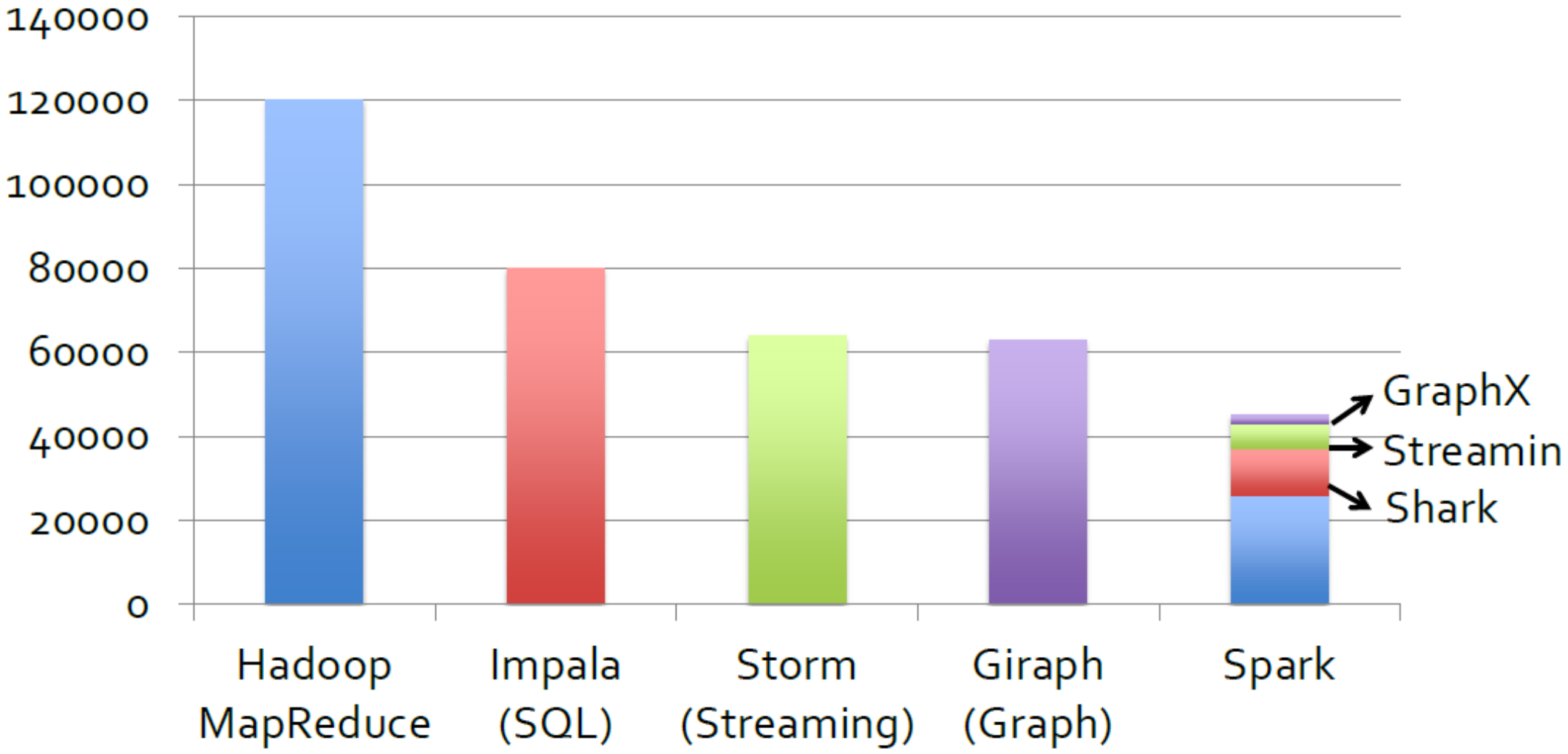
Reference Data Tasks	Latency
Read 1 MB sequentially from memory	250 µs
Read 1 MB sequentially from SSD (Fast Disk)	1 ms
SATA Disk Seek	10 ms
Read 1 MB sequentially from SATA Disk	20 ms

Main Memory versus SSD:
250 µs versus 1ms
4x speedup

Main Memory versus SATA:
250 µs versus 10ms+20ms
120x speedup

Spark can accelerate an iterative or interactive task from **10x** to **100x times** compared to MapReduce !

The Power of Unified BDA Engine: Code Size



Difficulty of Programming in MR

Word Count implementations

- Hadoop MR – 61 lines in Java
- Spark – 1 line in interactive shell

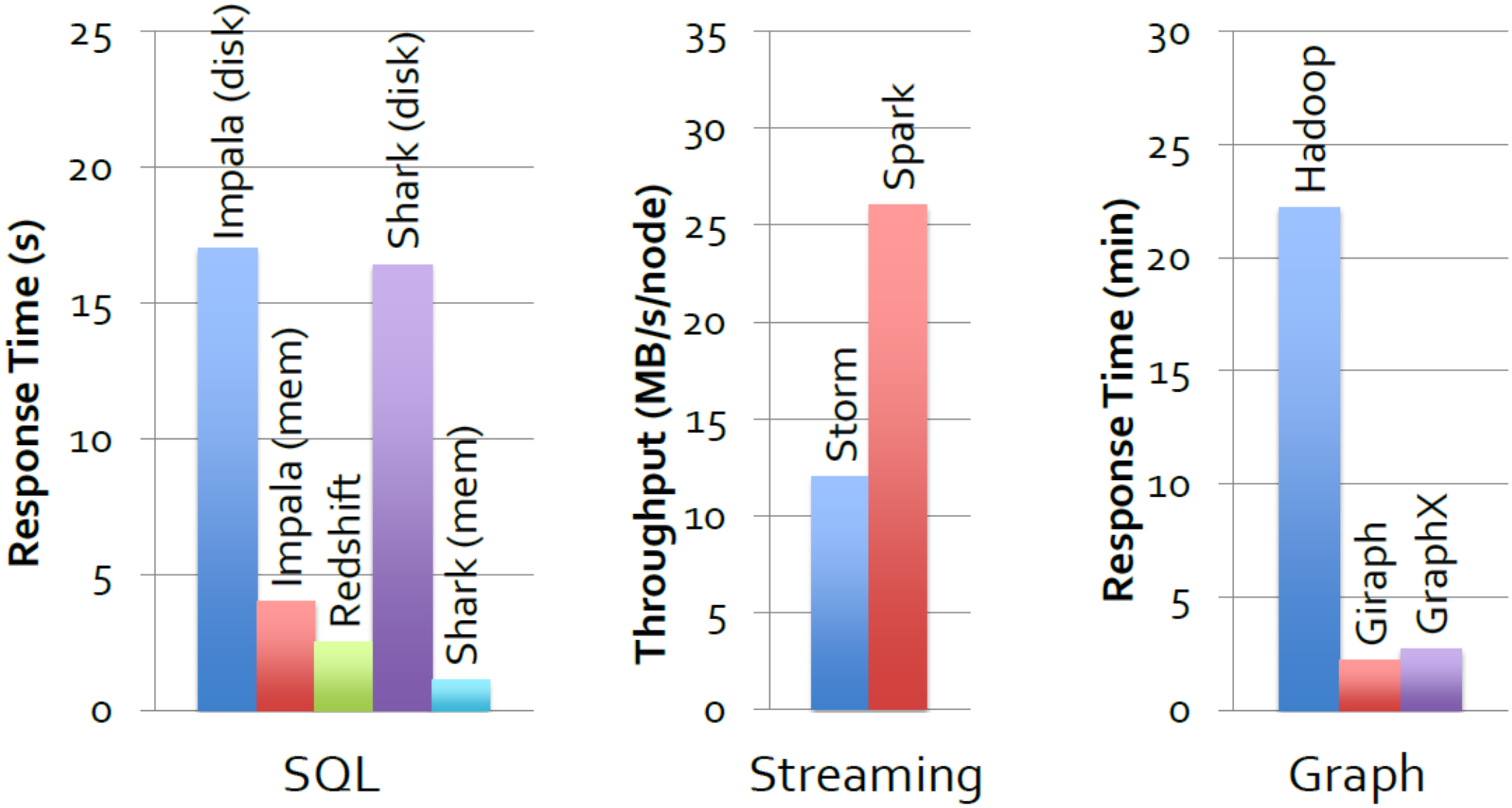
```
sc.textFile('...').flatMap(lambda x: x.split())
    .map(lambda x: (x, 1)).reduceByKey(lambda x, y: x+y)
    .saveAsTextFile('...')
```

