DS501 Big Data Technologies – Spark

By: John

BITTIGER

The Lifelong Learning Platform of Silicon Valley

BITTIGER

版权声明

所有太阁官方网站以及在第三方平台课程中所产生的课程内容，如文本，图形，徽标，按钮图标，图像，音频剪辑，视频剪辑，直播流，数字下载，数据编辑和软件均属于太阁所有并受版权法保护。

对于任何尝试散播或转售BitTiger的所属资料的行为，太阁将采取适当的法律行动。

Copyright Policy

All content included on the Site or third-party platforms as part of the class, such as text, graphics, logos, button icons, images, audio clips, video clips, live streams, digital downloads, data compilations, and software, is the property of BitTiger or its content suppliers and protected by copyright laws.

Any attempt to redistribute or resell BitTiger content will result in the appropriate legal action being taken.

We thank you in advance for respecting our copyrighted content.

For more info:
see https://www.bittiger.io/termsofuse
and https://www.bittiger.io/termsofservice

Outline

Spark EcoSystem

RDD, Data Frame, Data Set

Spark SQL

What is Spark

BITTIGER

- API + Engine
- API
  - RDD
  - DataFrame
  - DataSet
- Engine
  - Not rely on MapReduce/Tez
  - Can run standalone, or on Yarn (Mesos)

Programming Language

- Core: Scala
- API
  - Scala
  - Python
  - Java
- SparkR

Spark EcoSystem

- Spark core
- Spark SQL
- Spark streaming
- MLLib
- GraphX

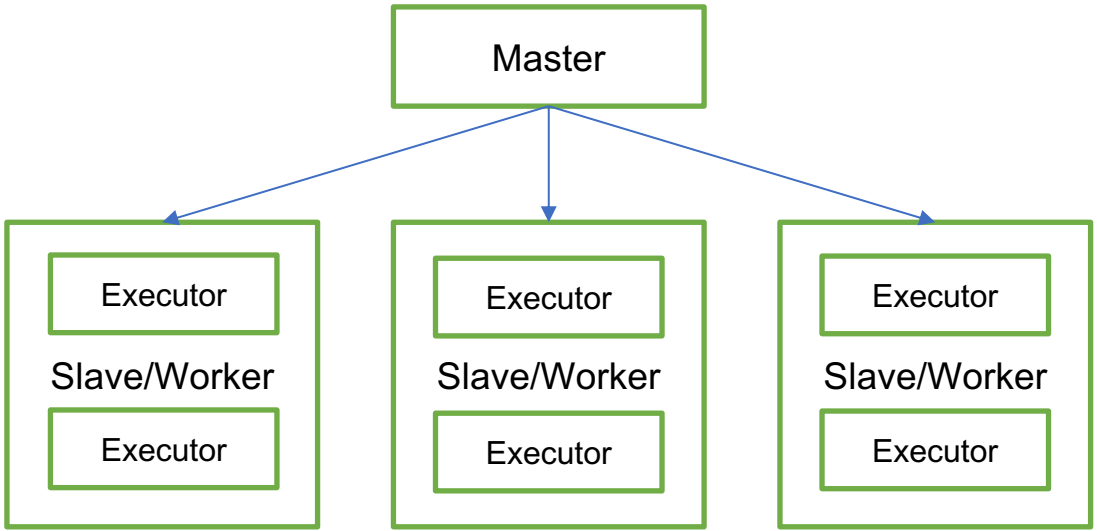
Do I need Hadoop to run Spark?

BITTIGER

- Not necessary
  - Standalone cluster (not require Yarn)
  - Data source other than HDFS (not require HDFS)
- Most case we see Spark and Hadoop run together
  - Multiple tool used together
  - Data is on HDFS
  - Shared workload

