CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

# Big Data Processing

## L13-14: Spark SQL

**Dr. Ignacio Castineiras**
Department of Computer Science

## Outline

1. Setting Up the Context.
2. Functional Spark: Spark Core.
3. Structured Spark: Spark SQL.
4. DataFrames & Datasets: Public Side.
5. DataFrames & Datasets: Private Side.

## Outline

1. Setting Up the Context.
2. Functional Spark: Spark Core.
3. Structured Spark: Spark SQL.
4. DataFrames & Datasets: Public Side.
5. DataFrames & Datasets: Private Side.

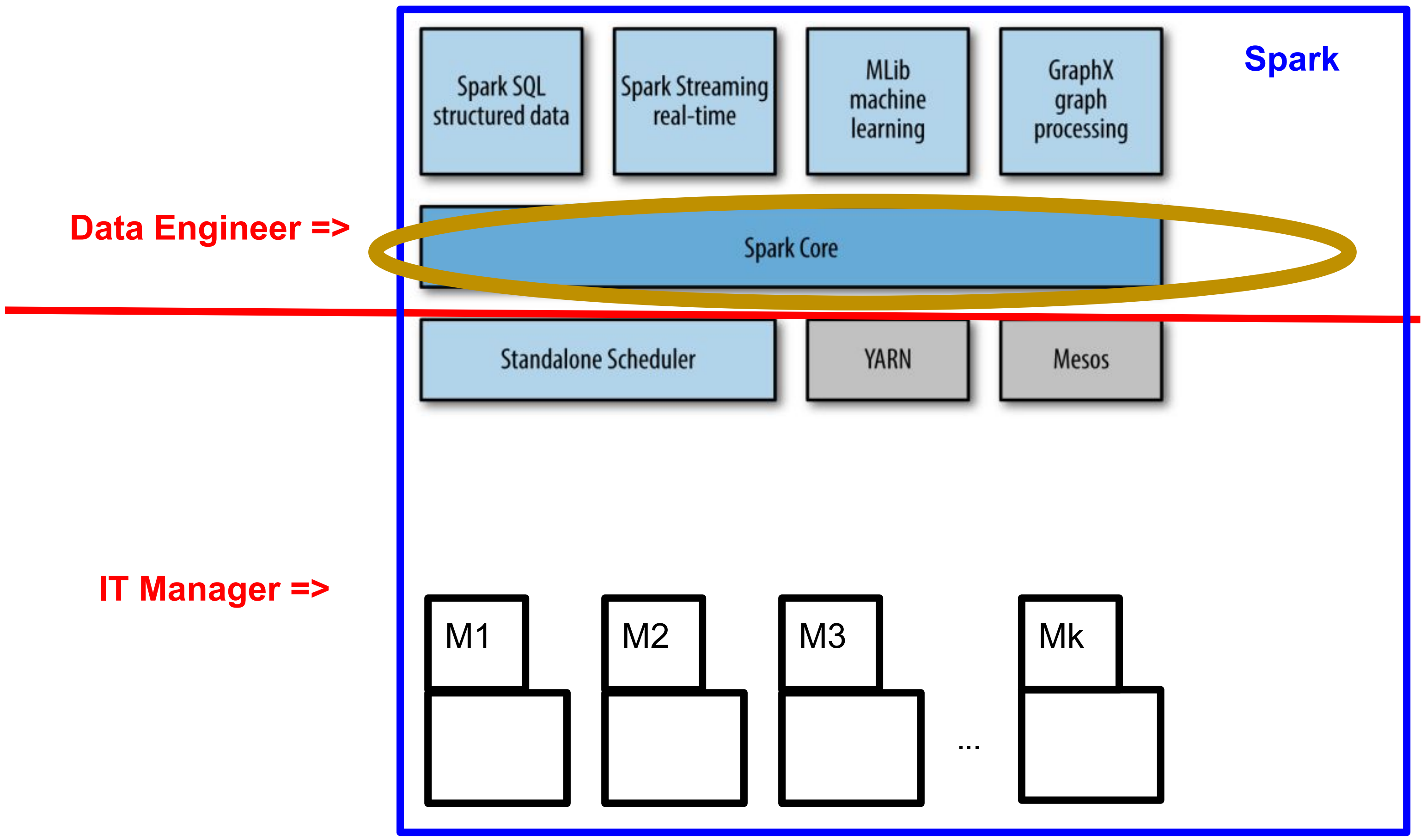
## Setting the Context

1. At this stage we are fully familiar with Spark, our:
   - open-source
   - distributed
   - general-purpose
   - cluster-computing

   framework.

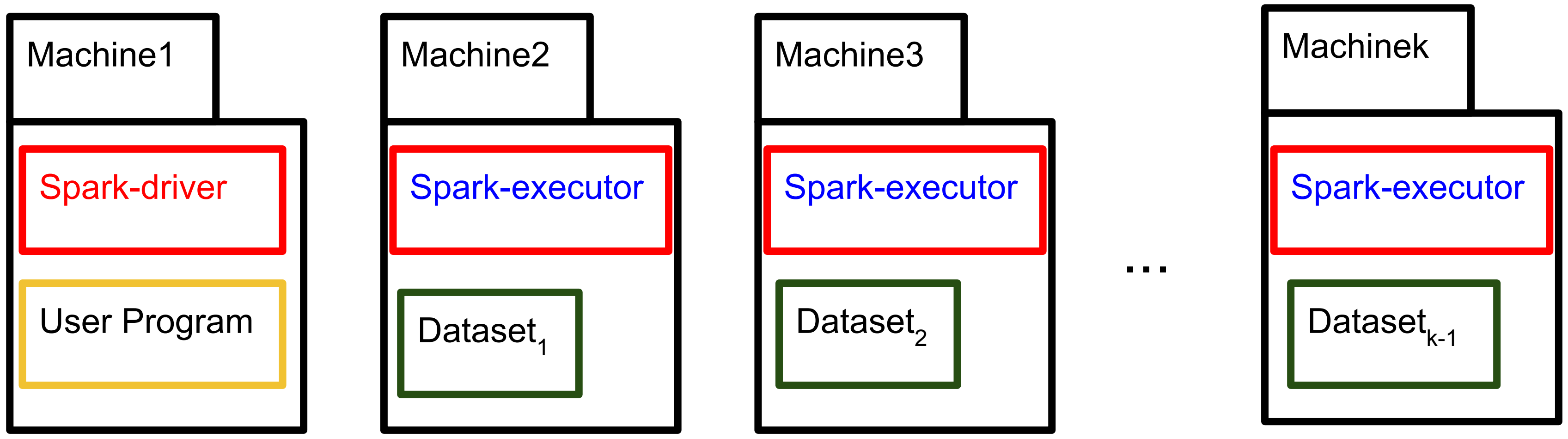
## Setting the Context

1. In particular, we are fully familiar with the Data Engineer role after having explored in detail the Spark Core component of Spark:



## Setting the Context

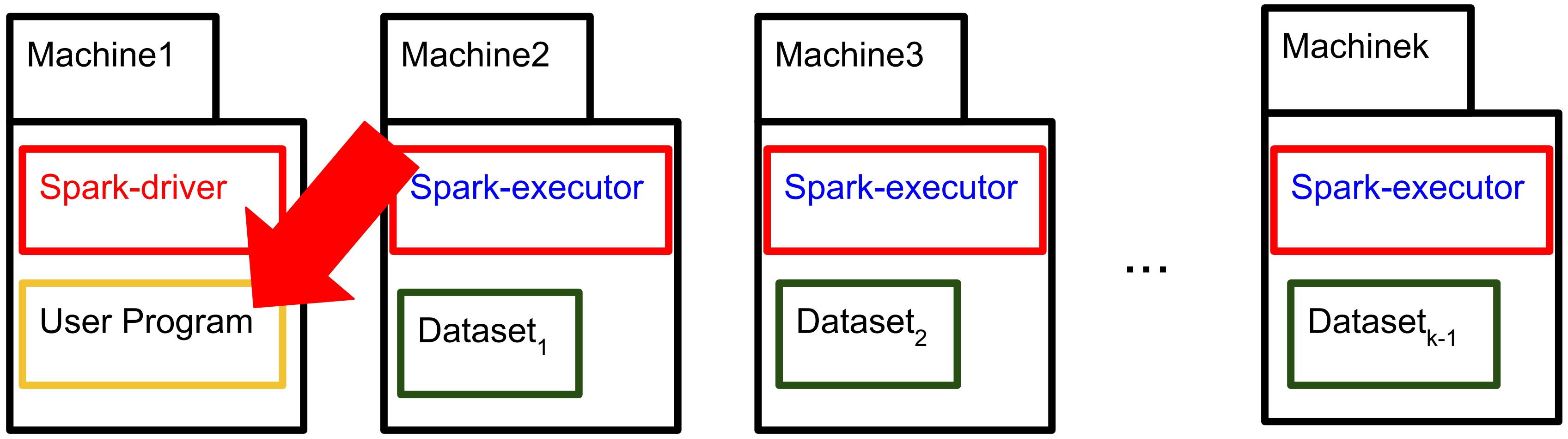
2. We have seen that a Spark program runs in a **cluster of computers**, connected among them so as to support the distributed computation.
   - The Spark driver coordinates the execution of the User program.
   - The Spark executor provide their CPU and memory for the execution of such program.



## Setting the Context

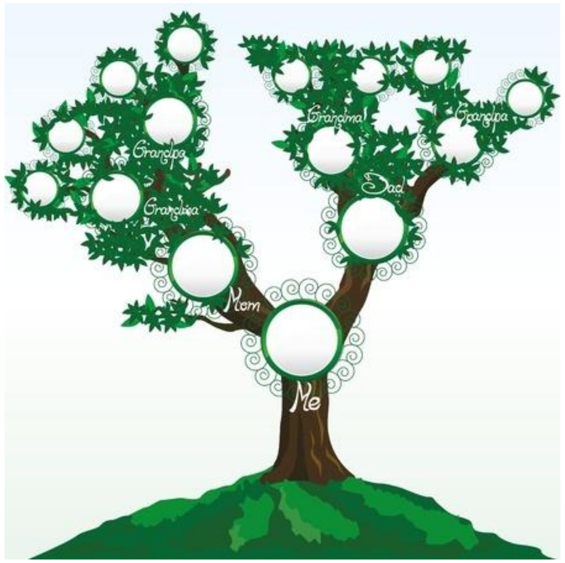
3. We know by now that a Spark Core user program is based on RDDs, and has the following life-cycle:

   a. **Create** some input RDDs from external data.
   b. Transform them to define new RDDs using **transformations**.
   c. **Persist** any intermediate RDDs that will need to be reused.
   d. Launch **actions** to kick off a distributed computation.



## Setting the Context

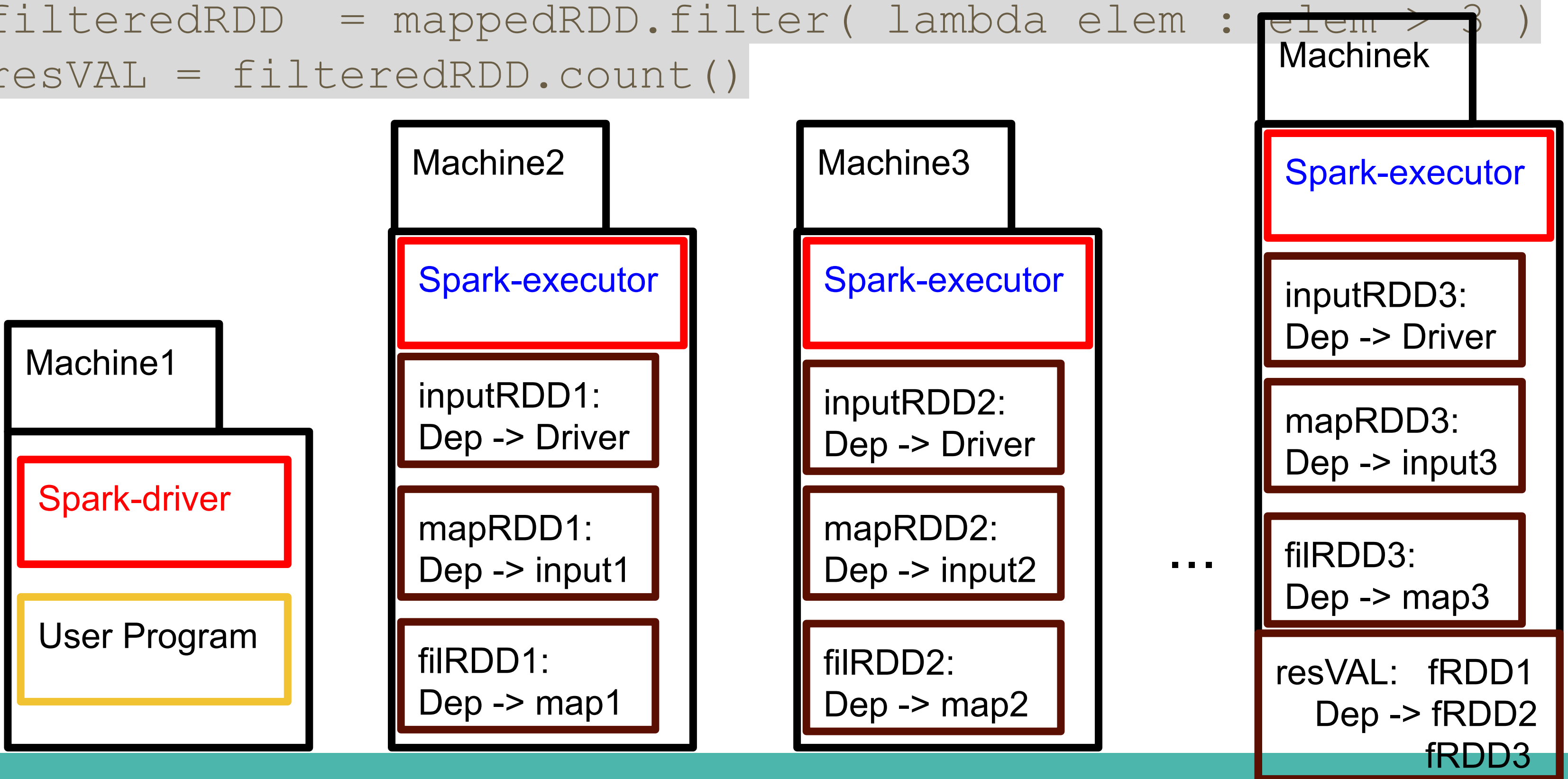
4. We know by now that RDDs are internally represented via...
   - A set of **partitions**
   - Enriched with **lineage** metadata for their re-computation.



## Setting the Context

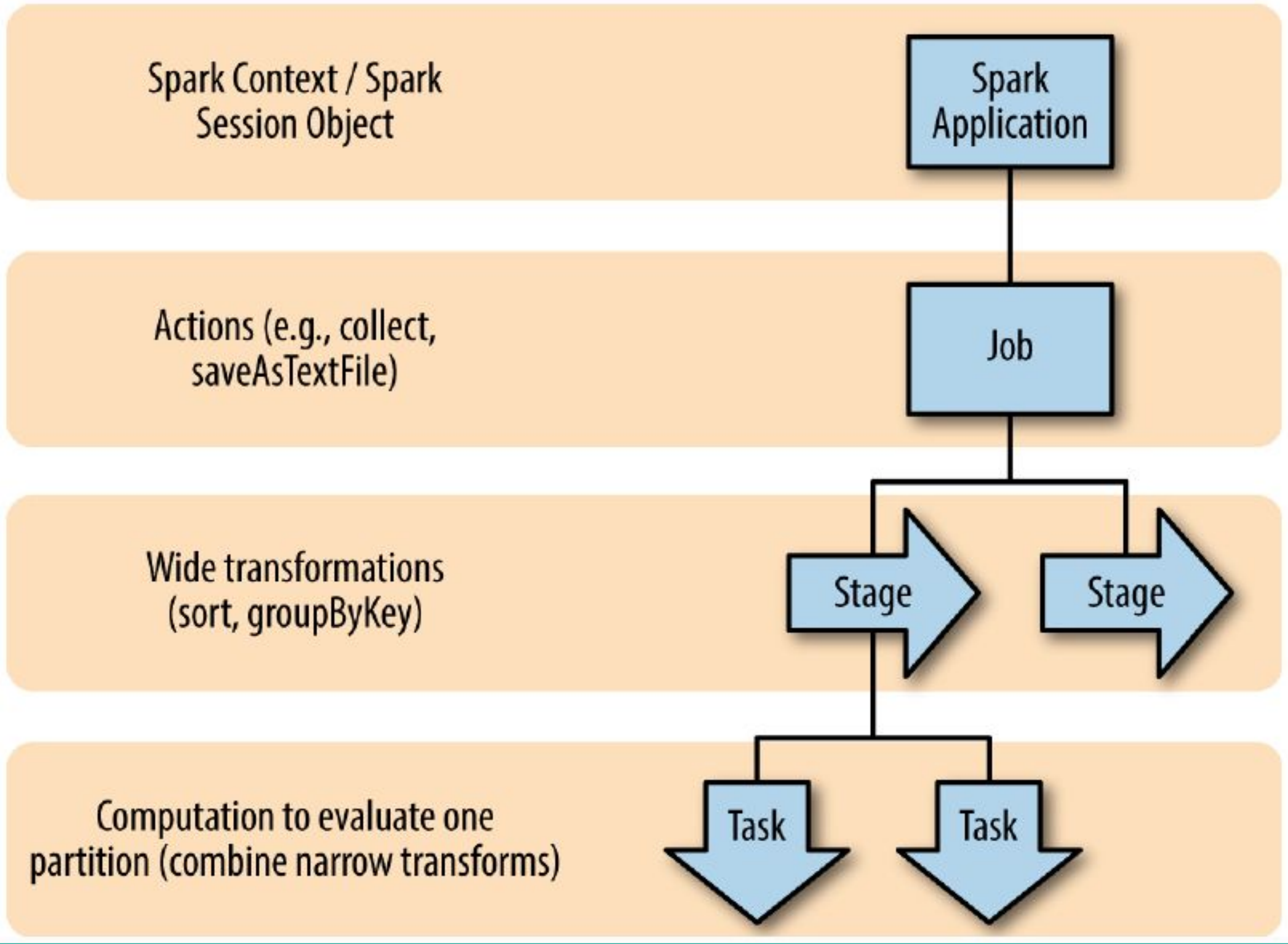
4. We know by now that RDDs are internally represented via...
   - A set of **partitions**
   - Enriched with **lineage** metadata for their re-computation.

```
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()
```



