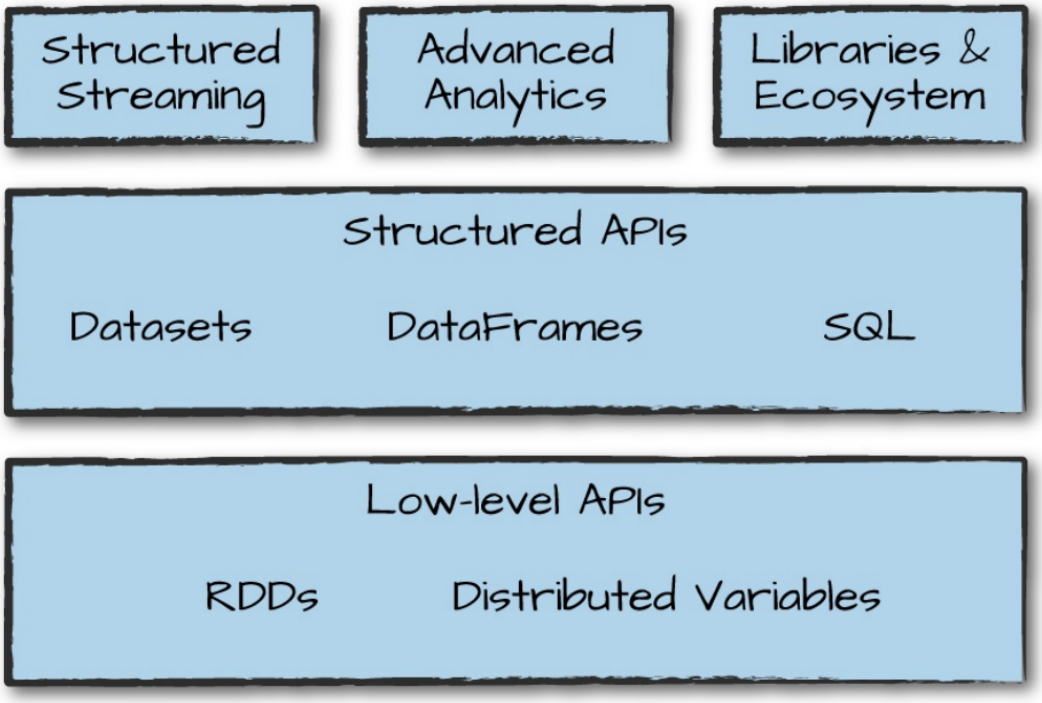
Part I. Structure Overview



提供计算和多源数据获取接口，集中计算，不提供存储。管理和协调执行跨计算机集群执行数据任务。

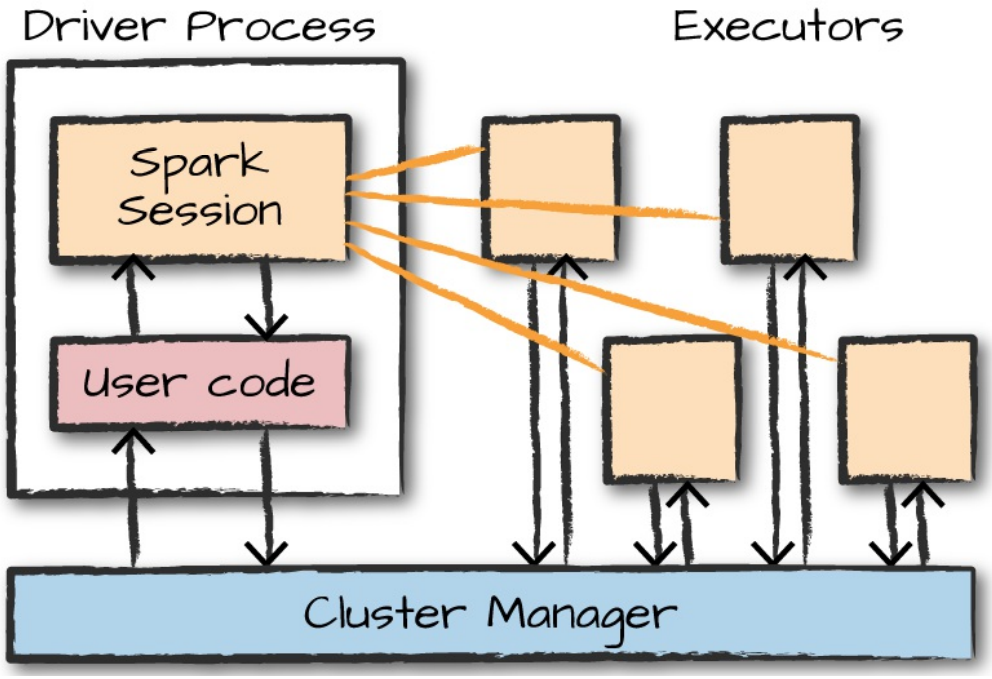
Spark's standalone cluster manager, YARN, or Mesos

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

driver process is absolutely essential—it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things: **maintaining information** about the Spark Application; **responding to a user's program or input**; and **analyzing, distributing**, **and scheduling work** across the executors.

The **executors** are responsible for **actually carrying out the work** that the driver assigns them



cluster managers: Spark's standalone cluster manager, YARN, or Mesos.

The driver and executors are simply **processes**.

The user can specify how many executors should fall on each node through configurations.

每个节点拥有多个executors

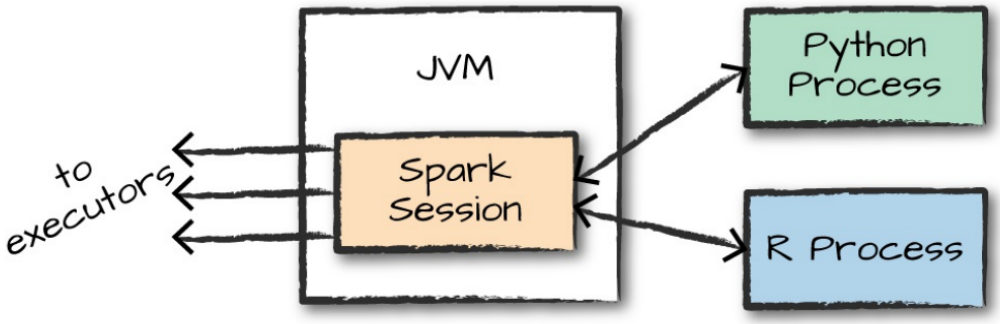


Figure 2-2. The relationship between the SparkSession and Spark's Language API

SparkSession object available to the user, which is the **entrance point** to running Spark code.

Spark has two fundamental sets of APIs: the low-level "unstructured" APIs（RDD）, and the higher-level structured APIs.

> **SparkContext**是低级API函数库的入口，无论是DataFrame还是Dataset，运行的所有Spark代码实际都将编译成一个RDD。**简单来说，RDD是一个只读不可变且已分块的记录集合，并可以被并行处理。RDD与DataFrame不同，DataFrame中每个记录就是一个结构化的数据行，各字段已知且schema已知，而RDD中的记录仅仅是程序员选择的Java、Scala 或Python 对象**，正因为RDD中每个记录仅仅是一个Java或Python 对象，因此能以任何格式在这些RDD对象中存储任何内容，这使用户具有很大的控制权，同时也带来一些潜在问题：比如值之间的每个操作和交互都必须手动定义，也就是说，无论实现什么任务，都必须从底层开发。另外，因为Spark不像对结构化API那样清楚地理解记录的内部结构，所以往往需要用户自己写优化代码。

```
val spark = SparkSession.builder().master("local").appName("price analysis")
    .config("dfs.client.use.datanode.hostname", "true").getOrCreate()
```

```
SparkSession spark = SparkSession
                .builder()
                .master("local[*]")
                .appName("Chapter2AGentleIntorductionToSpark")
                .getOrCreate();
```

Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). These different abstractions all represent **distributed collections of data.** (either the data is too large to fit on one machine or it would simply take too long to perform that computation on one machine)

> **DataFrames** 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

Partitions

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

重点：如何将数据尽可能平均地分配（数据倾斜问题）

If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors. If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource.

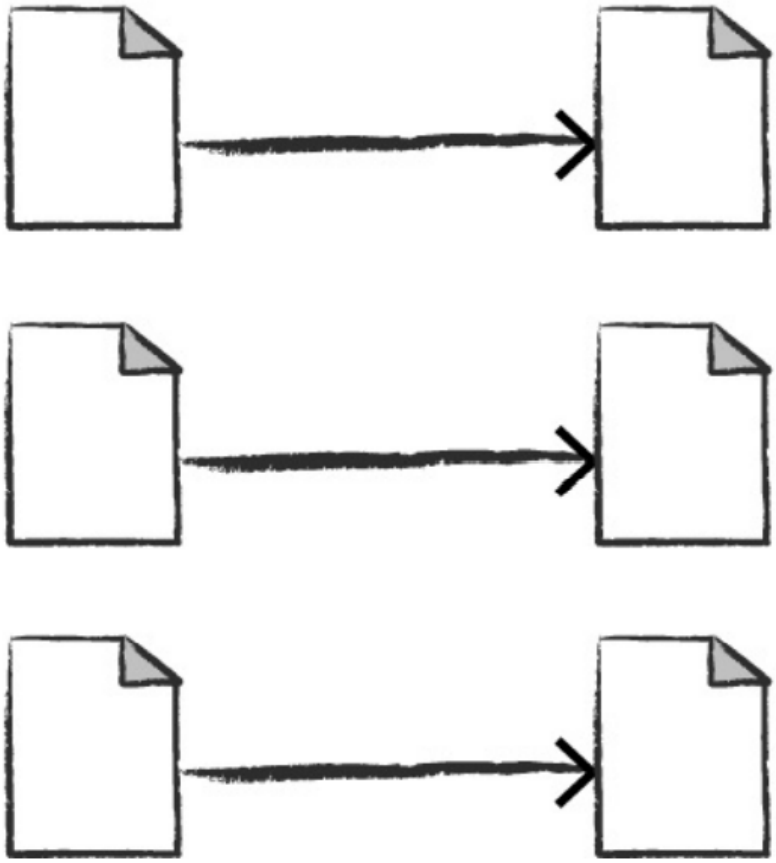
Transformations

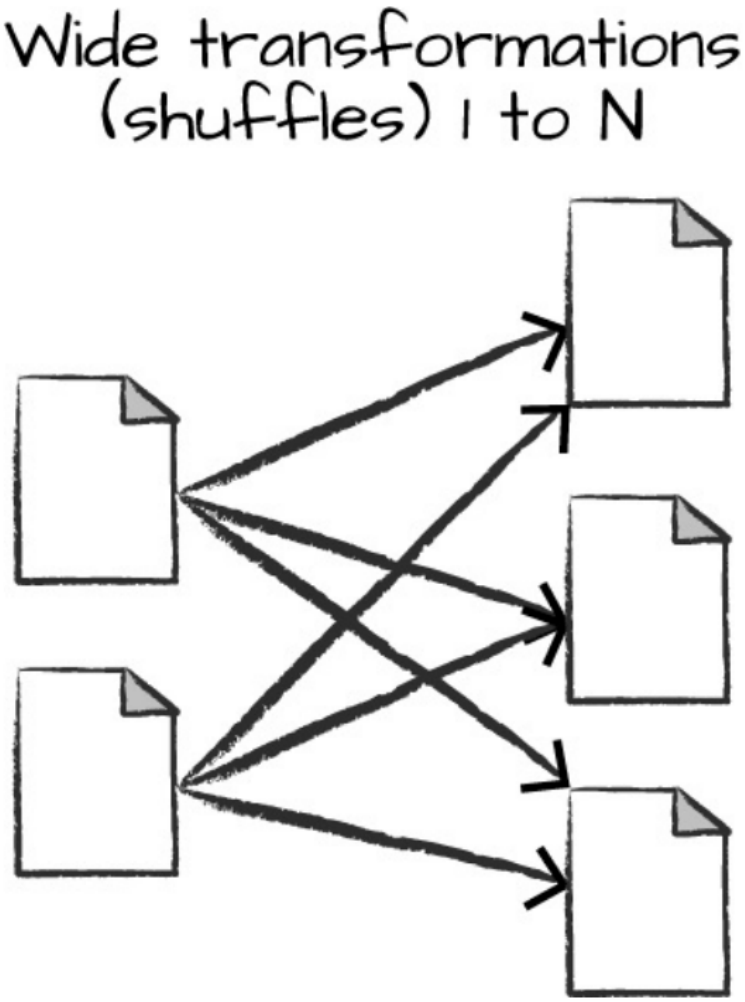
In Spark, the core data structures are **immutable,** meaning they cannot be changed after they're created.

To "change" a DataFrame, you need to instruct Spark how you would like to modify it to do what you want. These instructions are called transformations.

```
// in Scala
val divisBy2 = myRange.where("number % 2 = 0")
```

Narrow transformations
1 to 1





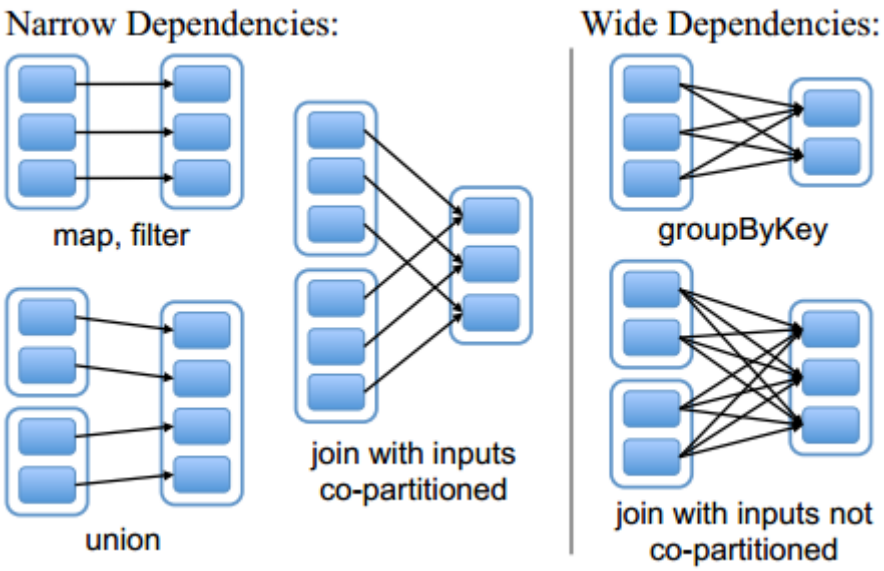
- 窄依赖(narrow dependencies): 子RDD的每个分区依赖于常数个父分区（即与数据规模无关）
- 宽依赖（wide dependencies）：父RDD被多个子RDD所用。例如，map产生窄依赖，而join则是宽依赖（除非父RDD被哈希分区）

窄依赖的函数有：map, filter, union, join(父RDD是hash-partitioned), mapPartitions, mapValues

宽依赖的函数有：groupByKey, join(父RDD不是hash-partitioned), partitionBy

窄依赖允许在一个集群节点上以流水线的方式（pipeline）计算所有父分区。例如，逐个元素地执行map、然后filter操作；而宽依赖则需要首先计算好所有父分区数据，然后在节点之间进行Shuffle，这与MapReduce类似。第二，窄依赖能够更有效地进行失效节点的恢复，即只需重新计算丢失RDD分区的父分区，而且不同节点之间可以并行计算；

如果父分区能够被一个以上子分区使用的，也就是说**用到shuffle过程的，那就是宽依赖**；如果没有shuffle过程，那就是窄分区



Lazy Evaluation

Lazy evaulation means that Spark will wait until the very last moment to execute the graph of computation instructions.

spark直到action 动作之前，数据不会先被计算；（spark的算子中存在action和transform两种，transform就是常见的map，union，flatmap，groupByKey, join等不需要系统返回啥的算子。而collect，count，reduce等需要拉回产生结果的算子就是action算子，可以简单的说，action算的的个数是job提交的个数

19

For transformations, Spark adds them to a DAG of computation and only when driver requests some data, does this DAG actually gets executed.

One advantage of this is that Spark can make many optimization decisions after it had a chance to look at the DAG in entirety. This would not be possible if it executed everything as soon as it got it.

For example -- if you executed every transformation eagerly, what does that mean? Well, it means you will have to materialize that many intermediate datasets in memory. This is evidently not efficient -- for one, it will increase your GC costs. (Because you're really not interested in those intermediate results as such. Those are just convnient abstractions for you while writing the program.) So, what you do instead is -- you tell Spark what is the eventual answer you're interested and it figures out best way to get there.

share improve this answer

edited Jun 25 '16 at 12:32

answered Jun 25 '16 at 12:09

Sachin Tyagi

1,991 ● 8 ● 23

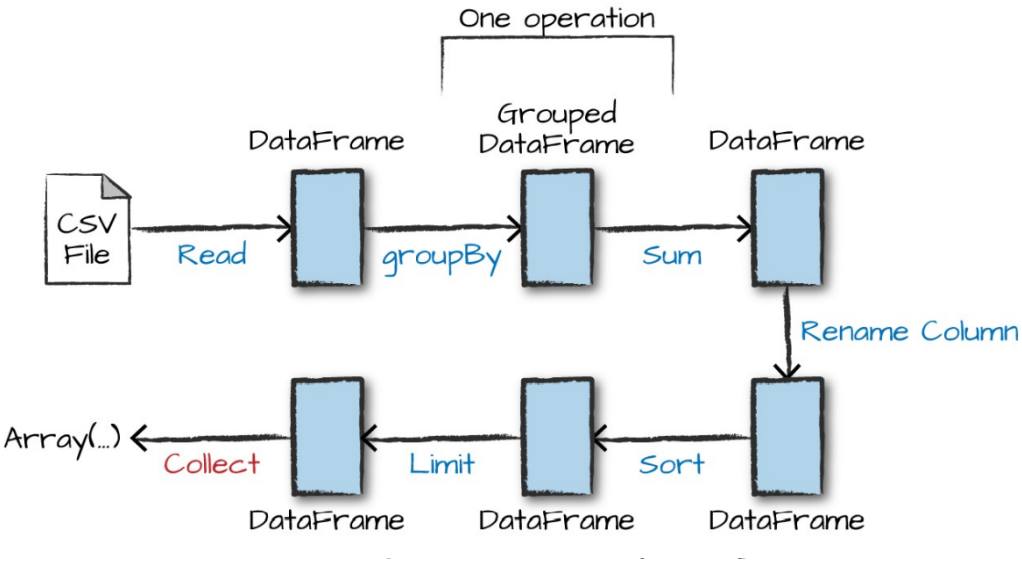
add a comment

> 其中一个case翻译一下就是，假设有一个需求是需要先加载一下一天的订单信息，如果不是lazy的特性，spark会先根据你的需求把这一天的订单信息加载到内存中(太大就溢写)，然后你需求又变更了，只想看一下这一天的订单的基本样式，取个一条就可以了，然后spark对内存中的数据取一条吐出来；其实这样是很低效的，最终诉求其实可以理解为，取出订单的第一条数据看看；我们把第一个需求（取出所有订单），第二个需求（订单中的一条）当做两个transform算子，那么他们会最终成为一个链chain；如果操作再多一些，就会形成DAG，这样spark会理解整个链路以及最后的需求后，优化整个DAG，以消耗最少资源的情况下满足需求，这就是lazy特性带来的好处，并不是立刻执行，而是see the big picture，概览全局后，在最后一顿骚操作优化，然后再执行计算；虽然mapreduce也可以实现，但是对开发人员成本比较高，需要写代码去规避这些资源浪费；而spark自己自动进行优化

Action

An action instructs Spark to compute a result from a series of transformations.

DataFrames and SQL



The first step is to read in the data. We defined the DataFrame previously but, as a reminder, Spark does **not actually read it in until an action** is called on that DataFrame or one derived from the original DataFrame.

```
import org.apache.spark.sql.functions.max
source                                                                          : DataFrame
  .groupBy( col1 = "DEST_COUNTRY_NAME")                                         : RelationalGroupedDataset
  .sum( colNames = "count")                                                     : DataFrame
  .withColumnRenamed( existingName = "sum(count)",  newName = "destination_total") : DataFrame
  .sort(desc( columnName = "destination_total"))                                : Dataset[Row]
  .limit(5)                                                                     : Dataset[Row]
  .show()                                                                       : Unit
```

The second step is grouping， perform an aggregation over each one of those keys. It's an **action.**

the third step is to specify the aggregation. It’s important to reinforce (again!) that **no computation** has been performed. **This is simply another transformation** that we’ve expressed, and Spark is simply able to trace our type information through it.**(?)**

The fourth step is a simple renaming. this doesn’t perform computation: this is just another **transformation**!

The fifth step sorts the data, Penultimately, we’ll specify a limit. This just specifies that we only want to return the first five values in our final DataFrame instead of all the data. this two are **transformations**.

The last step is our **action**! Now we actually begin the process of **collecting the results** of our DataFrame, and Spark will give us back a list or array in the language that we're executing.

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#35L DESC NULLS LAST], output=[DEST_COUNTRY_NAME#16,destination_total#35L])
+- *(2) HashAggregate(keys=[DEST_COUNTRY_NAME#16], functions=[sum(cast(count#18 as bigint))])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#16, 200), true, [id=#41]
      +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#16], functions=[partial_sum(cast(count#18 as bigint))])
         +- FileScan csv [DEST_COUNTRY_NAME#16,count#18] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[file:/D:/Project/Spark_Project/src/data/flight-data/csv/2
```

High API

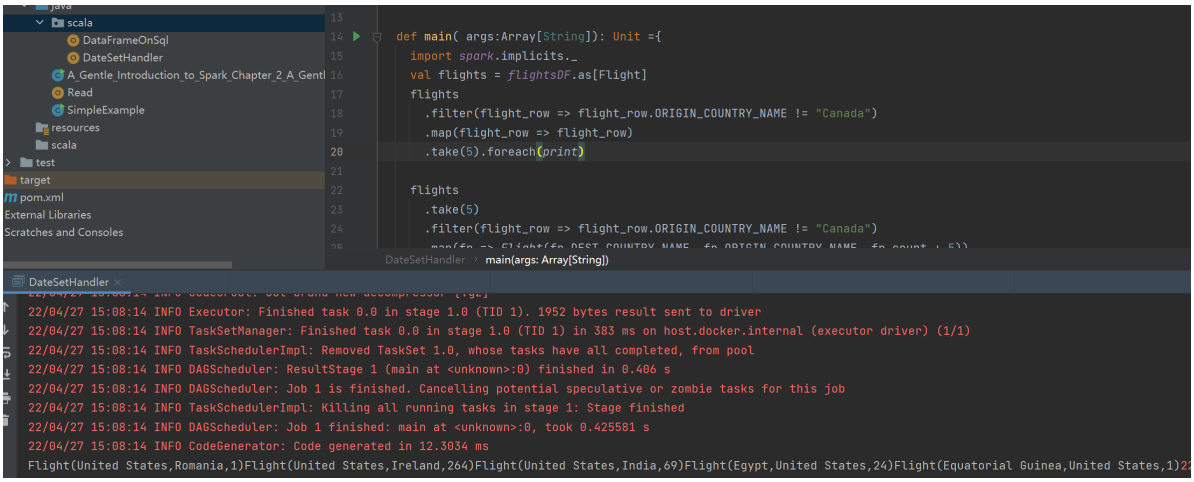
Datasets: Type-Safe Structured APIs

static , only used in java and scala

DataFrames are a **distributed collection** of objects of **type Row** that can hold various types of tabular data.

The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of **typed objects, similar to a Java ArrayList** or Scala Seq.

The APIs available on Datasets are **type-safe**, meaning that you cannot accidentally view the objects in a Dataset as being of another class than the class you put in initially.(类似泛型的概念)



Structured Streaming

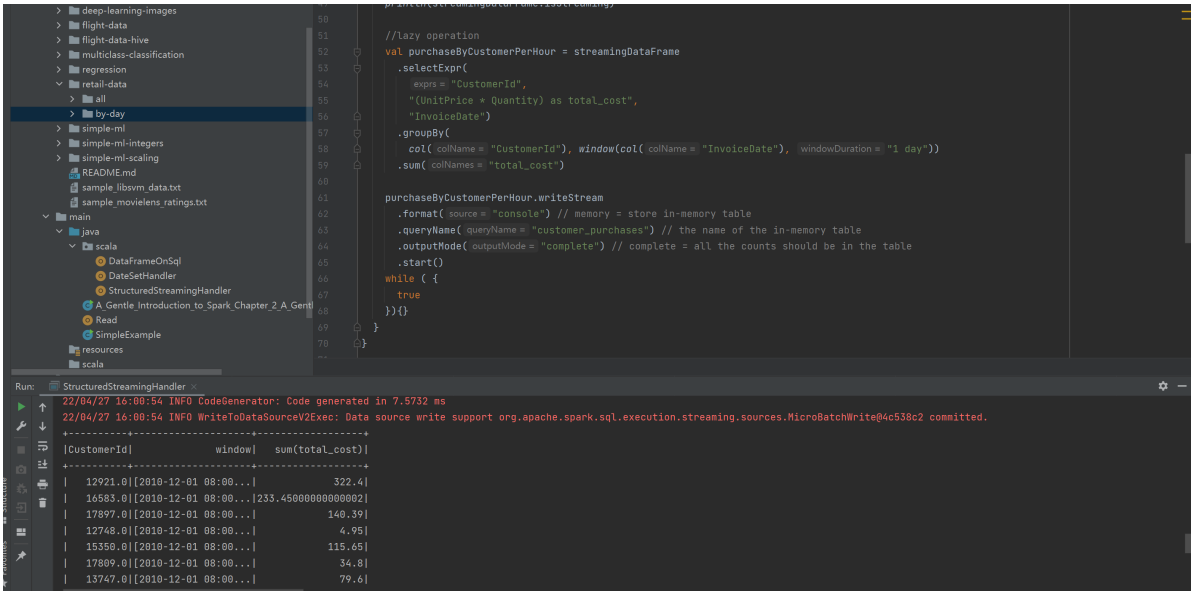
Structured Streaming is that it allows you to rapidly and quickly extract value out of streaming systems with virtually no code changes.

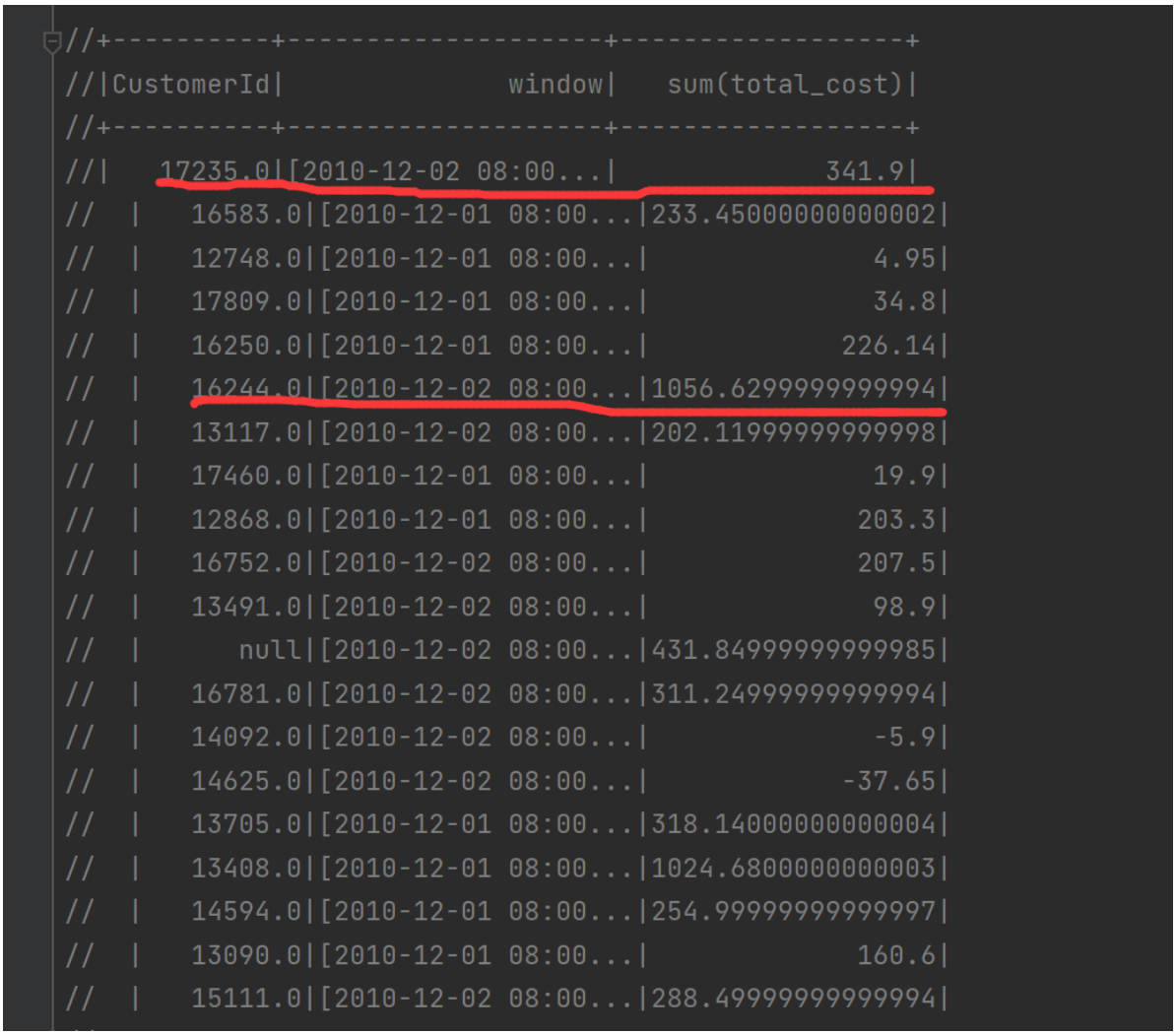
Streaming **actions** are a bit different from our conventional static action because we're going to be populating data somewhere instead of just calling something like count

The action we will use will **output to an in-memory table** that we will **update after each trigger**.

In this case, each trigger is based on an **individual file** (the read option that we set)

类似线程不断访问目标文件夹是否有最新数据





ML

There are always two types for every algorithm in MLlib's DataFrame API. They follow the naming pattern of **Algorithm**, for the untrained version, and **AlgorithmModel** for the trained version. In our example, this is KMeans and then KMeansModel.

Lower-Level API

Spark includes a number of lower-level primitives to allow for arbitrary Java and Python object manipulation via Resilient Distributed Datasets (RDDs).

There are some things that you might use RDDs for, especially when you're reading or manipulating **raw data**, but for the most part you should stick to the Structured APIs.

```
//Implicit methods available in Scala for converting common Scala objects into
DataFrames.
import spark.implicits._
spark.sparkContext.parallelize(Seq(1, 2, 3)).toDF()
```

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

Structured API Overview

The majority of the Structured APIs apply to **both batch and streaming computation**.

Spark is a distributed programming model in which the user specifies transformations. **Multiple transformations build up a directed acyclic graph of instructions**. An action begins the process of executing that graph of instructions, as a single job, by breaking it down into **stages and tasks to execute across the cluster**. The logical structures that we manipulate with transformations and actions are DataFrames and Datasets. **To create a new DataFrame or Dataset, you call a transformation. To start computation or convert to native language types, you call an action.**

Datasets and DataFrames

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(**can use null** to specify the absence of a value)

Each column has type information that must be consistent for every row in the collection

DataFrames and Datasets represent **immutable, lazily evaluated** plans that specify what operations to apply to data residing at a location to generate some output.

When we perform an action on a DataFrame, we instruct Spark to perform the actual transformations and **return** the result.

Schemas

A schema defines the column names and types of a DataFrame.

You can define schemas manually or read a schema from a data source (often called schema on read)

```
    //声明Rdd对应的schema
    val schema = new StructType().add(StructField("date", StringType,true))

.add(StructField("market_count",LongType,true)).add(StructField("variety_count",
LongType,true))

.add(StructField("category_count",LongType,true)).add(StructField("total_count",
LongType,true))
    //Rdd(纯数据) + schema -------->具有格式、包含数据的DataFrame
    var top5DF = spark.createDataFrame(top5Rdd,schema)
```

```
//read from a data source
val staticDataFrame = spark.read.format("csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load("src/data/retail-data/by-day/*.csv")
```

Overview of Structured Spark Types

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. Even if we use Spark's Structured APIs from Python or R, the majority of our manipulations will operate strictly on Spark types, not Python types.

DataFrames Versus DateSet

"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**.

```
  case class Flight(DEST_COUNTRY_NAME: String,
                    ORIGIN_COUNTRY_NAME: String,
                    count: BigInt)
val flightsDF = spark.read.parquet("src/data/flight-data/parquet/2010-
summary.parquet/")
val flights = flightsDF.as[Flight]
```

```
Dataset<Row> flightData2015 = spark
                .read()
                .option("inferSchema", "true")
                .option("header", "true")
                .csv("src/data/flight-data/csv/2015-summary.csv");

        // Taking 3 rows from the flight dataset
        Object [] dataObjects = (Object[]) flightData2015.take(3);
        for(Object object: dataObjects) {
            System.out.println(object);
        }
```

DataFrames are simply Datasets of Type Row.

The “Row” type is Spark’s internal representation of its optimized in-memory format for computation.

This format makes for highly specialized and efficient computation because rather than using **JVM types, which can cause high garbage-collection and object instantiation costs**, Spark can operate on its own internal format without incurring any of those costs.

Columns

Columns represent a **simple type** like an integer or string, a complex type like an array or map, or a null value.

For the most part you can think about Spark Column types as columns in a table

Rows

A row is nothing more than a record of data.

Each record in a DataFrame must be of type Row.

Spark Types

how we instantiate, or declare, a column to be of a certain type.

```
import org.apache.spark.sql.types._
val b = ByteType
```

```
import org.apache.spark.sql.types.DataTypes;
//必须强转，书中有误
ByteType x = (ByteType) DataTypes.ByteType;
```

```
public class DataTypes {
    public static final DataType StringType;
    public static final DataType BinaryType;
    public static final DataType BooleanType;
    public static final DataType DateType;
    public static final DataType TimestampType;
    public static final DataType CalendarIntervalType;
    public static final DataType DoubleType;
    public static final DataType FloatType;
    public static final DataType ByteType;
    public static final DataType IntegerType;
```

```
    public static final DataType LongType;
    public static final DataType ShortType;
    public static final DataType NullType;

    static {
        StringType = org.apache.spark.sql.types.StringType..MODULE$;
        BinaryType = org.apache.spark.sql.types.BinaryType..MODULE$;
        BooleanType = org.apache.spark.sql.types.BooleanType..MODULE$;
        DateType = org.apache.spark.sql.types.DateType..MODULE$;
        TimestampType = org.apache.spark.sql.types.TimestampType..MODULE$;
        CalendarIntervalType =
org.apache.spark.sql.types.CalendarIntervalType..MODULE$;
        DoubleType = org.apache.spark.sql.types.DoubleType..MODULE$;
        FloatType = org.apache.spark.sql.types.FloatType..MODULE$;
        ByteType = org.apache.spark.sql.types.ByteType..MODULE$;
        IntegerType = org.apache.spark.sql.types.IntegerType..MODULE$;
        LongType = org.apache.spark.sql.types.LongType..MODULE$;
        ShortType = org.apache.spark.sql.types.ShortType..MODULE$;
        NullType = org.apache.spark.sql.types.NullType..MODULE$;
    }
}
```

Table 4-3. Java type reference

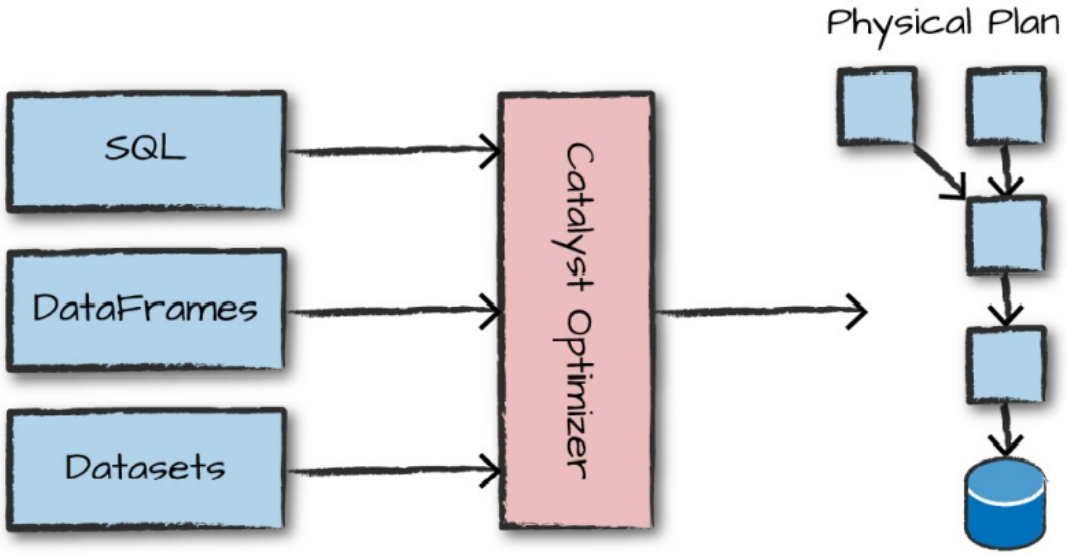
Data type	Value type in Java	API to access or create a data type
ByteType	byte or Byte	DataTypes.ByteType
ShortType	short or Short	DataTypes.ShortType
IntegerType	int or Integer	DataTypes.IntegerType
LongType	long or Long	DataTypes.LongType
FloatType	float or Float	DataTypes.FloatType
DoubleType	double or Double	DataTypes.DoubleType
DecimalType	java.math.BigDecimal	DataTypes.createDecimalType() DataTypes.createDecimalType(precision, scale).
StringType	String	DataTypes.StringType
BinaryType	byte[]	DataTypes.BinaryType
BooleanType	boolean or Boolean	DataTypes.BooleanType
TimestampType	java.sql.Timestamp	DataTypes.TimestampType
DateType	java.sql.Date	DataTypes.DateType
ArrayType	java.util.List	DataTypes.createArrayType(elementType). Note: The value of containsNull will be true DataTypes.createArrayType(elementType, containsNull).
MapType	java.util.Map	DataTypes.createMapType(keyType, valueType). Note: The value of valueContainsNull will be true. DataTypes.createMapType(keyType, valueType, valueContainsNull)
StructType	org.apache.spark.sql.Row	DataTypes.createStructType(fields). Note: fields is a List or an array of StructFields. Also, two fields with the same name are not allowed.
StructField	The value type in Java of the data type of this field (for example, int for a StructField with the data type IntegerType)	DataTypes.createStructField(name, dataType, nullable)

Overview of Structured API Execution

The execution of a single structured API query from user code to executed code.

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 Planning

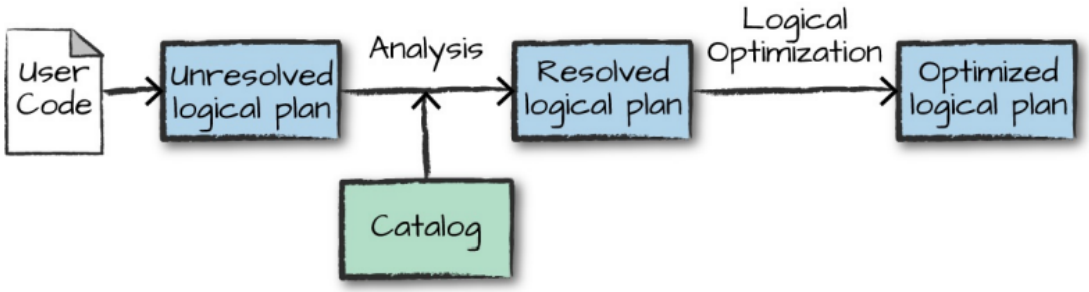


Figure 4-2. The structured API logical planning process

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**. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections.

Physical Planning

After successfully creating an optimized logical plan, Spark then begins the physical planning process. 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.

An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are).

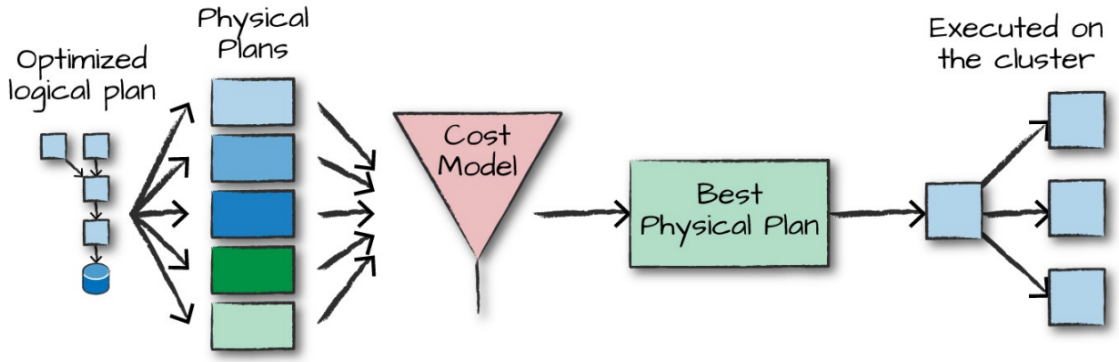


Figure 4-3. The physical planning process

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. 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.

conclusion

Deep Dive into Spark SQL's Catalyst Optimizer - The Databricks Blog

At the core of Spark SQL is the **Catalyst optimizer**, which leverages advanced programming language features

Catalyst's extensible design had two purposes.

- First, we wanted to make it easy to **add new optimization techniques and features to Spark SQL**, especially for the purpose of tackling various problems we were seeing with big data (e.g., semistructured data and advanced analytics).
- Second, we wanted to **enable external developers to extend the optimizer** — for example, by adding data source specific rules that can push filtering or aggregation into external storage systems, or support for new data types. Catalyst supports both rule-based and cost-based optimization.

Catalyst contains a general library for representing **trees** and applying **rules** to manipulate them.

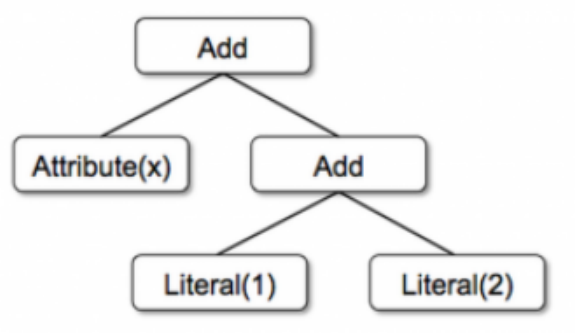
The main data type in Catalyst is **a tree composed of node objects**. Each node has a node type and zero or more children. New node types are defined in Scala as subclasses of the TreeNode class. These objects are **immutable and can be manipulated using functional transformations**.

As a simple example, suppose we have the following three node classes for a very simple expression language:

- `Literal(value: Int)`: a constant value
- `Attribute(name: String):` an attribute from an input row, e.g.,"x"
- `Add(left: TreeNode, right: TreeNode):` sum of two expressions.

These classes can be used to build up trees; for example, the tree for the expression `x+(1+2)`, would be represented in Scala code as follows:

```
Add(Attribute(x), Add(Literal(1), Literal(2)))
```



Trees can be manipulated using **rules**, which are functions from a tree to another tree.

the most common approach is to **use a set of pattern matching functions that find and replace subtrees with a specific structure.**

Pattern matching is a feature of many functional languages that allows **extracting values from potentially nested structures** of algebraic data types.

In Catalyst, trees offer a transform method that applies a pattern matching function recursively on all nodes of the tree, transforming the ones that match each pattern to a result.

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

Applying this to the tree for `x+(1+2)` would yield the new tree `x+3`. The `case` keyword here is Scala's standard pattern matching syntax, and **can be used to match on the type of an object** as well as **give names to extracted values (`c1` and `c2` here).**

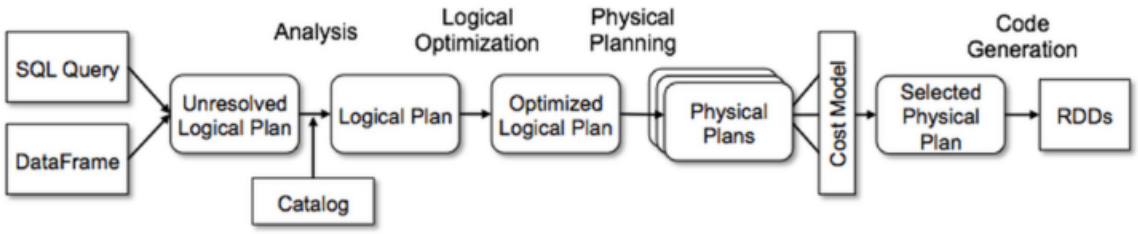
Rules (and Scala pattern matching in general) can match multiple patterns in the same transform call, making it very concise to implement multiple transformations at once:

```
tree.transform {
case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
case Add(left, Literal(0)) => left
case Add(Literal(0), right) => right
}
```

In practice, rules may need to execute multiple times to fully transform a tree.

Catalyst groups rules into batches, and **executes each batch until it reaches a fixed point**, that is, until the tree stops changing after applying its rules. often written via **recursive matching**

functional transformations on immutable trees make the whole optimizer very easy to reason about and debug.



(1) analyzing a logical plan to resolve references, (2) logical plan optimization, (3) physical planning, and (4) code generation to compile parts of the query to Java bytecode.

Analysis

In both cases, the relation may contain unresolved attribute references or relations: for example, in the SQL query

`SELECT col FROM sales`

the type of col, or even whether it is a valid column name, is not known until we look up the table sales.

An attribute is called **unresolved** if we do not know its type or have not matched it to an input table (or an alias).

Spark SQL uses **Catalyst rules** and a **Catalog object** that tracks the tables in all data sources to resolve these attributes. It starts by **building an "unresolved logical plan" tree with unbound attributes and data types**, then applies rules that do the following:

- Looking up relations by name from the catalog.
- Mapping named attributes, such as col, to the input provided given operator's children.
- Determining which attributes refer to the same value to give them a unique ID (which later allows optimization of expressions such as `col = col`).
- Propagating and coercing types through expressions: for example, we cannot know the return type of `1 + col` until we have resolved col and possibly casted its subexpressions to a compatible types.

Logical Optimizations

The logical optimization phase applies standard rule-based optimizations to the logical plan.

These include constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules.

For example, when we added the fixed-precision DECIMAL type to Spark SQL, we wanted to optimize aggregations such as sums and averages on DECIMALs with small precisions; it took 12 lines of code to write a rule that finds such decimals in SUM and AVG expressions, and casts them to unscaled 64-bit LONGs, does the aggregation on that, then converts the result back. A simplified version of this rule that only optimizes SUM expressions is reproduced below:

```
object DecimalAggregates extends Rule[LogicalPlan] {
  /** Maximum number of decimal digits in a Long */
  val MAX_LONG_DIGITS = 18
  def apply(plan: LogicalPlan): LogicalPlan = {
    plan transformAllExpressions {
      case Sum(e @ DecimalType.Expression(prec, scale))
          if prec + 10 <= MAX_LONG_DIGITS =>
        MakeDecimal(Sum(UnscaledValue(e)), prec + 10, scale) }
}
```

Physical Planning

In the physical planning phase, Spark SQL takes a logical plan and generates one or more physical plans, using physical operators that match the Spark execution engine. It then selects a plan using a **cost model**. cost-based optimization is only used to select **join algorithms**: for relations that are known to be small, Spark SQL uses a broadcast join, using a peer-to-peer broadcast facility available in Spark.

Code Generation

The final phase of query optimization involves generating Java bytecode to run on each machine.

Spark SQL often operates on **in-memory datasets,** where processing is CPU-bound, we wanted to support code generation to speed up execution.

Catalyst relies on a special feature of the Scala language, **quasiquotes, to make code generation simpler**.

Quasiquotes allow the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode.

We use **Catalyst** to transform a tree representing an expression in SQL to an **AST** for Scala code to evaluate that expression, and then compile and run the generated code.

With code generation, we can write a function to translate a specific expression tree to a Scala AST as follows:

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) => q"${compile(left)} + ${compile(right)}"
}
```

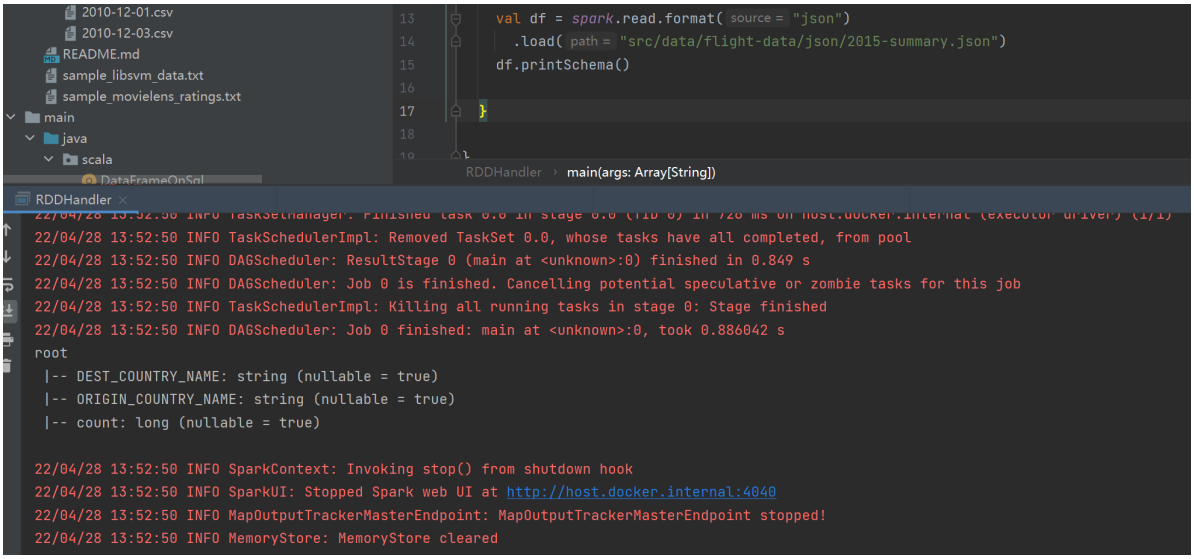
The strings beginning with `q` are quasiquotes, meaning that although they look like strings, they are parsed by the Scala compiler at compile time and represent ASTs for the code within. Quasiquotes can have variables or other ASTs spliced into them, indicated using `$` notation. For example, `Literal(1)` would become the Scala AST for 1, while `Attribute("x")` becomes `row.get("x")`. In the end, a tree like `Add(Literal(1), Attribute("x"))` becomes an AST for a Scala expression like `1+row.get("x")`.

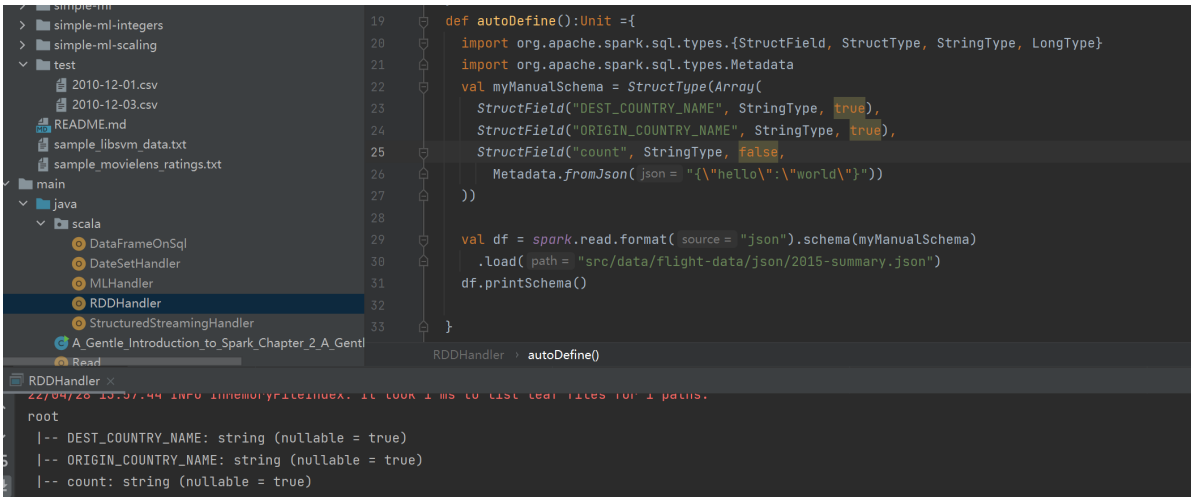
quasiquotes在编译时进行类型检查，以确保只替换适当的AST，这使得它们比字符串连接更有用，并且它们直接生成Scala AST，而不是在运行时运行Scala解析器。此外，它们是高度可组合的，因为每个节点的代码生成规则不需要知道其他子节点返回的树是如何构造的。最后，Scala编译器会进一步优化生成的代码，以防Catalyst遗漏了表达式的优化。quasiquotes让我们生成的代码的性能可以媲美手动调优的程序。

Basic Structured Operation

Partitioning of the DataFrame defines the layout of the DataFrame or Dataset's **physical distribution across the cluster**.

When using Spark for production Extract, Transform, and Load (ETL), it is often a good idea to define your schemas **manually,** especially when working with untyped data sources like CSV and JSON because **schema inference can vary depending on the type of data that you read in.**





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.

```
$"myColumn" equals to 'myColumn
```

An expression is a set of transformations on one or more values in a record in a DataFrame.

expr("someCol - 5") is the same transformation as performing col("someCol") - 5, or even expr("someCol") - 5. That's because Spark compiles these to a logical tree specifying the order of operations. This might be a bit confusing at first, but remember a couple of key points:

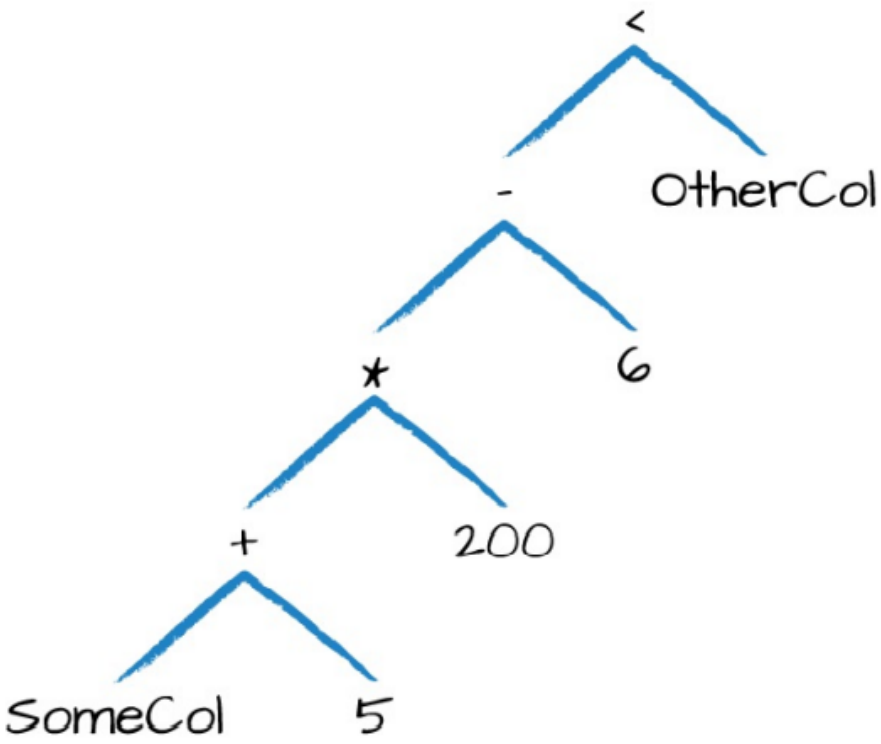
Columns are just expressions.

Columns and transformations of those columns compile to the **same logical plan** as parsed expressions.

Let's ground this with an example:

```
(((col("someCol") + 5) * 200) - 6) < col("otherCol")
```

Figure 5-1 shows an overview of that logical tree.




```
// in Scala equal to above
import org.apache.spark.sql.functions.expr
expr("(((someCol + 5) * 200) - 6) < otherCol")
```

This is an extremely important point to reinforce,SQL expression and the previous DataFrame code compile to the same underlying logical tree prior to execution. This means that you can write your expressions as DataFrame code or as SQL expressions and get the exact same performance characteristics.

