PYSPARK

zzak00

1 What is Spark?

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes **in-memory caching and optimized query execution** for fast queries against data of any size. Simply put, Spark is a fast and general engine for large-scale data processing.

The fast part means that it's faster than previous approaches to work with Big Data like classical MapReduce. The secret for being faster is that Spark runs on *memory (RAM)*, and that makes the processing much faster than on disk drives.

The general part means that it can be used for multiple things like running distributed SQL, creating data pipelines, ingesting data into a database, running Machine Learning algorithms, working with graphs or data streams, and much more.

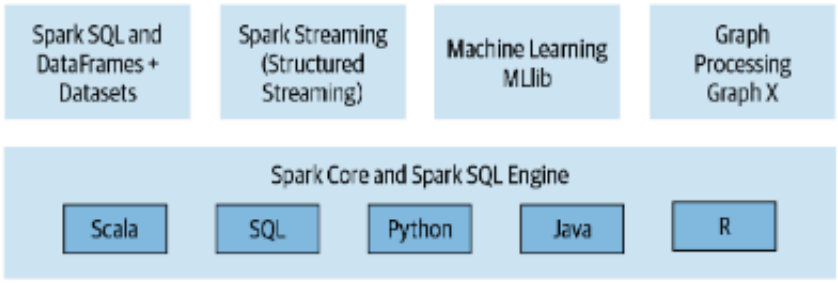

Figure 1: Spark EcoSystem.

- **Apache Spark Core** – Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built upon. It provides in-memory computing and referencing datasets in external storage systems.
- **Spark SQL** – This module works well with structured data. You can read data stored in an RDBMS table or from file formats with structured data (CSV, text, JSON, Avro, ORC, Parquet,etc.) and then construct permanent or temporary tables in Spark. Also, when using Spark's Structured APIs in Java, Python, Scala, or R, you can combine SQL-like queries to query the data just read into a Spark DataFrame.

- **Spark Streaming** – This component allows Spark to process real-time streaming data. Data can be ingested from many sources like Kafka, Flume, and HDFS (Hadoop Distributed File System). Then the data can be processed using complex algorithms and pushed out to file systems, databases, and live dashboards.
- **MLlib (Machine Learning Library)** – Apache Spark is equipped with a rich library known as MLlib. This library contains a wide array of machine learning algorithms- classification, regression, clustering, and collaborative filtering. It also includes other tools for constructing, evaluating, and tuning ML Pipelines. All these functionalities help Spark scale out across a cluster.
- **GraphX** – Spark also comes with a library to manipulate graph databases and perform computations called GraphX. GraphX unifies ETL (Extract, Transform, and Load) process, exploratory analysis, and iterative graph computation within a single system.

2 Apache Spark's Distributed Execution

At a high level in the Spark architecture, a Spark application consists of a driver program that is responsible for orchestrating parallel operations on the Spark cluster. The driver accesses the distributed components in the cluster—the Spark executors and cluster manager—through a SparkSession.

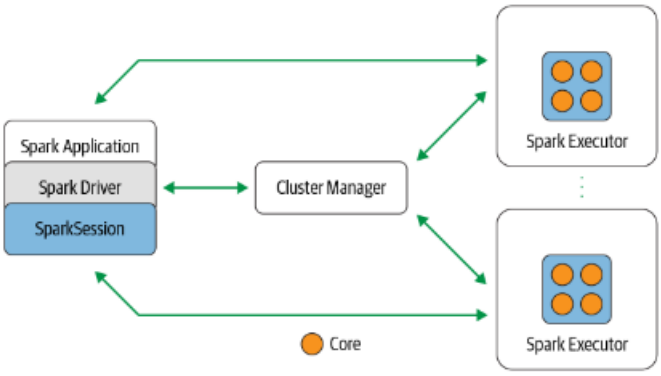

Figure 2: Apache Spark components and architecture.

Java Virtual Machine

JVM is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementation. Having a specification ensures interoperability of Java programs across different

implementations so that program authors using the Java Development Kit (JDK) need not worry about idiosyncrasies of the underlying hardware platform.

Spark Driver

As the part of the Spark application responsible for instantiating a SparkSession, the Spark driver has multiple roles: it communicates with the cluster manager; it requests resources (CPU, memory, etc.) from the cluster manager for Spark's executors (JVMs); and it transforms all the Spark operations into DAG computations, schedule them, and distributes their execution as tasks across the Spark executors. Once the resources are allocated, it communicates directly with the executors.

