# CS 555: Distributed Systems [Spark]

Shrideep Pallickara
Computer Science
Colorado State University



## Frequently asked questions from the previous class survey

- Custom Partitioners
  - How often, example?
- How does Hadoop decide whether or not to call the combiner?

## Topics covered in this lecture

- Spark
  - Software stack
  - Interactive shells in Spark
  - Core Spark concepts



# APACHE SPARK

## Spark: What is it?

- **Cluster computing platform**
  - Designed to be fast and general purpose
- Speed
  - Often considered to be a design alternative for Apache MapReduce
  - Extends MapReduce to support more types of computations
    - Interactive queries, iterative tasks, and stream processing
- Why is speed important?
  - Difference between waiting for hours versus exploring data interactively



## Spark: Influences and Innovations

- Spark has inherited parts of its API, design, and supported formats from other existing computational frameworks
  - Particularly DryadLINQ
- Spark's internals, especially how it handles failures, differ from many traditional systems
- Spark's ability to leverage **lazy evaluation** within memory computations makes it particularly unique

## Where does Spark fit in the Analytics Ecosystem?

- Spark provides methods to process data in parallel that are **generalizable**
- On its own, Spark is ***not*** a data storage solution
  - Performs computations on Spark JVMs that last only for the duration of a Spark application
- Spark is used in tandem with:
  - A distributed storage system (e.g., HDFS, Cassandra, or S3)
    - To house the data processed with Spark
  - A cluster manager — to orchestrate the distribution of Spark applications across the cluster



## Key enabling idea in Spark

- **Memory-resident data**
- Spark loads data into the memory of worker nodes
  - Processing is performed on memory-resident data

## A look at the memory hierarchy

| Item | time | Scaled time in human terms (2 billion times slower) |
|---|---|---|
| Processor cycle | 0.5 ns (2 GHz) | 1 second |
| Cache access | 1 ns (1 GHz) | 2 seconds |
| Memory access | 70 ns | 140 seconds |
| Context switch | 5,000 ns (5 µs) | 167 minutes |
| Disk access | 7,000,000 ns (7 ms) | 162 days |
| Quantum | 100,000,000 ns (100 ms) | 6.3 years |

Source: Kay Robbins & Steve Robbins. *Unix Systems Programming*, 2nd edition, Prentice Hall.



## Spark covers a wide range of workloads

- Batch applications
- Iterative algorithms
- Queries
- Stream processing

- This has previously required multiple, independent tools

## Running Spark

- You can use Spark from Python, Java, Scala, R, or SQL
- Spark itself is written in **Scala**, and runs on the Java Virtual Machine (JVM)
  - You can Spark either on your laptop or a cluster, all you need is an installation of Java
- If you want to use the Python API, you will also need a Python interpreter (version 2.7 or later)
- If you want to use R, you will need a version of R on your machine.



## Spark integrates well with other tools

- Can run in Hadoop clusters
- Access Hadoop data sources, including Cassandra

## At its core, Spark is a **computational engine**

- Spark is responsible for several aspects of applications that comprise
  - Many tasks across many machines (compute clusters)
- Responsibilities include:
  1. Scheduling
  2. Distributions
  3. Monitoring



## Spark execution

- The cluster of machines that Spark will use to execute tasks is managed by a cluster manager
  - Spark's standalone cluster manager, YARN, or Mesos
- We submit Spark Applications to these cluster managers, which will grant resources to the application to complete the work

## Spark applications

- Spark Applications consist of
  - A **driver** process
    - The driver process is absolutely essential
    - The heart of a Spark Application and maintains all relevant information during the lifetime of the application
  - A set of **executor processes**



