Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Dataframe Internals

Lecture:
Evolution of Spark APIs

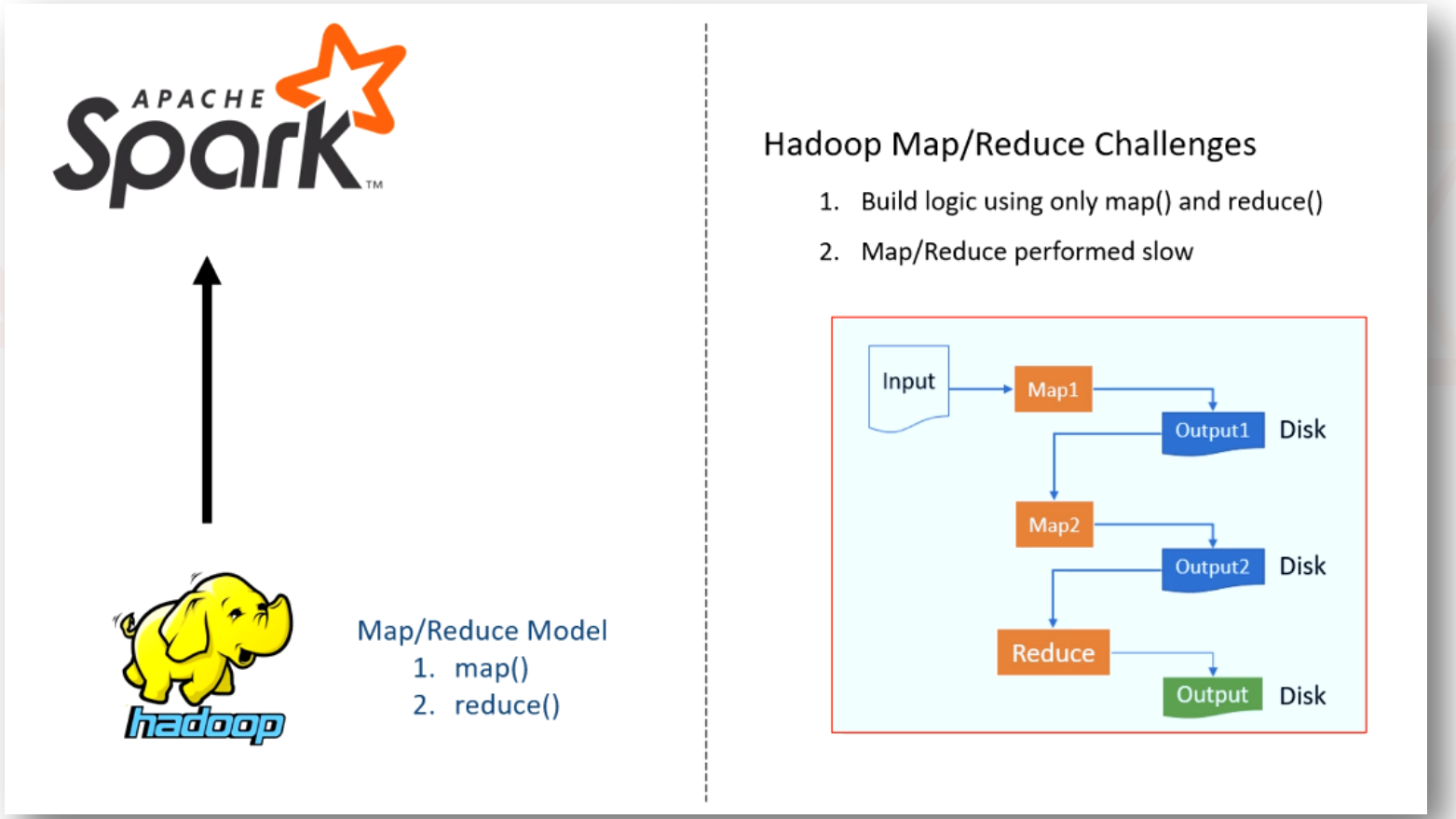
Evolution of Spark APIs

Let us go back to the Spark evolution and understand how Spark evolved over the time since its inception.

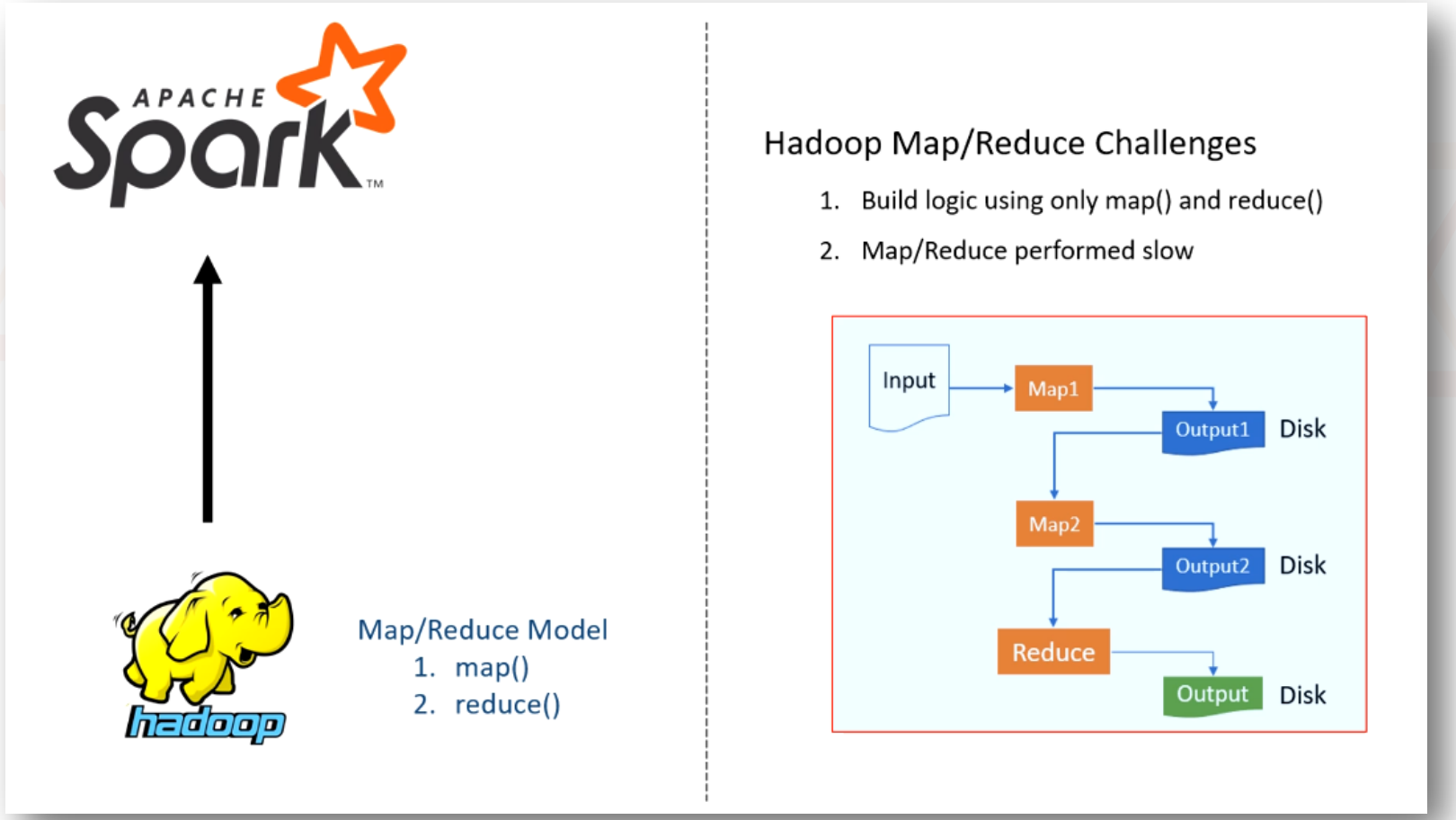
Apache Spark started to simplify and improve the Hadoop Map/Reduce programming model. Map/Reduce approach allowed us to write the application logic using two functions:

1. Map function
2. Reduce function

You could write a chain of the map/reduce functions as highlighted in the image below. But as you can see, almost everything was limited to map() and reduce() functions. And that was a problem.
The Map/Reduce framework was slow, which was another big problem.
But why was it slow? Because Map/Reduce was a two-step approach. The first step was to run the map() function and generate some intermediate output. You are forced to store this intermediate output in the storage, so we use the output as input for the next step in the process. And that's what this diagram shows.



In the highlighted diagram, we read input data from a file and apply the logic using the map-1 function. We store the intermediate result of map-1 in the filesystem. The output-1 becomes the input for the next step in the logic, and we generate the output-2. I am showing a small chain of two maps and one reduce. However, the real-life implementation was done using long chains of the map and reduced functions. Map/reduce applications lost a lot of time writing output-1, output-2, and then reading these from the storage.

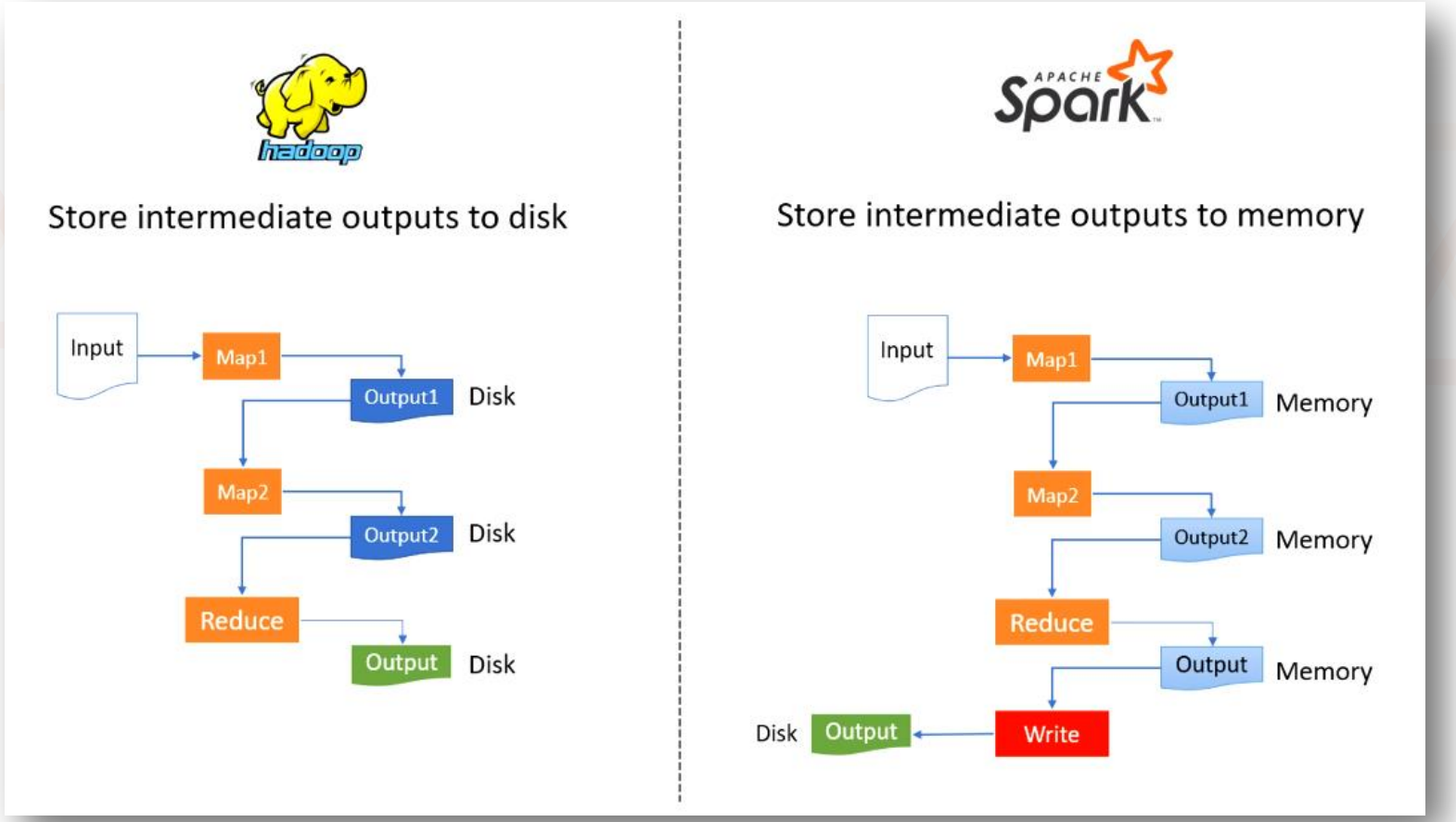


So we had two significant problems with Hadoop Map/Reduce approach, Spark initially started with an approach to solve these two problems.

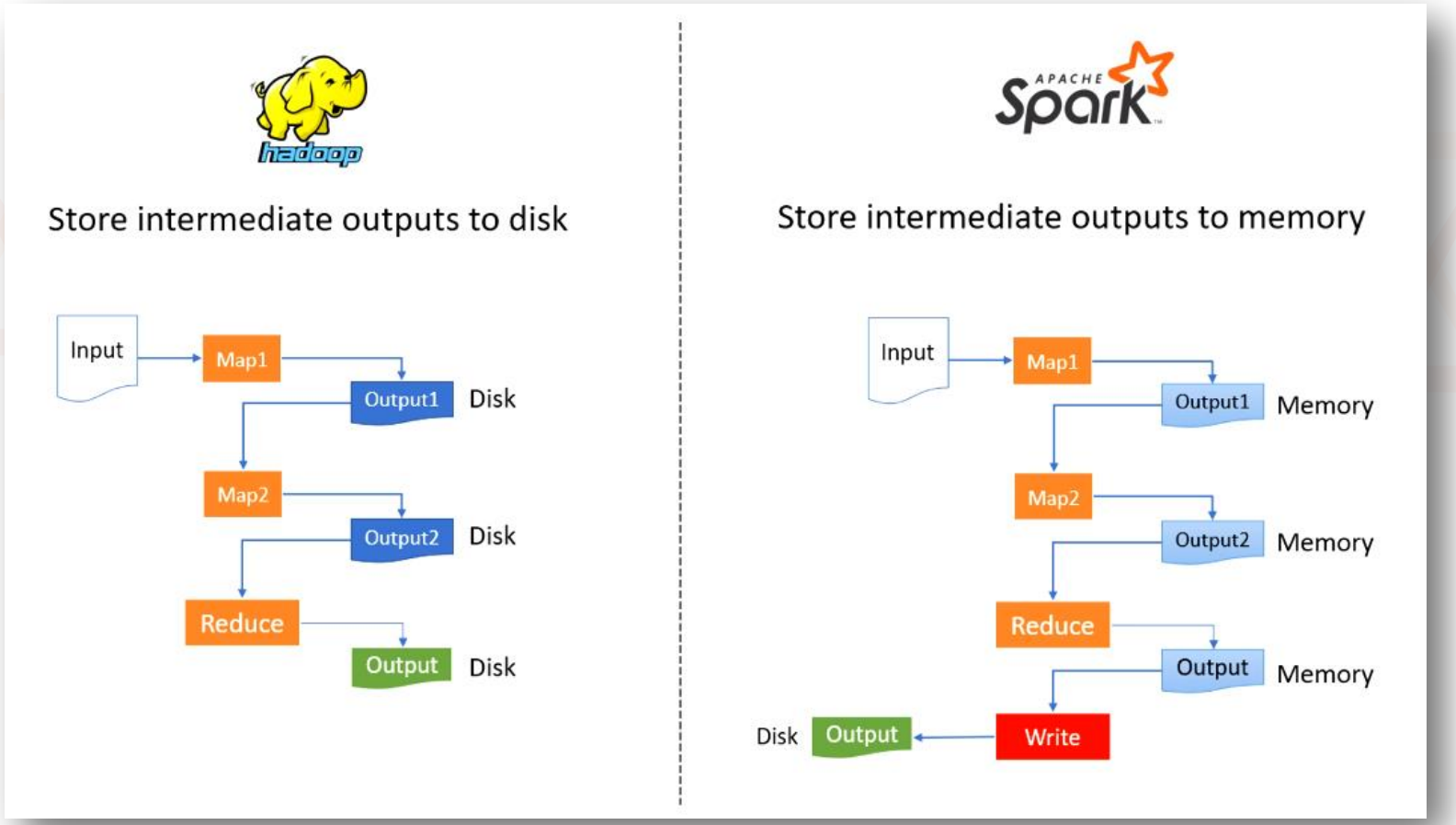
1. Eliminate the need for storing intermediate results on the disk. We will significantly improve the Map/Reduce performance if we can do that.
2. Add more programming facilities beyond the map() and reduce() functions.

These two were the initial goals of the Spark project. Spark went too far from the Hadoop Map/Reduce, and now it has become a defacto of distributed data processing. However, Spark started as a research project for some students, and these two were the initial goals for the Spark project.

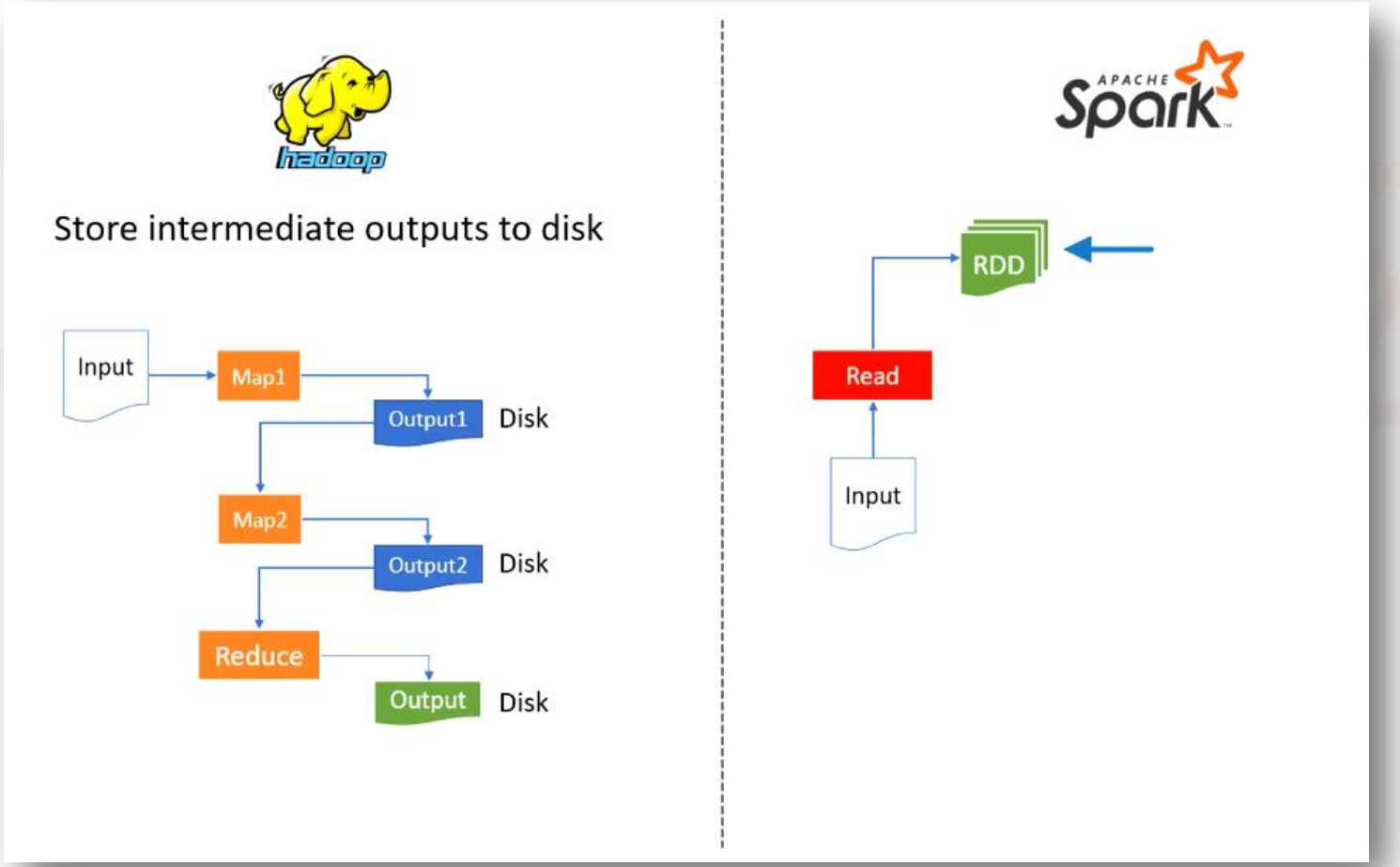
Now let's talk about the solution to the previous two problems. The first problem was simple. Instead of storing intermediate results on the disk, Spark decided to buffer the results in the memory. And finally, when you want to write the final results, you can use the write method. But until you write, the data remains in the memory. And that is how Spark is being termed as an in-memory computing engine. It keeps the intermediate results in the memory as long as enough memory is available. The result goes to disk when you do not have enough memory. But if you have the memory, the data remains there.



But what about fault tolerance. What if something failed? How do we get the intermediate results because the memory is volatile?
You would lose the results if the system failed. They said we would recompute the results. The failure happens once in a while. So why do we slow down the process every time, even if there is no failure. And that is a valid argument, Isn't it? So, Spark offered performance improvements over Map/Reduce, and they achieved it using memory instead of disk.



Now let's talk about the second improvement. The Map/Reduce offered only two functions: Map and Reduce. Spark wanted to offer a lot many other functions. And they came up with a super simple idea. Here is the diagram on right side that shows the idea. Spark offered a read method to read input data and create an object called RDD. The RDD was a data structure similar to the List in other programming languages. And that was revolutionary.



Let's assume you want to read a CSV file and process it. The Map/Reduce approach allows you to write the map() function, which reads the input file directly. So in the Map/Reduce approach, your data comes as an input to the map() function. And you write the logic to process the data in the map function. Then you write the result to the output on the disk to re-read it in the next step.

But Spark came up with a different approach. They said we give you a method to read the data from the file. You can read the CSV file, and we will give you a list of lines in an RDD object.
So a programmer can read a file, and they will get a dataList called RDD. Then they can apply the map() function on the RDD. They can also apply many other functions similar to the map() function. For example, Spark allowed count(), filter(), foreach(), groupBy(), aggregate(), union() and many more.

And they changed the way we program on the Hadoop platform. They turned the Map/Reduce upside down and changed the programming approach.

And all this magic happened because they came up with the idea of offering a dataList. They called it RDD - Resilient Distributed Dataset.
The RDD was not a new concept. It was similar to the List data structure in any programming language.

For example, you can look at the Python List data structure. Let me show you Python documentation. **(Reference: https://docs.python.org/3/tutorial/datastructures.html)**

To create a list, you will read some values in the List.

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```


Then you can apply list methods to process those values. You can use the count(), reverse(), append(), sort() and many more.

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```


A Python list is a data structure that comes with two things:

1. It stores some data as a list.
2. It allows you to process the data using some methods.

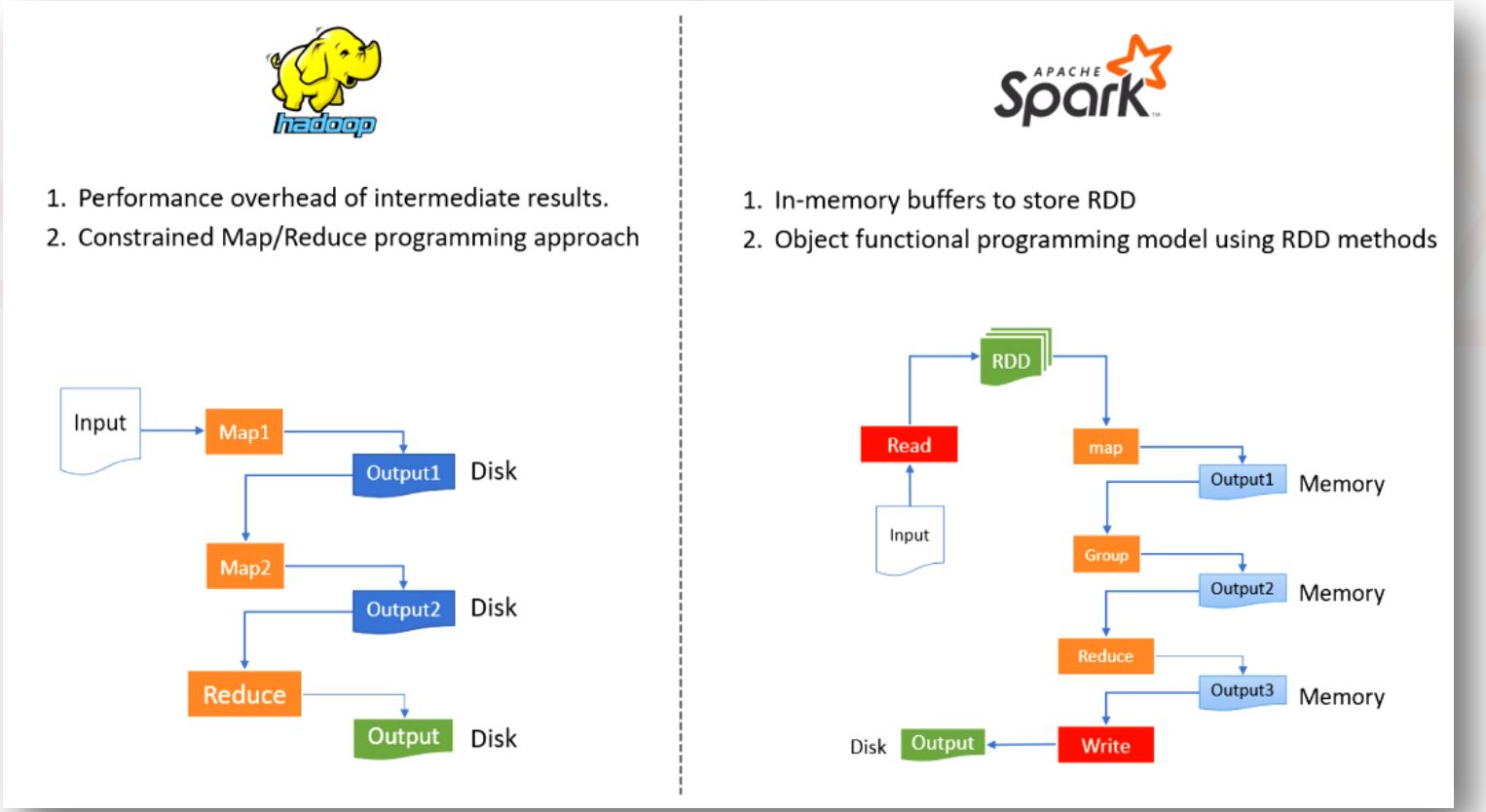
The RDD is the same. A Spark RDD is a data structure that comes with two things:

1. It stores some data similar to the List.
2. It allows you to process the data using some methods.

This approach was programmer-friendly. If you know any programming language, you already know how to work with a List like Data structure. So learning Spark was super easy compared to the black magic of Map/Reduce.

Spark brought the idea of RDD programming and revolutionized distributed computing. They almost killed the Hadoop Map/Reduce, and now you won't find anyone writing Map/Reduce applications.

Spark started as an improvement project of Hadoop Map/Reduce. They initially targeted to solve two problems. The performance overhead of intermediate results. And the constrained Map/Reduce programming approach. They came up with the RDD implementation that fixed both the problems. The RDD APIs are used by the programmers to process data. Using RDD was as simple as using a List in any programming language. But RDD was not a simple List. It was a distributed list. Internally, RDD was processing large volumes of distributed data on the Hadoop platform. But programmers were writing code like they were processing data using a simple list object.

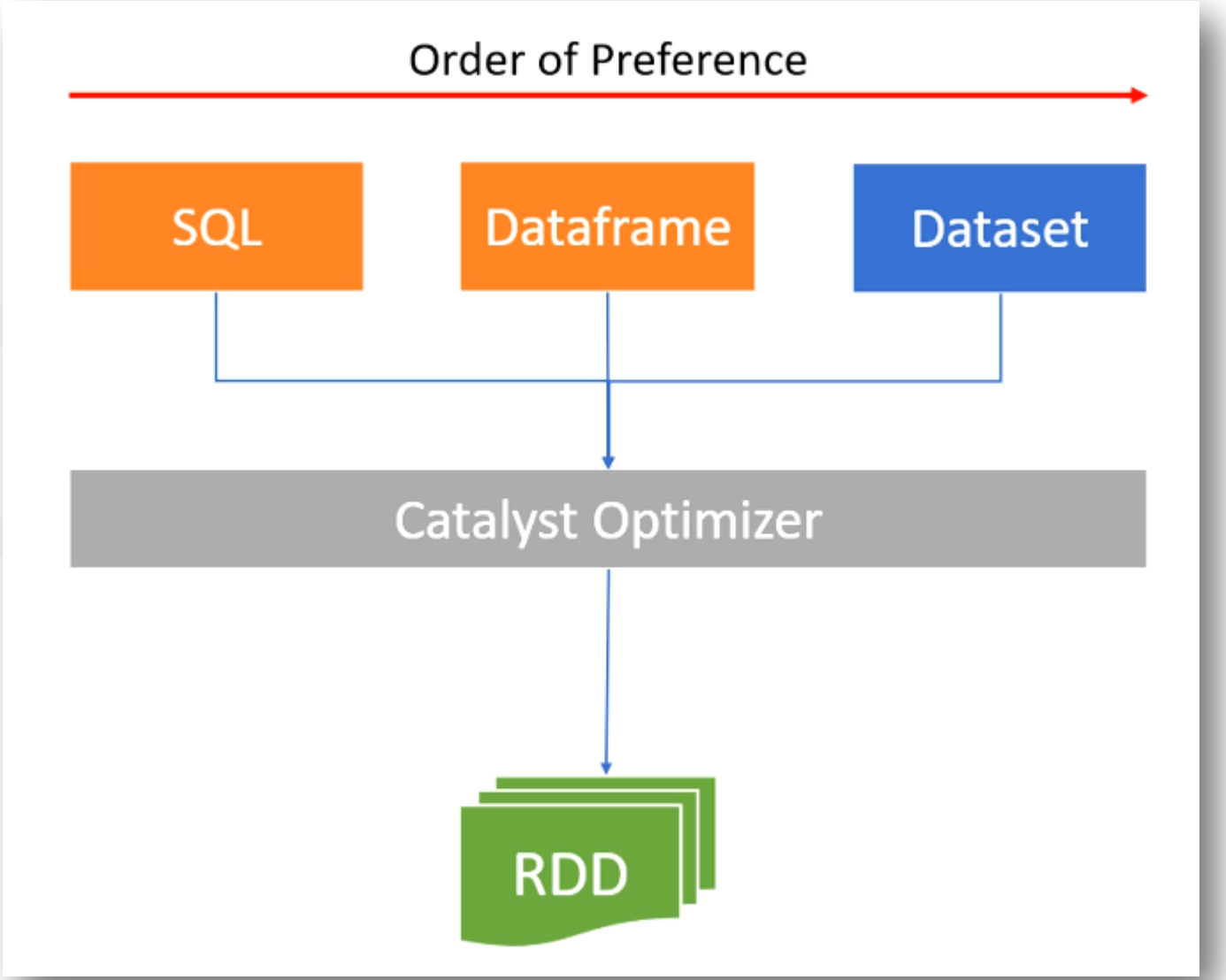


Now let's move forward to learn what happened next.
Spark didn't stop at the RDD and came up with improvements such as Dataset and Dataframe. These two are similar to RDD but are even easier to use and more advanced than the RDD. The RDD represents raw data.
The Dataframe and the Dataset were introduced to include a schema with the raw data.

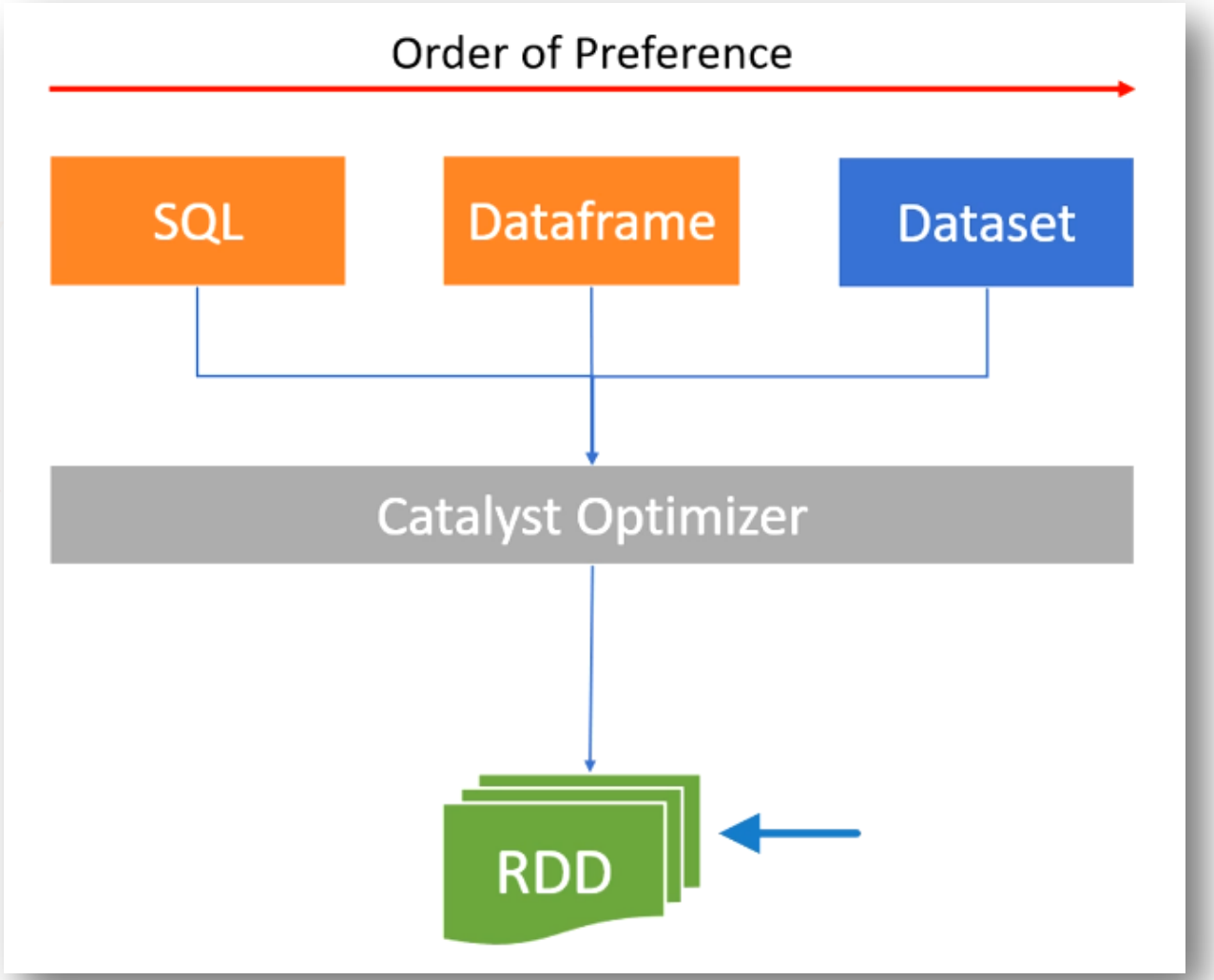
So the RDD represents only one thing: the raw data.
And the Dataframe and Dataset implement two things: Raw data and a Schema for the raw data.

Spark also came up with the SQL layer and a catalyst optimizer to simplify our life.

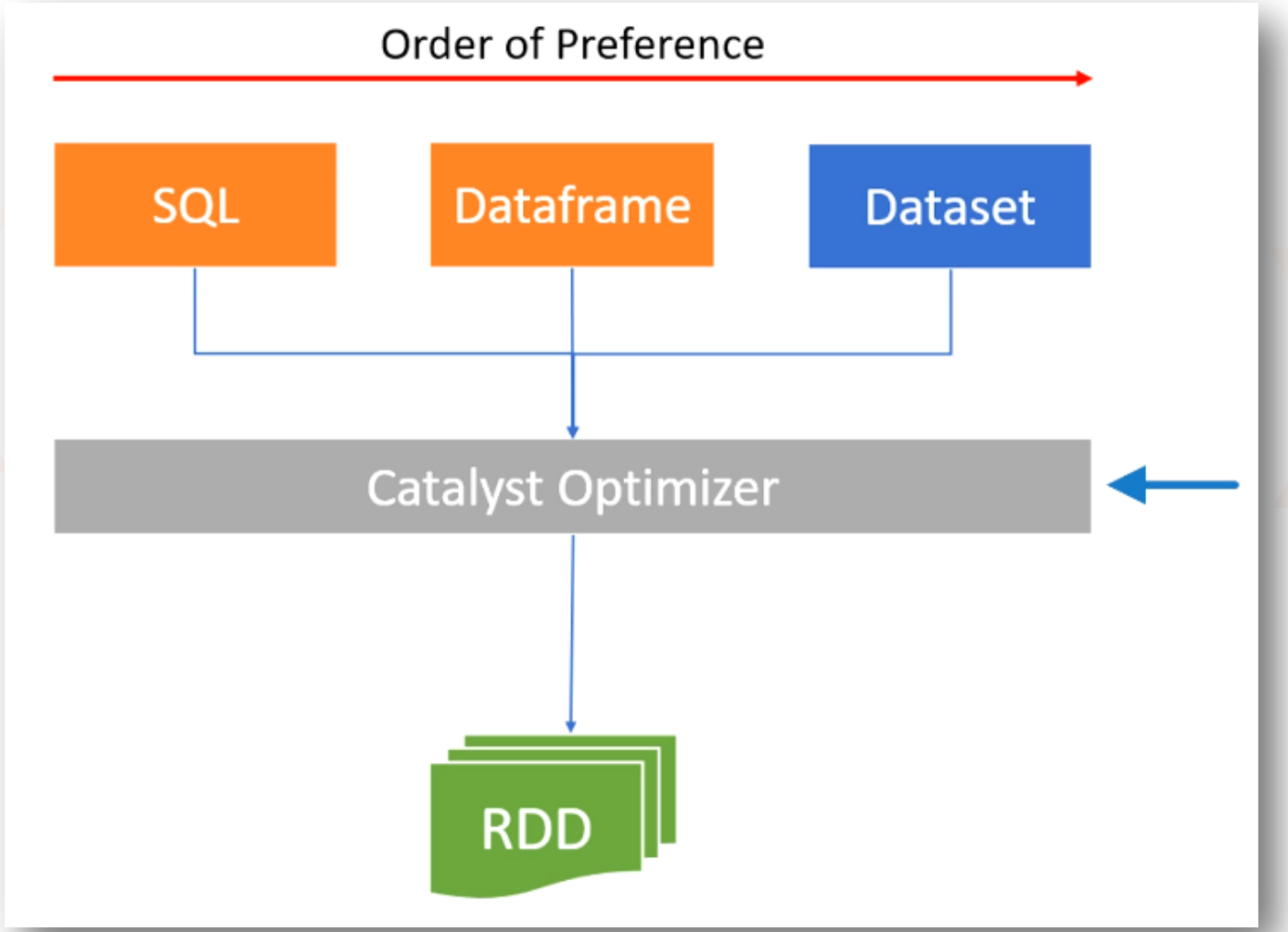
So, the current Spark data processing tools can be visually arranged, as shown in the following figure.



The RDD is at the core. You can use RDD APIs to develop your applications. However, it is the most basic and old tool. Learning and using RDD is difficult compared to SQL and Dataframe. The RDD also lacks all the optimization brought to you by the catalyst optimizer. The Spark community is not recommending using RDD. They now recommend using the top layer.



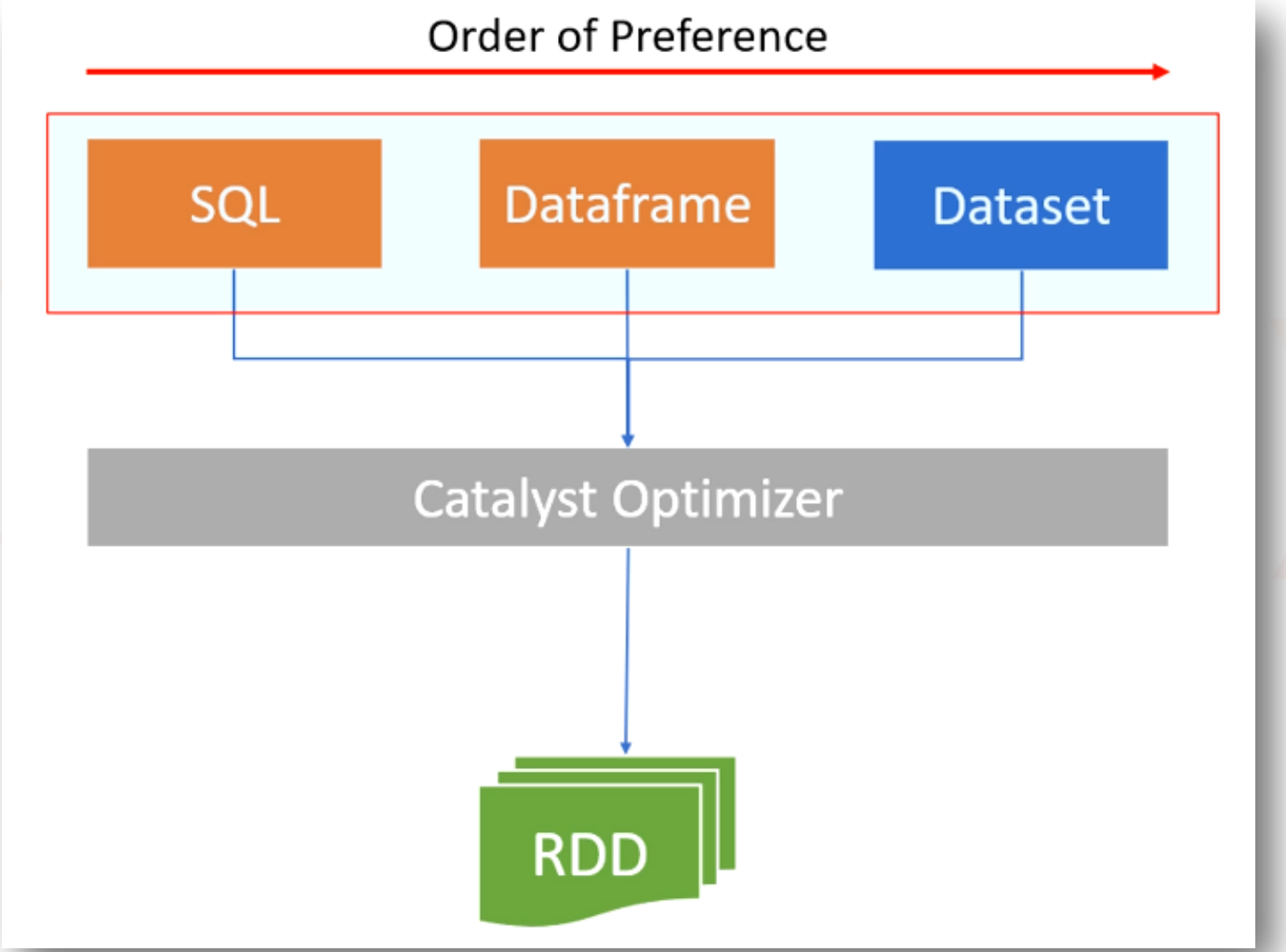
The next one is the catalyst optimizer. So we write code using Spark SQL, Dataframe, and Dataset. This code is then submitted to Spark for execution. However, the code passes through the Catalyst Optimizer, which decides how it should be executed and lays out an execution plan.



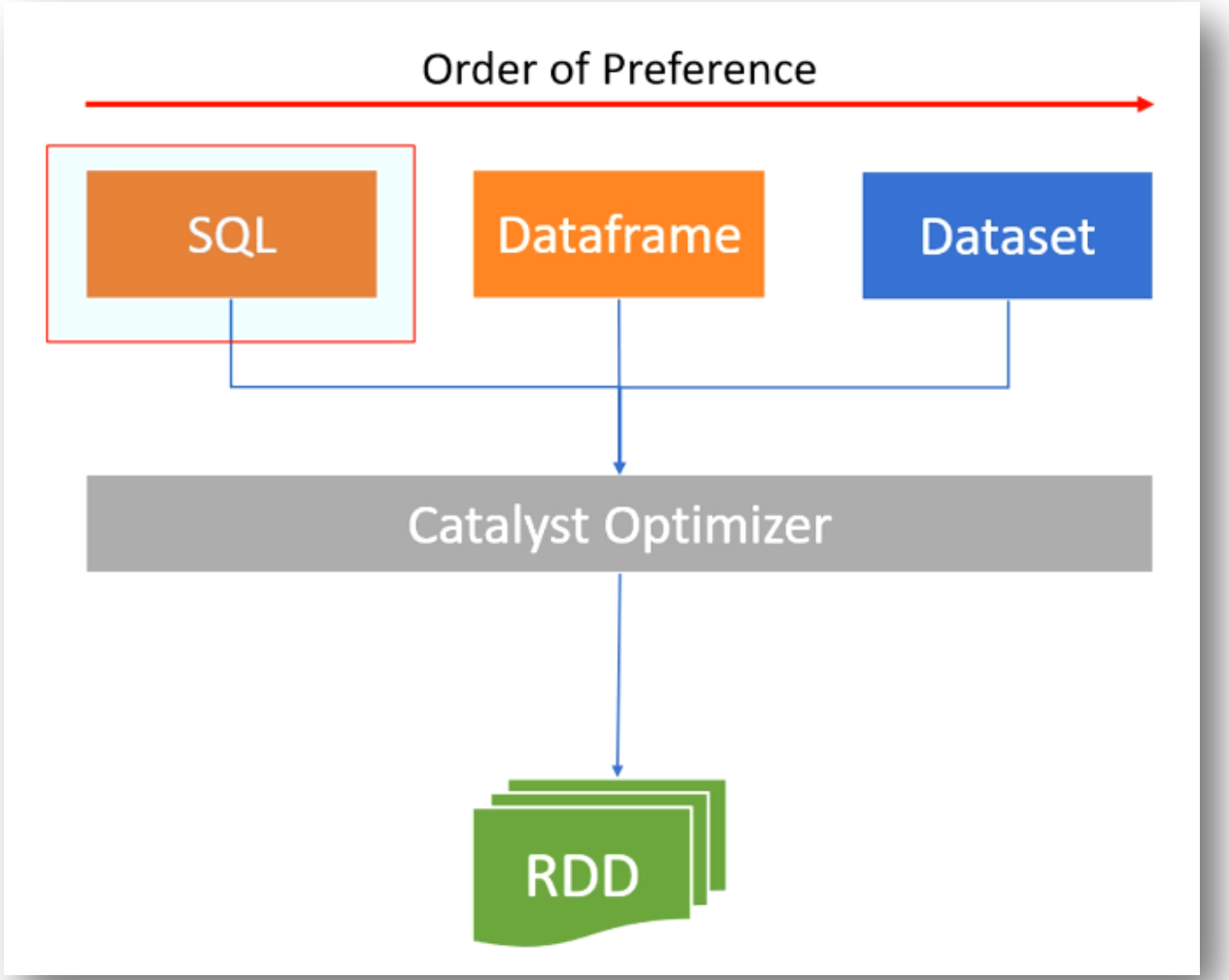
You should understand that the SQL, Dataframe, and Dataset are optimized by the Catalyst optimization and are finally converted to RDD for execution. If you directly write RDD, Spark will simply run it, and the optimizer will have nothing to do.

Why? Because the optimizer comes above the RDD. It reads SQL and Dataframe to produce RDD. If you directly write RDD, you will skip the optimization. So the RDD cannot use optimization, so we should avoid using RDD.

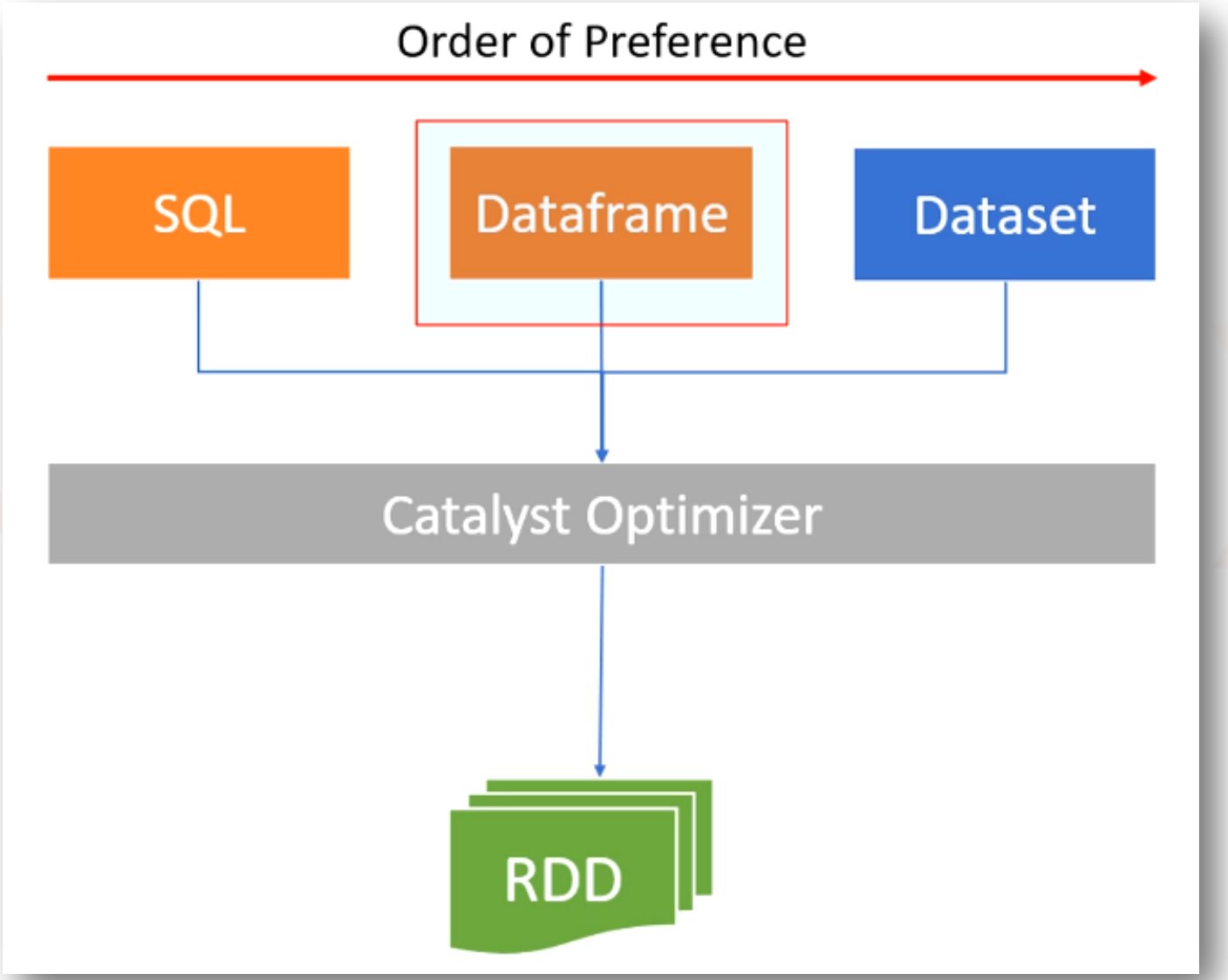
So Spark SQL, Dataframe, and Dataset are your preferred choices. I recommend you prefer them in left-to-right order. I mean, Spark SQL is the most convenient option to use.



