[Spark Preliminaries](https://play.fresco.me/course/178)

Getting to Know Spark

Welcome to the course on **Spark Preliminaries**. Here, you will learn about:

* Spark and its ecosystem
* The need for Spark and its applications
* Spark Data Model and Runtime Architecture
* Storage and Caching Concepts

And followed by an assessment.

What is Spark?



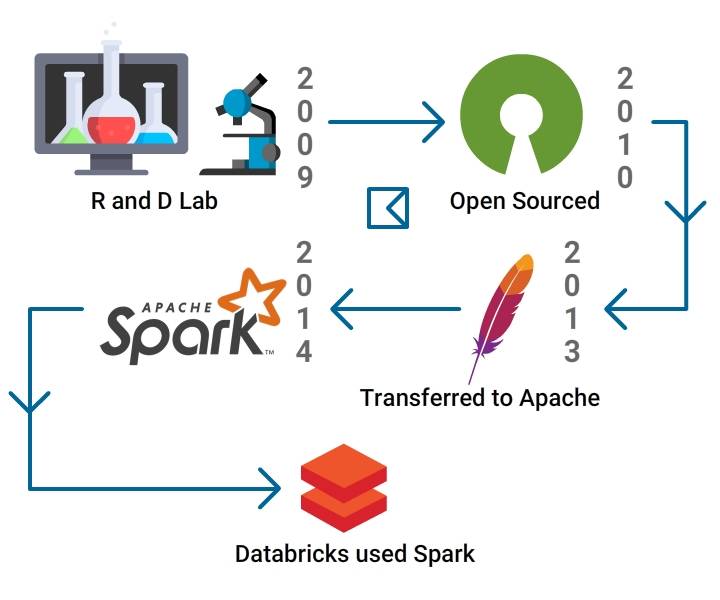
According to [Wikipedia](https://en.wikipedia.org/wiki/Apache_Spark),

**Apache Spark** is an open source cluster computing framework that provides an interface for entire programming clusters with implicit data parallelism and fault-tolerance.

Apache Spark is devised to serve as a **general-purpose** and **fast cluster computing platform**.

* Spark runs **computations in memory** and provides a quicker system for complex applications operating on disk.
* Spark covers various **workloads needing a dedicated distributed systems** namely streaming, interactive queries, iterative algorithms, and batch applications.

Brief History



* Spark was founded by Matei Zaharia at AMPLab in UC Berkeley in 2009. It was later open-sourced under a BSD license in 2010.
* The Spark project was donated to the Apache Software Foundation in 2013 and then licensed to Apache 2.0.
* Spark was recognized as a Top-Level Apache Project in February 2014.
* M. Zaharia's company Databricks created a

new world record using Spark in large scale sorting.

* Spark 2 was launched in June 2016.

##### Key Features

**Performance**:

* Faster than Hadoop MapReduce up to 10x (on disk) - 100x (In-Memory)
* Caches datasets in memory for interactive data analysis
* In Spark, tasks are threads, while in Hadoop, a task generates a separate JVM.

**Rich APIs and Libraries**

* Offers a deep set of high-level APIs for languages such as R, Python, Scala, and Java.
* Very less code than Hadoop MapReduce program because it uses functional programming constructs.

**Scalability and Fault Tolerant**

* Scalable above 8000 nodes in production.
* Utilizes Resilient Distributed Datasets (RDDs) a logical collection of data partitioned across machines, which produces an intelligent fault tolerance mechanism.

Key Features...

**Supports HDFS**

* Integrated with Hadoop and its ecosystem
* It can read existing data.

**Realtime Streaming**

* Supports streams from a variety of data sources like Twitter, Kinesis, Flume, and Kafka.
* We defined a high-level library for stream processing, utilizing Spark Streaming.

**Interactive Shell**

* Provides an Interactive command line interface (in Python or Scala) for horizontally scalable, low-latency, data exploration.
* Supports structured and relational query processing (SQL), via Spark SQL.

**Machine Learning**

* Higher level libraries for graph processing and machine learning.
* Various machine learning algorithms such as pattern-mining, clustering, recommendation, and classification.

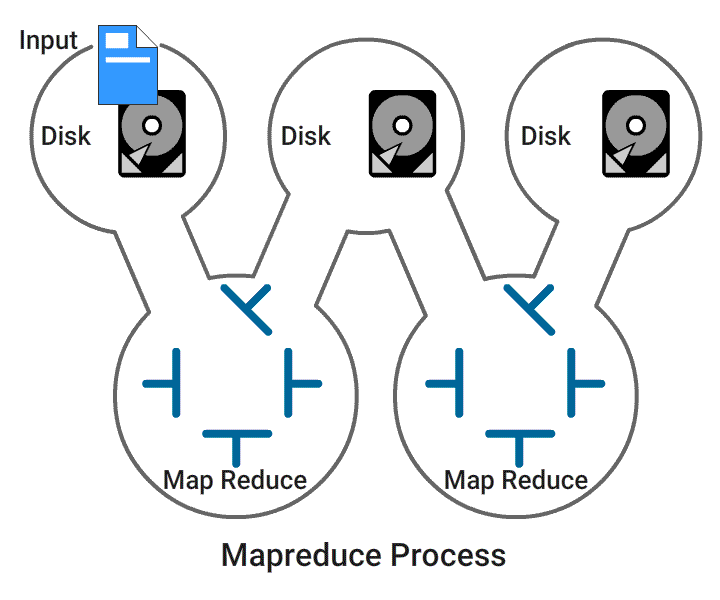
##### Advantages of Spark Over MapReduce

Let us compare **MapReduce** and **Spark** based on the following essential aspects in detail.

