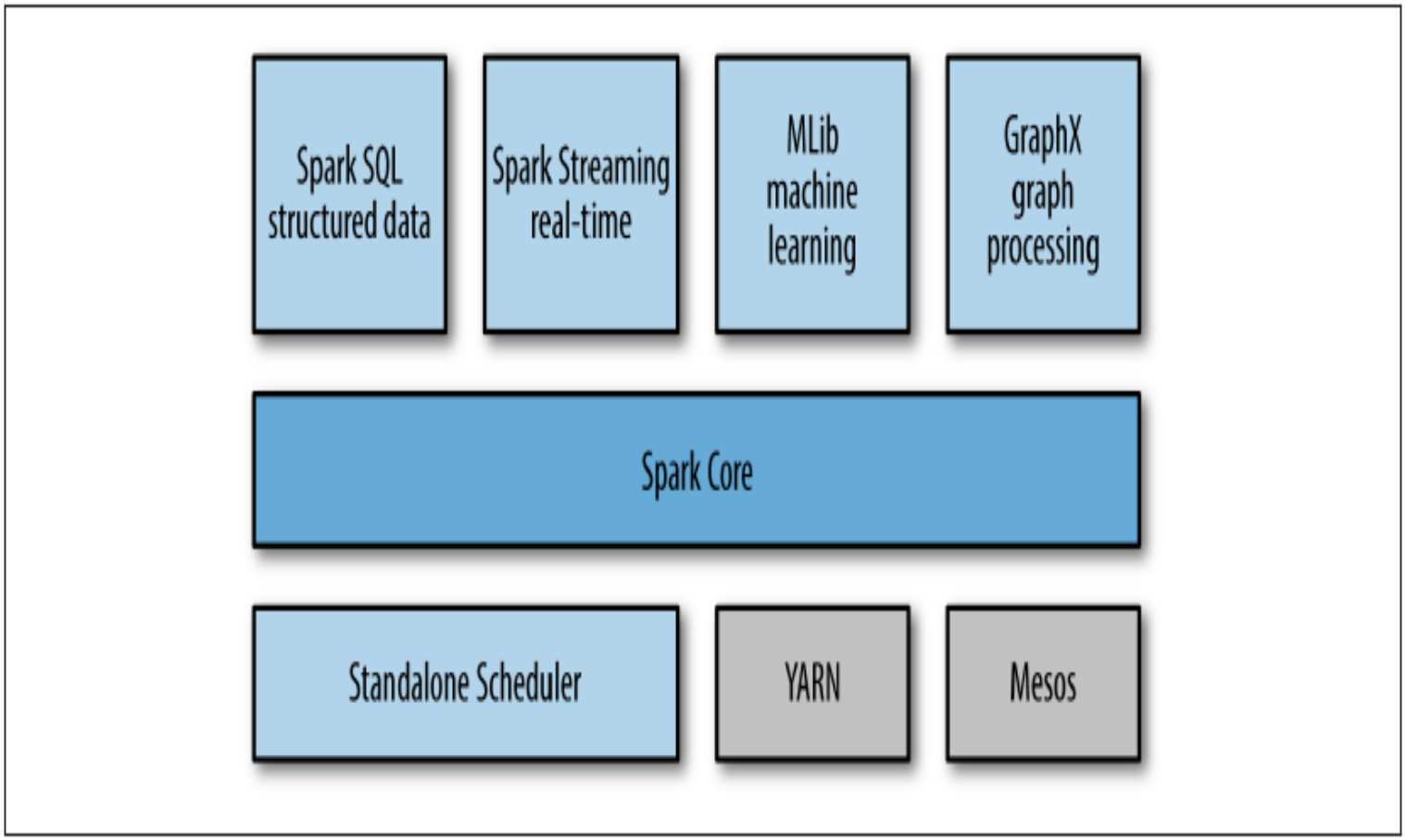
Spark

Unit V

Introduction

- Apache Spark is a cluster computing platform designed to be *fast and general purpose.*
- On the speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing.
- Speed is important in processing large datasets, One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also more efficient than MapReduce for complex applications running on disk.
- On the generality side, Spark is designed to cover a wide range of workloads : separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming.
- By supporting these workloads in the same engine, Spark makes it easy and inexpensive to *combine different processing* types, which is often necessary in production data analysis pipelines.
- In addition, it reduces the management burden of maintaining separate tools.
- Spark is designed to be highly accessible, offering simple APIs in Python, Java, Scala, and SQL, and rich built-in libraries.
- It also integrates closely with other Big Data tools. Hadoop clusters and access any Hadoop data source, including Cassandra.

