RDD - Operations

In this section we will cover:

- RDD – Resilient Distributed Data Set
- DAG – Directed Acyclic Graph
- RDD Persistence

RDD – Resilient Distributed Dataset

- As we know a Spark application consists of a driver program that runs the user's main function and executes various parallel operations on the cluster.
- The main data abstraction Spark provides is a Resilient Distributed Dataset (RDD), which is a collection of elements partitioned (and distributed) across the nodes of the cluster that can be operated on in parallel.
- In Spark all work is expressed as – creating new RDDs, calling operations on RDDs to transform RDDs to generate new ones, and/or to compute a result.
- So, every Spark program and shell session will work as follows:
  - Create some input RDDs from external data.
  - Transform them to define new RDDs using transformations like filter(), flatMap(), map() etc.
  - Ask Spark to persist() any intermediate RDDs that will need to be reused.
  - Launch actions such as count() or first() etc to kick off a parallel computation, which is then optimized and executed by Spark.

RDD Operations

- RDD Creation:
  - RDDs are created by loading an external dataset, or
  - By distributing an existing collection of objects by calling SparkContext's parallelize method
- Transformations:
  - Each transformation creates a new RDD and does not affect the current RDD
  - Spark keeps track of the set of dependencies between different RDDs in a *lineage graph*.
  - It uses this information to re-compute each RDD on demand and to recover lost data if part of a persistent RDD is lost
  - Transformations are lazily evaluated and executed i.e. they are not executed immediately unless an action is called
  - When a transformation is invoked Spark makes a note of it in metadata and all the transformations are performed once an action is called
  - Spark uses lazy evaluation/execution model to reduce the number of passes it has to take over our data by grouping the operations together in an optimized way
- Actions:
  - Actions are the operations that return a final value to the driver program or write data to an external storage system.
  - Actions actually force the evaluation and execution of the transformations that are required to produce output.
  - Spark's RDDs are by default recomputed each time an action is run on them.

DAG – Directed Acyclic Graph

A Spark application is executed in the following steps:

- Create *RDD graph*, i.e. DAG (directed acyclic graph) of *RDDs* to represent entire computation.
- Create *stage graph*, i.e. a DAG of *stages* that is a logical execution plan based on the RDD graph. Stages are defined by breaking the RDD graph at shuffle boundaries.
- Schedule and execute tasks on workers based on the plan.

RDD Persistence

- Spark re-computes or re-constructs the RDD and all of its dependencies each time we call an action on the RDD which can slow down the performance.
- To avoid computing an RDD multiple times, we can ask Spark to persist the data. This allows future actions to be much faster (often by more than 10x).
- When you persist an RDD, each node stores any partitions of the RDD that it has computed and reuses them in other actions on that dataset (RDD) and the datasets derived from it.
- You can mark an RDD to be persisted using the persist() or cache() methods on it.

```
lines = sc.textFile("war_and_peace.txt",2)
lines.persist()
or
lines.cache()
```

- The first time it is computed in an action, it will be kept in memory on the nodes.
- Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

- Spark allows different storage levels to be specified while persisting the RDDs. For example:

  `lines.persist(MEMORY_ONLY)` **or** `lines.persist(MEMORY_AND_DISK)`

- Default storage level is MEMORY_ONLY which is applied when we use:

  `lines.persist()` **or** `lines.cache`

Storage Level	**Meaning**
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.