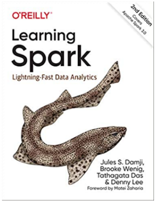
UMD DATA605 - Big Data Systems

9.1: Apache Spark

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu
- **References**:
  - Concepts in the slides
  - Academic paper
    - "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing", 2012
  - Mastery
    - "Learning Spark: Lightning-Fast Data Analytics" (2nd Edition)
    - Not my favorite, but free here



SCIENCE ACADEMY

Hadoop MapReduce: Shortcomings

- **Hadoop is hard to administer**
  - Many layers (HDFS, Yarn, Hadoop, . . .)
  - Extensive configuration
- **Hadoop is hard to use**
  - Verbose API
  - Limited language support (e.g., Java is native)
  - MapReduce jobs read / write data on disk

- **Large but fragmented ecosystem**
  - No native support for:
    - Machine learning
    - SQL, streaming
    - Interactive computing
  - New systems developed on Hadoop for new workloads
  - E.g., Apache Hive, Storm, Impala, Giraph, Drill

(Apache) Spark

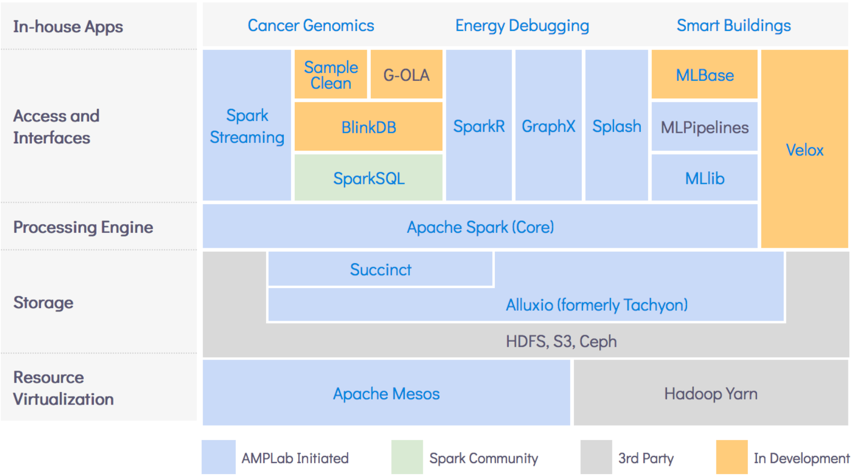
- **Open-source**
  - DataBrick monetizes it ($40B startup)
- **General processing engine**
  - Large set of operations beyond `Map()` and `Reduce()`
  - Combine operations in any order
  - Transformations vs Actions
  - Computation organized as a DAG
    - DAGs decomposed into parallel tasks
  - Scheduler/optimizer for parallel workers
- **Supports several languages**
  - Java, Scala (preferred), Python supported through bindings
- **Data abstraction**
  - Resilient Distributed Dataset (RDD)
  - DataFrames, Datasets built on RDDs
- **Fault tolerance through RDD lineage**
- **In-memory computation**

Berkeley: From Research to Companies

- Amplab
- Rise lab

Berkeley AMPLab Data Analytics Stack

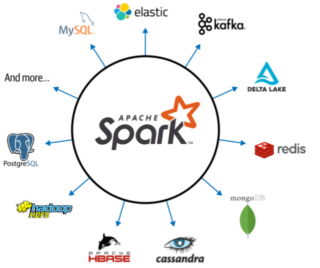
- So many tools that they have their own big data stack! https://amplab.cs.berkeley.edu/software/



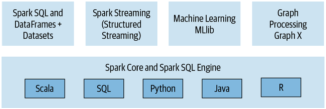
[widt

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

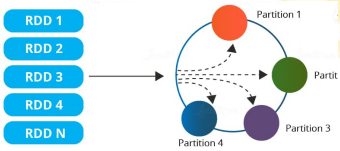


One computation engine



General purpose applications

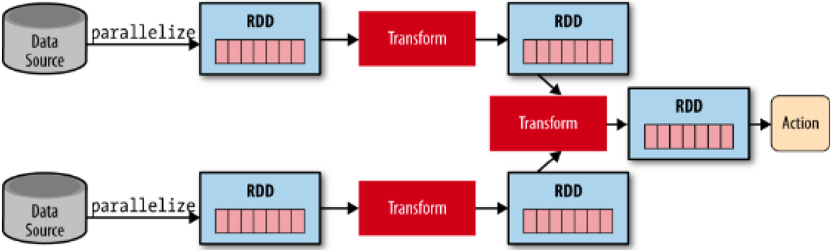
Resilient Distributed Dataset (RDD)