DataFrame Transformations

- We can add rows or columns
- We can remove rows or columns
- We can transform a row into a column (or vice versa)
- We can change the order of rows based on the values in columns

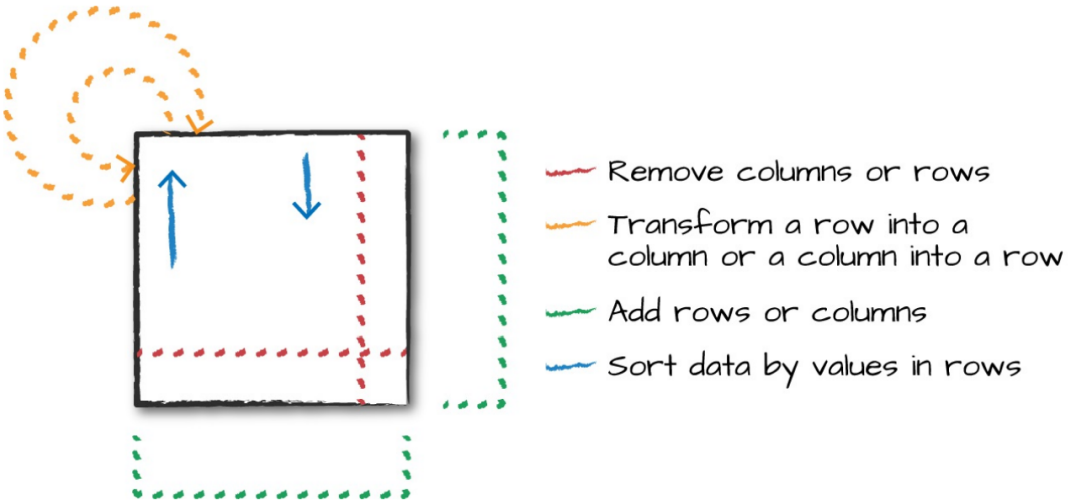
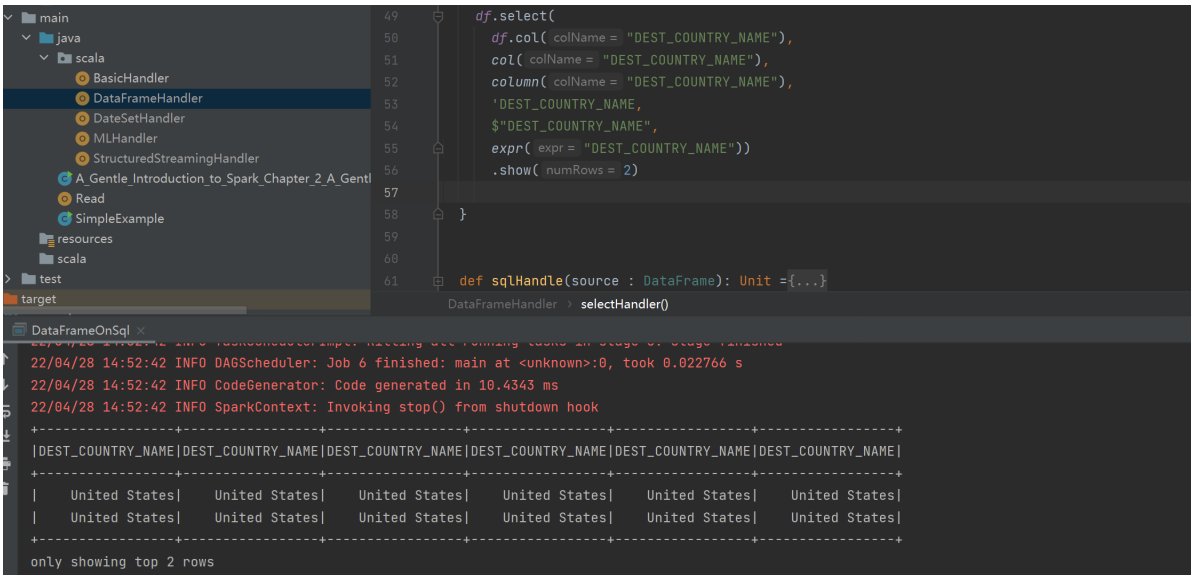
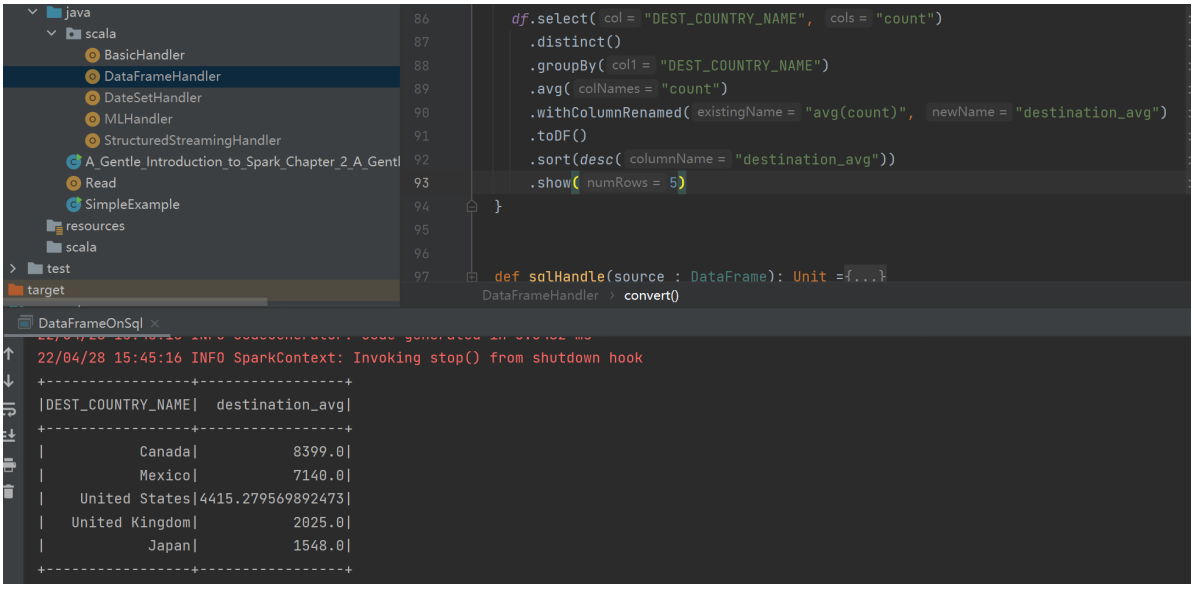


Figure 5-2. Different kinds of transformations




```
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)
df.select(col("DEST_COUNTRY_NAME"))
  .withColumnRenamed("DEST_COUNTRY_NAME", "destination")
  .show(2)
df.select(expr("DEST_COUNTRY_NAME").alias("destination"))
  .show(2)
df.selectExpr("DEST_COUNTRY_NAME as destination").show(2)
```



One thing that you might come across is reserved characters like spaces or dashes in column names. Handling these means escaping column names appropriately. In Spark, we do this by using backtick (`) characters.

```
import org.apache.spark.sql.functions.expr
    val dfWithLongColName = df.withColumn(
      "This Long Column-Name",
      expr("ORIGIN_COUNTRY_NAME"))

    dfWithLongColName.selectExpr(
      "`This Long Column-Name`",
      "`This Long Column-Name` as `new col`")
      .show(2)
    //another way
    dfWithLongColName.select(col("This Long Column-Name"), col("This Long
Column-Name")
      .as("new col")).show(2)
```

default Spark is **case insensitive**; however, you can make Spark case sensitive by setting the configuration:

```
set spark.sql.caseSensitive true
```

To filter rows, we create an expression that evaluates to true or false. You then filter out the rows with an expression that is equal to false.

```
    df.filter(col("count").%(2).equalTo(0)).show(2)
    df.where("count % 2 == 0").filter("ORIGIN_COUNTRY_NAME !=
'Ireland'").show(2)

    df.where(col("count") <
2).where(!col("ORIGIN_COUNTRY_NAME").equalTo("Croatia"))
      .show(2)
    df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") =!= "Croatia")
      .show(2)
```

```
df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct().count()
```

```
    val dataFrames = df.randomSplit(Array(0.25, 0.75), seed)
    //val dataFrames = df.randomSplit(Array(1, 3), seed)  #equals to above
    dataFrames(0).show(5)
    dataFrames(1).show(5)
    //+-----------------+-------------------+-----+
```

```
    //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
    //+-----------------+-------------------+-----+
    //|          Algeria|      United States|    4|
    //|        Argentina|      United States|  180|
    //|          Belgium|      United States|  259|
    //|           Canada|      United States| 8399|
    //|   Cayman Islands|      United States|  314|
    //+-----------------+-------------------+-----+
    //+-------------------+-------------------+-----+
    //|  DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
    //+-------------------+-------------------+-----+
    //|             Angola|      United States|   15|
    //|           Anguilla|      United States|   41|
    //|Antigua and Barbuda|      United States|  126|
    //|              Aruba|      United States|  346|
    //|          Australia|      United States|  329|
    //+-------------------+-------------------+-----+
```

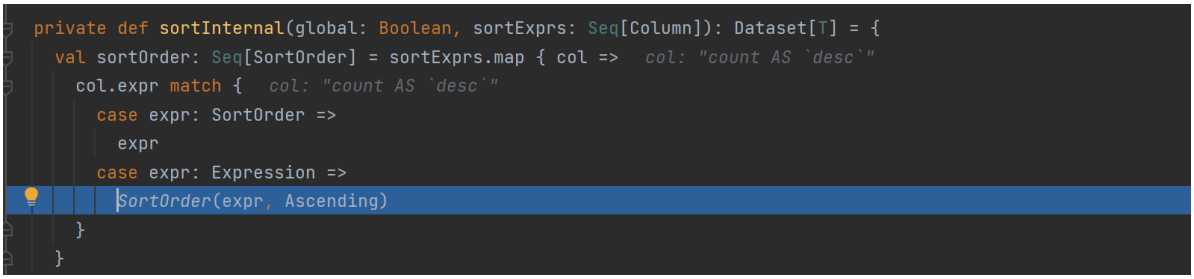
An advanced tip is to use **asc_nulls_first, desc_nulls_first, asc_nulls_last, or desc_nulls_last** to specify where you would like your null values to appear in an ordered DataFrame.

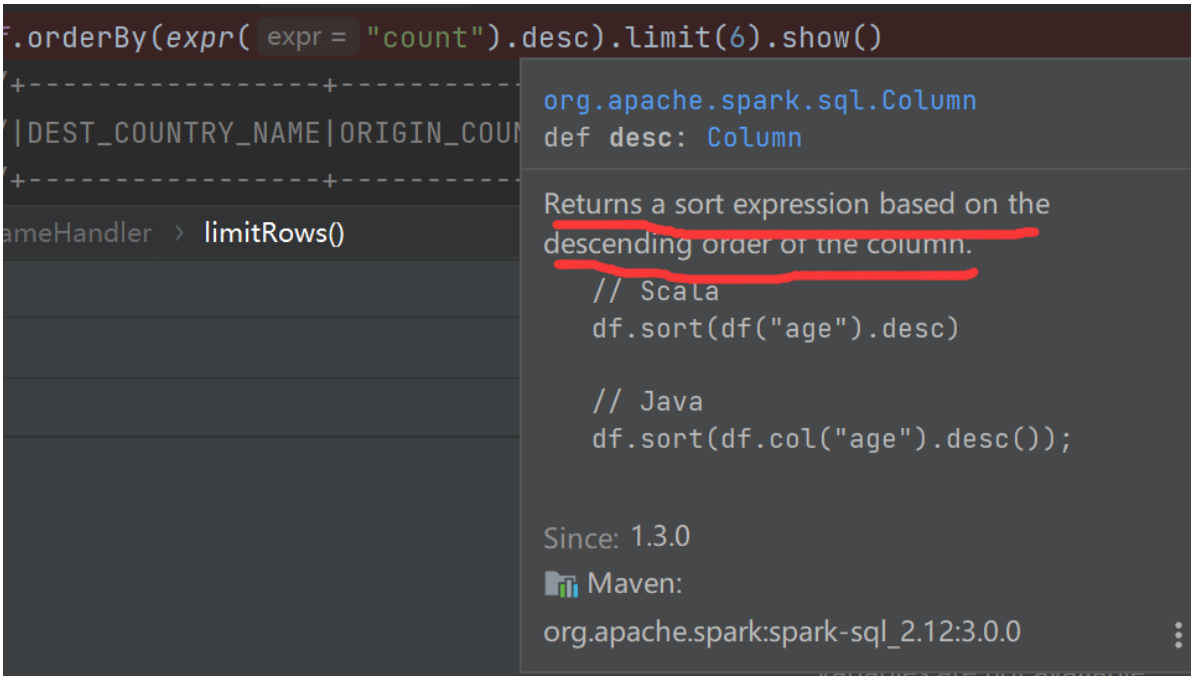
For optimization purposes, it's sometimes advisable to sort within each partition before **another set of transformations**. You can use the sortWithinPartitions method to do this:

```
    spark.read.format("json").load("src/data/flight-data/json/*-summary.json")
      .sortWithinPartitions("count")
```

```
df.orderBy(expr("count desc")).limit(6).show() //失效
    //+-------------------+-------------------+-----+
    //|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
    //+-------------------+-------------------+-----+
    //|              Malta|      United States|    1|
    //|Saint Vincent and...|      United States|    1|
    //|       United States|            Croatia|    1|
    //|       United States|          Gibraltar|    1|
    //|       United States|          Singapore|    1|
    //|             Moldova|      United States|    1|
    //+-------------------+-------------------+-----+
    df.orderBy(desc("count")).limit(6).show()
    df.orderBy(expr("count").desc).limit(6).show()
    df.orderBy($"count".desc).limit(6).show()
    //+-----------------+-------------------+------+
    //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME| count|
    //+-----------------+-------------------+------+
    //|    United States|      United States|370002|
    //|    United States|             Canada|  8483|
    //|           Canada|      United States|  8399|
    //|    United States|             Mexico|  7187|
    //|           Mexico|      United States|  7140|
    //|   United Kingdom|      United States|  2025|
    //+-----------------+-------------------+------+
```

失效原因，最好用desc函数修饰而不是使用表达式






```
    println(expr("count desc").expr.getClass) //class
org.apache.spark.sql.catalyst.expressions.Alias
    println(expr("count").desc.expr.getClass) //class
org.apache.spark.sql.catalyst.expressions.SortOrder
```

Repartition and Coalesce

Another important optimization opportunity is to partition the data according to some frequently filtered columns, which control the physical layout of data across the cluster including the **partitioning scheme** and the **number of partitions. Repartition will incur a full shuffle of the data**, regardless of whether one is necessary. This means that you should typically only repartition when **the future number of partitions is greater than your current number of partitions** or when you are looking to **partition by a set of columns**:

Coalesce, on the other hand, **will not incur a full shuffle and will try to combine partitions**. This operation will shuffle your data into five partitions based on the destination country name, and then coalesce them (without a full shuffle):

```
    val dfPartByNum = df.repartition(5)
    println(dfPartByNum.rdd.getNumPartitions) //5

    val dfPartByCol = df.repartition(col("DEST_COUNTRY_NAME"))
    println(dfPartByCol.rdd.getNumPartitions) //200

    val dfPartByMixed = df.repartition(5, col("DEST_COUNTRY_NAME"))
    println(dfPartByMixed.rdd.getNumPartitions) //5

    val dfPartByCoalesce = df.repartition(5,
col("DEST_COUNTRY_NAME")).coalesce(2)
    println(dfPartByCoalesce.rdd.getNumPartitions) //2
```

Any **collection of data to the driver can be a very expensive operation**! If you have a large dataset and call collect, you can crash the driver. If you use toLocalIterator and have very large partitions, you can easily crash the driver node and lose the state of your application. This is also expensive because we can operate on a one-by-one basis, instead of running computation in parallel.

Working with Different Types of Data

boolean

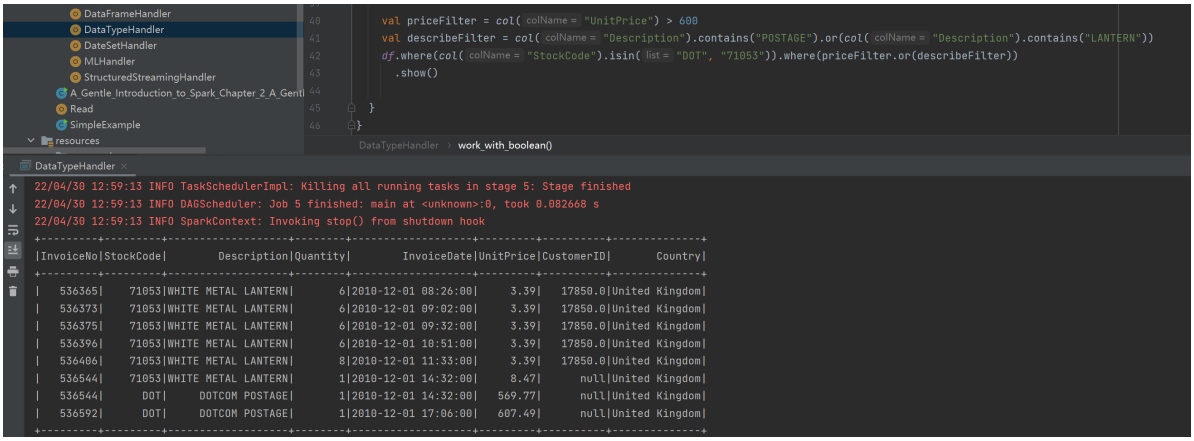
Boolean statements consist of four elements: **and, or, true, and false**.

Scala has some particular semantics regarding the use of == and ===. In Spark, if you want to filter by equality you should use **=== (equal) or =!= (not equal)**. You can also use the not function and the equalTo method.

```
def work_with_boolean(): Unit ={
  df.where(col( colName = "InvoiceNo").equalTo( other = 536365))
    .select( col = "InvoiceNo",  cols = "Description")
    .show( numRows = 5,  truncate = false)
  df.where(col( colName = "InvoiceNo") === 536365)
    .select( col = "InvoiceNo",  cols = "Description")
    .show( numRows = 5,  truncate = false)
  df.where( conditionExpr = "InvoiceNo = 536365")
    .show( numRows = 5,  truncate = false)
}
```

In Spark, you should always **chain together and filters as a sequential filter.**

Reason: they're often **easier to understand and to read** if you specify them serially.




```
val DOTCodeFilter = col("StockCode") === "DOT"
    df.withColumn("isExpensive",
DOTCodeFilter.and(priceFilter.or(describeFilter)))
      .where("isExpensive")
      .select("unitPrice", "isExpensive").show(5)
    //+---------+-----------+
    //|unitPrice|isExpensive|
    //+---------+-----------+
    //|   569.77|       true|
    //|   607.49|       true|
    //+---------+-----------+
```

> if you're working with null data when creating Boolean expressions. If there is a null in your data, you'll need to treat things a bit differently. Here's how you can ensure that you perform a null-safe equivalence test:
>
> ```
> df.where(col("Description").eqNullSafe("hello")).show()
> ```
>
> Although not currently available (Spark 2.2), IS [NOT] DISTINCT FROM will be coming in Spark 2.3 to do the same thing in SQL.

Numbers

```
  def work_with_numbers(): Unit ={
    df.select(col("CustomerId"), col("Quantity"), col("UnitPrice")).show(5)
    //+----------+--------+---------+
    //|CustomerId|Quantity|UnitPrice|
    //+----------+--------+---------+
    //|   17850.0|       6|     2.55|
    //|   17850.0|       6|     3.39|
    //|   17850.0|       8|     2.75|
    //|   17850.0|       6|     3.39|
    //|   17850.0|       6|     3.39|
    //+----------+--------+---------+
    import org.apache.spark.sql.functions.{expr, pow}
    val fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
    df.select(expr("CustomerId"),
fabricatedQuantity.alias("realQuantity")).show(5)
    df.selectExpr(
      "CustomerId",
      "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity").show(5)
    //+----------+------------------+
```

```
    //|CustomerId|      realQuantity|
    //+----------+------------------+
    //|   17850.0|239.08999999999997|
    //|   17850.0|          418.7156|
    //|   17850.0|             489.0|
    //|   17850.0|          418.7156|
    //|   17850.0|          418.7156|
    //+----------+------------------+
    df.select(expr("CustomerId"),
round(fabricatedQuantity,4).alias("realQuantity")).show(5)
    //+----------+------------+
    //|CustomerId|realQuantity|
    //+----------+------------+
    //|   17850.0|      239.09|
    //|   17850.0|    418.7156|
    //|   17850.0|       489.0|
    //|   17850.0|    418.7156|
    //|   17850.0|    418.7156|
    //+----------+------------+

    import org.apache.spark.sql.functions.{round, bround}
    df.select(round(col("UnitPrice"), 1).alias("rounded"),
col("UnitPrice")).show(5)
    //+-------+---------+
    //|rounded|UnitPrice|
    //+-------+---------+
    //|    2.6|     2.55|
    //|    3.4|     3.39|
    //|    2.8|     2.75|
    //|    3.4|     3.39|
    //|    3.4|     3.39|
    //+-------+---------+

    import org.apache.spark.sql.functions.lit
    df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)
    //+-------------+--------------+
    //|round(2.5, 0)|bround(2.5, 0)|
    //+-------------+--------------+
    //|          3.0|           2.0|
    //|          3.0|           2.0|
    //+-------------+--------------+

    import org.apache.spark.sql.functions.{corr}
    println(df.stat.corr("Quantity", "UnitPrice"))  //-0.04112314436835551
    df.select(corr("Quantity", "UnitPrice")).show()


 df.select(col("StockCode"),col("Description"),col("Description")).describe().sh
ow()
    //+-------+-----------------+--------------------+--------------------+
    //|summary|        StockCode|         Description|         Description|
    //+-------+-----------------+--------------------+--------------------+
    //|  count|             3108|                3098|                3098|
    //|   mean|27834.304044117645|                null|                null|
    //| stddev|17407.897548583845|                null|                null|
    //|    min|            10002| 4 PURPLE FLOCK D...| 4 PURPLE FLOCK D...|
    //|    max|             POST|ZINC WILLIE WINKI...|ZINC WILLIE WINKI...|
    //+-------+-----------------+--------------------+--------------------+
```

```
    val quantileProbs = Array(0.5)
    val relError = 0.05
    df.stat.approxQuantile("UnitPrice", quantileProbs, relError) //2.51

    import org.apache.spark.sql.functions.monotonically_increasing_id
    df.select(monotonically_increasing_id().alias("id"), expr("*")).show(5)
    //+---+---------+---------+--------------------+--------+------------------
+---------+----------+--------------+
    //| id|InvoiceNo|StockCode|         Description|Quantity|
InvoiceDate|UnitPrice|CustomerID|       Country|
    //+---+---------+---------+--------------------+--------+------------------
+---------+----------+--------------+
    //|  0|   536365|   85123A|WHITE HANGING HEA...|       6|2010-12-01 08:26:00|
    2.55|   17850.0|United Kingdom|
    //|  1|   536365|    71053| WHITE METAL LANTERN|       6|2010-12-01 08:26:00|
    3.39|   17850.0|United Kingdom|
    //|  2|   536365|   84406B|CREAM CUPID HEART...|       8|2010-12-01 08:26:00|
    2.75|   17850.0|United Kingdom|
    //|  3|   536365|   84029G|KNITTED UNION FLA...|       6|2010-12-01 08:26:00|
    3.39|   17850.0|United Kingdom|
    //|  4|   536365|   84029E|RED WOOLLY HOTTIE...|       6|2010-12-01 08:26:00|
    3.39|   17850.0|United Kingdom|
    //+---+---------+---------+--------------------+--------+------------------
+---------+----------+--------------+
  }
```

There are a number of statistical functions available in the **StatFunctions Package** (accessible using stat).

String

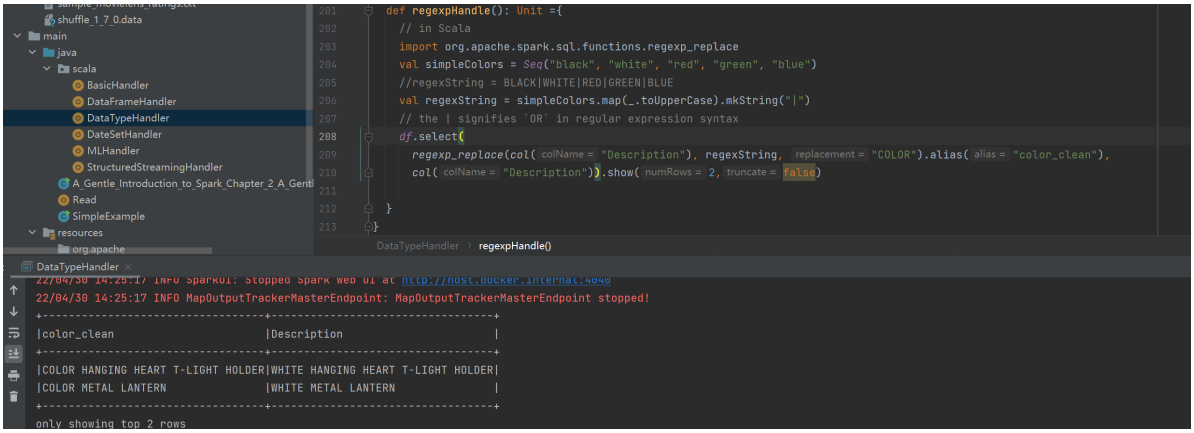
```
def work_with_String(): Unit ={
    import org.apache.spark.sql.functions.{initcap}
    df.select("Description").show(2, false)
    //+----------------------------------+
    //|Description                       |
    //+----------------------------------+
    //|WHITE HANGING HEART T-LIGHT HOLDER|
    //|WHITE METAL LANTERN               |
    //+----------------------------------+
    df.select(initcap(col("Description")),
      lower(col("Description")),
      upper(col("Description")),

    ).show(2, false)
    //+----------------------------------+----------------------------------+----
------------------------------+
    //|initcap(Description)              |lower(Description)
|upper(Description)               |
    //+----------------------------------+----------------------------------+----
------------------------------+
    //|White Hanging Heart T-light Holder|white hanging heart t-light
holder|WHITE HANGING HEART T-LIGHT HOLDER|
    //|White Metal Lantern               |white metal lantern
|WHITE METAL LANTERN               |
    //+----------------------------------+----------------------------------+----
------------------------------+
```

```
    import org.apache.spark.sql.functions.{lit, ltrim, rtrim, rpad, lpad, trim}
    df.select(
      ltrim(lit(" HELLO ")).as("ltrim"),
      rtrim(lit(" HELLO ")).as("rtrim"),
      trim(lit(" HELLO ")).as("trim"),
      lpad(lit("HELLO"), 6, " ").as("lp"),
      rpad(lit("HELLO"), 6, " ").as("rp")).show(2,false)
    //lp len = 3, rp len = 10
    // +------+------+-----+---+----------+
    //| ltrim| rtrim| trim| lp|        rp|
    //+------+------+-----+---+----------+
    //|HELLO | HELLO|HELLO|HEL|HELLO     |
    //|HELLO | HELLO|HELLO|HEL|HELLO     |
    //+------+------+-----+---+----------+
    //lp len = 6, rp len = 6
    // +------+------+-----+------+------+
    //|ltrim |rtrim |trim |lp    |rp    |
    //+------+------+-----+------+------+
    //|HELLO | HELLO|HELLO| HELLO|HELLO |
    //|HELLO | HELLO|HELLO| HELLO|HELLO |
    //+------+------+-----+------+------+
  }
```

Regex

There are two key functions in Spark that you'll need in order to perform regular expression tasks: **regexp_extract** and **regexp_replace**. These functions extract values and replace values, respectively.

```
import org.apache.spark.sql.functions.regexp_replace
val simpleColors = Seq("black", "white", "red", "green", "blue")
val regexString = simpleColors.map(_.toUpperCase).mkString("|")
// the | signifies `OR` in regular expression syntax
df.select(
regexp_replace(col("Description"), regexString, "COLOR").alias("color_clean"),
col("Description")).show(2)
```



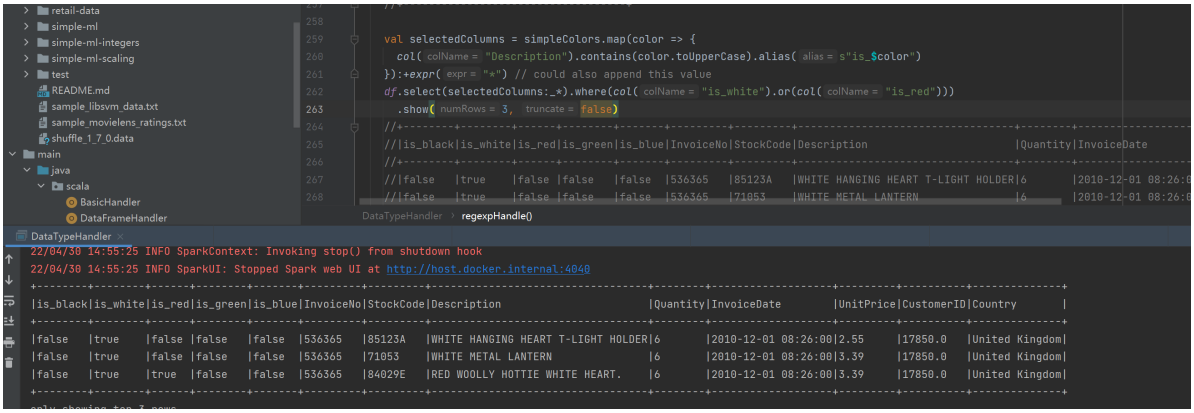
```
//replace given characters with other characters, translate
import org.apache.spark.sql.functions.translate
df.select(translate(col("Description"), "LEET", "1337"), col("Description"))
.show(2)
//+---------------------------------+--------------------+
//|translate(Description, LEET, 1337)|         Description|
//+---------------------------------+--------------------+
//|              WHI73 HANGING H3A...|WHITE HANGING HEA...|
//|               WHI73 M37A1 1AN73RN| WHITE METAL LANTERN|
//+---------------------------------+--------------------+
```

```
//extract values
import org.apache.spark.sql.functions.regexp_extract
regexString = simpleColors.map(_.toUpperCase).mkString("(", "|", ")")
// the | signifies OR in regular expression syntax
df.select(
regexp_extract(col("Description"), regexString, 1).alias("color_clean"),
col("Description")).show(2,false)
//+-----------+---------------------------------+
//|color_clean|Description                      |
//+-----------+---------------------------------+
//|WHITE      |WHITE HANGING HEART T-LIGHT HOLDER|
//|WHITE      |WHITE METAL LANTERN              |
//+-----------+---------------------------------+
```

```
//check values' exsistence, using 'contains'
val containsPURPLE = col("Description").contains("PURPLE")
val containsWhite = col("DESCRIPTION").contains("WHITE")
df.withColumn("hasSimpleColor", containsPURPLE.or(containsWhite))
.where("hasSimpleColor")
.select("Description").show(2, false)
//+---------------------------------+
//|Description                      |
//+---------------------------------+
//|WHITE HANGING HEART T-LIGHT HOLDER|
//|WHITE METAL LANTERN              |
//+---------------------------------+
df.withColumn("hasSimpleColor", containsPURPLE)
.where("hasSimpleColor")
.select("Description").show(2, false)
//+----------------------------------+
//|Description                       |
//+----------------------------------+
//|PURPLE DRAWERKNOB ACRYLIC EDWARDIAN|
//|FELTCRAFT HAIRBAND PINK AND PURPLE |
//+----------------------------------+
```

! contains(str) : str is pattern, it's case sensitivity

This is trivial with just two values, but it becomes more complicated when there are values. Let's work through this in a more rigorous way and take advantage of Spark's ability to **accept a dynamic number of arguments**. When we convert a list of values into a set of arguments and pass them into a function, we use a language feature called **varargs**. Using this feature, we can **effectively unravel an array of arbitrary length and pass it as arguments to a function**. This, coupled with select makes it possible for us to create arbitrary numbers of columns dynamically:



This simple feature can often help you programmatically **generate columns or Boolean filters** in a way that is simple to understand and extend. We could extend this to **calculating the smallest common denominator for a given input value, or whether a number is a prime.**

类似构造位图，做与或操作

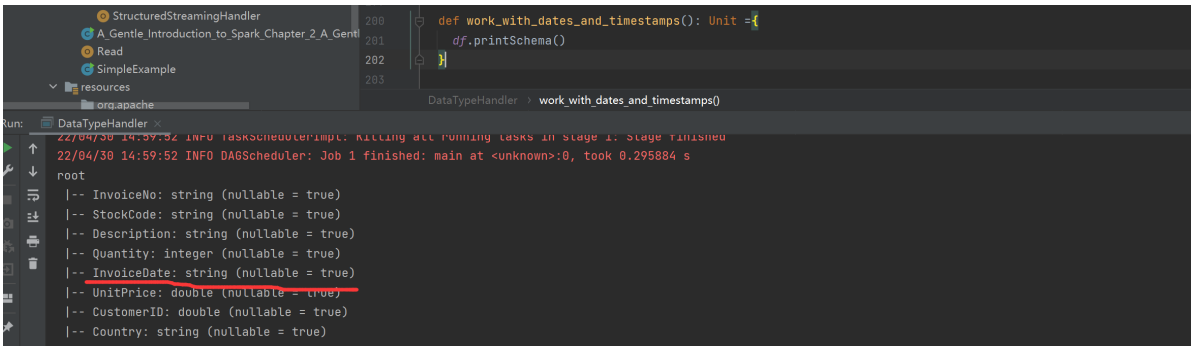
Dates and Timestamps

Dates and times are a constant challenge in programming languages and databases.

- **Dates focus exclusively on calendar dates**
- **Timestamps, which include both date and time information.**

Spark, as we saw with our current dataset, will make a best effort to correctly identify column types, **including dates and timestamps when we enable inferSchema.** We can see that this worked quite well with our current dataset because it was able to identify and read our date format without us having to provide some specification for it.

As we hinted earlier, working with dates and timestamps closely relates to working with strings because **we often store our timestamps or dates as strings and convert them into date types at runtime**. This is less common when working with databases and structured data but much more common when we are working with text and CSV files. We will experiment with that shortly.



Although Spark will do read dates or times on a best-effort basis. However, sometimes there will be no getting around working with strangely formatted dates and times. The key to understanding the transformations that you are going to need to apply is to ensure that you know exactly what type and format you have at each given step of the way. Another common "gotcha" is that Spark's TimestampType class **supports only second-level precision**, which means that if you're going to be working with milliseconds or microseconds, **you'll need to work around this problem by potentially operating on them as longs.** Any more precision when coercing to a TimestampType will be **removed.**

```
import org.apache.spark.sql.functions.{current_date, current_timestamp}
val dateDF = spark.range(10)
```

```
.withColumn("today", current_date())
.withColumn("now", current_timestamp())
dateDF.show(false)
dateDF.printSchema()
//+---+----------+-----------------------+
//|id |today     |now                    |
//+---+----------+-----------------------+
//|0  |2022-04-30|2022-04-30 15:26:13.822|
//|1  |2022-04-30|2022-04-30 15:26:13.822|
//|2  |2022-04-30|2022-04-30 15:26:13.822|
//|3  |2022-04-30|2022-04-30 15:26:13.822|
//|4  |2022-04-30|2022-04-30 15:26:13.822|
//|5  |2022-04-30|2022-04-30 15:26:13.822|
//|6  |2022-04-30|2022-04-30 15:26:13.822|
//|7  |2022-04-30|2022-04-30 15:26:13.822|
//|8  |2022-04-30|2022-04-30 15:26:13.822|
//|9  |2022-04-30|2022-04-30 15:26:13.822|
//+---+----------+-----------------------+
//root
// |-- id: long (nullable = false)
// |-- today: date (nullable = false)
// |-- now: timestamp (nullable = false)
dateDF.createOrReplaceTempView("dateTable")
import org.apache.spark.sql.functions.{date_add, date_sub}
dateDF.select(date_sub(col("today"), 5), date_add(col("today"), 5)).show(1)
//+------------------+------------------+
//|date_sub(today, 5)|date_add(today, 5)|
//+------------------+------------------+
//|        2022-04-25|        2022-05-05|
//+------------------+------------------+

//datediff: function that will return the number of days in between two dates
//,months_between: function that gives you the number of months between two
dates
import org.apache.spark.sql.functions.{datediff, months_between, to_date}
dateDF.withColumn("week_ago", date_sub(col("today"), 7))
.select(datediff(col("week_ago"), col("today"))).show(1)
//+-------------------------+
//|datediff(week_ago, today)|
//+-------------------------+
//|                       -7|
//+-------------------------+
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"),false)).show(1)
//+--------------------------------++---------------------------------+
//|months_between(start, end, true)||months_between(start, end, false)|
//+--------------------------------++---------------------------------+
//|                     -16.67741935||              -16.677419354838708|
//+--------------------------------++---------------------------------+
```

```
//to_date and to_timestamp
import org.apache.spark.sql.functions.to_date
val dateFormat = "yyyy-dd-MM"
val 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")
cleanDateDF.show()
//+----------+----------+
//|      date|     date2|
//+----------+----------+
//|2017-11-12|2017-12-20|
//+----------+----------+
cleanDateDF.printSchema()
//root
// |-- date: date (nullable = true)
// |-- date2: date (nullable = true)
import org.apache.spark.sql.functions.to_timestamp
cleanDateDF.select(to_timestamp(col("date"), dateFormat)).show()
//+----------------------------------+
//|to_timestamp(`date`, 'yyyy-dd-MM')|
//+----------------------------------+
//|               2017-11-12 00:00:00|
//+----------------------------------+
cleanDateDF.filter(col("date2") > lit("2017-12-12")).show()
cleanDateDF.filter(col("date2") > to_date(lit("2017-12-12"))).show() //recommend
cleanDateDF.filter(col("date2") > "2017-12-12").show()
cleanDateDF.filter(col("date2") > "'2017-12-12'").show() //failed
```

The following doesn't work!

> One minor point is that we can also set this as a string, which Spark parses to a literal:

```
cleanDateDF.filter(col("date2") > "'2017-12-12'").show()
```

Implicit type casting is an easy way to shoot yourself in the foot, especially when dealing with null values or dates in different timezones or formats. We recommend that you parse them explicitly instead of relying on implicit conversions.

NULL

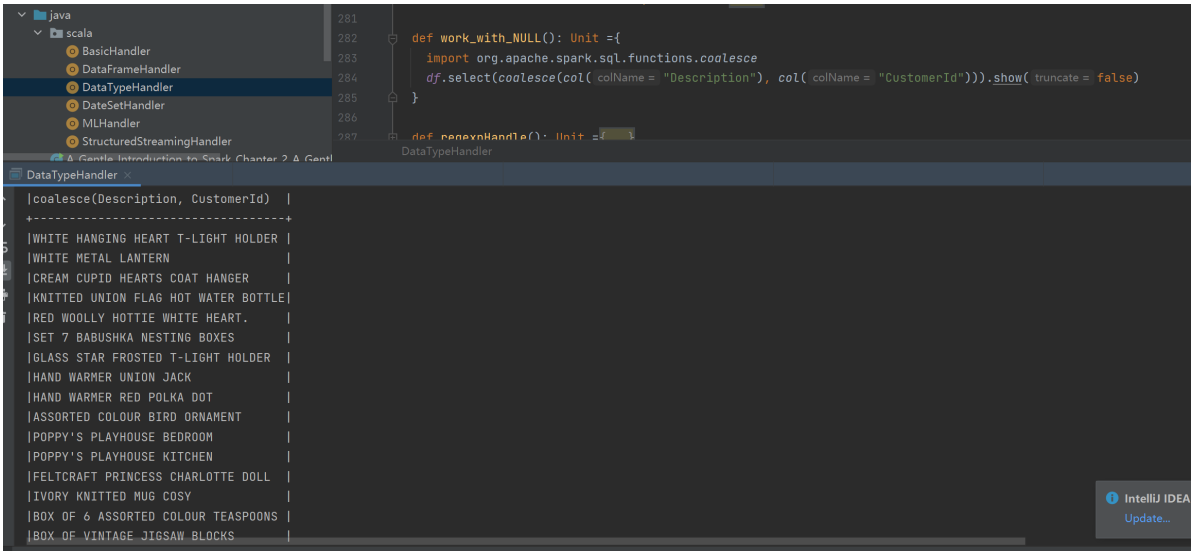
As a best practice, you **should always use nulls to represent missing or empty data in your DataFrames**. **Spark can optimize working with null values** more than it can if you use empty strings or other values. The primary way of interacting with null values, at DataFrame scale, is to use the **.na subpackage** on a DataFrame. There are also several functions for performing operations and explicitly specifying how Spark should handle null values.

When you define a schema in which all columns are declared to not have null values, **Spark will not enforce that** and will happily let null values into that column.

If you have null values in columns that should not have null values, you can get an incorrect result or see strange exceptions that can be difficult to debug.

Coalesce

Spark includes a function to allow you to **select the first non-null value from a set of columns by using the coalesce function.** In this case, there are no null values, so it simply returns the first column:



ifnull, nullIf, nvl, and nvl2

There are several other SQL functions that you can use to achieve similar things. **ifnull** allows you to select the second value if the first is null, and defaults to the first. Alternatively, you could use **nullif**, which returns null if the two values are equal or else returns the second if they are not. **nvl** returns the second value if the first is null, but defaults to the first. Finally, **nvl2** returns the second value if the first is not null; otherwise, it will return the last specified value (else_value in the following example):

```
SELECT
ifnull(null, 'return_value'),
nullif('value', 'value'),
nvl(null, 'return_value'),
nvl2('not_null', 'return_value', "else_value")
FROM dfTable LIMIT 1
```

```
+------------+----+------------+------------+
|           a|   b|           c|           d|
+------------+----+------------+------------+
|return_value|null|return_value|return_value|
+------------+----+------------+------------+
```

Drop Fill Replace

```
def work_with_NULL(): Unit ={
    import org.apache.spark.sql.functions.coalesce
    df.show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //| Hello|     null|    1|
    //|Second|         |    2|
    //| Third|     null|    3|
    //| Forth|     null| null|
    //|Normal|normalCol|    5|
    //+------+---------+-----+
```

```
    df.select(coalesce(col("some"), col("col"))).show(false)
    //+------------------+
    //|coalesce(some, col)|
    //+------------------+
    //|Hello             |
    //|Second            |
    //|Third             |
    //|Forth             |
    //|Normal            |
    //+------------------+
    //removes rows that contain nulls. The default is to drop any row in which
any value is null
    df.na.drop().show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //|Second|         |    2|
    //|Normal|normalCol|    5|
    //+------+---------+-----+
    df.na.drop("any").show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //|Second|         |    2|
    //|Normal|normalCol|    5|
    //+------+---------+-----+

    // Using “all” drops the row only if all values are null or NaN for that row
    df.na.drop("all").show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //| Hello|     null|    1|
    //|Second|         |    2|
    //| Third|     null|    3|
    //| Forth|     null| null|
    //|Normal|normalCol|    5|
    //+------+---------+-----+

    //certain sets of columns by passing in an array of columns
    df.na.drop("all", Seq("col", "names")).show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //| Hello|     null|    1|
    //|Second|         |    2|
    //| Third|     null|    3|
    //|Normal|normalCol|    5|
    //+------+---------+-----+

    //fill all null values in column
    df.na.fill("All Null values become this string").show(false)
    //+------+-----------------------------------+-----+
    //|some  |col                                |names|
    //+------+-----------------------------------+-----+
    //|Hello |All Null values become this string|1    |
    //|Second|                                   |2    |
    //|Third |All Null values become this string|3    |
```

```
    //|Forth |All Null values become this string|null |
    //|Normal|normalCol                          |5    |
    //+------+-----------------------------------+-----+
    df.na.fill(5, Seq("col", "names")).show()
    df.na.fill("fill", Seq("col", "names")).show()
    //+------+---------+-----++------+---------+-----+
    //|  some|      col|names||  some|      col|names|
    //+------+---------+-----++------+---------+-----+
    //| Hello|     null|    1|| Hello|     fill|    1|
    //|Second|         |    2||Second|         |    2|
    //| Third|     null|    3|| Third|     fill|    3|
    //| Forth|     null|    5|| Forth|     fill| null|
    //|Normal|normalCol|    5||Normal|normalCol|    5|
    //+------+---------+-----++------+---------+-----+

    val fillColValues = Map("names" -> 5, "col" -> "No Value") //recommend
    df.na.fill(fillColValues).show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //| Hello| No Value|    1|
    //|Second|         |    2|
    //| Third| No Value|    3|
    //| Forth| No Value|    5|
    //|Normal|normalCol|    5|
    //+------+---------+-----+

    //Probably the most common use case is to replace all values in a certain
column according to their current value.
    df.na.replace("col", Map("" -> "UNKNOWN")).show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //| Hello|     null|    1|
    //|Second|  UNKNOWN|    2|
    //| Third|     null|    3|
    //| Forth|     null| null|
    //|Normal|normalCol|    5|
    //+------+---------+-----+
    /**
    An advanced tip is to use asc_nulls_first, desc_nulls_first, asc_nulls_last,
or
    desc_nulls_last to specify where you would like your null values to appear in
an ordered
    DataFrame.
    */
    df.orderBy(col("names").desc).show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //|Normal|normalCol|    5|
    //| Third|     null|    3|
    //|Second|         |    2|
    //| Hello|     null|    1|
    //| Forth|     null| null|
    //+------+---------+-----+
    df.orderBy(col("names").desc_nulls_first).show()
    //+------+---------+-----+
```

```
    //|  some|      col|names|
    //+------+---------+-----+
    //| Forth|     null| null|
    //|Normal|normalCol|    5|
    //| Third|     null|    3|
    //|Second|         |    2|
    //| Hello|     null|    1|
    //+------+---------+-----+
    df.orderBy(col("names").asc_nulls_last).show()
    //+------+---------+-----+
    //|  some|      col|names|
    //+------+---------+-----+
    //| Hello|     null|    1|
    //|Second|         |    2|
    //| Third|     null|    3|
    //|Normal|normalCol|    5|
    //| Forth|     null| null|
    //+------+---------+-----+
}
```

Complex Types

Structs

You can think of structs as DataFrames within DataFrames.

```
def structTypeHandle(): Unit ={
  import org.apache.spark.sql.functions.struct
  //val complexDF = df.select(struct("Description",
"InvoiceNo").alias("complex"))
  //val complexDF = df.selectExpr("struct(Description, InvoiceNo) as complex")
  val complexDF = df.selectExpr("(Description, InvoiceNo) as complex")
  complexDF.createOrReplaceTempView("complexDF")
  complexDF.show(2,false)
  //+-----------------------------------------+
  //|complex                                  |
  //+-----------------------------------------+
  //|[WHITE HANGING HEART T-LIGHT HOLDER, 536365]|
  //|[WHITE METAL LANTERN, 536365]              |
  //+-----------------------------------------+
  complexDF.select("complex.Description").show(2,false)
  complexDF.select(col("complex").getField("Description")).show(2,false)
  //+---------------------------------+
  //|Description                      |
  //+---------------------------------+
  //|WHITE HANGING HEART T-LIGHT HOLDER|
  //|WHITE METAL LANTERN               |
  //+---------------------------------+
  complexDF.select("complex.*").show(2,false)
  //+---------------------------------+---------+
  //|Description                      |InvoiceNo|
  //+---------------------------------+---------+
  //|WHITE HANGING HEART T-LIGHT HOLDER|536365   |
  //|WHITE METAL LANTERN               |536365   |
  //+---------------------------------+---------+
}
```

Arrays

How to get an array

The first task is to turn our Description column into a complex type, an array.

The **explode function** takes a column that consists of arrays and creates one row (with the rest of the values duplicated) per value in the array.

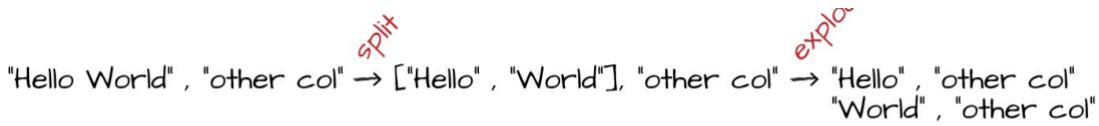


Figure 6-1. Exploding a column of text

```
def arrayTypeHandle(): Unit ={
  import org.apache.spark.sql.functions.split
  //using the split function and specify the delimiter
  df.select(split(col("Description"), " ")).show(2,false)
  //+-----------------------------------------+
  //|split(Description,  , -1)                |
  //+-----------------------------------------+
  //|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|
  //|[WHITE, METAL, LANTERN]                  |
  //+-----------------------------------------+
  /**
   特别地， 如果切割列为null, 则切割后的array也为null
   如： df.select(split(col("col"), "
").alias("array_col")).selectExpr("*").show()
          +-----------+
          |  array_col|
          +-----------+
          |       null|
          |         []|
          |       null|
          |       null|
          |[normalCol]|
          +-----------+
   */
  //query the values of the array using Python-like syntax
  df.select(split(col("Description"), " ").alias("array_col"))
    .selectExpr("array_col[4]").show(2)
  //+------------+
  //|array_col[4]|
  //+------------+
  //|      HOLDER|
  //|        null|
  //+------------+
  //querying its size
  import org.apache.spark.sql.functions.size
  df.select(size(split(col("Description"), " "))).show(2)
  //+------------------------------+
  //|size(split(Description,  , -1))|
  //+------------------------------+
  //|                             5|
  //|                             3|
  //+------------------------------+
  //see whether this array contains a value
```

```
  import org.apache.spark.sql.functions.array_contains
  df.select(array_contains(split(col("Description"), " "), "HEART")).show(2)
  //需要完全匹配， 例如指定value = “H”， 则全为false
  //+---------------------------------------------+
  //|array_contains(split(Description,  , -1), HEART)|
  //+---------------------------------------------+
  //|                                         true|
  //|                                        false|
  //+---------------------------------------------+
  import org.apache.spark.sql.functions.{split, explode}
  df.withColumn("splitted", split(col("Description"), " "))
    .withColumn("exploded", explode(col("splitted")))
    .select("Description", "InvoiceNo", "exploded").show(2)
  //+--------------------+---------+--------+
  //|         Description|InvoiceNo|exploded|
  //+--------------------+---------+--------+
  //|WHITE HANGING HEA...|   536365|   WHITE|
  //|WHITE HANGING HEA...|   536365| HANGING|
  //+--------------------+---------+--------+
}
```

Maps

Maps are created by using the map function and **key-value pairs of columns**. You then can select them just like you might select from an array

```
def mapTypeHandle(): Unit ={
  import org.apache.spark.sql.functions.map
  df.select(map(col("Description"), col("InvoiceNo"))
    .alias("complex_map")).show(2,false)
  //+--------------------+
  //| complex_map|
  //+--------------------+
  //|Map(WHITE HANGING...|
  //|Map(WHITE METAL L...|
  //+--------------------+
  df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))
    .selectExpr("complex_map['WHITE METAL LANTERN']").show(2)
  //+--------------------------------+
  //|complex_map[WHITE METAL LANTERN]|
  //+--------------------------------+
  //| null|
  //| 536365|
  //+--------------------------------+
  df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))
    .selectExpr("explode(complex_map)").show(2)
  //+--------------------+------+
  //|                 key| value|
  //+--------------------+------+
  //|WHITE HANGING HEA...|536365|
  //| WHITE METAL LANTERN|536365|
  //+--------------------+------+
}
```

JSON

!书中有误

> Here’s the equivalent in SQL:
>
> ```
> jsonDF.selectExpr(
>   "json_tuple(jsonString, '$.myJSONKey.myJSONValue[1]') as column").show(2)
> ```
>
> This results in the following table:
>
> ```
> +------+--------------------+
> |column|                  c0|
> +------+--------------------+
> |     2|{"myJSONValue":[1...|
> +------+--------------------+
> ```

正确写法：


```
def work_with_JSON(): Unit ={
  val jsonDF = spark.range(1)
    .selectExpr("""'{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as
jsonString""")
  jsonDF.show(false)
  //+-----------------------------------------+
  //|jsonString                               |
  //+-----------------------------------------+
  //|{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}|
  //+-----------------------------------------+

  import org.apache.spark.sql.functions.{get_json_object, json_tuple}
  jsonDF.select(
    get_json_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]") as
"column",
    json_tuple(col("jsonString"), "myJSONKey")).show(2,false)
  //+------+----------------------+
  //|column|c0                    |
  //+------+----------------------+
  //|2     |{"myJSONValue":[1,2,3]}|
  //+------+----------------------+
  jsonDF.selectExpr(
    "get_json_object(jsonString, '$.myJSONKey.myJSONValue[1]') as column"
    , "json_tuple(jsonString, 'myJSONKey')").show(2,false)
  //+------+----------------------+
  //|column|c0                    |
  //+------+----------------------+
  //|2     |{"myJSONValue":[1,2,3]}|
  //+------+----------------------+
  //turn a StructType into a JSON string by using the to_json function
```

```
  import org.apache.spark.sql.functions.to_json
  df.selectExpr("(InvoiceNo, Description) as myStruct")
    .select(to_json(col("myStruct"))).show(2,false)
  //+-----------------------------------------------------------------------+
  //|to_json(myStruct)                                                      |
  //+-----------------------------------------------------------------------+
  //|{"InvoiceNo":"536365","Description":"WHITE HANGING HEART T-LIGHT HOLDER"}|
  //|{"InvoiceNo":"536365","Description":"WHITE METAL LANTERN"}               |
  //+-----------------------------------------------------------------------+
  /**
   * You can use the from_json function to parse this (or other JSON data) back
in.
   * This naturally requires you to specify a schema, and optionally you can
specify a map of options
   */
  import org.apache.spark.sql.functions.from_json
  import org.apache.spark.sql.types._
  val parseSchema = new StructType(Array(
    new StructField("InvoiceNo",StringType,true),
    new StructField("Description",StringType,true)))
  df.selectExpr("(InvoiceNo, Description) as myStruct")
    .select(to_json(col("myStruct")).alias("newJSON"))
    .select(from_json(col("newJSON"), parseSchema),
col("newJSON")).show(2,false)
  //+----------------------------------------+------------------------------
----------------------------------------+
  //|from_json(newJSON)                      |newJSON
                                         |
  //+----------------------------------------+------------------------------
----------------------------------------+
  //|[536365, WHITE HANGING HEART T-LIGHT HOLDER]|
{"InvoiceNo":"536365","Description":"WHITE HANGING HEART T-LIGHT HOLDER"}|
  //|[536365, WHITE METAL LANTERN]               |
{"InvoiceNo":"536365","Description":"WHITE METAL LANTERN"}               |
  //+----------------------------------------+------------------------------
----------------------------------------+
}
```

UDF

User-defined functions (UDFs) make it possible for you to write your own custom transformations using Python or Scala and even use external libraries. UDFs can take and return one or more columns as input.

If the function is written in **Scala or Java**, you can **use it within the Java Virtual Machine (JVM)**. This means that there will be little performance penalty aside from the fact that you can't take advantage of code generation capabilities that Spark has for built-in functions.

