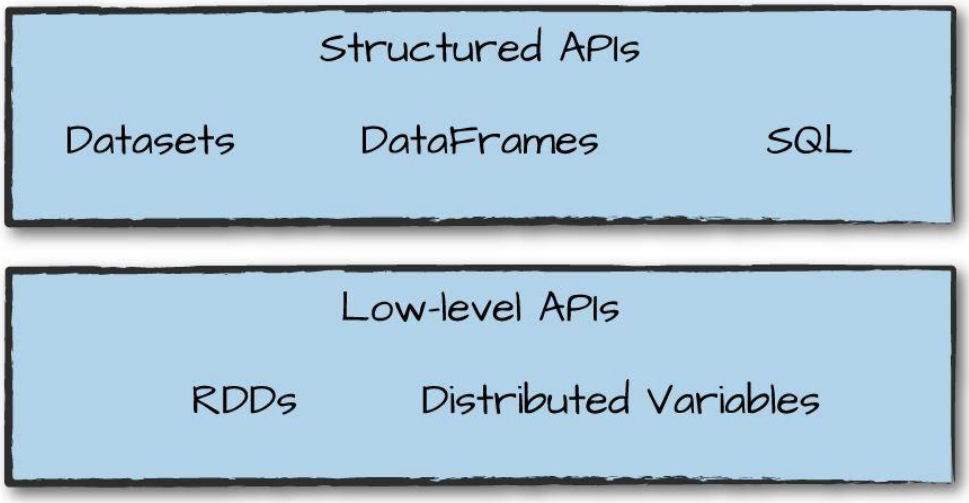
BIG DATA ANALYTICS

UNIT-5

Structured API: DataFrames, SQL:Overview of Structured Spark Types, Schemas, Columns and Expressions, Data Frame Transformations, Working with different types of data,Aggregations- Aggregation Functions, Grouping, User-Defined Aggregation Functions,Joins-Inner Joins, Outer Joins, Processing CSV Files, JSON Files, Text Files and Parquet Files, Spark SQL

Structured API:

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



- The **Structured APIs** are a tool for manipulating all sorts of data, from unstructured log files to semi-structured **CSV** (comma-separated values(a text file that has a specific format which allows data to be saved in a table structured format.) files etc.
- These APIs refer to three core types of distributed collection APIs:

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

Limitations of RDD

1. There is no built-in optimization engine in RDD.
2. RDD cannot handle structured data.

These are the following drawbacks, due to which Spark SQL **dataframe** comes in picture. Dataframe overcomes limitations of RDD as it provides memory management and optimized execution plan. This feature is not available in RDD.

DataFrames:

- DataFrames are a fundamental data structure in structured application programming interfaces (APIs) used for working with structured data.
- In Spark, DataFrames are the distributed collections of data, organized into rows and columns.
- Each column in a DataFrame has a name and an associated type.
- DataFrame is based on RDD, it translates SQL code and domain-specific language (DSL) expressions into optimized low-level RDD operations.
- DataFrames have become one of the most important features in Spark and made Spark SQL the most actively developed Spark component. Since Spark 2.0, DataFrame is implemented as a special case of Dataset.
- DataFrame has two main advantages over RDD:

  Optimized execution plans via Catalyst Optimizer.

  Custom Memory management via Project Tungsten.
- Spark DataFrames can be created from various sources, such as Hive tables, log tables, external databases, or the existing **RDDs**.
- It supports structured and semi-structured data's and has various data sources transforming into the dataframe that loses the RDD.
- It does not have compile-time safety, only detects the runtime error, and it takes the query optimization through the catalyst optimizer; the serialization happens with the memory in the binary format.
- It manually avoids garbage collection for creating or destroying the objects and operations performed only on the serialized data without deserialization.
- DataFrames provide a way to represent and manipulate structured data efficiently, allowing users to perform various operations such as filtering, aggregating, transforming, and analyzing data.
- They are widely used in data analysis, data manipulation, and data preprocessing tasks.

- DataFrames allow the processing of huge amounts of data

- In Spark, createDataFrame() and toDF() methods are used to create a DataFrame manually, using these methods we can create a Spark DataFrame from already **existing RDD, DataFrame, Dataset, List, Seq data objects** etc.

creation of dataframe with Scala examples:

- In Spark, createDataFrame() and toDF() methods are used to create a DataFrame manually, using these methods you can create a Spark DataFrame from already existing RDD, DataFrame, Dataset, List, Seq data objects, here below are some Scala examples.
- we can also create a DataFrame from different sources like Text, CSV, JSON, XML, Parquet, Avro, files, RDBMS Tables, Hive, HBase, and many more.

1. Spark Create DataFrame from RDD

- One easy way to create Spark DataFrame manually is from an existing RDD. first, let's create an RDD from a collection Seq by calling **parallelize().**

```
// Spark Create DataFrame from RDD
val rdd = spark.sparkContext.parallelize(data)
```

1.1 Using toDF() function:

Once we have an RDD, let's use toDF() to create DataFrame in Spark. By default, it creates column names as "_1" and "_2" as we have two columns for each row.

```
val dfFromRDD1 = rdd.toDF()
dfFromRDD1.printSchema()
```

- Since RDD is schema-less without column names and data type, converting from RDD to DataFrame gives you default column names as _1, _2 and so on and data type as String. Use DataFrame **printSchema() to print the schema** to console.

```
root
 |-- _1: string (nullable = true)
 |-- _2: string (nullable = true)
```

- toDF() has another signature to assign a column name, this takes a variable number of arguments for column names as shown below.

```
val dfFromRDD1 = rdd.toDF("language","users_count")
dfFromRDD1.printSchema()
```

```
root
 |-- language: string (nullable = true)
 |-- users: string (nullable = true)
```

- By default, the datatype of these columns assigns to String. We can change this behavior by supplying schema – where we can specify a column name, data type and nullable for each field/column.

