#Spark #DataEngineering

Spark Internal Working doubts

that ~~people~~ I use to have, when I was a beginner.

@shubhamDey

Swipe >>>>

#Spark #DataEngineering

What is Spark Driver? And what it does?

Driver Program calls the main function of your application and creates **SparkContext** to connect to the **Spark cluster**, and further it is used to create **RDD**s, **accumulators** and **broadcast variables** on that **cluster**, and also to know what resource manager (**YARN**, **Mesos** or **Standalone**) to communicate to.

Spark Driver contains various other components such as **DAG Scheduler**, **Task Scheduler**, **Backend Scheduler**, and **Block Manager**, which are responsible for translating the user-written code into jobs that are actually executed on the cluster.

A detailed description of its tasks is as follows:

- **Resource Negotiation**: The driver program that runs on the master node of the Spark cluster schedules the job execution and negotiates with the cluster manager.
- **DAG Creation**: It translates the RDD's into the execution graph and splits the graph into multiple stages.
- **Metadata Storage**: The driver stores the metadata about all the RDD's and their partitions, for further data-processing.
- **Tasks creation and Execution**: The driver program uses DAG Scheduler and converts the DAG into stages and again stages into smaller execution units known as tasks. Tasks are then executed by the executors with the help of Task Scheduler.
- **Execution Handling**: In case any task fails due to executor failure; it restarts the execution for that task as well. And after the task has been completed, all the executors submit their results to the Driver.
- **Procedure representation**: Driver exposes the information about the running spark application through a Spark UI.

#Spark #DataEngineering

Why we can create multiple SparkSession object but not SparkContext?

When we create multiple **SparkSession** like I have done below, the **getOrCreate**() function will return a **SparkSession** object if already exists, and creates a new one if not exist.

```
    //Spark session creation
    val spark = SparkSession.builder().master( master = "local").appName( name = "ScalaSparkCode")
      .getOrCreate()
    val spark1 = SparkSession.builder().master( master = "local").appName( name = "ScalaSparkCode1")
      .getOrCreate()

    println(spark.version+"-"+spark.sparkContext.appName + " - " +spark1.version+"-"+spark1.sparkContext.appName)
}
```

Hence, when we print the spark **appName**, it will return the first **appName** that has been created as mentioned below.

```
3.3.0-ScalaSparkCode - 3.3.0-ScalaSparkCode
```

When you create a **SparkSession** object, **SparkContext** is also created and can be retrieved using spark. **SparkContext**. **SparkContext** will be created only once for an application; even if you try to create another **SparkContext** it will give you error.

```
38 Exception in thread "main" org.apache.spark.SparkException: Only one SparkContext should
   be running in this JVM (see SPARK-2243).The currently running SparkContext was created at
   :
39 org.apache.spark.SparkContext.<init>(SparkContext.scala:87)
```

@shubhamDey

Swipe >>>>

#Spark #DataEngineering

Continue...

We will try to find out what happens when we create multiple **SparkSession** using **getOrCreate()**:

```
//creating spark session 1
val spark1 = SparkSession.builder().master( master = "local").appName( name = "ScalaSparkCode1")
  .getOrCreate()

//trying to create spark session 2
val spark2 = SparkSession.builder().master( master = "local").appName( name = "ScalaSparkCode")
  .getOrCreate()

//println the configuration for both sparkSessions without using newSession
spark1.conf.getAll.foreach(println)
println("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
spark2.conf.getAll.foreach(println)
```

We tried to create 2 **sparkSessions** using **getOrCreate()** but it will return same **SparkSession**
As we are using **getOrCreate()**

