Big Data Processing

L06-11: Spark Core Model of Parallel Computing: RDDs

Dr. Ignacio Castineiras
Department of Computer Science

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
6. Lazy Evaluation.

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
6. Lazy Evaluation.

Introduction to Apache Spark

Apache Spark is an

- open-source
- distributed
- general-purpose
- cluster-computing

framework designed for 3 purposes:

Introduction to Apache Spark

Apache Spark is an

- open-source
- distributed
- general-purpose
- cluster-computing

framework designed for 3 purposes:

1. Be easy to use - you can develop applications locally, using a high-level API letting you focus on the content of your computation.

Introduction to Apache Spark

Apache Spark is an

- open-source
- distributed
- general-purpose
- cluster-computing

framework designed for 3 purposes:

1. Be easy to use - you can develop applications locally, using a high-level API letting you focus on the content of your computation.
2. Be fast - enabling interactive use and complex algorithms.

Introduction to Apache Spark

Apache Spark is an

- open-source
- distributed
- general-purpose
- cluster-computing

framework designed for 3 purposes:

1. Be easy to use - you can develop applications locally, using a high-level API letting you focus on the content of your computation.
2. Be fast - enabling interactive use and complex algorithms.
3. Be general - allowing to combine multiple types of computations, including text, SQL, graph and machine learning processing (both offline and online) that might previously have required different engines.

Introduction to Apache Spark

Apache Spark is an

- open-source
- distributed
- general-purpose
- cluster-computing

framework designed for 3 purposes:

1. Be easy to use - you can develop applications locally, using a high-level API letting you focus on the content of your computation.
2. Be fast - enabling interactive use and complex algorithms.
3. Be general - allowing to combine multiple types of computations, including text, SQL, graph and machine learning processing (both offline and online) that might previously have required different engines.

All in all, Spark enables to process large quantities of data, beyond what can fit on a single machine, with a high-level, transparent parallelizable, system agnostic and relatively easy-to-use API.

Introduction to Apache Spark

- Spark started in 2009 as a research project in the UC Berkeley RAD Lab, later to become the AMPLab: https://amplab.cs.berkeley.edu/

Introduction to Apache Spark

- Spark started in 2009 as a research project in the UC Berkeley RAD Lab, later to become the AMPLab: https://amplab.cs.berkeley.edu/
- Spark was later donated to the Apache Software Foundation, which has maintained it since: http://spark.apache.org/

  Nowadays, Spark has over 1,000 contributors.

Introduction to Apache Spark

- Spark started in 2009 as a research project in the UC Berkeley RAD Lab, later to become the AMPLab: https://amplab.cs.berkeley.edu/
- Spark was later donated to the Apache Software Foundation, which has maintained it since: http://spark.apache.org/
  Nowadays, Spark has over 1,000 contributors.
- On top of the Apache avenue, the creators of Spark also created Databricks, a spin-off company leveraging Spark as a cloud-based big data processing tool https://databricks.com/. It would be our access point to Spark this semester.

Introduction to Apache Spark

- Spark is the state-of-the-art computational engine for big data analytics, and it is used by some of the most important companies, including:

- The full list of companies creating products and projects for use with Apache Spark can be seen at: http://spark.apache.org/powered-by.html

Introduction to Apache Spark

- Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM). It offers simple APIs the proper Scala, and also in Java, Python and R.

Introduction to Apache Spark

- Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM).
  It offers simple APIs the proper Scala, and also in Java, Python and R.

- In this module we are going to work with Spark in Python. Full stop.
  That being said, for completeness and, in some cases, to ease some explanations, for each code example presented in this module, I will provide you with the Python and Scala versions of it.
  But, once again, I don't expect you to write a single line of code in Scala.

Introduction to Apache Spark

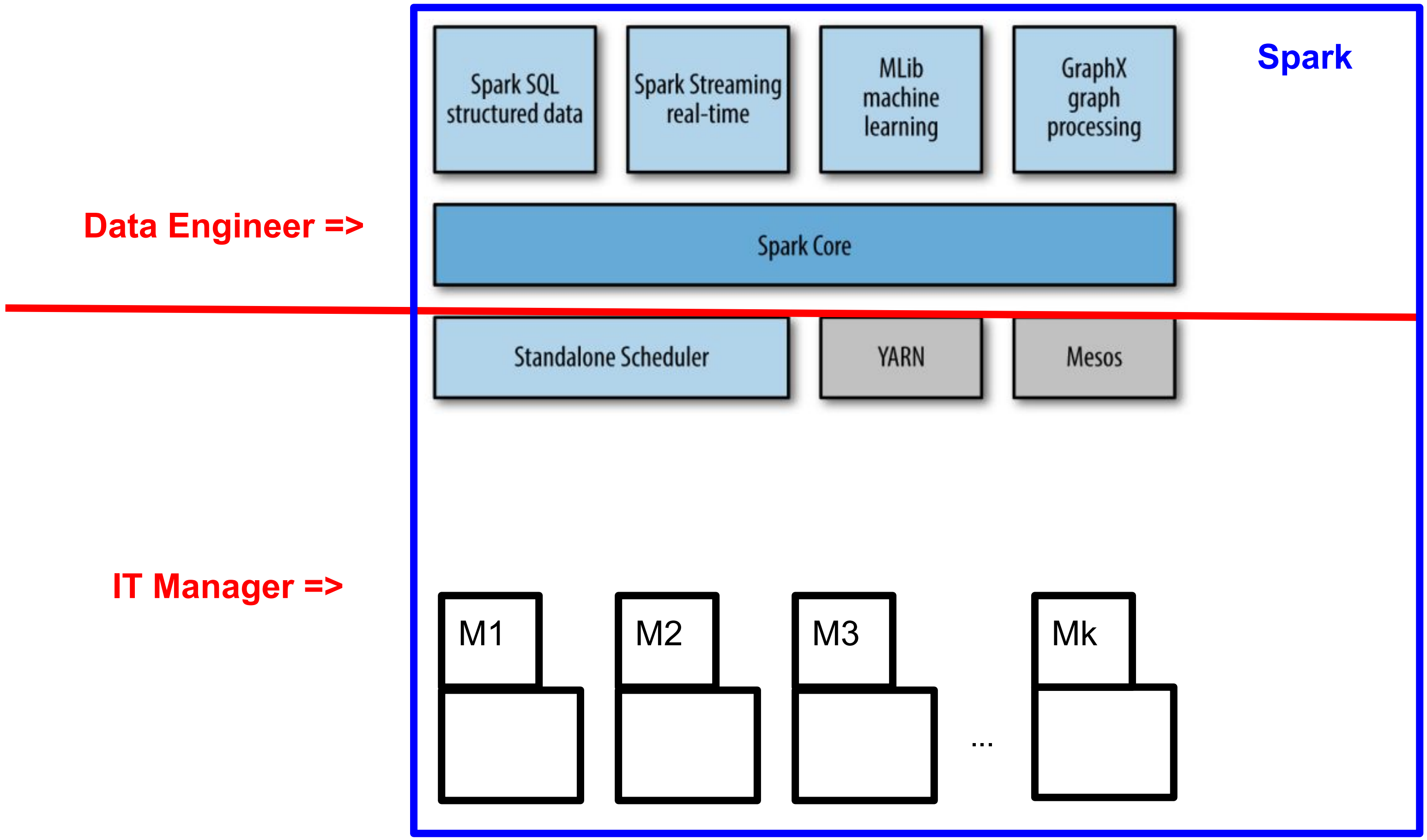
- Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM). It offers simple APIs the proper Scala, and also in Java, Python and R.

- In this module we are going to work with Spark in Python. Full stop. That being said, in some cases and to ease some explanations, for the code examples of Spark Core I will provide you with the Scala version as well. But, once again, I don't expect you to write a single line of code in Scala.

- It is important to remark, though, that the Spark code written in Python is often slower than equivalent code written in the JVM, since Scala is statically typed, and the cost of JVM communication (from Python to Scala) can be very high.

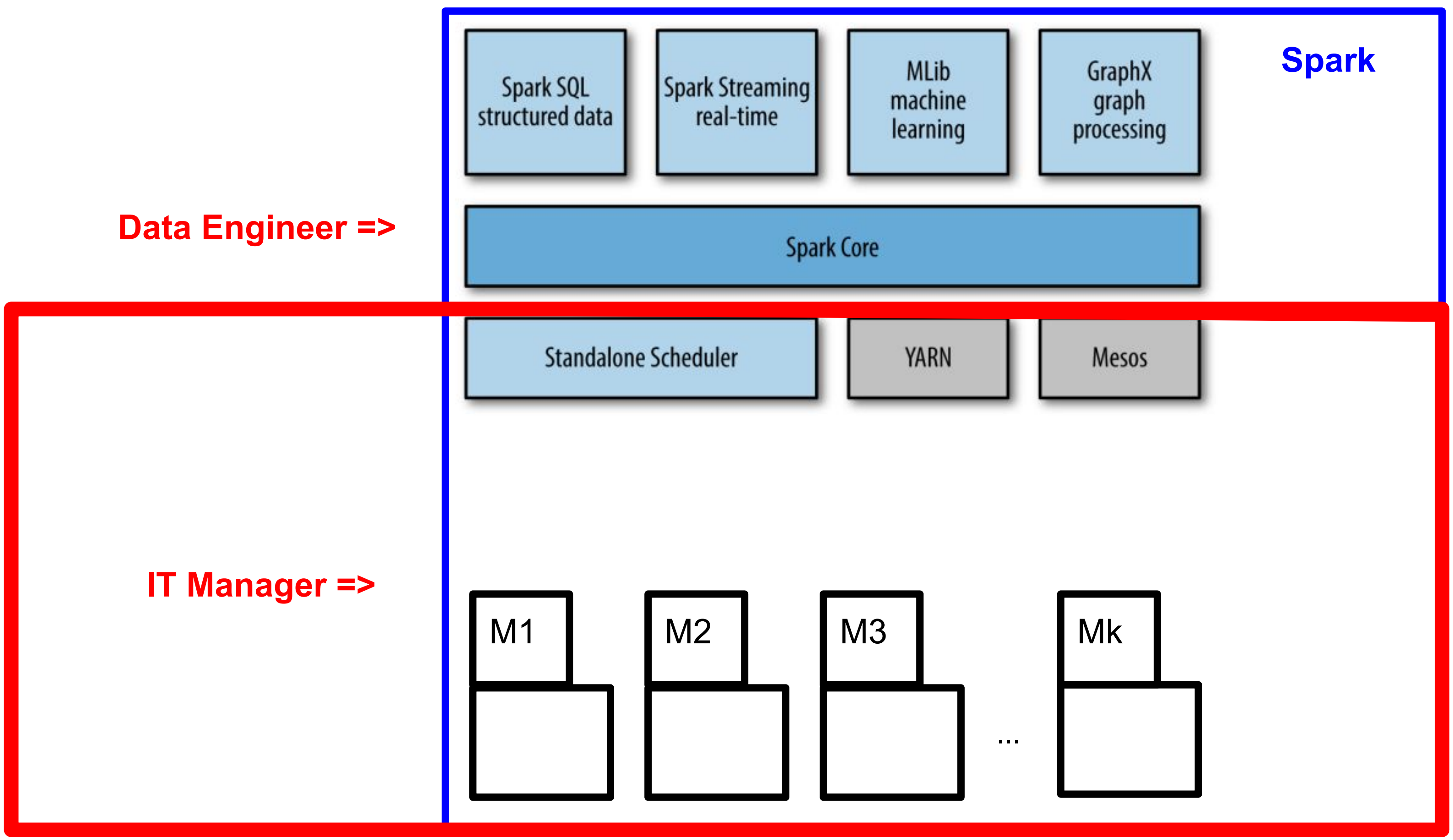
Introduction to Apache Spark

This picture presents the typical operation of Spark in a cluster. On it we can see the distinction between the role of IT Manager and Data Engineer.



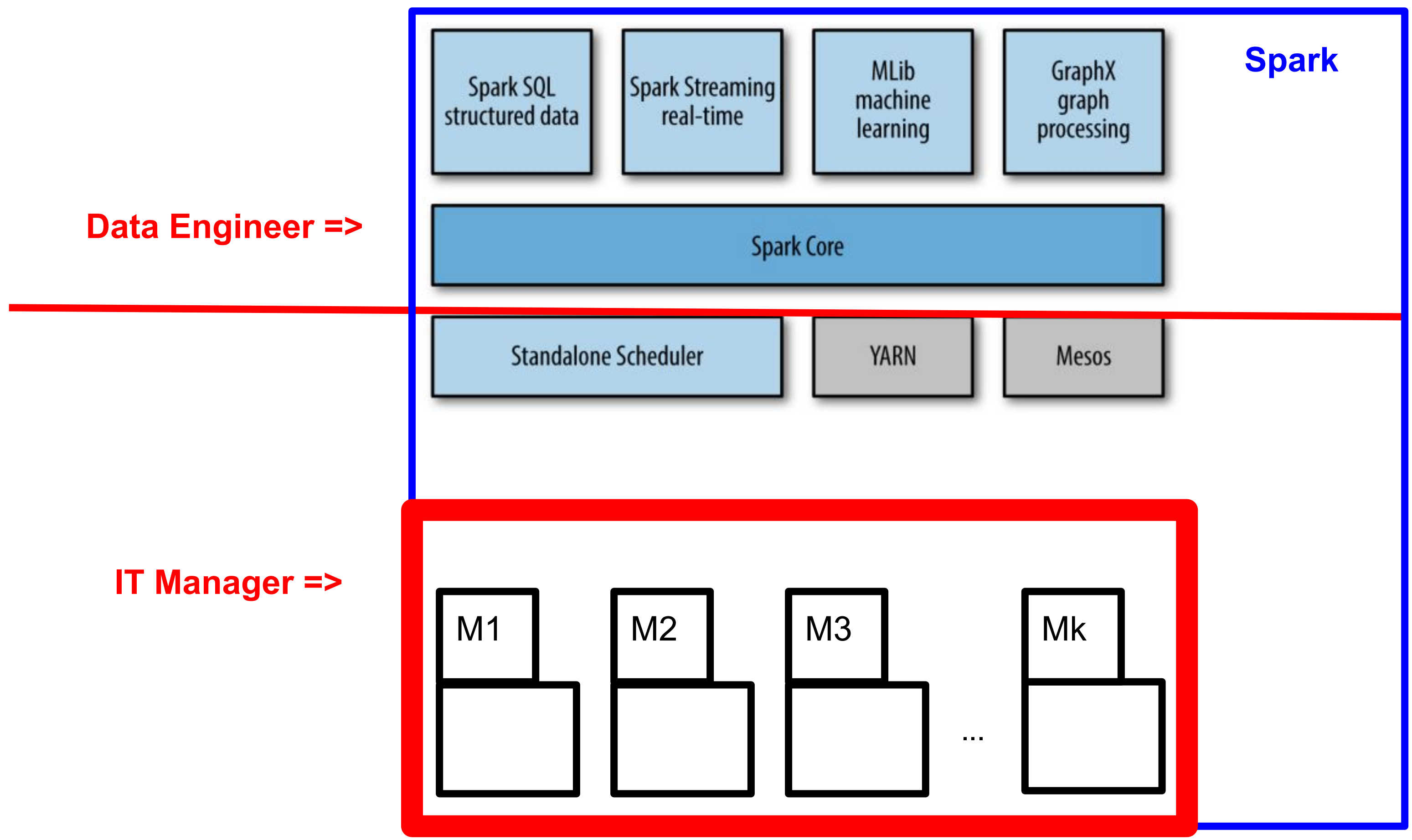
Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer



Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer



Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer

- The fact of Spark being a cluster-computing framework for data analysis requires to have a cluster on the first place.

Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer

- The fact of Spark being a cluster-computing framework for data analysis requires to have a cluster on the first place.

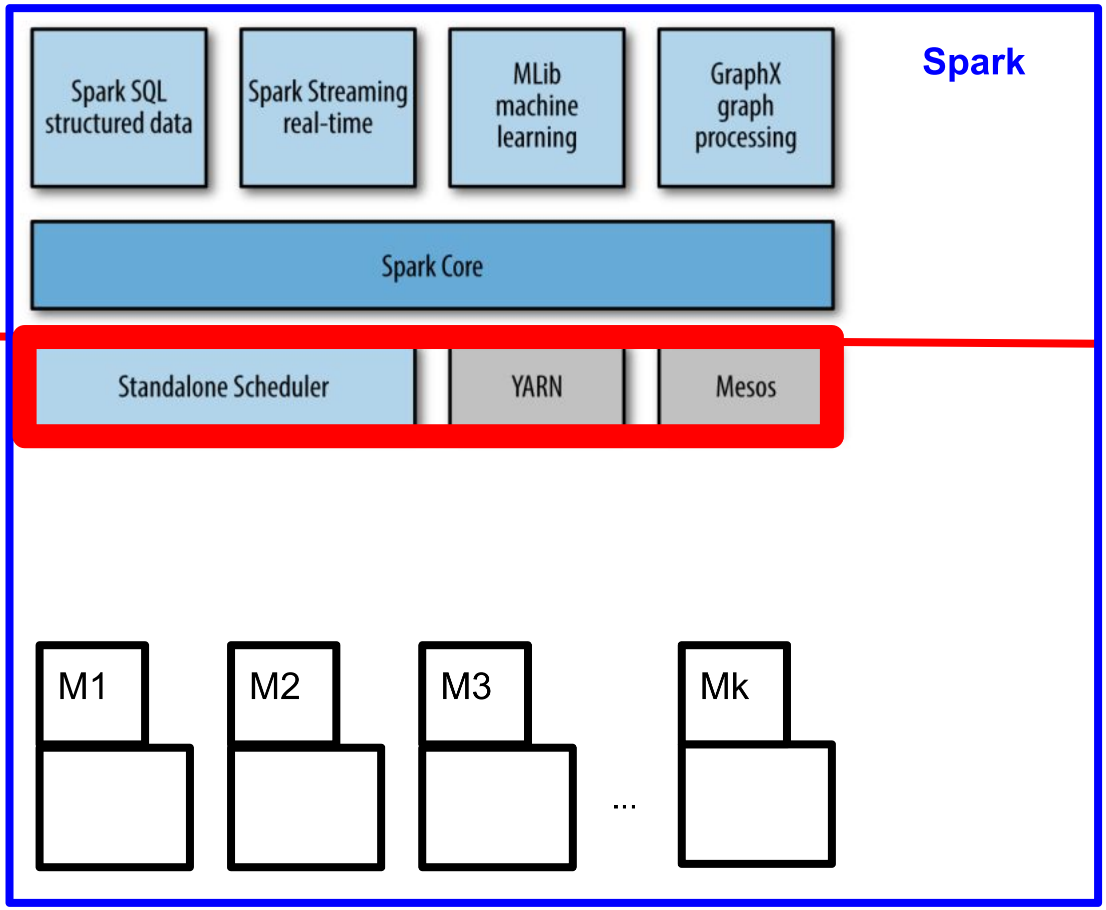
- Having a cluster of n > 1 computers is not complicated,
  but to have it *"ready"* for using Spark has additional requirements:
  1. All computers must be able to communicate.
  2. A common Spark version has to be installed across the cluster.
  3. A Distributed File System has to be in place to offer a global storage view of the cluster.
  4. OS and security updates have to be taken into account as well.

Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer

Data Engineer =>

IT Manager =>



Two Different Roles: IT Manager vs. Data Engineer

Two different Roles: IT Manager and Data Engineer

- On its own, Spark is not a data storage solution; it performs computations on Java Virtual Machines (JVMs) that last only for the duration of a Spark application.

Two Different Roles: IT Manager vs. Data Engineer

Two different Roles: IT Manager and Data Engineer

- On its own, Spark is not a data storage solution; it performs computations on Java Virtual Machines (JVMs) that last only for the duration of a Spark application.

- Spark can be run locally on a single machine with a single JVM (called local mode). However, Spark is designed to efficiently scale up from one to many thousands of compute nodes.

Two Different Roles: IT Manager vs. Data Engineer

Two different Roles: IT Manager and Data Engineer

- On its own, Spark is not a data storage solution; it performs computations on Java Virtual Machines (JVMs) that last only for the duration of a Spark application.

- Spark can be run locally on a single machine with a single JVM (called local mode). However, Spark is designed to efficiently scale up from one to many thousands of compute nodes.

- When used in a cluster, Spark is used in tandem with a distributed storage system and a cluster manager.
  - The storage system is used to provide the input dataset used by Spark application, as well as to stable store the results such application produces.
  - The cluster manager is used to orchestrate the distribution of Spark applications across the cluster.

Two Different Roles: IT Manager vs. Data Engineer

Two different Roles: IT Manager and Data Engineer

- This cluster manager is in charge of putting in contact the storage of the dataset with its further processing (via one or more of the data engineering components).
  Spark can run over a variety of cluster managers, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler.

Two Different Roles: IT Manager vs. Data Engineer

Two different Roles: IT Manager and Data Engineer

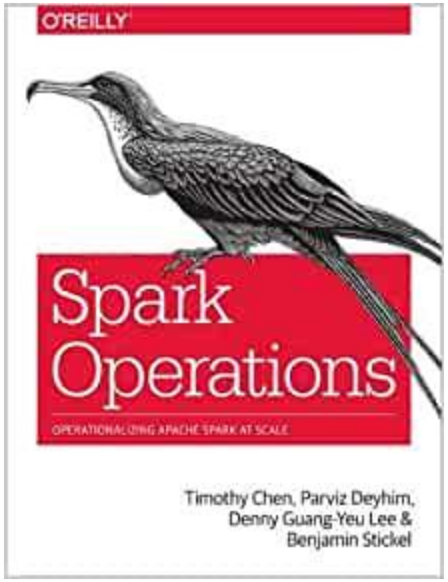
- This cluster manager is in charge of putting in contact the storage of the dataset with its further processing (via one or more of the data engineering components).
  Spark can run over a variety of cluster managers, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler.

- In terms of file storage, Spark can work on Hadoop Distributed File System (HDFS) or similar (when using Spark with the Databricks online system we will use their specific Databricks File System - DBFS).

Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer

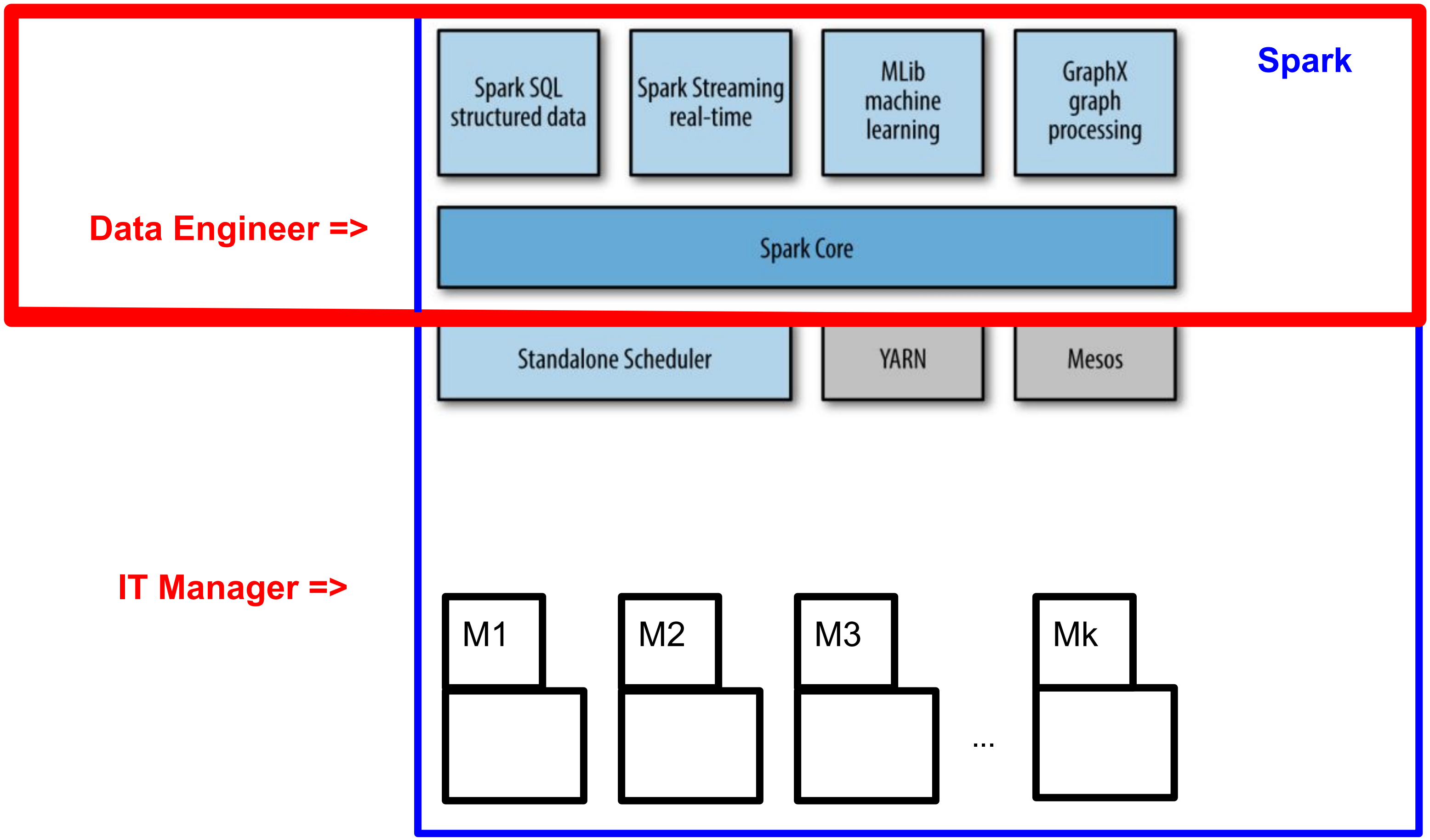
- All these tasks make room for a role as Cluster IT Manager, which is out of the scope of this module, and thus we are not going to cover at all.
  If you want to learn more about the topic perhaps a good book is:
  - Spark Operations: Operationalizing Apache Spark at Scale.
    Timothy Chen et al. O'Reilly, 2016.



Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer

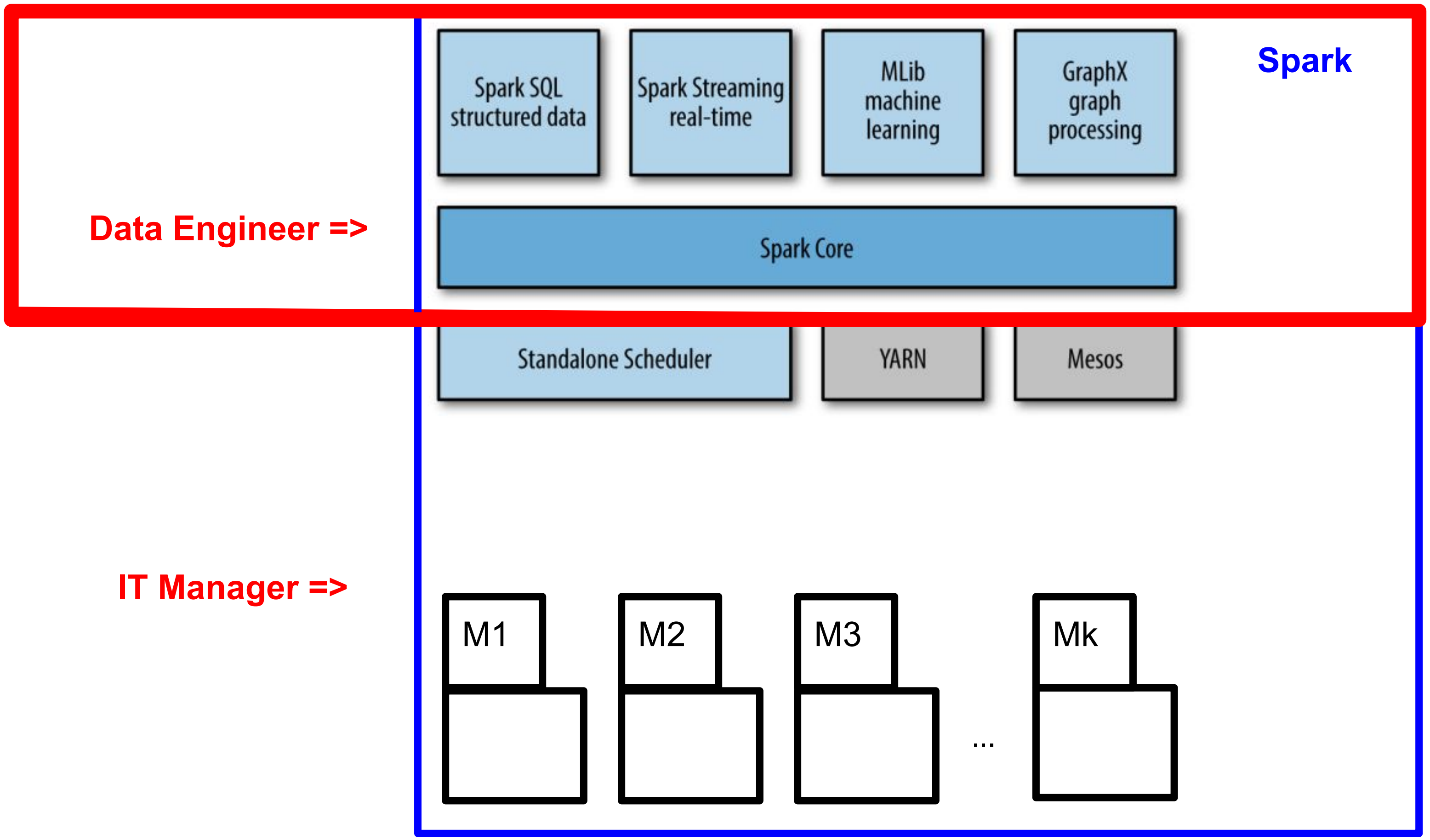
In this semester we will just focus on the role of Data Engineer.



Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer

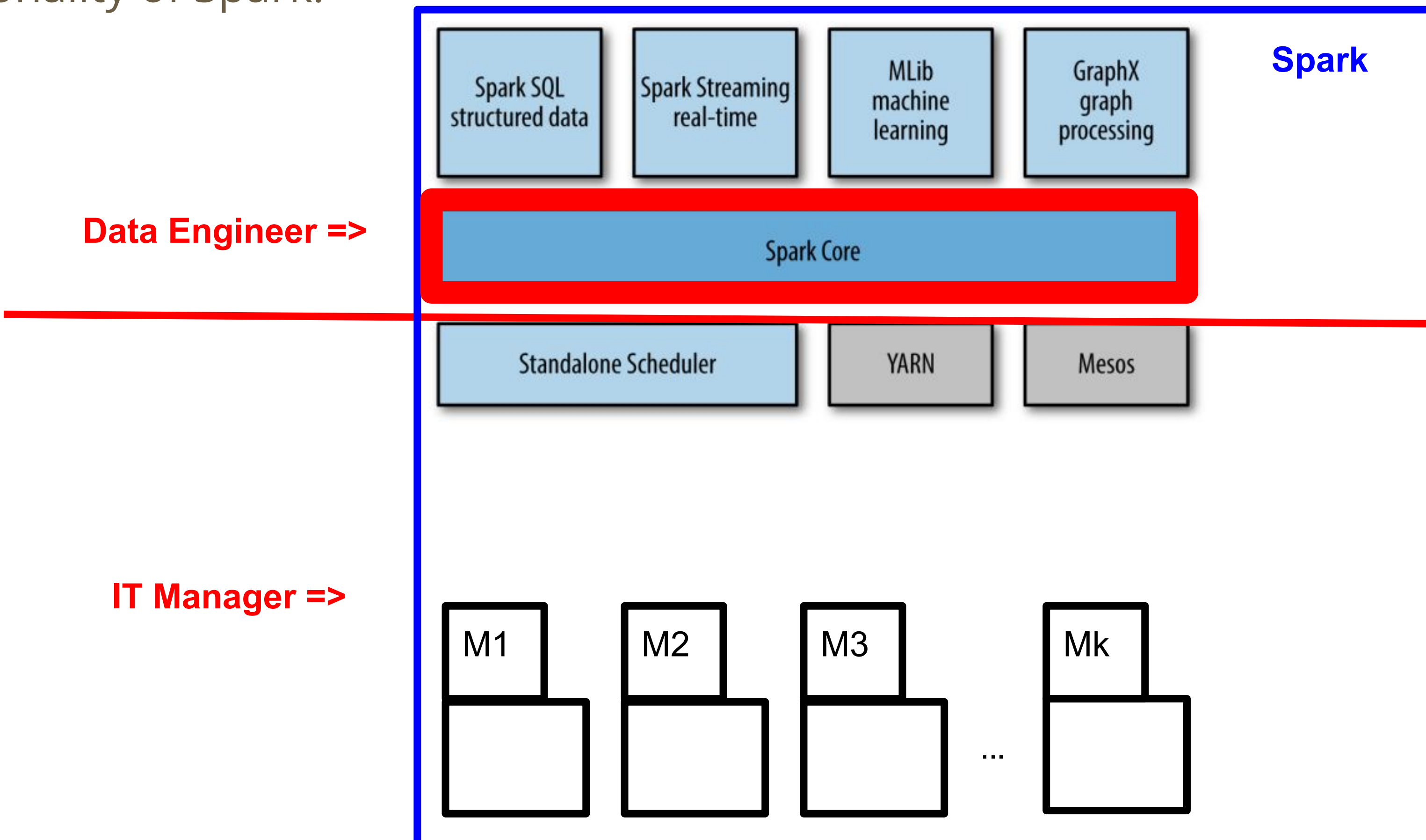
This Data Engineer uses Spark on top of a *"ready"* cluster.



Introduction to Apache Spark

Two different Roles: IT Manager and Data Engineer

- We start today with Spark Core, the component containing all the main functionality of Spark.



Outline

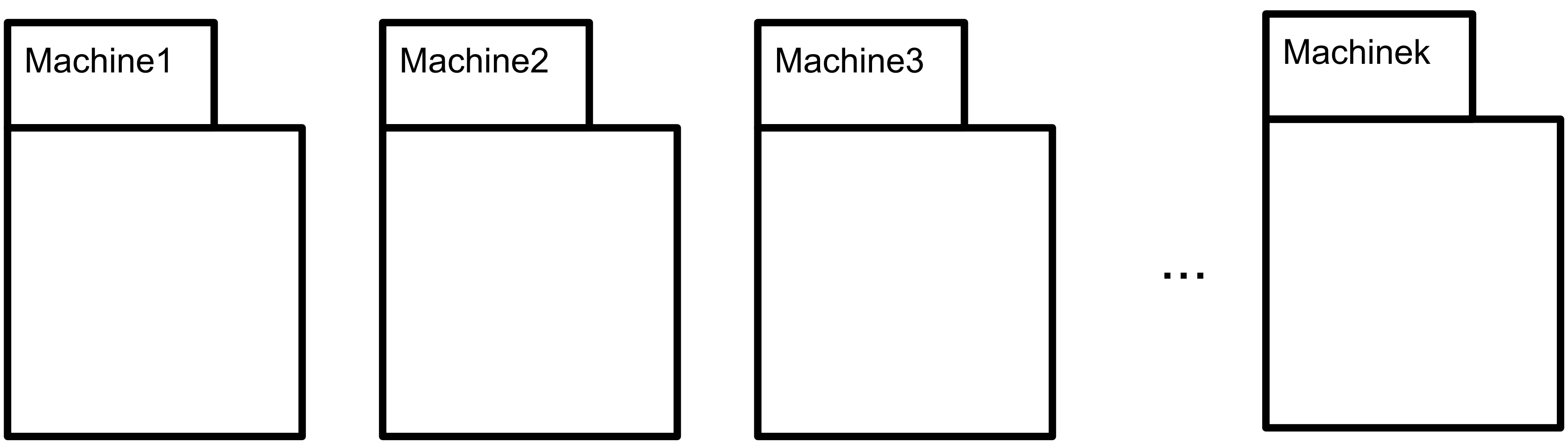
1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
6. Lazy Evaluation.

Setting the Context

Let's put together all the ingredients we have mentioned so far…

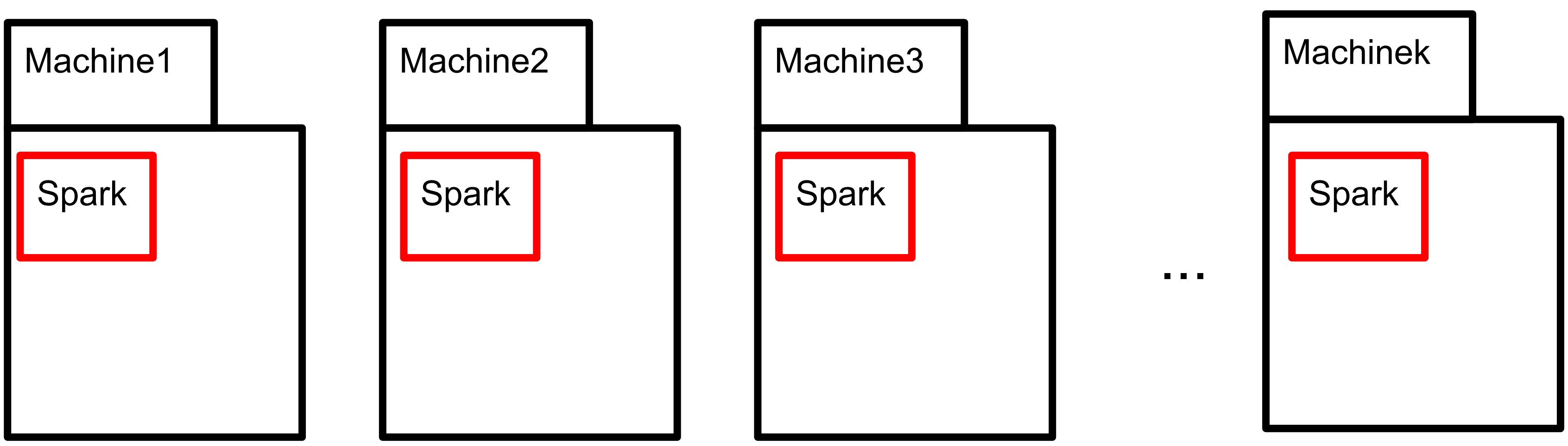
Setting the Context

1. We need a **cluster of computers**, connected among them so as to support the distributed computation.



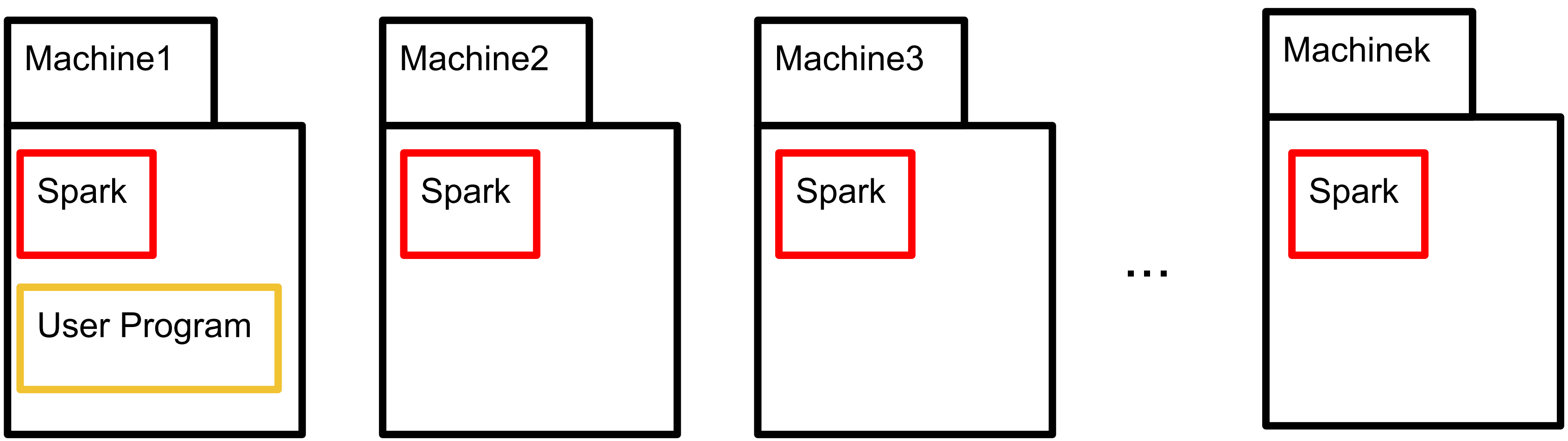
Setting the Context

2. These computers need **Spark** to be installed on them, to proceed with the desired computation.



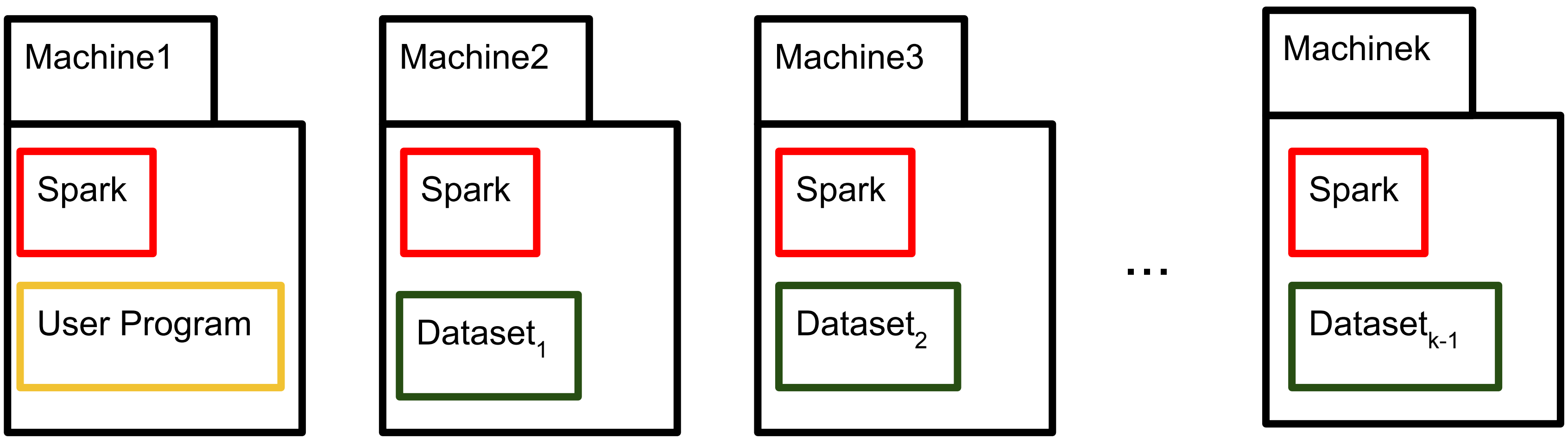
Setting the Context

3. We need a **user program**, stating the desired data analysis to be performed.



Setting the Context

4. We need a **dataset**, possibly splitted among the cluster:
   dataset = $\text{dataset}_1 + \text{dataset}_2 + \ldots + \text{dataset}_{k-1}$

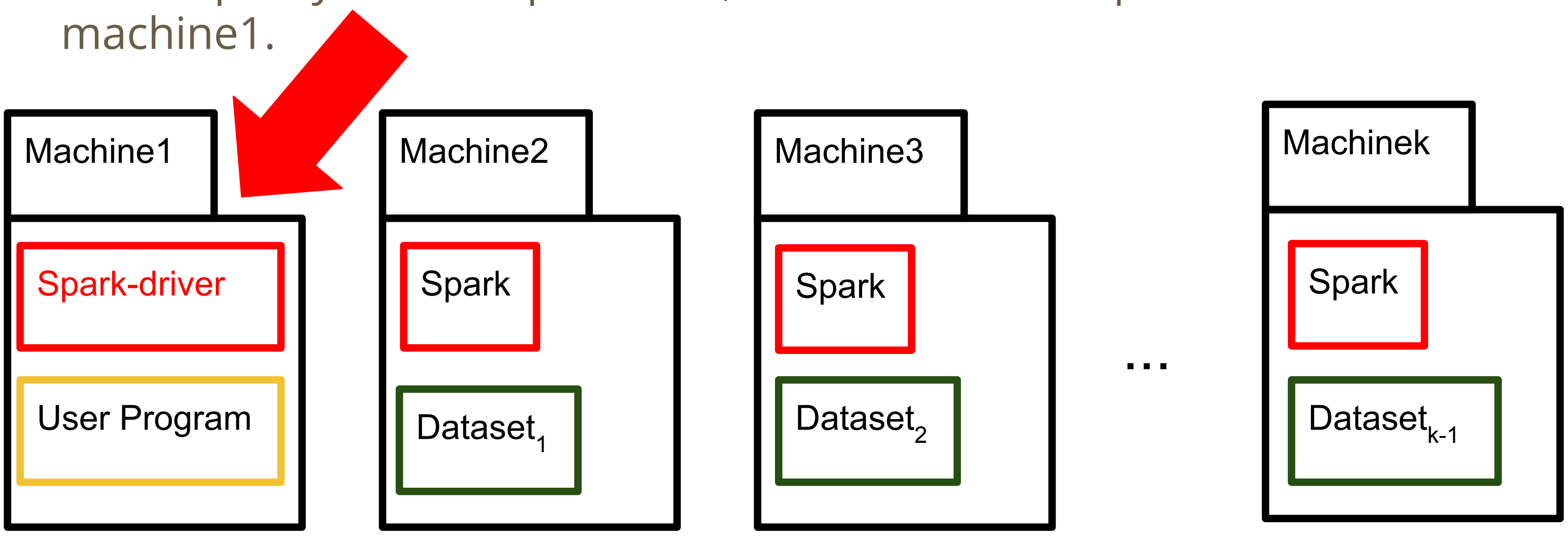


Setting the Context

5. The machine hosting the user program - more specifically, the CPU core of the machine running the user program by executing its main( ) method - has to be a **Spark driver process (master)**.

   This driver process is nothing but a Java process running on the JVM.

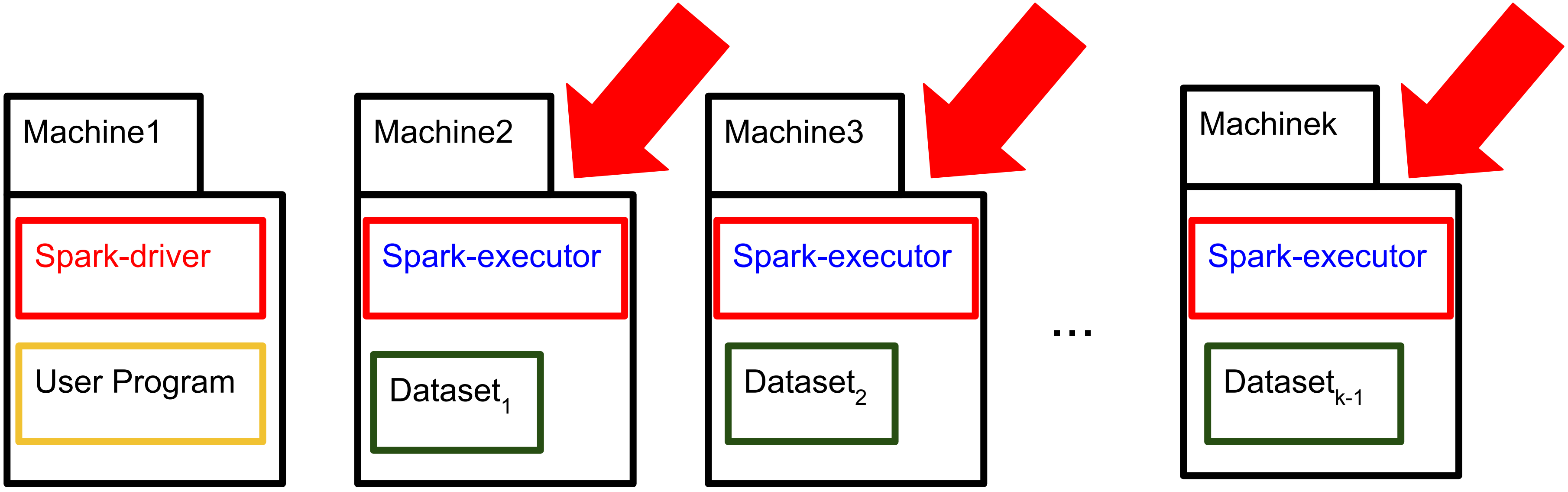
   For simplicity of the explanation, let's assume the Spark-driver runs in machine1.



Setting the Context

5. The remaining CPU cores of machine 1 and
all CPU cores of [machine2, ..., machinek]
are susceptible of running a **Spark executor process (slaves)**.

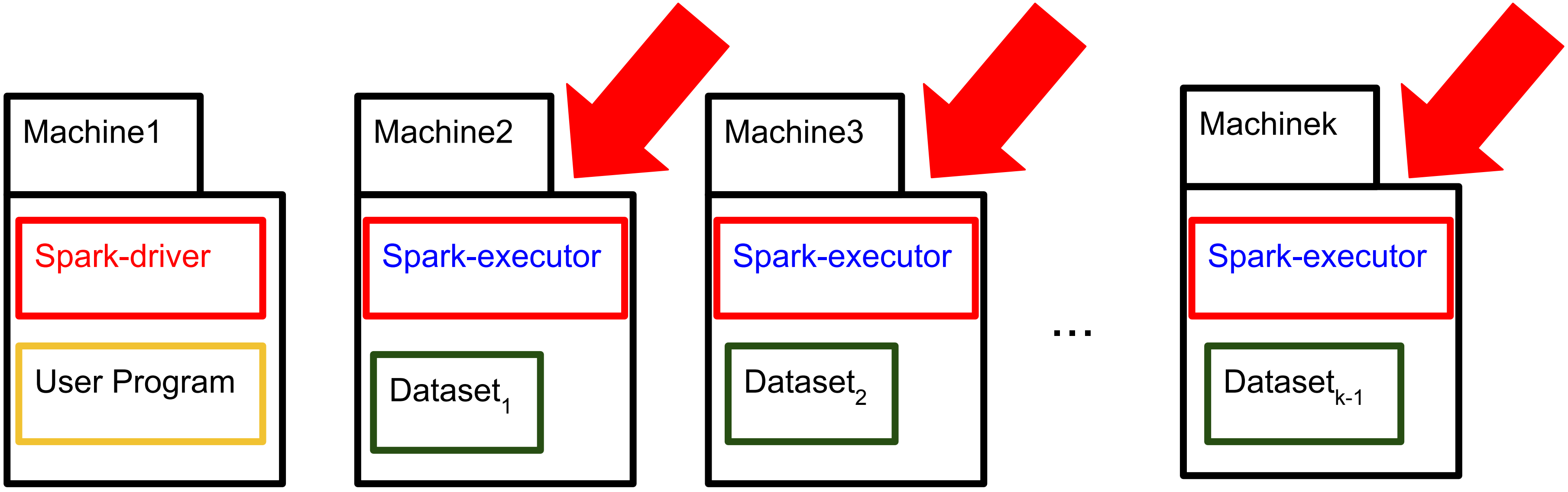
Again, each of these executor processes are nothing but a Java process running on the JVM.



Setting the Context

5. The remaining CPU cores of machine 1 and
all CPU cores of [machine2, …, machinek]
are susceptible of running a **Spark executor process (slaves)**.

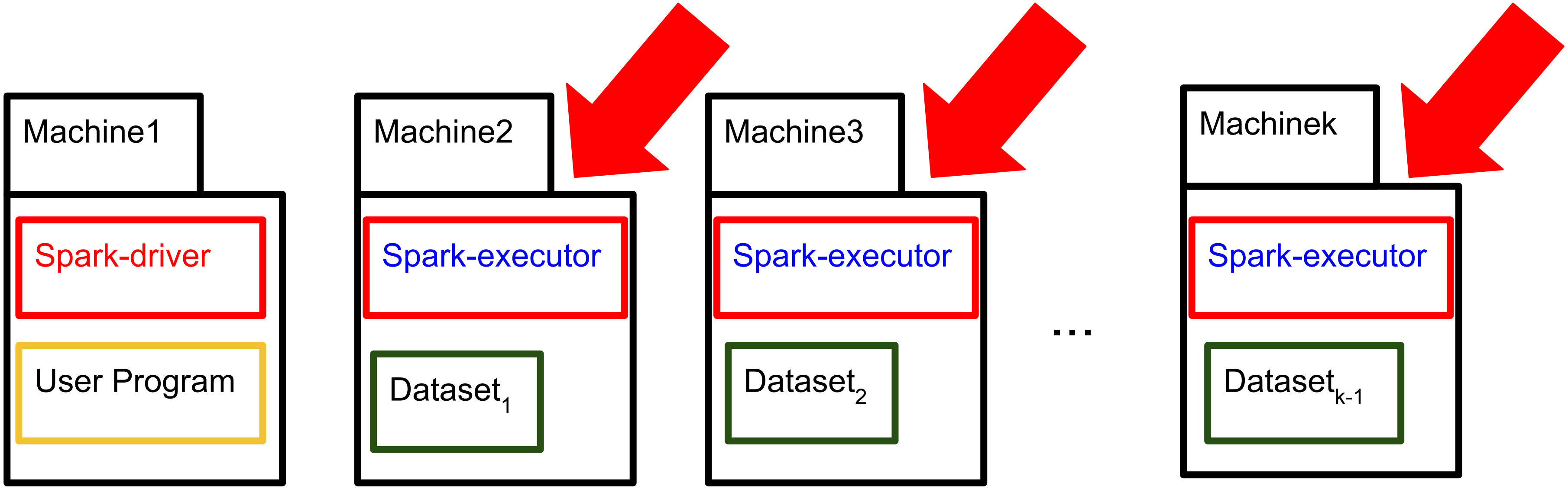
The Spark cluster manager handles starting and distributing the Spark executors across a distributed system according to the configuration parameters set by the Spark application.



Setting the Context

5. The remaining CPU cores of machine 1 and all CPU cores of [machine2, …, machinek] are susceptible of running a **Spark executor process (slaves)**.

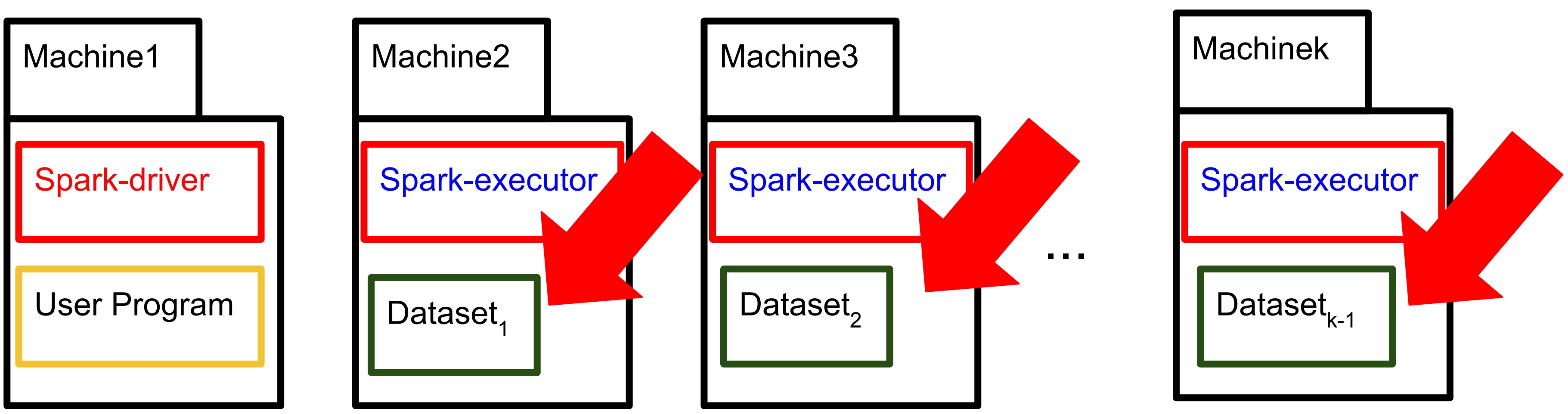
   The number of cores per executor can be configured at the user program, but typically they correspond to the physical cores on a machine, and an executor cannot span cores of different machines.



Setting the Context

5. The remaining CPU cores of machine 1 and
   all CPU cores of [machine2, …, machinek]
   are susceptible of running a **Spark executor process (slaves)**.

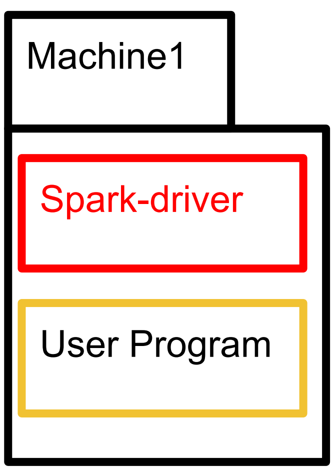
   The Spark execution engine itself distributes data across the executors for a computation.



Setting the Context

6. When we think this way we can see the **Spark driver** as the director of orchestra, co-ordinating the execution of the user program.

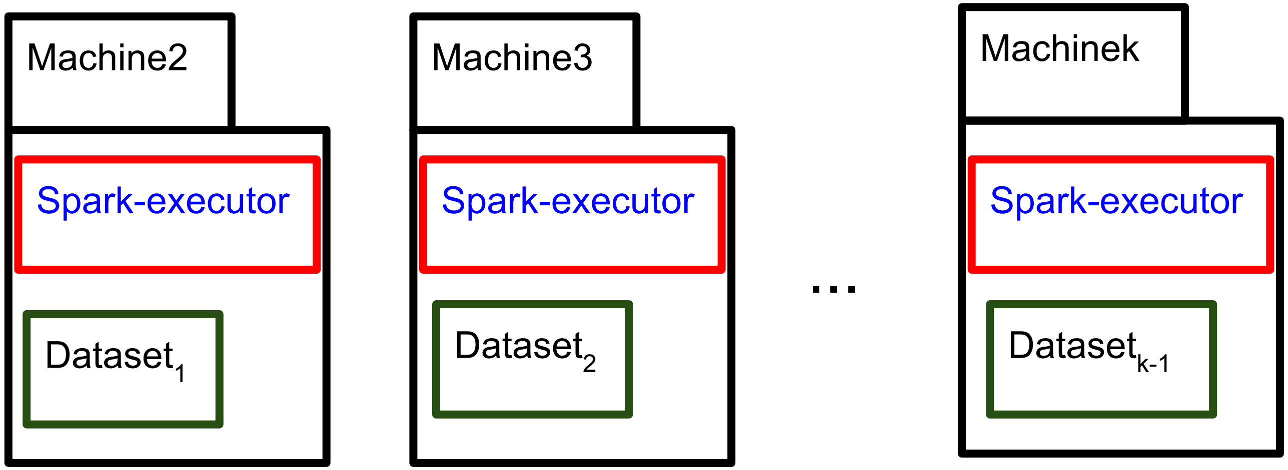
   This program will be all about the Resilient Distributed Datasets (RDDs) abstraction we will introduce next.



