**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 sorunlar 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 olan ihtiyac devam ediyor. 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 islem icin 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 yapmis olmamiz 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** |

**Columns**

Represent a “simple type” like an integer or string, etc.

**Rows**

A row is nothing more than a record of data. Each record in a DataFrame must be of type Row, as we can see when we collect the following DataFrames. We can create these rows manually from SQL, from Resilient Distributed Datasets (RDDs), from data sources, or manually from scratch. Here, we create one by using a range:

|  |  |
| --- | --- |
| spark.range(2).collect() | This results in an array of Row objects. |

**Overview of Structured API Execution**

the execution of a single structured API query from user code to executed code. Here’s

an overview of the steps:

1. Write DataFrame/Dataset/SQL Code.

2. If valid code, Spark converts this to a Logical Plan.

3. Spark transforms this Logical Plan to a Physical Plan, checking for optimizations along the way.

4. Spark then executes this Physical Plan (RDD manipulations) on the cluster.

To execute code, we must write code. This code is then submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user.

|  |
| --- |
|  |

**Logical Plan**

Spark uses the **catalog**, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer. **The analyzer** might reject the unresolved logical plan if the required table or column name does not exist in the catalog. **Catalyst Optimizer**, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections.

**Physical Planning**

The physical plan, often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model.

Physical planning results in a series of RDDs and transformations. This result is why you might have heard Spark referred to as a compiler—it takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations for you.

**Execution**

Upon selecting a physical plan, Spark runs all of this code over RDDs, the lower-level programming interface of Spark (which we cover in Part III). Spark performs further optimizations at runtime, generating native Java bytecode that can remove entire tasks or stages during execution. Finally the result is returned to the user.

# Chapter 5. Basic Structured Operations

**DataFrame** consists of a series of records (like rows in a table), that are of type Row, and a number of columns (like columns in a spreadsheet) that represent a computation expression that can be performed on each individual record in the Dataset.

**Schemas** define the name as well as the type of data in each column.

**Partitioning** of the DataFrame defines the layout of the DataFrame or Dataset’s physical distribution across the cluster. The **partitioning scheme** defines how that is allocated.

We can either let a data source define the schema (called schema-on-read) or we can define it explicitly ourselves.

|  |  |  |
| --- | --- | --- |
| Creating DataFrame | df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json") |  |
| look at the schema | df.printSchema() |  |
|  | spark.read.format("json").load("/data/flight-data/json/2015-summary.json").schema  StructType(List(StructField(DEST\_COUNTRY\_NAME,StringType,true),  StructField(ORIGIN\_COUNTRY\_NAME,StringType,true),  StructField(count,LongType,true))) |  |
| create and enforce a  specific schema on a DataFrame | from pyspark.sql.types import StructField, StructType, StringType, LongType  myManualSchema = StructType([  StructField("DEST\_COUNTRY\_NAME", StringType(), True),  StructField("ORIGIN\_COUNTRY\_NAME", StringType(), True),  StructField("count", LongType(), False, metadata={"hello":"world"})  ])  df = spark.read.format("json").schema(myManualSchema)\  .load("/data/flight-data/json/2015-summary.json") | “shema”yi kendimiz belirleyerek, okunan datanin buna gore okunmasini saglayabiliriz. Tabi data’nin da buna uygun olmasi lazim. |

**Columns and Expressions**

You can select, manipulate, and remove columns from DataFrames and these operations are represented as expressions.

You cannot manipulate an individual column outside the context of a DataFrame; you must use Spark transformations within a DataFrame to modify the contents of a column.

**Columns**

the two simplest ways to construct are by using the col or column functions:

|  |  |  |
| --- | --- | --- |
| Construct column | from pyspark.sql.functions import col, column  col("someColumnName")  column("someColumnName") |  |
| Explicit column references | df.col("count") | you can use the col method on the specific DataFrame.  Spark does not need to resolve this column itself (during the analyzer phase) because we did that for Spark: |

**Expressions**

columns are expressions, but what is an expression? An expression is a set of transformations on one or more values in a record in a DataFrame. Think of it like a function that takes as input one or more column names, resolves them, and then potentially applies more expressions to create a single value for each record in the dataset.

In the simplest case, an expression, created via the expr function, is just a DataFrame column reference.

expr("someCol") == col("someCol").

**Columns as expressions**

Columns provide a subset of expression functionality. If you use col() and want to perform transformations on that column, you must perform those on that column reference. When using an expression, the expr function can actually parse transformations and column references from a string and can subsequently be passed into further transformations. expr("someCol - 5") == col("someCol") – 5.

**Records and Rows**

In Spark, each row in a DataFrame is a single record. Spark represents this record as an object of type Row. Row objects internally represent arrays of bytes.

**select and selectExpr**

select and selectExpr allow you to do the DataFrame equivalent of SQL queries on a table of data.

expr is the most flexible reference that we can use. It can refer to a plain column or a string manipulation of a column.