## Setting the Context

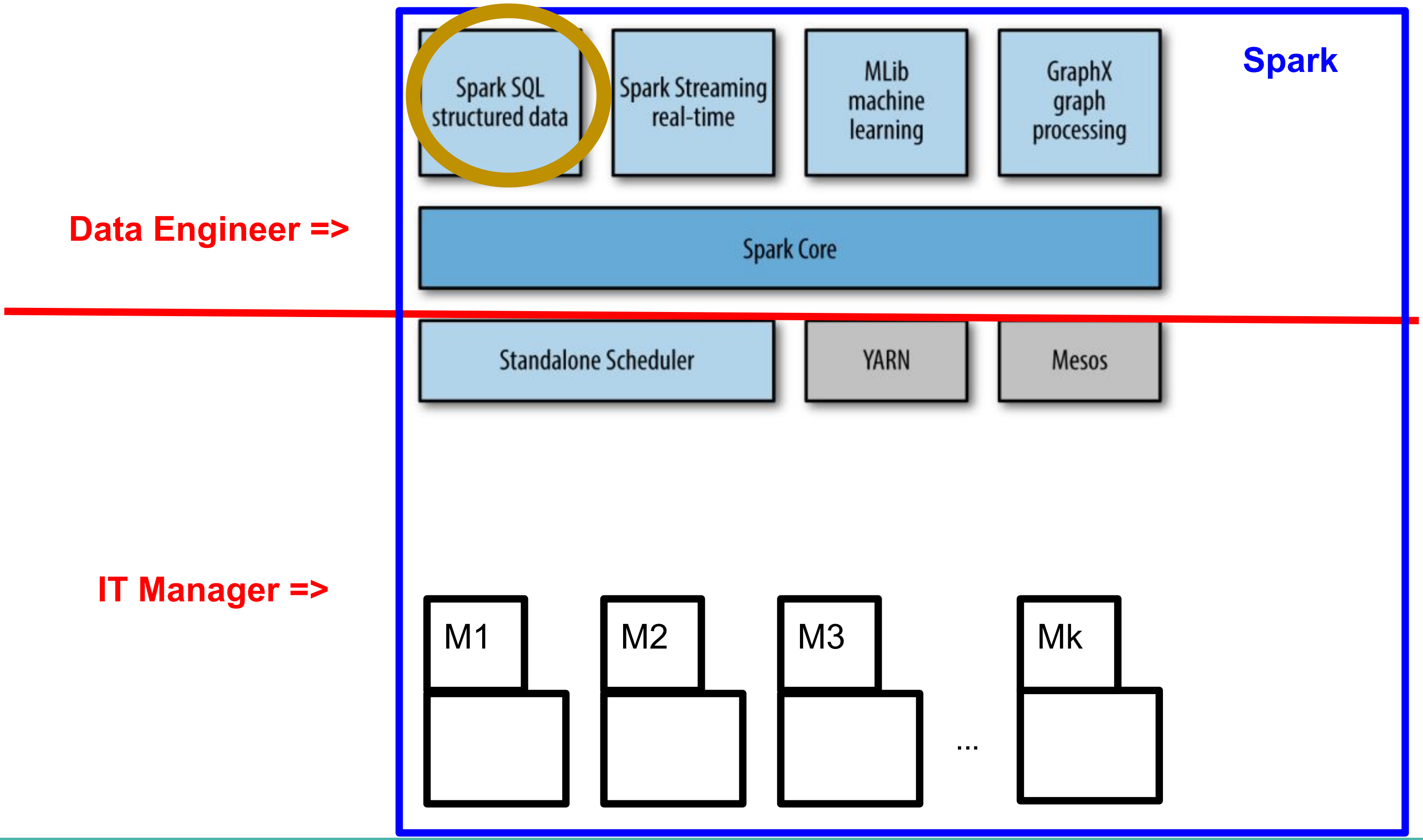
5. We know there are 4 basic concepts for the execution of a Spark Core user program: Application, Job, Stage and Task.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

## Setting the Context

**Goal of this lecture:**

Focus on some of the limitations of Spark Core, and how some improvements can be achieved by using a new component built on top of it: Spark SQL.



## Setting the Context

Spark SQL Motivation:

❖ **Michael Armbrust**, principal software engineer at Databricks, and one of the creators of Spark SQL: https://www.youtube.com/watch?v=EmJjPxOht8s

*"When you look at the original APIs for Spark, the RDDs, they look pretty much like assembly for programming: they give you lots of fine grain control but its a level that most programmers don't care about.*

*Spark SQL gives you these high level abstractions that let you say, declaratively, what you want to compute without saying how it needs to be done.*

*And that gives Spark the freedom to figure out the most efficient way around that computation automatically. It can perform tricks like automatically compiling efficient bytecode, or re-encoding your data in efficient binary format in order to make the computation as fast as possible."*

## Setting the Context

This lecture has been done using as sources:

- Spark SQL: Relational Data Processing in Spark, *Michael Armbrust et al.*, ACM SIGMOD'15 - International Conference on Management of Data, pp. 1383-1394, 2015: https://dl.acm.org/citation.cfm?id=2742797
- Structuring Apache Spark 2.0: SQL, DataFrames, Datasets And Streaming, Michael Armbrust: https://www.youtube.com/watch?v=1a4pgYzeFwE
- High Performance Spark: Best Practices for Scaling & Optmising Apache Spark, *Holden Karau & Rachel Warren*, O'Reilly, 2017.

## Outline

## Functional Spark: Spark Core

Spark Core provides a very expressive functional programming API consisting on:

## Functional Spark: Spark Core

Spark Core provides a very expressive functional programming API consisting on:

- A small set of higher-order transformations and actions…

## Functional Spark: Spark Core

Spark Core provides a very expressive functional programming API consisting on:

- A small set of higher-order transformations and actions…
- …supporting very general functions as part of its higher-order parameters.

## Functional Spark: Spark Core

Spark Core provides a very expressive functional programming API consisting on:

- A small set of higher-order transformations and actions…
- …supporting very general functions as part of its higher-order parameters.

```
newRDD = inputRDD.map( f ) where f: a -> b

resVAL = inputRDD.reduce( f ) where f: a -> a -> a

resVAL = inputRDD.aggregate( accum, f1, f2 )
          where accum:: b, f1: b -> a -> b, f2: b -> b -> b

newRDD = inputRDD.reduceByKey( f ) where f: a -> a

newRDD = inputRDD.combineByKey( f1, f2, f3 )
          where f1: a -> b, f2: b -> a -> b and f3: b -> b -> b
```

## Functional Spark: Spark Core

This gives great flexibility to the Spark Core user:

## Functional Spark: Spark Core

<u>This gives great flexibility to the Spark Core user:</u>

Who has to memorise a small set of primitives…

## Functional Spark: Spark Core

<u>This gives great flexibility to the Spark Core user:</u>

Who has to memorise a small set of primitives...
...and can then apply them in a very wide range of settings.

## Functional Spark: Spark Core

Unfortunately not all are good news.

## Functional Spark: Spark Core

1. Functional Programming-based API.

## Functional Spark: Spark Core

1. Functional Programming-based API.

- We strongly disagree with the quote of Michael Armbrust naming RDDs as the assembly of programming.

## Functional Spark: Spark Core

1. Functional Programming-based API.

- We strongly disagree with the quote of Michael Armbrust naming RDDs as the assembly of programming.
- But is true that functional programming might pose an additional difficulty for programming Spark (specially if you don’t have a Computer Scientist background), and has probably required you to spend some time until getting comfortable around it.

## Functional Spark: Spark Core

2. Internal representation of the RDDs.

# Functional Spark: Spark Core

2. Internal representation of the RDDs.

- We haven't seen it, but the RDD internal function `iterator(p, parentIters)*` is the one in charge of computing (as an iterator) the elements of partition **`p`** given iterators for each of its parent partitions **`parentIters`**.

* Note: This function is not intended to be called directly by the user.
Rather, this is called internally by Spark when performing a job.

## Functional Spark: Spark Core

2. Internal representation of the RDDs.

- The idea of having the function `iterator(p, parentIters)` might look great at first sight, as it provides a unique API entry point for computing the partitions of any RDD, regardless of the transformation being applied.

## Functional Spark: Spark Core

2. Internal representation of the RDDs.

- The idea of having the function `iterator(p, parentIters)` might look great at first sight, as it provides a unique API entry point for computing the partitions of any RDD, regardless of the transformation being applied.

- However, the function is quite opaque about the computation being performed behind the scenes:
  - parentIters can be any list [ pIter1[ T1 ], pIter2[ T1 ], ..., pIterN[ T1 ] ]
  - the function returns a new iterator of type T representing the partition.

## Functional Spark: Spark Core

2. Internal representation of the RDDs.

- The idea of having the function `iterator(p, parentIters)` might look great at first sight, as it provides a unique API entry point for computing the partitions of any RDD, regardless of the transformation being applied.

- However, the function is quite opaque about the computation being performed behind the scenes:
    - parentIters can be any list [ pIter1[ T1 ], pIter2[ T1 ], ..., pIterN[ T1 ] ]
    - the function returns a new iterator of type T representing the partition

**This opaqueness precludes Spark Core from perfoming a number of optimisations under the hood when analysing the user program.**

## Outline

## Structured Spark: Spark SQL

The whole idea of Spark SQL is to provide some structure into the computation.

## Structured Spark: Spark SQL

The whole idea of Spark SQL is to provide some structure into the computation.

That is, to take your computation and arrange it according to some sort of a plan.

## Structured Spark: Spark SQL

- For doing so, it defines a **Domain Specific Language (DSL)** representing the most common patterns present in a data analysis:
    - Selection
    - Filtering
    - Aggregations
    - Joining
    - etc.

## Structured Spark: Spark SQL

- For doing so, it defines a **Domain Specific Language (DSL)** representing the most common patterns present in a data analysis:
  - Selection
  - Filtering
  - Aggregations
  - Joining
  - etc.

- These common patterns are now presented as high level operators that Spark can indeed *understand/reason with*.

## Structured Spark: Spark SQL

- For doing so, it defines a **Domain Specific Language (DSL)** representing the most common patterns present in a data analysis:
  - Selection
  - Filtering
  - Aggregations
  - Joining
  - etc.

- These common patterns are now presented as high level operators that Spark can indeed *understand/reason with*.
- This leverages a number of optimisations when analysing the user program with the whole point of making the job computations more performant.

## Structured Spark: Spark SQL

Now,
moving from the general Spark Core primitives
to these DSL operators
indeed <u>reduce our expressiveness</u>.

## Structured Spark: Spark SQL

- Whereas in Spark Core we could express pretty much anything, in Spark SQL we are tightened by the Catalog of DSL operators.

## Structured Spark: Spark SQL

This is undesirable, but absolutely needed.

## Structured Spark: Spark SQL

This is undesirable, but absolutely needed.

- For any new DSL operator to be included in the Catalog by the Spark developers…

## Structured Spark: Spark SQL

This is undesirable, but absolutely needed.

- For any new DSL operator to be included in the Catalog by the Spark developers...
- ...it has to come with a clear understanding on how can Spark reasons with it in order to leverage the aforementioned optimisations.

## Structured Spark: Spark SQL

- To alleviate the lack of expressiveness,
  on each new release version of Spark the developers include
  new DSL operators (sometimes after users requesting for them).

- Version ...
- Version 2.4.3:

- Version 2.4.4:

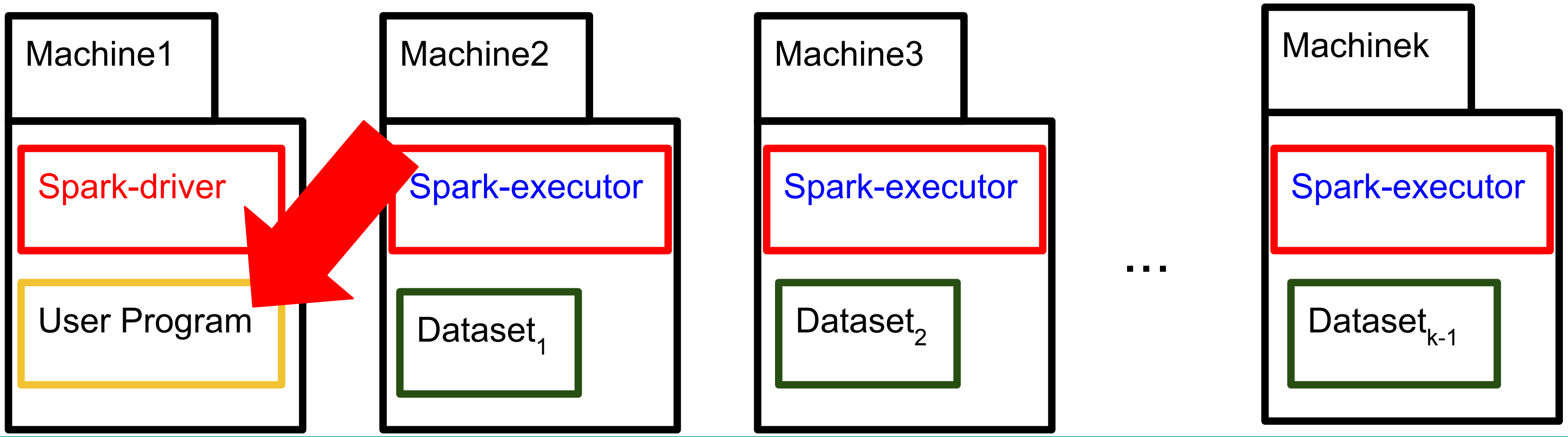
## Structured Spark: Spark SQL

- Nowadays, it is fair to say that the DSL language is expressive enough to accommodate the vast majority of computations.

## Structured Spark: Spark SQL

- That is, the Catalog of **creation**, **transformations**, **persistence** and **actions** operations of Spark SQL is expressive enough that it can express nearly the same amount of things we can express with the **creation**, **transformations**, **persistence** and **actions** primitives of Spark Core.

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

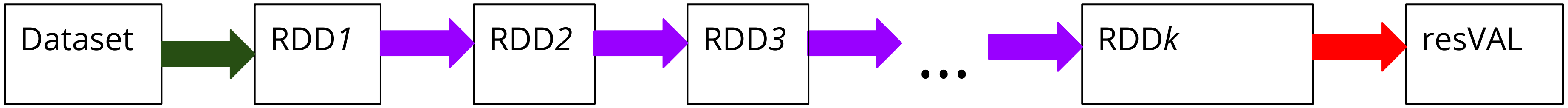


## Structured Spark: Spark SQL

- That is, the Catalog of **creation**, **transformations**, **persistence** and **actions** operations of Spark SQL is expressive enough that it can express nearly the same amount of things we can express with the **creation**, **transformations**, **persistence** and **actions** primitives of Spark Core.

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

Following Spark Core RDD-based primitives:

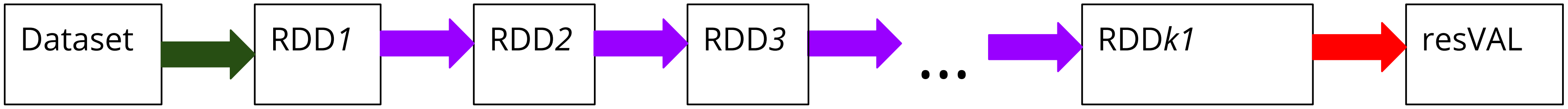


# Structured Spark: Spark SQL

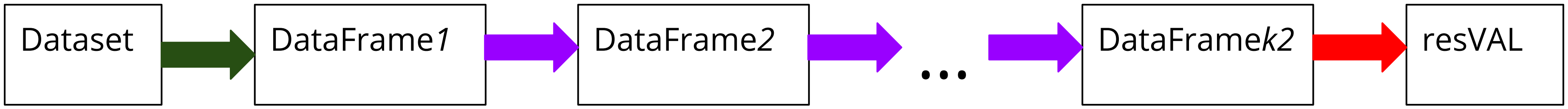
- That is, the Catalog of **creation**, **transformations**, **persistence** and **actions** operations of Spark SQL is expressive enough so that it can express nearly the same amount of things we can express with the **creation**, **transformations**, **persistence** and **actions** primitives of Spark Core.

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

Following Spark Core RDD-based primitives:



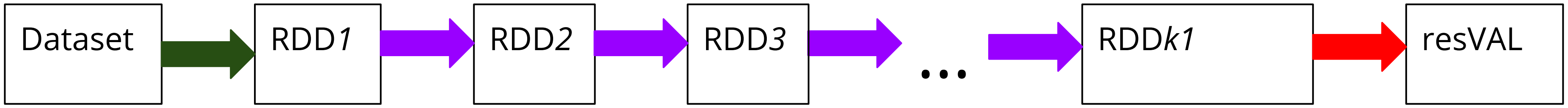
Following Spark SQL Catalog of DSL operators:



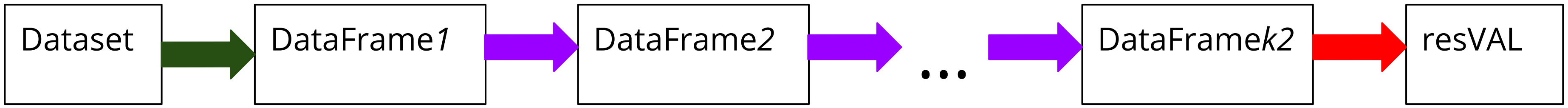
## Structured Spark: Spark SQL

- That is, the Catalog of **creation**, **transformations**, **persistence** and **actions** operations of Spark SQL is expressive enough so that it can express nearly the same amount of things we can express with the **creation**, **transformations**, **persistence** and **actions** primitives of Spark Core.

Following Spark Core RDD-based primitives:



Following Spark SQL Catalog of DSL operators:

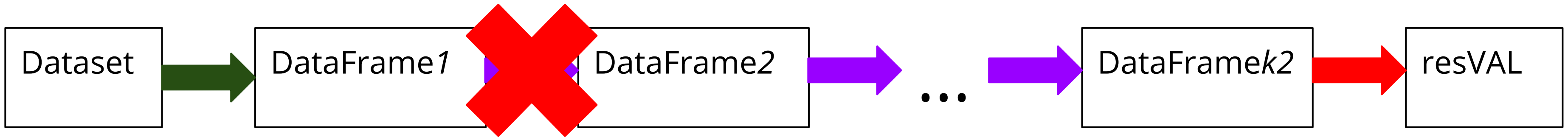


**Important remark:**

**The above diagram does not mean, in any way, that there is a 1 to 1 conversion between Spark Core RDD-based primitives and Spark SQL DSL operators. Computations will be different, both in number and type of operations being used.**

## Structured Spark: Spark SQL

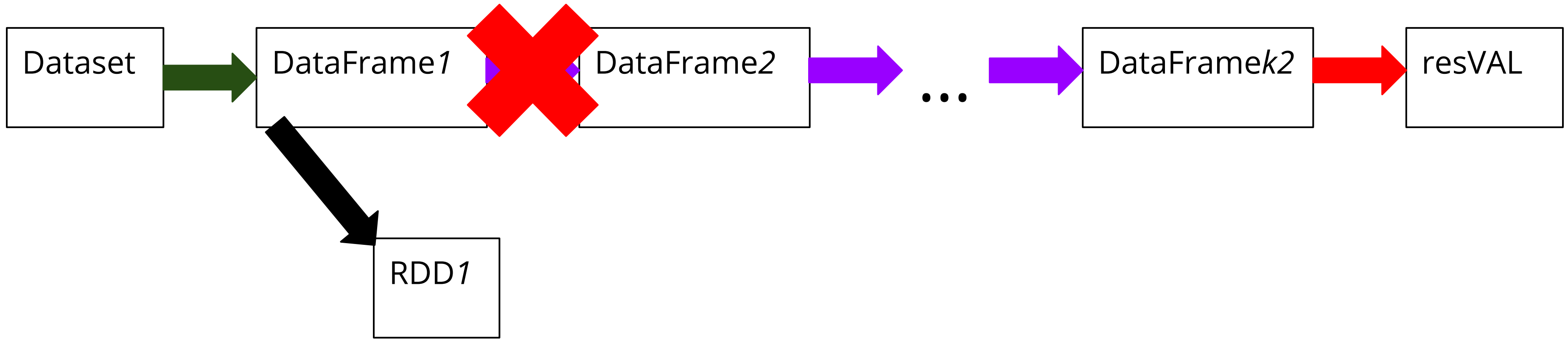
Finally, if worst comes to worst and there is something that cannot be expressed with the DSL catalog...