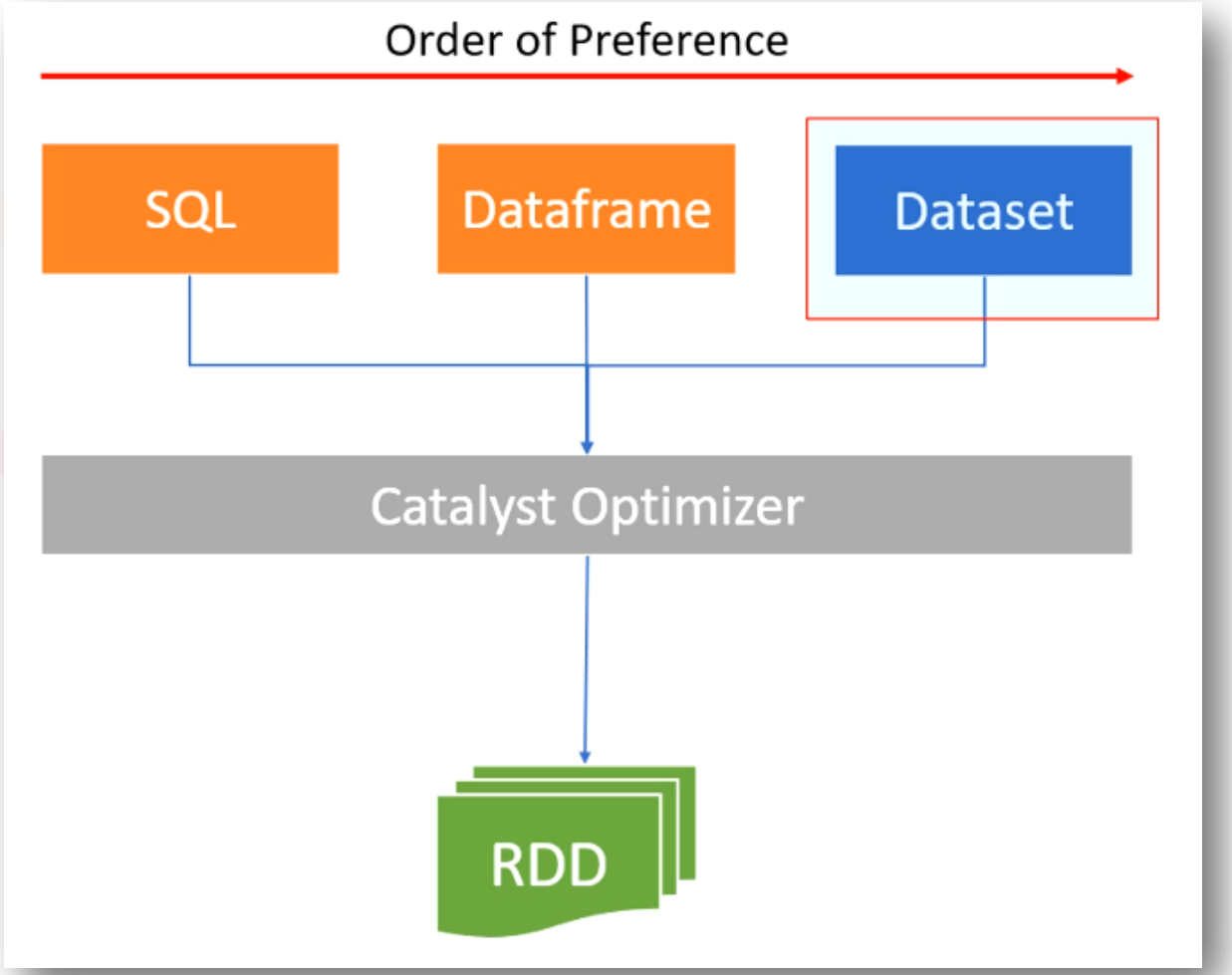
So, you should prefer using it wherever applicable. The Spark SQL is excellent and can do a lot of the work for you. However, a single SQL expression or even a SQL script with a series of SQLs may not be helpful in many scenarios. You do not get some necessary facilities like debugging, implementing application logs, unit testing, and other apparent capabilities of a programming language.

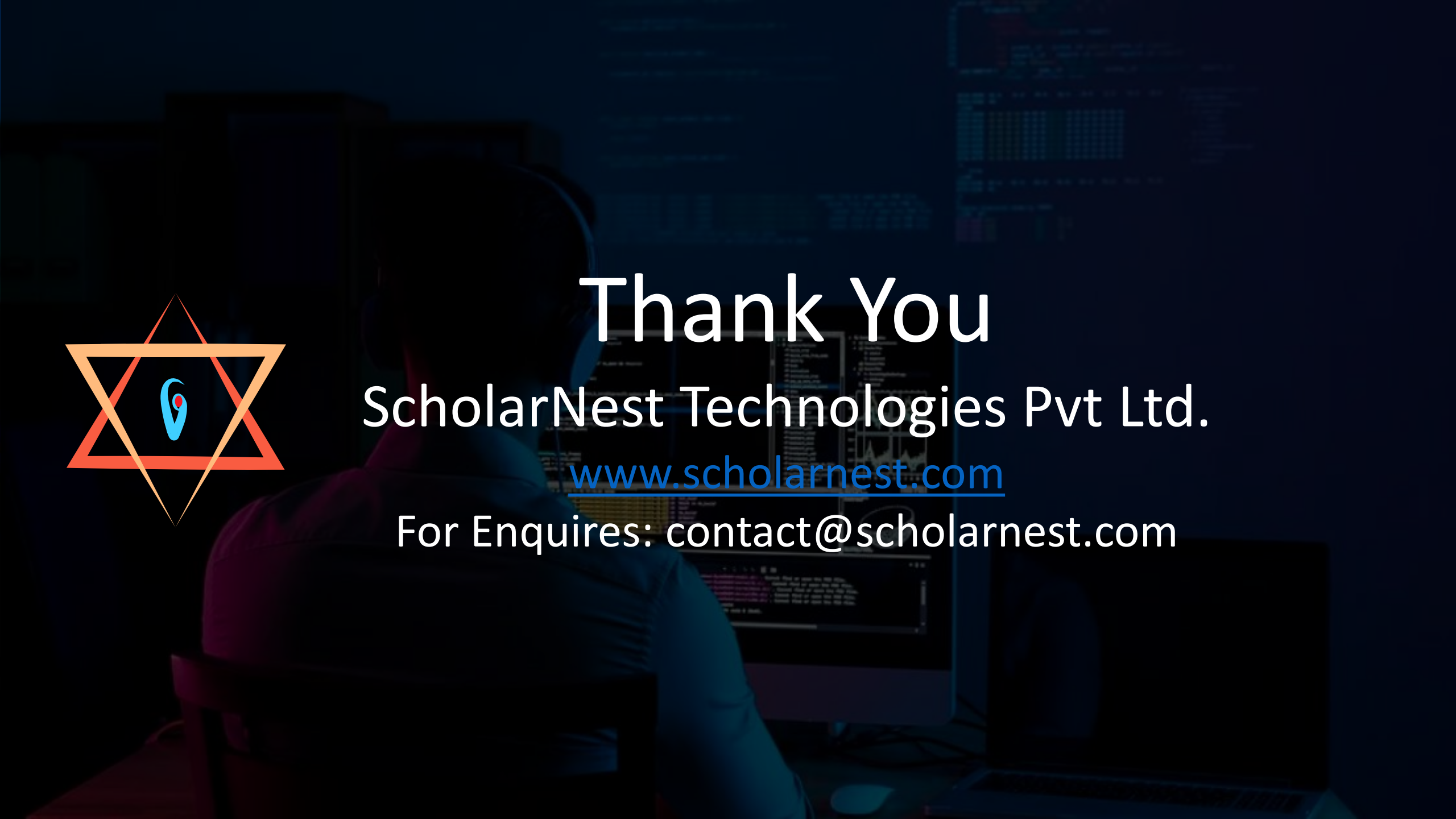


However, a sophisticated data pipeline will push you to use Dataframes and rely more on your favourite programming language. And that's where the Dataframe APIs are your obvious choice.



The last one is the Dataset API. The Dataset APIs are the language-native APIs in Scala and Java. What we mean by language native is that these APIs are strongly typed objects in your JVM-based language, such as Scala. And they are not at all available in dynamically typed languages such as Python.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

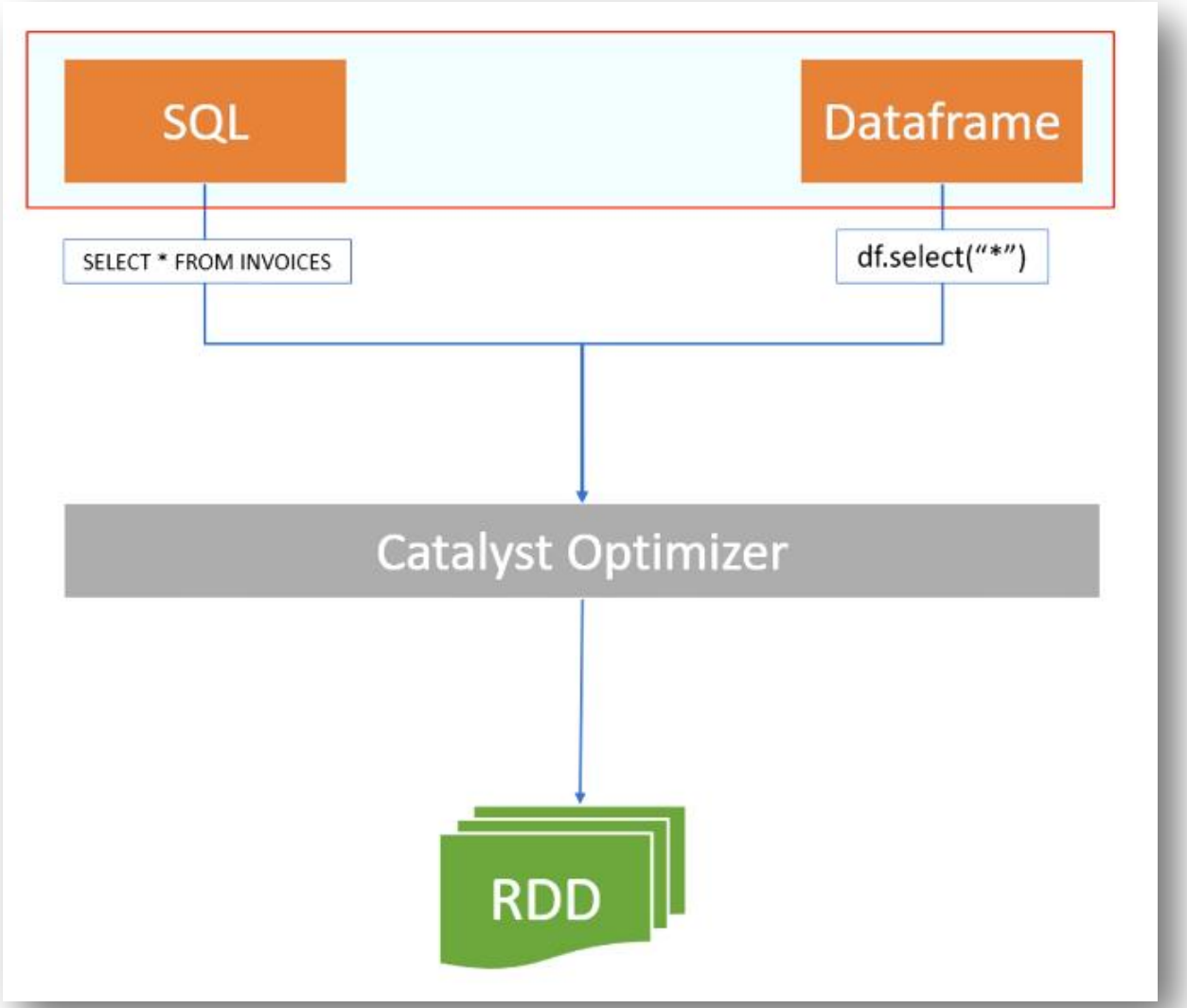
Databricks Spark Certification and Beyond

Module:
Spark Dataframe Internals

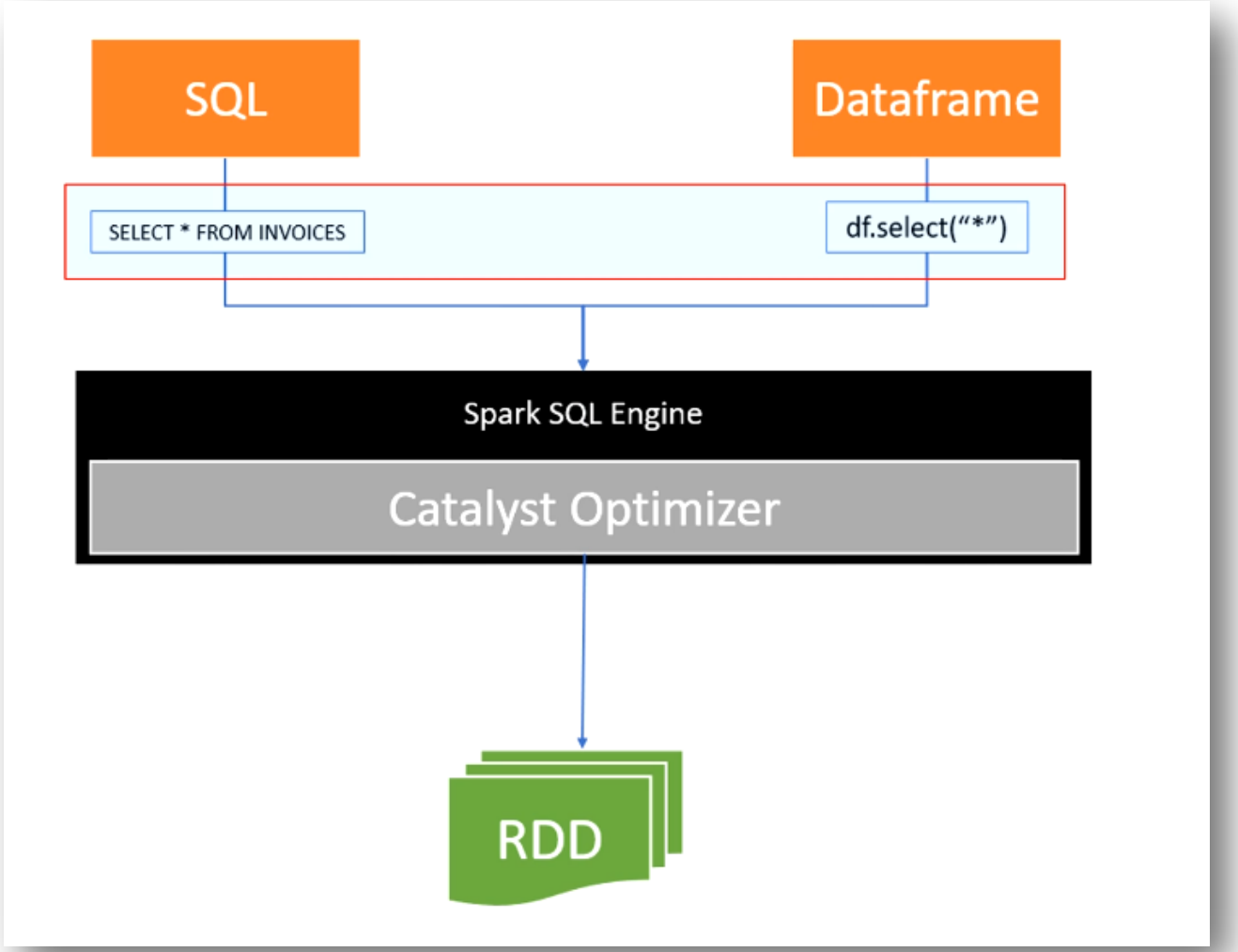
Lecture:
Dataframe, RDD and Distributed Partitions

Dataframes, RDDs and Distributed Partitions

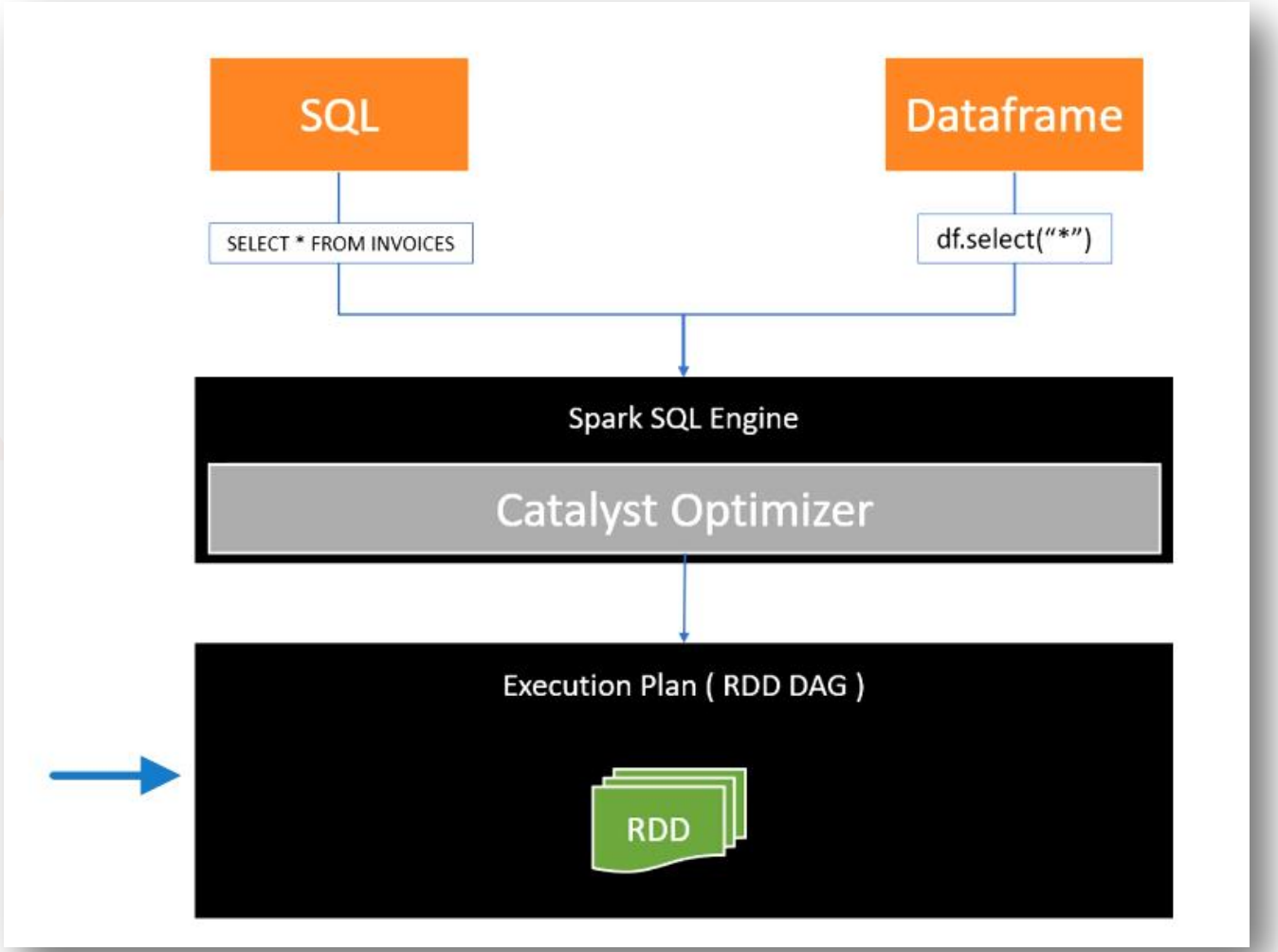
We learned the following design of the Spark APIs. At the top, we have Spark SQL, Dataframe, and Dataset. The Dataset is not available in Python so let me remove it for brevity.
So we have Spark SQL, and we also have Spark Dataframe.



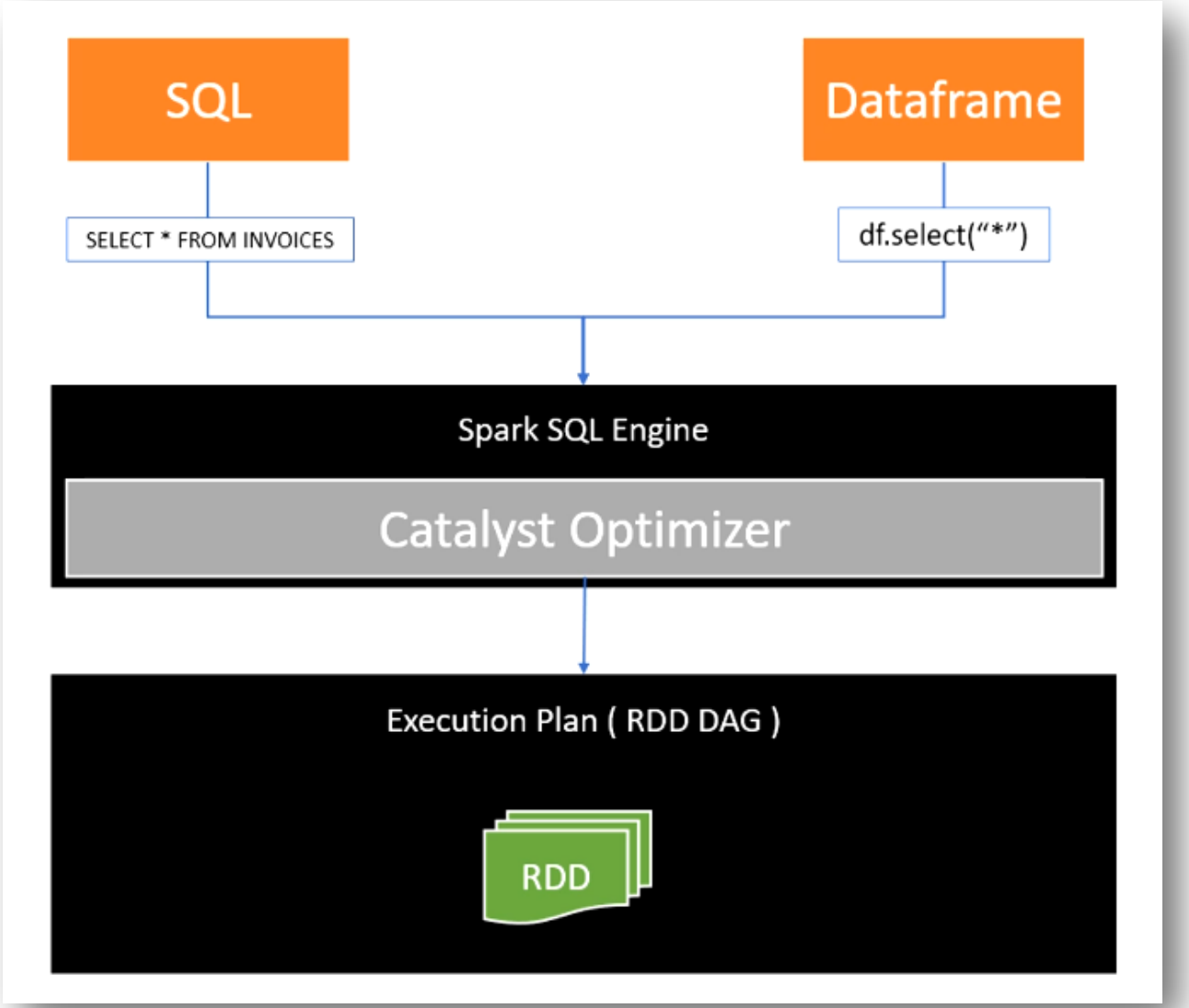
Both of these codes will go to the Spark SQL engine. Your Spark SQL code and the Spark Dataframe code go to the same Spark SQL engine. The Spark SQL engine understands both the syntax. It understands spark SQL and also knows the Dataframe API code.



Now the Spark SQL engine will take your code, parse it and give it to the catalyst optimizer.
The Optimizer will apply some optimizations and create an execution plan.
The execution plan is nothing but a step-by-step approach to How Spark will run your code using the RDD APIs. And that's why Spark RDD is shown at the bottom of the hierarchy.



Spark SQL engine will compile your SQL queries and Dataframe code into RDD APIs. During the compilation, it will also apply a bunch of optimizations. And finally, your code runs as RDD code.



Now let's try to understand the relation between Spark Table, Dataframe, and the RDD. Here is your Spark table shown below. It is a logical view of your table. And the logical view of the Dataframe is also the same.

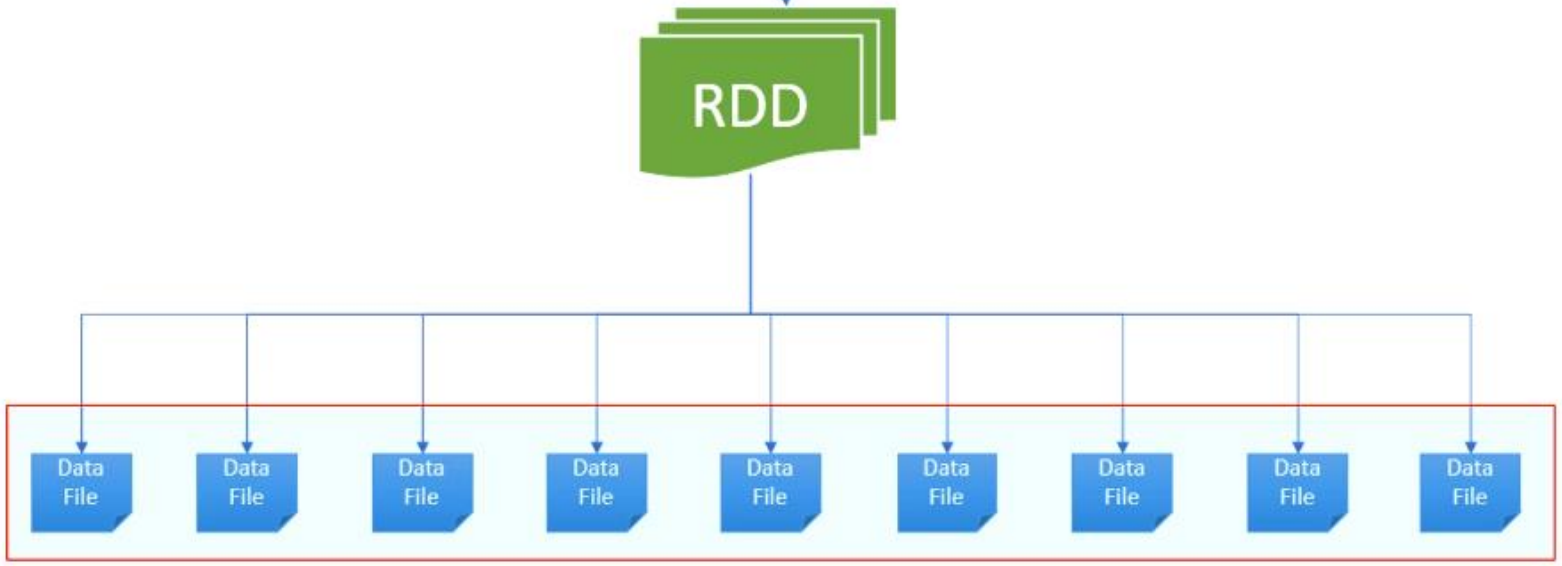
InvoiceNo	StockCode	Description	InvoiceDate	Quantity	UnitPrice	Country
565231	47504K	ENGLISH ROSE GARDEN SECATEURS	01-06-2022 9.26	1	3.29	United Kingdom
565231	47566	PARTY BUNTING	01-06-2022 9.26	2	10.79	United Kingdom
565231	51014A	FEATHER PEN,HOT PINK	01-06-2022 9.26	2	0.83	United Kingdom
565231	72760B	VINTAGE CREAM 3 BASKET CAKE STAND	01-06-2022 9.26	1	20.79	United Kingdom
565231	72807A	SET/3 ROSE CANDLE IN JEWELLED BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	72807C	SET/3 VANILLA SCENTED CANDLE IN BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	82551	LAUNDRY 15C METAL SIGN	01-06-2022 9.26	1	2.46	United Kingdom
565231	82567	AIRLINE LOUNGE,METAL SIGN	01-06-2022 9.26	1	1.63	United Kingdom

But internally, your table and the Dataframe are nothing but Spark RDD.
Why? Because Spark SQL engine compiles everything to RDD. So The data frame and the Spark table are nothing but a logical view over and above the Spark RDD.

InvoiceNo	StockCode	Description	InvoiceDate	Quantity	UnitPrice	Country
565231	47504K	ENGLISH ROSE GARDEN SECATEURS	01-06-2022 9.26	1	3.29	United Kingdom
565231	47566	PARTY BUNTING	01-06-2022 9.26	2	10.79	United Kingdom
565231	51014A	FEATHER PEN,HOT PINK	01-06-2022 9.26	2	0.83	United Kingdom
565231	72760B	VINTAGE CREAM 3 BASKET CAKE STAND	01-06-2022 9.26	1	20.79	United Kingdom
565231	72807A	SET/3 ROSE CANDLE IN JEWELLED BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	72807C	SET/3 VANILLA SCENTED CANDLE IN BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	82551	LAUNDRY 15C METAL SIGN	01-06-2022 9.26	1	2.46	United Kingdom
565231	82567	AIRLINE LOUNGE,METAL SIGN	01-06-2022 9.26	1	1.63	United Kingdom

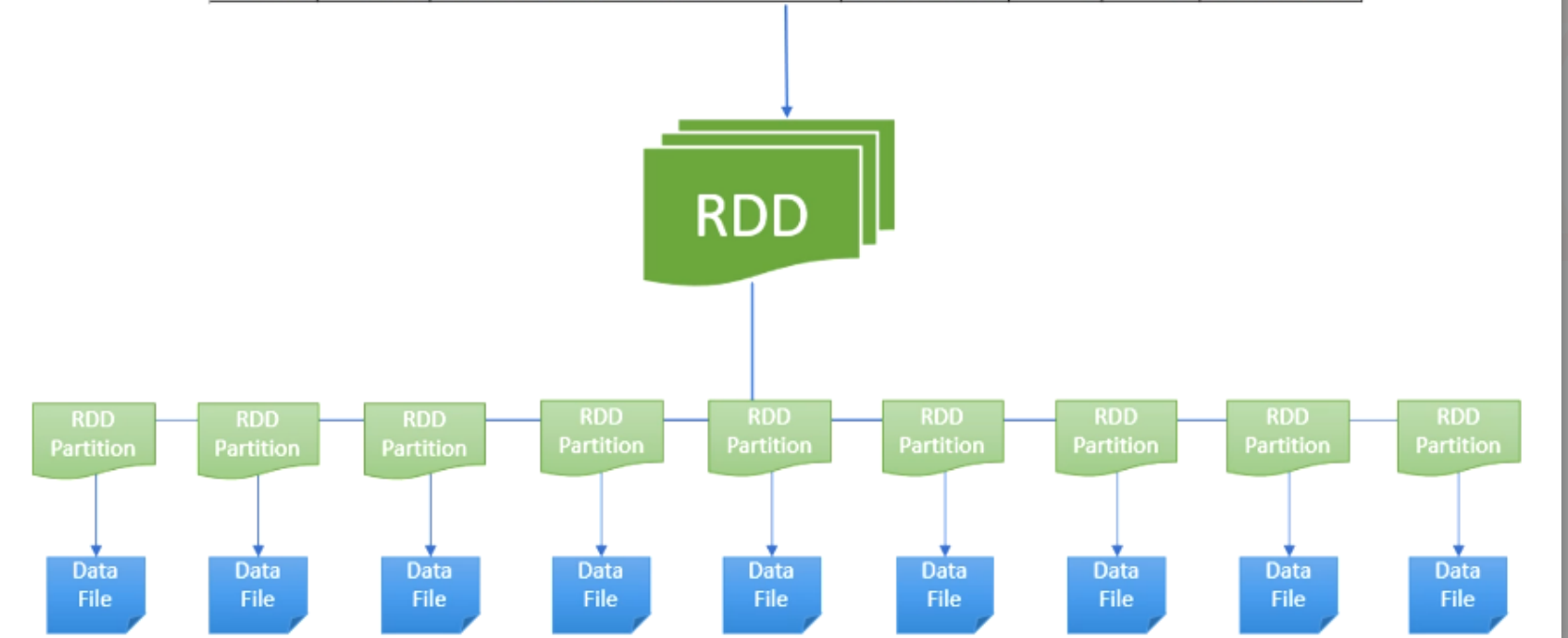
What is an RDD? It is similar to a Data List.
The data is stored in the data files. But RDD points to those data files, and they know where the data is stored. And at runtime, the RDD will read data from the files and bring it to memory.

InvoiceNo	StockCode	Description	InvoiceDate	Quantity	UnitPrice	Country
565231	47504K	ENGLISH ROSE GARDEN SECATEURS	01-06-2022 9.26	1	3.29	United Kingdom
565231	47566	PARTY BUNTING	01-06-2022 9.26	2	10.79	United Kingdom
565231	51014A	FEATHER PEN,HOT PINK	01-06-2022 9.26	2	0.83	United Kingdom
565231	72760B	VINTAGE CREAM 3 BASKET CAKE STAND	01-06-2022 9.26	1	20.79	United Kingdom
565231	72807A	SET/3 ROSE CANDLE IN JEWELLED BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	72807C	SET/3 VANILLA SCENTED CANDLE IN BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	82551	LAUNDRY 15C METAL SIGN	01-06-2022 9.26	1	2.46	United Kingdom
565231	82567	AIRLINE LOUNGE,METAL SIGN	01-06-2022 9.26	1	1.63	United Kingdom



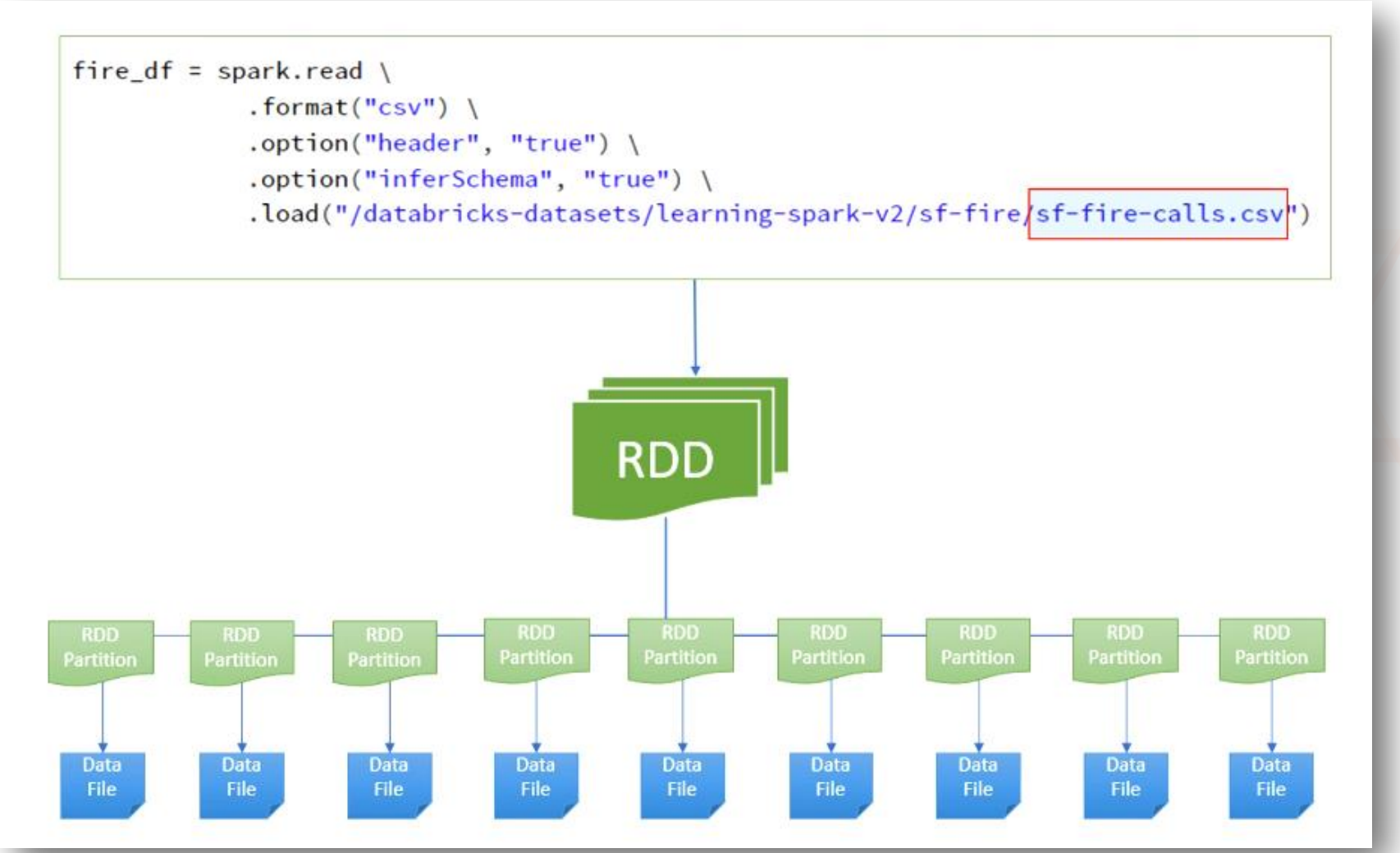
But the RDD is not a simple data list.
It is a distributed data structure. So we see one single RDD here, but internally it is made up of RDD partitions.

InvoiceNo	StockCode	Description	InvoiceDate	Quantity	UnitPrice	Country
565231	47504K	ENGLISH ROSE GARDEN SECATEURS	01-06-2022 9.26	1	3.29	United Kingdom
565231	47566	PARTY BUNTING	01-06-2022 9.26	2	10.79	United Kingdom
565231	51014A	FEATHER PEN,HOT PINK	01-06-2022 9.26	2	0.83	United Kingdom
565231	72760B	VINTAGE CREAM 3 BASKET CAKE STAND	01-06-2022 9.26	1	20.79	United Kingdom
565231	72807A	SET/3 ROSE CANDLE IN JEWELLED BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	72807C	SET/3 VANILLA SCENTED CANDLE IN BOX	01-06-2022 9.26	1	8.29	United Kingdom
565231	82551	LAUNDRY 15C METAL SIGN	01-06-2022 9.26	1	2.46	United Kingdom
565231	82567	AIRLINE LOUNGE,METAL SIGN	01-06-2022 9.26	1	1.63	United Kingdom

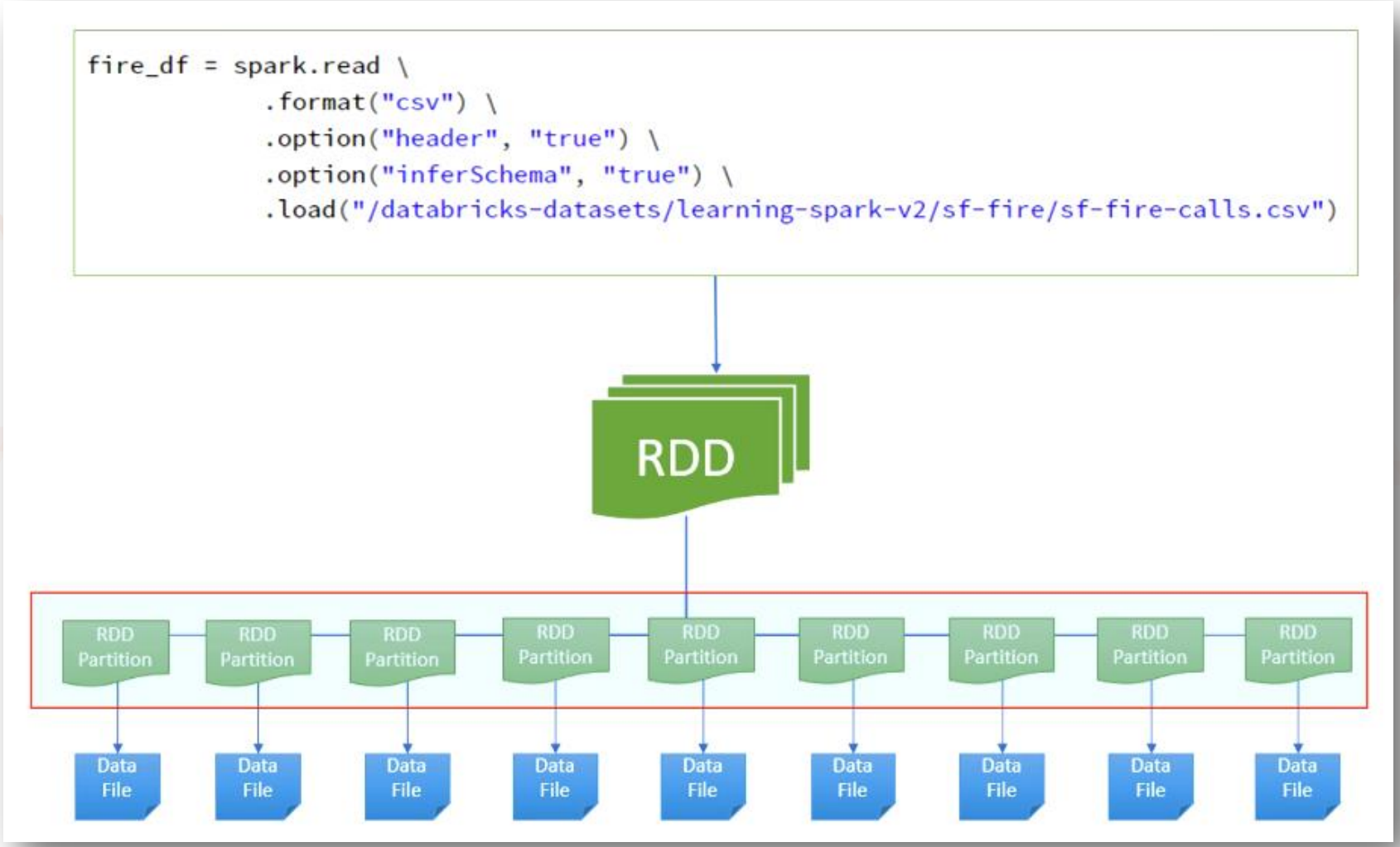


I loaded a data file to create a Dataframe. Here is the code for the same shown below. So I am loading the *sf-fire-calls.csv* file. This file has more than 43 million records. And all those 43 million records are not loaded into one single RDD. They are loaded into n number of RDD partitions.

```
fire_df = spark.read \
            .format("csv") \
            .option("header", "true") \
            .option("inferSchema", "true") \
            .load("/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv")
```



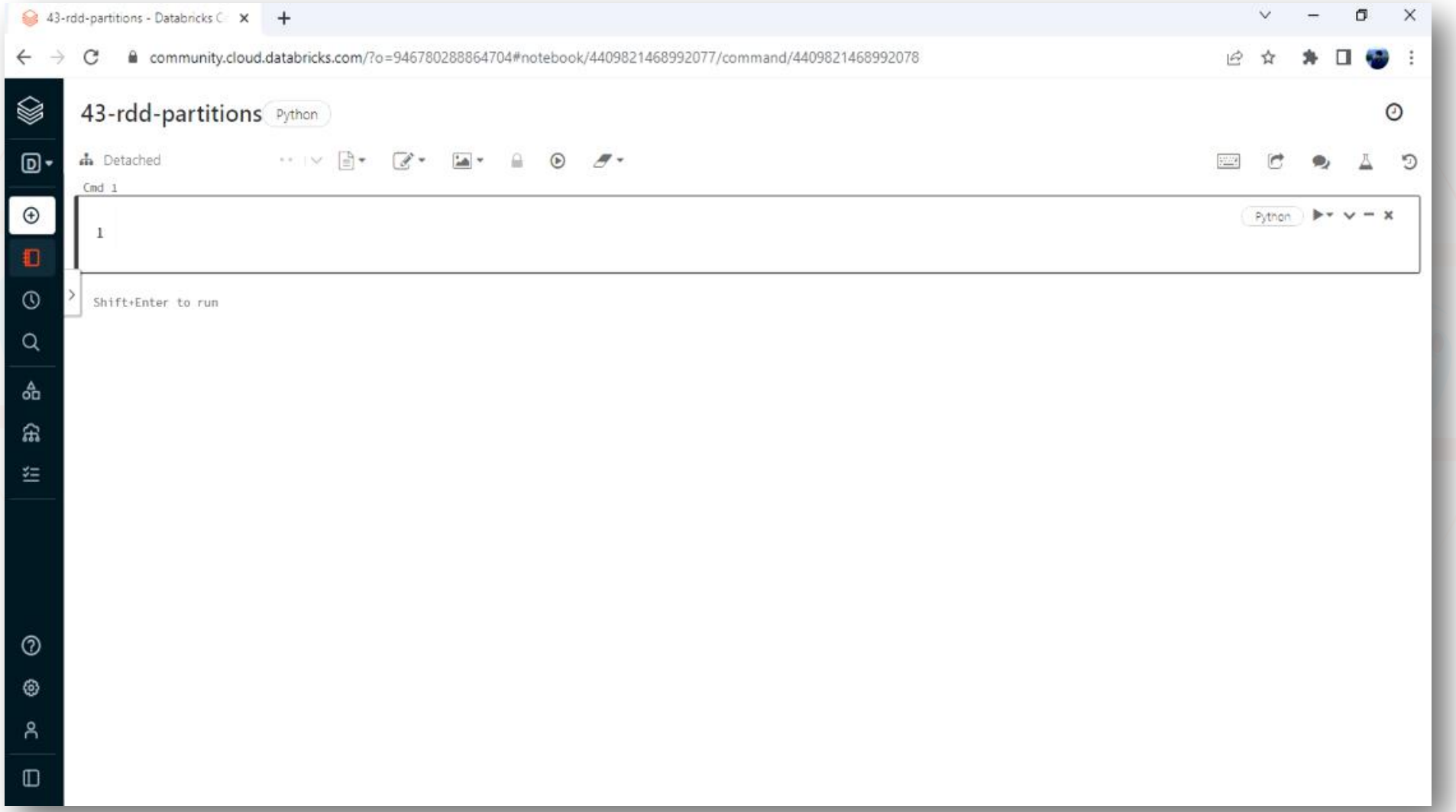
Let's assume we have 9 RDD partitions in this example. So this single large logical RDD comprises nine physical RDD partitions. The Dataframe knows about the one logical RDD. The logical RDD knows about the nine physical RDD partitions.



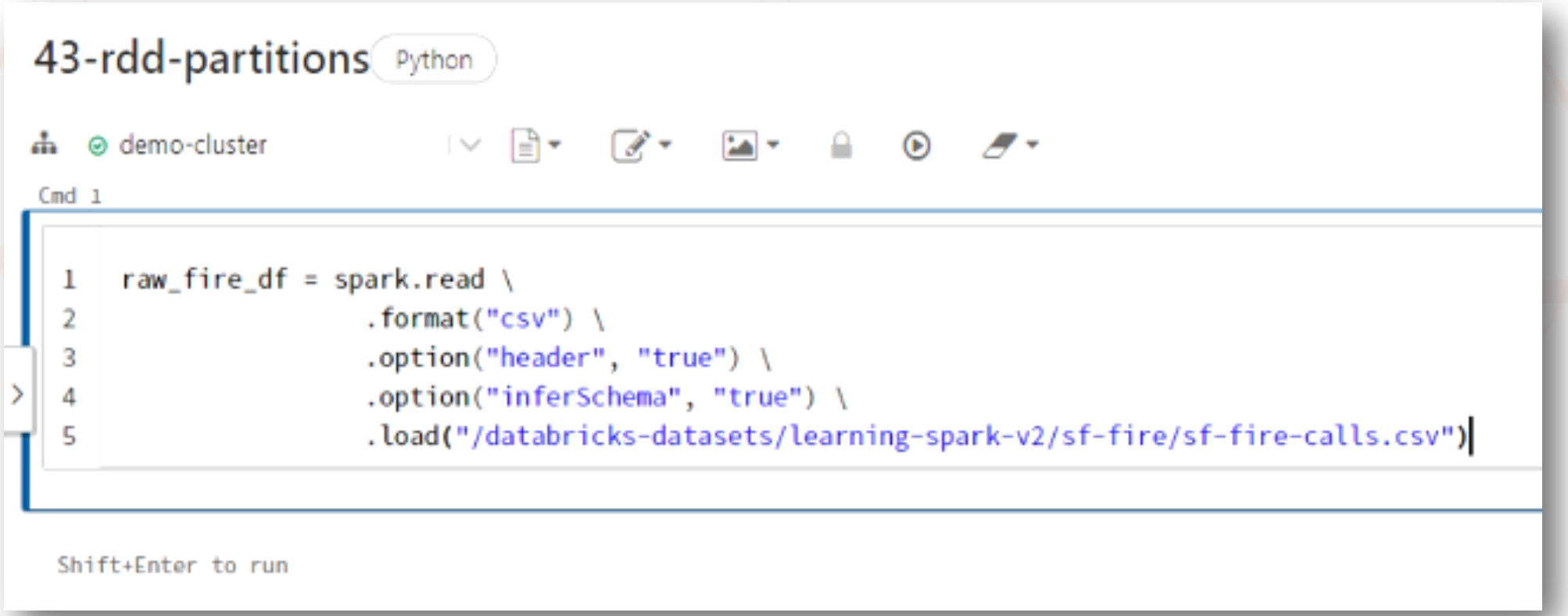
We see one single Dataframe, but it is made up of n number of physical RDD partitions. And the main logical RDD knows how to create and recreate the RDD partitions. If one RDD partition is lost, the primary logical RDD knows how to recreate that RDD partition. And that's why RDD is named a resilient distributed Dataset.

It is resilient because Spark knows how to recreate the lost RDD partition.
It is distributed because the logical RDD comprises physical RDD partitions. It is a dataset because it stores in-memory data.