If the function is written in Python, something quite different happens. Spark starts a Python **process on the worker**, serializes **all of the data to a format** that Python can understand (remember, it was in the JVM earlier), **executes the function row by row** on that data in the Python process, and then finally **returns the results of the row operations to the JVM and Spark.**

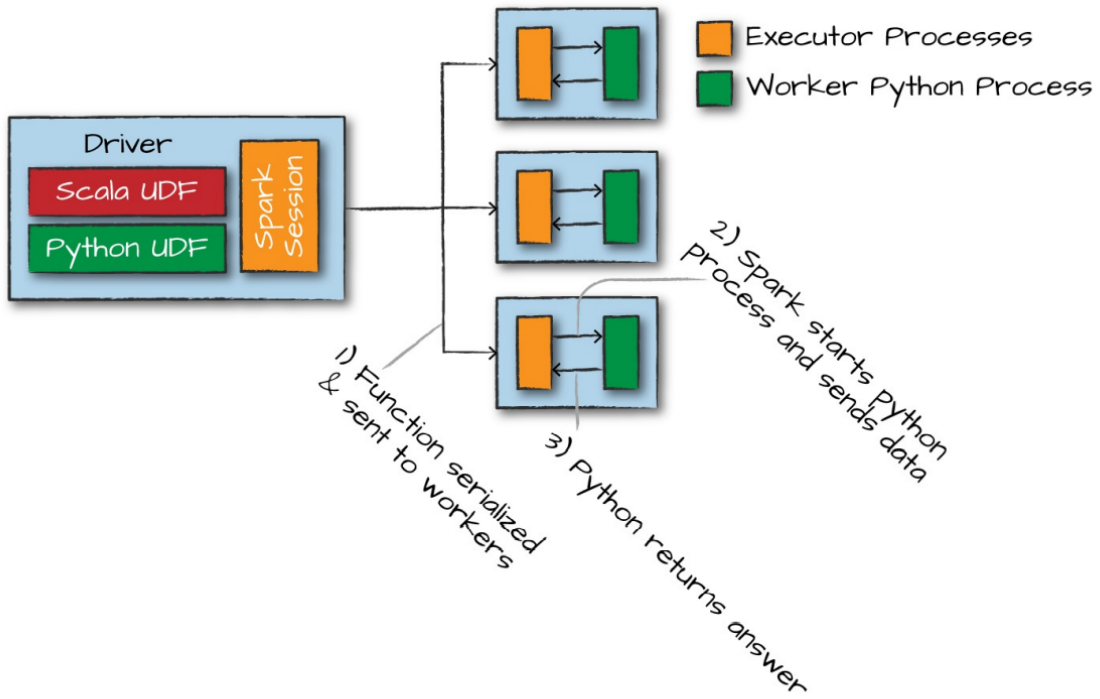


Figure 6-2. Figure caption

Starting this Python process is expensive, but the real cost is in **serializing the data** to Python. This is costly for two reasons: it is an **expensive computation**, but also, after the data enters Python, **Spark cannot manage the memory of the worker**.

We recommend that you write your UDFs in Scala or Java—the small amount of time it should take you to write the function in Scala will always yield significant speed ups, and on top of that, you can still use the function from Python!

It is important to note that **specifying the return type is not necessary, but it is a best practice.**

```
  def work_with_UDF(): Unit ={
    // in Scala
    val udfExampleDF = spark.range(5).toDF("num")
    println(power3(2.0))
    import org.apache.spark.sql.functions.udf
    //val power3udf = udf(power3(_:Double):Double)
    val power3udf = udf(SimpleExample.power(_:Double):Double)
    //val power3udf = udf(SimpleExample.power(_))  //not recommend
    udfExampleDF.select(power3udf(col("num"))).show()
    //+--------+
    //|UDF(num)|
    //+--------+
    //|     0.0|
    //|     1.0|
    //|     8.0|
    //|    27.0|
    //|    64.0|
    //+--------+
    /**
        At this juncture, we can use this only as a DataFrame function. That is
to say, we can't use it
```

```
        within a string expression, only on an expression. However, we can also
register this UDF as a
        Spark SQL function. This is valuable because it makes it simple to use
this function within SQL
        as well as across languages.
     */
    //register the function, 对df api不可见
    spark.udf.register("power3", power3(_:Double):Double)
    udfExampleDF.selectExpr("power3(num)").show(2)
    udfExampleDF.createOrReplaceTempView("udf_table")

    //所谓持久化，外部引入方法
    val sqlS = "CREATE OR REPLACE FUNCTION pow3 AS 'MyUDF' USING JAR
'src/data/udf-1.0-SNAPSHOT.jar'"
    spark.sql(sqlS)
    val exeS = "SELECT pow3(num) AS function_return_value FROM udf_table"
    spark.sql(exeS).show()
  }
```

As a last note, you can also use UDF/UDAF creation via a Hive syntax. To allow for this, first you must enable Hive support when they create their SparkSession (via `SparkSession.builder().enableHiveSupport()`). Then you can register UDFs in SQL. This is only supported with precompiled Scala and Java packages, so you'll need to specify them as a dependency:

```
-- in SQL
CREATE TEMPORARY FUNCTION myFunc AS 'com.organization.hive.udf.FunctionName'
```

Additionally, you can register this as a permanent function in the Hive Metastore by removing `TEMPORARY`.

吐槽

```
import org.apache.hadoop.hive.ql.exec.UDF;
import java.io.Serializable;

public class MyUDF extends UDF implements Serializable {
    public double evaluate(double val){
        return val * val * val;
    }
}
```

！外部jar包内的方法名一定要是evaluate，不然无法被识别

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.

The simplest grouping is to just summarize a complete DataFrame by performing an aggregation in a select statement.

A "group by" allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns.

A “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.

A “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.

A “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.

A “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.

Each grouping returns a **RelationalGroupedDataset** on which we specify our aggregations.

An important thing to consider is how exact you need an answer to be. When performing calculations over big data, **it can be quite expensive to get an exact answer to a question**, and it’s often much cheaper to simply request an approximate to a reasonable degree of accuracy.

count

count will perform as a transformation instead of an action

we can do one of two things: specify a **specific column** to count, or **all the columns** by using count(*) or count(1)

```
def countHandle(): Unit ={
  //count is actually an action as opposed to a transformation
  print(df.count()) //541909
  import org.apache.spark.sql.functions.count
  df.select(count("StockCode")).show()
  df.selectExpr("count(1)").show()
  //+----------------++--------+
  //|count(StockCode)||count(1)|
  //+----------------++--------+
  //|          541909||  541909|
  //+----------------++--------+
  df.select(countDistinct("StockCode")).show() //4070 took 2.771206 s
  //Aggregate function: returns the approximate number of distinct items in a
group.
  //Params:  rsd - maximum estimation error allowed (default = 0.05)
  df.select(approx_count_distinct("StockCode", 0.1)).show() // 3364 took
0.214926 s

  df = loadNullData()
  df.selectExpr("count(1)").show()
  df.select(count("col")).show()
  //+--------++----------+
  //|count(1)||count(col)|
  //+--------++----------+
  //|       5||         2|
  //+--------++----------+

}
```

> NOTE

> when performing a count(*), Spark will count null values (including rows containing all nulls). However, when counting an individual column, Spark will not count the null values.

first and last

```
def first_and_last(): Unit ={
  import org.apache.spark.sql.functions.{first, last}
  df.select(first("StockCode"), last("StockCode")).show()
  //+-----------------------+----------------------+
  //|first(StockCode, false)|last(StockCode, false)|
  //+-----------------------+----------------------+
  //|                 85123A|                 22138|
  //+-----------------------+----------------------+
}
```

min and max

To **extract the minimum and maximum values** from a DataFrame, use the min and max functions:

```
def min_and_max(): Unit ={
  import org.apache.spark.sql.functions.{min, max}
  df.select(min("Quantity"), max("Quantity")).show()
  //+-------------+-------------+
  //|min(Quantity)|max(Quantity)|
  //+-------------+-------------+
  //|       -80995|        80995|
  //+-------------+-------------+
}
```

sum

```
def sumHandle(): Unit ={
  import org.apache.spark.sql.functions.sum
  df.select(sum("Quantity")).show() // 5176450
  //sum a distinct set of values
  df.select(sumDistinct("Quantity")).show() // 29310
}
```

avg

Although you can calculate average by **dividing sum by count**, Spark provides an easier way to get that value via the **avg or mean functions**

```
override lazy val evaluateExpression = child.dataType match {
  case _: DecimalType =>
    DecimalPrecision.decimalAndDecimal(sum / count.cast(DecimalType.LongDecimal)).cast(resultType)
  case _ =>
    sum.cast(resultType) / count.cast(resultType)
}
```

```
def avgHandle(): Unit ={
  import org.apache.spark.sql.functions.{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()
  //+-----------------------------------+----------------+----------------+
  //|(total_purchases / total_transactions)|   avg_purchases|  mean_purchases|
  //+-----------------------------------+----------------+----------------+
  //|                    9.55224954743324|9.55224954743324|9.55224954743324|
  //+-----------------------------------+----------------+----------------+
}
```

Variance and Standard Deviation

Calculating the mean naturally brings up questions about the variance and standard deviation. **These are both measures of the spread of the data around the mean.** (Standard Deviation = sqrt (Variance))

You can calculate these in Spark by using their respective functions. However, something to note is that Spark has both the formula for the **sample standard deviation** as well as the formula for the **population standard deviation**

```
def variance_and_standard_deviation(): Unit ={
  import org.apache.spark.sql.functions.{var_pop, stddev_pop}
  import org.apache.spark.sql.functions.{var_samp, stddev_samp}
  //population and sample
  df.select(var_pop("Quantity"), var_samp("Quantity"),
    stddev_pop("Quantity"), stddev_samp("Quantity")).show()
  //+-----------------+------------------+--------------------+-----------------
----+
  //|var_pop(Quantity)|var_samp(Quantity)|stddev_pop(Quantity)|stddev_samp(Quant
ity)|
  //+-----------------+------------------+--------------------+-----------------
----+
  //|47559.30364660879| 47559.39140929848|  218.08095663447733|
218.08115785023355|
  //+-----------------+------------------+--------------------+-----------------
----+
}
```

skewness and kurtosis

Skewness and kurtosis are both measurements of extreme points in your data.

Skewness measures the **asymmetry of the values in your data around the mean**, whereas kurtosis is a measure of the tail of data.

```
def skewness_and_kurtosis(): Unit ={
  import org.apache.spark.sql.functions.{skewness, kurtosis}
  df.select(skewness("Quantity"), kurtosis("Quantity")).show()
  //+------------------+------------------+
  //|skewness(Quantity)|kurtosis(Quantity)|
  //+------------------+------------------+
  //|-0.264075576105298|119768.05495534067|
  //+------------------+------------------+
}
```

Covariance and Correlation

compare the interactions of the values in two difference columns

```
def covariance_and_correlation(): Unit ={
  import org.apache.spark.sql.functions.{corr, covar_pop, covar_samp}
  df.select(corr("InvoiceNo", "Quantity"),
    covar_samp("InvoiceNo", "Quantity"),
    covar_pop("InvoiceNo", "Quantity")).show()
  //+-------------------------+-----------------------------+-----------------
-------------+
  //|corr(InvoiceNo, Quantity)|covar_samp(InvoiceNo,
Quantity)|covar_pop(InvoiceNo, Quantity)|
  //+-------------------------+-----------------------------+-----------------
-------------+
  //|     4.912186085636837E-4|             1052.7280543912716|
1052.7260778751674|
  //+-------------------------+-----------------------------+-----------------
-------------+
}
```

Aggregating to Complex Types

```
def complexTypeHandle(): Unit ={
  // collect a list of values present in a given column or only the unique
values by collecting to a set.
  import org.apache.spark.sql.functions.{collect_set, collect_list}
  df.agg(collect_set("Country"), collect_list("Country")).show()
  //+--------------------+---------------------+
  //|collect_set(Country)|collect_list(Country)|
  //+--------------------+---------------------+
  //|[Portugal, Italy,...| [United Kingdom, ...|
  //+--------------------+---------------------+
}
```

Group

```
def groupHandle(): Unit ={
  df.groupBy("InvoiceNo", "CustomerId").count().show(3,false)
  //+---------+----------+-----+
  //|InvoiceNo|CustomerId|count|
  //+---------+----------+-----+
  //|536846   |14573     |76   |
  //|537026   |12395     |12   |
```

```
  //|537883   |14437     |5    |
  //+---------+----------+-----+
  df.groupBy("InvoiceNo").agg(
    count("Quantity").alias("quan"),
    expr("count(Quantity)")).show(3,false)
  //+---------+----+---------------+
  //|InvoiceNo|quan|count(Quantity)|
  //+---------+----+---------------+
  //|   536596|   6|              6|
  //|   536938|  14|             14|
  //|   537252|   1|              1|
  //+---------+----+---------------+
  df.groupBy("InvoiceNo").agg("Quantity"->"avg", "Quantity"-
>"stddev_pop").show(2)
  //+---------+------------------+--------------------+
  //|InvoiceNo|     avg(Quantity)|stddev_pop(Quantity)|
  //+---------+------------------+--------------------+
  //|   536596|               1.5|  1.1180339887498947|
  //|   536938|33.142857142857146|  20.698023172885524|
  //+---------+------------------+--------------------+
}
```

Sometimes we want something a bit more complete—**an aggregation across multiple groups**. We achieve this by using **grouping sets**.

Let's work through an example to gain a better understanding. Here, we would like to **get the total quantity of all stock codes and customers.**

> **WARNING**
>
> Grouping sets depend on null values for aggregation levels. If you do not filter-out null values, you will get incorrect results. This applies to cubes, rollups, and grouping sets.

groupset rollup cube

A **rollup** is a multidimensional aggregation that performs a variety of group-by style calculations for us.

```
org.apache.spark.sql.Dataset
@varargs
def rollup(col1: String, cols: String*): RelationalGroupedDataset
```

Create a multi-dimensional rollup for the current Dataset using the specified columns, so we can run aggregation on them. See `RelationalGroupedDataset` for all the available aggregate functions. This is a variant of rollup that can only group by existing columns using column names (i.e. cannot construct expressions).

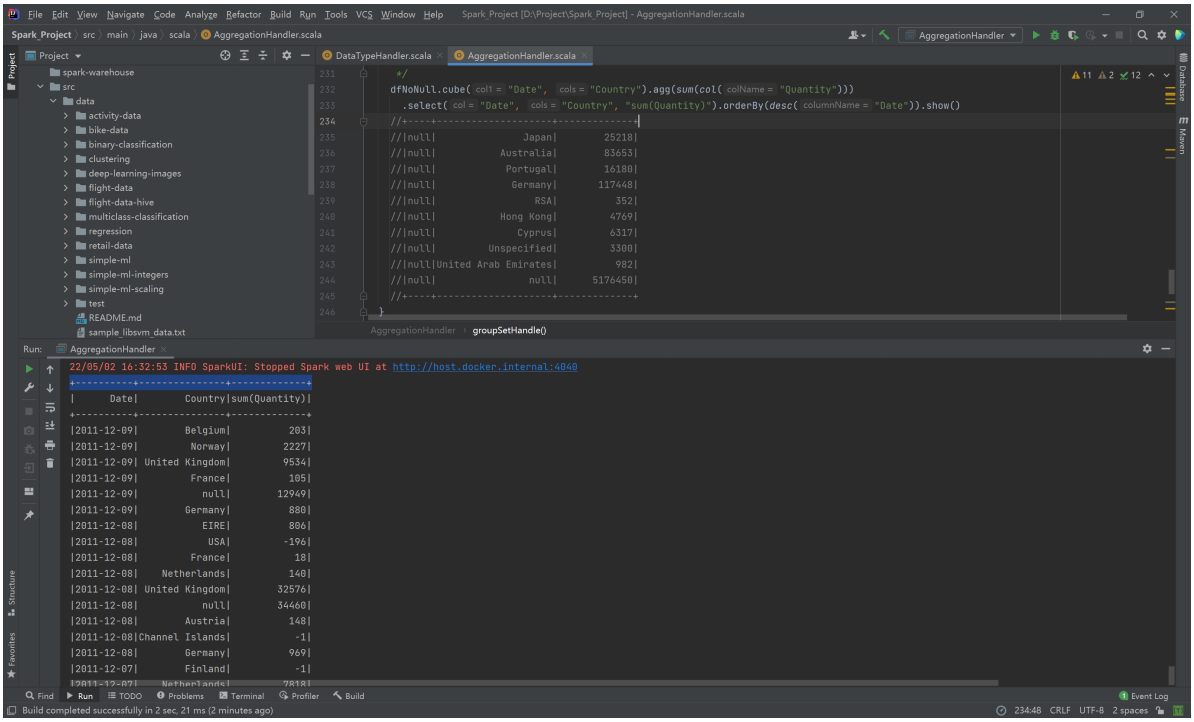
```
// Compute the average for all numeric columns rollupped by department and group.
ds.rollup("department", "group").avg()

// Compute the max age and average salary, rollupped by department and gender.
ds.rollup($"department", $"gender").agg(Map(
  "salary" -> "avg",
  "age" -> "max"
))
```

Since: 2.0.0

Maven: org.apache.spark:spark-sql_2.12:3.0.0

A **cube** takes the rollup to a level deeper. Rather than treating elements hierarchically, a cube does the same thing across all dimensions. This means that it won't just go by date over the entire time period, but also the country. To pose this as a question again, 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 date




```
def groupSetHandle(): Unit ={
  //Get the total quantity of all stock codes and customers

  val dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"),
    "MM/d/yyyy H:mm"))
  dfWithDate.show(2)
  dfWithDate.createOrReplaceTempView("dfWithDate")
  val dfNoNull = dfWithDate.drop()
  dfNoNull.show()
  dfNoNull.createOrReplaceTempView("dfNoNull")
  spark.sql(
    """
      |SELECT CustomerId, stockCode, sum(Quantity) FROM dfNoNull
      |GROUP BY customerId, stockCode
      |ORDER BY CustomerId DESC, stockCode DESC
      |""".stripMargin).show()
  spark.sql(
    """
      |SELECT CustomerId, stockCode, sum(Quantity) FROM dfNoNull
      |GROUP BY GROUPING SETS((customerId, stockCode))
      |ORDER BY CustomerId DESC, stockCode DESC
      |""".stripMargin).show()
  spark.sql(
    """
      |SELECT CustomerId, stockCode, sum(Quantity) FROM dfNoNull
      |GROUP BY GROUPING SETS((customerId, stockCode),())
```

```
      |ORDER BY CustomerId DESC, stockCode DESC
      |""".stripMargin).show()
  //+----------+---------+-------------+
  //|CustomerId|stockCode|sum(Quantity)|
  //+----------+---------+-------------+
  //|     18287|    85173|           48|
  //|     18287|   85040A|           48|
  //|     18287|   85039B|          120|
  //+----------+---------+-------------+
  dfNoNull.groupBy("Date", "Country").agg(sum("Quantity"))
    .selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")
    .orderBy("Date").show()
  //+----------+--------------+--------------+
  //|      Date|       Country|total_quantity|
  //+----------+--------------+--------------+
  //|2010-12-01|          EIRE|           243|
  //|2010-12-01|United Kingdom|         23949|
  //|2010-12-01|        Norway|          1852|
  //|2010-12-01|     Australia|           107|
  //|2010-12-01|       Germany|           117|
  //|2010-12-01|   Netherlands|            97|
  //|2010-12-01|        France|           449|
  //|2010-12-02|United Kingdom|         20873|
  //+----------+--------------+--------------+

  val rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))
    .selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")
    .orderBy("Date")
  rolledUpDF.show()
  //+----------+--------------+--------------+
  //|      Date|       Country|total_quantity|
  //+----------+--------------+--------------+
  //|      null|          null|       5176450|
  //|2010-12-01|          null|         26814|
  //|2010-12-01|     Australia|           107|
  //+----------+--------------+--------------+
  rolledUpDF.where("Country IS NULL").show()
  //+----------+-------+--------------+
  //|      Date|Country|total_quantity|
  //+----------+-------+--------------+
  //|      null|   null|       5176450|
  //|2010-12-01|   null|         26814|
  //|2010-12-02|   null|         21023|
  //+----------+-------+--------------+
  /**
   *  This is a quick and easily accessible summary of nearly all of the
information in our table,
   and it's a great way to create a quick summary table that others can use
later on.
   */
  dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))
    .select("Date", "Country", "sum(Quantity)").orderBy(desc("Date")).show()
  //+----+--------------------+-------------+
  //|null|               Japan|        25218|
  //|null|           Australia|        83653|
  //|null|            Portugal|        16180|
  //|null|             Germany|       117448|
  //|null|                 RSA|          352|
```

```
  //|null|           Hong Kong|         4769|
  //|null|              Cyprus|         6317|
  //|null|         Unspecified|         3300|
  //|null|United Arab Emirates|          982|
  //|null|                null|      5176450|
  //+----+--------------------+-------------+
}
```

group by:主要用来对查询的结果进行分组，相同组合的分组条件在结果集中只显示一行记录。可以添加聚合函数。

grouping sets：对分组集中指定的组表达式的每个子集执行group by，group by A,B grouping sets(A,B)就等价于 group by A union group by B,其中A和B也可以是一个集合，比如group by A,B,C grouping sets((A,B),(A,C))。

rollup：在指定表达式的每个层次级别创建分组集。group by A,B,C with rollup首先会对(A、B、C)进行group by，然后对(A、B)进行group by，然后是(A)进行group by，最后对全表进行group by操作。

cube:为指定表达式集的每个可能组合创建分组集。首先会对(A、B、C)进行group by，然后依次是(A、B)，(A、C)，(A)，(B、C)，(B)，(C)，最后对全表进行group by操作。

Grouping Metadata

Sometimes when using cubes and rollups, you want to be able to query the aggregation levels so that you can easily filter them down accordingly. We can do this by using the `grouping_id`, which gives us a column specifying the level of aggregation that we have in our result set. The query in the example that follows returns four distinct grouping IDs:

Table 7-1. Purpose of grouping IDs

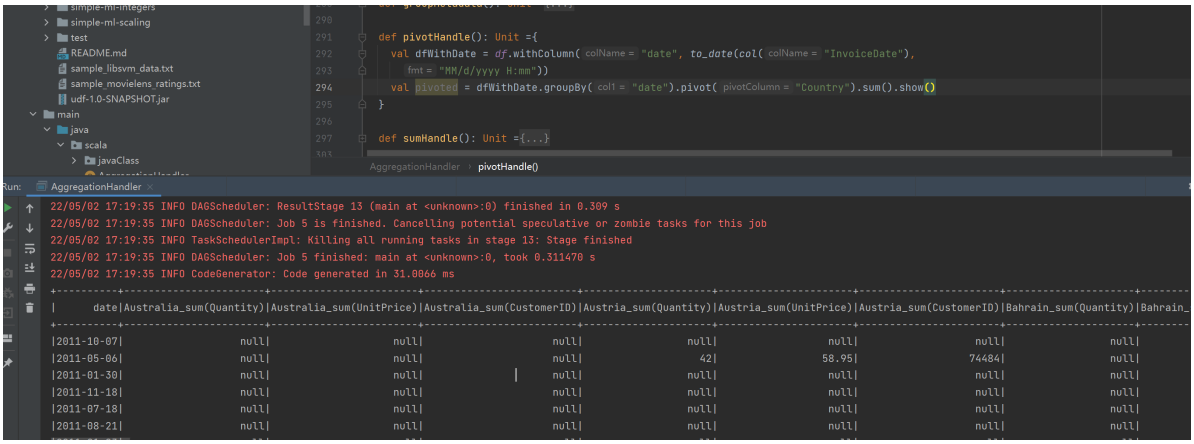
Grouping ID	Description
3	This will appear for the highest-level aggregation, which will gives us the total quantity regardless of `customerId` and `stockCode`.
2	This will appear for all aggregations of individual stock codes. This gives us the total quantity per stock code, regardless of customer.
1	This will give us the total quantity on a per-customer basis, regardless of item purchased.
0	This will give us the total quantity for individual `customerId` and `stockCode` combinations.

```
def groupMetadata(): Unit ={
  import org.apache.spark.sql.functions.{grouping_id, sum, expr}
  val dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"),
    "MM/d/yyyy H:mm"))
  dfWithDate.show(2)
  dfWithDate.createOrReplaceTempView("dfWithDate")
  val dfNoNull = dfWithDate.drop()
  dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))
    .select("Date", "Country", "sum(Quantity)").orderBy(desc("Date")).show()
  //+----------+--------------+-------------+
  //|      Date|       Country|sum(Quantity)|
  //+----------+--------------+-------------+
  //|2011-12-09|       Belgium|          203|
  //|2011-12-09|        Norway|         2227|
  //|2011-12-09| United Kingdom|         9534|
```

```
  //|2011-12-09|         France|          105|
  //|2011-12-09|           null|        12949|
  //|2011-12-09|        Germany|          880|
  //|2011-12-08|           EIRE|          806|
  //|2011-12-08|            USA|         -196|
  //|2011-12-08|         France|           18|
  //|2011-12-08|    Netherlands|          140|
  //+----------+---------------+-------------+
  dfNoNull.cube("Date", "Country").agg(grouping_id(), sum("Quantity"))
    .orderBy(desc("Date"),expr("grouping_id()").asc)
    .show()
  //+----------+---------------+-------------+-------------+
  //|      Date|        Country|grouping_id()|sum(Quantity)|
  //+----------+---------------+-------------+-------------+
  //|2011-12-09|         Norway|            0|         2227|
  //|2011-12-09| United Kingdom|            0|         9534|
  //|2011-12-09|         France|            0|          105|
  //|2011-12-09|        Belgium|            0|          203|
  //|2011-12-09|        Germany|            0|          880|
  //|2011-12-09|           null|            1|        12949| //give us the total
quantity on a day, regardless of Country.
  //|2011-12-08|           EIRE|            0|          806|
  //|2011-12-08|        Germany|            0|          969|
  //|2011-12-08|         France|            0|           18|
  //+----------+---------------+-------------+-------------+
}
```

Pivots

Pivots make it possible for you to convert a row into a column.



Window Functions

A **group-by** takes data, and **every row can go only into one grouping.**

A **window function** calculates a return value for **every input row** of a table based on a group of rows, called a frame.**Each row can fall into one or more frames.**

A common use case is to take a look at a rolling **average of some value for which each row represents one day**. If you were to do this, each row would end up in seven different frames.

Figure 7-1 illustrates how a given row can fall into multiple frames.

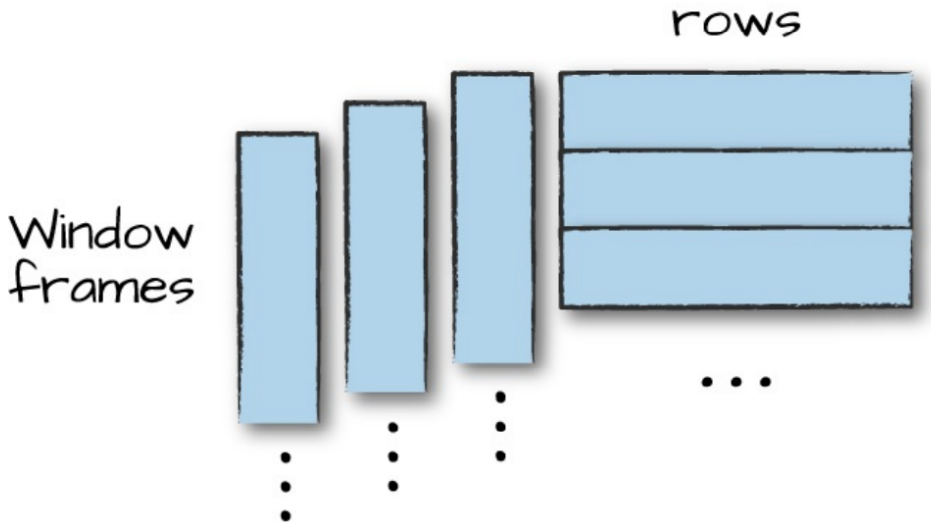


Figure 7-1. Visualizing window functions

Spark supports three kinds of window functions: **ranking functions, analytic functions, and aggregate functions**.

The first step to a window function is to create a window specification. Note that the partition by is unrelated to the partitioning scheme concept that we have covered thus far. It's just a similar concept that describes how we will be breaking up our group. The ordering determines the ordering within a given partition, and, finally, the frame specification (the rowsBetween statement) states which rows will be included in the frame based on its reference to the current input row. In the following example, we look at all previous rows up to the current row:

below is the function of rowsBetween

Defines the frame boundaries, from `start` (inclusive) to `end` (inclusive).

Both `start` and `end` are relative positions from the current row. For example, "0" means "current row", while "-1" means the row before the current row, and "5" means the fifth row after the current row.

We recommend users use `Window.unboundedPreceding`, `Window.unboundedFollowing`, and `Window.currentRow` to specify special boundary values, rather than using integral values directly.

A row based boundary is based on the position of the row within the partition. An offset indicates the number of rows above or below the current row, the frame for the current row starts or ends. For instance, given a row based sliding frame with a lower bound offset of -1 and a upper bound offset of +2. The frame for row with index 5 would range from index 4 to index 6.

```
import org.apache.spark.sql.expressions.Window
val df = Seq((1, "a"), (1, "a"), (2, "a"), (1, "b"), (2, "b"), (3, "b"))
  .toDF("id", "category")
val byCategoryOrderedById =
  Window.partitionBy('category).orderBy('id).rowsBetween(Window.currentRow, 1)
df.withColumn("sum", sum('id) over byCategoryOrderedById).show()

+---+--------+---+
| id|category|sum|
+---+--------+---+
|  1|       b|  3|
|  2|       b|  5|
|  3|       b|  3|
|  1|       a|  2|
|  1|       a|  3|
|  2|       a|  2|
+---+--------+---+
```

Params: start – boundary start, inclusive. The frame is unbounded if this is the minimum long value (`Window.unboundedPreceding`).
end – boundary end, inclusive. The frame is unbounded if this is the maximum long value (`Window.unboundedFollowing`).

```
def windowHandle(): Unit ={
  import org.apache.spark.sql.functions.{col, to_date}
  val dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"),
    "MM/d/yyyy H:mm"))
  dfWithDate.show(2)
  dfWithDate.createOrReplaceTempView("dfWithDate")

  import org.apache.spark.sql.expressions.Window
  import org.apache.spark.sql.functions.col
  val windowSpec = Window
    .partitionBy("CustomerId", "date")
    .orderBy(col("Quantity").desc)
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)
  import org.apache.spark.sql.functions.max
  val maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)
  //max('Quantity) windowspecdefinition('CustomerId
  // , 'date, 'Quantity DESC NULLS LAST
  // , specifiedwindowframe(RowFrame, unboundedpreceding$(), currentrow$()))

  //use the dense_rank function to determine which date had the maximum purchase
quantity for every customer
  import org.apache.spark.sql.functions.{dense_rank, rank}
  val purchaseDenseRank = dense_rank().over(windowSpec)
  val purchaseRank = rank().over(windowSpec)
  //Window function rank() requires window to be ordered
  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()
  //+----------+----------+--------+------------+----------------+-------------
------+
  //|CustomerId|
date|Quantity|quantityRank|quantityDenseRank|maxPurchaseQuantity|
  //+----------+----------+--------+------------+----------------+-------------
------+
  //|     12346|2011-01-18|   74215|           1|               1|
74215|
  //|     12346|2011-01-18|  -74215|           2|               2|
74215|
  //|     12347|2010-12-07|      36|           1|               1|
    36|
  //|     12347|2010-12-07|      30|           2|               2|
    36|
  //|     12347|2010-12-07|      24|           3|               3|
    36|
  //|     12347|2010-12-07|      12|           4|               4|
    36|
```

```
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|      12|           4|                4|
    36|
  //|     12347|2010-12-07|       6|          17|                5|
    36|
  //|     12347|2010-12-07|       6|          17|                5|
    36|
  //+----------+----------+--------+------------+-----------------+-------------
------+
}
```

UDAF

User-defined aggregation functions (UDAFs) are a way for users to define their own aggregation functions based on custom formulae or business rules. You can use UDAFs to compute custom calculations over groups of input data (as opposed to single rows). Spark maintains a single AggregationBuffer to store intermediate results for every group of input data.

The following is deprecated.

To create a UDAF, you must inherit from the `UserDefinedAggregateFunction` base class and implement the following methods:

- `inputSchema` represents input arguments as a `StructType`
- `bufferSchema` represents intermediate UDAF results as a `StructType`
- `dataType` represents the return `DataType`
- `deterministic` is a Boolean value that specifies whether this UDAF will return the same result for a given input
- `initialize` allows you to initialize values of an aggregation buffer
- `update` describes how you should update the internal buffer based on a given row
- `merge` describes how two aggregation buffers should be merged
- `evaluate` will generate the final result of the aggregation

```
import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
class BoolAnd extends UserDefinedAggregateFunction {
  def inputSchema: org.apache.spark.sql.types.StructType =
    StructType(StructField("value", BooleanType) :: Nil)
  def bufferSchema: StructType = StructType(
    StructField("result", BooleanType) :: Nil
  )
  def dataType: DataType = BooleanType
  def deterministic: Boolean = true
  def initialize(buffer: MutableAggregationBuffer): Unit = {
    buffer(0) = true
  }
  def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    buffer(0) = buffer.getAs[Boolean](0) && input.getAs[Boolean](0)
  }
  def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    buffer1(0) = buffer1.getAs[Boolean](0) && buffer2.getAs[Boolean](0)
  }
  def evaluate(buffer: Row): Any = {
    buffer(0)
  }
}
```

```
def udafHandle(): Unit ={
  val ba = new BoolAnd
  /**
   * this method and the use of UserDefinedAggregateFunction are deprecated.
   * Aggregator[IN, BUF, OUT] should now be registered as a UDF via the
functions.udaf(agg) method.
   */
  spark.udf.register("booland", new BoolAnd)
  //+----------+----------+
  //|booland(t)|booland(f)|
  //+----------+----------+
  //|      true|     false|
  //+----------+----------+

  //recommend
  val avgAgg = new Aggregator[Boolean, Boolean, Boolean] {
    //初始值
    override def zero: Boolean = true
    //每个分组区局部聚合的方法，
    override def reduce(b: Boolean, a: Boolean): Boolean = {
      b && a
    }
    //全局聚合调用的方法
    override def merge(b1: Boolean, b2: Boolean): Boolean = {
      b1 && b2
    }
    //计算最终的结果
    override def finish(reduction: Boolean): Boolean = {
      reduction
    }
```

```
    //中间结果的encoder
    override def bufferEncoder: Encoder[Boolean] = {
      Encoders.scalaBoolean;
    }
    //返回结果的encoder
    override def outputEncoder: Encoder[Boolean] = {
      Encoders.scalaBoolean
    }
  }
  spark.udf.register("booland", udaf(avgAgg))

  import org.apache.spark.sql.functions._
  spark.range(1)
    .selectExpr("explode(array(TRUE, TRUE, TRUE)) as t")
    .selectExpr("explode(array(TRUE, FALSE, TRUE)) as f", "t")
    .select(ba(col("t")), expr("booland(f)"))
    .show()
  //+----------+---------+
  //|booland(t)|anon$1(f)|
  //+----------+---------+
  //|      true|    false|
  //+----------+---------+

}
```

For more info, [User Defined Aggregate Functions (UDAFs) - Spark 3.2.1 Documentation (apache.org)]

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

case class Invoice(InvoiceNo:String, quantity: Long, unitPrice: Double)
case class Average(var quantity_sum: Long, var quantity_count: Long, var
unitPrice_sum: Double, var unitPrice_count: Long)

object MyAverage extends Aggregator[Invoice, Average, (Double, Double)] {
  // A zero value for this aggregation. Should satisfy the property that any b +
zero = b
  def zero: Average = Average(0L, 0L, 0.0, 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, invoice: Invoice): Average = {
    buffer.quantity_sum += invoice.quantity
    buffer.quantity_count += 1
    buffer.unitPrice_sum += invoice.unitPrice
    buffer.unitPrice_count += 1
    buffer
  }
  // Merge two intermediate values
  def merge(b1: Average, b2: Average): Average = {
    b1.quantity_count += b2.quantity_count
    b1.quantity_sum += b2.quantity_sum
    b1.unitPrice_sum += b2.unitPrice_sum
    b1.unitPrice_count += b2.unitPrice_count
    b1
  }
```

```
  // Transform the output of the reduction
  def finish(reduction: Average): (Double, Double) =
(reduction.quantity_sum.toDouble / reduction.quantity_count
    , reduction.unitPrice_sum / reduction.unitPrice_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,Double)] =
Encoders.tuple(Encoders.scalaDouble,Encoders.scalaDouble)
}
```

```
def myUDAF(): Unit ={
  import spark.implicits._
  val myDF = df.select("InvoiceNo", "Quantity", "UnitPrice").drop().as[Invoice]
  val averageSalary = MyAverage.toColumn.name("average_salary")
  myDF.agg(avg("Quantity")).show()
  //+----------------+
  //|   avg(Quantity)|
  //+----------------+
  //|9.55224954743324|
  //+----------------+
  myDF.agg(avg("UnitPrice")).show()
  //+-----------------+
  //|   avg(UnitPrice)|
  //+-----------------+
  //|4.611113626083471|
  //+-----------------+
  myDF.select(averageSalary).show()
  //+----------------+-----------------+
  //|              _1|               _2|
  //+----------------+-----------------+
  //|9.55224954743324|4.611113626083471|
  //+----------------+-----------------+
  spark.udf.register("myAvg", udaf(MyAverage))
  myDF.createOrReplaceTempView("invoices")
  myDF.selectExpr("myAvg(*)").show(false)
  //+------------------------------------------------------------+
  //|myaverage$(InvoiceNo, CAST(Quantity AS BIGINT), UnitPrice)|
  //+------------------------------------------------------------+
  //|[9.55224954743324, 4.611113626083471]                       |
  //+------------------------------------------------------------+
  spark.sql("SELECT myAvg(*) as average_salary FROM invoices").show(false)
  //+--------------------------------------+
  //|average_salary                        |
  //+--------------------------------------+
  //|[9.55224954743324, 4.611113626083471]|
  //+--------------------------------------+
}
```

Joins

A join brings together two sets of data, the left and the right, by comparing the value of one or more keys of the left and right and evaluating the result of a join expression that determines whether Spark should bring together the left set of data with the right set of data.

The most common join expression, an **equi-join**, compares whether the **specified keys** in your left and right datasets are equal.

Join Type

- Inner joins (keep rows with keys that exist in the left and right datasets)
- Outer joins (keep rows with keys in either the left or right datasets)
- Left outer joins (keep rows with keys in the left dataset)
- Right outer joins (keep rows with keys in the right dataset)
- Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)
- Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)
- Natural joins (perform a join by implicitly matching the columns between the two datasets with the same names)
- Cross (or Cartesian) joins (match every row in the left dataset with every row in the right dataset)

```
var person : DataFrame = null
var graduateProgram : DataFrame = null
var sparkStatus : DataFrame = null
  /**
   * Supported join types include:
   * 'inner', 'outer', 'full', 'fullouter', 'full_outer',
   * 'leftouter', 'left', 'left_outer', 'rightouter', 'right',
   * 'right_outer', 'leftsemi', 'left_semi', 'semi', 'leftanti',
   * 'left_anti', 'anti', 'cross'
   */
var joinType = ""
var joinExpression : Column = null

def loadData(): Unit ={
  import spark.implicits._
  person = Seq(
    (0, "Bill Chambers", 0, Seq(100)),
    (1, "Matei Zaharia", 1, Seq(500, 250, 100)),
    (2, "Michael Armbrust", 1, Seq(250, 100)))
    .toDF("id", "name", "graduate_program", "spark_status")
  graduateProgram = Seq(
    (0, "Masters", "School of Information", "UC Berkeley"),
    (2, "Masters", "EECS", "UC Berkeley"),
    (1, "Ph.D.", "EECS", "UC Berkeley"))
    .toDF("id", "degree", "department", "school")
  sparkStatus = Seq(
    (500, "Vice President"),
    (250, "PMC Member"),
    (100, "Contributor"))
    .toDF("id", "status")
  person.createOrReplaceTempView("person")
  graduateProgram.createOrReplaceTempView("graduateProgram")
  sparkStatus.createOrReplaceTempView("sparkStatus")
  joinExpression = person.col("graduate_program") === graduateProgram.col("id")
}
```

Inner Joins

Inner joins evaluate the keys in **both of the DataFrames** or tables and include (and join together) only the rows that **evaluate to true.**

```
def innerJoin(): Unit ={
  val wrongJoinExpression = person.col("name") === graduateProgram.col("school")
  joinType = "inner"
  person.join(graduateProgram, joinExpression).show()
  person.join(graduateProgram, joinExpression, joinType).show()
  //+---+----------------+----------------+---------------+---+-------+---------
----------+-----------+
  //| id|            name|graduate_program|   spark_status| id| degree|
department|     school|
  //+---+----------------+----------------+---------------+---+-------+---------
----------+-----------+
  //|  0|   Bill Chambers|               0|          [100]|  0|Masters|School of
Informa...|UC Berkeley|
  //|  2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|
       EECS|UC Berkeley|
  //|  1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|
       EECS|UC Berkeley|
  //+---+----------------+----------------+---------------+---+-------+---------
----------+-----------+
}
```

Outer Joins

Outer joins evaluate the keys in **both of the DataFrames** or tables and includes (and joins together) the rows that **evaluate to true or false.**

```
def outerJoin(): Unit ={
  joinType = "outer"
  person.join(graduateProgram, joinExpression, joinType).show()
  //+----+----------------+----------------+---------------+---+-------+--------
-----------+-----------+
  //|  id|            name|graduate_program|   spark_status| id| degree|
department|     school|
  //+----+----------------+----------------+---------------+---+-------+--------
-----------+-----------+
  //|   1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|
       EECS|UC Berkeley|
  //|   2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|
       EECS|UC Berkeley|
  //|null|            null|            null|           null|  2|Masters|
       EECS|UC Berkeley|
  //|   0|   Bill Chambers|               0|          [100]|  0|Masters|School
of Informa...|UC Berkeley|
  //+----+----------------+----------------+---------------+---+-------+--------
-----------+-----------+
}
```

Left Outer Joins

Left outer joins evaluate the keys in **both of the DataFrames or tables and includes all rows** from the **left DataFrame** as well as any rows in the right DataFrame that have a match in the left DataFrame.

```
def leftOuterJoin(): Unit ={
  joinType = "left_outer"
  graduateProgram.join(person, joinExpression, joinType).show()
  //+---+-------+--------------------+-----------+----+---------------+--------
--------+--------------+
  //| id| degree|          department|     school|  id|
name|graduate_program|   spark_status|
  //+---+-------+--------------------+-----------+----+---------------+--------
--------+--------------+
  //|  0|Masters|School of Informa...|UC Berkeley|   0|   Bill Chambers|
      0|          [100]|
  //|  2|Masters|                EECS|UC Berkeley|null|           null|
   null|           null|
  //|  1|  Ph.D.|                EECS|UC Berkeley|   2|Michael Armbrust|
      1|     [250, 100]|
  //|  1|  Ph.D.|                EECS|UC Berkeley|   1|   Matei Zaharia|
      1|[500, 250, 100]|
  //+---+-------+--------------------+-----------+----+---------------+--------
--------+--------------+
  val gradProgram2 = graduateProgram.union(Seq(
        (0, "Masters", "Duplicated", "Duplicated")).toDF())
  person.join(gradProgram2, joinExpression, joinType).show()
  //+---+----------------+---------------+---------------+---+-------+---------
----------+----------+
  //| id|            name|graduate_program|   spark_status| id| degree|
department|     school|
  //+---+----------------+---------------+---------------+---+-------+---------
----------+----------+
  //|  0|   Bill Chambers|              0|          [100]|  0|Masters|
Duplicated| Duplicated|
  //|  0|   Bill Chambers|              0|          [100]|  0|Masters|School of
Informa...|UC Berkeley|
  //|  1|   Matei Zaharia|              1|[500, 250, 100]|  1|  Ph.D.|
      EECS|UC Berkeley|
  //|  2|Michael Armbrust|              1|     [250, 100]|  1|  Ph.D.|
      EECS|UC Berkeley|
  //+---+----------------+---------------+---------------+---+-------+---------
----------+----------+
}
```

Right Outer Joins

Right outer joins evaluate the keys in both of the DataFrames or tables and **includes all rows from the right DataFrame** as well as any rows in the left DataFrame that have a match in the right DataFrame.

```
def rightOuterJoin(): Unit ={
  joinType = "right_outer"
  person.join(graduateProgram, joinExpression, joinType).show()
```

```
  //+----+----------------+----------------+---------------+---+-------+--------
-----------+-----------+
  //|  id|            name|graduate_program|   spark_status| id| degree|
department|     school|
  //+----+----------------+----------------+---------------+---+-------+--------
-----------+-----------+
  //|   0|   Bill Chambers|               0|          [100]|  0|Masters|School
of Informa...|UC Berkeley|
  //|null|            null|            null|           null|  2|Masters|
       EECS|UC Berkeley|
  //|   2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|
       EECS|UC Berkeley|
  //|   1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|
       EECS|UC Berkeley|
  //+----+----------------+----------------+---------------+---+-------+--------
-----------+-----------+
  graduateProgram.join(person, joinExpression, joinType).show()
  //+---+-------+--------------------+-----------+---+----------------+---------
-------+---------------+
  //| id| degree|          department|     school| id|
name|graduate_program|   spark_status|
  //+---+-------+--------------------+-----------+---+----------------+---------
-------+---------------+
  //|  0|Masters|School of Informa...|UC Berkeley|  0|   Bill Chambers|
      0|          [100]|
  //|  1|  Ph.D.|                EECS|UC Berkeley|  1|   Matei Zaharia|
      1|[500, 250, 100]|
  //|  1|  Ph.D.|                EECS|UC Berkeley|  2|Michael Armbrust|
      1|     [250, 100]|
  //+---+-------+--------------------+-----------+---+----------------+---------
-------+---------------+
}
```

Left Semi Joins

Semi joins are a bit of a departure from the other joins. **They do not actually include any values from the right DataFrame.**

If the value does exist, those rows will be kept in the result, even if there are duplicate keys in the left DataFrame.

Think of left semi joins as **filters** on a DataFrame.

```
def leftSemiJoin(): Unit ={
  joinType = "left_semi"
  graduateProgram.join(person, joinExpression, joinType).show()
  //+---+-------+--------------------+-----------+
  //| id| degree|          department|     school|
  //+---+-------+--------------------+-----------+
  //|  0|Masters|School of Informa...|UC Berkeley|
  //|  1|  Ph.D.|                EECS|UC Berkeley|
  //+---+-------+--------------------+-----------+
  person.join(graduateProgram, joinExpression, joinType).show()
  //+---+----------------+----------------+---------------+
  //| id|            name|graduate_program|   spark_status|
  //+---+----------------+----------------+---------------+
  //|  0|   Bill Chambers|               0|          [100]|
```

```
  //|  1|   Matei Zaharia|               1|[500, 250, 100]|
  //|  2|Michael Armbrust|               1|     [250, 100]|
  //+---+----------------+----------------+---------------+
  val gradProgram2 = graduateProgram.union(Seq(
    (0, "Masters", "Duplicated Row", "Duplicated School")).toDF())
  gradProgram2.createOrReplaceTempView("gradProgram2")
  gradProgram2.join(person, joinExpression, joinType).show()
  //+---+-------+--------------------+-----------------+
  //| id| degree|          department|           school|
  //+---+-------+--------------------+-----------------+
  //|  0|Masters|School of Informa...|      UC Berkeley|
  //|  1|  Ph.D.|                EECS|      UC Berkeley|
  //|  0|Masters|      Duplicated Row|Duplicated School|
  //+---+-------+--------------------+-----------------+
}
```

Left Anti Joins

Left anti joins are the opposite of left semi joins.Think of anti joins as a NOT IN SQL-style filter.

```
def leftAntiJoin(): Unit ={
  joinType = "left_anti"
  graduateProgram.join(person, joinExpression, joinType).show()
  //+---+-------+----------+-----------+
  //| id| degree|department|     school|
  //+---+-------+----------+-----------+
  //|  2|Masters|      EECS|UC Berkeley|
  //+---+-------+----------+-----------+
  val gradProgram2 = graduateProgram.union(Seq(
    (0, "Masters", "Duplicated Row", "Duplicated School"),
    (5, "Masters", "Duplicated Row", "Duplicated School")).toDF())
  gradProgram2.createOrReplaceTempView("gradProgram2")
  gradProgram2.join(person, joinExpression, joinType).show()
  //+---+-------+--------------+-----------------+
  //| id| degree|    department|           school|
  //+---+-------+--------------+-----------------+
  //|  2|Masters|          EECS|      UC Berkeley|
  //|  5|Masters|Duplicated Row|Duplicated School|
  //+---+-------+--------------+-----------------+
}
```

Natural Joins

Natural joins make implicit guesses at the columns on which you would like to join. (not recommend)

```
def naturalJoin(): Unit ={
  spark.sql("SELECT * FROM graduateProgram NATURAL JOIN person").show()
  //+---+-------+--------------------+-----------+---------------+------------
---+---------------+
  //| id| degree|          department|     school|
name|graduate_program|   spark_status|
  //+---+-------+--------------------+-----------+---------------+------------
---+---------------+
  //|  0|Masters|School of Informa...|UC Berkeley|   Bill Chambers|
  0|          [100]|
  //|  2|Masters|                EECS|UC Berkeley|Michael Armbrust|
  1|     [250, 100]|
  //|  1|  Ph.D.|                EECS|UC Berkeley|   Matei Zaharia|
  1|[500, 250, 100]|
  //+---+-------+--------------------+-----------+---------------+------------
---+---------------+
}
```

Cross (Cartesian) Joins

Cross-joins in simplest terms are inner joins that do not specify a predicate. **Cross joins will join every single row in the left DataFrame to ever single row in the right DataFrame.**

If you have 1,000 rows in each DataFrame, the cross-join of these will result in 1,000,000 (1,000 x 1,000) rows. For this reason, you must very explicitly state that you want a cross-join by **using the cross join keyword:**

```
def crossJoin(): Unit ={
  joinType = "cross"
  graduateProgram.join(person, joinExpression, joinType).show()
  //+---+-------+--------------------+-----------+---+---------------+---------
-------+---------------+
  //| id| degree|          department|     school| id|
name|graduate_program|   spark_status|
  //+---+-------+--------------------+-----------+---+---------------+---------
-------+---------------+
  //|  0|Masters|School of Informa...|UC Berkeley|  0|   Bill Chambers|
      0|          [100]|
  //|  1|  Ph.D.|                EECS|UC Berkeley|  2|Michael Armbrust|
      1|     [250, 100]|
  //|  1|  Ph.D.|                EECS|UC Berkeley|  1|   Matei Zaharia|
      1|[500, 250, 100]|
  //+---+-------+--------------------+-----------+---+---------------+---------
-------+---------------+
  person.crossJoin(graduateProgram).show()
  //+---+---------------+---------------+---------------+---+-------+---------
-----------+-----------+
  //| id|           name|graduate_program|   spark_status| id| degree|
department|     school|
  //+---+---------------+---------------+---------------+---+-------+---------
-----------+-----------+
  //|  0|   Bill Chambers|              0|          [100]|  0|Masters|School of
Informa...|UC Berkeley|
  //|  1|   Matei Zaharia|              1|[500, 250, 100]|  0|Masters|School of
Informa...|UC Berkeley|
```

```
  //|  2|Michael Armbrust|               1|     [250, 100]|  0|Masters|School of
Informa...|UC Berkeley|
  //|  0|   Bill Chambers|               0|          [100]|  2|Masters|
      EECS|UC Berkeley|
  //|  1|   Matei Zaharia|               1|[500, 250, 100]|  2|Masters|
      EECS|UC Berkeley|
  //|  2|Michael Armbrust|               1|     [250, 100]|  2|Masters|
      EECS|UC Berkeley|
  //|  0|   Bill Chambers|               0|          [100]|  1|  Ph.D.|
      EECS|UC Berkeley|
  //|  1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|
      EECS|UC Berkeley|
  //|  2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|
      EECS|UC Berkeley|
  //+---+----------------+----------------+---------------+---+-------+---------
----------+-----------+
}
```

> **WARNING**
>
> You should use cross-joins only if you are absolutely, 100 percent sure that this is the join you need. There is a reason why you need to be explicit when defining a cross-join in Spark. They're dangerous! Advanced users can set the session-level configuration `spark.sql.crossJoin.enable` to true in order to allow cross-joins without warnings or without Spark trying to perform another join for you.

Challenges When Using Joins

Joins on Complex Types

```
def join_on_complex_types(): Unit ={
  import org.apache.spark.sql.functions.expr
  person.withColumnRenamed("id", "personId")
    .join(sparkStatus, expr("array_contains(spark_status, id)")).show()
  //+--------+----------------+----------------+---------------+---+-----------
--+
  //|personId|            name|graduate_program|   spark_status| id|
status|
  //+--------+----------------+----------------+---------------+---+-----------
--+
  //|       0|   Bill Chambers|               0|          [100]|100|
Contributor|
  //|       1|   Matei Zaharia|               1|[500, 250, 100]|500|Vice
President|
  //|       1|   Matei Zaharia|               1|[500, 250, 100]|250|    PMC
Member|
  //|       1|   Matei Zaharia|               1|[500, 250, 100]|100|
Contributor|
  //|       2|Michael Armbrust|               1|     [250, 100]|250|    PMC
Member|
  //|       2|Michael Armbrust|               1|     [250, 100]|100|
Contributor|
  //+--------+----------------+----------------+---------------+---+-----------
--+
}
```

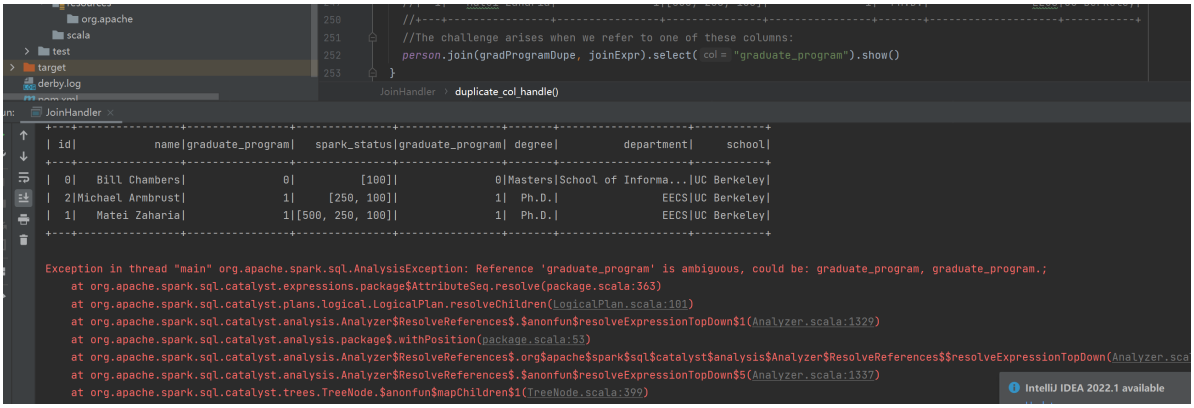
Handling Duplicate Column Names

One of the tricky things that come up in joins is dealing with **duplicate column** names in your results DataFrame.

In a DataFrame, each column has a unique ID within Spark's SQL Engine, Catalyst.

This can occur in two distinct situations:

- The join expression that you specify does not remove one key from one of the input DataFrames and the keys have the same column name
- Two columns on which you are not performing the join have the same name




```
def duplicate_col_handle(): Unit ={
  val gradProgramDupe = graduateProgram.withColumnRenamed("id",
"graduate_program")
  var joinExpr = gradProgramDupe.col("graduate_program") === person.col(
    "graduate_program")
  person.join(gradProgramDupe, joinExpr).show()
  //+---+----------------+----------------+---------------+----------------+----
---+--------------------+-----------+
  //| id|            name|graduate_program|   spark_status|graduate_program|
degree|          department|     school|
  //+---+----------------+----------------+---------------+----------------+----
---+--------------------+-----------+
  //|  0|   Bill Chambers|               0|          [100]|
0|Masters|School of Informa...|UC Berkeley|
  //|  2|Michael Armbrust|               1|     [250, 100]|               1|
Ph.D.|                EECS|UC Berkeley|
  //|  1|   Matei Zaharia|               1|[500, 250, 100]|               1|
Ph.D.|                EECS|UC Berkeley|
  //+---+----------------+----------------+---------------+----------------+----
---+--------------------+-----------+
  //The challenge arises when we refer to one of these columns:
  //person.join(gradProgramDupe, joinExpr).select("graduate_program").show()
//false
  /**
   * specific col, my Method
   */
  person.join(gradProgramDupe, joinExpr,
"right_outer").select(gradProgramDupe("graduate_program")).show() //true
  //+------person----++granProgramDupe+
  //+----------------++----------------+
  //|graduate_program||graduate_program|
  //+----------------++----------------+
  //|               0||               0|
```

