Big Data Computing

Master's Degree in Computer Science

2022-2023

Gabriele Tolomei

Department of Computer Science

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

Recap from Last Lecture

- MapReduce → new distributed computing framework suitable for woriking with large scale datasets
- Useful in all those situations where data need to be accessed sequentially
- May be hard to program and does not support well multiple map-reduce rounds

Data-Flow Systems

- MapReduce uses 2 "ranks" of tasks: one for map the other for reduce

Data-Flow Systems

- MapReduce uses 2 "ranks" of tasks: one for map the other for reduce
- Data flows from the first rank to the second

Data-Flow Systems

- MapReduce uses 2 "ranks" of tasks: one for map the other for reduce
- Data flows from the first rank to the second
- Generalized Data-Flow Systems abstract from this in two ways:
  - Allow any number of "ranks"/tasks
  - Allow functions other than just map and reduce

Data-Flow Systems

- MapReduce uses 2 "ranks" of tasks: one for map the other for reduce
- Data flows from the first rank to the second
- Generalized Data-Flow Systems abstract from this in two ways:
  - Allow any number of "ranks"/tasks
  - Allow functions other than just map and reduce
- As long as data goes in one direction only, recovery at intermediate rank is possible

Spark: Most Popular Data-Flow System

- Expressive computing framework not limited to map-reduce model

Spark: Most Popular Data-Flow System

- Expressive computing framework not limited to map-reduce model
- In addition to MapReduce, Spark provides:
  - Fast data sharing (no intermediate saving to local disks + caching)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce

Spark: Most Popular Data-Flow System

- Expressive computing framework not limited to map-reduce model
- In addition to MapReduce, Spark provides:
  - Fast data sharing (no intermediate saving to local disks + caching)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce
- Compatible with Hadoop

Spark: Introduction

- Originally developed at UC Berkeley in 2009 and later donated to the Apache Software Foundation (open-source)

Spark: Introduction

- Originally developed at UC Berkeley in 2009 and later donated to the Apache Software Foundation (open-source)
- Implemented in Scala (running on top of the Java Virtual Machine)

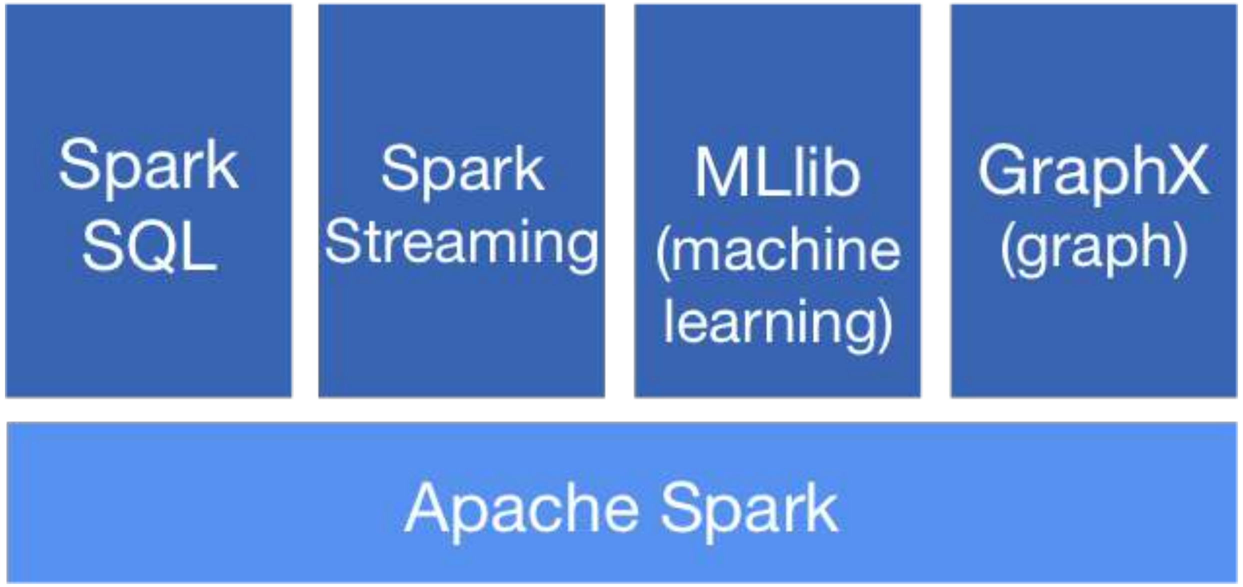
Spark: Introduction

- Originally developed at UC Berkeley in 2009 and later donated to the Apache Software Foundation (open-source)
- Implemented in Scala (running on top of the Java Virtual Machine)
- Unified computing engine (Spark Core)

