Introduction to Apache Spark

Incheon Paik

Contents

- **Spark Motivation**
- Spark Pillars
- Spark Architecture
- Spark Shuffle
- Spark DataFrame

Spark Motivation

- Difficultly of programming directly in Hadoop MapReduce
- Performance bottlenecks, or batch not fitting use cases
- Better support iterative jobs typical for machine learning

Performance Bottleneck

Spark offers you

- Lazy Computations
  - Optimize the job before executing
- In-memory data caching
  - Scan HDD only once, then scan your RAM
- Efficient pipelining
  - Avoids the data hitting the HDD by all means

Spark Pillars

Two main abstractions of Spark

- **RDD** – Resilient Distributed Dataset
- **DAG** – Direct Acyclic Graph

RDD

- Simple view
  - RDD is collection of data items split into partitions and stored in memory on worker nodes of the cluster
- Complex view
  - RDD is an interface for data transformation
  - RDD refers to the data stored either in persisted store (HDFS, Cassandra, HBase, etc.) or in cache (memory, memory+disks, disk only, etc.) or in another RDD

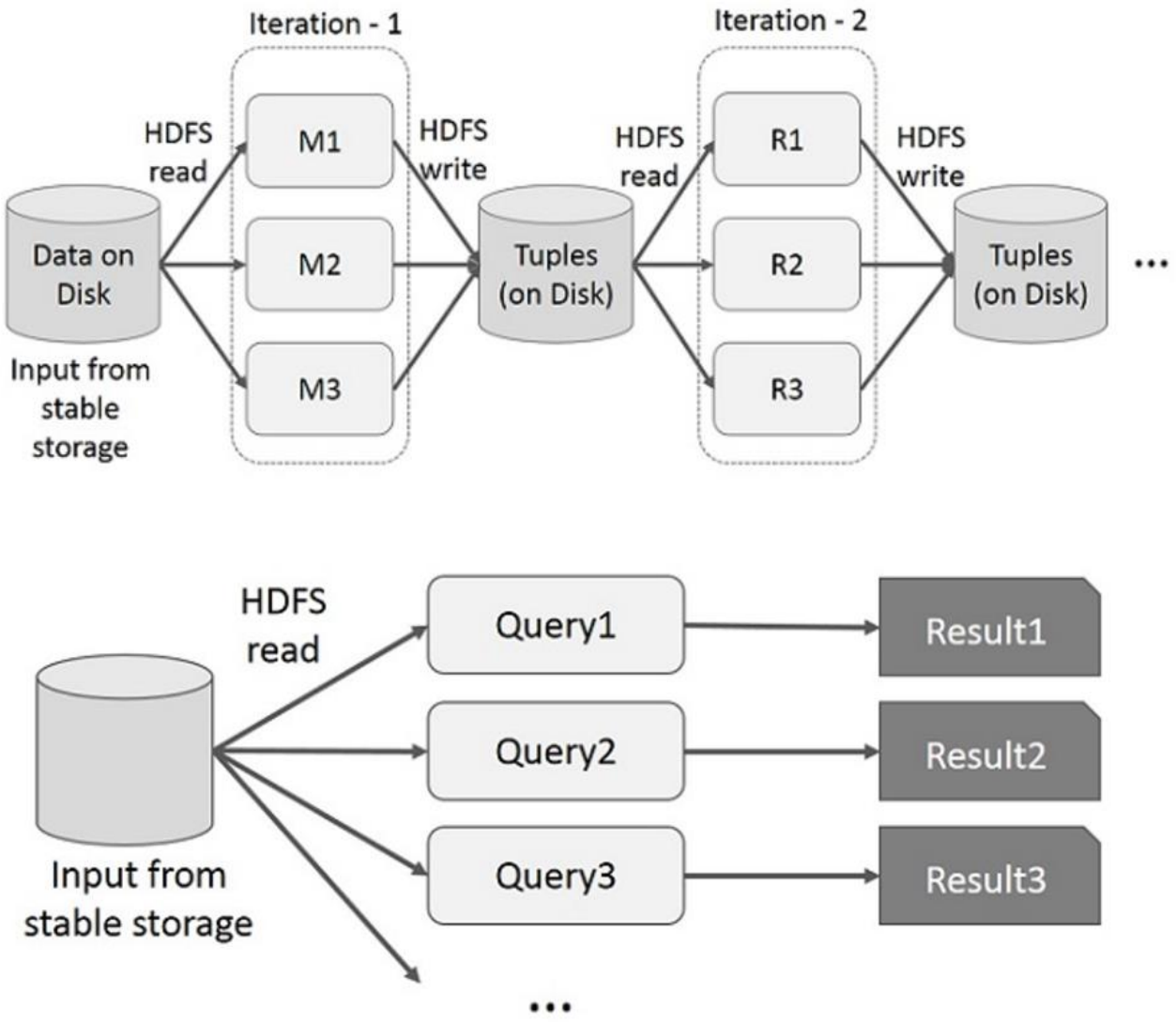
RDD

- Complex view (cont'd)
  - Partitions are recomputed on failure or cache eviction
  - Metadata stored for interface
    - *Partitions* – set of data splits associated with this RDD
    - *Dependencies* – list of parent RDDs involved in computation
    - *Compute* – function to compute partition of the RDD given the parent partitions from the *Dependencies*
    - *Preferred Locations* – where is the best place to put computations on this partition (data locality)
    - *Partitioner* – how the data is split into partitions

