22MCA2052 – Big Data Analytics
Module – 4: Big Data and Spark

Text Book: Bill Chambers, Matei Zaharia, "Spark: The Definitive Guide", O'reilly, 2018.

What is Apache Spark? Spark's Architecture, it's language API, Data Frames, Partitions, Lazy Evaluation, Spark's Toolset, Overview of Structured API Execution.

Before Spark:

Need for Spark:

- Need for a powerful engine that can process the data in Real-Time (streaming) as well as in Batch mode
- Need for a powerful engine that can respond in Sub-second and perform In-memory analytics
- Need for a powerful engine that can handle diverse workloads:
  - Batch
  - Streaming
  - Interactive
  - Graph
  - Machine Learning

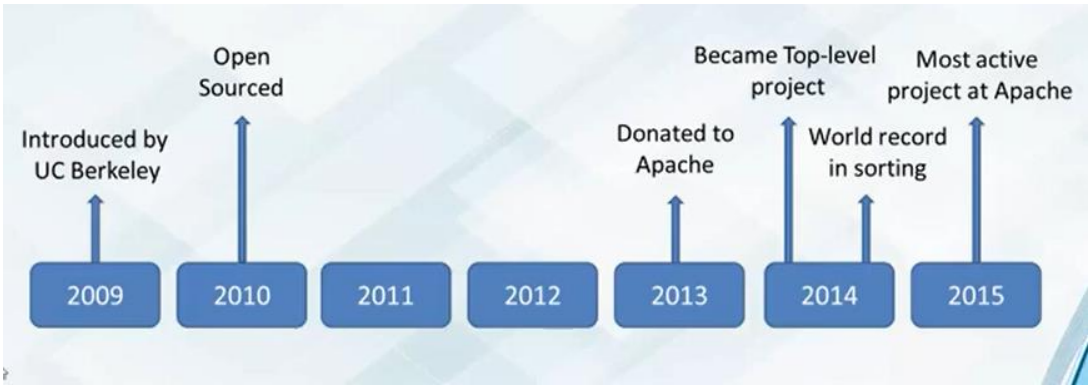
Why Spark:

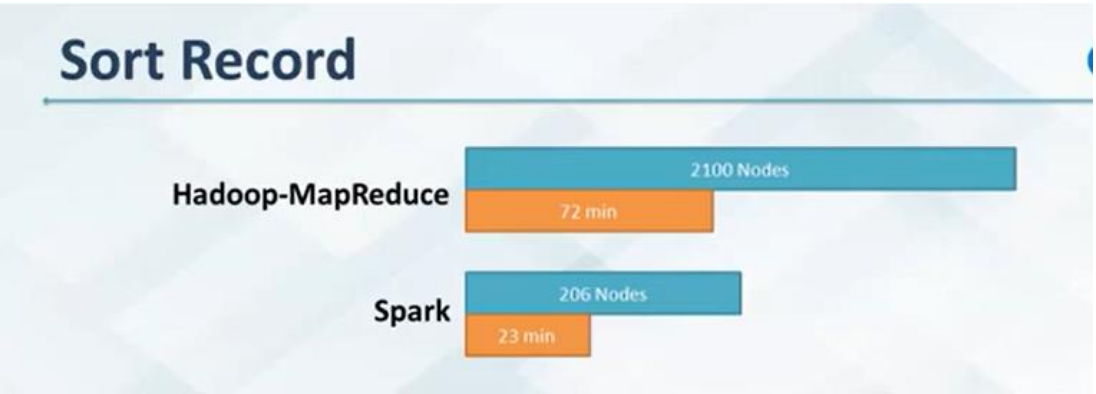
Apache Spark is a powerful open source engine which can handle:

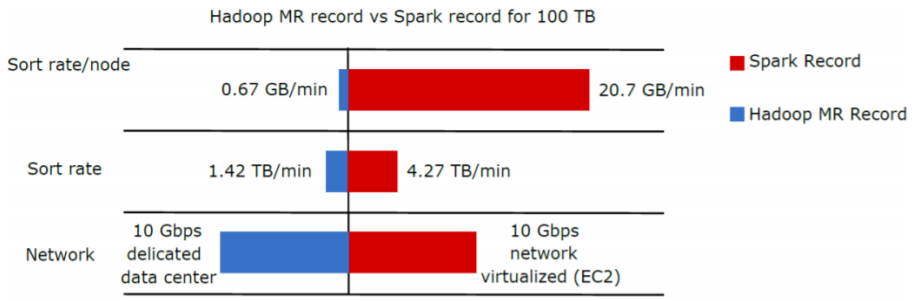
- Batch processing
- Real-time (stream)
- Interactive
- Graph
- Machine Learning (Iterative)
- In-memory

Introduction to Spark:

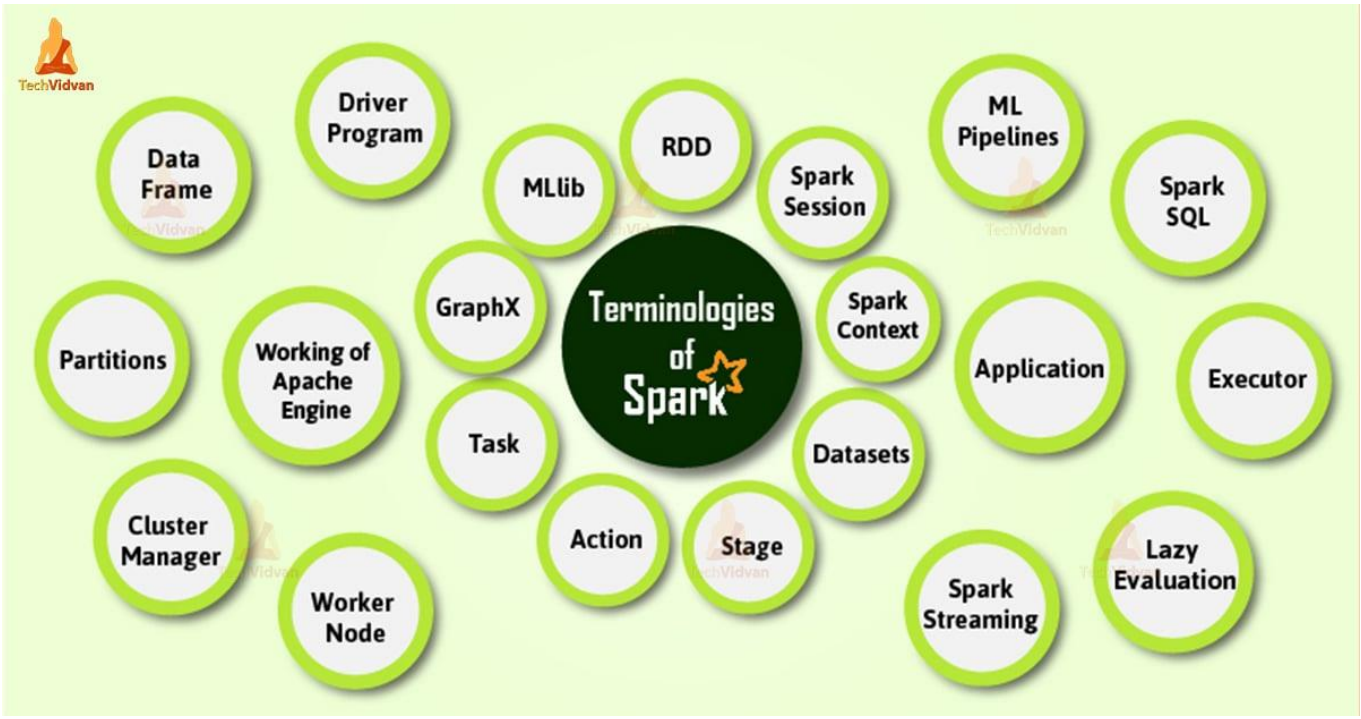
History of Spark:

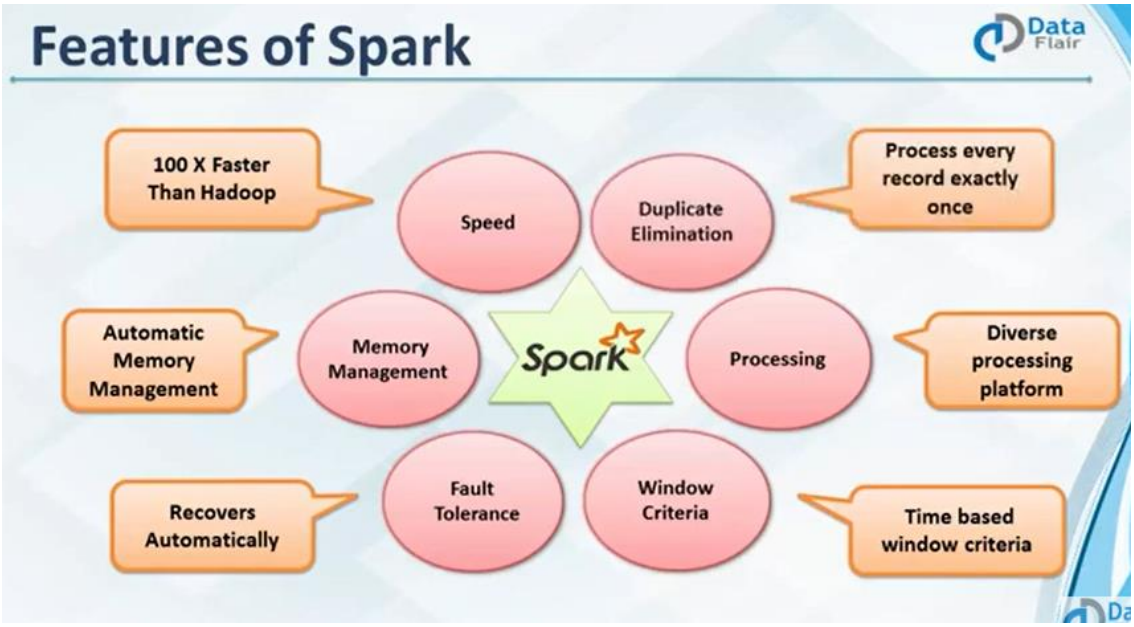






Key Terminologies associated with Spark:





https://data-flair.training/blogs/introduction-to-apache-spark-video-tutorial/

https://data-flair.training/blogs/spark-notes/

About Spark

Introduction:

- Industries are using Hadoop extensively, since 2008, to analyse their data sets as the 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.
- However, the main concern with Hadoop is the speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.
- Apache Spark is considered as the solution at this stage to mitigate the concerns of speed encountered during processing in Hadoop.
- Spark was introduced by Apache Software Foundation (ASF) in 2013 for speeding up the Hadoop computational computing software process.
- Spark is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.
- Spark is **not a modified version / extension of Hadoop**.
- Spark is **independent of Hadoop** because it has its own cluster management system**.**
- Spark uses Hadoop for **storage** purposes only.

Spark:

- Apache Spark is a **lightning-fast cluster computing** technology, designed for fast computation.
- Spark is a **unified computing engine and a set of libraries** for parallel data processing on computer clusters.
- Spark supports **multiple widely used programming languages** (Python, Java, Scala, and R), includes libraries for diverse tasks ranging from SQL to streaming and machine learning, and runs anywhere from a laptop to a cluster of thousands of servers.
- The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application.
- Spark is **simple, fast, scalable and unified**.
- Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming.
- Spark is the most widely-used engine for scalable computing.
- Spark is designed in such a way that **it integrates with all the Big data tools**.
- Thousands of companies, including 80% of the Fortune 500, use Apache Spark.
- Over 2,000 developers contribute to the open source project from industry and academia.
- Spark is completely developed in Scala and present version of Spark available is **3.4.1.**

Features of Apache Spark:

a. Swift Processing

The key feature required for Bigdata evaluation is speed. With Apache Spark, we get swift processing speed of up to 100x faster in memory and 10x faster than Hadoop even when running on disk. It is achieved by reducing the number of read-write to disk.

b. Dynamic

Because of 80 high-level operators present in Apache Spark, it makes it possible to develop parallel applications. Scala being defaulted language for Spark. We can also work with Java, Python, R. Hence, it provides dynamicity and overcomes the limitation of Hadoop MapReduce that it can build applications only in java.

c. In – Memory Processing

Disk seeks is becoming very costly with increasing volumes of data. Reading terabytes to petabytes of data from disk and writing back to disk, again and again, is not acceptable. Hence in-memory processing in Spark works as a boon to increasing the processing speed. Spark keeps data in memory for faster access. Keeping data in servers' RAM as it makes accessing stored data quickly /faster. Spark owns advanced DAG execution engine which facilitates in-memory computation and acyclic data flow resulting high speed. You can find out the more detail about Spark in-memory computation.

d. Reusability

Apache Spark provides the provision of code reusability for batch processing, join streams against historical data, or run adhoc queries on stream state.

e. Fault Tolerance

Spark RDD (Resilient Distributed Dataset), abstraction are designed to seamlessly handle failures of any worker nodes in the cluster. Thus, the loss of data and information is negligible.

f. Real-Time Stream Processing

Spark Streaming can handle real-time stream processing along with the integration of other frameworks which concludes that sparks streaming ability are easy, fault tolerance and Integrated.

g. Pillar to Sophisticated Analytics

Spark comes with tools for interactive/declarative queries, streaming data, machine learning which is an addition to the simple map and reduces, so that user can combine all this into a single workflow.

h. Compatibility with Hadoop & existing Hadoop Data

Apache Spark is compatible with both versions of Hadoop ecosystem. Be it YARN (Yet Another Resource Negotiator) or SIMR (Spark in MapReduce). It can read anything existing Hadoop data that's what makes it suitable for migration of pure Hadoop applications. It can run independently too.

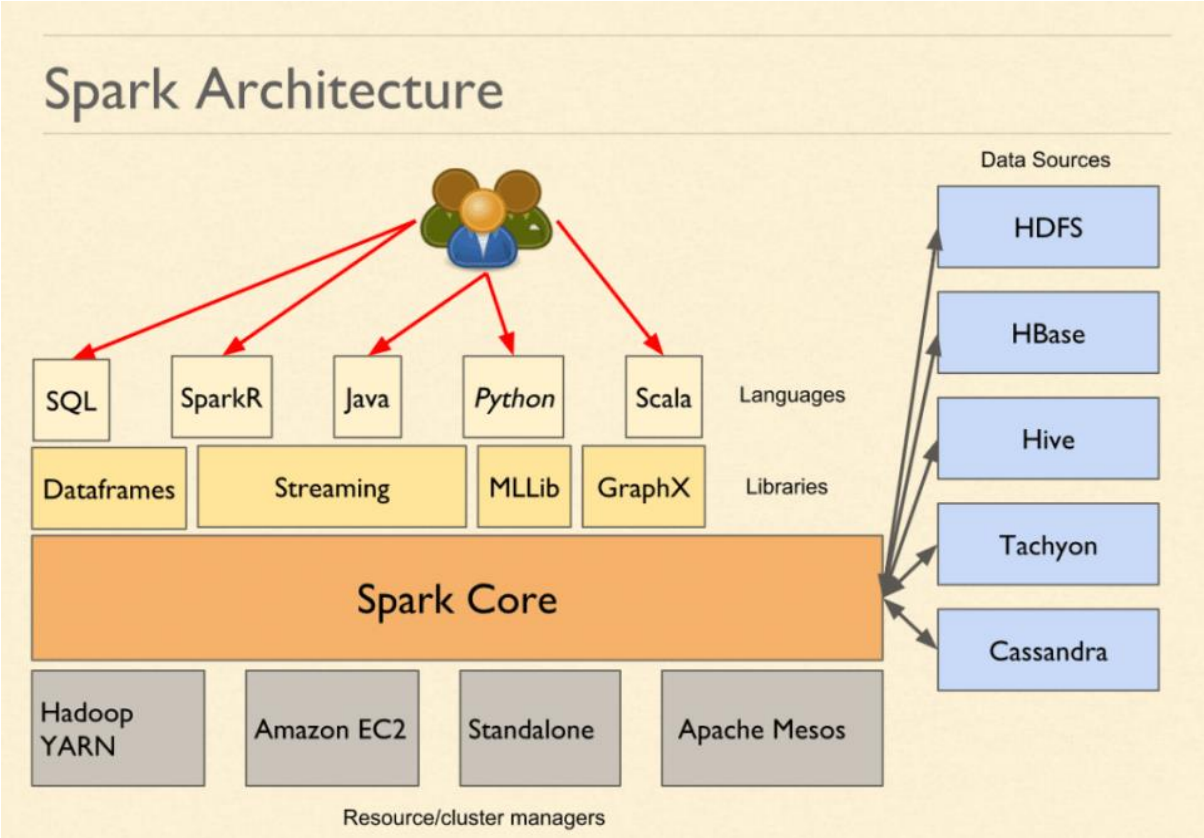
i. Lazy Evaluation

Lazy Evaluation is an Another outstanding feature of Apache Spark is called by need or memorization. It waits for instructions before providing a final result which saves time.

j. Active, Progressive and Expanding Community

A wide set of developers from over 50 companies build Apache Spark. It has active mailing state and JIRA for issue tracking.

Spark Architecture:



a. Spark Core

It is the foundation of Spark. Spark core is also shelter to API that contains the backbone of Spark i.e. RDDs (resilient distributed datasets). The basic functionality of Spark is present in Spark Core like memory management, fault recovery, interaction with the storage system. It is in charge of essential I/O functionalities like:

- Programming and observing the role of Spark cluster
- Task dispatching
- Fault recovery
- It overcomes the snag of MapReduce by using in-memory computation.

b. SparkSQL

Spark SQL component is a distributed framework. It is a Spark package that allows working with structured and semi-structured data. Spark SQL allows querying in **SQL** and **HQL** too which provides declarative query with optimized storage running in parallel.

It enables powerful interactive and analytical application across both streaming and historical data. SparkSQL allows accessing data from multiple sources like Hive table, Parquet and JSON. It also lets you intermix SQL query with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics.

c. Spark Streaming

Spark Streaming enables processing of the large stream of data. It makes easy for the programmer to move between an application that manipulates data stored in memory, on disk and arriving in real time. Micro-batching is used for real time

streaming. Initially the small batches are formed from the live data and then delivered to the Spark batch processing System for processing. In short, it provides data abstraction known as DStream. It also provides fault tolerance characteristics.

d. MLlib

Spark MLlib (Machine learning library) provides algorithms like **machine learning** as well as statistical, which further includes classification, regression, clustering, collaborative filtering as well as supporting functions such as model evaluation and data import. MLlib is a scalable learning library that discusses high-quality algorithms and high speed. It also contains some lower-level primitives.

e. GraphX

GraphX is an API for graphs and graph parallel execution. It is a network graph analytics engine. GraphX is a library that performs graph-parallel computation and manipulates graph. It has various Spark RDD API so it can help to create directed graphs with arbitrary properties linked to its vertex and edges. Spark GraphX also provides various operator and algorithms to manipulate graph. Clustering, classification, traversal, searching, and pathfinding is possible in GraphX.

f. SparkR

It allows data scientists to analyze large datasets and interactively run jobs on them from the **R** shell. The main idea behind SparkR is to explore different techniques to integrate the usability of R with the scalability of Spark. It is R package that gives light-weight frontend to use Apache Spark from R.