## The Driver

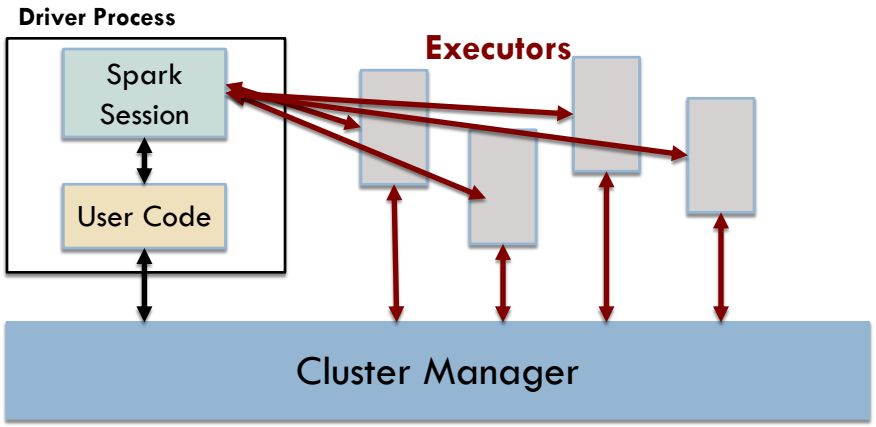
- The driver process runs your main() function, sits on a node in the cluster
- Driver is responsible for three things:
  - Maintaining information about the Spark Application
  - Responding to a user's program or input
  - Analyzing, distributing, and scheduling work across the executors

## The executors

- The executors are responsible for actually carrying out the work that the driver assigns them
- Each executor is responsible for only two things:
  - Executing code assigned to it by the driver, and
  - Reporting the state of the computation on that executor back to the driver node

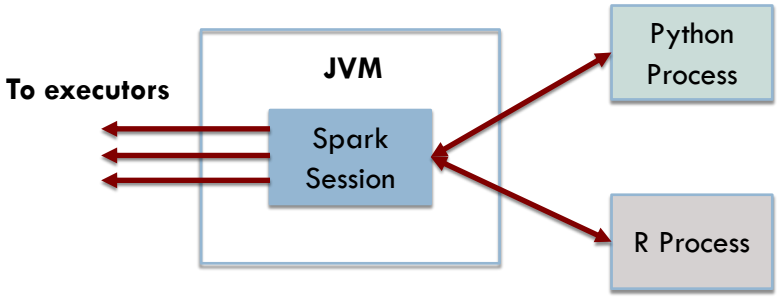


## Architecture of a Spark Application



## How Spark runs Python or R

- You write Python and R code that Spark translates into code that it then can run on the executor JVM






## SparkSession

- We need a way to send user commands and data to a Spark Application
  - We do that by first creating a SparkSession
- The SparkSession instance is the way Spark executes user-defined manipulations across the cluster.
  - There is a one-to-one correspondence between a SparkSession and a Spark Application
  - In Scala and Python, the variable is available as `spark` when you start the console.

## Partitions

- To allow every executor to perform work in parallel, Spark breaks up the data into chunks called **partitions**
  - For e.g., a partition may be a collection of rows that sit on one physical machine in your cluster
- If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors
- If you have many partitions but only one executor?
  - Spark will still have a parallelism of only one because there is only one computation resource



## Manipulation of Partitions

- An important thing to note is that you do not (for the most part) manipulate partitions manually or individually
  - You simply specify high-level transformations of data in the physical partitions
    - Spark determines how this work will actually execute on the cluster.

## Transformations

- In Spark, the core data structures are **immutable**
  - They cannot be changed after they're created
- If you cannot change it, how are you supposed to use it?
  - You need to *instruct Spark* how you would like to modify it
  - These instructions are called **transformations**



## More about transformations

- Transformations do not return an output
  - Spark will not act on transformations until we call an **action**

## Lazy evaluation

- Spark will **wait until the very last moment** to execute the graph of computation instructions
- When you express some operation?
  - You do not modify the data immediately
  - Rather, you build up a **plan** of transformations that you would like to apply to your source data.