Setting the Context

6. Likewise, when we think this way we can see the **Spark executors** as the orchestra musicians.

   Their job will be basically to:
   - Use their CPU to compute RDDs.
   - Use their RAM and disk memory to store RDDs.



Setting the Context

Let's move on and discover
what is the story with these RDDs…

Setting the Context

But, before doing so…

Programming with RDDs will require us to understand basic concepts from Functional Programming!

So let’s start with these concepts first.

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
6. Lazy Evaluation.

Prerequisites: Functional Programming

Computer Science aim:
solve real-world problems.

Prerequisites: Functional Programming

Computer Science aim: solve real-world problems.

1. Represent information of the real-world → Data Structures.

Prerequisites: Functional Programming

Computer Science aim: solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

Prerequisites: Functional Programming

Computer Science aim: solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

Example:

Prerequisites: Functional Programming

Computer Science aim: Solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

- Programming: The art of implementing algorithms.

Prerequisites: Functional Programming

Computer Science aim: Solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

- Programming: The art of implementing algorithms.
  - By programming, you express to a computer...
    - The problem you want to solve (what)

Prerequisites: Functional Programming

Computer Science aim: Solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

- Programming: The art of implementing algorithms.
  - By programming, you express to a computer...
    - The problem you want to solve (what)
    and/or
    - The way you want it to solve your problem (how)

Prerequisites: Functional Programming

Computer Science aim: Solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

- A programming paradigm is nothing but a style of programming, a concrete way of communicating your problem to the computer.

  How do I communicate with a computer?

Prerequisites: Functional Programming

Computer Science aim: Solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

- A programming paradigm is nothing but a style of programming, a concrete way of communicating your problem to the computer.

  How do I communicate with a computer?

Prerequisites: Functional Programming

Computer Science aim: Solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

- A programming paradigm is nothing but a style of programming, a concrete way of communicating your problem to the computer.

  How do I communicate with a computer?

Prerequisites: Functional Programming

Computer Science aim: Solve real-world problems.

1. Represent information of the real-world → Data Structures.
2. Manipulate such this information to find a solution → Algorithm.

- A programming paradigm is nothing but a style of programming, a concrete way of communicating your problem to the computer.

  How do I communicate with a computer?

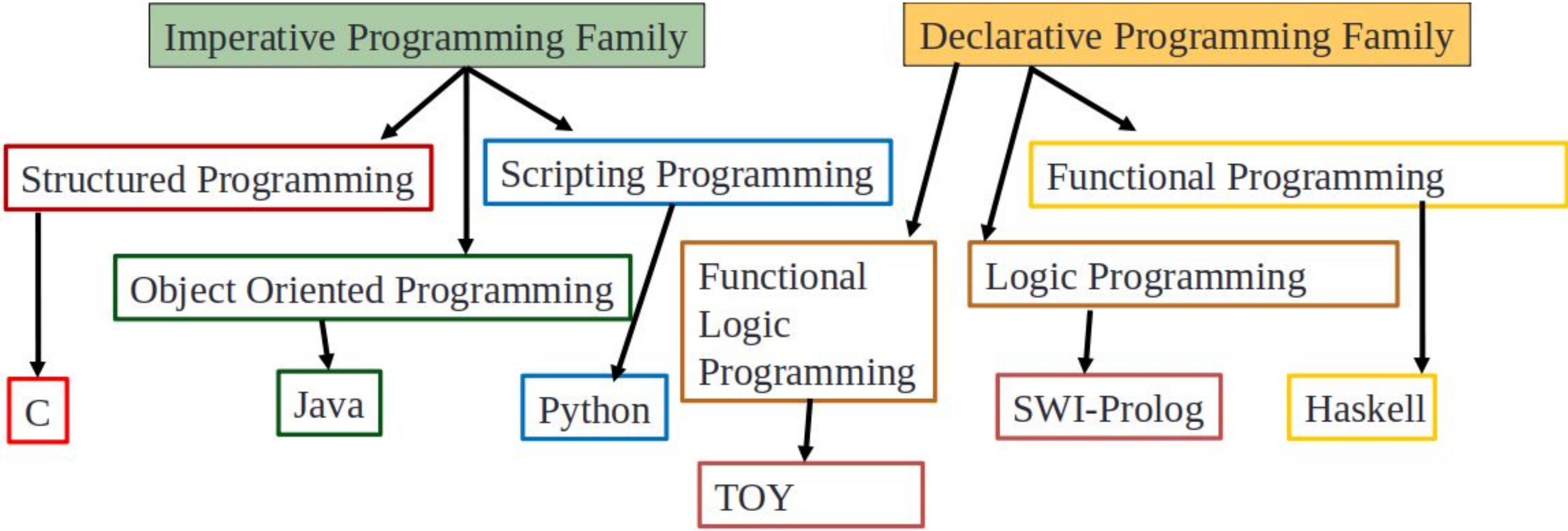
Prerequisites: Functional Programming

There are many different programming paradigms:

Prerequisites: Functional Programming

There are many different programming paradigms:

Just to name a few we have experienced with



Prerequisites: Functional Programming

- Imperative Programming: Focused on the *how?*

Prerequisites: Functional Programming

- Imperative Programming: Focused on the *how?*
  - Very associated to the computation itself.
    - How is the computation done?

Prerequisites: Functional Programming

- Imperative Programming: Focused on the *how?*
  - Very associated to the computation itself.
    - How is the computation done?
  - Notion of state.
    - How does the value of the variables evolve while the computation is performed?

Prerequisites: Functional Programming

- Declarative Programming: Focused on the *what?*

Prerequisites: Functional Programming

- Declarative Programming: Focused on the *what?*
  - What are the properties a solution to my problem fulfill?

Prerequisites: Functional Programming

This declarative nature produces an isolation between the:

- ➢ Declarative semantics: Meaning of program / user specification (*what*)
- ➢ Operational semantics: Interpretation / solving of the program (*how*)

Prerequisites: Functional Programming

This declarative nature produces an isolation between the:

- ➢ Declarative semantics: Meaning of program / user specification
- ➢ Operational semantics: Interpretation / solving of the program

The latter requires the presence of an oracle / planner that, given a program, decides how to solve it best.

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.

  - Declarative semantics considers a program as a theory based on a set of deterministic functions.
    - Functions are first class citizens:
      In the sense that functions can be used as any object in the language (i.e., results, input arguments and elements of data structures).

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.

  - Declarative semantics consider a program as a theory based on a set of deterministic functions.
    - Functions are first class citizens:
      In the sense that functions can be used as any object in the language (i.e., results, input arguments and elements of data structures).
  - Operational semantics solve a program by reducing its expressions via rewriting.
    - Rewriting is attempted via efficient and (under particular conditions) optimal evaluation strategies based on demand-driven evaluation.

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.
  - For using Spark Core we don't need to fully understand Functional Programming, just the following features:

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.

  - For using Spark Core we don't need to fully understand Functional Programming, just the following features:
    - Polymorphism.

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.

  - For using Spark Core we don’t need to fully understand Functional Programming, just the following features:
    - Polymorphism.
    - Higher-order functions.

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.
  - For using Spark Core we don't need to fully understand Functional Programming, just the following features:
    - Polymorphism.
    - Higher-order functions.
    - Partial application.

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.

  - For using Spark Core we don't need to fully understand Functional Programming, just the following features:
    - Polymorphism.
    - Higher-order functions.
    - Partial application.
    - Lazy evaluation.

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.
  - For using Spark Core we don't need to fully understand Functional Programming, just the following features:
    - Polymorphism.
    - Higher-order functions.
    - Partial application.
    - Lazy evaluation.
- For each of these features we will follow a 2 steps approach:

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.

  - For using Spark Core we don't need to fully understand Functional Programming, just the following features:
    - Polymorphism.
    - Higher-order functions.
    - Partial application.
    - Lazy evaluation.

- For each of these features we will follow a 2 steps approach:
  1. Explain the concept in the Functional programming language Haskell, using the interpreter Hugs 98.

Prerequisites: Functional Programming

- Functional Programming (FP):
  One of the most important declarative programming-based paradigms.

  - For using Spark Core we don't need to fully understand Functional Programming, just the following features:
    - Polymorphism.
    - Higher-order functions.
    - Partial application.
    - Lazy evaluation.

- For each of these features we will follow a 2 steps approach:
  1. Explain the concept in the Functional programming language Haskell, using the interpreter Hugs 98.
  2. Explain how the same concept can be simulated/applied in Python.

Prerequisites: Functional Programming

- Polymorphism is the ability of functions to support parametric input/output arguments.

- This way, a single implementation of the function can operate on different data types for the same input/output argument.

Prerequisites: Functional Programming

- The Haskell function myFst is polymorphic:
  - It receives two parametric arguments. One of type A (basically meaning “whatever datatype”) and a second of type B (meaning “whatever datatype and not necessarily the same one as before”). The function returns a value of type A.
    `myFst:: A -> B -> A`

Prerequisites: Functional Programming

- The Haskell function myFst is polymorphic:
  - It receives two parametric arguments. One of type A (basically meaning “whatever datatype”) and a second of type B (meaning “whatever datatype and not necessarily the same one as before”). The function returns a value of type A.
    ```
    myFst:: A -> B -> A
    ```
  - The implementation of the function is very simple: Given the two input parameters, just return the first one.
    ```
    myFst X Y = X
    ```

Prerequisites: Functional Programming

- The Haskell function myFst is polymorphic:
  - It receives two parametric arguments. One of type A (basically meaning “whatever datatype”) and a second of type B (meaning “whatever datatype and not necessarily the same one as before”). The function returns a value of type A.
    `myFst:: A -> B -> A`
  - The implementation of the function is very simple: Given the two input parameters, just return the first one.
    `myFst X Y = X`
  - In this context, the function can be applied with whatever datatypes:
    myFst 3 5 → 3
    myFst [true, false, true] 5 → [true, false, true]
    myFst 3 [true, false, true] → 3

Prerequisites: Functional Programming

- In Python, dynamic typing allows us to “simulate” polymorphism.

```
def fst(x, y):
    return x
```

  - In this context, we can apply the functions with whatever datatypes:
    fst(3, 5) → 3
    fst([True, False, True], 5) → [True, False, True]
    fst(3, [True, False, True]) → 3

Prerequisites: Functional Programming

- In Python, dynamic typing allows us to "simulate" polymorphism.

```
def fst(x, y):
    return x
```

  - Now, this doesn't mean dynamic typing == parametric polymorphism:

Prerequisites: Functional Programming

- In Python, dynamic typing allows us to “simulate” polymorphism.

```
def fst(x, y):
    return x
```

  - Now, this doesn’t mean dynamic typing == parametric polymorphism:
    - myFst [True, 5] 3 → Type system error.
      Whereas this goal is not supported by Hugs type system (Datatype A can be a list of booleans, or a list of integers, etc. But not a list of booleans_and_integers).

Prerequisites: Functional Programming

- In Python, dynamic typing allows us to “simulate” polymorphism.

```
def fst(x, y):
    return x
```

  - Now, this doesn’t mean dynamic typing == parametric polymorphism:
    - myFst [True, 5] 3 → Type system error.
      Whereas this goal is not supported by Hugs type system (Datatype A can be a list of booleans, or a list of integers, etc. But not a list of booleans_and_integers).
    - fst([True, 5], 3) → [True, 5]
      This is supported in Python, as dynamic typing implies no type-checking before runtime.

Prerequisites: Functional Programming

- A higher-order function is a function taking one or more functions as arguments, and/or returning a function as a result.

Prerequisites: Functional Programming

- The Haskell function map is a higher-order function:
  - map takes as input arguments:
    - A function with no name (for simplicity on referencing to it, let’s call it ‘F’) which: receives an input argument of type A and produces an output value of type B.
    - A list of items (all of them of type A).

Prerequisites: Functional Programming

- The Haskell function map is a higher-order function:
  - map takes as input arguments:
    - A function with no name (for simplicity on referencing to it, let's call it 'F') which: receives an input argument of type A and produces an output value of type B.
    - A list of items (all of them of type A).
  - map returns as a result a list of items (all of them of type B).

```
myMap:: (a -> b) -> [a] -> [b]
```

Prerequisites: Functional Programming

- The Haskell function map is a higher-order function:
  - The declarative definition of map consists on two cases:

Prerequisites: Functional Programming

- The Haskell function map is a higher-order function:
  - The declarative definition of map consists on two cases:
    1. Apply map over the empty list returns an empty list.
       `myMap f [ ] = [ ]`

Prerequisites: Functional Programming

- The Haskell function map is a higher-order function:
  - The declarative definition of map consists on two cases:
    1. Apply map over the empty list returns an empty list.
       `myMap f [ ] = [ ]`
    2. Apply map over a list containing x as its first element returns the result of applying f to x concatenated with the recursive application of map over the rest of the list xs.
       `myMap f (x:xs) = f x : myMap f xs`

Prerequisites: Functional Programming

- In Python, dynamic typing allows us to “simulate” higher-order functions, as a function can be accepted as an input parameter.

Prerequisites: Functional Programming

- In Python, dynamic typing allows us to “simulate” higher-order functions, as a function can be accepted as an input parameter.
  - In the case of my_map, the argument funct can be a function.

```
def my_map(funct, my_list):
    # 1. We create the output variable
    res = [ ]

    # 2. We populate the list with the higher application
    for item in my_list:
        sol = funct(item)
        res.append(sol)

    # 3. We return res
    return res
```

Prerequisites: Functional Programming

- Please note the difference between:
  - The declarative way of programming map in Haskell (associated to the *what*), where we declare both the type and properties such this function must satisfy).
  - The imperative way of programming my_map in Python (associated to the how), where we express how to do the computation for getting a solution.

```
myMap:: (a -> b) -> [a] -> [b]

myMap f [ ] = [ ]
myMap f (x : xs) = f x : myMap f xs
```

```
def my_map(funct, my_list):
        # 1. We create the output variable
                res = [ ]

        # 2. We populate the list
        for item in my_list:
                sol = funct(item)
                res.append(sol)

        # 3. We return res
        return res
```

Prerequisites: Functional Programming

- Partial application (or partial function application) refers to the process of fixing a number of arguments to a function, producing another function of smaller arity.

Prerequisites: Functional Programming

- The Hugs operator (+) stands for adding two numbers, and thus has arity 2.

Prerequisites: Functional Programming

- The Hugs operator (+) stands for adding two numbers, and thus has arity 2.
    - However, by having (+) defined, we can also use any of its multiple partial application variants (where the first argument of the function is hard-coded):
      (+) 3 5 → 8 Original Function (+)
      (+1) 4 → 5 Partial application with first argument hard-coded to 1
      (+8) 9 → 17 Partial application with first argument hard-coded to 8

Prerequisites: Functional Programming

- The Hugs operator (+) stands for adding two numbers, and thus has arity 2.
    - This also applies to the use of (+) into the higher-order functions presented before.
      myMap (+1) [1,2,3] → [2,3,4]
      myMap (>3) [1,8,2] → [False, True, False]
- Moreover, in the case of (+), as it has arity 2, all its partial applications must have arity 1. But, if a function has arity n, we can have partial applications of arities 1, 2, …, (n-1).

Prerequisites: Functional Programming

- In Python we can “simulate” partial application via lambda functions.
    - ➢ These functions are inline functions (defined on the fly) and can serve to act as middleman between the original function being defined and the partial (hard-coded) version we want to use.

Prerequisites: Functional Programming

- In Python we can "simulate" partial application via lambda functions.
  - ➢ These functions are inline functions (defined on the fly) and can serve to act as middleman between the original function being defined and the partial (hard-coded) version we want to use.

Given the function my_add with arity 2:

```
def my_add(x, y):
    return x + y
```

```
def my_bigger(x, y):
    return (x > y)
```

The following lines are the equivalent to the previous Hugs calls:

Hugs: map (+1) [1,2,3] → [2,3,4]
Python: res = my_map(lambda x : my_add(1, x), [1,2,3])

Hugs: map (>3) [1,8,2]
Python: res = my_map(lambda x : my_bigger(3, x), [1,8,2])

Prerequisites: Functional Programming

- Lazy evaluation (also known as call-by-need) is an evaluation strategy which delays the evaluation of an expression until its value is needed.
- It is often combined with sharing, which allows for the evaluation of an expression to be computed just once. The result of such evaluation is stored and reused later on for each further appearance of the expression.
- Although lazy evaluation is very appealing, it has an inherent overhead from using a “program oracle”, an external evaluator that takes in the program and drives the computation by deciding what is needed to be computed.

Prerequisites: Functional Programming

- The Haskell function loop recursively calls itself in a non-termination manner.

```
myLoop :: A
myLoop = myLoop
```

  Thus, any Hugs computation requiring to compute loop does not terminate.

Prerequisites: Functional Programming

- However, what about the computation

  myFst 3 myLoop ⟶ ?

Prerequisites: Functional Programming

- However, what about the computation

  myFst 3 myLoop ⟶ 3

  - It terminates with result 3.
  - The Haskell program is analysing by the external oracle.
    It realises that, in the goal “myFst 3 loop”, the second argument (myLoop) is not needed to be computed.
    Thus, it just computes the first argument 3 and return it straight away.

Prerequisites: Functional Programming

- However, what about the computation

    myFst 3 myLoop ⟶ 3

    - It terminates with result 3.
    - The Haskell program is analysing by the external oracle.
      It realises that, in the goal “myFst 3 loop”, the second argument (myLoop) is not needed to be computed.
      Thus, it just computes the first argument 3 and return it straight away.

        - *Note: In this case, 3 is already a value, so nothing to compute. But, if it had been a expression, then it would have been needed to compute it.*

Prerequisites: Functional Programming

- In contrast, the evaluation in Python of fst(3, loop( )) leads to non-termination.

Prerequisites: Functional Programming

- In contrast, the evaluation in Python of fst(3, loop( )) leads to non-termination.
    - This is because Python has a call-by-value or eager evaluation strategy, which means that all the input arguments of a function must be evaluated before proceed to evaluate the function itself.

Prerequisites: Functional Programming

- In contrast, the evaluation in Python of fst(3, loop( )) leads to non-termination.
    - This is because Python has a call-by-value or eager evaluation strategy, which means that all the input arguments of a function must be evaluated before proceed to evaluate the function itself.
    - In this case, Python is forced to evaluate both arguments of fst:
        - Whereas 3 is already a value, and nothing is required.
        - loop( ) is not a value, and thus Python tries to evaluate it before proceed to evaluate fst itself.
            - By trying to evaluate loop( ) Python enters an infinite recursion leading to non-termination.

Prerequisites: Functional Programming

- In contrast, the evaluation in Python of fst(3, loop( )) leads to non-termination.
- One might argue why to use eager or call-by-need evaluation, if it might lead to non-optimal computations (or even to non-termination, as in the example before).

Prerequisites: Functional Programming

- In contrast, the evaluation in Python of fst(3, loop( )) leads to non-termination.

- One might argue why to use eager or call-by-need evaluation, if it might lead to non-optimal computations (or even to non-termination, as in the example before).
  - Well, there is an overhead in the use of the external oracle (specifically, the time it takes to analyse the expression and come up with the optimal rewriting strategy).

Prerequisites: Functional Programming

- In contrast, the evaluation in Python of fst(3, loop( )) leads to non-termination.

- One might argue why to use eager or call-by-need evaluation, if it might lead to non-optimal computations (or even to non-termination, as in the example before).
  - Well, there is an overhead in the use of the external oracle (specifically, the time it takes to analyse the expression and come up with the optimal rewriting strategy).
  - In some cases such trade-off might make lazy evaluation faster, but in some many other cases it will not.

Prerequisites: Functional Programming

And now yes,
now that we understand the basic concepts of Functional Programming...

Prerequisites: Functional Programming

And now yes,
now that we understand the basic concepts of Functional Programming…

Let's move on and discover what is the story with these RDDs!

Outline

1. Introduction to Apache Spark.
2. Prerequisites: Functional Programming.
3. Setting the Context.
4. An RDD is An Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
6. Lazy Evaluation.

An RDD is an Abstract Data Type

Let's come back to the basics…

Solve a problem requires:

1. Translate the real-world concepts of the problem into information (data).
2. Algorithms to provide a solution to the problem by manipulating this data.

Regarding the first point, there are two roles with different perspectives:

- User: The user is interested in **What** can the data be used for?
- Developer: The developer of the data has more deeper concerns, as **How** is the data represented, stored and manipulated?

Data can be represented via:

(1) Constant and Variables.
(2) Data types.
(3) Data Structures.
(4) Abstract Data Types (ADTs).

An RDD is an Abstract Data Type

An Abstract Data Type (ADT) is perfect to make explicit these aforementioned 2 perspectives, as it separates:

- The specification of the data (**what** operations can be performed).
- The implementation of the data (**how** are these operations actually implemented).

An RDD is an Abstract Data Type

On doing this, an ADT provides an interface (a protocol of communication) between:

- The data user: Needs to make use of the data and its operations.
- The data developer: Needs to choose concrete data structures to represent the data and concrete algorithms to implement its operations.

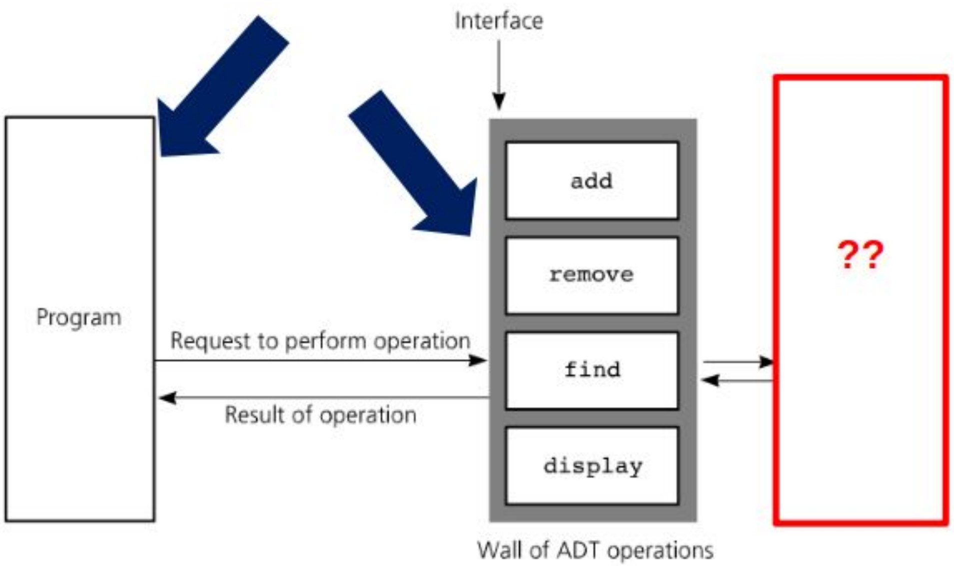
USER

DEVELOPER

An RDD is an Abstract Data Type

- The **ADT public side** puts on the feet of the data user.
  To do so, it has to sort out 2 main questions:

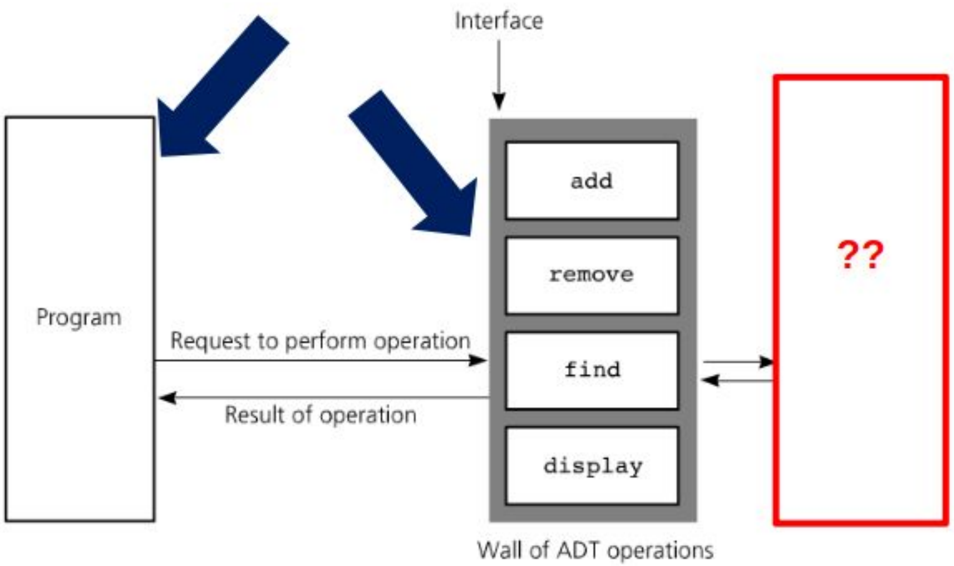
  1. **What** defines or specifies the type of data being offered?



An RDD is an Abstract Data Type

- The **ADT public side** puts on the feet of the data user.
  To do so, it has to sort out 2 main questions:

  1. **What** defines or specifies the type of data being offered?
  2. **What** are the operations that can be performed with this data? Specify each of them.

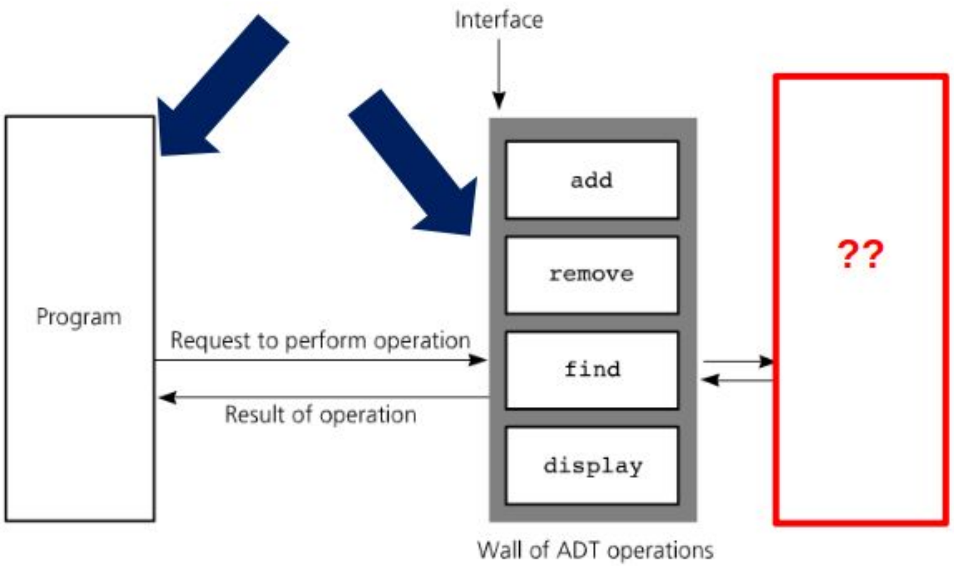


An RDD is an Abstract Data Type

- The **ADT public side** puts on the feet of the data user.
  To do so, it has to sort out 2 main questions:

  1. **What** defines or specifies the type of data being offered?
  2. **What** are the operations that can be performed with this data?
     Specify each of them.

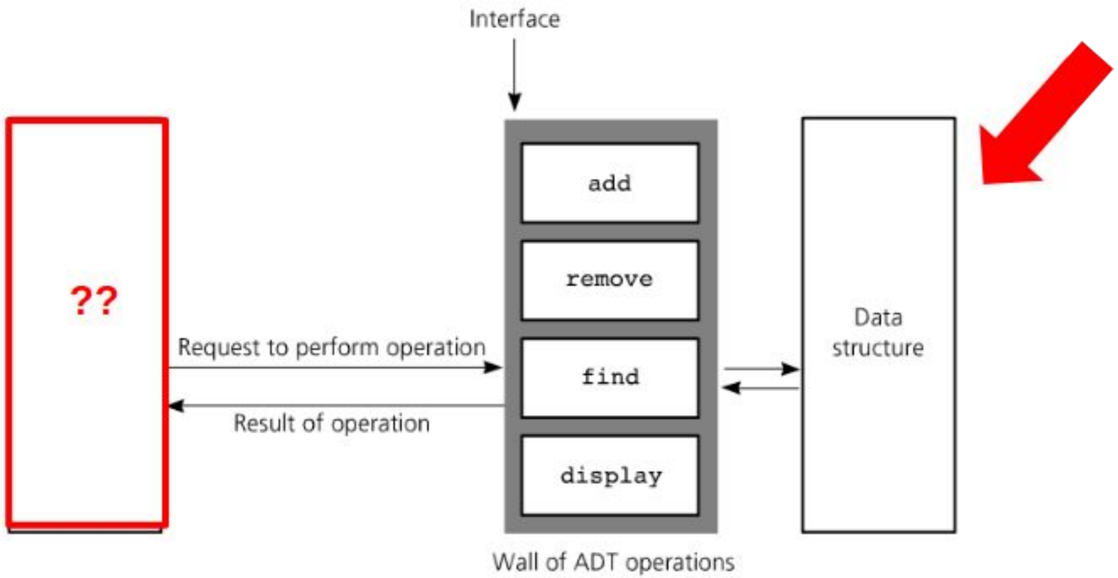
  - However, the public side does not need to worry about internal representation and implementation of the data.



