01 Spark Basics

Apache Spark

Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs in Scala, Java, Python, and R, and an optimized engine that supports general computation graphs for data analysis. It also supports a rich set of higher-level tools including Spark SQL for SQL and DataFrames, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for stream processing.

https://spark.apache.org/

> Every computation expressed in high-level Structured APIs is decomposed into low-level optimized and generated RDD operations and then converted into Scala bytecode for the executors' JVMs. This generated RDD operation code is not accessible to users, nor is it the same as the user-facing RDD APIs.

Understanding Spark Application Concepts

Application

A user program built on Spark using its APIs. It consists of a driver program and executors on the cluster.

SparkSession

An object that provides a point of entry to interact with underlying Spark functionality and allows programming Spark with its APIs. In an interactive Spark shell, the Spark driver instantiates a SparkSession for you, while in a Spark application, you create a SparkSession object yourself.

Job

A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g., save(), collect()).

Stage

Each job gets divided into smaller sets of tasks called stages that depend on each other.

Task

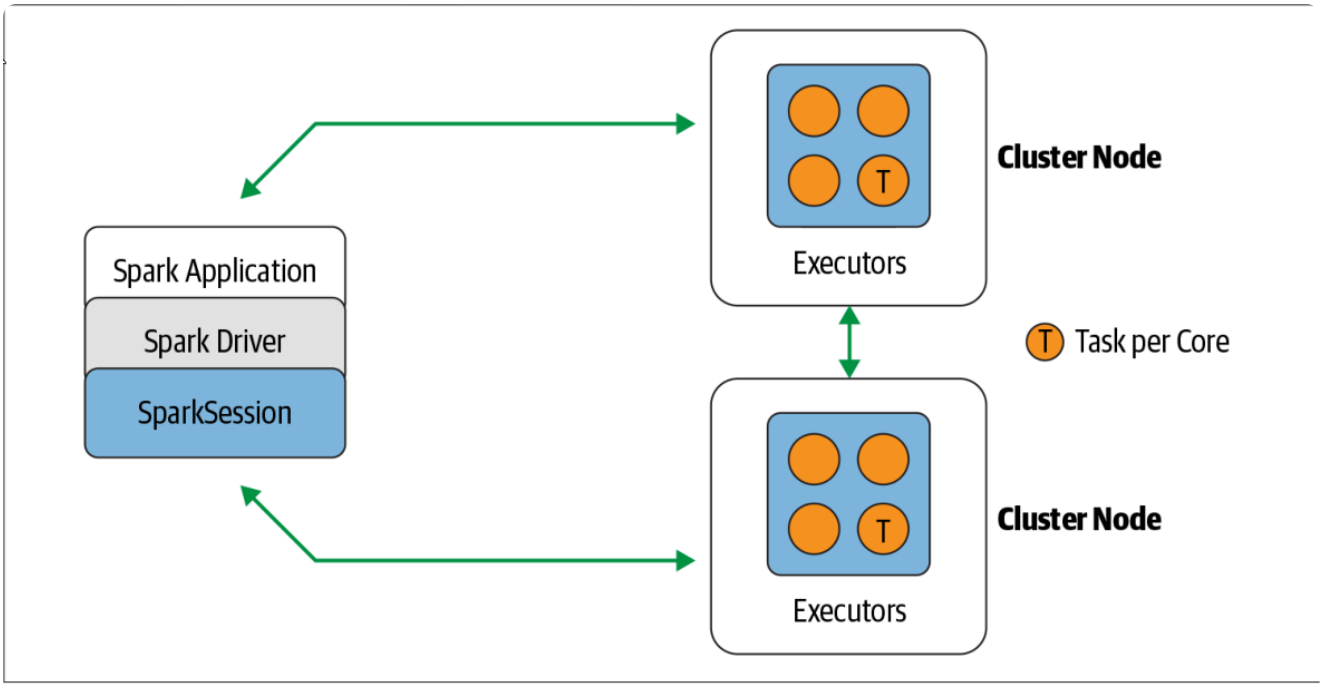
A single unit of work or execution that will be sent to a Spark executor.

Spark Application and SparkSession

At the core of every Spark application is the Spark driver program, which creates a SparkSession object.

- When you're working with a Spark shell, the driver is part of the shell and the SparkSession object (accessible via the variable spark) is created for you.
- When you launch the Spark shell locally on your laptop, all the operations run locally, in a single JVM.
- But you can just as easily launch a Spark shell to analyze data in parallel on a cluster as in local mode.

