Big Data
Lecture 1 – Introduction to Apache Spark

Gianluca Quercini

gianluca.quercini@centralesupelec.fr

Centrale DigitalLab, 2023

What you will learn

In this lecture you will learn:

- **Big data** notions, motivations and challenges.
- Introduction to Apache Spark.
- Spark RDD programming.

Structured data

Definition (Structured data)

Structured data describe the *properties* (e.g., the name, address, credit card number and phone number) of *entities* (e.g., customer, products) following a fixed *template* or *model*.

- Records stored in the tables of a **relational database**.
- Each property is easily distinguishable from the others.
  - It fits one unit of the structure (e.g., a column of a table).

Unstructured data

Definition (Unstructured data)

Unstructured data describe entities that lack a clear structure due to their properties not being immediately distinguishable.

- **Text** is unstructured.
  - Description of entity properties drowned in a rich context.
  - No direct access to these properties.

Semi-structured data

Definition (Semi-structured data)

Semi-structured data have a structure in which the entities and their properties are easily distinguishable, but the organization of the structure is not as rigorous as in a table of a relational database.

- Examples: XML, JSON, HTML documents, spreadsheets...

Example (XML document)

```
<book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
</book>
```

What is Big Data?

Definition (Big Data)

The term **Big Data** refers to an accumulation of data that is too large and complex for processing by traditional database management tools. ▸ Source

The term Big Data also refers to:

- The (hardware and software) **solutions** developed to manage large volumes of data.
- The **branch of computing** studying solutions to manage large volumes of data.

Other sources define Big Data in terms of its characteristics.

The 3 Vs of Big Data

Definition (3V)

Big data is high-**Volume**, high-**Velocity** and high-**Variety** information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making (Gartner).

- **Volume**, the size of a dataset.
- **Velocity**, the necessity of processing data as they arrive.
- **Variety**, the heterogeneous nature of data (structured, unstructured, semi-structured).

Scalability

- **Scalability**: the ability of a system to handle growing amounts of data, without a significant decrease of its performances.
- Two techniques:
  - **Vertical scaling** (**scale-up**).
    - Upgrade the existing infrastructure (more memory, computing power. . .).
  - **Horizontal scaling** (**scale-out**).
    - Add machines to the existing infrastructure.
    - Distribute the data and the workload across several machines.
- **Advantages** of vertical scaling:
  - **Easier** to maintain a single machine than many.
  - **Centralized control** over the data and the computations.
- **Advantages** of horizontal scaling:
  - **Limitless upgrade** of the computing power of a system.
  - **Fault tolerance**.

Big Data challenges

- In this course, we study two main challenges: **processing** and **storage**.

Processing.

- Parallelize the computation across machines.
- Parallel Databases.
- Distributed processing frameworks (e.g., Hadoop MapReduce/Spark)

Storage.

- Distributed file systems.
- Distributed (relational/NoSQL) databases.

Processing big data

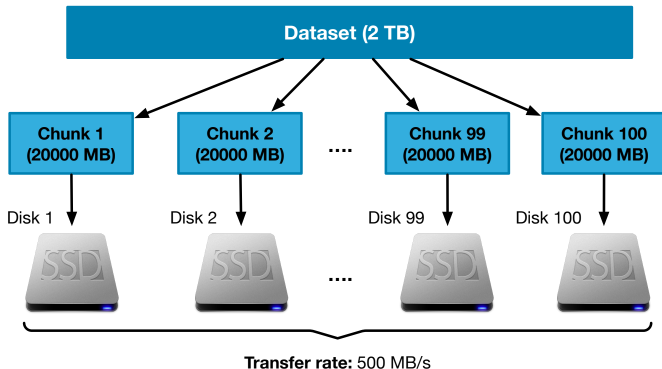
Why processing Big Data is challenging?

- **Disk storage** capacities have increased rapidly over the years.
  - A typical disk from 1990 could store **1,370 MB of data** (cf. Seagate ST-41600n).
  - A typical disk (SSD) today can store **2 TB of data** (cf. Seagate Barracuda 120).
- **Disk access** increased much slower.
  - A typical disk from 1990 had a **transfer speed** of 4.4 MB/s.
  - A typical disk (SSD) from today has a **transfer speed** of 500 MB/s.

- In 1990 it could take 5 minutes to read all the data from a disk.
- In 2020 it takes more than one hour to read all the data from a disk.

Processing big data: parallelization

- Split the data into many smaller chunks stored on separate disks.
- Read **in parallel** from each disk.



☞ Each disk may contain chunks from different datasets.

Parallelization: challenges

- **Hardware failure.** Lots of disks → higher chances of failures.
    - Use **redundancy**. Replicate data across several disks.

☞ This is where **HDFS (Hadoop Distributed File System)** comes into play.

- **Combine data** from different sources.

☞ This is where **Apache Spark** comes into play.

