
# Spark SQL

## Spark Structured API

# Spark SQL

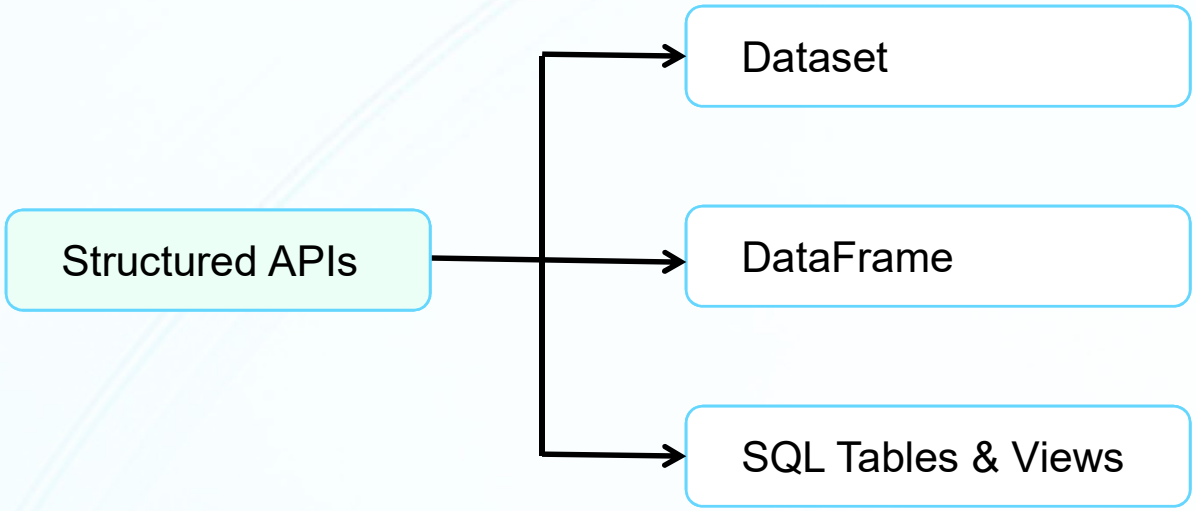
# Agenda

**In this module, we are going to look at the following topics:**

- ✓ Spark Structured APIs
- ✓ DataFrames
- ✓ Spark SQL Types
- ✓ Spark SQL Code Execution
- ✓ DataFrame Operations
- ✓ DataFrame Aggregation Functions
- ✓ Grouping, Windows & Joins
- ✓ Spark SQL Data Sources

# Spark Structured APIs

Spark Structured APIs refer to three core types of distributed collection APIs




These Structured APIs are the fundamental abstractions used to write the majority of our data flows.

# Dataset

- A Dataset is a distributed collection of data.
- Dataset provides the benefits of RDDs with the benefits of Spark SQL's optimized execution engine.
- A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.).
- The Dataset API is available in Scala and Java. Python does not have the support for the Dataset API.

# DataFrame

- DataFrame is a Dataset organized into named columns.
- It is conceptually equivalent to a table in RDBMS, but with richer optimizations under the hood.
- DataFrames can be constructed from a wide array of sources such as: structured data files, Hive tables, external databases, or existing RDDs.
- The DataFrame API is available in Scala, Java, Python, and R.
- In Scala and Java, a DataFrame is represented by a Dataset of `Row`s.

# DataFrames Vs. Datasets

- To Spark (in Scala & Java), DataFrames are 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 GC and object instantiation costs, Spark can operate on its own internal format without incurring any of those costs.
- Datasets are collection of JVM objects, hence they provide compile-time type-safety
- As DataFrames are represented in Spark's Row format, they don't provide compile-time type-safety. The data is validated against a schema only at run-time.

# Schema

- A schema defines the column names and types of a DataFrame. You can define schemas manually or read a schema from a data source.
- Schemas consist of types, meaning that you need a way of specifying what lies where.

# Columns & Rows

## Columns

- Columns represent a simple type like an integer or string, a complex type like an array or map, or a null value.
- Spark tracks all of this type information for you and offers a variety of ways, with which you can transform columns.

