Structured API in Spark with Focus on Scala

Presented by

Pavithra Purushothaman
Logeshwari Purushothaman

Professor

Dr. Christoph Scheich
Hochschule Fulda

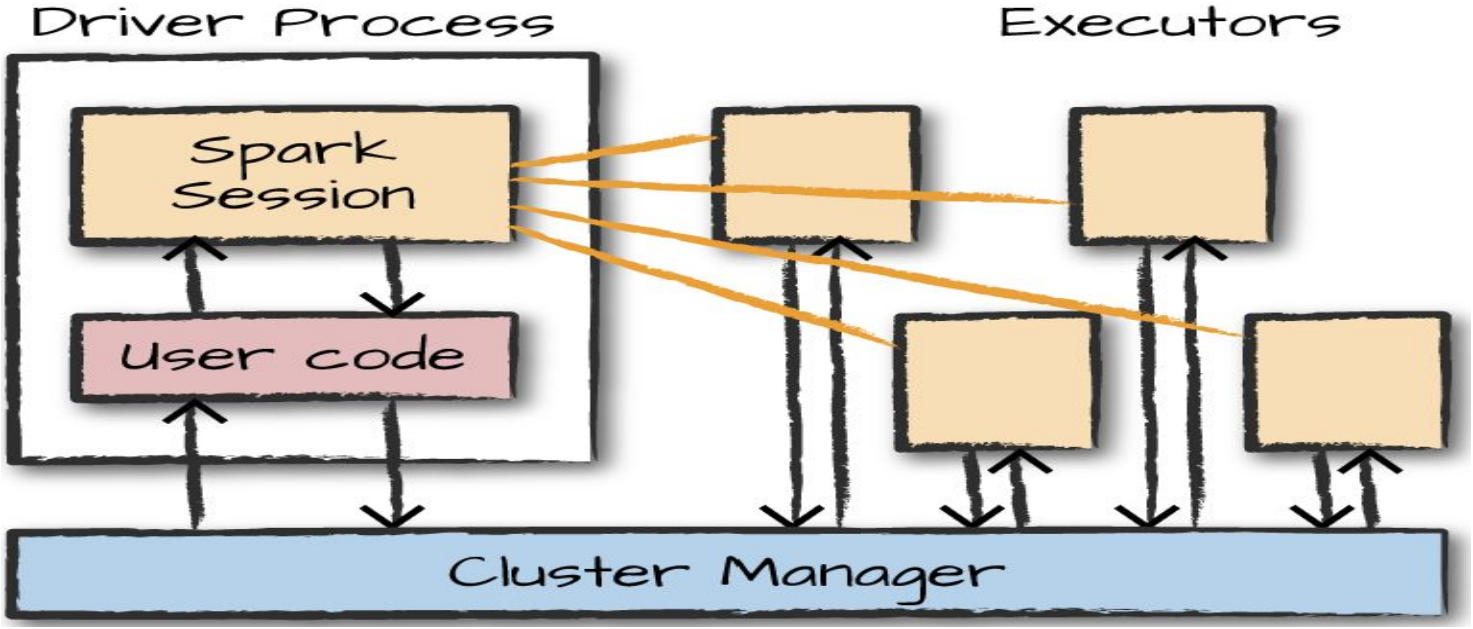
Agenda

1. **Recap - Installation & Spark Applications**
2. Spark Session
3. Introduction to Structured API
4. Spark Query Lifecycle
5. DataFrames vs Datasets vs RDDs
6. Transformations and Actions
7. Persistence (Caching)
8. Double RDD in Spark
9. Pair RDD (Key-Value RDD)
10. Conclusion

1. Recap - Installation

1. **Install via Package Manager:** Use Homebrew (macOS), APT (Ubuntu), or YUM (CentOS) for quick installation.
2. **Manual Installation:** Download the Spark tarball from the Apache Spark website.
3. **Extract the Tarball:** Unzip the downloaded .tgz file to your preferred directory.
4. **Set Environment Variables:** Update .bash_profile or .zshrc with SPARK_HOME and PATH.
5. **Verify Installation:** Run spark-shell to ensure Spark is correctly installed.