Apache Spark

Definition (Apache Spark)

Apache Spark is a *distributed computing framework* designed to be *fast* and *general-purpose*. Source

Main features

- **Speed**. Run computations in **memory**, as opposed to Hadoop that heavily relies on disks and HDFS.
- **General-purpose**. It integrates a wide range of workloads that previously required separate distributed systems.
  - Batch applications, iterative algorithms.
  - Interactive queries, streaming applications.
- **Accessibility.** It offers APIs in Python, Scala, Java and SQL and rich built-in libraries.
- **Integration.** It integrates with other Big Data tools, such as Hadoop.

Spark stack

Spark core

- A **computational engine** responsible for **scheduling**, **distributing**, and **monitoring** applications.
  - Spark applications consist of computational **tasks** distributed in a cluster.
- Provides the API that defines **Resilient Distributed Datasets** (RDDs), Spark's main programming abstraction.

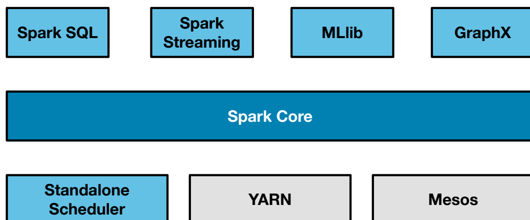


Image source

Spark stack

Spark SQL

- Spark's package for working with **(semi-)structured data**.
- Supports SQL, the Hive Query Language and many data sources:
  - Hive tables, Parquet, JSON
- Allows developers to use a combination of SQL queries and programmatic data manipulations in Python, Java or Scala.

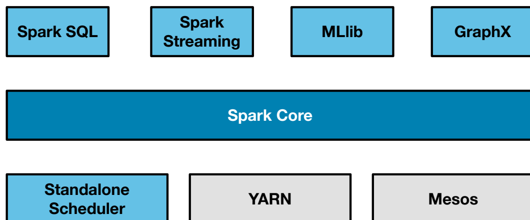


Image source

Spark stack

Spark streaming

- Spark's package for processing live **streams** of data.
- **Example.** Logfiles generated by Web servers, status updates in social network platforms . . .

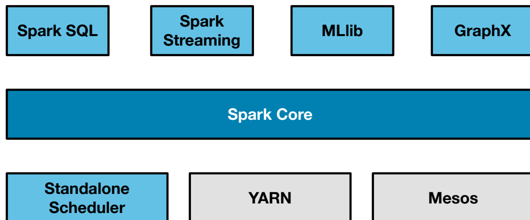


Image source

Spark stack

MLlib

- Spark's package providing numerous **machine learning** algorithms.
  - Classification, regression, clustering, model evaluation and data import.
- The ML algorithms **scale out** across a cluster.

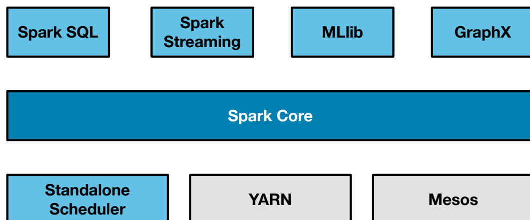


Image source

Spark stack

GraphX

- Spark's library for manipulating **graphs**.
- Extends the Spark RDD API to allow the representation of **directed graphs**.
- Provides the implementation of common graph algorithms (e.g., PageRank).

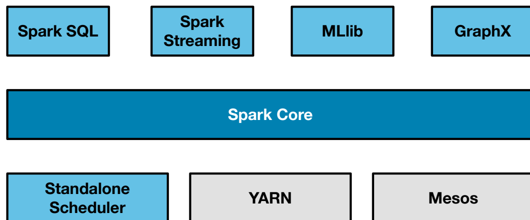
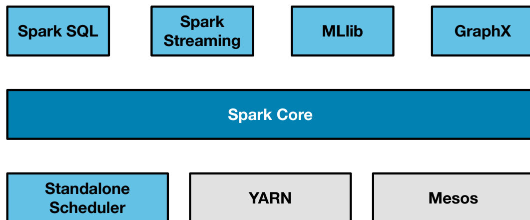


Image source

Spark stack

Cluster managers

- A **cluster manager** is a component that controls how tasks are distributed in a cluster.
- Spark provides its own **standalone** cluster manager.
- Spark can be used on other cluster managers, such as Yarn and Mesos.



▸ Image source

Spark stack: benefits

- **Improvements** on the bottom layers are automatically reflected on high-level libraries.
  - Optimizations in the Spark core result in better performances in Spark SQL and MLlib.
- **Remove the costs** of using **different independent systems**.
  - Deployment, maintenance, test, support of different systems (streaming, SQL, machine learning. . .).
- **Different programming models** in the same application.
  - Application that reads a stream of data.
  - Applies machine learning algorithms.
  - Uses SQL to analyze the results.

