DATASTAX

Introduction to Spark

DuyHai DOAN, Technical Advocate

Shameless self-promotion

Duy Hai DOAN

Cassandra technical advocate

- talks, meetups, confs
- open-source devs (**Achilles**, …)
- OSS Cassandra point of contact
  ☞ **duy_hai.doan@datastax.com**
- **production** troubleshooting

Datastax

- Founded in **April 2010**
- We contribute **a lot** to Apache Cassandra™
- **400+** customers (25 of the Fortune 100), **200+** employees
- Headquarter in San Francisco Bay area
- EU headquarter in **London**, offices in **France** and **Germany**
- **Datastax Enterprise** = OSS Cassandra + **extra features**

Agenda

- Spark eco-system
- RDD abstraction
- Spark architecture & job life-cycle
- Spark core API
- Spark SQL
- Spark Streaming

Spark eco-system

Technology landscape

Spark eco-system

What is Apache Spark ?

Apache Project since 2010

General data processing framework

MapReduce **is not** the α & ω

One-framework-many-components approach

Data processing landscape

Pregel Graph Google

Giraph Graph Apache

… GraphLab Graph Dato

Data processing landscape

Pregel Graph Google

Giraph Graph Apache

Drill SQL Apache

Dremel SQL Google

Impala SQL Cloudera

…

GraphLab Graph Dato

Data processing landscape

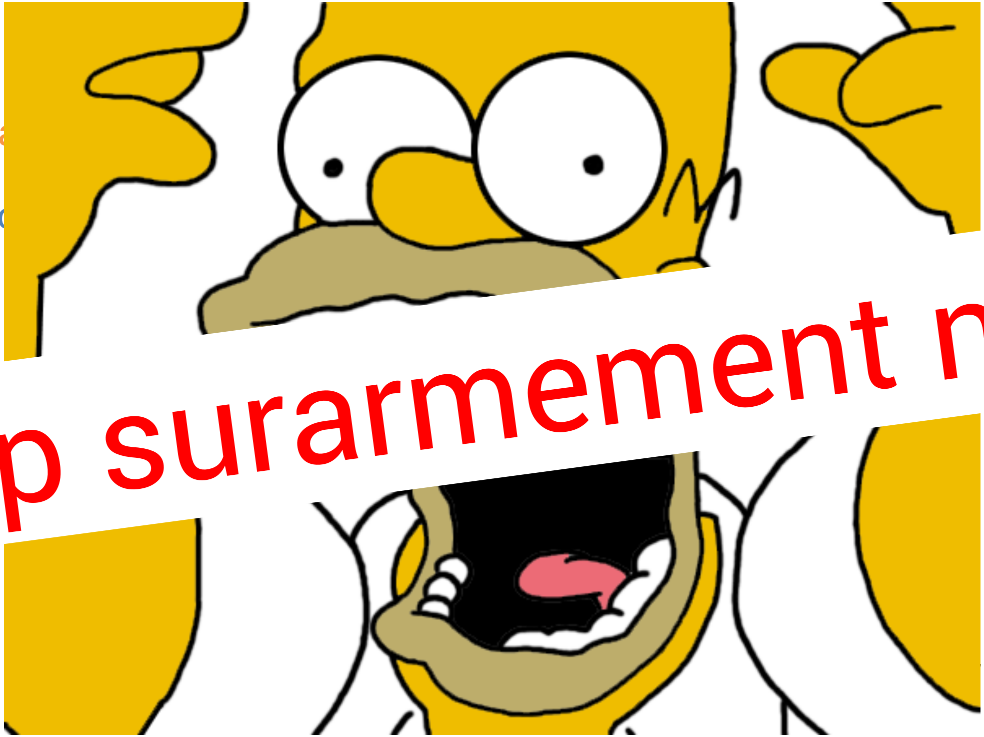
Pregel	Graph	Google
Giraph	Graph	Apache
Drill	SQL	Apache
Dremel	SQL	Google
Tez	DAG	Apache
Impala	SQL	Cloudera
Storm	Stream	Apache
GraphLab	Graph	Dato

…

Data processing landscape

Pregel Gra
Goo

Giraph Graph
Apach



Stop surarmement now !

ez
Apache

aphLab Graph
Dato

Apache

Spark characteristics

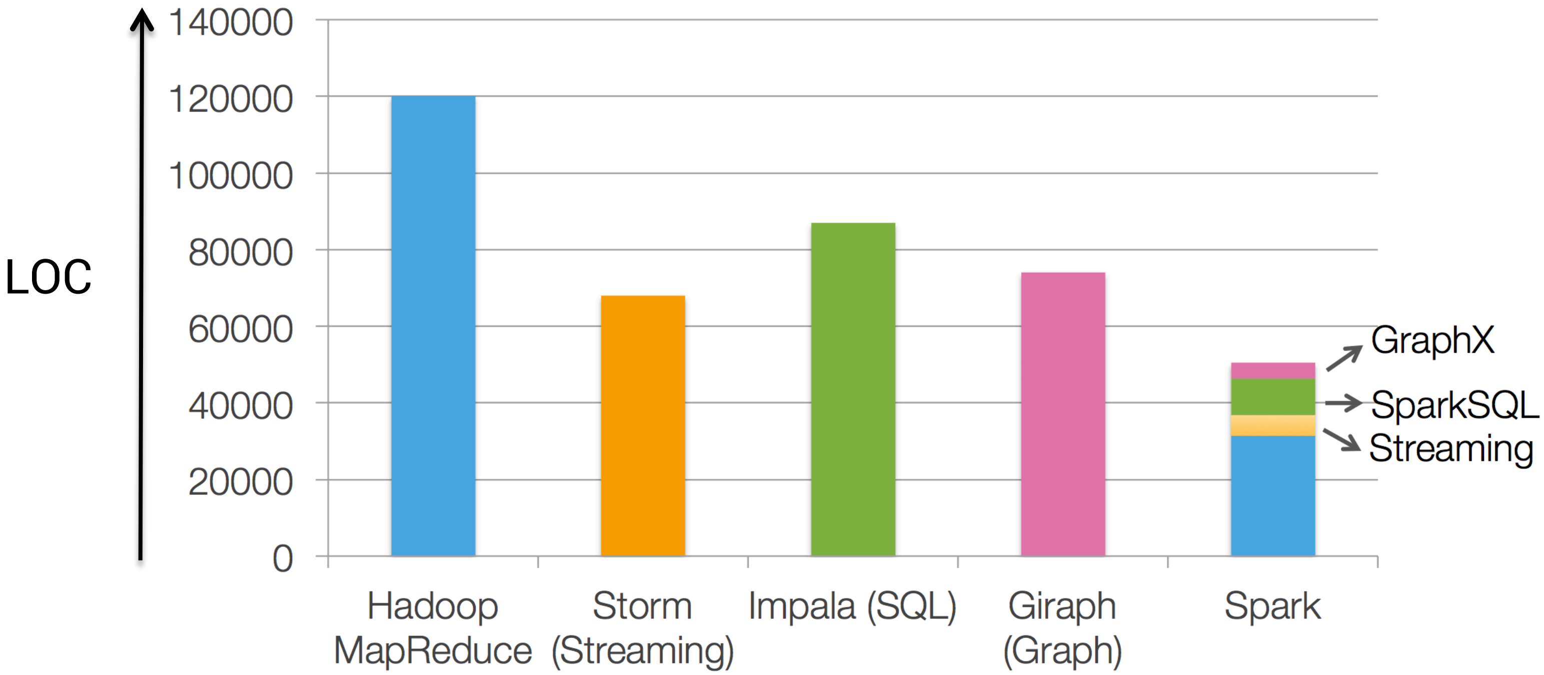
Fast

- 10x-100x faster than Hadoop MapReduce
- In-memory storage
- Single JVM process per node, multi-threaded