1.2 Using Spark createDataFrame() from SparkSession

- Using createDataFrame() from SparkSession is another way to create and it takes rdd object as an argument. and chain with toDF() to specify names to the columns.

```
val dfFromRDD2 = spark.createDataFrame(rdd).toDF(columns:_*)
```

2.Create Spark DataFrame from List and Seq Collection

- There are several approaches to create Spark DataFrame from collection Seq[T] or List[T]. These examples would be similar to what we have seen in the above section with RDD, but we use “data” object instead of “rdd” object.

2.1 Using toDF() on List or Seq collection

- toDF() on collection (Seq, List) object creates a DataFrame. make sure importing import spark.implicits._ to use toDF()

```
import spark.implicits._
val dfFromData1 = data.toDF()
```

2.2 Using createDataFrame() from SparkSession

- Calling createDataFrame() from SparkSession is another way to create and it takes collection object (Seq or List) as an argument. and chain with toDF() to specify names to the columns.

```
// From Data (USING createDataFrame)
var dfFromData2 = spark.createDataFrame(data).toDF(columns:_*)
```

3. Create Spark DataFrame from CSV file

how to create DataFrame from data sources like CSV, text, JSON, Avro e.t.c

Spark by default provides an API to read a delimiter files like comma, pipe, tab separated files and it also provides several options on handling with header, with out header, double quotes, data types e.t.c.

```
// Create Spark DataFrame from CSV
val df2 = spark.read.csv("/src/resources/file.csv")
```

4. Creating from text (TXT) file: Here, will see how to create from a TXT file.

```
// Creating from text (TXT) file
val df2 = spark.read
.text("/src/resources/file.txt")
```

5. Creating from JSON file

Here, will see how to create from a JSON file.

```
val df2 = spark.read
.json("/src/resources/file.json")
```

6. Creating from Hive

```
// Creating from Hive
val hiveContext = new
org.apache.spark.sql.hive.HiveContext(spark.sparkContext)
val hiveDF = hiveContext.sql(“select * from emp”)
```

In PYTHON :

Here's an example of creating a DataFrame using pandas in Python:import pandas as pd

```
# Create a DataFrame from a dictionary

data = {'Name': ['John', 'Jane', 'Mike'],
        'Age': [25, 30, 35],
        'City': ['New York', 'London', 'Sydney']}


df = pd.DataFrame(data)

# Print the DataFrame
print(df)
```

```
output: Name  Age      City
0  John   25 New York
1  Jane   30    London
2  Mike   35    Sydney
```

Dataframe Features:-

- **Distributed collection of Row Object:** A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent it is as similar to a table in RDBMS.
- Dataframes are able to process the data in different sizes, like the size of kilobytes to petabytes on a single node cluster to large cluster.
- **Data Processing:** Processing structured and unstructured data formats (Avro, CSV, elastic search, and Cassandra) It also provides storage systems like HDFS, HIVE tables, MySQL, etc.
- It can read and write from all these various datasources.
- **Optimization using catalyst optimizer:** It powers both SQL queries and the DataFrame API. Dataframe use catalyst tree transformation framework in four phases,
    1. Analyzing a logical plan to resolve references
    2. Logical plan optimization
    3. Physical planning
    4. Code generation to compile parts of the query to Java bytecode.
- **Hive Compatibility:** Using Spark SQL, you can run unmodified Hive queries on your existing Hive warehouses. It reuses Hive frontend and MetaStore and gives you full compatibility with existing Hive data, queries, and UDFs.
- We can integrate dataframe with all big data tools and frameworks by spark-core.
- **ProgrammingLanguagessupported:**
  Dataframe provides several API, such as Python, Java, Scala, and R programming.

Dataset :

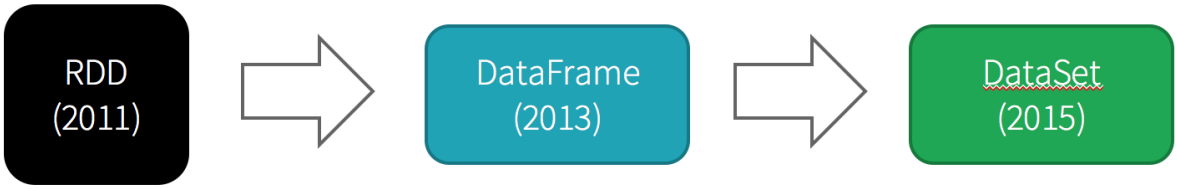
- Dataset is a data structure in **SparkSQL** which is strongly typed and is a map to a relational schema.
- A Dataset is a distributed collection of data elements spread across machines and combined and configured into clusters. It is an extension to **data frame API** that provides the benefits of strong typing, compile-time type safety, and object-oriented programming.
- It is essentially a strongly-typed version of a DataFrame, where each row of the Dataset is an object of a specific type, defined by a case class or a Java class.
- The dataset is a combination of RDD and dataframe; also, the original RDD regenerates after transformation.
- Datasets in Spark Scala can be created from a variety of sources, such as RDDs, DataFrames, structured data files (e.g., CSV, JSON, Parquet), Hive tables, or external databases.
- One of the key benefits of using Datasets in Spark Scala is their ability to provide compile-time type safety and object-oriented programming, which can help catch errors at compile time rather than runtime. This can help improve code quality and reduce the likelihood of errors.
- It is the compile-time safety and tuning of the query optimization through the catalyst optimizers like dataframes.
- When we use an encoder, it handles the data conversion between the objects and the tables, and no need for garbage collection, so it saves memory.
- It accesses the individual attributes and elements without deserializing the objects.
- Here is an example of how to create and work with a Spark Scala Dataset:

```scala
// create a Dataset from a sequence of case class objects
val data = Seq(Person("John", 25), Person("Jane", 30), Person("Bob", 45))
val ds = spark.createDataset(data)(Encoders.product[Person])

// display the Dataset
ds.show()

//Result
+----+---+
|name|age|
+----+---+
|John| 25|
|Jane| 30|
| Bob| 45|
+----+---+
```

History of Spark APIs



Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

Internally rows, externally JVM objects

Almost the "Best of both worlds": type safe + fast

But slower than DF
Not as good for interactive analysis, especially Python

databricks

SQL tables and views

- A table consists of rows and columns to store and organized data in a structured format, while the view is a result set of SQL statements.
- With **Spark SQL**, you can register any DataFrame as a table or view (a temporary table) and query it using pure SQL. There is no performance difference between writing SQL queries or writing DataFrame code, they both "compile" to the same underlying plan that we specify in DataFrame code.
- A table is **structured** with columns and rows, while a view is a virtual table **extracted** from a database.

