Simple | Smart | Speed

CHOOLS

Apache Spark

We live in an era of “Big Data” where data of various types are being generated at an unprecedented pace, and this pace seems to be only accelerating astronomically. This data can be categorized broadly into transactional data, social media content (such as text, images, audio, and video), and sensor feeds from instrumented devices.

But one may ask why it is important to pay any attention to it. The reason being: “**Data is valuable because of the decisions it enables**”.

Up until a few years ago, there were only a few companies with the technology and money to invest in storing and mining huge amounts of data to gain invaluable insights. However, everything changed when Yahoo open sourced Apache Hadoop in 2009. It was a disruptive change that lowered the bar on Big Data processing considerably. As a result, many industries, such as Health care, Infrastructure, Finance, Insurance, Telematics, Consumer, Retail, Marketing, E-commerce, Media, Manufacturing, and Entertainment, have since tremendously benefited from practical applications built on Hadoop.

Apache Hadoop provides two major capabilities:

1. **HDFS**, a fault tolerant way to store vast amounts of data inexpensively using horizontally scalable commodity hardware.
2. **Map-Reduce computing paradigm**, which provide programming constructs to mine data and derive insights.

Figure 1 illustrates how data are processed through a series of Map-Reduce steps where output of a Map-Reduce step is input to the next in a typical Hadoop job.

Figure 1

The intermediate results are stored on the disk, which means that most Map-Reduce jobs are I/O bound, as opposed to being computationally bound. This is not an issue for use cases such as ETLs, data consolidation, and cleansing, where processing times are not much of a concern, but there are other types of Big Data use cases where processing time matters. These use cases are listed below:

1. **Streaming data processing** to perform near real-time analysis. For example, clickstream data analysis to make video recommendations, which enhances user engagement. We have to trade-off between accuracy and processing time.
2. **Interactive querying** of large datasets so a data scientist may run ad-hoc queries on a data set.

These are the challenges that Apache Spark solves! *Spark is a lightning fast in-memory cluster-computing platform, which has unified approach to solve Batch, Streaming, and Interactive use cases* as shown in Figure 3

About Apache Spark

Apache Spark is an open source, Hadoop-compatible, fast and expressive cluster-computing platform. It was created at AMPLabs in UC Berkeley as part of Berkeley Data Analytics Stack (BDAS). It has emerged as a top level Apache project. Figure 4 shows the various components of the current Apache Spark stack.

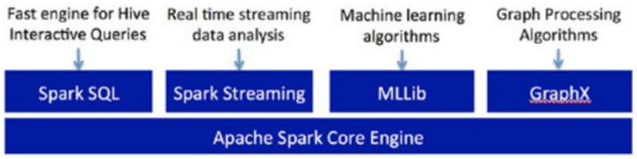

Fast engine for Hive Interactive Queries
Real time streaming data analysis
Machine learning algorithms
Graph Processing Algorithms
Spark SQL
Spark Streaming
MLLib
GraphX
Apache Spark Core Engine

Figure 4

It provides five major benefits:

1. **Lightning speed of computation** because data are loaded in distributed memory (RAM) over a cluster of machines. Data can be quickly transformed iteratively and cached on demand for subsequent usage. It has been noted that Apache Spark processes data 100x faster than Hadoop Map Reduce when all the data fits in memory and 10x faster when some data spills over onto disk because of insufficient memory.

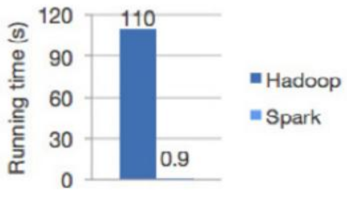


Figure 5

2. **Highly accessible** through standard APIs built in Java, Scala, Python, or SQL (for interactive queries), and a rich set of machine learning libraries available out of the box.
3. **Compatibility** with the existing Hadoop v1 (SIMR) and 2.x (YARN) ecosystems so companies can leverage their existing infrastructure.

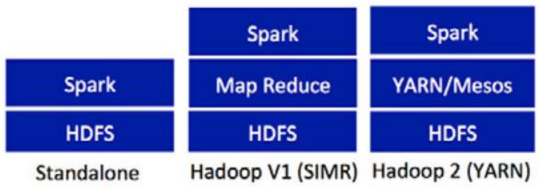


Figure 6

4. **Convenient** download and installation processes. Convenient shell (REPL: Read-Eval-Print-Loop) to interactively learn the APIs.
5. **Enhanced productivity** due to high level constructs that keep the focus on content of computation.

