UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

The Spark Ecosystem

Antonio Caliò[1]

[1]DIMES Dept., University of Calabria
Rende (CS), IT
a.calio@unical.it
Big Data Analytics - Computer Engineering for the IoT

Outline

Presentation agenda

Introduction

Spark Key Concepts

RDDs Main Operations

Dockerize your environment

What is Spark

- Apache SparkTM is a fast and general-purpose engine for large-scale data processing
- Spark is designed to achieve the following goals:
  - Generality: diverse workloads, operators, job sizes
  - Low latency: sub-second
  - Fault tolerance: faults are the norm, not the exception
  - Simplicity: often comes from generality

Motivation

- Spark's main goal is to overcome one major limitation of the Map-Reduce approach:
  - The need for a lot of I/O operation.

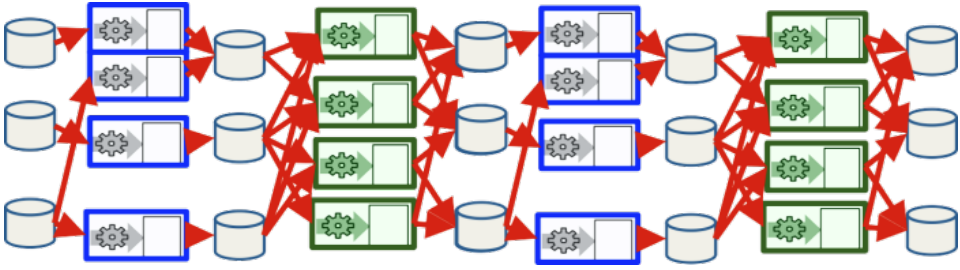


Figure: Map-Reduce

Motivation

- Spark's main goal is to overcome one major limitation of the Map-Reduce approach:
  - The need for a lot of I/O operation.

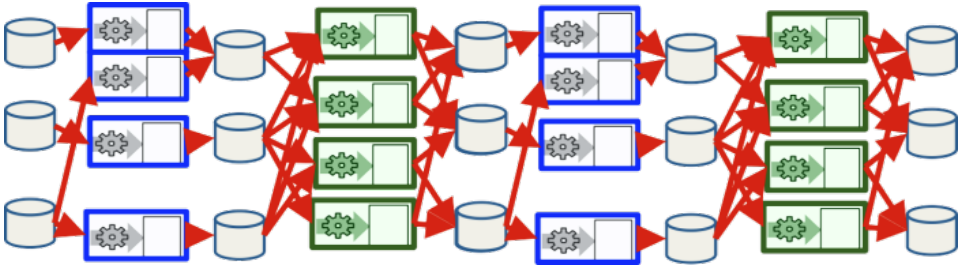


Figure: Map-Reduce

- Solution: Keep data in main memory

MapReduce vs Spark

- Iterative jobs

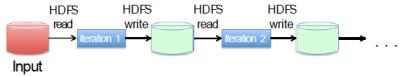
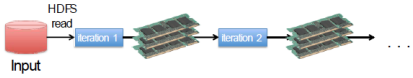
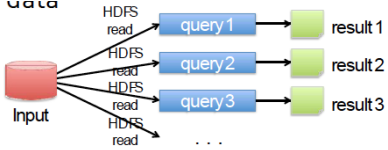