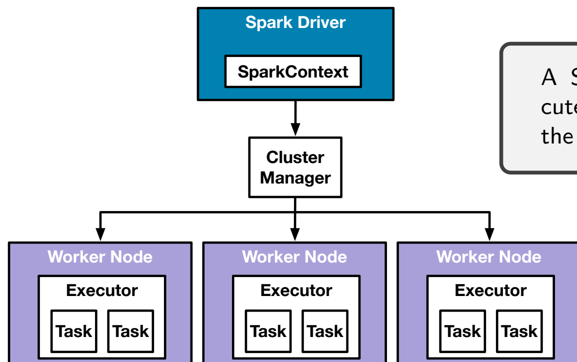
Who uses Spark

- **Amazon**.
- **eBay**. Log transaction aggregation and analytics.
- **Groupon**.
- **Stanford DAWN**. Research project aiming at democratizing AI.
- **TripAdvisor**.
- **Yahoo!**

Full list available at http://spark.apache.org/powered-by.html

Spark architecture

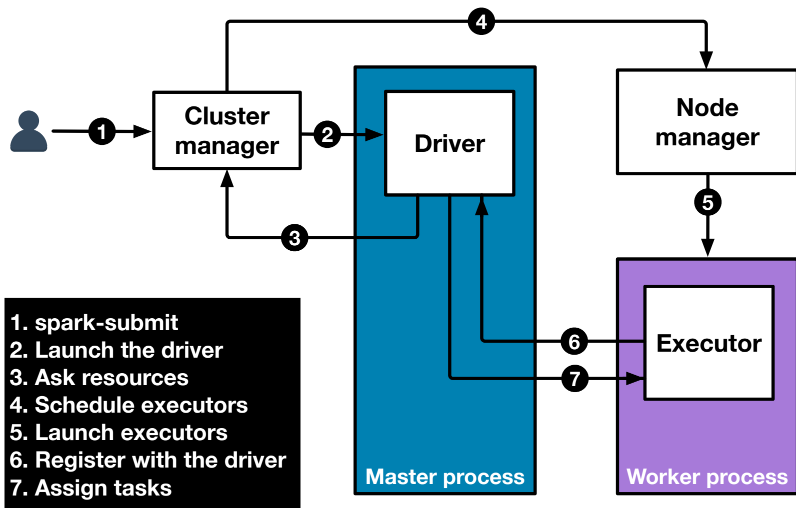
- **Master/slave** architecture: one coordinator (the **Driver**) and many distributed workers, called **executors**.
- The driver and the executors are separate **Java processes**.
- **Spark application.** Driver + executors.



A Spark application is executed in the cluster through the **cluster manager**.

Image source

Launching a Spark application



Writing a Spark program

- The program accesses Spark through an object called **SparkContext**.
- **SparkContext** represents a connection to a cluster.

Initializing the SparkContext

```
from pyspark import SparkCon  f, SparkContext

conf = SparkConf().setMaster(<cluster URL>).setAppName(<app_name>)
sc = SparkContext(conf = conf)
```

- A Spark program is a sequence of operations invoked on the **SparkContext**.
- These operations manipulate a special type of data structure, called **Resilient Distributed Dataset (RDD)**.

Resilient Distributed Dataset (RDD)

Definition (Resilient Distributed Dataset)

A **Resilient Distributed Dataset**, or simply **RDD**, is an **immutable**, **distributed** collection of objects. ▸ Source

- The data in each RDD is split across multiple **partitions**.
- Each partition resides on one node of the cluster.
- Two partitions can reside on the same node.

Resilient Distributed Dataset (RDD)

Input file

Call me Ishmael. Some years ago--never mind how long precisely--having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off--then, I account it high time to get to sea as soon as I can. This is my substitute for pistol and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me.

File in HDFS
By default
1 block = 1 partition

It is possible to specify a different number of partitions

RDD

Partition 0

Call me Ishmael. Some years ago--never mind how long precisely--having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and

Partition 1

regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially

Partition 2

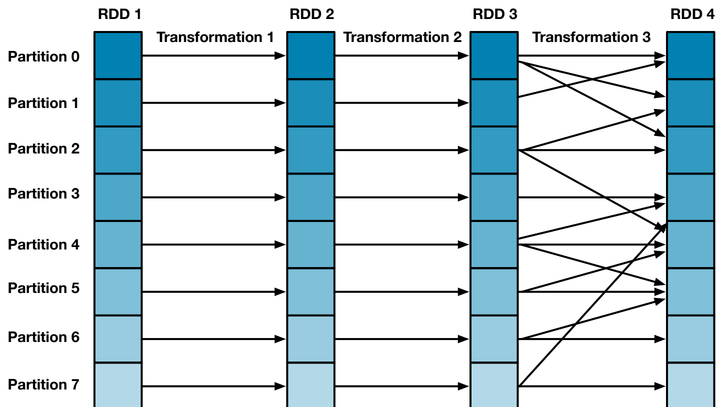
whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off--then, I account it high time to get to sea as soon as I can. This is my substitute for pistol

Partition 3

and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me.

