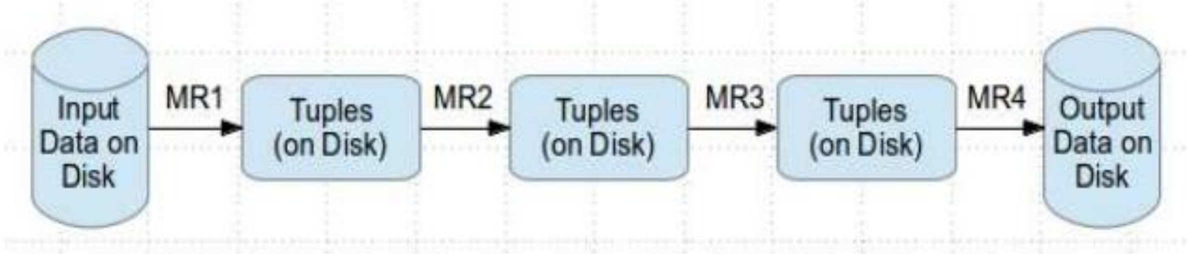
APACHE SPARK

IN-MEMORY DATA PROCESSING

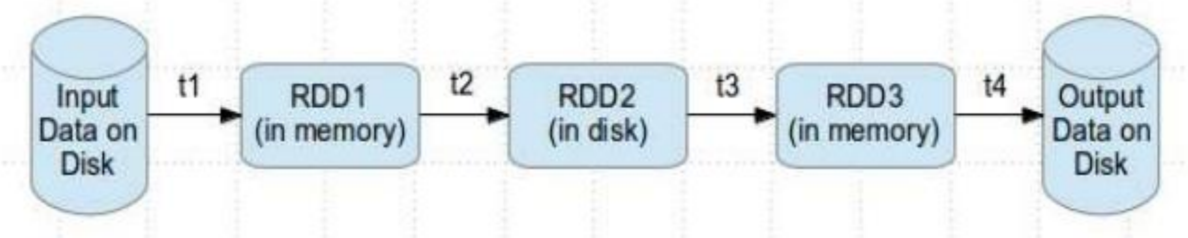
Why Spark ?

- Most of Machine Learning Algorithms are iterative because each iteration can improve the results
- With Disk based approach each iteration's output is written to disk making it slow

Hadoop execution flow



Spark execution flow



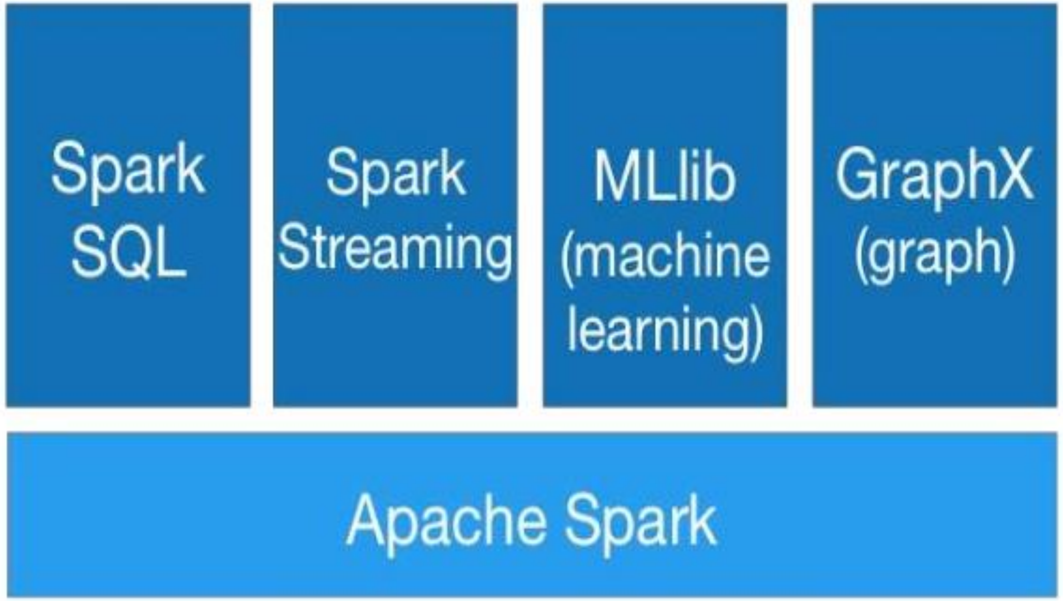
http://www.wiziq.com/blog/hype-around-apache-spark/

