Getting Started with Spark 2

UNDERSTANDING DIFFERENCES BETWEEN SPARK 2.X AND SPARK 1.X

Janani Ravi
CO-FOUNDER, LOONYCORN
www.loonycorn.com

Overview

Spark 1.x was already a great general purpose computing engine

Spark 2.0 takes it to a new level in several ways

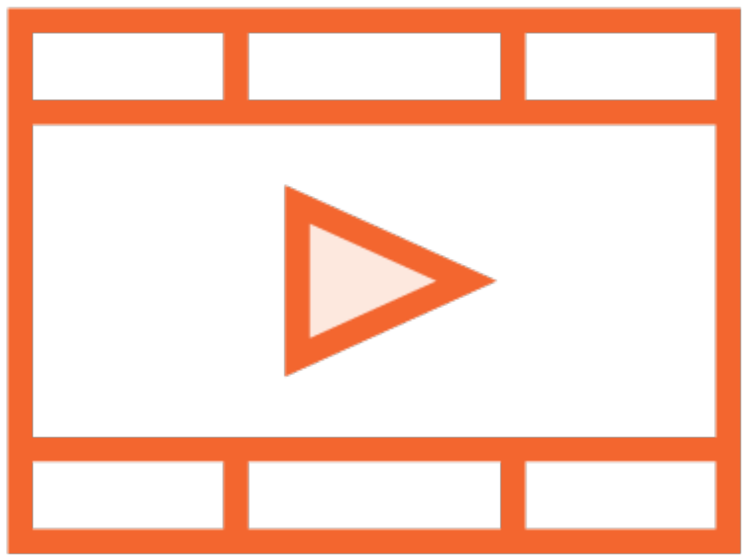
2nd generation Tungsten engine provides 10X performance improvement

Unified APIs for Datasets and DataFrames and Spark SQL

Higher level ML APIs

Unified batch and streaming queries

Prerequisites and Course Outline



Prerequisite Courses

Python: Getting Started

Python Fundamentals

Advanced Python

Related Courses

Beginning Data Exploration and Analysis with Apache Spark

- Programming in Spark 1.x using Python

Handling Fast Data with Apache Spark SQL and Streaming

- Programming in Spark 2 using Scala

Software and Skills

Be very comfortable programming in Python (Python 3)

Be comfortable working with Jupyter notebooks

Understand basics of distributed computing

Course Outline

Spark 1.x vs. Spark 2.x

- Architecture overview, representing structured data as DataFrames
- SparkContext, SQLContext

Exploring and Analyzing Data with DataFrames

- Transformations and actions, built-in aggregations, sampling, grouping, sorting data
- Accumulators and broadcast variables

Querying Data Using Spark SQL

- SQL queries on DataFrames, temporary views, windowing operations

Introducing Spark

Hadoop

HDFS	MapReduce	YARN
A file system to manage the storage of data	A framework to define a data processing task	A framework to run the data processing task