RDD vs. DataFrame vs. Dataset Differences

The table outlines the significant distinctions between the three Spark APIs:

	RDD	DataFrame	Dataset
Release version	Spark 1.0	Spark 1.3	Spark 1.6
Data Representation	Distributed collection of elements.	Distributed collection of data organized into columns.	Combination of RDD and DataFrame.
Data Formats	Structured and unstructured are accepted.	Structured and semi-structured are accepted.	Structured and unstructured are accepted.
Data Sources	Various data sources.	Various data sources.	Various data sources.
Immutability and Interoperability	Immutable partitions that easily transform into DataFrames.	Transforming into a DataFrame loses the original RDD.	The original RDD regenerates after transformation.
Compile-time type	Available compile-	No compile-time	Available compile-

	RDD	DataFrame	Dataset
safety	time type safety.	type safety. Errors detect on runtime.	time type safety.
Optimization	No built-in optimization engine. Each RDD is optimized individually.	Query optimization through the Catalyst optimizer.	Query optimization through the Catalyst optimizer, like DataFrames.
Serialization	RDD uses Java serialization to encode data and is expensive. Serialization requires sending both the data and structure between nodes.	There is no need for Java serialization and encoding. Serialization happens in memory in binary format.	Encoder handles conversions between JVM objects and tables, which is faster than Java serialization.
Garbage Collection	Creating and destroying individual objects creates garbage collection overhead.	Avoids garbage collection when creating or destroying objects.	No need for garbage collection
Efficiency	Efficiency decreased for serialization of individual objects.	In-memory serialization reduces overhead. Operations performed on serialized data without the need for deserialization.	Access to individual attributes without deserializing the whole object.
Lazy Evaluation	Yes.	Yes.	Yes.
Programming	Java Scala Python R	Java Scala Python R	Java Scala

	RDD	DataFrame	Dataset
Language Support			
Schema Projection	Schemas need to be defined manually.	Auto-discovery of file schemas.	Auto-discovery of file schemas.
Aggregation	Hard, slow to perform simple aggregations and grouping operations.	Fast for exploratory analysis. Aggregated statistics on large datasets are possible and perform quickly.	Fast aggregation on numerous datasets.

Overview of Structured API Execution:

- This section will demonstrate how this code is actually executed across a cluster. This will help you understand (and potentially debug) the process of writing and executing code on clusters, so let's walk through the execution of a single structured API query from user code to executed code. Here's an overview of the steps:

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

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

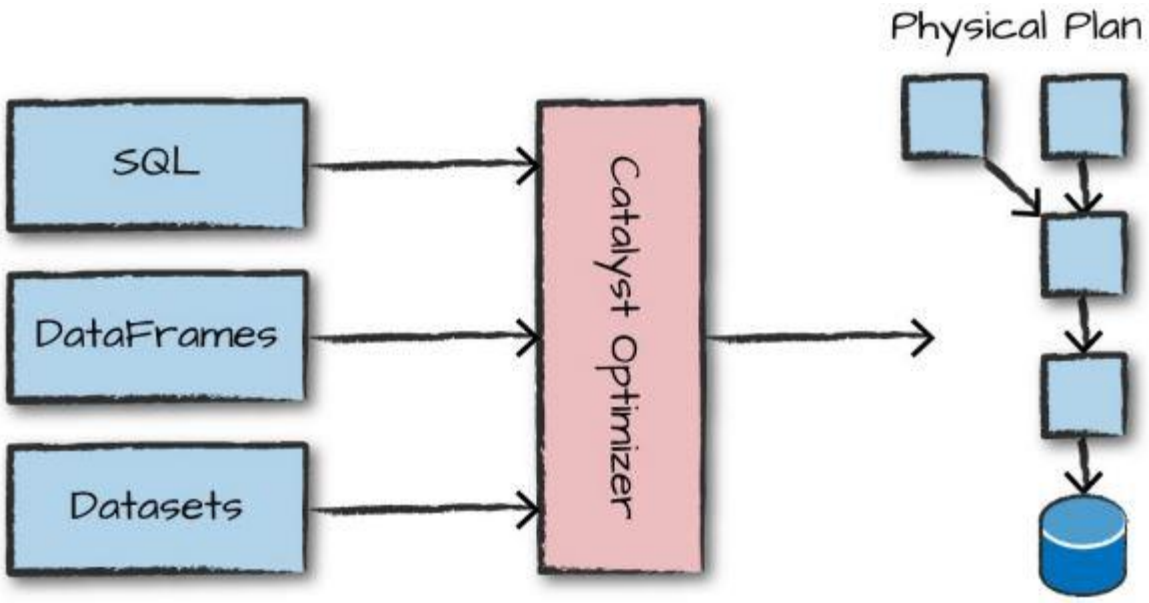


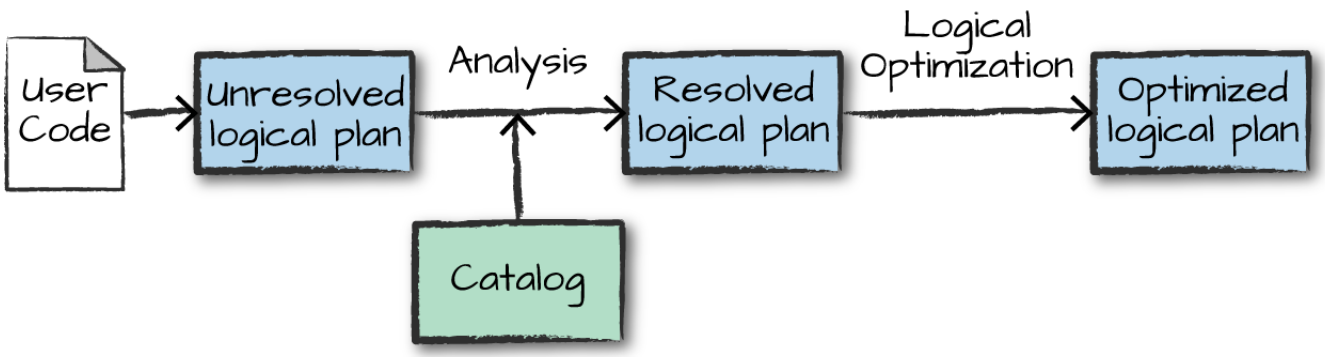
Figure 4-1. The Catalyst Optimizer

Logical Planning

The first phase of execution is meant to take user code and convert it into a logical plan.

Figure below illustrates this process.

Figure 4-2. The structured API logical planning process



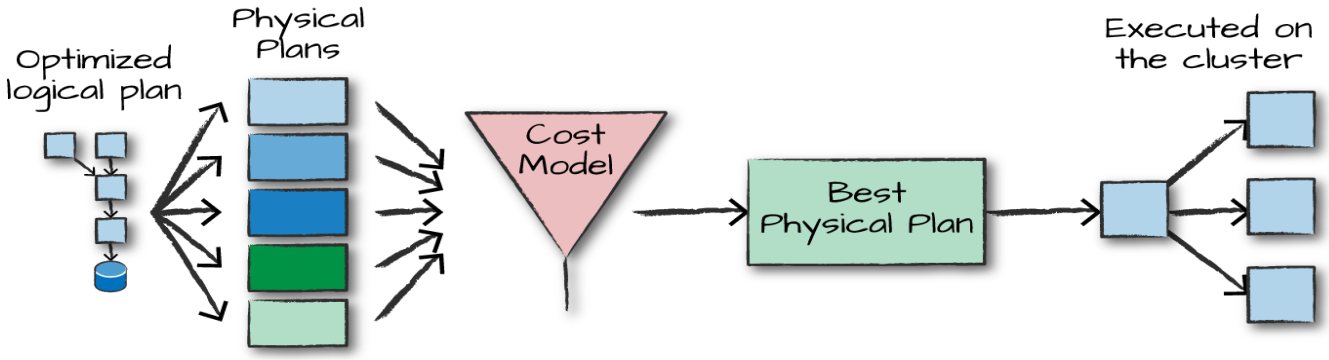
- This logical plan only represents a set of abstract transformations that do not refer to executors or drivers, it's purely to convert the user's set of expressions into the most optimized version.
- It does this by converting user code into an unresolved logical plan. This plan is unresolved because 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.

- 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.
- Packages can extend the Catalyst to include their own rules for domain-specific optimizations.

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, as depicted in Figure 4
- 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 .

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 Results are then returned to user in a file .

Overview of Structured Spark Types:

- Spark Is Effectively A Programming Language Of Its Own. Internally, Spark Uses An Engine Called Catalyst That Maintains Its Own Type Information Through The Planning And Processing Of Work. In Doing So, This Opens Up A Wide Variety Of Execution Optimizations That Make Significant Differences.

- 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.

- For Example, The Following Code Does Not Perform Addition In Scala Or Python; It Actually Performs Addition Purely In Spark:

```
# in Python
df = spark.range(500).toDF("number")
df.select(df["number"] + 10)
// in Scala
val df = spark.range(500).toDF("number")
df.select(df.col("number") + 10)
```

  This Addition Operation Happens Because Spark Will Convert An Expression Written In An Input Language To Spark's Internal Catalyst Representation Of That Same Type Information.

The spark.sql is a module in Spark that is used to perform SQL-like operations on the data stored in memory.

Following are the important classes from the SQL module.

- spark.sql.SparkSession – SparkSession is the main entry point for DataFrame and SQL functionality.
- spark.sql.DataFrame – DataFrame is a distributed collection of data organized into named columns.
- spark.sql.Column – A column expression in a DataFrame.
- spark.sql.Row – A row of data in a DataFrame.
- spark.sql.GroupedData – An object type that is returned by DataFrame.groupBy().
- spark.sql.DataFrameNaFunctions – Methods for handling missing data (null values).
- spark.sql.functions – List of standard built-in functions.
- spark.sql.types – Available SQL data types in Spark.

All data types of Spark SQL are located in the package org.apache.spark.sql.types. You can access them by doing

```
importorg.apache.spark.sql.types._
```

Spark SQL and DataFrames support the following data types:

- **Numeric types**
  - ByteType
  - ShortType
  - IntegerType
  - LongType.
  - FloatType.
  - DoubleType
  - DecimalType
- **String type**
  - StringType: Represents character string values.
  - VarcharType(length): A variant of StringType which has a length limitation. Data writing will fail if the input string exceeds the length limitation. this type can only be used in table schema, not functions/operators.
  - CharType(length): A variant of VarcharType(length) which is fixed length.
- **Binary type**
  - BinaryType: Represents byte sequence values.
- **Boolean type**
  - BooleanType: Represents booleanvalues.(true/false)

- To work with the correct Scala types, use the following:

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

Scala (data types)

Data type	Value type in Scala	API to access or create a data type
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
DecimalType	java.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Byte]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.time.Instant or java.sql.Timestamp	TimestampType
DateType	java.time.LocalDate or java.sql.Date	DateType
YearMonthIntervalType	java.time.Period	YearMonthIntervalType
DayTimeIntervalType	java.time.Duration	DayTimeIntervalType
MapType	scala.collection.Map	MapType(*keyType*, *valueType*, [*valueContainsNull]*) **Note:** The default value of *valueContainsNull* is true.
StructType	org.apache.spark.sql.Row	StructType(*fields*) **Note:** *fields* is a Seq of StructFields. Also, two fields with the same name are not allowed.
StructField	The value type in Scala of the data type of this field(For example, Int for a StructField with the data type IntegerType)	StructField(*name*, *dataType*, [*nullable*])