Also, Spark is implemented in Scala, which means that the code is very succinct.

How to Install Apache Spark

The following table lists a few important links and prerequisites:

Current Release	1.0.1 @ http://d3kbcqa49mib13.cloudfront.net/spark-1.0.1.tgz
Downloads Page	https://spark.apache.org/downloads.html
JDK Version (Required)	1.6 or higher
Scala Version (Required)	2.10 or higher
Python (Optional)	[2.6, 3.0)
Simple Build Tool (Required)	http://www.scala-sbt.org
Development Version	git clone git://github.com/apache/spark.git
Building Instructions	https://spark.apache.org/docs/latest/building-with-maven.html
Maven	3.0 or higher

As shown in Figure 6, Apache Spark can be configured to run standalone, or on Hadoop V1 SIMR, or on Hadoop 2 YARN/Mesos. Apache Spark requires moderate skills in Java, Scala or Python. Here we will see how to install and run Apache Spark in the standalone configuration.

1. Install JDK 1.6+, Scala 2.10+, Python [2.6,3) and sbt
2. Download Apache Spark 1.0.1 Release
3. Untar & Unzip spark-1.0.1.tgz in a specified directory

```
akuntamukkala@localhost~/Downloads$ pwd
/Users/akuntamukkala/Downloads
akuntamukkala@localhost~/Downloads$ tar -zxvf spark-1.0.1.tgz -C
/Users/akuntamukkala/spark
```

4. Go to the directory from #4 and run sbt to build Apache Spark

```
akuntamukkala@localhost~/spark/spark-1.0.1$ pwd
/Users/akuntamukkala/spark/spark-1.0.1
akuntamukkala@localhost~/spark/spark-1.0.1$ sbt/sbt assembly
```

5. Launch Apache Spark standalone REPL

   For Scala, use:
   `/Users/akuntamukkala/spark/spark-1.0.1/bin/spark-shell`

   For Python, use: `/Users/akuntamukkala/spark/spark-1.0.1/bin/ pyspark`

6. Go to SparkUI @ http://localhost:4040

How Apache Spark works

Spark engine provides a way to process data in distributed memory over a cluster of machines. Figure 7 shows a logical diagram of how a typical Spark job processes information.

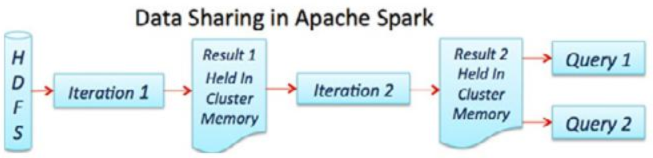

Data Sharing in Apache Spark
HDFS
Iteration 1
Result 1 Held In Cluster Memory
Iteration 2
Result 2 Held In Cluster Memory
Query 1
Query 2

Figure 7

Figure 8 shows how Apache Spark executes a job on a cluster

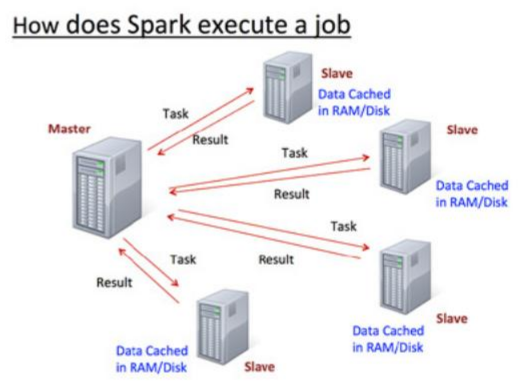


Figure 8

The Master controls how data is partitioned, and it takes advantage of data locality while keeping track of all the distributed data computation on the Slave machines. If a certain Slave machine is unavailable, the data on that machine is reconstructed on other available machine(s). "Master" is currently a single point of failure, but it will be fixed in upcoming releases.

