Lecture Notes

Apache Spark

Getting Started with Apache Spark

Spark Overview

"**Apache Spark™** is a unified analytics engine for large-scale data processing." It is an **open-source, distributed-computing engine**, and it provides a productive environment for data analysis owing to its lightning speed and support for various libraries.

According to Spark's documentation, the four features listed below make Spark a powerful unified engine for data processing at a massive scale:

- **Speed:** This is one of the defining characteristics of Apache Spark. It is known for its speed, which is considered to be 100x faster than that of the MapReduce framework. Spark's speed is a result of its in-memory computation and the use of a DAG scheduler.

- **Ease of use:** Spark offers language support in Java, Scala, SQL, Python and R to run queries interactively using more than 80 different operators.

- **Generality:** Spark's support for various libraries includes Spark SQL, Spark Streaming, MLlib and GraphX. This ecosystem strengthens Spark's data analysis.

- **Runs everywhere:** Spark can use its standalone cluster manager to run on Apache Mesos, Hadoop YARN or Kubernetes. It can access data stored in various storages, including HDFS, S3 and others.

Some typical use cases of Spark include the following:

- To perform exploratory analyses on data sets of size in the order of hundreds of GBs (or even TBs) within realistic time frames
- To run near-real-time reports from streaming data
- To build machine learning models

Spark vs MapReduce

Storage systems are primarily of the following two types:

- Memory
- Disk

Every time you want to analyse or use stored data, it is first loaded into memory and then operations are performed on it. MapReduce utilises **disk-based processing concepts**. In such systems, data is stored on **hard disk drives** (large storage systems such as HDFS and S3), and the final output is arrived at in two phases: **Map and Reduce**.

In the Map phase, first, data is processed and then it is split into **partitions** (mapped). The output of the Map phase is again transferred to the disk as intermediate output. The same output acts as input for the Reduce phase. So, the data is again read from the disk to memory in the Reduce phase. The output of the Reduce phase is then stored in the disk.

Here, you can see that the two phases, Map and Reduce, are not performed together in memory. There is an intermediate output that is stored in the disk. This back-and-forth movement of data between the disk and memory creates an overhead, which slows down the entire cycle of data processing.

However, in **in-memory processing systems**, once data is loaded into memory, it remains there until all operations are performed. Since there is no intermediate output involved, there is no time required to write/read files to the intermediate disk.

There are three main differences between Spark and MapReduce. These are as follows:

A. MapReduce involves disk-based processing and processing is slow in this framework, whereas Spark involves in-memory processing and processing is fast.
B. MapReduce involves only batch processing, whereas Spark involves both batch processing and real-time data processing.
C. In MapReduce, interactive queries access data from the disk repeatedly, whereas in Spark, interactive queries are performed on distributed in-memory data.

Spark Ecosystem

Apache Spark comes with a bundle of features along with high-speed processing. Different packages and features are built over Spark, serving different purposes. Spark supports different APIs, which allow it to work as a unified platform for various aspects of data processing and analysis. You have the comfort of loading data from a wide range of sources on which these APIs can work and produce the desired output.

The Spark ecosystem adds to Spark's 'Runs Everywhere' feature by supporting several libraries and functionalities. All of these make Spark a highly powerful and easy-to-use framework.

Spark Architecture

The architecture of Spark consists of:

- **A Driver node** (master), which runs the Driver program, and
- **Worker nodes** (slave), which run the Executor program.

The driver program is responsible for managing the Spark applications that you submit, while the executor program uses the worker nodes as the distributed storage and the processing space to run those applications. The driver program and the executor program are managed by the cluster manager. A cluster manager is a pluggable component of Spark. Because of this, Spark can run in various cluster manager modes, which include the following:

- In the **Standalone** mode, Spark uses its own cluster manager and does not require any external infrastructure.
- At an enterprise level, for running large Spark jobs, the framework can be integrated with external cluster managers such as **Apache YARN** or **Apache Mesos**.
- This facility to run Spark with external cluster managers allows Spark applications to be deployed on the same infrastructure as Hadoop, and it serves as an advantage for companies that are looking to use Spark as an analytics platform.

Spark Context **does not execute** the code, but it creates an optimised physical plan of the execution within the Spark architecture. It is the initial entry point of Spark into the distributed environment.

Spark APIs

Unstructured APIs

- Unstructured data is generally free-form text that lacks a schema (which defines the organisation of the data).
- Examples of such data include text files, log files, images, videos, etc.
- To deal with unstructured data, Spark uses an unstructured API in the form of a Resilient Distributed Dataset (RDD).
- RDD is the core component of Spark, and it helps in working with unstructured data.

Structured APIs

- Structured data includes a schema.
- The data could be structured in a columnar or a row format.
- Structured data formats include ORC files, Parquet files, tables or dataframes in SQL, Python, etc.