VS

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Ref: MapReduce WordCount Tutorial

The Power of Unified BDA Engine: Performance



I. Course Methodology & Logistics Overview

II. Spark: Background & Position in Big Data Analytics

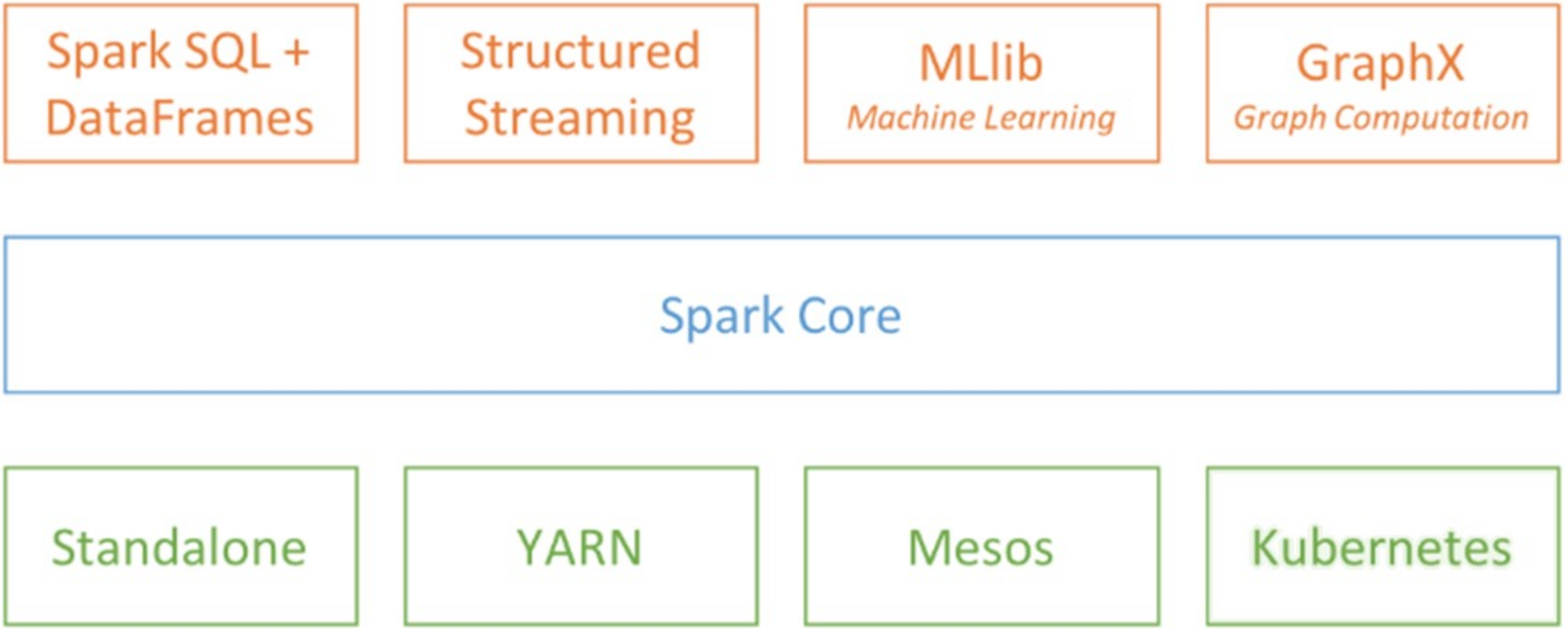
III. Core Concepts & Challenges of Distributed Computing

IV. Overview of Spark Components

V. Conceptual Introduction to Spark Application

VI. Appendix

Apache Spark Components



Spark Core

Spark Core contains the basic & core functionalities of Spark Applications like:

- ✓ Task Scheduling
- ✓ Memory Management
- ✓ Fault Recovery
- ✓ Interaction with Storage Systems
- ✓ and other low level capabilities which support higher level Spark components like SQL, Streaming, Machine Learning and Graph processing.

Spark Core is the main component to process unstructured data (with exceptions such as in case of Image Data).

Spark Core API resides Resilient Distributed Datasets (RDDs).

Spark Core API provides different methods for building and manipulating RDDs.

Spark SQL (1/2)

- Spark SQL is the main component to process structured & semi-structured data.

- It allows querying data via SQL as well as Hive Query Language (HQL). Spark SQL supports ANSI SQL 2003 Standard.

- Spark Dataframes are built-in in this component and they officially supported since version Spark 1.4. They are available for all programming languages.

- Spark Datasets (Type-safe dataframes) are also built-in in this component and they are officially supported in version Spark 2.0. They are available for only Scala & Java as they are type-safe programming languages natively.

- It supports many data sources, including Hive tables, external databases, Parquet, Avro, CSV, XML and JSON file formats and many more

Spark SQL (2/2)

- **Spark SQL** is built-on **Spark Core** but Spark SQL is the basis of the optimizations and main API for user friendly Spark Components like Structured Streaming, *Dataframe based Spark MLlib* or or third-party projects like *Graphframes*, *Tensorframes* etc.

- Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the functions and standard of data science libraries supported by RDDs in Python, Java, and Scala, all within a single application. Thus it is possible to intermix Spark SQL with streaming data, complex analytics, functional & sometimes even object oriented programming.

