skillspeed
for the serious learner

Apache Spark and Scala

Module 2: Understanding RDDs

Course Topics

Module 1

Getting Started / Introduction to Scala

Module 2

RDD and Spark Streaming

Module 3

Scala Basics

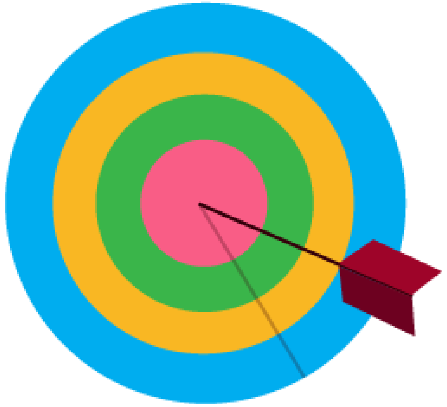
Module 4

SparkSQL – Real-time Analysis

Session Objectives

This session will help you to:

- Exploring RDDs
- Understand:
  - Data Loading in RDD
  - Manipulating RDD’s
  - RDD functions
  - Fault Recovery
  - RDD Dependency’s
  - RDD JOB scheduler
  - Memory Management
  - RDDs in distributed environments
  - Hadoop StreamingSpark + Yarn integration

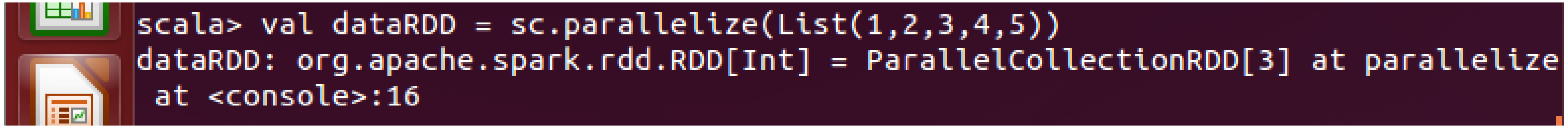


RDD (Resilient Distributed Datasets)

- Resilient distributed dataset (RDD), is a fault-tolerant collection of elements that can be operated on in parallel.
- Restricted form of distributed shared memory – read-only
- Partitioned collection of records – can only be built through coarse-grained deterministic transformations
  - Transformations from other RDDs
  - Express computation by – defining RDDs
- RDDs can be created from any data source e.g. Scala collection, local file system, Hadoop, Amazon S3, HBase table etc.
- Spark supports text files, Sequence Files, and any other Hadoop Input Format, and can also take a directory or a glob
- Even though the RDDs are defined, they don’t contain any data
- The computation to create the data in a RDD is only done when the data is referenced;
- Example: Caching results or writing out the RDD

Data Loading in RDD

- RDD is re-computed every time when it is materialized
- So it is a good idea to improve performance by caching a RDD if it is accessed frequently
- One of the easiest way to load data in RDD is to load from a Scala collection
- SparkContext provides parallelize function, which converts the Scala collection into the RDD of the same type



```
scala> val dataRDD = sc.parallelize(List(1,2,3,4,5))
dataRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize
 at <console>:16
```

Data Loading in RDD (Cont’d)

- Another simple way of loading data is loading text from a file
- In local mode (single node operation), you just need to specify the file location
- In distributed mode, the pre-requisite is the availability of the file in all nodes of the cluster
- Spark’s addFile() functionality is used to copy the file to designated location to all machines in the cluster

```
import spark.SparkFiles;
    ...
    sc.addFile(“Sample.txt")
    val inFile = sc.textFile(SparkFiles.get(“Sample.txt "))
```

Note: pyspark doesn’t currently support advanced data loads like loading data from sequence files, HBase etc.