Recap - Spark Applications



Agenda

1. Recap - Installation & Spark Applications
2. **Spark Session**
3. Introduction to Structured API
4. Spark Query Lifecycle
5. DataFrames vs Datasets vs RDDs
6. Transformations and Actions
7. Persistence (Caching)
8. Double RDD in Spark
9. Pair RDD (Key-Value RDD)
10. Conclusion

2. Spark Session

SparkSession is the entry point to interact with the Spark framework and serves as a unified interface for programming Spark with both structured and unstructured APIs. It simplifies the process of setting up Spark contexts and configurations, making it easier to manage and run Spark applications.

Capabilities:

- Reading and Writing Data: The SparkSession provides methods to load and save data in various formats.
- Executing SQL Queries: You can register DataFrames as temporary views and run SQL queries.
- Creating DataFrames: You can create a DataFrame directly from a collection or RDD.
- Managing Spark Context: The SparkSession internally manages the SparkContext, so you don’t need to explicitly create one.

```
import org.apache.spark.sql.SparkSession

// Create SparkSession with custom configurations
val spark = SparkSession.builder()
  .appName("Custom Spark Session") // Set the application name
  .config("spark.ui.enabled", "true") // Enable Spark UI
  .config("spark.ui.port", "4041") // Set a custom port for the Spark UI
  .config("spark.executor.memory", "2g") // Set executor memory to 2GB
  .config("spark.driver.memory", "2g") // Set driver memory to 2GB
  .config("spark.sql.shuffle.partitions", "50") // Set number of shuffle partitions
  .getOrCreate()

// Verify SparkSession is created
println(s"SparkSession created: ${spark.version}")

// Show Spark UI link
println("Spark UI is available at http://localhost:4041")

// Sample DataFrame operation to see it in action
val df = spark.read.option("header", "true").csv("/Users/pavithrapurushothaman/Documents/bigdata/people.csv")


df.show()
```

Agenda

1. Recap - Installation & Spark Applications
2. Spark Session
3. **Introduction to Structured API**
4. Spark Query Lifecycle
5. DataFrames vs Datasets vs RDDs
6. Transformations and Actions
7. Persistence (Caching)
8. Double RDD in Spark
9. Pair RDD (Key-Value RDD)
10. Conclusion

3. Introduction to Structured API

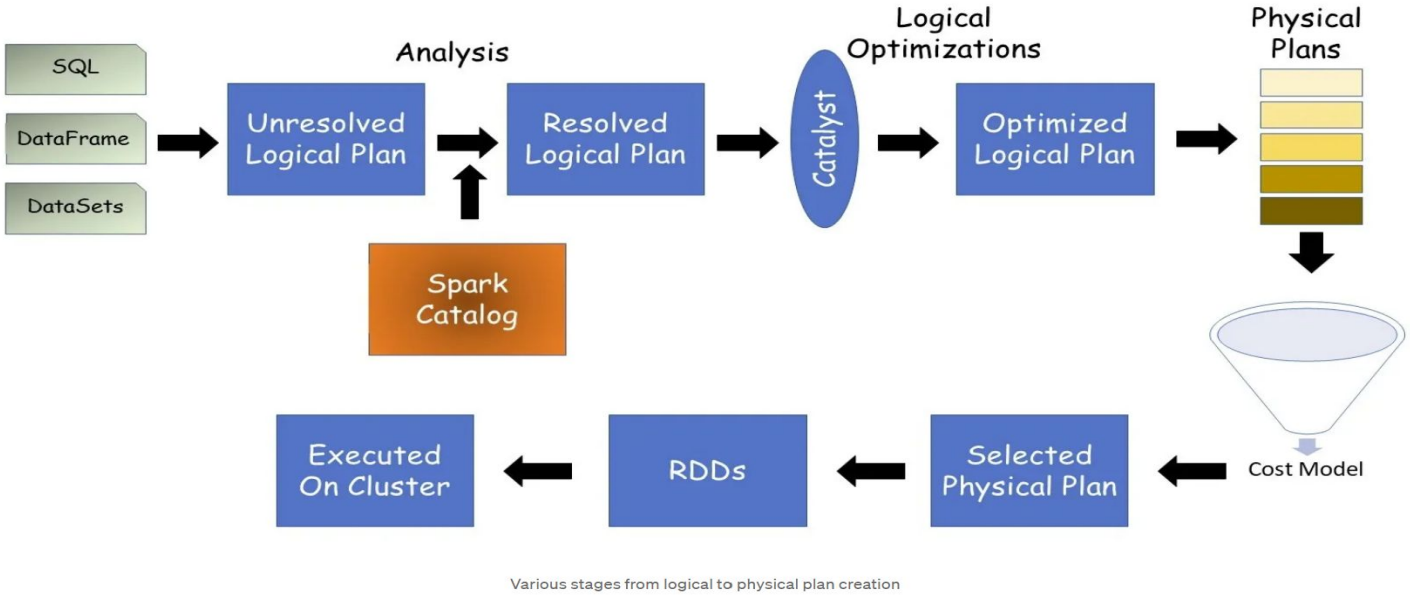
Structured APIs in Spark provide a **high-level abstraction** for data processing, allowing users to work with **structured data** efficiently. They include **DataFrames, Datasets, and SQL tables/views**. These APIs are designed to handle both **batch and streaming computations** seamlessly, making it easier to transition between the two without significant changes to the code.