## Structured Spark: Spark SQL

Finally, if worst comes to worst and there is something that cannot be expressed with the DSL catalog...

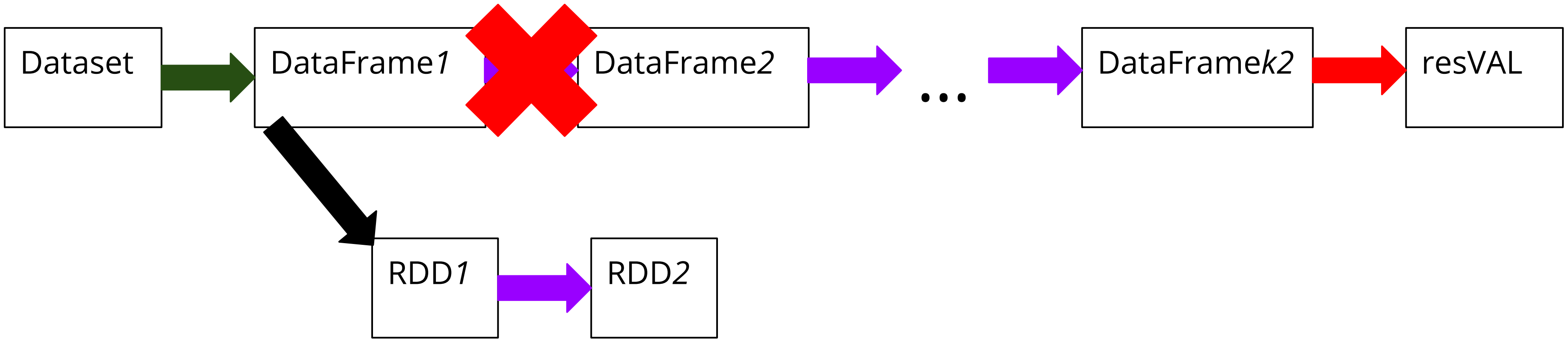
- We can always revert back to RDDs.



# Structured Spark: Spark SQL

Finally, if worst comes to worst and there is something that cannot be expressed with the DSL catalog…

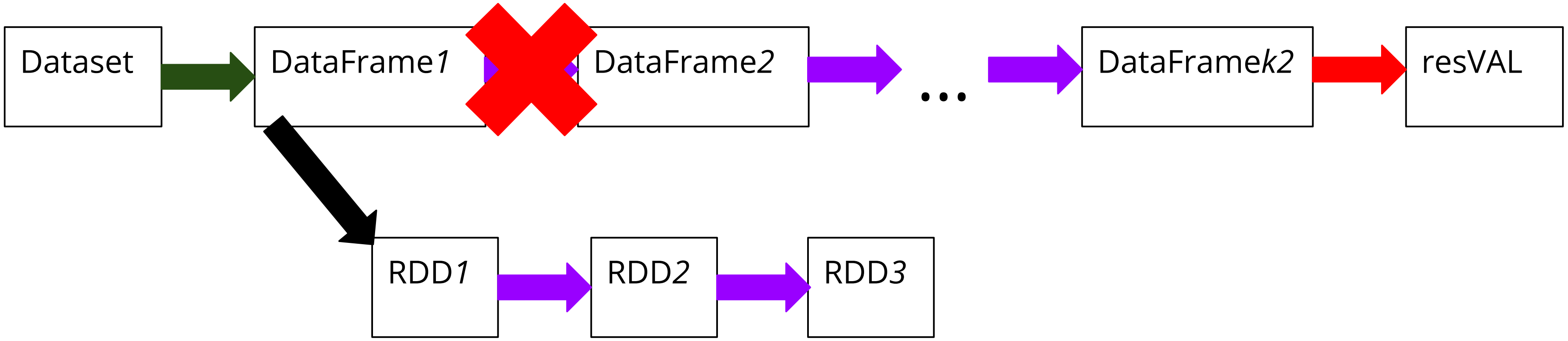
- We can always revert back to RDDs.
- Express the operation using one or more RDDs primitives.



## Structured Spark: Spark SQL

Finally, if worst comes to worst and there is something that cannot be expressed with the DSL catalog...

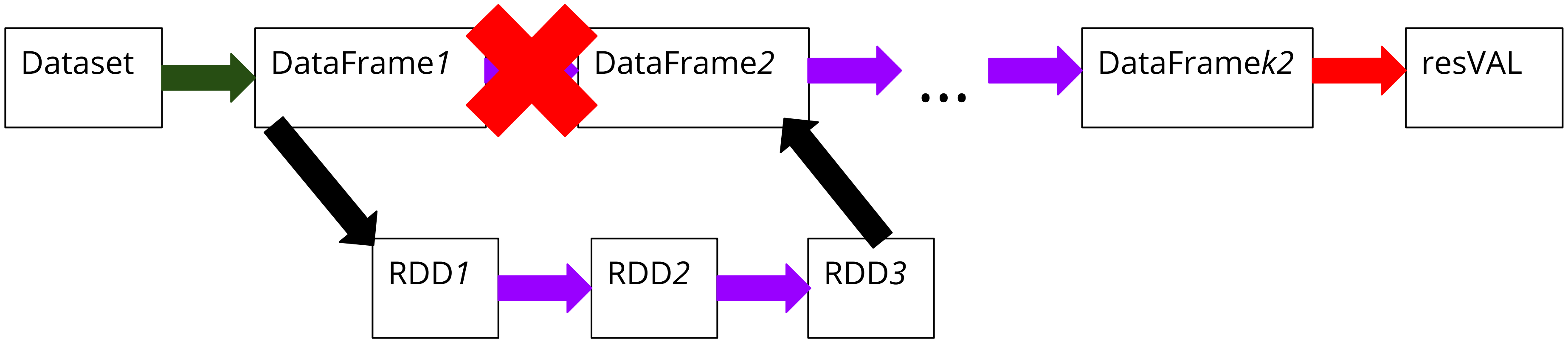
- We can always revert back to RDDs.
- Express the operation using one or more RDDs primitives.



## Structured Spark: Spark SQL

Finally, if worst comes to worst and there is something that cannot be expressed with the DSL catalog…

- We can always revert back to RDDs.
- Express the operation using one or more RDDs primitives.
- And come back to Spark SQL afterwards to continue from there.



## Structured Spark: Spark SQL

The whole Catalog of DSL operators is available at:
https://spark.apache.org/docs/latest/api/python/pyspark.sql.html

## Structured Spark: Spark SQL

But, before we study the most representative operations of this Catalog…

## Structured Spark: Spark SQL

But, before we study the most representative operations of this Catalog...

...When looking at the last slices,
you probably would have been wondering:

## Structured Spark: Spark SQL

But, before we study the most representative operations of this Catalog...

...When looking at the last slices,
you probably would have been wondering:

***who are these DataFrames?***

## Outline

## DataFrames & Datasets: Public Side

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

## DataFrames & Datasets: Public Side

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

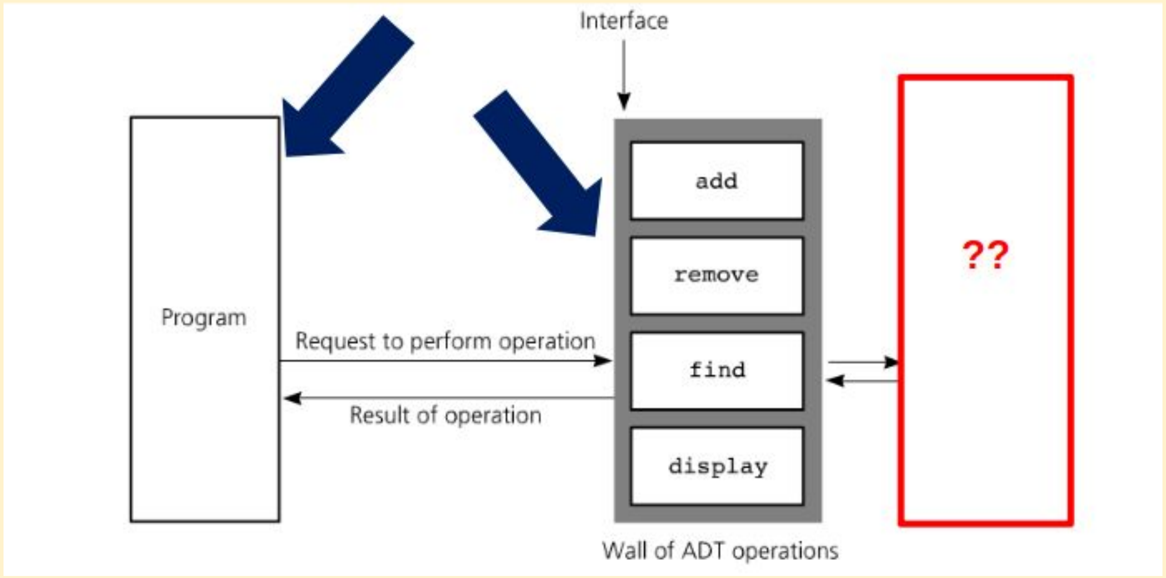
...now Spark SQL makes possible *structured Spark* via the ADTs DataFrame and Dataset.

## DataFrames & Datasets: Public Side

*Let's do a brief recap about ADTs*

- 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?

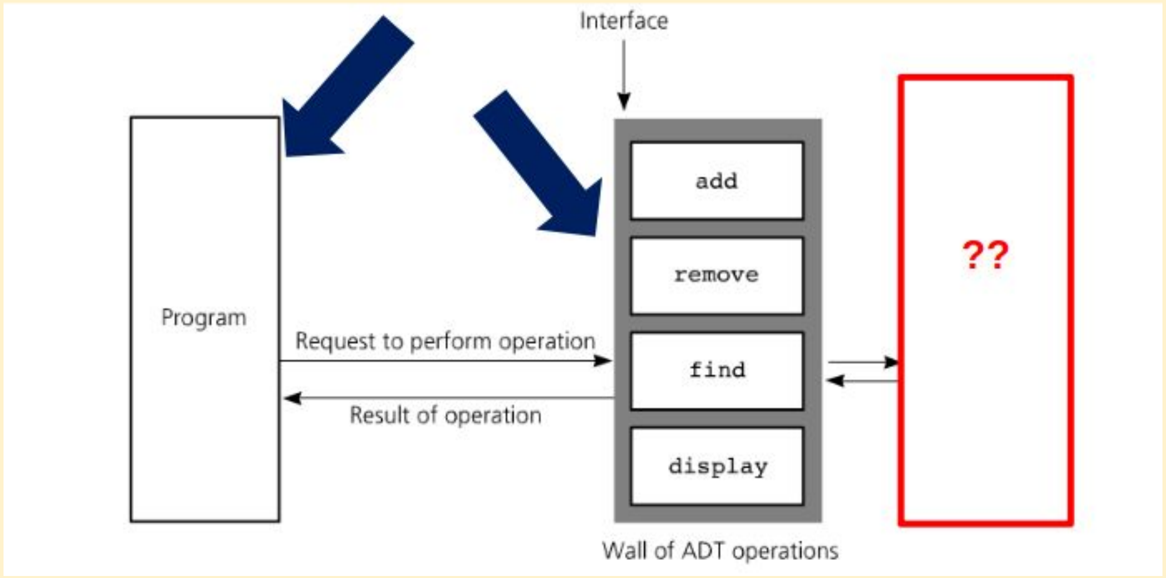


## DataFrames & Datasets: Public Side

*Let's do a brief recap about ADTs*

- 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.

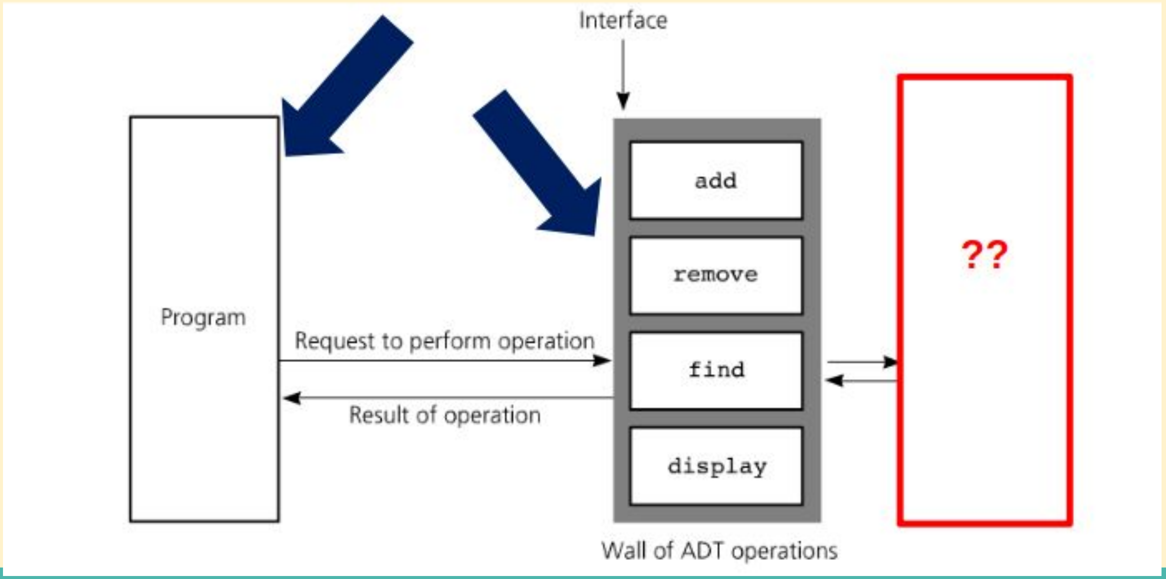


## DataFrames & Datasets: Public Side

*Let's do a brief recap about ADTs*

- 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.



## DataFrames & Datasets: Public Side

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**.

## DataFrames & Datasets: Public Side

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.

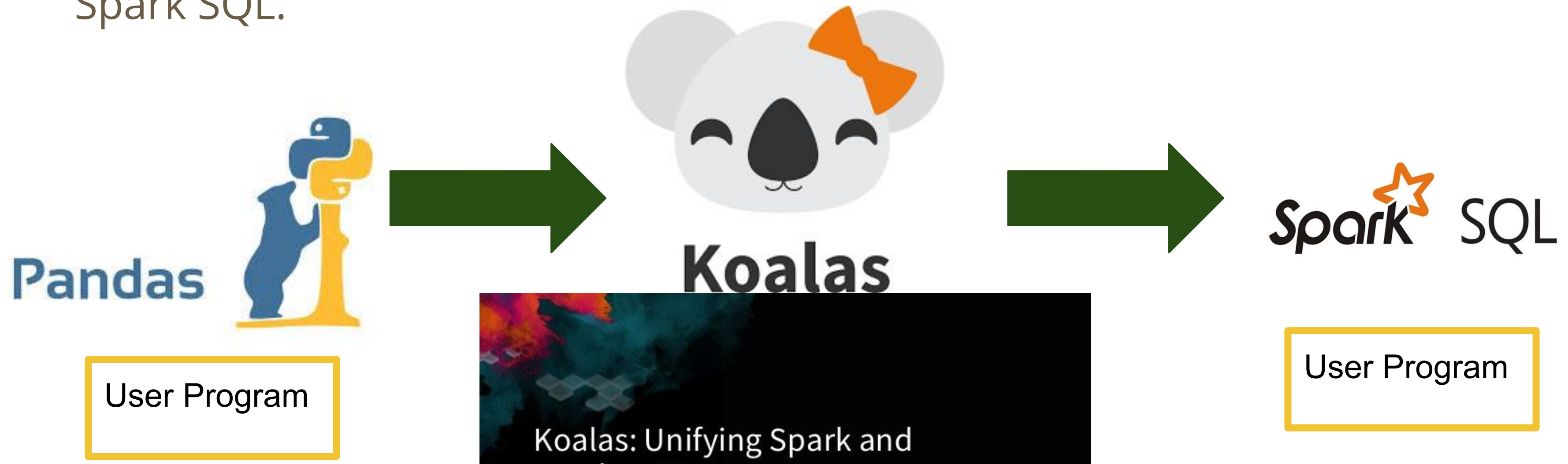
## DataFrames & Datasets: Public Side

- DataFrames are similar enough to Python Pandas DataFrames. Although, again, they work in a distributed environment setting.

## DataFrames & Datasets: Public Side

- Indeed, as a side note, there is a recent enough Databricks project called Koalas.

- It aims to take a pure Python Pandas DataFrame-based program, and turn it into an equivalent Python Spark DataFrame-based program, ready to work on Spark SQL.



Koalas: Unifying Spark and pandas APIs

Takuya UESHIN
Spark Meetup Tokyo #1, Jun 2019

databricks

## DataFrames & Datasets: Public Side

Let's go with the Dataset ADT public side:

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

- A Dataset enriches a DataFrame with type-safety features, and thus it is only available in Scala and Java.

