**Spark the Definitive Guide**

Part I. Gentle Overview of Big Data and Spark

# Chapter 1

# What is Apache Spark?

Apache Spark is a unified computing engine and a set of libraries for parallel data processing on

computer clusters.

You can use Spark from Python, Java, Scala, R, or SQL. Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM).

Spark is often (mis)classified as a part of the “Hadoop Ecosystem,” in reality, Spark has little to do with Hadoop. Spark does natively support the Hadoop YARN cluster manager and Hadoop HDFS but it requires nothing from Hadoop itself (s.286).

|  |
| --- |
|  |

|  |
| --- |
| “Unified” kavrami, data analytics kapsamindaki islemlerin (data loading, SQL queries, ML, streaming comutation, vs) ayni “computing engine” uzerinde ve degismeyen bir API seti ile yapilabilmesine tekabul ediyor.  Spark’in temel farkliligi bu aslinda. Spark’tan once application yapabilmek icin, ayri ayri API’lar ile yapilarak elde edilenlerin birbirine ilistirilmesi gerekirken, Spark sayesinde tum adimlar tek bir ortamda yapilabiliyor.  Spark’in uzun sureli veri depolama gibi bir fonksiyonu yok. Odak noktasi, data’nin nerede olursa olsun alinip islenmesi. |

Context: The Big Data Problem

|  |
| --- |
| Tarihsel surecte islemcilerin hizlari surekli olarak artarak gelistirildi. Islemcilerin isinma problemi nedeniyle bu trend 2005 yilinda durdu ve tek bir islemcinin hizinin daha da artirilmasi yerine, paralel calisan CPU core’larinin bir arada kullanan islemciler gelistirilmeye baslandi. Bu kapasiteye artirmisti ama bu sefer cozulmesi gereken farkli alanlar ortaya cikmisti, gelistirilen bu kapasitenin kullanilmasi icin data’nin farkli core’lar uzerinde islenmesinin koordinasyonu ve kaynaklarin etkin kullanilmasi.  Spark’in devreye girdigi yer burasi, data’nin farkli CPU’lar uzerinde ayni anda ve ayni process icersinde islenmesi. Gunumuzde cozum yonelik gelistirilmelere hala ihtiyac duyuluyor. Zira hala temel sorun datanin bulunmasi degil, bunun islenmesi ile ilgili. |

# Chapter 2

# A Gentle Introduction to Spark

A cluster, or group, of computers, pools the resources of many machines together, giving us the ability to use all the cumulative resources as if they were a single computer.

Now, a group of machines alone is not powerful, you need a framework to coordinate work across them. Spark does just that, managing and coordinating the execution of tasks on data across a cluster of computers.

The cluster of machines that Spark will use to execute tasks is managed by a cluster manager like Spark’s standalone cluster manager, YARN, or Mesos. We then submit Spark Applications to these cluster managers, which will grant resources to our application so that we can complete our work.

**Spark Applications**

Spark Applications consist of a **driver process** and a set of **executor processes**.

The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things:

1. maintaining information about the Spark Application;
2. responding to a user’s program or input; and
3. analyzing, distributing, and scheduling work across the executors.

each executor is responsible for only two things:

1. executing code assigned to it by the driver, and
2. reporting the state of the computation on that executor back to the driver node.

|  |
| --- |
|  |

Here are the key points to understand about Spark Applications at this point:

* Spark employs a cluster manager that keeps track of the resources available.
* The driver process is responsible for executing the driver program’s commands across the executors to complete a given task.