* **Solving Iterative problems**
* **Solving Interactive problems**

In the subsequent slides, we will look into these two problems in detail.

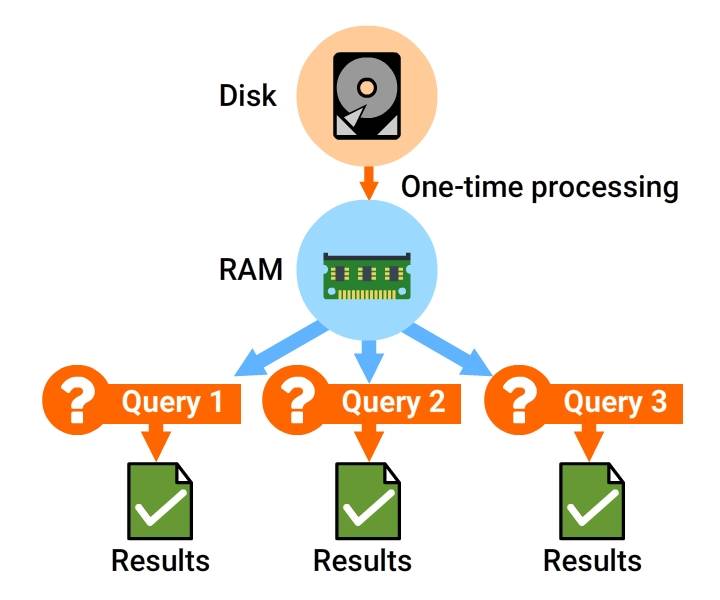
Solving Iterative Problems



The figure demonstrates how **MapReduce** and **Spark** respectively handle iterative problems.

* The **first figure** shows how in MapReduce, the intermediate results of each iteration are stored to disk and then read back for the next processing.
* The **second figure** shows how in the case of Spark processing, the results can be kept in RAM and fetched easily for each iteration. Thus there is no disk i/o related latency.

Solve Interactive Problems



The figure demonstrates how **Spark** handles interactive problems.

* In MapReduce, the same data is repeatedly read from disk for different queries.
* The **figure** shows how in the case of Spark processing, the input is read just once into memory where different queries act on the data to give their results.

##### Spark vs MapReduce

The other aspects by which Spark differs from MapReduce are summarized below.

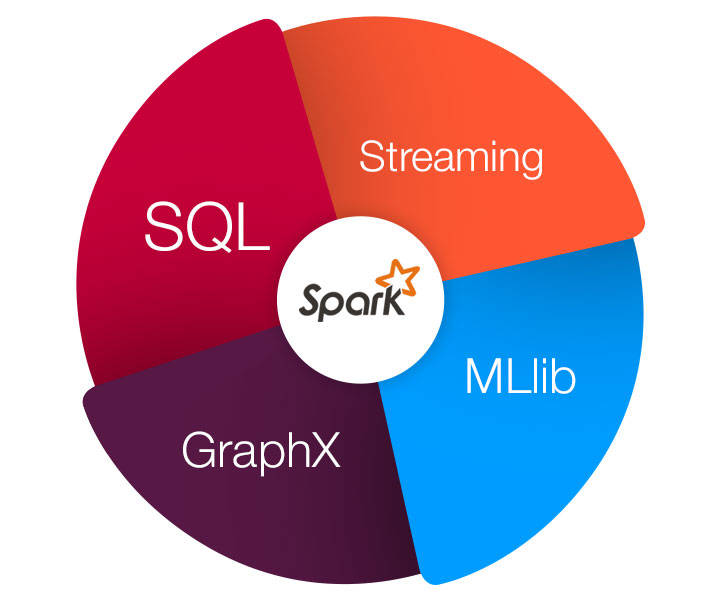
**Difficulty**: Apache Spark is a simpler to program and does not require any abstractions whereas MapReduce is hard to program with abstractions.

**Interactivity**: Spark provides an interactive mode whereas MapReduce has no inbuilt interactive mode except for Pig and Hive.

##### Spark vs MapReduce...

* **Streaming**: Hadoop MapReduce offers batch processing on historical data whereas Spark provides streaming of data and processing in real-time.
* **Latency**: Spark caches partial results over its memory of distributed workers thereby ensuring lower latency computations. In contrast to Spark, MapReduce is disk-oriented.
* **Speed**: Spark places the data in memory, by storing the data in Resilient Distributed Datasets (RDD). Spark is 100X quicker than Hadoop MapReduce for big data processing.

Spark Ecosystem



Now let us have a look at the components that make up Spark Ecosystem.

