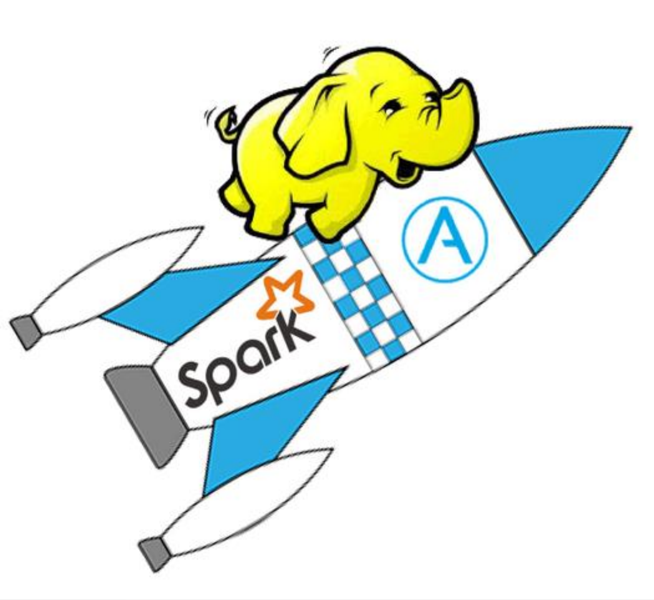


Spark Computational Model

2017.4 XenRon

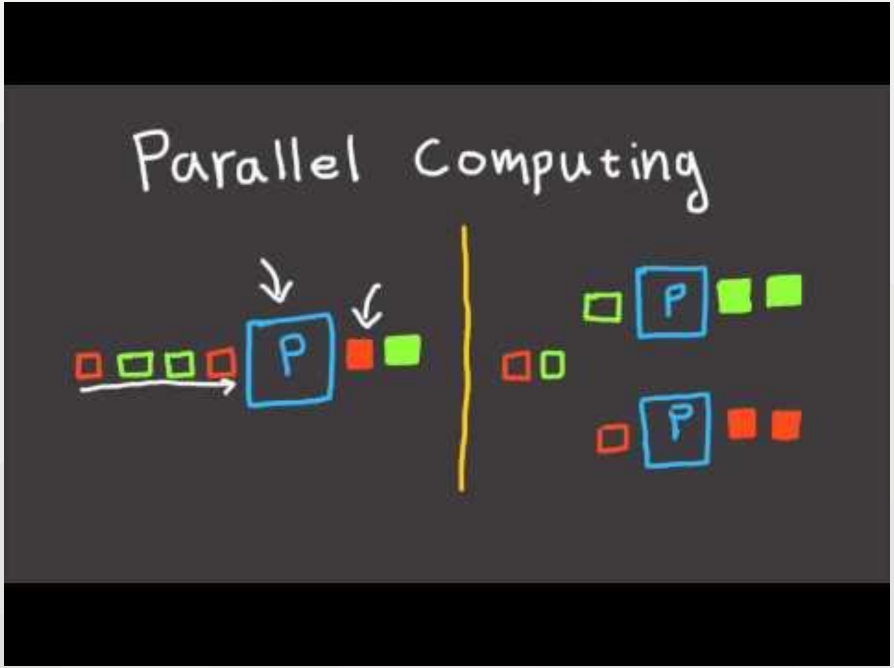
CONTENTS

Spark RDD 01

Parallel Computing 02

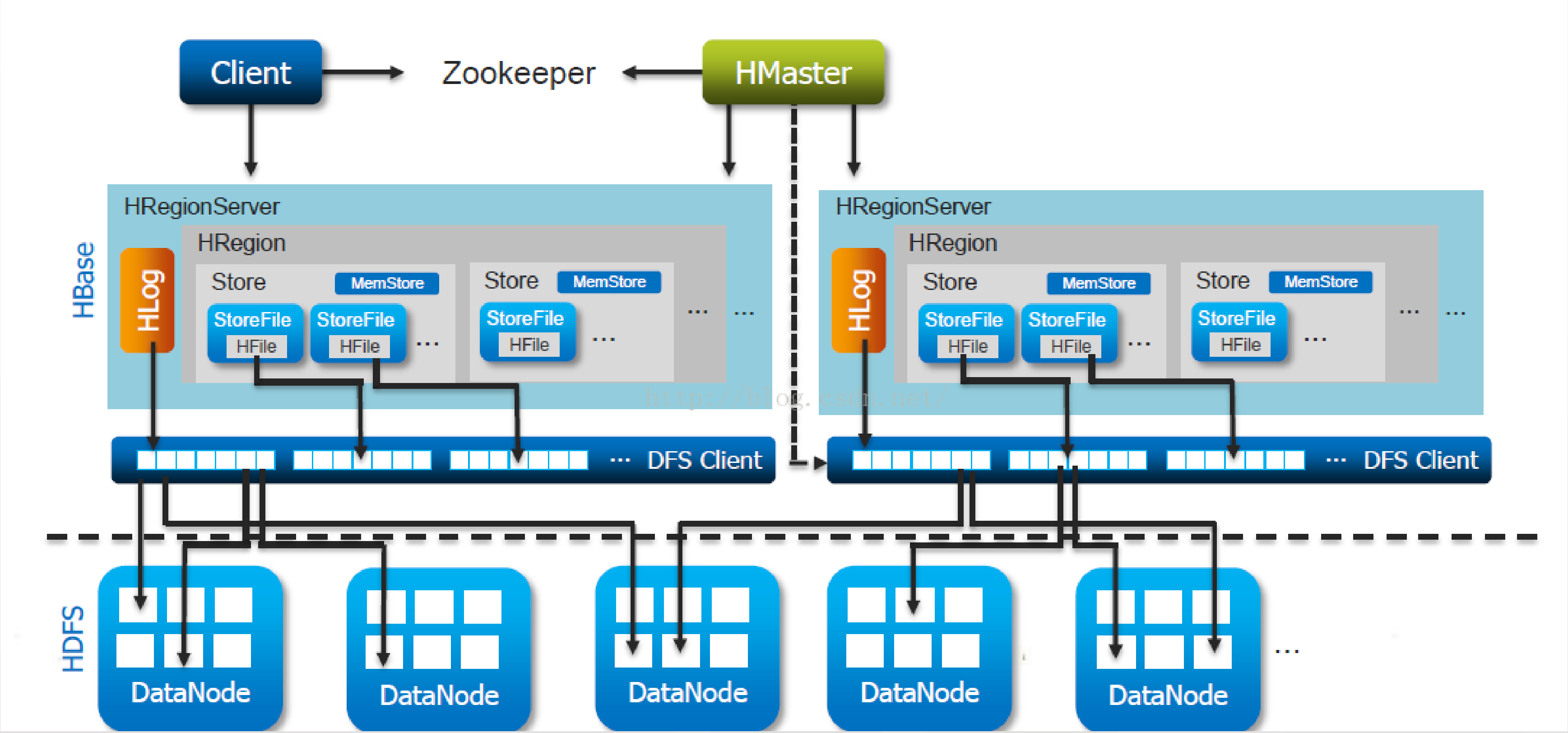
Computational Model 03

Spark SQL 04

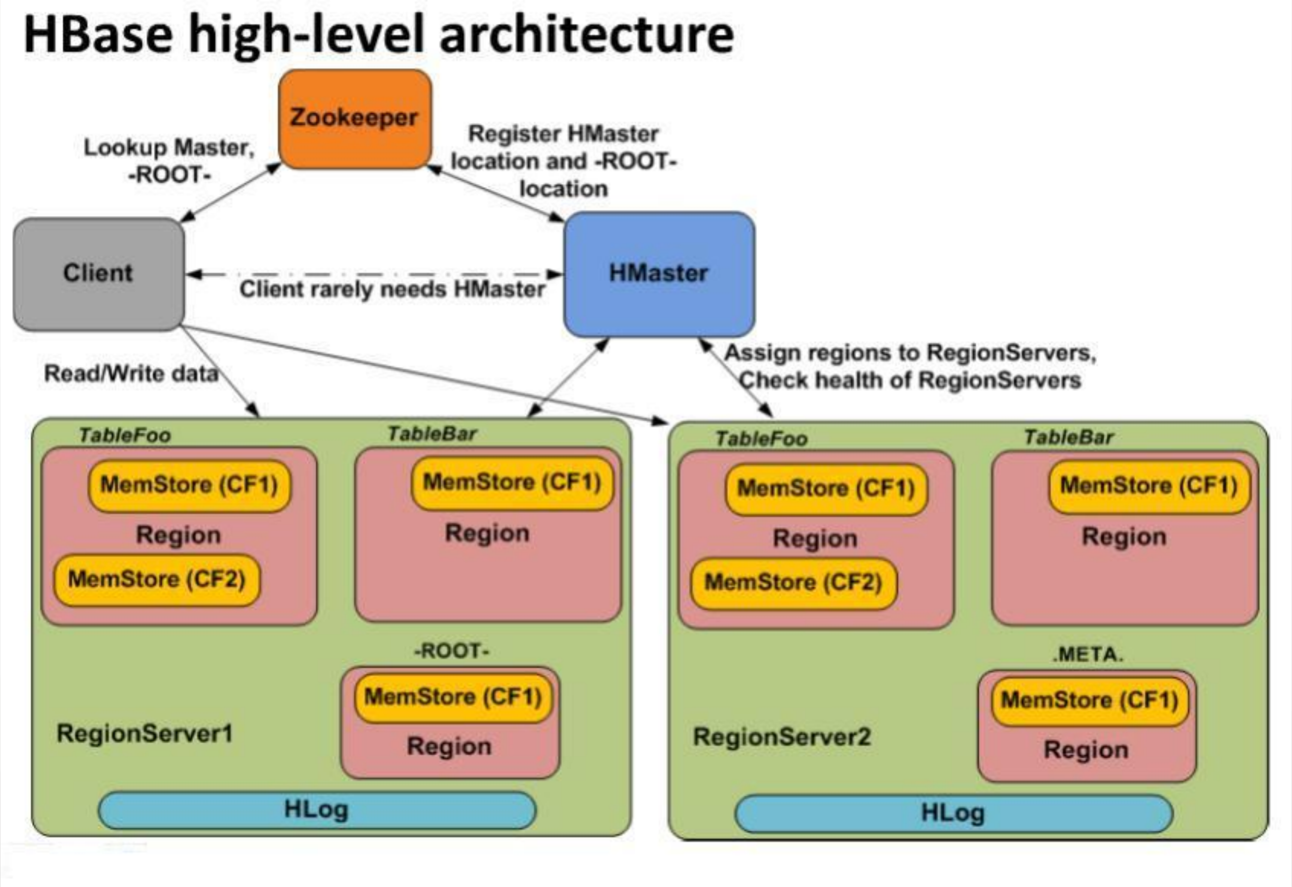


Review

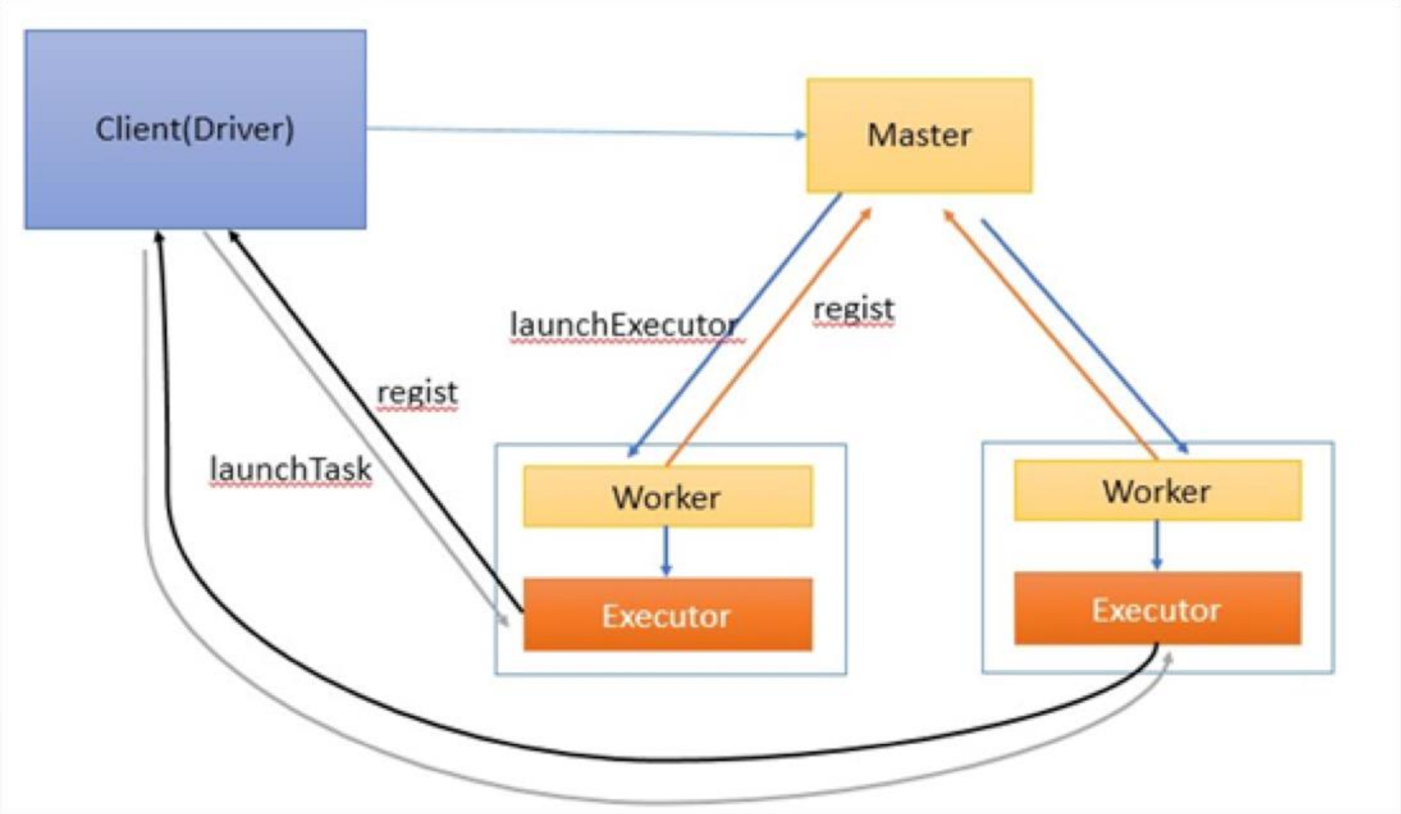
HBase


Client
Zookeeper
HMaster
HBase
HRegionServer
HRegion
HLog
Store
MemStore
StoreFile
HFile
...
DFS Client
HRegionServer
HRegion
HLog
Store
MemStore
StoreFile
HFile
...
DFS Client
HDFS
DataNode
DataNode
DataNode