Abstractions of Apache Spark:

Spark has several core abstractions namely **Datasets**, **DataFrames** and **Resilient Distributed Datasets (RDDs).** These different abstractions represent distributed collections of data. The easiest and most efficient are **Data Frames** and **RDDs** which are available in all languages.

Resilient Distributed Datasets (RDD):
RDDs (Resilient Distributed Datasets) are a read-only collection of data elements without a schema. RDDs are a set of Java or Scala objects that are partitioned across cluster nodes. RDDs allow parallel computation. It helps in re-computing data in case of failures. RDD is the fundamental data structure of Spark. It allows a programmer to perform in-memory computations on large clusters in a fault-tolerant manner. Thus, speed up the task. RDDs are the most basic and low-level API, providing more control over the data but with lower-level optimizations.

DataFrames:
DataFrames are a distributed collection of data organized into named columns. DataFrames are similar to tables in relational databases. It is an immutable distributed collection of data. DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction. DataFrames provide a higher-level API that is optimized for performance and easier to work with for structured data.

Datasets:
Datasets are an extension of DataFrames with added features like type-safety and object-oriented interface. Datasets are a type-safe version of Spark's structured API for Java and Scala Dataset takes advantage of Spark's Catalyst optimizer by exposing expressions and data fields to a query planner. Datasets are similar to DataFrames in performance but with stronger typing and code generation, making them a good choice for high-performance batch and stream processing with strong typing.

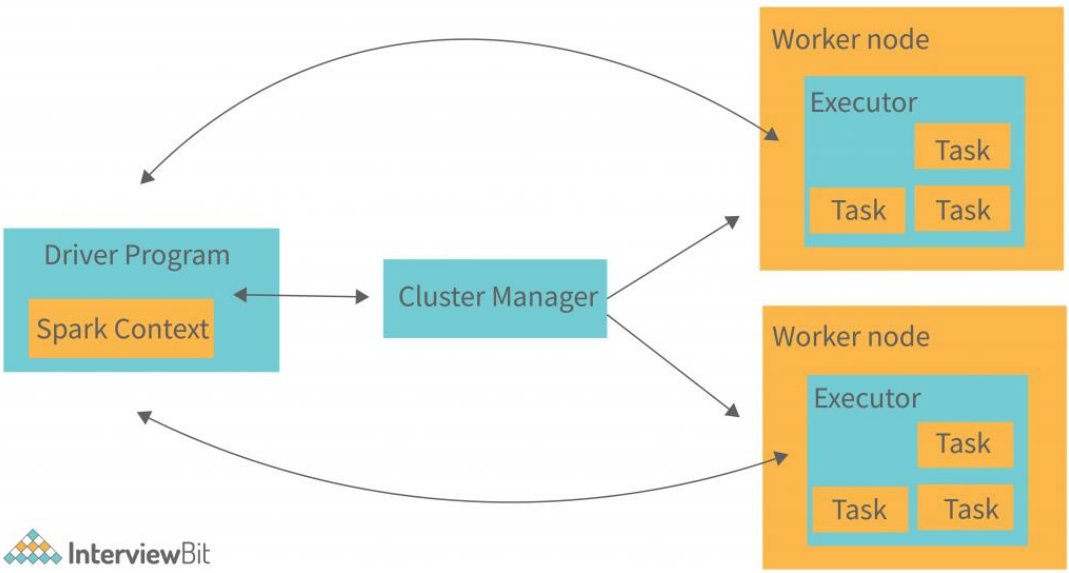
RDD vs DataFrame vs Dataset			
CONTEXT	**RDD**	**DATAFRAME**	**DATASET**
Interoperability	Can be easily converted to DataFrames and vice versa using the `toDF()` and `rdd()` methods.	Can be easily converted to RDDs and Datasets using the `rdd()` and `as[]` methods.	Can be easily converted to DataFrames using the `toDF()` method, and to RDDs using the `rdd()` method.
Performance	Low-level API with more control over the data, but lower-level optimizations compared to DataFrames and Datasets.	Optimized for performance, with high-level API, Catalyst optimizer, and code generation.	Datasets are faster than DataFrames as they take advantage of the JVM's optimization capabilities, such as just-in-time (JIT) compilation, to speed up processing.
Memory Management	Provide full control over memory management, as	Have more optimized memory management,	Support most of the available dataTypes

RDD vs DataFrame vs Dataset			
CONTEXT	**RDD**	**DATAFRAME**	**DATASET**
	they can be cached in memory or disk as per the user's choice.	with a Spark SQL optimizer that helps to reduce memory usage.	
APIs	Provide a low-level API that requires more code to perform transformations and actions on data	Provide a high-level API that makes it easier to perform transformations and actions on data.	Datasets provide a richer set of APIs. Datasets support both functional and object-oriented programming paradigms and provide a more expressive API for working with data
Schema enforcement	Do not have an explicit schema, and are often used for unstructured data.	DataFrames enforce schema at runtime. Have an explicit schema that describes the data and its types.	Datasets enforce schema at compile time. With Datasets, errors in data types or structures are caught earlier in the development cycle. Have an explicit schema that describes the data and its types, and is strongly typed.
Programming Language Support	RDD APIs are available in Java, Scala, Python, and R languages. Hence, this feature provides flexibility to the developers.	Available In 4 languages like Java, Python, Scala, and R.	Only available in Scala and Java.
Optimization	No inbuilt optimization engine is available in RDD.	It uses a catalyst optimizer for optimization.	It includes the concept of a Dataframe Catalyst optimizer for optimizing query plans.
Use Cases	Suitable for low-level data processing and batch jobs that require fine-grained control over data	Suitable for structured and semi-structured data processing with a higher-level of abstraction.	Suitable for high-performance batch and stream processing with strong typing and functional programming.

Spark Working Model:

When the **Driver Program** in the Apache Spark architecture executes, it calls the real program of an application and creates a **SparkContext.**

SparkContext contains all of the basic functions.



The Spark Driver includes several other components, including a DAG Scheduler, Task Scheduler, Backend Scheduler, and Block Manager, all of which are responsible for translating user-written code into jobs that are actually executed on the cluster.

The **Cluster Manager** manages the execution of various jobs in the cluster.

Spark Driver works in conjunction with the Cluster Manager to control the execution of various other jobs.

The cluster Manager does the task of allocating resources for the job. Once the job has been broken down into smaller jobs, which are then distributed to worker nodes, SparkDriver will control the execution.

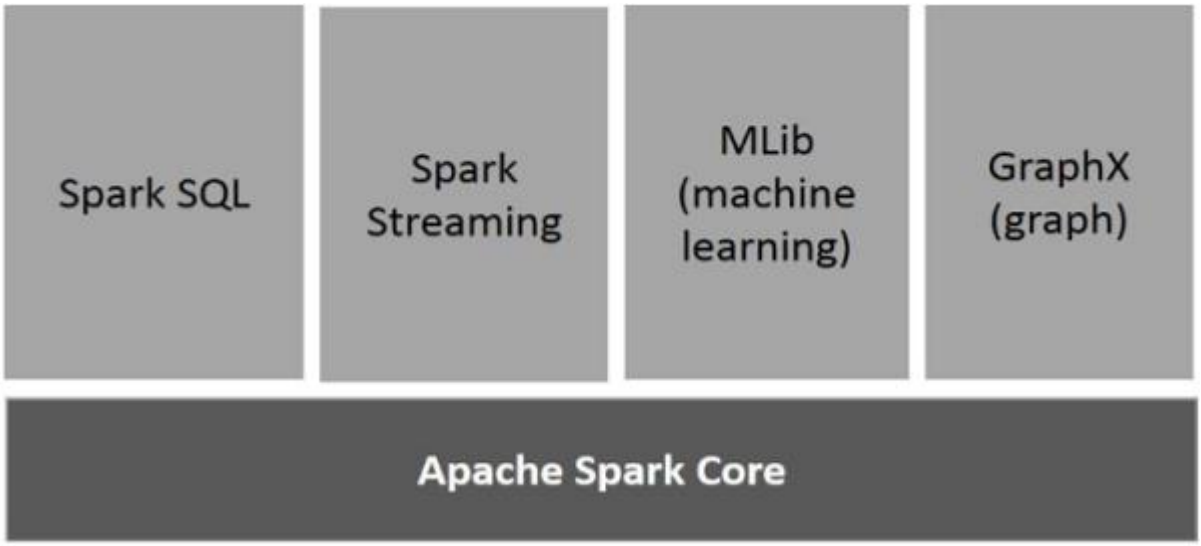
Many **worker nodes** can be used to process an RDD created in the SparkContext, and the results can also be cached.

The Spark Context receives task information from the Cluster Manager and en-queues it on worker nodes.

The **executor** is in charge of carrying out these duties. The lifespan of executors is the same as that of the Spark Application.

We can increase the number of workers if we want to improve the performance of the system. In this way, we can divide jobs into more coherent parts.

