[**http://aptuz.com/blog/is-apache-spark-going-to-replace-hadoop/**](http://aptuz.com/blog/is-apache-spark-going-to-replace-hadoop/)

**About Spark: Is it Replacement for MapReduce?**

Cloudera and Intel join roadmap for Spark. Cloudera and Intel engineers are major contributor for spark

Spark was developed in response to limitations in the MapReduce cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs: MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results on disk. Spark's RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory.

Spark requires a cluster manager and a distributed storage system. For cluster management, Spark supports standalone (native Spark cluster), Hadoop YARN, or Apache Mesos. For distributed storage, Spark can interface with a wide variety, including Hadoop Distributed File System (HDFS), MapR File System (MapR-FS), Cassandra, OpenStack Swift, Amazon S3, Kudu, or a custom solution can be implemented

Data Sharing is Slow in MapReduce

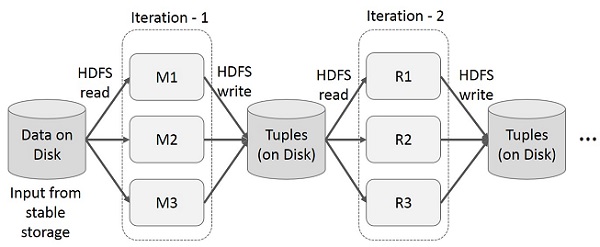
MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex − between two MapReduce jobs) is to write it to an external stable storage system (Ex − HDFS). Although this framework provides numerous abstractions for accessing a cluster’s computational resources, users still want more.

Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Iterative Operations on MapReduce

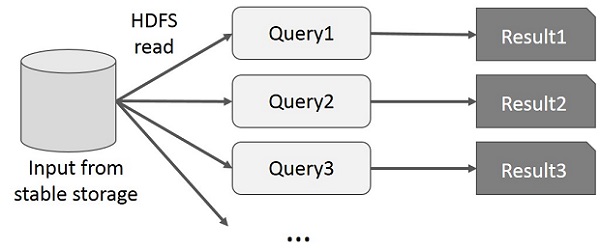
Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.



Interactive Operations on MapReduce

User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominates application execution time.

The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



Data Sharing using Spark RDD

Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is **R**esilient **D**istributed **D**atasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Let us now try to find out how iterative and interactive operations take place in Spark RDD.

Iterative Operations on Spark RDD

The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

**Note** − If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.

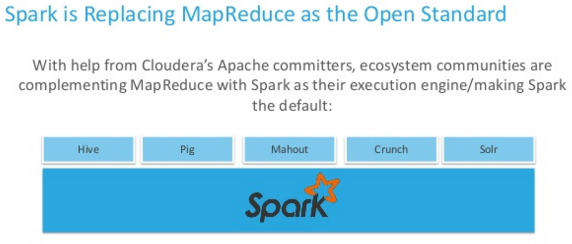


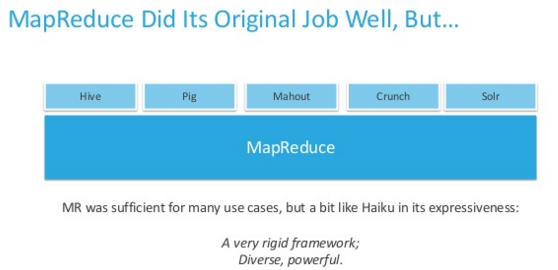
Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.

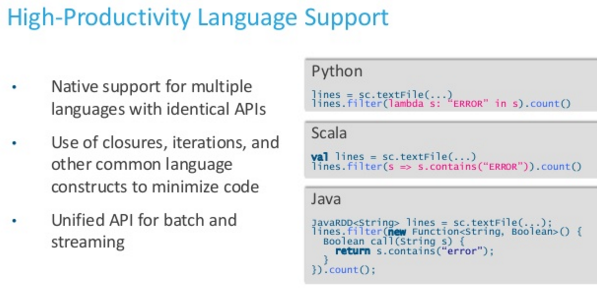


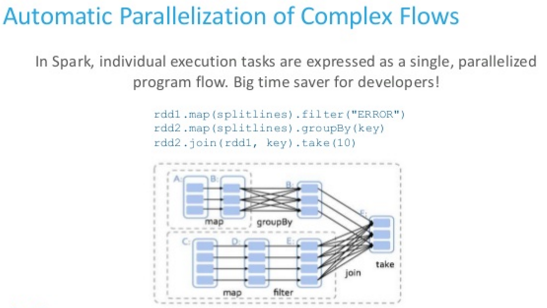
By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also **persist** an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