Co-ordination Between Hadoop Blocks

MapReduce

User defines map and reduce tasks using the MapReduce API

YARN

HDFS

Co-ordination Between Hadoop Blocks

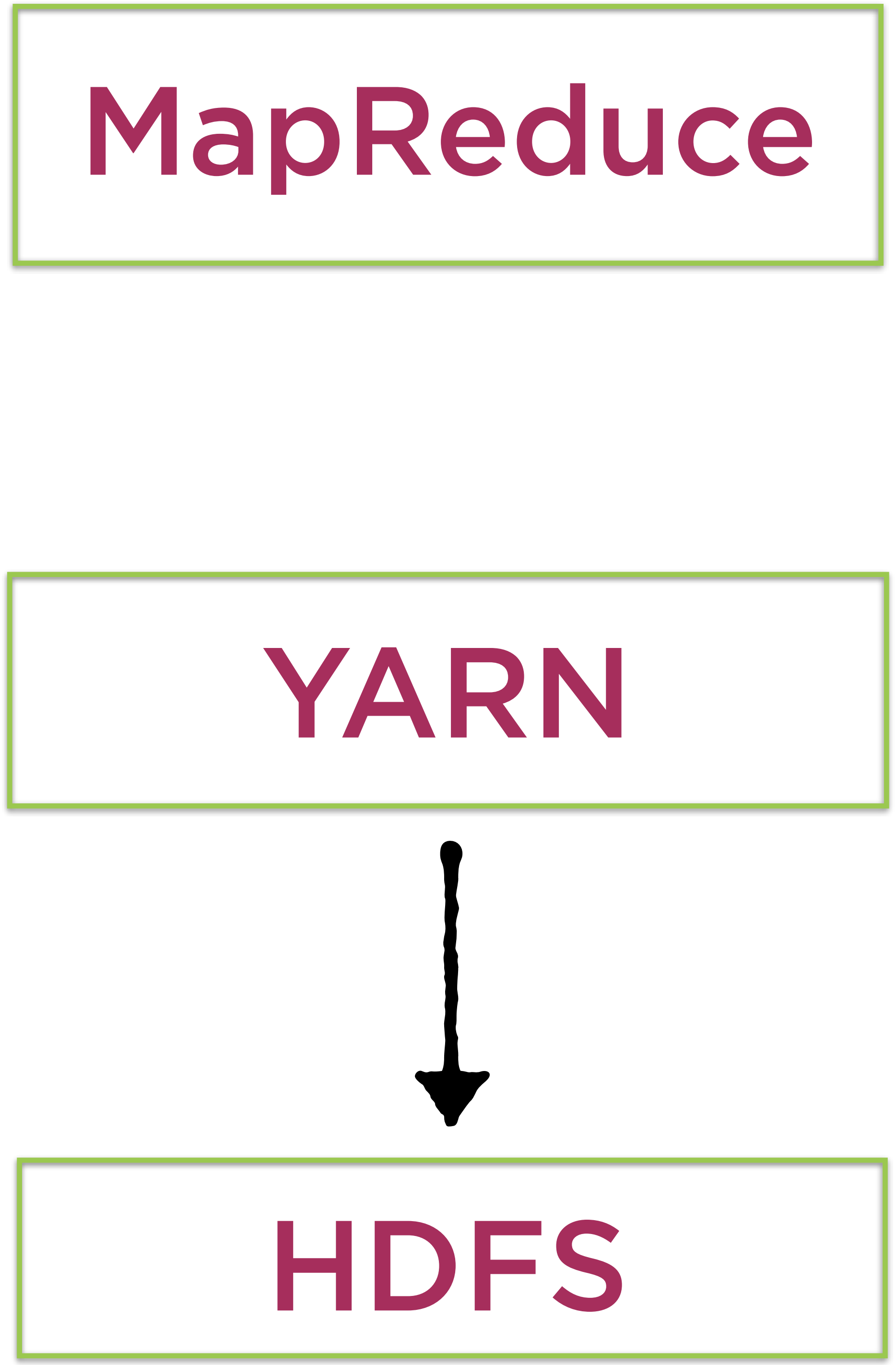
MapReduce

YARN

HDFS

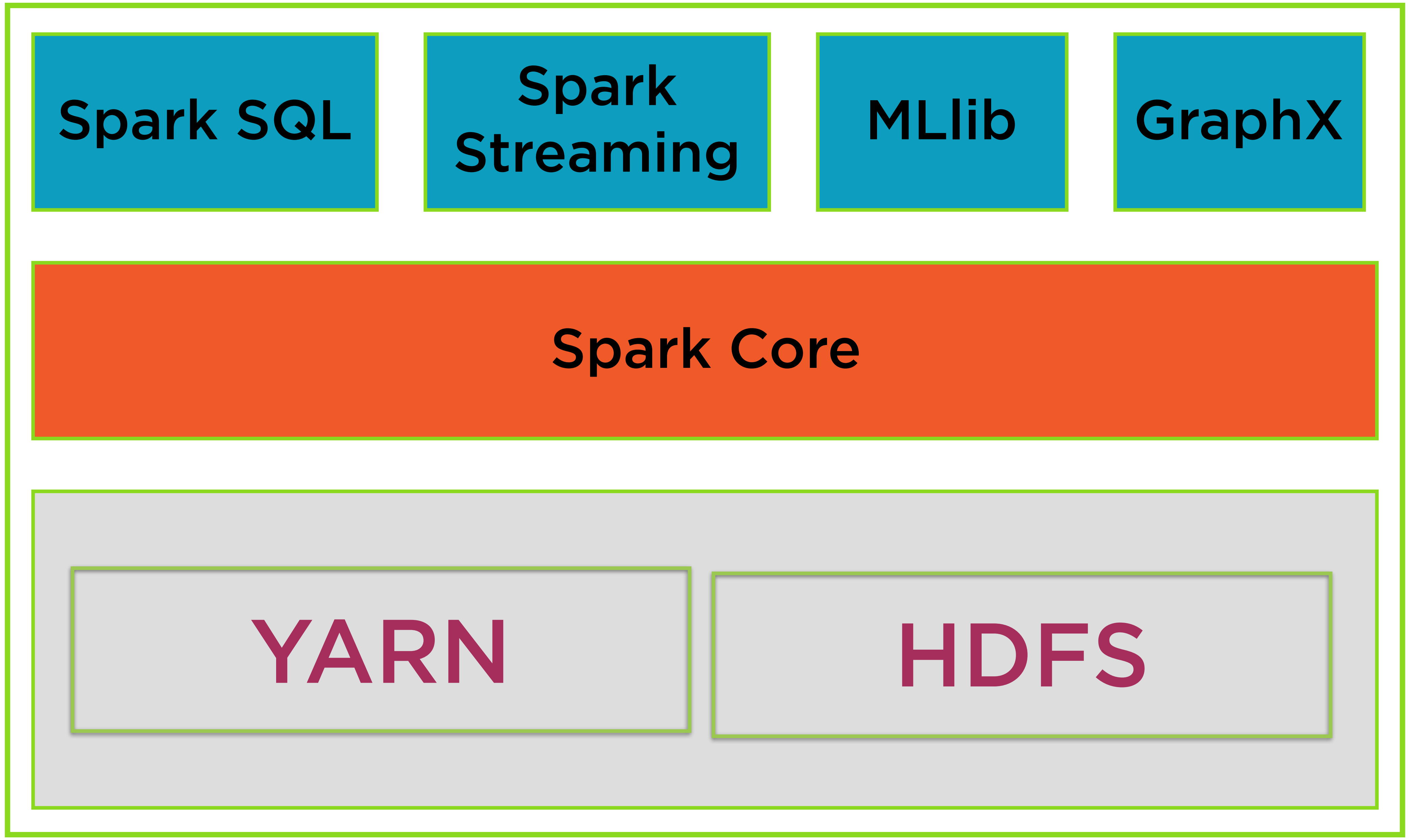
A job is triggered on the cluster

Co-ordination Between Hadoop Blocks

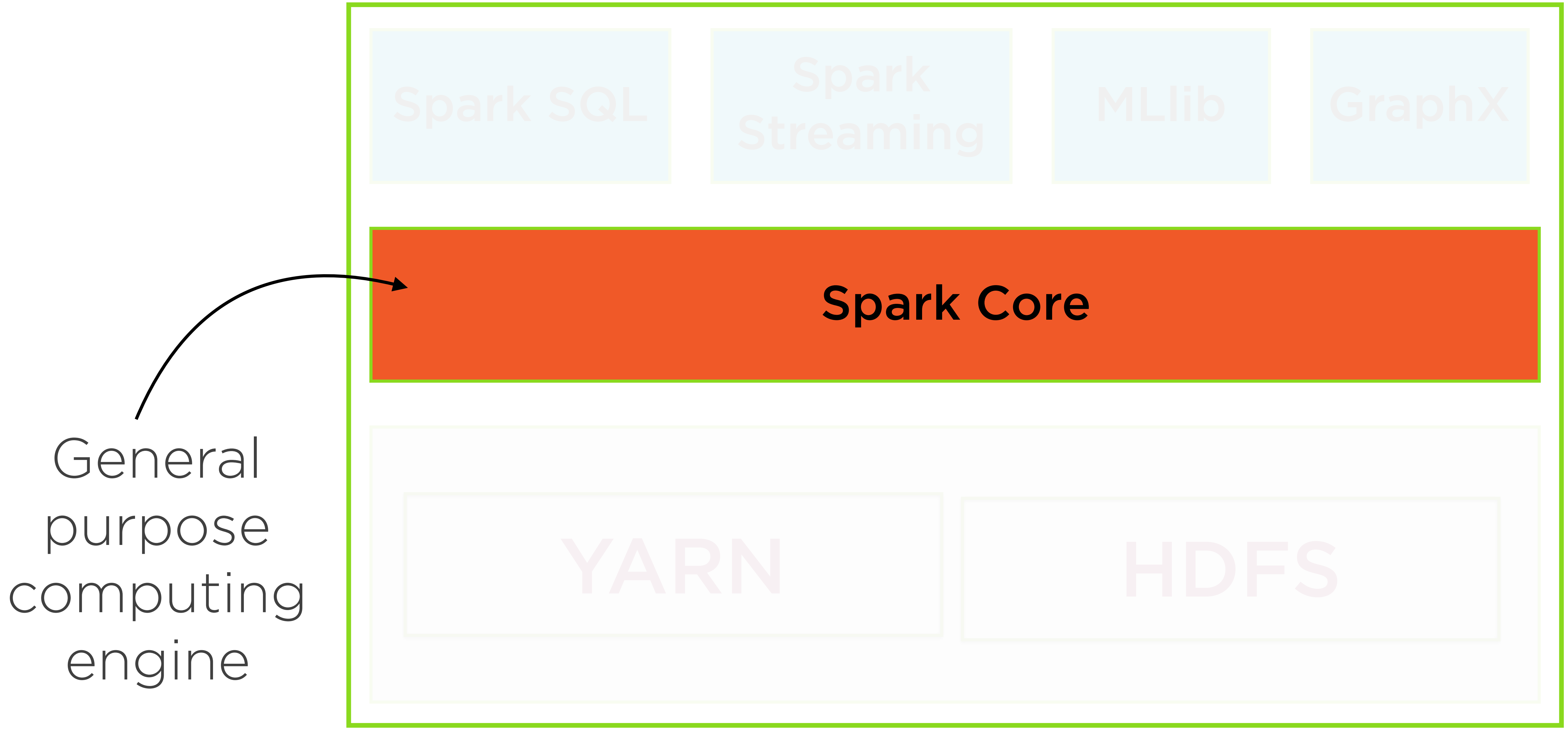


YARN figures out where and how to run the job, and stores the result in HDFS

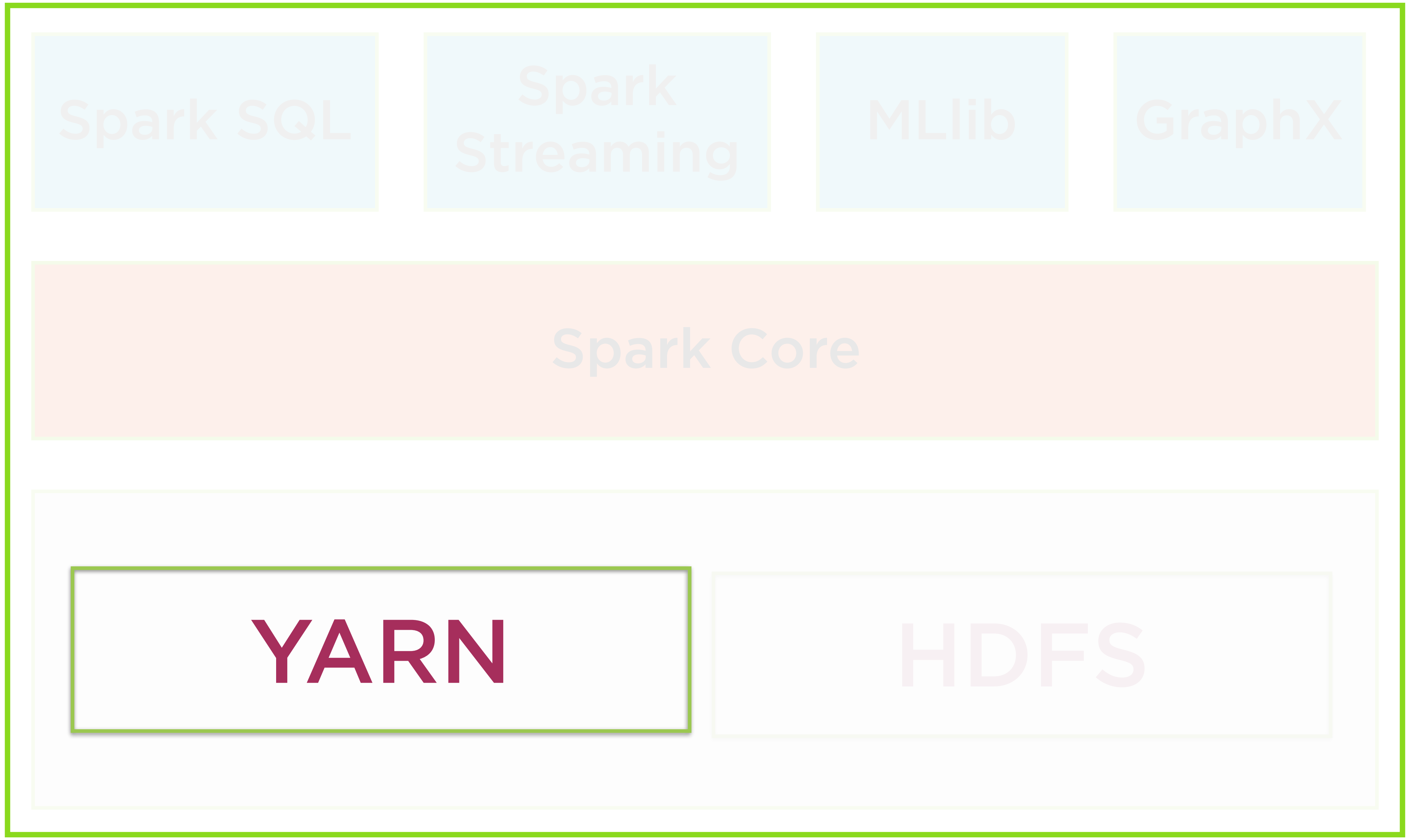
Apache Spark



Apache Spark

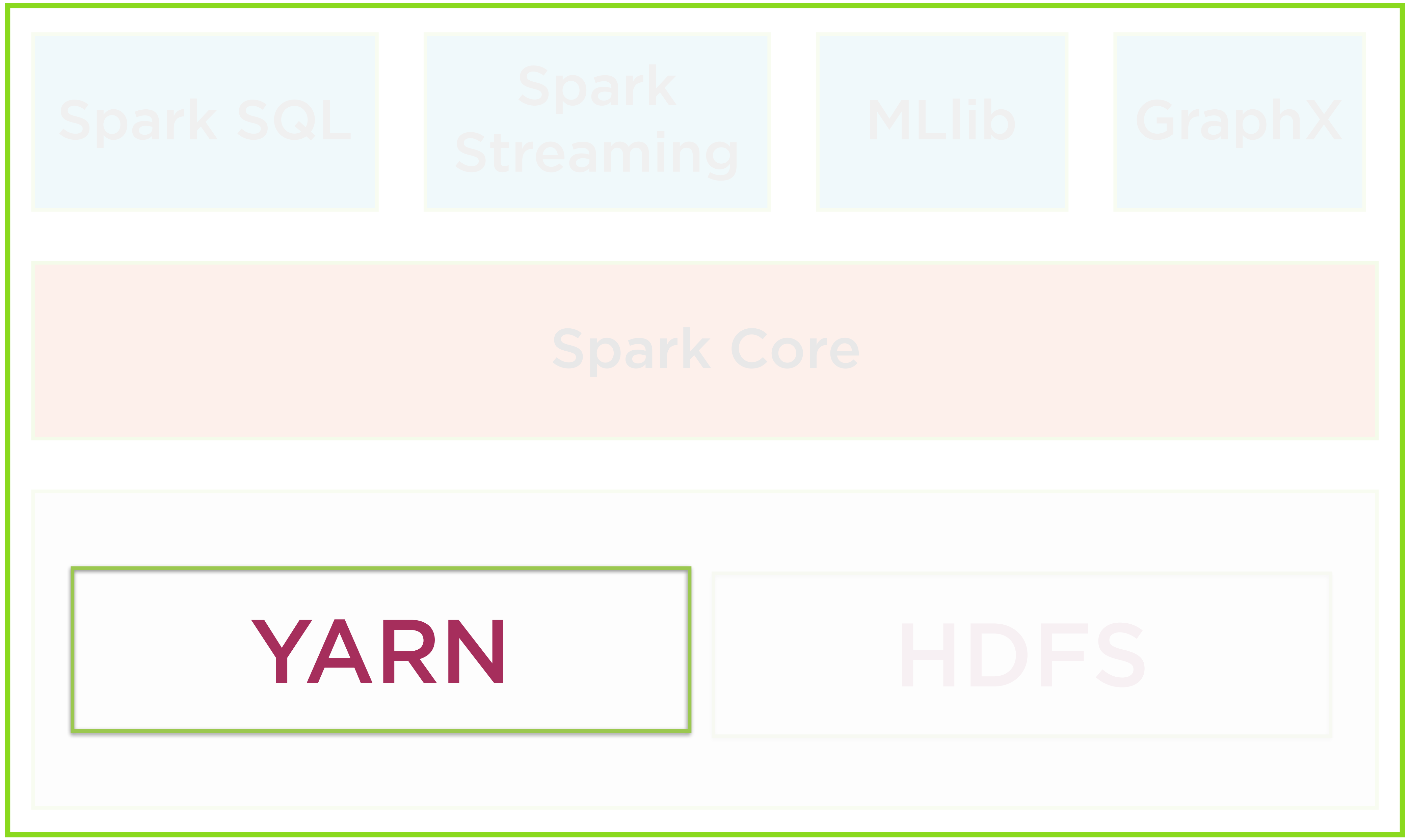


Apache Spark



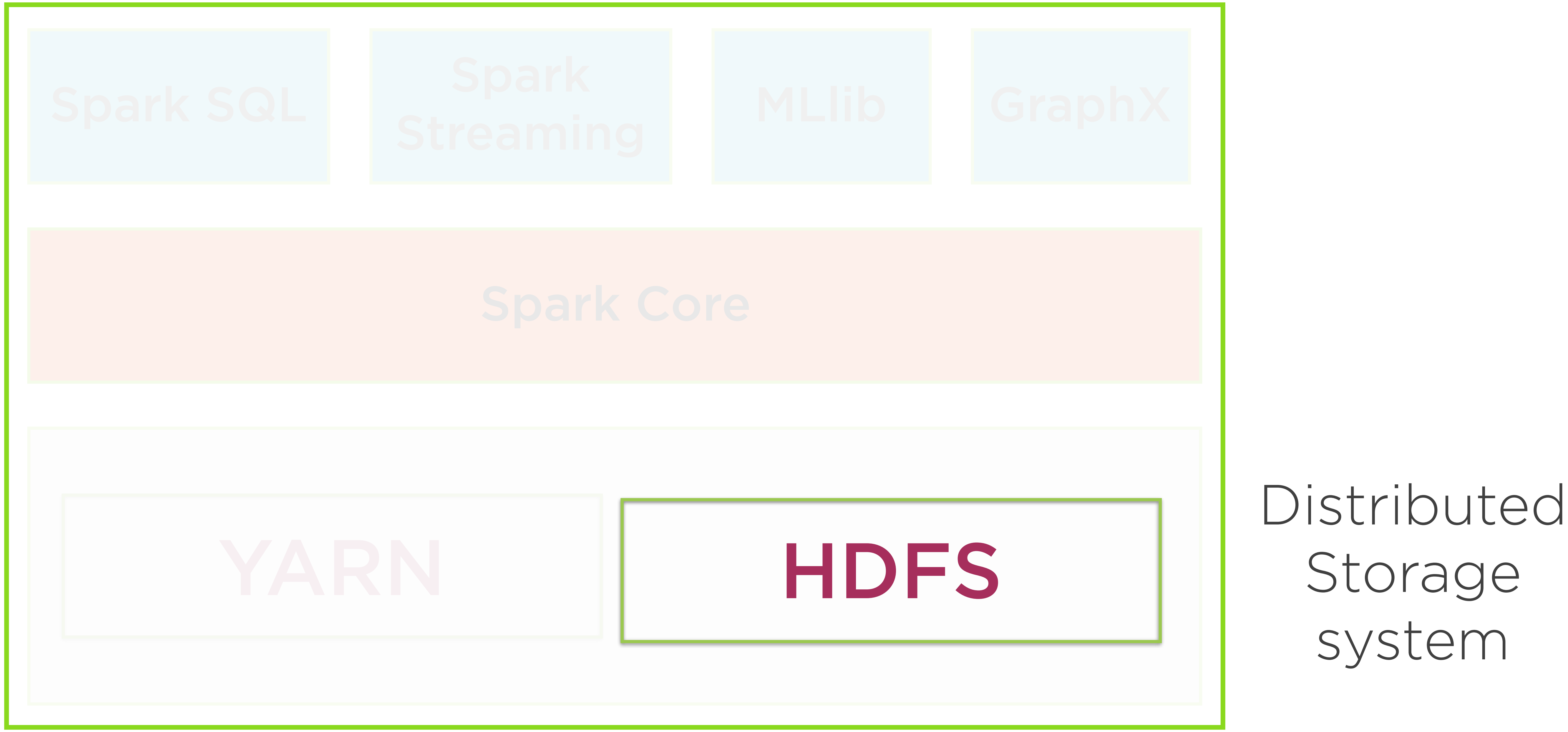
Cluster manager

Apache Spark

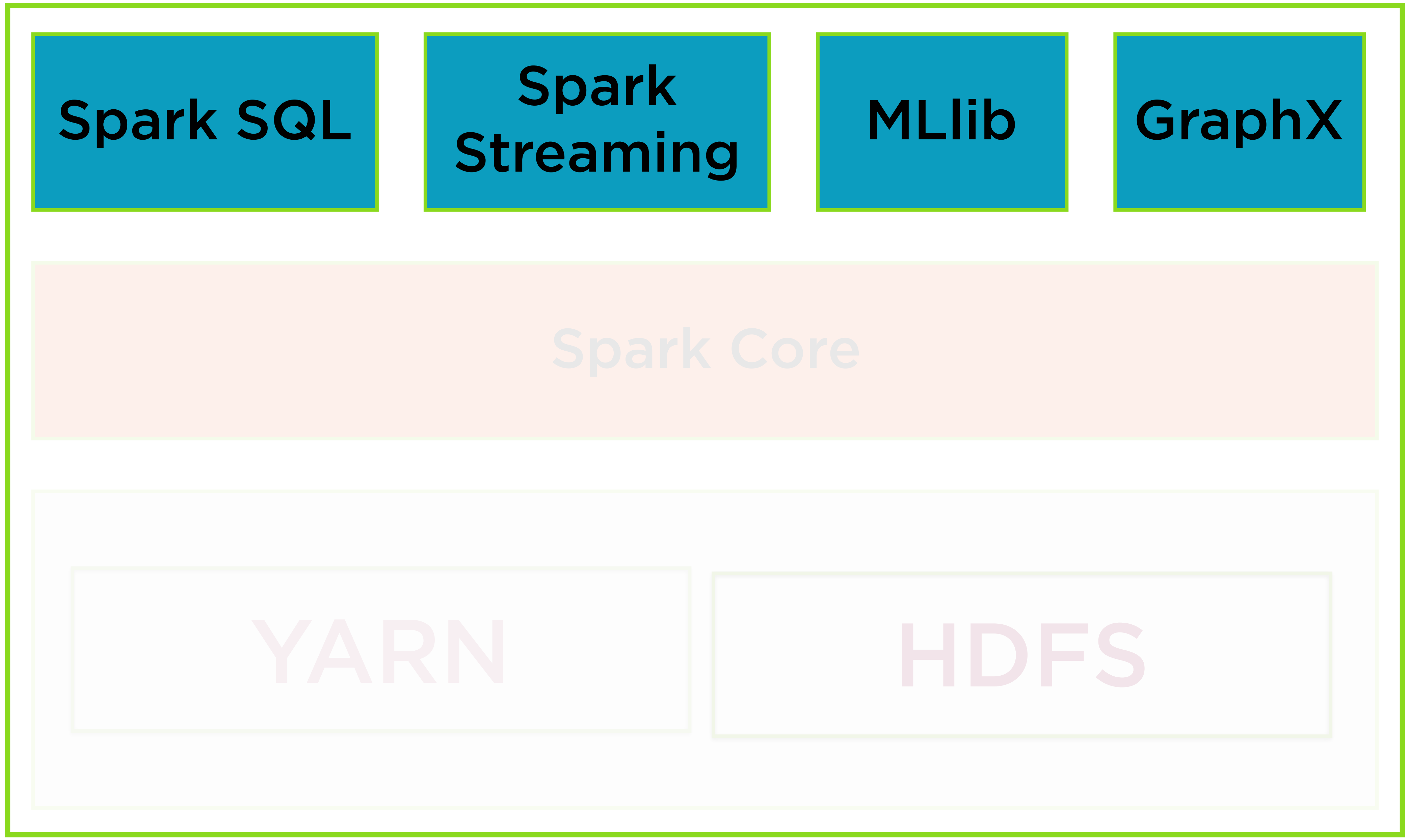


Alternatives: Mesos, Spark Standalone

Apache Spark

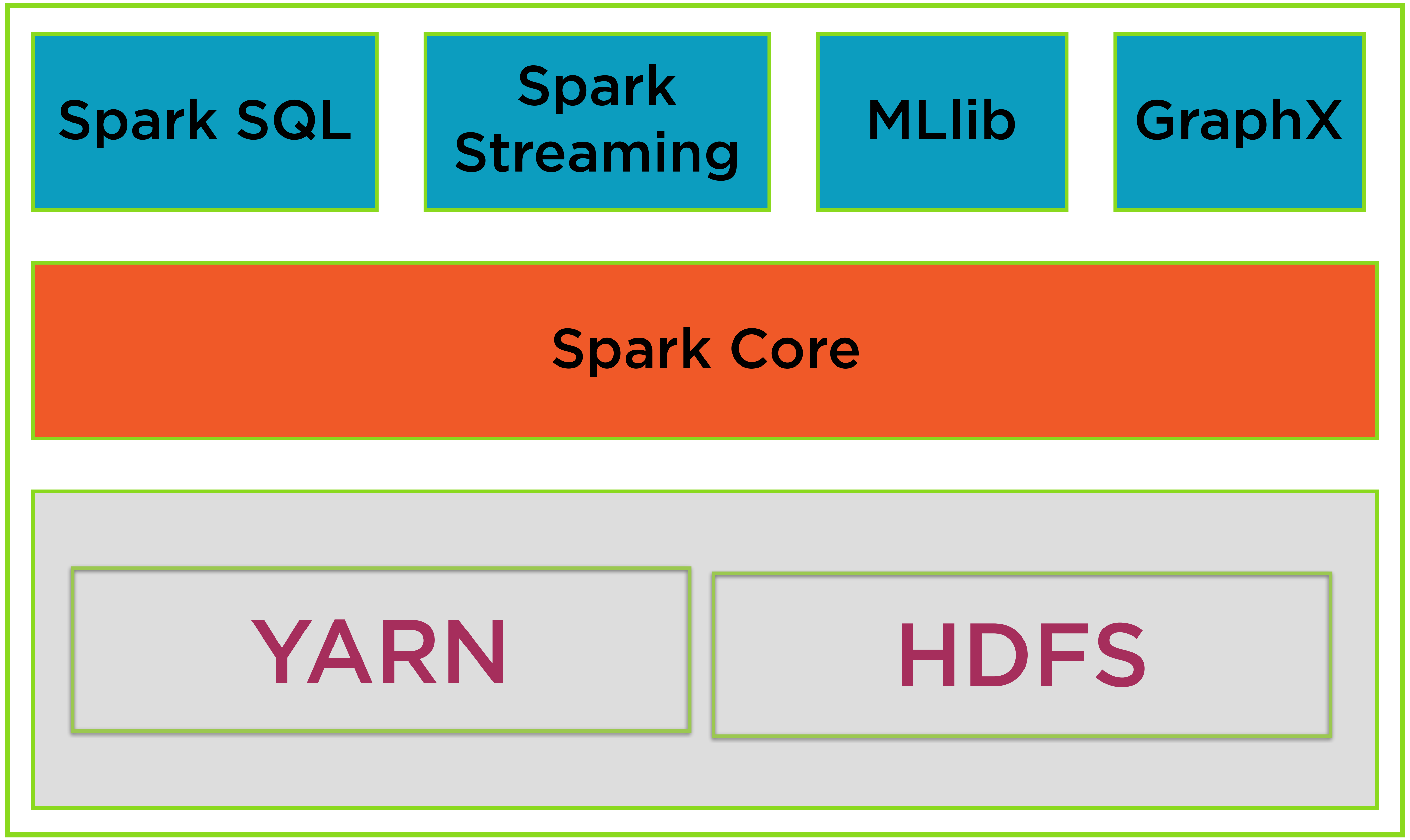


Apache Spark



Spark libraries

Apache Spark



Apache Spark

Real-time as well as batch

Interactive REPL environment

Support for

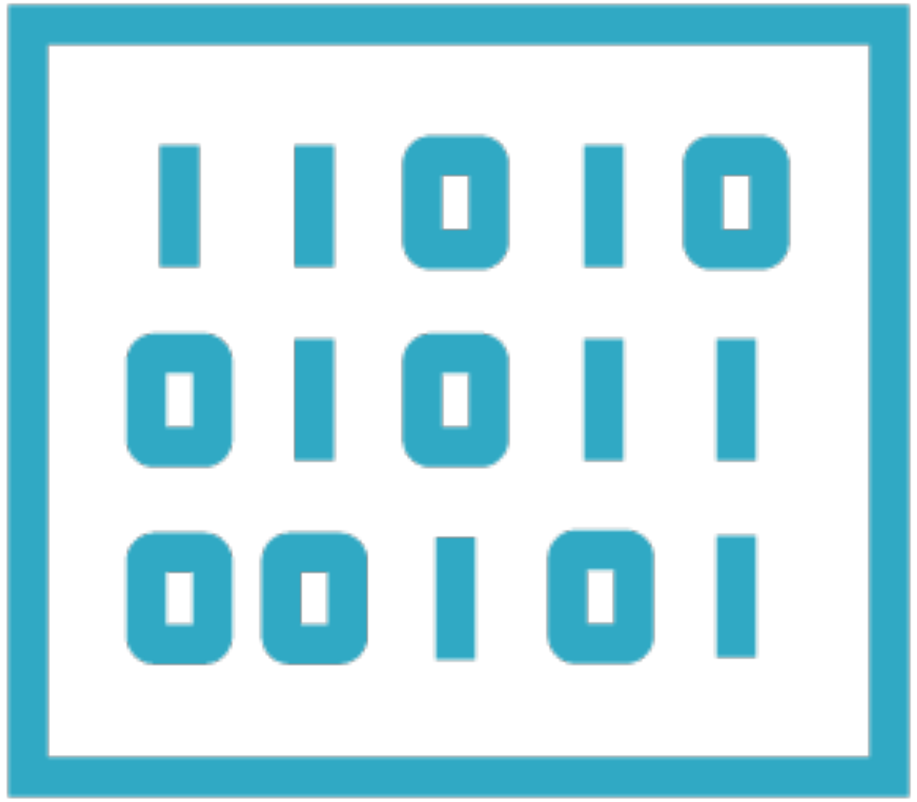
- Python
- Java
- Scala
- R

RDDs and Spark 1.x

Why is this relevant in Spark 2?

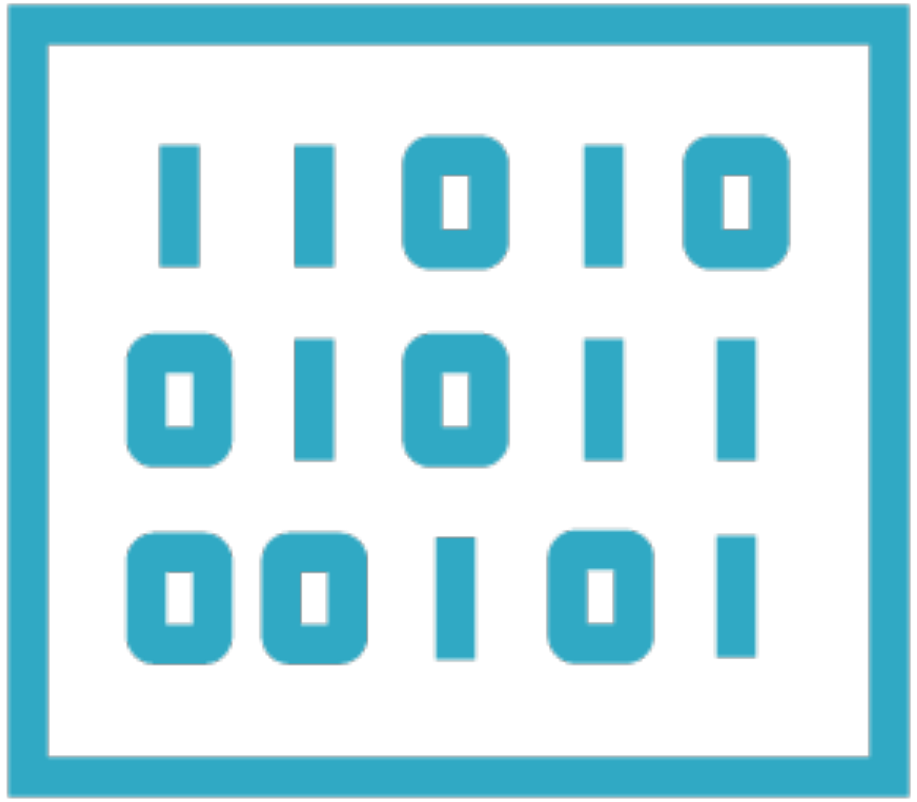
RDDs are still the **fundamental building blocks** of Spark

Resilient Distributed Datasets

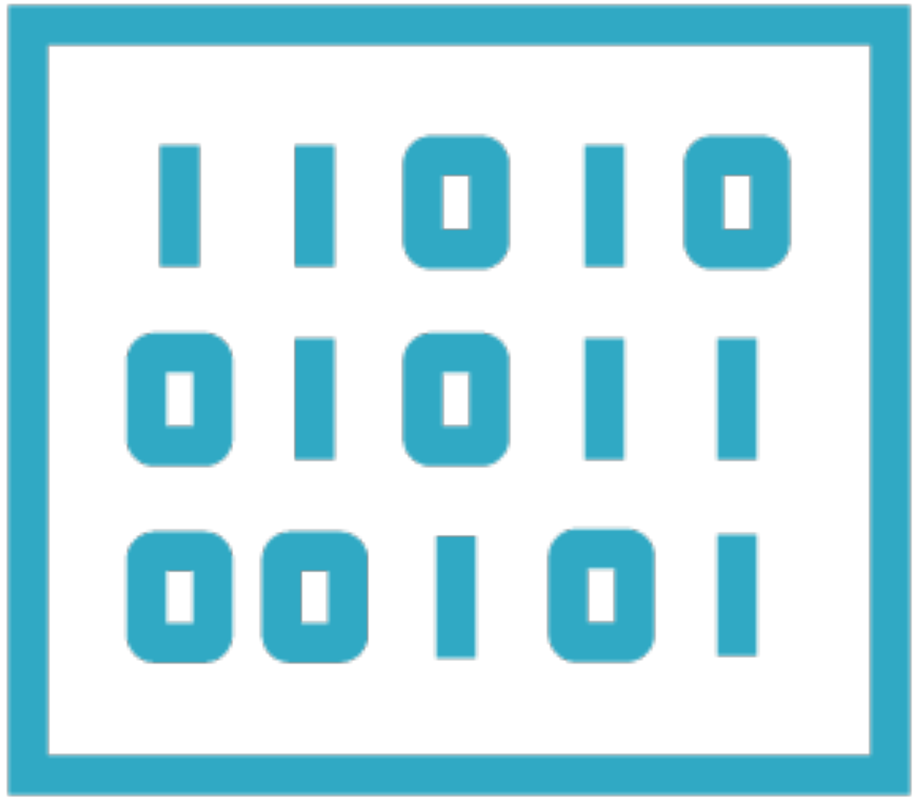


All operations in Spark are performed on in-memory objects

Resilient Distributed Datasets



An RDD is a collection of entities - rows, records



Resilient Distributed Datasets

An RDD in Spark is analogous to a **Collection in Java**

