

# Apache Spark

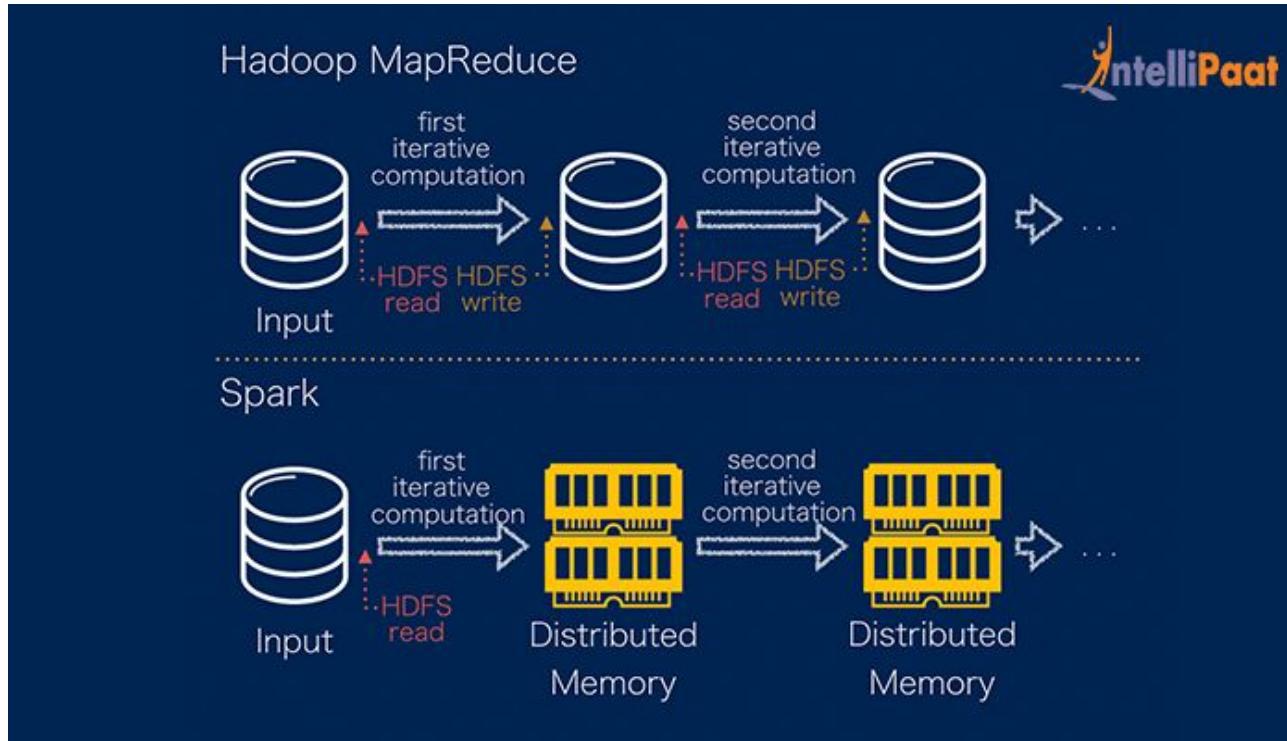
---

Lecture 8

# Challenges with Hadoop

- Hadoop = HDFS + MapReduce + YARN + (Hive and/or Pig)
- **Programming** MapReduce jobs might not be desirable all the time
- MapReduce requires data to be copied to the multiple times (3+) to the **disk** while processing
  - Initial copy to HDFS
  - Intermediate Key Value pairs
  - Shuffling
  - Final result on HDFS
- A **hive** query or a **pig** latin will translate into a chain of MapReduce jobs
  - Each job interactive with the disk 3+ times

