Données massives et apprentissage profond
Lecture 1 – Apache Spark

Gianluca Quercini

gianluca.quercini@centralesuplec.fr

Polytech Paris-Saclay, 2022

CentraleSupélec

Organization of the course

- MapReduce and Spark.
- Spark programming.
- SQL and NoSQL.
- MongoDB practice.
- Hadoop technologies.
- Scaling.

Class material

Available online
https://tinyurl.com/p7jb5wra
Click here

- Slides of the lectures.
- Tutorials and lab assignments.
- References (books and articles).

Evaluation

- **Lab assignments**. Lab assignments 1 and 2 will be graded.
  - Lab assignment 1. Spark programming
  - Lab assignment 2. MongoDB
  - **Submission**: Code source + written report.
- **Written exam.** 1 hour.
  - Spark programming.
  - Data modeling in MongoDB.
  - Querying in MongoDB.

Contact

Email: gianluca.quercini@centralesupelec.fr

Email: stephane.vialle@centralesupelec.fr

What you will learn

In this lecture you will learn:

- What **Spark** is and its main features.
- The components of the **Spark stack**.
- The high-level **Spark architecture**.
- The notion of **Resilient Distributed Dataset** (RDD).
- The main **transformations** and **actions** on RDDs.

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** (Hadoop relies on disks).
- **General-purpose**. Different workloads in the same system.
  - Batch applications, iterative algorithms.
  - Interactive queries, streaming applications.
- **Accessibility.** Python, Scala, Java, SQL and R; rich built-in libraries.
- **Integration.** With other Big Data tools, such as **Hadoop**.

Spark components

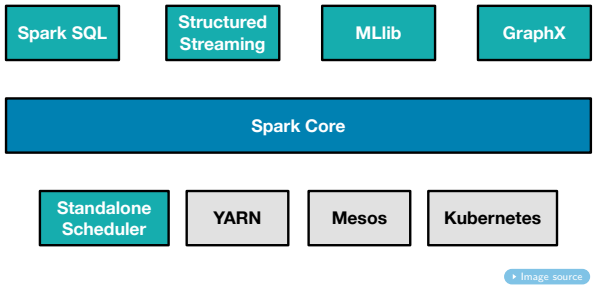


Image source

Spark components

Spark core

- **Scheduling**, **distributing**, and **monitoring** applications.
- Data structures for manipulating data (RDDs, DataFrames).

Spark SQL

- Spark's package for working with **(semi-)structured data**.
- Data querying with **SQL** and **HQL** (Hive Query Language).
- Many sources of data: JSON, XML, Parquet...

Spark components

Structured streaming

- Processing of live **streams** of data (e.g., real-time event logs)
- Similar API to batch processing.

MLlib

- **Machine learning** algorithms (e.g., classification, regression, clustering)
- All methods designed to scale out across a cluster.

Spark components

GraphX

- Manipulation of **graph data**.
- Library with common graph algorithms (e.g., PageRank)

Cluster managers

- Control how tasks are distributed across a cluster.
- Spark provides its own **standalone** cluster manager.
- Spark can also use other cluster managers.

Spark unified stack: benefits

- **Shallow learning curve.** Same **programming model** across all components.
- **Optimization propagation.** Higher-level components automatically benefit from improvements on lower-layer components.
- **Cost minimization.** No need for further software components.
- **Heterogeneous processing models** in the same application.
  - Read a stream of data.
  - Apply machine learning algorithms.
  - Uses SQL to analyze the results.

Using Spark

Interactive mode

Using a command-line interface (CLI) or **shell**.

- Python and Scala shell.
- SparkSQL shell.
- SparkR shell.

Data processing applications

Building an **application** by using the **Spark APIs**.

- Scala (Spark's native language).
- Python.
- Java.

Who uses Spark

Several important actors use 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 application

- Spark application: set of independent processes called **executors**.
- Executor run computations and store the data for the application.
- Executors are coordinated by the **driver**.

