Big Data Processing with Apache Spark

What is Spark ?

- Apache Spark is a fast and general engine for large-scale data processing

- Apache Spark is an open source big data processing framework built around speed, ease of use, and sophisticated analytics. It was originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project.

- Spark enables applications in Hadoop clusters to run up to 100 times faster in memory and 10 times faster even when running on disk.

- **Speed**
    - Run program up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk
    - Spark has an advanced DAG (Directed Acyclic Graph) execution engine that supports cyclic data flow and in-memory computing
- **Ease of use**
    - Write application quickly in Java, Python, Scala, R
    - Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python and R shells.
- **Generality**
    - Combine SQL, streaming, and complex analytics.
    - Spark powers a stack of high-level tools including Spark SQL, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.
- **Runs Everywhere**
    - Spark runs on Hadoop YARN, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, Hive and S3.
- Spark runs on top of existing Hadoop Distributed File System (HDFS) infrastructure to provide enhanced and additional functionality. It provides support for deploying Spark applications in an existing Hadoop v1 cluster (with SIMR – Spark-Inside-MapReduce) or Hadoop v2 YARN cluster or even Apache Mesos.

Spark Features

- Spark takes MapReduce to the next level with less expensive shuffles in the data processing
- Spark holds intermediate results in memory rather than writing them to disk
- It's designed to be an execution engine that works both in-memory and on-disk.

- Spark will attempt to store as much as data in memory and then will spill to disk. It can store part of a data set in memory and the remaining data on the disk. You have to look at your data and use cases to assess the memory requirements. With this in-memory data storage, Spark comes with performance advantage.

- Supports more than just Map and Reduce functions.
- Optimizes arbitrary operator graphs.
- Lazy evaluation of big data queries which helps with the optimization of the overall data processing workflow.
- Provides concise and consistent APIs in Scala, Java and Python.

- Offers interactive shell for Scala and Python. This is not available in Java yet.
- Spark is written in Scala Programming Language and runs on Java Virtual Machine (JVM) environment.
- Languages support
    - Java
    - Python
    - Scala
    - Clojure
    - R

Spark Ecosystem

Spark Streaming:

- Spark Streaming can be used for processing the real-time streaming data. This is based on micro batch style of computing and processing. It uses the DStream which is basically a series of RDDs, to process the real-time data.

Spark SQL:

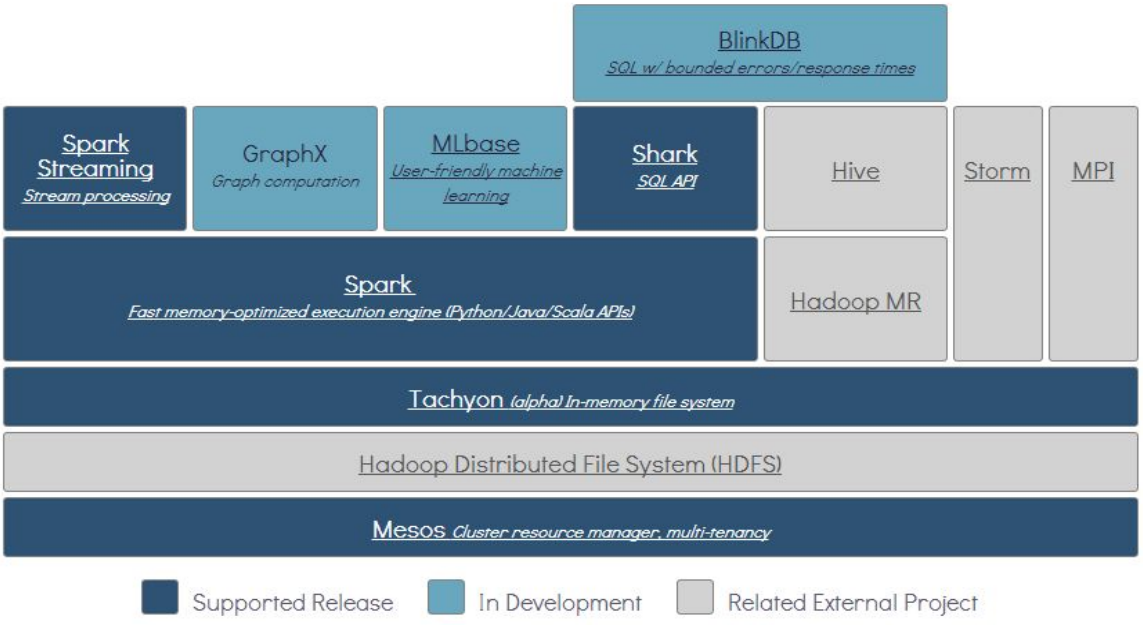
- Spark SQL provides the capability to expose the Spark datasets over JDBC API and allow running the SQL like queries on Spark data using traditional BI and visualization tools. Spark SQL allows the users to ETL their data from different formats it's currently in (like JSON, Parquet, a Database), transform it, and expose it for ad-hoc querying.