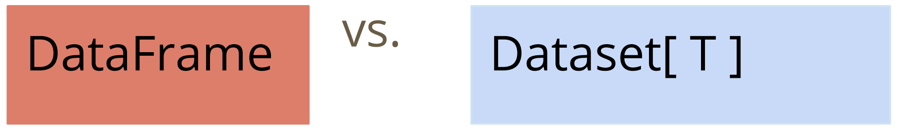
DataFrame vs. Dataset[ T ]

## DataFrames & Datasets: Public Side

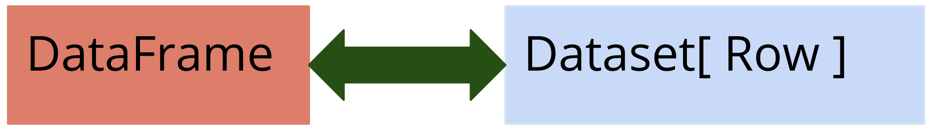
Let's go with the Dataset ADT public side:

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

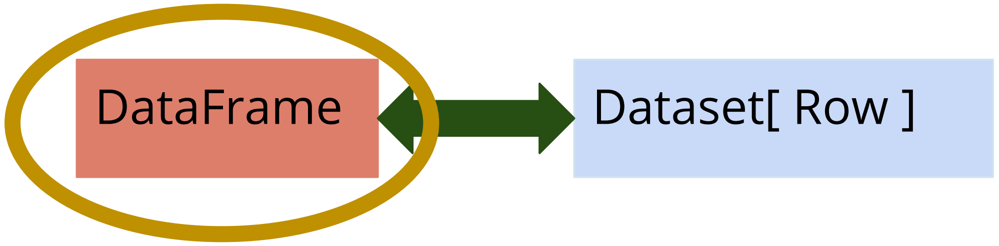
- A Dataset enriches a DataFrame with type-safety features, and thus it is only available in Scala and Java.



Indeed, a DataFrame can be viewed as a Dataset of generic type Row.



## DataFrames & Datasets: Public Side



<u>Thus, we can use DataFrames when:</u>

- Programming in Python

  or

- When programming in Scala <u>and</u> hitting a case in which you don't know all the fields of your data entries ahead of time, and you don't want to compile a class that is going to hold your data.

  After all, a DataFrame or Dataset of generic Rows can be later on cast/specialised to a more concrete data type if needed.

## DataFrames & Datasets: Public Side

The data model for DataFrames/Datasets is pretty extensive, supporting:

## DataFrames & Datasets: Public Side

The data model for DataFrames/Datasets is pretty extensive, supporting:

- **Primitive datatypes:**
  - Integer
  - Float
  - String
  - Boolean
  - Timestamp
  - Date
  - etc…

## DataFrames & Datasets: Public Side

The data model for DataFrames/Datasets is pretty extensive, supporting:

- **Primitive datatypes:**
  - Integer
  - Float
  - String
  - Boolean
  - Timestamp
  - Date
  - etc...

- **Composable datatypes:**
  - Arrays
  - Tuples (Structs)
  - Dictionaries (Maps)

## Creator Operations

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

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

- As discussed, the API for DataFrames/Datasets is based in the Catalog of DSL operators.

## Creator Operations

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.

- As discussed, the API for DataFrames/Datasets is based in the Catalog of DSL operators.
- But, essentially, these DSL operators can be classified into **Creation**, **Transformation**, **Persistance** and **Actions**, as it was the case for the Spark Core RDD-based primitives.

## Creator Operations

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.

- In this sense, a DataFrame/Dataset simply represents a logical plan (i.e., a way of computing an amount of data, for example by reading and filtering).

## Creator Operations

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.

- In this sense, a DataFrame/Dataset simply represents a logical plan (i.e., a way of computing an amount of data, for example by reading and filtering).
- But, as it is computed lazily, nothing happens again until an action is called. When it happens, Spark SQL builds a physical plan to compute the final result.

## DataFrames & Datasets: Public Side

But,
before discussing these logical & physical plans,
(which indeed belong to the private side of
DataFrames and Datasets)
let's study the most representative subset of the
Catalogue of DSL operators.

## DataFrames & Datasets: Public Side

As Python is our main programming language, we will mainly focus on DataFrames.

If needed, we will switch sometimes to Scala to see the counterpart example in Datasets.

## Outline

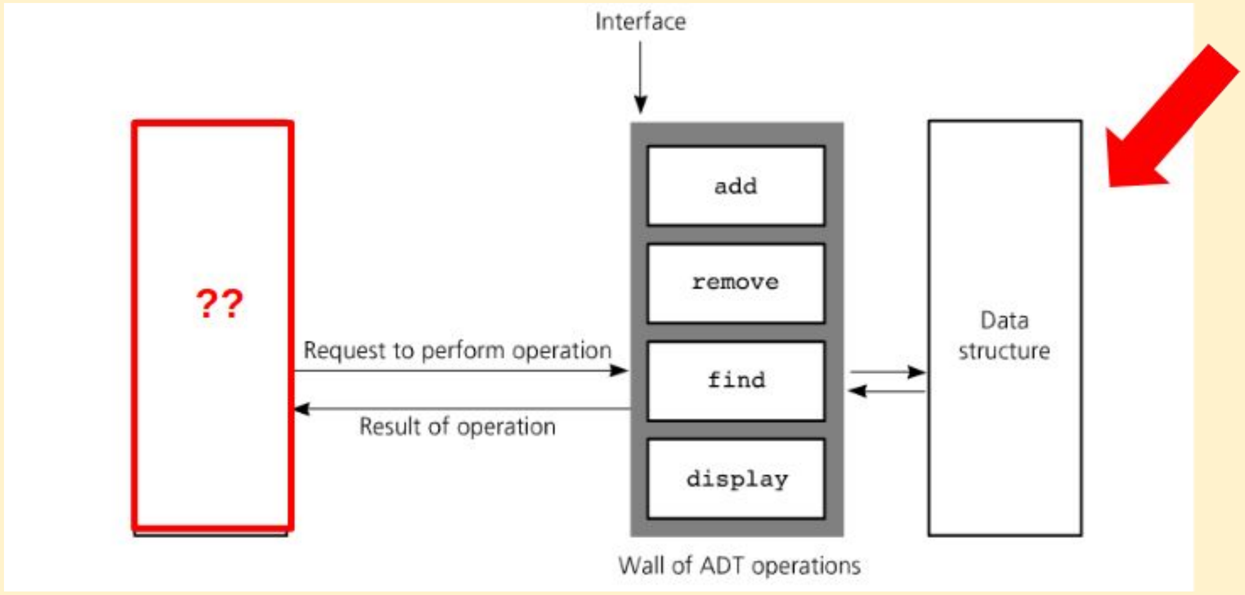
1. Setting Up the Context.
2. Functional Spark: Spark Core.
3. Structured Spark: Spark SQL.
4. DataFrames & Datasets: Public Side.
5. DataFrames & Datasets: Private Side.

## DataFrames & Datasets: Private Side

*Let's do a brief recap about ADTs*

- 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.

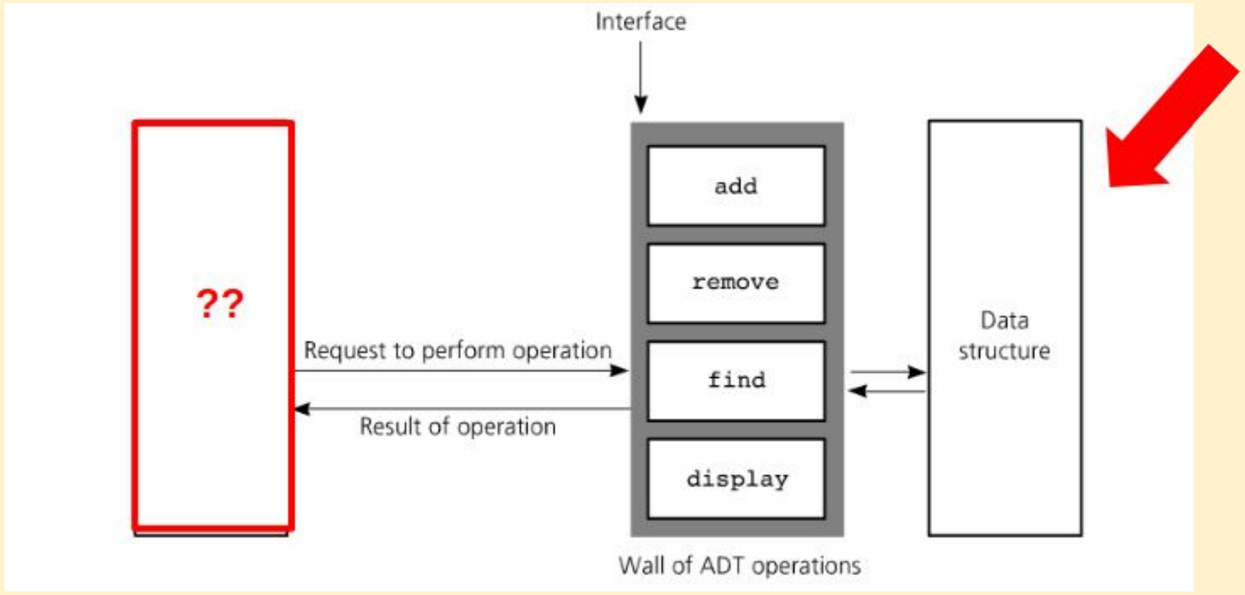


## DataFrames & Datasets: Private Side

*Let's do a brief recap about ADTs*

- 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?



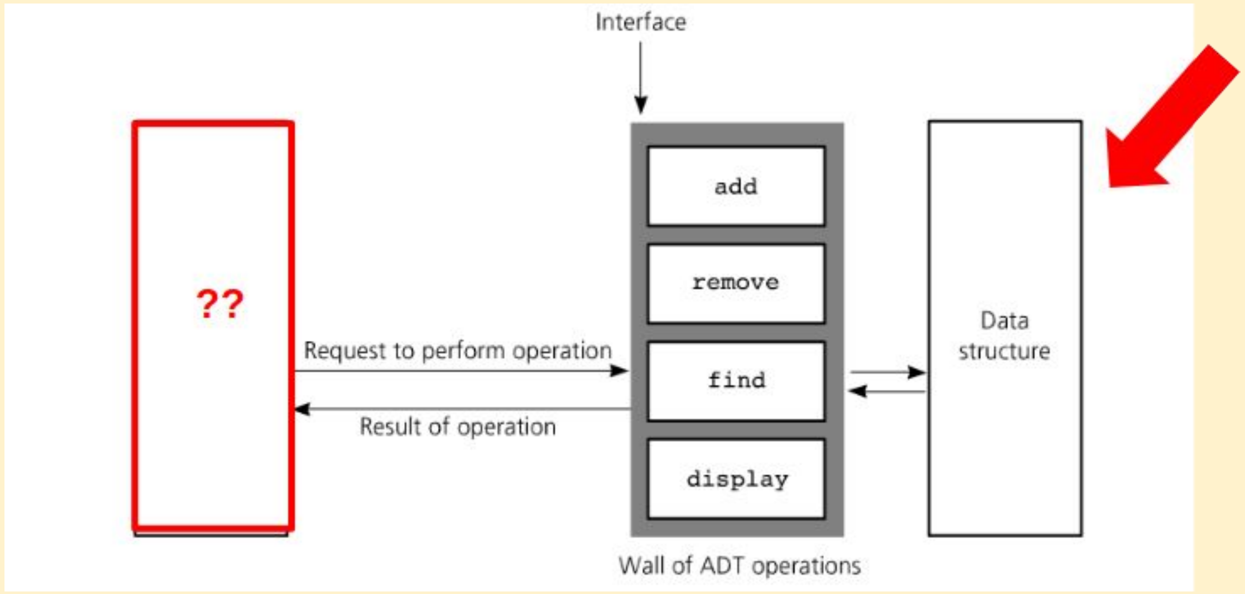
## DataFrames & Datasets: Private Side

*Let's do a brief recap about ADTs*

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

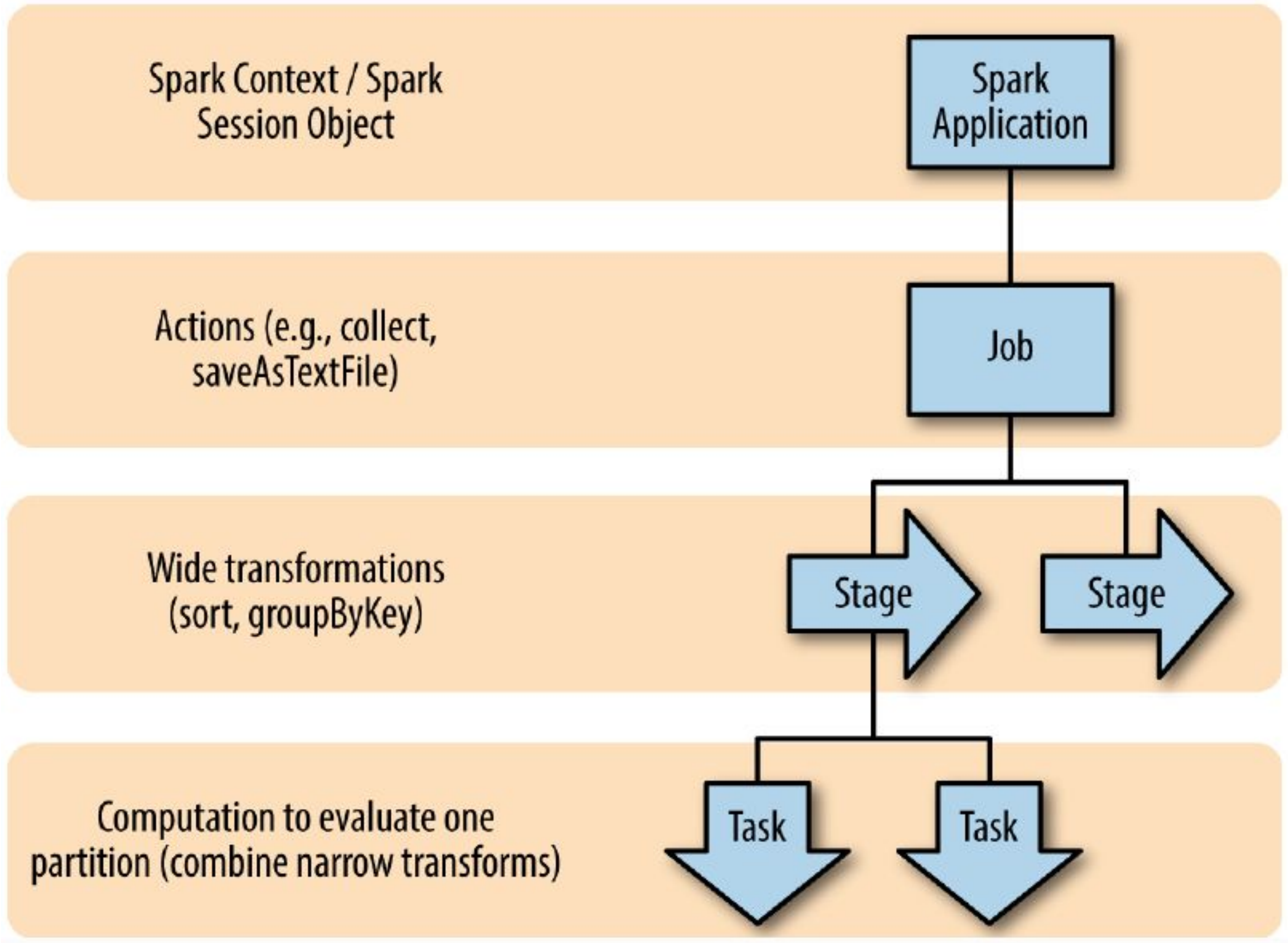
3. **How** is the data internally represented?
   Specify the concrete data structures used to layout the data.
4. **How** is each operation internally implemented?

- However, the private side does not need to worry about the future user of the data.



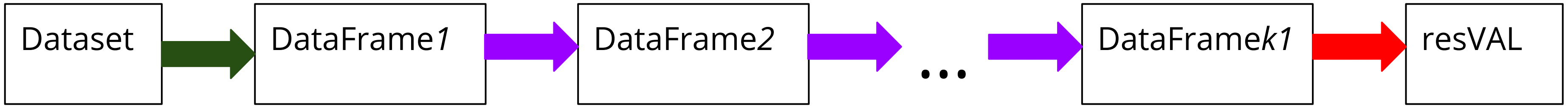
## DataFrames & Datasets: Private Side

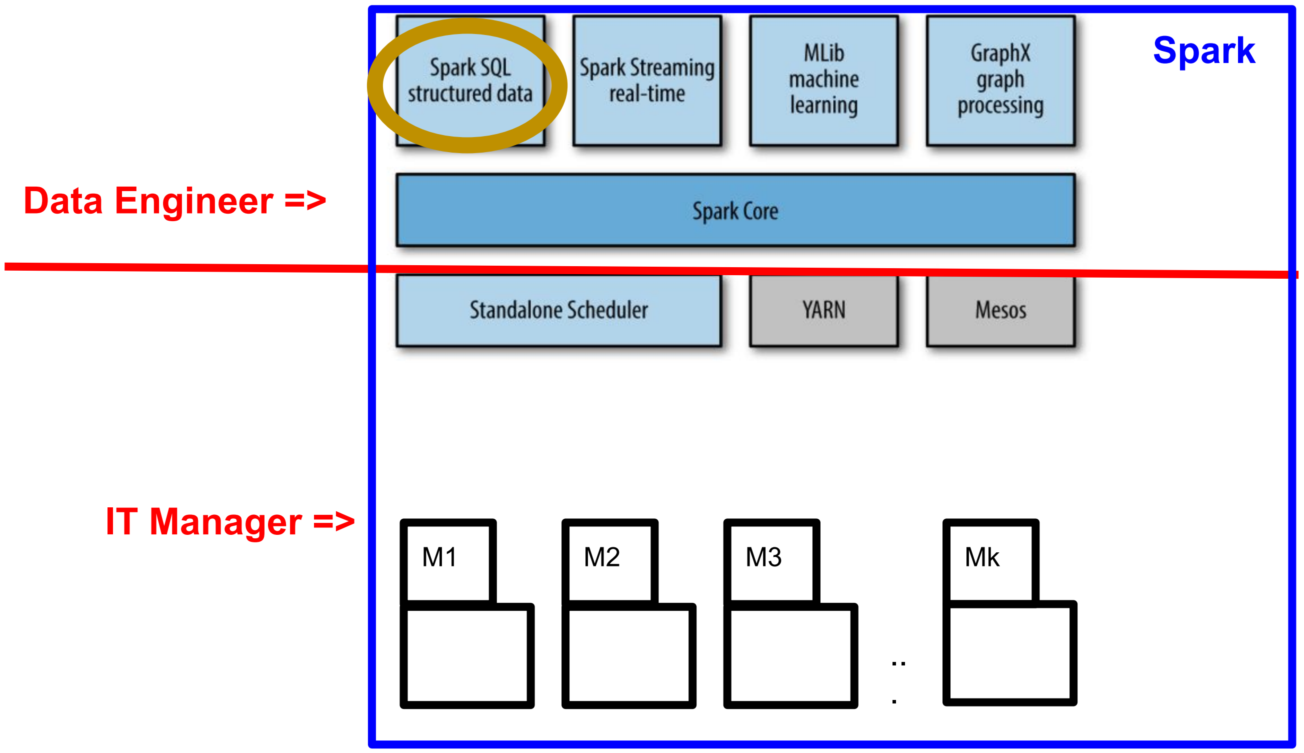
- The only way Spark can execute an application is by splitting it into jobs, stages and tasks operating in RDDs, as it was explained for Spark Core.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

## DataFrames & Datasets: Private Side

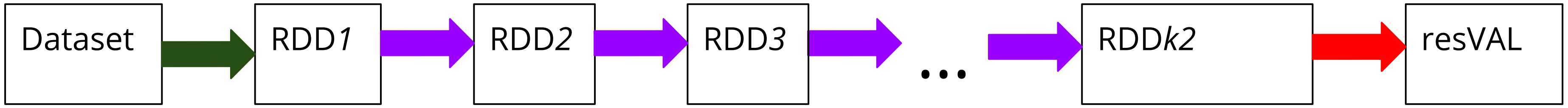
- This means that any Spark SQL program (DataFrames/Datasets-based) has to be first translated into an equivalent Spark Core RDD-based program.

