Link -
https://sparkbyexamples.com/spark/what-is-spark-executor/

PySpark

What are the Features of PySpark?

- In-memory computation
- Distributed processing using parallelize
- Can be used with many cluster managers (Spark, Yarn, Mesos e.t.c)
- Fault-tolerant
- Immutable
- Lazy evaluation
- Cache & persistence
- Inbuild-optimization when using DataFrames
- Supports ANSI SQL

Advantages of PySpark

- PySpark is a general-purpose, in-memory, distributed processing engine that allows you to process data efficiently in a distributed fashion.
- Applications running on PySpark are 100x faster than traditional systems.
- You will get great benefits from using PySpark for data ingestion pipelines.
- Using PySpark we can process data from Hadoop HDFS, AWS S3, and many file systems.
- PySpark also is used to process real-time data using Streaming and Kafka.
- Using PySpark streaming you can also stream files from the file system and also stream from the socket.
- PySpark natively has machine learning and graph libraries.

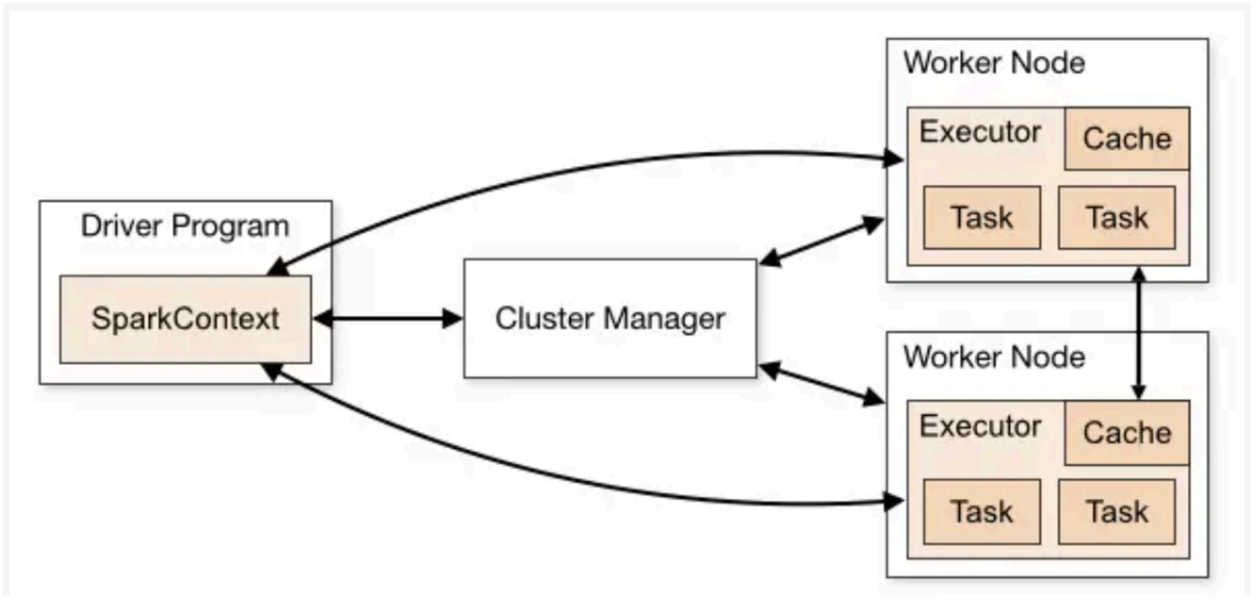
What Version of Python PySpark Supports

PySpark 3.5 is compatible with Python 3.8 and newer, as well as R 3.5, Java versions 8, 11, and 17, and Scala versions 2.12 and 2.13, beyond.

PySpark Architecture

Apache Spark works in a **master-slave architecture** where the master is called the "**Driver**" and slaves are called "**Workers**".

