**Q: We have transformation and actions for RDD. RDD will be formed using SparkContext right? My query is whether these transformations and actions are the same for SparkSession. What is the difference between the context and session? What we have to use for big data which is suitable? Explain briefly with an example**

Yes, you're right! RDDs (Resilient Distributed Datasets) are created using SparkContext; both transformations and actions apply to RDDs. With the introduction of SparkSession in Spark 2.0, some things have changed, but the core principles remain. Let's break this down.

Key Concepts:

1. SparkContext:
   * The SparkContext is the entry point to the original Spark application.
   * It connects to the cluster, manages resources, and creates RDDs.
   * RDDs (low-level API) are created using SparkContext.
   * Transformations and actions for RDDs (e.g., map(), filter(), reduce()) are all triggered via SparkContext.

2.SparkSession:

* The SparkSession was introduced in Spark 2.0 as a unified entry point for Spark applications, replacing both SQLContext and HiveContext.
* It allows you to work with DataFrames, a higher-level abstraction over RDDs (optimized for large-scale structured data). You can also create RDDs through SparkSession.
* SparkSession internally manages the creation of SparkContext and handles SQL queries, DataFrame operations, and streaming tasks.
* It’s more user-friendly and suitable for big data because it optimizes execution plans for large datasets via Catalyst Optimizer and Tungsten Engine.
* **Key Differences:**

| Feature | SparkContext (RDD) | SparkSession (DataFrame, Dataset) |
| --- | --- | --- |
| API Level | Low-level API (RDDs) | High-level API (DataFrames, Datasets) |
| Optimization | No automatic optimization | Optimized using Catalyst Optimizer and Tungsten Engine |
| Data Abstraction | Unstructured, distributed collection of objects | Schema-based (structured) distributed collection of data |
| Entry Point | SparkContext (plus SQLContext and HiveContext in older Spark) | SparkSession (unified entry point for all operations) |
| Ease of Use | More control but more verbose | Easier to use, high-level API |
| Operations | Transformations and actions on RDDs | Transformations and actions on DataFrames or Datasets |
| Suitability | Suitable for low-level operations and fine control | Suitable for big data processing, optimization, and structured data |
| When to Use | For working with custom algorithms or low-level transformations | For handling large-scale structured or semi-structured data, SQL-like queries |

**Q: give full list of transformations and actions for both spark context and session with example for each**

**Transformations and Actions in Spark**

**In Spark, both RDD (SparkContext) and DataFrame (SparkSession) have their own sets of transformations and actions. Below is a comprehensive list of the most commonly used transformations and actions for each, along with examples.**

**1. SparkContext (RDD API)**

**Transformations**

**Transformations are lazy operations that are applied on an RDD to create a new RDD. They do not execute until an action is called.**

| **Transformation** | **Description** | **Example** |
| --- | --- | --- |
| **map()** | **Applies a function to each element in the RDD** | **rdd.map(lambda x: x \* 2)** |
| **filter()** | **Filters elements that meet a condition** | **rdd.filter(lambda x: x > 10)** |
| **flatMap()** | **Similar to map() but flattens nested results** | **rdd.flatMap(lambda x: x.split(" "))** |
| **distinct()** | **Removes duplicates from the RDD** | **rdd.distinct()** |
| **union()** | **Combines two RDDs** | **rdd1.union(rdd2)** |
| **intersection()** | **Returns common elements between two RDDs** | **rdd1.intersection(rdd2)** |
| **groupByKey()** | **Groups values by keys in a pair RDD** | **rdd.groupByKey()** |
| **reduceByKey()** | **Combines values by key using an associative function** | **rdd.reduceByKey(lambda x, y: x + y)** |
| **sortByKey()** | **Sorts an RDD by key** | **rdd.sortByKey()** |
| **join()** | **Joins two RDDs based on keys** | **rdd1.join(rdd2)** |
| **cartesian()** | **Returns the Cartesian product of two RDDs** | **rdd1.cartesian(rdd2)** |
| **coalesce()** | **Reduces the number of partitions** | **rdd.coalesce(2)** |
| **repartition()** | **Increases or decreases the number of partitions** | **rdd.repartition(4)** |
| **sample()** | **Randomly samples a subset of the RDD** | **rdd.sample(False, 0.1)** |
| **pipe()** | **Transforms RDD using external programs** | **rdd.pipe('wc -l')** |

