**What is Spark Shell?**

* **Spark Shell** is an **interactive REPL (Read-Eval-Print Loop)** for Apache Spark.
* It uses **Scala** by default and lets you test Spark commands instantly.
* It connects automatically to a **SparkContext** (sc) and **SparkSession** (spark in Spark 2.0+).

**RDD Spark Tutorial**

RDD ([Resilient Distributed Dataset)](https://sparkbyexamples.com/spark-rdd-tutorial/)is a fundamental data structure of Spark and it is the primary data abstraction in Apache Spark and the Spark Core. **RDDs are fault-tolerant, immutable distributed collections of objects, which means once** you create an RDD you cannot change it. Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.

**RDD creation**

**RDDs are created primarily in two different ways**, **first parallelizing an existing collection** **and secondly referencing a dataset in an external storage system (HDFS, HDFS, S3 and many more).**

**sparkContext.parallelize()**

sparkContext.parallelize is used to parallelize an existing collection in your driver program. This is a basic method to create RDD.

//Create RDD from parallelize

**val dataSeq = Seq(("Java", 20000), ("Python", 100000), ("Scala", 3000))**

**val rdd=spark.sparkContext.parallelize(dataSeq)**

**sparkContext.textFile()**

Using textFile() method we can read a text (.txt) file from many sources like HDFS, S#, Azure, local e.t.c into RDD.

//Create RDD from external Data source

val rdd2 = spark.sparkContext.textFile("/path/textFile.txt")

**RDD Operations**

On Spark RDD, you can perform **two kinds of operations.**

**RDD Transformations**

[Spark RDD Transformations](https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-transformations/) are lazy operations meaning they don’t execute until you call an action on RDD. Since RDDs are immutable, When you run a transformation(for example map()), instead of updating a current RDD, it returns a new RDD.

Some transformations on RDDs are flatMap(), map(), reduceByKey(), filter(), sortByKey() and all these return a new RDD instead of updating the current.

**RDD Actions**

[RDD Action operation](https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-actions/) returns the values from an RDD to a driver node. In other words, any RDD function that returns non RDD[T] is considered as an action. RDD operations trigger the computation and return RDD in a List to the driver program.

Some actions on RDDs are count(),  collect(),  first(),  max(),  reduce()  and more.

**RDD Lineage**

**What is Lineage?**

* **RDD lineage** refers to the **sequence of transformations** that created an RDD.
* It forms a **DAG (Directed Acyclic Graph)**.
* If a partition of an RDD is lost, Spark **recomputes it** using its lineage.

**Example of Lineage:**

val rdd1 = sc.textFile("data.txt") // 1. Load

val rdd2 = rdd1.flatMap(line => line.split(" ")) // 2. Split words

val rdd3 = rdd2.map(word => (word, 1)) // 3. Map to (word, 1)

val rdd4 = rdd3.reduceByKey(\_ + \_) // 4. Count

val result = rdd4.collect() // 5. Trigger action

Here’s the **lineage DAG**:

textFile("data.txt")

↓

flatMap

↓

map

↓

reduceByKey

↓

collect()

**What is Partitioning?**

Partitioning refers to **how data is split across the cluster**. Spark processes data **partition-by-partition** in parallel.

**Why Partitioning Matters:**

* Controls **data locality**.
* Reduces **data shuffling** in operations like join, groupByKey, etc.
* Enables **parallel processing** for faster computation.

**Persistence and Caching in Spark**

**What is Persistence?**

Persistence is used to **store intermediate RDD/DataFrame results** in memory or disk so they don’t get recomputed.

**Why Persist?**

* Avoid recomputation of lineage.
* Improve performance in iterative algorithms (e.g., ML, GraphX).
* Used when same RDD is accessed **multiple times**.

**What is a DAG in Spark?**

A **DAG (Directed Acyclic Graph)** in Spark represents:

A **logical execution plan** of **RDD transformations**, where each **node is an RDD** and **edges are operations** (like map, filter, etc.).

**DAG Characteristics:**

* **Directed**: Data flows in one direction (from source to result).
* **Acyclic**: No loops/cycles — once a transformation is applied, it moves forward.
* **Graph**: Shows dependencies among RDDs.