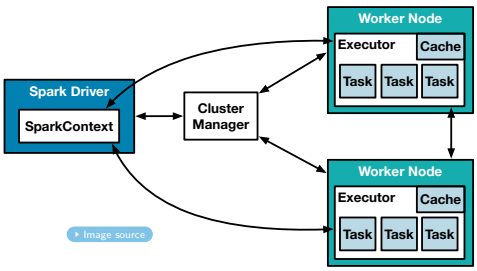
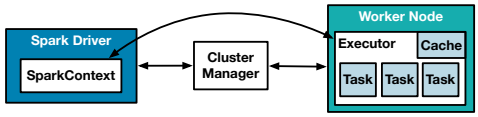


Image source

Spark application execution

- The **driver** is launched and creates the **SparkContext** object.
- The **SparkContext** obtains **executors** from the **cluster manager**.
- The driver sends the **user's code** to the executors.
- The driver assigns each executor a set of **tasks**.
- A **task** is a computation on a **chunk of data**.



Spark application execution

- Applications are **isolated** from one another.
  - Each application has its own SparkContext.
  - An executor only runs tasks of one application.
  - A driver only schedules tasks for one application.
  - Data cannot be shared across different applications.
- Spark is **agnostic** to the underlying cluster manager.
- The driver listens to incoming connections from the executors on a network port.
- The driver should be in the same local network as the executors.

☞ Two different Spark applications can still share data through an external storage system (e.g., a database or HDFS files).

Spark programming

Two options exist to write a Spark application:

- **Low-level** programming, using operations on a low-level data structure called **Resilient Distributed Dataset (RDD)**.
- **High-level** programming, using high-level libraries, such as SparkSQL and Structured Streaming.

☞ In this lecture, we'll focus on low-level programming to better understand the inner workings of Spark.

Low-level Spark programming

- A Spark program uses 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 (sc)**.
- These operations manipulate a special type of data structure, called **Resilient Distributed Dataset (RDD)**.

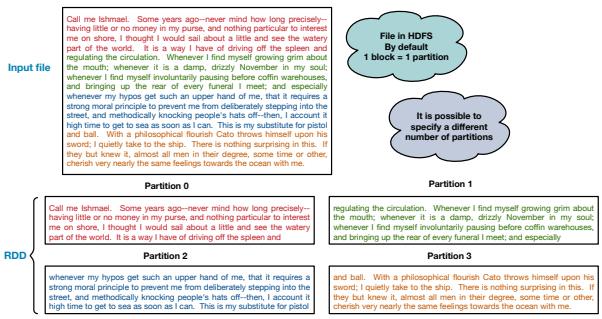
Resilient Distributed Dataset (RDD)

Definition (Resilient Distributed Dataset)

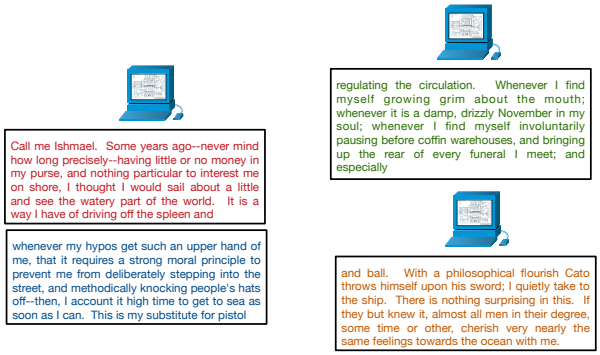
A **Resilient Distributed Dataset**, or simply **RDD**, is an **immutable**, **distributed** collection of objects. Source

- 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)



Resilient Distributed Dataset (RDD)



Creating an RDD

1. From an **in-memory collection** (e.g., a list or a set).

```
sc.parallelize([1, 5, 3, 2, 6, 7])
```

> ☞ This method is used for **debugging** and **prototyping** on **small datasets**.