# Hadoop Vs Spark



# What is Apache Spark?

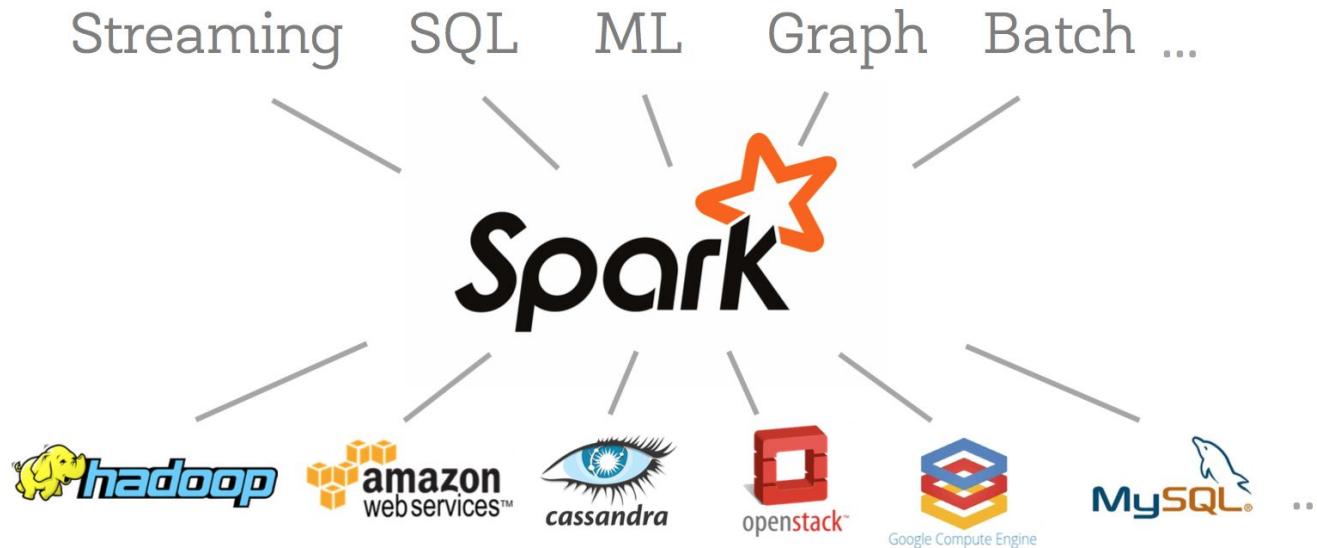
Fast and general cluster computing system for analytics and large scale data processing. Interoperable with Hadoop,

- Improves efficiency through:
  - In-memory computing primitives
  - General computation graphs
  - Up to 100× faster
- Improves usability through:
  - Rich APIs in Scala, Java, Python
  - Interactive shell
  - 2-5× less code
  - 80+ inbuilt operators