**The SparkSession**

you control your Spark Application through a driver process called the SparkSession. The SparkSession instance is the way Spark executes user-defined manipulations across the cluster. There is a one-to-one correspondence between a SparkSession and a Spark Application.

spark == “SparkSession” instance == driver process

**DataFrames**

A DataFrame is the most common Structured API and simply represents a table of data with rows and columns. The list that defines the columns and the types within those columns is called the schema.

|  |
| --- |
| DataFrame, pandas’taki dataframe gibi dusunulebilir.  Shema, “metadata” diye de ifade edebilecegimiz data hakkindaki data. Hangi feature’lar hangi veri tipleri ile kullanilmis, onu gosterir. |

Spark has several **core abstractions** which all represent distributed collections of data:

1. Datasets,
2. DataFrames,
3. SQL Tables, and
4. Resilient Distributed Datasets (RDDs).

**Partitions**

To allow every executor to perform work in parallel, Spark breaks up the data into chunks called

partitions. A partition is a collection of rows that sit on one physical machine in your cluster.

|  |
| --- |
| Partitions: tek bir makina (executor) uzerinde islenmek uzere bolunmus data blogu. Bolunme satirlar uzerinden yapiliyor. |

**Transformations**

transformations are simply ways of specifying different series of data manipulation. In Spark, the core data structures are immutable. To “change” a DataFrame, you need to instruct Spark. These instructions are called transformations. Spark will not act on transformations until we call an action.

Transformations are the core of how you express your business logic using Spark. There are two

types of transformations:

* **narrow dependencies** (narrow transformations): each input partition will contribute to only one output partition. With narrow transformations, Spark will automatically perform an operation called **pipelining**, meaning that if we specify multiple filters on DataFrames, they’ll all be performed in-memory.
* **wide dependencies** (wide transformation): input partitions contributing to many output partitions. You will often hear this referred to as a **shuffle** whereby Spark will exchange partitions across the cluster. When we perform a shuffle, Spark writes the results to disk.

transformations == express your business logic == build up our logical transformation plan

|  |
| --- |
| Transformations, data manupulation komutlari ile data’nin donusturulmesinden ibarettir. Burada “shuffle” kavrami onemli, transformation, farkli clusterlar uzerinde islenen partition’lar arasinda bilgi alisverisini gerektiriyorsa, bu donusum “shuffle” olarak isimlendiriliyor. Haliyle bu iletisimi gerektirmeyen de “narrow olmus oluyor.  Cok basit bit ornek vermek gerekirse, bir degiskenin karakokunu alacaginiz zaman, bu herbir “value”nun kendisini kullanarak yapabileceginiz bir donusumdur ve diger cluster’lardaki satirlarda yer alan “value”lara ihtiyaciniz olmaz, dolayisiyla bu “narrow transformation”dir. Ancak ortalama almak isterseniz, bu islemin diger cluster’lara dagitilmis olan partition’lardaki degerlere de ihtiyaci vardir ve burada cluster’lar arasinda bir bilgi alisverisi gerekir, dolayisyla bu “wide transformation”a, yani “shuffle”a girer. |

**Lazy Evaluation**

Lazy evaluation means that Spark will wait until the very last moment to execute the graph of computation instructions. Spark compiles this plan from your raw DataFrame transformations to a streamlined physical plan.

|  |
| --- |
| Insan “lazy” kelimesini bilgisayar performansina kondurasi gelmiyor ama, terim olarak ifade ettigi sey, kullanicinin girdigi komutlarin aninda ve tek tek yapilmasi yerine, tum komutlarin bir butun olarak ele alip, fiziksel bir plan cikarildiktan sonra islenmeye baslanmasi anlamina geliyor. Bu islem performansi icin kritik bir husus. |

**Actions**

To trigger the computation, we run an action. An action instructs Spark to compute a result from a series of transformations.

There are three kinds of actions:

1. Actions to view data in the console
2. Actions to collect data to native objects in the respective language
3. Actions to write to output data sources

**Spark UI**

You can monitor the progress of a job through the Spark web UI. The Spark UI displays information on the state of your Spark jobs, its environment, and cluster state.

a Spark job represents a set of transformations triggered by an individual action, and you can monitor that job from the Spark UI.

