Spark Workshop

internals, architecture & coding

Anton Kirillov

Ooyala, Mar 2016

Roadmap

- RDDs
  - Definition
  - Operations
- Execution workflow
  - DAG
  - Stages and tasks
  - Shuffle
- Architecture
  - Components
  - Memory model
- Coding
  - spark-shell
  - building and submitting Spark applications to YARN

Meet Spark

- Generalized framework for distributed data processing (batch, graph, ML)
- Scala collections functional API for manipulating data at scale
- In-memory data caching and reuse across computations
- Applies set of coarse-grained transformations over partitioned data
- Failure recovery relies on lineage to recompute failed tasks
- Supports majority of input formats and integrates with Mesos / YARN

Spark makes data engineers happy

Backup/restore of Cassandra tables in Parquet

```
def backup(sc: SparkContext, sqlContext: SQLContext, config: Config) {
  sc.cassandraTable(config.keyspace, config.table).map(_.toEvent).toDF()
  .write.parquet(s"${config.targetDir}/${config.table}.parquet")
}

def restore(sc: SparkContext, sqlContext: SQLContext, config: Config) {
  sqlContext.read.parquet(s"${config.targetDir}/${config.table}.parquet")
  .map(_.toEvent).saveToCassandra(config.keyspace, config.table)
}
```

Query different data sources to identify discrepancies

```
sqlContext.sql {
  """
    SELECT count()
    FROM cassandra_event_rollups
    JOIN mongo_event_rollups
    ON cassandra_event_rollups.uuid = cassandra_event_rollups.uuid
    WHERE cassandra_event_rollups.value != cassandra_event_rollups.value
  """.stripMargin
}
```

Core Concepts

RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure
- Provides API for
  - manipulating the collection of elements (transformations and materialization)
  - persisting intermediate results in memory for later reuse
  - controlling partitioning to optimize data placement
- Can be created through deterministic operation
  - from storage (distributed file system, database, plain file)
  - from another RDD
- Stores information about parent RDDs
  - for execution optimization and operations pipelining
  - to recompute the data in case of failure

RDD: a developer's view

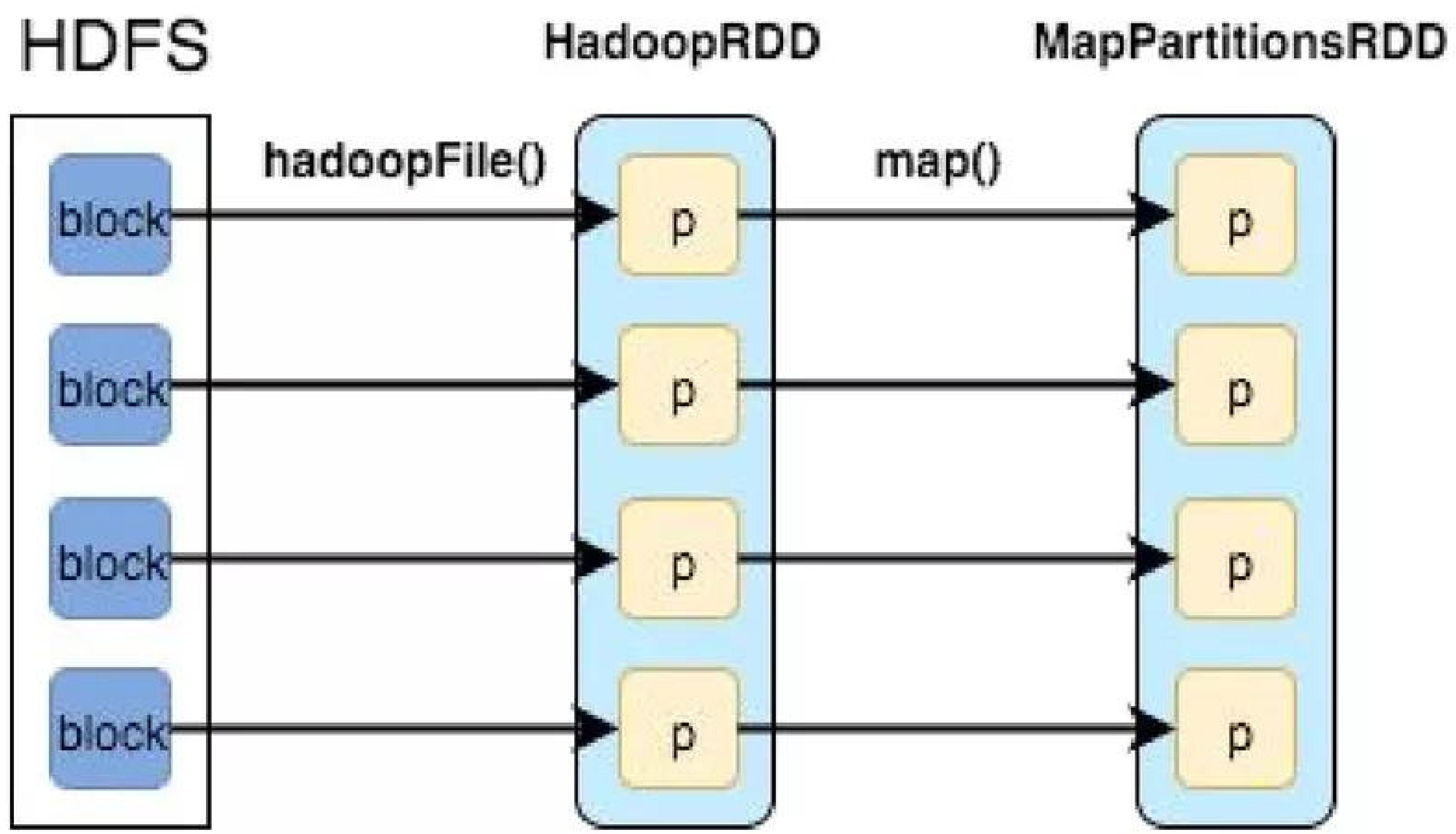
- Distributed immutable data + lazily evaluated operations
  - partitioned data + iterator
  - transformations & actions
- An interface defining 5 main properties

```
a list of partitions (e.g. splits in Hadoop)
def getPartitions: Array[Partition]

a list of dependencies on other RDDs
def getDependencies: Seq[Dependency[_]]

a function for computing each split
def compute(split: Partition, context: TaskContext): Iterator[T]

(optional) a list of preferred locations to compute each split on
def getPreferredLocations(split: Partition): Seq[String] = Nil

(optional) a partitioner for key-value RDDs
val partitioner: Option[Partitioner] = None
```