Manipulating RDDs

- Manipulating RDDs is quite similar to Scala collections manipulation
- Manipulating your RDD in Scala is quite simple, especially if you are familiar with Scala's collection library
- Many of the standard functions are available directly on Spark's RDDs with the primary catch being that they are immutable
- Programmer need not to worry about the RDDs being executed on same machine or multiple machines
- The hallmark functions of map and reduce are available by default
- The map and other Spark functions DO NOT transform the existing elements, they always create a new RDD with transformed elements (Immutability in Action!)

```
f(a,b) == f(b,a) and f(a, f(b,c)) == f(f(a,b), c)
```

For example, to sum all the elements,

```
use rdd.reduce(x,y => x+y)
              ( or )
rdd.reduce(new Function2<Integer, Integer, Integer>()
{
public Integer call(Integer x, Integer y)
 { return x+y;  } }
```

Scala RDD Functions

Function	Parameter options	Explanation	Return Type
foldByKey	(zeroValue) (func(V,V)=>V)	Merges the values using the provided function. Unlike a traditional fold function over a list, the zeroValue can be added an arbitrary number of times	RDD[K,V]
reduceByKey	(func(V,V)=>V, numTasks)	Parallel version of reduce that merges the values for each key using the provided function and returns an RDD	RDD[K,V]
groupByKey	(numPartitions)	Groups elements together by key	RDD[K,Seq[V]]

Pair RDD Functions

Function	Parameter options	Explanation	Return Type
lookup	(key: K)	Looks up a specific element in the RDD. Uses the RDD's partitioner to figure out which partition(s) to look at	Seq[V]
mapValues	(f: v=>u)	A specialized version of map for PairRDD when you only want to change the value of the key-value pair. If you need to make your changes based on both key and value, you must use one of the normal RDD Map functions	RDD[K,U]
collectAsMap	()	Takes an RDD and returns a concrete map. Your RDD must be able to fit into the memory	Map[K, V]

Pair RDD Functions (Cont’d)

Function	Parameter options	Explanation	Return Type
countByKey	()	Counts the number of elements for each key in RDD	Map[K, Long]
partitionBy	(partitioner:P, mapSideCombine: Boolean)	Returns a new RDD with the same data but partitioned by the new Partitioner, and mapSideCombine controls Spark group values with the same key together before repartitioning. Defaults to false	RDD[K,V]
flatMapValues	(f:V => TraversableOnce[U])	Similar to mapValues. A specialized version of flatMap for PairRDDs when you only want to change the value of the key-value pair. Takes the provided Map function and applies it to the value. The resulting sequence is then "flattened“	RDD[K, U]

Double RDD Functions

Spark provides a number of Utility functions for the RDDs if they consist of double data type

Function	Argument	Returns
mean	()	Average of RDDs elements
Sum	()	Sum of Elements
variance	()	Variance of RDD elements
Stats	()	Mean, Variance and Count as Stats Counter

General RDD Functions

Function	Argument	Returns
cache	()	Caches an RDD reused without re-computing. Same as persist(StorageLevel. MEMORY_ONLY)
collect	()	An array of element in RDD
count	()	Number of elements in RDD
countByValue	()	A map of value to the number of times the value occurs.
distinct	()	RDD containing only distinct elements.
filter	(f: T => Boolean)	RDD containing elements only matching f.
foreach	(f: T => Unit)	Applies the function f to each element.
persist	() ,(newLevel: StorageLevel)	Sets the RDD storage level, which can cause the RDD to be stored after it is computed. Different StorageLevels can be seen in StorageLevel.

General RDD Functions (Cont’d)

Function	Argument	Returns
sample	(fraction: double)	RDD of that fraction
toDebugString	()	A handy function that outputs the recursive deps of the RDD
count	()	Number of elements in RDD
unpersist	()	Remove all the persistent blocks of the RDD from the memory/disk
union	(other: RDD[T])	An RDD containing elements of both RDDs. Duplicates are not removed

Example: HadoopRDD

Partition	=	One per HDFS block
Dependencies	=	None
Compute (part)	=	Read corresponding block
Preferred location (part)	=	HDFS block location
Partitioner	=	None