Easy

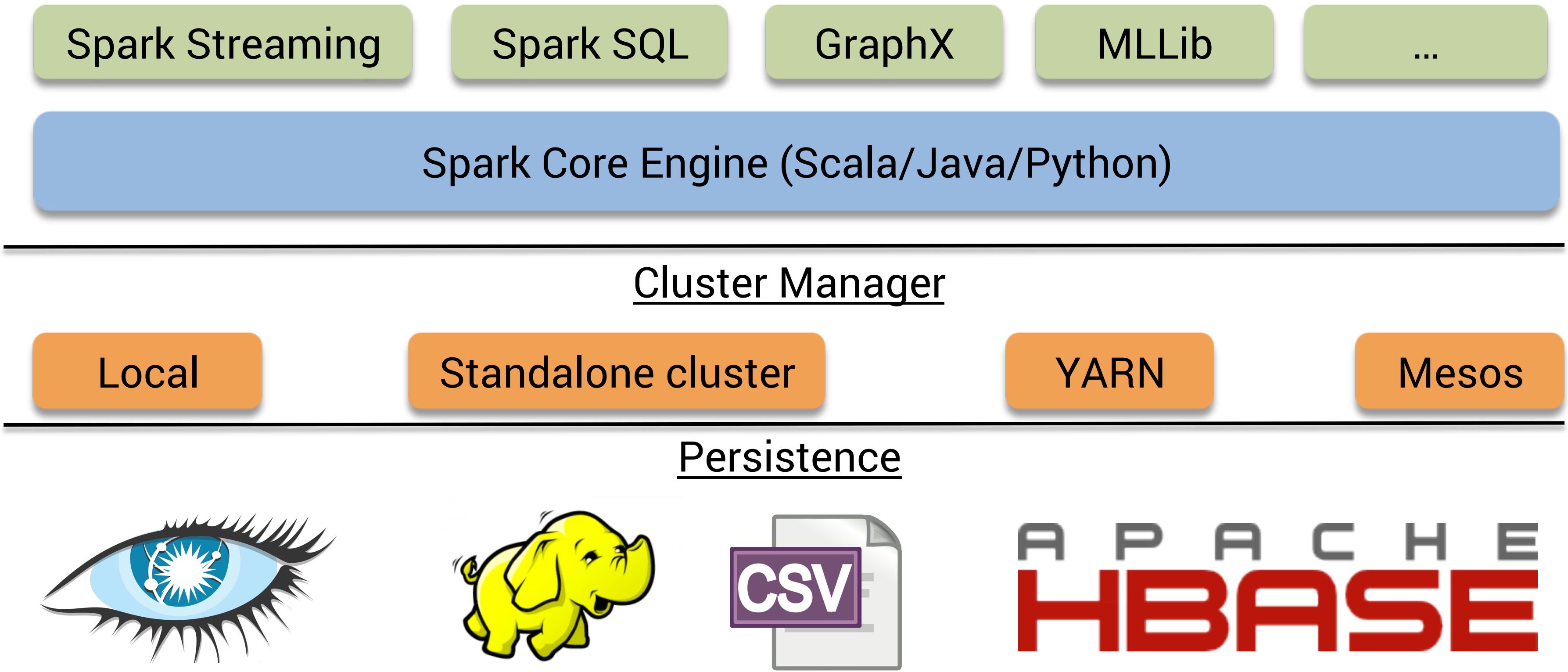
- Rich Scala, Java and Python APIs (R is coming …)
- 2x-5x less code
- Interactive shell

Spark comparison



non-test, non-example source lines

Spark eco-system



Spark Streaming

Spark SQL

GraphX

MLLib

…

Spark Core Engine (Scala/Java/Python)

Cluster Manager

Local

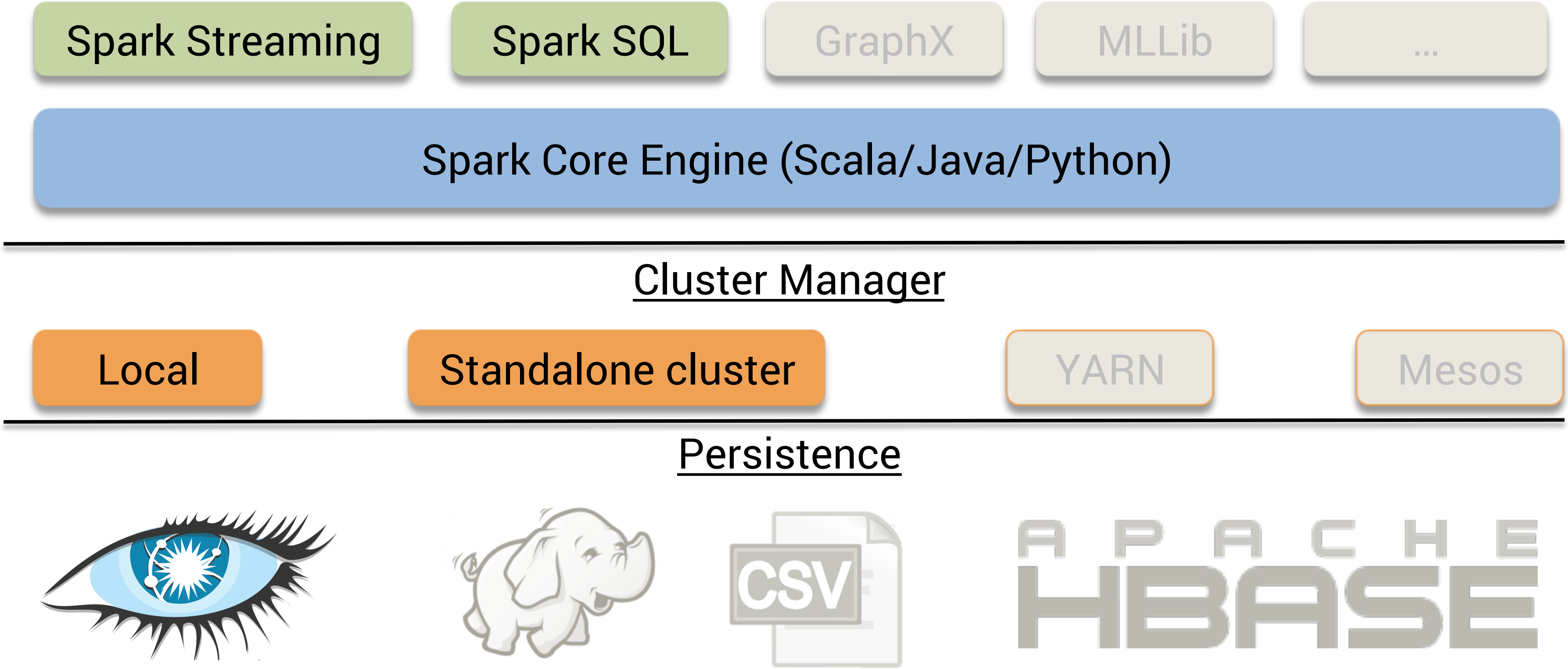
Standalone cluster

YARN

Mesos

Persistence

Spark eco-system



Spark Streaming

Spark SQL

GraphX

MLLib

...

Spark Core Engine (Scala/Java/Python)

Cluster Manager

Local

Standalone cluster

YARN

Mesos

Persistence

Q & R

RDD abstraction

Code example

RDD interface

Code example

Setup

```
val conf = new SparkConf(true)
        .setAppName("basic_example")
        .setMaster("local[3]")

val sc = new SparkContext(conf)
```

Data-set (can be from text, CSV, Json, Cassandra, HDFS, …)

```
val people = List(("jdoe","John DOE", 33),
                  ("hsue","Helen SUE", 24),
                  ("rsmith", "Richard Smith", 33))
```

Code example

Processing

```
// count users by age
 val counByAge = sc.parallelize(people)
                 .map(tuple => (tuple._3, tuple))
                 .groupByKey()
                 .countByKey()

 println("Count by age : "+countByAge)
```

Code example

Decomposition

```
// count users by age
 val counByAge = sc.parallelize(people)  //split into different chunks (partitions)
    .map(tuple => (tuple._3, tuple))  //("jdoe","John DOE", 33) => (33,(("jdoe",…))
    .groupByKey()  //{33 -> (("jdoe",…), ("rsmith",…)), 24->("hsue",…))}
    .countByKey(); //{33 -> 2, 24->1}

 println("Count by age : "+countByAge);  //Count by age = Map(33 -> 2, 24 -> 1)
```

RDDs

RDD = **R**esilient **D**istributed **D**ataset

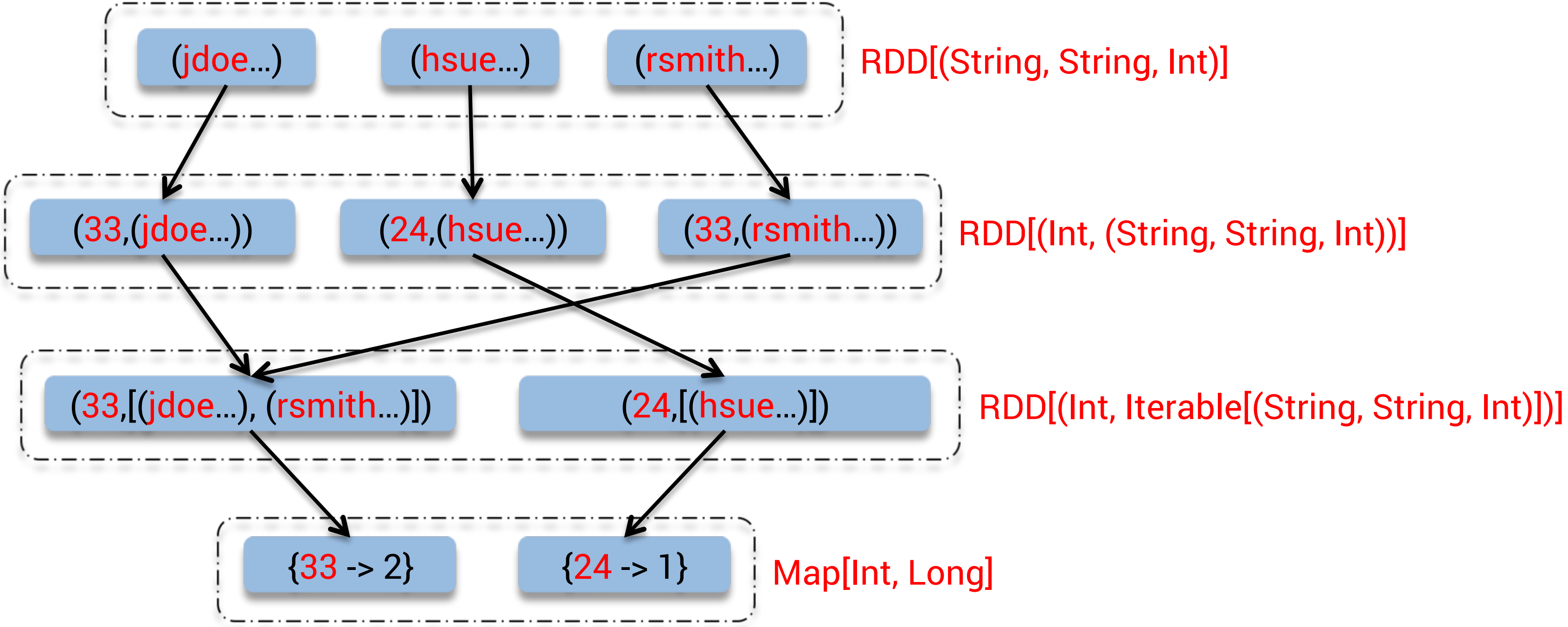
```
val parallelPeople: RDD[(String, String, Int)] = sc.parallelize(people)

val extractAge: RDD[(Int, (String, String, Int))] = parallelPeople
                                                  .map(tuple => (tuple._3, tuple))

val groupByAge: RDD[(Int, Iterable[(String, String, Int)])]=extractAge.groupByKey()

val countByAge: Map[Int, Long] = groupByAge.countByKey()
```

