Spark Modules

- **Spark Core :-** heart of Spark and is responsible for management functions such as task scheduling.
    - Spark Core implements and depends upon a programming abstraction known as Resilient Distributed Datasets (RDDs).

- **Spark SQL :-** for working with structured data, and it is designed to support workloads that combine SQL database queries with more complicated, algorithm-based analytics.
    - supports Hive and its HiveQL query syntax.
    - supports JDBC and ODBC connections, enabling integration with existing databases, data warehouses and BI tools.

- **Spark Streaming :-** supports scalable and fault-tolerant processing of streaming data and can integrate with established sources of data streams like Flume (optimized for data logs) and Kafka (optimized for distributed messaging).
    - Spark Streaming's design, and its use of Spark's RDD abstraction, are meant to ensure that applications written for streaming data can be repurposed to analyze batches of historical data with little modification.

- **Spark Mlib :-** scalable machine learning library, which implements a set of commonly used machine learning and statistical algorithms.
    - These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.

- **Spark GraphX**: **:-** supports analysis of and computation over graphs of data and supports a version of graph processing's Pregel API.
    - GraphX includes a number of widely understood graph algorithms, including PageRank.

- **Spark R :-** This module was added to the 1.4.x release of Apache Spark, providing data scientists and statisticians using R with a lightweight mechanism for calling upon Spark's capabilities.

Spark SQL

One of the cool features of the Spark SQL module is the ability to execute SQL queries to perform data processing and the result of the queries will be returned as a Dataset or DataFrame.

DataFrames API

- DataFrames offer:
    - Automatic code optimizations
    - Ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster
    - Support for a wide array of data formats and storage systems

    - State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
    - Seamless integration with all big data tooling and infrastructure via Spark APIs for Python, Java, Scala, and R
- This extended API eases application development, while helping to improve performance via the optimizations and code generation.

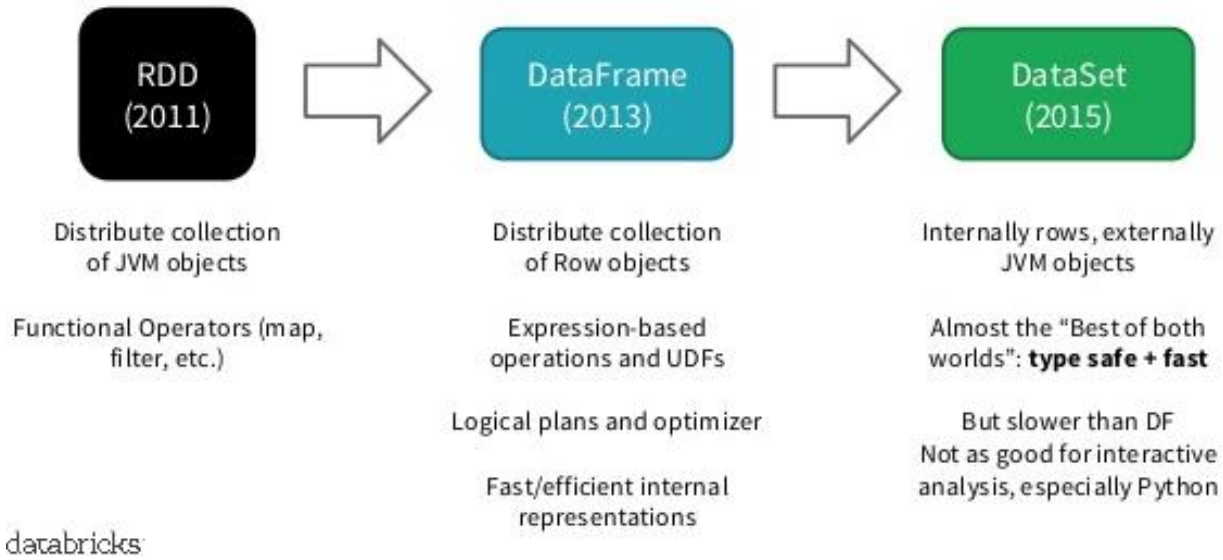
DataSets API

- DataSets API is an extension to DataFrames.
- A Dataset is a strongly typed, immutable collection of data. Similar to a DataFrame, the data in a Dataset is mapped to a defined schema. It is more about type safety and is object-oriented.

There are a few important differences between a DataFrame and a Dataset.

- Each row in a Dataset is represented by a user-defined object so that you can refer to an individual column as a member variable of that object. This provides you with compile-type safety.
- A Dataset has helpers called encoders, which are smart and efficient encoding utilities that convert data inside each user-defined object into a compact binary format. This translates into a reduction of memory usage if and when a Dataset is cached in memory as well as a reduction in the number of bytes that Spark needs to transfer over a network during the shuffling process.