**How It Works:**

1. **User writes code with transformations** (e.g. map, filter, reduceByKey)
2. Spark **doesn’t execute immediately** — it builds a **DAG** in memory.
3. When an **action** is called (e.g. collect(), count()), Spark:
   * Submits the DAG to the **DAG Scheduler**
   * Divides it into **stages**
   * Further divides stages into **tasks**
   * Executes tasks in **parallel** using **Task Scheduler**

Example DAG

val rdd1 = sc.textFile("data.txt")

val rdd2 = rdd1.flatMap(\_.split(" "))

val rdd3 = rdd2.map(word => (word, 1))

val rdd4 = rdd3.reduceByKey(\_ + \_)

val rdd5 = rdd4.mapValues(\_ \* 2)

val result = rdd5.collect()

DAG Visualization

textFile("data.txt")

↓

flatMap

↓

map (word, 1)

↓

reduceByKey

↓

mapValues(\*2)

↓

collect()

**DAG vs Lineage**

| **Concept** | **Description** |
| --- | --- |
| **DAG** | Execution plan (logical graph) |
| **Lineage** | History of how an RDD was created |

Lineage is **used for fault recovery**, DAG is used for **execution planning**.

**DAG Execution Flow**

**1. Logical Plan (DAG creation)**

* Built when you chain transformations.

**2. DAG Scheduler**

* Splits the DAG into **stages** based on wide/narrow dependencies.
  + **Narrow dependency**: No shuffle (e.g. map, filter)
  + **Wide dependency**: Shuffle needed (e.g. reduceByKey)

**3. Stage Division**

* Each **stage** contains a **set of tasks** that can be executed in parallel.

**4. Task Scheduler**

* Assigns tasks to executors.

| **Component** | **Role** |
| --- | --- |
| **DAG** | Represents the complete job graph |
| **Stage** | A set of transformations that can be pipelined |
| **Task** | A single unit of execution (on a partition) |
| **Job** | Created for each action |

**RDD Programming Patterns in Scala**

**1. Transformation Patterns**

Transformations are **lazy operations** that define a new RDD from the existing one.

\*map- Apply a function to each elements

val rdd = sc.parallelize(1 to 5)

val squared = rdd.map(x => x \* x)

**\*flatMap-**Split elements into multiple items.

val lines = sc.parallelize(Seq("hello world", "spark scala"))

val words = lines.flatMap(\_.split(" "))

**\*filter -**Filter elements based on a predicate.

val even = rdd.filter(\_ % 2 == 0)

**\*distinct-**Removes duplicates

val distinctRDD = sc.parallelize(Seq(1, 2, 2, 3)).distinct()

\***union, intersection, subtract-** Set operations on RDDs.

val rdd1 = sc.parallelize(1 to 5)

val rdd2 = sc.parallelize(3 to 7)

val union = rdd1.union(rdd2)

val intersection = rdd1.intersection(rdd2)

val diff = rdd1.subtract(rdd2)

2. **Key-Value (Pair RDD) Patterns**

**-** **Key-based transformations and aggregations.**

**\*** **mapToPair / map to tuple**

val pairs = sc.parallelize(Seq("apple", "banana"))

.map(word => (word.length, word))

\***reduceByKey -** Aggregates values with the same key.

val data = sc.parallelize(Seq(("math", 90), ("math", 80), ("eng", 85)))

val totals = data.reduceByKey(\_ + \_)

\***groupByKey -** Groups all values with the same key. (Less efficient)

val grouped = data.groupByKey()

\***sortByKey**

val sorted = data.sortByKey()