Other Benefits -Optimized by Catalyst, Interoperability,Ease of Use,Declarative

Agenda

1. Recap - Installation & Spark Applications
2. Spark Session
3. Introduction to Structured API
4. **Spark Query Lifecycle**
5. DataFrames vs Datasets vs RDDs
6. Transformations and Actions
7. Persistence (Caching)
8. Double RDD in Spark
9. Pair RDD (Key-Value RDD)
10. Conclusion

4. Spark Query Life Cycle



Various stages from logical to physical plan creation

https://medium.com/nerd-for-tech/apache-sparks-logical-and-physical-plans-using-explain-method-2487b9be537

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.SparkSession

// Create SparkSession
val spark = SparkSession.builder()
  .appName("EXPLAIN Query Lifecycle Example")
  .config("spark.ui.enabled", "true")
  .config("spark.ui.port", "4041")
  .getOrCreate()

// Sample DataFrame operation
val df = spark.read.option("header", "true").csv("""C:\Users\loges\Documents\Semester_1\advance_bigdata\spark_presentation\people.csv""")

// Apply some transformations
val result = df.filter(df("age") > 18)
  .groupBy("Name")
  .count()

// Show the plan using EXPLAIN to understand the query lifecycle
println("Query Plan with EXPLAIN:")

// Unoptimized logical plan
result.explain(true)  // "true" gives detailed plan, including optimizations

// Show the result
result.show()

// Exiting paste mode, now interpreting.

25/01/16 15:51:28 WARN SparkSession: Using an existing Spark session; only runtime SQL configurations will take effect.
Query Plan with EXPLAIN:
== Parsed Logical Plan ==
Aggregate [Name#41], [Name#41, count(1) AS count#49L]
+- Filter (cast(age#42 as int) > 18)
   +- Relation [Name#41,Age#42] csv

== Analyzed Logical Plan ==
Name: string, count: bigint
Aggregate [Name#41], [Name#41, count(1) AS count#49L]
+- Filter (cast(age#42 as int) > 18)
   +- Relation [Name#41,Age#42] csv

== Optimized Logical Plan ==
Aggregate [Name#41], [Name#41, count(1) AS count#49L]
+- Project [Name#41]
   +- Filter (isnotnull(age#42) AND (cast(age#42 as int) > 18))
      +- Relation [Name#41,Age#42] csv

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[Name#41], functions=[count(1)], output=[Name#41, count#49L])
   +- Exchange hashpartitioning(Name#41, 200), ENSURE_REQUIREMENTS, [plan_id=53]
      +- HashAggregate(keys=[Name#41], functions=[partial_count(1)], output=[Name#41, count#53L])
         +- Project [Name#41]
            +- Filter (isnotnull(age#42) AND (cast(age#42 as int) > 18))
               +- FileScan csv [Name#41,Age#42] Batched: false, DataFilters: [isnotnull(Age#42), (cast(Age#42 as int) > 18)], Format: CSV, Location: InMemoryFileIndex(1 paths)[file:/C:/Users/loges/Documents/Semester_1/advance_bigdata/spark_presen..., Partitio
nFilters: [], PushedFilters: [IsNotNull(Age)], ReadSchema: struct<Name:string,Age:string>