**Spark Core**:

* **Includes the primary functionality of Spark**, namely components for task scheduling, fault recovery, memory management, interacting with storage systems, etc.
* **Home to the API that represents RDD**, which is the primary programming abstraction of Spark.

**Spark SQL**:

* **Package for working with structured data**.
* **Enables querying data** through SQL and as the Apache Hive variant of SQL — termed as the Hive Query Language (HQL).
* **Supports various data**, including JSON, Parquet, and Hive tables.

Spark Ecosystem

* **Spark Streaming**:

Spark component that allows live-streaming data processing. Eg: includes log files created by production web servers, or queues of messages including status updates raised by web service users.

* **MLlib**: Spark appears with a library including common machine learning (ML) feature, named MLlib. Here, MLlib offers many types of machine learning algorithms, namely collaborative filtering, clustering, regression, and classification.
* **GraphX**: A library for performing graph-parallel computations and manipulating graphs.

##### Supported Languages

* Apache Spark currently supports multiple programming languages, including **Java, Scala, R** and **Python**. The final language is chosen based on the efficiency of the functional solutions to tasks, but most developers prefer **Scala.**
* Apache Spark is built on Scala, thus being proficient in Scala helps you to dig into the source code when something does not work as you expect.
* Scala is a multi-paradigm programming language and supports functional as well as object oriented paradigms. It is a JVM based statically typed language that is safe and expressive.
* Python is in general slower than Scala while Java is too verbose and does not support Read-Evaluate-Print-Loop (REPL).

To learn more about Scala, please refer the [course on Scala Constructs](https://play.fresco.me/course/176)

##### Applications of Spark

* **Interactive analysis** – MapReduce supports batch processing, whereas Apache Spark processes data quicker and thereby processes exploratory queries without sampling.
* **Event detection** – Streaming functionality of Spark permits organizations to monitor unusual behaviors for protecting systems. Health/security organizations and financial institutions utilize triggers to detect potential risks.
* **Machine Learning** – Apache Spark is provided with a scalable Machine Learning Library named as **MLlib**, which executes advanced analytics on iterative problems. Few of the critical analytics jobs such as sentiment analysis, customer segmentation, and predictive analysis make Spark an intelligent technology.

What kind of data can be handled by Spark?

Semi-structured

Structured

Unstructured

The following are characteristics shared by Hadoop and Spark, except \_\_\_\_\_\_\_\_\_\_\_

Both have their own file system

In which year was Apache Spark made an open-source technology?

2010

Which of the following application types can Spark run in addition to batch-processing jobs?

Stream processing

Machine learning

Graph processing

Apache Spark has which of the following capabilities?

Scheduling

Distributing

Monitoring

The programming paradigm used in Spark is \_\_\_\_\_\_\_\_\_\_.

Generalized

**Spark Data Model**

##### SparkConf

* **SparkConf** stores configuration parameters for a Spark application.
* These configuration parameters can be properties of the Spark driver application or utilized by Spark to allot resources on the cluster, like memory size and cores.
* SparkConf object can be created with **new SparkConf()** and permits you to configure standard properties and arbitrary key-value pairs via the set() method.

##### SparkConf

val conf = new SparkConf()

.setMaster("local[4]")

.setAppName("FirstSparkApp")

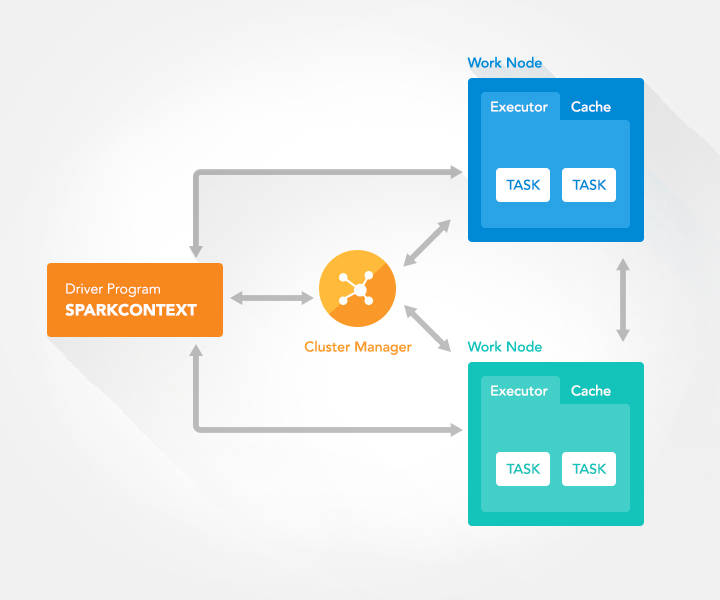
val sc = new SparkContext(conf)

Here, we have created a SparkConf object specifying the master URL and application name and passed it to a SparkContext.

##### SparkContext

* Main entry point for Spark functionality
* **SparkContext** can be utilized to create broadcast variables, RDDs, and accumulators, and denotes the connection to a Spark cluster.
* To create a **SparkContext**, you first have to develop a **SparkConf** object that includes details about your application.

SparkContext Contd..



