

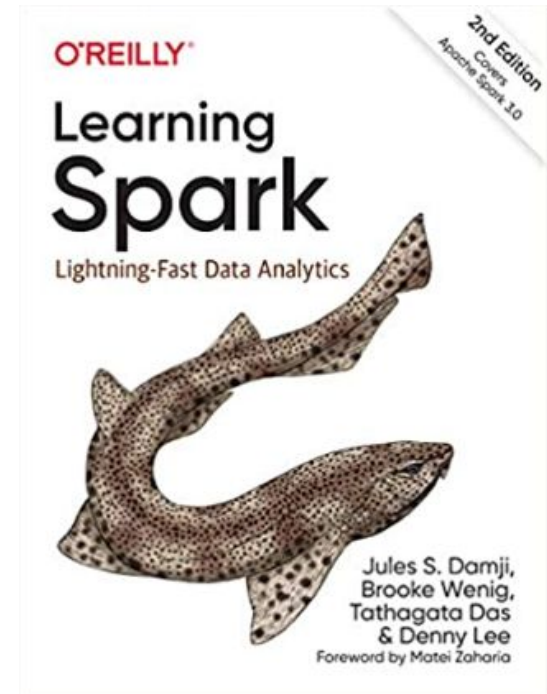
# UMD DATA605 - Big Data Systems (Apache) Spark

Dr. GP Saggese  
[gsaggese@umd.edu](mailto:gsaggese@umd.edu)

with thanks to Alan Sussman, Amol Deshpande,  
David Wheeler (GMU), T. Yang (UCSB)  
and Apache documentation

# Apache Spark - Resources

- Concepts in the slides
- Academic paper
  - [“Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”](#), 2012
- Web resources
  - [Spark programming guide](#)
  - [Coursera Spark in Python tutorial](#)
- Mastery
  - “Learning Spark: Lightning-Fast Data Analytics” (2nd Edition)
  - Not my favorite, but free [here](#)



# Hadoop MapReduce: Shortcomings

- **Hadoop is hard to administer**

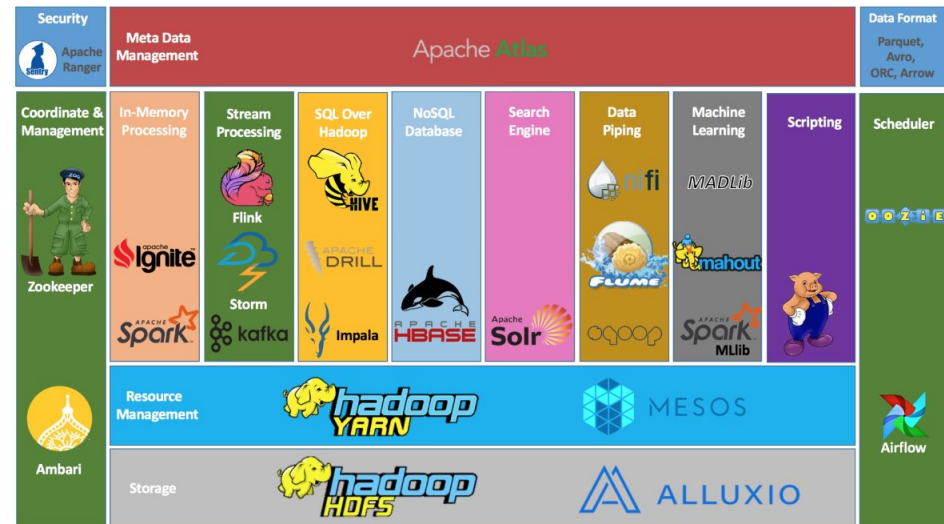
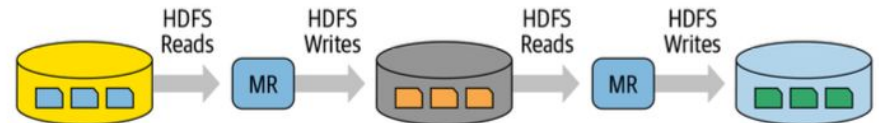
- Lots of layers (HDFS, Yarn, Hadoop, ...)
- Lots of configuration

- **Hadoop is hard to use**

- API is verbose (example later)
- Not great binding for multiple languages (e.g., Java is native)
- MapReduce jobs interact by writing data on disk

- **Large but fragmented ecosystem**

- No native support in Hadoop for
  - Machine learning
  - SQL, streaming
  - Interactive computing
  - ...
- To handle new workloads new systems developed on top of Hadoop
- E.g., Apache Hive, Storm, Impala, Giraph, Drill



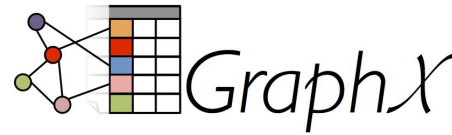
# (Apache) Spark

- **Open-source**
  - DataBrick monetizes it (\$40B startup)
- **General processing engine**
  - Large set of operations instead of just `Map()` and `Reduce()`
  - Operations can be arbitrarily combined in any order
  - Transformations vs Actions
  - Computation is organized as a DAG
  - DAGs are decomposed into tasks that can run in parallel
  - Scheduler / optimizer on parallel workers
- **Supports several languages**
  - Java, Scala (preferred)
  - Python good support through bindings, but not the main language
- **Data abstraction**
  - Resilient Distributed Dataset (RDD)
  - Other data structures (e.g., DataFrames, Datasets) built on top of RDDs
- **Fault tolerance through RDD lineage**
- **In-memory computation**
  - All intermediate results are kept in memory instead of disk
  - Persist data on disk or in memory, if needed -> speed
  - Initial winning advantage



