Université Constantine 2
جامعة قسنطينة 2

Big Data Processing and Analysis

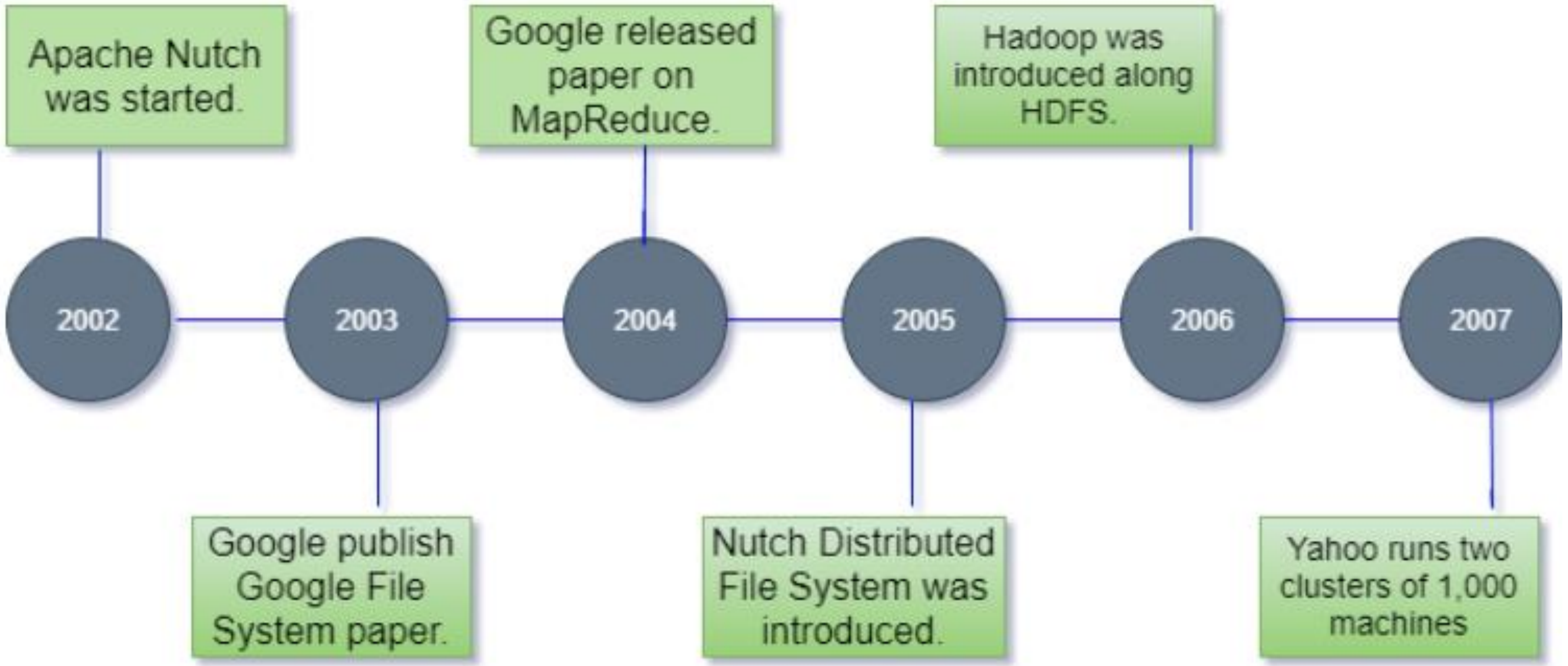
Reminder: Spark

Dr. Rostom Mennour
Faculty of New Technologies
rostom.mennour@univ-constantine2.dz

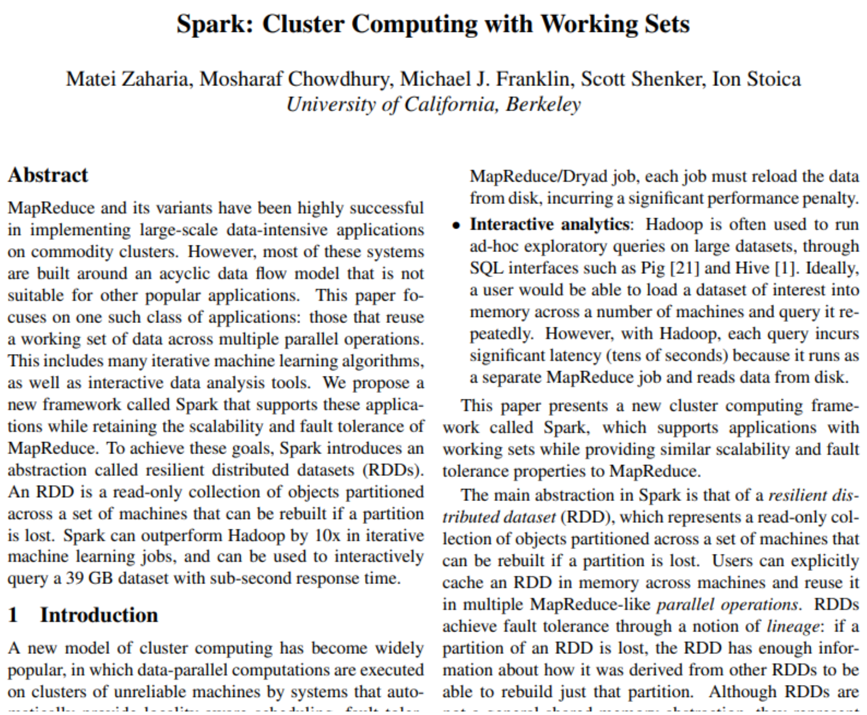
Etudiants concernés

Faculté/Institut	Département	Niveau	Spécialité
Nouvelles technologies	IFA	Master 2	SDIA