SparkSession

In Spark 2.0, the SparkSession became a unified conduit to all Spark operations and data. Not only did it subsume previous entry points to Spark like the SparkContext,SQLContext, HiveContext, SparkConf, and StreamingContext, but it also made working with Spark simpler and easier. Through this one conduit, you can create JVM runtime parameters, define DataFrames and Datasets, read from data sources, access catalog metadata, and issue Spark SQL queries. SparkSession provides a single unified entry point to all of Spark's functionality.

Cluster Manager

The cluster manager is responsible for managing and allocating resources for the cluster of nodes on which your Spark application runs. Currently, Spark supports four cluster managers: the built-in standalone cluster manager, Apache Hadoop YARN, Apache Mesos, and Kubernetes.

Spark Executor

A Spark executor runs on each worker node in the cluster. The executors communicate with the driver program and are responsible for executing tasks on the workers.In most deployments modes, only a single executor runs per node.

Deployment modes

An attractive feature of Spark is its support for myriad deployment modes, enabling Spark to run in different configurations and environments. Because the cluster manager is agnostic to where it runs (as long as it can manage Spark's executors and fulfill resource requests), Spark can be deployed in some of the most popular environments such as Apache Hadoop YARN and Kubernetes—and can operate in different modes. Table 1-1 summarizes the available deployment modes.

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Figure 3: Spark Deployment modes.

Distributed data and partitions

Actual physical data is distributed across storage as partitions residing in either HDFS or cloud storage (see Figure 1-5). While the data is distributed as partitions across the physical cluster, Spark treats each partition as a high-level logical data abstraction as a DataFrame in memory. Though this is not always possible, each Spark executor is preferably allocated a task that requires it to read the partition closest to it in the network, observing data locality.

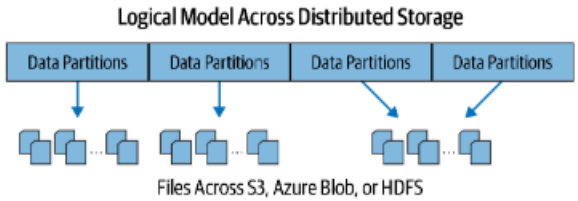

Figure 4: Data is distributed across physical machines.

Partitioning allows for efficient parallelism. A distributed scheme of breaking up data into chunks or partitions allows Spark executors to process only data that is close to them, minimizing network bandwidth. That is, each executor's core is assigned its own data partition to work on.

3 Understanding Spark Application Concepts

Spark Application and SparkSession

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

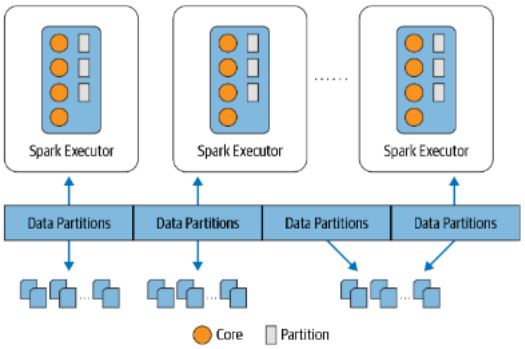

Figure 5: Each executor's core gets a partition of data to work on.

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.
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, as you saw in the earlier examples when you launched the shells.

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.

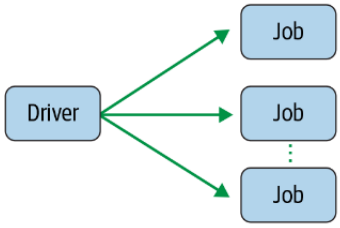

Figure 6: Spark driver creating one or more Spark jobs.

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.

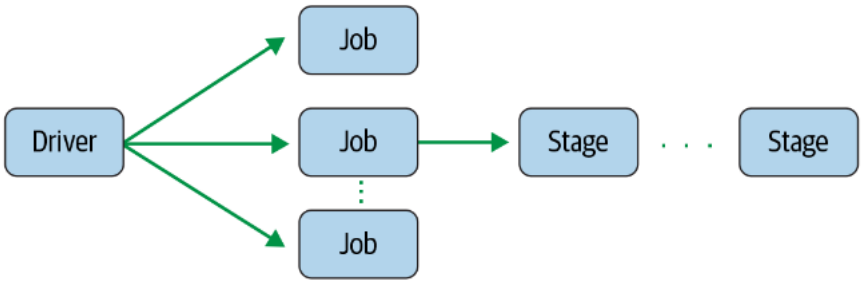

Figure 7: Spark job creating one or more stages.