# Berkeley: From Research to Companies

- [Amplab](#)
  - [Projects](#)



- [Rise lab](#)
  - [Projects](#)

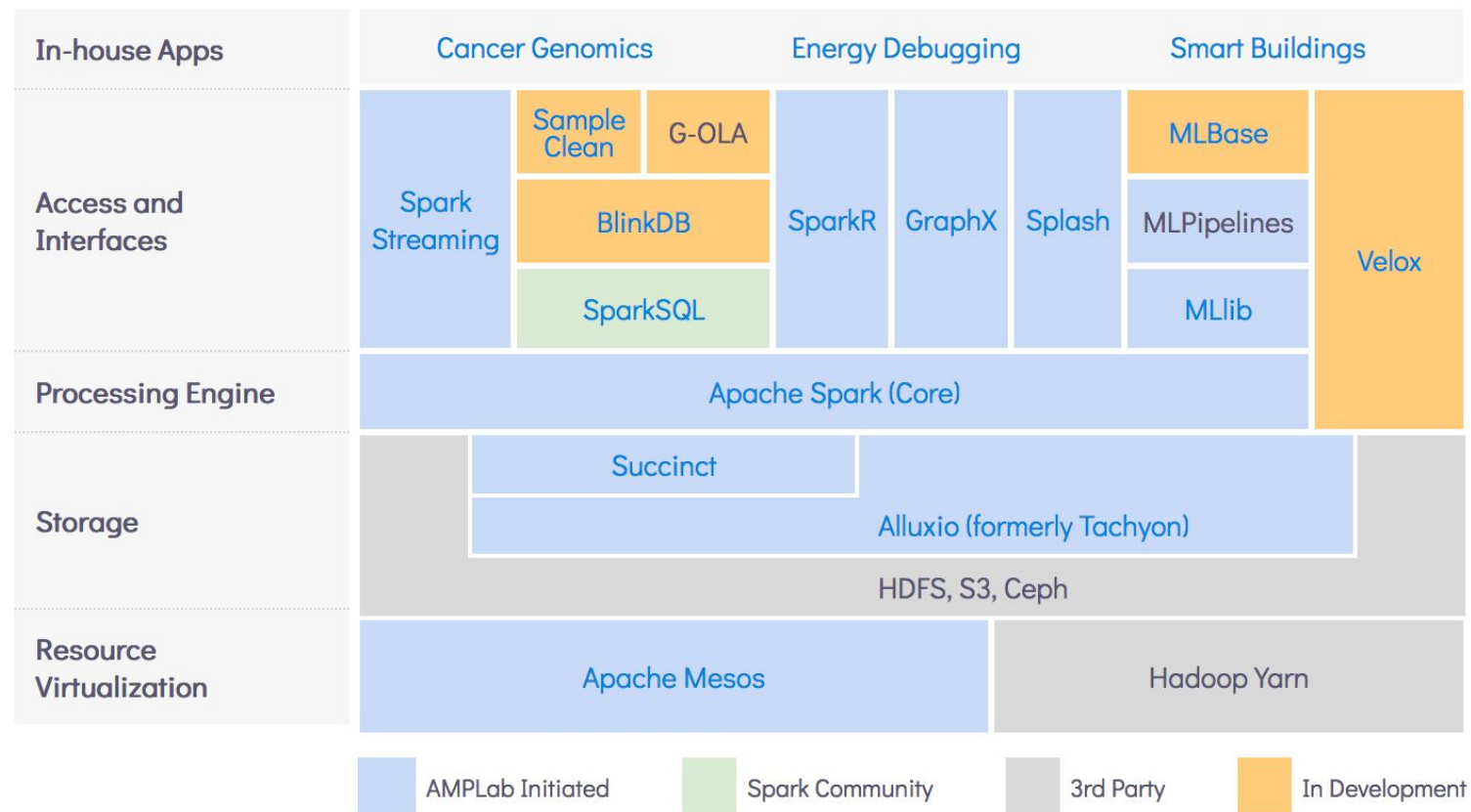


- [DataBricks](#)
  - Private company worth \$40B
  - [Accidental Billionaires: How Seven Academics Who Didn't Want To Make A Cent Are Now Worth Billions](#), 2023



# Berkeley AMPLab Data Analytics Stack

- So many tools that they have their own big data stack!