Example: FilteredRDD

Partition	=	Same as parent RDD
Dependencies	=	“One-to-one” on parent
Compute (part)	=	Compute parent and filter it
Preferred location(part)	=	None (ask parent)
Practitioner	=	None

RDD Graph

Dataset-level view:

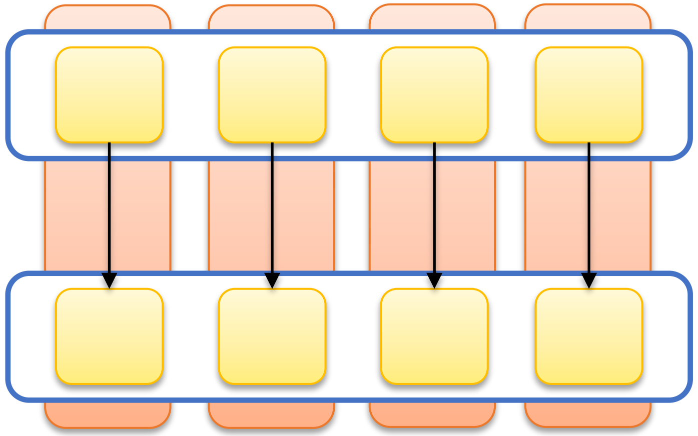
File:

HadoopRDD
Path=hdfs://...

Errors:

FIlteredRDD
Func=_.contains(...)
shouldCache=true

Partition-level view:



Task 1 Task 2 ...