|  |  |  |
| --- | --- | --- |
| (Directed Acyclic Graph) DAG | in Apache Spark is a set of Vertices and Edges, where vertices represent the RDDs and the edges represent the Operation to be applied on RDD. |  |
|  | from pyspark.sql.functions import expr  expr("(((someCol + 5) \* 200) - 6) < otherCol") | This code equivalently represents the above graph (directed acyclic graph). |
| Accessing a DataFrame’s columns | spark.read.format("json").load("/data/flight-data/json/2015-summary.json").columns | you can use the columns  property to see all columns on a DataFrame |
| See a row | df.first() |  |
| Creating rows | from pyspark.sql import Row  myRow = Row("Hello", None, 1, False) | Row’larin “schema”si yoktur. Girilen degerler DataFrame’in “schema”sina uygun olmalidir. |
| Accessing data in rows | myRow[0]  myRow[2] |  |
| create DataFrames from raw data sources | df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json") |  |
| register DataFrame as a temporary view | df.createOrReplaceTempView("dfTable") |  |
| create DataFrames on the fly | from pyspark.sql import Row  from pyspark.sql.types import StructField, StructType, StringType, LongType  myManualSchema = StructType([  StructField("some", StringType(), True),  StructField("col", StringType(), True),  StructField("names", LongType(), False)  ])  myRow = Row("Hello", None, 1)  myDf = spark.createDataFrame([myRow], myManualSchema)  myDf.show() |  |
|  | df.select("DEST\_COUNTRY\_NAME", "ORIGIN\_COUNTRY\_NAME").show(2) |  |
|  | from pyspark.sql.functions import expr, col, column  df.select(  expr("DEST\_COUNTRY\_NAME"),  col("DEST\_COUNTRY\_NAME"),  column("DEST\_COUNTRY\_NAME"))\  .show(2) | you can refer to columns in a number of different ways |
| “AS” keyword | df.select(expr("DEST\_COUNTRY\_NAME AS destination")).show(2) |  |
| “alias” method | df.select(expr("DEST\_COUNTRY\_NAME as destination").alias("DEST\_COUNTRY\_NAME"))\  .show(2) |  |
|  | df.selectExpr("DEST\_COUNTRY\_NAME as newColumnName", "DEST\_COUNTRY\_NAME").show(2) | selectExpr is probably the most convenient interface |
| a simple example that adds a  new column withinCountry to our DataFrame | df.selectExpr(  "\*", # all original columns  "(DEST\_COUNTRY\_NAME = ORIGIN\_COUNTRY\_NAME) as withinCountry")\  .show(2)  SELECT \*, (DEST\_COUNTRY\_NAME = ORIGIN\_COUNTRY\_NAME) as withinCountry  FROM dfTable  LIMIT 2 | This opens up the true power of Spark. We can treat selectExpr as a simple way to build up  complex expressions that create new DataFrames. In fact, we can add any valid non-aggregating SQL  statement, and as long as the columns resolve, it will be valid! |
| we can also specify aggregations over the entire DataFrame | df.selectExpr("avg(count)", "count(distinct(DEST\_COUNTRY\_NAME))").show(2)  SELECT avg(count), count(distinct(DEST\_COUNTRY\_NAME)) FROM dfTable LIMIT 2 |  |
| Converting to Spark Types (Literals) | from pyspark.sql.functions import lit  df.select(expr("\*"), lit(1).alias("One")).show(2)  SELECT \*, 1 as One FROM dfTable LIMIT 2 | Sometimes, we need to pass explicit values into Spark that are just a value (rather than a new column). The way we do this is through literals. Literals are expressions |
| **Adding Columns**  “withColumn” method. | .withColumn(<column\_name>, <expression>)  df.withColumn("numberOne", lit(1)).show(2)  -- in SQL  SELECT \*, 1 as numberOne FROM dfTable LIMIT 2 |  |
| **Renaming Columns**  “withColumnRenamed” method. | df.withColumnRenamed("DEST\_COUNTRY\_NAME", "dest").columns |  |
| **Reserved Characters** and Keywords  “backtick  (`)” characters. | dfWithLongColName = df.withColumn(  "This Long Column-Name",  expr("ORIGIN\_COUNTRY\_NAME")) | Burada “WithColumn”in ilk argument’i string olarak algilandigi icin “backtick’e gerek yok. |
|  | dfWithLongColName.selectExpr(  "`This Long Column-Name`",  "`This Long Column-Name` as `new col`")\  .show(2)  dfWithLongColName.select(expr("`This Long Column-Name`")).columns | “selectExpr”, cift tirnak icersindeki bir seri ifadeyi ayri ayri algilar, yani bosluklarla ayrilmis her bir ifadenin kendi basina anlamli olmasi lazim. Dolayisi ile cift tirnak icini bir “string” butunu olarak degil expression olarak algilar. Bu ifadenin icersinde, arasinda bosluk olan bir seri ifadeyi bir string butunu olarak algilamasini istiyorsak “backtick” character kullanilmasi gerekir. |
| **Case Sensitivity**  By default Spark is case insensitive. | set spark.sql.caseSensitive true | however, you can make Spark case sensitive by setting the  configuration |
| **Removing Columns**  “drop” method | df.drop("ORIGIN\_COUNTRY\_NAME").columns |  |
| Changing a Column’s Type (cast) | df.withColumn("count2", col("count").cast("long")) |  |
| **Filtering Rows**  where or filter methods | df.filter(col("count") < 2).show(2)  df.where("count < 2").show(2)  -- in SQL  SELECT \* FROM dfTable WHERE count < 2 LIMIT 2 |  |
| multiple AND filters | df.where(col("count") < 2).where(col("ORIGIN\_COUNTRY\_NAME") != "Croatia")\  .show(2)  -- in SQL  SELECT \* FROM dfTable WHERE count < 2 AND ORIGIN\_COUNTRY\_NAME != "Croatia"  LIMIT 2 |  |
| Getting unique rows | df.select("ORIGIN\_COUNTRY\_NAME", "DEST\_COUNTRY\_NAME").distinct().count()  -- in SQL  SELECT COUNT(DISTINCT(ORIGIN\_COUNTRY\_NAME, DEST\_COUNTRY\_NAME)) FROM dfTable |  |
| Random Samples | seed = 5  withReplacement = False  fraction = 0.5  df.sample(withReplacement, fraction, seed).count() |  |
| Random splits | dataFrames = df.randomSplit([0.25, 0.75], seed)  dataFrames[0].count() > dataFrames[1].count() # False | Machine learning icin de kullanilabilir. |
| Concatenating and Appending Rows (Union) | from pyspark.sql import Row  **schema** = df.schema  newRows = [  Row("New Country", "Other Country", 5L),  Row("New Country 2", "Other Country 3", 1L)  ]  **parallelizedRows** = spark.sparkContext.parallelize(newRows)  **newDF** = spark.createDataFrame(**parallelizedRows**, **schema**)  df.union(**newDF**)\  .where("count = 1")\  .where(col("ORIGIN\_COUNTRY\_NAME") != "United States")\  .show() | * Birlestirilecek “df”in schema’si yeni dataFrame’de kullanilmak uzere bir degiskene atandi * Yeni “record”lar olusturuldu * Satirlar, “parallelize” edildi. * Parallelized satirlar, schema degiskeni de kullanilarak yeni bir dataframe donusturuldu. * Df’ler union ile birlestirildi. |
| Sorting Rows  Sort and orderBy | df.sort("count").show(5)  df.orderBy("count", "DEST\_COUNTRY\_NAME").show(5)  df.orderBy(col("count"), col("DEST\_COUNTRY\_NAME")).show(5) | The default is to sort in ascending order |
| Specify the sort order | from pyspark.sql.functions import desc, asc  df.orderBy(expr("count desc")).show(2)  df.orderBy(col("count").desc() , col("DEST\_COUNTRY\_NAME").asc() ).show(2)  -- in SQL  SELECT \* FROM dfTable ORDER BY count DESC, DEST\_COUNTRY\_NAME ASC LIMIT 2 | advanced tip is to use:   * asc\_nulls\_first, * desc\_nulls\_first, * asc\_nulls\_last, or * desc\_nulls\_last |
| sortWithinPartitions method | spark.read.format("json").load("/data/flight-data/json/\*-summary.json")\  .sortWithinPartitions("count") | For optimization purposes |
| Limit | df.limit(5).show()  df.orderBy(expr("count desc")).limit(6).show() |  |
| Partition | df.rdd.getNumPartitions() # 1  df.repartition(5)  df.repartition(col("DEST\_COUNTRY\_NAME"))  df.repartition(5, col("DEST\_COUNTRY\_NAME")) | * Partition sayisini ogrenme. * Istedigimiz sayisda partlara bolme * Belirli bir column uzerinden bolme |
| Coalesce | df.repartition(5, col("DEST\_COUNTRY\_NAME")).coalesce(2) | Coalesce, sanki tersenten partition gibi, biraraya getirme gibi yani |

# **Chapter 6. Working with Different Types of Data**

Column Methods: All of these tools exist to achieve one purpose, to transform rows of data in one format or structure to another.

In Spark, you should always chain together and filters as a sequential filter. The reason for this is that even if Boolean statements are expressed serially (one after the other), Spark will flatten all of these filters into one statement and perform the filter at the same time, creating the and statement for us. Although you can specify your statements explicitly by using and if you like, they’re often easier to understand and to read if you specify them serially. or statements need to be specified in the same statement:

|  |  |  |
| --- | --- | --- |
| Read in the DataFrame | df = spark.read.format("csv")\  .option("header", "true")\  .option("inferSchema", "true")\  .load("/data/retail-data/by-day/2010-12-01.csv")  df.printSchema()  df.createOrReplaceTempView("dfTable") |  |
| **Working with Booleans**:  and, or, true, and false. | from pyspark.sql.functions import col  df.where(col("InvoiceNo") != 536365)\  .select("InvoiceNo", "Description")\  .show(5, False) |  |
|  | from pyspark.sql.functions import instr  DOTCodeFilter = col("StockCode") == "DOT"  priceFilter = col("UnitPrice") > 600  descripFilter = instr(col("Description"), "POSTAGE") >= 1  df.withColumn("isExpensive", DOTCodeFilter & (priceFilter | descripFilter))\  .where("isExpensive")\  .select("unitPrice", "isExpensive").show(5)  -- in SQL  SELECT UnitPrice, (StockCode = 'DOT' AND  (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1)) as isExpensive  FROM dfTable  WHERE (StockCode = 'DOT' AND  (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1)) |  |
| **Working with Numbers**  power | from pyspark.sql.functions import expr, pow  fabricatedQuantity = pow(col("Quantity") \* col("UnitPrice"), 2) + 5  df.select(expr("CustomerId"), fabricatedQuantity.alias("realQuantity")).show(2) |  |
| multiply | df.selectExpr(  "CustomerId",  "(POWER((Quantity \* UnitPrice), 2.0) + 5) as realQuantity").show(2) | Feature’larin ikisi de numeric oldugu icin aritmetik islemler yapilabiliyor. |
| round | from pyspark.sql.functions import lit, round, bround  df.select(round(lit("2.5")), bround(lit("2.5"))).show(2) |  |
| correlation | from pyspark.sql.functions import corr  df.stat.corr("Quantity", "UnitPrice")  df.select(corr("Quantity", "UnitPrice")).show() |  |
| summary statistics | df.describe().show() | count, mean, standard deviation, min, and max |
| approxQuantile | quantileProbs = [0.5]  relError = 0.05  df.stat.approxQuantile("UnitPrice", quantileProbs, relError) # 2.51 |  |
| cross-tabulation | df.stat.crosstab("StockCode", "Quantity").show() |  |
| frequent item pairs | df.stat.freqItems(["StockCode", "Quantity"]).show() |  |
| monotonically\_increasing\_id | from pyspark.sql.functions import monotonically\_increasing\_id  df.select(monotonically\_increasing\_id()).show(2) | This function generates a unique value for each row, starting with |
| **Working with Strings**  initcap | from pyspark.sql.functions import initcap  df.select(initcap(col("Description"))).show() | The initcap function will  capitalize every word in a given string |
| uppercase and lowercase | from pyspark.sql.functions import lower, upper  df.select(col("Description"),  lower(col("Description")),  upper(lower(col("Description")))).show(2) |  |
| trim,  ltrim,  rtrim,  lpad,  rpad | from pyspark.sql.functions import lit, ltrim, rtrim, rpad, lpad, trim  df.select(  ltrim(lit(" HELLO ")).alias("ltrim"),  rtrim(lit(" HELLO ")).alias("rtrim"),  trim(lit(" HELLO ")).alias("trim"),  lpad(lit("HELLO"), 3, " ").alias("lp"),  rpad(lit("HELLO"), 10, " ").alias("rp")).show(2) | adding or removing spaces around a string |
| **Regular Expressions**  regexp\_replace | from pyspark.sql.functions import regexp\_replace  **regex\_string** = "BLACK|WHITE|RED|GREEN|BLUE"  df.select(  regexp\_replace(col("Description"), **regex**\_**string**, "COLOR").alias("color\_clean"),  col("Description")).show(2) |  |
| translate | from pyspark.sql.functions import translate  df.select(translate(col("Description"), "LEET", "1337"),col("Description"))\  .show(2) |  |
|  | from pyspark.sql.functions import regexp\_extract  **extract\_str** = "(BLACK|WHITE|RED|GREEN|BLUE)"  df.select(  regexp\_extract(col("Description"), **extract\_str**, 1).alias("color\_clean"),  col("Description")).show(2) | pulling out the first mentioned color |
| instr function | from pyspark.sql.functions import instr  containsBlack = instr(col("Description"), "BLACK") >= 1  containsWhite = instr(col("Description"), "WHITE") >= 1  df.withColumn("hasSimpleColor", containsBlack | containsWhite)\  .where("hasSimpleColor")\  .select("Description").show(3, False)  -- in SQL  SELECT Description FROM dfTable  WHERE instr(Description, 'BLACK') >= 1 OR instr(Description, 'WHITE') >= 1 | This will return a Boolean declaring whether the value  you specify is in the column’s string |
| Working with Dates and Timestamps  current date and the current timestamps | from pyspark.sql.functions import current\_date, current\_timestamp  dateDF = spark.range(10)\  .withColumn("today", current\_date())\  .withColumn("now", current\_timestamp())  dateDF.createOrReplaceTempView("dateTable")  dateDF.printSchema() |  |
| date\_add  date\_sub | from pyspark.sql.functions import date\_add, date\_sub  dateDF.select(date\_sub(col("today"), 5), date\_add(col("today"), 5)).show(1) | add and subtract five days from today |
| datediff  months\_between | from pyspark.sql.functions import datediff, months\_between, to\_date  dateDF.withColumn("week\_ago", date\_sub(col("today"), 7))\  .select(datediff(col("week\_ago"), col("today"))).show(1)  dateDF.select(  to\_date(lit("2016-01-01")).alias("start"),  to\_date(lit("2017-05-22")).alias("end"))\  .select(months\_between(col("start"), col("end"))).show(1) | return the number of days in between two dates |
| to\_date function | from pyspark.sql.functions import to\_date, lit  spark.range(5).withColumn("date", lit("2017-01-01"))\  .select(to\_date(col("date"))).show(1) | converts a string to a date, optionally with a specified format. |
| Date error | dateDF.select(to\_date(lit("2016-20-12")),to\_date(lit("2017-12-11"))).show(1) | Spark will not throw an error if it cannot parse the date; it will just return null. |
| specify our date format | from pyspark.sql.functions import to\_date  **dateFormat** = "yyyy-dd-MM"  cleanDateDF = spark.range(1).select(  to\_date(lit("2017-12-11"), **dateFormat**).alias("date"),  to\_date(lit("2017-20-12"), **dateFormat**).alias("date2"))  cleanDateDF.createOrReplaceTempView("dateTable2")  -- in SQL  SELECT to\_date(date, 'yyyy-dd-MM'), to\_date(date2, 'yyyy-dd-MM'), to\_date(date)  FROM dateTable2 | “dateFormat”, value’daki hangi degerin nasil algilanacagini belirliyor. Aslinda her bir karakteri etiketlemek gibi.  “to\_date” ise etiketlenmis olan bilgileri, catalog’a uygun olarak, yani gerekiyorsa yerini degistirerek aliyor. |
| to\_timestamp | from pyspark.sql.functions import to\_timestamp  cleanDateDF.select(to\_timestamp(col("date"), dateFormat)).show()  -- in SQL  SELECT to\_timestamp(date, 'yyyy-dd-MM'), to\_timestamp(date2, 'yyyy-dd-MM')  FROM dateTable2  Equivalent:  SELECT cast(to\_date("2017-01-01", "yyyy-dd-MM") as timestamp) | always requires a format to be specified |
| Working with Nulls in Data  coalesce | from pyspark.sql.functions import coalesce  df.select(coalesce(col("Description"), col("CustomerId"))).show() | Bu fonksiyon, yeni bir sutun halinde getirir bilgiyi, null olmayan ilk veriyi getirir. Yani belirtilen 1’inci column’daki veri “null” is 2’inci column’a bakar ve onu getirir. |
| ifnull, nullIf, nvl, and nvl2 | -- in SQL  SELECT  ifnull(null, 'return\_value'), # null varsa, degeri ikinciden alir  nullif('value', 'value'), # degerler esitse “null” yap, degilse ilk degeri al  nvl(null, 'return\_value'), # “fillna” gibi, “null”lari ikinci “argument” ile doldur  nvl2('not\_null', 'return\_value', "else\_value") # “null” ise 2nci, degilse 3ncu arg’i al.  FROM dfTable LIMIT 1 |  |
| drop null | df.na.drop()  df.na.drop("any")  -- in SQL  SELECT \* FROM dfTable WHERE Description IS NOT NULL  df.na.drop("all")  df.na.drop("all", subset=["StockCode", "InvoiceNo"]) |  |
| Fill null | df.na.fill("All Null values become this string")  df.na.fill("all", subset=["StockCode", "InvoiceNo"])  **fill\_cols\_vals** = {"StockCode": 5, "Description" : "No Value"}  df.na.fill(**fill\_cols\_vals)** |  |
| replace | df.na.replace([""], ["UNKNOWN"], "Description") |  |
| Working w Complex Types | 1. structs, 2. arrays, 3. maps. |  |
| structs | df.selectExpr("(Description, InvoiceNo) as complex", "\*")  df.selectExpr("struct(Description, InvoiceNo) as complex", "\*")  from pyspark.sql.functions import struct  complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))  complexDF.createOrReplaceTempView("complexDF")  complexDF.select("complex.Description")  complexDF.select(col("complex").getField("Description"))  complexDF.select("complex.\*")  -- in SQL  SELECT complex.\* FROM complexDF | Structs’lar dictianary gibi dusunulebilir.  Column name key, value’lar da value olur |
| array | # creating an array  from pyspark.sql.functions import split  df.select(split(col("Description"), " ")).show(2)  # query the values  df.select(split(col("Description"), " ").alias("array\_col"))\  .selectExpr("array\_col[0]").show(2)  -- in SQL  SELECT split(Description, ' ')[0] FROM dfTable |  |
| array: length | from pyspark.sql.functions import size  df.select(size(split(col("Description"), " "))).show(2) # shows 5 and 3 |  |
| array: array\_contains | from pyspark.sql.functions import array\_contains  df.select(array\_contains(split(col("Description"), " "), "WHITE")).show(2)  -- in SQL  SELECT array\_contains(split(Description, ' '), 'WHITE') FROM dfTable |  |
| array: explode | from pyspark.sql.functions import split, explode  df.withColumn("splitted", split(col("Description"), " "))\  .withColumn("exploded", explode(col("splitted")))\  .select("Description", "InvoiceNo", "exploded").show(2)  -- in SQL  SELECT Description, InvoiceNo, exploded  FROM (SELECT \*, split(Description, " ") as splitted FROM dfTable)  LATERAL VIEW explode(splitted) as exploded | Array’deki her bir value’nun sirasiyla yer aldigi bir column olusturur, diger bilgiler aynen tekrar eder. |
| maps | from pyspark.sql.functions import create\_map  df.select(create\_map(col("Description"), col("InvoiceNo")).alias("complex\_map"))\  .show(2)  -- in SQL  SELECT map(Description, InvoiceNo) as complex\_map FROM dfTable  WHERE Description IS NOT NULL | Burda da key-value pairs var.  Structs’dan farkli olarak hem key hem de value, girilen column value’larindan olusuyor. |
|  | df.select(map(col("Description"), col("InvoiceNo")).alias("complex\_map"))\  .selectExpr("complex\_map['WHITE METAL LANTERN']").show(2) |  |
| Working with JSON  creating a JSON  column | jsonDF = spark.range(1).selectExpr("""  '{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as jsonString""") |  |
| get\_json\_object /  json\_tuple | from pyspark.sql.functions import get\_json\_object, json\_tuple  jsonDF.select(  get\_json\_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]") as "column",  json\_tuple(col("jsonString"), "myJSONKey")).show(2)  -- in SQL  jsonDF.selectExpr(  "json\_tuple(jsonString, '$.myJSONKey.myJSONValue[1]') as column").show(2) |  |
| Convert struck to json | from pyspark.sql.functions import to\_json  df.selectExpr("(InvoiceNo, Description) as myStruct")\  .select(to\_json(col("myStruct"))) |  |
| User-Defined Functions | By default, these functions are registered as temporary functions to be used in that specific SparkSession or Context. |  |
| UDF-step\_1:  Create actual function | udfExampleDF = spark.range(5).toDF("num")  def power3(double\_value):  return double\_value \*\* 3 | Python ile yazildiginda spark’in ilaveten data donusumu yapmasi gerekiyor, ayrica memory controlu mumkun olmuyor |
| UDF-step\_2  Register with Spark | from pyspark.sql.functions import udf  power3udf = udf(power3) |  |
| UDF-step\_3:  Using the function | from pyspark.sql.functions import col  udfExampleDF.select(power3udf(col("num"))).show(2) |  |