DataNode
DataNode
...

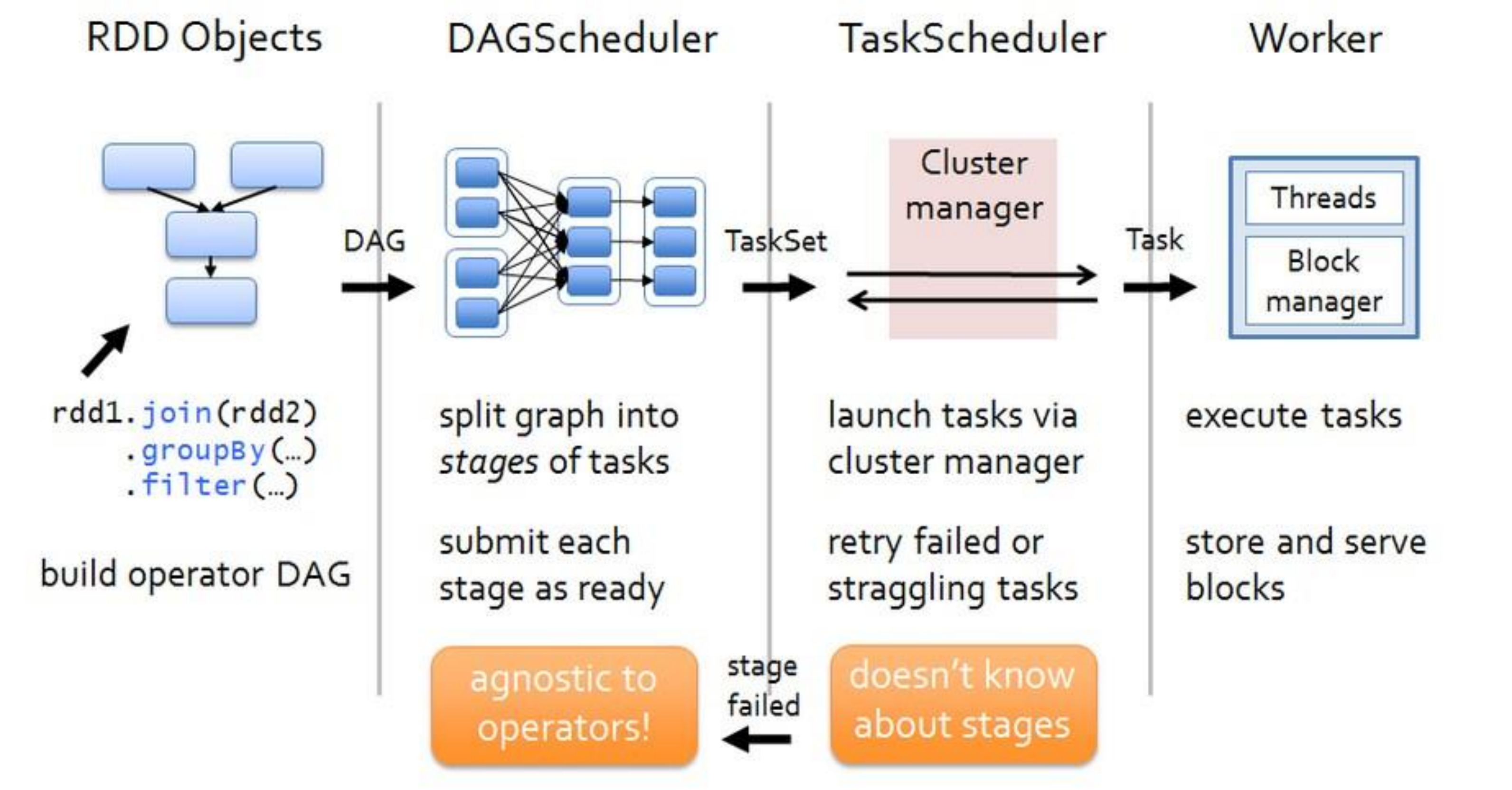
HBase


HBase high-level architecture
Zookeeper
Lookup Master, -ROOT-
Register HMaster location and -ROOT- location
Client
Client rarely needs HMaster
HMaster
Read/Write data
Assign regions to RegionServers, Check health of RegionServers
TableFoo
TableBar
MemStore (CF1)
Region
MemStore (CF2)
MemStore (CF1)
Region
-ROOT-
MemStore (CF1)
Region
RegionServer1
HLog
TableFoo
TableBar
MemStore (CF1)
Region
MemStore (CF2)
MemStore (CF1)
Region
.META.
MemStore (CF1)
Region
RegionServer2
HLog