Let us see all this in action.
Go to your data bricks workspace and create a new notebook. **(Reference: 43-rdd-partitions.ipynb)**



I have created a Dataframe as shown below. If I run this code, it will go to the Spark SQL engine. The SQL engine will create an optimized execution plan and run the plan. After running the execution plan, we will get a logical RDD which is made up of some physical RDD partitions.
You cannot see the logical RDD or the RDD partitions because Spark will free up the memory after running my code. But if we cache the Dataframe, Spark will keep everything in the memory, and we can see it.



So I am caching the Dataframe and counting the number of records, so everything comes in memory. Why counting? Because Spark is lazy. It will not create Physical RDD partitions until we really need the data in the memory. When we count it, Spark will know that we need the data in the memory, so it will create the physical RDD partitions, bring the data into memory, give the count, and cache the physical RDD partitions in the memory.

Cmd 1

```
1  raw_fire_df = spark.read \
2                  .format("csv") \
3                  .option("header", "true") \
4                  .option("inferSchema", "true") \
5                  .load("/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv")
```

Cmd 2

```
1  raw_fire_df.cache()
2  raw_fire_df.count()
```

Shift+Enter to run

Now, run both your cells and you will see the count displayed in the second cell's output.

Cmd 1

```
1  raw_fire_df = spark.read \
2                 .format("csv") \
3                 .option("header", "true") \
4                 .option("inferSchema", "true") \
5                 .load("/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv")
```

▸ (2) Spark Jobs

Command took 1.15 minutes -- by prashant@scholarnest.com at 7/18/2022, 3:53:37 PM on demo-cluster

Cmd 2

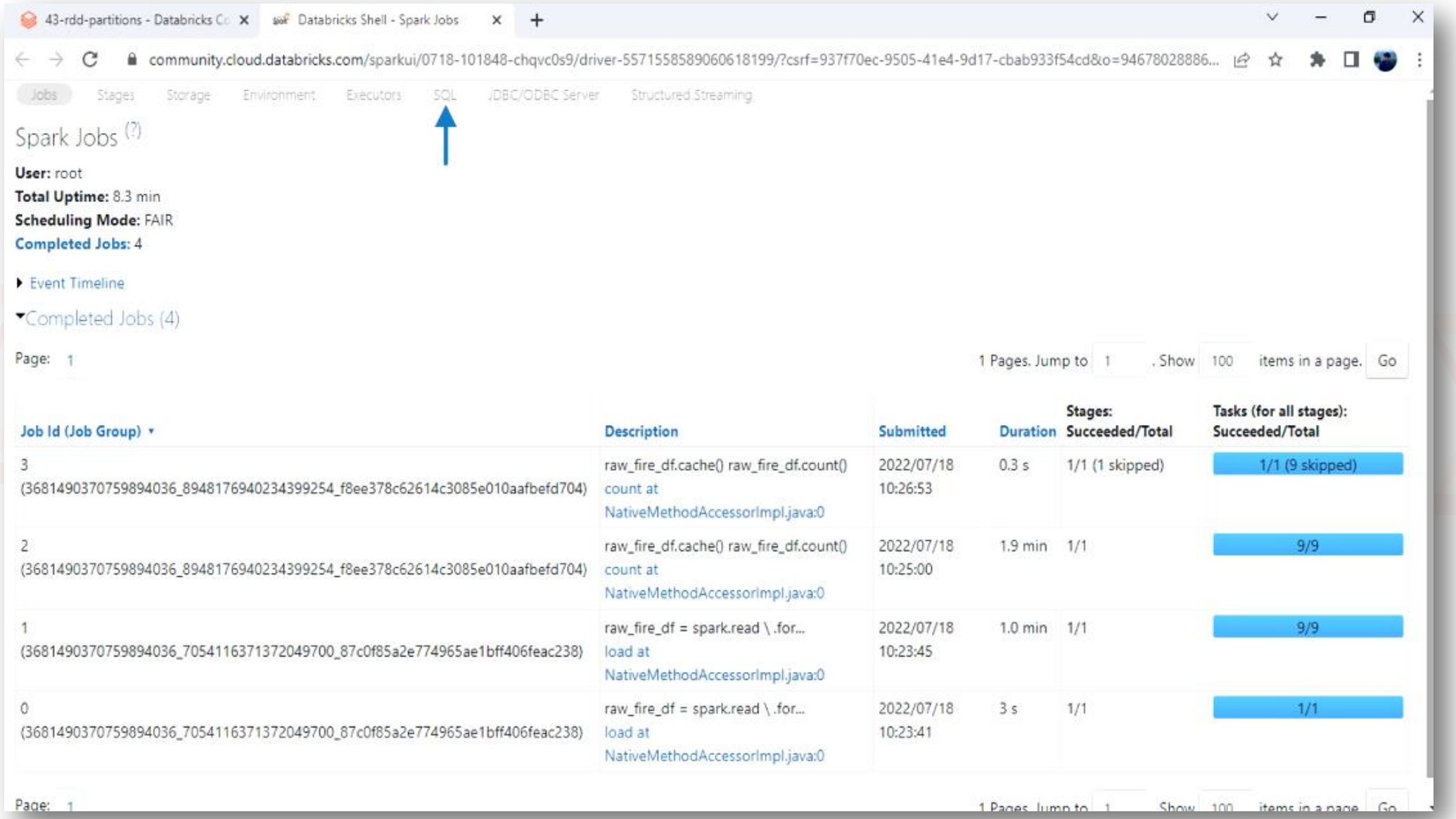
```
1  raw_fire_df.cache()
2  raw_fire_df.count()
```

▸ (2) Spark Jobs

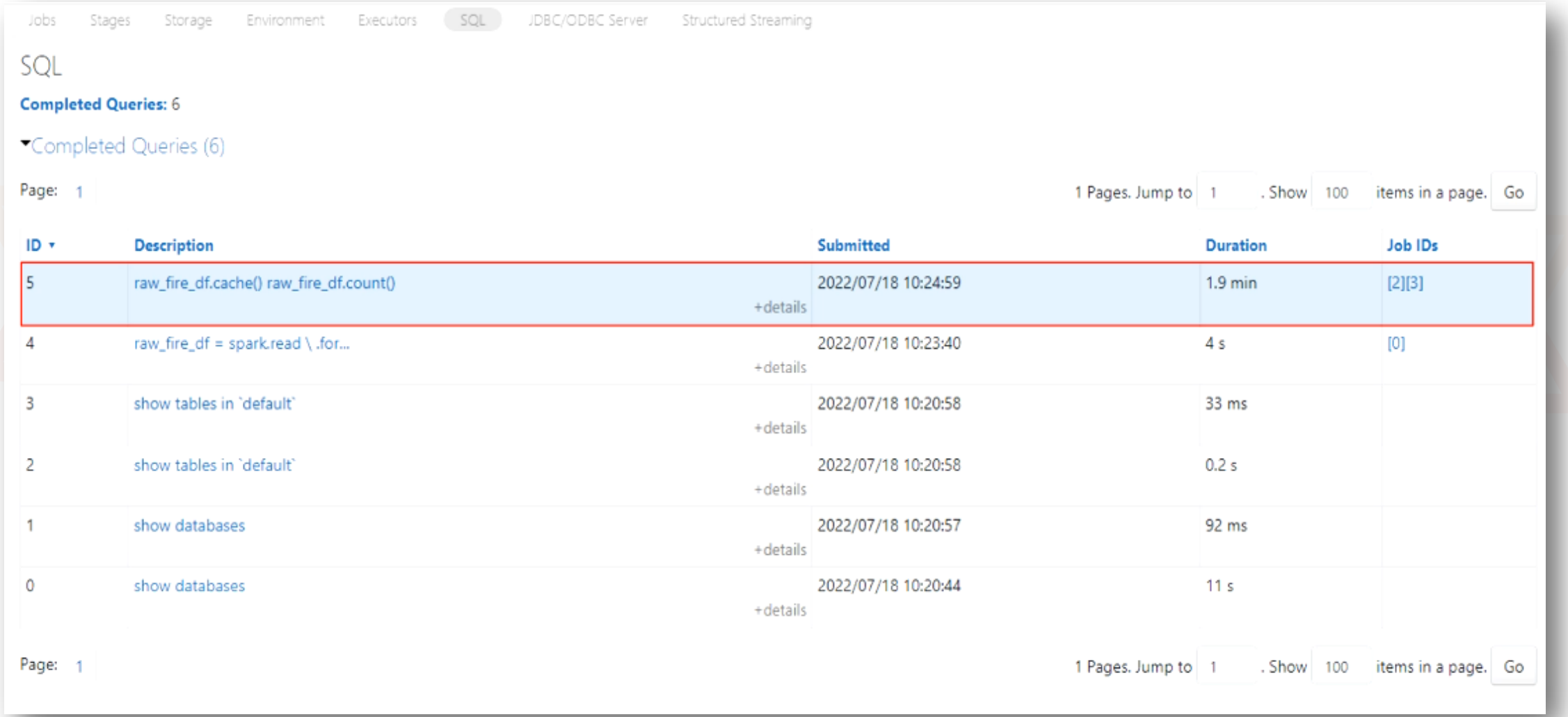
```
Out[2]: 4380660
```

Command took 1.91 minutes -- by prashant@scholarnest.com at 7/18/2022, 3:54:59 PM on demo-cluster

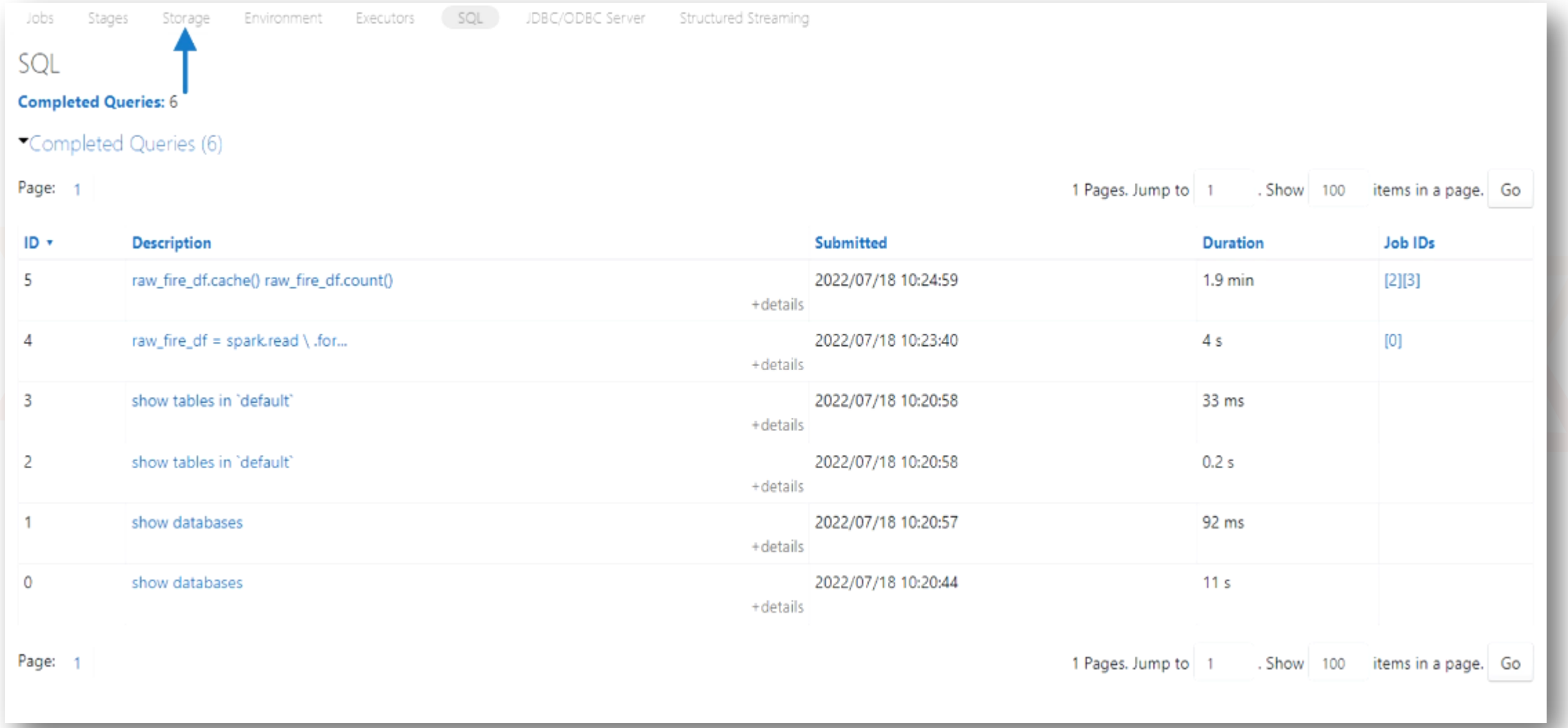
Then go to the Spark UI, and navigate to the SQL tab.



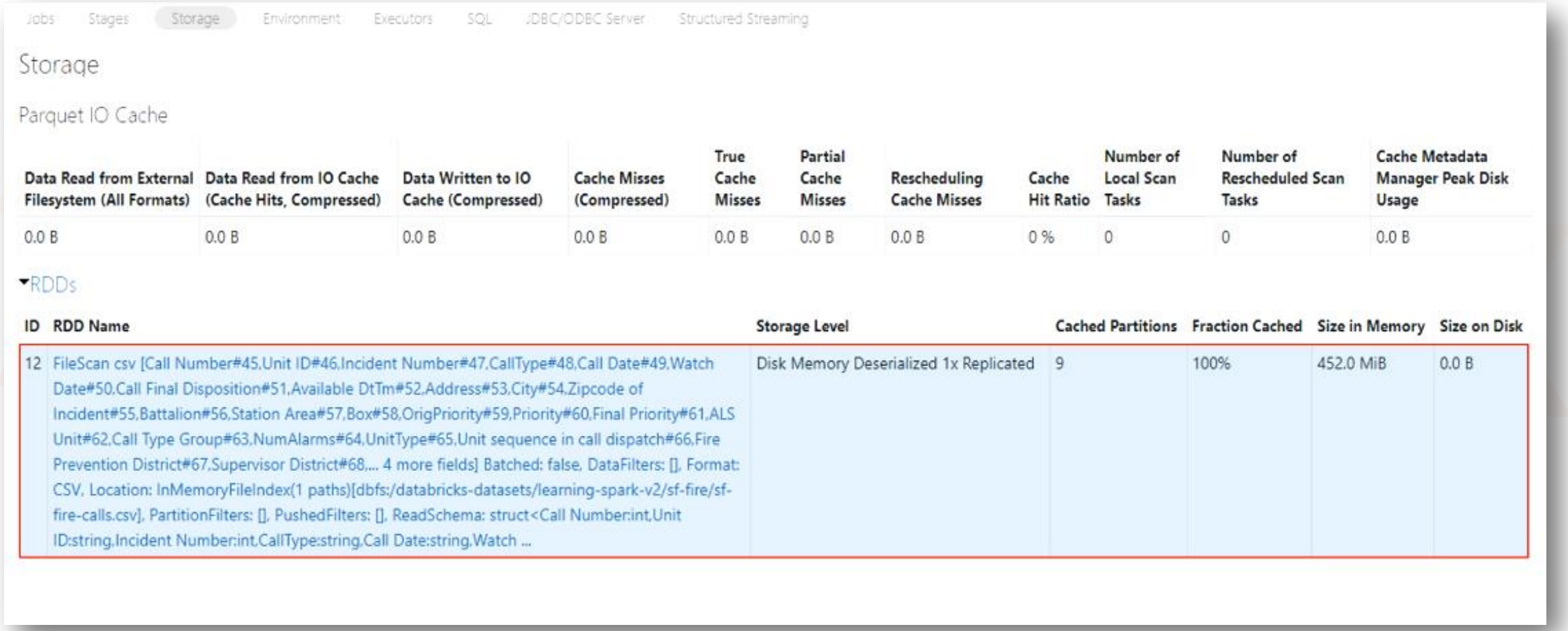
You will see your code here.
The last one is the raw_fire_df.count(). Before that, we see spark.read() method was executed.



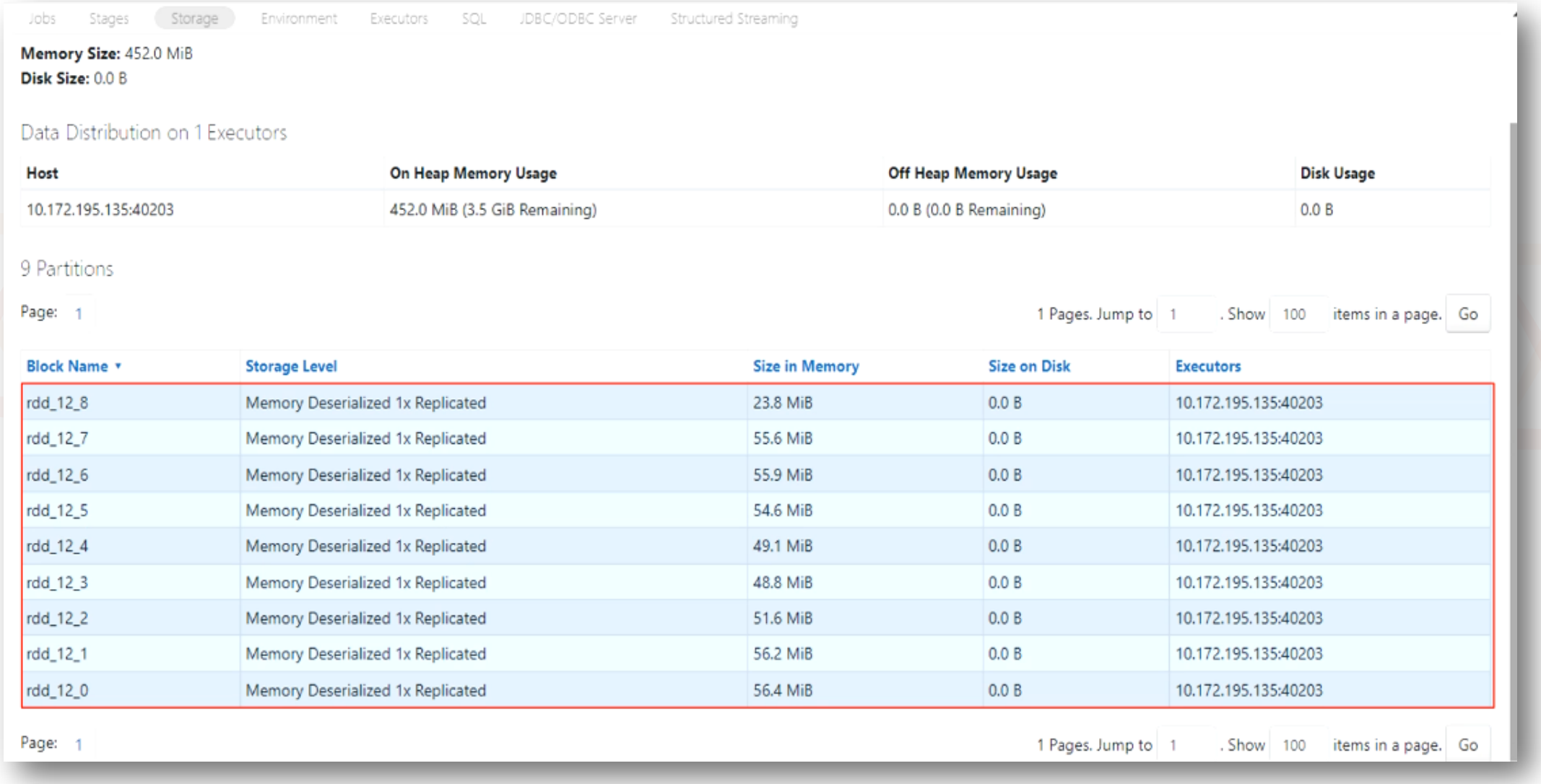
Now go to the storage tab to see the logical RDD and the physical RDD partitions.

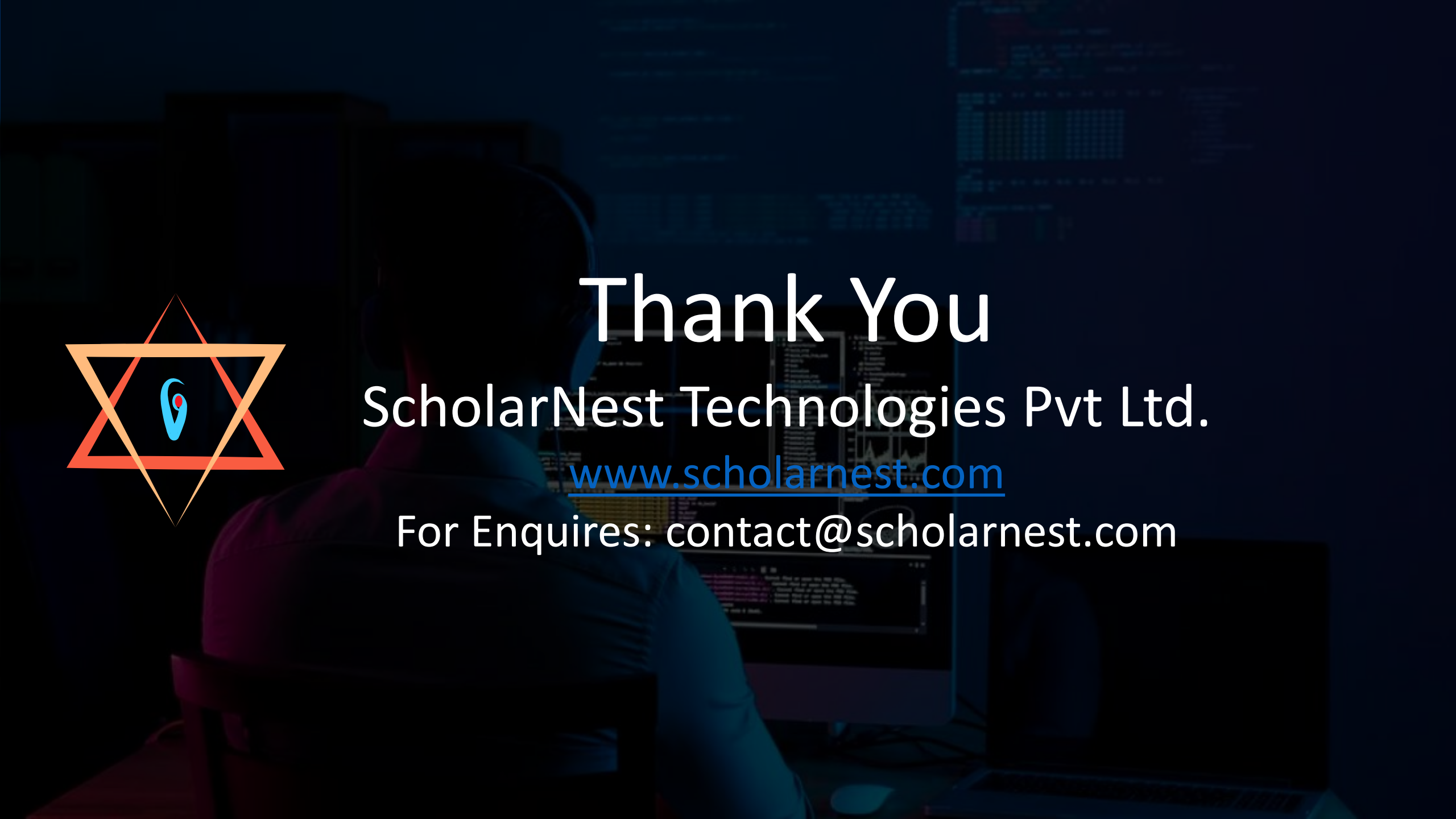


You will see an internal RDD ID, RDD name, and where is it cached.
Click the RDD name, and you will see more details.



So here is the list of RDD partitions.
You have the partition name. You can also see the size of the partition.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

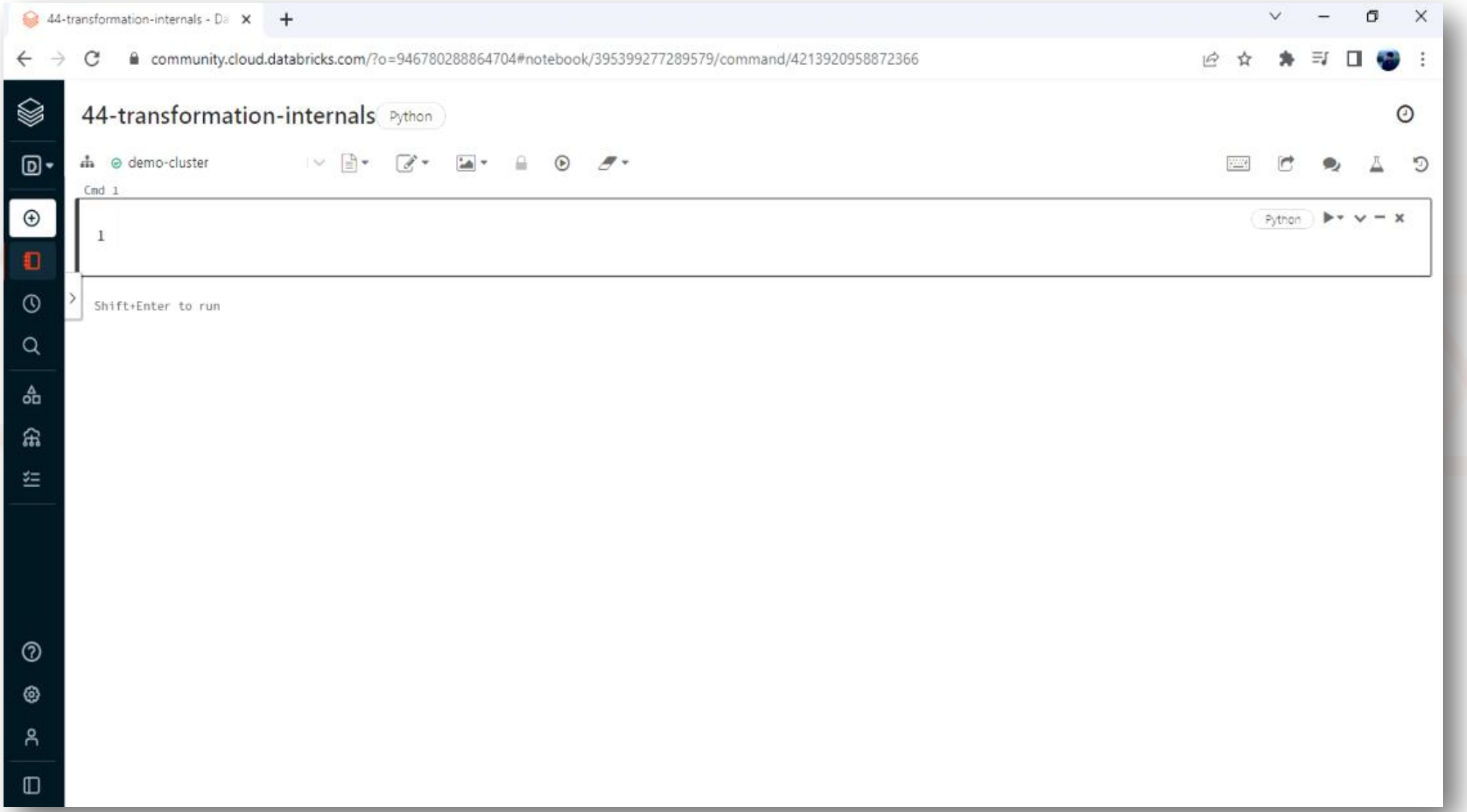
Databricks Spark Certification and Beyond

Module:
Spark Dataframe Internals

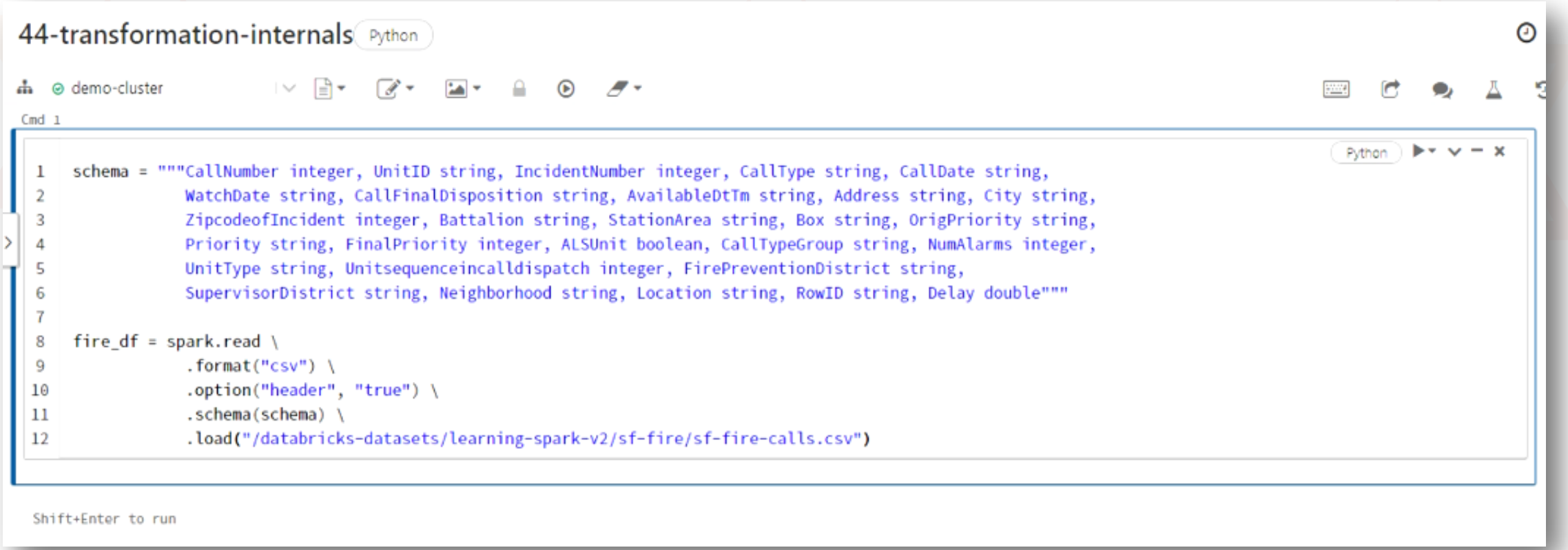
Lecture:
Internals of Transformation

Internals of Transformation

Go to your Databricks workspace and create a new notebook. **(Reference: 44-transformation-internals.ipynb)**



We already learned how to read a file and create a Spark Dataframe.
I have a *fire_df* Dataframe. You already learned that the Dataframe is internally made up of RDD partitions. So my *fire_df* Dataframe is also made up of RDD partitions. I can simply think of my Dataframe as a single large Dataframe structure. But internally, it is partitioned. And that's what Spark brings to us - hiding all the complexities of partitioning and distributed processing from the programmers. We can simply work with a Dataframe as we work with a simple non-distributed data structure.

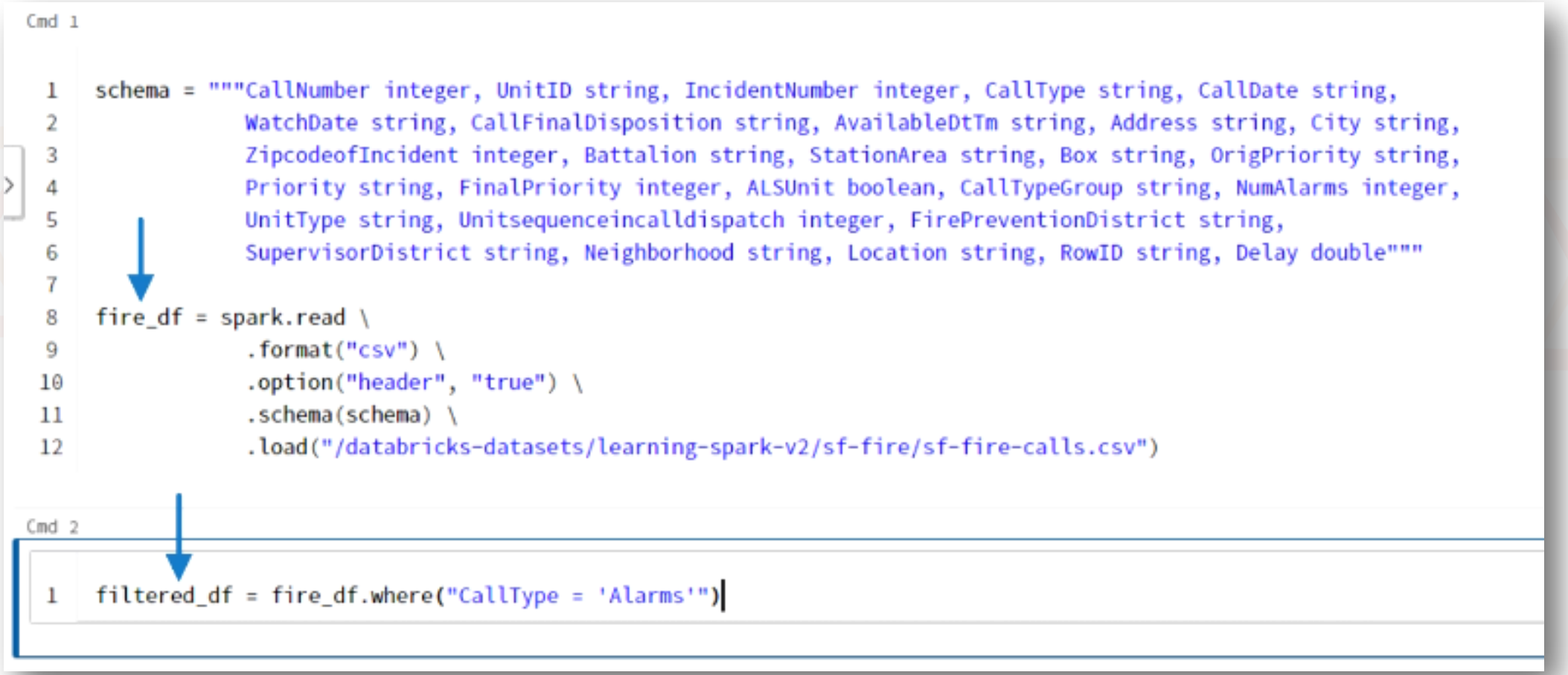


So after reading our data, we are now ready to process it. However, Spark Dataframe and the internal RDD are immutable data structures. That means you can create it once but cannot change it later.
However, you can always transform your old Dataframe and create a new Dataframe, as shown below.

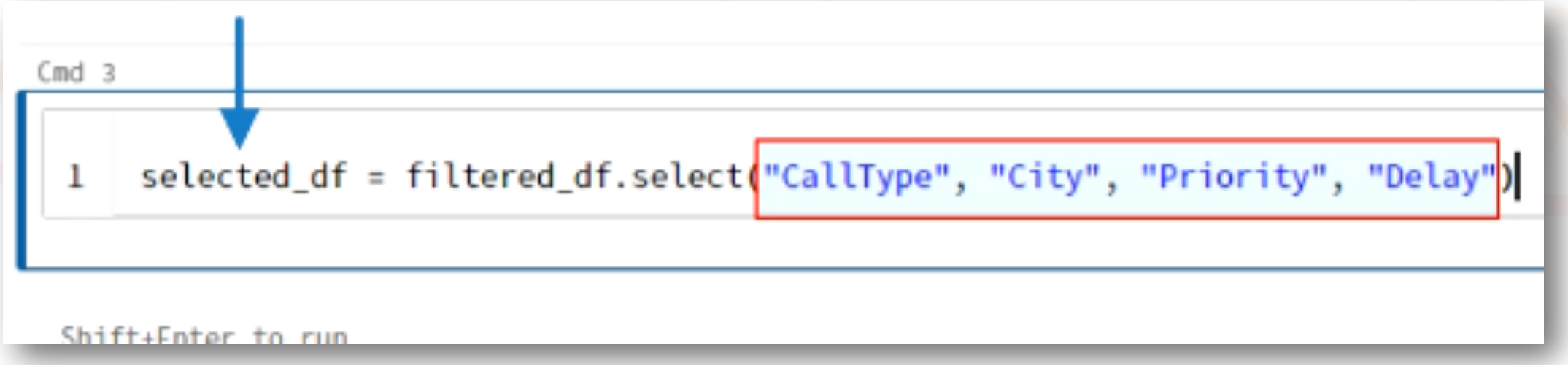
Cmd 2

```
1  filtered_df = fire_df.where("CallType = 'Alarms'")
```


So, I took only those records where the *CallType* is *Alarms* from the *fire_df* and created a new Dataframe. The new Dataframe name is *filtered_df*. I haven't changed the *fire_df*. I simply took some records from the *fire_df* and created a new Dataframe. The *fire_df* Dataframe remains unchanged.



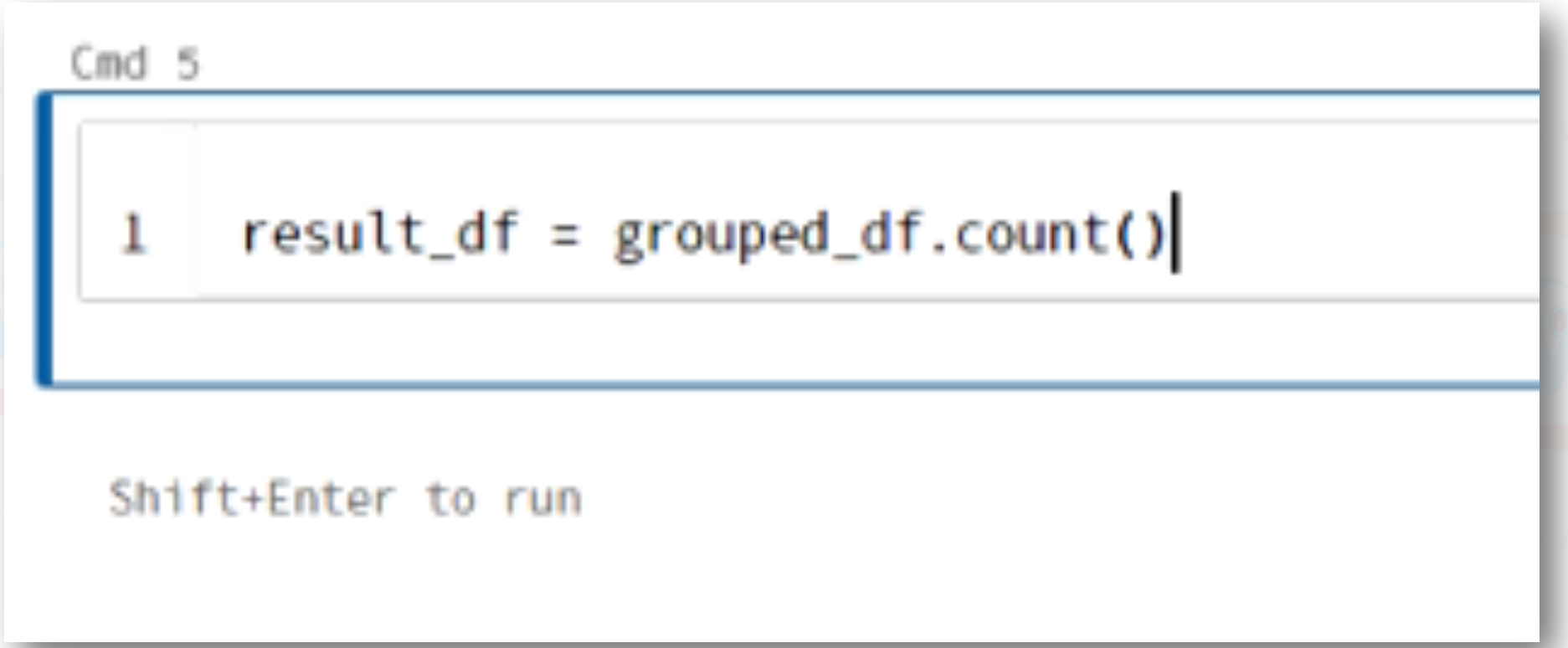
Here is another transformation.
I am selecting only four columns from the *filtered_df* Dataframe, and creating *selected_df*.



Here is one more.
I am grouping data on the Priority column and creating a new Dataframe.

```
Cmd 4

1   grouped_df = selected_df.groupBy("Priority")
```

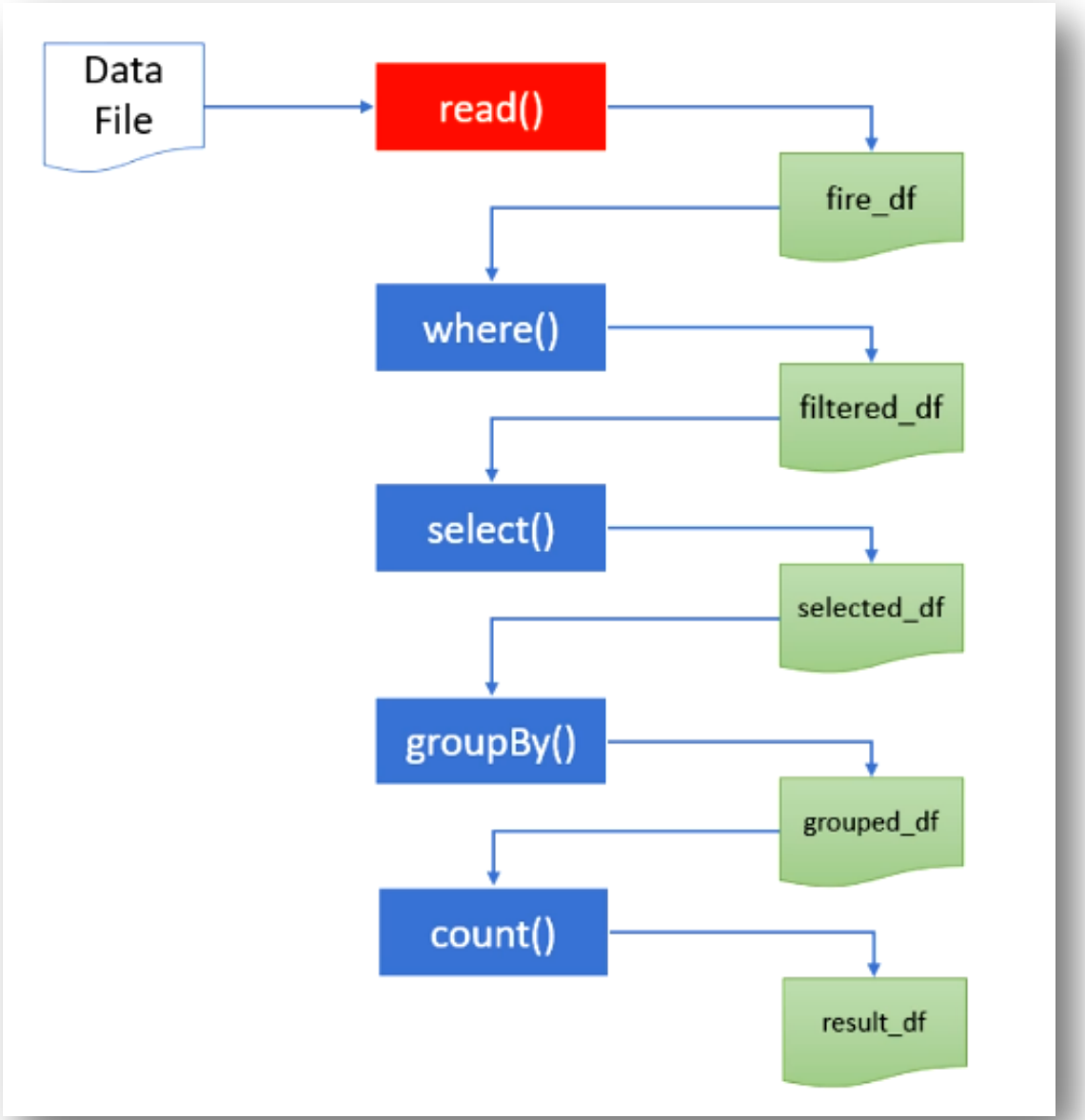

Finally, I took count from the *grouped_df* and created a new result Dataframe.



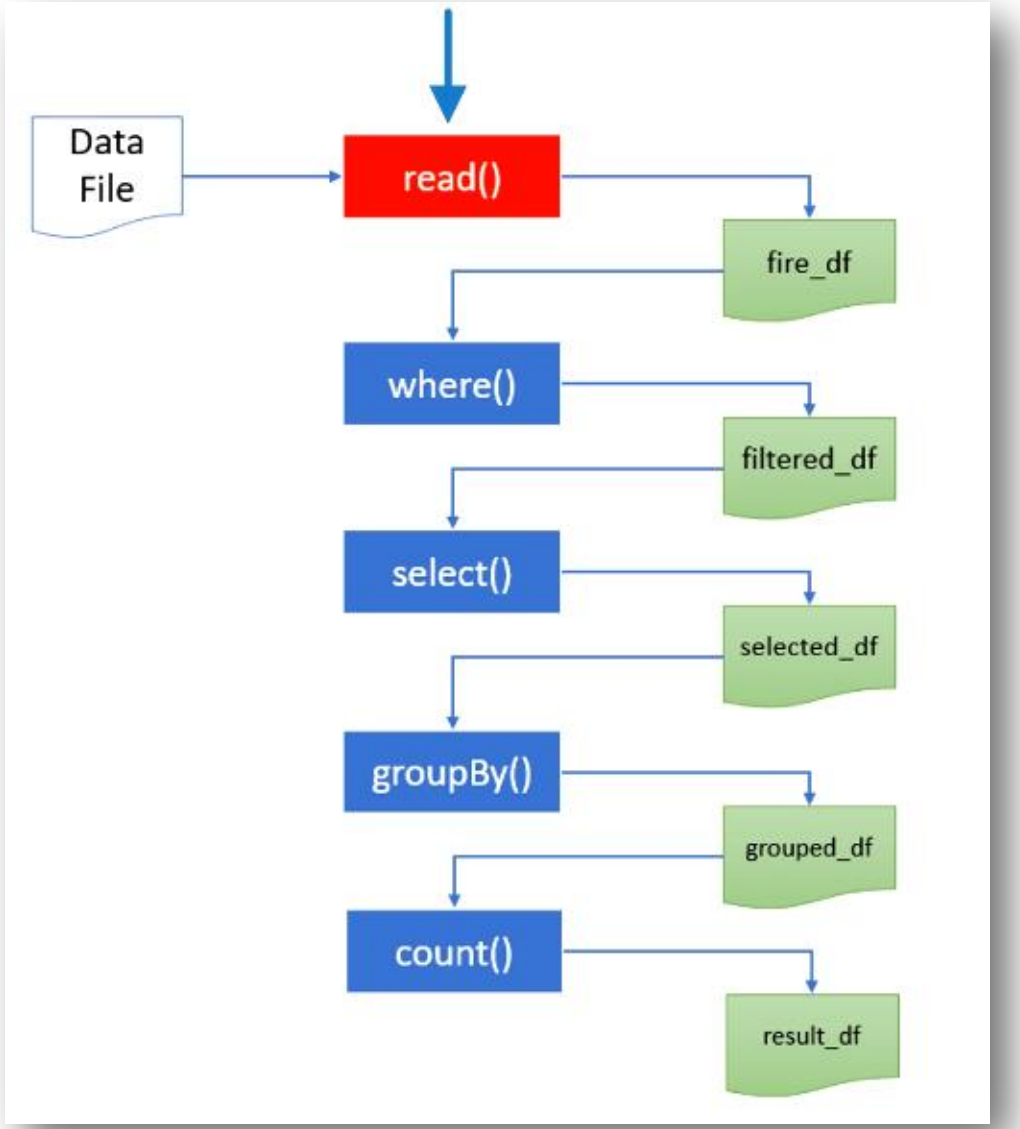
I want to highlight two things here:

1. I am not changing any Dataframe. I am simply taking some records from an existing Dataframe and transforming them to create a new Dataframe.
2. Look at all these transformations carefully. You will realize that we are creating a graph of the operations.

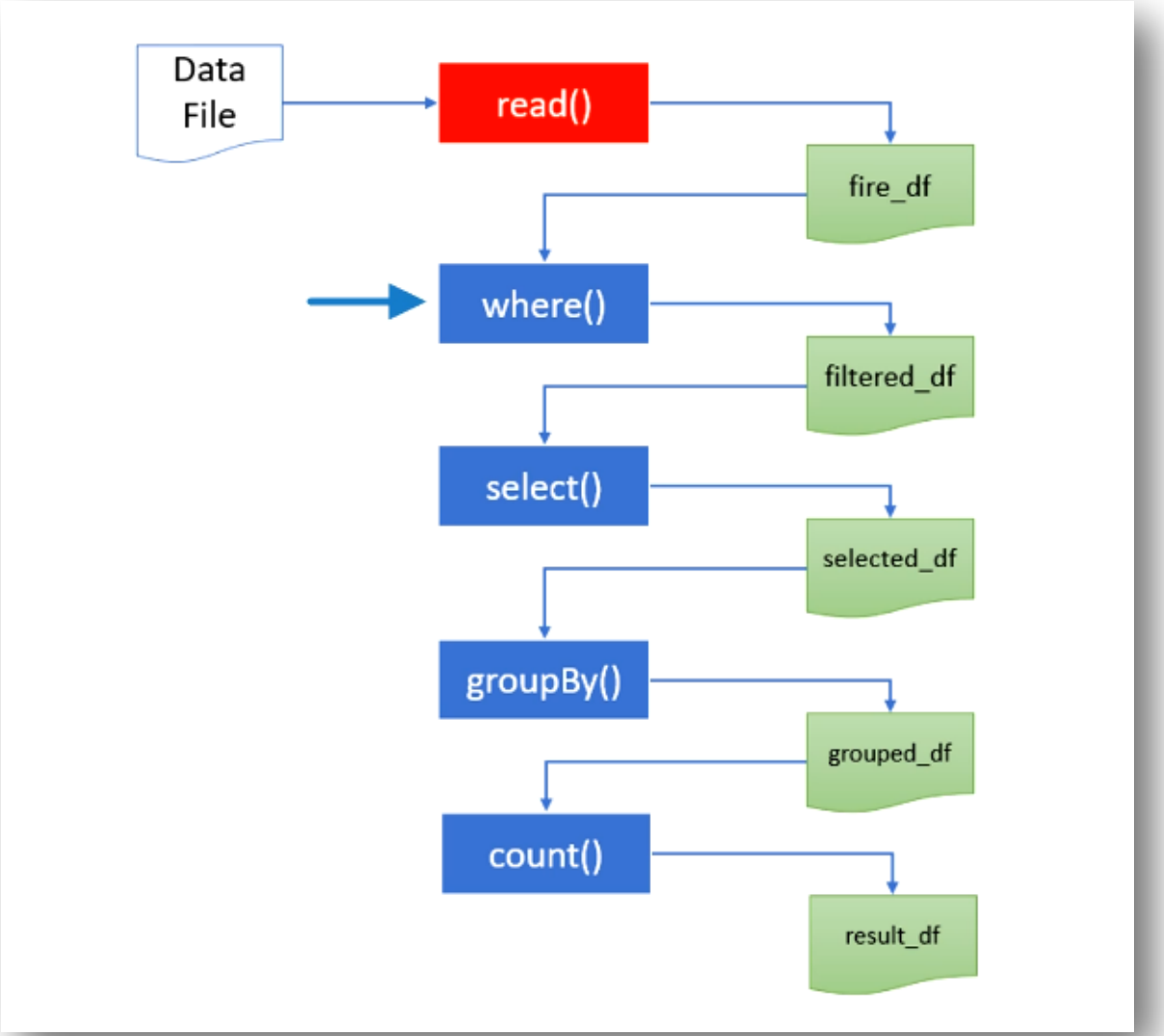
So the list of transformations shown in the previous slide might look like the following graph.