# **Chapter 7. Aggregations**

|  |  |  |
| --- | --- | --- |
|  | Aggregating is the act of collecting something together and is a cornerstone of big data analytics. In an aggregation, you will specify a key or grouping and an aggregation function that specifies how you should transform one or more columns. This function must produce one result for each group, given multiple input values. | Iki onemli nokta.  1. iki esas parcasi var:  - key ya da grouping  - agg function  Her grup icin bir sonuc uretilir. |
| -“group by” | allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns. |  |
| “window” | gives you the ability to specify one or more keys as well as one or more aggregation functions to transform the value columns. However, the rows input to the function are somehow related to the current row. |  |
| “grouping set” | which you can use to aggregate at multiple different levels. Grouping sets are available as a primitive in SQL and via rollups and cubes in DataFrames. |  |
| “rollup” | makes it possible for you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized hierarchically. |  |
| “cube” | allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized across all combinations of columns. |  |
| Aggreagations | approximation functions is a good opportunity to improve the speed and execution of your Spark jobs, especially for interactive and ad hoc analysis. |  |
| “cache” | df = spark.read.format("csv")\  .option("header", "true")\  .option("inferSchema", "true")\  .load("/data/retail-data/all/\*.csv")\  .coalesce(5)  df.cache()  df.createOrReplaceTempView("dfTable") | caching the results for rapid access |
| basic aggregations | df.count() == 541909 | apply to an entire DataFrame |
| count | from pyspark.sql.functions import count  df.select(count("StockCode")).show() # 541909  -- in SQL  SELECT COUNT(\*) FROM dfTable | - count(\*) or count(1), her satiri “1” olarak sayar.  - count(\*), “null” degerleri de sayar ancak, column’lar sayilirken “null”lar sayilmaz. |
| count\_distinct | from pyspark.sql.functions import countDistinct  df.select(countDistinct("StockCode")).show() # 4070 |  |
| approx\_count\_distinct | from pyspark.sql.functions import approx\_count\_distinct  df.select(approx\_count\_distinct("StockCode", 0.1)).show() # 3364 | Belirli bit hata payi kabulu ile sayar, daha kisa surer. |
| first and last | from pyspark.sql.functions import first, last  df.select(first("StockCode"), last("StockCode")).show() |  |
| min and max | from pyspark.sql.functions import min, max  df.select(min("Quantity"), max("Quantity")).show() |  |
| sum | from pyspark.sql.functions import sum  df.select(sum("Quantity")).show() # 5176450 |  |
| sumDistinct | from pyspark.sql.functions import sumDistinct  df.select(sumDistinct("Quantity")).show() # 29310 |  |
| avg | from pyspark.sql.functions import sum, count, avg, expr  df.select(  count("Quantity").alias("total\_transactions"),  sum("Quantity").alias("total\_purchases"),  avg("Quantity").alias("avg\_purchases"),  expr("mean(Quantity)").alias("mean\_purchases"))\  .selectExpr(  "total\_purchases/total\_transactions",  "avg\_purchases",  "mean\_purchases").show() | 3’u de ayni kapiya cikar:   * sum/count * average * mean |
| variance and stdv | from pyspark.sql.functions import var\_pop, stddev\_pop  from pyspark.sql.functions import var\_samp, stddev\_samp  df.select(var\_pop("Quantity"), var\_samp("Quantity"),  stddev\_pop("Quantity"), stddev\_samp("Quantity")).show() |  |
| skewness and kurtosis | from pyspark.sql.functions import skewness, kurtosis  df.select(skewness("Quantity"), kurtosis("Quantity")).show() | **Skewness** measures the  asymmetry of the values in your data around the mean, **kurtosis** is a measure of the tail of data. |
| covariance and correlation | from pyspark.sql.functions import corr, covar\_pop, covar\_samp  df.select(corr("InvoiceNo", "Quantity"), covar\_samp("InvoiceNo", "Quantity"),  covar\_pop("InvoiceNo", "Quantity")).show() |  |
| Aggregating to Complex Types | from pyspark.sql.functions import collect\_set, collect\_list  df.agg(collect\_set("Country"), collect\_list("Country")).show() |  |
| groupBy | df.groupBy("InvoiceNo", "CustomerId").count().show()  -- in SQL  SELECT count(\*) FROM dfTable GROUP BY InvoiceNo, CustomerId | typically done on categorical data for which we group our data on one column and perform some calculations on the other columns. |
| Grouping with Expressions | from pyspark.sql.functions import count  df.groupBy("InvoiceNo").agg(  count("Quantity").alias("quan"),  expr("count(Quantity)")).show() | “agg” ve “expr” ile kullanimi ayni kapiya cikiyor ama “agg”tercih ederiz diyor??? |
| Grouping with Maps | df.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),expr("stddev\_pop(Quantity)"))\  .show()  -- in SQL  SELECT avg(Quantity), stddev\_pop(Quantity), InvoiceNo FROM dfTable  GROUP BY InvoiceNo |  |
| Window functions | Group-by’da her bir satir, sadece bir gruba ait hesaplamaya dahil edilir. Window’da ise ayni satir birden fazla grubun icinde yer alabilir. Her bir satir icin son bir haftanin ortalam satis degerinin alinmasi gibi. Bir satir, 7 farkli window’da yer almis olur.  Spark supports three kinds of window functions:   * ranking functions, * analytic functions, and * aggregate functions. |  |
|  | from pyspark.sql.functions import col, to\_date  dfWithDate = df.withColumn("date", to\_date(col("InvoiceDate"), "MM/d/yyyy H:mm"))  dfWithDate.createOrReplaceTempView("dfWithDate") | “date” column’in olusturulmasi |
| Step-1:  Window (frame) specification | from pyspark.sql.window import Window  from pyspark.sql.functions import desc  windowSpec = Window\  .partitionBy("CustomerId", "date")\  .orderBy(desc("Quantity"))\  .rowsBetween(Window.unboundedPreceding, Window.currentRow) | Butun onceki satirlarin dahil edildigi bir “window” olusturuldu.  partition by is unrelated to the partitioning scheme |
| Step-2:  Aggregation function | from pyspark.sql.functions import max  maxPurchaseQuantity = max(col("Quantity")).over(windowSpec) | this returns a column (or expressions) |
|  | from pyspark.sql.functions import dense\_rank, rank  purchaseDenseRank = dense\_rank().over(windowSpec)  purchaseRank = rank().over(windowSpec)  from pyspark.sql.functions import col  dfWithDate.where("CustomerId IS NOT NULL").orderBy("CustomerId")\  .select(  col("CustomerId"),  col("date"),  col("Quantity"),  purchaseRank.alias("quantityRank"),  purchaseDenseRank.alias("quantityDenseRank"),  maxPurchaseQuantity.alias("maxPurchaseQuantity")).show() | the dense\_rank function to determine which date had the maximum purchase quantity for every customer. We use dense\_rank as opposed to rank to avoid gaps in the ranking sequence when there are tied values (or in our case, duplicate  rows)  perform a select to  view the calculated window values |
|  | -- in SQL  SELECT CustomerId, date, Quantity,  rank(Quantity) OVER (PARTITION BY CustomerId, date  ORDER BY Quantity DESC NULLS LAST  ROWS BETWEEN  UNBOUNDED PRECEDING AND  CURRENT ROW) as rank,  dense\_rank(Quantity) OVER (PARTITION BY CustomerId, date  ORDER BY Quantity DESC NULLS LAST  ROWS BETWEEN  UNBOUNDED PRECEDING AND  CURRENT ROW) as dRank,  max(Quantity) OVER (PARTITION BY CustomerId, date  ORDER BY Quantity DESC NULLS LAST  ROWS BETWEEN  UNBOUNDED PRECEDING AND  CURRENT ROW) as maxPurchase  FROM dfWithDate WHERE CustomerId IS NOT NULL ORDER BY CustomerId |  |
|  |  |  |
| Grouping Sets | -- in SQL  SELECT CustomerId, stockCode, sum(Quantity) FROM dfNoNull  GROUP BY customerId, stockCode GROUPING SETS((customerId, stockCode))  ORDER BY CustomerId DESC, stockCode DESC | Birden fazla groupBy gereken durumlar icin diyor ama tam oturmadi. |
| rollup | rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))\  .selectExpr("Date", "Country", "`sum(Quantity)` as total\_quantity")\  .orderBy("Date")  rolledUpDF.show() | A rollup is a multidimensional aggregation that performs a variety of group-by style calculations for us. |
| cube | can you make a table that includes the following?   * The total across all dates and countries * The total for each date across all countries * The total for each country on each date * The total for each country across all dates   The method call is quite similar, but instead of calling rollup, we call cube | A |
|  | from pyspark.sql.functions import sum  dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))\  .select("Date", "Country", "sum(Quantity)").orderBy("Date").show() |  |
| pivot | pivoted = dfWithDate.groupBy("date").pivot("Country").sum()  pivoted.where("date > '2011-12-05'").select("date" ,"`USA\_sum(Quantity)`").show() |  |
|  |  |  |
|  |  |  |

# **Chapter 8. Joins**

|  |  |  |
| --- | --- | --- |
| inner join | joinExpression = person["graduate\_program"] == graduateProgram['id']  person.join(graduateProgram, joinExpression).show()  -- in SQL  SELECT \* FROM person JOIN graduateProgram  ON person.graduate\_program = graduateProgram.id |  |
| outer join | joinType = "outer"  person.join(graduateProgram, joinExpression, joinType).show()  -- in SQL  SELECT \* FROM person FULL OUTER JOIN graduateProgram  ON graduate\_program = graduateProgram.id |  |
| Left outer | joinType = "left\_outer"  graduateProgram.join(person, joinExpression, joinType).show()  -- in SQL  SELECT \* FROM graduateProgram LEFT OUTER JOIN person  ON person.graduate\_program = graduateProgram.id |  |
| Right outer | joinType = "right\_outer" |  |
| Left Anti join | joinType = "left\_anti"  graduateProgram.join(person, joinExpression, joinType).show()  -- in SQL  SELECT \* FROM graduateProgram LEFT ANTI JOIN person  ON graduateProgram.id = person.graduate\_program  Alternative sql command:  Select \* from Dept  Where not exists (select \* from person where Dept.id = person.Dept) |  |
| Left Semi Join | joinType = "left\_semi"  graduateProgram.join(person, joinExpression, joinType).show() | Inner join’den farki, soldaki df’e ait columnlar result’a dahil edilmez. Yani result, left ile kesisen sag “df” satirlaridir. |
| Cross (Cartesian) join | joinType = "cross"  graduateProgram.join(person, joinExpression, joinType).show()  -- in SQL  SELECT \* FROM graduateProgram CROSS JOIN person  ON graduateProgram.id = person.graduate\_program |  |