The Spark Stack



Spark

- At its core, Spark is a “computational engine” that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a *computing cluster.*
- *Because the core engine of Spark is both fast and* general-purpose, it powers multiple higher-level components specialized for various workloads, such as SQL or machine learning. These components are designed to interoperate closely, letting you combine them like libraries in a software project.

Spark Core

- Spark Core contains the basic functionality of Spark, including
- components for task scheduling,
- memory management,
- fault recovery,
- interacting with storage systems, and more.
- Spark Core is also home to the API that defines *resilient distributed datasets* (RDDs), which are Spark's main programming abstraction.
- RDDs Resilient Distributed Datasets
- RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections.

Spark SQL

- Spark SQL is Spark's package for working with structured data.
- It allows querying data via SQL as well as the Apache Hive variant of SQL—called the Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet,and JSON.
- Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics.
- This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike Shark was an older SQL-on-Spark project out of the University of California, Berkeley, that modified Apache Hive to run on Spark.
- It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs.

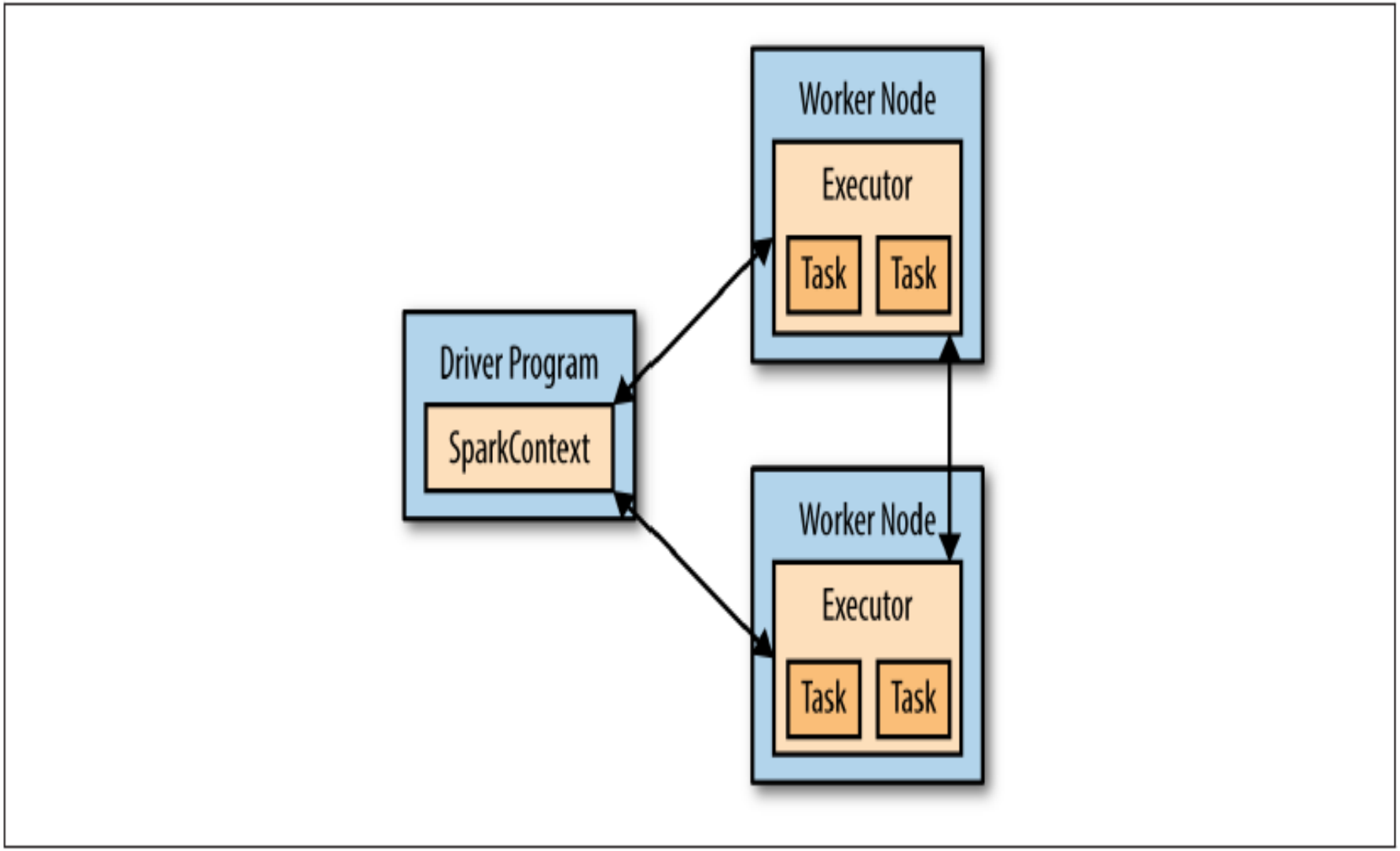
Spark Streaming

- Spark Streaming is a Spark component that enables processing of live streams of data.
- Examples of data streams include log files generated by production web servers, or queues of messages containing status updates posted by users of a web service. Spark Streaming provides an API for manipulating data streams that closely matches the
- Spark Core's RDD API, making it easy for programmers to learn the project and
- move between applications that manipulate data stored in memory, on disk, or arriving
- in real time. Underneath its API, Spark Streaming was designed to provide the
- same degree of fault tolerance, throughput, and scalability as Spark Core.

MLlib

- Spark comes with a library containing common machine learning (ML) functionality, called MLlib. MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import. It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm. All of these methods are designed to scale out across a cluster.
