

Big Data Processing

— L24: Module Wrap Up —
22 Ideas from 22 Lectures

Dr. Ignacio Castineiras
Department of Computer Science

Outline

Let's summarise the main ideas presented
in our module Big Data Processing.

Outline

Idea 1:

A View for Artificial Intelligence

Idea 1: A View for Artificial Intelligence

Example Project:

Google HashCode'18 Self-driving rides



Idea 1: A View for Artificial Intelligence

Example Project:
Google HashCode'18 Self-driving rides



Idea 1: A View for Artificial Intelligence

Role of Big Data:

Put your hands into tons of data, to analyse what has happened and what is happening...

Big
Data



Machine
Learning



Combinatorial
Optimisation



Idea 1: A View for Artificial Intelligence

Role of Machine Learning:

...based on what has happened and is happening,
build models to predict what will happen...

Big
Data



Machine
Learning



Combinatorial
Optimisation

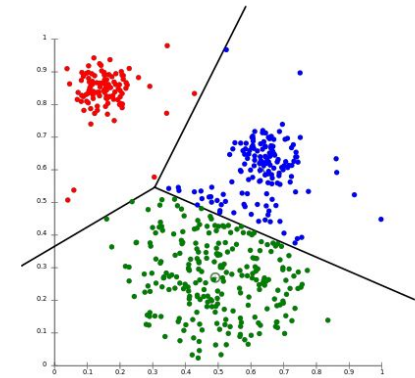
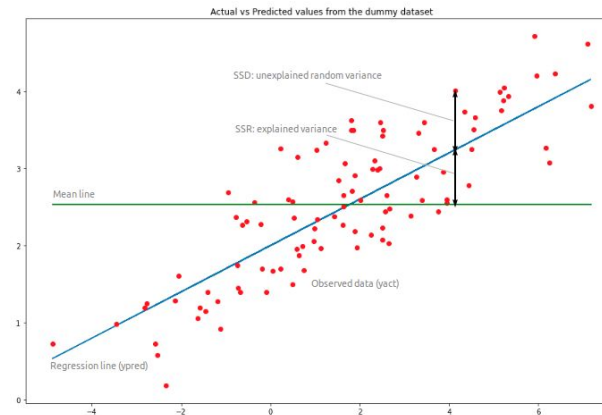


Idea 1: A View for Artificial Intelligence

Role of Machine Learning:

...based on what has happened and is happening,
build models to predict what will happen...

Machine Learning

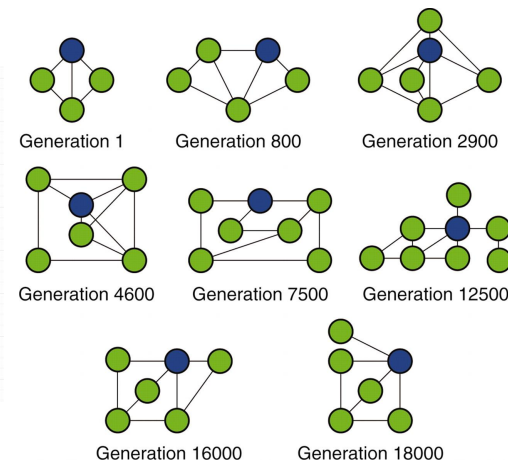
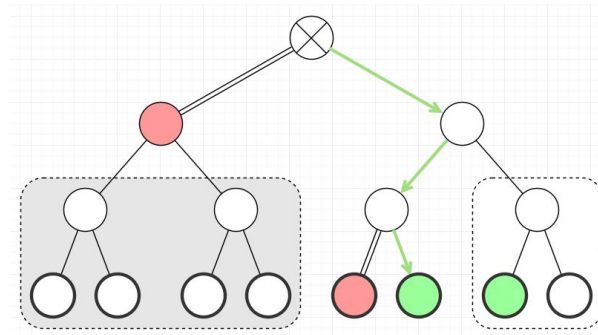


Idea 1: A View for Artificial Intelligence

Role of Combinatorial Optimisation:

...based on what we predict it will happen,
make decisions to achieve an optimal performance.

Combinatorial Optimisation



Outline

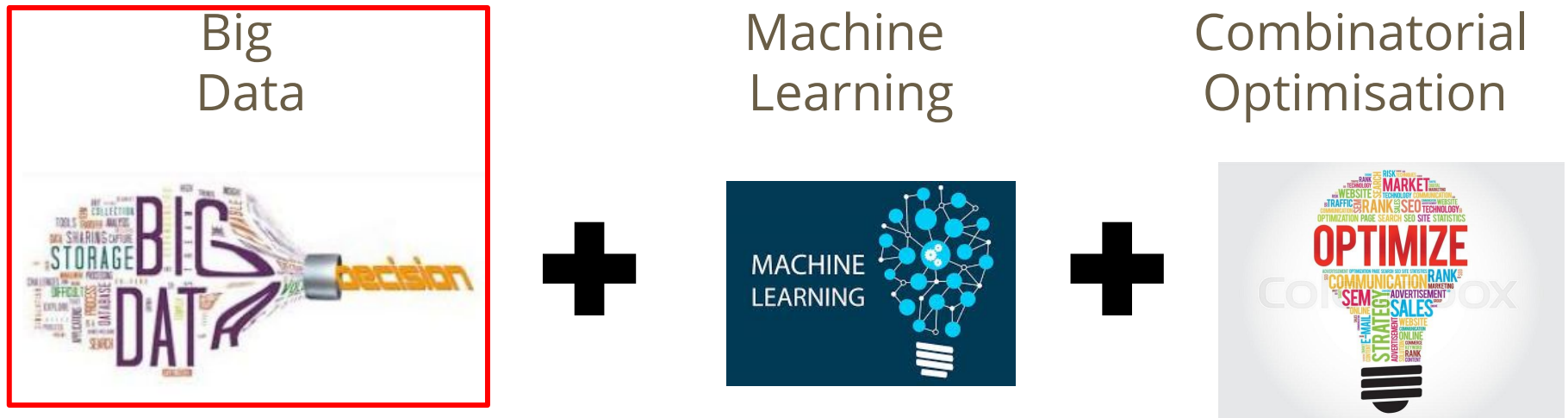
Idea 2:

A View for Big Data Processing

Idea 2: A View for Big Data Processing

Based on our view of Artificial Intelligence...

Studying Big Data Processing is crucial,
as it will be the start of the journey
for whatever problem you take!



Outline

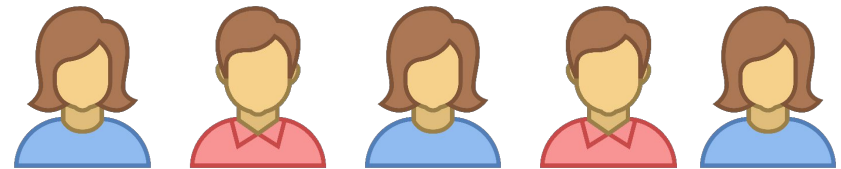
Idea 3:

A View for Interesting Problems

Idea 3: A View for Interesting Problems

Interesting problems
might involve a complex formulation...

Google HashCode'18 Self-driving rides



Idea 3: A View for Interesting Problems

Interesting problems
...might involve a simple formulation

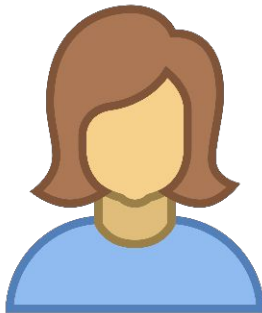
List Inversions

	1	2	3	4	5	6
A	1	3	5	2	4	6

Idea 3: A View for Interesting Problems

Interesting problems
...might involve a simple formulation

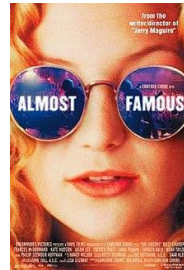
P1



1



2



3



4



5



6



1



2



3



4



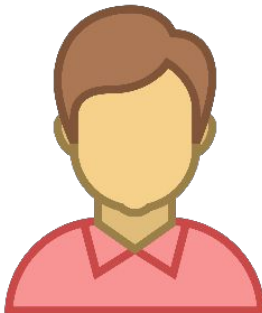
5



6



P2



Idea 3: A View for Interesting Problems

But, whether simple or complex formulated,
interesting problems are truly interesting...

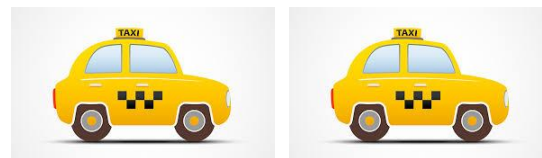
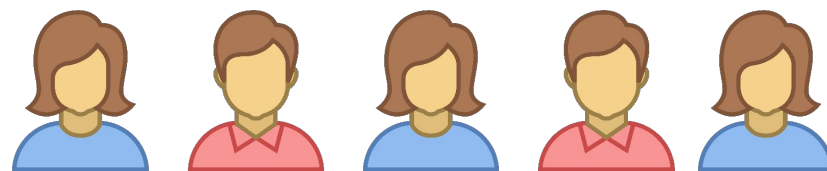
Idea 3: A View for Interesting Problems

But, whether simple or complex formulated,
interesting problems are truly interesting...
at scale! (when computational challenging)

Idea 3: A View for Interesting Problems

But, whether simple or complex formulated, interesting problems are truly interesting...

*Google HashCode'18
Self-driving rides*



Idea 3: A View for Interesting Problems

But, whether simple or complex formulated,
interesting problems are truly interesting...
at scale! (when computational challenging)

Google HashCode'18
Self-driving rides

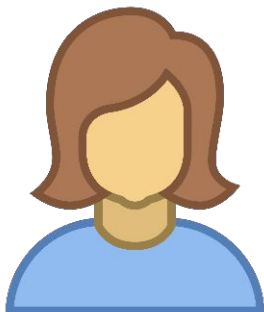


Idea 3: A View for Interesting Problems

But, whether simple or complex formulated, interesting problems are truly interesting...

List Inversions

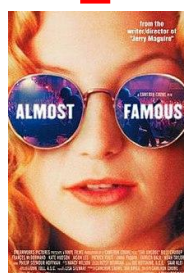
P1



1



2



3



4



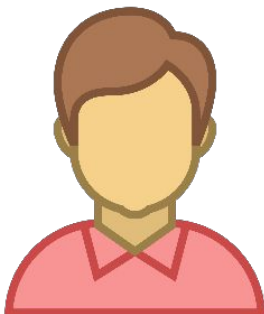
5



6



P2



1



2



3



4



5



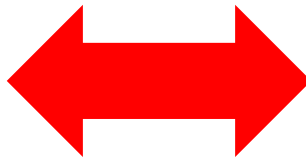
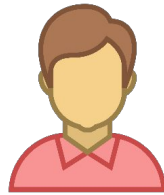
6



Idea 3: A View for Interesting Problems

But, whether simple or complex formulated,
interesting problems are truly interesting...
at scale! (when computational challenging)

List Inversions



A1

1	2	3	4	...	n
1	3	5	2	..	6



Ak

1	2	3	4	...	n
5	9	4	1	..	2

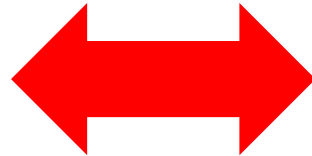
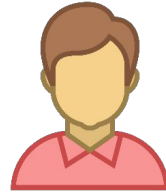


Outline

Idea 4:

Scalability Tackled Via
Distributed Programming

Idea 4: Scalability Tackled Via Distributed Programming



The problem workload can be balanced among multiple cores.



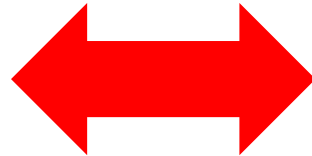
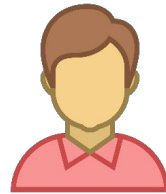
Idea 4: Scalability Tackled Via Distributed Programming



The problem workload can be balanced among multiple cores.



Idea 4: Scalability Tackled Via Distributed Programming

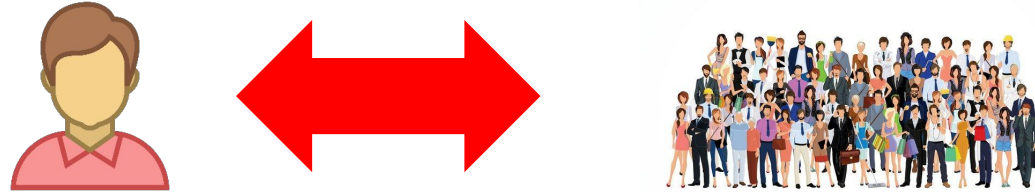


The problem workload can be balanced among multiple cores.



Shaunae Miller (400m)

Idea 4: Scalability Tackled Via Distributed Programming

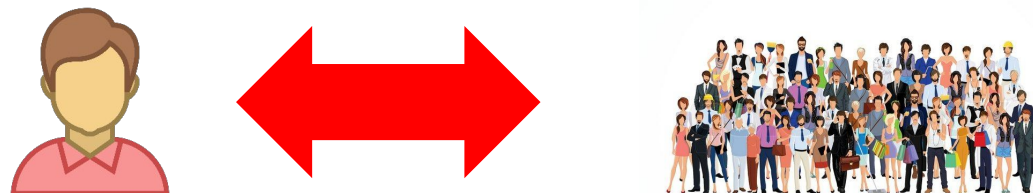


The problem workload can be balanced among multiple cores.



Elaine Thompson (100m) Tori Bowie (100m) Shelly-Ann Fraser-Pryce (100m) Marie-Josée Ta Lou (100m)

Idea 4: Scalability Tackled Via Distributed Programming



The problem workload can be balanced among multiple cores.

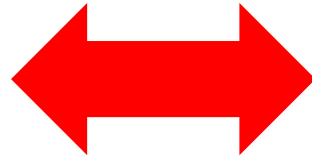
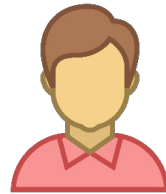


vs.



- **TOTAL COMPUTATION TIME:** Same in both approaches.
- **TOTAL ELAPSED TIME:** Distributed approach is $\frac{1}{4}$ of classical single-thread sequential approach.

Idea 4: Scalability Tackled Via Distributed Programming



The elapsed time improvement is determined by the number of cores (size of the cluster)



Outline

Idea 5:

Distributed Programming
Centralised Coordination

Idea 5: Distributed Programming Centralised Coordination

Passing from sequential to distributed mode requires the use of a meta-algorithm, in charge of controlling the distributed execution.

*Meta-algorithm core
coordinating that
the job is done*

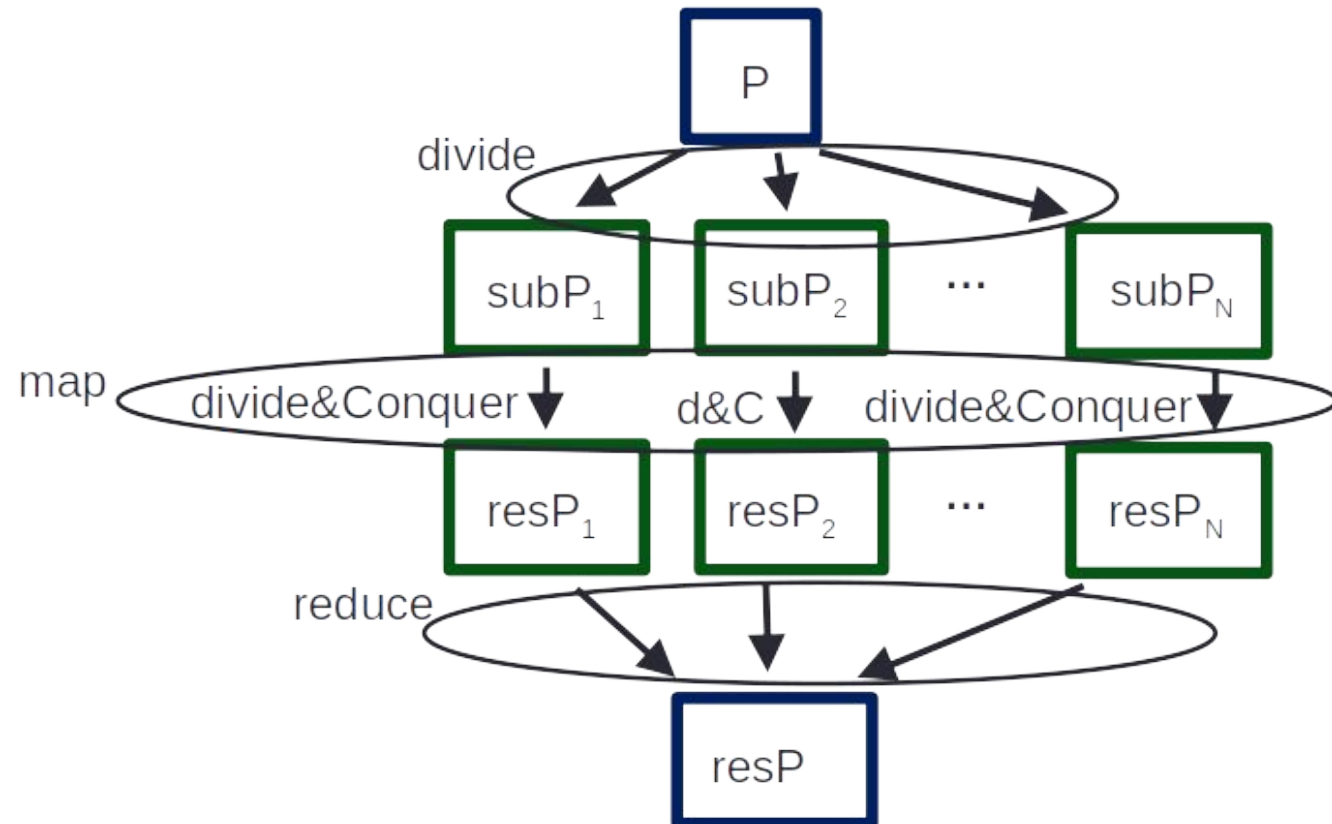


*Working cores
actually doing the job*



Idea 5: Distributed Programming Centralised Coordination

This meta-algorithm follows the **divide-map-reduce** approach of the classical Divide and Conquer programming paradigm!

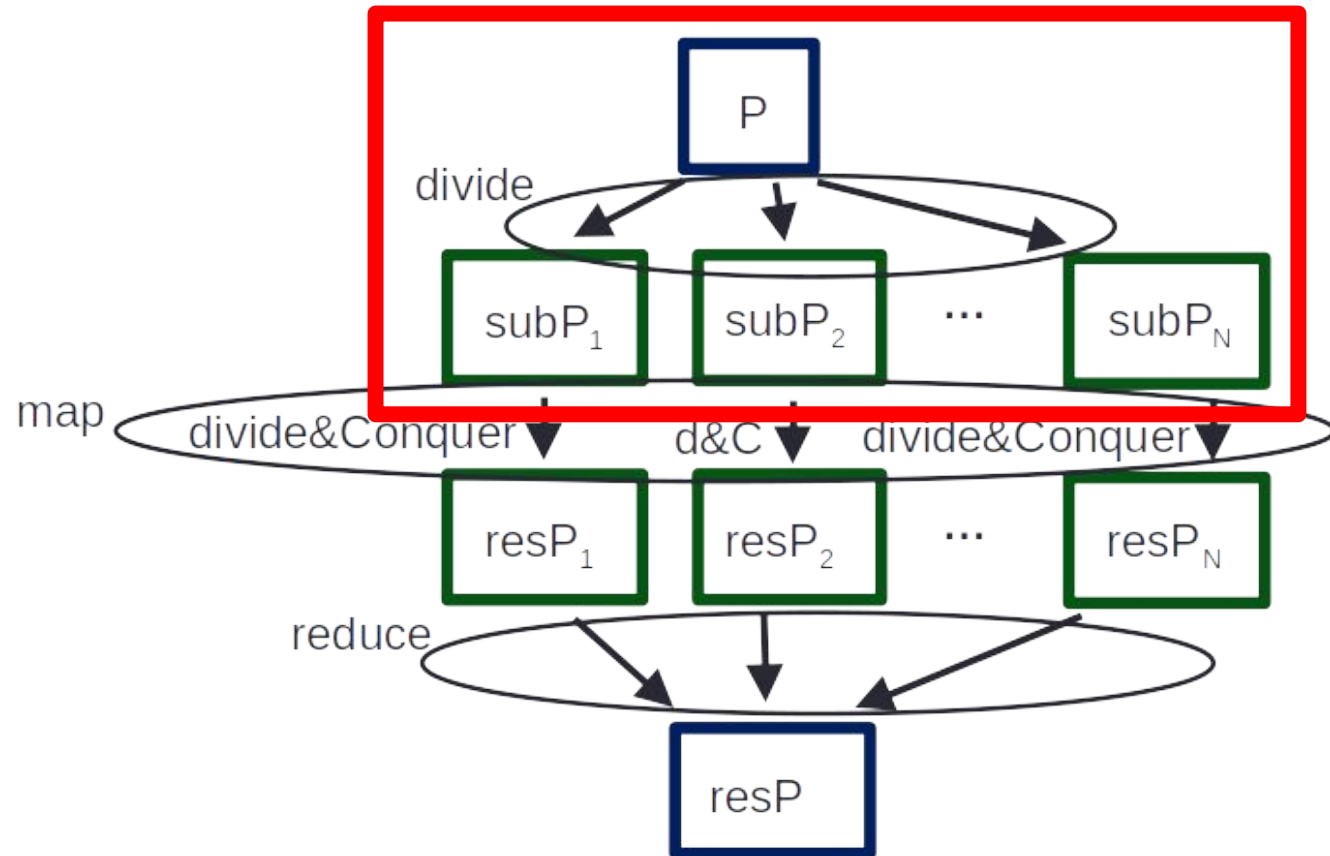


Idea 5: Distributed Programming Centralised Coordination

This meta-algorithm follows the **divide-map-reduce** approach of the classical Divide and Conquer programming paradigm!

1. DIVIDE STAGE.

It divides the original problem by splitting it into n sub-parts, to be assigned to the working cores.

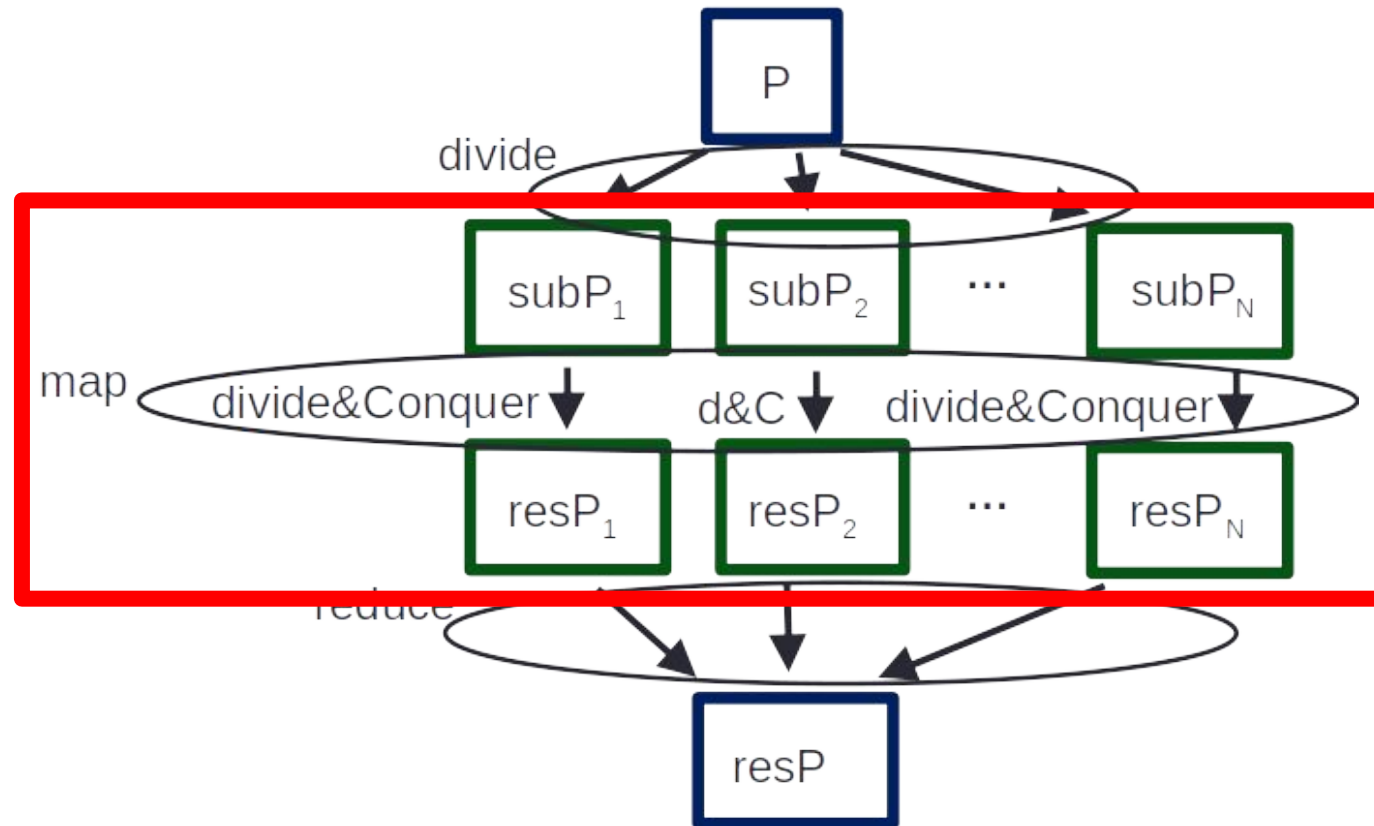


Idea 5: Distributed Programming Centralised Coordination

This meta-algorithm follows the **divide-map-reduce** approach of the classical Divide and Conquer programming paradigm!

2. MAP STAGE

Each working core solves its sub-part separately.

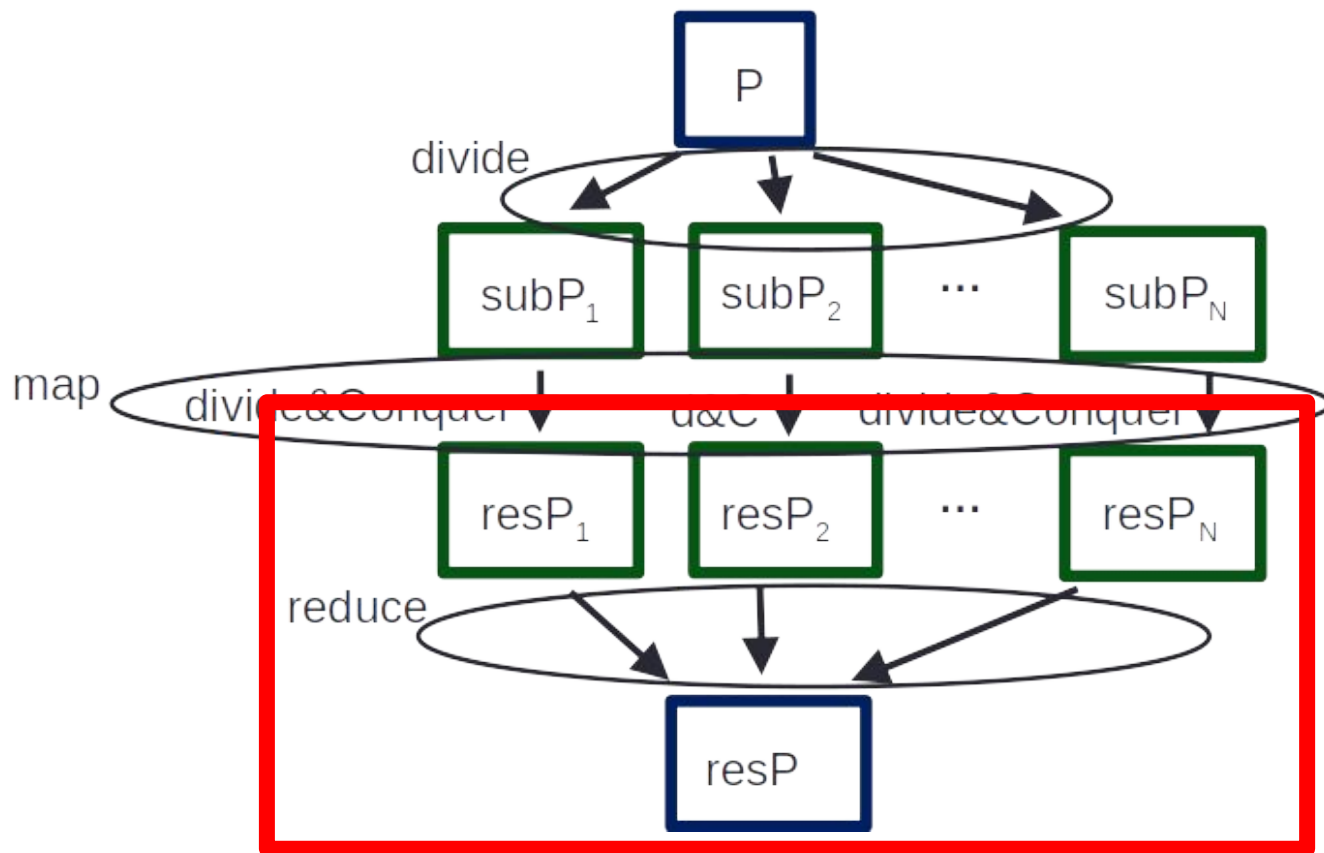


Idea 5: Distributed Programming Centralised Coordination

This meta-algorithm follows the **divide-map-reduce** approach of the classical Divide and Conquer programming paradigm!

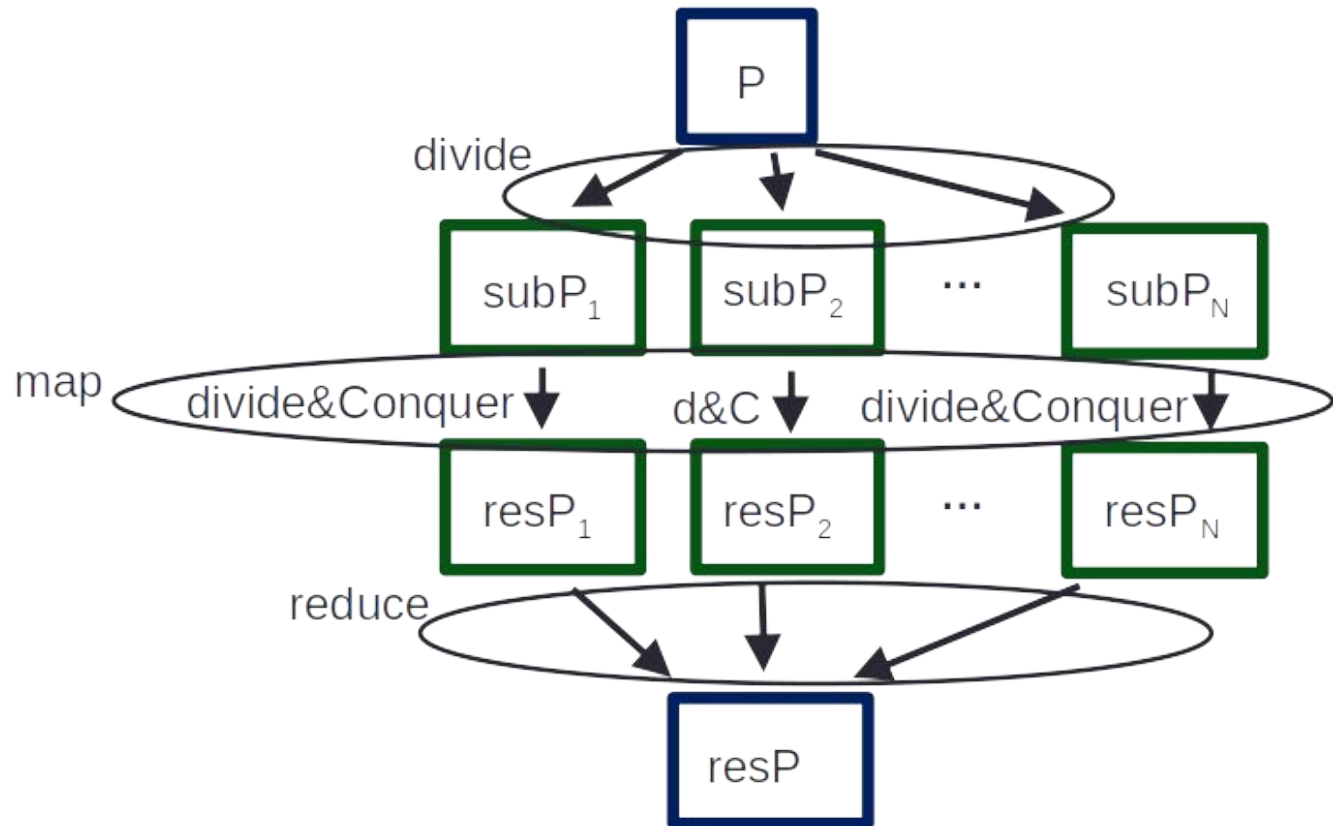
3. DIVIDE STAGE

The sub-part solutions found by the working cores are combined (*this additional combination work is also done by working cores*) so as to get the solution to the original problem.



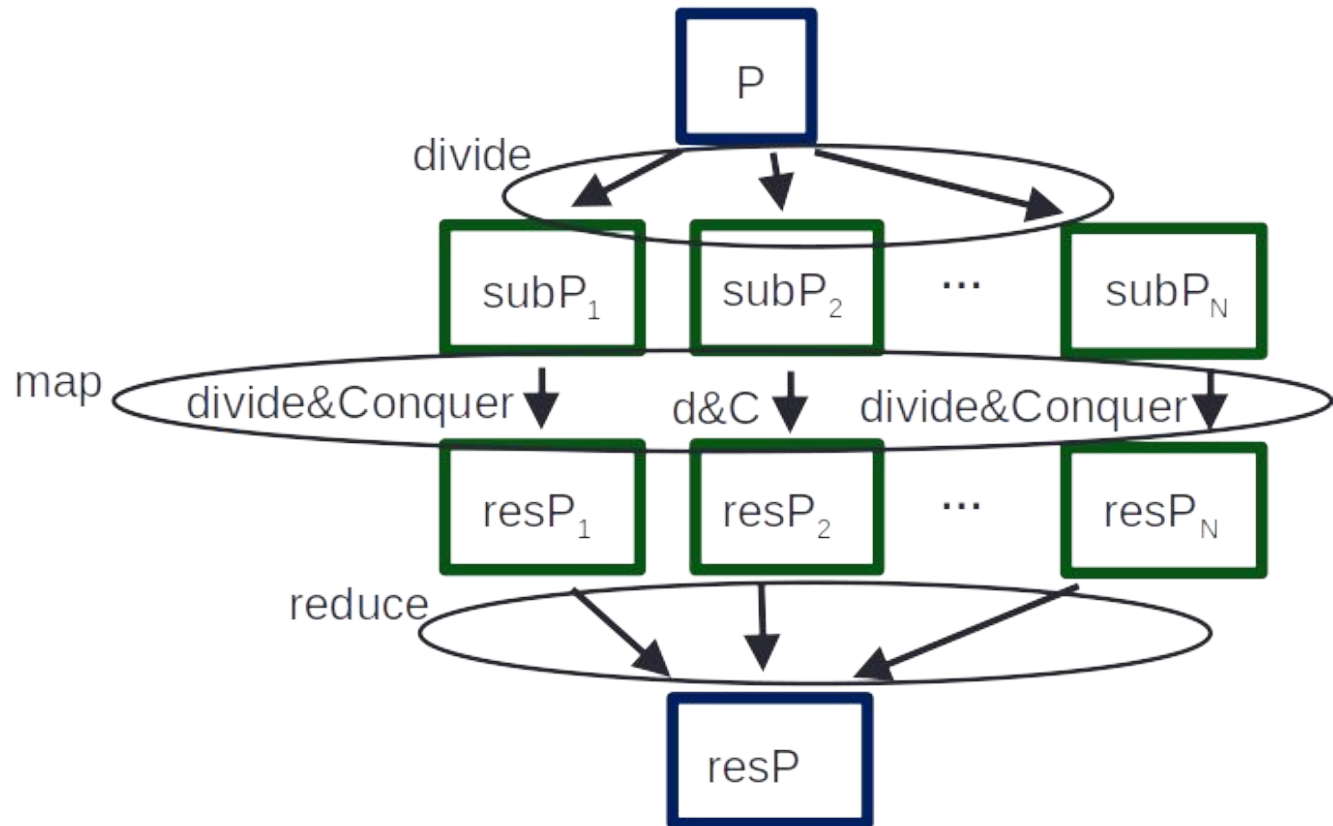
Idea 5: Distributed Programming Centralised Coordination

Now, programming this meta-algorithm is difficult.



Idea 5: Distributed Programming Centralised Coordination

Moreover, the meta-algorithm is problem domain-specific (it might not necessarily be reusable for a different problem).



Outline

Idea 6:

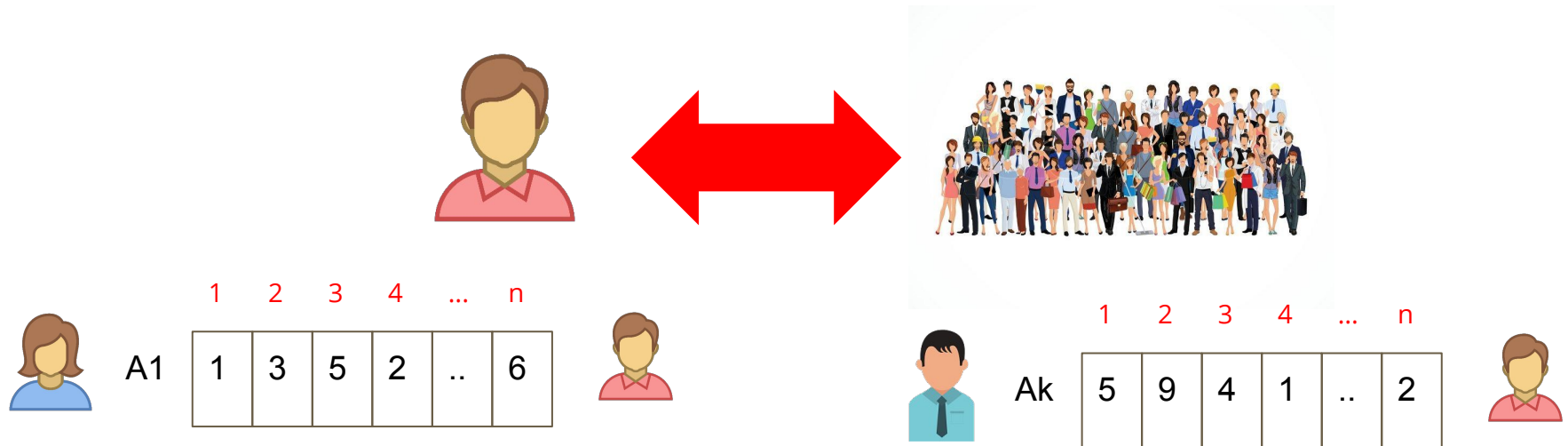
Scalability Tackled Via
A More Efficient Algorithm

Idea 6: Scalability Tackled Via A More Efficient Algorithm

As computer scientists we always
have to challenge our algorithms:

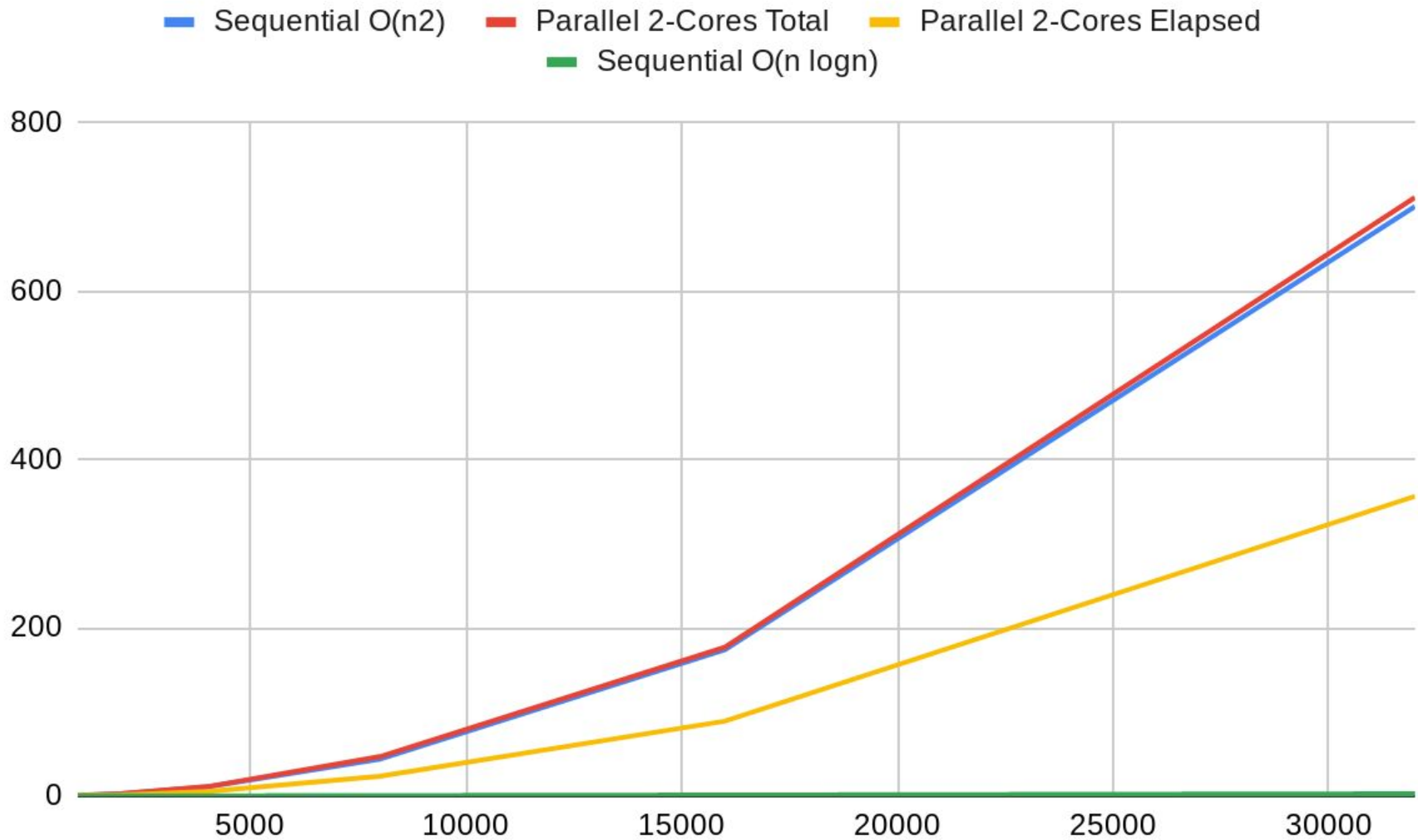
Can we do better?

