**Spark Jobs:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Job ID | Description | submitted |  | Duration | Stages: Succeeded/Total | Tasks(for all stages): Succeeded/Total |

**Details for Job (Stages):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Stage Id | Description | submitted | Duration | Tasks: Succeeded/Total | Input |

**Cached Data (Under Storage Tab):**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ID | RDD Name | Storage Level | Cached Partitions | Fraction Cached | Size in Memory | Size on Disk |

* InMemoryTableScan indicates that the data was taken from Cache
* Scan csv indicates that the data was taken from the disk
* Exchange indicates that the data was being shuffled
* When a wide transformation is invoked, 200 partitions are created by default. Therefore, the local distinct results from each of the executors of Stage 1 will be spanned across 200 partitions.
* Stage 1: Initial loading of data( 9 tasks)
* Stage 2: Wide Transformation – distinct() applied, leading to shuffling of data and thereby creation of 200 partitions. ( 200 tasks)
* Stage 3: Intermediate results are shuffled to a single node where the required results are given after the final aggregation. ( 1 task )

Spark.sql(“create database caching\_demo\_db”)

Orders\_df.write.format(“csv”).saveAsTable(“caching\_demo\_db.orders1”)

Spark.sql(“select count(\*) from caching\_demo\_db.orders1”).show() **//Without cache 30 seconds**

Spark.sql(“cache table caching\_demo\_db.orders1”) **//Incase of spark SQL caching is not lazy unlike data frames**

Spark.sql(“cache lazy table caching\_demo\_db.orders1”) **//Cache is eager in case of spark SQL, we can make it lazy by adding lazy keyword**

**Different ways of uncaching -**

Spark.sql(“uncache table caching\_demo\_db.orders1”)  **//clears only the specified table.**

spark.sql(“clear cache”) **//clears all the cached objects of the session.**

spark.catalog.clearCache()

**Points to note( In case of external tables ) :**

- When data is inserted using the insert command, then spark automatically keeps track of the changes made and it will refresh the invalidated cache for subsequent execution.

- However, when the files are manually added / removed from the backend, then spark cannot track these changes to refresh the cache.

- On inserting incremental data, it is updated at the backend but the cache does not automatically get refreshed. It needs to be manually refreshed to display the correct results using the following command

spark.sql(“refresh table ”)

- Cache gets invalidated when the data is changed at the back-end

**Persist**

Persist works exactly the same way as cache but with persist, there is an additional flexibility of changing the default storage levels with an optional parameter.

Persist Storage Level arguments

1. Disk - Whether the data has to be persisted in Disk? True/False

2. Memory - Whether the data has to be persisted in Memory? True/False

3. Off heap - Whether the data has to be persisted Off heap? True/False

4. Deserialized - Whether the data is serialized? True/False

5. Number of Cache Replicas

from pyspark.storagelevel import StorageLevel

orders\_df.persist(StorageLevel(True,True,False,True,1))

Other ways of executing Persist

=> orders\_df.persist(StorageLevel(True,False,False,True,1))

|| Equal to ||

orders\_df.persist(StorageLevel.DISK\_ONLY)

=> orders\_df.persist(StorageLevel(True,True,False,True,1))

|| Equal to ||

orders\_df.persist(StorageLevel.MEMORY\_AND\_DISK)

=> orders\_df.persist(StorageLevel(True,True,False,False,1))

|| Equal to ||

orders\_df.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)