* As shown in the diagram, the Spark driver program uses **SparkContext** to connect to the cluster manager for resource allocation, submit Spark jobs and knows what resource manager (YARN, Mesos or Standalone) to communicate.
* Via **SparkContext**, the driver can access other contexts like StreamingContext, HiveContext, and SQLContext to program Spark.

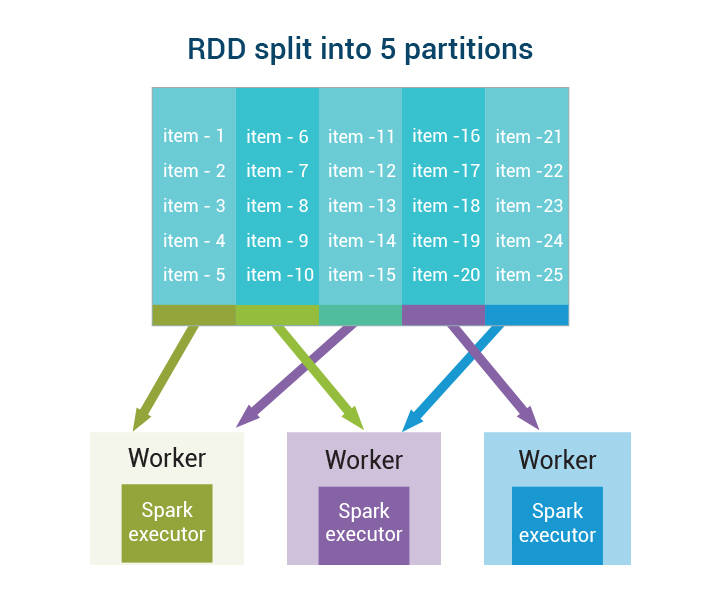
##### SparkContext

**There may be only one SparkContext active per JVM**. Before creating a new one, you have to stop() the active SparkContext.

* In the Spark shell, there is already a special interpreter-aware **SparkContext** created in the variable named as ***sc***.

val sc = new SparkContext(conf)

RDD



* **Resilient distributed datasets (RDDs)** are the known as the main abstraction in Spark.
* It is a partitioned collection of objects spread across a cluster, and can be persisted in memory or on disk.
* Once created, RDDs are immutable.

##### Features of RDDs

* **Resilient**, i.e. tolerant to faults using RDD lineage graph and therefore ready to recompute damaged or missing partitions due to node failures.
* **Dataset** - A set of partitioned data with primitive values or values of values, For example, records or tuples.
* **Distributed** with data remaining on multiple nodes in a cluster.

##### Creating RDDs

***Parallelizing a collection*** in driver program.

E.g., here is how to create a parallelized collection holding the numbers 1 to 5:

val data = Array(1, 2, 3, 4, 5)

val newRDD = sc.parallelize(data)

Here, newRDD is the new RDD created by calling SparkContext’s parallelize method.

***Referencing one dataset in an external storage system***, like a shared filesystem, HBase, HDFS, or any data source providing a Hadoop InputFormat.

For example, text file RDDs can be created using SparkContext’s textFile method. This method takes an URI for the file (either a local path on the machine, or a hdfs://, s3n://, etc URI) and reads it as a collection of lines to produce RDD newRDD.

val newRDD = sc.textFile("data.txt")

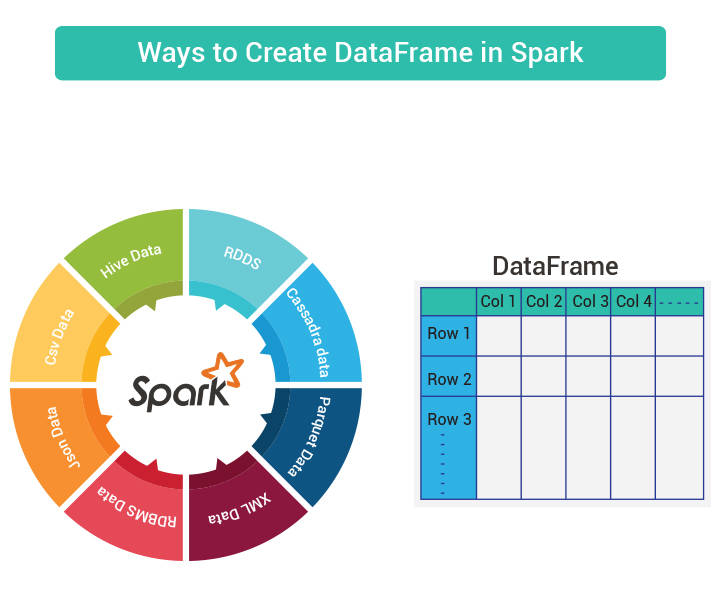
##### DataFrames

Similar to an RDD, a **DataFrame** is an immutable distributed set of data.

Unlike an RDD, data is arranged into named columns, similar to a table in a relational database.

Created to make processing simpler, **DataFrame** permits developers to impose a structure onto a distributed collection of data, enabling higher-level abstraction.

##### Creating DataFrames



*DataFrames* can be created from a wide array of sources like existing RDDs, external databases, tables in Hive, or structured data files.