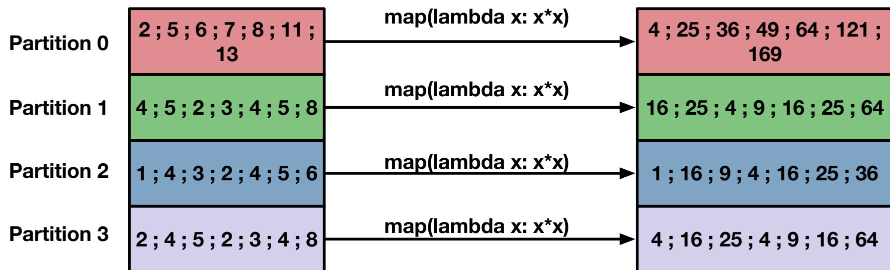
RDD transformations

A **transformation** is an operation that takes in one or more RDDs and returns a **new RDD**. A transformation is applied **in parallel** on each partition.



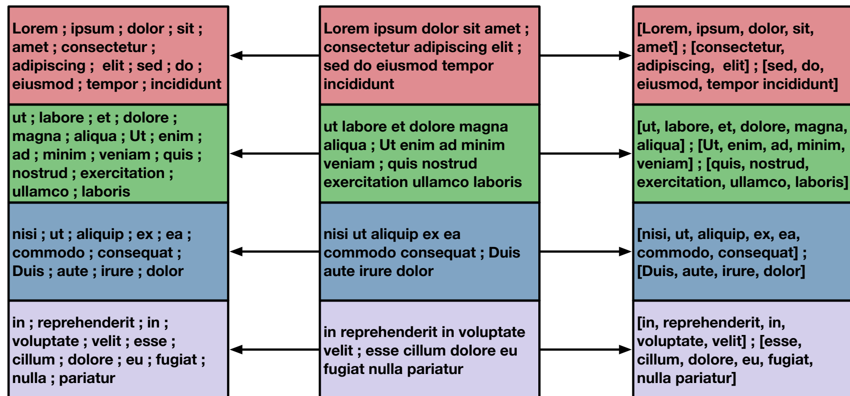
RDD transformations: `map`

`map()` takes in a **function** f and a RDD $< x_i \mid 0 \leq i \leq n >$; returns a **new RDD** $< f(x_i) \mid 0 \leq i \leq n >$.



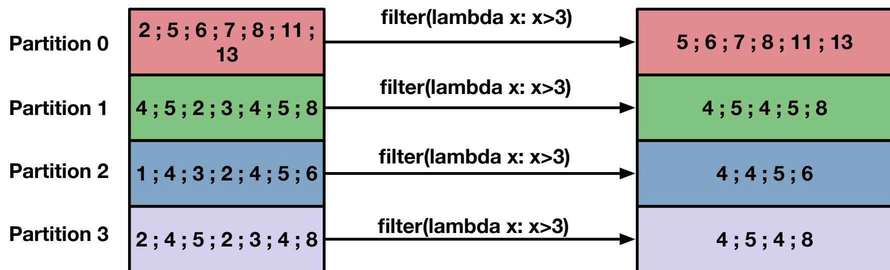
RDD transformations: `flatMap`

`flatMap` is used instead of `map` when the function f returns a list and we need the results to be flattened.



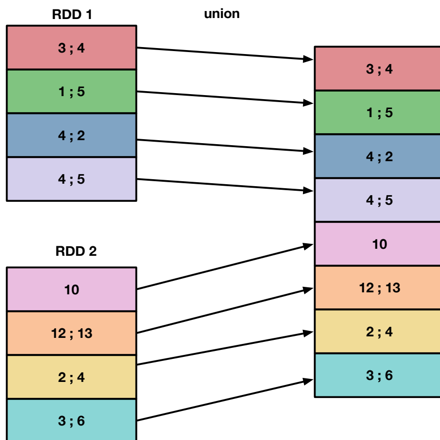
RDD transformations: `filter`

`filter()` takes in a **predicate** p and a RDD $< x_i \mid 0 \leq i \leq n >$; returns a **new RDD** $< x_i \mid 0 \leq i \leq n,\ p(x_i)\ is\ true >$



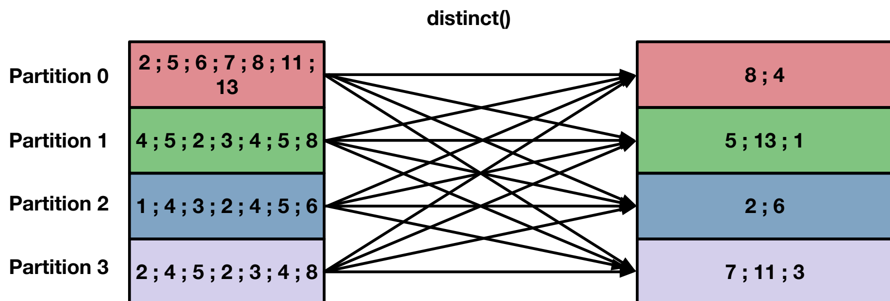
RDD transformations: `union`

`union()` takes in two RDDs and returns a **new RDD** containing the items of the first and second RDD **with repetitions**.