Spark: Introduction

- Originally developed at UC Berkeley in 2009 and later donated to the Apache Software Foundation (open-source)
- Implemented in Scala (running on top of the Java Virtual Machine)
- Unified computing engine (Spark Core)
- Set of high-level APIs for data analysis:
  - Spark SQL (structured data), MLlib (machine learning), GraphX (graph analytics), Spark Streaming (stream data processing)

Spark: Overview



Spark: Introduction

- Unlike Hadoop, Spark does not come with a storage system

Spark: Introduction

- Unlike Hadoop, Spark does not come with a storage system
- In fact, it provides interfaces for many local and distributed storage systems:
  - HDFS, Amazon S3, Cassandra, Hive Metastore, or classical RDBMS

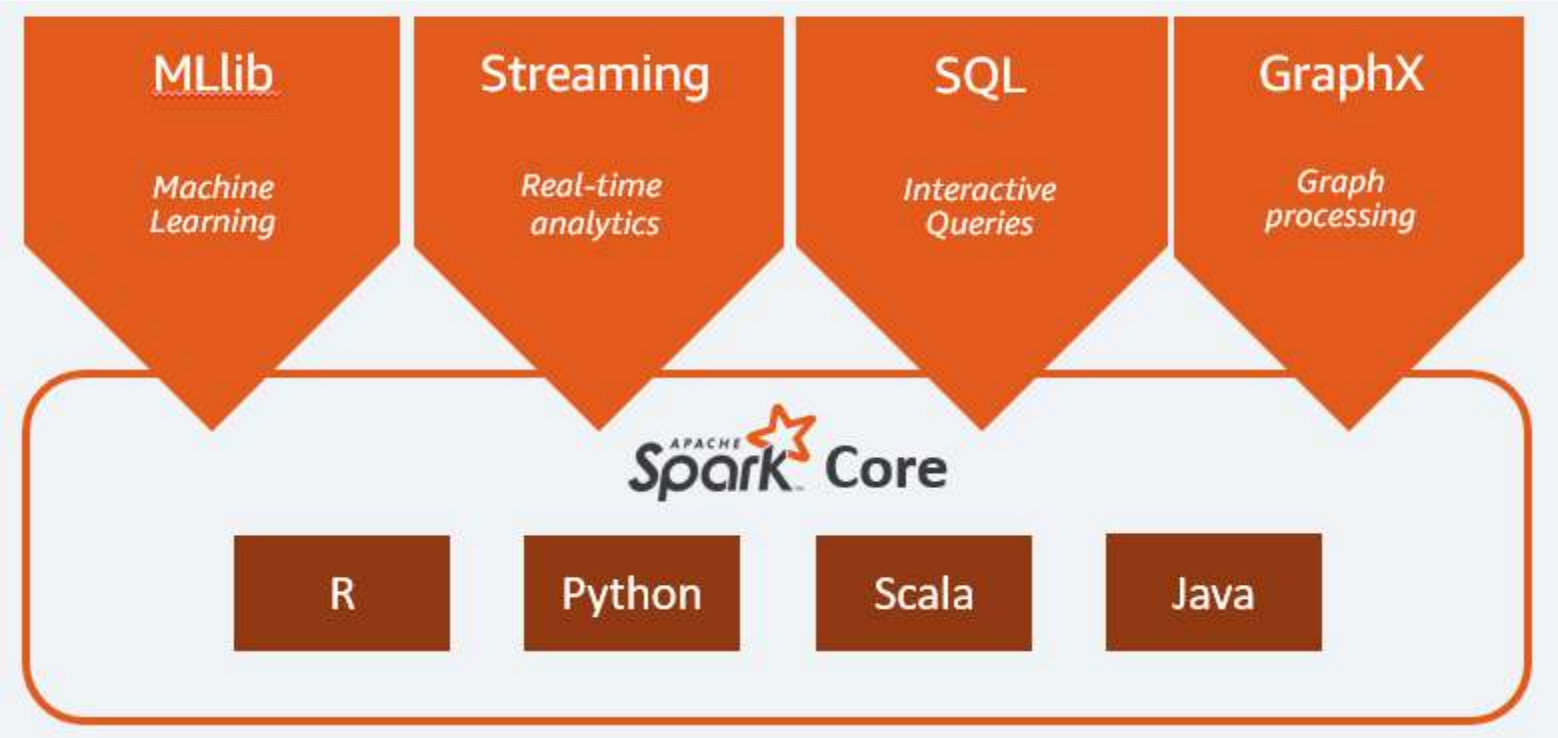
Spark: Introduction

- Unlike Hadoop, Spark does not come with a storage system
- In fact, it provides interfaces for many local and distributed storage systems:
  - HDFS, Amazon S3, Cassandra, Hive Metastore, or classical RDBMS
- Additionally, Spark's APIs are available for many programming languages: Scala, Java, Python, and R