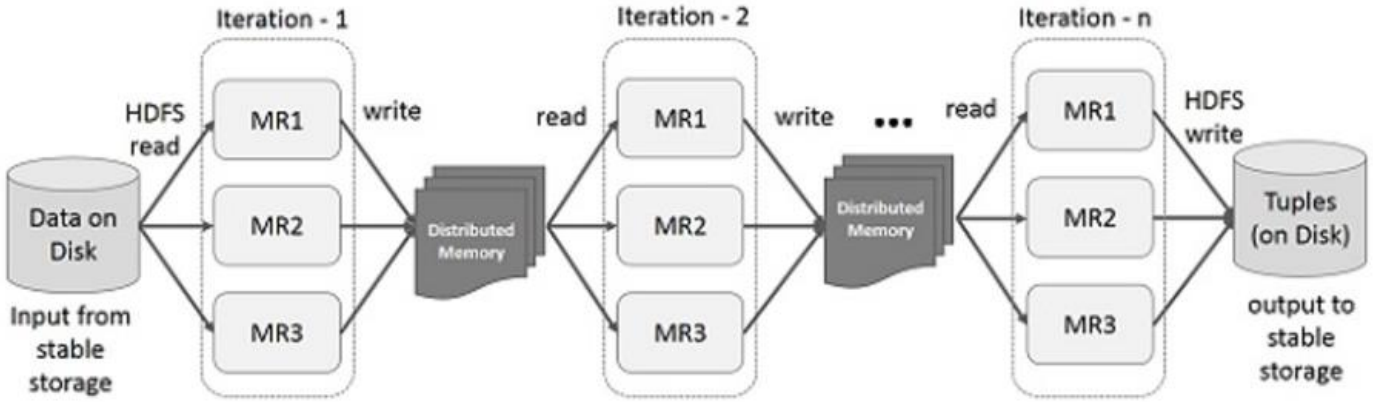
RDD

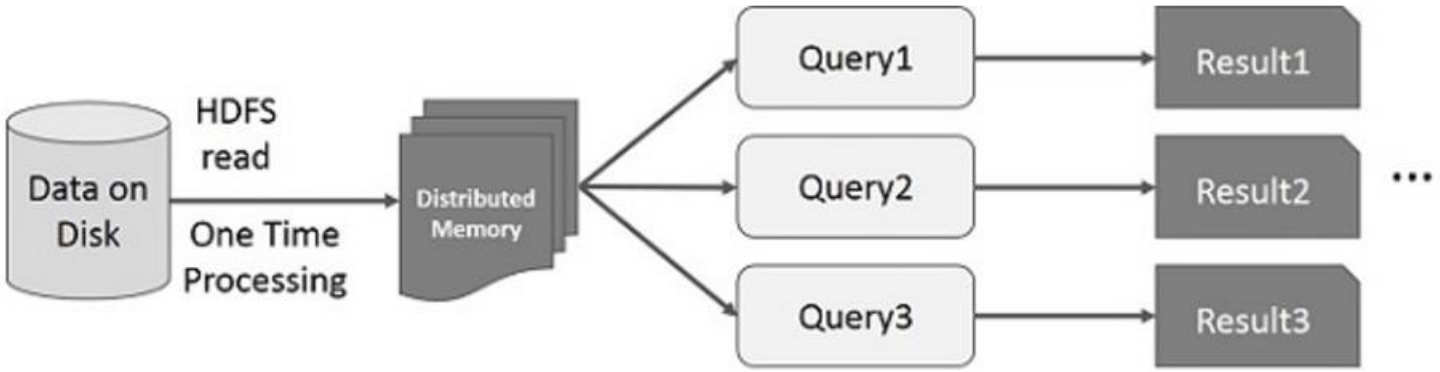
- RDD is the main and only tool for data manipulation in Spark
- Two classes of operations
  - Transformations
  - Actions

Iterative Operations on MapReduce



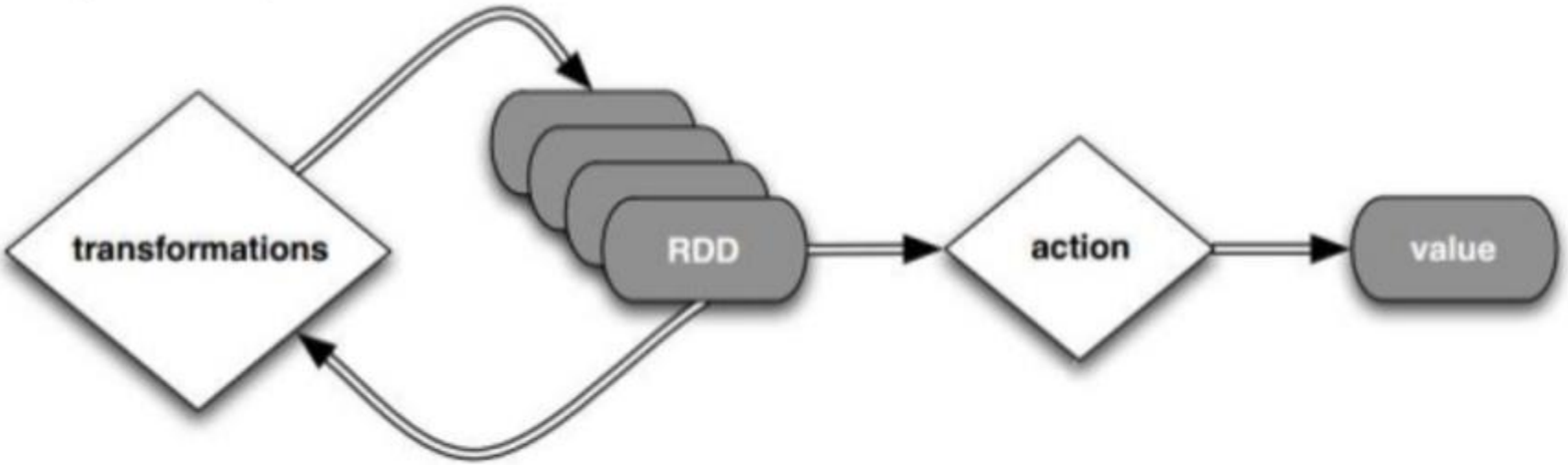
Iterative Operations on Spark RDD





RDD

Lazy computations model



Transformation cause only metadata change

DAG

Direct Acyclic Graph – sequence of computations performed on data

- *Node* – RDD partition
- *Edge* – transformation on top of data
- *Acyclic* – graph cannot return to the older partition
- *Direct* – transformation is an action that transitions data partition state (from A to B)

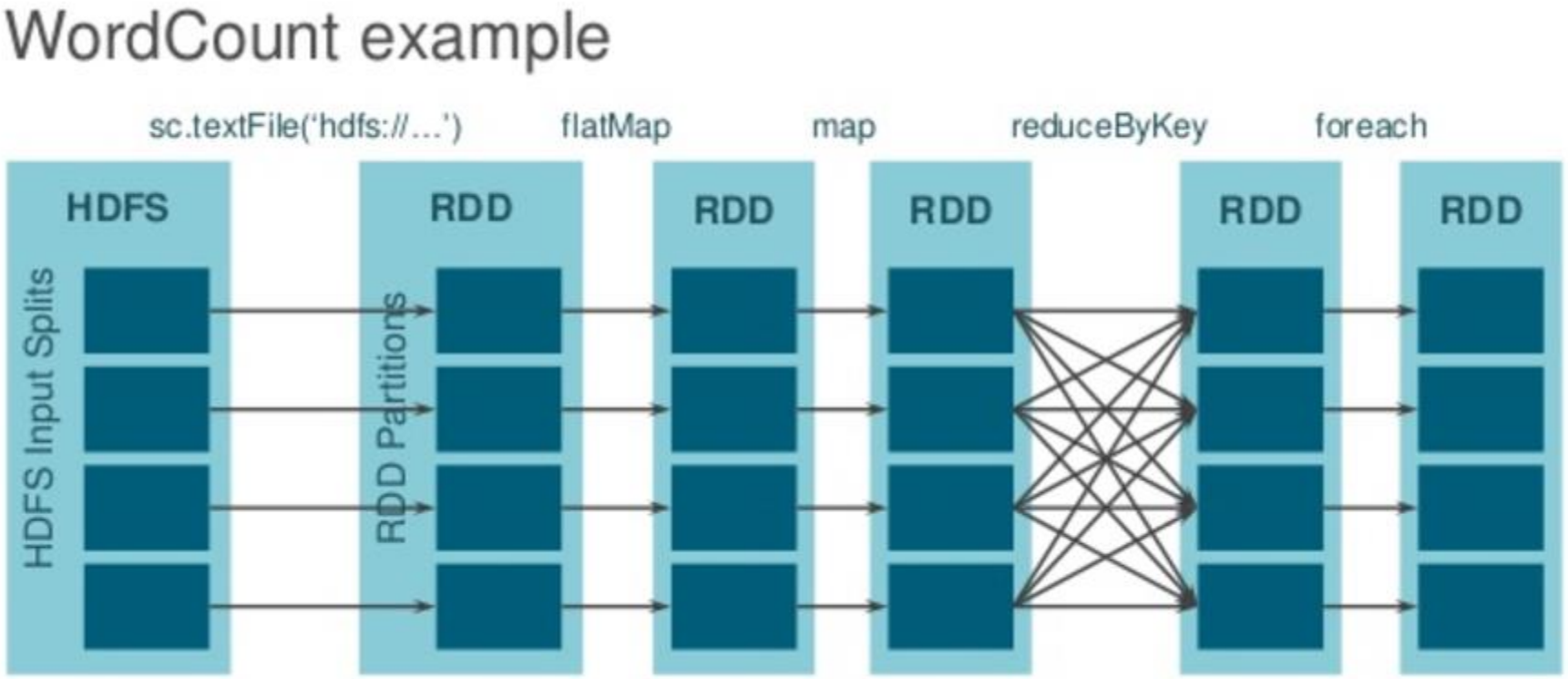
DAG

WordCount example

```
def printfunc (x):
    print 'Word "%s" occurs %d times' % (x[0], x[1])

infile = sc.textFile('hdfs://sparkdemo:8020/sparkdemo/textfiles/README.md', 4)
rdd1 = infile.flatMap(lambda x: x.split())
rdd2 = rdd1.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x+y)
print rdd2.toDebugString()
rdd2.foreach(printfunc)
```

