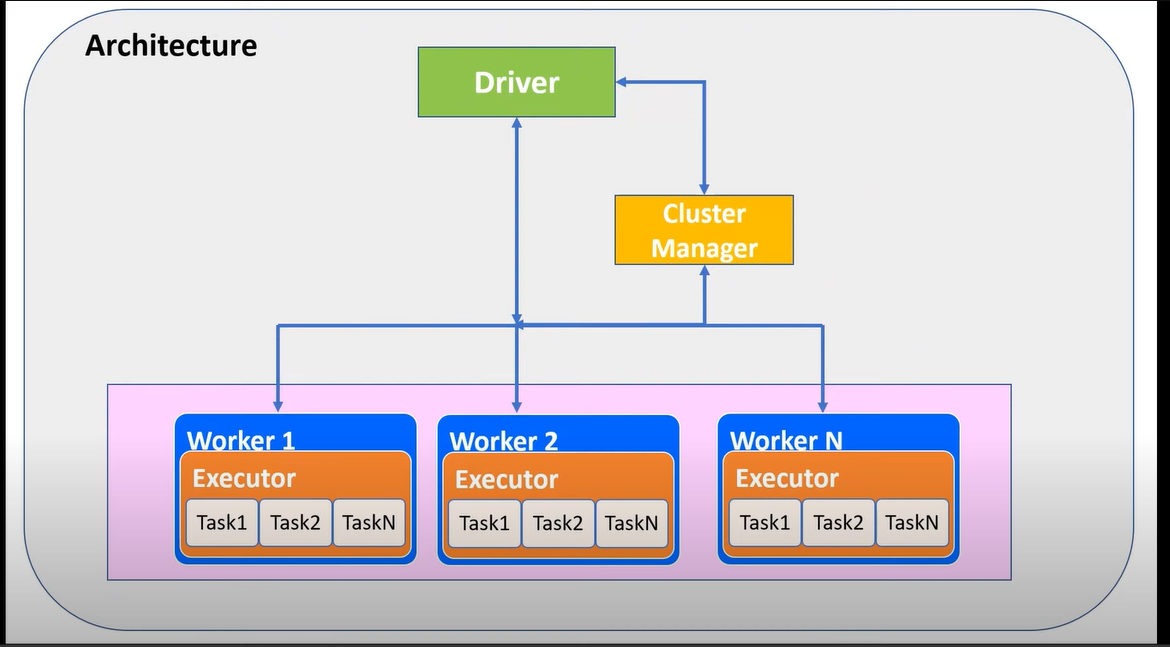
**Spark**

Spark is an open-source distribution computing engine. We use it for processing and analyzing a large amount of data. Like Hadoop, spark also works in distributed nature but differs in-memory processing.

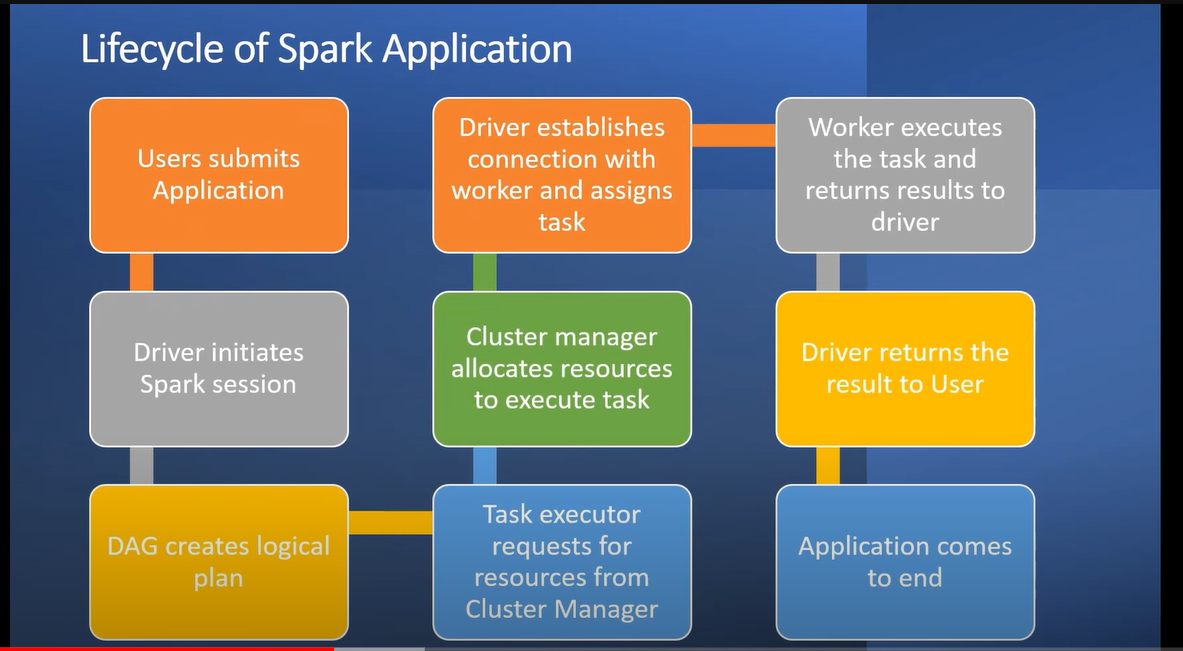
Compared to Hadoop or traditional systems, 100 times faster in memory and 10 times faster in disc.