##### Creating DataFrames...

***Applications can create DataFrames from a Hive table, data sources, or from an existing RDD with an SQLContext.***

The subsequent example creates a DataFrame based on the content of a JSON file:

val sc: SparkContext // An existing SparkContext.

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json("home/spark/input.json")

// Shows the content of the DataFrame to stdout

df.show()

##### SQL on DataFrames

The **sql** function on a SQLContext allows applications to run SQL queries programmatically and returns the result as a DataFrame.

val sc: SparkContext // An existing SparkContext.

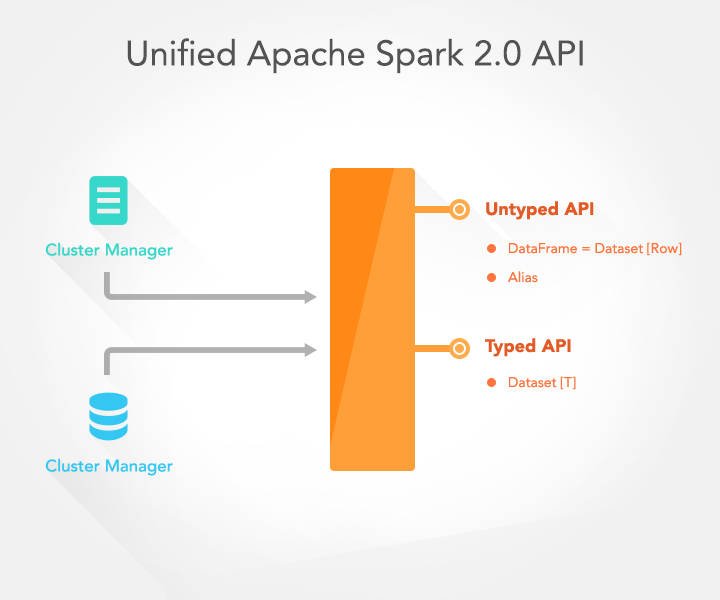
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json("home/spark/input.json")

df.registerTempTable("students")

val teenagers = sqlContext.sql("SELECT name, age FROM students WHERE age >= 13 AND age <= 19")

Dataset



* Dataset is a new interface included in Spark 1.6, which provides the advantages of RDDs with the advantages of Spark SQL’s optimized execution engine.
* It is an immutable, strongly-typed set of objects that are mapped to a relational schema.
* A DataFrame is a known as a Dataset organized into named columns.
* Dataset will act as the new abstraction layer for Spark from Spark 2.0.

##### Creating a Dataset

val lines = sqlContext.read.text("/log\_data").as[String]

val words = lines

.flatMap(\_.split(" "))

.filter(\_ != "")

Here, we have created a dataset **lines** on which RDD operations like filter and split are applied.

##### Benefits of Dataset APIs

1. Static-typing and runtime type-safety
2. High-level abstraction and custom view into structured and semi-structured data
3. Higher performance and Optimization

##### SparkSession - a New Entry Point

* **SparkSession**, introduced in Apache Spark 2.0, **offers a single point of entry** to communicate with underlying Spark feature and enables programming Spark with Dataset APIs and DataFrame.
* In previous versions of Spark, **spark context** was the entry point for Spark. For streaming, you required StreamingContext for hive HiveContext and for SQL SQLContext.
* As Dataframe and DataSet APIs are the new standards, Spark 2.0 features SparkSession as the new entry point.
* SparkSession is **a combination of *HiveContext*, *StreamingContext*, and *SQLContext***. All the APIs available on these contexts are available on SparkSession also. It internally has a spark context for actual computation.

##### Creating SparkSession

A **SparkSession** can be built utilizing a **builder pattern**. The builder will automatically reuse an existing SparkContext if one exists; and create a SparkContext if it does not exist.

val dataLocation = "file:${system:user.dir}/spark-data"

// Create a SparkSession

val spark = SparkSession

.builder()

.appName("SparkSessionExample")

.config("spark.sql.data.dir", dataLocation)

.enableHiveSupport()

.getOrCreate()

##### Configuring Properties

Once the **SparkSession** is instantiated, you can configure the runtime config properties of Spark. E.g: In this code snippet, we can alter the existing runtime config options.

//set new runtime options

spark.conf.set("spark.executor.memory", "1g")

spark.conf.set("spark.sql.shuffle.partitions", 4)

//get all settings

val configMap:Map[String, String] = spark.conf.getAll()

##### Running SQL Queries

SparkSession is the entry point for reading data, akin to the old SQLContext.read. It can be utilized to execute SQL queries across data, getting the results back as a DataFrame.

val jsonData = spark.read.json("/home/user/employee.json")

display(spark.sql("select \* from employee"))

##### Access to Underlying SparkContext

SparkSession.sparkContext returns the subsequent SparkContext, employed for building RDDs and managing cluster resources.

spark.sparkContext

res17: org.apache.spark.SparkContext = org.apache.spark.SparkContext@2debe9ac

##### Shared Variables

Usually, when a function passed to a Spark operation is run on a remote cluster node, it runs on individual copies of all the variables used in the function.

These variables are copied to every machine, and no updates to the variables on the remote machine are delivered back to the driver program.