RDDs in action



(jdoe…) (hsue…) (rsmith…)

RDD[(String, String, Int)]

(33,(jdoe…)) (24,(hsue…)) (33,(rsmith…))

RDD[(Int, (String, String, Int))]

(33,[(jdoe…), (rsmith…)]) (24,[(hsue…)])

RDD[(Int, Iterable[(String, String, Int)])]

{33 -> 2} {24 -> 1}

Map[Int, Long]

RDD interface

Interface RDD[A]

- ≈ collection of A objects
- lazy, pull by children transformations

Operations

- transformations
- actions

RDD interface

protected def getPartitions: Array[Partition]

- define partitions (chunks)

protected def getDependencies: Seq[Dependency[_]]

- defines parents RDD
- direct transformations (map, filter,…): 1-to-1 relationship
- aggregations (join, groupByKey …n-to-n relationship

def compute(split: Partition, context: TaskContext): Iterator[T]

lineage

RDD interface

protected def getPreferredLocations(split: Partition): Seq[String]

- for data-locality (HDFS, Cassandra, …)

@transient **val** *partitioner*: Option[Partitioner]

- HashPartitioner, RangePartitioner
- Murmur3Partitioner for Cassandra
- can be none

optimization

Partitions

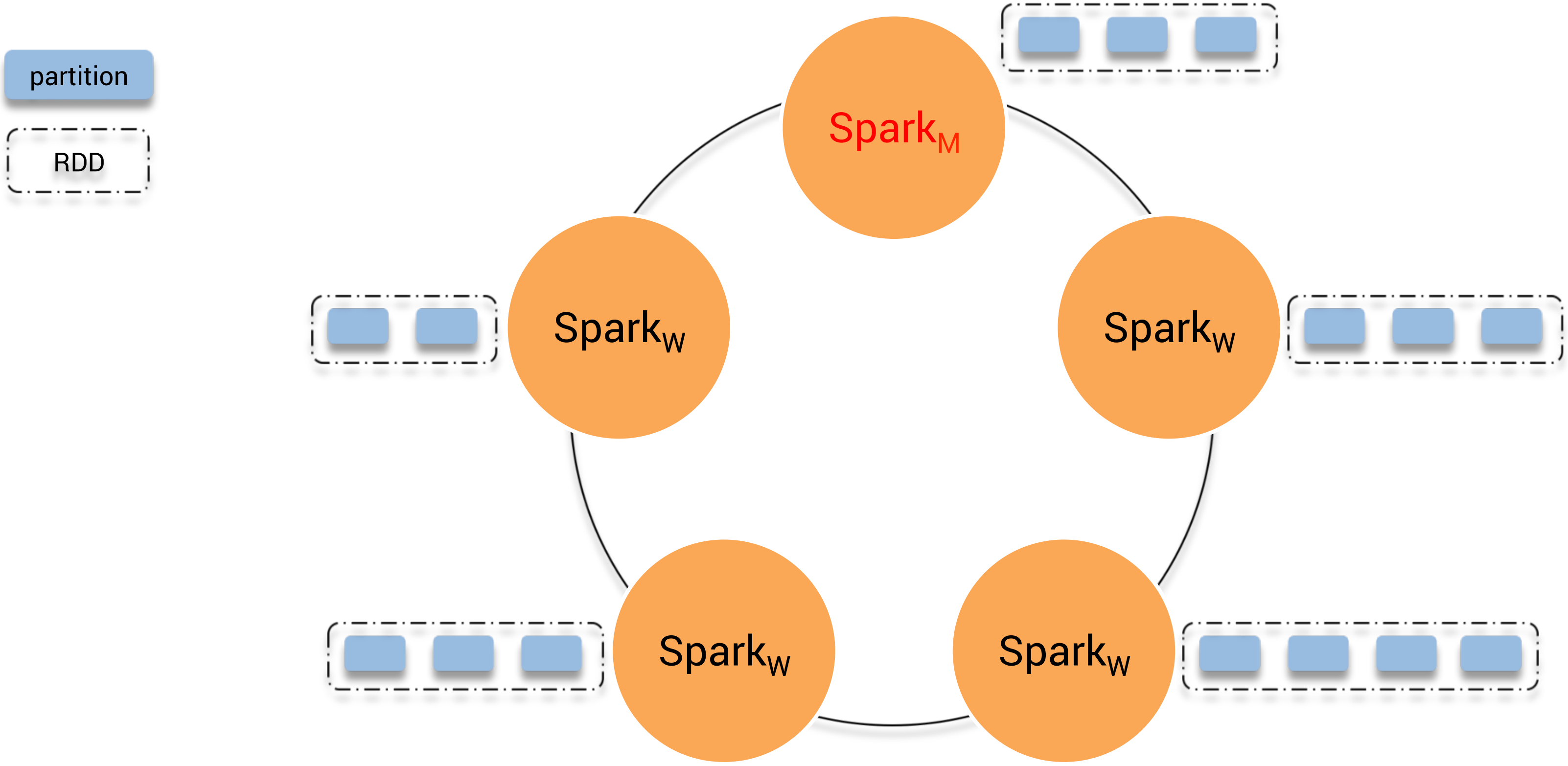
Definition

- chunks for an RDD
- allow to parallelize operations on an RDD
- 1 RDD has [1, n] partitions
- partitions are distributed across Spark workers