About Apache Spark

- Initially started at UC Berkeley in 2009
- Fast and general purpose cluster computing system
- 10x (on disk) - 100x (In-Memory) faster
- Most popular for running *Iterative Machine Learning Algorithms.*
- Provides high level APIs in
    - Java
    - Scala
    - Python
- Integration with Hadoop and its eco-system and can **read existing data.**
- http://spark.apache.org/

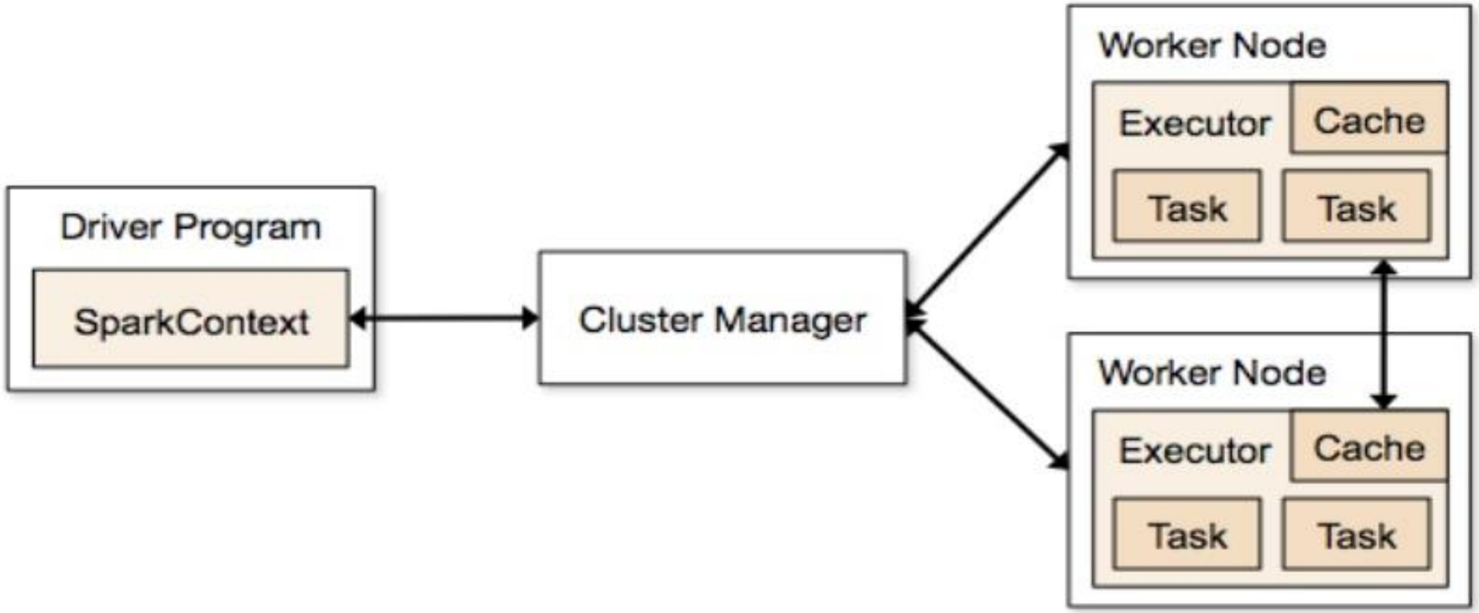
Spark Stack

- Spark SQL
  - For SQL and unstructured data processing
- MLib
  - Machine Learning Algorithms
- GraphX
  - Graph Processing
- Spark Streaming
  - stream processing of live data streams