History


Apache Nutch was started.
2002
2003
Google publish Google File System paper.
Google released paper on MapReduce.
2004
2005
Nutch Distributed File System was introduced.
Hadoop was introduced along HDFS.
2006
2007
Yahoo runs two clusters of 1,000 machines

History



Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

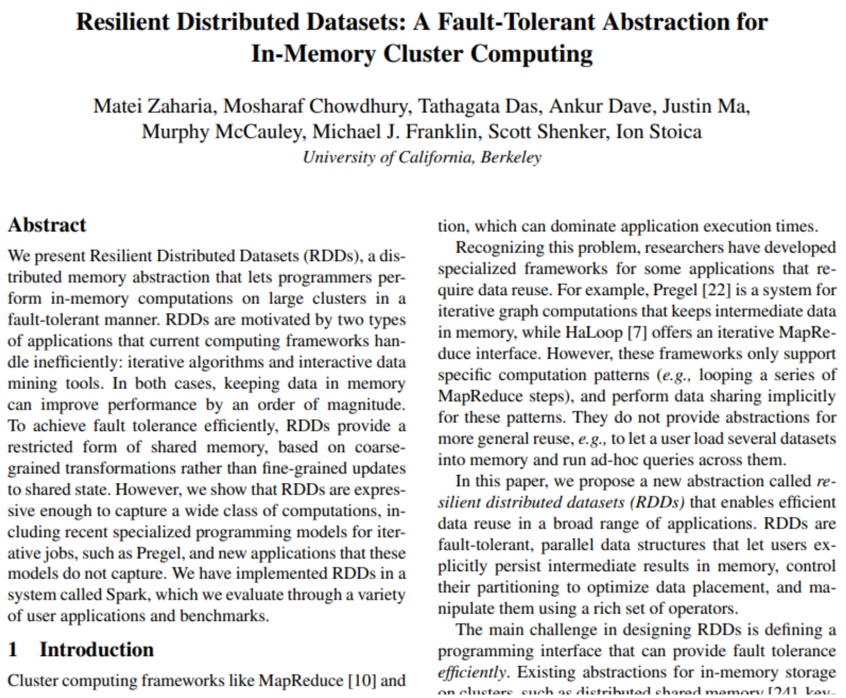
A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that auto-

MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.

- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are



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

tion, which can dominate application execution times.

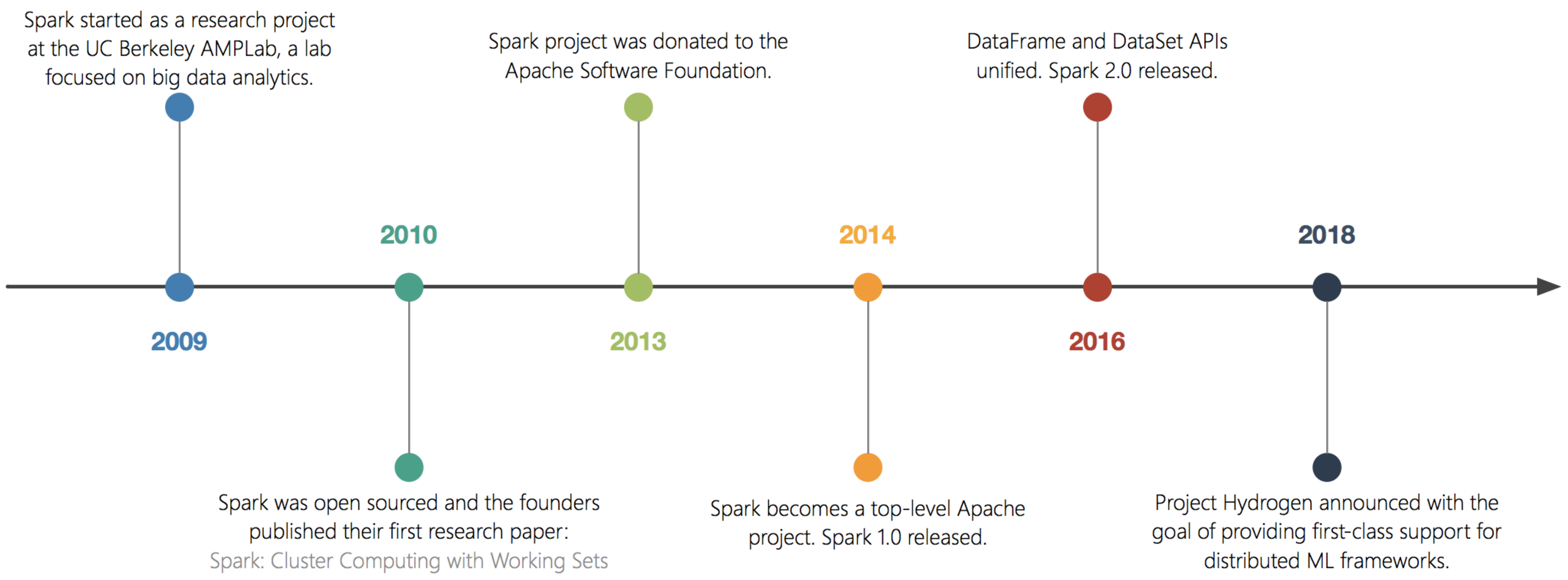
Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these 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 programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage

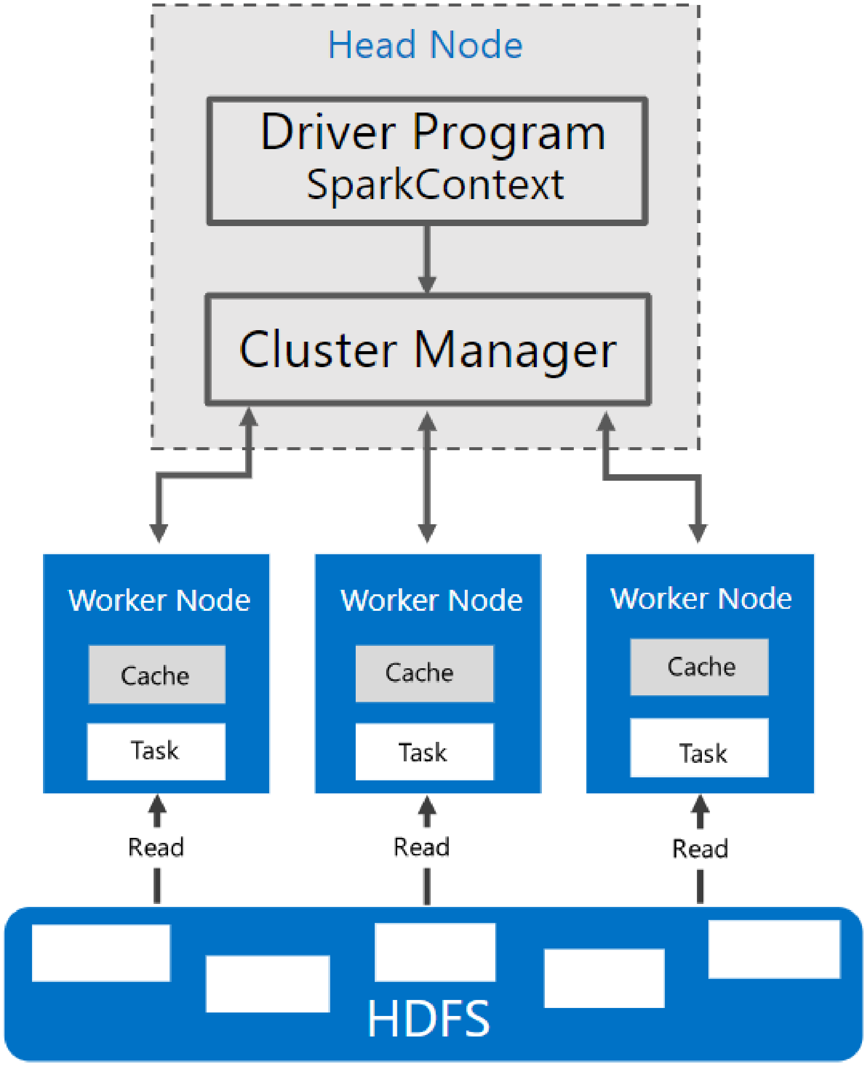
- *Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. In 2nd USENIX workshop on hot topics in cloud computing (HotCloud 10).*
- *Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., ... & Stoica, I. (2012). Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In 9th USENIX symposium on networked systems design and implementation (NSDI 12) (pp. 15-28).*

History


Spark started as a research project at the UC Berkeley AMPLab, a lab focused on big data analytics.
2009
2010
Spark was open sourced and the founders published their first research paper: Spark: Cluster Computing with Working Sets
Spark project was donated to the Apache Software Foundation.
2013
2014
Spark becomes a top-level Apache project. Spark 1.0 released.
DataFrame and DataSet APIs unified. Spark 2.0 released.
2016
2018
Project Hydrogen announced with the goal of providing first-class support for distributed ML frameworks.

Spark Core Concepts and Architecture

- **Spark clusters**
- **Resource management system**
- **Spark applications**
- **Spark drivers**
- **Spark executors**



Spark Core Concepts and Architecture

Spark cluster and resource management system

- Spark is a distributed system, it runs on a collection of machines called Cluster
- A cluster can be as small as a few machines or as large as thousands of machines.
- Companies rely on a resource management system like Apache YARN or Apache Meso to efficiently and intelligently manage a collection of machines.
- the world’s largest Spark cluster has more than 8000 machines.