- To deal with this type of data, Spark provides multiple APIs, including SparkSQL, DataFrame and Dataset.

Programming with Spark RDDs

Introduction to Spark RDDs

The main properties of RDDs are as follows:

- **Distributed collection of data:** RDDs exist in a distributed form over multiple worker nodes. This property helps them store large data sets. The driver node is responsible for creating and tracking this distribution.
- **Fault tolerance:** This refers to the ability to generate RDDs if they are lost during computation. Intuitively, fault tolerance implies that if somehow an RDD gets corrupted (lost due to the volatility of memory), then you can recover the uncorrupted RDD. You will learn more about this in the next session.
- **Parallel operations:** Although RDDs exist as distributed files across worker nodes, their processing takes place in parallel. Multiple worker nodes work simultaneously to execute a complete job.
- **Ability to use varied data sources:** RDDs are not dependent on any specific structure of an input data source. They are adaptive and can be built from different sources.

Creating RDDs

There are two methods to create RDDs. These are summarised below:

- **parallelize( ):** This method is used when data is present in the Spark driver program.
- **textFile( ):** This method is used when data has to be loaded from external file storage systems.

Due to their large sizes, data sets are mostly present in external storage systems instead of in-memory. Hence, the textFile( ) method is used more compared with the parallelize( ) method.

Operations on RDDs

Some examples of transformations are as follows:

- *map*: It runs a function over each element of an RDD.
- *flatMap*: It runs a function where the output of each element may not be a single element.
- *groupByKey*: It is used to group data as key–value pairs.
- *union*: It takes the set union of two RDDs.

Some examples of actions are as follows:

- *count*: It counts the number of elements in an RDD.
- *collect*: It persists an RDD to a local machine.
- *reduce*: It reduces the elements through an aggregation function. The most common type of the reduce function is a sum.

Transformation Operations

Here is a list of some transformation operations that can be performed in Spark:

filter(): This operation is useful for filtering out the contents of an RDD based on a condition.
map(): When applied to an RDD, this method will return a new RDD based on the operation performed on that RDD.
flatMap(): This is another operation, which is similar to the map() operation, although the number of elements in the output can be different from the number of elements in the input in this operation.
distinct(): This function is used to identify unique elements in an RDD and put them in a new RDD.
sorted(): This method is not used to perform an operation on an RDD but to sort the elements in a list.
union(): This operation will work on two RDDs and will result in an output that contains all the elements present in both the RDDs: “rdd1.union(rdd2)”.

intersection(): This operation will work on two RDDs and will result in an output that contains only those elements that are present in both the RDDs: “rdd1.intersection(rdd2)”.

subtract(): This operation will work on two RDDs and will result in an output that contains all the elements present in rdd1 but not those present in rdd2: “rdd1.subtract(rdd2)”.

cartesian(): This operation will work on two RDDs and will result in an output that contains pairs of each element of rdd1 with each element of rdd2: “rdd1.cartesian(rdd2)”.

Action Operations

Here is a list of some actions operations that can be performed in Spark:

collect(): This operation collects all the elements of an RDD from every partition and returns the result as an array to the driver node.

count(): This operation returns the total number of elements of an RDD.

take(num): This operation takes the first 'num' elements of an RDD.
It first scans one partition and then uses the results from that partition to estimate the number of additional partitions that are needed to satisfy the limit.

top(num): This operator will return the num highest values in an RDD in descending order.

countByValue(): This is a powerful operator. When applied to an RDD, it returns a dictionary of (key, value) pairs, where each key is one element of the RDD and each value is the number of times an element has appeared in that RDD.

Lazy Evaluation

Transformations in Spark follow **lazy evaluation**, i.e., they are evaluated only when required. When you perform a transformation on an RDD, it is stored in the background as **metadata**. Spark creates an object, which stores the necessary steps to create a new RDD from the existing one. Then, it waits for an **action** to execute all the transformations that are stored in the metadata object.

Lazy evaluation has the following advantages:

- Due to lazy evaluation, Spark does not perform unnecessary computation.
- More specifically, it lets Spark perform computation only when a final result is required. It eliminates redundant steps, if any, or merges logically independent tasks into a single parallel step.
- It also helps Spark to use the executor memory efficiently, as it will not create and store multiple RDDs, which occupy the limited memory space, until an RDD is required.

When a user submits a code file, SparkContext creates an optimal plan in the background. This plan (the lineage of the RDD) is stored in the **Directed Acyclic Graph (DAG) Scheduler** within SparkContext:

- **Directed:** The arrows link one point to another point in a single direction.
- **Acyclic:** All the nodes have a property, which states that if you start from any node in the graph and follow the arrows, then you cannot come back to the same node. That is, the graph does not have any cyclic loop.