# Apache Spark

- **Unified stack**

- Different computation models in a single framework

- **Spark SQL**

- ANSI SQL compliant
- Work with structured relational data

- **Spark MLlib**

- Build ML pipelines
- Support popular ML algorithms
- Built on top of Spark DataFrame

- **Spark Streaming**

- Handle continually growing tables
- Tables are treated as static table

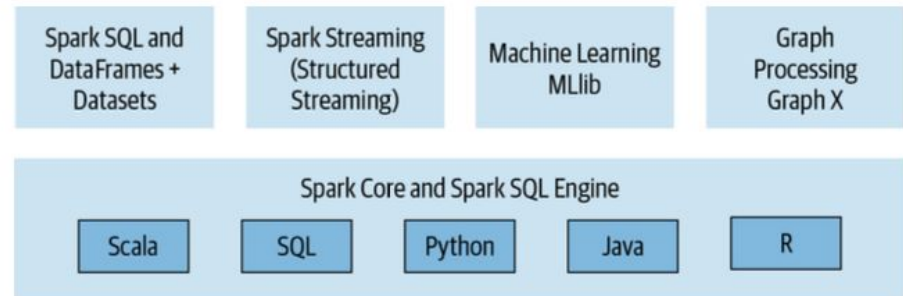
- **GraphX**

- Manipulate graphs
- Perform graph-parallel computation

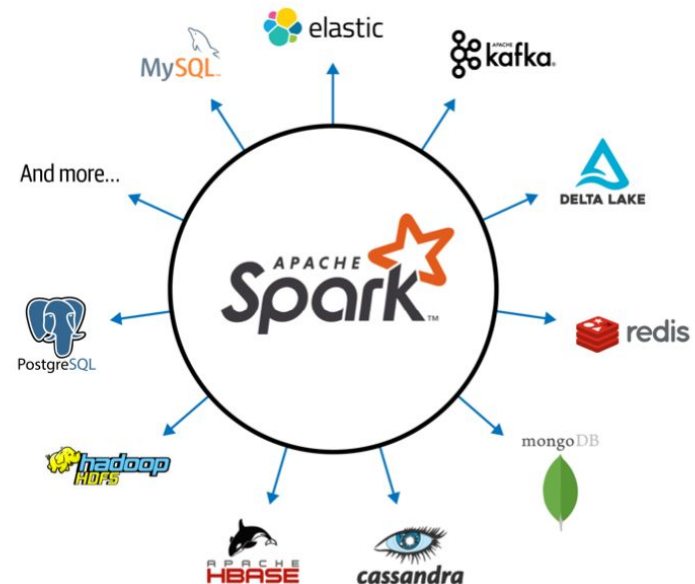
- **Extensibility**

- Read from a many sources
- Write to many backends

## General purpose applications



## One computation engine



# Resilient Distributed Dataset (RDD)

- **A Resilient Distributed Dataset (RDD)**

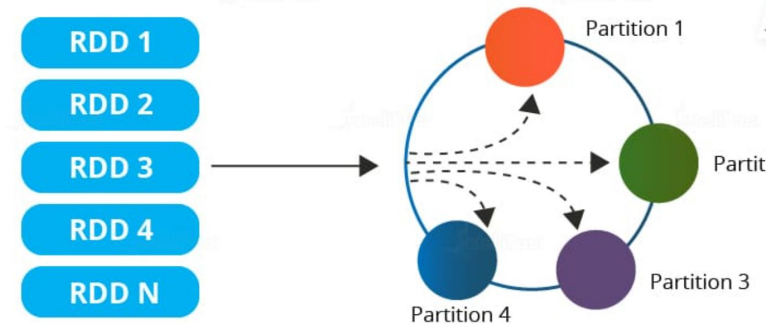
- Collection of data elements
- Partitioned across nodes
- Can be operated on in parallel
- Fault-tolerant
- In-memory / serializable

- **Applications**

- Best suited for applications that apply the same operation to all elements of a dataset (vectorized)
- Less suitable for applications that make asynchronous fine-grained updates to shared state
  - E.g., updating one value in a dataframe

- **Ways to create RDDs**

- *Reference* data in an external storage system
  - E.g., a file-system, HDFS, HBase
- *Parallelize* an existing collection in your driver program
- *Transform* RDDs into other RDDs





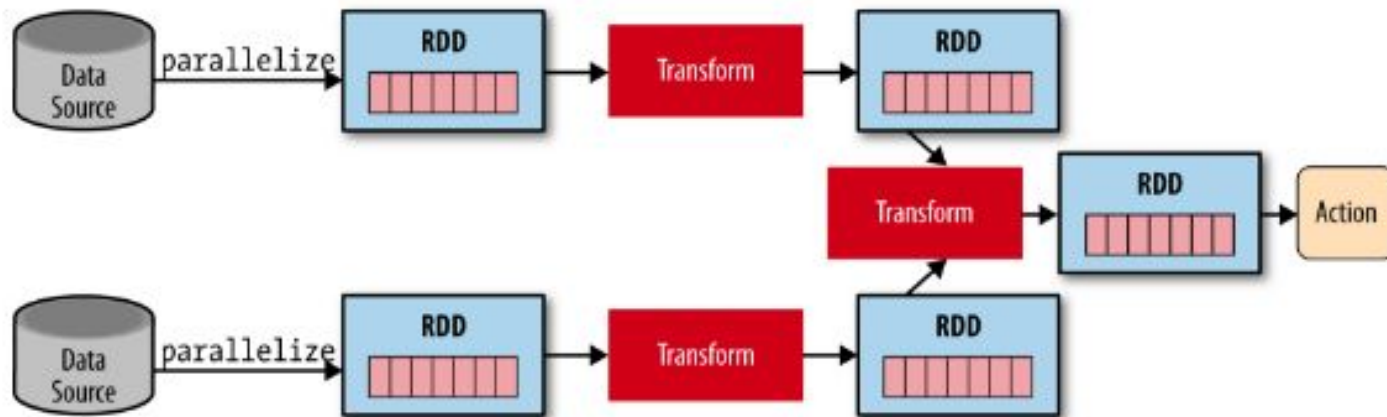
# Transformation vs Actions

- **Transformations**

- Lazy evaluation
- Nothing computed until an Action requires it
- Build a graph of transformations