- This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike any other open source data warehouse tool.

Spark Streaming (Streaming & Structured Streaming)

- Spark Streaming enables processing of live streams of data (for example log files from a production web server, video steaming, sensor data or tweets from Twitter).
- There are two streaming libraries in Apache Spark:
  - ✓ *Spark Structured Streaming* (Dataframe Based Streaming API with mini-batch or record-at-time options extending Spark SQL API)
  - ✓ *Spark Streaming* (RDD Based mini-batch Streaming API extending Spark Core API
- Underneath its API, Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability as Spark Core.
- If Spark Structured Streaming is used, then the performance, storage and usability efficiencies inherited from Spark SQL will be added.
- It can be integrated with other stream processing tools like Apache Kafka, Apache Flume, Apache NIFI, AWS Kinesis, Google Pub/Sub etc.

Spark Machine Learning (MLlib)

- Spark comes with a component containing common machine learning and data analysis functionality, via model training via data & task (model) parallelism.
- There are two built-in libraries in Apache Spark MLlib :
  - ✓ *Dataframe based API is* based on Spark SQL API
  - ✓ *RDD based API is* based on Spark Core API

❖Spark MLlib API involves several machine learning algorithms and statistical methods which are designed to scale out across a cluster.

❖Spark MLlib include variety of options for feature extraction & engineering, classification, regression, clustering, dimensionality reduction and collaborative filtering, as well as supporting functionality such as model evaluation and data import.

❖It offers model parallelism in model tuning as well as data parallelism for training and prediction.

❖Spark MLlib also provides some lower-level ML primitives for commonly used Optimization Algorithms

Spark Graph Processing (GraphX & GraphFrames)

- Spark Graph Processing is used for manipulating graphs and performing graph-parallel computations. (e.g., a social network's friend graph, metro network in Madrid, or any dataset which is efficient or useful to be represented as graphs).
- There are two graph processing libraries in Apache Spark:
  - ✓ *GraphFrames* (Dataframe based API extending Spark SQL)
  - ✓ *GraphX* (RDD based API extending Spark Core API)
- Spark Graph Processing allows us to create directed & undirected graphs with arbitrary *properties* attached to each vertex and edge.
- It also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting, shortest path calculation).

I. Course Methodology & Logistics Overview

II. Spark: Background & Position in Big Data Analytics

III. Core Concepts & Challenges of Distributed Computing

IV. Overview of Spark Components

V. Conceptual Introduction to Spark Application

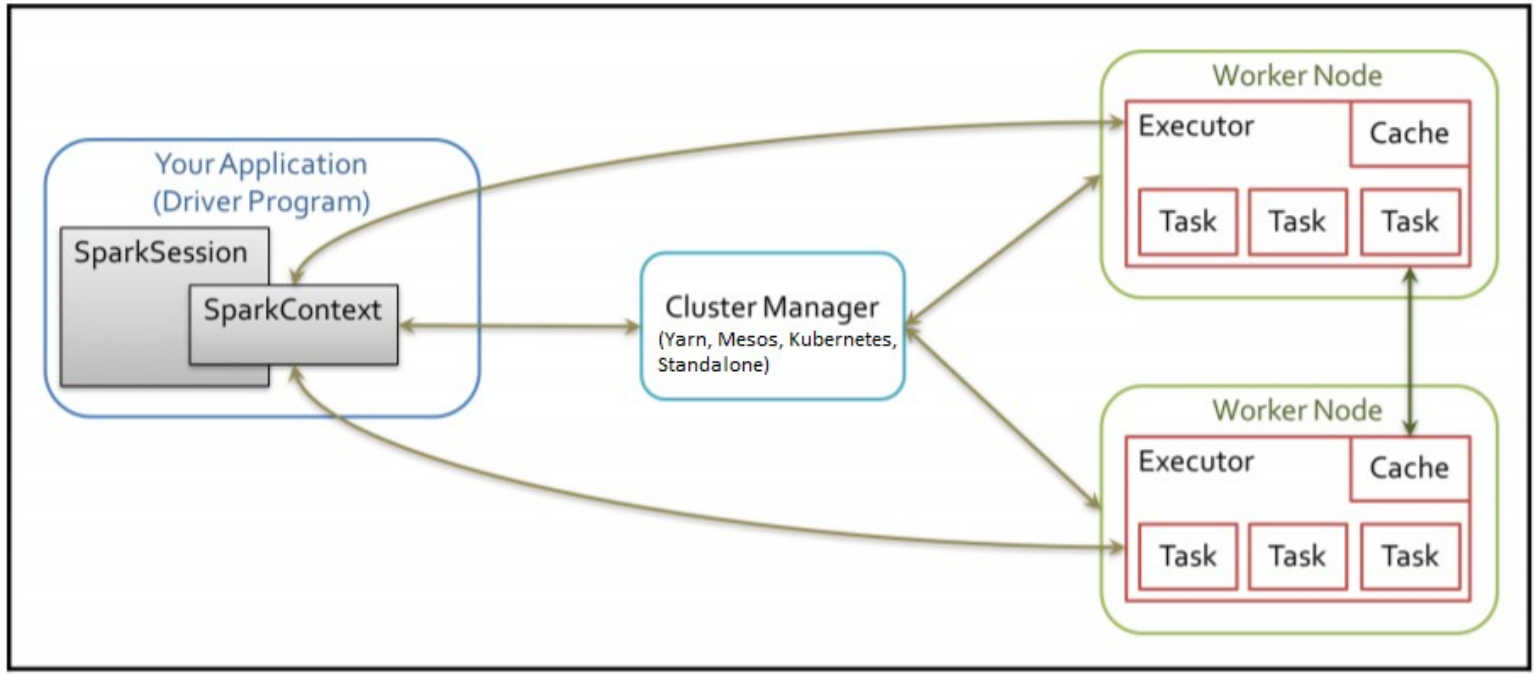
VI. Appendix

Spark Application – Basic Program Execution

Every Spark Application consists of a *Driver Program* with a *Spark Session* that launches various parallel data processing operations on a cluster.

Driver programs access Spark through a ***Spark Context*** object, which represents a connection to a computing cluster.

Driver program takes the program code and hands them over to a ***Cluster Manager***.



The ***Cluster Manager***, in turn, launches OS processes called ***Executors*** in multiple worker nodes, each having a set of tasks.

When **Executors** are launched, they are directly managed by the **Driver Program**.

Executors run the ***Tasks*** over each block/partition of RDDs.

Resilient Distributed Datasets (RDDs) (1/2)

Resilient Distributed Dataset (aka **RDD**) is the primary data abstraction in Apache Spark and the core of Spark (that I often refer to as "Spark Core").