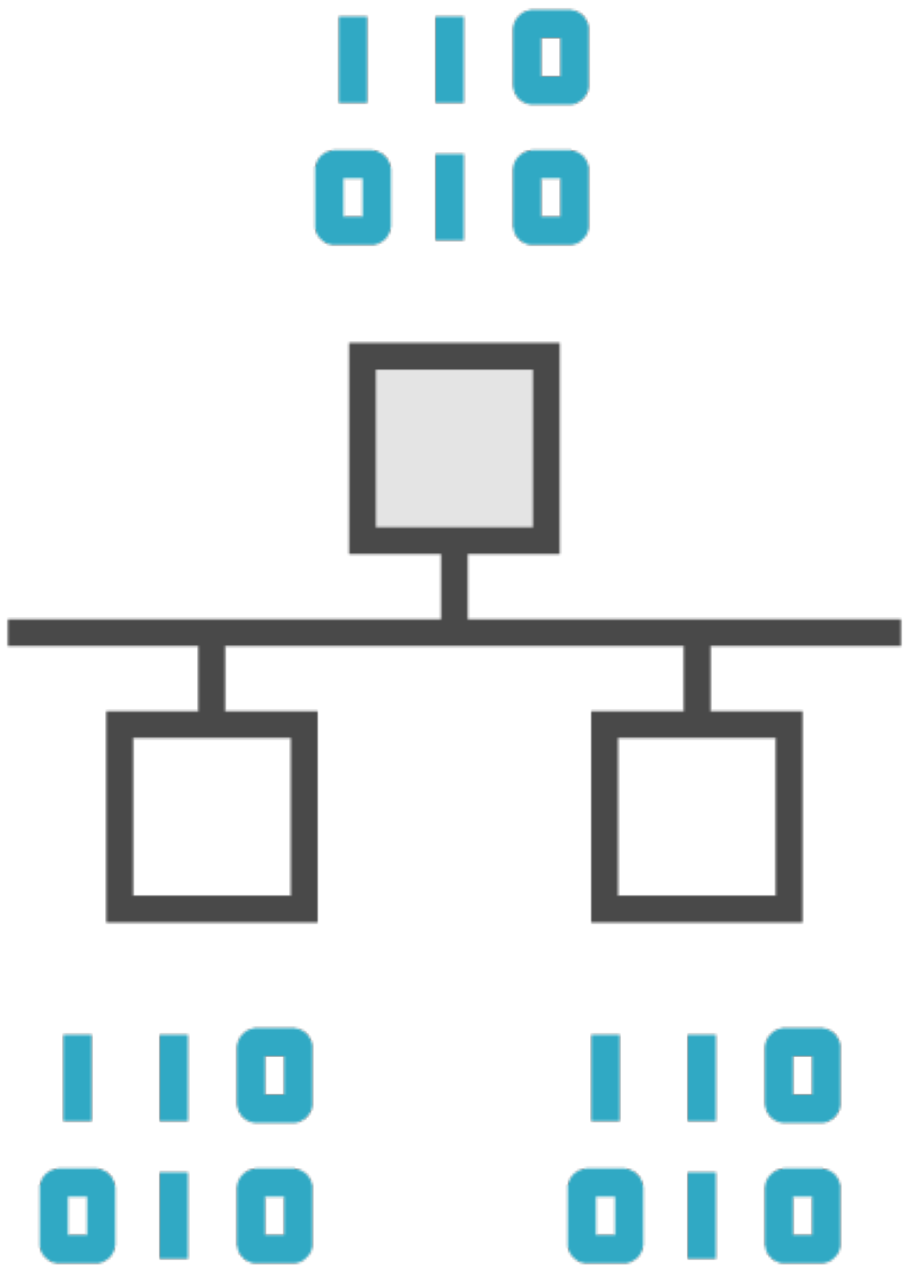
It can be **assigned to a variable** and methods can be invoked on it

Methods **return values** or **apply transformations** on the RDDs

Characteristics of RDDs

Partitioned	Immutable	Resilient
Split across data nodes in a cluster	RDDs, once created, cannot be changed	Can be reconstructed even if a node crashes

Partitioned



RDDs represent data in-memory

Partitions

1	Indigo	06:45	Bangalore
2	Jet Air	08:45	New Delhi
3	SpiceJet	09:15	Mumbai
4	Indigo	10:45	New Delhi
5	Air India	11:15	Mumbai
6	Vistara	12:00	New Delhi

Data is divided into partitions

Distributed to multiple machines

Partitions

1	Indigo	06:45	Bangalore
2	Jet Air	08:45	New Delhi
3	SpiceJet	09:15	Mumbai
4	Indigo	10:45	New Delhi
5	Air India	11:15	Mumbai
6	Vistara	12:00	New Delhi

Partitions

Data is divided into partitions

1	Indigo	06:45	Bangalore
2	Jet Air	08:45	New Delhi
3	SpiceJet	09:15	Mumbai
4	Indigo	10:45	New Delhi
5	Air India	11:15	Mumbai
6	Vistara	12:00	New Delhi

Data is divided into partitions

Partitions

1	Indigo	06:45	Bangalore
2	Jet Air	08:45	New Delhi

3	SpiceJet	09:15	Mumbai
4	Indigo	10:45	New Delhi

5	Air India	11:15	Mumbai
6	Vistara	12:00	New Delhi

Distributed to multiple machines, called nodes

Partitions

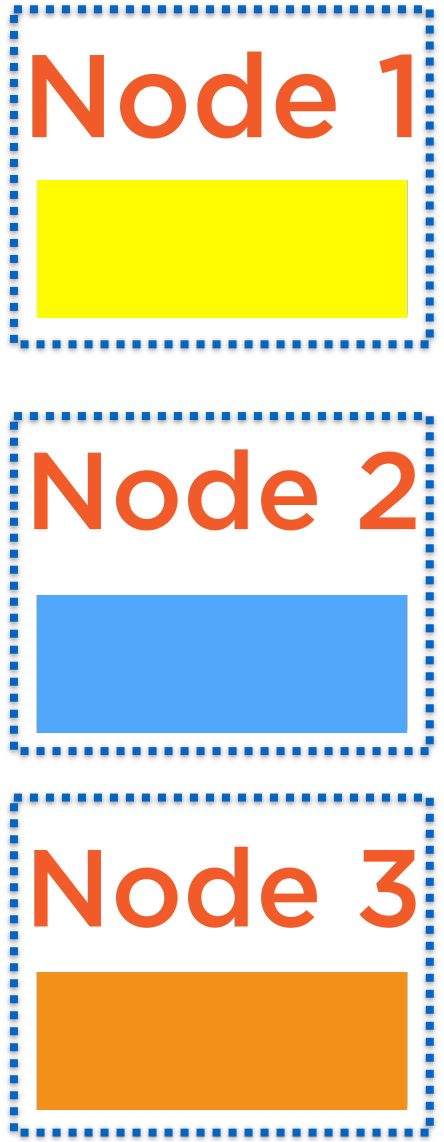
1	Indigo	06:45	Bangalore
2	Jet Air	08:45	New Delhi

3	SpiceJet	09:15	Mumbai
4	Indigo	10:45	New Delhi

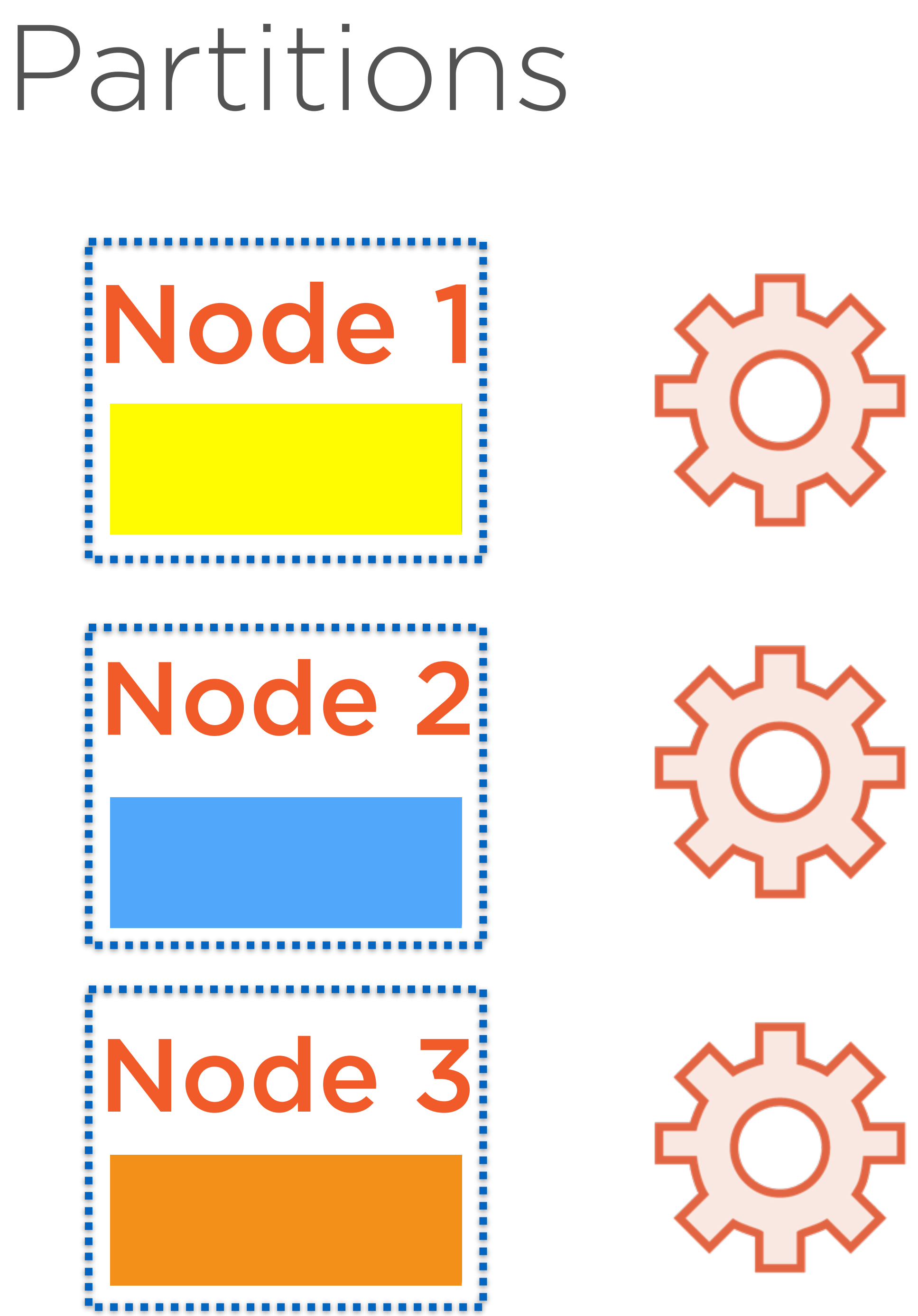
5	Air India	11:15	Mumbai
6	Vistara	12:00	New Delhi