Spark offers two limited types of shared variables for two common usage patterns: **accumulators** and **broadcast variables**.

##### Broadcast Variables

* Enables the programmer to keep a read-only variable cached on each machine instead of shipping a copy of it with tasks.
* Generated from a variable v by calling ***SparkContext.broadcast(v)***
* Its value can be accessed by calling **the value method**. The subsequent code shows this:

scala> val broadcastVar = sc.broadcast(Array(1, 2, 3, 4, 5))

broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

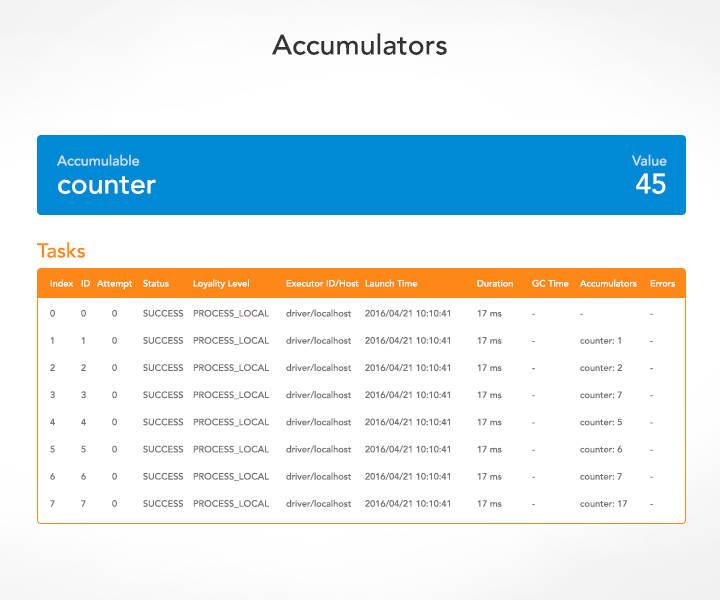
scala> broadcastVar.value

res0: Array[Int] = Array(1, 2, 3, 4, 5)

##### Accumulators

* Accumulators are known as the variables that are only “added” via an associative and commutative operation and can, hence, be efficiently supported in parallel.
* They can be utilized to implement sums or counters.
* Programmers can include support for new types.
* Spark natively offers support for accumulators of numeric types.

##### Named Accumulator



You can create unnamed or named accumulators as a user. As seen in the image, a named accumulator (here, **counter**) will be displayed in the web UI for the stage that modifies that accumulator. Spark shows the value for each accumulator modified by a task in the “Tasks” table.

Introduction to RDD Operations

* The following example shows the implementation of Spark - MapReduce’s example, word count. You can view that Spark offers support for **operator chaining**.
* It turns out to be handy when doing pre- or post-processing on data, like filtering data before running a complex MapReduce job.

val file = sc.textFile("hdfs://.../wordcounts-\*.gz")

val counts = file.flatMap(line => line.split(" "))

.map(word => (word, 1))

.reduceByKey(\_ + \_)

counts.saveAsTextFile("hdfs://.../wordcountOutput")

* Here *flatMap*, *map*,*reduceByKey* and *saveAsTextFile* are the operations on the RDDs.

##### Transformations

* Transformations are functions that use an RDD as the input and return one or more RDDs as the output.
* randomSplit, cogroup, join, reduceByKey, filter, and map are examples of few transformations.
* Transformations do not change the input RDD, but always create one or more new RDDs by utilizing the computations they represent.
* By using transformations, you incrementally create an RDD lineage with all the parent RDDs of the last RDD.
* Transformations are lazy, i.e. are not run immediately. Transformations are done on demand.
* Transformations are executed only after calling an action.

##### Example of Transformations

* **filter(func)**: Returns a new dataset (RDD) that are created by choosing the elements of the source on which the function returns true.
* **map(func)**: Passes each element of the RDD via the supplied function.
* **union()**: New RDD contains elements from source argument and RDD.
* **intersection()**: New RDD includes only common elements from source argument and RDD.
* **cartesian()**: New RDD cross product of all elements from source argument and RDD.

##### Actions

* Actions return concluding results of RDD computations.
* Actions trigger execution utilizing lineage graph to load the data into original RDD, and then execute all intermediate transformations and write final results out to file system or return it to Driver program.
* *Count, collect, reduce, take, and first* are few actions in spark.

##### Example of Actions

* **count()**: Get the number of data elements in the RDD
* **collect()**: Get all the data elements in an RDD as an array
* **reduce(func)**: Aggregate the data elements in an RDD using this function which takes two arguments and returns one
* **take (n)**: Fetch first n data elements in an RDD computed by driver program.
* **foreach(func)**: Execute function for each data element in RDD. usually used to update an accumulator or interacting with external systems.
* **first()**: Retrieves the first data element in RDD. It is similar to take(1).
* **saveAsTextFile(path)**: Writes the content of RDD to a text file or a set of text files to local file system/HDFS.