**Joins in Spark (Scala)**

**Joins are operations to combine datasets based on a key.**

**1. RDD Joins (Key-Value RDDs)**

**🔸 join – Inner Join-Returns only matching keys from both RDDs.**

**val rdd1 = sc.parallelize(Seq((1, "Alice"), (2, "Bob")))**

**val rdd2 = sc.parallelize(Seq((1, "Math"), (3, "Science")))**

**val joined = rdd1.join(rdd2)**

**joined.collect()**

**// Output: (1,(Alice,Math))**

**🔸 leftOuterJoin**

**val left = rdd1.leftOuterJoin(rdd2)**

**left.collect()**

**// Output: (1,(Alice,Some(Math))), (2,(Bob,None))**

**🔸 rightOuterJoin**

**val right = rdd1.rightOuterJoin(rdd2)**

**right.collect()**

**// Output: (1,(Some(Alice),Math)), (3,(None,Science))**

**🔸 fullOuterJoin**

**val full = rdd1.fullOuterJoin(rdd2)**

**full.collect()**

**// Output: (1,(Some(Alice),Some(Math))), (2,(Some(Bob),None)), (3,(None,Some(Science)))**

**🔸 Important: Joins require partitioning and shuffling.**

**To optimize:**

**rdd1.partitionBy(new HashPartitioner(2)).join(rdd2)**

**Aggregations in Spark**

**1. RDD Aggregations**

**🔸 reduceByKey**

**val scores = sc.parallelize(Seq(("math", 90), ("math", 80), ("eng", 85)))**

**val totals = scores.reduceByKey(\_ + \_)**

**🔸 groupByKey**

**val grouped = scores.groupByKey()**

**⚠️ Not efficient for large shuffles. Prefer reduceByKey or aggregateByKey.**

**🔸 aggregateByKey**

**val marks = sc.parallelize(Seq(("math", 90), ("math", 85), ("eng", 70)))**

**val avg = marks.aggregateByKey((0, 0))(**

**(acc, value) => (acc.\_1 + value, acc.\_2 + 1),**

**(a, b) => (a.\_1 + b.\_1, a.\_2 + b.\_2)**

**)**

**val averages = avg.mapValues { case (sum, count) => sum / count }**

**Aggregations in Spark SQL / DataFrame API**

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

**val df = Seq(**

**("math", 90),**

**("math", 85),**

**("eng", 70)**

**).toDF("subject", "score")**

**// Sum**

**df.groupBy("subject").agg(sum("score").as("total"))**

**// Average**

**df.groupBy("subject").agg(avg("score").as("average"))**

**// Count**

**df.groupBy("subject").agg(count("\*").as("count"))**

**// Multiple Aggregations**

**df.groupBy("subject").agg(**

**count("\*").as("cnt"),**

**sum("score").as("total"),**

**avg("score").as("avg")**

**)**

**What Are Complex Transformations?**

These are combinations or sequences of multiple transformations (e.g., map, filter, flatMap, join, groupByKey, reduceByKey, aggregateByKey, etc.) applied in a functional and efficient way to transform big datasets.