Distributed to multiple machines, called nodes

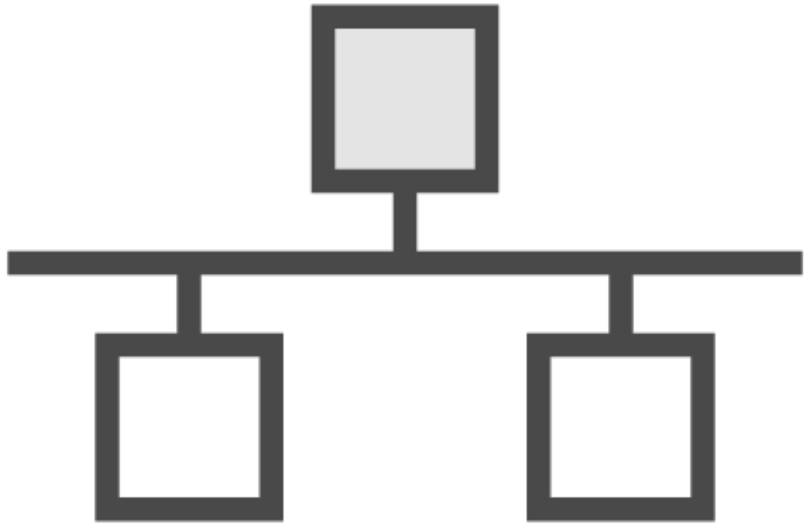
Partitions



Nodes process data in parallel



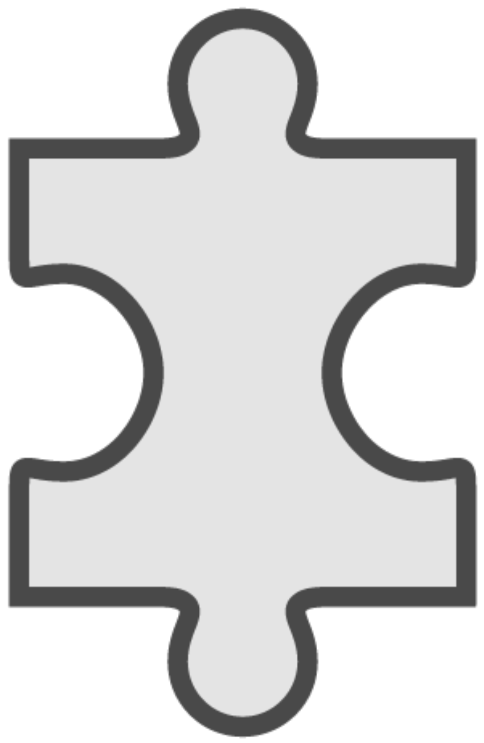
Partitioned



Processing occurs on nodes in parallel

Data is stored in memory for each node in the cluster

Immutable



An RDD cannot be mutated

Only two operations are permitted on an RDD

Only Two Types of Operations

Transformation

Transform into another RDD

Action

Request a result

Only Two Types of Operations

Transformation	Action
Transform into another RDD	Request a result

Transformation

1	Indigo	06:45	Bangalore
2	Jet Air	08:45	New Delhi
3	SpiceJet	09:15	Mumbai
4	Indigo	10:45	New Delhi
5	Air India	11:15	Mumbai
6	Vistara	12:00	New Delhi

A data set loaded into an RDD

The user may define a chain of transformations on the dataset

Transformation

1	Indigo	06:45	Bangalore
2	Jet Air	08:45	New Delhi
3	SpiceJet	09:15	Mumbai
4	Indigo	10:45	New Delhi
5	Air India	11:15	Mumbai
6	Vistara	12:00	New Delhi

1. Load data
2. Pick only the 3rd column
3. Sort the values

Transformations are **executed** only when a result is requested

Only Two Types of Operations

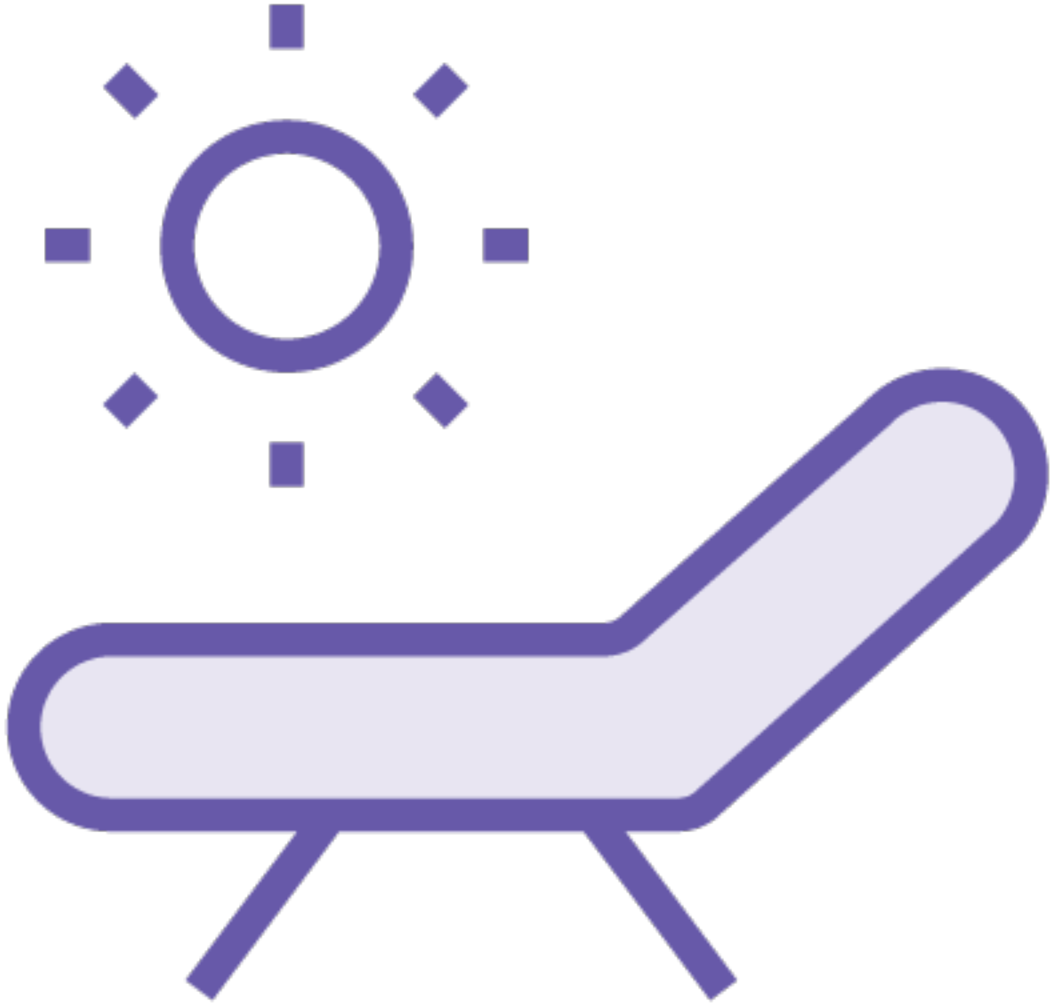
Transformation	Action
Transform into another RDD	Request a result

Action

Request a result using an action

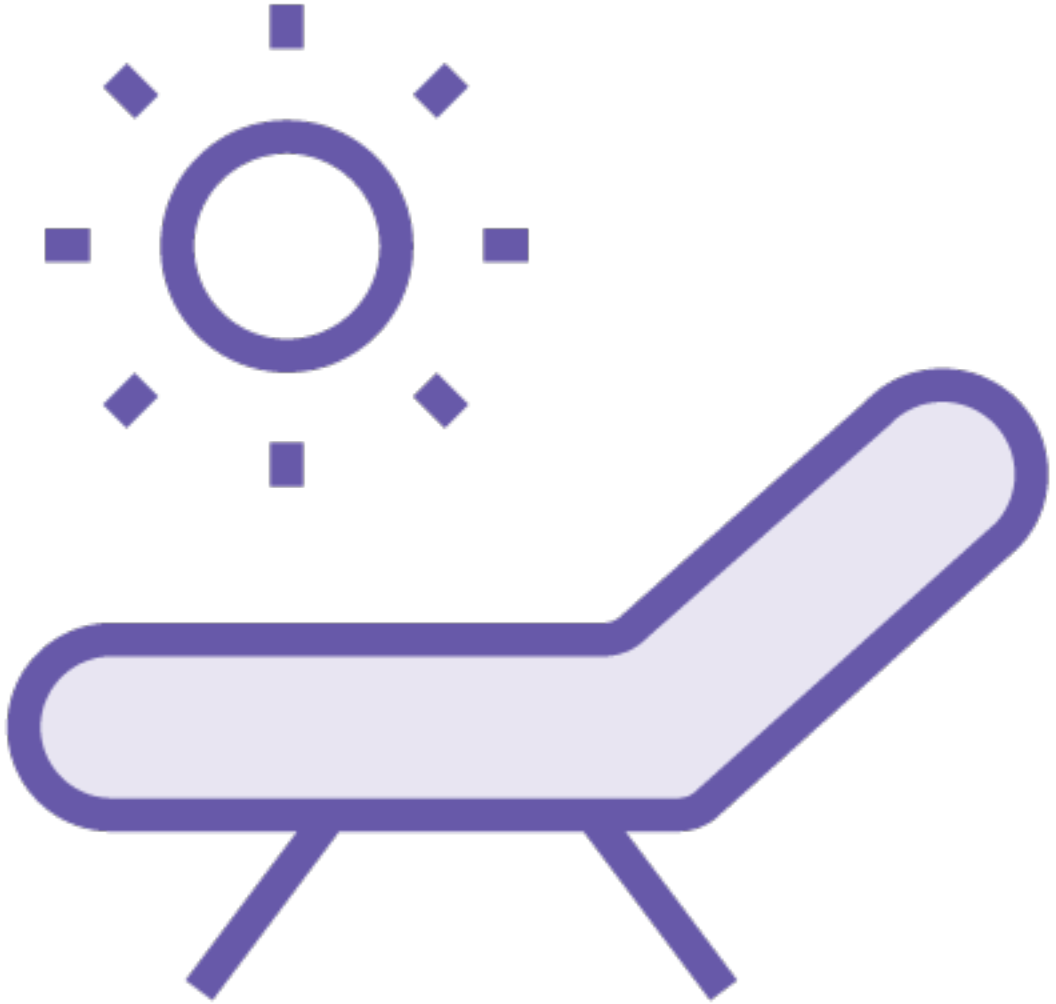
1. The first 10 rows
2. A count
3. A sum

Lazy Evaluation



Spark keeps a record of the series of transformations requested by the user

Lazy Evaluation



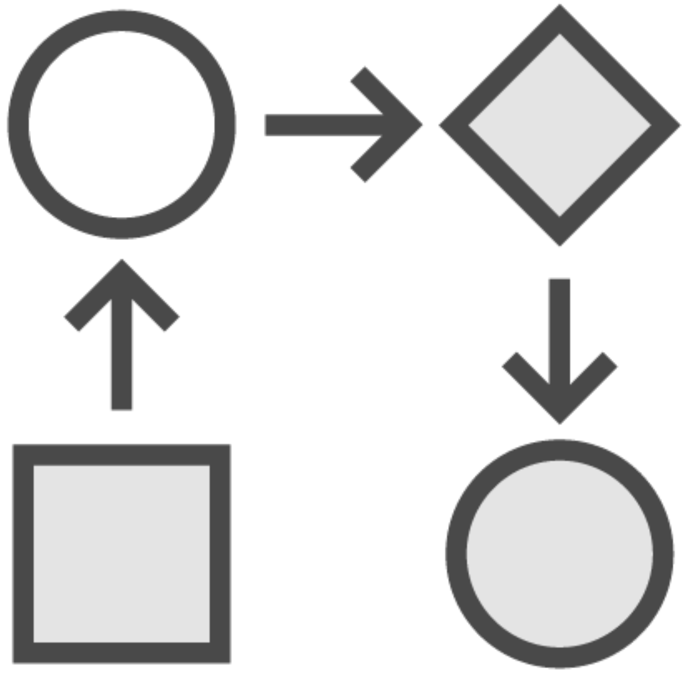
It groups the transformations in an efficient way when an Action is requested

Immutable

Action: Read data from an RDD

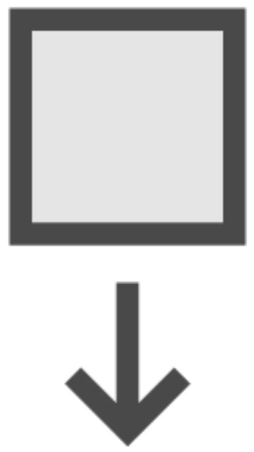
Transformation: Transform the RDD to create another RDD

Resilient



RDDs can be reconstructed even if the node it lives on crashes

RDDs can be created in 2 ways



Reading a file

Transforming another RDD

RDDs Are Resilient

Reading a file

Transforming another RDD

Every RDD keeps track of **where** it came from

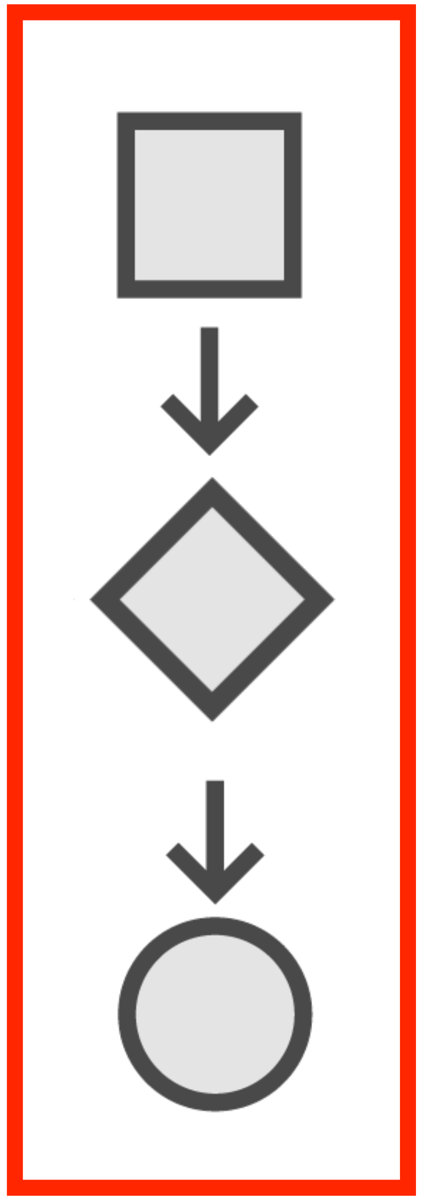
RDDs Are Resilient

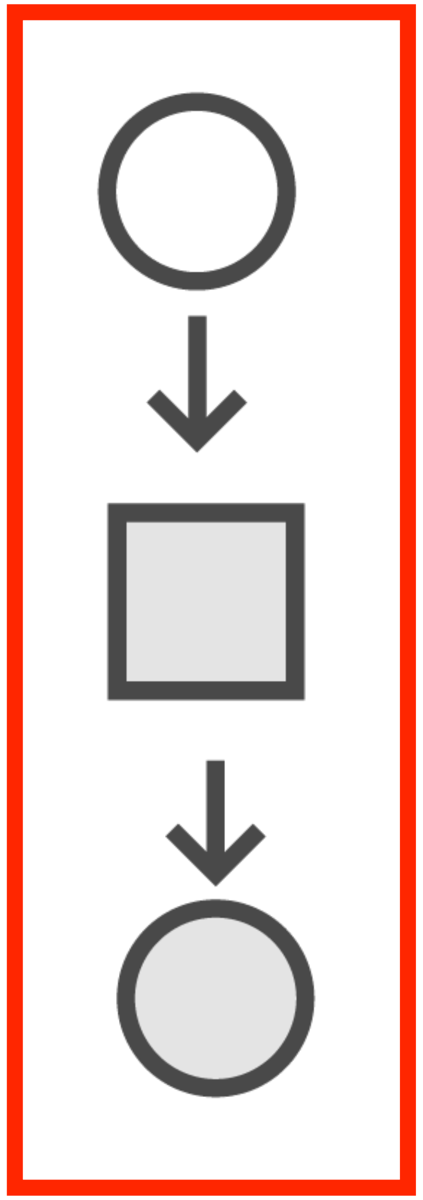
It tracks every transformation which led to the current RDD