http://spark.apache.org

Execution Flow



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

Terminology

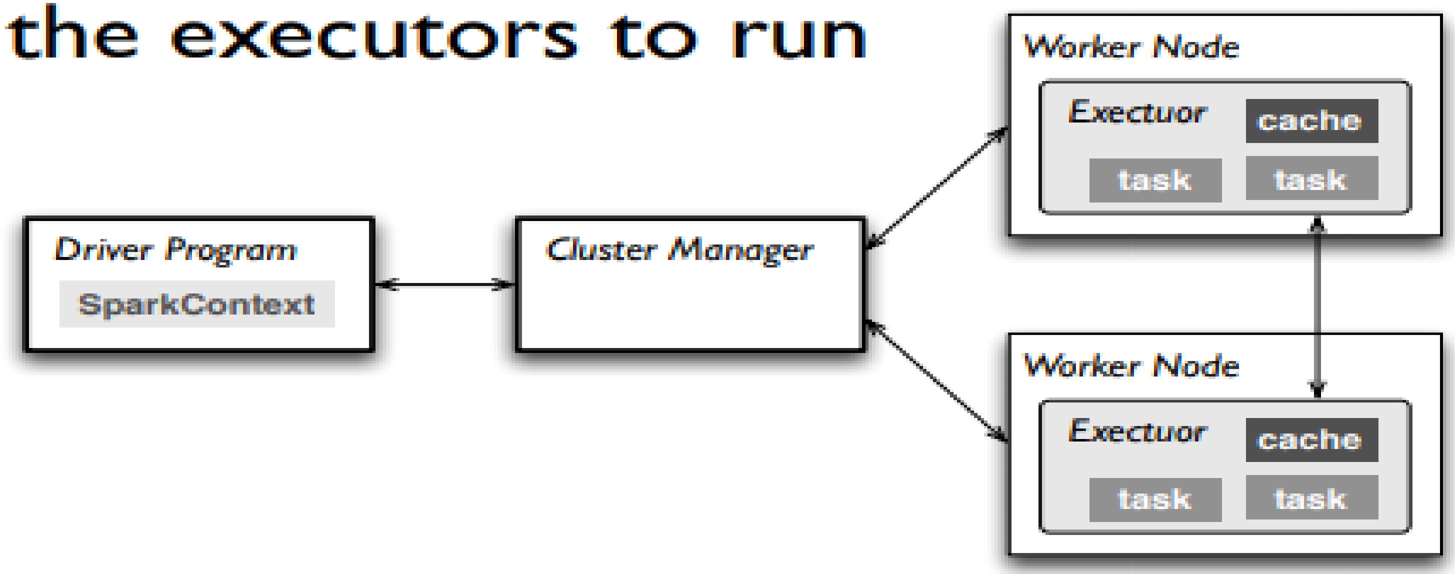
- **Application Jar**
  - User Program and its dependencies except Hadoop & Spark Jars bundled into a Jar file
- **Driver Program**
  - The process to start the execution (main() function)
- **Cluster Manager**
  - An external service to manage resources on the cluster (standalone manager, YARN, Apache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

Spark Essentials: *Master*

1. connects to a *cluster manager* which allocate resources across applications
2. acquires *executors* on cluster nodes – worker processes to run computations and store data
3. sends *app code* to the executors
4. sends *tasks* for the executors to run



Spark Essentials: *RDD*

Resilient **D**istributed **D**atasets (RDD) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel

There are currently two types:

- *parallelized collections* – take an existing Scala collection and run functions on it in parallel
- *Hadoop datasets* – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop

Spark Essentials: *RDD*

- two types of operations on RDDs: *transformations* and *actions*
- transformations are lazy (not computed immediately)
- the transformed RDD gets recomputed when an action is run on it (default)
- however, an RDD can be *persisted* into storage in memory or disk

Features of RDD

In-memory Computation

Spark RDDs have a provision of in-memory computation. It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).

Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program.

Fault Tolerance

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure.

They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself.

Features of RDD

Immutability

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

Partitioning

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

Persistence

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

Coarse-grained Operations

It applies to all elements in datasets through maps or filter or group by operation.

Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD.

The **DAGScheduler** places the partitions in such a way that task is close to data as much as possible. Thus, speed up computation.

Spark Essentials: *RDD*

Scala:

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e
```

Python:

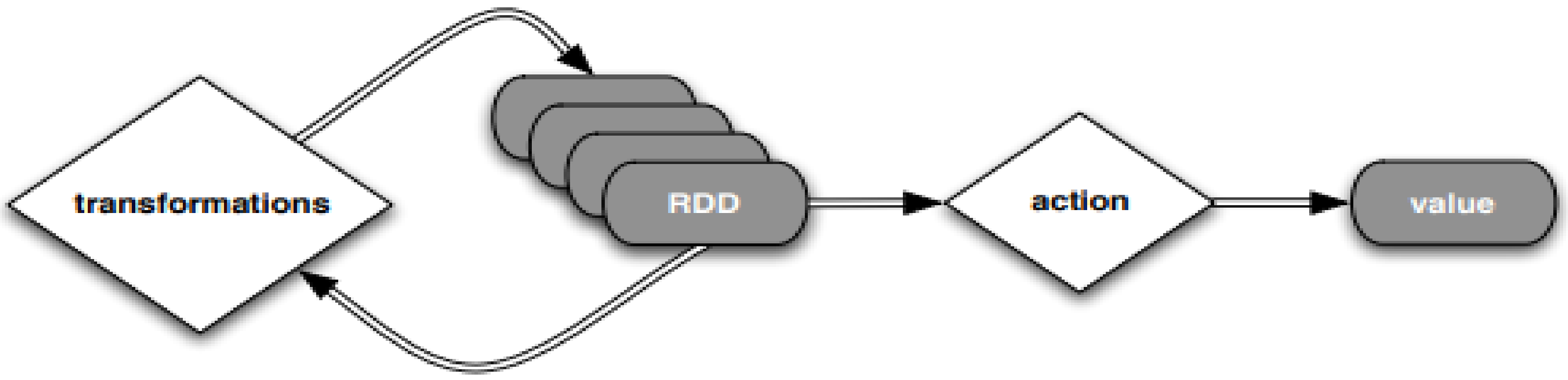
```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

Spark Essentials: *RDD*

Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.

Spark supports text files, SequenceFiles, and any other Hadoop `InputFormat`, and can also take a directory or a glob (e.g. `/data/201404*`)



Spark Essentials: *RDD*

Scala:

```
scala> val distFile = sc.textFile("README.md")
distFile: spark.RDD[String] = spark.HadoopRDD@1d4cee08
```

Python:

```
>>> distFile = sc.textFile("README.md")
14/04/19 23:42:40 INFO storage.MemoryStore: ensureFreeSpace(36827) called
with curMem=0, maxMem=318111744
14/04/19 23:42:40 INFO storage.MemoryStore: Block broadcast_0 stored as
values to memory (estimated size 36.0 KB, free 303.3 MB)
>>> distFile
MappedRDD[2] at textFile at NativeMethodAccessorImpl.java:-2
```

Spark Essentials: *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

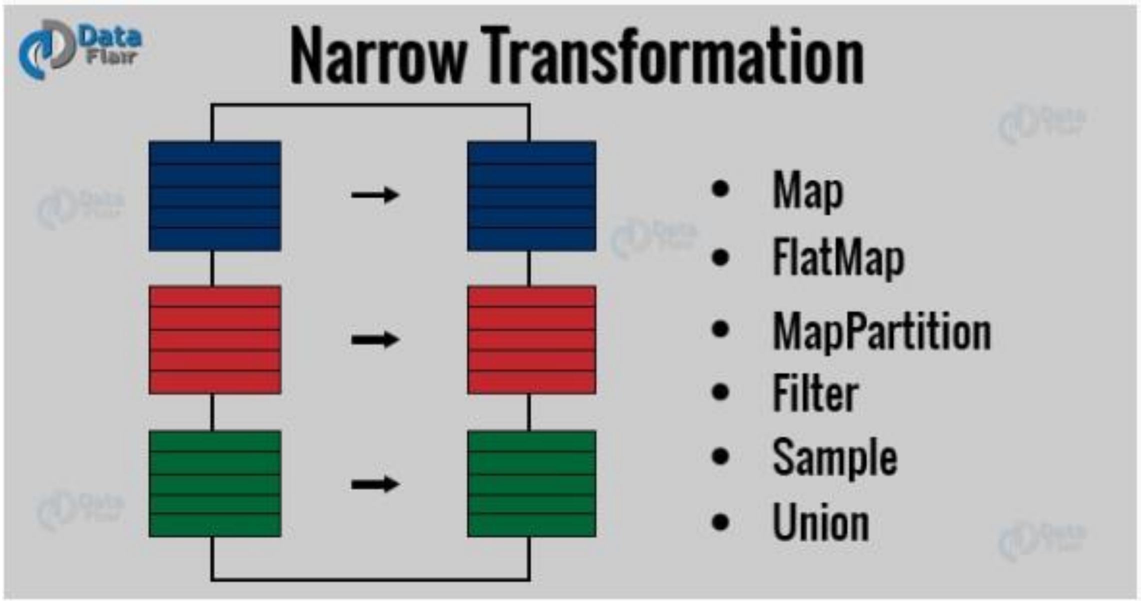
- optimize the required calculations
- recover from lost data partitions

Transformations

There are two kinds of transformations: narrow transformation, wide transformation.

Narrow Transformations

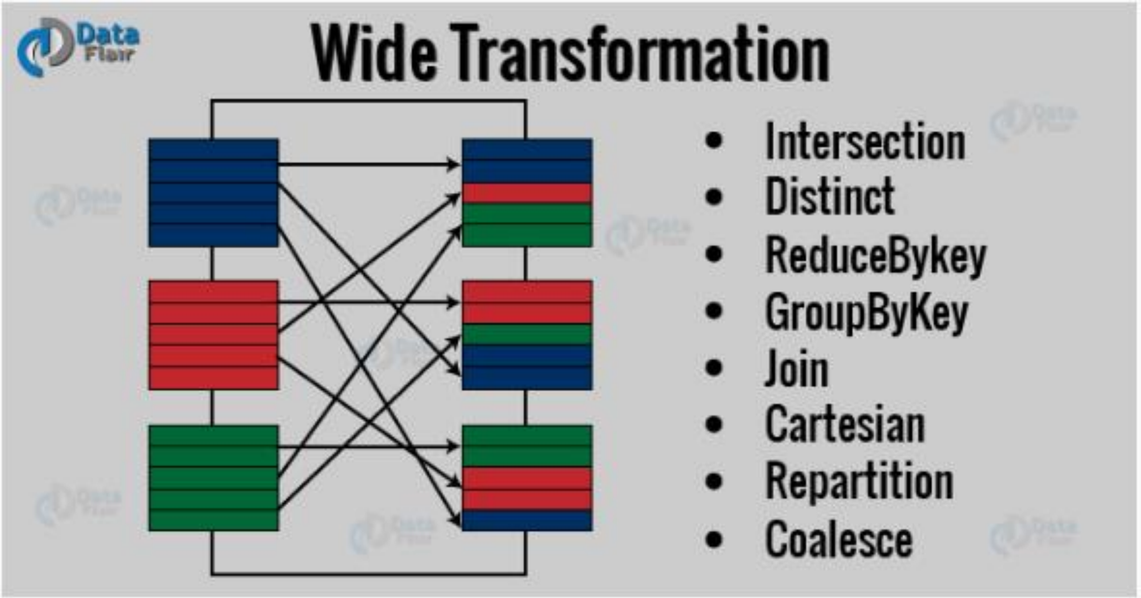
- It is the result of map, filter and such that the data is from a single partition only, i.e. it is self-sufficient.
- An output RDD has partitions with records that originate from a single partition in the parent RDD.
- Spark groups narrow transformations as a stage known as **pipelining**.



Transformations

Wide Transformations

- It is the result of groupByKey() and reduceByKey() like functions.
- The data required to compute the records in a single partition may live in many partitions of the parent RDD.
- Wide transformations are also known as *shuffle transformations* because they may or may not depend on a shuffle.



Spark Essentials: *Transformations*

transformation	*description*
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
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

Spark Essentials: *Transformations*

transformation	*description*
groupByKey([*numTasks*]**)**	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[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
sortByKey([*ascending*], [*numTasks*]**)**	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
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
cogroup(*otherDataset*, [*numTasks*]**)**	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset***)**	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

Spark Essentials: *Transformations*

Scala:

```
val distFile = sc.textFile("README.md")
distFile.map(l => l.split(" ")).collect()
distFile.flatMap(l => l.split(" ")).collect()
```

distFile is a collection of lines

Python:

```
distFile = sc.textFile("README.md")
distFile.map(lambda x: x.split(' ')).collect()
distFile.flatMap(lambda x: x.split(' ')).collect()
```

Spark Essentials: *Actions*

action	*description*
`reduce(func)`	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
`collect()`	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
`count()`	return the number of elements in the dataset
`first()`	return the first element of the dataset – similar to *take(1)*
`take(n)`	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
`takeSample(withReplacement, fraction, seed)`	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Spark Essentials: *Actions*

action	*description*
`saveAsTextFile(`*path*`)`	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*path*`)`	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*func*`)`	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Spark Essentials: *Actions*

Scala:

```scala
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split(" ")).map(word => (word, 1))
words.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```python
from operator import add
f = sc.textFile("README.md")
words = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1))
words.reduceByKey(add).collect()
```

Spark Essentials: *Persistence*

Spark can *persist* (or cache) a dataset in memory across operations

Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster

The cache is *fault-tolerant*: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

Spark Essentials: *Persistence*

transformation	*description*
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

Spark Essentials: *Persistence*

Scala:

```
val f = sc.textFile("README.md")
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).cache()
w.reduceByKey(add).collect()
```

Spark Essentials: *Broadcast Variables*

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

Spark Essentials: *Broadcast Variables*

Scala:

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Python:

```
broadcastVar = sc.broadcast(list(range(1, 4)))
broadcastVar.value
```

Spark Essentials: *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

Spark Essentials: *Accumulators*

Scala:

```
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
   global accum
   accum += x

rdd.foreach(f)

accum.value
```

driver-side

Driver and SparkContext

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val sc = new SparkContext(...)
```

- A SparkContext initializes the application driver, the latter then registers the application to the cluster manager, and gets a list of executors
- Since then, the driver takes full responsibilities

WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

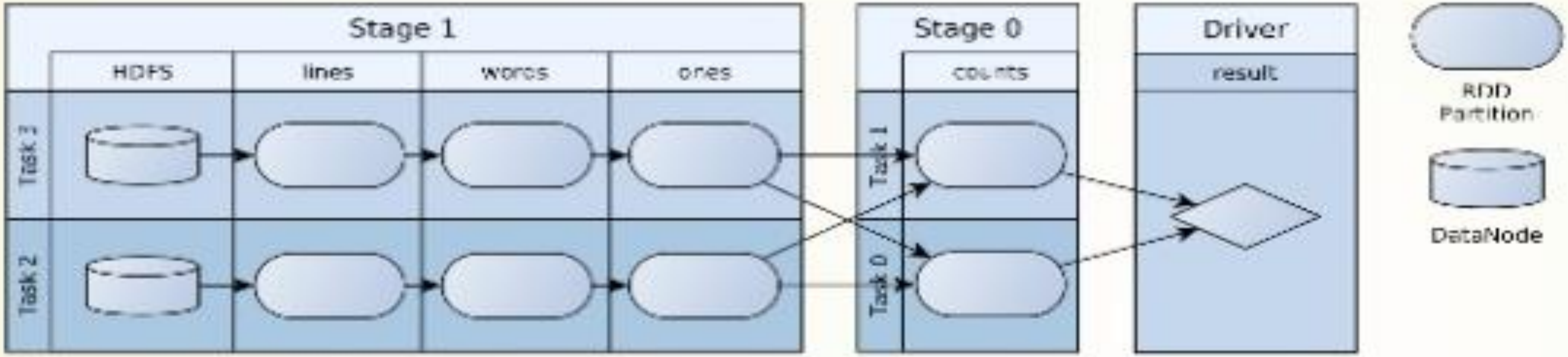
- RDD lineage DAG is built on *driver* side with:
  - Data source RDD(s)
  - Transformation RDD(s), which are created by transformations

WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

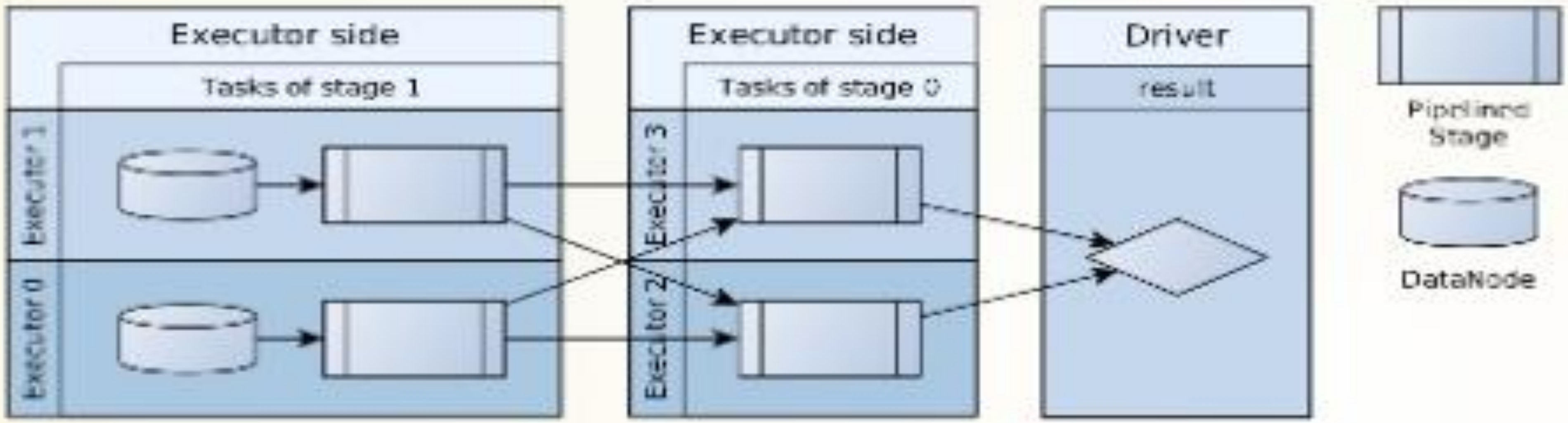
- Once an action is triggered on *driver* side, a job is submitted to the *DAG scheduler* of the driver

WordCount Revisited



- DAG scheduler cuts the DAG into stages and turns each *partition* of a stage into a single task.
- DAG scheduler decides *what to run*

WordCount Revisited



- Tasks are then scheduled to executors by driver side *task scheduler* according to *resource* and *locality* constraints
- Task scheduler decides *where to run*

WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- Within a task, the lineage DAG of corresponding stage is serialized together with closures of transformations, then sent to and executed on scheduled *executors*

WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- The `reduceByKey` transformation introduces in a shuffle
- Shuffle outputs are written to local FS on the mapper side, then downloaded by reducers

WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- ReduceByKey automatically combines values within a single partition locally on the mapper side and then reduce them globally on the reducer side.

WordCount Revisited

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- At last, results of the action are sent back to the driver, then the job finishes.

How To Control Parallelism?

- Can be specified in a number of ways
  - RDD partition number
    - `sc.textFile("input", minSplits = 10)`
    - `sc.parallelize(1 to 10000, numSlices = 10)`
  - Mapper side parallelism
    - Usually inherited from parent RDD(s)
  - Reducer side parallelism
    - `rdd.reduceByKey(_ + _, numPartitions = 10)`
    - `rdd.reduceByKey(partitioner = p, _ + _)`
    - …

How To Control Parallelism?

- "Zoom in/out"
  - `RDD.repartition(numPartitions: Int)`
  - RDD.coalesce(

```
RDD.coalesce(
   numPartitions: Int,
   shuffle: Boolean)
```

A Trap of Partitions

```
sc.textFile("input", minSplits = 2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```

- In this case, the final split size equals to local FS block size, which is 32MB by default, and 60GB / 32MB ≈ 2K
- `ReduceByKey` generates $2K^2$ shuffle outputs

A Trap of Partitions

```
sc.textFile("input", minSplits = 2).coalesce(2)
  .map { line =>
    val Array(key, value) = line.split(",")
    key.toInt -> value.toInt
  }
  .reduceByKey(_ + _)
  .saveAsText("output")
```

- Use `RDD.coalesce()` to control partition number precisely.