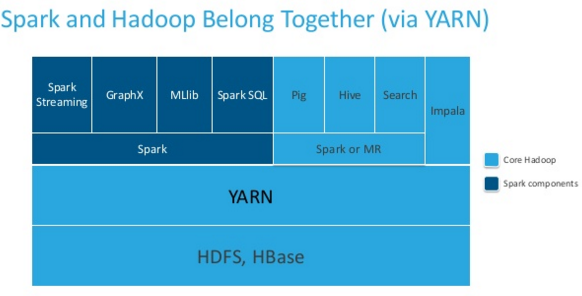
****

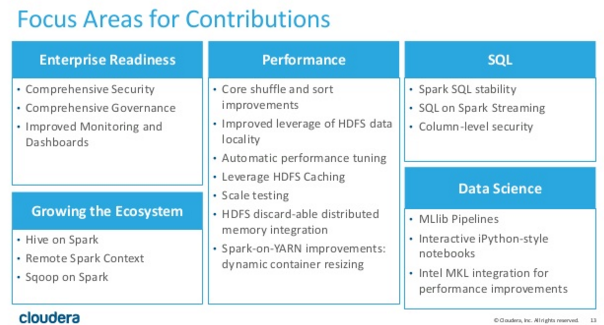












|  |  |  |
| --- | --- | --- |
| **S.No** | **Topic** | **Desc** |
|  | **Weblink** |  |
| 1 | Streaming programming guide | <http://spark.apache.org/docs/latest/streaming-programming-guide.html> |
| 2 | Scala & Scala SBT download & setup | <https://www.youtube.com/watch?v=UxubzWUo4ag&index=2&list=PLf0swTFhTI8pQcKpwJHs4irN-0YOPwhvm> |
| 3 | Scala download | <http://www.scala-lang.org/download/> |
| 4 | Scala SBT (need both Scala & Scala SBT) | <http://www.scala-sbt.org/download.html> |
| 5 | Tutorial | <http://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm> |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  | **General Info** |  |
| 1 | Spark overview | Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. |
| 2 | Spark Overview | Spark is a 3rd generation of distributed system   1. The first generation of distributed system is Teradata 2. The second generation of distributed system is Hadoop/MapReduce 3. The third generation of distributed system is Spark   Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches |
| 3 | Dstreams | Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of [RDDs](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD) |
| 4 | Resilient Distributed Datasets (RDD) | Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.  There are two ways to create RDDs − parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.  Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.  Internally, each RDD is characterized by five main properties:   1. A list of partitions 2. A function for computing each split 3. A list of dependencies on other RDDs 4. Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned) 5. Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)   All of the scheduling and execution in Spark is done based on these methods, allowing each RDD to implement its own way of computing itself |
| 4.1 | RDD in Nutshell | 1. For easy understand, you can consider RDD as a big array located in multiple nodes 2. RDD Can be persisted in memory 3. RDD Auto Recover from node failures: if data lost, then it will automatically read the data from source 4. RDD can have any data type but has a special dataset type for key-value 5. Supports two type of operations: transformation and action 6. RDD is read only |
| 5 | Pre-request | Java, Scala /Python. This doc uses scala 2.11.7 version |
| 6 | A sample exercise using Spark | <http://www.cloudera.com/training/certification/cca-spark.html> |
| 7 | Sqoop vs Flume vs Spark  *<An example for Data Ingest>* | The skills to transfer data between external systems and your cluster. This includes the following:   1. Import data from a MySQL database into HDFS using Sqoop 2. Export data to a MySQL database from HDFS using Sqoop 3. Change the delimiter and file format of data during import using Sqoop 4. Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume 5. Load data into and out of HDFS using the Hadoop File System (FS) commands |
| 8 | Sqoop vs Flume vs Spark  *<An example for Transform, Stage, Store>* | Convert a set of data values in a given format stored in HDFS into new data values and/or a new data format and write them into HDFS. This includes writing Spark applications in both Scala and Python (see note above on exam question format for more information on using either Scale or Python):   1. Load data from HDFS and store results back to HDFS using Spark 2. Join disparate datasets together using Spark 3. Calculate aggregate statistics (e.g., average or sum) using Spark 4. Filter data into a smaller dataset using Spark 5. Write a query that produces ranked or sorted data using Spark |