RDDs Are Resilient

However many transformations it takes

RDDs Are Resilient

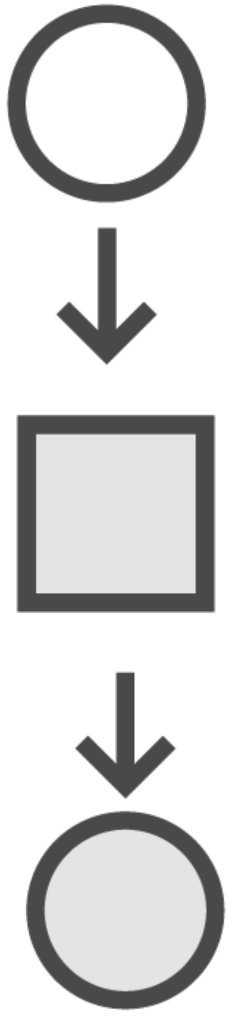


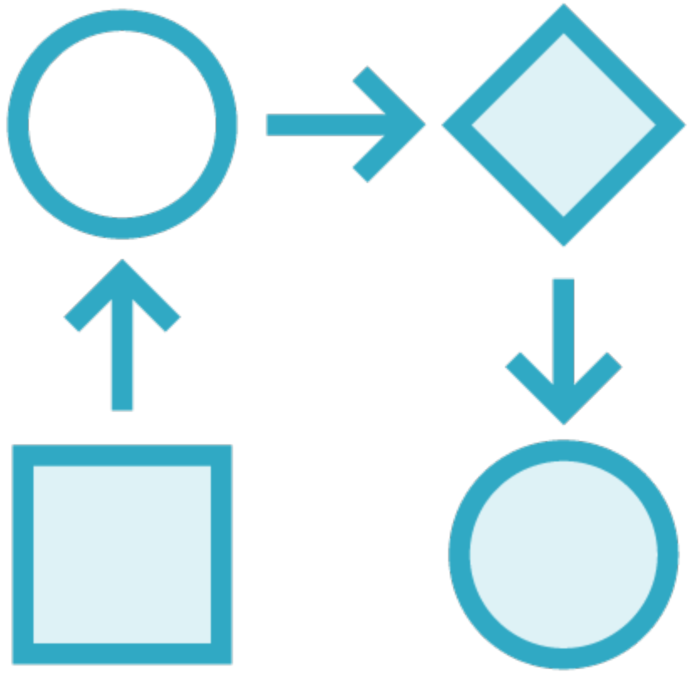


This is the RDD's lineage

RDDs Are Resilient

None of the transformations are applied till we access the results

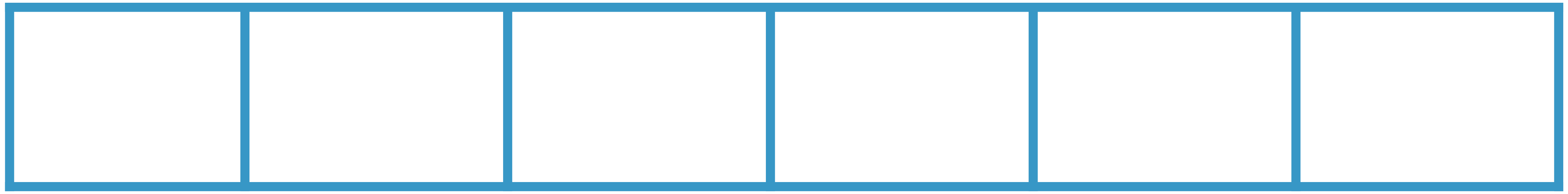




Lineage

Allows RDDs to be reconstructed when nodes crash

Allows RDDs to be lazily instantiated (materialized) when accessing the results



```
airlines = sc.textFile(airlinesDataPath)
```

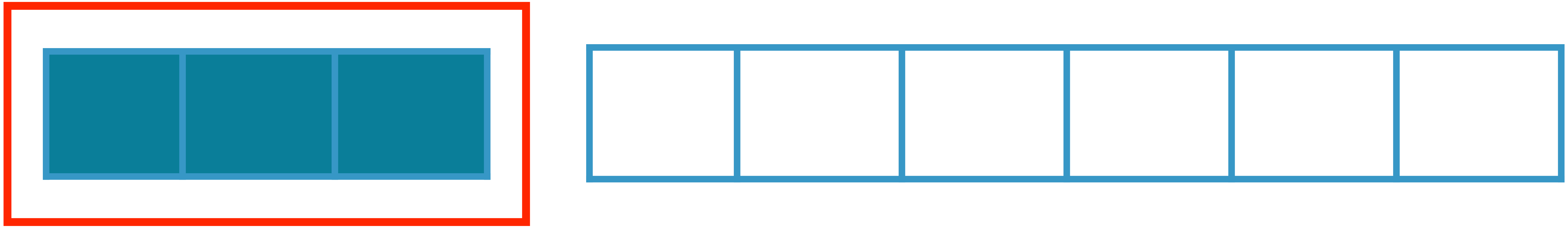
Create an RDD with Flight Data

Create an RDD from raw data in an input file

```
airlinesFiltered = airlines.filter(lambda x:
'Lufthansa' in x)
```

Create a New RDD with Filtered Flight Data

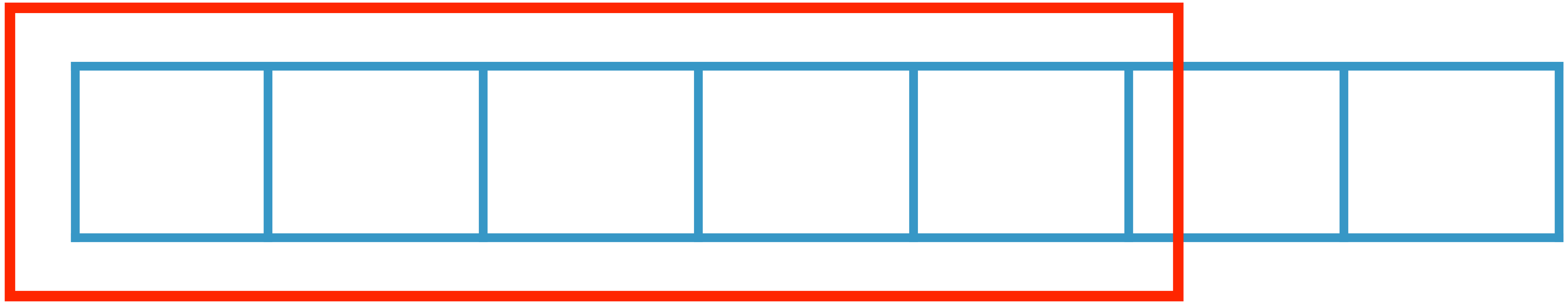
Include the rows referring to Lufthansa from the original RDD



```
airlinesFiltered = airlines.filter(lambda x:
‘Lufthansa’ in x)
```

Create a New RDD with Filtered Flight Data

Include the rows referring to Lufthansa from the original RDD



```
airlinesFiltered.take(5)
```

View the First 5 Rows in the RDD

The rows will be displayed on screen

Characteristics of RDDs

Partitioned	Immutable	Resilient
Split across data nodes in a cluster	RDD once created cannot be changed	Can be reconstructed even if a node crashes

RDDs, DataFrames, Datasets

DataFrame: Data in Rows and Columns

DATE	OPEN	...	PRICE
2016-12-01	772	...	779
2016-11-01	758	...	747
2006-01-01	302	...	309

Each row represents 1 observation

DataFrame: Data in Rows and Columns

DATE	OPEN	...	PRICE
2016-12-01	772	...	779
2016-11-01	758	...	747
2006-01-01	302	...	309

Each column represents 1 variable (a list or vector)

From File to DataFrame

DATE	OPEN	…	PRICE
2016-12-01	772	…	779
2016-11-01	758	…	747
2006-01-01	302	…	309

File

read

DATE	OPEN	...	PRICE
2016-12-01	772	…	779
2016-11-01	758	…	747
2006-01-01	302	…	309

DataFrame

DataFrames: From R to Spark

Relational databases

Data in rows and columns

Strict schema, constraints

R DataFrames

Primary abstraction in R

Convenient to read, sort and filter

Pandas in Python

Modeled on R DataFrames

Compatible with Numpy, TF...

RDDs in Spark

Similar to Python collections

map, flatMap, reduceByKey...

Datasets in Spark

Added in Spark 1.6

Scala and Java, not Python or R

DataFrames in Spark

Dataset with named columns

Like R or Pandas!

RDDs to Datasets

RDDs	Datasets
Primary abstraction since initial versions	Added to Spark in 1.6
Immutable and distributed	Also immutable and distributed
Strong typing, use of lambda	Also support strong typing, lambdas
No optimized execution	Leverage optimizers in recent versions
Available in all languages	Present in Scala and Java, not Python or R

Datasets to DataFrames

Datasets	DataFrames
Added to Spark in 1.6	Added to Spark in 1.3
Immutable and distributed	Also immutable and distributed
No named columns	Named columns, like Pandas or R
Extension of DataFrames - type-safe, OOP interface	Conceptually equal to a table in an RDBMS
Compile-time type safety	No type safety at compile time
Present in Scala, Java, not Python, R	Available in all languages

Starting Spark 2.0, APIs for Datasets and DataFrames have merged

Datasets to DataFrames

Datasets

Scala and Java*

*Datasets of the Row() object in Scala/ Java often called DataFrames

DataFrames

Python, R, Scala, Java

Equivalent to Dataset<Row> in Java or Dataset[Row] in Scala

DataFrames Built On Top of RDDs

Partitioned

Split across data nodes in a cluster

Immutable

Once created, cannot be changed

Resilient

Can be reconstructed even if a node crashes

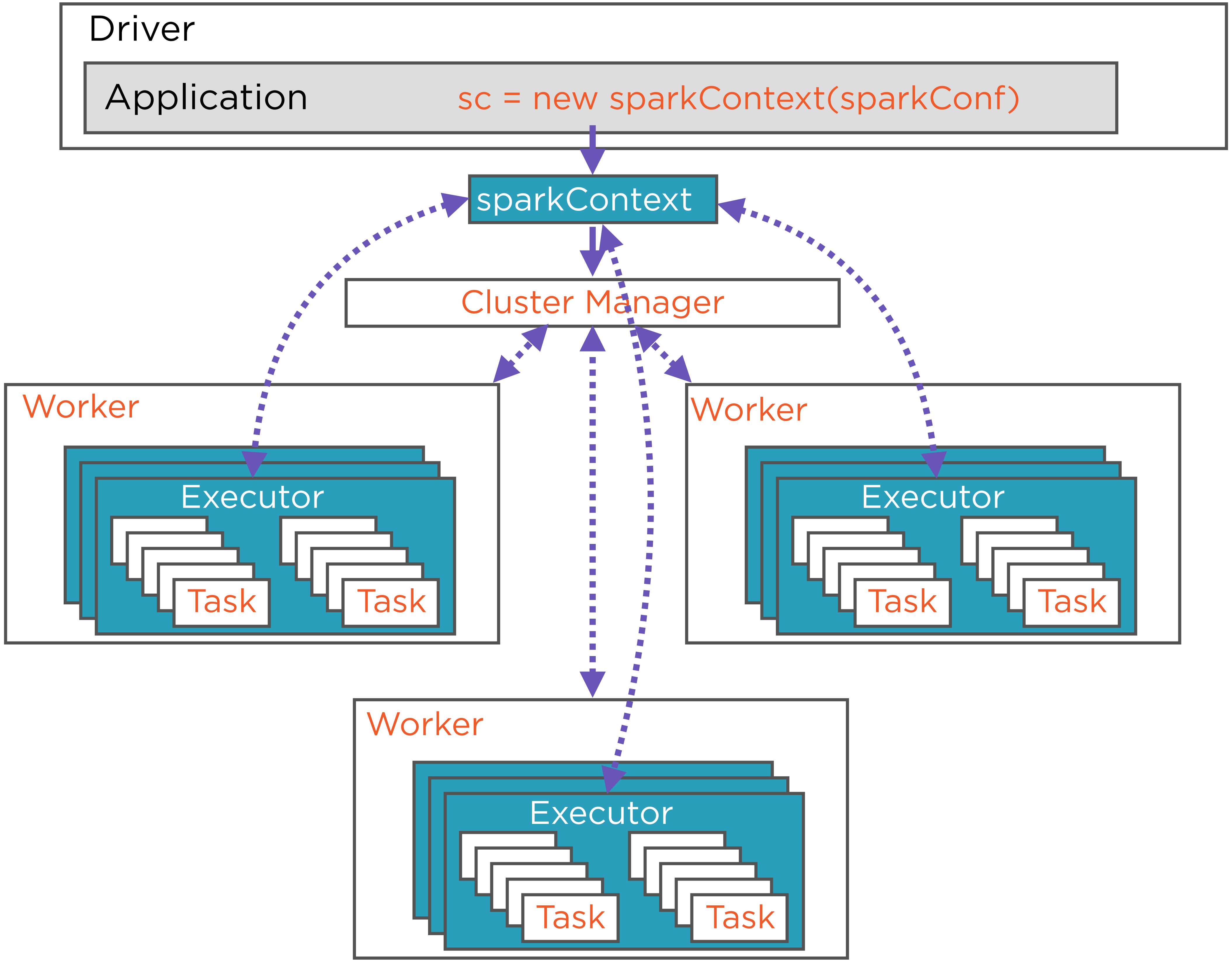
Demo

Install standalone Spark on your local machine

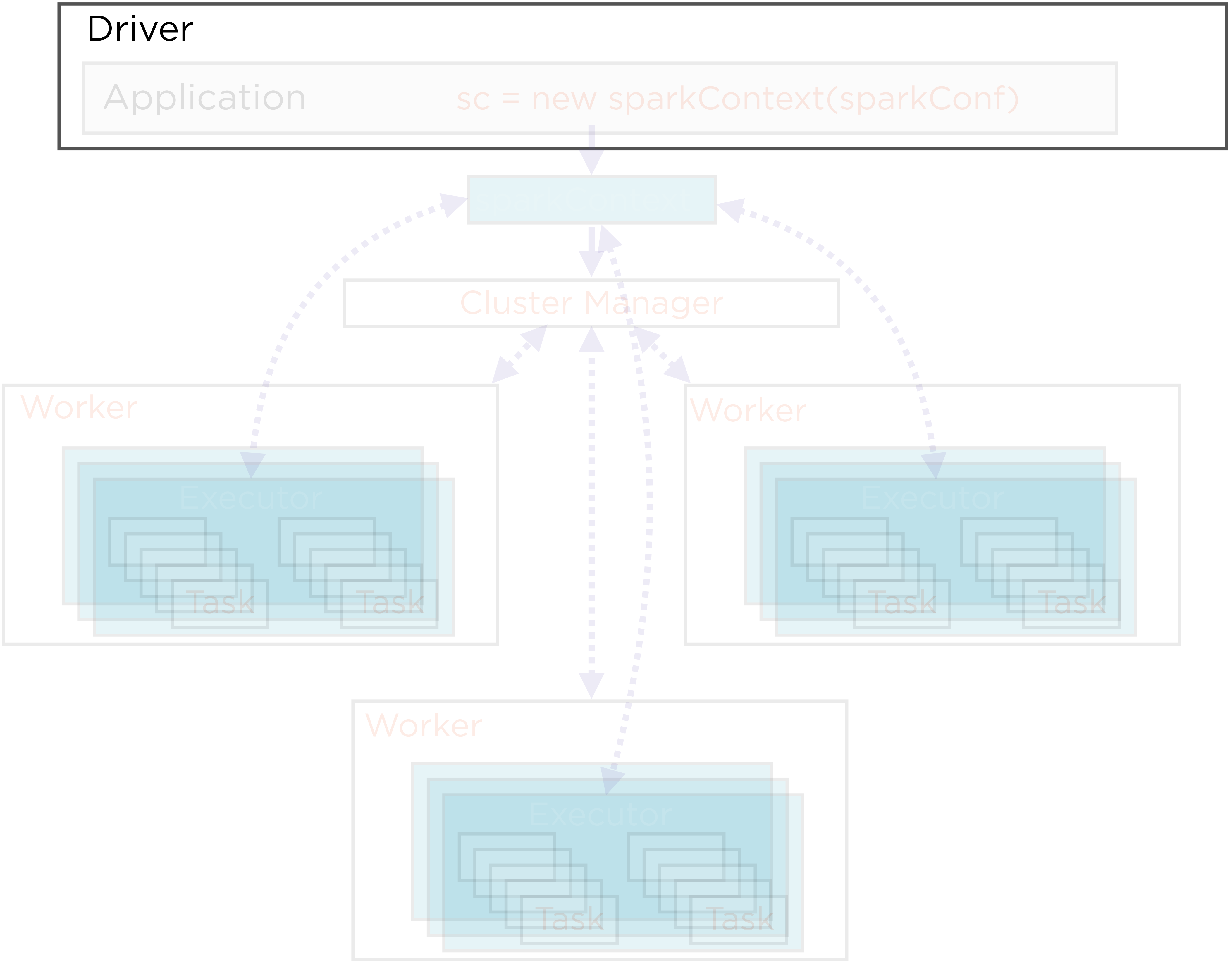
Set up the PySpark REPL interface

The Architecture of Spark

Spark 1.x Architecture



Spark 1.x Architecture



Driver

Separate process (JVM)