The DAG scheduler prepares the flow of how an RDD is derived from the parent RDD. This process helps to make the RDDs fault-tolerant:

- At any instance, if a Spark job fails, then you can use the stage flow mentioned in the graph to return to the same state.
- This way, your progress is not lost, and, hence, the term 'resilient' is associated with RDDs. Note that the **lineage stores the details of the transformations, and not the data itself**.
- Once the transformations leading up to an RDD are known, any stage (representing a subset of the original data) can be recreated.

Paired RDDs

Paired RDDs

A paired RDD is a special RDD class, which holds data as (Key, Value) pairs. There are two important methods in paired RDDs: reduceByKey() and groupByKey(). Let's understand both of them one by one.

reduceByKey(): The reduceByKey() method is used to perform a particular operation on the elements of RDDs.

groupByKey(): The groupByKey() method is used to perform the same operation as the reduceByKey() method, but it creates an iterable for the values for a particular key. groupByKey() involves a lot of shuffling as it does not combine the keys present in the same executor.

mapValues(): This function is used for operating on the value part of a key–value pair.

flatMapValues(): To understand flatMapValues(), let's take a look at an example.

keys(): The keys() function is used to create a new RDD that contains only the keys from the paired RDD.

values(): The values() function is used to create a new RDD that contains only the values from the paired RDD.

Operations on Paired RDDs

Here are some operations that can be performed on paired RDDs:

sortByKey(): This operation is used to sort the elements of a paired RDD. The sorting is done based on the key of the paired RDD.

join(): Whenever you perform an inner join, the key must be present in both the paired RDDs; however, for an outer join, the key may or may not be present in both the paired RDDs.

Consider the following example:
rdd1 = [('Spark', 50), ('Dataframe', 100), ('API', 150), ('Dataset', 120)]
rdd2 = [('Spark', 100), ('Dataframe', 120), ('RDD',150)]

Now, if you apply a join operation on these two RDDs, then the output would be another RDD:
rdd3 = [('Spark', (50,100)), ('Dataframe', (100,120))]

rightOuterJoin(): rightOuterJoin() has an option to skip the keys that are present on the left side of the operator; however, all the keys that are present on the right side of the operator must be present.

leftOuterJoin(): leftOuterJoin() has an option to skip the keys that are present on the right side of the operator; however,all the keys that are present on the left side of the operator must be present.

cogroup(): In the case of cogroup(), if a key is present in any of the RDDs, then it will be present in the output.

countByKey(): This function is used to count the number of elements for each key.

lookup(key): This function is used to find all the values associated with the provided key.

Introduction to Structured APIs

RDDs have the following limitations:

1. Data stored with RDD abstraction is unstructured. While dealing with unstructured data, Spark recognises that there are parameters (or attributes) associated with each datapoint object. However, Spark still cannot read the inside object to get more details of the parameters.
2. RDD is a low-level abstraction. The code has very-low-level details about the execution of a job.

Spark has the following three structured APIs:

1. **DataFrames:** These are collections of data organised in a tabular form. DataFrames allow processing large amounts of structured data. One of the major differences between DataFrames and RDDs is that in DataFrames, data is organised in rows and columns, in contrast to RDDs. However, DataFrames do not have compile-time type safety.
2. **Datasets:** Datasets are an extension of DataFrames, and they include the features of both DataFrames and RDDs. Datasets provide an object-oriented interface for processing data

safely. Object-oriented interface refers to an interface where all the entities are treated as objects, and one has to call objects in order to access them. Note that Datasets are available only in JVM-based languages – Scala and Java – and not in Python and R. Datasets have compile-time type safety.

3. **SQL tables and views (SparkSQL):** With SparkSQL, you can run SQL-like queries against views or tables organised into databases.

Dataframes and Datasets

DataFrames offer the following benefits:

- **High-Level API:** Code written using DataFrame abstraction is structured similarly to DataFrames in Pandas. The code is highly readable and easy to write. Therefore, it becomes accessible to a lot of people other than specialists like data engineers or data scientists.

- **Catalyst Optimizer:** DataFrames have the ability to optimise code internally with the help of Catalyst Optimizer. It has the ability to rearrange code to make it run faster without any difference in output.

- **Memory Management:** As you have learnt in the previous module, transformations on RDDs are performed in executors. These transformations take place in the heap memory of the executors, which are essentially JVMs. On the other hand, DataFrames make use of off-heap memory, in addition to the heap memory on JVMs. Hence, managing both heap and off-heap memory requires custom memory management. As you go through the next couple of points, you will understand why it is beneficial to utilise off-heap memory.

- **Garbage Collection:** In languages like C/C++, the programmer is responsible for creating and deleting objects, although objects that are not useful are usually neglected. Such objects are not removed and remain in memory. Over time, these objects keep on accumulating in memory, and, sooner or later, the memory might become full with such garbage objects, resulting in Out Of Memory (OOM) errors. Java has a garbage collection program to manage such errors. This program collects all objects that are not in use and removes them from memory. You can say that the main objective of garbage collection (GC) is to free up heap memory.