- GraphX
- GraphX is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting).
- Cluster Managers
- Under the hood, Spark is designed to efficiently scale up from one to many thousands of compute nodes. To achieve this while maximizing flexibility, Spark can run over a variety of *cluster managers, including Hadoop YARN, Apache Mesos, and a simple* cluster manager included in Spark itself called the Standalone Scheduler. If you are just installing Spark on an empty set of machines, the Standalone Scheduler provides an easy way to get started; if you already have a Hadoop YARN or Mesos cluster, however, Spark's support for these cluster managers allows your applications to also run on them.

Components for Distributed execution in Spark



Programming with RDDs

- An RDD is simply a distributed collection of elements. In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result.
- Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them.

RDD

- An RDD in Spark is simply an immutable distributed collection of objects.
- Each RDD is split into multiple *partitions, which may be computed on different nodes of the* cluster.
- RDDs can contain any type of Python, Java, or Scala objects, including userdefined classes.

- Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program.
- We have already seen loading a text file as an RDD of strings using SparkContext.textFile(). *Creating an RDD of strings with textFile() in Python*
- lines = sc.textFile("README.md“)

Operations on RDDs

- *Transformations*
  - *Transformations construct a new RDD from a previous one.*
- *Actions.*
  - *Actions compute a result based on an RDD, and either return it to* the driver program or save it to an external storage system (e.g., HDFS).

Creating RDDs RDD.persist()

- Spark provides two ways to create RDDs: loading an external dataset and parallelizing a collection in your driver program.
- lines = sc.parallelize(["pandas", "i like pandas"])
- lines = sc.textFile("/path/to/README.md")
- If you would like to reuse an RDD in multiple actions, you can ask Spark to *persist it using RDD.persist().*

- RDD Operations
- *Transformations are operations on RDDs that return a new RDD, such as* map() and filter().
- Actions are operations that return a result to the driver program or write it to storage, and kick off a computation, such as count() and first().

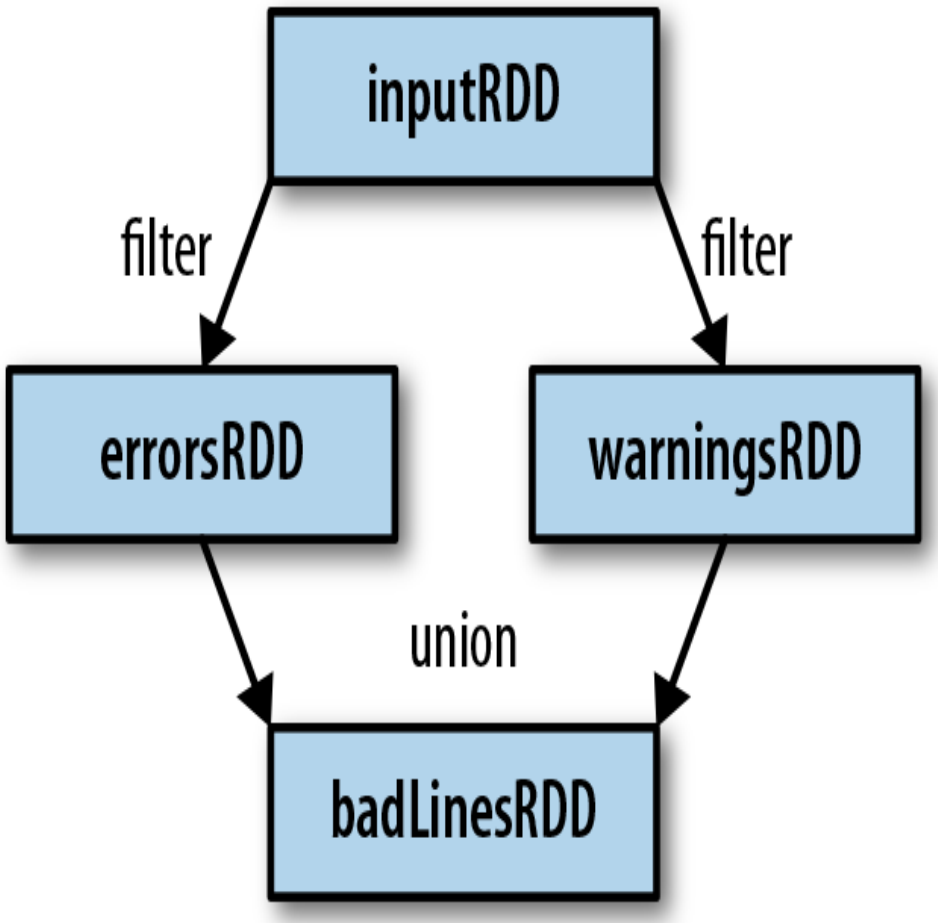
To select only error messages from logfile

