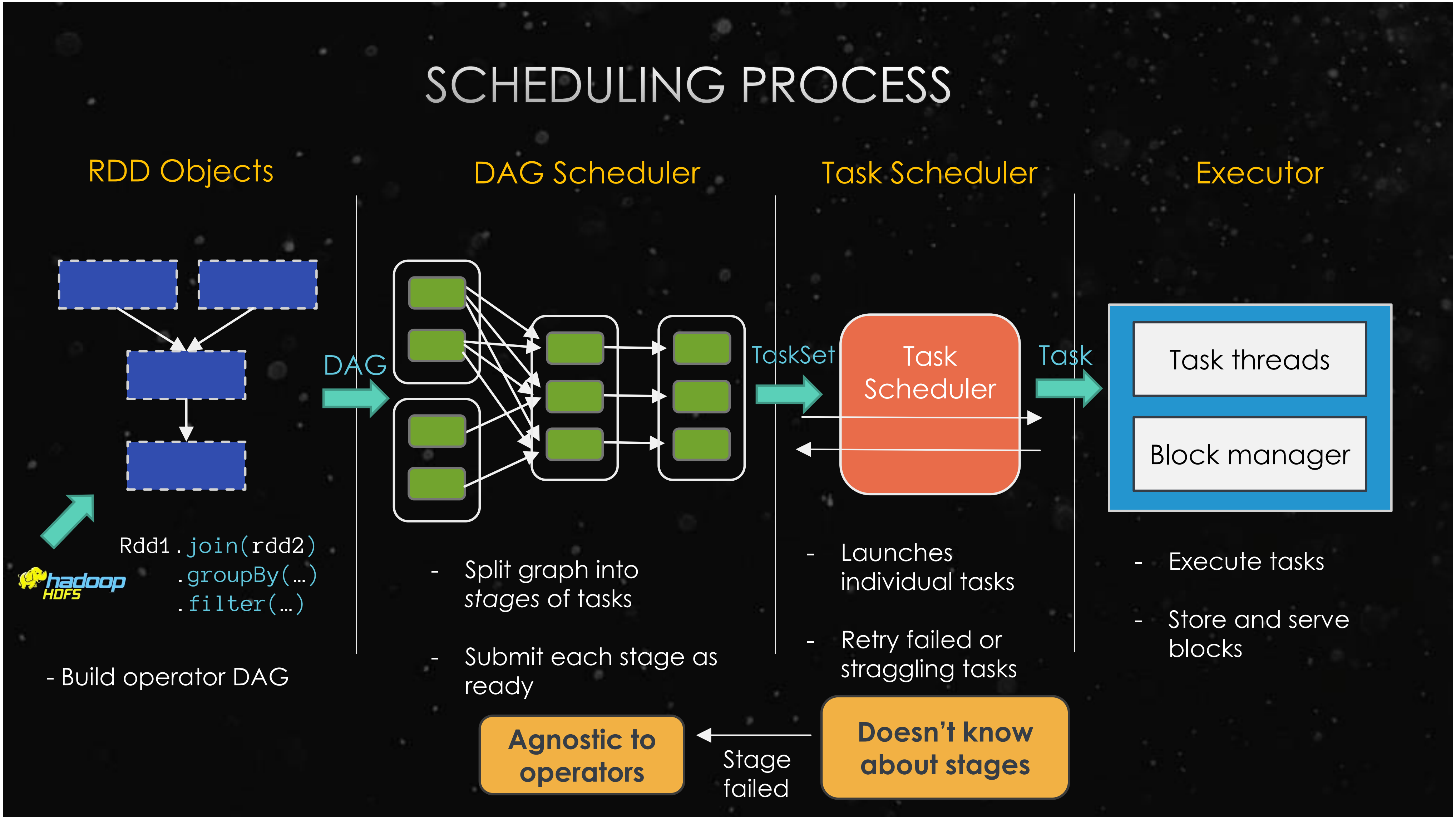
JOBS → STAGES → TASKS

databricks


.collect( )
Job #1
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Task #1
Task #2
Task #3



SCHEDULING PROCESS
RDD Objects
DAG Scheduler
Task Scheduler
Executor
DAG
TaskSet
Task
Task Scheduler
Task threads
Block manager
Rdd1.join(rdd2)
.groupBy(…)
.filter(…)
hadoop HDFS
- Build operator DAG
- Split graph into stages of tasks
- Submit each stage as ready
- Launches individual tasks
- Retry failed or straggling tasks
- Execute tasks
- Store and serve blocks
Agnostic to operators
Stage failed
Doesn't know about stages

LINEAGE

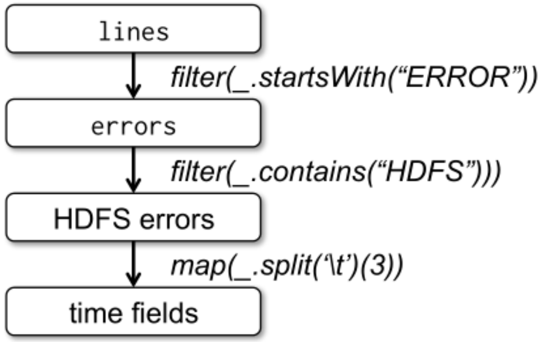


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

LINEAGE

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a

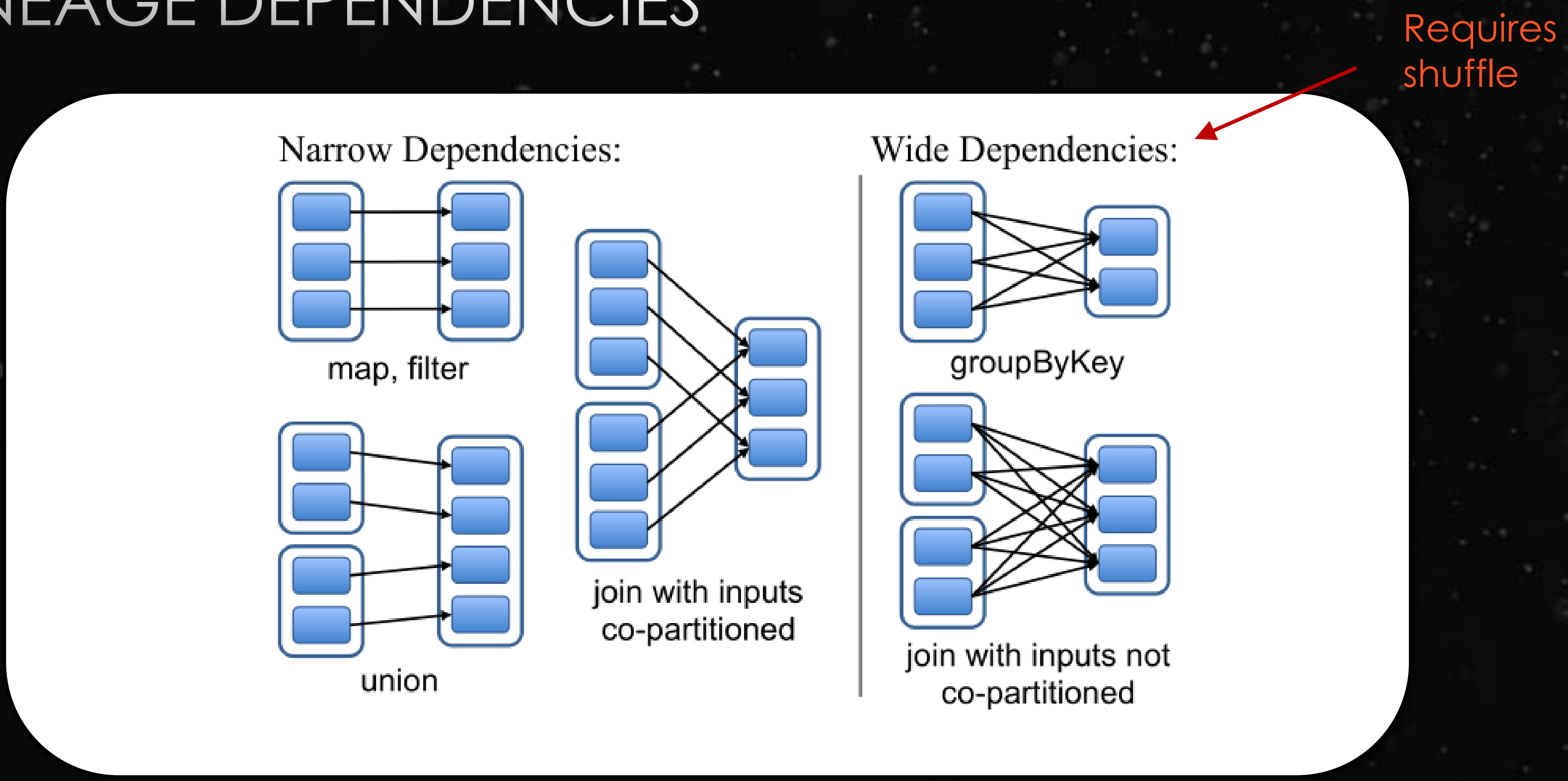
"One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations."

"The most interesting question in designing this interface is how to represent dependencies between RDDs."

"We found it both sufficient and useful to classify dependencies into two types:

- narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
- wide dependencies, where multiple child partitions may depend on it."

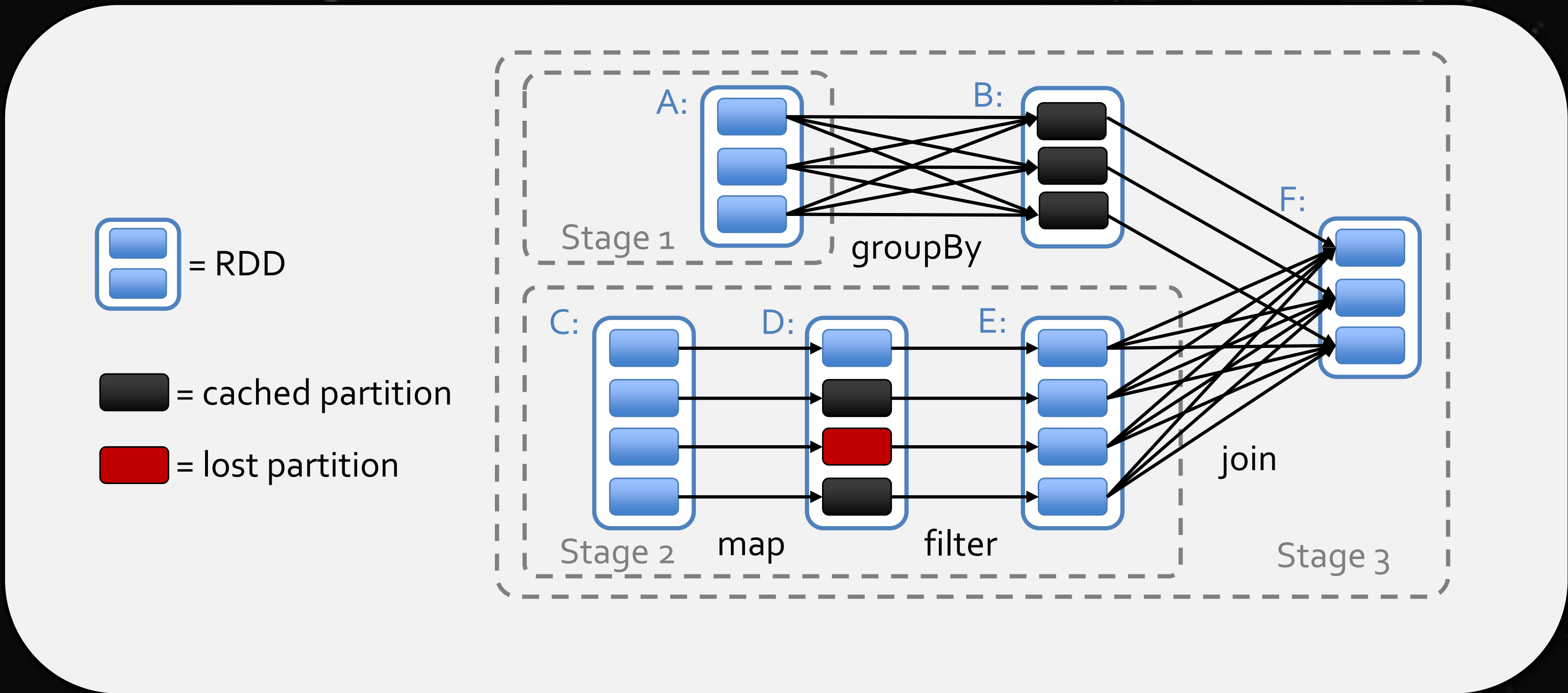
LINEAGE DEPENDENCIES



Examples of narrow and wide dependencies.

Each box is an RDD, with partitions shown as shaded rectangles.

STAGES



= RDD
= cached partition
= lost partition
A:
B:
Stage 1
groupBy
F:
C:
D:
E:
join
Stage 2
map
filter
Stage 3

LINEAGE

Dependencies: Narrow vs Wide

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

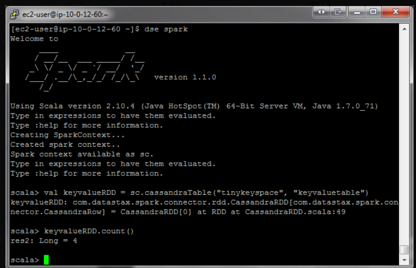
"This distinction is useful for two reasons:

1) Narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis.

In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation.

2) Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution."

To display the lineage of an RDD, Spark provides a **toDebugString** method:



```
scala> input.toDebugString

res85: String =
(2) data.text MappedRDD[292] at textFile at <console>:13
 | data.text HadoopRDD[291] at textFile at <console>:13

scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
 +-(2) MappedRDD[295] at map at <console>:17
    | FilteredRDD[294] at filter at <console>:15
    | MappedRDD[293] at map at <console>:15
    | data.text MappedRDD[292] at textFile at <console>:13
    | data.text HadoopRDD[291] at textFile at <console>:13
```

How do you know if a shuffle will be called on a Transformation?

- repartition , join, cogroup, and any of the *By or *ByKey transformations can result in shuffles
- If you declare a numPartitions parameter, it'll probably shuffle
- If a transformation constructs a shuffledRDD, it'll probably shuffle
- combineByKey calls a shuffle (so do other transformations like groupByKey, which actually end up calling combineByKey)

Note that repartition just calls coalese w/ True:

RDD.scala

```
def repartition(numPartitions: Int)(implicit
ord: Ordering[T] = null): RDD[T] = {
    coalesce(numPartitions, shuffle = true)
  }
```

How do you know if a shuffle will be called on a Transformation?

Transformations that use “numPartitions” like distinct will probably shuffle:

```
def distinct(numPartitions: Int)(implicit ord: Ordering[T] =
null): RDD[T] =
    map(x => (x, null)).reduceByKey((x, y) => x,
numPartitions).map(_._1)
```

PERSERVES PARTITIONING

- An extra parameter you can pass a k/v transformation to let Spark know that you will not be messing with the keys at all

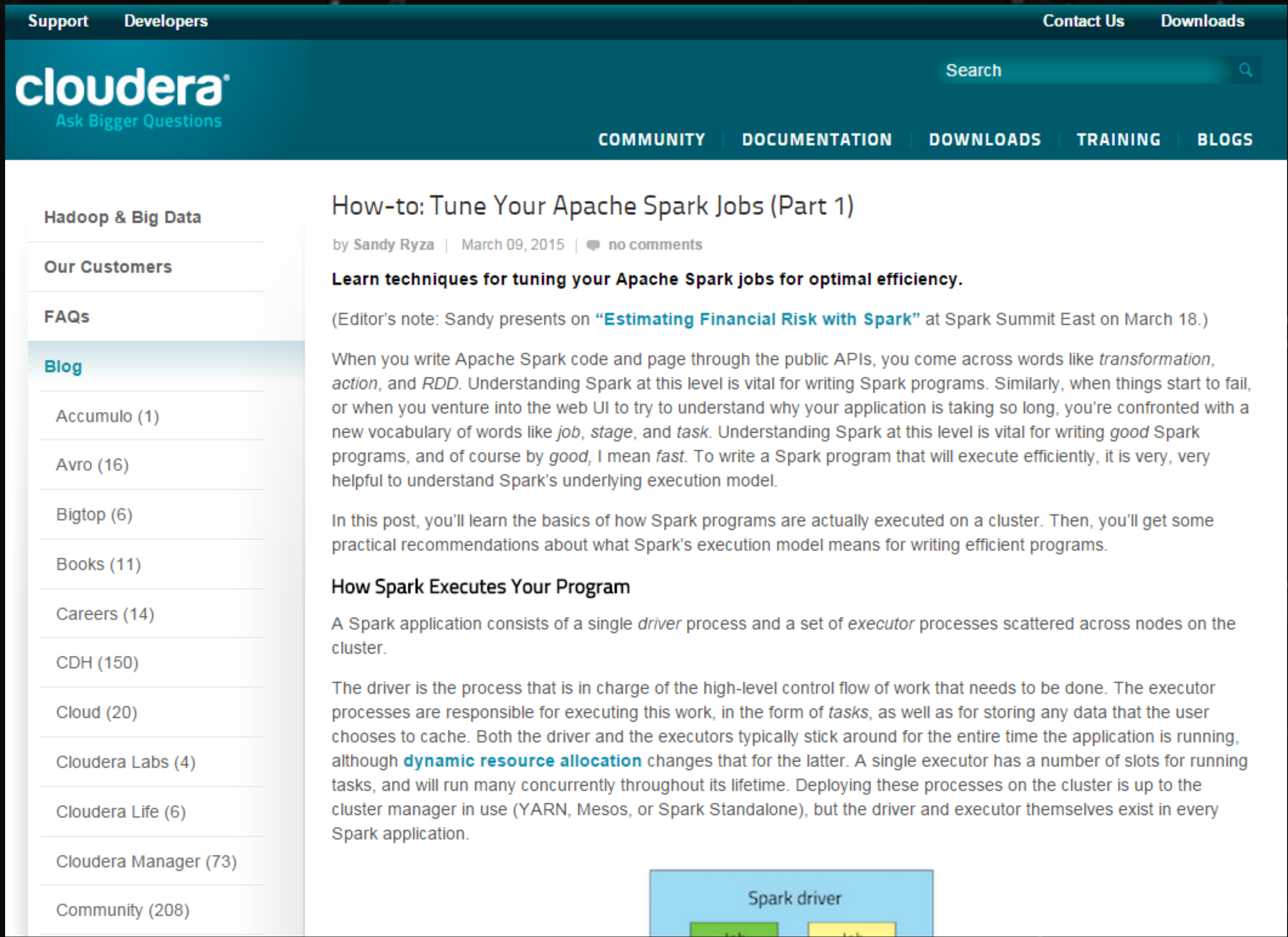
- All operations that shuffle data over network will benefit from partitioning

- Operations that benefit from partitioning:

`cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey, reduceByKey, combineByKey, lookup, . . .`

https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L302

```scala
/**
 * Return a new RDD containing only the elements that satisfy a predicate.
 */
def filter(f: T => Boolean): RDD[T] = {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[T, T](
    this,
    (context, pid, iter) => iter.filter(cleanF),
    preservesPartitioning = true)
}
```



Support Developers Contact Us Downloads

cloudera
Ask Bigger Questions

Search

COMMUNITY DOCUMENTATION DOWNLOADS TRAINING BLOGS

- Hadoop & Big Data
- Our Customers
- FAQs
- Blog
  - Accumulo (1)
  - Avro (16)
  - Bigtop (6)
  - Books (11)
  - Careers (14)
  - CDH (150)
  - Cloud (20)
  - Cloudera Labs (4)
  - Cloudera Life (6)
  - Cloudera Manager (73)
  - Community (208)

How-to: Tune Your Apache Spark Jobs (Part 1)

by Sandy Ryza | March 09, 2015 | no comments

Learn techniques for tuning your Apache Spark jobs for optimal efficiency.

(Editor's note: Sandy presents on **"Estimating Financial Risk with Spark"** at Spark Summit East on March 18.)

When you write Apache Spark code and page through the public APIs, you come across words like *transformation*, *action*, and *RDD*. Understanding Spark at this level is vital for writing Spark programs. Similarly, when things start to fail, or when you venture into the web UI to try to understand why your application is taking so long, you're confronted with a new vocabulary of words like *job*, *stage*, and *task*. Understanding Spark at this level is vital for writing *good* Spark programs, and of course by *good*, I mean *fast*. To write a Spark program that will execute efficiently, it is very, very helpful to understand Spark's underlying execution model.

In this post, you'll learn the basics of how Spark programs are actually executed on a cluster. Then, you'll get some practical recommendations about what Spark's execution model means for writing efficient programs.

How Spark Executes Your Program

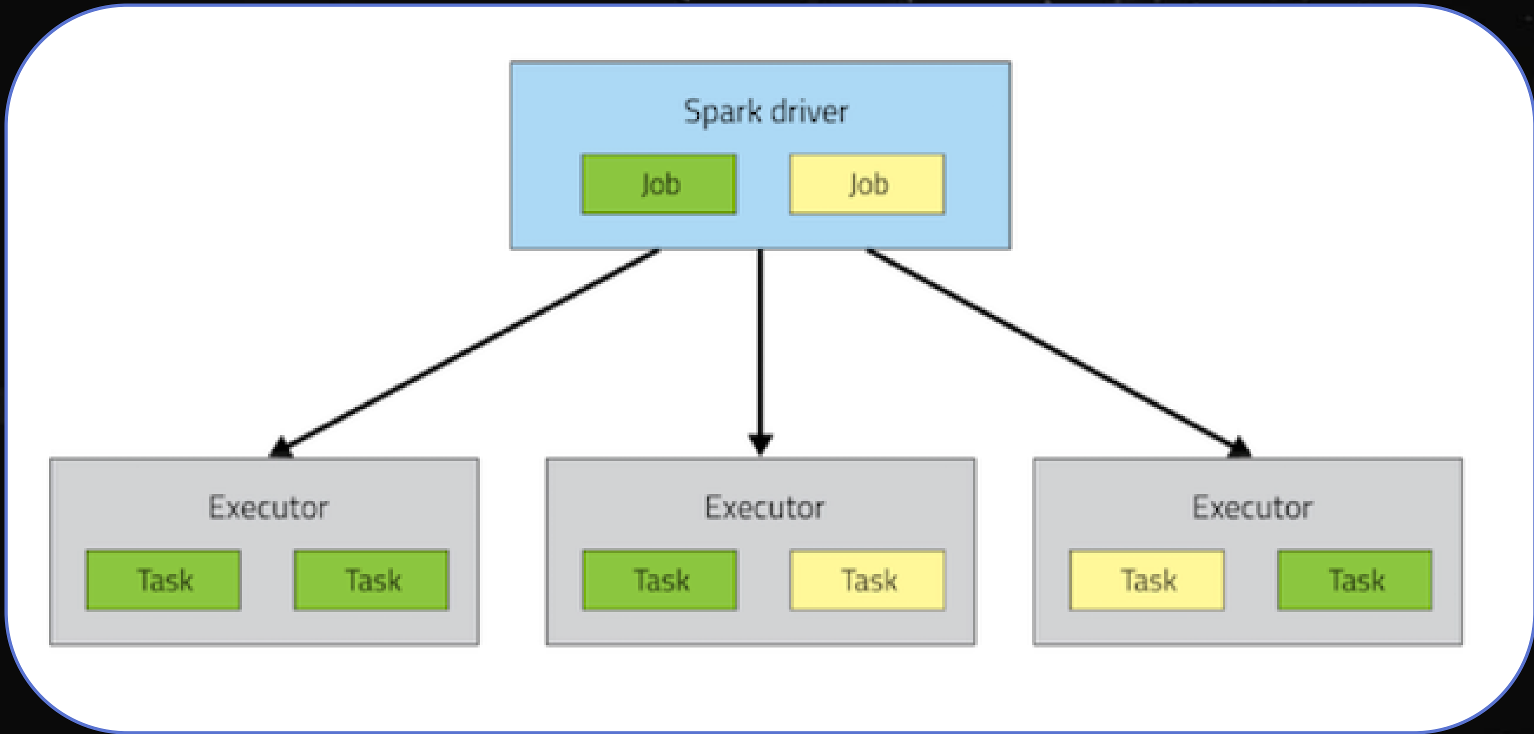
A Spark application consists of a single *driver* process and a set of *executor* processes scattered across nodes on the cluster.

The driver is the process that is in charge of the high-level control flow of work that needs to be done. The executor processes are responsible for executing this work, in the form of *tasks*, as well as for storing any data that the user chooses to cache. Both the driver and the executors typically stick around for the entire time the application is running, although **dynamic resource allocation** changes that for the latter. A single executor has a number of slots for running tasks, and will run many concurrently throughout its lifetime. Deploying these processes on the cluster is up to the cluster manager in use (YARN, Mesos, or Spark Standalone), but the driver and executor themselves exist in every Spark application.

Spark driver

Link

Source: Cloudera



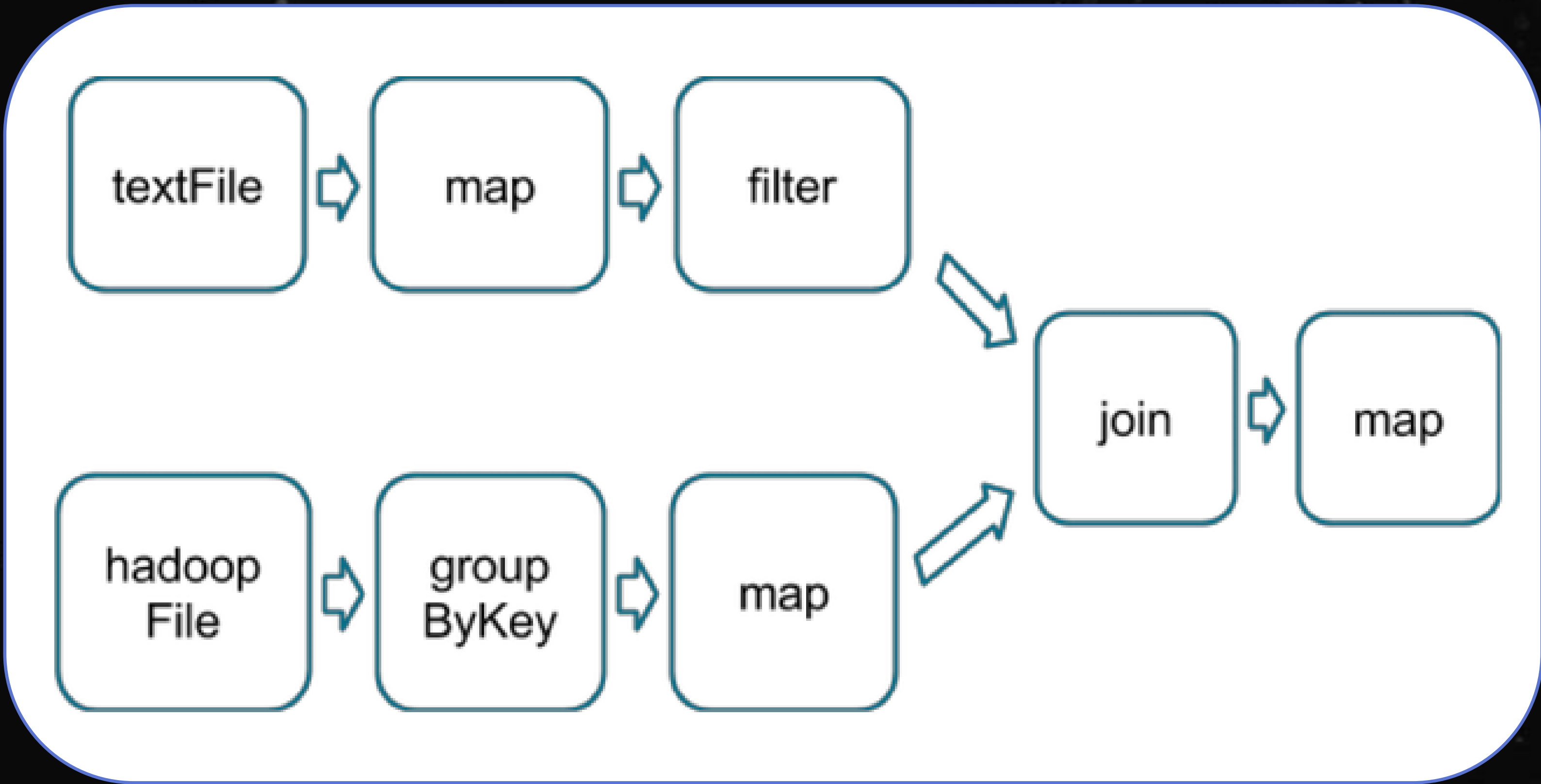
Source: Cloudera

How many Stages will this code require?

```
sc.textFile("someFile.txt").
  map(mapFunc).
  flatMap(flatMapFunc).
  filter(filterFunc).
  count()
```

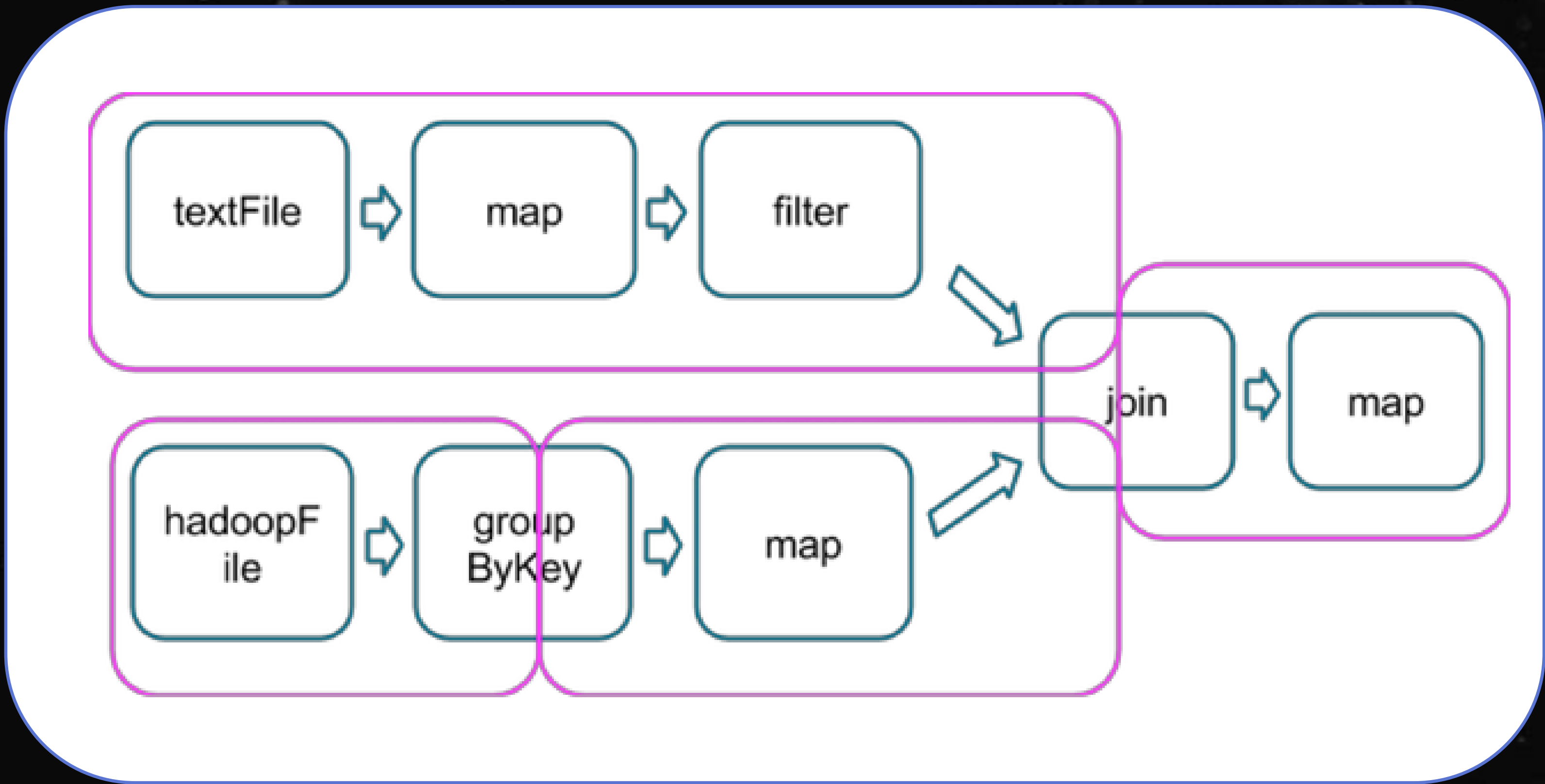
Source: Cloudera

How many Stages will this DAG require?



Source: Cloudera

How many Stages will this DAG require?



Source: Cloudera