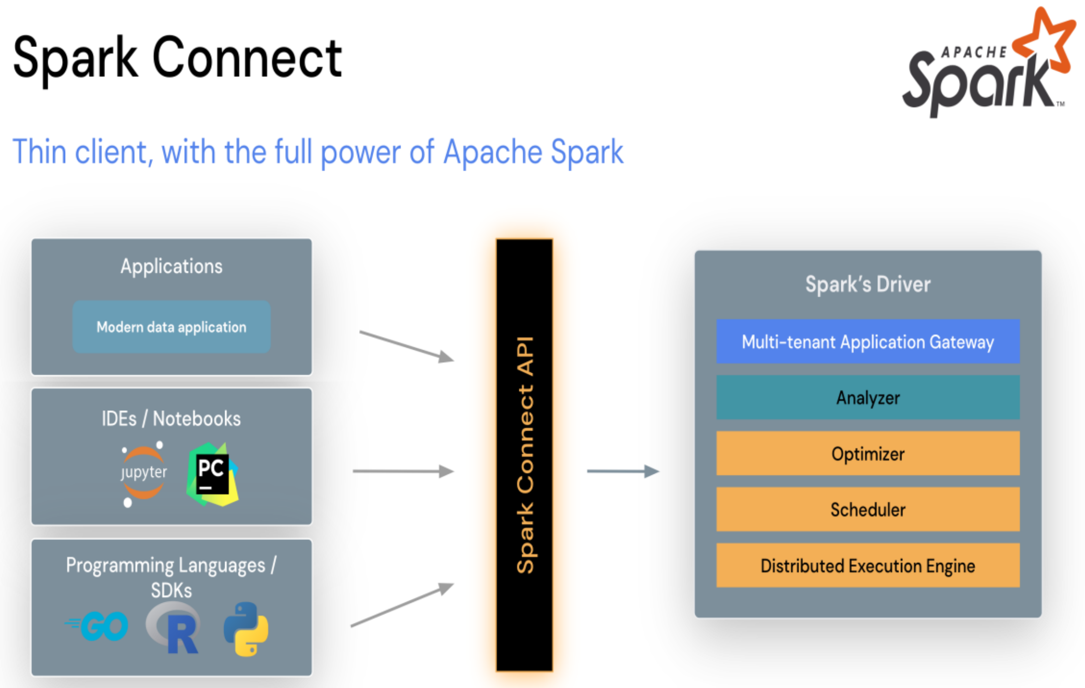
Spark Core Concepts and Architecture

Spark applications

A Spark application consists of two parts.

Data processing logic expressed using Spark APIs.

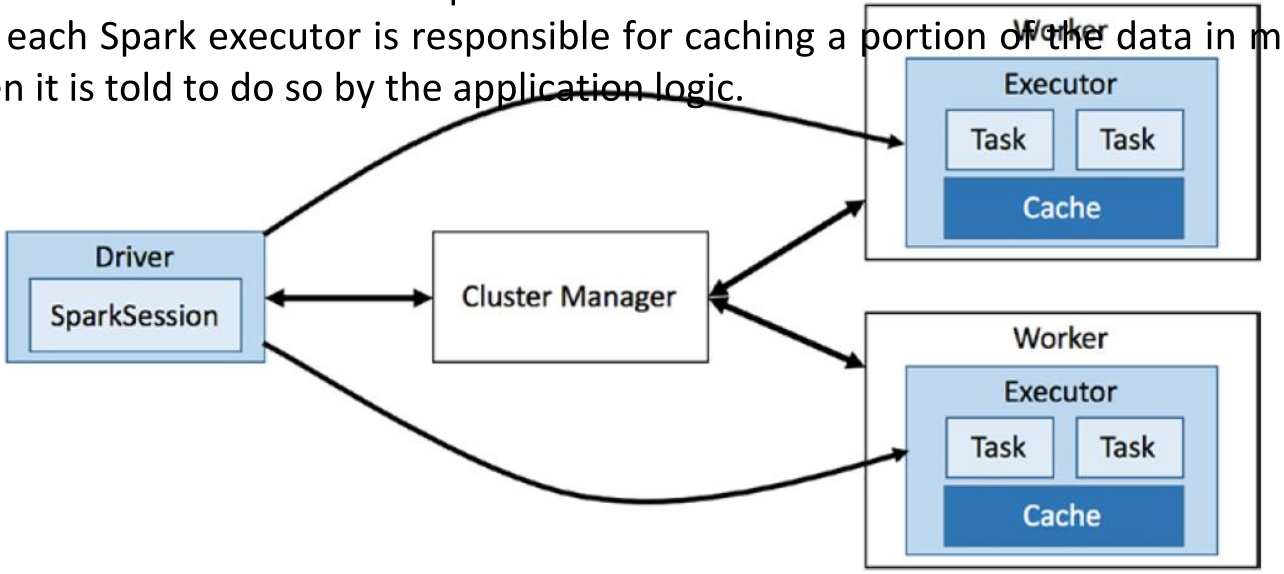
Driver: is the central coordinator of a Spark application to interact with a cluster manager to figure out which machines to run the data processing logic.