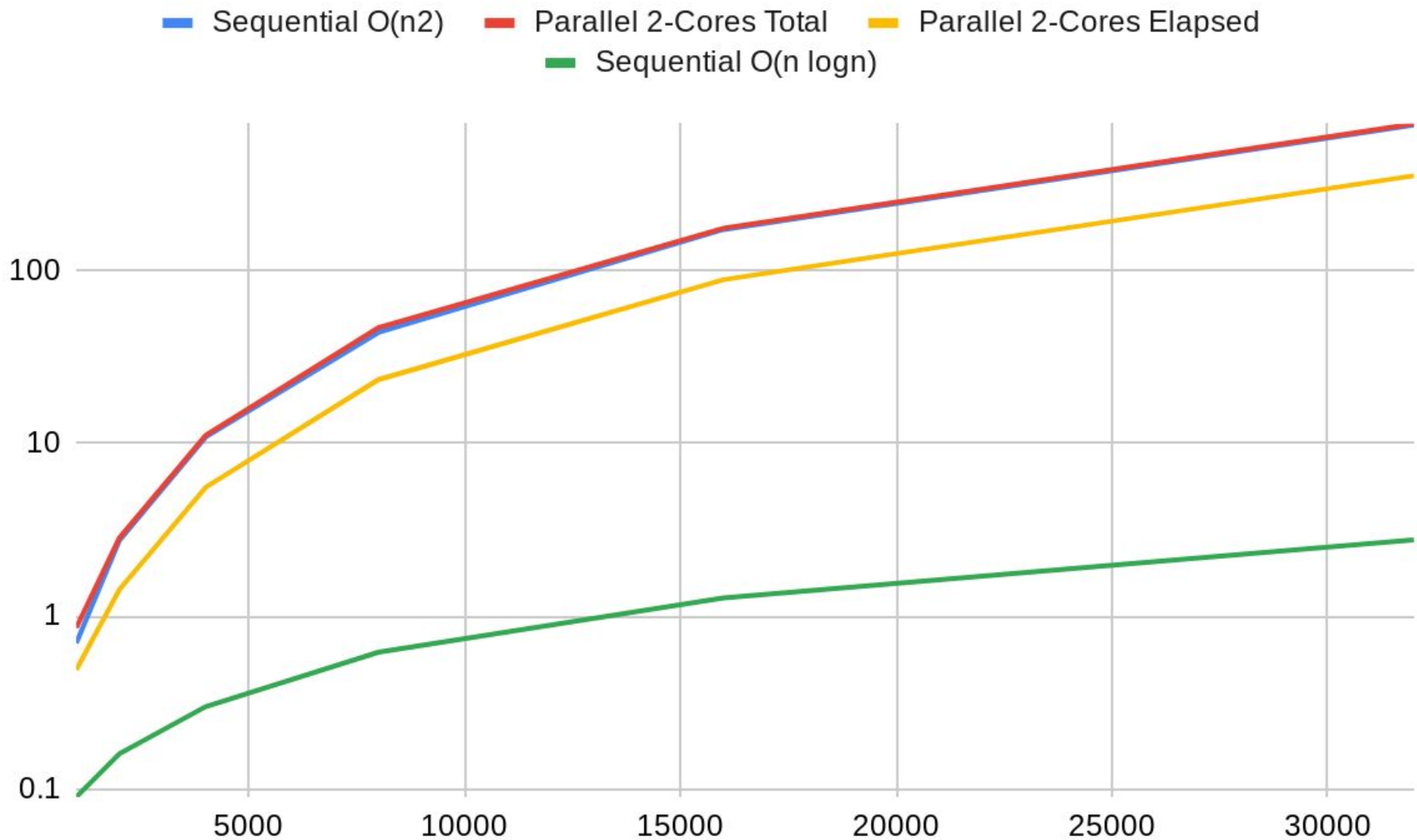
Idea 6: Scalability Tackled Via A More Efficient Algorithm



When evaluating our List Inversions problem **at scale**...

- Our MergeSort-based algorithm with complexity $O(n \log n)$ improved the sequential one with $O(n^2)$ by orders of magnitude.





These are the results in logarithmic scale, as it is the only way to see the green line

Idea 6: Scalability Tackled Via A More Efficient Algorithm

The performance of one core using the $O(n \log n)$ algorithm was comparable to the performance of hundreds of cores working together using the $O(n^2)$ algorithm.

$O(n \log n)$



$O(n^2)$



Idea 6: Scalability Tackled Via A More Efficient Algorithm

So, all in all...

Yes! Distributed Programming is indeed the way to tackle
Big Data Processing problems.

Idea 6: Scalability Tackled Via A More Efficient Algorithm

So, all in all...

Yes! Distributed Programming is indeed the way to tackle
Big Data Processing problems.

However, we still must challenge our algorithms
for them to be as efficient as possible.

Idea 6: Scalability Tackled Via A More Efficient Algorithm

So, all in all...

Yes! Distributed Programming is indeed the way to tackle
Big Data Processing problems.

However, we still must challenge our algorithms
for them to be as efficient as possible.

In many cases,
the performance improvement we get from these better algorithms
outperforms the improvement achieved by parallelization itself.

Outline

Idea 7:

Distributed Programming
with Apache Spark

Idea 7: Distributed Programming with Apache Spark

In this module we have used
Apache Spark!

The state-of-the-art technology
for Big Data Processing
(based on distributed programming)

Idea 7: Distributed Programming with Apache Spark

Apache Spark is an



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

Idea 7: Distributed Programming with Apache Spark

Apache Spark is an



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

It has been designed to fit 3 purposes:

Be easy to use.

Be fast.

Be general.

Idea 7: Distributed Programming with Apache Spark



Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM).



Spark offers simple APIs the proper Scala, and also in Java, Python and R.



Idea 7: Distributed Programming with Apache Spark



Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM).

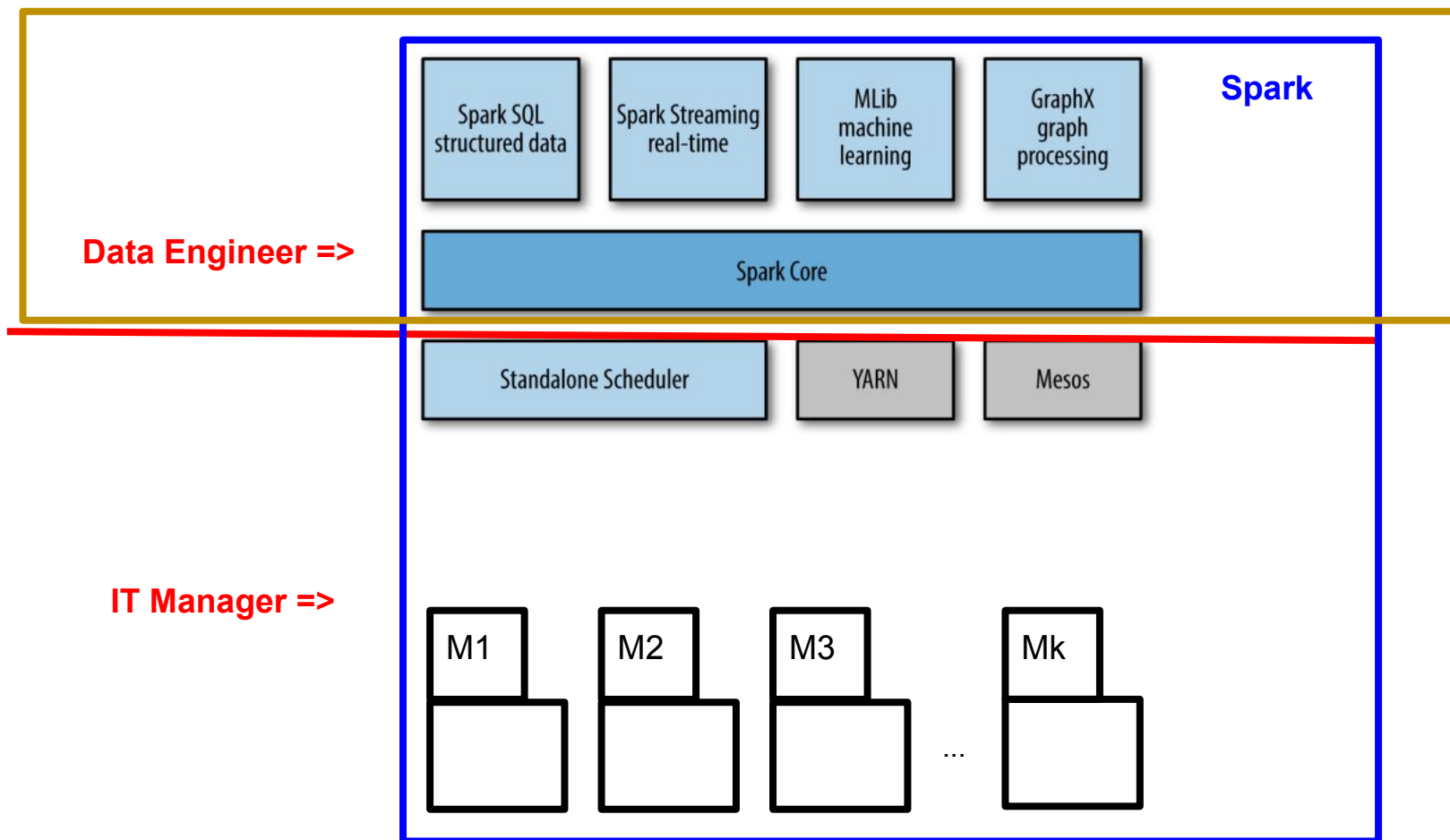


Spark offers simple APIs the proper Scala, and also in Java, Python and R.
In our module we have focused mainly in Python (and sometimes in Scala).



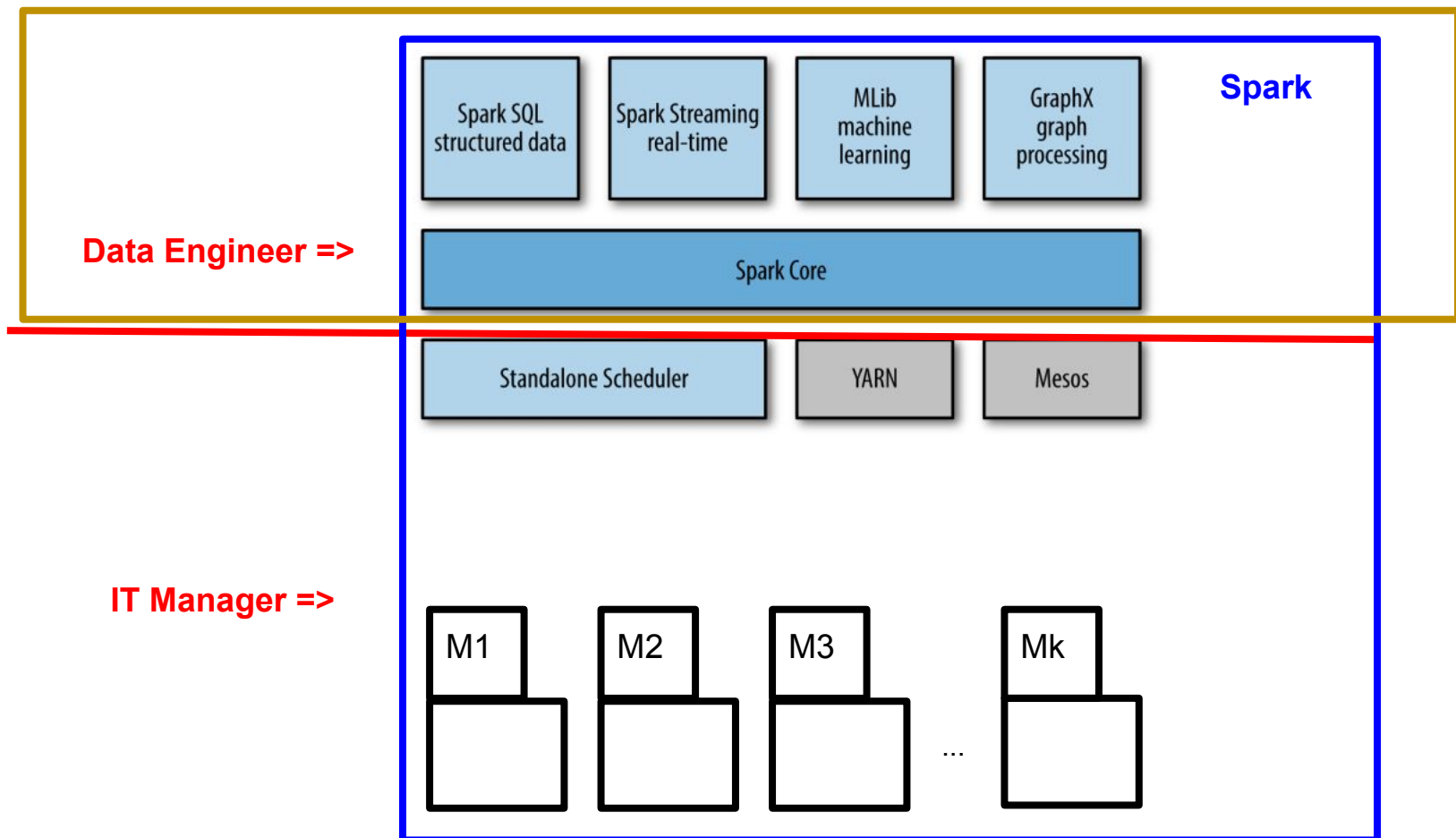
Idea 7: Distributed Programming with Apache Spark

We have focused in the role of Data Engineer, who uses Spark on top of a “ready” cluster.



Idea 7: Distributed Programming with Apache Spark

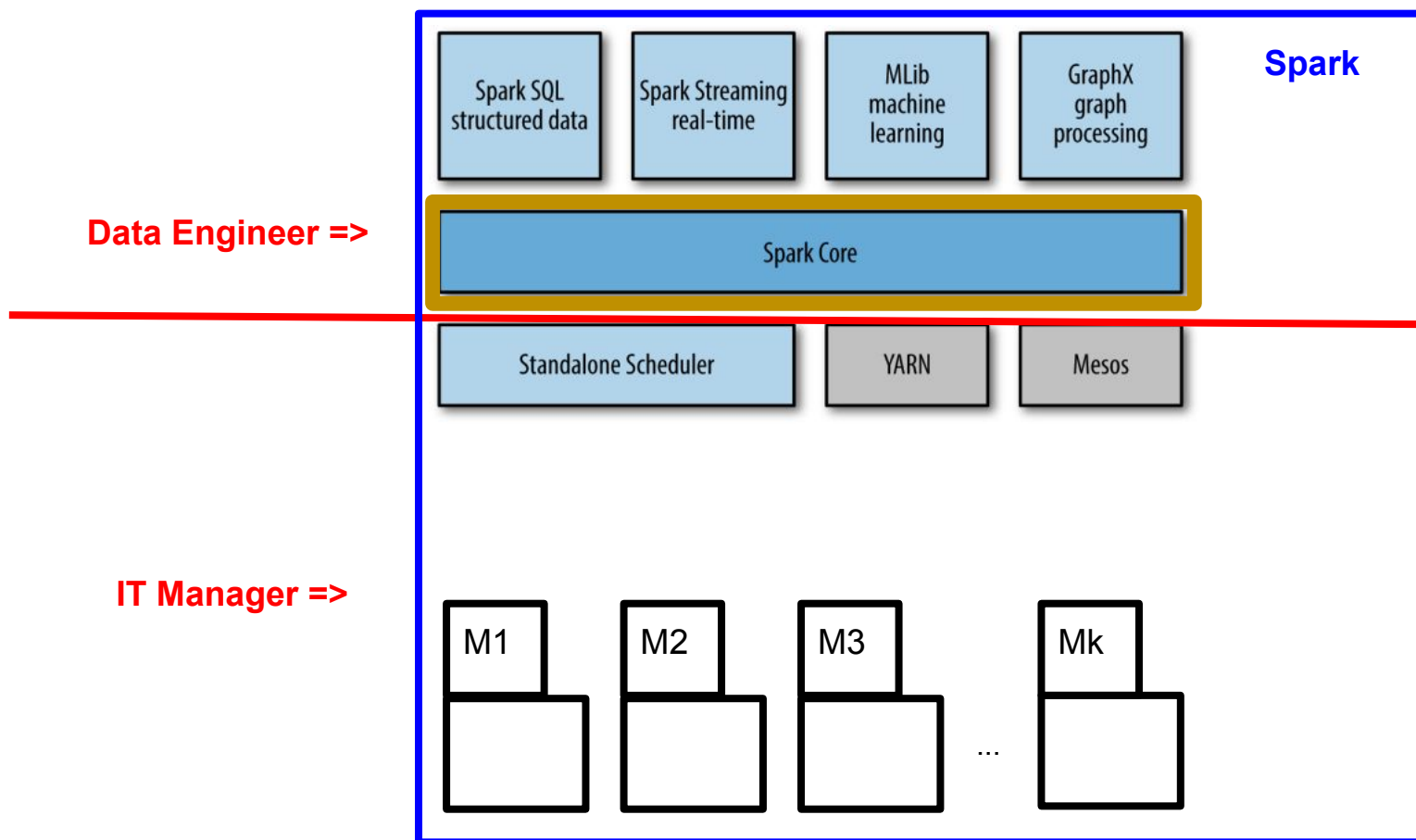
Among the different components for the Data Engineer role, we have focus on:



Idea 7: Distributed Programming with Apache Spark

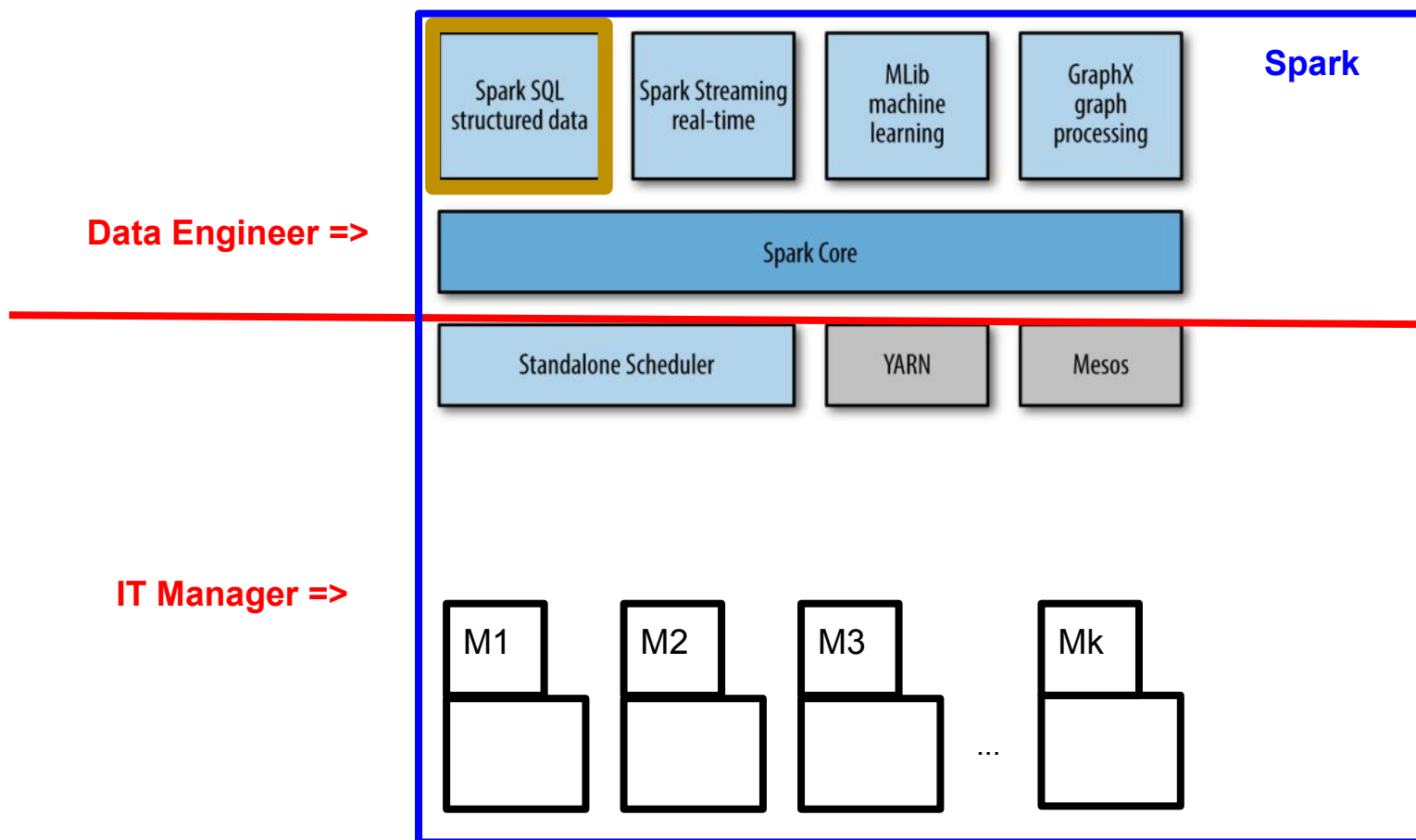
Among the different components for the Data Engineer role, we have focus on:

Spark Core



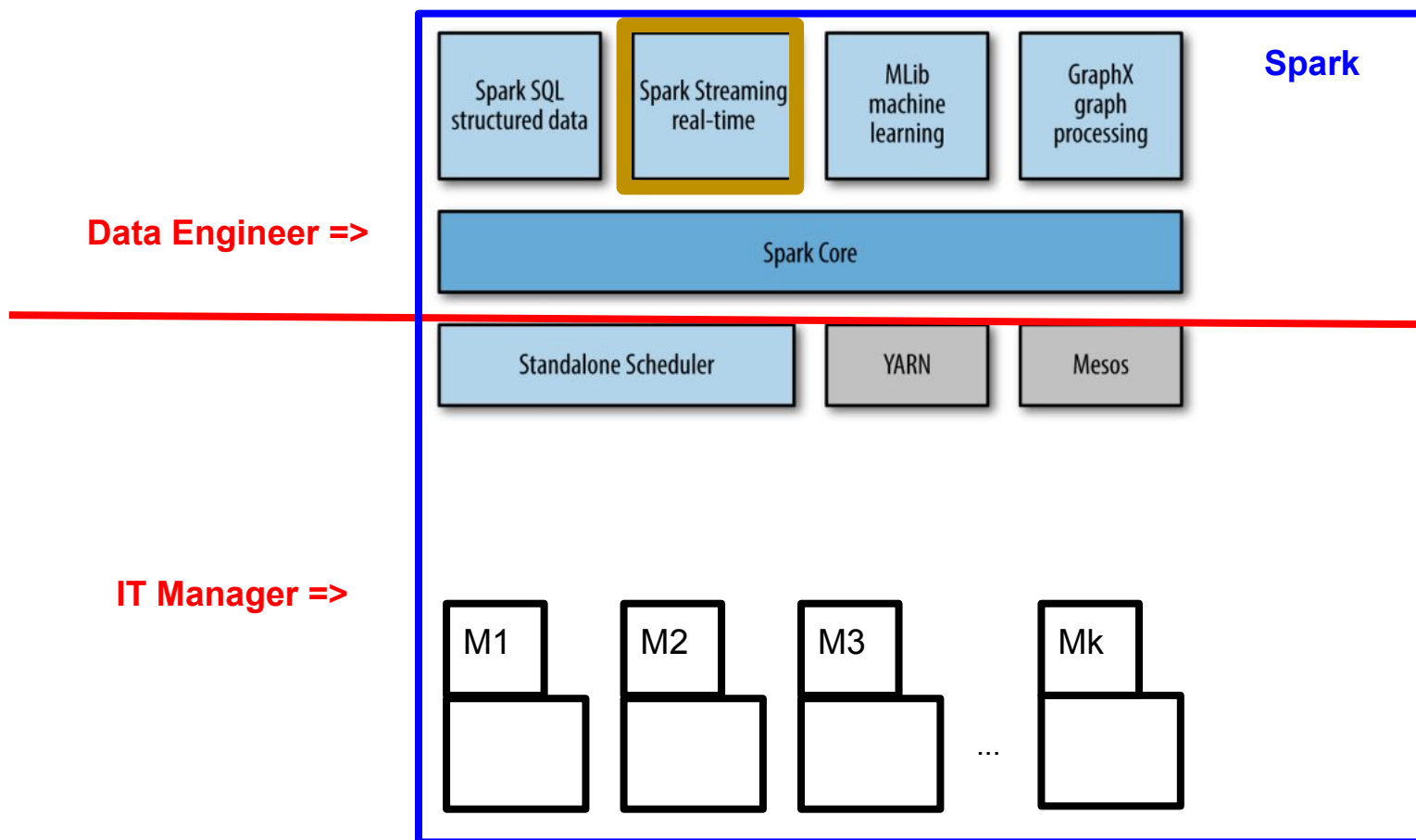
Idea 7: Distributed Programming with Apache Spark

Among the different components for the Data Engineer role, we have focus on:
Spark SQL



Idea 7: Distributed Programming with Apache Spark

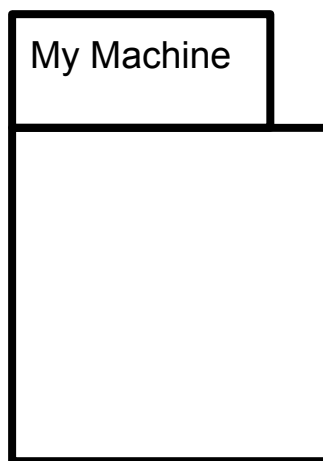
Among the different components for the Data Engineer role, we have focus on:
The two libraries for extending Spark with streaming capabilities



Idea 7: Distributed Programming with Apache Spark

For running Spark applications we have used:

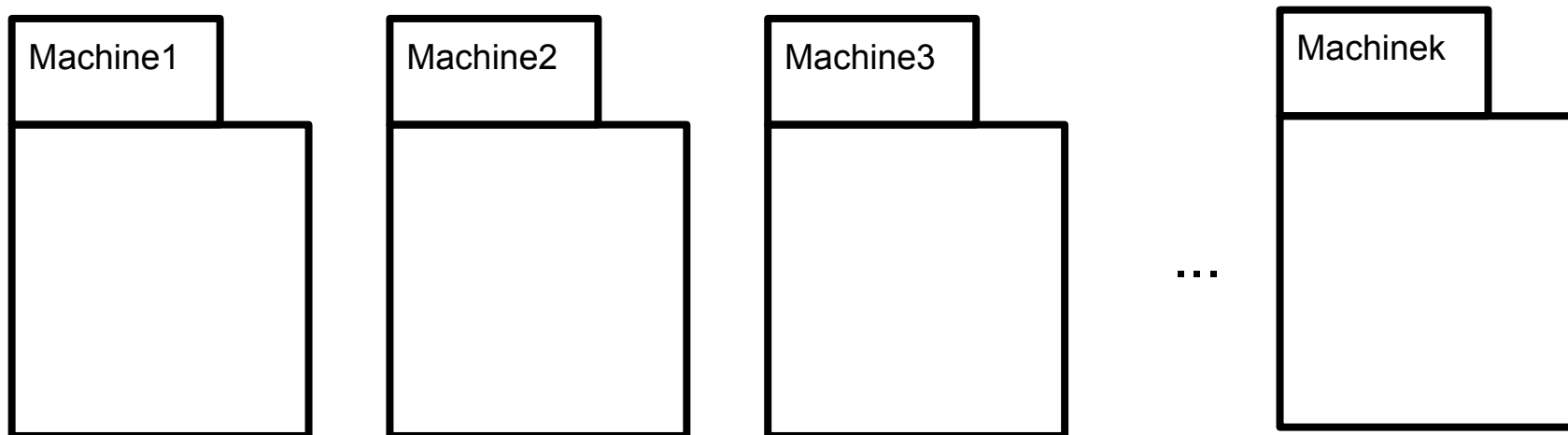
1. Local Mode (using our own local machine)



Idea 7: Distributed Programming with Apache Spark

For running Spark applications we have used:

1. Local Mode (using our own local machine).
2. Databricks: An online cluster-based platform.



Outline

Idea 8:

Databricks - Big Data Made Simple

Idea 8: Databricks - Big Data Made Simple



- Databricks is a spin-off company leveraging Spark as a **cloud-based** big data processing tool <https://databricks.com/>

Idea 8: Databricks - Big Data Made Simple



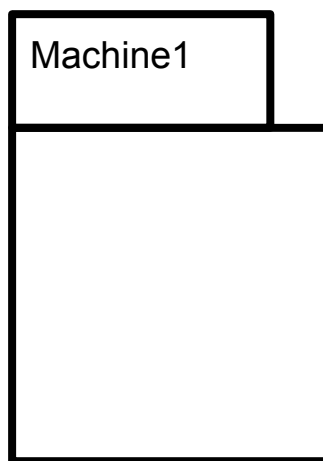
- Databricks is a spin-off company leveraging Spark as a **cloud-based** big data processing tool <https://databricks.com/>
- It abstracts the IT management tasks by allowing to set up and configure a cluster with just a few clicks.
It provides an easy interface for performing Data Engineering tasks (such as import, edit and running Spark applications) on top of the cluster being created.

Idea 8: Databricks - Big Data Made Simple



databricks

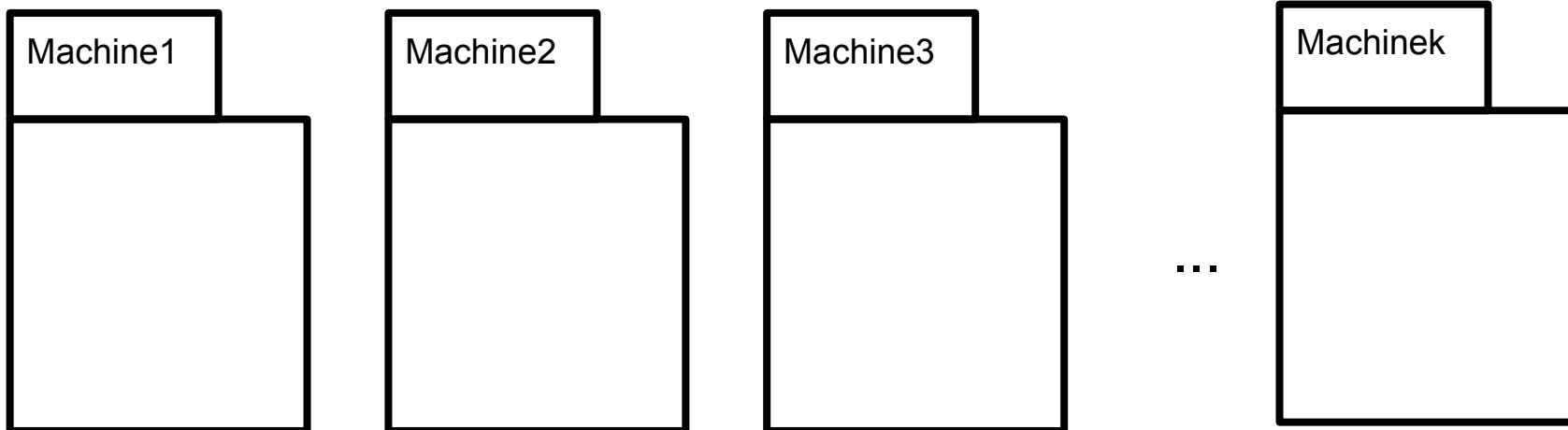
- The Databricks Community Edition is free to use. It provides us with an equivalent to **local mode** Spark configuration:
 - 1 physical machine with
 - 2 cores and
 - 8GB of memory storage.



Idea 8: Databricks - Big Data Made Simple



- If needed, the Databricks Enterprise Edition would allow us to rent a cluster in a price-per-usage basis, and configure it to target the Business Unit specific needs.



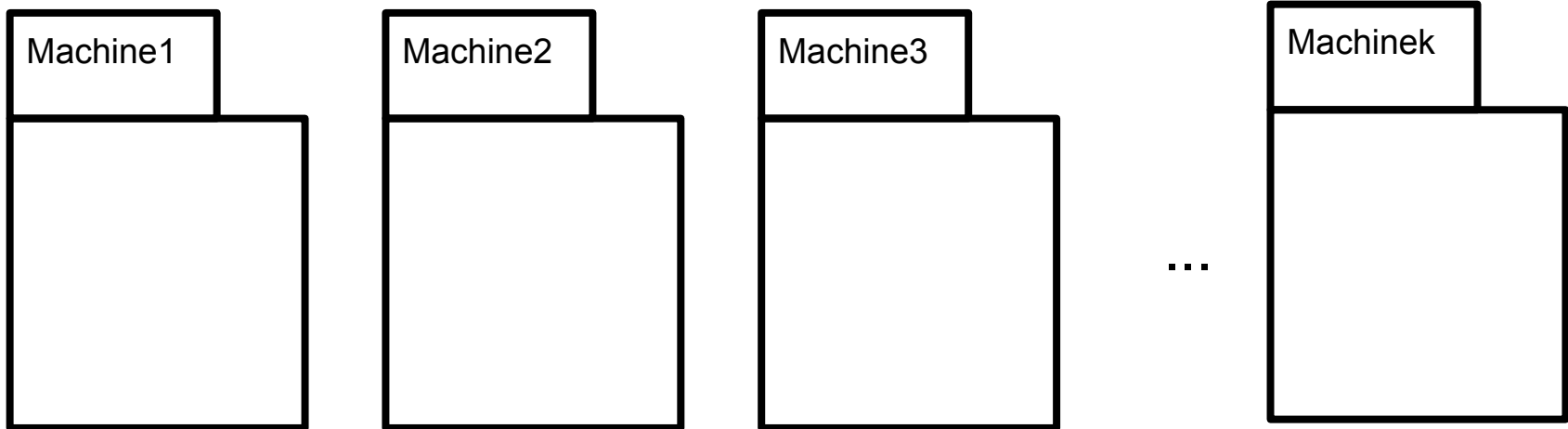
Outline

Idea 9:

Spark Recipe
Set of Ingredients

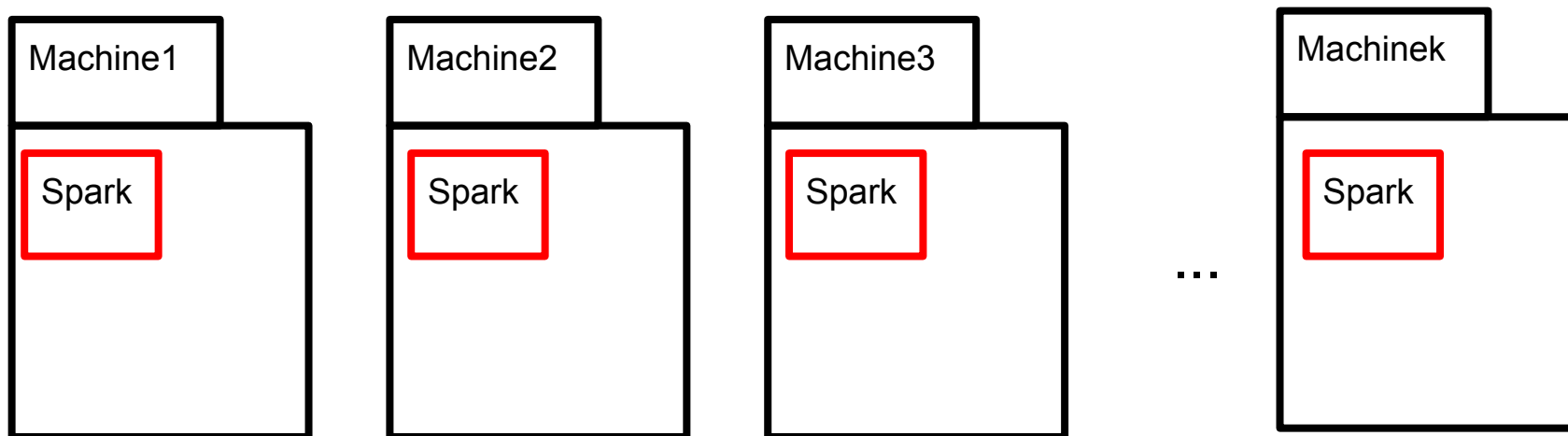
Idea 9: Spark Recipe - Set of Ingredients

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



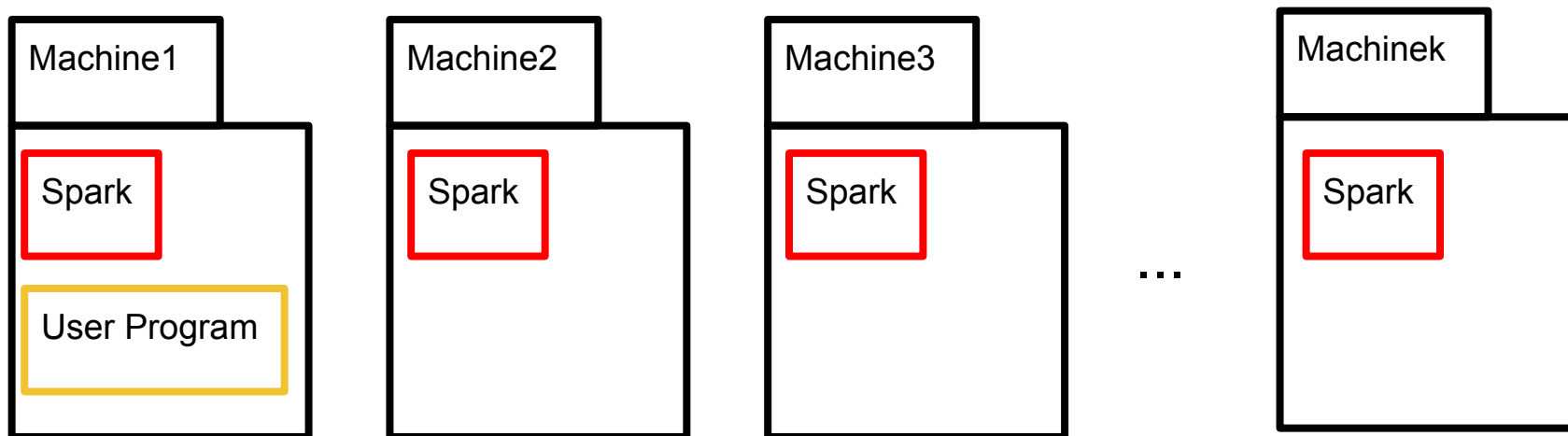
Idea 9: Spark Recipe - Set of Ingredients

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



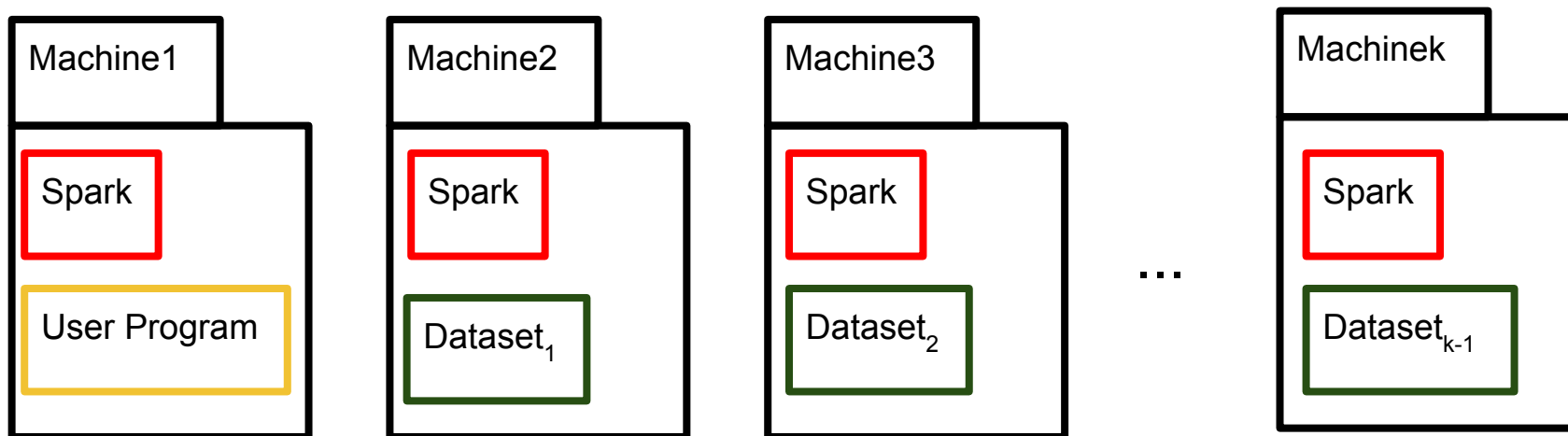
Idea 9: Spark Recipe - Set of Ingredients

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



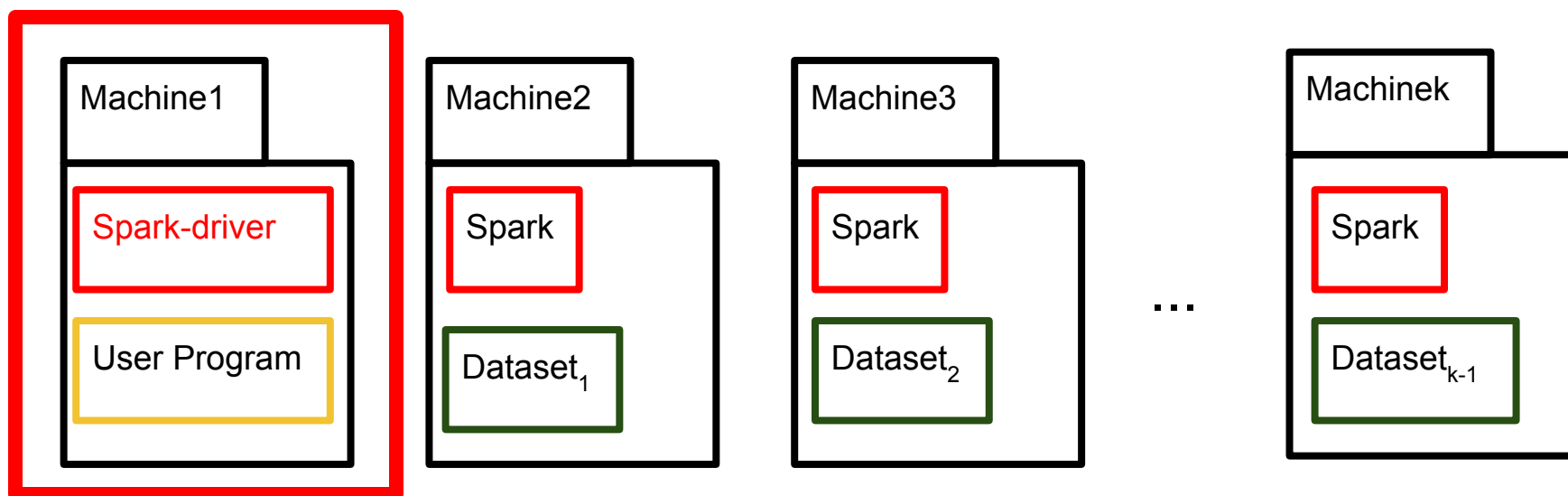
Idea 9: Spark Recipe - Set of Ingredients

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



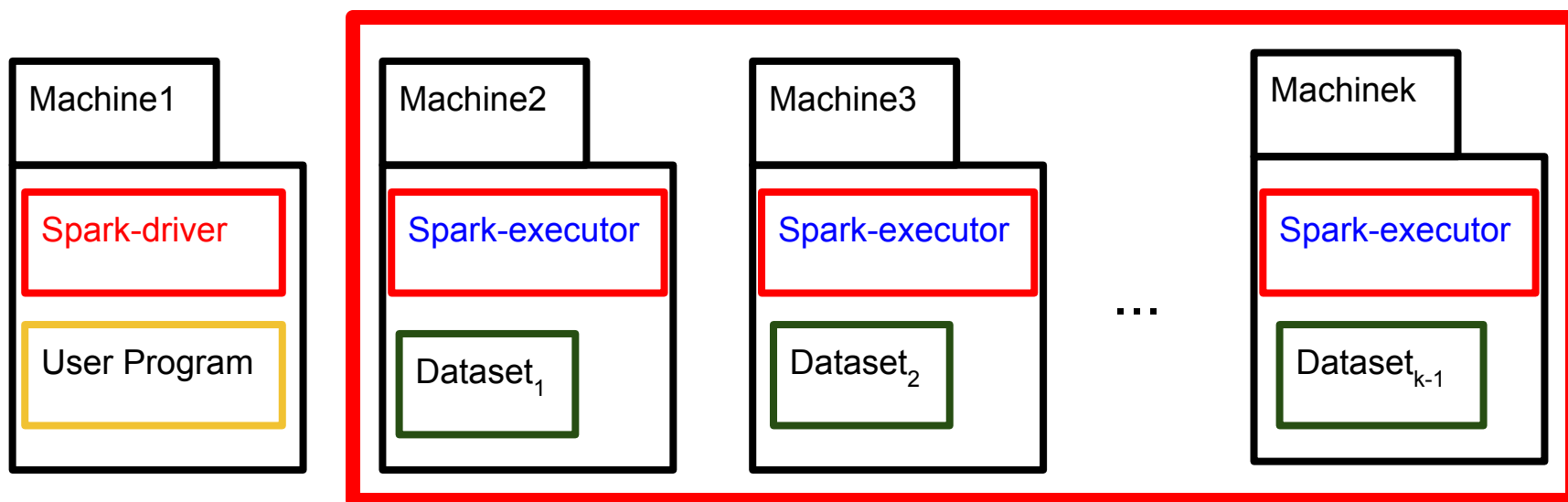
Idea 9: Spark Recipe - Set of Ingredients

5. One core in the cluster (e.g., a core of Machine 1) runs the **Spark Driver** JVM process.



Idea 9: Spark Recipe - Set of Ingredients

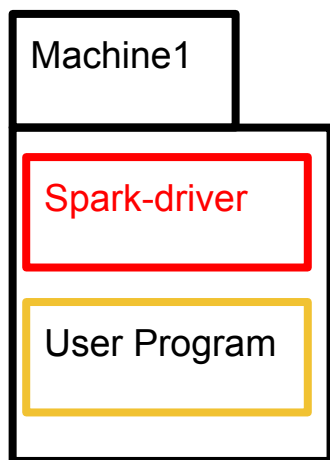
5. Remaining cores in the cluster run a **Spark Executor** JVM process.