**Actions**

**Actions trigger the execution of transformations and return a result.**

| **Action** | **Description** | **Example** |
| --- | --- | --- |
| **collect()** | **Returns all elements of the RDD** | **rdd.collect()** |
| **count()** | **Returns the number of elements in the RDD** | **rdd.count()** |
| **first()** | **Returns the first element in the RDD** | **rdd.first()** |
| **take(n)** | **Returns the first n elements** | **rdd.take(3)** |
| **reduce()** | **Reduces the RDD to a single element** | **rdd.reduce(lambda x, y: x + y)** |
| **takeSample()** | **Returns a sample of elements from the RDD** | **rdd.takeSample(False, 2)** |
| **foreach()** | **Applies a function to each element (no return)** | **rdd.foreach(lambda x: print(x))** |
| **countByKey()** | **Returns the count of elements for each key** | **rdd.countByKey()** |
| **saveAsTextFile()** | **Saves the RDD to a text file** | **rdd.saveAsTextFile("path")** |
| **saveAsSequenceFile()** | **Saves the RDD as a Sequence file** | **rdd.saveAsSequenceFile("path")** |

**2. SparkSession (DataFrame API)**

**Transformations**

**DataFrame transformations are similar to RDD transformations but are optimized by Spark's Catalyst optimizer.**

| **Transformation** | **Description** | **Example** |
| --- | --- | --- |
| **select()** | **Selects specific columns from the DataFrame** | **df.select("name")** |
| **filter()** | **Filters rows based on a condition** | **df.filter(df["age"] > 18)** |
| **groupBy()** | **Groups rows based on a column** | **df.groupBy("department").agg({"salary": "sum"})** |
| **agg()** | **Applies aggregate functions** | **df.agg({"salary": "max"})** |
| **join()** | **Joins two DataFrames based on a column** | **df1.join(df2, df1["id"] == df2["id"])** |
| **withColumn()** | **Adds or modifies a column in the DataFrame** | **df.withColumn("age\_new", df["age"] + 10)** |
| **orderBy()** | **Sorts rows based on one or more columns** | **df.orderBy("salary", ascending=False)** |
| **drop()** | **Removes one or more columns from the DataFrame** | **df.drop("age")** |
| **distinct()** | **Removes duplicates from the DataFrame** | **df.distinct()** |
| **union()** | **Combines two DataFrames** | **df1.union(df2)** |
| **repartition()** | **Increases or decreases the number of partitions** | **df.repartition(4)** |
| **coalesce()** | **Reduces the number of partitions** | **df.coalesce(2)** |
| **dropDuplicates()** | **Removes duplicate rows** | **df.dropDuplicates()** |
| **sample()** | **Randomly samples a subset of the DataFrame** | **df.sample(0.1)** |
| **pivot()** | **Pivots a DataFrame for group aggregation** | **df.groupBy("year").pivot("month").sum("sales")** |

**Actions**

**DataFrame actions trigger the execution of transformations.**

| **Action** | **Description** | **Example** |
| --- | --- | --- |
| **show()** | **Displays the content of the DataFrame** | **df.show()** |
| **collect()** | **Returns all rows of the DataFrame as a list** | **df.collect()** |
| **count()** | **Returns the number of rows in the DataFrame** | **df.count()** |
| **first()** | **Returns the first row of the DataFrame** | **df.first()** |
| **take(n)** | **Returns the first n rows** | **df.take(3)** |
| **write()** | **Saves the DataFrame to a file** | **df.write.csv("path")** |
| **head()** | **Returns the first n rows as a list** | **df.head(2)** |
| **toPandas()** | **Converts DataFrame to a Pandas DataFrame** | **df.toPandas()** |
| **describe()** | **Generates statistics for numeric columns** | **df.describe().show()** |
| **cache()** | **Caches the DataFrame in memory** | **df.cache()** |
| **unpersist()** | **Removes the DataFrame from memory** | **df.unpersist()** |