##### Sample Code Snippet

Let us look at the outline of a code snippet that uses transformations and actions on a RDD.

records = spark.textFile(“hdfs://...”)

errors = records.filter(\_.startsWith(“ERROR”))

messages = errors.map(\_.split(‘\t’)(2))

cachedMessages = messages.cache()

cachedMessages.filter(\_.contains(“400”)).count

In this program, **records** is the Base RDD and **errors** is a transformed RDD created by applying the filter transformation.

**Count** is the action called upon which the transformations start to execute.

Lineage Graph

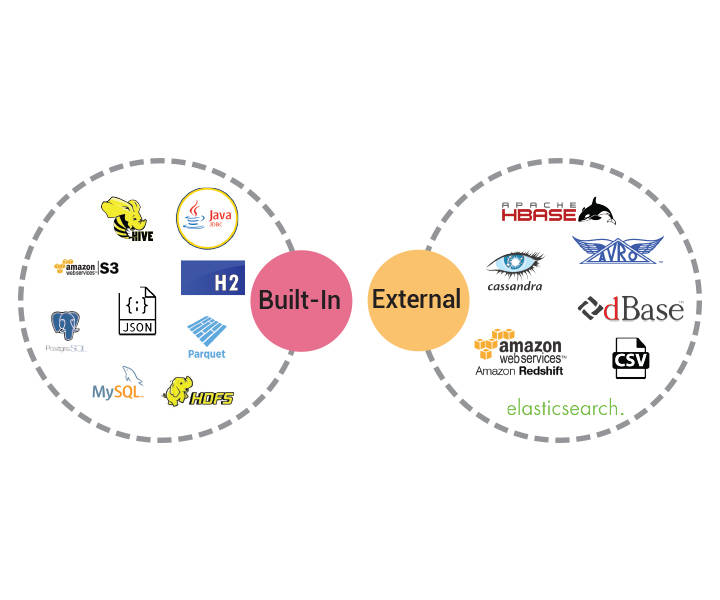


* In the above figure, you can view how transformations like map, filter, combineByKey act on each RDD.
* RDDs maintain a graph of one RDD transforming into another termed as *lineage graph*, which assists Spark to recompute any common RDD in the event of failures. This way Spark achieves fault tolerance.

##### Lazy Evaluation

* When we call a transformation on RDD, the operation is not immediately executed. Alternatively, Spark internally records meta-data to show this operation has been requested. It is called as **Lazy evaluation**.
* Loading data into RDD is lazily evaluated as similar to how transformations are.
* In Hadoop, developers often spend a lot of time considering how to group together operations to minimize the number of MapReduce passes. It is not required in case of Spark.
* Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together. Hence, users are free to arrange their program into smaller, more controllable operations.

Spark Sources



* The Data Sources API offers a single interface for storing and loading data using Spark SQL.
* In addition to the sources that come prepackaged with the Apache Spark distribution, this API offers an integration point for external developers to add support for custom data sources.

##### File Formats

The following are some of the file formats supported by Spark.

* Text
* JSON
* CSV
* Sequence File
* Parquet
* Hadoop InputOutput Format

##### Storage/Source Integrations

Although often linked to the Hadoop Distributed File System (HDFS), Spark can combine with various open source or commercial third-party data storage systems, including:

* Google Cloud
* Elastic Search
* JDBC
* Apache Cassandra
* Apache Hadoop (HDFS)
* Apache HBase
* Apache Hive

Developers are most expected to choose the data storage system they are previously utilizing elsewhere in their workflow.

##### Hive Integration

* Hive comes packaged with the Spark library as **HiveContext** that inherits from SQLContext. Utilizing HiveContext, you can create and find tables in the **HiveMetaStore** and write queries on it using **HiveQL**.
* When hive-site.xml is not configured, the context automatically produces a metastore named as **metastore\_db** and a folder known as **warehouse** in the current directory.

##### Data Load from Hive to Spark

Now let us see an example code to load data from Hive in Spark.

Consider the following example of employee in a text file named employee.txt. We will first create a hive table, load the employee record data into it using HiveQL language, and apply some queries on it.

Use the subsequent command for initializing the HiveContext into the Spark Shell

scala> val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)

Let us now create a table named employee with the fields id, name, and age using HQL.

scala> sqlContext.sql("CREATE TABLE IF NOT EXISTS employee(id INT, name STRING, age INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")

##### Data Load from Hive to Spark...

Now we shall load the employee data into the employee table in Hive.

scala> sqlContext.sql("LOAD DATA LOCAL INPATH 'employee.txt' INTO TABLE employee")

Next, we shall fetch all records using HiveQL select query.

scala> val result = sqlContext.sql("FROM employee SELECT id, name, age")

To show the record data, call the show() method on the result DataFrame.

scala> result.show()

What is/are the benefit(s) of using appropriate file formats in Spark?