Spark Standalone



Architecture

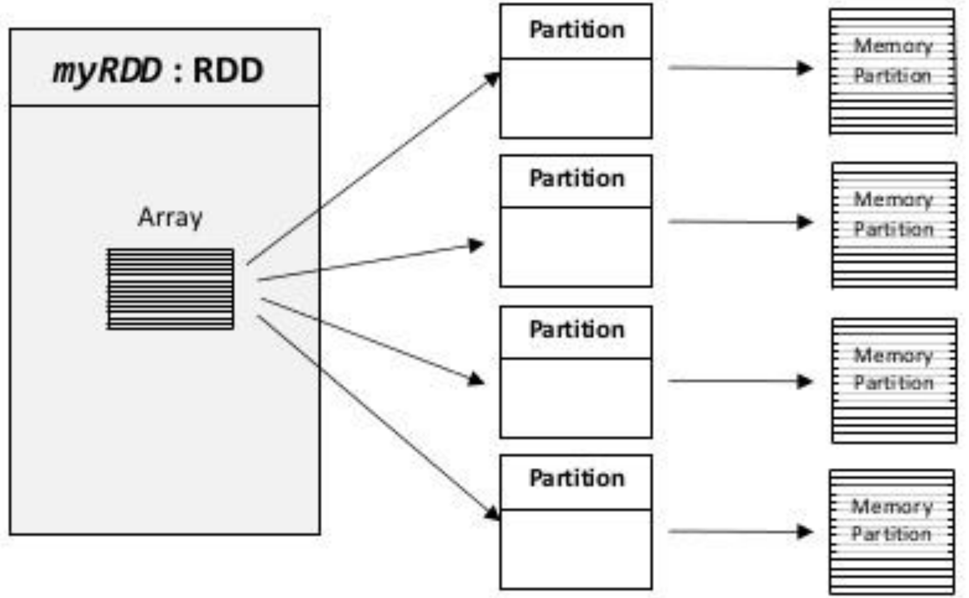


PART1

Spark RDD

RDD

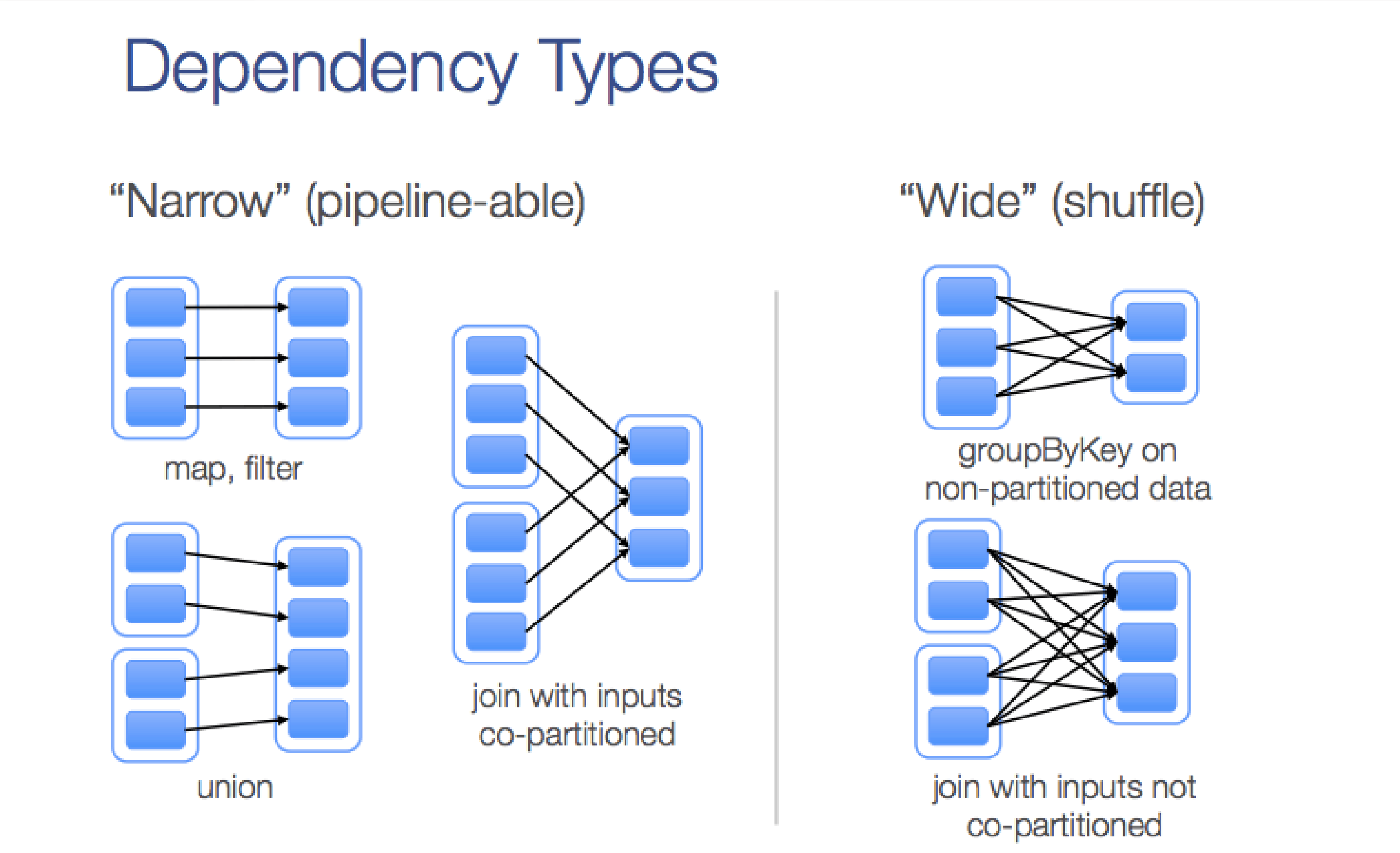
What is an RDD?



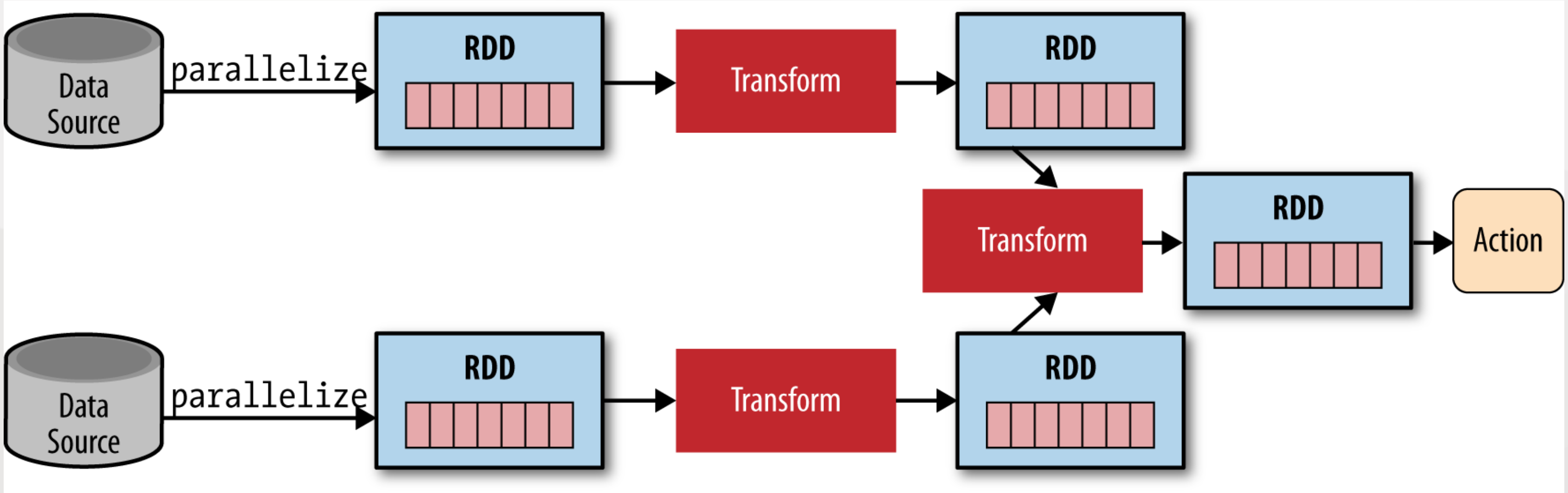
Some RDD Characteristics

- Hold references to Partition objects
- Each Partition object references a subset of your data
- Partitions are assigned to nodes on your cluster
- Each partition/split will be in RAM (by default)

Dependency Type



Dependency Type



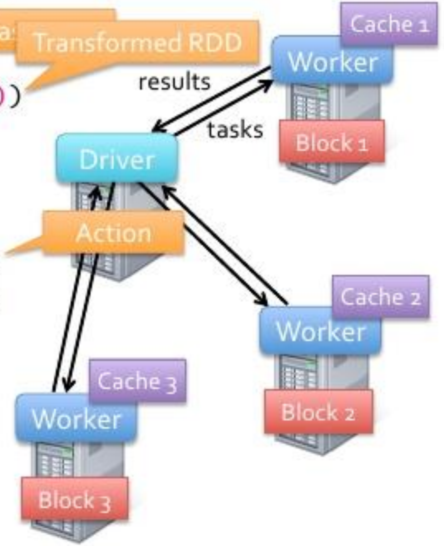
Example

Transformations	$map(f : T \Rightarrow U)$	:	RDD[T] ⇒ RDD[U]
	$filter(f : T \Rightarrow Bool)$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : T \Rightarrow Seq[U])$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction : Float)$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (V, V) \Rightarrow V)$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : V \Rightarrow W)$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : Comparator[K])$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : Partitioner[K])$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (T, T) \Rightarrow T)$	:	RDD[T] ⇒ T
	$lookup(k : K)$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : String)$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```