```
  //|            null||               2|
  //|               1||               1|
  //|               1||               1|
  //+----------------++----------------+
  /**
   * Different join expression
   */
  person.join(gradProgramDupe,"graduate_program").select("graduate_program").sho
w()
  /**
   * Dropping the column after the join
   */
  person.join(gradProgramDupe, joinExpr).drop(person.col("graduate_program"))
    .select("graduate_program").show()
  //+----------------+
  //|graduate_program|
  //+----------------+
  //|               0|
  //|               1|
  //|               1|
  //+----------------+
  joinExpr = person.col("graduate_program") === graduateProgram.col("id")
  person.join(graduateProgram, joinExpr).drop(graduateProgram.col("id")).show()
  /**
   * Notice how the column uses the .col method instead of a column function.
   * That allows us to implicitly specify that column by its specific ID.
   */
  //+---+----------------+----------------+---------------+-------+------------
-------+-----------+
  //| id|            name|graduate_program|   spark_status| degree|
department|     school|
  //+---+----------------+----------------+---------------+-------+------------
-------+-----------+
  //|  0|   Bill Chambers|               0|          [100]|Masters|School of
Informa...|UC Berkeley|
  //|  2|Michael Armbrust|               1|     [250, 100]|  Ph.D.|
   EECS|UC Berkeley|
  //|  1|   Matei Zaharia|               1|[500, 250, 100]|  Ph.D.|
   EECS|UC Berkeley|
  //+---+----------------+----------------+---------------+-------+------------
-------+-----------+
  /**
   *  Renaming a column before the join
   */
  val gradProgram3 = graduateProgram.withColumnRenamed("id", "grad_id")
  joinExpr = person.col("graduate_program") === gradProgram3.col("grad_id")
  person.join(gradProgram3, joinExpr).show()
  //+---+----------------+----------------+---------------+-------+-------+-----
--------------+-----------+
  //| id|            name|graduate_program|   spark_status|grad_id| degree|
     department|     school|
  //+---+----------------+----------------+---------------+-------+-------+-----
--------------+-----------+
  //|  0|   Bill Chambers|               0|          [100]|
0|Masters|School of Informa...|UC Berkeley|
  //|  2|Michael Armbrust|               1|     [250, 100]|      1|  Ph.D.|
          EECS|UC Berkeley|
```

```
  //|  1|   Matei Zaharia|              1|[500, 250, 100]|      1|  Ph.D.|
          EECS|UC Berkeley|
  //+---+----------------+---------------+--------------+-------+-------+-----
--------------+-----------+
}
```

How Spark Performs Joins

To understand how Spark performs joins, you need to understand the two core resources at play: the **node-to-node communication strategy** and **per node computation strategy**

Communication Strategies

Spark approaches cluster communication in two different ways during joins. It either incurs **a shuffle join**, which results in an all-to-all communication or **a broadcast join.**

The core foundation of our simplified view of joins is that in Spark you will have either a big table or a small table. Although this is obviously a spectrum (and things do happen differently if you have a "medium-sized table"), it can help to be binary about the distinction for the sake of this explanation.

Big table–to–big table

When you join a big table to another big table, you end up with a **shuffle join**

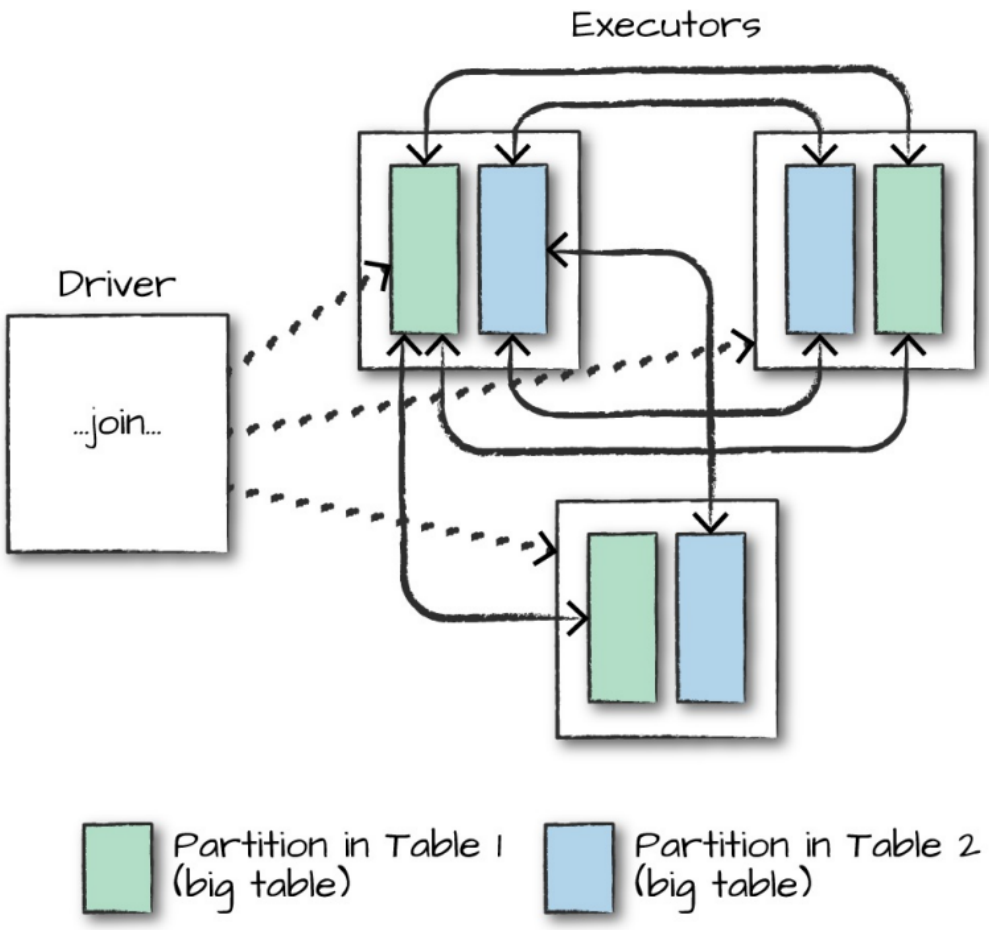


Figure 8-1. Joining two big tables

In a shuffle join, every node talks to every other node and they **share data according to which node has a certain key or set of keys** (on which you are joining). These joins are **expensive** because the **network can become congested with traffic**, especially if your data is **not partitioned well.**

Big table–to–small table

When the table is **small enough to fit into the memory of a single worker node**, with some breathing room of course, we can optimize our join. Although we can use a big table–to–big table communication strategy, it can often be **more efficient to use a broadcast join**.

We will **replicate our small DataFrame onto every worker node in the cluster**. (空间换时间)

This sounds expensive, but **prevent us from performing the all-to-all communication during the entire join process**. Instead, we perform it only once at the beginning and then let each individual worker node perform the work without having to wait or communicate with any other worker node.（并行处理）

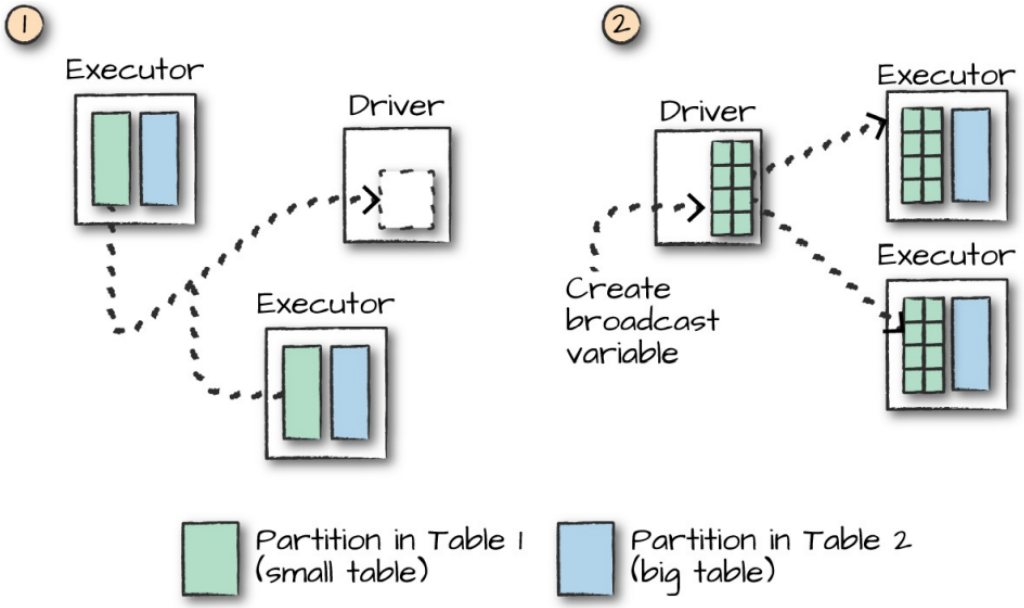


Figure 8-2. A broadcast join

This doesn't come for free either: if you try to broadcast something too large, you can **crash your driver node** (because that collect is expensive). **This is likely an area for optimization in the future.**

> With the DataFrame API, we can also explicitly give the optimizer a hint that we would like to use a broadcast join by using the correct function around the small DataFrame in question. In this example, these result in the same plan we just saw; however, this is not always the case:
>
> ```
> import org.apache.spark.sql.functions.broadcast
> val joinExpr = person.col("graduate_program") === graduateProgram.col("id")
> person.join(broadcast(graduateProgram), joinExpr).explain()
> ```
>
> The SQL interface also includes the ability to provide *hints* to perform joins. These are not *enforced*, however, so the optimizer might choose to ignore them. You can set one of these hints by using a special comment syntax. `MAPJOIN`, `BROADCAST`, and `BROADCASTJOIN` all do the same thing and are all supported:
>
> ```
> -- in SQL
> SELECT /*+ MAPJOIN(graduateProgram) */ * FROM person JOIN graduateProgram
>   ON person.graduate_program = graduateProgram.id
> ```

Little table–to–little table

When performing joins with small tables, **it's usually best to let Spark decide how to join them**. You can always force a broadcast join if you're noticing strange behavior.

> One thing we did not mention but is important to consider is if you **partition** your data **correctly prior to a join**, you can end up with much more efficient execution because even if a shuffle is planned, if data from two different DataFrames is already located on the same machine, Spark can avoid the shuffle.
>
> **Experiment with some of your data and try partitioning beforehand to see if you can notice the increase in speed when performing those joins.**

Data Source

Spark has **six "core" data sources** and **hundreds of external data sources** written by the community.

- CSV
- JSON
- Parquet
- ORC
- JDBC/ODBC connections
- Plain-text files

Spark has numerous community-created data sources.

- Cassandra
- HBase
- MongoDB
- AWS Redshift
- XML
- others

The Structure of the Data Sources API

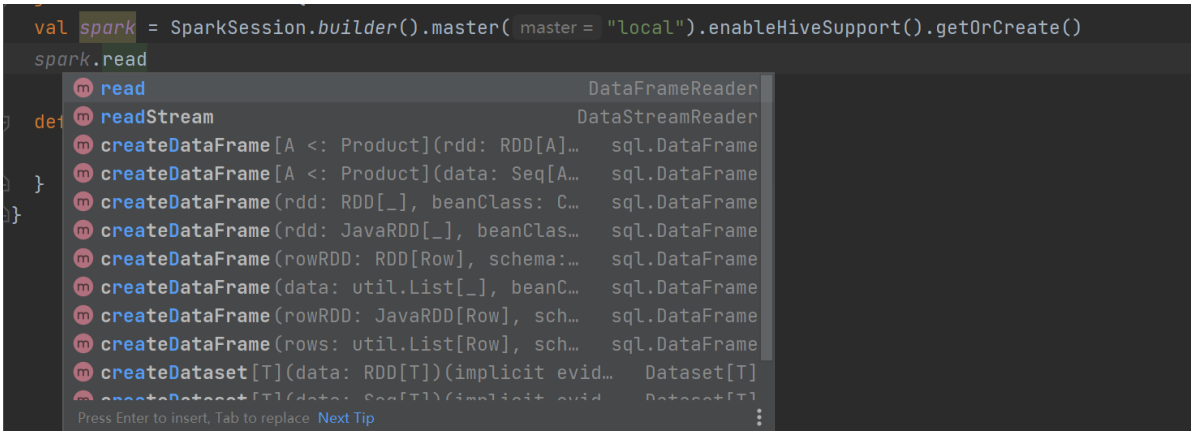
Read API Structure

```
DataFrameReader.format(...).option("key", "value").schema(...).load()
```

- **format** is optional because by default Spark will use the Parquet format.
- **option** allows you to set key-value configurations to parameterize how you will read data.
- **schema** is optional if the data source provides a schema or if you intend to use schema inference.

The foundation for reading data in Spark is the DataFrameReader. We access this through the SparkSession via the read attribute:

```
spark.read
```



The **format, options, and schema** each return a **DataFrameReader** that can undergo further transformations and are all optional, except for one option.

At a minimum, you must supply the DataFrameReader **a path** to from which to read. (路径不正确在运行时会抛出Analysis Exception, for resolve logical plan)

```
spark.read.format("csv")
.option("mode", "FAILFAST")
.option("inferSchema", "true")
.option("path", "path/to/file(s)")
.schema(someSchema) .load()
```

> **Read modes**
>
> Reading data from an external source naturally entails encountering malformed data, especially when working with only semi-structured data sources. Read modes specify what will happen when Spark does come across malformed records.
>
> *Table 9-1. Spark's read modes*
>
Read mode	Description
`permissive`	Sets all fields to `null` when it encounters a corrupted record and places all corrupted records in a string column called `_corrupt_record`
`dropMalformed`	Drops the row that contains malformed records
`failFast`	Fails immediately upon encountering malformed records

Write API Structure

```
DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(..
.).save()
```

- **format** is optional because by default, Spark will use the arquet format.
- **option**, allows us to configure how to write out our given data.
- **PartitionBy**, **bucketBy, and sortBy** work **only for file-based data sources.**

Instead of the DataFrameReader, we have the DataFrameWriter. Because we always need to write out some given data source, we access the **DataFrameWriter** on a **per-DataFrame basis via the write attribute**:

```
dataFrame.write
```

```
dataframe.write.format("csv")
.option("mode", "OVERWRITE")
.option("dateFormat", "yyyy-MM-dd")
.option("path", "path/to/file(s)")
.save()
```

> **Save modes**
>
> Save modes specify what will happen if Spark finds data at the specified location.

Save mode	Description
`append`	Appends the output files to the list of files that already exist at that location
`overwrite`	Will completely overwrite any data that already exists there
`errorIfExists`	Throws an error and fails the write if data or files already exist at the specified location
`ignore`	If data or files exist at the location, do nothing with the current DataFrame

```
result.write.mode("append").jdbc(url,tableName,props)
```

The default is **errorIfExists.** This means that if Spark finds data at the location to which you're writing, it will fail the write immediately.

CSV

CSV stands for commma-separated values. This is a common text file format in which each line represents a single record, and commas separate each field within a record. CSV files, **while seeming well structured, are actually one of the trickiest file formats you will encounter because not many assumptions can be made in production scenarios about what they contain or how they are structured**. For this reason, **the CSV reader has a large number of options.** These options give you the ability to work around issues like certain characters needing to be escaped—for example, commas inside of columns when the file is also comma-delimited or null values labeled in an unconventional way.

Table 9-3. CSV data source options

Read/write	Key	Potential values	Default	Description
Both	`sep`	Any single string character	,	The single character that is used as separator for each field and value.
Both	`header`	`true`, `false`	`false`	A Boolean flag that declares whether the first line in the file(s) are the names of the columns.
Read	`escape`	Any string character	\	The character Spark should use to escape other characters in the file.
Read	`inferSchema`	`true`, `false`	`false`	Specifies whether Spark should infer column types when reading the file.
Read	`ignoreLeadingWhiteSpace`	`true`, `false`	`false`	Declares whether leading spaces from values being read should be skipped.
Read	`ignoreTrailingWhiteSpace`	`true`, `false`	`false`	Declares whether trailing spaces from values being read should be skipped.
Both	`nullValue`	Any string character	“”	Declares what character represents a `null` value in the file.
Both	`nanValue`	Any string character	NaN	Declares what character represents a NaN or missing character in the CSV file.
Both	`positiveInf`	Any string or character	Inf	Declares what character(s) represent a positive infinite value.
Both	`negativeInf`	Any string or character	-Inf	Declares what character(s) represent a negative infinite value.
Both	`compression` or `codec`	None, `uncompressed`, `bzip2`, `deflate`, `gzip`, `lz4`, or	none	Declares what compression codec Spark should use to read or write the file.

Both	`dateFormat`	Any string or character that conforms to java's `SimpleDataFormat`.	yyyy-MM-dd	Declares the date format for any columns that are date type.
Both	`timestampFormat`	Any string or character that conforms to java's `SimpleDataFormat`.	yyyy-MM-dd'T'HH:mm:ss.SSSZZ	Declares the timestamp format for any columns that are timestamp type.
Read	`maxColumns`	Any integer	20480	Declares the maximum number of columns in the file.
Read	`maxCharsPerColumn`	Any integer	1000000	Declares the maximum number of characters in a column.
Read	`escapeQuotes`	`true`, `false`	`true`	Declares whether Spark should escape quotes that are found in lines.
Read	`maxMalformedLogPerPartition`	Any integer	10	Sets the maximum number of malformed rows Spark will log for each partition. Malformed records beyond this number will be ignored.
Write	`quoteAll`	`true`, `false`	`false`	Specifies whether all values should be enclosed in quotes, as opposed to just escaping values that have a quote character.
Read	`multiLine`	`true`, `false`	`false`	This option allows you to read multiline CSV files where each logical row in the CSV file might span multiple rows in the file itself.

Reading CSV Files

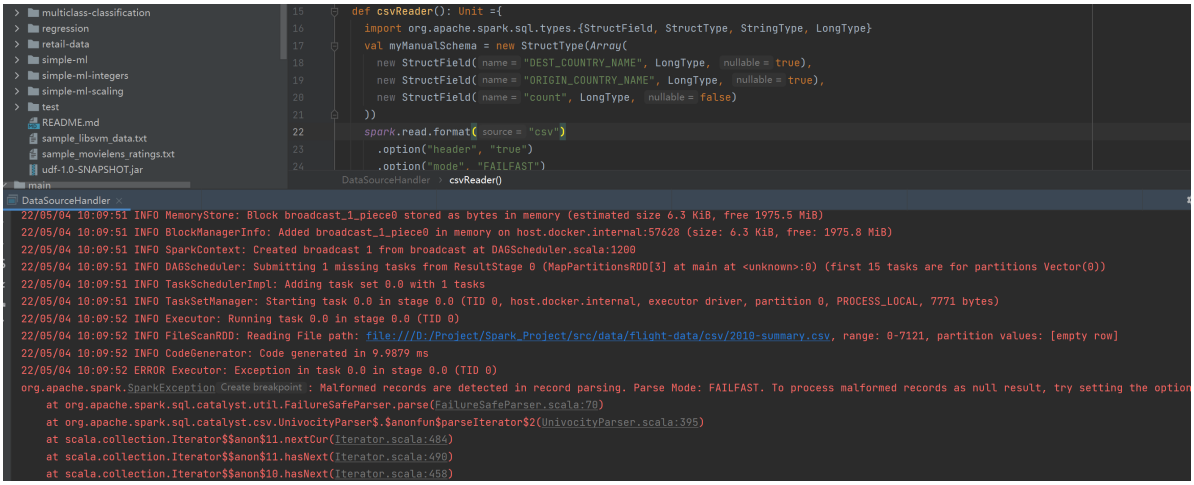
```
spark.read.format("csv")
```

```
spark.read.format("csv")
.option("header", "true")
.option("mode", "FAILFAST")
.option("inferSchema", "true")
.load("some/path/to/file.csv")
```

```
def csvReader(): Unit ={
  import org.apache.spark.sql.types.{StructField, StructType, StringType,
LongType}
  val myManualSchema = new StructType(Array(
    new StructField("DEST_COUNTRY_NAME", StringType, true),
    new StructField("ORIGIN_COUNTRY_NAME", StringType, true),
    new StructField("count", LongType, false)
  ))
  spark.read.format("csv")
    .option("header", "true")
    .option("mode", "FAILFAST")
    .schema(myManualSchema)
    .load("src/data/flight-data/csv/2010-summary.csv")
```

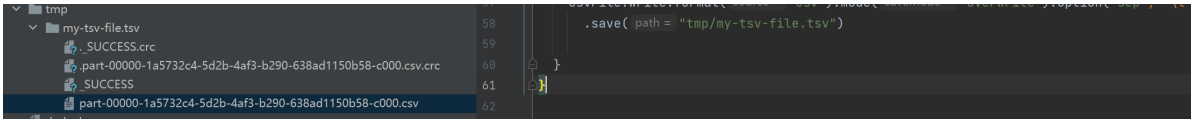
```
    .show(5)
  spark.read
    .option("header", "true")
    .option("mode", "FAILFAST")
    .schema(myManualSchema)
    .csv("src/data/flight-data/csv/2010-summary.csv")
    .show(5)
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|    United States|            Romania|    1|
  //|    United States|            Ireland|  264|
  //|    United States|              India|   69|
  //|            Egypt|      United States|   24|
  //|Equatorial Guinea|      United States|    1|
  //+-----------------+-------------------+-----+
}
```

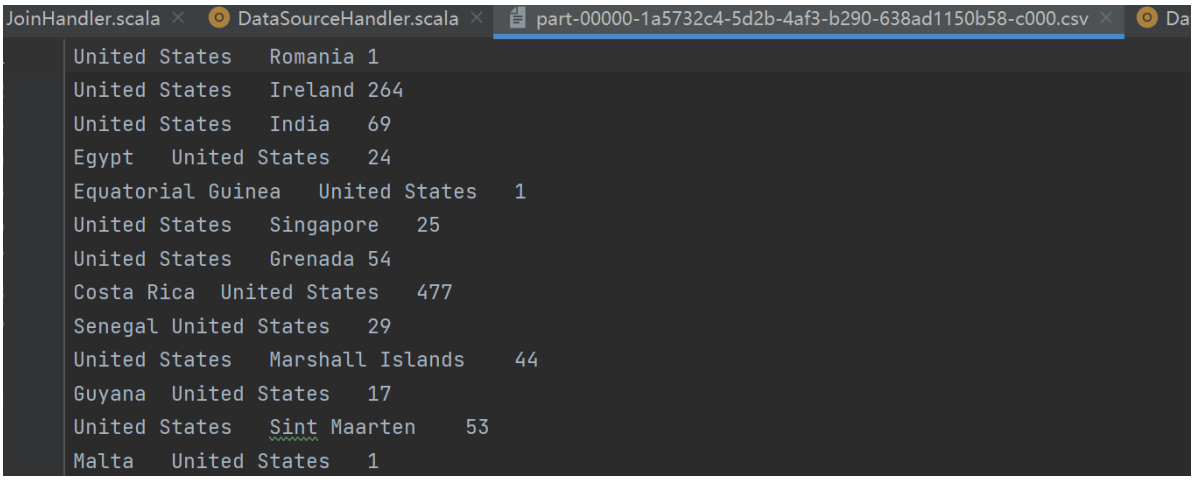
For example, let's take our current schema and change all column types to LongType. This **does not match the actual schema**, but Spark has no problem with us doing this. **The problem will only manifest itself when Spark actually reads the data**. As soon as we start our Spark job, it will immediately fail (after we execute a job) due to the data not conforming to the specified schema:



Writing CSV Files

```
def csvWriter(): Unit ={
  val myManualSchema = new StructType(Array(
    new StructField("DEST_COUNTRY_NAME", StringType, true),
    new StructField("ORIGIN_COUNTRY_NAME", StringType, true),
    new StructField("count", LongType, false)
  ))
  val csvFile = spark.read.format("csv")
    .option("header", "true").option("mode", "FAILFAST").schema(myManualSchema)
    .load("src/data/flight-data/csv/2010-summary.csv")
  //For instance, we can take our CSV file and write it out as a TSV file quite
easily:
  csvFile.write.format("csv").mode("overwrite").option("sep", "\t")
    .save("tmp/my-tsv-file.tsv")
}
```





JSON

In Spark, when we refer to JSON files, we refer to **line-delimited JSON files**. This contrasts with files that have a large JSON object or array per file.

The **line-delimited** versus **multiline** trade-off is controlled by a single option: **multiLine**

Line-delimited JSON is actually a much **more stable format** because it allows you to append to a file with a new record (rather than having to read in an entire file and then write it out), which is what we **recommend** that you use.

Another key reason for the popularity of line-delimited JSON is because **JSON objects have structure, and JavaScript (on which JSON is based) has at least basic types.**This makes it easier to work with because Spark can make more assumptions on our behalf about the data.

Table 9-4. JSON data source options

Read/write	Key	Potential values	Default
Both	`compression` or `codec`	None, `uncompressed`, `bzip2`, `deflate`, `gzip`, `lz4`, or `snappy`	none
Both	`dateFormat`	Any string or character that conforms to Java's `SimpleDataFormat`.	yyyy-MM-dd
Both	`timestampFormat`	Any string or character that conforms to Java's `SimpleDataFormat`.	yyyy-MM-dd'T'HH:mm:ss.SSSZZ

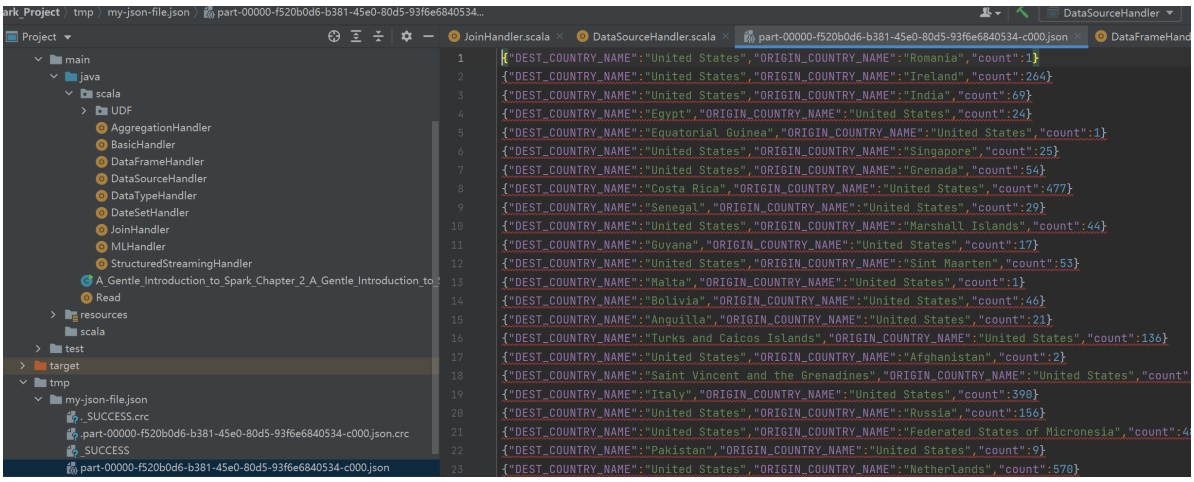
Read	primitiveAsString	true, false	false
Read	allowComments	true, false	false
Read	allowUnquotedFieldNames	true, false	false
Read	allowSingleQuotes	true, false	true
Read	allowNumericLeadingZeros	true, false	false
Read	allowBackslashEscapingAnyCharacter	true, false	false
Read	columnNameOfCorruptRecord	Any string	Value of spark.sql.column&NameOfCorruptRecord
Read	multiLine	true, false	false

Reading JSON Files

```
def jsonReader(): Unit ={
  spark.read.format("json").option("mode", "FAILFAST").schema(myManualSchema)
    .load("src/data/flight-data/json/2010-summary.json").show(5)
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|    United States|            Romania|    1|
  //|    United States|            Ireland|  264|
  //|    United States|              India|   69|
  //|            Egypt|      United States|   24|
  //|Equatorial Guinea|      United States|    1|
  //+-----------------+-------------------+-----+
}
```

Writing JSON Files

```
def jsonWriter(): Unit ={
  csvFile.write.format("json").mode("overwrite").save("tmp/my-json-file.json")
}
```



Parquet Files

Parquet is an open source **column-oriented** data store that provides a variety of **storage optimizations**, especially for analytics workloads.

We recommend writing data out to Parquet for **long-term storage** because reading from a Parquet file will always be **more efficient than JSON or CSV.**

Another advantage of Parquet is that **it supports complex types**. This means that if your column is an array (which would fail with a CSV file, for example), map, or struct, you'll still be able to read and write that file without issue.

Parquet has very few options because it **enforces its own schema when storing data**.

We **can set the schema** if we have **strict requirements** for what our DataFrame should look like.Oftentimes this is not necessary because we can use **schema** on read, which is similar to the **inferSchema** with CSV files. with Parquet files, this method is more powerful because **the schema is built into the file itself**.

[大数据：Parquet文件存储格式 - ^TONY^ - 博客园 (cnblogs.com)]

Reading Parquet Files

There are very few Parquet options—precisely two, in fact—because it has a well-defined specification that aligns closely with the concepts in Spark.

```
def parquetReader(): Unit ={
  spark.read.format("parquet")
    .load("src/data/flight-data/parquet/2010-summary.parquet").show(5)
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|    United States|            Romania|    1|
  //|    United States|            Ireland|  264|
  //|    United States|              India|   69|
  //|            Egypt|      United States|   24|
  //|Equatorial Guinea|      United States|    1|
  //+-----------------+-------------------+-----+
}
```

Read/Write	Key	Potential Values	Default	Description
Write	`compression` or `codec`	None, `uncompressed`, `bzip2`, `deflate`, `gzip`, `lz4`, or `snappy`	None	Declares what compression codec Spark should use to read or write the file.
Read	`mergeSchema`	`true`, `false`	Value of the configuration `spark.sql.parquet.mergeSchema`	You can incrementally add columns to newly written Parquet files in the same table/folder. Use this option to enable or disable this feature.

WARNING

Even though there are only two options, you can still encounter problems if you're working with incompatible Parquet files. Be careful when you write out Parquet files with different versions of Spark (especially older ones) because this can cause significant headache.

Writing Parquet Files

```
def parquetWriter(): Unit ={
  csvFile.write.format("parquet").mode("overwrite")
    .save("tmp/my-parquet-file.parquet")
  /**
    22/05/04 10:43:25 INFO ParquetWriteSupport: Initialized Parquet WriteSupport
with Catalyst schema:
    {
      "type" : "struct",
      "fields" : [ {
        "name" : "DEST_COUNTRY_NAME",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
```

```
      }, {
        "name" : "ORIGIN_COUNTRY_NAME",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "count",
        "type" : "long",
        "nullable" : true,
        "metadata" : { }
      } ]
    }
    and corresponding Parquet message type:
    message spark_schema {
      optional binary DEST_COUNTRY_NAME (UTF8);
      optional binary ORIGIN_COUNTRY_NAME (UTF8);
      optional int64 count;
    }
   */
}
```

ORC Files

ORC is a **self-describing, type-aware columnar file format** designed for Hadoop workloads. It is **optimized for large streaming reads**, but with integrated **support for finding required rows quickly.**

ORC actually **has no options** for reading in data because Spark understands the file format quite well.

What is the difference between ORC and Parquet?

For the most part, they're quite similar; the fundamental difference is that **Parquet is further optimized for use with Spark**, whereas **ORC is further optimized for Hive**.

Reading Orc Files

```
def orcReader(): Unit ={
  spark.read.format("orc").load("src/data/flight-data/orc/2010-
summary.orc").show(5)
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|    United States|            Romania|    1|
  //|    United States|            Ireland|  264|
  //|    United States|              India|   69|
  //|            Egypt|      United States|   24|
  //|Equatorial Guinea|      United States|    1|
  //+-----------------+-------------------+-----+
}
```

Writing Orc Files

```
def orcWriter(): Unit ={
  csvFile.write.format("orc").mode("overwrite").save("tmp/my-json-file.orc")
}
```

SQL Databases

SQL datasources are one of the more powerful connectors because there are a variety of systems to which you can connect .

For instance you can connect to **a MySQL database**, a **PostgreSQL database**, or an **Oracle database**. You also can connect to **SQLite**.(JDBC, the official website to SQLite as an example, but I did not install, the following writing are **Mysql**)

Databases aren't just a set of raw files, so there are **more options to consider regarding how you connect to the database.**

To read and write from these 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.

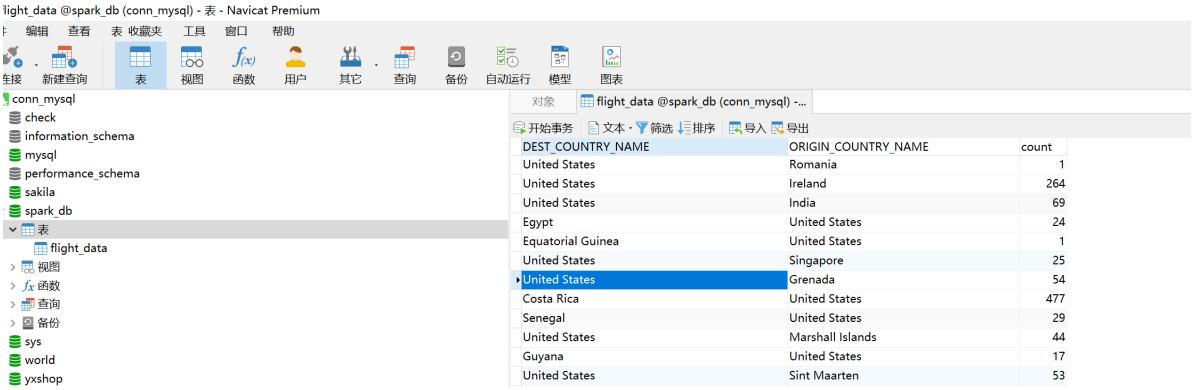
```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.18</version>
</dependency>
```

Table 9-6. JDBC data source options

Property Name	Meaning
`url`	The JDBC URL to which to connect. The source-specific connection properties can be specified in the URL; for example, *jdbc:postgresql://localhost/test?user=fred&password=secret*.
`dbtable`	The JDBC table to read. Note that anything that is valid in a FROM clause of a SQL query can be used. For example, instead of a full table you could also use a subquery in parentheses.
`driver`	The class name of the JDBC driver to use to connect to this URL.
`partitionColumn, lowerBound, upperBound`	If any one of these options is specified, then all others must be set as well. In addition, `numPartitions` must be specified. These properties describe how to partition the table when reading in parallel from multiple workers. `partitionColumn` must be a numeric column from the table in question. Notice that `lowerBound` and `upperBound` are used only to decide the partition stride, not for filtering the rows in the table. Thus, all rows in the table will be partitioned and returned. This option applies only to reading.
`numPartitions`	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections. If the number of partitions to write exceeds this limit, we decrease it to this limit by calling `coalesce(numPartitions)` before writing.
`fetchsize`	The JDBC fetch size, which determines how many rows to fetch per round trip. This can help performance on JDBC drivers, which default to low fetch size (e.g., Oracle with 10 rows). This option applies only to reading.
	The JDBC batch size, which determines how many rows to insert per round trip.

batchsize	This can help performance on JDBC drivers. This option applies only to writing. The default is 1000.
isolationLevel	The transaction isolation level, which applies to current connection. It can be one of NONE, READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ, or SERIALIZABLE, corresponding to standard transaction isolation levels defined by JDBC's Connection object. The default is READ_UNCOMMITTED. This option applies only to writing. For more information, refer to the documentation in java.sql.Connection.
truncate	This is a JDBC writer-related option. When SaveMode.Overwrite is enabled, Spark truncates an existing table instead of dropping and re-creating it. This can be more efficient, and it prevents the table metadata (e.g., indices) from being removed. However, it will not work in some cases, such as when the new data has a different schema. The default is false. This option applies only to writing.
createTableOptions	This is a JDBC writer-related option. If specified, this option allows setting of database-specific table and partition options when creating a table (e.g., CREATE TABLE t (*name string*) ENGINE=InnoDB). This option applies only to writing.
createTableColumnTypes	The database column data types to use instead of the defaults, when creating the table. Data type information should be specified in the same format as CREATE TABLE columns syntax (e.g., "name CHAR(64), comments VARCHAR(1024)"). The specified types should be valid Spark SQL data types. This option applies only to writing.

Reading from SQL Databases




```
def sqlReader(): Unit ={
  val url = "jdbc:mysql://127.0.0.1:3306/spark_db?
characterEncoding=utf8&serverTimezone=Asia/Shanghai&useSSL=false"
  val tableName = "flight_data"
  val props = new java.util.Properties()
  props.put("user","root")
  props.put("password","ROOTroot_1")
  props.put("driver","com.mysql.cj.jdbc.Driver")
  val connection = DriverManager.getConnection(url, props)
  val dbDataFrame = spark.read.jdbc(url, tableName, props)
  dbDataFrame.show(5) //recommend
  //not recommend
  spark.read.format("jdbc").option("url", url)
    .option("dbtable", tableName)
    .option("driver", "com.mysql.cj.jdbc.Driver")
    .option("user", "root")
    .option("password", "ROOTroot_1")
    .load().show(5)
  ////+-----------------+-------------------+-----+
  //    //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
  //    //+-----------------+-------------------+-----+
  //    //|    United States|            Romania|    1|
```

```
  //    //|    United States|            Ireland|  264|
  //    //|    United States|              India|   69|
  //    //|            Egypt|      United States|   24|
  //    //|Equatorial Guinea|      United States|    1|
  //    //+-----------------+-------------------+-----+
  connection.close()
}
```

Query Pushdown

First, Spark makes a best-effort attempt to **filter data in the database itself before creating the DataFrame.**

```
    dbDataFrame.select("DEST_COUNTRY_NAME").distinct().explain
    /**
 == Physical Plan ==
(2) HashAggregate(keys=[DEST_COUNTRY_NAME#6], functions=[])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#6, 200), true, [id=#33]
   +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#6], functions=[])
      +- *(1) Scan JDBCRelation(flight_data) [numPartitions=1]
[DEST_COUNTRY_NAME#6] PushedFilters: [], ReadSchema:
struct<DEST_COUNTRY_NAME:string>
     */
    dbDataFrame.filter("DEST_COUNTRY_NAME in ('Anguilla', 'Sweden')").explain
    /*
    == Physical Plan ==
*(1) Scan JDBCRelation(flight_data)
[DEST_COUNTRY_NAME#6,ORIGIN_COUNTRY_NAME#7,count#8L] PushedFilters:
[*In(DEST_COUNTRY_NAME, [Anguilla,Sweden])], ReadSchema:
struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:bigint>
     */
```

Spark can't translate all of its own functions into the functions available in the SQL database in which you're working. Therefore, sometimes you're going to want to pass an entire query into your SQL that will return the results as a DataFrame.

Spark can't translate all of its own functions into the functions available in the SQL database in which you're working. Therefore, sometimes you're going to want to **pass an entire query into your SQL** that will return the results as a DataFrame.

```
val pushdownQuery = """(SELECT DISTINCT(DEST_COUNTRY_NAME) FROM flight_data) as
flight_data"""
spark.read.jdbc(url, pushdownQuery, props).explain()
//== Physical Plan ==
//*(1) Scan JDBCRelation((SELECT DISTINCT(DEST_COUNTRY_NAME) FROM flight_data) as
flight_data) [numPartitions=1] [DEST_COUNTRY_NAME#45] PushedFilters: [],
ReadSchema: struct<DEST_COUNTRY_NAME:string>
```

> as flight_data is necessary, it's the alias name of the derived table

All throughout this book, we have talked about **partitioning** and its importance in data processing. Spark has an underlying algorithm that can **read multiple files into one partition**, or conversely, **read multiple partitions out of one file**, depending on the file size and the "splitability" of the file type and compression. The same flexibility that exists with files, also exists with SQL databases except that you must configure it a bit more manually. What you can

configure, as seen in the previous options, is the ability to specify a maximum number of partitions to allow you to limit how much you are **reading and writing in parallel**:

```
spark.read.option("numPartitions", 10).jdbc(url, tableName, props).explain()
```

In this case, this will still remain as one partition because there is not too much data.

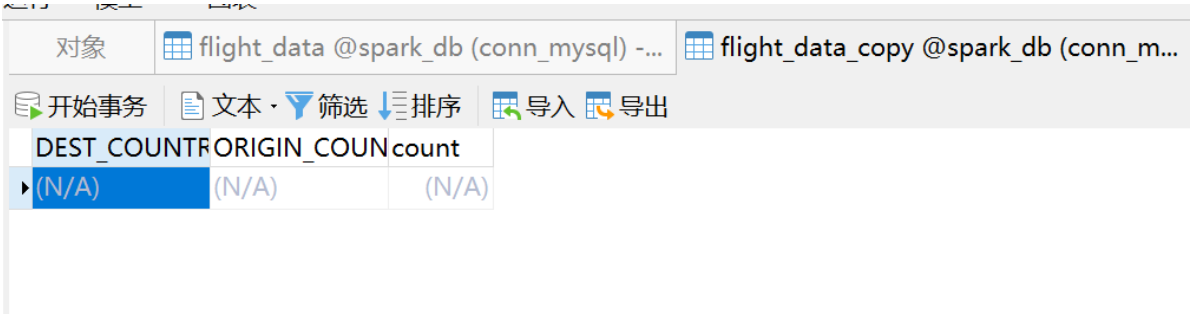
```
var predicates = Array(
  "DEST_COUNTRY_NAME = 'Sweden' OR ORIGIN_COUNTRY_NAME = 'Sweden'",
  "DEST_COUNTRY_NAME = 'Anguilla' OR ORIGIN_COUNTRY_NAME = 'Anguilla'")
spark.read.jdbc(url, tableName, predicates, props).show()
//+-----------------+-------------------+-----+
//|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
//+-----------------+-------------------+-----+
//|           Sweden|      United States|   65|
//|    United States|             Sweden|   73|
//|         Anguilla|      United States|   21|
//|    United States|           Anguilla|   20|
//+-----------------+-------------------+-----+
println(spark.read.jdbc(url, tableName, predicates, props).rdd.getNumPartitions)
//2

predicates = Array(
  "DEST_COUNTRY_NAME != 'Sweden' OR ORIGIN_COUNTRY_NAME != 'Sweden'",
  "DEST_COUNTRY_NAME != 'Anguilla' OR ORIGIN_COUNTRY_NAME != 'Anguilla'")
println(spark.read.jdbc(url, tableName, predicates, props).count()) //510
```

Partitioning based on a sliding window (没看懂)

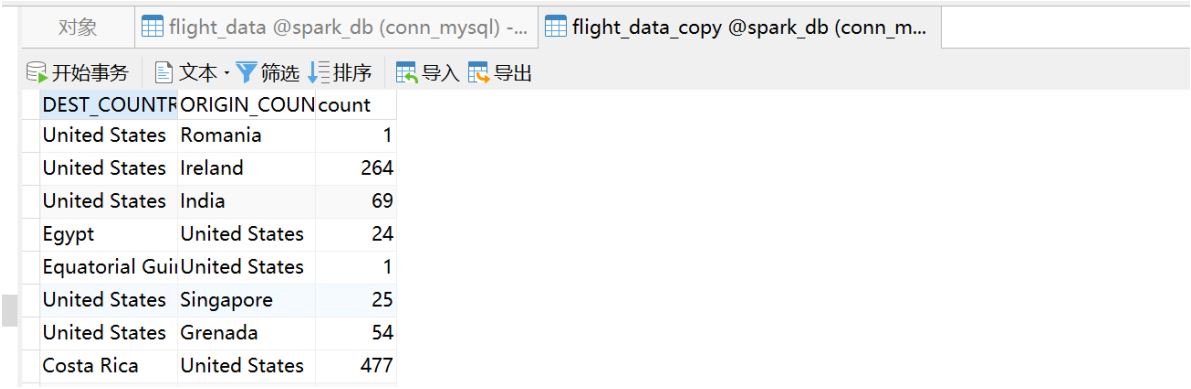
Let's take a look to see how we can partition based on predicates. In this example, we'll partition based on our numerical count column. Here, we specify a minimum and a maximum for both the first partition and last partition. Anything outside of these bounds will be in the first partition or final partition. Then, we set the number of partitions we would like total (this is the level of parallelism). Spark then queries our database in parallel and returns numPartitions partitions. We simply modify the upper and lower bounds in order to place certain values in certain partitions. No filtering is taking place like we saw in the previous example:

```
val colName = "count"
val lowerBound = 0L
val upperBound = 348113L // this is the max count in our database
val numPartitions = 10
println(spark.read.jdbc(url, tableName, colName, lowerBound, upperBound,
numPartitions, props)
  .count()) //255
```

Writing to SQL Databases



```
def sqlWriter(): Unit ={
  val url = "jdbc:mysql://127.0.0.1:3306/spark_db?
characterEncoding=utf8&serverTimezone=Asia/Shanghai&useSSL=false"
  val tableName = "flight_data_copy"
  val props = new java.util.Properties()
  props.put("user","root")
  props.put("password","ROOTroot_1")
  props.put("driver","com.mysql.cj.jdbc.Driver")
  csvFile.write.mode("overwrite").jdbc(url, tableName, props)
}
```



DEST_COUNTR	ORIGIN_COUN	count
United States	Romania	1
United States	Ireland	264
United States	India	69
Egypt	United States	24
Equatorial Gui	United States	1
United States	Singapore	25
United States	Grenada	54
Costa Rica	United States	477

```
def sqlWriter(): Unit ={
  val url = "jdbc:mysql://127.0.0.1:3306/spark_db?
characterEncoding=utf8&serverTimezone=Asia/Shanghai&useSSL=false"
  val tableName = "flight_data_copy"
  val props = new java.util.Properties()
  props.put("user","root")
  props.put("password","ROOTroot_1")
  props.put("driver","com.mysql.cj.jdbc.Driver")
  csvFile.write.mode("overwrite").jdbc(url, tableName, props) //255
  csvFile.write.mode("append").jdbc(url, tableName, props) //510, 去重问题
  csvFile.write.mode("overwrite").jdbc(url, tableName, props) //255
}
```

Text Files

Spark also allows you to read in plain-text files. Each line in the file becomes a record in the DataFrame.

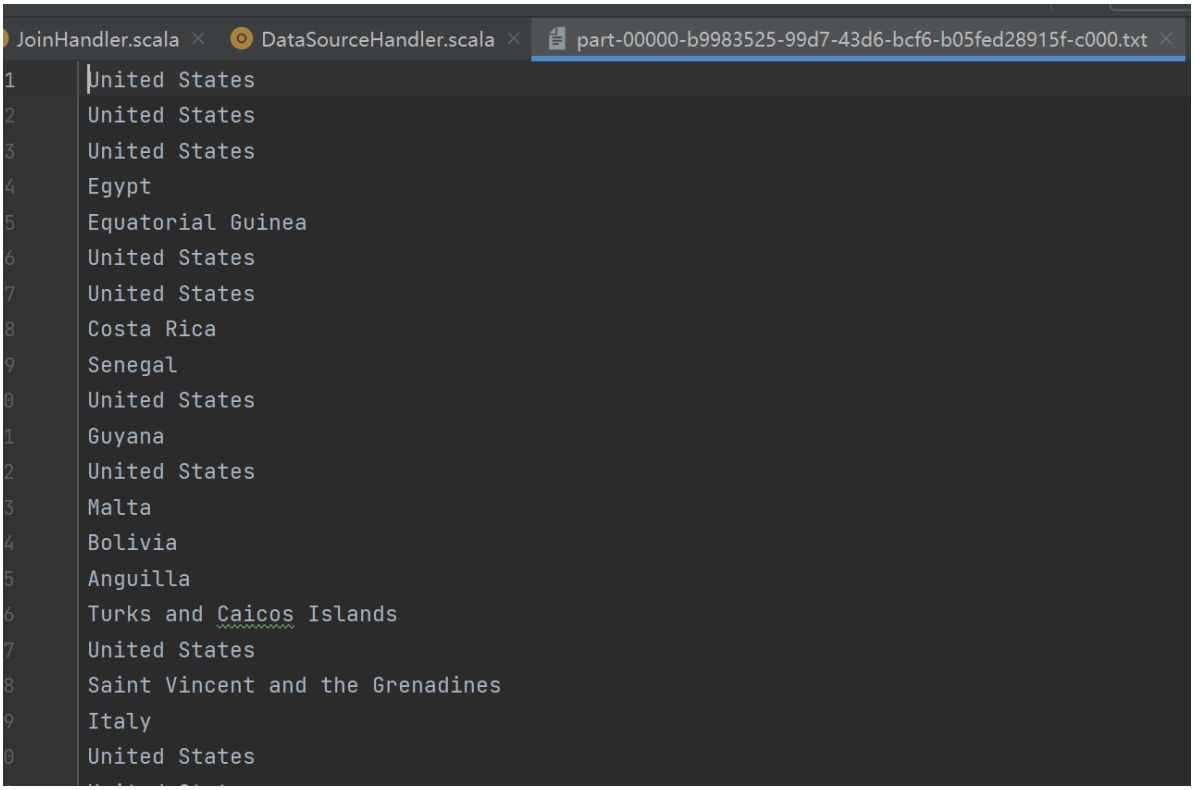
Text files make a great argument for the Dataset API due to its ability to take advantage of the flexibility of native types.

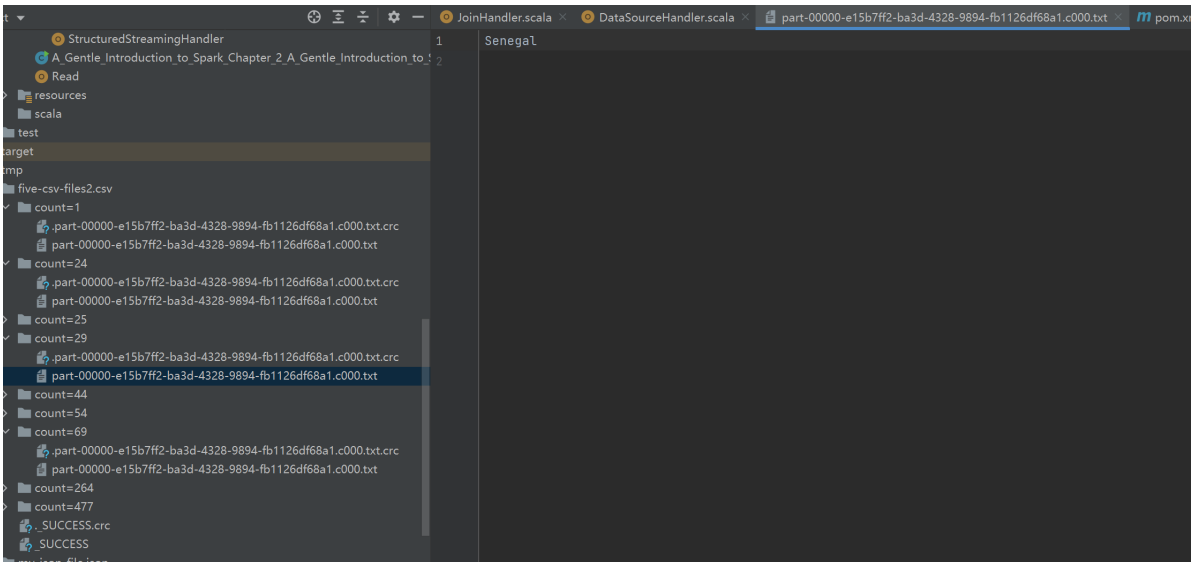
```
case class Price(region:String, market_name:String, variety_name:String,
lowest_price:String, highest_price:String, avg_price:String, date:String)
```

```
val sourceRdd=
spark.sparkContext.textFile("hdfs://hadoop01:9000/flume/*/").map(_.split(","))
  .map(x => Price(x(0),x(1),x(2),x(3),x(4),x(5),x(6))).distinct()
val sourceDF = spark.createDataFrame(sourceRdd)
```

```
def textFileReader(): Unit ={
  spark.read.textFile("src/data/flight-data/csv/2010-summary.csv")
    .selectExpr("split(value, ',') as rows").show(2,false)
```

```
  //+-------------------------------------------------+
  //|rows                                             |
  //+-------------------------------------------------+
  //|[DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count]|
  //|[United States, Romania, 1]                      |
  //+-------------------------------------------------+
}

def textFileWriter(): Unit ={
  csvFile.select("DEST_COUNTRY_NAME").write.mode("overwrite").text("tmp/simple-text-file.txt")
  csvFile.limit(10).select("DEST_COUNTRY_NAME", "count")
  .write.partitionBy("count").text("tmp/five-csv-files2.csv")
}
```





Advanced I/O Concepts

Splittable File Types and Compression

Certain file formats are fundamentally "splittable." This can improve speed because it makes it possible for Spark to **avoid reading an entire file**, and **access only the parts of the file necessary** to satisfy your query.

In conjunction with this is a need to manage **compression**. **Not all compression schemes are splittable**. How you store your data is of immense consequence when it comes to making your Spark jobs run smoothly. We recommend **Parquet with gzip compression**.

Reading Data in Parallel

Multiple executors cannot read from the same file at the same time necessarily, but they can read different files at the same time. In general, this means that when you **read from a folder with multiple files** in it, **each one of those files will become a partition** in your DataFrame and be read in by available executors in parallel (with the remaining queueing up behind the others).

Spark Partitioning & Partition Understanding - Spark by {Examples} (sparkbyexamples.com)

3.1 Local mode

When you running on local in standalone mode, Spark partitions data into the number of CPU cores you have on your system or the value you specify at the time of creating `SparkSession` object

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com') \
        .master("local[5]").getOrCreate()
```

The above example provides `local[5]` as an argument to `master()` method meaning to run the job locally with 5 partitions. Though if you have just 2 cores on your system, it still creates 5 partition tasks.

Python Copy

```
df = spark.range(0,20)
print(df.rdd.getNumPartitions())
```

Above example yields output as 5 partitions.

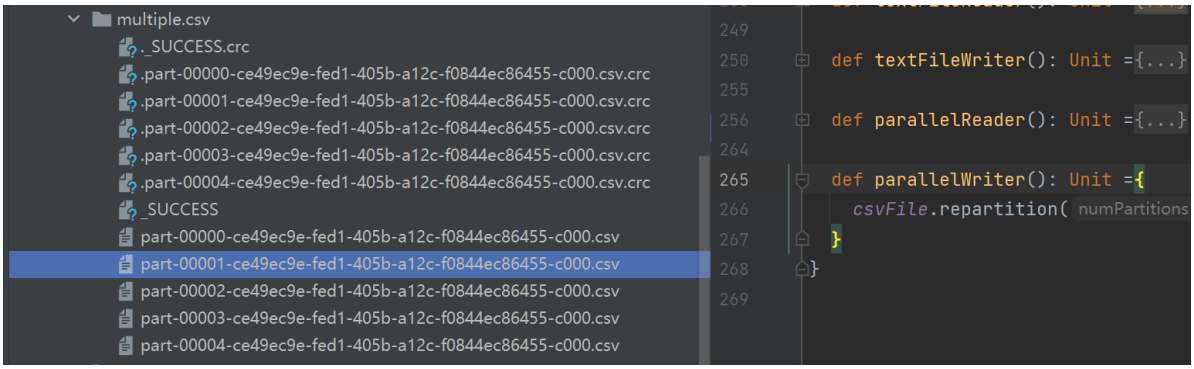
```
val spark =
SparkSession.builder().master("local[*]").enableHiveSupport().getOrCreate()
```

```
def parallelReader(): Unit ={
  println(spark.read.format("csv")
    .option("header", "true")
    .option("mode", "FAILFAST")
    .schema(myManualSchema)
    .load("src/data/flight-data/csv/*.csv")
    .rdd.getNumPartitions) //6
}
```

Writing Data in Parallel

The number of files or data written is dependent on the **number of partitions the DataFrame has** at the time you write out the data.

```
def parallelWriter(): Unit ={
  csvFile.repartition(5).write.format("csv").save("tmp/multiple.csv")
}
```

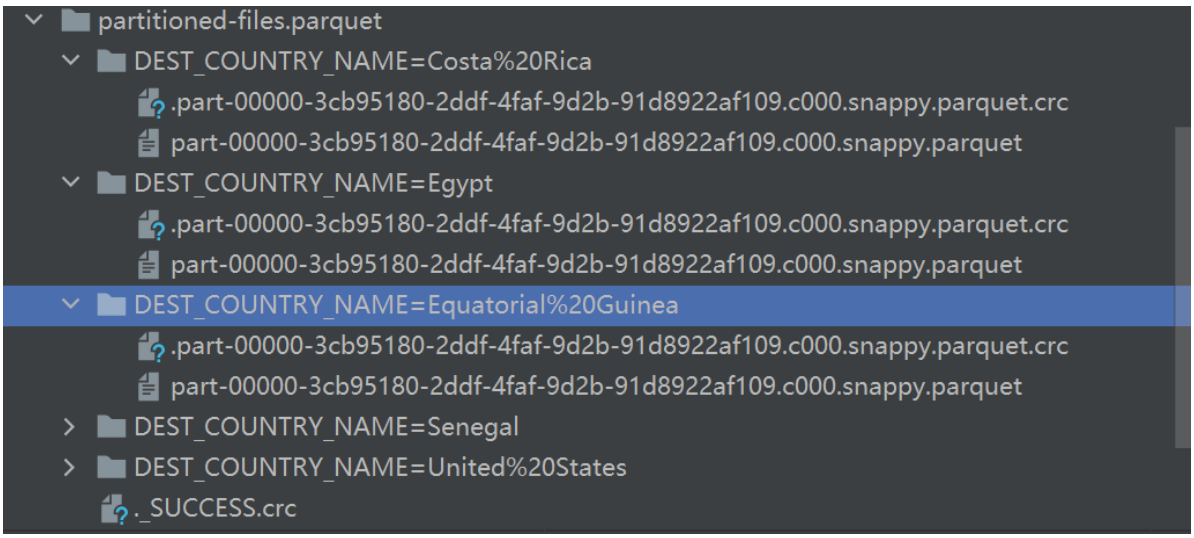


Partitioning

Partitioning is a tool that allows you to control what data is stored (and where) as you write it

When you write a file to a partitioned directory (or table), you basically **encode a column as a folder.**

```
csvFile.limit(10).write.mode("overwrite").partitionBy("DEST_COUNTRY_NAME")
  .save("tmp/partitioned-files.parquet")
```



This is probably the lowest-hanging optimization that you can use when you have a table that readers frequently filter by before manipulating. often we want to look at only the previous week's data (instead of scanning the entire list of records). This can **provide massive speedups** for readers.