An RDD is an Abstract Data Type

- The **ADT private side** puts on the feet of the data developer.
  To do so, it has to sort out another 2 main questions:

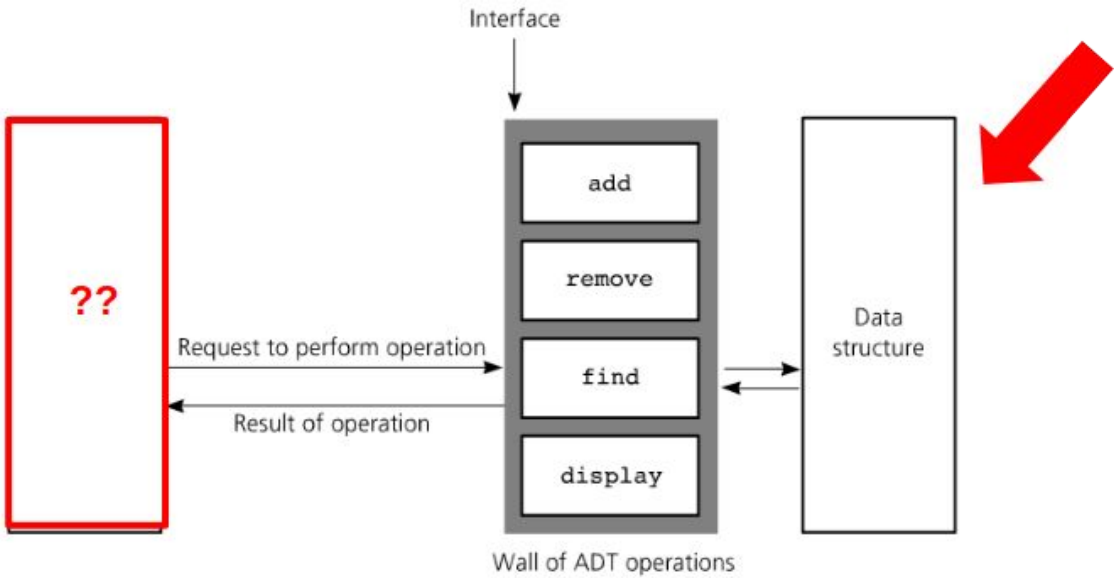
3. **How** is the data internally represented?
   Specify the concrete data structures used to layout the data.



An RDD is an Abstract Data Type

- The **ADT private side** puts on the feet of the data developer.
  To do so, it has to sort out another 2 main questions:

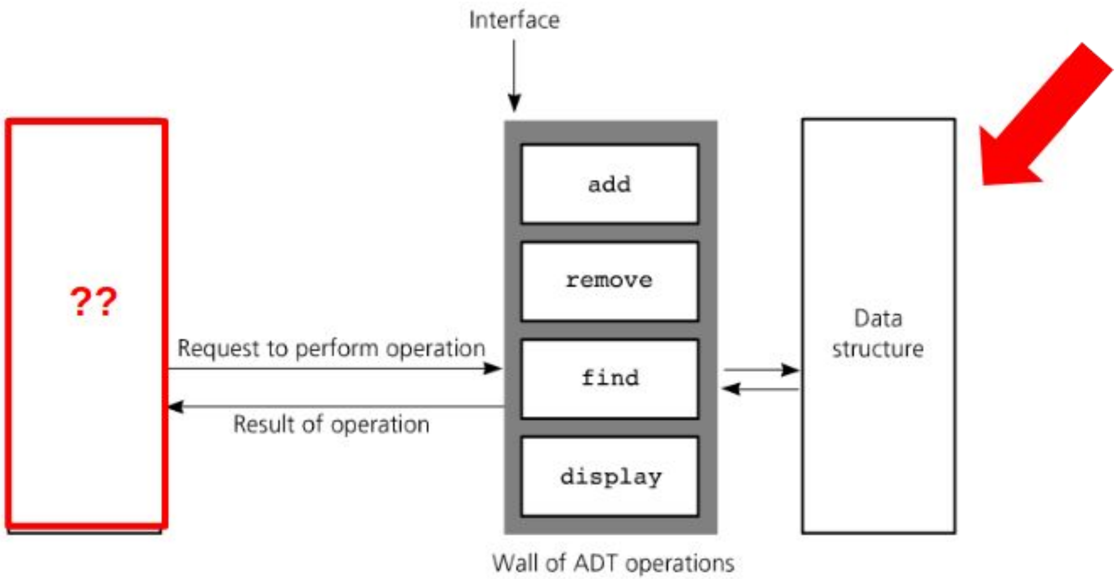
3. **How** is the data internally represented?
   Specify the concrete data structures used to layout the data.
4. **How** is each operation internally implemented?



An RDD is an Abstract Data Type

- The **ADT private side** puts on the feet of the data developer.
  To do so, it has to sort out another 2 main questions:

  3. **How** is the data internally represented?
     Specify the concrete data structures used to layout the data.
  4. **How** is each operation internally implemented?
  - However, the private side does not need to worry about the future user of the data.



An RDD is an Abstract Data Type

More in general…

- A datatype can be:
  - Mutable: It supports operations that modify its status (e.g., int, boolean, double, int[ ], etc).
  - Inmutable: Its status cannot be modified (e.g., String).

An RDD is an Abstract Data Type

More in general…

- Each operation can be classified as:
  1. Creator: Create a new object/entity of this ADT.
  2. Mutator: Given an existing object/entity of this ADT, modifies its status in some way.
  3. Observer: Given an existing object/entity of this ADT, returns some property/info of its internal state (but does not modify the object/entity at all).

An RDD is an Abstract Data Type

More in general…

- Each operation can be classified as:
  1. Creator: Create a new object/entity of this ADT.
  2. Mutator: Given an existing object/entity of this ADT, modifies its status in some way.
  3. Observer: Given an existing object/entity of this ADT, returns some property/info of its internal state (but does not modify the object/entity at all).

- Also, each of these operations can be either:
  - Total: If it always success.
  - Partial: If it can either success or return an error.

An RDD is an Abstract Data Type

The main data abstraction of Spark Core,
the so known Resilient Distributed Dataset (RDD)
is nothing but an ADT!

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
6. Lazy Evaluation.

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
   a. Data Definition.
   b. Creation Operations.
   c. Mutator/Transformation Operations.
   d. Observer/Action Operations.
   e. All together: A Spark User Program.
6. Lazy Evaluation.

Data Definition

Let’s go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...

...collection of elements.

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)

...collection of elements.

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statically-typed** (available for any data type T you want, as long as it sticks to T for all its elements)

...collection of elements.

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statically-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)

...collection of elements.

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statically-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

...collection of elements.

Data Definition

Let's go with the ADT public side:

1. <u>**What** defines or specifies the type of data being offered?</u>

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statycally-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

<u>...collection of elements</u>.

You can think of it as:
- A kind of list, in the sense that elements can be repeated.
- A kind of set, in the sense that elements have no particular default order.

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statycally-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

...collection of elements.

For example, we can have an RDD of integers:

firstRDD → [1, 5, 2, 3, 1, 7, 2, 9]

see that some elements might be repeated

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statycally-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

...collection of elements.

For example, we can have an RDD of integers:

firstRDD → [5, 3, 7, 9, 1, 2, 1, 2]

see that the order of the elements does not matter; this is the same RDD as before

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statycally-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

...collection of elements.

We can have as well an RDD of tuples (String, Boolean):

secondRDD → [("Hello", True), ("Goodbye, False), ("Hello", True), ("Hi", False)]

see that some elements might be repeated

Data Definition

Let's go with the ADT public side:

1. <u>**What** defines or specifies the type of data being offered?</u>

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statycally-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

<u>**...collection of elements**</u>.

<u>We can have as well an RDD of tuples (String, Boolean):</u>

secondRDD → [("Goodbye, False), ("Hi", False), ("Hello", True), ("Hello", True)]

see that the order of the elements does not matter; this is the same RDD as before

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statycally-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

...collection of elements.

We can have as well an RDD of maps with a String key and an integer value:

thirdRDD → [{ "Hi" -> 3, "Bonjour" -> 5 } , { }, { "Hola" -> 4, "Danke" -> 6 }]

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic**, but **statycally-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

...collection of elements.

We can have an RDD of tuples (String, Boolean) being empty:

fourthRDD → [ ]

Data Definition

Let's go with the ADT public side:

1. **What** defines or specifies the type of data being offered?

- An RDD defines or specifies an...
  1. **indivisible** (logically presented as an atomic variable)
  2. **generic,** but **statycally-typed** (available for any data type T you want, as long as it sticks to T for all its element)
  3. **lazily-evaluated** (only computed if required, and as much as required)
  4. **non-mutable** (cannot change type nor value)

...collection of elements.

But we cannot have an RDD with elements of different types:

fifthRDD → [1, 5, True, 4, False] ← This is not correct and is not supported!

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
    a. Data Definition.
    b. Creator Operations.
    c. Mutator/Transformation Operations.
    d. Observer/Action Operations.
    e. All together: A Spark User Program.
6. Lazy Evaluation.

Creator Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

Creator Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

Creator Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  1. **Creator**: They create a new RDD from an existing collection or dataset.

Creator Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u> Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  1. **Creator**: They create a new RDD from an existing collection or dataset.

For example, the creator operation "parallelize" takes a List and creates an RDD:

```
myRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
```

myRDD → [1, 2, 3, 4, 5] or myRDD → [5, 1, 3, 2, 4]

Creator Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u> Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  1. **Creator**: They create a new RDD from an existing collection or dataset.

For example, the creator operation "parallelize" takes a List and creates an RDD:

```
myRDD  =  sc.parallelize( [ 1, 2, 3, 4, 5 ] )
```

myRDD → [1, 2, 3, 4, 5]   or   myRDD → [5, 1, 3, 2, 4]

By the moment we don't know what sc is; no worries, we will explain it later.

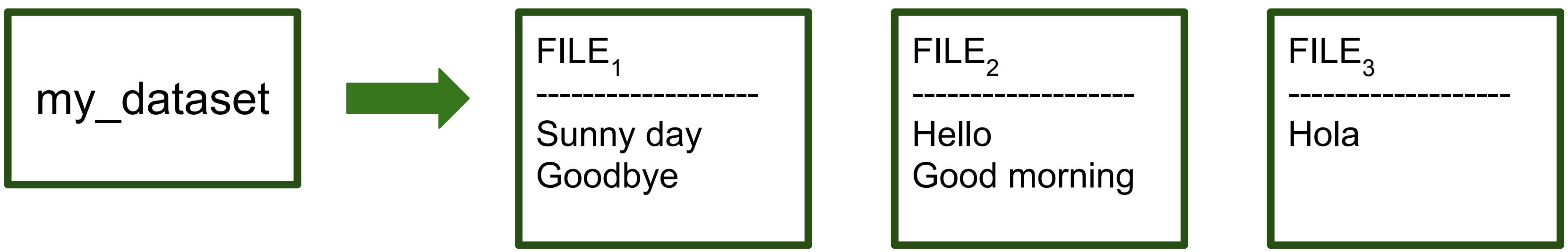
Creator Operations

Let’s go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u>
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  1. **Creator**: They create a new RDD from an existing collection or dataset.

The creator operation “textFile” takes a Dataset of text files and creates an RDD:



Creator Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  1. **Creator**: They create a new RDD from an existing collection or dataset.

The creator operation "textFile" takes a Dataset of text files and creates an RDD:

```
myRDD  = sc.textFile( my_dataset )
```

myRDD → ["Sunny day\n", "Goodbye\n", "Hello\n", "Good morning\n", "Hola\n"]

or

myRDD → ["Goodbye\n", "Hola\n", "Hello\n", "Sunny day\n", "Good morning\n"]

Creator Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u> Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  1. **Creator**: They create a new RDD from an existing collection or dataset.

The creator operation "textFile" takes a Dataset of text files and creates an RDD:

```
myRDD  = sc.textFile( my_dataset )
```

myRDD → ["Sunny day\n", "Goodbye\n", "Hello\n", "Good morning\n", "Hola\n"]

or

myRDD → ["Goodbye\n", "Hola\n", "Hello\n", "Sunny day\n", "Good morning\n"]

As we can see it is an RDD of Strings, with one element per line of file.

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
    a. Data Definition.
    b. Creator Operations.
    c. Mutator/Transformation Operations.
    d. Observer/Action Operations.
    e. All together: A Spark User Program.
6. Lazy Evaluation.

Mutator/Transformation Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  1. **Creator**: They create a new RDD from an existing collection or dataset.
  2. **Mutator**: These operations are called **Transformations**. They take one or more RDDs and produce a new RDD.

Mutator/Transformation Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  2. **Mutator**: These operations are called **Transformations**.
     They take one or more RDDs and produce a new RDD.

A Transformation is a "coarse-grained" operation (a function that is applied, element by element, to the entire collection of elements of an RDD).

Mutator/Transformation Operations

Let’s go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  2. **Mutator**: These operations are called **Transformations**. They take one or more RDDs and produce a new RDD.

A Transformation is a “coarse-grained” operation (a function that is applied, element by element, to the entire collection of elements of an RDD).

➢ Since RDDs are statically typed and immutable, calling a transformation on one RDD will not modify it, but instead will produce a new RDD.

Mutator/Transformation Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u> Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  2. **Mutator**: These operations are called **Transformations**. They take one or more RDDs and produce a new RDD.

A Transformation is a "coarse-grained" operation (a function that is applied, element by element, to the entire collection of elements of an RDD).

- ➢ Since RDDs are statically typed and immutable, calling a transformation on one RDD will not modify it, but instead produce a new one.
- ➢ The key ingredient of the role of transformations is that they are lazily evaluated (i.e., they are only computed once it is absolutely needed). But we will see this in detail later on.

Mutator/Transformation Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  2. **Mutator**: These operations are called **Transformations**.
     They take one or more RDDs and produce a new RDD.

The mutator (transformation) operation "map" takes an RDD and applies a function F to each of its elements:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem + 1 )
```

inputRDD → [1, 2, 3, 4, 5] or inputRDD → [5, 1, 3, 2, 4]
newRDD → [2, 3, 4, 5, 6] or newRDD → [6, 2, 4, 3, 5]

Mutator/Transformation Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  2. **Mutator**: These operations are called **Transformations**. They take one or more RDDs and produce a new RDD.

The mutator (transformation) operation "map" takes an RDD and applies a function F to each of its elements:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem > 3 )
```

inputRDD → [1, 2, 3, 4, 5]
newRDD → [False, False, False, True, True]

Mutator/Transformation Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u> Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  2. **Mutator**: These operations are called **Transformations**. They take one or more RDDs and produce a new RDD.

The mutator (transformation) operation "map" takes an RDD and applies a function F to each of its elements:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem > 3 )
```

inputRDD → [5, 1, 3, 2, 4]
newRDD → [True, False, False, False, True]

Mutator/Transformation Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  2. **Mutator**: These operations are called **Transformations**. They take one or more RDDs and produce a new RDD.

The mutator (transformation) operation "filter" takes an RDD and applies a function/property checking F to filter the elements holding it:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.filter( lambda elem : elem > 3 )
```

inputRDD → [1, 2, 3, 4, 5] or inputRDD → [5, 1, 3, 2, 4]
newRDD → [4, 5] or newRDD → [5, 4]

Mutator/Transformation Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u>
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  2. **Mutator**: These operations are called **Transformations**.
     They take one or more RDDs and produce a new RDD.

The mutator (transformation) operation "join" makes an inner join of RDD1-RDD2:

```
input1RDD = sc.parallelize( [ ("A", 1), ("A", 2),
                              ("B", 1), ("C", 1) ] )
input2RDD = sc.parallelize( [ ("A", True), ("B", True),
                              ("B", False), ("D", True) ] )
newRDD = input1RDD.join(input2RDD)
```

newRDD → [("A", (1, True)), ("A", (2, True)), ("B", (1, True)), ("B", (1, False))]

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
    a. Data Definition.
    b. Creator Operations.
    c. Mutator/Transformation Operations.
    d. Observer/Action Operations.
    e. All together: A Spark User Program.
6. Lazy Evaluation.

Observer/Action Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
  1. **Creator**: They create a new RDD from an existing collection or dataset.
  2. **Mutator**: These operations are called **Transformations**.
     They take one or more RDDs and produce a new RDD.
  3. **Observer**: These operations are called **Actions**.
     They return some property/info from an RDD without modifying it.

Observer/Action Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**. They return some property/info from an RDD without modifying it.

An Action is what produces an output for a Spark program, either by printing some result by screen or saving it to stable storage.

Observer/Action Operations

In this sense, we pictured **creation** and **transformation** operations as the ones producing RDDs respectively from scratch or from other RDDs...

Observer/Action Operations

In this sense, we pictured **creation** and **transformation** operations as the ones producing RDDs respectively from scratch or from other RDDs...

Hi, I'm parallelize, I'm a **creator** operation,
and my work is to produce inputRDD from a list

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
```

Observer/Action Operations

In this sense, we pictured **creation** and **transformation** operations as the ones producing RDDs respectively from scratch or from other RDDs...

Hi there, I'm map, I'm a **transformation** operation,
and my work is to produce mappedRDD from inputRDD

```
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
```

Observer/Action Operations

In this sense, we pictured **creation** and **transformation** operations as the ones producing RDDs respectively from scratch or from other RDDs...

Hello, I'm filter, I'm a **transformation** operation,
and my work is to produce filteredRDD from mappedRDD

```
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
```

Observer/Action Operations

In this sense, we pictured **creation** and **transformation** operations as the ones producing RDDs respectively from scratch or from other RDDs...

...but, as we said, **creation** and **transformation** operations are lazily evaluated. That is, they are only computed once required, and just as much as required.

Observer/Action Operations

In this sense, we pictured **creation** and **transformation** operations as the ones producing RDDs respectively from scratch or from other RDDs...

...but, as we said, **creation** and **transformation** operations are lazily evaluated. That is, they are only computed once required, and just as much as required.

So, after this lovely piece of code:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
```

What can we expect our **creation** and **transformation** operations to be doing?

Observer/Action Operations

Observer/Action Operations

But, hey, no worries, that's why we need an **Action**!

Hi, I'm an **action** and I'm going to wake up
these **creation** and **transformation** operations.
We need to start working, right now, so as to compute an actual result!

Observer/Action Operations

Observer/Action Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u>
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:
3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

The observer (action) operation "count" computes the number of elements of an RDD:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
resVAL = filterRDD.count()
```

inputRDD → [5, 1, 3, 2, 4]
filterRDD → [5, 3, 4]
resVAL → 3

Observer/Action Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

The observer (action) operation "take" gets a subset of the elements from an RDD:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
resVAL = filterRDD.take(2)
```

inputRDD → [5, 1, 3, 2, 4]
filterRDD → [5, 3, 4]
resVAL → [5, 3] ← This is an actual list that we can manipulate in Python

Observer/Action Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

The observer (action) operation "take" gets a subset of the elements from an RDD:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
resVAL = filterRDD.take(2)
```

inputRDD → [5, 1, 3, 2, 4]
filterRDD → [5, 3, 4] Or, for the same code as before, we could have get...
resVAL → [3, 4] ← This is an actual list that we can manipulate in Python

Observer/Action Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u>
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

The observer (action) operation "take" gets a subset of the elements from an RDD:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
resVAL = filterRDD.take(2)
```

inputRDD → [5, 1, 3, 2, 4]
filterRDD → [5, 3, 4] Or any other combination...
resVAL → [4, 5] ← This is an actual list that we can manipulate in Python

Observer/Action Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

The observer (action) operation "collect" gets a full set of the elements from an RDD:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
resVAL = filterRDD.collect()
```

inputRDD → [5, 1, 3, 2, 4]
filterRDD → [5, 3, 4]
resVAL → [5, 3, 4] ← This is an actual list that we can manipulate in Python

Observer/Action Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

The observer (action) operation "collect" gets a full set of the elements from an RDD:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
resVAL = filterRDD.collect()
```

inputRDD → [5, 1, 3, 2, 4]
filterRDD → [5, 3, 4]   But, being the same numbers, we could have get...
resVAL → [3, 4, 5]   ← This is an actual list that we can manipulate in Python

Observer/Action Operations

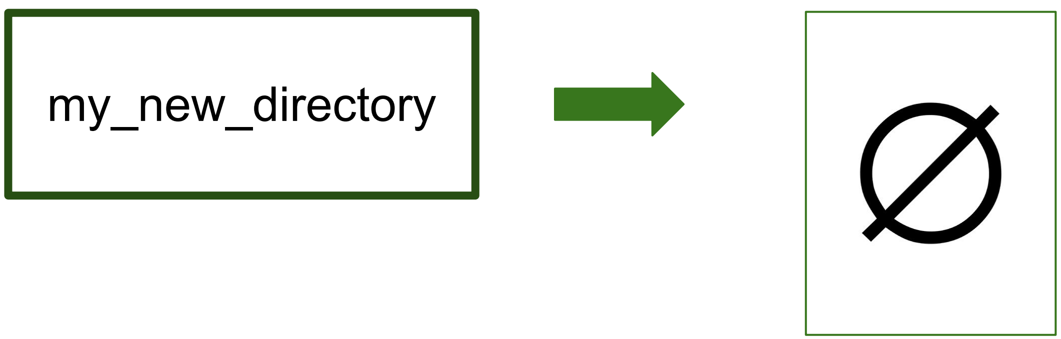
Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data?
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

The observer (action) operation "saveAsTextFile" stores the full set of the elements from an RDD into stable storage:



As we can see, the directory has to be initially empty

Observer/Action Operations

Let's go with the ADT public side:

2. **What** are the operations that can be performed with this data? Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**. They return some property/info from an RDD without modifying it.

The observer (action) operation "saveAsTextFile" stores the full set of the elements from an RDD into stable storage:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7, 8 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
filterRDD.saveAsTextFile(my_new_directory)
```

Observer/Action Operations

Let's go with the ADT public side:

2. <u>**What** are the operations that can be performed with this data?</u>
   Specify each of them.

- An RDD offers an extense API, with plenty of **operations**:

3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

The observer (action) operation "saveAsTextFile" stores the full set of the elements from an RDD into stable storage:

my_dataset →

$FILE_1$	$FILE_2$	$FILE_3$
5	7	6
3		8
4		

filterRDD → [5, 3, 4, 7, 6, 8]

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
    a. Data Definition.
    b. Creator Operations.
    c. Mutator/Transformation Operations.
    d. Observer/Action Operations.
    e. All together: A Spark User Program.
6. Lazy Evaluation.

All Together: A Spark User Program

And this is it!

We have seen, at a very high level,

the API of operations offered by RDDs.

All Together: A Spark User Program

And this is it!
We have seen, at a very high level,
the API of operations offered by RDDs.

1. **Creator**: They create a new RDD from an existing collection or dataset.
2. **Mutator**: These operations are called **Transformations**.
   They take one or more RDDs and produce a new RDD.
3. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

All Together: A Spark User Program

- Actually, there is a last type of operations: persistent ones.

1. **Creator**: They create a new RDD from an existing collection or dataset.
2. **Mutator**: These operations are called **Transformations**.
   They take one or more RDDs and produce a new RDD.
3. **Persistent**: They keep an RDD permanently stored until the Spark program finishes.
4. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

All Together: A Spark User Program

- Actually, there is a last type of operations: persistent ones.

1. **Creator**: They create a new RDD from an existing collection or dataset.
2. **Mutator**: These operations are called **Transformations**.
   They take one or more RDDs and produce a new RDD.
3. **Persistent**: They keep an RDD permanently stored until the Spark program finishes.
4. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

Persistent operations will be much better understood later on.
But, by the moment, let's just follow this rule of thumb:

➢ If an RDD is used more than once, then it has to be persisted as soon as it is declared.

All Together: A Spark User Program

- Actually, there is a last type of operations: persistent ones.

1. **Creator**: They create a new RDD from an existing collection or dataset.
2. **Mutator**: These operations are called **Transformations**.
   They take one or more RDDs and produce a new RDD.
3. **Persistent**: They keep an RDD permanently stored until the Spark program finishes.
4. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7, 8 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
resVAL1 = filterRDD.count()    ← Here filterRDD is used for first time.
resVAL2 = filterRDD.take(3)    ← Here filterRDD is used again.
```

Here we have an example of a bad practice, where filterRDD is used more than once, but it is not persisted after having been declared.

All Together: A Spark User Program

- Actually, there is a last type of operations: persistent ones.

  1. **Creator**: They create a new RDD from an existing collection or dataset.
  2. **Mutator**: These operations are called **Transformations**.
     They take one or more RDDs and produce a new RDD.
  3. **Persistent**: They keep an RDD permanently stored until the Spark program finishes.
  4. **Observer**: These operations are called **Actions**.
     They return some property/info from an RDD without modifying it.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7, 8 ] )
filterRDD = inputRDD.filter( lambda elem : elem >= 3 )
filterRDD.persist()
resVAL1 = filterRDD.count()
resVAL2 = filterRDD.take(3)
```

We fix the issue by persisting it as soon as it is first declared.

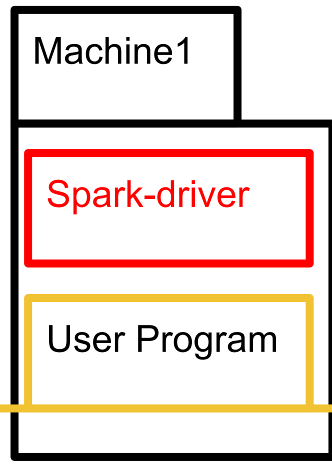
All Together: A Spark User Program

Now yes, this is it!
We have seen, at a very high level,
the API of operations offered by RDDs.

1. **Creator**: They create a new RDD from an existing collection or dataset.
2. **Mutator**: These operations are called **Transformations**.
   They take one or more RDDs and produce a new RDD.
3. **Persistent**: They keep an RDD permanently stored until the Spark program finishes.
4. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

All Together: A Spark User Program

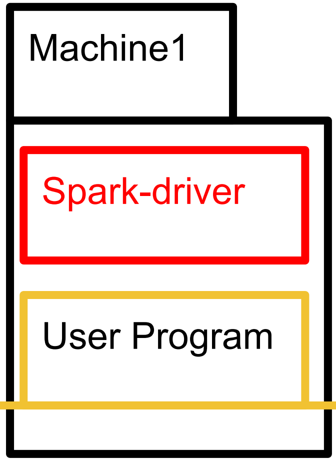
- Indeed, the prefix number 1, 2, 3 and 4 in these operations is not casual: They determine the structure of a Spark typical user program!



1. **Creator**: They create a new RDD from an existing collection or dataset.
2. **Mutator**: These operations are called **Transformations**. They take one or more RDDs and produce a new RDD.
3. **Persistent**: They keep an RDD permanently stored until the Spark program finishes.
4. **Observer**: These operations are called **Actions**. They return some property/info from an RDD without modifying it.

All Together: A Spark User Program

- Indeed, the prefix number 1, 2, 3 and 4 in these operations is not casual: They determine the structure of a Spark user program!



Typical Spark User Program:

1. Create some input RDDs from external data.
2. Transform them to define new RDDs using transformations.
3. Persist any intermediate RDDs that will need to be reused.
4. Launch actions to kick off a distributed computation.

All Together: A Spark User Program

- Indeed, the prefix number 1, 2, 3 and 4 in these operations is not casual: They determine the structure of a Spark user program!

1. Creation:

```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5] )
```

2. Transformations:

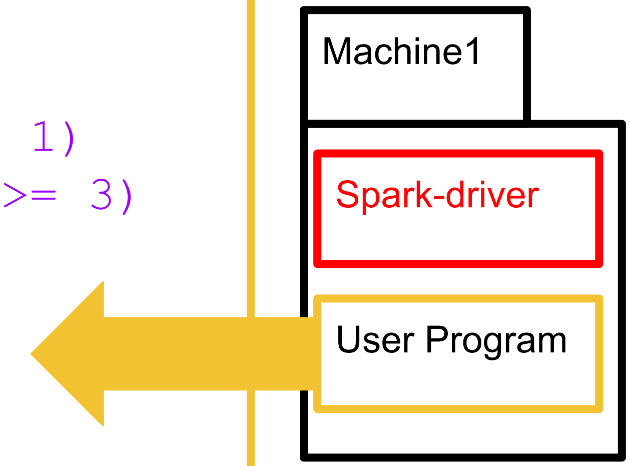
```
mappedRDD = inputRDD.map(lambda x: x + 1)
solRDD = mappedRDD.filter(lambda x: x >= 3)
```

3. Persistence:

```
solRDD.persist( )
```

4. Actions:

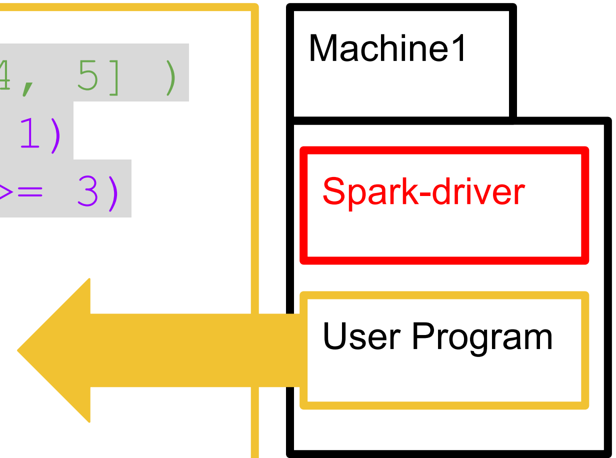
```
resVAL = filterRDD.count( )
solRDD.saveAsTextFile()
print(resVAL)
```



All Together: A Spark User Program

- Indeed, the prefix number 1, 2, 3 and 4 in these operations is not casual: They determine the structure of a Spark user program!