Idea 9: Spark Recipe - Set of Ingredients

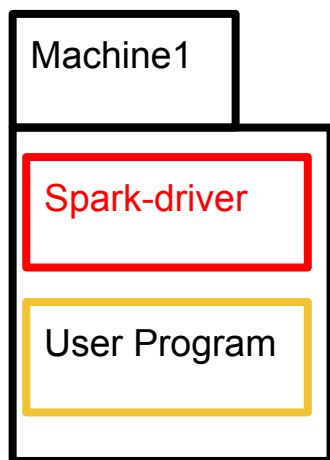
5. The **Spark Driver** JVM process plays the role of the meta-algorithm, or distributed programming centralised coordinator.

Meta-algorithm core
coordinating that
the job is done



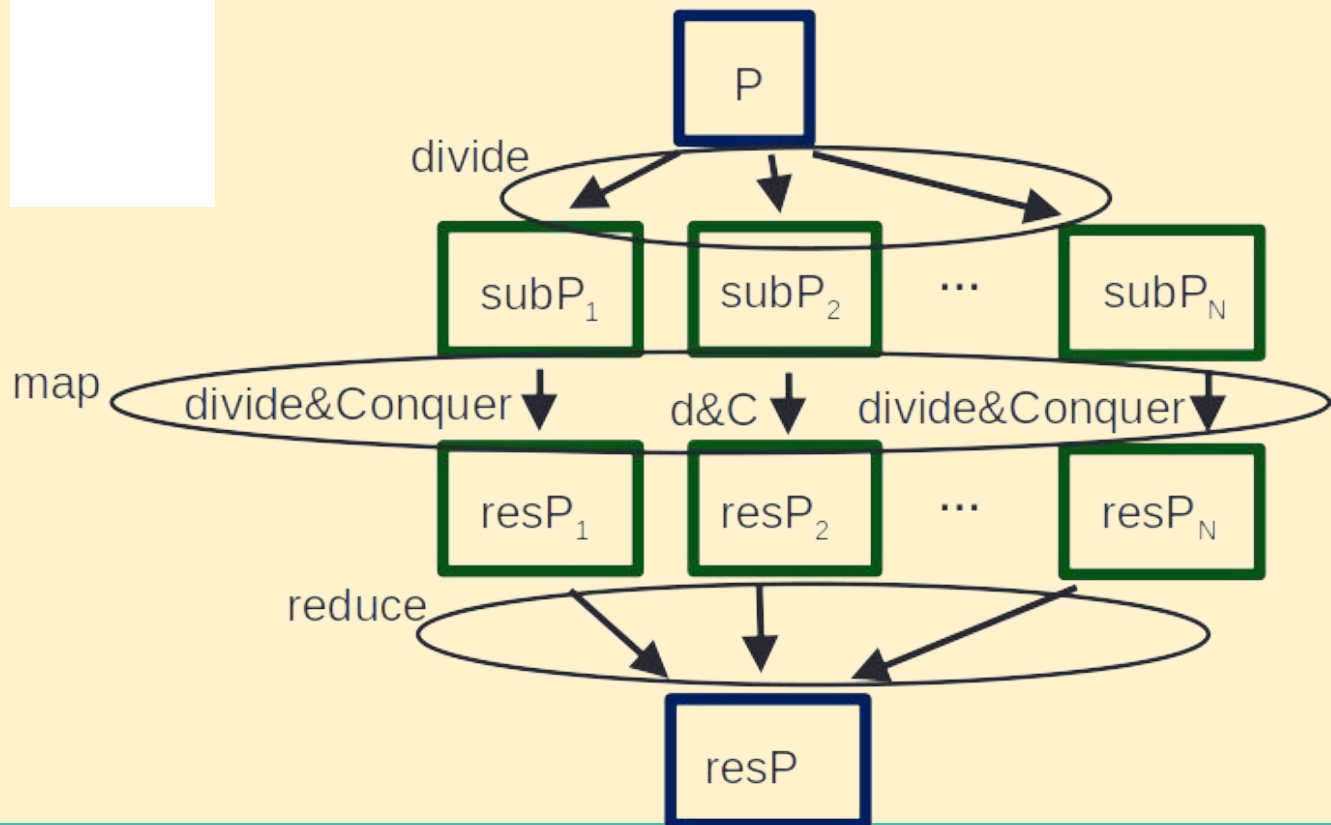
Idea 9: Spark Recipe - Set of Ingredients

5. This is really nice, as the **Spark Driver** takes care of the 2 difficulties mentioned beforehand for distributed programming:



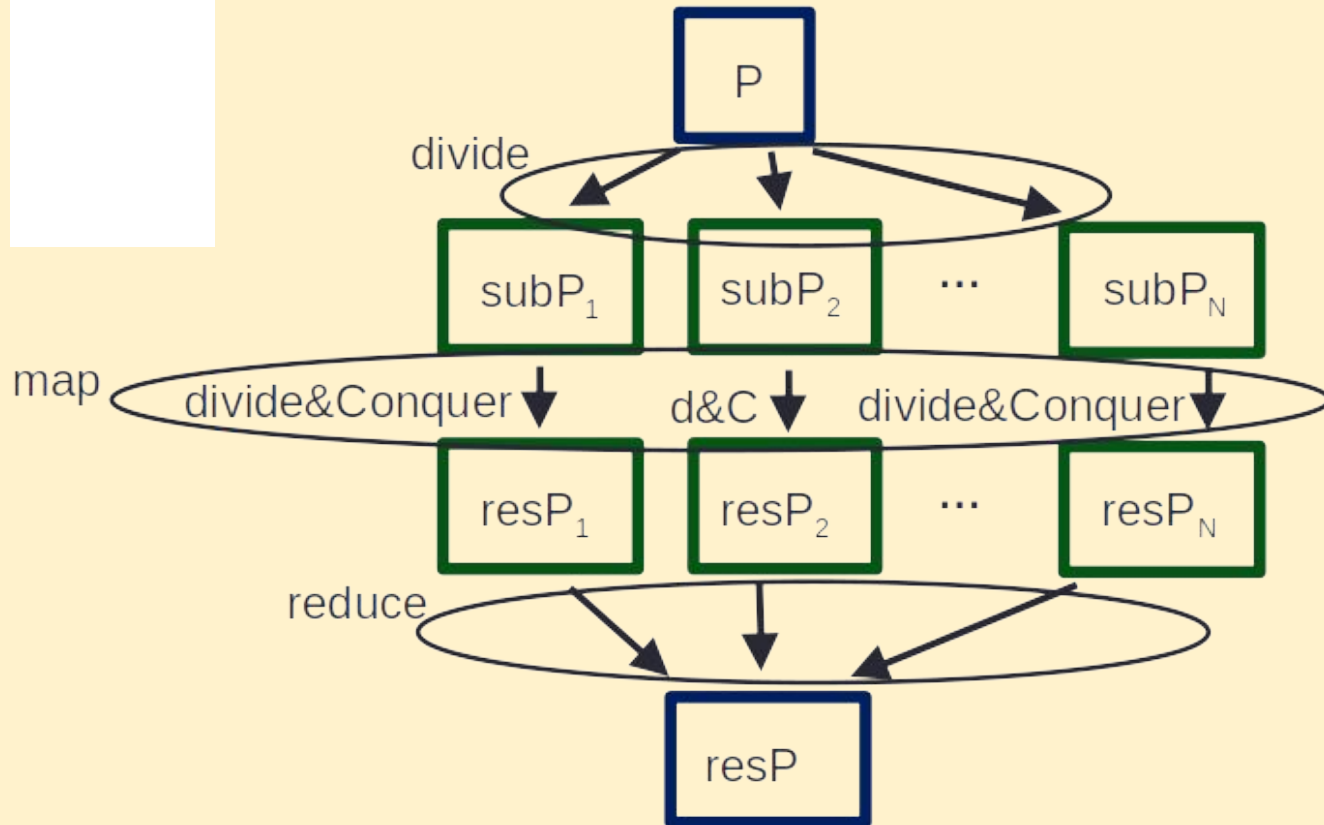
Idea 9: Spark Recipe - Set of Ingredients

Programming the meta-algorithm is difficult.



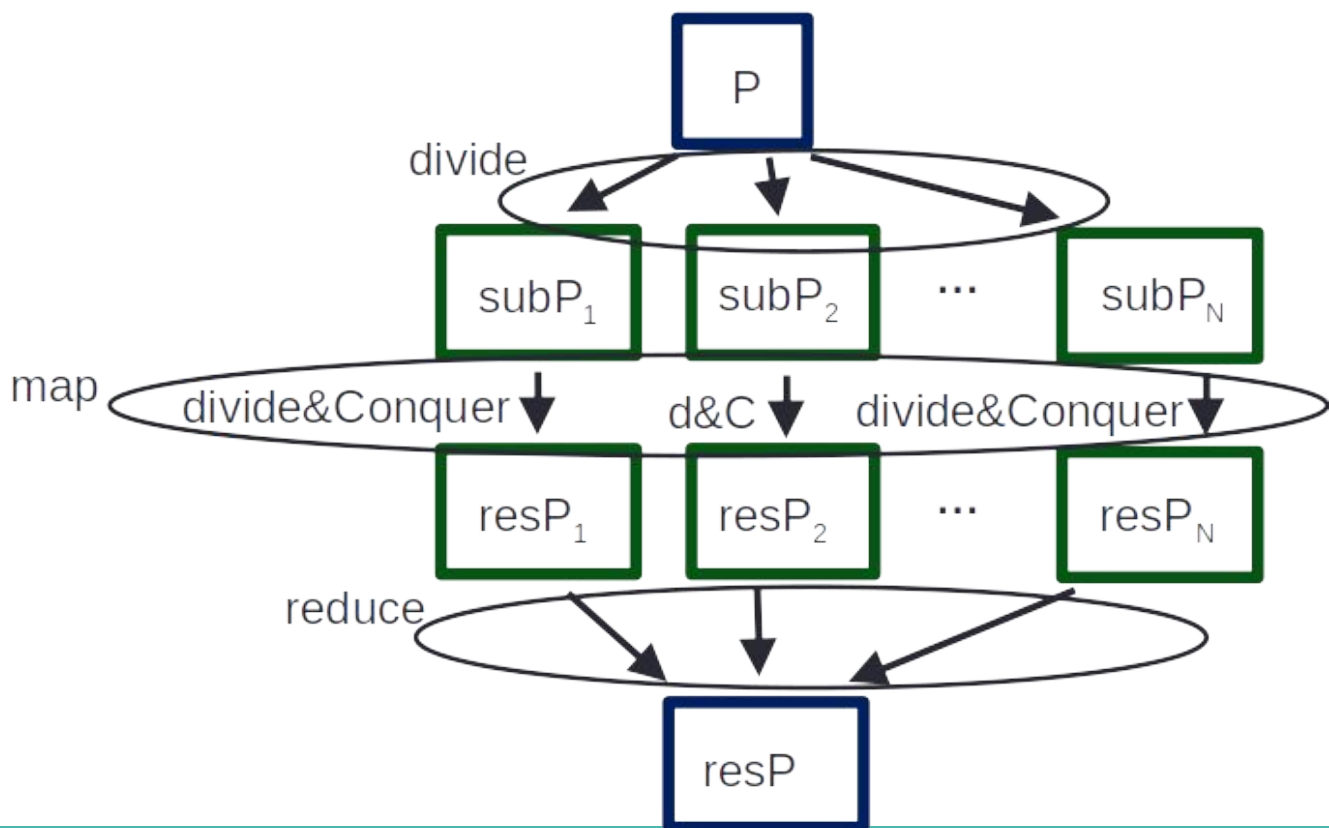
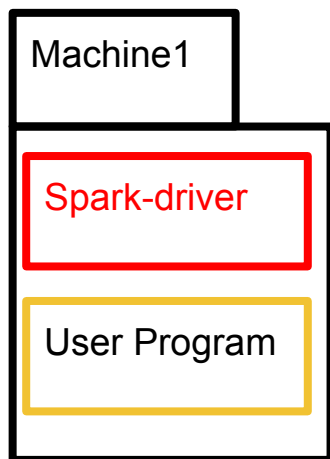
Idea 9: Spark Recipe - Set of Ingredients

Moreover, the meta-algorithm is problem domain-specific (it might not necessarily be reusable for a different problem).



Idea 9: Spark Recipe - Set of Ingredients

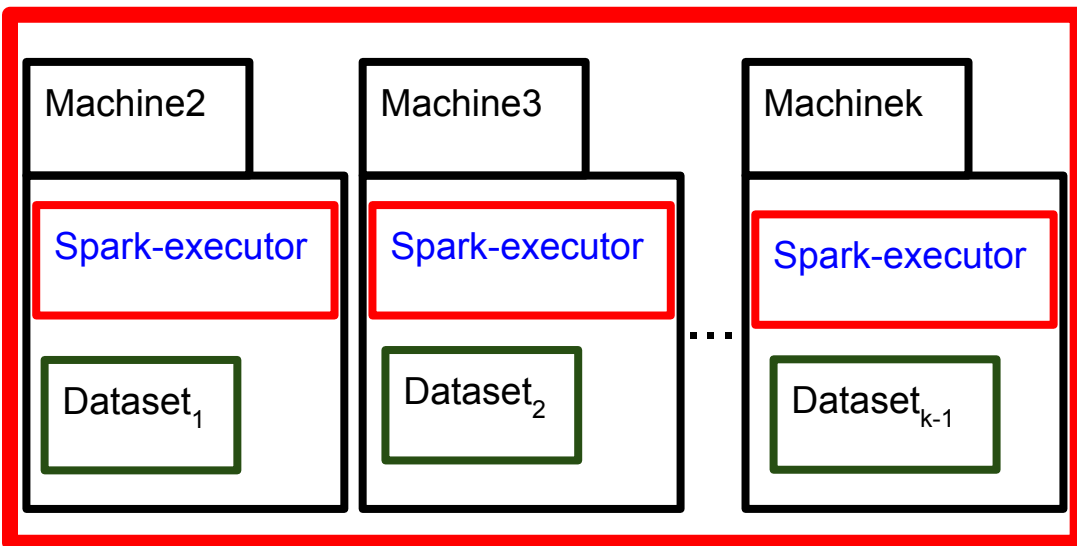
Now with Spark, the **Spark driver** makes this meta-algorithm (and its adaptation to the problem being tackled) transparent to the user.



Idea 9: Spark Recipe - Set of Ingredients

5. The **Spark Executor** JVM processes play the role of the working cores, accomplishing the tasks requested by the **Spark Driver**.

Working cores
actually doing the job

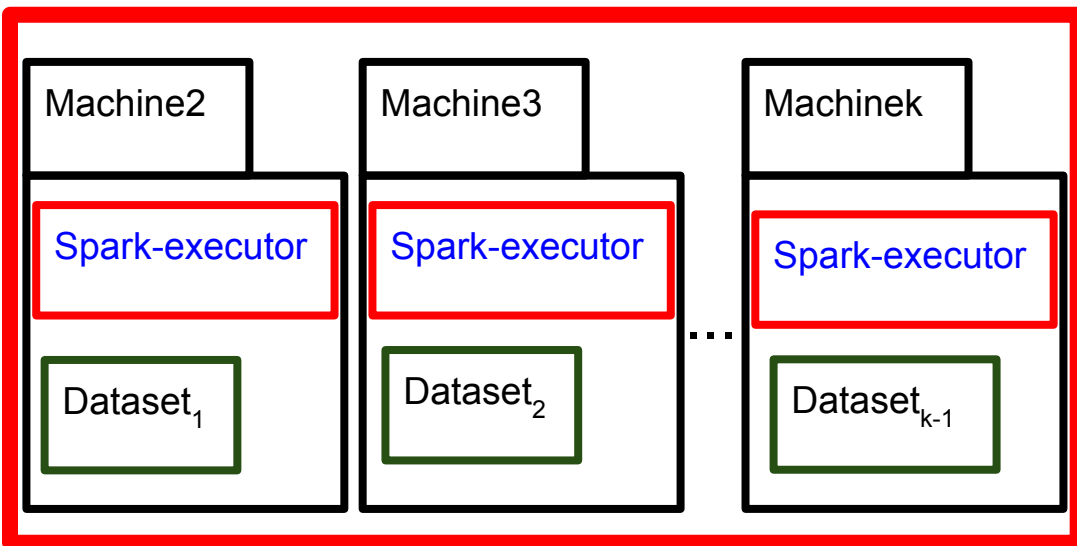


Idea 9: Spark Recipe - Set of Ingredients

5. The **Spark Executor** JVM processes play the role of the working cores, accomplishing the tasks requested by the **Spark Driver**.

This job is all about:

- Using their CPU to compute new data.
- Using their RAM and disk memory to store the computed data.



Outline

Idea 10:

Spark Core is
the Home of RDDs

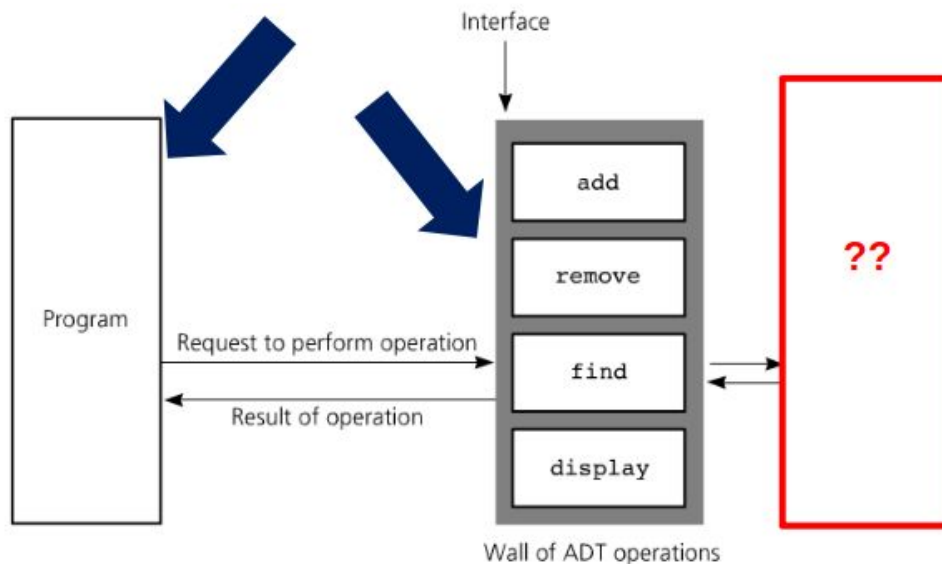
Idea 10: Spark Core is the Home of RDDs

The main **data abstraction** of Spark Core are the well-known **Resilient Distributed Datasets (RDDs)**.

An RDD is nothing but an Abstract Data Type (ADT).

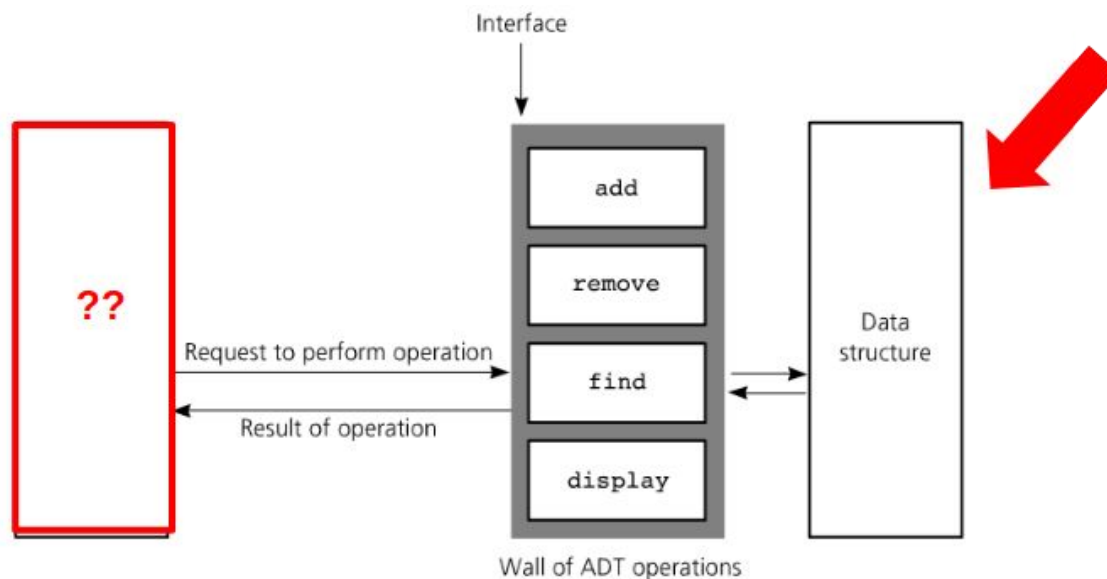
Idea 10: Spark Core is the Home of RDDs

- 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.



Idea 10: Spark Core is the Home of RDDs

- 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.



Outline

Idea 11:

Public Side of RDDs

Idea 11: Public Side of RDDs

Let's go with the [ADT public side](#):

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

Idea 11: Public Side of RDDs

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.**

Idea 11: Public Side of RDDs

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.**

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.

Idea 11: Public Side of RDDs

Let's go with the [ADT public side](#):

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

Idea 11: Public Side of RDDs

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**:

Idea 11: Public Side of RDDs

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.

Idea 11: Public Side of RDDs

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.

Idea 11: Public Side of RDDs

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. **Persistent**: They keep an RDD permanently stored until the Spark program finishes.

Idea 11: Public Side of RDDs

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. **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.

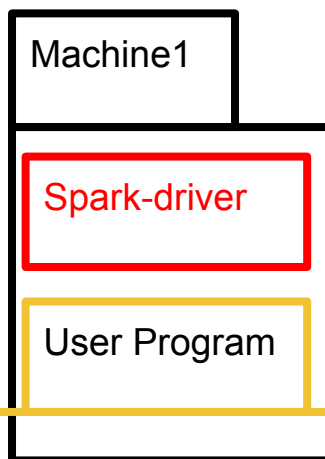
Outline

Idea 12:

Spark Core User Program Life-cycle

Idea 12: Spark Core User Program Life-Cycle

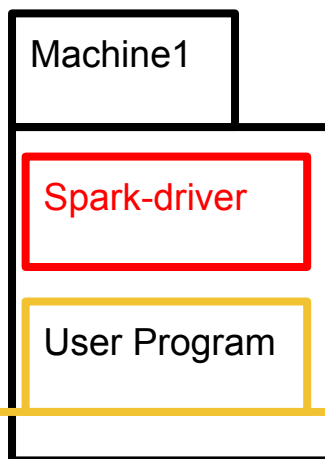
- The prefix number 1, 2, 3 and 4 in these operations is not casual: They determine the structure of a Spark typical user program!



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.

Idea 12: Spark Core User Program Life-Cycle

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

Idea 12: Spark Core User Program Life-Cycle

- 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. Creation:

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

2. Transformations:

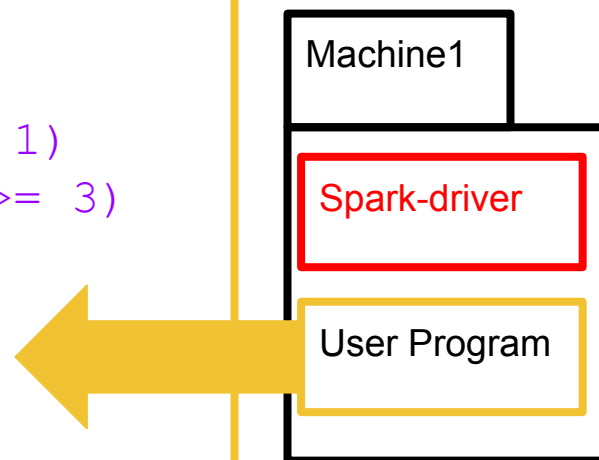
```
mappedRDD = inputRDD.map(lambda x: x + 1)  
solRDD = mappedRDD.filter(lambda x: x >= 3)
```

3. Persistence:

```
solRDD.persist( )
```

4. Actions:

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



Idea 12: Spark Core User Program Life-Cycle

- The prefix number 1, 2, 3 and 4 in these operations is not casual: They determine the structure of a Spark typical 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)
```

Machine1

Spark-driver

User Program



Outline

Idea 13:

Functional Programming Flavour of
RDD Operations

Idea 13: Functional Programming Flavour of RDD Operations

RDD operations support the following
Functional Programming (FP) features:

Idea 13: Functional Programming Flavour of RDD Operations

RDD operations support the following
Functional Programming (FP) features:

1. Polymorphism.

- Functions support parametric input/output arguments.

Idea 13: Functional Programming Flavour of RDD Operations

RDD operations support the following
Functional Programming (FP) features:

1. **Polymorphism.**

- Functions support parametric input/output arguments.

2. **Higher-order functions.**

- Functions support receiving functions as arguments, and/or returning a function as a result.

Idea 13: Functional Programming Flavour of RDD Operations

RDD operations support the following
Functional Programming (FP) features:

1. Polymorphism.

- Functions support parametric input/output arguments.

2. Higher-order functions.

- Functions support receiving functions as arguments, and/or returning a function as a result.

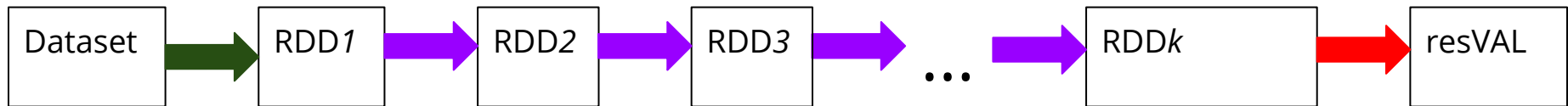
3. Partial application.

- Functions support fixing a number of its arguments, leading to another function of smaller arity.

Idea 13: Functional Programming Flavour of RDD Operations

All in all, a Spark Core program life-cycle is based in the RDD ops:

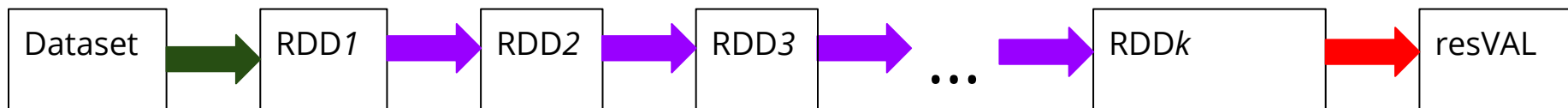
A very small set of FP primitives,
each of them supporting the application of very general functions.



Idea 13: Functional Programming Flavour of RDD Operations

All in all, a Spark Core program life-cycle is based in the RDD ops:

A very small set of FP primitives,
each of them supporting the application of very general functions.



```
newRDD = inputRDD.map( f ) with any  $f: a \rightarrow b$ 
```

```
resVAL = inputRDD.reduce( f ) with any  $f: a \rightarrow a \rightarrow a$ 
```

```
resVAL = inputRDD.aggregate( accum, f1, f2 )  
With any  $accum:: b, f1: b \rightarrow a \rightarrow b, f2: b \rightarrow b \rightarrow b$ 
```

```
newRDD = inputRDD.reduceByKey( f ) with any  $f: a \rightarrow a$ 
```

```
newRDD = inputRDD.combineByKey( f1, f2, f3 )  
with any  $f1: a \rightarrow b, f2: b \rightarrow a \rightarrow b$  and  $f3: b \rightarrow b \rightarrow b$ 
```

Outline

Idea 14:

RDDs are Lazily Evaluated

Idea 14: RDDs are Lazily Evaluated

Creation, **Transformation** and **Persist** operations are lazy!



Idea 14: RDDs are Lazily Evaluated

They only start working when an **Action** takes place!

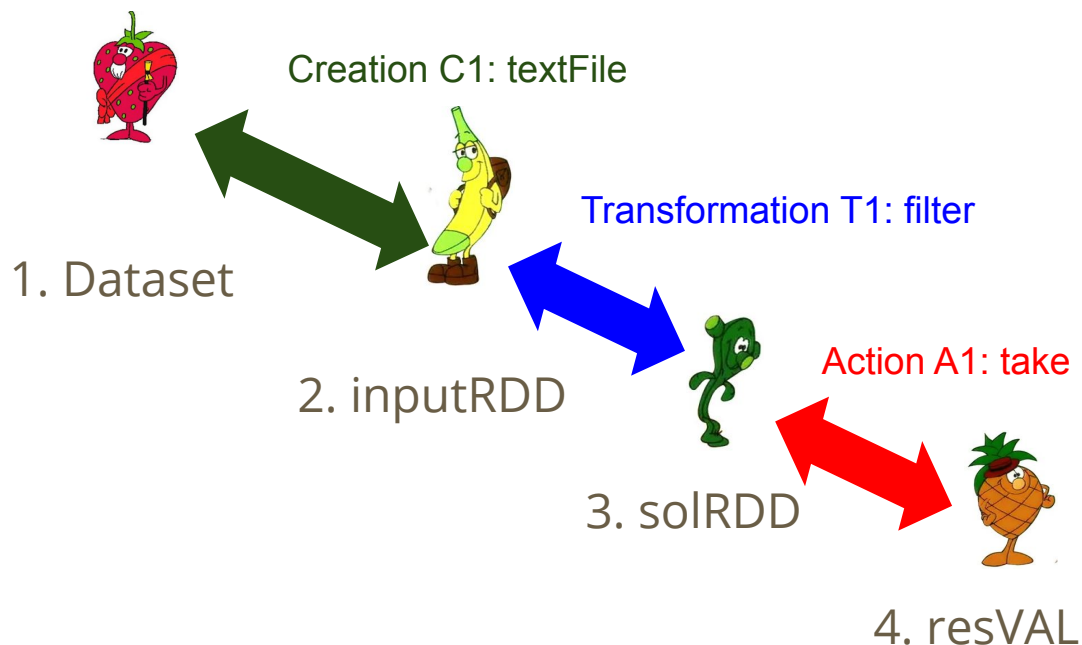


Idea 14: RDDs are Lazily Evaluated



Idea 14: RDDs are Lazily Evaluated

Moreover, when it's time for these operations to work, they only compute as much data as it is absolutely required.

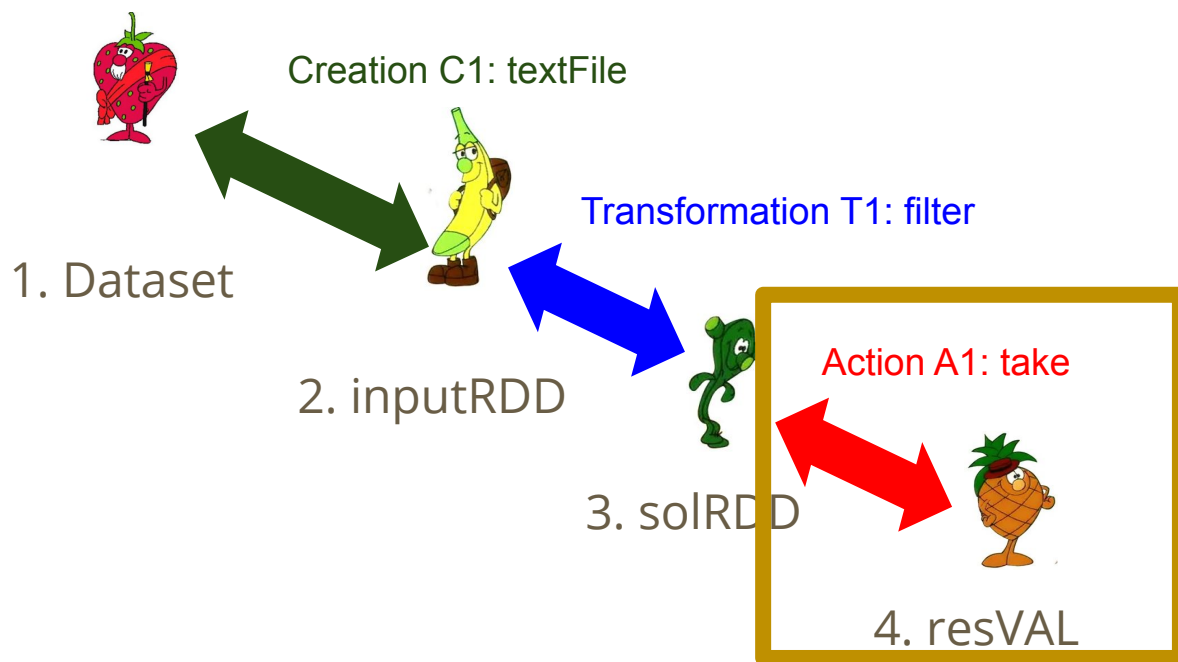
**Spark-driver**

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)
```

Idea 14: RDDs are Lazily Evaluated

Specifically, the **Spark driver** process focuses on the **action** operation triggering the computation...

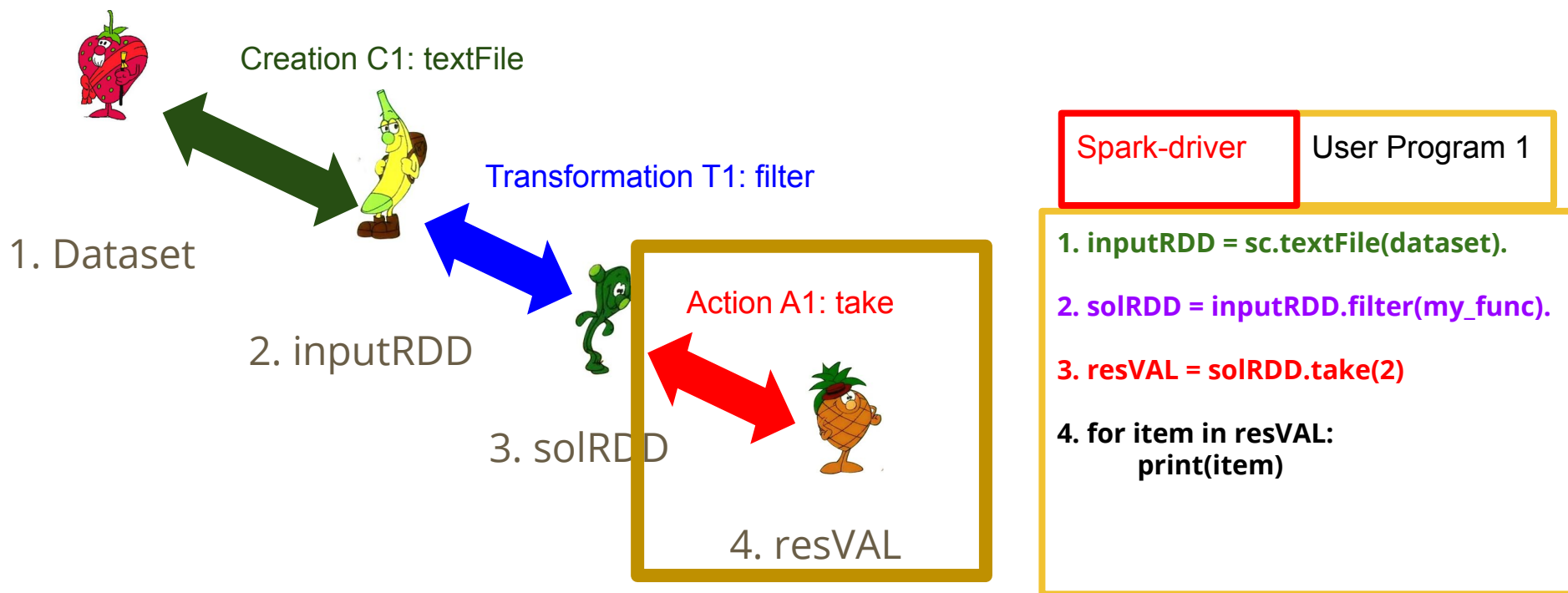
**Spark-driver**

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)
```


Idea 14: RDDs are Lazily Evaluated

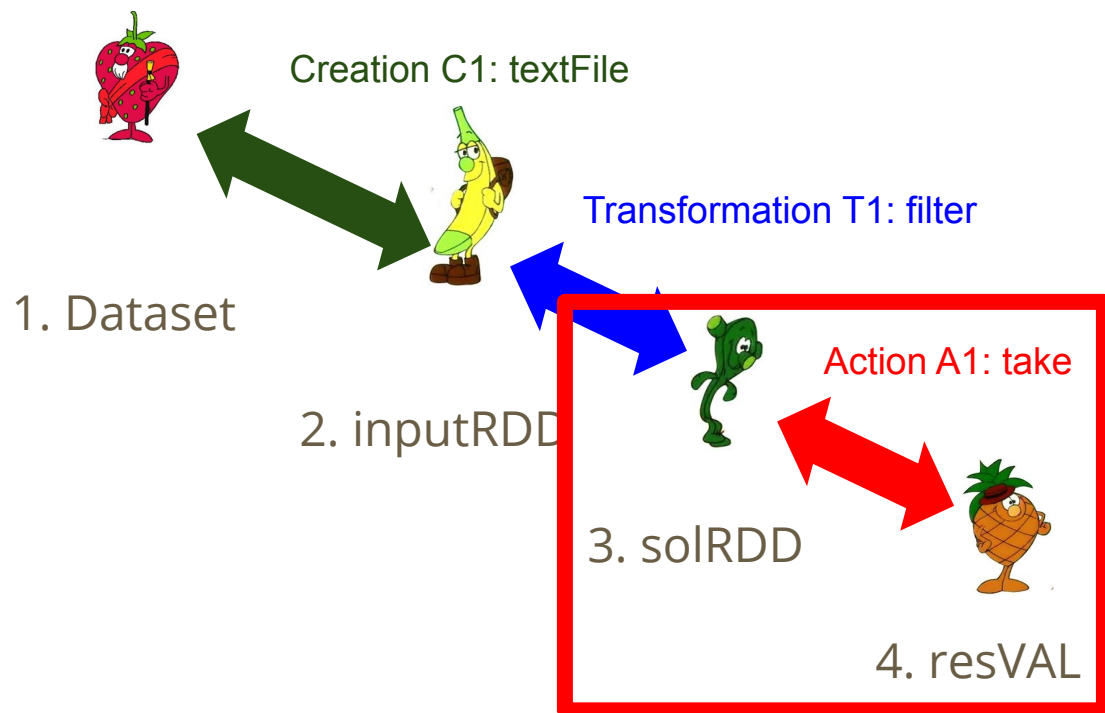
Specifically, the **Spark driver** process focuses on the **action** operation triggering the computation...
...to start tracing backwards from it, so as to reason what is the minimum amount of data that must be computed.



Idea 14: RDDs are Lazily Evaluated



What is the minimum data we have to compute?



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)
```

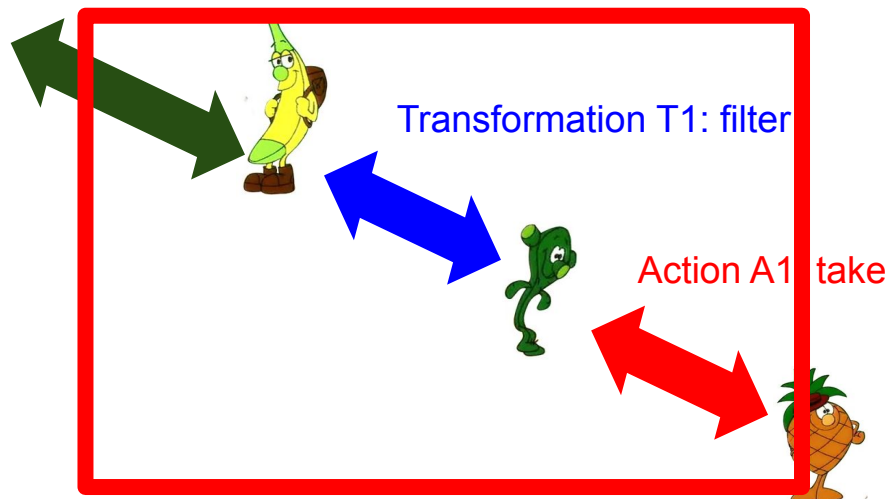
Idea 14: RDDs are Lazily Evaluated



What is the minimum data we have to compute?