Spark: Introduction

- Unlike Hadoop, Spark does not come with a storage system
- In fact, it provides interfaces for many local and distributed storage systems:
  - HDFS, Amazon S3, Cassandra, Hive Metastore, or classical RDBMS
- Additionally, Spark's APIs are available for many programming languages: Scala, Java, Python, and R
- This flexibility is the key of its success in the Big Data landscape

Spark: More Detailed Overview



Spark: Features

- Fault-tolerant system

Spark: Features

- Fault-tolerant system
- In-memory caching which enables efficient execution of multi-round algorithms (i.e., multiple sequential tasks)
  - performance improvement w.r.t. Hadoop

Spark: Features

- Fault-tolerant system
- In-memory caching which enables efficient execution of multi-round algorithms (i.e., multiple sequential tasks)
  - performance improvement w.r.t. Hadoop
- Spark can run:
  - on a single machine → local mode
  - on a cluster managed by a cluster manager (e.g., Spark Standalone, YARN, Mesos)

Spark: Features

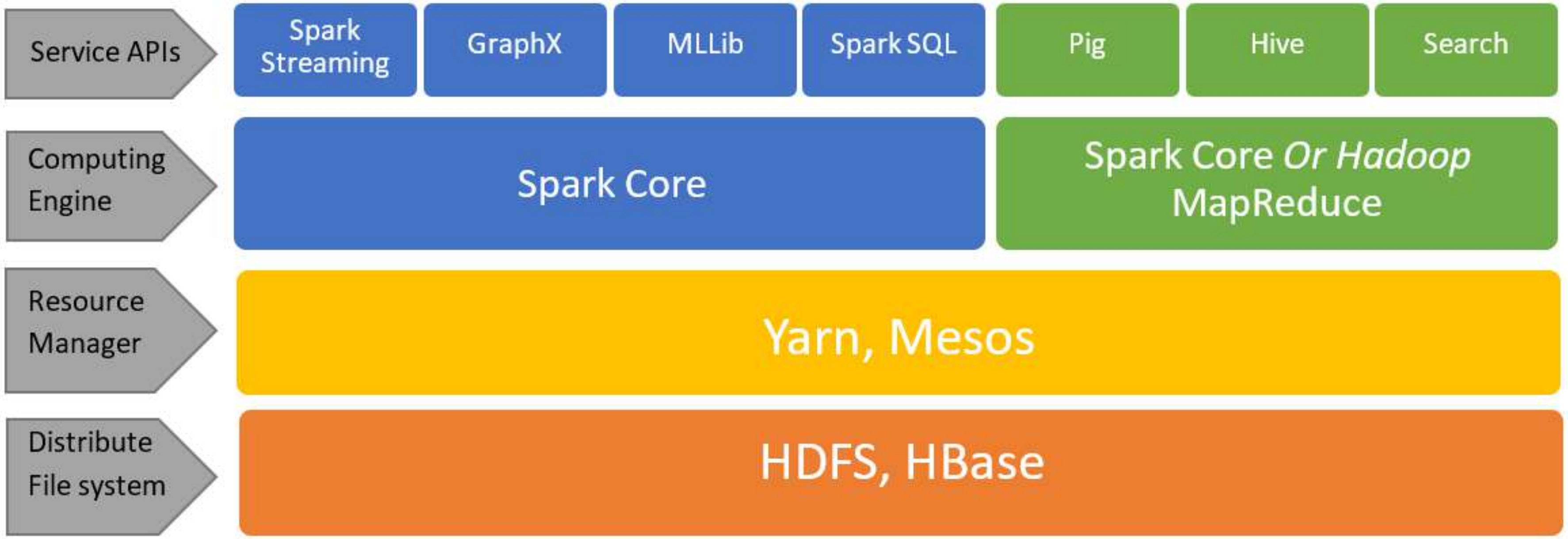


Figure 1 - Spark Context

Spark Application: Driver

- The driver process (a.k.a. master in MapReduce terminology) runs the application's entry point from a node in the cluster

Spark Application: Driver

- The driver process (a.k.a. master in MapReduce terminology) runs the application's entry point from a node in the cluster
- The driver is responsible for:
  - Maintaining information about the application
  - Responding to a user program or input
  - Analyzing, distributing, and scheduling work across executors

Spark Application: Driver

- The driver process (a.k.a. master in MapReduce terminology) runs the application's entry point from a node in the cluster
- The driver is responsible for:
  - Maintaining information about the application
  - Responding to a user program or input
  - Analyzing, distributing, and scheduling work across executors
- The driver is represented by an object called Spark Context

Spark Application: Executor(s) and Cluster Manager

- Executor processes (a.k.a. workers in Hadoop terminology) actually compute the tasks assigned by the driver