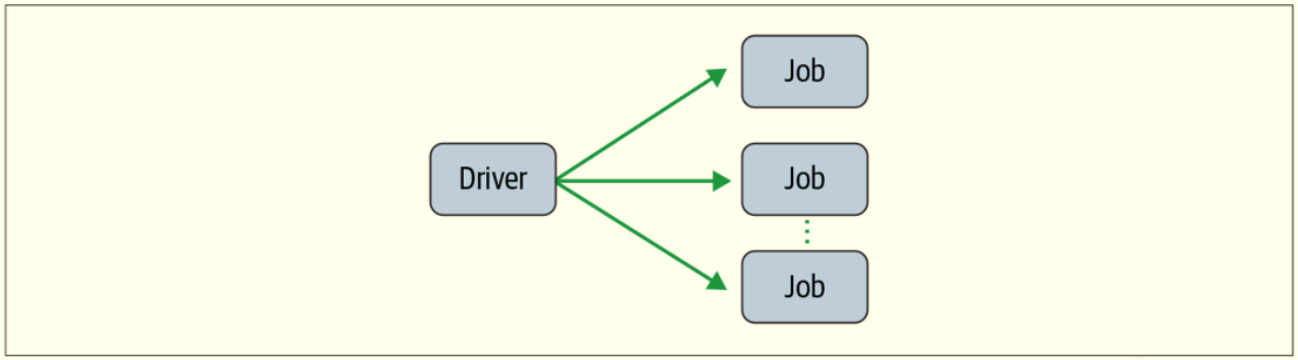
The commands `spark-shell --help` or `pyspark --help` will show you how to connect to the Spark cluster manager.



Spark components communicate through the Spark driver in Spark's distributed architecture

Spark Jobs

During interactive sessions with Spark shells, the driver converts your Spark application into one or more Spark jobs. It then transforms each job into a DAG. This, in essence, is Spark's execution plan, where each node within a DAG could be a single or multiple Spark stages.

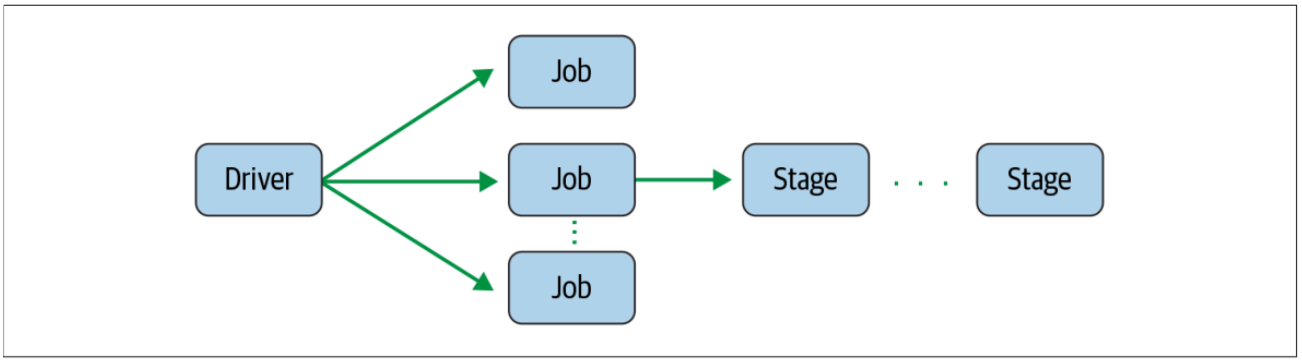


Spark driver creating one or more Spark jobs

Spark Stages

As part of the DAG nodes, stages are created based on what operations can be performed serially or in parallel. Not all Spark operations can happen in a single stage, so they may be divided into multiple stages. Often stages are

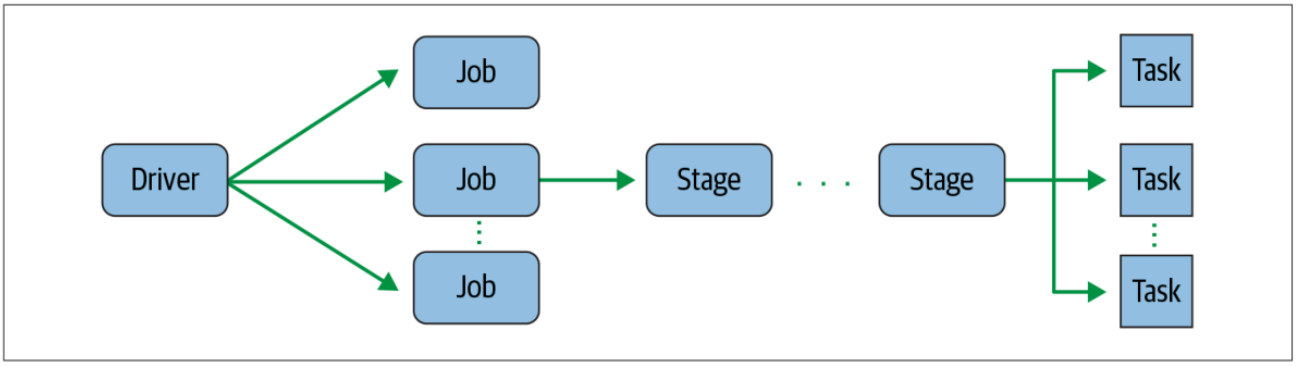
delineated on the operator's computation boundaries, where they dictate data transfer among Spark executors.