History of Spark APIs



Additional Transformations and Actions

Where	Function	Description
SparkContext	doubleRDDToDoubleRDDFunctions	Extra functions available on RDDs of Doubles
SparkContext	numericRDDToDoubleRDDFunctions	Extra functions available on RDDs of Doubles
SparkContext	rddToPairRDDFunctions	Extra functions available on RDDs of (key, value) pairs
SparkContext	hadoopFile()	Get an RDD for a Hadoop file with an arbitrary InputFormat
SparkContext	hadoopRDD()	Get an RDD for a Hadoop file with an arbitrary InputFormat
SparkContext	makeRDD()	Distribute a local Scala collection to form an RDD
SparkContext	parallelize()	Distribute a local Scala collection to form an RDD
SparkContext	textFile()	Read a text file from a file system URI
SparkContext	wholeTextFiles()	Read a directory of text files from a file system URI

Broadcast Variables, Accumulators

➢ **How can you minimize data transfers when working with Spark?**

- Minimizing data transfers and avoiding shuffling helps write spark programs that run in a fast and reliable manner. The various ways in which data transfers can be minimized when working with Apache Spark are:
    - Using Broadcast Variable- Broadcast variable enhances the efficiency of joins between small and large RDDs.
    - Using Accumulators – Accumulators help update the values of variables in parallel while executing.
- The most common way is to avoid operations ByKey, repartition or any other operations which trigger shuffles.

➢ **Why is there a need for broadcast variables when working with Apache Spark?**

- These are read only variables, present in-memory cache on every machine. When working with Spark, usage of broadcast variables eliminates the necessity to ship copies of a variable for every task, so data can be processed faster. Broadcast variables help in storing a lookup table inside the memory which enhances the retrieval efficiency when compared to an RDD lookup ().

Is Spark better than Hadoop?

- Spark is not, despite the hype, a replacement for Hadoop. Nor is MapReduce dead.
- Spark can run on top of Hadoop, benefiting from Hadoop's cluster manager (YARN) and underlying storage (HDFS, HBase, etc.).
- Spark can also run completely separately from Hadoop, integrating with alternative cluster managers like Mesos and alternative storage platforms like Cassandra and Amazon S3.
- Much of the confusion around Spark's relationship to Hadoop dates back to the early years of Spark's development. At that time, Hadoop relied upon MapReduce for the bulk of its data processing.
- MapReduce also managed scheduling and task allocation processes within the cluster; even workloads that were not best suited to batch processing were passed through MapReduce engine, adding complexity and reducing performance.
- Hadoop has come a long way since its early versions which only performed batch processing of MapReduce jobs on large volumes of data stored in HDFS.
- With the development of YARN cluster manager, Hadoop is not completely dependent upon MapReduce. Hadoop is now better able to manage a wide range of data processing tasks, from batch processing to streaming data and graph analysis.
- Other data processing tasks can be assigned to different processing engines (including Spark), with YARN handling the management and allocation of cluster resources.
- MapReduce is still best for running static batch processes. Spark is a viable alternative to MapReduce in a range of other circumstances.
- Spark is not a replacement for Hadoop, but is instead a great companion to a modern Hadoop cluster deployment.

What Hadoop Gives Spark?

- Apache Spark is often deployed in conjunction with a Hadoop cluster, and Spark gets benefitted from a number of Hadoop capabilities as a result.
    - **YARN resource manager**, which takes responsibility for scheduling tasks across available nodes in the cluster;
    - **Distributed File System**, which stores data when the cluster runs out of free memory, and which persistently stores historical data when Spark is not running;
    - **Disaster Recovery capabilities**, inherent to Hadoop, which enable recovery of data when individual nodes fail.

- **Data Security**, as Spark tackles production workloads in regulated industries such as healthcare and financial services, projects like Apache Knox and Apache Ranger offer data security capabilities that augment Hadoop.
- **A distributed data platform**, Spark jobs can be deployed on available resources anywhere in a distributed cluster, without the need to manually allocate and track those individual jobs.

What Spark Gives Hadoop?

- Spark is able to contribute, via YARN, to Hadoop-based jobs. In particular, Spark's machine learning module delivers capabilities not easily exploited in Hadoop without the use of Spark.
- Spark's original design goal, to enable rapid in-memory processing of sizeable data volumes, also remains an important contribution to the capabilities of a Hadoop cluster.
- In certain circumstances, Spark's SQL capabilities, streaming capabilities (otherwise available to Hadoop through Storm, for example), and graph processing capabilities (otherwise available to Hadoop through Neo4J or Giraph) may also prove to be of value in enterprise use cases.