```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5] )
mappedRDD = inputRDD.map(lambda x: x + 1)
solRDD = mappedRDD.filter(lambda x: x >= 3)
solRDD.persist( )
resVAL = filterRDD.count( )
solRDD.saveAsTextFile()
print(resVAL)
```



Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
6. Lazy Evaluation.

Lazy Evaluation

Now that we understand how
a Spark user program looks like,
let’s put our focus on how lazy evaluation
can make its computation more efficient.

Lazy Evaluation

To do so, let's compare the evaluation of the following 2 programs:

- User Program 1 benefits from lazy evaluation.
  That is, thanks to Spark lazy evaluation-based model, this program is executed using less resources as if Spark had been eager evaluation-based.

User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
    print(item)
```

Lazy Evaluation

To do so, let's compare the evaluation of the following 2 programs:

- User Program 2 do not benefit from lazy evaluation.
  That is, it doesn't matter whether Spark had been lazy evaluation-based or eager evaluation-based. In both cases, the program would have been executed using the same resources.

User Program 2

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_directory)
```

Lazy Evaluation

Let's focus on User Program 1…

<u>What are the main variables (represented as cartoon characters) in this program?</u>

1. Dataset

2. inputRDD

3. solRDD

4. resVAL

User Program 1

1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

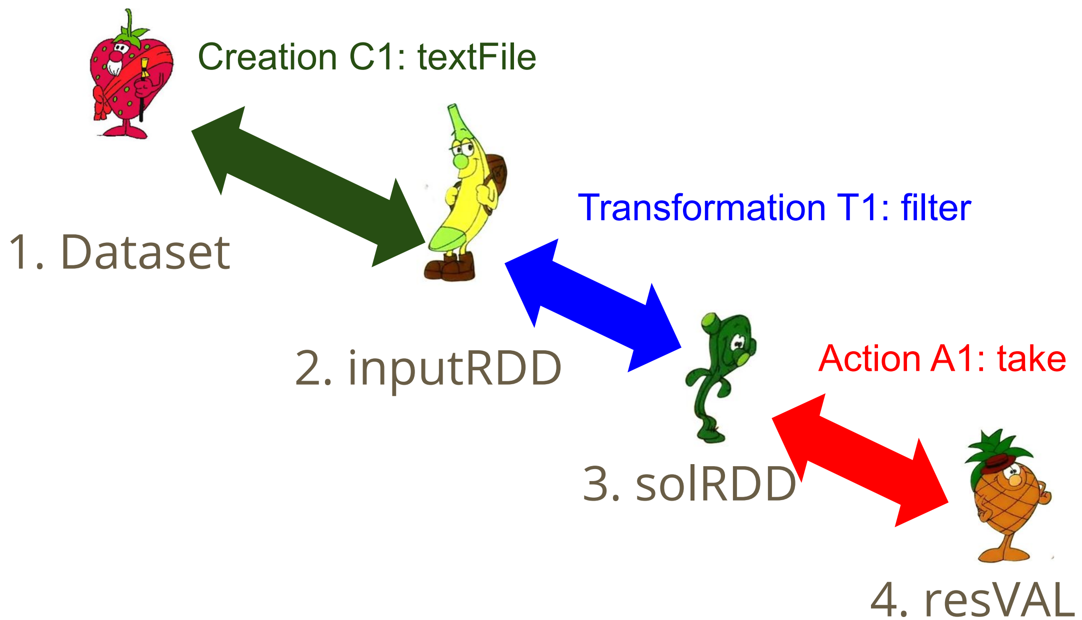
3. resVAL = solRDD.take(2)

4. for item in resVAL:
print(item)

Lazy Evaluation

Let's focus on User Program 1…

What RDD operations are these characters related by?



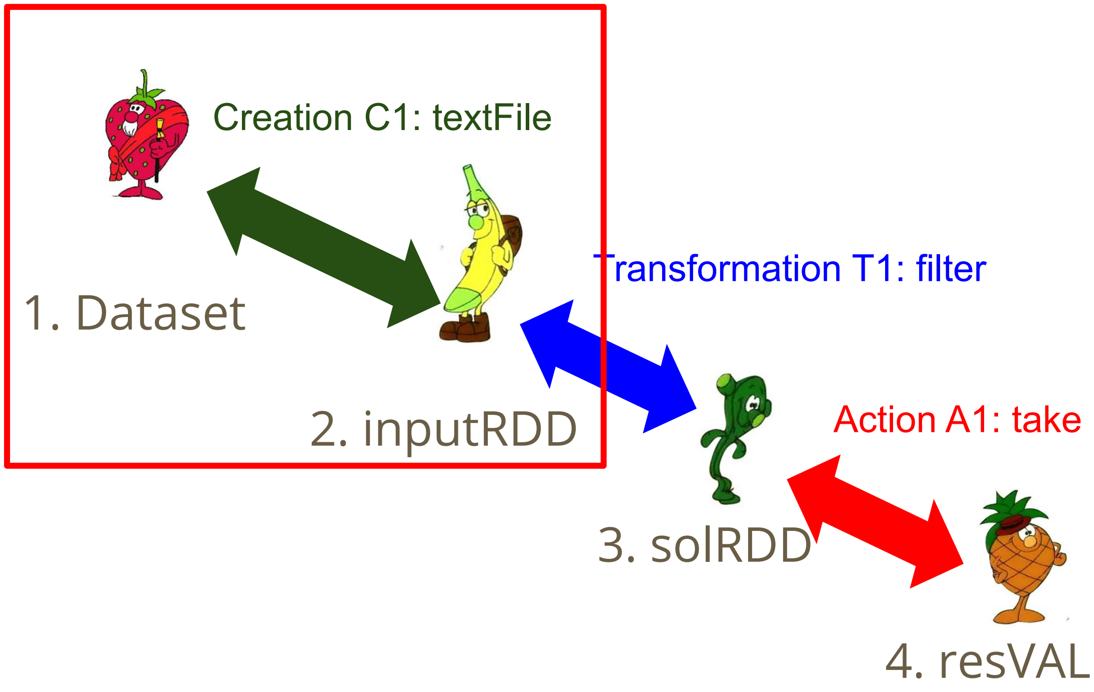
User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

Let's focus on User Program 1…

What RDD operations are these characters related by?



User Program 1

1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

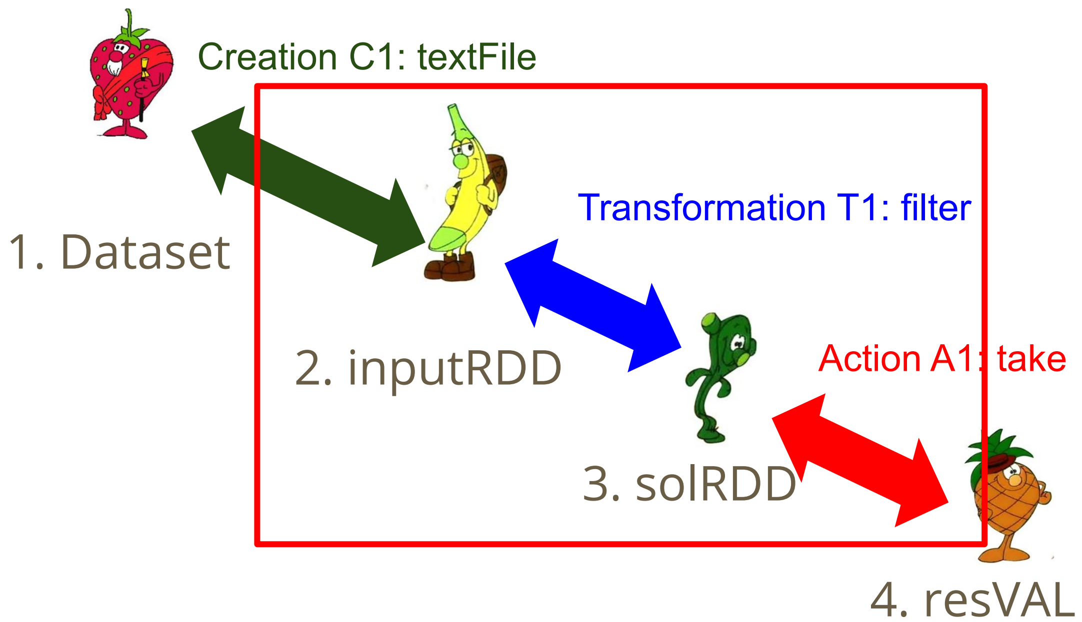
3. resVAL = solRDD.take(2)

4. for item in resVAL:
print(item)

Lazy Evaluation

Let's focus on User Program 1…

<u>What RDD operations are these characters related by?</u>



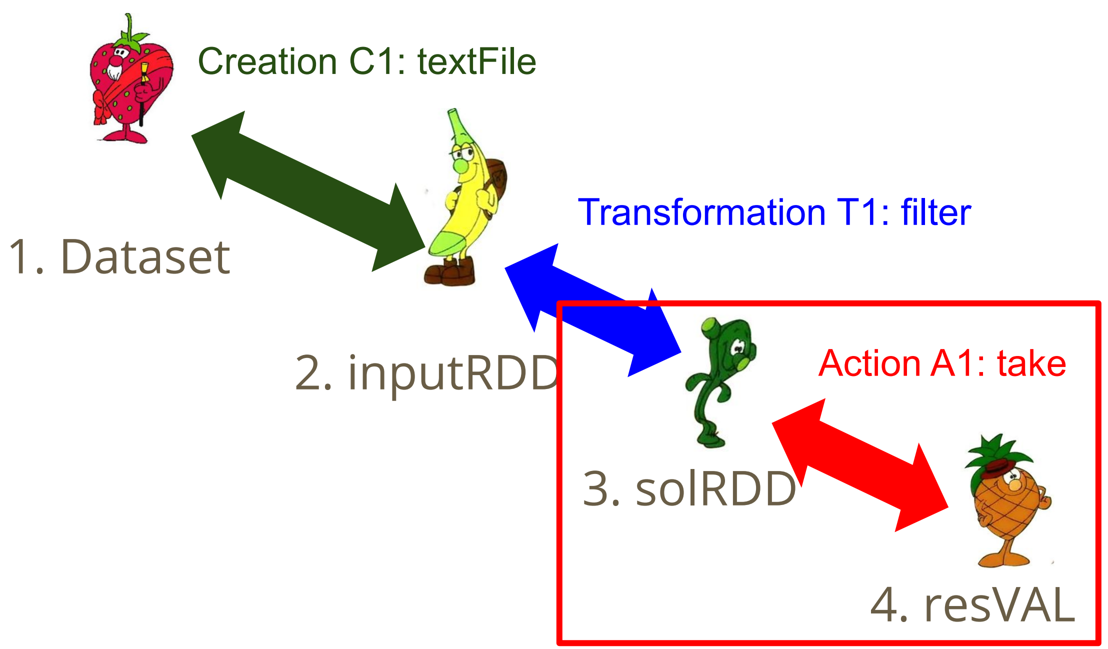
User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

Let's focus on User Program 1…

What RDD operations are these characters related by?



User Program 1

1. inputRDD = sc.textFile(dataset).

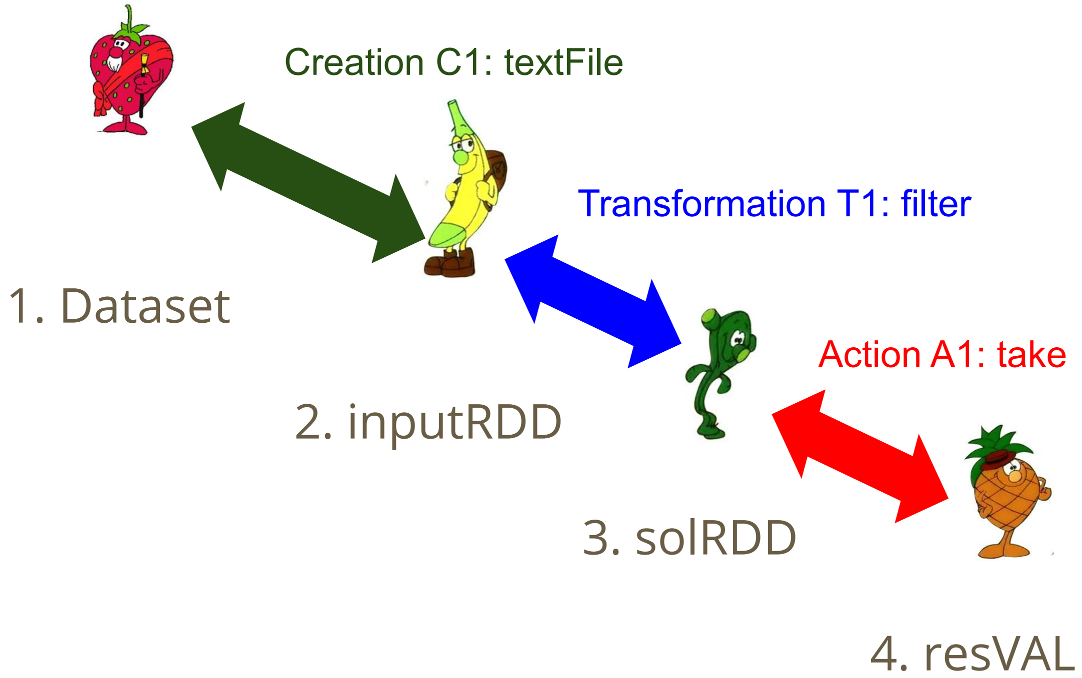
2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
print(item)

Lazy Evaluation

Let's put on the shoes of the **driver process**
and start reasoning about the program…



Spark-driver | User Program 1

1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

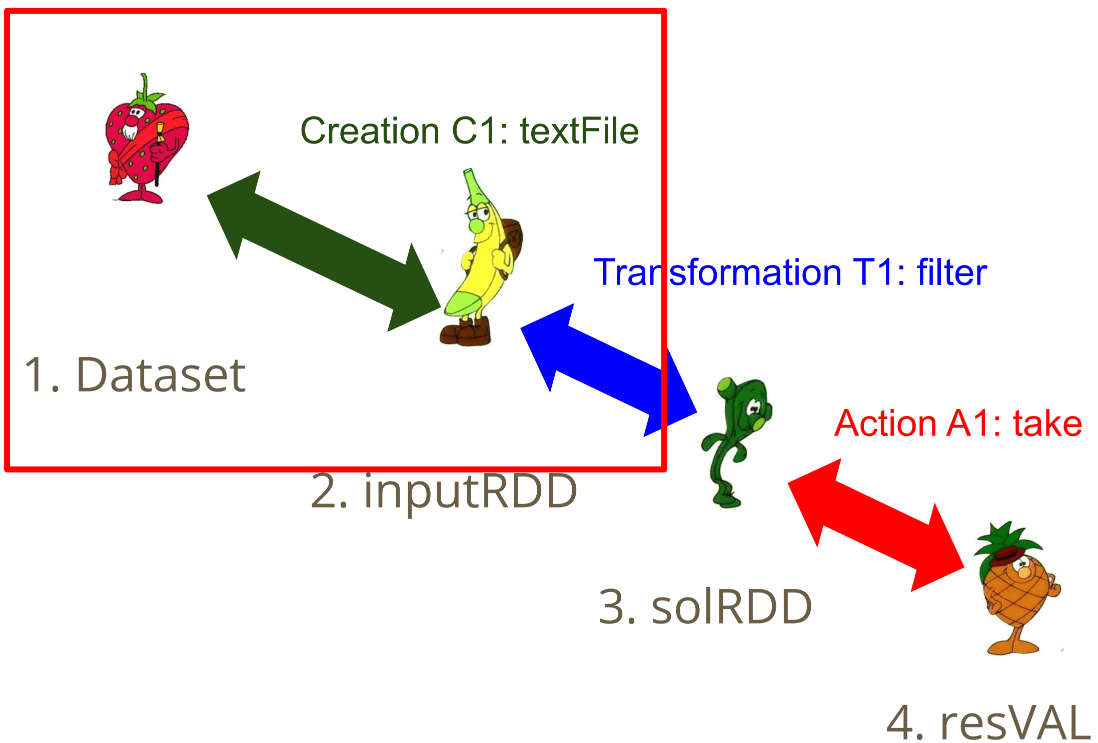
```
4. for item in resVAL:
        print(item)
```

Lazy Evaluation

1. **textFile** requests to read Dataset for filling inputRDD

"What for?" wonders Spark driver.

"I still don't know, so as I'm lazy and I still won't compute anything"

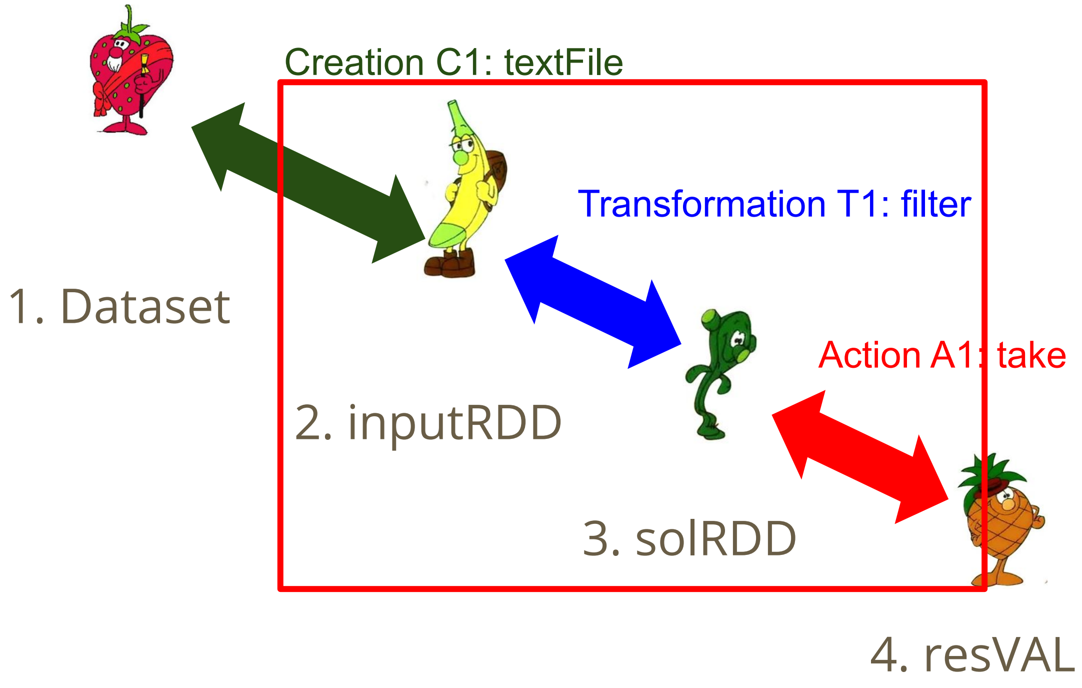


Spark-driver | User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

2. **filter** requests to filter inputRDD to fill solRDD
"What for?" wonders Spark driver.
"I still don't know, so as I'm lazy and I still won't compute anything"

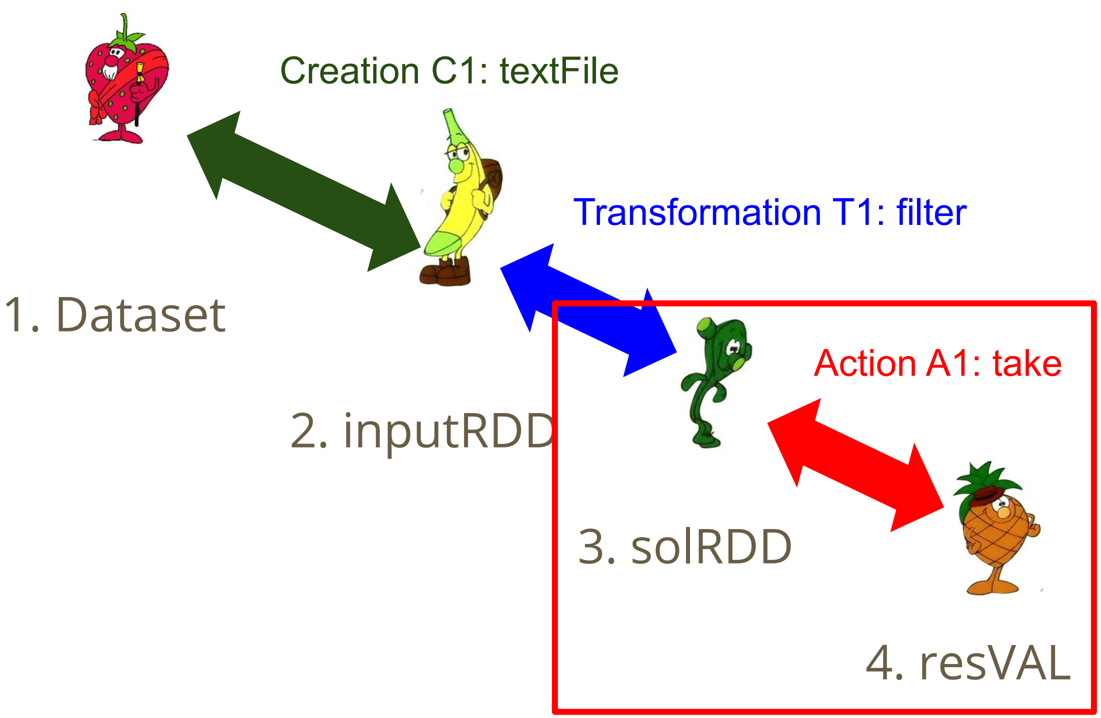


Spark-driver | User Program 1

```
1. inputRDD = sc.textFile(dataset).
2. solRDD = inputRDD.filter(my_func).
3. resVAL = solRDD.take(2)
4. for item in resVAL:
       print(item)
```

Lazy Evaluation

3. **take** requires to get a subset of solRDD to fill resVAL
"Ah, damn it", says the Spark driver.
"I will finally need to compute resVAL.
And so, I need to compute inputRDD and solRDD as well.
Pity, how comfy I was feeling in my lazyness-mode!"

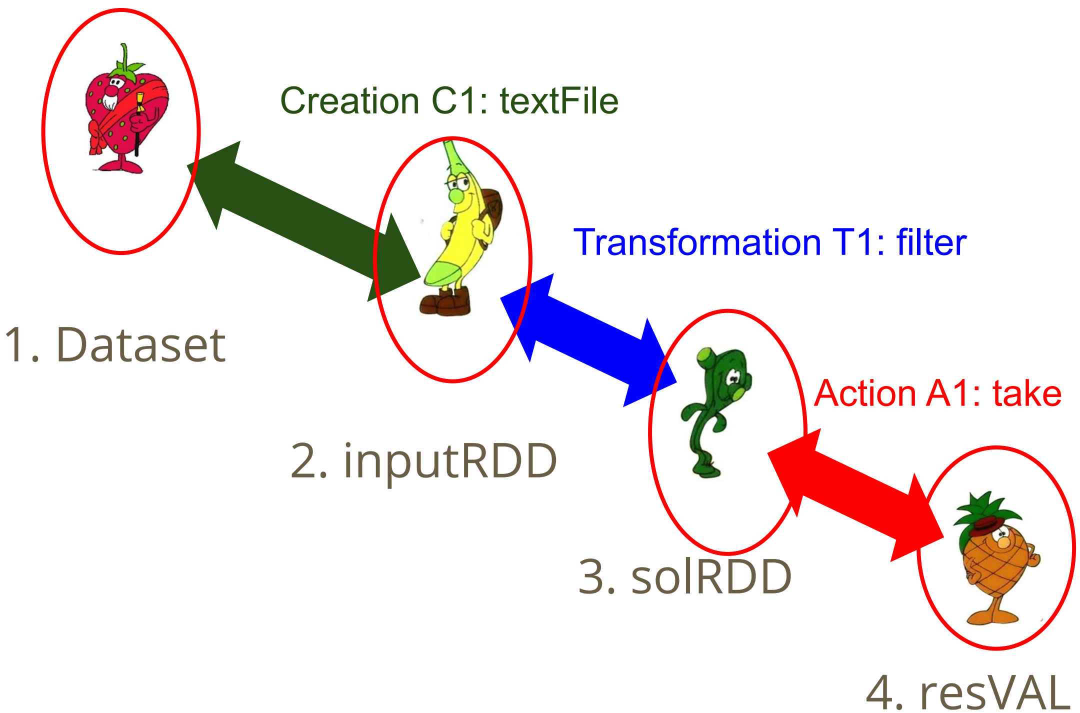


Spark-driver | User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
       print(item)
```

Lazy Evaluation

“But, wait a minute Spark driver”, say the characters of this story.
“Not all is lost! Yes, full lazyness is not possible, but let all of us have a discussion about the minimum amount of things that have to be done.”



Spark-driver | User Program 1

1. inputRDD = sc.textFile(dataset).

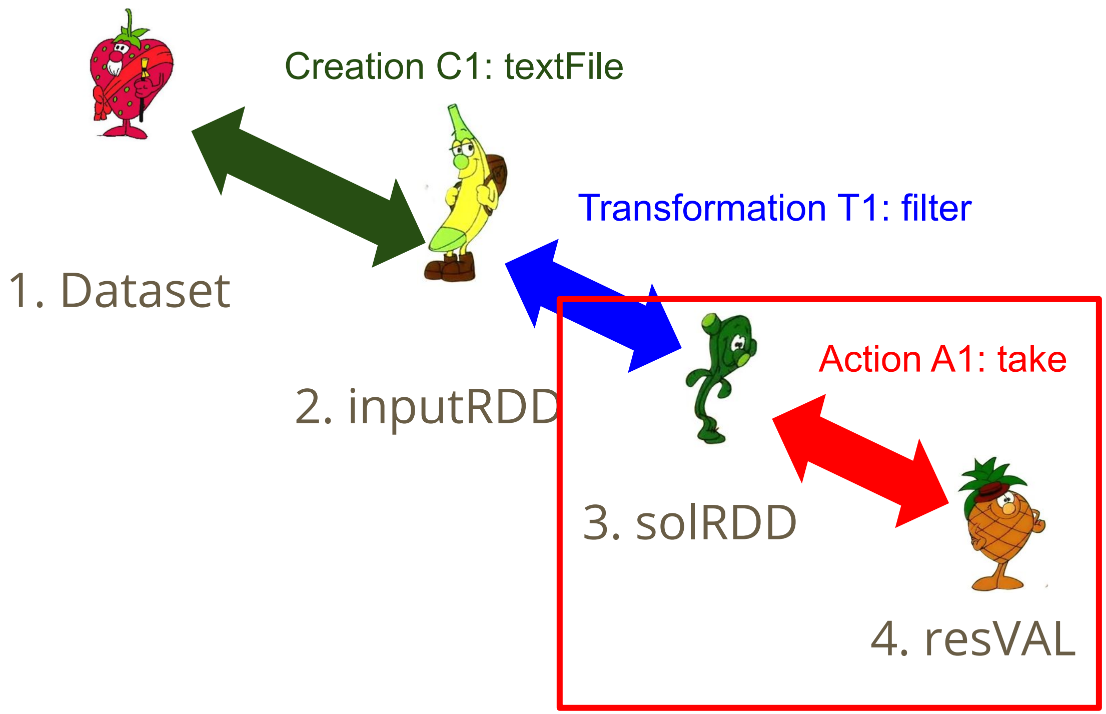
2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

```
4. for item in resVAL:
       print(item)
```

Lazy Evaluation

Let's listen to the conversation between  and 



Spark-driver | User Program 1

1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

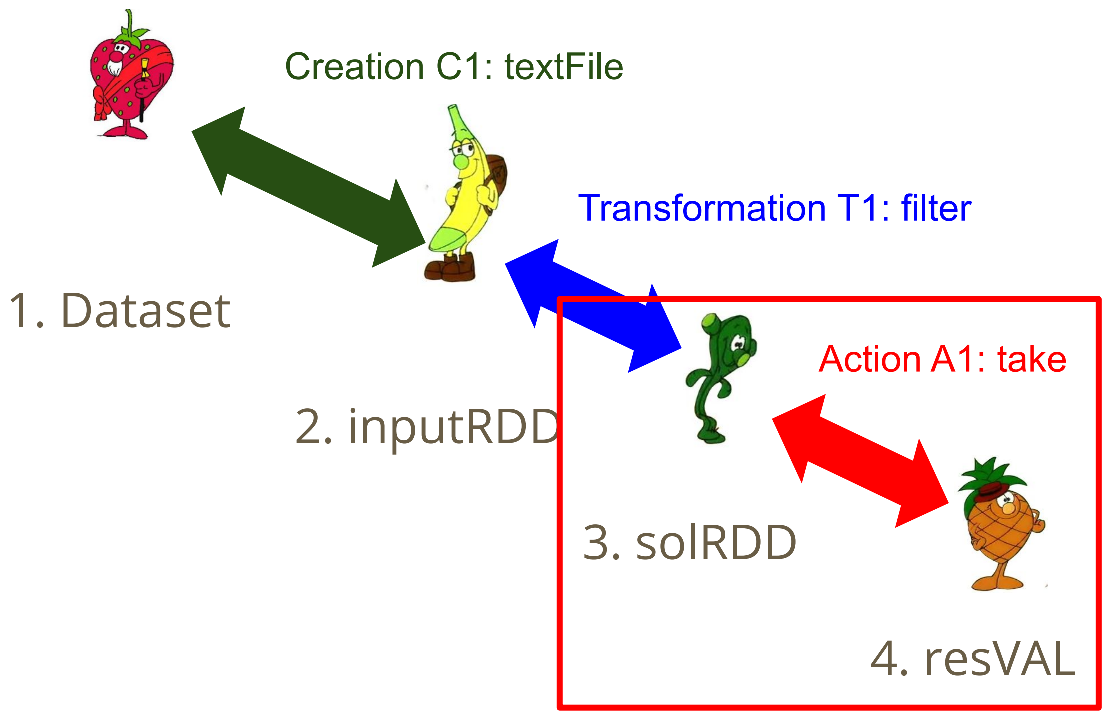
3. resVAL = solRDD.take(2)

4. for item in resVAL:
print(item)

Lazy Evaluation

- “solRDD, I just want to take 2 lines from you, so there is no need for you to be fully computed”.