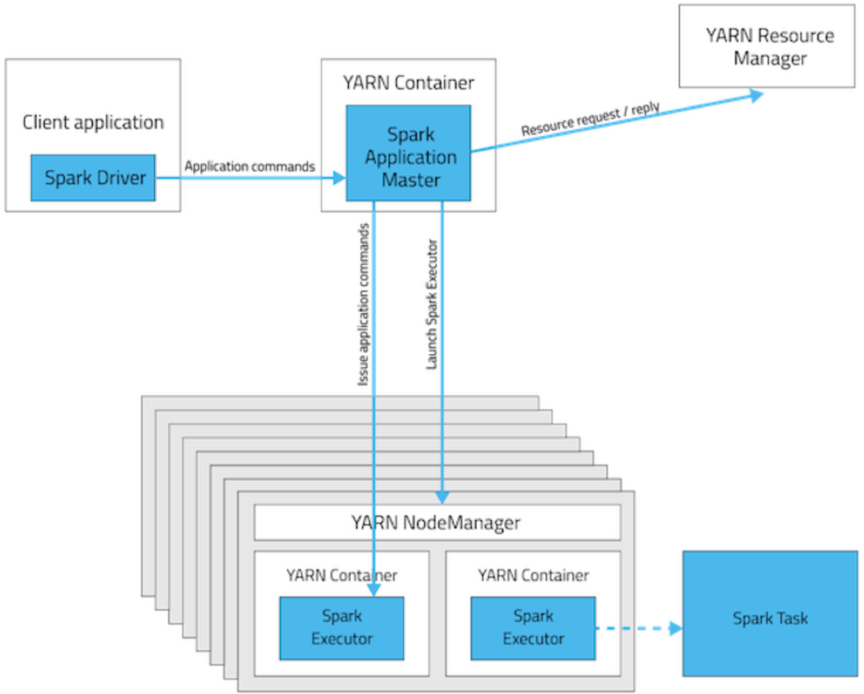
Standalone Spark Cluster

BITTIGER



Spark on Yarn

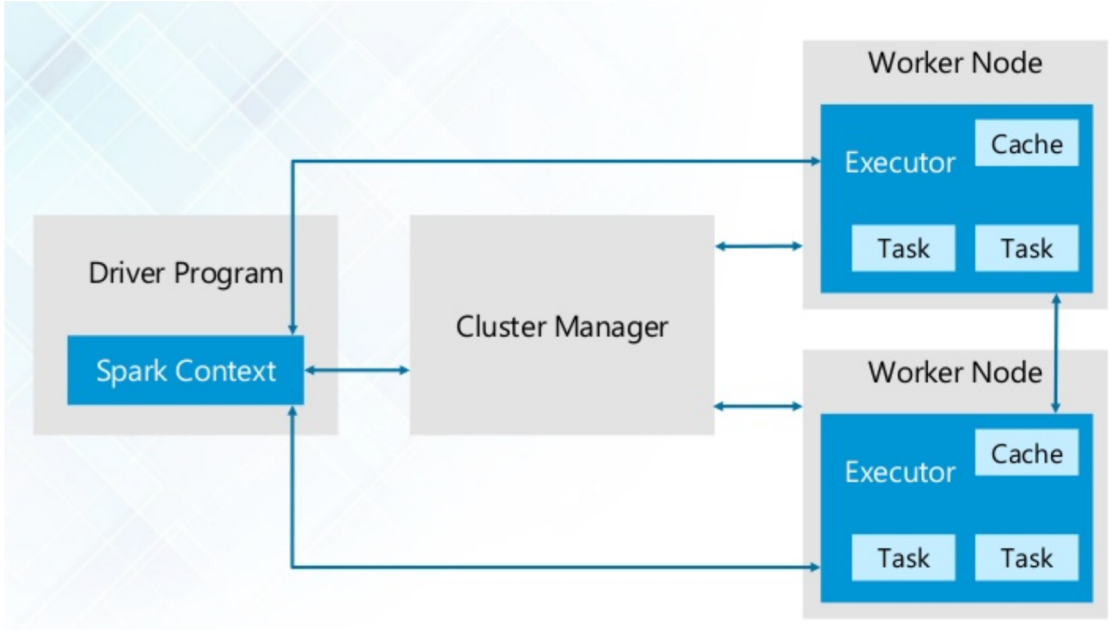
- Cluster mode
  - Driver running in AM
- Client mode (default)
  - Driver running in client



Spark Program

BITTIGER

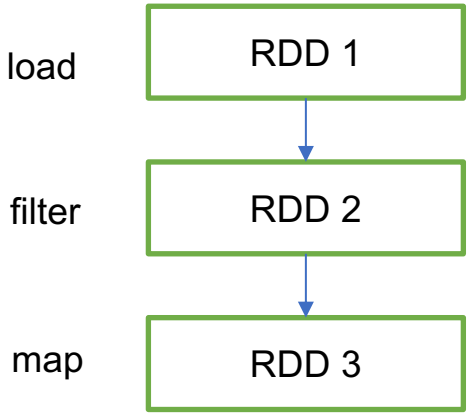
- Multiple threads in executor
  - Your UDF must be thread safe!



What is RDD

BITTIGER

- Resilient distributed dataset
- Programing API
- Distributed dataset
- May cache in memory, or computed from input



RDD Programing

BITTIGER

```
val textFile=sc.textFile("hdfs://...")
val counts=textFile.flatMap(line=>line.split(" "))
                   .map(word=>(word,1))
                   .reduceByKey(_+_)counts
                   .saveAsTextFile("hdfs://...")
```

RDD Features

BITTIGER

RDD Features:

- **Distributed collection:** RDD uses MapReduce operations which is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster.
- **Immutable:** RDDs composed of a collection of records which are partitioned. A partition is a basic unit of parallelism in an RDD, and each partition is one logical division of data which is immutable and created through some transformations on existing partitions.
- **Fault tolerant:** In a case of we lose some partition of RDD , we can replay the transformation on that partition in lineage to achieve the same computation, rather than doing data replication across multiple nodes.
- **Lazy evaluations:** All **TRANSFORMATION**s in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset . The transformations are only computed when an **ACTION** requires a result to be returned to the driver program.
- **Functional transformations:** RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset.
- **Data processing formats:** structured and unstructured data.
- **Programming Languages supported:** RDD API is available in Java, Scala, Python and R.