lineage

execution optimization

RDDs Example



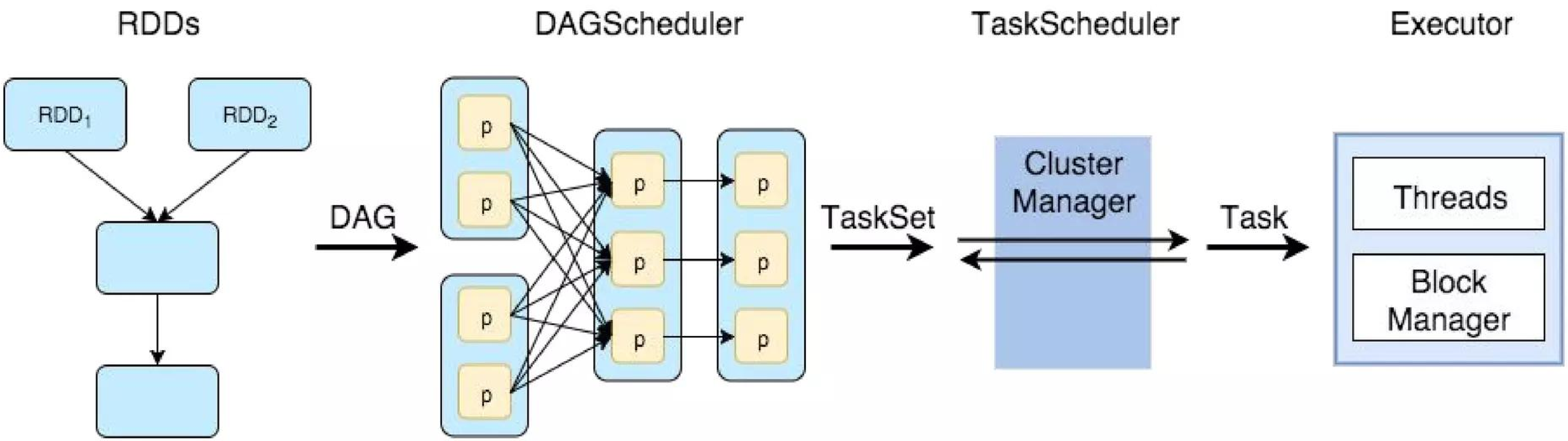
sparkContext.textFile("hdfs://...")

- HadoopRDD
  - getPartitions = HDFS blocks
  - getDependencies = None
  - compute = load block in memory
  - getPrefferedLocations = HDFS block locations
  - partitioner = None
- MapPartitionsRDD
  - getPartitions = same as parent
  - getDependencies = parent RDD
  - compute = compute parent and apply map()
  - getPrefferedLocations = same as parent
  - partitioner = None

RDD Operations

- Transformations
  - apply user function to every element in a partition (or to the whole partition)
  - apply aggregation function to the whole dataset (groupBy, sortBy)
  - introduce dependencies between RDDs to form DAG
  - provide functionality for repartitioning (repartition, partitionBy)
- Actions
  - trigger job execution
  - used to materialize computation results
- Extra: persistence
  - explicitly store RDDs in memory, on disk or off-heap (cache, persist)
  - checkpointing for truncating RDD lineage

Execution workflow



rdd1.**join**(rdd2)
.**groupBy**(...)
.**filter**(...)

