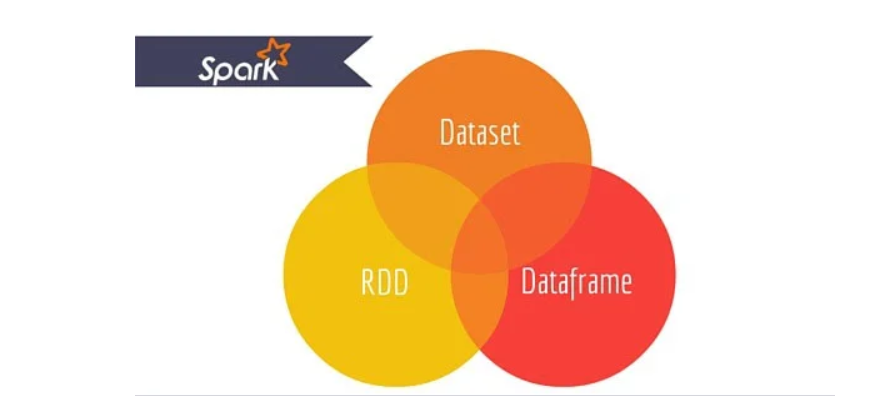
**Difference between DataSet API and DataFrame API**



**RDD -** The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.

## RDD Features:-

* **Distributed collection:**  
  RDD uses MapReduce operations which is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.
* **Immutable:** RDDs composed of a collection of records which are partitioned. A partition is a basic unit of parallelism in an RDD, and each partition is one logical division of data which is immutable and created through some transformations on existing partitions.Immutability helps to achieve consistency in computations.
* **Fault tolerant:** In a case of we lose some partition of RDD , we can replay the transformation on that partition in lineage to achieve the same computation, rather than doing data replication across multiple nodes.This characteristic is the biggest benefit of RDD because it saves a lot of efforts in data management and replication and thus achieves faster computations.
* **Lazy evaluations:** All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset . The transformations are only computed when an action requires a result to be returned to the driver program.
* **Functional transformations:** RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset.
* **Data processing formats:**  
  It can easily and efficiently process data which is structured as well as unstructured data.
* **Programming Languages supported:**  
  RDD API is available in Java, Scala, Python and R.

## RDD Limitations:-

* **No inbuilt optimization engine:** When working with structured data, RDDs cannot take advantages of Spark’s advanced optimizers including catalyst optimizer and Tungsten execution engine. Developers need to optimize each RDD based on its attributes.
* **Handling structured data:** Unlike Dataframe and datasets, RDDs don’t infer the schema of the ingested data and requires the user to specify it.

## Dataframes

Spark introduced Dataframes in Spark 1.3 release. Dataframe overcomes the key challenges that RDDs had.

A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a R/Python Dataframe. Along with Dataframe, Spark also introduced catalyst optimizer, which leverages advanced programming features to build an extensible query optimizer.

## Dataframe Features:-

* **Distributed collection of Row Object:** A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database, but with richer optimizations under the hood.
* **Data Processing:** Processing structured and unstructured data formats (Avro, CSV, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, MySQL, etc). It can read and write from all these various datasources.
* **Optimization using catalyst optimizer:** It powers both SQL queries and the DataFrame API. Dataframe use catalyst tree transformation framework in four phases,
* 1.Analyzing a logical plan to resolve references
* 2.Logical plan optimization
* 3.Physical planning
* 4.Code generation to compile parts of the query to Java bytecode.
* **Hive Compatibility:** Using Spark SQL, you can run unmodified Hive queries on your existing Hive warehouses. It reuses Hive frontend and MetaStore and gives you full compatibility with existing Hive data, queries, and UDFs.
* **Tungsten:** Tungsten provides a physical execution backend whichexplicitly manages memory and dynamically generates bytecode for expression evaluation.
* **Programming Languages supported:**  
  Dataframe API is available in Java, Scala, Python, and R.

## Dataframe Limitations:-

* **Compile-time type safety:** As discussed, Dataframe API does not support compile time safety which limits you from manipulating data when the structure is not know. The following example works during compile time. However, you will get a Runtime exception when executing this code.