Resilient Distributed Dataset

The core concept in **Apache Spark is the Resilient Distributed Dataset** (RDD). *It is an immutable distributed collection of data, which is partitioned across machines in a cluster*. It facilitates two types of operations: **transformation** and **action**. A transformation is an operation such as filter(), map(), or union() on an RDD that yields another RDD. An action is an operation such as count(), first(), take(n), or collect() that triggers a computation, returns a value back to the Master, or writes to a stable storage system. Transformations are **lazily evaluated**, in that they don't run until an action warrants it. Spark Master/Driver remembers the transformations applied to an RDD, so if a partition is lost (say a slave machine goes down), that partition can easily be reconstructed on some other machine in the cluster. That is why is it called "Resilient."

Figure 9 shows how transformations are lazily evaluated:

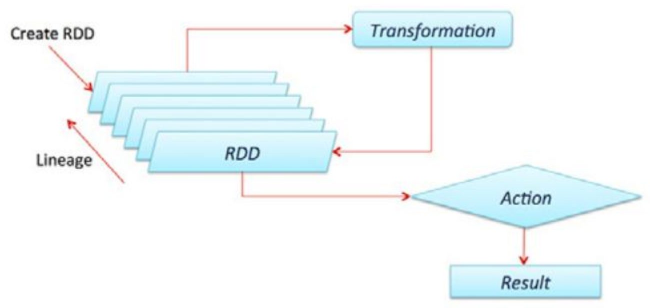
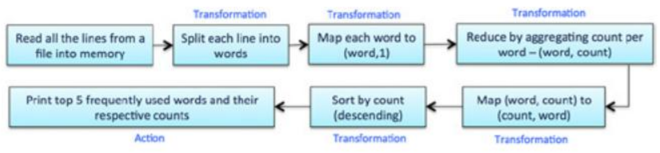


Figure 9

Let's understand this conceptually by using the following example: Say we want to find the 5 most commonly used words in a text file. A possible solution is depicted in Figure 10.


Transformation
Transformation
Transformation
Read all the lines from a file into memory
Split each line into words
Map each word to (word,1)
Reduce by aggregating count per word – (word, count)
Print top 5 frequently used words and their respective counts
Sort by count (descending)
Map (word, count) to (count, word)
Action
Transformation
Transformation
Figure 10

The following code snippets show how we can do this in Scala using Spark Scala REPL shell:

```
scala> val hamlet =
sc.textFile("/Users/akuntamukkala/temp/gutenburg.txt")
hamlet: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at
textFile at <console>:12
```

In the above command, we read the file and create an RDD of strings. Each entry represents a line in the file.

```
scala> val topWordCount = hamlet.flatMap(str=>str.split(" ")).
filter(!_.isEmpty).map(word=>(word,1)).reduceByKey(_+_).map{case (word, count) =>
(count, word)}.sortByKey(false) topWordCount: org.apache.spark.rdd.RDD[(Int, String)]
= MapPartitionsRDD[10] at sortByKey at <console>:14
```

1. The above commands shows how simple it is to chain the transformations and actions using succinct Scala API. We split each line into words using hamlet.flatMap(str=>str.split(" ")).
2. There may be words separated by more than one whitespace, which leads to words that are empty strings. So we need to filter those empty words using filter(!_.isEmpty).
3. We map each word into a key value pair: map(word=>(word,1)).
4. In order to aggregate the count, we need to invoke a reduce step using reduceByKey(_+_). The _+_ is a shorthand function to add values per key.
5. We have words and their respective counts, but we need to sort by counts. In Apache Spark, we can only sort by key, not values. So, we need to reverse the (word, count) to (count, word) using map{case (word, count) => (count, word)}.
6. We want the top 5 most commonly used words, so we need to sort the counts in a descending order using sortByKey(false).

```
scala> topWordCount.take(5).foreach(x=>println(x)) (1044,the)

(730,and)
(679,of)
(648,to)
(511,I)
```

The above command contains.take(5) (an action operation, which triggers computation) and prints the top ten most commonly used words in the input text file: /Users/akuntamukkala/temp/gutenburg.txt.

The same could be done in the Python shell also.