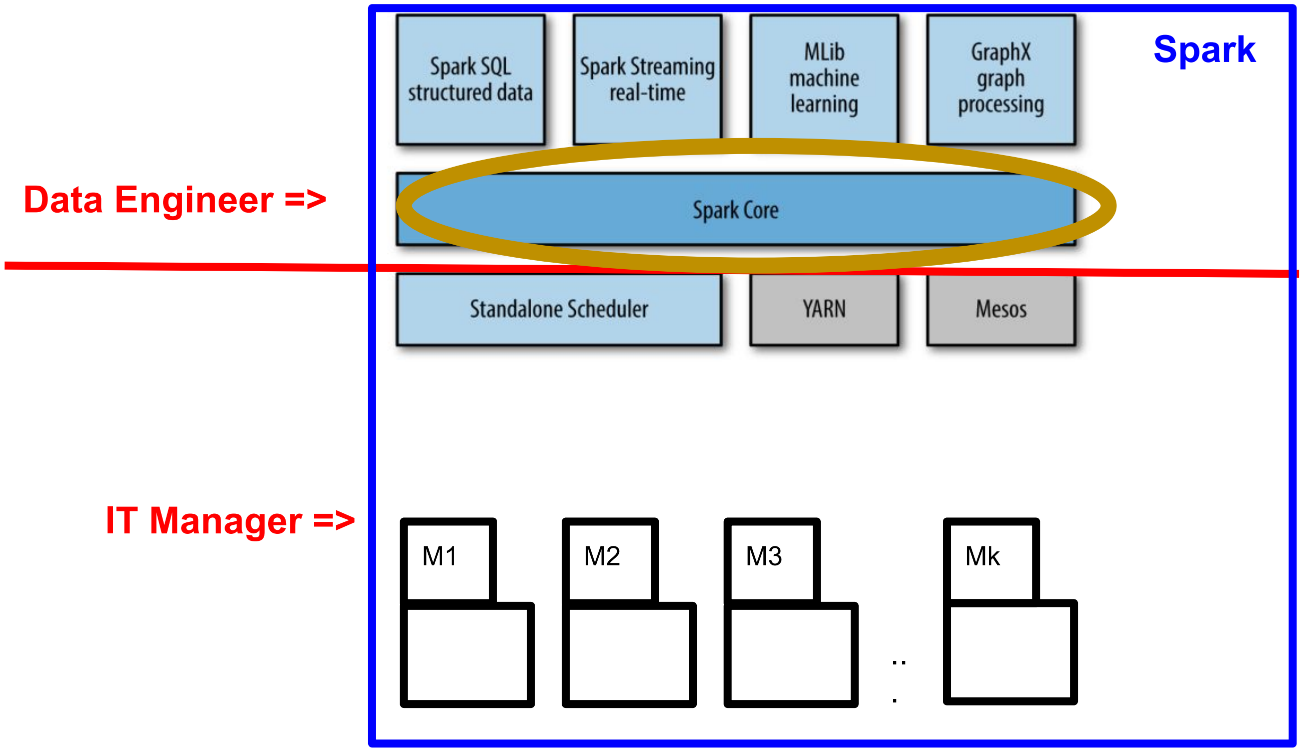




## DataFrames & Datasets: Private Side

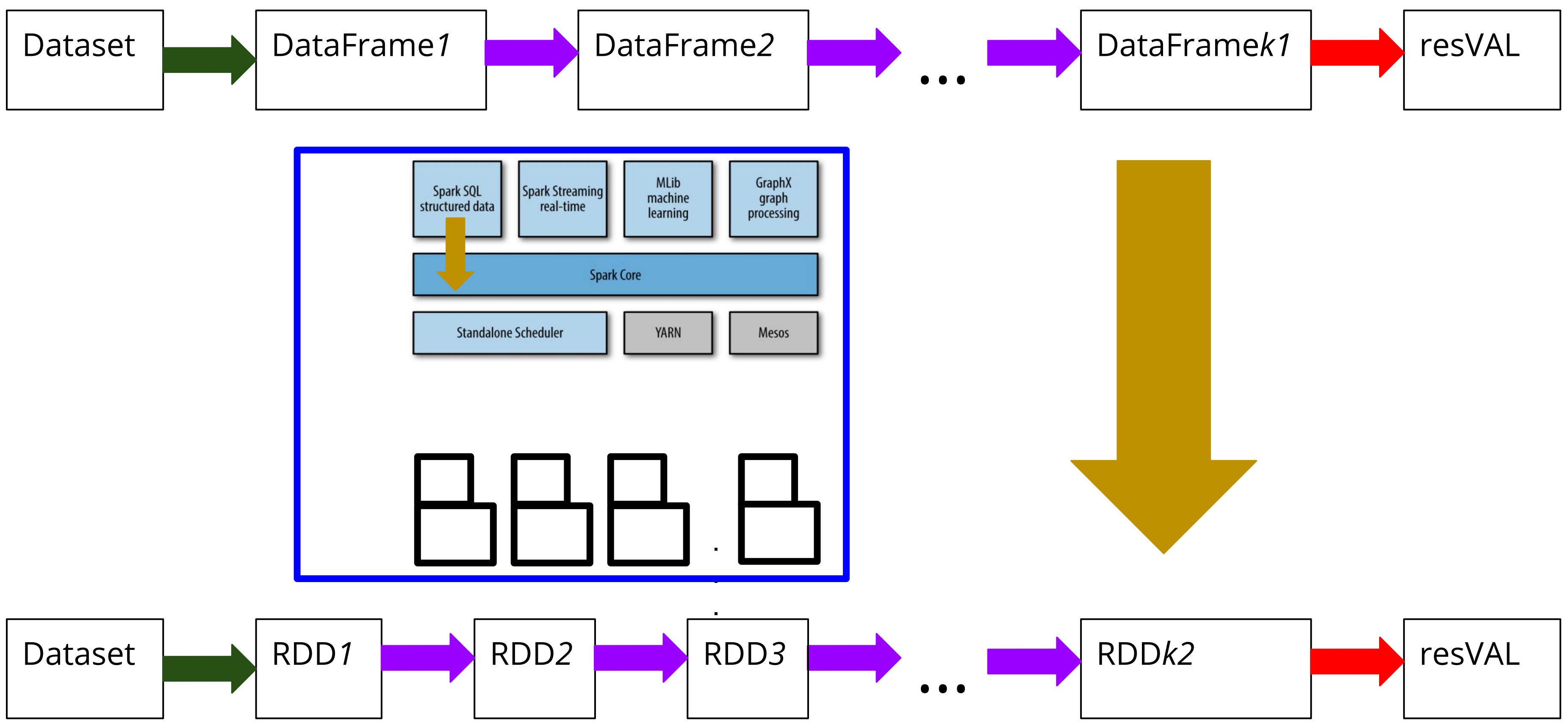
- This means that any Spark SQL program (DataFrames/Datasets-based) has to be first translated into an equivalent Spark Core RDD-based program.





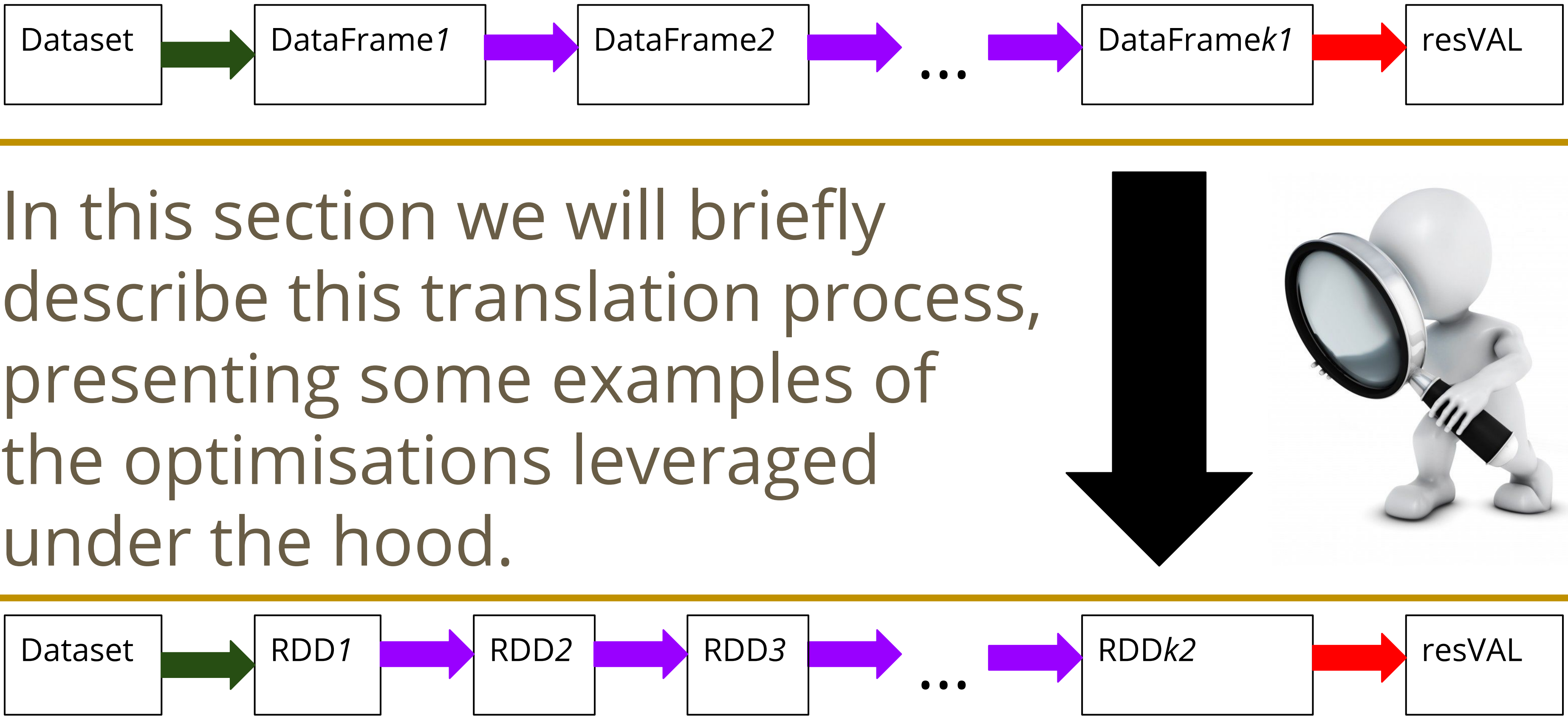
## DataFrames & Datasets: Private Side

- This means that any Spark SQL program (DataFrames/Datasets-based) has to be first translated into an equivalent Spark Core RDD-based program.



# DataFrames & Datasets: Private Side

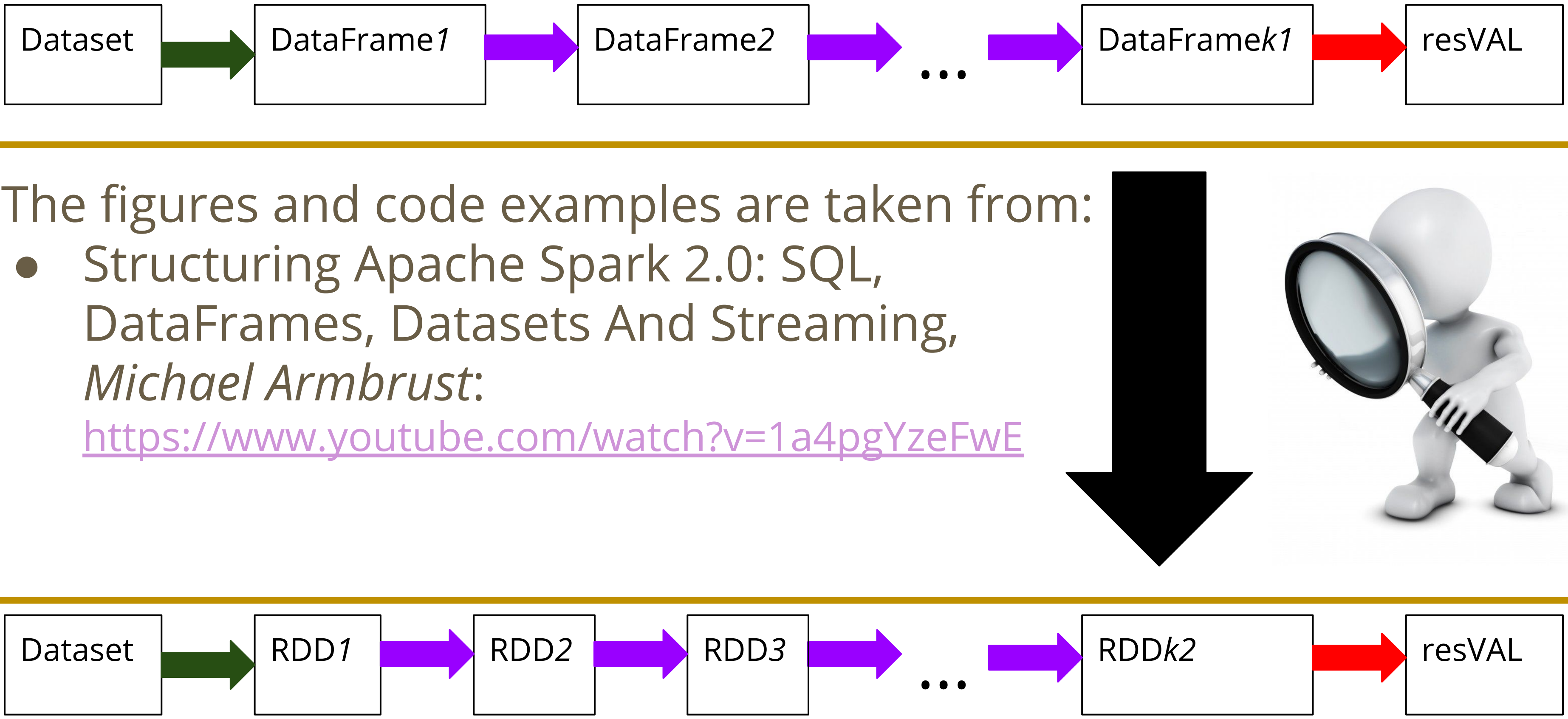
- This means that any Spark SQL program (DataFrames/Datasets-based) has to be first translated into an equivalent Spark Core RDD-based program.



In this section we will briefly describe this translation process, presenting some examples of the optimisations leveraged under the hood.

## DataFrames & Datasets: Private Side

- This means that any Spark SQL program (DataFrames/Datasets-based) has to be first translated into an equivalent Spark Core RDD-based program.



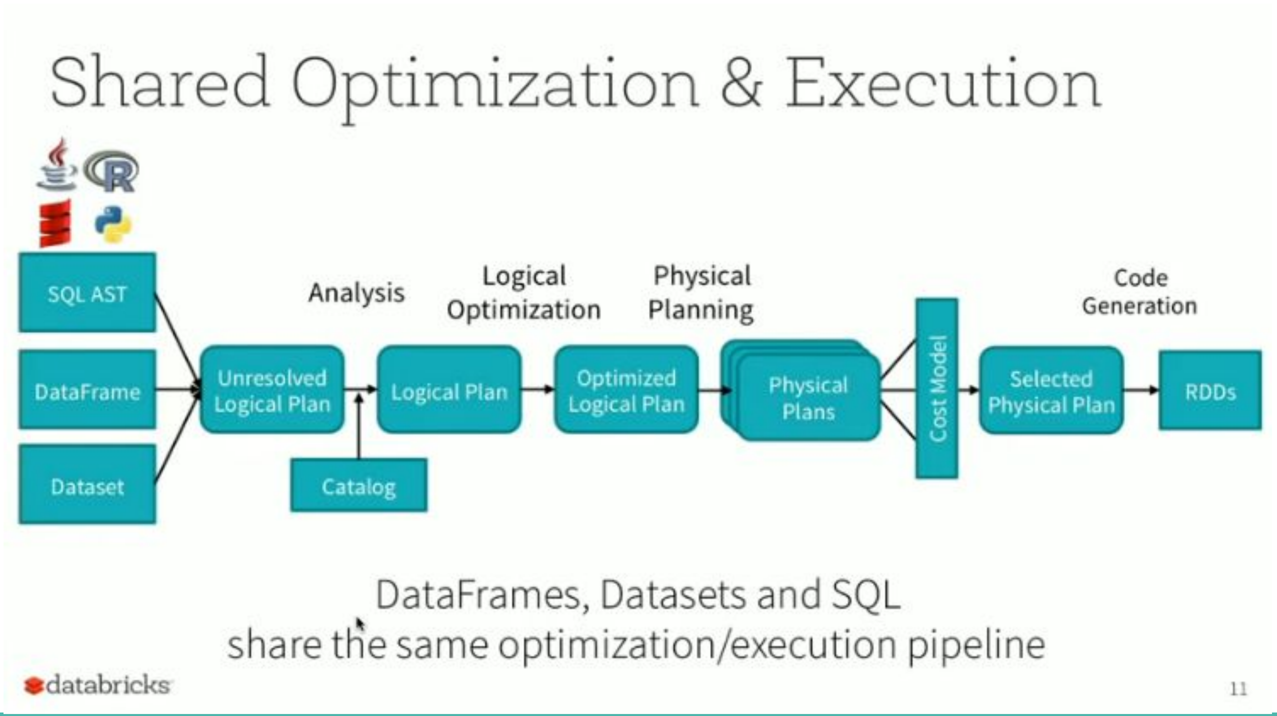
The figures and code examples are taken from:

- Structuring Apache Spark 2.0: SQL, DataFrames, Datasets And Streaming, *Michael Armbrust*: https://www.youtube.com/watch?v=1a4pgYzeFwE

## DataFrames & Datasets: Private Side

The figure below presents a high-level overview of the translation process.

- Whereas the low-level details of such translation are out of the scope of this lecture, we will enumerate the steps and present some examples of the optimisations taking place.



