APACHE SPARK

Apache spark- Advantages over Hadoop, lazy evaluation, In memory processing, DAG, Spark context, Spark Session, RDD, Transformations- Narrow and Wide, Actions, Data frames ,RDD to Data frames, Catalyst optimizer, Data Frame Transformations, Working with Dates and Timestamps, Working with Nulls in Data, Working with Complex Types, Working with JSON, Grouping, Window Functions, Joins, Data Sources, Broadcast Variables, Accumulators, Deploying Spark- On-Premises Cluster Deployments, Cluster Managers- Standalone Mode, Spark on YARN , Spark Logs, The Spark UI- Spark UI History Server, Debugging and Spark First Aid

4.0 Introduction:

Industries are using Hadoop extensively to analyze their data sets. The reason is that Hadoop framework is based on a simple programming model (MapReduce) and it enables a computing solution that is scalable, flexible, fault-tolerant and cost effective. Here, the main concern is to maintain speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.

Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational computing software process.

Spark is not a modified version of Hadoop and is not, really, dependent on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark.

Spark uses Hadoop in two ways – one is **storage** and second is **processing**. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

4.1 What is Apache Spark?

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

4.1.1 Features of Apache Spark –

Apache Spark has following features.

1. **Fast processing** – The most important feature of Apache Spark that has made the big data world choose this technology over others is its speed. Big data is characterized by volume, variety, velocity, and veracity which needs to be processed at a higher speed.

Spark contains Resilient Distributed Dataset (RDD) which saves time in reading and writing operations, allowing it to run almost **ten to one hundred times faster than Hadoop**.

2. **Flexibility** – Apache Spark supports multiple languages and allows the developers to write applications in Java, Scala, R, or Python.
3. **In-memory computing** – Spark stores the data in the RAM of servers which allows quick access and in turn accelerates the speed of analytics.
4. **Real-time processing** – Spark is able to process real-time streaming data. Unlike MapReduce which processes only stored data, Spark is able to process real-time data and is, therefore, able to produce instant outcomes.
5. **Better analytics** – In contrast to MapReduce that includes Map and Reduce functions, Spark includes much more than that. Apache Spark consists of a rich set of SQL queries, machine learning algorithms, complex analytics, etc. With all these functionalities, analytics can be performed in a better fashion with the help of Spark.

4.2 Core components of Spark –

1. **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.
2. **Spark SQL** – Spark SQL is Apache Spark's module for working with structured data. The interfaces offered by Spark SQL provides Spark with more information about the structure of both the data and the computation being performed.
3. **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.
4. **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.
5. **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.

4.3 Advantages of Spark over Hadoop –

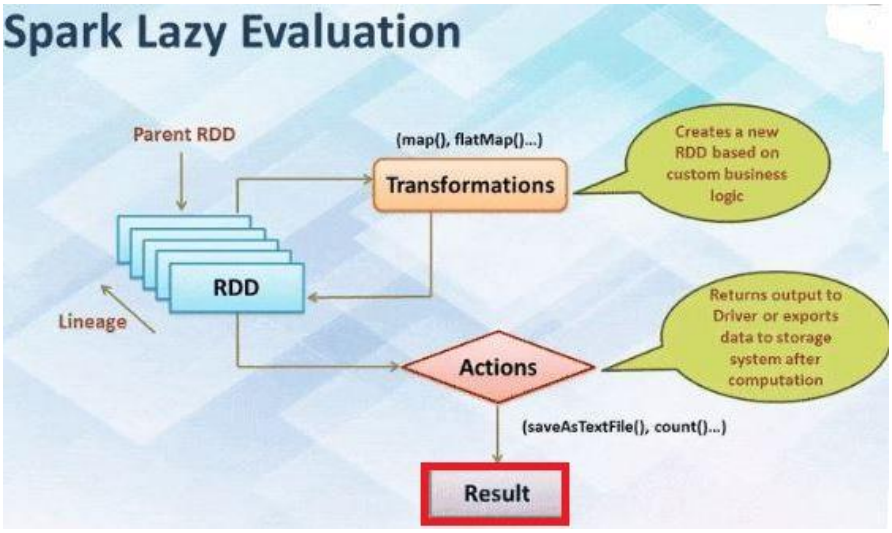
Apache Spark and Hadoop MapReduce are two popular big data processing frameworks used in the industry. Both of these frameworks have the capability to handle large-scale data processing, but they differ in terms of their architecture and design.

The five key differences between **MapReduce vs. Spark**:

1. **Processing speed:** Apache Spark is much faster than Hadoop MapReduce.
2. **Data processing paradigm:** Hadoop MapReduce is designed for batch processing, while Apache Spark is more suited for real-time data processing and iterative analytics.
3. **Ease of use:** Apache Spark has a more user-friendly programming interface and supports multiple languages, while Hadoop MapReduce requires developers to write code in Java.
4. **Fault tolerance:** Apache Spark's Resilient Distributed Datasets (RDDs) offer better fault tolerance than Hadoop MapReduce's Hadoop Distributed File System (HDFS).
5. **Integration:** Apache Spark has a more extensive ecosystem and integrates well with other big data tools, while Hadoop MapReduce is primarily designed to work with Hadoop Distributed File System (HDFS).

4.4 Lazy Evaluation –

Lazy evaluation in Spark means that the execution will not start until an action is triggered. In Spark, the picture of lazy evaluation comes when Spark transformations occur.



Transformations are lazy in nature meaning when we call some operation in RDD, it does not execute immediately. Spark maintains the record of which operation is being called (Through DAG). We can think Spark RDD as the data that we built up through transformation. Since transformations are lazy in nature, so we can execute operation any time by calling an action on data. Hence, in lazy evaluation data is not loaded until it is necessary.

4.5 In memory processing –

In Apache Spark, In-memory computation defines as instead of storing data in some slow disk drives the data is kept in random access memory(RAM). Also, that data is processed in parallel. By using in-memory processing, we can detect a pattern, analyze large data. It reduces the cost of memory, therefore, it became popular. So, it resulted very economic for applications. Main columns of in-memory computation are categorized as-

1. RAM storage
2. Parallel distributed processing.

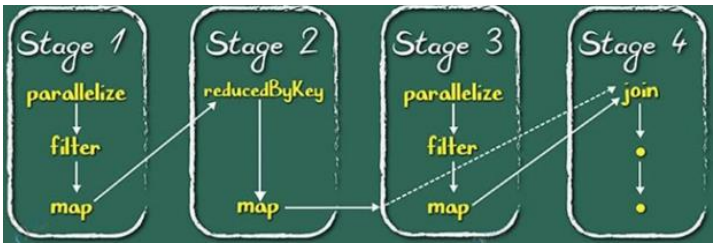
If we Keep the data in-memory, it improves the performance by an order of magnitudes. As RDDs are the main abstraction in Spark, RDDs are cached using persist() or the cache() method. All the RDD is stored in-memory, while we use cache() method. As RDD stores the value in memory, the data which does not fit in memory is either recalculated or the excess data is sent to disk. In addition, it can be extracted without going to disk, whenever we want RDD. This process decreases the space-time complexity and also reduces the overhead of disk storage.

Spark's in-memory capability is good for micro-batch processing and machine learning. It also offers faster execution of iterative jobs.

The RDDs can also be stored in-memory while we use persist() method. Also, we can use it across parallel operations. There is only one difference between cache() and persist(). while using cache() the default storage level is MEMORY_ONLY. And, while using persist() we can use various storage levels.

4.6 Directed Acyclic Graph –

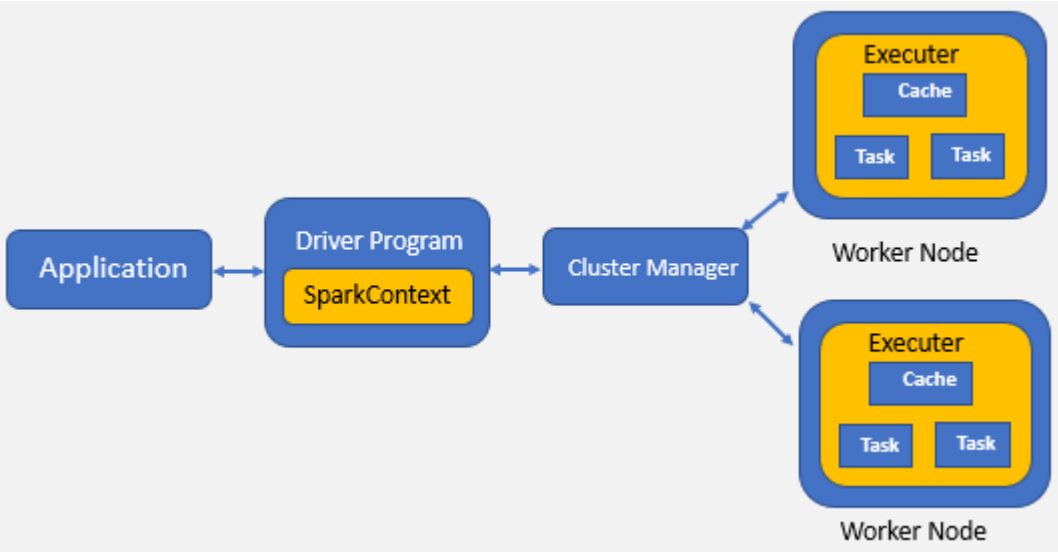
(Directed Acyclic Graph) DAG in Apache Spark is a set of **Vertices** and **Edges**, where vertices represent the **RDDs** and the edges represent the **Operation to be applied on RDD**. In Spark DAG, every edge directs from earlier to later in the sequence. On the calling of Action, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task.



4.7 Spark Context –

SparkContext is the entry gate of Apache Spark functionality. The most important step of any Spark driver application is to generate SparkContext. It allows your Spark Application to access Spark Cluster with the help of Resource Manager (YARN/Mesos). To create SparkContext, first SparkConf should be made. The SparkConf has a configuration parameter that our Spark driver application will pass to SparkContext.

A SparkContext represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.



Note:

Only one SparkContext should be active per JVM. You must stop() the active SparkContext before creating a new one.

4.7.1 Spark Applications-

Spark Applications consist of a driver process and a set of executor processes. The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user's program or input; and analyzing, distributing, and scheduling work across the executors (discussed momentarily). The driver process is absolutely essential—it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The executors are responsible for actually carrying out the work that the driver assigns them. This means that each executor is responsible for only two things: executing code assigned to it by the driver, and reporting the state of the computation on that executor back to the driver node.