**An End-to-End Example**

|  |  |
| --- | --- |
| flightData2015 = spark\  .read\  .option("inferSchema", "true")\  .option("header", "true")\  .csv("/data/flight-data/csv/2015-summary.csv") | Burada data’nin okunmasi islemi de bir data transformation’dir. Bu asamada spark, data’nin tamamini okumaz. “unspecified number of rows” okur ki, bu da datanin neye benzedigini anlamak icin. Transformation’larin akabinde kullanilacak olan action, okunacak satir sayisini netlestirmis olur. |
| flightData2015.take(3) | .take(), bir action’dir ve buraya kadar olan transformation’larin uygulanmasini saglar. |
| flightData2015.sort("count").explain() | sort(), bir transformation’dir.  .explain(), bizim “query”mizin spark tarafindan nasil “execute” edileceginin planini gosterir. (“lineage”) |
| spark.conf.set("spark.sql.shuffle.partitions", "5") | “.conf.set” ile partition sayisini belirleyebiliriz. Default 200’dur. |

the heart of Spark’s programming model—functional programming where the same inputs always result in the same outputs when the transformations on that data stay constant.

We do not manipulate the physical data; instead, we configure physical execution characteristics

through things like the shuffle partitions parameter.

DataFrames and SQL

Spark can run the same transformations, regardless of the language, in the exact same way. You can express your business logic in SQL or DataFrames (either in R, Python, Scala, or Java) and Spark will compile that logic down to an underlying plan before actually executing your code.

|  |
| --- |
| Spark, business logic’inizi, SQL ya da DatFrame ile; Python ya da Scala kullanarak yamis olmaniz arasinda hicbir fark gozetmez ve onlari ayni uygulama planina indirger. SQL ve DataFrame ile ifade edilmesi arasinda bir performans farki da olmaz. Her ikisi icin de halihazirda yuzlerce “built-in” function’lar var manupulation icin. |

|  |  |
| --- | --- |
| **SQL syntax** | **DataFrame syntax** |
| sqlWay = spark.sql("""  SELECT DEST\_COUNTRY\_NAME, count(1)  FROM flight\_data\_2015  GROUP BY DEST\_COUNTRY\_NAME  """) | dataFrameWay = flightData2015\  .groupBy("DEST\_COUNTRY\_NAME")\  .count() |
| maxSql = spark.sql("""  SELECT DEST\_COUNTRY\_NAME, sum(count) as destination\_total  FROM flight\_data\_2015  GROUP BY DEST\_COUNTRY\_NAME  ORDER BY sum(count) DESC  LIMIT 5  """)  maxSql.show() | from pyspark.sql.functions import desc  flightData2015\  .groupBy("DEST\_COUNTRY\_NAME")\  .sum("count")\  .withColumnRenamed("sum(count)", "destination\_total")\  .sort(desc("destination\_total"))\  .limit(5)\  .show() |

This execution plan is a directed acyclic graph (DAG) of transformations, each resulting in a

new immutable DataFrame, on which we call an action to generate a result.

|  |
| --- |
|  |

# Chapter 3

# A Tour of Spark’s Toolset

Spark is composed of these primitives—

* the lower-level APIs
* Structured APIs
* series of standard libraries for additional functionality.

Spark’s libraries support a variety of different tasks, from

* graph analysis and
* machine learning to
* streaming and
* integrations with a host of computing and storage systems.

**Running Production Applications**

spark-submit does one thing: it lets you send your application code to a cluster and launch it to

execute there. Upon submission, the application will run until it exits (completes the task) or encounters an error. You can do this with all of Spark’s support **cluster managers** including **Standalone**, **Mesos**, and **YARN**.

**Structured Streaming**

With Structured Streaming, which is a high-level API for stream processing, you can take the same operations that you perform in batch mode using Spark’s structured APIs and run them in a streaming fashion. This can reduce latency and allow for incremental processing.

|  |
| --- |
| “structured streaming”, kod degisikligine gerek kalmadan (sanirim, “structured” ifadesi, ayni yapinin kullanilmasina vurgu yapiyor), surekli olarak akan yeni bilginin islenebilmesini saglayan API. Bunu da diyor “batch job” (islenecek bilgi miktarinin belirli oldugu) icin tasarladigin logic uzerinde kucuk bir degisiklik yaparak donusturebilirsin diyor. |