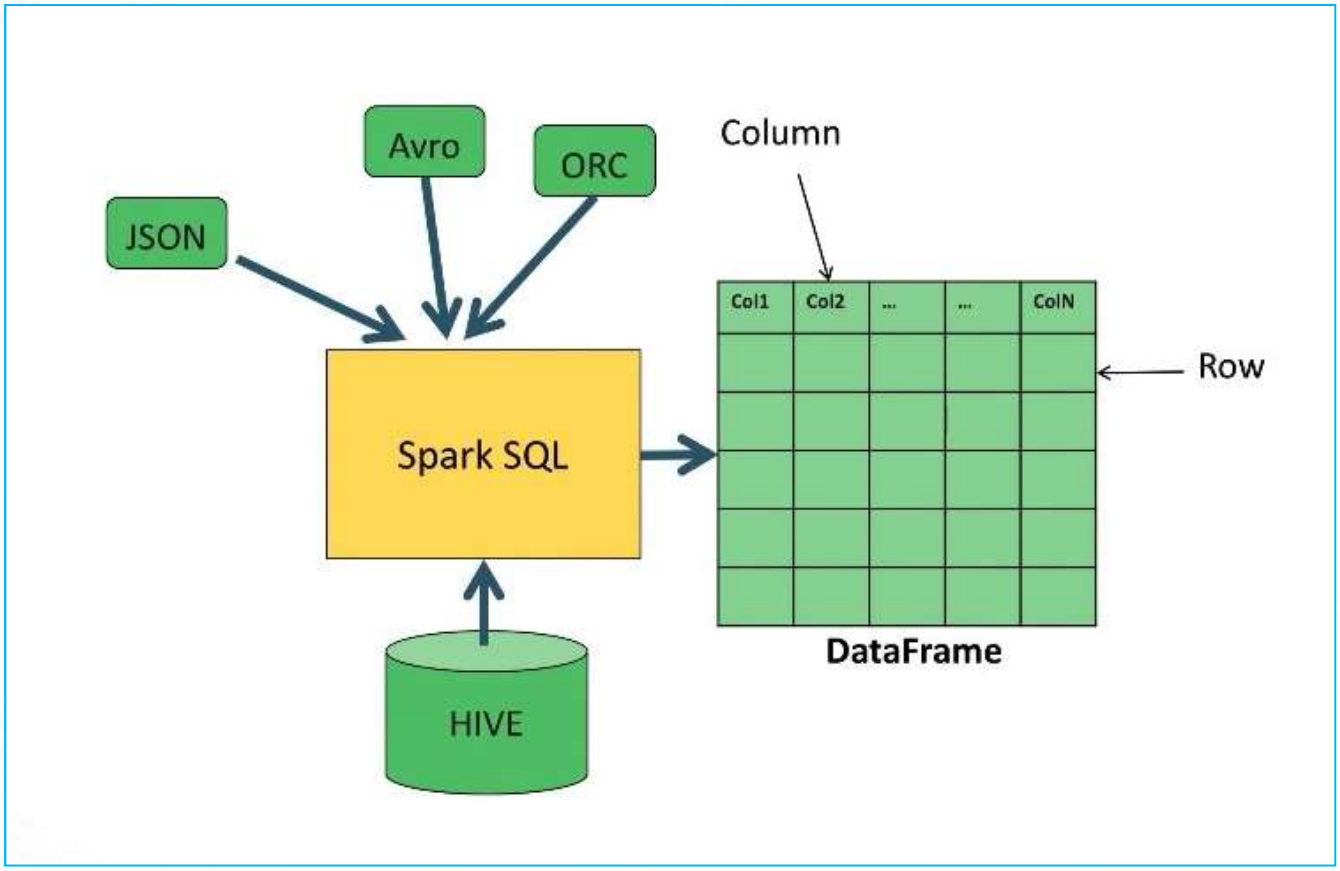
## Rows

- A row is nothing more than a record of data. Each record in a DataFrame must be of type Row
- We can create these rows manually from SQL, from Resilient Distributed Datasets (RDDs), from data sources, or manually from scratch

```
// in Scala
spark.range(2).toDF().collect()

# in Python
spark.range(2).collect()
```

Both result in an array of **`Row`** objects

# DataFrame


Avro
ORC
JSON
Column
Col1
Col2
...
...
ColN
Row
Spark SQL
HIVE
DataFrame

# Spark Types

- Spark has a large number of **internal type representations**.
- 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.

# Spark Types

| Data type | Value type in Python | API to access or create a data type |
|---|---|---|
| FloatType | float. Note: Numbers will be converted to 4-byte single-precision floating-point numbers at runtime. | FloatType() |
| DoubleType | float | DoubleType() |
| DecimalType | decimal.Decimal | DecimalType() |
| StringType | string | StringType() |
| BinaryType | bytearray | BinaryType() |
| BooleanType | bool | BooleanType() |
| TimestampType | datetime.datetime | TimestampType() |

A partial list of Spark types

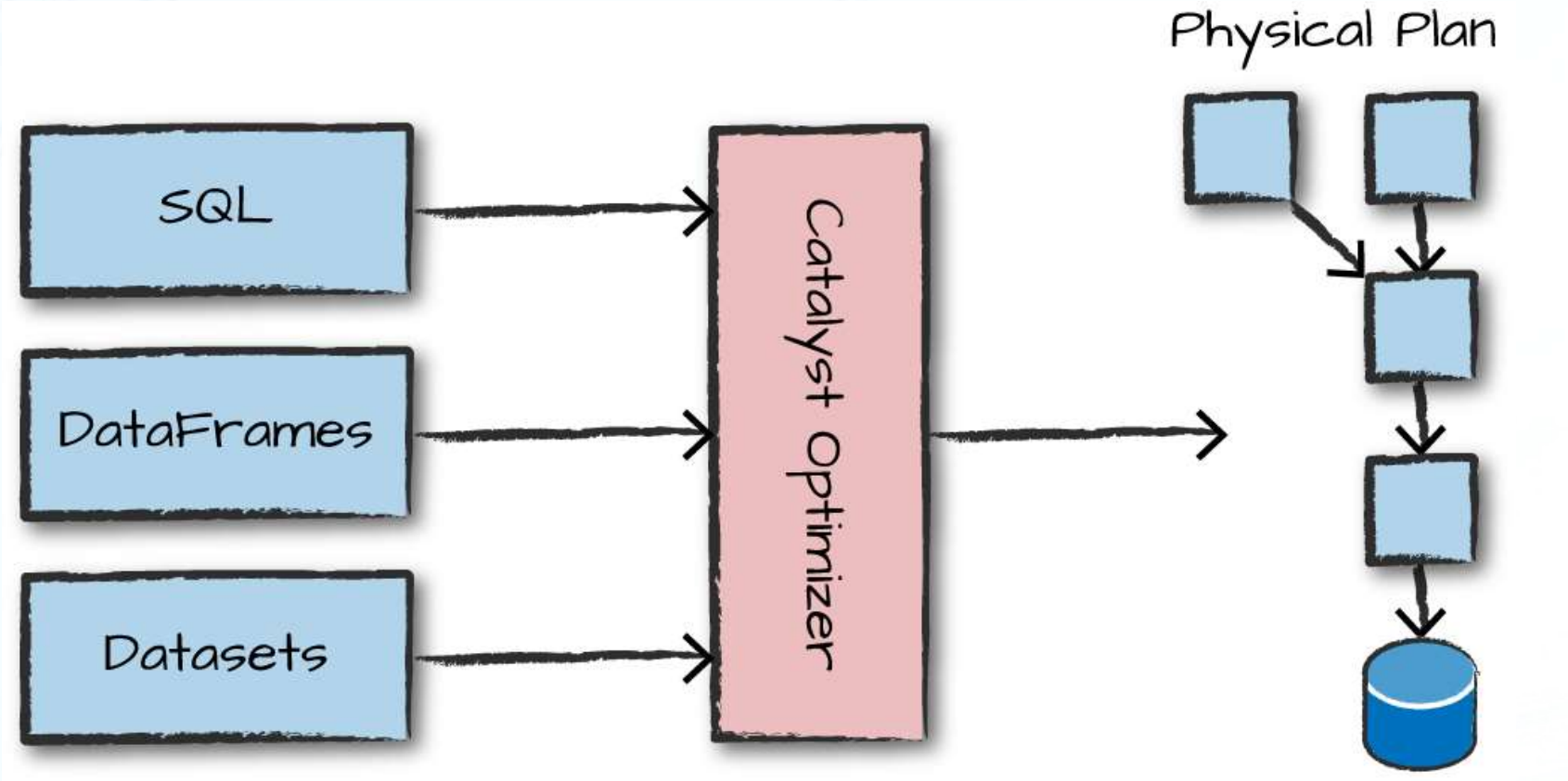
```
from pyspark.sql.types import *
b = ByteType()
```