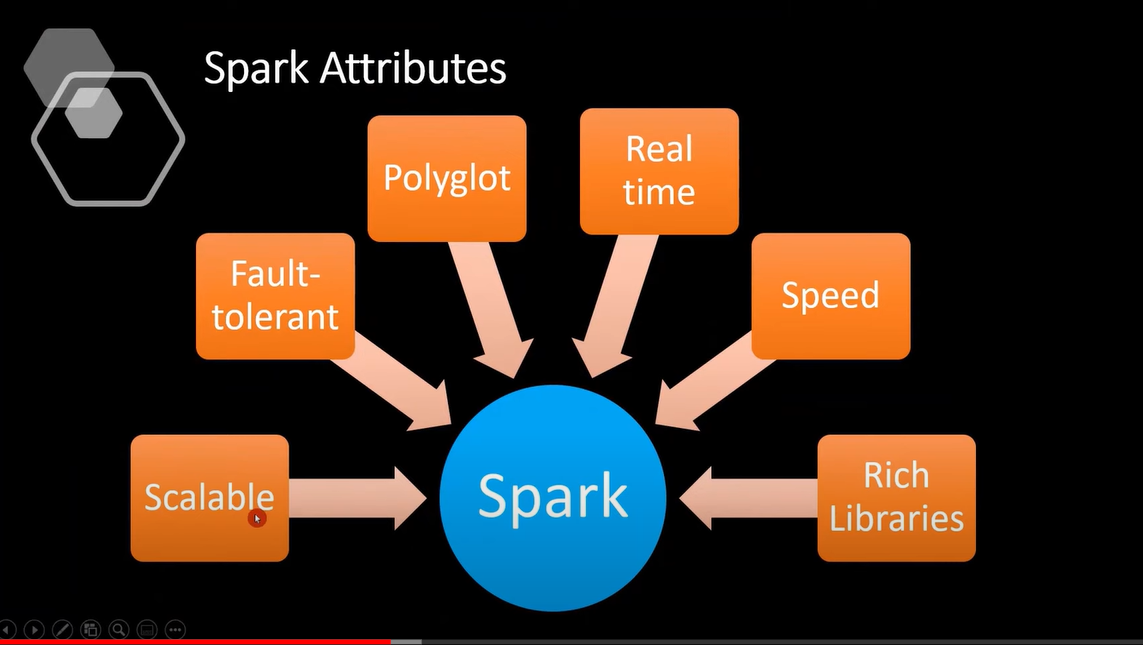
Its lightning speed is from in-memory and parallel processing.

Spark has well designed layered architecture. It follows master/slave concept. That is the driver and worker concept. Between driver and worker layer comes Cluster manager layer. These layers, driver, worker and cluster manager and designed well within its boundary and loosely coupled to each other.









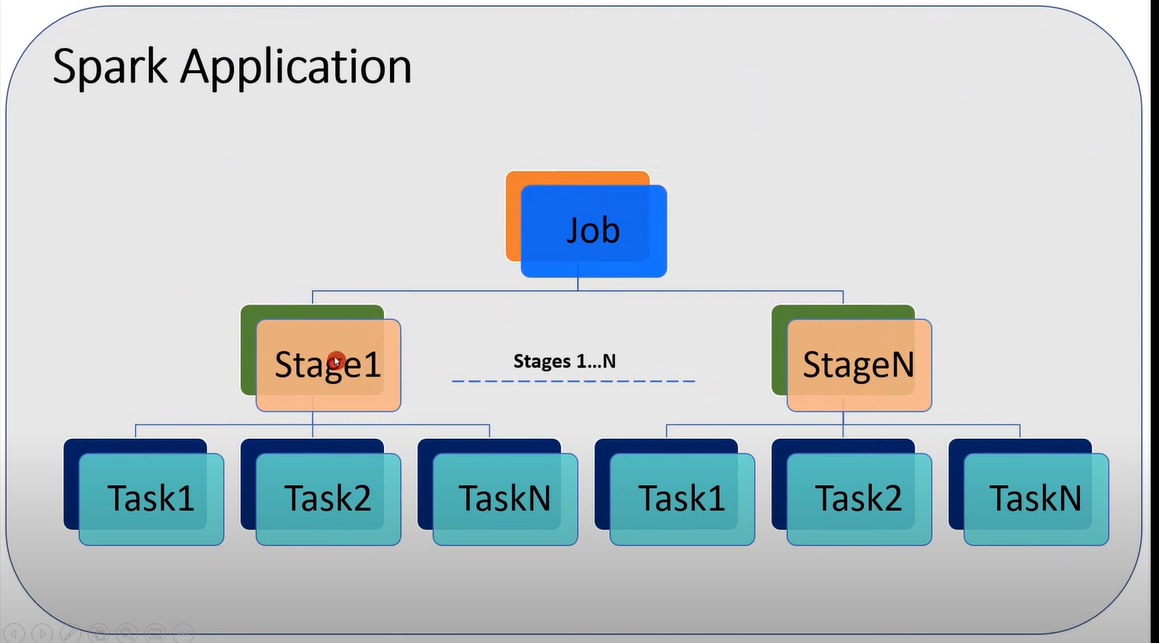
**Driver and worker process:** These are mothing but JVM process. Within one worker node, there could be multiple executors. Each executor runs its own JVM process.