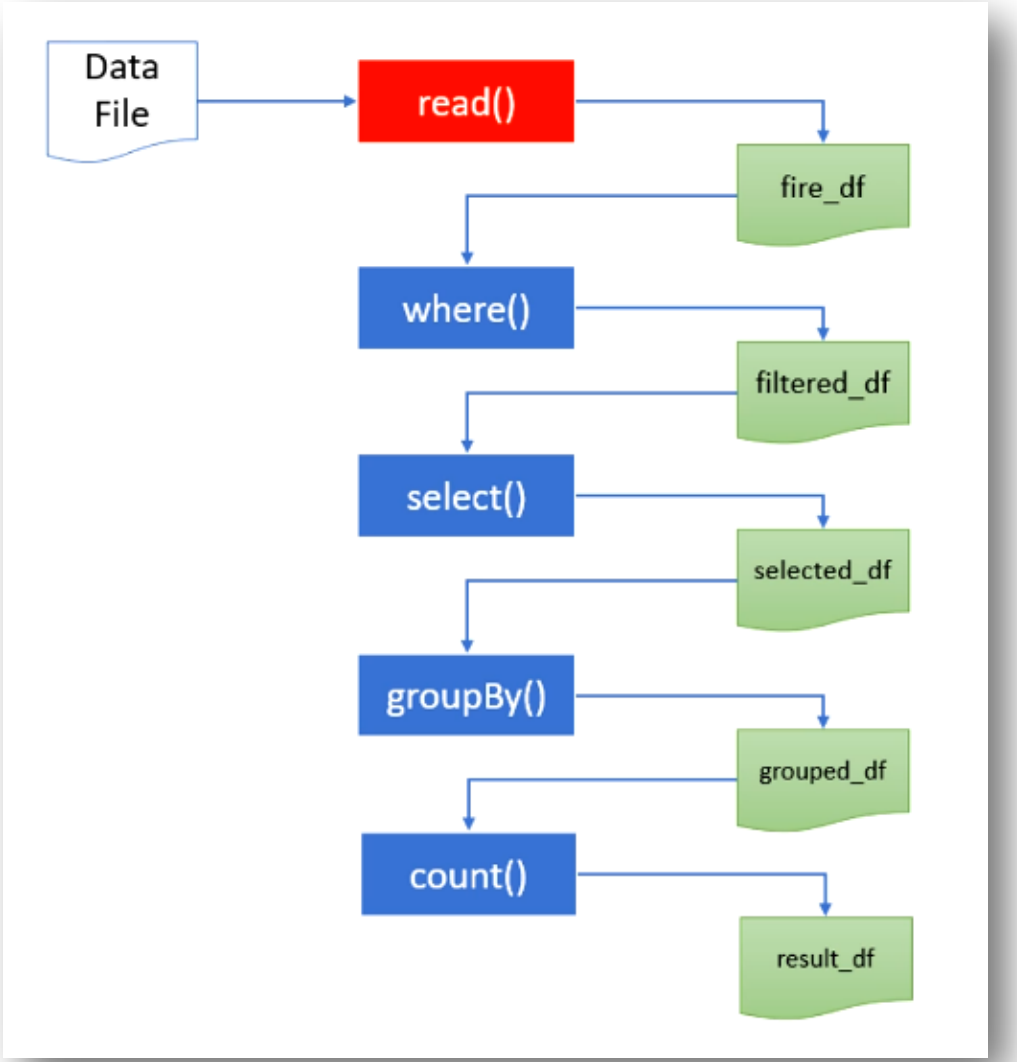
We start with the read() operation. The input to the read operation is a data file. And the output of the read() is the *fire_df* Dataframe.



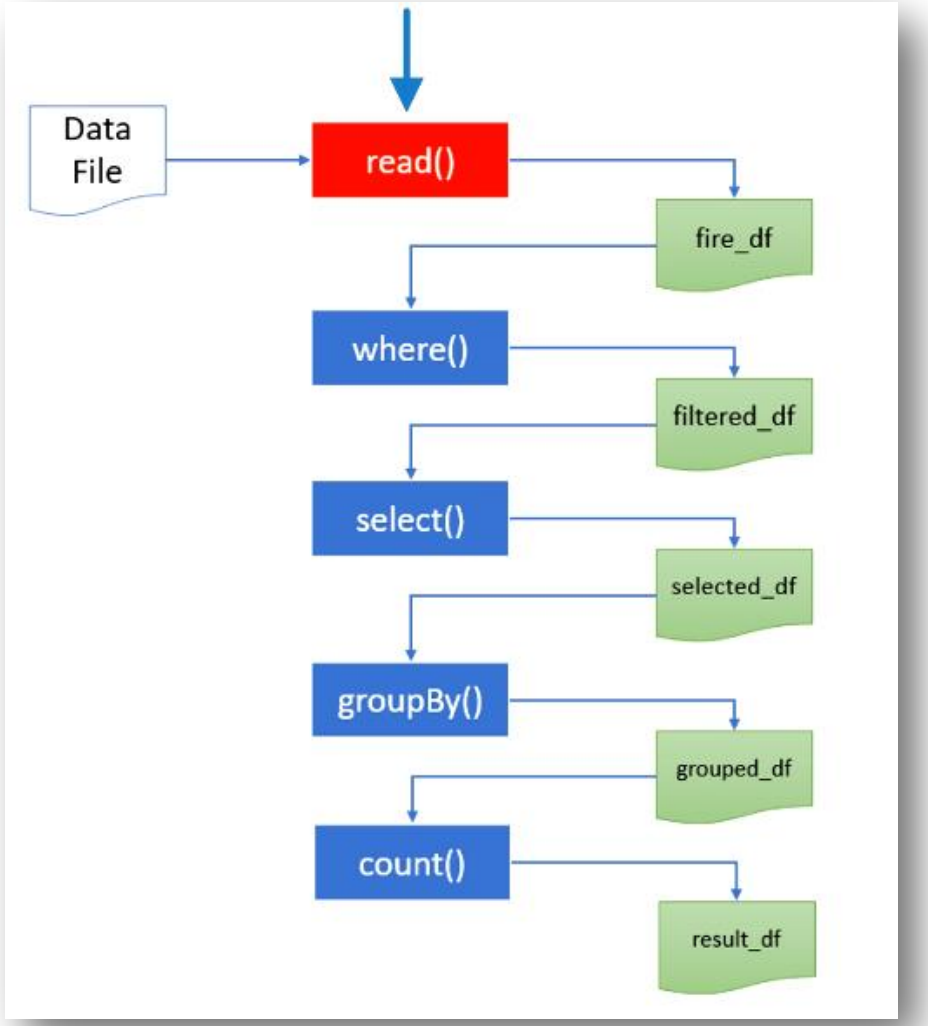
The next step in this graph is to use the where() method. The input for the where() method is the *fire_df*, and the output is the *filtered_df*. Similarly, each operation in this directed acyclic graph takes an input Dataframe and produces an output Dataframe.



All operations that take input Dataframe and produce output Dataframe are known as Dataframe transformations. The chain of transformations can be shown in a directed acyclic graph, and we call it Spark transformation DAG. The DAG is nothing but a visual representation of the sequence of operations.



The read() method in this DAG is not a transformation. Because it does not take Dataframe input. Instead, it takes file input to create a Dataframe. A Spark transformation reads a Dataframe and generates a new Dataframe. The read() method does not meet that condition, so it is not a transformation. However, everything else shown here is a transformation.



I wrote all these transformations one at a time creating intermediate data frames. However, you can chain all these transformations one after the other without explicitly creating intermediate data frames, as highlighted in the image below.

Cmd 2

```
result_df = fire_df.where("CallType = 'Alarms'") \
    .select("CallType", "City", "Priority", "Delay") \
    .groupBy("Priority") \
    .count()
```

Cmd 3

```
filtered_df = fire_df.where("CallType = 'Alarms'")
```

Cmd 4

```
selected_df = filtered_df.select("CallType", "City", "Priority", "Delay")
```

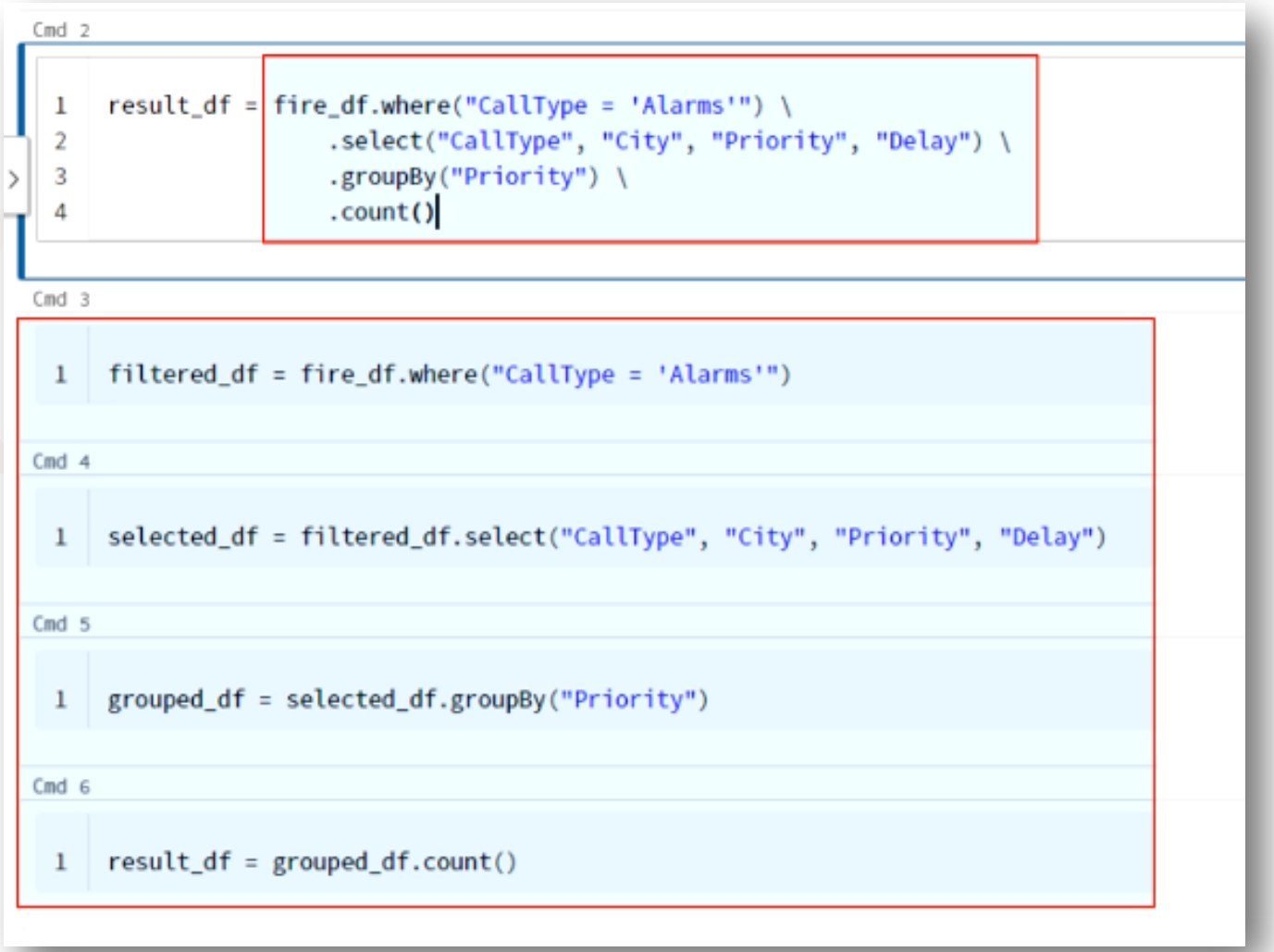
Cmd 5

```
grouped_df = selected_df.groupBy("Priority")
```

Cmd 6

```
result_df = grouped_df.count()
```


I took all the individual transformations and chained them one after other. Both the approach of highlighted below are the same. The DAG of this chain remains the same as the earlier code. We do not see intermediate data frames in this approach, but they are hidden behind the scenes. So I can delete all individual transformations and keep the chained transformations only because we do not need them.



I have a chain of Spark transformations shown in the second cell below. And here is the DAG of the transformation chain shown below. The DAG is nothing but a visual representation of the Spark transformation chain. Now, we can run this notebook once.

44-transformation-internals Python

demo-cluster

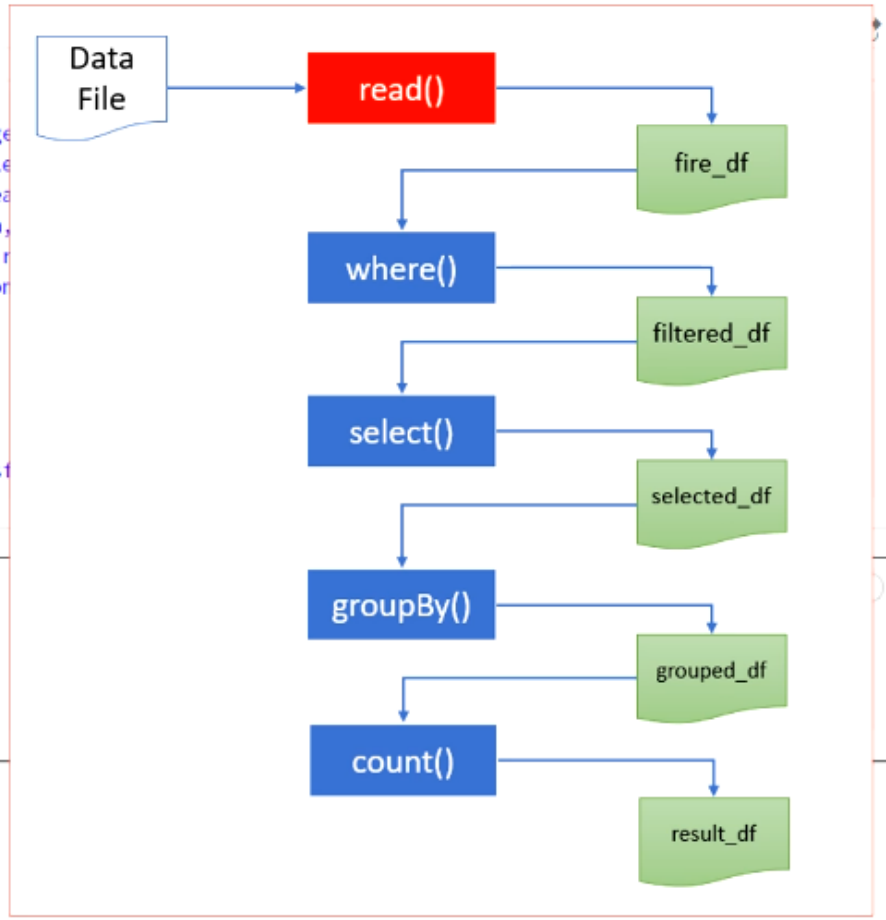
Cmd 1

```
schema = """CallNumber integer, UnitID string, IncidentNumber intege
            WatchDate string, CallFinalDisposition string, Available
            ZipcodeofIncident integer, Battalion string, StationArea
            Priority string, FinalPriority integer, ALSUnit boolean,
            UnitType string, Unitsequenceincalldispatch integer, Fir
            SupervisorDistrict string, Neighborhood string, Location

fire_df = spark.read \
            .format("csv") \
            .option("header", "true") \
            .schema(schema) \
            .load("/databricks-datasets/learning-spark-v2/sf-fire/sf
```

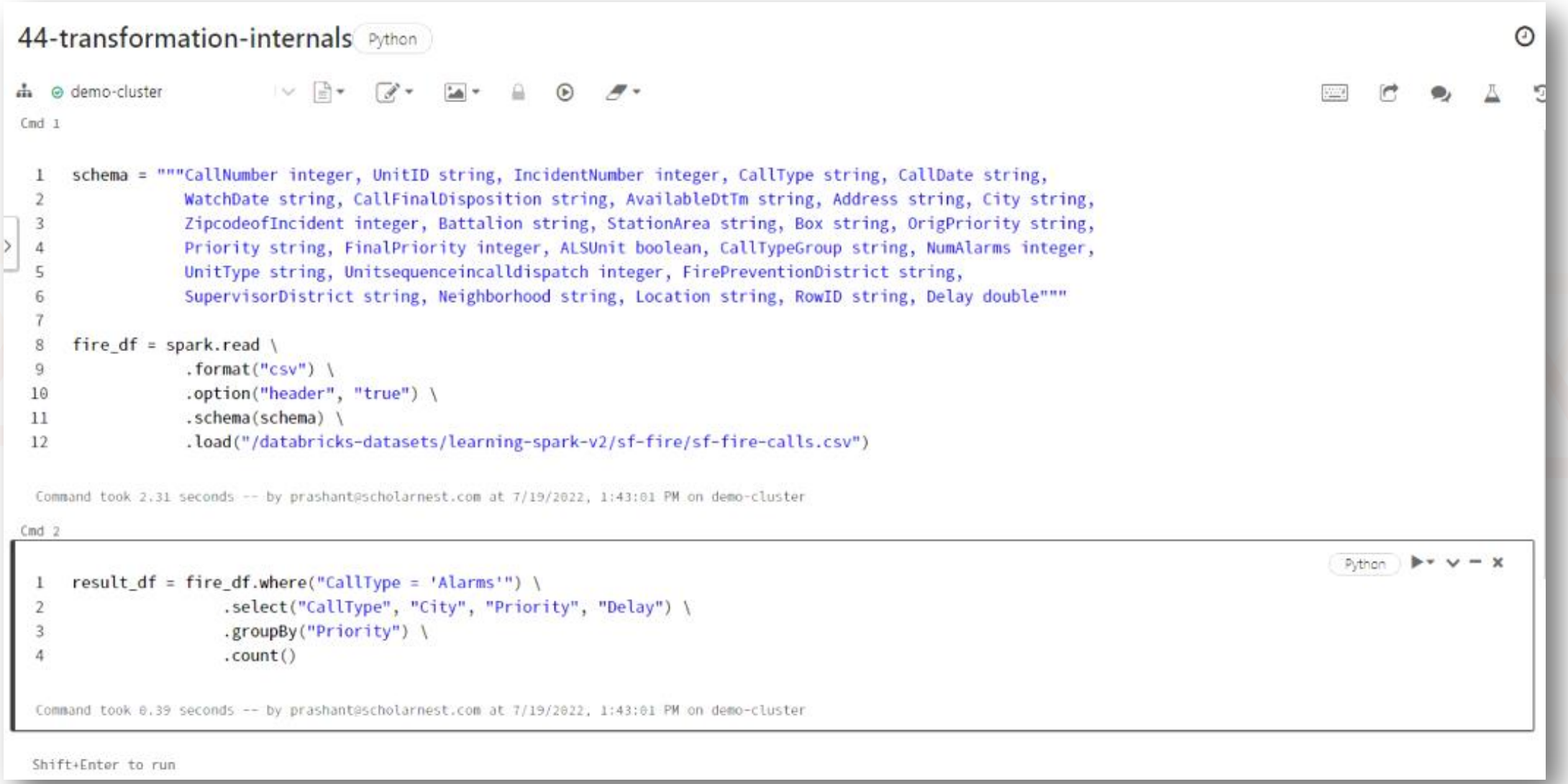
Cmd 2

```
result_df = fire_df.where("CallType = 'Alarms'") \
                .select("CallType", "City", "Priority", "Delay") \
                .groupBy("Priority") \
                .count()
```

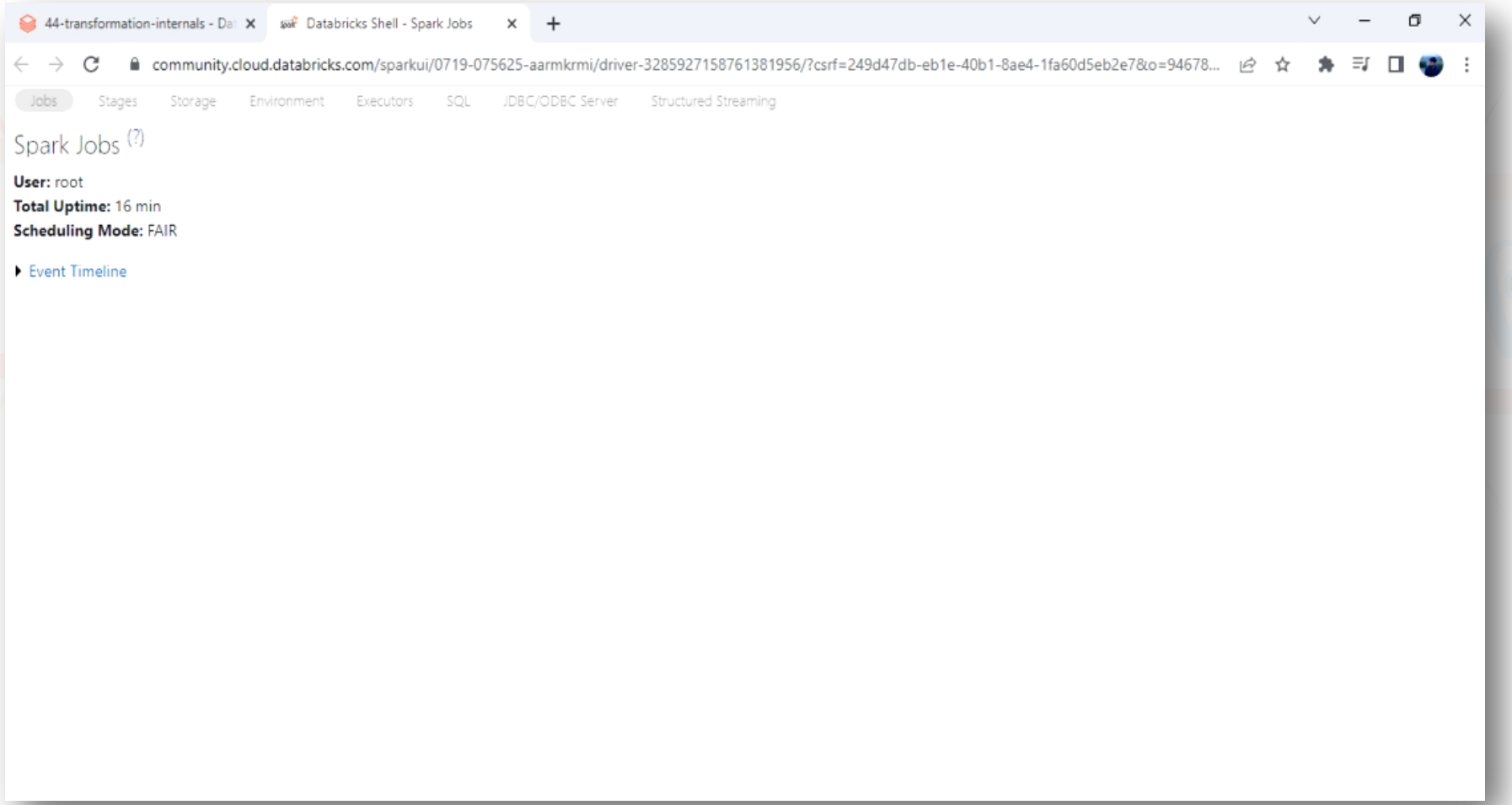
Shift+Enter to run



We ran our notebook, but we don’t see any output because we do not have code to display anything.

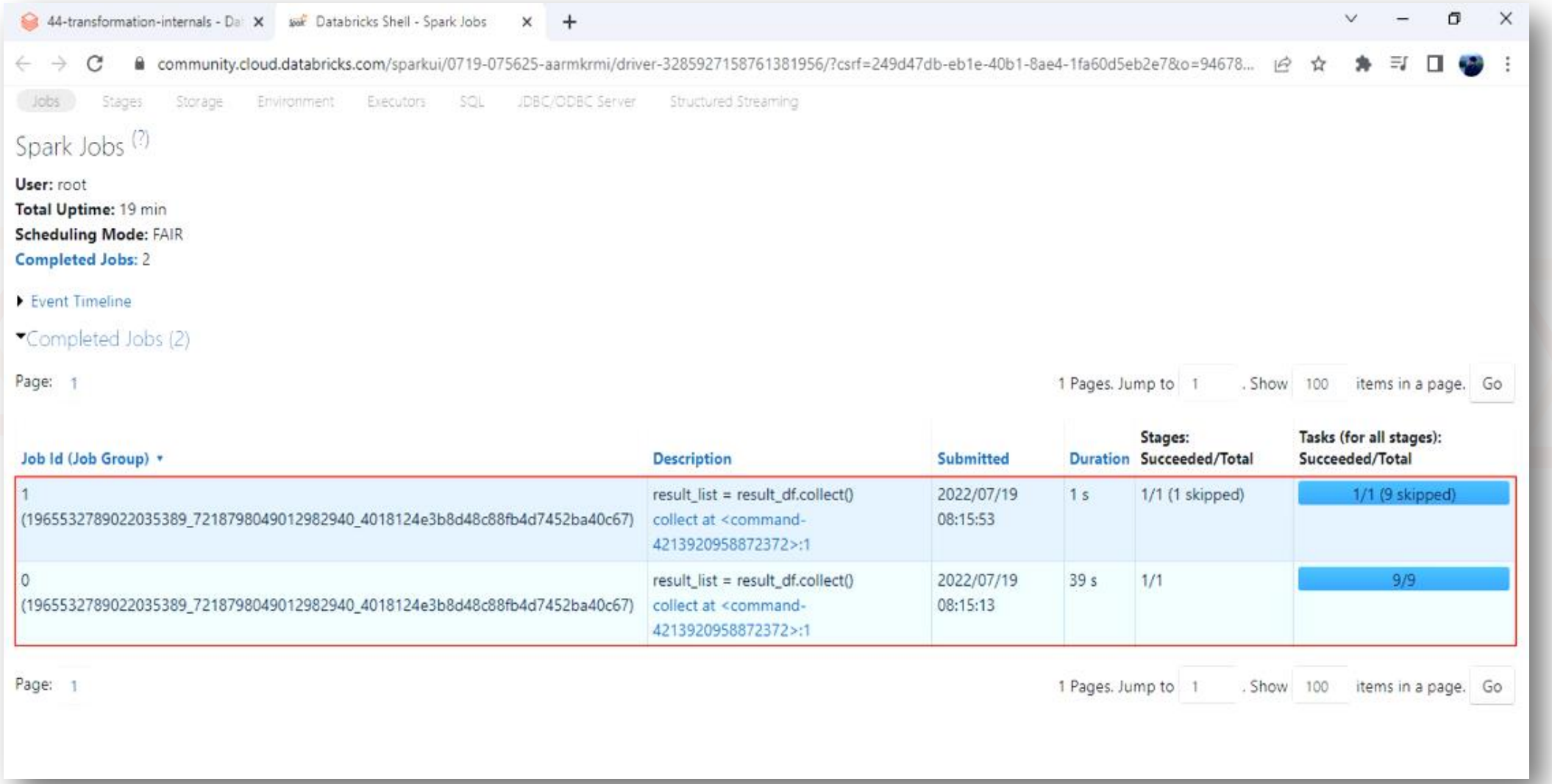


Now come to the Spark UI.
You will see that the page is blank. I am here in the Jobs tab. But I do not see anything here. I ran the notebook, and it executed successfully, but I do not see anything here on the Spark UI. Why? Because Spark transformations are lazy. It means, my code will compile and get ready to execute. But it will not execute until I take action. That's why we do not see anything here.



Now go back to your notebook and take an action as depicted below.
I have a sequence of transformations defined above. I start with the where, then do a select(), groupBy() and count(). But all of these are transformations. The transformations are lazy, so they do not run until we take action.
I am collecting the records from the *result_df* into a Python list. And that's an action. The collect() method is an action. We have many other actions and you already learned actions like show(), summary(), describe() and take() etc.

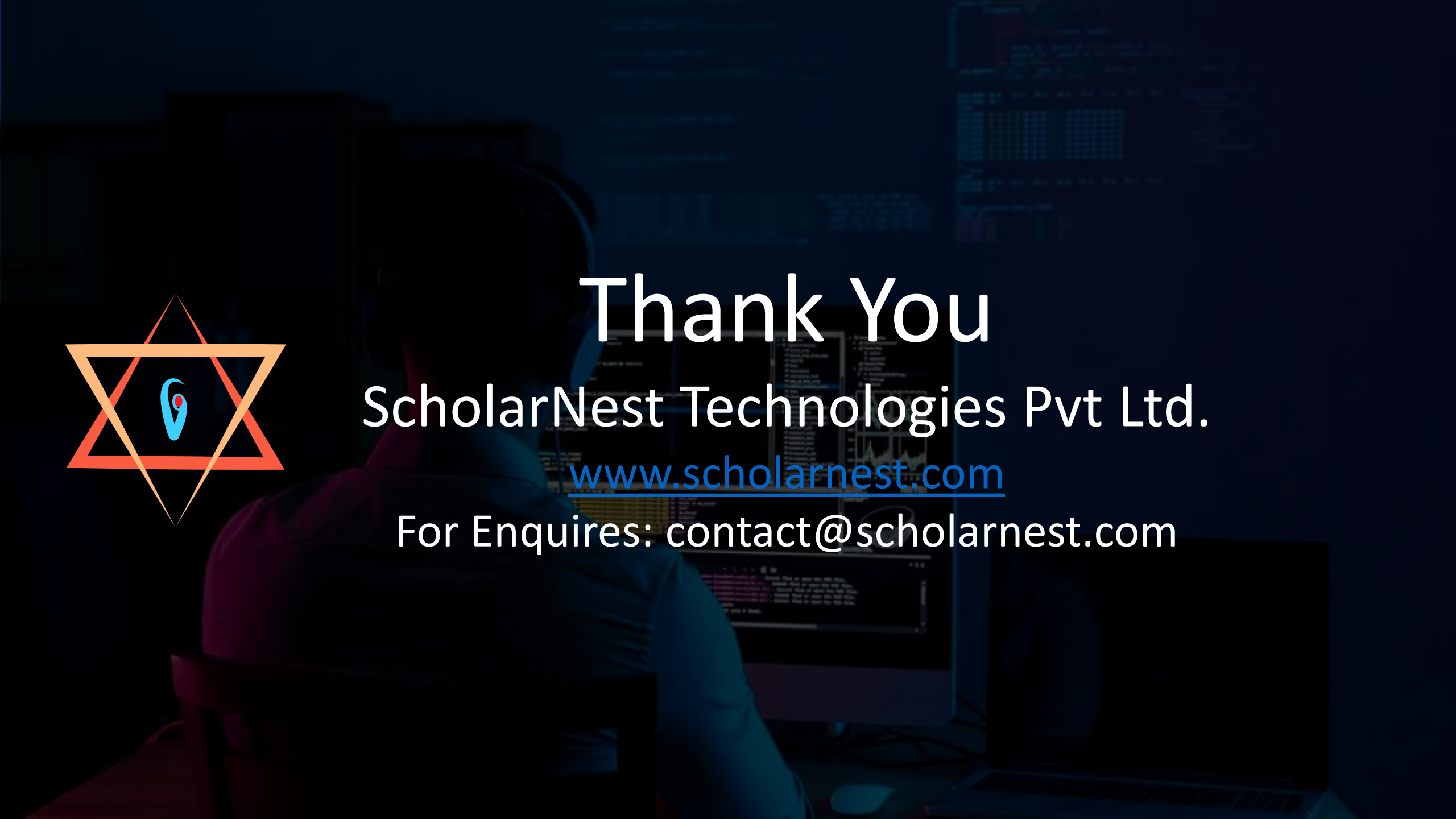
Now you can go back to your Spark UI after executing the action, and you will see that we got some jobs here.



So, Spark Dataframe offers your two types of operations:

1. Transformations
2. Actions

Transformations are lazy, and we use them to transform one Dataframe into another Dataframe. You can chain transformations one after the other to create a DAG of operations. You can terminate your DAG of transformations using an action.
Transformations are lazy, so they will not run until you take action. And that is why we need action at the end of the transformation DAG.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Dataframe Internals

Lecture:
Narrow and Wide Transformation

Narrow and Wide Transformations

Spark Transformation is classified into two types:

1. Narrow Dependency Transformation
2. Wide Dependency Transformation

Now let's try to understand it.

Here is a Dataframe code shown at the top in the screenshot.
I am reading a file and creating a fire_df Dataframe. And assume that you got the following Dataframe.

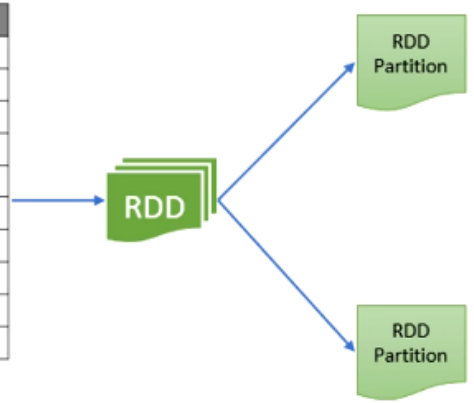
```
fire_df = spark.read \
          .format("csv") \
          .option("header", "true") \
          .schema(schema) \
          .load("/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv")
```

S.No	CallType	City	Priority	Delay
1	Alarm	SF	1	5
2	Alarm	SF	2	6
3	Alarm	SF	1	2
4	Alarm	SF	2	7
5	Medical Incident	SF	3	15
6	Alarm	SF	1	4
7	Alarm	SF	1	8
8	Medical Incident	SF	3	9
9	Alarm	SF	2	6
10	Alarm	SF	2	12

Now this fire_df Dataframe is internally made up of an RDD. So we have RDD adjacent to the Dataframe shown below. But the logical RDD is made up of some physical RDD Partitions. Let's assume my RDD is composed of two partitions.
I see ten records here in the Dataframe. But internally, these ten records are stored in the physical RDD partitions.

```
fire_df = spark.read \
          .format("csv") \
          .option("header", "true") \
          .schema(schema) \
          .load("/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv")
```

S.No	CallType	City	Priority	Delay
1	Alarm	SF	1	5
2	Alarm	SF	2	6
3	Alarm	SF	1	2
4	Alarm	SF	2	7
5	Medical Incident	SF	3	15
6	Alarm	SF	1	4
7	Alarm	SF	1	8
8	Medical Incident	SF	3	9
9	Alarm	SF	2	6
10	Alarm	SF	2	12



So we can assume each partition stores five records. And this is how a Dataframe is internally structured.

```
fire_df = spark.read \
          .format("csv") \
          .option("header", "true") \
          .schema(schema) \
          .load("/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv")
```

S.No	CallType	City	Priority	Delay
1	Alarm	SF	1	5
2	Alarm	SF	2	6
3	Alarm	SF	1	2
4	Alarm	SF	2	7
5	Medical Incident	SF	3	15
6	Alarm	SF	1	4
7	Alarm	SF	1	8
8	Medical Incident	SF	3	9
9	Alarm	SF	2	6
10	Alarm	SF	2	12

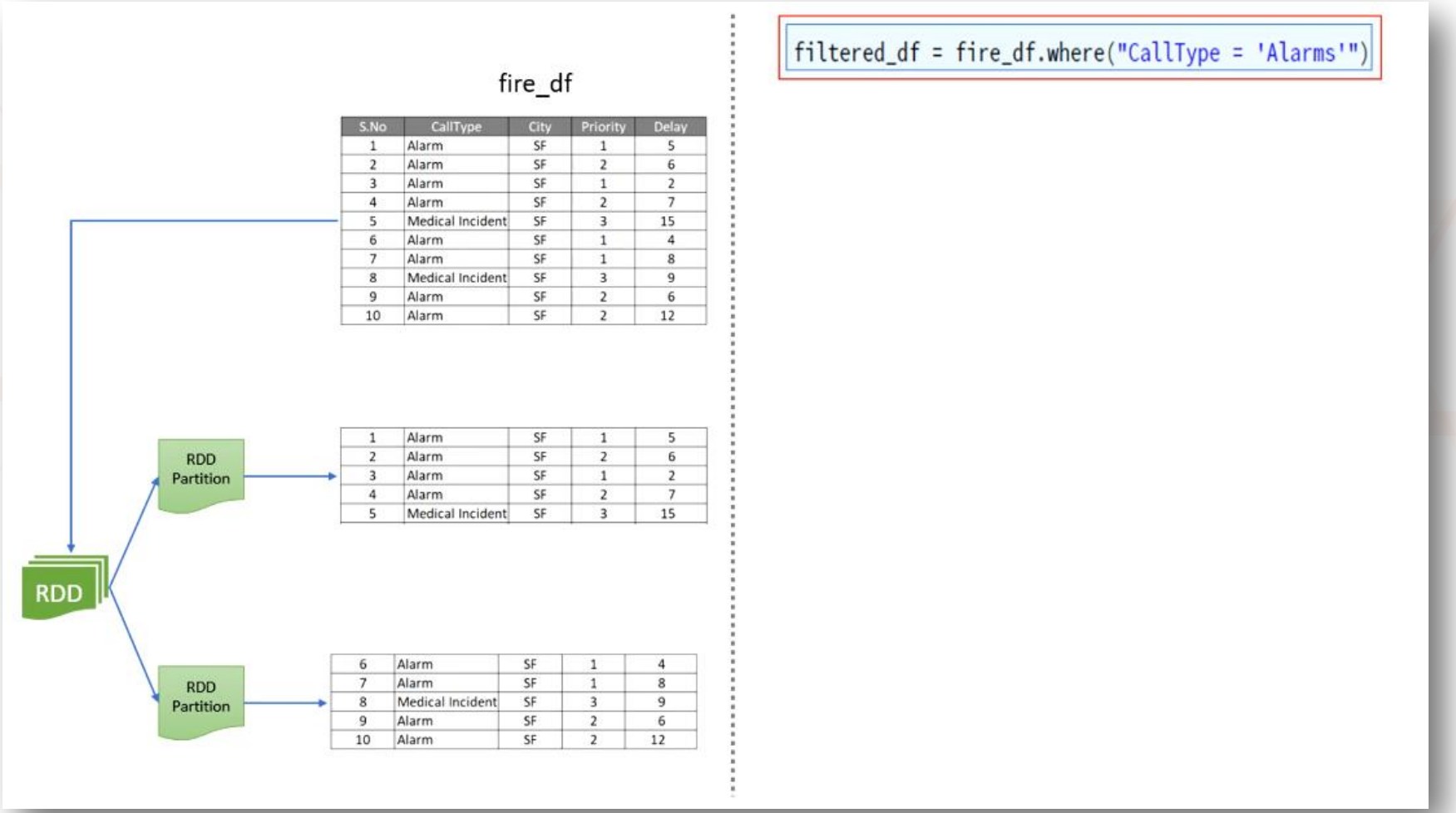
RDD Partition

1	Alarm	SF	1	5
2	Alarm	SF	2	6
3	Alarm	SF	1	2
4	Alarm	SF	2	7
5	Medical Incident	SF	3	15

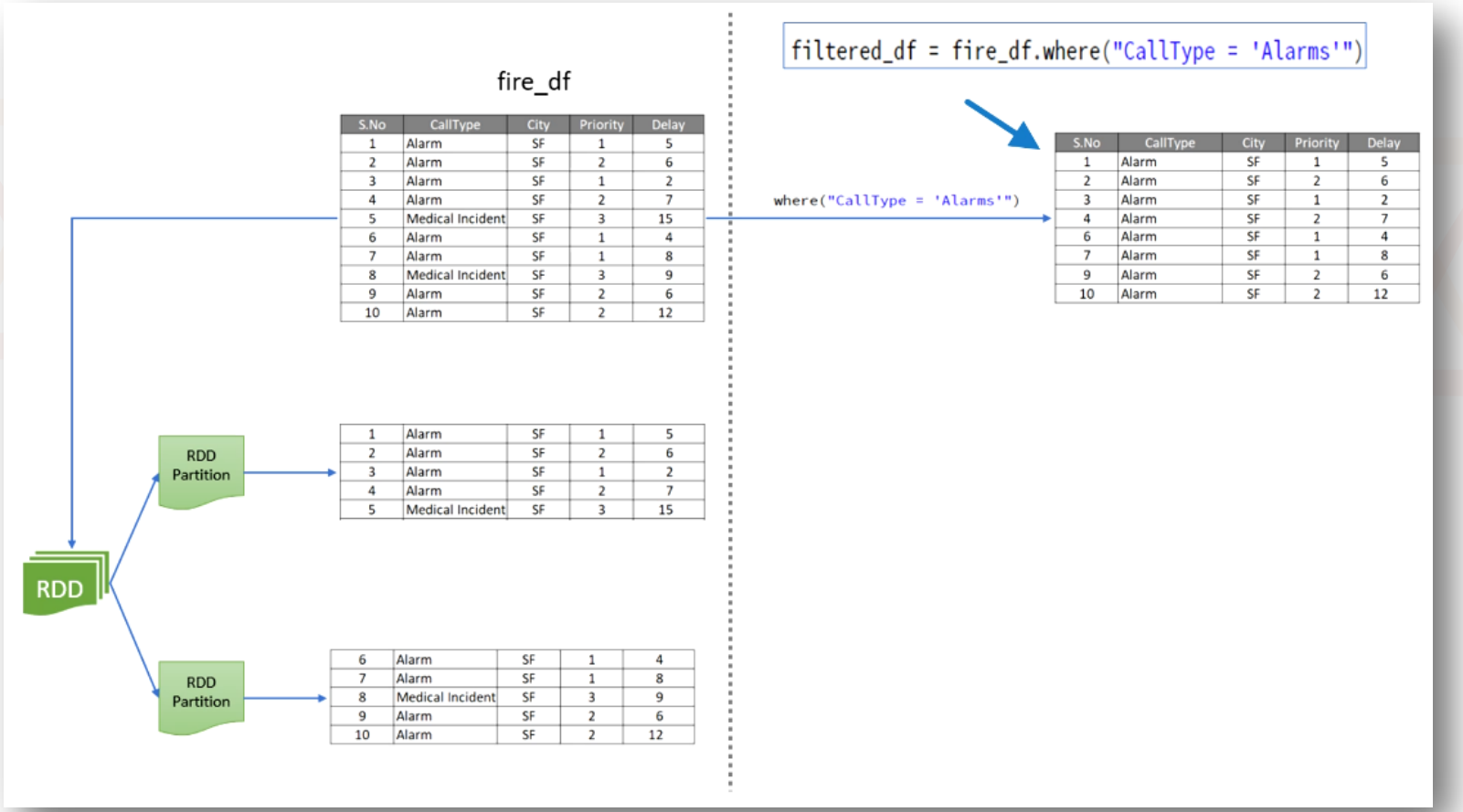
RDD Partition

6	Alarm	SF	1	4
7	Alarm	SF	1	8
8	Medical Incident	SF	3	9
9	Alarm	SF	2	6
10	Alarm	SF	2	12

Now let's apply some transformations to our given Dataframe. Here is the code highlighted below. The left side of the slide shows the Dataframe internals. It shows we have two partitions. The data is physically stored in these two partitions. We logically have one Dataframe but physically two partitions.



I applied the where() method to the fire_df Dataframe. So I want to take only those records where the call type is Alarm and create a new Dataframe filtered_df. You can see the logical view of filtered_df.
Spark should apply the where() method on the input Dataframe and produce a new Dataframe with only eight rows.

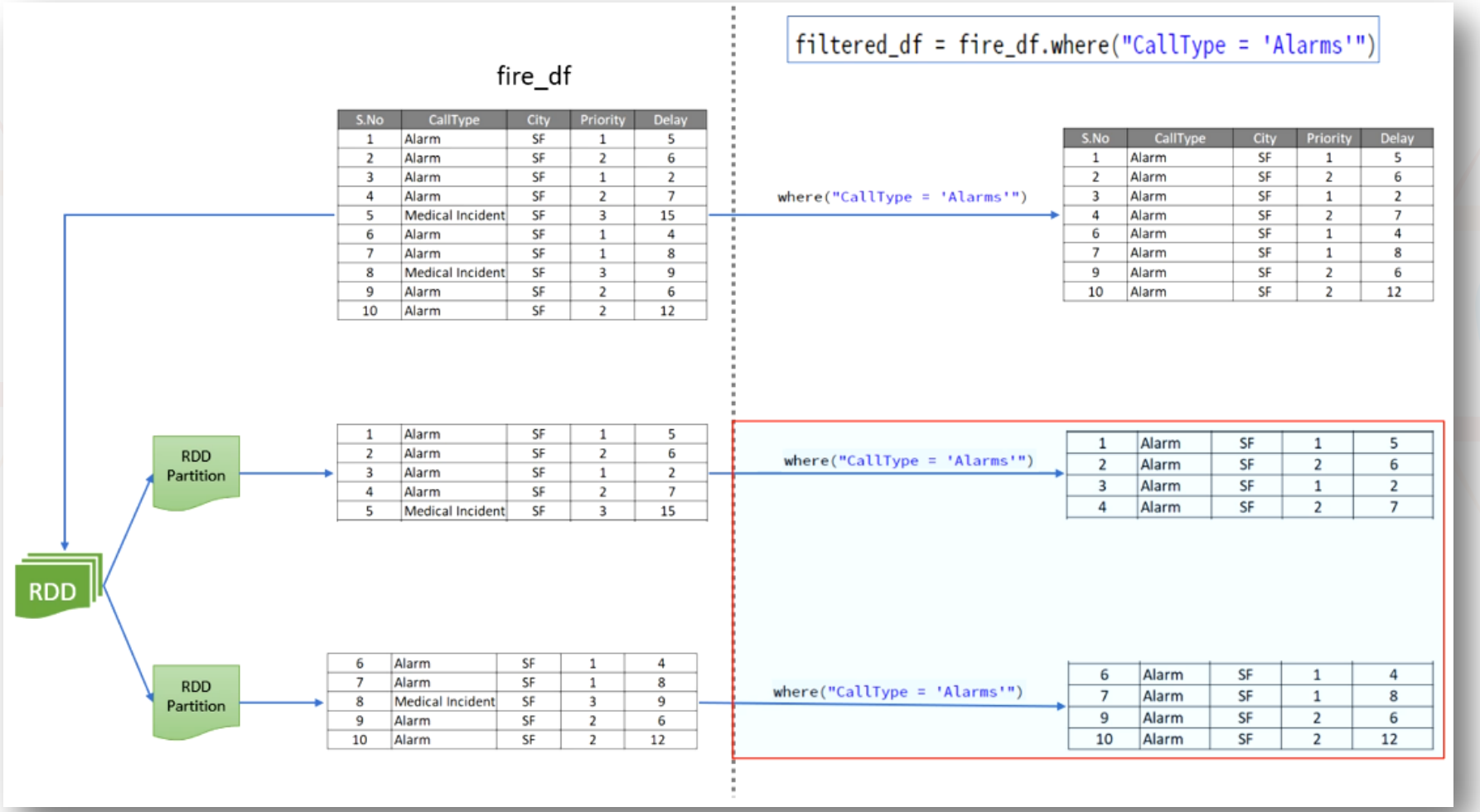


S.No	CallType	City	Priority	Delay
1	Alarm	SF	1	5
2	Alarm	SF	2	6
3	Alarm	SF	1	2
4	Alarm	SF	2	7
5	Medical Incident	SF	3	15
6	Alarm	SF	1	4
7	Alarm	SF	1	8
8	Medical Incident	SF	3	9
9	Alarm	SF	2	6
10	Alarm	SF	2	12

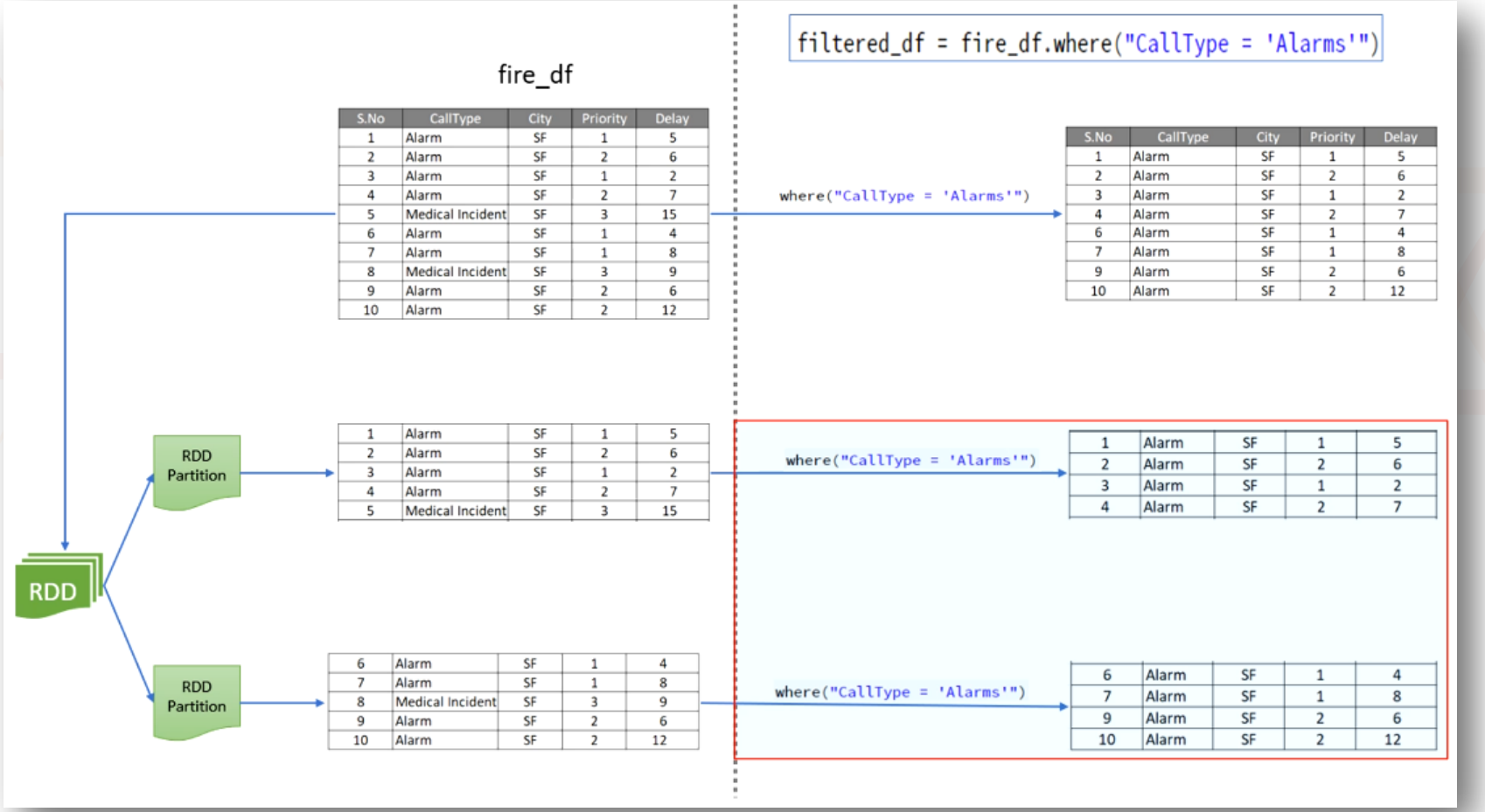
S.No	CallType	City	Priority	Delay
1	Alarm	SF	1	5
2	Alarm	SF	2	6
3	Alarm	SF	1	2
4	Alarm	SF	2	7
5	Medical Incident	SF	3	15

6	Alarm	SF	1	4
7	Alarm	SF	1	8
8	Medical Incident	SF	3	9
9	Alarm	SF	2	6
10	Alarm	SF	2	12

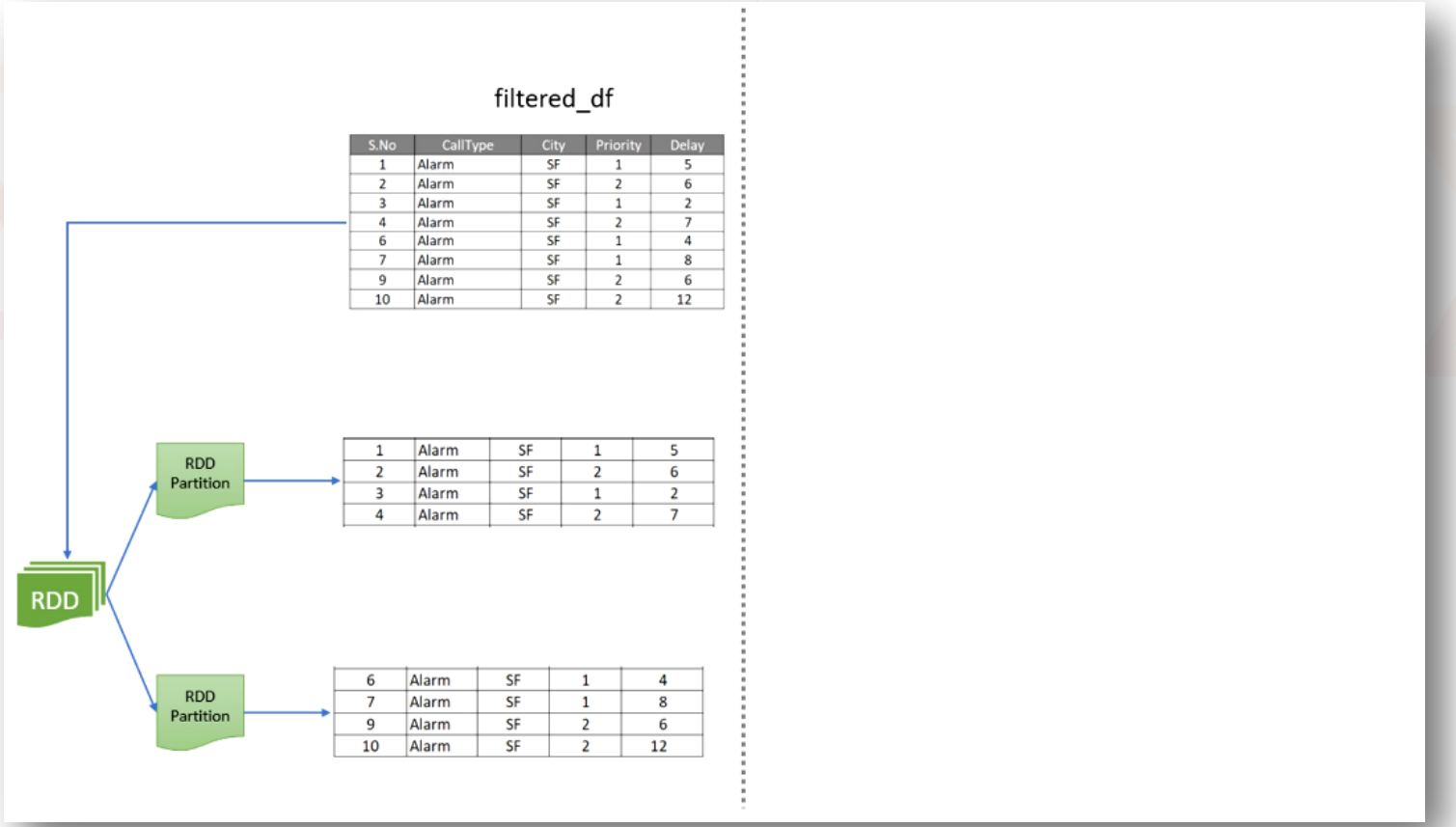
What we saw in the previous slide was the logical representation, but what happens physically? We have two physical partitions. But Spark can do it super fast. Spark will look at the first partition and take only those records where the call type is Alarm. And do the same for the second partition.