DAG

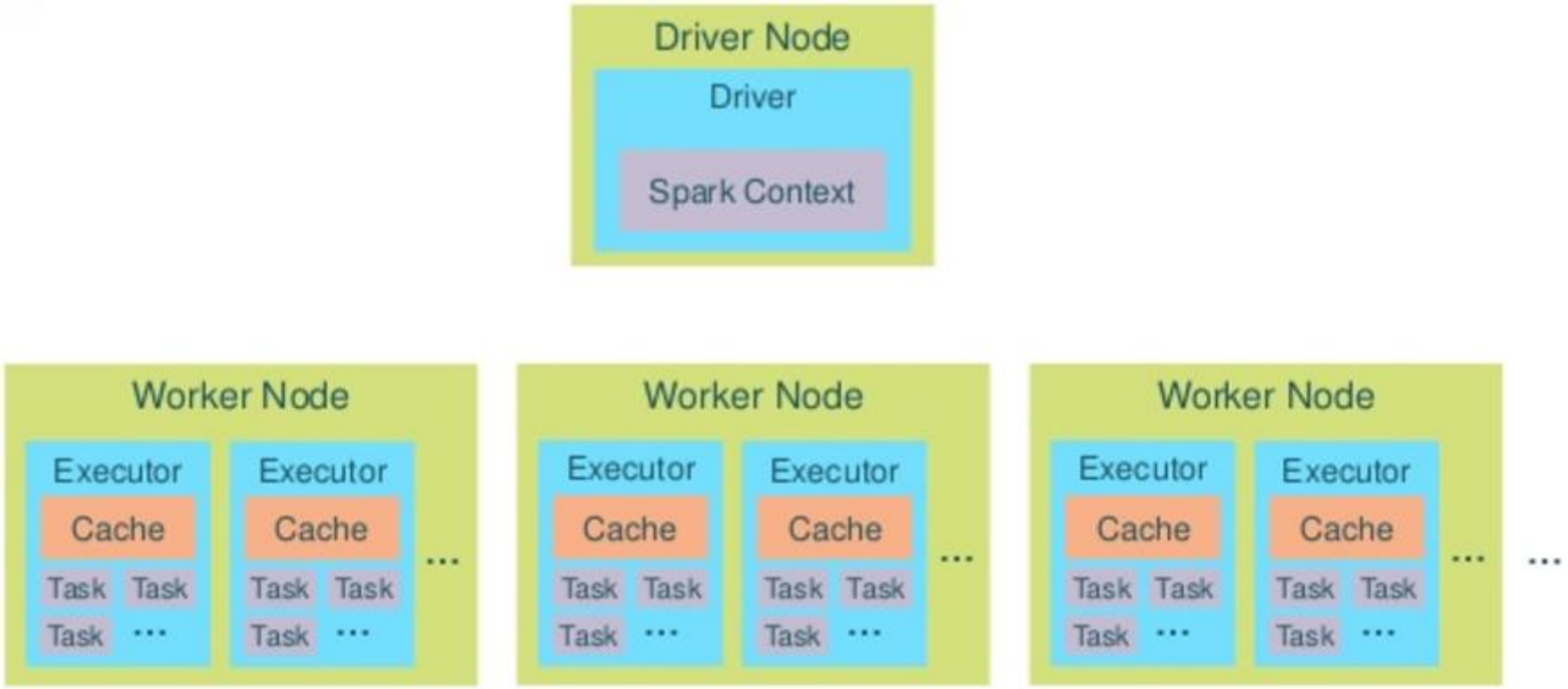

WordCount example
sc.textFile('hdfs://...')
flatMap
map
reduceByKey
foreach
HDFS
RDD
RDD
RDD
RDD
RDD
HDFS Input Splits
RDD Partitions

Outline

Outline

- Spark Motivation
- Spark Pillars
- **Spark Architecture**
- Spark Shuffle
- Spark DataFrames

Spark Cluster



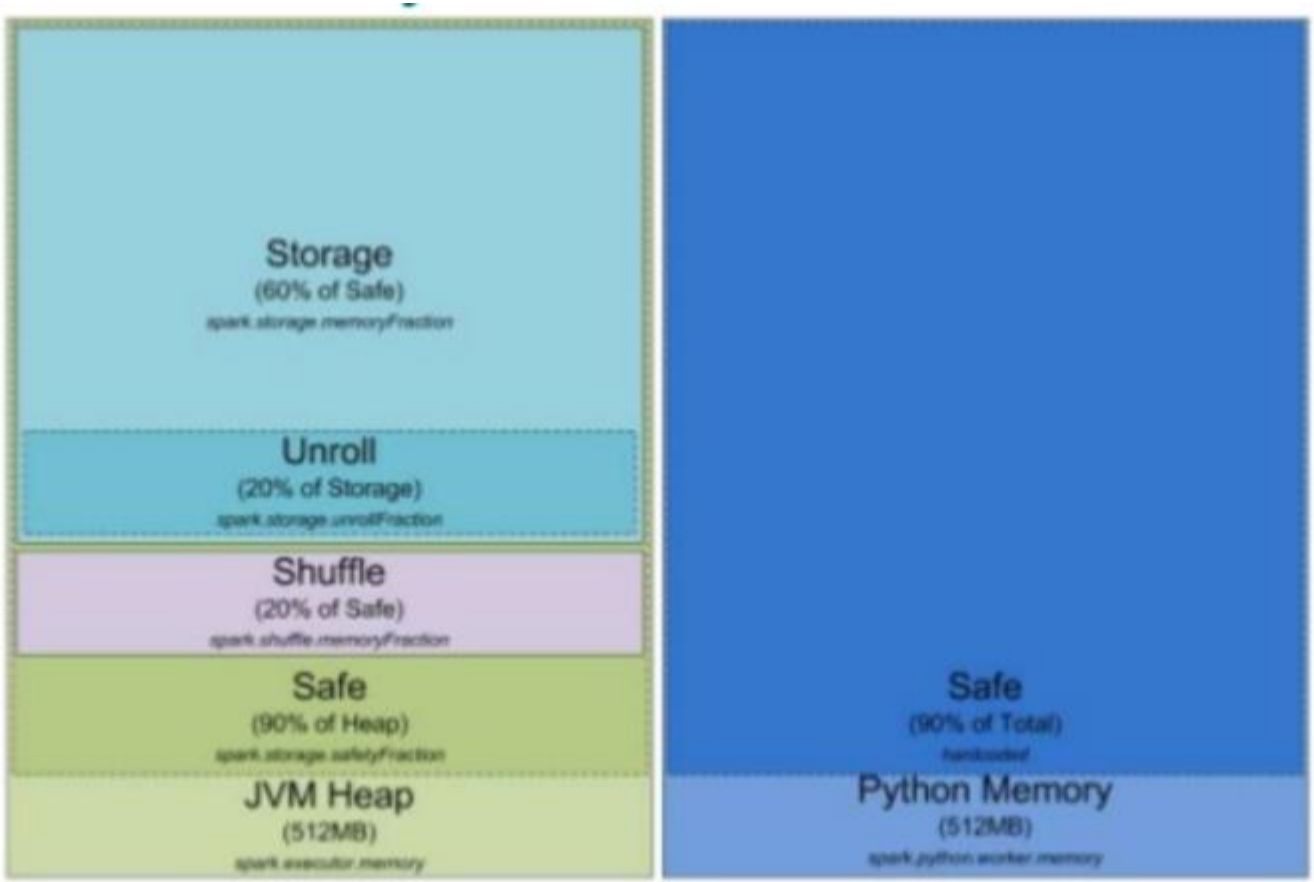
Spark Cluster

- **Driver**
  - Entry point of the Spark Shell (Scala, Python, R)
  - The place where SparkContext is created
  - Translates RDD into the execution graph
  - Splits graph into stages
  - Schedules tasks and controls their execution
  - Stores metadata about all the RDDs and their partitions
  - Brings up Spark WebUI with job information

