1. SPARK & HDFS SETUP

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching, and optimized query execution for fast analytic queries against data of any size [1]. While HDFS is a distributed file system that splits a large data file into multiple chunks and stores them across the cluster, Spark and HDFS together offer powerful capabilities that can quickly compute massive amounts of data in parallel. In this lab, we need 3 Linux VMs (or at least 2) to create a cluster of Spark and HDFS by following the steps below.

*** Initialized setup:**

- **Step 1:** First, each VM needs to identify the others in the cluster associated with its IP address. In each VM, open */etc/hosts* and append the lines below.

```
<IP machine 1>  node1   # Update this value when joining new network
<IP machine 2>  node2   # Update this value when joining new network
<IP machine 3>  node3   # Update this value when joining new network
```

- **Step 2:** Install ssh server in each VM for remote access to other hosts.

```
$ sudo apt install openssh-server -y
$ sudo systemctl enable ssh
$ sudo systemctl restart ssh
```

- **Step 3:** in each VM, creates a new user hadoop who is responsible for running all Spark related tasks.

```
$ sudo useradd -s /bin/bash -m hadoop
$ sudo passwd hadoop
$ su hadoop
$ cd /home/hadoop
```

- **Step 4:** Create key for remote access without password.

```
# Create encrypted key and distribute it to other host for direct access
$ ssh-keygen
$ ssh-copy-id node1
$ ssh-copy-id node2
$ ssh-copy-id node3

# Verify ssh-keygen, if success exit to return
$ ssh node2
$ exit
```

- **Step 5:** Export all necessary environment variables by appending the following lines to /home/hadoop/.bashrc.

```
# Open file ~/.bashrc and add the lines below
export HADOOP_HOME=/home/hadoop/hadoop
export YARN_CONF_DIR=${HADOOP_HOME}/etc/hadoop
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/  # must be modified if other java version is installed
export SPARK_HOME=/home/hadoop/spark
export HADOOP_CONF_DIR="/home/hadoop/hadoop/etc/hadoop"
export SPARK_YARN_QUEUE="default"
export PATH=${PATH}:${HADOOP_HOME}/bin:${HADOOP_HOME}/sbin:${SPARK_HOME}/bin
```

[1] https://aws.amazon.com/what-is/apache-spark/

- **Step 6:** Update environment variables.

```
$ source /home/hadoop/.bashrc
```

*** HDFS setup:** Do in 1 VM then distribute to other VMs.

- **Step 1:** Download and install HDFS.

```
$ curl -O https://mirror.downloadvn.com/apache/hadoop/common/stable/hadoop-3.4.0.tar.gz
$ tar -xzf hadoop-3.4.0.tar.gz
$ mv hadoop-3.4.0 hadoop
$ cd /home/hadoop/hadoop/etc/hadoop
```

- **Step 2:** Update JAVE_HOME enviroment variable at *hadoop-env.sh*.

```
# Open file hadoop-env.sh
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/
```

- **Step 3:** Config HDFS in *core-site.xml*, *hdfs-site.xml* and *yarn-site.xml* respectively.

```
# At file core-site.xml
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://node1:9000</value>
  </property>
</configuration>
```

```
# At file hdfs-site.xml
<configuration>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/home/hadoop/data/namenode</value>
  </property>
  <property>
    <name>dfs.block.size</name>
    <value>10485760</value>
    <description>Block size</description>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>/home/hadoop/data/datanode</value>
  </property>
</configuration>
```

```
# At file yarn-site.xml
<configuration>
  <property>
    <name>yarn.acl.enable</name>
    <value>0</value>
  </property>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value> <IP machine 1> </value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>1536</value>
  </property>
  <property>
    <name>yarn.schedule.maximum-allocation-mb</name>
    <value>1536</value>
  </property>
  <property>
    <name>yarn.schedule.minimum-allocation-mb</name>
    <value>128</value>
  </property>
  <property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
  </property>
</configuration>
```

- **Step 3:** Define all hosts participate in Hadoop-HDFS. Open file *workers* and replace by the following content.

```
node1
node2
node3
```

- **Step 4:** Copy the complete installation to other hosts.

```
$ cd /home/hadoop
$ scp -r hadoop/ node2:/home/hadoop/
$ scp -r hadoop/ node3:/home/hadoop/
```

- **Step 5:** Start HDFS.

```
# Start HDFS, to stop HDFS use stop-dfs.sh instead
$ hdfs namenode -format
$ start-dfs.sh

# Start yarn resource manger or use stop-yarn.sh to stop
$ start-yarn.sh

# Verify HDFS
$ hdfs dfs -mkdir -p /user/hadoop
$ hdfs dfs -ls  # execute this command on all VMs

# Create a checkpoint for later use in Spark streaming
$ hdfs dfs -mkdir /checkpoint
```

*** Spark setup:** Do in 1 VM then distribute to other VMs.

- **Step 1:** Download and install Spark.

```
$ curl -O https://dlcdn.apache.org/spark/spark-3.5.3/spark-3.5.3-bin-hadoop3.tgz
$ tar -xzf spark-3.5.3-bin-hadoop3.tgz
$ mv spark-3.5.3-bin-hadoop3 /home/hadoop/spark
$ cd /home/hadoop/spark/conf/
```

- **Step 2:** Define all workers participate in Spark cluster.

```
# vim workers
node1
node2
node3
```