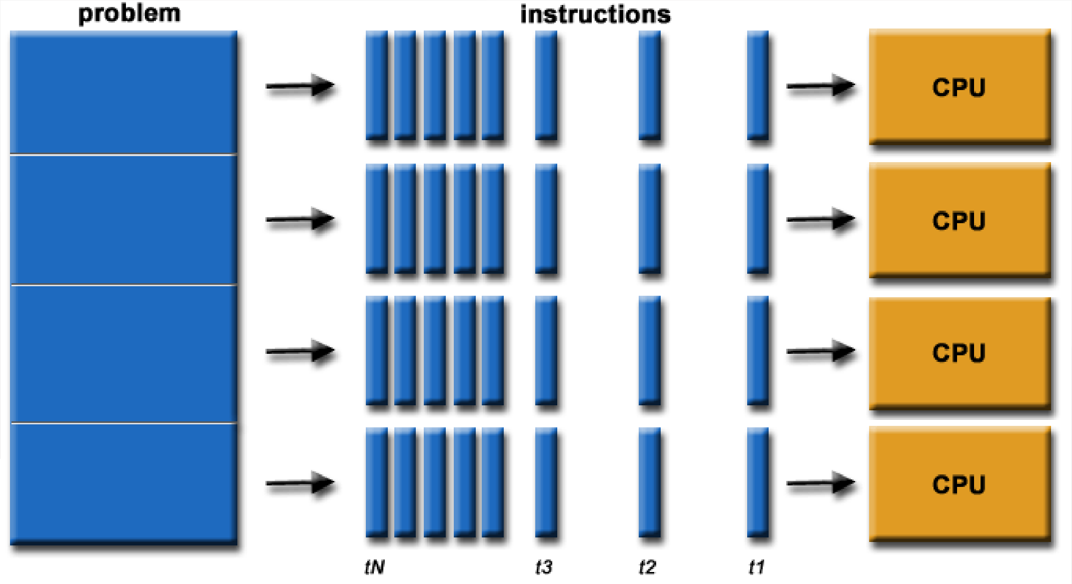


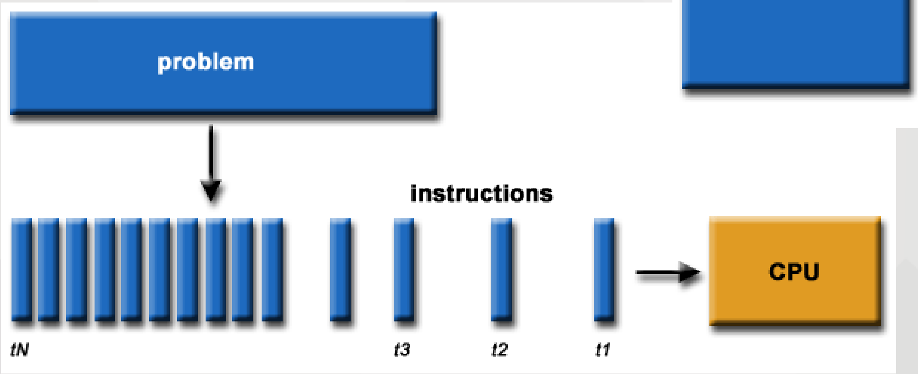
Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