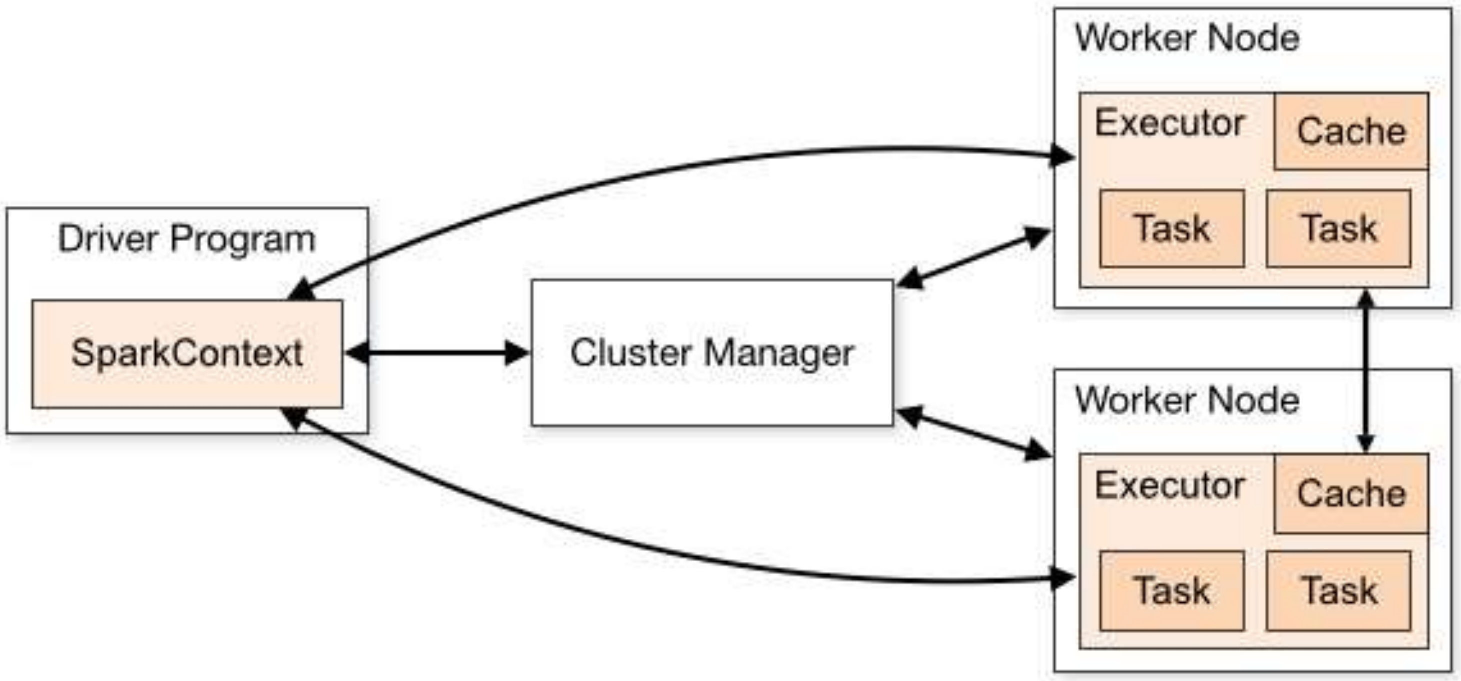
Spark Application: Executor(s) and Cluster Manager

- Executor processes (a.k.a. workers in Hadoop terminology) actually compute the tasks assigned by the driver
- Each executor is responsible for:
  - Running the code assigned to it by the driver
  - Reporting the state of the computation back to the driver

Spark Application: Executor(s) and Cluster Manager

- **Executor processes** (a.k.a. **workers** in Hadoop terminology) actually compute the tasks assigned by the driver
- Each executor is responsible for:
    - Running the code assigned to it by the driver
    - Reporting the state of the computation back to the driver
- The cluster manager controls physical machines and allocates resources to applications

Spark Application



Spark Application: Considerations

- Driver and executors are processes which can live on the same machines or on different nodes

Spark Application: Considerations

- Driver and executors are processes which can live on the same machines or on different nodes
- When Spark is running in local mode, both the driver and executors are running as separate threads on the same machine

Spark Application: Considerations

- Driver and executors are processes which can live on the same machines or on different nodes
- When Spark is running in local mode, both the driver and executors are running as separate threads on the same machine
- Executors mostly run Scala code

Spark Application: Considerations

- Driver and executors are processes which can live on the same machines or on different nodes
- When Spark is running in local mode, both the driver and executors are running as separate threads on the same machine
- Executors mostly run Scala code
- Driver can be governed by different languages using Spark's APIs

Resilient Distributed Dataset (RDD)