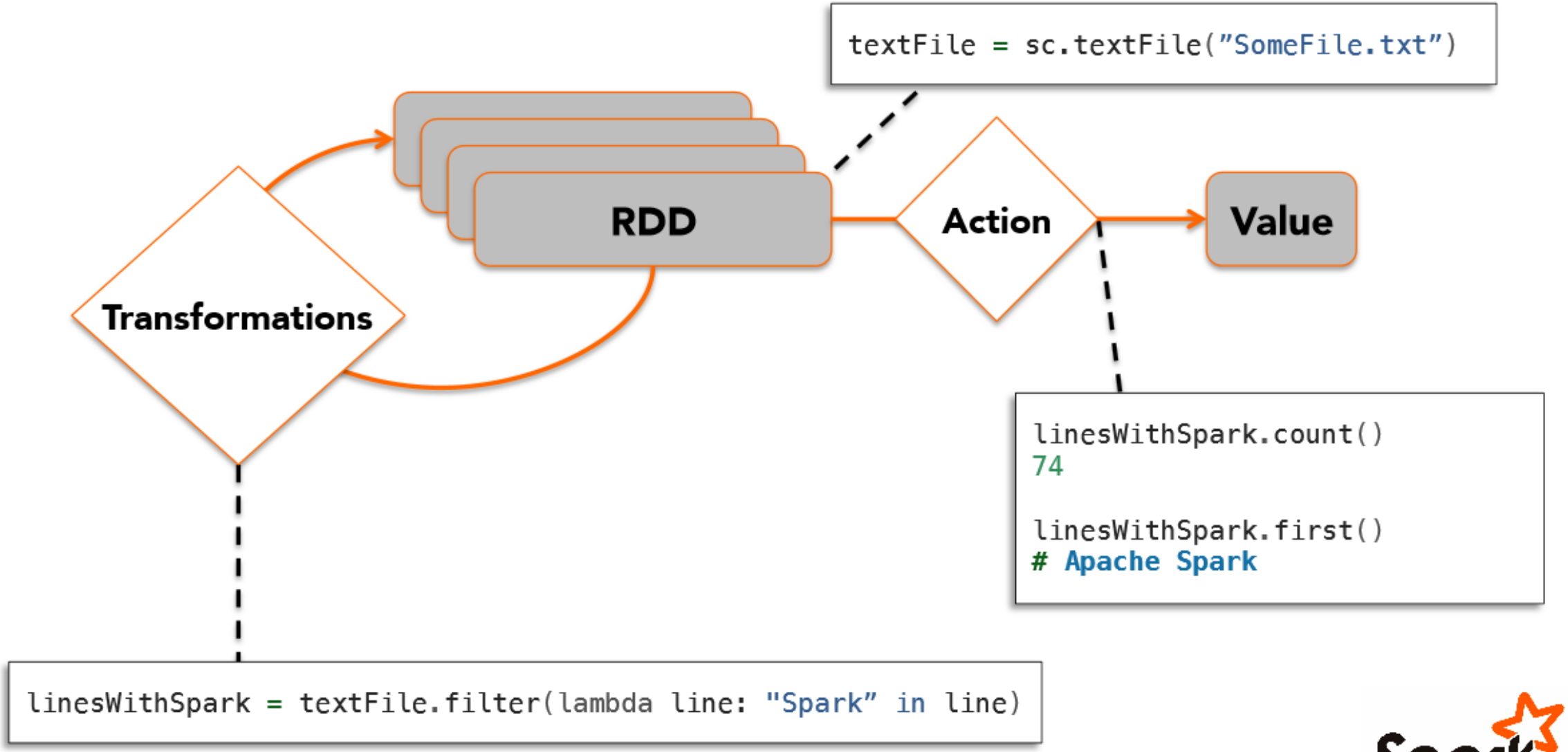
A RDD is a resilient and distributed collection of records spread over one or many partitions.

The features of RDDs (decomposing the name):

- ✓ **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph and so able to re-compute missing or damaged partitions due to node failures.
- ✓ **Distributed** with data residing on multiple nodes in a cluster.
- ✓ **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with).

Paper: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Resilient Distributed Datasets (RDDs) (2/2)



Transformations versus Actions

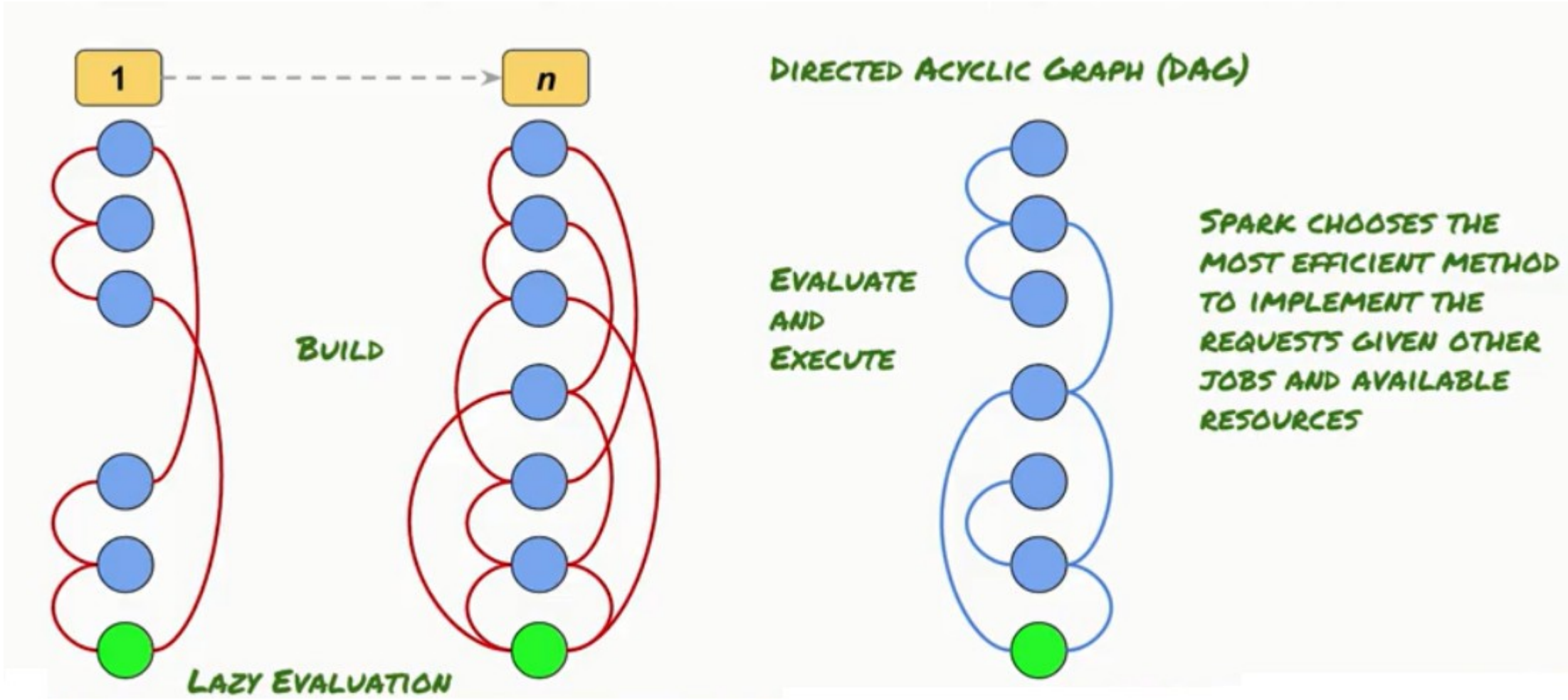
Transformations are **lazily evaluated** operations,

