## For implicit conversions like converting RDDs to DataFrames

import spark.implicits.\_

## Read from JSON file

val df = spark.read.json("examples/src/main/resources/people.json")

## This import is needed to use the $-notation

import spark.implicits.\_

## Print the schema in a tree format

df.printSchema()

## Select only the "name" column

df.select("name").show()

## Select everybody, but increment the age by 1

df.select($"name", $"age" + 1).show()

## Select people older than 21

df.filter($"age" > 21).show()

## Count people by age

df.groupBy("age").count().show()

## Register the DataFrame as a SQL temporary view

df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT \* FROM people")

sqlDF.show()

## For implicit conversions from RDDs to DataFrames

import spark.implicits.\_

## Create an RDD of Person objects from a text file, convert it to a Dataframe

val peopleDF = spark.sparkContext

.textFile("examples/src/main/resources/people.txt")

.map(\_.split(","))

.map(attributes => Person(attributes(0), attributes(1).trim.toInt))

.toDF()

## Register the DataFrame as a temporary view

peopleDF.createOrReplaceTempView("people")

## SQL statements can be run by using the sql methods provided by Spark

val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")

## The columns of a row in the result can be accessed by field index

teenagersDF.map(teenager => "Name: " + teenager(0)).show()

## Type-Safe User-Defined Aggregate Functions

import org.apache.spark.sql.{Encoder, Encoders, SparkSession}

import org.apache.spark.sql.expressions.Aggregator

case class Employee(name: String, salary: Long)

case class Average(var sum: Long, var count: Long)

object MyAverage extends Aggregator[Employee, Average, Double] {

// A zero value for this aggregation. Should satisfy the property that any b + zero = b

def zero: Average = Average(0L, 0L)

// Combine two values to produce a new value. For performance, the function may modify `buffer`

// and return it instead of constructing a new object

def reduce(buffer: Average, employee: Employee): Average = {

buffer.sum += employee.salary

buffer.count += 1

buffer

}

// Merge two intermediate values

def merge(b1: Average, b2: Average): Average = {

b1.sum += b2.sum

b1.count += b2.count

b1

}

// Transform the output of the reduction

def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count

// Specifies the Encoder for the intermediate value type

def bufferEncoder: Encoder[Average] = Encoders.product

// Specifies the Encoder for the final output value type

def outputEncoder: Encoder[Double] = Encoders.scalaDouble

}

val ds = spark.read.json("examples/src/main/resources/employees.json").as[Employee]

ds.show()

// +-------+------+

// | name|salary|

// +-------+------+

// |Michael| 3000|

// | Andy| 4500|

// | Justin| 3500|

// | Berta| 4000|

// +-------+------+

// Convert the function to a `TypedColumn` and give it a name

val averageSalary = MyAverage.toColumn.name("average\_salary")

val result = ds.select(averageSalary)

result.show()

// +--------------+

// |average\_salary|

// +--------------+

// | 3750.0|

// +--------------+

## Load and Save

val peopleDF = spark.read.format("json").load("/main/resources/people.json")

peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")

// Load

val peopleDFCsv = spark.read.**format**("csv")

.**option**("sep", ";")

.option("inferSchema", "true")

.**option**("header", "true")

.**load**("examples/src/main/resources/people.csv")

// Save

usersDF.write.format("orc")

.option("orc.bloom.filter.columns", "favorite\_color")

.option("orc.dictionary.key.threshold", "1.0")

.save("users\_with\_options.orc")

## Read JSON files

import spark.implicits.\_

// A JSON dataset is pointed to by path.

// The path can be either a single text file or a directory storing text files

val path = "examples/src/main/resources/people.json"

val peopleDF = spark.read.json(path)

## Create a dataset

/ Alternatively, a DataFrame can be created for a JSON dataset represented by

// a Dataset[String] storing one JSON object per string

val otherPeopleDataset = spark.createDataset(

"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)

val otherPeople = spark.read.json(otherPeopleDataset)

## Bucketing, Sorting and Partitioning

peopled

.write

.bucketBy(42, "name")

.sortBy("age")

.saveAsTable("people\_bucketed")

usersDF

.write

.partitionBy("favorite\_color")

.format("parquet")

.save("namesPartByColor.parquet")

usersDF

.write

.partitionBy("favorite\_color")

.bucketBy(42, "name")

.saveAsTable("users\_partitioned\_bucketed")

val newType = StructType().add("id", IntType).add("name", StringType)

val staticInputDF =

spark

.read

.schema(jsonSchema)

.json(inputPath)

val jsonSchema = new StructType().add("time", TimestampType).add("action", StringType)