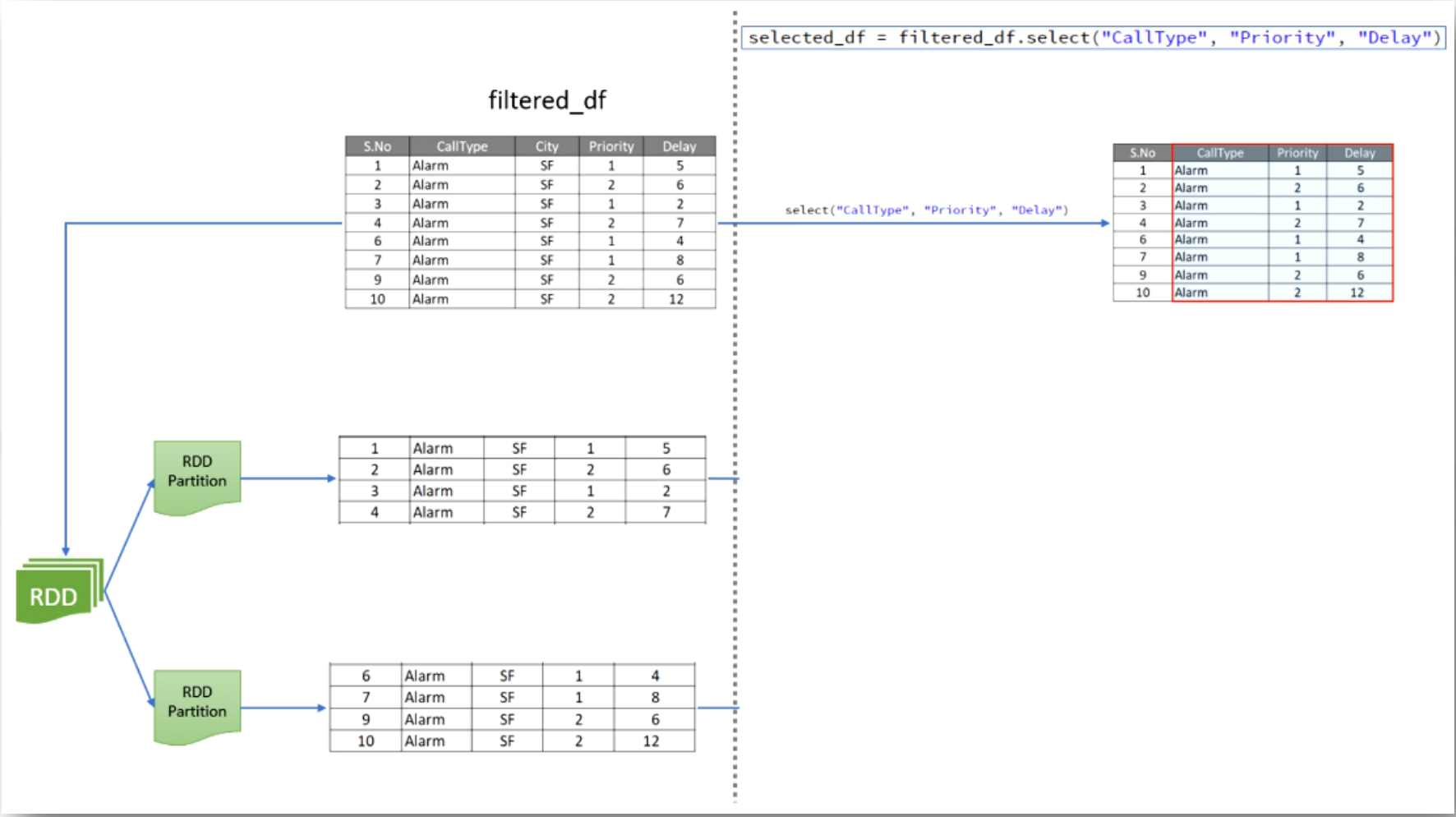
These types of transformations are narrow dependency transformations. Why? Because the where() method does not have a cross partition dependency. Spark can easily apply the where() method on Dataframe partitions and create a new Dataframe. The where() method does not require combining the two partitions. So all the transformations that do not depend on combining partitions are known as narrow dependency transformations.



Now let's see one more transformation. This time, I will start with the filtered_df and apply the next transformation.
So here is my filtered_df. I still have two partitions. I started with fire_df having two partitions.
I applied the narrow dependency where() method to filter records in each partition and created a new filtered_df. So I still have two partitions. Remember that! The narrow dependency transformation does not increase or alter the partition structure of your RDD.



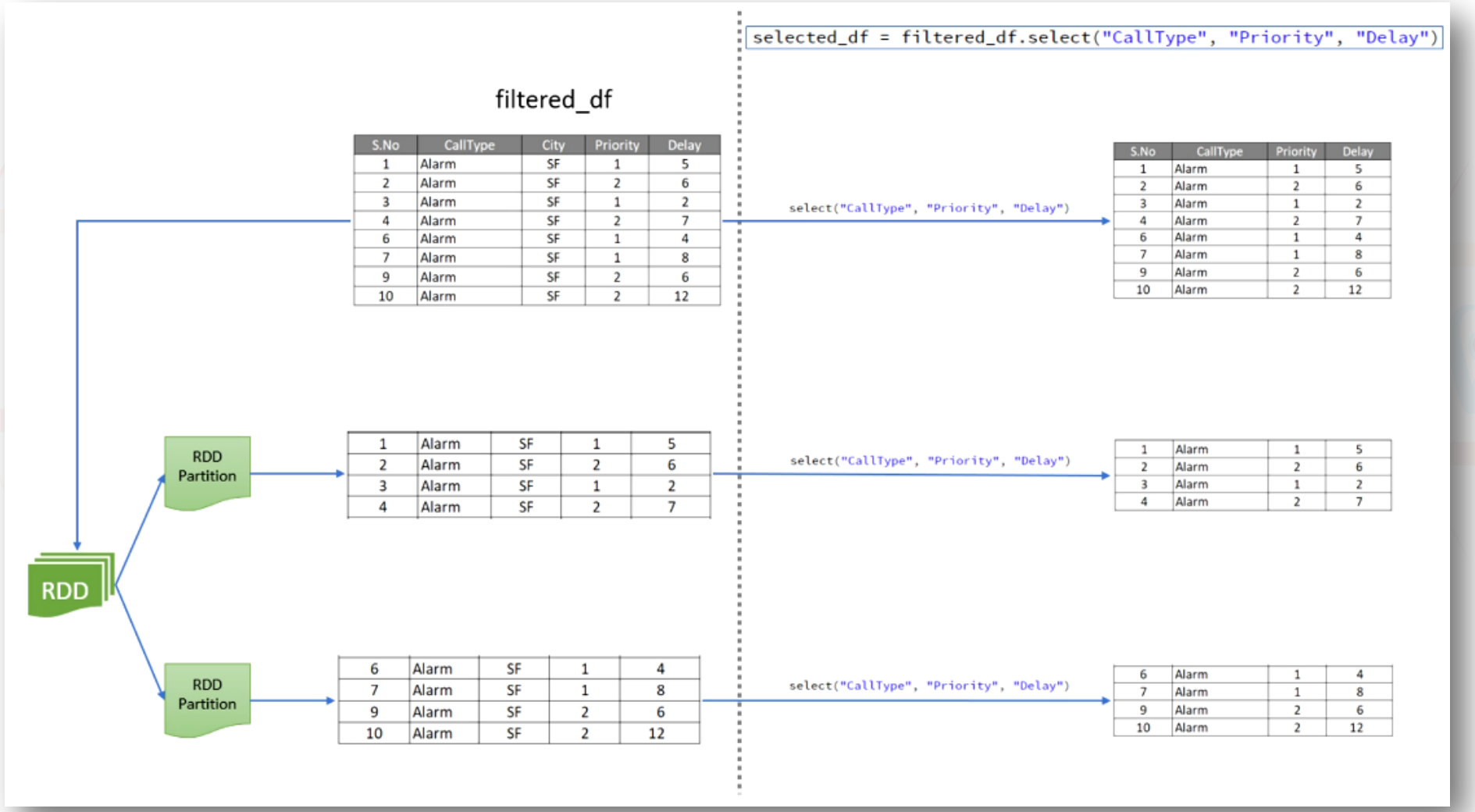
Now, I am applying the select method. And you can see the logical result. We had four columns in filtered_df, and now we need only three columns in selected_df.



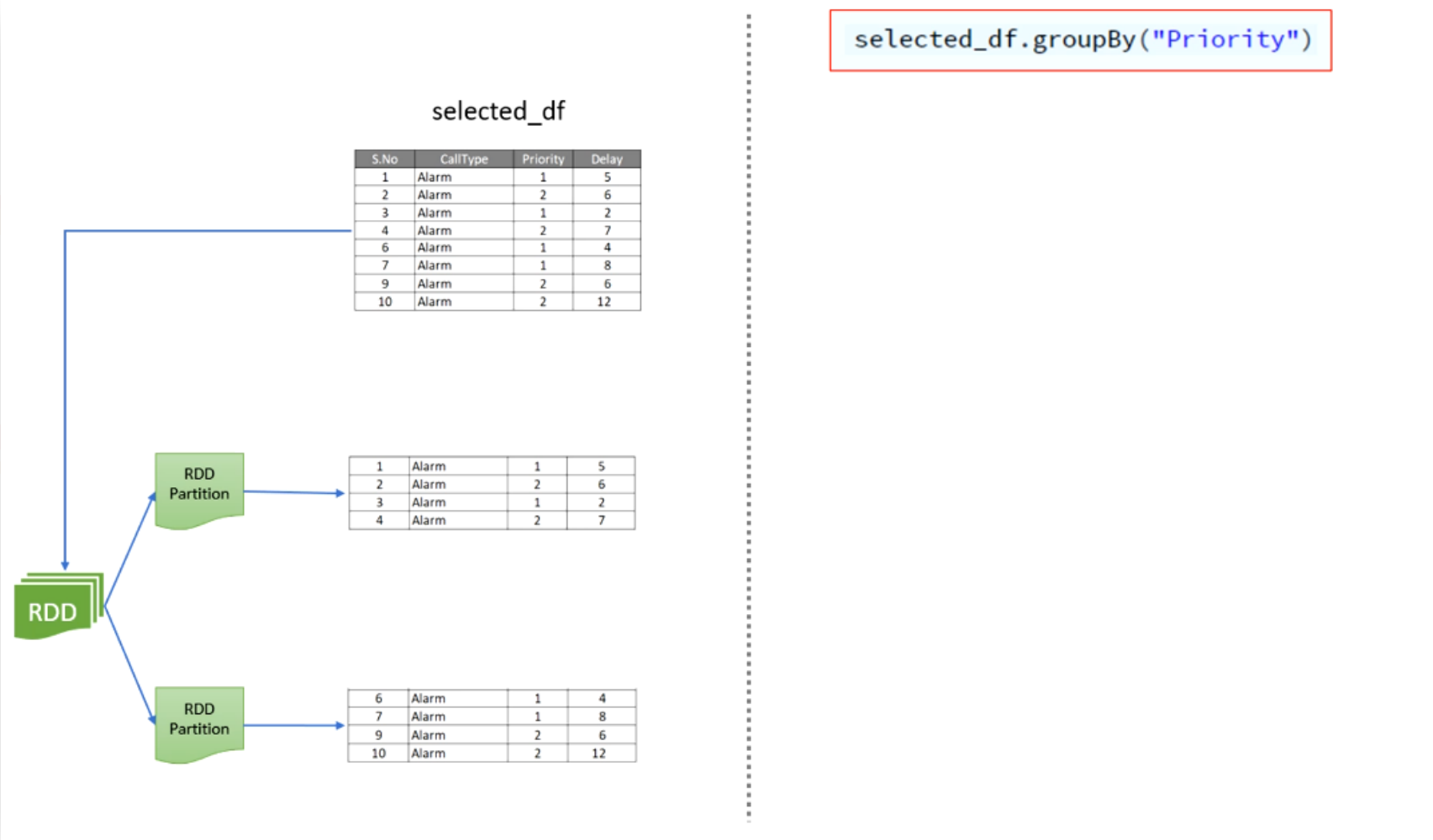
Here is the physical output.

Is it a narrow dependency transformation? Yes. It is.

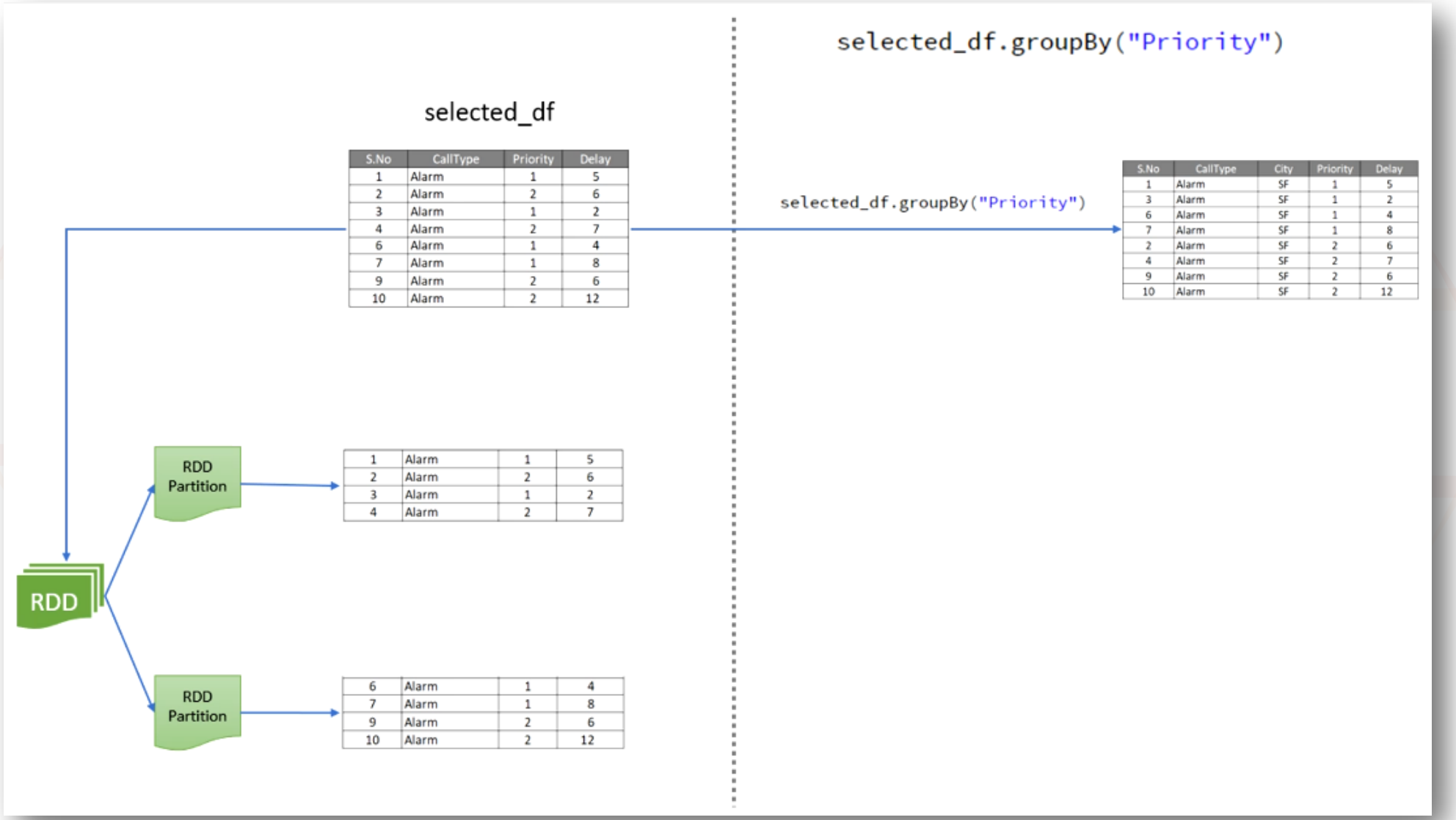
Why? Because we can apply it on each partition without requiring to combine the partitions.



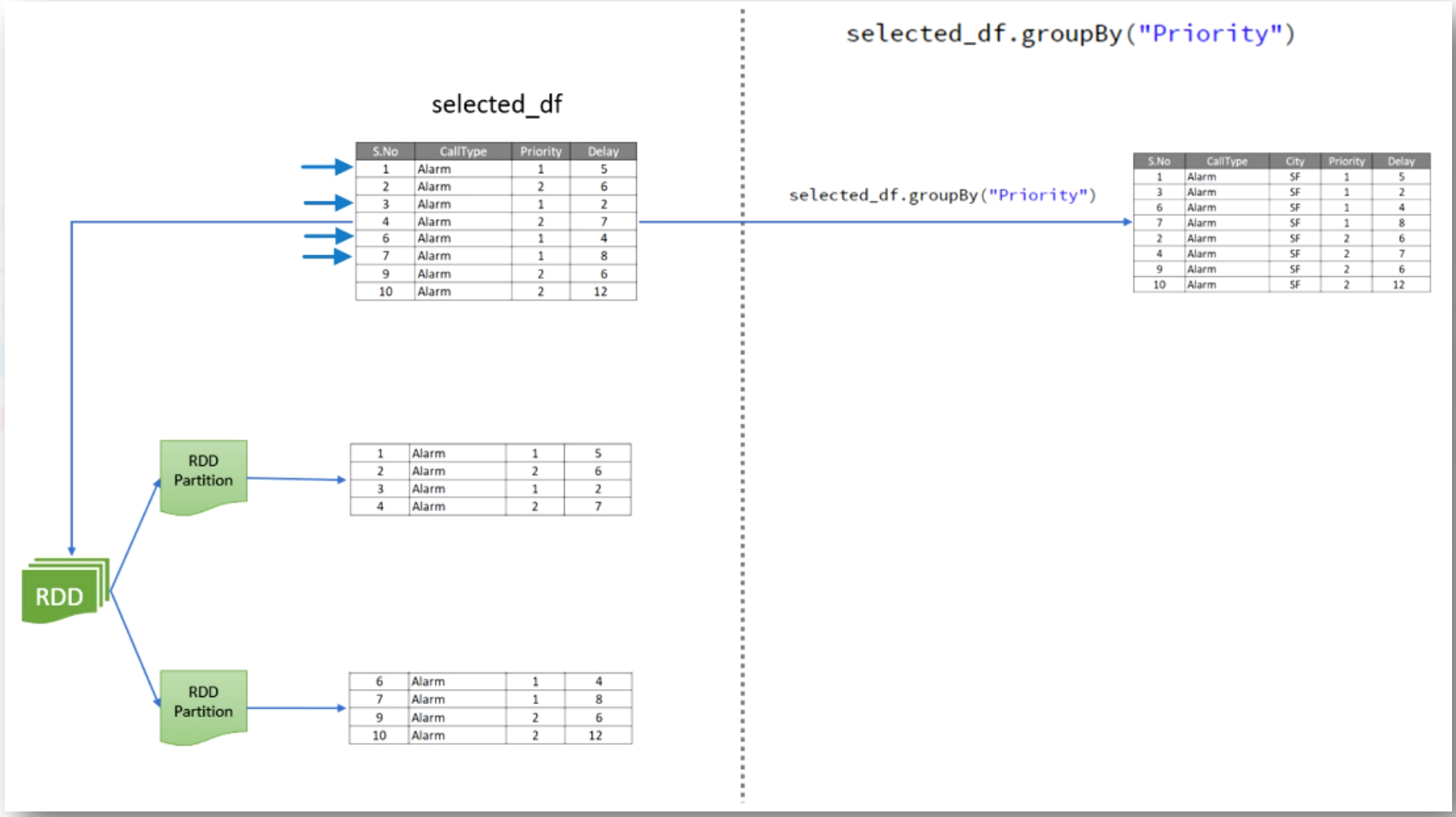
So the select() method is also a narrow transformation. Now, let's do one more on top of the selected_df.
So this time, I want to apply the groupBy() transformation.



Here is the logical result of the operation.

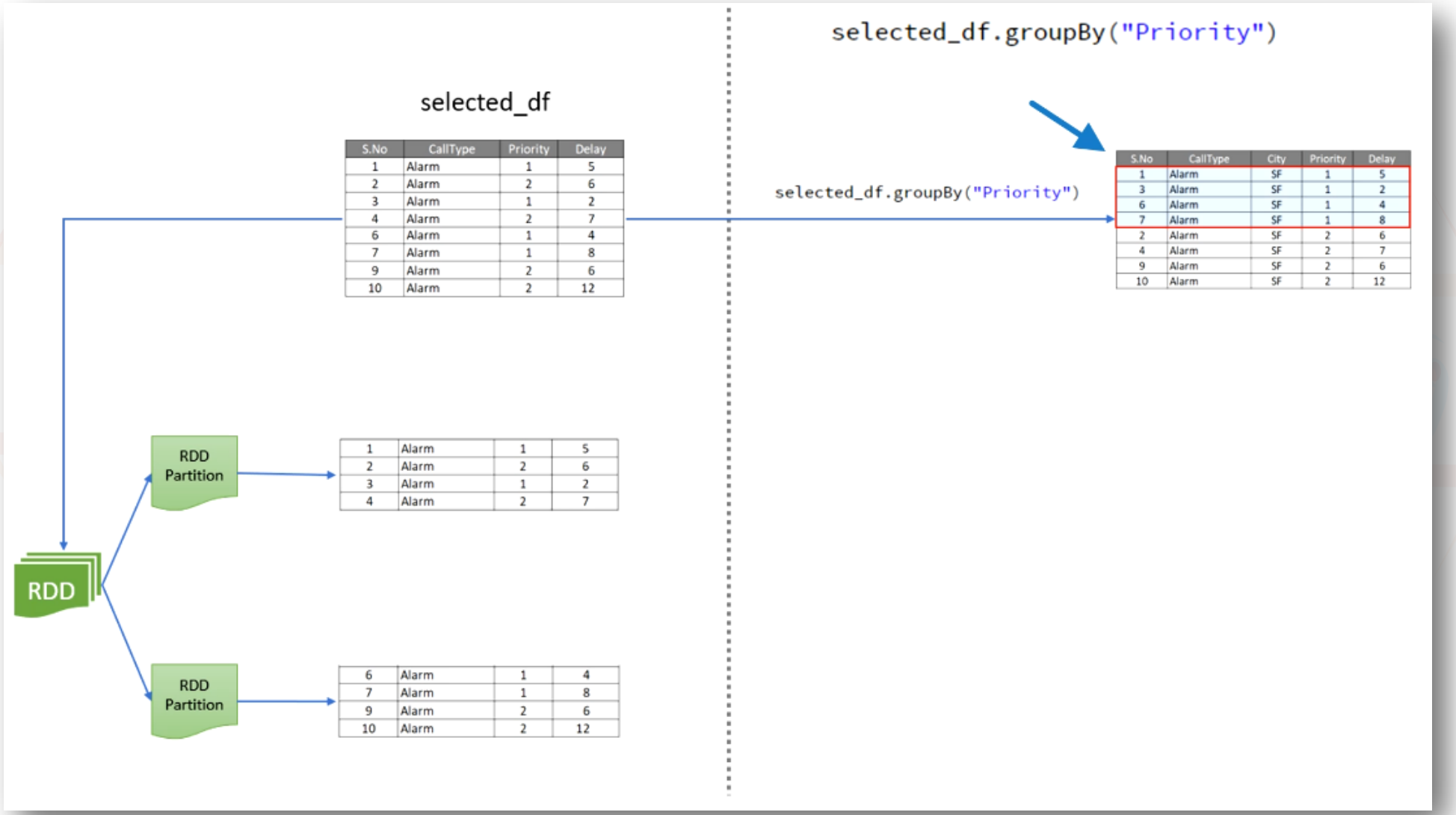


The groupBy("Priority") brings records of the same priority together.
Look at the input Dataframe. Record 1, 3, 6, and 7 are priority one records. Rest all are priority two records.

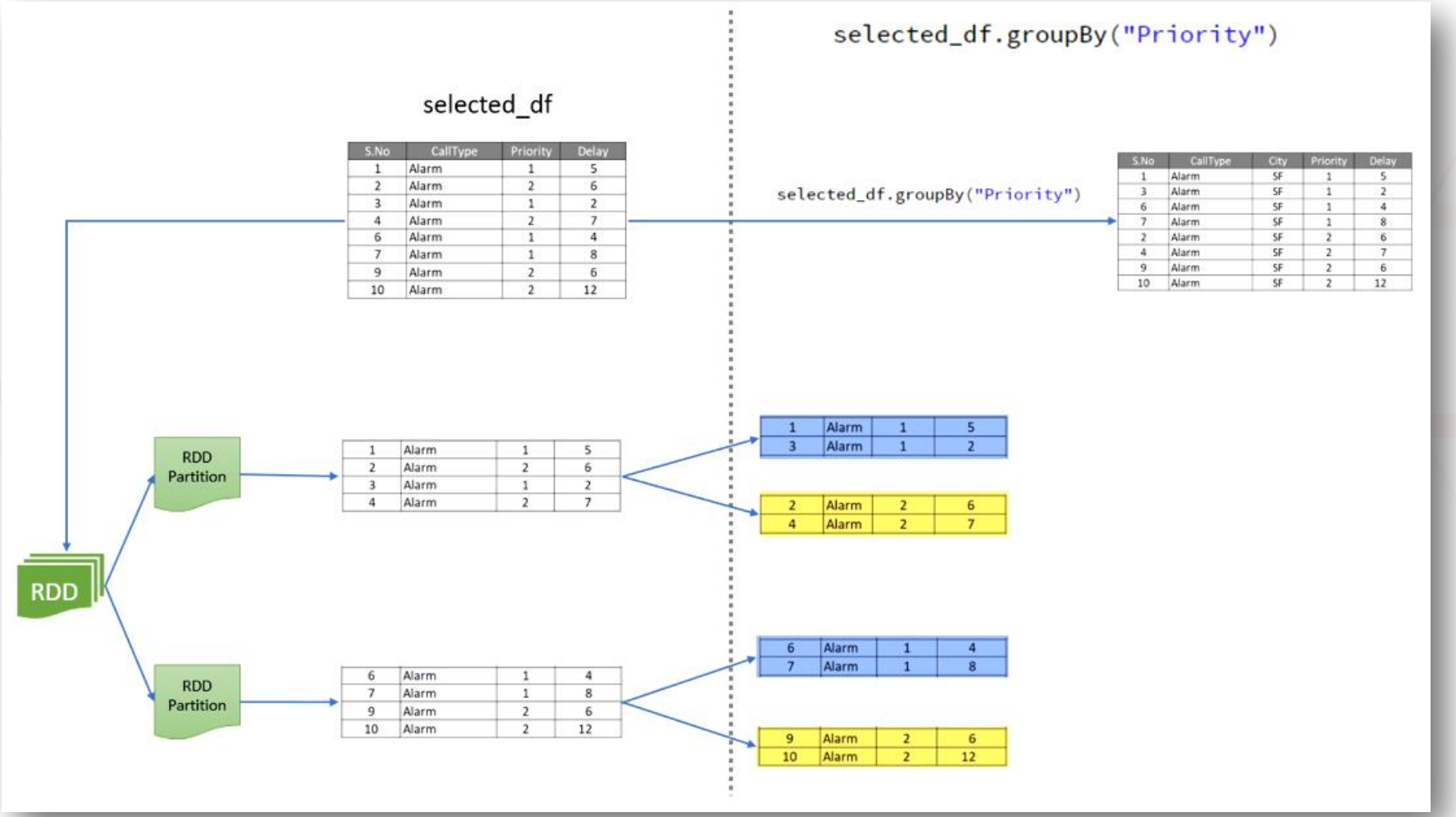


Now, look at the result.

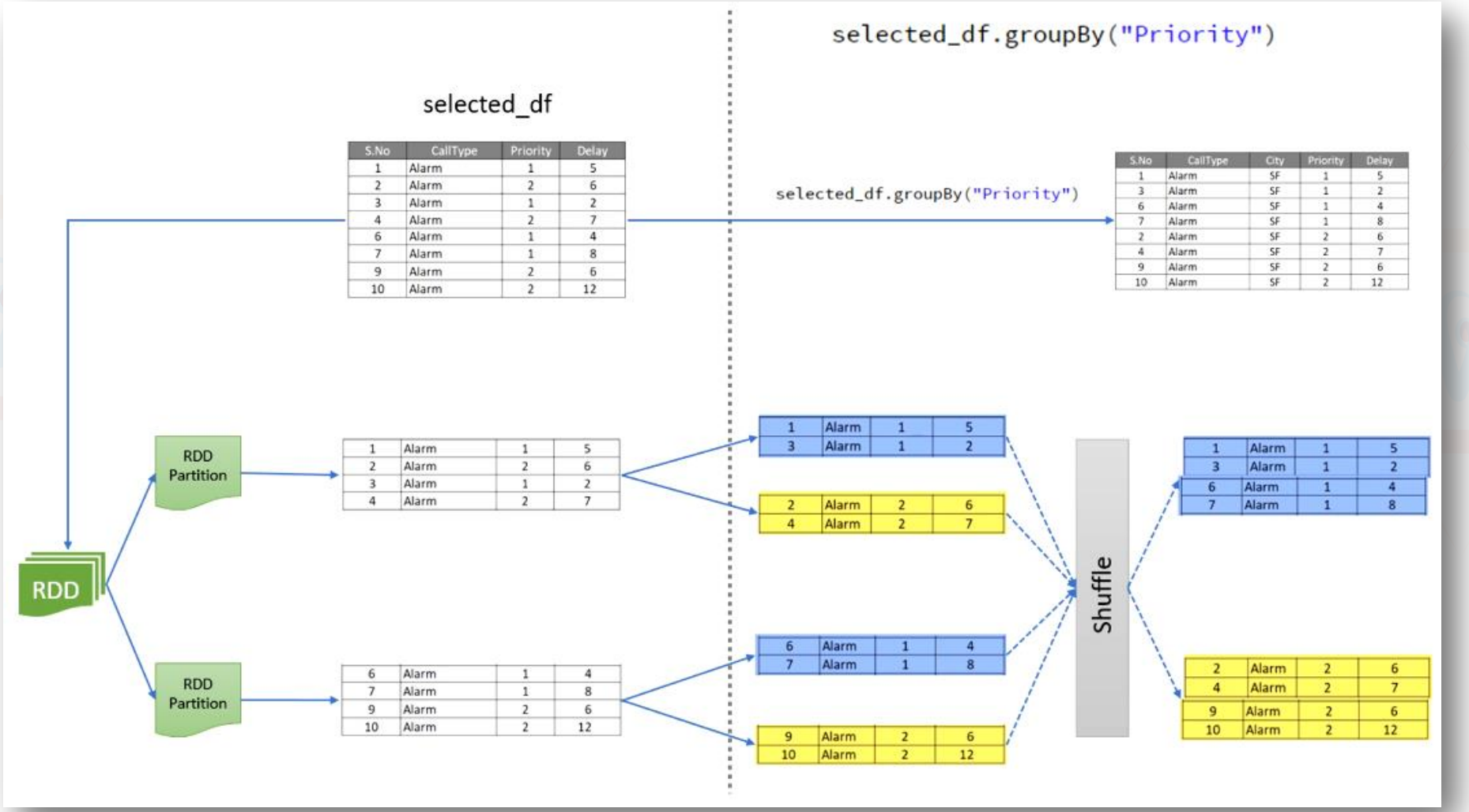
All priority one records are together. Similarly, all priority two records are lined up together.



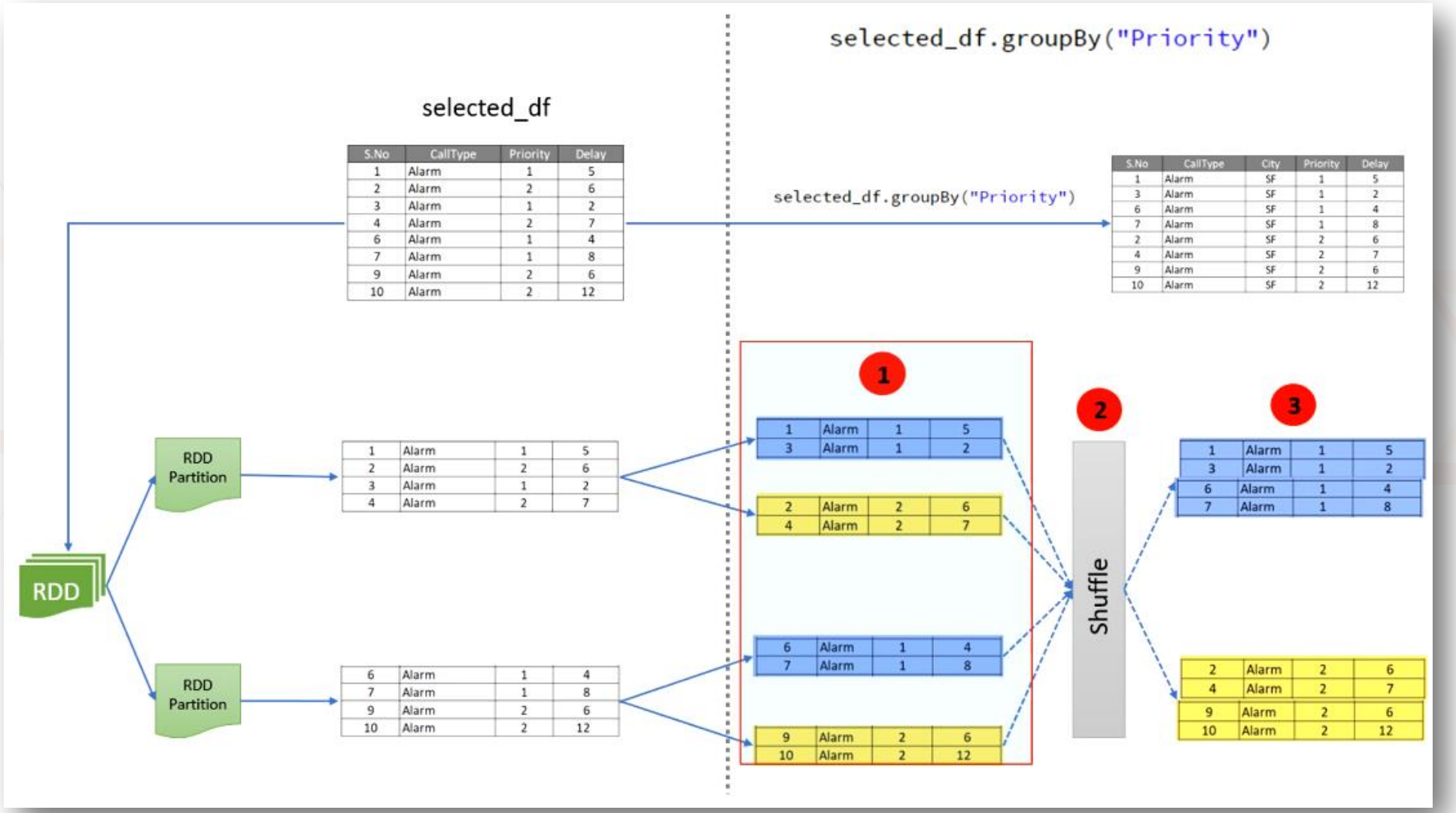
Now let's try achieving the same result at the physical level.
So Spark will do it at the partition level. Now the first partition will break into two partitions: priority one and priority two. Similarly, the second partition will also break into two. But we wanted to group all priority-one in one place. That's what the groupBy() means. As of now, we have it at two different places.



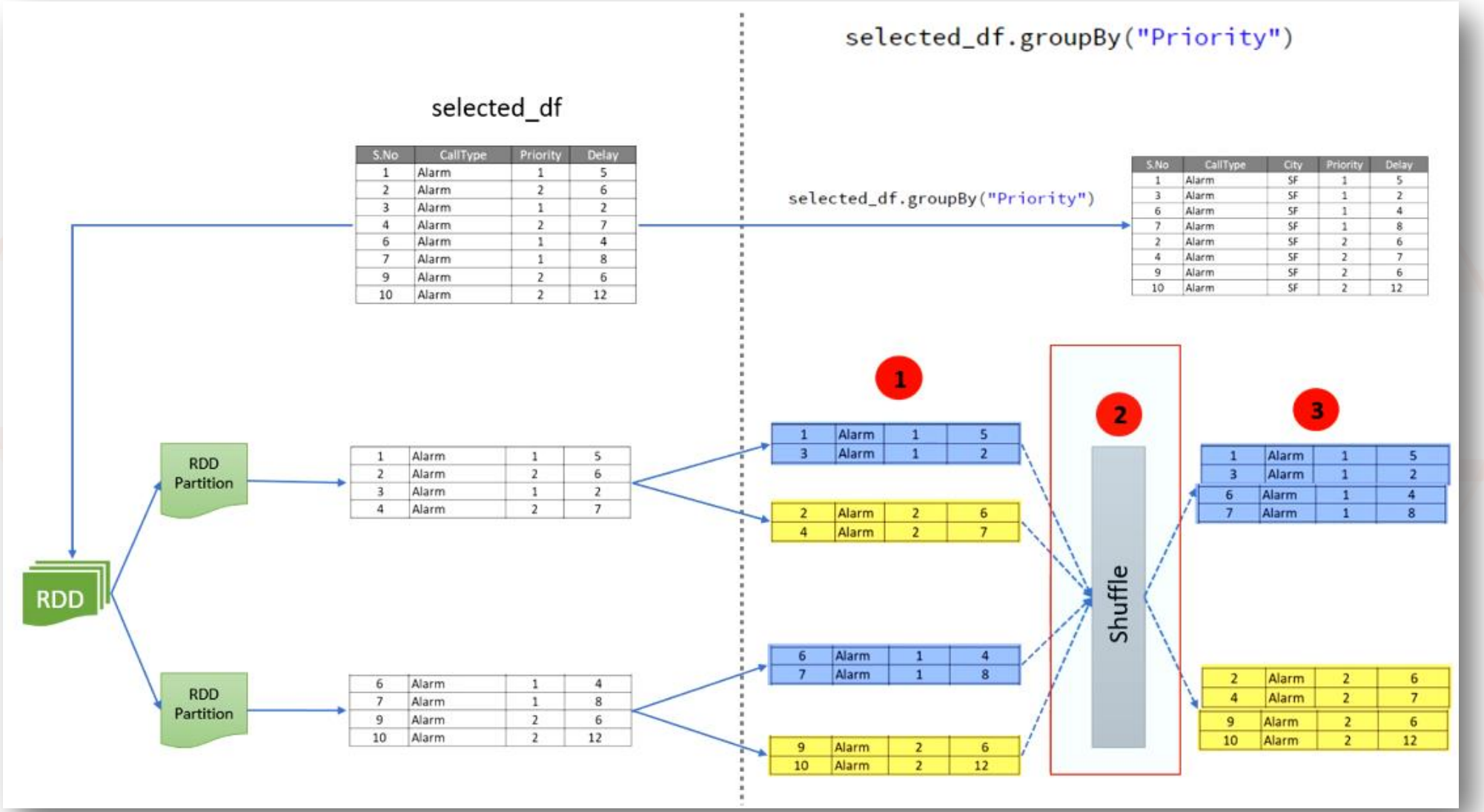
So Spark will combine all the same priority partitions.
So, in the end, we will have two new partitions: The priority one partition and a separate one for priority two.



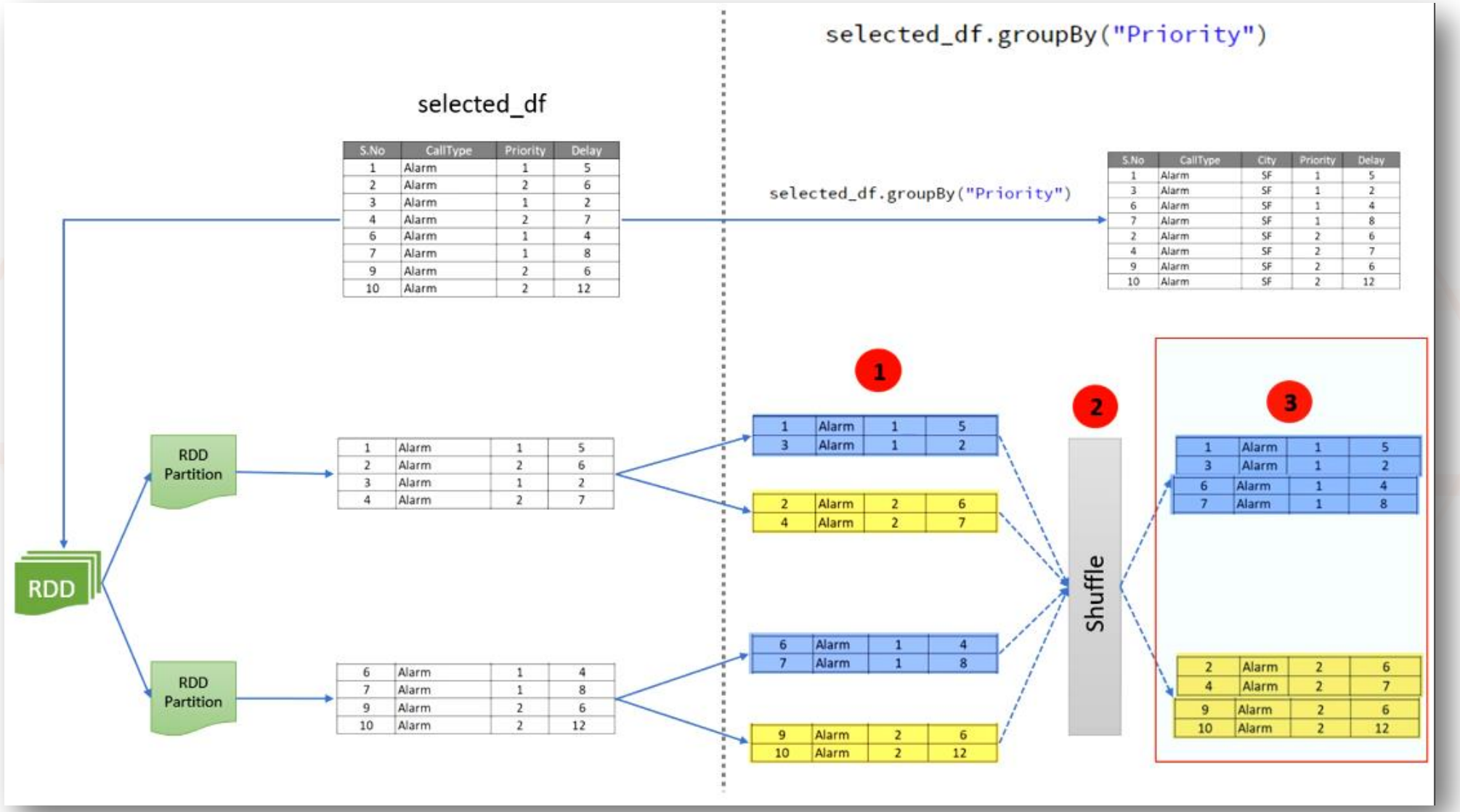
We did it in three steps.
Step one is to sort the data at each partition and break it into sub-partitions.



The second step is to shuffle the data.
The shuffle means you are exchanging the data between the partitions to regroup it.



Step three is to regroup the data as per your requirement.