they transform one or more input RDD to an output RDD

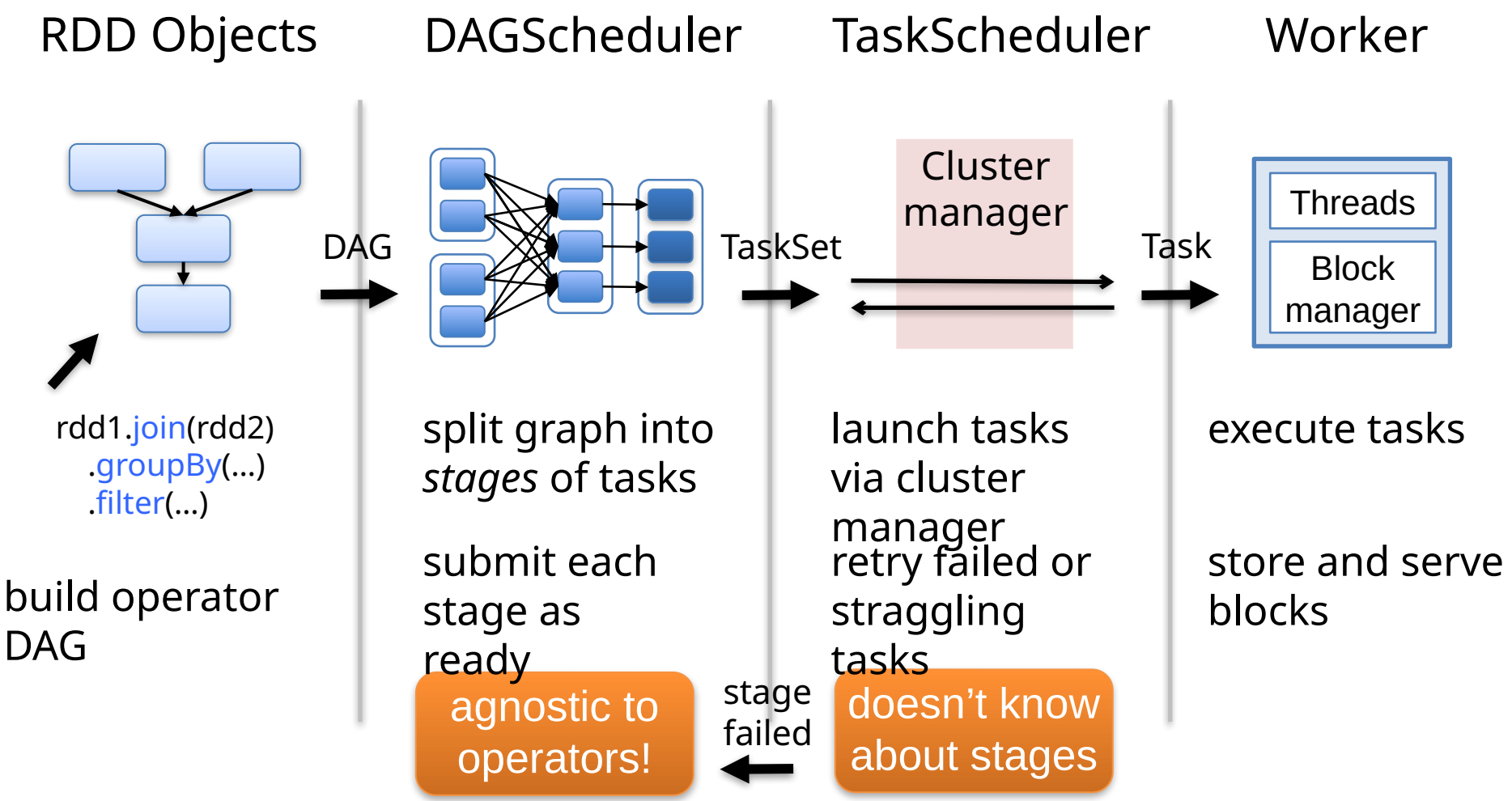
Actions are **eagerly executed** operations which initiate the real computation on a input RDD

to produce a Scalar or Collection Output

Lazy Evaluation and DAG



Spark Job Scheduling Process



Generality of Spark RDDs

- Spark RDD model is generic & flexible enough to implement the tasks suitables for several previous Big Data Programming Models in the same Spark code base:
  - ✓ MapReduce
  - ✓ Dryad
  - ✓ Pregel
  - ✓ Iterative MapReduce
  - ✓ GraphLab
  - ✓ SQL (Shark)
- Allows apps to *intermix* these models

Spark & Cluster Managers

Running Spark On Yarn

cloudera

MAPR

Cloud Dataproc

Running Spark on Mesos

MESOS

Running Spark On Kubernetes

kubernetes

Amazon EKS

Spark on Serverless Platforms

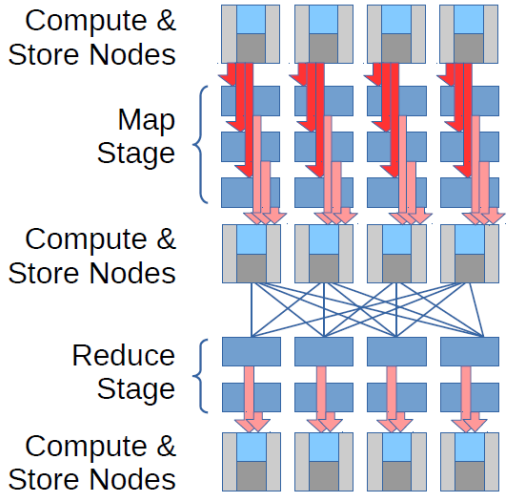
Serverless Computing:

- ✓ No need to setup or manage a server cluster
- ✓ Automatic, dynamic and fine-grained scaling of resource pool according to processing needs
- ✓ Better capacity utilization and billing based on sub-second usage in the cloud platforms
- ✓ Sparkon a Serverless Platform is slower than on premise Spark Setups in terms of response time. This is due to the challenges of Serverless Computing:

- *Sub-Optimal Scheduling*,
- *Container Start-up Time*,
- *Remote Storage of Input Data*,
- *Overhead of Intermediate Data Storage* (shuffled, broadcasted)