+------+-----+
|  Name|count|
+------+-----+
|Anitha|    1|
|  Loge|    1|
|  Pavi|    1|
+------+-----+
```

Agenda

1. Recap - Installation & Spark Applications
2. Spark Session
3. Introduction to Structured API
4. Spark Query Lifecycle
5. **DataFrames vs Datasets vs RDDs**
6. Transformations and Actions
7. Persistence (Caching)
8. Double RDD in Spark
9. Pair RDD (Key-Value RDD)
10. Conclusion

5. DataFrames vs Datasets vs RDDs

DataFrames: DataFrames are represented as distributed collections of data organized into named columns, similar to an SQL table. They are visualized as tables with rows and columns, where each row represents a record and each column represents a field.

Datasets: Datasets are similar to DataFrames but provide an additional layer of type safety. Datasets are represented as distributed collections of objects, where each object is an instance of a user-defined class. It can be visualized as a collection of objects, each containing fields with specific data types. They combine the benefits of RDDs and DataFrames, providing both type safety and performance optimization.

RDDs (Resilient Distributed Dataset): RDDs are represented as distributed collections of elements, partitioned across the nodes of a cluster where each partition contains a subset of the data. RDDs can be created from various data sources, including Hadoop Distributed File System (HDFS), local file systems, or data stored in relational databases.

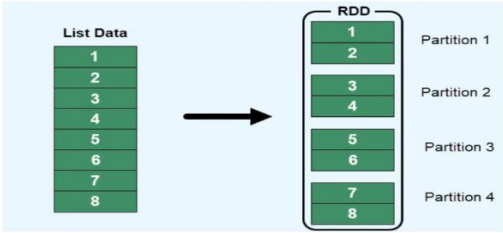
DataFrame

	name	country	age
0	john	USA	23
1	david	UK	45
2	anna	USA	45

Dataset

```
User
{
  userId: Int
  userName: String
}
```

RDD



https://spark.apache.org/docs/latest/sql-programming-guide.html

Now let's create DataFrames, Datasets and RDDs

DataFrame

```
scala> {
     |   import org.apache.spark.sql.SparkSession
     |
     |   val spark = SparkSession.builder
     |     .appName("Create DataFrame")
     |     .master("local")
     |     .getOrCreate()
     |
     |   // Load the CSV file as a DataFrame
     |   val df = spark.read
     |     .option("header", "true") // First row as header
     |     .option("inferSchema", "true") // Automatically infer data types
     |     .csv("/Users/pavithrapurushothaman/Documents/bigdata/people.csv")
     |
     |   // Show the DataFrame
     |   df.show()
     |
     |   // Print schema of the DataFrame
     |   df.printSchema()
     | }
25/01/01 17:18:01 WARN SparkSession: Using an existing Spark session; only runtim
+-------+---+
|   Name|Age|
+-------+---+
|   Pavi| 19|
|   Loge| 20|
| Anitha| 22|
|Jasmine| 16|
|   Luna| 18|
+-------+---+

root
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = true)
```

Dataset

```
scala> // Case class should be defined outside the block

scala> case class Person(Name: String, Age: Int)
defined class Person

scala>

scala> {
     |   import org.apache.spark.sql.SparkSession
     |
     |   // Initialize SparkSession
     |   val spark = SparkSession.builder
     |     .appName("Create Dataset")
     |     .master("local")
     |     .getOrCreate()
     |
     |   // Import Spark implicits
     |   import spark.implicits._
     |
     |   // Load the CSV file as a DataFrame
     |   val df = spark.read
     |     .option("header", "true") // Use the first row as the header
     |     .option("inferSchema", "true") // Automatically infer data types
     |     .csv("/Users/pavithrapurushothaman/Documents/bigdata/people.csv")
     |
     |   // Convert DataFrame to Dataset
     |   val ds = df.as[Person]
     |
     |   // Show the Dataset
     |   ds.show()
     |
     |   // Print the schema of the Dataset
     |   ds.printSchema()
     | }
+-------+---+
|   Name|Age|
+-------+---+
|   Pavi| 19|
|   Loge| 20|
| Anitha| 22|
|Jasmine| 16|
|   Luna| 18|
+-------+---+