Bucketing

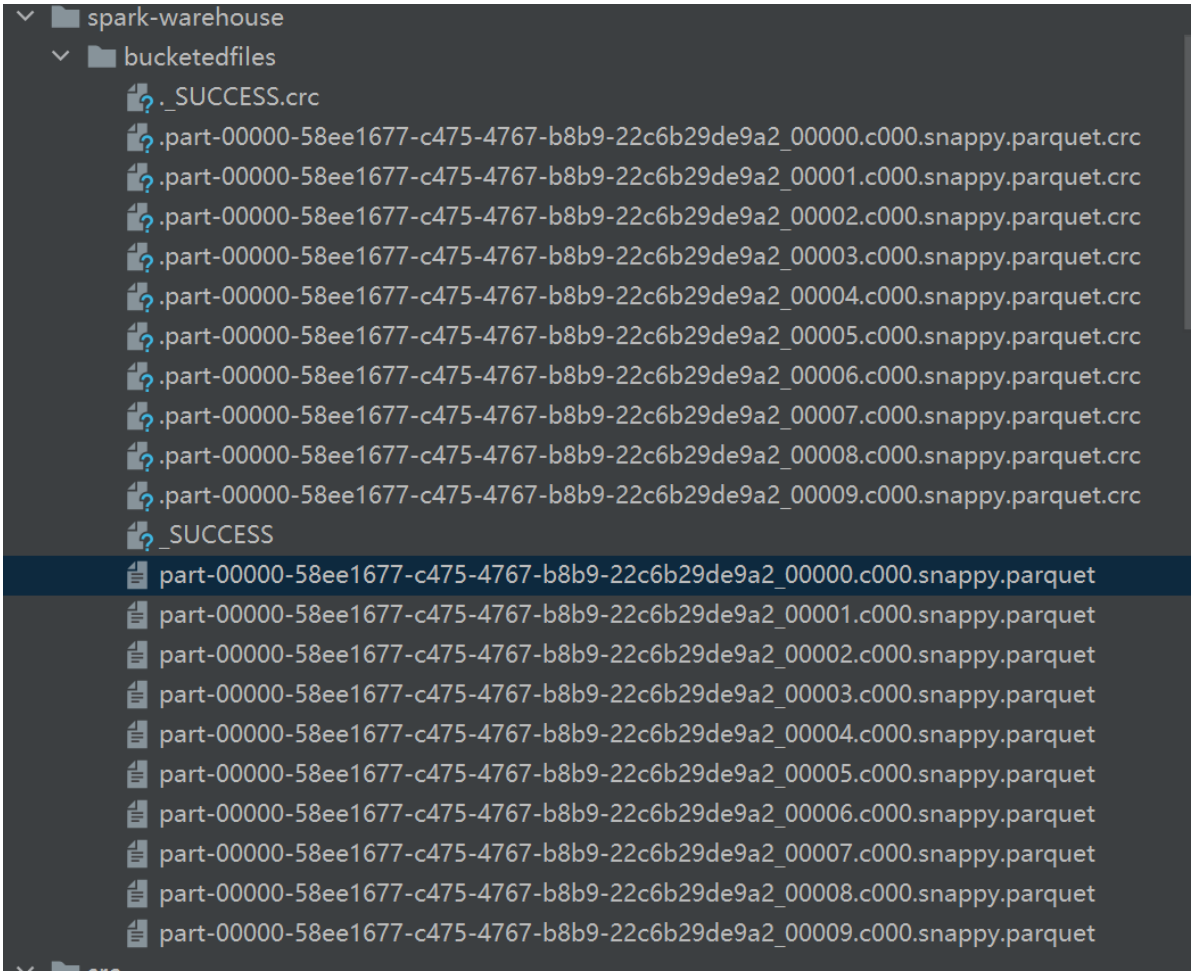
Bucketing is another file **organization approach** with which you can **control the data** that is specifically written to each file.

This can help **avoid shuffles** later when you go to read the data because **data with the same bucket ID will all be grouped together into one physical partition.**

Avoid expensive shuffles when joining or aggregating.

Rather than partitioning on a specific column (which might write out a ton of directories), it's probably worthwhile to explore bucketing the data instead. This will **create a certain number of files and organize our data into those "buckets":**

```
val numberBuckets = 10
val columnToBucketBy = "count"
csvFile.write.format("parquet").mode("overwrite")
  .bucketBy(numberBuckets, columnToBucketBy).saveAsTable("bucketedFiles")
```



> Bucketing is supported **only for Spark-managed tables.**

Writing Complex Types

Although Spark can work with all of these types, **not every single type works well with every data file format**. For instance, CSV files do not support complex types, whereas Parquet and ORC do

Managing File Size

Managing file sizes is **an important factor** not so much for writing data but **reading** it later on.

When you're writing **lots of small files**, there's a significant **metadata overhead** that you incur managing all of those files.

You might hear this referred to as the "**small file problem**." The opposite is also true: you **don't want files that are too large** either, because it becomes **inefficient to have to read entire blocks of data** when you **need only a few rows.**

maxRecordsPerFile option allows you to better control file sizes by controlling the number of records that are written to each file. if you set an option for a writer as df.write**.option("maxRecordsPerFile", 5000)**, Spark will ensure that files will contain **at most 5,000 records.**

> we omitted instructions for how to do this because the API is currently evolving to better support Structured Streaming. If you're interested in seeing how to implement your own custom data sources, the **Cassandra Connector** is well organized and maintained and could provide a reference for the adventurous.

Spark SQL

Spark SQL is arguably one of the most important and powerful features in Spark.

SQL or Structured Query Language is a domain-specific language for expressing relational operations over data.

The power of Spark SQL derives from several key facts: **SQL analysts can now take advantage of Spark's computation abilities** by plugging into the Thrift Server or Spark's SQL interface, whereas data engineers and scientists can use Spark SQL where appropriate in any data flow. **This unifying API 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.**

> **NOTE**
>
> Spark SQL is intended to operate as an online analytic processing (OLAP) database, not an online transaction processing (OLTP) database. This means that it is not intended to perform extremely low-latency queries. Even though support for in-place modifications is sure to be something that comes up in the future, it's not something that is currently available.

Spark's Relationship to Hive

Spark SQL has a great relationship with Hive because it can connect to Hive metastores.

With Spark SQL, you can connect to your Hive metastore (if you already have one) and access table metadata to reduce file listing when accessing information.

How to Run Spark SQL Queries

Spark SQL CLI

The Spark SQL CLI is a convenient tool with which you can make basic Spark SQL queries in local mode from the command line. Note that the Spark SQL CLI cannot communicate with the Thrift JDBC server. To start the Spark SQL CLI, run the following in the Spark directory:

```
./bin/spark-sql
```

You configure Hive by placing your *hive-site.xml*, *core-site.xml*, and *hdfs-site.xml* files in *conf/*. For a complete list of all available options, you can run `./bin/spark-sql --help`.

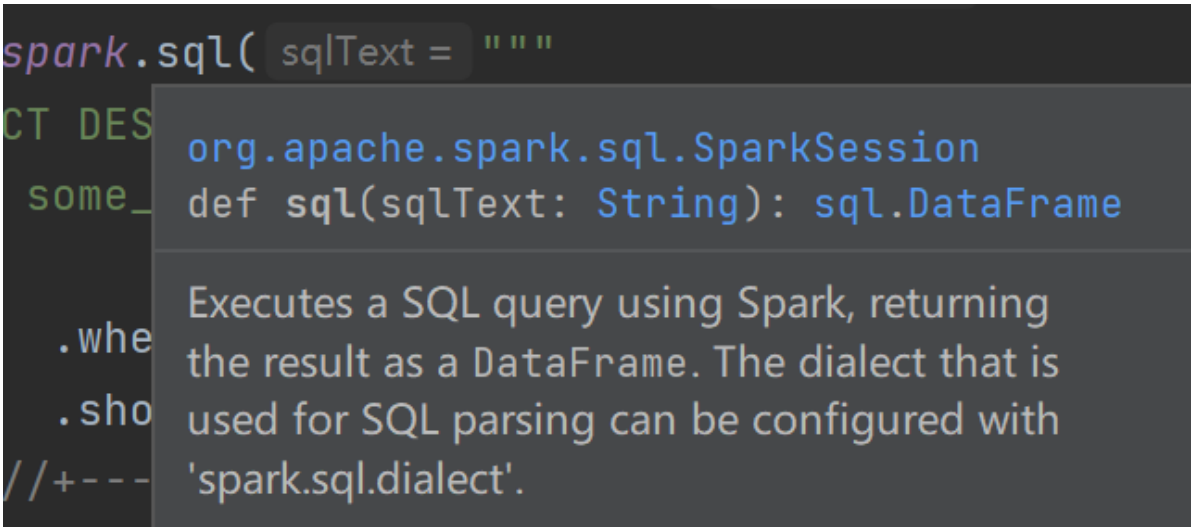
Spark's Programmatic SQL Interface

You can do this via the method sql on the **SparkSession object.** This returns a DataFrame.

```
spark.sql("SELECT 1 + 1").show()
//+-------+
//|(1 + 1)|
//+-------+
//|      2|
//+-------+
```

Even more powerful, you can completely interoperate between SQL and DataFrames.you can create a DataFrame, manipulate it with SQL, and then manipulate it again as a DataFrame.

```
spark.read.json("src/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")
      .show() // SQL => DF
    //+--------------------+----------+
    //|   DEST_COUNTRY_NAME|sum(count)|
    //+--------------------+----------+
    //|             Senegal|        40|
    //|              Sweden|       118|
    //|               Spain|       420|
    //|    Saint Barthelemy|        39|
    //|Saint Kitts and N...|       139|
    //|         South Korea|      1048|
    //|        Sint Maarten|       325|
    //|        Saudi Arabia|        83|
    //|         Switzerland|       294|
    //|         Saint Lucia|       123|
    //|               Samoa|        25|
    //|        South Africa|        36|
    //+--------------------+----------+
```



SparkSQL Thrift JDBC/ODBC Server

Spark provides a Java Database Connectivity (JDBC) interface by which either you or a remote program connects to the Spark driver in order to execute Spark SQL queries.

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**.

The catalog is available in the **org.apache.spark.sql.catalog.Catalog package**

contains a number of helpful functions for doing things like **listing tables, databases, and functions.**

if you're using the programmatic interface, keep in mind that **you need to wrap everything in a spark.sql function call to execute the relevant code.**

Tables

Tables are **logically equivalent to a DataFrame** in that they are a structure of data against which you run commands.

We can **join tables, filter them, aggregate them, and perform different manipulations** that we saw in previous chapters.

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.This means that when you create a table (assuming you never changed the database), it will belong to the **default database.**

> An important thing to note is that in Spark 2.X, **tables always contain data**. There is no notion of a temporary table, only a view, which does not contain data. This is important because if you go to drop a table, you can risk losing the data when doing so.

Spark-Managed Tables

managed versus **unmanaged tables.**

Tables store two important pieces of information:

- data within the tables
- data about the tables (metadata)

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.

In the explain plan, you will also notice that this writes to the **default Hive warehouse location.** You can set this by setting the **spark.sql.warehouse.dir configuration** to the directory of your choosing when you **create your SparkSession**. By default Spark sets this to /user/hive/warehouse:

you can also see tables in a specific database by using the query **show tables IN databaseName**, where databaseName represents the name of the database that you want to query

Creating Tables

You can create tables from a variety of sources. Something fairly unique to Spark is the capability of **reusing the entire Data Source API within SQL.**

```
def createTable(): Unit ={
  val sql = """CREATE TABLE flights (
              |DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count LONG)
              |USING JSON OPTIONS (path 'src/data/flight-data/json/2015-
summary.json')
              |""".stripMargin
  spark.sql(sql) //store in disk, second execute will throws table exist
Exception
  spark.sql("""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 'src/data/flight-
data/csv/2015-summary.csv')
              |""".stripMargin)
  spark.sql("""CREATE TABLE flights_from_select USING parquet AS SELECT * FROM
flights""")
  spark.sql("""CREATE TABLE partitioned_flights USING parquet PARTITIONED BY
(DEST_COUNTRY_NAME)
              |AS SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count FROM
flights LIMIT 5""".stripMargin)

  spark.sql(
    """
      |SELECT * FROM flights
      |""".stripMargin).show(5,false)
  spark.sql(
    """
      |SELECT * FROM flights_csv
      |""".stripMargin).show(5,false)
  spark.sql(
    """
      |SELECT * FROM flights_from_select
      |""".stripMargin).show(5,false)

  spark.sql(
    """
      |SELECT * FROM partitioned_flights
      |""".stripMargin).show(6,false)
```

```
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|United States    |Romania            |15   |
  //|United States    |Croatia            |1    |
  //|United States    |Ireland            |344  |
  //|Egypt            |United States      |15   |
  //|United States    |India              |62   |
  //+-----------------+-------------------+-----+
}
```

Creating External Tables

As we mentioned in the beginning of this chapter, Hive was one of the first big data SQL systems, and Spark SQL is completely compatible with Hive SQL (HiveQL) statements. One of the use cases that you might encounter is to port your legacy Hive statements to Spark SQL. Luckily, you can, for the most part, just copy and paste your Hive statements directly into Spark SQL. For example, in the example that follows, we create an unmanaged table. **Spark will manage the table's metadata; however, the files are not managed by Spark at all**. You create this table by using the CREATE EXTERNAL TABLE statement.

```
def createExternalTable(): Unit ={
  spark.sql("""drop table hive_flights""")
  spark.sql("""drop table hive_flights_2""")
  spark.sql("""CREATE EXTERNAL TABLE hive_flights (
              |      DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count
LONG)
              |    ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION
'src/data/flight-data-hive'""".stripMargin)
  spark.sql("""CREATE EXTERNAL TABLE hive_flights_2
              |ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
              |LOCATION 'src/data/flight-data-hive' AS SELECT * FROM flights
              |""".stripMargin)
  /**
   * 执行以下查询会跳转到创建外部表的table路径，一定要确保文件的权限
   */
  spark.sql("Select * From hive_flights").show(5,false)
  spark.sql("Select * From hive_flights_2").show(5,false)
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|United States    |Romania            |15   |
  //|United States    |Croatia            |1    |
  //|United States    |Ireland            |344  |
  //|Egypt            |United States      |15   |
  //|United States    |India              |62   |
  //+-----------------+-------------------+-----+
}
```

Inserting into Tables

```
def insert(): Unit ={
  spark.sql("""CREATE TABLE IF NOT EXISTS flights_empty (
              |DEST_COUNTRY_NAME STRING,
              |ORIGIN_COUNTRY_NAME STRING,
              |count LONG)
```

```
              |""".stripMargin)

  spark.sql(
    """
      |INSERT INTO flights_empty
      |SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count FROM flights ORDER
BY count desc LIMIT 20
      |""".stripMargin)
  spark.sql(
    """
      |select * from flights_empty
      |""".stripMargin).show(5, false)
  //+-----------------+-------------------+------+
  //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count |
  //+-----------------+-------------------+------+
  //|United States    |United States      |370002|
  //|United States    |Canada             |8483  |
  //|Canada           |United States      |8399  |
  //|United States    |Mexico             |7187  |
  //|Mexico           |United States      |7140  |
  //+-----------------+-------------------+------+
}
```

You can optionally provide a partition specification if you want to write only into a certain partition. Note that a write will respect a partitioning scheme, as well (which may cause the above query to run quite slowly); however, it will add additional files only into the end partitions:

```
INSERT INTO partitioned_flights
  PARTITION (DEST_COUNTRY_NAME="UNITED STATES")
  SELECT count, ORIGIN_COUNTRY_NAME FROM flights
  WHERE DEST_COUNTRY_NAME='UNITED STATES' LIMIT 12
```

Describing Table Metadata

```
def describeMetadata(): Unit ={
  spark.sql("""DESCRIBE TABLE flights_csv
              |""".stripMargin).show(false)
  //+------------------+---------+-------------------------------------+
  //|col_name          |data_type|comment                              |
  //+------------------+---------+-------------------------------------+
  //|DEST_COUNTRY_NAME |string   |null                                 |
  //|ORIGIN_COUNTRY_NAME|string   |remember, the US will be most prevalent|
  //|count             |bigint   |null                                 |
  //+------------------+---------+-------------------------------------+
  spark.sql("""SHOW PARTITIONS partitioned_flights
              |""".stripMargin).show(false)
  //+-------------------------------+
  //|partition                      |
  //+-------------------------------+
  //|DEST_COUNTRY_NAME=Egypt        |
  //|DEST_COUNTRY_NAME=United%20States|
  //+-------------------------------+
}
```

Refreshing Table Metadata

REFRESH TABLE refreshes all **cached entries** (essentially, files) associated with the table. If the table were previously cached, it would be cached lazily the next time it is scanned:

```
REFRESH table partitioned_flights
```

Another related command is **REPAIR TABLE**, which refreshes the partitions maintained **in the catalog** for that given table.

```
MSCK REPAIR TABLE partitioned_flights
```

Dropping Tables

```
DROP TABLE flights_csv; #Dropping a table deletes the data in the table, so you need to be very careful when doing this
```

```
DROP TABLE IF EXISTS flights_csv #This deletes the data in the table, so exercise caution when doing this,  also the table itself, not only data. If you want to delete the data only, use truncate, but truncate doen't work in external table
```

```
def deleteTable(): Unit ={
  spark.sql("""drop table hive_flights""")
  spark.sql("""drop table if exists hive_flights_2""")
  try{
    /**
     * org.apache.spark.sql.AnalysisException: Table or view not found: hive_flights; line 1 pos 14;
    'Project [*]
    +- 'UnresolvedRelation [hive_flights]
     */
    //spark.sql("Select * From hive_flights").show(5,false)

    /**
     *org.apache.spark.sql.AnalysisException: Table or view not found: hive_flights_2; line 1 pos 14;
      'Project [*]
      +- 'UnresolvedRelation [hive_flights_2]
     */
    spark.sql("Select * From hive_flights_2").show(5,false)
  }catch{
    case e: Throwable => println(e)
  }
}
```

> If you are dropping an unmanaged table (e.g., hive_flights), no data will be removed but you will no longer be able to refer to this data by the table name.

Caching Tables

Just like DataFrames, you can cache and uncache tables.

```
CACHE TABLE flights
```

```
UNCACHE TABLE FLIGHTS
```

Cache 的产生其实由spark 的lazy evalution引起的，在Spark中有时候我们很多地方都会**用到同一个RDD**, 按照常规的做法的话,那么**每个地方遇到Action操作的时候都会对同一个算子计算多次。**

Views

A view specifies a set of transformations on top of an existing table—basically just saved query plans, which can be convenient for organizing or reusing your query logic.

Spark has several different notions of views. Views can be **global**, **set to a database**, or **per session.**

Creating Views

Effectively, **views are equivalent to creating a new DataFrame from an existing DataFrame**.

```
def createView(): Unit ={
  spark.sql("""
              |CREATE VIEW just_usa_view AS
              |SELECT * FROM flights WHERE dest_country_name = 'United States'
              |""".stripMargin)
  /**
   * Like tables, you can create temporary views
   * that are available only during the current session and are not registered
to a database
   */
  spark.sql(
    """
      |CREATE TEMP VIEW just_usa_view_temp AS
      |SELECT * FROM flights WHERE dest_country_name = 'United States'
      |""".stripMargin)

  /**
   * Global temp views are resolved regardless of database and are viewable
across the entire Spark application,
   * but they are removed at the end of the session
   */
  spark.sql(
    """
      |CREATE GLOBAL TEMP VIEW just_usa_global_view_temp AS
      |SELECT * FROM flights WHERE dest_country_name = 'United States'
      |""".stripMargin)
  spark.sql("SHOW TABLES").show()
  //+--------+-------------------+-----------+
  //|database|          tableName|isTemporary|
  //+--------+-------------------+-----------+
  //| default|      bucketedfiles|      false|
  //| default|            flights|      false|
  //| default|        flights_csv|      false|
```

```
  //| default|      flights_empty|      false|
  //| default|flights_from_select|      false|
  //| default|       hive_flights|      false|
  //| default|      just_usa_view|      false|
  //| default|partitioned_flights|      false|
  //|        | just_usa_view_temp|       true|
  //+--------+-------------------+-----------+
  /**
   * overwrite a view
   */
  spark.sql(
    """
      |CREATE OR REPLACE TEMP VIEW just_usa_view_temp AS
      |SELECT * FROM flights WHERE dest_country_name = 'United States'
      |""".stripMargin)
  spark.sql(
    """
      |SELECT * FROM just_usa_view_temp
      |""".stripMargin).show()
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME| ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|    United States|            Romania|   15|
  //|    United States|             Russia|  161|
  //|       ..................................... |
  //|    United States|          Costa Rica|  608|
  //+-----------------+-------------------+-----+
  spark.sql(
    """
      |SELECT * FROM global_temp.just_usa_global_view_temp
      |""".stripMargin).show()
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME| ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|    United States|            Romania|   15|
  //|    United States|             Russia|  161|
  //|       ..................................... |
  //|    United States|          Costa Rica|  608|
  //+-----------------+-------------------+-----+
  val flights = spark.read.format("json")
    .load("src/data/flight-data/json/2015-summary.json")
  val just_usa_df = flights.where("dest_country_name = 'United States'")
  just_usa_df.selectExpr("*").show(3,false)
  //+-----------------+-------------------+-----+
  //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
  //+-----------------+-------------------+-----+
  //|United States    |Romania            |15   |
  //|United States    |Croatia            |1    |
  //|United States    |Ireland            |344  |
  //+-----------------+-------------------+-----+
}
```

Global Temporary View

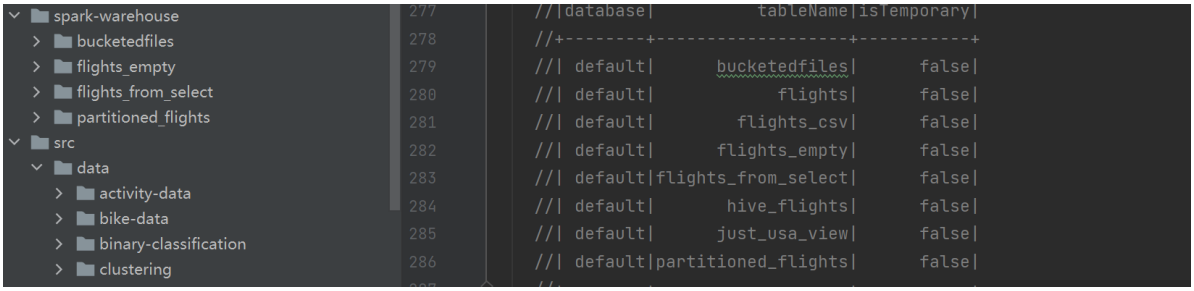
Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view. Global temporary view is tied to a system preserved database `global_temp`, and we must use the qualified name to refer it, e.g. `SELECT * FROM global_temp.view1`.

Scala | Java | Python | SQL

```scala
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Drop View

The main difference between dropping a view and dropping a table is that with a view, **no underlying data is removed,** only the **view definition itself**:（类似外部表，可以类比为一个引用）




```scala
def dropView(): Unit ={
  spark.sql("SHOW TABLES").show()
  //+--------+------------------+-----------+
  //|database|         tableName|isTemporary|
  //+--------+------------------+-----------+
  //| default|      bucketedfiles|      false|
  //| default|           flights|      false|
  //| default|       flights_csv|      false|
  //| default|     flights_empty|      false|
  //| default|flights_from_select|      false|
  //| default|      hive_flights|      false|
  //| default|     just_usa_view|      false|
  //| default|partitioned_flights|      false|
  //+--------+------------------+-----------+
  spark.sql("""DROP VIEW IF EXISTS just_usa_view""")
  spark.sql("SHOW TABLES").show()
  //+--------+------------------+-----------+
```

```
  //|database|          tableName|isTemporary|
  //+--------+-------------------+-----------+
  //| default|      bucketedfiles|      false|
  //| default|            flights|      false|
  //| default|        flights_csv|      false|
  //| default|      flights_empty|      false|
  //| default|flights_from_select|      false|
  //| default|       hive_flights|      false|
  //| default|partitioned_flights|      false|
  //+--------+-------------------+-----------+
}
```

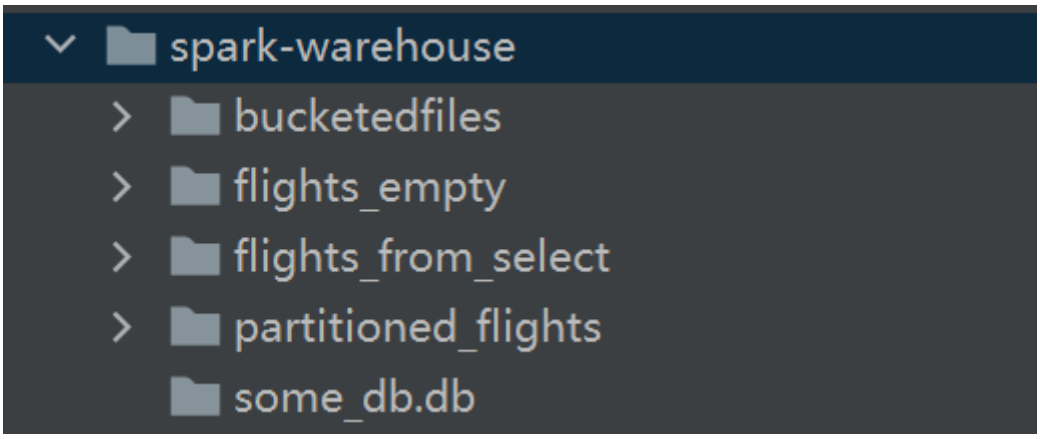
Databases

Databases are a tool for organizing tables. As mentioned earlier, if you do not define one, Spark will use the default database. Any SQL statements that you run from within Spark (including DataFrame commands) execute **within the context of a database.** This means that if you change the database, any user-defined tables will remain in the previous database and will need to be queried differently.（以上面global_temp为例，需要指定查询的数据库才能搜索到just_usa_global_view_temp）

> 为什么建立一个global_temp view 却无法查询到数据库global_temp 或者global temp view ?
>
> 为什么spark的global_temp数据库不可见？ - 问答 - 云+社区 - 腾讯云 (tencent.com)

```
  def databaseHandle(): Unit ={
    spark.sql("DROP DATABASE IF EXISTS some_db")
    spark.sql("CREATE DATABASE some_db")
    spark.sql("""SHOW DATABASES""").show()
    //+---------+
    //|namespace|
    //+---------+
    //|  default|
    //|  some_db|
    //+---------+
    spark.sql("USE some_db")
    spark.sql("SHOW TABLES").show()
    //+--------+---------+-----------+
    //|database|tableName|isTemporary|
    //+--------+---------+-----------+
    //+--------+---------+-----------+
//    spark.sql("SELECT * FROM flights") //fail
    spark.sql("SELECT * FROM default.flights").show()
    //+-----------------+-------------------+-----+
    //|DEST_COUNTRY_NAME| ORIGIN_COUNTRY_NAME|count|
    //+-----------------+-------------------+-----+
    //|    United States|             Romania|   15|
    //|    United States|              Russia|  161|
    //|       ..................................... |
    //|    United States|          Costa Rica|  608|
    //+-----------------+-------------------+-----+
    spark.sql("SELECT current_database()").show()
    //+-----------------+
    //|current_database()|
    //+-----------------+
    //|           some_db|
    //+-----------------+
```

```
  }
```




```
def showGlobalTemp(): Unit ={
  spark.sql(
    """
      |CREATE GLOBAL TEMP VIEW just_usa_global_view_temp AS
      |SELECT * FROM flights WHERE dest_country_name = 'United States'
      |""".stripMargin)
  spark.catalog.listTables("global_temp").show
  //+--------------------+-----------+-----------+---------+-----------+
  //|                name|   database|description|tableType|isTemporary|
  //+--------------------+-----------+-----------+---------+-----------+
  //|just_usa_global_v...|global_temp|       null|TEMPORARY|       true|
  //+--------------------+-----------+-----------+---------+-----------+
}
```

Select Statements

Queries in Spark **support the following ANSI SQL requirements** (here we list the layout of the SELECT expression)

```
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

This is essentially the equivalent of programmatic **if** statements

```
def selectStatement(): Unit ={
  spark.sql("""DROP TABLE IF EXISTS partitioned_flights""".stripMargin)
  spark.sql("""CREATE TABLE partitioned_flights USING parquet PARTITIONED BY
(DEST_COUNTRY_NAME)
              |AS SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count FROM
flights LIMIT 5""".stripMargin)
  spark.sql("""SELECT
              |CASE WHEN DEST_COUNTRY_NAME = 'UNITED STATES' THEN 1
              |WHEN DEST_COUNTRY_NAME = 'Egypt' THEN 0
              |ELSE -1 END AS case_when
              |FROM partitioned_flights""".stripMargin).show(5,false)
  //+---------+
  //|case_when|
  //+---------+
  //|0        |
  //|-1       |
  //|-1       |
  //|-1       |
  //|-1       |
  //+---------+
}
```

Advanced Topics

SQL statements can define **manipulations, definitions, or controls**. The most common case are the **manipulations.**

Complex Types

Structs

```
spark.sql("""
            |CREATE VIEW IF NOT EXISTS nested_data AS
            |SELECT (DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME)
            |as country, count FROM flights""".stripMargin)
spark.sql("SELECT * FROM nested_data").show(3,false)
//+-----------------------+-----+
//|country                |count|
//+-----------------------+-----+
//|[United States, Romania]|15   |
//|[United States, Croatia]|1    |
//|[United States, Ireland]|344  |
//+-----------------------+-----+
spark.sql("SELECT country.DEST_COUNTRY_NAME, count FROM
nested_data").show(3,false)
//+-----------------+-----+
//|DEST_COUNTRY_NAME|count|
//+-----------------+-----+
//|United States    |15   |
//|United States    |1    |
//|United States    |344  |
//+-----------------+-----+
spark.sql("SELECT country.*, count FROM nested_data").show(3,false)
```

```
//+-----------------+-------------------+-----+
//|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
//+-----------------+-------------------+-----+
//|United States    |Romania            |15   |
//|United States    |Croatia            |1    |
//|United States    |Ireland            |344  |
//+-----------------+-------------------+-----+
```

Lists

You can use the **collect_list** function, **which creates a list of values**. You can also use the function **collect_set**, which **creates an array without duplicate values.**

```
spark.sql("""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""".stripMargin).show(2,false)
//+--------+------------+--------------+
//|new_name|flight_counts|origin_set     |
//+--------+------------+--------------+
//|Anguilla|[41]         |[United States]|
//|Paraguay|[60]         |[United States]|
//+--------+------------+--------------+
spark.sql("SELECT DEST_COUNTRY_NAME, ARRAY(1, 2, 3) FROM flights").show(2,false)
//+-----------------+-------------+
//|DEST_COUNTRY_NAME|array(1, 2, 3)|
//+-----------------+-------------+
//|United States    |[1, 2, 3]     |
//|United States    |[1, 2, 3]     |
//+-----------------+-------------+
spark.sql("""SELECT DEST_COUNTRY_NAME as new_name, collect_list(count)[0]
            |FROM flights GROUP BY DEST_COUNTRY_NAME
            |""".stripMargin).show(2,false)
//+--------+----------------------+
//|new_name|collect_list(count)[0]|
//+--------+----------------------+
//|Anguilla|41                    |
//|Paraguay|60                    |
//+--------+----------------------+
/**
 * You can also do things like convert an array back into rows.
 * You do this by using the explode function.
 * To demonstrate, let's create a new view as our aggregation:
 */
spark.sql("""CREATE OR REPLACE TEMP VIEW flights_agg AS
            |SELECT DEST_COUNTRY_NAME, collect_list(count) as collected_counts
            |FROM flights GROUP BY DEST_COUNTRY_NAME""".stripMargin)
spark.sql("SELECT * FROM flights_agg").show(2,false)
//+-----------------+----------------+
//|DEST_COUNTRY_NAME|collected_counts|
//+-----------------+----------------+
//|Anguilla         |[41]            |
//|Paraguay         |[60]            |
//+-----------------+----------------+
spark.sql("SELECT explode(collected_counts), DEST_COUNTRY_NAME FROM
flights_agg").show(2,false)
```

```
//+---+-----------------+
//|col|DEST_COUNTRY_NAME|
//+---+-----------------+
//|41 |Anguilla         |
//|60 |Paraguay         |
//+---+-----------------+
```

Functions

```
def functionHandle(): Unit ={
  spark.sql("SHOW FUNCTIONS").show()
  //+--------+
  //|function|
  //+--------+
  //|       !|
  //|      !=|
  //|       %|
  //+--------+
  spark.sql("SHOW SYSTEM FUNCTIONS").show(2)
  //+--------+
  //|function|
  //+--------+
  //|       !|
  //|      !=|
  //+--------+
  /**
   *
  spark.udf.register("power3", power3(_:Double):Double)
  udfExampleDF.selectExpr("power3(num)").show(2)
  udfExampleDF.createOrReplaceTempView("udf_table")
  //持久化
  val sqlS = "CREATE OR REPLACE FUNCTION pow3 AS 'MyUDF' USING JAR
'src/data/udf-1.0-SNAPSHOT.jar'"
  spark.sql(sqlS)
  val exeS = "SELECT pow3(num) AS function_return_value FROM udf_table"
  spark.sql(exeS).show()
   * You can also register functions through the Hive CREATE TEMPORARY FUNCTION
syntax.
   */
  spark.sql("SHOW USER FUNCTIONS").show(2)
  //+------------+
  //|    function|
  //+------------+
  //|default.pow3|
  //+------------+
  spark.sql("SHOW FUNCTIONS  's*'").show(2)
  //+--------------+
  //|      function|
  //+--------------+
  //| schema_of_csv|
  //|schema_of_json|
  //+--------------+
  spark.sql("SHOW FUNCTIONS LIKE 'collect*'").show(2)
  //+------------+
  //|    function|
  //+------------+
  //|collect_list|
```

```
  //| collect_set|
  //+------------+
}
```

Subqueries

With subqueries, you can specify **queries within other queries**. This makes it possible for you to **specify some sophisticated logic within your SQL.**

In Spark, there are two fundamental subqueries.

- **Correlated subqueries** use some information from the outer scope of the query in order to supplement information in the subquery.
- **Uncorrelated subqueries** include no information from the outer scope.

Spark also includes support for **predicate subqueries,** which **allow for filtering based on values.**

```
spark.sql("""SELECT dest_country_name FROM flights
            |GROUP BY dest_country_name
            |ORDER BY sum(count) DESC
            |LIMIT 5""".stripMargin).show()
//+-----------------+
//|dest_country_name|
//+-----------------+
//|    United States|
//|           Canada|
//|           Mexico|
//|   United Kingdom|
//|            Japan|
//+-----------------+
```

Uncorrelated predicate subqueries

```
/**
 * This query is uncorrelated because it does not include any information from
the outer scope of the query.
 * It's a query that you can run on its own.
 */
spark.sql("""SELECT * FROM flights
            |WHERE origin_country_name IN (SELECT dest_country_name FROM flights
            |GROUP BY dest_country_name
            |ORDER BY sum(count) DESC
            |LIMIT 5)""".stripMargin).show(5)
//+-----------------+-------------------+-----+
//|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
//+-----------------+-------------------+-----+
//|            Egypt|      United States|   15|
//|       Costa Rica|      United States|  588|
//|          Senegal|      United States|   40|
//|          Moldova|      United States|    1|
//|           Guyana|      United States|   64|
//+-----------------+-------------------+-----+
```

Correlated predicate subqueries

```
    /**
EXISTS just checks for some existence in the subquery and returns true if there
is a value. You
can flip this by placing the NOT operator in front of it. This would be
equivalent to finding a flight
to a destination from which you won't be able to return!
     */
    spark.sql("""SELECT * FROM flights f1
                |WHERE NOT EXISTS (SELECT 1 FROM flights f2
                |WHERE f1.dest_country_name = f2.origin_country_name)
                |AND EXISTS (SELECT 1 FROM flights f2
                |WHERE f2.dest_country_name = f1.origin_country_name)"""
      .stripMargin).show(5)
    //+-------------------+-------------------+-----+
    //|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
    //+-------------------+-------------------+-----+
    //|             Moldova|      United States|    1|
    //|             Algeria|      United States|    4|
    //|Saint Vincent and...|      United States|    1|
    //|        Burkina Faso|      United States|    1|
    //|            Djibouti|      United States|    1|
    //+-------------------+-------------------+-----+
```

Uncorrelated scalar queries

```
/**
 * Using uncorrelated scalar queries, you can bring in some supplemental
information that you might not have previously.
 * For example, if you wanted to include the maximum value as its own column
from the entire counts dataset, you could do this:
 */
spark.sql("""SELECT *, (SELECT max(count) FROM flights) AS maximum FROM
flights""").show(2)
//+-----------------+-------------------+-----+-------+
//|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|maximum|
//+-----------------+-------------------+-----+-------+
//|    United States|            Romania|   15| 370002|
//|    United States|            Croatia|    1| 370002|
//+-----------------+-------------------+-----+-------+
```

Miscellaneous Features

Configuration

Property Name	Default	Meaning
`spark.sql.inMemoryColumnarStorage.compressed`	`true`	When set to true, Spark SQL automatically selects a compression codec for each column based on statistics of the data.
`spark.sql.inMemoryColumnarStorage.batchSize`	10000	Controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk `OutOfMemoryErrors` (OOMs) when caching data.
`spark.sql.files.maxPartitionBytes`	134217728 (128 MB)	The maximum number of bytes to pack into a single partition when reading files.
`spark.sql.files.openCostInBytes`	4194304 (4 MB)	The estimated cost to open a file, measured by the number of bytes that could be scanned in the same time. This is used when putting multiple files into a partition. It is better to overestimate; that way the partitions with small files will be faster than partitions with bigger files (which is scheduled first).
`spark.sql.broadcastTimeout`	300	Timeout in seconds for the broadcast wait time in broadcast joins.
`spark.sql.autoBroadcastJoinThreshold`	10485760 (10 MB)	Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. You can disable broadcasting by setting this value to -1. Note that currently statistics are supported only for Hive Metastore tables for which the command `ANALYZE TABLE COMPUTE STATISTICS noscan` has been run.
`spark.sql.shuffle.partitions`	200	Configures the number of partitions to use when shuffling data for joins or aggregations.

Setting Configuration Values in SQL

```
def configuration(): Unit ={
  spark.conf.set("spark.sql.shuffle.partitions",30)
  println(spark.conf.get("spark.sql.shuffle.partitions")) //30
  spark.sql("SET spark.sql.shuffle.partitions=20")
  println(spark.conf.get("spark.sql.shuffle.partitions")) //20
}
```

DataSet

Datasets are the **foundational type** of the Structured APIs.

Datasets are a strictly Java Virtual Machine (JVM) language feature that work only with Scala and Java. Using Datasets, you can define the object that each row in your Dataset will consist of. In Scala, this will be a case class object that essentially defines a schema that you can use, and in Java, you will define a Java Bean.

In fact, if you use Scala or Java, all “DataFrames” are actually Datasets of type Row. To efficiently support **domain-specific objects,** a special concept called an **“Encoder” is required**. The encoder maps the domain-specific type T to Spark’s internal type system.

Spark converts the Spark **Row forma**t to the object you **specified** (a case class or Java class). **This conversion slows down** your operations **but can provide more flexibility.**

When to use

- When the operation(s) you would like to perform cannot be expressed using DataFrame manipulations
- When you want or need type-safety, and you're willing to accept the cost of performance to achieve it
- When you would like to reuse a variety of transformations of entire rows between single-node workloads and Spark workloads.

> you might have a large set of business logic that you'd like to **encode in one specific function** instead of in SQL or DataFrames.
>
> the Dataset API is type-safe. Operations that are not valid for their types, say subtracting two string types, will **fail at compilation time not at runtime.**
>
> one advantage of using Datasets is that if you define all of your data and transformations as accepting case classes it is trivial to **reuse** them for both distributed and local workloads.
>
> when you collect your DataFrames to local disk, they will be of the correct class and type, sometimes making further manipulation easier.

> use DataFrames and Datasets in tandem, manually **trading off between performance and type safety** when it is most relevant for your workload.

Creating Datasets

In Java: Encoders

```
public class Flight implements Serializable {
    String DEST_COUNTRY_NAME;
    String ORIGIN_COUNTRY_NAME;
    Long count;
}
```

```
public class DatasetsInJAVA {
    public final static SparkSession spark = SparkSession
            .builder()
            .master("local[*]")
            .getOrCreate();
    public static void main(String[] args) {
        Dataset<Flight> flights = spark.read()
                .parquet("src/data/flight-data/parquet/2010-summary.parquet/")
                .as(Encoders.bean(Flight.class));
        flights.show(5,false);
        //+-----------------+-------------------+-----+
        //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
        //+-----------------+-------------------+-----+
        //|United States    |Romania            |1    |
        //|United States    |Ireland            |264  |
        //|United States    |India              |69   |
        //|Egypt            |United States      |24   |
        //|Equatorial Guinea|United States      |1    |
        //+-----------------+-------------------+-----+
    }
}
```

In Scala: Case Classes

```
object DateSetHandler {
  /**
   * case class can't put in method
   * @param DEST_COUNTRY_NAME
   * @param ORIGIN_COUNTRY_NAME
   * @param count
   */
  case class Flight(DEST_COUNTRY_NAME: String,
                    ORIGIN_COUNTRY_NAME: String,
                    count: BigInt)
  val spark = SparkSession.builder().master("local[*]").getOrCreate()
  var flightsDF: DataFrame = null

  def main( args:Array[String]): Unit ={
    flightsDF = loadData()
    createDatasets()
  }

  def createDatasets(): Unit ={
    import spark.implicits._
    val flights = flightsDF.as[Flight]
    flights.show(5,false)
  }

  def loadData(): DataFrame = {
    spark.read.parquet("src/data/flight-data/parquet/2010-summary.parquet/")
  }
}
```

Actions

```
flights.show(5,false)
//+-----------------+-------------------+-----+
//|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
//+-----------------+-------------------+-----+
//|United States    |Romania            |1    |
//|United States    |Ireland            |264  |
//|United States    |India              |69   |
//|Egypt            |United States      |24   |
//|Equatorial Guinea|United States      |1    |
//+-----------------+-------------------+-----+
println(flights.first.DEST_COUNTRY_NAME) //United States
```

Transformations

Transformations on Datasets are the same as those that we saw on DataFrames.

In addition to those transformations, Datasets allow us to specify more complex and strongly typed transformations than we could perform on DataFrames alone because we **manipulate raw Java Virtual Machine (JVM) types.**

Filtering

```
def transformations(): Unit ={
  /**
   * a simple example by creating a simple function that accepts a Flight and
returns a Boolean value
   * that describes whether the origin and destination are the same.
   */
  def originIsDestination(flight_row: Flight): Boolean = {
    flight_row.ORIGIN_COUNTRY_NAME == flight_row.DEST_COUNTRY_NAME
  }
  println(flights.filter(flight_row => originIsDestination(flight_row)).first())
//Flight(United States,United States,348113)
  /**
   * we can use it and test it on data on our local machines before using it
within Spark
   */
  println(flights.collect().filter(flight_row =>
originIsDestination(flight_row)).mkString("Array(", ", ", ")"))
//Array(Flight(United States,United States,348113))
}
```

Mapping

```
    /**
     * extract one value from each row
     */
import spark.implicits._
val destinations = flights.map(f => f.DEST_COUNTRY_NAME)
val localDestinations = destinations.take(5)
localDestinations.foreach(print)
//United StatesUnited StatesUnited StatesEgyptEquatorial Guinea
```

Joins

Apply just the same as they did for DataFrames. However Datasets also provide a more sophisticated method, the **joinWith** method. **joinWith** is roughly equal to a co-group (in RDD terminology) and you basically end up with **two nested Datasets inside of one**. Each column represents one Dataset and these can be manipulated accordingly. This can be useful when you need to maintain more information in the join or perform some more sophisticated manipulation on the entire result, like an advanced map or filter.

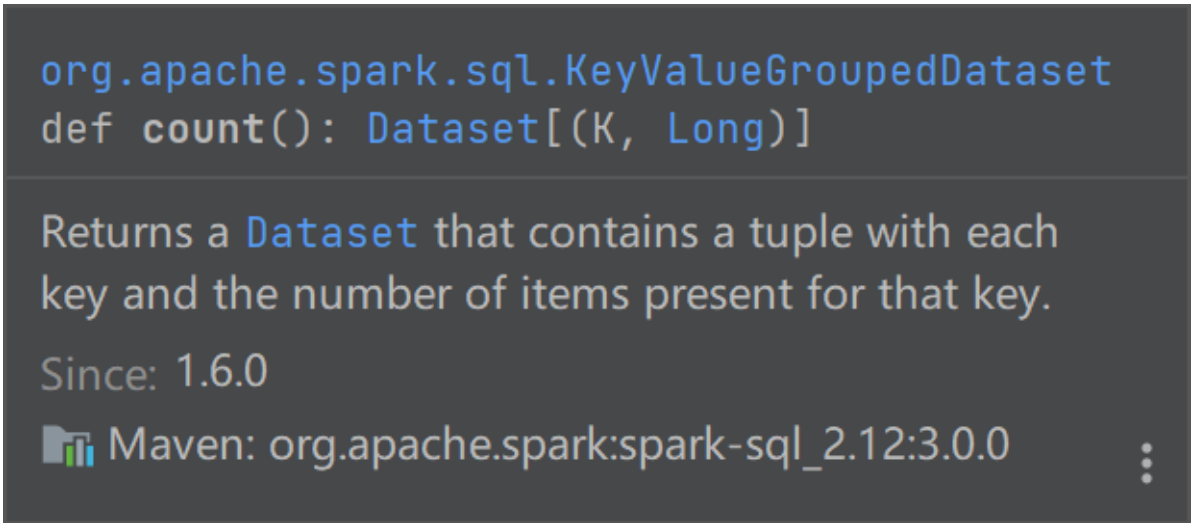
```
def joinHandle(): Unit ={
  import spark.implicits._
  val flightsMeta = spark.range(500).map(x => (x, scala.util.Random.nextLong))
    .withColumnRenamed("_1", "count").withColumnRenamed("_2", "randomData")
    .as[FlightMetadata]
  flightsMeta.show(2)
  //+-----+--------------------+
  //|count|          randomData|
  //+-----+--------------------+
  //|    0|-6150346972571577543|
  //|    1|-4669615282119679869|
  //+-----+--------------------+
  val flights2 = flights
```

```
    .joinWith(flightsMeta, flights.col("count") === flightsMeta.col("count"))
  flights2.show(5,false)
  //+---------------------------------+-----------------------+
  //|_1                               |_2                     |
  //+---------------------------------+-----------------------+
  //|[United States, Uganda, 1]       |[1, 3267990035442503191]|
  //|[United States, French Guiana, 1]|[1, 3267990035442503191]|
  //|[Bulgaria, United States, 1]     |[1, 3267990035442503191]|
  //|[United States, Slovakia, 1]     |[1, 3267990035442503191]|
  //|[United States, Cameroon, 1]     |[1, 3267990035442503191]|
  //+---------------------------------+-----------------------+
  flights2.selectExpr("_1.DEST_COUNTRY_NAME").show(2)
  //+-----------------+
  //|DEST_COUNTRY_NAME|
  //+-----------------+
  //|    United States|
  //|    United States|
  //+-----------------+
  /**
   * random,so the col _2.randomData is changed, use cache to avoid this
   */
  flights2.take(2).foreach(print)
  //(Flight(United States,Uganda,1),FlightMetadata(1,2736093704858406833))
  //(Flight(United States,French
Guiana,1),FlightMetadata(1,2736093704858406833))
  var flights3 = flights.join(flightsMeta, Seq("count"))
  flights3.show(2,false)
  //+-----+-----------------+-------------------+-------------------+
  //|count|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|randomData         |
  //+-----+-----------------+-------------------+-------------------+
  //|1    |United States    |Uganda             |6391289097340552651|
  //|1    |United States    |French Guiana      |6391289097340552651|
  //+-----+-----------------+-------------------+-------------------+
  /**
   * there are no problems joining a DataFrame and a Dataset
   */
  flights3 = flights.join(flightsMeta.toDF(), Seq("count"))
  flights3.show(2,false)
  //+-----+-----------------+-------------------+--------------------+
  //|count|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|randomData          |
  //+-----+-----------------+-------------------+--------------------+
  //|1    |United States    |Uganda             |-5963343169707404521|
  //|1    |United States    |French Guiana      |-5963343169707404521|
  //+-----+-----------------+-------------------+--------------------+
}
```

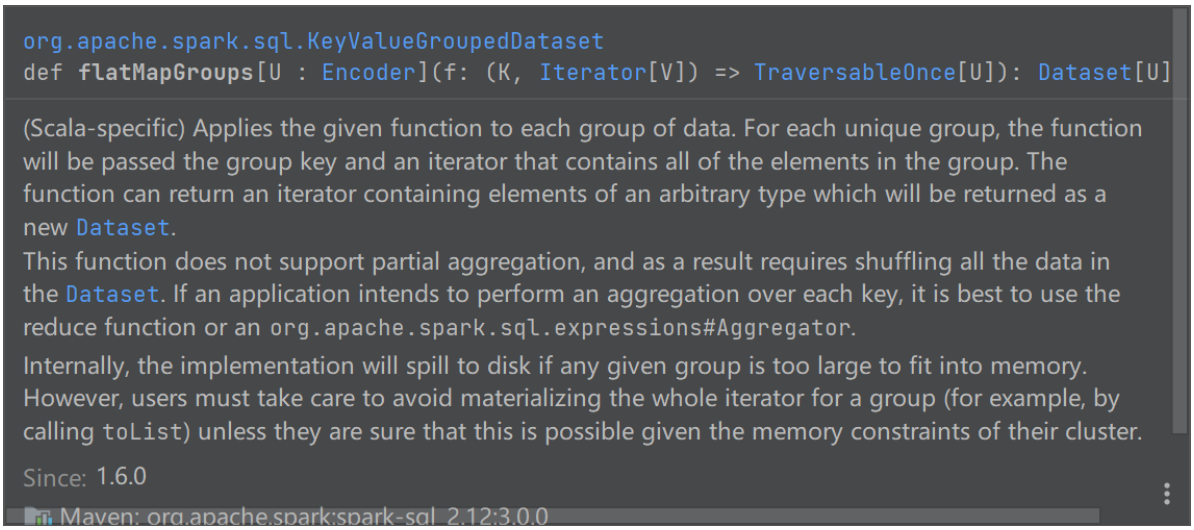
Grouping and Aggregations

groupBy rollup and cube still apply, but these return DataFrames instead of Datasets((you **lose type information**)

If you want to keep type information around there are other groupings and aggregations that you can perform. An excellent example is the **groupByKey** method. This allows you to group by a specific key in the Dataset and **get a typed Dataset in return**.



It should be straightfoward enough to understand that this is a **more expensive process** than aggregating immediately after scanning.




```
  def group_and_aggregation(): Unit ={
    import spark.implicits._
    flights.groupBy("DEST_COUNTRY_NAME").count().show()
    flights.groupBy("DEST_COUNTRY_NAME").count().explain()
    /**
 == Physical Plan ==
(2) HashAggregate(keys=[DEST_COUNTRY_NAME#0], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#0, 200), true, [id=#78]
   +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#0], functions=
[partial_count(1)])
      +- *(1) ColumnarToRow
         +- FileScan parquet [DEST_COUNTRY_NAME#0] Batched: true, DataFilters:
[], Format: Parquet, Location:
InMemoryFileIndex[file:/D:/Project/Spark_Project/src/data/flight-
data/parquet/2010-summary.parquet], PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<DEST_COUNTRY_NAME:string>

     */
    //+-------------------+-----+
    //|    DEST_COUNTRY_NAME|count|
```

```
    //+--------------------+-----+
    //|            Anguilla|    1|
    //|              Russia|    1|
    //|            Paraguay|    1|
    //|             Senegal|    1|
    //|              Sweden|    1|
    //+--------------------+-----+
    flights.groupByKey(x => x.ORIGIN_COUNTRY_NAME).count().explain()
    /**== Physical Plan ==
(3) HashAggregate(keys=[value#49], functions=[count(1)])
+- Exchange hashpartitioning(value#49, 200), true, [id=#92]
   +- *(2) HashAggregate(keys=[value#49], functions=[partial_count(1)])
      +- *(2) Project [value#49]
         +- AppendColumns scala.DateSetHandler$$$Lambda$2443/1969632323@5fafa76d,
newInstance(class scala.DateSetHandler$Flight), [staticinvoke(class
org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0,
java.lang.String, true], true, false) AS value#49]
            +- *(1) ColumnarToRow
               +- FileScan parquet
[DEST_COUNTRY_NAME#0,ORIGIN_COUNTRY_NAME#1,count#2L] Batched: t........
     */
    flights.groupByKey(x => x.ORIGIN_COUNTRY_NAME).count().show()
    //+--------------------+--------+
    //|                 key|count(1)|
    //+--------------------+--------+
    //|              Russia|       1|
    //|            Anguilla|       1|
    //|             Senegal|       1|
    //|              Sweden|       1|
    //+--------------------+--------+
    /**
     * After we perform a grouping with a key on a Dataset,
     * we can operate on the Key Value Dataset with functions that will
manipulate the groupings as raw objects:
     */
    def grpSum(countryName:String, values: Iterator[Flight]) = {
      values.dropWhile(_.count < 5).map(x => (countryName, x))
    }
    flights.groupByKey(x =>
x.DEST_COUNTRY_NAME).flatMapGroups(grpSum).show(5,false)
    //+--------+-----------------------------+
    //|_1      |_2                           |
    //+--------+-----------------------------+
    //|Anguilla|[Anguilla, United States, 21]|
    //|Paraguay|[Paraguay, United States, 90]|
    //|Russia  |[Russia, United States, 152] |
    //|Senegal |[Senegal, United States, 29] |
    //|Sweden  |[Sweden, United States, 65]  |
    //+--------+-----------------------------+
    def grpSum2(f:Flight):Integer = {
      1
    }
    flights.groupByKey(x =>
x.DEST_COUNTRY_NAME).mapValues(grpSum2).count().take(5).foreach(print)
    //(Anguilla,1)(Russia,1)(Paraguay,1)(Senegal,1)(Sweden,1)

    def sum2(left:Flight, right:Flight) = {
      Flight(left.DEST_COUNTRY_NAME, null, left.count + right.count)
```

```
  }
    flights.groupByKey(x => x.DEST_COUNTRY_NAME).reduceGroups((l, r) => sum2(l,
r))
      .map(x => (x._1, x._2.ORIGIN_COUNTRY_NAME, x._2.count))
      .withColumnRenamed("_1", "DEST_COUNTRY_NAME")
      .withColumnRenamed("_2", "ORIGIN_COUNTRY_NAME")
      .withColumnRenamed("_3", "count")
      .as[Flight]
      .orderBy(col("count").desc)
      .show(3)
    //+-----------------+-------------------+------+
    //|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME| count|
    //+-----------------+-------------------+------+
    //|    United States|               null|384932|
    //|           Canada|      United States|  8271|
    //|           Mexico|      United States|  6200|
    //+-----------------+-------------------+------+
      //.take(2).foreach(print)
    //(Anguilla,Flight(Anguilla,United States,21))(Russia,Flight(Russia,United
States,152)))
  }
```

Part lll. Low-Level APIs

Resilient Distributed Datasets (RDDs)

There are times when higher-level manipulation will not meet the business or engineering problem you are trying to solve. For those cases, you might need to use Spark's lower-level APIs, specifically **the Resilient Distributed Dataset (RDD)**, the **SparkContext**, and **distributed shared variables like accumulators and broadcast variables.**

What Are the Low-Level APIs?