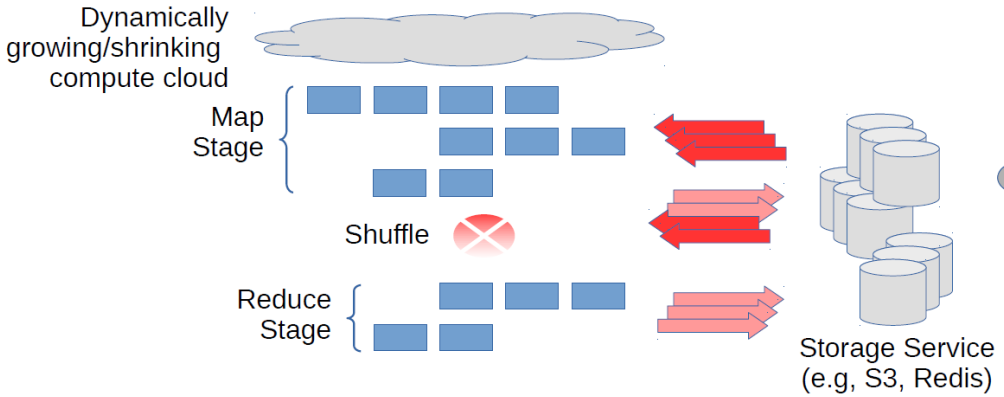
MapReduce on a Server Cluster or Serverless Platforms

Example: MapReduce (Cluster)



data is mostly written and read locally

Example: MapReduce (Serverless)



data is exclusively written and read remotely

I. Course Methodology & Logistics Overview
II. Spark: Background & Position in Big Data Analytics
III. Core Concepts & Challenges of Distributed Computing
IV. Overview of Spark Components
V. Conceptual Introduction to Spark Application
VI. Appendix

Easy Spark Installation with Python or R Package Managers

Spark itself is written in Scala, and runs on the *Java Virtual Machine (JVM)* and therefore to run Spark either on your laptop or a cluster, all you need is an installation of Java 1.8 or newer.

If you wish to use the Python API you will also need a Python interpreter (Python 3.5.x or newer).

Python Package Index:

https://pypi.org/project/pyspark/2.3.1/ PPI is available only for versions Spark 2.2+

R Cran Libraries:
If you wish to use R you'll also need a version of R on your machine. You have two library options:

1- sparklyr

2- SparkR

Manual Installation of Standalone Spark Application Framework

You can install Apache Spark 2.3.1 (latest stable version) to your local environment:

https://spark.apache.org/releases/spark-release-2-3-1.html

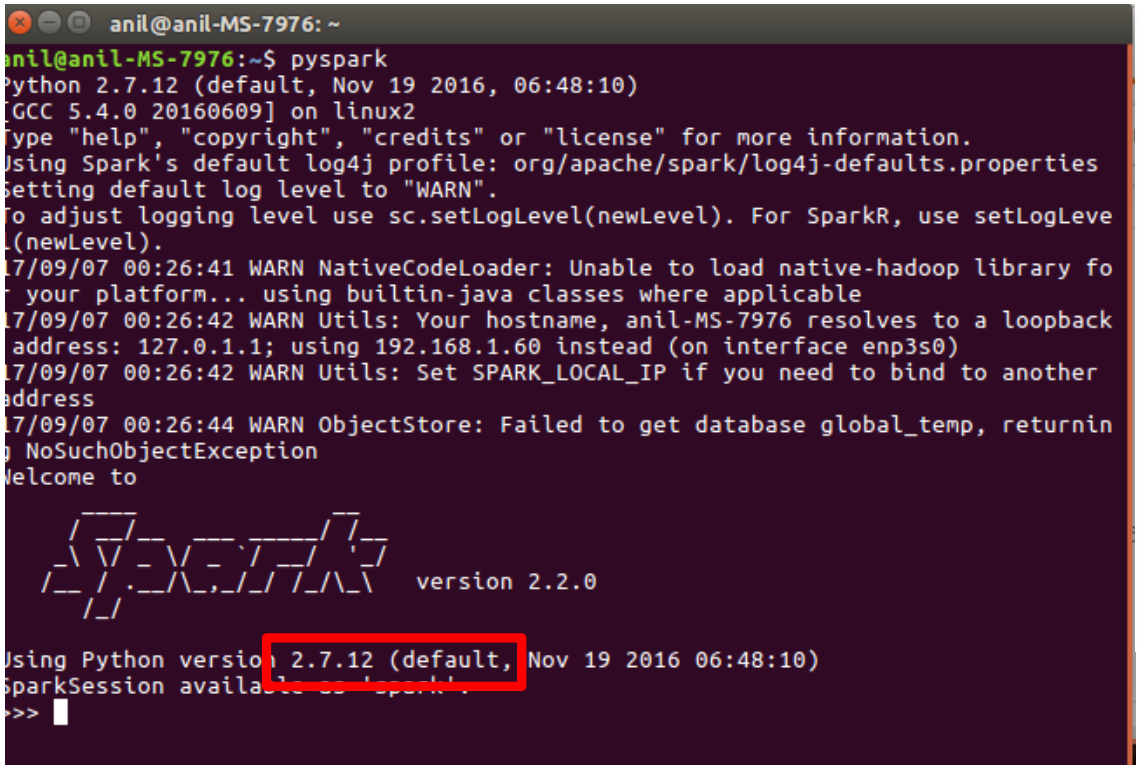
Important Notes:

- You should have installed Python 3.6.x
- You should have installed Java 1.8+
- It is also recommended to install Scala 2.11.x (Not scala 2.10.x or 2.12.x)

Windows Installation might be a bit tricky:

https://hernandezpaul.wordpress.com/2016/01/24/apache-spark-installation-on-windows-10/

Standalone Spark Installation: Two potential problems, One Solution



Solution: https://mengdong.github.io/2016/08/08/fully-armed-pyspark-with-ipython-and-jupyter/

Standalone Spark Installation: To Login spark-shell or pyspark (shell)

sudo vi ~/.bashrc

After lines below…

#Spark Configurations

export SPARK_HOME=/opt/spark-2.3.1-bin-hadoop2.7
export PATH=$PATH:$SPARK_HOME/bin
#Pyspark Configurations
export PYSPARK_PYTHON=/usr/bin/python3
export PYSPARK_DRIVER_PYTHON=notebook