RDD Limitations:

- **No inbuilt optimization engine:** When working with structured data, RDDs cannot take advantages of Spark's advanced optimizers including catalyst optimizer and Tungsten execution engine.
- **Handling structured data:** Unlike Dataframe and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

DataFrame Programing


```
val df=spark.read.json("people.json")
df.select("name").show()
df.filter($"age">21).show()
df.groupBy("age").count().show()
```

DataFrame Features

Dataframe Features:

- **Distributed collection of Row Object:** A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database, but with richer optimizations under the hood.
- **Data Processing:** Processing structured and unstructured data formats (Avro, CSV, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, MySQL, etc). It can read and write from all these various datasources.
- **Optimization using catalyst optimizer:** It powers both SQL queries and the DataFrame API. Dataframe use catalyst tree transformation framework
- **Hive Compatibility:** Using Spark SQL, you can run unmodified Hive queries on your existing Hive warehouses. It reuses Hive frontend and MetaStore and gives you full compatibility with existing Hive data, queries, and UDFs.
- **Tungsten:** Tungsten provides a physical execution backend which explicitly manages memory and dynamically generates bytecode for expression evaluation.
- **Programming Languages supported:** Dataframe API is available in Java, Scala, Python, and R.

Dataframe Limitations:

- **Compile-time type safety:** As discussed, Dataframe API does not support compile time safety which limits you from manipulating data when the structure is not known.

DataSet Programing*

BITTIGER

```
case class Person(name:String,age:Long)
val peopleDS=spark.read.json(path).as[Person]
peopleDS.filter(p=>p.age>20)
```

Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). The Dataset API is available in Scala and Java.

A DataFrame is a Dataset organized into named columns. In the Scala API, DataFrame is simply a type alias of Dataset[Row].

Dataset Features*

Dataset Features:

- **Provides best of both RDD and Dataframe:** RDD(functional programming, type safe), DataFrame (relational model, Query optimazation , Tungsten execution, sorting and shuffling)
- **Encoders:** With the use of Encoders, it is easy to convert any JVM object into a Dataset, allowing users to work with both structured and unstructured data unlike Dataframe.
- **Programming Languages supported:** Scala and Java
- **Type Safety:** Datasets API provides compile time safety which was not available in Dataframes.
- **Interoperable:** Datasets allows you to easily convert your existing RDDs and Dataframes into datasets without boilerplate code.

Datasets API Limitation:

- **Requires type casting to String:** Querying the data from datasets currently requires us to specify the fields in the class as a string.

Which API to Choose?

- For data processing, DataFrame API is the best
- Supported by APIs: **PySpark**, Scala, SparkR
- DataFrame and DataSet operations are also known as Spark SQL
- DataFrame can also support SQL syntax (though not recommended for engineering development)

Load

BITTIGER

```
df = spark.read.option("delimiter","\t").csv("studenttab10k")

      name:StringType()
      gender:StringType()
      age:IntegerType()
      gpa:DoubleType()

voterDf = spark.read.option("delimiter","\t").csv("votertab10k")

      name:StringType()
      age:IntegerType()
      Registration: StringType()
      contributions:DoubleType()
```

Select


```
from pyspark.sql import functions as F

df.select("name","age",F.col("gpa"),
                       (F.col("gpa")/10).cast("int").alias("gpa_bin"))
```

Add / Rename Column

BITTIGER

Add Column:

```
df.select("name","age",(F.col("gpa")/10).cast("int").alias("gpa_bin"))

df.withColumn("gpa_value", F.col("gpa")) \
  .withColumn("gpa_bin", (F.col("gpa")/10).cast("int")) \
  .withColumn("isFemale",
        F.when(F.col("gender")=="Female",1).otherwise(1)
        )
```

Rename Column:

```
df.select("name","age",F.col("gpa").alias("gpa_value"))

df.withColumnRenamed("gpa","gpa_value")
```

Filter


```
from pyspark.sql import functions as F

df.filter(F.col("age")>18)

df.filter("age > 18")

df.filter(df.age > 18)
```

Group

BITTIGER

```
df.groupBy("age").avg("gpa")

df.groupBy("age").agg(F.mean("gpa").alias("gpa_mean") )

df.groupBy ("age", "gender").avg("gpa")

df.groupBy ("gender").agg(
                          F.mean("gpa").alias("gpa_mean"),
                          F.mean("age").alias("age_mean")
                          )
```

Note:

- **groupBy** can handle multiple fields
- **agg** can aggregate multiple fields at same time, scan data once to save time
- **agg** functions: sum, avg, mean, min, max

Group all


```
df.agg(sum("gpa"))
```

Join


```
df.join(voterDf,"name")

df.join(voterDf,on=["name”,…],how="inner/left/outer left/…")
```

Limit

BITTIGER

```
df.limit(5)
```

Union


```
df1.union(df2) #behave like SQL union, will have duplication
```

Note:
In Spark 2.0, unionAll was renamed to union,
with unionAll kept in for backward compatibility

Distinct

BITTIGER

```
df.distinct() # distinct the whole row
```

Order


```
df.sort("name")
df.orderBy(desc("name"))
```

Action

BITTIGER

```
df.show()
df.show(100)
df.write.save("hdfs://output")
df.count()
```