- Fundamental abstraction of Spark to indicate a collection of elements of the same type
  - Generalization of MapReduce's key-value pairs

Resilient Distributed Dataset (RDD)

- Fundamental **abstraction** of Spark to indicate a collection of elements of the same type
  - Generalization of MapReduce's key-value pairs
- RDDs are **partitioned** and possibly spread across multiple nodes of the cluster

Resilient Distributed Dataset (RDD)

- Fundamental abstraction of Spark to indicate a collection of elements of the same type
  - Generalization of MapReduce's key-value pairs
- RDDs are partitioned and possibly spread across multiple nodes of the cluster
- Best suited for applications that apply the same operation across all the elements of the dataset

RDD: Partitions

- Each RDD is split into chunks called partitions distributed across nodes

RDD: Partitions

- Each RDD is split into chunks called partitions distributed across nodes
- A program can specify the number of partitions for an RDD (otherwise Spark will choose one)

RDD: Partitions

- Each RDD is split into chunks called partitions distributed across nodes
- A program can specify the number of partitions for an RDD (otherwise Spark will choose one)
- Programmer can also decide whether to use the default Hash Partitioner or a custom one

RDD: Partitions

- Each RDD is split into chunks called partitions distributed across nodes
- A program can specify the number of partitions for an RDD (otherwise Spark will choose one)
- Programmer can also decide whether to use the default Hash Partitioner or a custom one
- A typical number of partitions is 2 or 3 times the number of cores

RDD: Partitions

- Partitioning enables the following:
  - Data reuse → data is kept in executors' main memory so as to avoid expensive access to external disks

RDD: Partitions

- Partitioning enables the following:
  - Data reuse → data is kept in executors' main memory so as to avoid expensive access to external disks
  - Parallelism → Some data transformations are applied independently to each partition thereby avoiding expensive data transfers

RDD: Characteristics

- RDDs are immutable (i.e., read-only)

RDD: Characteristics

- RDDs are immutable (i.e., read-only)
- Can be created either from data stored on distributed file system (e.g., HDFS) or as a result of transformations of other RDDs

RDD: Characteristics

- RDDs are immutable (i.e., read-only)
- Can be created either from data stored on distributed file system (e.g., HDFS) or as a result of transformations of other RDDs
- RDDs do not need to be always materialized
  - Each RDD maintains a sort of "trace" of transformations (lineage) that led to the current status
  - This way, RDD can always be re-created even upon a failure

RDD Operations

- Let A be an RDD, the following 3 operations are possible:

RDD Operations

- Let A be an RDD, the following 3 operations are possible:
  - Transformations → generate a new RDD B from the data in A

RDD Operations

- Let A be an RDD, the following 3 operations are possible:
  - Transformations → generate a new RDD B from the data in A
  - Actions → launch a computation on the data in A, which returns a value to the application

RDD Operations

- Let A be an RDD, the following 3 operations are possible:
  - Transformations → generate a new RDD B from the data in A
  - Actions → launch a computation on the data in A, which returns a value to the application
  - Persistence → save the RDD in memory for later actions

RDD Operations: Transformations

- Narrow: each partition of A contributes at most to one partition of B (e.g., **`map`**)
  - No need to shuffle data across nodes

RDD Operations: Transformations

- Narrow: each partition of A contributes at most to one partition of B (e.g., **`map`**)
  - No need to shuffle data across nodes
- Wide: each partition of A may contribute to multiple partitions of B (e.g., **`groupBy`**)
  - Possible need to transfer data across nodes

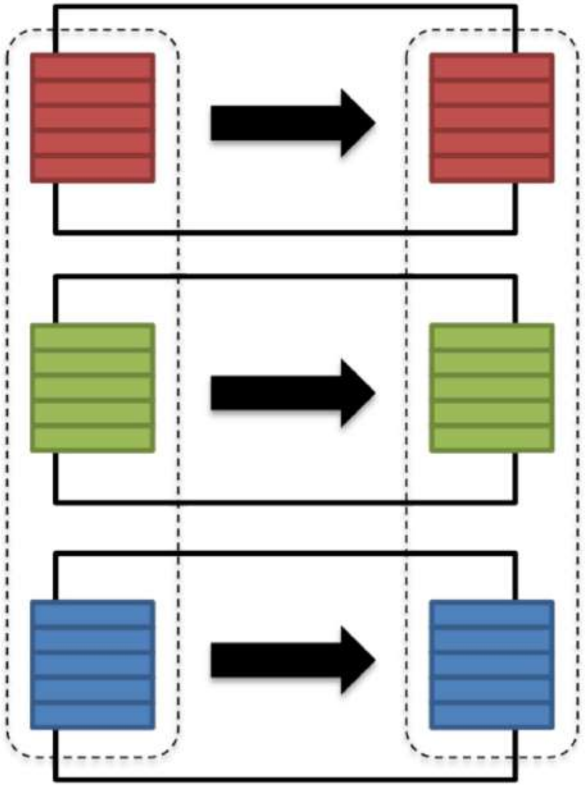
RDD Operations: Transformations

- Narrow: each partition of A contributes at most to one partition of B (e.g., **`map`**)
  - No need to shuffle data across nodes
- Wide: each partition of A may contribute to multiple partitions of B (e.g., **`groupBy`**)
  - Possible need to transfer data across nodes
- Lazy evaluation: nothing is computed unless required by an action