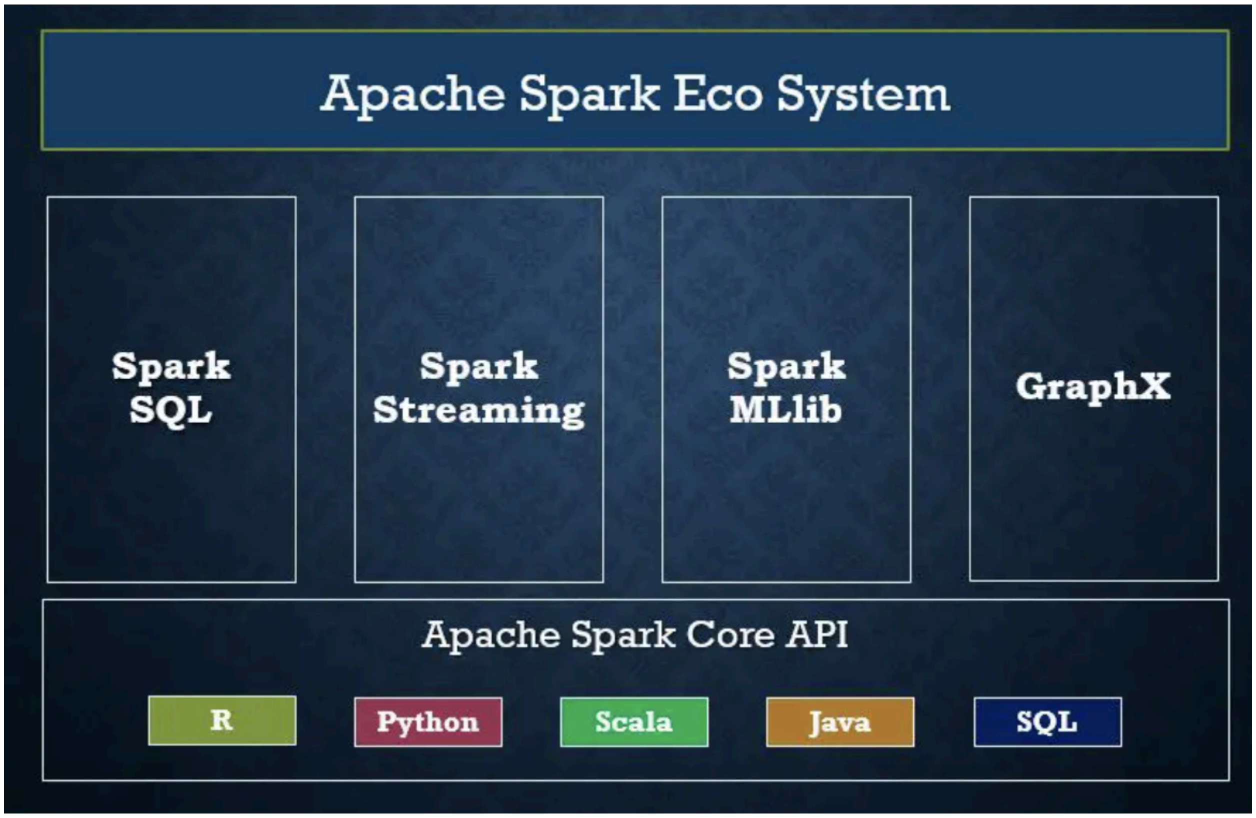
When you **run a Spark application,** Spark Driver creates a **context** that is an **entry point** to your application, and all operations (transformations and actions) are **executed** on **worker** nodes, and the resources are managed by **Cluster Manager.**



Cluster Manager Types

As of writing this Spark with Python (PySpark) tutorial for beginners, Spark supports below cluster managers:

- **Standalone** – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- **Apache Mesos** – Mesons is a Cluster manager that can also run Hadoop MapReduce and PySpark applications.
- **Hadoop YARN** – the resource manager in Hadoop 2. This is mostly used as a cluster manager.
- **Kubernetes** – an open-source system for automating deployment, scaling, and management of containerized applications.



What is RDD (Resilient Distributed Dataset)?

RDD (Resilient Distributed Dataset) is a fundamental building block of PySpark which is fault-tolerant , immutable distributed collections of objects. Immutable meaning once you create an RDD you cannot change it. Each record in RDD is divided into logical partitions, which can be computed on different nodes of the cluster

This Apache PySpark RDD tutorial describes the basic operations available on RDDs, such as **map()**, **filter()**, **and** **persist()** and many more. In addition, this tutorial also explains Pair RDD functions that operate on RDDs of key-value pairs such as **groupByKey()** **and** **join()** etc.

Note: RDD's can have a name and unique identifier (id)

PySpark RDD Benefits

1. **In-Memory Computation**
2. **Immutability** - PySpark RDD's are immutable in nature meaning, once RDDs are created you cannot modify. When we apply transformations on RDD, PySpark creates a new RDD and maintains the RDD Lineage.
3. **Fault Tolerance** - PySpark operates on fault-tolerant data stores on HDFS, S3 e.t.c hence any RDD operation fails, it automatically reloads the data from other partitions
4. **Lazy Evolution** - PySpark does not evaluate the RDD transformations as they appear/encountered by Driver instead it keeps the all transformations as it encounters(DAG) and evaluates the all transformation when it sees the first RDD action.
5. **Partitioning -** When you create RDD from a data, It by default partitions the elements in a RDD. By default it partitions to the number of cores available.

PySpark RDD Limitations

1. PySpark RDDs are not much suitable for applications that make updates to the state store such as storage systems for a web application(Database)

Creating RDD

1. parallelizing an existing collection and
2. referencing a dataset in an external storage system (HDFS, S3 and many more).

```
from pyspark.sql import SparkSession
spark:SparkSession = SparkSession.builder()
    .master("local[1]")
    .appName("SparkByExamples.com")
    .getOrCreate()
```