Example:

case class Person(name : String , age : Int)

val dataframe = sqlContext.read.json("people.json")

dataframe.filter("salary > 10000").show

=> throws Exception : cannot resolve 'salary' given input age , name

This is challenging specially when you are working with several transformation and aggregation steps.

* **Cannot operate on domain Object (lost domain object):** Once you have transformed a domain object into dataframe, you cannot regenerate it from it. In the following example, once we have create personDF from personRDD, we won’t be recover the original RDD of Person class (RDD[Person]).

Example:

case class Person(name : String , age : Int)

val personRDD = sc.makeRDD(Seq(Person("A",10),Person("B",20)))

val personDF = sqlContext.createDataframe(personRDD)

personDF.rdd // returns RDD[Row] , does not returns RDD[Person]

## Datasets API

Dataset API is an extension to DataFrames that provides a type-safe, object-oriented programming interface. It is a strongly-typed, immutable collection of objects that are mapped to a relational schema.

At the core of the Dataset, API is a new concept called an encoder, which is responsible for converting between JVM objects and tabular representation. The tabular representation is stored using Spark internal Tungsten binary format, allowing for operations on serialized data and improved memory utilization. Spark 1.6 comes with support for automatically generating encoders for a wide variety of types, including primitive types (e.g. String, Integer, Long), Scala case classes, and Java Beans.

## Dataset Features:-

* **Provides best of both RDD and Dataframe:** RDD(functional programming, type safe), DataFrame (relational model, Query optimazation , Tungsten execution, sorting and shuffling)
* **Encoders:** With the use of Encoders, it is easy to convert any JVM object into a Dataset, allowing users to work with both structured and unstructured data unlike Dataframe.
* **Programming Languages supported:** Datasets API is currently only available in Scala and Java. Python and R are currently not supported in version 1.6. Python support is slated for version 2.0.
* **Type Safety:** Datasets API provides compile time safety which was not available in Dataframes. In the example below, we can see how Dataset can operate on domain objects with compile lambda functions.

Example:

case class Person(name : String , age : Int)

val personRDD = sc.makeRDD(Seq(Person("A",10),Person("B",20)))

val personDF = sqlContext.createDataframe(personRDD)

val ds:Dataset[Person] = personDF.as[Person]

ds.filter(p => p.age > 25)

ds.filter(p => p.salary > 25)

// error : value salary is not a member of person

ds.rdd // returns RDD[Person]

* **Interoperable:** Datasets allows you to easily convert your existing RDDs and Dataframes into datasets without boilerplate code.

## Datasets API Limitation:-

* **Requires type casting to String:** Querying the data from datasets currently requires us to specify the fields in the class as a string. Once we have queried the data, we are forced to cast column to the required data type. On the other hand, if we use map operation on Datasets, it will not use Catalyst optimizer.

Example:

ds.select(col("name").as[String], $"age".as[Int]).collect()

No support for Python and R: As of release 1.6, Datasets only support Scala and Java. Python support will be introduced in Spark 2.0.

The Datasets API brings in several advantages over the existing RDD and Dataframe API with better type safety and functional programming.With the challenge of type casting requirements in the API, you would still not the required type safety and will make your code brittle

Because DataFrame is weakly typed and developers aren’t getting the benefits of the type system. For example, lets say you want to read something from SQL and run some aggregation on it:

val people = sqlContext.read.parquet("...")

val department = sqlContext.read.parquet("...")

people.filter("age > 30")

.join(department, people("deptId") === department("id"))

.groupBy(department("name"), "gender")

.agg(avg(people("salary")), max(people("age")))

When you say people("deptId"), you’re not getting back an Int, or a Long, you’re getting back a Column object which you need to operate on. In languages with a rich type systems such as Scala, you end up losing all the type safety which increases the number of run-time errors for things that could be discovered at compile time.

On the contrary, DataSet[T] is typed. when you do:

val people: People = val people = sqlContext.read.parquet("...").as[People]

You’re actually getting back a People object, where deptId is an actual integral type and not a column type, thus taking advantage of the type system.

As of Spark 2.0, the DataFrame and DataSet APIs will be unified, where DataFrame will be a type alias for DataSet[Row].