There are two sets of low-level APIs: there is 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 that you cannot find in the higher-level APIs; for example, if you need very tight control over physical data placement across the cluster.
- You need to maintain some legacy codebase written using RDDs.
- You need to do some custom shared variable manipulation.

When you're calling a DataFrame transformation, it actually just becomes a set of RDD transformations.

How to Use the Low-Level APIs?

A **SparkContext** is the entry point for low-level API functionality. You access it through the **SparkSession**, which is the tool you use to perform computation across a Spark cluster

```
spark.sparkContext
```

About RDDs

> **In short, an RDD represents an immutable, partitioned collection of records that can be operated on in parallel**

in RDDs the records are just Java, Scala, or Python objects of the programmer's choosing

Every manipulation and interaction between values must be **defined by hand**, meaning that you must "reinvent the wheel" for whatever task you are trying to carry out.

The RDD API is similar to the Dataset, which we saw in the previous part of the book, except that **RDDs are not stored in, or manipulated with, the structured data engine.**

Types of RDDs

As a user, however, you will likely only be creating **two types of RDD**s: the **"generic" RDD type** or a **key-value RDD** that provides additional functions, such as aggregating by key.

Each RDD is characterized by **five main properties:**

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Optionally, a **Partitioner** for key-value RDDs (e.g., to say that the RDD is hash-partitioned)
- Optionally, a list of preferred locations on which to compute each split (e.g., block locations for a Hadoop Distributed File System [HDFS] file)

> The Partitioner is probably one of the core reasons why you might want to use RDDs in your code. **Specifying your own custom Partitioner can give you significant performance and stability improvements if you use it correctly.**

These properties determine all of Spark's ability to schedule and execute the user program. Different kinds of RDDs implement their own versions of each of the aforementioned properties, allowing you to define new data sources.

RDDs follow the exact same Spark programming paradigms that we saw in earlier chapters. They provide **transformations, which evaluate lazily, and actions**, which evaluate eagerly, to manipulate data in a distributed fashion

there is no concept of "rows" in RDDs; individual records are just raw Java/Scala/Python objects

When to Use RDDs?

For the vast majority of use cases, DataFrames will be more efficient, more stable, and more expressive than RDDs.

The most likely reason for why you'll want to use RDDs is because you need **fine-grained control over the physical distribution of data** (custom partitioning of data).

Datasets and RDDs of Case Classes

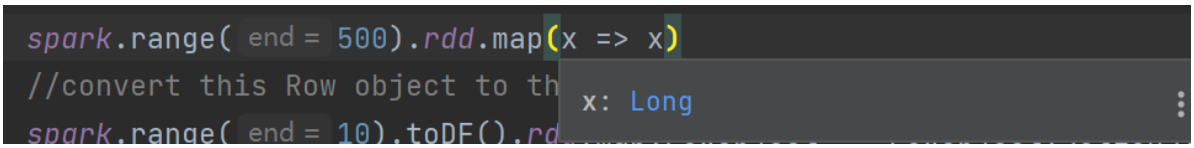
What is the difference between RDDs of Case Classes and Datasets?

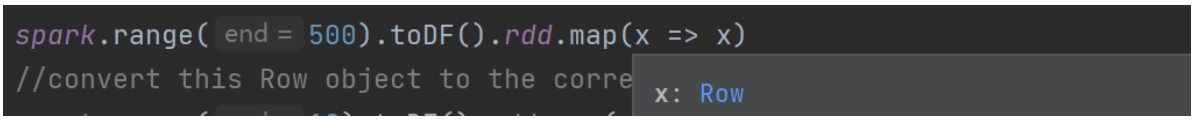
The difference is that **Datasets can still take advantage of the wealth of functions and optimizations that the Structured APIs** have to offer. With Datasets, you do not need to choose between only operating on JVM types or on Spark types, you can choose whatever is either easiest to do or most flexible. You get the both of best worlds.

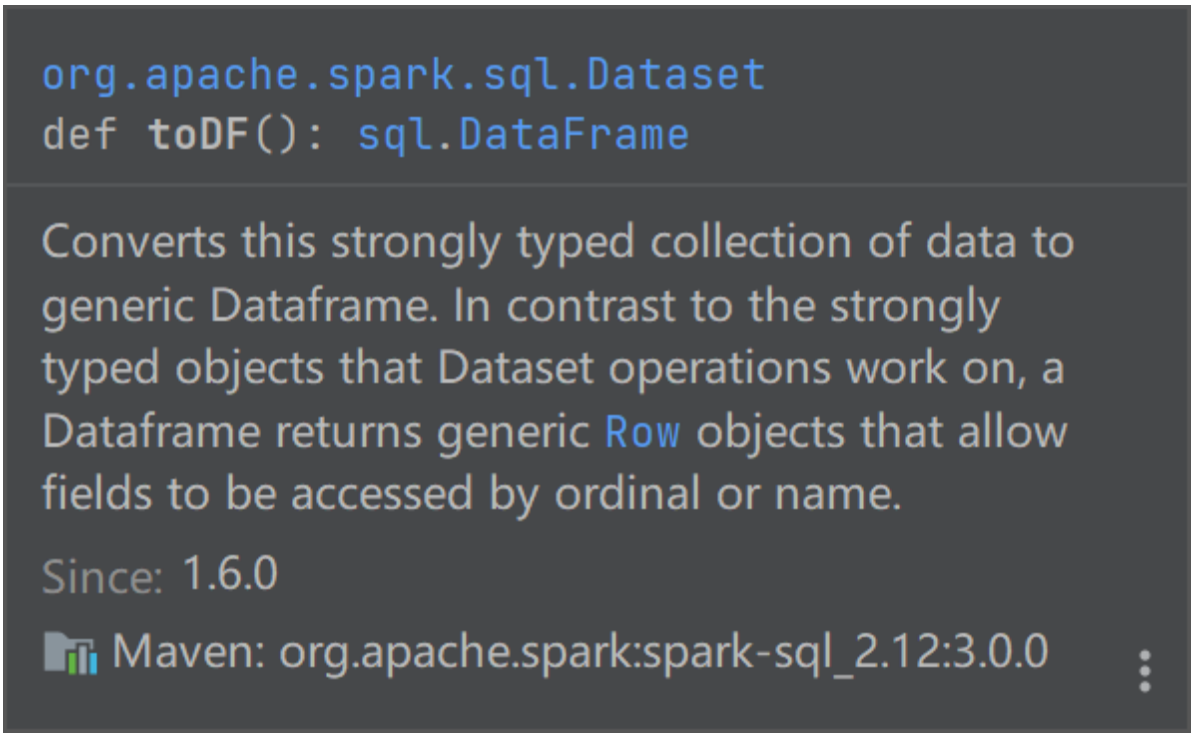
Creating RDDs

Interoperating Between DataFrames, Datasets, and RDDs

toDF()








```
def interoperate(): Unit ={
  import spark.implicits._
  //converts a Dataset[Long] to RDD[Long]
  spark.range(500).toDF().rdd.map(x => x)
  //convert this Row object to the correct data type or extract values out of it
  spark.range(10).toDF().rdd.map(rowObject => rowObject.getLong(0))
  //n use the same methodology to create a DataFrame or Dataset from an RDD
  spark.range(10).toDF().rdd.map(rowObject =>
rowObject.getLong(0)).toDF().show(3)
  //+-----+
  //|value|
  //+-----+
  //|    0|
  //|    1|
  //|    2|
  //+-----+
}
```

From a Local Collection

```
def createRddFromLocalCollection(): Unit ={
  val myCollection = "Spark The Definitive Guide : Big Data Processing Made
Simple"
    .split(" ")
  val words = spark.sparkContext.parallelize(myCollection, 2)
  words.toDF().show()
  //+----------+
  //|     value|
  //+----------+
  //|     Spark|
  //|       The|
  //|Definitive|
  //|     Guide|
  //|         :|
  //|       Big|
  //|      Data|
  //|Processing|
  //|      Made|
  //|    Simple|
  //+----------+
  //An additional feature is that you can then name this RDD to show up in the
Spark UI according to a given name
  words.setName("myWords")
  println(words.name)
}
```

From Data Sources

```
spark.sparkContext.textFile("/some/path/withTextFiles")
```

This creates an RDD for which each record in the RDD represents a line in that text file or files. Alternatively, you can read in data for which each text file should become a single record.

```
spark.sparkContext.wholeTextFiles("/some/path/withTextFiles")
```

Manipulating RDDs

Transformations

```
def transformations(): Unit ={
  val myCollection = "Spark The Definitive Guide : Big Data Processing Made
Simple"
    .split(" ")
  val words = spark.sparkContext.parallelize(myCollection, 2)
  /**
   * distinct
   */
  println(words.distinct().count()) //10

  /**
   * filter
   */
  def startsWithS(individual:String) = {
```

```
    individual.startsWith("S")
  }
  words.filter(word => startsWithS(word)).toDF().show(5)
  //+------+
  //| value|
  //+------+
  //| Spark|
  //|Simple|
  //+------+
  /**
   * map
   */
  val words2 = words.map(word => (word, word(0), word.startsWith("S")))

  /**
   * words2 can't apply words2.toDF() now
   Exception in thread "main" java.lang.UnsupportedOperationException: No
Encoder found for Char
  - field (class: "scala.Char", name: "_2")
  - root class: "scala.Tuple3"
   if you want to convert to DF
       words2.map(x => (x._1, x._2.toString, x._3)).toDF().show()
   */
  words2.filter(record => record._3).take(5).foreach(print)
  //(Spark,S,true)(Simple,S,true)
  /**
   * flatMap
   * flatMap requires that the ouput of the map function be an iterable that can
be expanded
   */
  words.flatMap(word => word.toSeq).take(5).foreach(print)
  //Spark
  words.flatMap(word => word.toSeq).map(x => x.toString).toDF().show(5)
  //+-----+
  //|value|
  //+-----+
  //|    S|
  //|    p|
  //|    a|
  //|    r|
  //|    k|
  //+-----+
  /**
   * sort
   */
  words.sortBy(word => word.length() * -1).take(2).foreach(print)
  //Definitive Processing
  /**
   * Random Splits
   * randomly split an RDD into an Array of RDDs by using the randomSplit method
   * returns an array of RDDs that you can manipulate individually
   */
  val fiftyFiftySplit = words.randomSplit(Array[Double](0.5, 0.5))
  fiftyFiftySplit.foreach(x => x.toDF().show())
  //+----------+
  //|     value|
  //+----------+
  //|       The|
```

```
  //|         :|
  //|      Data|
  //|Processing|
  //+----------+
  //+----------+
  //|     value|
  //+----------+
  //|     Spark|
  //|Definitive|
  //|     Guide|
  //|       Big|
  //|      Made|
  //|    Simple|
  //+----------+
}
```

Actions

```
  def actionsHandle(): Unit ={
    val myCollection = "Spark The Definitive Guide : Big Data Processing Made
Simple"
      .split(" ")
    val words = spark.sparkContext.parallelize(myCollection, 2)
    /**
     * reduce
     */
    println(spark.sparkContext.parallelize(1 to 20).reduce(_ + _)) //210
    //get the longest word
    /**
     * This reducer is a good example because you can get one of two outputs.
Because the reduce
operation on the partitions is not deterministic, you can have either
“definitive” or “processing”
(both of length 10) as the “left” word. This means that sometimes you can end up
with one,
whereas other times you end up with the other.
     * @param leftWord
     * @param rightWord
     * @return
     */
    def wordLengthReducer(leftWord:String, rightWord:String): String = {
      if (leftWord.length > rightWord.length)
        return leftWord
      else
        return rightWord
    }
    words.reduce(wordLengthReducer).foreach(print) //Processing

    /**
     * count
     */
    println(words.count()) //10
    /**
     * countApprox
     * confidence is the probability that the error bounds of the result will
contain the true value
```

```
     *  countApprox were called repeatedly with confidence 0.9, we would expect
90% of the results to contain the true count.
     */
    val confidence = 0.95
    val timeoutMilliseconds = 400
    println(words.countApprox(timeoutMilliseconds, confidence)) //(final:
[10.000, 10.000])
    /**
     * countApproxDistinct
     * There are two implementations of this, both based on streamlib's
implementation of
“HyperLogLog in Practice: Algorithmic Engineering of a State-of-the-Art
Cardinality Estimation
Algorithm.”
     *  In the first implementation, the argument we pass into the function is
the relative accuracy.
Smaller values create counters that require more space. The value must be greater
than 0.000017
     */
    println(words.countApproxDistinct(0.05)) //10
    /**
     * countApproxDistinct
     *  you specify the relative accuracy based on two parameters:
     *  one for “regular” data and another for a sparse representation.
The two arguments are p and sp where p is precision and sp is sparse precision.
The relative
accuracy is approximately 1.054 / sqrt(2 ). Setting a nonzero (sp > p) can reduce
the
memory consumption and increase accuracy when the cardinality is small. Both
values are
integers
     */
    println(words.countApproxDistinct(4, 10)) //10
    /**
     * countByValue
     * You should use this method only if the resulting map is expected to be
small
     * because the entire thing is loaded into the driver's memory
     */
    println(words.countByValue())
    //Map(Definitive -> 1, Simple -> 1, Processing -> 1, The -> 1, Spark -> 1,
Made -> 1, Guide -> 1, Big -> 1, : -> 1, Data -> 1)
    /**
     * countByValueApprox
     */
    println(words.countByValueApprox(1000, 0.95))
    //(final: Map(
    // Definitive -> [1.000, 1.000], Simple -> [1.000, 1.000],
    // Processing -> [1.000, 1.000], The -> [1.000, 1.000],
    // Spark -> [1.000, 1.000], Made -> [1.000, 1.000],
    // Guide -> [1.000, 1.000], Big -> [1.000, 1.000],
    // : -> [1.000, 1.000], Data -> [1.000, 1.000]))
    /**
     * first
     */
    println(words.first()) //Spark

    /**
```

```
     * max and min
     */
    println(spark.sparkContext.parallelize(1 to 20).max()) //20
    println(spark.sparkContext.parallelize(1 to 20).min()) // 1

    /**
     * take
     * This works by first scanning one partition and then using the results from
that partition
     * to estimate the number of additional partitions needed to satisfy the
limit
     */
    words.take(5).foreach(print) //Spark The Definitive Guide :
    println()
    words.takeOrdered(5).foreach(print) //: Big Data Definitive Guide
    println()
    words.top(5).foreach(print) //The Spark Simple Processing Made
    val withReplacement = true
    val numberToTake = 6
    val randomSeed = 100L
    words.takeSample(withReplacement, numberToTake, randomSeed).foreach(print)
    //Guide Spark : Simple Simple Spark
  }
```

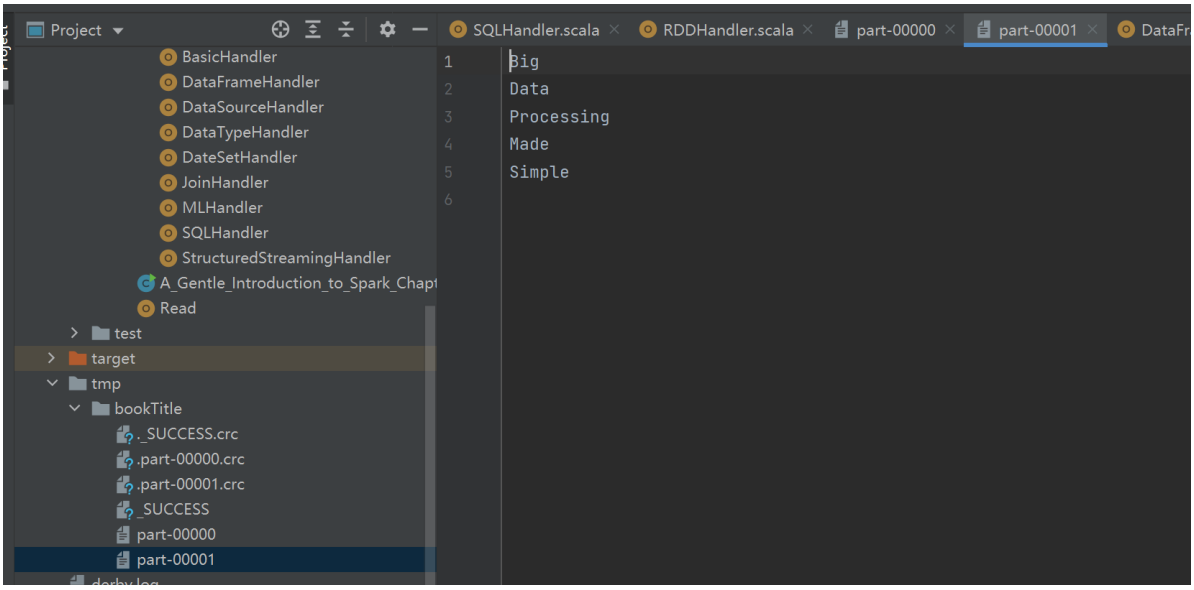
Saving Files

Saving files means writing to **plain-text files**. With RDDs, you cannot actually "save" to a data source in the conventional sense.

You must iterate over the partitions in order to save the contents of each partition to some external database. This is a low-level approach that reveals the underlying operation that is being performed in the higher-level APIs. **Spark will take each partition, and write that out to the destination.**

saveAsTextFile

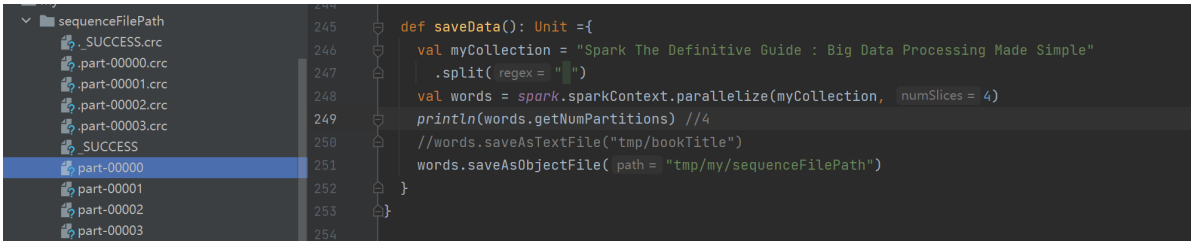
```
val myCollection = "Spark The Definitive Guide : Big Data Processing Made
Simple"
  .split(" ")
val words = spark.sparkContext.parallelize(myCollection, 4)
println(words.getNumPartitions) //4
words.saveAsTextFile("tmp/bookTitle")
```



SequenceFiles

A sequenceFile is a **flat file consisting of binary key–value pairs**. It is extensively **used in MapReduce as input/output formats.**

```
words.saveAsObjectFile("tmp/my/sequenceFilePath")
```



Hadoop Files

There are a variety of different Hadoop file formats to which you can save. These allow you to specify **classes, output formats, Hadoop configurations, and compression schemes.**

> For information on these formats, read **Hadoop: The Definitive Guide [O'Reilly, 2015].**

Caching

You can either **cache or persist an RDD.** By default, cache and persist only handle data in memory.

We can specify a **storage level** as any of the storage levels in the singleton object: **org.apache.spark.storage.StorageLevel**

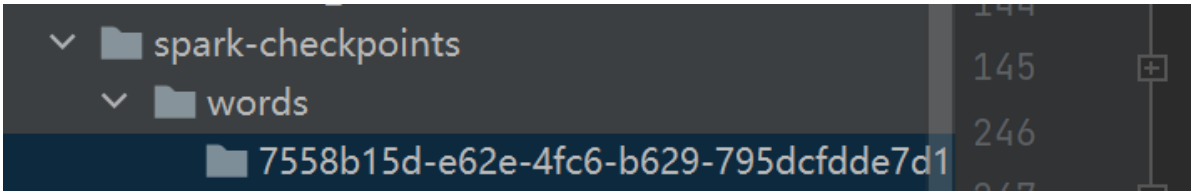
```
def cacheHandle(): Unit ={
  val myCollection = "Spark The Definitive Guide : Big Data Processing Made
Simple"
    .split(" ")
  val words = spark.sparkContext.parallelize(myCollection, 4)
  println(words.getStorageLevel) //StorageLevel(1 replicas)
  words.cache()
  println(words.getStorageLevel) //StorageLevel(memory, deserialized, 1
replicas)
  words.persist()
  println(words.getStorageLevel) //StorageLevel(memory, deserialized, 1
replicas)
}
```

Checkpointing

One feature **not available in the DataFrame API** is the concept of **checkpointing.**

Checkpointing is the **act of saving an RDD to disk** so that future references to this RDD point to those intermediate partitions on disk **rather than recomputing the RDD from its original source.**

```
spark.sparkContext.setCheckpointDir("spark-checkpoints/words")
words.checkpoint()
```



checkpoint会将结果写到hdfs上，当driver 关闭后数据不会被清除。所以可以在其他driver上重复利用该checkpoint的数据。

Contents

checkpoint write data:

```
sc.setCheckpointDir("data/checkpoint")
val rddt = sc.parallelize(Array((1,2),(3,4),(5,6)),2)
rddt.checkpoint()
rddt.count() //要action才能触发checkpoint
```

read from checkpoint data:

```
package org.apache.spark

import org.apache.spark.rdd.RDD

object RDDUtilsInSpark {
  def getCheckpointRDD[T](sc:SparkContext, path:String) = {
        //path要到part-000000的父目录
    val result : RDD[Any] = sc.checkpointFile(path)
    result.asInstanceOf[T]
  }
}
```

note:因为sc.checkpointFile(path)是private[spark]的，所以该类要写在自己工程里新建的package org.apache.spark中

example:

```
val rdd : RDD[(Int, Int)]= RDDUtilsInSpark.getCheckpointRDD(sc, "data/checkpoint/963afe46-eb23-430f-8eae-8a6c5a1e41ba/rdd-0")
  println(rdd.count())
  rdd.collect().foreach(println)
```

> ？网上说要先cache再checkpoint 避免两次写入，没遇到过，先记一下

Pipe RDDs to System Commands

With pipe, you can return an RDD created by piping elements to a forked external process.

The resulting RDD is computed by **executing the given process once per partition.**

All elements of each input partition are written to a process's stdin as lines of input separated by a newline. The resulting partition consists of the process's stdout output, with each line of stdout resulting in one element of the output partition. A process is invoked even for empty partitions.

We can use a simple example and pipe each partition to the command **wc**. Each row will be passed in as a new line, so if we perform a line count, we will get the number of lines, one per partition:

```
//本机没跑起来，需要linux环境
words.pipe("wc -l").collect()
```

mapPartitions

You also might have noticed earlier that the return signature of a map function on an RDD is actually MapPartitionsRDD. This is because **map is just a row-wise alias for mapPartitions**, which makes it possible for you to map an individual partition (represented as an iterator). That's because physically on the cluster we operate on each partition individually (and not a specific row).

Naturally, this means that we operate on a per-partition basis and allows us to perform an operation on that entire partition. This is valuable for performing something on an entire subdataset of your RDD. You can gather all values of a partition class or group into one partition and then operate on that entire group using arbitrary functions and controls.

```
    println(words.mapPartitions(part => Iterator[Int](1)).sum()) //4.0
    /**
     * Other functions similar to mapPartitions include mapPartitionsWithIndex.
With this you
specify a function that accepts an index (within the partition) and an iterator
that goes through all
items within the partition. The partition index is the partition number in your
RDD, which
identifies where each record in our dataset sits (and potentially allows you to
debug). You might
use this to test whether your map functions are behaving correctly:
     */
    def indexedFunc(partitionIndex:Int, withinPartIterator: Iterator[String]) =
{
      withinPartIterator.toList.map(
        value => s"Partition: $partitionIndex => $value").iterator
    }
    words.mapPartitionsWithIndex(indexedFunc).collect().foreach(print)
    //Partition: 0 => Spark Partition: 0 => The
    // Partition: 1 => Definitive Partition: 1 => Guide Partition: 1 => :
    // Partition: 2 => Big Partition: 2 => Data
    // Partition: 3 => ProcessingPartition: 3 => Made Partition: 3 => Simple
```

foreachPartition

Although mapPartitions needs a return value to work properly, this next function does not. foreachPartition simply **iterates over all the partitions of the data.**

glom

glom is an interesting function that **takes every partition** in your dataset and converts them to **arrays.**

This can be useful if you're going to **collect the data to the driver and want to have an array for each partition.**

> **However, this can cause serious stability issues because if you have large partitions or a large number of partitions, it's simple to crash the driver!**

```
spark.sparkContext.parallelize(Seq("Hello", "World"),
2).glom().collect().foreach(x => x.foreach(println))
// Hello
// World
/*Array(Array(Hello), Array(World))*/
```

Advanced RDD

This chapter covers the **advanced RDD** operations and focuses on **key–value RDDs,** a powerful abstraction for manipulating data.

We also touch on some more advanced topics like **custom partitioning**, a reason you might want to use RDDs in the first place.

Key-Value Basics (Key-Value RDDs)

There are many methods on RDDs that require you to **put your data in a key–value format**. A hint that this is required is that the method will include **ByKey**. Whenever you see **ByKey** in a method name, it means that you can perform this only on a **PairRDD type.**

The easiest way is to just map over your current RDD to a basic **key–value structure.**

```
words.map(word => (word.toLowerCase, 1)).toDF().show(2)
//+-----+---+
//|   _1| _2|
//+-----+---+
//|spark|  1|
//|  the|  1|
//+-----+---+
```

keyBy

You can also use the **keyBy** function to achieve the same result by specifying a function that **creates the key from your current value.**

```
val keyword = words.keyBy(word => word.toLowerCase.toSeq(0).toString)
keyword.toDF().orderBy(col("_1").desc).show(3)
//+---+------+
//| _1|    _2|
//+---+------+
//|  t|   The|
//|  s|Simple|
//|  s| Spark|
//+---+------+
```

Mapping over Values

```
keyword.mapValues(word => word.toUpperCase).collect().foreach(println)
//(s,SPARK)
//(t,THE)
//(d,DEFINITIVE)
//(g,GUIDE)
//(:,:)
//(b,BIG)
//(d,DATA)
//(p,PROCESSING)
//(m,MADE)
//(s,SIMPLE)
/**
 * expand the number of rows that you have to make it so that each row
represents a character
 * like explode the value to a char
 */
keyword.flatMapValues(word => word.toUpperCase).collect().foreach(println)
//(s,S)
//(s,P)
//(s,A)
//(s,R)
//(s,K)
//(t,T)
```

Extracting Keys and Values

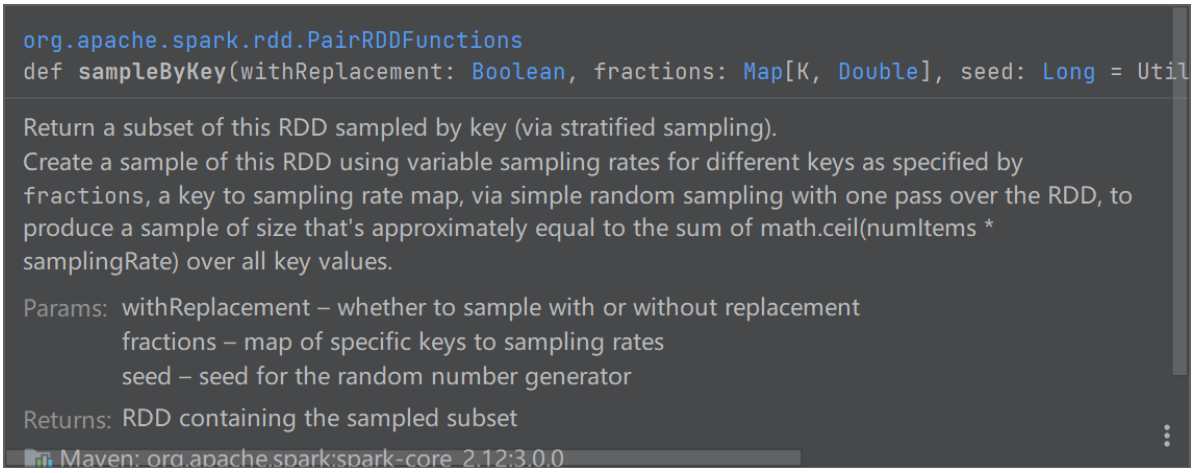
```
keyword.keys.collect()
keyword.values.collect()
```

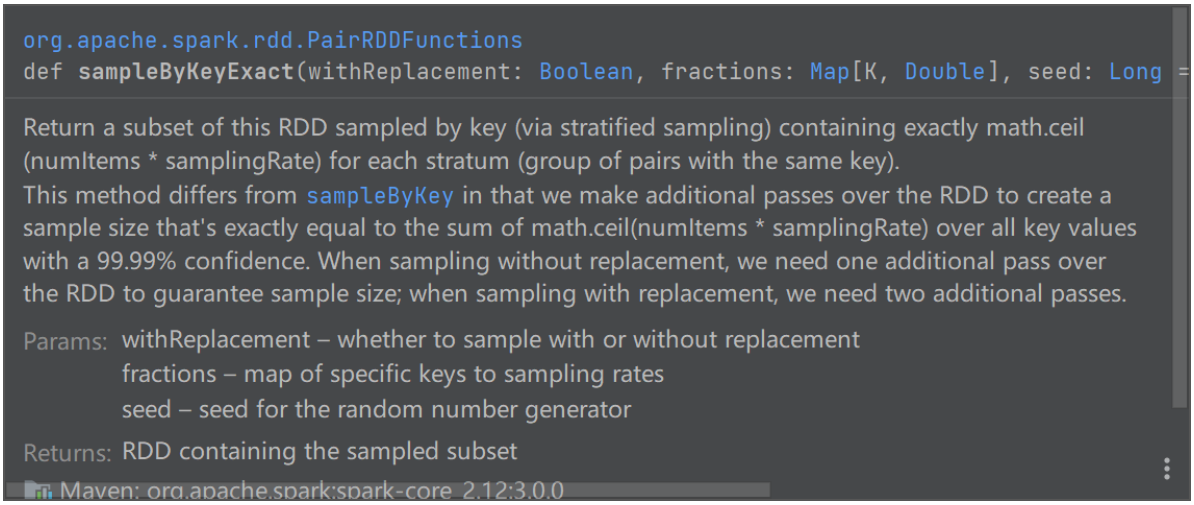
lookup

like map.get

```
keyword.lookup("s").toDF().show()
//+------+
//| value|
//+------+
//| Spark|
//|Simple|
//+------+
```

sampleByKey






```
val distinctChars = words.flatMap(word => word.toLowerCase.toSeq).distinct
  .collect()
import scala.util.Random
val sampleMap = distinctChars.map(c => (c, new Random().nextDouble())).toMap
words.map(word => (word.toLowerCase.toSeq(0), word))
  .sampleByKey(true, sampleMap, 6L)
  .collect()

words.map(word => (word.toLowerCase.toSeq(0), word))
  .sampleByKeyExact(true, sampleMap, 6L).collect()
```

Aggregations

```
val chars = words.flatMap(word => word.toLowerCase.toSeq)
val KVcharacters = chars.map(letter => (letter, 1))
def maxFunc(left:Int, right:Int) = math.max(left, right)
def addFunc(left:Int, right:Int) = left + right
val nums = sc.parallelize(1 to 30, 5)
```

countByKey

```
def countByKey(KVcharacters:  RDD[(Char, Int)]): Unit ={
  val timeout = 1000L //milliseconds
  val confidence = 0.95
  KVcharacters.countByKey().foreach(println)
  //(e,7)
  //(s,4)
  //(n,2)
  //(t,3)
  //(u,1)
  println(KVcharacters.countByKeyApprox(timeout, confidence))
  //final: Map(e -> [7.000, 7.000], s -> [4.000, 4.000], n -> [2.000, 2.000], t
-> [3.000, 3.000], u -> [1.000, 1.000]
}
```

Understanding Aggregation Implementations

compare the two fundamental choices, **groupBy** and **reduce**.

groupByKey

you might think groupByKey with a map over each grouping is the best way to sum up the counts for each key, However, this is, for the majority of cases, the wrong way to approach the problem.

The fundamental issue here is that **each executor must hold all values for a given key in memory** before applying the function to them. If you have massive key skew, some partitions might be completely overloaded with a ton of values for a given key, and you will get **OutOfMemoryErrors.**

There is a **preferred approach** for additive use cases: **reduceByKey.**

reduceByKey

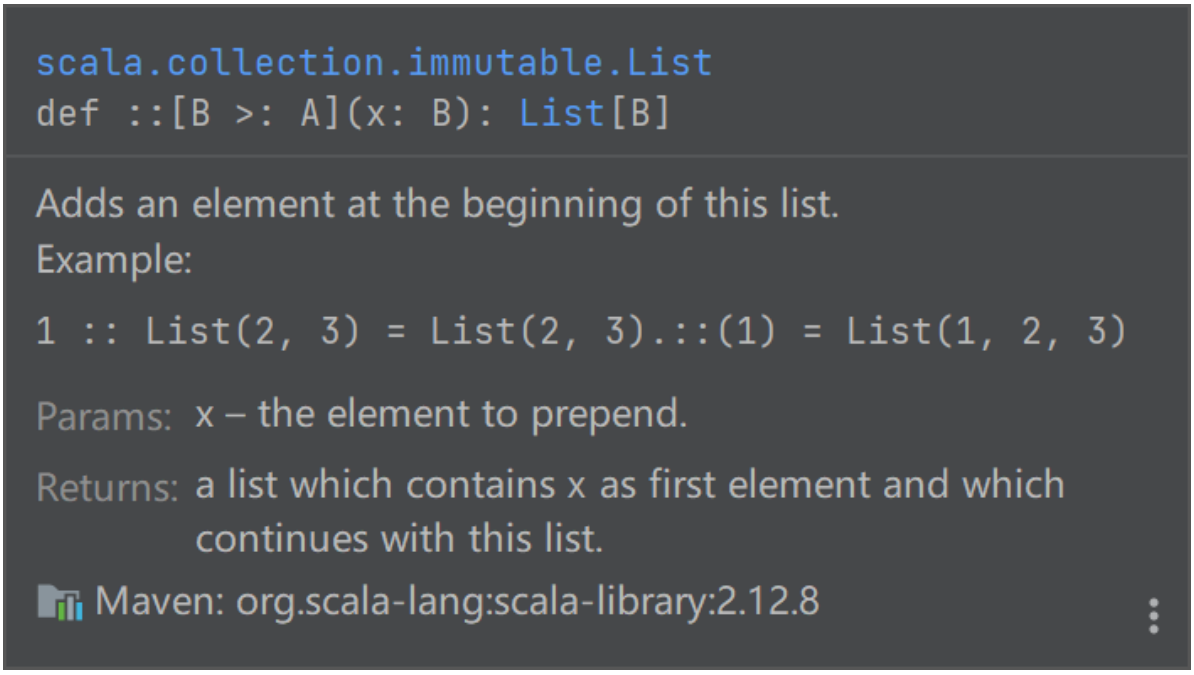
This implementation is much **more stable** because the **reduce happens within each partition** and **doesn't need to put everything in memory.**

Additionally, there is no incurred shuffle during this operation; everything happens at each worker individually before performing a final reduce.

```
def groupBy_and_reduce(KVcharacters:  RDD[(Char, Int)]): Unit ={
  def addFunc(left:Int, right:Int) = left + right
  KVcharacters.groupByKey().map(row => (row._1,
row._2.reduce(addFunc))).collect().foreach(println)
  KVcharacters.reduceByKey(addFunc).collect().foreach(print) //recommend
  //(d,4)(p,3)(t,3)(b,1)(h,1)(n,2)(f,1)(v,1)(:,1)(r,2)(l,1)(s,4)(e,7)(a,4)(i,7)
(k,1)(u,1)(o,1)(g,3)(m,2)(c,1)
}
```

Other Aggregation Methods

We find it very rare that users come across this sort of workload (or need to perform this kind of operation) in modern-day Spark. There just aren't that many reasons for using these extremely low-level tools when **you can perform much simpler aggregations using the Structured APIs.** These functions largely allow you very specific, very low-level control on exactly how a given aggregation is performed on the cluster of machines.



spark常用算子的区别与联系 - 知乎 (zhihu.com)

```
def aggregationsHandle(): Unit ={
  /**
   * load data
   */
  val chars = words.flatMap(word => word.toLowerCase.toSeq)
  val KVcharacters = chars.map(letter => (letter, 1))
  def maxFunc(left:Int, right:Int) = math.max(left, right)
  def addFunc(left:Int, right:Int) = left + right
  val nums = sc.parallelize(1 to 30, 5)
  //countByKey(KVcharacters)
  //groupBy_and_reduce(KVcharacters)
  //nums.saveAsTextFile("tmp/nums") //分区查看, 最大值分别为 6 12 18 24 30 总和为90
  /**
   *  The first aggregates within partitions, the second aggregates across
partitions.
   */
  println(nums.aggregate(0)(maxFunc, addFunc)) //90
  println(nums.aggregate(0)(maxFunc, maxFunc)) //30
  println(nums.aggregate(0)(addFunc, addFunc)) //465

  /**
   * 有reduceByKey味道了, 继续实现类似reduceByKey效果
   */
  KVcharacters.aggregateByKey(0)(addFunc, addFunc).foreach(print)
  //(s,4)(e,7)(a,4)(i,7)(k,1)(u,1)(o,1)(g,3)(m,2)(c,1)(d,4)(p,3)(t,3)(b,1)(h,1)
(n,2)(f,1)(v,1)(:,1)(r,2)(l,1)
  //KVcharacters.saveAsTextFile("tmp/KVcharacters") //分成两个区
  KVcharacters.aggregateByKey(0)(addFunc, maxFunc).collect().foreach(print)
```

```
  //(d,2)(p,2)(t,2)(b,1)(h,1)(n,1)(f,1)(v,1)(:,1)(r,1)(l,1)(s,3)(e,4)(a,3)(i,4)
(k,1)(u,1)(o,1)(g,2)(m,2)(c,1)
  /**
   * treeAggregate
   * treeAggregate that does the same thing as aggregate (at the user level)
   * but does so in a different way. It basically "pushes down"
   * some of the subaggregations (creating a tree from executor to executor)
   * before performing the final aggregation on the driver.
   */
  val depth = 3
  println(nums.treeAggregate(0)(maxFunc, addFunc, depth)) //90

  /**
   * combineByKey
   * This combiner operates on a given key and merges the values according to
some function.
   */
  val valToCombiner = (value:Int) => List(value)
  val mergeValuesFunc = (vals:List[Int], valToAppend:Int) => valToAppend :: vals
  val mergeCombinerFunc = (vals1:List[Int], vals2:List[Int]) => vals1 ::: vals2
  // now we define these as function variables
  val outputPartitions = 6
  val res = KVcharacters.combineByKey(
      valToCombiner,
      mergeValuesFunc,
      mergeCombinerFunc,
      outputPartitions)
  println(res.getNumPartitions) //6
  res.foreach(print)
  //(f,List(1))(r,List(1, 1))(l,List(1))(s,List(1, 1, 1, 1))(a,List(1, 1, 1, 1))
(g,List(1, 1, 1))(m,List(1, 1))(e,List(1, 1, 1, 1, 1, 1, 1))
  /**
   * foldByKey
   * foldByKey merges the values for each key using an associative function and
a neutral "zero value,"
   *  which can be added to the result an arbitrary number of times, and must
not change the result
   *  (e.g., 0 for addition, or 1 for multiplication)
   */
  KVcharacters.foldByKey(0)(addFunc).collect().foreach(print)
  //(d,4)(p,3)(t,3)(b,1)(h,1)(n,2)(f,1)(v,1)(:,1)(r,2)(l,1)(s,4)(e,7)(a,4)(i,7)
(k,1)(u,1)(o,1)(g,3)(m,2)(c,1)

}
```

CoGroups

CoGroups give you the ability to group together up to **three key–value RDDs** together in Scala and two in Python.

When doing this, you can also specify a number of **output partitions** or a custom partitioning function to control exactly how this data is distributed across the cluster

```
def coGroupHandle(): Unit ={
  import scala.util.Random
  val distinctChars = words.flatMap(word => word.toLowerCase.toSeq).distinct
  val charRDD = distinctChars.map(c => (c, new Random().nextDouble()))
  val charRDD2 = distinctChars.map(c => (c, new Random().nextDouble()))
  val charRDD3 = distinctChars.map(c => (c, new Random().nextDouble()))
  charRDD.cogroup(charRDD2, charRDD3).take(5).foreach(println)
  //(d,
(CompactBuffer(0.712732059741726),CompactBuffer(0.0652448690418771),CompactBuffe
r(0.361572231661929)))
  //(p,
(CompactBuffer(0.43600639026926347),CompactBuffer(0.8529280617098401),CompactBuf
fer(0.9393796625399176)))
  //(t,
(CompactBuffer(0.3484979108092584),CompactBuffer(0.8264329346284515),CompactBuff
er(0.47785931901197143)))
  //(b,
(CompactBuffer(0.45260022261432786),CompactBuffer(0.8962319635607489),CompactBuf
fer(0.5629007510889266)))
  //(h,
(CompactBuffer(0.8970724778298017),CompactBuffer(0.6730143374003479),CompactBuff
er(0.8022313767156555)))
}
```

Joins

RDDs joins all follow the same basic format: the two RDDs we would like to join, and, optionally, either the number of output partitions or the customer partition function to which they should output.

Inner Join

```
def joinHandler(): Unit ={
  val distinctChars = words.flatMap(word => word.toLowerCase.toSeq).distinct
  val chars = words.flatMap(word => word.toLowerCase.toSeq)
  val KVcharacters = chars.map(letter => (letter, 1))

  val keyedChars = distinctChars.map(c => (c, new Random().nextDouble()))
  val outputPartitions = 10
  KVcharacters.join(keyedChars).count()
  KVcharacters.join(keyedChars, outputPartitions).take(2).foreach(println)
  //(d,(1,0.40632910564886093))
  //(d,(1,0.40632910564886093))
}
```

```
def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))] = self.withScope {
  this.cogroup(other, partitioner).flatMapValues( pair =>
    for (v <- pair._1.iterator; w <- pair._2.iterator) yield (v, w)
  )
}
```

```
Perform a left outer join of this and other. For each element (k, v) in this, the resulting RDD will either
contain all pairs (k, (v, Some(w))) for w in other, or the pair (k, (v, None)) if no elements in other have key
k. Uses the given Partitioner to partition the output RDD.
def leftOuterJoin[W](
    other: RDD[(K, W)],
    partitioner: Partitioner): RDD[(K, (V, Option[W]))] = self.withScope {
  this.cogroup(other, partitioner).flatMapValues { pair =>
    if (pair._2.isEmpty) {
      pair._1.iterator.map(v => (v, None))
    } else {
```

We won't provide an example for the other joins, but they all follow the same basic format. You can learn about the following join types at the conceptual level in Chapter 8:

- `fullOuterJoin`
- `leftOuterJoin`
- `rightOuterJoin`
- `cartesian` (This, again, is very dangerous! It does not accept a join key and can have a massive output.)

zip

```
    /**
     * zip, zip allows you to “zip” together two RDDs, assuming that they have
the
same length. This creates a PairRDD. The two RDDs must have the same number of
partitions as
well as the same number of elements:
     */
//    val numRange = sc.parallelize(0 to 9, 4)
//    //Exception in thread "main" java.lang.IllegalArgumentException: Can't zip
RDDs with unequal numbers of partitions: List(2, 4)
    val numRange = sc.parallelize(0 to 9, 2)
    words.zip(numRange).mapPartitions(x => x.map(line =>
println(line))).collect()
    //(Spark,0)
    //(Big,5)
    //(The,1)
    //(Data,6)
    //(Definitive,2)
    //(Processing,7)
    //(Guide,3)
    //(Made,8)
    //(:,4)
    //(Simple,9)
    //按数字顺序排序就是正确的书名
```

Controlling Partitions

The key addition (that does not exist in the Structured APIs) is the ability to specify a **partitioning function** (formally a custom Partitioner, which we discuss later when we look at basic methods).

coalesce

coalesce effectively **collapses partitions on the same worker** in order to **avoid a shuffle** of the data when repartitioning.

```
def coalesce(): Unit ={
  println(words.coalesce(4).getNumPartitions) //2
  println(words.coalesce(1).getNumPartitions) //1
}
```

repartition

The **repartition** operation allows you to **repartition your data up or down** but **performs a shuffle across nodes in the process.**

```
def repartition(): Unit ={
  println(words.repartition(10).getNumPartitions) //10
  println(words.repartition(1).getNumPartitions) //1
}
```

repartitionAndSortWithinPartitions

We'll omit the example because the documentation for it is good, but both the partitioning and the key comparisons can be specified by the user.

Custom Partitioning(data skew problem)

This ability is one of the primary reasons you'd want to use RDDs.

Custom partitioners are not available in the Structured APIs because **they don't really have a logical counterpart.**

The canonical example to motivate custom partition for this operation is PageRank whereby we seek to **control the layout of the data on the cluster** and **avoid shuffles.**

In our shopping dataset, this might mean partitioning by each customer ID.

In short, the sole goal of custom partitioning is to even out the distribution of your data across the cluster so that you can work around problems like data skew.

If you're going to use custom partitioners, you should drop down to RDDs from the Structured APIs, apply your custom partitioner, and then convert it back to a DataFrame or Dataset. This way, you get the best of both worlds, only dropping down to custom partitioning when you need to.

To perform custom partitioning you need to implement your own class that extends **Partitioner.**

Spark has two built-in Partitioners that you can leverage off in the RDD API, a **HashPartitioner** for **discrete values** and a **RangePartitioner**. These two work for discrete values and **continuous values.**

Although the hash and range partitioners are useful, they're fairly **rudimentary.**

> **Key skew** simply means that some keys have many, many more values than other keys. You want to break these keys as much as possible to improve parallelism and prevent OutOfMemoryErrors during the course of execution.

One instance might be that you need to partition more keys if and only if the key matches a certain format. For instance, we might know that there are two customers in your dataset that always crash your analysis and we need to **break them up further than other customer IDs.** In fact, these two are so skewed that they need to be operated on alone, whereas all of the others can be lumped into large groups. This is obviously a bit of a caricatured example, but you might see similar situations in your data, as well

```
def customPartitionHandle(): Unit ={
  val df = spark.read.option("header", "true").option("inferSchema", "true")
    .csv("src/data/retail-data/all/")
  val rdd = df.coalesce(10).rdd

  import org.apache.spark.HashPartitioner
  rdd.map(r => r(6)).take(5).foreach(println)
  //17850
  //17850
  //17850
  //17850
  //17850
  val keyedRDD = rdd.keyBy(row => row(6).asInstanceOf[Int].toDouble)
  keyedRDD
    .partitionBy(new
HashPartitioner(4)).map(_._1).glom().map(_.toSet.toSeq.length)
    .take(5).foreach(println)
  //4373
  //0
  //0
  //0
  keyedRDD.partitionBy(new HashPartitioner(10)).take(10).foreach(println)
  //(15100.0,[536374,21258,VICTORIAN SEWING BOX LARGE,32,12/1/2010
9:09,10.95,15100,United Kingdom])
  //(16250.0,[536388,21754,HOME BUILDING BLOCK WORD,3,12/1/2010
9:59,5.95,16250,United Kingdom])
  //(16250.0,[536388,21755,LOVE BUILDING BLOCK WORD,3,12/1/2010
9:59,5.95,16250,United Kingdom])
  //(16250.0,[536388,21523,DOORMAT FANCY FONT HOME SWEET HOME,2,12/1/2010
9:59,7.95,16250,United Kingdom])
  //(16250.0,[536388,21363,HOME SMALL WOOD LETTERS,3,12/1/2010
9:59,4.95,16250,United Kingdom])
  //(16250.0,[536388,21411,GINGHAM HEART  DOORSTOP RED,3,12/1/2010
9:59,4.25,16250,United Kingdom])
  //(16250.0,[536388,22318,FIVE HEART HANGING DECORATION,6,12/1/2010
9:59,2.95,16250,United Kingdom])
  //(16250.0,[536388,22464,HANGING METAL HEART LANTERN,12,12/1/2010
9:59,1.65,16250,United Kingdom])
  //(16250.0,[536388,22915,ASSORTED BOTTLE TOP  MAGNETS ,12,12/1/2010
9:59,0.42,16250,United Kingdom])
  //(16250.0,[536388,22922,FRIDGE MAGNETS US DINER ASSORTED,12,12/1/2010
9:59,0.85,16250,United Kingdom])

  import org.apache.spark.Partitioner
  class DomainPartitioner extends Partitioner {
    def numPartitions = 3
```

```
    def getPartition(key: Any): Int = {
      val customerId = key.asInstanceOf[Double].toInt
      if (customerId == 17850.0 || customerId == 12583.0) {
        return 0
      } else {
        return new java.util.Random().nextInt(2) + 1
      }
    }
  }
  keyedRDD
    .partitionBy(new DomainPartitioner)
    .take(10).foreach(println)
  //(17850.0,[536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,12/1/2010
8:26,2.55,17850,United Kingdom])
  //(17850.0,[536365,71053,WHITE METAL LANTERN,6,12/1/2010
8:26,3.39,17850,United Kingdom])
  //(17850.0,[536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,12/1/2010
8:26,2.75,17850,United Kingdom])
  //(17850.0,[536365,84029G,KNITTED UNION FLAG HOT WATER BOTTLE,6,12/1/2010
8:26,3.39,17850,United Kingdom])
  //(17850.0,[536365,84029E,RED WOOLLY HOTTIE WHITE HEART.,6,12/1/2010
8:26,3.39,17850,United Kingdom])
  //(17850.0,[536365,22752,SET 7 BABUSHKA NESTING BOXES,2,12/1/2010
8:26,7.65,17850,United Kingdom])
  //(17850.0,[536365,21730,GLASS STAR FROSTED T-LIGHT HOLDER,6,12/1/2010
8:26,4.25,17850,United Kingdom])
  //(17850.0,[536366,22633,HAND WARMER UNION JACK,6,12/1/2010
8:28,1.85,17850,United Kingdom])
  //(17850.0,[536366,22632,HAND WARMER RED POLKA DOT,6,12/1/2010
8:28,1.85,17850,United Kingdom])
  //(12583.0,[536370,22728,ALARM CLOCK BAKELIKE PINK,24,12/1/2010
8:45,3.75,12583,France])
  /**
   * count of results in each partition
   */
  keyedRDD
    .partitionBy(new
DomainPartitioner).map(_._1).glom().map(_.toSet.toSeq.length)
    .take(5).foreach(println)
  //2
  //4306
  //4306
  class myPartitioner extends Partitioner {
    def numPartitions = 4
    def getPartition(key: Any): Int = {
      //key.hashCode() % 4
      //4373
      //0
      //0
      //0
      if (key.asInstanceOf[Double].toInt == 17850.0) return 0
      else return new java.util.Random().nextInt(3) + 1
    }
  }
  keyedRDD
    .partitionBy(new myPartitioner).map(_._1).glom().map(_.toSet.toSeq.length)
    .take(4).foreach(println)
  //1
```

```
  //4255
  //4239
  //4228
}
```

Custom Serialization

The last advanced topic that is worth talking about is the issue of **Kryo serialization**.

Any object that you hope to parallelize (or function) must be serializable.

The default serialization can be quite slow. Spark can use the **Kryo library** (version 2) to serialize objects **more quickly.**

Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all serializable types and requires you to **register the classes you'll use in the program in advance for best performance.**

You can use Kryo by initializing your job with a SparkConf and setting the value of **"spark.serializer"** to **"org.apache.spark.serializer.KryoSerializer"**

To register your own custom classes with Kryo, use the **registerKryoClasses** method:

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

Distributed Shared Variables

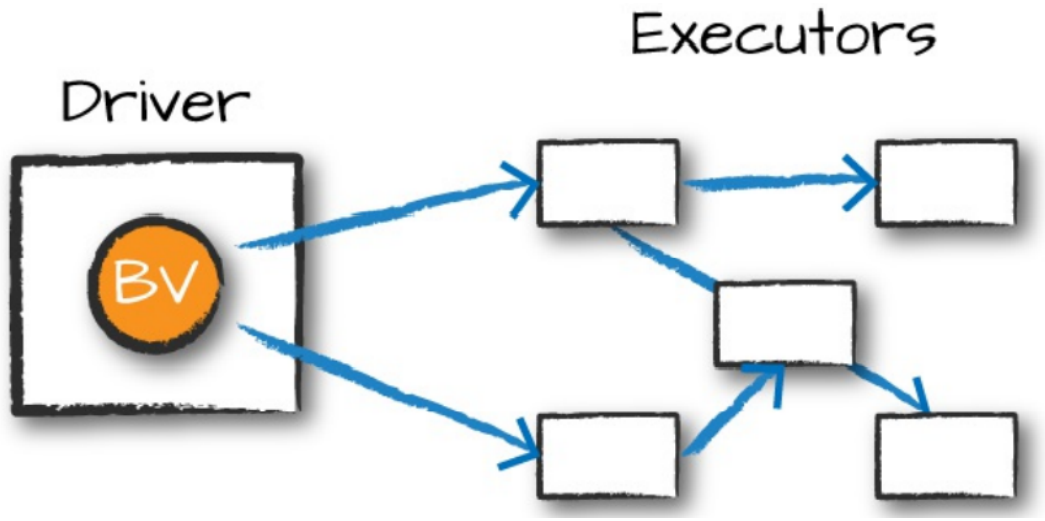
In addition to the Resilient Distributed Dataset (RDD) interface, the second kind of **low-level API** in Spark is two types of **"distributed shared variables"**: **broadcast variables** and **accumulators**.

- **accumulators** let you add together data from all the tasks into a shared result (e.g., to implement a counter so you can see how many of your job's input records failed to parse),
- **broadcast variables** let you save a large value on all the worker nodes and reuse it across many Spark actions without re-sending it to the cluster.

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 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.



The canonical use case is to pass around a large lookup table that fits in memory on the executors and use that in a function:

```
def broadcastVariablesHandle(): Unit ={
  val myCollection = "Spark The Definitive Guide : Big Data Processing Made
Simple"
    .split(" ")
  val words = spark.sparkContext.parallelize(myCollection, 2)

  val supplementalData = Map("Spark" -> 1000, "Definitive" -> 200,
    "Big" -> -300, "Simple" -> 100)

  /**
   * We can broadcast this structure across Spark and reference it by using
suppBroadcast
   * This value is immutable and is lazily replicated across all nodes in the
cluster when we trigger an action
   */
  val suppBroadcast = spark.sparkContext.broadcast(supplementalData)

  suppBroadcast.value.foreach(println)
  //(Spark,1000)
  //(Definitive,200)
  //(Big,-300)
  //(Simple,100)
  println(suppBroadcast.value.getClass) //class
scala.collection.immutable.Map$Map4

  words.map(word => (word, suppBroadcast.value.getOrElse(word, 0)))
    .sortBy(wordPair => wordPair._2)
    .collect().foreach(println)
  //(Big,-300)
  //(The,0)
  //(Guide,0)
  //(:,0)
  //(Data,0)
  //(Processing,0)
  //(Made,0)
  //(Simple,100)
  //(Definitive,200)
  //(Spark,1000)
```

```
}
```

The only difference between this and passing it into the closure is that we have done this in a much **more efficient manner.**

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.