splits graph into *stages* of tasks

submits each stage as ready

launches tasks via cluster manager

retries failed or struggling tasks

executes tasks

stores and serves blocks

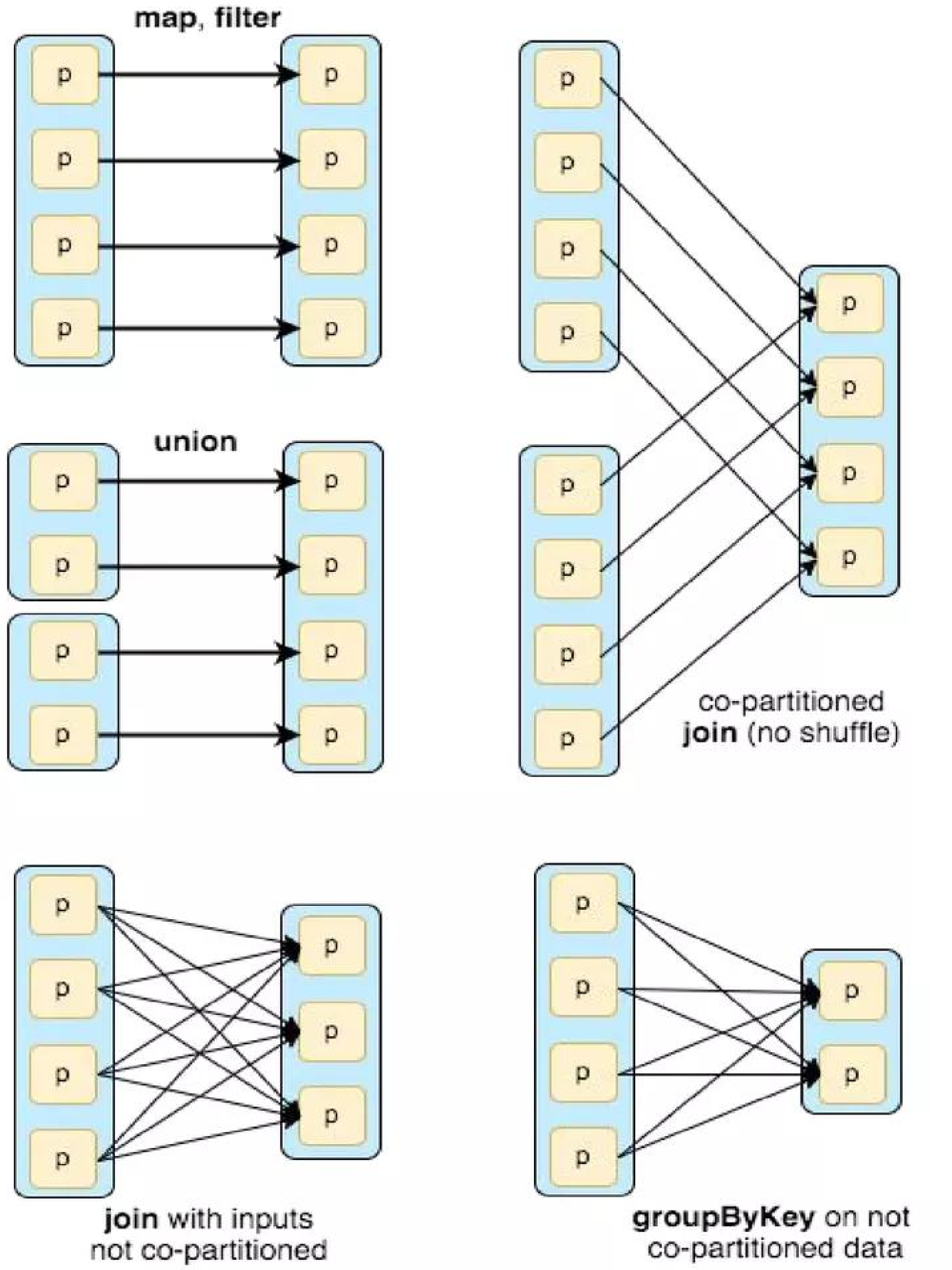
Code sample: joining aggregated and raw data

```
//aggregate events after specific date for given campaign
val events =  sc.cassandraTable("demo", "event")
                         .map(_.toEvent)
                         .filter(event => event.campaignId == campaignId && event.time.isAfter(watermark))
                         .keyBy(_.`type`)
                         .reduceByKey(_ + _)
                         .cache()

//aggregate campaigns by type
val campaigns = sc.cassandraTable("demo", "campaign")
                              .map(_.toCampaign)
                              .filter(campaign => campaign.id == campaignId && campaign.time.isBefore(watermark))
                              .keyBy(_.eventType)
                              .reduceByKey(_ + _)
                              .cache()

//joined rollups and raw events
val joinedTotals = campaigns.join(events)
                                           .map { case (key, (campaign, event)) => CampaignTotals(campaign, event) }
                                           .collect()
//count totals separately
val eventTotals = events.map{ case (t, e) => s"$t -> ${e.value}" }.collect()
val campaignTotals = campaigns.map{ case (t, e) => s"$t -> ${e.value}" }.collect()
```