PART2

Parallel Computing

Parallel Computing



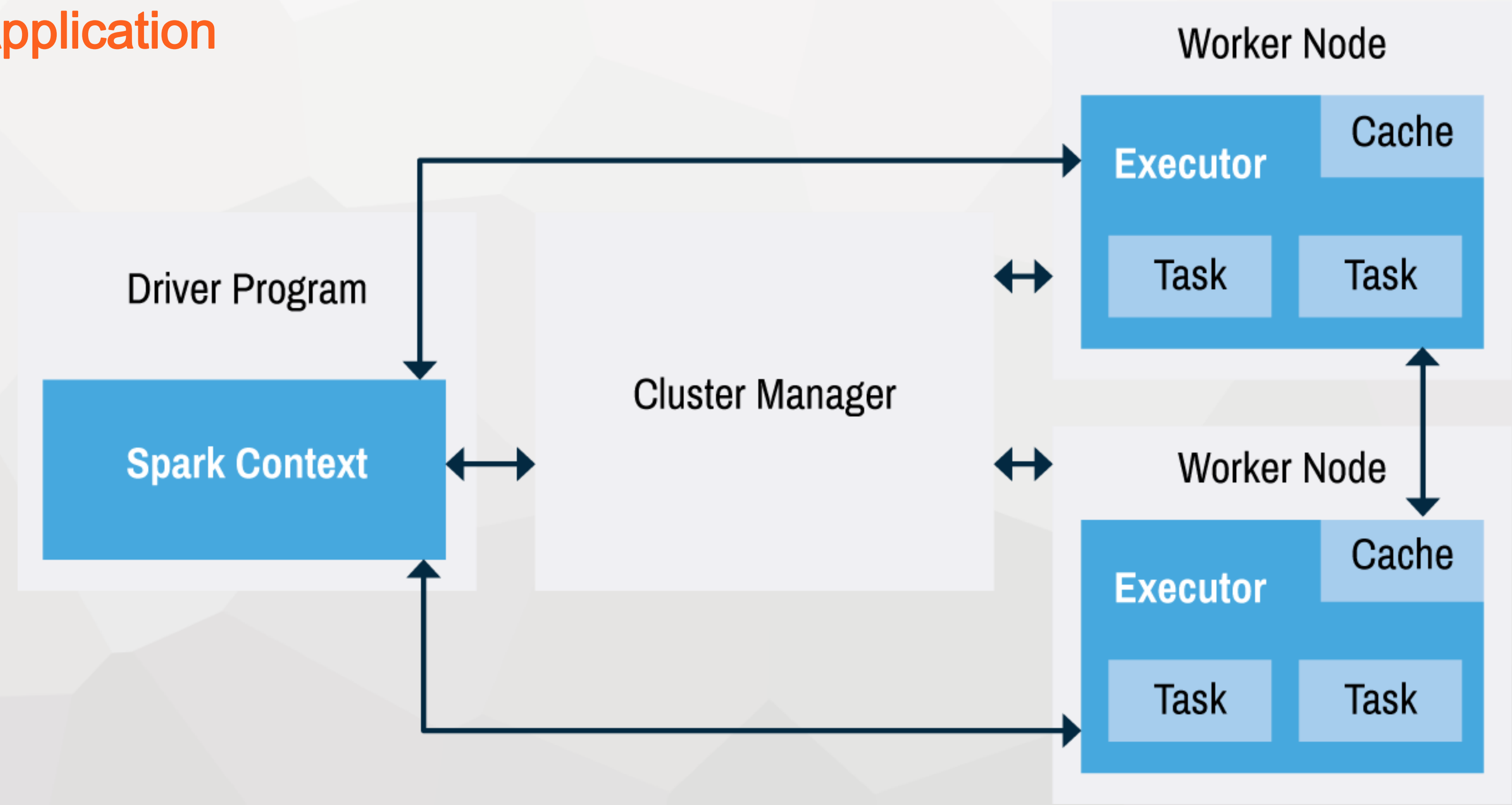


Parallel Computing

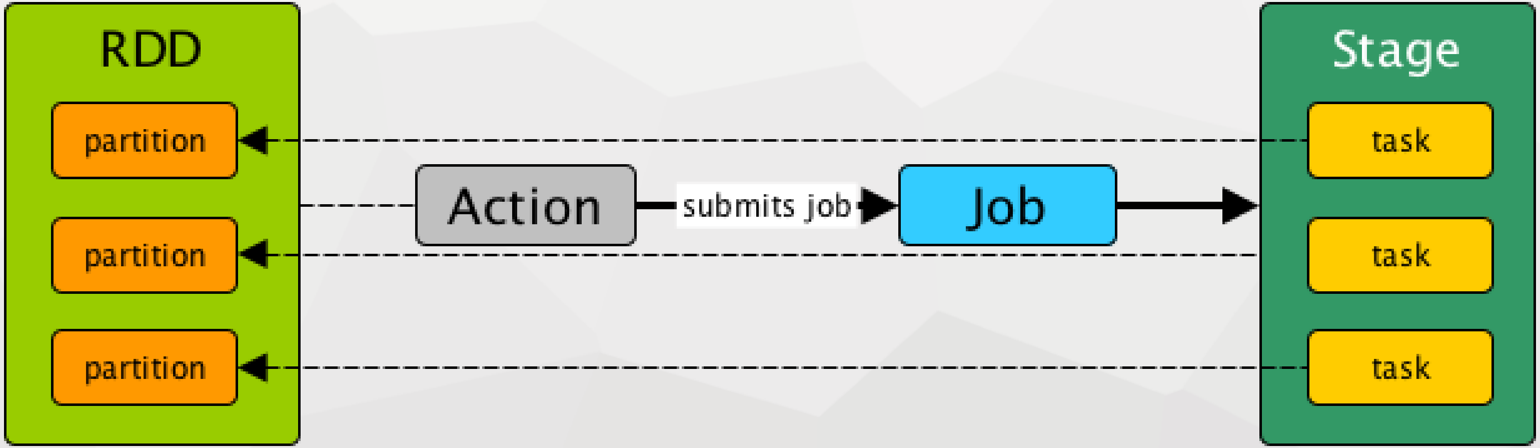
PART3

Computational Model

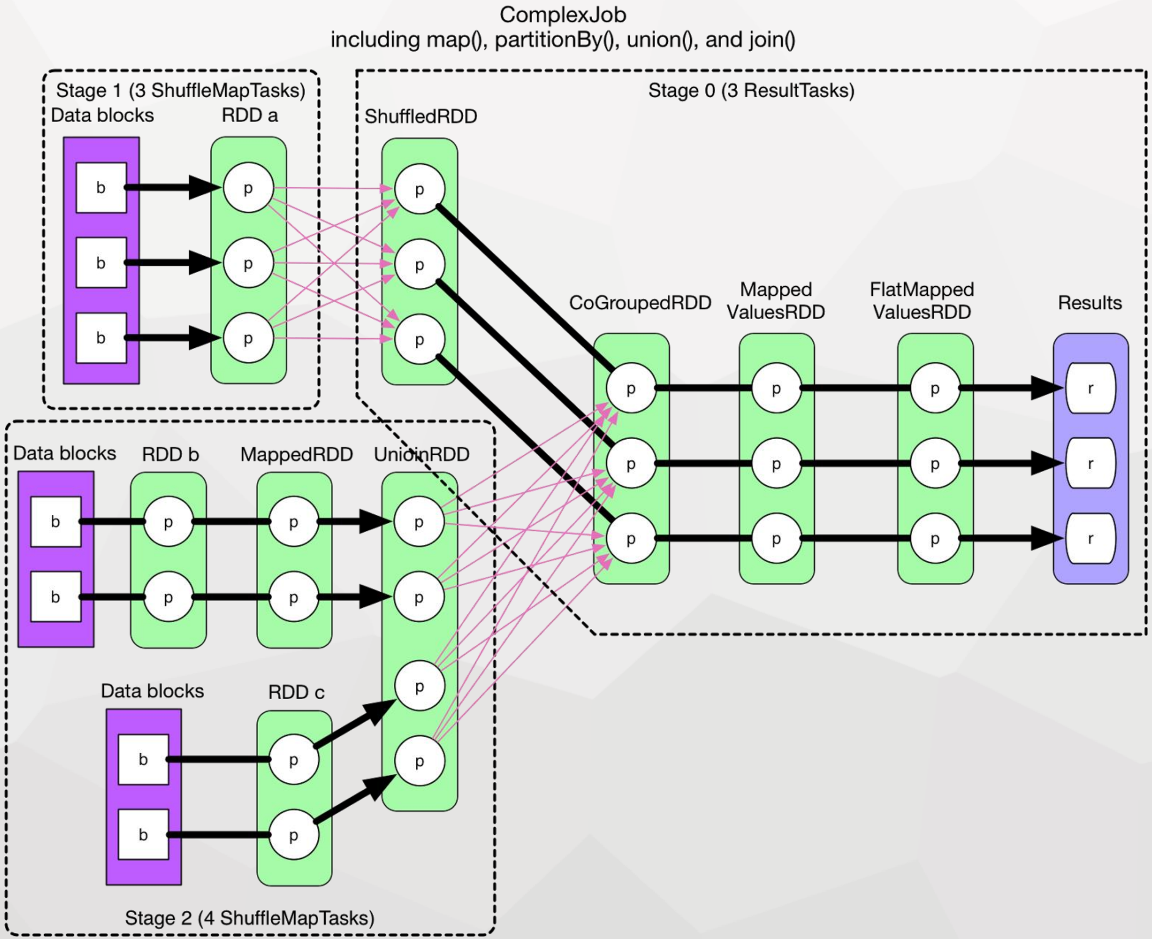
Driver
Executor
Application



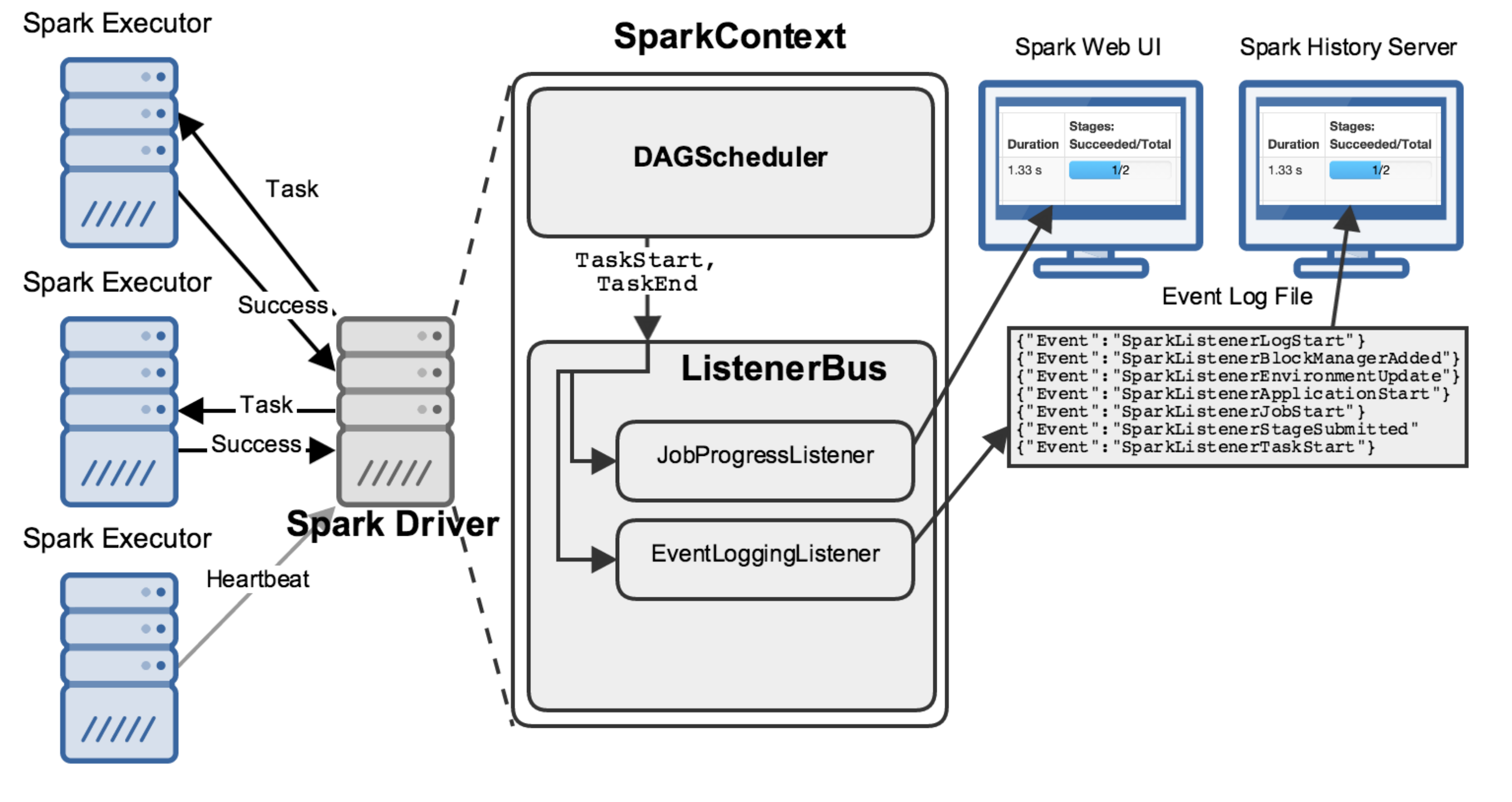
Job


RDD
partition
partition
partition
Action
submits job
Job
Stage
task
task
task

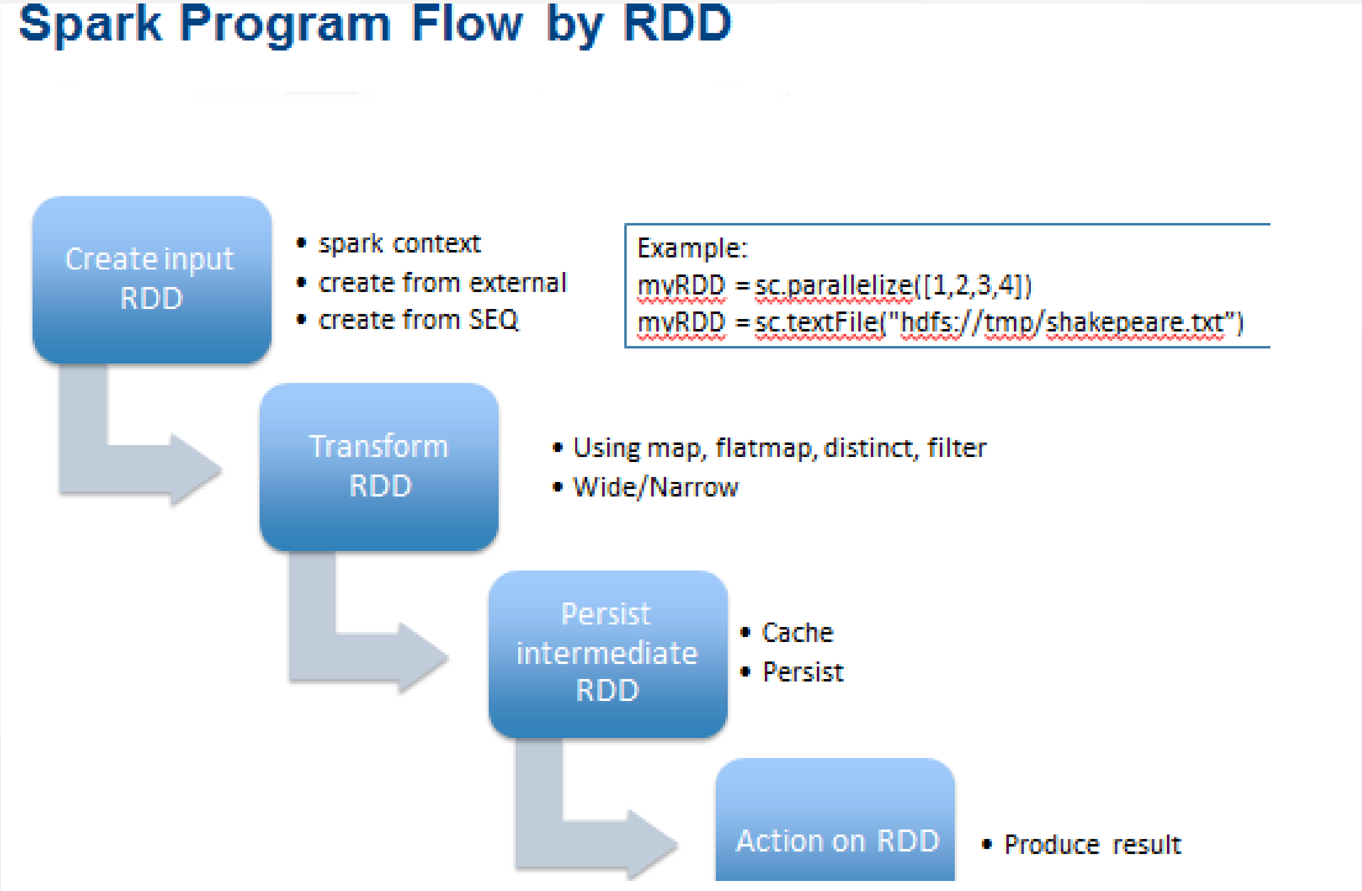
Stage


ComplexJob
including map(), partitionBy(), union(), and join()
Stage 1 (3 ShuffleMapTasks)
Data blocks
RDD a
b
p
Stage 0 (3 ResultTasks)
ShuffledRDD
CoGroupedRDD
Mapped ValuesRDD
FlatMapped ValuesRDD
Results
r
Data blocks
RDD b
MappedRDD
UnionRDD
Data blocks
RDD c
Stage 2 (4 ShuffleMapTasks)