**Application:** It could be single command or combination of multiple notebooks with complex logic. When code is submitted to spark for execution, application starts.

**Jobs:** When an application is submitted to spark for execution, application starts.

**Stage:** Jobs are divided into stages. If the application code demands shuffling the data across nodes, a new stage is created. The number of stages are determined by the number of shuffling operations. Join is an example of shuffling operation.

**Tasks:** Stages are further divided into multiple tasks. All tasks would execute the same logic. Each task would process one task at a time.



**Transformations:** It is a kind of operation which will transform DataFrame from one form to another form.

It transforms the input RDD and creates new RDD until actions are called and transformations are evaluated lazily.

Ex: Filter, Union etc.

**DAG:** Directed Acyclic graph keeps record of all the transformations. For each transformations logical plan is created and lineage graph is maintained by the graph.

**Action:** When the data output is needed for developer or for storage purpose action is called. Action would be executed based on the DAG and processes the actual data.

Ex: Count, Collect and Save etc.

**RDD:** Resilient distribution dataset is basic data structure of spark. When spark reads or creates data, it creates RDD which is distributed across nodes in the form partition.

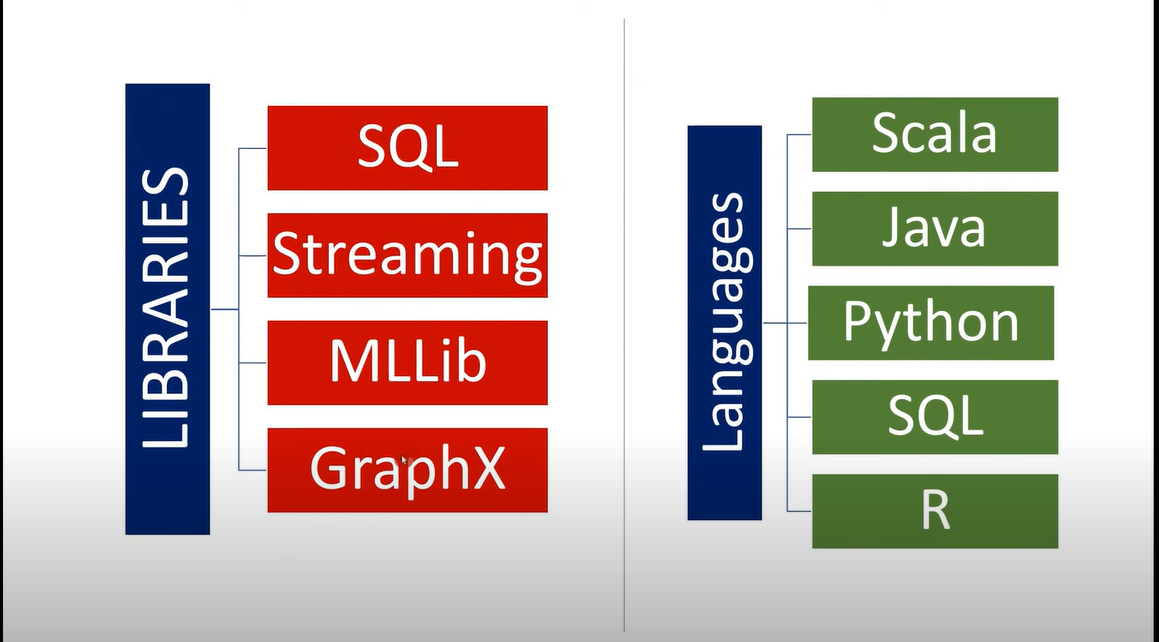
**Executor:** Each worker node can consist of many executors. It can be configured by spark settings.

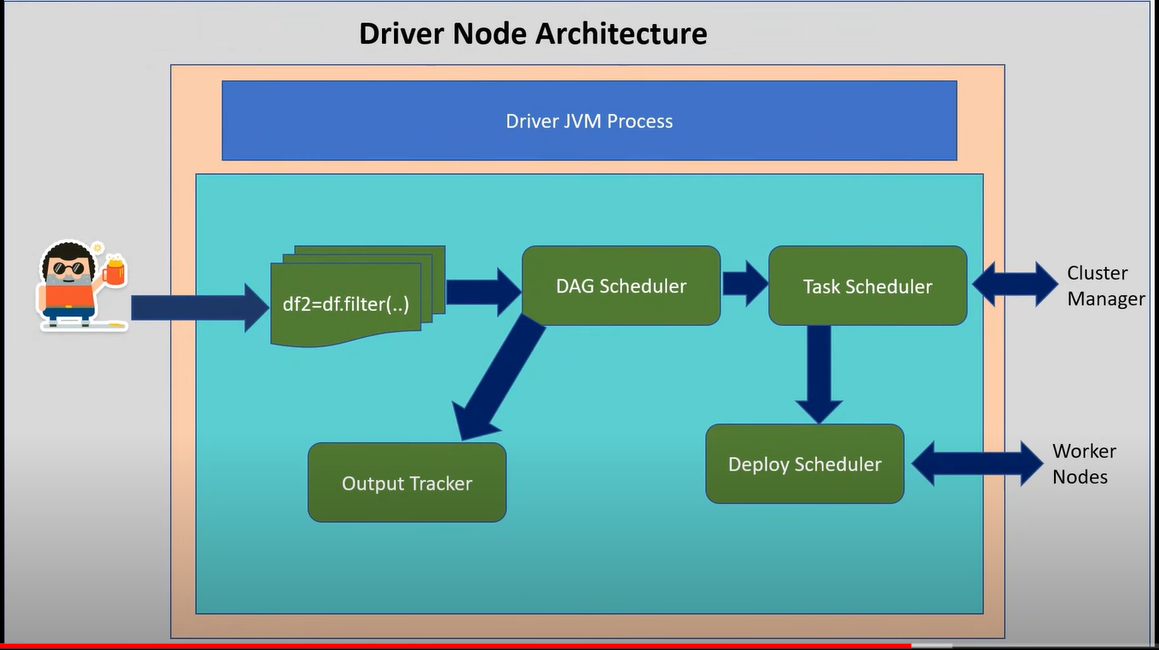
**Partitions:** RDD/Data is stored in-memory of cluster in the form of partition.

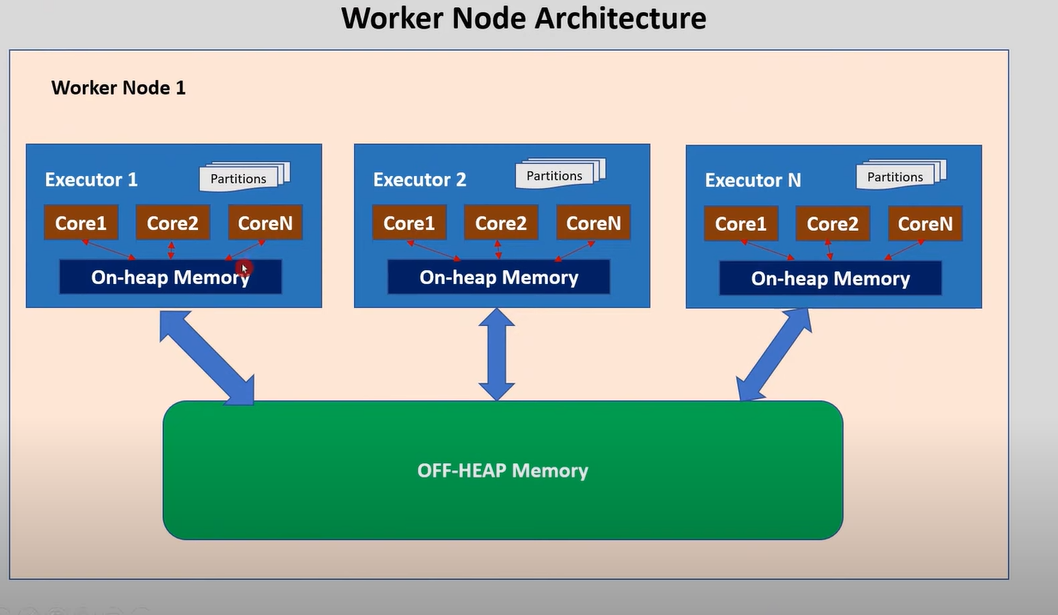
**Core:** Each executor can consist of multiple cores. This is configurable by spark settings.