- **A Resilient Distributed Dataset (RDD)**
  - Collection of data elements
  - Partitioned across nodes
  - Operated on in parallel
  - Fault-tolerant
  - In-memory / serializable
- **Applications**
  - Best for applications applying the same operation to all dataset elements (vectorized)
  - Less suitable for asynchronous fine-grained updates to shared state
    - E.g., updating one value in a dataframe
- **Ways to create RDDs**
  - *Reference* data in external storage
    - E.g., file-system, HDFS, HBase
  - *Parallelize* an existing collection in your driver program
  - *Transform* RDDs into other RDDs

Transformations 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

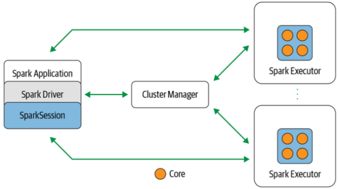
```
# Estimate π (compute-intensive task).
# Pick random points in the unit square [(0,0)-(1,1)].
# See how many fall in the unit circle center=(0, 0), radius=1.
# The fraction should be π / 4.

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

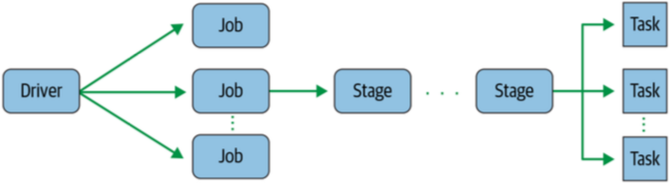
- **Architecture**
  - Who does what
  - Responsibilities of each piece
- **Spark Application**
  - Code describing computation
  - E.g., Python code calling Spark



- **Spark Driver**
  - Instantiate *SparkSession*
  - Communicate with *Cluster Manager* for resources
  - Transform operations into DAG computations
  - Distribute task execution across *Executors*
- **Spark Session**
  - Interface to Spark system
- **Cluster Manager**
  - Manage and allocate resources
  - Support Hadoop, YARN, Mesos, Kubernetes
- **Spark Executor**
  - Run worker node to execute tasks
  - Typically one executor per node
  - JVM

Spark: Computation Model

- **Architecture** = who does what
- **Computational model** = how are things done



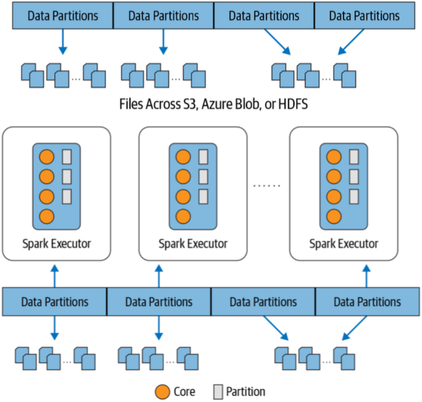
- **Spark Driver**
  - Converts Spark application into one or more Spark *Jobs*
  - Describes computation with *Transformations* and triggers with *Actions*
- **Spark Job**
  - Parallel computation in response to a Spark *Action*
  - Each *Job* is a DAG with one or more dependent *Stages*
- **Spark Stage**
  - Smaller operation within a *Job*
  - *Stages* can run serially or in parallel
- **Spark Task**
  - Each *Stage* has multiple *Tasks*

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** or **Kubernetes**
    - Driver, Cluster Manager, Executors run on different pods (i.e., containers)

Distributed Data and Partitions

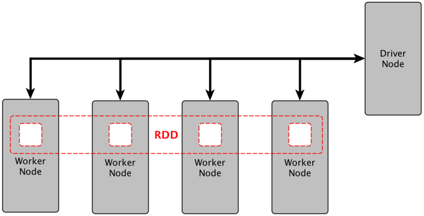
- **Data is distributed** as partitions across different physical nodes
  - Store each partition in memory
  - Enable efficient parallelism
- **Spark Executors** process data "close" to them
  - Minimize network bandwidth
  - Ensure data locality
  - Similar approach to Hadoop



SCIENCE ACADEMY

Parallelized Collections

- Parallelized collections created by calling *SparkContext* `parallelize()` on an existing collection

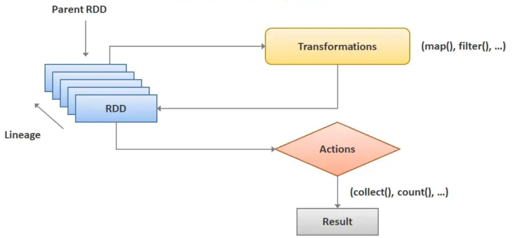


- Data spread across nodes
- Number of *partitions* to cut dataset into
  - Spark runs one *Task* per partition
  - Aim for 2-4 partitions per CPU
    - Spark sets partitions automatically based on your cluster
    - Set manually by passing as a second parameter to `parallelize()`

SCIENCE ACADEMY

Transformations vs Actions

- 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