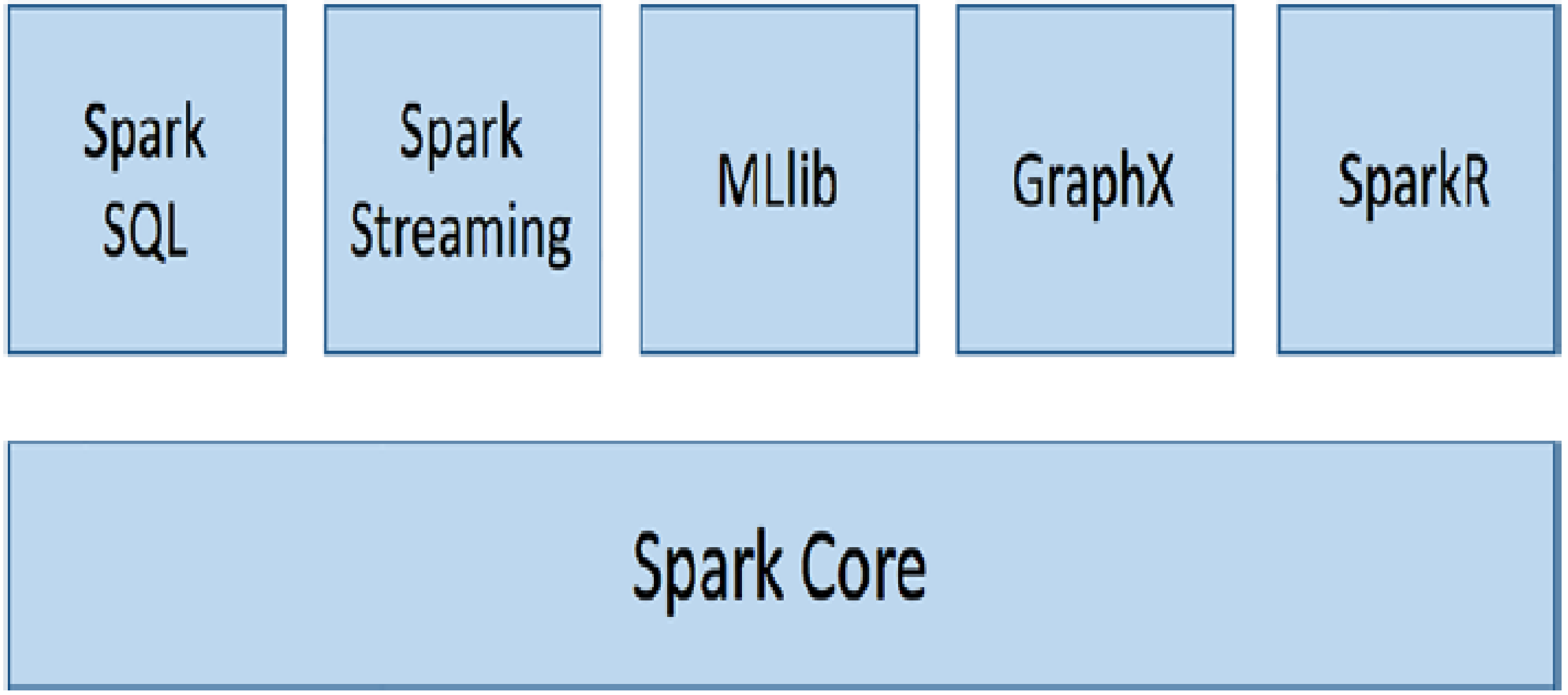
Spark Core Concepts and Architecture

Spark driver and executors

- Spark employs a master/slave architecture, where the driver is the master, and the executor is the slave. Each of these components runs as an independent process on a Spark cluster. A Spark application consists of one driver and one or more executors.
- Each Spark executor is a JVM process and is dedicated to a specific Spark application.
- The life span of a Spark executor is the duration of a Spark application.
- There was a conscious design decision not to share a Spark executor between different multiple Spark applications.
- A Spark executor does what is being told, which is to execute the data processing logic in the form of tasks. Each task is executed on a separate CPU core.
- In addition, each Spark executor is responsible for caching a portion of the data in memory and/or on disk when it is told to do so by the application logic.



Spark unified stack



Spark unified stack

Spark core

- Spark Core is the bedrock of the Spark distributed data processing engine. It consists of
- an RDD, a distributed computing infrastructure and programming abstraction.
- The distributed computing infrastructure is responsible for distributing, coordinating, and scheduling computing tasks across many machines in the cluster.
- Two other important responsibilities of the distributed computing infrastructure are handling computing task failures and the efficient way of moving data across machines, known as data shuffling.
- Advanced Spark users should have intimate knowledge of Spark distributed computing infrastructure to effectively design high-performance Spark applications.
- The RDD key programming abstraction is a fault-tolerant collection of objects partitioned across a cluster that can be manipulated in parallel.
- Essentially it provides a set of APIs for Spark application developers to easily and efficiently perform large-scale data processing without worrying where data resides on the cluster and
- machine failures.
- The RDD APIs are exposed to multiple programming languages, including Scala, Java, and Python. They allow users to pass local functions to run on the cluster, which is very powerful and unique. RDDs are covered in detail in a later chapter.