The master node in a Spark application

Launches tasks

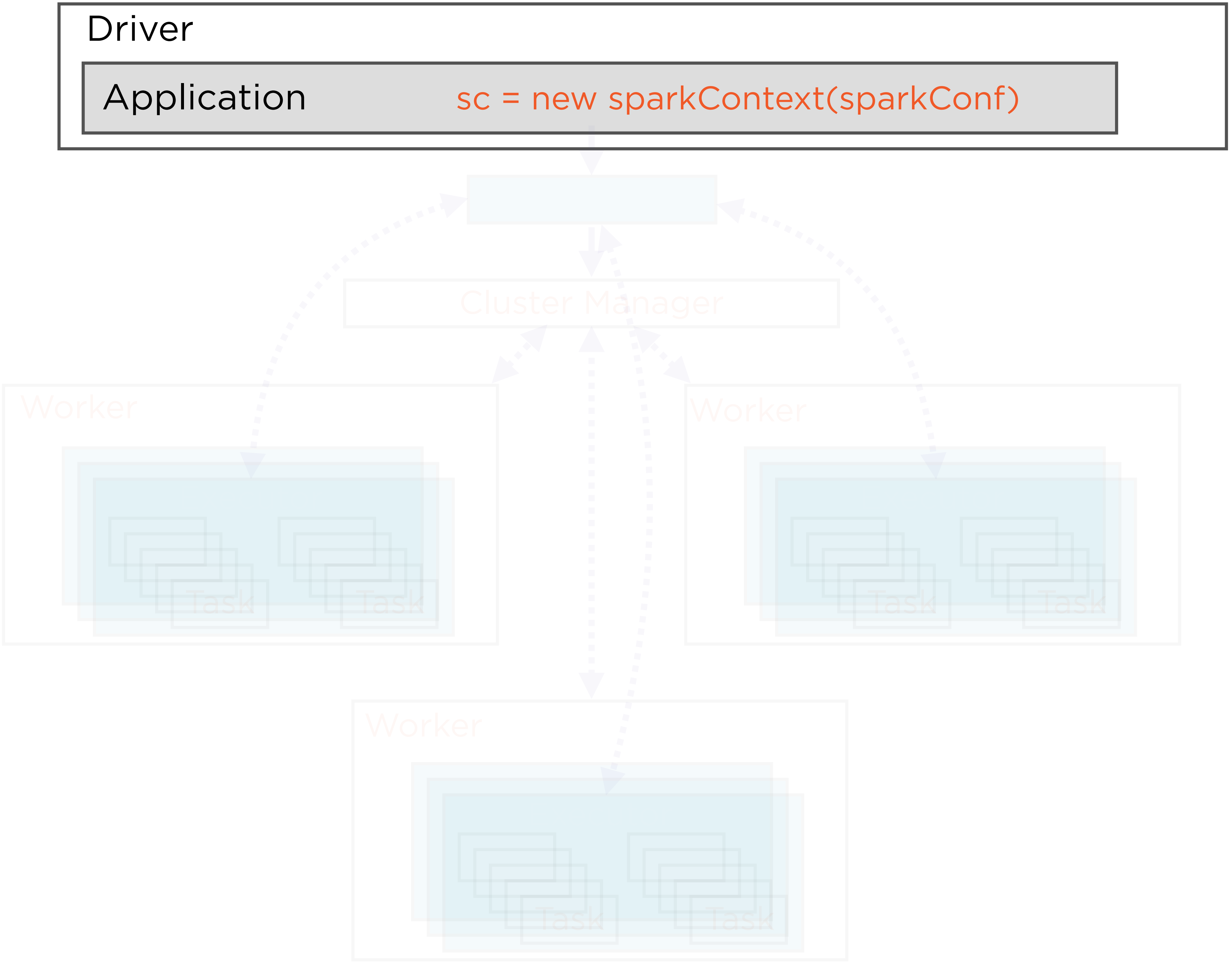
Hosts SparkContext

Driver

Several groups of services run inside the driver

- SparkEnv
- DAGScheduler
- Task Scheduler
- SparkUI
- ...

Spark 1.x Architecture



Spark Application

Uses SparkContext as entry point

Creates RDD *D*irected *A*cyclic *G*raph

Internally, Spark creates *Stages* (physical execution plan)

Each stage is split into operations on RDD partitions called *Tasks*

Execution in Spark 1.x

In Spark 1.x, execution optimization resembled traditional DBMS

“Volcano Iterator Model”

Missed several code/compiler optimizations

Execution in Spark 2.x

Tungsten engine (2nd generation)

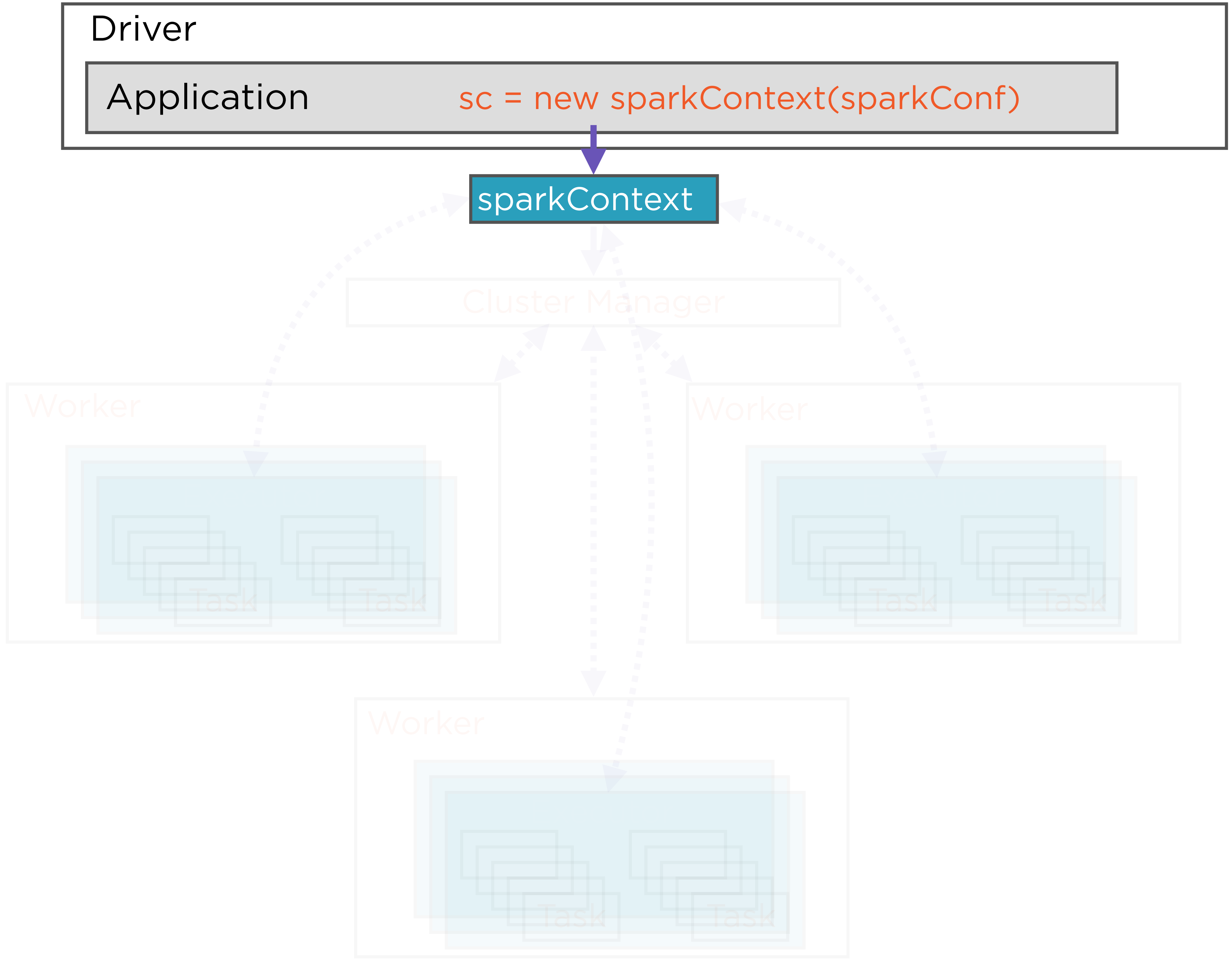
Eliminate virtual function calls

Store data in registers, not RAM/cache

Compiler loop unrolling, pipelining

Spark 2.0 uses Tungsten, an engine that speeds up execution 10-20X

Spark 1.x Architecture



SparkContext (Spark 1.x)

Familiar code entry point to Spark

```
sc = new sparkContext(…)
```

Create RDDs, accumulators…

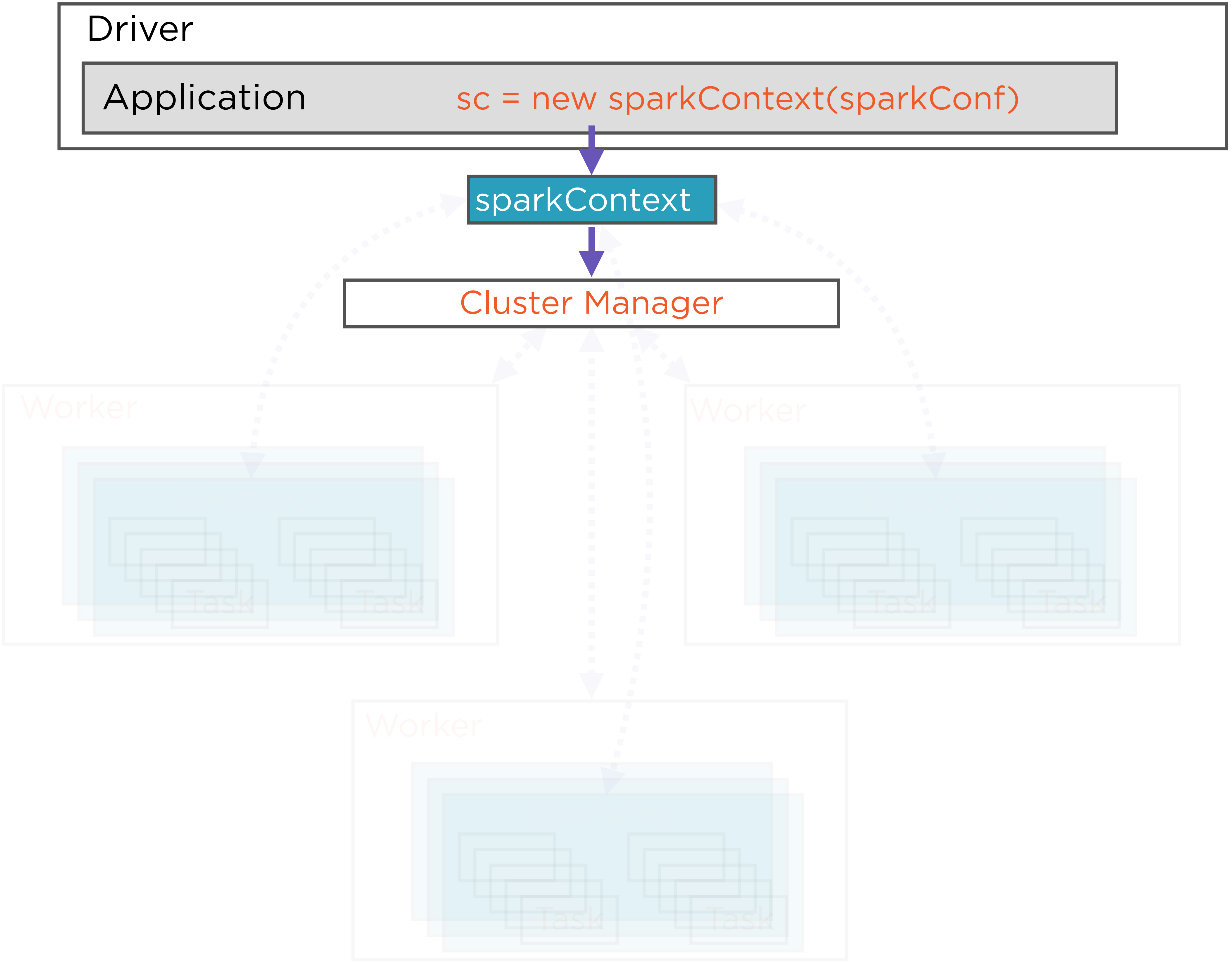
Run jobs

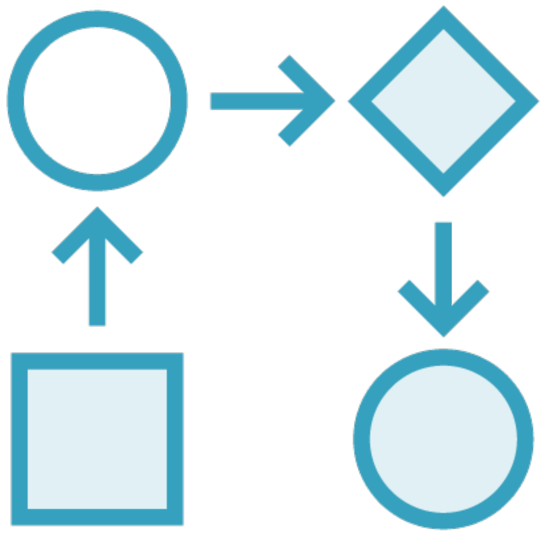
SparkSession (Spark 2.x)

In Spark 2.x, SparkContext is wrapped in SparkSession

Encapsulates SQLContext, HiveContext...