Spark Components/Ecosystem:



Apache Spark Core

Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems.

Spark SQL

Spark SQL is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.

Spark Streaming

Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.

MLlib (Machine Learning Library)

MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture. It is, according to benchmarks, done by the MLlib developers against the Alternating Least Squares (ALS) implementations. Spark MLlib is nine times as fast as the Hadoop disk-based version of **Apache Mahout** (before Mahout gained a Spark interface).

GraphX

GraphX is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API. It also provides an optimized runtime for this abstraction.

(Architecture of) Spark Applications:

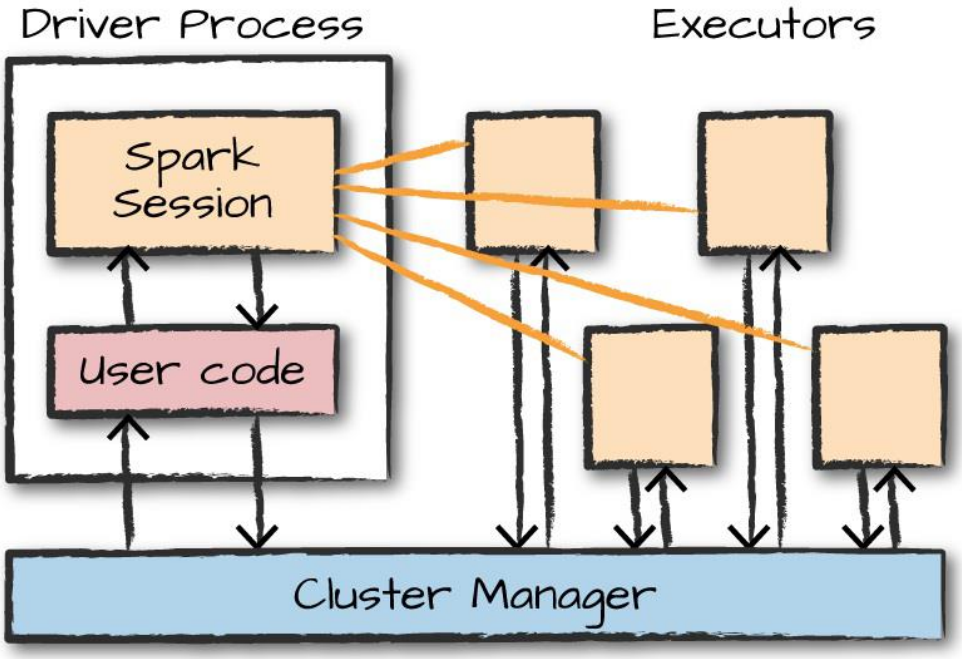
Spark Applications consist of a **driver process** and a set of **executor processes**. The driver process runs your main() function, sits on a node in the cluster.

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 **driver process** is responsible for maintaining information about the Spark Application; responding to a user's program or input; and analysing, distributing, and scheduling work across the executors.

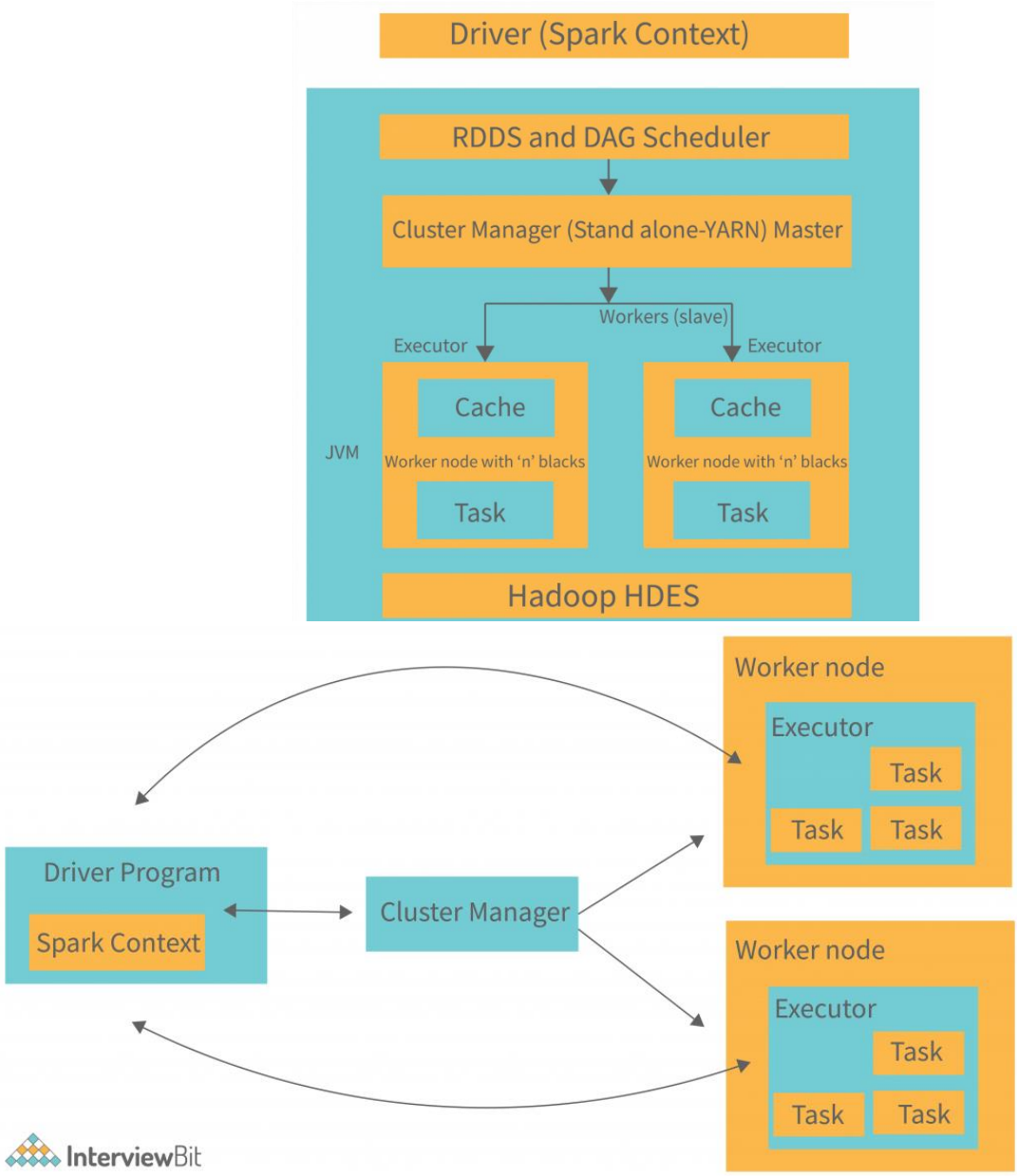
The **executors** are responsible for **actually carrying out the work** that the driver assigns them. This means that each executor is responsible for executing code assigned to it by the driver and reporting the state of the computation on that executor back to the driver node.

The following diagram demonstrates how the cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of three core cluster managers: **Spark's standalone cluster manager** or **YARN**, or **Mesos**. This means that there can be multiple Spark Applications running on a cluster at the same time.



Here are the key points to understand about Spark Applications at this point: Spark employs a **cluster manager** that keeps track of the resources available.

The **driver process** is responsible for executing the driver program's commands across the **executors** to complete a given task. The executors, for the most part, will always be running Spark code. However, the driver can be "driven" from a number of different languages through Spark's language APIs.



Spark driver:

The master node (process) in a driver process coordinates workers and oversees the tasks. Spark is split into jobs and scheduled to be executed on executors in clusters. Spark contexts are created by the driver to monitor the job working in a specific cluster and to connect to a Spark cluster. In the diagram, the driver programmes call the main application and create a spark context that jointly monitors the job working in the cluster and connects to a Spark cluster. Everything is executed using the spark context. Each Spark session has an entry in the Spark context. Spark drivers include more components to execute jobs in clusters, as well as cluster managers. Context acquires worker nodes to execute and store data as Spark clusters are connected to different types of cluster managers. When a process is executed in the cluster, the job is divided into stages and further into scheduled tasks.

Spark executors:

An executor is responsible for executing a job and storing data in a cache at the outset. Executors first register with the driver programme at the beginning. These executors have a number of time slots to run the application concurrently. The executor runs the task when it has loaded data and they are removed in idle mode. The executor runs in the Java process when data is loaded and removed during the execution of the tasks. The executors are allocated dynamically and constantly added and removed during the execution of the tasks. A driver program monitors the executors during their performance. Users' tasks are executed in the Java process.

Cluster Manager:

A cluster manager in Spark is a platform that provides resources to worker nodes as needed. The cluster manager is a daemon that runs on each cluster node. It manages resources, such as CPUs and RAM to launch tasks. The cluster manager also responds to unplanned events, such as recovering from software and hardware failures. The cluster manager's main task is to divide resources across applications. It works as an external service for acquiring resources on the cluster. The cluster manager dispatches work for the cluster. The SparkContext can connect to several types of cluster managers, including Spark's own standalone cluster manager and Mesos.

Worker Nodes:

The slave nodes function as executors, processing tasks, and returning the results back to the spark context. The master node issues tasks to the Spark context and the worker nodes execute them. They make the process simpler by boosting the worker nodes (1 to n) to handle as many jobs as possible in parallel by dividing the job up into sub-jobs on multiple machines. A Spark worker monitors worker nodes to ensure that the computation is performed simply. Each worker node handles one Spark task. In Spark, a partition is a unit of work and is assigned to one executor for each one.