# Let’s understand how the Structured API code gets executed

# Structured API Execution

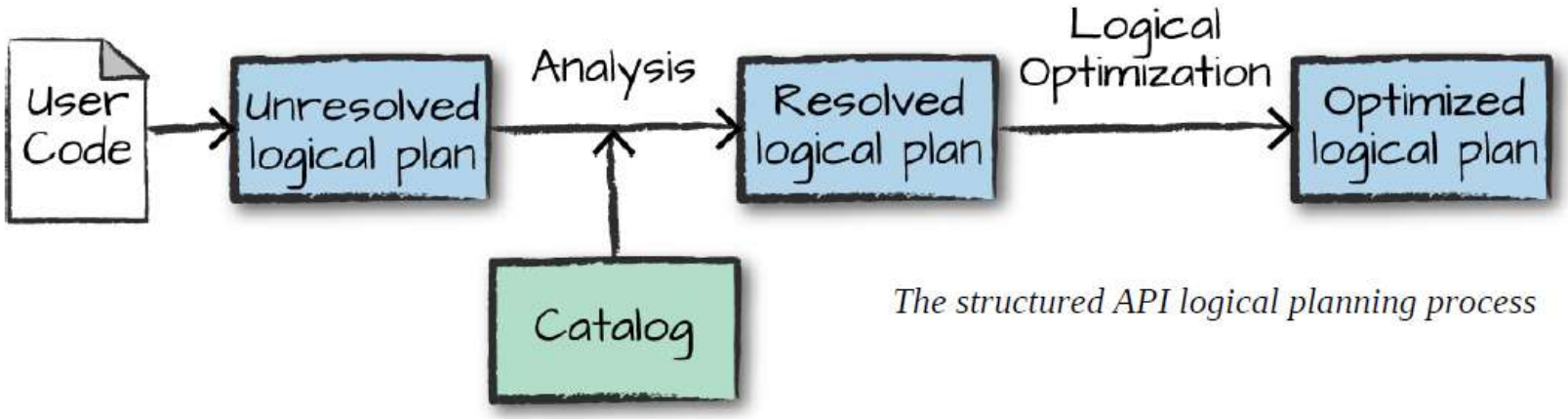
1. Write DataFrame/Dataset/SQL Code.
2. If valid code, Spark converts this to a ***Logical Plan***.
3. Spark transforms this Logical Plan to a ***Physical Plan***, checking for optimizations along the way.
4. Spark then executes this Physical Plan (RDD manipulations) on the cluster

# Structured API Execution



# Structured API Execution - Logical Planning

The first phase of execution – convert *user code* into a *logical plan.*




*The structured API logical planning process*

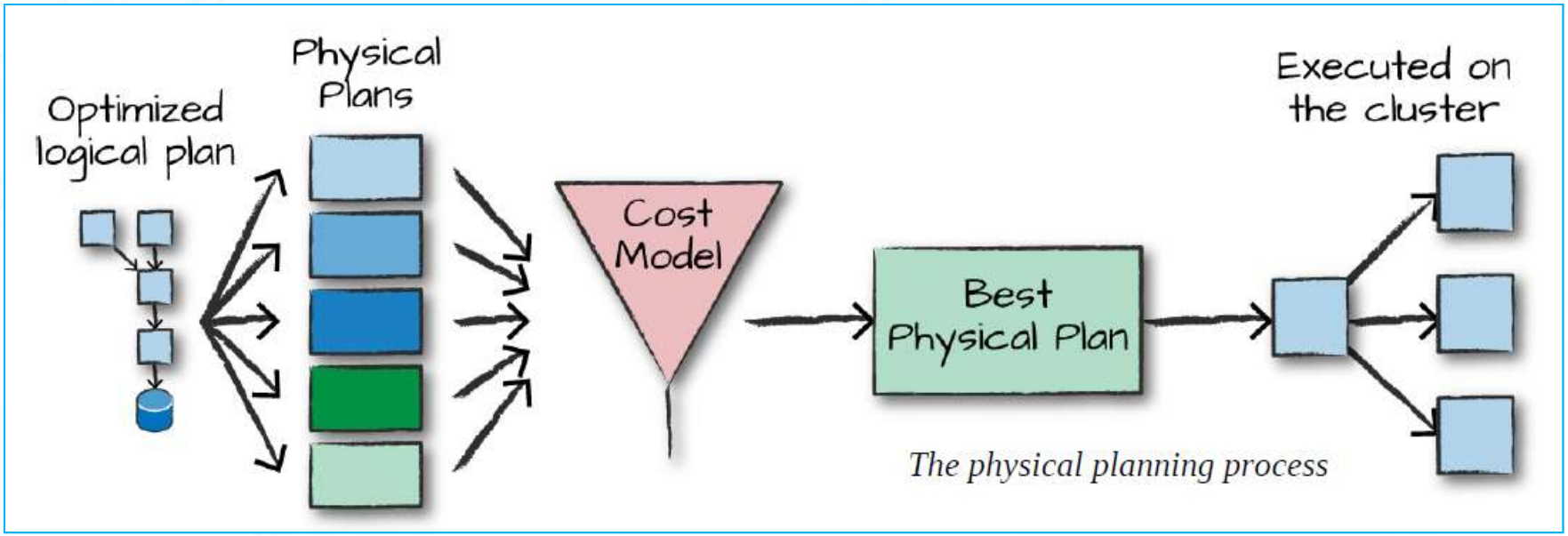
# Structured API Execution – Logical Planning

- This **logical plan** only represents a ***set of abstract transformations*** that do not refer to executors or drivers. It's purely to convert the user's set of expressions into the most optimized version.
- **Unresolved Logical Plan**
  - It does this by converting user code into an unresolved logical plan.
  - This plan is unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist.
  - 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.