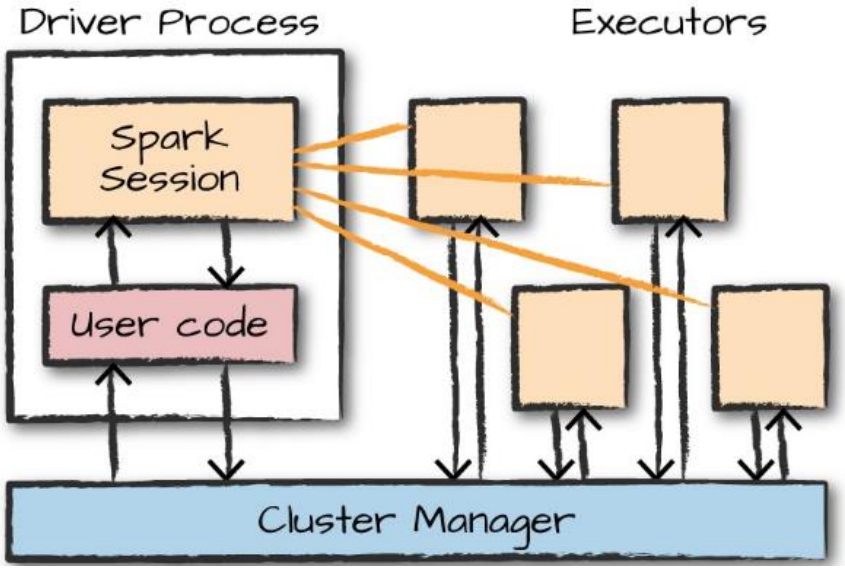


Figure4.1: The architecture of Spark application

4.7.2 SparkSession -

Spark Application through a driver process called the SparkSession. The SparkSession instance is the way Spark executes user-defined manipulations across the cluster. There is a one-to-one correspondence between a SparkSession and a Spark Application. In Scala and Python, the variable is available as spark when you start the console. Let's go ahead and look at the SparkSession in both Scala and/or Python:

Spark:

In Scala, you should see something like the following:

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@...
```

In Python you'll see something like this:

```
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

Example: perform the simple task of creating a range of numbers. This range of numbers is just like a named column in a spreadsheet:

```
// in Scala
val myRange = spark.range(1000).toDF("number")
# in Python
myRange = spark.range(1000).toDF("number")
```

You just ran your first Spark code! We created a DataFrame with one column containing 1,000 rows with values from 0 to 999. This range of numbers represents a distributed collection. When run on a cluster, each part of this range of numbers exists on a different executor. This is a Spark DataFrame.

4.7.3 RDD -

RDDs are the main logical data units in Spark. They are a distributed collection of objects, which are stored in memory or on disks of different machines of a cluster. A single RDD can be divided into multiple logical partitions so that these partitions can be stored and processed on different machines of a cluster.

RDDs are immutable (read-only) in nature. You cannot change an original RDD, but you can create new RDDs by performing coarse-grain operations, like transformations, on an existing RDD.

An RDD in Spark can be cached and used again for future transformations, which is a huge benefit for users. RDDs are said to be lazily evaluated, i.e., they delay the evaluation until it is really needed. This saves a lot of time and improves efficiency.

Features of an RDD in Spark

- **Resilience:** RDDs track data lineage information to recover lost data, automatically on failure. It is also called fault tolerance.
- **Distributed:** Data present in an RDD resides on multiple nodes. It is distributed across different nodes of a cluster.
- **Lazy evaluation:** Data does not get loaded in an RDD even if you define it. Transformations are actually computed when you call action, such as count or collect, or save the output to a file system.

Operations on RDDs -

There are two basic operations that can be done on RDDs. They are transformations and actions.

1. Transformations

Coarse grained operations like join, union, filter or map on existing RDDs which produce a new RDD, with the result of the operation, are referred to as transformations All transformations in Spark are lazy.

2. Actions

Actions in Spark are functions that return the end result of RDD computations. It uses a lineage graph to load data onto the RDD in a particular order. After all of the transformations are done, actions return the final result to the Spark Driver. Actions are operations like count, first and reduce which return values after computations on existing RDDs are referred to as Actions.

4.7.4 Transformations –

In Spark, the core data structures are immutable, meaning they cannot be changed after they're created. This might seem like a strange concept at first: if you cannot change it, how are you supposed to use it? To "change" a DataFrame, you need to instruct Spark how you would like to modify it to do what you want. These instructions are called transformations. Let's perform a simple transformation to find all even numbers in our current DataFrame:

// in Scala

```
val divisBy2 = myRange.where("number % 2 = 0")
```

 # in Python

```
divisBy2 = myRange.where("number % 2 = 0")
```

Notice that these return no output. This is because we specified only an abstract transformation, and Spark will not act on transformations until we call an action.

There are two types of transformations: **narrow dependencies**, and those that specify **wide dependencies**.

1. Narrow Transformations

Transformations consisting of narrow dependencies (we'll call them narrow transformations) are those for which each input partition will contribute to only one output partition. In the preceding code snippet, the where statement specifies a narrow dependency, where only one partition contributes to at most one output partition, as you can see in following figure.

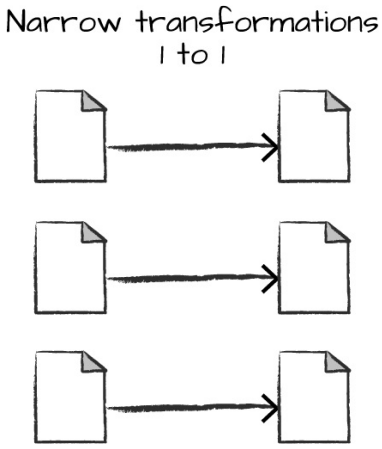


Figure4.2: A narrow dependency

2. Wide Transformations

A **wide dependency** (or wide transformation) style transformation will have input partitions contributing to many output partitions. Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called pipelining,

meaning that if we specify multiple filters on DataFrames, they'll all be performed in-memory. The same cannot be said for shuffles. When we perform a shuffle, Spark writes the results to disk. Wide transformations are illustrated in Figure.

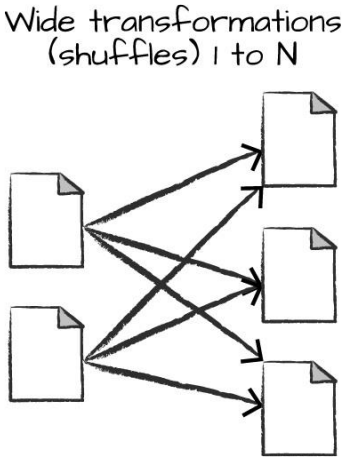


Figure4.3: A wide dependency

4.7.5 Actions –

Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an action. An action instructs Spark to compute a result from a series of transformations. The simplest action is count, which gives us the total number of records in the DataFrame:

Example: divisBy2.count()

There are three kinds of actions:

- Actions to view data in the console
- Actions to collect data to native objects in the respective language
- Actions to write to output data sources

In specifying this action, we started a Spark job that runs our filter transformation (a narrow transformation), then an aggregation (a wide transformation) that performs the counts on a per partition basis, and then a collect, which brings our result to a native object in the respective language. You can see all of this by inspecting the Spark UI, a tool included in Spark with which you can monitor the Spark jobs running on a cluster.

4.7.6 DataFrames –

A **DataFrame** is the most common Structured API and simply represents a table of data with rows and columns. The list that defines the columns and the types within those columns is called the schema. A DataFrame as a spreadsheet with named columns.

The following figure illustrates the fundamental difference: a **spreadsheet** sits on one computer in one specific location, whereas a Spark **DataFrame** can span thousands of computers. The reason for putting the data on more than one computer should be intuitive: either the data is too large to fit on one machine or it would simply take too long to perform that computation on one machine.

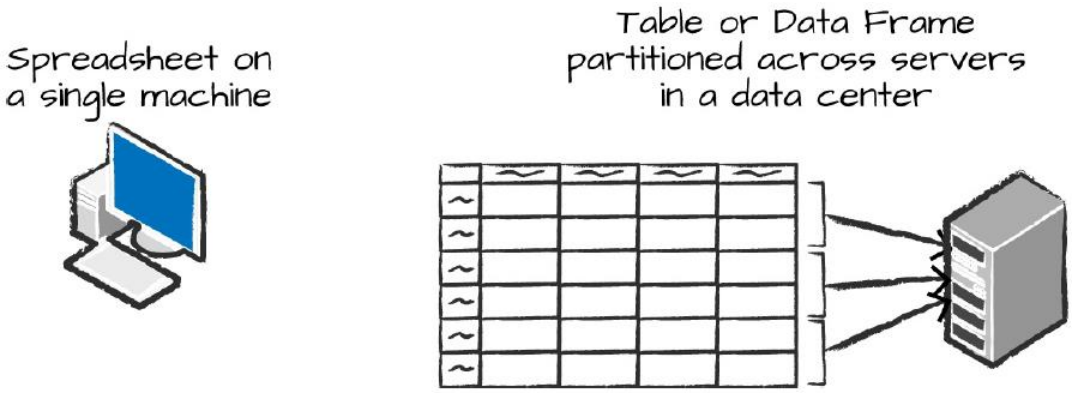


Figure4.4: Distributed versus single-machine analysis

DataFrame creation:

There are various ways to create a DataFrame:

1. The most basic way is to transform another DataFrame.

For example:

```
# transformation of one DataFrame creates another DataFrame
df2 = df1.orderBy('age')
```

2. You can also create a DataFrame from an RDD. RDD is a low-level data structure in Spark which also represents distributed data, and it was used mainly before Spark 2.x. It is slowly becoming more like an internal API in Spark but you can still use it if you want and in particular, it allows you to create a DataFrame as follows:

```
df = spark.createDataFrame(rdd, schema)
```

3. The next and more useful way (especially for prototyping) is to create a DataFrame from a local collection, for example, from a list:

```
l = [(1, 'Alice'), (2, 'Bob')] # each tuple will become a row
df = spark.createDataFrame(l, schema=['id', 'name'])
```

4. For prototyping, it is also useful to quickly create a DataFrame that will have a specific number of rows with just a single column id using a sequence:

```
df = spark.range(10) # creates a DataFrame with one column id
```

4.7.7 RDD to DataFrames –

RDD and DataFrame are two major APIs in Spark for holding and processing data. RDD provides us with low-level APIs for processing distributed data. On the other hand, DataFrame provides us with higher-level APIs that support SQL methods.

Converting RDDs to DataFrames and vice versa. There are two methods.

1. Let's first create our **SparkContext** and an **RDD** using a sequence of tuples:

```
val spark: SparkSession = SparkSession.builder().master("local").getOrCreate
val sc = spark.sparkContext
-----------------------------------------------------------------------------------------------------------
val rdd = sc.parallelize(
  Seq(
    ("John", "Manager", 38),
    ("Mary", "Director", 45),
    ("Sally", "Engineer", 30)
  )
)
```

2. **Convert Using createDataFrame Method -**

The SparkSession object has a utility method for creating a DataFrame – createDataFrame. This method can take an RDD and create a DataFrame from it. The createDataFrame is an overloaded method, and we can call the method by passing the RDD alone or with a schema.