Spark 1.x Architecture





Cluster Manager

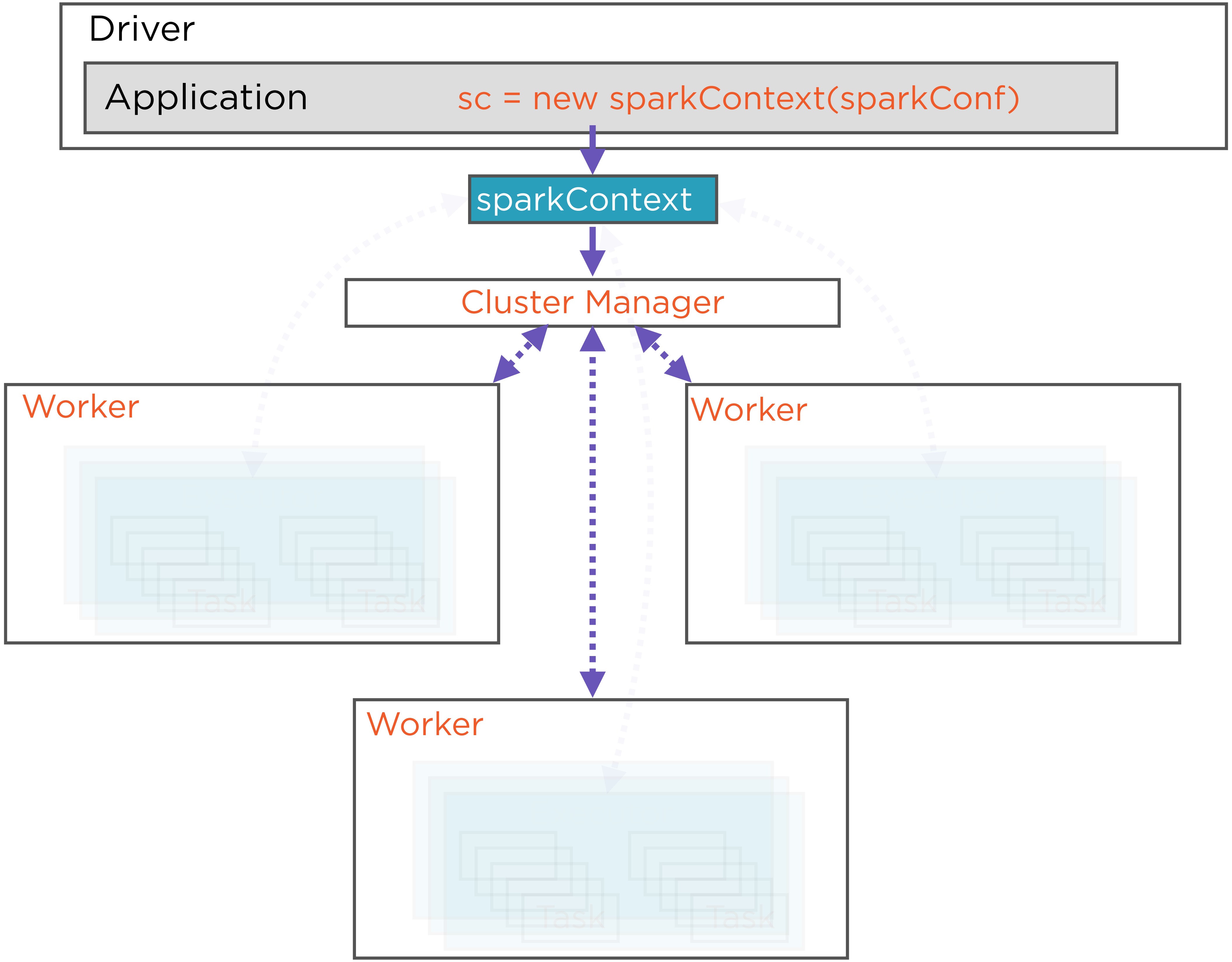
Hadoop's YARN

Apache Mesos

Spark Standalone

Orchestrates execution

Spark 1.x Architecture



Workers

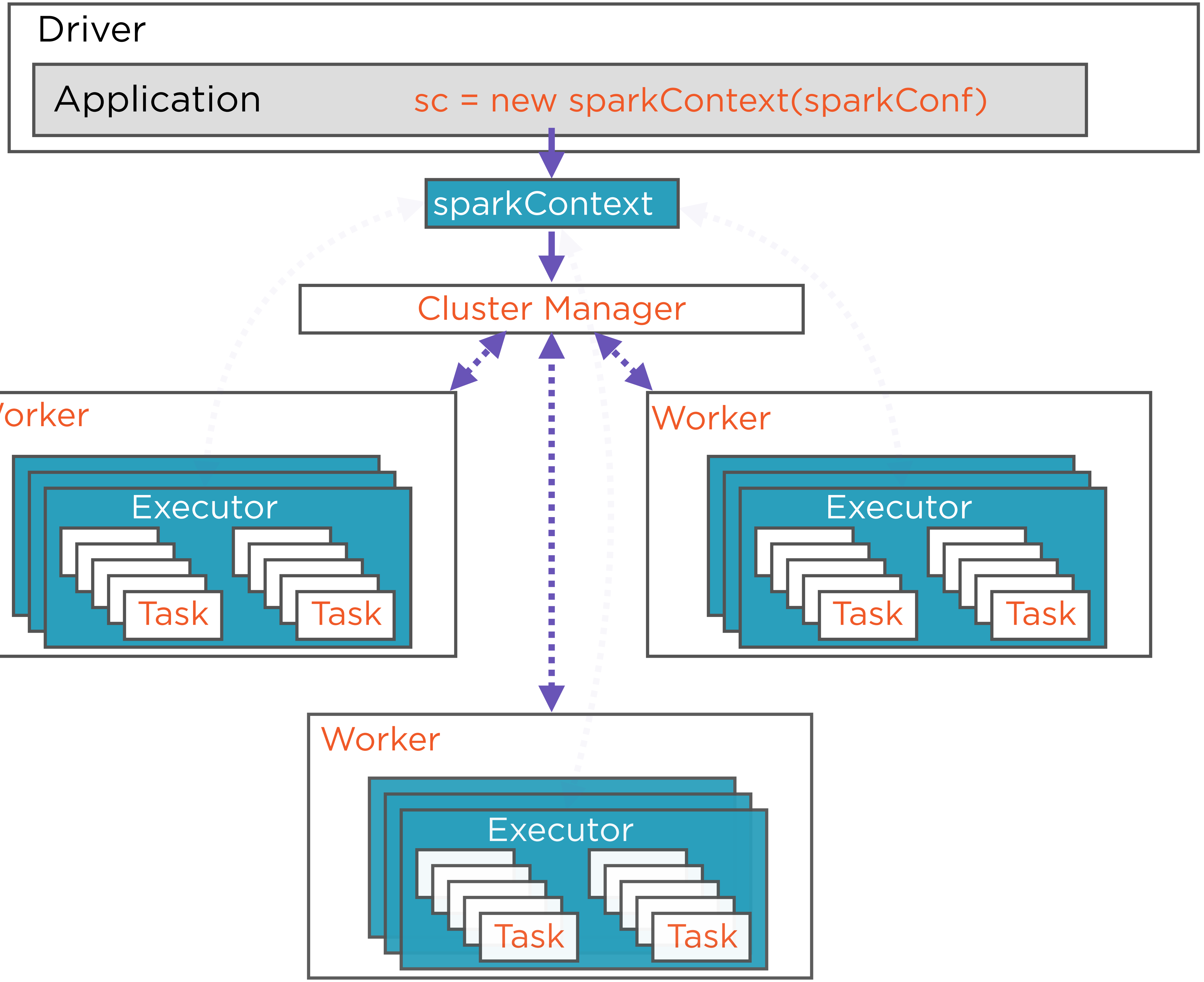
Compute nodes in cluster

Runs the Spark application code

When SparkContext created...