DAG

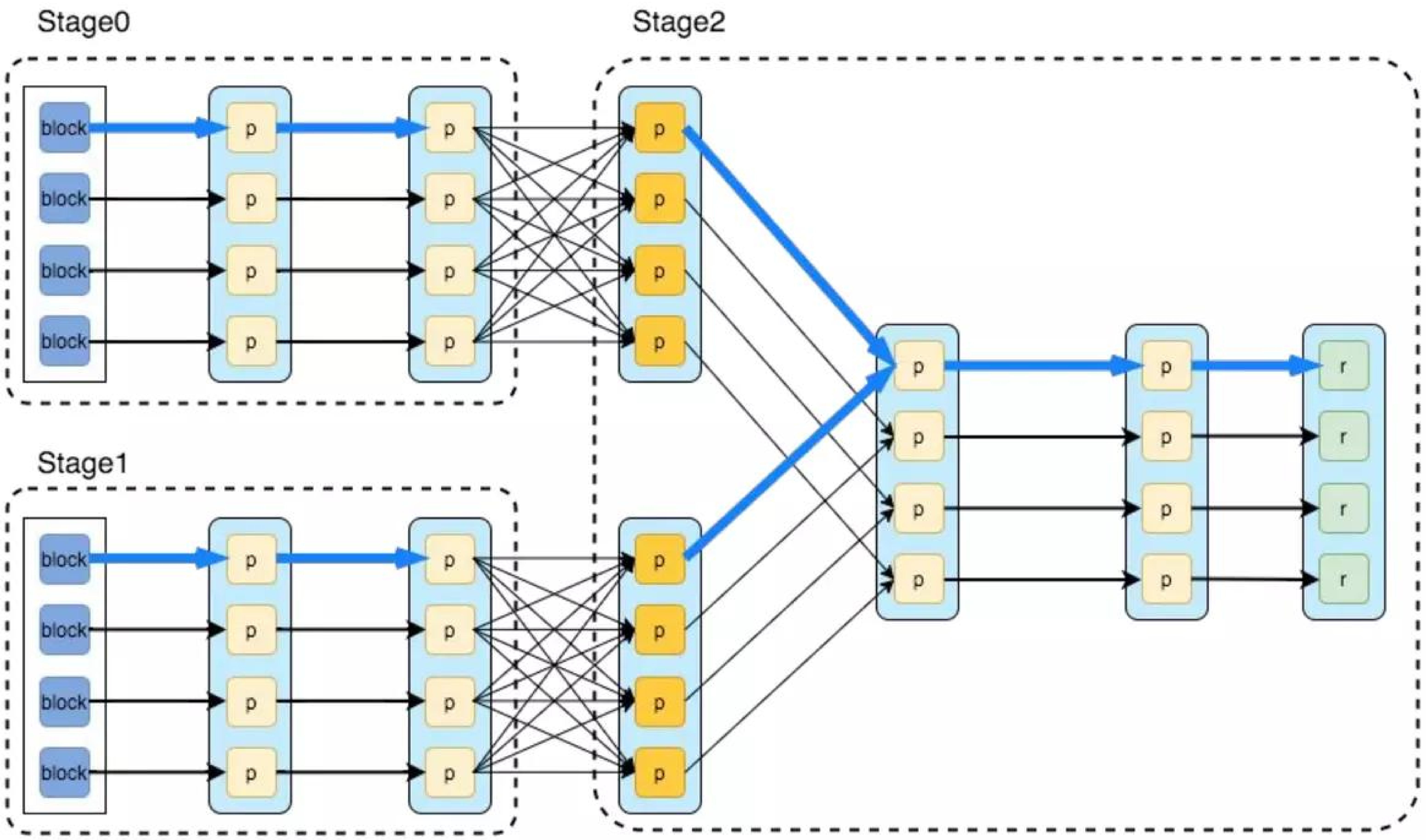
Storage
load
block
block
block
block
RDD1
p
transformations
RDDn
shuffle
RDDm
co-partitioned
join (no shuffle)
Storage
RDD1
RDDi
RDDj
RDDy
transformations
RDDz
action
Results
r
Driver
final result