- Ok, perfect" - replies solRDD.



Spark-driver	User Program 1

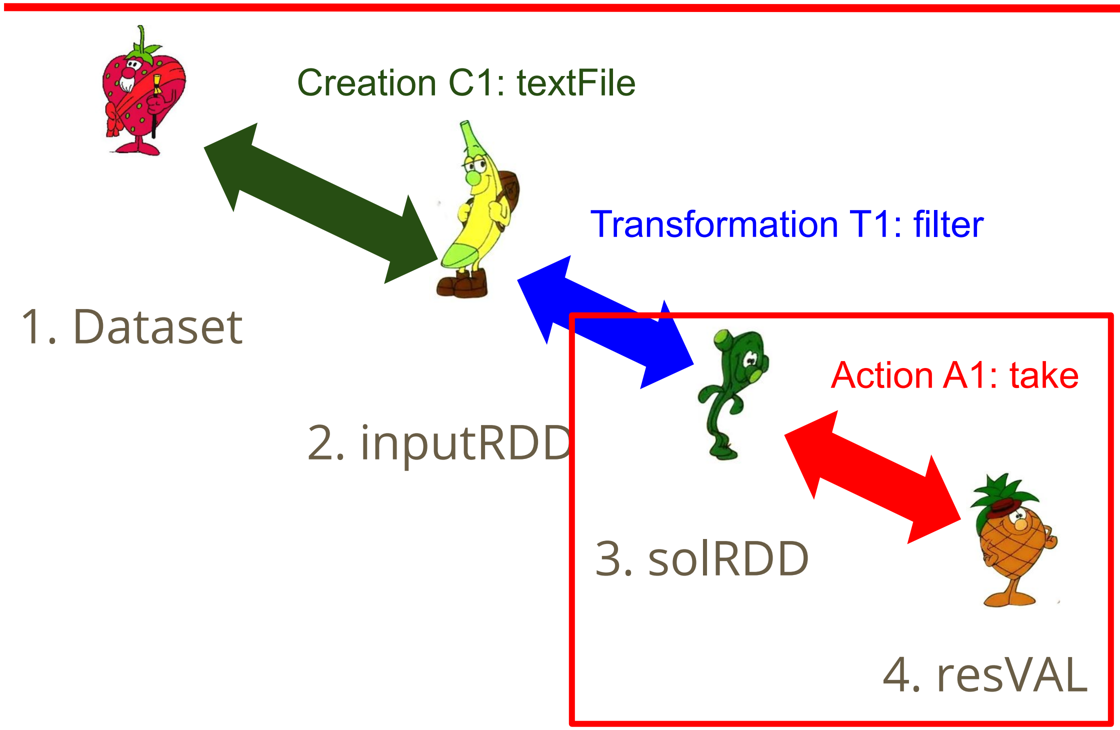
```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
       print(item)
```

Lazy Evaluation

- “solRDD, according to the **take** operation we are related by, I don’t need you to be fully computed, just to get 2 of your elements”.

- Ok, perfect, that’s brilliant!" - replies solRDD.

As you see, the agreement here was easy :)



Spark-driver | User Program 1

1. inputRDD = sc.textFile(dataset).

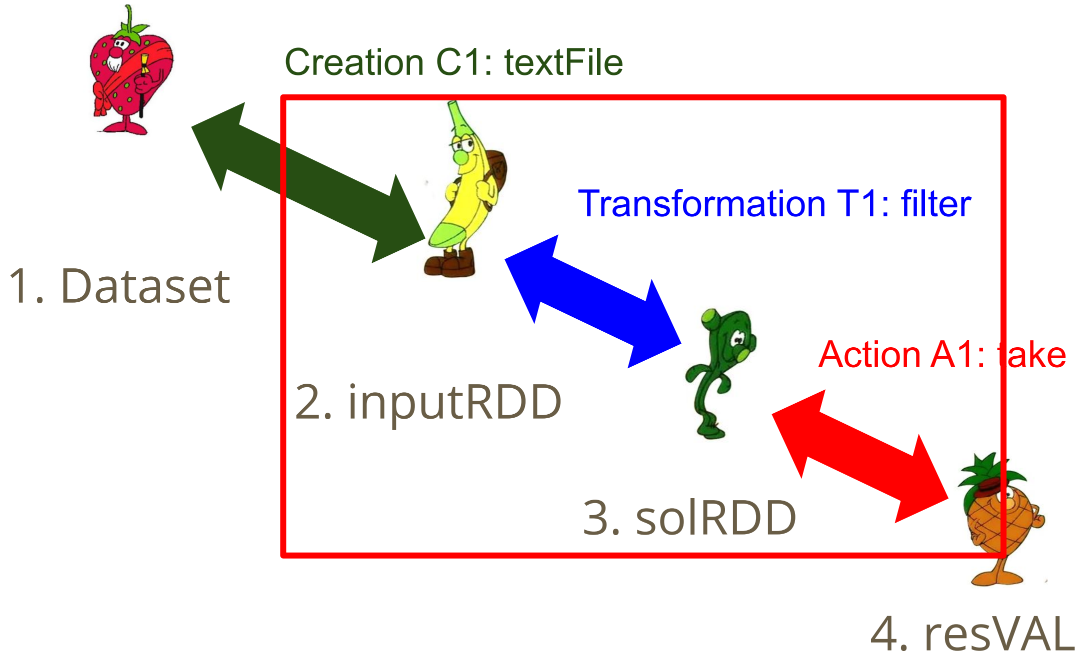
2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
print(item)

Lazy Evaluation

Let's listen to the conversation between  and 



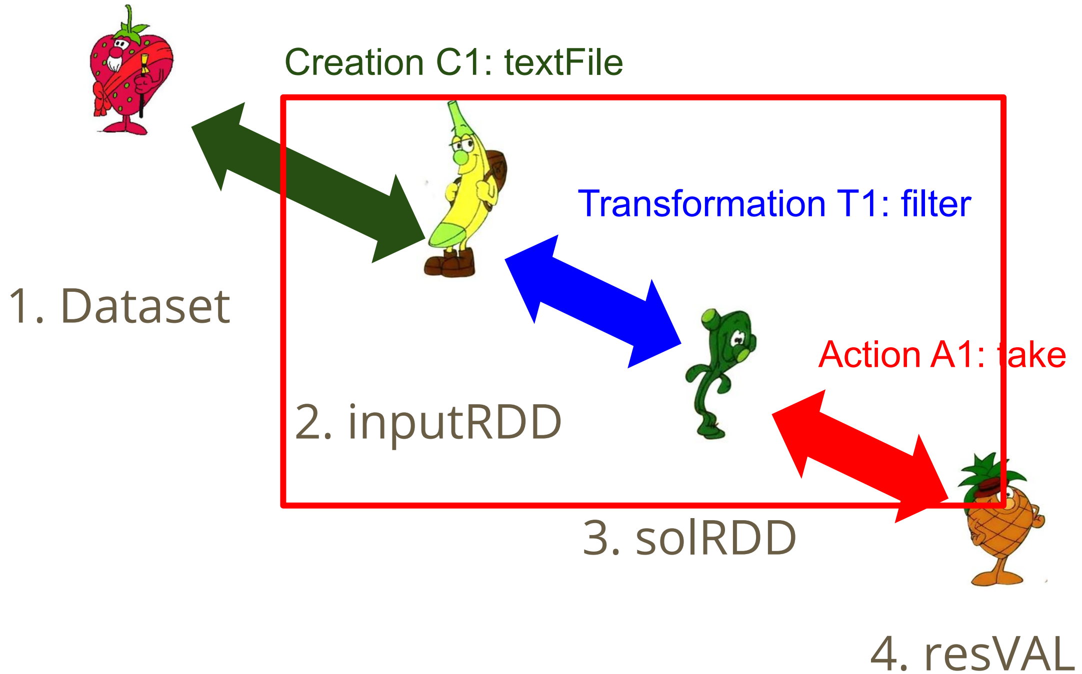
Spark-driver	User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

- "inputRDD, according to the **filter** operation we are related by, you are supposed to be fully computed" says solRDD.

- "Oh no, really?", replies inputRDD, who doesn't like to work and was hoping to be as lazy as possible.



Spark-driver | User Program 1

1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

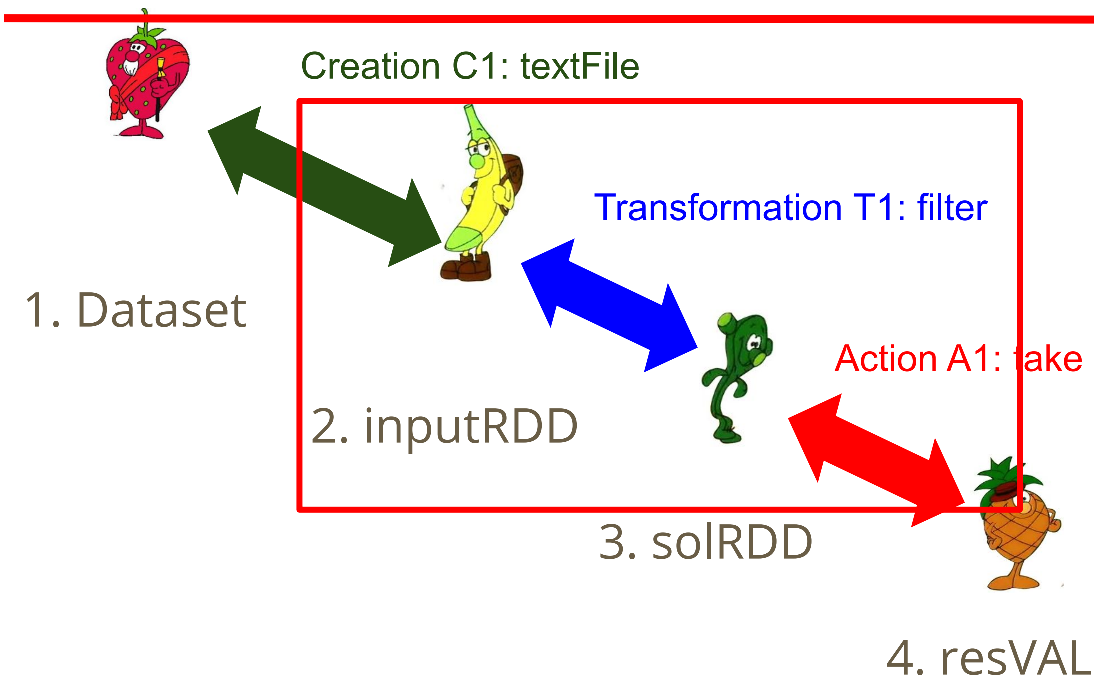
3. resVAL = solRDD.take(2)

4. for item in resVAL:
print(item)

Lazy Evaluation

- "Uhm, no, I also have some good news from resVAL. Apparently it only needs 2 elements from me. Thus, I will only need 2 elements from you.

- "Oh, these are indeed great news!
So, does this mean I only need to compute 2 elements myself?", wonders inputRDD, now happy as a kid on her/his birthday.



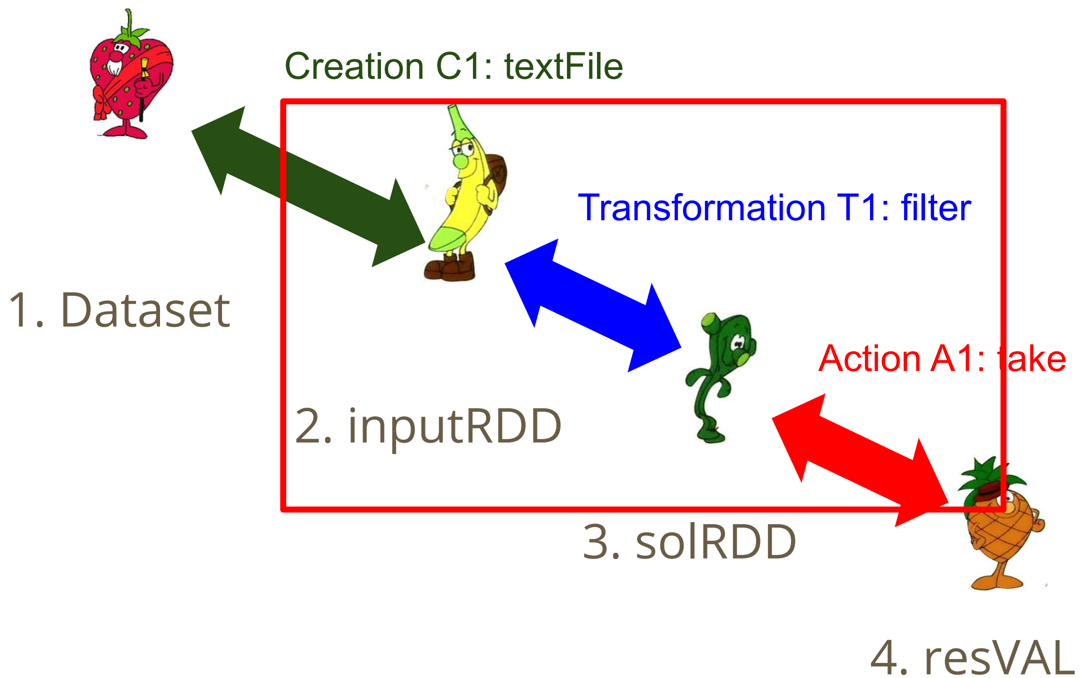
Spark-driver	User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

-"No, unfortunately I cannot guarantee you that. Maybe 2 elements is enough, but maybe you will need you to compute 3, 4, 5, or even 1 million. Indeed, you will need to compute as many elements as needed, until 2 of these elements satisfies the property of our **filter** transformation".

- "Oh, I see."

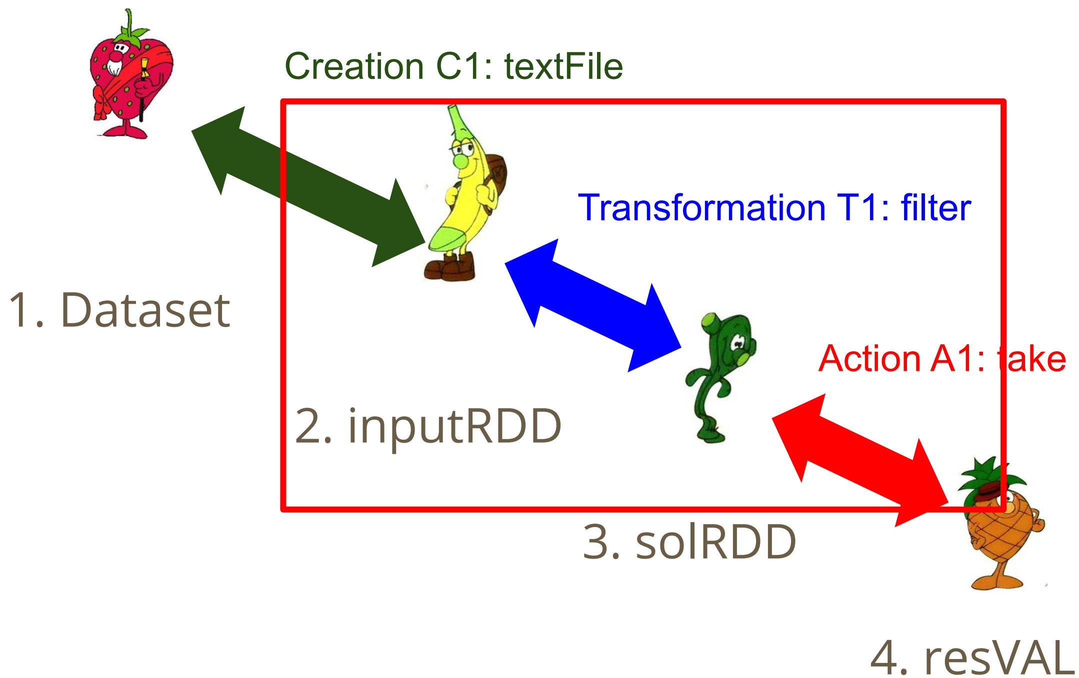


Spark-driver | User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

"I know, but let's do something:
You compute yourself <u>one</u> element at a time. Just one!
Then, you come back to me straight away, and we both ask **filter** if this element satisfies the property.
And we repeat this process until 2 elements satisfy it. Does it sound ok?"

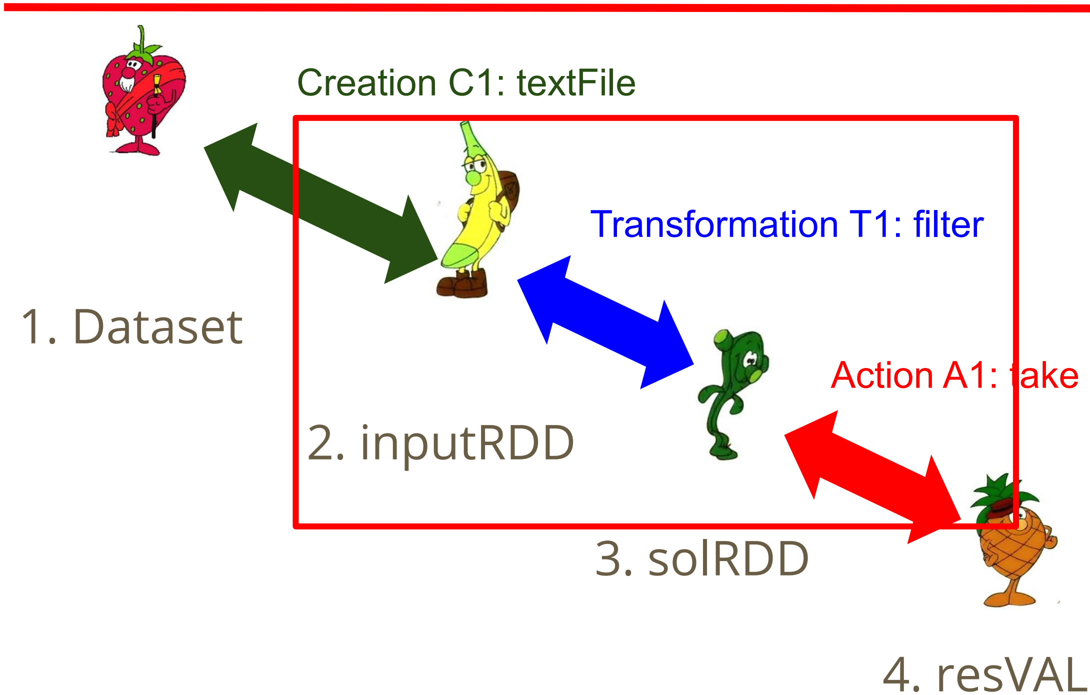


Spark-driver	User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
       print(item)
```

Lazy Evaluation

"Yes, it definitely sounds much better than computing myself entirely", replies inputRDD.

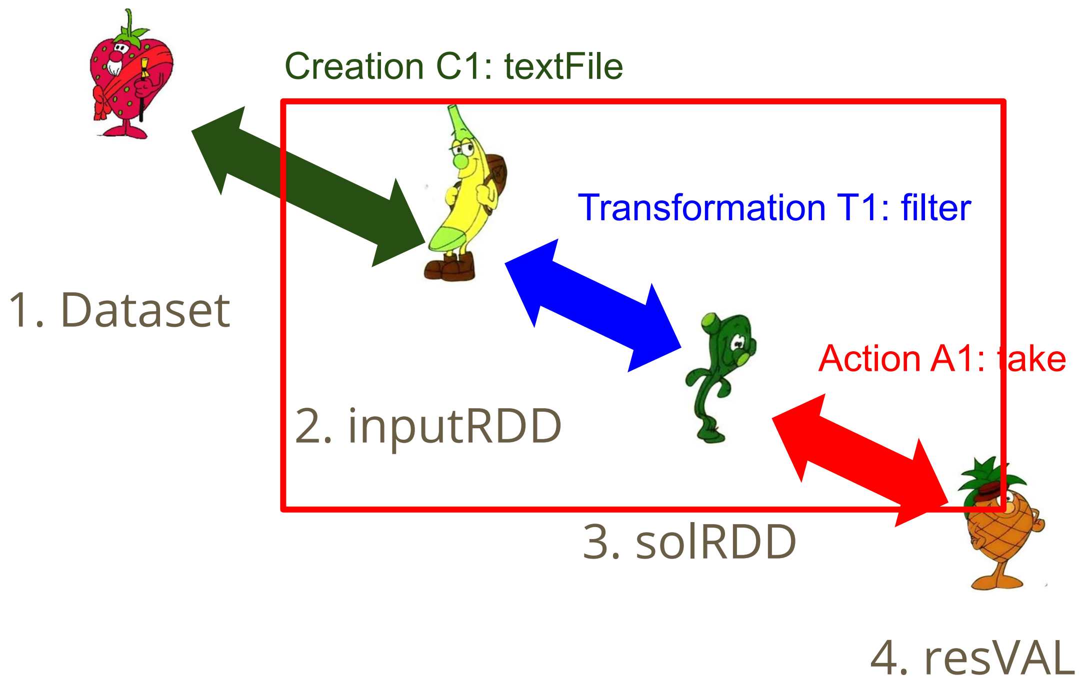


Spark-driver | User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

"I know, lazy evaluation is great!
Do you think this story will make the students to understand lazy evaluation? Or they might not even bother in reading this?", wonders solRDD.

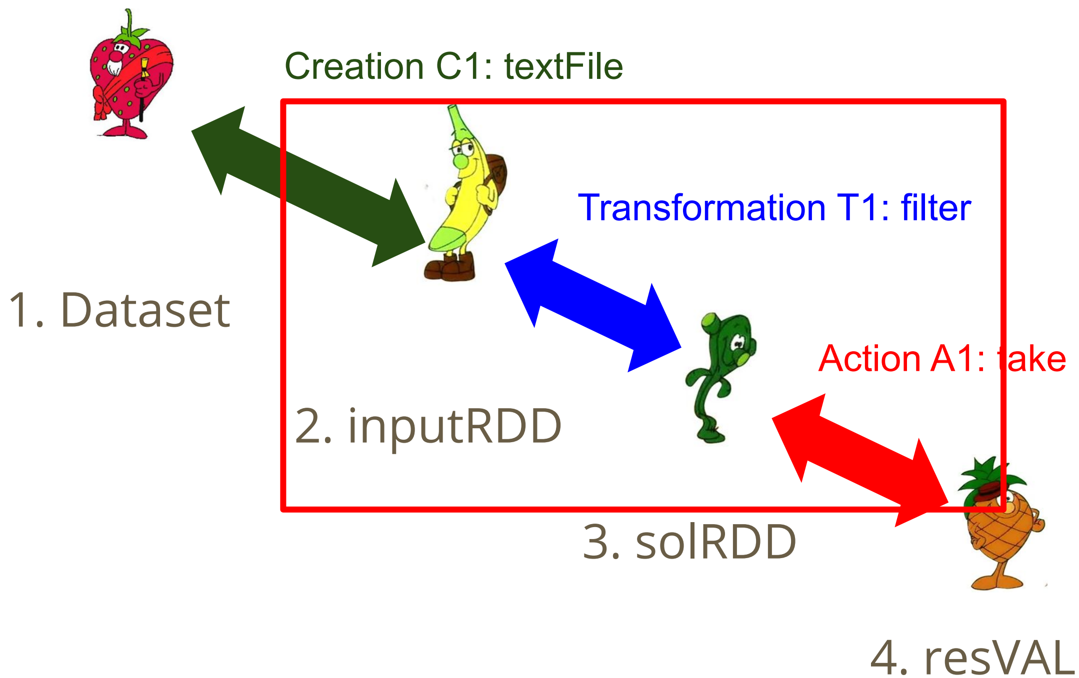


Spark-driver	User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

"I don't know. It is indeed a great story, so I will give a 50%-50% chance. But, anyway, why do we care? We are only RDDs, so this is not our business", inputRDD finishes.



Spark-driver | User Program 1

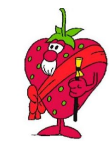
1. inputRDD = sc.textFile(dataset).

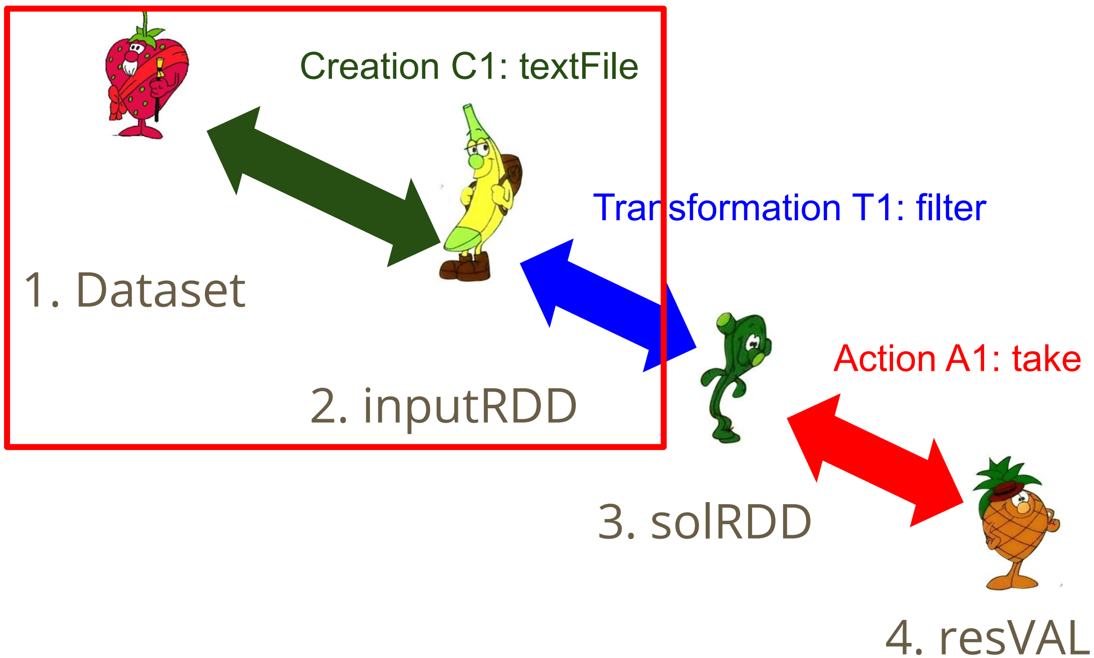
2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)

Lazy Evaluation

Let's listen to the conversation between  and 



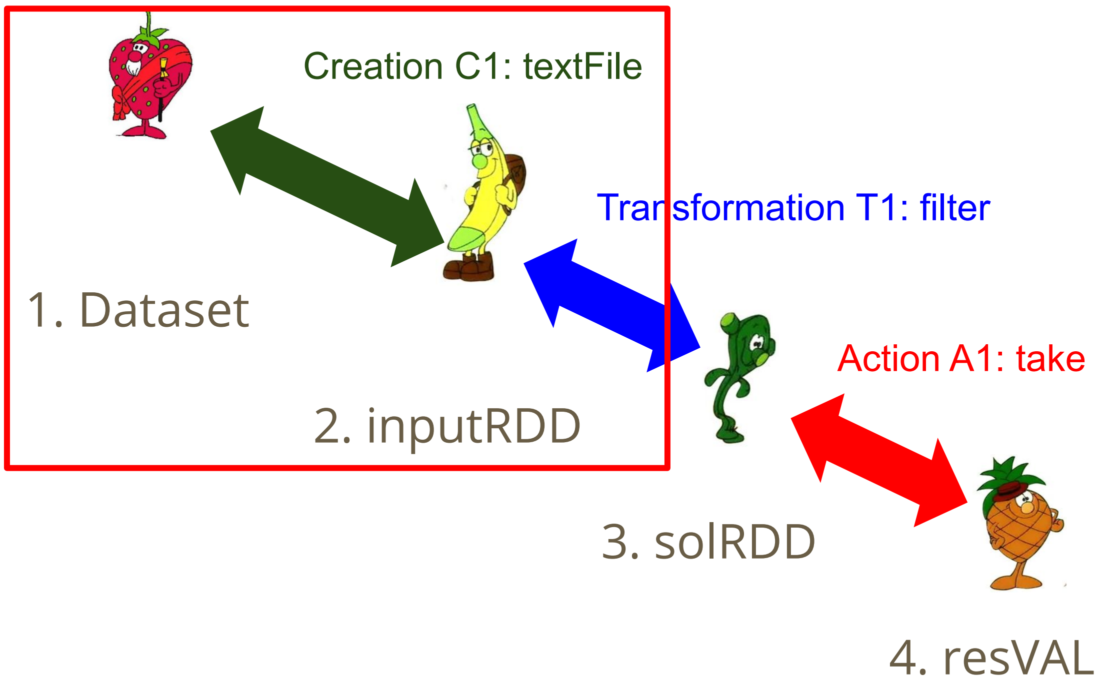
Spark-driver	User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
       print(item)
```

Lazy Evaluation

- "Dataset, according to the operation **textFile** we are related by, you are supposed to be fully computed", says inputRDD.