2. From an **data source on disk** (e.g., a file or a database).

```
sc.textFile("hdfs://sar01:9000/data/sample_text.txt")
```

> ☞ This method is used **in production** to process **large datasets**.

About the number of partitions

RDDs created with `parallelize`

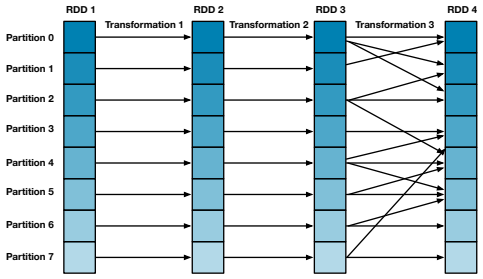
- **Local mode**: number of cores on the local machines.
- **Cluster mode**: total **number of cores** on all executor nodes, or 2, whichever is larger.

RDDs created from files stored in HDFS

- Number of HDFS blocks in the input file, or 2, whichever is larger.

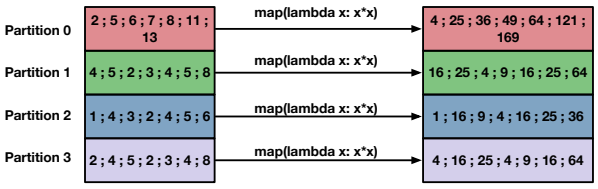
RDD transformations

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`

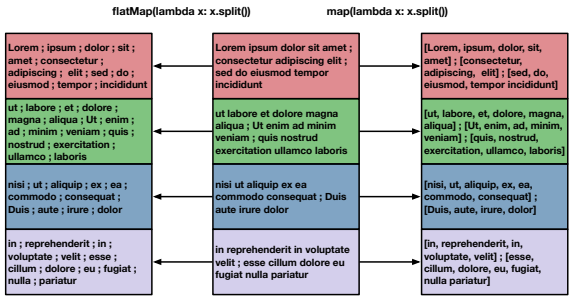
`map()` takes in a **function** f and a RDD $< x_i \mid 0 \leq i \leq n >$; returns a **new RDD** $< f(x_i) \mid 0 \leq i \leq n >$.



☞ Partition i of the input RDD is on the same node as partition i of the output RDD.

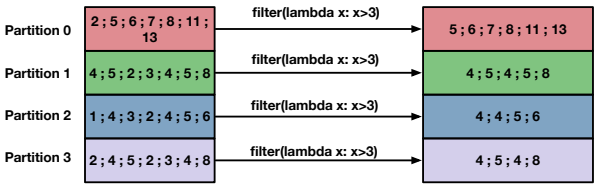
RDD transformations: `flatMap`

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



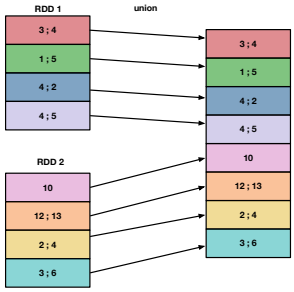
RDD transformations: `filter`

`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* $>$



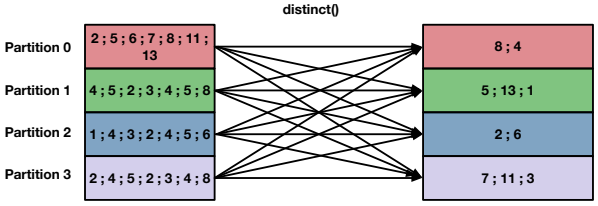
RDD transformations: `union`

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



RDD transformations: `distinct`

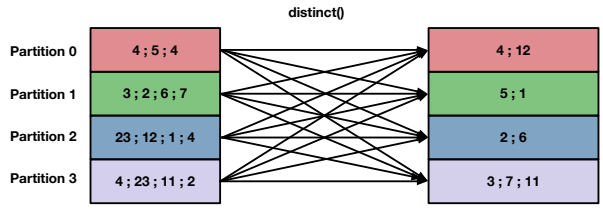
`distinct()` takes in one RDD and returns a **new RDD** containing the items of the input RDD **without repetitions**.