JAVA(data types)

All data types of Spark SQL are located in the package of org.apache.spark.sql.types

```
import org.apache.spark.sql.types.DataTypes;
ByteType x = DataTypes.ByteType;
```

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
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.time.Instant or java.sql.Timestamp	DataTypes.TimestampType
DateType	java.time.LocalDate or java.sql.Date	DataTypes.DateType
YearMonthIntervalType	java.time.Period	DataTypes.YearMonthIntervalType
DayTimeIntervalType	java.time.Duration	DataTypes.DayTimeIntervalType
MapType	java.util.Map	DataTypes.createMapType(*keyType*, *valueType*)
StructType	org.apache.spark.sql.Row	DataTypes.createStructType(*fields*) *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*)

To work with the correct **Python types**, use the following:

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

Data type	Value type in Python	API to access or create a data type
ByteType	int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of –128 to 127.	ByteType()
ShortType	int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of –32768 to 32767.	ShortType()
IntegerType	int or long. Note: Python has a lenient definition of “integer.” Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It’s best practice to use LongType.	IntegerType()
LongType	long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of –9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType.	LongType()

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()

Spark Schema:

- Spark schema is the structure of the DataFrame or Dataset, we can define itusing StructType class which is a collection of StructField that define the column name(String), column type (DataType), nullable column (Boolean) and metadata (MetaData)
- You Can Define Schemas Manually Or Read A Schema From A Data Source (Often Called Schema On Read). Schemas Consist Of Types, Meaning That You Need A Way Of Specifying What Lies Where.

  - The `org.apache.spark.sql.types` package must be imported to access `StructType`, `StructField`, `IntegerType`, and `StringType`.
  - The `schema()` method returns a `StructType` object:

```
df.schema

StructType(StructField(number,IntegerType,true),
StructField(word,StringType,true)
)
```

StructField

StructFields model each column in a DataFrame.

- StructField objects are created withthe name, dataType, and nullable properties. Here's an example:

```
Eg: StructField("word", StringType, true)
```

- The StructField above sets the name field to "word", the dataType field to StringType, and the nullable field to true.
- "word" is the name of the column in the DataFrame.
- StringType means that the column can only take string values like "hello" – it cannot take other values like 34 or false.
- When the nullable field is set to true, the column can accept null values.

Spark Column:

- Spark withColumn() is a DataFrame function that is used to add a new column to DataFrame, change the value of an existing column, convert the datatype of a column, derive a new column from an existing column.

- Columns in Spark can be accessed using the column name or by using the col() function from the pyspark.sql.functions module.

Spark withColumn() Syntax and Usage

- Spark withColumn() is a transformation function of DataFrame that is used to manipulate the column values of all rows or selected rows on DataFrame.
- withColumn() function returns a new Spark DataFrame after performing operations like adding a new column, update the value of an existing column, derive a new column from an existing column, and many more.

Below is a syntax of withColumn() function.

withColumn(colName : String, col : Column) : DataFrame

colName:String – specify a new column you wanted to create. use an existing column to update the value.
col:Column – column expression.
Since withColumn() is a transformation function it doesn't execute until action is called.

Data Frame Transformations:

- In Apache Spark, DataFrame transformations are operations that modify or manipulate the data within a DataFrame.
- These transformations are typically performed using the DataFrame API and allow you to apply various operations to modify the structure, filter rows, aggregate data, join datasets, and more.
- Transformations in Spark are lazy evaluated, meaning that they don't execute immediately but instead build up a plan for execution.

Here are some commonly used DataFrame transformations in Spark:

- **select()**: Selects specific columns from a DataFrame or creates new columns based on existing columns. It allows you to specify the columns to include in the result DataFrame.

```
df.select("name", "age")
```

- **filter()**: Filters rows of a DataFrame based on a condition. It returns a new DataFrame containing only the rows that satisfy the specified condition.

```
df.filter(df["age"] > 25)
```

- **where()**: An alternative to **filter()**, it filters rows based on a condition and returns a new DataFrame.

  df.where(df["age"] > 25)

- **groupBy()**: Groups the DataFrame based on one or more columns and allows you to perform aggregation operations on the grouped data.

  df.groupBy("city").agg({"age": "avg"})

- **orderBy()**: Orders the rows of a DataFrame based on one or more columns in ascending or descending order.

  df.orderBy(df["age"].desc())

- **join()**: Joins two DataFrames together based on a join condition.

  df1.join(df2, df1["id"] == df2["id"], "inner")

- **distinct()**: Returns a new DataFrame with unique rows by eliminating duplicates.

  df.distinct()

- **drop()**: Drops one or more columns from the DataFrame and returns a new DataFrame without those columns.

  df.drop("age")

Working with different types of data :

- Working with different types of data often involves understanding the data format, processing it appropriately, and leveraging the right tools or libraries to handle specific data types. Here are some common types of data and considerations for working with them:
- **Text Data**:

- Text data, such as documents or textual datasets, can be processed using natural language processing (NLP) techniques .
- Tasks like text cleaning, tokenization, stemming, and sentiment analysis are commonly performed on text data.
- Regular expressions can be used for pattern matching and extraction of specific information from text.

- **Numeric Data**:

- Numeric data includes continuous or discrete numerical values.
- Statistical analysis, data visualization, and machine learning algorithms are often applied to numeric data.
- Libraries like NumPy and pandas in Python provide powerful tools for numerical data manipulation and analysis.

➢ **Time Series Data**:

- Time series data represents observations recorded at specific time intervals.
- Libraries like pandas and NumPy offer functionality for handling time series data, including time-based indexing, resampling, and time-based calculations.
- Time series analysis techniques, such as forecasting, seasonality detection, and trend analysis, are commonly used for this type of data.

➢ **Categorical Data**:

- Categorical data represents discrete values or categories.
- Techniques like encoding, label encoding, and ordinal encoding can be used to represent categorical data numerically.