To convert RDD to a DataFrame using a predefined schema object. The overloaded method createDataFrame takes schema as a second parameter, but it now accepts only RDDs of type Row. Therefore, we'll convert our initial RDD to an RDD of type Row:

```
val rowRDD:RDD[Row] = rdd.map(t => Row(t._1, t._2, t._3))
```

Next, we need is a schema object, so let's create one:

```
val schema = new StructType()
  .add(StructField("Name", StringType, false))
  .add(StructField("Job", StringType, true))
```

```
  .add(StructField("Age", IntegerType, true))
```

Let's call the method once again, now with an additional schema parameter:

```
val dfWithSchema:DataFrame = spark.createDataFrame(rowRDD, schema)
```

Print the schema information once again:

```
dfWithSchema.printSchema()
```

|-- Name: string (nullable = false)

|-- Job: string (nullable = true)

|-- Age: integer (nullable = true)

3. Conversion Using toDF() Implicit Method –

Another popular method of converting RDD to DataFrame is by using the .toDF() implicit method. Before we start, we must import the implicits from SparkSession:

```
import spark.implicits._
```

We're now ready to convert our RDD. However, this method works only for selected types of RDDs – Int, Long, String, or any sub-classes of scala.Product. We have an RDD created using a sequence of Tuples. Let's convert that using our imported implicit method:

```
val dfUsingToDFMethod = rdd.toDF("Name", "Job", "Age")
```

Let's inspect the schema of our new DataFrame:

```
dfUsingToDFMethod.printSchema()
```

|-- Name: string (nullable = true)

|-- Job: string (nullable = true)

|-- Age: integer (nullable = false)

4.8 Catalyst optimizer –

At the core of Spark SQL is the Catalyst optimizer, the execution of a single structured API query from user code to executed code. Here's an overview of the steps:

1. Write DataFrame/Dataset/SQL Code.
2. If valid code, Spark converts this to a Logical Plan.
3. Spark transforms this Logical Plan to a Physical Plan, checking for optimizations along the way.

4. Spark then executes this Physical Plan (RDD manipulations) on the cluster.

To execute code, we must write code. This code is then submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user. The following figure shows the process.

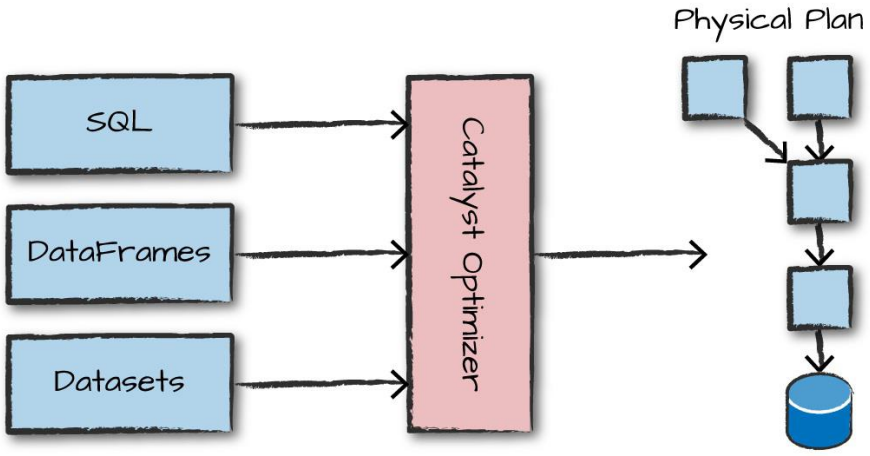


Figure4.5: Catalyst optimizer

4.9 DataFrame Transformations -

When working with individual DataFrames there are some fundamental objectives. These break down into several core operations such as –

- We can add rows or columns
- We can remove rows or columns
- We can transform a row into a column (or vice versa)
- We can change the order of rows based on the values in columns

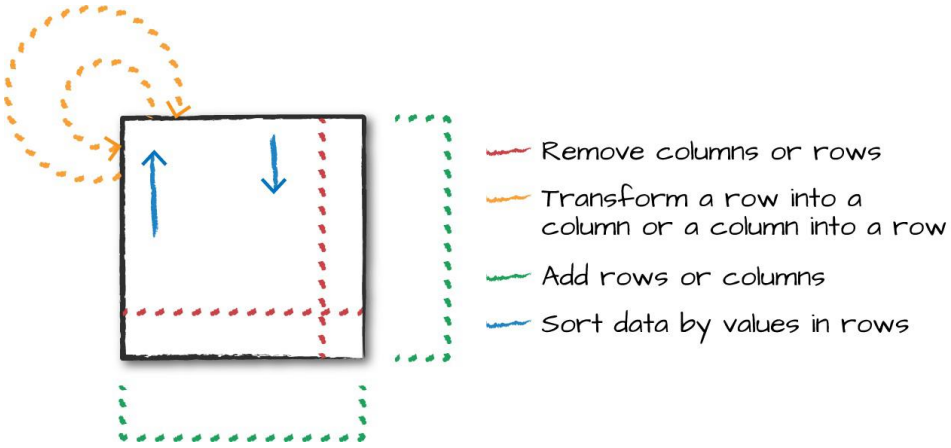


Figure4.6. Different kinds of transformations

To do DataFrame transformations. We can create DataFrames from raw data sources. We can query it with SQL and show off basic transformations:

1. By transforming a set of rows to column

// in Scala

```
val df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
df.createOrReplaceTempView("dfTable")
```

 # in Python

```
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
df.createOrReplaceTempView("dfTable")
```

We can also create DataFrames by taking a set of rows and converting them to a DataFrame.

// in Scala

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}
val myManualSchema = new StructType(Array(
new StructField("some", StringType, true),
new StructField("col", StringType, true),
new StructField("names", LongType, false)))
val myRows = Seq(Row("Hello", null, 1L))
val myRDD = spark.sparkContext.parallelize(myRows)
val myDf = spark.createDataFrame(myRDD, myManualSchema)
myDf.show()
```

// in Scala

```
val myDF = Seq(("Hello", 2, 1L)).toDF("col1", "col2", "col3")
```

 # in Python

```
from pyspark.sql import Row
```

```
from pyspark.sql.types import StructField, StructType, StringType, LongType
myManualSchema = StructType([
StructField("some", StringType(), True),
StructField("col", StringType(), True),
StructField("names", LongType(), False)
])
myRow = Row("Hello", None, 1)
myDf = spark.createDataFrame([myRow], myManualSchema)
myDf.show()
```

Output:

```
+-----+----+-----+
| some| col|names|
+-----+----+-----+
|Hello|null|    1|
+-----+----+-----+
```

2. select and selectExpr

select and **selectExpr** allow you to do the DataFrame equivalent of SQL queries on a table of data:

a) Using select -- in SQL

SELECT * FROM dataFrameTable

SELECT columnName FROM dataFrameTable

SELECT columnName * 10, otherColumn, someOtherCol as c FROM dataFrameTable

In the simplest possible terms, you can use them to manipulate columns in your DataFrames is to use the select method and pass in the column names as strings with which you would like to work:

// in Scala

```
df.select("DEST_COUNTRY_NAME").show(2)
```

 # in Python

```
df.select("DEST_COUNTRY_NAME").show(2)
```

Output:

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
+-----------------+
```

b) Using selectExpr

We can treat selectExpr as a simple way to build up complex expressions that create new DataFrames. In fact, we can add any valid non-aggregating SQL statement, and as long as the columns resolve, it will be valid! Here's a simple example that adds a new column withinCountry to our DataFrame that specifies whether the destination and origin are the same:

// in Scala

```
df.selectExpr(
        "*", // include all original columns
        "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")
        .show(2)
```

 # in Python

```
df.selectExpr(
        "*", # all original columns
        "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")\
        .show(2)
```

-- in SQL

```
SELECT *, (DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry
FROM dfTable
LIMIT 2
```

Output:

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
```

4.10 Working with Dates and Timestamps -

Spark SQL supports almost all date and time functions that are supported in Apache Hive. You can use these Spark DataFrame date functions to manipulate the date frame columns that contains date type values.

The Spark SQL built-in date functions are user and performance friendly. Use these functions whenever possible instead of Spark SQL user defined functions.

Spark Date Functions

Following are the Spark SQL date functions. The list contains pretty much all date functions that are supported in Apache Spark.

Spark Date Function	**Description**
date_format(date, format)	Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument.
current_date()	Returns the current date as a date column.
date_add(start, days)	Add days to the date.
add_months(start, months)	Add months to date.
datediff(end, start)	Returns difference between two dates in days.
year(col)	Extract the year of a given date as integer.

quarter(col)	Extract the quarter of a given date as integer.
month(col)	Extract the month of a given date as integer.
hour(col)	Extract the hours of a given date as integer.
minute(col)	Extract the minutes of a given date as integer.
second(col)	Extract the seconds of a given date as integer.
dayofmonth(col)	Extract the day of the month of a given date as integer.
date_sub(start, days)	Subtract the days from date field.
dayofyear(col)	Extract the day of the year of a given date as integer.
last_day(date)	Returns the last day of the month which the given date belongs to.
months_between(date1, date2)	Returns number of months between two dates.
next_day(date, dayOfWeek)	Returns the first date which is later than the value of the date column.
trunc(date, format)	Returns date truncated to the unit specified by the *format*.
weekofyear(col)	Extract the week number of a given date as integer.
to_date(col)	Convert string type containing date value to date format.

Spark Timestamp Functions

Following are the timestamp functions supported in Apache Spark.

Spark Timestamp Function	Description
current_timestamp()	Returns the current timestamp as a timestamp column
from_unixtime(timestamp, format=”yyyy-MM-dd HH:mm:ss”)	Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format.
unix_timestamp(timestamp=None, format=’yyyy-MM-dd HH:mm:ss’)	Convert time string with given pattern (‘yyyy-MM-dd HH:mm:ss’, by default) to Unix time stamp (in seconds), using the default timezone and the default locale, return null if fail.
from_utc_timestamp(timestamp, tz)	Given a timestamp, which corresponds to a certain time of day in UTC, returns another timestamp that corresponds to the same time of day in the given timezone.
to_utc_timestamp(timestamp, tz)	Given a timestamp, which corresponds to a certain time of day in the given timezone, returns another timestamp that corresponds to the same time of day in UTC.

We will be using following sample DataFrame in our date and timestamp function examples.

```
testDF = sqlContext.createDataFrame([("2020-01-01","2020-01-31")], ["start_date", end_date"])
```

Import Functions in PySpark Shell

Before trying to use Spark date functions, you need to import the functions in pyspark shell. Otherwise, you will end up getting “NameError: name ‘current_date’ is not defined” error.