|  |  |
| --- | --- |
| **Static code** |  |
| staticDataFrame = spark.read.format("csv")\  .option("header", "true")\  .option("inferSchema", "true")\  .load("/data/retail-data/by-day/\*.csv")  staticDataFrame.createOrReplaceTempView("retail\_data")  staticSchema = staticDataFrame.schema | - dosyanin okunmasi  - TempView olusturulmasi  - Schema’nin bir degiskene atanmasi |
| from pyspark.sql.functions import window, column, desc, col  staticDataFrame\  .selectExpr(  "CustomerId",  "(UnitPrice \* Quantity) as total\_cost",  "InvoiceDate")\  .groupBy(  col("CustomerId"), window(col("InvoiceDate"), "1 day"))\  .sum("total\_cost")\  .show(5) | query:  - Bir gunluk zaman dilimleri (window) icersinde, musteriler tarafindan yapilan alisverislerin toplam bedelini verecek olan query. |
|  | - Ekran ciktisi |

|  |  |
| --- | --- |
| **Streaming code** |  |
| streamingDataFrame = spark.readStream\  .schema(staticSchema)\  .option("maxFilesPerTrigger", 1)\  .format("csv")\  .option("header", "true")\  .load("/data/retail-data/by-day/\*.csv") | Farkliliklar:  - “read” yerine “**readStream**”  - “**maxFilesPerTrigger**” option: bir seferde okunacak dosya sayisini belirtmek icin.  - “**schema**” option: daha once degiskene atanmis semayi kullanmak icin |
| purchaseByCustomerPerHour = streamingDataFrame\  .selectExpr(  "CustomerId",  "(UnitPrice \* Quantity) as total\_cost",  "InvoiceDate")\  .groupBy(  col("CustomerId"), window(col("InvoiceDate"), "1 day"))\  .sum("total\_cost")  \*\*\*\*\*\*\*  purchaseByCustomerPerHour.writeStream\  .format("memory")\ // memory = store in-memory table  .queryName("customer\_purchases")\ // the name of the in-memory table  .outputMode("complete")\ // complete = all the counts should be in the table  .start() | Farkliliklar:  - “query” kismi, static code’taki “show()”a kadar olan bolum ile ayni. Ancak action bolumu farkli cunku islem tek seferlik degil. Yani bir result bulup orda birakmasini istemiyorum.  - kodun ikinci bolumu aslinda bir “action” blogu gibi.  - “query” icin memory’de yeni bir table olusturuyor ve her “trigger”da bu table’I guncelliyor.  - Stream baslatildiktan sonra, yeni bir query ile sorgulama yapabiliriz.  spark.sql("""  SELECT \*  FROM customer\_purchases  ORDER BY `sum(total\_cost)` DESC  """)\  .show(5) |

Machine Learning and Advanced Analytics

MLlib is a built-in library of machine learning algorithms. MLlib allows for preprocessing, munging, training of models, and making predictions at scale on data. You can even use models trained in MLlib to make predictions in Strucutred Streaming.

|  |  |
| --- | --- |
| from pyspark.sql.functions import date\_format, col  preppedDataFrame = staticDataFrame\  .na.fill(0)\  .withColumn("day\_of\_week", date\_format(col("InvoiceDate"), "EEEE"))\  .coalesce(5) | Filling missing values  Creating a new feature from “date” feature |
| trainDataFrame = preppedDataFrame\  .where("InvoiceDate < '2011-07-01'")  testDataFrame = preppedDataFrame\  .where("InvoiceDate >= '2011-07-01'") | Train-test split |
| from pyspark.ml.feature import StringIndexer  indexer = StringIndexer()\  .setInputCol("day\_of\_week")\  .setOutputCol("day\_of\_week\_index") | Indexer - Label encoding (of “day\_of\_week”) |
| from pyspark.ml.feature import OneHotEncoder  encoder = OneHotEncoder()\  .setInputCol("day\_of\_week\_index")\  .setOutputCol("day\_of\_week\_encoded") | Encoder – OneHot encoding (of label encoded feature) |
| from pyspark.ml.feature import VectorAssembler  vectorAssembler = VectorAssembler()\  .setInputCols([“UnitPrice”, “Quantity”, “day\_of\_week\_encoded”])\  .setOutputCol(“features”) | Vector assembler - X ve y’nin tanimlanmasi |
| from pyspark.ml import Pipeline  transformationPipeline = Pipeline()\  .setStages([indexer, encoder, vectorAssembler]) | Creating pipeline |
| fittedPipeline = transformationPipeline.fit(trainDataFrame)  transformedTraining = fittedPipeline.transform(trainDataFrame) | Fitting and Transforming Pipeline |
| from pyspark.ml.clustering import KMeans  kmeans = KMeans()\  .setK(20)\  .setSeed(1L) | Instantiate/initialize the model |
| kmModel = kmeans.fit(transformedTraining) | Train the model |
| kmModel.computeCost(transformedTraining) | Evaluation |