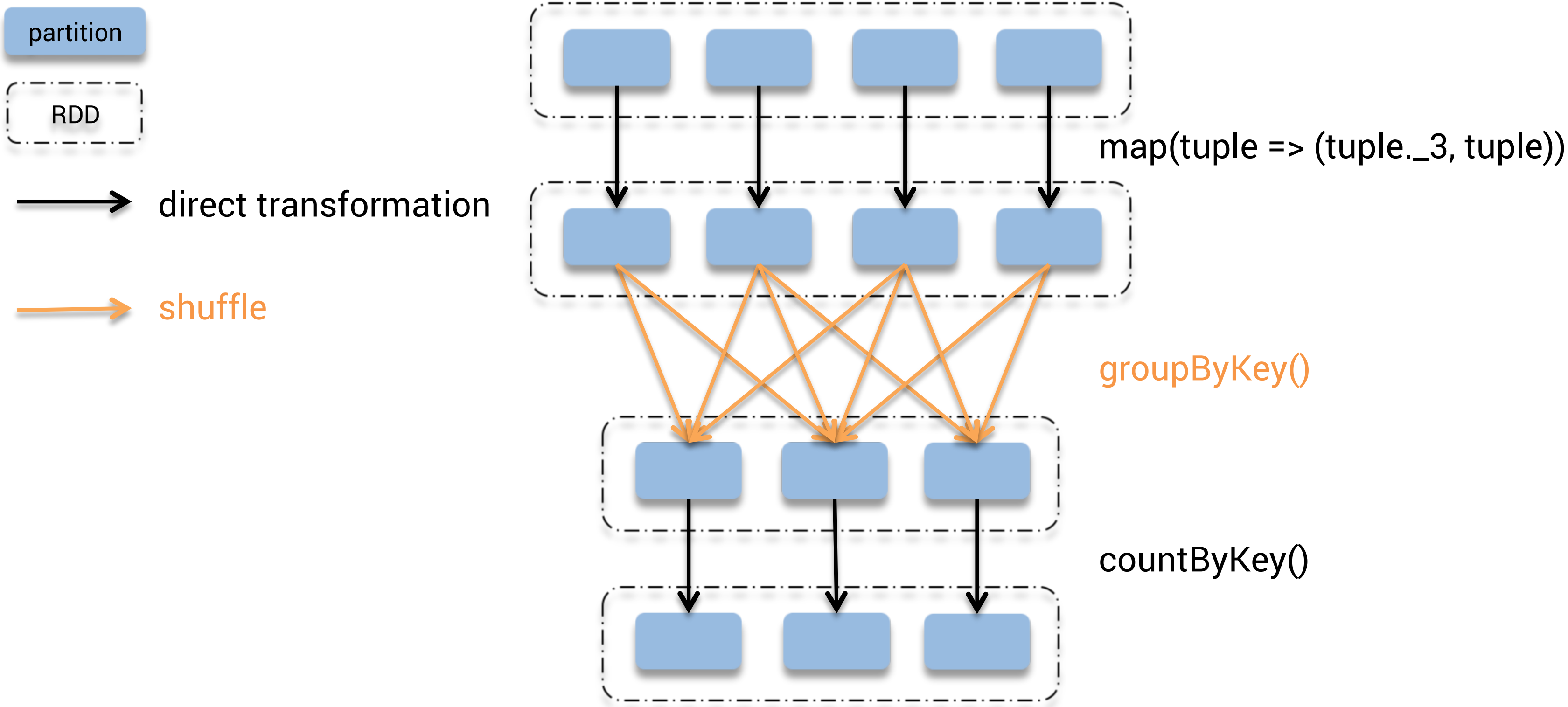
Partionning

- impacts parallelism
- impacts performance

Partitions in the cluster

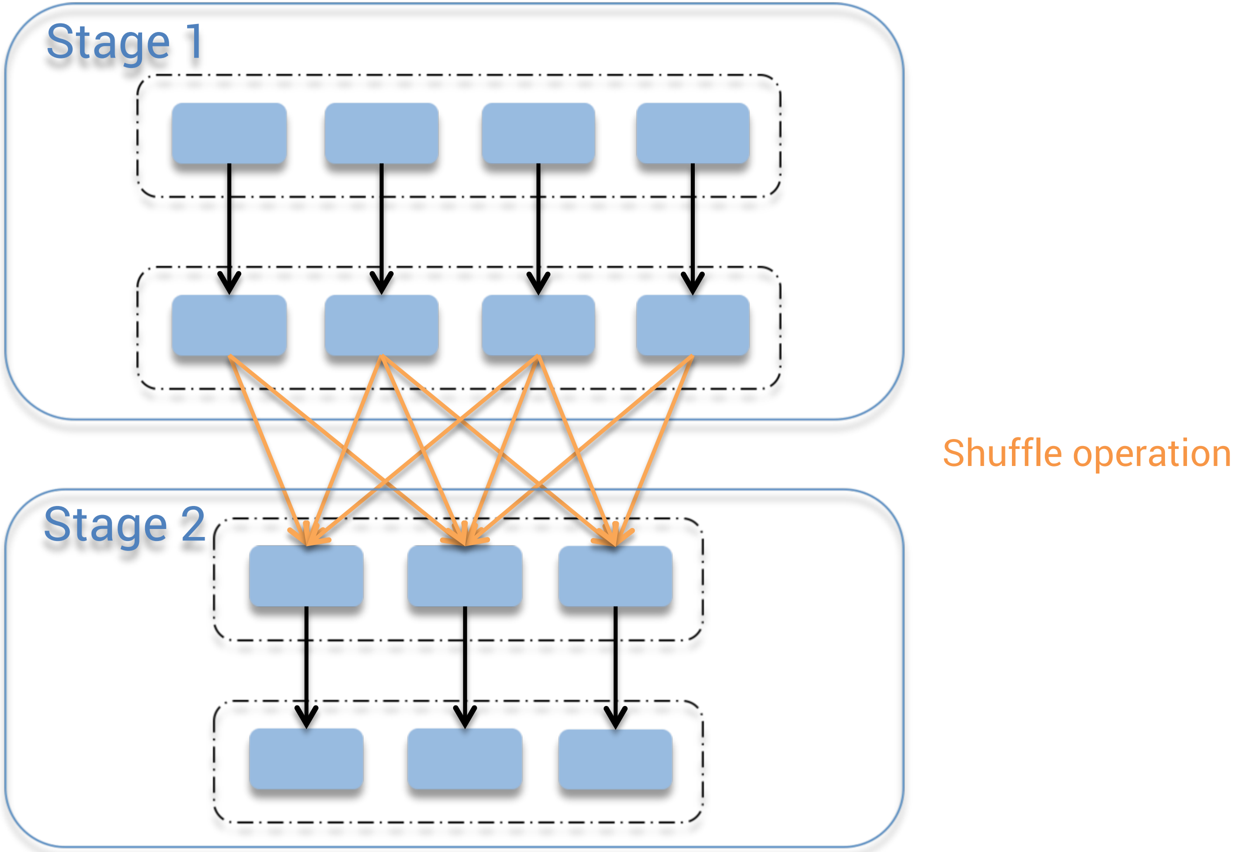


Partitions transformations



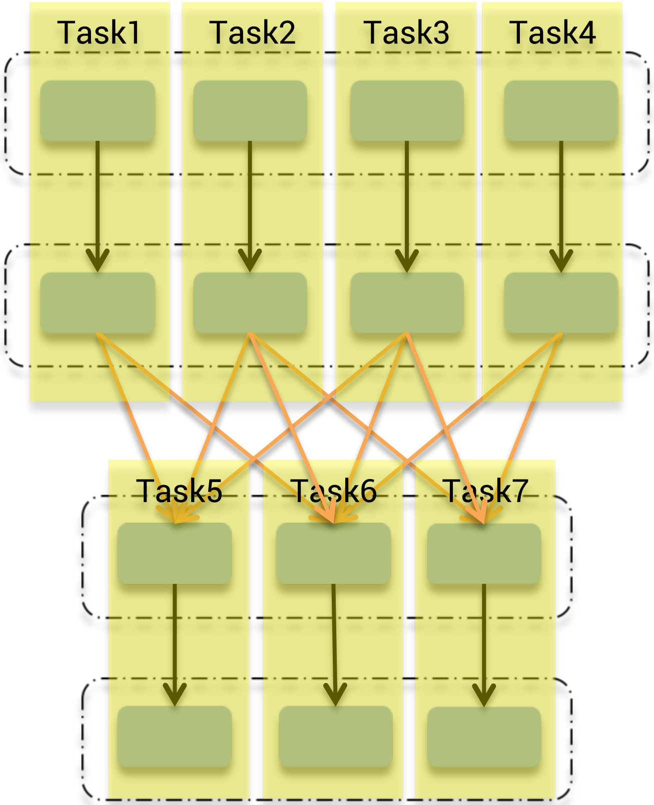
Stages

Delimits "shuffle" frontiers



Tasks

Pipelinable transformations inside a stage