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!

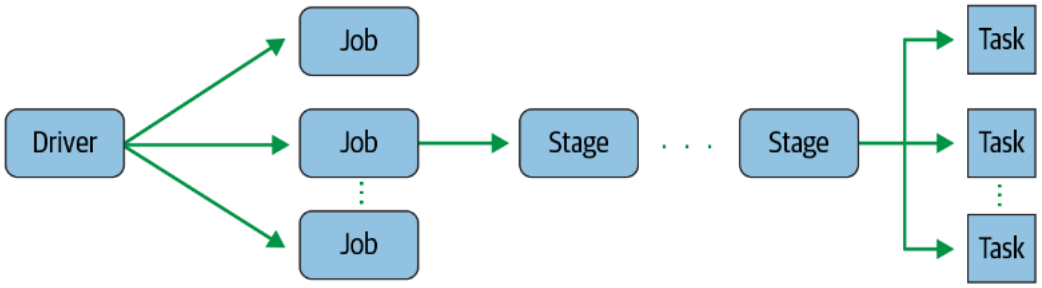

Figure 8: Spark stage creating one or more tasks to be distributed to executors.

Transformations, Actions and 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 Figure 8, all transformations T are recorded until the action A is invoked. Each transformation T produces a new DataFrame. While lazy

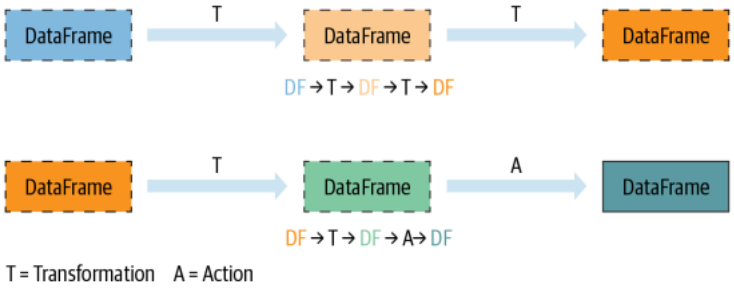

Figure 9: Lazy transformations and eager actions.

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()

Figure 10: Transformations and actions as Spark operations.

Narrow and Wide Transformations

Transformations can be classified as having either narrow dependencies or wide dependencies. Any transformation where a single output partition can be computed from a

single input partition is a narrow transformation. For example, *filter()* and *contains()* represent narrow transformations because they can operate on a single partition and produce the resulting output partition without any exchange of data. However, *groupBy()* or *orderBy()* instruct Spark to perform wide transformations, where data from other partitions is read in, combined, and written to disk.
Since each partition will have its own count of the word that contains the "Spark" word in its row of data, a count *groupBy()* will force a shuffle of data from each of the executor's partitions across the cluster. In this transformation, *orderBy()* requires output from other partitions to compute the final aggregation.

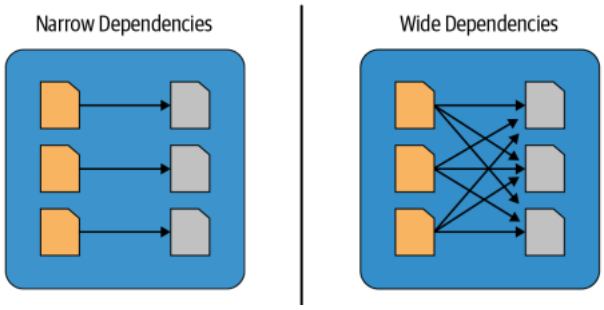

Figure 11: Narrow versus wide transformations

3.1 What happens when a client submits a Spark Job?

When the job enters the driver converts the code into a logical directed acyclic graph (DAG). Afterwards, the driver performs certain optimizations like pipelining transformations.

Furthermore, it converts the DAG into physical execution plan with the set of stages. Meanwhile, it creates small execution units under each stage referred to as tasks. Then it collects all tasks and sends it to the cluster.

It is the driver program that talks to the cluster manager and negotiates for resources. After this cluster manager launches executors on behalf of the driver. At this point based on data, placement driver sends tasks to the cluster manager.

Executors register themselves with the driver program before executors begin execution. So that the driver has the holistic view of all the executors.

Now, Executors executes all the tasks assigned by the driver. Meanwhile, the application is running, the driver program monitors the executors that run. In the spark architecture driver program schedules future tasks.