## Outline

1. Setting Up the Context.
2. Functional Spark: Spark Core.
3. Structured Spark: Spark SQL.
4. DataFrames & Datasets: Public Side.
5. DataFrames & Datasets: Private Side.
    a. Logical Query Plan.
    b. Optimised Logical Query Plan.
    c. Code Generation.

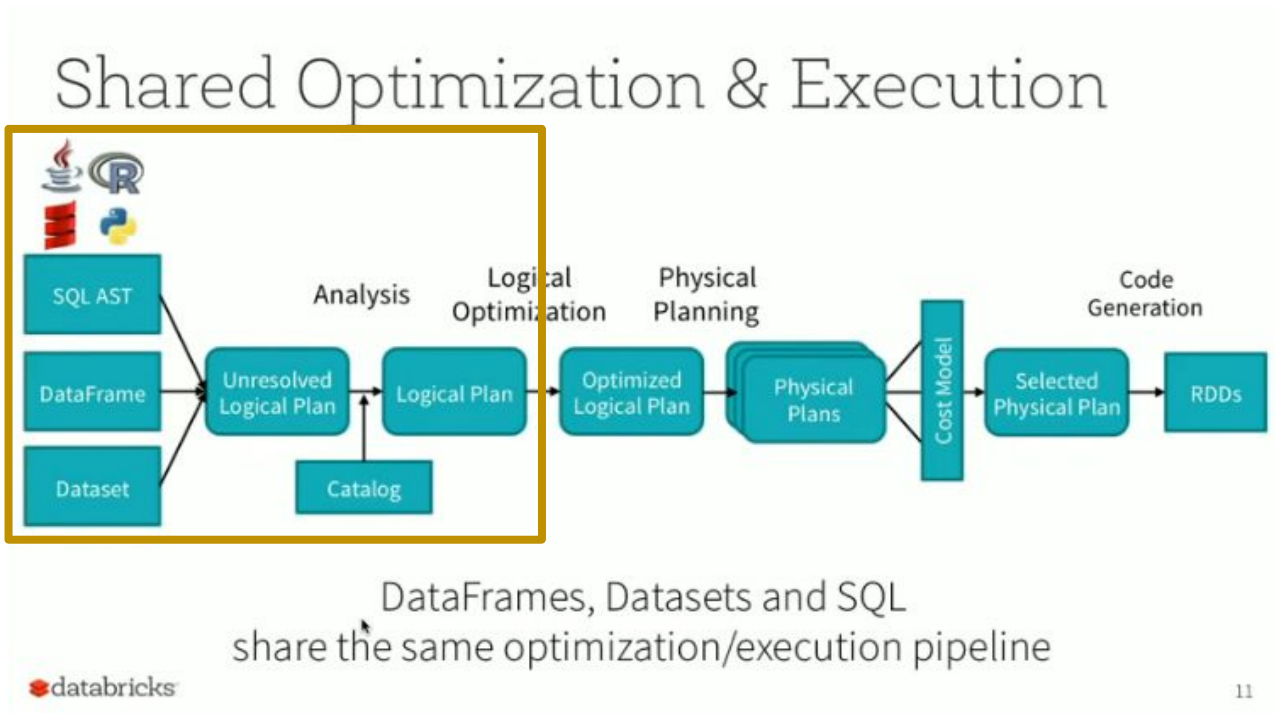
## Outline

1. Setting Up the Context.
2. Functional Spark: Spark Core.
3. Structured Spark: Spark SQL.
4. DataFrames & Datasets: Public Side.
5. DataFrames & Datasets: Private Side.
    a. Logical Query Plan.
    b. Optimised Logical Query Plan.
    c. Code Generation.

## Logical Query Plan

The figure below presents a high-level overview of the translation process.

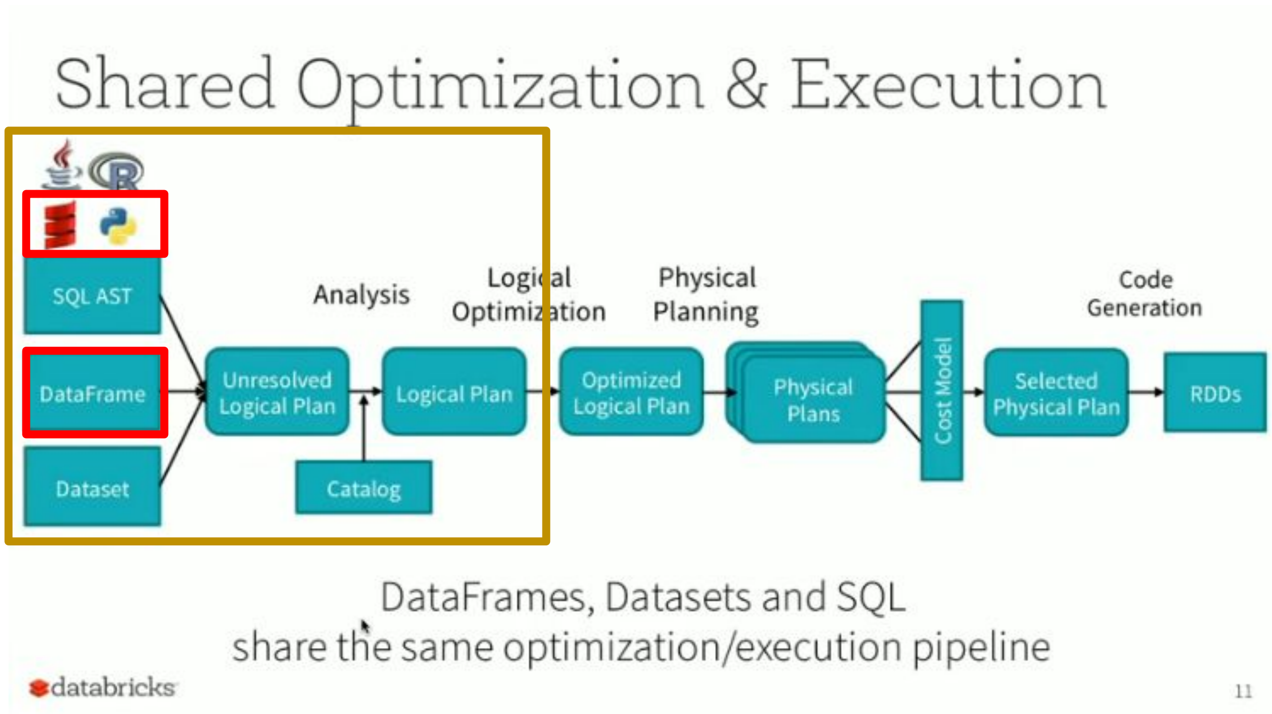
- Step 1:
  Spark SQL user program/application => Construction of a Logical Query Plan.



## Logical Query Plan

The figure below presents a high-level overview of the translation process.

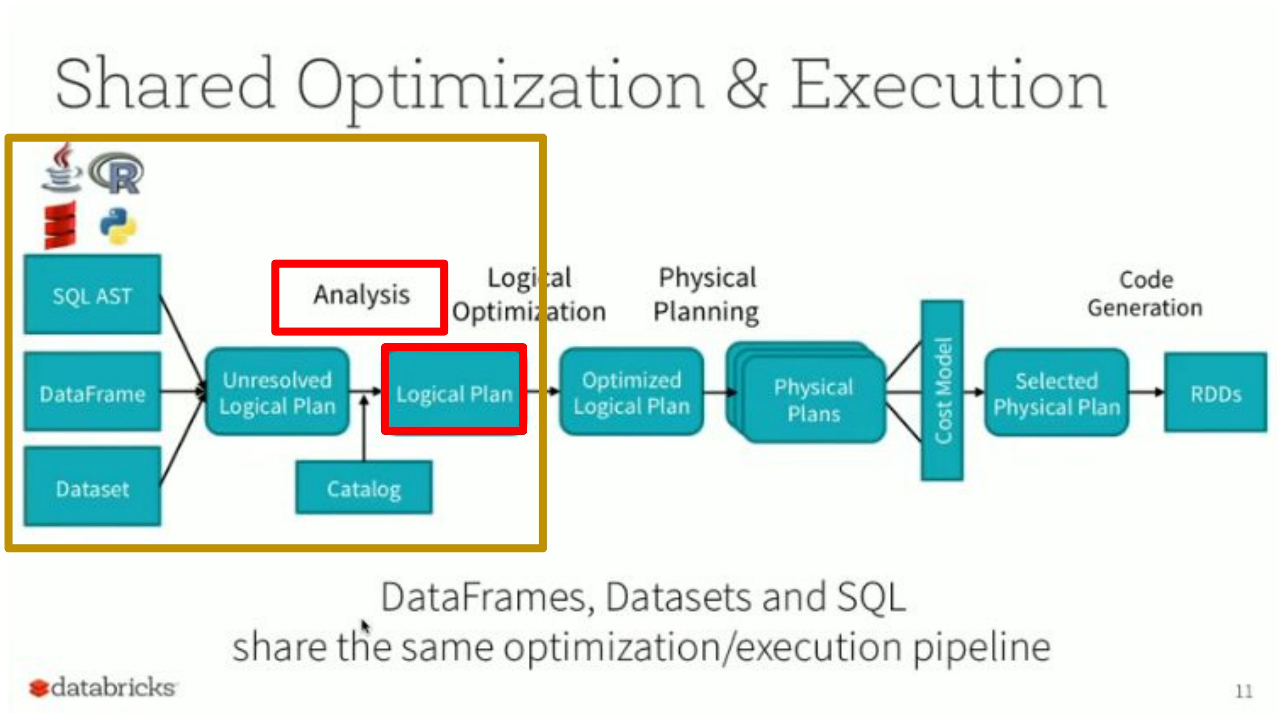
- Step 1:
  Spark SQL user program/application => Construction of a Logical Query Plan.



## Logical Query Plan

The figure below presents a high-level overview of the translation process.

- Step 1:
  Spark SQL user program/application => Construction of a Logical Query Plan.



## Logical Query Plan

Given an user Spark SQL application, the goal of the Logical Query Plan is to:

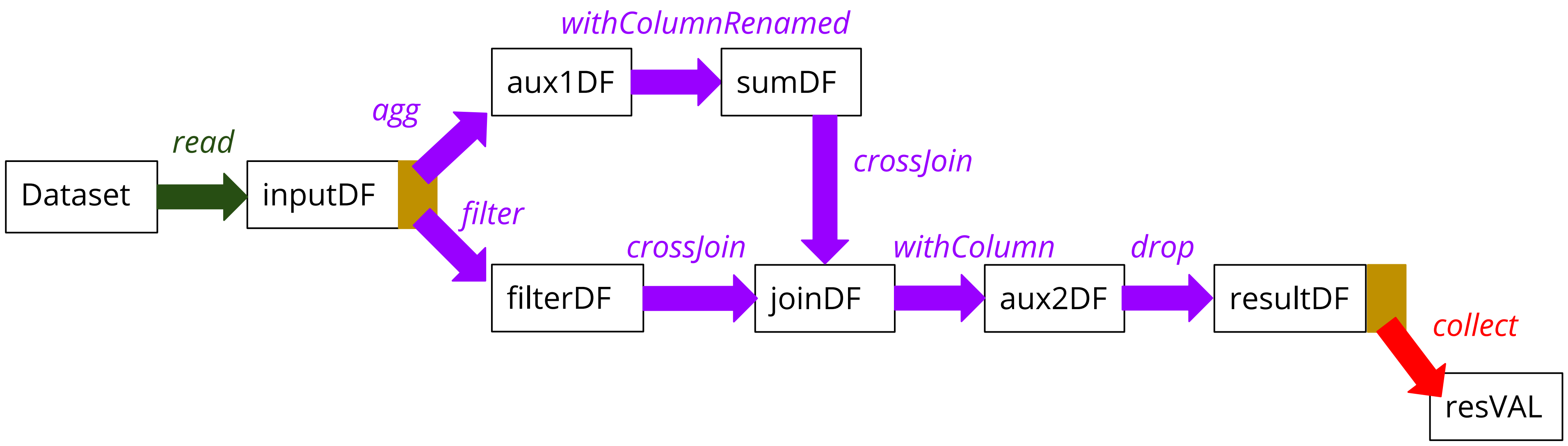
- Describe the structure of the computation required.

## Logical Query Plan

Given an user Spark SQL application, the goal of the Logical Query Plan is to:

- Describe the structure of the computation required.

Example: User Spark SQL application of p28_explain.py

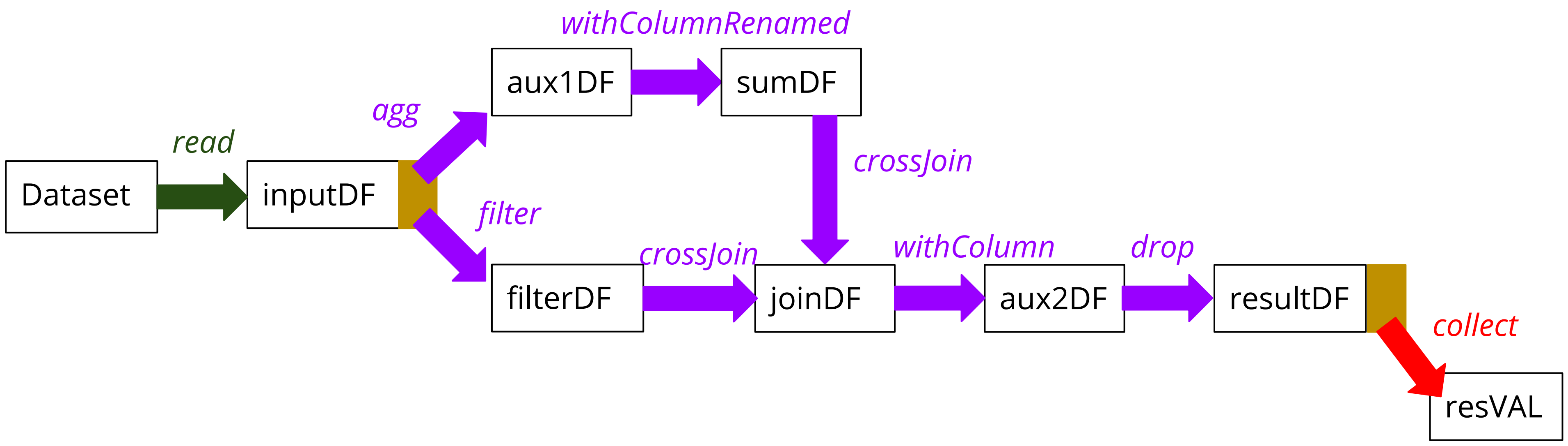

$$\begin{tikzcd}
 & & \text{aux1DF} \arrow[r, "withColumnRenamed"] & \text{sumDF} \arrow[d, "crossJoin"] & & & \\
\text{Dataset} \arrow[r, "read"] & \text{inputDF} \arrow[ur, "agg"] \arrow[dr, "filter"'] & & & & & \\
 & & \text{filterDF} \arrow[r, "crossJoin"] & \text{joinDF} \arrow[r, "withColumn"] & \text{aux2DF} \arrow[r, "drop"] & \text{resultDF} \arrow[dr, "collect"] & \\
 & & & & & & \text{resVAL}
\end{tikzcd}$$

## Logical Query Plan

Given an user Spark SQL application, the goal of the Logical Query Plan is to:

- Describe the structure of the computation required.

Example: User Spark SQL application of p28_explain.py



In this case the logical plan states that:

- The dataset is first to be read, then it is to be persisted, then aggregated and filtered in two ways, for the two resulting DF to be then joined again, modify the resulting one and finally collect it.

## Logical Query Plan

Given an user Spark SQL application, the goal of the Logical Query Plan is to:

- Describe the structure of the computation required.

**This Logical Query Plan looks pretty similar to the Direct Acyclic Graph of Spark Core that we saw before.**

## Logical Query Plan

Given an user Spark SQL application, the goal of the Logical Query Plan is to:

- Describe the structure of the computation required.

**This Logical Query Plan looks pretty similar to the Direct Acyclic Graph of Spark Core that we saw before.**

Moreover, the Logical Query Plan isolates the program written by the user (what language, what API) from the rest of the optimisation pipeline to be applied.

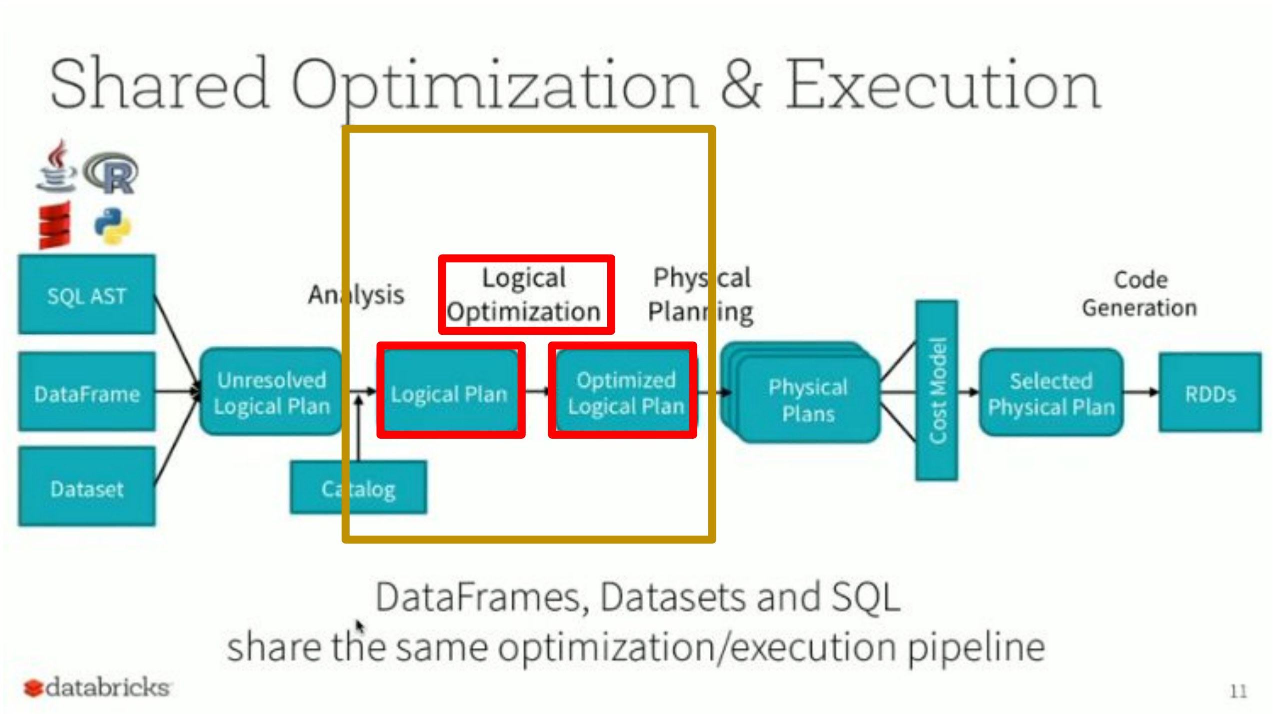
## Outline

1. Setting Up the Context.
2. Functional Spark: Spark Core.
3. Structured Spark: Spark SQL.
4. DataFrames & Datasets: Public Side.
5. DataFrames & Datasets: Private Side.
   a. Logical Query Plan.
   b. Optimised Logical Query Plan.
   c. Code Generation.