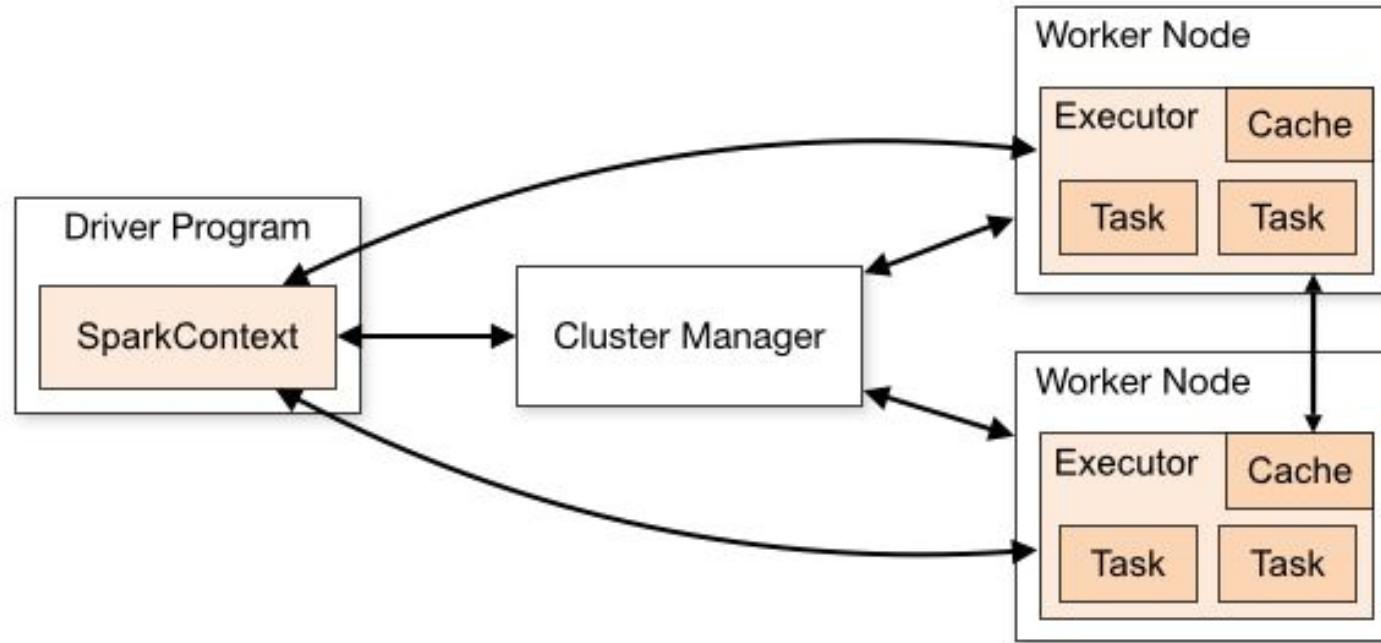
<http://spark.apache.org/>

# Spark Echo System

Unified engine across data workloads and platforms



# Cluster Architecture



# Cluster Architecture

- **Driver**

- Separate JVM running the client program
- Includes the ‘SparkContext’, which coordinates the spark applications
- Negotiates with cluster manager for task scheduling and execution

- **Cluster Manager**

- Can be Kubernetes, Mesos, YARN or In Built resource manager.
- Helps the driver program to acquire resources (executors)