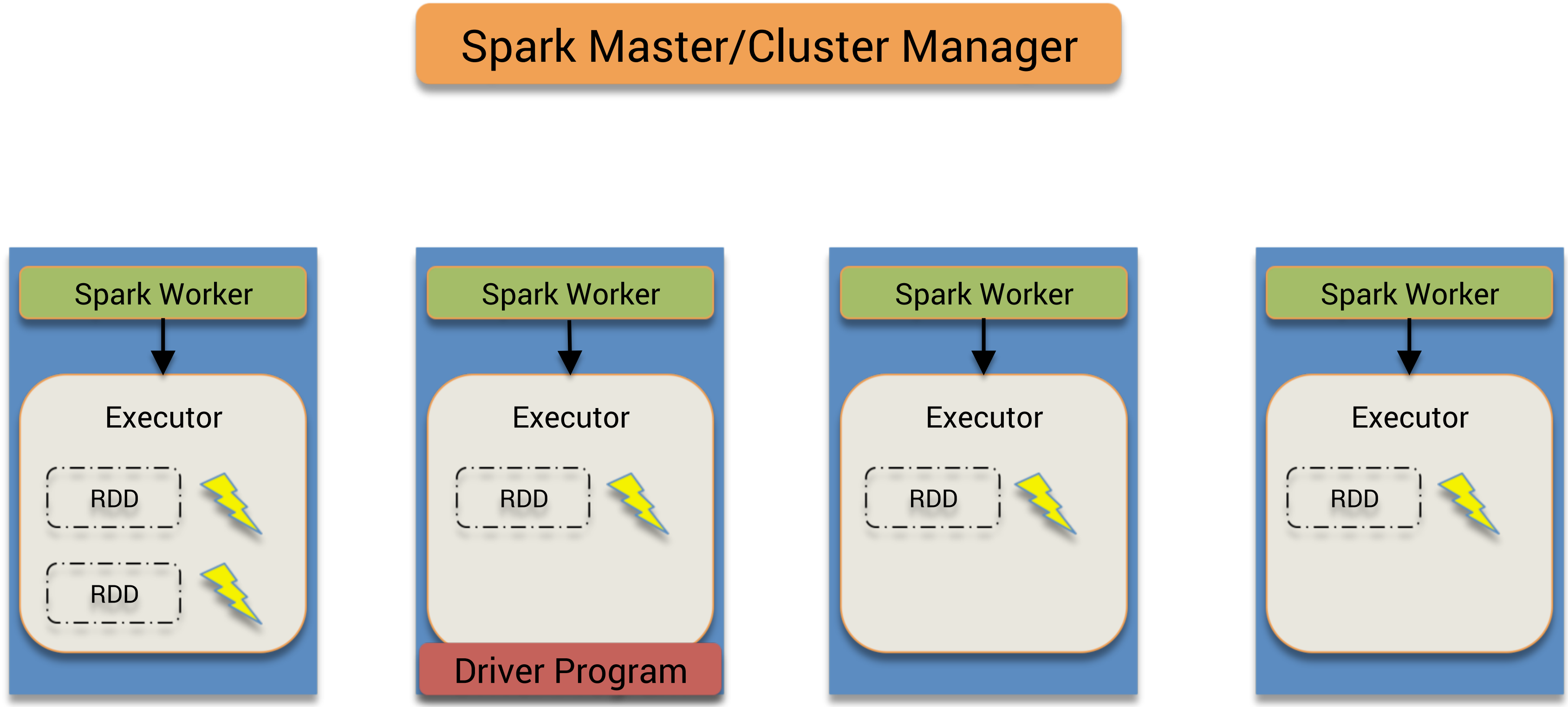
Q & R

Architecture & job life-cycle

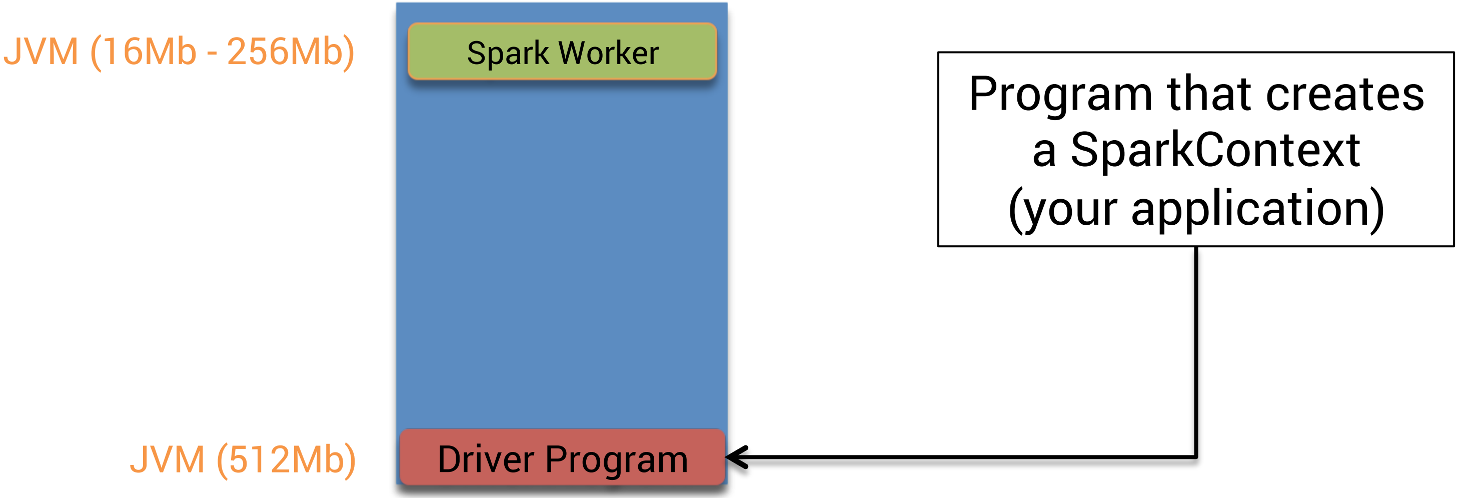
Spark components

Spark job life-cycle

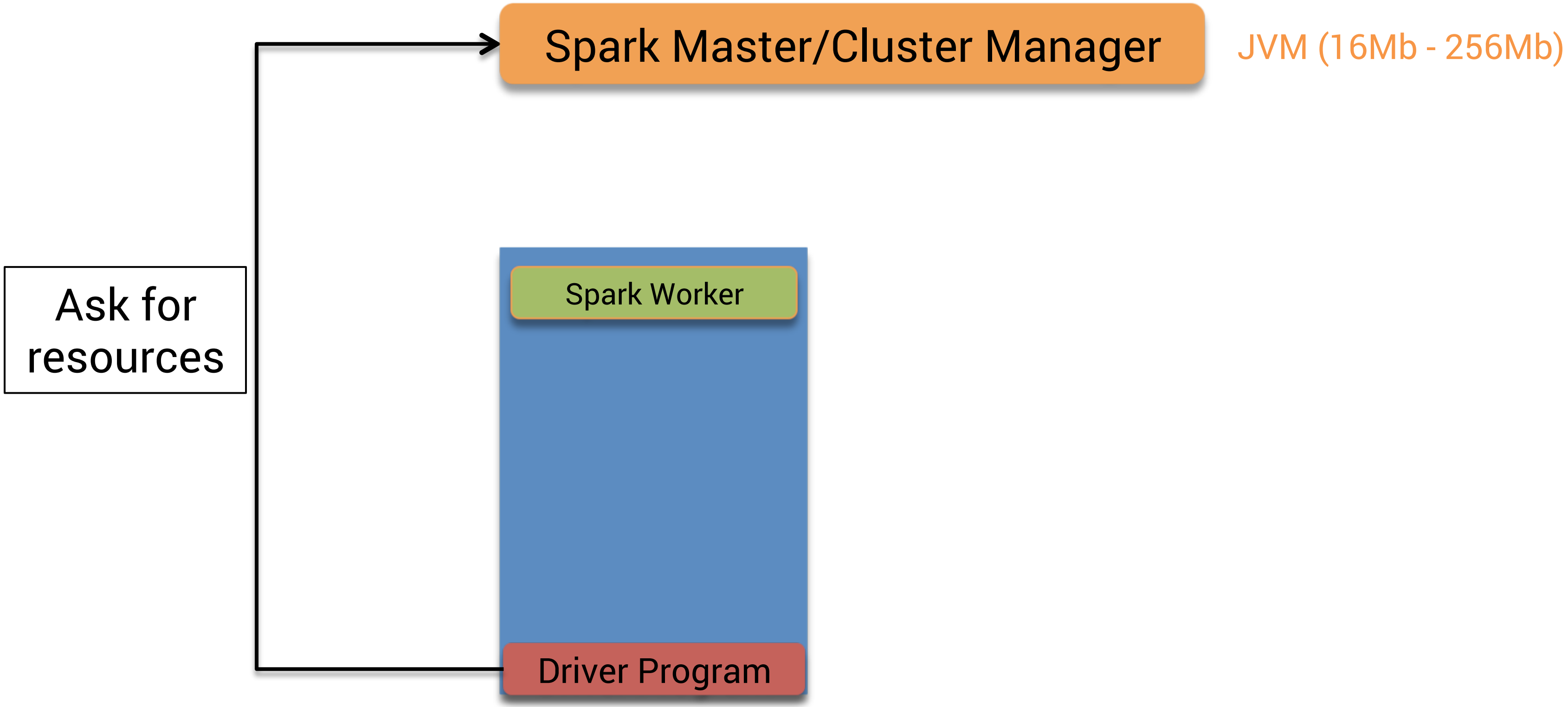
Spark Components



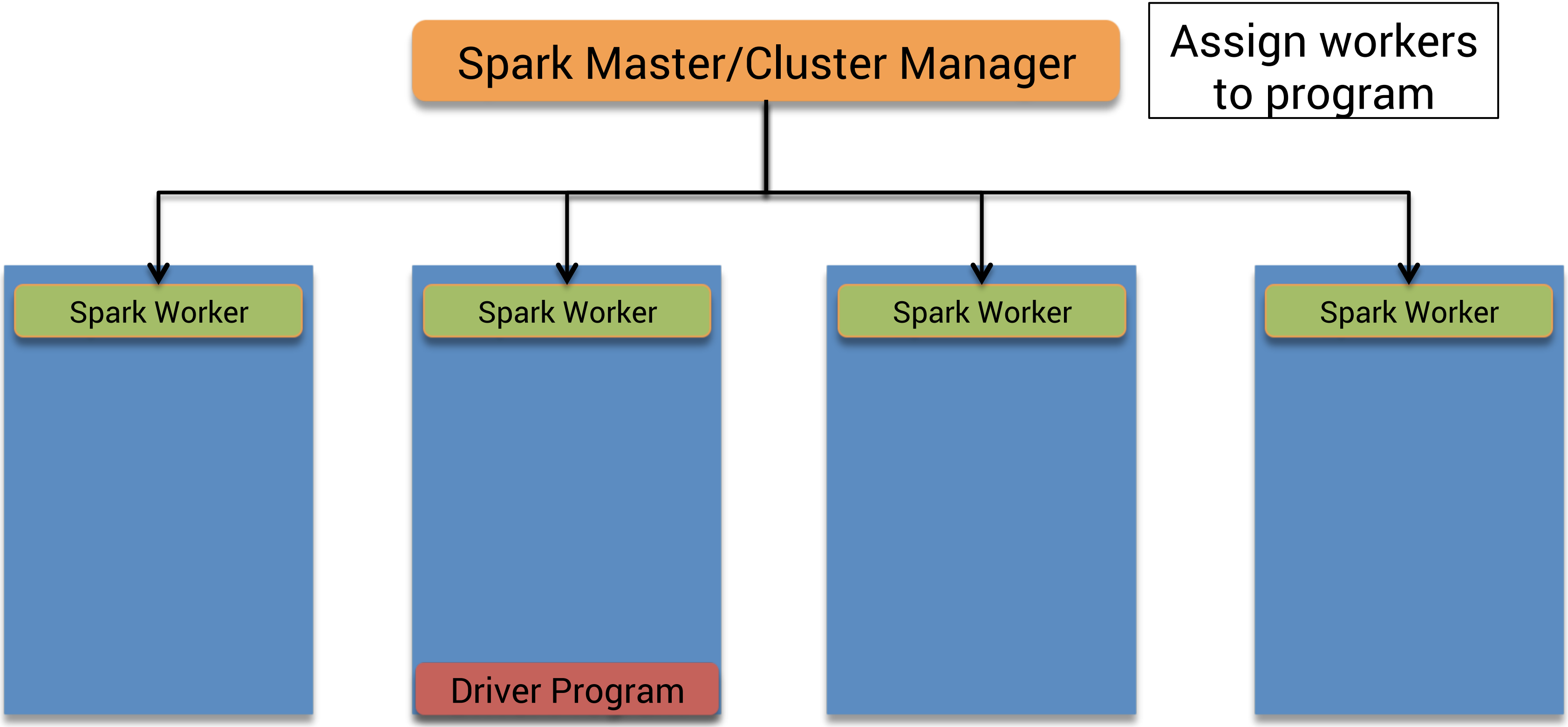
Spark Components



Spark Components

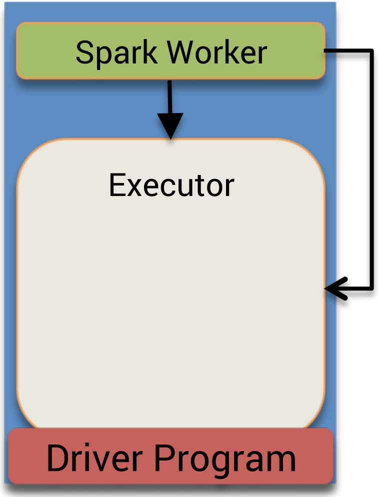


