UMD DATA605 - Big Data Systems

(Apache) Spark

Instructor: Dr. GP Saggese - gsaggese@umd.edu**
TAs: Krishna Pratardan Taduri, kptaduri@umd.edu Prahar Kaushikbhai Modi, pmodi08@umd.edu
v1.1

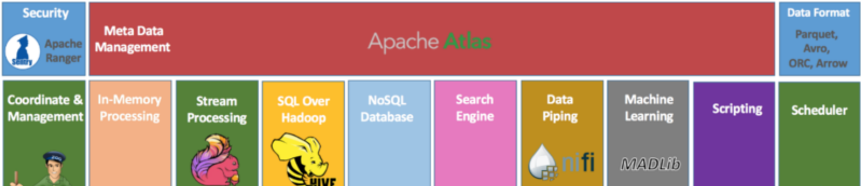
SCIENCE ACADEMY

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

-

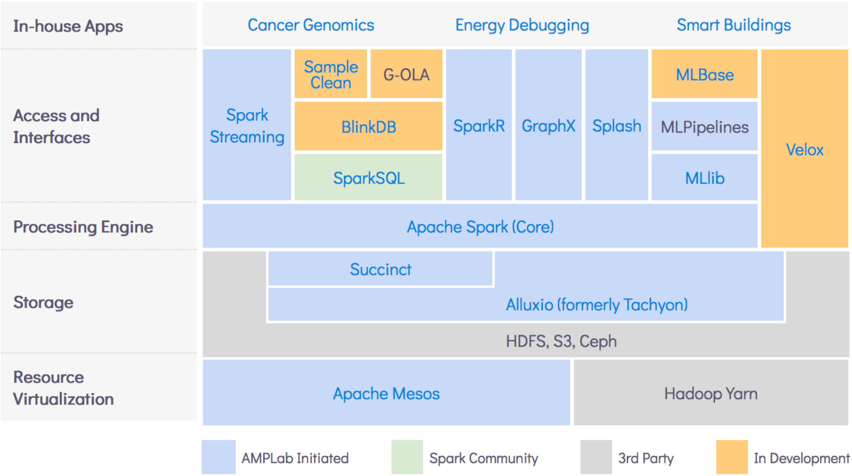
Open-source - DataBrick monetizes it (40B startup) - **General processing engine** - Large set of operations instead of just **Map()** and **Reduce()** - Operations can be arbitrarily combined in any order - Transformations vs Actions - Computation is organized as a DAG - DAGs are decomposed into tasks that can run in parallel - Scheduler / optimizer on parallel workers - **Supports several languages** - Java, Scala (preferred) - Python good support

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

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



- 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

Resilient Distributed Dataset (RDD)

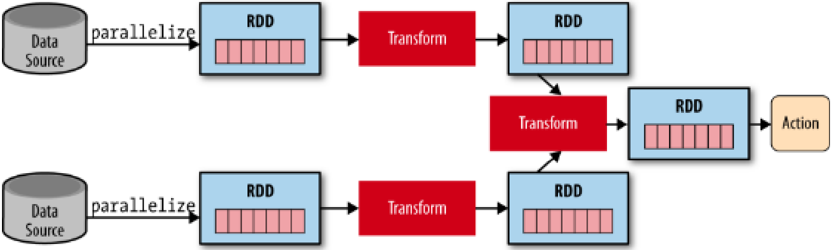
- **A Resilient Distributed Dataset (RDD)**
  - Collection of data elements
  - Partitioned across nodes
  - Can be operated on in parallel
  - Fault-tolerant
  - In-memory / serializable
- **Applications**
  - Best suited for applications that apply the same operation to all elements of a dataset (vectorized)
  - 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

UMD DATA605 - Big Data Systems

Dr. GP Saggese gsaggese@umd.edu with thanks to Alan Sussman, Amol Deshpande, David Wheeler (GMU), T. Yang (UCSB) and Apache documentation

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

```python
# 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
```