**On-Heap-Memory:** The executor memory that lies within the JVM process managed JVM.

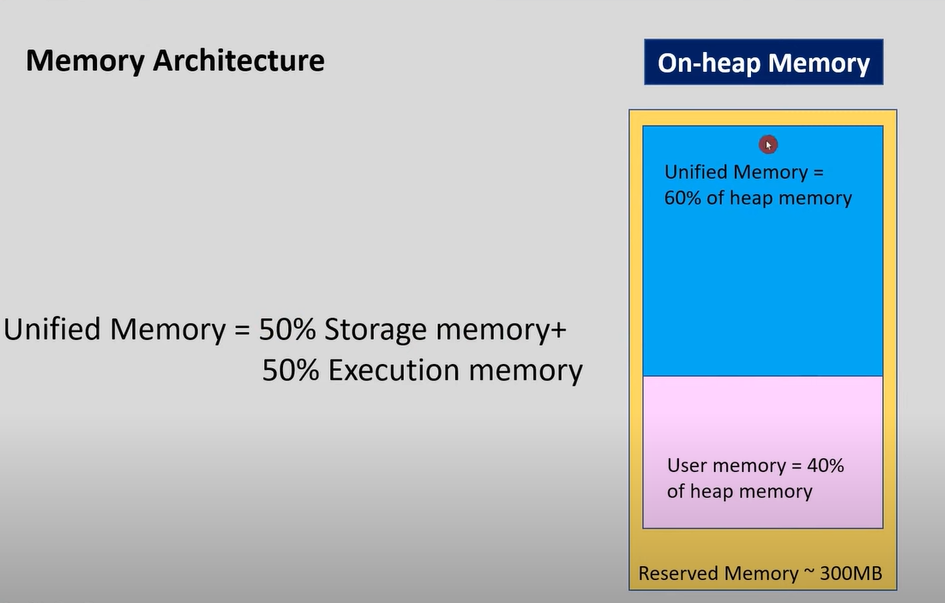
**Off -Heap-Memory:** The executor memory that lies outside the JVM process managed by the OS.