If we print the **SparkSession** conf details and check the **appName**, you won't be able to find any change, it's showing the details of latest created **SparkSession**.

```
(spark.sql.warehouse.dir,file:/D:/Workspace/MySparkApplication/spark-warehouse)
(spark.driver.port,[redacted])
(spark.app.name,ScalaSparkCode)
(spark.driver.extraJavaOptions,-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke=AL
(spark.master,local)
(spark.app.id,local-1683034713208)
(spark.executor.extraJavaOptions,-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke=
(spark.driver.host,[redacted])
(spark.app.startTime,1683034711190)
(spark.executor.id,driver)
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
(spark.sql.warehouse.dir,file:/D:/Workspace/MySparkApplication/spark-warehouse)
(spark.driver.port,[redacted])
(spark.app.name,ScalaSparkCode)
(spark.driver.extraJavaOptions,-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke=AL
(spark.master,local)
(spark.app.id,local-1683034713208)
(spark.executor.extraJavaOptions,-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke=
(spark.driver.host,[redacted])
(spark.app.startTime,1683034711190)
(spark.executor.id,driver)
```

#Spark #DataEngineering

Continue...

Now, we will analyse what happens when we using **spark.newSession()**

```scala
//creating spark session 1
val spark1 = SparkSession.builder().master( master = "local").appName( name = "ScalaSparkCode1")
  .getOrCreate()

//trying to create spark session 2
val spark2 = SparkSession.builder().master( master = "local").appName( name = "ScalaSparkCode")
  .getOrCreate()

//Created 2nd spark session
val spark3 = spark2.newSession()

//println the configuration for both sparkSessions without using newSession
spark1.conf.getAll.foreach(println)
println("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
spark3.conf.getAll.foreach(println)
```

However, In spark 2.0 allows us to create multiple isolated **SparkSession** using **newSession**() which returns a new **SparkSession** as new session, that has separate **SQLConf**, registered temporary views and UDFs, but shared **SparkContext** and table cache.

If we print the **SparkSession** conf details again and check the **appName**, you will be able to find the changes, it's showing the details of the both **SparkSessions**. But still both of the **SparkSessions** are sharing same **SparkContext**. This is why we can create multiple **SparkSession** but not **SparkContext**.

```
(spark.sql.warehouse.dir,file:/D:/Workspace/MySparkApplication/spark-warehouse)
(spark.driver.port,[redacted])
(spark.app.name,ScalaSparkCode)
(spark.driver.extraJavaOptions,-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke
(spark.master,local)
(spark.app.id,local-1683039698427)
(spark.executor.extraJavaOptions,-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invo
(spark.driver.host,[redacted])
(spark.app.startTime,1683039697257)
(spark.executor.id,driver)
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
(spark.sql.warehouse.dir,file:/D:/Workspace/MySparkApplication/spark-warehouse)
(spark.driver.port,[redacted])
(spark.app.name,ScalaSparkCode1)
(spark.driver.extraJavaOptions,-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke
(spark.master,local)
(spark.app.id,local-1683039698427)
(spark.executor.extraJavaOptions,-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invo
(spark.driver.host,[redacted])
(spark.app.startTime,1683039697257)
(spark.executor.id,driver)
```