# Structured API Execution - Logical Planning

- **Resolved Logical Plan**
  - If the analyzer can resolve it, then a (resolved) logical plan gets created.
- **Optimized Logical Plan**
  - This logical plan is passed through the ***Catalyst Optimizer***, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections.
  - After applying all the optimizations on the logical plan an optimized logical plan is finalized by the Catalyst Optimizer.

# Structured API Execution – Physical Planning



# Structured API Execution – Physical Planning

- **Physical Plan**
  - The physical plan (or 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).

# Structured API Execution

- Physical planning results in a series of RDDs and transformations.
  - This is why Spark is referred to as a compiler - it takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations for you.
- Upon selecting a physical plan, Spark runs all this code over RDDs.
- Spark performs further optimizations at runtime, generating native Java byte-code that can remove entire tasks or stages during execution.
- Finally the result is returned to the user.

## Let's look at

# Basic DataFrame Operations

# Definitions

### DataFrame

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

### Schema

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

### Partitioning

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

### Partitioning Scheme

The partitioning scheme defines how that is allocated. You can set this to be based on values in a certain column or non-deterministically.

# Schema Inference (Schema on read)

A schema defines the column names and types of a DataFrame. We can either let a data source define the schema (called schema-on-read) or we can define it explicitly ourselves.

```
# in Python
spark.read.format("json").load("/data/flight-data/json/2015-summary.json").schema
```

```
StructType(List(StructField(DEST_COUNTRY_NAME,StringType,true),
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true)))
```

A schema is a `StructType` made up of a number of fields, `StructField`s, that have a *name*, *type*, a *boolean flag* which specifies whether that column can contain missing or null values. You can optionally specify associated metadata with that column as well.

# Enforcing schema on a DataFrame

```
# in Python
from pyspark.sql.types import StructField, StructType, StringType, LongType

myManualSchema = StructType([
  StructField("DEST_COUNTRY_NAME", StringType(), True),
  StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
  StructField("count", LongType(), False, metadata={"hello":"world"})
])
df = spark.read.format("json").schema(myManualSchema)\
  .load("/data/flight-data/json/2015-summary.json")
```

# Records and Rows

- In Spark, each row in a DataFrame is a single record. Spark represents this record as an object of type Row.
- Spark manipulates Row objects using column expressions in order to produce usable values.
- Row objects internally represent arrays of bytes. The byte array interface is never shown to users because we only use column expressions to manipulate them.

```
from pyspark.sql import Row
myRow = Row("Hello", None, 1, False)

myRow[0]
myRow[2]
```

# Explicitly Creating DataFrames

```
from pyspark.sql import Row
from pyspark.sql.types import StructField, StructType, StringType, IntegerType

customSchema = StructType([
  StructField("name", StringType(), True),
  StructField("age", IntegerType(), True),
  StructField("city", StringType(), False)
])

row1 = Row("Raju", 40, "Hyderabad")
row2 = Row("Ravi", 30, "Bangalore")
row3 = Row("Vijay", 25, "Chennai")

myDf = spark.createDataFrame([row1, row2, row3], customSchema)

myDf.show()
```

# Spark Programmatic SQL Interface

# Running SQL Queries

- You can run the **`sql`** function on a **`SparkSession`** and get a DataFrame with the results of the sql query
- You can create a temporary view object from a DataFrame using **`createOrReplaceTempView`** method. You can then use **`sql`** to perform query operations on that view.

```
df = spark.read.json(data_path + "users.json")
df.createOrReplaceTempView("users")
sqlDF = spark.sql("SELECT * FROM users")
sqlDF.show()
```

# Global Temp View

- Global temporary views (introduced in Spark 2.1) allows you to share data among different sessions and keep alive until your application ends.
- In Spark SQL, temporary views are session-scoped and will be automatically dropped if the session terminates.
- All the global temporary views are tied to a system preserved temporary database **`global_temp`**.

```
df = spark.read.json(data_path + "users.json")

df.createGlobalTempView("gtv_users")

gtvDF = spark.sql("SELECT * FROM global_temp.gtv_users")
gtvDF.show()

gtvDF2 = spark.newSession().sql("SELECT * FROM global_temp.gtv_users")
gtvDF2.show()
```

# 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 *pyspark.sql.catalog.Catalog* package and contains a number of helpful functions for doing things like listing tables, databases, and functions.