Figure: M/R

n#+caption: Spark



- Same data multiple jobs



Presentation agenda

Introduction

Spark Key Concepts

RDDs Main Operations

Dockerize your environment

Spark Components

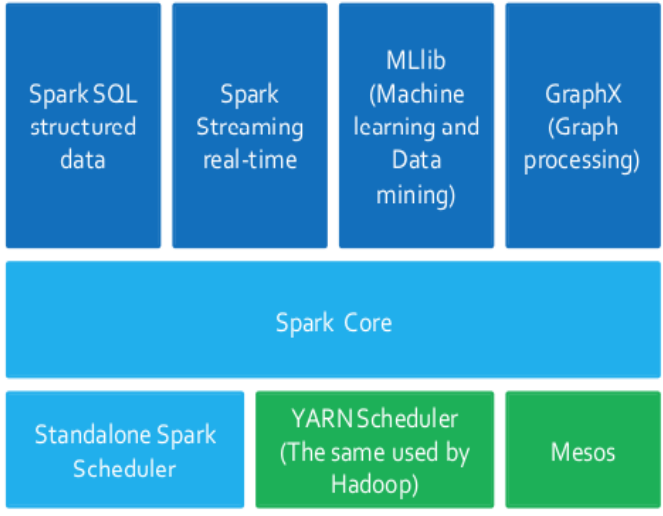


Figure: Spark overview

Spark Core

- It is the main building block that is exploited by all the high-level data analytics components
- It contains the basic functionalities of Spark exploited by all components
  - Task scheduling
  - Memory management
  - Fault recovery
- Provides the APIs that are used to create RDDs and applies transformations and actions upon them

Cluster Mode

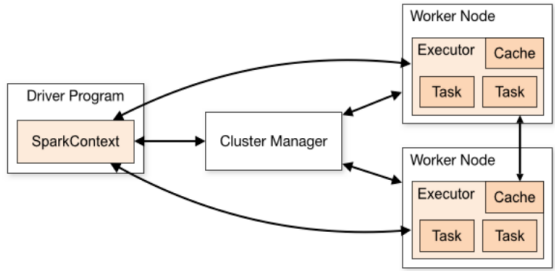


Figure: Cluster mode

- There are four possible options for the cluster manager:
  - Standalone (included in spark and used by default)
  - Apache Mesos
  - Hadoop YARN
  - Kubernets

Resilient Distributed Datasets (RDDs)

- Partitioned/Distributed collections of objects spread across the nodes of a clusters
- Stored in main memory (when it is possible) or on local disk
- Spark programs are written in terms of operations on resilient distributed data sets
- RDDs are built and manipulated through a set of parallel:
  - Transformations (e.g., map, filter, join)
  - Actions (e.g., count, collect, save)
  - RDDs are automatically rebuilt on machine failure
- They are split in partitions
  - Each node of the cluster that is running an application is assigned with at least one partition of some RDD
- RDDs are can be stored:
  - In the main memory of the executor node
  - In the local disk of the executor node
- RDD enables parallel and distributed computation on the node-level

Example

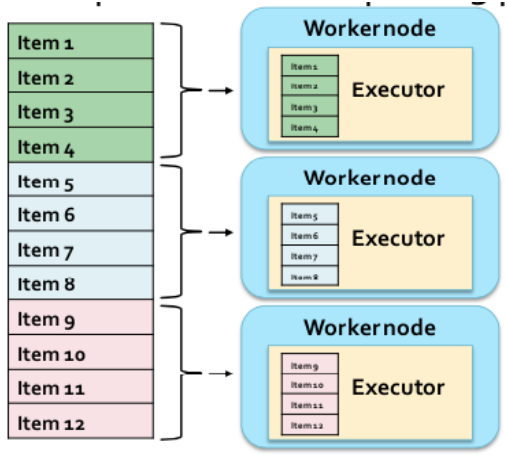


Figure: RDDs partition. Having more partitions implies having more parallelsm

RDDs Properties

- An RDD is *immutable*, i.e., its content cannot be modified
- An RDD can be created starting from:
  - a parallelized collection of objects
  - any file stored in a HDFS or in a regular file system
  - a database
  - another existing RDD

Spark program

- Spark programs are written in terms of operations on RDDs
  - Transformations
  - Actions
- Spark programs are responsible for:
  - Scheduling and synchronization of the jobs
  - Splitting of RDDs in partitions and allocation RDDs' partitions in the nodes of the cluster
  - Hiding complexities of fault-tolerance and slow machines
- Spark programs can be written in a variety of languages: Java, Python, Scala, R

The driver program

- It is basically the file containing the main method
- It defines the workflow of the application
- It defines the necessary RDDs
- It accesses the spark cluster through a *SparkContext* object
  - The driver program actually runs on the executors involved in the cluster
  - Each executor runs on its partition of the RDD