Spark unified stack

Spark SQL

- Spark SQL is a module built on top of Spark Core, and it is designed for structured data processing at scale.
- Spark users can issue SQL queries to perform data processing or use the high-level abstraction exposed through the DataFrame API.
- A DataFrame is effectively a distributed collection of data organized into named columns. This is not a new idea. It is inspired by data frames in R and Python. An easier way to think about a DataFrame is that it is conceptually equivalent to a table in a relational database.
- Behind the scenes, the Spark SQL Catalyst optimizer performs optimizations commonly done in many analytical database engines.
- Another Spark SQL feature that elevates Spark's flexibility is the ability to read and write data to and from various structured formats and storage systems, such as JavaScript Object Notation (JSON), comma-separated values (CSV), Parquet or ORC files, relational databases, Hive, and others.

Spark unified stack

Spark Structured Streaming

- The Spark Structured Streaming module enables the ability to process real-time streaming data from various data sources in a high-throughput and fault-tolerant manner.
- Data can be ingested from sources like Kafka, Flume, Kinesis, Twitter, HDFS, or TCP socket.
- Spark's main abstraction for processing streaming data is a discretized stream (DStream), which implements an incremental stream processing model by splitting the input data into small batches (based on a time interval) that can regularly combine the current processing state to produce new results.
- A new scalable and fault-tolerant stream processing engine called Structured Streaming was introduced in Spark version 2.1. This engine further simplifies stream processing app developers' lives by treating streaming computation the same way as you express a batch computation on static data. This new engine automatically executes the stream processing logic incrementally and continuously and produces the result as new streaming data arrives.
- Another unique feature in the Structured Streaming engine is the guarantee of end-to-end exactly-once support, which makes "big data" engineer's life much easier than before in terms of saving data to a storage system like a relational database or a NoSQL database.

Spark unified stack

MLLib

- MLlib is Spark’s machine learning library. It provides more than 50 common machine learning algorithms and abstractions for managing and simplifying model-building tasks, such as featurization, a pipeline for constructing, an evaluating and tuning model, and the persistence of models to help move models from development to production.
- Starting with Spark 2.0 version, the MLlib APIs are based on DataFrames to take advantage of the user-friendliness and many optimizations provided by the Catalyst and Tungsten components in the Spark SQL engine.
- Machine learning algorithms are iterative, meaning they run through many iterations until the desired objective is achieved. Spark makes it extremely easy to implement those algorithms and run them in a scalable manner through a cluster of machines.
- Commonly used machine learning algorithms such as classification, regression, clustering, and collaborative filtering are available out of the box for data scientists and engineers to use.

Spark unified stack

GraphX

- Graph processing operates on a data structure consisting of vertices and edges connecting them. A graph data structure is often used to represent real-life networks of interconnected entities, including professional social networks on LinkedIn, a network of connected web pages on the Internet, and so on.
- Spark GraphX is a library that enables graph-parallel computations by providing an abstraction of a directed multi-graph with properties attached to each vertex and edge.
- GraphX includes a collection of common graph processing algorithm, including page ranks, connected components, shortest paths, and others.