# Accessing data from DataFrame - `select`

```
df.select("DEST_COUNTRY_NAME").show(2)

df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)

from pyspark.sql.functions import expr, col, column

df.select(expr("DEST_COUNTRY_NAME"),
          col("DEST_COUNTRY_NAME"),
          column("DEST_COUNTRY_NAME")) \
  .show(2)
```

One common error is attempting to mix Column objects and strings. For example, the following code will result in a **compiler error**:

```
df.select(col("DEST_COUNTRY_NAME"), "DEST_COUNTRY_NAME")
```

# Accessing data from DataFrame - `select`

- **`expr`** is the most flexible reference that we can use. It can refer to a plain column or a string manipulation of a column.
- To illustrate, let's change the column name, and then change it back by using the **`AS`** keyword and then the **`alias`** method on the column

```
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)

df.select(expr("DEST_COUNTRY_NAME as destination").alias("DEST_COUNTRY_NAME"))\
.show(2)
```

# Accessing data from DataFrame - `selectExpr`

- Because select followed by a series of expr is such a common pattern, Spark has a shorthand for doing this efficiently: **`selectExpr`**.
- This is probably the most convenient interface for general use.
- We can treat **`selectExpr`** as a way to build up complex expressions that create new DataFrames. We can add any valid non-aggregating SQL statement, and as long as the columns resolve, it will be valid.

```
df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME").show(2)

df.selectExpr(                                    ⇦ Adding new column to a dataframe
  "*", # all original columns
  "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")\
  .show(2)
```

# Using Literals - `lit`

- Sometimes, we need to pass explicit values into Spark that are just a value. You can do this through ***literals.***
- This is basically a translation from a given programming language's literal value to one that Spark understands. Literals are expressions and you can use them in the same way

```
from pyspark.sql.functions import lit
df.select(expr("*"), lit(1).alias("One")).show(2)
```

```
+-----------------+-------------------+-----+---+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----------------+-------------------+-----+---+
|    United States|            Romania|   15|  1|
|    United States|            Croatia|    1|  1|
+-----------------+-------------------+-----+---+
```

# Adding Columns - `withColumn`

- Using `withColumn` method, we can add new columns to a DataFrame.

```
df.withColumn("numberOne", lit(1)).show(2)

df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME"))\
  .show(2)
```

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
```

# Renaming Columns - `withColumnRenamed`

- **`withColumnRenamed`** method can be used to rename the column in a DataFrame with the name of the string in the first argument to the string in the second argument.

```
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns
```

# Removing Columns - `drop`

- `drop` method can be used to drop one or more columns from a DataFrame.

```
df.drop("ORIGIN_COUNTRY_NAME").columns
df.drop("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME")
```

# Changing a Column's Type - `cast`

- Sometimes, we might need to convert from one type to another; for example, if we have a set of StringType that should be integers. We can convert columns from one type to another by ***casting*** the column from one type to another.

```
df.withColumn("count2", col("count").cast("long")).printSchema()
```

# Filtering data from DataFrame - `filter` & `where`

- To filter rows, *we create an expression that evaluates to true or false*.
- The most common way to do this with DataFrames is to:
  - Create either an expression as a String
  - Build an expression by using a set of column manipulations.
- You can use **`where`** or **`filter`** methods to perform filtering of data. Both methods are equivalent.

```
df.filter(col("count") < 2).show(2)
df.filter("count < 2").show(2)

df.where(col("count") < 2).show(2)
df.where("count < 2").show(2)
```

# Getting unique rows - `distinct`

- `distinct` method allows us to de-duplicate any rows that are in a DataFrame and return unique values.

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

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

# Getting random samples - `sample`

- You can do random sampling of data on a DataFrame using **`sample`** method.
- You can specify a fraction of rows to extract from a DataFrame and whether you'd like to sample *with* or *without replacement*

```
seed = 5
withReplacement = False
fraction = 0.4
df.sample(withReplacement, fraction, seed).count()
```

# Create random splits - `randomSplit`

- `randomSplit` method allows us to break up a DataFrame into random "splits" of the original DataFrame

```
seed = 5
randomDfs = df.randomSplit([0.25, 0.35, 0.4], seed)
randomDfs[0].count()
randomDfs[1].count()
randomDfs[2].count()
```

# Concatenating Rows - `union`

- To append to a DataFrame, you must `union` the original DataFrame along with the new DataFrame. This just concatenates the two DataFrames.
- To union two DataFrames, you must be sure that they have the same schema and number of columns; otherwise, the union will fail.

# Sorting - `sort` & `orderBy`

- We can use `sort` & `orderBy` methods to do sorting on DataFrames.

```
df.sort("count").show(5)
df.orderBy("count", "DEST_COUNTRY_NAME").show(5)
df.orderBy(col("count"), col("DEST_COUNTRY_NAME")).show(5)

from pyspark.sql.functions import desc, asc
df.orderBy(expr("count desc")).show(2)
df.orderBy(col("count").desc(), col("DEST_COUNTRY_NAME").asc()).show(2)
```

# Limit - `limit`

```
df.limit(5).show()
df.orderBy(expr("count desc")).limit(6).show()
```

# Repartition and Coalesce

- An important optimization 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**, *will not incur a full shuffle* and will try to combine partitions.

# Repartition and Coalesce

```
df.rdd.getNumPartitions()
df.repartition(col("DEST_COUNTRY_NAME"))
df.repartition(5, col("DEST_COUNTRY_NAME"))
df.repartition(5, col("DEST_COUNTRY_NAME")).coalesce(2)
```

# Collecting rows to the Driver

- Spark maintains the state of the cluster in the driver. There are times when you'll want to collect some of your data to the driver in order to manipulate it on your local machine.

- Methods:

  - `collect` – gets all data from the entire DataFrame
  - `take` – selects the first N rows
  - `show` – prints out a number of rows nicely

```
collectDF = df.limit(10)
collectDF.take(5)
collectDF.show()
collectDF.show(5)
collectDF.collect()
```

# DataFrame Aggregation Functions

# Aggregation Functions

- `count`
  - Count number of rows
  - Specify a specific column to count, or all the columns by using count(*) or count(1)
- `countDistinct`
  - Get the number of unique groups that you want
- `approx_count_distinct`
  - Get the approximation of the distinct count to a certain degree of accuracy.