- "Oh no, really?", replies Dataset, who doesn't like to work and was hoping to be as lazy as possible.



Spark-driver | User Program 1

1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

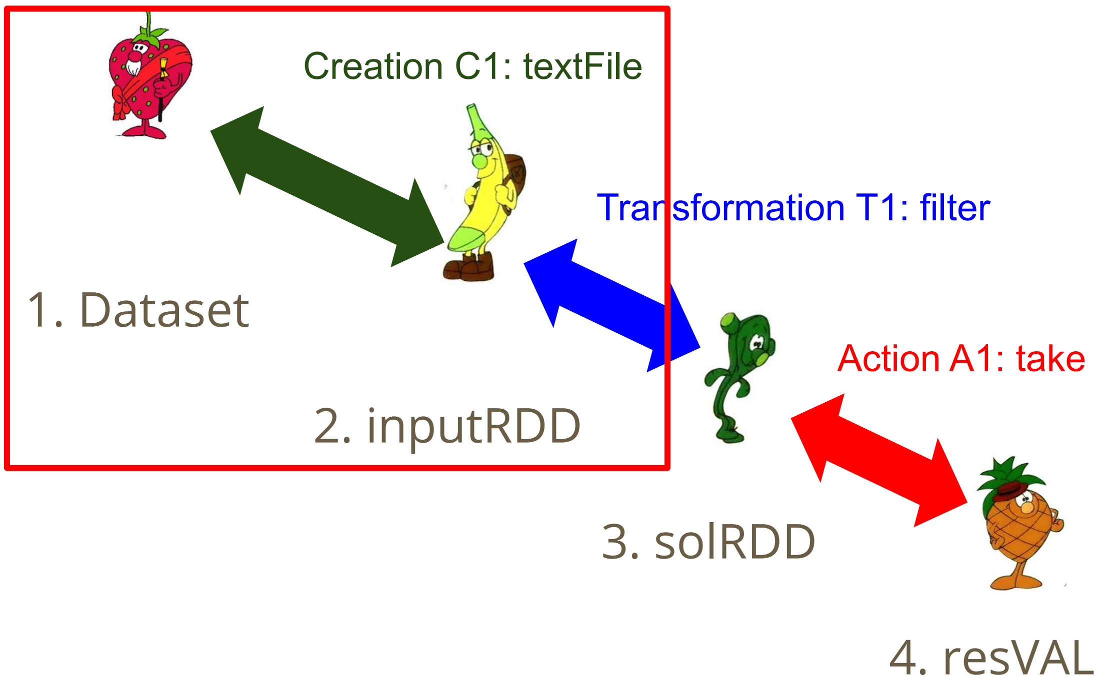
3. resVAL = solRDD.take(2)

4. for item in resVAL:
print(item)

Lazy Evaluation

- "Uhm, no. I have an agreement with solRDD; it only needs me to compute my elements one by one at a time, until 2 of them satisfy certain property.

- Oh, I see.



Spark-driver	User Program 1

1. inputRDD = sc.textFile(dataset).

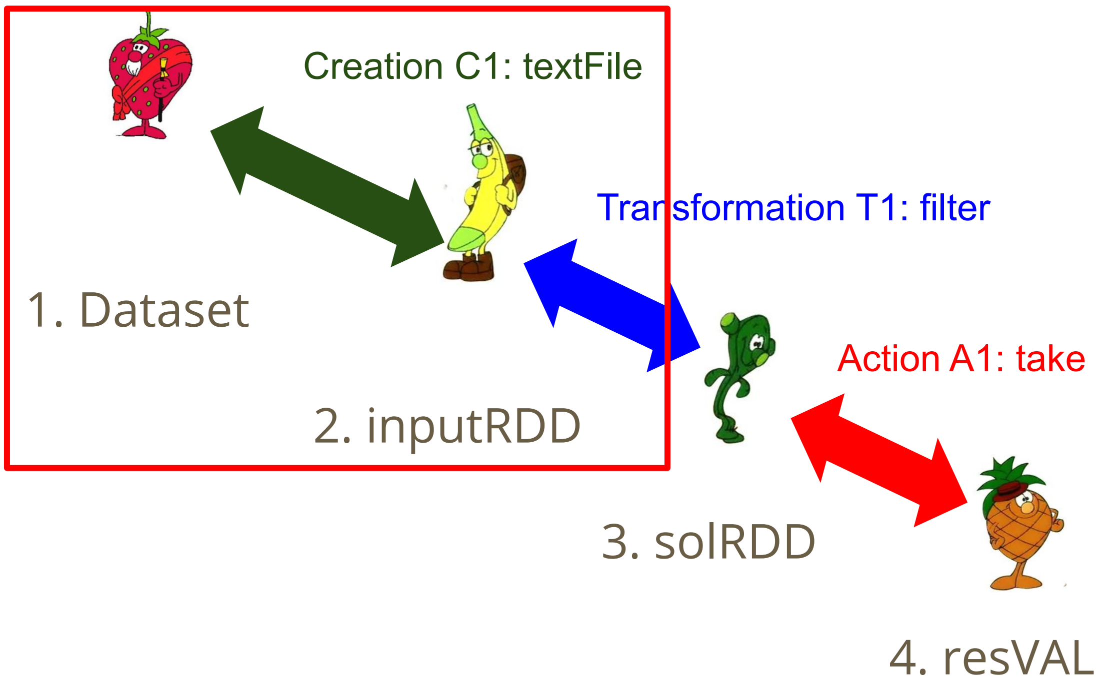
2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

```
4. for item in resVAL:
       print(item)
```

A First Example: Lazy Evaluation

- So let's take advantage of this and proceed as follows:
I will ask you for one element at a time. Just one!
I might come back to you multiple times, requesting for a new element, until either solRDD gets its 2 elements, or you give me the entire dataset, whatever happens first.



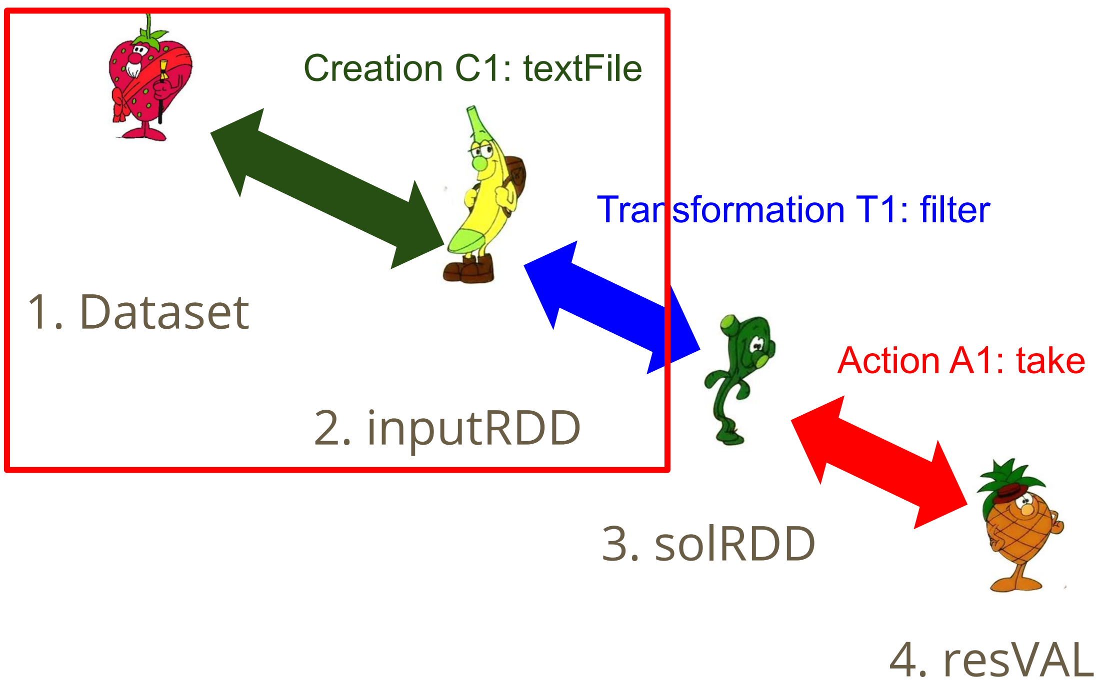
Spark-driver | User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
       print(item)
```

Lazy Evaluation

- Once I don't need more elements from you, I will let you know. Does it sound ok?"

- "Yes, it definitely sounds much better than me doing the effort of passing you the entire Dataset straight away".



Spark-driver | User Program 1

1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

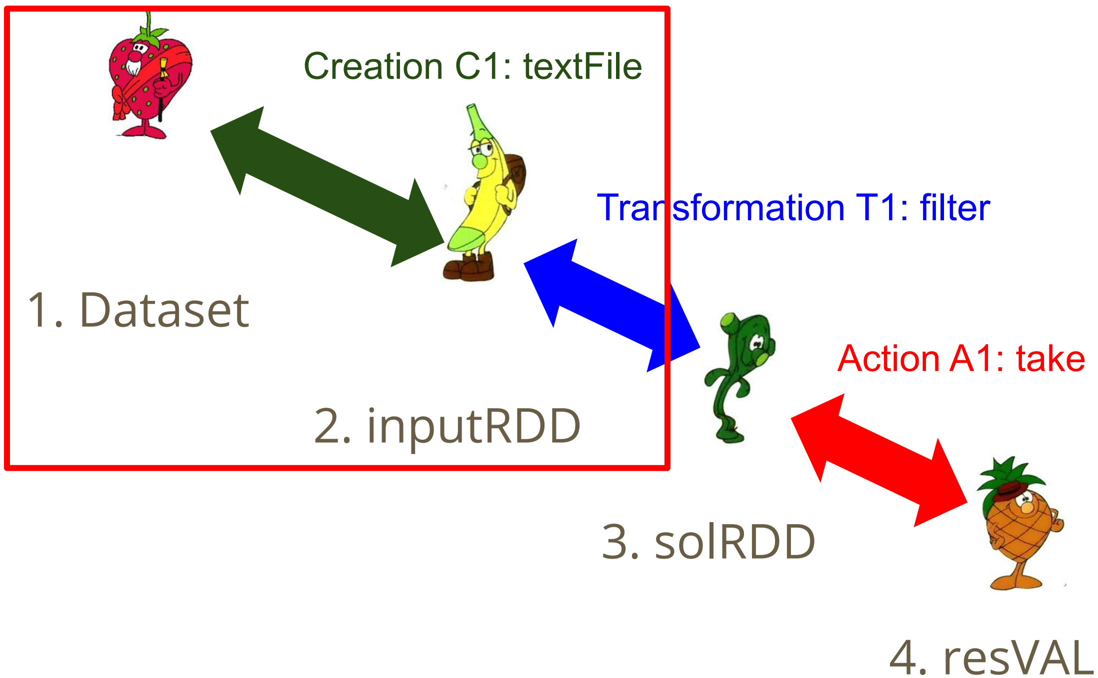
3. resVAL = solRDD.take(2)

4. for item in resVAL:
print(item)

Lazy Evaluation

- "I know, lazy evaluation is great!
  Do you think..."

- "Oh, please stop!
Perfect, we have reach an agreement, so let's move on".

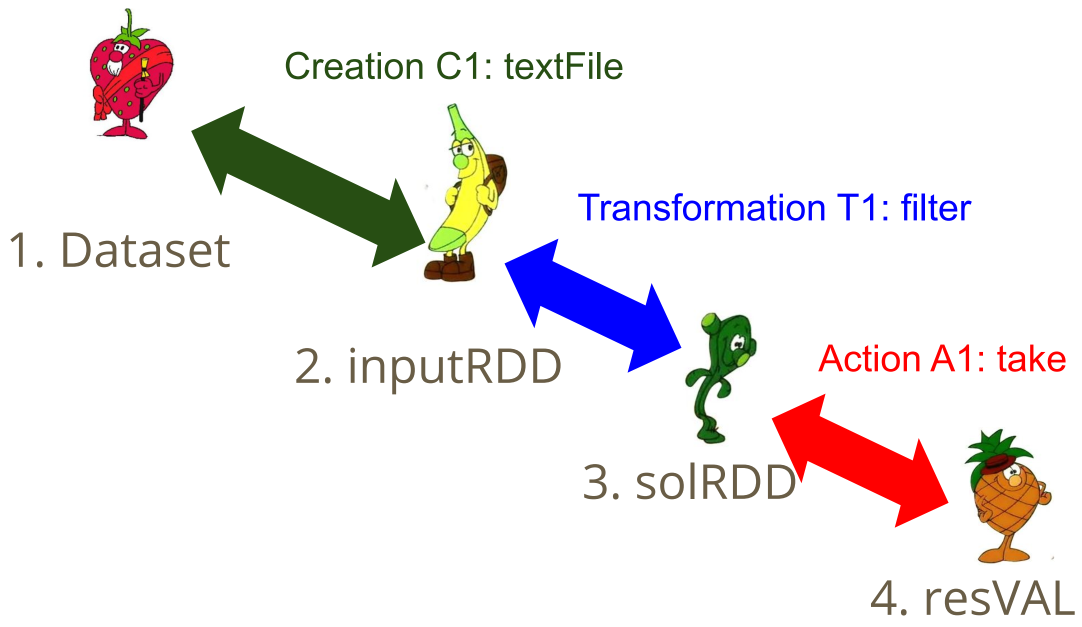


Spark-driver | User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
        print(item)
```

Lazy Evaluation

And that's how lazy evaluation makes the execution of this program more efficient, requiring way less computation.



Spark-driver | User Program 1

```
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. resVAL = solRDD.take(2)

4. for item in resVAL:
       print(item)
```

Lazy Evaluation

Let's focus on User Program 2…

<u>What are the main variables (represented as cartoon characters) in this program?</u>

1. Dataset

2. inputRDD

3. solRDD

4. new_dir

User Program 2

1. inputRDD = sc.textFile(dataset).

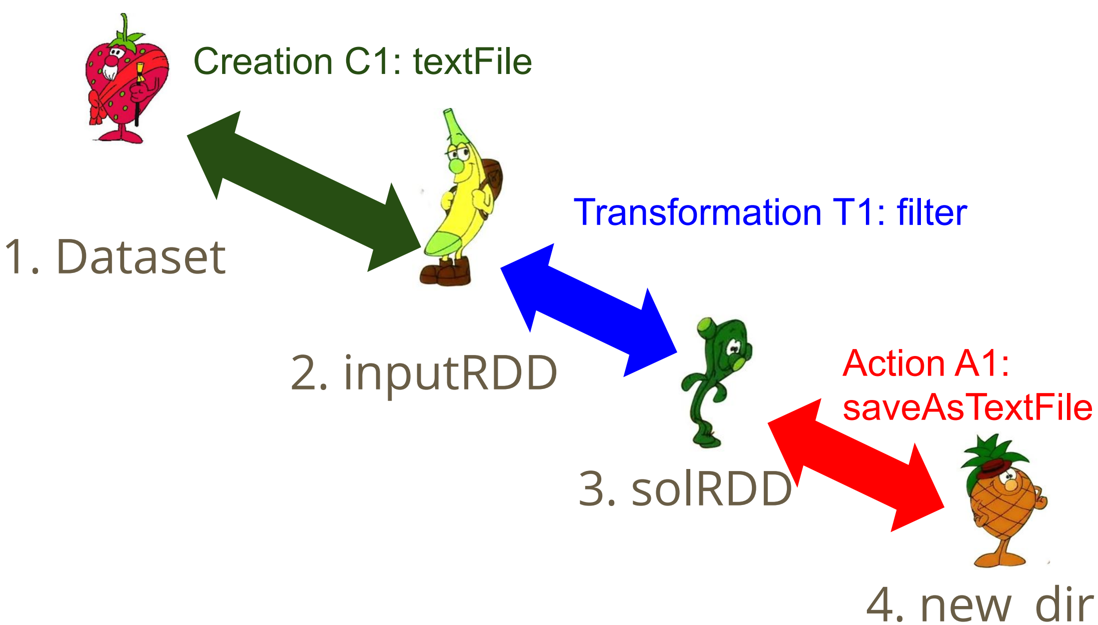
2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

Let's focus on User Program 2…

What RDD operations are these characters related by?



User Program 2

1. inputRDD = sc.textFile(dataset).

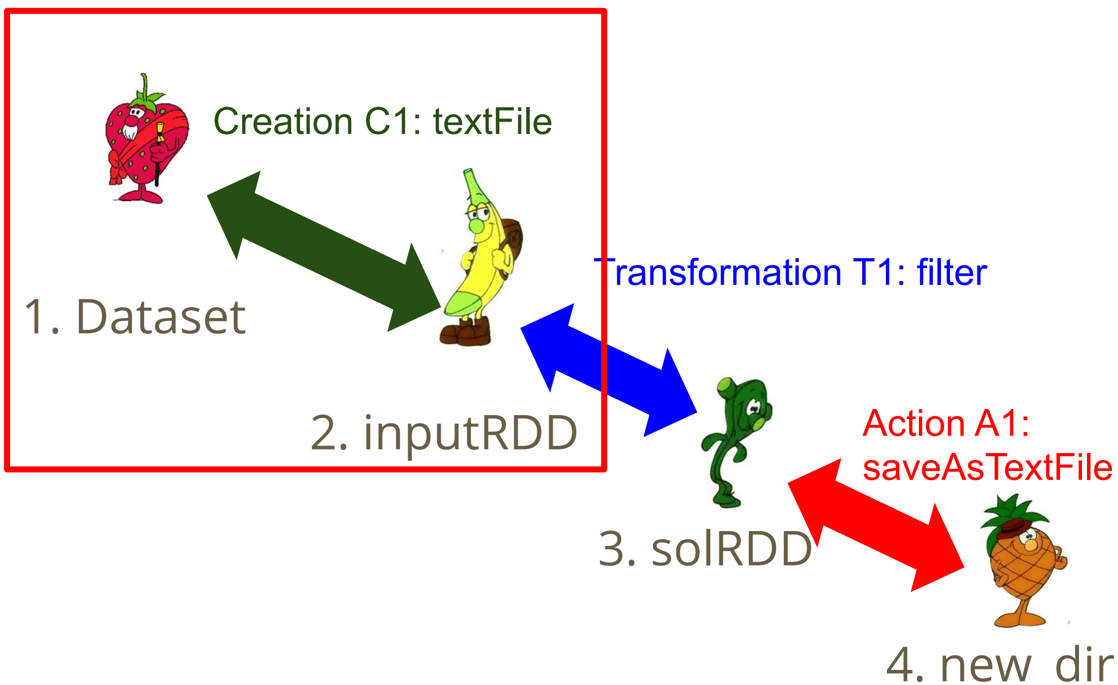
2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

Let's focus on User Program 2…

What RDD operations are these characters related by?



User Program 2

1. inputRDD = sc.textFile(dataset).

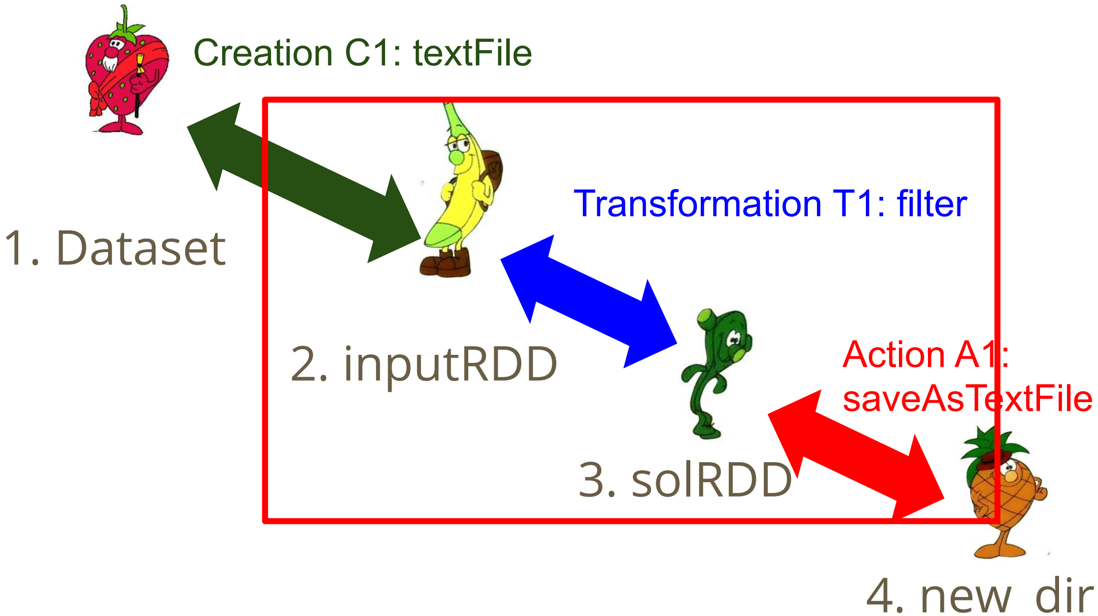
2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

Let's focus on User Program 2…

What RDD operations are these characters related by?



User Program 2

1. inputRDD = sc.textFile(dataset).

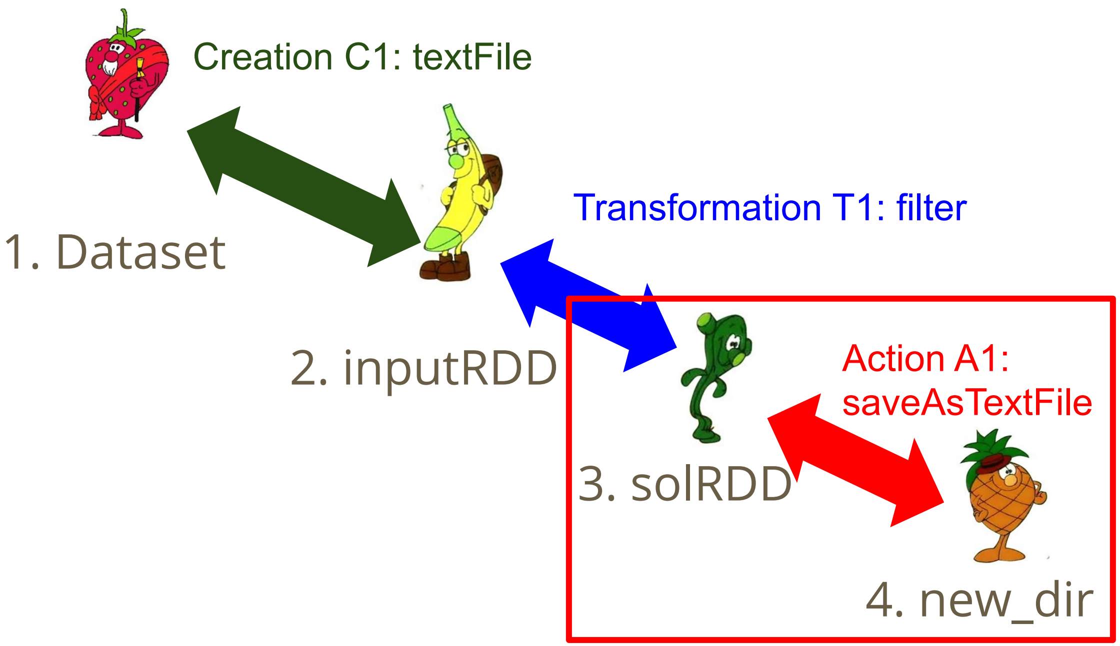
2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

Let's focus on User Program 2…

<u>What RDD operations are these characters related by?</u>

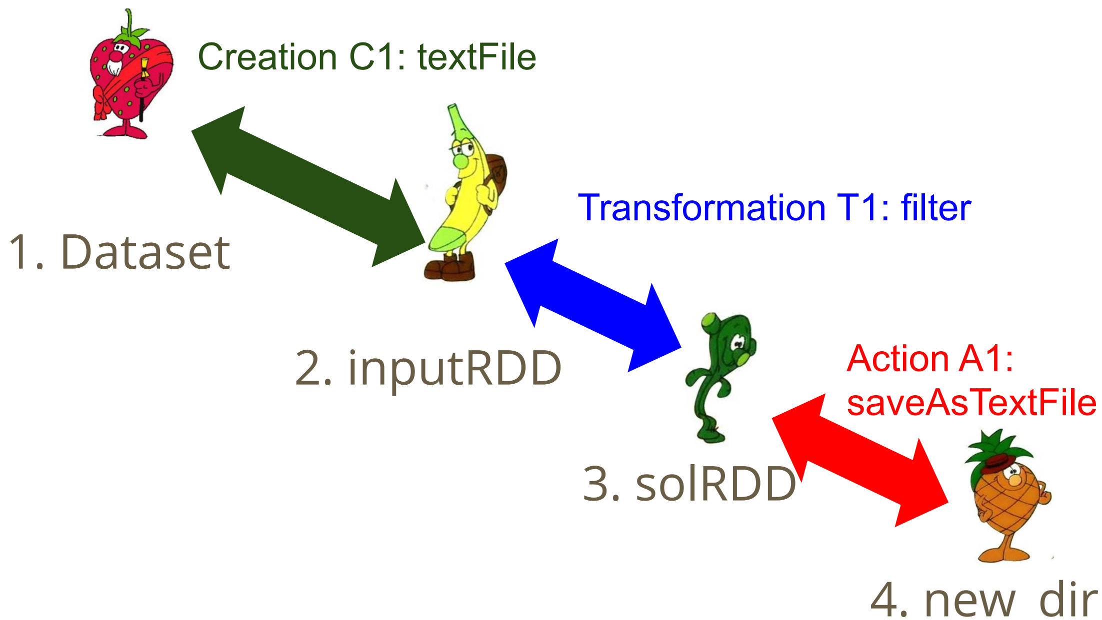


User Program 2

```
1. inputRDD = sc.textFile(dataset).
2. solRDD = inputRDD.filter(my_func).
3. solRDD.saveAsTextFile(new_dir)
```

Lazy Evaluation

Let’s put on the shoes of the **driver process**
and start reasoning about the program…



Spark-driver	User Program 2

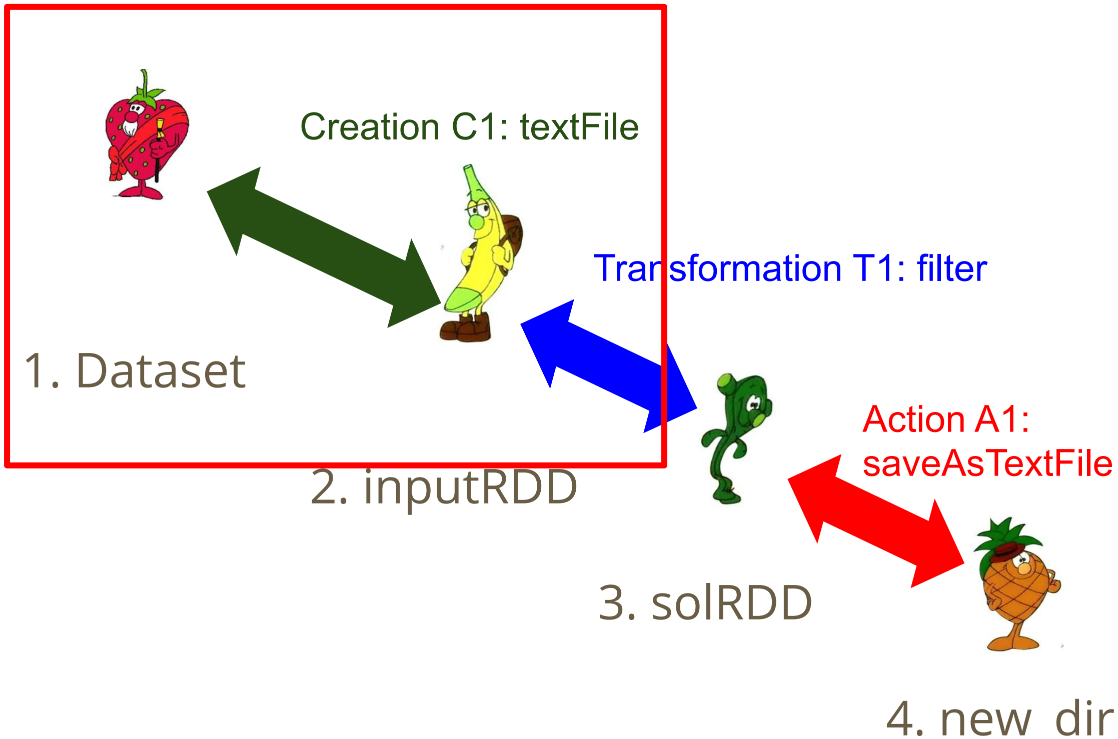
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

1. **textFile** requests to read Dataset for filling inputRDD
"What for?" wonders Spark driver.
"I still don't know, so as I'm lazy and I still won't compute anything"