Spark unified stack

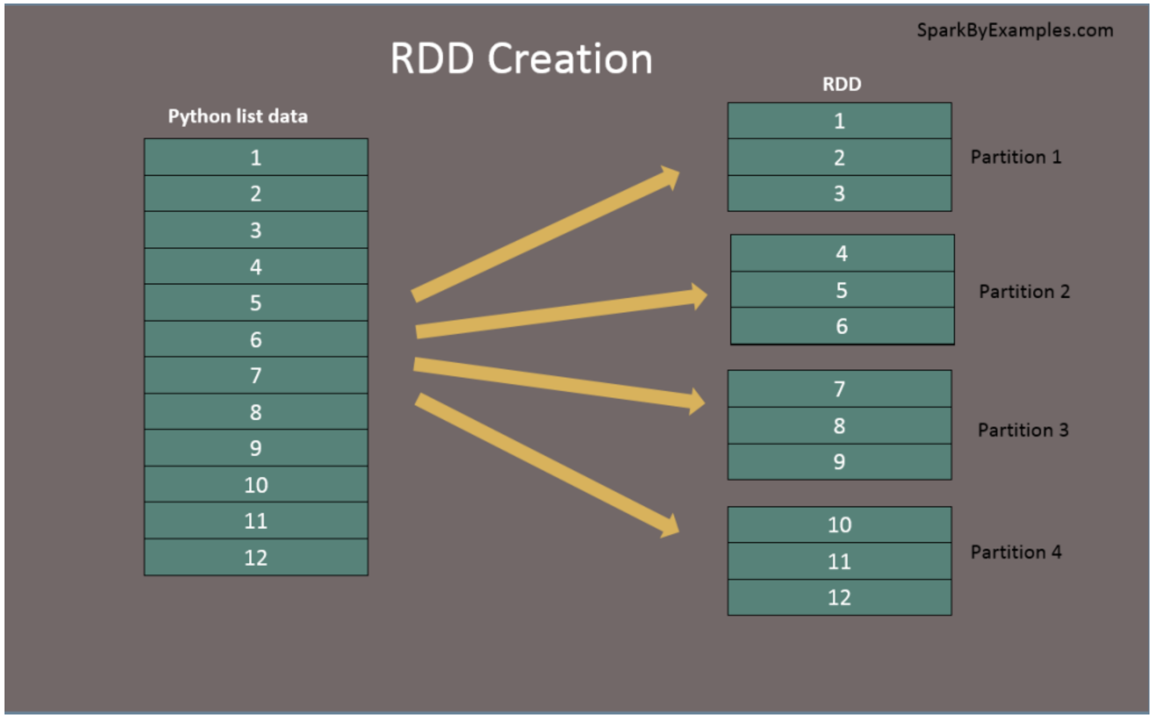
SparkR

- SparkR is an R package that provides a lightweight frontend to use Apache Spark.
- R is a popular statistical programming language that supports data processing and machine learning tasks.
- However, R was not designed to handle large datasets that cannot fit on a single machine.
- SparkR leverages Spark’s distributed computing engine to enable large-scale data analysis using familiar R shell and popular APIs that many data scientists love.

Spark RDDs

Spark revolves around the concept of a *resilient distributed dataset* (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs:

- *parallelizing* an existing collection in your driver program,
- or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

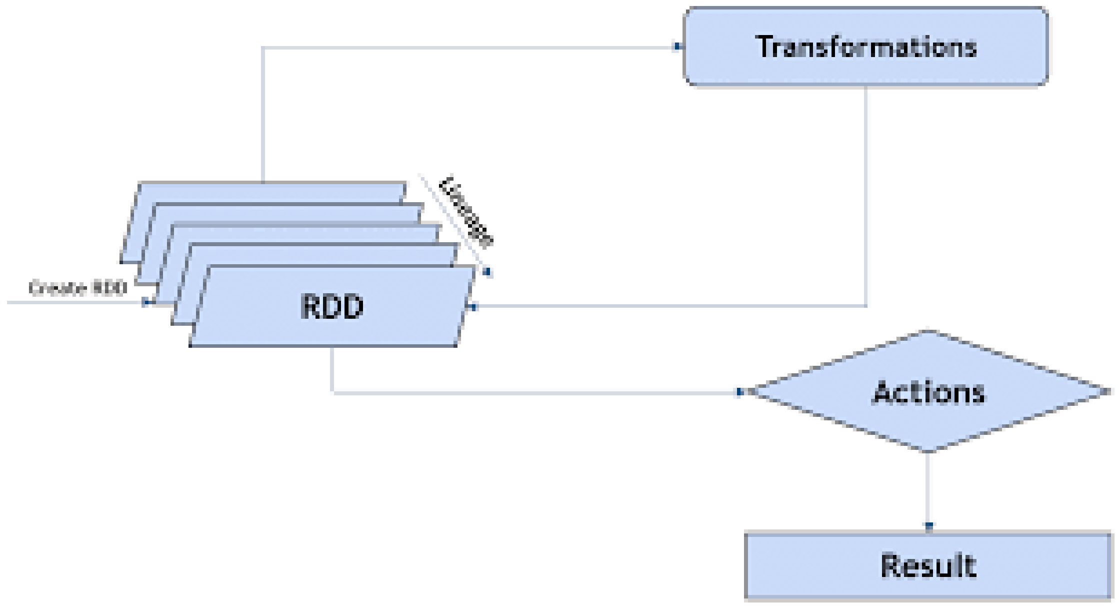


RDD Operations

RDDs support two types of operations:

- *transformations*, which create a new dataset from an existing one,
- and *actions*, which return a value to the driver program after running a computation on the dataset.

For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel reduceByKey that returns a distributed dataset).



RDD operations

- All transformations in Spark are *lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program.
- This design enables Spark to run more efficiently. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.
- By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also *persist* an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it.
- There is also support for persisting RDDs on disk, or replicated across multiple nodes.