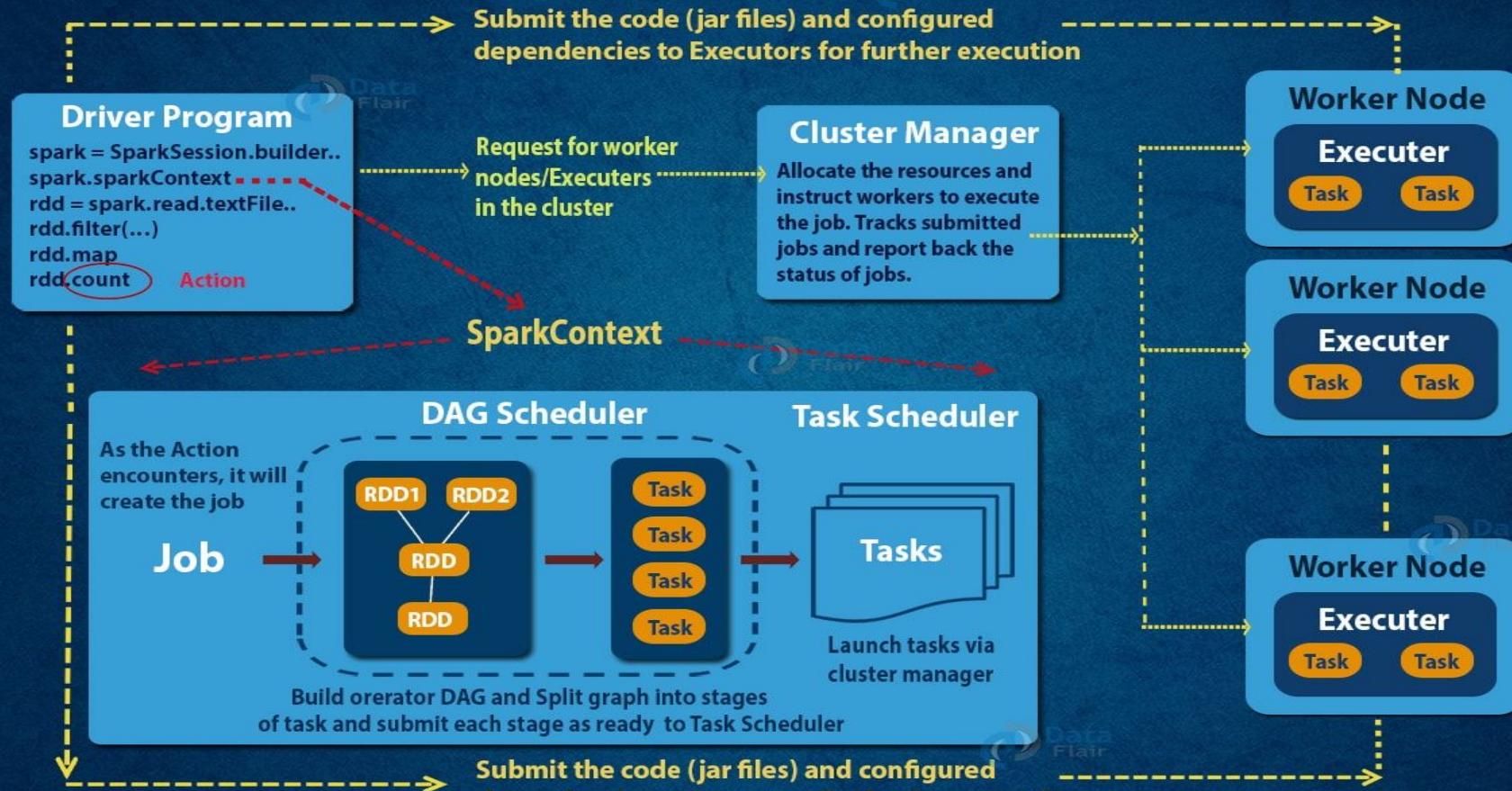
- **Worker Node**

- A physical machine or a VM that runs spark executors in the cluster

- **Executor**

- A separate process/JVM which runs ‘tasks’ in separate threads
- Keeps results in memory/disk
- Each Application has its own set of executors
- Interact with external storages for reading and writing