Spark job creating one or more stages

Spark Tasks

Each stage is comprised of Spark tasks (a unit of execution), which are then federated across each Spark executor; each task maps to a single core and works on a single partition of data. As such, an executor with 16 cores can have 16 or more tasks working on 16 or more partitions in parallel, making the execution of Spark's tasks exceedingly parallel!



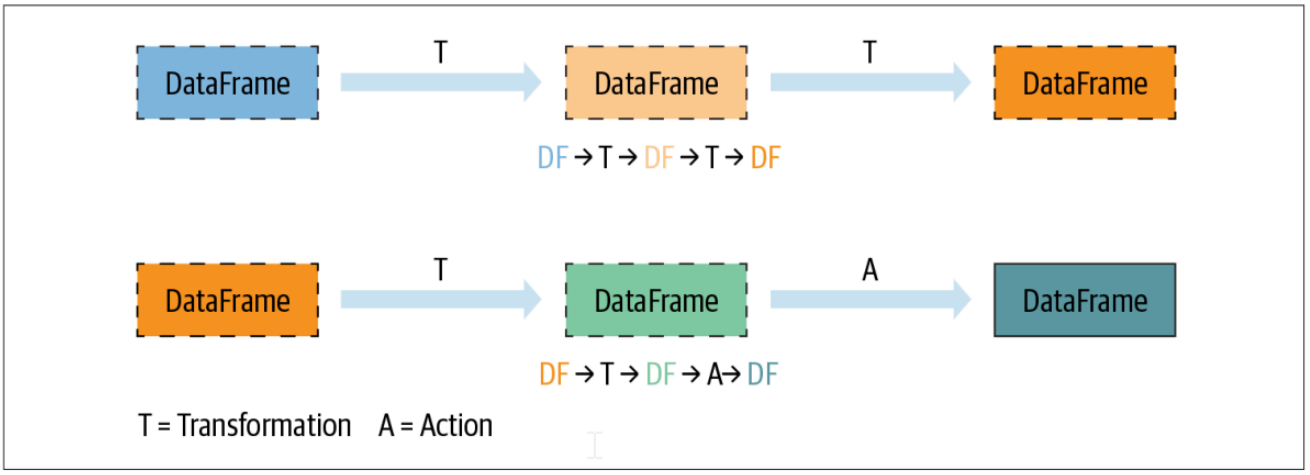
Spark stage creating one or more tasks to be distributed to executors

Transformations, Actions, and Lazy Evaluation

Spark operations on distributed data can be classified into two types: transformations and actions. Transformations, as the name suggests, transform a Spark DataFrame into a new DataFrame without altering the original data, giving it the property of immutability. Put another way, an operation such as `select()` or `filter()` will not change the original DataFrame; instead, it will return the transformed results of the operation as a new DataFrame.

All transformations are evaluated lazily. That is, their results are not computed immediately, but they are recorded or remembered as a lineage. A recorded lineage allows Spark, at a later time in its execution plan, to rearrange certain transformations, coalesce them, or optimize transformations into stages for more efficient execution. Lazy evaluation is Spark's strategy for delaying execution until an action is invoked or data is "touched" (read from or written to disk).

An action triggers the lazy evaluation of all the recorded transformations. In the below figure, all transformations `T` are recorded until the action `A` is invoked. Each transformation T produces a new DataFrame.



Lazy transformations and eager actions

While lazy evaluation allows Spark to optimize your queries by peeking into your chained transformations, lineage and data immutability provide fault tolerance. Because Spark records each transformation in its lineage and the DataFrames are immutable between transformations, it can reproduce its original state by simply replaying the recorded lineage, giving it resiliency in the event of failures.

Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

The actions and transformations contribute to a Spark query plan. Nothing in a query plan is executed until an action is invoked. The following example:

- has two transformations— `read()` and `filter()` and
- one action— `count()`.

The action is what triggers the execution of all transformations recorded as part of the query execution plan.

```
strings = spark.read.text("../README.md")
filtered = strings.filter(strings.value.contains("Spark"))
filtered.count()
```