**Q-1 Default Partitions after wide transformation?**

Spark handles massive amounts of information and at its core lies Resilient Distributed Datasets (RDDs) which are the organized collections of data spread out across multiple computers. Certain operations are performed on these RDD’s that are called as transformations. Since, RDD’s are immutable data structure of spark hence every time data is transformed a new RDD is created forming a lineage.

However, while transforming the data sometimes the data is computed only between the single partition hence no shuffling, that is called narrow transformation. But when data movement is there while transformation then it is called wide transformation. Some functions such as groupByKey, aggregateByKey, aggregate, join, repartition etc. need shuffling of data i.e. it distributes the data across different executors or machines for further processing. And for that the number of shuffle partitions is set to 200. The reason for setting the number of shuffle partitions to a fixed value, such as 200, regardless of the data size or cluster configuration, is primarily for performance optimization and resource management. Though the reason why the default is 200 is from real-world experience that was found to be a very good default. But in practice, that value is usually always bad and it's important to adjust the number of shuffle partitions accordingly.

There might be a few reasons why a fixed number of shuffle partitions was chosen:

* Balance between data skew and overhead: By having a fixed number of shuffle partitions, the system can achieve a balance between handling data skew (where some partitions have much more data than others) and minimizing overhead from excessive partitioning.
* Memory and resource efficiency: Too many partitions can lead to excessive memory consumption and overhead, particularly in cases where the data itself is not very large. By keeping the number fixed, the system can avoid unnecessary resource allocation.
* Consistency and predictability: Fixed partitioning ensures consistency and predictability of performance across different datasets and cluster configurations. This makes it easier to tune and manage the system.
* Simplicity: Having a fixed number of shuffle partitions simplifies the implementation and management of the system, as it removes the need for dynamic partitioning logic based on data size or cluster configuration.
* Fixed shuffle partitioning prevents inefficient disk spills and Out of Memory errors caused by a few excessively large partitions, ensuring more balanced resource utilization and reducing the risk of performance issues

**Q2 – Persist storage levels options in spark**

Cache and persist are both ways to store your data in memory to speed up future computations. Caching or persisting in Spark is like having all the essential ingredients and tools laid out neatly on your kitchen counter before you start cooking. Instead of constantly fetching items from the pantry or drawer, everything you need is right at your fingertips, allowing you to work more efficiently and complete your recipe faster.

So, caching or persisting data in Spark is basically for keeping frequently accessed information readily available in memory, much like having your cooking essentials conveniently within reach in the kitchen.

The only difference is when we cache data in Spark, it's stored in memory (RAM) in its original form and if there's a need for memory, Spark might remove some of the cached data from memory. But when we persist data in Spark, we have more control over how it's stored in memory. We can choose different storage levels such as

* MEMORY\_ONLY: Stores data in JVM memory as deserialized objects; may recomputes unsaved partitions if memory is insufficient.
* MEMORY\_ONLY\_SER: Stores data in JVM memory as serialized objects, saving memory space but requiring additional CPU cycles for deserialization.
* MEMORY\_ONLY\_2: Same as MEMORY\_ONLY but replicates each partition to two cluster nodes.
* MEMORY\_ONLY\_SER\_2: Same as MEMORY\_ONLY\_SER but replicates each partition to two cluster nodes.
* MEMORY\_AND\_DISK: Stores data in JVM memory and spills to disk if memory is insufficient; slower due to I/O involvement.
* MEMORY\_AND\_DISK\_SER: Same as MEMORY\_AND\_DISK but stores serialized objects in memory and on disk when space is not available.
* MEMORY\_AND\_DISK\_2: Same as MEMORY\_AND\_DISK but replicates each partition to two cluster nodes.
* MEMORY\_AND\_DISK\_SER\_2: Same as MEMORY\_AND\_DISK\_SER but replicates each partition to two cluster nodes.
* DISK\_ONLY: Stores data only on disk; high CPU computation time due to I/O involvement.
* DISK\_ONLY\_2: Same as DISK\_ONLY but replicates each partition to two cluster nodes.

Following is the syntax:

import org.apache.spark.storage.StorageLevel

rdd.persist(StorageLevel.MEMORY\_ONLY)

rdd.persist(StorageLevel.MEMORY\_ONLY\_SER)

rdd.persist(StorageLevel.MEMORY\_ONLY\_2)

rdd.persist(StorageLevel.MEMORY\_ONLY\_SER\_2)

rdd.persist(StorageLevel.MEMORY\_AND\_DISK)

rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)

rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_2)

rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_SER\_2)

rdd.persist(StorageLevel.DISK\_ONLY)

rdd.persist(StorageLevel.DISK\_ONLY\_2)

Also,

* Cache doesn't provide options; it simply caches the data with the default storage level.
* With persist you can specify additional options such as:
  + blockSize: Size of data blocks for caching.
  + partitioner: Partitioning strategy.
  + replication: Number of copies to replicate for fault tolerance.
  + serializer: Serializer for storing data.