Spark MLlib:

- MLlib is Spark's scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.

Spark GraphX:

- GraphX is the new (alpha) Spark API for graphs and graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph: a directed multi-graph with properties attached to each vertex and edge. To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API. In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.



Spark Architecture

- **3 main components**
  - **Data Storage**
    Spark uses HDFS file system for data storage purposes. It works with any Hadoop compatible data source including HDFS, HBase, Cassandra, etc.

  - **API**
    The API provides the application developers to create Spark based applications using a standard API interface. Spark provides API for Scala, Java, and Python programming languages.

  - **Resource Management**
    Spark can be deployed as a Stand-alone server or it can be on a distributed computing framework like Mesos or YARN.

Resilient Distributed Datasets

- Resilient Distributed Dataset (based on Matei's research paper) or RDD is the core concept in Spark framework. Think about RDD as a table in a database. It can hold any type of data. Spark stores data in RDD on different partitions.
- They are also fault tolerance because an RDD know how to recreate and recompute the datasets.
- RDDs are immutable. You can modify an RDD with a transformation but the transformation returns you a new RDD whereas the original RDD remains the same.
- RDD supports two types of operations:
  - Transformation
  - Action

Transformation: Transformations don't return a single value, they return a new RDD. Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.

Some of the Transformation functions are **map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe, and coalesce.**

Action: Action operation evaluates and returns a new value. When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.

Some of the Action operations are **reduce, collect, count, first, take, countByKey, and foreach.**

Spark SQL

Spark SQL, part of Apache Spark big data framework, is used for structured data processing and allows running SQL like queries on Spark data. We can perform ETL on the data from different formats like JSON, Parquet, Database) and then run ad-hoc querying.

Spark SQL Components

- **DataFrame**
  - A DataFrame is a distributed collection of data organized into named columns. It is based on the data frame concept in R language and is similar to a database table in a relational database.
  - SchemaRDD in prior versions of Spark SQL API, has been renamed to DataFrame.
  - DataFrames can be converted to RDDs by calling the rdd method which returns the content of the DataFrame as an RDD of Rows.
  - DataFrames can be created from different data sources such as:
    - Existing RDDs
    - Structured data files
    - JSON datasets
    - Hive tables
    - External databases

Ref :
https://spark.apache.org/docs/1.3.0/api/scala/index.html#org.apache.spark.sql.DataFrame

Example :

```
// Create a DataFrame from Parquet files
val people = sqlContext.parquetFile("...")

// Create a DataFrame from data sources
val df = sqlContext.load("...", "json")

val ageCol = people("age")  // in Scala
Column ageCol = people.col("age")  // in Java

// The following creates a new column that increases everybody's age by 10.
people("age") + 10  // in Scala
people.col("age").plus(10);  // in Java
```

In Scala

```
// To create DataFrame using SQLContext
val people = sqlContext.parquetFile("...")
val department = sqlContext.parquetFile("...")

people.filter("age" > 30)
 .join(department, people("deptId") === department("id"))
 .groupBy(department("name"), "gender")
 .agg(avg(people("salary")), max(people("age")))
```

In Java

```
// To create DataFrame using SQLContext
DataFrame people = sqlContext.parquetFile("...");
DataFrame department = sqlContext.parquetFile("...");

people.filter("age".gt(30))
```

```
 .join(department, people.col("deptId").equalTo(department("id")))
 .groupBy(department.col("name"), "gender")
 .agg(avg(people.col("salary")), max(people.col("age")));
```

- **SQL Context**

  Spark SQL provides SQLContext to encapsulate all relational functionality in Spark.