☞ Unlike the previous transformations, `distinct` leads to data being **shuffled**.

About data shuffling ★

Which partition does the element 23 belong to in the RDD obtained after applying the transformation `distinct`?



About data shuffling ★

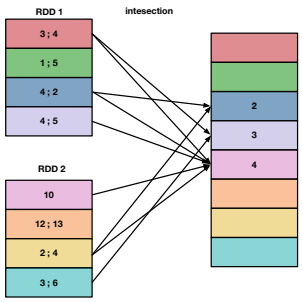
The element 23 belongs to the **partition 3**.

While shuffling, the **destination partition** p of an element K in a RDD with n partitions is computed as follows:

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

RDD transformations: `intersection`

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



Narrow transformations ★

Definition (Narrow transformation)

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

Which of the above transformations are narrow?

- Narrow transformations are **inexpensive**.
- No need for **communication** between executors.

Narrow transformations ★

Definition (Narrow transformation)

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

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

- Narrow transformations are **inexpensive**.
- No need for **communication** between executors.

Wide 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.

Which of the above transformations are wide?

- Wide transformations are more **costly**.
- Executors need to communicate.
- Data is **shuffled** across the cluster network.

Wide 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**.
- Executors need to communicate.
- Data is **shuffled** across the cluster network.

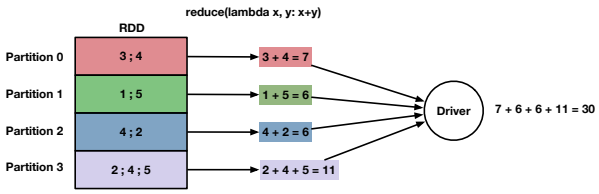
RDD actions

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 to the **local file system** or **HDFS**.

RDD actions: `reduce`

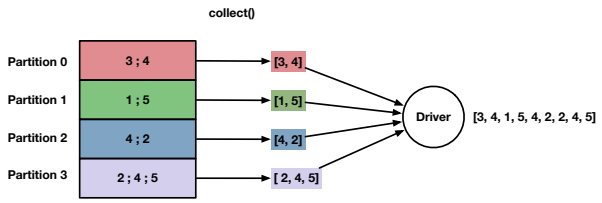
`reduce()` takes in an RDD and a function f and applies the function pair-wise to all elements of the input RDD.



- 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.



Is `collect()` a safe action? ★

What are the risks, if any, while invoking `collect()` on a large RDD?

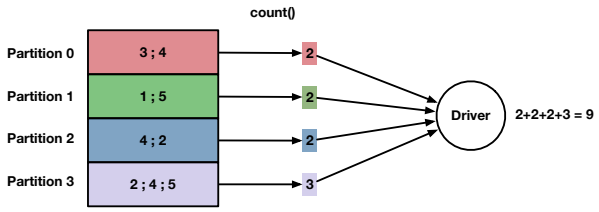
Is `collect()` a safe action? ★

What are the risks, if any, while invoking `collect()` on a large RDD?

- High network traffic.
- The driver's memory may not enough to store all the RDD elements.

RDD actions: count

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



Understanding the code ★

What does the following code?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.map(lambda x: x.capitalize())
```


Understanding the code ★

What does the following code?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.map(lambda x: x.capitalize())
```

- `r2` is an RDD (result of a transformation).
- `r2` has as many elements as `r1`.
- Each item of `r2` is a string from `r1` with the first letter capitalized.

What does the following code? ★

What does the following code?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.filter(lambda x: len(x) > 10)
```


What does the following code? ★

What does the following code?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.filter(lambda x: len(x) > 10)
```

- `r2` is an RDD (result of a transformation).
- `r2` has less elements than `r1`.
- `r2` only contains the items from `r1` that have more than 10 characters.