...Each worker starts *executors*

Spark 1.x Architecture



Executor

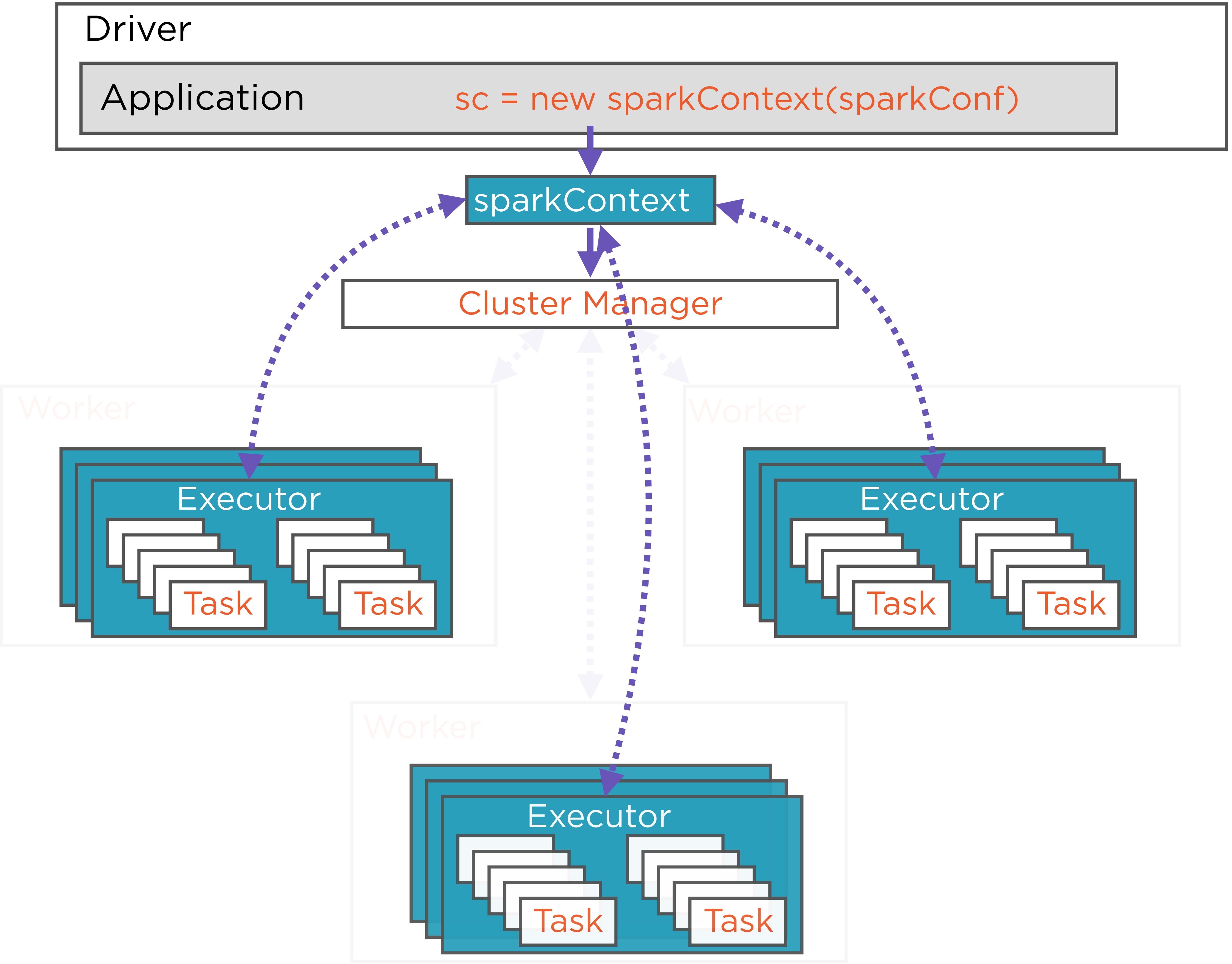
Distributed agents that execute *tasks*

Tasks are basic units of execution

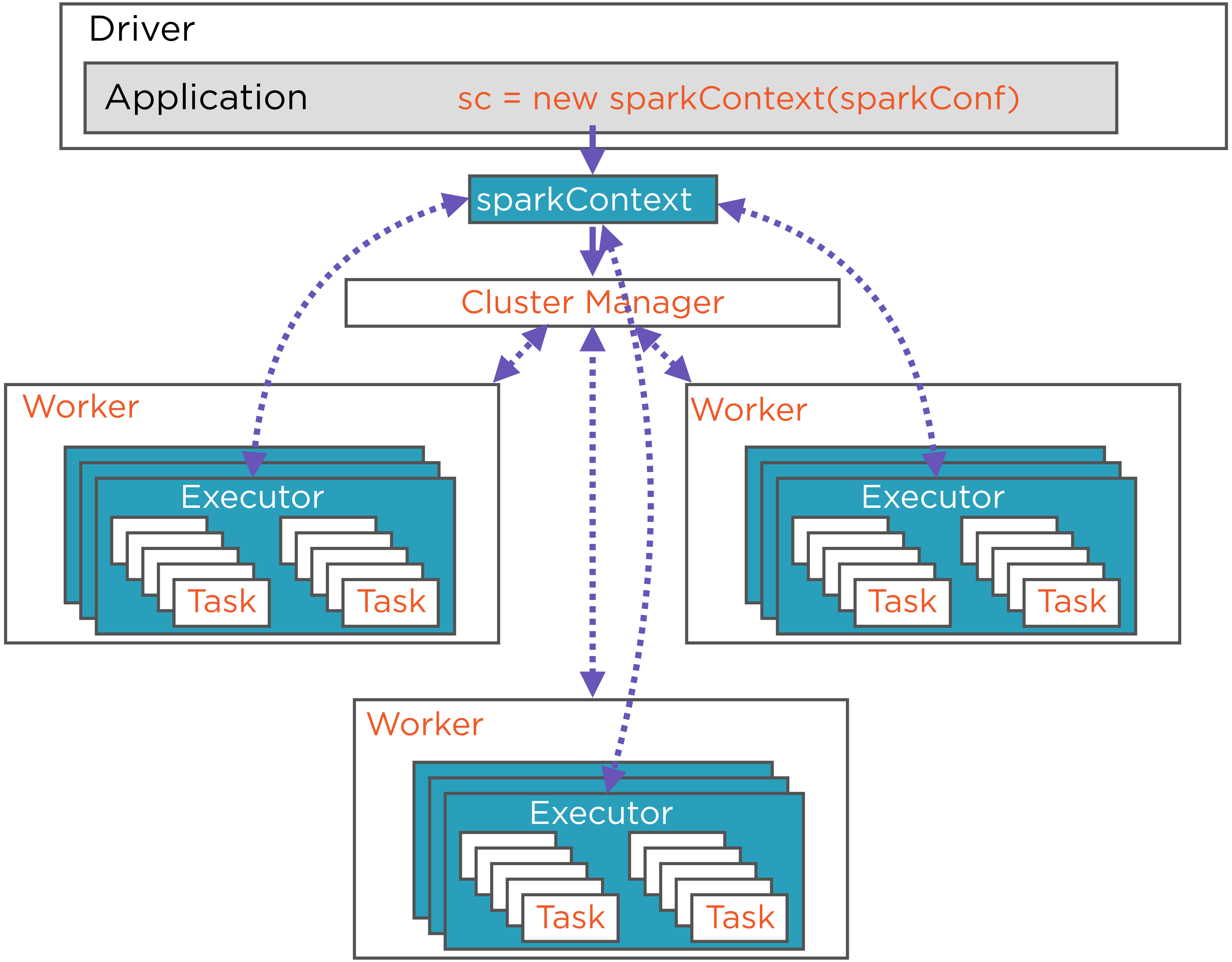
Tasks belong inside *stages*

Stages are physical units of execution

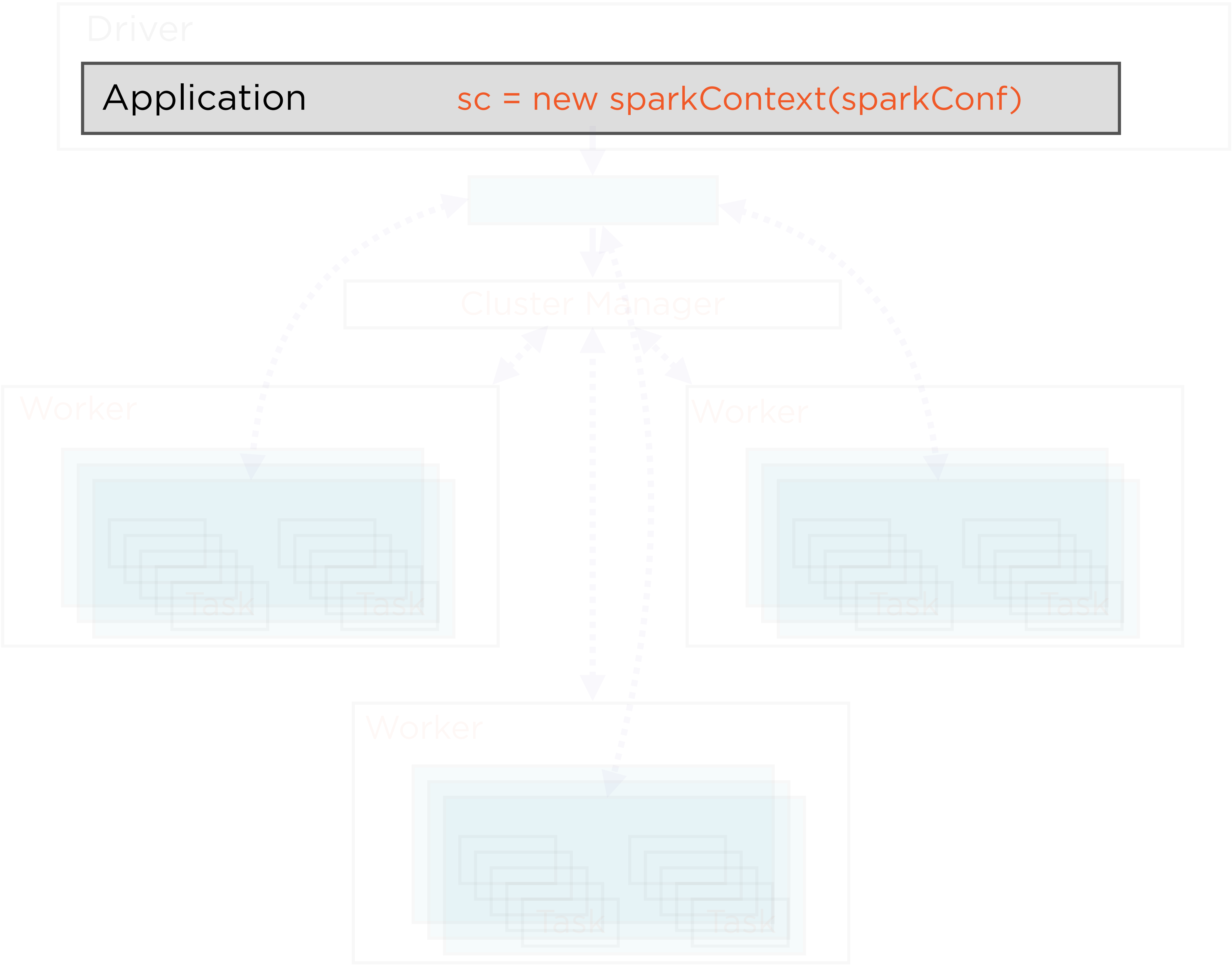
Spark 1.x Architecture



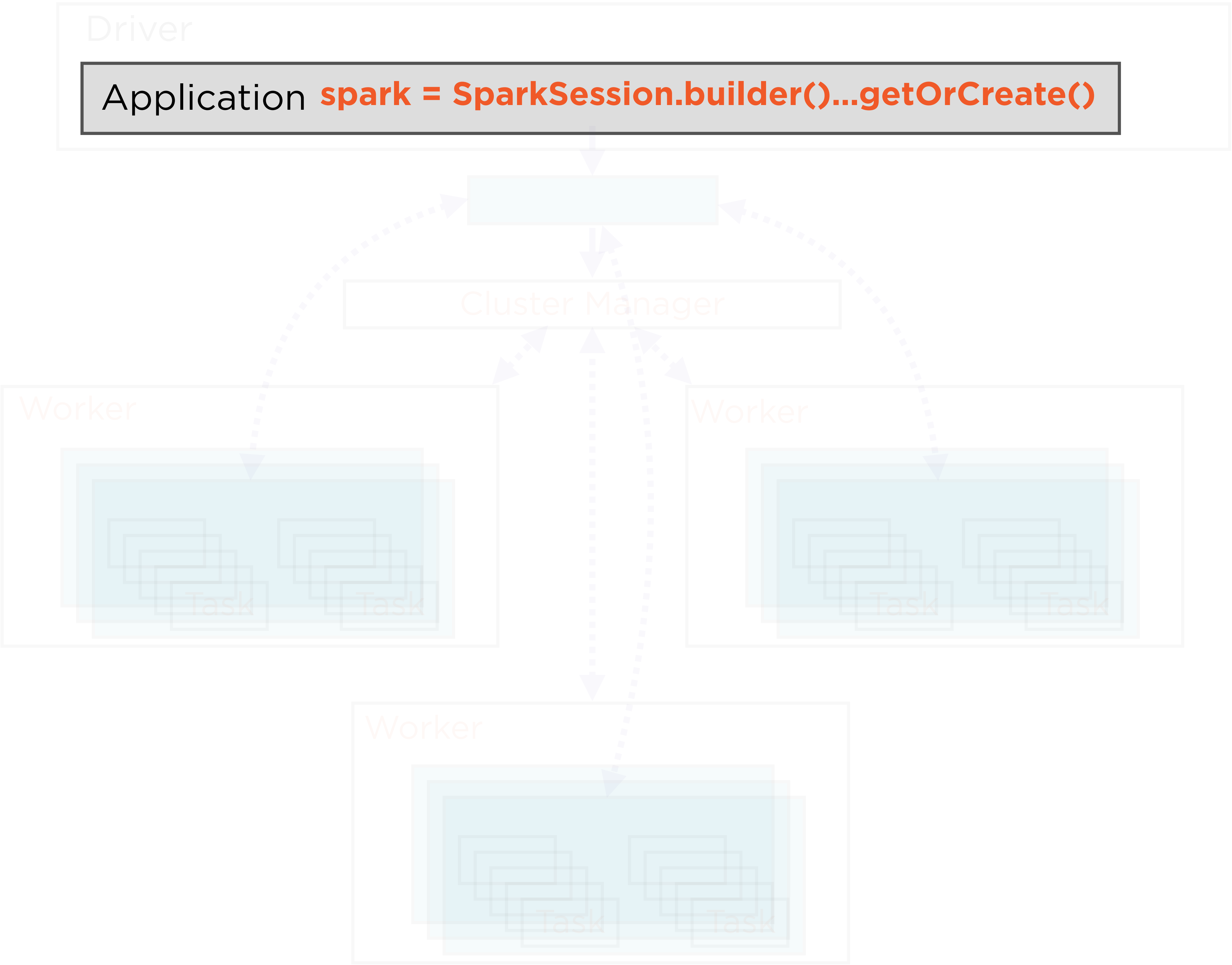
Spark 1.x Architecture



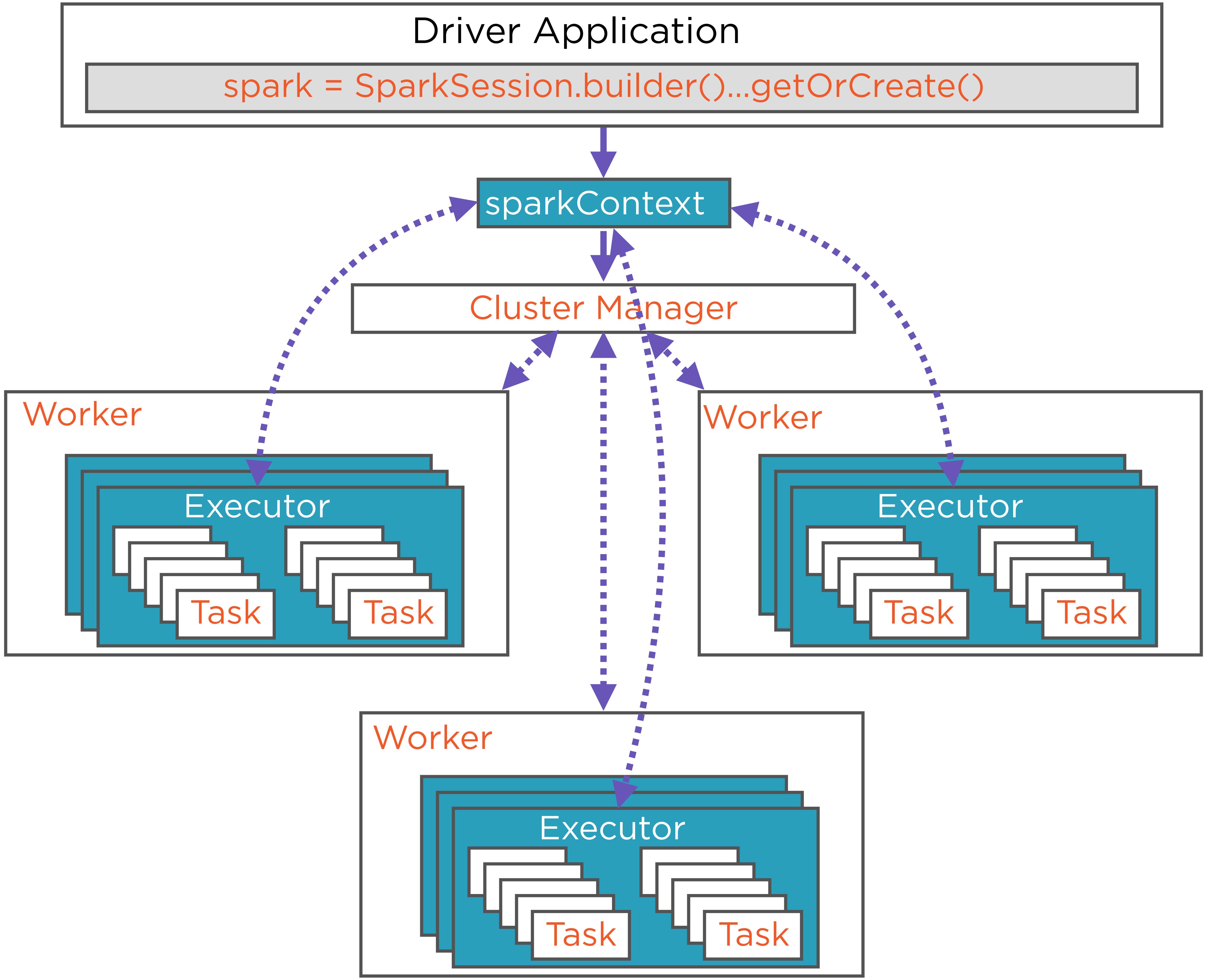
Spark 1.x Architecture



Spark 2.x Architecture



Spark 2.x Architecture



The basic architecture is largely the same in Spark 1.x and 2.x

The real difference is in the **speed** of execution due to **Tungsten optimizations**

Demo

Working with RDDs and DataFrames

Interoperability between RDD and DataFrames

Introducing the SparkContext and the SQLContext

Spark 2.0 vs. Spark 1.x

Changes Starting Spark 2.0

Easier

Unifying Datasets and DataFrames, SQL support…

Faster

Optimize like a compiler, not a DBMS

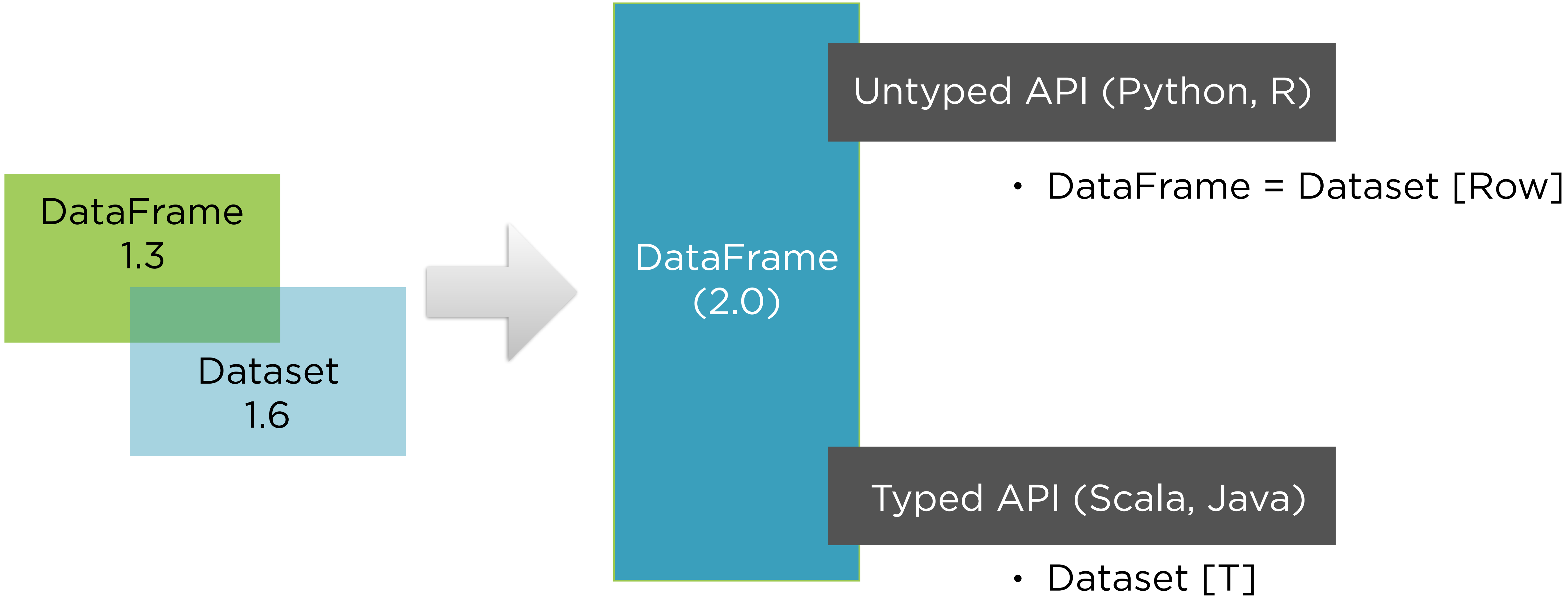
Ease of Use

Unified API for DataFrames

spark.ml and ML pipelines

Advanced streaming

Unified API for DataFrames



spark.ml and spark.mllib

spark.mllib	spark.ml
Older	Newer
RDDs	DataFrames
For now, more functionality	Functionality catching up
	Support for ML pipelines

Continuous Applications

A stream is not a stream, it is simply an unbounded dataset

Unify batch and streaming pipelines

Same queries for both

Streaming and Structured Streaming

Streaming	Structured Streaming
Older	Newer
RDDs	DataFrames
No optimizations	Optimizations on DataFrames
Batch and streaming support not unified	Unified support for batch and streaming

Project Tungsten

Changes Starting Spark 2.0

Easier

Unifying Datasets and DataFrames, SQL support…

Faster

Optimize like a compiler, not a DBMS

Project Tungsten

Umbrella project, launched in April 2015, to make changes to Apache Spark’s execution engine that focuses on substantially improving the efficiency of memory and CPU

Backdrop

In early 2015...

- Disks are getting faster: SSDs
- Networks are getting faster: 10 Gbps links
- CPU is now the bottleneck

Backdrop

But…

- Spark was optimizing for IO or network communication
- Java makes heavy use of virtual functions
- JVM garbage collection costs were heavy

Volcano Iterator Model

Used before Project Tungsten

Classic query evaluation strategy

Each query consisted of operators

Each operator implemented an interface

Volcano Iterator Model

Design Choice	Implication
Heavy use of interfaces	Lots of virtual function dispatches
Query represented as complex tree of function calls	Compiler features such as pipelining, prefetching, loop unrolling become hard
Interface has single method next()	Each tuple placed on call stack (memory, not CPU register)

Project Tungsten

Memory management: Go beyond JVM object model and GC

Cache-aware computation: Refine algorithms and data structures

Code generation: Learn from modern compilers

From Volcano to Tungsten

Volcano in Spark 1.x	Tungsten in Spark 2.x
Classic DBMS strategy, optimizes IO and network access	Focus on CPU optimization; disk and network no longer the bottleneck
Rely on Java objects and JVM garbage collection	Take control of object creation and garbage collection
Hard to leverage modern compiler optimizations	Heavily leverage loop unrolling, 1 CPU instruction for multiple tuples
Lots of virtual function dispatches	Entirely avoid virtual function calls

Advances

Far leaner object serialization than Java or Kryo

Break with Java objects and GC

Efficient algorithms (e.g. sorting without deserialization)

Many more

Results

First generation Tungsten engine default in Spark 1.5

Second generation Tungsten engine powers Spark 2.0

Strong improvements in

- DataFrames
- SparkSQL
- Some RDD APIs

Tungsten in Spark 2.0

Optimize like a compiler, not DBMS

Tungsten engine (2nd generation)

Eliminate virtual function calls

Store data in registers, not RAM/cache

Compiler loop unrolling, pipelining

Performance Improvements

Comparison of time per row, on 1 billion records on single thread

Primitive	Spark 1.6	Spark 2.0	Speedup Factor
filter	15ns	1.1ns	13.6
sum w/o group	14ns	0.9ns	15.6
sum w/ group	79ns	10.7ns	7.4
hash join	115ns	4.0ns	28.8
sort (8-bit)	620ns	5.3ns	117.0
sort (64-bit)	620ns	40ns	15.5
sort-merge-join	750ns	700ns	1.1

Source: https://databricks.com/blog/2016/07/26/introducing-apache-spark-2-0.html

Summary

Spark 1.x was already a great general purpose computing engine

Spark 2.0 takes it to a new level in several ways

2nd generation Tungsten engine provides 10X performance improvement

Unified APIs for Datasets and DataFrames and Spark SQL

Higher level ML APIs

Unified batch and streaming queries