RDD lineage can be tracked using a useful operation: toDebugString

```
scala> topWordCount.toDebugString
res8: String = MapPartitionsRDD[19] at sortByKey at <console>:14 ShuffledRDD[18] at
sortByKey at <console>:14
          MappedRDD[17] at map at <console>:14 MapPartitionsRDD[16] at
            reduceByKey at <console>:14
                ShuffledRDD[15] at reduceByKey at <console>:14 MapPartitionsRDD[14] at
                  reduceByKey at <console>:14
                      MappedRDD[13] at map at <console>:14 FilteredRDD[12] at
                        filter at <console>:14
                            FlatMappedRDD[11] at flatMap at <console>:14
                                MappedRDD[1] at textFile at <console>:12
                                    HadoopRDD[0] at textFile at <console>:12
```

Commonly Used Transformations:

Transformation & Purpose	Example & Result
filter(*func*) **Purpose:** new RDD by selecting those data elements on which func returns true	scala> val rdd = sc.parallelize(List("ABC","BCD","DEF")) scala> val filtered = rdd.filter(_.contains("C")) scala> filtered.collect() **Result:** Array[String] = Array(ABC, BCD)
map(*func*) **Purpose:** return new RDD by applying func on each data element	scala> val rdd=sc.parallelize(List(1,2,3,4,5)) scala> val times2 = rdd.map(_*2) scala> times2.collect() **Result:** Array[Int] = Array(2, 4, 6, 8, 10)

flatMap(*func*) **Purpose:** Similar to map but func returns a Seq instead of a value. For example, mapping a sentence into a Seq of words	scala> val rdd=sc.parallelize(List("Spark is awesome","It is fun")) scala> val fm=rdd.flatMap(str=>str.split(" ")) scala> fm.collect() **Result:** Array[String] = Array(Spark, is, awesome, It, is, fun)
reduceByKey(*func*,[*numTasks*]) **Purpose:** : To aggregate values of a key using a function. "numTasks" is an optional parameter to specify number of reduce tasks	scala> val word1=fm.map(word=>(word,1)) scala> val wrdCnt=word1.reduceByKey(_+_) scala> wrdCnt.collect() **Result:** Array[(String, Int)] = Array((is,2), (It,1), (awesome,1), (Spark,1), (fun,1))
groupByKey([*numTasks*]) **Purpose:** To convert (K,V) to (K,Iterable<V>)	scala> val cntWrd = wrdCnt.map{case (word, count) => (count, word)} scala> cntWrd.groupByKey().collect() **Result:** Array[(Int, Iterable[String])] = Array((1,ArrayBuffer(It, awesome, Spark, fun)), (2,ArrayBuffer(is)))
distinct([*numTasks*]) **Purpose:** Eliminate duplicates from RDD	scala> fm.distinct().collect() **Result:** Array[String] = Array(is, It, awesome, Spark, fun)

Commonly Used Set Operations

Transformation & Purpose	Example & Result
union() **Purpose:** new RDD containing all elements from source RDD and argument.	Scala> val rdd1=sc.parallelize(List('A','B')) scala> val rdd2=sc.parallelize(List('B','C')) scala> rdd1.union(rdd2).collect() **Result:** Array[Char] = Array(A, B, B, C)
intersection() **Purpose:** new RDD containing all elements from source RDD and argument.	Scala> rdd1.intersection(rdd2).collect() **Result:** Array[Char] = Array(B)
cartesian() **Purpose:** new RDD cross product of all elements from source RDD and argument.	Scala> rdd1.cartesian(rdd2).collect() **Result:** Array[(Char, Char)] = Array((A,B), (A,C), (B,B), (B,C))
subtract() **Purpose:** new RDD created by removing data elements in source RDD in common with argument	scala> rdd1.subtract(rdd2).collect() **Result:** Array[Char] = Array(A)
join(RDD,[*numTasks*]) **Purpose:** When invoked on (K,V) and (K,W), this operation creates a new RDD of (K, (V,W))	scala> val personFruit = sc.parallelize(Seq(("Andy", "Apple"), ("Bob", "Banana"), ("Charlie", "Cherry"), ("Andy","Apricot"))) scala> val personSE = sc.parallelize(Seq(("Andy", "Google"), ("Bob", "Bing"), ("Charlie", "Yahoo"), ("Bob","AltaVista"))) scala> personFruit.join(personSE).collect() **Result:** Array[(String, (String, String))] = Array((Andy,(Apple,Google)), (Andy,(Apricot,Google)), (Charlie,(Cherry,Yahoo)), (Bob,(Banana,Bing)), (Bob,(Banana,AltaVista)))
cogroup(*RDD*,[*numTasks*]) **Purpose:** To convert (K,V) to (K,Iterable<V>)	scala> personFruit.cogroup(personSE).collect() **Result:** Array[(String, (Iterable[String], Iterable[String]))] = Array((Andy,(ArrayBuffer(Apple, Apricot),ArrayBuffer(Google))), (Charlie,(ArrayBuffer(Cherry),ArrayBuffer(Yahoo))), (Bob,(ArrayBuffer(Banana),ArrayBuffer(Bing, AltaVista))))