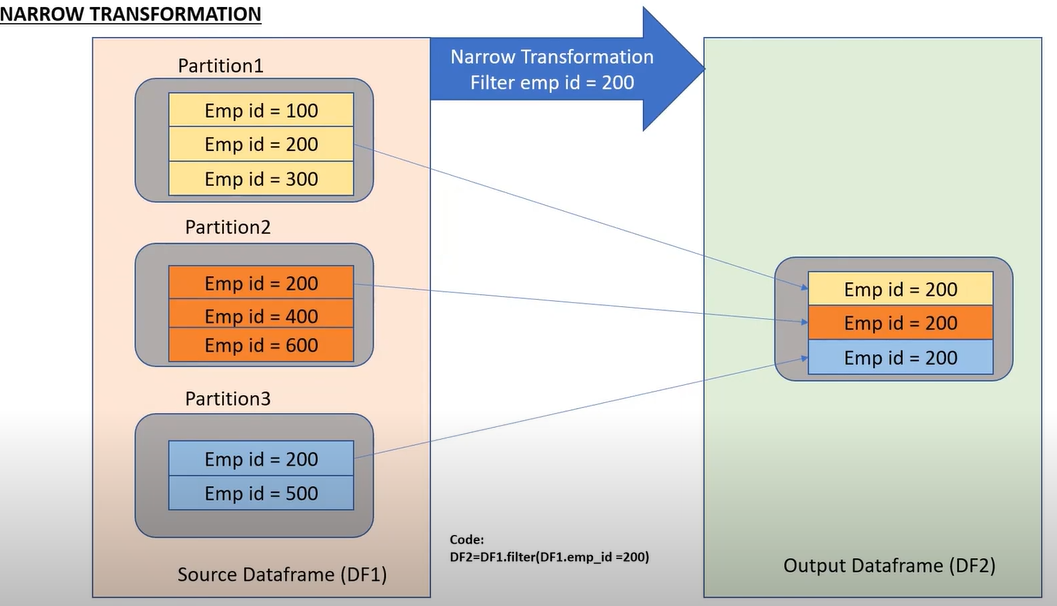


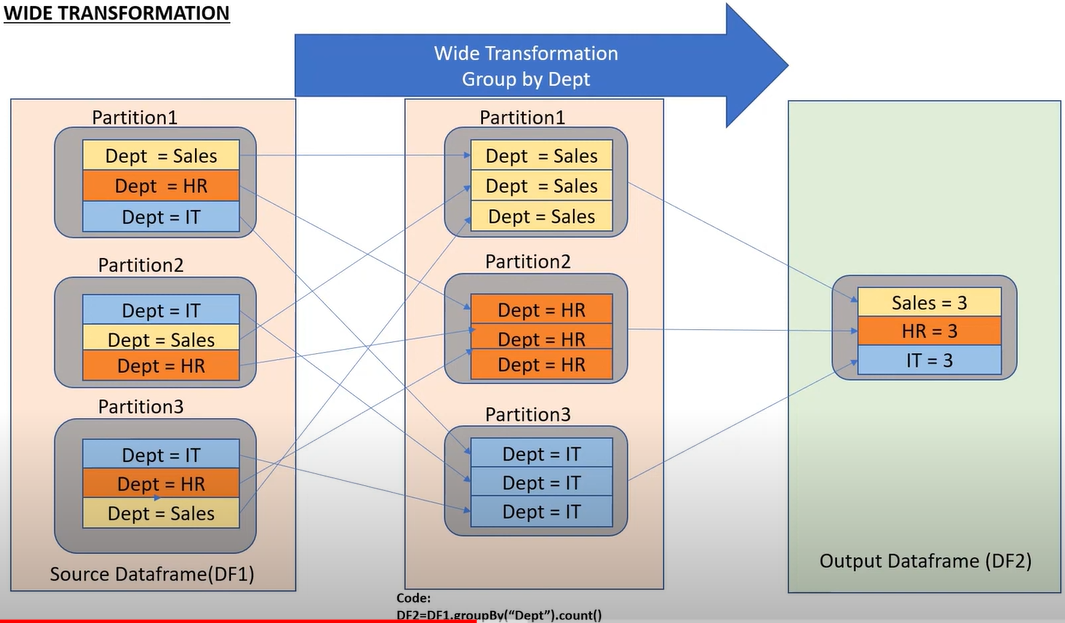


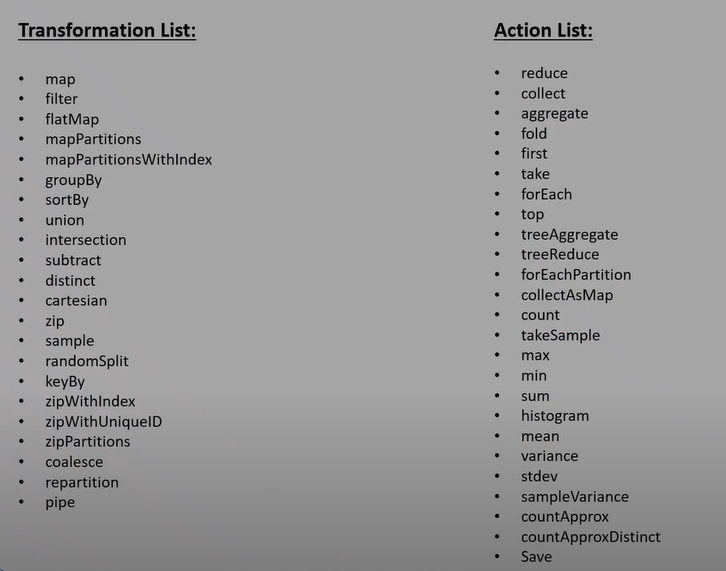


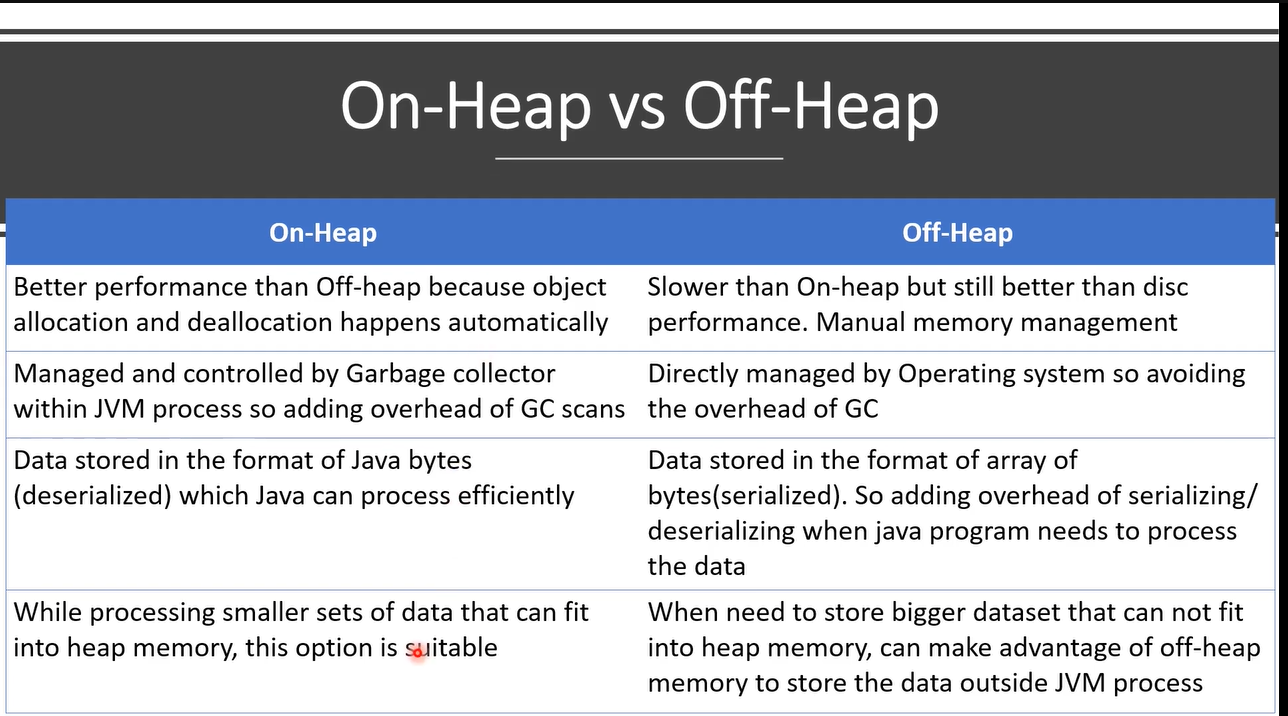
* Each executor within worker node has access to off-heap memory.
* Off-Heap memory can be used by the spark explicit storing its data.
* The amount of off-heap memory used by the spark to store actual data frames is governed by spark.memory.offHeap.size.
* To enable off-heap memory set spark.memory.offHeap.use to true.
* Accessing off-heap is slightly slower than accessing the on-heap storage but still faster than reading/writing from disk.
* GC (Garbage collector) scan can be avoided by using off-heap memory.