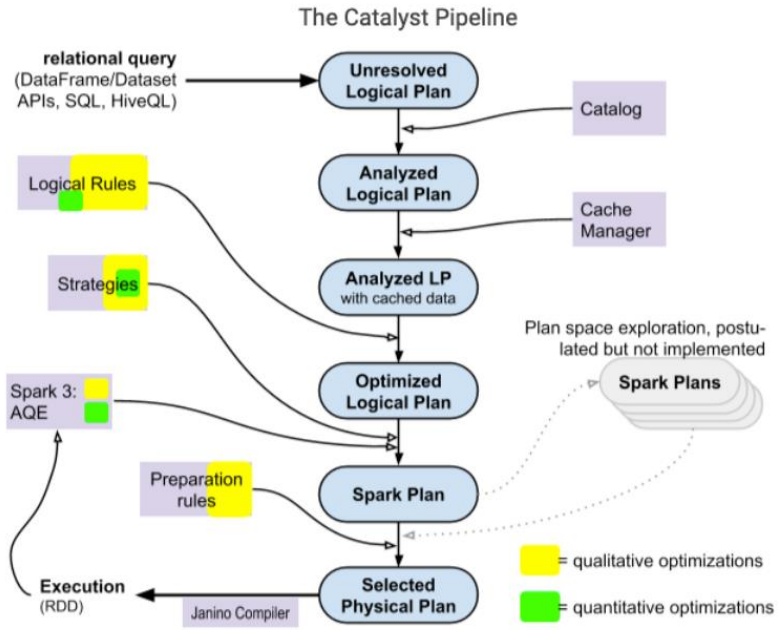
#Spark #DataEngineering

What is Catalyst Optimiser and how does it optimize our spark code execution?

The Catalyst optimizer is a crucial component of Apache Spark. It optimizes structural queries – expressed in SQL, or via the DataFrame/Dataset APIs – which can reduce the runtime of programs and save costs.

Once the Resolved Logical plan is generated it is then passed on to a "**Catalyst Optimizer**" which will apply its own rule and will try to optimize the plan. Basically, Catalyst Optimizer performs logical optimization in short it performs the below tasks:

- It checks for all the tasks which can be performed and computed together in one Stage.
- In a multi-join query, it decides the order/approach of execution of query for better performance.
- Tries to optimize the query by evaluating the filter clause before any project. This, in turn, generates an **Optimized Logical Plan**.



@shubhamDey

Swipe >>>>

#Spark #DataEngineering

What happens internally when your Spark Job is submitted?

Spark performs multiple layers of operations, once you submit your spark code for execution. Here's how it works:

- **Driver** calls the main function of an application and creates **SparkContext** to connect to the **Spark cluster**, and further it is used to create **RDDs**, **accumulators** and **broadcast variables** on that cluster.
- **Driver** identifies **transformations** and **actions** present in the spark application. Based on the flow of program, these tasks are arranged in a graph like structure with directed flow of execution from task to task forming no loops in the graph, which is also known as **logical DAG**.
- The Driver performs certain optimizations like pipelining transformations with the help of **Catalyst Optimizer** and converts **logical DAG** into **Physical Execution Plan** which contains **stages**.
- Some of the subsequent tasks in **DAG** could be combined together in a **single stage** based on the nature of **transformations**, driver sets **stage boundaries**.
    - There are two transformations, namely **narrow transformation**s and **wide transformations**, that can be applied on **RDD** (**Resilient Distributed Databases**)
    - **narrow transformations**: Transformations like **Map** and **Filter** that does not require the data to be shuffled across the partitions.
    - **wide transformations**: Transformations like **ReduceByKey** that does require the data to be shuffled across the partitions. Transformation that requires data shuffling between partitions are called **wide transformation** which results in **stage boundary**.

Continued...

- When the **Physical Execution Plan** is ready, it gets passed on to **DAGScheduler,** and It computes a **DAG** of stages for each job, keeps track of which **RDDs** and stage outputs are materialized, and finds a minimal schedule to run the job. It then submits stages as **TaskSets** to an underlying **TaskScheduler**.
- **TaskScheduler** get **sets of tasks** submitted to them from the **DAGScheduler** for each stage. It further breaks each set of tasks into individual tasks (**Smallest unit of execution**) and sends the tasks to the cluster for executing them. It also retries, if there are failures and mitigates stragglers. And finally, they return events to the **DAGScheduler**.
- **Driver** negotiates for resources from **cluster manager**. The **cluster manager** then launches **executors** on the **worker nodes** on behalf of the **driver**. At this point the driver sends tasks to the cluster manager based on data placement.
- Before executors begin execution, they register themselves with the driver program so that the driver has holistic view of all the executors. Now executors start executing the various tasks assigned by the driver program. At any point of time when the spark application is running, the driver program will monitor the set of executors that run. **Driver** also schedules future tasks based on data placement by tracking the location of cached data.
- **Driver** exposes the information about the running spark application through a Spark UI.