For a more detailed list of transformations, please refer to: http://spark.apache.org/docs/latest/programming-guide.html#transformations

Commonly Used Actions

Action & Purpose	Example & Result
count() **Purpose:** Get the number of data elements in the RDD	scala> val rdd = sc.parallelize(List('A','B','C')) scala> rdd.count() **Result:** Long = 3
collect() **Purpose:** get all the data elements in an RDD as an array	scala> val rdd = sc.parallelize(List('A','B','C')) scala> rdd.collect() **Result:** Array[Char] = Array(A, B, C)
reduce(*func*) **Purpose:** Aggregate the data elements in an RDD using this function which takes two arguments and returns one	scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.reduce(_+_) **Result:** Int = 10
take (*n*) **Purpose:** : fetch first n data elements in an RDD. Computed by driver program.	Scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.take(2) Result: Array[Int] = Array(1, 2)
foreach(*func*) **Purpose:** execute function for each data element in RDD. Usually used to update an accumulator(discussed later) or interacting with external systems.	Scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.foreach(x=>println("%s*10=%s". format(x,x*10))) **Result:** 1*10=10 4*10=40 3*10=30 2*10=20
first() **Purpose:** retrieves the first data element in RDD. Similar to take(1)	scala> val rdd = sc.parallelize(List(1,2,3,4)) scala> rdd.first() **Result:** Int = 1
saveAsTextFile(*path*) **Purpose:** Writes the content of RDD to a text file or a set of text files to local file system/ HDFS	scala> val hamlet = sc.textFile("/Users/akuntamukkala/temp/gutenburg.txt") scala> hamlet.filter(_.contains("Shakespeare")).saveAsTextFile("/Users/akuntamukkala/temp/filtered") **Result:** akuntamukkala@localhost~/temp/filtered$ ls _SUCCESS part-00000 part-00001

For a more detailed list of actions, please refer to:
http://spark.apache.org/docs/latest/programming-guide.html#actions

rdd persistence

One of the key capabilities of Apache Spark is persisting/caching an RDD in cluster memory. This speeds up iterative computation. The following table shows the various options Spark provides:

Storage Level	Purpose
MEMORY_ONLY (Default level)	This option stores RDD in available cluster memory as deserialized Java objects. Some partitions may not be cached if there is not enough cluster memory. Those partitions will be recalculated on the fly as needed.
MEMORY_AND_DISK	This option stores RDD as deserialized Java objects. If RDD does not fit in cluster memory, then store those partitions on the disk and read them as needed.
MEMORY_ONLY_SER	This options stores RDD as serialized Java objects (One byte array per partition). This is more CPU intensive but saves memory as it is more space efficient. Some partitions may not be cached. Those will be recalculated on the fly as needed.
MEMORY_ONLY_DISK_SER	This option is same as above except that disk is used when memory is not sufficient.
DISC_ONLY	This option stores the RDD only on the disk
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as other levels but partitions are replicated on 2 slave nodes

The above storage levels can be accessed by using the persist() operation on an RDD. The cache() operation is a convenient way of specifying the MEMORY_ONLY option

For a more detailed list of persistence options, please refer to:

http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence

Spark uses the Least Recently Used (LRU) algorithm to remove old, unused, cached RDDs to reclaim memory. It also provides a convenient unpersist() operation to force removal of cached/persisted RDDs.

Shared Variables

Accumulators

Spark provides a very handy way to avoid mutable counters and counter synchronization issues by providing accumulators. The accumulators are initialized on a Spark context with a default value. These accumulators are available on Slave nodes, but Slave nodes can't read them. Their only purpose is to fetch atomic updates and forward them to Master. Master is the only one that can read and compute the aggregate of all updates. For example, say we want to find the number of statements in a log file of log level 'error'…

```
akuntamukkala@localhost~/temp$ cat output.log error

warning
info
trace
error
info
info
scala> val nErrors=sc.accumulator(0.0)
scala> val logs = sc.textFile("/Users/akuntamukkala/temp/output. log")

scala> logs.filter(_.contains("error")).foreach(x=>nErrors+=1) scala> nErrors.value

Result: Int = 2
```