# **Chapter 9. Data Sources**

|  |  |  |
| --- | --- | --- |
| Spark’s core data sources: | 1. CSV 2. JSON 3. Parquet 4. ORC 5. JDBC/ODBC connections 6. Plain-text files |  |
| community-created data sources samples | * Cassandra * HBase * MongoDB * AWS Redshift * XML * And many, many others |  |
|  | Before proceeding with how to read and write from certain formats, let’s visit the overall  organizational structure of the data source APIs.  The core structure for reading data is as follows: |  |
| The Structure of the Data Sources API |  |  |
| Read API Structure | **DataFrameReader**.**format**(...).**option**("key", "value").**schema**(...).**load**() |  |
| Basics of Reading Data | The foundation for reading data in Spark is the **DataFrameReader**. We access this SparkSession object = spark  “SparkSession object” + “read attribute” = **spark.read** = DataFrameReader  After we have a DataFrame reader, we specify several values:   * The format (default=”parquet”) * The schema * The read mode (1.permissive / 2. dropMalformed / 3. failFast) * A series of options (at a minumum = “path”) | **Permissive** -> Sets all fields to null when it encounters a corrupted record  **dropMalformed** -> Drops the row that contains malformed records  **failFast** -> Fails immediately upon encountering malformed |
| Reading example | **spark.read**.**format**("csv")  **.option**("mode", "FAILFAST")  .**option**("inferSchema", "true")  .**option**("path", "path/to/file(s)")  **.schema**(someSchema)  .**load**() |  |
| Write API Structure | **DataFrameWriter**.**format**(...).**option**(...).partitionBy(...).bucketBy(...).sortBy(  ...).**save**() |  |
| Basics of Writing Data | **PartitionBy**, **bucketBy**, and **sortBy** work only for file-based data sources; you can use them to control the specific layout of files at the destination.  After we have a DataFrameWriter, we specify three values::   * The format (optinal, because default is ”parquet”) * The save mode (1. append / 2. overwrite / 3. **errorIfExists** / 4. ignore) * A series of options (at a minumum = “path”) | **append** -> Appends the output files to the existing ones.  **overwrite** -> overwrite any data that already exists there  **errorIfExists** Throws an error if data or files already exist.  **ignore** If data or files exist do nothing with the current |
| Writing example | dataframe.write.format("csv")  .option("mode", "OVERWRITE")  .option("dateFormat", "yyyy-MM-dd")  .option("path", "path/to/file(s)")  .save() |  |
| Parquet Files | Parquet is an open source **column-oriented** data store that provides a variety of storage optimizations. It provides columnar compression, which saves storage space and allows for reading individual columns instead of entire files. It is the **default** file format of Spark. reading from a Parquet file will always be more **efficient than JSON or CSV**. Another advantage of Parquet is that it supports complex types. |  |
| ORC Files  (Optimized Row Columnar) | ORC is a self-describing, type-aware columnar file format **designed for Hadoop** workloads. It is optimized for large streaming reads. ORC and Parquet are quite similar; the fundamental difference is that Parquet is further optimized for use with Spark, whereas ORC is **further optimized for Hive**. |  |
| Csv | **reading:**  csvFile = spark.read.format("csv")\  .option("header", "true")\  .option("mode", "FAILFAST")\  .option("inferSchema", "true")\  .**load**("/data/flight-data/csv/2010-summary.csv")  Writing:  csvFile.write.format("csv").mode("overwrite").option("sep", "\t")\  .**save**("/tmp/my-tsv-file.tsv") |  |
| Json | spark.read.format("json").option("mode", "FAILFAST")\  .option("inferSchema", "true")\  .**load**("/data/flight-data/json/2010-summary.json").show(5)  csvFile.write.format("json").mode("overwrite").**save**("/tmp/my-json-file.json") |  |
| parquet | spark.read.format("parquet")\  .**load**("/data/flight-data/parquet/2010-summary.parquet").show(5)  csvFile.write.format("parquet").mode("overwrite")\  .**save**("/tmp/my-parquet-file.parquet") |  |
|  | To read and write from SQL databases, you need to do two things:   * include the Java Database Connectivity (**JDBC**) driver for you particular database on the spark classpath, * and provide the proper **JAR** for the driver itself.   For example, to be able to read and write from PostgreSQL:  ./bin/spark-shell \  --driver-class-path postgresql-9.4.1207.jar \  --jars postgresql-9.4.1207.jar |  |
| Reading from SQLite | driver = "org.sqlite.JDBC"  path = "/data/flight-data/jdbc/my-sqlite.db"  url = "jdbc:sqlite:" + path  tablename = "flight\_info"  dbDataFrame = spark.read.format("jdbc").option("url", url)\  .option("dbtable", tablename).option("driver", driver).load() |  |
| Reading from PostgreSQL | pgDF = spark.read.format("jdbc")\  .option("driver", "org.postgresql.Driver")\  .option("url", "jdbc:postgresql://database\_server")\  .option("dbtable", "schema.tablename")\  .option("user", "username").option("password", "my-secret-password").load() |  |
| Query Pushdown | dbDataFrame.filter("DEST\_COUNTRY\_NAME in ('Anguilla', 'Sweden')").explain() |  |
|  | pushdownQuery = """(SELECT DISTINCT(DEST\_COUNTRY\_NAME) FROM flight\_info)  AS flight\_info"""  dbDataFrame = spark.read.format("jdbc")\  .option("url", url).option("dbtable", pushdownQuery).option("driver", driver)\  .load() | Ihtiyac duyuldugunda SQL query’ler bir variable’a atanarak kullanilabilir. |
|  | Bu bolumu kisa kestim. Simdilik gerekli degil diye dusunuyorum |  |

Chapter 10. Spark SQL

|  |  |  |
| --- | --- | --- |
| What Is SQL? | * SQL or Structured Query Language is a domain-specific language for expressing relational operations over data. * Spark implements a subset of ANSI SQL:2003. This SQL standard is one that is available in the majority of SQL databases. * with Spark SQL you can run SQL queries against views or tables organized into databases. * you can choose to express some of your data manipulations in SQL and others in DataFrames and they will compile to the same underlying code. | Ayni logical plan icersinde, hem dataframe hem de sql manipulation’i yapilabilir, spark bunlarin hepsini ayni zemine indirger. |