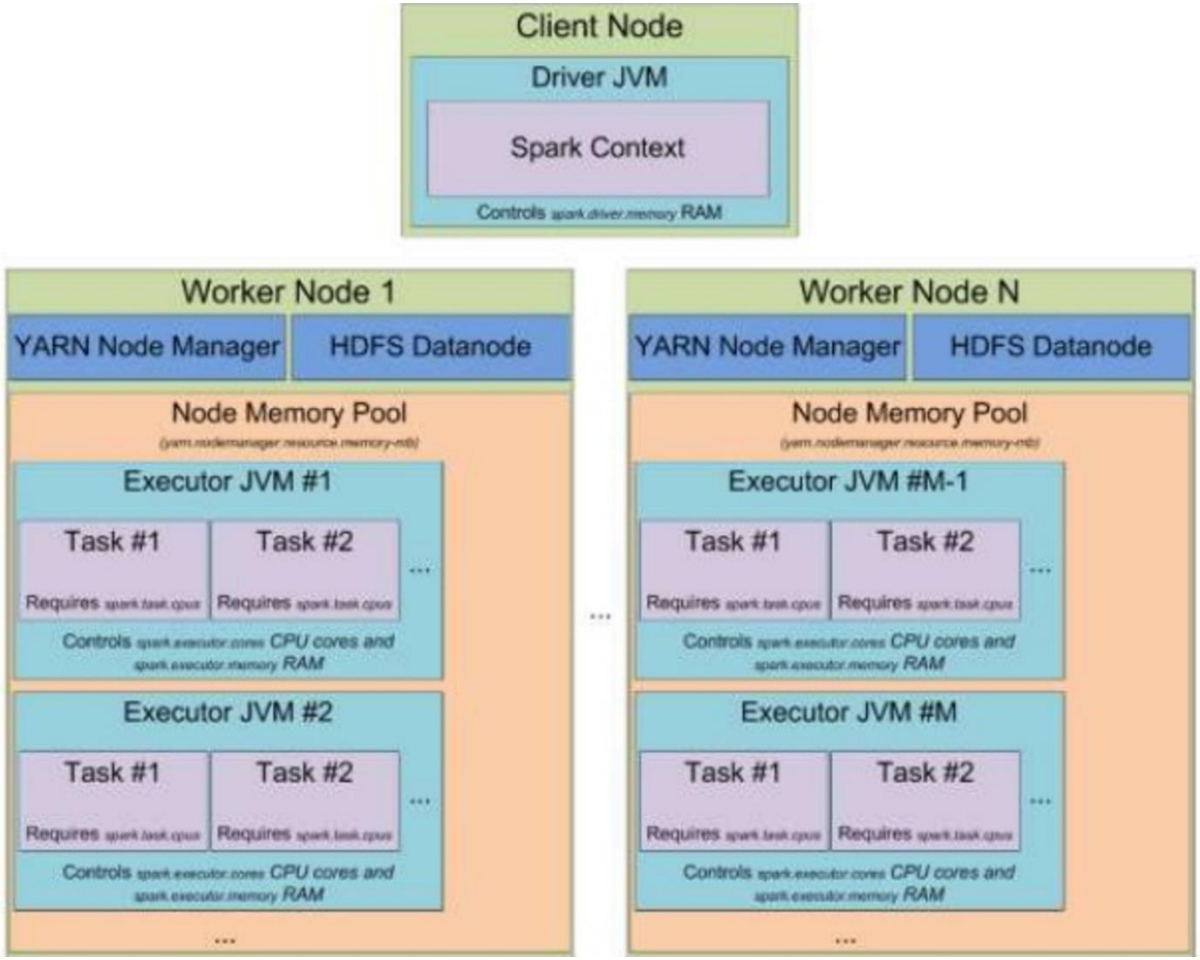
Spark Cluster

- **Executor**
  - Stores the data in cache in JVM heap or on HDDs
  - Reads data from external sources
  - Writes data to external sources
  - Performs all the data processing