Narrow vs. Wide Transformations

Narrow

Input and output stay on the same partition

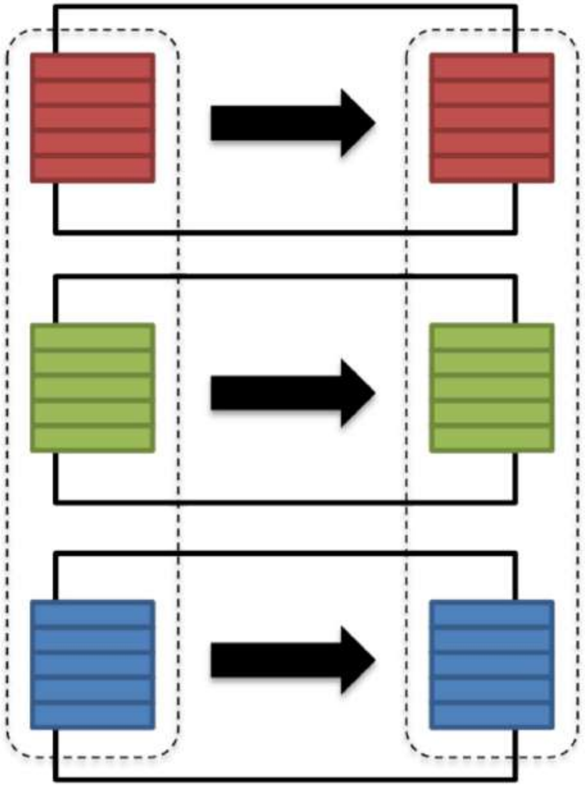


Narrow vs. Wide Transformations

Narrow

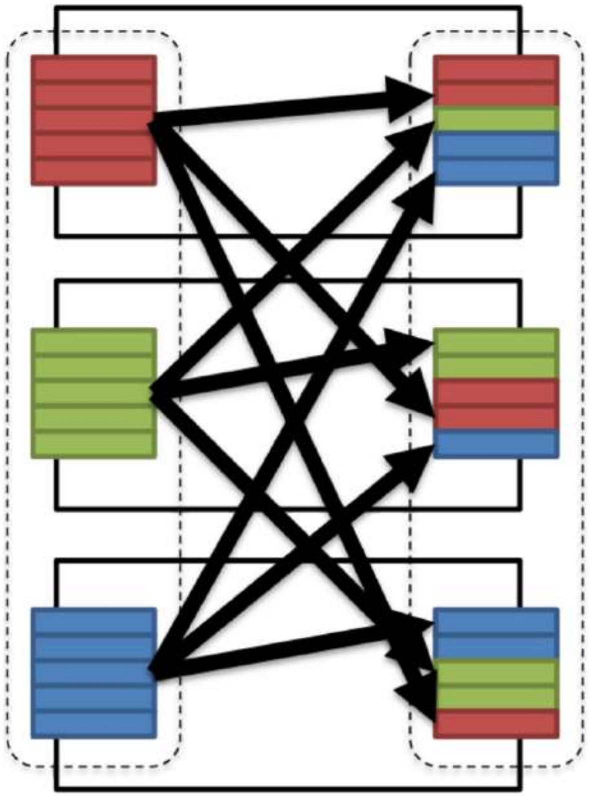
Input and output stay on the same partition