- As the executor works on the job tasks, at some point, due to an increase in garbage objects, it has to run a ‘full garbage collection’. During this process, the machine slows down as it has to scan the entire heap memory. So, the larger the JVM memory, the more time that the machine would take to perform full GC. Note that GC runs only on JVM memory, which is heap memory.

- **Off-Heap Memory Storage:** An elegant solution to the problem above is to store data off the JVM heap, although it would still be in the RAM and not on the disk. With this method, one can allocate a large amount of memory off-heap without GC slowing down the machine. If the data is being stored off-heap, then the size of the JVM memory can be reduced, and the GC process would not affect performance as much as this process runs on only JVM memory, which is heap memory. Another advantage of off-heap memory is that the storage space required is much smaller compared with that for JVM memory, as Java objects in JVM require more storage space.

A Dataset has the following features:

- **High-Level API:** If you compare a code written in RDDs to a code written in Datasets, then you will notice that the code written in Dataset abstraction is elegant, readable and short. Since the instructions given are high-level, the Spark optimisation engine can determine ways to improve the performance of the code. This is one of the features of DataFrames that is also offered by Datasets.

- **Compile-Time Type Safety:** It is the ability of Dataset abstraction to detect errors in code during compile time itself.

- **Encoders:** The primary function of an encoder is to translate between heap memory, that is, JVM memory, and off-heap memory. The encoder achieves this by serializing and deserialising the data. Serialisation is the process of converting a JVM object into a byte stream. This conversion is necessary as off-heap memory understands byte streams and not objects. The data stored in off-heap memory is serialised in nature.

- When a Spark job is run, the job query is first optimised by Spark. At runtime, the encoder takes this optimised query and converts it into byte instructions. In response to these instructions, only the required data is deserialised. This also happens in the case of DataFrames.

- **Reduction in Memory Usage:** Datasets “understand” the data as it is structured and can create efficient layouts to reduce memory usage while caching.

Catalyst Optimizer

Catalyst Optimizer creates an internal representation of a user’s program, called a query plan. Once you have the initial version of the query plan, the Catalyst Optimizer will apply different transformations to convert it to an optimised query plan.

Catalyst Optimizer supports both rule-based and cost-based optimisation:

- In the rule-based optimisation phase, Catalyst Optimizer uses a set of rules to determine how to execute a query. During this optimisation phase, it generates multiple physical plans.

- In the cost-based optimisation phase, the cost for each physical plan is calculated and the physical plan with the lowest cost is selected for execution.

Once the best physical plan is ready, the optimised query plan gets converted into a DAG of RDDs, ready for execution.

Getting Started with Dataframe APIs

The following syntax is used for loading a csv file:

spark.read.load("filename.fileformat", format = "fileformat", inferSchema = True, header = True)

This syntax can be used to load any file format. There is one more method to specifically load csv files. This is as follows:

df = spark.read.csv("file name.csv", inferSchema = True, header = True)

For saving a DataFrame in JSON format, use the following command:
df.write.json("File_Name")

For saving a DataFrame in Parquet format, use the following command:
df.write.parquet("File_Name")

For saving a DataFrame in ORC format, use the following command:
df.write.orc("File_Name")

Dataframe Operations

Filter Command
The filter() command can be called on a DataFrame directly. As an argument to the filter method, we provide the column name and specify the condition over the column.

select(): Just as SQL select, this operation is used to select the columns that must be present in the output.

groupBy(): Just as Spark SQL, groupBy() is used for grouping rows in a table based on some value in a column. It is used to perform some aggregation on the grouped data.

orderBy(): This operation is used to arrange rows in certain order, which could be either ascending or descending.

Disclaimer: *All content and material on the upGrad website is copyrighted material, belonging to either upGrad or its bona fide contributors, and is purely for the dissemination of education. You are permitted to access, print, and download extracts from this site purely for your own education only and on the following basis:*

- *You can download this document from the website for self-use only.*
- *Any copies of this document, in part or full, saved to disc or to any other storage medium, may be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.*
- *Any further dissemination, distribution, reproduction, and copying of the content of the document herein, or the uploading thereof on other websites, or use of the content for any other commercial/unauthorized purposes in any way that could infringe the intellectual property rights of upGrad or its contributors is strictly prohibited.*
- *No graphics, images, or photographs from any accompanying text in this document will be used separately for unauthorized purposes.*
- *No material in this document will be modified, adapted, or altered in any way.*
- *No part of this document or upGrad content may be reproduced or stored on any other website or included in any public or private electronic retrieval system or service without upGrad's prior written permission.*
- *Any rights not expressly granted in these terms are reserved.*