Creation C1: textFile



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)
```

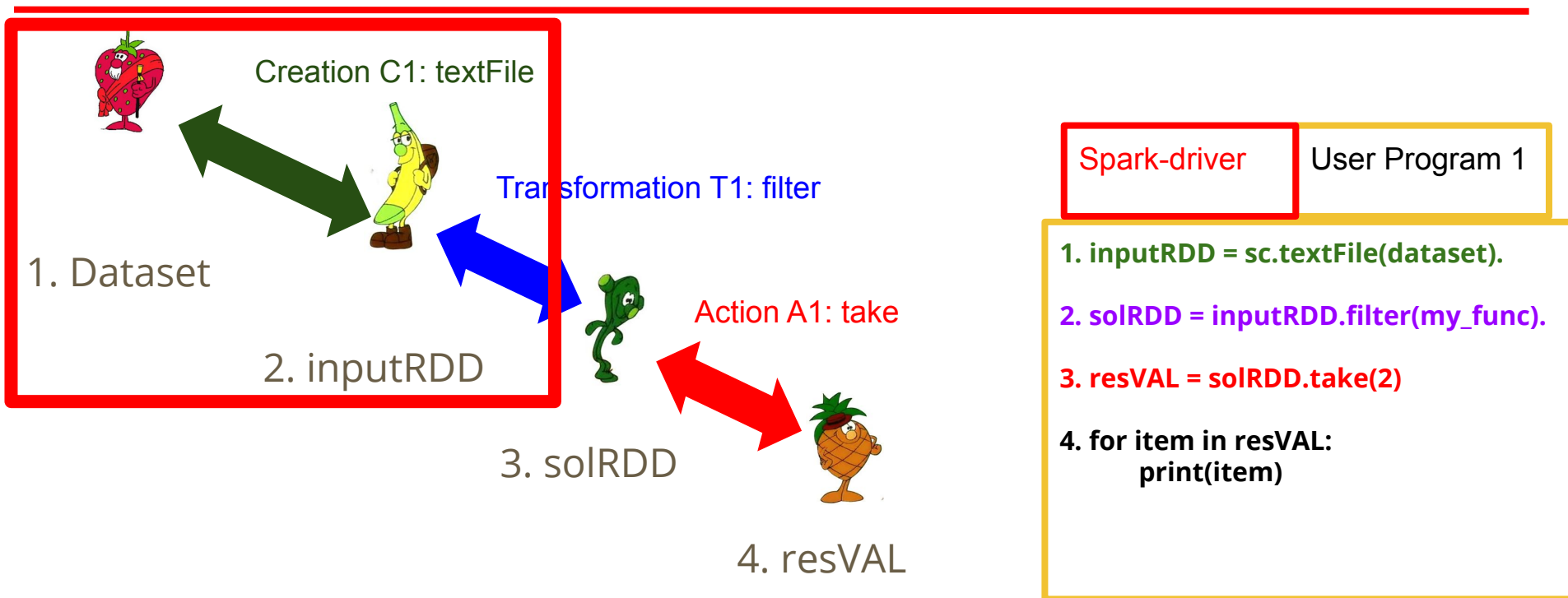
1. Dataset

2. inputRDD

Idea 14: RDDs are Lazily Evaluated



What is the minimum data we have to compute?



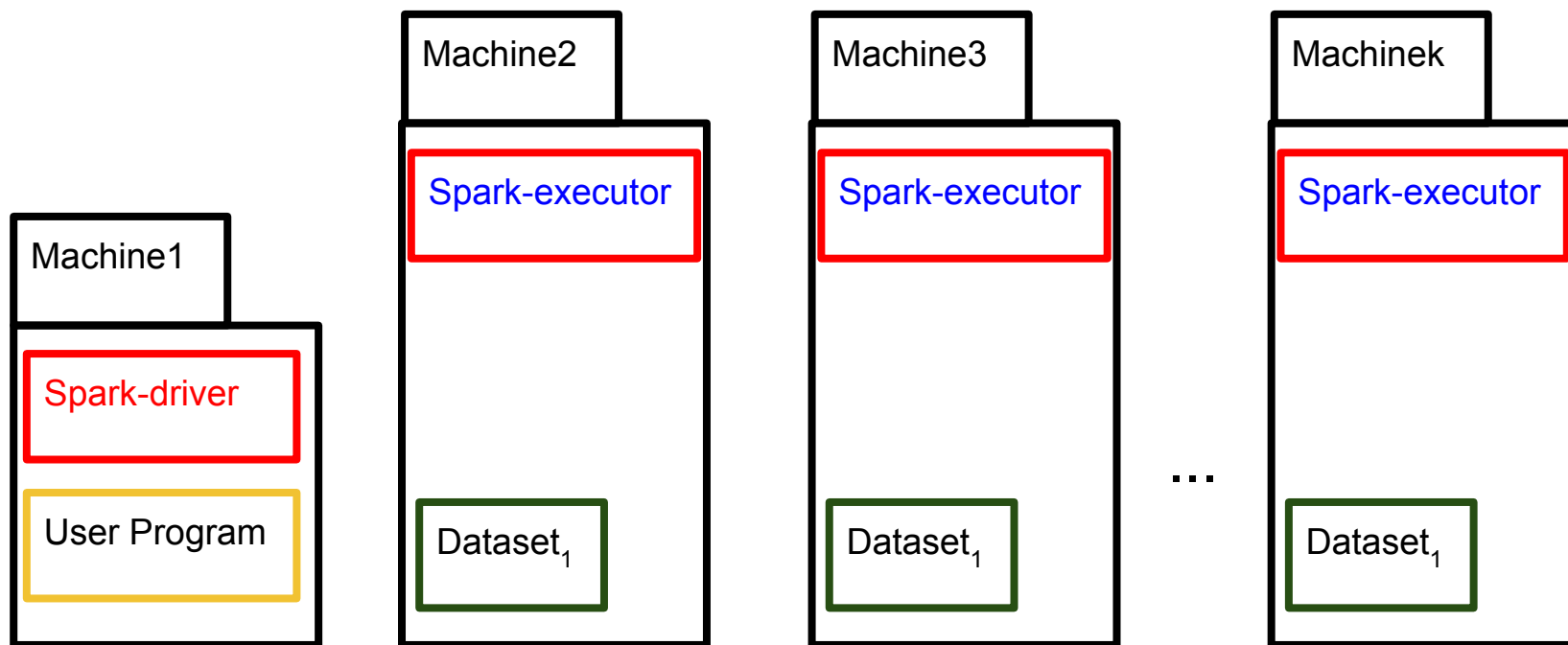
Outline

Idea 15:

Private Side of RDDs

Idea 15: Private Side of RDDs

- Partitions allow to distribute an RDD over the machines of the cluster.

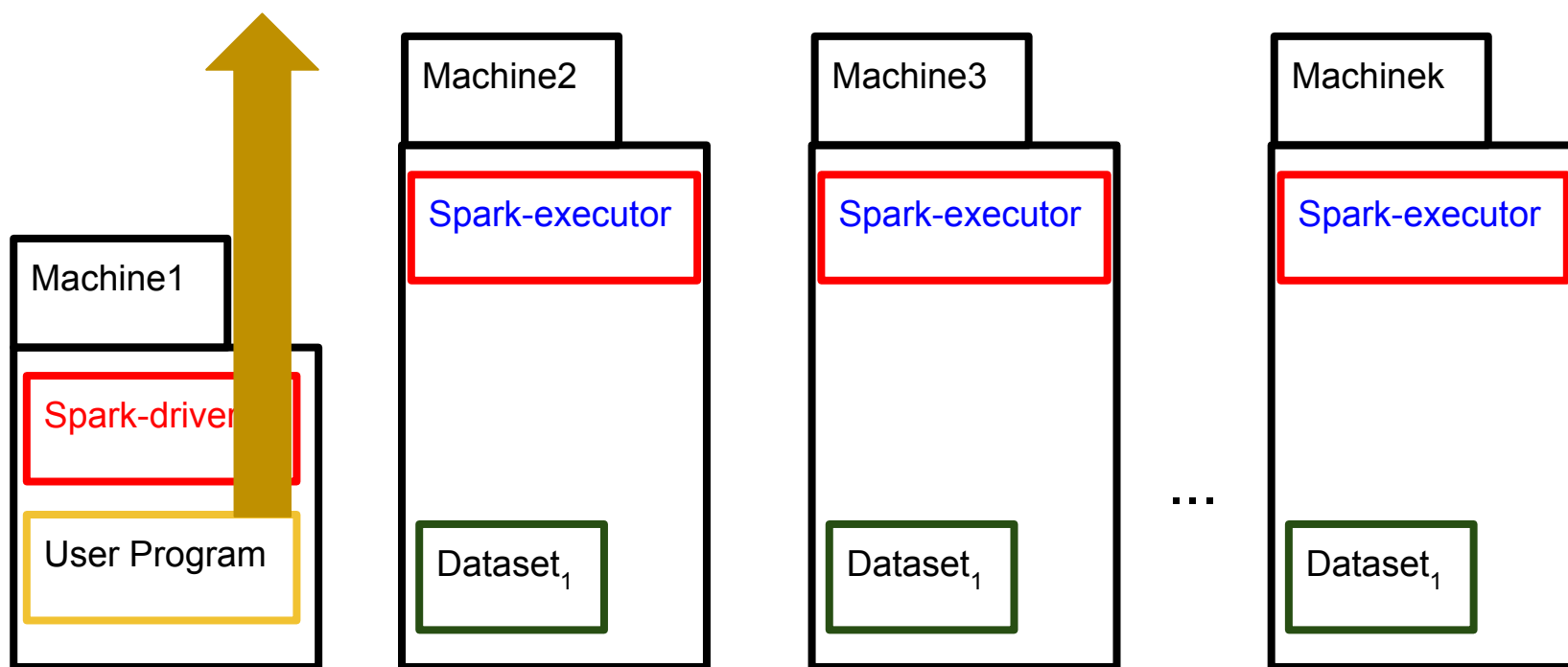


Idea 15: Private Side of RDDs

- Partitions allow to distribute an RDD over the machines of the cluster.



```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7 ] )
```

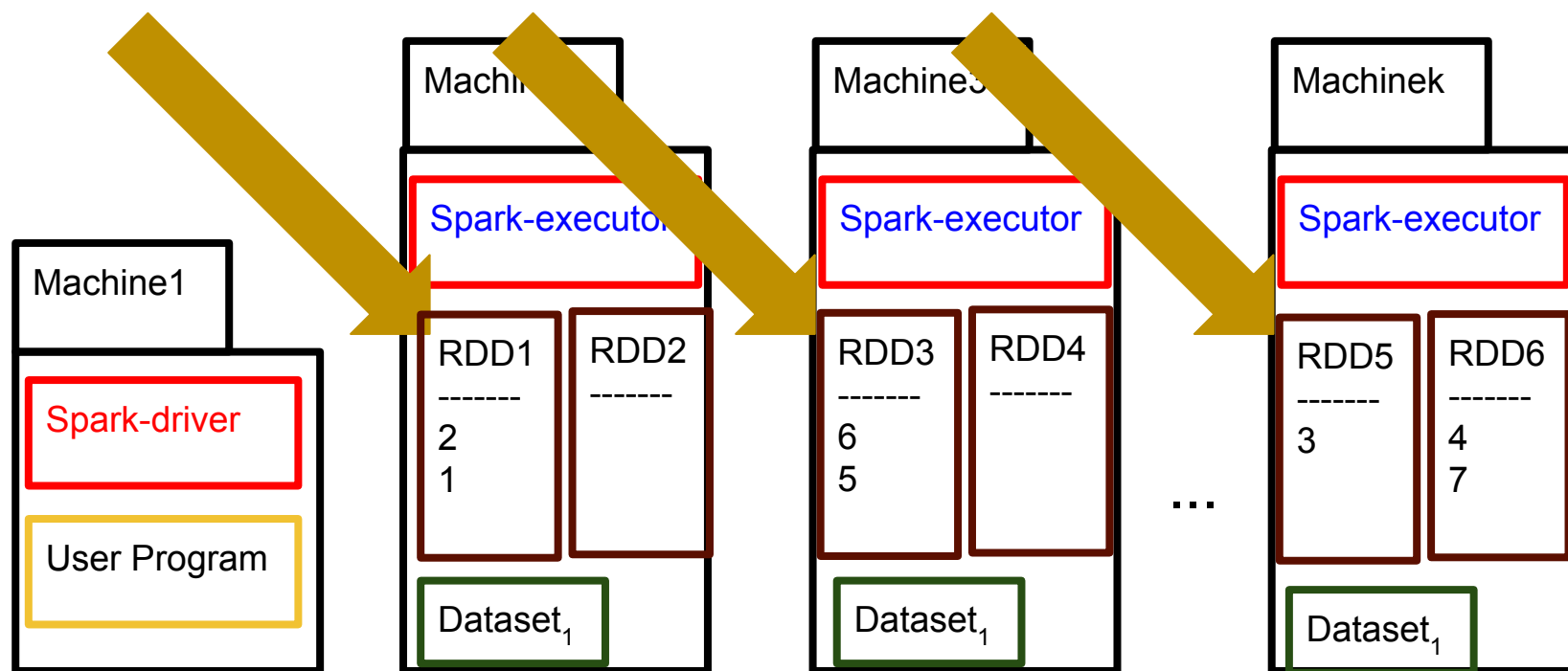


Idea 15: Private Side of RDDs

- Partitions allow to distribute an RDD over the machines of the cluster.



```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7 ] )
```

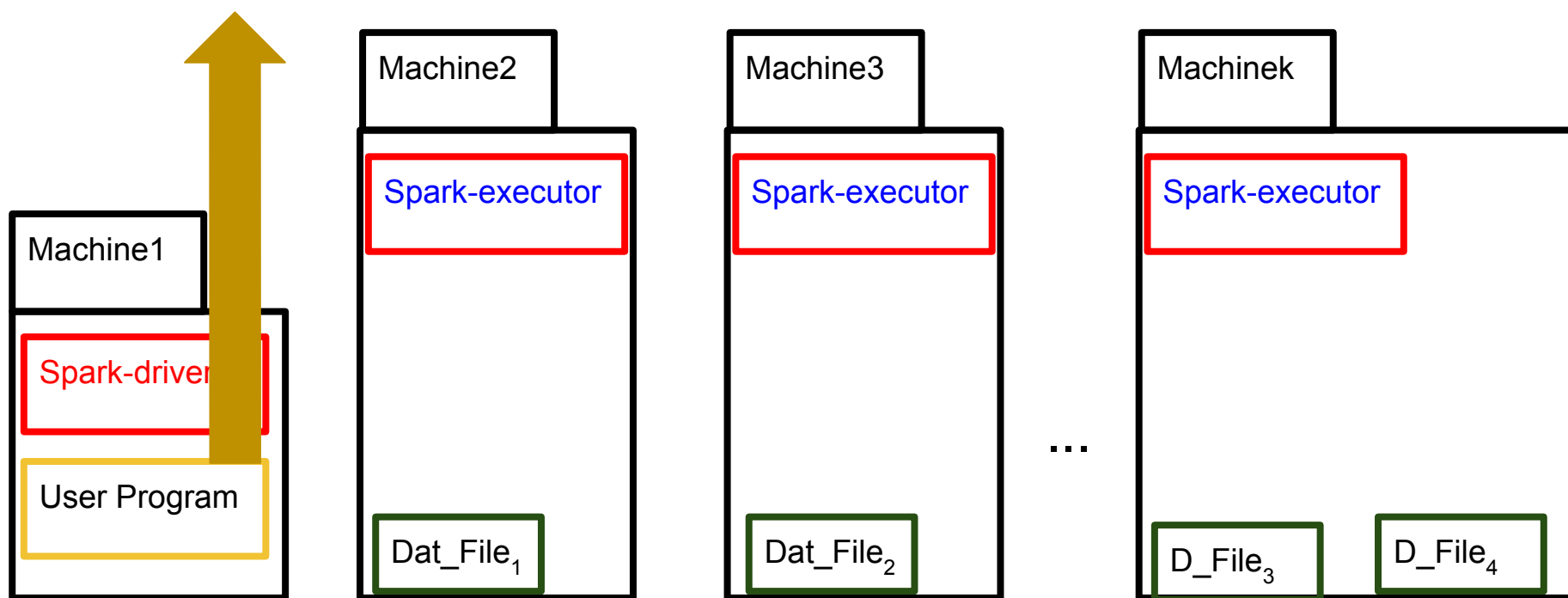


Idea 15: Private Side of RDDs

- Partitions allow to distribute an RDD over the machines of the cluster.



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

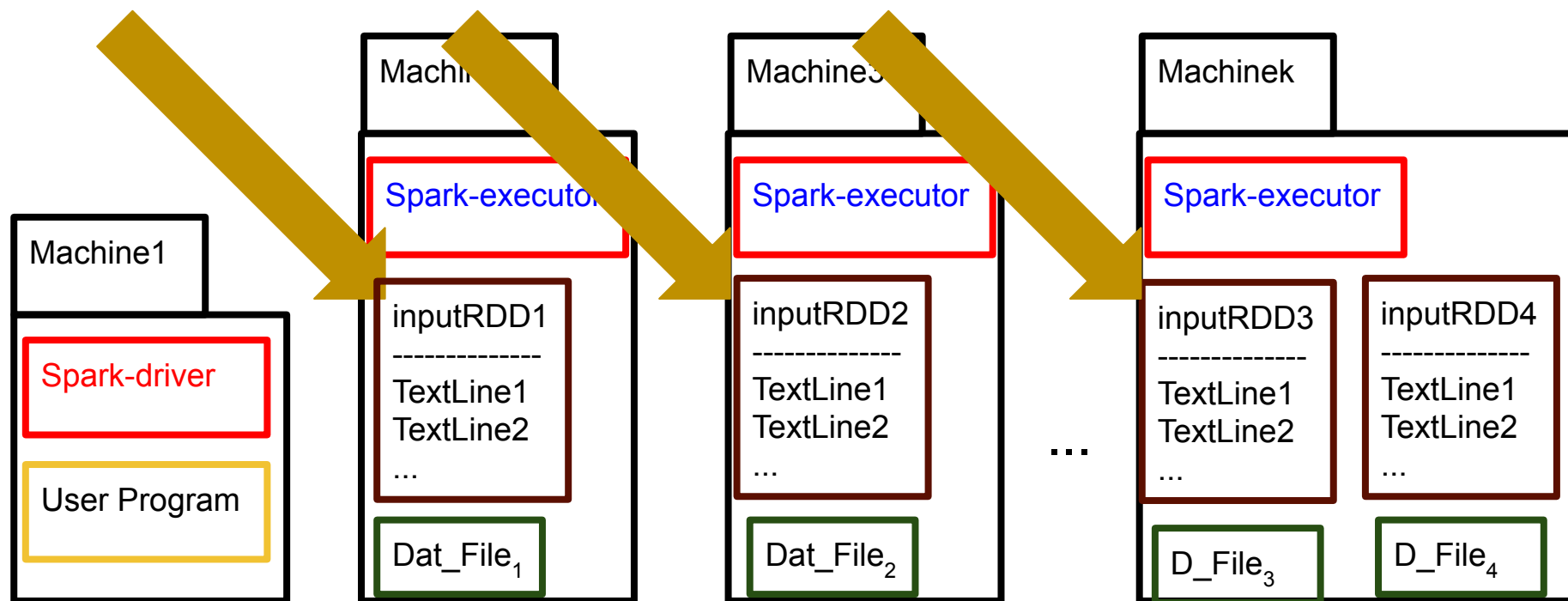


Idea 15: Private Side of RDDs

- Partitions allow to distribute an RDD over the machines of the cluster.

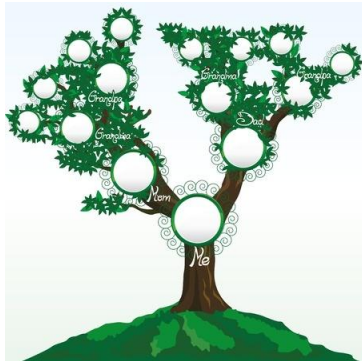


```
inputRDD = sc.textFile( my_dataset )
```

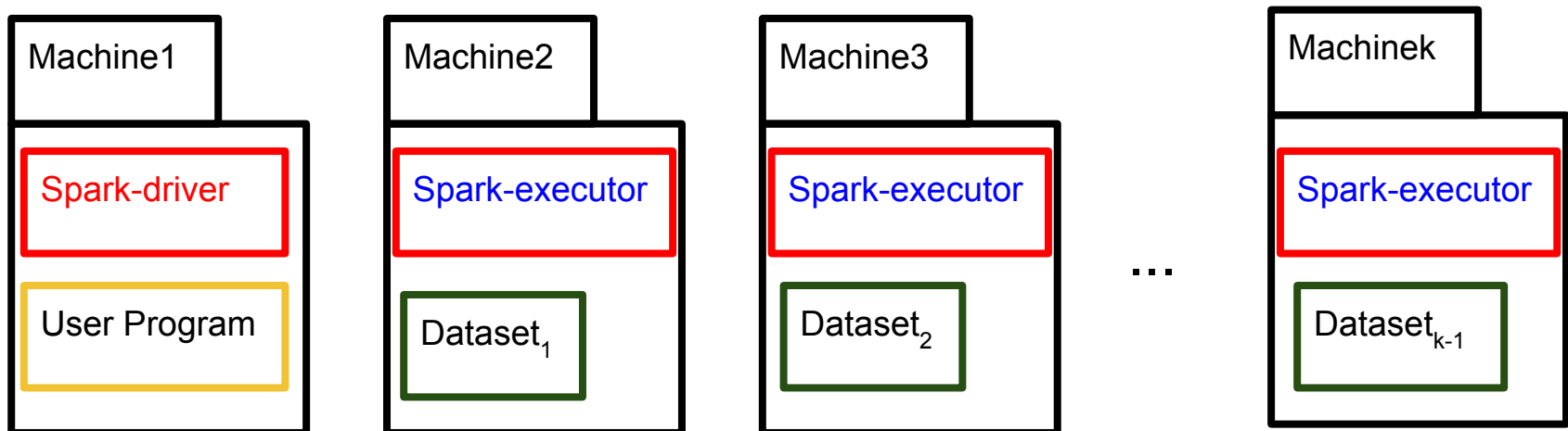


Idea 15: Private Side of RDDs

- Lineage allows us to (re-)compute partitions based on their dependencies. This is crucial for lazy evaluation and fault tolerance.



RDDs

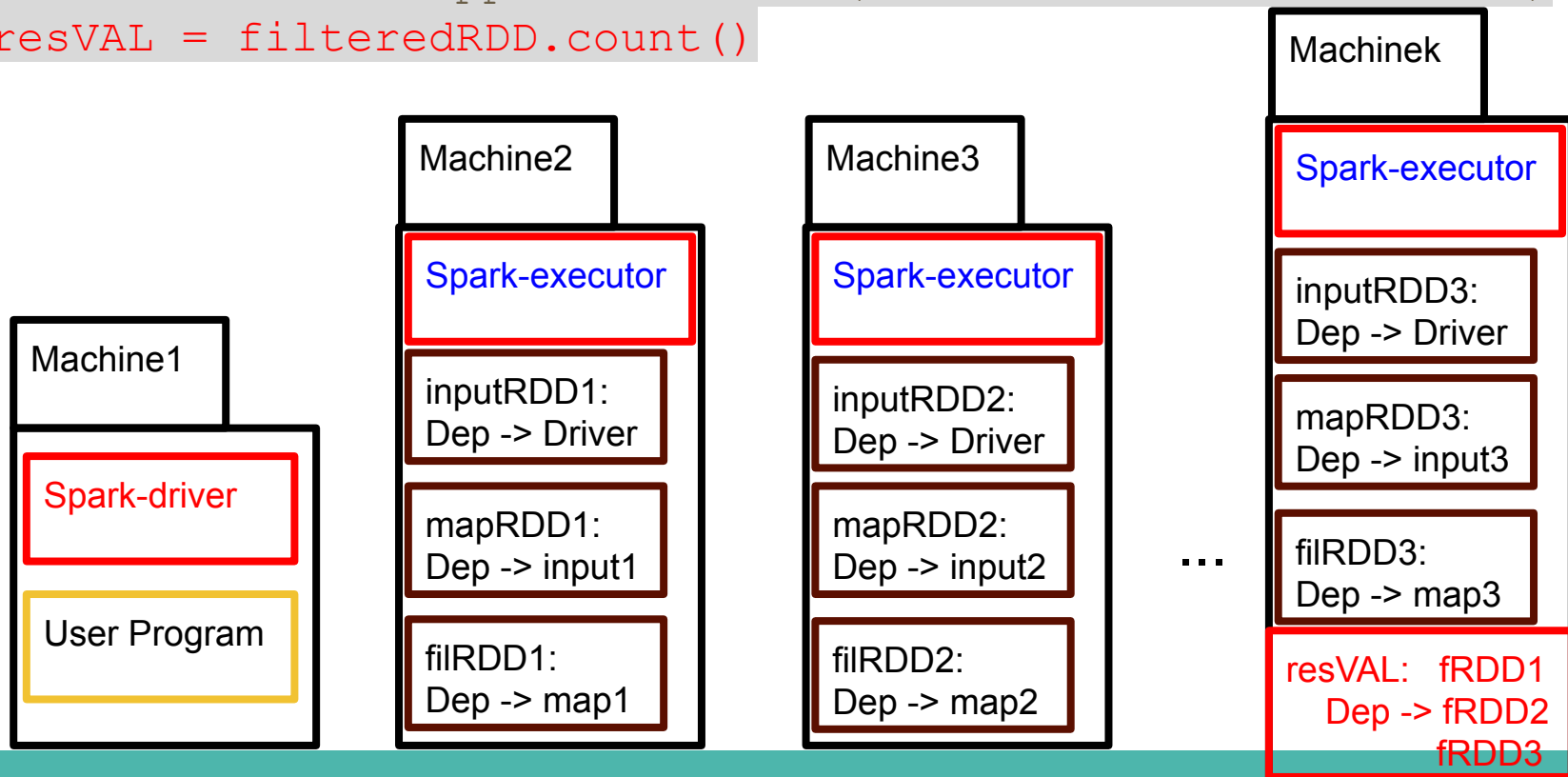


Idea 15: Private Side of RDDs



When an **action** takes place, computation is triggered by tracing this lineage backwards.

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

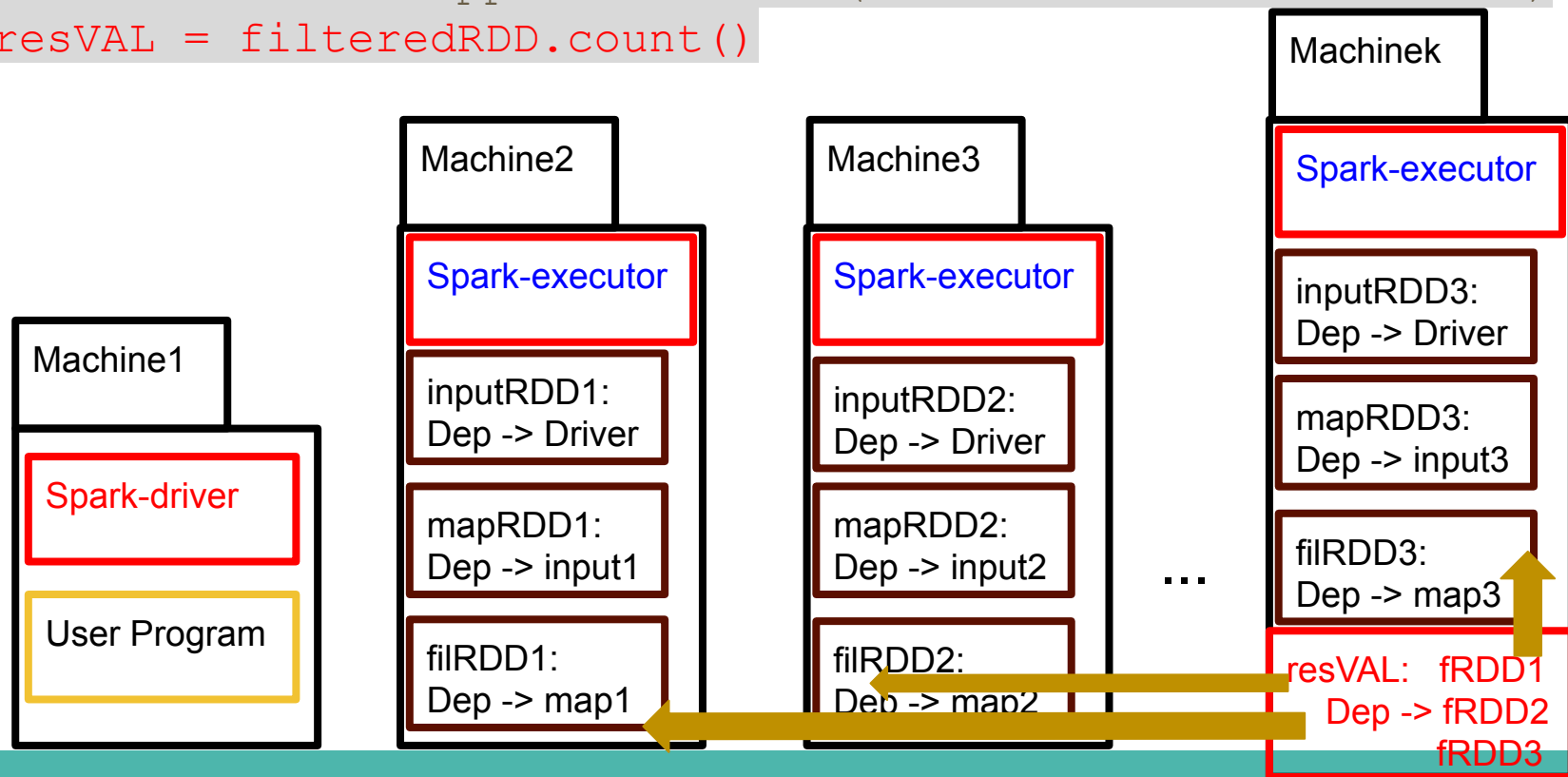


Idea 15: Private Side of RDDs



Who do I depend on?

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

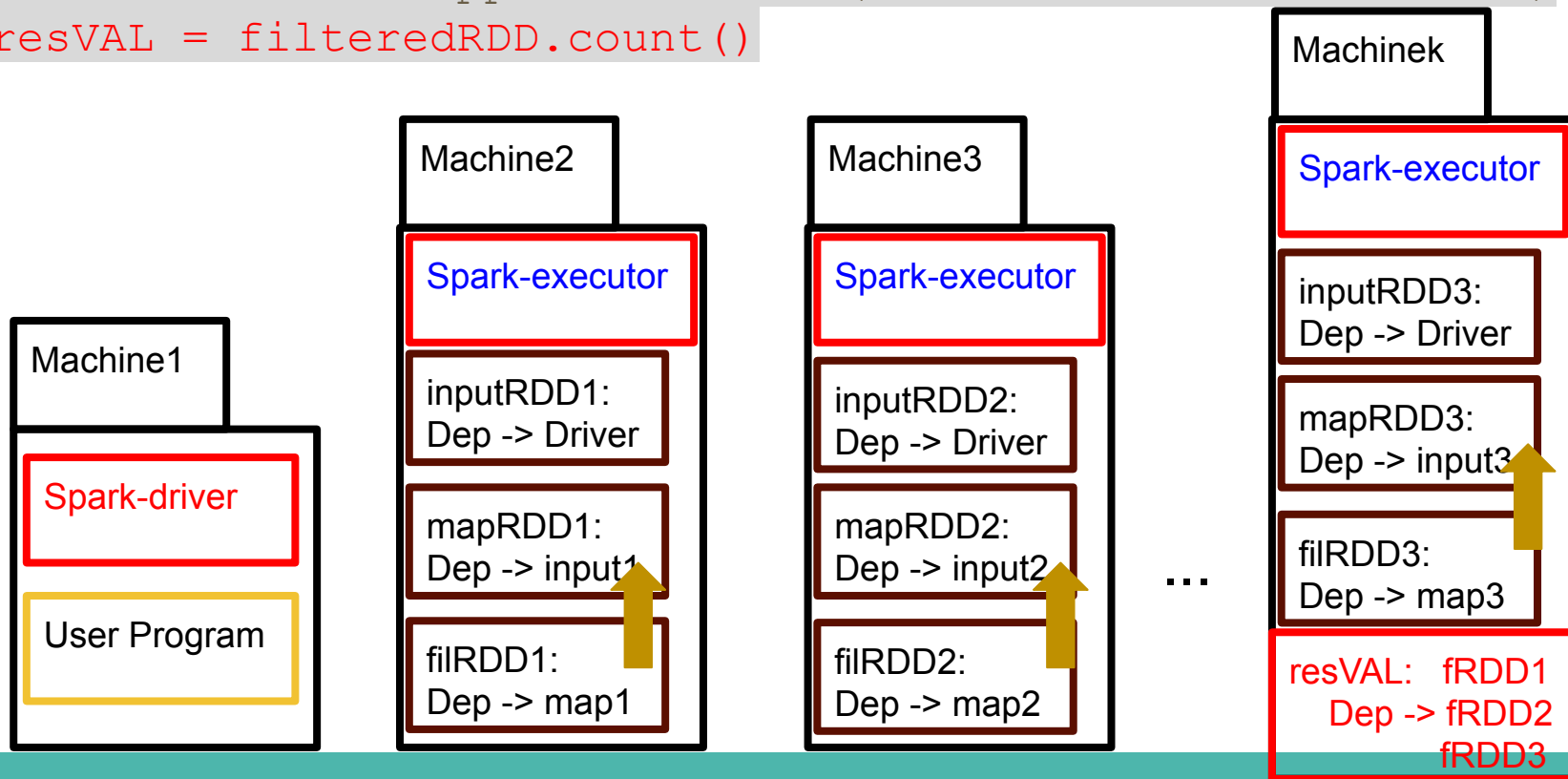


Idea 15: Private Side of RDDs



And, likewise, who do these RDD partitions depend on?

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

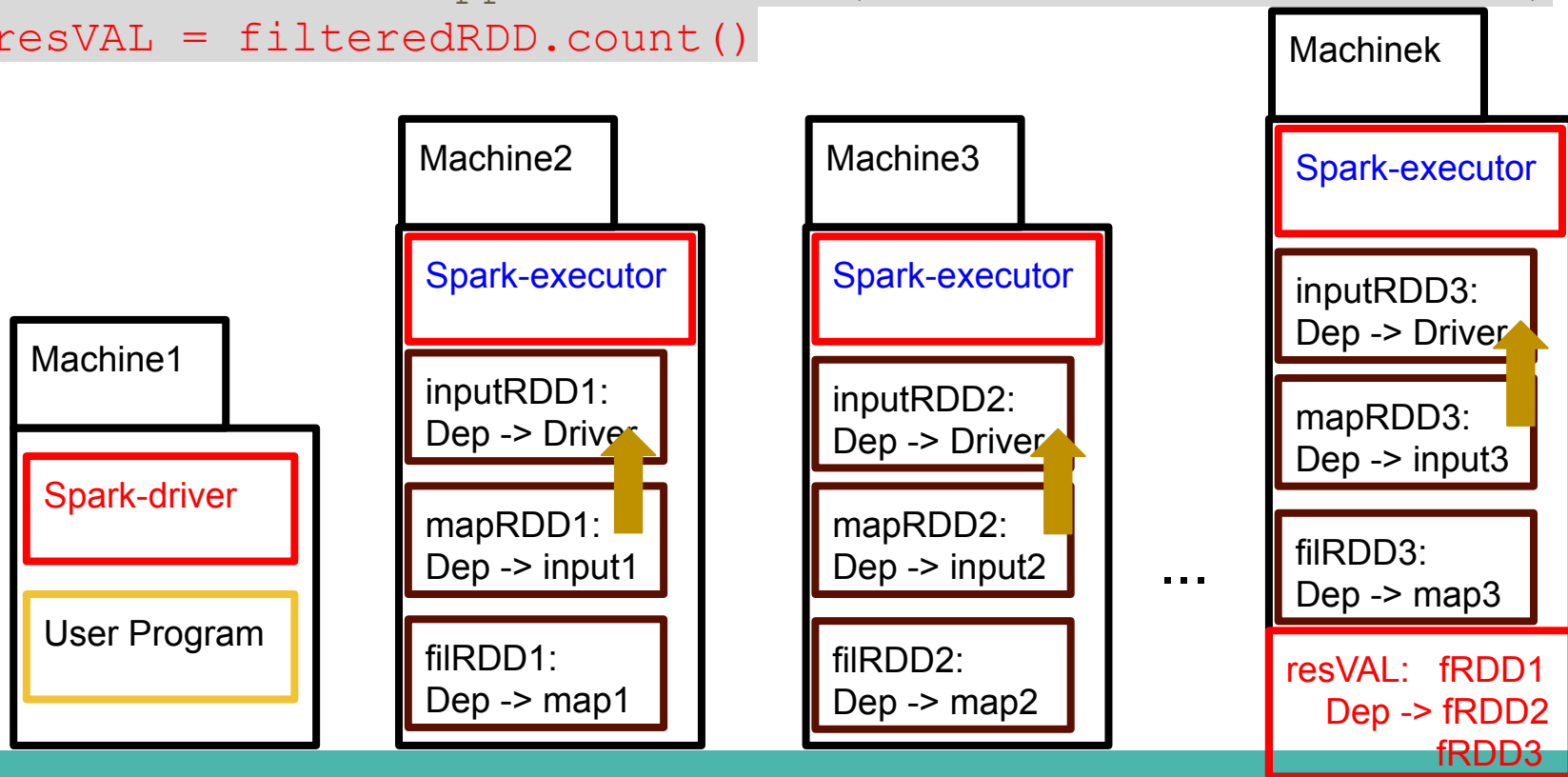


Idea 15: Private Side of RDDs

And so on and so on...



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

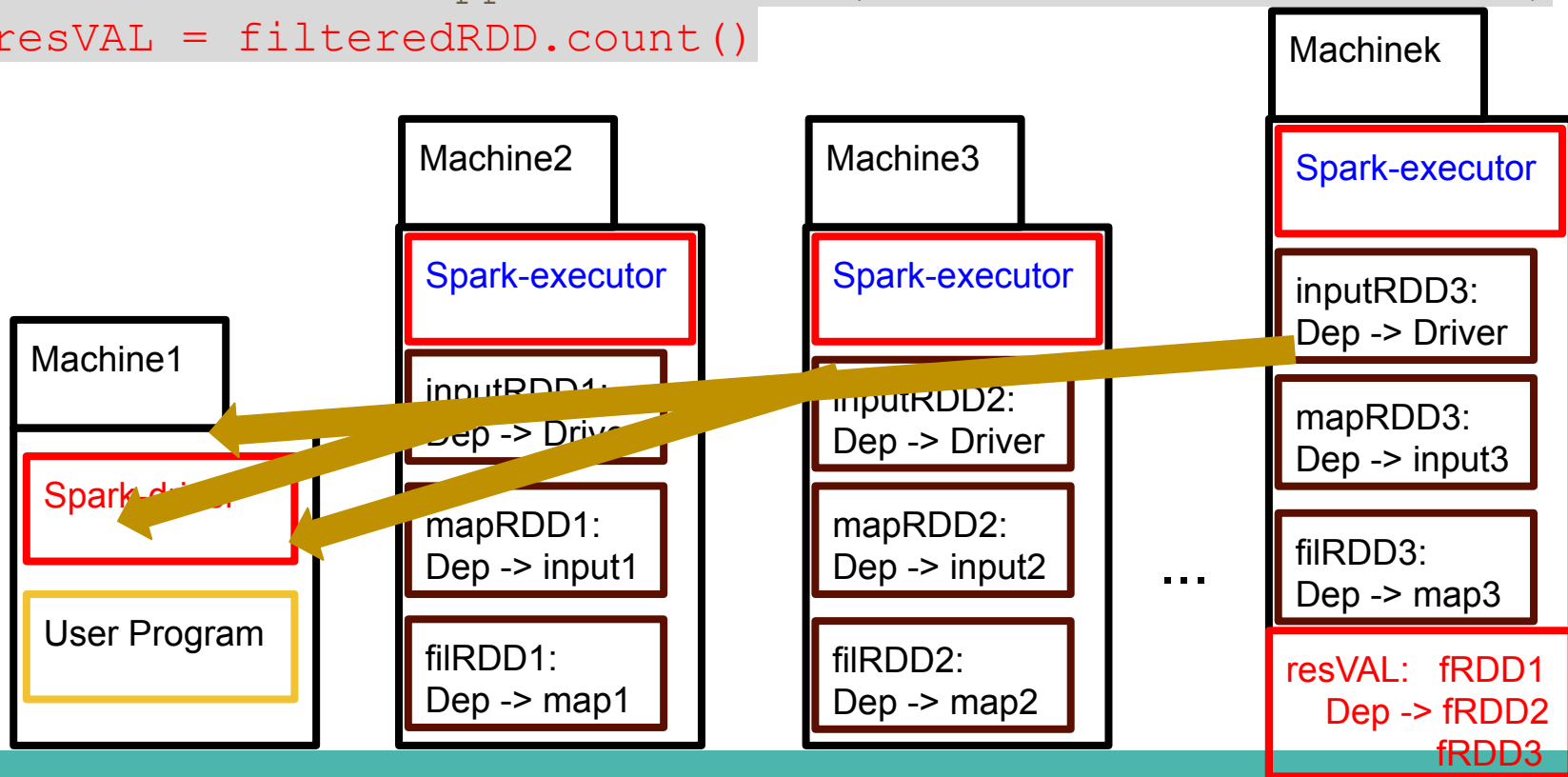


Idea 15: Private Side of RDDs

And so on and so on...



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

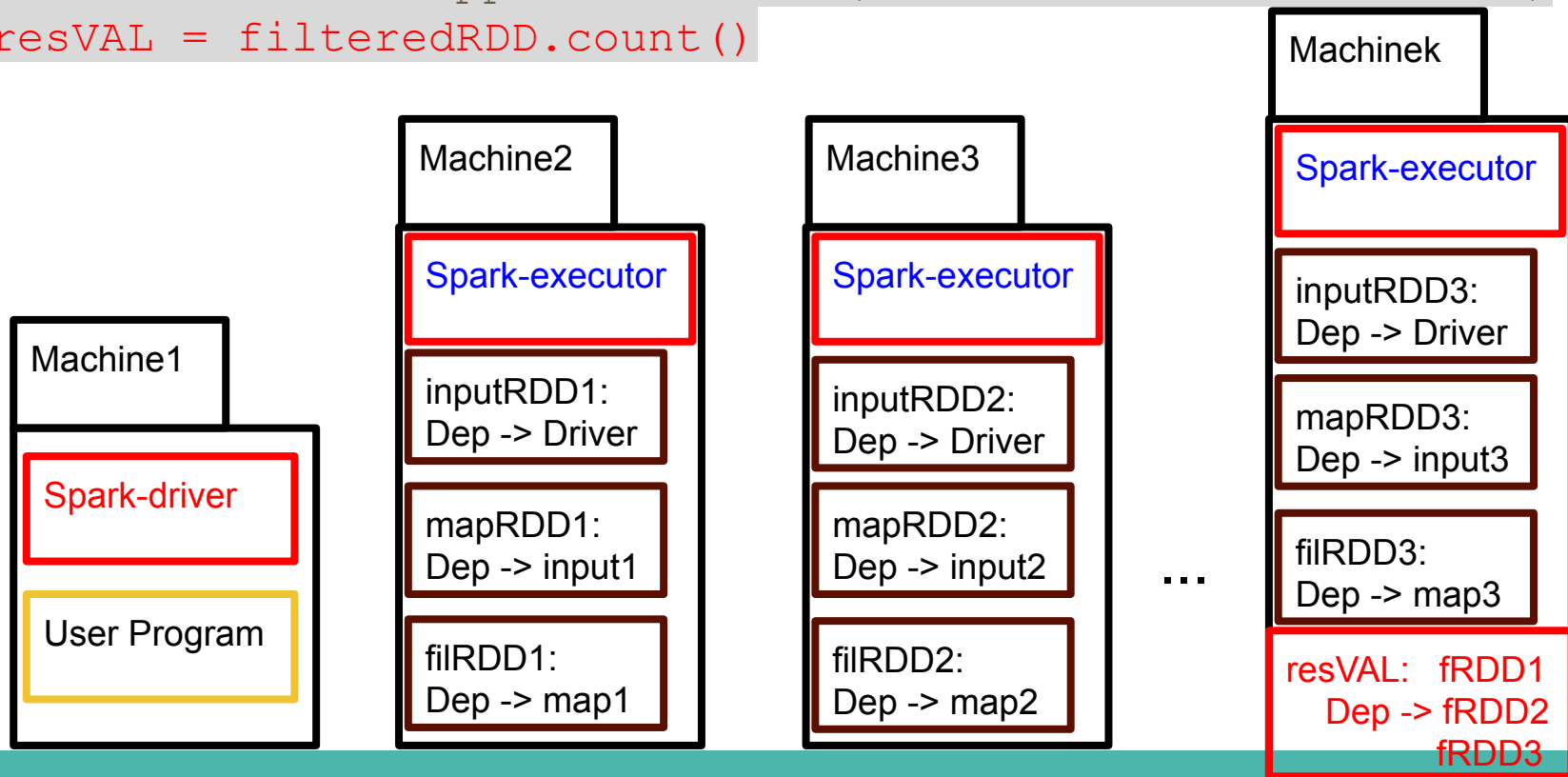


Idea 15: Private Side of RDDs



And now that I know the full lineage, computation can start lazily.

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

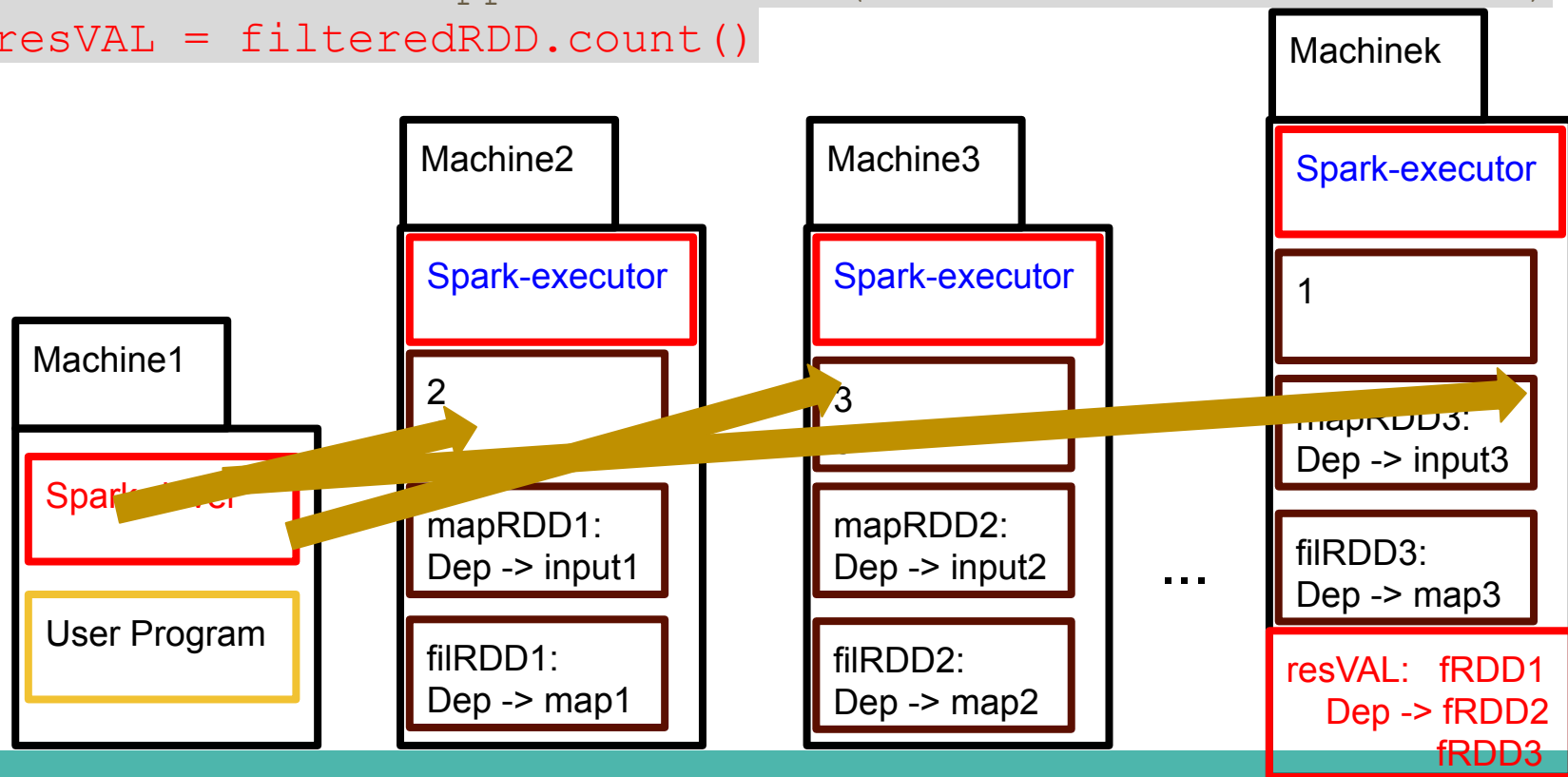


Idea 15: Private Side of RDDs



In this case, going forward!

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

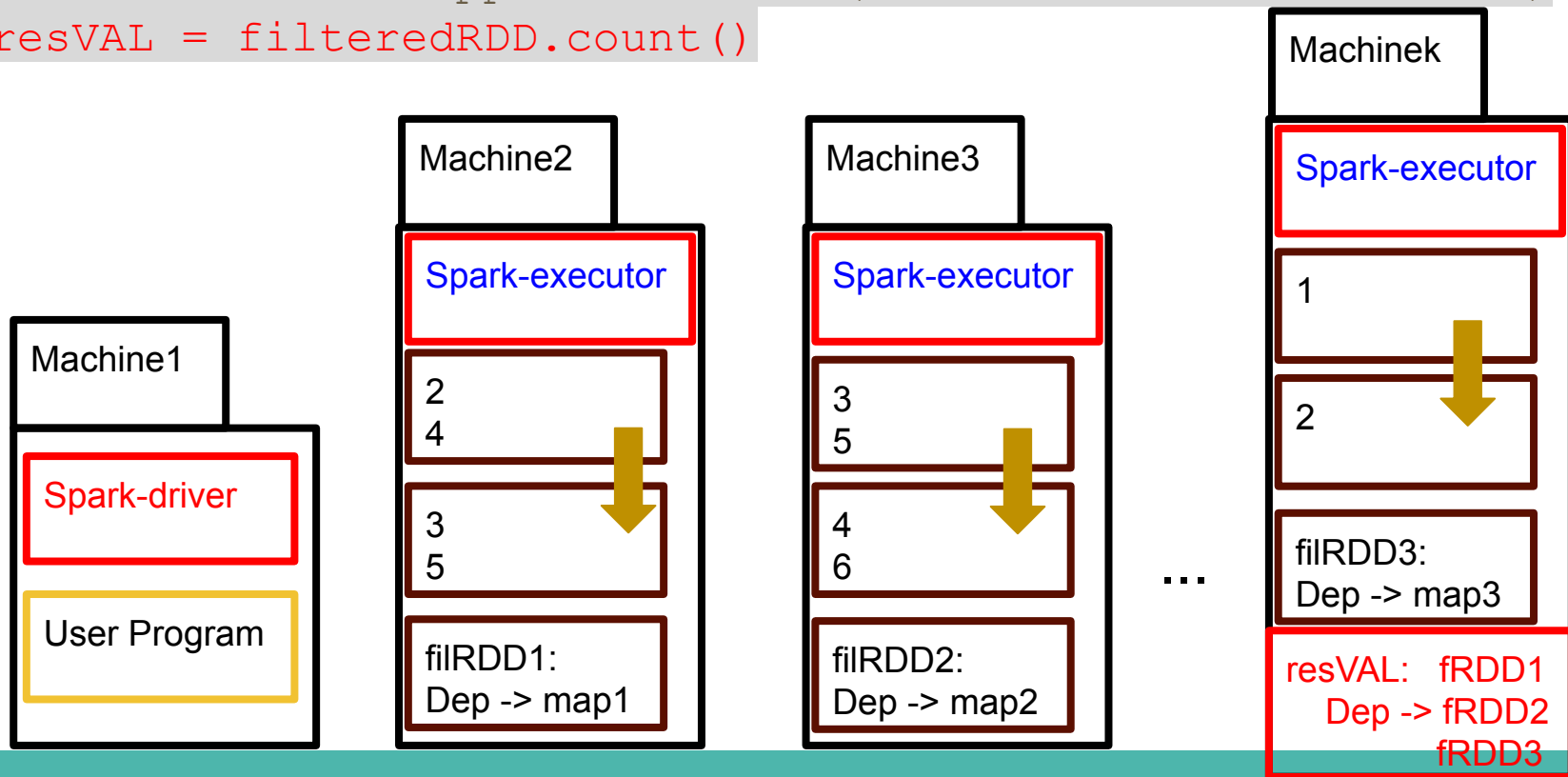


Idea 15: Private Side of RDDs



In this case, going forward!

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

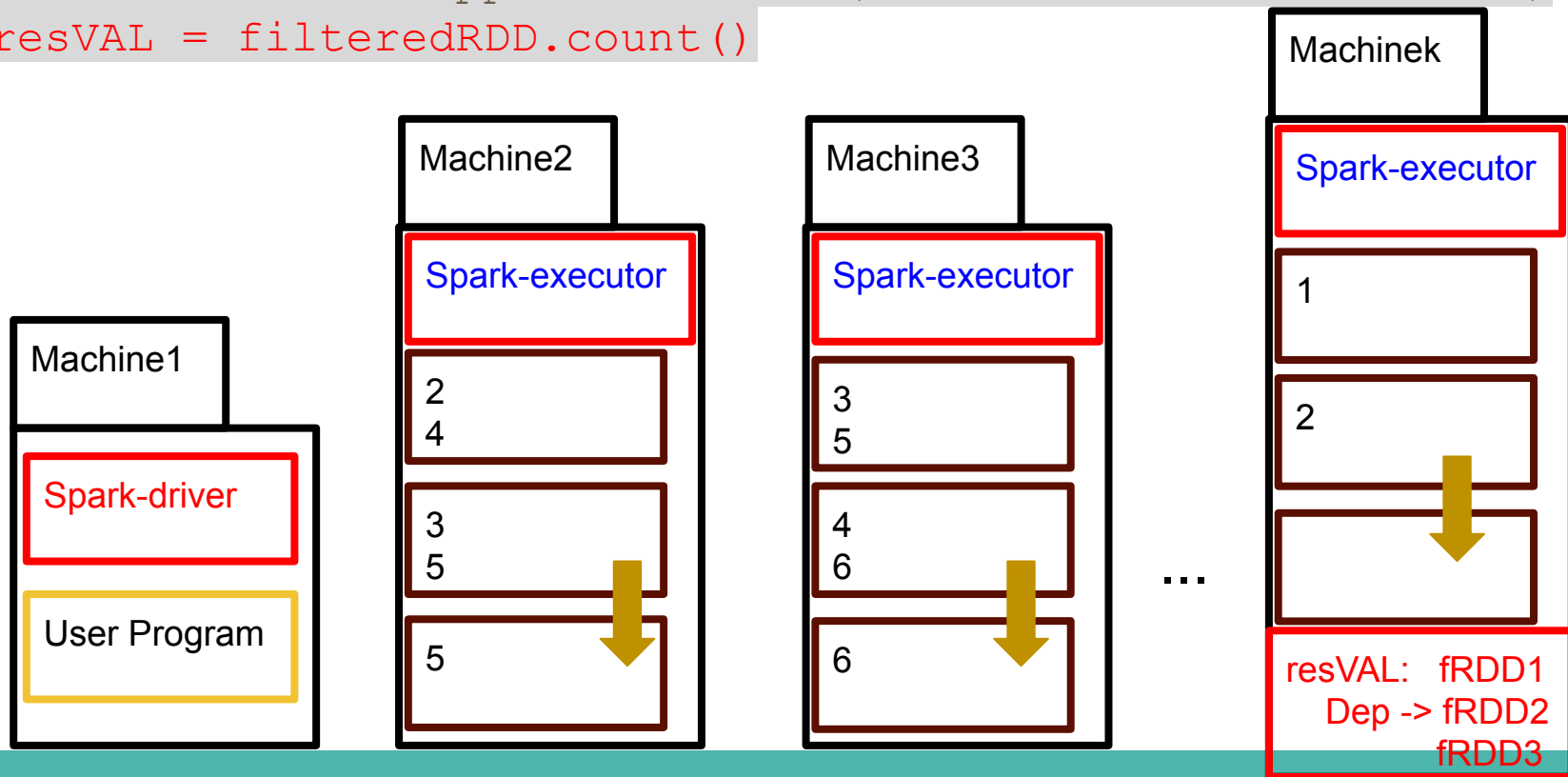


Idea 15: Private Side of RDDs



In this case, going forward!

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

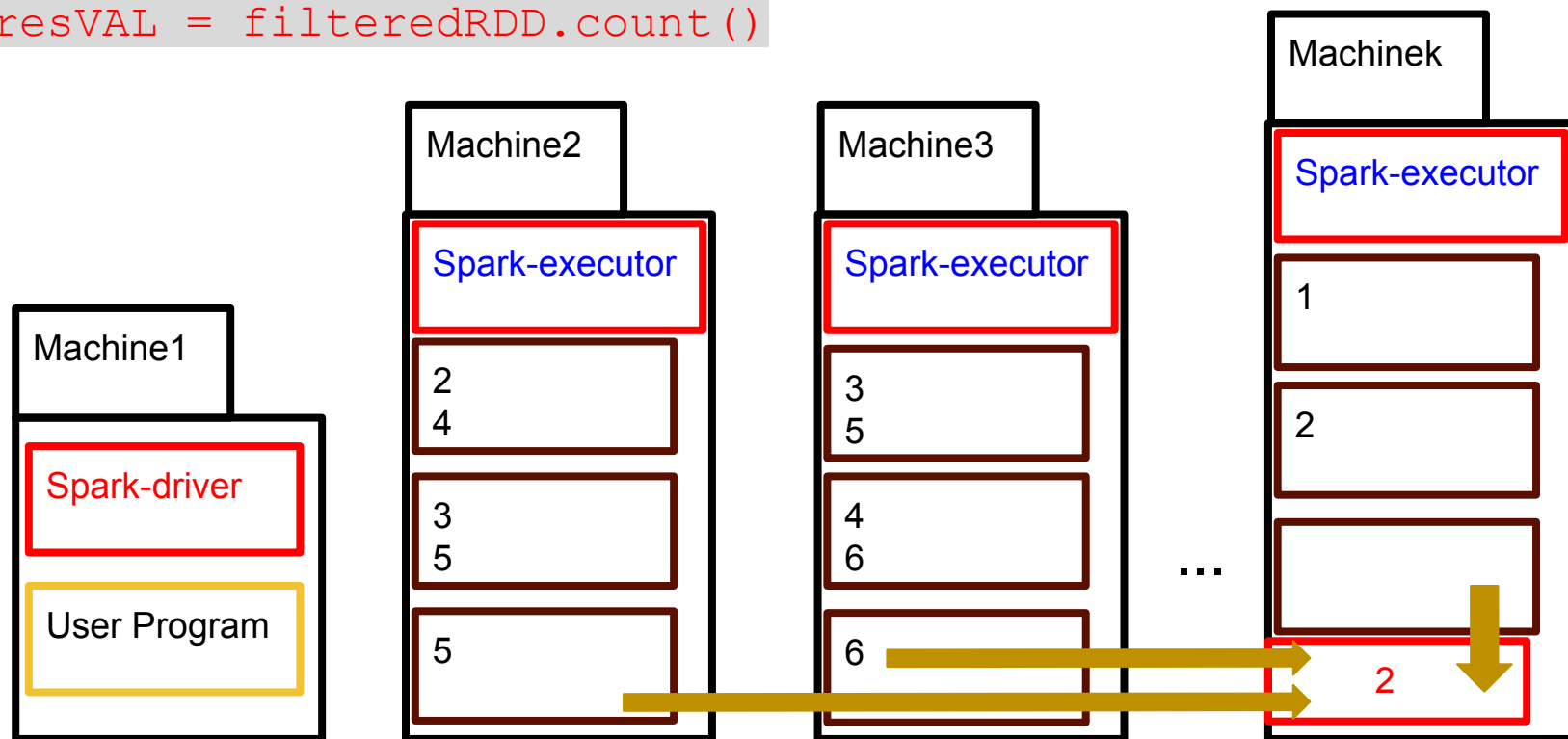


Idea 15: Private Side of RDDs



In this case, going forward!

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

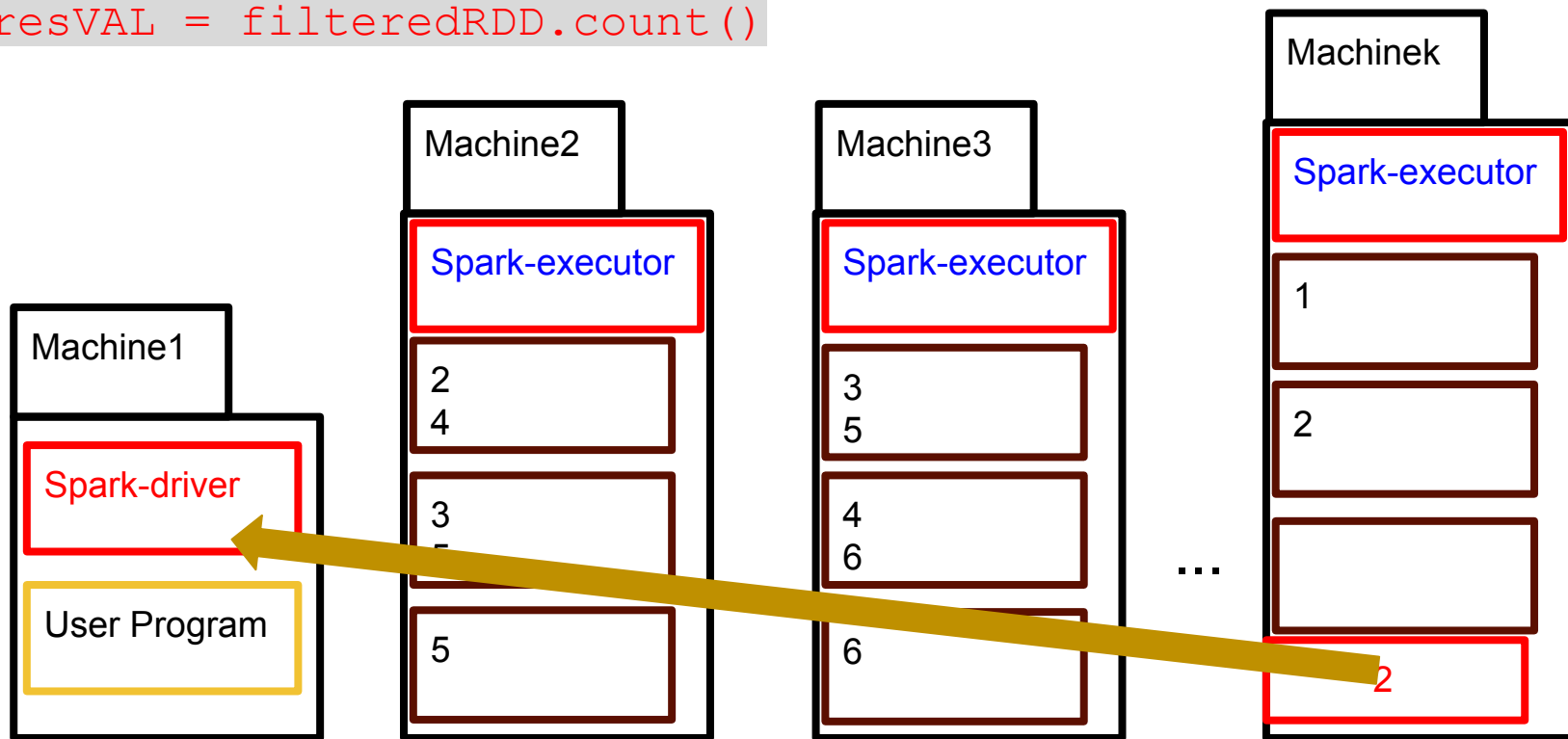


Idea 15: Private Side of RDDs



In this case, going forward!

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



Outline

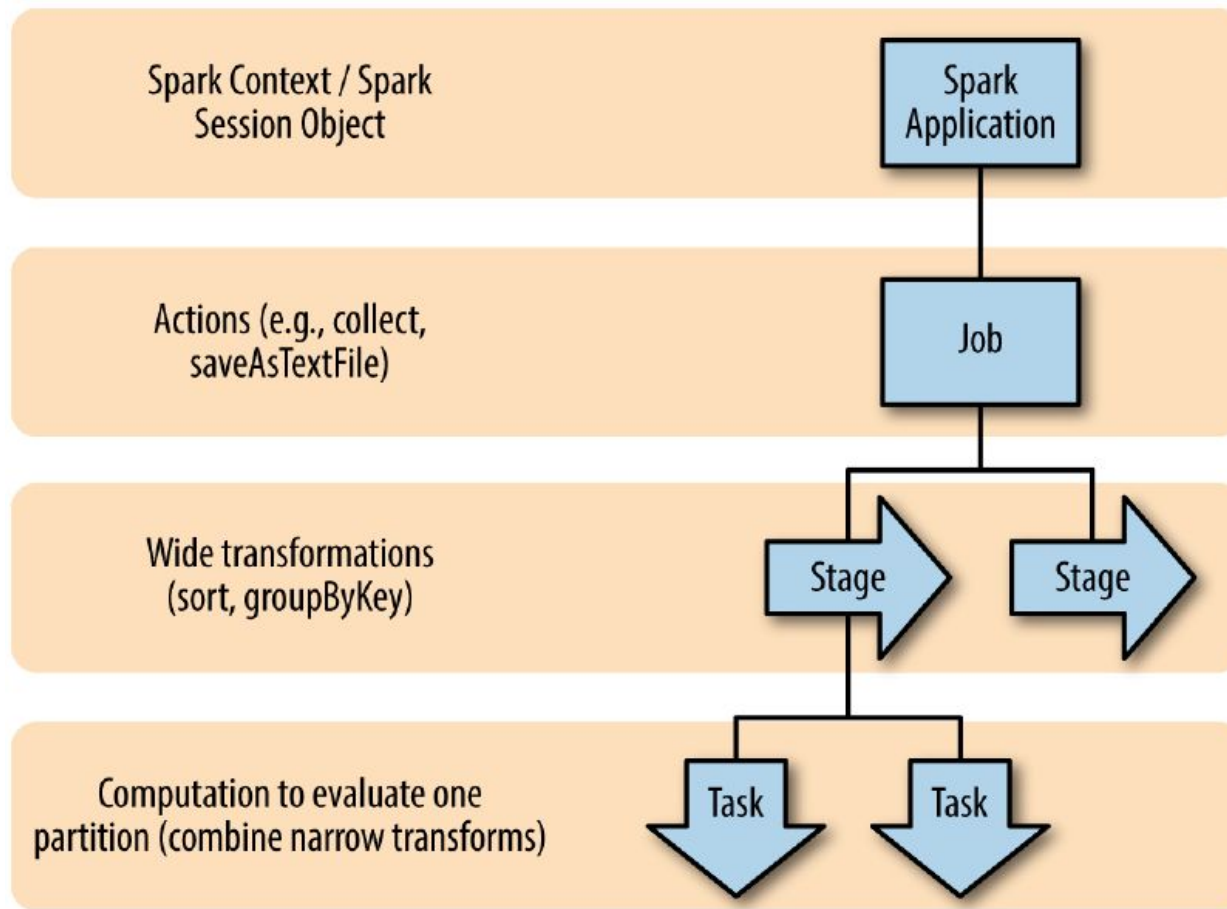
Idea 16:

Execution of a Spark User program

Idea 16: Execution of a Spark User Program

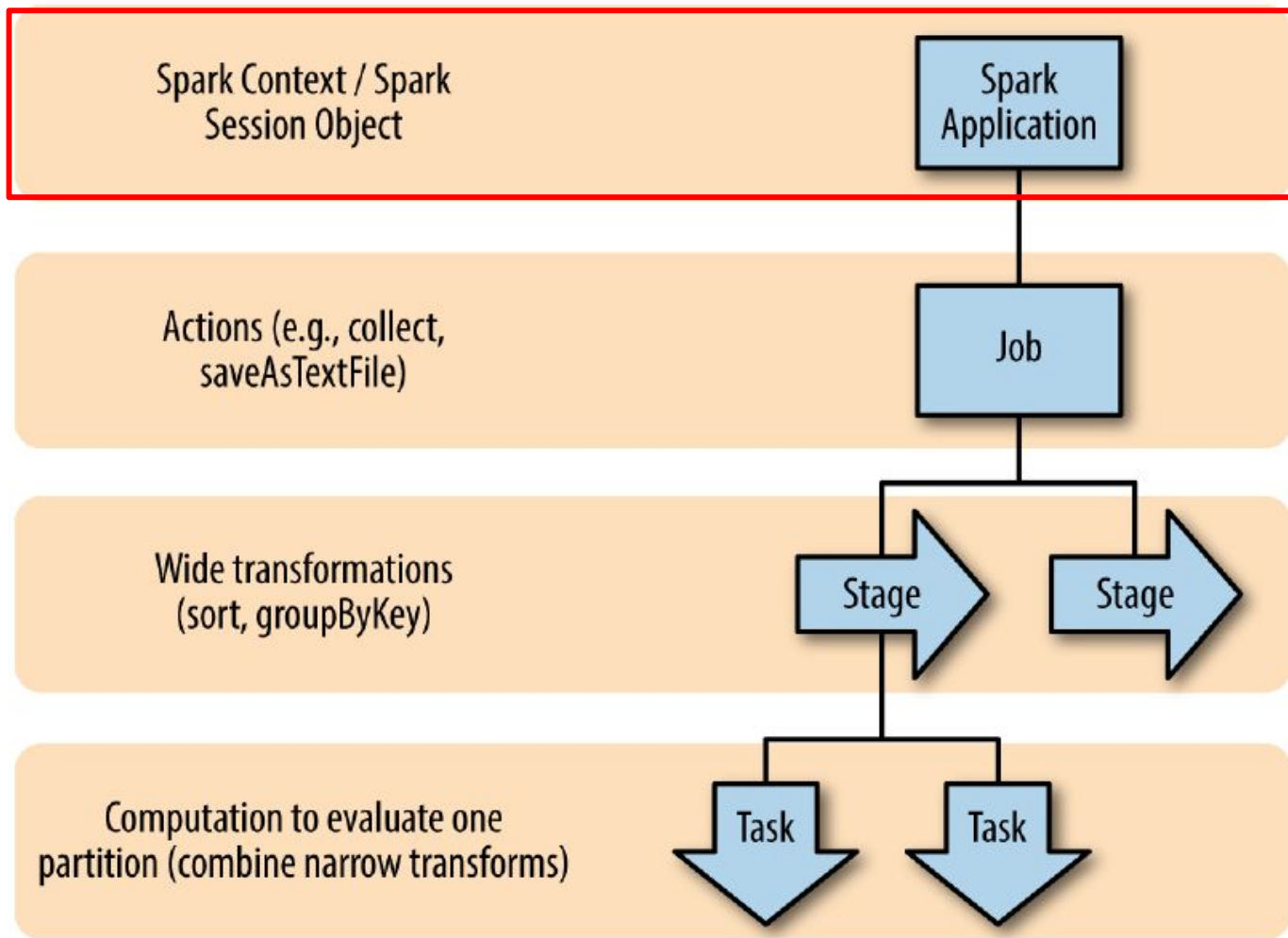
There are 4 concepts we must be familiar with in order to understand the execution of a Spark user program:

Application, Job, Stage and Task.



Idea 16: Execution of a Spark User Program

A Spark Application corresponds to a Spark User program.



Idea 16: Execution of a Spark User Program