  Example

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

- **JDBC Datasource**

  JDBC data source can be used to read data from relational databases using JDBC API.

Spark SQL Application

customers.txt

```
100, John Smith, Austin, TX, 78727
200, Joe Johnson, Dallas, TX, 75201
300, Bob Jones, Houston, TX, 77028
400, Andy Davis, San Antonio, TX, 78227
500, James Williams, Austin, TX, 78727
```

Scala example

```
// Create the SQLContext first from the existing Spark Context
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Import statement to implicitly convert an RDD to a DataFrame
import sqlContext.implicits._

// Create a custom class to represent the Customer
case class Customer(customer_id: Int, name: String, city: String, state: String, zip_code:
String)

// Create a DataFrame of Customer objects from the dataset text file.
val dfCustomers = sc.textFile("data/customers.txt").map(_.split(",")).map(p =>
Customer(p(0).trim.toInt, p(1), p(2), p(3), p(4))).toDF()

// Register DataFrame as a table.
dfCustomers.registerTempTable("customers")

// Display the content of DataFrame
dfCustomers.show()

// Print the DF schema
dfCustomers.printSchema()
```

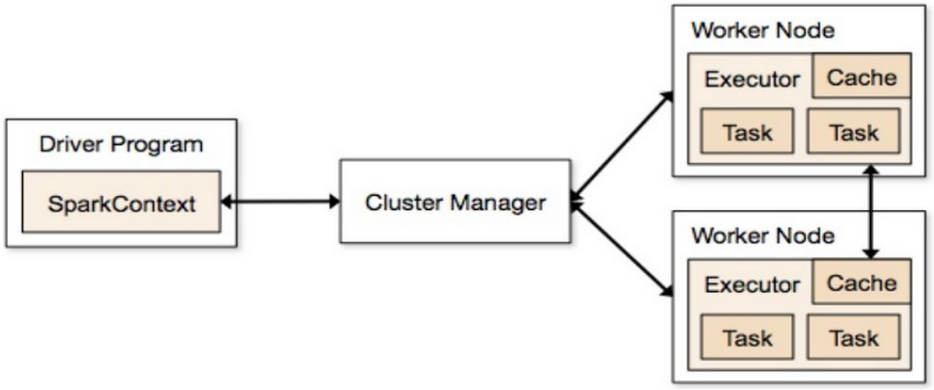
```
// Select customer name column
dfCustomers.select("name").show()

// Select customer name and city columns
dfCustomers.select("name", "city").show()

// Select a customer by id
dfCustomers.filter(dfCustomers("customer_id").equalTo(500)).show()

// Count the customers by zip code
dfCustomers.groupBy("zip_code").count().show()
```