- In python
- inputRDD = sc.textFile("log.txt")
- errorsRDD = inputRDD.filter(**lambda x: "error" in x)**

- **In scala**
- **val inputRDD = sc.textFile("log.txt")**
- **val errorsRDD = inputRDD.filter(line => line.contains("error"))**

to print out the number of lines that contained either

- errorsRDD = inputRDD.filter(**lambda x: "error" in x)**
- warningsRDD = inputRDD.filter(**lambda x: "warning" in x)**
- badLinesRDD = errorsRDD.union(warningsRDD)
- union() is a bit different than filter(), in that it operates on two RDDs instead of one.



RDD actions

- Actions force the evaluation of the transformations required for the RDD they were called on, since they need to actually produce output.
- **print "Input had " + badLinesRDD.count() + " concerning lines"**
- **print "Here are 10 examples:"**
- **for line in badLinesRDD.take(10):**
- **print line**

Lazy Evaluation

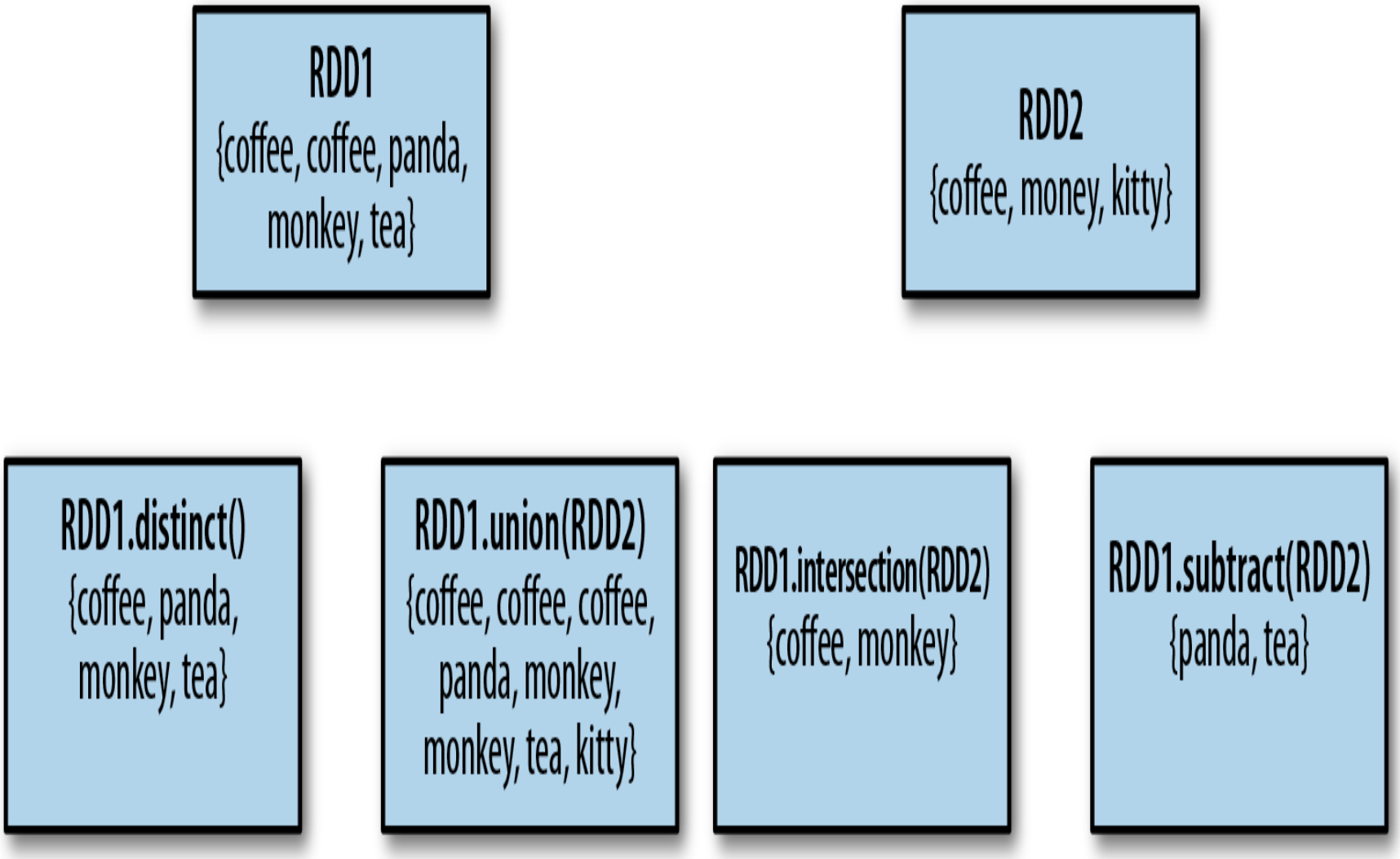
- Transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.
- Lazy evaluation means that when we call a transformation on an RDD (for instance, calling map()), the operation is not immediately performed.
- Instead, Spark internally records metadata to indicate that this operation has been requested. Rather than thinking of an RDD as containing specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations.
- Loading data into an RDD is lazily evaluated in the same way transformations are.