## Why lazy evaluation works …

- By waiting until the last minute to execute the code
  - Spark compiles this plan from your transformations to a streamlined physical plan that will run as efficiently as possible across the cluster
- This provides immense benefits because Spark can optimize the entire data flow from end to end
  - E.g. **predicate pushdowns**

## Spark in a nutshell

- Spark is a distributed programming model in which the user specifies transformations
  - Multiple transformations build up a directed acyclic graph of instructions
- An action begins the process of executing that graph of instructions
  - As a single job
  - By breaking it down into stages and tasks to execute across the cluster



## Spark APIs

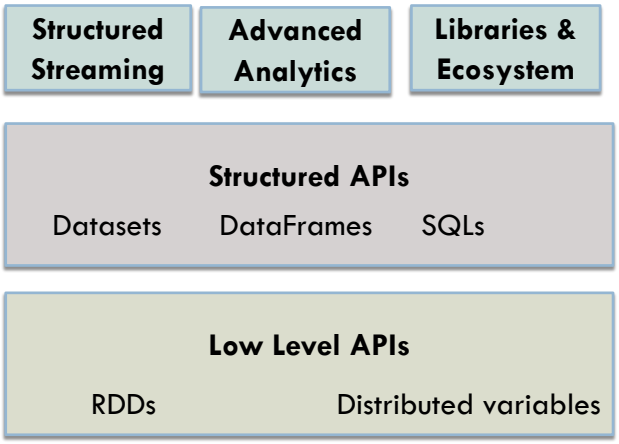
- Spark has two fundamental sets of APIs:
  - The low-level "unstructured" APIs, and
  - The higher-level structured APIs

## Structured APIs

- Structured APIs are a tool for manipulating all sorts of data
  - From unstructured log files to semi-structured CSV files and highly structured Parquet files
- Refers to three core types of distributed collection APIs:
  - Datasets
  - DataFrames
  - SQL tables and views
- Majority of the Structured APIs apply to both batch and streaming computation



## Spark's Toolset



## Spark has two notions of structured collections: DataFrames and Datasets

- DataFrames and Datasets are (distributed) table-like collections with well-defined rows and columns
- Each column:
  - Must have the same number of rows as all the other columns (although you can use null to specify the absence of a value)
  - Has type information that must be consistent for every row in the collection.



## DataFrames versus Datasets

- DataFrames are considered "untyped"
- Datasets are considered "typed"

## How does Spark view DataFrames and Datasets?

- To Spark, DataFrames and Datasets represent immutable, lazily evaluated plans that specify what operations to apply to data residing at a location to generate some output
- When we perform an action on a DataFrame, we instruct Spark to perform the actual transformations and return the result
- These represent **plans** of how to manipulate rows and columns to compute the user's desired result.



## The DataFrame is the most common Structured API

- Simply represents a **table** of data with rows and columns
- The list that defines the columns and the types within those columns is called the **schema**

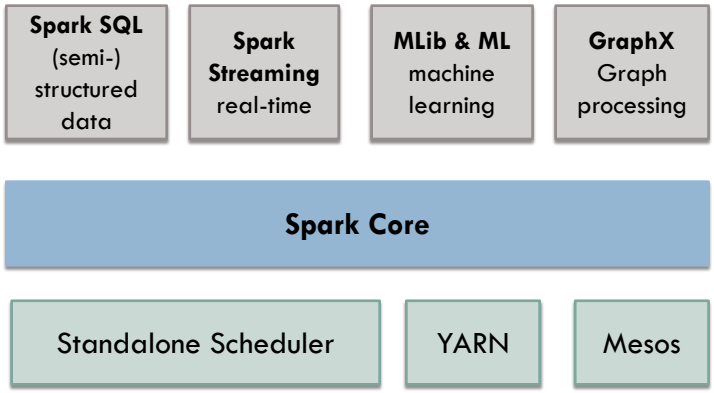
## The DataFrame concept is not unique to Spark

- R and Python both have similar concepts.
  - However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines
  - This limits what you can do with a given DataFrame to the resources that exist on that specific machine
- A Spark DataFrame can span thousands of computers.



## THE SPARK SOFTWARE STACK

## The Spark stack






## Spark Core

- **Basic functionality** of Spark
- Task scheduling, memory management, fault recovery, and interacting with storage systems
- Also, the API that defines Resilient Distributed Datasets (**RDD**s)
  - Spark's *core programming abstraction*
  - Represents collection of data items dispersed across many compute nodes
    - Can be manipulated concurrently (parallel)

## Spark SQL

- Package for working with **structured data**
- Allows querying data using SQL and HQL (Hive Query Language)
  - Data sources: Hive tables, Parquet, and JSON
- Allows intermixing queries with programmatic data manipulations support by RDDs
  - Using Scala, Java, and Python



## **Semi-structured** data and Spark SQL

- Spark SQL defines an interface for a semi-structured data type, called **DataFrames**
  - And as of Spark 1.6, a semi-structured, typed version of RDDs called **Datasets**
- Spark SQL is a very important component for Spark performance
- Much of what can be accomplished with Spark Core can be done by leveraging Spark SQL.