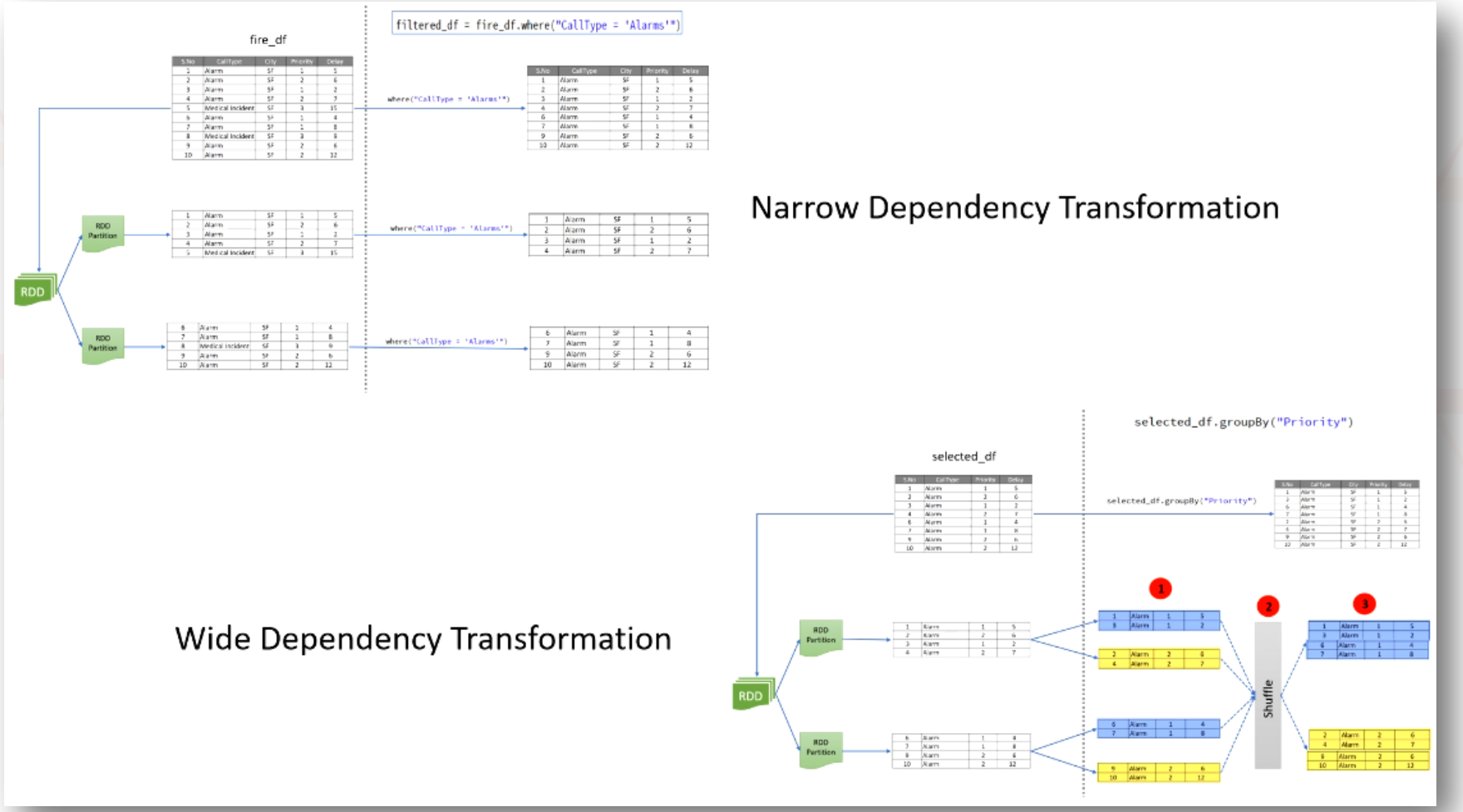


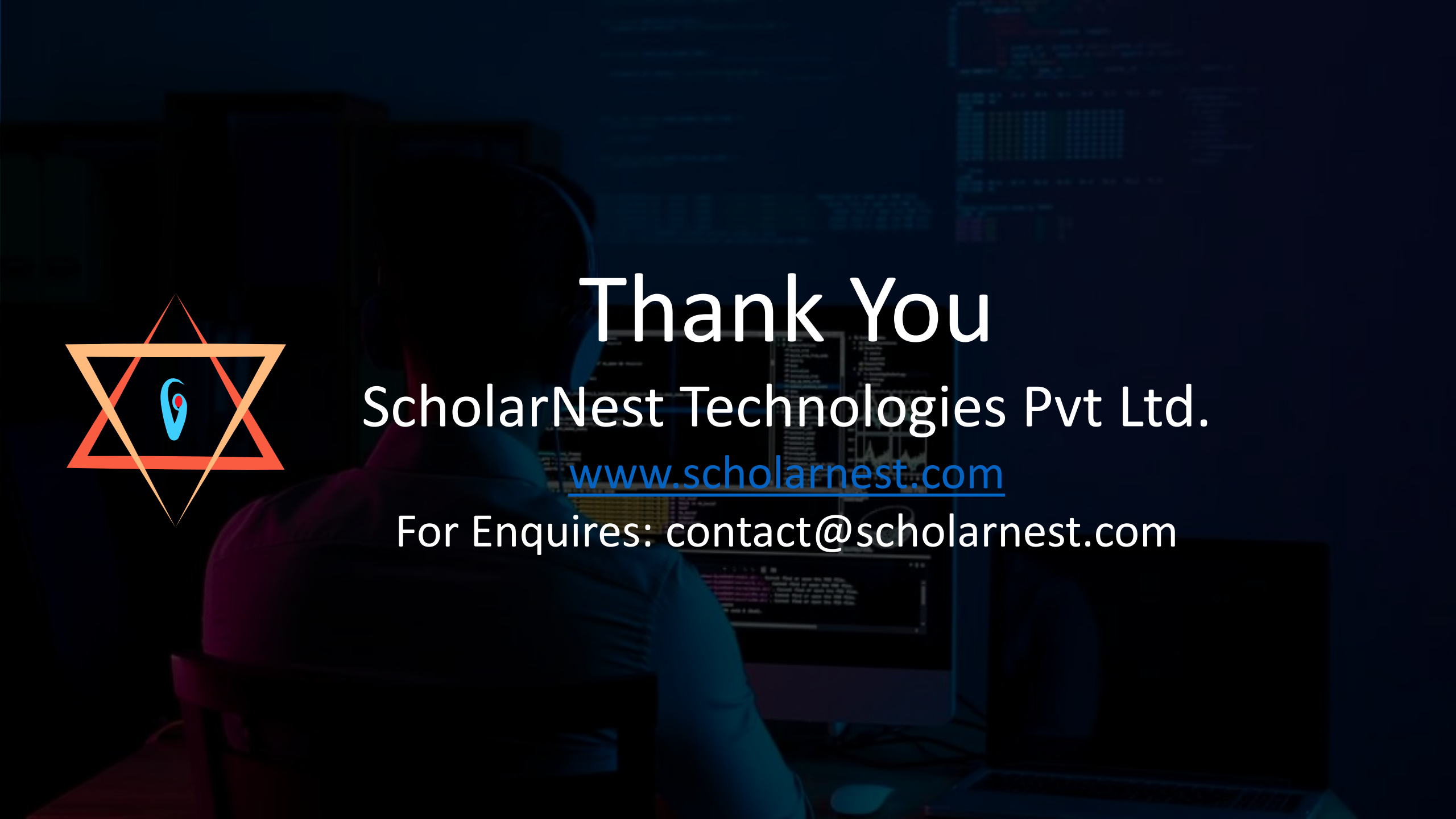
The groupBy() operation requires exchanging or shuffling the data.
We cannot implement it without the exchange.
So the groupBy() is a wide dependency transformation.

We learned two types of transformations:

1. Narrow dependency
2. Wide dependency

The difference between narrow and wide dependency transformation lies in the shuffle. A wide dependency transformation requires a shuffle. And a narrow dependency transformation does not require a shuffle. Wide dependency transformations are expensive because they take extra time to shuffle the data. They are often the main cause behind many performance problems in spark. However, with a careful design, you can handle it gracefully.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

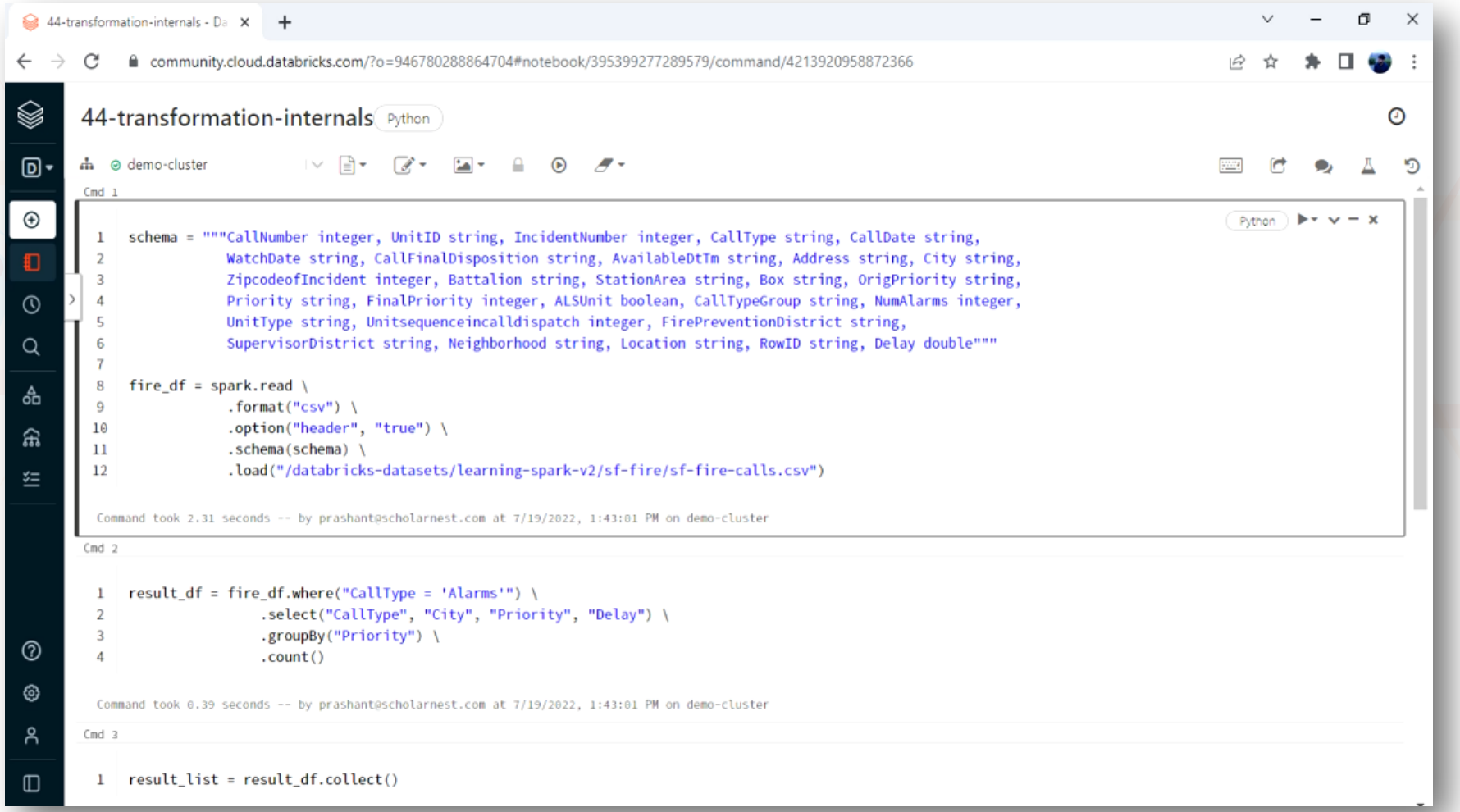
Databricks Spark Certification and Beyond

Module:
Spark
Dataframe
Internals

Lecture:
Spark
Jobs

Spark Jobs - Stages Shuffle Task and Slots

In this lecture, I am going to elaborate on the execution plan.
Let's come back to our example code. **(Reference: 44-transformation-internals.ipynb)**
We are doing three things here. I am reading a data file to create a Dataframe. Then I apply a chain of transformations. The result is another Dataframe. Finally, We are using an action.

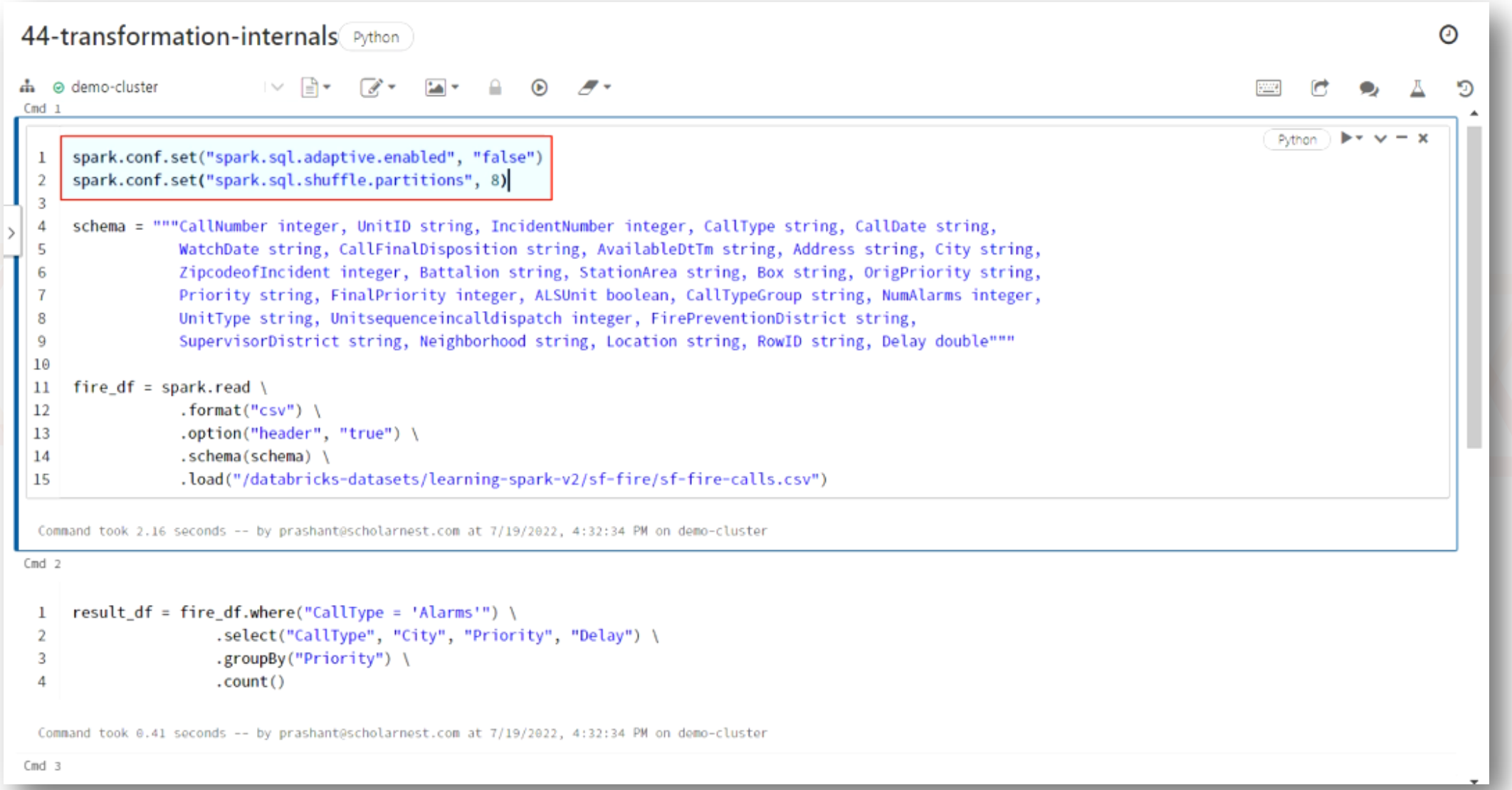


Now, based on this example, we want to understand the internal execution plan.
How are these things executed internally?

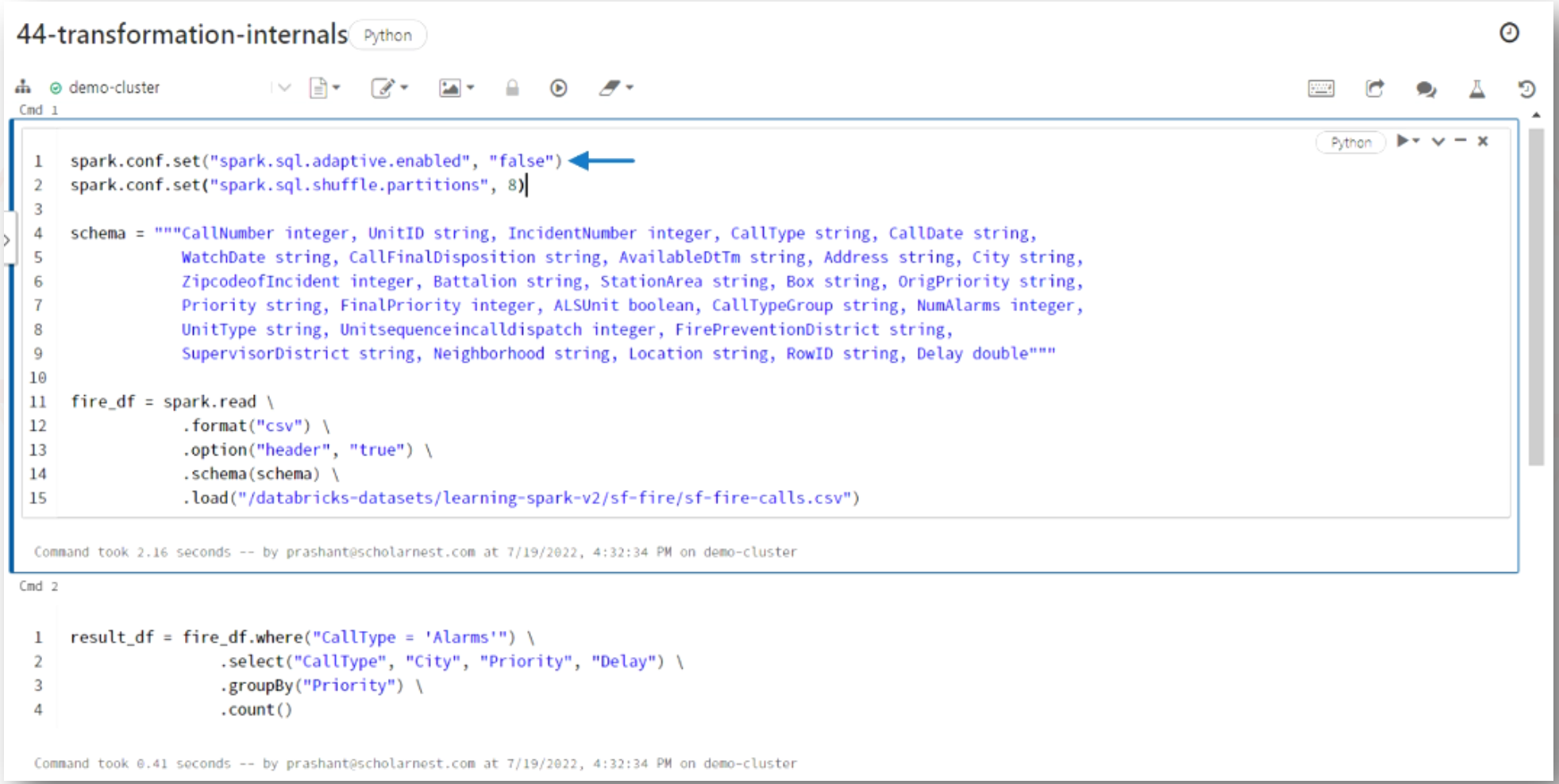
That's not an easy goal.
Why? Because Spark is similar to a compiler.
It takes your straightforward code, compiles it to generate low-level Spark code, and creates an execution plan.
All those things are too complicated, and understanding what is internally happening is difficult.

But let me try giving you some good sense about it.

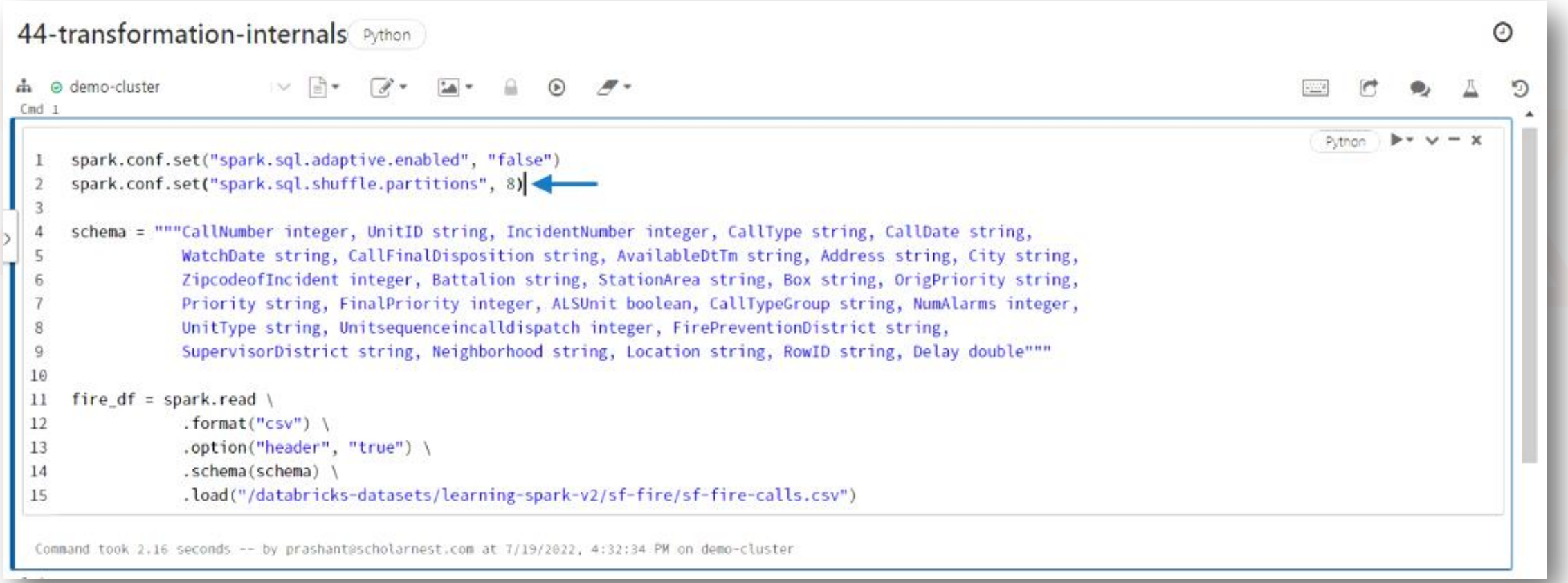
Go to the first line in the top most cell and add the following two lines as shown below.



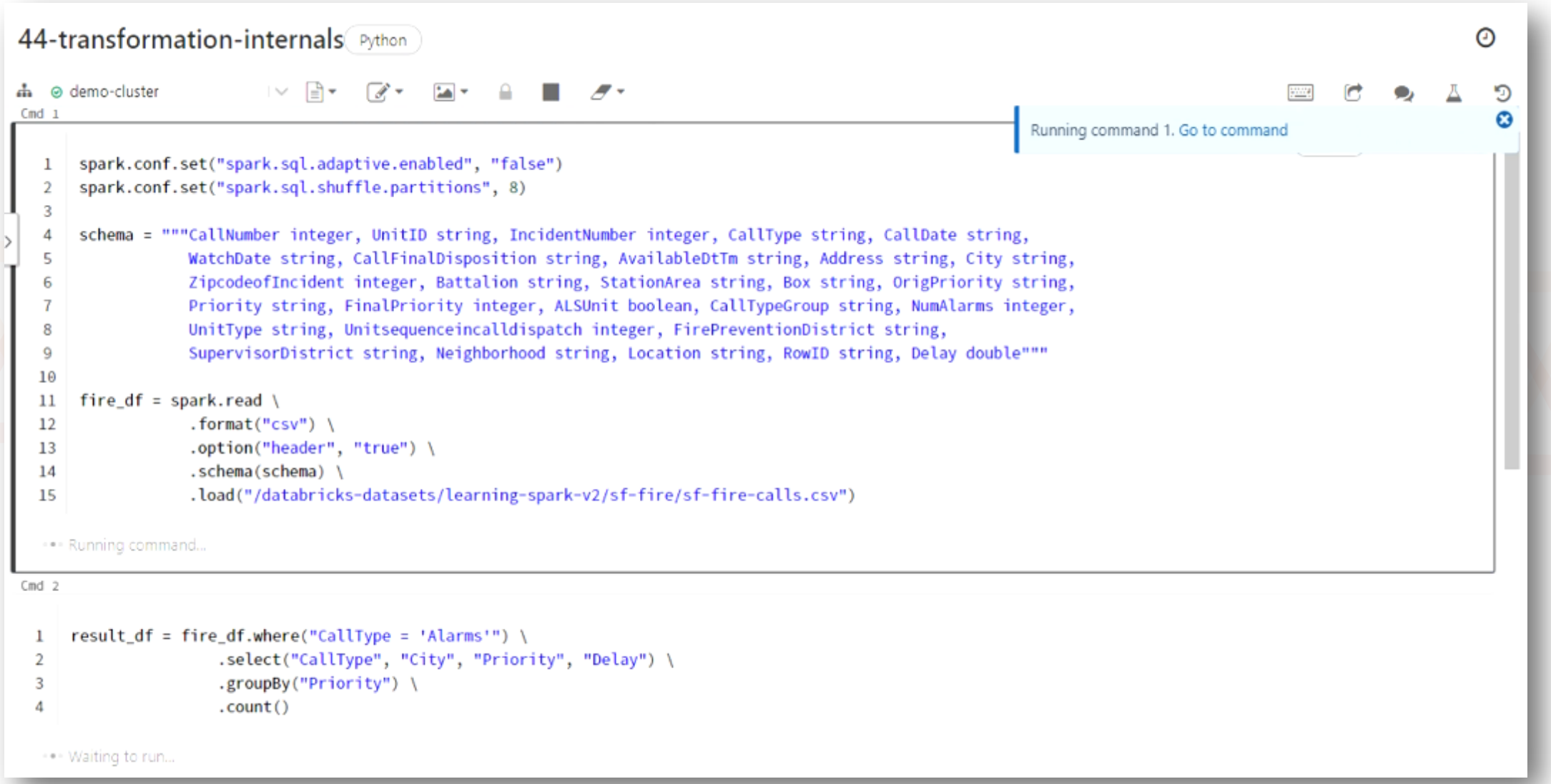
I am disabling adaptive query optimization. Why? Because it dynamically changes the execution plan and makes things difficult to understand. So, I am disabling it to avoid confusion with dynamically changing plans.



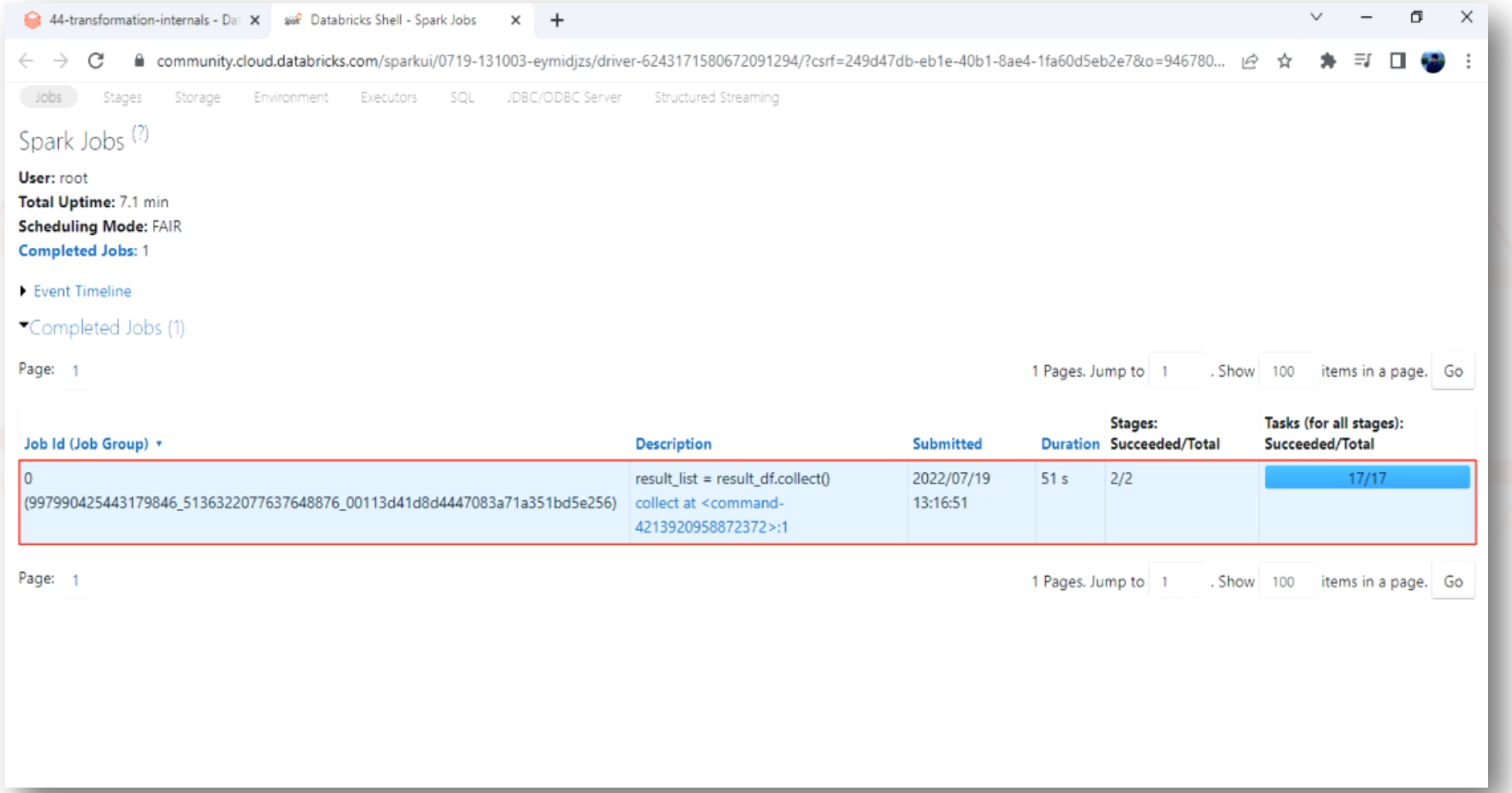
The second configuration is to optimize the shuffle partitions. We already know we are applying groupBy(), a wide dependency transformation. So, the groupBy() is going to result in a shuffle. Spark defaults to 200 partitions after the Shuffle, which is not meaningful. So I am setting it to eight shuffle partitions. Why eight? Because I know I am grouping on the priority, and I have eight unique priority values. So it makes sense to set eight shuffle partitions.



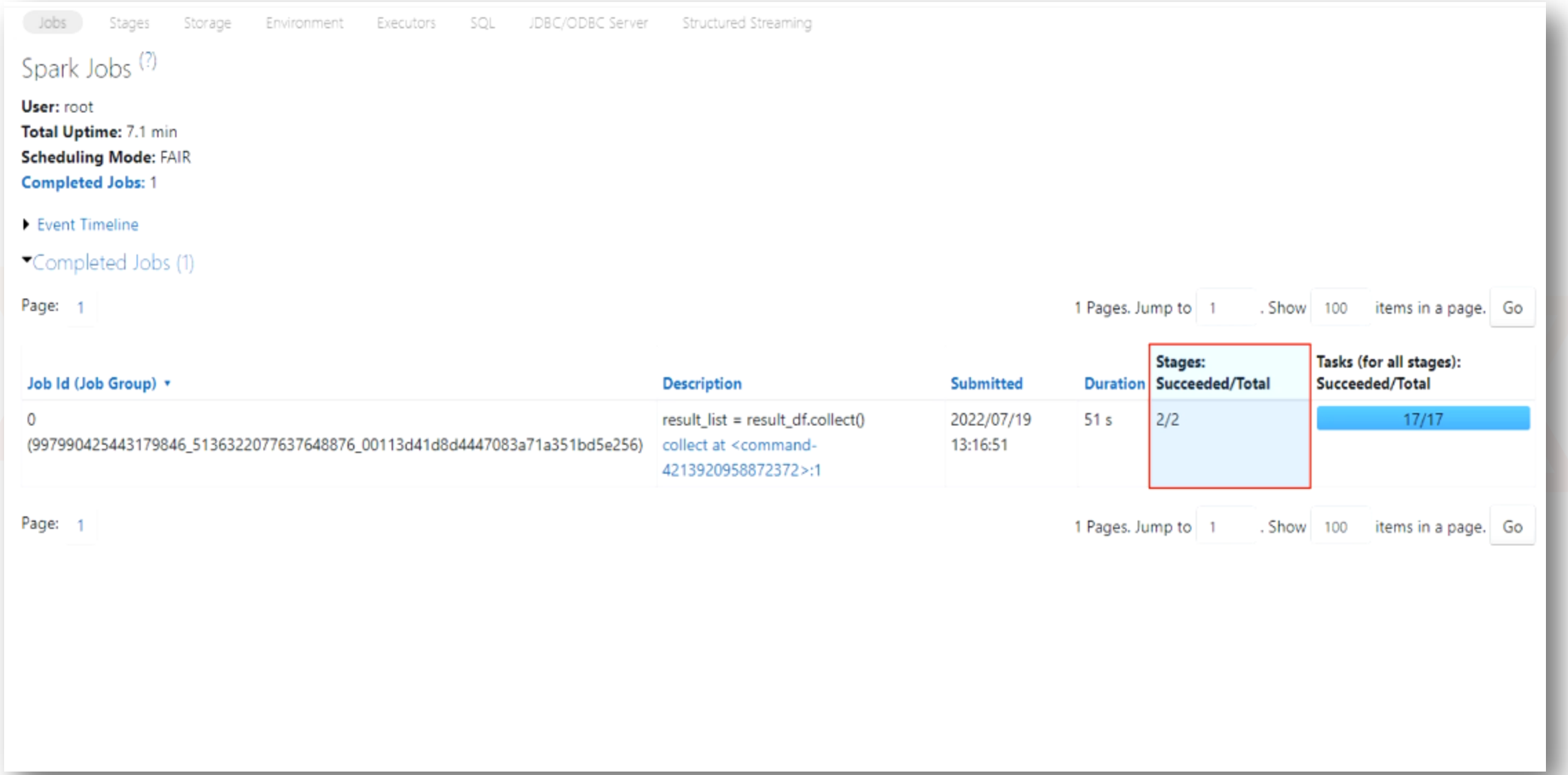
Now, run the notebook. And once executed you can go to the Spark UI.



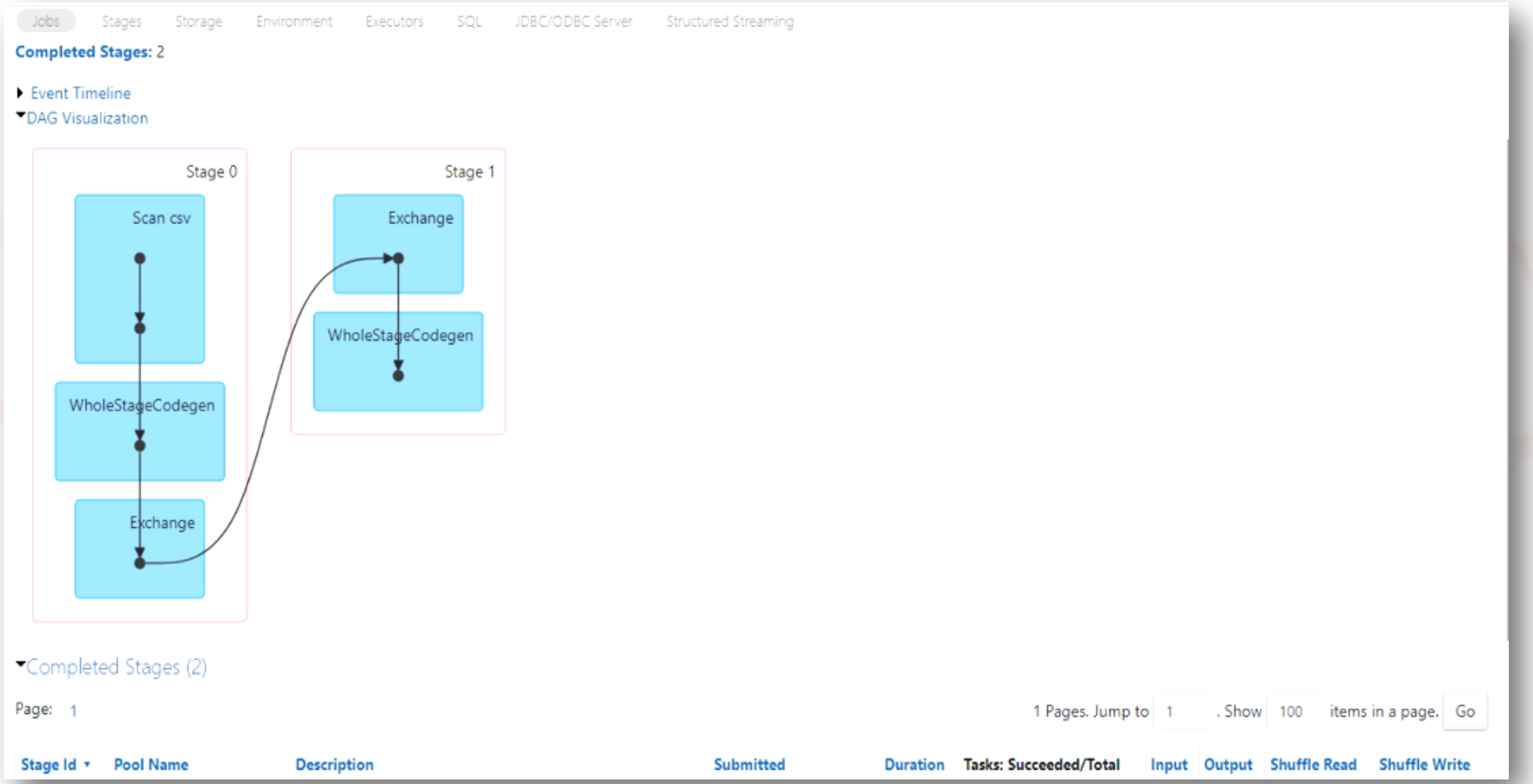
Here is the Spark UI. I am looking at the Jobs tab. I have a simple example, I got one job. A Spark job is broken down into stages.



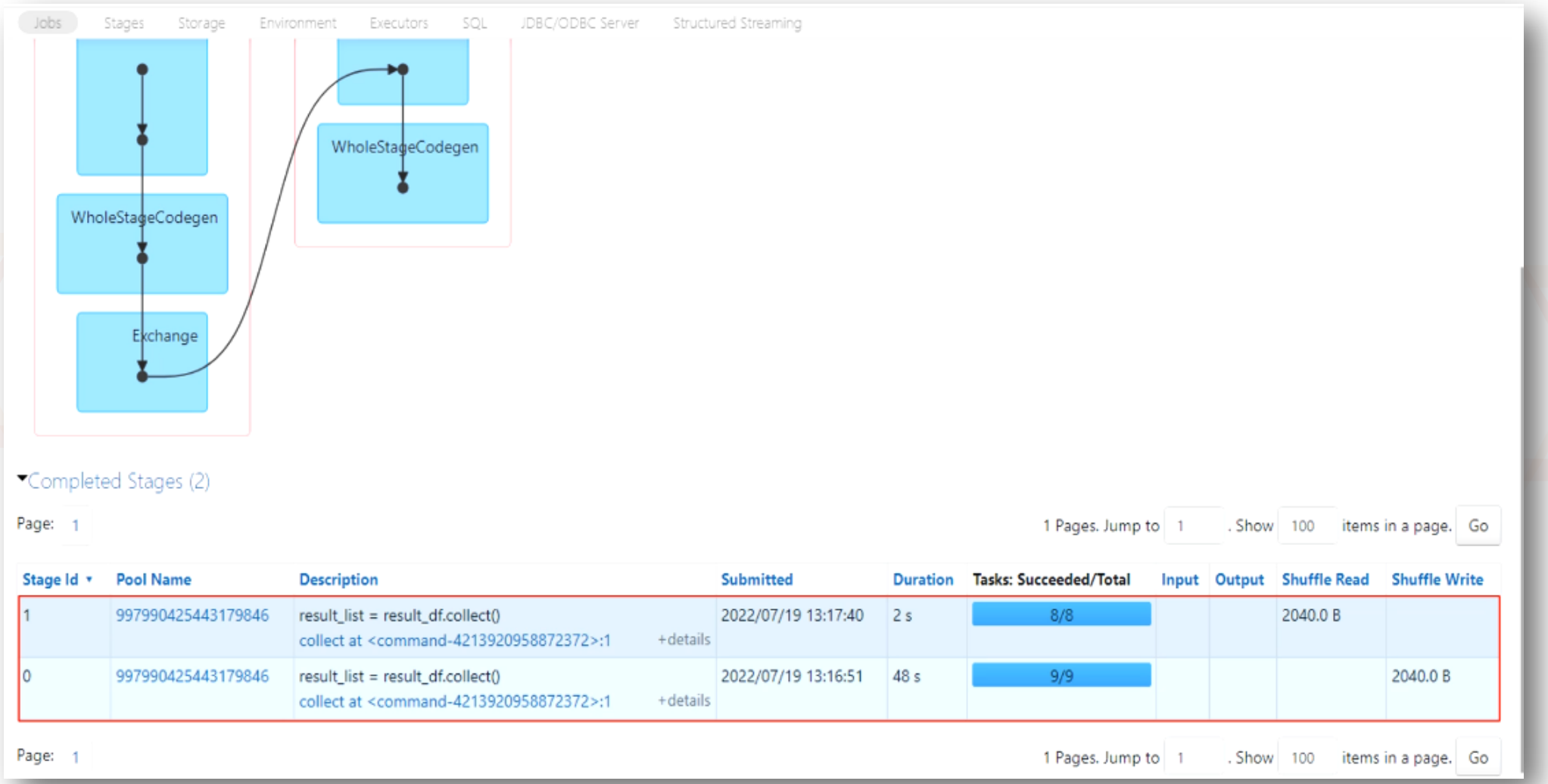
My Job has got two stages.



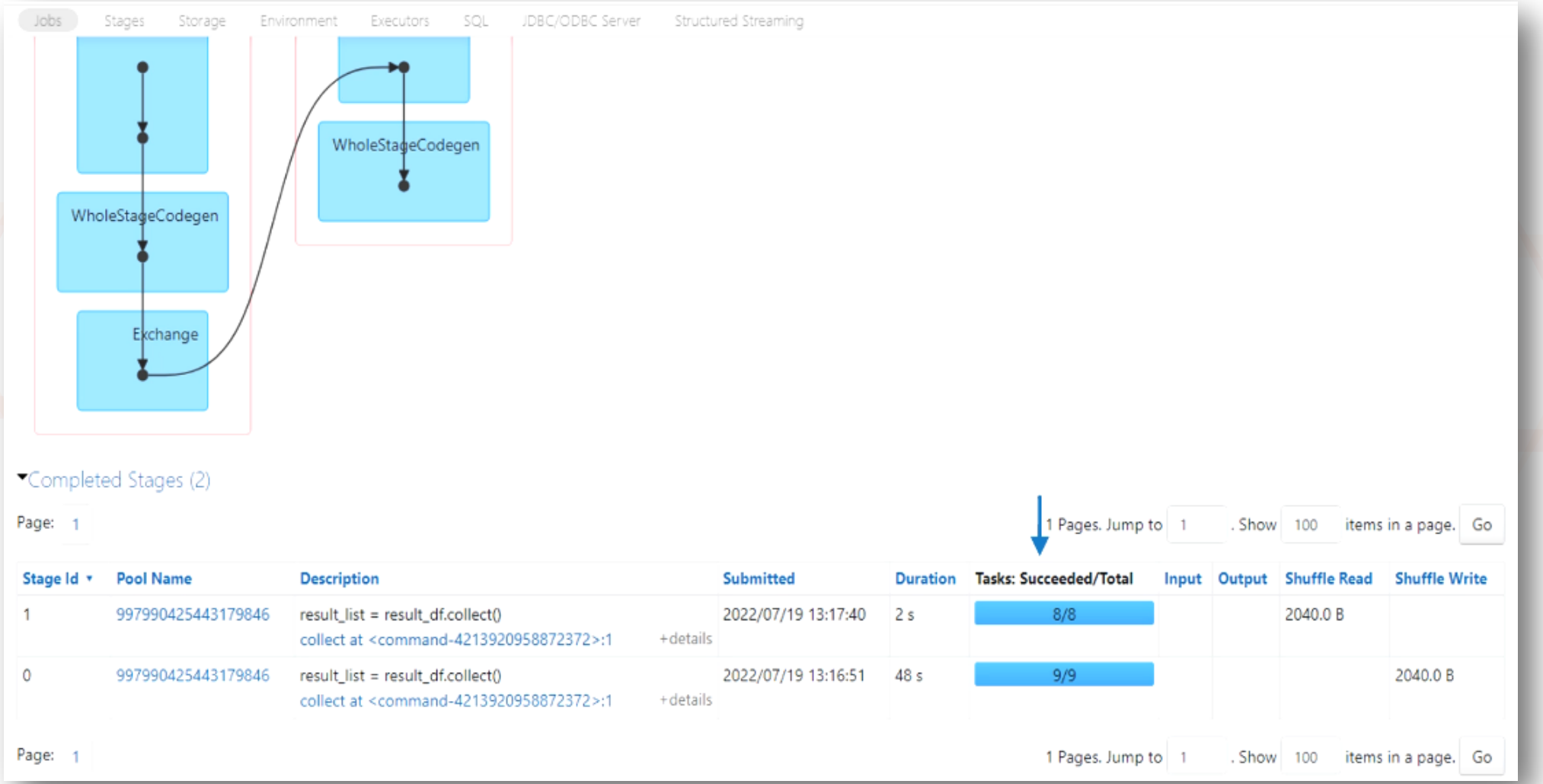
If you click on the job description, you will land on the page shown below. The diagram shown below is the visual DAG representation of the Spark Job. It shows two stages: Stage 0 and stage 1.



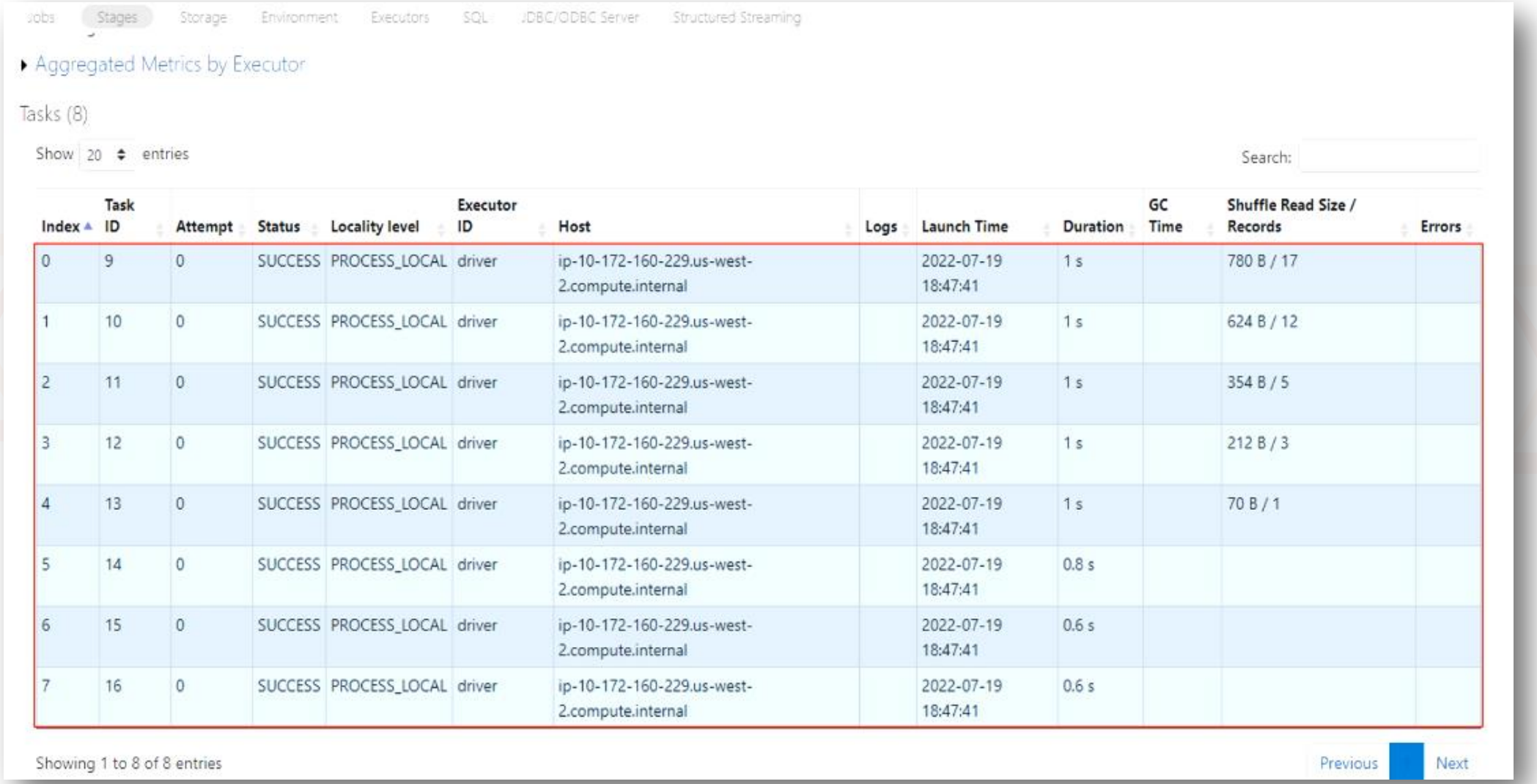
Scroll down, and you will see a list of stages.
We have two stages to I see two items in the list.



Do you see the column Tasks: Succeeded/Total.
Stage zero ran nine tasks. But stage one executed only eight tasks. If you click the stage description, you will see the task details.



I clicked on the description of stage 1 and landed on the page shown below.
You can see the list of 8 tasks executed in stage 1.