All the tasks by tracking the location of cached data based on data placement. When it calls the stop method of sparkcontext, it terminates all executors. After that, it releases the resources from the cluster manager.

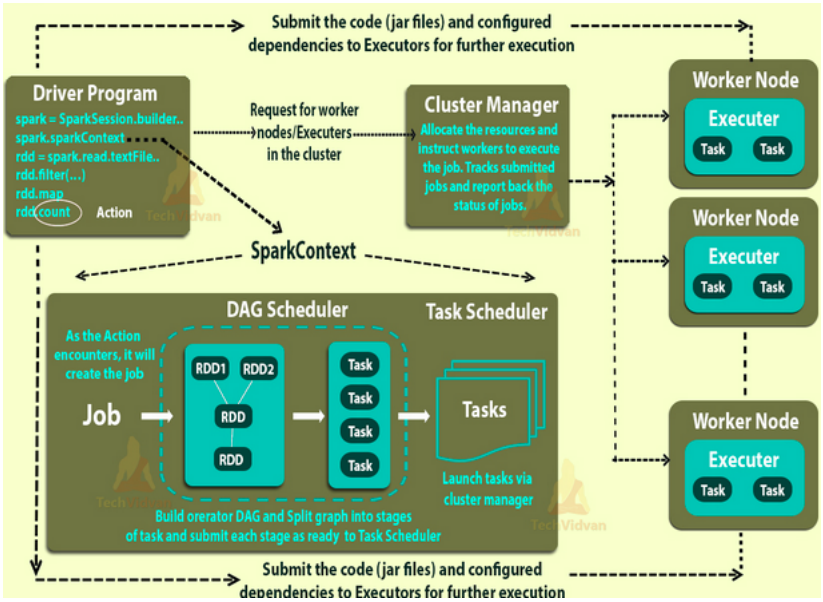

Figure 12: What happens when a client submits a Spark Job?

4 Apache Spark's Structured APIs

Spark: What's Underneath an RDD?

The RDD is the most basic abstraction in Spark. At the core, an RDD is an immutable distributed collection of elements of data, partitioned across nodes of cluster tat can be operated in parallel with low-level API that offers *transformations* and *actions*. There are three vital characteristics associated with an RDD:

- Dependencies
- Partitions (with some locality information)
- Compute function: Partition => Iterator[T]

Sinple and Elegant!. Yet there are a couple of problems with this original model. For one, the compute function (or computation) is opaque to Spark. That is, Spark does not know what you are doing in the compute function. Whether you are performing a join, filter, select, or aggregation, Spark only sees it as a lambda expression. Another problem is that the Iterator[T] data type is also opaque for Python RDDs; Spark only knows that it's a generic object in Python. Furthermore, because it's unable to inspect the computation or expression in the function, Spark has no way to optimize the expression—it has no comprehension of its intention. And finally, Spark has no knowledge of the specific data type in T . To Spark it's an opaque object; it has no idea if you are accessing a column

of a certain type within an object. Therefore, all Spark can do is serialize the opaque object as a series of bytes, without using any data compression techniques.

4.1 The solution: Structuring Spark

Spark 2.x introduced a few key schemes for structuring Spark. One is to express computations by using common patterns found in data analysis. These patterns are expressed as high-level operations such as filtering, selecting, counting, aggregating, averaging, and grouping. This provides added clarity and simplicity. And the final scheme of order and structure is to allow you to arrange your data in a tabular format, like a SQL table or spreadsheet, with supported structured data types.

4.2 The DATAFRAME API

Inspired by pandas DataFrames in structure, format, and a few specific operations, Spark DataFrames are like distributed in-memory tables with named columns and schemas, where each column has a specific data type: integer, string, array, map, real, date, timestamp, etc.

4.3 Spark's Data Types

Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

Figure 13: Basic Python data types in Spark

Data type	Value assigned in Python	API to instantiate
`BinaryType`	`bytearray`	`BinaryType()`
`TimestampType`	`datetime.datetime`	`TimestampType()`
`DateType`	`datetime.date`	`DateType()`
`ArrayType`	List, tuple, or array	`ArrayType(dataType, [nullable])`
`MapType`	`dict`	`MapType(keyType, valueType, [nullable])`
`StructType`	List or tuple	`StructType([fields])`
`StructField`	A value type corresponding to the type of this field	`StructField(name, dataType, [nullable])`

Figure 14: Python structured data types in Spark

References