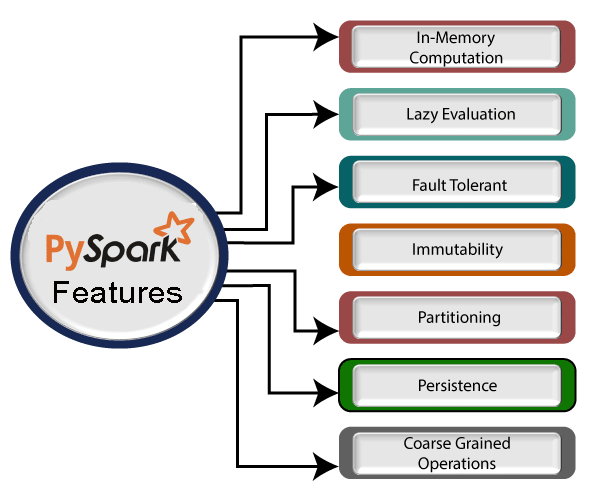












**Difference between Spark and PySpark:**

* **Language:** Spark is written in Scala, while PySpark is a Python API for Spark. This means that PySpark can be used to run Spark jobs from Python.
* **Data structures:** Spark uses a variety of data structures, such as RDDs, DataFrames, and Datasets. PySpark also supports these data structures, but it provides a more Pythonic interface for working with them.
* **API:** Spark has a rich API that allows you to perform a variety of tasks, such as data processing, machine learning, and graph processing. PySpark provides a subset of the Spark API, but it is still a powerful tool for working with large datasets.
* **Performance:** PySpark can be slightly slower than Spark because of the overhead introduced by the Python interpreter. However, PySpark is still a very fast tool for processing large datasets.
* **Community:** Spark has a large and active community of users and contributors. PySpark also has a large and active community, but it is not as large as the Spark community.
* Overall, Spark is a powerful framework for processing large datasets. PySpark is a Python API for Spark that makes it easy to use Spark from Python. PySpark is a good choice for users who are familiar with Python and who want to use Spark to process large datasets.

Here are some additional things to consider when choosing between Spark and PySpark:

* **Your programming language skills:** If you are more comfortable with Python than Scala, then PySpark is a good choice.
* **The size of your dataset:** If you are working with a large dataset, then Spark is a good choice. PySpark can be used to process large datasets, but it may be slower than Spark.
* **The availability of resources:** If you have a lot of resources available, then Spark is a good choice. PySpark can also be used with limited resources, but it may not be as performant.

**What is optimization in pyspark ?**

Optimization in PySpark refers to the process of improving the performance of Spark jobs. This can be done by using a variety of techniques, such as:

* **Using the right data structures:** Spark supports a variety of data structures, such as RDDs, DataFrames, and Datasets. Each data structure has its own advantages and disadvantages. The best data structure to use will depend on the specific task.
* **Using the right algorithms:** Spark provides a variety of algorithms for common tasks, such as sorting, aggregation, and joins. The best algorithm to use will depend on the specific task and the available resources.
* **Using the right execution plan:** Spark uses a cost-based optimizer to choose the best execution plan for a Spark job. The optimizer takes into account factors such as the number of partitions, the size of the data, and the available resources. The best execution plan to use will depend on the specific Spark job.
* **Using the right configuration settings:** Spark provides a number of configuration settings that can affect performance. For example, you can configure the number of cores used by each worker node, the amount of memory used by each worker node, and the number of shuffle partitions. The best configuration settings to use will depend on the specific Spark job and the available resources.
* **Using the right tools:** There are a number of tools available to help you optimize Spark jobs. These tools can help you profile your Spark jobs, identify bottlenecks, and make recommendations for improvement.

Here are some specific examples of optimization techniques that can be used in PySpark:

* **Use DataFrames and Datasets over RDDs:** DataFrames and Datasets are more optimized than RDDs. They provide a higher level of abstraction and allow Spark to perform more optimizations.
* **Use broadcast variables:** Broadcast variables are used to share data across all worker nodes in a Spark cluster. This can improve performance by avoiding the need to transfer data between nodes.
* **Use coalescing and repartitioning:** Coalescing and repartitioning are used to change the number of partitions in a Spark DataFrame or Dataset. This can improve performance by reducing the amount of data that needs to be transferred between nodes.
* **Use caching and persisting:** Caching and persisting are used to store data in memory or on disk. This can improve performance by avoiding the need to read data from the underlying storage system.
* **Use optimized libraries:** Spark provides a number of optimized libraries for common tasks, such as machine learning and graph processing. These libraries can improve performance by using specialized algorithms and data structures.
* **Use the right execution plan:** Spark uses a cost-based optimizer to choose the best execution plan for a Spark job. The optimizer takes into account factors such as the number of partitions, the size of the data, and the available resources. You can improve performance by using the right execution plan.
* **Use the right configuration settings:** Spark provides a number of configuration settings that can affect performance. For example, you can configure the number of cores used by each worker node, the amount of memory used by each worker node, and the number of shuffle partitions. You can improve performance by using the right configuration settings.