Broadcast Variables

It is common to perform join operations on RDDs to consolidate data by a certain key. In such cases, it is quite possible to have large datasets sent around to slave nodes that host the partitions to be joined. This presents a huge performance bottleneck, as network I/O is 100 times slower than RAM access. In order to mitigate this issue, Spark provides broadcast variables, which, as the name suggests, are broadcasted to slave nodes. The RDD operations on the nodes can quickly access the broadcast variable value. For example, say we want to calculate the shipping cost of all line items in a file. We have a static look-up table that specifies cost per shipping type. This look-up table can be a broadcast variable.

```
akuntamukkala@localhost~/temp$ cat packagesToShip.txt ground

express
media
priority
priority
ground
express
media
scala> val map = sc.parallelize(Seq(("ground",1),("med",2),
("priority",5),("express",10))).collect().toMap
map: scala.collection.immutable.Map[String,Int] = Map(ground -> 1, media -> 2, priority -
> 5, express -> 10) scala> val bcMailRates = sc.broadcast(map)
```

In the above command, we create a broadcast variable, a map containing cost by class of service.

```
scala> val pts = sc.textFile("/Users/akuntamukkala/temp/ packagesToShip.txt")

scala> pts.map(shipType=>(shipType,1)).reduceByKey(_+_). map{case
(shipType,nPackages)=>(shipType,nPackages*bcMailRates. value(shipType))}.collect()
```

In the above command we calculate shipping cost by looking up mailing rates from broadcast variable.

```
Array[(String, Int)] = Array((priority,10), (express,20), (media,4), (ground,2))

scala> val shippingCost=sc.accumulator(0.0) scala>
pts.map(x=>(x,1)).reduceByKey(_+_).map{case
(x,y)=>(x,y*bcMailRates.value(x))}.foreach(v=>shippingCost+=v._2) scala>
shippingCost.value
Result: Double = 36.0
```

In the above command we use accumulator to calculate total cost to ship. The following presentation provides more information:

http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-amp-camp-2012-advanced-spark.pdf

Spark SQL

Spark SQL provides a convenient way to run interactive queries over large data sets using Spark Engine, using a special type of RDD called SchemaRDD. SchemaRDDs can be created from existing RDDs or other external data formats such as Parquet files, JSON data or by running HQL on Hive. SchemaRDD is similar to a table in RDBMS. Once data are in SchemaRDD, the Spark engine will unify it with batch and streaming use cases. Spark SQL provides two types of contexts: SQLContext & HiveContext that extend SparkContext functionality.

SQLContext provides access to a simple SQL parser whereas HiveContext provides access to HiveQL parser. HiveContext enables enterprises to leverage their existing Hive infrastructure.

Let's see a simple example using SQLContext.

Say we have the following '|' delimited file containing customer data:

```
John Smith|38|M|201 East Heading Way #2203,Irving, TX,75063 Liana
Dole|22|F|1023 West Feeder Rd, Plano,TX,75093
Craig Wolf|34|M|75942 Border Trail,Fort Worth,TX,75108 John
Ledger|28|M|203 Galaxy Way,Paris, TX,75461 Joe Graham|40|M|5023
Silicon Rd,London,TX,76854
```

Define Scala case class to represent each row:

```
case class Customer(name:String,age:Int,gender:String,address:
String)
```

The following code snippet shows how to create SQLContext using SparkContext, read the input file, convert each line into a record in SchemaRDD and then query in simple SQL to find male consumers under the age of 30:

```
val sparkConf = new SparkConf().setAppName("Customers")
val sc = new SparkContext(sparkConf)
val sqlContext = new SQLContext(sc)
val r = sc.textFile("/Users/akuntamukkala/temp/customers.txt")
val records = r.map(_.split('|'))
val c = records.map(r=>Customer(r(0),r(1).trim.toInt,r(2),r(3)))
c.registerAsTable("customers")

sqlContext.sql("select * from customers where gender='M' and age <
30").collect().foreach(println)
Result:
[John Ledger,28,M,203 Galaxy Way,Paris, TX,75461]
```

For more practical examples using SQL & HiveQL, please refer to the following links:

https://spark.apache.org/docs/latest/sql-programming-guide.html
https://databricks-training.s3.amazonaws.com/data-exploration-using-spark-sql.html

