Introduction to Data Analytics

Spark

András Varga
IBM Consulting

Bratislava, 2022

Contents

Spark

Definition

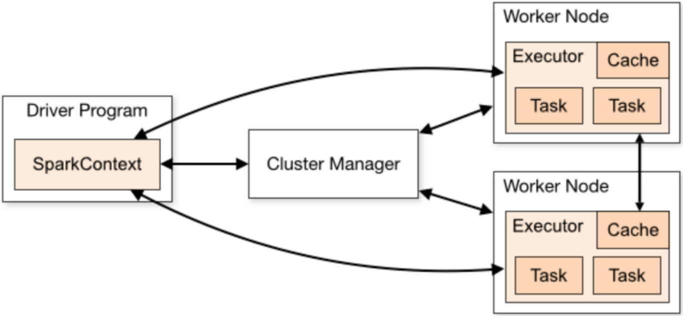
Apache Spark is a unified analytics engine for large-scale data processing. It can run in Hadoop clusters through YARN or Spark's standalone mode, and it can process data from various sources.

- Provides APIs for several languages: Java, Scala, Python, R (SparkR)
- Additionally supports SQL for certain operations
- MLlib - Spark Machine Learning library
- GraphX - Graph Processing library
- Stream analysis is supported

Spark Fundamentals

- Expands the MapReduce framework
- Speed is achieved by in memory operation
- Runs on top of YARN or Mesos, or in a stand-alone mode
- The Spark application is divided into a *Driver* Program and *Executors*
- Executors are running on Worker nodes and executing *Tasks*, i.e., units of work

Spark Application



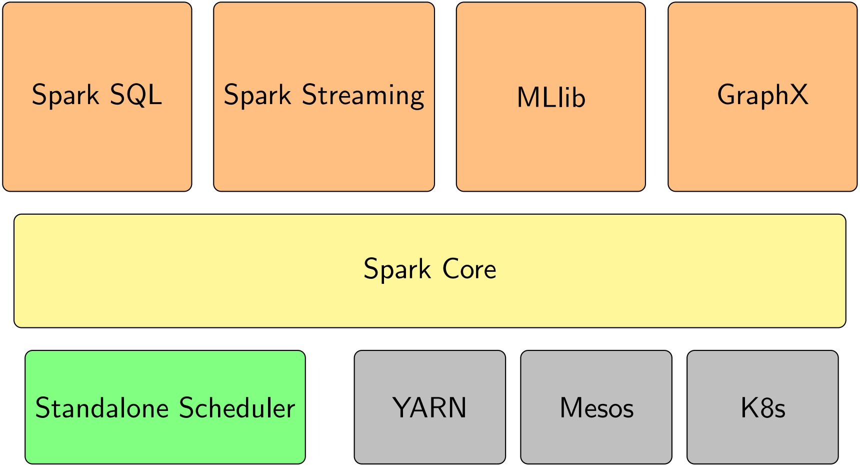
Installation

- Install Scala or Python (versions must match across the cluster)
- Download and unpack Spark
- Configure

Configuration

- Each application gets its own executor
- Each application runs in isolation, no sharing data between applications
- Three configuration options:
  - SparkConf object inside the application
  - Environment variables *conf/spark-env.sh*
  - Logging *log4j.properties*
- Configuration can be set using the SparkConf object or dynamically during spark-submit
- Or by using global defaults from *conf/spark-defaults.conf*

Spark Stack


Spark SQL
Spark Streaming
MLlib
GraphX
Spark Core
Standalone Scheduler
YARN
Mesos
K8s

Scala Basics

- Spark itself is written in Scala
- Scala is an OOP running on JVM
- Statically typed language
- Scala is a functional language
- Side effects, like JVM exceptions, are usually handled early and without breaking execution

Scala Class

```
class Study(name: String, val promotedParam: Int){
    println("New instance: "++name)
    val inMutableField: String = "This is immutable"
    var mutableField: String = "This is mutable"
}
new Study()
```

- Classes are instantiated via a constructor using "new" keyword
- Field is part of the class, visible to outside of the class
  - Mutable
  - Inmutable
- Scala provides type inference, but it is a good practice to not overusing it
- Constructor arguments are private, unless "promoted" using keywords

Methods

```
def echo(voice: String): String = voice
def addInt( a:Int, b:Int ): Int = {
    var sum:Int = a + b
    return sum
}
```

- Methods return at most one value, a type of which must be defined
- Methods can look like fields: **def** myValue: Int = 3
- Methods with one argument can be called using an infix notation, i.e., without the dots and parentheses: "Andras Varga" split " "

Arguments

- Default
  - Set a default value for an argument at definition time, in case it is not defined
    def echo(voice: String **= "Nothing"**): String = voice
    echo()
- Named
  - Names allow to omit the leading arguments with default values
    def addInt(a:Int = 0, b:Int): Int = a + b
    addInt(**b =** 5)

Objects

```
object MySingleton {
    def interesting: String = "This will never change"
}
MySingleton.interesting
```

- Provides a simple way to define singletons
- It is instantiated lazyly, but automatically during runtime
- Scala application is started by the main method being defined in any object:
  ```
  object MyApplication {
      def main(args: Array[String]): Unit = {
          println("Hello World!")
      }
  }
  ```
- Unit ≈ void

Accessibility of Fields and Methods

- Accessibility:
  - public (default)
  - private
  - protected
- An *object* and a *class* can share a name in the same source file as so called **companions**
- Companion class can access private fields and methods inside a companion object

Data Structures - *Collections*

- Array (fixed size)
  val numbers = **Array**(1, 2, 3, 4)
- List (can grow using *append* or *prepend*)
  val fruit: List[String] = **List**("apples", "oranges", "pears")
- Vector (immutable, indexed by hashing)
  val strings = **Vector**("one", "two")
- Set (no duplicates, no indices)
  val fruit = **Set**("apple", "orange", "peach", "banana")
- Tuple
  val values = (1,"2",3,"h")
  values._3 returns 2
- Map ("x" -> 24 is actually a pair = tuple of two elements)
  var mapping = **Map**("x" -> 24, "y" -> 25, "z" -> 26)
  mapping.*getOrElse*("v",16)

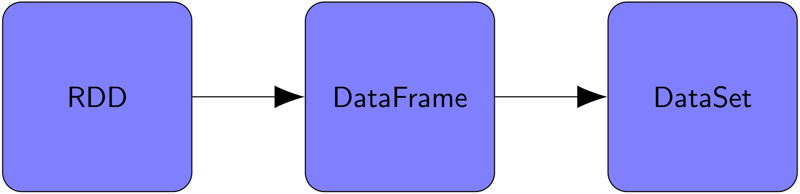
Higher Order Functions

- Higher order functions function takes another function as an argument
- Notable higher order functions:
  - map()
  - flatmap() - it also flattens one layer
  - filter()
  - foreach() - applies the function to the original collection
  - reduce(), foldLeft() or foldRight()
  - groupBy()

Code examples:

```
something.foreach(println)
mylist.map(x => x * x)
myCollection.flatmap(_ + 1)
```

Storing Data in Spark



Resilient Distributed Dataset (RDD)

- Sparks oldest data abstraction
- Distributed collection of data
- Immutable
- Operations:
  - Transformation - no return value, lazy evaluation
  - Actions - returns a value

Creating an RDD

- Parallelizing existing data from Spark (Driver)
  - S val data = Array(1, 2, 3, 4, 5)
    val distData = sc.parallelize(data)
  - P data = [1, 2, 3, 4, 5]
    distData = sc.parallelize(data)
- Referencing a Hadoop dataset (or s3 buckets, Cassandra)
  - S val distFile = sc.textFile("data.txt")
  - P distFile = sc.textFile("data.txt")
- Transforming an existing RDD to a new one

RDD Architecture

- RDD is partitioned
- when an RDD is created from another RDD (Or based on HDFS dataset) the partitioning is inherited
- It is better to distribute partitions evenly on the cluster, but shuffling is expensive

S someRDD.partitions.size

P someRDD.getNumPartitions()

- *partitionBy(numPartition, partitioningFunc)* changes partitions for RDD, causes shuffling

RDD Transformations I

- When an RDD is created an empty DAG is created
- Each transformation defined on the RDD is added to the DAG, but it is not performed
- Actions start the execution of the transformations from the DAG, consequently executing the action itself
- Transformation examples
  - map(func)
  - reduceByKey(func)
  - filter(func)
  - join(other dataset,[numTasks])

RDD Transformations II

- toDebugString() method returns the DAG for a given RDD
- The lazy behaviour supports fault tolerance - a node does not need to copy data to catch up after failure, only copies a DAG of transformations

RDD Actions

- Data is partitioned into blocks for Executors across the cluster
- Code is sent to data blocks to be executed
- Action example
  - collect() - returns all elements as an array to the driver, make sure dataset is small so driver can handle it
  - count()
  - first(), take(n)
  - foreach(func) - apply func on each element in a dataset

RDD Persistence I

Two functions: *persist*() and *cache*()

- persist() take an option of storage to use: MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, etc.
- cache() = persist(MEMORY_ONLY)
- Lazy evaluation
- When evaluated (per partition!) stores data to the storage
- Acts as a safe point for additional transformations or actions
- So intermediate data does not need to be re-created again

RDD Persistence II

- It is fault-tolerant, when data partition is lost a new Worker recreates lost data automatically
- There is an option to replicate the partitions in two cluster nodes
- When data partition does not fit into the storage it is recomputed on the fly
- Data can be serialized

Best Practices for Caching

- It is good idea to cache after preparation for downstream processing (e.g. filtering)
- When an cached RDD is no longer needed call *unpersist()* to free up memory
- calling the *count()* action on the RDD forces all partitions to be cached - call count separately
- Split data into equisized partitions

Shared Variables

- When a function passed to a Spark operation (e.g. *map*) is executed, it works on separate copies of all the variables
- Two types of shared variables are provided:
  - Broadcast Variables
    - A read-only variable cached on each machine rather than shipping a copy of it with tasks
    - S `val broadcastVar = sc.broadcast(Array(1, 2, 3))`
    - P `broadcastVar = sc.broadcast([1, 2, 3])`
  - Accumulators
    - Collect data from workers, through associative and commutative operations
    - Usually used as counters, numbers are natively supported by Spark
    - Read-only for the driver
    - S
      ```
      val accum = sc.longAccumulator("My Accumulator")
      mydata.foreach(x => accum.add(x))
      ```
    - P
      ```
      accum = sc.accumulator(0)
      mydata.foreach(lambda x: accum.add(x))
      ```

Variable Scope and Life-cycle

```
var counter = 0
rdd.foreach(x => counter += x)
println("Counter: " + counter)
```

- The *counter* sent to the executors is a **copy** and not the same as in the **driver**
- This is the use case for accumulators

SparkContext I

- The main entrypoint, represent a connection to a Spark cluster
- Usually named "sc"
- Created by loading libraries into the application
  - S
    ```
    import org.apache.spark.SparkContext
    import org.apache.spark.SparkConf
    val conf = new
    SparkConf().setAppName(appName).setMaster(master)
    new SparkContext(conf)
    ```
  - P
    ```
    from pyspark import SparkContext, SparkConf
    conf =
    SparkConf().setAppName(appName).setMaster(master)
    sc = SparkContext(conf=conf)
    ```

SparkContext II

- It is a good practice to don't hardcode the master information into the application, but to pass it as a parameter, simplifies releases

Passing Functions to Spark Using Scala

Code is sent to workers as functions

- Anonymous functions
  - S (x: Int, y: Int) => x + y
- Static methods in a global singleton object
  - S
```
object MyFunctions {
    def func1(s: String): String = { ... }
}
myRdd.map(MyFunctions.func1)
```
- Sending the reference of the object

Passing Functions to Spark Using Python

Code is sent to workers as functions

- Lamba expressions
    - lambda x, y : x + y
- Top-level functions in a module
- Local *defs* inside the function calling into Spark, for longer code
- When calling objects using reference the whole object is sent to Spark. To send smaller objects copy external variables to local variables:
    - ```
      def doStuff(self, rdd):
          field = self.field
          return rdd.map(lambda s: field + s)
      ```
      ```

Submitting Spark Applications

./bin/spark-submit Options:

- **class** - main class to start
- **master** - the master URL, if not specified in the code itself
- **deploy-mode** - cluster or client (runs locally, sometimes causes things to work, which would not work in the cluster)
- **conf** - additional configuration in a key-value pairs format
- **application-jar** (can be from HDFS or a local file) and *application arguments*
- And there are additional options controlling the application execution
- .bin/pyspark is a Python alternative

Submitting Spark Applications on Yarn - Example

./bin/spark-submit --class org.apache.spark.examples.SparkPi
--master **yarn**
--deploy-mode cluster
--*driver-memory* 4g
--*executor-memory* 2g
examples/jars/spark-examples*.jar
10

- In *cluster mode*, the driver application itself runs on YARN as well

Submitting Spark Applications Locally

- deploy-mode client is default
- In this case the amount of parallelism can be defined in the master parameter: *local*[K] - Start the application locally with K workers

P ./bin/pyspark −−master local[2]

Monitoring

- Spark provides a Web UI for monitoring (port 4040)
- Or by using external monitoring tools

Tuning

- Data serialization
  - Java - slower, flexible
  - Kyro - faster, but less types are supported
- Memory Tuning
  - Use primitives and arrays
  - Avoid nested structures
  - Analyze GC (SPARK_JAVA_OPTS)
- Level of parallelism (2-3 tasks per CPU core in the cluster)
- OutOfMemory error can be resolved by increased parallelism
- Broadcasting variables

DataFrame

- DataFrame is an immutable collection of rational data, i.e., organized into columns
- An SQLContext supports additional functionality
- It is partitioned and can be persisted as RDDs
- DataFrames can be loaded from several external sources, e.g. *spark.read.load(filename)*
- Or by adding schema to an existing RDD *sqlContext.createDataFrame(RDD,Schema)*

DataFrame Operations

DataFrames support many relational operations:

- select(colName)
- df.filter(condition)
- df.groupBy()
- df.printSchema() shows the shcema itself

Running SQL in Spark

- The simplest way to execute SQL in Spark is to use the sql() method of the SparkContext, returning a new FataFrame
- This requires a DataFrame to be registered as a local/global temporary view

S df.createOrReplaceTempView("people")
val sqlDF = spark.sql("SELECT * FROM people")

P df.create**Global**TempView("people")
spark.sql("SELECT * FROM **global_temp**.people")

Executing SQL Over Hive Tables

- ./bin/spark-sql
- Provides a CLI to execute SQL over a pre-defined Hive connection
- Spark can become the execution engine of Hive itself, speeding it up

Datasets

- Datasets are present only in Scala and Java
- It provides an abstraction for DataFrame (DataFrame become an alias for Dataset[Row])
- The advantage of Datasets is its ability to throw some of the analytical errors during compile time
- Datasets can hold semi-structured data, while DataFrames only relational data
- Backed by the Spark SQL's optimized execution engine
- Datasets have different internal encoding than RDDs, making them smaller in size for most data types

Local Data Types

- A local data type is stored on a single executor, it is not distributed
- Double typed values
- Vector
  - Dense - the usual representation of a vector
  - Sparse - represented by a binary search tree on indices
- LabeledPoint - a point with assigned label (name)

Matrices

- Local matrix representations:
  - Dense
  - Sparse - represented by three vectors:
    - values: [1.5, 2.2, 3.0, 5.0, 4.0, 1.0]
    - rowIndices: [0, 2, 0, 0, 1, 2]
    - colPointers: [0, 2, 3, 6] - which values (indices) represent the start of the new column
- Distributed Matrices:
  - RowMatrix - RDD of local vectors
  - IndexedRowMatrix - each row is named, so it is better for joins
  - CoordinateMatrix - sparse with huge possible dimensions

Machine Learning in Spark

Two libraries are provided, both providing the same functionality:

- MLlib
  - Older one
  - Using RDDs - RDD[LabeledPoint]
- Spark.ml
  - Newer one
  - Utilizes DataFrame and Dataset

Data transformation can be built into data pipelines for simpler maintenance

Simpler Functionaity

- Dataframe.describe() - computes statistics
- .stats() - additional statistics
- random split
- na methods - dropping or filling missing data
- dropDuplicates()
- *transformation()* and estimators (*fit()* functions)

ML Capabilities

- Classification
- Clustering
- Feature detection
- Evaluation
- Regression
- Outlier detection (Mahalanobis)
- Decision Trees and Random Forests
- ...

GraphX

- Dedicated to Graph computations
- **import** org.apache.spark.graphx.*
- The basic data structure is called *Property Graph*
  - It is distributed, immutable, fault-tolerant and can be persisted, similarly to RDDs
  - It is partitioned using vertex partitioning
  - When changing a graph substantial parts of the original graph is reused, reducing the cost

Property Graph

- Directed multigraph
- User defined objects attached to each vertex and edge
- Vertices are assigned an unique numeric ID
- VertexID is used to define edges
- Additionally EdgeTriplet[VD, ED] view is provided for the graph

```
class Graph[VD, ED] {
    val vertices: VertexRDD[VD]
    val edges: EdgeRDD[ED]
}
```

Graph Operations

- mapVertices, mapEdges and mapTriplets - changing the objects themselves, but not the graph structure
- collectNeighbors
- reverse
- subgraph
- joinVertices
- sendMsg and mergeMsg - graph based map/reduce

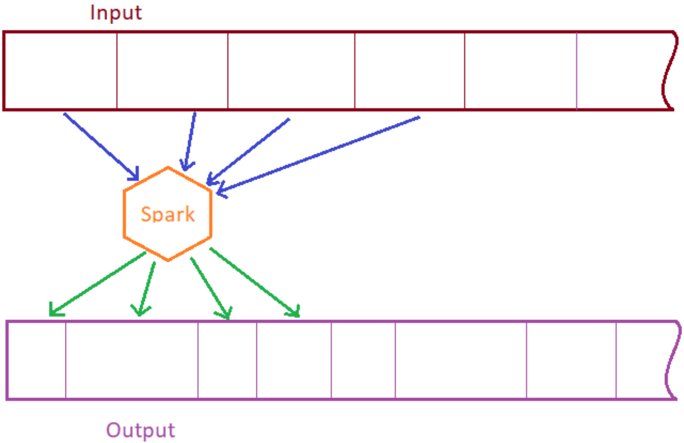
Building Graphs

- Several graph file format can be read from RDD or disk to build graphs
- Once the graph is built the edges are not repartitioned, so *partitionBy* must be called for efficient computations