```
from pyspark.sql.functions import *
```

1. Get current date in PySpark SQL

```
testDF.select( current_date().alias("current_dt")).show()
+----------+
|current_dt|
+----------+
|2020-01-31|
+----------+
```

2. Change the date format using date_format function in spark SQL

```
testDF.select("start_date", date_format("start_date",
'dd/MM/yyyy').alias("dt_format")).show()
+----------+----------+
|start_date| dt_format|
+----------+----------+
|2020-01-01|01/01/2020|
+----------+----------+
```

3. Add days to date using date_add function in Spark SQL

```
testDF.select("start_date",
date_add("start_date",2).alias("date_days")).show()
+----------+----------+
|start_date| date_days|
+----------+----------+
|2020-01-01|2020-01-03|
+----------+----------+
```

4. Add months to the date using add_months function in Spark SQL

```
testDF.select("start_date", add_months("start_date",2).alias(
"add_months")).show()
+----------+----------+
|start_date|add_months|
+----------+----------+
|2020-01-01|2020-03-01|
+----------+----------+
```

5. Get difference between two dates using datediff function in Spark SQL

```
testDF.select("start_date", "end_date", datediff("end_date",
"start_date").alias("date_diff")).show()
+----------+----------+---------+
|start_date|  end_date|date_diff|
+----------+----------+---------+
|2020-01-01|2020-01-31|       30|
+----------+----------+---------+
```

6. Extract year from date using Spark SQL Function

```
testDF.select("start_date", year( "start_date").alias("year")).show()
+----------+----+
|start_date|year|
+----------+----+
|2020-01-01|2020|
+----------+----+
```

7. Subtract days from date using date_sub function in Spark SQL

```
testDF.select("start_date", date_sub("start_date",
3).alias("date_sub")).show()
+----------+----------+
|start_date|  date_sub|
+----------+----------+
|2020-01-01|2019-12-29|
+----------+----------+
```

8. Get last day of the month using last_day function in Spark SQL

```
testDF.select("start_date", last_day( "start_date").alias("last_day")).show()
+----------+----------+
|start_date|  last_day|
+----------+----------+
|2020-01-01|2020-01-31|
+----------+----------+
```

9. Identify months between two dates using months_between function in Spark SQL

```
testDF.select("start_date", "end_date", months_between(
"end_date","start_date").alias("months")).show()
+----------+----------+----------+
|start_date|  end_date|    months|
+----------+----------+----------+
|2020-01-01|2020-01-31|0.96774194|
+----------+----------+----------+
```

10. Get next Monday using next_day function in Spark SQL

```
testDF.select("start_date", next_day( "start_date",
"mon").alias("next_monday")).show()
+----------+-----------+
|start_date|next_monday|
+----------+-----------+
|2020-01-01| 2020-01-06|
+----------+-----------+
```

11. Convert String to Date using to_date function in Spark SQL

```
testDF.select("start_date", to_date( "start_date").alias("to_date")).show()
+----------+----------+
|start_date|   to_date|
+----------+----------+
|2020-01-01|2020-01-01|
+----------+----------+
```

Spark SQL Timestamp Functions

1. Get Current Timestamp using current_timestamp() function

```
testDF.select(current_timestamp().alias( "current_timestamp")).show()
+--------------------+
|   current_timestamp|
+--------------------+
|2020-01-31 23:12:...|
+--------------------+
```

4.11 Working with NULLs in Data -

"null means that some value is unknown, missing, or irrelevant." Spark DataFrame best practices are aligned with SQL best practices, so DataFrames should use null for values that are unknown, missing or irrelevant.

Spark can optimize working with null values more than it can if you use empty strings or other values. The primary way of interacting with null values, at DataFrame scale, is to use the .na subpackage on a DataFrame. There are two things you can do with null values: you can explicitly drop nulls or you can fill them with a value.

1) Coalesce

Spark includes a function to allow you to select the first non-null value from a set of columns by using the coalesce function. In this case, there are no null values, so it simply returns the first column:

```
// in Scala
import org.apache.spark.sql.functions.coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()

# in Python
from pyspark.sql.functions import coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

2) drop

The simplest function is drop, which removes rows that contain nulls. The default is to drop any row in which any value is null:

```
df.na.drop()
df.na.drop("any")
```

Specifying "any" as an argument drops a row if any of the values are null. Using "all" drops the row only if all values are null or NaN for that row:

df.na.drop("all")

We can also apply this to certain sets of columns by passing in an array of columns:

```
// in Scala
df.na.drop("all", Seq("StockCode", "InvoiceNo"))

# in Python
df.na.drop("all", subset=["StockCode", "InvoiceNo"])
```

3) fill

Using the fill function, you can fill one or more columns with a set of values. This can be done by specifying a map—that is a particular value and a set of columns.

For example, to fill all null values in columns of type String, you might specify the following:

```
df.na.fill("All Null values become this string")
```

We could do the same for columns of type Integer by using df.na.fill(5:Integer), or for Doubles df.na.fill(5:Double). To specify columns, we just pass in an array of column names like we did in the previous example:

```
// in Scala
df.na.fill(5, Seq("StockCode", "InvoiceNo"))

# in Python
df.na.fill("all", subset=["StockCode", "InvoiceNo"])
```

We can also do this with with a Scala Map, where the key is the column name and the value is the value we would like to use to fill null values:

```
// in Scala
val fillColValues = Map("StockCode" -> 5, "Description" -> "No Value")
df.na.fill(fillColValues)

# in Python
fill_cols_vals = {"StockCode": 5, "Description" : "No Value"}
df.na.fill(fill_cols_vals)
```

4) replace

In addition to replacing null values like we did with drop and fill, there are more flexible options that you can use with more than just null values. Probably the most common use case is to replace all values in a certain column according to their current value. The only requirement is that this value be the same type as the original value:

```
// in Scala
df.na.replace("Description", Map("" -> "UNKNOWN"))

# in Python
df.na.replace([""], ["UNKNOWN"], "Description")
```

5) Ordering

you can use asc_nulls_first, desc_nulls_first, asc_nulls_last, or desc_nulls_last to specify where you would like your null values to appear in an ordered DataFrame.

4.12 Working with Complex Types -

Complex types can help you organize and structure your data in ways that make more sense for the problem that you are hoping to solve. There are three kinds of complex types: structs, arrays, and maps.

1) Structs

You can think of structs as DataFrames within DataFrames. We can create a struct by wrapping a set of columns in parenthesis in a query:

```
df.selectExpr("(Description, InvoiceNo) as complex", "*")

df.selectExpr("struct(Description, InvoiceNo) as complex", "*")

// in Scala
import org.apache.spark.sql.functions.struct
val complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")

# in Python
from pyspark.sql.functions import struct
complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")
```

2) Arrays

To define arrays, with our current data, our objective is to take every single word in our Description column and convert that into a row in our DataFrame.

The first task is to turn our Description column into a complex type, an array.

a. split

We do this by using the split function and specify the delimiter:

// in Scala

```
import org.apache.spark.sql.functions.split

df.select(split(col("Description"), " ")).show(2)
```

 # in Python

```
from pyspark.sql.functions import split

df.select(split(col("Description"), " ")).show(2)
```

-- in SQL

```
SELECT split(Description, ' ') FROM dfTable
```

```
+--------------------+
|split(Description,  )|
+--------------------+
| [WHITE, HANGING, ...|
| [WHITE, METAL, LA...|
+--------------------+
```

b. Array Length

We can determine the array's length by querying for its size:

c. array_contains

We can also see whether this array contains a value:

// in Scala

```
import org.apache.spark.sql.functions.array_contains

df.select(array_contains(split(col("Description"), " "), "WHITE")).show(2)
```

 # in Python

```
from pyspark.sql.functions import array_contains

df.select(array_contains(split(col("Description"), " "), "WHITE")).show(2)
```

-- in SQL

```
SELECT array_contains(split(Description, ' '), 'WHITE') FROM dfTable
```

Output:

```
+-----------------------------------------+
|array_contains(split(Description,  ), WHITE)|
+-----------------------------------------+
|                                     true|
|                                     true|
+-----------------------------------------+
```

3) Maps

Maps are created by using the map function and key-value pairs of columns. You then can select them just like you might select from an array:

```
// in Scala
import org.apache.spark.sql.functions.map
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map")).show(2)

# in Python
from pyspark.sql.functions import create_map
df.select(create_map(col("Description"), col("InvoiceNo")).alias("complex_map"))\
  .show(2)
```

Output:

```
+--------------------+
|         complex_map|
+--------------------+
|Map(WHITE HANGING...|
|Map(WHITE METAL L...|
+--------------------+
```

4.13 Grouping -

A more common task is to perform calculations based on groups in the data. This is typically done on categorical data for which we group our data on one column and perform some calculations on the other columns that end up in that group.

Spark also supports advanced aggregations to do multiple aggregations for the same input record set via GROUPING SETS, CUBE, ROLLUP clauses.

Syntax:

```
GROUP BY group_expression [ , group_expression [ , ... ] ] [ WITH { ROLLUP | CUBE } ]

GROUP BY { group_expression | { ROLLUP | CUBE | GROUPING SETS } (grouping_set [ , ...]) } [ , ... ]
```

Parameters –

1) group_expression

Specifies the criteria based on which the rows are grouped together. The grouping of rows is performed based on result values of the grouping expressions. A grouping expression may be a column name like GROUP BY a, a column position like GROUP BY 0, or an expression like GROUP BY a + b.

2) grouping_set

A grouping set is specified by zero or more comma-separated expressions in parentheses. When the grouping set has only one element, parentheses can be omitted. **For example**, GROUPING SETS ((a), (b)) is the same as GROUPING SETS (a, b).

Syntax: { ([expression [, ...]]) | expression }

3) ROLLUP

Specifies multiple levels of aggregations in a single statement. This clause is used to compute aggregations based on multiple grouping sets. ROLLUP is shorthand for GROUPING SETS.

For example, Let's create a rollup that looks across time (with our new Date column) and space (with the Country column) and creates a new DataFrame that includes the grand total over all dates, the grand total for each date in the DataFrame, and the subtotal for each country on each date in the DataFrame:

```
val rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))

.selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")

.orderBy("Date")

rolledUpDF.show()
```

 # in Python

```
rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))\

.selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")\

.orderBy("Date")

rolledUpDF.show()
```

```
+----------+--------------+--------------+
|      Date|       Country|total_quantity|
+----------+--------------+--------------+
|      null|          null|       5176450|
|2010-12-01|United Kingdom|         23949|
|2010-12-01|       Germany|           117|
|2010-12-01|        France|           449|
...
|2010-12-03|        France|           239|
|2010-12-03|         Italy|           164|
|2010-12-03|       Belgium|           528|
+----------+--------------+--------------+
```

Now where you see the null values is where you'll find the grand totals. A null in both rollup columns specifies the grand total across both of those columns:

```
rolledUpDF.where("Country IS NULL").show()

rolledUpDF.where("Date IS NULL").show()
```

```
+----+-------+--------------+
|Date|Country|total_quantity|
+----+-------+--------------+
|null|   null|       5176450|
+----+-------+--------------+
```

4) CUBE

CUBE clause is used to perform aggregations based on combination of grouping columns specified in the GROUP BY clause. CUBE is shorthand for GROUPING SETS.