## Spark Streaming

- Enables processing of **live streams** of data from sources such as:
  - Logfiles generated by production webservers
  - Messages containing web service status updates
- Uses the scheduling of the Spark Core for streaming analytics on **minibatches** of data
- Has a number of unique considerations, such as the *window sizes* used for batches



## MLib

- MLlib is a package of machine learning and statistics algorithms written with Spark
- Algorithms include:
  - Clustering, classification, regression, clustering, and collaborative filtering
- Low-level primitives
  - Generic gradient descent optimization algorithm
- Alternatives?
  - Mahout, sci–kit learn, VW, WEKA, and R among others

## What about Spark ML?

- Still in the early stages, and has only existed since Spark 1.2
- Spark ML provides a higher-level API than MLlib
  - Goal is to allow users to more easily create practical machine learning **pipelines**
  - Spark MLlib is primarily built on top of RDDs and uses functions from Spark Core, while ML is **built on top of Spark SQL DataFrames**
- Eventually the Spark community plans to move over to ML and deprecate MLlib



## Graph X

- Library for manipulating graphs
- Graph-parallel computations
- Extends Spark RDD API
  - Create a **directed graph**, with arbitrary properties attached to each vertex and edge

## Cluster Managers

- Spark runs over a variety of cluster managers
- These include:
  - Hadoop YARN
  - Apache Mesos
  - Standalone Scheduler
    - Included within Spark



## Storage Layers for Spark

- Spark can create distributed datasets from any file stored in HDFS
- Plus, other storage systems supported by the Hadoop API
  - Amazon S3, Cassandra, Hive, HBase, etc.

# INTERACTIVE SHELLS IN SPARK



## Spark Shells

- Interactive [Python and Scala]
  - Similar to shells like Bash or Windows command prompt
- *Ad hoc* data analysis
- Traditional shells manipulate data using disk and memory on a single machine
  - Spark shells allow interaction with **data that is distributed** across many machines
  - Spark manages complexity of distributing processing

## Several software were designed to run on the Java Virtual Machine

- Languages that compile to run on the JVM and can interact with Java software packages but are *not actually* Java
- There are a number of non-Java JVM languages
  - The two most popular ones used in real-time application development: **Scala** and **Clojure**



## Scala

- Has spent most of its life as an academic language
  - Still largely developed at universities
  - Has a rich standard library that has made it appealing to developers of high-performance server applications
- Like Java, Scala is a strongly typed object-oriented language
  - Includes many features from functional programming languages that are not in standard Java
  - Interestingly, Java 8 incorporate several of the more useful features of Scala and other functional languages.

## What is functional programming?

- When a method is compiled by Java, it is converted to instructions called byte code and …
  - Then largely disappears from the Java environment
    - Except when it is called by other methods
- In a functional language, **functions are treated the same way as data**
  - Can be stored in objects similar to integers or strings, returned from functions, and passed to other functions



## What about Clojure?

- Based on Lisp
- Javascript?
  - Name was a marketing gimmick
  - Closer to Clojure and Scala than it is to Java

## The contents of this slide-set are based on the following references

- *Learning Spark: Lightning-Fast Big Data Analysis. 1st Edition. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. O'Reilly. 2015. ISBN-13: 978-1449358624.* [Chapters 1-4]
- Karau, Holden; Warren, Rachel. High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark. O'Reilly Media. 2017. ISBN-13: 978-1491943205. [Chapter 2]
- Chambers, Bill,Zaharia, Matei. Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media. ISBN-13: 978-1491912218. 2018. [Chapters 1, 2, and 3].
- *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*. Byron Ellis. Wiley. [Chapter 2]