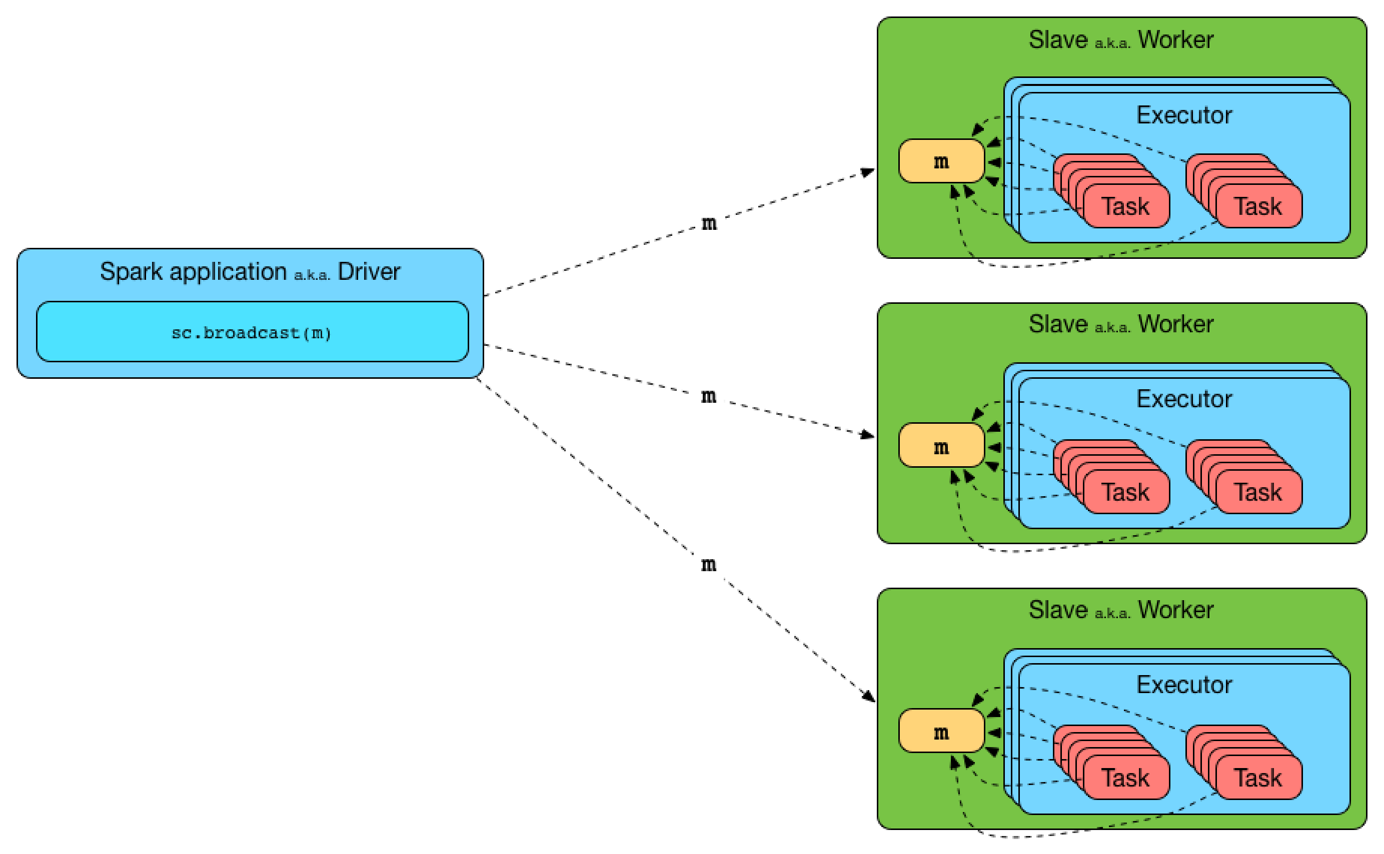
Architecture



Persist & Cache



Broad Cast


Spark application a.k.a. Driver
sc.broadcast(m)
m
m
m
Slave a.k.a. Worker
Executor
m
Task
Task
Slave a.k.a. Worker
Executor
m
Task
Task
Slave a.k.a. Worker
Executor
m
Task
Task

Accumulators

ACCUMULATORS

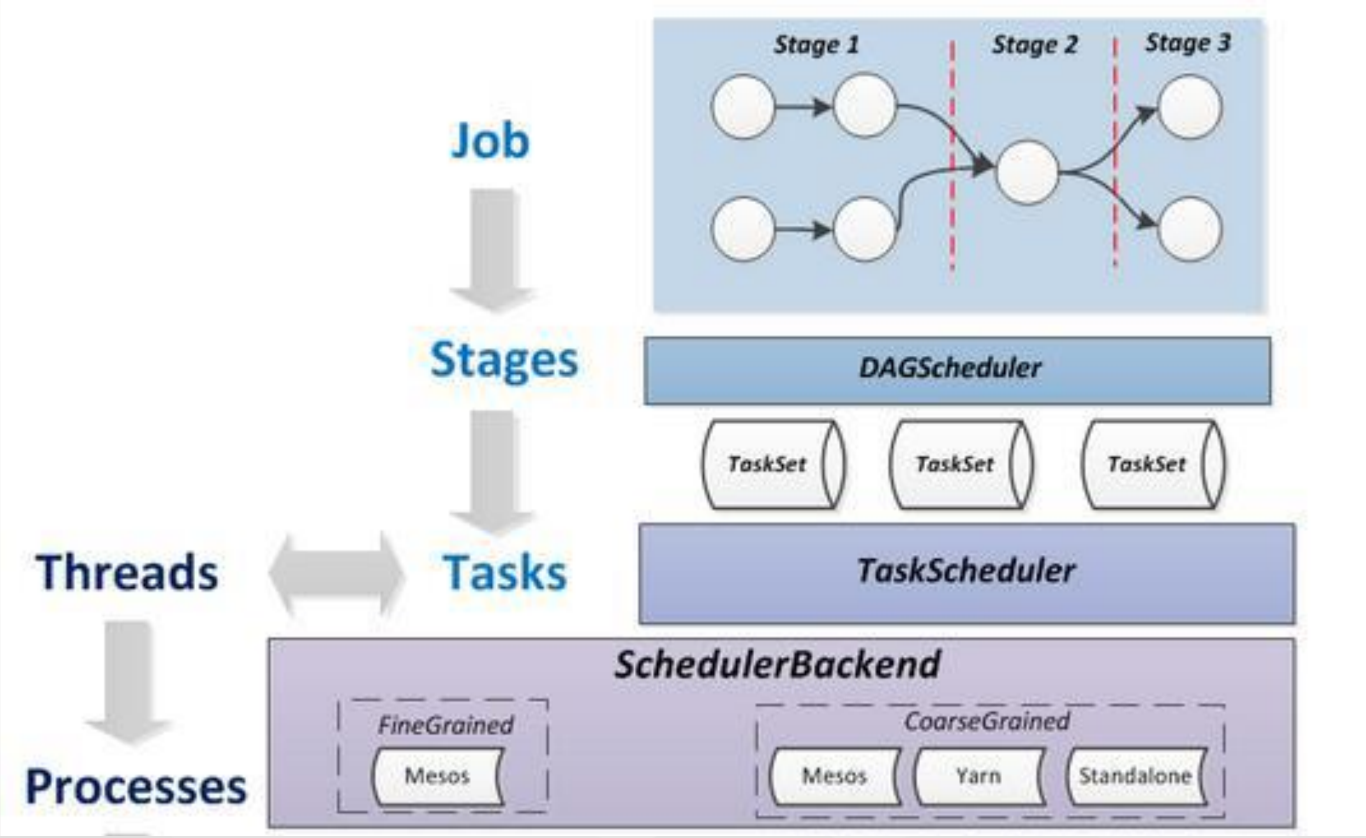
- variables that are only "added" to through an associative operation (add())
- only the driver program can read the accumulator's value

```
Accumulator<Integer> accum = sc.accumulator(0);

sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x ->
  accum.add(x));

accum.value();
// returns 10
```

