Lab : Apache Spark RDD Caching and Persistence

Pre-reqs:

- Google Chrome (Recommended)

RDD Caching and Persistence

RDD Caching and RDD Persistence play very important role in processing data with Spark. With caching and persistence, we will be able to store the RDD in-memory so that we do not have to recompute or evaluate the same RDD again, if required. This is an optimization technique which helps to complete jobs more quickly by saving the evaluation of RDD in memory.

Prerequisites

We need following packages to perform the lab exercise:

- Java Development Kit
- pyspark

JAVA

Verify the installation with: `java -version`

You'll see the following output:

```
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
```

RDD

Fire up the spark-shell from the terminal and create a list as shown below. `spark-shell`

Let us understand this better with an example. The default behavior is that an RDD is computed every time an action is called on the RDD. Look at the following piece of code below.

```
val data = sc.textFile("/headless/Desktop/ernesto-spark/Files/chapter_4/treasure_island.txt")
```

The above line simply loads a text file using the textFile API and stores it to an RDD called data.

```
data.take(5)
```

The above line uses the take() function to return first five elements of the RDD. This is an action which triggers the evaluation. The RDD data is now computed by loading it from the file system and then the action is performed.

Now, let's say we need to count the elements in our RDD. `data.count()`

We are now running a new action, which is causing the RDD to compute again by loading it from the file system and then the action count is performed. As you can see, we are evaluating the RDD twice. This takes lot of time if the data is very big. To overcome this problem we have cache() and persist() method which can be cache or persist the RDD in memory or on the disk. We will discuss more in the next step.

The difference between cache() and persist() methods is that the cache() uses the default storage level of StorageLevel.MEMORY_ONLY, while the persist() method can have the combination of various storage levels as seen below.

Persistence Storage Levels

Level	Description
MEMORY_ONLY`	This is the default storage level. The RDD when cached is stored in memory only. If the RDD doesn't fit in the memory, few partitions which do not fit are computed on the fly when an action is called. The RDDs are stored as deserialized Java objects.
`MEMORY_AND_DISK`	This storage level uses disk to store few partitions of RDD if they do not fit in the memory. So, instead of recomputing the RDD partitions which do not fit in memory, they are spilled to disk. The RDDs are stored as deserialized Java objects.
`MEMORY_ONLY_SER`	This storage level is same as MEMORY_ONLY but RDDs are stored as serialized Java objects. Serialization is more space efficient when compared to deserialized objects but is CPU intensive operation.
`MEMORY_AND_DISK_SER`	This storage level is same as MEMORY_AND_DISK but RDDs are stored as serialized Java objects in memory. The partitions which don't fit in memory are spilled to disk.
`DISK_ONLY`	In this storage level, the RDDs are stored to the disk only and not in memory. This requires low space when compared to persisting in memory but is CPU intensive.

`MEMORY_ONLY_2`
`MEMORY_AND_DISK_2`
`MEMORY_AND_DISK_2`
	All these levels are same as above but store the RDD partitions with replication factor of 2. Meaning each partition is stored on two nodes of a cluster with replication.

Let's go back to our example and see how we can use cache() and persist() methods.

```
val data2 = sc.textFile("/headless/Desktop/ernesto-spark/Files/chapter_4/treasure_island.txt")
```

Once we load the file using the TextFile API, we can now cache or persist the data RDD. Before we can cache or persist we have to import the following.

```
import org.apache.spark.storage.StorageLevel
```

And then use the cache() method, if we need the default implementation of storage only.

```
data2.cache()
```

Persist

However, if we want to use the various storage levels as explained above, we have to use the persist() method and specify the desired storage level. So the code will look like:

```
data2.persist(StorageLevel.MEMORY_AND_DISK_SER)
```

At this point of time, we have simply specified out storage level for persistence. The actual persistence happens when the action on the RDD is called.

```
data2.take(5)
```

After this action is completed, the RDD is stored in the memory and any partitions that do not fit in memory are spilled to disk. When we trigger another action as below, the RDD will not be computed again as it is already computed and persisted. This will reduce the total time taken to complete the job without having to compute the same RDD over and over again.

```
data2.count()
```

Unpersist

It is also possible to remove the persisted RDDs manually. You simply have to use the unpersist() function to the RDD you want to unpersist.

```
data2.unpersist()
```

However, if you choose not to remove the persisted RDDs manually, Spark automatically removes the partitions based on Least Recently Used (LRU) cache policy, when there is too much data cached in memory. So, you need not worry about breaking a job when memory is full.