- **Actions**

- When applied to RDDs force calculations and return values
- Aka Materialize



# Spark Example: Estimate Pi

- Estimate Pi with MapReduce in Spark

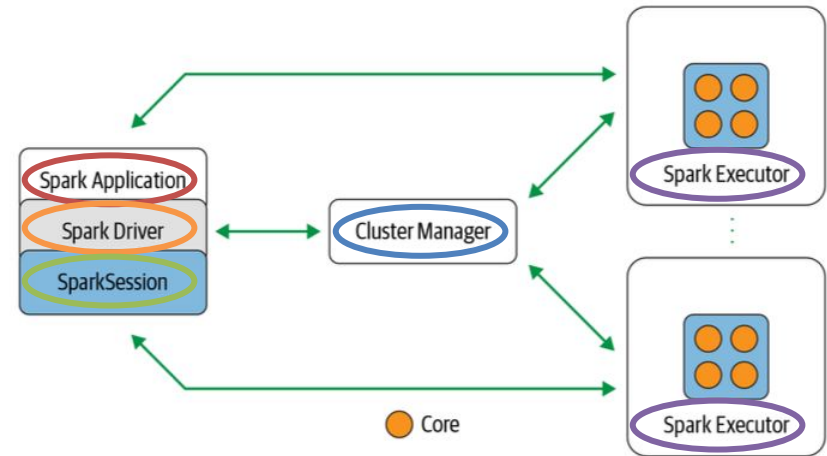
```
# Estimate  $\pi$  (compute-intensive task).  
# Pick random points in the unit square [(0,0)-(1,1)].  
# See how many fall in the unit circle center=(0, 0), radius=1.  
# The fraction should be  $\pi / 4$ .  
  
import random  
random.seed(314)  
  
def sample(p):  
    x, y = random.random(), random.random()  
    in_unit_circle = 1 if x*x + y*y < 1 else 0  
    return in_unit_circle  
  
# "parallelize" method creates an RDD.  
NUM_SAMPLES = int(1e6)  
count = sc.parallelize(range(0, NUM_SAMPLES)) \  
    .map(sample) \  
    .reduce(lambda a, b: a + b)  
approx_pi = 4.0 * count / NUM_SAMPLES  
print("pi is roughly %f" % approx_pi)
```

executed in 386ms, finished 04:27:53 2022-11-23

pi is roughly 3.141400

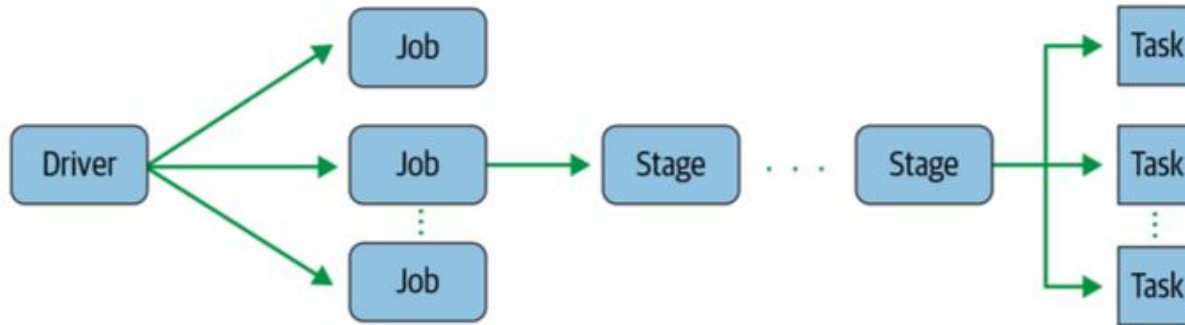
# Spark: Architecture

- **Architecture** = who does what, what are the responsibilities of each piece
- **Spark Application**
  - Code that the user writes to describe the computation
  - E.g., Python code calling into Spark
- **Spark Driver**
  - Instantiate a *SparkSession*
  - Communicate with *Cluster Manager* to request resources
  - Transform operations into DAG computations
  - Distribute execution of tasks across *Executors*
- **Spark Session**
  - Represent the interface to Spark system
- **Cluster Manager**
  - Manage and allocate resources
  - Support Hadoop, YARN, Mesos, Kubernetes
- **Spark Executor**
  - Run a worker node to execute tasks
  - Typically one executor per node
  - JVM



# Spark: Computation Model

- **Architecture** = who does what, what are the responsibilities of each piece
- **Computational model** = how are things done