What does the following code? ★

What does the following code?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.reduce(lambda x, y: "{} - {}".format(x, y))
```

What does the following code? ★

What does the following code?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.reduce(lambda x, y: "{} - {}".format(x, y))
```

- `r2` is a string, **not a RDD** (result of an action).
- `r2` is the string "computer science - geology - chemistry - biology - astronomy".

What does the following code? ★

What does the following code?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.reduce(lambda x, y: [x + y])
```


What does the following code? ★

What does the following code?

```
r1 = sc.parallelize(["computer science", "geology", \
    "chemistry", "biology", "astronomy"])
r2 = r1.reduce(lambda x, y: [x + y])
```

The code is incorrect, because the return type (`list`) of the `reduce` function is different from the type of the input RDD elements (`string`).

What does the following code? ★

What does the following code?

```
r1 = sc.parallelize(["author", "title", "edition"])
r2 = r1.flatMap(lambda x: [c for c in x])
```


What does the following code? ★

What does the following code?

```
r1 = sc.parallelize(["author", "title", "edition"])
r2 = r1.flatMap(lambda x: [c for c in x])
```

- `r2` is an RDD (result of a transformation).
- Each element of `r2` is a letter from a string in `r1`. How would that be different if we had used `map` instead of `flatMap`?

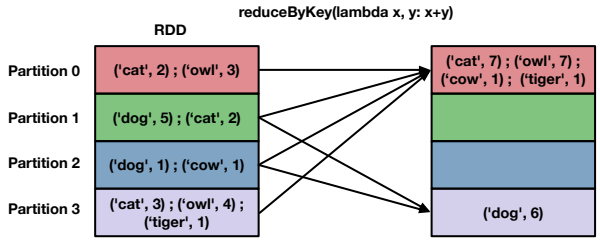
Key-value RDDs

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 $(V, V) \rightarrow V$.



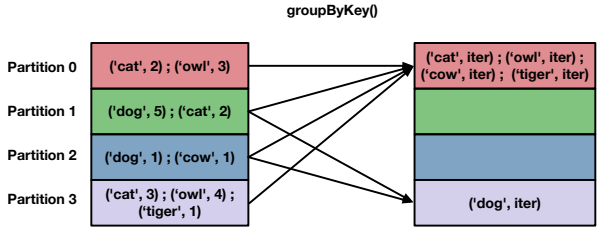
Key-value RDDs transformations: `reduceByKey`

- The input RDD has a certain number of partitions n.
- No assumption can be made on which elements belong to which partition.
- The RDD returned by `reduceByKey` is **hash partitioned**. Each item belongs to a precise partition.
- The partition number p of a pair (K, V) is derived as follows:

$$p = hashCode(K)\ mod\ num\_partitions$$

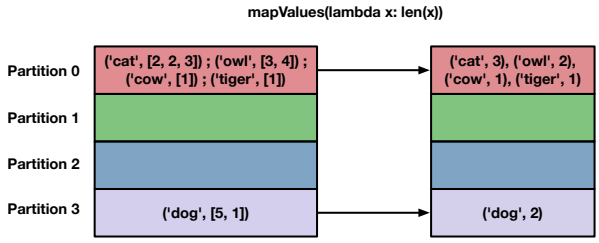
Key-value RDDs transformations: `groupByKey`

`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).



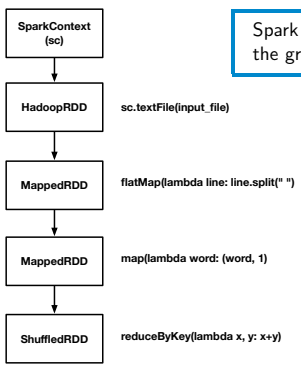
Example: Word count

```
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`).

