12 Spark Caching Persistence Storage -KirkYagami 👨‍💻🕵️

1. Introduction to Caching and Persistence in Spark

Caching and persistence are optimization techniques in Apache Spark that allow you to store intermediate results of transformations, making subsequent actions much faster. Instead of recomputing the entire lineage, Spark retrieves the data from the cache or persisted storage.

2. Why Use Caching and Persistence?

- **Performance Improvement:** By caching or persisting data, Spark reduces the need to recompute transformations repeatedly, which can save time and computational resources.
- **Multiple Actions:** When you perform multiple actions on the same dataset, caching or persisting prevents redundant computations.
- **Iterative Algorithms:** Caching is particularly useful for iterative algorithms like machine learning, where the same RDD or DataFrame is used multiple times.

3. Difference Between Caching and Persistence

- **Caching (`cache()`):** This is a shorthand for persisting data using the default storage level, `MEMORY_ONLY`.
- **Persistence (`persist()`):** Allows you to specify a storage level (e.g., MEMORY_ONLY, MEMORY_AND_DISK, etc.), giving you more control over how the data is stored.

4. Storage Levels in Spark

Spark provides several storage levels for caching and persistence:

- **MEMORY_ONLY:** Stores RDD or DataFrame in memory only. If the data doesn't fit in memory, some partitions won't be cached.
- **MEMORY_AND_DISK:** Stores RDD or DataFrame in memory and spills it to disk if it doesn't fit in memory.
- **MEMORY_ONLY_SER:** Similar to MEMORY_ONLY, but stores data in a serialized format, reducing memory usage.
- **MEMORY_AND_DISK_SER:** Similar to MEMORY_AND_DISK, but stores data in a serialized format.
- **DISK_ONLY:** Stores RDD or DataFrame on disk only.
- **OFF_HEAP:** Stores RDD in off-heap memory (outside the JVM heap space).

5. Caching in Spark

When you cache a DataFrame or RDD, Spark stores the data in memory, making future operations faster.

- **Example: Caching a DataFrame**

```
disney_cached = disney_raw.cache()
disney_cached.count()  # Action to trigger caching
```

In this example, the `disney_raw` DataFrame is cached in memory.

```
import pyspark
disney_cached2 = disney_raw.persist(pyspark.StorageLevel.MEMORY_ONLY)
disney_cached2.count()
```

```
 print(disney_cached2.storageLevel)
```

- **Example: Checking Storage Level**

```
print(disney_cached.storageLevel)
```

This will display the storage level of the `disney_cached` DataFrame.

6. Persisting Data in Spark

Persistence provides more flexibility compared to caching, as you can choose different storage levels.

- **Example: Persisting a DataFrame**

```
from pyspark import StorageLevel

disney_persisted = disney_raw.persist(StorageLevel.MEMORY_AND_DISK)
disney_persisted.count()  # Action to trigger persistence
```

Here, the `disney_raw` DataFrame is persisted with the `MEMORY_AND_DISK` storage level.

7. Using Different Storage Levels

- **MEMORY_ONLY Example:**

```
disney_mem_only = disney_raw.persist(StorageLevel.MEMORY_ONLY)
disney_mem_only.show()
```

This persists the `disney_raw` DataFrame in memory. If the DataFrame is too large, some partitions may not be cached.

- **MEMORY_AND_DISK Example:**

```
disney_mem_disk = disney_raw.persist(StorageLevel.MEMORY_AND_DISK)
disney_mem_disk.show()
```

This persists the DataFrame in memory and spills to disk if necessary.

- **DISK_ONLY Example:**

```
disney_disk_only = disney_raw.persist(StorageLevel.DISK_ONLY)
disney_disk_only.show()
```

This example persists the DataFrame only on disk.

8. Unpersisting Data

Unpersisting a DataFrame or RDD manually frees up the memory or disk space used.

- **Example: Unpersisting Data**

```
disney_persisted.unpersist()
```

This frees the memory and disk space used by the `disney_persisted` DataFrame.

9. Impact of Caching and Persistence

- **Performance Impact:** By caching or persisting data, you can significantly reduce the runtime of jobs that involve multiple actions or iterative processing.
- **Memory Usage:** Depending on the storage level, caching and persistence can use a considerable amount of memory. Choose the appropriate storage level based on your resource availability and data size.
- **Disk Usage:** When using storage levels that spill to disk, be mindful of the disk space available.

10. Use Case Scenarios

- **Iterative Algorithms:** Machine learning algorithms that require multiple passes over the same data benefit greatly from caching or persisting.
- **Multiple Actions:** If your Spark job performs multiple actions on the same DataFrame or RDD, caching can avoid redundant computations.
- **Large Data Sets:** For large datasets, `MEMORY_AND_DISK` is often a good choice, as it balances memory and disk usage.

11. Best Practices

- **Cache Selectively:** Not all DataFrames or RDDs need to be cached. Only cache or persist data that will be reused multiple times.
- **Monitor Resources:** Use the Spark UI to monitor memory and disk usage. Adjust storage levels based on resource availability.
- **Unpersist When Done:** Always unpersist data when it's no longer needed to free up resources.

12. Conclusion

Caching and persistence are powerful tools in Spark that can dramatically improve the performance of your jobs by avoiding unnecessary recomputations. By understanding the different storage levels and applying them appropriately, you can optimize your Spark applications to make the best use of available resources.