// in Scala

```
dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))

.select("Date", "Country", "sum(Quantity)").orderBy("Date").show()
```

 # in Python

```
from pyspark.sql.functions import sum

dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))\

.select("Date", "Country", "sum(Quantity)").orderBy("Date").show()
```

```
+----+--------------------+-------------+
|Date|             Country|sum(Quantity)|
+----+--------------------+-------------+
|null|               Japan|        25218|
|null|            Portugal|        16180|
|null|         Unspecified|         3300|
|null|                null|      5176450|
|null|           Australia|        83653|
...
|null|              Norway|        19247|
|null|           Hong Kong|         4769|
|null|               Spain|        26824|
|null|      Czech Republic|          592|
+----+--------------------+-------------+
```

4.14 Window Functions -

Window functions operate on a group of rows, referred to as a window, and calculate a return value for each row based on the group of rows. Window functions are useful for processing tasks such as calculating a moving average, computing a cumulative statistic, or accessing the value of rows given the relative position of the current row.

There are mainly three types of Window function:

1. Analytical Function
2. Ranking Function
3. Aggregate Function

To perform window function operation on a group of rows first, we need to partition i.e. define the group of data rows using window.partition() function, and for row number and rank function we need to additionally order by on partition data using ORDER BY clause.

Syntax for Window.partition:

Window.partitionBy("column_name").orderBy("column_name")

Syntax for Window function:

DataFrame.withColumn("new_col_name", Window_function().over(Window_partition))

1. Analytical functions –

An analytic function is a function that returns a result after operating on data or a finite set of rows partitioned by a SELECT clause or in the ORDER BY clause. It returns a result in the same number of rows as the number of input rows. E.g. lead(), lag(), cume_dist().

Creating dataframe for demonstration:

Before we start with these functions, first we need to create a DataFrame. We will create a DataFrame that contains employee details like Employee_Name, Age, Department, Salary. After creating the DataFrame we will apply each analytical function on this DataFrame df.

```
# importing pyspark

from pyspark.sql.window import Window

import pyspark

# importing sparksession

from pyspark.sql import SparkSession
```

```
# creating a sparksession object and providing appName
spark = SparkSession.builder.appName("pyspark_window").getOrCreate()
# sample data for dataframe
sampleData = (("Ram", 28, "Sales", 3000),
                        ("Meena", 33, "Sales", 4600),
                        ("Robin", 40, "Sales", 4100),
                        ("Kunal", 25, "Finance", 3000),
                        ("Ram", 28, "Sales", 3000),
                        ("Srishti", 46, "Management", 3300),
                        ("Jeny", 26, "Finance", 3900),
                        ("Hitesh", 30, "Marketing", 3000),
                        ("Kailash", 29, "Marketing", 2000),
                        ("Sharad", 39, "Sales", 4100)
                        )
# column names for dataframe
columns = ["Employee_Name", "Age", "Department", "Salary"]
# creating the dataframe df
df = spark.createDataFrame(data=sampleData, schema=columns)
# importing Window from pyspark.sql.window
# creating a window
# partition of dataframe
windowPartition = Window.partitionBy("Department").orderBy("Age")
# print schema
df.printSchema()
# show df
```

df.show()

Output:

```
root
 |-- Employee_Name: string (nullable = true)
 |-- Age: long (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)

+-------------+---+----------+------+
|Employee_Name|Age|Department|Salary|
+-------------+---+----------+------+
|          Ram| 28|     Sales|  3000|
|        Meena| 33|     Sales|  4600|
|        Robin| 40|     Sales|  4100|
|        Kunal| 25|   Finance|  3000|
|          Ram| 28|     Sales|  3000|
|      Srishti| 46|Management|  3300|
|         Jeny| 26|   Finance|  3900|
|       Hitesh| 30| Marketing|  3000|
|      Kailash| 29| Marketing|  2000|
|       Sharad| 39|     Sales|  4100|
+-------------+---+----------+------+
```

This is the DataFrame on which we will apply all the analytical functions.

Example 1: Using cume_dist()

cume_dist() window function is used to get the cumulative distribution within a window partition. It is similar to CUME_DIST in SQL. Let's see an example:

```
# importing cume_dist()

# from pyspark.sql.functions

from pyspark.sql.functions import cume_dist


# applying window function with

# the help of DataFrame.withColumn

df.withColumn("cume_dist", cume_dist().over(windowPartition)).show()
```

Output:

Employee_Name	Age	Department	Salary	cume_dist
Ram	28	Sales	3000	0.4
Ram	28	Sales	3000	0.4
Meena	33	Sales	4600	0.6
Sharad	39	Sales	4100	0.8
Robin	40	Sales	4100	1.0
Srishti	46	Management	3300	1.0
Kunal	25	Finance	3000	0.5
Jeny	26	Finance	3900	1.0
Kailash	29	Marketing	2000	0.5
Hitesh	30	Marketing	3000	1.0

Example 2: Using lag()

A lag() function is used to access previous rows' data as per the defined offset value in the function. This function is similar to the LAG in SQL.

```
# importing lag() from pyspark.sql.functions

from pyspark.sql.functions import lag

df.withColumn("Lag", lag("Salary", 2).over(windowPartition)) \ .show()
```

Output:

Employee_Name	Age	Department	Salary	Lag
Ram	28	Sales	3000	null
Ram	28	Sales	3000	null
Meena	33	Sales	4600	3000
Sharad	39	Sales	4100	3000
Robin	40	Sales	4100	4600
Srishti	46	Management	3300	null
Kunal	25	Finance	3000	null
Jeny	26	Finance	3900	null
Kailash	29	Marketing	2000	null
Hitesh	30	Marketing	3000	null

Example 3: Using lead()

A lead() function is used to access next rows data as per the defined offset value in the function. This function is similar to the LEAD in SQL and just opposite to lag() function or LAG in SQL.

```
# importing lead() from pyspark.sql.functions
```

```
from pyspark.sql.functions import lead

df.withColumn("Lead", lead("salary", 2).over(windowPartition)) \ .show()
```

Output:

```
+-------------+---+----------+------+----+
|Employee_Name|Age|Department|Salary|Lead|
+-------------+---+----------+------+----+
|          Ram| 28|     Sales|  3000|4600|
|          Ram| 28|     Sales|  3000|4100|
|        Meena| 33|     Sales|  4600|4100|
|       Sharad| 39|     Sales|  4100|null|
|        Robin| 40|     Sales|  4100|null|
|      Srishti| 46|Management|  3300|null|
|        Kunal| 25|   Finance|  3000|null|
|         Jeny| 26|   Finance|  3900|null|
|      Kailash| 29| Marketing|  2000|null|
|       Hitesh| 30| Marketing|  3000|null|
+-------------+---+----------+------+----+
```

2. Ranking Function –

The function returns the statistical rank of a given value for each row in a partition or group. The goal of this function is to provide consecutive numbering of the rows in the resultant column, set by the order selected in the Window.partition for each partition specified in the OVER clause. **E.g.** row_number(), rank(), dense_rank(), etc.

Creating Dataframe for demonstration:

Before we start with these functions, first we need to create a DataFrame. We will create a DataFrame that contains student details like Roll_No, Student_Name, Subject, Marks. After creating the DataFrame we will apply each Ranking function on this DataFrame df2.

```
# importing pyspark

from pyspark.sql.window import Window

import pyspark

# importing sparksession

from pyspark.sql import SparkSession

# creating a sparksession object and providing appName

spark = SparkSession.builder.appName("pyspark_window").getOrCreate()
```

```
# sample data for dataframe
sampleData = ((101, "Ram", "Biology", 80),
                (103, "Meena", "Social Science", 78),
                (104, "Robin", "Sanskrit", 58),
                (102, "Kunal", "Phisycs", 89),
                (101, "Ram", "Biology", 80),
                (106, "Srishti", "Maths", 70),
                (108, "Jeny", "Physics", 75),
                (107, "Hitesh", "Maths", 88),
                (109, "Kailash", "Maths", 90),
                (105, "Sharad", "Social Science", 84)
                )
# column names for dataframe
columns = ["Roll_No", "Student_Name", "Subject", "Marks"]
# creating the dataframe df
df2 = spark.createDataFrame(data=sampleData, schema=columns)
# importing window from pyspark.sql.window
# creating a window partition of dataframe
windowPartition = Window.partitionBy("Subject").orderBy("Marks")
# print schema
df2.printSchema()
# show df
df2.show()
```

Output:

```
root
 |-- Roll_No: long (nullable = true)
 |-- Student_Name: string (nullable = true)
 |-- Subject: string (nullable = true)
 |-- Marks: long (nullable = true)
```

```
+-------+------------+--------------+-----+
|Roll_No|Student_Name|       Subject|Marks|
+-------+------------+--------------+-----+
|    101|         Ram|       Biology|   80|
|    103|       Meena|Social Science|   78|
|    104|       Robin|      Sanskrit|   58|
|    102|       Kunal|       Physics|   89|
|    101|         Ram|       Biology|   80|
|    106|     Srishti|         Maths|   70|
|    108|        Jeny|       Physics|   75|
|    107|      Hitesh|         Maths|   88|
|    109|     Kailash|         Maths|   90|
|    105|      Sharad|Social Science|   84|
+-------+------------+--------------+-----+
```

This is the DataFrame df2 on which we will apply all the Window ranking function.

Example 1: Using row_number().

row_number() function is used to gives a sequential number to each row present in the table. Let's see the example:

importing row_number() from pyspark.sql.functions

from pyspark.sql.functions import row_number

applying the row_number() function

df2.withColumn("row_number", row_number().over(windowPartition)).show()

Output:

```
+-------+------------+--------------+-----+----------+
|Roll_No|Student_Name|       Subject|Marks|row_number|
+-------+------------+--------------+-----+----------+
|    103|       Meena|Social Science|   78|         1|
|    105|      Sharad|Social Science|   84|         2|
|    104|       Robin|      Sanskrit|   58|         1|
|    108|        Jeny|       Physics|   75|         1|
|    102|       Kunal|       Physics|   89|         2|
|    106|     Srishti|         Maths|   70|         1|
|    107|      Hitesh|         Maths|   88|         2|
|    109|     Kailash|         Maths|   90|         3|
|    101|         Ram|       Biology|   80|         1|
|    101|         Ram|       Biology|   80|         2|
+-------+------------+--------------+-----+----------+
```

Example 2: Using rank()

The rank function is used to give ranks to rows specified in the window partition. This function leaves gaps in rank if there are ties. **Let's see the example**:

```
# importing rank() from pyspark.sql.functions

from pyspark.sql.functions import rank

# applying the rank() function

df2.withColumn("rank", rank().over(windowPartition)) \ .show()
```