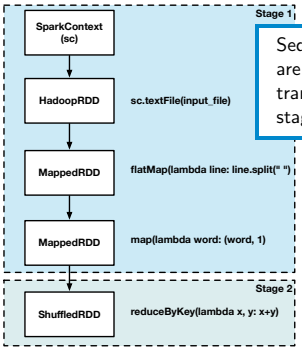
☞ Spark maintains a **logical execution plan** (called **RDD lineage**) described as a **Directed Acyclic Graph (DAG)**.

RDD lineage



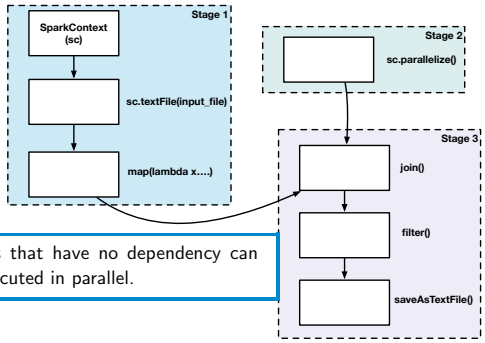
Spark has a **DAG scheduler** that splits the graph into multiple **stages**.

RDD lineage: stages



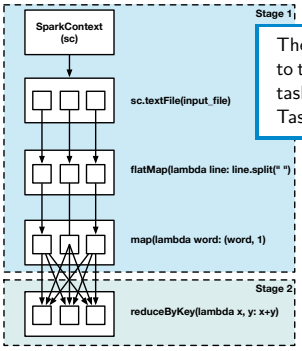
Sequences of **narrow transformations** are pipelined into a **single stage**. Wide transformations always trigger a new stage.

RDD lineage: stages



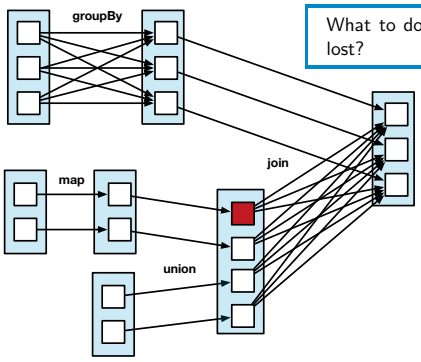
Stages that have no dependency can be executed in parallel.

RDD lineage: tasks



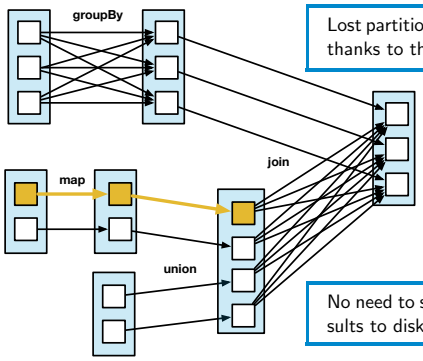
The DAG scheduler submits the stages to the **task scheduler**. Creates as many tasks as there are partitions in the RDD. Tasks are executed in parallel.

RDD lineage: fault tolerance



What to do when a partition is lost?

RDD lineage: fault tolerance



Lost partitions can be recomputed thanks to the lineage graph.

No need to save intermediate results to disk.

Lazy evaluation

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

Definition (Lazy evaluation)

When a transformation is invoked, Spark **does not execute it** immediately. Transformations are only executed when **the first action** is called.

- 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.

Lazy evaluation: motivating 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)
```

What happens if Spark executes **immediately** each transformation?

Lazy evaluation: motivating example ★

- 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.

```
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.

Lazy evaluation: consequences ★

- The following code invokes **two actions**: which ones?
- What happens when we invoke the **second action**?

Example

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


Lazy evaluation: consequences ★

- 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

- Karau, Holden, et al. *Learning spark: lightning-fast big data analysis*. O'Reilly Media, Inc., 2015. Click here

Playing with transformations and actions

Notebook available on Google Colab ▸ Click here

☞ Select File → Save a copy in Drive to create a copy of the notebook in your Drive and play with it.