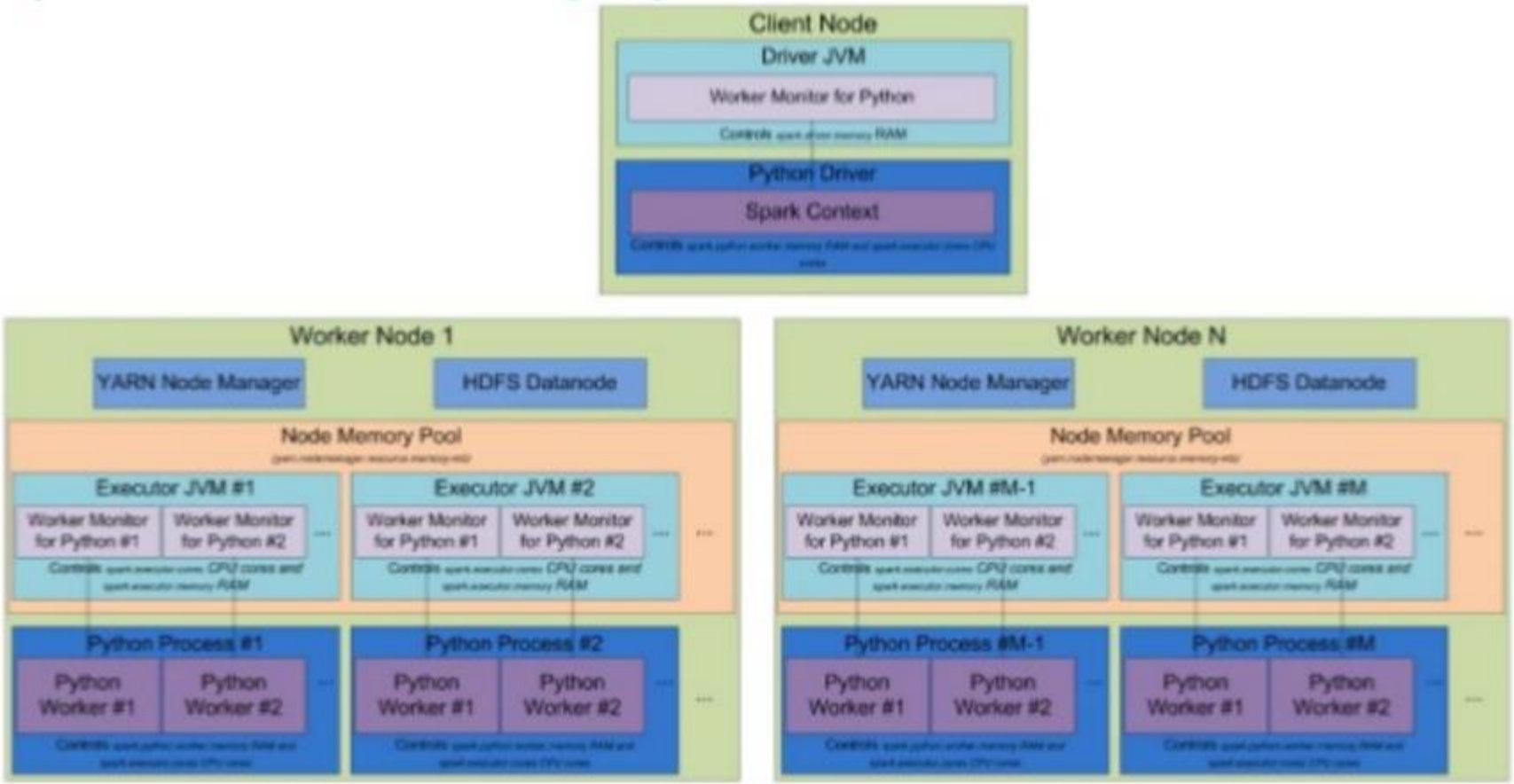
Excutor Memory



Spark Cluster - Detailed



Spark Cluster-PySpark



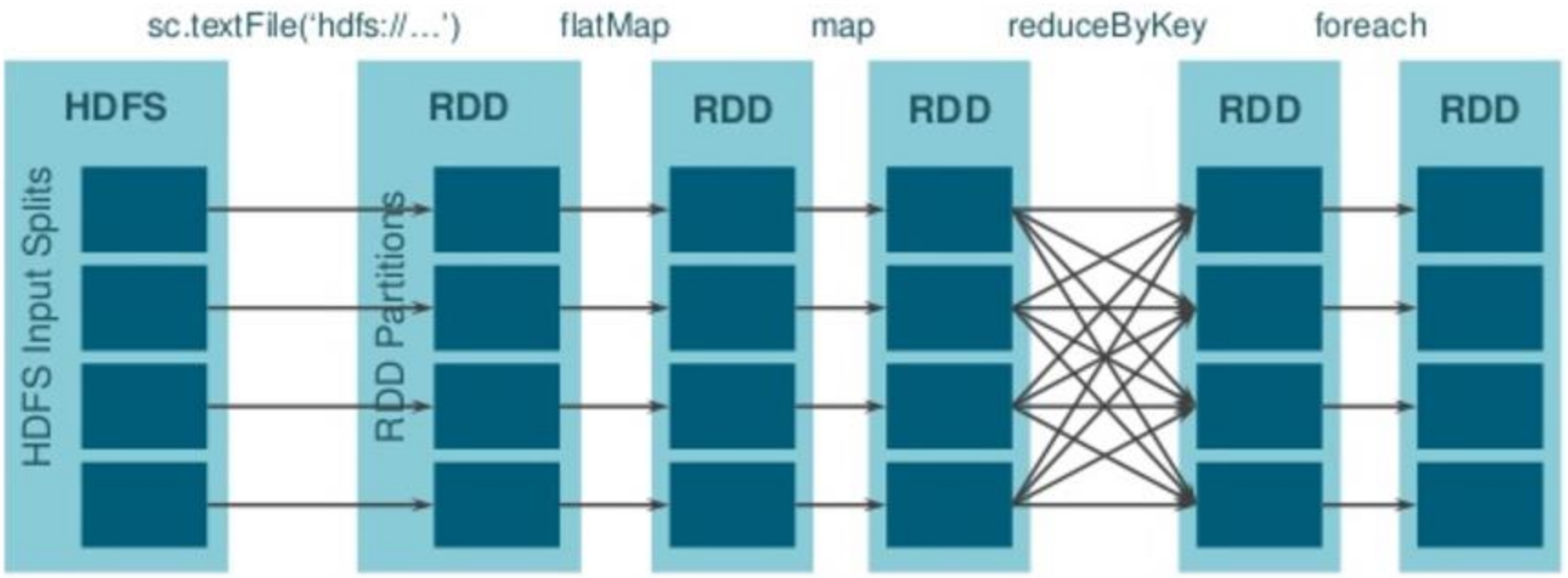
Application Decomposition

- **Application**
  - Single instance of SparkContext that stores some data processing logic and can schedule series of jobs, sequentially or in parallel (SparkContext is thread-safe)
- **Job**
  - Complete set of transformations on RDD that finishes with action or data saving, triggered by the driver application

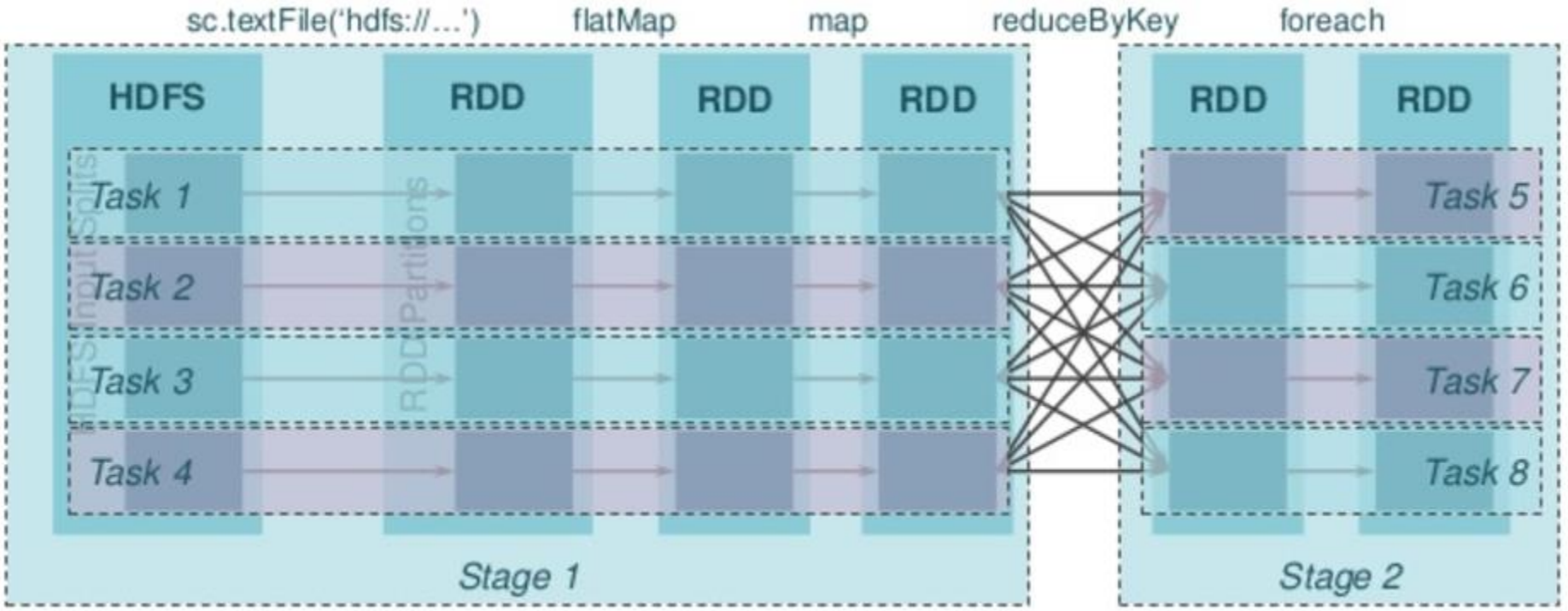
Application Decomposition

- **Stage**
  - Set of transformations that can be pipelined and executed by a single independent worker. Usually it is app the transformations between “read”, “shuffle”, “action”, “save”
- **Task**
  - Execution of the stage on a single data partition. Basic unit of scheduling

