Higher Level API's in Apache Spark

=> Dataframes

=> Spark SQL

RDD's do not consiste of any schema

it is raw data which is distributed across various partitions..

there is no schema or metadata associated...

Table in a Database (orders)

1. Data (1,23-06-1982,101,Closed) - Data Files in physical storage (Disks)

2. Metadata/Schema (column names... order_id integer,order_date date ,customer_id integer,order_status string...) - Metastore

select * from orders

your data + metadata comes together to give you a tabular view

select orderid,order_date from orders

if the column name is not present then it will give analysis exception

Spark SQL -

Data File on the Disk - on your Datalake

Metadata - Metastore

DataFrames -

your rdd with some structure (metadata/schema)

data - in memory

metadata - in memory

there is no metastore, your metadata is kept temporarily in metadata catalog.

Spark table is persistent

Spark Dataframe - we can see it only for the session, once we close the application we will not find it.

spark table is persistent but your dataframe is temporary

spark table is also accessible across other sessions...

but dataframes is only visible for your session..

rdds are not recommended...

Dataframes & Spark SQL

Dataframes are temporary

Spark tables are permanent

why the higher level API's are more performant

RDD + Schema

Dataframe...

1. you would load your file and create a spark dataframe

2. you would perform bunch of transformations..

3. you would write the results back to your storage

its not preferred to use inferSchema to infer the schema...

=> it might not infer it correctly

=> it can lead to performance issues, as spark has to scan the data in order to determine the data types..

======

```
orders_df = spark.read \
.format("csv") \
.option("header","true") \
.option("inferSchema","true") \
```

.load("/public/trendytech/orders_wh/*")

shortcut methods

csv, json, jdbc, orc, parquet, table

csv -

json -

parquet - /public/trendytech/datasets/ordersparquet
parquet is a file format where metadata is embedded in the data..
parquet is a column based file format, and it works very well with spark.

orc -

filtered_df = orders_df.where("customer_id = 11599")

filtered_df.show(truncate = False)

filtered_df = orders_df.filter("customer_id = 11599")

convert a dataframe to a table/view
orders_df.createOrReplaceTempView("orders")

convert a table to a dataframe

ordersdf = spark.read.table("orders")

=> standard dataframe reader
=> shortcut methods (csv,json,parquet,orc,table,jdbc)

.filter
.where

how to convert a dataframe to a spark table/view
df.createOrReplaceTempView("orders")

========

Spark table
============

database - default

spark.sql("create database if not exists itv005857_retail")

spark.sql("show databases").show()

spark.sql("show databases").filter("namespace like 'retail%'").show()

spark.sql("use retail")

spark.sql("show tables").show()

tables

created a database

created a table inside the database

2 tables - one of them is associated with the database and is a persistent table

other one is a temporary view

itv005857_retail - orders

orders which is tempView

its a managed table

hdfs://m01.itversity.com:9000/user/itv005857/warehouse/itv005857_retail.db/o
rders

Data + Metadata

in hdfs home

warehouse/<databasename.db>/<tablename>/files

in this case both the data and metadata are dropped...

Managed table
External table

=======

how we can create a database

how we can create a table

how to load the data from a tempView to a this table.

then we understood that it was a managed table and when we dropped it, both the data and metadata were deleted.

Managed vs External
====================

Managed table
===============
spark.sql("create table itv005857_retail.orders
(order_id integer, order_date string, customer_id integer, order_status string)
using csv")

spark.sql("insert into itv005857_retail.orders select * from orders")

data + metadata

when you drop the table you end up dropping both the data and metadata.

External table
===============
spark.sql("create table itv005857_retail.orders_ext (order_id integer, order_date string, customer_id integer, order_status string) using csv location '/public/trendytech/retail_db/orders'")

you only own the metadata but not the data

when you drop the external table you only end up dropping the metadata

DML operations

insert - is working

update - does not work

delete - does not work

select - this is working

as part of open source spark it wont work...

in databricks update and delete will work (delta lake we get this working)

2 types of table

=> Managed
=> External

===============

orders_df - dataframe (Dataframes API)

orders - table (spark sql)

======

count()

a transformation or action...

groupBy.count (tranformation)

.count (action)

orderBy - transformation

filter - transformation

show - action

head - action

tail - action

take - action

collect - action

distinct - transformation

join - transformation

printSchema - utility function which is neither a transformation nor an action

cache - utility function

createOrReplaceTempView - utility function

Spark Optimization - session 1
==================================

Performance tuning...

1. application code level optimization

cache, use reduceByKey instead of groupByKey

2. Cluster level optimization..

Containers/executors...

Spark Optimization Session - 2

==================================

resources - memory (RAM) , CPU cores (Compute)

our intention is to make sure our job should get the right amount of resources.

10 Node cluster (10 worker nodes)

16 cpu cores

64 GB RAM

Executor (it is like a container of resources)

1 node can hold more than one executor

in a single worker node we can have multiple executors (multiple containers)

container - cpu cores + memory (RAM)

executor/container/JVM

16 cores , 64 gb ram

1

there are 2 strategies when creating containers.

1. Thin executor - intention is to create more executors with each executor holding minimum

possible resources.

total of 16 executors , with each executor holding

each executor - 1 core, 4 gb ram

Drawback

==========

1. In this scenario we will be losing the benefits of multithreading.

2. A lot of copies of broadcast variable are required..

each executor should receive its own copy.

2. Fat executor - intention is to give maximum resources to each executor.

16 cores, 64 gb ram

you can create a executor which can hold 16 cpu cores and 64 gb ram..

Drawbacks

==========

1. It is obverved that if the executor holds more than 5 cpu cores then the hdfs throughput

suffers.

2. if the executor holds very huge amount of memory, then the garbage collection takes a lot of

time.

garbage collection means removing unused objects from memory.

Spark Optimization Session - 3

===============================

10 Nodes

16 cores

2

64 GB Ram

1. Tiny executors

2. Fat executors

16 cores , 64 GB Ram

1 core is given for other background activities

1 gb RAM is given for operating system

in each node we are now left with 15 cores, 63 GB Ram

=> we want multithreading within a executor (> 1 cpu core per executor)

=> we do not want our hdfs throughput to suffer (it suffers when we use more that 5 cores per

executor)

5 is the right choice of number of cpu cores in each executor.

15 cores, 63 GB Ram - each machine

we can have 3 executors running on each worker node

each executor will contain 5 cpu cores and 21 GB Ram.

out of this 21 GB RAM some of it will go as part of overhead (off heap memory)

max (384MB , 7% of executor memory)

= 1.5 GB (overhead / off heap memory) - this is not part of containers.

21 - 1.5 = ~19 GB

that mean in each worker node we have 3 executors.

each executor holds - 5 cpu cores, 19 GB RAM

we have a 10 node cluster/ worker nodes

10 * 3 = 30 (executors across the cluster)

30 executors with each executor holding -

5 cpu cores, 19 GB RAM

3

1 executor out of these 30 will be given for YARN Application Manager

30 - 1 = 29 executors...