```
from pyspark.sql.functions import count, countDistinct, approx_count_distinct
df.select(count("StockCode")).show()
df.select(countDistinct("StockCode")).show()
df.select(approx_count_distinct("StockCode", 0.1)).show()
```

# Aggregation Functions

- **`first & last`**
  - Get the first and last values from a DataFrame
- **`min & max`**
  - Get minimum and maximum values from a DataFrame

```
from pyspark.sql.functions import first, last
df.select(first("StockCode"), last("StockCode")).show()

from pyspark.sql.functions import min, max
df.select(min("Quantity"), max("Quantity")).show()

spark.sql("""SELECT first(StockCode) as first,
                    last(StockCode) as last,
                    min(Quantity)  as minQty,
                    max(Quantity) as maxQty
             FROM dfTable""").show()
```

# Aggregation Functions

- **`sum`**
  - Sum all the values of a column
- **`sumDistinct`**
  - Sum a distinct set of values of a column
- **`avg`**
  - Use **`avg`** & **`mean`** function to get the average

```
from pyspark.sql.functions import sum, sumDistinct, avg

df.select(sum("Quantity")).show()
df.select(sumDistinct("Quantity")).show()
df.select(avg("Quantity")).show()
```

# Aggregation Functions

- **`varience`** & **`standardDeviation`**
  - The variance is the average of the squared differences from the mean, and the standard deviation is the square root of the variance.
  - Spark has both the formula for the *sample standard deviation* as well as the formula for the *population standard deviation*.
  - By default, Spark performs the formula for the sample standard deviation or variance if you use the **`variance`** or **`stddev`** functions.

```
from pyspark.sql.functions import var_pop, stddev_pop, variance, stddev
from pyspark.sql.functions import var_samp, stddev_samp

df.select(variance("Quantity"), stddev("Quantity"),
  var_pop("Quantity"), var_samp("Quantity"),
  stddev_pop("Quantity"), stddev_samp("Quantity")).show()
```

# Aggregation Functions

- **skewness & kurtosis**
  - Skewness and Kurtosis are both measurements of extreme points in your data. These are both relevant specifically when modeling your data as a probability distribution of a random variable.
  - Skewness measures the asymmetry of the values in your data around the mean.
  - Kurtosis is a measure of the tail of data.

```
from pyspark.sql.functions import skewness, kurtosis
df.select(skewness("Quantity"), kurtosis("Quantity")).show()
spark.sql("SELECT skewness(Quantity), kurtosis(Quantity) FROM dfTable").show()
```

# Aggregation Functions

- **Covariance & Correlation (corr, covar_samp, covar_pop)**
  - Covariance and correlation compare the interactions of the values in two difference columns together.
  - Correlation measures the Pearson correlation coefficient, which is scaled between –1 and +1.
  - The covariance is scaled according to the inputs in the data. Covariance can be calculated either as the sample covariance or the population covariance.

```
from pyspark.sql.functions import corr, covar_pop, covar_samp
df.select(corr("InvoiceNo", "Quantity"), covar_samp("InvoiceNo", "Quantity"),
    covar_pop("InvoiceNo", "Quantity")).show()
```

# Aggregating to Complex Types

- In Spark, we can perform aggregations on complex types as well.
- For example, we can collect a list of values, or unique values of column and perform aggregations on those collections.

```
from pyspark.sql.functions import collect_set, collect_list
df.agg(collect_set("Country"), collect_list("Country")).show()

spark.sql("SELECT collect_set(Country), collect_list(Country) FROM dfTable") \
     .show()
```

## DataFrame Aggregations

# Grouping

# GroupBy

- We can perform calculations based on groups on categorical data for which we group our data on one column and perform some calculations on the other columns that end up in that group.
- We do this grouping in two phases:
  - First we specify the column(s) on which we would like to group using groupBy. This step returns a **`RelationalGroupedDataset`**
  - Then, we specify the aggregation(s). This returns a **`DataFrame`**

```
df.groupBy("InvoiceNo", "CustomerId").count().show()
```

**`RelationalGroupedDataset`**

**`DataFrame`**

# Grouping with Expressions

- Specifying expressions within **`agg`** makes it possible for you to pass-in arbitrary expressions that just need to have some aggregation specified.
- You can even do things like alias a column after transforming it.

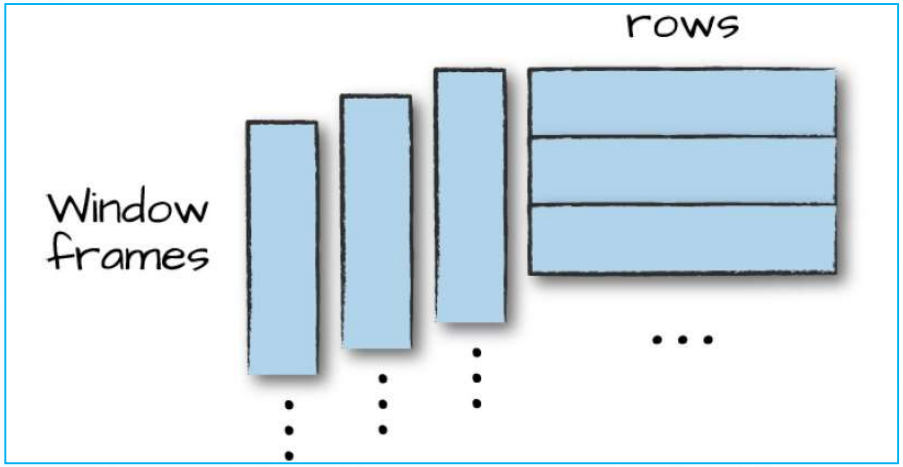
```
df.groupBy("InvoiceNo").agg(
     count("Quantity").alias("quan"),
     expr("count(Quantity)"))\
    .show()
```

# Window Functions

- Window functions carry out some unique aggregations by computing some aggregation on a specific “window” of data, which you define by using a reference to the current data.
- This window specification determines which rows will be passed in to this function.
- 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.

# Window Functions

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




# Joins

# Join Types

- ***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 dataset only where the key appears in the right dataset
- ***Left anti joins*** keep the rows in the left dataset only where they do not appear in the right dataset
- ***Cross (or Cartesian) joins*** match every row in the left dataset with every row in the right dataset

# Join Types

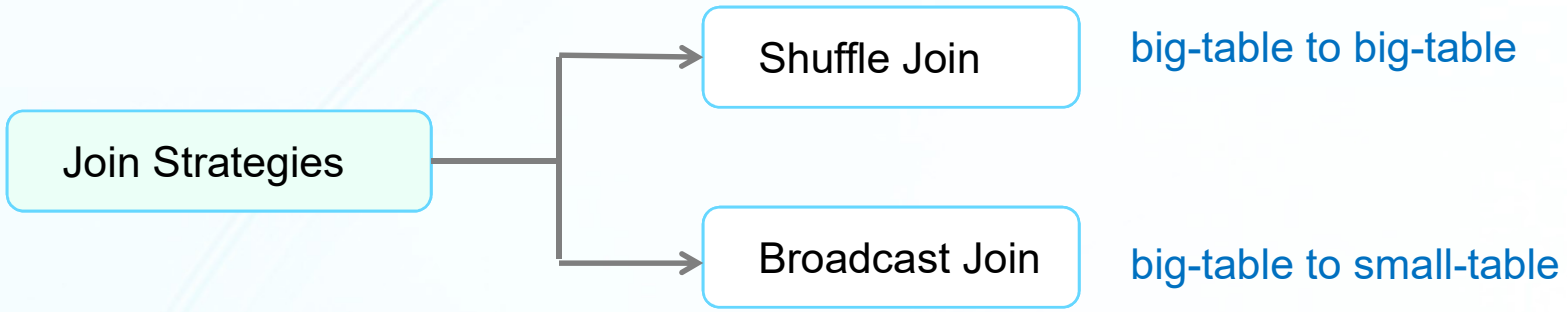
```
employee = spark.createDataFrame([
    (1, "Raju", 25, 101),
    (2, "Ramesh", 26, 101),
    (3, "Amrita", 30, 102),
    (4, "Madhu", 32, 102),
    (5, "Aditya", 28, 102),
    (6, "Aditya", 28, 10000)])\
  .toDF("id", "name", "age", "deptid")