Part II. Structured APIs—DataFrames, SQL, and Datasets

# Chapter 4. Structured API Overview

The Structured APIs are a tool for manipulating all sorts of data, from unstructured log files to semi-structured CSV files and highly structured Parquet files. These APIs refer to three core types of distributed collection APIs:

1. Datasets
2. DataFrames
3. SQL tables and views

Although they are distinct parts of the book, the majority of the Structured APIs apply to both batch and streaming computation.

**DataFrames and Datasets**

Spark has **two notions of structured collections**: DataFrames and Datasets.

* DataFrames and Datasets are (distributed) table-like collections with well-defined rows and columns.
* Each column must have the same number of rows as all the other columns (you can use null)
* each column has type information that must be consistent for every row.
* DataFrames and Datasets represent immutable, lazily evaluated plans that specify what operations to apply to data residing at a location to generate some output.
* Tables and views are basically the same thing as DataFrames. We just execute SQL against them instead of DataFrame code.

|  |
| --- |
| Iki onemli nokta;  SQL tables, ayri bir “structured collection” degil aslinda diyor, sadece ayni data (“table-like collections”) uzerinde SQL kodu kullanarak elde edilmis oldugu icin “SQL table” diyoruz hepsi bu.  Ikincisi, dataframe ve dataset’ler aslinda planlari temsil eder diyor, fiziksel kayitli data’yi degil. Kayitli degil derken database’de kayitli degil, disk’e kayitli. Ancak “temporary view”ler disk’e de kayitli degil ve sadece “Spark session” boyunca, sadece olusturuldugu notebook’ta ve olusturan kullanici tarafindan kullanilabilir. |

**Overview of Structured Spark Types**

Spark is effectively a programming language of its own. Internally, Spark uses an engine called Catalyst that maintains its own type information through the planning and processing of work. Spark types map directly to the different language APIs that Spark maintains and there exists a lookup table for each of these in Scala, Java, Python, SQL, and R.

|  |
| --- |
| Spark’in “planning and processing” boyunca kullandigi bilgi tipleri sabittir aslinda. Biz hangi dili kullanirsak kullanalim, Spark bunu kendi standart bilgi tiplerini iceren “lookup table”indaki bilgiler ile map eder. Bu bilgilerin standart hale donusturulmesi islemi “Catalyst” engine tarafindan gerceklestirilir. |

**DataFrames Versus Datasets**

In essence, within the Structured APIs, there are two more APIs, the **“untyped” DataFrames** and the **“typed” Datasets**. To say that DataFrames are untyped is aslightly inaccurate; they have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at runtime. Datasets, on the other hand, check whether types conform to the specification at compile time. Datasets are only available to Java Virtual Machine (JVM)–based languages (Scala and Java) and we specify types with case classes or Java beans.

To Spark (in Scala), DataFrames are simply Datasets of Type Row. The “Row” type is Spark’s internal representation of its optimized inmemory format for computation. To Spark (in Python or R), there is no such thing as a Dataset: everything is a DataFrame and therefore we always operate on that optimized format.

|  |
| --- |
| Burada, iki tane daha “structured API” var demis ama isimleri disinda diger tiplerle farkliliklarini ortaya koymamis. **“untyped” DataFrames’i, Spark, runtime’da (yani bilginin islenmesi sirasinda) DataFrame’deki bilgi tiplerini check ediyor. Yani user, DataFrame’e ait bilgi tiplerini bir option ile input olarak tanitmadi ise, Spark, runtime’a kadar bilgi tiplerinden** |