```
!more data.txt
```

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

```
One a penny, two a penny, hot cross buns
```

```python
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 4 lines

```python
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)]
```

MapReduce in 1 line (show-off version)

Same Code in Java Hadoop

```java
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 (from here)

```
# Load points
points = spark.textFile(...).map(parsePoint)

# Initial separating plane
w = numpy.random.ranf(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(
    ).reduce(lambda a, b: a + b)
    w -= alpha * gradient

print("Final separating plane: %s" % w)
```

Repeat {

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

}

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

Repeat until convergence {

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

}

Spark Transformations: 1 / 3

- `map(func)`
  - Return new RDD passing each element through `func()`
- `flatmap(func)`
  - Map each input item to 0 or more output items
  - `func()` returns a sequence
- `filter(func)`
  - Return new RDD selecting elements where `func()` returns true
- `union(otherDataset)`
  - Return new RDD with union of elements in source dataset and argument
- `intersection(otherDataset)`
  - Return new RDD with intersection of elements in source dataset and argument

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

Spark Transformations: 2 / 3

- `distinct([numTasks])`
  - Return new RDD with distinct elements of source dataset
- `join(otherDataset, [numTasks])`
  - On RDDs `(K, V)` and `(K, W)`, return dataset of `(K, (V, W))` pairs for each key
  - Support outer joins: `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`
- `cogroup(otherDataset, [numPartitions])`
  - Aka `groupWith()`
  - Like join but return dataset of `(K, (Iterable<V>, Iterable<W>))` tuples

SCIENCE ACADEMY

Spark Transformations: 3 / 3

- `groupByKey([numPartitions])`
  - On RDD of `(K, V)` pairs, returns `(K, Iterable<V>)` pairs
  - For aggregation (e.g., sum, average), use `reduceByKey` for better performance
    - Process data in place instead of iterators
  - Output parallelism depends on parent RDD partitions
    - Use `numPartitions` to set tasks
- `reduceByKey(func, [numPartitions])`
  - On RDD of `(K, V)` pairs, returns `(K, f(V_1, ..., V_n))` pairs with values aggregated by `func()`
  - `func(): (V, V)` $\rightarrow$ `V`
  - Shuffle + Reduce from MapReduce
  - Configure reduce tasks with `numPartitions`
- `sortByKey([ascending], [numPartitions])`
  - Returns `(K, V)` pairs sorted by keys in ascending or descending order

SCIENCE ACADEMY

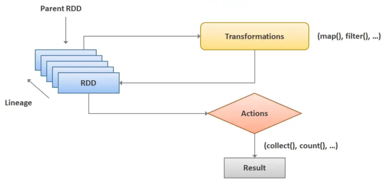
Spark Actions

- `reduce(func)`
  - Aggregate dataset elements using `func()`
  - `func()` takes two arguments, returns one
  - `func()` must be commutative and associative for parallel computation
- `collect()`
  - Return dataset elements as an array
  - Useful after operations returning a small data subset (e.g., `filter()`)
- `count()`
  - Return number of elements in the dataset
- `take(n)`
  - Return array with first `n` dataset elements
  - `.collect()[:n]` differs from `.take(n)`

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

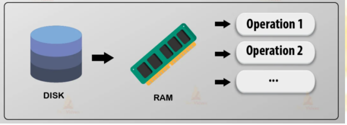
Spark: Fault-tolerance

- Spark uses *immutability* and *lineage* for fault tolerance
