APACHE Spark

2023

APACHE SPARK CHEATSHEET

DISHA MUKHERJEE

CREDITS/SOURCE:-- https://www.zuar.com/blog/apache-spark-cheat-sheet/

Contents

Benefits of Using Apache Spark

Apache Spark is an open-source, Hadoop-compatible, cluster-computing platform that processes 'big data' with built-in modules for SQL, machine learning, streaming, and graph processing. The main benefits of using Apache Spark with your preferred API are...

- Computes data at blazing speeds by loading it across the distributed memory of a group of machines.
- Leveraging standard APIs like Java, Python, Scala or SQL to be more accessible.
- Allowing enterprises to leverage their existing infrastructures by being compatible with Hadoop v1 and 2.x.
- Easy to install and provides a convenient shell for learning the APIs.
- Improves productivity by focusing on content computation.

Commands for Initializing Apache Spark

These are the most common commands for initiating Apache Spark shell in either Scala or Python. These are essential commands you need when setting up the platform:

Initializing Spark Shell Using Scala

$ ./bin/spark-shell --master local[4]

Initializing SparkContext Using Scala

val conf = new SparkConf().setAppName(appName).setMaster(master)

new SparkContext(conf)

Initializing Spark Shell Using Python

$ ./bin/pyspark --master local[4]

Initializing SparkContext Using Python

from pyspark import SparkContext

sc = SparkContext (master = ‘local[2]’)

Configuring Spark Using Python

from pyspark import SparkConf, Spark Context

conf = (SparkConf()

```
.setMaster("local")
.setAppName("My app")
.set("spark.executor.memory", "1g"))
sc = SparkContext(conf = conf)
```

Apache Spark Set Operations

Here is a list of the most common set operations to generate a new Resilient Distributed Dataset (RDD). An RDD is a fault-tolerant collection of data elements that can be operated on in parallel. You can create an RDD by referencing a dataset in an external storage system, or by parallelizing a collection in your driver program.

Joining All Elements From the Argument and Source

union()

Intersecting All Elements From the Argument and Source

intersection()

Creating a Cross Product From the Argument and Source

cartesian()

Removing Data Elements in the Source

subtract()

Joining Data Elements to Create a New RDD

join(RDD,[numtask])

Converting to an Iterable

cogroup(RDD,[numTasks])

Piping Each Partition of an RDD Through a Shell Command

pipe(command, [envVars])

Apache SparkAction & Transformation Commands

Most RDD operations are either:

- Transformations: creating a new dataset from an existing dataset
- Actions: returning a value to the driver program from computing on the dataset

We'll cover the most common actions and transformation commands below. Although, you should note that syntax can vary depending on the API you are using, such as Python, Scala, or Java.

Common Apache Spark Actions

Here are the bread and butter actions when calling an RDD to retrieve specific data elements.

Counting Number of Data Elements in the RDD

count()

Collecting an Array of All the Data Elements

collect()

Aggregating the Data Elements

reduce(func)

Getting the First N Data Elements

take(n)

Executing the Function for Each Data Element

foreach(func)

Retrieving the First Data Element

first()

Common Apache Spark Transformations

Here are the main operations when you're calling a new RDD by applying a transformation function to the data elements.

Selecting Data Elements Based on a Function

filter(func)

Applying a Function to Each Data Element

map(func)

Applying a Function to Each Data Element to Return a Sequence

flatMap(func)

Applying a Function to Each Data Element Running Separately on Each Partition

mapPartitions(func)

Applying a Function to Each Data Element Running on an Indexed Partition

mapPartitionsWithIndex(func)

Sampling a Fraction of the Data

Sample(withReplacement, fraction, seed)

Applying a Function to Aggregate Values

reduceByKey(func,[numTasks])

Eliminating Duplicates

distinct([numTasks])

RDD Persistence Commands

One of the best features of Apache Spark is its ability to cache an RDD in cluster memory, speeding up the iterative computation. Here are the most commonly used commands for RDD persistence.

Storing RDD in Cluster Memory as Deserialized Java Objects at a Default Level

MEMORY_ONLY

Storing RDD in Cluster Memory or on the Disk as Deserialized Java Objects

MEMORY_AND_DISK

Storing RDD As Serialized Java Objects One Byte Array per Partition

MEMORY_ONLY_SER

Storing RDD Only on the Disk

DISC_ONLY

Spark SQL & Dataframe Commands

These are common integrated commands for using SQL with Apache Spark for working with structured data:

Integrating SQL queries with Apache Spark

Results = spark.sql(“SELECT * FROM tbl_name)

data_name = results.map(lambda p: col_name)

Connecting to Any Data Source

spark.read.json(“s3n://…”)

.registerTempTable(“json”)

results = spark.sql (“””SELECT * FROM tbl_name JOIN json …”””)