Following code example shows how to specify the schema using the new data type classes StructType, StringType, and StructField.

```
//
// Programmatically Specifying the Schema
//

// Create SQLContext from the existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Create an RDD
val rddCustomers = sc.textFile("data/customers.txt")

// The schema is encoded in a string
val schemaString = "customer_id name city state zip_code"

// Import Spark SQL data types and Row.
import org.apache.spark.sql._

import org.apache.spark.sql.types._;

// Generate the schema based on the string of schema
val schema = StructType(schemaString.split(" ").map(fieldName => StructField(fieldName,
StringType, true)))

// Convert records of the RDD (rddCustomers) to Rows.
val rowRDD = rddCustomers.map(_.split(",")).map(p => Row(p(0).trim,p(1),p(2),p(3),p(4)))

// Apply the schema to the RDD.
val dfCustomers = sqlContext.createDataFrame(rowRDD, schema)

// Register the DataFrames as a table.
dfCustomers.registerTempTable("customers")

// SQL statements can be run by using the sql methods provided by sqlContext.
val custNames = sqlContext.sql("SELECT name FROM customers")

// The results of SQL queries are DataFrames and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
custNames.map(t => "Name: " + t(0)).collect().foreach(println)
```

```
// SQL statements can be run by using the sql methods provided by sqlContext.
val customersByCity = sqlContext.sql("SELECT name,zip_code FROM customers ORDER BY zip_code")

// The results of SQL queries are DataFrames and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
customersByCity.map(t => t(0) + "," + t(1)).collect().foreach(println)
```


Execution Flow



http://spark.apache.org/docs/latest/cluster-overview.html

Terminology

- **Application Jar**
  - User Program and its dependencies except Hadoop & Spark Jars bundled into a Jar file
- **Driver Program**
  - The process to start the execution (main() function)
- **Cluster Manager**
  - An external service to manage resources on the cluster (standalone manager, YARN, Apache Mesos)
- **Deploy Mode**
  - **cluster** : Driver inside the cluster
  - **client** : Driver outside of Cluster

Terminology (contd.)

- **Worker Node :** Node that run the application program in cluster
- **Executor**
  - Process launched on a worker node, that runs the Tasks
  - Keep data in memory or disk storage
- **Task :** A unit of work that will be sent to executor
- **Job**
  - Consists multiple tasks
  - Created based on a Action
- **Stage :** Each Job is divided into smaller set of tasks called Stages that is sequential and depend on each other
- **SparkContext :**
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.

Resilient Distributed Dataset (RDD)

- Resilient Distributed Dataset (RDD) is a basic Abstraction in Spark
- Immutable, Partitioned collection of elements that can be operated in parallel
- Basic Operations
  - map
  - filter
  - persist
- Multiple Implementation
  - PairRDDFunctions : RDD of Key-Value Pairs, groupByKey, Join
  - DoubleRDDFunctions : Operation related to double values
  - SequenceFileRDDFunctions : Operation related to SequenceFiles
- RDD main characteristics:
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)
- Custom RDD can be also implemented (by overriding functions)

http://www.slideshare.net/rahuldausa/real-time-analytics-with-apache-kafka-and-apache-spark?qid=e5b5328b-f403-49e4-938f-60bfbd68a573&v=default&b=&from_search=1

Apache Spark Statistics

http://www.slideshare.net/Typesafe_Inc/sneak-preview-apache-spark?qid=e609c399-dab8-4874-a90a-18338e6272c2&v=default&b=&from_search=7

http://www.slideshare.net/deanwampler/spark-the-next-top-compute-model-39976454

https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Spark APIs list
http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

- Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

- RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: **iterative algorithms and interactive data mining tools.**

- RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

- RDDs provide an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items.

- Formally, an RDD is a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs.

- Transformations include **map, filter, and join**

- users can control two other aspects of RDDs: **persistence and partitioning.**

- each dataset is represented as an object and transformations are invoked using methods on these objects.

- Programmers start by defining one or more RDDs through **transformations** on data in stable storage (**e.g., map and filter**). They can then use these RDDs in **actions**, which are operations that return a value to the application or export data to a storage system.

- Examples of actions include **count** (which returns the number of elements in the dataset), **collect** (which returns the elements themselves), and **save** (which outputs the dataset to a storage system).

- programmers can call a **persist** method to indicate which RDDs they want to reuse in future operations.

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

- Need to learn Scala
- More Spark examples such as Spark Streaming , Spark SQL
- Apache Kafka + Apache Spark Integration
- Scala + Spark Interview Questions
- Lambda Architecture --- done
- Go through existing frameworks such as Hadoop, YARN, Hive, Pig, HBase, etc.,
- Hive UDF
- Pig

https://www.mapr.com/developercentral/lambda-architecture