| Big Data and SQL | * Apache Hive Before Spark’s rise, Hive was the de facto big data SQL access layer. Originally developed at Facebook. * In many ways it helped propel Hadoop into different industries. * Although Spark began as a general processing engine with Resilient Distributed * Datasets (RDDs), a large cohort of users now use Spark SQL. * Spark 2.0’dan itibaren ANSI-SQL ile birlikte HiveQL’in de querileri Spark SQL tarafindan yapilabilir hale geldi. * SQL API for Spark allows for data to be extracted with SQL, manipulated as a DataFrame, passed into one of Spark MLlibs’ large-scale machine learning algorithms, written out to another data source, and everything in between. * Spark SQL is intended to operate as an online analytic processing (OLAP) database, not an online transaction processing (OLTP) database. | Hive, bundan onceki big data icn kullanilan SQL access layer’I idi.  Su anda Spark SQL daha yaygin |
| How to Run Spark SQL Queries | Spark provides several interfaces to execute SQL queries.   * **Spark SQL CLI**: you can make basic Spark SQL queries in local mode from the command line.   ./bin/spark-sql   * **Spark’s Programmatic SQL Interface**   via any of Spark’s language APIs. You can do this via the method sql on the SparkSession object. This returns a DataFrame, as we will see later in this chapter.  spark.sql("SELECT 1 + 1").show() |  |
|  | spark.read.json("/data/flight-data/json/2015-summary.json")\  .createOrReplaceTempView("some\_sql\_view") # DF => SQL  spark.sql("""  SELECT DEST\_COUNTRY\_NAME, sum(count)  FROM some\_sql\_view GROUP BY DEST\_COUNTRY\_NAME  """)\  .where("DEST\_COUNTRY\_NAME like 'S%'").where("`sum(count)` > 10")\  .count() # SQL => DF | SQL query’I birden fazla satirda yazmak  SQL’nin kullanildigi ayni yerde DataFrame olusturmak da mumkundur. |
| SparkSQL Thrift JDBC/ODBC Server | Spark provides a Java Database Connectivity (**JDBC**)/ Open Database Connectivity (**ODBC**) interface by which either you or a remote program connects to the Spark driver **in order to execute Spark SQL queries**. A common use case might be a for a business analyst to connect business intelligence software like Tableau to Spark. |  |
| Catalog | **The highest level abstraction in Spark SQL** is the Catalog. The Catalog is an abstraction for the storage of metadata about the data stored in your tables as well as other helpful things like databases, tables, functions, and views. |  |
| Tables | * To do anything useful with Spark SQL, you first need to define tables. * Tables are logically **equivalent to a DataFrame**. * The core difference between tables and DataFrames is this: you define **DataFrames** in the scope of a programming language, whereas you define **tables** within a database. |  |
| Spark-Managed Tables | One important note is the concept of **managed** versus **unmanaged** tables.   * When you define a table from files on disk, you are defining an unmanaged table. * When you use saveAsTable on a DataFrame, you are creating a managed table for which Spark will track of all of the relevant information.   Tables store two important pieces of information.   * The **data** within the tables, * as well as the data about the tables; that is, the **metadata**. |  |
| Creating Tables | Something fairly unique to Spark is the capability of reusing the entire Data Source API within SQL. This means that you do not need to define a table and then load data into it; Spark lets you create one on the fly.  **Read in:**  CREATE TABLE flights\_csv (  DEST\_COUNTRY\_NAME STRING,  ORIGIN\_COUNTRY\_NAME STRING COMMENT "remember, the US will be most prevalent",  count LONG)  USING csv OPTIONS (header true, path '/data/flight-data/csv/2015-summary.csv')  **From a query:**  CREATE TABLE flights\_from\_select USING parquet AS SELECT \* FROM flights | USING/STORED AS kullanilmaz ise default olarak Hive SerDe configuration kullanilir.  Ayrica burada “COMMENT” opsiyonel. |
|  | CREATE TABLE IF NOT EXISTS flights\_from\_select  AS SELECT \* FROM flights | Iki husus:  1. IF NOT EXIST” opsiyonel.  2. “USING” kullanilmadigi icin Hive uyumlu bir “table” olusturulacak. |
| Inserting into tables | INSERT INTO flights\_from\_select  SELECT DEST\_COUNTRY\_NAME, ORIGIN\_COUNTRY\_NAME, count FROM flights LIMIT 20 |  |
| Describing table metadata | DESCRIBE TABLE flights\_csv |  |
| Dropping tables | DROP TABLE [IF EXISTS] flights\_csv; |  |
| Caching tables | CACHE TABLE flights  UNCACHE TABLE FLIGHTS |  |
|  | Temp view: temporary views that are available only during the current session and are not registered to a database |  |
| Creating views | CREATE [OR REPLACE] [TEMP] VIEW just\_usa\_view\_temp AS  SELECT \* FROM flights WHERE dest\_country\_name = 'United States'  Now you can query this view just as if it were another table:  SELECT \* FROM just\_usa\_view\_temp |  |
|  | EXPLAIN SELECT \* FROM just\_usa\_view |  |
| Dropping views | DROP VIEW IF EXISTS just\_usa\_view |  |
| Select Statements | SELECT [ALL|DISTINCT] named\_expression[, named\_expression, ...]  FROM relation[, relation, ...]  [lateral\_view[, lateral\_view, ...]]  [WHERE boolean\_expression]  [aggregation [HAVING boolean\_expression]]  [ORDER BY sort\_expressions]  [CLUSTER BY expressions]  [DISTRIBUTE BY expressions]  [SORT BY sort\_expressions]  [WINDOW named\_window[, WINDOW named\_window, ...]]  [LIMIT num\_rows]  named\_expression:  : expression [AS alias]  relation:  | join\_relation  | (table\_name|query|relation) [sample] [AS alias]  : VALUES (expressions)[, (expressions), ...]  [AS (column\_name[, column\_name, ...])]  expressions:  : expression[, expression, ...]  sort\_expressions:  : expression [ASC|DESC][, expression [ASC|DESC], ...] |  |
| case…when…then Statements | SELECT  CASE WHEN DEST\_COUNTRY\_NAME = 'UNITED STATES' THEN 1  WHEN DEST\_COUNTRY\_NAME = 'Egypt' THEN 0  ELSE -1 END  FROM partitioned\_flights |  |
| Complex Types | There are three core complex types in Spark SQL:   * structs, * lists, and * maps |  |
| structs | CREATE VIEW IF NOT EXISTS nested\_data AS  SELECT (DEST\_COUNTRY\_NAME, ORIGIN\_COUNTRY\_NAME) as country, count FROM flights  SELECT \* FROM nested\_data  SELECT country.DEST\_COUNTRY\_NAME, count FROM nested\_data | Map’e benzer  Nested data olusturmak icin kullanilir |
| lists | SELECT DEST\_COUNTRY\_NAME as new\_name, collect\_list(count) as flight\_counts,  collect\_set(ORIGIN\_COUNTRY\_NAME) as origin\_set  FROM flights GROUP BY DEST\_COUNTRY\_NAME |  |