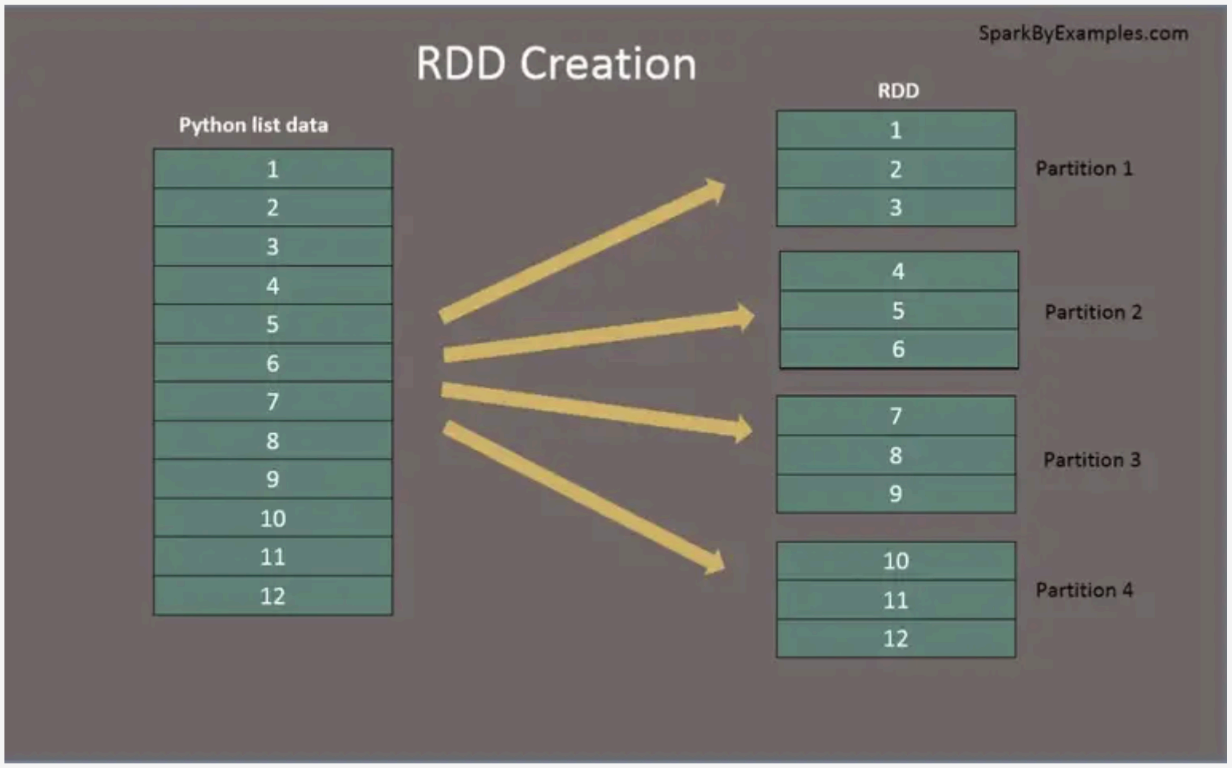
master() – If you are running it on the cluster you need to use your master name as an argument to master(). usually, it would be either yarn (Yet Another Resource Negotiator) or mesos depends on your cluster setup.

Use local[x] when running in Standalone mode. x should be an integer value and should be greater than 0; this represents how many partitions it should create when using RDD,
DataFrame, and Dataset. Ideally, x value should be the number of CPU cores you have.

appName() – Used to set your application name.

getOrCreate() – This returns a SparkSession object if already exists, and creates a new one if not exist.

Create RDD using sparkContext.parallelize()



RDD from list

#Create RDD from parallelize
data = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd=spark.sparkContext.parallelize(data)

Create RDD using sparkContext.textFile()
#Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/textFile.txt")

Create empty RDD using sparkContext.emptyRDD
# Creates empty RDD with no partition

```
rdd = spark.sparkContext.emptyRDD
# rddString = spark.sparkContext.emptyRDD[String]
```

Creating empty RDD with partition

```
#Create empty RDD with partition
rdd2 = spark.sparkContext.parallelize([],10) #This creates 10 partitions
```

RDD Parallelize

When we use parallelize() or textFile() or wholeTextFiles() methods of SparkContxt to initiate RDD, it automatically splits the data into partitions based on resource availability. **when you run it on a laptop it would create partitions as the same number of cores available on your system.**

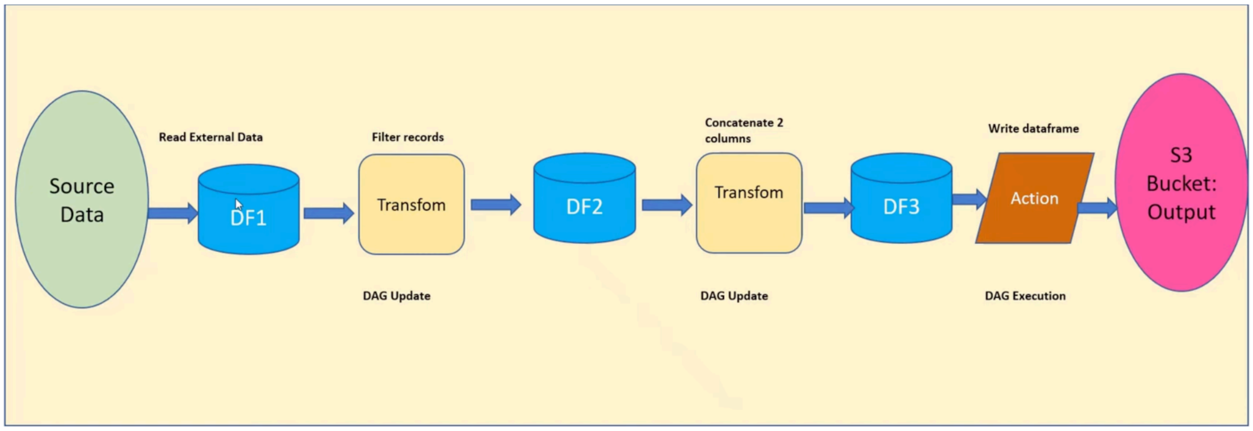
Repartition and Coalesce

- **repartition()** method which shuffles data from all nodes also called full shuffle.
- **coalesce()** method which shuffle data from minimum nodes, for example if you have data in 4 partitions and doing coalesce(2) moves data from just 2 nodes.

Note - repartition() method is a very expensive operation as it shuffles data from all nodes in a cluster.repartition() or coalesce() methods also returns a new RDD.

PySpark RDD Operations

RDD transformations – Transformations are lazy operations, instead of updating an RDD, these operations return another RDD. Only create DAG until action is called .
RDD actions – operations that trigger computation and return RDD values.



These transformations can be classified into two types based on their execution characteristics: narrow transformations and wide transformations.

1. **Narrow Transformations:**
    a. Narrow transformations are the transformations where each partition of the parent RDD contributes to only one partition of the child RDD.
    b. These transformations **do not require shuffling of data across partitions**, meaning they can be executed in parallel without data movement between partitions.
    c. In-expensive process
    d. **Examples of narrow transformations include map, filter, flatMap, mapPartitions, mapPartitionsWithIndex, etc.**

2. **Wide Transformations:**
    a. Wide transformations are the transformations where each partition of the parent RDD can contribute to multiple partitions of the child RDD.
    b. These transformations **require shuffling of data across partitions,** which can incur significant overhead in terms of network I/O and processing.
    c. Expensive process
    d. **Examples of wide transformations include groupBy, reduceByKey, join, sortByKey, cogroup, etc**.

map: It applies a function to each element of the RDD/DataFrame and returns a new RDD/DataFrame with the results.

```
# Example of map
rdd = sc.parallelize([1, 2, 3, 4])
mapped_rdd = rdd.map(lambda x: x * 2)
# Result: [2, 4, 6, 8]
```

filter: It applies a function to each element of the RDD/DataFrame and returns a new RDD/DataFrame with the elements that satisfy the condition

```
# Example of filter
rdd = sc.parallelize([1, 2, 3, 4])
filtered_rdd = rdd.filter(lambda x: x % 2 == 0)
# Result: [2, 4]
```

flatMap: Similar to map, but it allows each input item to map to 0 or more output items.

```
# Example of flatMap
rdd = sc.parallelize(["Hello world", "This is PySpark"])
flat_mapped_rdd = rdd.flatMap(lambda x: x.split(" "))
# Result: ['Hello', 'world', 'This', 'is', 'PySpark']
```

mapPartitions: It applies a function to each partition of the RDD and returns a new RDD. The function is called once for each partition.

```
# Example of mapPartitions
rdd = sc.parallelize([1, 2, 3, 4], 2)
def add_one(iterator):
   return map(lambda x: x + 1, iterator)
mapped_partition_rdd = rdd.mapPartitions(add_one)
# Result: [[2, 3], [4, 5]]
```

mapPartitionsWithIndex: Similar to mapPartitions but provides the index of the partition as well.

```
# Example of mapPartitionsWithIndex
rdd = sc.parallelize([1, 2, 3, 4], 2)
def add_index(index, iterator):
   return map(lambda x: (index, x), iterator)
```

mapped_partition_index_rdd = rdd.mapPartitionsWithIndex(add_index)
# Result: [(0, 1), (0, 2), (1, 3), (1, 4)]

RDD Actions with example -

RDD Action operations return the values from an RDD to a driver program. In other words, any RDD function that returns non-RDD is considered as an action.

In this section of the PySpark RDD tutorial, we will continue to use our word count example and performs some actions on it

count() – Returns the number of records in an RDD

```
# Action - count
print("Count : "+str(rdd6.count()))
```

first() – Returns the first record.

```
# Action - first
firstRec = rdd6.first()
print("First Record : "+str(firstRec[0]) + ","+ firstRec[1])
```

max() – Returns max record.

```
# Action - max
datMax = rdd6.max()
print("Max Record : "+str(datMax[0]) + ","+ datMax[1])
```

reduce() – Reduces the records to single, we can use this to count or sum.

```
# Action - reduce
totalWordCount = rdd6.reduce(lambda a,b: (a[0]+b[0],a[1]))
print("dataReduce Record : "+str(totalWordCount[0]))
```

take() – Returns the record specified as an argument.

```
# Action - take
data3 = rdd6.take(3)
for f in data3:
  print("data3 Key:"+ str(f[0]) +", Value:"+f[1])
```

collect() – Returns all data from RDD as an array. Be careful when you use this action when you are working with huge RDD with millions and billions of data as you may run out of memory on the driver.

```
# Action - collect
data = rdd6.collect()
for f in data:
   print("Key:"+ str(f[0]) +", Value:"+f[1])
```

RDD ACTION METHODS	**METHOD DEFINITION**
aggregate[U](zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): U	Aggregate the elements of each partition, and then the results for all the partitions.
collect():Array[T]	Return the complete dataset as an Array.
count():Long	Return the count of elements in the dataset.
countApprox(timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]	Return approximate count of elements in the dataset, this method returns incomplete when execution time meets timeout.
countApproxDistinct(relativeSD: Double = 0.05): Long	Return an approximate number of distinct elements in the dataset.

countByValue(): Map[T, Long]	Return Map[T,Long] key representing each unique value in dataset and value represent count each value present.
countByValueApprox(timeout: Long, confidence: Double = 0.95)(implicit ord: Ordering[T] = null): PartialResult[Map[T, BoundedDouble]]	Same as countByValue() but returns approximate result.
first():T	Return the first element in the dataset.
fold(zeroValue: T)(op: (T, T) ⇒ T): T	Aggregate the elements of each partition, and then the results for all the partitions.
foreach(f: (T) ⇒ Unit): Unit	Iterates all elements in the dataset by applying function f to all elements.
foreachPartition(f: (Iterator[T]) ⇒ Unit): Unit	Similar to foreach, but applies function f for each partition.
min()(implicit ord: Ordering[T]): T	Return the minimum value from the dataset.
max()(implicit ord: Ordering[T]): T	Return the maximum value from the dataset.

reduce(f: (T, T) ⇒ T): T	Reduces the elements of the dataset using the specified binary operator.
saveAsObjectFile(path: String): Unit	Saves RDD as a serialized object's to the storage system.
saveAsTextFile(path: String, codec: Class[_ <: CompressionCodec]): Unit	Saves RDD as a compressed text file.
saveAsTextFile(path: String): Unit	Saves RDD as a text file.
take(num: Int): Array[T]	Return the first num elements of the dataset.
takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]	Return the first num (smallest) elements from the dataset and this is the opposite of the take() action. Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
takeSample(withReplacement: Boolean, num: Int, seed: Long = Utils.random.nextLong): Array[T]	Return the subset of the dataset in an Array.

	Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
toLocalIterator(): Iterator[T]	Return the complete dataset as an Iterator. Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
top(num: Int)(implicit ord: Ordering[T]): Array[T]	Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
treeAggregate	Aggregates the elements of this RDD in a multi-level tree pattern.
treeReduce	Reduces the elements of this RDD in a multi-level tree pattern.

Shuffle Operations

Shuffling is a mechanism PySpark uses to redistribute the data across different executors and even across machines. PySpark shuffling triggers when we perform certain transformation operations like **gropByKey(), reduceByKey(), join() on RDDS**

PySpark Shuffle is an expensive operation since it involves the following

- Disk I/O
- Involves data serialization and deserialization
- Network I/O

For example, when we perform **reduceByKey()** operation, PySpark does the following

- PySpark first runs map tasks on all partitions which groups all values for a single key.
- The results of the map tasks are kept in memory.
- When results do not fit in memory, PySpark stores the data into a disk.
- PySpark shuffles the mapped data across partitions, some times it also stores the shuffled data into a disk for reuse when it needs to recalculate.
- Run the garbage collection
- Finally runs reduce tasks on each partition based on key.

PySpark RDD triggers shuffle and repartition for several operations like **repartition() and coalesce(), groupByKey(), reduceByKey(), cogroup()** and join() but not cou**ntByKey()** .

PySpark RDD Persistence

PySpark Cache and Persist are optimization techniques to improve the performance of the RDD jobs that are iterative and interactive

Though PySpark provides computation 100 x times faster than traditional Map Reduce jobs, If you have not designed the jobs to reuse the repeating computations you will see degrade in performance when you are dealing with billions or trillions of data.

Using cache() and persist() methods, PySpark provides an optimization mechanism to store the intermediate computation of an RDD so they can be reused in subsequent actions.

RDD Cache

PySpark RDD cache() method by default saves RDD computation to storage level **`MEMORY_ONLY`** meaning it will store the data in the JVM heap as unserialized objects.

PySpark **cache()** method in RDD class internally calls persist() method which in turn uses sparkSession.sharedState.cacheManager.cacheQuery to cache the result set of RDD. Let's look at an example.

```
 cachedRdd = rdd.cache()
```

RDD Persist

PySpark **persist()** method is used to store the RDD to one of the storage levels **MEMORY_ONLY,MEMORY_AND_DISK, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, DISK_ONLY, MEMORY_ONLY_2,MEMORY_AND_DISK_2 and more.**

MEMORY_ONLY – This is the default behavior of the RDD cache() method and stores the RDD as deserialized objects to JVM memory. When there is no enough memory available it will not save to RDD of some partitions and these will be re-computed as and when required. This takes more storage but runs faster as it takes few CPU cycles to read from memory.

MEMORY_ONLY_SER – This is the same as MEMORY_ONLY but the difference being it stores RDD as serialized objects to JVM memory. It takes lesser memory (space-efficient) then MEMORY_ONLY as it saves objects as serialized and takes an additional few more CPU cycles in order to deserialize.

MEMORY_ONLY_2 – Same as MEMORY_ONLY storage level but replicate each partition to two cluster nodes.

MEMORY_ONLY_SER_2 – Same as MEMORY_ONLY_SER storage level but replicate each partition to two cluster nodes.

MEMORY_AND_DISK – In this Storage Level, The RDD will be stored in JVM memory as a deserialized objects. When required storage is greater than available memory, it stores some of the excess partitions in to disk and reads the data from disk when it required. It is slower as there is I/O involved.

MEMORY_AND_DISK_SER – This is same as MEMORY_AND_DISK storage level difference being it serializes the RDD objects in memory and on disk when space not available.

MEMORY_AND_DISK_2 – Same as MEMORY_AND_DISK storage level but replicate each partition to two cluster nodes.

MEMORY_AND_DISK_SER_2 – Same as MEMORY_AND_DISK_SER storage level but replicate each partition to two cluster nodes.

DISK_ONLY – In this storage level, RDD is stored only on disk and the CPU computation time is high as I/O involved.

DISK_ONLY_2 – Same as DISK_ONLY storage level but replicate each partition to two cluster nodes.

RDD Unpersist

```
rddPersist2 = rddPersist.unpersist()
```

PySpark Shared Variables

When PySpark executes transformation using map() or reduce() operations, It executes the transformations on a remote node by using the variables that are shipped with the tasks and these variables are not sent back to PySpark Driver hence there is no capability to reuse and sharing the variables across tasks. PySpark shared variables solve this problem using the below two techniques. PySpark provides two types of shared variables.