```
sc = pyspark.SparkContext.getOrCreate()  
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)
```

Machine1

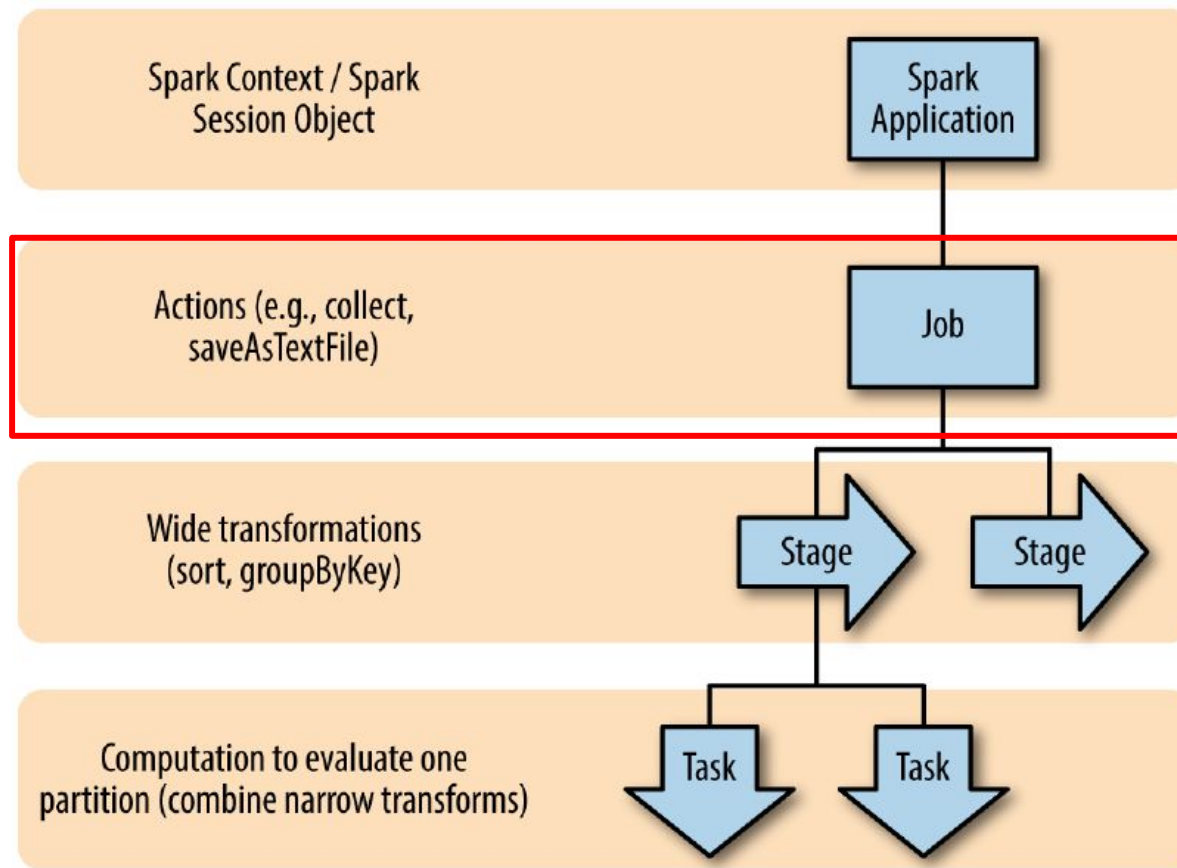
Spark-driver

User Program



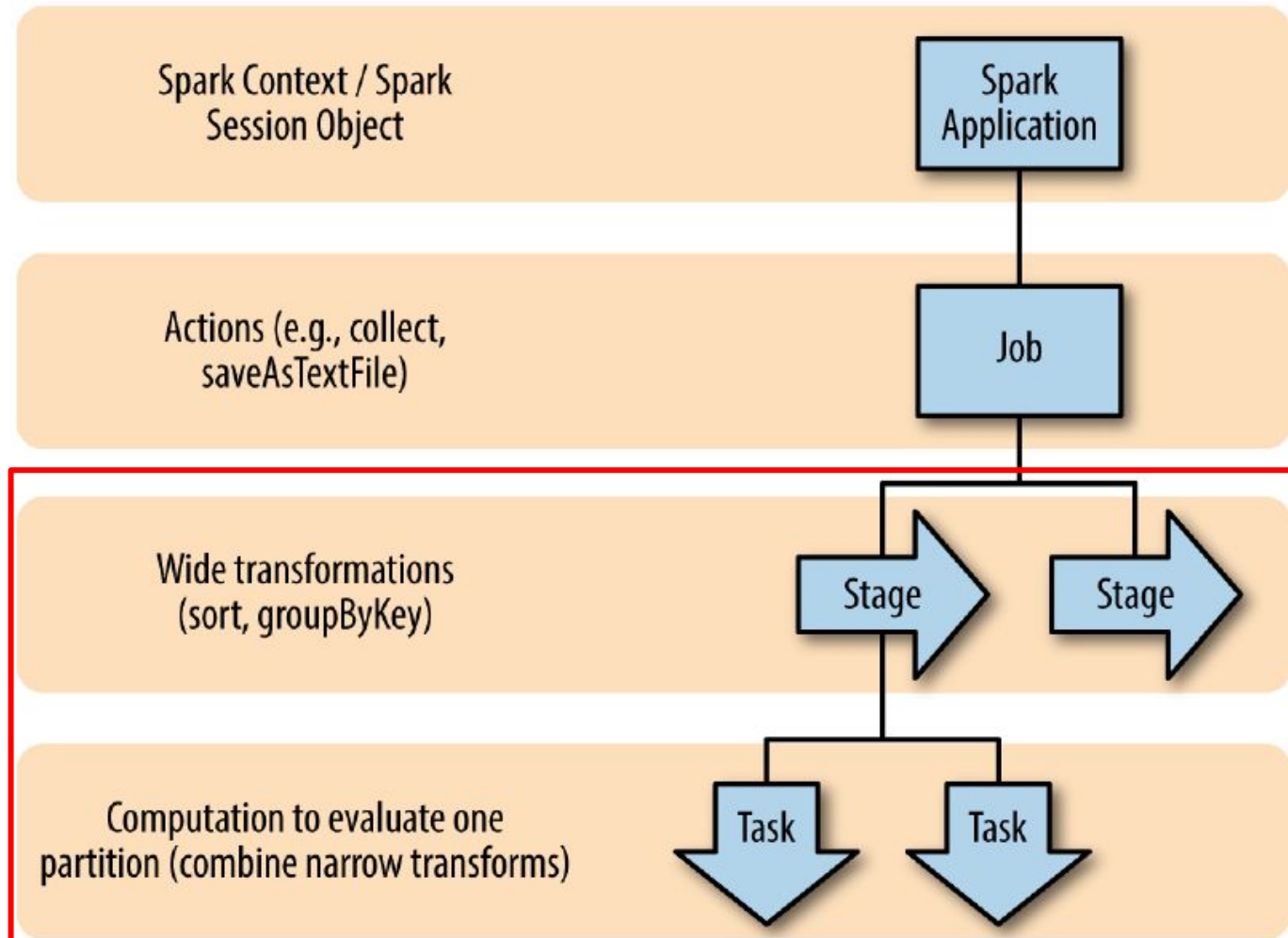
Idea 16: Execution of a Spark User Program

A Spark Job corresponds to one **action** operation in the user program. Thus, given a user program, it leads to as many jobs as actions it contains.



Idea 16: Execution of a Spark User Program

Stages and Tasks depend on the type of transformations used by the job.

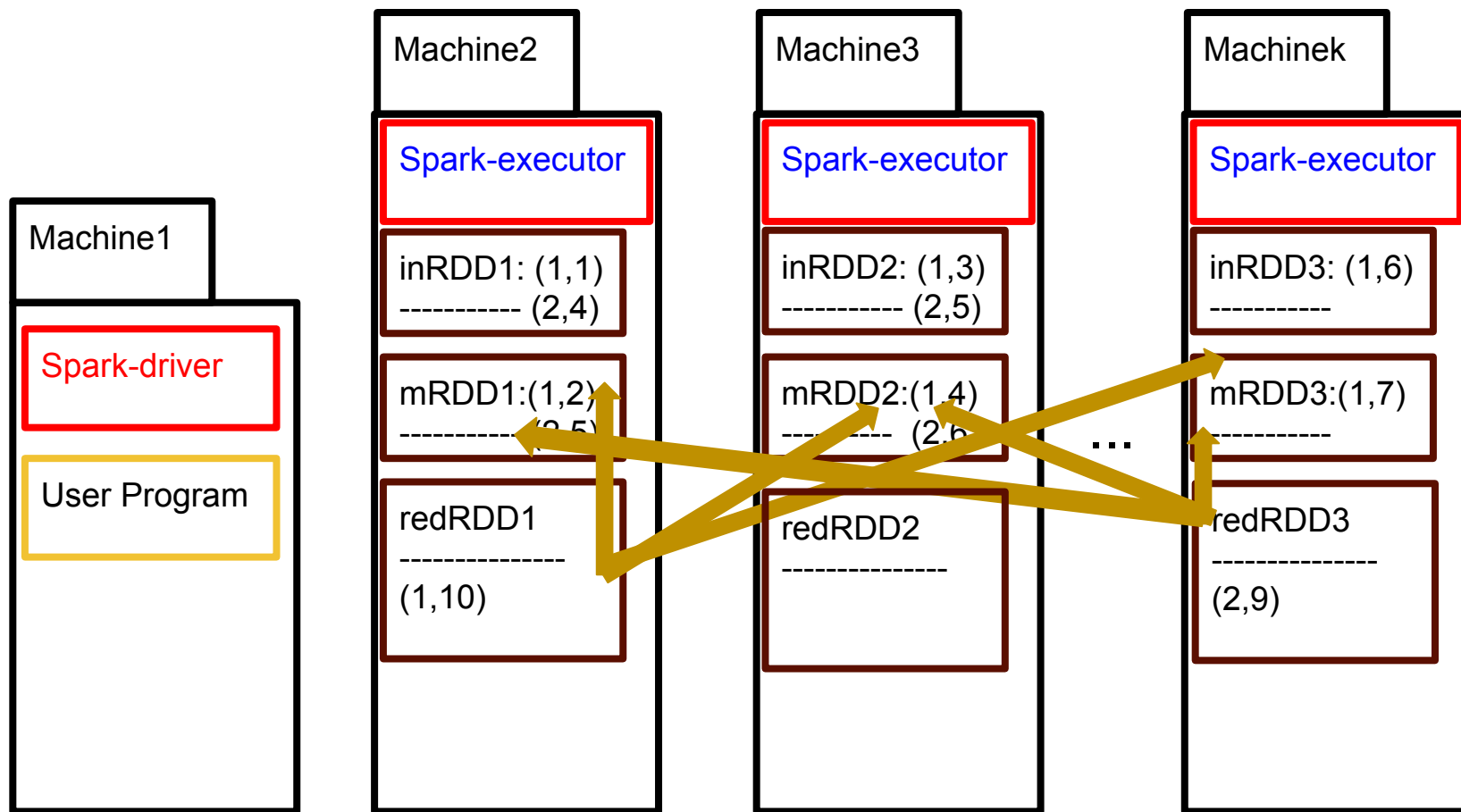


Idea 16: Execution of a Spark User Program

- Wide transformations require data to be shuffled over the network.

```
inputRDD = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])  
mapRDD = inputRDD.map( lambda elem : elem[1] + 1 )  
redRDD = mapRDD.reduceByKey( lambda x, y: x + y )  
solRDD = mapRDD.filter( lambda elem : elem[1] > 9 )
```

Idea 16: Execution of a Spark User Program



Idea 16: Execution of a Spark User Program

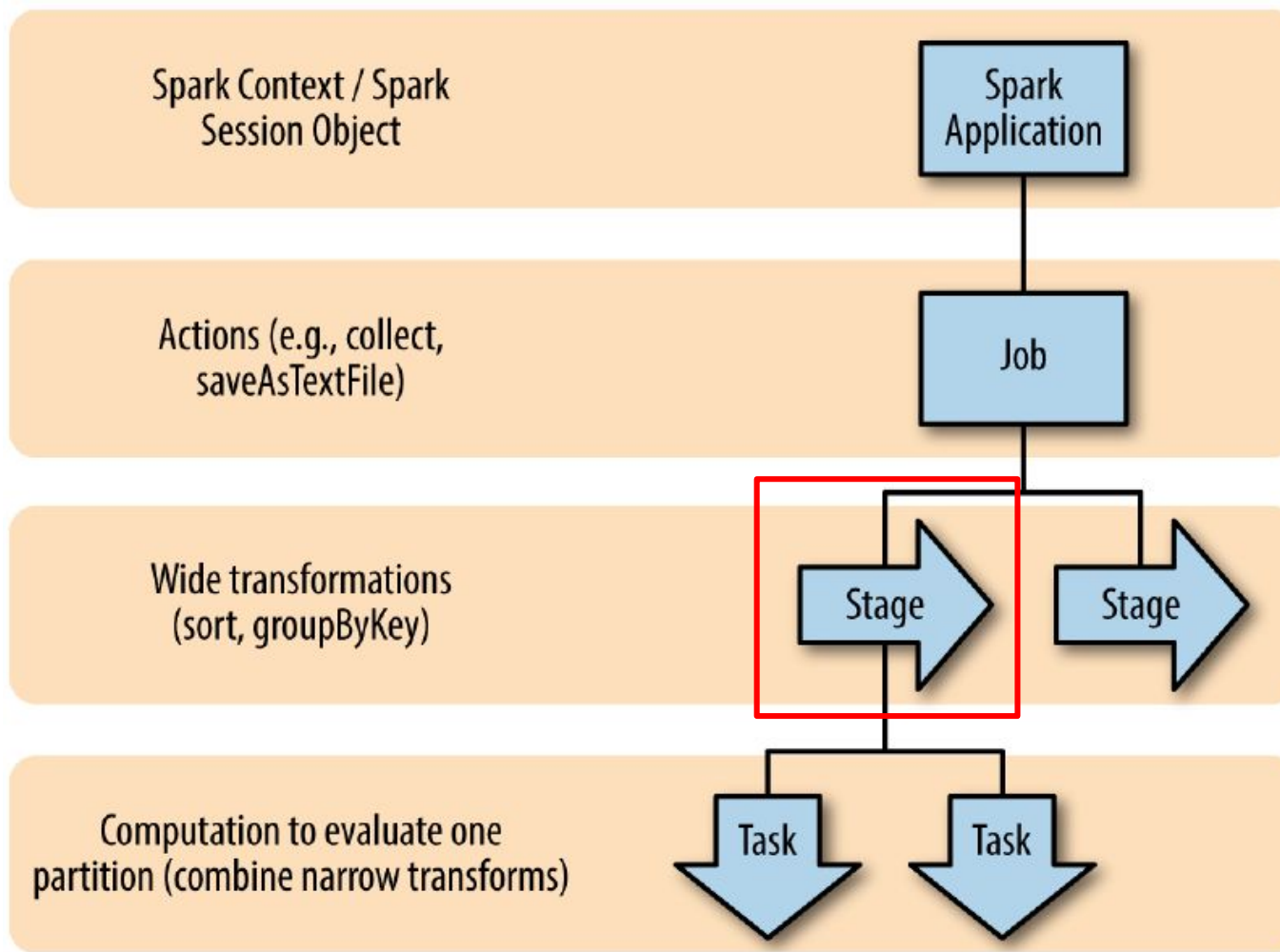
- Wide transformations require data to be shuffled over the network.