root
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = true)
```

RDD

```
scala> {
     |   import org.apache.spark.sql.SparkSession
     |
     |   val spark = SparkSession.builder
     |     .appName("Create RDD")
     |     .master("local")
     |     .getOrCreate()
     |
     |   val sc = spark.sparkContext
     |
     |   // Load the CSV file as an RDD
     |   val rdd = sc.textFile("/Users/pavithrapurushothaman/Documents/bigdata/people.csv")
     |
     |   // Extract header and split rows into a structured RDD
     |   val header = rdd.first()
     |   val dataRDD = rdd.filter(_ != header).map(line => {
     |     val cols = line.split(",")
     |     (cols(0), cols(1).toInt) // Map to (Name, Age)
     |   })
     |
     |   // Print the RDD
     |   dataRDD.collect().foreach(println)
     | }
25/01/01 17:22:02 WARN SparkSession: Using an existing Spark session; only runtime SQL confi
ct.
(Pavi,19)
(Loge,20)
(Anitha,22)
(Jasmine,16)
(Luna,18)
```

Agenda

1. Recap - Installation & Spark Applications
2. Spark Session
3. Introduction to Structured API
4. Spark Query Lifecycle
5. DataFrames vs Datasets vs RDDs
6. **Transformations and Actions**
7. Persistence (Caching)
8. Double RDD in Spark
9. Pair RDD (Key-Value RDD)
10. Conclusion

6. Transformations and Actions

6.1 Transformation - Create new DataFrames/Datasets from existing ones but don't trigger execution.

Transformation	Description
select()	Selects specific columns.
filter()	Filters rows based on a condition.
where()	Similar to filter().
groupBy()	Groups data for aggregation.
orderBy()	Sorts rows.
distinct()	Removes duplicate rows.
drop()	Drops specified columns.
withColumn()	Adds or modifies a column.
join()	Joins two DataFrames/Datasets.
union()	Combines two DataFrames with the same schema.
agg()	Performs aggregate operations.
explode()	Flattens nested structures (e.g., arrays).

https://spark.apache.org/docs/latest/rdd-programming-guide.html

6.2 Actions - Trigger computation and return results.

Action	Description
show()	Displays the first few rows.
collect()	Returns all data as an array (avoid for large datasets).
count()	Returns the number of rows.
take(n)	Returns the first n rows.
first()	Returns the first row.
head()	Similar to first().
foreach()	Applies a function to each row.
write()	Writes data to storage (CSV, JSON, Parquet, etc.).

DataFrame

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.functions._

val df = spark.read.option("header", "true").csv("/Users/pavithrapurush
haman/Documents/bigdata/people.csv")

// Transformation: Convert age to Integer
val dfTyped = df.withColumn("Age", col("Age").cast("int"))

// Transformation: Filter people older than 18
val dfAdults = dfTyped.filter(col("Age") > 18)

// Transformation: Uppercase names
val dfUpper = dfAdults.withColumn("Name", upper(col("Name")))

// Action: Show results
dfUpper.show()

// Exiting paste mode, now interpreting.

+------+---+
|  Name|Age|
+------+---+
|  PAVI| 19|
|  LOGE| 20|
|ANITHA| 22|
+------+---+
```

Transformations Used:

- withColumn() – Adds/modifies a column.
- filter() – Filters rows.
- upper() – Converts text to uppercase.

Actions Used:

- show() – Displays the DataFrame.