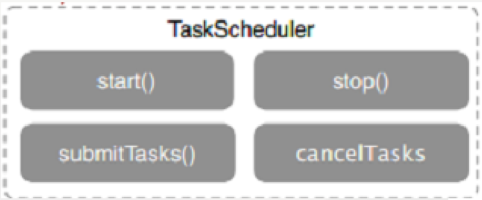
Word Count

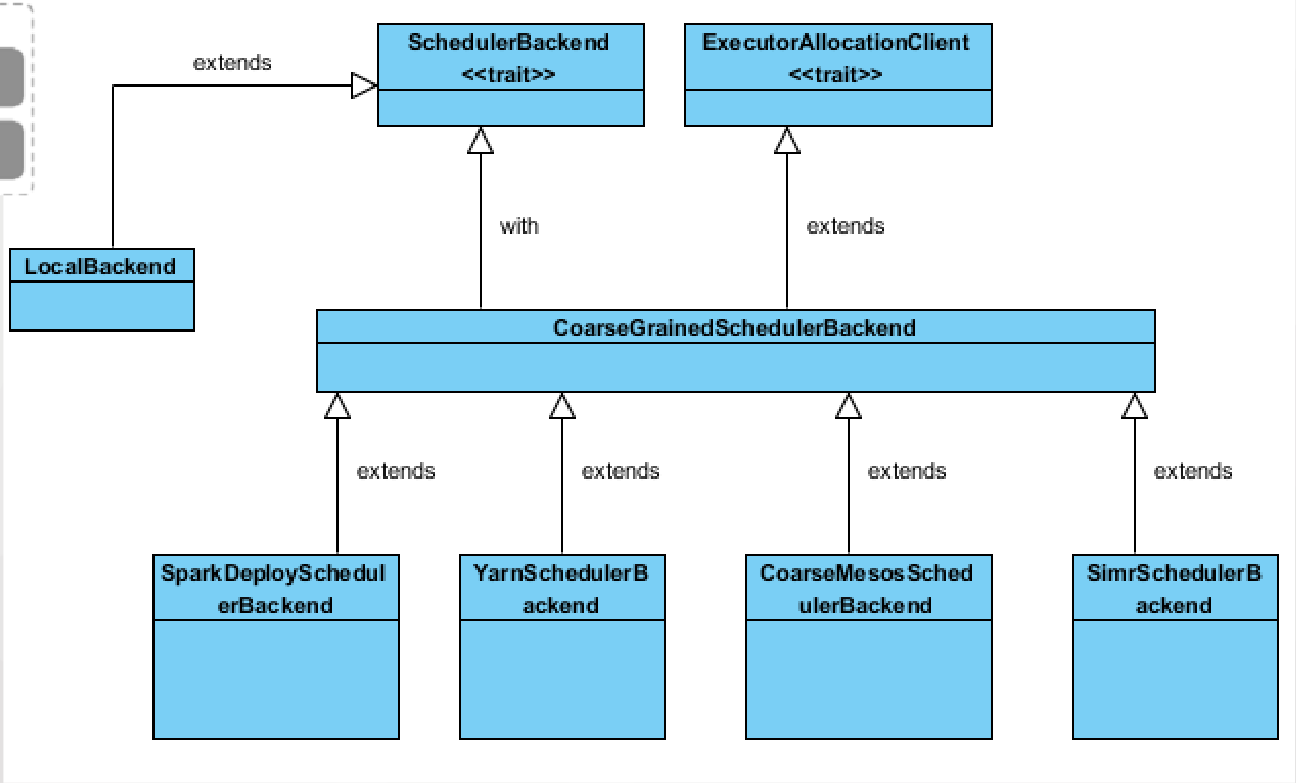

```
import sys
from pyspark import SparkContext
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD
from numpy import array

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.replace(',', ' ').split(' ')]
    return LabeledPoint(values[0], values[1:])

sc = SparkContext(appName="LinearRegressionPredict")
data = sc.textFile("data/mllib/ridge-data/lpsa.data")
parsedData = data.map(parsePoint)

# Build the model
model = LinearRegressionWithSGD.train(parsedData)

# Evaluate the model on training data
valuesAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
MSE = valuesAndPreds.map(lambda (v, p): (v - p)**2).reduce(lambda x, y: x + y) /
valuesAndPreds.count()
print("Mean Squared Error = " + str(MSE))
sc.stop()
```

DAGScheduler

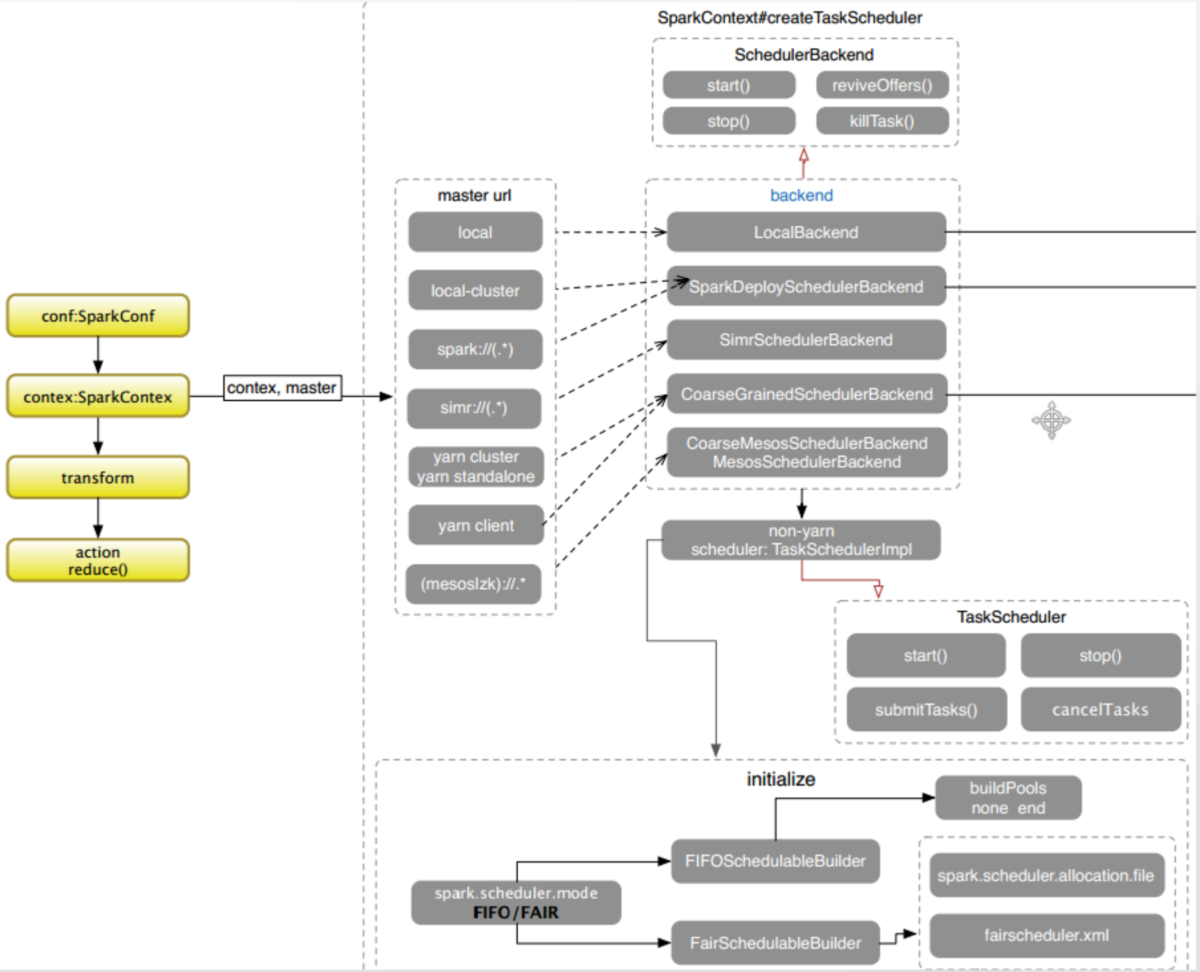


TaskScheduler

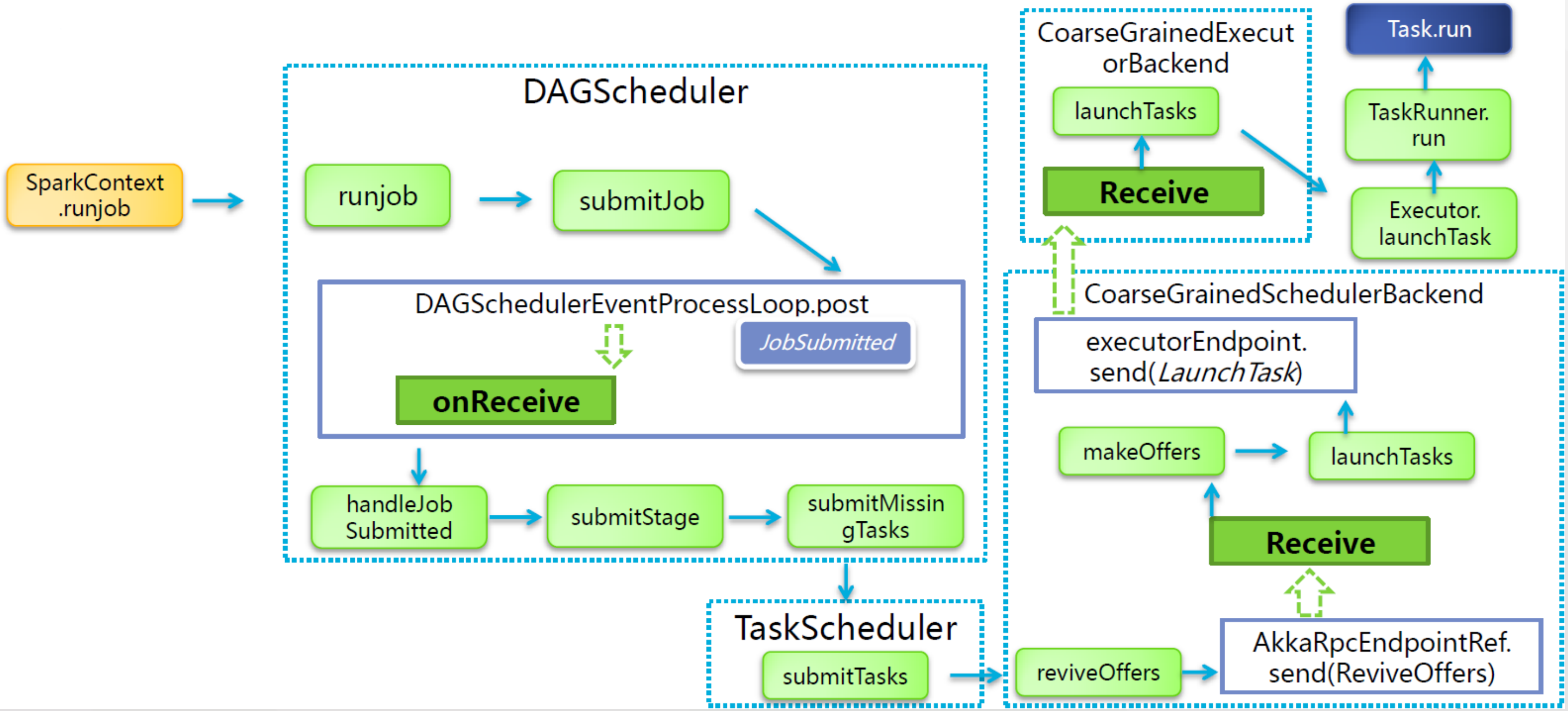




TaskScheduler



TaskScheduler

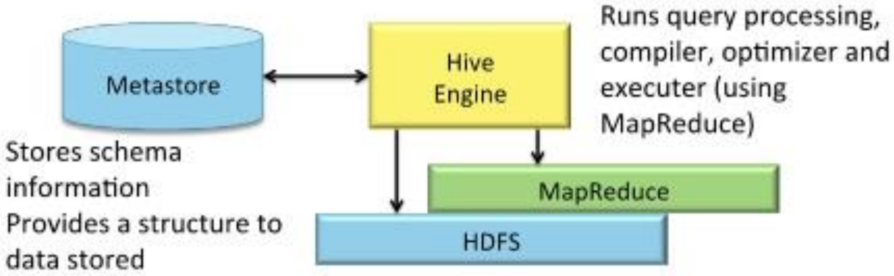


PART4

Spark SQL

Hive

- Data warehousing package built on top of Hadoop
- Bringing structure to unstructured data
- Query petabytes of data with HiveQL
- Schema on read