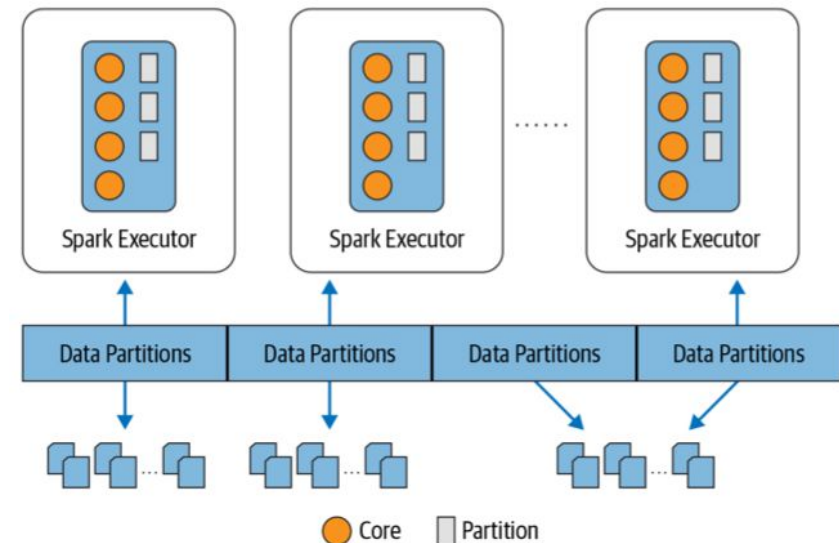
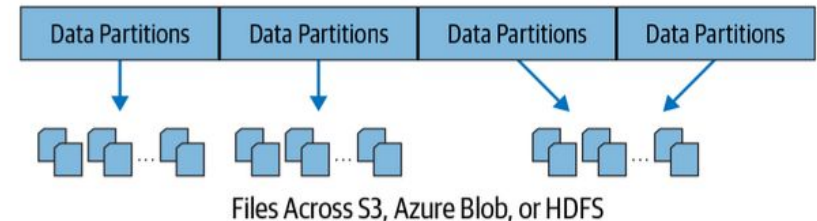
- **Spark Driver**
  - The driver converts the Spark application into one or more Spark *Jobs*
  - Computation is described by *Transformations* and triggered by *Actions*
- **Spark Job**
  - A parallel computation that runs in response to a Spark *Action*
    - E.g., `save()`, `collect()`
  - Each *Job* is a DAG containing one or more *Stages* depending on each other
- **Spark Stage**
  - Each *Job* is a smaller operation
  - *Stages* can be performed serially or in parallel
- **Spark Task**
  - Each *Stage* is comprised of multiple *Tasks*
  - A single unit of work sent to a *Spark Executor*
  - Each *Task* maps to a single core and works on a single partition of data

# Deployment Modes

- Spark can run on several different configurations
  - **Local**
    - E.g., run on your laptop
    - Driver, Cluster Manager, Executors all run in a single JVM on the same node
  - **Standalone**
    - Driver, Cluster Manager, Executors run in different JVMs on different nodes
  - **YARN**
  - **Kubernetes**
    - Driver, Cluster Manager, Executors run on different pods (i.e., containers)

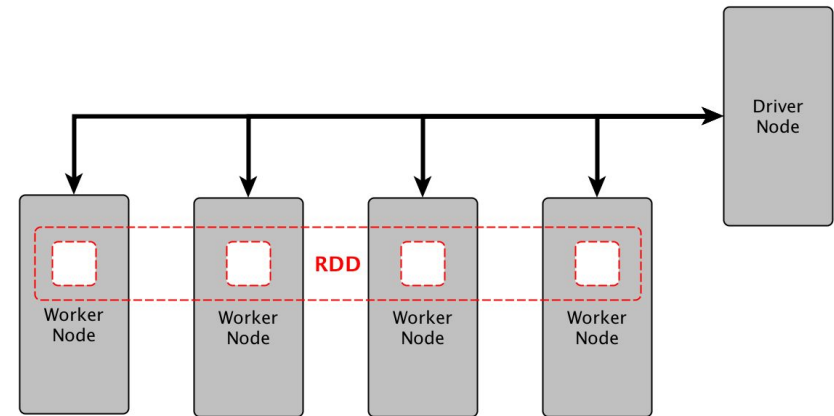
# Distributed Data and Partitions

- **Data is distributed as partitions across different physical nodes**
  - Each partition is typically stored in memory
  - Partitions allow efficient parallelism
- **Spark Executors process data that is “close” to them**
  - Minimize network bandwidth
  - Data locality
  - Same approach as Hadoop



# Parallelized Collections

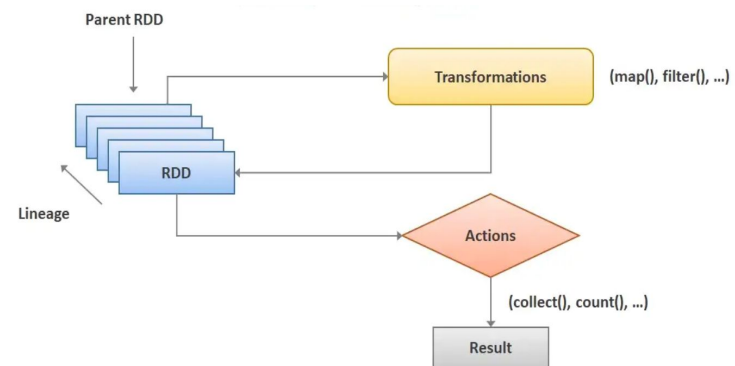
- Parallelized collections are created by calling *SparkContext* `parallelize()` method on an existing collection
- Data is spread across nodes
- Number of *partitions* to cut the dataset into
  - Spark will run one *Task* for each partition of the cluster
  - Typically you want 2-4 partitions for each CPU in your cluster
  - Spark tries to set the number of partitions automatically based on your cluster
  - You can also set it manually by passing it as a second parameter to `parallelize()`