Spark Components



Assign workers to program

Spark Components

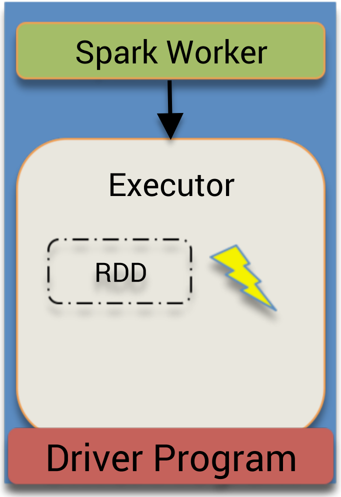


JVM (512Mb + … ≤ ≈ 30Gb)

Worker starts an Executor (JVM)

Spark Components

JVM (512Mb + … ≤ ≈ 30Gb)



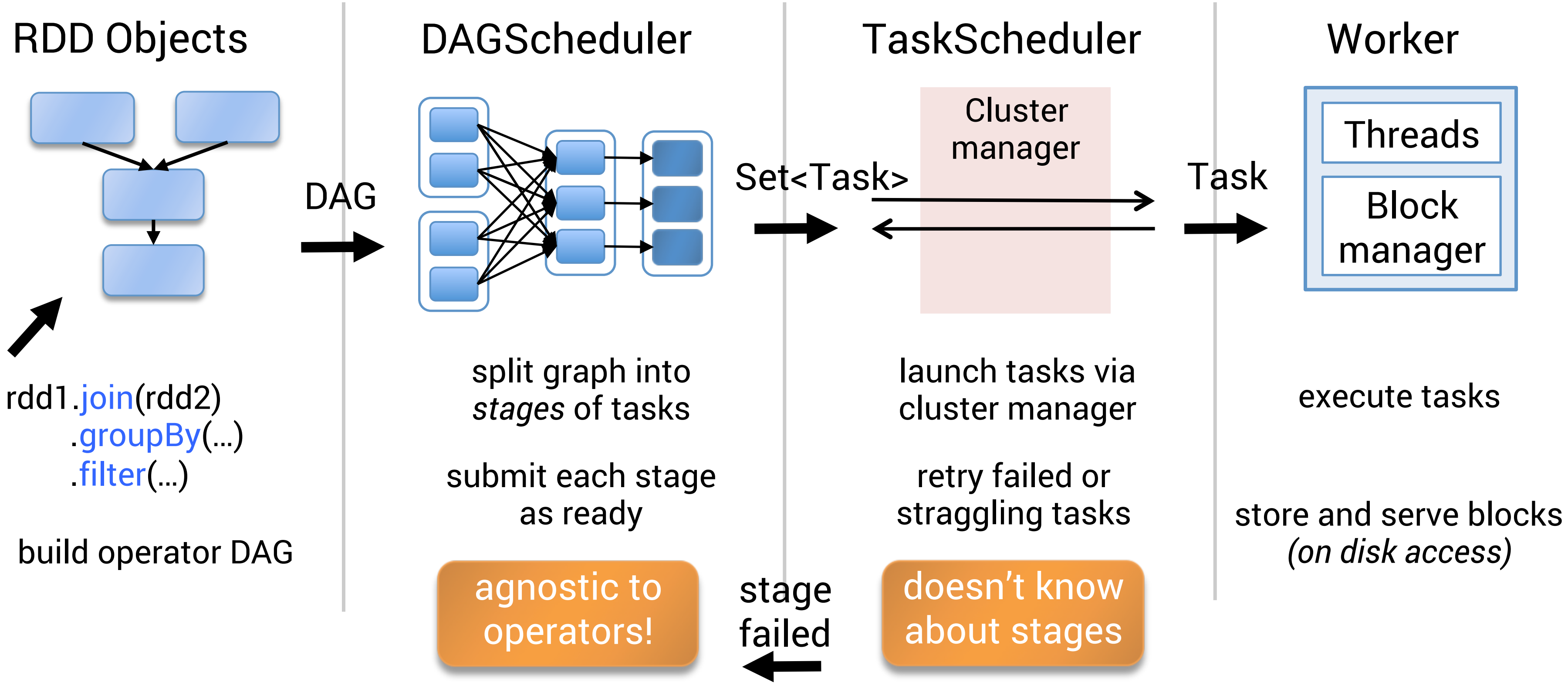
Executor has:

1. thread pool
2. local disk storage

Executor creates:

1. RDD
2. Tasks threads

Spark job life-cycle



RDD Objects

```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

DAGScheduler

split graph into *stages* of tasks

submit each stage as ready

agnostic to operators!

TaskScheduler

launch tasks via cluster manager

retry failed or straggling tasks

doesn't know about stages

Worker

execute tasks

store and serve blocks *(on disk access)*

© Matei Zaharia

Q & R

Spark Core API

Operations

Performance considerations

Proper partitioning

Spark Core API

map

filter

groupBy

sort

union

join

leftOuterJoin

rightOuterJoin

reduce

count

fold

reduceByKey

groupByKey

cogroup

cross

zip

sample

take

first

partitionBy

mapWith

pipe

save

...

+ Scala collection API

...

Direct transformations

Direct transformations	Description
map(f:A => B): RDD[B]	Return a new RDD by applying a function to all elements of this RDD
filter(f:A => Boolean): RDD[A]	Return a new RDD containing only the elements that satisfy a predicate
flatMap(f:A => Seq[B]): RDD[B]	Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results
mapPartitions(f: Iterator[A] => Iterator[B]): RDD[B]	Return a new RDD by applying a function to each partition of this RDD
mapPartitionsWithIndex(f: (Int,Iterator[A]) => Iterator[B]): RDD[B]	Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition
sample(withReplacement: Boolean, fraction: Double, seed: Long): RDD[A]	Return a sampled subset of this RDD
…	…

Transformations with shuffle

Transformations with shuffle	Description
union(RDD[A]: otherRDD): RDD[A]	Return the union of this RDD and another one
intersection(RDD[A]: otherRDD): RDD[A]	Return the intersection of this RDD and another one
distinct(): RDD[A]	Return a new RDD containing the distinct elements in this RDD
groupByKey(numTasks: Int): RDD[(K,V)]	Group the values for each key in the RDD into a single sequence. **Hash-partitions the resulting RDD with the existing partitioner/parallelism level**
reduceByKey(f:(V,V) => V, numTasks: Int): RDD[(K,V)]	Merge the values for each key using an associative reduce function. **Output will be hash-partitioned with numPartitions partitions**
join[W](otherRDD: RDD[(K, W)]): RDD[(K, (V, W))]	Return an RDD containing all pairs of elements with matching keys in \`this\` and \`other\`. Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in \`this\` and (k, v2) is in \`other\`. Performs a hash join across the cluster

Actions

Actions	Description
reduce(f: (T, T) => T): T	Reduces the elements of this RDD using the specified commutative and associative binary operator
collect(): Array[A]	Return an array that contains all of the elements in this RDD
count(): Long	Return the number of elements in the RDD
first(): A	Return the first element in this RDD
take(num: Int): Array[A]	Take the first *num* elements of the RDD
countByKey(): Map[K, Long]	Count the number of elements for each key, and return the result to the master as a Map
foreach(f: A => Unit): Unit	Applies a function f to all elements of this RDD
…	…

Performance considerations

Filter early to minimize memory usage

Fetch only necessary data

Minimize "shuffle" operations

Co-partition data whenever possible

Proper partitioning

Too few partitions

- poor parallelism
- sensitivity to data skew
- memory pressure for groupBy(), reduceByKey()…

Too many partitions

- framework overhead (more CPU for Spark than the job to be done)
- many CPU context-switching

Q & R

Spark SQL

Architecture

SchemaRDD

SQL to RDD translation

General ideas

SQL-like query **abstraction over RDDs**

Introduce schema to raw objects

SchemaRDD = RDD[Row]

Declarative vs imperative data transformations

Let’s the engine optimize the query!

Integration with Spark

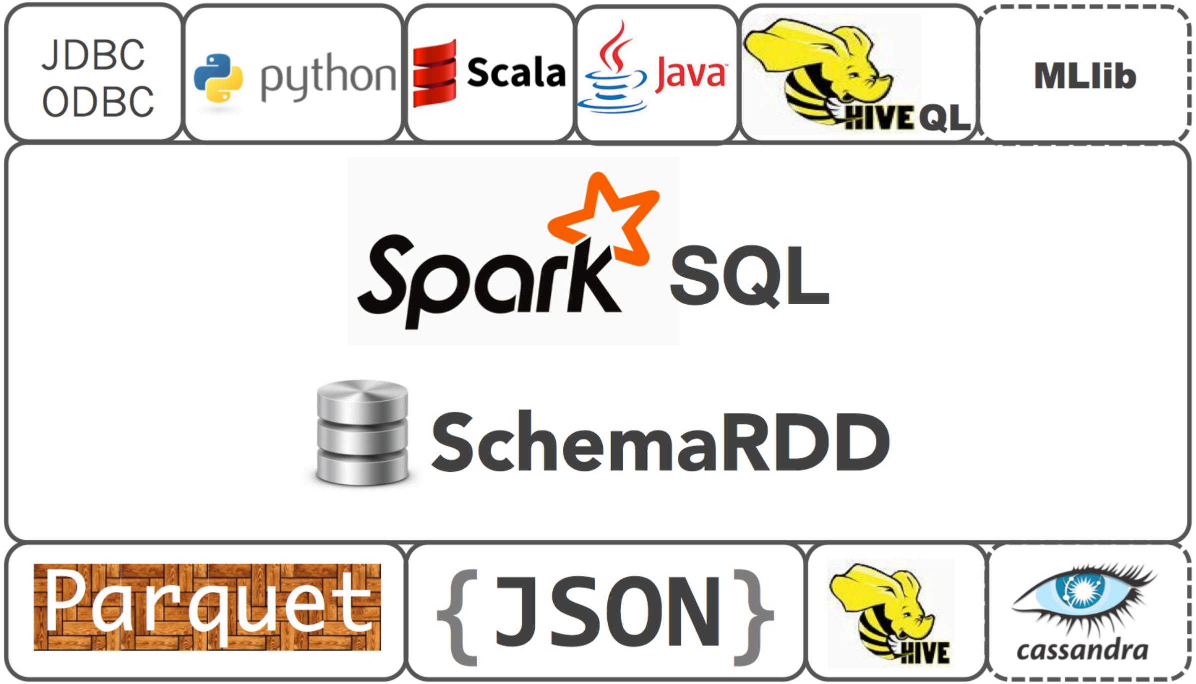
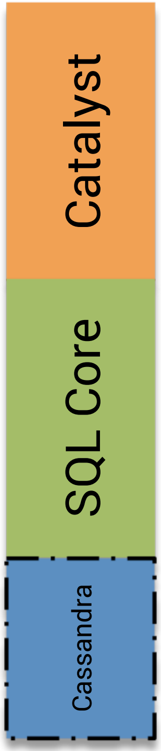


Image credit: http://barrymieny.deviantart.com/

amplab UC BERKELEY databricks™

Architecture



Relational algebra
Query planner, query optimizer

- predicates push-down
- re-ordering

SchemaRDD

- new methods
- schema handling + metadata
- Parquet, JSON integration

Custom Datastax connector

- custom plan for CQL
- predicates push-down for CQL

Code example

Setup

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext.createSchemaRDD
```

Data-set (can be from Parquet, Json, Cassandra, Hive, …)

```
case class Person(name: String, age: Int)

val people = sc.textFile("people.txt").map(_.split(","))
                .map(p => Person(p(0), p(1).trim.toInt))

people.registerTempTable("people")
```

Code example

Query

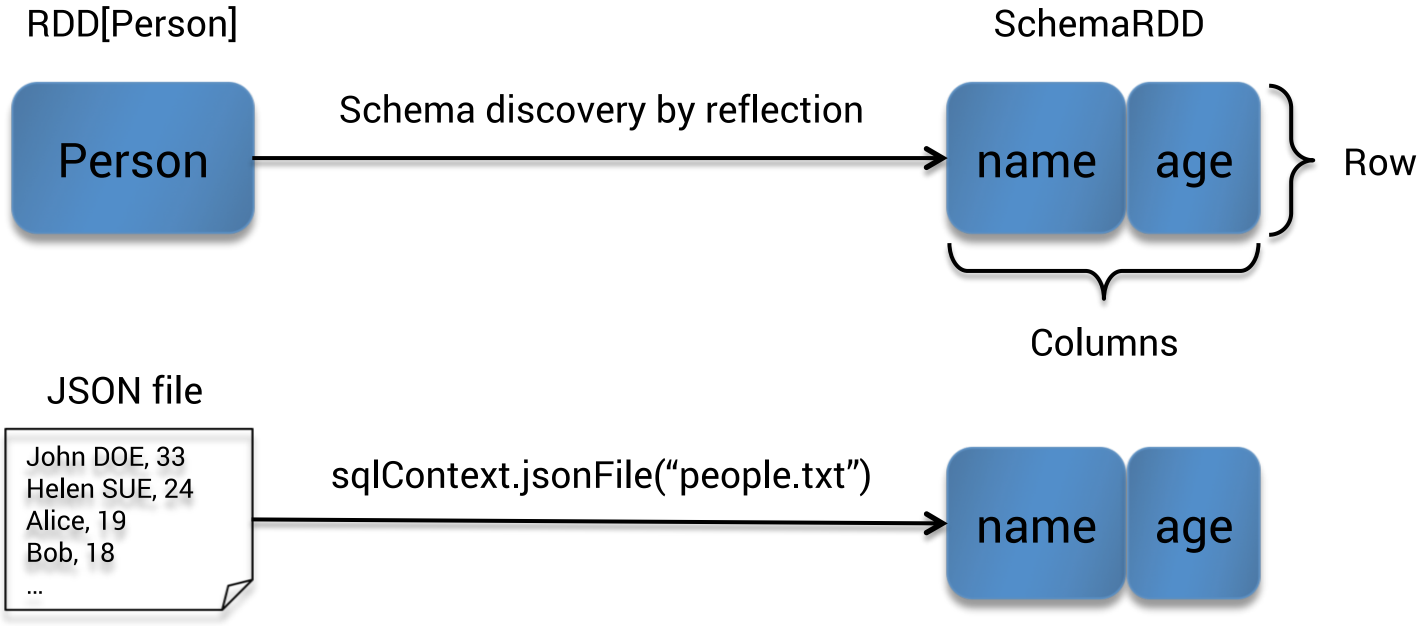
```
val teenagers: SchemaRDD = sqlContext.sql("SELECT name, age
                                           FROM people
                                           WHERE age ≥ 13 AND age ≤ 19");

// or

val teenagers: SchemaRDD = people.where('age ≥ 10).where('age ≤ 19).select('name)

teenages.map(row => "Name : "+row(0)).collect().foreach(println)
```

SchemaRDD



RDD[Person]

SchemaRDD

Person

Schema discovery by reflection

name

age

Row

Columns

JSON file

John DOE, 33
Helen SUE, 24
Alice, 19
Bob, 18
...

sqlContext.jsonFile(“people.txt”)

name

age

SQL to RDD translation

Projection & selection

```
SELECT name, age
FROM people
WHERE age ≥ 13 AND age ≤ 19
```

```
val people:RDD[Person]
val teenagers:RDD[(String,Int)]
    = people
        .filter(p => p.age ≥ 13 && p.age ≤ 19)
        .map(p => (p.name, p.age))
```

```
SELECT name, age
```

```
        .map(p => (p.name, p.age))
```

```
WHERE age ≥ 13 AND age ≤ 19
```

```
        .filter(p => p.age ≥ 13 && p.age ≤ 19)
```

SQL to RDD translation

Joins (naive version)

```
SELECT p.name, e.content,e.date
FROM people p JOIN emails e
ON p.login = e.login
WHERE p.age ≥ 28
AND p.age ≤ 32
AND e.date ≥ ‘2015-01-01 00:00:00’
```

SQL to RDD translation

Joins (naive version)

```
SELECT p.name, e.content,e.date
FROM people p JOIN emails e
ON p.login = e.login
WHERE p.age ≥ 28
AND p.age ≤ 32
AND e.date ≥ ‘2015-01-01 00:00:00’
```

```
val people:RDD[Person]
val emails:RDD[Email]
val p = people.map(p => (p.login, p))
val e = emails.map(e => (e.login,e))
val eavesdrop:RDD[(String,String,Date)]
    = p.join(e)
       .filter{ case(login,(p,e) =>
                p.age ≥ 28 && p.age ≤ 32 &&
                e.date ≥ ‘2015-01-01 00:00:00’
       }
       .map{ case(login,(p,e)) =>
                (p.name,e.content,e.date)
       }
```