```
inputRDD = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])  
mapRDD = inputRDD.map( lambda elem : elem[1] + 1 )  
redRDD = mapRDD.reduceByKey( lambda x, y: x + y )  
solRDD = mapRDD.filter( lambda elem : elem[1] > 9 )
```

- Thus, for it to happen, all partitions must be fully computed, creating dependencies among partitions.

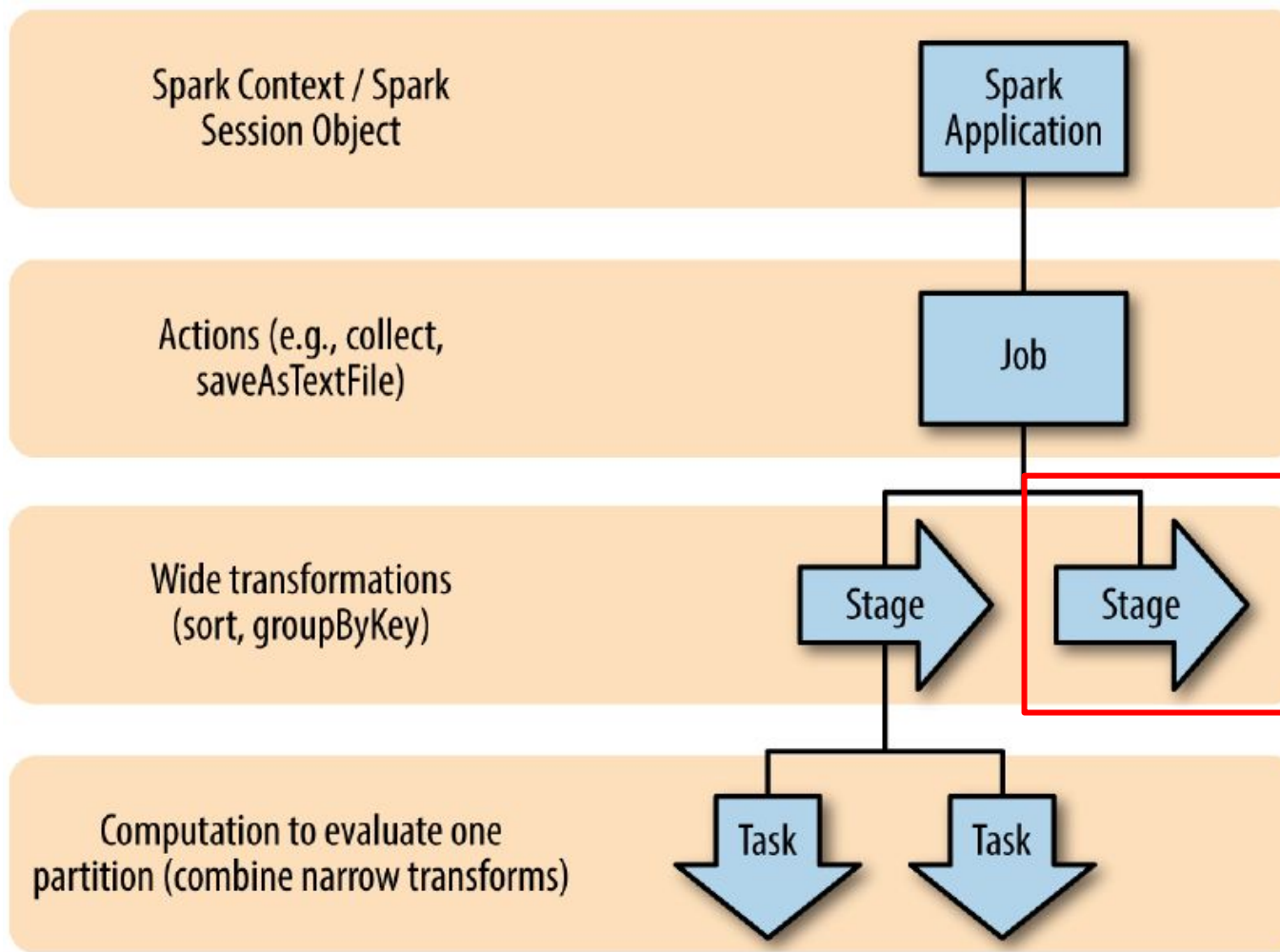
Idea 16: Execution of a Spark User Program

That's why stages are to be scheduled in sequential order.



Idea 16: Execution of a Spark User Program

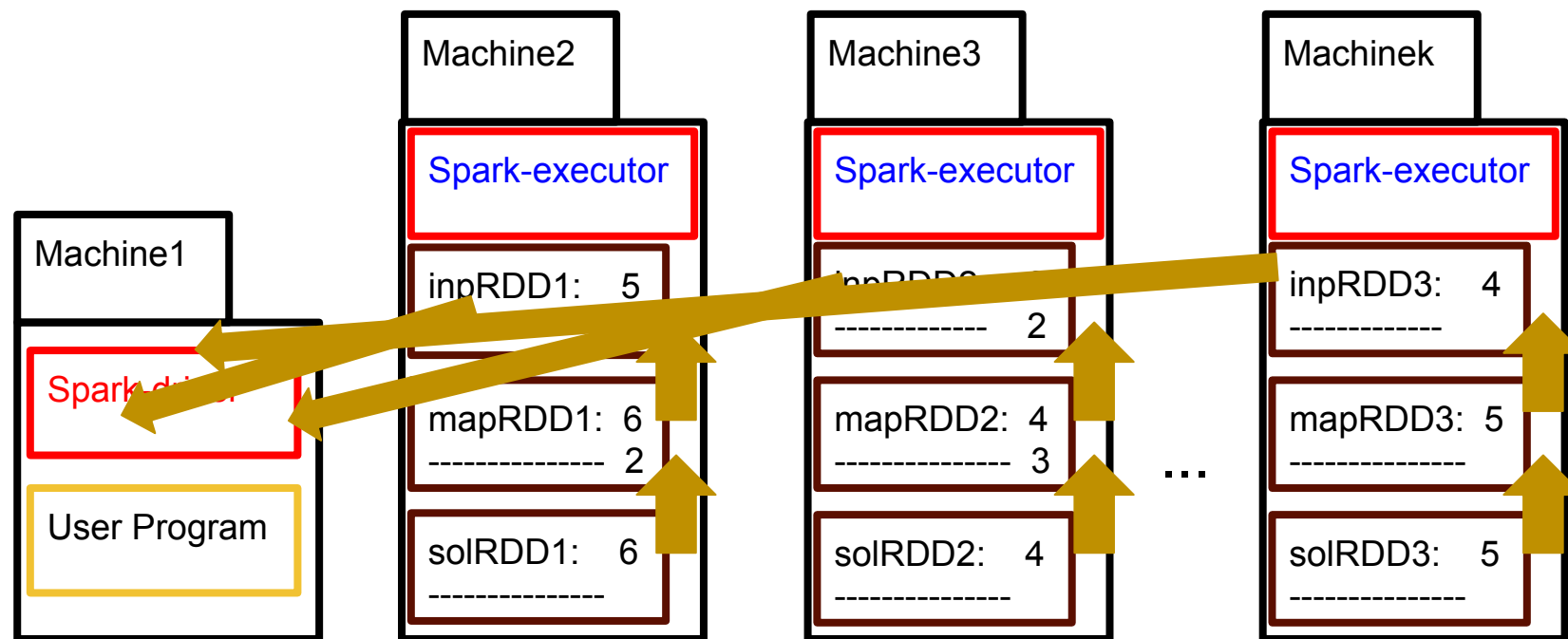
That's why stages are to be scheduled in sequential order.



Idea 16: Execution of a Spark User Program

- Within a stage, all narrow operations can be chained and executed on each partition separately, without any dependency among partitions.

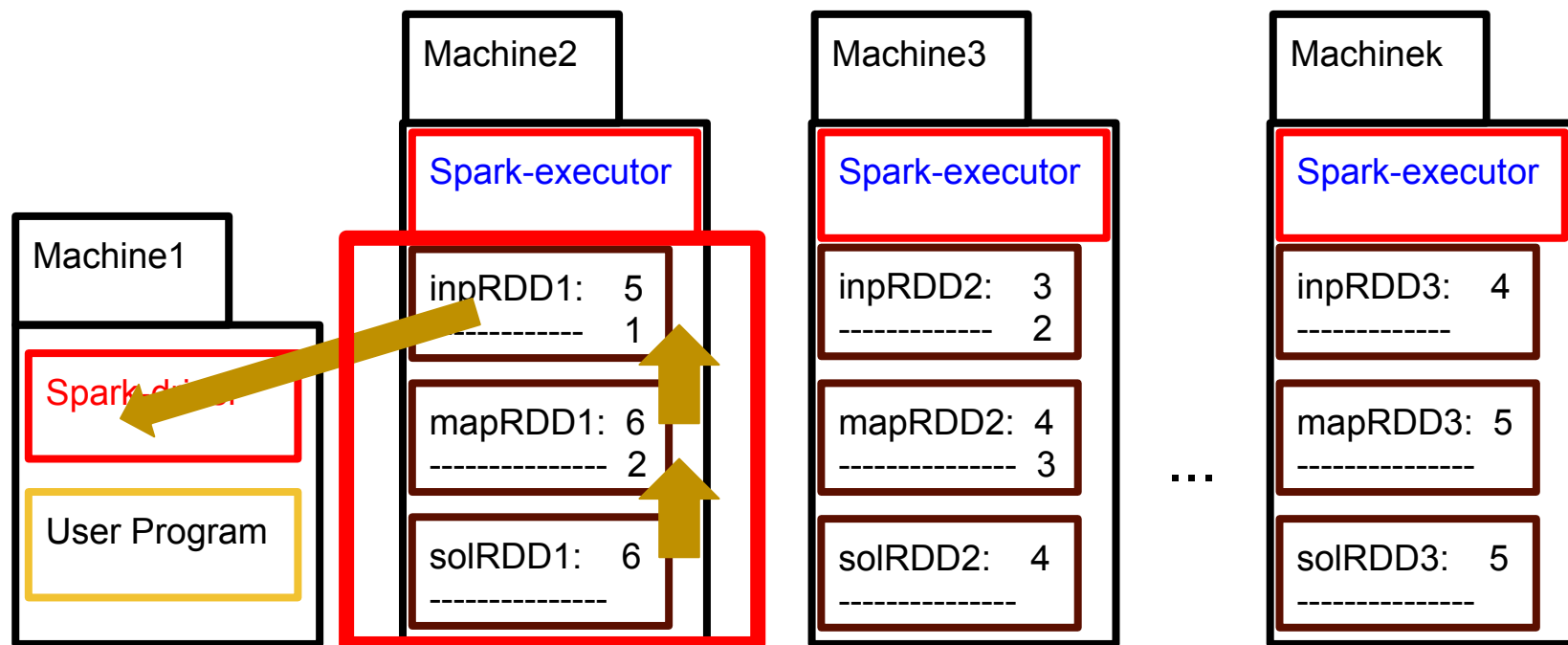
```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5 ] )  
mapRDD = inputRDD.map( lambda elem : elem + 1 )  
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Idea 16: Execution of a Spark User Program

- Within a stage, all narrow operations can be chained and execution on each partition separately, without any dependency among partitions.

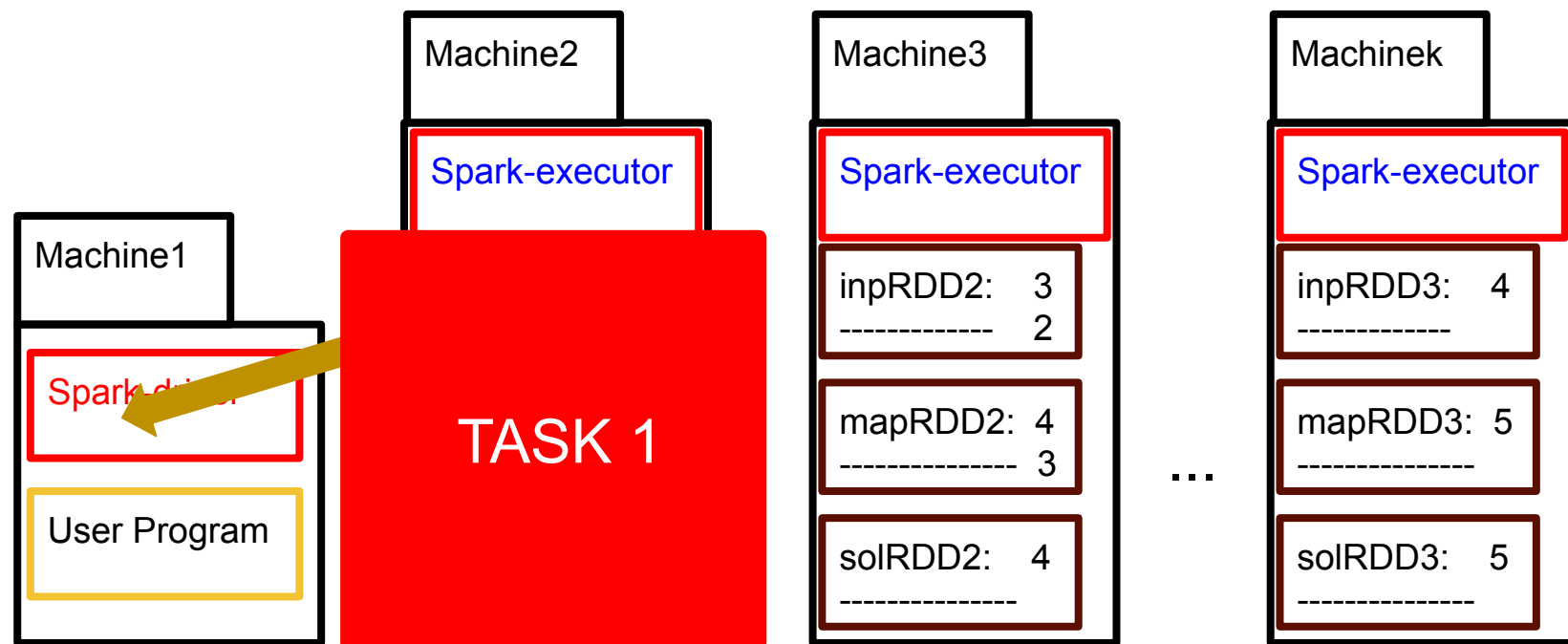
```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5 ] )  
mapRDD = inputRDD.map( lambda elem : elem + 1 )  
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Idea 16: Execution of a Spark User Program

- Within a stage, all narrow operations can be chained and execution on each partition separately, without any dependency among partitions.

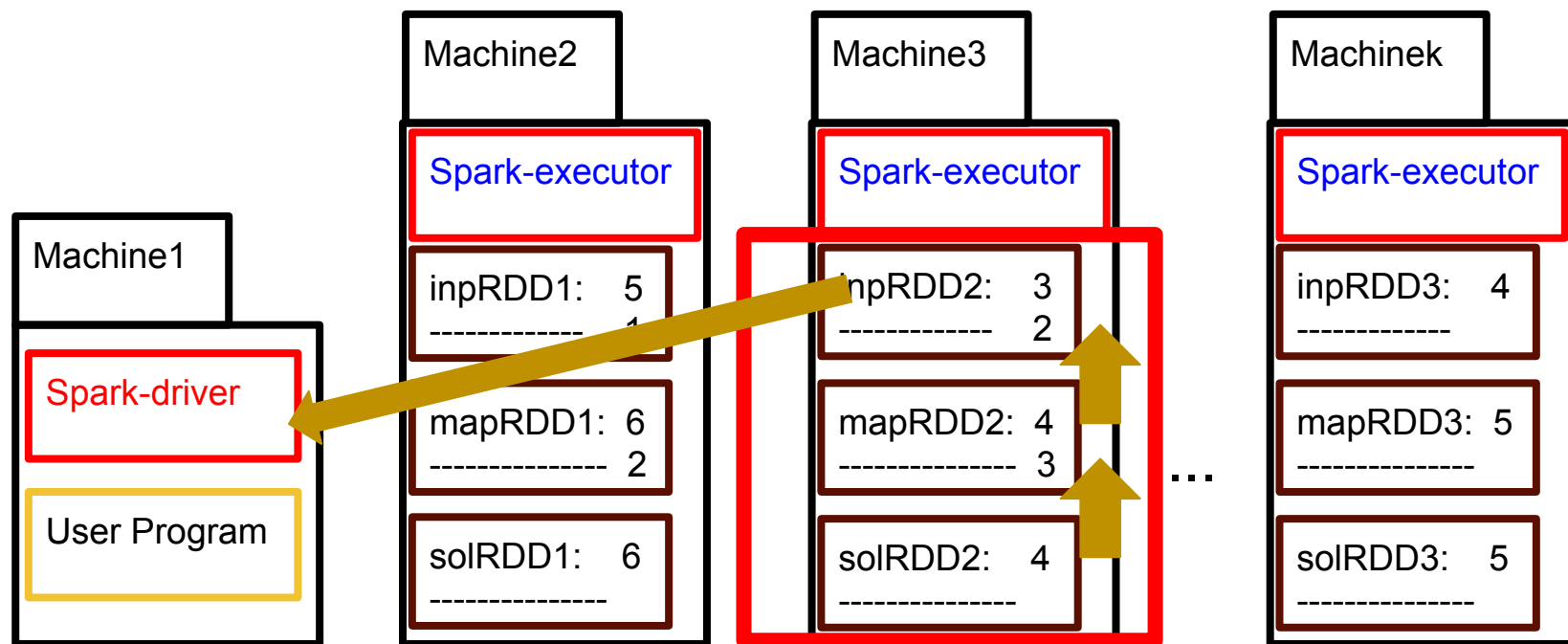
```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5 ] )  
mapRDD = inputRDD.map( lambda elem : elem + 1 )  
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Idea 16: Execution of a Spark User Program

- Within a stage, all narrow operations can be chained and execution on each partition separately, without any dependency among partitions.

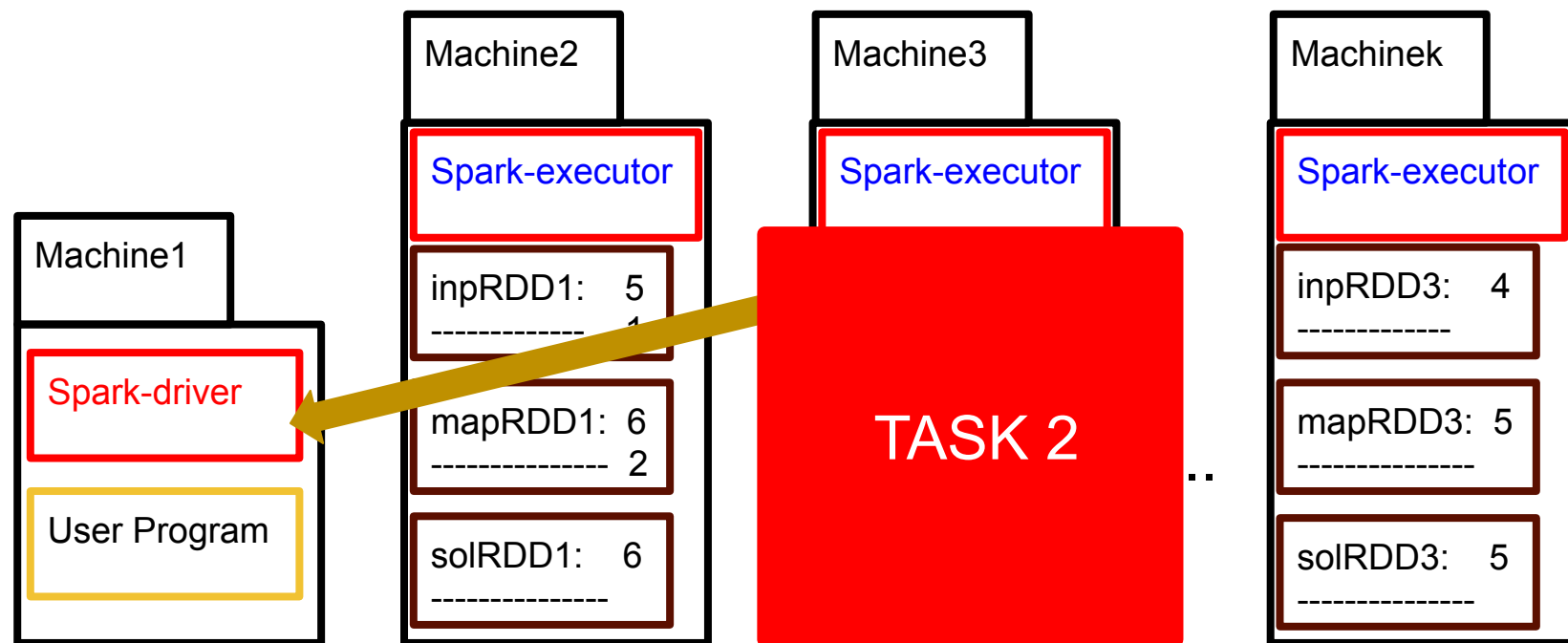
```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5 ] )  
mapRDD = inputRDD.map( lambda elem : elem + 1 )  
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Idea 16: Execution of a Spark User Program

- Within a stage, all narrow operations can be chained and execution on each partition separately, without any dependency among partitions.

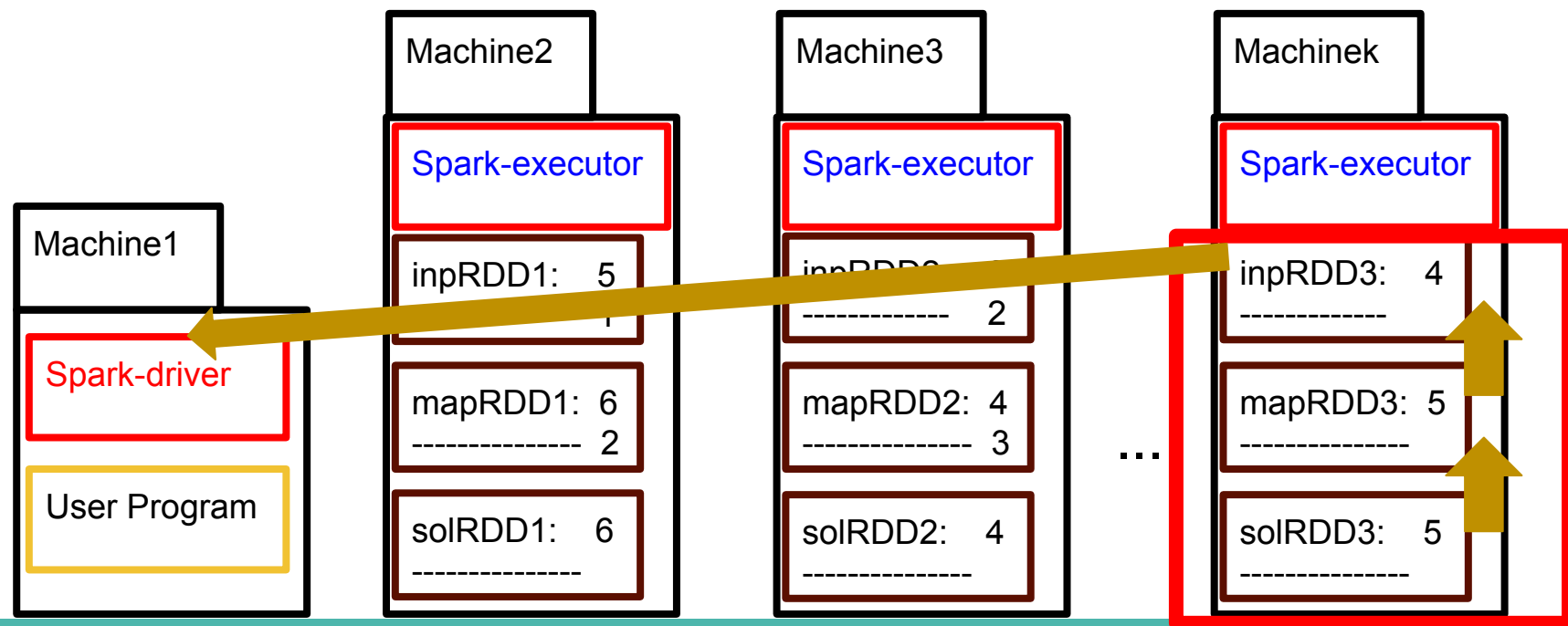
```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5 ] )  
mapRDD = inputRDD.map( lambda elem : elem + 1 )  
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Idea 16: Execution of a Spark User Program

- Within a stage, all narrow operations can be chained and execution on each partition separately, without any dependency among partitions.

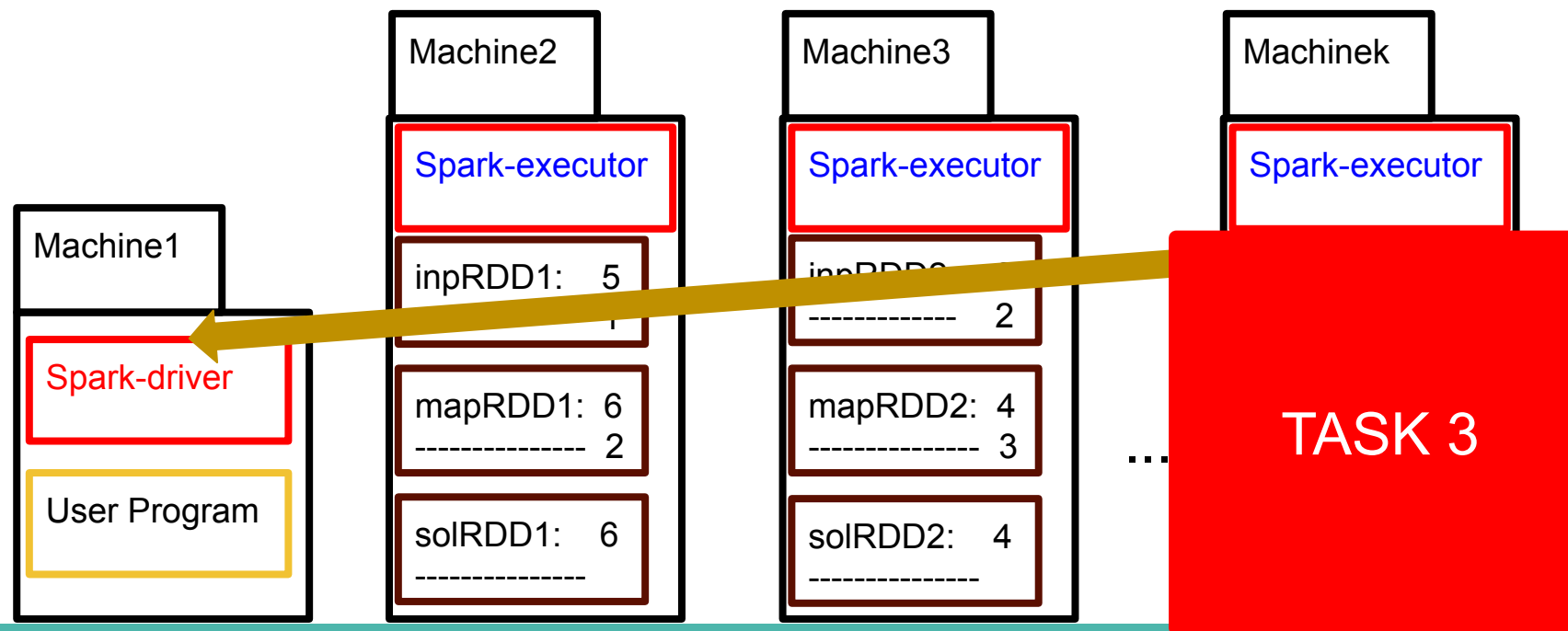
```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5 ] )  
mapRDD = inputRDD.map( lambda elem : elem + 1 )  
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Idea 16: Execution of a Spark User Program

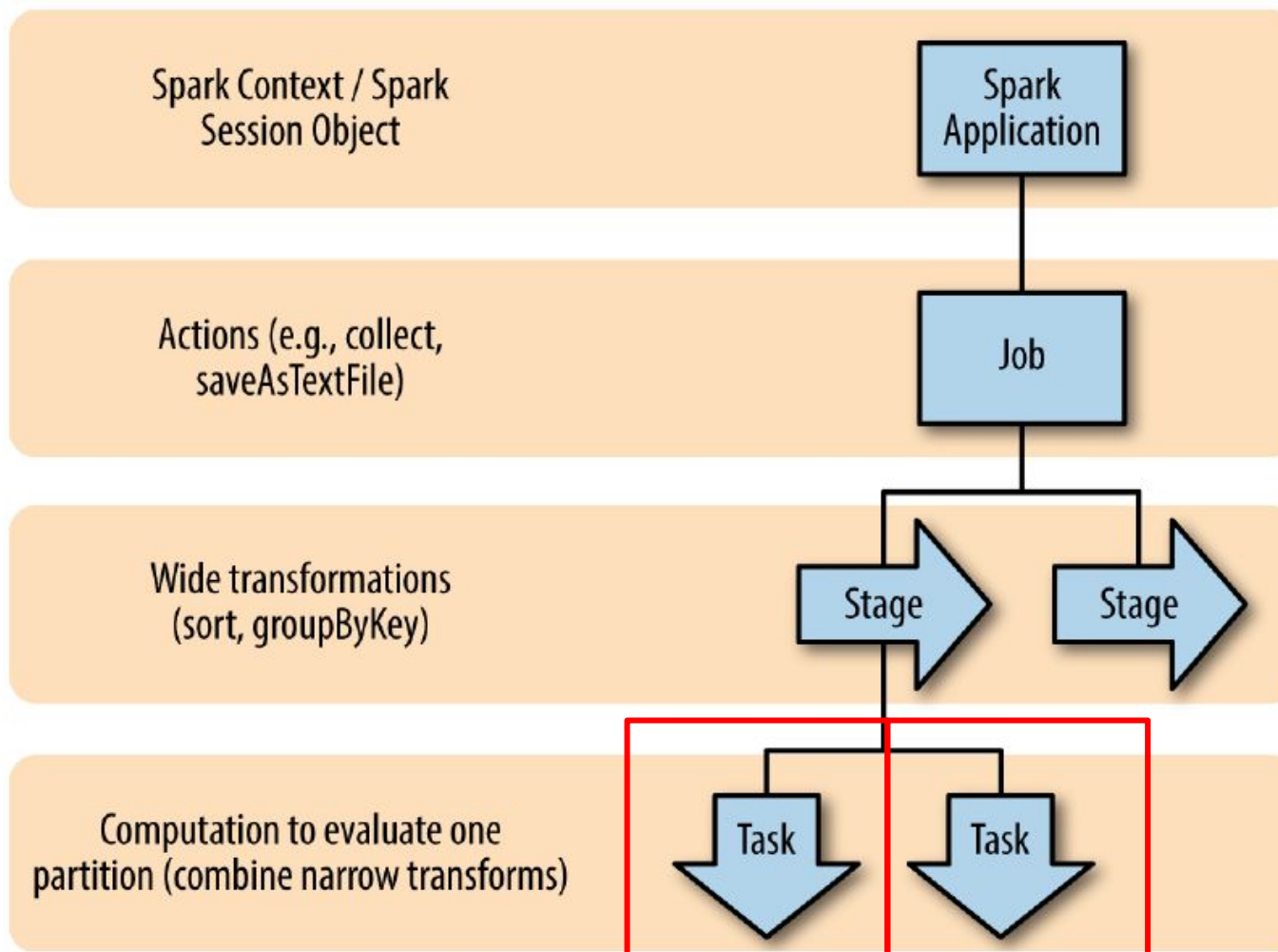
- Within a stage, all narrow operations can be chained and execution on each partition separately, without any dependency among partitions.

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



Idea 16: Execution of a Spark User Program

That's why within a stage, tasks can run in parallel.



Outline

Idea 17:

Spark SQL
A Higher-Level API

Idea 17: Spark SQL - A Higher Level API

In the same way Spark Core makes possible *functional Spark* via the ADT Resilient Distributed Datasets (RDD)...

...Spark SQL makes possible *structured Spark* via the ADTs DataFrame (DF).

Idea 17: Spark SQL - A Higher Level API

Let's go with the [DataFrame ADT public side](#):

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

Idea 17: Spark SQL - A Higher Level API

Let's go with the [DataFrame ADT public side](#):

1. **What** defines or specifies the type of data being offered?
- A DataFrame defines or specifies an...
 1. **indivisible** (logically presented as an atomic variable)
 2. **structured**, in the sense of having a fixed number of fields, each of them of a concrete data type T.
 3. **lazily-evaluated** (only computed if required, and as much as required)
 4. **non-mutable** (cannot change type nor value)

...collection of elements.

Idea 17: Spark SQL - A Higher Level API

Let's go with the [DataFrame ADT public side](#):

1. **What** defines or specifies the type of data being offered?
 - A DataFrame defines or specifies an...
 1. **indivisible** (logically presented as an atomic variable)
 2. **structured**, in the sense of having a fixed number of fields, each of them of a concrete data type T.
 3. **lazily-evaluated** (only computed if required, and as much as required)
 4. **non-mutable** (cannot change type nor value)
- ...collection of elements.**

You can think of it as:

- A table in a relational database, in the sense that each Row follows the schema.
- A collection in a NoSQL document oriented database, in the sense that the content of the collection is distributed.

Idea 17: Spark SQL - A Higher Level API

Let's go with the `DataFrame/Dataset` ADT public side:

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

Idea 17: Spark SQL - A Higher Level API

Let's go with the [DataFrame/Dataset ADT public side](#):

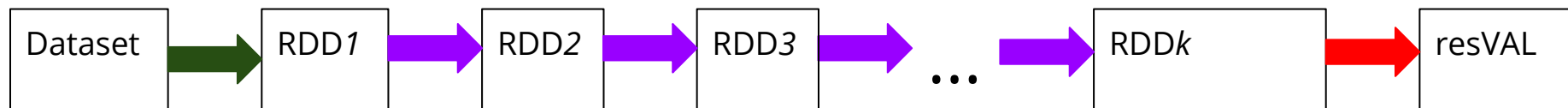
2. **What** are the operations that can be performed with this data?
Specify each of them.
- DataFrames provide a Catalog of Domain Specific Language (DSL) operators, which can be classified into **Creation**, **Transformation**, **Persistence** and **Actions**, as it was the case for the Spark Core RDD-based primitives.

Spark  SQL



Idea 17: Spark SQL - A Higher Level API

But, whereas RDDs provided a small set of primitives, each of them supporting the application of very general functions...



```
newRDD = inputRDD.map( f ) with any  $f: a \rightarrow b$ 
```

```
resVAL = inputRDD.reduce( f ) with any  $f: a \rightarrow a \rightarrow a$ 
```

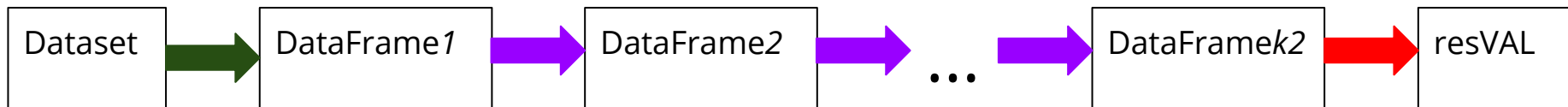
```
resVAL = inputRDD.aggregate( accum, f1, f2 )  
With any  $accum:: b, f1: b \rightarrow a \rightarrow b, f2: b \rightarrow b \rightarrow b$ 
```