**Functions:**

1. **Explode:** The explode () function in PySpark takes a column of arrays or maps and expands it to a new row for each element in the array or map.

The syntax is: df.explode(column\_name)

1. **Explode\_outer:** The explode\_outer() function in PySpark is similar to the explode () function, but it also creates a new row for the case when the array or map is null or empty.

The syntax is: df.explode\_outer(column\_name)

1. **Posexplode:** The posexplode() function in PySpark takes a column of arrays and expands it to a new row for each element in the array, along with the position of the element in the array.

The syntax is: df.posexplode(column\_name)

1. **Poexplode\_outer:** The posexplode\_outer() function in PySpark is similar to the posexplode() function, but it also creates a new row for the case when the array is null or empty.

The syntax is: df.posexplode\_outer(column\_name)

1. **withColumn():** The withColumn() function in PySpark is used to add a new column to a DataFrame.

The syntax is: df.withColumn(new\_column\_name, transformation\_function(column\_name))

**What is Z-Order By?**

* Z-order is a type of ordering that is used to colocate related information in the same files. This can improve the performance of queries that access multiple columns, as the data is more likely to be stored in close proximity.
* The zorder by clause in the optimize statement tells Spark to sort the data in the table fct\_mar2023\_pg\_innetwork\_main by the columns service code and billing code. This will cause the data to be stored in a z-order fashion, which can improve the performance of queries that access these columns.
* For example, a query that selects all rows where the service code is 123 and the billing code is 456 will be able to read the data more efficiently if the data is stored in z-order fashion. This is because the rows with the service code of 123 and the billing code of 456 will be stored in the same file.
* The z-order by clause can be used with any column in a table. However, it is most effective when used with columns that are frequently used together in queries.

**Window Functions: Lead and Lag in Pyspark:**

* The **lead**() and **lag()** functions in PySpark are window functions that are used to access previous or subsequent rows of data, respectively. They are both defined in the pyspark.sql.functions module.

**The syntax for the lead() function is:**

lead(column\_name, offset, default\_value)

* **column\_name** is the name of the column to access
* **offset** is the number of rows to offset, with a positive value indicating subsequent rows and a negative value indicating previous rows
* **default\_value** is the value to return if there are not enough rows to offset

following code uses the lead() function to add a new column to a DataFrame that contains the value of the salary column in the previous row:

* df = spark.read.csv("data.csv")  
  df = df.withColumn("salary\_lag", lead("salary", 1))

The lead() and lag() functions can be used to perform a variety of tasks, such as:

* Calculating moving averages
* Identifying trends
* Detecting outliers
* Comparing values over time

**Corrupt Record handling:**

Differences between the **FAILFAST, PERMISSIVE,** and **DROPMALFORMED** modes in Spark Dataframes:

**FAILFAST** mode throws an exception if any corrupted record is encountered. This is the strictest mode and is best used when the data is critical and cannot be corrupted.

**PERMISSIVE** mode sets other fields to null when it meets a corrupted record. This mode is more forgiving and can be used when the data is not critical.

**DROPMALFORMED** mode ignores the whole corrupted records. This mode is a good compromise between FAILFAST and PERMISSIVE modes.

**PERMISSIVE**

* The default mode is **PERMISSIVE**. You can specify the mode when you read a file into a DataFrame using the mode parameter.

For example, the following code reads a file in **PERMISSIVE mode:**

**df = spark.read.csv ("data.csv", mode="PERMISSIVE")**

**FAILFAST**

* You can also specify the mode when you create a DataFrame from a list of rows using the mode parameter.

**For example**, the following code creates a DataFrame in **FAILFAST mode**:

**df = spark.createDataFrame ([1, 2, 3], ["number"], mode="FAILFAST")**

**DROPMALFORMED**

* When a corrupted record is encountered in DROPMALFORMED mode, the record is simply skipped and not included in the DataFrame. The other records in the DataFrame are not affected.
* The DROPMALFORMED mode is the default mode for reading CSV files into Spark DataFrames. This is because CSV files are often not well-formed and can contain corrupted records.

**Here is an example** of how to use the **DROPMALFORMED mode:**

**df = spark.read.csv ("data.csv", mode="DROPMALFORMED")**

## What is Apache Spark?

* Apache Spark is an open-source distributed cluster-computing framework introduced by Apache Software Foundation. It is a general engine for big data analysis, processing and computation. It is built for high speed, ease of use, offers simplicity, stream analysis and runs virtually anywhere. It can analyze data in real-time. It provides fast computation over the big data.
* Fast computation means that it's faster than previous approaches to work with Big Data such as MapReduce. The main feature of Apache Spark is its in-memory cluster computing that enhances the processing speed of an application.
* It can be used for multiple things like running distributed SQL, creating data pipelines, ingesting data into a database, running Machine Learning algorithms, working with graphs or data streams, and many more.