department = spark.createDataFrame([
    (101, "IT", 1),
    (102, "Opearation", 1),
    (103, "HRD", 2)])\
  .toDF("id", "deptname", "locationid")
```

```
# inner join using SQL
employee.createOrReplaceTempView("employee")
department.createOrReplaceTempView("department")

sql_ij = """SELECT * FROM employee JOIN department
            ON employee.deptid = department.id"""

spark.sql(sql_ij).show()
```

```
joinEmpDept = employee["deptid"] == department['id']

employee.join(department, joinEmpDept).show()
employee.join(department, joinEmpDept, "outer").show()
employee.join(department, joinEmpDept, "left_outer").show()
employee.join(department, joinEmpDept, "right_outer").show()
employee.join(department, joinEmpDept, "left_semi").show()
employee.join(department, joinEmpDept, "left_anti").show()
employee.join(department, joinEmpDept, "cross").show()
```

# Join Strategies

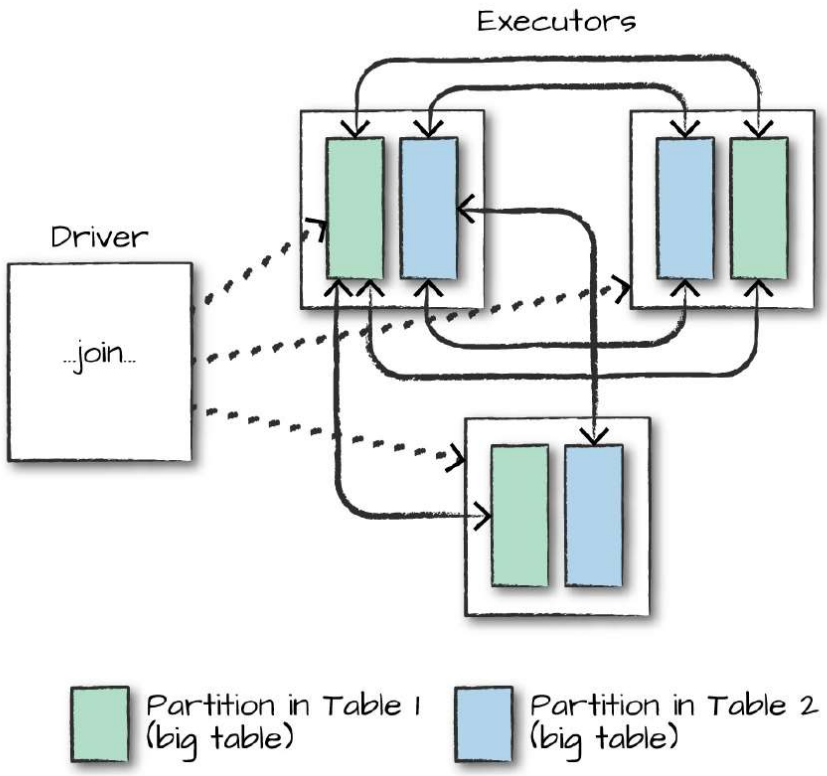
- Spark approaches cluster communication in two different ways during joins. It either incurs a ***shuffle join***, which results in an all-to-all communication or a ***broadcast join***.




# Shuffle Join

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.

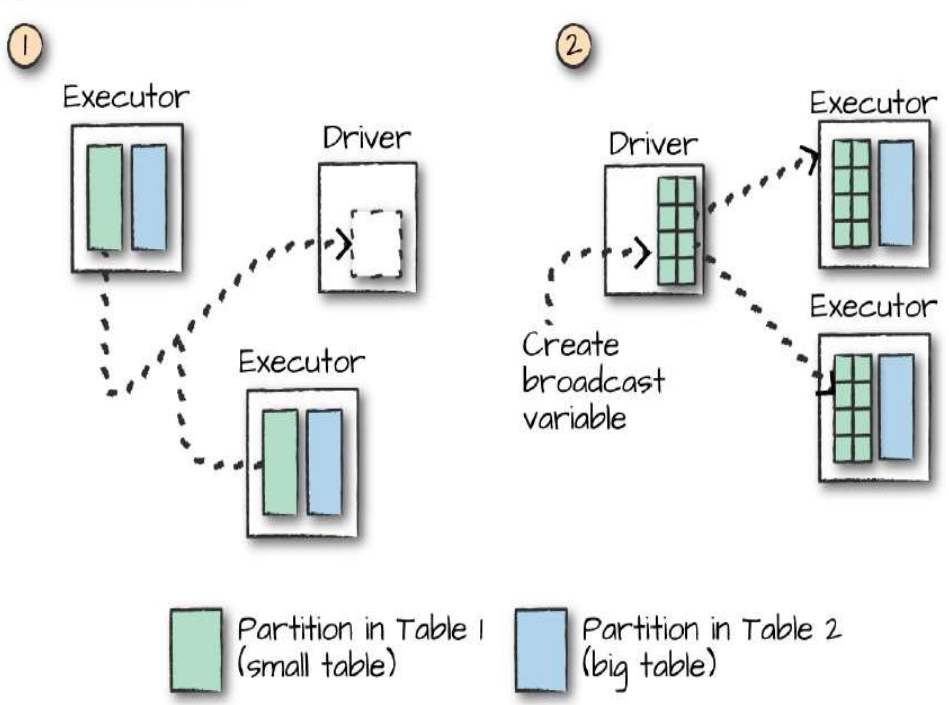
These joins are expensive because the network can become congested with traffic, especially if your data is not partitioned well.



# Broadcast Join

When the table is small enough to fit into the memory of a single worker node, we can use a broadcast join, where we replicate our small DataFrame onto every worker node in the cluster.

This will 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



# Data Sources

# Spark's Core Data Sources

Spark supports a variety of data sources out of the box

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

Spark supports some community created data sources as well

- Cassandra
- HBase
- MongoDB
- XML
- And many others

# Reading Data

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

**`spark.read`** returns **`DataFrameReader`**

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

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

# Read Modes

- Reading data from an external source entails encountering malformed data, especially when working with semi-structured data sources.
- Read modes specify what will happen with these malformed records.

| Read Mode | Description |
|---|---|
| Permissive (default) | 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 |

# Writing Data

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

**`dataframe.write`** returns **`DataFrameWriter`**

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

- The foundation for writing data in Spark is the **`DataFrameWriter`**. After we have a **`DataFrameWriter`**, we specify three values: the *format*, a series of *options*, and the *save mode*. At a minimum, you must supply a *path*.
- Options vary from data source to data source

# Write Modes

| Read 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` `(default)` | 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 |

# CSV Files

Check out the code example