```
newRDD = inputRDD.reduceByKey( f ) with any  $f: a \rightarrow a$ 
```

```
newRDD = inputRDD.combineByKey( f1, f2, f3 )  
with any  $f1: a \rightarrow b, f2: b \rightarrow a \rightarrow b$  and  $f3: b \rightarrow b \rightarrow b$ 
```

Idea 17: Spark SQL - A Higher Level API

Now, DFs provide an extensive set of DSL operators, each of them very restrictive in what expressions they accept.



Idea 17: Spark SQL - A Higher Level API

- These operators represent the most common patterns present in a data analysis. Moreover, each new version of Spark includes new DSL operators.
- If something cannot be expressed, DFs allow to revert back to RDDs, as well as to provide your own User-Defined Expression (at the price of Spark not being able to reason internally with it).

Idea 17: Spark SQL - A Higher Level API

- These operators represent the most common patterns present in a data analysis. Moreover, each new version of Spark includes new DSL operators.
- If something cannot be expressed, DFs allow to revert back to RDDs, as well as to provide your own User-Defined Expression (at the price of Spark not being able to reason internally with it).
- These DSL operators are indeed very restrictive with their inner expressions, so that Spark can understand/reason with/perform a semantic analysis of them.

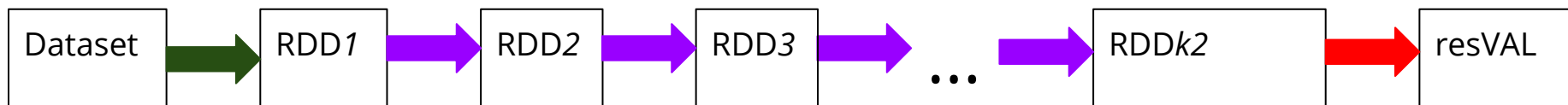
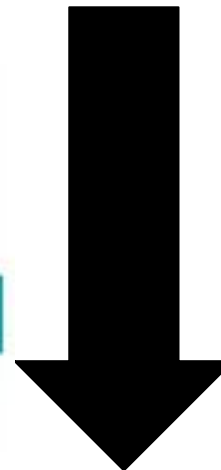
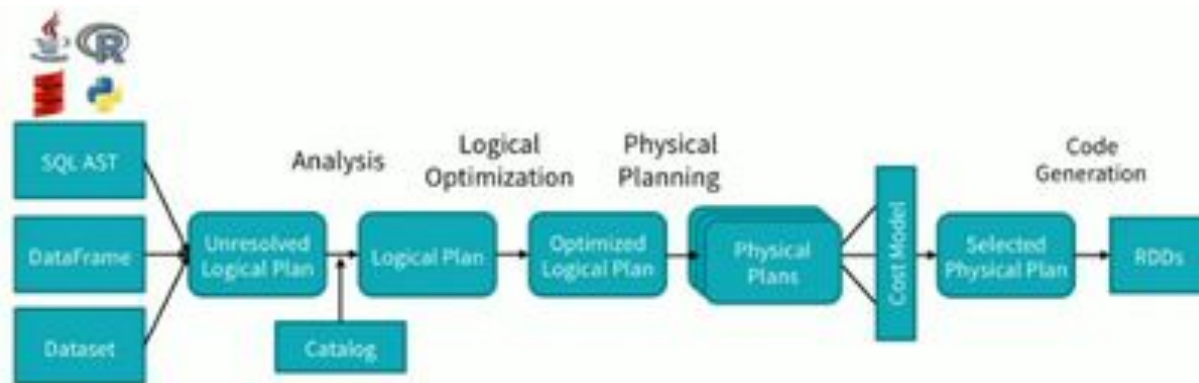
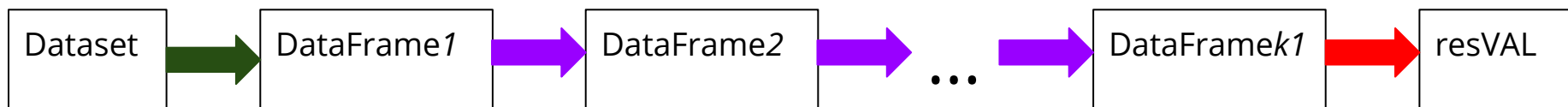
Outline

Idea 18:

Translation from Spark SQL to Spark Core

Idea 18: Translation from Spark SQL to Spark Core

For a Spark DF-based program to be executed, it has to be firstly translated into an equivalent Spark Core RDD-based program.



Idea 18: Translation from Spark SQL to Spark Core

This translation process includes the following stages:

1. Logical Query Plan.
2. Optimised Logical Query Plan.
3. Code Generation.

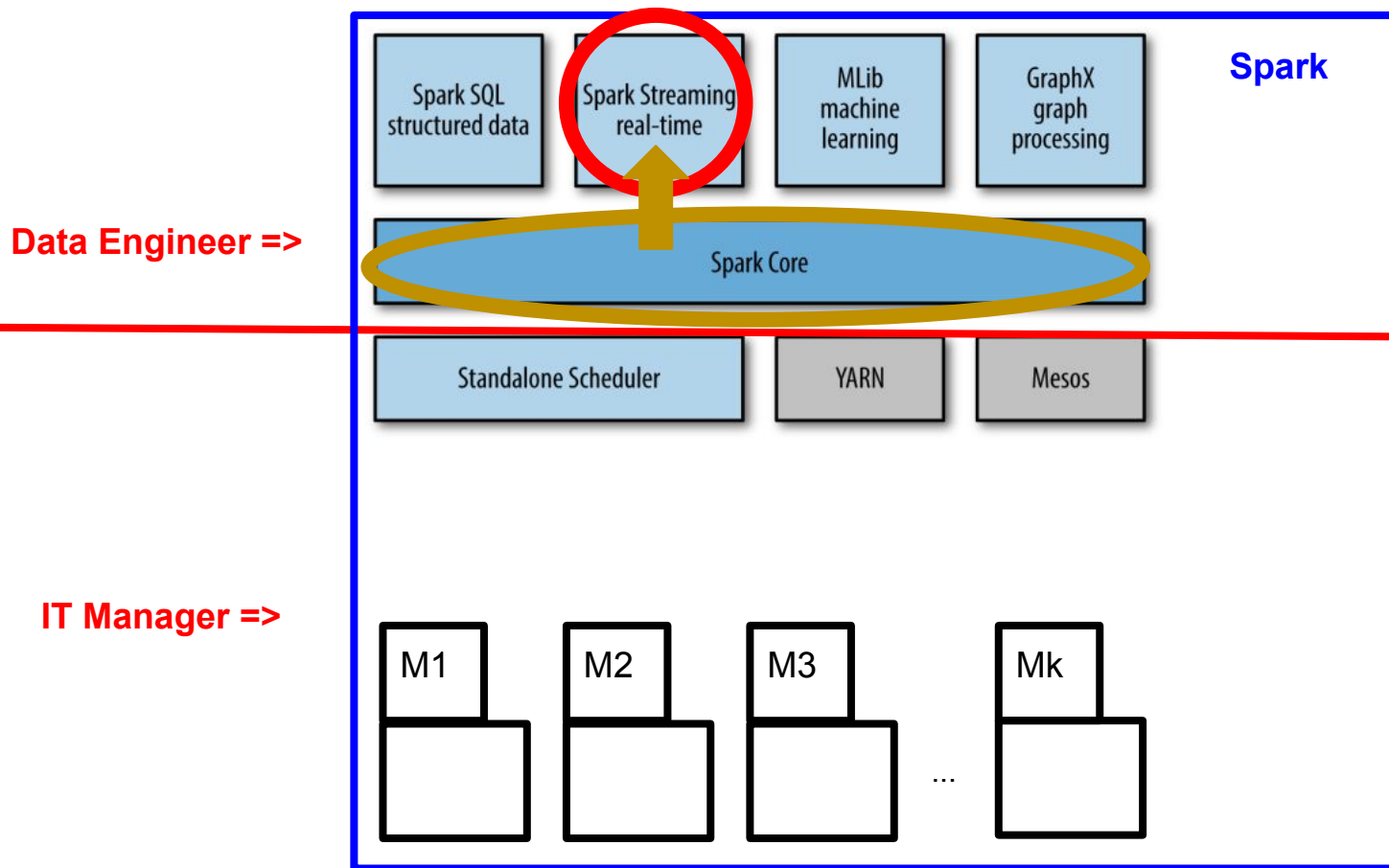
Outline

Idea 19:

Extending Spark Core
with Streaming Functionality

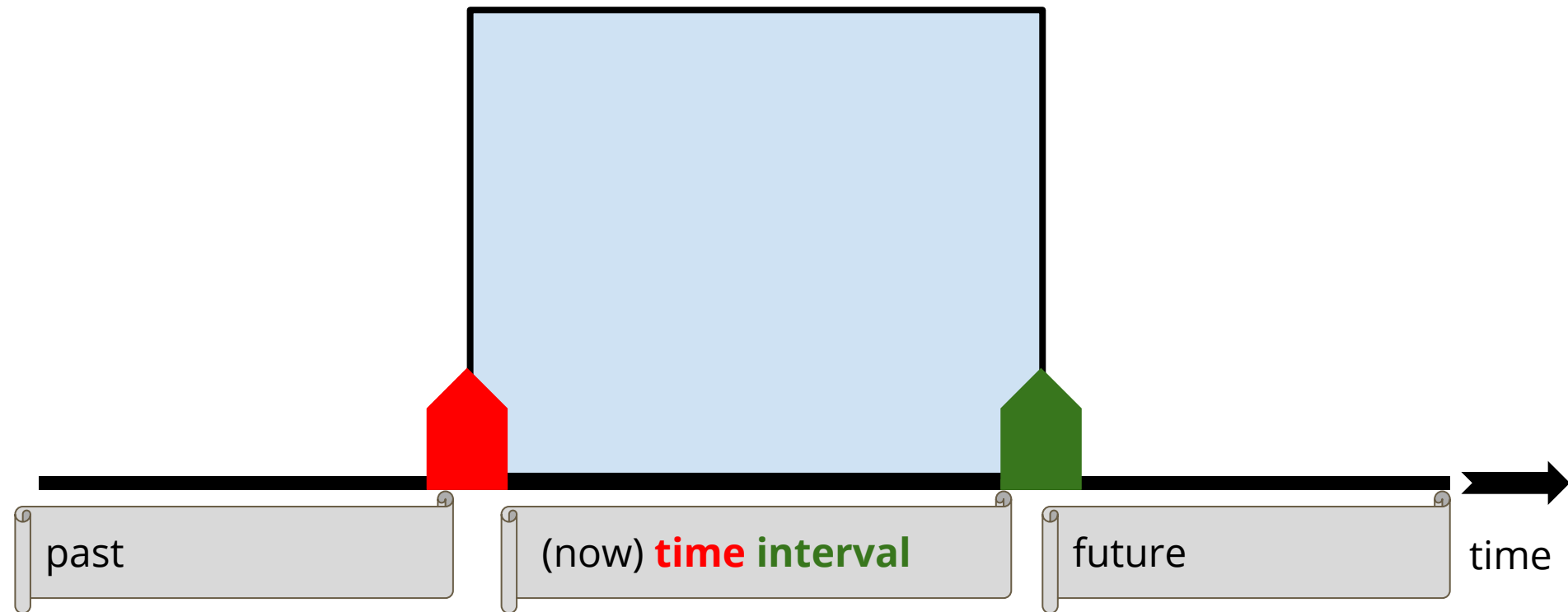
Idea 19: Extending Spark Core with Streaming Functionality

We have seen that Spark Core functionality can be extended to work in a Streaming fashion.



Idea 19: Extending Spark Core with Streaming Functionality

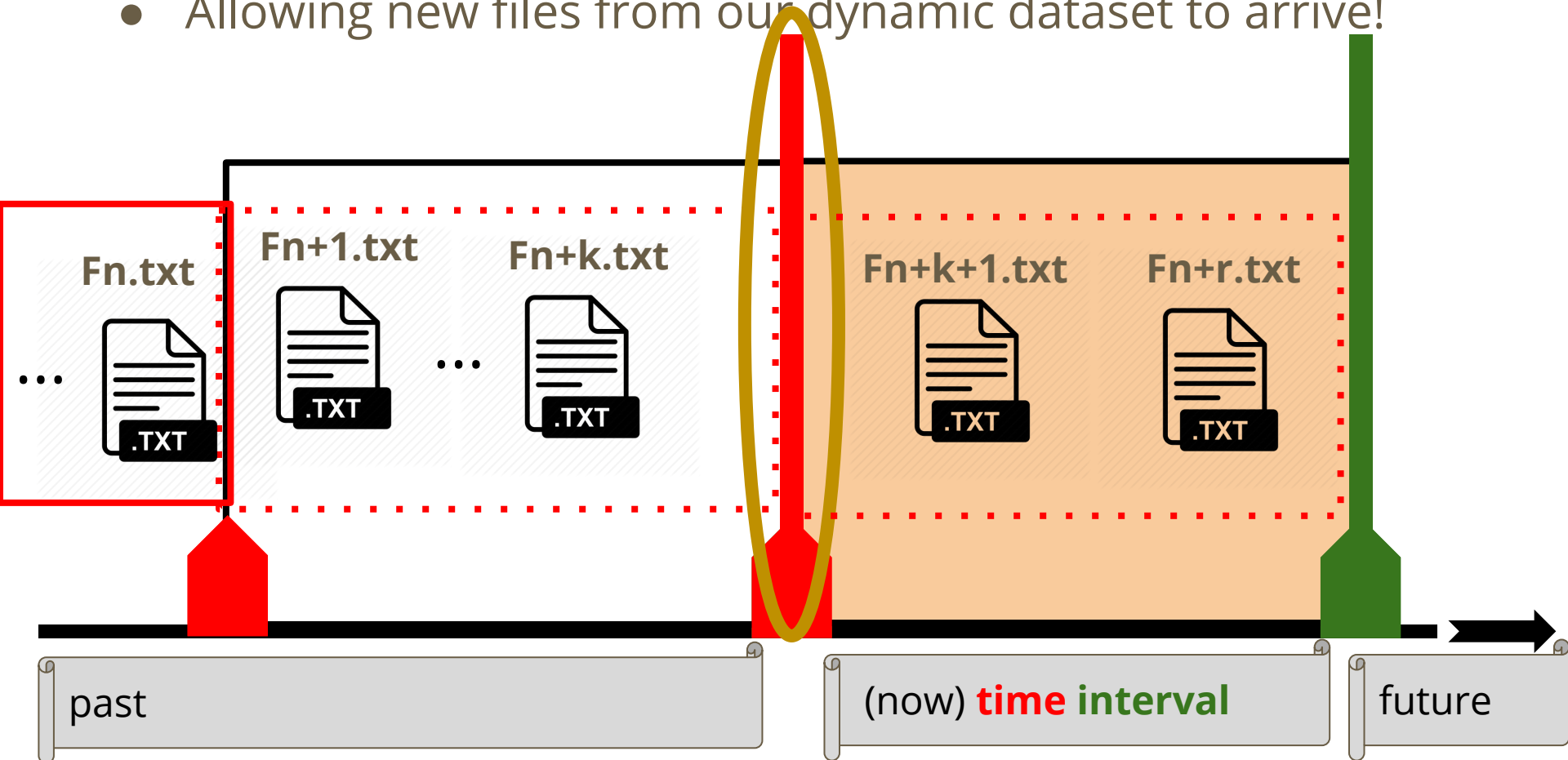
For it we need the notion of **time interval**...



Setting Up the Context

For it we need the notion of **time interval**...

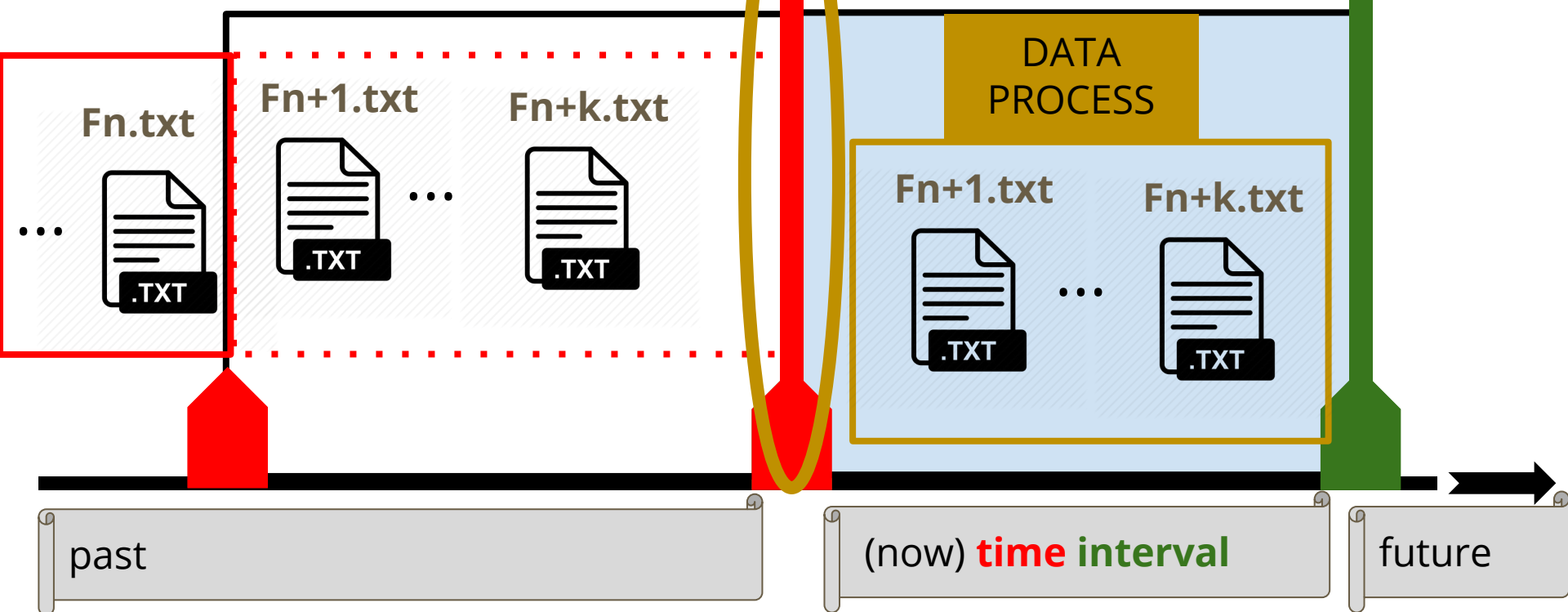
- Allowing new files from our dynamic dataset to arrive!



Setting Up the Context

For it we need the notion of **time interval**...

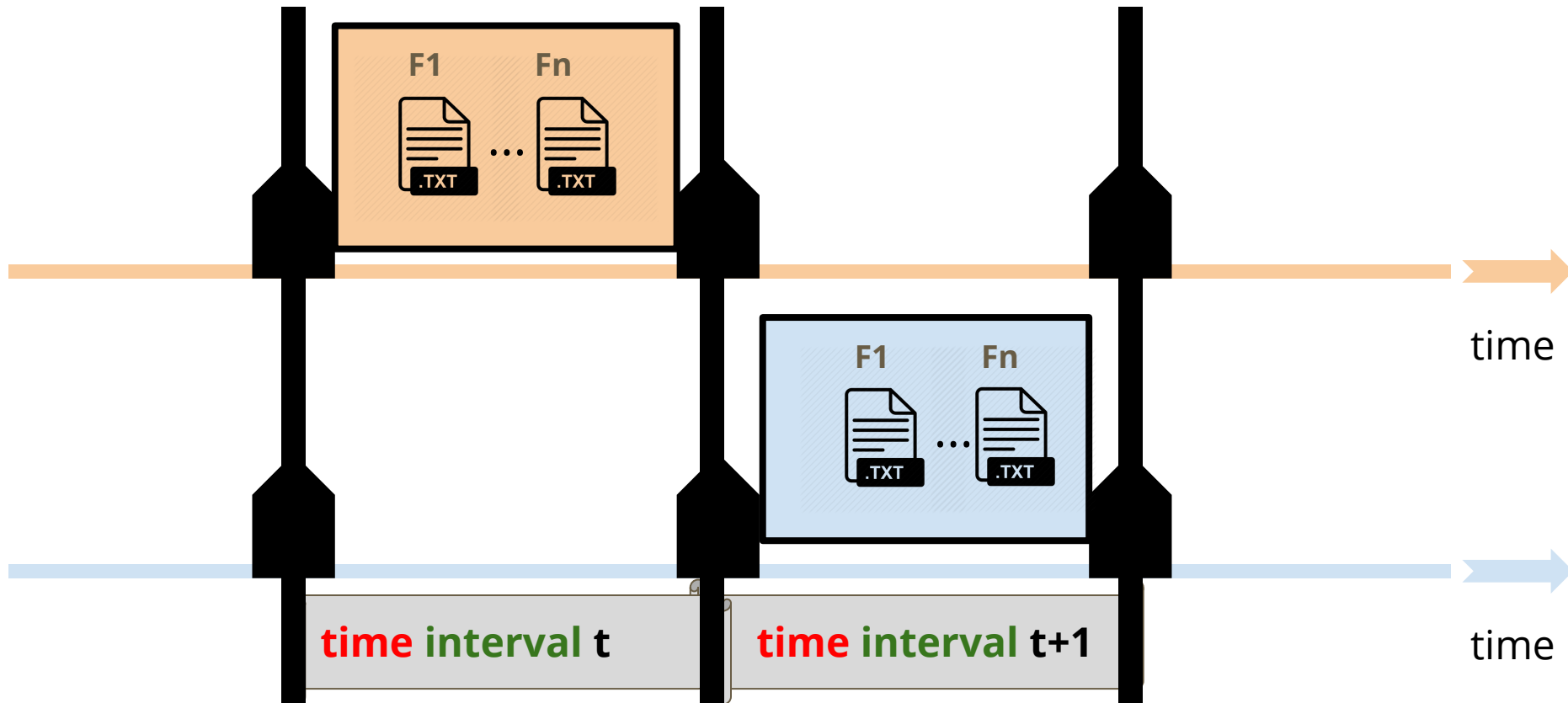
- Allowing new files from our dynamic dataset to arrive!
- Processing previously arrived files



Idea 19: Extending Spark Core with Streaming Functionality

For it we need the notion of **time interval**...

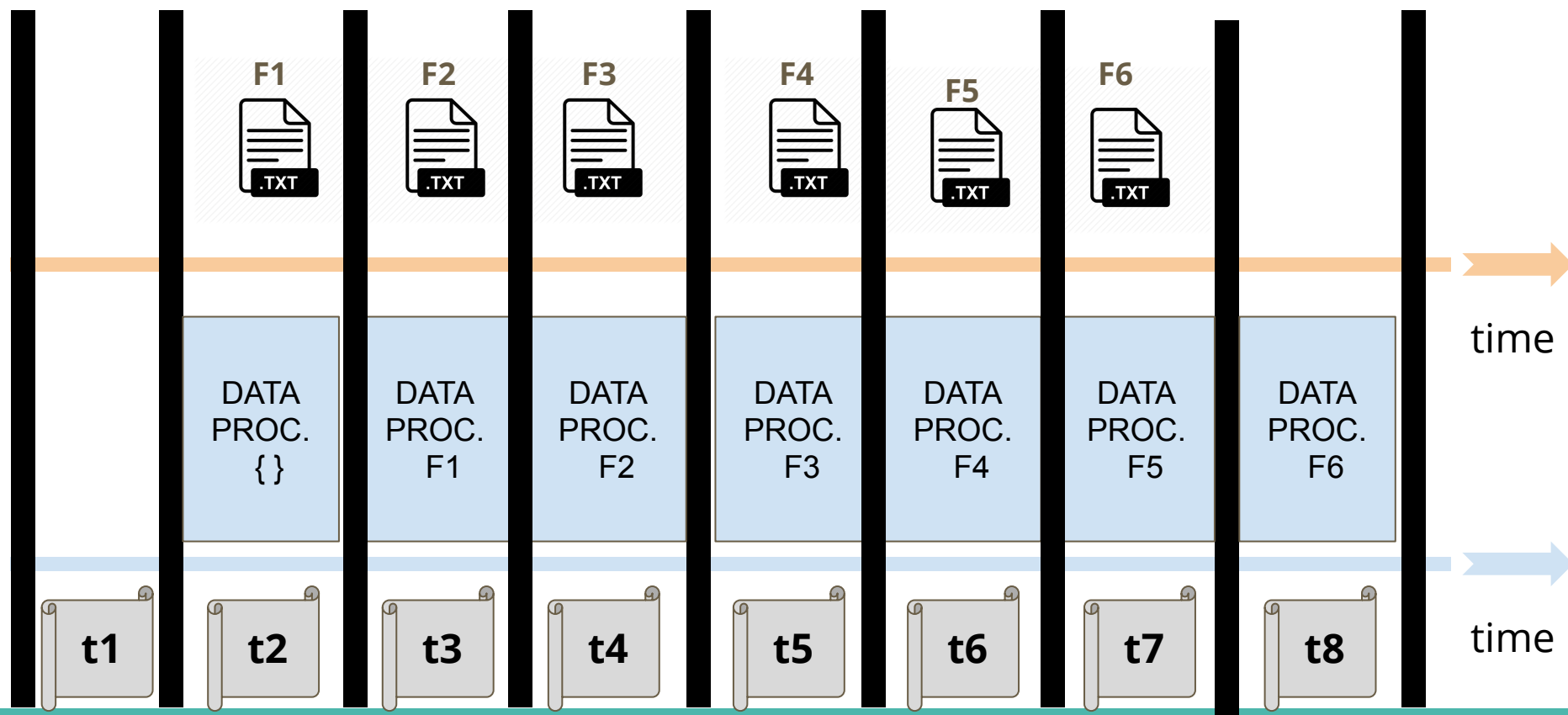
- In a process that is repeated over and over.



Idea 19: Extending Spark Core with Streaming Functionality

For it we need the notion of **time interval**...

- In a process that is repeated over and over.

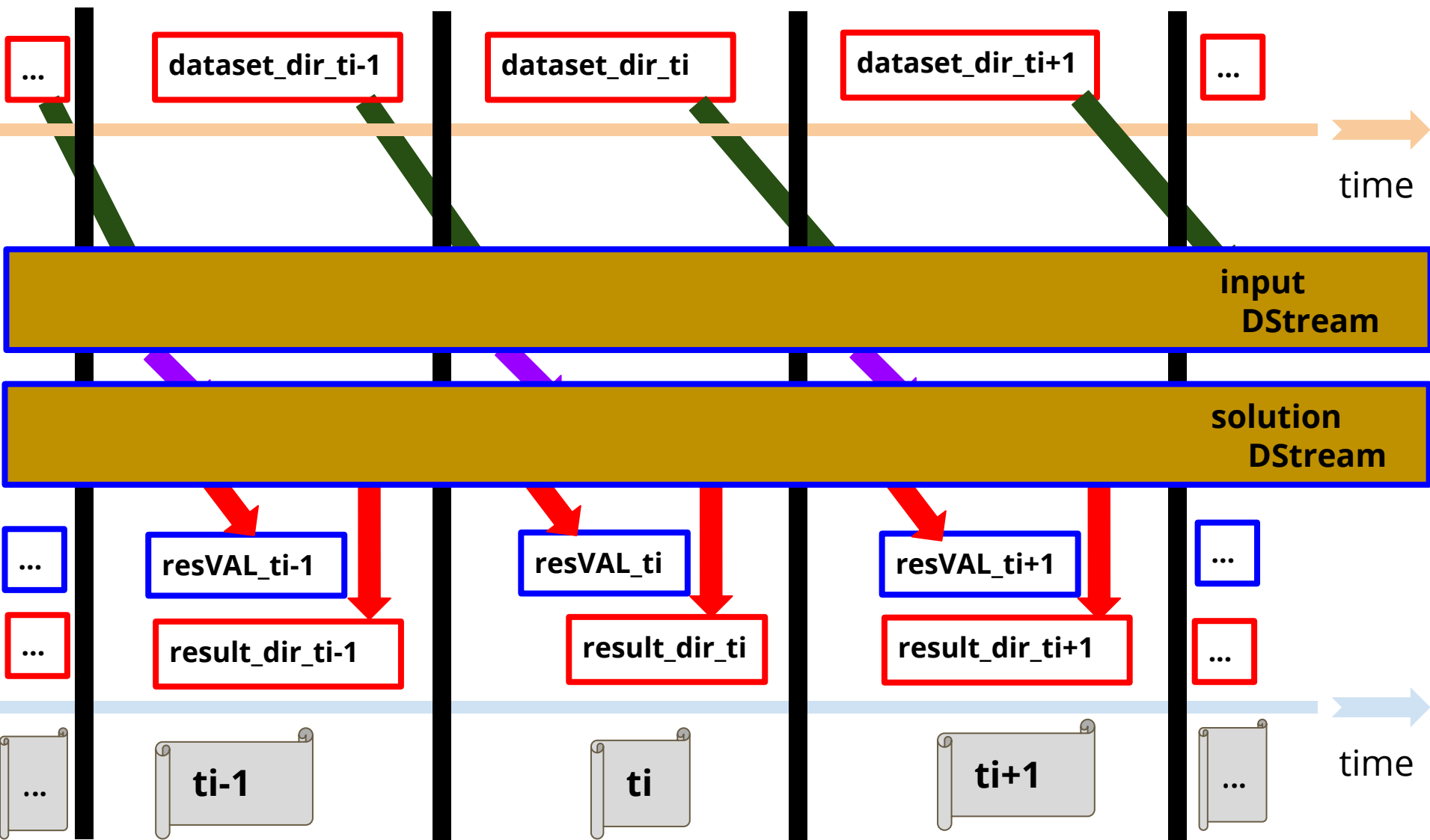


Idea 19: Extending Spark Core with Streaming Functionality

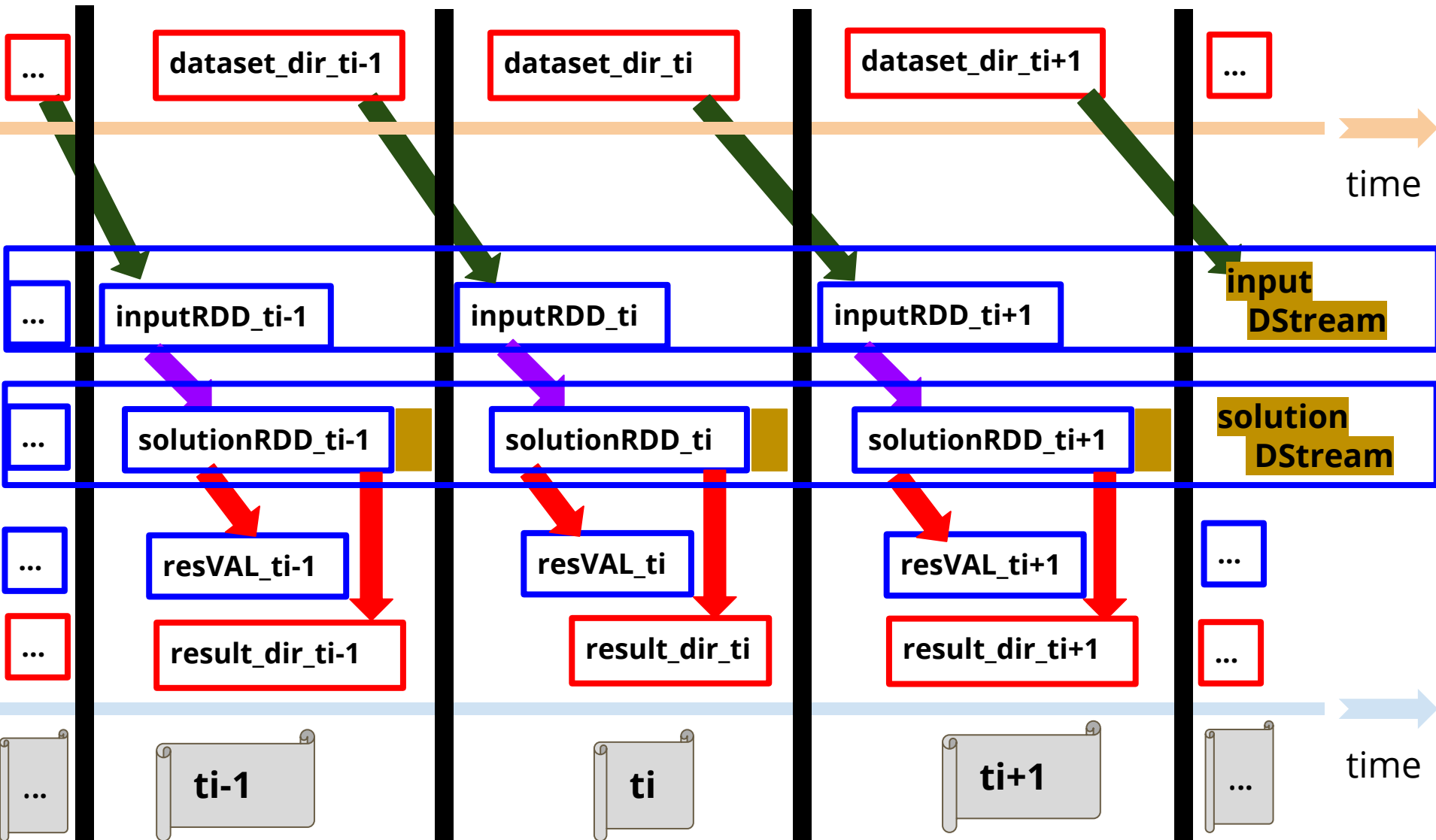
The data abstraction of Spark Streaming is a **DStream**, which can be seen as a **train** where each **wagon** represents the underlying **RDD** being processed during a concrete **time interval**.



Idea 19: Extending Spark Core with Streaming Functionality

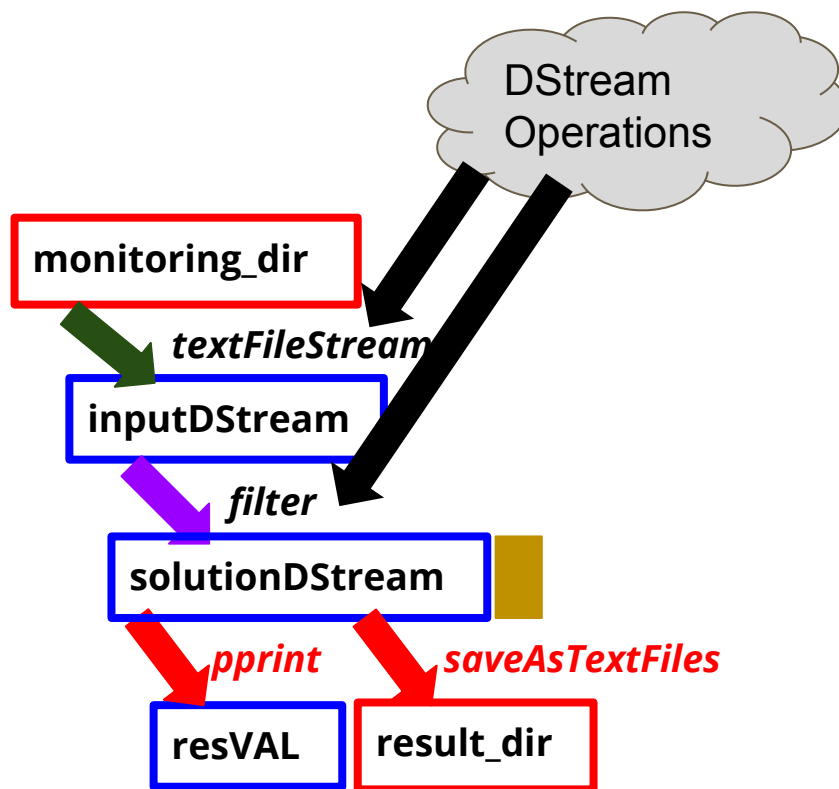


Idea 19: Extending Spark Core with Streaming Functionality

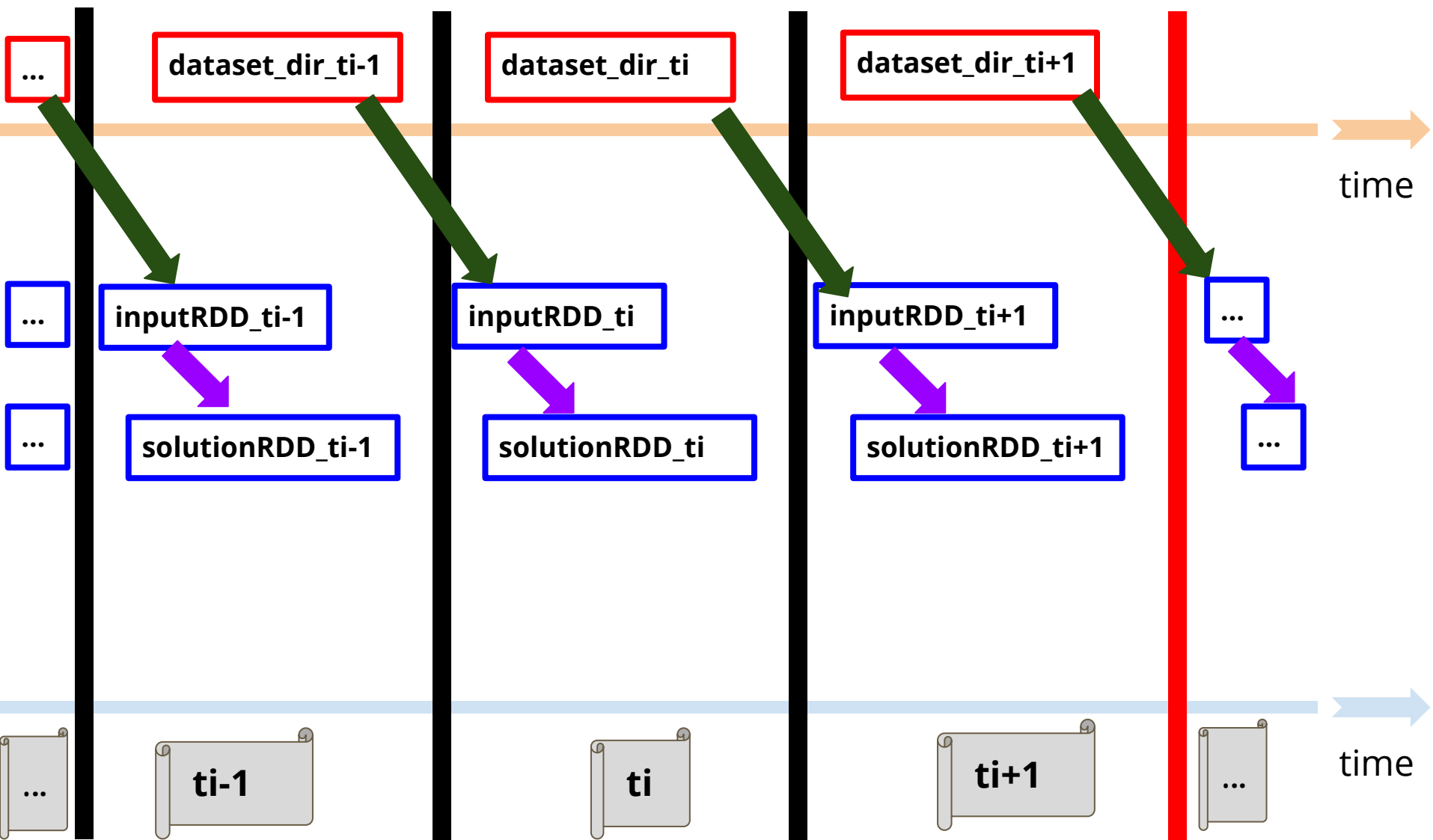


Idea 19: Extending Spark Core with Streaming Functionality

Regular **DStream** operations are applied to the underlying **RDDs** over time.



Idea 19: Extending Spark Core with Streaming Functionality



Outline

Idea 20:

On Altering Classical Time Period
Reasoning Pattern

Idea 20: On Altering Classical Time Period Reasoning Pattern

Window-based Operations can be used to group **wagons** together.



Idea 20: On Altering Classical Time Period Reasoning Pattern

Window-based Operations can be used to group **wagons** together.

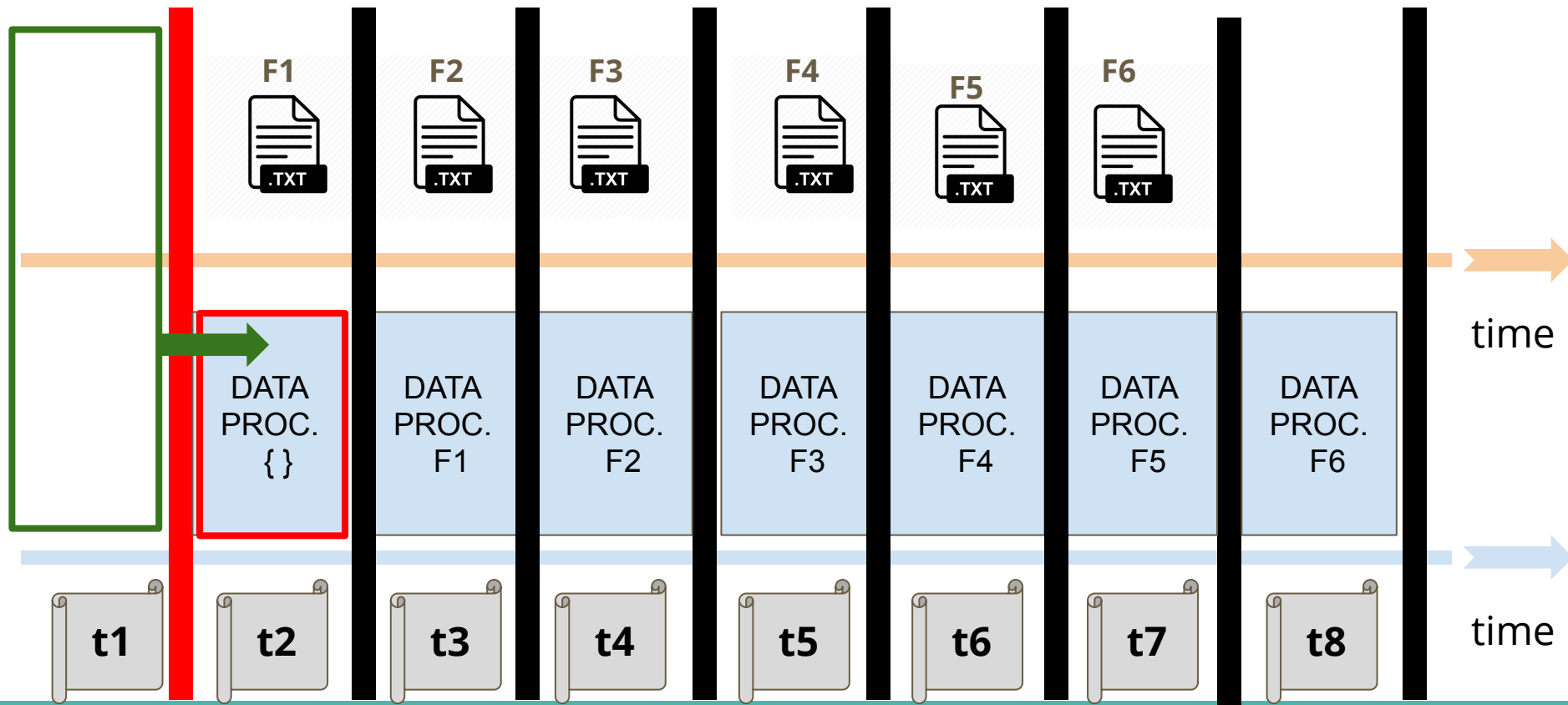


Idea 20: On Altering Classical Time Period Reasoning Pattern

Window-based Operations can be used to group **wagons** together.

Sliding Duration = 2 and Window Duration = 3

we create the following windows at the following times

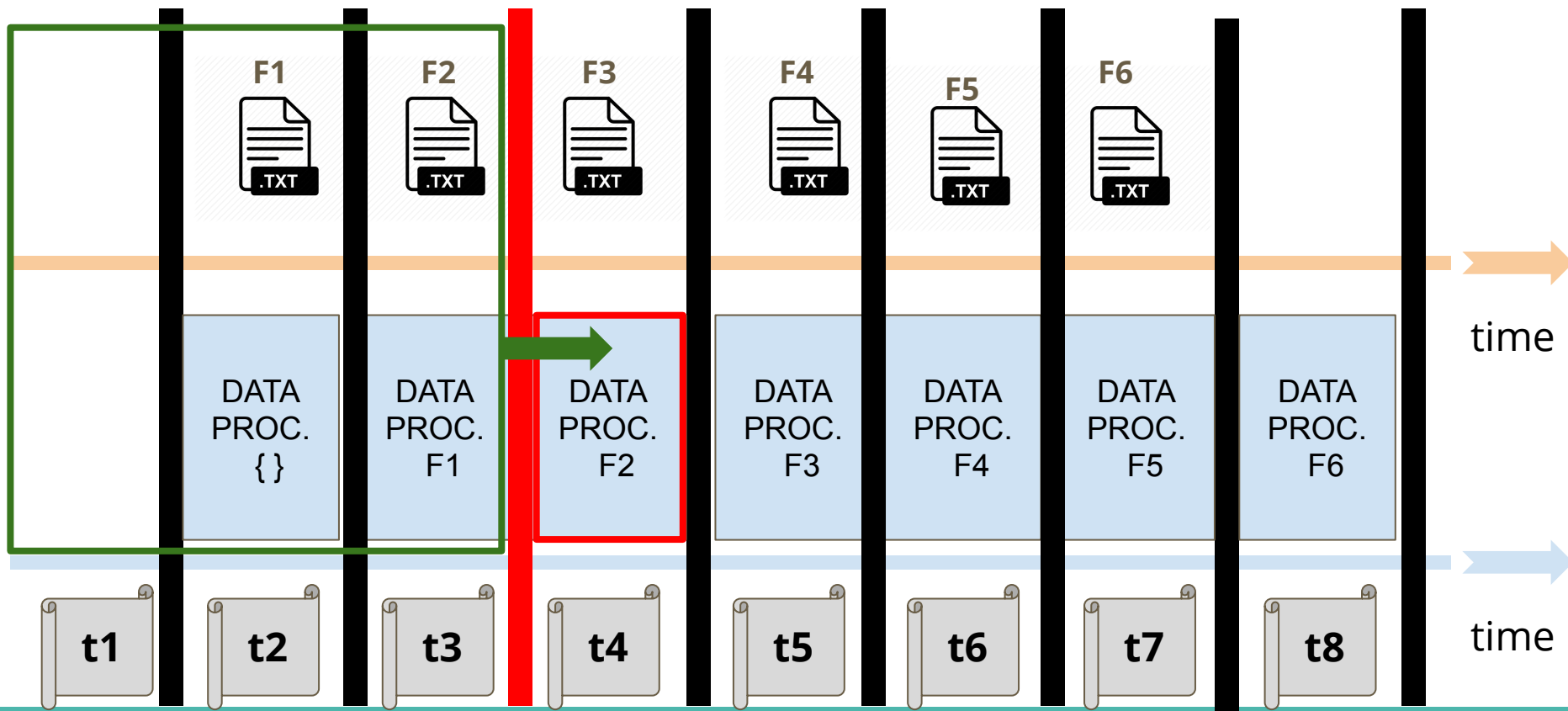


Idea 20: On Altering Classical Time Period Reasoning Pattern

Window-based Operations can be used to group **wagons** together.

Sliding Duration = 2 and **Window Duration = 3**

we create the following windows at the following times

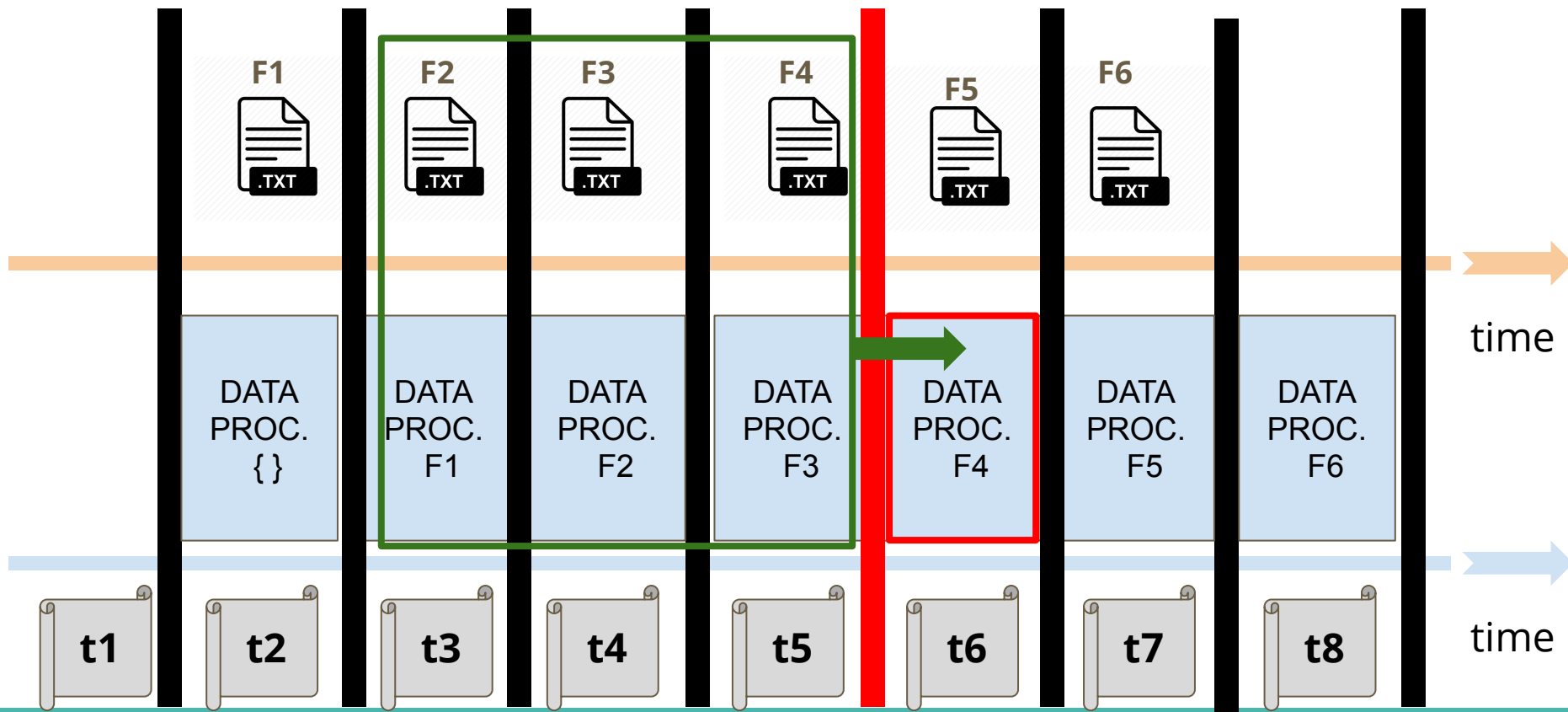


Idea 20: On Altering Classical Time Period Reasoning Pattern