Dependency types



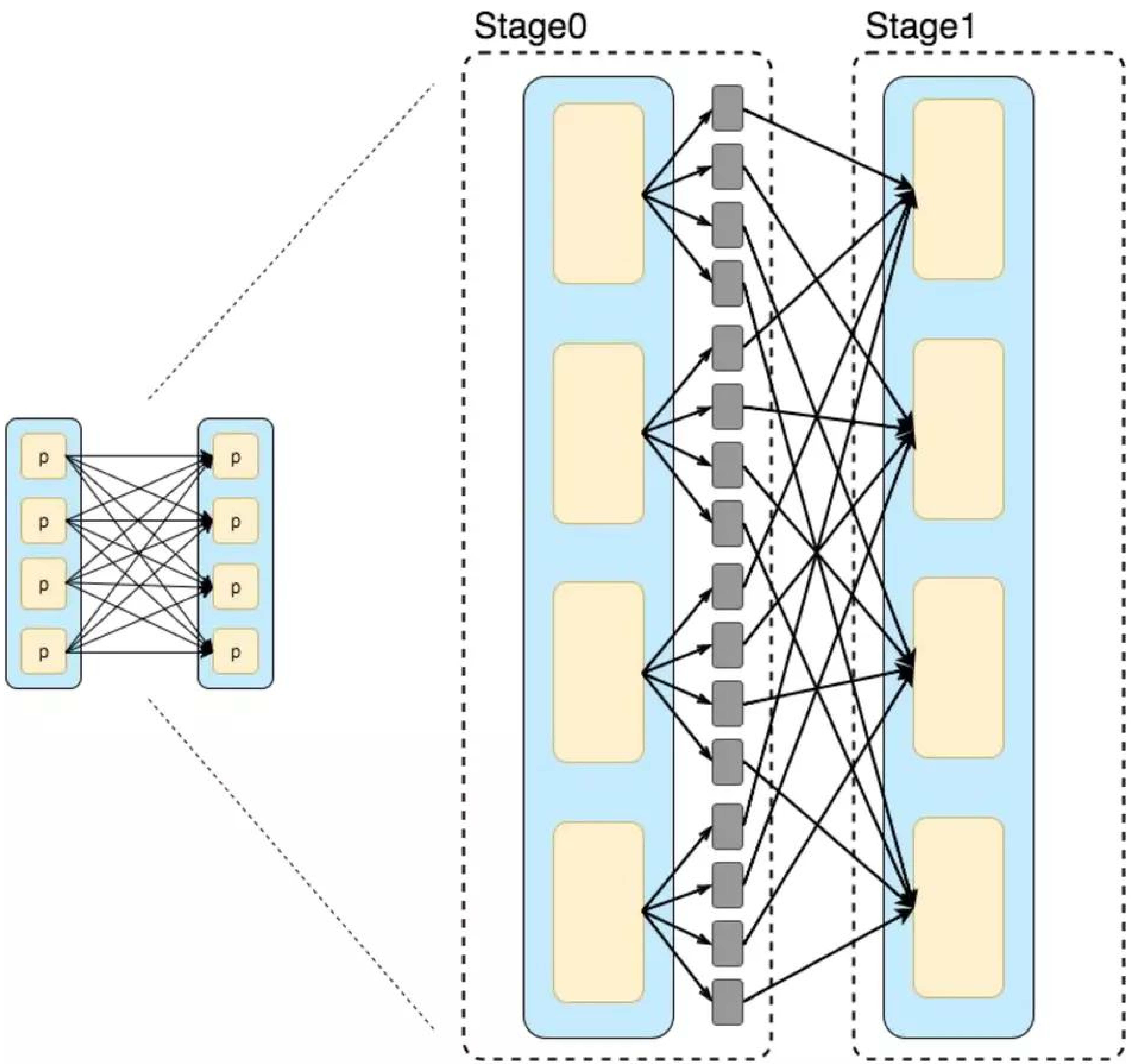
- Narrow (pipelineable)
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - failure recovery is more efficient as only lost parent partitions need to be recomputed
- Wide (shuffle)
  - multiple child partitions may depend on one parent partition
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - if some partition is lost from all the ancestors a complete recomputation is needed

Stages and Tasks



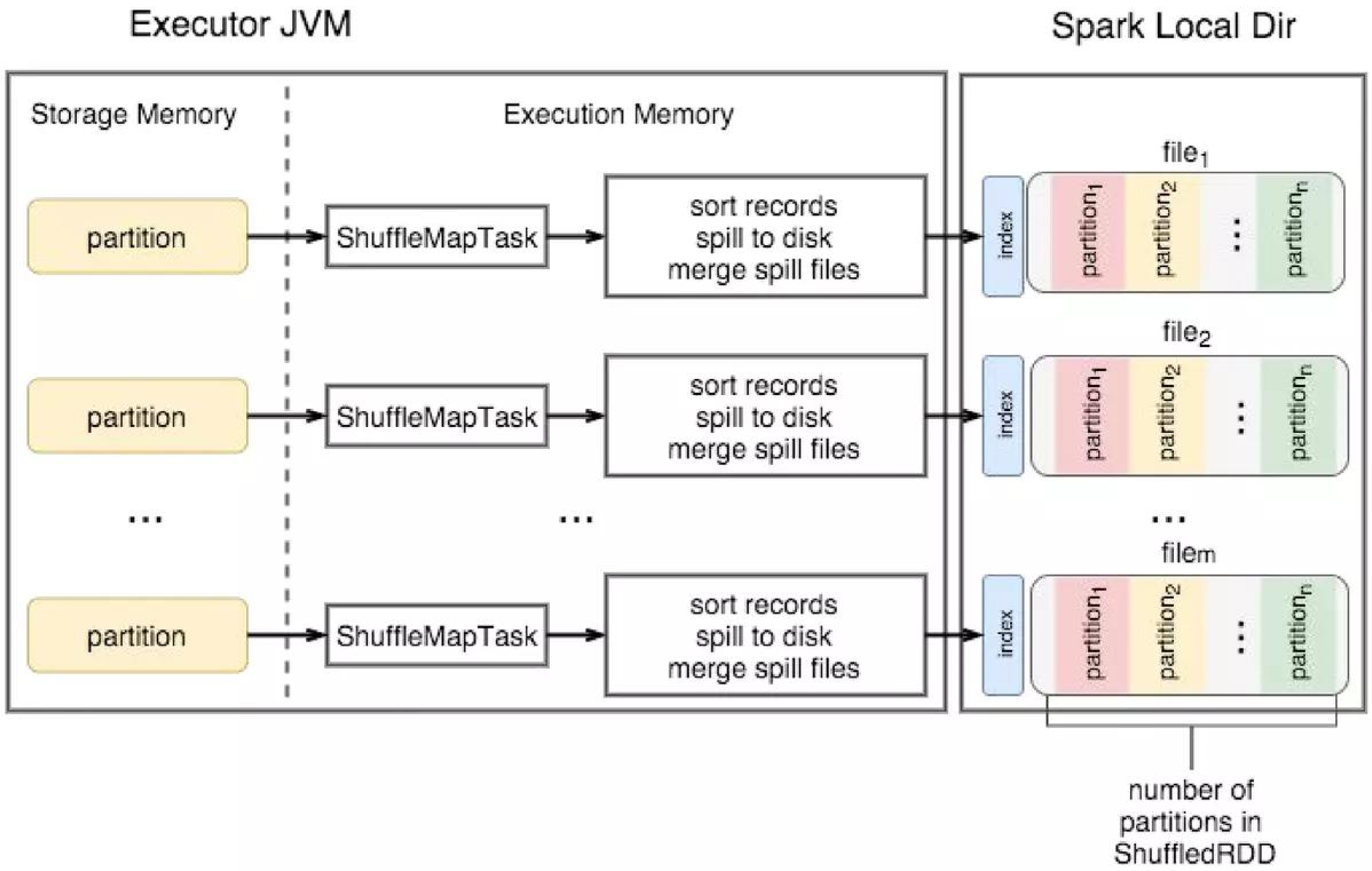
- Stages breakdown strategy
  - check backwards from final RDD
  - add each “narrow” dependency to the current stage
  - create new stage when there’s a shuffle dependency
- Tasks
  - *ShuffleMapTask* partitions its input for shuffle
  - *ResultTask* sends its output to the driver