```
csvFile = spark.read.format("csv")\
  .option("header", "true")\
  .option("mode", "FAILFAST")\
  .option("inferSchema", "true")\
  .load(data_path)

fitlerQry = col("DEST_COUNTRY_NAME") == "United States"
csvFiltered = csvFile.where(fitlerQry)

csvFiltered.write.format("csv")\
  .mode("overwrite")\
  .option("sep", "\t")\
  .save(data_output_path)
```

# JSON Files

Check out the code example

```
jsonFile = spark.read.format("json")\
  .option("mode", "FAILFAST")\
  .option("inferSchema", "true")\
  .load(data_path)

filterQry = col("count") > 100
jsonFiltered = jsonFile.where(filterQry)

jsonFiltered.write.format("json")\
  .mode("overwrite")\
  .save(data_output_path)
```

# Parquet Files

- Parquet is the default file format of Spark
- Parquet is an open source column-oriented data store that provides a variety of storage optimizations, especially for analytics workloads.
- It provides columnar compression, which saves storage space and allows for reading individual columns instead of entire files.
- Parquet is generally recommended for long-term storage because reading from Parquet will always be more efficient than JSON or CSV.
- Parquet supports complex types such as `array`, `map`, or `struct`. (which is not the case with CSV or JSON)

# Parquet Files

Check out the code example

```
parquetFile = spark.read\
    .format("parquet")\
    .load(data_path)

filterQry = col("count") > 100
parquetFiltered = parquetFile.where(filterQry)

parquetFiltered.write.format("parquet")\
  .mode("overwrite")\
  .save(data_output_path)
```

# 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.
- Parquet and ORC are quite similar; the fundamental difference is that Parquet is further optimized for use with Spark, whereas ORC is further optimized for Hive.

# ORC Files

Check out the code example

```
orcFile = spark.read\
    .format("orc")\
    .load(data_path)

filterQry = col("count") > 100
orcFiltered = orcFile.where(filterQry)

orcFiltered.write.format("orc")\
  .mode("overwrite")\
  .save(data_output_path)
```

# Reading from Databases

- To read and write from SQL databases, you need to do two things:
  - Include the JDBC driver for you database on the spark classpath
  - Provide the proper JAR for the driver itself.

```
./bin/spark-shell --driver-class-path postgresql-9.4.1207.jar
                  --jars postgresql-9.4.1207.jar
```

# SQL Databases

Check out the code example

```
// Reading from JDBC source
val jdbcDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .load()

// Saving data to a JDBC source
jdbcDF.write
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .save()
```

# Working with Hive

- Spark supports reading and writing data stored in Apache Hive.
- We must instantiate **`SparkSession`** with Hive support, including connectivity to a persistent Hive metastore, support for Hive SerDes, and Hive UDFs.
- Users who do not have an existing Hive deployment can still enable Hive support. When not configured by the hive-site.xml, the context automatically creates metastore_db in the current directory and creates a directory configured by **`spark.sql.warehouse.dir`**, which defaults to the directory **`spark-warehouse`** in the current directory that the Spark application is started.

# Working with Hive

Check out the code example

```
spark = SparkSession \
    .builder \
    .appName("Datasorces") \
    .config("spark.master", "local") \
    .config("spark.sql.warehouse.dir", warehouse_location)\
    .enableHiveSupport() \
    .getOrCreate()

spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
spark.sql("LOAD DATA LOCAL INPATH 'C:/PySpark/data/hive/kv1.txt' INTO TABLE src")

spark.sql("SELECT * FROM src").show()

spark.sql("SELECT COUNT(*) FROM src").show()
```

# THANK YOU