# Internals of Job Execution In Spark



# Resilient, Distributed, DataSets (RDDs)

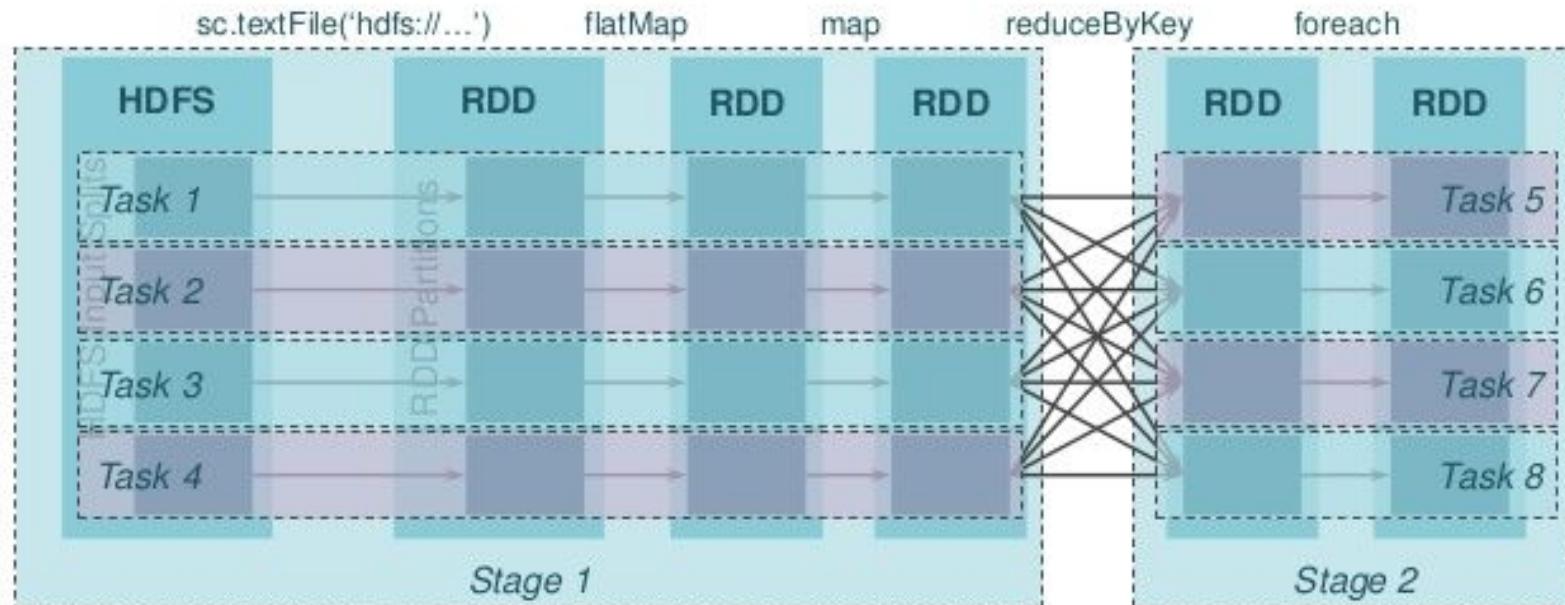
- RDD is an collection of data items split across the cluster as one or more partitions
  - Immutable : Once created can't be changed
  - Refers to data stored in memory, in persisted stores or in another RDD in different nodes.
  - Stores both structured and unstructured data
- RDD Operations
  - **Transformations** : lazyly executed operation on a RDD which creates one or more RDDs
    - **Narrow** : Transformations that do not require data to be shuffled (map, filter)
    - **Wide**: Transformations that requires the data to be shuffled (reduceByKey)
  - **Actions** : Performed on one or more RDDs and generates a result and store or display the results (count)

```
val rdd = sc.textFile("spam.txt") ← Initial Loading
val filtered = rdd.filter(line => line.contains("money")) ← Transform
filtered.count() ← Action
```

# Directed Acyclic Graph(DAG)

- The logical execution plan or the RDD lineage is transformed into a physical execution plan modeled as a DAG
  - **Node** : RDD Data partition
  - **Edge**: Transformation or action
- **DAGScheduler** schedules the physical implementation plan in a ***stage-oriented*** manner
  - The DAG is split into multiple '**stages**'
  - Usually narrow transformations are grouped together
  - Each stage is submitted into the task scheduler
  - A '**task**' is an execution of a 'stage' on a 'single partition'. Hence the number of tasks are calculated based on the number of partitions

# Word Count Example



# RDD

- Spark SQL let's uses perform structured data processing using the spark engine
  - RDDs does not require to have a structure/schema to the data that it holds.

```
// Load CSV as RDD
val rdd = sc.textFile("employees.csv")
// Split by commas and map to a tuple (name, age)
val rddMapped = rdd.map(line => {
    val parts = line.split(",")
    (parts(0), parts(1).toInt) // (name, age)
})
// Filter employees over 30 years old
val filteredRDD = rddMapped.filter { case (name, age) => age > 30 }
// Collect the results and print
filteredRDD.collect().foreach(println)
```

# DataFrames

- **DataFrames** organizes data into names columns like in a relational table. But not type safe.
  - Introduced first in spark 1.3
  - Only handles structured or semi-structured data
  - Offers high performance by allowing more optimizations

```
// Load CSV as DataFrame
val df = spark.read.option("header",
  "false").csv("employees.csv").toDF("name", "age")
// Perform transformations: Filter employees over 30 years old
val filteredDF = df.filter($"age" > 30)
// Show the results
filteredDF.show()
```