➢ **Image Data**:

- Image data consists of visual information represented in pixel values.
- Libraries like OpenCV offer image processing capabilities, such as image resizing, cropping, filtering, and feature extraction.
- Deep learning frameworks like TensorFlow or PyTorch are often used for tasks like image classification, object detection, and image generation.

➢ **Spatial Data**:

- Spatial data represents information about physical locations or geographic features.
- Libraries like GeoPandas or ArcPy (for ArcGIS) provide tools for spatial data manipulation, geocoding, spatial querying, and spatial analysis.
- Visualization libraries like Folium or Matplotlib can help in creating maps and spatial visualizations.

➢ **Structured Data**:

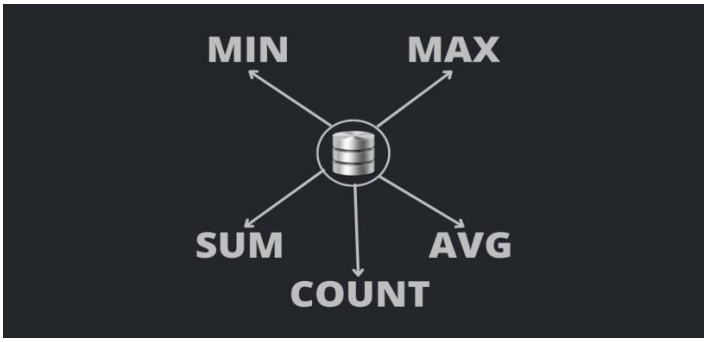
- Structured data is organized in a tabular format with rows and columns, similar to a database table or a spreadsheet.
- Libraries like pandas and Apache Spark's DataFrame API offer comprehensive capabilities for working with structured data, including filtering, aggregation, joining, and transformations.

➢ **Unstructured Data**:

- Unstructured data refers to data that does not have a predefined structure, such as text documents, social media posts, or multimedia files.

- NLP techniques, text mining, or deep learning models can be applied to extract meaningful insights from unstructured data.
- the specific tools and techniques for working with different data types may vary based on the programming language or framework being used. It's important to identify the data type, understand the characteristics of the data, and choose appropriate methods and libraries accordingly to effectively process and analyze the data.

Aggregations- Aggregation Functions:



- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- In data analysis and processing, aggregations refer to the process of summarizing or combining data across multiple rows or groups.
- Aggregations allow you to derive insights from data by computing metrics such **as counts, sums, averages, maximums, minimums,** and more.
- In Apache Spark, you can perform aggregations on DataFrames or Datasets using the DataFrame API or Spark SQL.

Here are some common aggregation operations in Spark:

Aggregate Functions Examples: Eg: First Create a data frame

```
importspark.implicits._

valsimpleData = Seq(("James", "Sales", 3000),
   ("Michael", "Sales", 4600),
   ("Robert", "Sales", 4100),
   ("Maria", "Finance", 3000),
   ("James", "Sales", 3000),
   ("Scott", "Finance", 3300),
   ("Jen", "Finance", 3900),
   ("Jeff", "Marketing", 3000),
   ("Kumar", "Marketing", 2000),
   ("Saif", "Sales", 4100)
  )
valdf = simpleData.toDF("employee_name", "department", "salary")
df.show()
```

Output:

```
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|        James|     Sales| 3000|
|      Michael|     Sales| 4600|
|       Robert|     Sales| 4100|
|        Maria|   Finance|  3000|
|        James|     Sales|  3000|
|        Scott|   Finance|  3300|
|          Jen|   Finance|  3900|
|         Jeff| Marketing|  3000|
|        Kumar| Marketing|  2000|
|         Saif|     Sales|  4100|
+-------------+----------+------+
```

- **Count():** count() function returns number of elements in a column.

```
println("count: "+
df.select(count("salary")).collect()(0))

Prints county: 10
```

- **Sum():sum() function Returns the sum of all values in column.**

```
df.select(sum("salary")).show(false)

+-----------+
|sum(salary)|
+-----------+
|34000      |
+-----------+
```

- **avg():**avg() function returns the average of values in the input column.
- Output:

```
//avg
println("avg: "+
df.select(avg("salary")).collect()(0)(0))

//Prints avg: 3400.0
```

- **min():**Finds the minimum value in a column.
- Output:

```
df.select(min("salary")).show(false)

+-----------+
|min(salary)|
+-----------+
|2000       |
+-----------+
```

- **max():** function returns the maximum value in a column.
  Output:

```
df.select(max("salary")).show(false)

+-----------+
|max(salary)|
+-----------+
|4600       |
+-----------+
```

User-defined aggregate functions:

- User-defined aggregate functions (UDAFs) are user-programmable routines that act on multiple rows at once and return a single aggregated value as a result.

Aggregator

Syntax Aggregator[-IN, BUF, OUT]

- A base class for user-defined aggregations, which can be used in Dataset operations to take all of the elements of a group and reduce them to a single value.
  - **IN**: The input type for the aggregation.
  - **BUF**: The type of the intermediate value of the reduction.

- **OUT**: The type of the final output result.
- **bufferEncoder: Encoder[BUF]**

  The Encoder for the intermediate value type.

- **finish(reduction: BUF): OUT**

  Transform the output of the reduction.

- **merge(b1: BUF, b2: BUF): BUF**

  Merge two intermediate values.

- **outputEncoder: Encoder[OUT]**

  The Encoder for the final output value type.

- **reduce(b: BUF, a: IN): BUF**
- Aggregate input value a into current intermediate value. For performance, the function may modify b and return it instead of constructing new object for b.
- **zero: BUF**

The initial value of the intermediate result for this aggregation.

Joins-Inner Joins, Outer Joins:

JOINS:

- As the name indicates, JOIN means to combine something. In case of SQL, JOIN means "to combine two or more tables".
- In SQL, JOIN clause is used to combine the records from two or more tables in a database.
- Spark DataFrame supports all basic SQL Join Types like INNER, LEFT OUTER, RIGHT OUTER, LEFT ANTI, LEFT SEMI, CROSS, SELF JOIN.