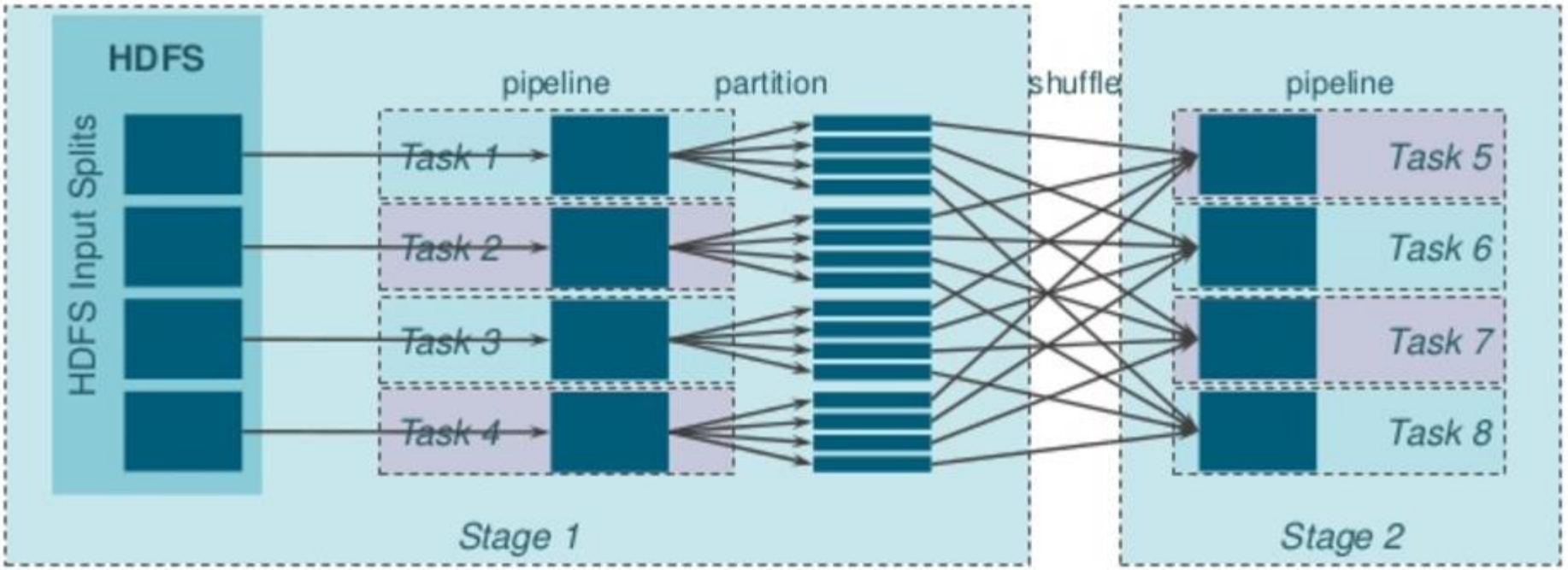
Word Count Example



Word Count Example



Word Count Example



Persistence in Spark

Persistence Level	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, DISK_ONLY_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

Persistence in Spark

- Spark considers memory as a cache with LRU eviction rules
- If “Disk” is involved, data is evicted to disks

```
rdd = sc.parallelize(xrange(1000))
rdd.cache().count()
rdd.persist(StorageLevel.MEMORY_AND_DISK_SER).count()
rdd.unpersist()
```

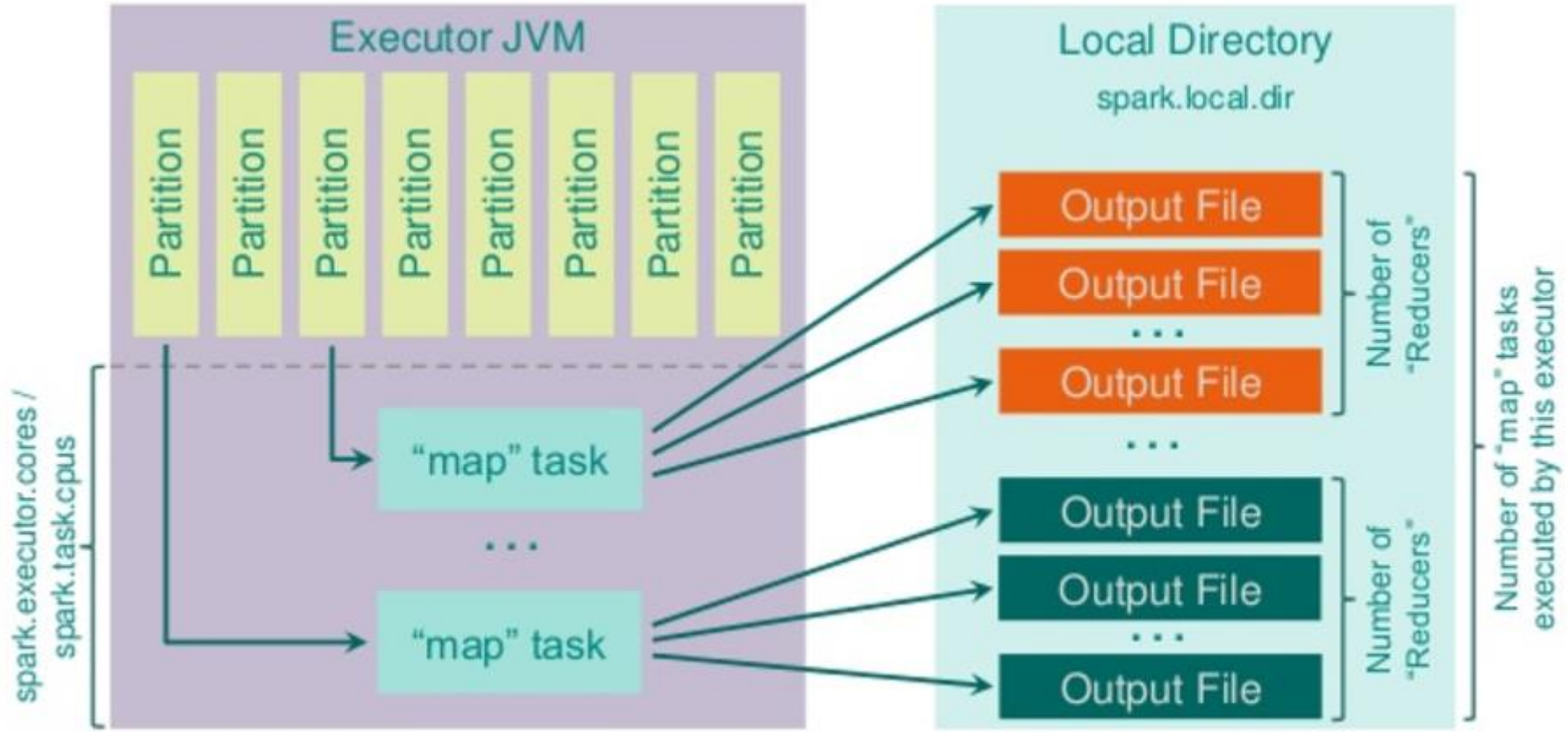
Outline

- Spark Motivation
- Spark Pillars
- Spark Architecture
- **Spark Shuffle**
- Spark DataFrame