## Optimised Logical Query Plan

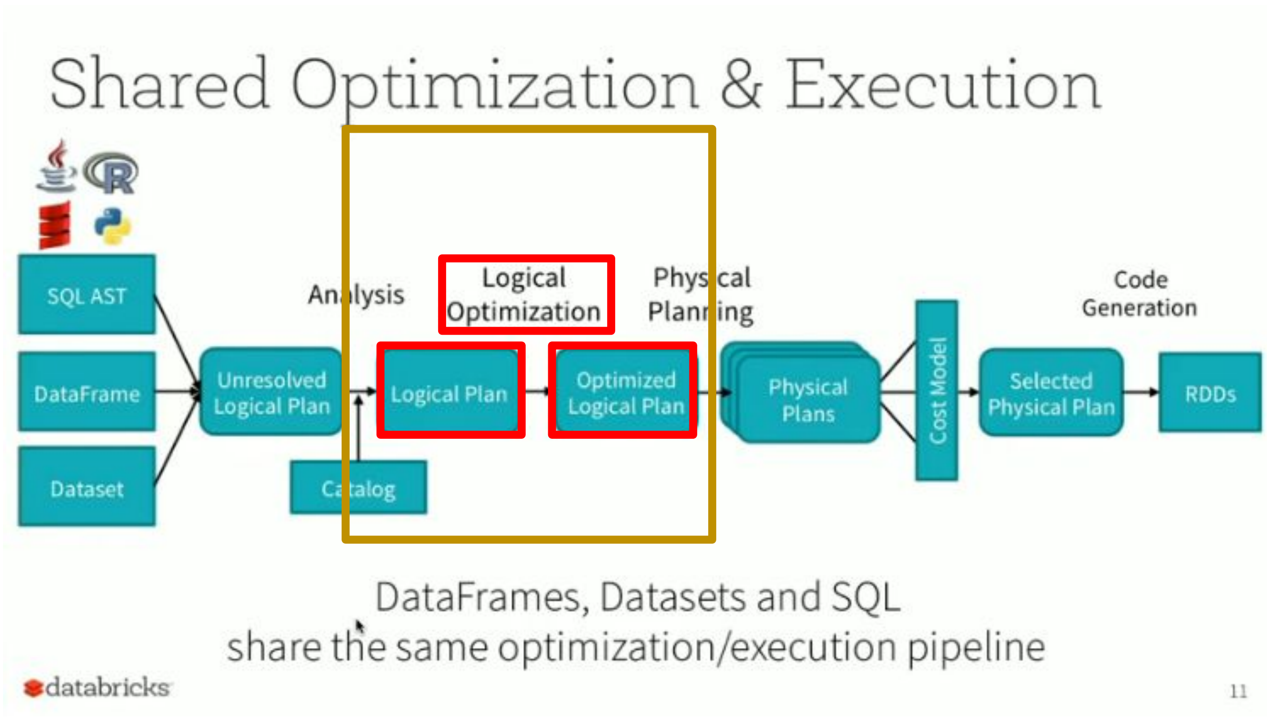
The figure below presents a high-level overview of the translation process.

- Step 2:
  The component Catalyst Optimiser is in charge of turning the Logical Query Plan into an Optimised Version of it.



## Optimised Logical Query Plan

Let's see some examples of such optimisations



## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

```
--DATASET FORMAT--

John;35
Mary;30
Harry;25
Janet;20
```

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK CORE ---

```
inputRDD = sc.textFile(my_dataset_dir)

formatRDD = inputRDD.map(              f1              )

   f1:: lambda line: tuple(line.replace("\n", "").split(";")) )

newRDD = formatRDD.filter( f2 )

   f2:: lambda x: x[1] == 30
```

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK CORE ---

```
inputRDD = sc.textFile(my_dataset_dir)

formatRDD = inputRDD.map(              f1              )

   f1:: lambda line: tuple(line.replace("\n", "").split(";")) )

newRDD = formatRDD.filter( f2 )

   f2:: lambda x: x[1] == 30
```

**In this case, Spark does not understand the semantics of f2!**

# Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK CORE ---

```
inputRDD = sc.textFile(my_dataset_dir)

formatRDD = inputRDD.map(              f1              )

   f1:: lambda line: tuple(line.replace("\n", "").split(";")) )

newRDD = formatRDD.filter( f2 )

   f2:: lambda x: x[1] == 30
```

**In this case, Spark does not understand the semantics of f2!**

Indeed, f2 can be any general function containing arbitrary code, as long as it goes from type T into Boolean.

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

---SPARK SQL---

```
my_schema = StructType(
 [StructField("name", StringType(), True),
  StructField("age", IntegerType(), True) ])

inputDF = spark.read.format("csv") \
          .option("delimiter", ";") \
          .option("quote", "") \
          .option("header", "false") \
          .schema(my_schema) \
          .load(my_dataset_dir)

newDF = inputDF.filter( inputDF["age"] == 30 )
```

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

---SPARK SQL---

```
my_schema = StructType(
 [StructField("name", StringType(), True),
  StructField("age", IntegerType(), True) ])

inputDF = spark.read.format("csv") \
          .option("delimiter", ";") \
          .option("quote", "") \
          .option("header", "false") \
          .schema(my_schema) \
          .load(my_dataset_dir)

newDF = inputDF.filter( inputDF["age"] == 30 )
```

**In this case, Spark can understand/reason with the expression provided.**

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

**In this case, Spark can understand/reason with the expression provided.**

- inputDF has a concrete schema, known by Spark.

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

**In this case, Spark can understand/reason with the expression provided.**

- inputDF has a concrete schema, known by Spark.
- The expression `inputDF["age"] == 30` inside the DSL operator filter is captured as an abstract syntax tree which can be further analysed.

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

**In this case, Spark can understand/reason with the expression provided.**

- inputDF has a concrete schema, known by Spark.
- The expression `inputDF["age"] == 30` inside the DSL operator filter is captured as an abstract syntax tree which can be further analysed.
- Spark understands that column "age" is of type integer and it infers that it is being compared to the literal 30.

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

**In this case, Spark can understand/reason with the expression provided.**

- inputDF has a concrete schema, known by Spark.
- The expression `inputDF["age"] == 30` inside the DSL operator filter is captured as an abstract syntax tree which can be further analysed.
- Spark understands that column "age" is of type integer and it infers that it is being compared to the literal 30.
- So the Spark analyser creates an object `EqualTo(age, Lit(30))`, which provides much more information to reason and optimise on it.

## Optimised Logical Query Plan

**That’s why we try to avoid the use of User Defined Functions in Spark SQL!**

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- DATASET 1 FORMAT ---

John;35
Mary;30
Harry;25
Janet;20

--- DATASET 2 FORMAT ---

Vincent;45
Ruth;30
Gearoid;25
Sarah;40

--- SPARK SQL ---

```
my_schema = StructType( [StructField("name", StringType(), True),
                         StructField("age", IntegerType(), True) ])

input1DF = spark.read.format("csv").schema(my_schema).load(my_dataset_dir_1)
input2DF = spark.read.format("csv").schema(my_schema).load(my_dataset_dir_2)
```

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK SQL VERSION 1 ---

```
my_compareUDF = udf(lambda x, y: x == y, BooleanType())

newDF = input1DF.join(input2DF,
                      my_compareUDF(input1DF["age"]), input2DF["age"]))
```

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK SQL VERSION 1 ---

```
my_compareUDF = udf(lambda x, y: x == y, BooleanType())

newDF = input1DF.join(input2DF,
                                 my_compareUDF(input1DF["age"]), input2DF["age"]))
```

By using an UDF Spark can not understand its internals!

- Again, within this lambda function the user could have written whatever expression, as long as it returns a Boolean value.

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK SQL VERSION 1 ---

```
my_compareUDF = udf(lambda x, y: x == y, BooleanType())

newDF = input1DF.join(input2DF,
                     my_compareUDF(input1DF["age"]), input2DF["age"]))
```

By using an UDF Spark can not understand its internals!

- So for Spark to return the tuples matching the predicate it has to try every single possible combination of rows in input1DF and input2DF (i.e., their cartesian product) and pass it to the function.

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK SQL VERSION 2 ---

```
newDF = input1DF.join( input2DF, input1DF["age"] == input2DF["age"]) )
```

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK SQL VERSION 2 ---

```
newDF = input1DF.join( input2DF, input1DF["age"] == input2DF["age"]) )
```

**In this case, Spark can understand/reason with the expression provided.**

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK SQL VERSION 2 ---

```
newDF = input1DF.join( input2DF, input1DF["age"] == input2DF["age"]) )
```

**In this case, Spark can understand/reason with the expression provided.**

- Spark can create again an object EqualTo(age, age) for which further optimisations can be applied.

## Optimised Logical Query Plan

**Example 1:** Semantic analysis of expressions.

--- SPARK SQL VERSION 2 ---

```
newDF = input1DF.join( input2DF, input1DF["age"] == input2DF["age"]) )
```

**In this case, Spark can understand/reason with the expression provided.**

- Spark can create again an object EqualTo(age, age) for which further optimisations can be applied.
- In particular, the two DataFrames can be initially hash partitioned by their "age" column values, dramatically reducing the further data shuffle when doing the join.

## Optimised Logical Query Plan

**Example 2:** Predicate Pushdown.

## Optimised Logical Query Plan

**Example 2:** Predicate Pushdown.

- The basic idea of predicate pushdown is that certain parts of the Logical Query Plan can be “pushed” to the Dataset.

## Optimised Logical Query Plan

**Example 2:** Predicate Pushdown.

- The basic idea of predicate pushdown is that certain parts of the Logical Query Plan can be “pushed” to the Dataset.

- This can drastically reduce the processing time by filtering out data before it is transferred over the network and load into memory.
  Example: User Spark SQL application of p29_explain.py

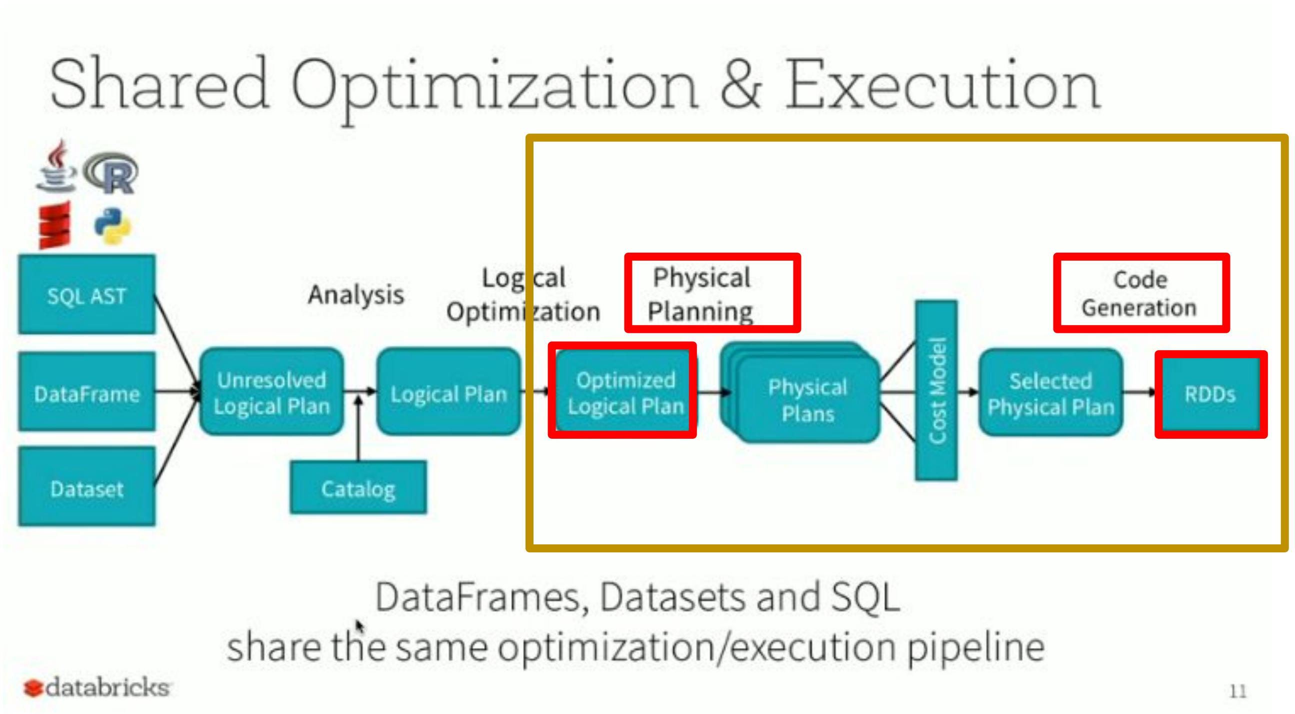
## Outline

1. Setting Up the Context.
2. Functional Spark: Spark Core.
3. Structured Spark: Spark SQL.
4. DataFrames & Datasets: Public Side.
5. DataFrames & Datasets: Private Side.
    a. Logical Query Plan.
    b. Optimised Logical Query Plan.
    c. Code Generation.

## Code Generation

The figure below presents a high-level overview of the translation process.

- Step 3:
  Optimised Logical Query Plan => Equivalent RDD-based program.



## Code Generation

- The structure of the data (and its mapping to the Spark SQL type system) can be exploited to achieve a more efficient representation of it.

## Code Generation

- The structure of the data (and its mapping to the Spark SQL type system) can be exploited to achieve a more efficient representation of it.

- Likewise, we can then exploit this more efficient data representation to customise the operations (transformations and actions of the Optimal Logical Query Plan) applied to it.

## Code Generation

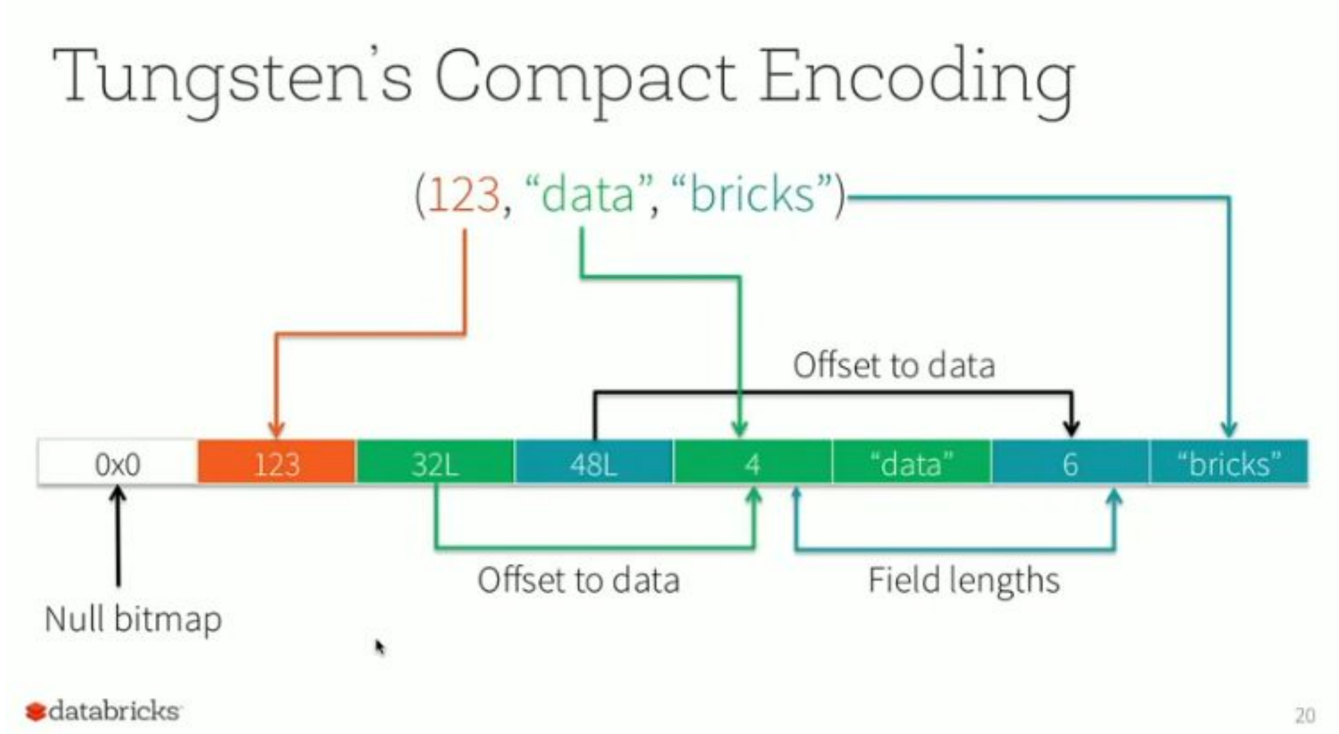
- The structure of the data (and its mapping to the Spark SQL type system) can be exploited to achieve a more efficient representation of it.

- Likewise, we can then exploit this more efficient data representation to customise the operations (transformations and actions of the Optimal Logical Query Plan) applied to it.

- In translating the Spark SQL program (whether based in DataFrames and/or Datasets) into an equivalent Spark Core RDD-based one, Spark SQL uses its own customised Tungsten encoding, again exploiting the fact that the DataFrame/Dataset items have a well-known structure.