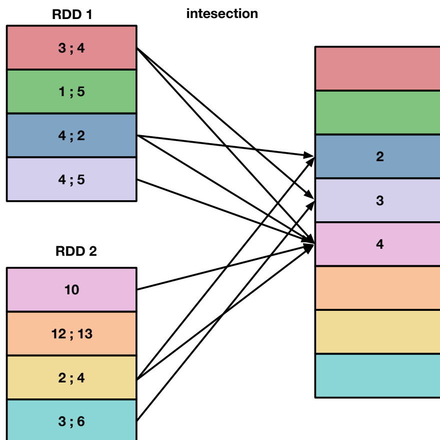
RDD transformations: `distinct`

`distinct()` takes in one RDD and returns a **new RDD** containing the items of the input RDD **without repetitions**.



RDD transformations: `intersection`

`intersection()` takes in one two RDDs and returns a **new RDD** containing the items that occur in both RDDs.



RDD transformations: narrow and wide

Definition (Narrow transformation)

A **narrow transformation** is one where each partition of the output RDD depends on at most on partition of the input RDD.

`filter`, `map` and `flatMap` are narrow transformations.

Definition (Wide transformation)

A **wide transformation** is one where each partition of the output RDD may depend on several partitions of the input RDD

`distinct` and `intersection` are wide transformations.

☞ Wide transformations are more costly than narrow transformations.

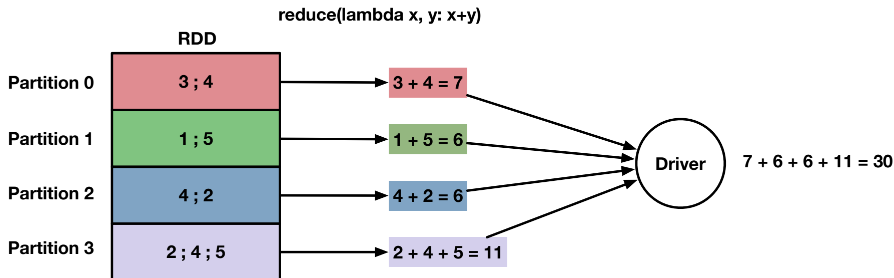
RDD actions

An **action** is an operation that takes in a RDD and returns a value to the **driver** after running a computation of the dataset.

- The result of an action is sent to the driver.
- If the result is a list of values, **all values** are sent to the driver.
- The result of an action can also be **written to disk**.
- Disk writes can be on the **local file system** or **HDFS**.

RDD actions: `reduce`

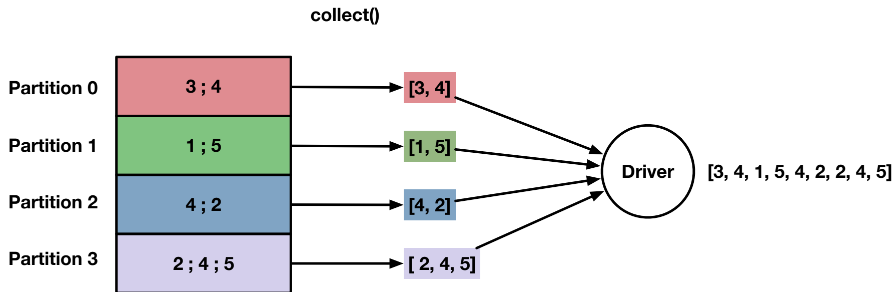
`reduce()` takes in an RDD and a function *f* and applies the function pair-wise to all elements of the input RDD.



- The function *f* **must** take in **2 arguments**.
- The type of the value returned by the function *f* must be the same as the type of the elements of the input RDD.

RDD actions: `collect`

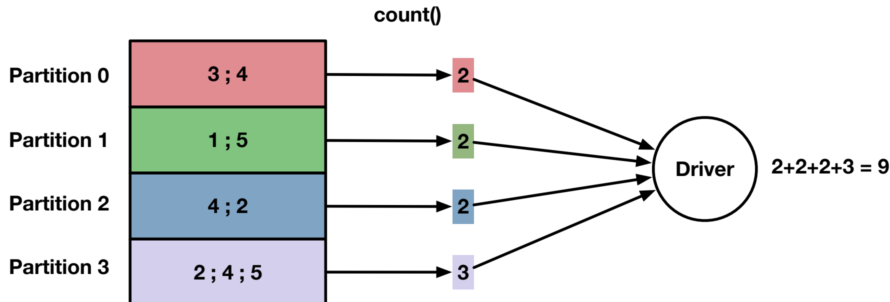
`collect()` takes in an RDD and returns the **list** of the elements in the RDD.