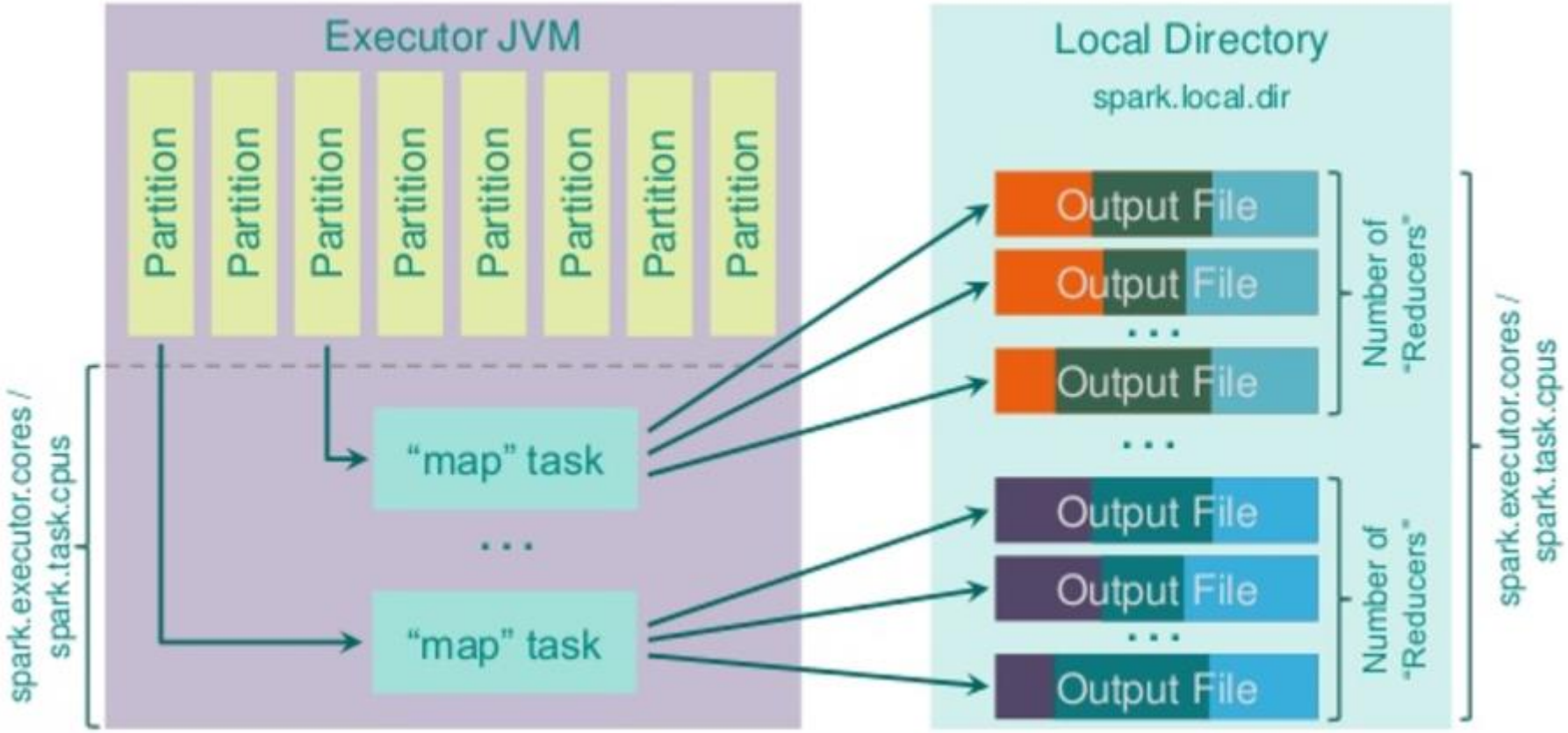
Shuffles in Spark

- *Hash Shuffle* – default prior to 1.2.0
- *Sort Shuffle* – default now
- *Tungsten Sort* – new optimized one!