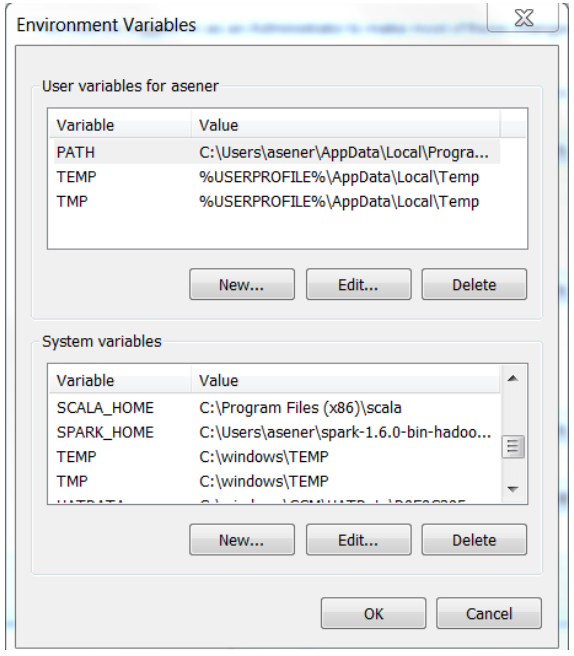
After saving…

source ~/.bashrc

Navigate to Computer System Properties
 Advanced System Settings

Advanced Tab Environmental variables

You need all 4 variables on the left as in Linux & MacOS



Shark (Hive on Spark Execution Engine)

- ✓ Shark was an older Hive-on-Spark project out of the University of California, Berkeley.
- ✓ Shark modified Apache Hive to run on Spark Execution Engine, previously it was supporting MapReduce and Apache Tez Execution Engines.
- ✓ It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs. **Therefore, it is recommended to use Spark SQL directly, instead of Shark.**

Shark versus other Big Data SQL Tools Benchmarking

System	SQL variant	Execution engine	UDF Support	Mid-query fault tolerance	Open source	Commercial support	HDFS Compatible
Hive	Hive QL (HQL)	MapReduce	Yes	Yes	Yes	Yes	Yes
Tez	Hive QL (HQL)	Tez	Yes	Yes	Yes	Yes	Yes
Shark	Hive QL (HQL)	Spark	Yes	Yes	Yes	Yes	Yes
Impala	Some HQL + some extensions	DBMS	Yes (Java/C++)	No	Yes	Yes	Yes
Redshift	Full SQL 92 (?)	DBMS	No	No	No	Yes	No

Ref: https://amplab.cs.berkeley.edu/benchmark/#

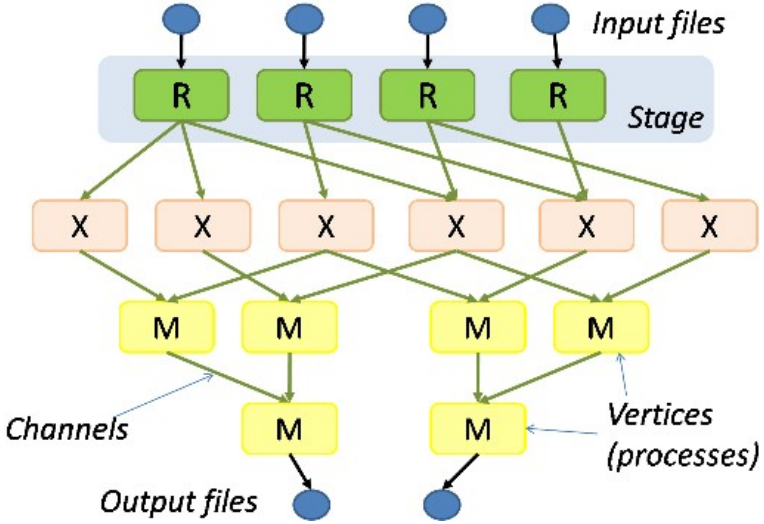
Spark RDDs versus a standard Distributed Shared Memory Implementation

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

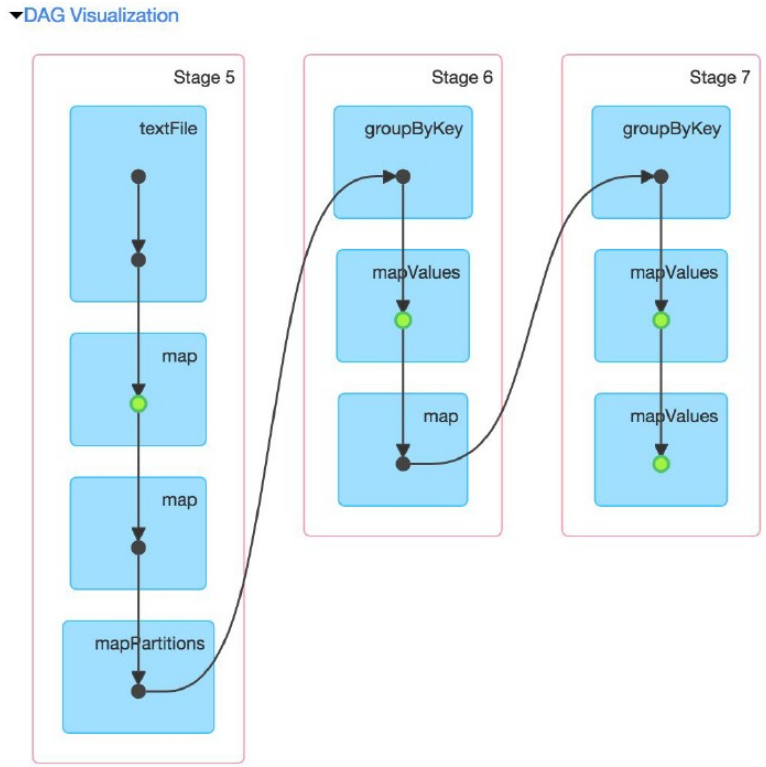
Ref: Matei Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012.

Directed Acyclic Graph (DAG) Model

Dryad and DryadLINQ (2007)
Michael Isard
et al. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks,
EuroSys, 2007

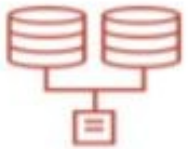


Spark (2010)
Matei Zahara
et al. Spark: Cluster Computing with Working Sets, 2010
Matei Zaharia
et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012.



What is new on Apache Spark 2.3?