# Transformations vs Actions

## • Transformations

- Transform a Spark RDD into a new RDD without modifying the input data
  - Immutability like in functional programming
  - E.g., `select()`, `filter()`, `join()`, `orderBy()`
- Transformations are evaluated lazily
  - Inspect computation and decide how to optimize it
  - E.g., joining, pipeline operations, breaking into stages
- Results are recorded as "lineage"
  - A sequence of stages that can be rearranged, optimized without changing results



## • Actions

- An action triggers the evaluation of a computation
  - E.g., `show()`, `take()`, `count()`, `collect()`, `save()`



# Spark Example: MapReduce in 1 (or 4) Line

- MapReduce in 4 lines

```
!more data.txt
```

executed in 1.77s, finished 04:37:35 2022-11-23

One a penny, two a penny, hot cross buns

```
lines = sc.textFile("data.txt").flatMap(lambda line: line.split(" "))
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
result = counts.collect()
print(result)
```

executed in 428ms, finished 04:36:24 2022-11-23

```
[('One', 1), ('two', 1), ('hot', 1), ('cross', 1), ('a', 2), ('penny', 2), ('buns', 1)]
```

- MapReduce in 1 line (show-off version)

```
result = sc.textFile("data.txt").flatMap(lambda line: line.split(" ")).map(
    lambda s: (s, 1)).reduceByKey(lambda a, b: a + b).collect()
print(result)
```

executed in 591ms, finished 05:06:00 2022-11-23

```
[('One', 1), ('two', 1), ('hot', 1), ('cross', 1), ('a', 2), ('penny', 2), ('buns', 1)]
```

# Same Code in Java Hadoop

```
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);
}
```

# Spark Example: Logistic Regression in MapReduce

```
# Logistic Regression
# - iterative machine learning algorithm
# - find best hyperplane that separates two sets
#   of points in a multi-dimensional feature space
#
# Apply MapReduce operation repeatedly to the same
# dataset, so it benefits greatly from caching the
# input in RAM.

points = spark.textFile(...).map(parsePoint).cache()
# Initial separating plane.
w = numpy.random.rand(size = D)
# Until convergence.
for i in range(ITERATIONS):
    # Parallel loop over the samples i=1...m
    gradient = points.map(
        lambda p:
            (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1)
            * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= alpha * gradient
print("Final separating plane: %s" % w)
```

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

Repeat {

$$\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

}

# Spark Transformations: 1 / 3

- **map(func)**
  - Return a new RDD passing each element through a function *func()*
- **flatMap(func)**
  - Similar to map, but each input item can be mapped to 0 or more output items
  - *func()* returns a sequence rather than a single item
- **filter(func)**
  - Return a new RDD selecting elements on which *func()* returns true
- **union(otherDataset)**
  - Return a new RDD with the union of the elements in the source dataset and the argument
- **intersection(otherDataset)**
  - Return a new RDD with the intersection of elements in the source dataset and the argument

# Spark Transformations: 2 / 3

- `distinct([numTasks])`
  - Return a new RDD that contains the distinct elements of the source dataset
- `join(otherDataset, [numTasks])`
  - When called on RDDs (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
  - Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`
- `cogroup(otherDataset, [numPartitions])`
  - Aka `groupWith()`
  - Same as `join` but returning a dataset of (K, (Iterable<V>, Iterable<W>)) tuples

# Spark Transformations: 3 / 3

- **groupByKey([numPartitions])**
  - When called on a RDD of (K, V) pairs, return a dataset of (K, Iterable<V>) pairs
  - If you are grouping in order to perform an aggregation (e.g., a sum or average) over each key, **reduceByKey** yields better performance
    - Gathering data and processing in place is better than iterators
  - By default, the level of parallelism in the output depends on the number of partitions of the parent RDD
    - Pass an optional numPartitions argument to set a different number of tasks
- **reduceByKey(func, [numPartitions])**
  - When called on a RDD of (K, V) pairs, return a dataset of (K, f(V\_1, ..., V\_n)) pairs where the values for each key are aggregated using the given reduce function *func()*
  - *func()*: (V, V) → V
  - This is Shuffle + Reduce from MapReduce
  - Number of reduce tasks is configurable through *numPartitions*
- **sortByKey([ascending], [numPartitions])**
  - Return a dataset of (K, V) pairs sorted by keys in ascending or descending order