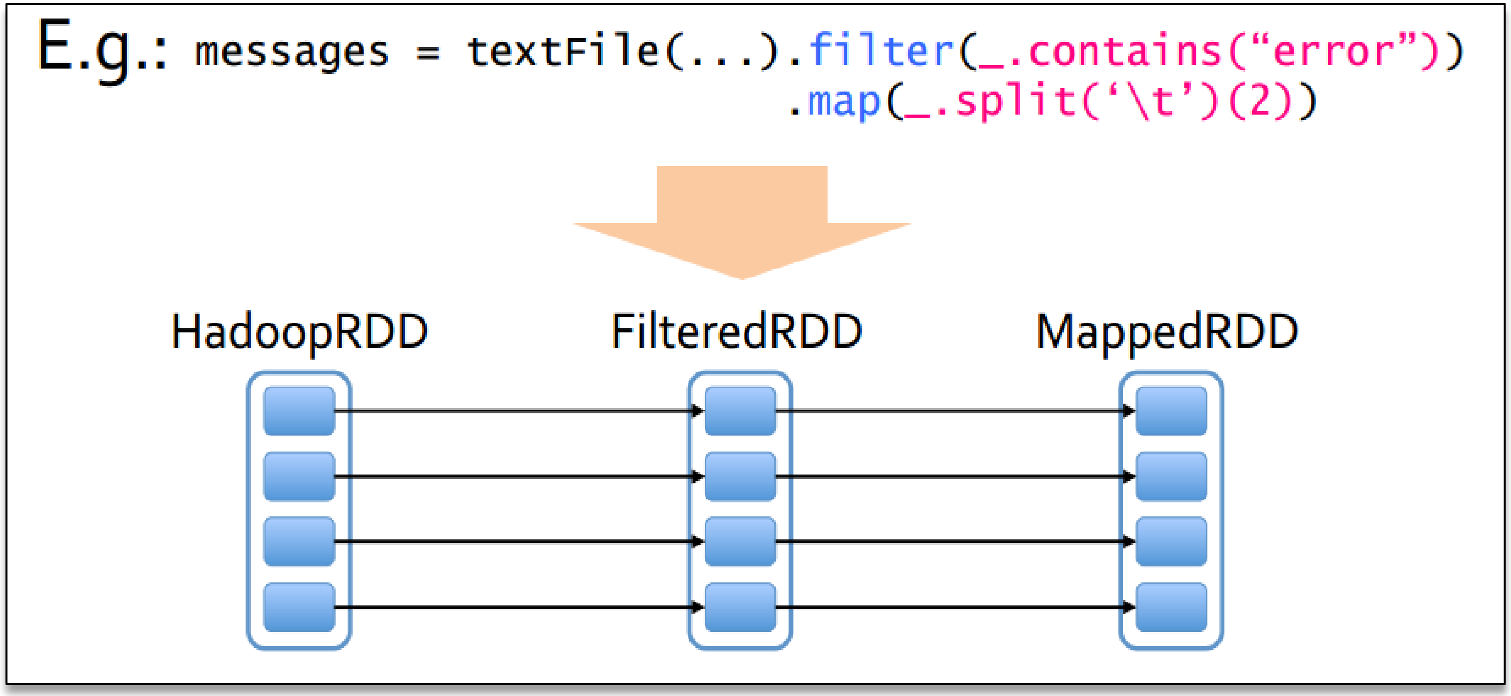
Example: JoinedRDD

Partition	=	One per reduce task
Dependencies	=	"Shuffle" on each parent
Compute (partition)	=	Read and join shuffle data
Preferred location(part)	=	None
Partitioner	=	Hashpartitioner(numtasks)

Spark will now know this data is hashed!

Fault Recovery

Efficient fault recovery using lineage – log one operation to apply to many elements (lineage) – recomputed lost partitions on failure



E.g.:

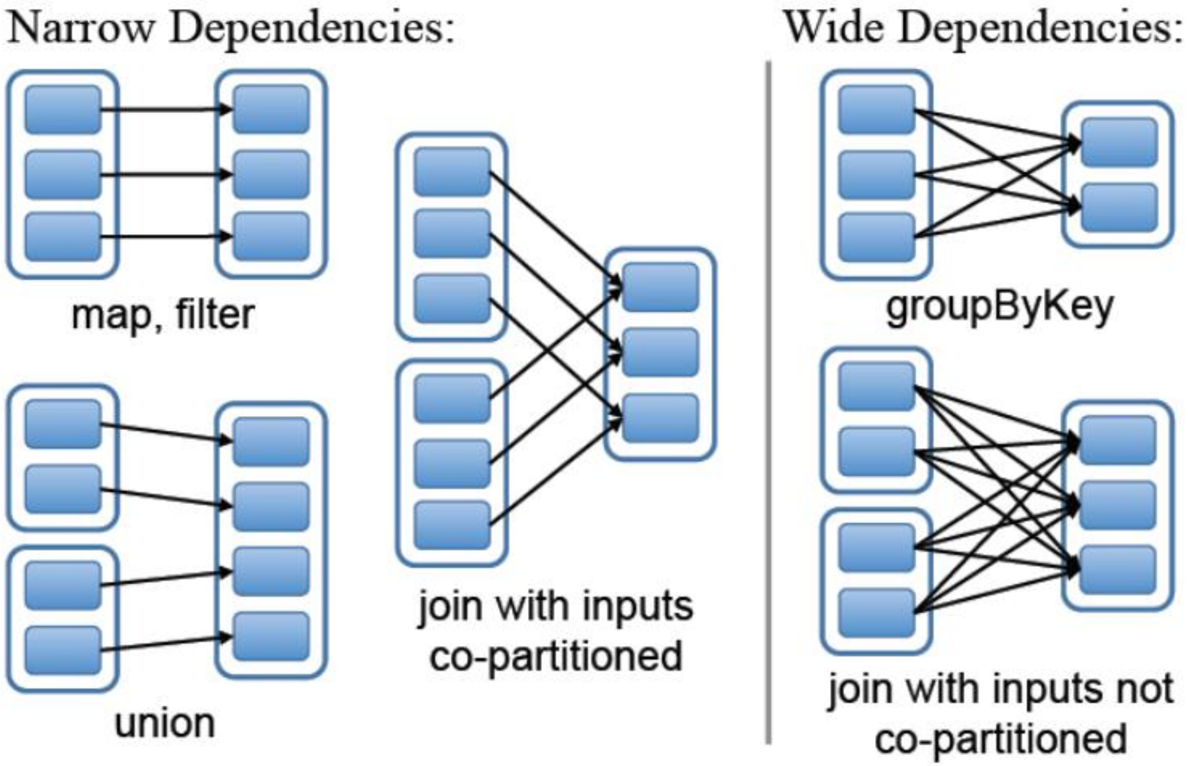
```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```

Advantages of the RDD Model

- Efficient fault recovery – fine-grained and low-overhead using lineage
- Immutable nature can mitigate stragglers – backup tasks to mitigate stragglers
- Graceful degradation when RAM is not enough

RDD Dependencies

- Narrow dependencies – each partition of the parent RDD is used by at most one partition of the child RDD
- Wide dependencies – multiple child partitions may depend on it


Narrow Dependencies:
map, filter
union
join with inputs co-partitioned
Wide Dependencies:
groupByKey
join with inputs not co-partitioned

Supported RDD Operations – Transformations

- Transformations create a new dataset from an existing one
- All transformations in Spark are lazy: they do not compute their results right away
- Instead they remember the transformations applied to some base dataset
- This helps in:
  - Optimizing the required calculations
  - Recover from lost data partitions

Transformations (Cont'd)

Transformation	Meaning
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument
groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs
reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V

Transformations (Cont'd)

Transformation	Meaning
join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument
coalesce(numPartitions)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset
cogroup(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples

Supported RDD Operations – Actions

Spark forces the calculations for execution only when actions are invoked on the RDDs

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel
first()	Return the first element of the dataset (similar to take(1))
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator
count()	Return the number of elements in the dataset

Actions (Cont’d)

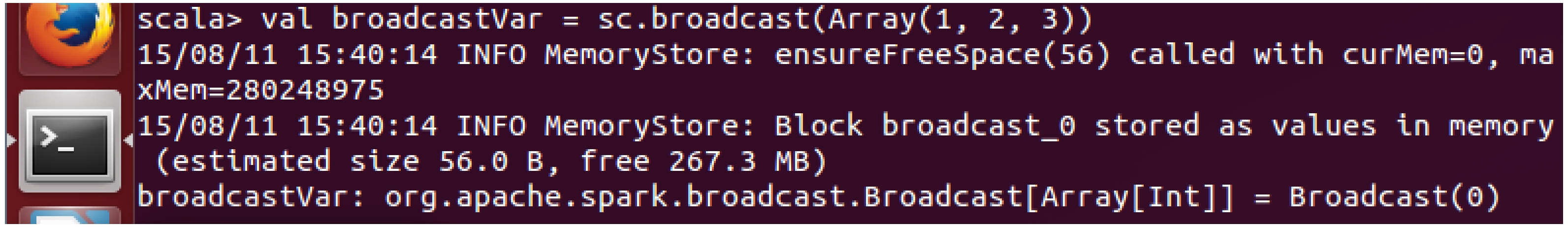
Action	Meaning
Collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data
saveAsSequenceFile(path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local file system, HDFS or any other Hadoop-supported file system
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator variable
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key

Memory Management

- Three options for persistent RDDs – in-memory storage as desterilized Java objects – in-memory storage as serialized data – on-disk storage
- LRU eviction policy at the level of RDDs – when there's not enough memory, evict a partition from the least recently accessed RDD

Shared Variables – Broadcast Variables

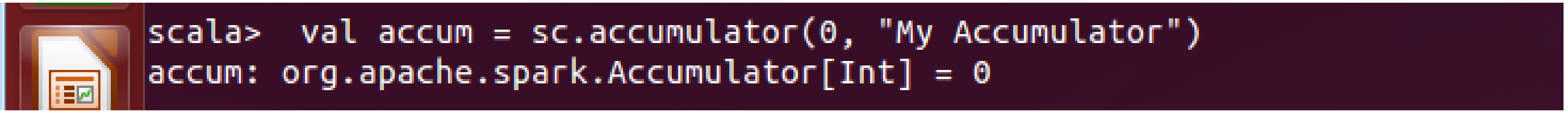
- Normally a function passed over a Spark transformation or Action works on separate copies of the variables of the function
- The variables are local to each machine and updates to these variables are not propagated back to the driver program
- Spark provides two types of shared variables:
  - Broadcast variables
  - Accumulators
- Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
- For example, to give every node a copy of a large input dataset efficiently Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost



```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
15/08/11 15:40:14 INFO MemoryStore: ensureFreeSpace(56) called with curMem=0, ma
xMem=280248975
15/08/11 15:40:14 INFO MemoryStore: Block broadcast_0 stored as values in memory
 (estimated size 56.0 B, free 267.3 MB)
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
```

Shared Variables – Accumulators

- Accumulators are variables that can only be “added” to through an associative operation
- Used to implement counters and sums, efficiently in parallel
- Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types
- Only the driver program can read an accumulator’s value, not the tasks

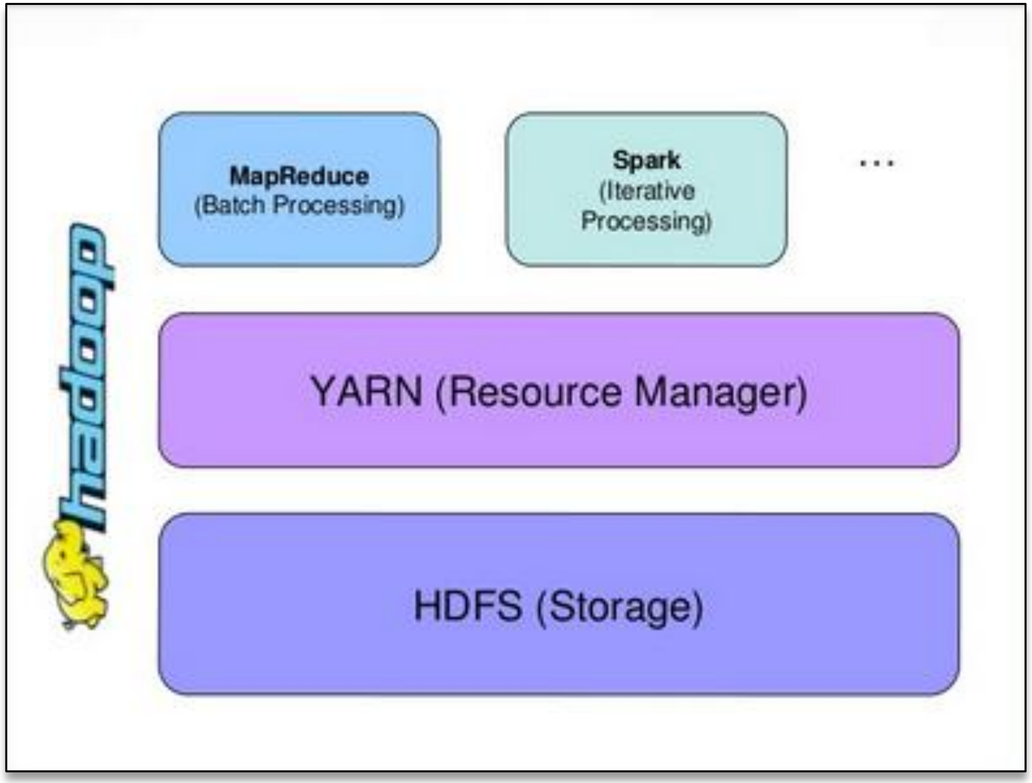