# Datasets

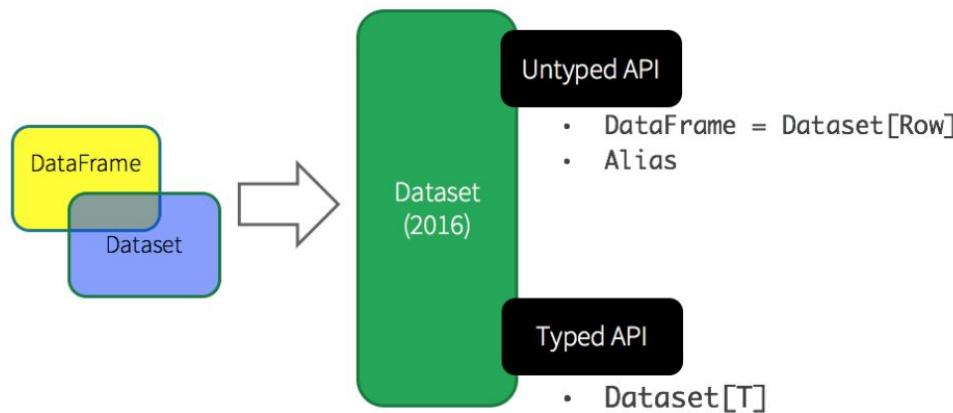
- **DataSets** extension to DataFrame API which brings in benefits of RDD such as type safety, OOP interface
  - Introduced first in spark 1.6
  - Can process both structured and unstructured data
  - Slower than DataFrames but faster than RDDs

```
// Define a case class for strongly typed data
case class Employee(name: String, age: Int)
// Load CSV as DataFrame, then convert to Dataset[Employee]
val df = spark.read.option("header",
  "false").csv("employees.csv").toDF("name", "age")
val dataset: Dataset[Employee] = df.as[Employee]
// Perform transformations: Filter employees over 30 years old
val filteredDataset = dataset.filter(_.age > 30)
// Show the results
filteredDataset.show()
```

# Datasets and DataFrames

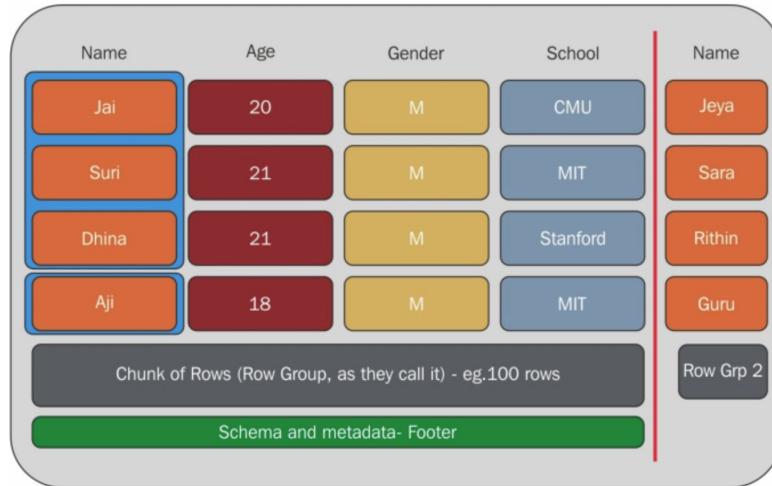
- From Spark 2.0 onwards DataSets and DataFrames APIs are unified
  - DataFrame = Dataset[Row]. Where ‘Row’ is a generic, **untyped** JVM object
  - DataSet = Dataset[T]. Where T is a **type**, or a class name (e.g., Dataset[Employee])

Unified Apache Spark 2.0 API



# Parquet Files

- ‘Parquet’ is a file format compatible with Hadoop
  - Spark supports both reading and writing
- Store data in a columnar storage format
  - Efficient when it’s only required to access a smaller subset of columns in a larger table
  - Uses ‘Row Group’ logical horizontal partitioning for easy organization
- Uses compression and encoding to minimize the disk usage
  - each column can be compressed with what best fits for it.