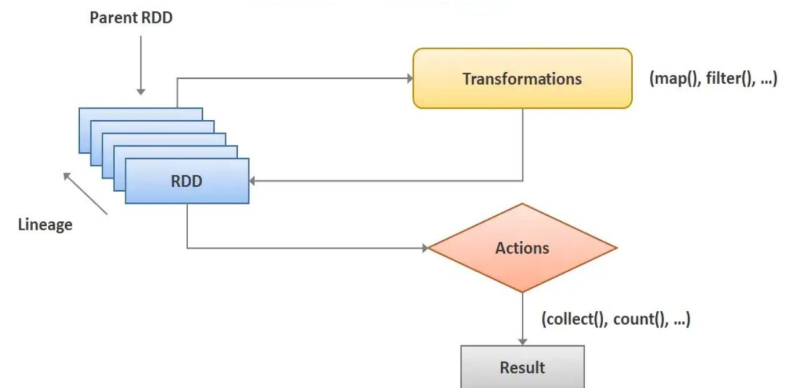
# Spark Actions

- **reduce(func)**
  - Aggregate the elements of the dataset using a function *func()*
  - *func()* takes two arguments and returns one
  - *func()* should be commutative and associative so that it can be computed correctly in parallel
- **collect()**
  - Return all the elements of the dataset as an array
  - This is usually useful after operation that returns a small subset of the data (e.g., **filter()**)
- **count()**
  - Return the number of elements in the dataset
- **take(n)**
  - Return an array with the first *n* elements of the dataset
  - Note that **.collect()[:n]** is not the same as **.take(n)**

<https://spark.apache.org/docs/latest/rdd-programming-guide.html>

# Spark: Fault-tolerance

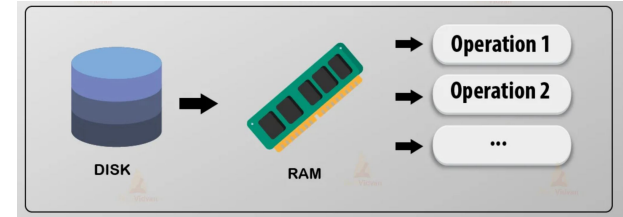
- Spark uses *immutability* and *lineage* to provide fault tolerance
- In case of failure:
  - A RDD can be reproduced by simply replaying the recorded lineage
  - No need to store checkpoints
  - Data can be kept in memory to increase performance
- Fault-tolerance comes for free!





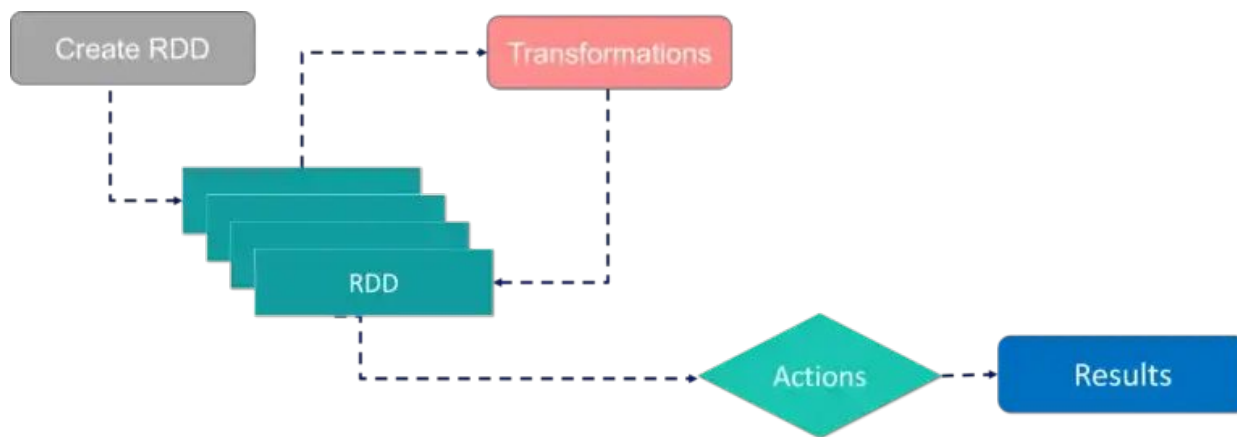
# Spark: RDD Persistence

- **User explicitly persists (aka cache) an RDD**
  - User can call `persist()`, `unpersist()` on RDD
  - Cache only if RDD is expensive to compute
    - E.g., filtering large amount of data
  - When you persist an RDD, each node:
    - Stores (in memory or disk) partitions of the RDD
    - Reuses cached partitions on datasets derived from it
- **Cache**
  - Allows future actions to be much faster (often >10x)
  - Managed by Spark with an LRU policy + garbage collector
- **User can choose storage level**
  - MEMORY\_ONLY (default level)
  - DISK\_ONLY (e.g., Python Pickle)
  - MEMORY\_AND\_DISK
    - If RDD doesn't fit in memory, store partitions on disk
  - MEMORY\_AND\_DISK\_2
    - Same as above, but replicate each partition on two nodes
  - Forcing to cache on disk can be more expensive than not caching!
  - Caching everything is often a bad idea