```
scala>  val accum = sc.accumulator(0, "My Accumulator")
accum: org.apache.spark.Accumulator[Int] = 0
```

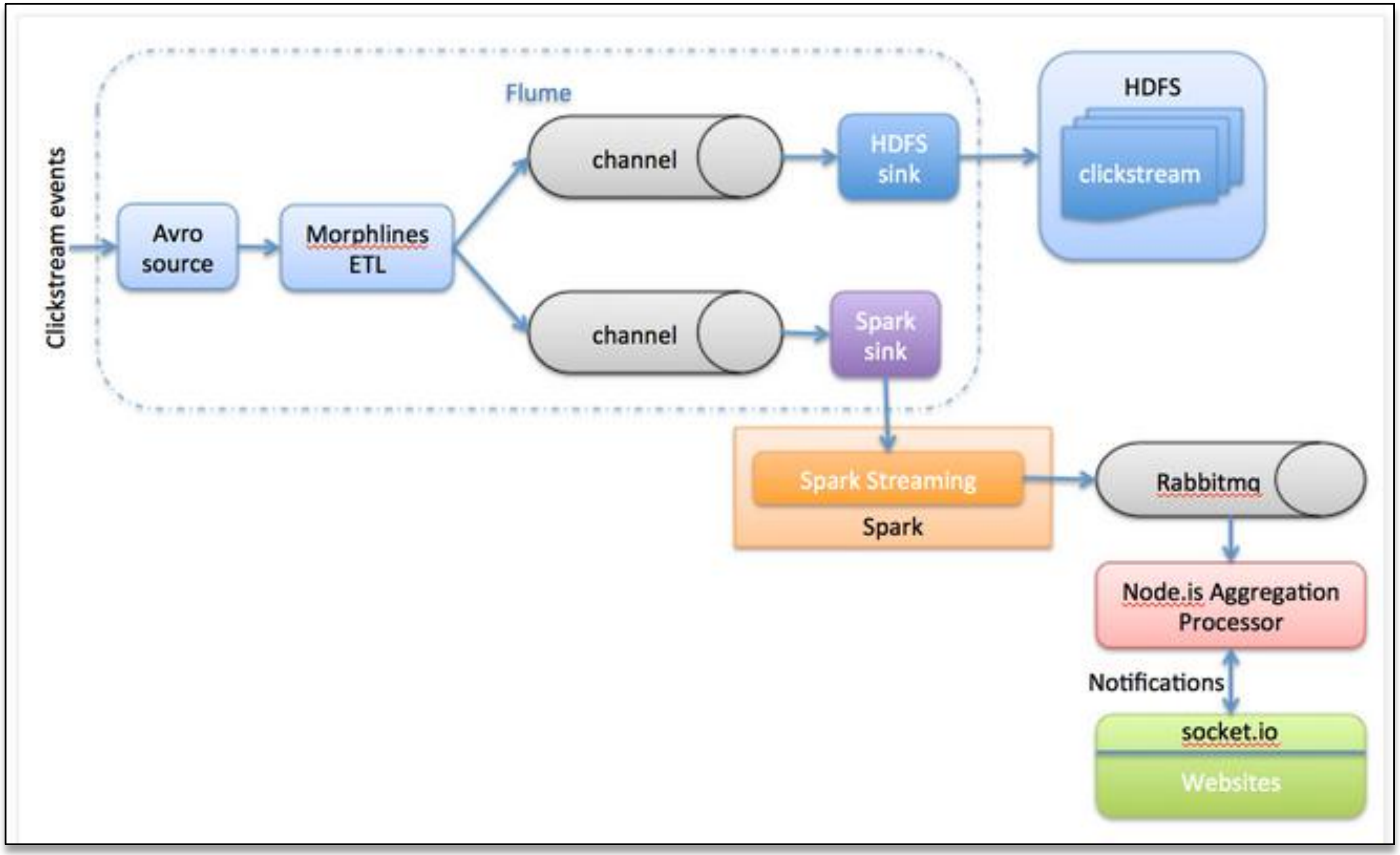
Spark with YARN

- Spark can be used along with Hadoop 2.x
- Spark can use YARN as the Cluster resource Manager in Spark – YARN mode
- Spark has a different build for YARN specific integration
- YARN spawns the spark program as YARN process, where the App Master process is actually the driver program and YARN child processes are the Spark workers
- This integration is preferable mode if the data size per node is way more than the memory available to cache the RDDs
- On lower volumes of the data per node, launching Spark without YARN yields better results

Spark with YARN