SQL to RDD translation

Joins (optimized version, selection & projection push-down)

```
SELECT p.name, e.content,e.date
FROM people p JOIN emails e
ON p.login = e.login
WHERE p.age ≥ 28
AND p.age ≤ 32
AND e.date ≥ ‘2015-01-01 00:00:00’
```

```
val p = people.filter(p =>
                p.age ≥ 28 && p.age ≤ 32
        )
        .map(p => (p.login,p.name))

val e = emails.filter(e =>
                e.date ≥ ‘2015-01-01 00:00:00’
        )
        .map(e => (e.login,(e.content,e.date))

val eavesdrop:RDD[(String,String,Date)]
    = p.join(e)
        .map{case (login,(name,(content,date)) =>
                (name,content,date)
        }
```

Q & R

Spark Streaming

General ideas

Streaming example

Window-based reduce

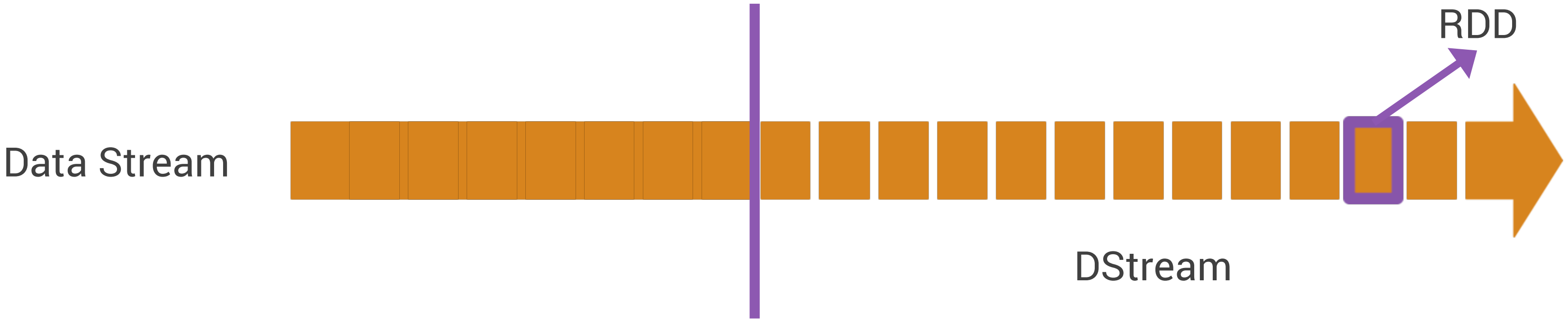
Outputs

General ideas

Micro batching, each batch = RDD

Fault tolerant, exactly-once processing

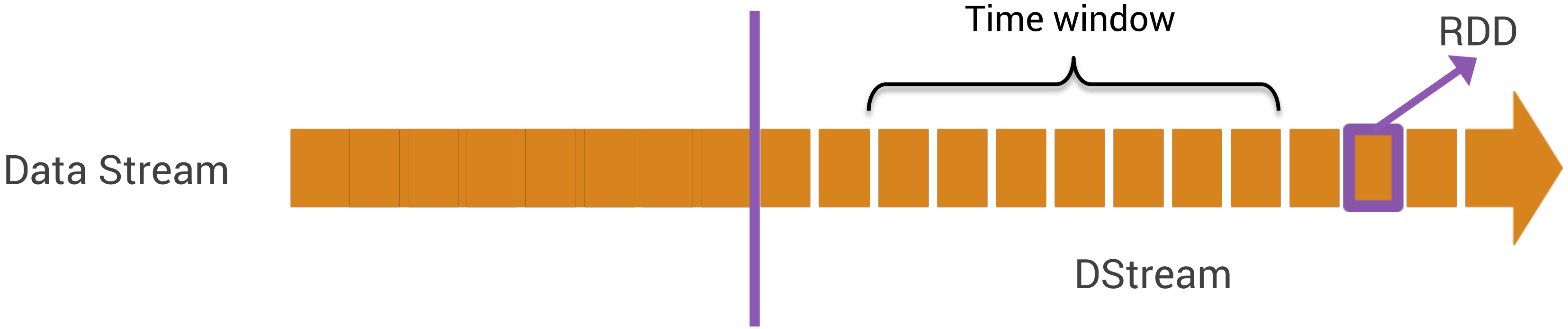
Unified stream and batch processing framework



General ideas

Base unit = time window

Enable computations by time window



Streaming Example

Set up

```
val ssc = new SparkStreamContext("local", "test")
ssc.setBatchDuration(Seconds(1))
```

Words stream

```
val words = ssc.createNetworkStream("http://...")
val ones = words.map(w => (w, 1))
val freqs = ones.reduceByKey{ case (count1, count2) => count1 + count2}

freqs.print()

// Start the stream computation
ssc.run
```

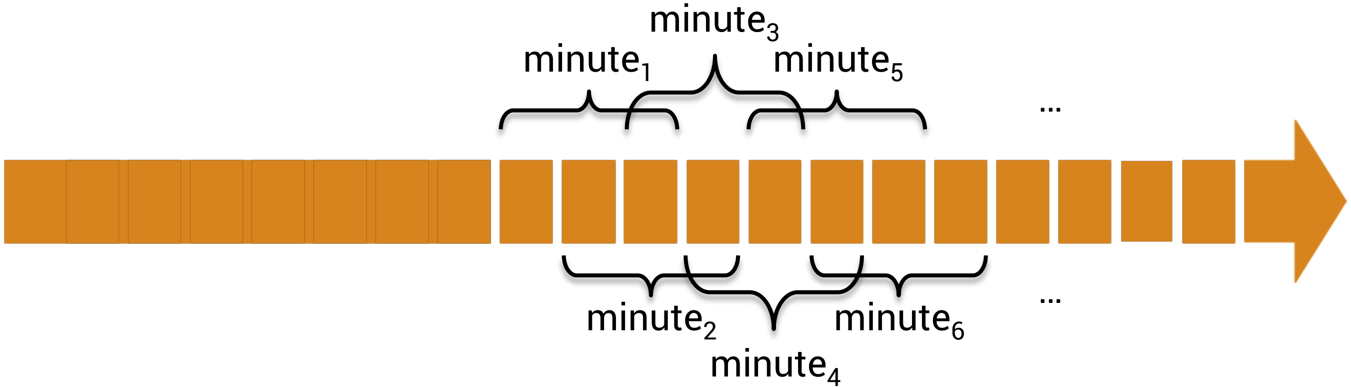
Window-based reduce

```
val freqs = ones.reduceByKey { case (count1, count2) => count1 + count2}
val freqs_60s = freqs.window(Seconds(60), Second(1))
                .reduceByKey { case (count1, count2) => count1 + count2}

// or

val freqs_60s = ones.reduceByKeyAndWindow(couple => couple._1+couple._2,
                                        Seconds(60), Seconds(1))
```

Window-based reduce



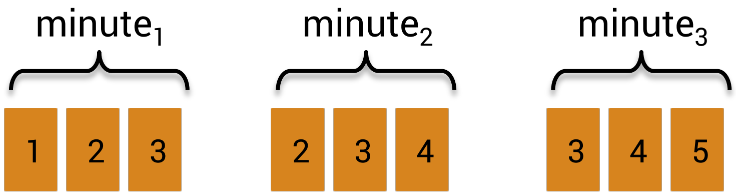
- keeps whole time window RDDs in memory (automatic caching)
- compute sum using all RDDs in the time windows

Optimized window-based reduce

```
val freqs_60s = ones
                .reduceByKeyAndWindow(couple =>
                                couple._1+couple._2, // reduce
                                couple => couple._1 - couple._2, //inverse reduce
                                Seconds(60), Seconds(1))
```

Optimized window-based reduce

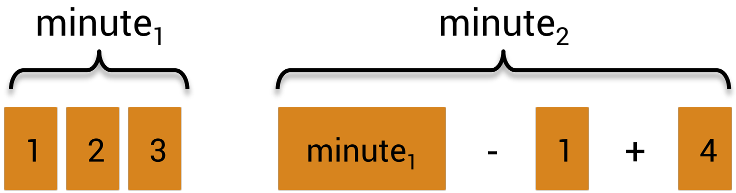
Non optimized computation



$minute_1$ $minute_2$ $minute_3$

1 2 3 2 3 4 3 4 5

Optimized with inverse reduce



$minute_1$ $minute_2$

1 2 3 $minute_1$ - 1 + 4

$inverseReduce(minute_1,RDD_1).reduce(minute_1,RDD_4)$

Outputs

Actions	Description
print(): Unit	Prints first ten elements of every batch of data in a DStream on the driver. This is useful for development and debugging
saveAsTextFiles(prefix: String, suffix: String= ""): Unit	Save this DStream's contents as a text files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]"
saveAsHadoopFiles(prefix: String, suffix: String= ""): Unit	Save this DStream's contents as a Hadoop files. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]"
foreachRDD(f: RDD[A] = Unit): Unit	Apply a function to each RDD in this DStream
…	…

Q & R

Thank You

@doanduyhai

duy_hai.doan@datastax.com

https://academy.datastax.com/