Jobs | Stages | Storage | Environment | Executors | SQL | JDBC/ODBC Server | Structured Streaming

▸ Aggregated Metrics by Executor

Tasks (8)

Show 20 entries

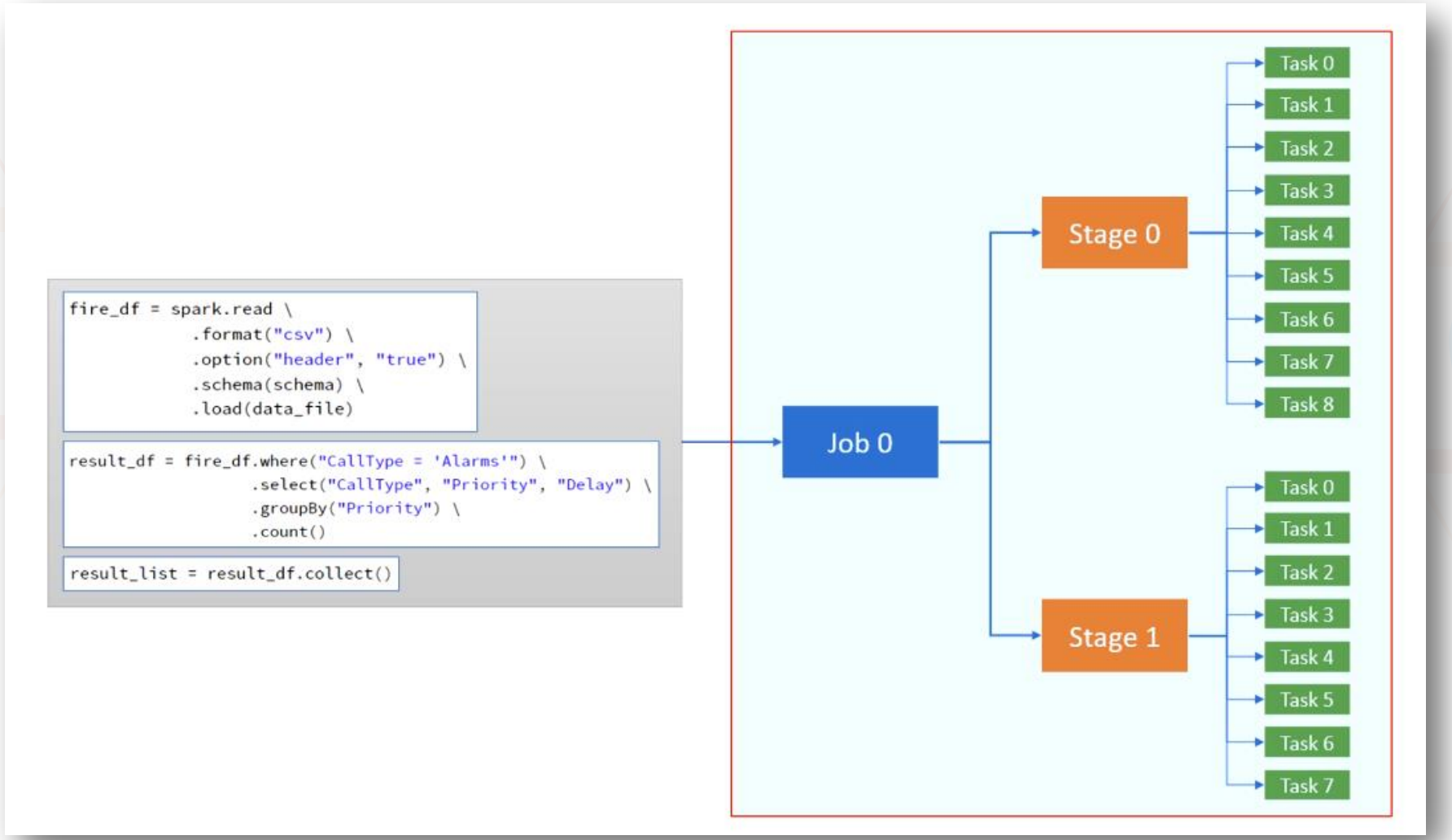
Search:

Index ▲	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	9	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-160-229.us-west-2.compute.internal		2022-07-19 18:47:41	1 s		780 B / 17	
1	10	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-160-229.us-west-2.compute.internal		2022-07-19 18:47:41	1 s		624 B / 12	
2	11	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-160-229.us-west-2.compute.internal		2022-07-19 18:47:41	1 s		354 B / 5	
3	12	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-160-229.us-west-2.compute.internal		2022-07-19 18:47:41	1 s		212 B / 3	
4	13	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-160-229.us-west-2.compute.internal		2022-07-19 18:47:41	1 s		70 B / 1	
5	14	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-160-229.us-west-2.compute.internal		2022-07-19 18:47:41	0.8 s			
6	15	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-160-229.us-west-2.compute.internal		2022-07-19 18:47:41	0.6 s			
7	16	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-160-229.us-west-2.compute.internal		2022-07-19 18:47:41	0.6 s			

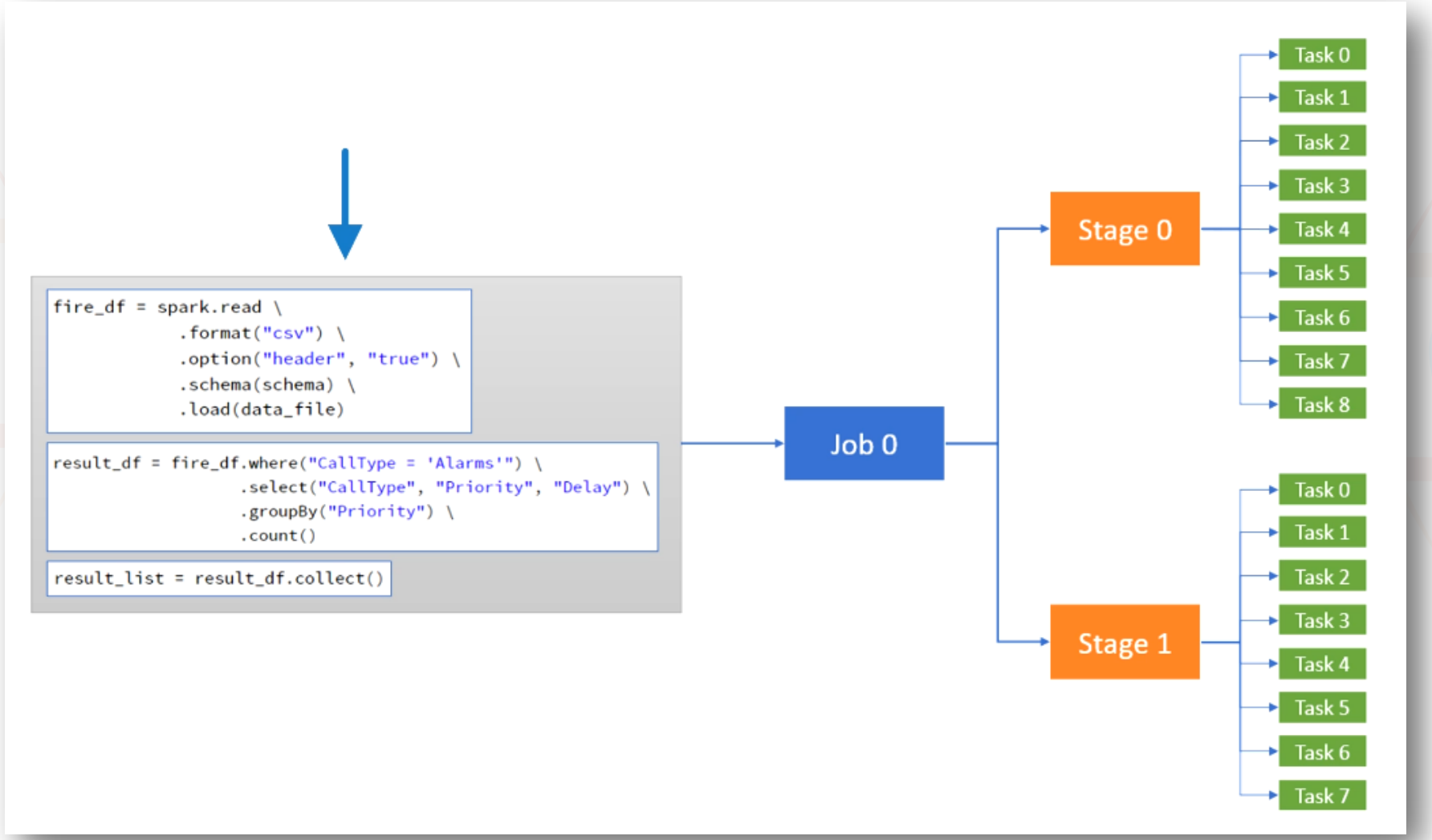
Showing 1 to 8 of 8 entries

Previous 1 Next

So, Spark code runs as Jobs. A job is again broken into one or more stages. In our case, we got two stages. These stages are further split into tasks. We got nine tasks for stage zero and eight tasks for stage 1. This complete hierarchy is your execution plan.



However, you might be wondering how this all happens?
I mean, can we map our code with this diagram. Can we understand how my code translates into these tasks? Let's try to understand that.

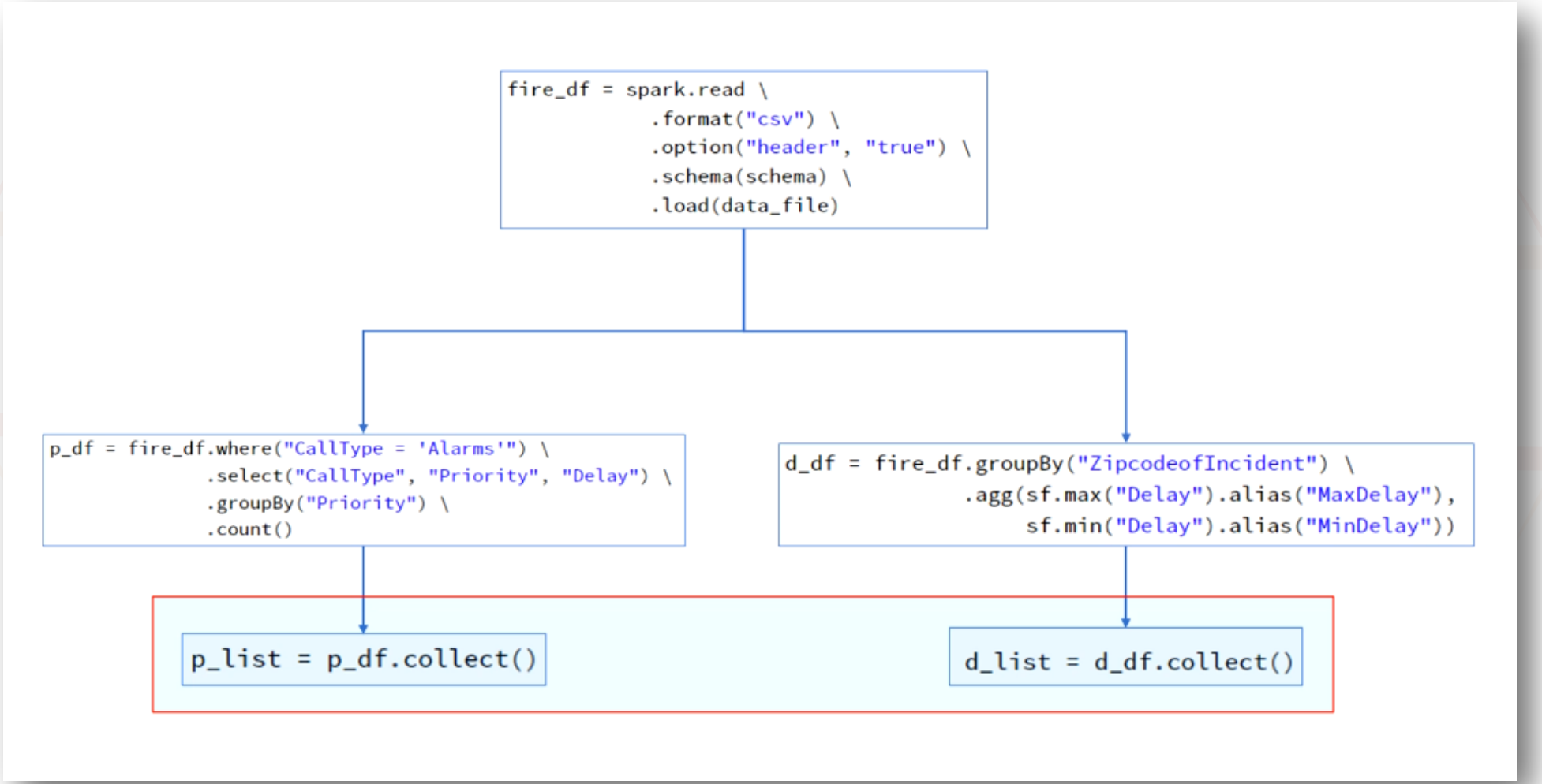


We have the following code. Let us go from top to bottom. I am reading a data file to create a Spark Dataframe. Then I apply a chain of transformations in the fire_df Dataframe. The following line is an action. I have another chain of transformations on the same fire_df. Then I again have an action. This code snippet has got two execution hierarchies.

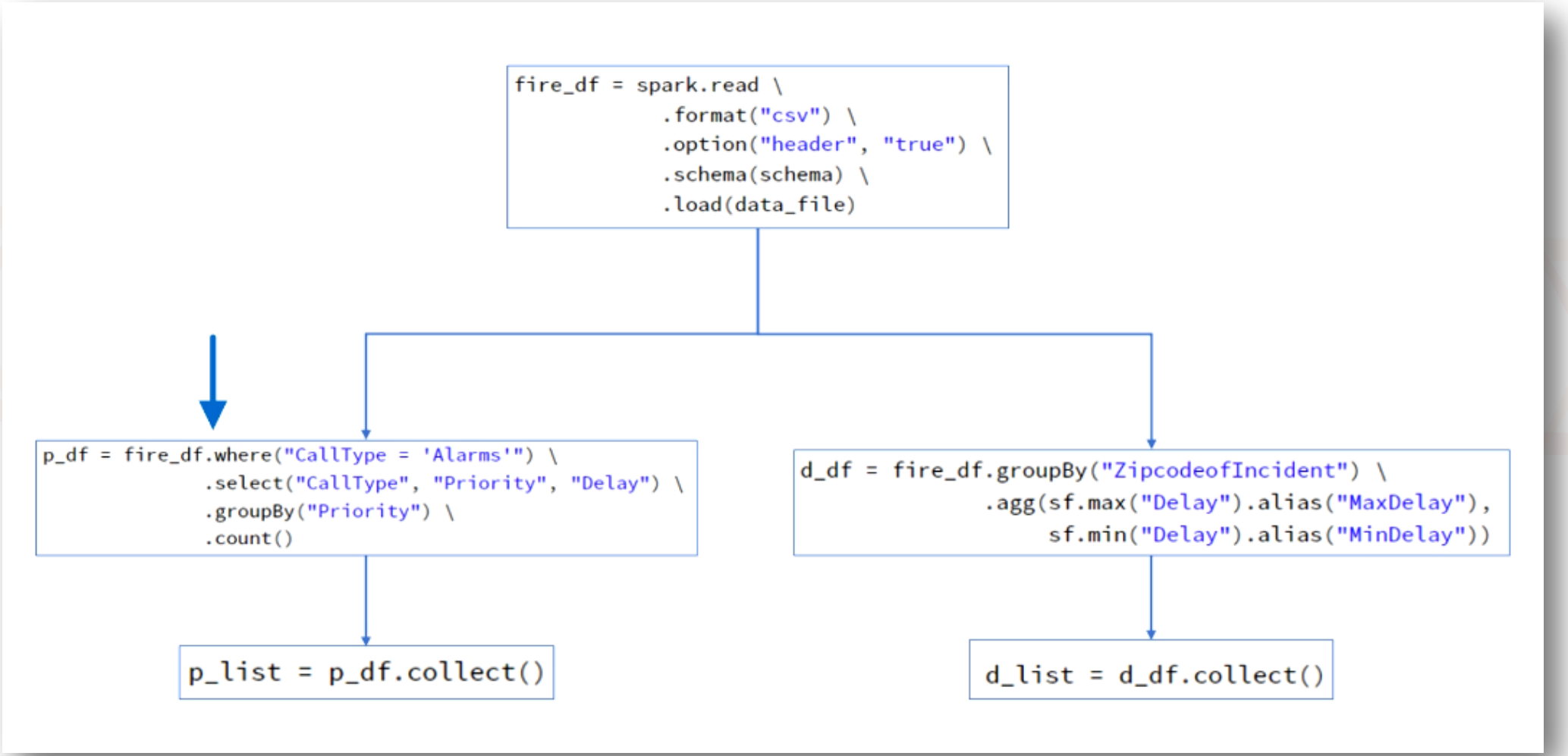
Spark Code Snippet

```
fire_df = spark.read \
            .format("csv") \
            .option("header", "true") \
            .schema(schema) \
            .load(data_file)

p_df = fire_df.where("CallType = 'Alarms'") \
              .select("CallType", "Priority", "Delay") \
              .groupBy("Priority") \
              .count()

p_list = p_df.collect()

d_df = fire_df.groupBy("ZipcodeofIncident") \
              .agg(sf.max("Delay").alias("MaxDelay"),
                   sf.min("Delay").alias("MinDelay"))

d_list = d_df.collect()
```

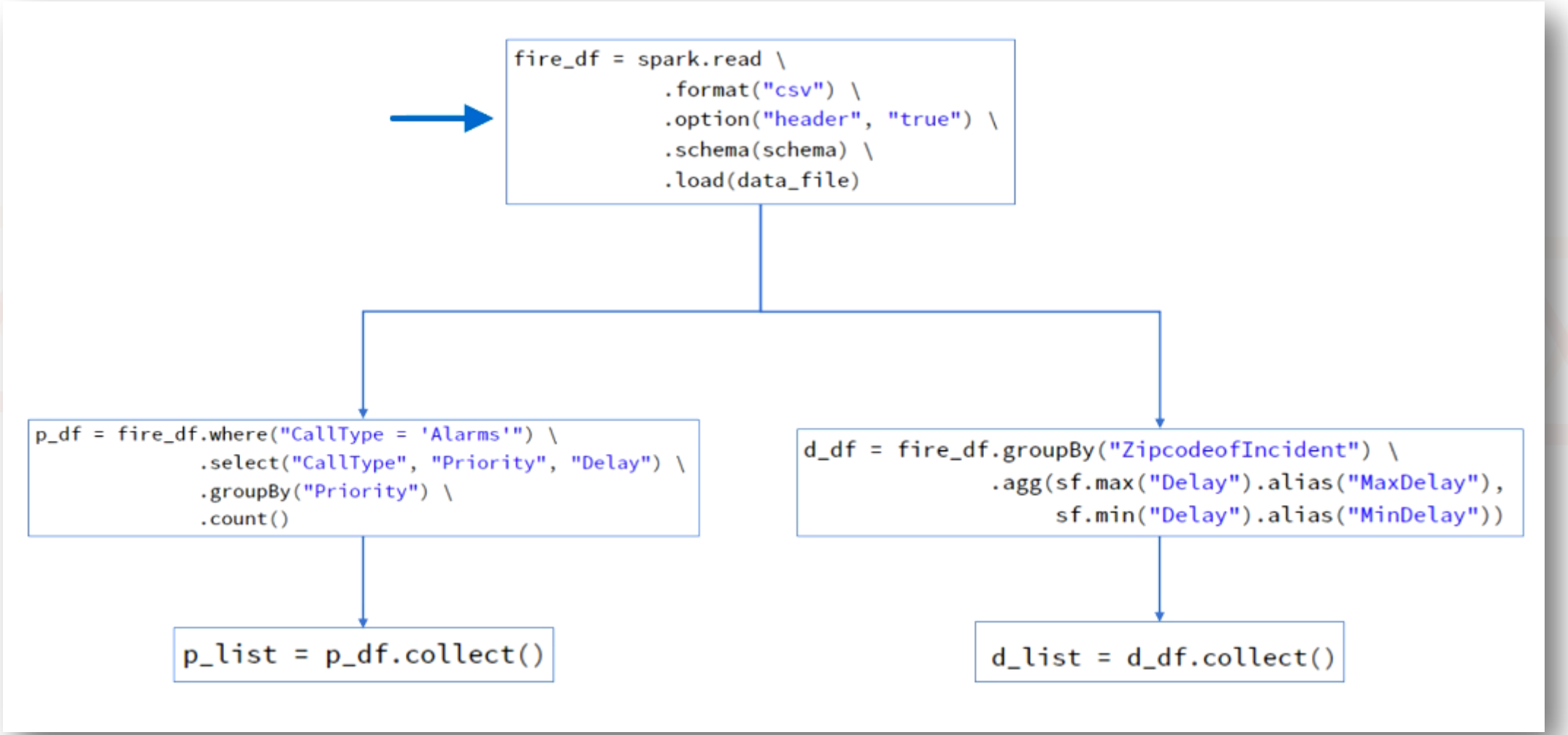

I have arranged our code according to the dependencies.
So we have actions at the leaf nodes. Both of these are actions. Then we trace it back using the dependencies.



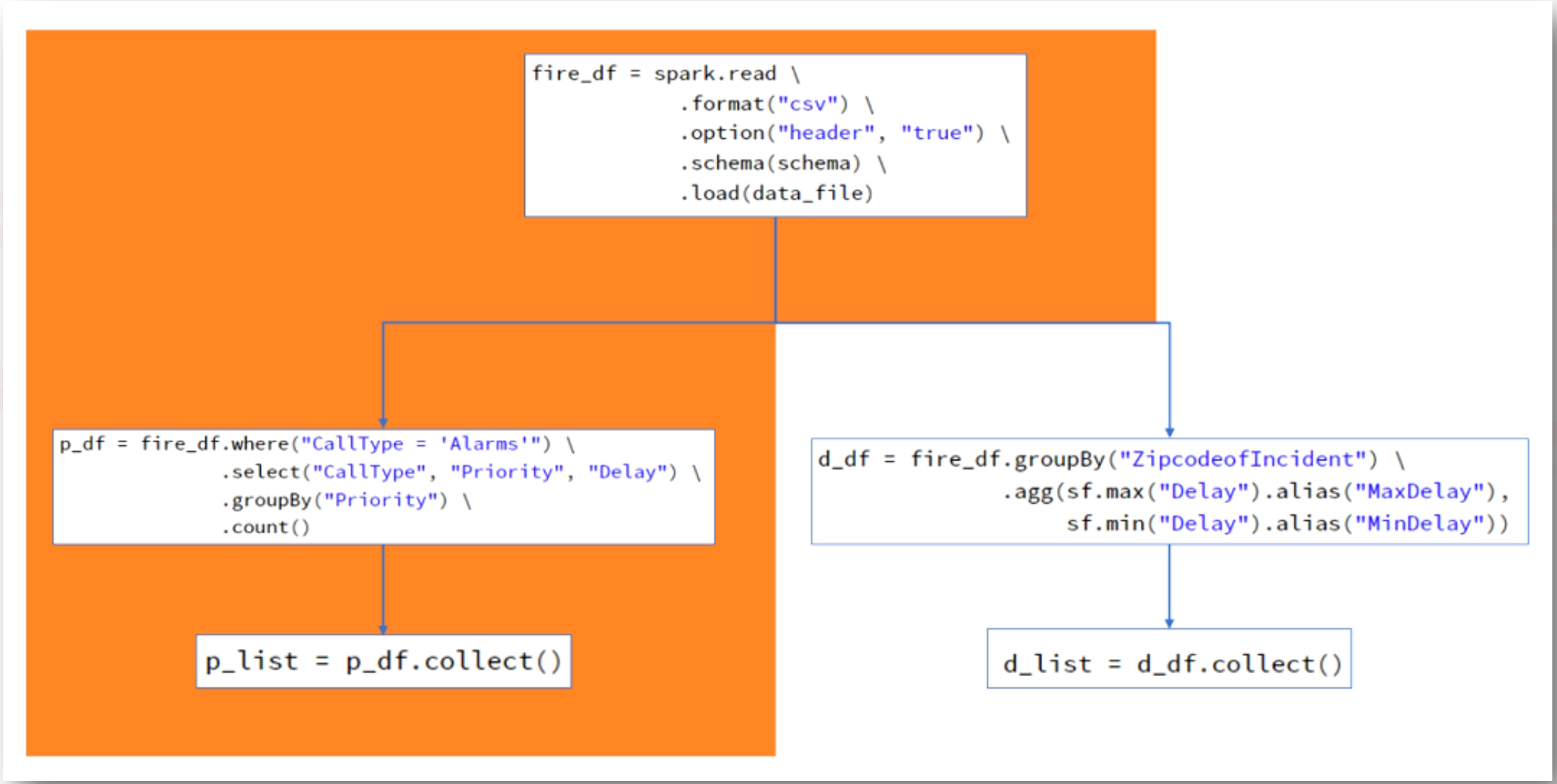
The collect() action depends on the p_df.
So we trace it back to the p_df.



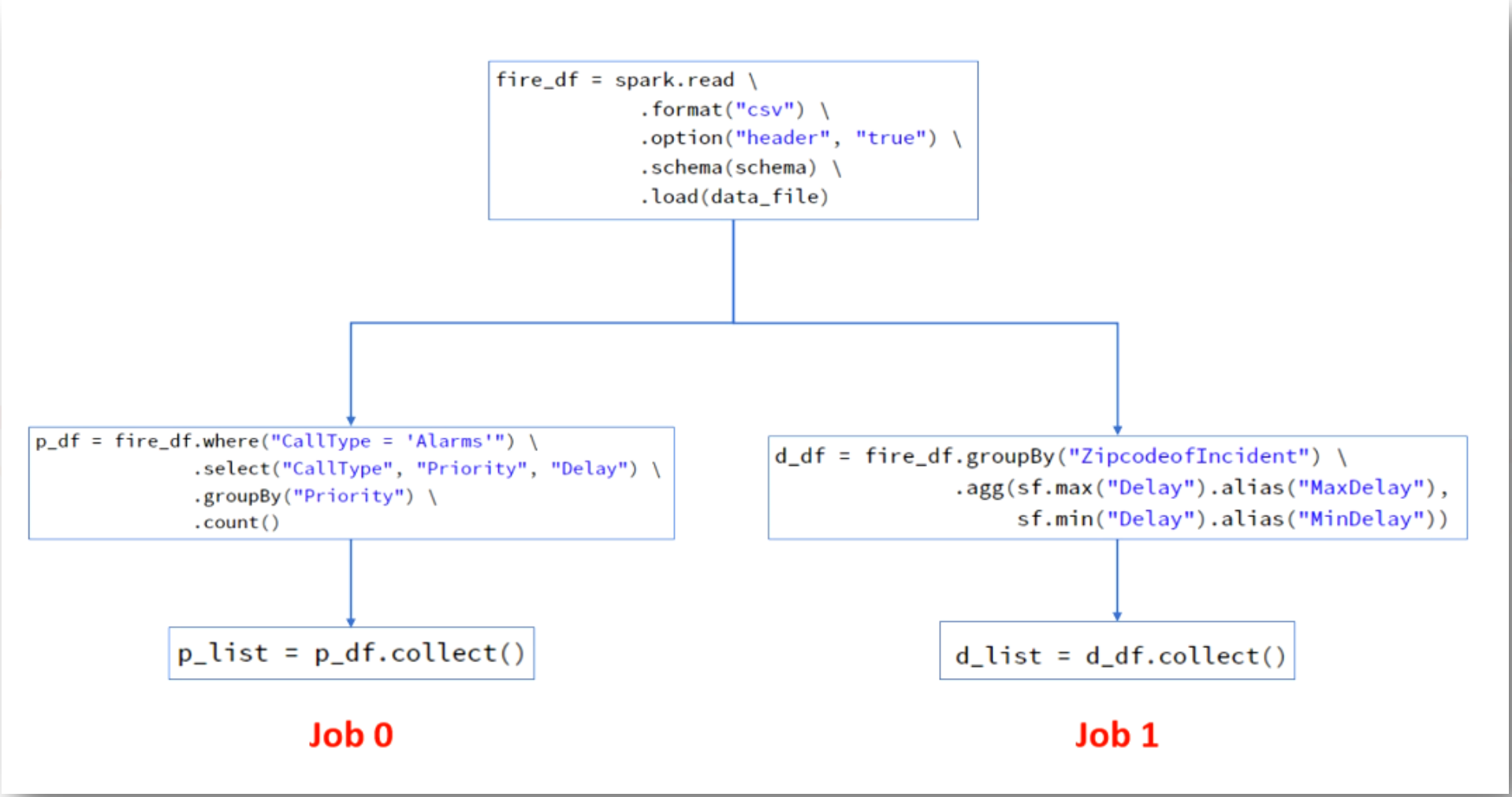
However, the p_df depends on the fire_df. So we trace it back to the fire_df.
So this tree is a dependency hierarchy that terminates with a spark action.



So Spark will make a dependency graph and run each action as a Spark job. We have two actions here. So we can expect two jobs. The first Job will cover action with the entire code hierarchy. Similarly, the second job covers the other action with code hierarchy.

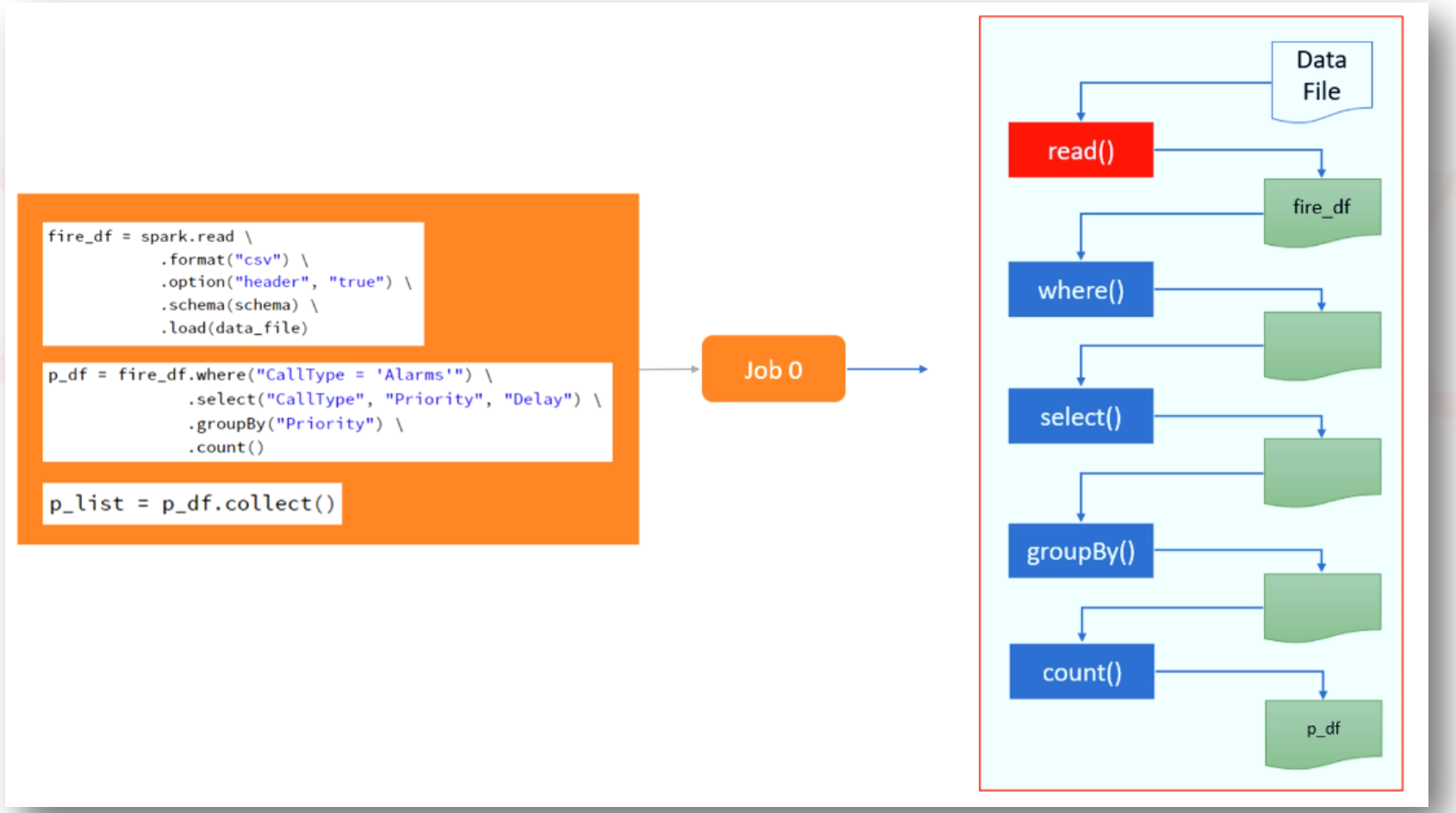


So I hope you understand that How Spark translates your code into Spark Jobs. It is as simple as each spark action with its hierarchy will run as a spark job. So in this example, we will see two Spark jobs.

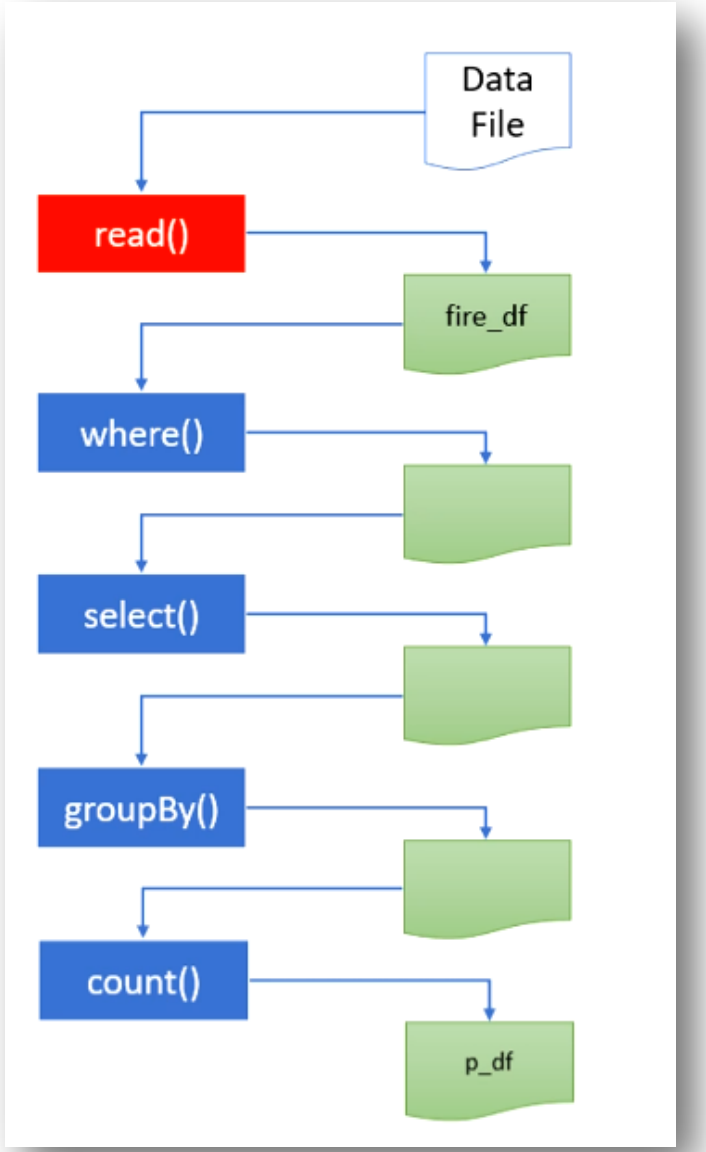


The next step is understanding how Spark breaks a job into stages.
So here is the code which creates one Spark job. Now Spark will take the code and create a logical plan.

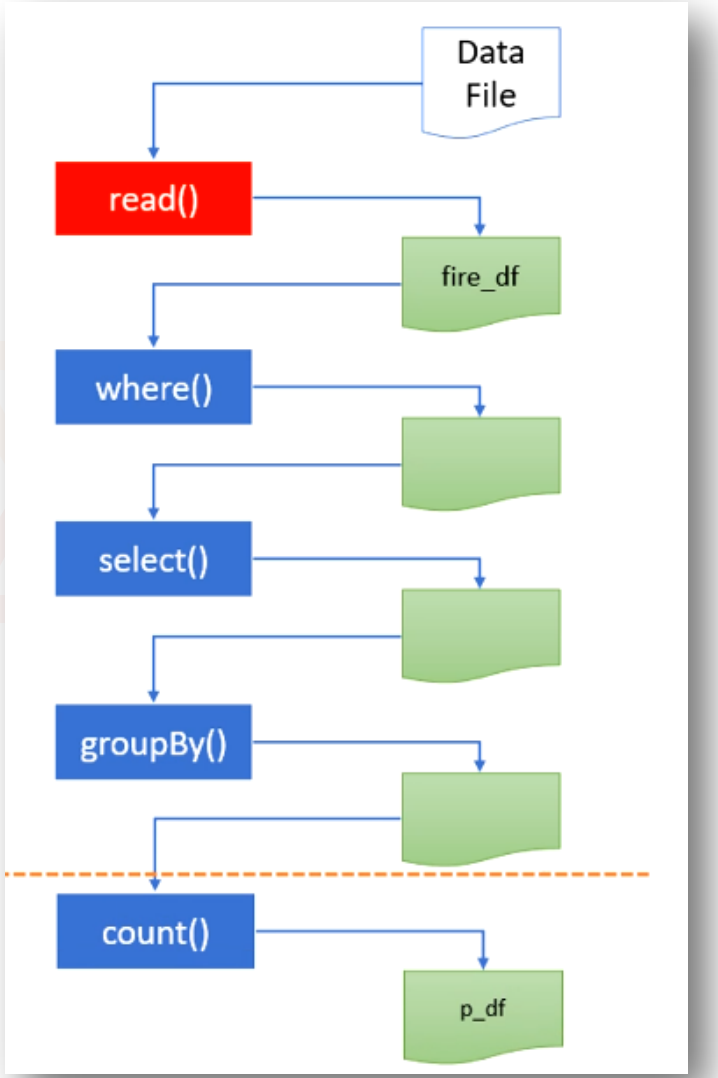
Here is my logical plan for the code shown in the left side.
The job must start with reading the data file. Then it must apply the four transformations to create the p_df. That's all we want to do as part of this job. I ignored the collect action for now, and we will come back to it later. So this is what we call the logical query plan of your spark job.
Spark will create a logical query plan for each spark job.



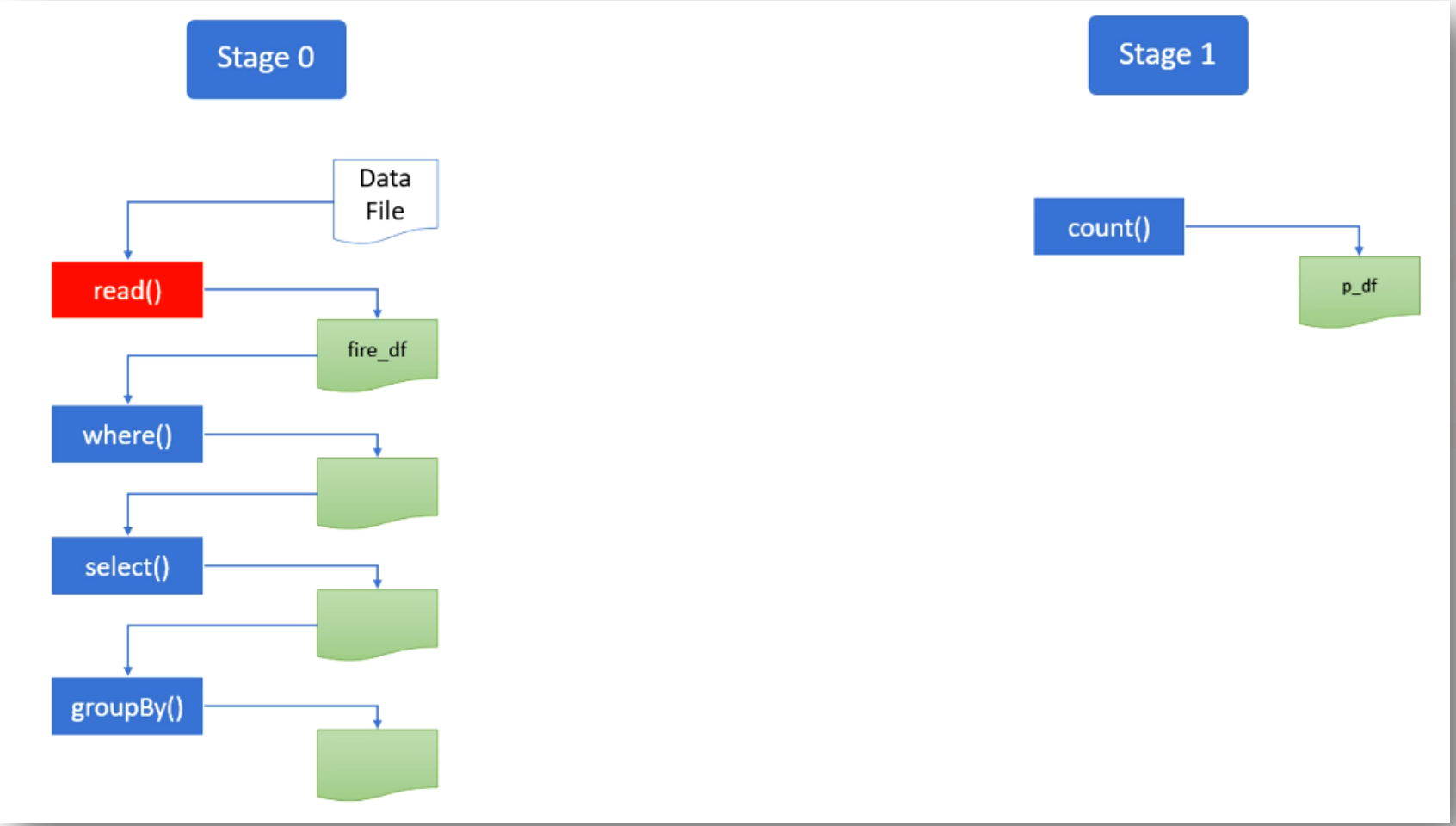
Once we have the logical plan, Spark will start breaking this plan into stages.
Spark will look at this plan to find out the wide dependency transformations.



I have one wide dependency on this plan. Where is it? The groupBy() method.
So Spark will break this plan at the wide dependency. The first part becomes the first stage. The second one goes to the second stage.

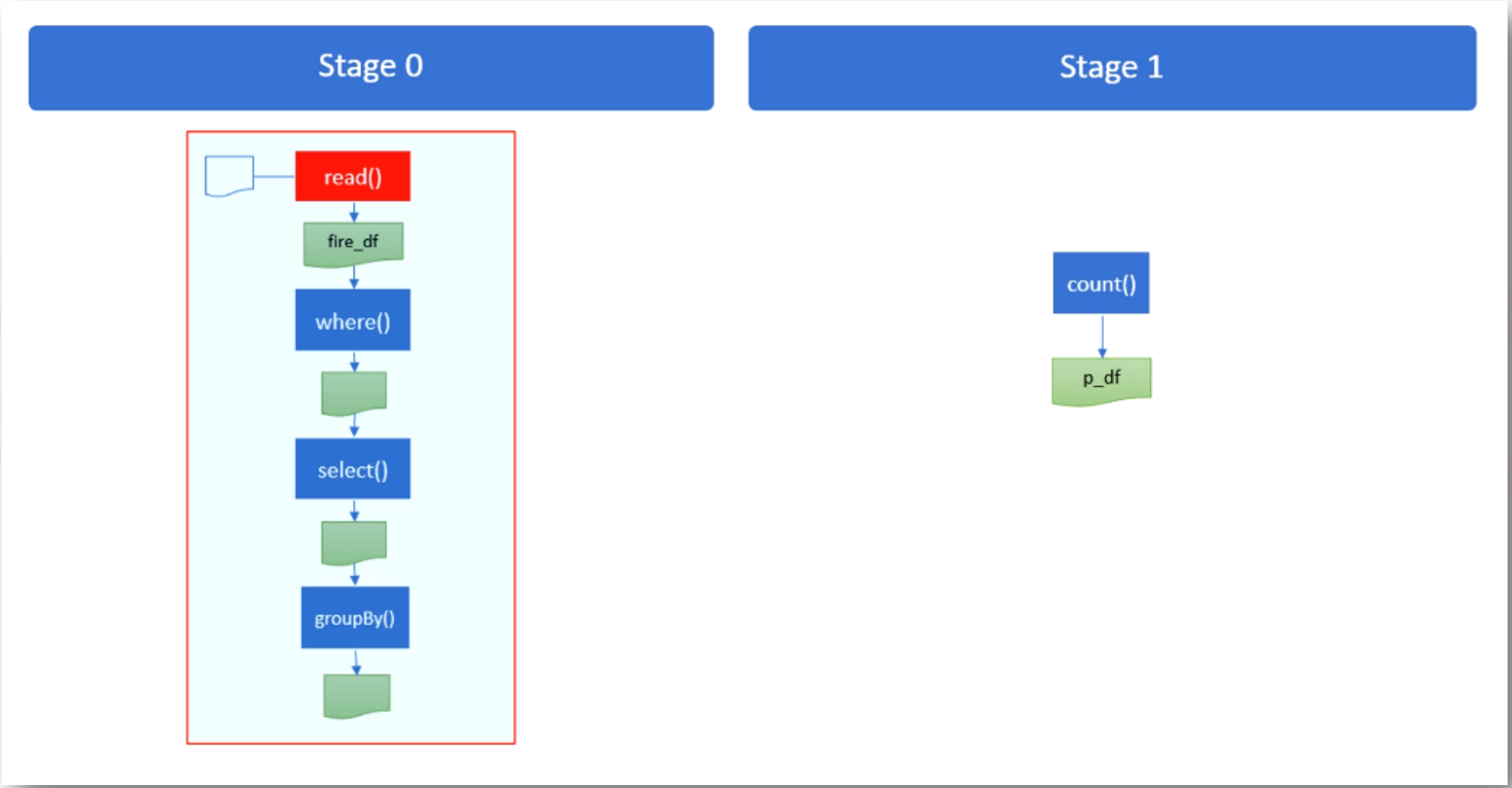


So a Job is broken down into stages using the wide dependency transformation. Each wide dependency breaks your job into two stages. In my example, I have only one wide dependency, So I got two stages. But if you have two wide dependencies, you will see three stages.
The calculation is straight. If you have N-wide dependencies, you will see N+1 stages.

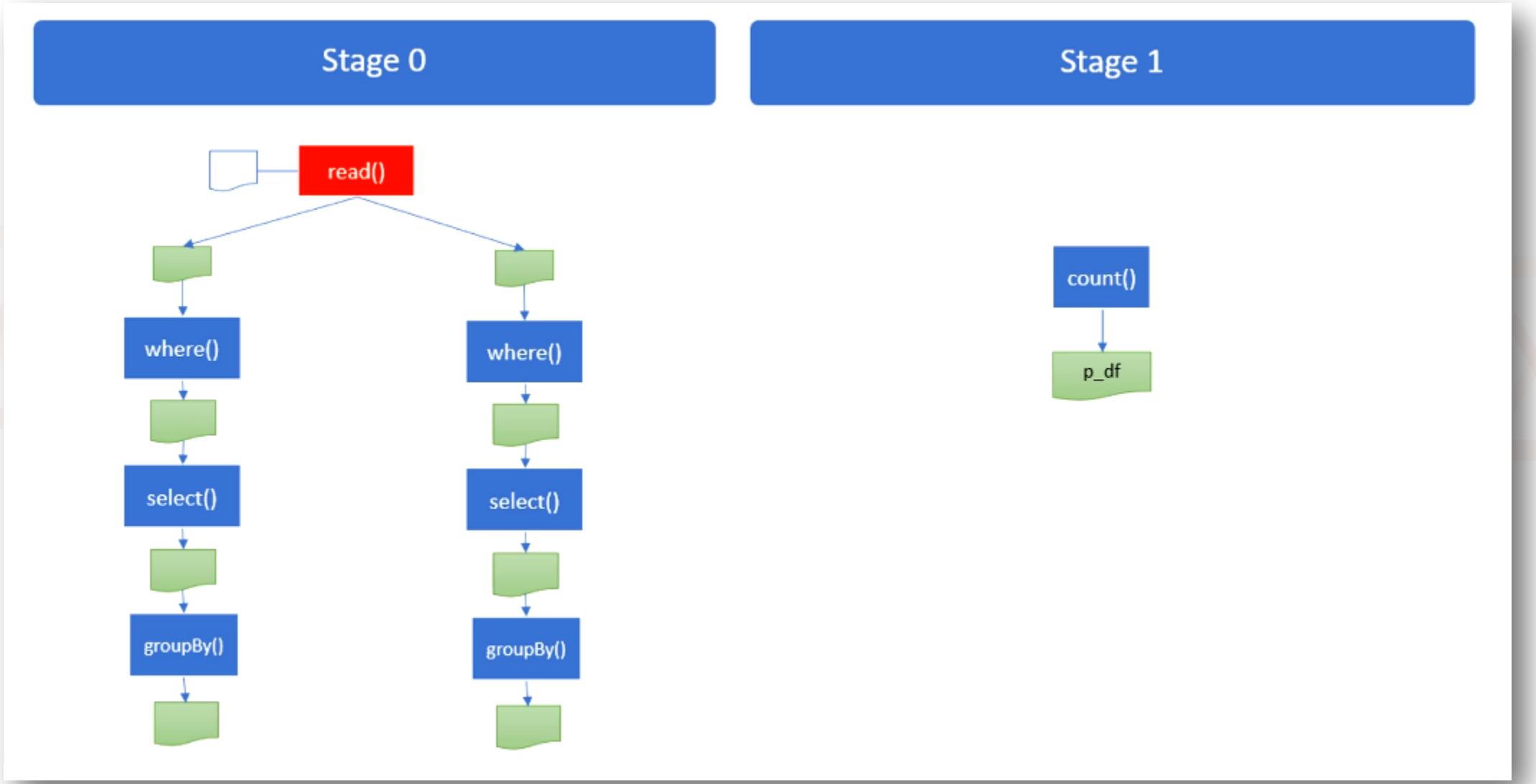


My logical plan is now broken down into two stages. Each stage can have one or more narrow transformations, and the last operation of the stage is a wide dependency transformation.
Spark cannot run these stages in parallel. We should finish the first stage, and then only we can start the next stage. Why? Because the output of the first stage is an input for the next stage. So whatever we see here is one spark job, broken down into two stages. These stages can run one after the other because the output of one stage is input for the next stage.

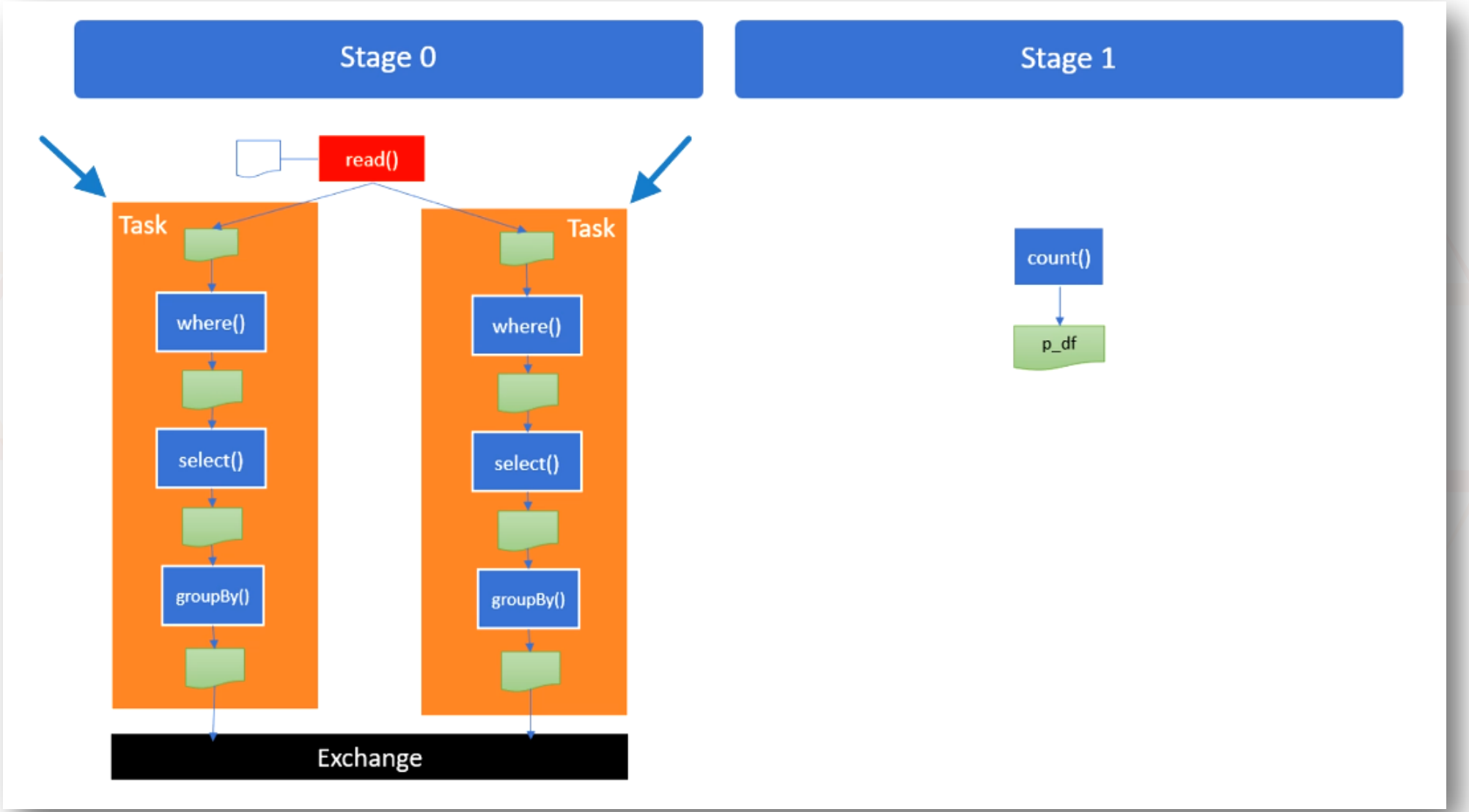
But how the output of one stage goes to the next stage? Let's try to understand it by tracing it from the first stage. What am I doing in the first stage? I am reading a file and creating fire_df.