Stream Processing Pipeline

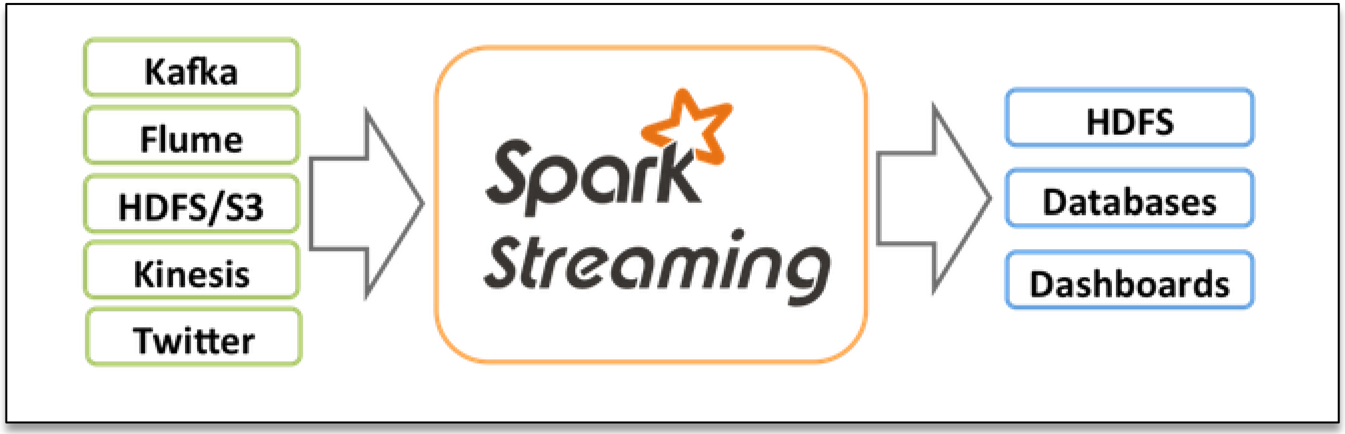


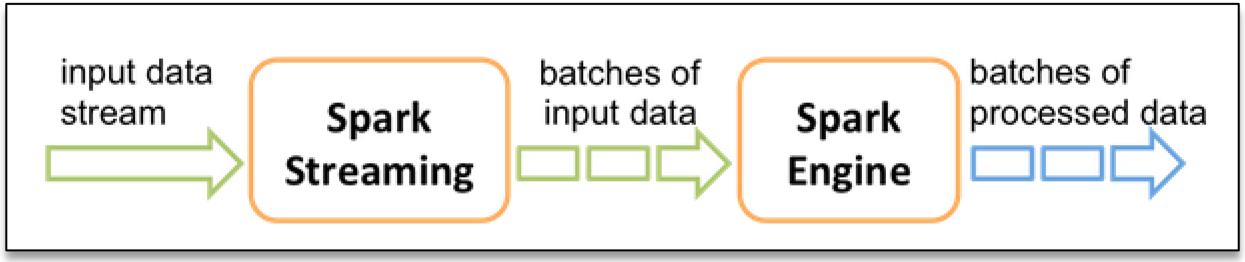
Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams
- Data ingestion can happen from many data sources e.g. Streams, Flume, Kafka or ZeroMQ
- Processed data can be sent to many systems like file systems, databases or app dashboards
- Internally spark streaming converts live input data streams into stream batches
- These batches are processed by Spark engine to generate final stream of batch results

Spark Streaming (Cont'd)

- Spark provides a high level abstraction, discretized stream or Dstream, representing a continuous stream of data
- A DStream can be represented as a sequence of RDDs





Questions

thank you!