☞ If the RDD has a lot of items, the memory of the driver might overflow. Use `collect()` with care.

RDD actions: `count`

`count()` takes in an RDD and returns the number of items in the RDD.



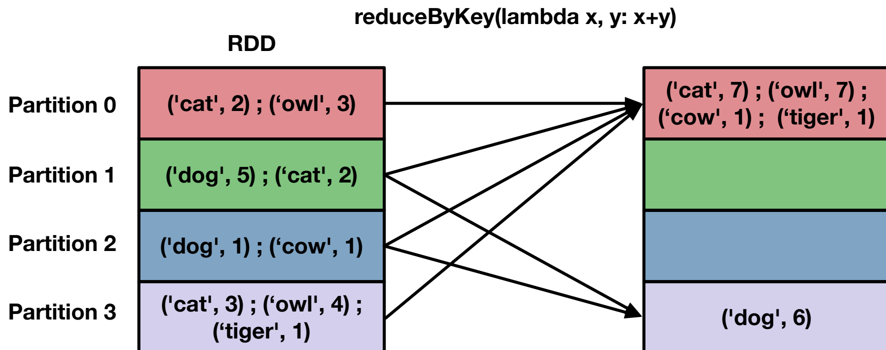
Key-value RDDs

Key-value RDDs (a.k.a., **Pair RDDs**) are RDDs where each item is a pair (k, v), k being the key and v being the value.

- Key-value RDDs are important building blocks in many applications.
- Key-value RDDs support all the transformations and actions that can be applied on regular RDDs.
- Key-value RDDs support special transformations and actions.

Key-value RDDs transformations: `reduceByKey`

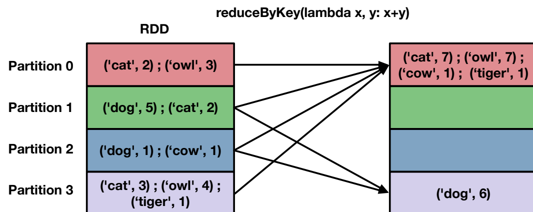
`reduceByKey` takes in a RDD with (K, V) pairs and a function f and returns a **new RDD** of (K, V) pairs where the values for each key are aggregated using f, which must be of type $(V, V) \rightarrow V$.



Key-value RDDs transformations: `reduceByKey`

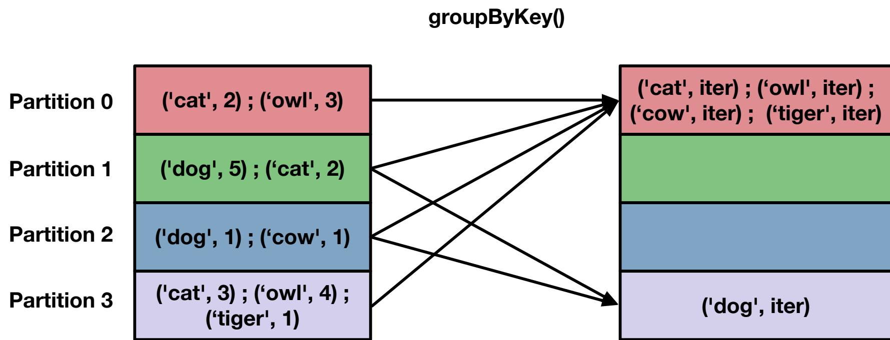
- Partitions in the input RDD is random. Items are balanced across partitions.
- The output RDD is **hash partitioned**.
- The partition number p of a pair (K, V) is derived as follows:

$$p = hashCode(K) \ mod \ num_partitions$$



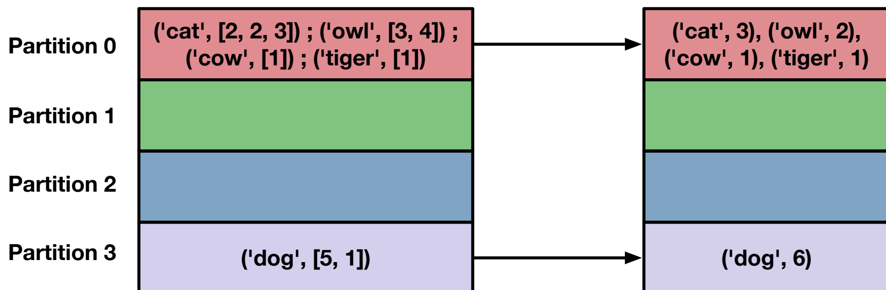
Key-value RDDs transformations: `groupByKey`

`groupByKey` takes in a RDD with (K, V) pairs and returns a **new RDD** of $(K, Iterable < V >)$ pairs.