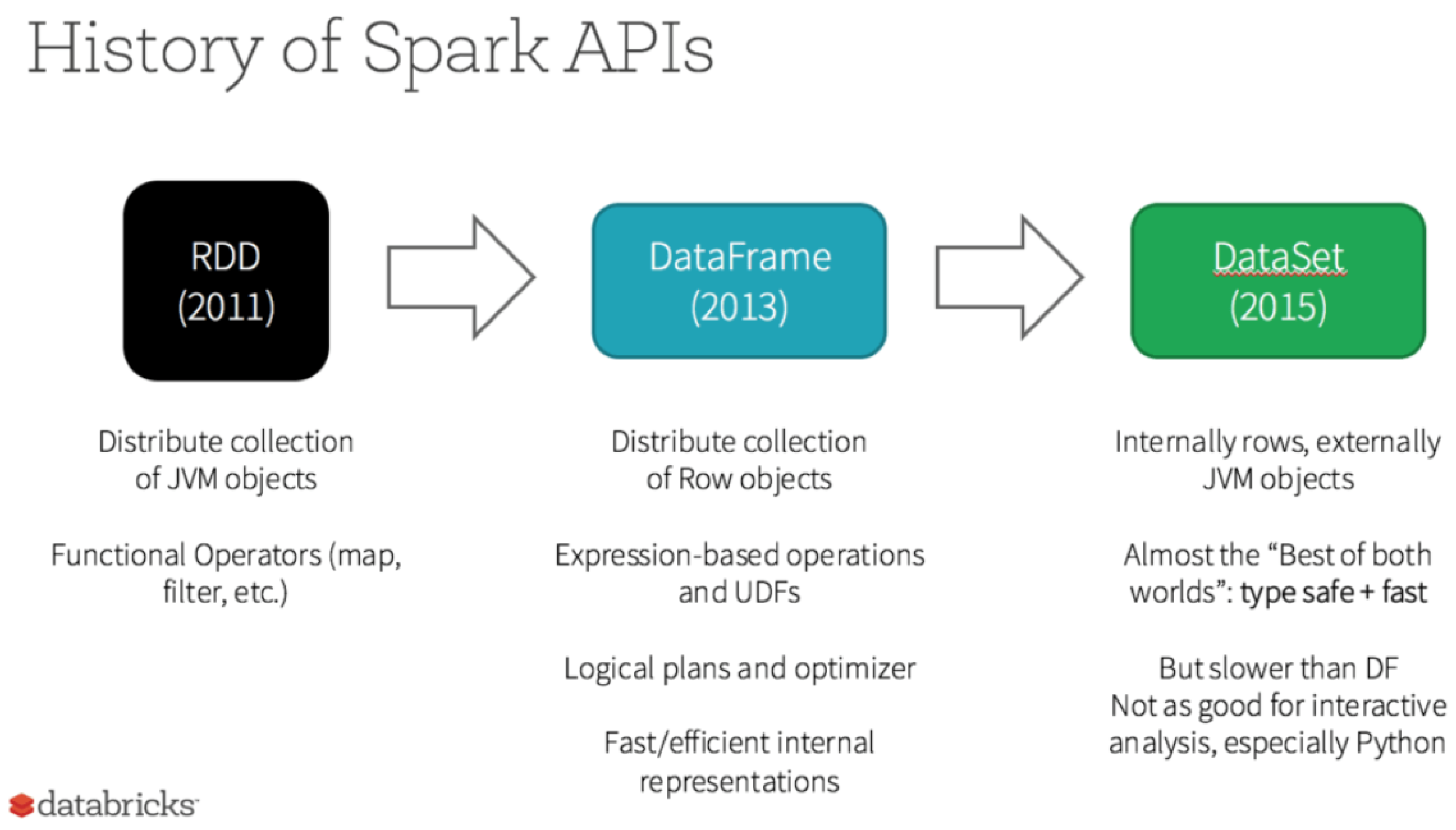
RDD Transformation (Ex)

Transfomation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

RDD Actions (Ex)

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior.

Spark Data Representation



History of Spark APIs

$$\begin{tikzcd}
\boxed{\text{RDD (2011)}} \arrow[r] & \boxed{\text{DataFrame (2013)}} \arrow[r] & \boxed{\text{DataSet (2015)}}
\end{tikzcd}$$

RDD (2011)	DataFrame (2013)	DataSet (2015)
Distribute collection of JVM objects	Distribute collection of Row objects	Internally rows, externally JVM objects
Functional Operators (map, filter, etc.)	Expression-based operations and UDFs	Almost the "Best of both worlds": type safe + fast
	Logical plans and optimizer	But slower than DF Not as good for interactive analysis, especially Python
	Fast/efficient internal representations	

databricks

Dataframes

- A DataFrame is an immutable, distributed collection of data organized into rows. Each one consists of a set of columns and each column has a name and an associated type.
- In other words, this distributed collection of data has a structure defined by a schema.
- If you are familiar with the table concept in a relational database management system (RDBMS), you realize that a DataFrame is essentially equivalent.
- A generic Row object represents each row in the DataFrame. Unlike RDD APIs, DataFrame APIs offer a set of domain specific operations that are relational and have rich semantics. Like the RDD APIs, the DataFrame APIs are classified into two types: transformation and action.
- The evaluation semantics are identical in RDD. Transformations are lazily evaluated, and actions are eagerly evaluated.
- A DataFrame can be created by reading data from many structured data sources
- and by reading data from tables in Hive or other databases.
- In addition, the Spark SQL module provides APIs to easily convert an RDD to a DataFrame by providing the schema information about the data in the RDD.
- The DataFrame API is available in Scala, Java, Python, and R.