| 9 | **Exercise1 :** Exercise to be done using Spark | 1. Load data from HDFS and store results back to HDFS using Spark 2. Join disparate datasets together using Spark 3. Calculate aggregate statistics (e.g., average or sum) using Spark 4. Filter data into a smaller dataset using Spark 5. Write a query that produces ranked or sorted data using Spark |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  | **Admin Commands** |  |
|  | Web URL To monitor Spark Jobs | http://Localhost:4040/jobs  http://Ipaddress:4040/jobs |
|  | Pre-request | Java 7 for Python 2.7 and Scala 2.11.7 |
| 1 | Check the spark availability | >spark-shell |
| 2 | Download the file | >wget filepath |
| 3 | Unzip | > tar -xzf spark-1.3.1-bin-hadoop1.tgz |
| 4 | Update bashrc | >gedit ~/.bashrc  export SPARK\_HOME=/home/hadoop/spark\_path...  export PATH=$SPARK\_HOME/bin: $JAVA\_HOME/bin:$HADOOP\_HOME/bin : existing\_other\_paths\_like: |
| 5 | Copy spark-env.sh | >cd conf  >cp spark-env.sh.template spark-env.sh |
| 6 | Edit spark-env.sh | >vi /conf/spark-env.sh  SPARK\_MASTER\_IP=  SPARK\_WORKER\_CORES=1  SPARK\_WORKER\_MEMORY=1g  SPARK\_WORKER\_INSTANCES=2  SPARK\_WORKER\_DIR=/home/spark\_work\_dir |
| 7 | Git availability | # Spark build depends on git  # To check git is available by  >git |
| 8 | Git installation | # To install git  >sudo apt-get install git |
| 9 | Check the spark availability again | # To validate the spark availability  >spark-shell  **Note:** Spark context available as sc. |
| 10 | Port | # Spark Port: 4040 and 8080 (to be verified about 4040) |
| 11 | Start master.sh | # Start the Spark Matser  >cd $SPARK\_HOME  > ./sbin/start-master.sh |
| 12 | WUI | # Spark jobs in IE  http://ip\_where\_spark\_installed:4040/jobs  http://ip\_where\_spark\_installed:8080 |
| 13 | Run a test script | # Run a test spark script  >val textfile = sc.textfile("/home/hadoop/desktop/testfile.txt")  >textfile.count() |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  | **Development Command – Part1** |  |
| 1 | To verify spark availability | # Cloudera quick start will automatically install Spark  **To Login to scala**  >spark-shell  Scala>  **To login to spark sql for Hive queries**  >spark-sql  Spark-sql>  **To login to python contect**  >pyspark |
|  | To assign value to collection | Scala> val data = 1 to 100  It will automatically assign value 1 to 100 and show the details |
| 2 | To assign values to variables | # To assign the value 0 to the variable myint  scala>val myint: Int = 0  # To assign some string to the variable mystr  scala>val mystr: String = new String("Hi") |
| 3 | To print the variable | # To print the mystr  scala>println(mystr)  Hi |
| 4 | SparkContext  >spark-sql | # To process the data using SQL type of syntax from HDFS using Spark  # Pre-request: Hive, Pig, etc  # The difference is Hive uses MapReduce, but using spark-sql, you can avoid MapReduce and it uses its own way of execution (so it replaces MapReduce)  # if you have ‘spark-sql’ then you can process the data in SQL itself and minimize the usage of Scala and Python   1. SQLContext 2. HiveContext   >spark-sql (only latest version) |
| 5 | RDBMS connection using JDBC | # Spark can also connect to RDBMS using JDBC Driver |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |