

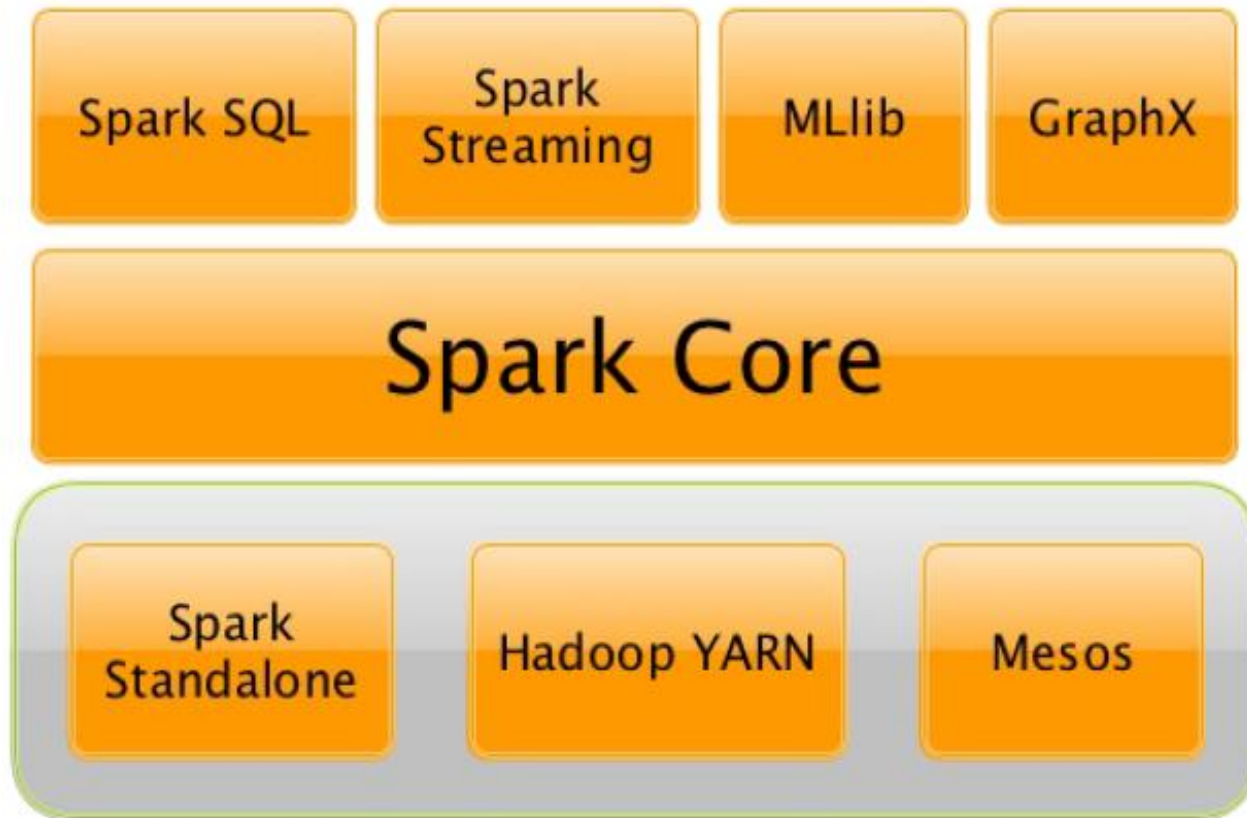


ADVANCED SPARK CORE

Bin Jiang

04/22/2017

Overview of Spark



History of Spark

Apache Spark implements a type of data parallelism that seeks to improve upon the MapReduce paradigm popularized by Apache Hadoop. It extended MapReduce in four key areas:

- **Improved programming model**

Spark provides a higher level of abstraction through its APIs than Hadoop

- **Introduces workflow**

Spark allows analytics to be decomposed into tasks and expressed as Directed Acyclic Graphs (DAGs)

History of Spark

- **Better Memory Utilization**

Spark exploits the memory on each node for in-memory caching of datasets. It permits access to caches between operations to improve performance over basic MapReduce

- **Integrated Approach**

With support for streaming, SQL execution, graph processing, machine learning, database integration, and much more, it offers one tool to rule them all

Spark Architecture

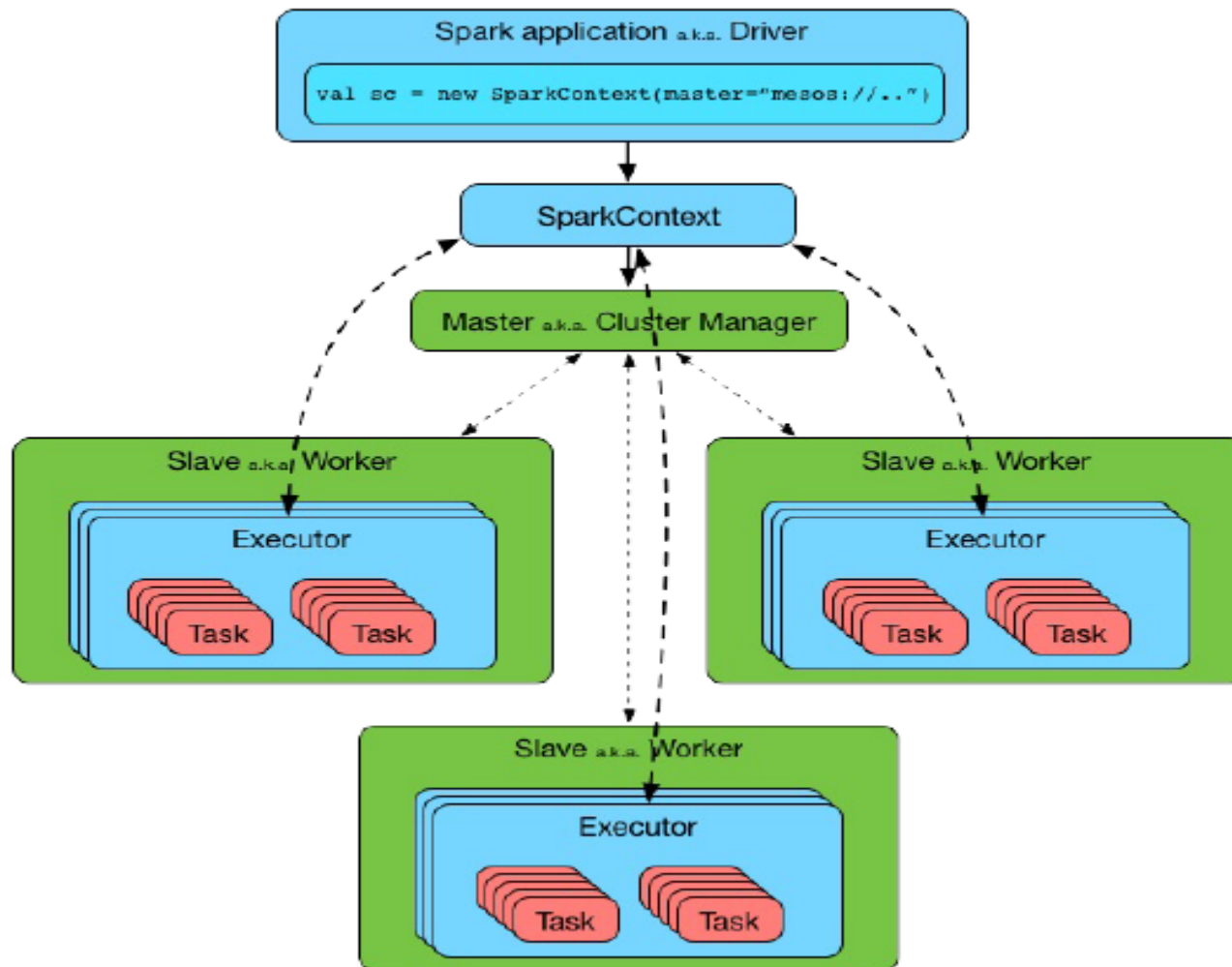
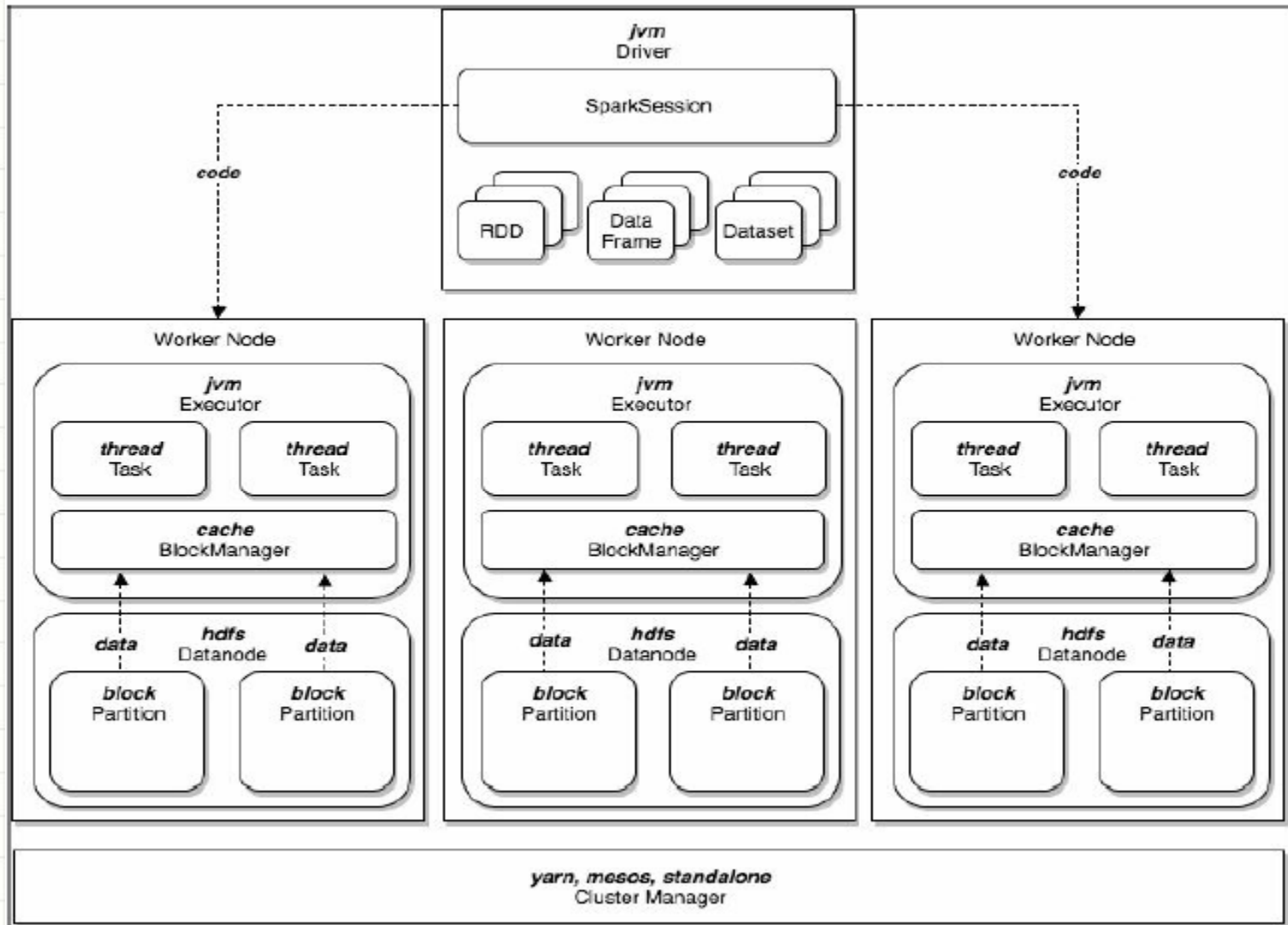


Figure 2. Spark architecture in detail

Spark Architecture



The Spark Shell

- Spark interactively through a modified version of the Scala shell
- The Spark Shell is often referred to as REPL (Read/Eval/Print Loop)
- Spark shell provides an easy and convenient way to prototype certain operations quickly
- No need to develop a full program, packaging it and then deploying it
- The Spark Shell session acts as the Driver process
- Once loaded the shell, both the SparkContext (sc) and the SQLContext (sqlContext) are already loaded and ready to go
- The Spark Shell supports only Scala and Python
- A great way to learn the framework

Launching the Spark Shell

- There are three ways to launch the spark shell
 - `spark-shell [options]` – scala
 - `pyspark [options]` – python
 - `sparkR [options]` - R
- Note: using the following command to launch spark application:
 - `spark-submit [options]`

Launching the Spark Shell with Options

- Usage: `spark-shell [options]` `spark-shell [options]`

Options:

- `--master MASTER_URL` - `spark://host:port`, `mesos://host:port`, `yarn`, or `local`.
 - `--deploy-mode DEPLOY_MODE` - Whether to launch the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster") (Default: client).
 - `--class CLASS_NAME` - Your application's main class (for Java / Scala apps).
 - `--name NAME` - A name of your application.
 - `--jars JARS` - Comma-separated list of local jars
- You can also launch the spark shell like if you want to download the jars online and add dependencies :
 - `spark-shell --packages com.couchbase.client:spark-connector_2.10:1.0.0`

Using the Spark Shell Command

- Examples of commands:
 - `sc`
 - `sqlContext`
 - `import com.couchbase.spark._`
 - `sc.parallelize`
 - `sc.stop()`
 - `sc.textFile`

Spark Core Concepts

- Driver
- SparkConf
- Spark Session
- Spark Context
- Executor
- RDD
- RDD Operation (Shuffle, Transformation and Action)
- RDD Cache
- RDD Persistence
- Shared Variable
- DAG

Spark Core Concepts

- DAG Scheduler
- Stage
- Task
- Task Scheduler
- Checkpointing
- Serialization
- SchedulerBackend
- ExecutorBackend
- BlockManager

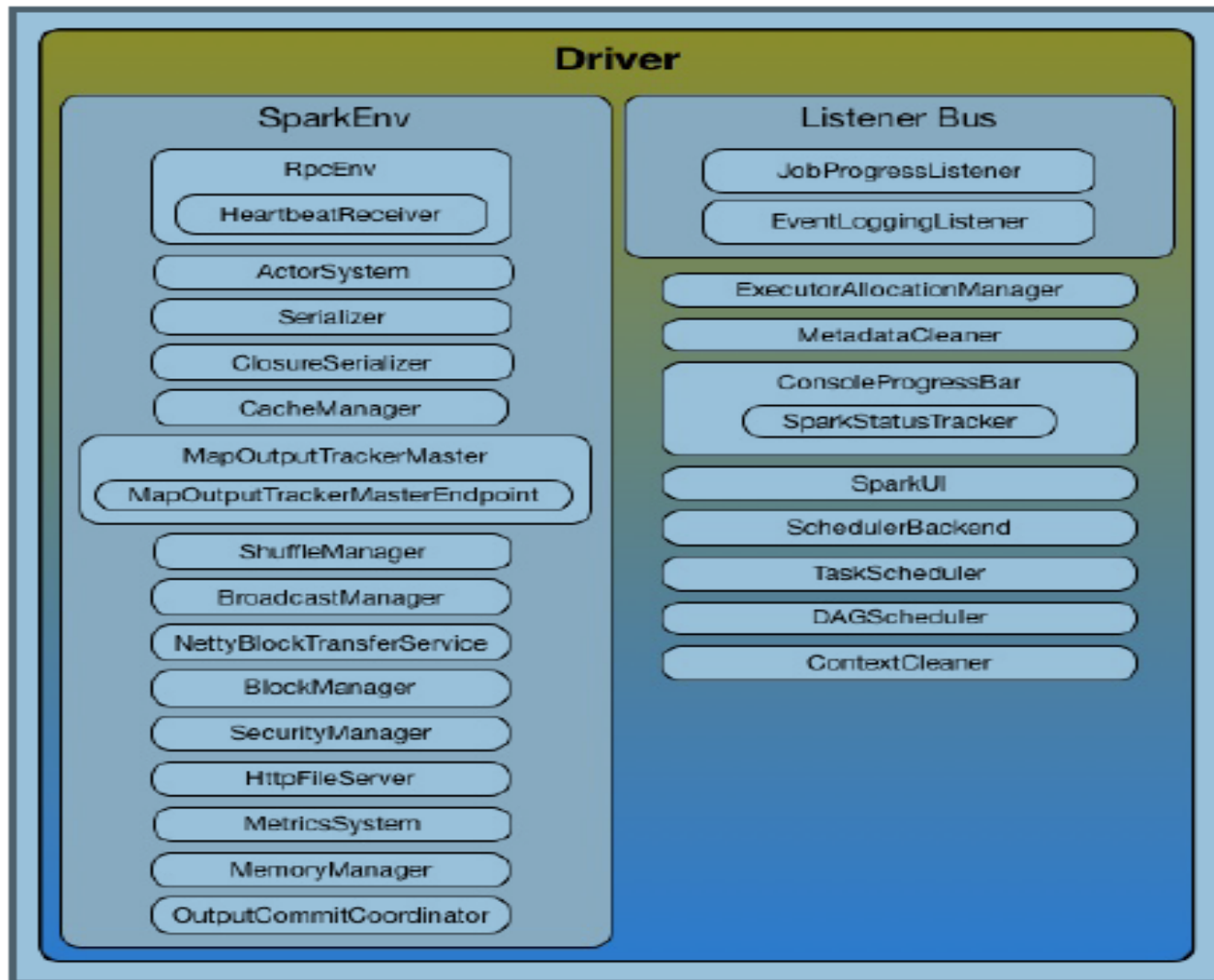
Spark Core Concepts

- ShuffleManager
- MapOutputTracker
- BroadcastManager
- Dynamic Allocation

Driver

- The Driver is the main entry point for Spark. It's the program that you start, it runs in a single JVM, and it initiates and controls all of the operations in your job
- In terms of performance, it's likely that you'll want to avoid bringing large datasets back to the driver, as running such operations (such as `rdd.collect`) can often cause an `OutOfMemoryError`

Driver



SparkConf

- Every user program starts with creating an instance of SparkConf that holds the master URL to connect to (`spark.master`), the name for your Spark application (that is later displayed in web UI and becomes `spark.app.name`) and other Spark properties required for proper runs. The instance of SparkConf can be used to create SparkContext
- Three ways to configure Spark
 - `conf/spark-defaults.conf`
 - `--conf`
 - SparkConf

Spark Session

- As the driver is starting, the `SparkSession` class is initialized. The `SparkSession` class provides access to all of Spark's services, via the relevant context, such as `SQLContext`, `SparkContext`, and `StreamingContext` classes.

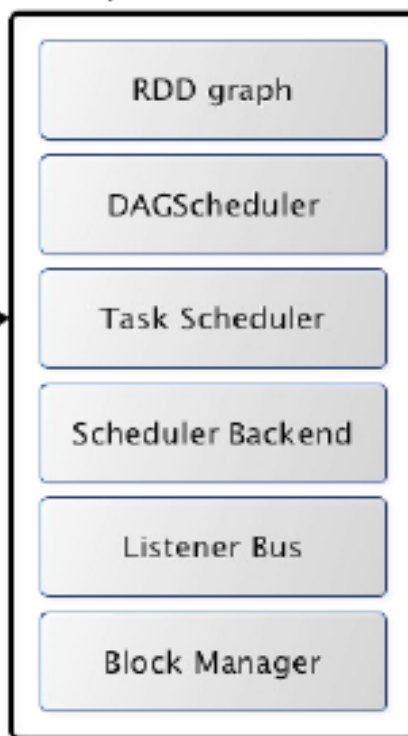
The Spark Context and SQL Context

- The Spark Context (sc) is the main Spark API object
- In the Spark shell, a special interpreter-aware SparkContext is already created for you, which is called (sc)
- Making your own SparkContext will not work
- SQLContext are already loaded and ready to go, which is called (sqlContext)

Spark Context

```
val sc = new SparkContext(master='local[*]',  
    appName='SparkMe App', new SparkConf)  
val lines = sc.textFile(...).cache()  
val c = lines.count()  
println(s'There are $c lines in $fileName')
```

Spark context



Spark Context

SparkContext offers the following functions:

- **Getting current status of a Spark application**
 - SparkEnv
 - SparkConf
 - deployment environment (as master URL)
 - application name
 - unique identifier of execution attempt
 - deploy mode
 - default level of parallelism
 - Spark user
 - the time (in milliseconds) when SparkContext was created
 - Spark version
 - Storage status

Spark Context

SparkContext offers the following functions:

- **Setting Configuration**

- master URL
- Local Properties — Creating Logical Job Groups
- Setting Local Properties to Group Spark Jobs
- Default Logging Level

Spark Context

SparkContext offers the following functions:

- **Creating Distributed Entities**
 - RDDs
 - Accumulators
 - Broadcast variables

Spark Context

SparkContext offers the following functions:

- **Accessing services**

- TaskScheduler
- LiveListenerBus
- BlockManager
- SchedulerBackends
- ShuffleManager
- ContextCleaner

Spark Context

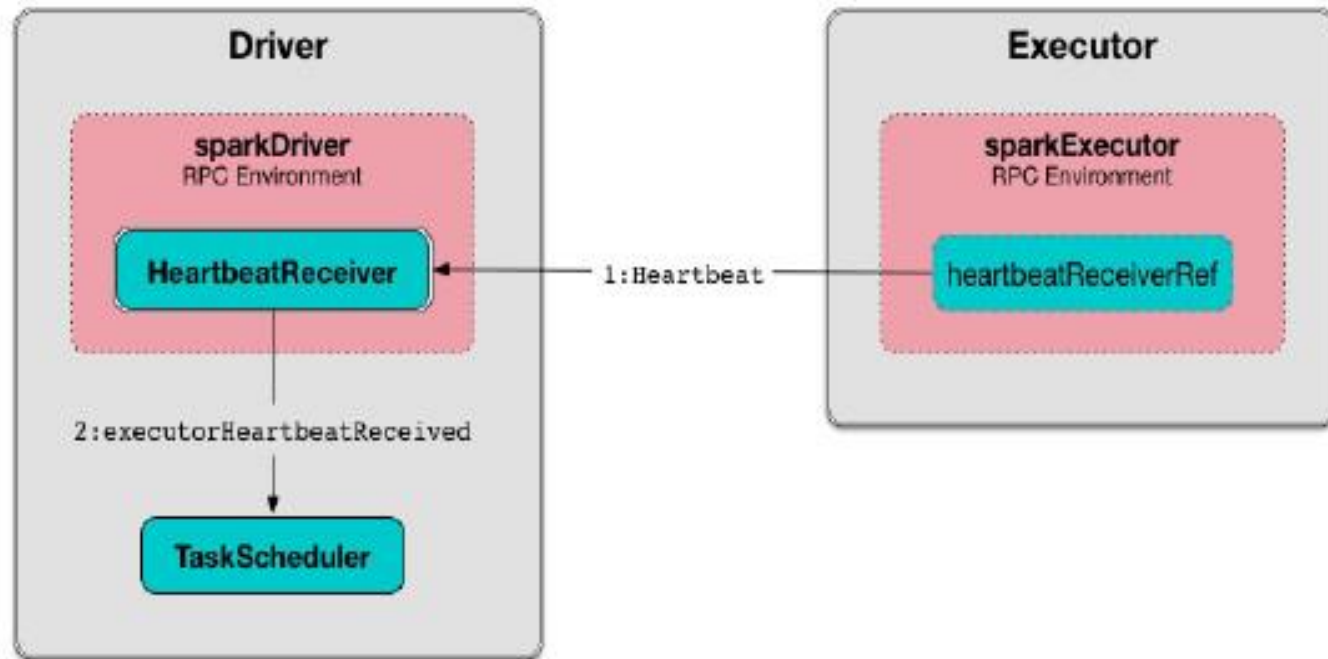
SparkContext offers the following functions:

- **Running jobs synchronously**
- **Submitting jobs asynchronously**
- **Cancelling a job**
- **Cancelling a stage**
- **Assigning custom Scheduler Backend, TaskScheduler and DAGScheduler**
- **Closure cleaning**
- **Accessing persistent RDDs**
- **Unpersisting RDDs, i.e. marking RDDs as non-persistent**
- **Registering SparkListener**
- **Programmable Dynamic Allocation**

Executor

- Executors are processes that run on the worker nodes of your cluster. When launched, each executor connects back to the driver and waits for instructions to run operations over data.
- You decide on how many executors your analytic needs and this becomes your maximum level of parallelism.

Executor



RDD

- A Resilient Distributed Dataset (RDD is the fundamental abstraction in Spark representing data partitioned across machines in the cluster
- A fault-tolerant collection of elements that can be operated on in parallel
- RDDs can be re-created and also distributed cross the cluster, which is called partitioning
- Two ways to create RDDs:
 - Parallelizing an existing collection in your driver program
 - Referencing a dataset in an external storage system
 - ✓ Shared filesystem
 - ✓ HDFS
 - ✓ Hbase
 - ✓ Any data source offering a Hadoop InputFormat
 - ✓ S3

RDD - Traits

- In-Memory
- Immutable
- Lazy evaluated
- Cachable
- Parallel
- Typed
- Partitioned
- Location-Stickiness

Types of RDD

- ParallelCollectionRDD
- CoGroupedRDD
- HadoopRDD
- MapPartitionsRDD
- CoalescedRDD
- ShuffledRDD
- PipedRDD
- PairRDD
- DoubleRDD
- SequenceFileRDD

Parallelized Collections

- Created by calling SparkContext's parallelize method on an existing collection
- The elements of the collection are copied to form a distributed dataset that can be operated on in parallel
- Number of partitions automatically based on your cluster
- You can set the number of partitions manually by passing it as a second parameter to parallelize
- Examples:

```
val data = Array(1, 2, 3, 4, 5)
```

```
val distData = sc.parallelize(data, 3)
```

```
distData.reduce((a, b) => a + b)
```

More on Parallelize Collections

- When use `collect()` operation on RDD, which will collect all processed partitioned RDD's data elements back in a collection in the Driver's memory

External Datasets

- Create distributed datasets from any storage source supported by Hadoop
 - Local file system, HDFS, Cassandra, HBase, Amazon s3,
 - Text files, SequenceFiles and any other Hadoop InputFormat.

Reading Files with Spark

- Text file RDDs can be created using SparkContext's `textFile` method
 - ✓ This method takes an URI for the file (either a local path on the machine `file://`, or a `hdfs://` for HDFS, `s3://` for Amazon S3, etc URI)
 - ✓ Reads it as a collection of lines
 - ✓ Once created, RDD can be acted on by dataset operations
- If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes
- Support running on directories, compressed files, and wildcards as well. For example, you can use `textFile("/my/directory")`, `textFile("/my/directory/*.txt")`, and `textFile("/my/directory/*.gz")`

Reading Files with Spark

- The `textFile` method also takes an optional second argument for controlling the number of partitions of the file
- By default, Spark creates one partition for each block of the file (blocks being 64MB by default in HDFS)
- Can also ask for a higher number of partitions by passing a larger value
- Cannot have fewer partitions than blocks

Loading Small-sized Files

- `SparkContext.wholeTextFiles` lets you read a directory containing multiple small text files
- Create each of them as (filename, content) pairs, which means create a key-value RDD, the key is the filename and value is the content of the file
- This is in contrast with `textFile`, which would return one record per line in each file

Loading Sequence Files

- Use SparkContext's `sequenceFile[K, V]` method
- K and V are the types of key and values in the file
- K and V should be subclasses of Hadoop's Writable interface, like `IntWritable` and `Text`
- You can also specify native types for a few common Writables
- For example, `sequenceFile[Int, String]` will automatically read `IntWritables` and `Texts`

Loading other Hadoop Files

- Use the `SparkContext.hadoopRDD` method
- Take an arbitrary `JobConf` and input format class, key class and value class
- Set these the same way you would for a Hadoop job with your input source
- You can also use `SparkContext.newAPIHadoopRDD` for `InputFormats` based on the “new” MapReduce API

Saving Files

- The `saveAsTextFile(path)` method of an RDD reference allows you to write the elements of the dataset as a text files
- You can save the file on HDFS or any other Hadoop-supported file system
- When saving the file, Spark calls the `toString` method of each data element to convert it to a line of text in the output file
- Methods for saving files in other output file formats:
 - `saveAsSequenceFile`
 - `saveAsObjectFile`
- `RDD.saveAsObjectFile` and `SparkContext.objectFile` support saving an RDD in a simple format consisting of serialized Java objects

RDD Operations

- There are two types of RDD operations:
 - Actions
 - Transformations

Spark Actions

- **Actions** are RDD operations that produce non-RDD values
- Materialize a value in a Spark program
- Actions are synchronous
- Action evaluates the RDD lineage graph
- Actions are one of two ways to send data from executors to the driver(the other being accumulators).
- Some of the Spark actions:
 - **count()** – return the number of data elements in an RDD
 - **take(X)**- return first x data elements in an RDD
 - **collect()** – combine all the data elements in an RDD in an array
 - **reduce()** – return the result of an aggregation operation on data elements in an RDD
 - **aggregate()**
 - **foreach()**
 - **foreachPartition()**
 - **saveAs*Actions()**

RDD Transformations

- Transformations are *functions* that take a RDD as the input and produce one or many RDDs as the output
- Do not change the input RDD, always produce one or more new RDDs
- Applying transformations you incrementally build a RDD lineage with all the parent RDDs of the final RDD(s)
- **Transformations** are lazy operations on a RDD that create one or many new RDDs
- Certain transformations can be **pipelined** which is an optimization that Spark uses to improve performance of computations

Narrow Transformations

- Narrow transformations are the result of map, filter and such that is from the data from a single partition only, i.e. it is self-sustained
- An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result
- Spark groups narrow transformations as a stage which is called **pipelining**

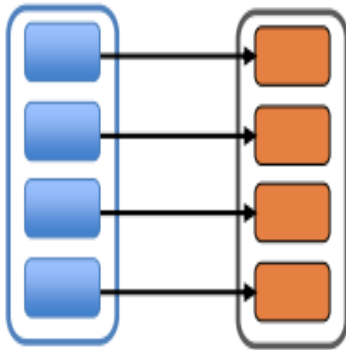
Wide Transformations

- Wide transformations are the result of `groupByKey` and `reduceByKey`
- The data required to compute the records in a single partition may reside in many partitions of the parent RDD
- All of the tuples with the same key must end up in the same partition, processed by the same task

RDD Transformations

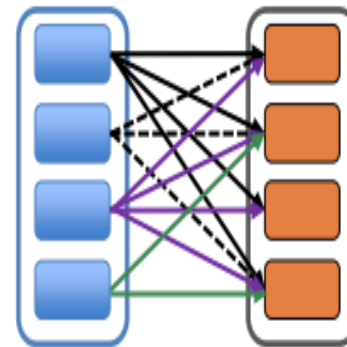
narrow

each partition of the parent RDD is used by at most one partition of the child RDD



wide

multiple child RDD partitions may depend on a single parent RDD partition



RDD Transformations

- Narrow Transformations:

- filter () – return a new RDD by applying the predicate function
- map() – return a new RDD by applying the input function
- flatMap - Similar to map, but each input item can be mapped to 0 or more output items
- union() – merges this RDD with another RDD
- mapValues
- flatMapValues
- Glom
- Pipe
- zipWithIndex
- cartesian
- mapPartitionsWithInputSplit
- mapPartitions, mapPartitionsWithIndex,
- mapPartitionsWithContext, sample, randomSplit.

RDD Transformations

- Wide Transformations:

- `groupByKey` - When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs
- `reduceByKey` - When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs
- `aggregateByKey` - When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs
- `sortByKey`
- `Join`
- `cartesian`
- `combineByKey`
- `partitionBy`
- `repartition,`
- `repartitionAndSortWithinPartitions`

RDD Transformations

- Wide Transformations:

- **coalesce**
- **subtractByKey**
- **cogroup**

Note: The coalesce, subtractByKey and cogroup transformations could be narrow depending on where data is physically situated.

Other Pair RDD Operations

- `countByKey` – return a map with the count of occurrences of each key in an RDD
- `groupByKey` – return the result of grouping operation for each key in an RDD
- `Join` – return a composite RDD with data elements from two RDDs joined by matching keys

Creating a Pair RDD with Map

- Let's say the input text file's rows consist of two tab-delimited columns:

key1 val1

key2 val2

- The following Scala on Spark code will produce a Pair RDD containing two-elements tuples (keyX, valX)
- Note the () in (f(0),f(1)) – that's the making of a tuple

```
val clientRDD = sc.textFile("/root/labs/datasets/labs/people.txt");
val pairRDD = clientRDD.map(_.split('\t')).map(f=>(f(0),f(1)))
```

Working with Key-Value Pairs

- A few special operations are only available on RDDs of key-value pairs
- These operations are automatically available on RDDs containing tuple2 objects
- The key-value pair operations are available in the PairRDDFunctions class, which automatically wraps around an RDD of tuples
- Example

```
val lines = sc.textFile("/root/labs/datasets/labs/people.txt")
```

```
val pairs = lines.map(s => (s, 1))
```

```
val counts = pairs.reduceByKey((a, b) => a + b)
```


Spark Core Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

Spark Core PairRDD Operations

TRANSFORMATIONS

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure

- partitionBy

Spark Core Operations

ACTIONS



- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

AggregateByKey

The **aggregateByKey** function requires 3 parameters:

- An initial 'zero' value that will not effect the total values to be collected. For example if we were adding numbers the initial value would be 0. Or in the case of collecting unique elements per key, the initial value would be an empty set.
- A combining function accepting two paremeters. The second paramter is merged into the first parameter. This function combines/merges values within a partition.
- A merging function function accepting two parameters. In this case the paremters are merged into one. This step merges values across partitions.

CombineByKey

The **combineByKey** function requires 3 parameters:

- Create combiner function: x
- Merge value function: y
- Merge combiners function: z

The API format is `combineByKey(x, y, z)`.

Comparison of ByKey

reduceByKey vs groupByKey vs aggregateByKey vs combineByKey vs foldByKey:

- **reduceByKey** will aggregate y key before shuffling, and groupByKey will shuffle all the value key
- **combineByKey** can be used when combining elements but the return type differs from the input value type
- **foldByKey** merges the values for each key using an associative function and a neutral "zero value", foldByKey gives the chance to initialize the reduce operation

Comparison of ByKey

reduceByKey vs groupByKey vs aggregateByKey vs combineByKey vs foldByKey:

- **combineByKey** is more general than aggregateByKey. Actually, the implementation of aggregateByKey, reduceByKey and groupByKey is achieved by combineByKey
- **aggregateByKey** is similar to reduceByKey but you can provide initial values when performing aggregation.
- **aggregateByKey** is suitable for compute aggregations for keys
- **combineByKey** is more general and you have the flexibility to specify whether you'd like to perform map side combine

Glom

- Glom on spark rdd which treats a partition as an array rather as single row at time. This will speed up some operations with some increased memory usage
- Glom is highly useful when to represent rdd operations as matrix manipulations
- In many machine learning algorithms there will be needed to find weighted value of rows , i.e multiplying each row by a given weight vector. Doing this row by row, using map operation will be very costly as will be not able to use matrix libraries optimization
- Glom can help to multiply with whole partition at a time so that the computation will speed up significantly

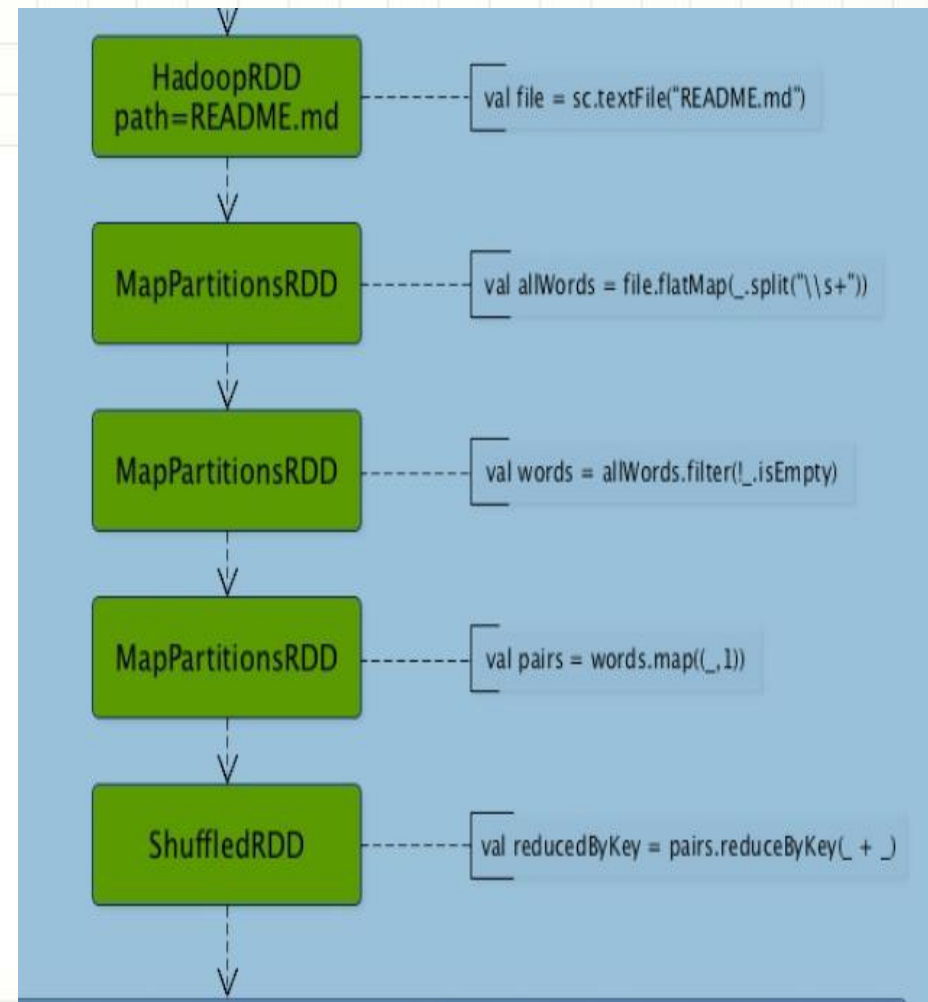
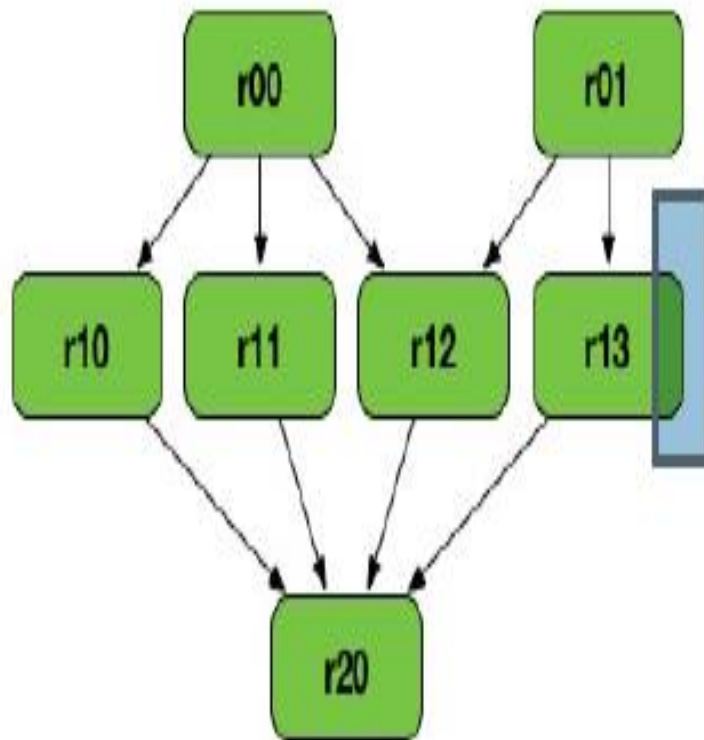
RDD Lineage

- **RDD Lineage Graph** (aka *RDD operator graph*) is a graph of all the parent RDDs of a RDD
- It is built as a result of applying transformations to the RDD
- Logical Execution Plan will be created
- A RDD lineage graph is hence a graph of what transformations need to be executed after an action has been called
- learn about a RDD lineage graph using `RDD.toDebugString` method

RDD – Logical Execution Plan

Logical Execution Plan starts with the earliest RDDs (those with no dependencies on other RDDs or reference cached data) and ends with the RDD that produces the result of the action that has been called to execute.

RDD Lineage



RDD Caching

- *Cache* dataset in memory across operations
- each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset
- Allows future actions to be much faster
- Caching is a key tool for iterative algorithms and fast interactive use
- Mark an RDD to be cached using the `cache()` methods on it
- Spark's cache is fault-tolerant

RDD Persistence

- Each persisted RDD can be stored using a different *storage level*
 - *Persist the dataset on disk*
 - *persist it in memory but as serialized Java objects*
 - *replicate it across nodes*
 - *store it off-heap in Tachyon*
- *levels can be set by passing a StorageLevel object (Scala, Java, Python) to persist()*
- *The cache() method is a shorthand for using the default storage level*

RDD Persistence

- These Storage options are set as flags passed to the `persist()` method:
 - `MEMORY_ONLY` – this is the default value, equivalent to `cache()`
 - `MEMORY_AND_DISK` – a hybrid approach, spilling data to disk only when the `MEMORY_ONLY` options has been exhausted
 - `DISK_ONLY` – all RDD partitions are committed to disk
 - `MEMORY_ONLY_SER` - Store RDD as *serialized* Java objects
 - `MEMORY_AND_DISK_SER` - Similar to `MEMORY_ONLY_SER`
 - `MEMORY_ONLY_2`, `MEMORY_AND_DISK_2` - replicate each partition on two cluster nodes
 - `OFF_HEAP` – store an RDD in serialized format in Tachyon; this option is attractive in environments with large heaps or multiple concurrent applications

RDD Persistence

- Spark also automatically persists some intermediate data in shuffle operations
- Avoid recomputing the entire input if a node fails during the shuffle
- Spark's storage levels are meant to provide different trade-offs between memory usage and CPU efficiency
- Spark automatically monitors cache usage on each node and drops out old data partitions
- To manually stop RDD persistence and purge all its partitions from memory and disk, use `unpersist()`

Storage Level Recommendation

- Choose MEMORY_ONLY if RDDs fit the memory
- Choose MEMORY_ONLY_SER and Kryo serialization library if RDDs doesn't fit the memory
- Don't choose MEMORY_AND_DISK if the computation is not expensive
- Choose MEMORY_ONLY_2 or MEMORY_AND_DISK_2 if want fast fault recovery

The Tachyon Storage

- In Spark, tachyon is the default off-heap option for storing RDDs outside of the JVM's heap avoiding GC overheads and making RDDs resilient
- Open Source Memory Speed Virtual Distributed Storage
- Tachyon is a clustered in-memory fault-tolerant file system developed at UC Berkeley AMPLab
- It supports the standard file system operations, such as format, mv, rm, mkdir, cat, touch etc
- It offers applications (e.g. multiple Executor processes) an efficient file sharing mechanism by creating a RAM disk
- Tachyon is renamed as Alluxio now

Shared Variables

- Separate copies of all the variables used when pass the function to Spark operation
- No updates to the variables on the remote machine are propagated back to the driver program
- Supporting general, read-write shared variables across tasks would be inefficient
- Spark does provide two limited types of shared variables for two common usage patterns: broadcast variables and accumulators

Broadcast Variables

- keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
- Give every node a copy of a large input dataset in an efficient manner
- Distribute broadcast variables using efficient broadcast algorithms to reduce communication cost
- Spark automatically broadcasts the common data needed by tasks within each stage
- Broadcast variables are created from a variable by calling `SparkContext.broadcast(v)`
- The broadcast variable is a wrapper around `v`, and its value can be accessed by calling the `value` method

Broadcast Variables

- Example:

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

```
broadcastVar.value
```

- After the broadcast variable is created, it should be used instead of the value *v*
- The object *v* should not be modified after it is broadcast

Accumulators

- Variables that are only “added” to through an associative operation
- Can therefore be efficiently supported in parallel
- Used to implement counters (as in MapReduce) or sums
- Spark natively supports accumulators of numeric types
- Programmers can add support for new types. If accumulators are created with a name
- Displayed in Spark’s UI
- An accumulator is created from an initial value *v* by calling `SparkContext.accumulator(v)`
- Tasks running on the cluster can then add to it using the `add` method or the `+=` operator
- Only the driver program can read the accumulator’s value, using its `value` method

Accumulators

- Example:

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

- For accumulator updates performed inside actions only
- Spark guarantees that each task's update to the accumulator will only be applied once
- Accumulators do not change the lazy evaluation model of Spark
- Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like `map()`

Developing Custom RDDs

- Users can implement custom RDDs by extending RDD
- Overwrite compute method, which computes value for each partition of RDD
- Overwrite getPartitions method, which specify the new partitions for the RDD

Submitting Application

- The spark-submit script in Spark's bin directory is used to launch applications on a cluster
- It can use all of Spark's supported cluster managers through a uniform interface
- Bundling Your Application's Dependencies
- Launching Applications with spark-submit
- Example

```
spark-submit --class <main-class> --master <master-url> --deploy-mode <deploy-mode> --executor-memory --total-executor-cores --supervise --num-executors --conf <key>=<value> <application-jar> [application-arguments]
```

- The system currently supports four cluster managers: Standalone, Apache Mesos, YARN and Amazon EC2

Running Spark on a Cluster

- Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications.
- Run on a Mesos cluster in cluster deploy mode with supervise

```
spark-submit --class org.apache.spark.examples.SparkPi --master  
mesos://localhost:7077 --deploy-mode cluster --supervise --executor-  
memory 1G --total-executor-cores 1 http://path/to/examples.jar 1000
```

- Hadoop YARN – the resource manager in Hadoop 2
- Run on a YARN cluster

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn --  
deploy-mode cluster --executor-memory 1G --num-executors 1  
/path/to/examples.jar 1000
```


Spark Stand-alone

- A simple cluster manager included with Spark that makes it easy to set up a cluster
- Run on a Spark standalone cluster in client deploy mode

```
spark-submit --class org.apache.spark.examples.SparkPi --master  
spark://localhost:7077 --executor-memory 1G --total-executor-cores 1  
/path/to/examples.jar 1000
```

- Run on a Spark standalone cluster in cluster deploy mode with supervise

```
spark-submit --class org.apache.spark.examples.SparkPi --master local[1]  
/path/to/examples.jar 100
```

- Run application locally on 1 core

```
spark-submit --class org.apache.spark.examples.SparkPi --  
master local[1] /path/to/examples.jar 100
```

The High-Level Execution Flow in Stand-alone Spark Cluster

- The Spark Driver program contacts the Spark Master daemon
- The Master engages the Spark Worker daemons on Worker nodes
- Each engaged Spark Worker daemon launches an Executor process on the same node
- Tasks are sent to respective Executors for execution
- Computed results are sent back to the Driver program

Data Partitioning

- A **partition** (aka *split*) is a logical chunk of a large distributed data set
- Every partition is processed in one Executor instance
- Spark manages data using partitions that helps parallelize distributed data processing with minimal network traffic for sending data between executors
- Spark tries to read data into an RDD from the nodes that are close to it
- RDDs get partitioned automatically without programmer intervention
- There are times when you'd like to adjust the size and number of partitions or the partitioning scheme according to the needs of your application

Single Local File System RDD Partitioning

- By default, a single local file is partitioned into two splits
- When loading, you can configure the number of partitions of your RDD, e.g.

`sc.textFile("file:/user/root/filename", 5)` will split the source file into 5 partitions

- The source file size, the number of machines in the cluster and some other metrics are the factors in determining the optimal number of partitions

Multiple File RDD Partitioning

- When you load multiple files
`sc.textFile("file:/user/root/*.log")`
- Spark makes each file (at least)one partition

Hadoop File RDD Partitioning

- When you load files from HDFS (`sc.textFile("hdfs://")`). Spark allocates partitions per HDFS block size
- By default, a partition is created for each HDFS partition, which by default is 64MB
- Spark tries to read data into an RDD from the nodes that are close to it
- When the processing of HDFS-based data files on a cluster is done, the computing results are returned to the Driver program by way of executing the `collect()` method on the resulting RDD that returns an array of the RDD elements

Parallel Data Processing of Partitions

- RDD operations on each partition are done in isolation from and in parallel with operations in other partitions
- Some RDD operations preserve the partitioning boundaries:
 - Filter, map, flatMap
- Other RDD operations require repartitioning – creating a new partition, possibly in another Executor:
 - groupBy, groupByKey, join, reduce, sort
 - repartition and coalesce transformation
 - Repartitioning is resource-expensive operation

Using Spark's Data Partitioners

Using Spark's data partitioners

- Understanding HashPartitioner
 - `rdd.foldByKey (afunction, 100)`
- Understanding RangePartitioner
 - `rdd.foldByKey(afunction, new HashPartitioner(100))`
- Understanding pair RDD custom partitioners

Using Spark's Data Partitioners

Shuffling when explicitly changing partitioners

- `rdd.aggregateByKey(zeroValue, 100)(seqFunc, comboFunc).collect()`
- `rdd.aggregateByKey(zeroValue, new CustomPartitioner())(seqFunc, comboFunc).collect()`

Using Spark's Data Partitioners

Repartitioning RDDs

- Repartitioning with `partitionBy`
- Repartitioning with `coalesce` and `repartition`
- Repartitioning with `repartitionAndSortWithinPartition`

Using Spark's Data Partitioners

Mapping data in partitions

- Understanding mapPartitions
- Understanding mapPartitionsWithIndex
- Understanding zipPartitions

Spark Application, Jobs and Tasks

- After an RDD action has been called it becomes a job that is then transformed into a set of stages that are submitted as TaskSets for execution
- Each Spark Job may consist of multiple stages
- A Spark Application(defined as an instance of SparkContext) contains multiple jobs
- An application may have multiple parallel jobs running concurrently, if Jobs are submitted from separate threads
- RDD operations are performed in parallel as tasks – each task is processed by an Executor process
- An Executor process can run multiple tasks and has a built-in cache

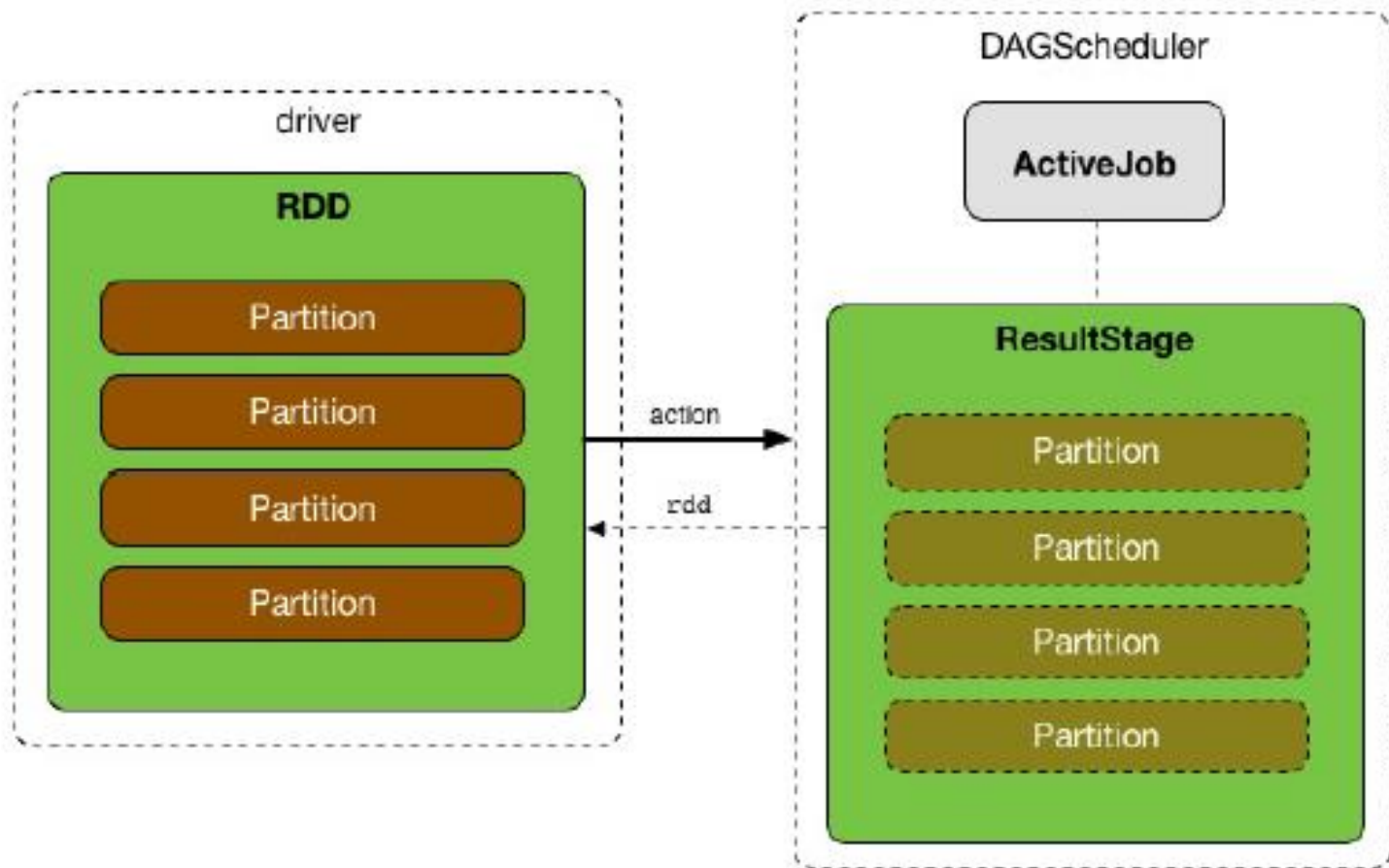
DAG

- A DAG represents the logical execution plan of all transformations involved in the execution of an action. Its optimization is fundamental to the performance of the analytic. In the case of SparkSQL and Datasets optimization is performed on your behalf by the catalyst optimizer

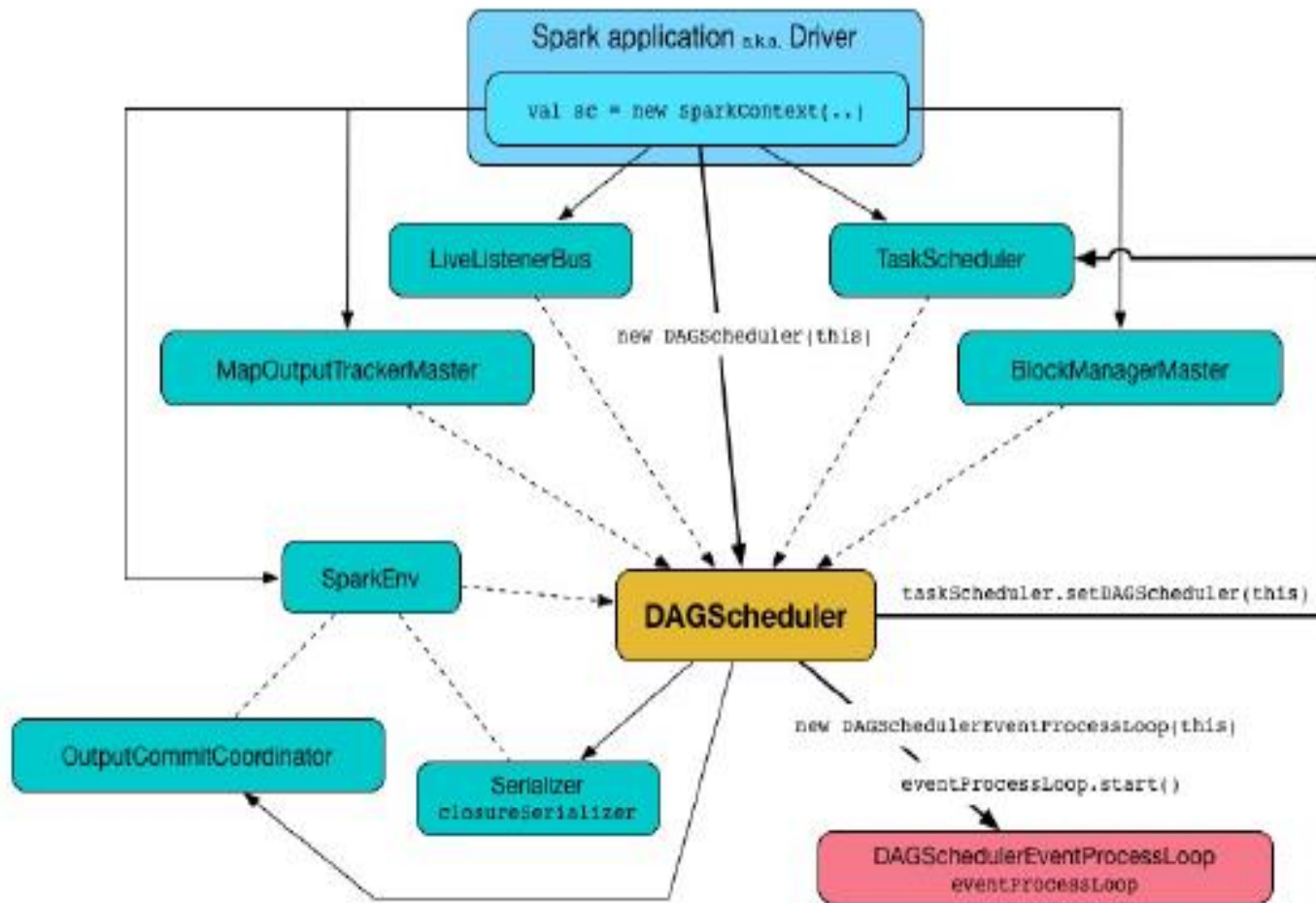
DAGScheduler

- DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling
- DAGScheduler works solely on the driver
- The fundamental concepts of DAGScheduler are jobs and stages
- DAGScheduler does three things in Spark:
 - Computes an execution DAG, i.e. DAG of stages, for a job
 - Determines the preferred locations to run each task on
 - Handles failures due to shuffle output files being lost

DAGScheduler



DAGScheduler



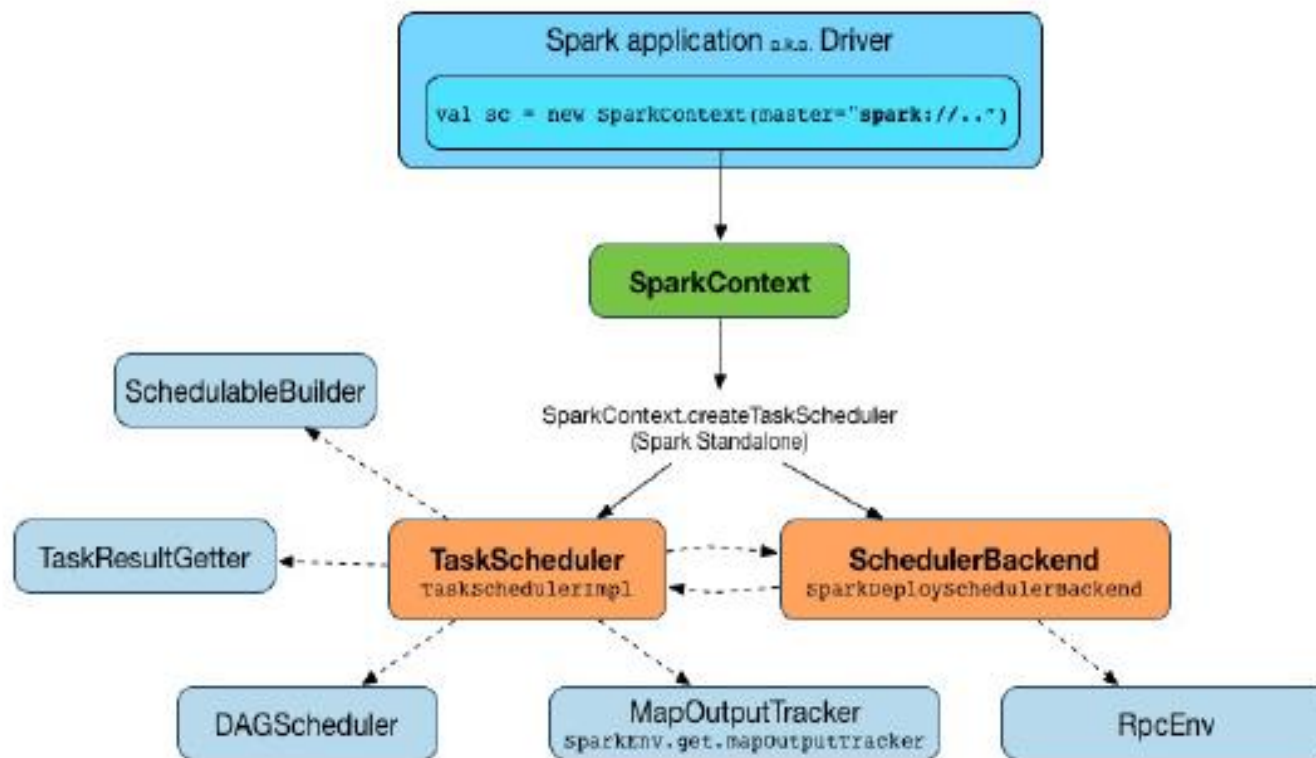
Task

- A Task represents an instruction to run a set of operations over a single partition of data. Each task is serialized over to an executor by the driver and, is in effect, what is referred to by the expression moving the processing to the data.

Task scheduler

- The task scheduler receives a set of tasks determined by the DAG scheduler (one task per partition) and schedules each to run on an appropriate executor in conjunction with data locality

Task scheduler



Stages

- A stage represents a group of operations that can be physically mapped to a task (one per partition).
- There are a couple of things to note about stages:
 - Any sequence of narrow transformations appearing consecutively in a DAG are pipelined together into a single stage. In other words, they execute in order, on the same executor and hence against the same partition and do not need a shuffle
 - Whenever a wide transformation is encountered in a DAG, a stage boundary is introduced. Two stages (or more in the case of join, and so on) now exist and the second cannot begin until the first has finished (see ShuffledRDD class for more details).

Stages and Shuffles

- Operations that run on the same data partition are grouped in a stage
- Repartitioning, also called shuffles, breaks processing flow in stages and perform an all-to-all operation
- Operations which can cause a shuffle
 - Repartition operations like repartition and coalesce,
 - ByKey operations (except for counting) like groupByKey and reduceByKey
 - join operations like cogroup and join

Stages and Shuffles

- If one desires predictably ordered data following shuffle then it's possible to use:
 - mapPartitions to sort each partition using, for example, .sorted
 - repartitionAndSortWithinPartitions to efficiently sort partitions while simultaneously repartitioning
 - sortBy to make a globally ordered RDD
- Certain shuffle operations can consume significant amounts of heap memory
- Shuffle also generates a large number of intermediate files on disk
- Shuffling steps are potentially expensive operations as they may involve any or all of three:
 - Data transfer over network
 - Data sorting
 - Disk I/O for storing intermediate results

Shuffles

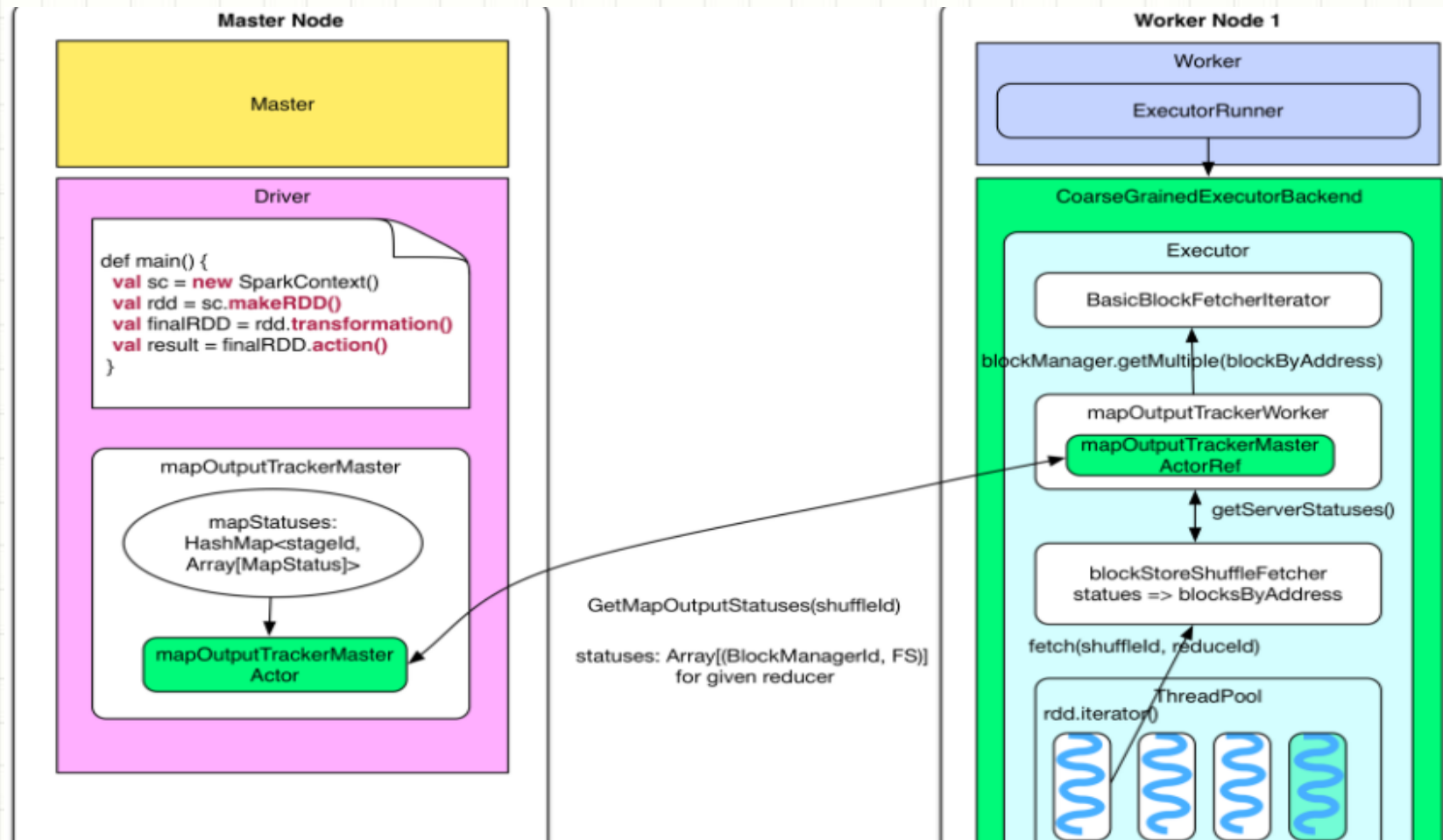
- In the shuffle operation, the task that emits the data in the source executor is “mapper”, the task that consumes the data into the target executor is “reducer”, and what happens between them is “shuffle”
- 2 important compression parameters:
 - **spark.shuffle.compress** – whether the engine would compress shuffle outputs or not
 - **spark.shuffle.spill.compress** – whether to compress intermediate shuffle spill files or not

Shuffles

- A number of shuffle implementations available in Spark, which is determined by the value of `spark.shuffle.manager`
 - **Hash Shuffle** (`spark.shuffle.manager = sort`)
 - **Sort Shuffle** (`spark.shuffle.manager = sort`)
 - **Unsafe Shuffle or Tungsten Sort** (`spark.shuffle.manager = tungsten-sort`)

How Shuffle works in Spark

Shuffle Read



How Shuffle works in Spark

Shuffle Read

- **How does reducer know where to fetch data**

Reducer needs to know on which node the FileSegments produced by ShuffleMapTask of parent stage are. The information is sent to driver's mapOutputTrackerMaster when ShuffleMapTask is finished. The information is also stored in mapStatuses: `HashMap[stageId, Array[MapStatus]]`

How Shuffle works in Spark

Shuffle Read

- **When does reducer to fetch data**

After all ShuffleMapTasks of parent stage end and then fetch. The fetched FileSegments have to be buffered in memory in the reducer which is set by `spark.reducer.maxMblnFlight` (48M by default)

How Shuffle works in Spark

Shuffle Read

- **Fetch and process the records at the same time or not?**
 - Fetch and process the records at the same time
 - In MapReduce, the shuffle stage fetches the data and then applies combine() logic at the same time but the reduce() logic
 - is applied after the shuffle-sort process
 - Spark does not require a sorted order for the reducer input data, so don't need to wait until all the data gets fetched to start processing

How Shuffle works in Spark

Shuffle Read

- **Where to store the fetched data**
 - The fetched data is saved in buffer. Then the data is processed, and written to a configurable location.
 - If `spark.shuffle.spill` is false, then the write location is only memory which is `AppendOnlyMap`
 - If `spark.shuffle.spill` is true, the processed data will be written to memory and disk, using `ExternalAppendOnlyMap`

How Shuffle works in Spark

Shuffle Read

- **Typical Transformations**

- `reduceByKey`
- `groupByKey`
- `distinct`
- `cogroup`
- `intersection`
- `join`
- `sortByKey`
- `coalesce`

How Shuffle works in Spark

Shuffle Read

- **HashMap used in Shuffle Read**
 - AppendOnlyMap
 - ExternalAppendOnlyMap

How Shuffle works in Spark

Shuffle Write

- Add the shuffle write logic at the end of ShuffleMapStage (in which there's a ShuffleMapTask)
- Each output record of the final RDD in this stage is partitioned and persisted
- The persistence of data here has two advantages: reducing heap pressure and enhancing fault-tolerance

How Shuffle works in Spark

Shuffle Comparison between Hadoop and Spark

- **Similarities**

- Both partition the mapper's output and send each partition to its corresponding reducer. The reducer buffers the data in memory, shuffles and aggregates the data, and applies the `reduce()` logic once the data is aggregated

How Shuffle works in Spark

Shuffle Comparison between Hadoop and Spark

- **Differences**

- The steps in a Hadoop workflow: map, spill, merge, shuffle, sort and reduce(). Each step has a predefined responsibility and it fits the procedural programming model well.
- There're no such fixed steps in Spark, instead stages and a series of transformations. So operations like spill, merge and aggregate need to be somehow included in the transformations.

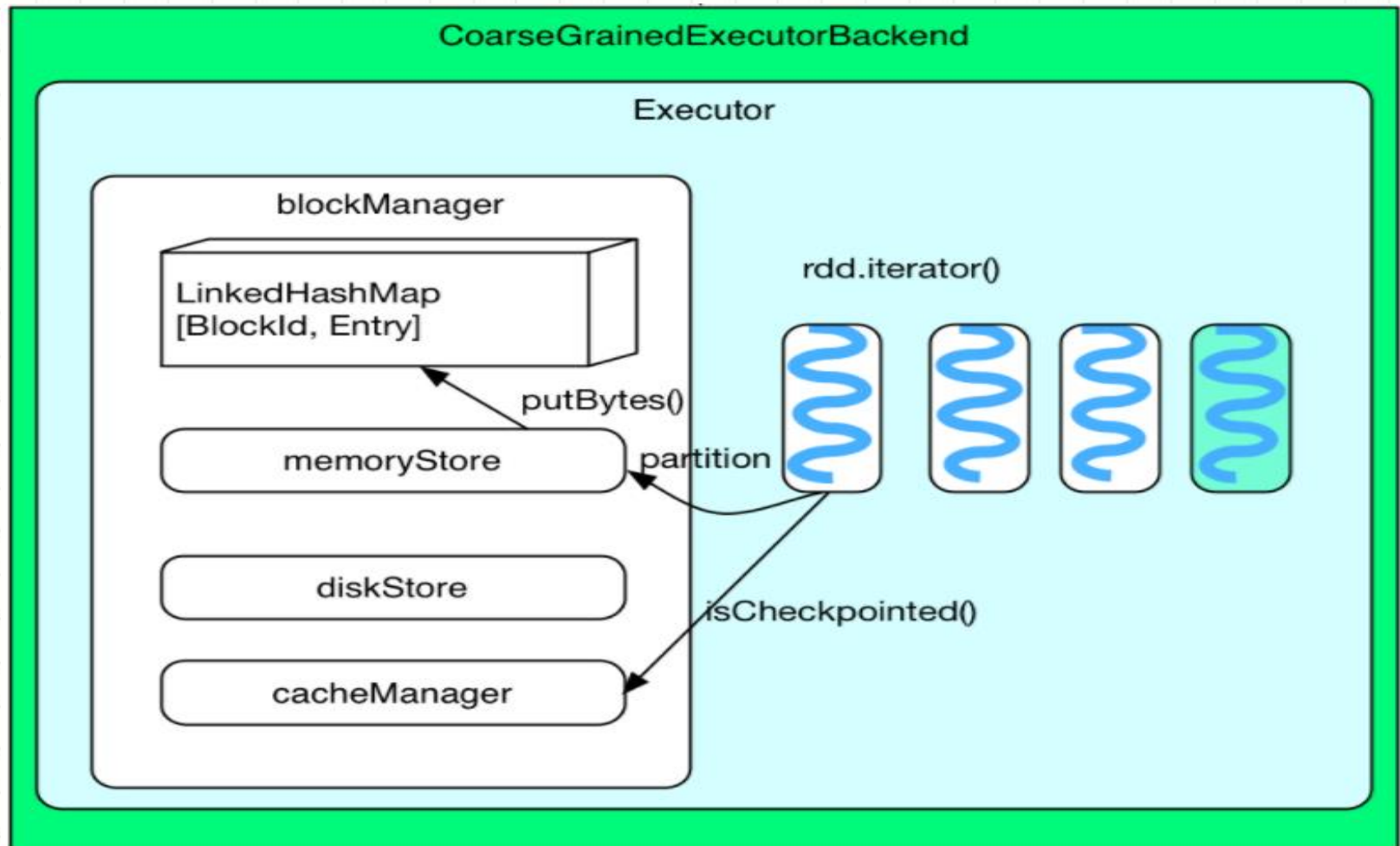
Cache Mechanism

- **What kind of RDD needs to be cached**
 - Those which will be repeatedly computed and are not too large
- **How to cache an RDD**
 - Driver program: **rdd.cache()**
 - RDD produced by user and transformation can be cached
 - RDD produced by Spark (not user) during the execution of a transformation can not be cached

Cache Mechanism

- **How does Spark cache RDD**
 - Spark will test whether the RDD should be cached or not just before computing the first partition. If the RDD should be cached, the partition will be computed and cached into memory. cache only uses memory.
 - RDD becomes persistRDD whose storage Level is MEMORY_ONLY

Cache Mechanism



Cache Mechanism

- How does Spark cache RDD

```
rdd.iterator()
=> SparkEnv.get.cacheManager.getOrCompute(thisRDD, split, context, storageLevel)
=> key = RDDBlockId(rdd.id, split.index)
=> blockManager.get(key)
=> computedValues = rdd.computeOrReadCheckpoint(split, context)
    if (isCheckedpointed) firstParent[T].iterator(split, context)
    else compute(split, context)
=> elements = new ArrayBuffer[Any]
=> elements += computedValues
=> updatedBlocks = blockManager.put(key, elements, tellMaster = true)
```

Cache Mechanism

- **How to read cached RDD**
 - The storage location of cached partition: the blockManager of the node on which a partition is cached will notify the blockManagerMasterActor on master by saying that an RDD partition is cached
 - The data will be stored in the blockLocations: HashMap of blockMangerMasterActor
 - When a task needs a cached RDD, it will send blockManagerMaster.getLocations(blockId) request to driver to get the partition's location, and the driver will lookup blockLocations to send back location info

Checkpoint Mechanism

Checkpointing is a process of truncating RDD lineage graph and saving it to a reliable distributed (HDFS) or local file system

There are two types of checkpointing:

- reliable - in Spark (core), RDD checkpointing that saves the actual intermediate RDD data to a reliable distributed file system, e.g. HDFS
- local - in Spark Streaming - RDD checkpointing that truncates RDD lineage graph

It's up to a Spark application developer to decide when and how to checkpoint using `RDD.checkpoint()` method

Before checkpointing is used, a Spark developer has to set the checkpoint directory using `SparkContext.setCheckpointDir(directory: String)` method

Checkpoint Mechanism

- **What kind of RDD needs checkpoint**
 - The computation takes a long time
 - The computing chain is too long
 - Depends too many RDDs

Checkpoint Mechanism

- **When to checkpoint**
 - Use `rdd.checkpoint()`, but checkpoint waits until the end of a job, and launches another job to finish checkpoint
 - An RDD which needs to be checkpointed will be computed twice; thus it is suggested to do a `rdd.cache()` before `rdd.checkpoint()`.

Checkpoint Mechanism

- **How to implement checkpoint**

Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed

- User should set the storage path for check point (on hdfs).
- On driver side, after `rdd.checkpoint()` is called, the RDD will be managed by `RDDCheckpointData`
- After initialization, `RDDCheckpointData` will mark RDD `MarkedForCheckpoint`

Checkpoint Mechanism

- **How to implement checkpoint**

Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed

- When a job is finished, `finalRdd.doCheckpoint()` will be called. `finalRDD` scans the computing chain backward. When meeting an RDD which needs to be checkpointed, the RDD will be marked `CheckpointingInProgress`
- When a job is finished, `finalRdd.doCheckpoint()` will be called. `finalRDD` scans the computing chain backward. When meeting an RDD which needs to be checkpointed, the RDD will be marked `CheckpointingInProgress`

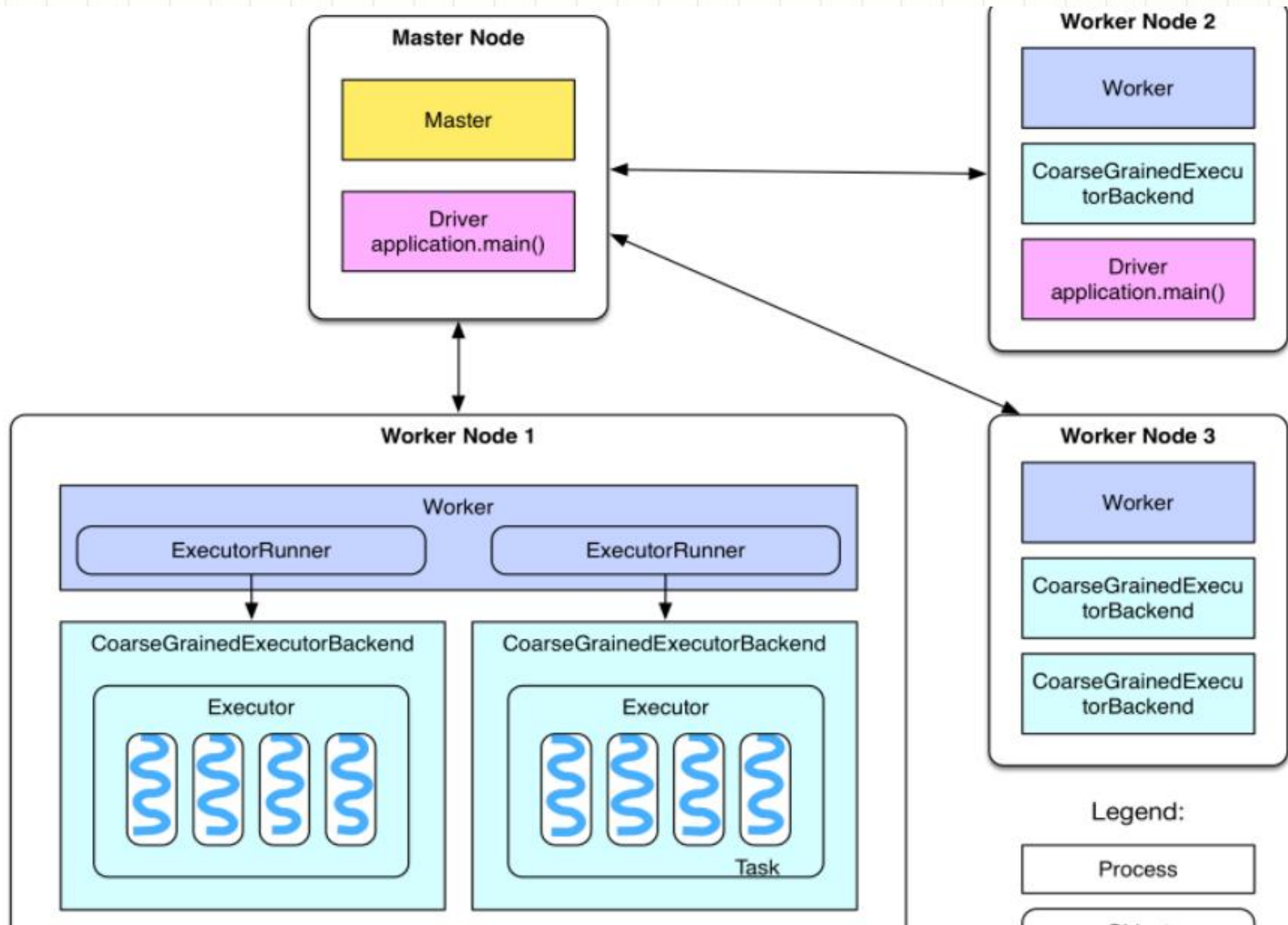
Checkpoint Mechanism

- **How to read checkpointed RDD**
 - `runJob()` will call `finalRDD.partitions()` to determine how many tasks there will be
 - `rdd.partitions()` checks if the RDD has been checkpointed, If yes, return the partitions of the RDD (`Array[Partition]`)
 - When `rdd.iterator()` is called to compute RDD's partition, `computeOrReadCheckpoint(split: Partition)` is also called to check if the RDD is checkpointed, If yes, the parent RDD's `iterator()`, a.k.a `CheckpointRDD.iterator()` will be called
 - `CheckpointRDD` reads files on file system to produce RDD partition

Cache and Checkpoint

- **The difference between cache and checkpoint**
 - Cache materializes the RDD and keeps it in memory (and/or disk) and the lineage of RDD will be remembered
 - Checkpoint saves the RDD to an HDFS file and actually forgets the lineage completely
 - Cache `rdd.persist(StorageLevel.DISK_ONLY)` persists RDD partitions to disk but once driver program finishes, the RDD cached to disk will be dropped
 - Checkpoint will persist RDD to HDFS or local directory. If not removed manually, they will always be on disk, so they can be used by the next driver program

Spark Execution Model



Spark Execution Model

What are the running services in each node after submit the application

- A Spark cluster has a Master node and multiple Worker nodes
- Each worker manages one or multiple ExecutorBackend processes
- Each ExecutorBackend launches and manages an Executor instance
- Each Executor maintains a thread pool, in which each task runs as a thread
- Each application has one Driver and multiple Executors
- Worker manages each CoarseGrainedExecutorBackend process through an ExecutorRunner instance

Spark Execution Model

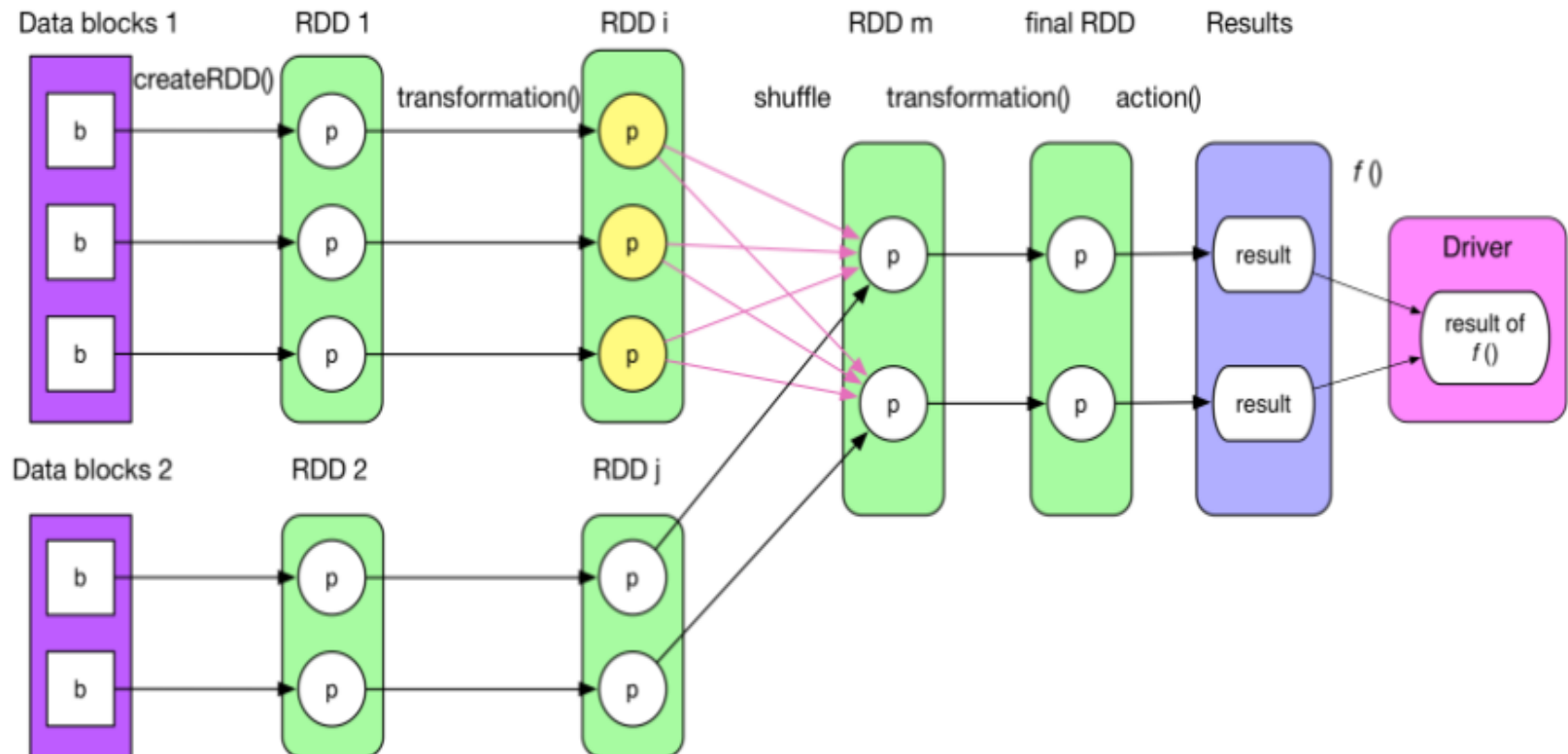
How is the Spark application created and executed

- Creates a logical plan (dependency graph) for each application
- Transforms the logical plan into a physical plan (a DAG graph of map/reduce stages and map/reduce tasks)
- Concrete map/reduce tasks will be launched to process the input data.
-

Spark Execution Model

Generate Job Logic Plan

- Example



Spark Execution Model

Generate Job Logic Plan

- Four steps to generate final results for logic plan
 - Create the initial RDD from any source
 - A series of transformation operations on RDD
 - Action operation is called on the final RDD then each partition generates computing result
 - Results will be sent to the driver

Spark Execution Model

Generate Job Logic Plan

- How to produce RDDs
 - Logical plan is a computing chain
 - Every RDD has a `compute()` method, which reads input records from the previous RDD or data source, performs the transformation
 - A transformation produces a new RDD
 - Some transformations can produce multiple RDDs

Spark Execution Model

Generate Job Logic Plan

- What RDD should be produced
 - Depends on the computing logic of the transformation

| Transformation | Generated RDDs | Compute() |
|--|-------------------------|--|
| <code>map(func)</code> | MappedRDD | <code>iterator(split).map(f)</code> |
| <code>filter(func)</code> | FilteredRDD | <code>iterator(split).filter(f)</code> |
| <code>flatMap(func)</code> | FlatMappedRDD | <code>iterator(split).flatMap(f)</code> |
| <code>mapPartitions(func)</code> | MapPartitionsRDD | <code>f(iterator(split))</code> |
| <code>mapPartitionsWithIndex(func)</code> | MapPartitionsRDD | <code>f(split.index, iterator(split))</code> |
| <code>sample(withReplacement, fraction, seed)</code> | PartitionwiseSampledRDD | <code>PoissonSampler.sample(iterator(split))</code> <code>BernoulliSampler.sample(iterator(split))</code> |
| <code>pipe(command, [envVars])</code> | PipedRDD | |

Spark Execution Model

Generate Job Logic Plan

- What RDD should be produced
 - Depends on the computing logic of the transformation

| | | |
|---|--|--|
| <code>union(otherDataset)</code> | | |
| <code>intersection(otherDataset)</code> | | |
| <code>distinct([numTasks])</code> | | |
| <code>groupByKey([numTasks])</code> | | |
| <code>reduceByKey(func, [numTasks])</code> | | |
| <code>sortByKey([ascending], [numTasks])</code> | | |
| <code>join(otherDataset, [numTasks])</code> | | |
| <code>cogroup(otherDataset, [numTasks])</code> | | |
| <code>cartesian(otherDataset)</code> | | |
| <code>coalesce(numPartitions)</code> | | |
| <code>repartition(numPartitions)</code> | | |

Spark Execution Model

Generate Job Logic Plan

- How to build data dependency between RDDs
 - A RDD depends on one parent RDD or several parent RDDs
 - The number of partitions for RDDs by default takes $\max(\text{partitionNum of parentRDD1}, \dots, \text{partitionNum of parentRDDn})$, can be defined by user
 - Different transformations have different data dependencies
 - The relationship of RDD partitions are NarrowDependency and ShuffleDependency

Spark Execution Model

Generate Job Logic Plan

- Transformation in driver program builds a computing chain (a series of RDD)
- RDD `compute()` function defines the computation of records for its partitions
- RDD `getDependencies()` function defines the dependency relationship across RDD partitions.

Spark Execution Model

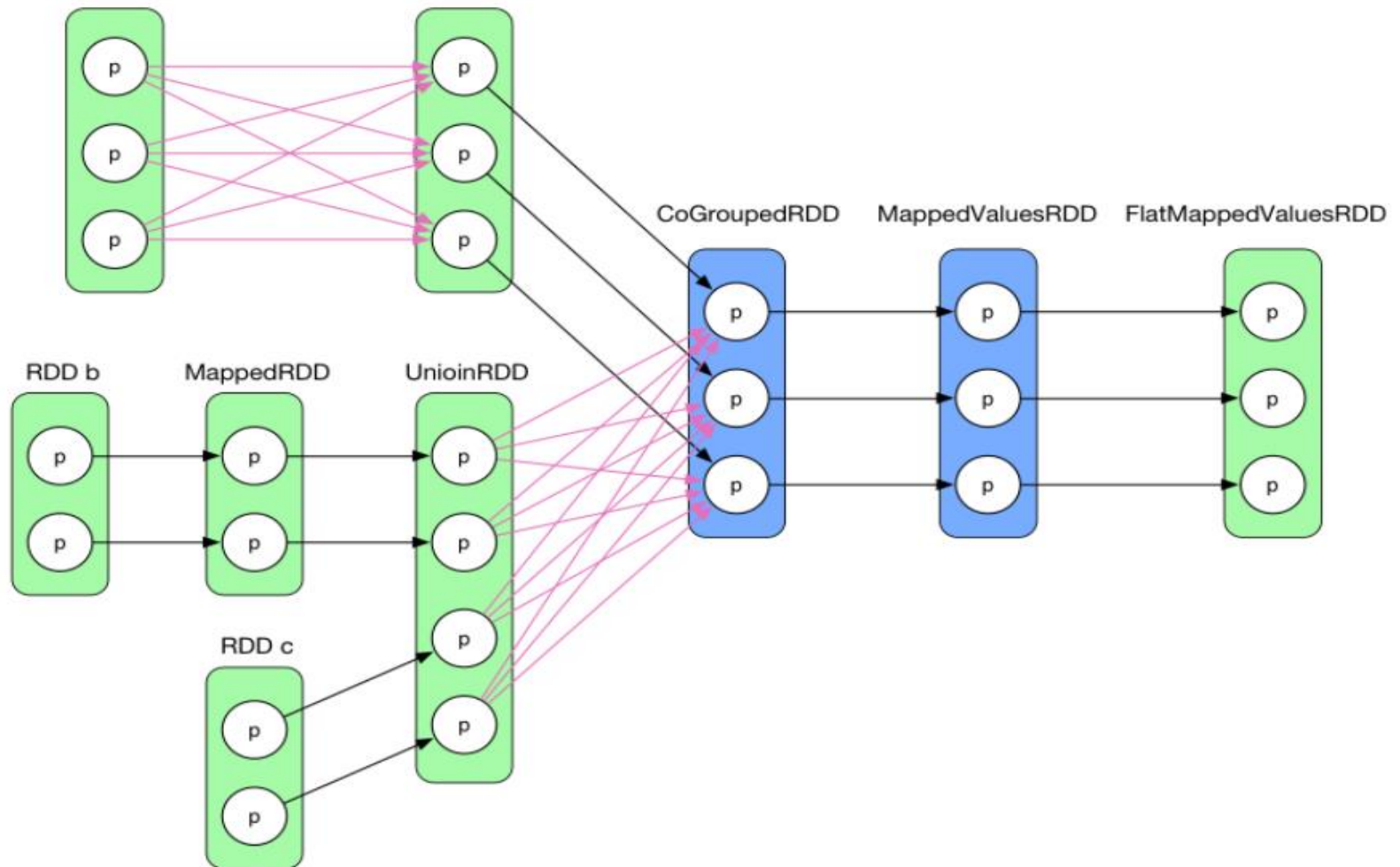
Generate Job Logic Plan

- `RDD.toDebugString` can return the logical plan

```
MapPartitionsRDD[3] at groupByKey at GroupByTest.scala:51 (36 partitions)
  ShuffledRDD[2] at groupByKey at GroupByTest.scala:51 (36 partitions)
    FlatMappedRDD[1] at flatMap at GroupByTest.scala:38 (100 partitions)
      ParallelCollectionRDD[0] at parallelize at GroupByTest.scala:38 (100 partitions)
```

Spark Execution Model

Generate Job Physical Plan



Spark Execution Model

Generate Job Physical Plan

- Generate stages and tasks
 - DAG-like physical plan, which contains stages and tasks
 - Associate one RDD and its preceding RDD to form a stage
 - Strategy for creating stages is to check backwards from the final RDD, add each NarrowDependency into the current stage, and break out for a new stage when there's a ShuffleDependency
 - The task number is determined by the partition number of the last RDD in the stage
 - If a stage generates the final result, the tasks in this stage are of type ResultTask, otherwise they are ShuffleMapTask

Spark Execution Model

Generate Job Physical Plan

- Generate stages and tasks
 - The whole computation chain is created by checking backwards the data dependency from the last RDD. Each ShuffleDependency separates stages. In each stage, each RDD's `compute()` method calls `parentRDD.iterator()` to receive the upstream record stream.

Spark Execution Model

Generate Job Physical Plan

- Each action() triggers a job
 - During dagScheduler.runJob(), different stages are defined
 - During submitStage(), ResultTasks and ShuffleMapTasks needed by the stage are produced, then they are packaged in TaskSet and sent to TaskScheduler
 - If TaskSet can be executed, tasks will be submitted to sparkDeploySchedulerBackend which will distribute tasks.

Spark Execution Model

Submit and Schedule the Job

- Job creation
 - Each time there is an `action()` in user's driver program, a job will be created

| Action | <code>finalRDD(records) => result</code> | <code>compute(results)</code> |
|---|---|--|
| <code>reduce(func)</code> | <code>(record1, record2) => result, (result, record i) => result</code> | <code>(result1, result 2) => result, (result, result i) => result</code> |
| <code>collect()</code> | <code>Array(records) => result</code> | <code>Array(result)</code> |
| <code>count()</code> | <code>count(records) => result</code> | <code>sum(result)</code> |
| <code>foreach(f)</code> | <code>f(records) => result</code> | <code>Array(result)</code> |
| <code>take(n)</code> | <code>record (i<=n) => result</code> | <code>Array(result)</code> |
| <code>first()</code> | <code>record 1 => result</code> | <code>Array(result)</code> |
| <code>takeSample()</code> | <code>selected records => result</code> | <code>Array(result)</code> |
| <code>takeOrdered(n, [ordering])</code> | <code>TopN(records) => result</code> | <code>TopN(results)</code> |
| <code>saveAsHadoopFile(path)</code> | <code>records => write(records)</code> | <code>null</code> |
| <code>countByKey()</code> | <code>(K, V) => Map(K, count(K))</code> | <code>(Map, Map) => Map(K, count(K))</code> |

Spark Execution Model

Submit and Schedule the Job

- Job submission
 - `rdd.action()` calls `DAGScheduler.runJob` to create a job
 - `runJob()` gets the partition number and type of the final RDD by calling `rdd.getPartitions()`
 - `runJob(rdd, cleanedFunc, partitions, allowLocal, resultHandler)` in `DAGScheduler` is called to submit the job. `processPartition` will be serialized and sent to the different worker nodes
 - `DAGScheduler's runJob()` calls `submitJob(rdd, func, partitions, allowLocal, resultHandler)` to submit a job
 - `submitJob()` gets a `jobId`, then the actor calls `dagScheduler.handleJobSubmitted()` to handle the submitted job

Spark Execution Model

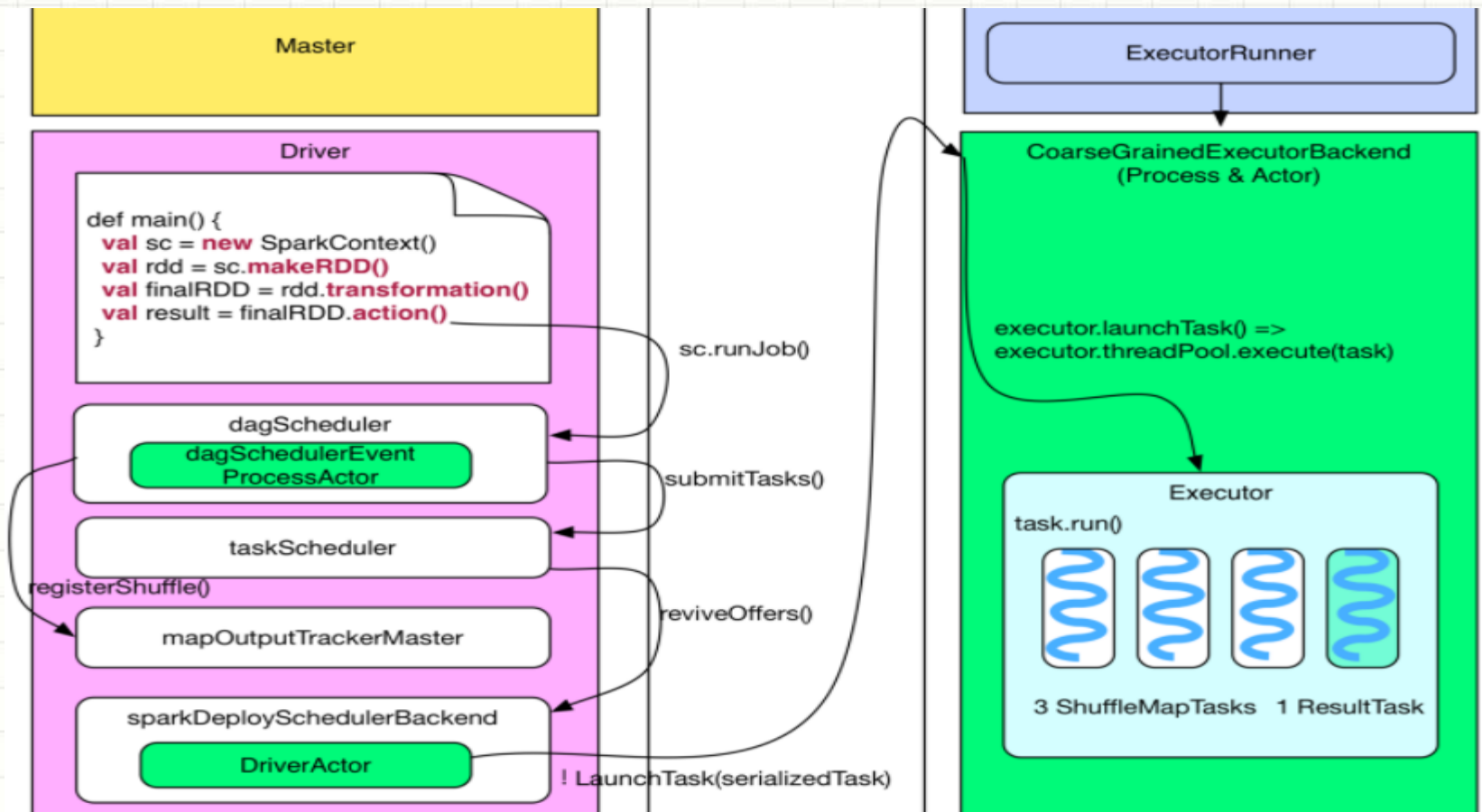
Submit and Schedule the Job

- Job submission
 - `handleJobSubmitted()` firstly calls `finalStage = newStage()` to create stages, then it `submitStage(finalStage)`
 - If `finalStage` has parents, the parent stages will be submitted first

Spark Execution Model

Submit and Schedule the Job

- Job submission



Spark Execution Model

Submit and Schedule the Job

- Job submission

```
finalRDD.action()
=> sc.runJob()

// generate job, stages and tasks
=> dagScheduler.runJob()
=> dagScheduler.submitJob()
=> dagSchedulerEventProcessActor ! JobSubmitted
=> dagSchedulerEventProcessActor.JobSubmitted()
=> dagScheduler.handleJobSubmitted()
=> finalStage = newStage()
=> mapOutputTracker.registerShuffle(shuffleId, rdd.partitions.size)
=> dagScheduler.submitStage()
=> missingStages = dagScheduler.getMissingParentStages()
=> dagScheduler.subMissingTasks(readyStage)

// add tasks to the taskScheduler
=> taskScheduler.submitTasks(new TaskSet(tasks))
=> fifoSchedulableBuilder.addTaskSetManager(taskSet)

// send tasks
=> sparkDeploySchedulerBackend.reviveOffers()
=> driverActor ! ReviveOffers
=> sparkDeploySchedulerBackend.makeOffers()
=> sparkDeploySchedulerBackend.launchTasks()
=> foreach task
    CoarseGrainedExecutorBackend(executorId) ! LaunchTask(serializedTask)
```


Spark Execution Model

Submit and Schedule the Job

- Create, Schedule and Launch Task
 - The tasks in a stage form a TaskSet.
taskScheduler.submitTasks(taskSet) is called to submit the whole task set
 - Either FIFOSchedulableBuilder or FairSchedulableBuilder is used to schedule the tasks
 - launchTasks() in the Driver serialize each task and finally sent to the executor for execution

Spark Execution Model

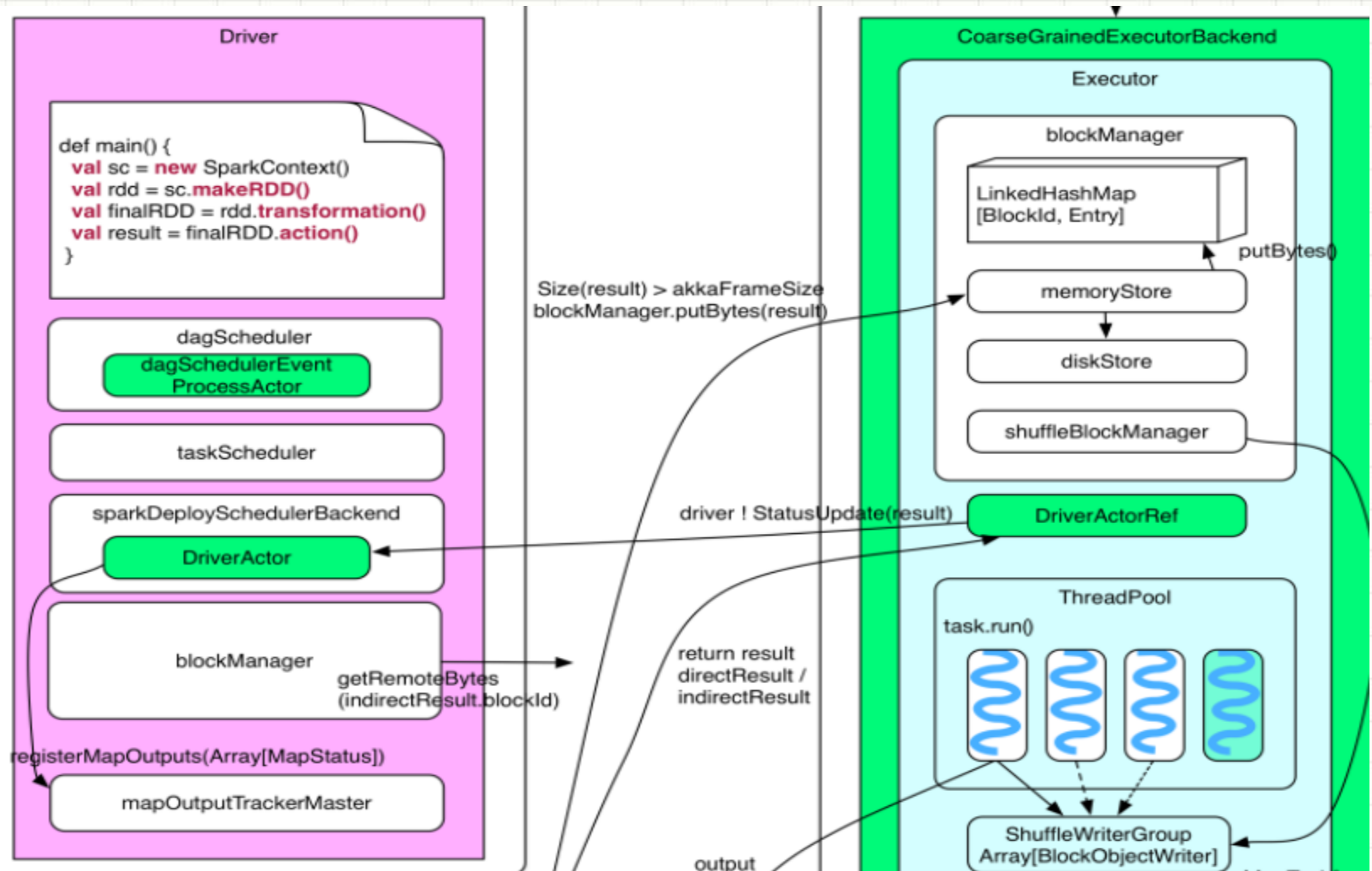
Job Reception

- After receiving tasks, worker will do the following things

```
coarseGrainedExecutorBackend ! LaunchTask(serializedTask)
=> executor.launchTask()
=> executor.threadPool.execute(new TaskRunner(taskId, serializedTask))
```

Spark Execution Model

Task Execution



Spark Execution Model

Task Execution

```
In TaskRunner.run()
// deserialize task, run it and then send the result to
=> coarseGrainedExecutorBackend.statusUpdate()
=> task = ser.deserialize(serializedTask)
=> value = task.run(taskId)
=> directResult = new DirectTaskResult(ser.serialize(value))
=> if( directResult.size() > akkaFrameSize() )
    indirectResult = blockManager.putBytes(taskId, directResult, MEMORY+DISK+SER)
    else
        return directResult
=> coarseGrainedExecutorBackend.statusUpdate(result)
=> driver ! StatusUpdate(executorId, taskId, result)
```

```
In task.run(taskId)
// if the task is ShuffleMapTask
=> shuffleMapTask.runTask(context)
=> shuffleWriterGroup = shuffleBlockManager.forMapTask(shuffleId, partitionId, numOutputSplits)
=> shuffleWriterGroup.writers(bucketId).write(rdd.iterator(split, context))
=> return MapStatus(blockManager.blockManagerId, Array[compressedSize(fileSegment)])

//If the task is ResultTask
=> return func(context, rdd.iterator(split, context))
```

Spark Execution Model

Task Execution

```
After driver receives StatusUpdate(result)
=> taskScheduler.statusUpdate(taskId, state, result.value)
=> taskResultGetter.enqueueSuccessfulTask(taskSet, tid, result)
=> if result is IndirectResult
    serializedTaskResult = blockManager.getRemoteBytes(IndirectResult.blockId)
=> scheduler.handleSuccessfulTask(taskSetManager, tid, result)
=> taskSetManager.handleSuccessfulTask(tid, taskResult)
=> dagScheduler.taskEnded(result.value, result.accumUpdates)
=> dagSchedulerEventProcessActor ! CompletionEvent(result, accumUpdates)
=> dagScheduler.handleTaskCompletion(completion)
=> Accumulators.add(event.accumUpdates)

// If the finished task is ResultTask
=> if (job.numFinished == job.numPartitions)
    listenerBus.post(SparkListenerJobEnd(job.jobId, JobSucceeded))
=> job.listener.taskSucceeded(outputId, result)
=> jobWaiter.taskSucceeded(index, result)
=> resultHandler(index, result)

// If the finished task is ShuffleMapTask
=> stage.addOutputLoc(smt.partitionId, status)
=> if (all tasks in current stage have finished)
    mapOutputTrackerMaster.registerMapOutputs(shuffleId, Array[MapStatus])
    mapStatuses.put(shuffleId, Array[MapStatus]() ++ statuses)
=> submitStage(stage)
```

Spark Execution Model

- **Understanding Execution Model**

- WebUI
- RDD.toDebugString
- Logs
- Spark Listeners

Spark Execution Model

- **Understanding Execution Model using Spark Listeners**

- **Schedule Listeners**

- ❑ SparkListener is a class that listens to execution events from Spark's DAGScheduler
 - ❑ It extends `org.apache.spark.scheduler.SparkListener`
 - ❑ receive events about when applications, jobs, stages and tasks start or complete, drivers being added or removed, RDD is unpersisted, environment properties change

Spark Execution Model

- **Understanding Execution Model using Spark Listeners**

- **spark.extraListeners**

- ❑ comma-separated list of listener class names that are registered with Spark's listener bus when SparkContext is initialized

- **StatsReportListener**

- ❑ a SparkListener that logs summary statistics when a stage completes

Spark Memory Management

Type of Memory

- Heap Memory
- Off Heap Memory

Spark Memory Management

Executor memory overview

- An executor is the Spark application's JVM process launched on a worker node. It runs tasks in threads and is responsible for keeping relevant partitions of data. Each process has an allocated heap with available memory (executor/driver).
 - **spark.executor.memory** – specifies the executor's process memory heap (default 1 GB)
 - **spark.driver.memory** – specifies the driver's process memory heap (default 1 GB)
 - **spark.memory.fraction** – a fraction of the heap space (minus 300 MB * 1.5) reserved for execution and storage regions (default 0.6)

Spark Memory Management

Off-heap

- Off-heap refers to objects (serialised to byte array) that are managed by the operating system but stored outside the process heap in native memory (therefore, they are not processed by the garbage collector).
- Off-heap memory usage is available for execution and storage regions
 - **spark.memory.offHeap.enabled** – the option to use off-heap memory for certain operations (default false)
 - **spark.memory.offHeap.size** – the total amount of memory in bytes for off-heap allocation. It has no impact on heap memory usage, so make sure not to exceed your executor's total limits (default 0)

Spark Memory Management

Spark Memory Management Model

- before 1.6: StaticMemoryManager
- 1.6+: UnifiedMemoryManager

Spark Memory Management

Memory Regions

- Reserved Memory
 - Memory reserved by the system
 - Its value is 300MB
 - Its size cannot be changed in any way without Spark recompilation or setting `spark.testing.reservedMemory`
 - It is not used by Spark
 - Sets the limit on what you can allocate for Spark usage
 - Store lots of Spark internal objects
 - Need to give Spark executor at least $1.5 * \text{Reserved Memory} = 450\text{MB}$ heap

Spark Memory Management

Memory Regions

- User Memory
 - Your own data structures there that would be used in RDD transformations
 - $(\text{"Java Heap"} - \text{"Reserved Memory"}) * (1.0 - \text{spark.memory.fraction})$
 - Default value is $(\text{"Java Heap"} - 300\text{MB}) * 0.25$

Spark Memory Management

Memory Regions

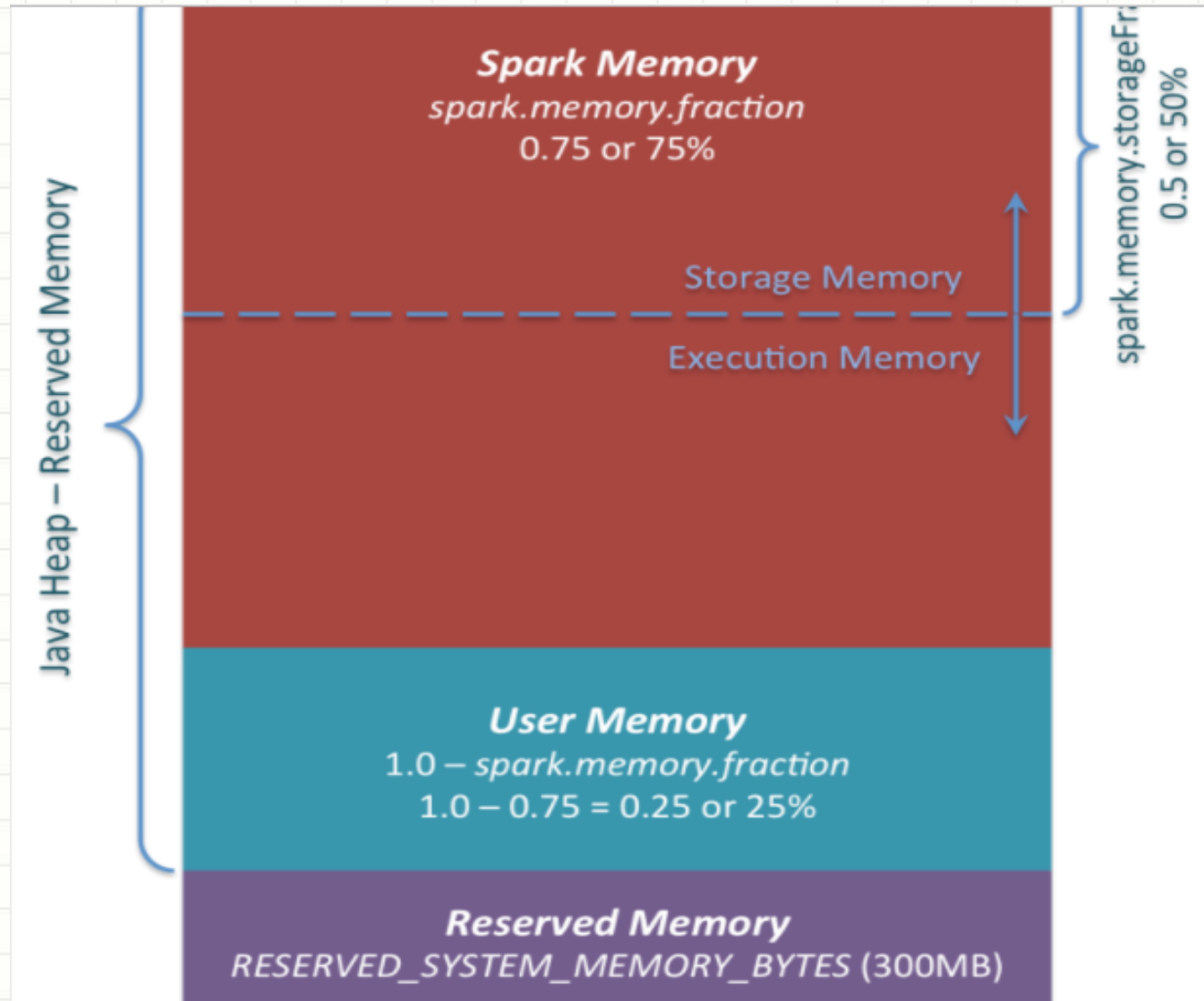
- Spark Memory
 - The memory pool managed by Apache Spark
 - $(\text{"Java Heap"} - \text{"Reserved Memory"}) * \text{spark.memory.fraction}$
 - Default is $(\text{"Java Heap"} - 300\text{MB}) * 0.75$
 - Storage Memory and Execution Memory, and the boundary between them is set by `spark.memory.storageFraction` parameter, which defaults to 0.5.
 - This boundary is not static, region would grow by borrowing space from another one

Spark Memory Management

Spark Memory

- **Storage Memory**
 - Storing Apache Spark cached data and for temporary space serialized data “unroll”
 - All the “broadcast” variables
- **Execution Memory**
 - Storing the objects required during the execution of Spark tasks.
 - Shuffle, joins, sorts and aggregation
 - Store shuffle intermediate buffer on the Map side in memory
 - Store hash table for hash aggregation step
 - Supports spilling on disk if not enough memory is available

Spark Memory Management



Spark Memory Management

The moving boundary between Storage Memory and Execution Memory

- Can forcefully evict the block from Storage Memory, but cannot do so from Execution Memory
- Storage Memory pool can borrow some space from Execution Memory pool only if there is some free space in Execution Memory pool available

Spark Memory Management

When Execution Memory pool can borrow some space from Storage Memory

- There is free space available in Storage Memory pool
- Storage Memory pool size exceeds the initial Storage Memory region size and it has all this space utilized

Spark Memory Management

Spark Memory Challenges

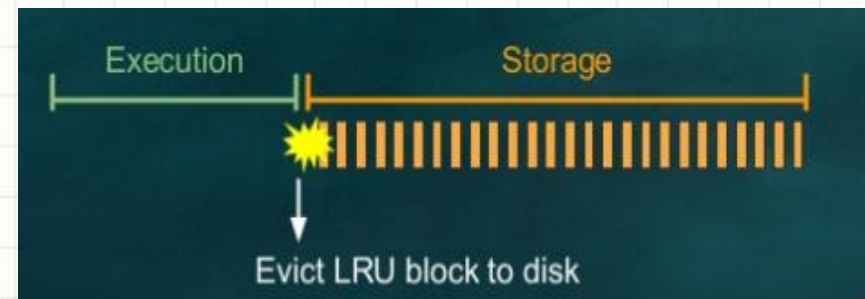
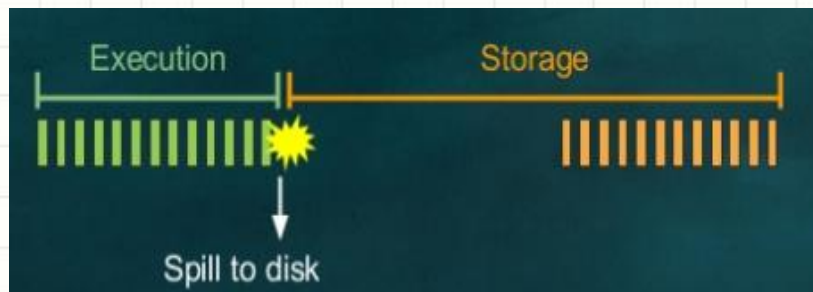
- How to arbitrate memory between execution and storage
- How to arbitrate memory across tasks running in parallel
- How to arbitrate memory across operators running within the same tasks

Spark Memory Management

Spark Memory Challenges

- How to arbitrate memory between execution and storage

➤ Static Memory Management



Spark Memory Management

Spark Memory Challenges

- How to arbitrate memory between execution and storage
 - Unified Memory Management



Spark Memory Management

Spark Memory Challenges

- How to arbitrate memory across tasks running in parallel
 - Static assignment - share the total memory evenly
 - Dynamic assignment - the share of each task depends on number of actively running tasks

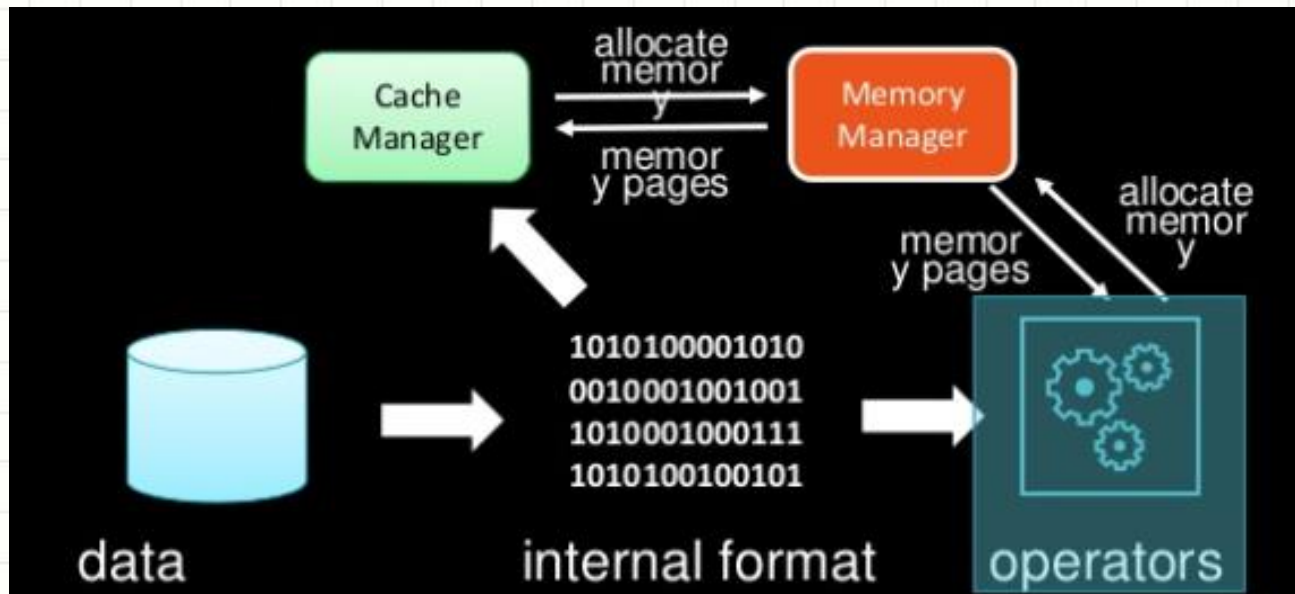
Spark Memory Management

Spark Memory Challenges

- How to arbitrate memory across operators running within the same tasks
 - Reserve a page for each operator (Sort, Aggregation, Project and Scan)
 - Cooperative spilling

Spark Memory Management

Spark Memory Model inside Executor



Spark Tungsten

Goals

- Improve the memory and CPU efficiency of spark application's
- push performance closer to the limit of modern hardware

Spark Tungsten

CPU is the new bottleneck

- Hardware has improved
- Spark's IO has been optimized
- Data formats have improved
- Serialization and hashing are CPU-bound bottlenecks

Spark Tungsten

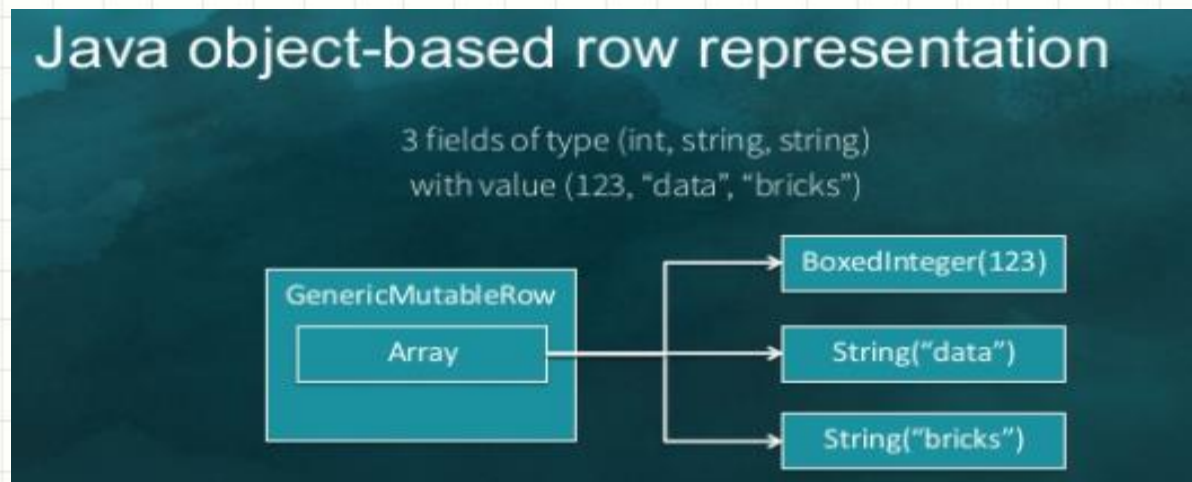
How Tungsten improves CPU & memory efficiency

- Memory Management and Binary Processing
- Cache-aware computation
- Code generation

Spark Tungsten

Binary in-memory data representation

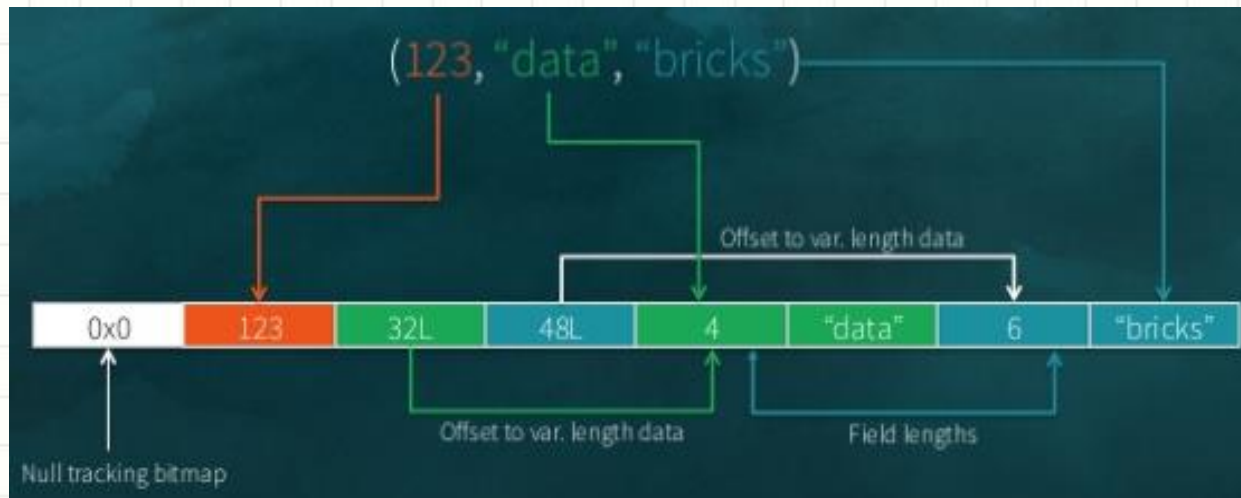
- Java objects based row format
 - Java objects have large overhead, high space overhead, expensive hash code



Spark Tungsten

Binary in-memory data representation

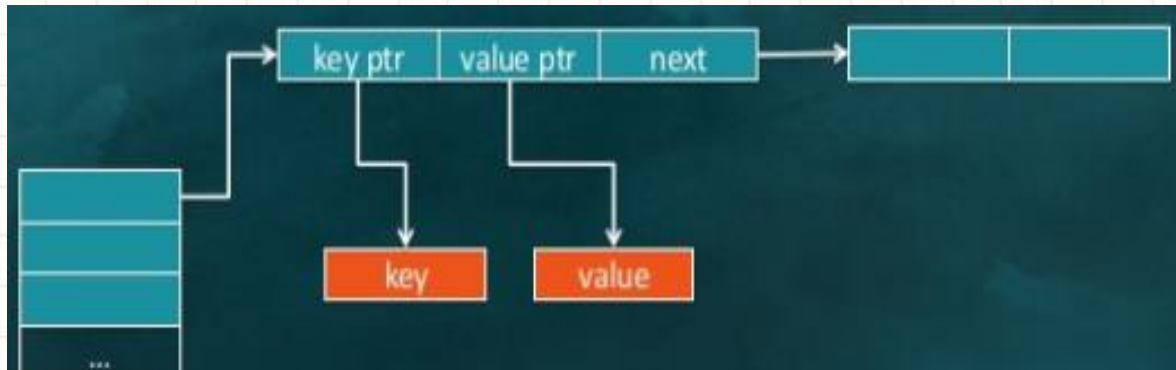
- Tungsten row format



Spark Tungsten

Binary in-memory data representation

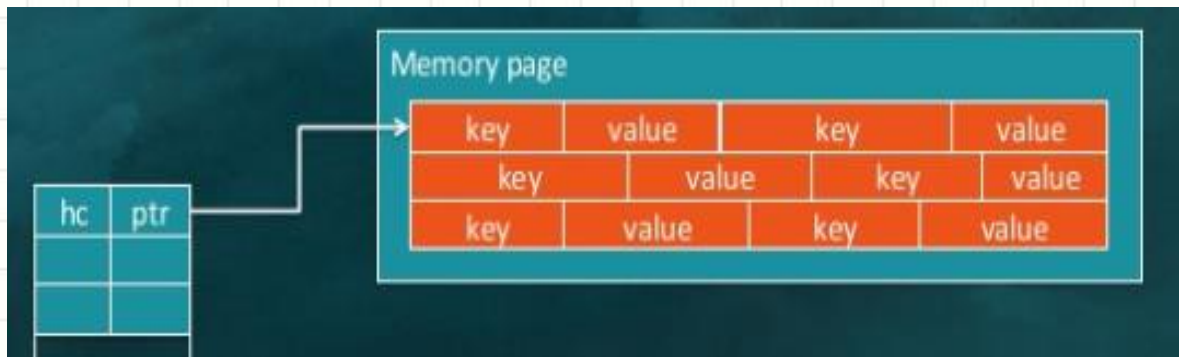
- Java HashMap
 - Huge object overheads
 - Poor memory locality
 - Size estimation is hard



Spark Tungsten

Binary in-memory data representation

- Tungsten BytesToBytesMap
 - Low space overhead
 - Good memory locality, especially for scans



Spark Tungsten

How to process binary data more efficiently

- **Understanding CPU Cache**
 - Memory become slower than CPU
 - Pre-fetch frequently accessed data into CPU cache
- **Two Important Algorithms in Big Data**
 - Sort
 - Hash

Spark Tungsten

How to process binary data more efficiently

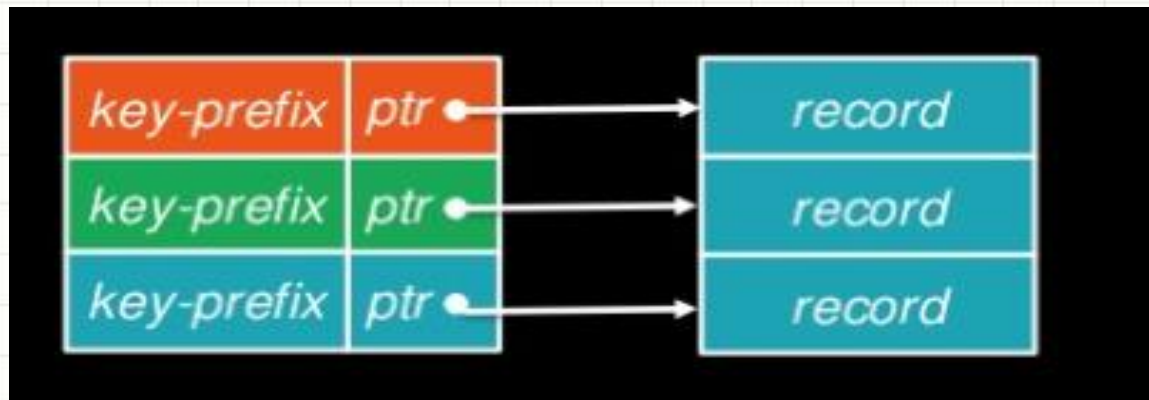
- **Naive Sort**

Each comparison needs to access 2 different memory regions, which makes it hard for CPU cache to pre-fetch data, poor cache locality

Spark Tungsten

How to process binary data more efficiently

- Cache-aware Sort



Spark Tungsten

How to process binary data more efficiently

- Cache-aware computation



Spark Tungsten

How to process binary data more efficiently

- **Naive Hash Map**

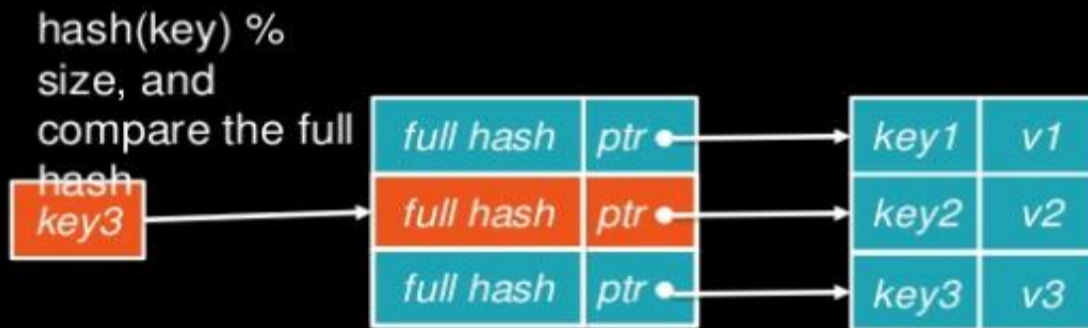
Each lookup needs many pointer dereference and key comparison when hash collision happens, and jumps between 2 memory regions, bad cache locality

Spark Tungsten

How to process binary data more efficiently

- Cache-aware Hash Map

Cache-aware Hash Map



Spark Tungsten

Off-heap memory

- memory sharing
- zero copy I/O
- dynamic allocation

Spark Tungsten

Code generation

- Generic evaluation of expression logic is very expensive on the JVM
- Tungsten uses the Janino compiler to reduce code generation time

Spark Tungsten

Which Spark jobs can benefit from Tungsten

- DataFrame
- Spark SQL queries
- Some Spark RDD API

Spark Tungsten

Enable Tungsten optimizations

- `spark.sql.codegen = true`
- `spark.sql.unsafe.enabled = true`
- `spark.shuffle.manager=tungsten-sort`

Spark Tungsten

How to encode memory addresses

- Off heap: addresses are raw memory pointers
- On heap: addresses are base object + offset pairs
- Page table to enable more compact encoding of on-heap addresses

Spark Job Scheduling

Scheduling Across Applications

When running on a cluster, each Spark application gets an independent set of executor JVMs that only run tasks and store data for that application. If multiple users need to share your cluster, there are different options to manage allocation, depending on the cluster manager.

- **Static Resource Allocation - YARN**
 - The `--num-executors` option to the Spark YARN client controls how many executors it will allocate on the cluster (`spark.executor.instances` as configuration property), while `--executor-memory` (`spark.executor.memory` configuration property) and `--executor-cores` (`spark.executor.cores` configuration property) control the resources per executor

Spark Job Scheduling

Scheduling Across Applications

- **Dynamic Resource Allocation**

- Spark feature that allows for adding or removing Spark executors dynamically to match the workload. You get as much as needed and no more. It scales the number of executors up and down based on workload
- Dynamic allocation is enabled using `spark.dynamicAllocation.enabled` setting
- External Shuffle Service needs to be enabled via `spark.shuffle.service.enabled` setting

Spark Job Scheduling

Scheduling Across Applications

- **Dynamic Resource Allocation**

- Request Policy

- ✓ `spark.dynamicAllocation.initialExecutors`
 - ✓ `spark.dynamicAllocation.minExecutors`
 - ✓ `spark.dynamicAllocation.maxExecutors`
 - ✓ `spark.dynamicAllocation.schedulerBacklogTimeout`
 - ✓ `spark.dynamicAllocation.sustainedSchedulerBacklogTimeout`

- Remove Policy

- ✓ `spark.dynamicAllocation.executorIdleTimeout`

Spark Job Scheduling

Scheduling Across Applications

- **Dynamic Resource Allocation**

- Graceful Decommission of Executors

- ✓ Decommission an executor gracefully by preserving its state before removing it
- ✓ Preserving shuffle files is to use an external shuffle service
- ✓ In addition to writing shuffle files, executors also cache data either on disk or in memory

Spark Job Scheduling

Scheduling Within Applications

Inside a given Spark application (SparkContext instance), multiple parallel jobs can run simultaneously if they were submitted from separate threads

- **Fair Scheduler**

- `conf.set("spark.scheduler.mode", "FAIR")`

- **Fair Scheduler Pool**

```
<?xml version="1.0"?>
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>
```


Dynamic Allocation (of Executors)

- Spark instrumentation
 - Web UI
 - REST API
 - Eventlog
 - Executor/Task Metrics
 - Dropwizard metrics library
 - Spline
 - Sparklint
 - SparkMeasure
- Complement with
 - OS tools

Performance Tuning

- **Data Serialization**

- Serialization significantly impacts the performance of any distributed application
- Kryo is significantly faster and more compact than Java serialization, switch to using Kryo by calling `conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`

- **Memory Tuning**

- The best way to size the amount of memory consumption a dataset will require is to create an RDD, put it into cache, and look at the Storage page in the web UI
- Use `SizeEstimator`'s `estimate` method to estimate the memory consumption of an object

Performance Tuning

- **Data Structure Tuning**

- Avoid using java pointer-based data structures and wrapper objects
- Design your data structures to prefer arrays of objects, and primitive types
- Avoid nested structures with a lot of small objects and pointers when possible.
- Consider using numeric IDs or enumeration objects instead of strings for keys.
- Set the JVM flag `-XX:+UseCompressedOops` if memory is less than 32G

- **Serialized RDD Storage**

- if objects are still too large, store them in serialized form such as `MEMORY_ONLY_SER`

Performance Tuning

- **Garbage Collection Tuning**

- Adding `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps` to the Java options to collect statistics
- Try the G1GC garbage collector with `-XX:+UseG1GC`
- GC tuning flags for executors can be specified by setting `spark.executor.extraJavaOptions` in a job's configuration
- Try to use off-heap memory to avoid GC

- **Level of Parallelism**

- Recommend 2-3 tasks per CPU core
- 5 cores per executor
- Set the config property `spark.default.parallelism` to change the default

Performance Tuning

- **Broadcasting Large Variable**
- **Data Locality**
 - Spark prefers to schedule all tasks at the best locality level
 - If not possible to switch to best locality levels, Spark either waits until a busy CPU frees up to start a task on data on the same server or immediately start a new task in a farther away place

Spark and Monitoring Tools

- Spark instrumentation
 - Web UI
 - REST API
 - Eventlog
 - Executor/Task Metrics
 - Dropwizard metrics library
 - Spline
 - Sparklint
 - SparkMeasure
- Complement with
 - OS tools

Spark and Monitoring Tools

- Web UI
 - Info on Jobs, Stages, Executors, Metrics, SQL and Storage
 - Execution Plans and DAGs
 - Event Timeline - Show task execution details by activity and time
 - URL - <http://localhost:4040>
- REST API - Spark Metrics
 - History server URL + /api/v1/applications/ApplicationID

Spark and Monitoring Tools

- EventLog - Store Web UI History
 - Config - `spark.eventLog.enabled=true`
 - Config - `spark.eventLog.dir=<path>`
 - JSON files store info displayed by Spark History Server
- Spark Executor Task Metrics
 - `df.filter("Event='SparkListenerTaskEnd']").select("Task Metrics.*").printSchema`

Spark and Monitoring Tools

- Task Info, Accumulables, SQL Metrics
 - `df.filter("Event='SparkListenerTaskEnd']").select("Task Info.*").printSchema`
- EventLog Analytics Using Spark SQL
 - `spark.sql("select Name, sum(Value) as value from aggregatedStageMetrics group by Name order by Name").show()`

Enterprise Spark Solutions

Modification of Spark Application on the fly

- **Challenges**

- Change the function/parameters
- Switch to different data source

- **Solutions**

- Create REST Server to provide the function or parameter at the runtime, the spark application can communicate with the REST server to pull the parameters from or call the functions from
- Dynamic Data Source API to provide the data source selection at the runtime
- Custom Data Source

Enterprise Spark Solutions

Automatic checkpointing of Spark Application

- **Challenges**

- Organizing batch processing pipelines in Apache Spark and handling automatic checkpointing of intermediate results

- **Solutions**

- <https://github.com/bloomberg/spark-flow>

Best Practice

Executors, Cores and Memory

- 6 Nodes, 40 cores each and 64G RAM each
 - Spark submit parameters
 - number of executors (`--num-executors`)
 - cores for each executors (`--executor-cores`)
 - memory for each executor (`--executor-memory`)

Best Practice

Executors, Cores and Memory

- Consideration:
 - Running multiple tasks in the same JVM
 - Need to leave some memory overhead for OS and Hadoop daemons
 - YARM application master needs a core
 - 5 cores per executor, too much might cause bad HDFS I/O throughput
- Correct assignment: 7 executors, 8GB RAM each, 5 cores each executor

Best Practice

DAG Management

- Shuffles are to be avoided
 - Map Side Reducing if possible
 - Partitioning/bucketing
 - Do as much as possible with a single shuffle
 - Only send what you have to send
 - Avoid data skew
- ReduceByKey, aggregateByKey, foldByKey, combineByKey over GroupByKey
- Use the built in aggregateByKey() instead of writing your own aggregation

Best Practice

DAG Management

- Filter input earlier rather than later
- TreeReduce over Reduce
- Use Complex Types
- Degree of parallelism
 - If the number of mostly idle tasks are huge, then it's good to coalesce
 - If not using all slots in the cluster, repartition can increase parallelism

Best Practice

Choice of Serializer

- Serialization is sometimes a bottleneck during shuffling and caching
- Using the Kryo serializer is often faster

Best Practice

Cache Format

- MEMORY_ONLY - By Default
- MEMORY_ONLY_SER can help cut down on GC
- MEMORY_ANY_DISK can avoid expensive re-computations

Best Practice

Other Performance Tuning

- Switching to LZF compression can improve shuffle performance
- Turn on speculative execution to help prevent stragglers
- Make sure to give Spark as many as disks as possible to allow striping shuffle output (SPARK_LOCAL_DIRS)

Best Practice

Execution on the Driver vs Worker

- Transformations are executed on the Spark worker
- Jobs are triggered by the actions on the Spark driver
- Do not call `collect()` on a large RDD
 - Option1: Choose `take(n)`
 - Option2: Choose `saveAsTextFile()`

Best Practice

Commonly Serialization Errors

- Hadoop Writables
- Capturing a full Non-Serializable object
- Network Connections
- Map to/from a serialization form
- Copy the required serializable parts locally
- Create the connection on the worker

Best Practice

RDDs within RDDs

- map+get:
 - `rdd.map{(key,value)=>otherRdd.get(key)}` can be replaced with `rdd.join(otherRDD).map{}`
- map+map:
 - `rdd.map{e=>otherRDD.map{}}` can be replaced with `rdd.cartesian(otherRdd).map{}`

Best Practice

Writing a Large RDD to a Database

- Option 1: `foreachPartition`
- Option 2: `saveAsHadoopDataset`

Trouble Shooting

Size exceeds Integer.MAX_VALUE (BlockManager)

- Cause:
 - Spark shuffle block can not be greater than 2G
 - Default number of partitions for shuffling is 200
 - Low number of partitions leads to high shuffle block size

Trouble Shooting

Size exceeds Integer.MAX_VALUE (BlockManager)

- Solution:
 - Increase the number of partitions
 - ❑ Increase the value of `spark.sql.shuffle.partitions` in Spark SQL
 - ❑ `rdd.repartition()` or `rdd.coalesce()` in Spark applications
 - Get rid of skew in the dataset

Trouble Shooting

Size exceeds Integer.MAX_VALUE (BlockManager)

- Best Practice:
 - 128 MB per partitions
 - Spark uses a different data structure during shuffles when number of partitions is less than 2000 vs more than 2000
 - Don't have too few partitions, otherwise your job will be slow

Trouble Shooting

Spark jobs take quite a long time during join or shuffling

- Cause:
 - Data Skew
- Solution:
 - Key Salting - Two stage aggregation
 - Isolation Salting
 - Isolation Map Joins

Logging

- Logging Levels
 - OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE, ALL
- log4j.properties
 - Set up the default logging for Spark shell in conf/log4j.properties
- Setting Default Log Level Programmatically
- Setting Log Levels in Spark Applications
 - `Logger.getLogger("org").setLevel(Level.OFF)`