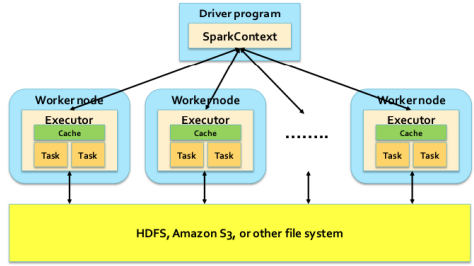


Figure: Driver program

Presentation agenda

Creation

- From a text file

  ```
  val logData = spark.read.textFile(logFile)
  ```

- From a collection

  ```
  spark.sparkContext.parallelize(<somelist>)
  ```

- From another RDD

  ```
  val nextRdd = rdd.<someRDDTransformation>((x, y) => x+y)
  ```

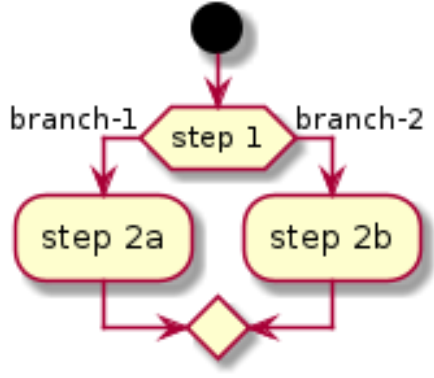
Storage

- Calling the cache function

```
val logData = spark.read.textFile(logFile).cache()
```

This function actually calls: `persist(SotrageLevel.MEMORY_AND_DISK)`

- Why caching? - If your spark program defines a similar DAG as the one in .. Then it makes sense to save the RDD generated after step 2 as it will feed two distinct branches of your application
- More generally, you should cache any RDD that will be used more than once



StorageLevel

- There are four different storage levels:
  - DISK_ONLY: Persist data on disk only in serialized format
  - MEMORY_ONLY: Persist data in memory only in deserialized format
  - MEMORY_AND_DISK: Persist data in memory and if enough memory is not available evicted blocks will be stored on disk
  - OFF_HEAP: Data is persisted in off-heap memory

Retrieval

- The content of an RDD can be retrieved from the nodes of the cluster and "stored" in a local variable of the Driver process.

```
val collectedVariable: Array[<RDD objects'type>] = RddVariable.collect()
```

- The `collect()` method returns all the elements in the RDD as a collection of objects
  - Be aware that if the size of the RDD is to large it may not fit inside a regular (not-distributed) variable

Transformations

- Operations on RDDs that return a new RDD
- Apply a transformation on the elements of the input RDD(s) and the result of the transformation is stored in a new RDD
  - Remember that RDDs are immutable, i.e., we cannot change the content of an already existing RDD
  - We can only apply a transformation on the content of an RDD and store/assign the result in/to a new RDD
- Transformations are computed lazily
  - i.e., transformations are computed only when an action is applied on the RDDs generated by the transformation operations
  - When a transformation is invoked Spark keeps only track of the dependency between the input RDD and the new RDD returned by the transformation The content of the new RDD is not computed

Different types of transformations

Two kinds of transformations:

- Some basic transformations analyze the content of one single RDD and return a new RDD
- Some other transformations analyze the content of two RDDs and return a new RDD

Examples of transformations are:

- Filter - it applies a boolan-returning function
- Map - it applies a function returning a new objects starting from the ones contained in the original RDD - 1-to-1 mapping
- FlatMap - it applies a function returning a new objects starting from the ones contained in the original RDD - 1-to-many mapping
- Distinct - it returns a new RDD with no duplicates
- Sample - it randomly extracts a fraction of the entire RDD
- Set Transformations: e.g., `union`, `interesection`, `subtract`

Actions

Operations that:

- Return results to the Driver program i.e., return local variables
  - Attention should be put on the size of the returned results because they must be stored in the main memory of the Driver program
- Or write the result in the storage (output file/folder)
  - the size of the result can be large in this case since it is directly stored in the (distributed) file system

Examples of actions are:

- `count()/countByValue()`
- `take()/takeSample()`
- `reduce()`
- `foreach()`
- `collect()`

The lineage Directed-Acyclic-Graph (DAG)

- It represents the dependencies between the RDD generated by the driver program.
- It is need in to compute the content of an RDD:
  - when an action is performed for the first time
  - when the application needs to recover from a failure
- Spark automatically optimizes the operations based on the this graph