Shuffle



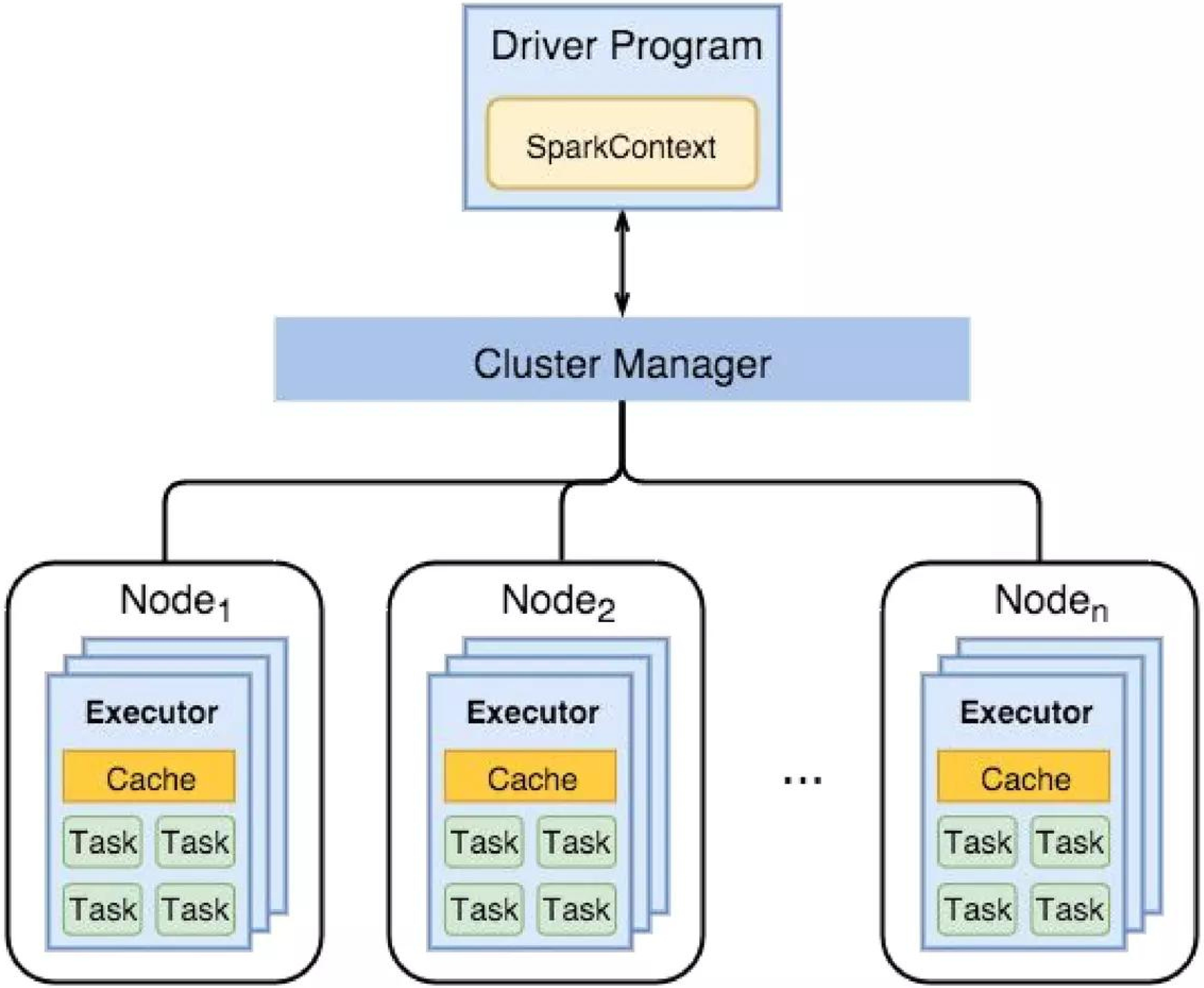
- Shuffle Write
  - redistributes data among partitions and writes files to disk
  - each *hash shuffle* task creates one file per "reduce" task (total = MxR)
  - *sort shuffle* task creates one file with regions assigned to reducer
  - *sort shuffle* uses in-memory sorting with spillover to disk to get final result
- Shuffle Read
  - fetches the files and applies *reduce()* logic
  - if data ordering is needed then it is sorted on "reducer" side for any type of shuffle (SPARK-2926)