- **Step 3:** Setup Spark libraries in HDFS.

```
$ hdfs dfs -mkdir -p /share/jars
$ hdfs dfs -put /home/hadoop/spark/jars/* /share/jars/
```

- **Step 4:** Setup Spark default config by adding the following lines to *spark-defaults.conf*.

```
# Create file spark-defaults.conf
$ cp spark-defaults.conf.template spark-defaults.conf
```

```
# Open file spark-default.conf
spark.master yarn
spark.yarn.jars hdfs://node1:9000/share/jars/*
```

- **Step 5:** Distribute Spark installation to other hosts.

```
$ cd /home/hadoop
$ scp -r /home/hadoop/spark/ node2:/home/hadoop/
$ scp -r /home/hadoop/spark/ node3:/home/hadoop/
```

- **Step 6:** Start Spark.

```
# Fresh start by closing all running services
$ stop-all.sh
$ start-all.sh

# Verify Spark installation using Python or Scala
$ pyspark       # Python
$ spark-shell   # Scala
```

2. Spark Structured Streaming

Spark provides high-level programming API in Java, Scala, and Python, and an optimized engine that supports general execution graphs. As depicted in Figure 1, it supports a rich set of processing libraries including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming for handling real-time workload [2].

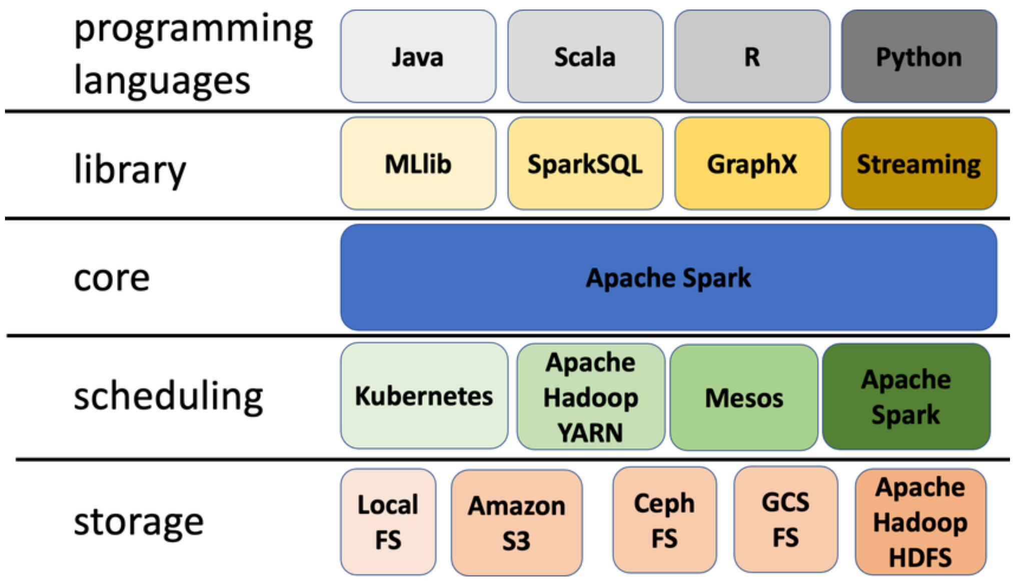

Figure 1: Apache Spark ecosystem.

In this course, we only focus on Spark SQL and Spark Structured Streaming. The following code demonstrates how to use Spark to cope with Kafka, which reads data from a specific topic and then writes it back to another Kafka topic.

```
# Start a pyspark shell
# Specify --num-executors with the number of executor
# Specify --executor-cores with the number of cores for each executor
# Specify --executor-memory with the memory allocating for each executor
$ pyspark --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.3
```

Create a topic name "spark-1" and "spark-2" before running these codes, start a consumer that subscribes to topic "spark-2", and a producer continuously produces messages to topic "spark-1".

```
df = spark.readStream.format("Kafka") \
  .option("kafka.bootstrap.servers", "<machine 1/2/3 IP>:9092") \
  .option("startingOffsets", "earliest") \
  .option("subscribe", "<topic name 1>").load()

# Chechpoint folder must be created before executing this code
# Using command: hdfs dfs -mkdir /checkpoint
query = df.writeStream.format("kafka") \
  .option("checkpointLocation", "hdfs://node1:9000/checkpoint") \
  .option("kafka.bootstrap.servers", "<machine 1/2/3 IP>:9092") \
  .option("topic", "<topic name 2>").start()
```

Like any other shell commands, PySpark shell provides several commands and options with –help flag [3]. On the other hand, Spark also provides a wide set of configurations to integrate with Kafka [4].

[2]https://downloads.apache.org/spark/docs/3.5.2/
[3]https://sparkbyexamples.com/pyspark/pyspark-shell-usage-with-examples/
[4]https://spark.apache.org/docs/3.5.2/structured-streaming-kafka-integration.html

3. Exercises

Students are required to use Spark (either in Python, Java, or Scala) to consume data from Kafka topic and then write it to HDFS in **"csv"** format. Additionally, students can take a look on how to customize Kafka serializer/deserializer using Python language. There is no submission required in this lab but students are encouraged to complete it as a premise for later exercises.

*** Note:** Yarn Resource Manager works well when using Spark with Hadoop ecosystem (HDFS in this case) but Spark can also run in standalone mode [5].

[5] `https://spark.apache.org/docs/latest/spark-standalone.html`