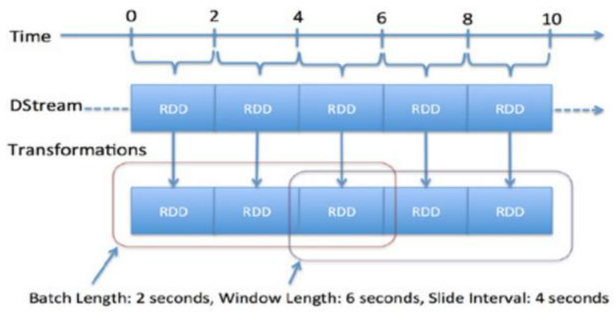

Time
0
2
4
6
8
10
DStream
RDD
Transformations
Batch Length: 2 seconds, Window Length: 6 seconds, Slide Interval: 4 seconds

Figure 11

Spark Streaming

Spark Streaming provides a scalable, fault tolerant, efficient way of processing streaming data using Spark's simple programming model. It converts streaming data into "micro" batches, which enable Spark's batch programming model to be applied in Streaming use cases. This unified programming model makes it easy to combine batch and interactive data processing with streaming. Figure 10 shows how Spark Streaming can be used to analyze data feeds from multitudes of data sources.

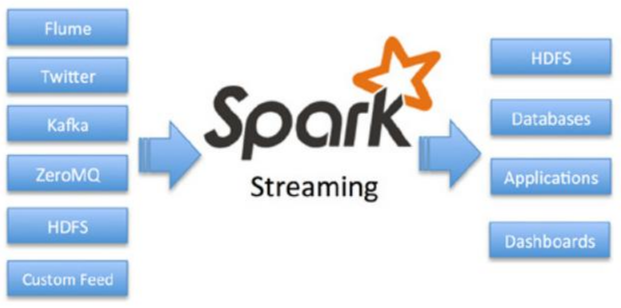


Figure 12

The core abstraction in Spark Streaming is Discretized Stream (DStream). DStream is a sequence of RDDs. Each RDD contains data received in a configurable interval of time. Figure 12 shows how Spark Streaming creates a DStream by converting incoming data into a sequence of RDDs. Each RDD contains streaming data received every 2 seconds as defined by interval length. This can be as small as ½ second, so latency for processing time can be under 1 second.

Spark Streaming also provides sophisticated window operators, which help with running efficient computation on a collection of RDDs in a rolling window of time. DStream exposes an API, which contains operators (transformations and output operators) that are applied on constituent RDDs. Let's try and understand this using a simple example given in Spark Streaming download. Say, you want to find the trending hash tags in your Twitter stream. Please refer to the following example to find the complete code snippet:

```
spark-1.0.1/examples/src/main/scala/org/apache/spark/examples/
streaming/TwitterPopularTags.scala
val sparkConf = new SparkConf().setAppName("TwitterPopularTags")
val ssc = new StreamingContext(sparkConf, Seconds(2))
val stream = TwitterUtils.createStream(ssc, None, filters)
```

The above snippet is setting up Spark Streaming Context. Spark Streaming will create an RDD in DStream containing Tweets retrieved every two seconds.

```
val hashTags = stream.flatMap(status => status.getText.split(" ").filter(_.startsWith("#")))
```

The above snippet converts the Tweets into a sequence of words, then filters only those beginning with a #.

```
val topCounts60 = hashTags.map((_, 1)).reduceByKeyAndWindow(_
+  _, Seconds(60)).map{case (topic, count) => (count, topic)}.
transform(_.sortByKey(false))
```

The above snippet shows how to calculate a rolling aggregate of the number of times a hashtag was mentioned in a window of 60 seconds.

```
topCounts60.foreachRDD(rdd => {
    val topList = rdd.take(10)
    println("\nPopular topics in last 60 seconds (%s
            total):".format(rdd.count())) topList.foreach{case (count, tag) =>
    println("%s (%s
            tweets)".format(tag, count))}
})
```

The above snippet shows how to extract the top ten trending Tweets and then print them out.

```
ssc.start()
```

The above snippet instructs the Spark Streaming Context to start retrieving Tweets. Let's look at a few popular operations. Assume that we are reading streaming text from a socket:

```
val lines = ssc.socketTextStream("localhost", 9999,
StorageLevel.MEMORY_AND_DISK_SER)
```