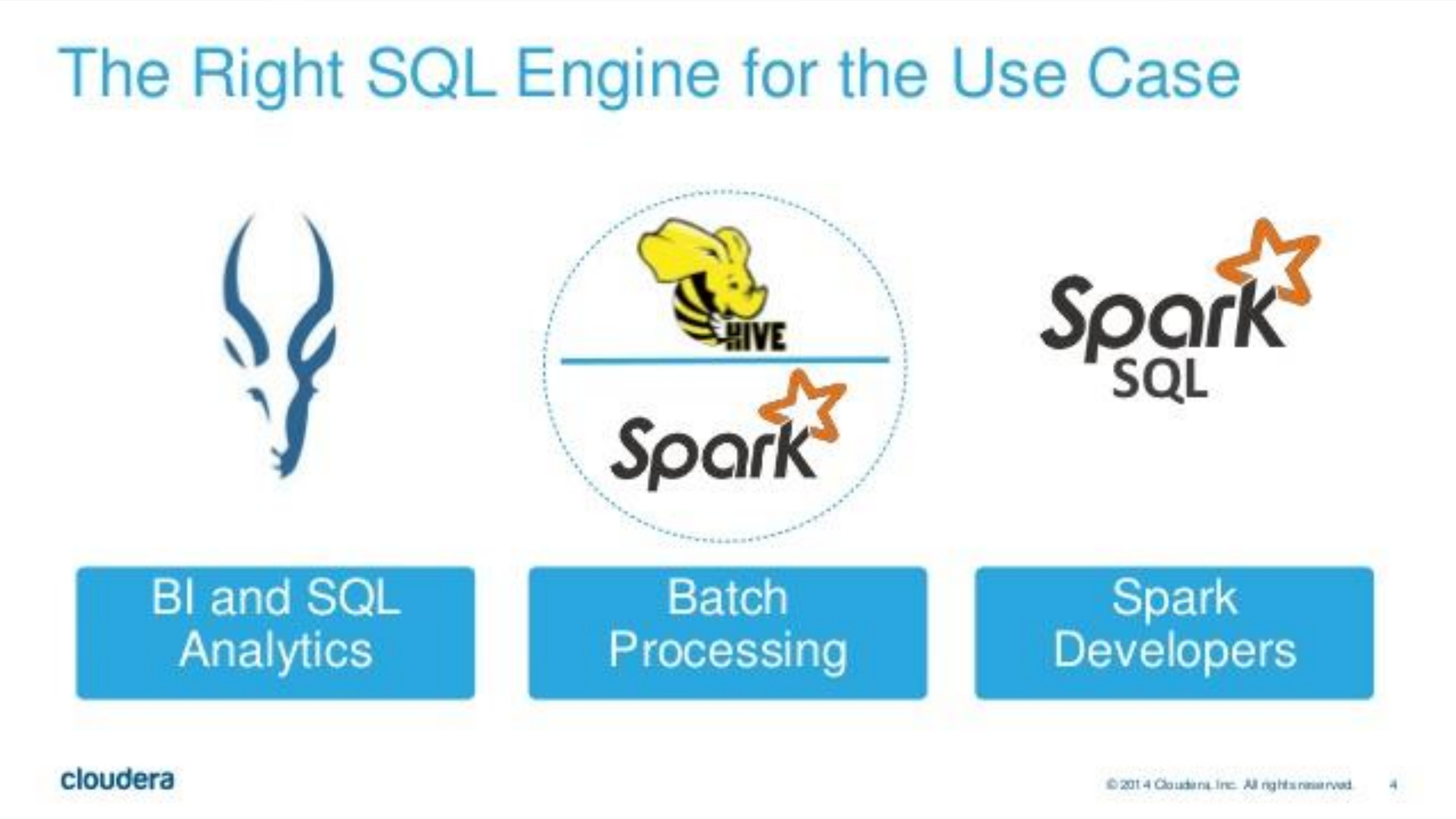
Hive

Hadoop MapReduce Vs Pig Vs Hive

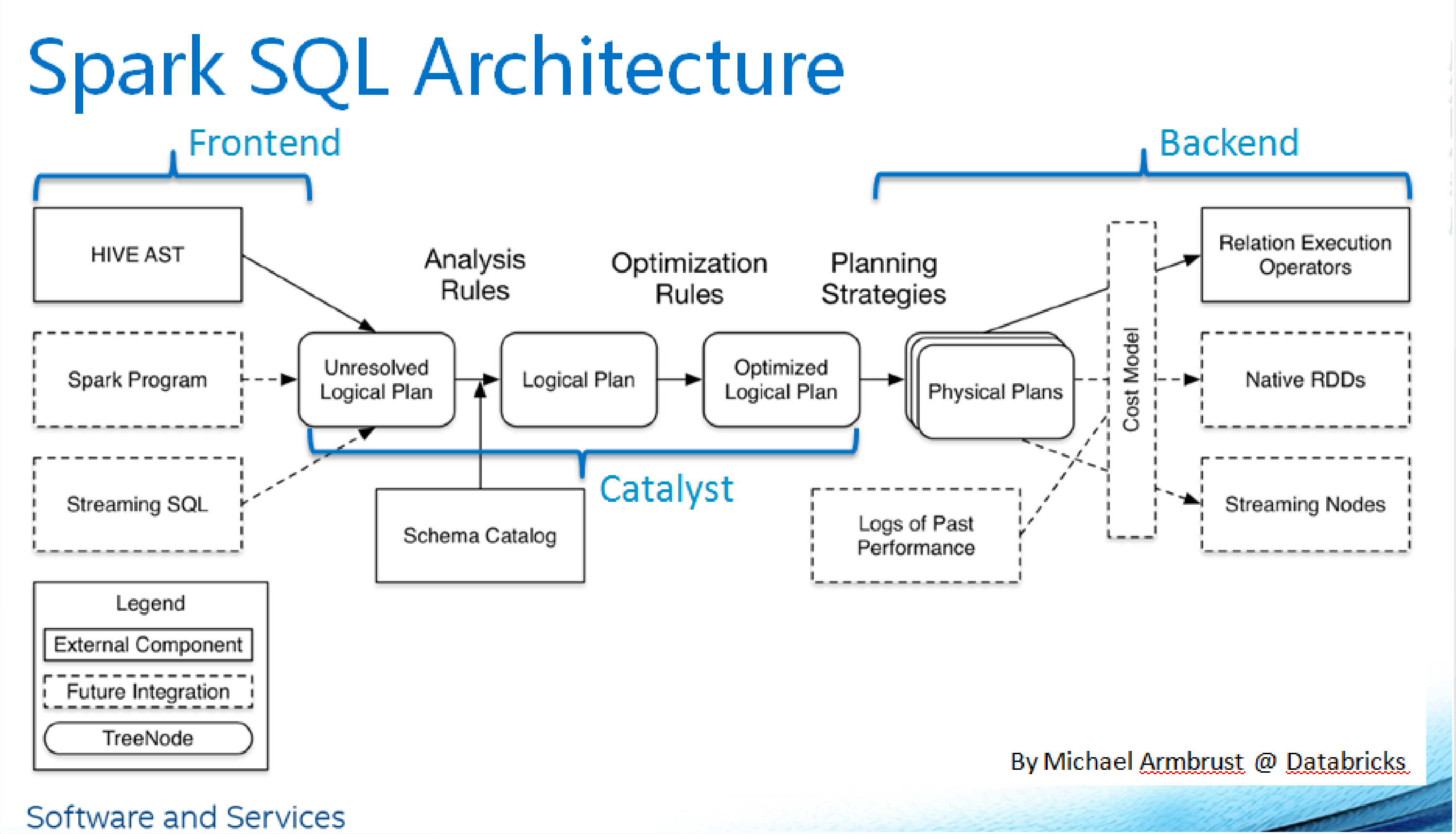
Hadoop MapReduce	Pig	Hive
Compiled Language	Scripting Language	SQL like query Language
Lower Level of Abstraction	Higher Level of Abstraction	Higher Level of Abstraction
More lines of Code	Comparatively less lines of Code than MapReduce	Comparatively less lines of Code than MapReduce and Apache Pig
More Development Effort is involved .	Development Effort is less Code Efficiency is relatively less	Development Effort is less Code Efficiency is relatively less
Code Efficiency is high when compared to Pig and Hive	Code Efficiency is relatively less	Code Efficiency is relatively less

DeZyre

Spark SQL


The Right SQL Engine for the Use Case
HIVE
Spark
Spark
SQL
BI and SQL Analytics
Batch Processing
Spark Developers
cloudera
© 2014 Cloudera, Inc. All rights reserved.
4

Spark SQL


Spark SQL Architecture
Frontend
Backend
HIVE AST
Spark Program
Streaming SQL
Unresolved Logical Plan
Analysis Rules
Logical Plan
Optimization Rules
Optimized Logical Plan
Planning Strategies
Physical Plans
Cost Model
Relation Execution Operators
Native RDDs
Streaming Nodes
Catalyst
Schema Catalog
Logs of Past Performance
Legend
External Component
Future Integration
TreeNode
By Michael Armbrust @ Databricks
Software and Services

The End