Window-based Operations can be used to group **wagons** together.

Sliding Duration = 2 and **Window Duration = 3**

we create the following windows at the following times

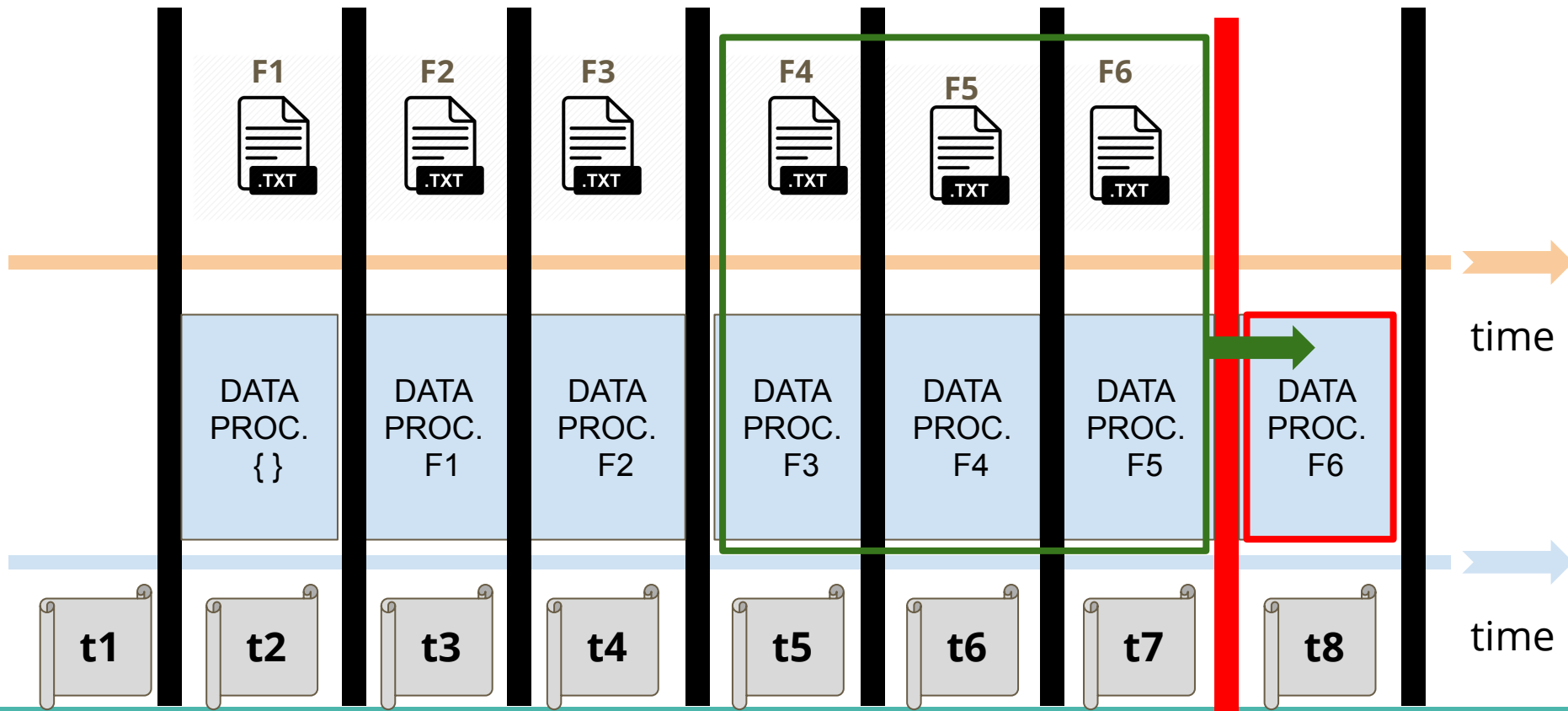


Idea 20: On Altering Classical Time Period Reasoning Pattern

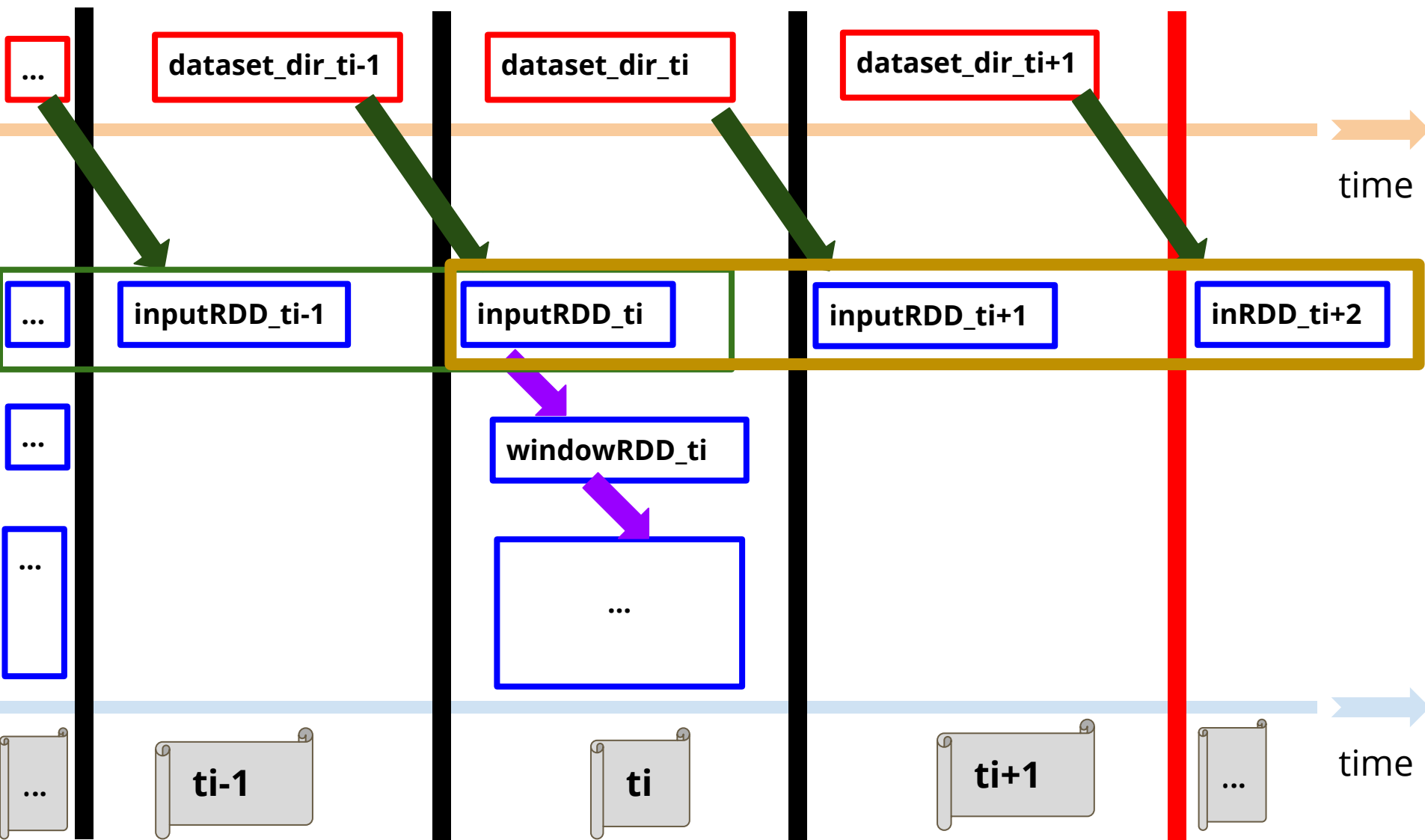
Window-based Operations can be used to group **wagons** together.

Sliding Duration = 2 and **Window Duration = 3**

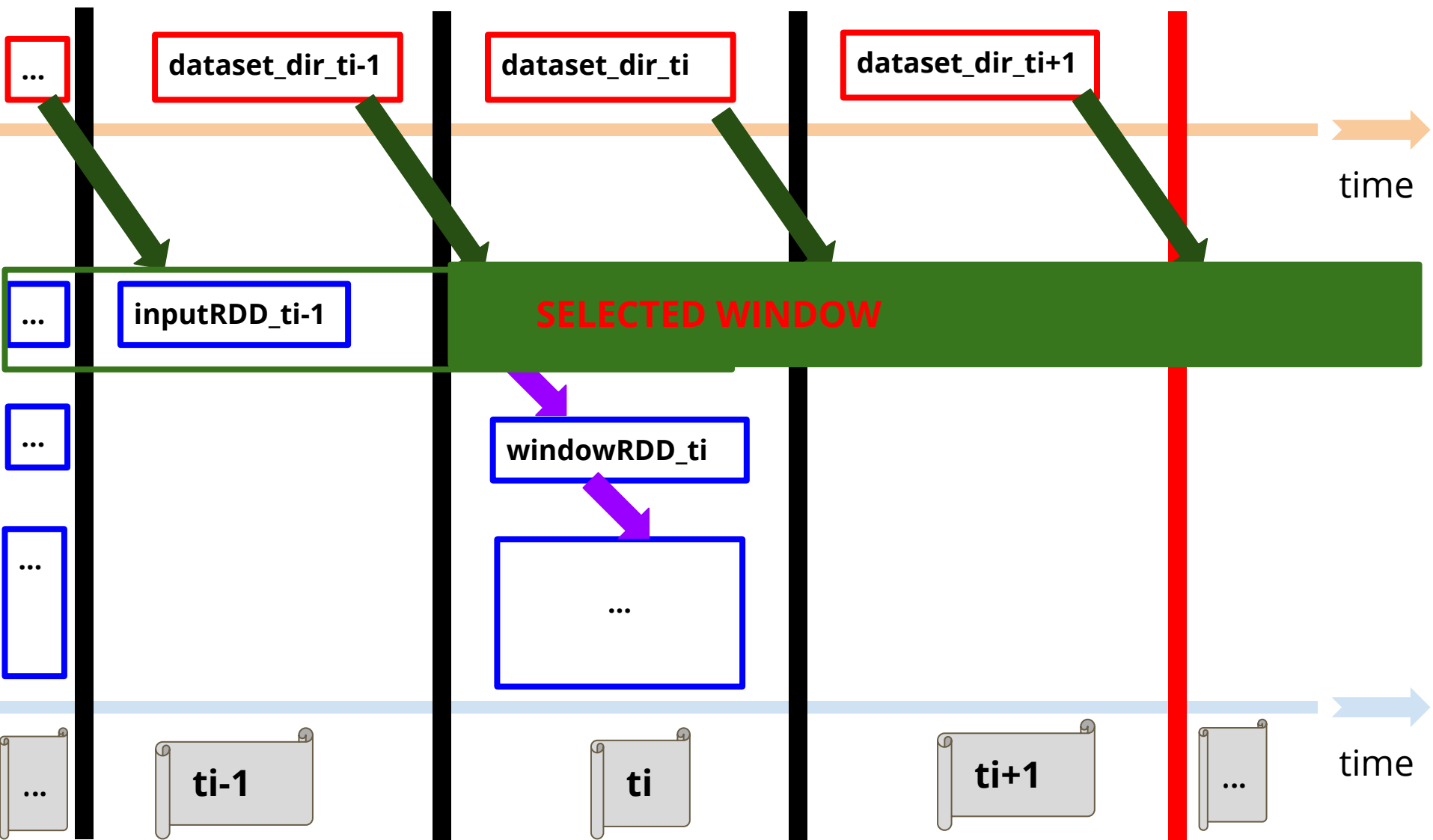
we create the following windows at the following times



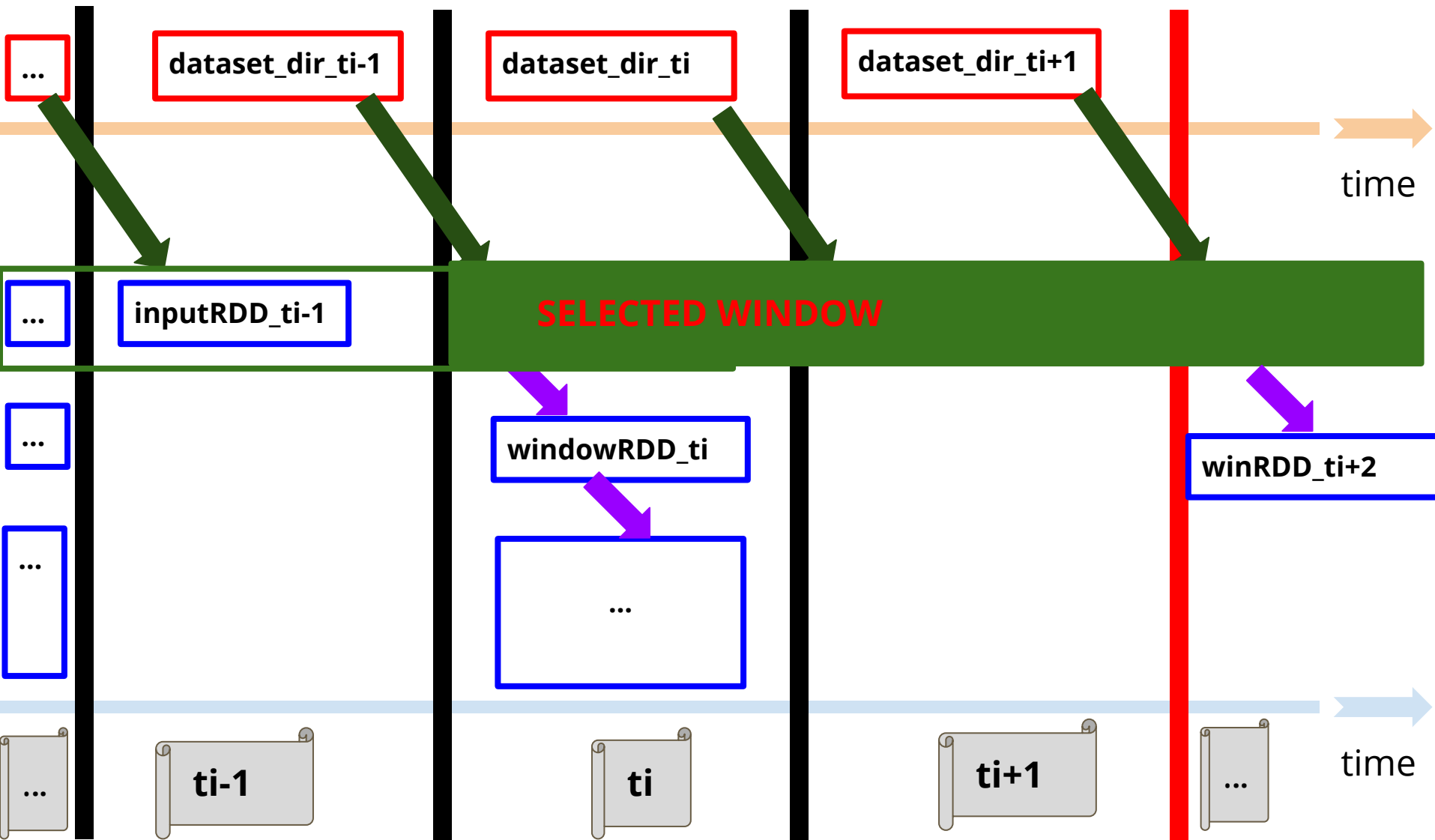
Idea 20: On Altering Classical Time Period Reasoning Pattern



Idea 20: On Altering Classical Time Period Reasoning Pattern



Idea 20: On Altering Classical Time Period Reasoning Pattern



Idea 20: On Altering Classical Time Period Reasoning Pattern

Update-based operations aggregate results for all the **wagons** processed so far.

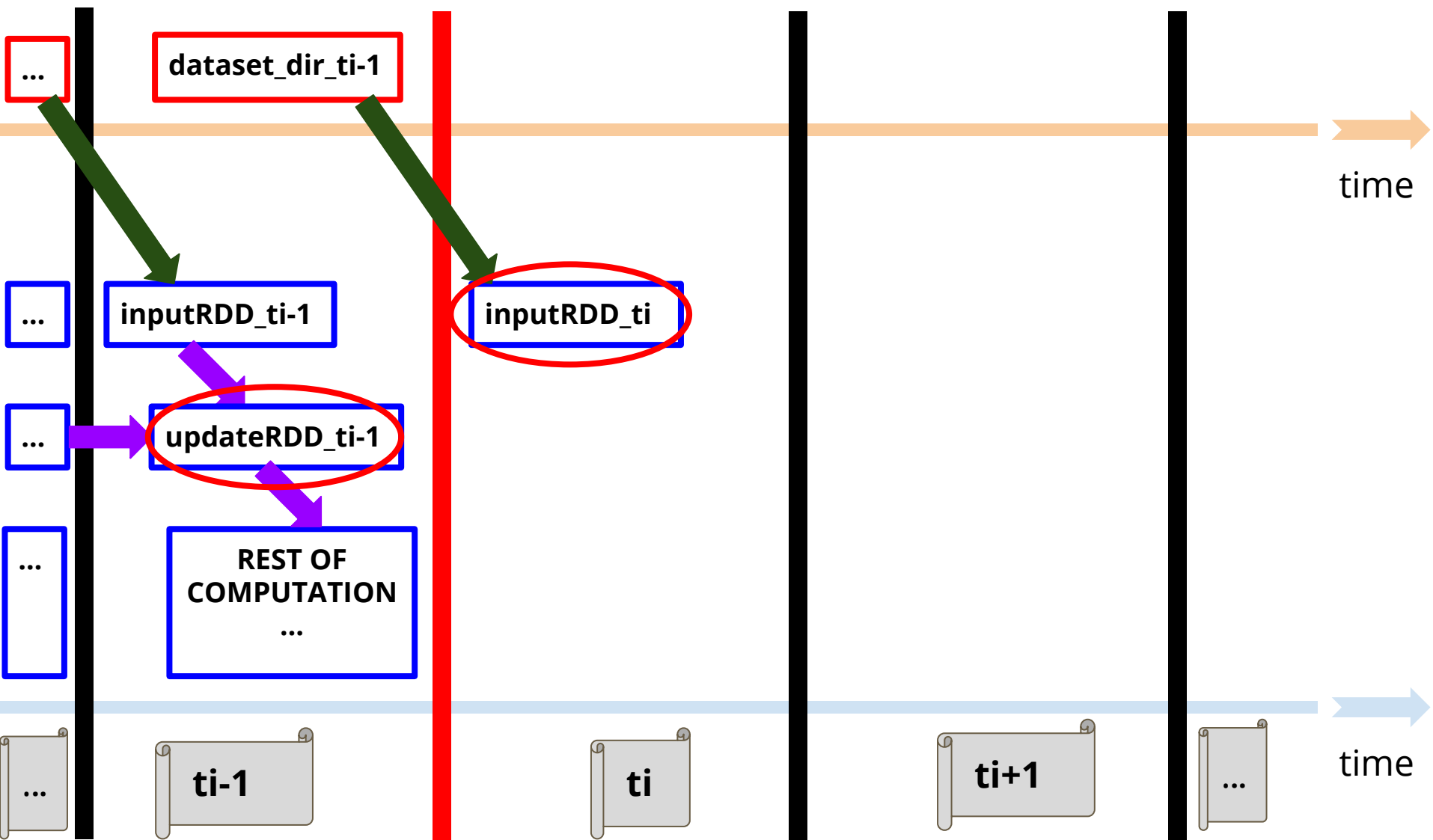


Idea 20: On Altering Classical Time Period Reasoning Pattern

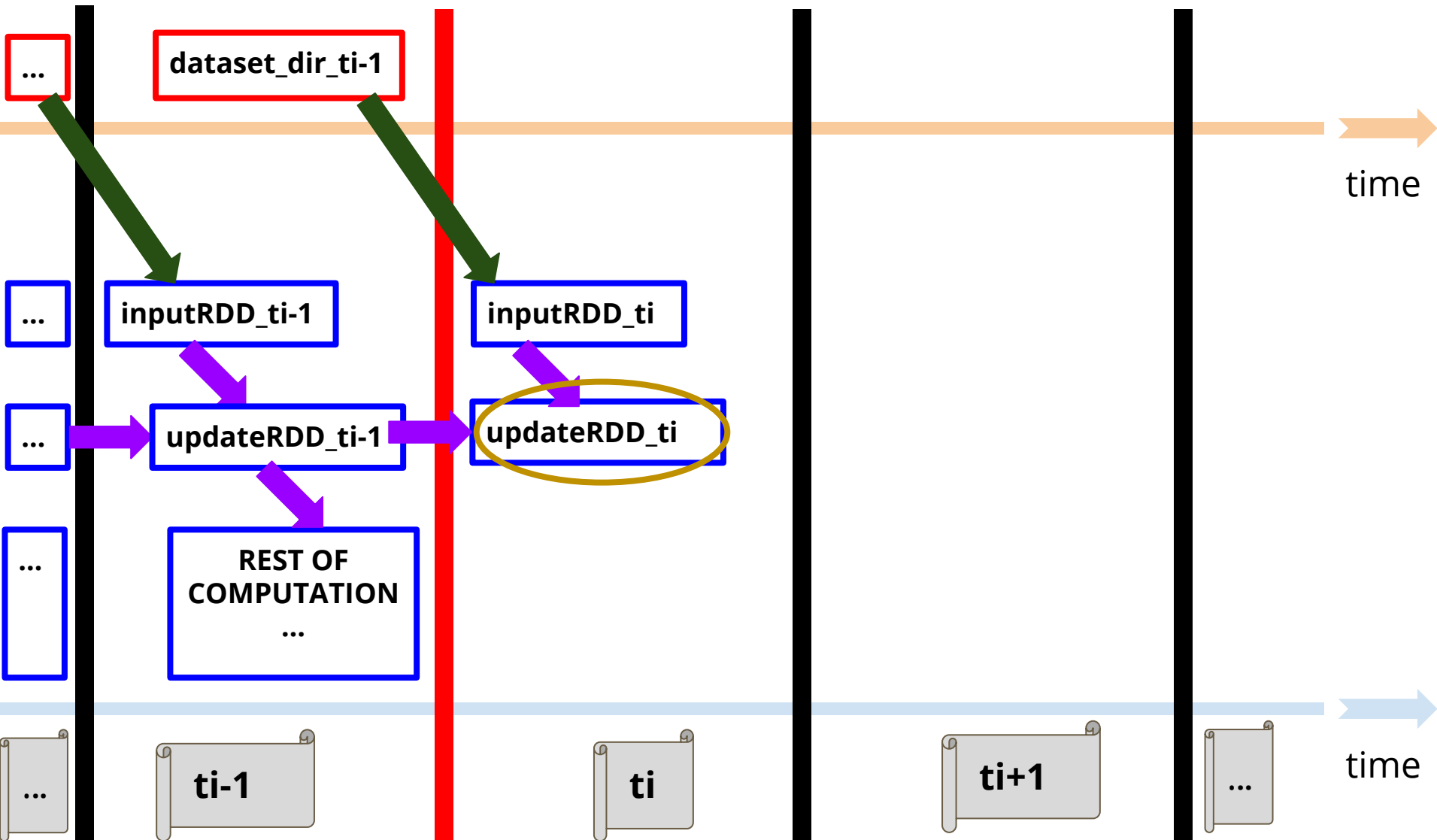
Update-based operations aggregate results for all the **wagons** processed so far.



Idea 20: On Altering Classical Time Period Reasoning Pattern



Idea 20: On Altering Classical Time Period Reasoning Pattern



Idea 20: On Altering Classical Time Period Reasoning Pattern

Update-based operations aggregate results for all the **wagons** processed so far.

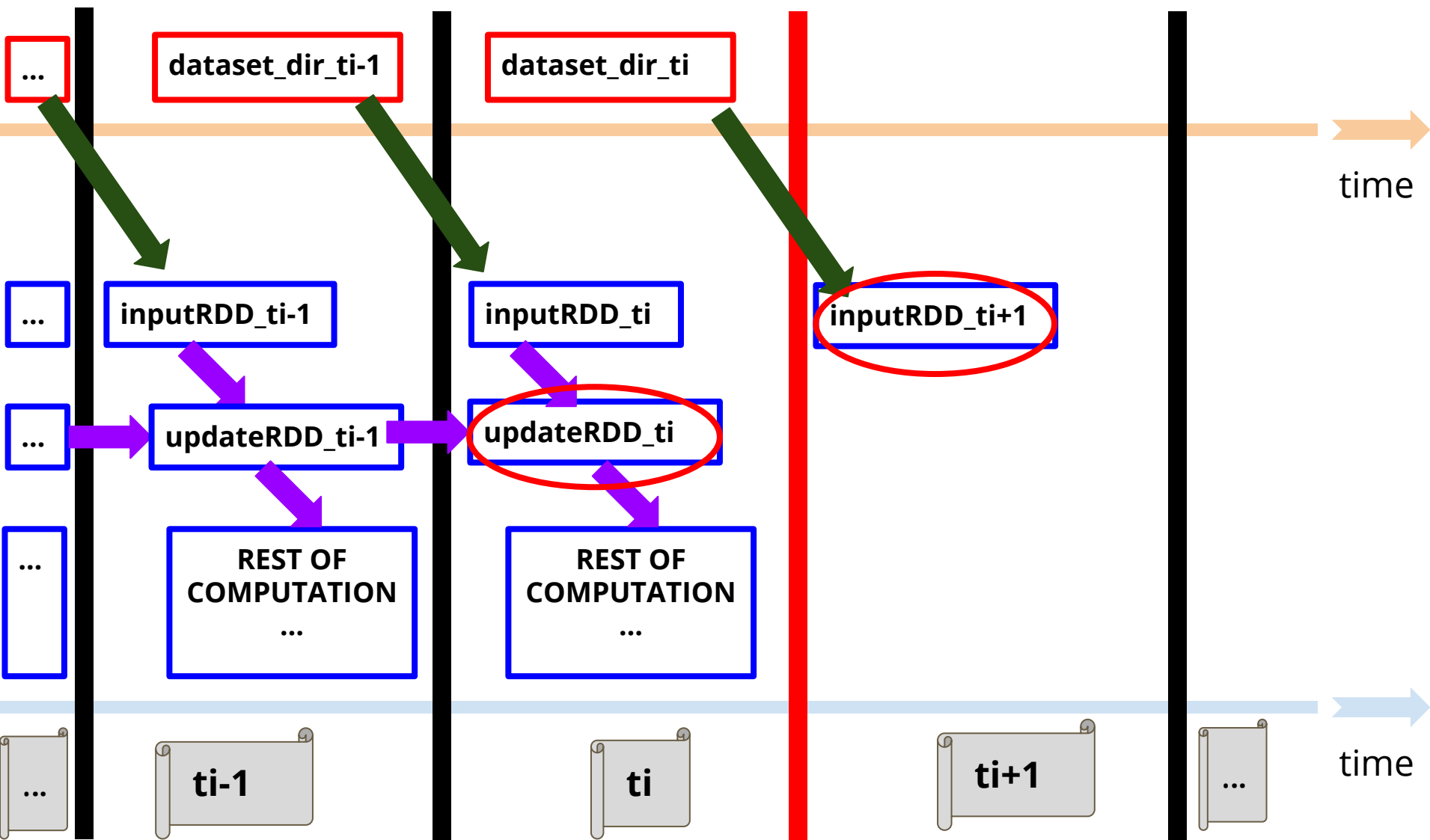


Idea 20: On Altering Classical Time Period Reasoning Pattern

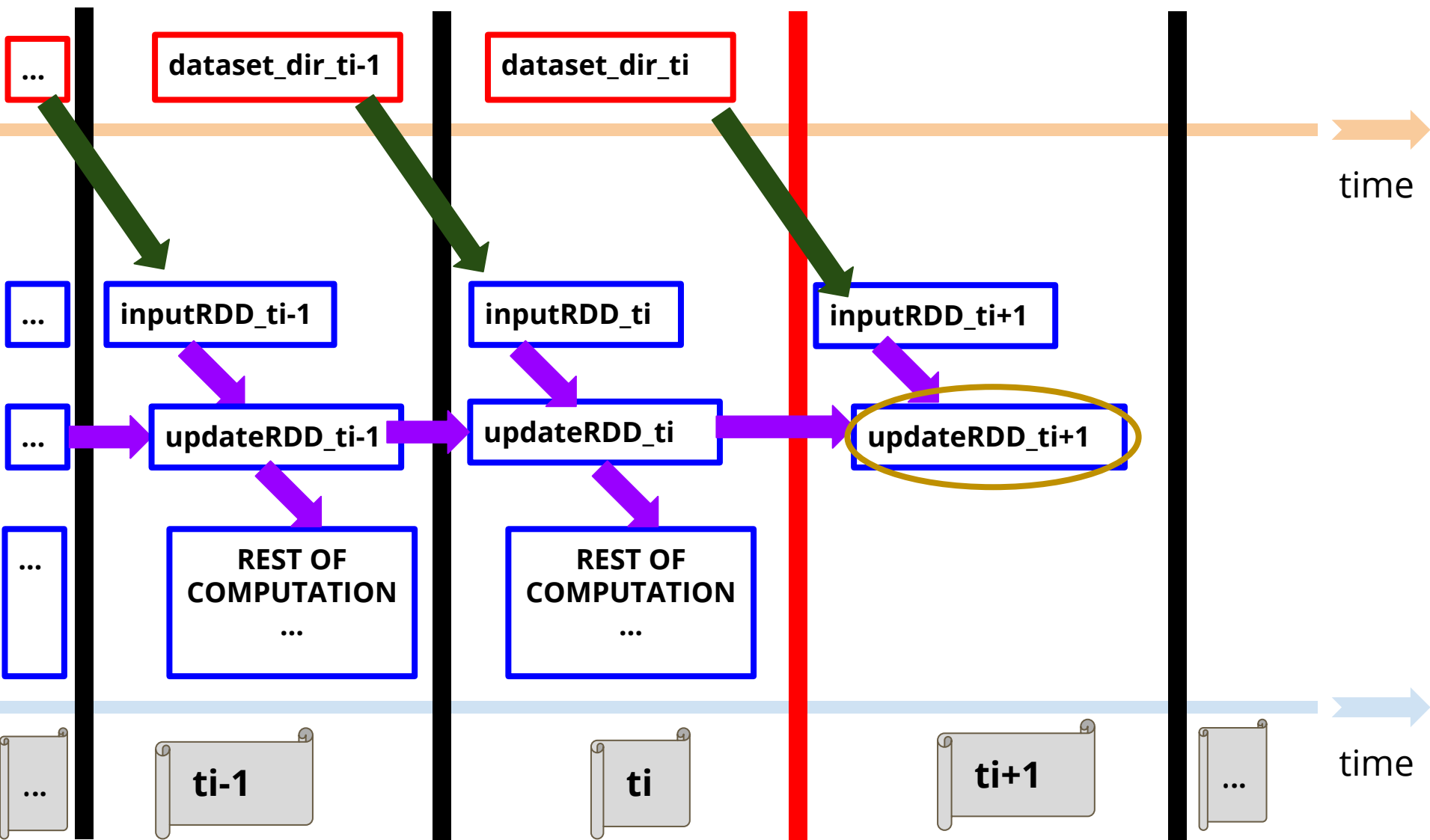
Update-based operations aggregate results for all the **wagons** processed so far.



Idea 20: On Altering Classical Time Period Reasoning Pattern



Idea 20: On Altering Classical Time Period Reasoning Pattern

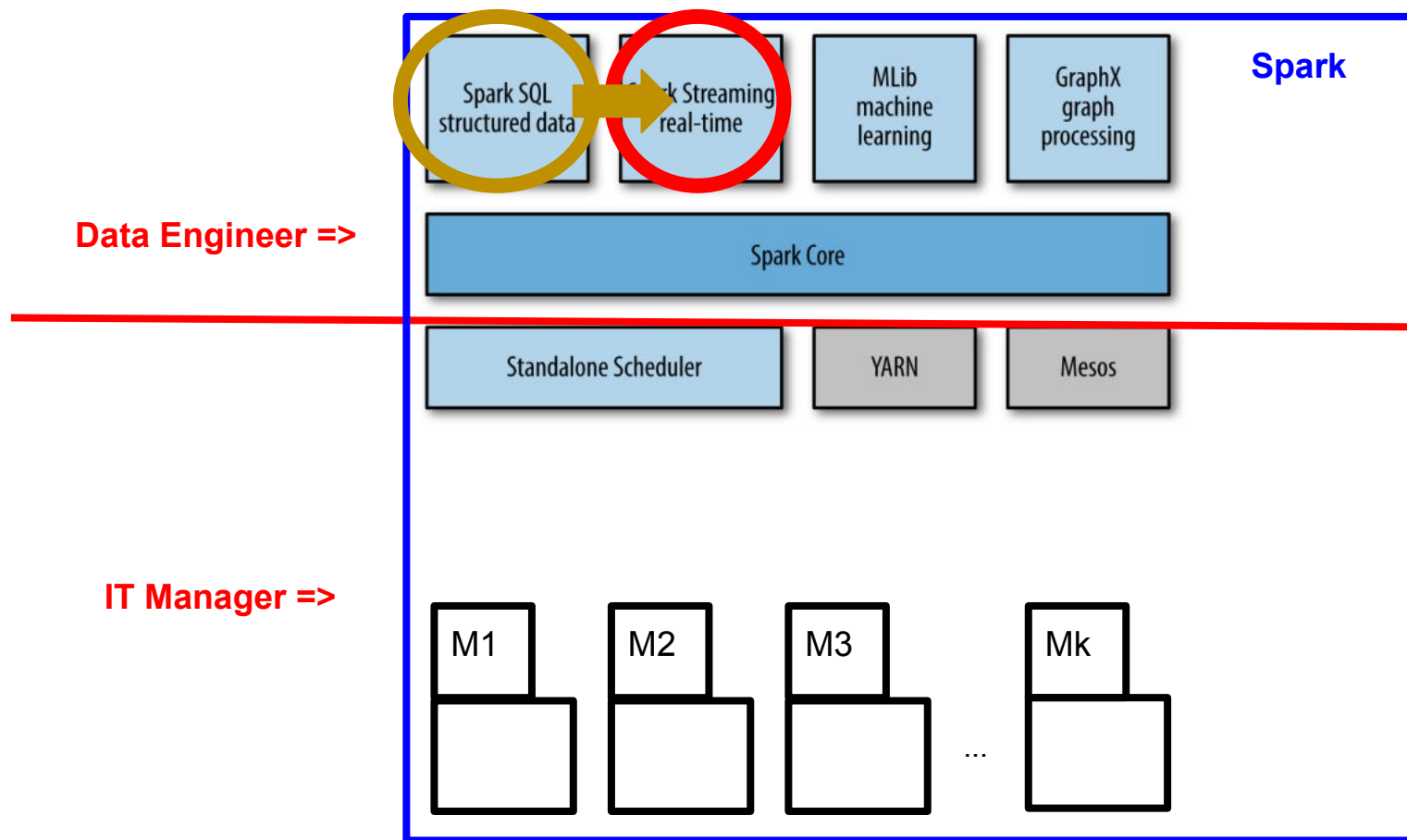


Outline

Idea 21:

Extending Spark SQL
with Streaming Functionality

Idea 21: Extending Spark SQL with Streaming Functionality



Idea 21: Extending Spark SQL with Streaming Functionality

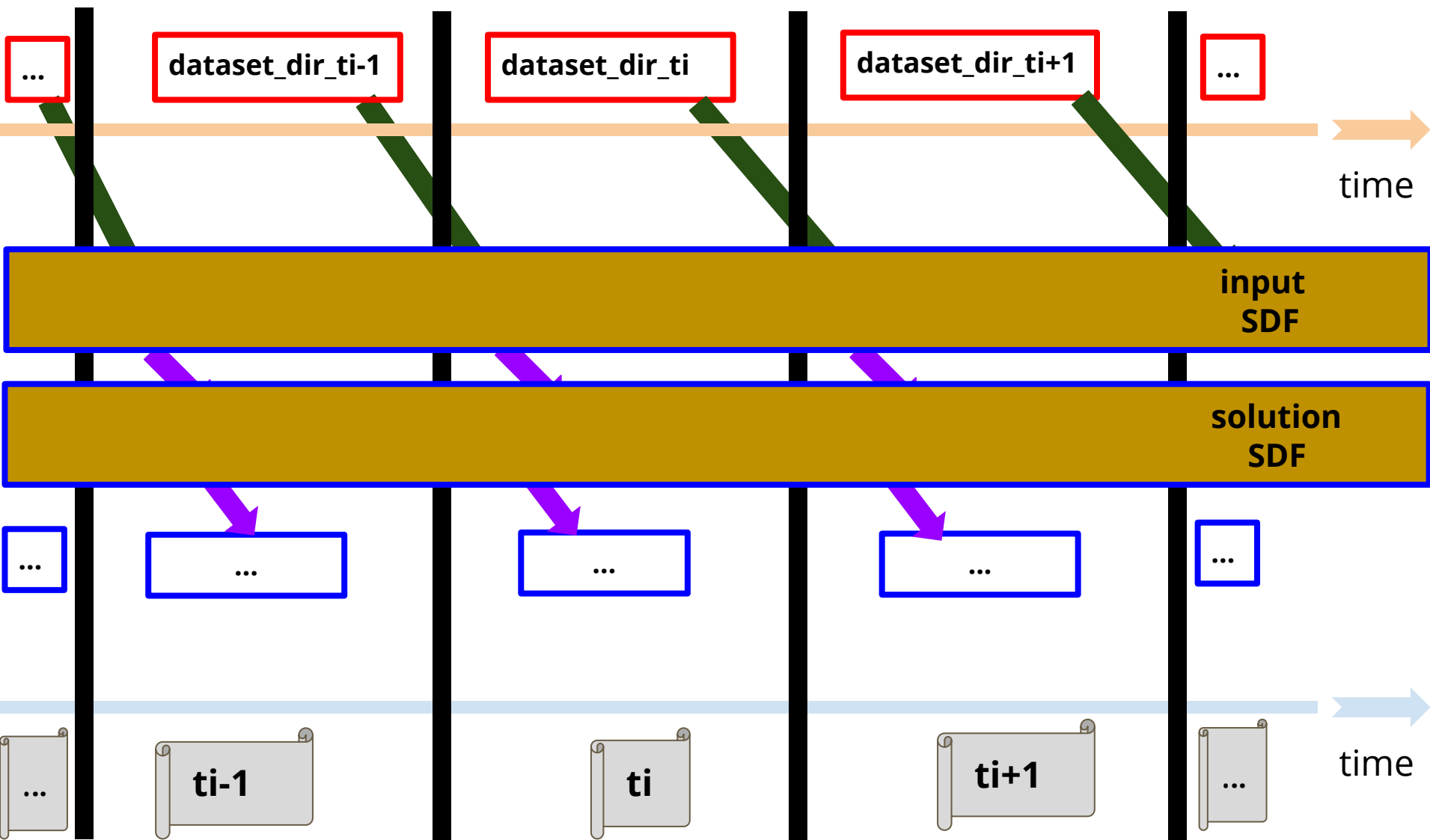
The same behaviour of **DStream** applies to Streaming DataFrames (**SDF**)!

Idea 21: Extending Spark SQL with Streaming Functionality

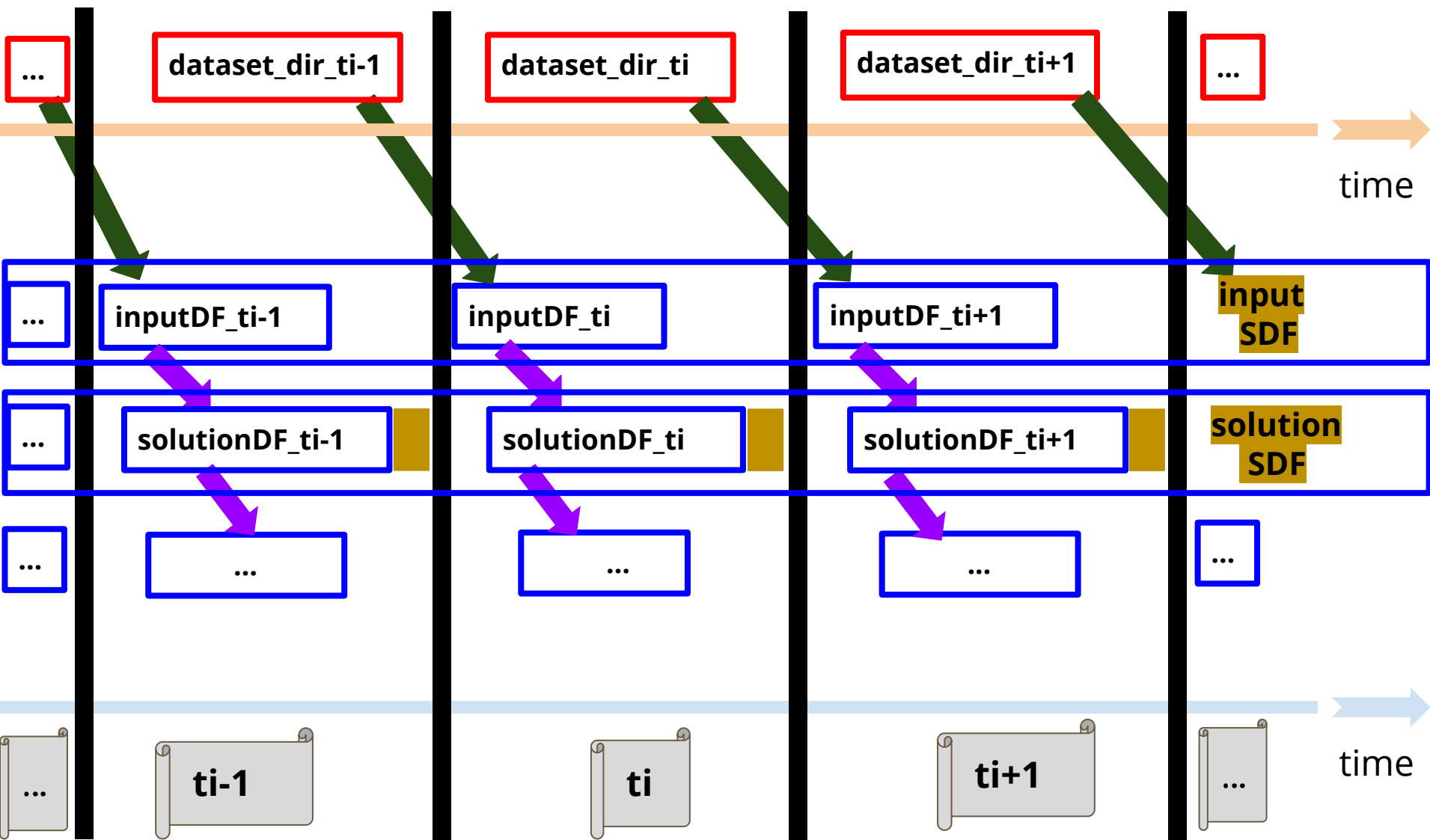
A **Streaming DataFrame (SDF)** also represents an amalgamation of **DataFrames (DF)** over time.



Idea 21: Extending Spark SQL with Streaming Functionality

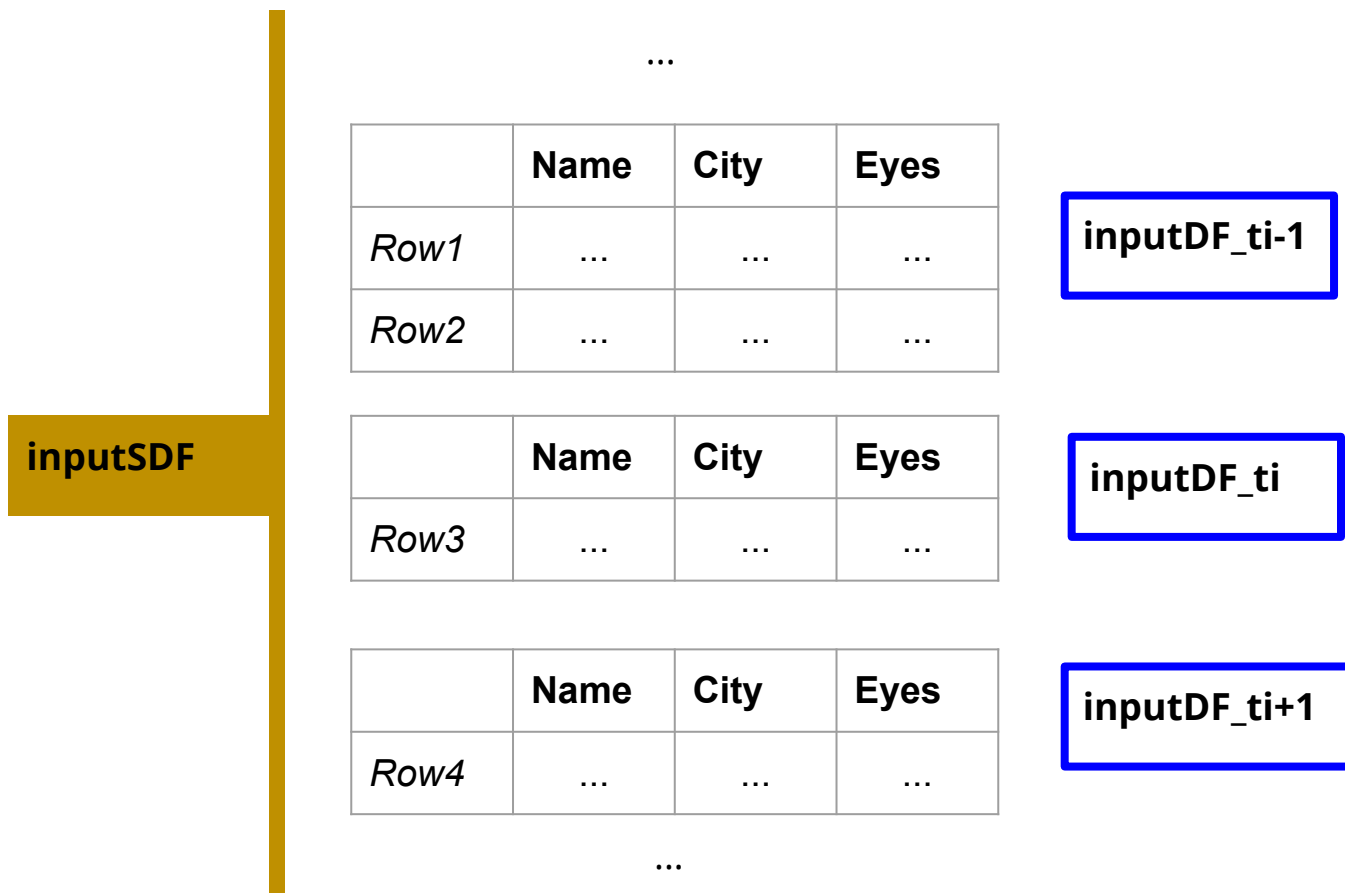


Idea 21: Extending Spark SQL with Streaming Functionality



Idea 21: Extending Spark SQL with Streaming Functionality

The SDF can also be visualised as an Unbound table, which keeps receiving Rows over time.



Idea 21: Extending Spark SQL with Streaming Functionality

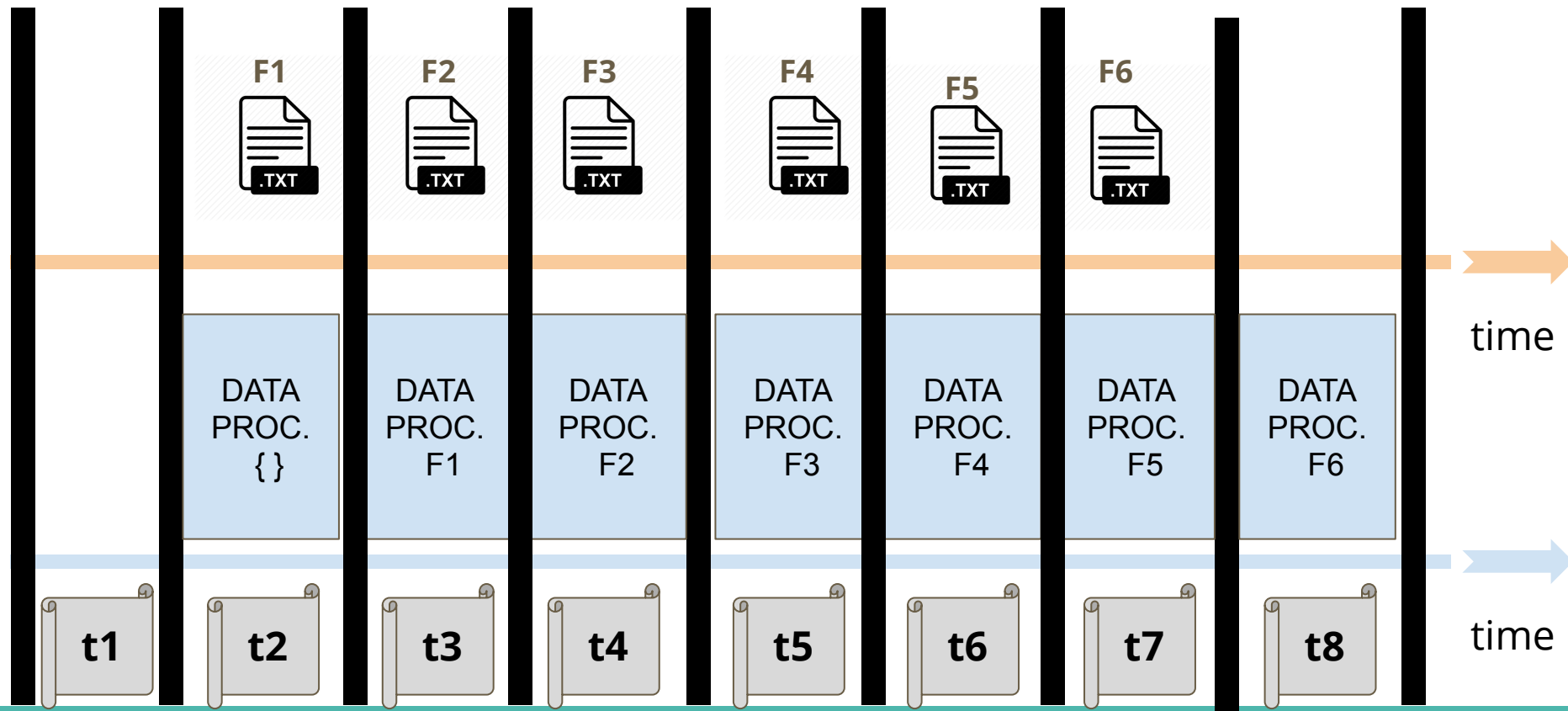
In any case, whether you use:

1. An horizontal viewing based on a **train** of **wagons** (DFs)
2. A vertical viewing based on an **unbound data structure** of **Rows**

the behaviour of **DStream** and **SDF** is equivalent!

Idea 21: Extending Spark SQL with Streaming Functionality

Our regular working mode is called Append Mode.



Idea 20: On Altering Classical Time Period Reasoning Pattern

Our **update**-based operations lead to Complete Mode.



Idea 20: On Altering Classical Time Period Reasoning Pattern

Our **update**-based operations lead to Complete Mode.



Idea 20: On Altering Classical Time Period Reasoning Pattern

Our **update**-based operations lead to Complete Mode.



Idea 21: Extending Spark SQL with Streaming Functionality

- Both Append and Complete Mode support reasoning over **windows**.

Idea 21: Extending Spark SQL with Streaming Functionality

- Both Append and Complete Mode support reasoning over **windows**.
- As a difference, in Structured Spark Streaming, we have to choose on whether:
 - Print the results by the screen (Console Sink mode).
 - Save the results to our file system (File Sink Output).

Outline

Idea 22:

Spark Structured Streaming
Limited Functionality

Idea 22: Spark Structured Streaming Limited Functionality

We have seen that we can apply some of the functionality of **Spark Streaming** to **Spark Structured Streaming** as well.

However, there are quite some differences and limitations making Spark Structured Streaming not as appealing as it should be.

Outline

We have summarised the main ideas presented in our module Big Data Processing.

Outline

Let's finish now with a final synthesis.

Outline

Let's finish now with a final synthesis.

If we were define our module
Big Data Processing
in just one slide,
what would we say?

Outline

Big Data Processing (COMP9062)

Big Data processing is the first stage of each AI problem's journey.

In this module we have done
Big Data Processing
using the state-of-the-art technology
Apache Spark.

Thank you so much for your attention,
I hope you have enjoyed it :)