Sort Shuffle



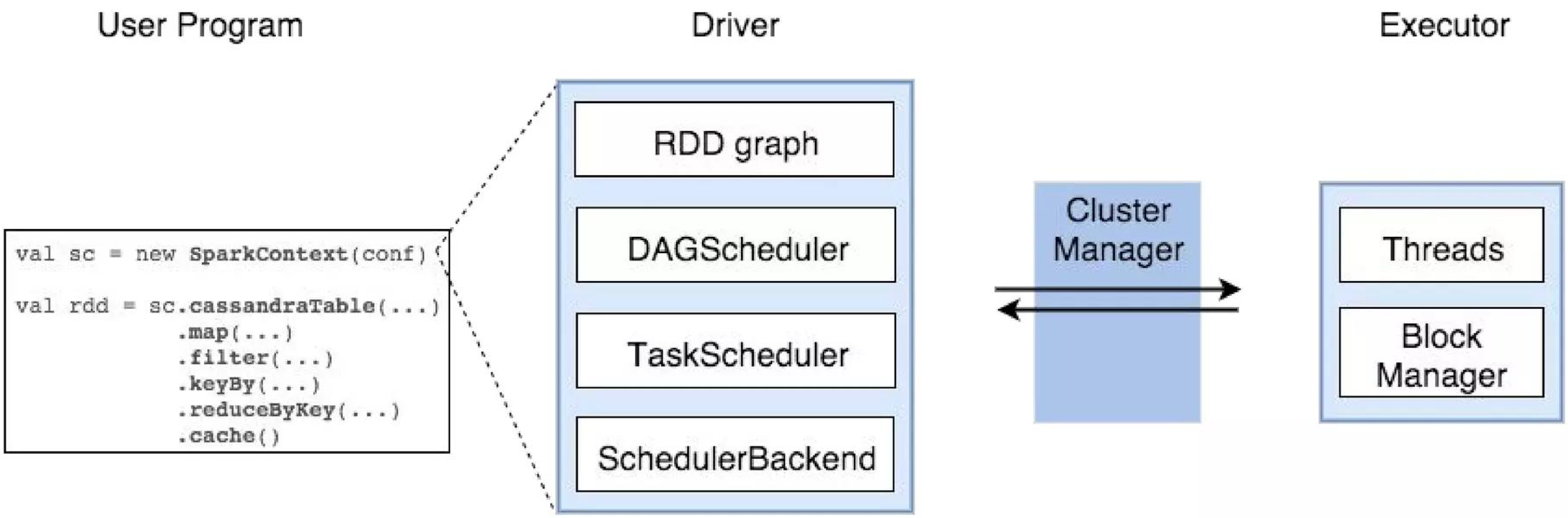
- Incoming records accumulated and sorted in memory according their target partition ids
- Sorted records are written to file or multiple files if spilled and then merged
- *index* file stores offsets of the data blocks in the data file
- Sorting without deserialization is possible under certain conditions (SPARK-7081)

Architecture Recap



- Spark Driver
  - separate process to execute user applications
  - creates *SparkContext* to schedule jobs execution and negotiate with cluster manager
- Executors
  - run tasks scheduled by driver
  - store computation results in memory, on disk or off-heap
  - interact with storage systems
- Cluster Manager
  - Mesos
  - YARN
  - Spark Standalone