- Estimate Pi with MapReduce in Spark

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

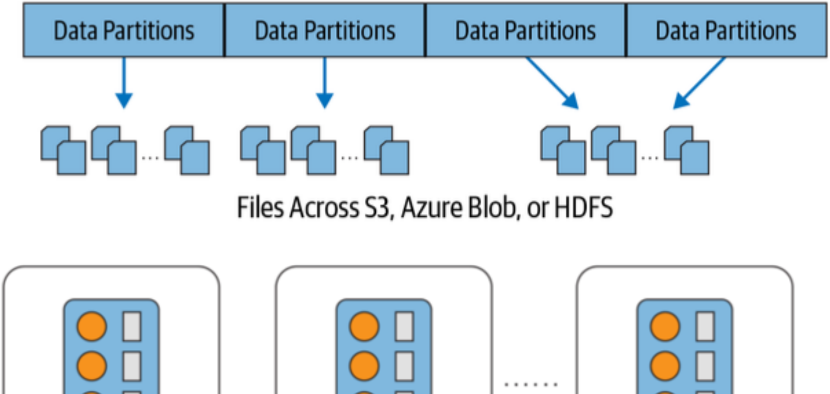
- **Architecture = who does what, what are the responsibilities of each piece**
- **Computational model = how are things done**
- **Spark Driver**
  - The driver converts the Spark application into one or more Spark *Jobs*
  - Computation is described by *Transformations* and triggered by *Actions*
- **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)

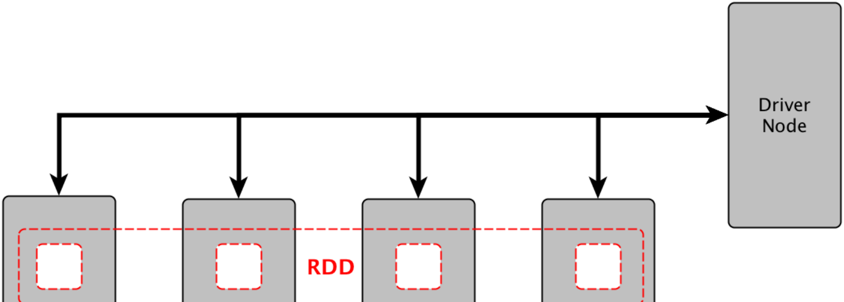
Distributed Data and Partitions

- **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)]
```

```
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);
      }
```

Spark Example: Logistic Regression in MapReduce

$$
\begin{aligned}
&\textit{Repeat } \{ \\
&\qquad \theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)} \\
&\}
\end{aligned}
$$

Logistic Regression

```
####
- _**\textcolor{red}{Load points}**_
- Initial separating plane
- Until convergence
- `var` is large variable.
- Create a broadcast variable.
- Do not modify `var`.
```

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

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

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, Iterable)) tuples

Spark Transformations: 3 / 3

- **groupByKey**(*[numPartitions]*)
  - When called on a RDD of (K, V) pairs, return a dataset of (K, Iterable) 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) $\rightarrow$ 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

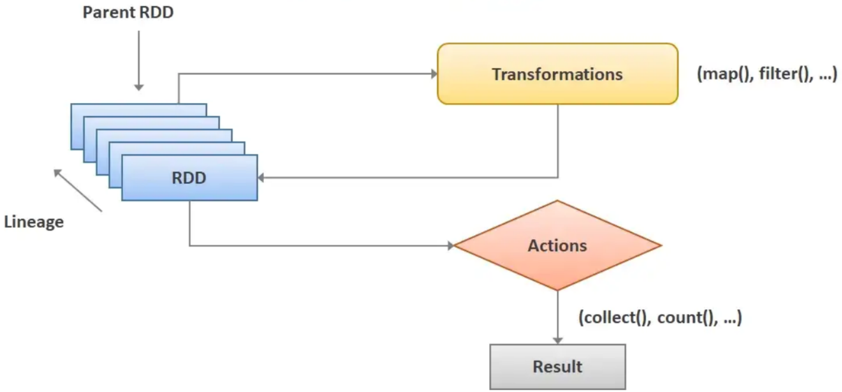
Spark Actions

- **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!



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

http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large

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

SCIENCE ACADEMY

Spark-based System 3x faster with 1/10 #### - Load points - Initial separating plane - Until convergence - var is large variable - Create a

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/