Example of Data Creation

We will use the following data to demonstrate the different types of joins:

Book Dataset:

```
case class Book(book_name: String, cost: Int, writer_id:Int)
val bookDS = Seq(
Book("Scala", 400, 1),
Book("Spark", 500, 2),
Book("Kafka", 300, 3),
Book("Java", 350, 5)
).toDS()
bookDS.show()
```

```
+---------+----+---------+
|book_name|cost|writer_id|
+---------+----+---------+
|    Scala| 400|        1|
|    Spark| 500|        2|
|    Kafka| 300|        3|
|     Java| 350|        5|
+---------+----+---------+
```

Writer Dataset:

```
case class Writer(writer_name: String, writer_id:Int)
val writerDS = Seq(
Writer("Martin",1),
Writer("Zaharia " 2),
Writer("Neha", 3),
Writer("James", 4)
).toDS()
writerDS.show()
```

```
+-----------+---------+
|writer_name|writer_id|
+-----------+---------+
|     Martin|        1|
|    Zaharia|        2|
|       Neha|        3|
|      James|        4|
+-----------+---------+
```

1. INNER JOIN

- The INNER JOIN returns the dataset which has the rows that have matching values in both the datasets i.e. value of the common field will be the same.

val BookWriterInner = bookDS.join(writerDS, bookDS("writer_id") === writerDS("writer_id"), "inner")BookWriterInner.show()

```
+---------+----+---------+-----------+---------+
|book_name|cost|writer_id|writer_name|writer_id|
+---------+----+---------+-----------+---------+
|    Scala| 400|        1|     Martin|        1|
|    Spark| 500|        2|    Zaharia|        2|
|    Kafka| 300|        3|       Neha|        3|
```

2. OUTER JOIN:

Operation: In join some records are missing, if we want that missing records than we have to use outer join.

Types: Three types of Outer Join

- Left Outer Join
- Right Outer Join
- Full Outer Join

➢ LEFT OUTER JOIN

- The LEFT OUTER JOIN returns the dataset that has all rows from the left dataset, and the matched rows from the right dataset.

Syntax: val BookWriterLeft = bookDS.join(writerDS, bookDS("writer_id") ===writerDS("writer_id"), "leftouter")BookWriterLeft.show()

```
+---------+----+---------+-----------+---------+
|book_name|cost|writer_id|writer_name|writer_id|
+---------+----+---------+-----------+---------+
|    Scala| 400|        1|     Martin|        1|
|    Spark| 500|        2|    Zaharia|        2|
|    Kafka| 300|        3|       Neha|        3|
|     Java| 350|        5|       null|     null|
+---------+----+---------+-----------+---------+
```

➢ RIGHT OUTER JOIN

- The RIGHT OUTER JOIN returns the dataset that has all rows from the right dataset, and the matched rows from the left dataset.

Syntax: val BookWriterRight = bookDS.join(writerDS, bookDS("writer_id") ===writerDS("writer_id"), "rightouter")BookWriterRight.show()

```
+---------+----+---------+-----------+---------+
|book_name|cost|writer_id|writer_name|writer_id|
+---------+----+---------+-----------+---------+
|    Scala| 400|        1|     Martin|        1|
|    Spark| 500|        2|    Zaharia|        2|
|    Kafka| 300|        3|       Neha|        3|
|     null|null|     null|      James|        4|
+---------+----+---------+-----------+---------+
```

➢ FULL OUTER JOIN

- The FULL OUTER JOIN returns the dataset that has all rows when there is a match in either the left or right dataset.

Syntax: val BookWriterFull = bookDS.join(writerDS, bookDS("writer_id") === writerDS("writer_id"), "fullouter")BookWriterFull.show()

example:

```
+---------+----+---------+-----------+---------+
|book_name|cost|writer_id|writer_name|writer_id|
+---------+----+---------+-----------+---------+
|    Scala| 400|        1|     Martin|        1|
|    Kafka| 300|        3|       Neha|        3|
|     Java| 350|        5|       null|     null|
|     null|null|     null|      James|        4|
|    Spark| 500|        2|    Zaharia|        2|
+---------+----+---------+-----------+---------+
```

Comma-separated values (CSV) File:

- CSV is a text file format that uses commas to separate values. A CSV file stores tabular data (numbers and text) in plain text, where each line of the file typically represents one data record.
- Each record consists of the same number of fields, and these are separated by commas in the CSV file.

Processing CSV Files, JSON Files, Text Files and Parquet Files:

- Spark SQL provides **spark.read.csv("path")** to read a CSV file into Spark DataFrame and **dataframe.write.csv("path")** to save or write to the CSV file. Spark supports reading pipe, comma, tab, or any other delimiter/seperator files.
- Here we will look into how to read a single file, multiple files, all files from a local directory into DataFrame, and applying some transformations finally writing DataFrame back to CSV file using Scala.

Spark Read CSV file into DataFrame:

- Using spark.read.csv("path") or spark.read.format("csv").load("path") you can read a CSV file with fields delimited by pipe, comma, tab (and many more) into a Spark DataFrame, These methods take a file path to read from as an argument.

```
val df = spark.read.csv("src/main/resources/zipcodes.csv")
   df.printSchema()
```

- This example reads the data into DataFrame columns "_c0" for the first column and "_c1" for second and so on. and by default type of all these columns would be String.

```
root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
```

- If you have a header with column names on file, you need to explicitly specify true for header option using option("header",true) not mentioning this, the API treats the header as a data record.

```
val df = spark.read.option("header",true)
  .csv("src/main/resources/zipcodes.csv")
```

- It also reads all columns as a string (**StringType**) by default.
- When you use **format("csv")** method, you can also specify the Data sources by their fully qualified name (i.e., **org.apache.spark.sql.csv**), but for built-in sources, you can also use their short names (**csv,json, parquet, jdbc, text** e.t.c)

Read multiple CSV files

- Using the spark.read.csv() method you can also read multiple CSV files, just pass all file names by separating comma as a path, for example :

```
val df = spark.read.csv("path1,path2,path3")
```

Read all CSV files in a directory

- We can read all CSV files from a directory into DataFrame just by passing the directory as a path to the csv() method.

```
val df = spark.read.csv("Folder path")
```

JSON (JavaScript Object Notation) File:

- JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values).
- It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.