# More than Map & Reduce

- map
- reduce
- filter
- count
- groupBy
- fold
- sort
- reduceByKey
- union
- groupByKey
- join
- cogroup
- leftOuterJoin
- cross
- rightOuterJoin
- zip
- sample
- take
- first
- partitionBy
- mapWith
- pipe
- save
- ...

# Notebooks - Apache Zeppelin

Analytics notebook for spark and many others

Multi-purpose Notebook

- Data Ingestion
- Data Discovery
- Data Analytics
- Data Visualization & Collaboration



# Lab Session

# Starting the Docker Containers

- Clone the below git repository

```
git clone https://github.com/ramindu-msc/iit
```

Or if you have the repository created already

```
cd iit
```

```
git pull origin main
```

- Change `/home/iitgcpuser/iit/lab7/hadoop-spark-dockercompose/`'s  
`/home/iitgcpuser/iit/lab7/hadoop-spark-dockercompose/` paths to your repository cloned path
- And start the containers with the following command(also by replacing the path)  
`sudo docker compose -f`  
`/home/iitgcpuser/iit/lab7/hadoop-spark-dockercompose/docker-compose.yaml up -d`
- Check the datanode is connected to namenode and fully started  
`docker logs -f datanode`

# Accessing files in HDFS

- sudo docker exec -it namenode bash
- hdfs dfs -mkdir -p /data/openbeer/breweries
- hdfs dfs -put /opt/resources/breweries.csv /data/openbeer/breweries/breweries.csv
  
- sudo docker exec -it spark-master bash
- spark/bin/spark-shell --master spark://spark-master:7077
- val df = spark.read.csv("hdfs://namenode:9000/data/openbeer/breweries/breweries.csv")
- df.show()

# Create Dataframe and Parquet Files

- docker exec -it namenode bash
- hdfs dfs -mkdir -p /data
- docker exec -it spark-master bash
- mkdir -p /tmp/spark-events
- chmod 777 /tmp/spark-events
- spark/bin/spark-shell --master spark://spark-master:7077 --conf "spark.eventLog.enabled=true" --conf "spark.eventLog.dir=/tmp/spark-events"
- :load /opt/resources/CreateParquet.scala
- CreateParquetFile.main(Array.empty)
- :load /opt/resources/AppendToParquet.scala
- AppendToParquet.main(Array.empty)

# Setup Zeppelin Notebooks

```
sudo docker pull apache/zeppelin:0.9.0
```

```
sudo docker run -it --name zeppelin -p 8089:8080 -v  
/home/iitgcpuser/iit/lab7/hadoop-spark-dockercompose/resources:/opt/resources  
apache/zeppelin:0.9.0
```

Url : <http://34.41.115.194:8089/>

Try the samples

Import spark-lab.json and try it out.

Refer to <https://github.com/apache/spark/tree/master/examples/src/main/resources> for other resources

# Resources

- <http://spark.apache.org/docs/latest/sql-programming-guide.html#jdbc-to-other-databases>
- [http://www.slideshare.net/databricks/jump-start-with-apache-spark-20-on-data-bricks?qid=78d548b5-9eeb-47cc-9474-9ea842384af3&v=&b=&from\\_search=3](http://www.slideshare.net/databricks/jump-start-with-apache-spark-20-on-data-bricks?qid=78d548b5-9eeb-47cc-9474-9ea842384af3&v=&b=&from_search=3)
- [http://www.slideshare.net/databricks/apache-spark-20-faster-easier-and-smarter?qid=34ba75fe-657b-4912-8c6c-7bbc82d62f8c&v=&b=&from\\_search=1](http://www.slideshare.net/databricks/apache-spark-20-faster-easier-and-smarter?qid=34ba75fe-657b-4912-8c6c-7bbc82d62f8c&v=&b=&from_search=1)
- Using hive table in Spark : <https://www.youtube.com/watch?v=RXfczaorrgI>
- Learn spark: [www.databricks.com/ce](http://www.databricks.com/ce)
- <https://www.youtube.com/watch?v=ssPBlqiRJGY>
- <http://www.slideshare.net/jeykottalam/spark-sqlamp-camp2014>