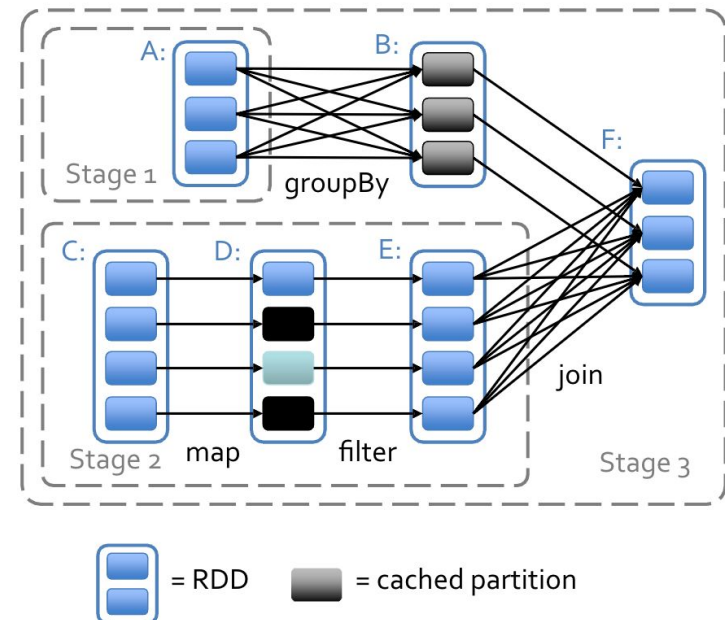
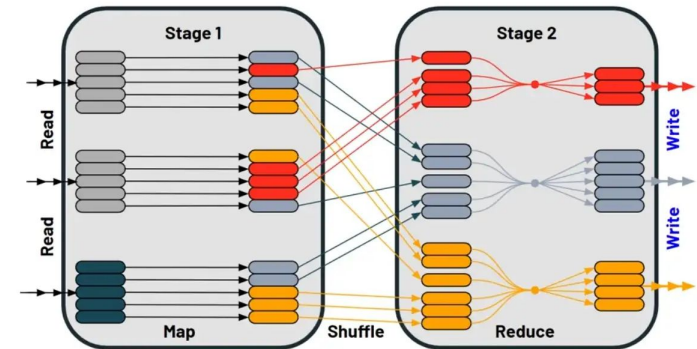
# Spark: RDD Persistence and Fault-tolerance

- Spark handles persistence and fault-tolerance in a similar way
- **Caching/Persistence**
  - Cache RDD (in memory or on disk) instead of recomputing it
- **Fault-tolerance**
  - If any partition of an RDD is lost
    - RDD is automatically recomputed (when needed) using the transformations that generated it
    - Based on immutability and lineage
- **Caching is fault-tolerant!**



# Spark Shuffle

- E.g., **reduceByKey()**
  - **Definition:** all values  $[v_1, \dots, v_n]$  for a single key  $k$  are combined into a tuple  $(k, v)$  where  $v = \text{reduce}(v_1, \dots, v_n)$
  - **Problem:** all the values for a single key need to reside on the same partition / machine
  - **Solution:** data shuffle moving the data across machines
- Certain Spark operations trigger a data shuffle
  - E.g., **reduceByKey()**, **groupByKey()**, join, repartition, transpose
- **Data shuffle** = re-distribute data grouped differently across partitions / machines
- **Data shuffle is expensive** since it involves:
  - Data serialization (pickle)
  - Disk I/O (save to disk)
  - Network I/O (copy across Executors)
  - Deserialization and memory allocation
- **Spark schedules general task graphs**
  - Automatically pipelining of functions
  - Data locality aware
  - Partitioning aware to avoid shuffles



# Broadcast Variables

## - Problem

- Common variables are shipped to the nodes together with the code
- Broadcasting means serializing, sending over the network, de-serializing
- If the data is constant and large, sending the data every time is expensive

## - Solution

- Keep read-only variables cached on each node, instead of shipping a copy with the tasks

```
# `var` is large variable.  
var = list(range(1, int(1e6)))  
# Create a broadcast variable.  
broadcast_var = sc.broadcast(var)  
# Do not modify `var`.  
# Use `broadcast_var.value` instead of `var`.
```

# Accumulators

- **Accumulator** = variable that can be "added to" through associative and commutative operations
  - They can be efficiently supported in parallel execution (e.g., MapReduce)
- Spark supports Accumulators with numerical types by default (e.g., integers)
  - User can define Accumulators for different types

```
>>> accum = sc.accumulator(0)
>>> accum
Accumulator<id=0, value=0>

>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
...

>>> accum.value
10
```

- Each node computes the value to add to the Accumulator and then the value added
- Usual semantic:
  - Accumulators work with the same logic of transformations (lazy evaluation) and actions

```
accum = sc.accumulator(0)
def g(x):
    accum.add(x)
    return f(x)
data.map(g)
# Here, accum is still 0 because no actions have caused the `map` to be computed.
```

# Gray Sort Competition

	Hadoop MR Record	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
<b>Sort rate</b>	<b>1.42 TB/min</b>	<b>4.27 TB/min</b>
<b>Sort rate/node</b>	<b>0.67 GB/min</b>	<b>20.7 GB/min</b>

Spark-based System  
3x faster  
with 1/10  
# of nodes

Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)

<http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>

# Spark vs Hadoop MapReduce

- **Performance:** Spark normally faster but with caveats
  - Spark can process data in-memory
  - Spark generally outperforms MapReduce, but it often needs lots of memory to do well
  - Hadoop MapReduce persists back to the disk after a map or reduce action
- **Ease of use:** Spark is easier to program
- **Data processing:** Spark more general

“Spark vs. Hadoop MapReduce”, Saggi Neumann, 2014

<https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/>