Graph Algorithms

- PageRank
- Triangle Counting
- Connected Components
- Strongly Connected Components

Problem Statement - Stream Analysis



Spark Streaming

- The new version is called *Structured Streaming*
- Built on top of the Spark SQL engine
- Two processing models:
  - Micro-batch processing model
  - Continuous processing model
- Fully integrated with Kafka Topics

Scala Example

```scala
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SparkSession

val spark = SparkSession
    .builder
    .appName("StreamingTest")
    .getOrCreate()

import spark.implicits._

val lines = spark.readStream
    .format("socket")
    .option("host", "localhost")
    .option("port", 9999)
    .load()
```

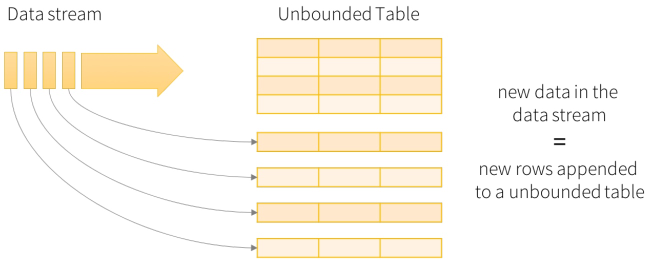
Python Example

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StreamingTest") \
    .getOrCreate()

lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()
```

Unbounded Table

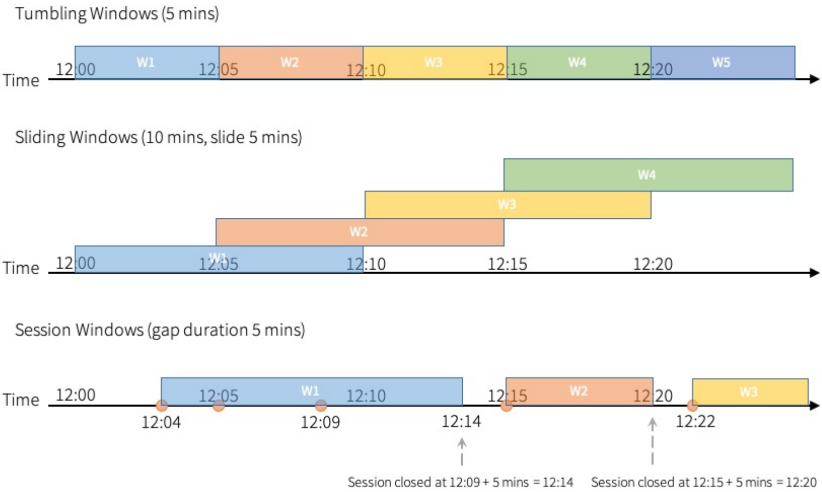


Data stream as an unbounded table

Incremental query execution on the unbounded input table, defined as standard query on a standard table.

Computation Windows I

How much of data should be aggregated?



Computation Windows II

- *Tumbling windows* are a series of fixed-sized, non-overlapping and contiguous time intervals
- *Sliding windows* are a series of fixed-sized, possibly overlapping time intervals, defined by their length and slide
- *Session window* has a dynamic length, depending on the input. A session window starts with an input, and expands itself if following input has been received within gap duration

Watermarking

- Handling "late" data
- m_1 sent before m_2 and m_3, but arrives after them
- To be able to properly aggregate late data the results of the aggregation must be kept in memory (and not written to output)
- Waiting indefinitely makes no sense
- Watermarking allows to identify late data by comparing event timestamps from data
- Threshold can be defined, data within the threshold will be aggregated, but data later than the threshold will start getting dropped

Examples

```
val sessionizedCounts = events
    .withWatermark("timestamp", "10 minutes")
    .groupBy(
        session_window($"timestamp", "5 minutes"),
        $"userId")
    .count()

sessionizedCounts = events \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        session_window(events.timestamp, "5 minutes"),
        events.userId) \
    .count()
```

References

- "Apache Spark Website," 2022.
- "Databricks Documentation and Glossary," 2022.
- "CognitiveClass.ai," 2022.
- "Scala official documentation," 2022.