Spark-driver	User Program 2

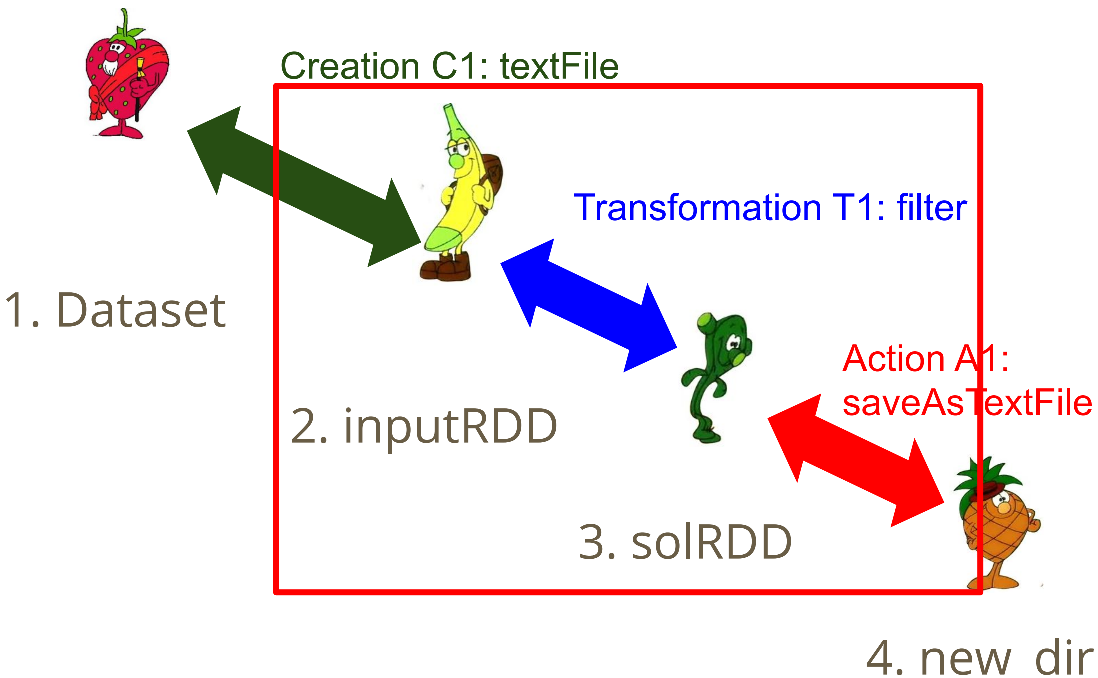
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

2. **filter** requests to filter inputRDD to fill solRDD
"What for?" wonders Spark driver.
"I still don't know, so as I'm lazy and I still won't compute anything"



Spark-driver	User Program 2

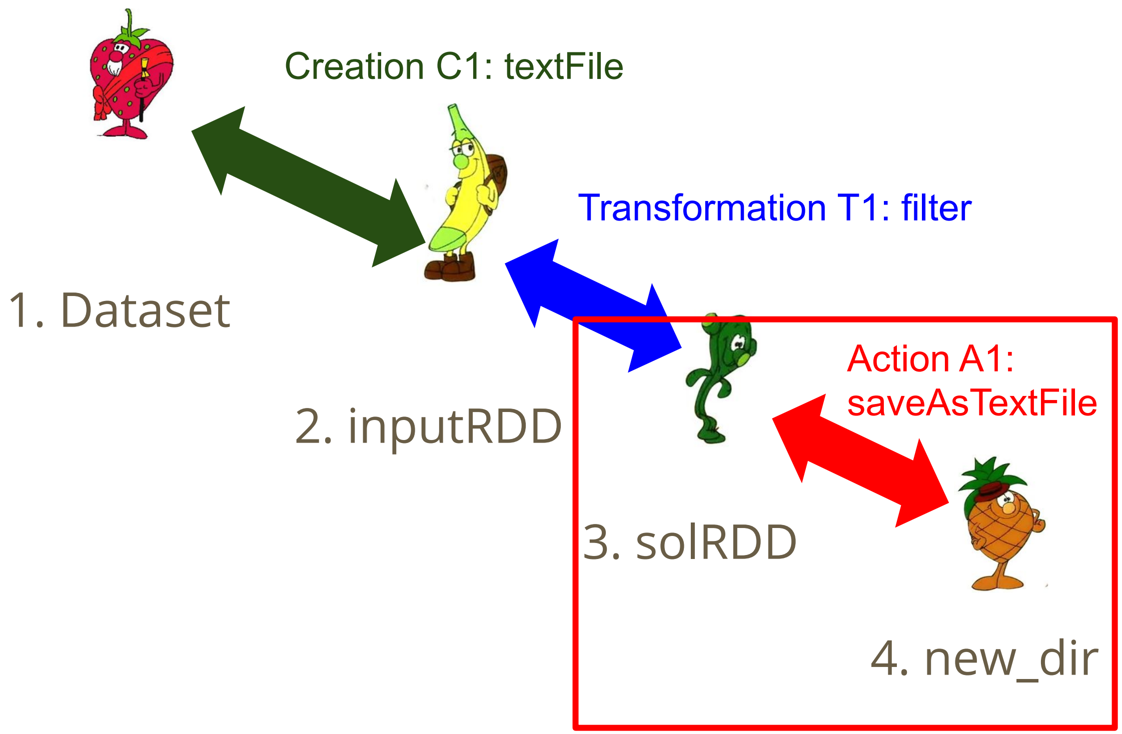
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

3. **saveAsTextFile** requires the whole content of solRDD to write it to new_dir
"Ah, damn it", says the Spark driver.
"I will finally need to do some action. And so, I need to compute inputRDD and solRDD as well.
Pity, how comfy I was feeling in my lazyness-mode!"



Spark-driver	User Program 2

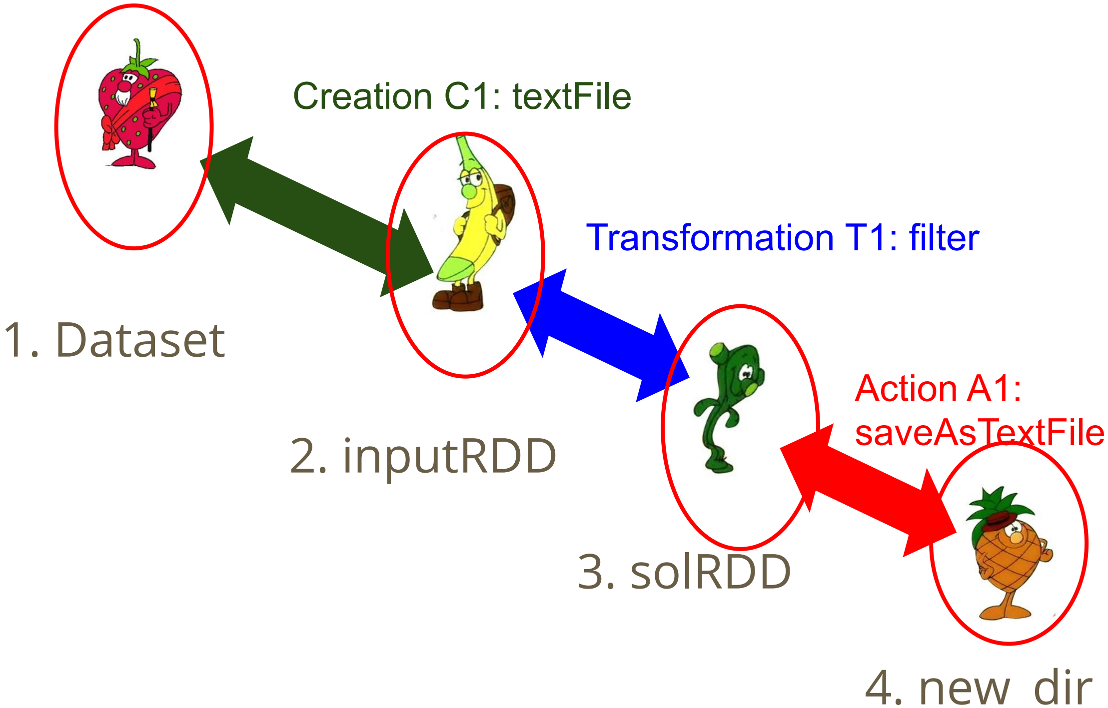
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

“But, wait a minute Spark driver”, say the characters of this story.
“Maybe all of us can have a discussion (as the one we had for the previous program) about the minimum amount of things that have to be done.”



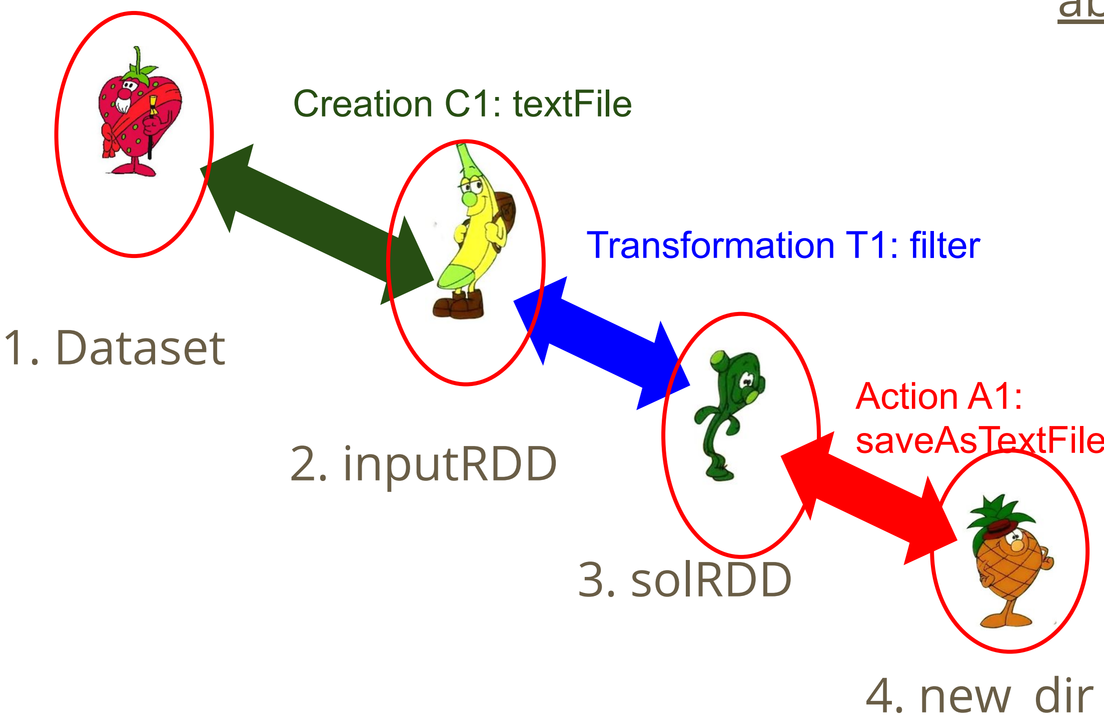
Spark-driver	User Program 2

1. inputRDD = sc.textFile(dataset).
2. solRDD = inputRDD.filter(my_func).
3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

“But, wait a minute Spark driver”, say the characters of this story.
“Maybe all of us can have a discussion (as the one we had for the previous program) about the minimum amount of things that have to be done.”

Unfortunately, in this case such this discussion cannot avoid having to computing absolutely everything. Let’s see it.



Spark-driver	User Program 2

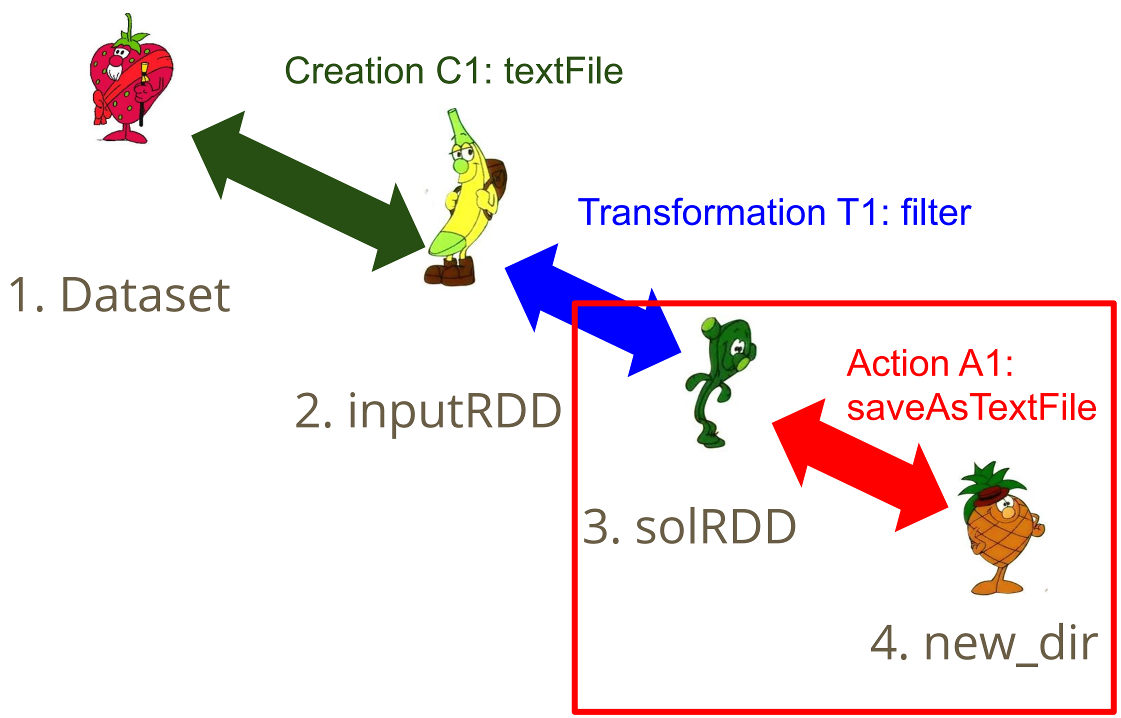
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

Let's listen to the conversation between  and 



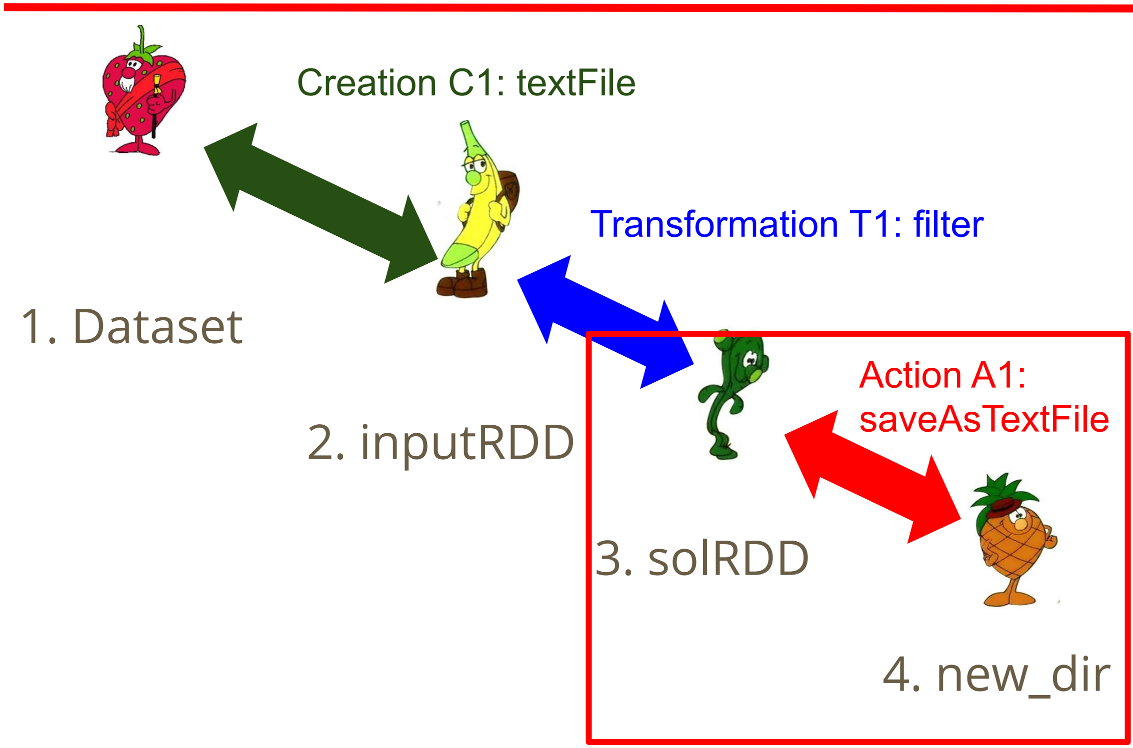
Spark-driver	User Program 2

1. inputRDD = sc.textFile(dataset).
2. solRDD = inputRDD.filter(my_func).
3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

- “solRDD, according to the operation **saveAsTextFile** we are related by, I need to store all your elements in new_dir, so I need you to be fully computed”.

- Aw, what a pity!”, replies solRDD.

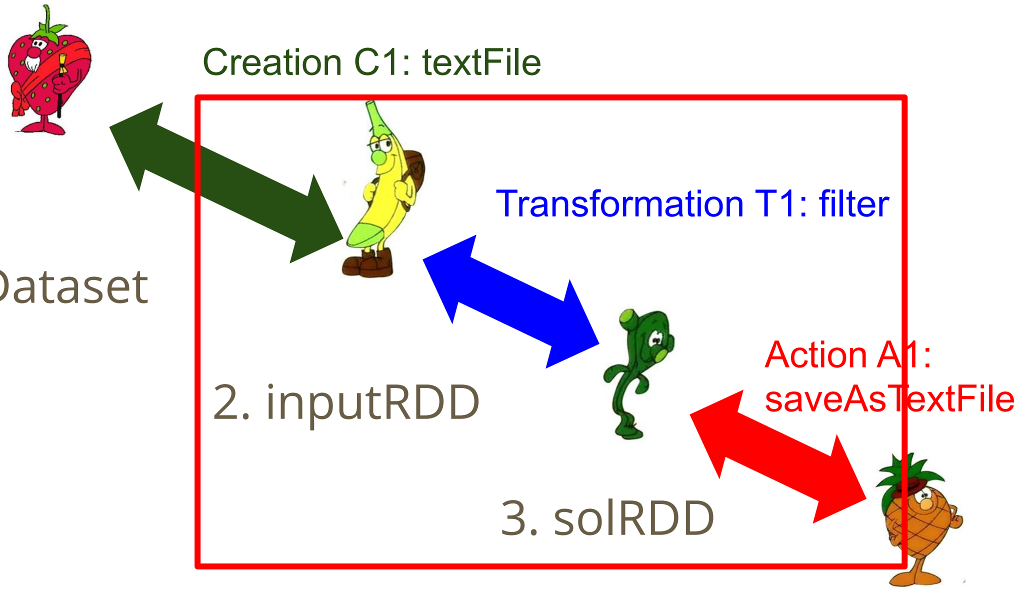


Spark-driver	User Program 2

1. inputRDD = sc.textFile(dataset).
2. solRDD = inputRDD.filter(my_func).
3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

Let's listen to the conversation between  and 



Spark-driver	User Program 2

1. inputRDD = sc.textFile(dataset).

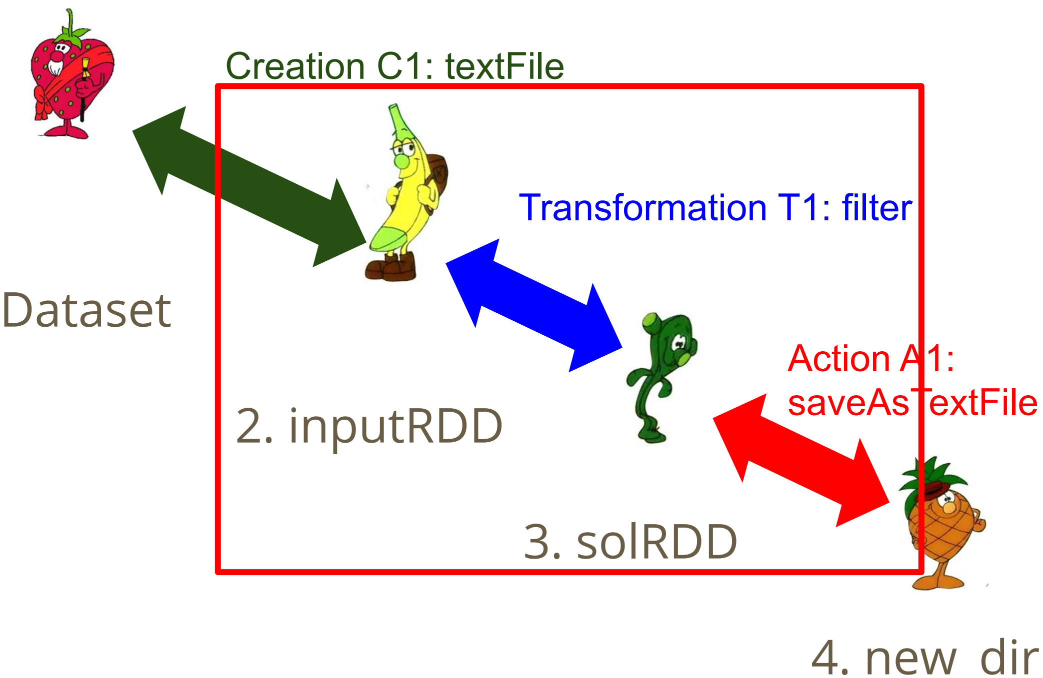
2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

- "inputRDD, according to the operation **filter** we are related by, you are supposed to be fully computed. I myself need to be fully computed, so there is no escape, you must be fully computed as well”.

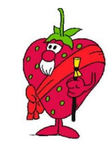
- Aw, what a pity!", replies inputRDD.

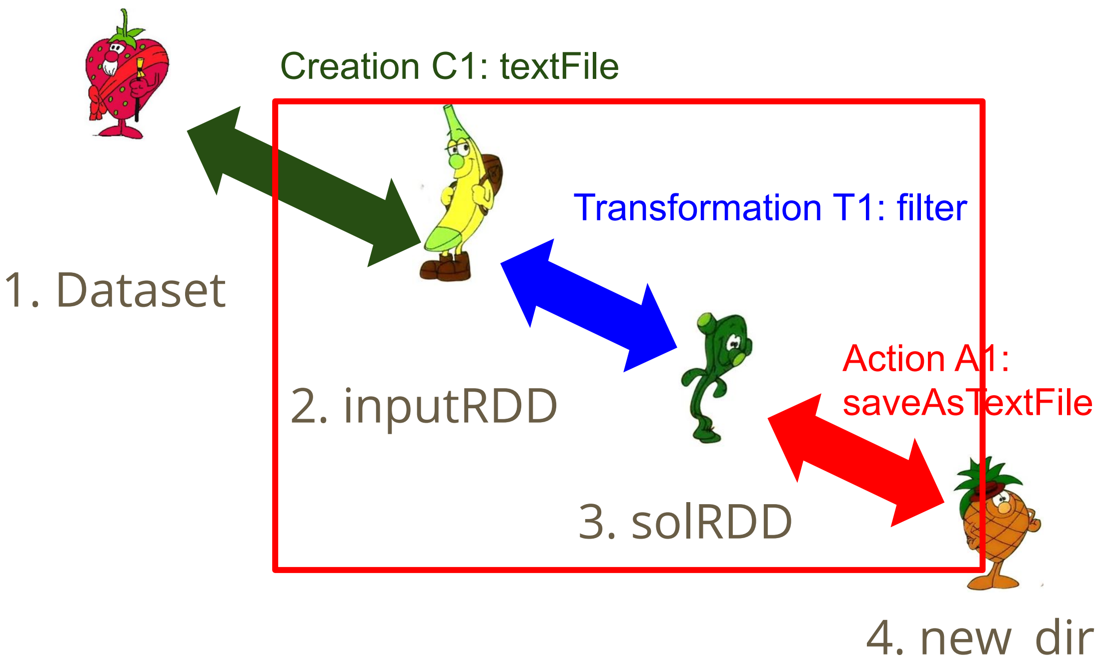


Spark-driver	User Program 2

1. inputRDD = sc.textFile(dataset).
2. solRDD = inputRDD.filter(my_func).
3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

Let's listen to the conversation between  and 



Spark-driver	User Program 2

1. inputRDD = sc.textFile(dataset).

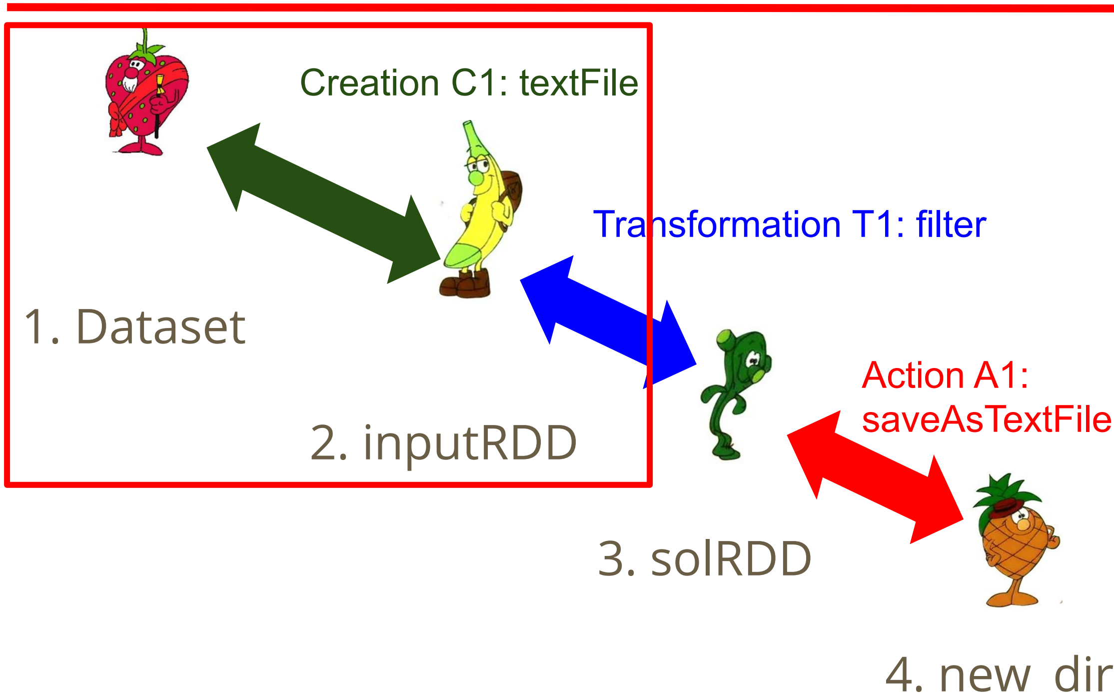
2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

- "Dataset, according to the operation **textFile** we are related by, you are supposed to be fully computed. I myself need to be fully computed, so there is no escape, you must pass me the entire content of the dataset”.

- Aw, what a pity!", replies Dataset.



Spark-driver	User Program 2

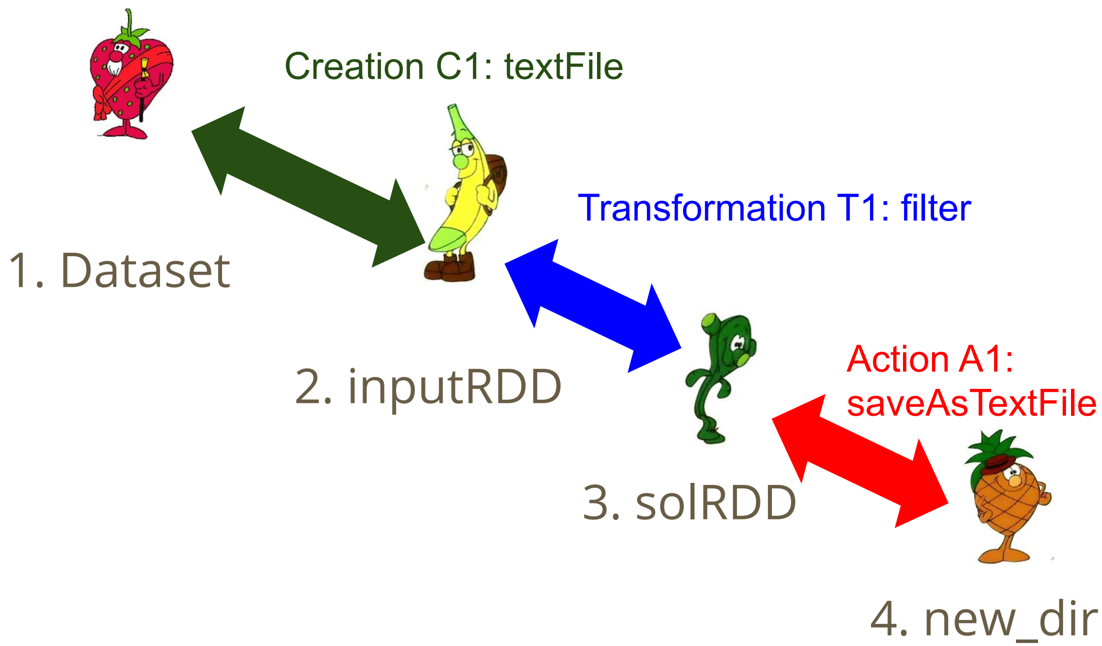
1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Lazy Evaluation

And that's why lazy evaluation cannot make the execution of this program more efficient.



Spark-driver | User Program 2

1. inputRDD = sc.textFile(dataset).

2. solRDD = inputRDD.filter(my_func).

3. solRDD.saveAsTextFile(new_dir)

Outline

1. Introduction to Apache Spark.
2. Setting the Context.
3. Prerequisites: Functional Programming.
4. An RDD is an Abstract Data Type.
5. RDD Public Side: Transformations and Actions.
6. Lazy Evaluation.

Thank you for your attention!