Chapter 12. Resilient Distributed Datasets (RDDs)

|  |  |  |
| --- | --- | --- |
| What Are the Low-Level APIs? | There are two sets of low-level APIs:   * one for manipulating distributed data (**RDDs**), and * another for distributing and manipulating **distributed shared variables** * broadcast variables and * accumulators |  |
| When to Use the Low-Level APIs? | You need some functionality  to maintain some legacy codebase written using RDDs.  to do some custom shared variable manipulation. | Bir dataFrame transformation’i aslinda bir RDD transformation seti.  Cok ozel bir nedeni yoksa manupulation’I RDD seviyesinde yapmayin diyor |
| About RDDs | * **all Spark code** you run, whether DataFrames or Datasets, **compiles down to an RDD**. * In short, an RDD represents an **immutable, partitioned collection of records** that can be operated on in parallel. * In Dataframe -> each record is a structured row containing fields with a known schema * In RDDs -> RDDs the records are just Java, Scala, or Python **objects** of the programmer’s choosing. * there is no concept of “rows” in RDDs; individual records are just Java/Scala/Python objects, and you manipulate those manually | Bildigimiz satir konseptinden farklidir diyor RDD, onlar Python objelerinden ibaretler |
| Types of RDDs | “generic” RDD type  key-value RDD  RDDs. Internally, each RDD is characterized by five: |  |
| main properties of RDDs | * A list of partitions * A function for computing each split * A list of dependencies on other RDDs * Optionally, a Partitioner for key-value RDDs * Optionally, a list of preferred locations on which to compute each split |  |
|  |  |  |
|  |  |  |
|  |  |  |

# **Chapter 14. Distributed Shared Variables**

**Broadcast Variables**

Broadcast variables are a way you can share an immutable value efficiently around the cluster without encapsulating that variable in a function closure. The normal way to use a variable in your driver node inside your tasks is to simply reference it in your function closures (e.g., in a map operation), but this can be inefficient, especially for large variables such as a lookup table or a machine learning model. The reason for this is that when you use a variable in a closure, it must be deserialized on the worker nodes many times (one per task). Moreover, if you use the same variable in multiple Spark actions and jobs, it will be re-sent to the workers with every job instead of once.

This is here broadcast variables come in. Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task.

|  |  |
| --- | --- |
| Normalde, “task” icersindeki “drive node”da bir variable kullanilmak istendiginde bu variable bir “function closure” incersine enkapsule edilir. Ancak bu yontem “bir lookup table” olusturmak veya ML modeli icin cok efektif olmaz. Bunun nedeni variable’nin “worker node”lari uzerinde cok fazla sayida (her bir task icin bir) “deseroalize” edilmesini gerektirmesidir. Bu variable’yi birden falza “action” ve”job”da kullanmak istedigimizde her defasinda her “job” ile birlikte “worker”a gonderilmesi gerekir.  Broadcast variables”larin devreye girdigi yer burasidir. Her bir “task” de desirialize etmek yerine, “cluster”daki her bir “machine”de cache edilir, shared ve immutable’dir. |  |

**Accumulators**

Accumulators, Spark’s second type of shared variable, are a way of updating a value inside of a variety of transformations and propagating that value to the driver node in an efficient and fault-tolerant way.

Accumulators provide a mutable variable that a Spark cluster can safely update on a per-row basis. You can use these for debugging purposes (say to track the values of a certain variable per partition in order to intelligently use it over time) or to create low-level aggregation. Accumulators are variables that are “added” to only through an associative and commutative operation and can therefore be efficiently supported in parallel. You can use them to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

For accumulator updates performed inside actions only, Spark guarantees that each task’s update to the accumulator will be applied only once, meaning that restarted tasks will not update the value.

|  |  |
| --- | --- |
| Numerik bir degerin, ayni transformations’lari tekrar yapamaya gerek kalmadan guncellenmesini saglamak icin oluturulan “variable”lerdir. “action” icersinde perform edilir. Bir task’de sadece bir update yapilir, “task” restart edildiginde degerin tekrar guncellenmesi soz konusu degildir. |  |

Part IV. Production Applications

# **Chapter 15. How Spark Runs on a Cluster**

|  |  |
| --- | --- |
| **The Spark driver** | The driver is **the process** of your Spark Application. It is **the controller** of the execution of a Spark Application and maintains all of the state of the Spark cluster (the state and tasks of the executors). It must interface with the cluster manager in order to actually get physical resources and launch executors. At the end of the day, this is just **a process on a physical machine** that is r**esponsible for maintaining the state of the application running on the cluster.** |
| **The Spark executors** | Spark executors are **the processes that perform the tasks assigned by the Spark driver**. Executors have one core responsibility: take the tasks assigned by the driver, run them, and report back their state and results. Each Spark Application has its own separate executor processes. |
| **The cluster manager** | The cluster manager is responsible for maintaining a cluster of machines that will run your Spark Application(s). Somewhat confusingly, a cluster manager will have **its own “driver”** (sometimes called **master**) and **“worker” abstractions**. The core difference is that these are tied to physical machines rather than processes (as they are in Spark). Figure 15-1 shows a basic cluster setup. The machine on the left of the illustration is the Cluster Manager Driver Node[[1]](#footnote-1). The circles represent daemon processes running on and managing each of the individual worker nodes. There is no Spark Application running as of yet—these are just the processes from the cluster manager.  Spark currently supports three cluster managers:   * built-in **standalone** cluster manager, * Apache **Mesos**, and * Hadoop **YARN**. |
| Execution Modes | Cluster mode  Client mode  Local mode |

|  |  |  |
| --- | --- | --- |
| Cluster mode | In cluster mode, a **user submits** a pre-compiled JAR, Python script, or R script to a cluster manager.  The cluster manager then launches the driver process on a worker node inside the cluster, in addition to the executor processes.  This means that the cluster manager is responsible for maintaining all Spark Application– related processes. Figure shows that the **cluster manager placed our driver on a worker node and the executors on other worker nodes**. |  |
| Client mode | Client mode is nearly the same as cluster mode except that the **Spark driver remains on the client machine** that submitted the application. This means that the client machine is responsible for maintaining the Spark driver process, and the cluster manager maintains the executor processses. we are running the Spark Application from a machine that is not colocated on the cluster.  These machines are commonly referred to as **gateway machines** or **edge nodes**. In Figure 15-3, you can see that the driver is running on a machine outside of the cluster but that the workers are located on machines in the cluster. |  |
| Local mode | it runs the entire Spark Application **on a single machine**. It achieves parallelism through threads on that single machine. This is a common way to learn Spark, to test your applications, or experiment iteratively with local development. However, we do not recommend using local mode for running production applications. |  |
|  | **The Life Cycle of a Spark Application (Outside Spark)** |  |
| Step-1  Client Request | * Client request’ini “spark application”i submit ediyor. terminal’den * pre-compiled JAR or * library. * Initial kod benim bilgisayarda calismaya basliyor. * “request”in yaptigi sey, “spark driver process” icin “cluster manager”dan kaynak (node) talep etmektir. * “cluster manager” bu istegi kabul edip, “driver”i “cluster”daki bir “node”a yerlestirir. * Artik client “original job”i submit etmistir ve application off’a gecer ve cluster’da calismaya baslar. |  |
| Step-2  Launch | * “user code” mutlaka “Spark cluster”i (driver + executors) baslatacak “spark session”i da icermelidir. * “spark session”, cluster icersindeki “executor process”lerin baslatilmasi icin “cluster manager” ile iletisime gecer. * Executor sayisi ve “configuration”, user tarafindan “command-line”da yazilan orjinal “spark-submit call”da yer alir. “cluster manager” da buna “executor process”leri baslatarak ve “location” bilgilerini gondererek cevap verir. * Hersey yolunda gitti ise artik bir “spark cluster”imiz var demektir. |  |
| Step-3  Execution | * “spark cluster” kodu calistirirken, “drive” ve “worker”lar iletisim halindedir. * “driver” gorev planlamasini yapar, “worker”lar gorevin son durumunu ilettikleri bir iletisim icersindedirler “drive” ile. |  |
| Step-4  Completion | * Application tamamlandiginda (success or failure) “cluster manager”, “driver”a bagli “executor”lari kapatir. “cluster manager” uzerinden yapilacak bir sorgulamayla basari ya da basarisiz oldugu ogrenilebilir. |  |
|  | **The Life Cycle of a Spark Application (Inside Spark)** |  |
| The SparkSession | The first step of any Spark Application is creating a SparkSession. In many interactive modes, this is done for you, but in an application, you must do it manually.  After you have a SparkSession, you should be able to run your Spark code. |  |
| Logical Instructions | As you saw in the beginning of the book, Spark code essentially consists of transformations and actions. How you build these is up to you—whether it’s through SQL, low-level RDD manipulation, or machine learning algorithms.  a three-step job: using a simple DataFrame,   1. we’ll repartition it, 2. perform a value-by-value manipulation, 3. and then aggregate some values and collect the final result.   df1 = spark.range(2, 10000000, 2)  df2 = spark.range(2, 10000000, 4)  step1 = df1.repartition(5)  step12 = df2.repartition(6)  step2 = step1.selectExpr("id \* 5 as id")  step3 = step2.join(step12, ["id"])  step4 = step3.selectExpr("sum(id)")  step4.collect() # 2500000000000 |  |
| A Spark Job | * one Spark job for one action. Actions always return results. * Each job breaks down into a series of stages, * the number of which depends on how many shuffle operations need to take place. |  |
| Stages | Ayni islemleri iceren ama farkli “machine”ler uzerinde isleyebilen “task” gruplarini temsil eder.  Spark, mumkun oldugunca en fazla sayida “transformations”i ayni stage’e toplamaya calisir.  Tabi islemler, ayni shuffle icersinde yapilabildigi surece, ayri bir shuffle gerekiyorsa farkli bir stage olusturulur.  Shuffle data’nin fiziksel olarak bolumlenmesini (repartition) temsil eder. Executor’lar arasinda ayri bir iletisimi ve koordinasyonu gerektiren bir islem, ayrica ele alinir, dolayisi ile ayri bir stage olur. |  |
|  | “Partition” sayisi “cluster”inizdaki “core” sayisina gore belirlenmelidir.  Kural olarak, “cluster”daki “executor” sayisindan buyuk olmali. Local’da calisiliyorsa daha kucuk olmali |  |
| Task | Her bir “task”, tek bir “executor” uzerinde islenecek olan data bloklari ve “transformations” setinin kombinasyonunu temsil eder.  Diger bir ifadeyle, bir “data unit”e uygulancak olan “computation unit”i ifade eder.  1000 partition varsa, paralel calisabilecek 1000 task var demektir. |  |
| Execution Details | in Spark have some important properties of Tasks and stages:   * First, Spark automatically pipelines stages and tasks that can be done together, such as a map operation followed by another map operation. * Second, for all shuffle operations, Spark writes the data to stable storage (e.g., disk), and can reuse it across multiple jobs.   Pipelining  Ayni data uzerinden yapilacak islemler icin, “data”yi her adimda yazmak yerine, “disk”e bir kere yazip, bitene kadar ayni “data”yi “pipeline”da tutar. Boylece “memory” ve “runtime” tasarrufu saglanmis olur. |  |
|  |  |  |

|  |  |
| --- | --- |
|  | Spark cluster components |

|  |
| --- |
| Complete Picture of Apache Spark Job Execution Flow. |

1. A node can be a couple of different things depending on whether the conversation is about computer science or networking. In networking a node is either a **connection point**, a redistribution point, or a **communication endpoint**. In computer science, nodes are **devices or data points on a large network**, devices such a PC, phone, or printer are considers nodes.

   In general, a node has a programmed or engineered capability that enables it to recognize, process, or forward transmissions to other nodes.

   Any system or device connected to a network is also called a node. For example, if a network connects a file server, five computers, and two printers, there are eight nodes on the network. Each device on the network has a network address, such as a MAC address, which uniquely identifies each device. This helps keep track of where data is being transferred to and from on the network. [↑](#footnote-ref-1)