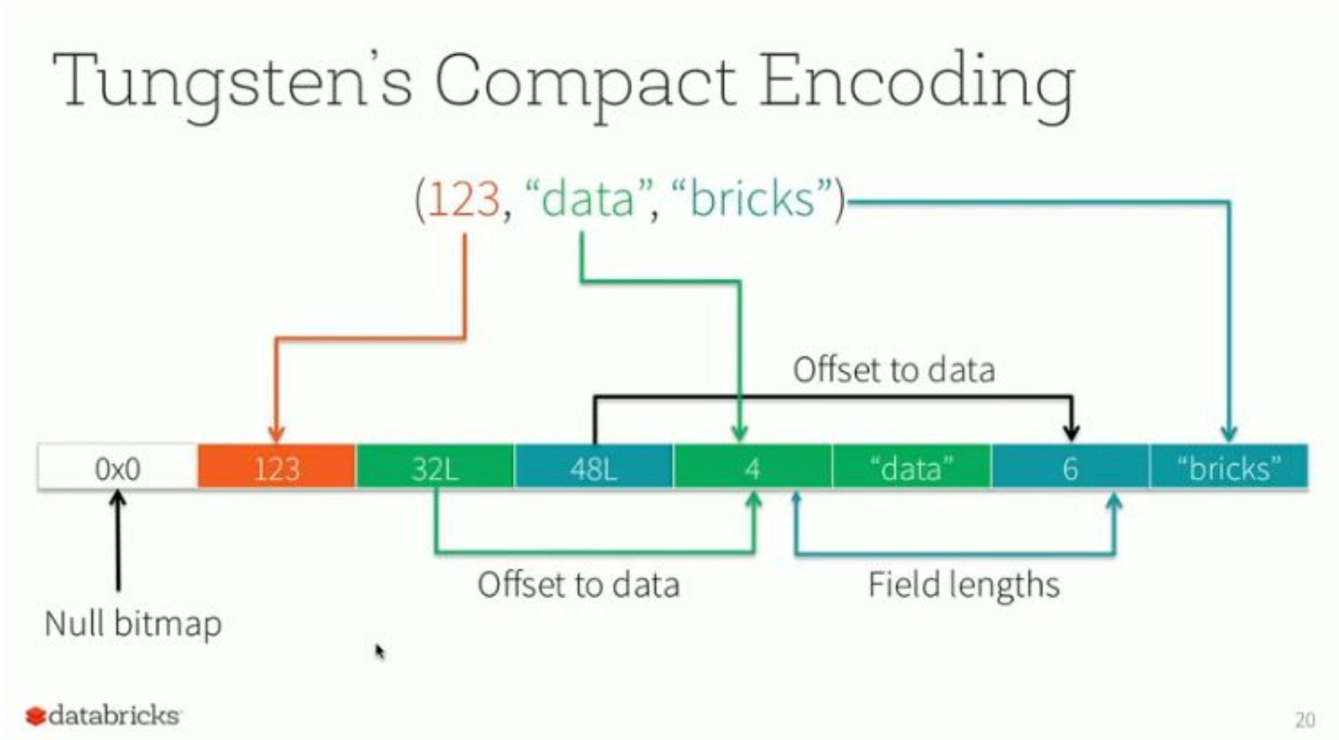
## Code Generation

- The Figure below represents the Tungsten encoding of an item from a DataFrame into an equivalent Binary Large Object (BLOB) item of an RDD.



## Code Generation

- The new representation of the item in the RDD as a BLOB occupies much less space than the one needed it the data had been loaded as an RDD of String (plain text), for its further parsing.



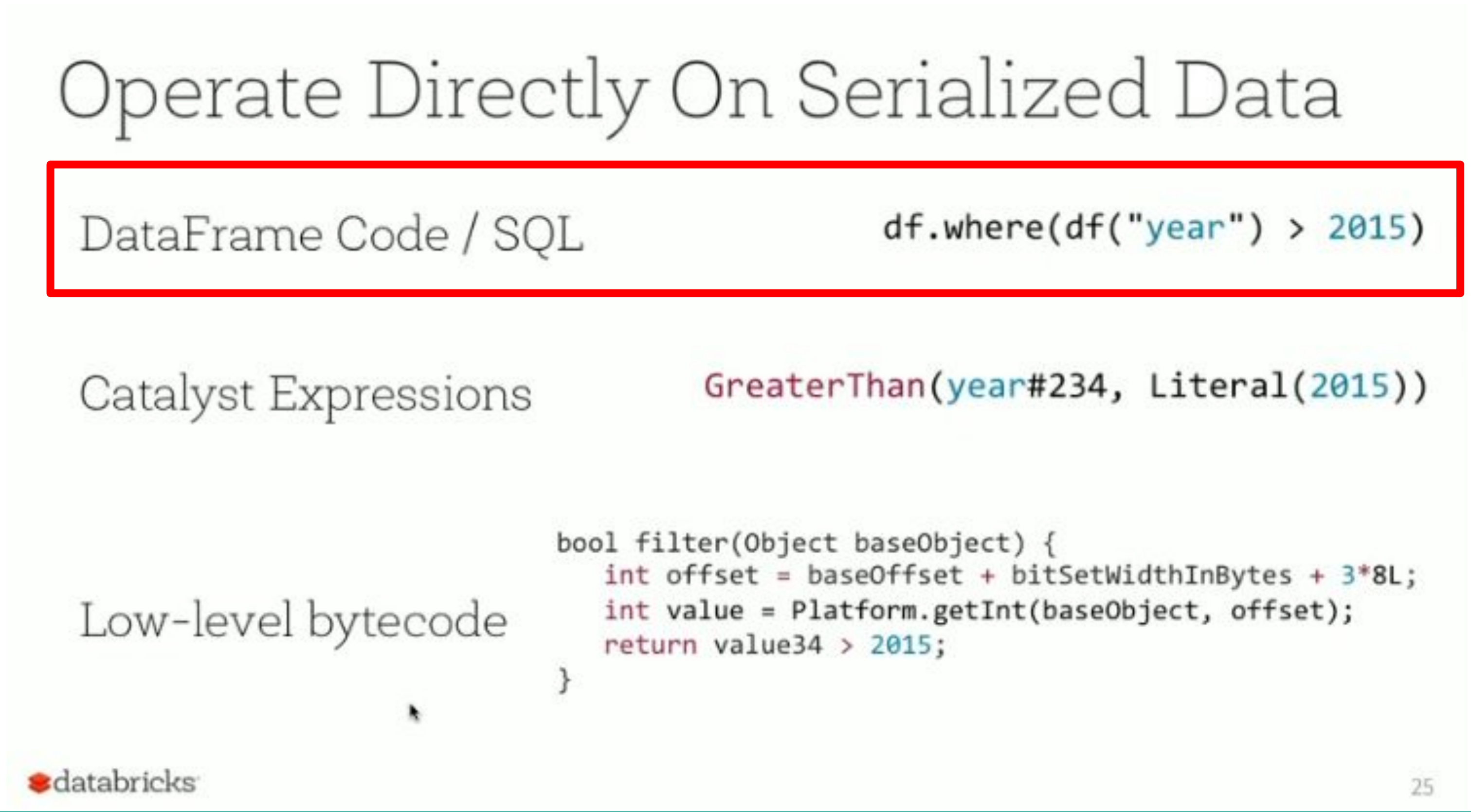
## Code Generation

Moreover,
by knowing the structure of these BLOB items,
specific bytecode can be generated
to customise the operations
to this BLOB encoding.

## Code Generation

**Example.**

If the Spark SQL program had a DataFrame filter operation in the field "year"...



Operate Directly On Serialized Data

DataFrame Code / SQL

```
df.where(df("year") > 2015)
```

Catalyst Expressions

```
GreaterThan(year#234, Literal(2015))
```

Low-level bytecode

```
bool filter(Object baseObject) {
   int offset = baseOffset + bitSetWidthInBytes + 3*8L;
   int value = Platform.getInt(baseObject, offset);
   return value34 > 2015;
}
```

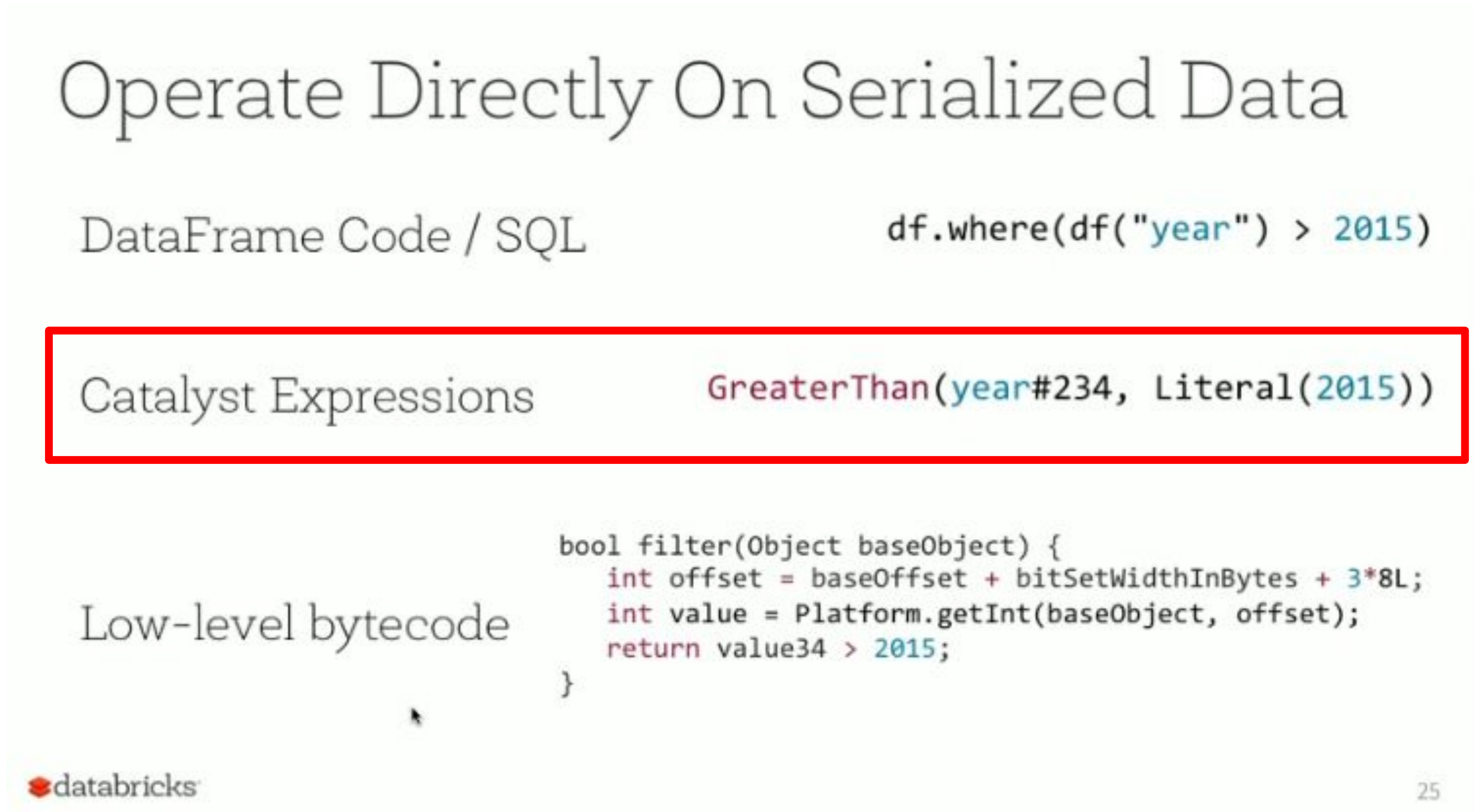
databricks

25

## Code Generation

**Example.**

...Catalyst will analyse it and generate more information about it...



Operate Directly On Serialized Data

DataFrame Code / SQL

```
df.where(df("year") > 2015)
```

Catalyst Expressions

```
GreaterThan(year#234, Literal(2015))
```

Low-level bytecode

```
bool filter(Object baseObject) {
   int offset = baseOffset + bitSetWidthInBytes + 3*8L;
   int value = Platform.getInt(baseObject, offset);
   return value34 > 2015;
}
```

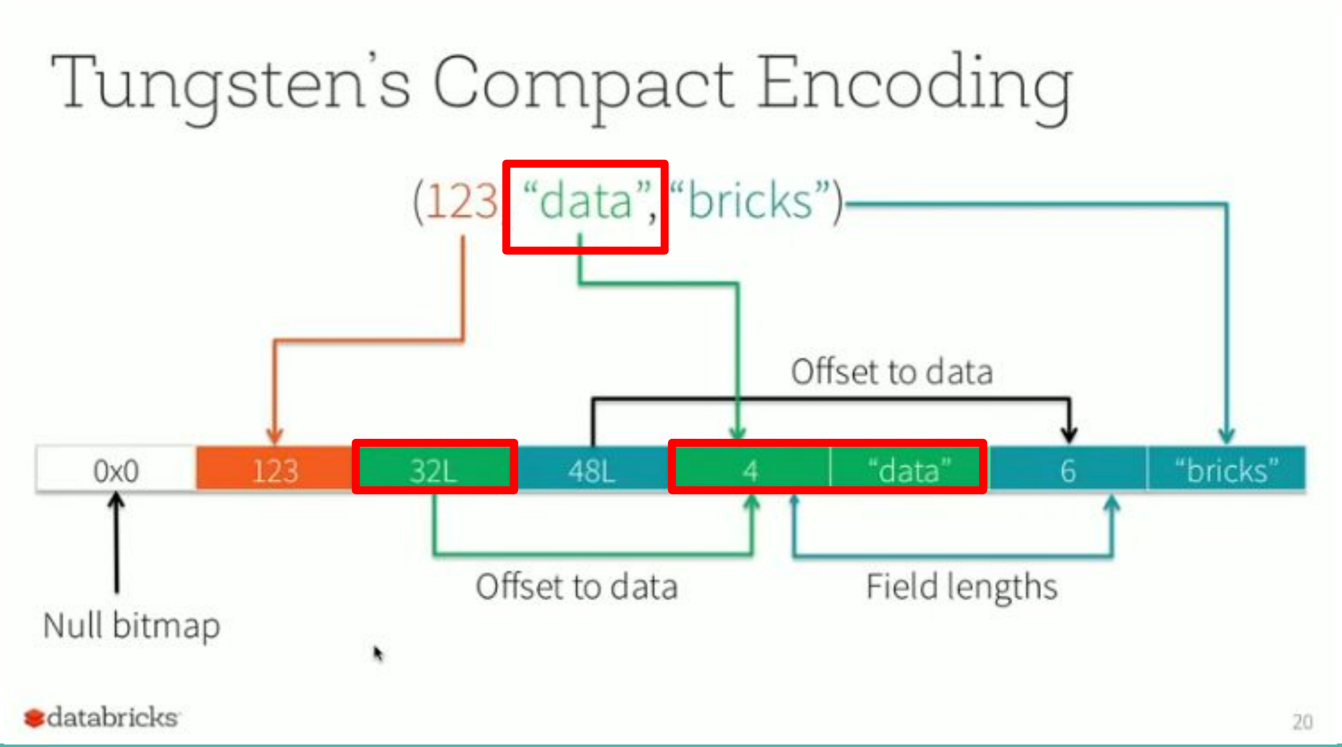
databricks

25

## Code Generation

**Example.**

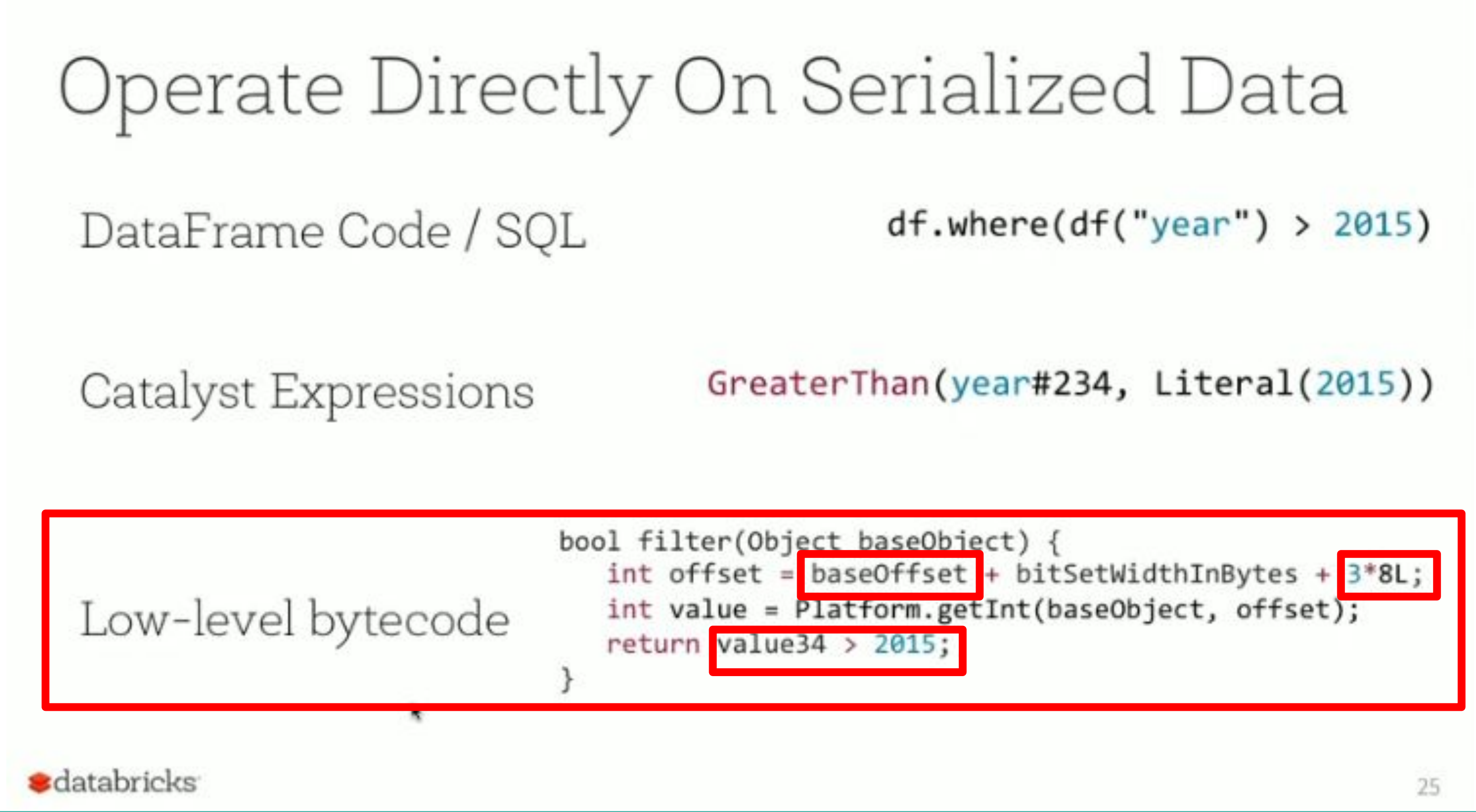
... if Tugsten encodes this DataFrame into an RDD of BLOBs, where the location of the field "year" is known to be at positions [Lb, Ub]...



## Code Generation

**Example.**

...then the translation from Spark SQL to Spark Core can generate bytecode for the RDD "filter" operation to look at positions [Lb, Ub] of the RDD items.



Operate Directly On Serialized Data

DataFrame Code / SQL

```
df.where(df("year") > 2015)
```

Catalyst Expressions

```
GreaterThan(year#234, Literal(2015))
```

Low-level bytecode

```
bool filter(Object baseObject) {
   int offset = baseOffset + bitSetWidthInBytes + 3*8L;
   int value = Platform.getInt(baseObject, offset);
   return value34 > 2015;
}
```

databricks
25

## Outline

## Outline

1. Setting Up the Context.
2. Functional Spark: Spark Core.
3. Structured Spark: Spark SQL.
4. DataFrames & Datasets: Public Side.
5. DataFrames & Datasets: Private Side.

Thank you for your attention!