Spark Read and Write JSON file into DataFrame:

- Spark SQL provides spark.read.json("path") to read a single line and multiline (multiple lines) JSON file into Spark DataFrame and dataframe.write.json("path") to save or write to JSON file, In this we will look into how to read a single file, multiple files, all files from a directory into DataFrame and writing DataFrame back to JSON file using Scala example.

1. Spark Read JSON File into DataFrame

- Using spark.read.json("path") or spark.read.format("json").load("path") you can read a JSON file into a Spark DataFrame, these methods take a file path as an argument.

```
//read json file into dataframe
val df = spark.read.json("src/main/resources/zipcodes.json")
df.printSchema()
df.show(false)
```

- When you use format("json") method, you can also specify the Data sources by their fully qualified name (i.e., org.apache.spark.sql.json), for built-in sources, you can also use short name “json”

2. Read JSON file from multiline

- Sometimes you may want to read records from JSON file that scattered multiple lines, In order to read such files, use-value true to multiline option, by default multiline option, is set to false.

Using spark.read.option("multiline","true")

```
//read multiline json file
val multiline_df = spark.read.option("multiline","true")
    .json("src/main/resources/multiline-zipcode.json")
multiline_df.show(false)
```

3. Reading Multiple Files at a Time

Using the spark.read.json() method you can also read multiple JSON files from different paths, just pass all file names with fully qualified paths by separating comma, for example

```
//read multiple files
val df2 = spark.read.json(
      "src/main/resources/zipcodes_streaming/zipcode1.json",
```

```
      "src/main/resources/zipcodes_streaming/zipcode2.json")
df2.show(false)
```

4. Reading all Files in a Directory

- We can read all JSON files from a directory into DataFrame just by passing directory as a path to the json() method. Below snippet, "zipcodes_streaming" is a folder that contains multiple JSON files.

```
//read all files from a folder
val df3 =
spark.read.json("src/main/resources/zipcodes_streaming")
df3.show(false)
```

Write Spark DataFrame to JSON file

Use the Spark DataFrameWriter object "write" method on DataFrame to write a JSON file.

```
df2.write
 .json("/tmp/spark_output/zipcodes.json")
```

Parquet File:

- Apache Parquet is a columnar file format that provides optimizations to speed up queries and is a far more efficient file format than CSV or JSON, supported by many data processing systems.
- It is compatible with most of the data processing frameworks in the Hadoop echo systems.
- It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.
- Spark SQL provides support for both reading and writing Parquet files that automatically capture the schema of the original data.

- It also reduces data storage by 75% on average. Below are some advantages of storing data in a parquet format. Spark by default supports Parquet in its library hence we don't need to add any dependency libraries.

Apache Parquet Advantages:

- Below are some of the advantages of using Apache Parquet. combining these benefits with Spark improves performance and gives the ability to work with structure files.

  - **Reduces IO operations.**
  - **Fetches specific columns that you need to access.**
  - **It consumes less space.**
  - **Support type-specific encoding.**

first, let's Create a Spark DataFrame from Seq object.

```
val data = Seq(("James ","","Smith","36636","M",3000),
               ("Michael ","Rose","","40288","M",4000),
               ("Robert ","","Williams","42114","M",4000),
               ("Maria ","Anne","Jones","39192","F",4000),
               ("Jen","Mary","Brown","","F",-1))

val columns =
Seq("firstname","middlename","lastname","dob","gender","salary")

import spark.sqlContext.implicits._
val df = data.toDF(columns:_*)
```

- The above example creates a data frame with columns "firstname", "middlename", "lastname", "dob", "gender", "salary"

Spark Write DataFrame to Parquet file format

- Using parquet() function of DataFrameWriter class, we can write Spark DataFrame to the Parquet file.
- As mentioned Spark doesn't need any additional packages or libraries to use Parquet as it by default provides with Spark.
- In this example, we are writing DataFrame to "people.parquet" file.

```
df.write.parquet("/tmp/output/people.parquet")
```

- Writing Spark DataFrame to Parquet format preserves the column names and data types, and all columns are automatically converted to be nullable for compatibility reasons. Notice that all part files Spark creates has parquet extension.

Spark Read Parquet file into DataFrame

- DataFrameReader provides parquet() function (spark.read.parquet) to read the parquet files and creates a Spark DataFrame.
- In this example snippet, we are reading data from an apache parquet file we have written before.

```
val parqDF = spark.read.parquet("/tmp/output/people.parquet")
```

printing schema of DataFrame returns columns with the same names and data types.

Append to existing Parquet file

- Spark provides the capability to append DataFrame to existing parquet files using “append” save mode. In case, if you want to overwrite use “overwrite” save mode.

```
df.write.mode('append').parquet("/tmp/output/people.parquet")
```

SPARK SQL:

- Spark SQL is a Spark module for structured data processing.
- It provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine.
- It enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data.
- It also provides powerful integration with the rest of the Spark ecosystem (e.g., integrating SQL query processing with machine learning).

Spark SQL will allow developers to:

- Import relational data from Parquet files and Hive tables
- Run SQL queries over imported data and existing RDDs
- Easily write RDDs out to Hive tables or Parquet files

Below are the Links for your Reference (using spark & python)

Reading CSV File, JSON File, Parquet file using Spark:

https://sparkbyexamples.com/spark/spark-read-csv-file-into-dataframe/
https://sparkbyexamples.com/spark/spark-read-and-write-json-file/
https://sparkbyexamples.com/spark/spark-read-write-dataframe-parquet-example/

Reading CSV File, JSON File, Parquet file using PYTHON:
https://data-flair.training/blogs/python-data-file-formats/