Transformation & Purpose	Example & Result	
map(*func*) **Purpose:** Create a new DStream by applying this function to all constituent RDDS in DStream	lines.map(x=>x.toInt*10).print()	
	prompt>nc –lk 9999 12 34	Output: 120 340
flatMap(*func*) **Purpose:** Same as map, but the mapping function can output zero or more items	lines.flatMap(_.split(" ")).print()	
	prompt>nc –lk 9999 Spark is fun	Output: Spark is fun
count() **Purpose:** create a DStream of RDDs containing a count of the number of data elements	lines.flatMap(_.split(" ")).count()	
	prompt>nc –lk 9999 say hello to spark	Output: 4
reduce(func) **Purpose:** Same as count, but instead of count, the value is derived by applying the function	lines.map(x=>x.toInt).reduce(_+_).print()	
	prompt>nc –lk 9999 1 3 5 7	Output: 16
countByValue() **Purpose:** Same as map, but the mapping function can output zero or more items	lines.countByValue().print()	
	prompt>nc –lk 9999 spark spark is fun fun	Output: (is,1) (spark,2) (fun,2)
reduceByKey(*func*,[*numTasks*])	val words = lines.flatMap(_.split(" ")) val wordCounts = words.map(x => (x, 1)).reduceByKey(_+_) wordCounts.print()	
	prompt>nc –lk 9999 spark is fun fun fun	Output: (is,1) (spark,1) (fun,3)

The *following* example shows how Apache Spark combines Spark batch with Spark Streaming. This is a powerful capability for an all-in-one technology stack. In this example, we read a file containing brand names and filter those streaming data sets that contain any of the brand names in the file.

transform(*func*)
Purpose: Creates a new DStream by applying RDD->RDD transformation to all RDDs in DStream.

brandNames.txt
coke
nike
sprite
reebok

```
val sparkConf = new SparkConf().
setAppName("NetworkWordCount")
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc,
Seconds(10))
val lines = ssc.
socketTextStream("localhost" 9999,
StorageLevel.MEMORY_AND_DISK_SER)
val brands = sc.textFile("/Users/
akuntamukkala/temp/brandNames.txt")
lines.transform(rdd=> {
            rdd.intersection(brands)
}).print()
```

```
prompt>nc –lk
9999                     Output:
msft                     sprite
apple
nike                     nike
sprite
ibm
```

updateStateByKey(*func*)
Purpose: Creates a new DStream where the value of each key is updated by applying given function.

Please refer to the StatefulNetworkCount example in Spark Streaming.
This helps with computing a running aggregate of the total number of times a word has occurred.

Common Window Operations

window(windowLength, slideInterval)
Purpose: Returns a new DStream computed from windowed batches of source DStream

```
val win = lines.
window(Seconds(30),Seconds(10));
win.foreachRDD(rdd => {
    rdd.foreach(x=>println(x+ " "))
})
```

```
prompt>nc –lk 9999          Output:
10 (0th second)             10
20 (10 seconds later)       10 20
30 (20 seconds later)       20 10 30
40 (30 seconds later)       20 30 40 (drops 10)
```

countByWindow(windowLength, slideInterval)
Purpose: Returns a new sliding window count of elements in a steam

```
lines.countByWindow(Seconds(30),
Seconds(10)).print()
```

```
                             Output:
prompt>nc –lk 9999           1
10 (0th second)              2
20 (10 seconds later)        3
30 (20 seconds later)        3
40 (30 seconds later)
```

For additional transformation operators, please refer to:
http://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations

Spark Streaming has powerful output operators. We already saw foreachRDD() in above example. For others, please refer to:

http://spark.apache.org/docs/latest/streaming-programming-guide.html#output-operations

- Wikipedia article (good):
  http://en.wikipedia.org/wiki/Apache_Spark
- Launching a Spark cluster on EC2:
  http://ampcamp.berkeley.edu/exercises-strata-conf-2013/launching-a-cluster.html
- Quick start:
  https://spark.apache.org/docs/1.0.1/quick-start.html
- The Spark platform provides MLLib(machine learning) and GraphX(graph algorithms). The following links provide more information:

- https://spark.apache.org/docs/latest/mllib-guide.html
- https://spark.apache.org/docs/1.0.1/graphx-programming-guide.html