No data transfers



Narrow vs. Wide Transformations

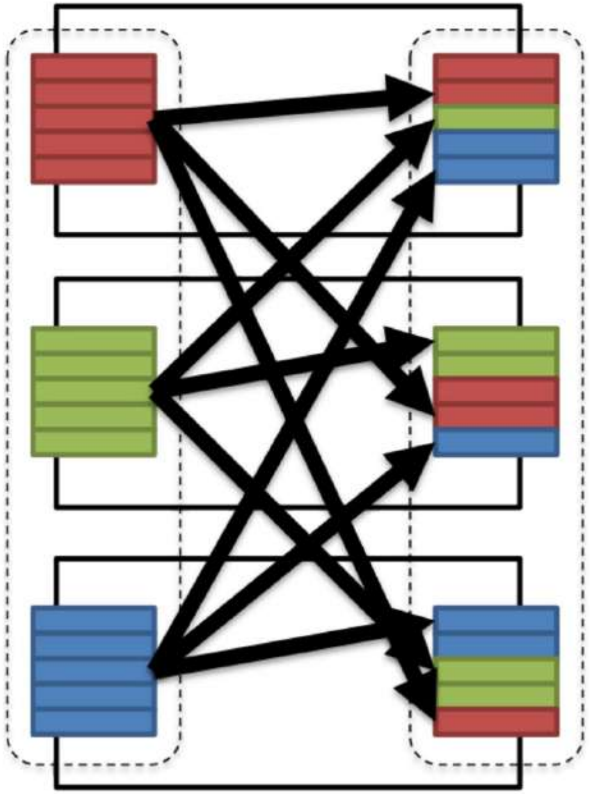
Wide



Input from other partitions may be needed

Narrow vs. Wide Transformations

Wide



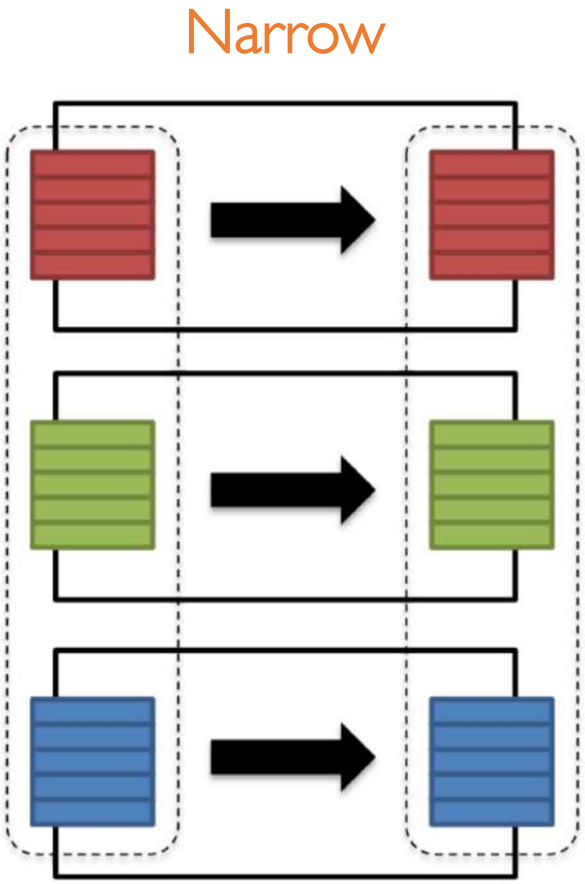
Input from other partitions may be needed

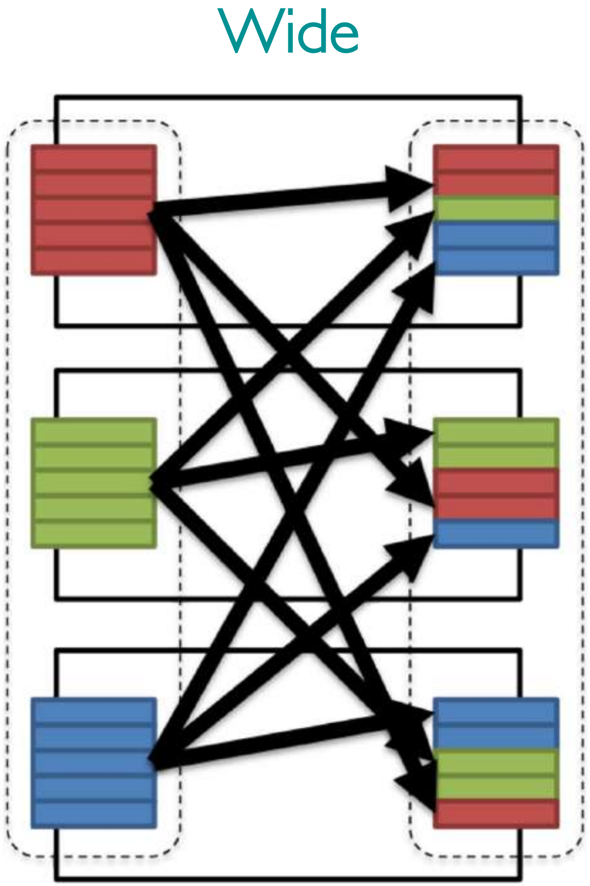
Data shuffling across nodes

Narrow vs. Wide Transformations

Input and output stay on the same partition

No data transfers





Input from other partitions may be needed

Data shuffling across nodes

RDD Operations: Actions

- Example: the count method returns the number of elements of the RDD

RDD Operations: Actions

- Example: the count method returns the number of elements of the RDD
- When the action is called the RDD is actually materialized (lazy evaluation)