Key-value RDDs transformations: `mapValues`

`mapValues` takes in a RDD with (K, V) pairs and a function f and returns a **new RDD** where the function f is applied to each value V (keys are not modified).



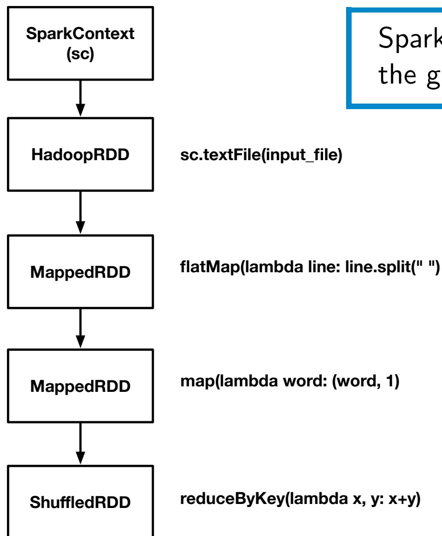
Example: Word count

```
def word_count(input_file):
    text = sc.textFile(input_file)
    return text.flatMap(lambda line: line.split(" "))\
              .map(lambda word: (word, 1))\
              .reduceByKey(lambda x, y: x+y)
```

- The function `textFile` reads a text file into a RDD.
- Two narrow transformations (`flatMap` and `map`) and one wide transformation (`reduceByKey`).

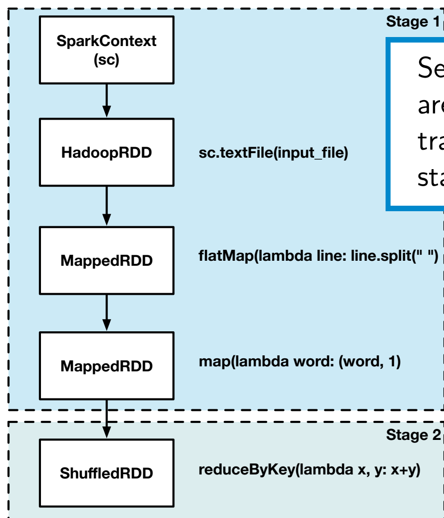
☞ Spark maintains a **logical execution plan** (called **RDD lineage**) described as a **Directed Acyclic Graph (DAG)**.

RDD lineage



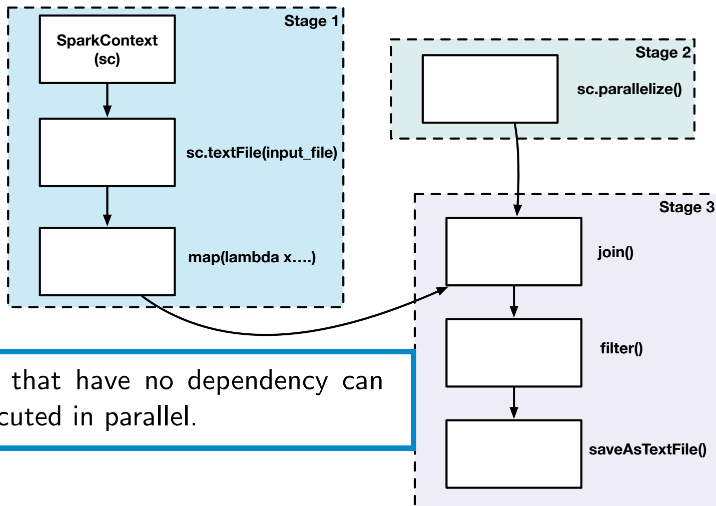
Spark has a **DAG scheduler** that splits the graph into multiple **stages**.

RDD lineage: stages



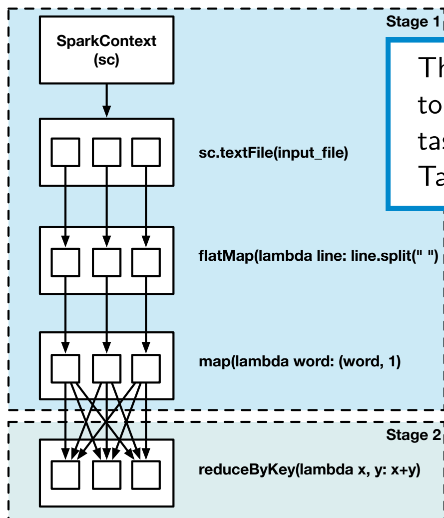
Sequences of **narrow transformations** are pipelined into a **single stage**. Wide transformations always trigger a new stage.