More compression support



All the options

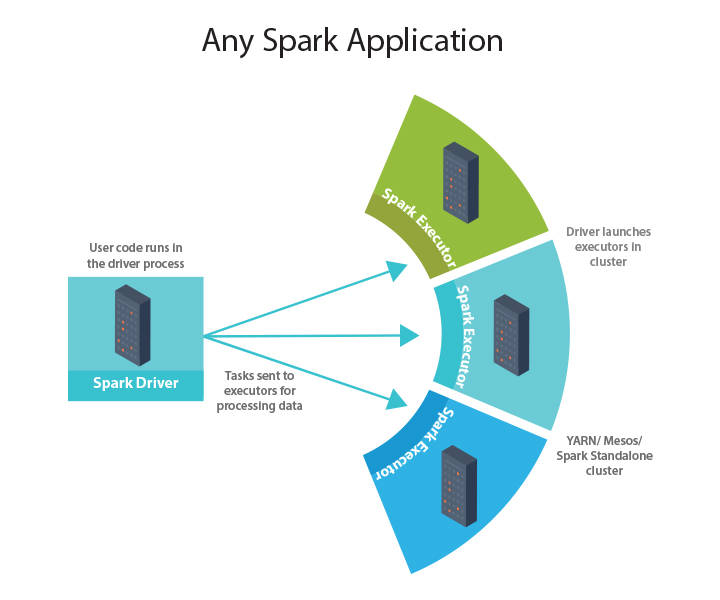


Faster accessing during read and write



Schema Oriented

Spark Cluster



* A Spark application includes a **single driver process** and a **collection of executor processes** scattered over nodes on the cluster.
* Both the executors and the driver usually run as long as the application runs.

##### Spark Driver

* Program that **produces the SparkContext**, connecting to a given Spark Master.
* **Declares the actions and transformations** on RDDs of data.

##### Spark Executors

* **Runs the tasks, return results** to the driver.
* **Offers in memory storage** for RDDs that are cached by user programs.
* Multiple executors per nodes possible

##### Cluster Managers

* **Standalone** – a simple cluster manager added with Spark that makes it simple to establish a cluster.
* **Apache Mesos** – a cluster manager that can run service applications and Hadoop MapReduce.
* **Hadoop YARN** – the resource manager in Hadoop 2.

##### Commonly Used Options

* **class**: entry point for your application (e.g. org.apache.spark.examples.SparkPi)
* **master**: master URL for the cluster (e.g. spark://23.195.26.187:7077)
* **deploy-mode**: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client)
* **conf**: Arbitrary Spark configuration property in key=value format.
* **application-jar**: Path to a bundled jar with the application and dependencies.
* **application-arguments**: Arguments passed to the main method of your main class, if any.

##### Launching Applications with Spark-submit

./bin/spark-submit \

--class <main-class> \

--master <master-url> \

--deploy-mode <deploy-mode> \

--conf <key>=<value> \

... # other options

<application-jar> \

[application-arguments]

##### Deployment Modes

Choose which mode to run using the --deploy-mode flag

1. **Client** - Driver runs on a dedicated server (e.g.: Edge node) inside a dedicated process. Submitter starts the driver outside of the cluster.
2. **Cluster** - Driver runs on one of the cluster's Worker nodes. The Master selects the worker. The driver operates as a dedicated, standalone process inside the Worker.

##### Running a Spark Job



Running a Spark Job

* Spark submit program initiated with spark driver; creates logical DAG.
* Spark Driver program checks with the cluster manager-YARN (Mesos or Standalone) for resource availability for executors and launches it.
* Executors created in Nodes, register to spark driver.
* Spark driver converts the actions and transformations defined in the main method and allocate to executors.
* Executors performs the transformations and; actions return

values to the driver.

* While reading from an HDFS, each executor directly applies the subsequent Operations, to the partition in the same task.

The DAG represents the logical flow of computations and allows Spark to optimize and partition the operations in a way that minimizes data movement and maximizes parallelism

To launch a Spark application in any one of the four modes (local, standalone, Mesos or YARN), use \_\_\_\_\_\_\_\_

./bin/spark-submit

Which is responsible for task scheduling and memory management?

Spark Core

Which of the following is true of running a Spark application on Hadoop YARN?

There are two deploy modes that can be used to launch Spark applications on YARN – client mode and cluster mode.

The type(s) of operation(s) that can be performed on RDDs is/are \_\_\_\_\_\_\_\_

Action and Map

Which tells Spark how and where to access a cluster?

Spark Context

RDD Caching and Persisting

* In Spark, you can utilize few RDDs multiple times. If we repeat the same process of RDD evaluation every time, it is needed or taken into action, this task can be time-consuming and memory-consuming, especially for iterative algorithms that look at data multiple times.
* To resolve the problem of repeated computation, the method of **caching** or **persistence** came into the picture.
* RDDs can be cached with the help of cache operation. They can also be persisted using persist operation.
* **Cache** **persists** with default storage level **MEMORY\_ONLY**.
* RDDs can also be unpersisted to eliminate RDD from a permanent storage like memory and disk.

How to Assign a Storage Level

* Let us see how to persist an RDD to a storage level.

result = input.map(<Computation>)

result.persist(LEVEL)

* By default, Spark uses the algorithm of **Least Recently Used (LRU)** to remove old and unused RDD to release more memory.
* We can also manually remove remaining RDD from memory by using **unpersist()**.

##### Storage Levels



These are the list of storage levels that can be assigned to an RDD.

By default, Spark uses which algorithm to remove old and unused RDD to release more memory?

Least Recently Used (LRU)

Which is the default Storage level in Spark?

MEMORY\_ONLY