Spark's Language APIs:

Spark is a cluster computing system that provides high-level APIs in Java, Scala, Python, and R.

Developers can write their Spark programs in any of these languages. The freedom of language is one reason why Spark is popular among developers.

Spark also has an interactive Python shell called PySpark. PySpark links the Python API to the Spark core and initializes the Spark context. We can launch PySpark directly from the command line for interactive use.

Spark's language APIs make it possible for us to run Spark code using various programming languages.

For the most part, Spark presents some core "concepts" in every language; these concepts are then translated into Spark code that runs on the cluster of machines.

Scala:
Spark is primarily written in Scala, making it Spark's "default" language.

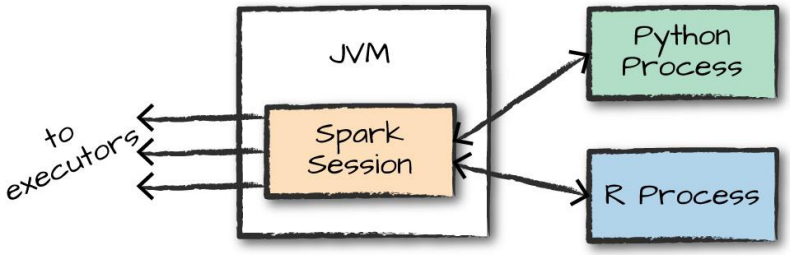
Java:
Even though Spark is written in Scala, Spark's authors have been careful to ensure that you can write Spark code in Java.

Python:
Python supports nearly all constructs that Scala supports.

SQL:
Spark supports a subset of the ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to take advantage of the big data powers of Spark.

R:
Spark has two commonly used R libraries: one as a part of Spark core (SparkR) and another as an R community-driven package (sparklyr).

The following diagram depicts the relationship between the SparkSession and Spark's Language API.



Each language API maintains the same Spark Core concepts. There is a SparkSession object available to the user, which is the entrance point to running Spark code.

When using Spark from Python or R, we don't write explicit JVM instructions; instead, we write Python and R code that Spark translates into code that it then can run on the executor JVMs.

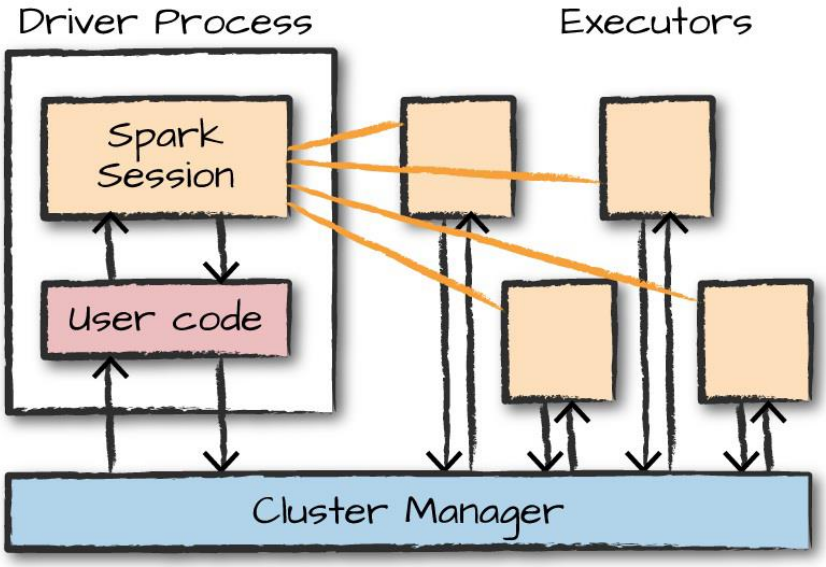
Although we can drive Spark from a variety of languages, what it makes available in those languages is worth mentioning. **Spark has two fundamental sets of APIs: the low-level "unstructured" APIs, and the higher-level "structured" APIs.**

RDD can process both structured as well as unstructured data where as a DataFrame/Dataset organizes the data into row-column format therefore works on structured data.

Starting Spark:

When we start Spark in the interactive mode, we implicitly create SparkSession that manages the Spark Application. When you start it through a standalone application, we must create the SparkSession object ourselves in our application code.

The SparkSession
We control our 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.



Let us create a SparkSession.

In Python we see something like this:

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

Let's now perform the simple task of creating a range of numbers. This range of numbers is just like a named column in a spreadsheet:

In Python

```
myRange = spark.range(1000).toDF("number")
```

We 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.

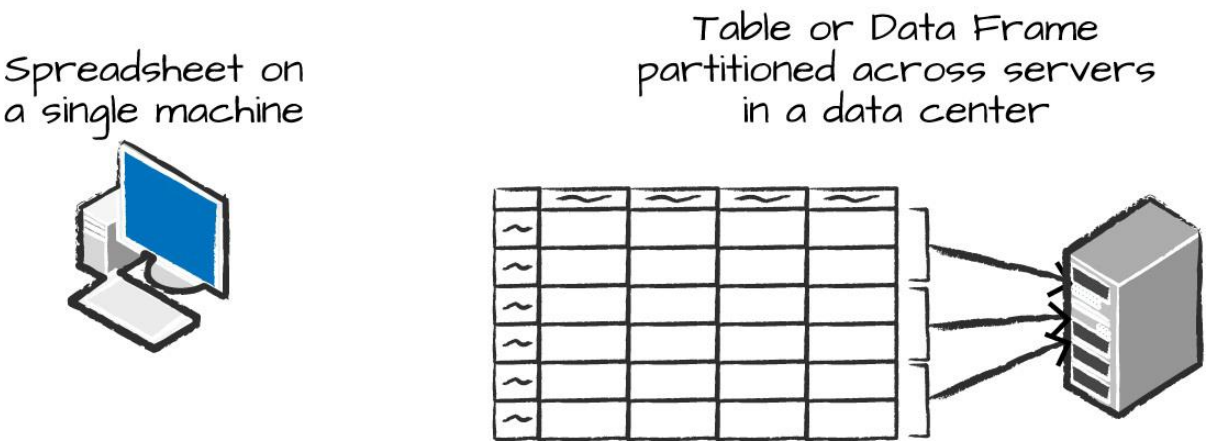
Data Frames:

A DataFrame is the most commonly used 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*.

We can think of a DataFrame as a spreadsheet with named columns.

The following figure illustrates the fundamental difference between a distributed versus single-machine analysis using with spreadsheet on one computer in one specific location, whereas a Spark DataFrame spanning thousands of computers.



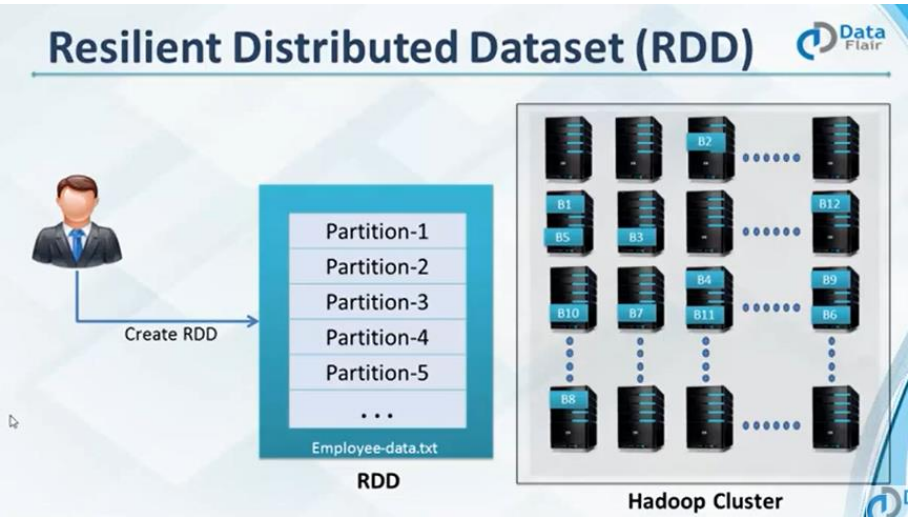
The reason for putting the data on more than one computer should be intuitive: either the data is too large to fit on one machine or it would simply take too long to perform that computation on one machine.

The DataFrame concept is not unique to Spark. R and Python both have similar concepts. However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines. This limits what you can do with a given DataFrame to the resources that exist on that specific machine. However, because Spark has language interfaces for both Python and R, it's quite easy to convert

Pandas (Python) DataFrames to Spark DataFrames, and R DataFrames to Spark DataFrames.

Partitions:
To allow every executor to perform work in parallel, Spark breaks up the data into chunks called partitions.

A partition is a collection of rows that sit on one physical machine in the cluster.



A DataFrame's partitions represent how the data is physically distributed across the cluster of machines during execution. If we have one partition, Spark will have a parallelism of only one, even if you have thousands of executors.

If we have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource.

An important thing to note is that with DataFrames we do not manipulate partitions manually or individually. We simply specify high-level transformations of data in the physical partitions, and Spark determines how this work will actually execute on the cluster. Lower-level APIs do exist (via the RDD interface).

Transformations:

In Spark, the core data structures (RDD, DataFrame, Dataset) 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 RDD/DataFrame/Dataset, we need to instruct Spark how we would like to modify it to do what you want. These instructions are called **transformations.**

Transformations are instructions that tell Spark how to modify a RDD/DataFrame.