- Example

```
...
val inputRDD = sc.textFile(<path>)
val errorsRDD = inputRDD.filter(<some function>)
val warningRDD = inputRDD.filter(<some function>)
val badRDD = errorsRDD.union(warningRDD)
val uniqueBadLinesRDD = badRDD.distinct()
```

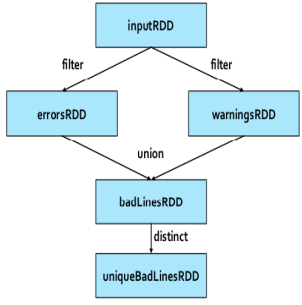


Figure: DAG corresponding to the listing

Presentation agenda

Introduction

Spark Key Concepts

RDDs Main Operations

Dockerize your environment

Requirements

- A working Docker environment on your machine – if your are on Fedora (>==32), you should use `podman` instead of Docker
- You should be familiar with the notions of:
  - Docker image
  - Docker container
  - Dockerfile

Dockerize your environment

- The following Dockerfile creates a working environment with everything you need: Scala, sbt and Spark

```
# This base image comes shipped with java 8 (needed for scala)
FROM openjdk:12-jdk-alpine

# Set env variables
ENV DAEMON_RUN=true
ENV SPARK_VERSION=3.0.1
ENV HADOOP_VERSION=2.7
ENV SCALA_VERSION=2.12.12
ENV SCALA_HOME=/usr/share/scala
ENV SPARK_HOME=/spark
ENV SBT_HOME=/sbt
ENV SPARK_OPTS --driver-java-options=-Xms1024M --driver-java-options=-Xmx4096M --driver-java-options=-Dlog4j.logLevel=info

# Add additional repo's for apk to use
RUN echo http://dl-cdn.alpinelinux.org/alpine/v3.3/main > /etc/apk/repositories; \
    echo http://dl-cdn.alpinelinux.org/alpine/v3.3/community  >> /etc/apk/repositories

# Update commands
RUN apk --update add wget tar bash coreutils procps openssl git

# Install Scala
RUN apk add --no-cache --virtual=.build-dependencies wget ca-certificates && \
    apk add --no-cache bash && \
    cd "/tmp" && \
    wget "https://downloads.typesafe.com/scala/${SCALA_VERSION}/scala-${SCALA_VERSION}.tgz" && \
    tar xzf "scala-${SCALA_VERSION}.tgz" && \
    mkdir "${SCALA_HOME}" && \
    rm "/tmp/scala-${SCALA_VERSION}/bin/"*.bat && \
    mv "/tmp/scala-${SCALA_VERSION}/bin" "/tmp/scala-${SCALA_VERSION}/lib" "${SCALA_HOME}" && \
    ln -s "${SCALA_HOME}/bin/"* "/usr/bin/" && \
    apk del .build-dependencies && \
    rm -rf "/tmp/"*

RUN export PATH="/usr/local/sbt/bin:$PATH" &&  apk update && apk add ca-certificates wget tar && mkdir -p "/usr/local/sbt"
```

Figure: First part

```
RUN cd "/tmp" && wget "https://github.com/sbt/sbt/releases/download/v1.3.4/sbt-1.3.4.tgz" && \
        tar xzf "sbt-1.3.4.tgz" && \
        mkdir "${SBT_HOME}" && \
        mv "/tmp/sbt/"* "${SBT_HOME}" && \
        ln -s "${SBT_HOME}/bin/"* "/usr/bin" && \
        rm -rf "/tmp/"*

# Get Apache Spark
RUN wget https://downloads.apache.org/spark/spark-${SPARK_VERSION}/spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz

# Install Spark and move it to the folder "/spark" and then add this location to the PATH env variable
RUN tar -xzf spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz && \
    mv spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION} /spark && \
    rm spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz && \
    ln -s "${SPARK_HOME}/bin/"* "/usr/bin"
```

Figure: Second part

Dockerize your application

- A very simple way to dockerize your application is to put the following file inside the root directory of your project

```
FROM <your-repository>/spark-base

RUN mkdir /app
COPY . /app
WORKDIR /app # this is the working directory inside your container
```

- Then within the same directory you build the container as follows:

```
docker image build -t <imageName> .
```

- Finally, you run the container as follows:

```
docker run --rm -it -v $(pwd):/app <imageName> spark-submit --class <MainCla
```