How to access the Spark UI
==========

YARN Resource Manager & Spark UI

ssh -ND 2222 itv005857@g02.itversity.com

http://172.16.1.104:19088/

==========

Cache & Persist
==========

RDD

RDD or a Dataframe

df1 = read a orders file from the disk

df1.cache()

df2 = df1.transformation1

df3 = df2.transformation2

df3.cache()

df4 = df3.transformation3

df4.count()

1. we should cache a dataframe which has to be reused a lot of times...

2. you should not cache a very large dataframe

3. cache is lazy

Dataframes,Spark tables

cache

RDD - memory

Dataframes - memory & disk

you have a 10 gb file in hdfs

80 blocks based on default block size of 128 mb

50 partitions in memory

30 would be cached in disk

disk has 2 parts....

worker node - hdfs / local disk

persist - memory & disk

this storage level can be changed by setting an optional parameter.

pyspark2 instead of pyspark3

==========

cache practicals
==========

1.1 gb

9 blocks in hdfs

count

9 partitions in your dataframe

task1 -> 1000

task2 -> 2000

task3 -> 1500

Memory Deserialized 1x Replicated

serialized - binary format (which will be more optimized in terms of space, will take less space to store)
in takes extra cpu cycles for the format conversion

deserialized - keeping it in object form
takes slightly more space
but in terms of computation this is fast.

on disk the data is always kept in serialized form

and in memory the data is kept in deserialized form

when we talk about caching
disk - serialized
memory - serialized , desiralized

worker node 1

hdfs - b1

it has dynamic resource allocation enabled

1 driver
2 executors

min - 2 executors
max - 10 executors

==========

2 mb file

68883

1.1 gb

9 blocks

9 partitions

9 tasks

e1 - 128 mb data - 25 lakh records - distinct - 68883

e2 - 128 mb data - 25 lakh records - 68883

.

.

e9

whenever we invoke a wide transformation

200 partitions are created...

the results from the 9 tasks will go to 200 partitions..

9 tasks -> 200 tasks -> 1 task

==========

```
orders_df.select("order_id","order_status").filter("order_status == 'CLOSED'").cache()

orders_df.filter("order_status == 'CLOSED'").select("order_id","order_status").count()
```

moving filters ahead so that the data gets limited in the initial stages - predicate pushdown

```
orders_df.select("order_id","order_status").count()

orders_df.select("order_id","order_status").filter("order_status == 'COMPLETE'").count()

cached_df = df.cache()
```

==========

RDD

Dataframes

itv005857_cachingdemo_db

spark.read

spark.write

209 tasks

9 tasks -> 200 tasks

you have 9 files...

200 mb

200 mb

spark.sql("clear cache") --this will uncache all the objects
spark.catalog.clearCache()

spark.sql("uncache table orders") --this will uncache only the specified table

spark.sql("create database itv005857_caching_demo_ext")

spark.sql("create table
itv005857_caching_demo_ext.itv005857_orders_ext(order_id long,
order_date string, customer_id long, order_status string) using csv location
'/user/itv005857/orders'")

spark.sql("insert into itv005857_caching_demo_ext.itv005857_orders_ext
values (111111, '2023-05-29', 222222, 'BOOKED')")

/user/itv005857/orders

orders folder has 1 file initially

and we inserted a record so total files are now 2...

when you insert using insert command then spark will know that the cache is invalidated and in next subsequent use it will refresh it.

but when we add or remove files in backend then spark cannot track it, and we have to refresh the table manually..

RDD - memory

Dataframes and spark table - Memory and disk

in case of RDD and dataframe the caching is a lazy operation

but in case of spark sql it is eager by default

df.filter(....).cache()

==========

why do we have to refresh it 2 times

spark.sql("refresh table itv005857_orders_ext")

========

can cache result in lower performance?

backend data -> dataframe -> cached results

changed -> changed -> changed

changed -> changed -> changed

can cache result in lower performance?

I have a table which is having data in parquet format...

Persist
=========

Cache - Dataframes / Spark tables (Memory & Disk)

Persist - Dataframes / Spark tables (Memory & Disk)

5 arguments

1. Disk

2. Memory -

3. Off heap

4. deserialized

5. number of replicas

orders_df.persist(StorageLevel(True,False,False,False,1))

orders_df.persist(StorageLevel(True,True,False,True,1))

memory - deserialized / serialized

disk - serialized

you have a worker node -

64 GB

16 cpu cores

3 executors - 20 GB / 5 CPU cores

60 GB

4 GB - this is off heap memory

disk only
orders_df.persist(StorageLevel(True,False,False,False,1))

disk only 2
orders_df.persist(StorageLevel(True,False,False,False,2))

memory and disk
orders_df.persist(StorageLevel(True,True,False,True,1))

memory and disk serialized
orders_df.persist(StorageLevel(True,True,False,False,1))