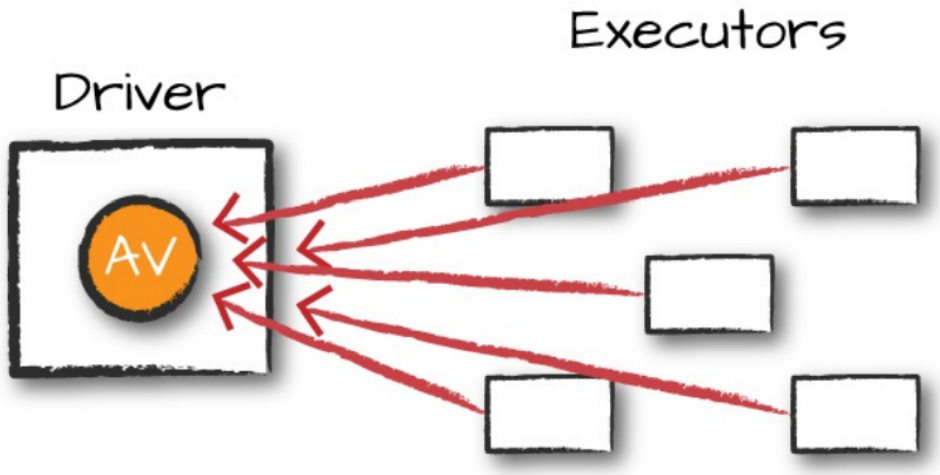


Figure 14-2. Accumulator variable

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.**

Spark natively supports accumulators of **numeric types**, and programmers **can add support for new types.**

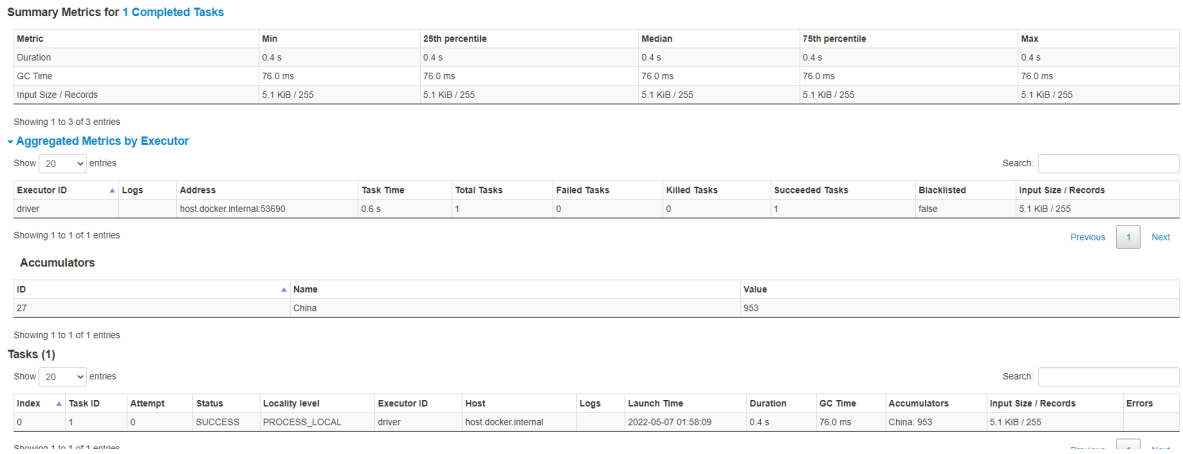
For accumulator updates performed inside **actions only**, Spark guarantees that **each task's update** to the accumulator will be applied **only once**, meaning that restarted tasks will not update the value. **In transformations, you should be aware that each task's update can be applied more than once if tasks or job stages are reexecuted.**

Accumulators do not change the **lazy evaluation model** of Spark.

Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like map().

```
def accumulatorHandle(): Unit ={
  val flights = spark.read
    .parquet("src/data/flight-data/parquet/2010-summary.parquet")
    .as[Flight]

  /**
   *  count the number of flights to or from China
   */
  import org.apache.spark.util.LongAccumulator
  val accUnnamed = new LongAccumulator
  val acc = spark.sparkContext.register(accUnnamed)

  /**
   * instantiate the accumulator and register it with a name
```

```
    */
  val accChina = new LongAccumulator
  val accChina2 = spark.sparkContext.longAccumulator("China")
  spark.sparkContext.register(accChina, "China")
  //Named accumulators will display in the Spark UI, whereas unnamed ones will
not.
  def accChinaFunc(flight_row: Flight): Unit = {
    val destination = flight_row.DEST_COUNTRY_NAME
    val origin = flight_row.ORIGIN_COUNTRY_NAME
    if (destination == "China") {
      accChina.add(flight_row.count.toLong)
    }
    else if (origin == "China") {
      accChina.add(flight_row.count.toLong)
    }
  }

  /**
   * iterate over every row in our flights dataset via the foreach method
   *  foreach is an action
   *  Spark can provide guarantees that perform only inside of actions
   */
  flights.foreach(flight_row => accChinaFunc(flight_row))

  println(accChina.value) //953
}
```



Custom Accumulators

Although Spark does provide some default accumulator types, sometimes you might want to **build your own custom accumulator.**

In order to do this you need to subclass the **AccumulatorV2 class**

```
class EvenAccumulator extends AccumulatorV2[BigInt, BigInt] {
  private var num:BigInt = 0
  private var _count:BigInt = 0
  def sum : BigInt = num
  def count : BigInt = _count
  def avg : Double = num.toDouble / _count.toLong
  def reset(): Unit = {
    this.num = 0
    this._count = 0
  }
```

```
  def add(intValue: BigInt): Unit = {
    if (intValue % 2 == 0) {
      this.num += intValue
      this._count += 1
    }
  }
  def merge(other: AccumulatorV2[BigInt,BigInt]): Unit = other match{
    case o: EvenAccumulator =>
      num += o.value()
    _count += o.count
    case _ =>
      throw new UnsupportedOperationException(
        s"Cannot merge ${this.getClass.getName} with ${other.getClass.getName}")
  }
  def value():BigInt = {
    this.num
  }
  def copy(): AccumulatorV2[BigInt,BigInt] = {
    new EvenAccumulator
  }
  def isZero():Boolean = {
    this.num == 0
  }
}
```

```
def customAccumulatorHandle(): Unit ={
  val flights = spark.read
    .parquet("src/data/flight-data/parquet/2010-summary.parquet")
    .as[Flight]
  import scala.collection.mutable.ArrayBuffer
  import org.apache.spark.util.AccumulatorV2
  val arr = ArrayBuffer[BigInt]()
  val acc = new EvenAccumulator
  sc.register(acc, "evenAcc")
  // in Scala
  println(acc.value) // 0
  flights.foreach(flight_row => acc.add(flight_row.count))
  println(acc.value) // 31390
  println(acc.count) //122
  println(acc.avg) //257.2950819672131
}
```

Part IV. Production Applications

How Spark Runs on a Cluster

This chapter focuses on what happens when Spark goes about executing that code.

The Architecture of a Spark Application

Some of the **high-level components** of a Spark Application

- **The Spark driver**

The **driver** is the **process** "in the driver seat" 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 responsible 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 (success or failure) and results.

Each Spark Application has its own separate executor processes.

- **The cluster manager**

The Spark Driver and Executors do not exist in a void, and this is where the cluster manager comes in.

The cluster manager is responsible for **maintaining a cluster of machines** that will run your Spark Application(s).

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)

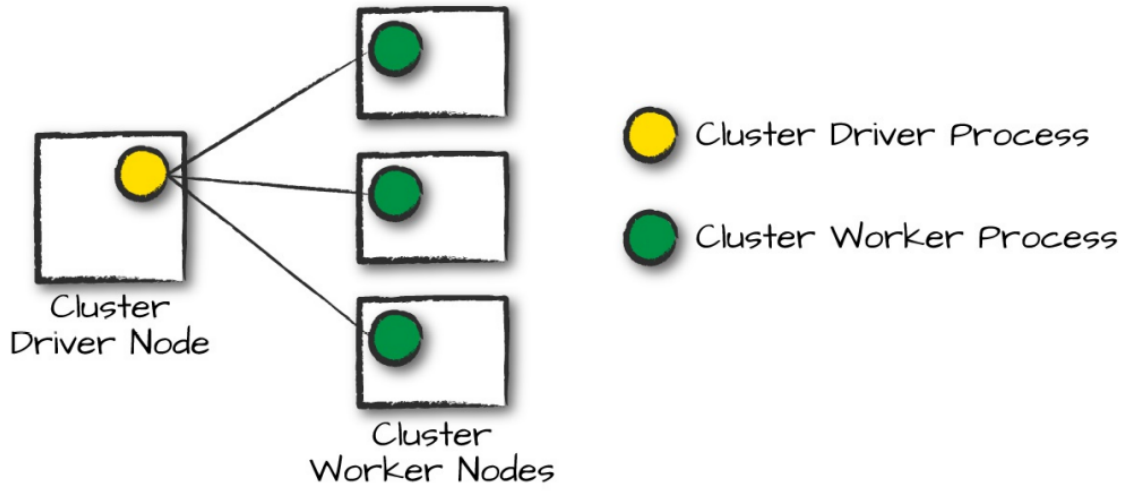


Figure 15-1. A cluster driver and worker (no Spark Application yet)

> The machine on the left of the illustration is the **Cluster Manager Driver Node.**

1. When it comes time to actually run a Spark Application, we request resources from the cluster manager to run it.
2. Depending on how our application is configured, this can include a place to run the Spark driver or might be just resources for the executors for our Spark Application.
3. Over the course of Spark Application execution, the cluster manager will be responsible for managing the underlying machines that our application is running on.

Spark currently supports three cluster managers: a simple built-in **standalone cluster manager**, **Apache Mesos**, and **Hadoop YARN**

Cluster Manager Types

The system currently supports several cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications. (Deprecated)
- Hadoop YARN – the resource manager in Hadoop 2.
- Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

> spark 3.2.1 support k8s

Execution Modes

An execution mode gives you the power to determine where the aforementioned resources are physically located when you go to run your application.

- Cluster mode
- Client mode
- Local mode

rectangles with **solid borders** represent Spark **driver process** whereas those with **dotted borders** represent the **executor processes.**

Cluster mode

Cluster mode is probably the most common way of running Spark Applications. 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.**

The cluster manager is responsible for maintaining all Spark Application–related processes.

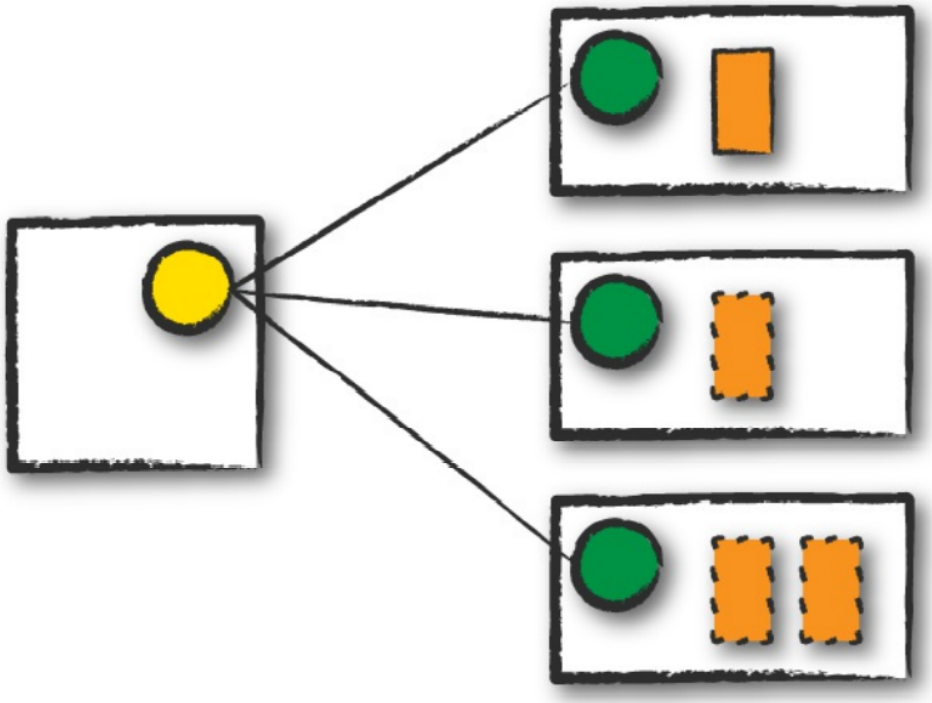


Figure 15-2. Spark's cluster mode

> Figure 15-2 shows that the cluster manager placed our driver on a worker node and the executors on other worker nodes

Client mode

The **client machine** is responsible for **maintaining the Spark driver process**, and the **cluster manager maintains the executor processses.**

In Figure 15-3, 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

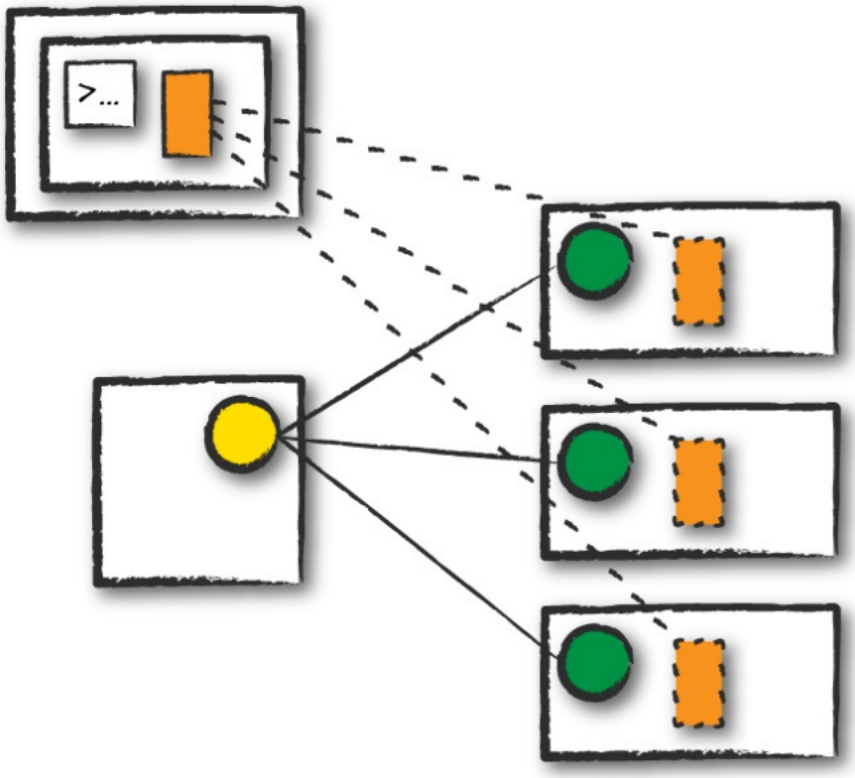


Figure 15-3. Spark's client mode

Local mode

Local mode is a significant departure from the previous two modes: it runs the entire Spark Application on a single machine. It achieves parallelism through **threads** on that **single machine.**

However, we **do not recommend using local mode for running production applications.**

The Life Cycle of a Spark Application (Outside Spark)

We assume that a cluster is already running with **four nodes, a driver** (not a Spark driver but cluster manager driver) and **three worker nodes**.

Client Request

The first step is for you to submit an actual application. This will be a pre-compiled JAR or library.

you are executing code on your local machine and you're going to make a request to the cluster manager driver node

Here, we are explicitly asking for resources for the **Spark driver process** only.

We assume that the cluster manager accepts this offer and **places the driver onto a node in the cluster.**

The **client process that submitted the original job exits** and the application is off and **running on the cluster.**

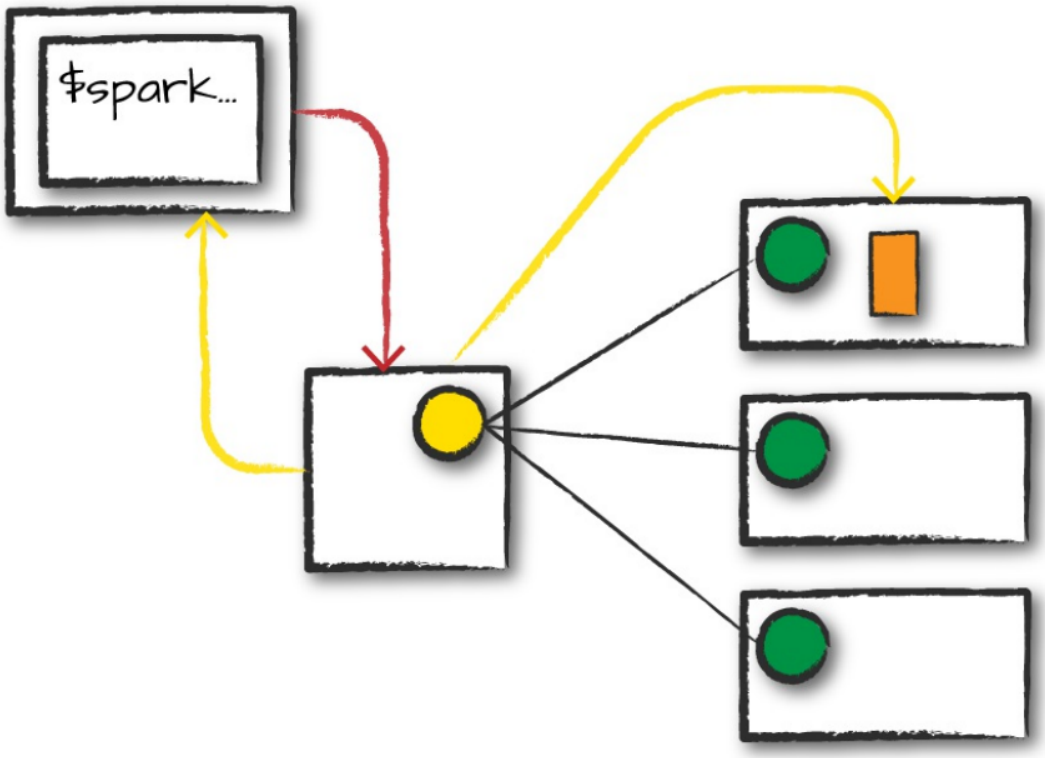


Figure 15-4. Requesting resources for a driver

```
[hadoop@hadoop01 spark_data]$ spark-submit --class cn.com.chinahitech.Analysis \
> --master local[2] \
> film_danmu_analysis.jar \
> --executor-memory 1G \
> --executor-cores 1 \
> --num-executors 1
2020-07-23 16:53:39,557 WARN util.NativeCodeLoader: Unable to load native-hadoop lib
plicable
-----------存储到hdfs成功！！-------------
-----------存储到MySQL成功！！-------------
```

```
./bin/spark-submit \
    --class   <main-class> \
    --master  <master-url> \
    --deploy-mode cluster \
    --conf = <key>=<value> \
    ... # other options
    <application-jar> \
    [application-arguments]
```

Launch

The driver process has been placed on the cluster, it begins **running user code.** This code must include a **SparkSession** that **initializes a Spark cluster** (e.g., driver + executors).

The **SparkSession** will subsequently communicate with the cluster manager (the darker line), asking it to launch Spark executor processes across the cluster (the lighter lines).

The **number of executors** and their relevant configurations are set by the user **via the command-line arguments** in the original **spark-submit** call.

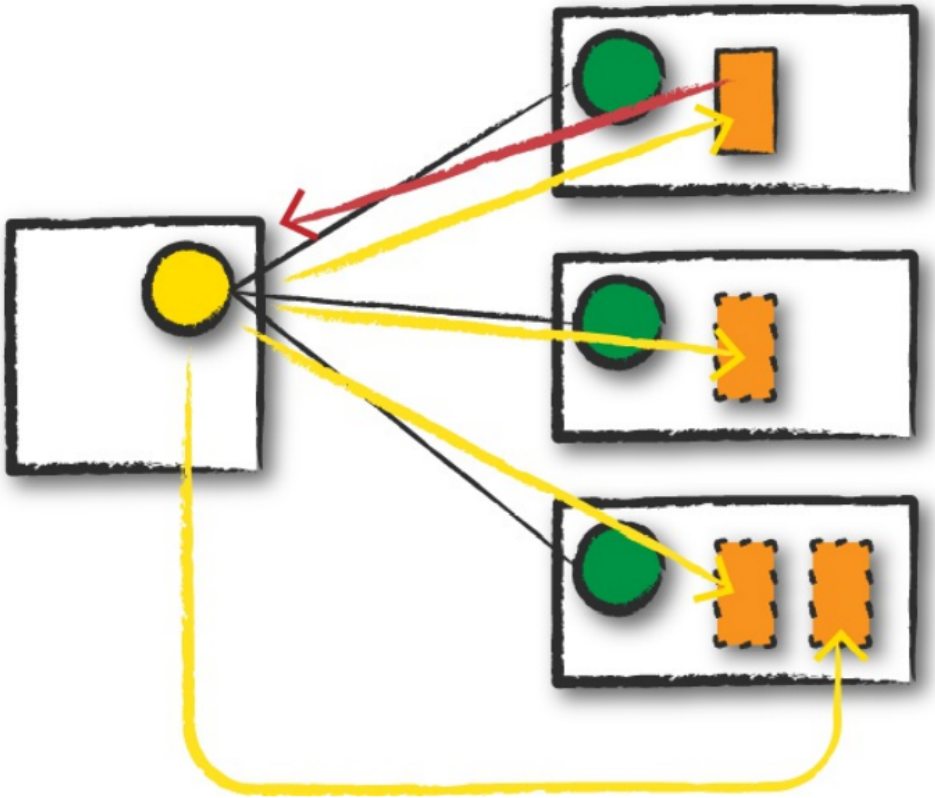


Figure 15-5. Launching the Spark Application

The cluster manager responds by launching the executor processes (assuming all goes well) and sends the relevant information about their locations to the driver process.

After everything is hooked up correctly, we have a **"Spark Cluster"**

Execution

Now that we have a "Spark Cluster," Spark goes about its merry way executing code.

The driver and the workers communicate among themselves, executing code and moving data around.

The driver schedules tasks onto each worker, and each worker responds with the **status of those tasks and success or failure.**

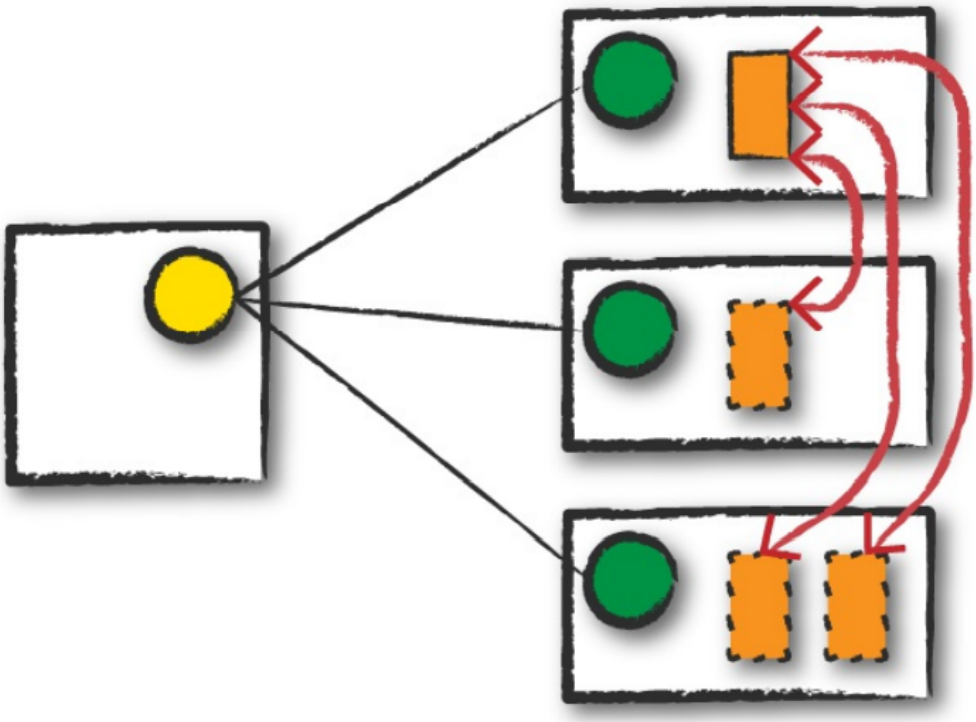


Figure 15-6. Application execution

Completion

After a Spark Application completes, **the driver processs exits** with either success or failure

The cluster manager then shuts down the executors in that Spark cluster for the driver.

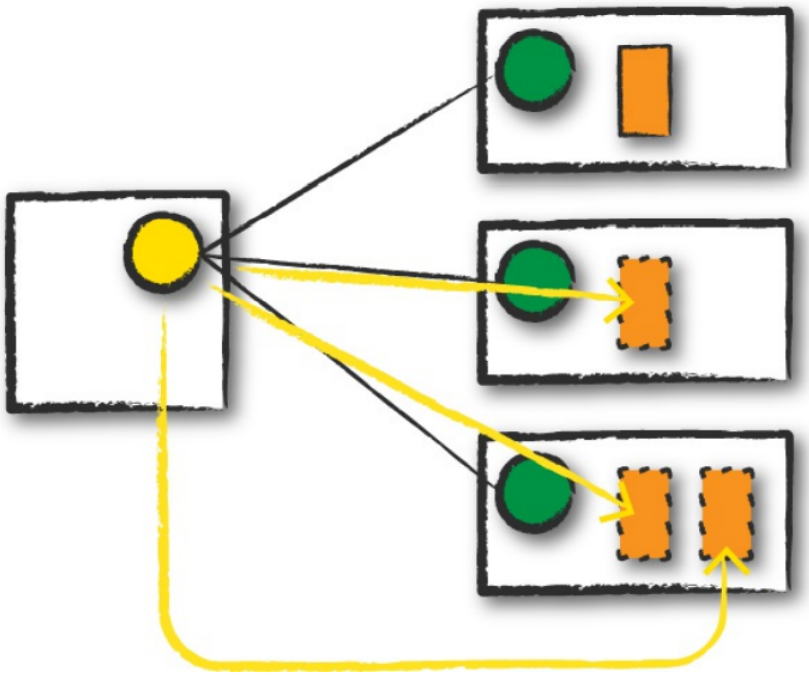


Figure 15-7. Shutting down the application

The Life Cycle of a Spark Application (Inside Spark)

Each application is made up of one or more **Spark jobs.**

Spark jobs within an application are executed **serially (unless you use threading to launch multiple actions in parallel).**

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.**

Some of your legacy code might use the new SparkContext pattern. This should be avoided in favor of **the builder method on the SparkSession, which more robustly instantiates the Spark and SQL Contexts and ensures that there is no context conflict**, given that there might be multiple libraries trying to create a session in the same Spark Appication:

```
// Creating a SparkSession in Scala
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("Databricks Spark Example")
.config("spark.sql.warehouse.dir", "/user/hive/warehouse")
.getOrCreate()
```

Older code you might find would instead directly create a **SparkContext** and a **SQLContext** for the structured APIs.

The SparkContext

A S**parkContext object within the SparkSession** represents the **connection to the Spark cluster.**

For the most part, you should not need to explicitly initialize a SparkContext; you should just be able to **access it through the SparkSession**

```
// in Scala
import org.apache.spark.SparkContext
val sc = SparkContext.getOrCreate()
```

THE SPARKSESSION, SQLCONTEXT, AND HIVECONTEXT

In previous versions of Spark, the `SQLContext` and `HiveContext` provided the ability to work with DataFrames and Spark SQL and were commonly stored as the variable `sqlContext` in examples, documentation, and legacy code. As a historical point, Spark 1.X had effectively two contexts. The `SparkContext` and the `SQLContext`. These two each performed different things. The former focused on more fine-grained control of Spark's central abstractions, whereas the latter focused on the higher-level tools like Spark SQL. In Spark 2.X, the communtiy combined the two APIs into the centralized `SparkSession` that we have today. However, both of these APIs still exist and you can access them via the `SparkSession`. It is important to note that you should never need to use the `SQLContext` and rarely need to use the `SparkContext`.

Logical Instructions

Spark code essentially consists of **transformations and actions.**

Logical instructions to physical execution

```
val df1 = spark.range(2, 10000000, 2)
val df2 = spark.range(2, 10000000, 4)
val step1 = df1.repartition(5)
val step12 = df2.repartition(6)
val step2 = step1.selectExpr("id * 5 as id2")
val step3 = step2.join(step12, step2("id2") === step12("id"))
val step4 = step3.select(expr("sum(id)"))
step4.collect().foreach(println) //[2500000000000]
step4.explain()
//== Physical Plan ==
//*(7) HashAggregate(keys=[], functions=[sum(id#2L)])
//+- Exchange SinglePartition, true, [id=#66]
//   +- *(6) HashAggregate(keys=[], functions=[partial_sum(id#2L)])
//      +- *(6) Project [id#2L]
//         +- *(6) SortMergeJoin [id2#8L], [id#2L], Inner
//            :- *(3) Sort [id2#8L ASC NULLS FIRST], false, 0
//            :  +- Exchange hashpartitioning(id2#8L, 200), true, [id=#50]
//            :     +- *(2) Project [(id#0L * 5) AS id2#8L]
//            :        +- Exchange RoundRobinPartitioning(5), false, [id=#46]
//            :           +- *(1) Range (2, 10000000, step=2, splits=8)
//            +- *(5) Sort [id#2L ASC NULLS FIRST], false, 0
//               +- Exchange hashpartitioning(id#2L, 200), true, [id=#57]
//                  +- Exchange RoundRobinPartitioning(6), false, [id=#56]
//                     +- *(4) Range (2, 10000000, step=4, splits=8)
```

A Spark Job

In general, there should be **one Spark job for one action**.

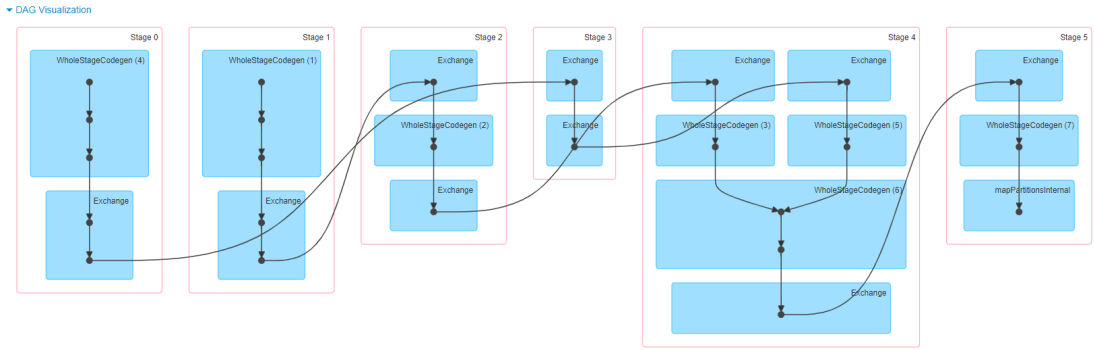
Actions always return results.

Each job breaks down into a series of **stages**, the number of which depends on how many **shuffle operations** need to take place.

- Stage 0 with 8 Tasks
- Stage 1 with 8 Tasks
- Stage 2 with 6 Tasks
- Stage 3 with 5 Tasks
- Stage 4 with 200 Tasks
- Stage 5 with 1 Task

Page: 1 — 1 Pages. Jump to 1. Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	main at <unknown>:0 +details	2022/05/07 14:49:33	0.1 s	1/1			11.5 KiB	
4	main at <unknown>:0 +details	2022/05/07 14:49:30	3 s	200/200			38.0 MiB	11.5 KiB
3	main at <unknown>:0 +details	2022/05/07 14:49:29	2 s	6/6			12.2 MiB	12.7 MiB
2	main at <unknown>:0 +details	2022/05/07 14:49:28	2 s	5/5			24.4 MiB	25.3 MiB
1	main at <unknown>:0 +details	2022/05/07 14:49:27	2 s	8/8				24.4 MiB
0	main at <unknown>:0 +details	2022/05/07 14:49:27	0.7 s	8/8				12.2 MiB



Stages

Stages in Spark **represent groups of tasks that can be executed together to compute the same operation on multiple machines.**

In general, **Spark will try to pack as much work as possible** (i.e., as many transformations as possible inside your job) into the same stage, but the engine **starts new stages after operations called *shuffles.***

A shuffle represents a physical repartitioning of the data—for example, sorting a DataFrame, or grouping data that was loaded from a file by key (which **requires sending records with the same key to the same node**). This type of repartitioning requires coordinating across executors to move data around.

Spark starts a new stage after each shuffle, and keeps track of what order the stages must run in to compute the final result.

1. In the job we looked at earlier, the first two stages correspond to the range that you perform in order to create your DataFrames. **By default when you create a DataFrame with range, it has eight partitions.**
2. The next step is the **repartitioning**. This changes the number of partitions by shuffling the data. These DataFrames are shuffled into six partitions and five partitions, corresponding to the number of tasks in stages 2 and 3.
3. Stages 2 and 3 perform on each of those DataFrames and the end of the stage represents the join (a shuffle). Suddenly, we have 200 tasks. This is because of a Spark SQL configuration. The **spark.sql.shuffle.partitions default value is 200,** which means that when **there is a shuffle performed during execution, it outputs 200 shuffle partitions by default.**

> **TIP**
>
> We cover the number of partitions in a bit more detail in Chapter 19 because it's such an important parameter. This value should be set according to the number of cores in your cluster to ensure efficient execution. Here's how to set it:
>
> ```
> spark.conf.set("spark.sql.shuffle.partitions", 50)
> ```

> **A good rule of thumb is that the number of partitions should be larger than the number of executors on your cluster, potentially by multiple factors depending on the workload.**

This is more of a default for a cluster in which there might be many more executor cores to use.

Regardless of the number of partitions, that **entire stage is computed in parallel.**

4. The final result aggregates those partitions individually, brings them all to a single partition before finally sending the final result to the driver.

Tasks

Stages in Spark consist of **tasks**.

Each task corresponds to a combination of blocks of data and a set of transformations that will run **on a single executor**.

If there is one big partition in our dataset, we will have one task. If there are 1,000 little partitions, we will have 1,000 tasks that can be executed in parallel.

A task is just a unit of computation applied to a unit of data (the partition).

Partitioning your data into a greater number of partitions means that more can be executed in parallel. This is **not a panacea**, but it is a simple place to begin with optimization.

Execution Details

- 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

An important part of what makes Spark **an "in-memory computation tool"** is that **Spark performs as many steps as it can at one point in time before writing data to memory or disk.**

One of the key optimizations that Spark performs is **pipelining**, which **occurs at and below the RDD level.**

With pipelining, any sequence of operations that feed data directly into each other, without needing to move it across nodes, is collapsed into a single stage of tasks that do all the operations together.

For example, if you write an RDD-based program that does **a map, then a filter, then another map**, these will **result in a single stage of tasks that immediately read each input record,** pass it through the first map, pass it through the filter, and pass it through the last map function if needed.

> This pipelined version of the computation is much faster than writing the intermediate results to memory or disk after each step.

pipelining will be transparent to you as you write an application —the Spark runtime will automatically do it

but you will see that multiple RDD or DataFrame operations were pipelined into a single stage.

Shuffle Persistence

The second property you'll sometimes see is shuffle persistence.

When Spark needs to run an operation that has to move data across nodes, such as a reduce-by-key operation (**where input data for each key needs to first be brought together from many nodes**), the engine can't perform pipelining anymore, and instead it performs **a cross-network shuffle.**

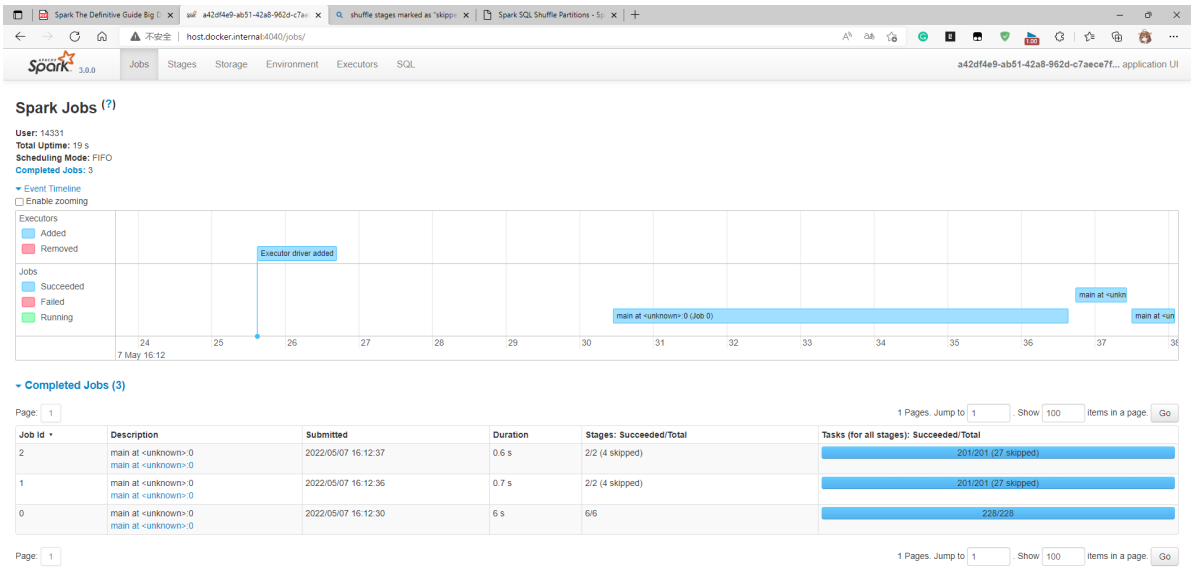
Spark always executes **shuffles** by first having the “source” tasks (those sending data) **write shuffle files to their local disks during their execution stage.**

Then, the stage that does the **grouping and reduction launches** and **runs tasks that fetch their corresponding records from each shuffle file** and **performs that computation**

Saving the shuffle files to disk

- lets Spark run this stage later in time than the source stage (e.g., if there are not enough executors to run both at the same time)
- lets the engine re-launch reduce tasks on failure without rerunning all the input tasks.

One side effect you’ll see for shuffle persistence is that running a new job over data that’s already been shuffled does not rerun the “source” side of the shuffle. Because the shuffle files were already written to disk earlier, Spark knows that it can use them to run the later stages of the job, and it need not redo the earlier ones. In the Spark UI and logs, you will see the pre-shuffle stages marked as **“skipped”**. This automatic optimization can save time in a workload that runs multiple jobs over the same data, but of course, for even **better performance you can perform your own caching with the DataFrame or RDD cache method,** which lets you control exactly which data is saved and where. You’ll quickly grow accustomed to this behavior after you run some Spark actions on aggregated data and inspect them in the UI.



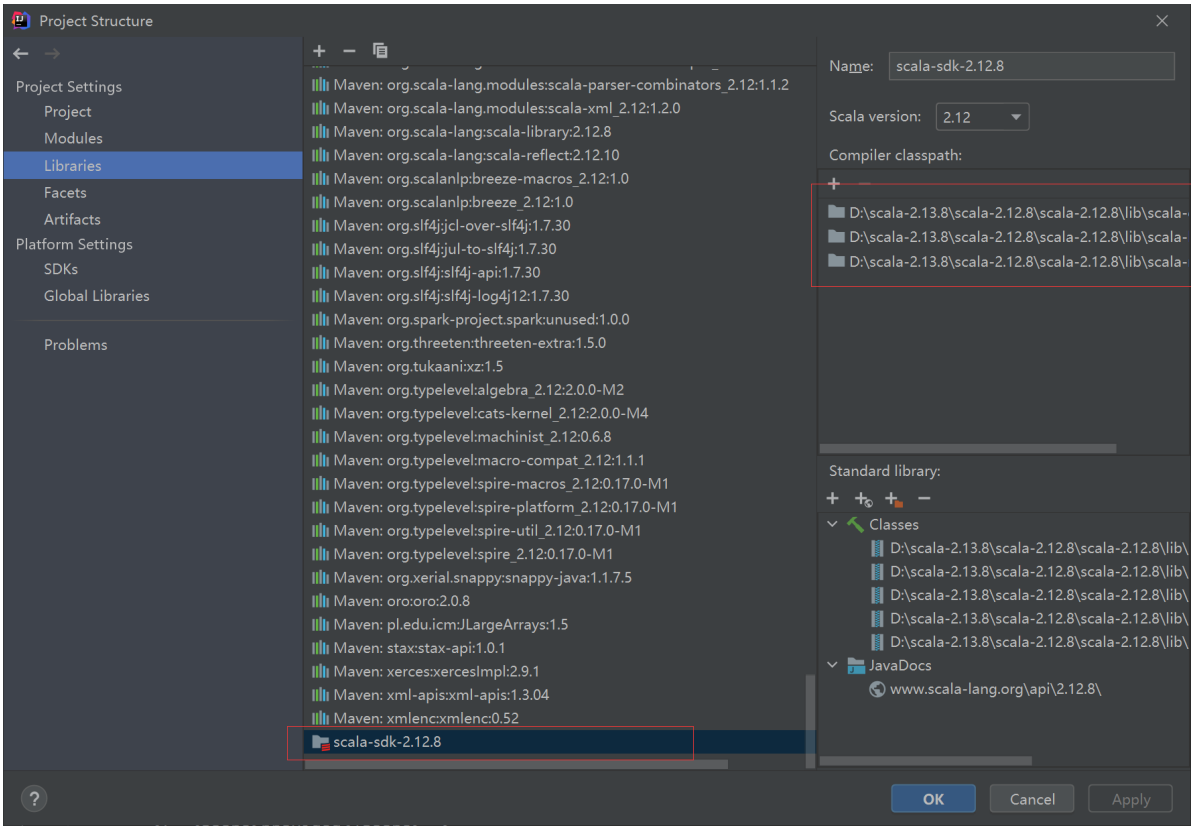

```
step3.cache()
step3.select(expr("avg(id)")).collect()
step3.select(expr("sum(id)")).collect()
```

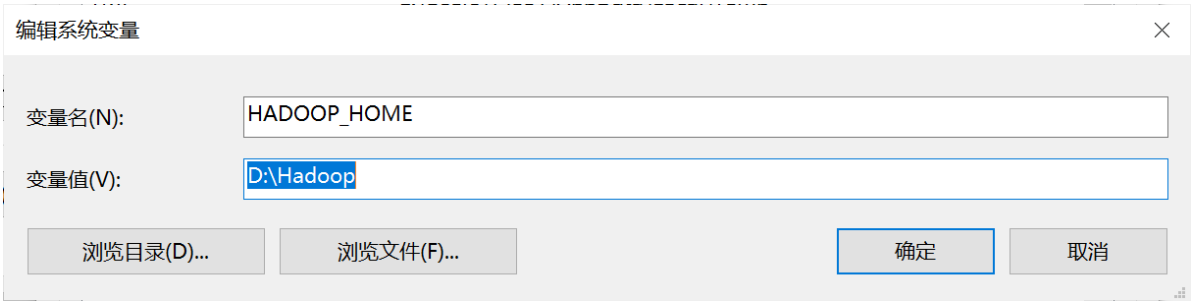
Developing Spark Applications

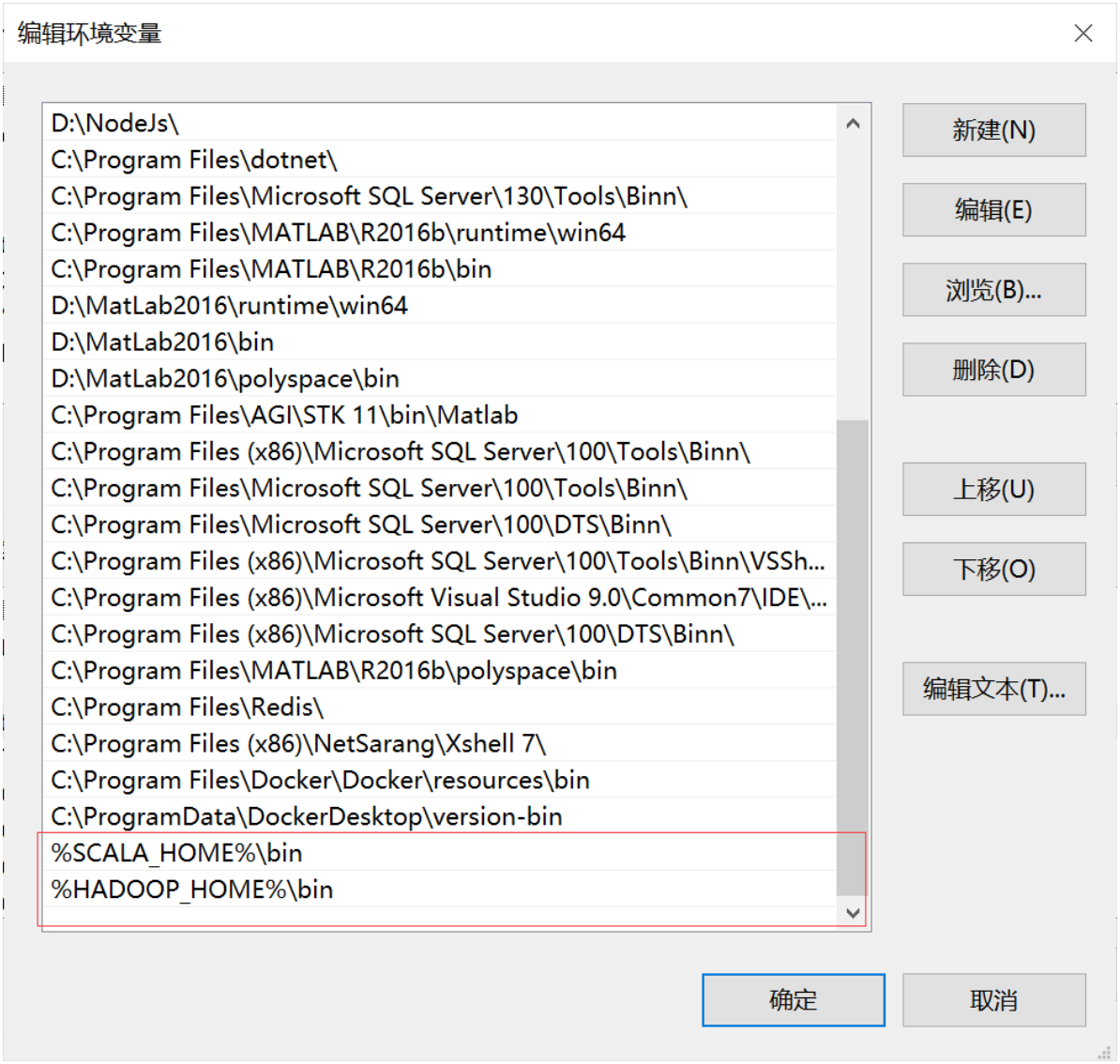
Writing Spark Applications

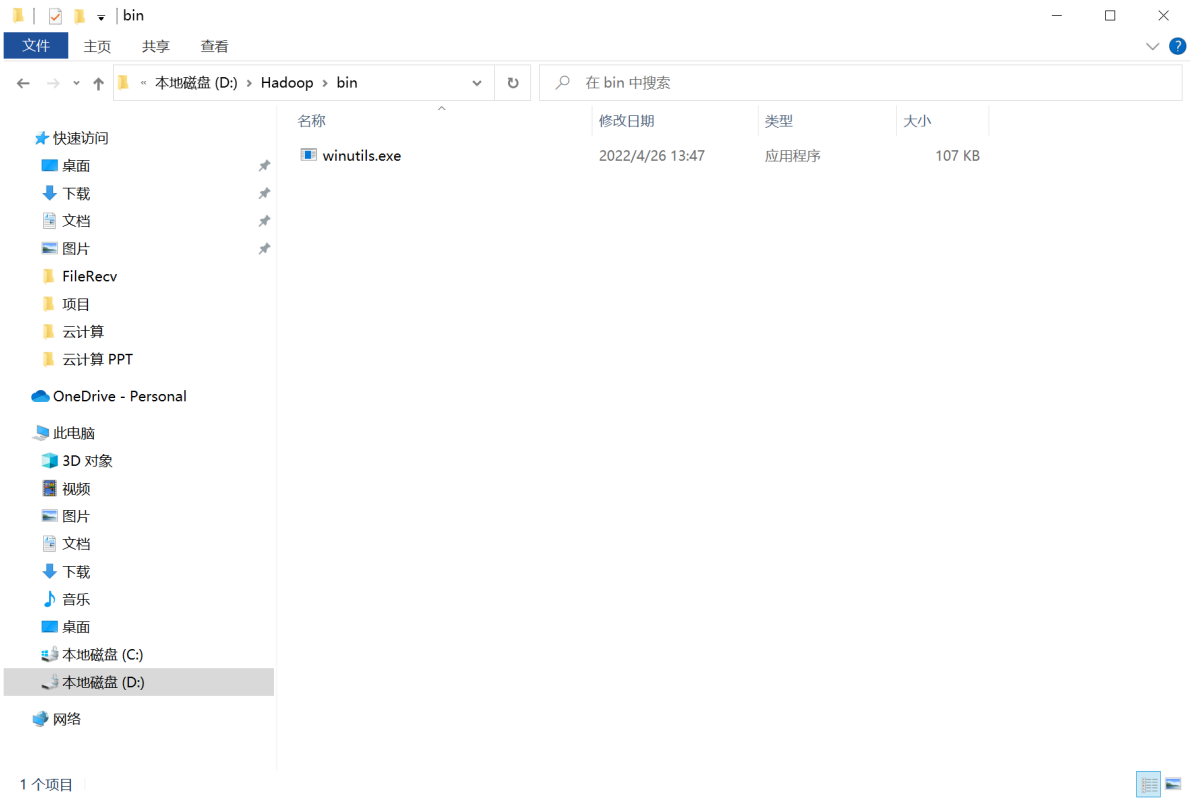
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>org.example</groupId>
    <artifactId>Spark_Project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>

        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-library</artifactId>
            <version>2.12.8</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.12</artifactId>
            <version>3.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.12</artifactId>
            <version>3.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-hive_2.12</artifactId>
            <version>3.0.0</version>
            <!--<scope>provided</scope>-->
        </dependency>
        <!-- scope 不能使用provide scope=provided时，引用的artifact只在编译、测试阶段被加载，而在运行阶段，程序会认为容器中已经提供了这个artifact的jar包，所以程序就会抛出ClassNotFoundException异常。
而scope=compile，则我们依赖的artifact会在编译、测试、运行阶段都被加载到容器中。-->
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-mllib_2.12</artifactId>
            <version>3.0.0</version>
            <scope>compile</scope>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