**RDD-Based Complex Transformations**

**🔸 1. Chained Transformations Example**

**val rdd = sc.parallelize(Seq("spark is fun", "scala is powerful", "big data rocks"))**

**val words = rdd**

**.flatMap(\_.split(" ")) // ["spark", "is", "fun", ...]**

**.filter(\_.length > 2) // remove short words**

**.map(word => (word.toLowerCase, 1))**

**.reduceByKey(\_ + \_) // count frequencies**

**words.collect()**

**🔸 2. Nested Joins and Aggregations**

**val students = sc.parallelize(Seq((1, "Alice"), (2, "Bob"), (3, "Charlie")))**

**val scores = sc.parallelize(Seq((1, 80), (2, 90), (1, 70), (3, 85)))**

**val grouped = scores**

**.mapValues(score => (score, 1)) // (id, (score, 1))**

**.reduceByKey { case ((s1, c1), (s2, c2)) => (s1 + s2, c1 + c2) }**

**val avgScores = grouped.mapValues { case (sum, count) => sum / count }**

**val result = students.join(avgScores)**

**result.collect()**

**// Output: (1, (Alice, 75)), (2, (Bob, 90)), (3, (Charlie, 85))**

**🔸 3. Multi-key Aggregation**

**val data = sc.parallelize(Seq(**

**("math", "Alice", 90),**

**("math", "Bob", 80),**

**("science", "Alice", 85),**

**("science", "Bob", 70)**

**))**

**val keyed = data.map { case (subject, student, marks) =>**

**((subject, student), marks)**

**}**

**val total = keyed.reduceByKey(\_ + \_)**

**total.collect()**

**// Output: ((math,Alice),90), ((math,Bob),80), ...**

**DataFrame-Based Complex Transformations**

**import spark.implicits.\_**

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

**val df = Seq(**

**("Alice", "math", 90),**

**("Alice", "science", 85),**

**("Bob", "math", 80),**

**("Bob", "science", 70)**

**).toDF("name", "subject", "score")**

**// Pivot + Aggregation**

**val pivoted = df**

**.groupBy("name")**

**.pivot("subject")**

**.agg(avg("score"))**

**pivoted.show()**

**🔸 4. Window Functions (Advanced)**

**import org.apache.spark.sql.expressions.Window**

**val df = Seq(**

**("Alice", "2023-01-01", 100),**

**("Alice", "2023-01-02", 200),**

**("Bob", "2023-01-01", 50),**

**("Bob", "2023-01-02", 60)**

**).toDF("name", "date", "sales")**

**val windowSpec = Window.partitionBy("name").orderBy("date")**

**val withRunningTotal = df.withColumn("running\_total", sum("sales").over(windowSpec))**

**withRunningTotal.show()**

**Optimization Techniques**

**1. Caching & Persistence**

* **Use cache() or persist() when reusing the same RDD/DataFrame multiple times.**
* **Avoid recomputation.**

**val df = spark.read.csv("big.csv").cache()**

**df.count()**

**df.groupBy("col1").count().show()**

**🔸 Use .persist(StorageLevel.MEMORY\_AND\_DISK) if data is too large for memory.**

**2. Partitioning Strategy**

* **Repartition to increase parallelism:**

**val df2 = df.repartition(8, $"keyColumn")**

* **Coalesce to reduce partitions (less shuffling):**

**val df3 = df.coalesce(2)**

**🔹 Rule of Thumb: Avoid too many small partitions or too few large ones.**

**3. Avoid Wide Transformations (Shuffles)**

* **groupByKey, distinct, join, etc., cause shuffles, which are expensive.**
* **Prefer reduceByKey over groupByKey.**

**// Better:**

**rdd.reduceByKey(\_ + \_)**

**// Avoid:**

**rdd.groupByKey().mapValues(\_.sum)**

**4. Broadcast Joins**

**Use broadcast joins when joining a small dataset with a large one.**

**import org.apache.spark.sql.functions.broadcast**

**val result = largeDF.join(broadcast(smallDF), "id")**

**Reduces data shuffling.**

**5. Pushdown Predicates**

**Let Spark push filters to the source level (like Parquet, JDBC):**

**val df = spark.read**

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

**.parquet("data.parquet")**

**.filter($"status" === "active")**

**6. Column Pruning**

**Read only necessary columns:**

**val df = spark.read.parquet("data.parquet").select("id", "name")**

**Reduces I/O and memory usage.**

**7. Use Efficient File Formats**

**Use Parquet or ORC over CSV/JSON:**

* **Columnar**
* **Compressed**
* **Supports predicate pushdown & schema evolution**

**df.write.parquet("output/")**

**8. Optimize Joins**

| **Join Type** | **Optimization Method** |
| --- | --- |
| **Large + Small** | **Broadcast join** |
| **Large + Large** | **Repartition on join key** |
|  |  |

**// Join optimization example**

**df1.repartition($"key").join(df2.repartition($"key"), "key")**

**9. Use explain() and Spark UI**

**Check logical & physical plans:**

**df.explain(true)**

**Use Spark UI:**

* **Check stages, tasks, memory usage**
* **Identify slow operations or data skew**

**10. Tungsten & Catalyst Optimization (built-in)**

**Spark 2.x+ includes:**

* **Tungsten: memory management & binary processing engine**
* **Catalyst: logical & physical query optimization**

**➡ Just write high-level transformations (don't micro-optimize unless needed).**

**11. Avoid UDFs Unless Necessary**

* **UDFs prevent Catalyst optimization.**
* **Use built-in Spark SQL functions instead:**

**// Instead of this:**

**val myUDF = udf((x: String) => x.toUpperCase)**

**df.withColumn("name", myUDF($"name"))**

**// Use this:**

**df.withColumn("name", upper($"name"))**

**12. Memory & Executor Tuning**

**Set via spark-submit or Spark config:**

| **Config** | **Purpose** |
| --- | --- |
| **spark.executor.memory** | **Executor heap memory** |
| **spark.executor.cores** | **Number of cores per executor** |
| **spark.sql.shuffle.partitions** | **Number of shuffle partitions** |
| **spark.memory.fraction** | **Memory used for execution** |