- **Broadcast variables (read-only shared variable)**
- **Accumulator variables (updatable shared variables)**

Broadcast read-only Variables

Broadcast variables are read-only shared variables that are cached and available on all nodes in a cluster in-order to access or use by the tasks. Instead of sending this data along with every task, PySpark distributes broadcast variables to the machine using efficient broadcast algorithms to reduce communication costs.

One of the best use-case of PySpark RDD Broadcast is to use with lookup data for example zip code, state, country lookups e.t.c

When you run a PySpark RDD job that has the Broadcast variables defined and used, PySpark does the following.

PySpark breaks the job into stages that have distributed shuffling and actions are executed with in the stage.

- Later Stages are also broken into tasks
- PySpark broadcasts the common data (reusable) needed by tasks within each stage.
- The broadcasted data is cache in serialized format and deserialized before executing each task.

- The PySpark Broadcast is created using the broadcast(v) method of the SparkContext class. This method takes the argument v that you want to broadcast.

```
broadcastVar = sc.broadcast([0, 1, 2, 3])
broadcastVar.value
```

Note - that broadcast variables are not sent to executors with sc.broadcast(variable) call instead, they will be sent to executors when they are first used.

Accumulators

PySpark Accumulators are another type shared variable that are only "added" through an associative and commutative operation and are used to perform counters (Similar to Map-reduce counters) or sum operations.

PySpark by default supports creating an accumulator of any numeric type and provides the capability to add custom accumulator types. Programmers can create following accumulators

- named accumulators
- unnamed accumulators

Creating RDD from DataFrame and vice-versa

Though we have more advanced API's over RDD, we would often need to convert DataFrame to RDD or RDD to DataFrame. Below are several examples

```
# Converts RDD to DataFrame
dfFromRDD1 = rdd.toDF()
# Converts RDD to DataFrame with column names
dfFromRDD2 = rdd.toDF("col1","col2")
# using createDataFrame() - Convert DataFrame to RDD
df = spark.createDataFrame(rdd).toDF("col1","col2")
# Convert DataFrame to RDD
rdd = df.rdd
```

PySpark UDF (User Defined Function)

PySpark UDF (a.k.a User Defined Function) is the most useful feature of Spark SQL & DataFrame that is used to extend the PySpark build in capabilities.

Note: UDF's are the most expensive operations hence use them only when you have no choice and when essential.

- PySpark UDF is a User Defined Function that is used to create a reusable function in Spark.
- Once UDF created, that can be reused on multiple DataFrames and SQL (after registering).
- The default type of the udf() is StringType.
- You need to handle nulls explicitly otherwise you will see side-effects.

Performance concern using UDF

- UDFs are a black box to PySpark hence it can't apply optimization and you will lose all the optimization PySpark does on Dataframe/Dataset.
- When possible you should use Spark SQL built-in functions as these functions provide optimization.
- **Consider creating UDF only when the existing built-in SQL function doesn't have it.**

—-----------------------------------

How to create SparkSession?

SparkSession is created using SparkSession.builder().master("master-details").appName("app-name").getOrCreate(); Here, getOrCreate() method returns SparkSession if already exists. If not, it creates a new SparkSession.

How many SparkSessions can I create?

You can create as many SparkSession as you want in a Spark application using either SparkSession.builder() or SparkSession.newSession(). Many Spark session objects are required when you want to keep Spark tables (relational entities) logically separated.

How to stop SparkSession?

To stop SparkSession in Apache Spark, you can use the stop() method of the SparkSession object. If you have spark as a SparkSession object then call spark.stop() to stop the session. Calling a stop() is important to do when you're finished with your Spark application. This ensures that resources are properly released and the Spark application terminates gracefully.

How SparkSession is different from SparkContext?

SparkSession and SparkContext are two core components of Apache Spark. Though they sound similar, they serve different purposes and are used in different contexts within a Spark application.

SparkContext provides the connection to a Spark cluster and is responsible for coordinating and distributing the operations on that cluster. SparkContext is used for low-level RDD (Resilient Distributed Dataset) programming.

SparkSession was introduced in Spark 2.0 to provide a more convenient and unified API for working with structured data. It's designed to work with DataFrames and Datasets, which provide more structured and optimized operations than RDDs.

Do we need to stop SparkSession?

It is recommended to end the Spark session after finishing the Spark job in order for the JVMs to close and free the resources.

How do I know if my Spark session is active?

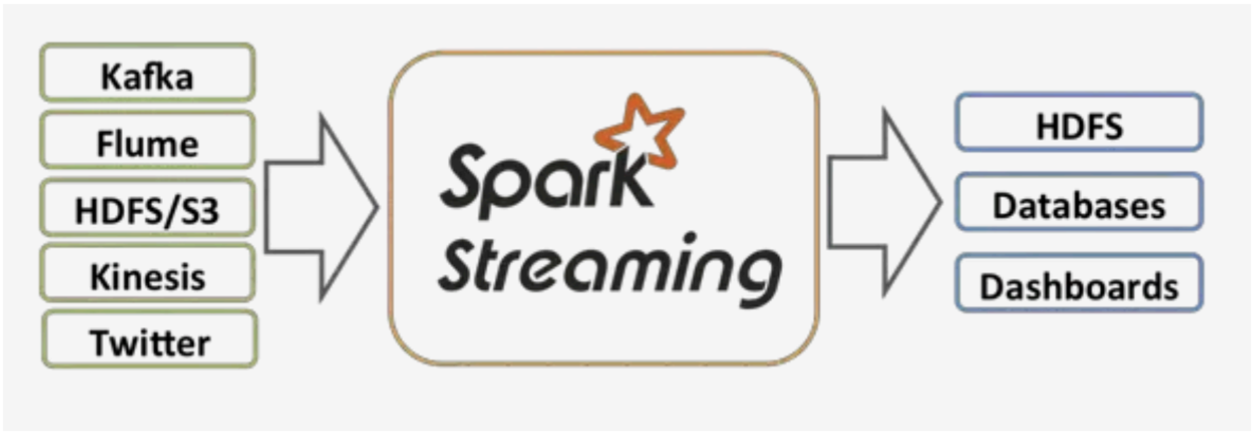
To check if your SparkSession is active, you can use the SparkSession object's sparkContext attribute and check its isActive property. If you have spark as a SparkSession object then call spark.sparkContext.isActive. This returns true if it is active otherwise false.

Can I have multiple SparkContext in Spark job?

There can only be one active SparkContext per JVM. Having multiple SparkContext instances in a single application can cause issues like resource conflicts, configuration conflicts, and unexpected behavior.

Spark Streaming

Spark Streaming is a scalable, high-throughput, fault-tolerant streaming processing system that supports both batch and streaming workloads. It is used to process real-time data from sources like file system folders, TCP sockets, S3, Kafka, Flume, Twitter, and Amazon Kinesis to name a few. The processed data can be pushed to databases, Kafka, live dashboards e.t.c



Memory Issues in pyspark

1. ****Out of Memory (OOM) Errors****:
  - ****Issue****: Spark applications may fail with OutOfMemoryError when they attempt to process data that cannot fit into the available memory.
  - ****Solution****:
    - Increase Executor Memory: Allocate more memory to Spark executors using the `--executor-memory` flag.
    - Tune Partition Size: **Adjust the size of partitions to ensure they fit into memory.**
    - Use Disk Storage: Utilize disk storage (`persist()` with `MEMORY_AND_DISK`) for RDDs that cannot fit entirely into memory.

2. ****Garbage Collection (GC) Overhead****:

- **Issue**: Frequent garbage collection can impact application performance and increase job execution time.
- **Solution**:
  - **Increase Heap Size: Adjust JVM heap size using `--driver-memory` and `--executor-memory` flags.**
  - Tune GC Settings: Configure garbage collection options (e.g., GC algorithm, GC interval) based on the workload and cluster configuration.
  - Use Off-Heap Memory: Store data off-heap to reduce GC pressure (`MEMORY_ONLY_SER` or `MEMORY_AND_DISK_SER`).

3. **Data Skew**:
- **Issue**: Uneven distribution of data across partitions can lead to skewed workload distribution and slow performance.
- **Solution**:
  - Partitioning: Repartition RDDs based on key to evenly distribute data.
  - Use Salting: Add randomness to keys to distribute skewed data evenly.
  - Aggregation Strategies: Use alternative aggregation strategies (e.g., **tree aggregation) to handle skewed data during shuffling.**

4. **Serialization Overhead**:
- **Issue**: Serialization and deserialization overhead can impact performance, especially for large datasets.
- **Solution**:
  - Use Efficient Serialization: Prefer Kryo serialization (`spark.serializer` set to `org.apache.spark.serializer.KryoSerializer`) over Java serialization.
  - **Serialize Data: Convert data to a serialized form (`MEMORY_ONLY_SER`, `MEMORY_AND_DISK_SER`) to reduce memory footprint.**

5. **Resource Contention**:
- **Issue**: Competition for resources (CPU, memory) among Spark applications running on the same cluster can lead to performance degradation.
- **Solution**:
  - Isolation: Use resource managers (e.g., YARN, Mesos) to isolate Spark applications and allocate resources based on priorities.
  - Dynamic Resource Allocation: Enable dynamic allocation (**`spark.dynamicAllocation.enabled=true`**) to scale resources dynamically based on workload.

Example:

Let's consider an example of addressing memory-related issues in Spark:

```python
from pyspark import SparkContext

# Initialize SparkContext with increased executor memory
sc = SparkContext("local", "MemoryExample", conf="spark.executor.memory=4g")

# Load input data
data = sc.textFile("input.txt")

# Perform transformations and actions on data
# (Example: Word count)
word_counts = data.flatMap(lambda line: line.split()) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a + b)

# Persist RDD in memory and disk
word_counts.persist()

# Save results to output file
word_counts.saveAsTextFile("output")

# Stop SparkContext
sc.stop()
```

In this example:

- We initialize the SparkContext with increased executor memory (`4g`) to mitigate OOM errors.
- We load input data and perform transformations (flatMap, map, reduceByKey) to compute word counts.

- We persist the RDD (`word_counts`) in memory and disk to optimize performance and handle potential OOM errors.
- Finally, we save the computed word counts to an output file.

PySpark Joins

Apache Spark provides several types of joins to combine datasets. These include:

1. **Inner Join**: Returns records that have matching values in both datasets.
2. **Outer Join** (Full Outer Join): Returns all records when there is a match in either left or right dataset.
3. **Left Join**: Returns all records from the left dataset and the matched records from the right dataset.
4. **Right Join**: Returns all records from the right dataset and the matched records from the left dataset.
5. **Left Semi Join**: Returns only the records from the left dataset for which there is a match in the right dataset.
6. **Left Anti Join**: Returns only the records from the left dataset for which there is no match in the right dataset.
7. **Cross Join**: Returns the Cartesian product of the two datasets (all possible combinations of records).

These joins are available in Spark SQL and can be performed using DataFrame or Dataset API.