Output:

```
+-------+------------+--------------+-----+----+
|Roll_No|Student_Name|       Subject|Marks|rank|
+-------+------------+--------------+-----+----+
|    103|       Meena|Social Science|   78|   1|
|    105|      Sharad|Social Science|   84|   2|
|    104|       Robin|      Sanskrit|   58|   1|
|    108|        Jeny|       Physics|   75|   1|
|    102|       Kunal|       Physics|   89|   2|
|    106|     Srishti|         Maths|   70|   1|
|    107|      Hitesh|         Maths|   88|   2|
|    109|     Kailash|         Maths|   90|   3|
|    101|         Ram|       Biology|   80|   1|
|    101|         Ram|       Biology|   80|   1|
+-------+------------+--------------+-----+----+
```

Example 3: Using percent_rank()

This function is similar to rank() function. It also provides rank to rows but in a percentile format. **Let's see the example**:

```
# importing percent_rank() from pyspark.sql.functions

from pyspark.sql.functions import percent_rank

# applying the percent_rank() function

df2.withColumn("percent_rank", percent_rank().over(windowPartition)).show()
```

Roll_No	Student_Name	Subject	Marks	percent_rank
103	Meena	Social Science	78	0.0
105	Sharad	Social Science	84	1.0
104	Robin	Sanskrit	58	0.0
108	Jeny	Physics	75	0.0
102	Kunal	Physics	89	1.0
106	Srishti	Maths	70	0.0
107	Hitesh	Maths	88	0.5
109	Kailash	Maths	90	1.0
101	Ram	Biology	80	0.0
101	Ram	Biology	80	0.0

Example 4: Using dense_rank()

This function is used to get the rank of each row in the form of row numbers. This is similar to rank() function, there is only one difference the rank function leaves gaps in rank when there are ties. **Let's see the example**:

importing dense_rank() from pyspark.sql.functions

from pyspark.sql.functions import dense_rank

applying the dense_rank() function

df2.withColumn("dense_rank", dense_rank().over(windowPartition)).show()

Output:

Roll_No	Student_Name	Subject	Marks	dense_rank
103	Meena	Social Science	78	1
105	Sharad	Social Science	84	2
104	Robin	Sanskrit	58	1
108	Jeny	Physics	75	1
102	Kunal	Physics	89	2
106	Srishti	Maths	70	1
107	Hitesh	Maths	88	2
109	Kailash	Maths	90	3
101	Ram	Biology	80	1
101	Ram	Biology	80	1

3. Aggregate function –

An aggregate function or aggregation function is a function where the values of multiple rows are grouped to form a single summary value. The definition of the groups of rows on which they operate is done by using the SQL GROUP BY clause. **E.g**. AVERAGE, SUM, MIN, MAX, etc.

Creating Dataframe for demonstration:

Before we start with these functions, we will create a new DataFrame that contains employee details like Employee_Name, Department, and Salary. After creating the DataFrame we will apply each Aggregate function on this DataFrame.

```
# importing pyspark

import pyspark

# importing sparksessio from pyspark.sql import SparkSession

# creating a sparksession

# object and providing appName

spark = SparkSession.builder.appName("pyspark_window").getOrCreate()

# sample data for dataframe

sampleData = (("Ram", "Sales", 3000),

                ("Meena", "Sales", 4600),

                ("Robin", "Sales", 4100),

                ("Kunal", "Finance", 3000),

                ("Ram", "Sales", 3000),

                ("Srishti", "Management", 3300),

                ("Jeny", "Finance", 3900),

                ("Hitesh", "Marketing", 3000),

                ("Kailash", "Marketing", 2000),

                ("Sharad", "Sales", 4100)

                )

# column names for dataframe

columns = ["Employee_Name", "Department", "Salary"]

# creating the dataframe df

df3 = spark.createDataFrame(data=sampleData, schema=columns)
```

```
# print schema

df3.printSchema()

# show df

df3.show()
```

Output:

```
root
 |-- Employee_Name: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)

+-------------+----------+------+
|Employee_Name|Department|Salary|
+-------------+----------+------+
|          Ram|     Sales|  3000|
|        Meena|     Sales|  4600|
|        Robin|     Sales|  4100|
|        Kunal|   Finance|  3000|
|          Ram|     Sales|  3000|
|      Srishti|Management|  3300|
|         Jeny|   Finance|  3900|
|       Hitesh| Marketing|  3000|
|      Kailash| Marketing|  2000|
|       Sharad|     Sales|  4100|
+-------------+----------+------+
```

This is the DataFrame df3 on which we will apply all the aggregate functions.

Example: Let's see how to apply the aggregate functions with this example

```
# importing window from pyspark.sql.window

from pyspark.sql.window import Window

# importing aggregate functions

# from pyspark.sql.functions

from pyspark.sql.functions import col,avg,sum,min,max,row_number

# creating a window partition of dataframe

windowPartitionAgg = Window.partitionBy("Department")

# applying window aggregate function

# to df3 with the help of withColumn
```

```
# this is average()

df3.withColumn("Avg", avg(col("salary")).over(windowPartitionAgg))

        #this is sum()

.withColumn("Sum", sum(col("salary")).over(windowPartitionAgg))

        #this is min()

.withColumn("Min", min(col("salary")).over(windowPartitionAgg))

        #this is max()

.withColumn("Max", max(col("salary")).over(windowPartitionAgg)).show()
```

Output:

```
+-------------+----------+------+------+-----+----+----+
|Employee_Name|Department|Salary|   Avg|  Sum| Min| Max|
+-------------+----------+------+------+-----+----+----+
|          Ram|     Sales|  3000|3760.0|18800|3000|4600|
|        Meena|     Sales|  4600|3760.0|18800|3000|4600|
|        Robin|     Sales|  4100|3760.0|18800|3000|4600|
|          Ram|     Sales|  3000|3760.0|18800|3000|4600|
|       Sharad|     Sales|  4100|3760.0|18800|3000|4600|
|      Srishti|Management|  3300|3300.0| 3300|3300|3300|
|        Kunal|   Finance|  3000|3450.0| 6900|3000|3900|
|         Jeny|   Finance|  3900|3450.0| 6900|3000|3900|
|       Hitesh| Marketing|  3000|2500.0| 5000|2000|3000|
|      Kailash| Marketing|  2000|2500.0| 5000|2000|3000|
+-------------+----------+------+------+-----+----+----+
```

4.15 Broadcast Variables & Accumulators -

For parallel processing, Apache Spark uses shared variables. A copy of shared variable goes on each node of the cluster when the driver sends a task to the executor on the cluster, so that it can be used for performing tasks.

Spark contains two different types of shared variables − one is broadcast variables and second is accumulators.

1. **Broadcast variables** − used to efficiently, distribute large values.
2. **Accumulators** − used to aggregate the information of particular collection.

1. Broadcast variables: Broadcast variables are used to save the copy of data across all nodes. This variable is cached on all the machines and not sent on machines with tasks. The following code block has the details of a Broadcast class for PySpark.

```
class pyspark.Broadcast (
   sc = None,
   value = None,
   pickle_registry = None,
   path = None
)
```

The following example shows how to use a Broadcast variable. A Broadcast variable has an attribute called value, which stores the data and is used to return a broadcasted value.

```
----------------------------------------broadcast.py-------------------------
from pyspark import SparkContext
sc = SparkContext("local", "Broadcast app")
words_new = sc.broadcast(["scala", "java", "hadoop", "spark", "akka"])
data = words_new.value
print "Stored data -> %s" % (data)
elem = words_new.value[2]
print "Printing a particular element in RDD -> %s" % (elem)
----------------------------------------broadcast.py-------------------------
```

Command − The command for a broadcast variable is as follows −

$SPARK_HOME/bin/spark-submit broadcast.py

Output − The output for the following command is given below.

```
Stored data -> [
   'scala',
   'java',
   'hadoop',
   'spark',
   'akka'
]
Printing a particular element in RDD -> hadoop
```

2. Accumulator

Accumulator variables are used for aggregating the information through associative and commutative operations. For example, you can use an accumulator for a sum operation or counters (in MapReduce). The following code block has the details of an Accumulator class for PySpark.

```
class pyspark.Accumulator(aid, value, accum_param)
```

The following example shows how to use an Accumulator variable. An Accumulator variable has an attribute called value that is similar to what a broadcast variable has. It stores the data and is used to return the accumulator's value, but usable only in a driver program.

In this example, an accumulator variable is used by multiple workers and returns an accumulated value.

```
----------------------------------------accumulator.py----------------------
from pyspark import SparkContext
sc = SparkContext("local", "Accumulator app")
num = sc.accumulator(10)
def f(x):
   global num
   num+=x
rdd = sc.parallelize([20,30,40,50])
rdd.foreach(f)
final = num.value
print "Accumulated value is -> %i" % (final)
----------------------------------------accumulator.py----------------------
```

Command − The command for an accumulator variable is as follows –

$SPARK_HOME/bin/spark-submit accumulator.py

Output − The output for the above command is given below.

Accumulated value is -> 150

4.16 Deploying Spark -

There are two high-level options for where to deploy Spark clusters: **deploy in an on-premises** cluster or **in the public cloud.**

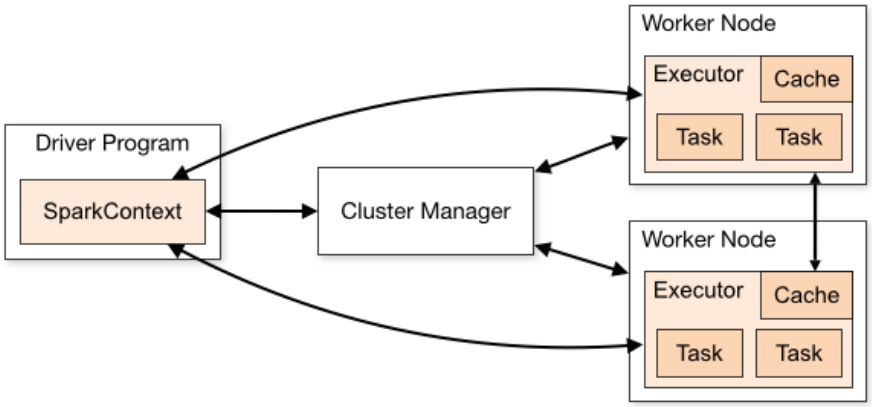
4.16.1 Cluster manager modes

Spark applications run as independent sets of processes on a cluster. These clusters are coordinated by the SparkContext object in the driver (main) program.

To run on a cluster:

- SparkContext must connect to multiple cluster managers that allocate resources across the application.
- Once connected, Spark acquires executors on nodes in the clusters. These are processes that run and store the computations.

- Spark then sends application code to the executors, and SparkContext sends tasks to the executors to run.



Spark has three officially supported cluster managers:

1. Standalone mode
2. Hadoop YARN
3. Apache Mesos

These cluster managers maintain a set of machines onto which you can deploy Spark applications.