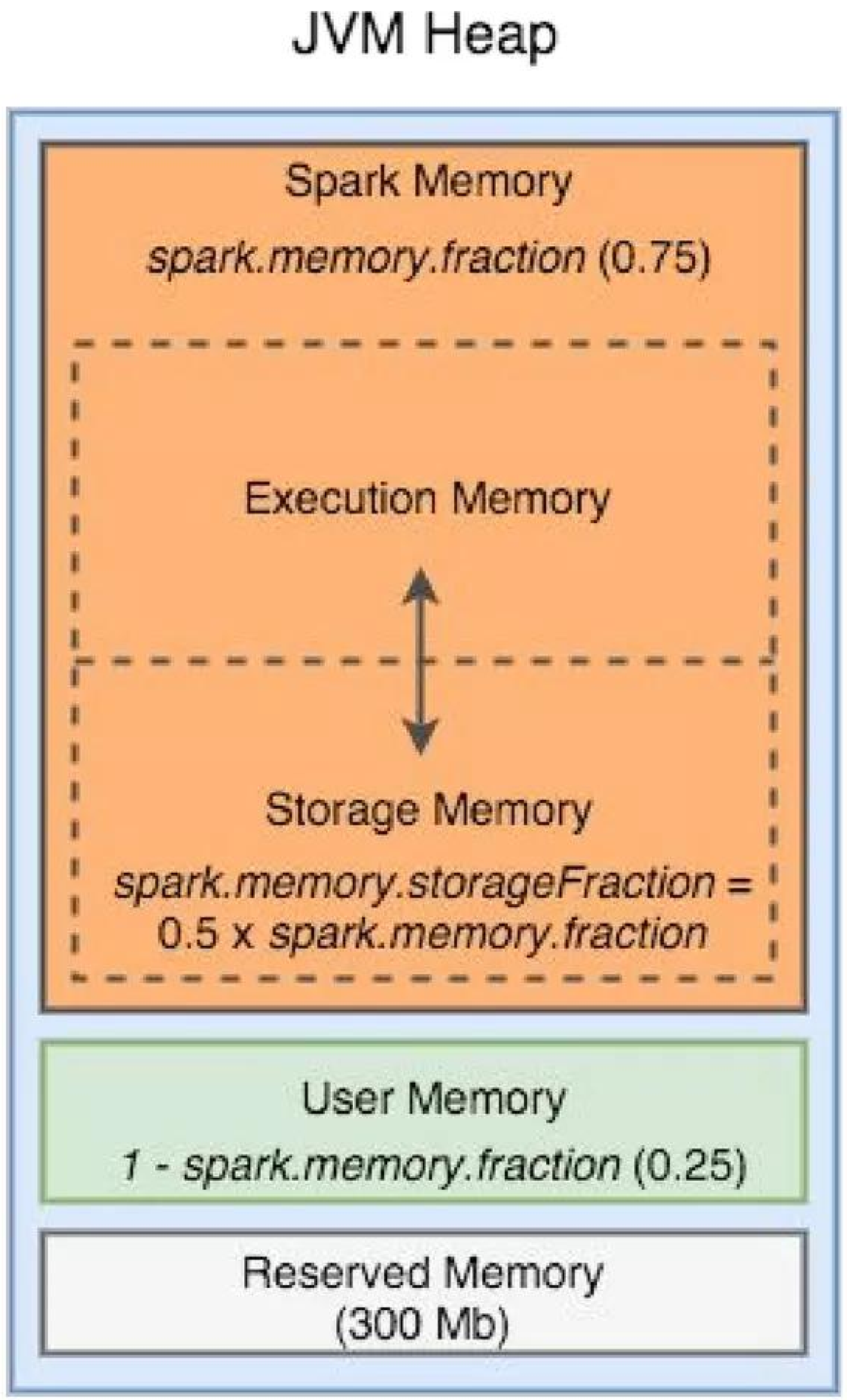
Spark Components



Spark Components

- SparkContext
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster
- DAGScheduler
  - computes a DAG of stages for each job and submits them to TaskScheduler
  - determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs
- TaskScheduler
  - responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers
- SchedulerBackend
  - backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)
- BlockManager
  - provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

Memory Management in Spark 1.6



- Execution Memory
  - storage for data needed during tasks execution
  - shuffle-related data
- Storage Memory
  - storage of cached RDDs and broadcast variables
  - possible to borrow from execution memory (spill otherwise)
  - safeguard value is 0.5 of Spark Memory when cached blocks are immune to eviction
- User Memory
  - user data structures and internal metadata in Spark
  - safeguarding against OOM
- Reserved memory
  - memory needed for running executor itself and not strictly related to Spark

Workshop

code available @ github.com/datastrophic/spark-workshop

Execution Modes

- spark-shell --master [local | spark | yarn-client | mesos]
  - launches REPL connected to specified cluster manager
  - always runs in client mode
- spark-submit --master [local | spark:// | mesos:// | yarn] spark-job.jar
  - launches assembly jar on the cluster

- Masters
  - ***local[k]*** - run Spark locally with K worker threads
  - ***spark*** - launches driver app on Spark Standalone installation
  - ***mesos*** - driver will spawn executors on Mesos cluster (***deploy-mode: client | cluster***)
  - **yarn** - same idea as with Mesos (***deploy-mode: client | cluster***)

- Deploy Modes
  - ***client*** - driver executed as a separate process on the machine where it has been launched and spawns executors
  - ***cluster*** - driver launched as a container using underlying cluster manager

Invocation examples

```
spark-shell \
--master yarn \
--deploy-mode client \
--executor-cores 1 \
--num-executors 2 \
--jars /target/spark-workshop.jar \
--conf spark.cassandra.connection.host=cassandra


spark-submit --class io.datastrophic.spark.workshop.ParametrizedApplicationExample \
--master yarn \
--deploy-mode cluster \
--num-executors 2 \
--driver-memory 1g \
--executor-memory 1g \
/target/spark-workshop.jar \
--cassandra-host cassandra \
--keyspace demo \
--table event \
--target-dir /workshop/dumps
```

Live Demo

- spark-shell
- Spark UI
- creating an app with Typesafe Activator
- Spark SQL and DataFrames API
- coding

Coding ideas

- get familiar with API through sample project
  - join data from different storage systems
  - aggregate data with breakdown by date
- play with caching and persistence
- check out join behavior applying different partitioning
- familiarize with Spark UI
- experiment with new DataSet API (since 1.6)
- [your awesome idea here]

Questions

@antonkirillov datastrophic.io