RDD lineage: stages



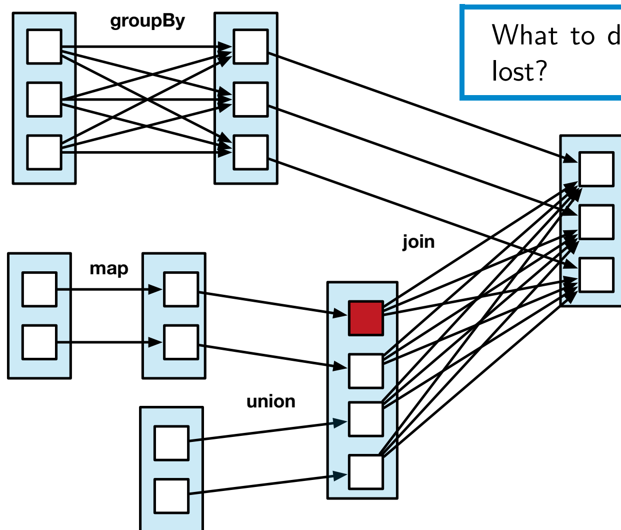
Stages that have no dependency can be executed in parallel.

RDD lineage: tasks



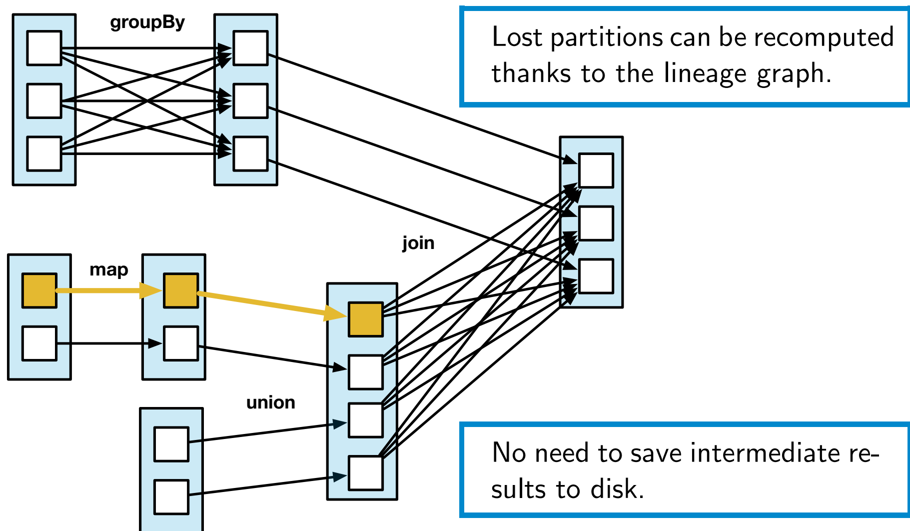
The DAG scheduler submits the stages to the **task scheduler**. Creates as many tasks as there are partitions in the RDD. Tasks are executed in parallel.

RDD lineage: fault tolerance



What to do when a partition is lost?

RDD lineage: fault tolerance



Lazy evaluation

- In Spark, transformations are **lazily evaluated**.

Definition (Lazy evaluation)

Lazy evaluation means that when a transformation is invoked, Spark **does not execute it** immediately. Transformations are only executed when Spark sees an action.

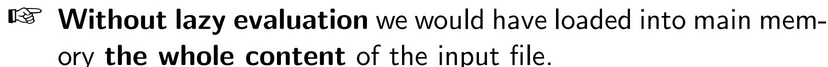
- An RDD can be thought of a set of *instructions* on how to compute the data that we build up through transformations.
- Lazy evaluation helps **reducing the number of passes** needed to load and transform the data.
  - In Hadoop, developers have to manually group operations in order to reduce the number of MapReduce iterations.
  - Spark does this optimization automatically.

Lazy evaluation

- Invoking `sc.textFile()` does not load immediately the data.
- The transformation `filter()` is not applied when it is invoked.
- Transformations are applied only when the action `count()` is invoked.
- **Only the data** that meet the constraint of the `filter` is loaded from the file.

Example

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
nb_lines = exceptions.count()
print("Number of exception lines ", nb_lines)
```



☞ **Without lazy evaluation** we would have loaded into main memory **the whole content** of the input file.

Persisting the data

- With lazy evaluation, transformations are computed **each time** an action is invoked on a given RDD.
- In the following example, all transformations are computed when we invoke the function `count()` **and** the function `collect()`.

Example

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
nb_lines = exceptions.count()
exceptions.collect()
```

- To avoid computing transformations multiple times, we can **persist** the data.

Persisting the data

- Persisting the data means **caching** the result of the transformations.
  - Either in main memory (default), or disk or both.
- If a node in the cluster fails, Spark **recomputes** the persisted partitions.
  - We can **replicate** persisted partitions on other nodes to recover from failures without recomputing.

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
exceptions.persist(StorageLevel.MEMORY_AND_DISK)
nb_lines = exceptions.count()
exceptions.collect()
```

- `persist()` is called right before the first action.
- `persist()` does not force the evaluation of transformations.
- `unpersist()` can be called to evict persisted partitions.

References

- White, Tom. *Hadoop: The definitive guide*. "O'Reilly Media, Inc.", 2012. Click here