1. Standalone Mode

Spark's standalone cluster manager is a lightweight platform built specifically for Apache Spark workloads. Using it, you can run multiple Spark Applications on the same cluster. It also provides simple interfaces for doing so but can scale to large Spark workloads. The main disadvantage of the standalone mode is that it's more limited than the other cluster managers—in Particular, your cluster can only run Spark.

a) Starting a standalone cluster

The first step is to start the master process on the machine that we want that to run on, using the following command:

```
$SPARK_HOME/sbin/start-master.sh
```

When we run this command, the cluster manager master process will start up on that machine. Once started, the master prints out a spark: //HOST:PORT URI. You use this when you start each of the worker nodes of the cluster, and you can use it as the master argument to your SparkSession on application initialization. You can also find this URI on the master's web UI, which is http://master-ip-address:8080 by default.

The master machine must be available on the network of the worker nodes you are using, and the port must be open on the master node, as well:

```
$SPARK_HOME/sbin/start-slave.sh <master-spark-URI>
```

This process is naturally a bit manual; thankfully there are scripts that can help to automate this process.

b) Cluster launch scripts

You can configure cluster launch scripts that can automate the launch of standalone clusters. To do this, create a file called conf/slaves in your Spark directory that will contain the hostnames of all the machines on which you intend to start Spark workers, one per line.

After you set up this file, you can launch or stop your cluster by using the following shell scripts, based on Hadoop's deploy scripts, and available in $SPARK_HOME/sbin:

$SPARK_HOME/sbin/start-master.sh

- Starts a master instance on the machine on which the script is executed.

$SPARK_HOME/sbin/start-slaves.sh

- Starts a slave instance on each machine specified in the conf/slaves file.

$SPARK_HOME/sbin/start-slave.sh

- Starts a slave instance on the machine on which the script is executed.

$SPARK_HOME/sbin/start-all.sh

- Starts both a master and a number of slaves as described earlier.

$SPARK_HOME/sbin/stop-master.sh

- Stops the master that was started via the bin/start-master.sh script.

$SPARK_HOME/sbin/stop-slaves.sh

- Stops all slave instances on the machines specified in the conf/slaves file.

$SPARK_HOME/sbin/stop-all.sh

- Stops both the master and the slaves as described earlier.

c) Standalone cluster configurations

Standalone clusters have a number of configurations that you can use to tune your application. These control everything from what happens to old files on each worker for terminated applications to the worker's core and memory resources. These are controlled via environment variables or via application properties.

d) Submitting applications

After you create the cluster, you can submit applications to it using the spark:// URI of the master. You can do this either on the master node itself or another machine using spark-submit.

2. Hadoop YARN

Hadoop YARN is a framework for job scheduling and cluster resource management. Even though Spark is often (mis)classified as a part of the "Hadoop Ecosystem," in reality, Spark has little to do with Hadoop. Spark does natively support the Hadoop YARN cluster manager but it requires nothing from Hadoop itself.

You can run your Spark jobs on Hadoop YARN by specifying the master as YARN in the spark-submit command-line arguments. Hadoop YARN is a generic scheduler for a large number of different execution frameworks.

3. Apache Mesos

Apache Mesos is another clustering system that Spark can run on. Apache Mesos abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively.

Mesos intends to be a datacenter scale-cluster manager that manages not just short-lived applications like Spark, but long-running applications like web applications or other resource interfaces. Mesos is the heaviest-weight cluster manager, simply because you might choose this cluster manager only if your organization already has a large-scale deployment of Mesos, but it makes for a good cluster manager nonetheless.

4.17 Spark Logs

When a Spark job or application fails, you can use the Spark logs to analyze the failures.

The QDS UI provides links to the logs in the Application UI and Spark Application UI.

- If you are running the Spark job or application from the Analyze page, you can access the logs via the Application UI and Spark Application UI.
- If you are running the Spark job or application from the Notebooks page, you can access the logs via the Spark Application UI.

4.17.1 The Spark UI

The Spark UI provides a visual way to monitor applications while they are running as well as metrics about your Spark workload, at the Spark and JVM level. Every SparkContext running launches a web UI, by default on port 4040, that displays useful information about the application. When you run Spark in local mode for example, just navigate to http://localhost:4040 to see the UI when running a Spark Application on your local machine. If you're running multiple applications, they will launch web UIs on increasing port numbers (4041, 4042, …). Cluster managers will also link to each application's web UI from their own UI.

Figure: shows all of the tabs available in the Spark UI.

These tabs are accessible for each of the things that we'd like to monitor. For the most part, each of these should be self-explanatory:

- The Jobs tab refers to Spark jobs.
- The Stages tab pertains to individual stages (and their relevant tasks).
- The Storage tab includes information and the data that is currently cached in our Spark Application.
- The Environment tab contains relevant information about the configurations and current settings of the Spark application.
- The SQL tab refers to our Structured API queries (including SQL and DataFrames).
- The Executors tab provides detailed information about each executor running our application.

Example of how you can drill down into a given query. Open a new Spark shell, run the following code, and we will trace its execution through the Spark UI:

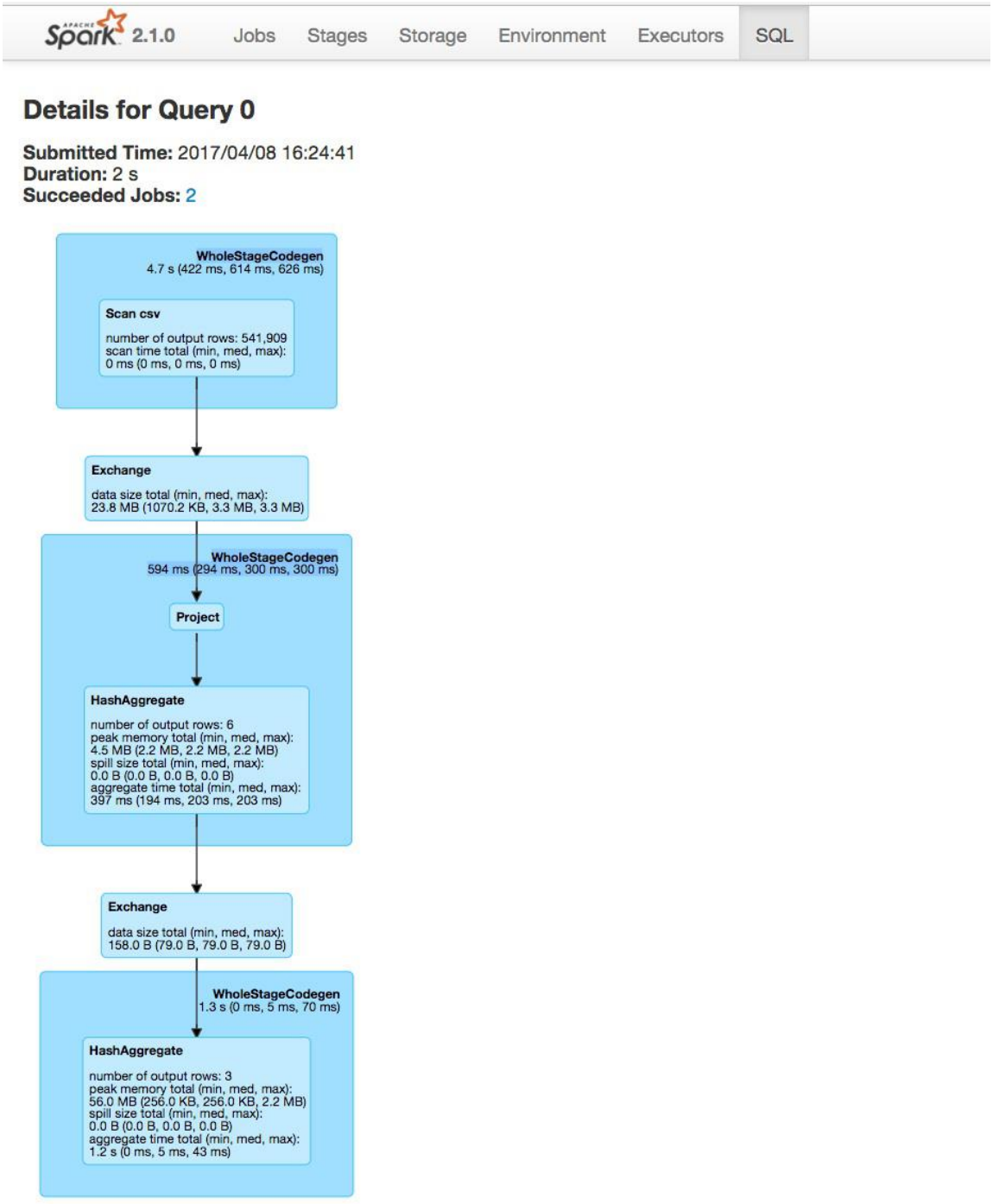
```
# in Python

spark.read\

.option("header", "true")\

.csv("/data/retail-data/all/online-retail-dataset.csv")\

.repartition(2)\

.selectExpr("instr(Description, 'GLASS') >= 1 as is_glass")\
```

.groupBy("is_glass")\

.count()\

.collect()

This results in three rows of various values. The code kicks off a SQL query, so let's navigate to the SQL tab.



The first thing you see is aggregate statistics about this query:

Submitted Time: 2017/04/08 16:24:41

Duration: 2 s

Succeeded Jobs: 2

These will become important in a minute, but first let's take a look at the Directed Acyclic Graph (DAG) of Spark stages. Each blue box in these tabs represents a stage of Spark tasks. The entire group of these stages represents our Spark job.

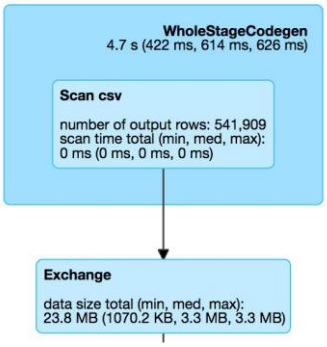


Figure: Stage one

The box on top, labeled WholeStateCodegen, represents a full scan of the CSV file. The box below that represents a shuffle that we forced when we called repartition. This turned our original dataset (of a yet to be specified number of partitions) into two partitions.

The next step is our projection (selecting/adding/filtering columns) and the aggregation.

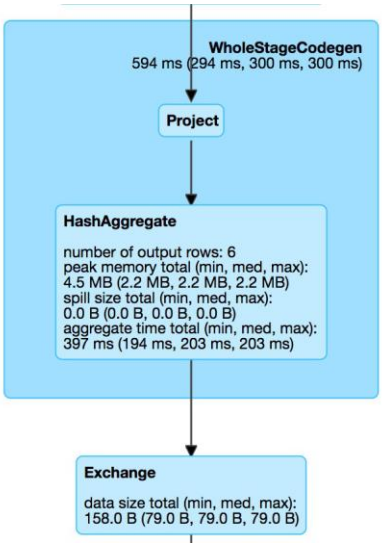


Figure: Stage two

The last stage is the aggregation of the subaggregations that we saw happen on a per-partition basis in the previous stage. We combine those two partitions in the final three rows that are the output of our total query.