When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime.

Submitting Spark Application

- The *spark-submit* script in Spark's bin directory is used to launch applications on a cluster
- Bundling Application's Dependencies
  - If application code depends on other projects, then package them alongside your application in order to distribute the code to a Spark cluster.
  - Create an assembly jar (or "uber" jar) containing your code and its dependencies.
  - Both sbt and Maven have assembly plugins
  - When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime.

When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime.

Submitting Spark Application Cont..

- Once an user application is bundled, it can be launched using the bin/spark-submit script
- This script takes care of setting up the classpath with Spark and its dependencies, and can support different cluster managers and deploy modes that Spark supports

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime.

Submitting Spark Application Cont..

- Some of the commonly used options are

----class: The entry point for your application

--master: The 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. For values that contain spaces wrap "key=value" in quotes

application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.

application-arguments: Arguments passed to the main method of your main class, if any

- *Client Mode: In client mode, the driver is launched directly within the spark-submit process which acts as a client to the cluster. The input and output of the application is attached to the console.* Thus, this mode is especially suitable for applications that involve the REPL

When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime.

Submitting Spark Application Cont..

- *Cluster Mode: In cluster mode, the driver is launched within the spark cluster on one of the worker node which has sufficient resources to run driver process. Cluster mode minimizes network latency between the drivers and the executors*

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100
# Run on a Spark standalone cluster in client deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime.

Submitting Spark Application Cont..

```
# Run on a Spark standalone cluster in cluster deploy mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime.

Submitting Spark Application Cont..

```
# Run on a YARN cluster
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```

When creating assembly jars, list Spark and Hadoop as provided dependencies; these need not be bundled since they are provided by the cluster manager at runtime.

Submitting Spark Application Cont..

The master URL passed to Spark can be in one of the following formats:

Master URL	Meaning
`local`	Run Spark locally with one worker thread (i.e. no parallelism at all).
`local[K]`	Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
`local[K,F]`	Run Spark locally with K worker threads and F maxFailures (see spark.task.maxFailures for an explanation of this variable)
`local[*]`	Run Spark locally with as many worker threads as logical cores on your machine.
`local[*,F]`	Run Spark locally with as many worker threads as logical cores on your machine and F maxFailures.
`spark://HOST:PORT`	Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.
`spark://HOST1:PORT1,HOST2:PORT2`	Connect to the given Spark standalone cluster with standby masters with Zookeeper. The list must have all the master hosts in the high availability cluster set up with Zookeeper. The port must be whichever each master is configured to use, which is 7077 by default.
`mesos://HOST:PORT`	Connect to the given Mesos cluster. The port must be whichever one your is configured to use, which is 5050 by default. Or, for a Mesos cluster using ZooKeeper, use `mesos://zk://....` To submit with `--deploy-mode cluster`, the HOST:PORT should be configured to connect to the MesosClusterDispatcher.
`yarn`	Connect to a YARN cluster in `client` or `cluster` mode depending on the value of `--deploy-mode`. The cluster location will be found based on the `HADOOP_CONF_DIR` or `YARN_CONF_DIR` variable.
`k8s://HOST:PORT`	Connect to a Kubernetes cluster in `cluster` mode. Client mode is currently unsupported and will be supported in future releases. The `HOST` and `PORT` refer to the [Kubernetes API Server](https://kubernetes.io/docs/reference/generated/kube-apiserver/). It connects using TLS by default. In order to force it to use an unsecured connection, you can use `k8s://http://HOST:PORT`.