To get squares of numbers

- nums = sc.parallelize([1, 2, 3, 4])
- squared = nums.map(**lambda x: x * x).collect()**
- **for num in squared:**
- **print "%i " % (num)**

- Sometimes we want to produce multiple output elements for each input element. The operation to do this is called flatMap(). As with map(), the function we provide to flatMap() is called individually for each element in our input RDD.
- Instead of returning a single element, we return an iterator with our return values.
- Rather than producing an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators.
- lines = sc.parallelize(["hello world", "hi"])
- words = lines.flatMap(**lambda line**

Set operations



- collect() Return all elements from the RDD. rdd.collect() {1, 2, 3, 3}
- count() Number of elements in the RDD.
- rdd.count() 4
- countByValue() Number of times each element
- occurs in the RDD.
- rdd.countByValue() {(1, 1),(2, 1),(3, 2)}

Creating Pair RDDs

- *Creating a pair RDD using the first word as the key in Python*
- pairs = lines.map(**lambda x: (x.split(" ")[0], x))**

Transformations on Pair RDDs

reduceByKey(func)	**Combine values with the same key.**	**rdd.reduceByKey((x, y) => x + y)**	**{(1, 2), (3, 10)}**
groupByKey()	Group values with the same key.	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
combineBy Key(createCombine r, mergeValue, mergeCombiners, partitioner)	Combine values with the same key using a different result type.		
mapValues(func)	Apply a function to	rdd.mapValues(x =>	{(1,

Transformation on pair RDDs

Function name	Purpose	Example	Result
subtractByKey	Remove elements with a key present in the other RDD.	rdd.subtractByKey(other)	{(1, 2)}
join	Perform an inner join between two RDDs.	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Perform a join between two RDDs where the key must be present in the first RDD.	rdd.rightOuterJoin(other)	{(3,(Some(4),9)), (3,(Some(6),9))}
leftOuterJoin	Perform a join between two RDDs where the key must be present in the other RDD.	rdd.leftOuterJoin(other)	{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}
cogroup	Group data from both RDDs sharing the same key.	rdd.cogroup(other)	{(1,([2],[])), (3, ([4, 6],[9]))}

Actions on pair RDDs

Function	Description	Example	Result
countByKey()	Count the number of elements for each key.	rdd.countByKey()	{(1, 1), (3, 2)}
collectAsMap()	Collect the result as a map to provide easy lookup.	rdd.collectAsMap()	Map{(1, 2), (3, 4), (3, 6)}
lookup(key)	Return all values associated with the provided key.	rdd.lookup(3)	[4, 6]