</project>
```

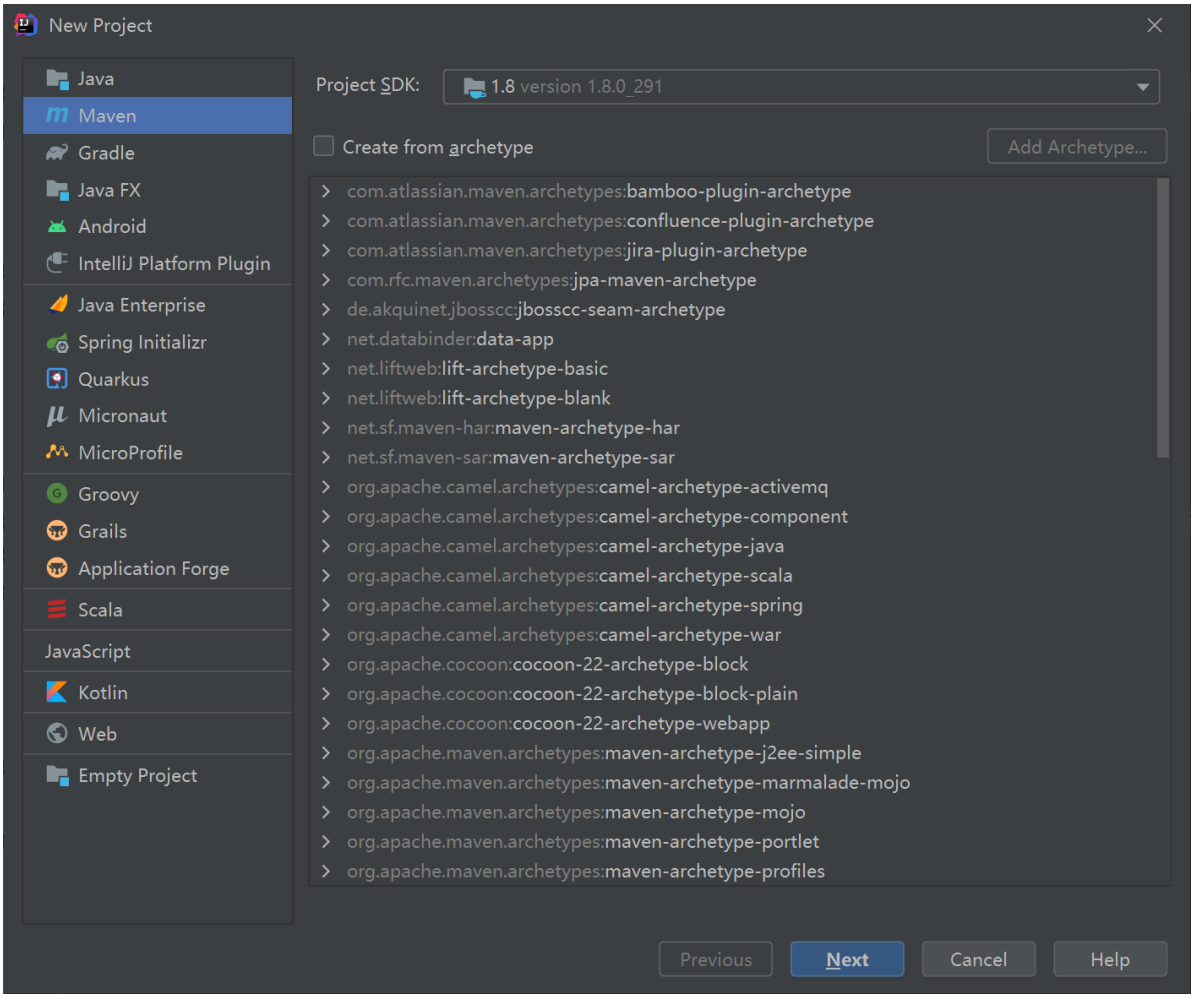

Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Maven: org.scala-lang.modules:scala-parser-combinators_2.12:1.1.2
Maven: org.scala-lang.modules:scala-xml_2.12:1.2.0
Maven: org.scala-lang:scala-library:2.12.8
Maven: org.scala-lang:scala-reflect:2.12.10
Maven: org.scalanlp:breeze-macros_2.12:1.0
Maven: org.scalanlp:breeze_2.12:1.0
Maven: org.slf4j:jcl-over-slf4j:1.7.30
Maven: org.slf4j:jul-to-slf4j:1.7.30
Maven: org.slf4j:slf4j-api:1.7.30
Maven: org.slf4j:slf4j-log4j12:1.7.30
Maven: org.spark-project.spark:unused:1.0.0
Maven: org.threeten:threeten-extra:1.5.0
Maven: org.tukaani:xz:1.5
Maven: org.typelevel:algebra_2.12:2.0.0-M2
Maven: org.typelevel:cats-kernel_2.12:2.0.0-M4
Maven: org.typelevel:machinist_2.12:0.6.8
Maven: org.typelevel:macro-compat_2.12:1.1.1
Maven: org.typelevel:spire-macros_2.12:0.17.0-M1
Maven: org.typelevel:spire-platform_2.12:0.17.0-M1
Maven: org.typelevel:spire-util_2.12:0.17.0-M1
Maven: org.typelevel:spire_2.12:0.17.0-M1
Maven: org.xerial.snappy:snappy-java:1.1.7.5
Maven: oro:oro:2.0.8
Maven: pl.edu.icm:JLargeArrays:1.5
Maven: stax:stax-api:1.0.1
Maven: xerces:xercesImpl:2.9.1
Maven: xml-apis:xml-apis:1.3.04
Maven: xmlenc:xmlenc:0.52
scala-sdk-2.12.8
Name: scala-sdk-2.12.8
Scala version: 2.12
Compiler classpath:
D:\scala-2.13.8\scala-2.12.8\scala-2.12.8\lib\scala-
D:\scala-2.13.8\scala-2.12.8\scala-2.12.8\lib\scala-
D:\scala-2.13.8\scala-2.12.8\scala-2.12.8\lib\scala-
Standard library:
Classes
D:\scala-2.13.8\scala-2.12.8\scala-2.12.8\lib\
D:\scala-2.13.8\scala-2.12.8\scala-2.12.8\lib\
D:\scala-2.13.8\scala-2.12.8\scala-2.12.8\lib\
D:\scala-2.13.8\scala-2.12.8\scala-2.12.8\lib\
D:\scala-2.13.8\scala-2.12.8\scala-2.12.8\lib\
JavaDocs
www.scala-lang.org\api\2.12.8\
OK
Cancel
Apply

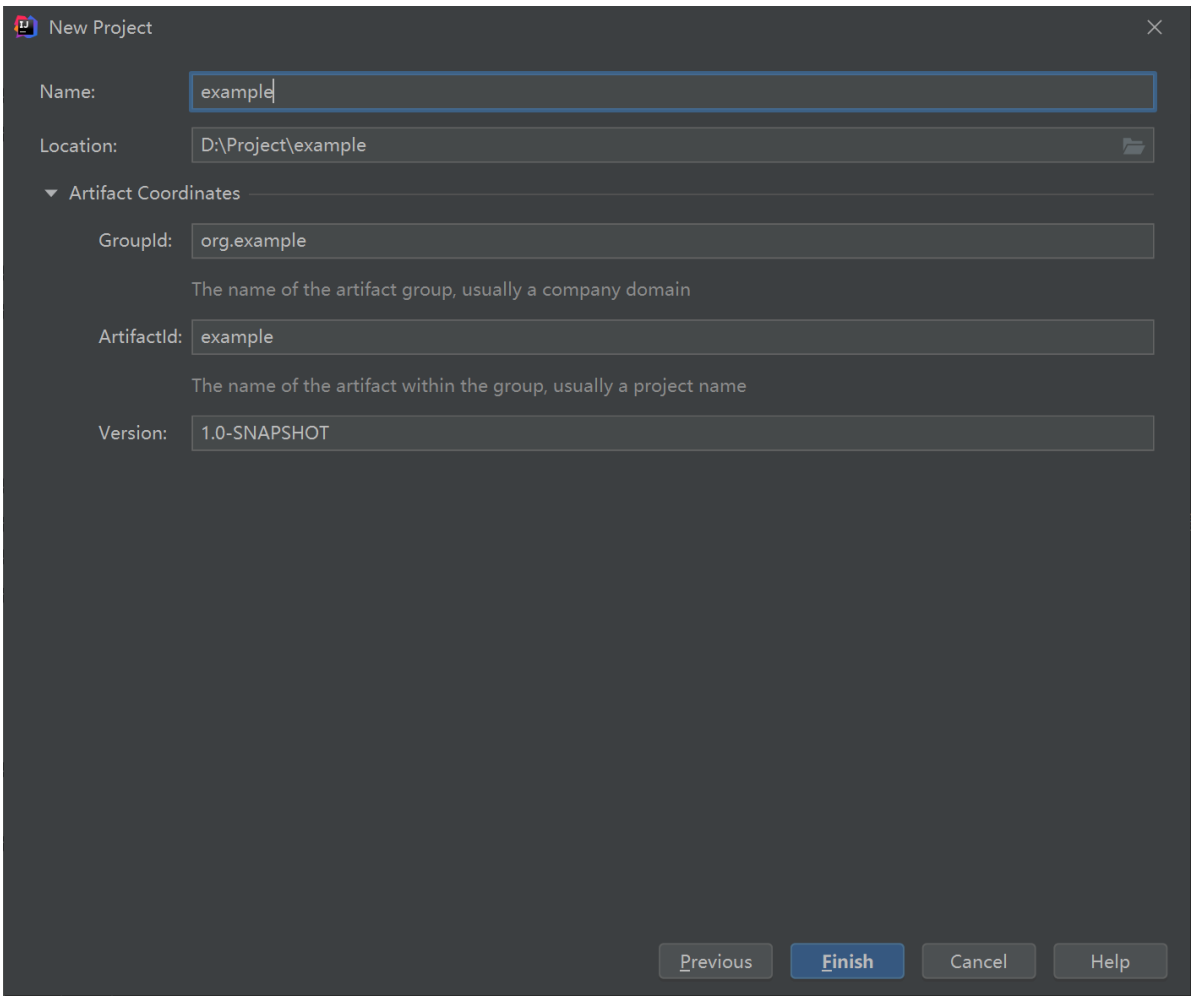

编辑系统变量
变量名(N):
HADOOP_HOME
变量值(V):
D:\Hadoop
浏览目录(D)...
浏览文件(F)...
确定
取消

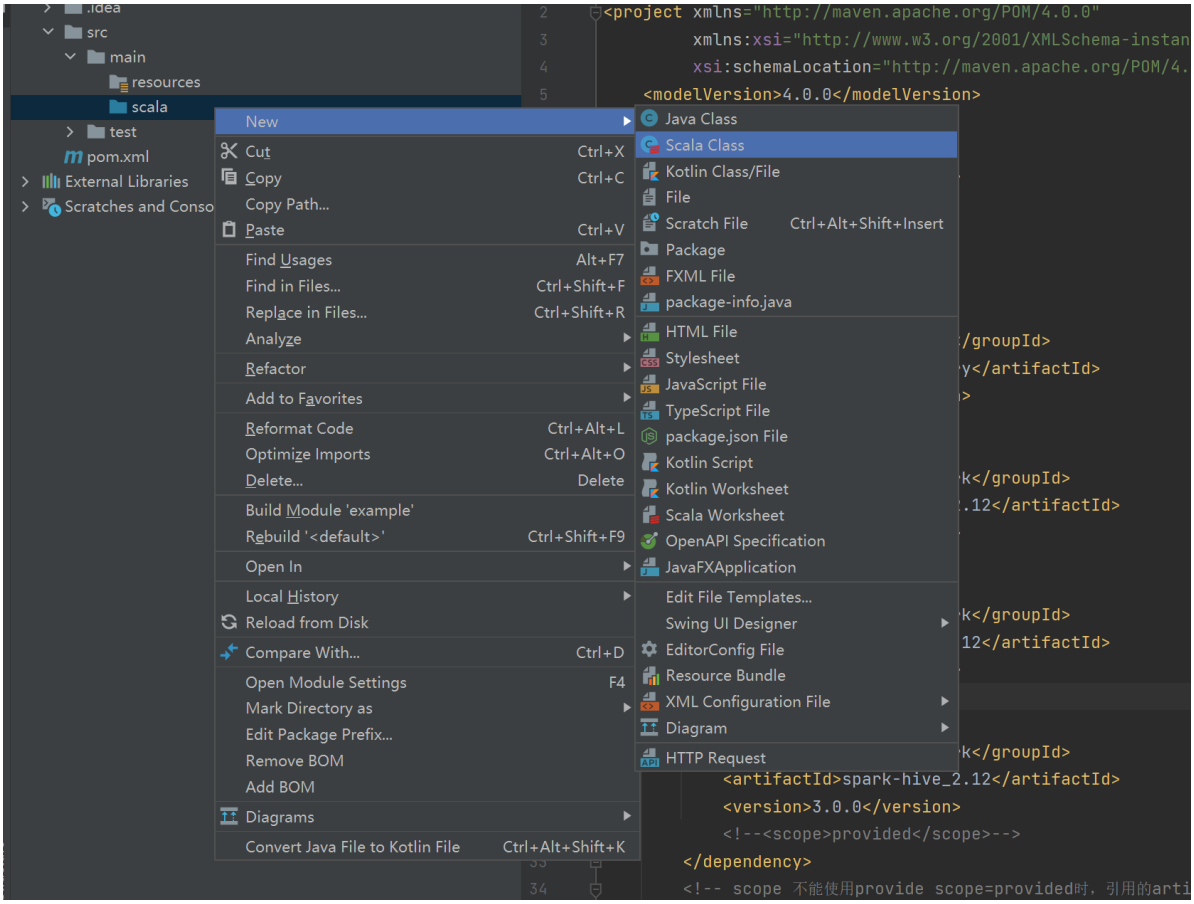


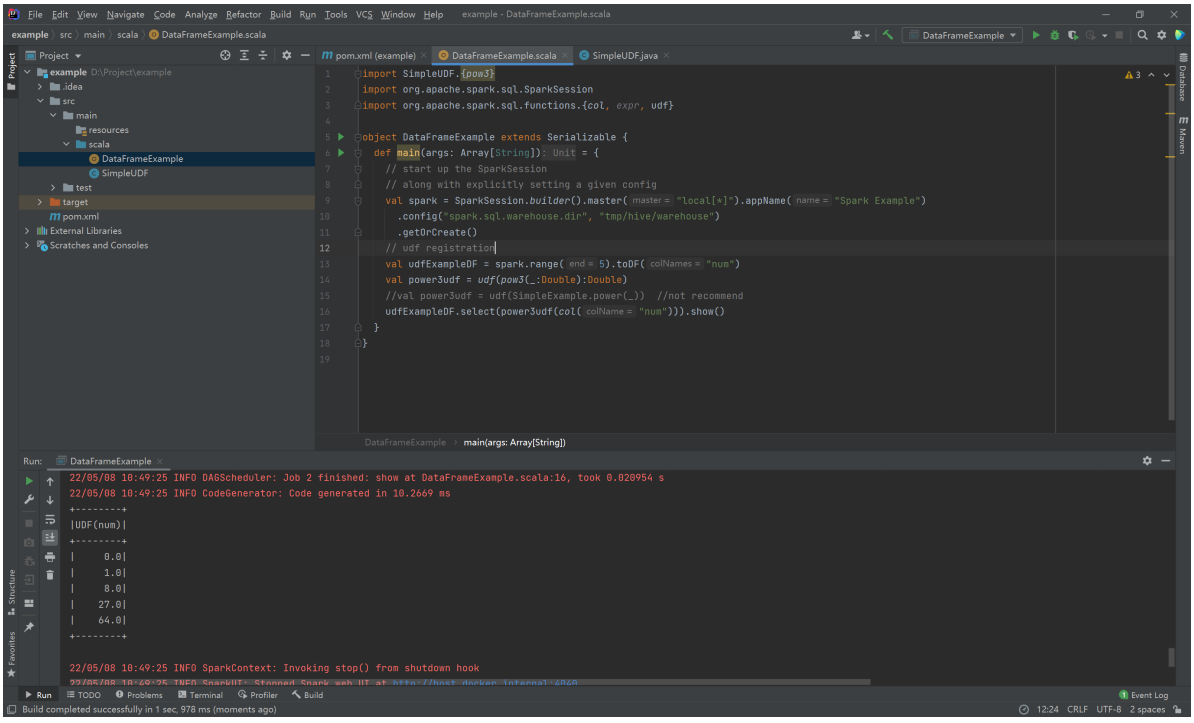


write an example


New Project
Java
Maven
Gradle
Java FX
Android
IntelliJ Platform Plugin
Java Enterprise
Spring Initializr
Quarkus
Micronaut
MicroProfile
Groovy
Grails
Application Forge
Scala
JavaScript
Kotlin
Web
Empty Project
Project SDK:
1.8 version 1.8.0_291
Create from archetype
Add Archetype...
com.atlassian.maven.archetypes:bamboo-plugin-archetype
com.atlassian.maven.archetypes:confluence-plugin-archetype
com.atlassian.maven.archetypes:jira-plugin-archetype
com.rfc.maven.archetypes:jpa-maven-archetype
de.akquinet.jbosscc:jbosscc-seam-archetype
net.databinder:data-app
net.liftweb:lift-archetype-basic
net.liftweb:lift-archetype-blank
net.sf.maven-har:maven-archetype-har
net.sf.maven-sar:maven-archetype-sar
org.apache.camel.archetypes:camel-archetype-activemq
org.apache.camel.archetypes:camel-archetype-component
org.apache.camel.archetypes:camel-archetype-java
org.apache.camel.archetypes:camel-archetype-scala
org.apache.camel.archetypes:camel-archetype-spring
org.apache.camel.archetypes:camel-archetype-war
org.apache.cocoon:cocoon-22-archetype-block
org.apache.cocoon:cocoon-22-archetype-block-plain
org.apache.cocoon:cocoon-22-archetype-webapp
org.apache.maven.archetypes:maven-archetype-j2ee-simple
org.apache.maven.archetypes:maven-archetype-marmalade-mojo
org.apache.maven.archetypes:maven-archetype-mojo
org.apache.maven.archetypes:maven-archetype-portlet
org.apache.maven.archetypes:maven-archetype-profiles
Previous
Next
Cancel
Help


New Project
Name:
example
Location:
D:\Project\example
Artifact Coordinates
GroupId:
org.example
The name of the artifact group, usually a company domain
ArtifactId:
example
The name of the artifact within the group, usually a project name
Version:
1.0-SNAPSHOT
Previous
Finish
Cancel
Help

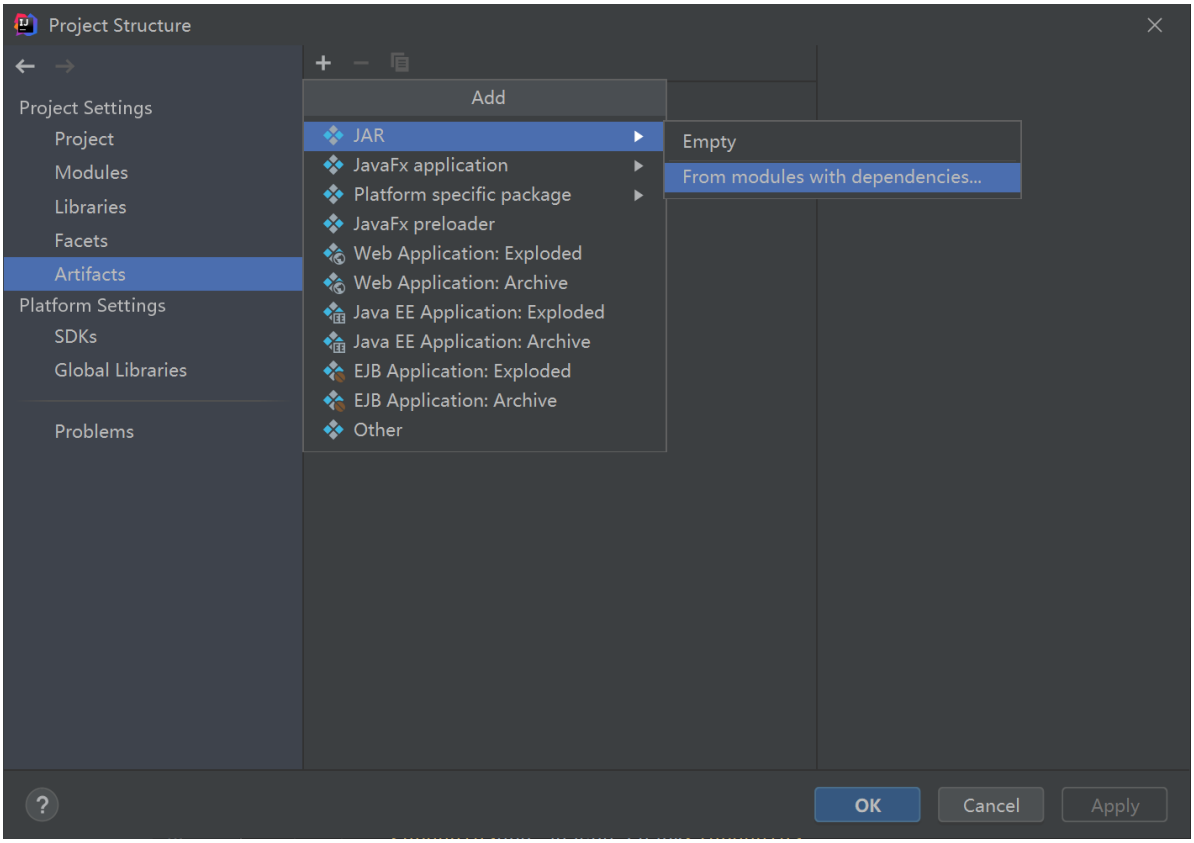


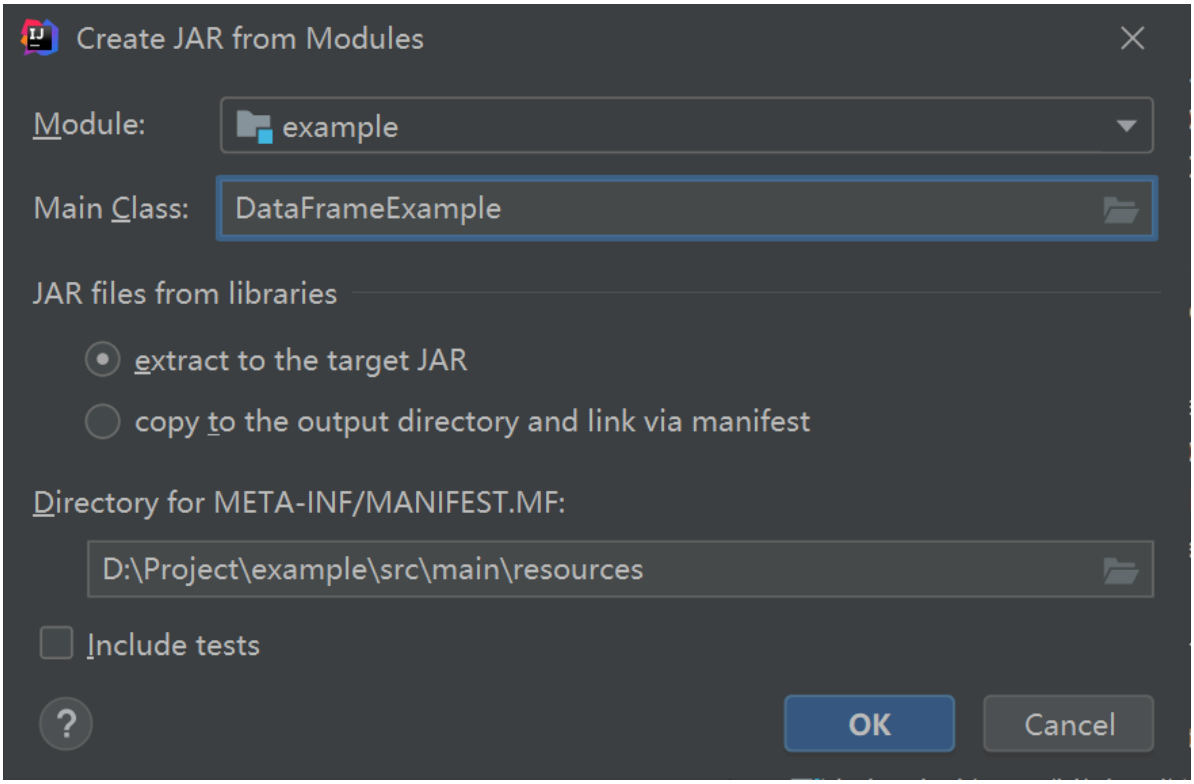


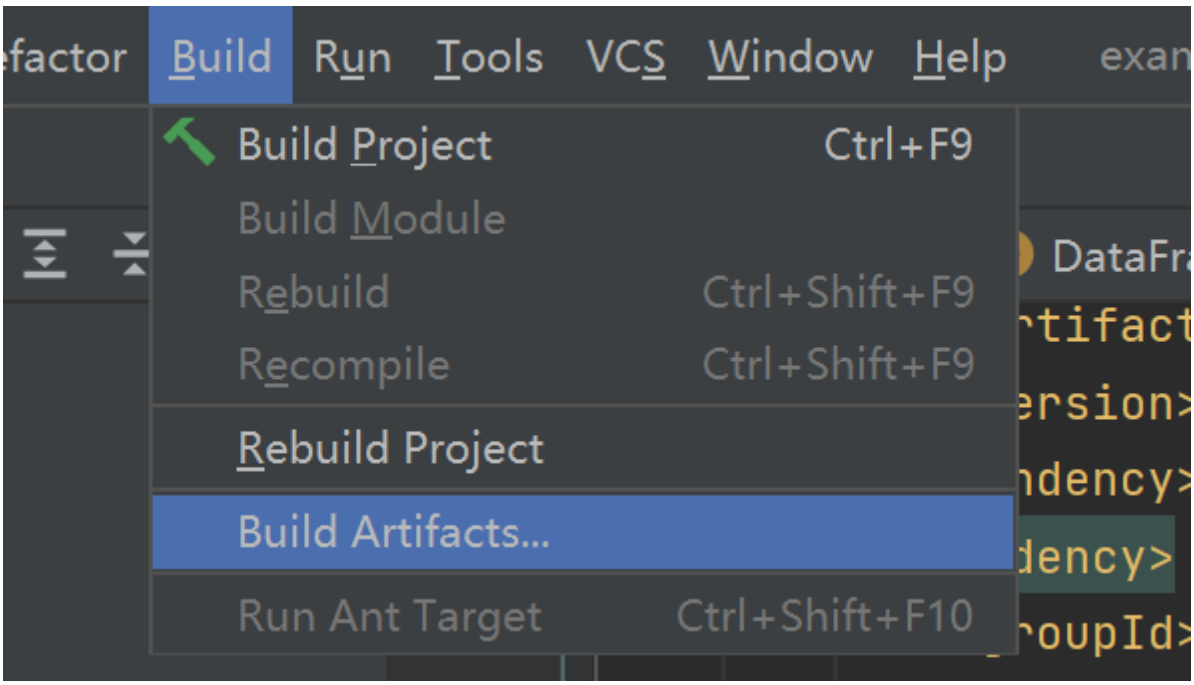

```
import SimpleUDF.{pow3}
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{col, expr, udf}

object DataFrameExample extends Serializable {
  def main(args: Array[String]) = {
    // start up the SparkSession
    // along with explicitly setting a given config
    val spark = SparkSession.builder().master("local[*]").appName("Spark
Example")
      .config("spark.sql.warehouse.dir", "tmp/hive/warehouse")
```

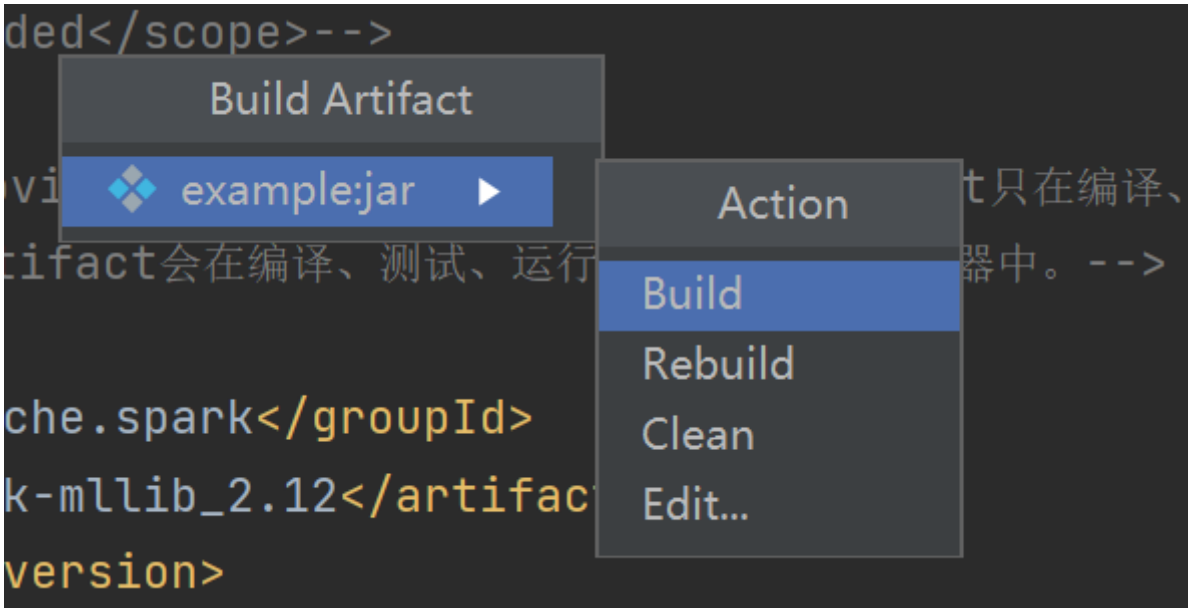
```
      .getOrCreate()
    // udf registration
    val udfExampleDF = spark.range(5).toDF("num")
    val power3udf = udf(pow3(_:Double):Double)
    //val power3udf = udf(SimpleExample.power(_))  //not recommend
    udfExampleDF.select(power3udf(col("num"))).show()
  }
}
```

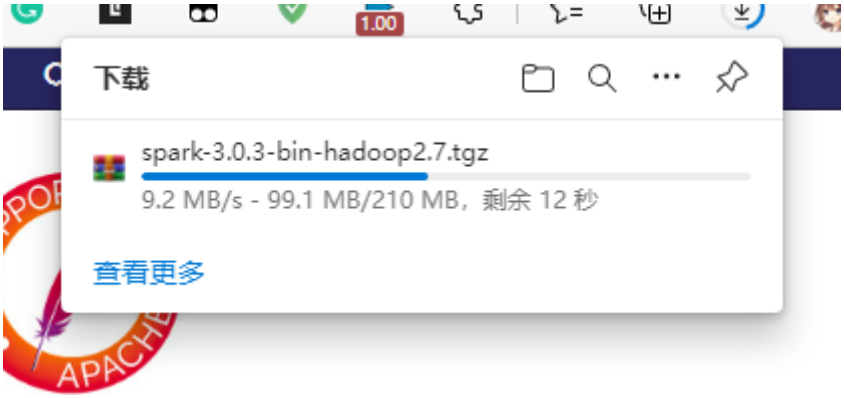
```
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.types.ByteType;
import org.apache.spark.sql.types.DataTypes;

/**
 * @ClassName SimpleUDF
 * @Author chenjia
 * @Date 2022/5/8 10:41
 * @Version
 */

public class SimpleUDF {

    public static double pow3(double a){
        return a * a * a;
    }
}
```




Create JAR from Modules
Module:
example
Main Class:
DataFrameExample
JAR files from libraries
extract to the target JAR
copy to the output directory and link via manifest
Directory for META-INF/MANIFEST.MF:
D:\Project\example\src\main\resources
Include tests
OK
Cancel

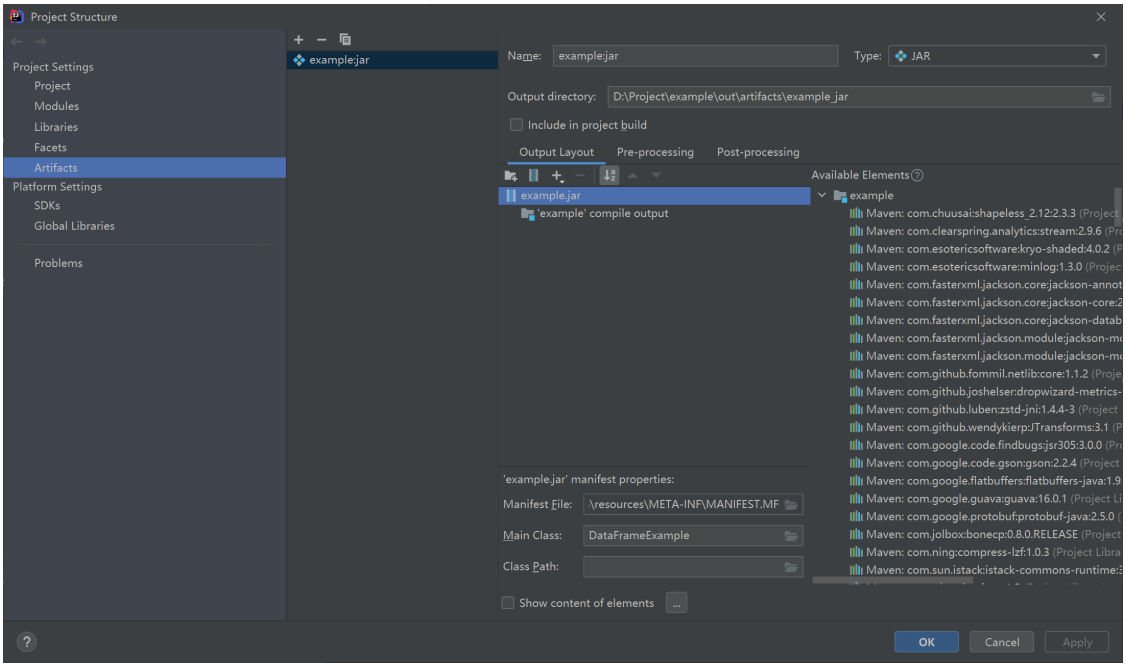

Build
Run
Tools
VCS
Window
Help
Build Project Ctrl+F9
Build Module
Rebuild Ctrl+Shift+F9
Recompile Ctrl+Shift+F9
Rebuild Project
Build Artifacts...
Run Ant Target Ctrl+Shift+F10


Build Artifact
example:jar
Action
Build
Rebuild
Clean
Edit...




```
D:\Edge\spark-3.0.3-bin-hadoop3.2\spark-3.0.3-bin-hadoop3.2\bin>spark-submit --class DataFrameExample example.jar
2/05/08 11:31:44 INFO TaskSetManager: Finished task 0.0 in stage 2.0 (TID 5) in 14 ms on host.docker.internal (executor driver)
2/05/08 11:31:44 INFO TaskSetManager: Finished task 1.0 in stage 2.0 (TID 6) in 15 ms on host.docker.internal (executor driver)
2/05/08 11:31:44 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from pool
2/05/08 11:31:44 INFO DAGScheduler: ResultStage 2 (show at DataFrameExample.scala:16) finished in 0.030 s
2/05/08 11:31:44 INFO DAGScheduler: Job 2 is finished. Cancelling potential speculative or zombie tasks for this job
2/05/08 11:31:44 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
2/05/08 11:31:44 INFO DAGScheduler: Job 2 finished: show at DataFrameExample.scala:16, took 0.035839 s
2/05/08 11:31:44 INFO CodeGenerator: Code generated in 10.4377 ms
--------+
UDF(num)|
--------+
     0.0|
     1.0|
     8.0|
    27.0|
    64.0|
--------+

2/05/08 11:31:44 INFO SparkContext: Invoking stop() from shutdown hook
2/05/08 11:31:44 INFO SparkUI: Stopped Spark web UI at http://host.docker.internal:4040
2/05/08 11:31:44 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
2/05/08 11:31:44 INFO MemoryStore: MemoryStore cleared
2/05/08 11:31:44 INFO BlockManager: BlockManager stopped
2/05/08 11:31:44 INFO BlockManagerMaster: BlockManagerMaster stopped
2/05/08 11:31:44 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
2/05/08 11:31:44 INFO SparkContext: Successfully stopped SparkContext
```

> Notice, remember to clean the output layout before build artifacts



Testing Spark Applications

Testing Spark Applications relies **on a couple of key principles and tactics** that you should keep in mind as you're writing your applications.

Strategic Principles

Input data resilience

Your Spark Applications and pipelines should be **resilient** to at least **some degree of change in the input data** or otherwise **ensure that these failures are handled in a graceful and resilient way.**

Business logic resilience and evolution

You'll need to do robust logical testing with realistic data to ensure that you're actually getting what you want out of it.

One thing to be wary of here is trying to **write a bunch of "Spark Unit Tests" that just test Spark's functionality.**

Resilience in output and atomicity

This means you will need to gracefully handle **output schema resolution.**

It's not often that data is simply dumped in some location, never to be read again—**most of your Spark pipelines are probably feeding other Spark pipelines.**

you're going to want to make certain that your downstream consumers understand the "state" of the data—this could mean how **frequently it's updated** as well as **whether the data is "complete"** (e.g., there is no late data) or that there won't be any last-minute corrections to the data

> 总结
>
> 保证输入数据的可靠性（对不同输入有正确的返回），逻辑正确性（单元测试），输出数据的可读性和完整性
>
> **Resilience**

Tactical Takeaways

Managing SparkSessions

Testing your Spark code using a unit test framework like **JUnit** or **ScalaTest** is relatively easy.

Initialize the SparkSession only once and pass it around to relevant functions and classes at runtime in a way that makes it easy to substitute during testing.

Which Spark API to Use?

Spark offers several choices of APIs, ranging from **SQL to DataFrames and Datasets**, and each of these can have different impacts for maintainability and testability of your application.

In general, we recommend documenting and testing the input and output types of each function regardless of which API you use.

A similar set of considerations applies to which programming language to use for your application: **there certainly is no right answer for every team, but depending on your needs, each language will provide different benefits.**

Connecting to Unit Testing Frameworks

To unit test your code, we recommend **using the standard frameworks** in your langage (e.g., JUnit or ScalaTest), and setting up your test harnesses to **create and clean up a SparkSession for each test**. Different frameworks offer different mechanisms to do this, such as “before” and “after” methods.

Connecting to Data Sources

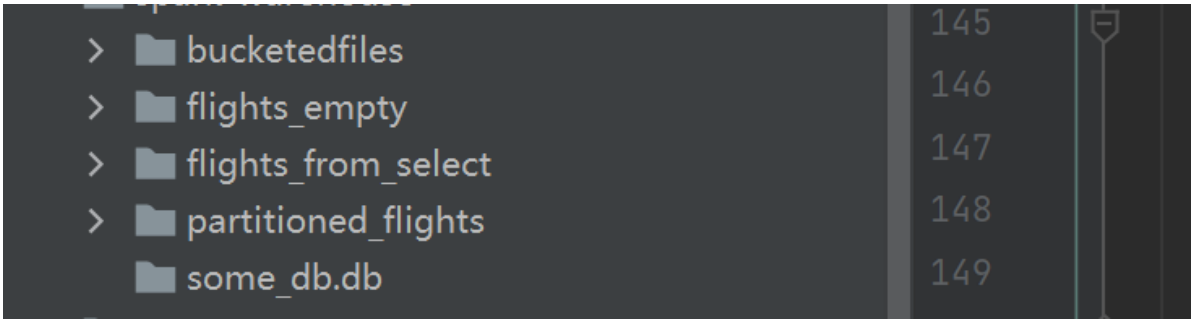
As much as possible, you should make sure your testing code **does not connect to production data sources,** so that developers can easily run it in isolation if these data sources change.

One easy way to make this happen is to **have all your business logic functions take DataFrames or Datasets as input** instead of directly connecting to various sources

```
def countByKey(KVcharacters:  RDD[(Char, Int)]): Unit ={...}
```

```
def getMarkets(source: DataFrame): Unit ={...}
```

If you are using the structured APIs in Spark, another way to make this happen is named tables: you can simply **register some dummy datasets** as various table names and go from there.



The Development Process

First, you might maintain a scratch space, such as an interactive notebook or some equivalent thereof, and then as you build key components and algorithms, you move them to a more permanent location like a library or package.

When running on your local machine, the **spark-shell** and its various language-specific implementations are probably the best way to develop applications.

The notebook experience is one that we often recommend (and are using to write this book) because of its simplicity in experimentation. There are also some tools, such as Databricks, that allow you to run notebooks as production applications as well.

> 不清楚scala是否有像jupyter一样的东西，用jupyter进行学习应该是最方便的

Launching Applications

The most common way for running Spark Applications is through **spark-submit.**

Table 16-1. Spark submit help text

Parameter	Description
`--master MASTER_URL`	spark://host:port, mesos://host:port, yarn, or local
`--deploy-mode DEPLOY_MODE`	Whether to launch the driver program locally (“client”) or on one of the worker machines inside the cluster (“cluster”) (Default: client)
`--class CLASS_NAME`	Your application’s main class (for Java / Scala apps).
`--name NAME`	A name of your application.
`--jars JARS`	Comma-separated list of local JARs to include on the driver and executor classpaths.
`--packages`	Comma-separated list of Maven coordinates of JARs to include on the driver and executor classpaths. Will search the local Maven repo, then Maven Central and any additional remote repositories given by `--repositories`. The format for the coordinates should be `groupId:artifactId:version`.
`--exclude-packages`	Comma-separated list of `groupId:artifactId`, to exclude while resolving the dependencies provided in `--packages` to avoid dependency conflicts.
`--repositories`	Comma-separated list of additional remote repositories to search for the Maven coordinates given with `--packages`.
`--py-files PY_FILES`	Comma-separated list of *.zip*, *.egg*, or *.py* files to place on the `PYTHONPATH` for Python apps.
`--files FILES`	Comma-separated list of files to be placed in the working directory of each executor.
`--conf PROP=VALUE`	Arbitrary Spark configuration property.
`--properties-file FILE`	Path to a file from which to load extra properties. If not specified, this will look for `conf/spark-defaults.conf`.
`--driver-memory MEM`	Memory for driver (e.g., 1000M, 2G) (Default: 1024M).
`--driver-java-options`	Extra Java options to pass to the driver.
`--driver-class-path`	Extra class path entries to pass to the driver. Note that JARs added with `--jars` are automatically included in the classpath.
`--executor-memory MEM`	Memory per executor (e.g., 1000M, 2G) (Default: 1G).
`--proxy-user NAME`	User to impersonate when submitting the application. This argument does not work with `--principal / --keytab`.
`--help, -h`	Show this help message and exit.
`--verbose, -v`	Print additional debug output.
`--version`	Print the version of current Spark.

There are some deployment-specific configurations

Table 16-2. Deployment Specific Configurations

Cluster Managers	Modes	Conf	Description
Standalone	Cluster	`--driver-cores NUM`	Cores for driver (Default: 1).
Standalone/Mesos	Cluster	`--supervise`	If given, restarts the driver on failure.
Standalone/Mesos	Cluster	`--kill SUBMISSION_ID`	If given, kills the driver specified.
Standalone/Mesos	Cluster	`--status SUBMISSION_ID`	If given, requests the status of the driver specified.
Standalone/Mesos	Either	`--total-executor-cores NUM`	Total cores for all executors.
Standalone/YARN	Either	`--executor-cores NUM1`	Number of cores per executor. (Default: 1 in YARN mode or all available cores on the worker in standalone mode)
YARN	Either	`--driver-cores NUM`	Number of cores used by the driver, only in cluster mode (Default: 1).
YARN	Either	`queue QUEUE_NAME`	The YARN queue to submit to (Default: “default”).
YARN	Either	`--num-executors NUM`	Number of executors to launch (Default: 2). If dynamic allocation is enabled, the initial number of executors will be at least `NUM`.
YARN	Either	`--archives ARCHIVES`	Comma-separated list of archives to be extracted into the working directory of each executor.
YARN	Either	`--principal PRINCIPAL`	Principal to be used to log in to KDC, while running on secure HDFS.
YARN	Either	`--keytab KEYTAB`	The full path to the file that contains the keytab for the principal specified above. This keytab will be copied to the node running the Application Master via the Secure Distributed Cache, for renewing the login tickets and the delegation tokens periodically.

Application Launch Examples

```
./bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://207.184.161.138:7077 \
--executor-memory 20G \
--total-executor-cores 100 \
replace/with/path/to/examples.jar \
1000
```

Configuring Applications

The majority of configurations fall into the following categories:

- Application properties
- Runtime environment
- Shuffle behavior
- Spark UI

- Compression and serialization
- Memory management
- Execution behavior
- Networking
- Scheduling
- Dynamic allocation
- Security
- Encryption
- Spark SQL
- Spark streaming
- SparkR

Spark provides **three locations to configure the system:**

- Spark properties control most application parameters and can be set by using a **SparkConf object**
- Java system properties
- Hardcoded configuration files

There are several templates that you can use, which you can find in the **/conf directory** available in the root of the Spark home folder. You can set these properties as hardcoded variables in your applications or by specifying them at runtime.You can use environment variables to set per-machine settings, such as the IP address, through the **conf/spark-env.sh** script on each node. Lastly, you can configure logging through **log4j.properties.**

The SparkConf

```
//After you create it, the SparkConf is immutable for that specific Spark
Application
import org.apache.spark.SparkConf
val conf = new SparkConf().setMaster("local[2]").setAppName("DefinitiveGuide")
.set("some.conf", "to.some.value")
```

You can configure these **at runtime**, as you saw previously in this chapter through **command-line arguments.**

```
./bin/spark-submit --name "DefinitiveGuide" --master local[4] ...
```

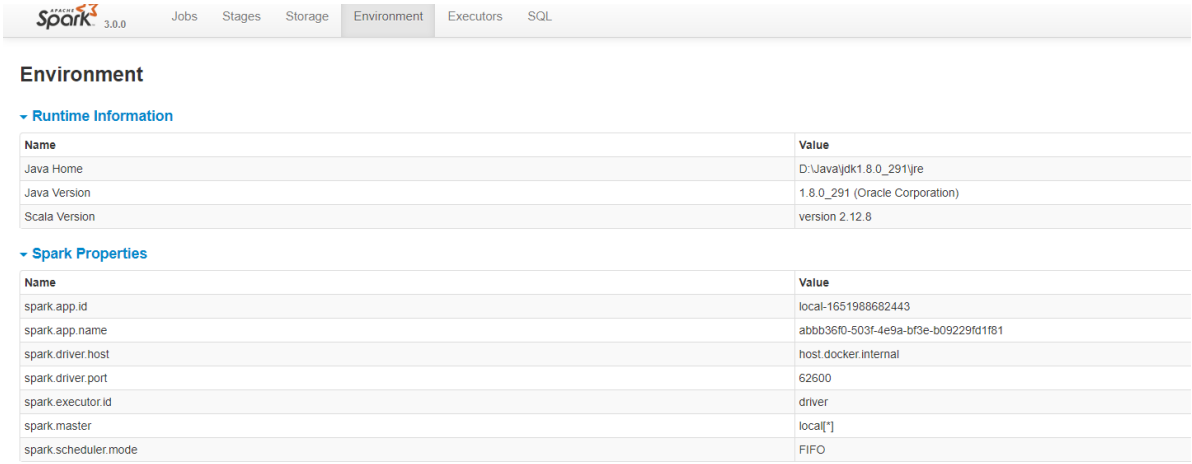
> Of note is that when setting time duration-based properties, you should use the following format:
>
> - 25ms (milliseconds)
> - 5s (seconds)
> - 10m or 10min (minutes)
> - 3h (hours)
> - 5d (days)
> - 1y (years)

Application Properties

Table 16-3. Application properties

Property name	Default	Meaning
`spark.app.name`	(none)	The name of your application. This will appear in the UI and in log data.
`spark.driver.cores`	`1`	Number of cores to use for the driver process, only in cluster mode.
`spark.driver.maxResultSize`	`1g`	Limit of total size of serialized results of all partitions for each Spark action (e.g., collect). Should be at least `1M`, or `0` for unlimited. Jobs will be aborted if the total size exceeds this limit. Having a high limit can cause `OutOfMemoryErrors` in the driver (depends on `spark.driver.memory` and memory overhead of objects in JVM). Setting a proper limit can protect the driver from `OutOfMemoryErrors`.
`spark.driver.memory`	`1g`	Amount of memory to use for the driver process, where `SparkContext` is initialized. (e.g. `1g`, `2g`). Note: in client mode, this must not be set through the `SparkConf` directly in your application, because the driver JVM has already started at that point. Instead, set this through the `--driver-memory` command-line option or in your default properties file.
`spark.executor.memory`	`1g`	Amount of memory to use per executor process (e.g., `2g`, `8g`).
`spark.extraListeners`	(none)	A comma-separated list of classes that implement `SparkListener`; when initializing `SparkContext`, instances of these classes will be created and registered with Spark's listener bus. If a class has a single-argument constructor that accepts a `SparkConf`, that constructor will be called; otherwise, a zero-argument constructor will be called. If no valid constructor can be found, the `SparkContext` creation will fail with an exception.
`spark.logConf`	`FALSE`	Logs the effective `SparkConf` as INFO when a `SparkContext` is started.
`spark.master`	(none)	The cluster manager to connect to. See the list of allowed master URLs.
`spark.submit.deployMode`	(none)	The deploy mode of the Spark driver program, either "client" or "cluster," which means to launch driver program locally ("client") or remotely ("cluster") on one of the nodes inside the cluster.
`spark.log.callerContext`	(none)	Application information that will be written into Yarn RM log/HDFS audit log when running on Yarn/HDFS. Its length depends on the Hadoop configuration `hadoop.caller.context.max`.size. It should be concise, and typically can have up to 50 characters.
`spark.driver.supervise`	`FALSE`	If true, restarts the driver automatically if it fails with a non-zero exit status. Only has effect in Spark standalone mode or Mesos cluster deploy mode.

You can ensure that you've correctly set these values by **checking the application's web UI on port 4040 of the driver on the "Environment" tab**.



Runtime Properties

You might also need to configure the runtime environment of your application.

These properties allow you to **configure extra classpaths and python paths for both drivers and executors**, Python worker configurations, as well as miscellaneous logging properties.

Execution Properties

These configurations are some of the most relevant for you to configure because they give you **finer-grained control on actual execution.**

The most common configurations to change are **spark.executor.cores** (to control the number of available cores) and **spark.files.maxPartitionBytes** (maximum partition size when reading files).

Configuring Memory Management

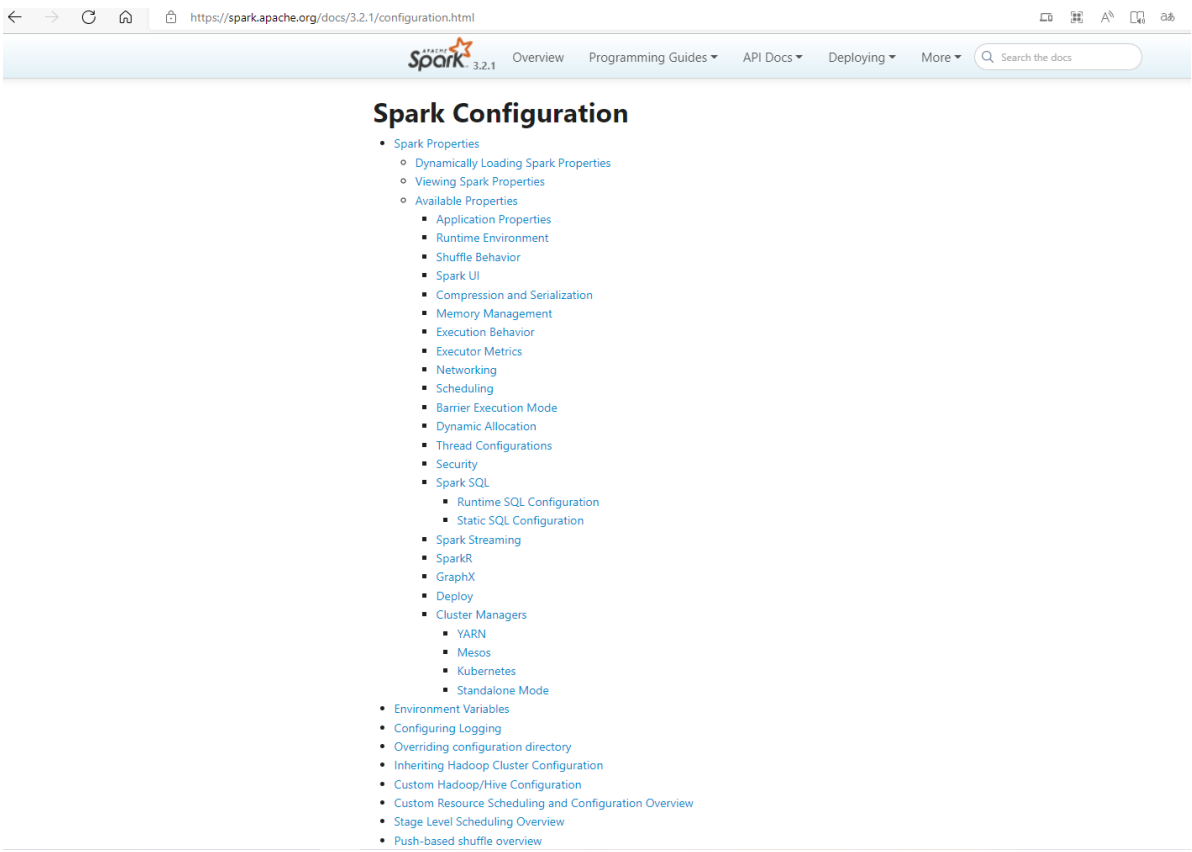
There are times when you might need to manually **manage the memory options to try and optimize your applications.**

Configuring Shuffle Behavior

Shuffles can be a bottleneck in Spark jobs because of their **high communication overhead**.

There are a number of low-level configurations for controlling shuffle behavior

Configuration - Spark 3.2.1 Documentation (apache.org)



Environmental Variables

You can configure certain Spark settings through environment variables, which are read from the **conf/spark-env.sh** script in the directory where Spark is installed.

```
[hadoop@hadoop01 conf]$ mv spark-env.sh.template spark-env.sh
[hadoop@hadoop01 conf]$ vi spark-env.sh
export JAVA_HOME=/opt/model/jdk1.8
export SCALA_HOME=/opt/model/scala-2.12.8
export HADOOP_HOME=/opt/model/hadoop-3.2.1
export HADOOP_CONF_DIR=/opt/model/hadoop-3.2.1/etc/hadoop
export SPARK_MASTER_IP=hadoop01 #Spark集群Master节点的IP
export SPARK_WORKER_MEMORY=4G #每个Worker能最大分配给Excecutors的内存
export SPARK_WORKER_CORES=2 #每个Worker节点占有的CPU核数
```

> When running Spark on YARN in cluster mode, you need to set environment variables by using the **spark.yarn.appMasterEnv.[EnvironmentVariableName]** property in your **conf/spark-defaults.conf file**. Environment variables that are **set in spark-env.sh will not be reflected in the YARN Application Master process in cluster mode**.

Job Scheduling Within an Application

Within a given Spark Application, **multiple parallel jobs** can run simultaneously if they were submitted from **separate threads.**

Spark's scheduler is fully **thread-safe** and supports this use case to **enable applications that serve multiple requests** (e.g., queries for multiple users).

By default, Spark's scheduler runs jobs in **FIFO fashion.** If the jobs at the head of the queue are large, later jobs might be **delayed significantly.**

Under **fair sharing**, Spark assigns tasks between jobs in a **round-robin fashion** so that all jobs get a roughly equal share of cluster resources. This means that short jobs submitted while a long job is running can begin receiving resources right away and still achieve good response times **without waiting for the long job to finish.**

To enable the fair scheduler, set the **spark.scheduler.mode** property to **FAIR** when configuring a SparkContext.

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.set("spark.scheduler.mode", "FAIR")
val sc = new SparkContext(conf)
```

```
val conf = new SparkConf()
conf.set("spark.scheduler.mode", "FAIR")
val spark = SparkSession.builder().config(conf).master("local[*]").getOrCreate()
```

▾ Spark Properties

Name	Value
spark.app.id	local-1651991445055
spark.app.name	ada08c81-1650-4f09-9694-5ea505a304e9
spark.driver.host	host.docker.internal
spark.driver.port	58496
spark.executor.id	driver
spark.master	local[*]
spark.scheduler.mode	FAIR

The fair scheduler also supports **grouping jobs into pools**, and setting different scheduling options, or weights, for each pool.

This can be useful to create a high-priority pool for more important jobs or to group the jobs of each user together and **give users equal shares** regardless of how many concurrent jobs they have **instead of giving jobs equal shares**. This approach is modeled after the Hadoop Fair Scheduler

Without any intervention, newly submitted jobs go into a default pool, but jobs pools can be set by adding the **spark.scheduler.pool** local property to the **SparkContext**

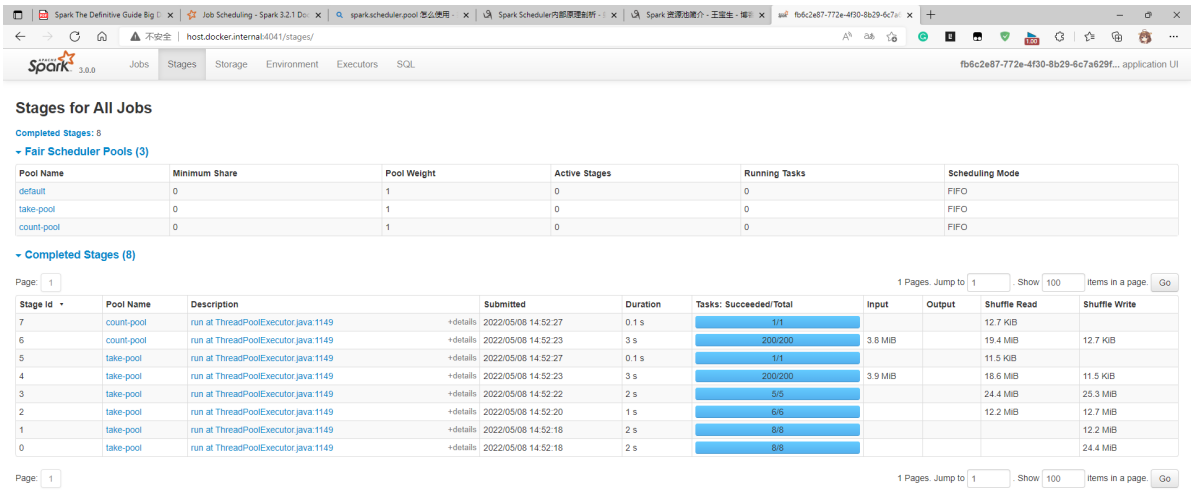
```
// Assuming sc is your SparkContext variable
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

After setting this local property, **all jobs submitted within this thread will use this pool name**. The setting is per-thread to make it easy to have a thread run multiple jobs on behalf of the same user.

If you'd like to clear the pool that a thread is associated with, set it to **null.**

```
object ThreadScheduleHandler {
  val conf = new SparkConf()
  conf.set("spark.scheduler.mode", "FAIR")
  val spark =
SparkSession.builder().config(conf).master("local[*]").getOrCreate()

  def main(args: Array[String]): Unit ={
    val df1 = spark.range(2, 10000000, 2)
    val df2 = spark.range(2, 10000000, 4)
    val step1 = df1.repartition(5)
    val step12 = df2.repartition(6)
    val step2 = step1.selectExpr("id * 5 as id2")
```

```
    val step3 = step2.join(step12, step2("id2") === step12("id"))
    val step4 = step3.select(expr("sum(id)"))
    step3.cache()

    val jobExecutor = Executors.newFixedThreadPool(2)

    jobExecutor.execute(new Runnable {
      override def run(): Unit = {
        spark.sparkContext.setLocalProperty("spark.scheduler.pool", "count-
pool")
        step3.select(expr("avg(id)")).collect()
      }
    })

    jobExecutor.execute(new Runnable {
      override def run(): Unit = {
        spark.sparkContext.setLocalProperty("spark.scheduler.pool", "take-pool")
        step3.select(expr("sum(id)")).collect()
      }
    })
    jobExecutor.shutdown()
    while (!jobExecutor.isTerminated) {}
    println("Done!")
  }
}
```



[Spark Scheduler内部原理剖析 - 虾皮 - 博客园 (cnblogs.com)]

Configuring Pool Properties

Specific pools' properties can also be modified through a configuration file. Each pool supports three properties:

- `schedulingMode`: This can be FIFO or FAIR, to control whether jobs within the pool queue up behind each other (the default) or share the pool's resources fairly.
- `weight`: This controls the pool's share of the cluster relative to other pools. By default, all pools have a weight of 1. If you give a specific pool a weight of 2, for example, it will get 2x more resources as other active pools. Setting a high weight such as 1000 also makes it possible to implement *priority* between pools—in essence, the weight-1000 pool will always get to launch tasks first whenever it has jobs active.
- `minShare`: Apart from an overall weight, each pool can be given a *minimum shares* (as a number of CPU cores) that the administrator would like it to have. The fair scheduler always attempts to meet all active pools' minimum shares before redistributing extra resources according to the weights. The `minShare` property can, therefore, be another way to ensure that a pool can always get up to a certain number of resources (e.g. 10 cores) quickly without giving it a high priority for the rest of the cluster. By default, each pool's `minShare` is 0.

The pool properties can be set by creating an XML file, similar to `conf/fairscheduler.xml.template`, and either putting a file named `fairscheduler.xml` on the classpath, or setting `spark.scheduler.allocation.file` property in your SparkConf. The file path respects the hadoop configuration and can either be a local file path or HDFS file path.

```
// scheduler file at local
conf.set("spark.scheduler.allocation.file", "file:///path/to/file")
// scheduler file at hdfs
conf.set("spark.scheduler.allocation.file", "hdfs:///path/to/file")
```

The format of the XML file is simply a `<pool>` element for each pool, with different elements within it for the various settings. For example:

```
<?xml version="1.0"?>
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>
```