Transformations are the core of how you express business logic using Spark.

Here are some examples of transformations in Spark:

- **select( )** and **withColumn( )** for projecting columns
- **filter( )** for filtering
- **orderBy( ), sort( ),** and **sortWithinPartitions( )** for sorting
- **distinct( )** and **dropDuplicates( )** for deduplication
- **join( )** for joining
- **groupBy( )** for aggregations

Let's perform a simple transformation to find all even numbers in our current DataFrame using Python:

```
myRange = spark.range(1000).toDF("number")

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.

Transformations are the core of how you express your business logic using Spark.

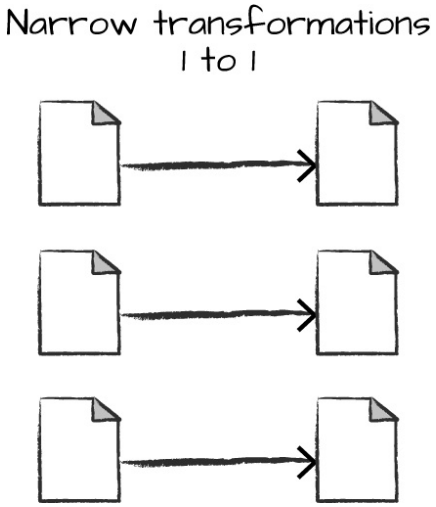
There are **two types of transformations** namely those that specify **narrow dependencies**, and those that specify **wide dependencies**.

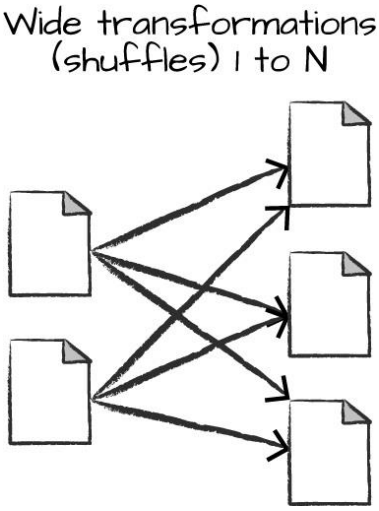
Transformations consisting of **narrow dependencies** (we call them narrow transformations) are those for which **each input partition will contribute to only one output partition.**

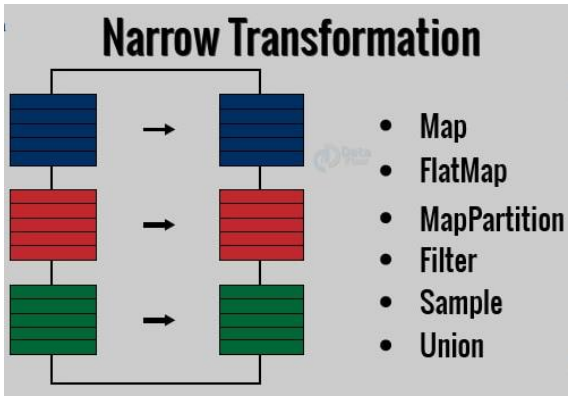
In the preceding code snippet, the ***where*** statement specifies a **narrow dependency,** where only one partition contributes to at most one output partition.

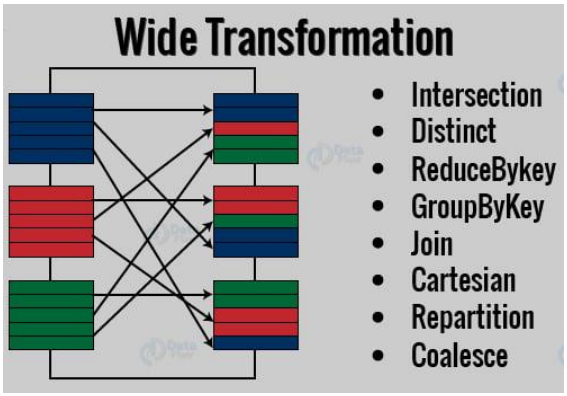
A **wide dependency** (or wide transformation) style transformation will have **input partitions contributing to many output partitions**. We will often hear this referred to as a **shuffle** whereby Spark will exchange partitions across the cluster.

With narrow transformations, Spark will automatically perform an operation called pipelining, meaning that if we specify multiple filters on DataFrames, they will all be performed in-memory.









Lazy Evaluation

Lazy evaluation means that Spark will wait until the very last moment to execute the graph of computation instructions.

We can apply as many **TRANSFORMATIONs** as we want, but Spark will not start the execution of the process until an **ACTION** is called.

In Spark, instead of modifying the data immediately when you express some operation, you build up a *plan* of transformations that you would like to apply to your source data.

By waiting until the last minute to execute the code, Spark compiles this plan from your raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster.

This provides immense benefits because Spark can optimize the entire data flow from end to end.

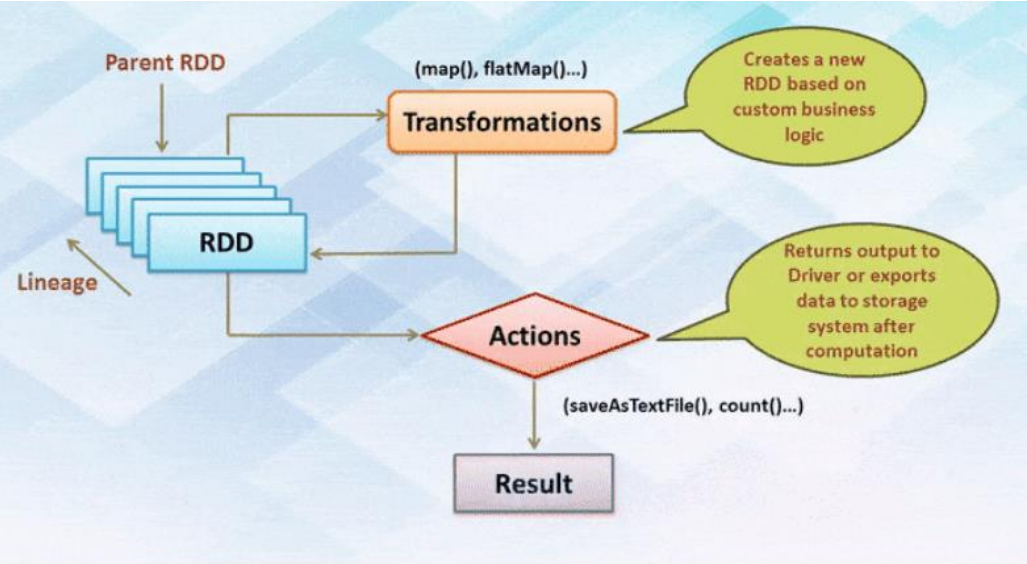
Lazy evaluation in Spark is a feature that improves the efficiency and performance of Spark. It delays the execution of transformations on distributed datasets until an action is called. Once an Action is called, Spark starts looking at all the transformations and creates a DAG.

When a transformation is called, Spark adds it to a DAG of computation. (The DAG is a sequence of operations that need to be performed to get the resultant output. When the driver requests some data, the DAG is executed.)

Lazy evaluation helps to:

- Apply various optimizations
- Pipeline computations
- Minimize unnecessary computations and data movement

Lazy evaluation can save time and unwanted processing power. For example, instead of loading the entire 1GB file, only the first line of the file is loaded and printed.



Apache Spark Lazy Evaluation Explanation.

Actions:

Transformations only **create RDDs** from each other.

When we want to work with the actual dataset of RDD then action is required to be performed.

When the action is triggered to get the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values.

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.

Actions return a result or write to the disc.

Ex: The simplest action is count, which gives us the total number of records in the DataFrame:

```
myRange = spark.range(1000).toDF("number")

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

divisBy2.count()
```

The output of the preceding code should be 500.

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.

We can see all of this by inspecting the Spark UI, a tool included in Spark with which we can monitor the Spark jobs running on a cluster. We can monitor the progress of a job through the Spark web UI. The Spark UI is available on port 4040 of the driver node.

If we are running in local mode, this will be http://localhost:4040. The Spark UI displays information on the state of your Spark jobs, its environment, and cluster state.

Spark UI

Hostname: ec2-35-167-29-186.us-west-2.compute.amazonaws.com Spark Version: 2.1.0

Jobs Stages Storage Environment Executors SQL JDBC/ODBC Server

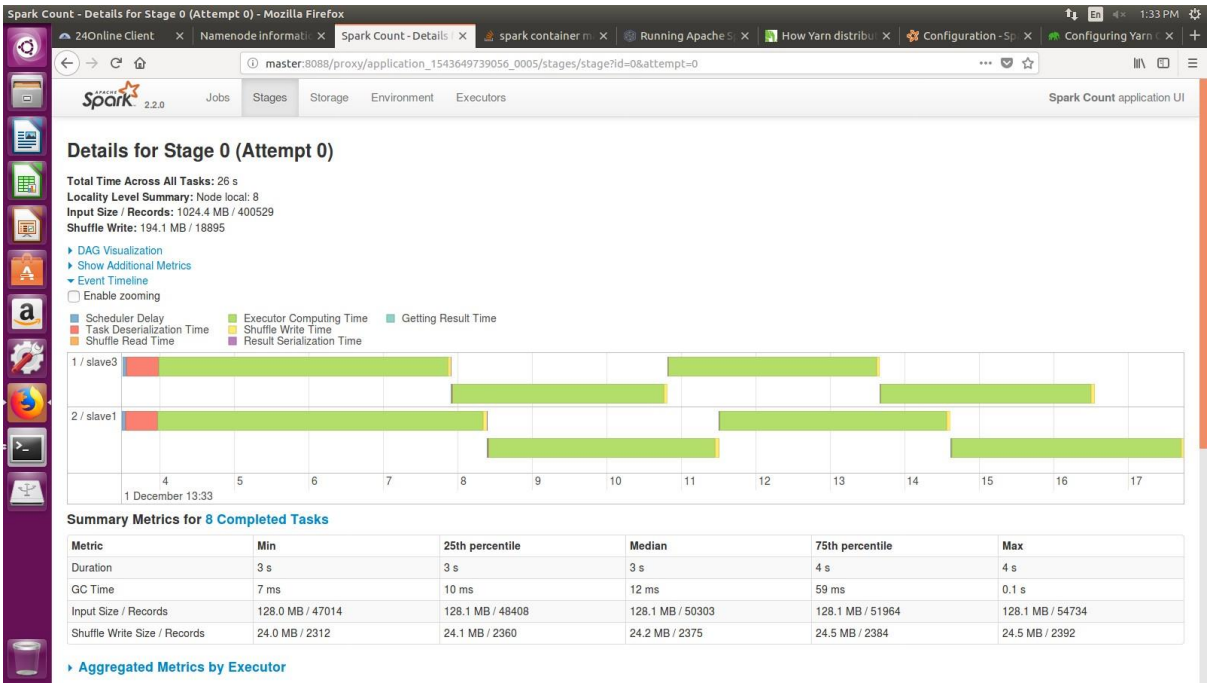
Spark Jobs (?)

User: root
Total Uptime: 39 min
Scheduling Mode: FAIR
Completed Jobs: 2

Event Timeline

Completed Jobs (2)

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1 (360049305052286855 2_5147566918362167263_1b1c589736794803a82581288fa2d915)	divisBy2.count() count at NativeMethodAccessorImpl.java:0	2017/01/19 17:22:51	91 ms	2/2	9/9
0 (442095639162785772_5532783187248264704_ab36733a32cf4803ac65a3ca545110be)	divisBy2.count() count at <console>:33	2017/01/19 17:22:50	0.8 s	2/2	9/9



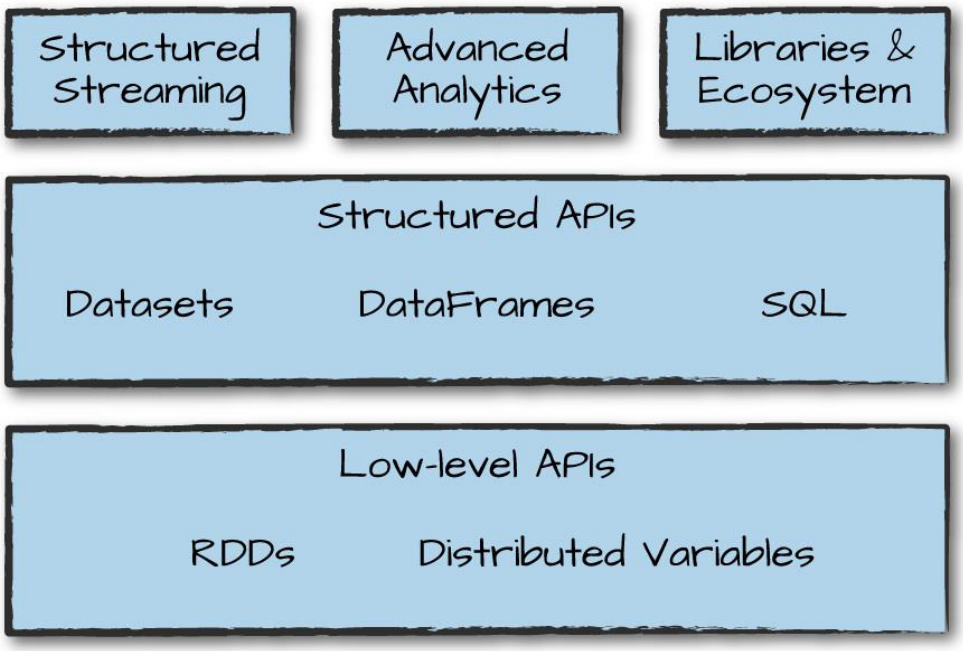
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	3 s	3 s	3 s	4 s	4 s
GC Time	7 ms	10 ms	12 ms	59 ms	0.1 s
Input Size / Records	128.0 MB / 47014	128.1 MB / 48408	128.1 MB / 50303	128.1 MB / 51964	128.1 MB / 54734
Shuffle Write Size / Records	24.0 MB / 2312	24.1 MB / 2360	24.2 MB / 2375	24.5 MB / 2384	24.5 MB / 2392

Spark's Toolset:

We learnt Spark's core concepts, like transformations and actions, in the context of Spark's Structured APIs.

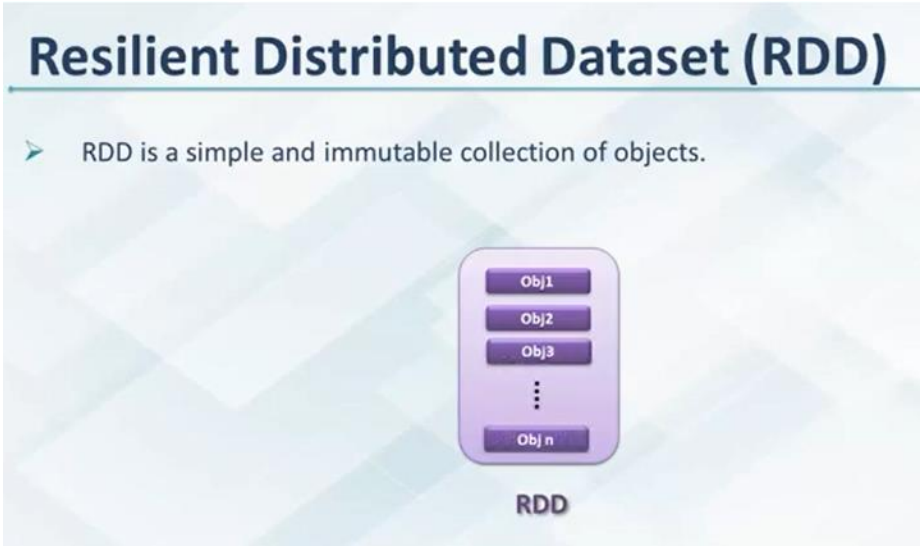
These simple conceptual building blocks are the foundation of Apache Spark's vast ecosystem of tools and libraries shown in the below diagram.

Spark is composed of these primitives—the lower-level APIs and the Structured APIs—and then a series of standard libraries for additional functionality.



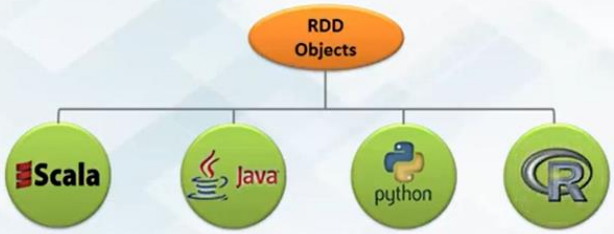
Spark's libraries support a variety of different tasks, from graph analysis and machine learning to streaming and integrations with a host of computing and storage systems.

RDD(Resilient Distributed Dataset):



Resilient Distributed Dataset (RDD)

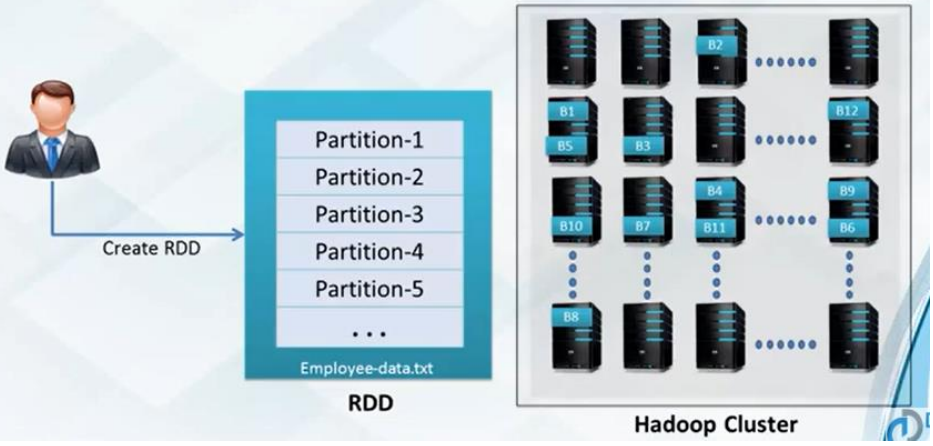
- RDD is a simple and immutable collection of objects.
- RDD can contain any type of (scala, java, python and R) objects.



Resilient Distributed Dataset (RDD)

- RDD is a simple and immutable collection of objects.
- RDD can contain any type of (scala, java, python and R) objects.
- Each RDD is split-up into different partitions, which may be computed on different nodes of clusters.

Resilient Distributed Dataset (RDD)



DataFrames:
DataFrames are a distributed collection of data organized into named columns. DataFrames are similar to tables in relational databases.

It is an immutable distributed collection of data. DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction.

DataFrames provide a higher-level API that is optimized for performance and easier to work with for structured data.

A DataFrame is the most commonly used 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*.

We can think of a DataFrame as a spreadsheet with named columns.

Can be easily converted to RDDs and Datasets using the **rdd()** and **as[]** methods.

Datasets:
Datasets are **type-safe** version of Spark's structured API, for writing statically typed code in **Java and Scala**. The Dataset API is not available in Python and R, because those languages are dynamically typed.

Recall that DataFrames, are a distributed collection of **objects of type Row** that can hold various types of tabular data.

The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java ArrayList or Scala Seq.

The APIs available on Datasets are *type-safe*, meaning that you cannot accidentally view the objects in a Dataset as being of another class than the class you put in initially. This makes Datasets especially attractive for writing large applications, with which multiple software engineers must interact through well-defined interfaces.

The Dataset class is parameterized with the type of object contained inside. For example, a Dataset[Person] will be guaranteed to contain objects of class Person.

One great thing about Datasets is that we can use them only when we need or want to. After we have performed our manipulations, Spark can automatically turn it back into a DataFrame, and we can manipulate it further by using the hundreds of functions that Spark includes.

Structured Streaming:
Structured Streaming is a **high-level API** for stream processing.

Spark Structured Streaming is a **stream processing** engine built on Spark SQL that processes data incrementally and updates the final results as more streaming data arrives. It brought a lot of ideas from other structured APIs in Spark (Dataframe and Dataset) and offered query optimizations similar to SparkSQL.

With Structured Streaming, we can take the same operations that we perform in **batch mode** using Spark's structured APIs and **run them in a streaming fashion**.

This can reduce latency and allow for incremental processing.

The best thing about Structured Streaming is that it allows us to **rapidly and quickly extract** value out of streaming systems with virtually no code changes. It also makes it easy to conceptualize because we can write our batch job as a way

to prototype it and then we can convert it to a streaming job. The way all of this works is by incrementally processing that data.

It is **consistent** because it provides a guarantee that application output will be equivalent to the execution of a batch job running on the input data processes at any time. Additionally, Spark Structured Streaming is **fault-tolerant** and manages interactions with the data source and sink (output), with semantics to handle out-of-order data events as well. For example, Spark will update results based on the received data if a data point is received late, you can filter and discard delayed data. The API is straightforward to use and has many similarities to SQL.

The Spark Streaming application has **three major components**: **source** (input), **processing engine** (business logic), and **sink** (output). Input sources are where the application receives the data, and these can be Kafka, Kinesis, HDFS, etc. The processing or streaming engine runs the actual business logic on the data coming from various sources. Finally, the sink stores the outcome of the processed data, which can be an HDFS, a relational database, etc.

Overview of Structured API Execution:

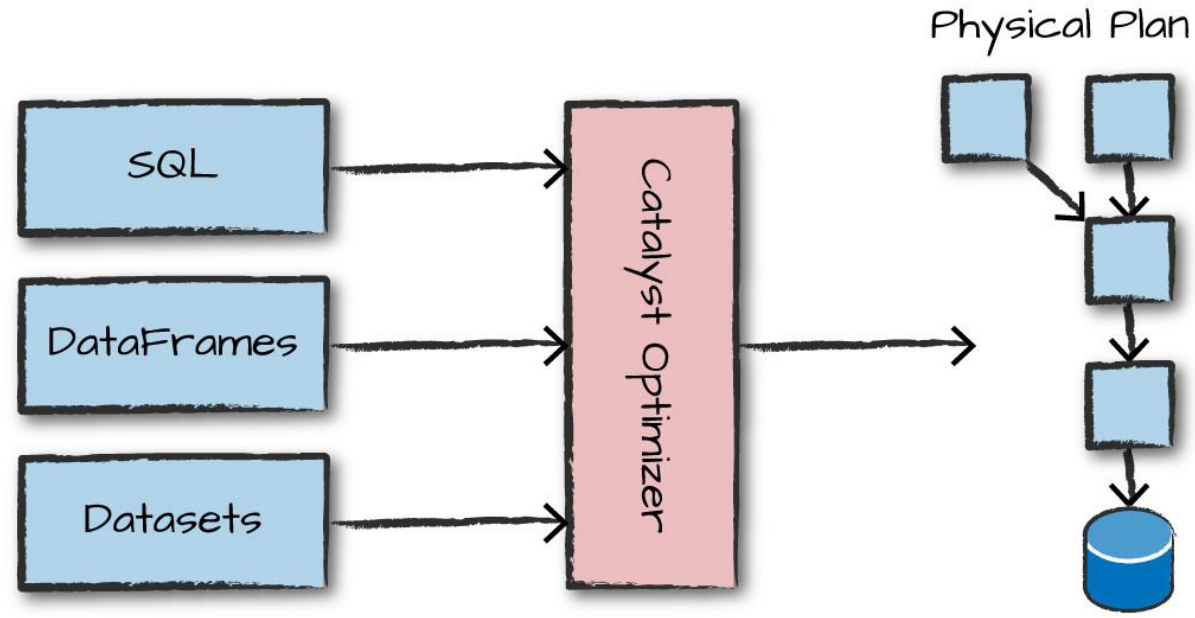
The Structured APIs are a tool for manipulating all sorts of data, from unstructured log files to semi-structured CSV files. These APIs refer to two core types of distributed collection APIs:

- DataFrames
- Datasets

The majority of the Structured APIs apply to both ***batch* and *streaming*** computation.

This means that when you work with the Structured APIs, it should be simple to migrate from batch to streaming (or vice versa) with little to no effort.

Let us walk through the execution of a single structured API query from user code to executed code.



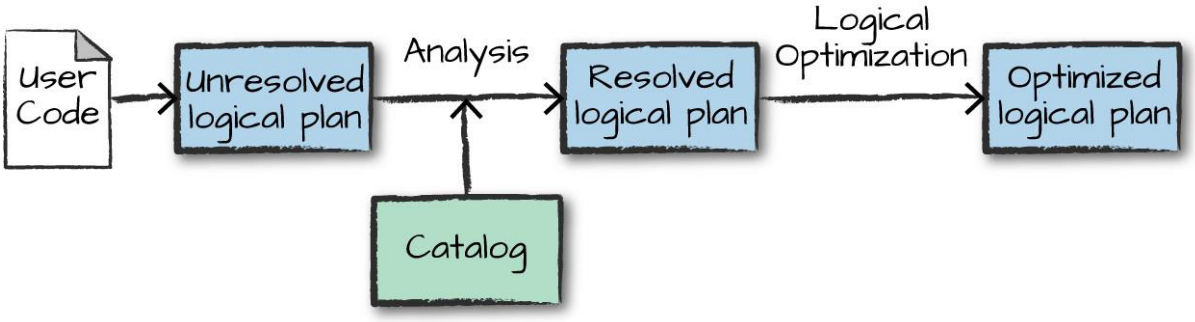
Here is an overview of the steps involved in this execution:

1. ***Write DataFrame/Dataset***
2. ***If code is valid, 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.***

The written 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 above figure shows the process.

Logical Planning

The first phase of execution is meant to take user code and convert it into a logical plan. The following figure illustrates this process.

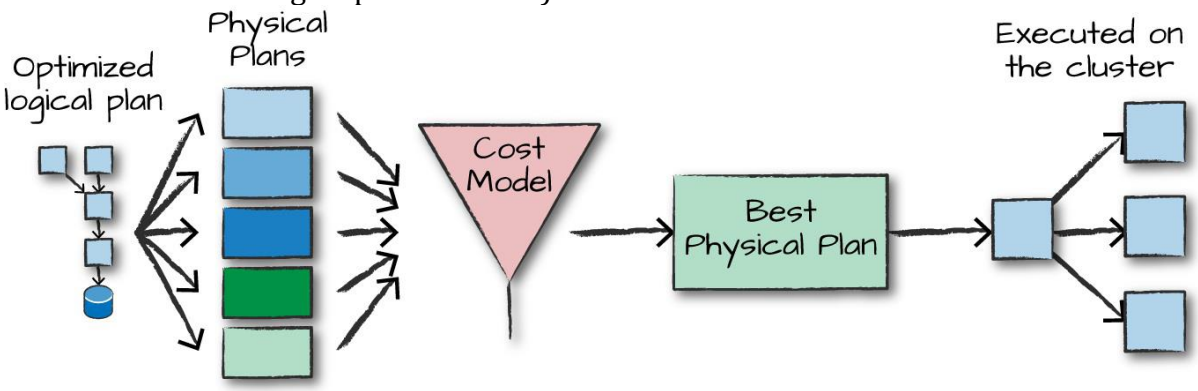


This logical plan only represents a set of abstract transformations that do not refer to executors or drivers and it is purely to convert the user's set of expressions into the most optimized version. It does this by converting user code into an unresolved logical plan. This plan is unresolved because although our code might be valid, the tables or columns that it refers to might or might not exist. Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer. The analyzer might reject the unresolved

logical plan if the required table or column name does not exist in the catalog. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections. Packages can extend the Catalyst to include their own rules for domain-specific optimizations.

Physical Planning

After successfully creating an optimized logical plan, Spark then begins the physical planning process. The physical plan, often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model, as depicted in the below figure. An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are).



Physical planning results in a series of RDDs and transformations. This result is why you might have heard Spark referred to as a compiler—it takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations for you.

Execution

Upon selecting a physical plan, Spark runs all of this code over RDDs, the lower-level programming interface of Spark (which we cover in Part III). Spark performs further optimizations at runtime, generating native Java bytecode that can remove entire tasks or stages during execution. Finally the result is returned to the user.