RDD

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder()
  .appName("RDD Example")
  .master("local[*]")
  .getOrCreate()

val sc = spark.sparkContext  // Get SparkContext

// Load CSV file as RDD
val rdd = sc.textFile("/Users/pavithrapurushothaman/Documents/bigdata
ple.csv")

// Remove header and split data
val header = rdd.first()
val dataRDD = rdd.filter(_ != header).map(line => {
  val cols = line.split(",")
  (cols(0), cols(1).toInt) // (Name, Age)
})

// Transformation: Filter people above 18
val adultsRDD = dataRDD.filter { case (_, age) => age > 18 }

// Transformation: Map to uppercase names
val upperRDD = adultsRDD.map { case (name, age) => (name.toUpperCase,
) }

// Action: Collect & Print
upperRDD.collect().foreach(println)

// Exiting paste mode, now interpreting.

25/01/11 19:47:47 WARN SparkSession: Using an existing Spark session;
y runtime SQL configurations will take effect.
(PAVI,19)
(LOGE,20)
(ANITHA,22)
```

Transformations Used:

- map() – Converts each row to a new format (uppercase names).
- filter() – Filters people older than 18.

Actions Used:

- collect() – Brings all data to the driver (need to use with caution for large data).
- foreach() – Prints each record.

Dataset

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.{SparkSession, Dataset, Encoders}
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._

// Define the case class for Dataset
case class Person(Name: String, Age: Int)

// Initialize SparkSession
val spark = SparkSession.builder()
  .appName("Dataset Example")
  .master("local[*]")
  .getOrCreate()

import spark.implicits._  // Important for implicit Encoders

// Load CSV into DataFrame
val df = spark.read
  .option("header", "true")  // Ensure the first row is a header
  .option("inferSchema", "true")  // Automatically infer data types
  .csv("/Users/pavithrapurushothaman/Documents/bigdata/people.csv")

// Check DataFrame Schema
df.printSchema()

// Display DataFrame
df.show()

// Convert DataFrame to Dataset
val ds: Dataset[Person] = df.as[Person]  // Strongly typed Dataset

// Show Dataset
ds.show()

// Transformation: Filter people older than 18
val adultsDS = ds.filter(_.Age > 18)

// Display Filtered Dataset
adultsDS.show()

// Exiting paste mode, now interpreting.

25/01/16 00:04:06 WARN SparkSession: Using an existing Spark session; only
fect.
root
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = true)

+-------+---+
|   Name|Age|
+-------+---+
|   Pavi| 19|
|   Loge| 20|
| Anitha| 22|
```

Transformations Used:

- filter() – Uses Scala lambda functions.

Actions Used:

- show() – Displays the Dataset.

Agenda

1. Recap - Installation & Spark Applications
2. Spark Session
3. Introduction to Structured API
4. Spark Query Lifecycle
5. DataFrames vs Datasets vs RDDs
6. Transformations and Actions
7. **Persistence (Caching)**
8. Double RDD in Spark
9. Pair RDD (Key-Value RDD)
10. Conclusion

7. Persistence (Caching)

To avoid computing an RDD multiple times, we can ask Spark to ***persist()*** the data.

1. Persisting helps **avoid recomputing RDDs** multiple times, saving processing time.
2. When persisted, data is **stored on nodes** that compute the RDD.
3. Spark can replicate data across nodes for fault tolerance.
4. If memory is full, Spark uses a **'Least Recently Used' (LRU) policy** to remove old data.
5. You can manually remove data from cache using the unpersist() method.

https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

7.1 Persistence Levels

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.SparkSession
import org.apache.spark.storage.StorageLevel

// Initialize SparkSession
val spark = SparkSession.builder()
  .appName("RDD Persistence Example")
  .master("local[*]")
  .getOrCreate()

val sc = spark.sparkContext  // Get SparkContext

val rdd = sc.textFile("/Users/pavithrapurushothaman/Documents/bigdata/people.csv") // Load Data into RDD

val header = rdd.first()
val dataRDD = rdd.filter(row => row != header) // Remove Header

val peopleRDD = dataRDD.map(line => {
  val cols = line.split(",")
  (cols(0), cols(1).trim.toInt)
}) // Convert to Key-Value Pairs

peopleRDD.persist(StorageLevel.MEMORY_ONLY) // Persist the RDD

val startTime1 = System.currentTimeMillis() // Measure execution time for first run

val adultsRDD = peopleRDD.filter { case (_, age) => age > 18 } // Transformation: Filter Adults (Age > 18)