Spark DataFrame and Dataset APIs

- RDDs are the most basic data model used by Spark
  - low-level and schema-less

Spark DataFrame and Dataset APIs

- RDDs are the most basic data model used by Spark
  - low-level and schema-less
- On top of RDD API, Spark SQL module provides 2 interfaces to operate on structured data like tables in relational databases:
  - DataFrame API
  - Dataset API

Spark: DataFrame

- Distributed collection of data organized into **named columns**

Spark: DataFrame

- Distributed collection of data organized into **named columns**
- Allows higher level abstraction than plain vanilla RDDs

Spark: DataFrame

- Distributed collection of data organized into **named columns**
- Allows higher level abstraction than plain vanilla RDDs
- Since Spark 2.0 it is part of a more general Dataset API
  - Dataset API is available only for Scala and Java as it extends DataFrame API with type-safe, object-oriented programming interface

Spark: DataFrame

- Distributed collection of data organized into **named columns**
- Allows higher level abstraction than plain vanilla RDDs
- Since Spark 2.0 it is part of a more general Dataset API
  - Dataset API is available only for Scala and Java as it extends DataFrame API with type-safe, object-oriented programming interface
- Similar to Pandas DataFrame unless few differences

Spark DataFrame vs. Pandas DataFrame

- Spark DataFrames are immutable: once created they cannot be modified

Spark DataFrame vs. Pandas DataFrame

- Spark DataFrames are immutable: once created they cannot be modified
- As for RDDs, Spark may apply 2 kinds of operations on DataFrames: transformations and actions

Spark DataFrame vs. Pandas DataFrame

- Spark DataFrames are immutable: once created they cannot be modified
- As for RDDs, Spark may apply 2 kinds of operations on DataFrames: transformations and actions
- Lazy evaluation allows to queue transformations applied to elements of a DataFrame until an action is called

Spark DataFrame vs. Pandas DataFrame

- Spark DataFrames are immutable: once created they cannot be modified
- As for RDDs, Spark may apply 2 kinds of operations on DataFrames: transformations and actions
- Lazy evaluation allows to queue transformations applied to elements of a DataFrame until an action is called
- DataFrame (and Dataset as well) can be turned back to RDD

Spark vs. Hadoop MapReduce

- **Performance:** Spark is usually faster
  - In-memory data processing vs. data persistencing to disk after any map/reduce step
  - Spark requires lots of memory to run fast, otherwise its performance deteriorates
  - MapReduce integrates better with other services running

Spark vs. Hadoop MapReduce

- Performance: Spark is usually faster
  - In-memory data processing vs. data persistencing to disk after any map/reduce step
  - Spark requires lots of memory to run fast, otherwise its performance deteriorates
  - MapReduce integrates better with other services running
- Ease of use: Spark provides a higher-level API which is easier to program

Spark vs. Hadoop MapReduce

- **Performance:** Spark is usually faster
  - In-memory data processing vs. data persistencing to disk after any map/reduce step
  - Spark requires lots of memory to run fast, otherwise its performance deteriorates
  - MapReduce integrates better with other services running
- **Ease of use:** Spark provides a higher-level API which is easier to program
- **Data processing:** Spark is more flexible and general

Useful Extra References

- Spark official website

Useful Extra References

- Spark official website
- What is Spark? [Databricks]

Useful Extra References

- Spark official website
- What is Spark? [Databricks]
- Getting Started with Spark [Databricks]

Useful Extra References

- Spark official website
- What is Spark? [Databricks]
- Getting Started with Spark [Databricks]
- Spark tutorial [Data-Flair]

Useful Extra References

- Spark official website
- What is Spark? [Databricks]
- Getting Started with Spark [Databricks]
- Spark tutorial [Data-Flair]
- PySpark Tutorial (video) [freeCodeCamp.org]
- …

Take-Home Message of Today

- Spark is a general-purpose distributed data processing engine which overcomes many of the Hadoop's limitations

Take-Home Message of Today

- Spark is a general-purpose distributed data processing engine which overcomes many of the Hadoop's limitations
- Spark provides a rich ecosystem of services to work on (big) data through APIs accessible via multiple programming languages

Take-Home Message of Today

- Spark is a general-purpose distributed data processing engine which overcomes many of the Hadoop's limitations
- Spark provides a rich ecosystem of services to work on (big) data through APIs accessible via multiple programming languages
- Lazy execution avoids frequent and costly disk operations

Take-Home Message of Today

- Spark is a general-purpose distributed data processing engine which overcomes many of the Hadoop's limitations
- Spark provides a rich ecosystem of services to work on (big) data through APIs accessible via multiple programming languages
- Lazy execution avoids frequent and costly disk operations
- Spark's DataFrame as the main abstraction for playing with data