Continuous Processing



Data Source API V2

Spark on Kubernetes

PySpark Performance

ML on Streaming

History Server V2

Stream-stream Join

UDF Enhancements

Image Reader

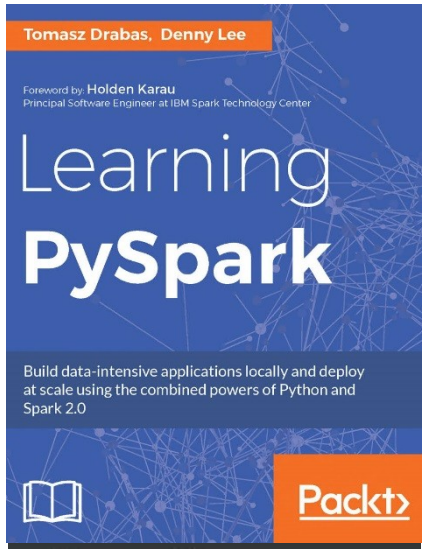
Native ORC Support

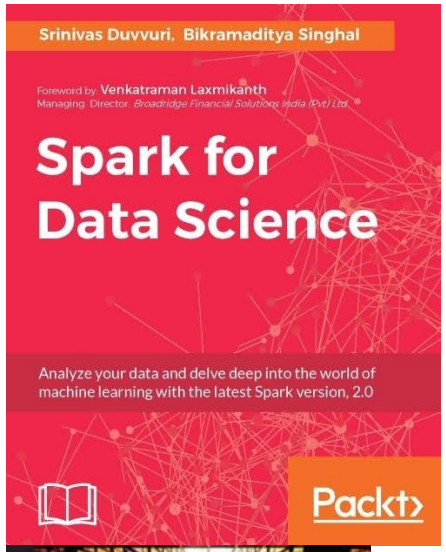
Stable Codegen

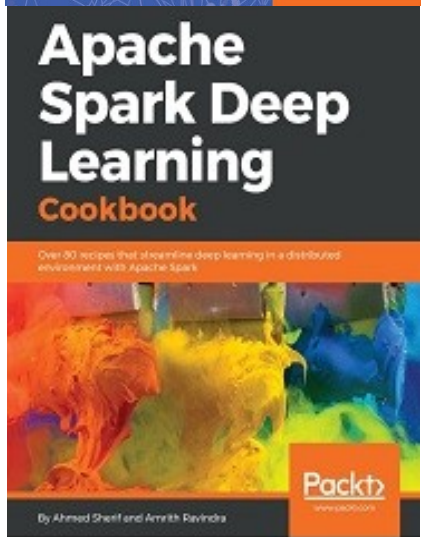
Various SQL Features

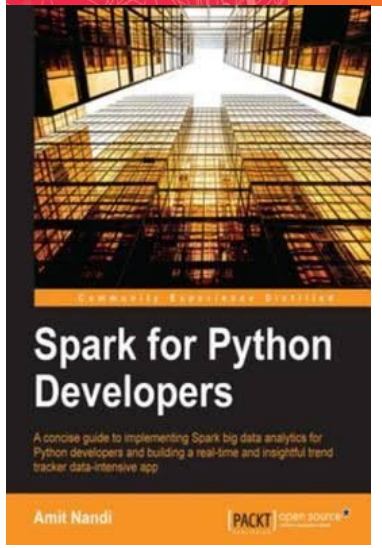
Reference Apache Spark Book Recommendations

EVERYTHING IS IN PYTHON

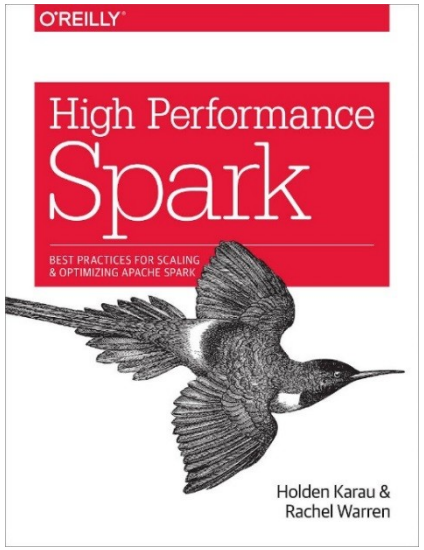


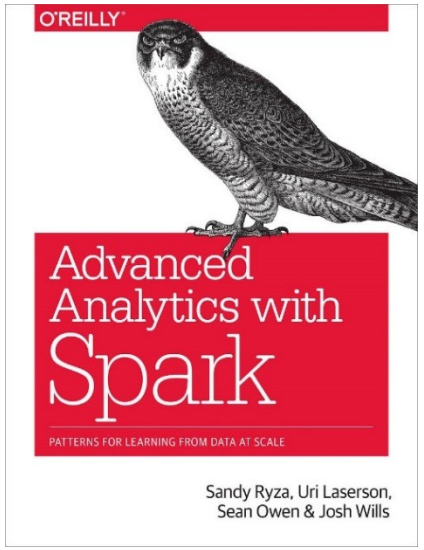


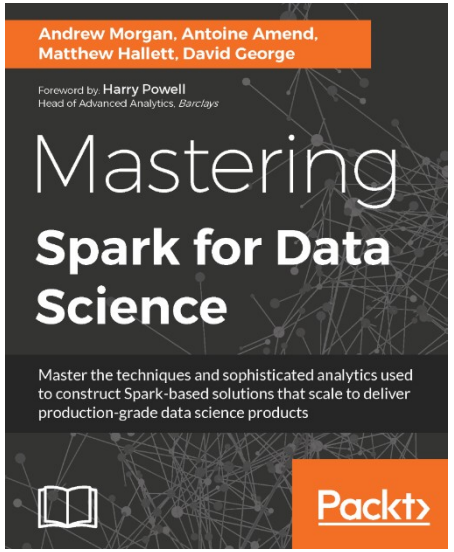




MOSTLY IN SCALA, EXTENSIVE REFERENCES



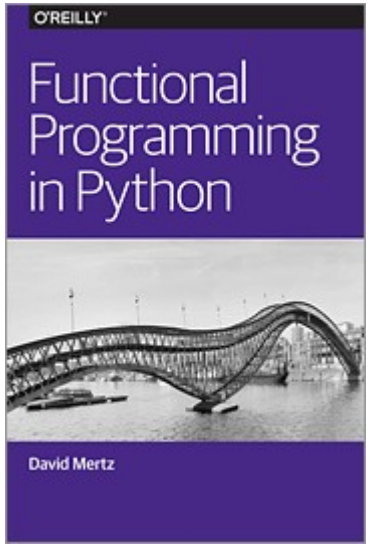


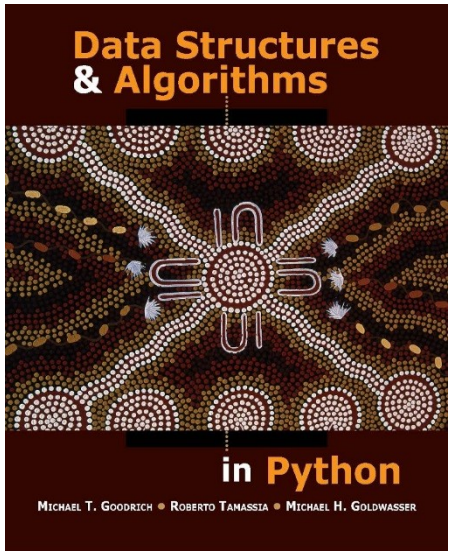


THIS ONE IS AVAILABLE IN IE CAMPUS

Python Book Recommendations

PYTHON IS NOT FULLY FUNCTIONAL, THIS BOOK IS IN OUR ONLINE CAMPUS





ONE OF THE BEST BOOKS EVER WRITTEN IN PYTHON, GREAT FOR INTERVIEWS

THIS BOOK IS BASED ON MIT'S OPENCOURSEWARE, MORE RELEVANT TO DATA SCIENCE