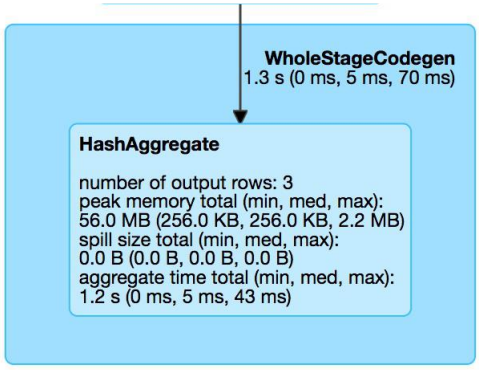


Figure: Stage three

Let's look further into the job's execution. On the Jobs tab, next to Succeeded Jobs, click 2. As in the Figure demonstrates, our job breaks down into three stages (which corresponds to what we saw on the SQL tab).

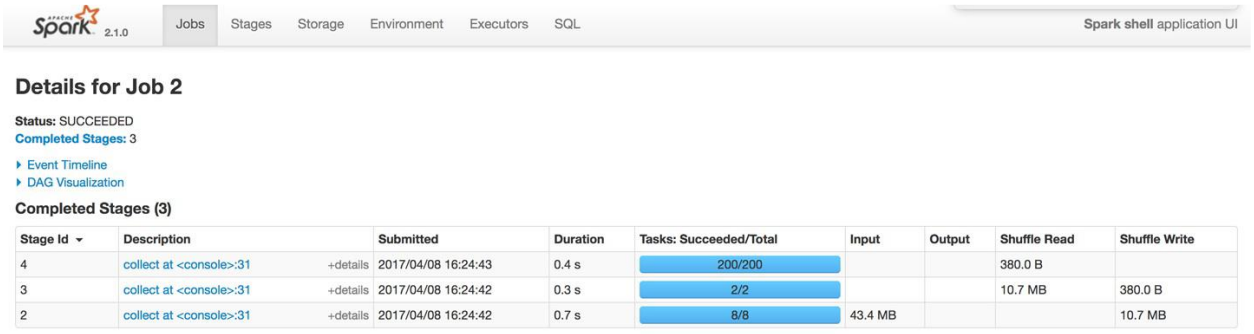
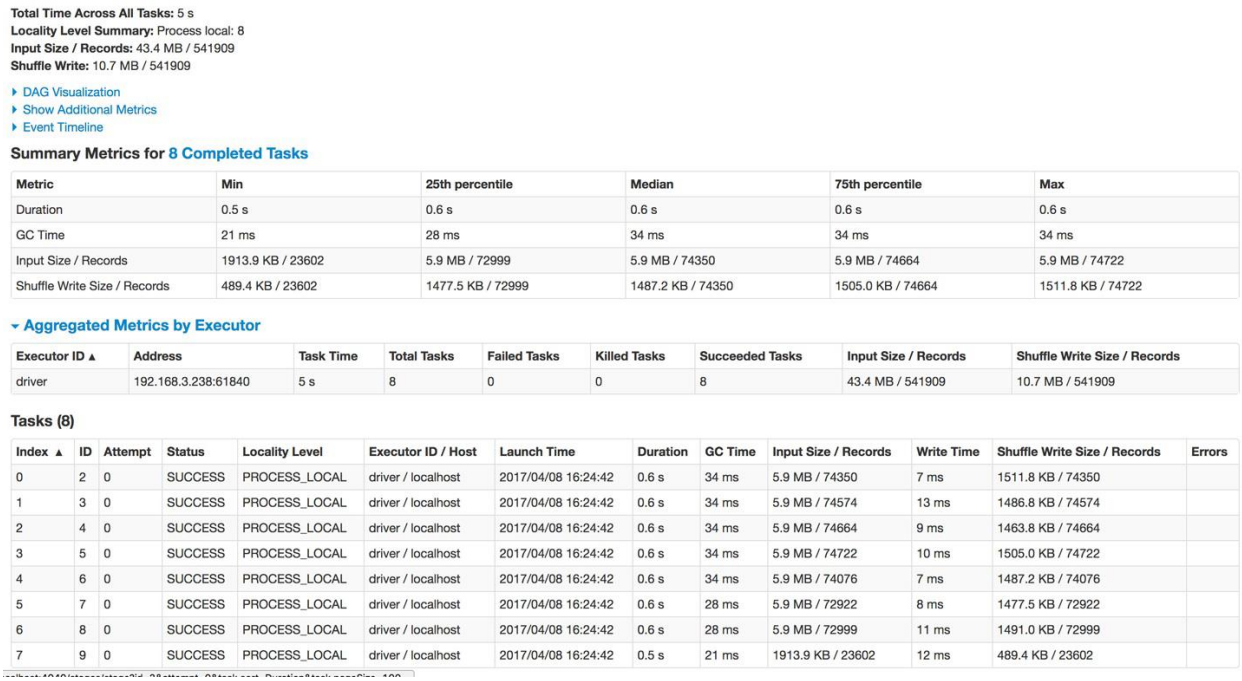


Figure: The jobs tab

The first stage has eight tasks. CSV files are splittable, and Spark broke up the work to be distributed relatively evenly between the different cores on the machine. This happens at the cluster level and points to an important optimization: how you store your files. The following stage has two tasks because we explicitly called a repartition to move the data into two partitions. The last stage has 200 tasks because the default shuffle partitions value is 200.

Now that we reviewed how we got here, click the stage with eight tasks to see the next level of detail, as shown in Figure 18-8.



Total Time Across All Tasks: 5 s
Locality Level Summary: Process local: 8
Input Size / Records: 43.4 MB / 541909
Shuffle Write: 10.7 MB / 541909

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.6 s	0.6 s	0.6 s	0.6 s
GC Time	21 ms	28 ms	34 ms	34 ms	34 ms
Input Size / Records	1913.9 KB / 23602	5.9 MB / 72999	5.9 MB / 74350	5.9 MB / 74664	5.9 MB / 74722
Shuffle Write Size / Records	489.4 KB / 23602	1477.5 KB / 72999	1487.2 KB / 74350	1505.0 KB / 74664	1511.8 KB / 74722

▾ Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
driver	192.168.3.238:61840	5 s	8	0	0	8	43.4 MB / 541909	10.7 MB / 541909

Tasks (8)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74350	7 ms	1511.8 KB / 74350	
1	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74574	13 ms	1486.8 KB / 74574	
2	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74664	9 ms	1463.8 KB / 74664	
3	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74722	10 ms	1505.0 KB / 74722	
4	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74076	7 ms	1487.2 KB / 74076	
5	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	28 ms	5.9 MB / 72922	8 ms	1477.5 KB / 72922	
6	8	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	28 ms	5.9 MB / 72999	11 ms	1491.0 KB / 72999	
7	9	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.5 s	21 ms	1913.9 KB / 23602	12 ms	489.4 KB / 23602	

Figure: Spark tasks

4.17.2 Spark UI History Server

Normally, the Spark UI is only available while a SparkContext is running, so how can you get to it after your application crashes or ends? To do this, Spark includes a tool called the Spark History Server that allows you to reconstruct the Spark UI and REST API, provided that the application was configured to save an event log. You can find up-to-date information about how to use this tool in the Spark documentation.

To use the history server, you first need to configure your application to store event logs to a certain location. You can do this by by enabling spark.eventLog.enabled and the event log location with the configuration spark.eventLog.dir. Then, once you have stored the events, you can run the history server as a standalone application, and it will automatically reconstruct the web UI based on these logs. Some cluster managers and cloud services also configure logging automatically and run a history server by default.

4.18 Debugging and Spark First Aid

There are many issues that may affect Spark jobs, some of the more common Spark issues you may encounter:

1. Slow Aggregations

If you have a slow aggregation, start by reviewing the issues in the “Slow Tasks” section before proceeding. Having tried those, you might continue to see the same problem.

Signs and symptoms

- Slow tasks during a groupBy call.
- Jobs after the aggregation are slow, as well.

Potential treatments

- Increasing the number of partitions, prior to an aggregation, might help by reducing the number of different keys processed in each task.
- Increasing executor memory can help alleviate this issue, as well. If a single key has lots of data, this will allow its executor to spill to disk less often and finish faster, although it may still be much slower than executors processing other keys.
- If you find that tasks after the aggregation are also slow, this means that your dataset might have remained unbalanced after the aggregation. Try inserting a repartition call to partition it randomly.
- Ensuring that all filters and SELECT statements that can be are above the aggregation can help to ensure that you're working only on the data that you need to be working on and nothing else. Spark's query optimizer will automatically do this for the structured APIs.
- Ensure null values are represented correctly (using Spark's concept of null) and not as some default value like " " or "EMPTY". Spark often optimizes for skipping nulls early in the job when possible, but it can't do so for your own placeholder values.
- Some aggregation functions are also just inherently slower than others. For instance, collect_list and collect_set are very slow aggregation functions because they must return all the matching objects to the driver, and should be avoided in performance-critical code.

2. Slow Joins

Joins and aggregations are both shuffles, so they share some of the same general symptoms as well as treatments.

Signs and symptoms

- A join stage seems to be taking a long time. This can be one task or many tasks.
- Stages before and after the join seem to be operating normally.

Potential treatments

- Many joins can be optimized (manually or automatically) to other types of joins.
- Experimenting with different join orderings can really help speed up jobs, especially if some of those joins filter out a large amount of data; do those first.
- Partitioning a dataset prior to joining can be very helpful for reducing data movement across the cluster, especially if the same dataset will be used in multiple join operations.

It's worth experimenting with different prejoin partitioning. Keep in mind, again, that this isn't "free" and does come at the cost of a shuffle.

- Slow joins can also be caused by data skew. There's not always a lot you can do here, but sizing up the Spark application and/or increasing the size of executors can help, as described in earlier sections.
- Ensuring that all filters and select statements that can be are above the join can help to ensure that you're working only on the data that you need for the join.
- Ensure that null values are handled correctly (that you're using null) and not some default value like " " or "EMPTY", as with aggregations.
- Sometimes Spark can't properly plan for a broadcast join if it doesn't know any statistics about the input DataFrame or table.

3. Slow Reads and Writes

Slow I/O can be difficult to diagnose, especially with networked file systems.

Signs and symptoms

- Slow reading of data from a distributed file system or external system.
- Slow writes from network file systems or blob storage.

Potential treatments

- Turning on speculation (set spark.speculation to true) can help with slow reads and writes. This will launch additional tasks with the same operation in an attempt to see whether it's just some transient issue in the first task. Speculation is a powerful tool and works well with consistent file systems. However, it can cause duplicate data writes with some eventually consistent cloud services, such as Amazon S3, so check whether it is supported by the storage system connector you are using.
- Ensuring sufficient network connectivity can be important—your Spark cluster may simply not have enough total network bandwidth to get to your storage system.
- For distributed file systems such as HDFS running on the same nodes as Spark, make sure Spark sees the same hostnames for nodes as the file system. This will enable Spark to do locality-aware scheduling, which you will be able to see in the "locality" column in the Spark UI.