val result1 = adultsRDD.collect() // Action: Collect and Print
result1.foreach(println)

val endTime1 = System.currentTimeMillis()
println(s"Execution Time WITHOUT Persistence (First Run): ${endTime1 - startTime1} ms")

val startTime2 = System.currentTimeMillis() // Measure execution time for second run (cached data)

val result2 = adultsRDD.collect()  // Using persisted data
result2.foreach(println)

val endTime2 = System.currentTimeMillis()
println(s"Execution Time WITH Persistence (Second Run): ${endTime2 - startTime2} ms")

peopleRDD.unpersist() // Unpersist when no longer needed

// Exiting paste mode, now interpreting.

(Pavi,19)
(Loge,20)
(Anitha,22)
Execution Time WITHOUT Persistence (First Run): 15 ms
(Pavi,19)
(Loge,20)
(Anitha,22)
Execution Time WITH Persistence (Second Run): 12 ms
```

Execution time with and without Persistence (caching)

Agenda

1. Recap - Installation & Spark Applications
2. Spark Session
3. Introduction to Structured API
4. Spark Query Lifecycle
5. DataFrames vs Datasets vs RDDs
6. Transformations and Actions
7. Persistence (Caching)
8. **Double RDD in Spark**
9. Pair RDD (Key-Value RDD)
10. Conclusion

8. DoubleRDD in Spark

A **DoubleRDD** is a special type of RDD in Spark that contains numerical values (Double) and provides additional mathematical operations such as:

- mean()
- sum()
- variance()
- stdev()
- histogram()

DoubleRDD Example

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder()
  .appName("DoubleRDD Example with Rounding")
  .master("local[*]")
  .getOrCreate()

val sc = spark.sparkContext  // Getting SparkContext

// Creating a DoubleRDD
val numbersRDD = sc.parallelize(Seq(10.5, 20.3, 30.7, 40.2, 50.9))

// Performing calculations with rounding
val sum = numbersRDD.sum()
val mean = numbersRDD.mean()
val variance = numbersRDD.variance()
val stdev = numbersRDD.stdev()

// Using BigDecimal to round to 2 decimal places
def round(value: Double, places: Int = 2): Double = BigDecimal(value).setSc
ale(places, BigDecimal.RoundingMode.HALF_UP).toDouble

println(s"Sum: ${round(sum)}")
println(s"Mean: ${round(mean)}")
println(s"Variance: ${round(variance)}")
println(s"Standard Deviation: ${round(stdev)}")

// Exiting paste mode, now interpreting.

Sum: 152.6
Mean: 30.52
Variance: 202.87
Standard Deviation: 14.24
```

Performing mathematical calculations on DoubleRDD and rounding of the result to 2 decimal places.

Agenda

1. Recap - Installation & Spark Applications
2. Spark Session
3. Introduction to Structured API
4. Spark Query Lifecycle
5. DataFrames vs Datasets vs RDDs
6. Transformations and Actions
7. Persistence (Caching)
8. Double RDD in Spark
9. **Pair RDD (Key-Value RDD)**
10. Conclusion

9. Pair RDD (Key-Value RDD)

A **Pair RDD** is an RDD where each element is a **key-value pair** ((K, V)). This is useful for **grouping, sorting, and reducing** operations.

In this example, all the values for the same key is added up.

```
[scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.SparkSession

// Initialize SparkSession
val spark = SparkSession.builder()
  .appName("PairRDD Example")
  .master("local[*]")  // Runs locally with all CPU cores
  .getOrCreate()

// Get SparkContext
val sc = spark.sparkContext

// Creating a Pair RDD (Key-Value Pairs)
val pairRDD = sc.parallelize(Seq(
  ("apple", 3),
  ("banana", 2),
  ("orange", 5),
  ("apple", 1),
  ("banana", 4)
))

val fruitCounts = pairRDD.reduceByKey(_ + _)

// Collect and Print
fruitCounts.collect().foreach(println)

// Stop Spark after computations (to free resources)
[spark.stop()

// Exiting paste mode, now interpreting.

(apple,4)
(banana,6)
(orange,5)
```