Required Libraries:
In Apache Spark, you typically don't need any additional libraries to perform joins. The necessary functionalities are provided within the Spark SQL module, which is part of the core Spark distribution.

Example:
Let's demonstrate how to perform joins using Spark DataFrame API with an example:

```python
from pyspark.sql import SparkSession

# Initialize SparkSession
```

```
spark = SparkSession.builder \
    .appName("JoinExample") \
    .getOrCreate()

# Sample dataframes
df1 = spark.createDataFrame([(1, 'Alice'), (2, 'Bob'), (3, 'Charlie')], ["id", "name"])
df2 = spark.createDataFrame([(1, 25), (2, 30), (4, 35)], ["id", "age"])

# Inner Join
inner_join = df1.join(df2, "id", "inner")

# Outer Join (Full Outer Join)
outer_join = df1.join(df2, "id", "outer")

# Left Join
left_join = df1.join(df2, "id", "left")

# Right Join
right_join = df1.join(df2, "id", "right")

# Left Semi Join
left_semi_join = df1.join(df2, "id", "leftsemi")

# Left Anti Join
left_anti_join = df1.join(df2, "id", "leftanti")

# Cross Join
cross_join = df1.crossJoin(df2)

# Show results
inner_join.show()
outer_join.show()
left_join.show()
right_join.show()
left_semi_join.show()
left_anti_join.show()
cross_join.show()
```

```
# Stop SparkSession
spark.stop()
```

In this example:
- We create two DataFrames, `df1` and `df2`, with sample data.
- We perform various types of joins: inner join, outer join, left join, right join, left semi join, left anti join, and cross join.
- Each join operation is performed on the `id` column, which is common to both DataFrames.
- Finally, we show the results of each join operation using the `show()` method.

Optimize spark job to handle large data –

(PMCDODORPM)

P = Partitioning
M = Memory Management
C = Caching and Persistence
D = Data Serialization
O = Optimized Transformations
D = Data Skew Handling
O = Optimized Joins
R = Resource Management
P = Parallelism and Concurrency
M = Monitoring and Profiling

Optimizing Spark jobs that process huge volumes of data is crucial for achieving better performance and resource utilization.

Here are some tips to optimize Spark jobs for handling large datasets efficiently:

1. ****Partitioning****: Ensure proper data partitioning to distribute workload evenly across Executors. Use **`repartition()` or `partitionBy()`** methods to control the number and distribution of partitions based on the characteristics of your data and cluster resources.

2. ****Memory Management****:
  - ****Executor Memory Configuration****: Tune executor memory settings (**`spark.executor.memory`**) to allocate sufficient memory for data processing and caching.

  - **Driver Memory Configuration**: Adjust driver memory (`spark.driver.memory`) to handle driver-related tasks efficiently, especially when dealing with large datasets.

3. **Caching and Persistence**:
  - **Cache Frequently Accessed Data**: Cache or persist intermediate RDDs or DataFrames in memory using `.cache()` or `.persist()` to avoid recomputation and reduce disk I/O.
  - **Memory Management**: Monitor and manage memory usage to prevent out-of-memory errors and excessive garbage collection overhead.

4. **Data Serialization**:
  - **Use Efficient Serialization Formats**: Choose appropriate serialization formats (e.g., Kryo serialization) to minimize object overhead and reduce memory footprint, especially when dealing with complex data types.

5. **Optimized Transformations**:
  - **Use Narrow Transformations**: Prefer narrow transformations (e.g., `map`, `filter`) over wide transformations (e.g., `groupByKey`, `reduceByKey`) to minimize data shuffling and reduce network overhead.
  - **Broadcast Variables**: Use broadcast variables for efficiently distributing read-only data to all Executors, reducing data transfer and improving performance for certain operations like joins and lookups.

6. **Data Skew Handling**:
  - **Detect Data Skew**: Identify skewed data distribution using profiling tools (e.g., Spark UI) and monitoring metrics.
  - **Mitigate Data Skew**: Apply techniques such as salting, custom partitioning, or using `repartition` after filtering to distribute skewed data more evenly across partitions.

7. **Optimized Joins**:
  - **Join Strategies**: Choose appropriate join strategies (e.g., broadcast join, shuffle hash join) based on the size of datasets and available memory.
  - **Join Predicate Optimization**: Optimize join predicates to minimize data movement and improve join performance.

8. **Resource Management**:
  - **Dynamic Resource Allocation**: Enable dynamic resource allocation (`spark.dynamicAllocation.enabled`) to scale Executors dynamically based on workload, optimizing resource utilization.
  - **YARN Configuration**: Configure YARN settings (if using Hadoop YARN) to allocate sufficient resources and manage container sizes effectively.

9. **Parallelism and Concurrency**:
  - **Adjust Parallelism**: Tune `spark.default.parallelism` and `spark.sql.shuffle.partitions` to control parallelism and concurrency based on cluster capacity and workload characteristics.
  - **Concurrency Control**: Limit concurrent tasks or stages using `spark.task.maxFailures` and `spark.scheduler.maxRegisteredResourcesWaitingTime` to prevent resource contention and improve stability.

10. **Monitoring and Profiling**:
   - **Spark UI**: Monitor Spark jobs using the Spark UI to identify bottlenecks, analyze execution stages, and track resource usage.
   - **Performance Profiling**: Use profiling tools (e.g Spark History Server, Ganglia) to analyze job performance, identify hotspots, and optimize resource allocation.

—----------------------------