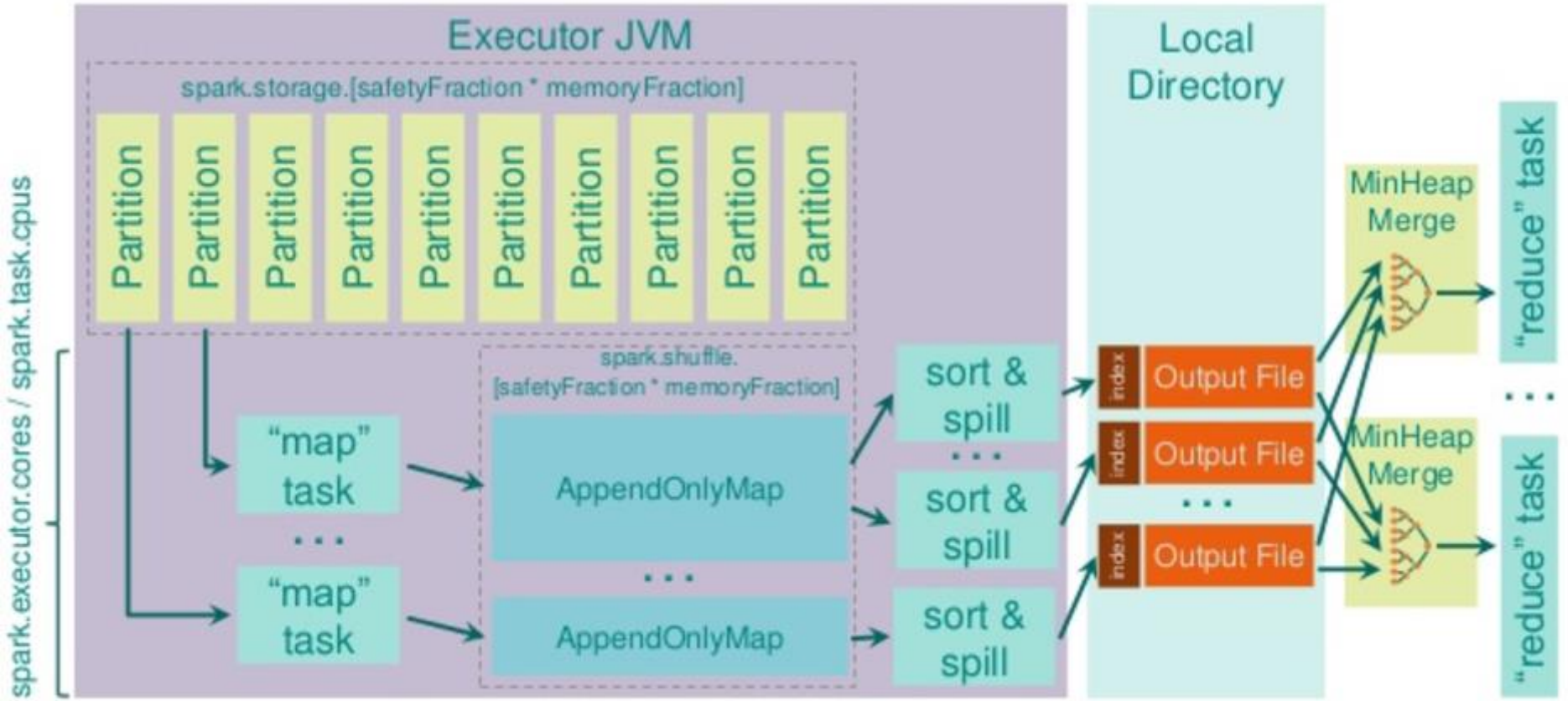
Hash Shuffles



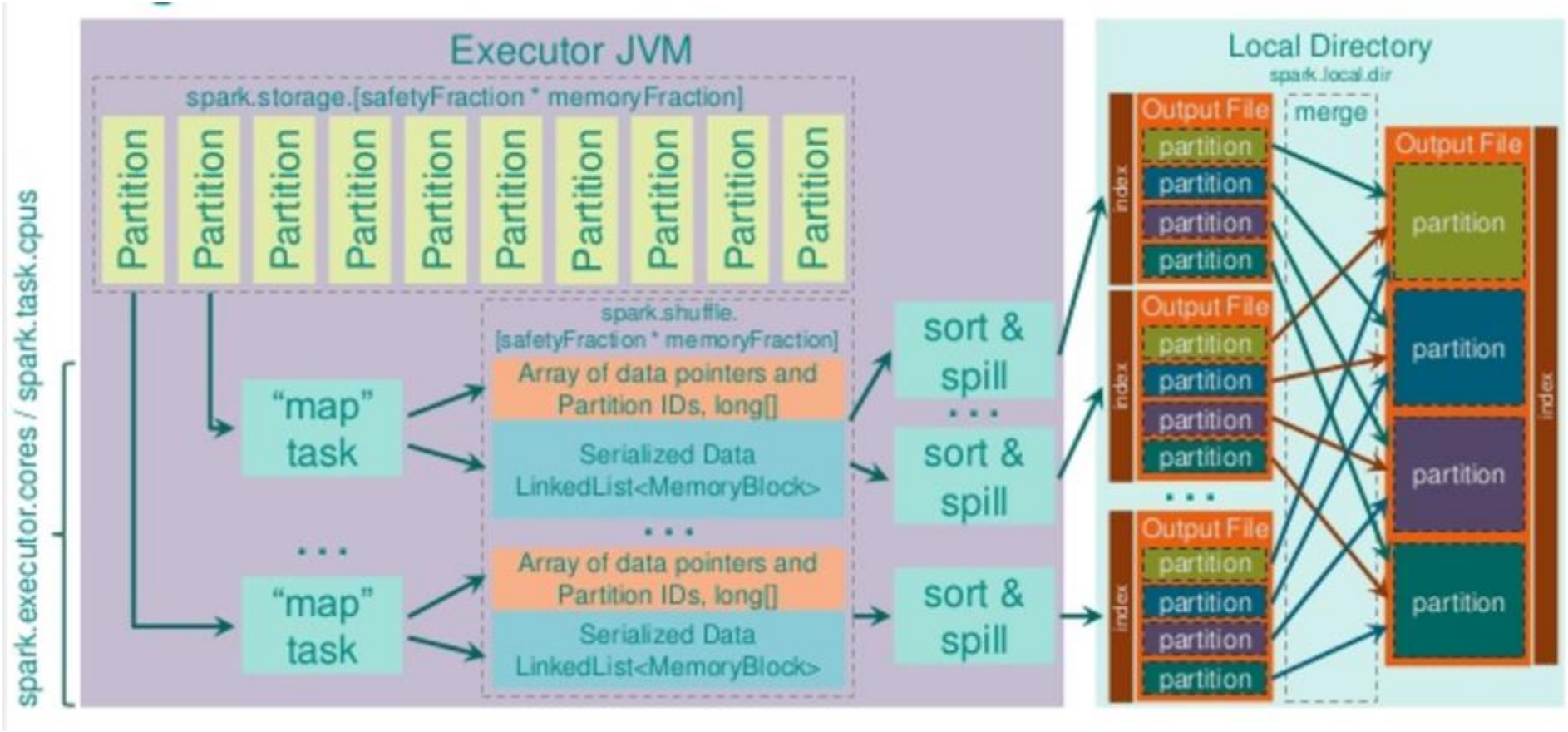
Hash Shuffle with Consolidation



Sort Shuffle



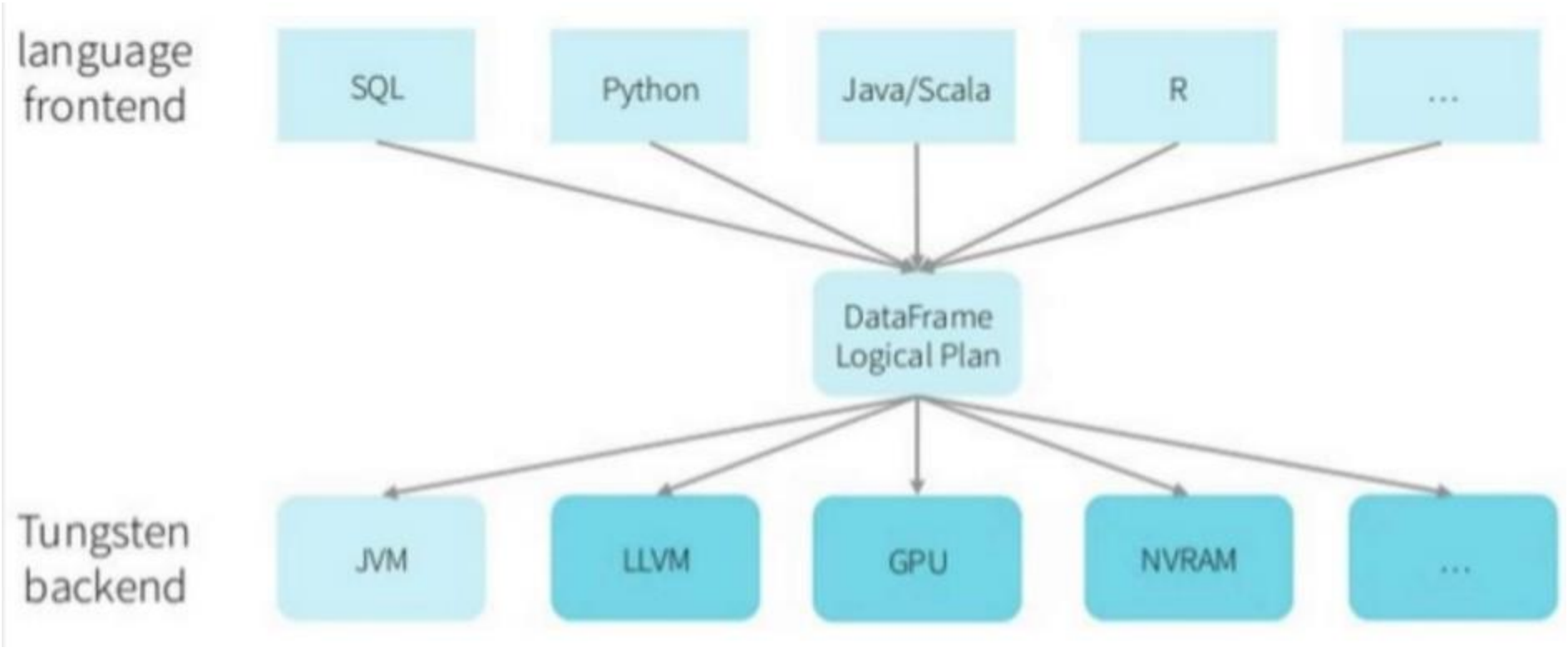
Tungsten Sort Shuffle



Outline

- Spark Motivation
- Spark Pillars
- Spark Architecture
- Spark Shuffle
- **Spark DataFrame**

Data Frame Idea



Data Frame Implementation

- **Interface**
  - DataFrame is an RDD with schema – field names, field data types and statistics
  - Unified transformation interface in all languages, all the transformations are passed to JVM
  - Can be accessed as RDD, in this case transformed to the RDD of Row objects

Data Frame Implementation

- **Internals**
  - Internally it is the same RDD
  - Data is stored in row-columnar format, row chunk size is set by *spark.sql.inMemoryColumnarStorage.batchSize*
  - Each column in each partition stores min-max values for partition pruning
  - Allows better compression ratio than standard RDD
  - Delivers faster performance for small subsets of columns

Example: Word Count Scala Code

```
val textFile = sc.textFile("/usr/local/spark/README.md")
val wordCounts = textFile.flatMap(line => line.split(" ")).  ¥
map(word => (word, 1)).reduceByKey((a, b) => a + b)
wordCounts.collect()
wordCounts.saveAsTextFile("/home/paikic/Spark/WordCountTest1")
```

Apache Saprk

Question?