9.1 Transformations on Pair RDD

Transformation	Description
reduceByKey(func)	Combines values of the same key.
groupByKey()	Groups values with the same key into an iterable.
mapValues(func)	Applies a function only to the values, keeping the keys unchanged.
flatMapValues(func)	Similar to mapValues, but allows multiple values per key.
sortByKey(ascending)	Sorts key-value pairs based on the key (ascending or descending).
keys()	Returns only the keys from the Pair RDD.
values()	Returns only the values from the Pair RDD.
join(otherRDD)	Joins two Pair RDDs by key, matches keys and merges values.
leftOuterJoin(otherRDD)	Keeps all keys from the left RDD and fills missing values from the right RDD with None.
rightOuterJoin(otherRDD)	Keeps all keys from the right RDD and fills missing values from the left RDD with None.
cogroup(otherRDD)	Groups values of multiple RDDs for the same key.
subtractByKey(otherRDD)	Removes all pairs with keys present in the other RDD.

https://www.oreilly.com/library/view/learning-spark-2nd/9781492050032/

9.2 Aggregations on Pair RDD

Aggregations	Description
countByKey()	Counts the number of values for each key.
foldByKey(zeroValue)(func)	Similar to reduceByKey, but starts with an initial zero value.
aggregateByKey(zeroValue)(seqOp, combOp)	More flexible than reduceByKey as it allows different operations for local and global aggregation.
combineByKey(createCombiner, mergeValue, mergeCombiners)	General aggregation function for Pair RDDs, used for custom aggregations (e.g., calculating average).

Example of Transformations and Aggregations on Pair RDD

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.SparkSession

// Initialize Spark
val spark = SparkSession.builder().appName("PairRDD Example").master("local[*]").getOrCreate()
val sc = spark.sparkContext

// Creating first Pair RDD (product -> price)
val productPrices = sc.parallelize(Seq(
  ("Laptop", 10),
  ("Phone", 20),
  ("Tablet", 30),
  ("Headphones", 40)
))

// Creating second Pair RDD (product -> units sold)
val productSales = sc.parallelize(Seq(
  ("Laptop", 1),
  ("Phone", 2),
  ("Tablet", 3),
  ("Headphones", 4),
  ("Phone", 5)
))

// Transformation: Joining both RDDs on product name
val joinedRDD = productPrices.join(productSales)

// Aggregation: Calculating total revenue per product
val revenueRDD = joinedRDD.mapValues { case (price, quantity) => price * quantity }
                          .reduceByKey(_ + _)

// Collect and Print Results
println("Joined RDD:")
joinedRDD.collect().foreach(println)

println("\nTotal Revenue per Product:")
revenueRDD.collect().foreach(println)

// Stop Spark
spark.stop()

// Exiting paste mode, now interpreting.

Joined RDD:
(Laptop,(10,1))
(Headphones,(40,4))
(Tablet,(30,3))
(Phone,(20,2))
(Phone,(20,5))

Total Revenue per Product:
(Laptop,10)
(Headphones,160)
(Tablet,90)
(Phone,140)
import org.apache.spark.sql.SparkSession
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@5fa9a360
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@7110c51e
```

join() is a **transformation** that merges datasets based on keys (product names).

mapValues() modifies only the **values** without changing keys.

reduceByKey(_ + _) is an **aggregation** that **combines revenues** per product.

Agenda

1. Recap - Installation & Spark Applications
2. Spark Session
3. Introduction to Structured API
4. Spark Query Lifecycle
5. DataFrames vs Datasets vs RDDs
6. Transformations and Actions
7. Persistence (Caching)
8. Double RDD in Spark
9. Pair RDD (Key-Value RDD)
10. **Conclusion**

10. Conclusion

The Structured API in Spark makes working with big data easier by using DataFrames and Datasets, which feel like working with tables in a database. It helps process large amounts of information quickly and efficiently while keeping the code simple and readable. With built-in optimizations, Spark takes care of performance so we can focus on solving problems rather than worrying about how the data is processed. It also supports multiple programming languages, making it flexible for different teams. Overall, Structured API is a powerful way to handle big data without getting lost in complexity.

Any Questions?

Thanks for your time.

Tschüss 👋