**Spark Assignment- Srishti**

1. **Why are the number of partitions after a wide transformation equal to 200 by default?**

The default number of partitions after a wide transformation in Apache Spark depends on various factors, including the Spark configuration and the characteristics of the data being processed. However, if you're observing 200 partitions after a wide transformation, it could be due to the default parallelism setting in Spark. The default parallelism is determined by the number of cores available in the Spark cluster, which is typically set to 200 by default when running on a standalone machine or in local mode.

**Why 200?**

* Spark aims to provide parallel processing capabilities to efficiently distribute work across available resources (e.g., CPU cores) while avoiding excessive overhead from managing a very large number of partitions. 200 partitions strike a balance between parallelism and the overhead of managing a larger number of partitions.
* Spark needs to consider the memory and CPU resources available in the environment. Having too few partitions may underutilize available resources, while having too many partitions may lead to excessive overhead and resource contention.
* Through experimentation and testing in various environments and workloads, Spark developers found that 200 partitions often provide good performance across a wide range of scenarios. This default setting helps users get started with Spark without needing to fine-tune partitioning parameters immediately.

1. **persist() and the different storage levels it provides-**

**What**- persist() is used to cache RDDs, Dataframes, and Datasets to improve the performance of subsequent operations.

**Why-**

logLinesRDD

|  |  |  |  |
| --- | --- | --- | --- |
| error,ts,msg1  warn,ts,msg2  error,ts,msg1 | info,ts,msg8  warn,ts,msg2  info,ts,msg8 | error,ts,msg3  info,ts,msg5  info,ts,msg5 | error,ts,msg4  warn,ts,msg9  error,ts,msg1 |

.filter(f(x)

|  |  |  |  |
| --- | --- | --- | --- |
| error,ts,msg1  error,ts,msg1 |  | error,ts,msg3 | error,ts,msg4  error,ts,msg1 |

|  |  |
| --- | --- |
| error,ts,msg1  error,ts,msg1  error,ts,msg3 | error,ts,msg4  error,ts,msg1 |

|  |  |
| --- | --- |
| error,ts,msg1  error,ts,msg1 | error,ts,msg1 |

cleanedRDD

o/p

.collect()

.count()

5

errorMsg1RDD

.filter(f(x))

.coalesce(2)

errorsRDD

In the above example as we can see, we want to perform more than one actions on a single RDD. And If you do count() action on cleanedRDD, you will get 5 as o/p in your driver program and once you get the output, your pipeline will be empty and if you perform another action like .collect() on errorMsg1RDD, the entire process will again start executing from the beginning, and for each subsequent action you want to perform on cleanedRDD, the process will start again from reading the file.

In such cases, when your actions depend on the same RDD, you can cache that RDD in memory or disk to avoid the process from starting again.

**Different Storage Levels-**

1. **MEMORY\_ONLY**

* Default storage level
* It caches the RDD in memory as deserialized Java objects. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they are accessed.
* Suitable when you have enough memory to cache your data and want the best performance.
* **RDD.persist(StorageLevel.MEMORY\_ONLY)**

1. **MEMORY\_ONLY\_SER**

* Similar to MEMORY\_ONLY, but it stores the data in serialized format, which can reduce memory usage but increases CPU overhead for serialization and deserialization.
* Suitable when memory usage is a concern and CPU overhead is acceptable.
* **RDD.persist(StorageLevel.MEMORY\_ONLY\_SER)**

1. **MEMORY\_AND\_DISK**

* Caches the RDD in memory first. If there is not enough memory to cache all partitions, it spills the excess partitions to disk.
* Suitable when you have limited memory and want to leverage disk storage for overflow data.
* **RDD.persist(StorageLevel.MEMORY\_AND\_DISK)**

1. **MEMORY\_AND\_DISK\_SER**

* Similar to MEMORY\_AND\_DISK, but it stores the data in serialized format to reduce memory usage.
* Suitable when both memory and disk space are limited.
* **RDD.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)**

1. **DISK\_ONLY**

* Caches the RDD only on disk.
* Suitable when memory is very limited, and disk space is abundant.
* **RDD.persist(StorageLevel.DISK\_ONLY)**

1. **How is .persist() different to .cache()?**

They both work same in terms of caching data but differ in how they specify the storage level.

cache() is equivalent to calling persist() with the default storage level MEMORY\_ONLY. It caches the RDD in memory as deserialized Java objects. If there is not enough memory to cache all partitions, Spark will spill the excess partitions to disk.

**RDD.cache()**

On the other hand, persist() allows you to specify the storage level explicitly using the StorageLevel enumeration. It gives you more control over how the data is cached, allowing you to choose different storage levels based on your memory and performance requirements.

1. **.unpersist()**

unpersist() method is used to remove the RDD from memory or disk caching.