- In case of failure:
    - Reproduce RDD by replaying lineage
    - No need for checkpoints
    - Keep data in memory to boost performance
- Fault-tolerance is free!



Spark: RDD Persistence

- **User explicitly persists (aka cache) an RDD**
  - Call `persist()`, `unpersist()` on RDD
  - Cache if RDD is expensive to compute
    - E.g., filtering large data
  - When you persist an RDD, each node:
    - Stores (in memory or disk) partitions of the RDD
    - Reuses cached partitions on derived datasets



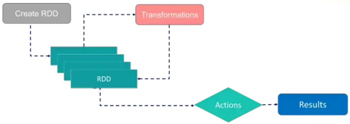
- **Cache**
  - Makes future actions faster (often >10x)
  - Managed by Spark with LRU policy + garbage collector

- **User can choose storage level**
  - `MEMORY_ONLY` (default)
  - `DISK_ONLY` (e.g., Python Pickle)
  - `MEMORY_AND_DISK`
    - If RDD doesn't fit in memory, store on disk
  - `MEMORY_AND_DISK_2`
    - Same as above, replicate each partition on two nodes
  - Caching on disk can be more expensive than not caching

SCIENCE ACADEMY

Spark: RDD Persistence and Fault-tolerance

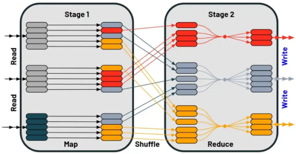
- Spark handles persistence and fault-tolerance similarly
- **Caching/Persistence**
  - Cache RDD (in memory or on disk) instead of recomputing
- **Fault-tolerance**
  - If any partition of an RDD is lost
    - Automatically recompute RDD using transformations that generated it
    - Based on immutability and lineage
- **Caching is fault-tolerant!**

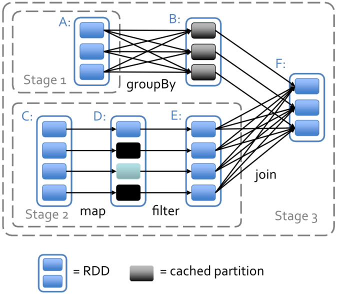


SCIENCE ACADEMY

Spark Shuffle

- E.g., **reduceByKey()**
  - *Definition*: Combine values `[v1, ..., vn]` for key `k` into `(k, v)` where `v = reduce(v1, ..., vn)`
  - *Problem*: Values for a key must be on the same partition/machine
  - *Solution*: Shuffle data across machines
- Certain Spark operations trigger a data shuffle
  - E.g., `reduceByKey()`, `groupByKey()`, join, repartition, transpose
- **Data shuffle** = Re-distribute data across partitions/machines
- **Data shuffle is expensive** due to:
  - Data serialization (pickle)
  - Disk I/O (save to disk)
  - Network I/O (copy across Executors)
  - Deserialization and memory allocation
- **Spark schedules general task graphs**
  - Automatic function pipelining
  - Data locality aware





Broadcast Variables

- **Problem**
  - Ship common variables to nodes with code
  - Broadcasting involves serialization, network transfer, de-serialization
  - Sending large, constant data repeatedly is costly
- **Solution**
  - Cache read-only variables on each node, avoid task copies

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

SCIENCE ACADEMY

Accumulators

- **Accumulator** = variable "added to" through associative, commutative operations
  - Efficient in parallel execution (e.g., MapReduce)
- Spark supports Accumulators with numerical types (e.g., integers)
  - 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 value to add to Accumulator
- Usual semantic:
  - Accumulators use 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 `m
```

SCIENCE ACADEMY

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
Sort rate	1.42 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min

- From here
- Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)
  - Spark-based System 3x faster with 1/10 number of nodes

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/