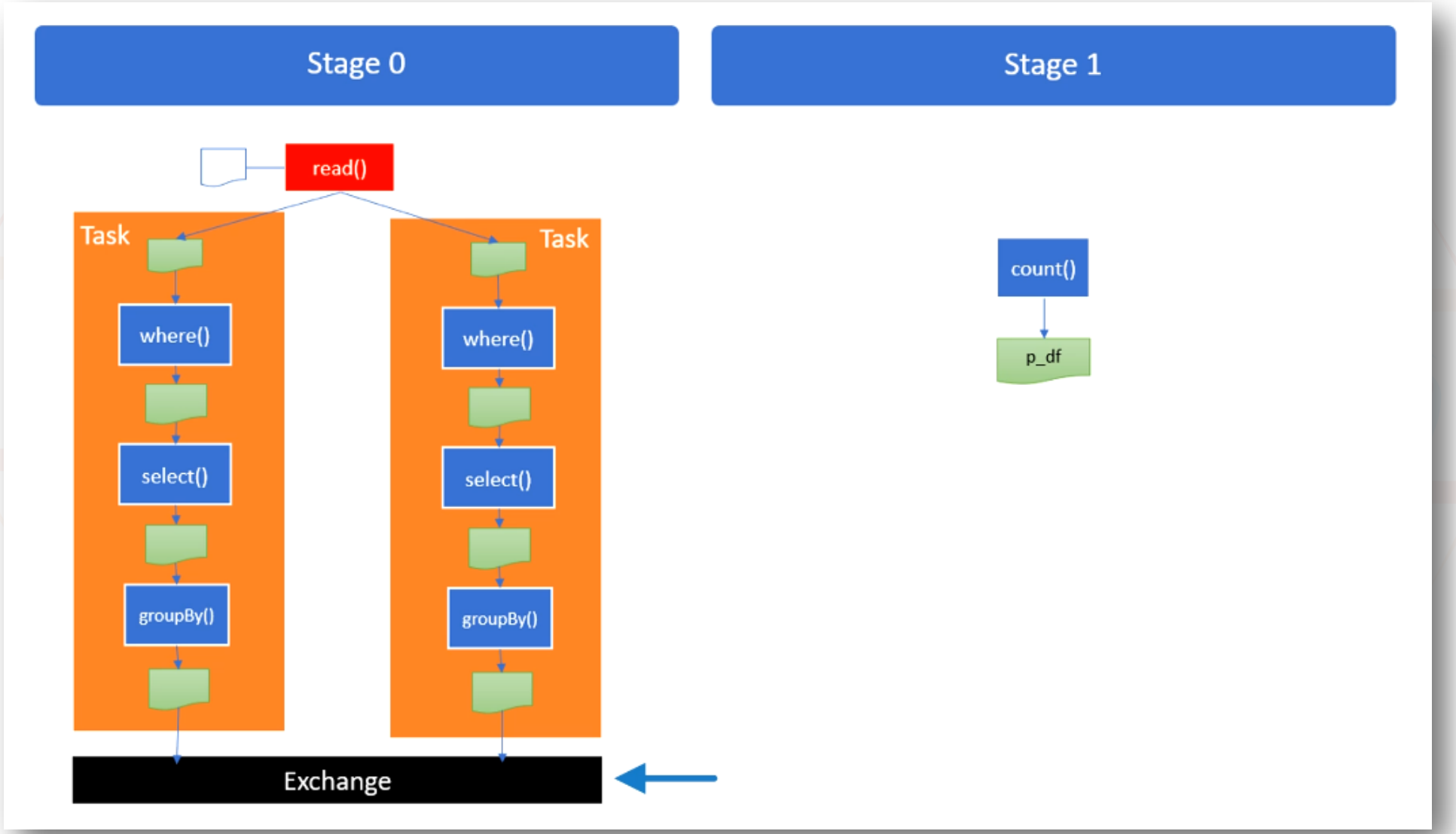
Let's assume I have two partitions, just for the simplicity.
So it should look like this shown in Stage 0. So this entire thing chain of transformation is applied to both the partitions.



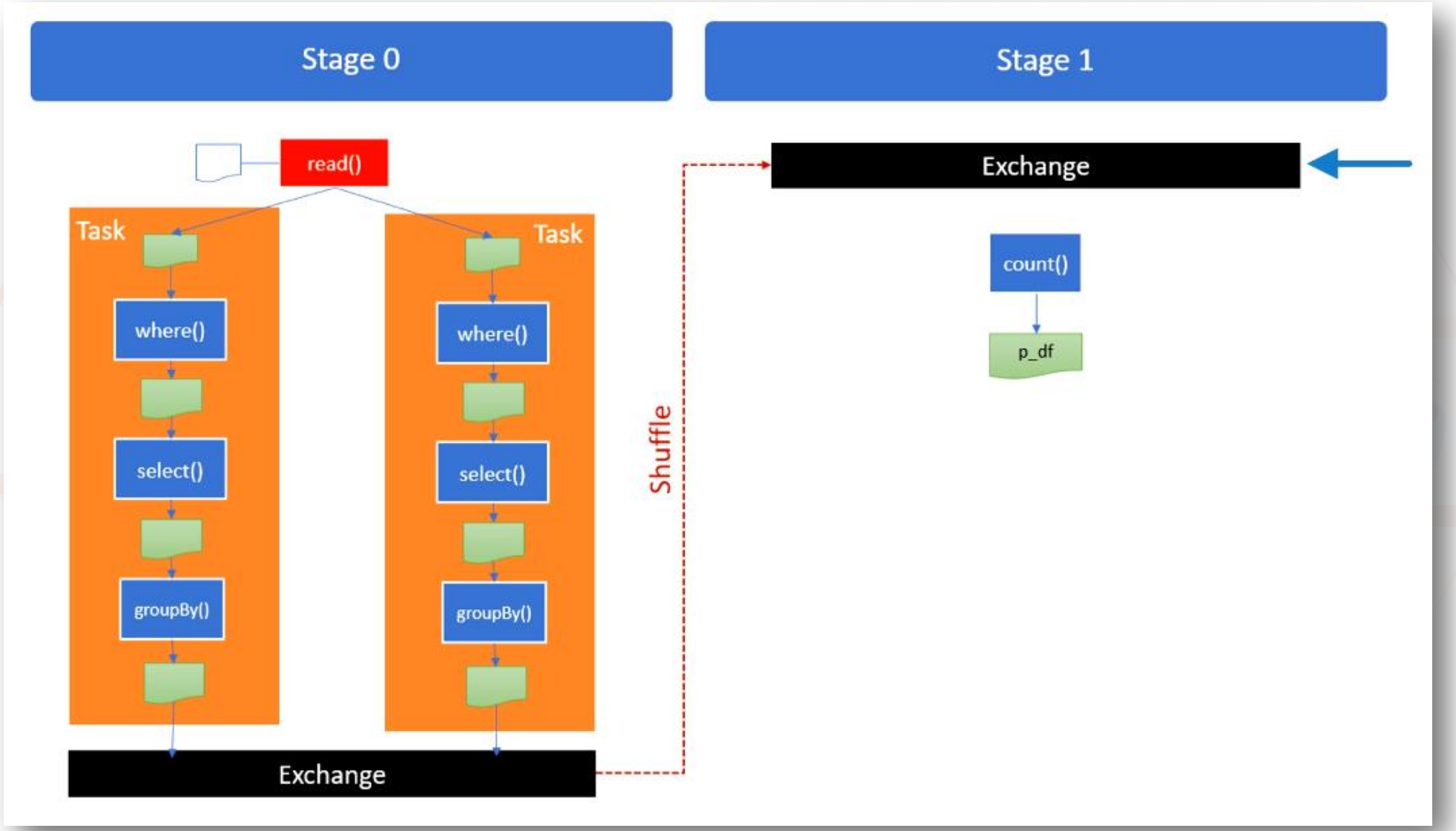
Each of these flows becomes a task.



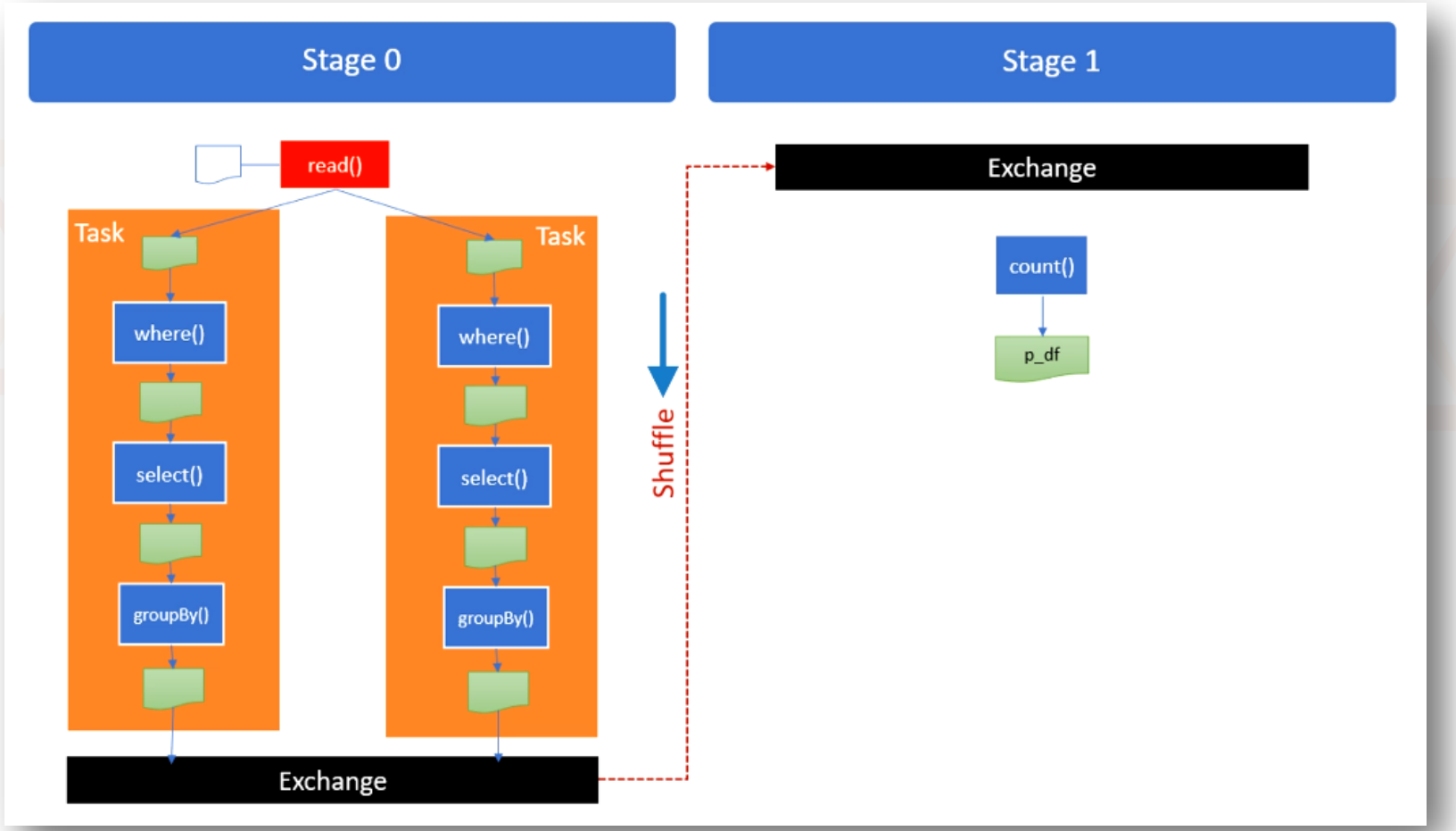
And the final output of the stage must be stored in an exchange buffer. Now the output of one stage becomes the input of the next stage.



So the next stage starts with the exchange buffer.

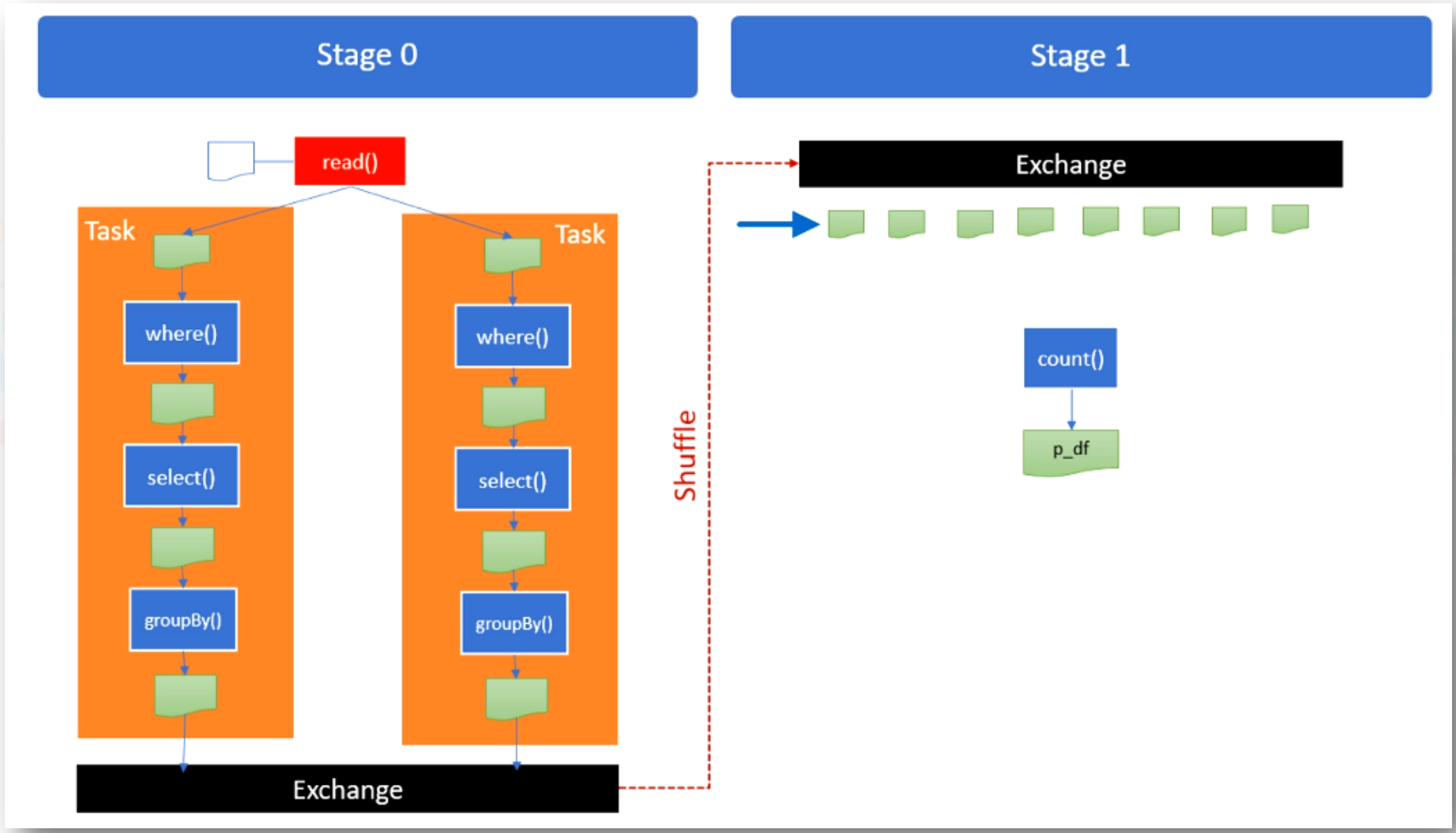


Spark is a distributed system. So the Write exchange and the read exchange may be on two different worker nodes. It may be on the same worker, but it can be on two different works. So we must consider a copy of data partitions from the Write exchange to the read exchange. And this copy operation is popularly known as the Shuffle operation. The shuffle/ is not a plain copy of data. Many things happen here, but you do not need to worry about everything.

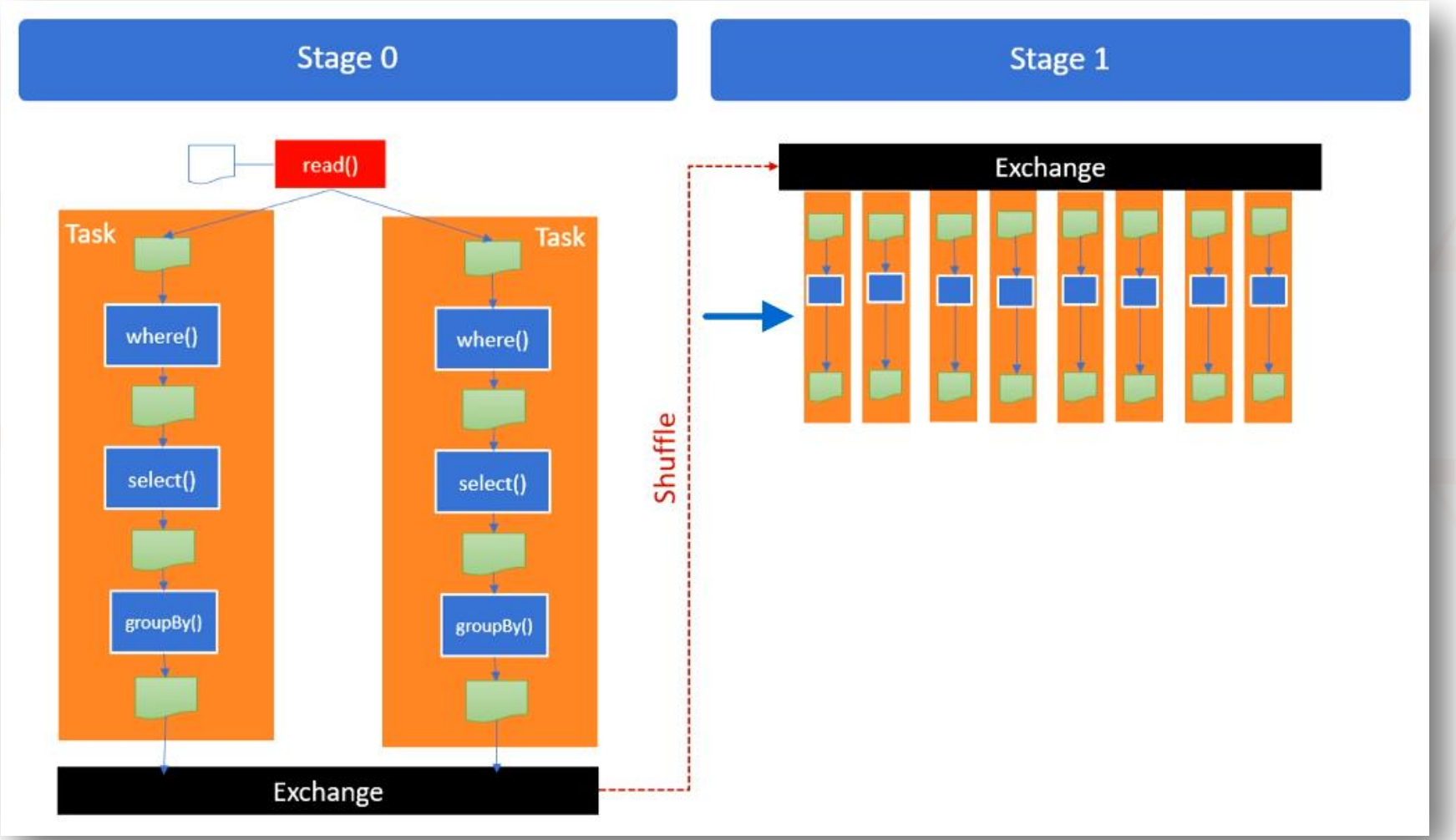


But do remember some points here.
The stage ends with a wide dependency transformation, requiring a data shuffle.
The Shuffle is an expensive operation in the Spark cluster.
It requires a write exchange buffer and a read exchange buffer.
The data from the write exchange buffer is sent to the read exchange buffer over the network.

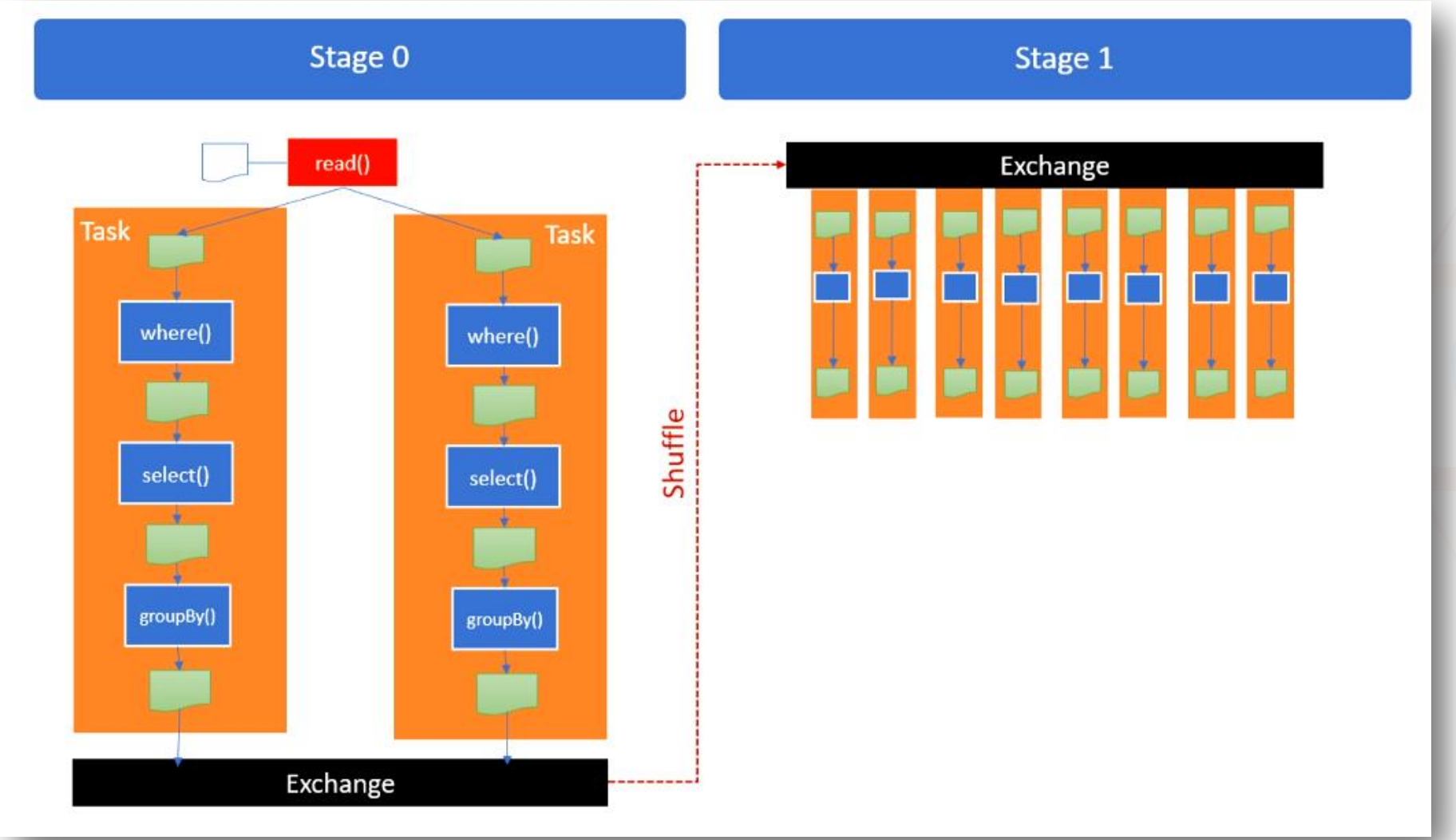
Now let's come to the second stage.
The read exchange has got eight partitions. Why? Because I have eight unique priority values and configured the eight shuffle partitions.



And I can run these transformations in parallel on those eight partitions, as shown below. Spark can execute the same plan in parallel on eight partitions because I have eight partitions. And this is what we call a task. So I can have eight parallel tasks in stage two.



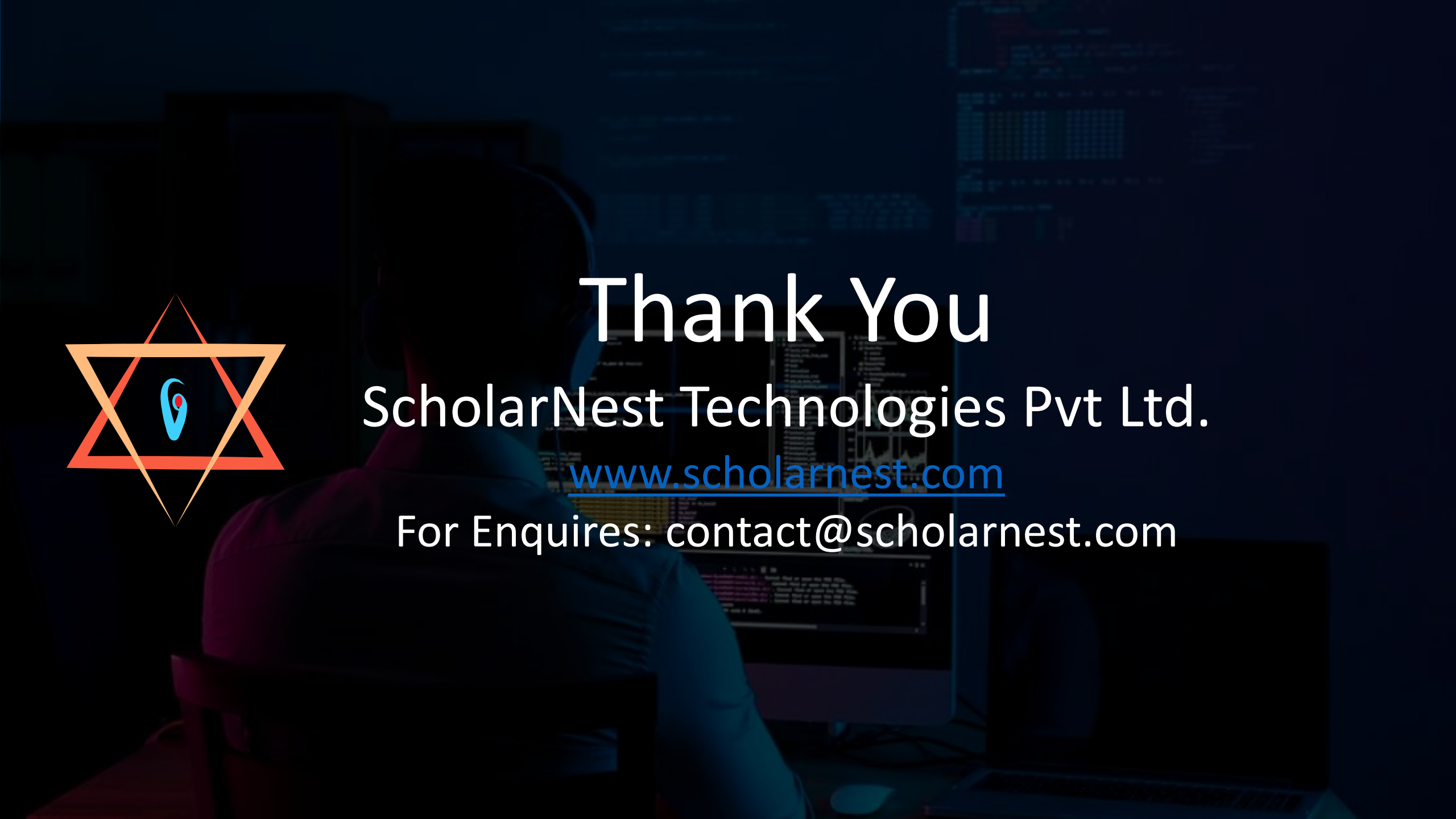
And this is what the final execution plan looks like.
So Spark starts with a logical query plan of a job and converts it to a runtime execution plan.
And this is how the runtime execution plan looks for my example spark job.



We learned about the following four topics in this chapter:

1. Job – Spark creates one job for each action. This job may contain a series of multiple transformations. The Spark engine will analyze those transformations and create a logical plan for the job.
2. Stage – Then spark will break the logical plan at the end of every wide dependency and create two or more stages. If you do not have any wide dependency, your plan will be a single-stage plan. But if you have N wide-dependencies, your plan should have N+1 stages.
3. Shuffle – Data from one stage to another stage is shared using the shuffle operation.
4. Task – Now each stage may be executed as one or more parallel tasks. The number of tasks in the stage equals the number of input partitions. In our example, the first stage starts with two partitions. So it can have a maximum of two parallel tasks. We made sure eight partitions in stage two. So we can have eight parallel tasks in stage two. If I create 100 partitions for stage two, I can have 100 parallel tasks for stage two.

The task is the most critical concept for a Spark job.
We learned all this to reach the task level and understand the task.
A task is the smallest unit of work in a Spark job.
The Spark driver assigns these tasks to the executors and asks them to do the work in parallel.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark
Dataframe
Internals

Lecture:
SQL Engine
And
Query
Planning

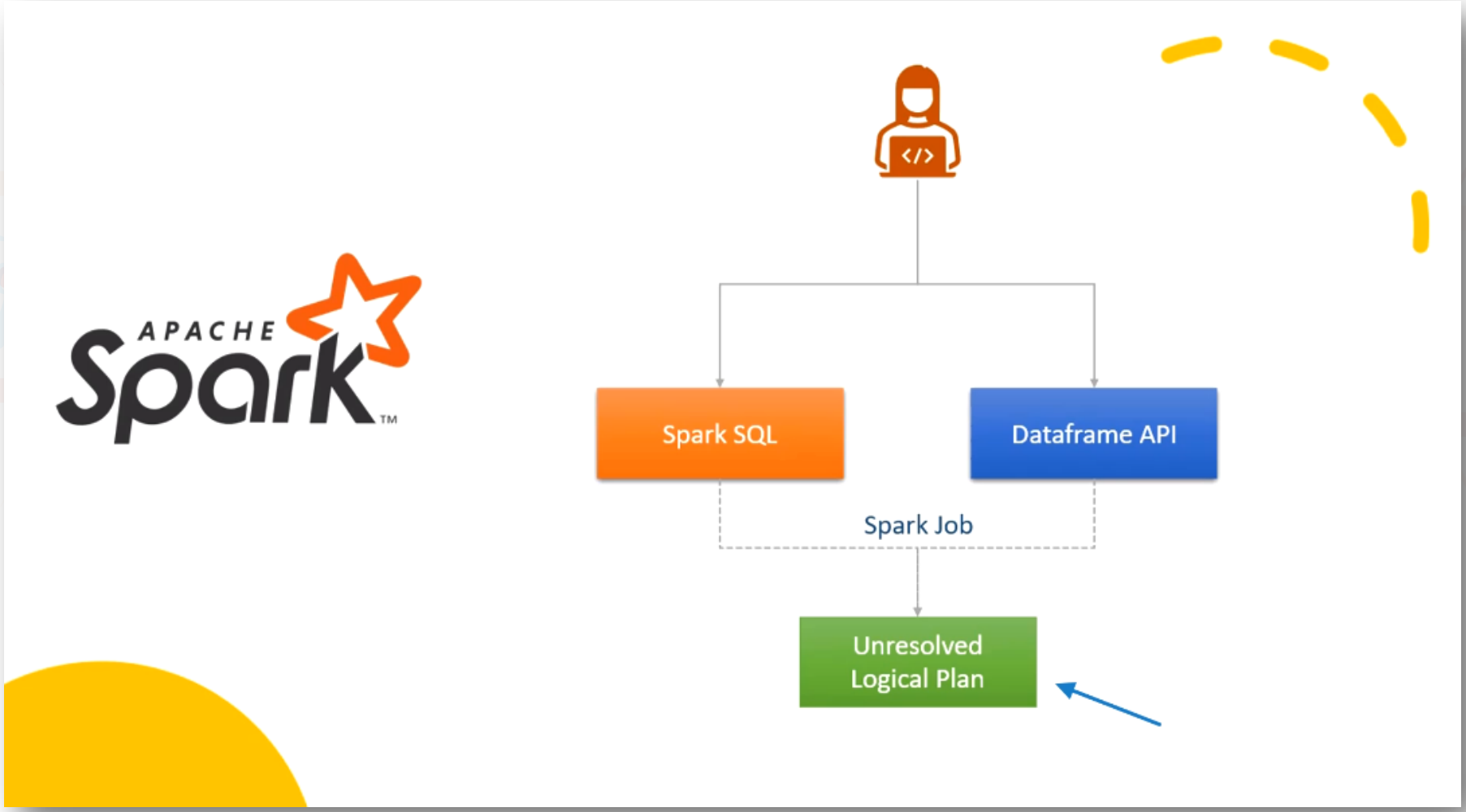
Spark SQL Engine and Query Planning

Apache Spark gives you two prominent interfaces to work with data:

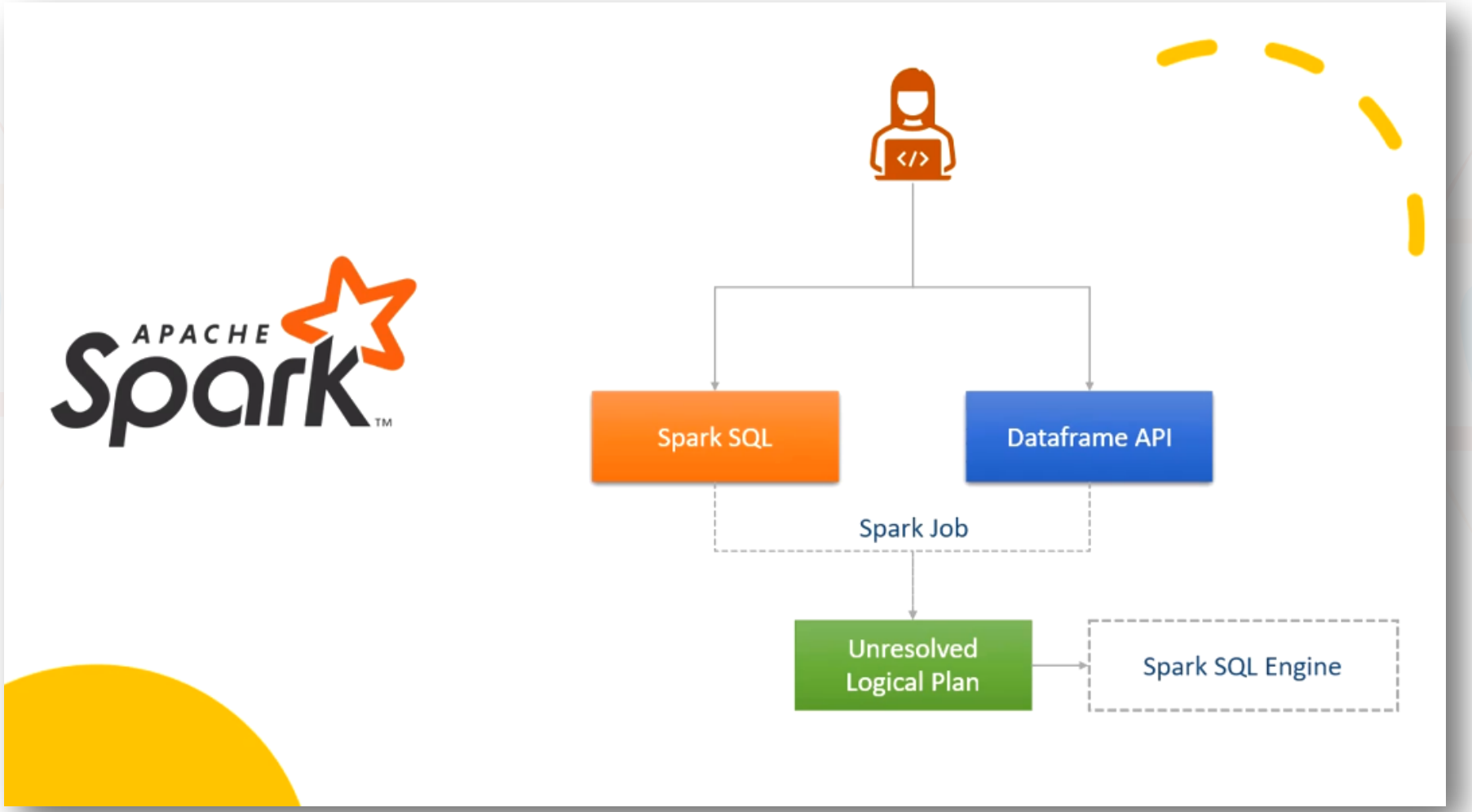
1. Spark SQL
2. Dataframe API

Spark SQL is compliant with ANSI SQL:2003 standard. So you can think of it as standard SQL.
Dataframes are function APIs, and they allow you to implement functional programming techniques to process your data.
Other than these two, you also have Dataset API available only for Scala and Java.
The Dataframe API internally uses Dataset APIs, but these Dataset APIs are not available in PySpark.
If you are using Scala or Java, you can directly use Dataset APIs. But Spark recommends using Dataframe APIs and avoid using Dataset. You can do almost everything using Dataframes and might not even see a requirement that forces you to use datasets.
So you can safely ignore it and let me remove the Dataset from the equation for brevity.

Spark looks at your code in terms of jobs. So if you have a Dataframe API code, Spark will take action and all its preceding transformations to consider it a single Job. Similarly, if you write a SQL expression, Spark considers one SQL expression as one Job. Spark code is nothing but a sequence of Spark Jobs. And each Spark Job represents a logical query plan.

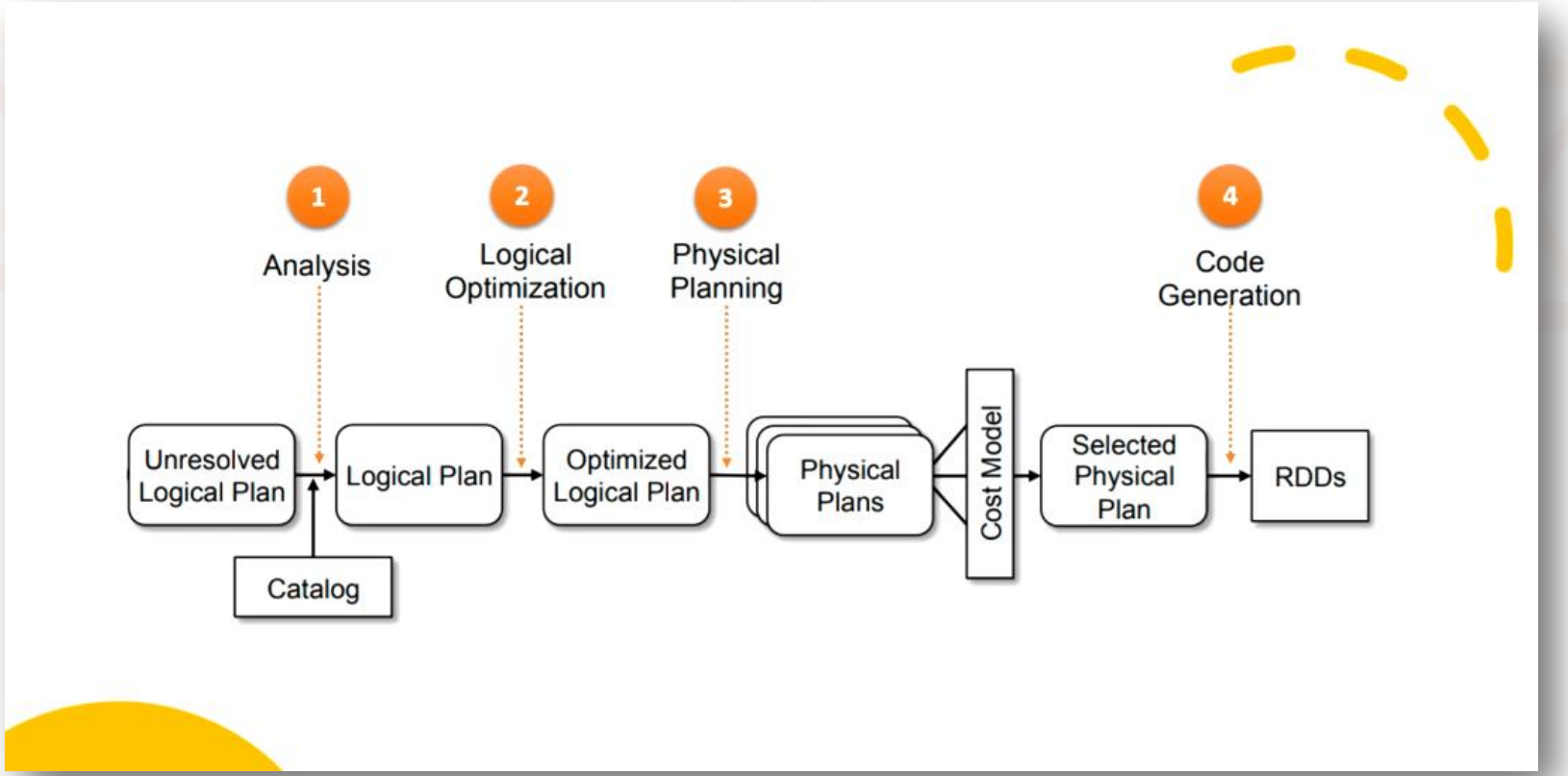


This first logical plan is a user-created logical query plan. Now this plan goes to Spark SQL engine. Remember! You may have Dataframe APIs, or you may have SQL. Both will go to the Spark SQL engine. For Spark, they are nothing but a Spark Job represented as a logical plan.



The Spark SQL Engine will process your logical plan in four stages listed in the image below.

- The Analysis stage will parse your code for errors and incorrect names.
- The logical optimization phase applies standard rule-based optimizations to the logical plan.
- Spark SQL takes a logical plan and generates one or more physical plans in the physical planning phase. Physical planning applies cost-based optimization.
- Finally, your best physical plan goes into code generation, and the engine will generate Java byte code for the RDD operations in the physical plan.



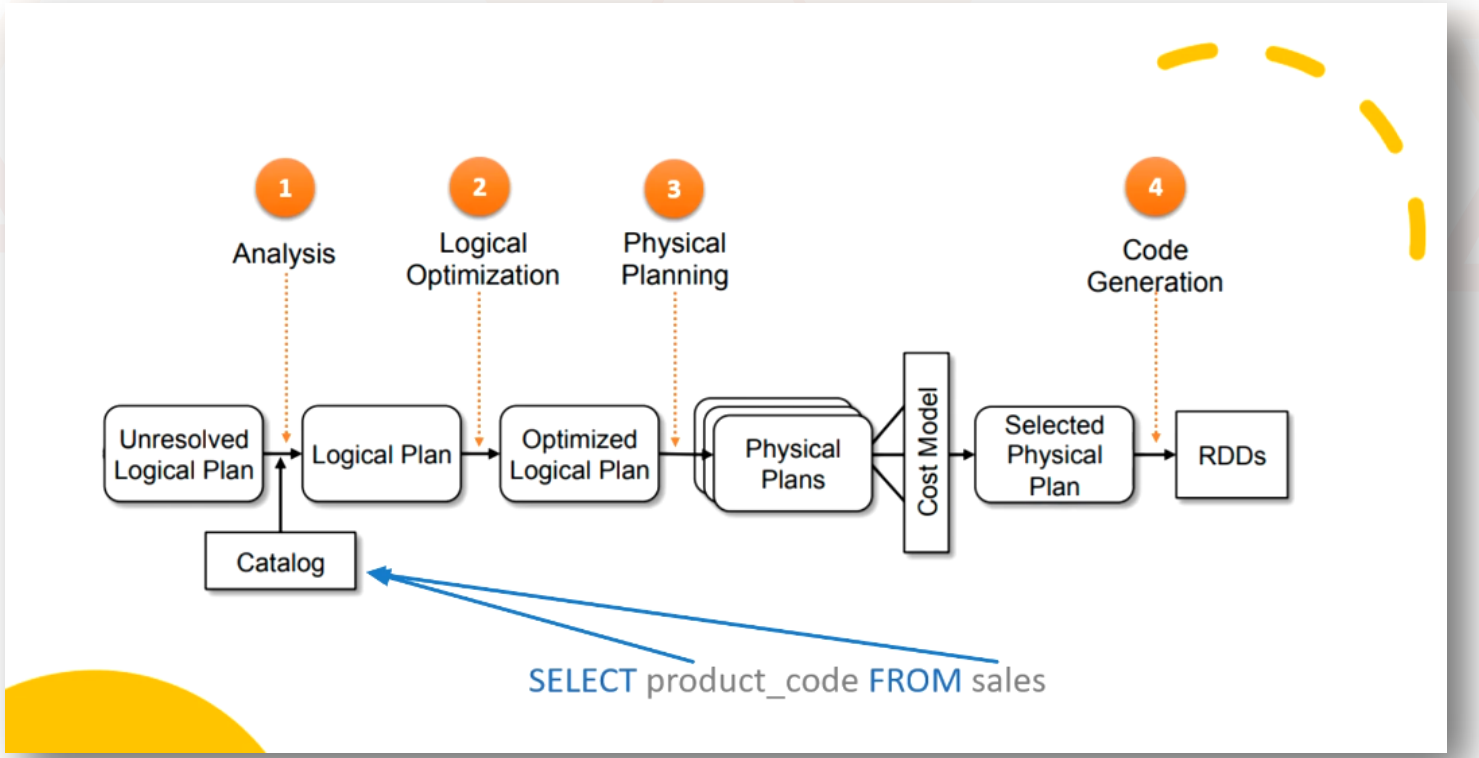
Let us go through each of the four stages one by one.

1. Analysis:

The Analysis stage will parse your code for errors and incorrect names. For example, let's assume your code represents the following SQL:

SELECT product_code FROM sales

Spark doesn't know if the column is a valid column name and the data type of this column. So the Spark SQL engine will look into the catalog to resolve the column name and its data type.

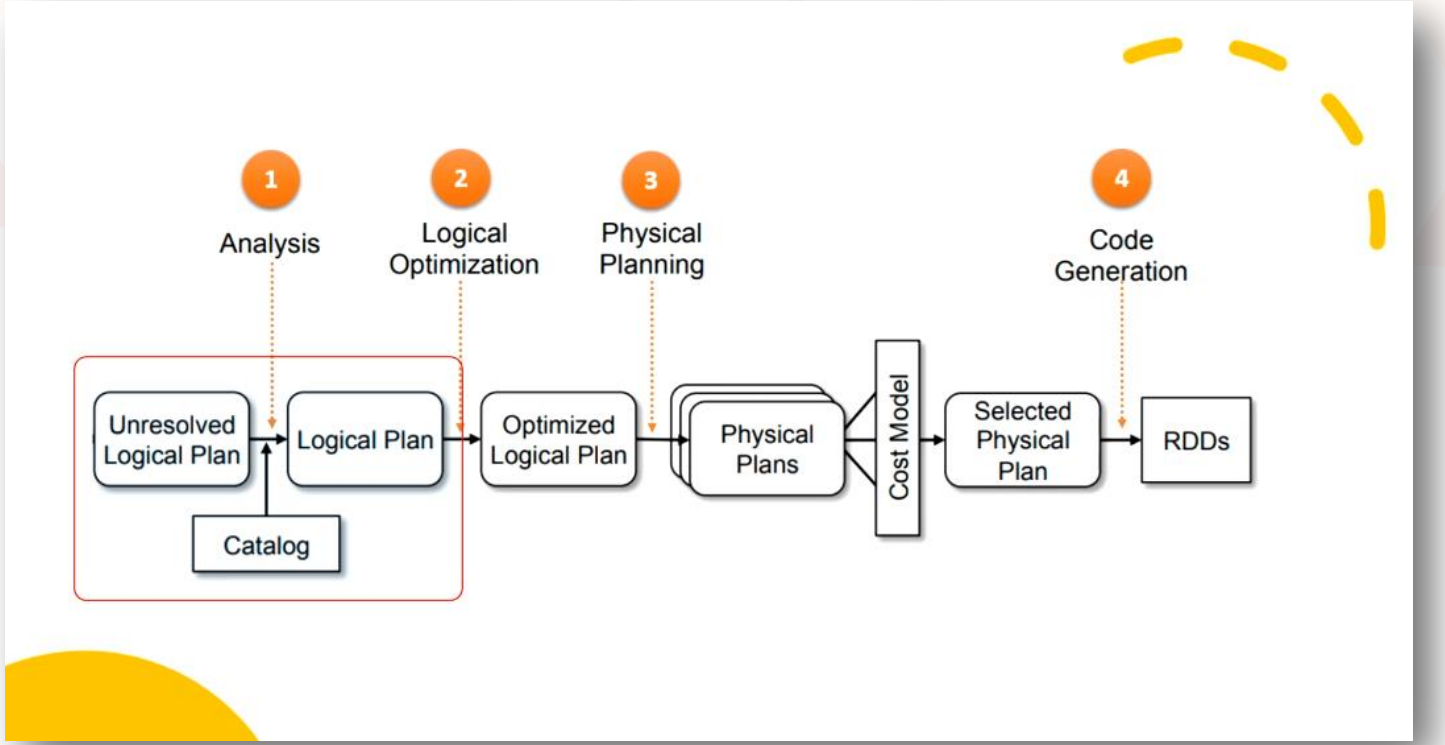


1. Analysis:

Now, assume you have a SQL like this:

SELECT product_qty + 5 FROM sales

Then Spark SQL engine might also apply an implicit type casting to the *product_qty* to perform and validate the addition operation. The Analysis phase will parse your code and create a fully resolved logical plan. You might see an Analysis exception if the column names do not resolve or have some incorrect typecasting, or you used an invalid function name, etc. If your code passed the Analysis phase, that means you have a valid code, and you are not using something that Spark doesn't know or cannot resolve.



2. Logical Optimization:

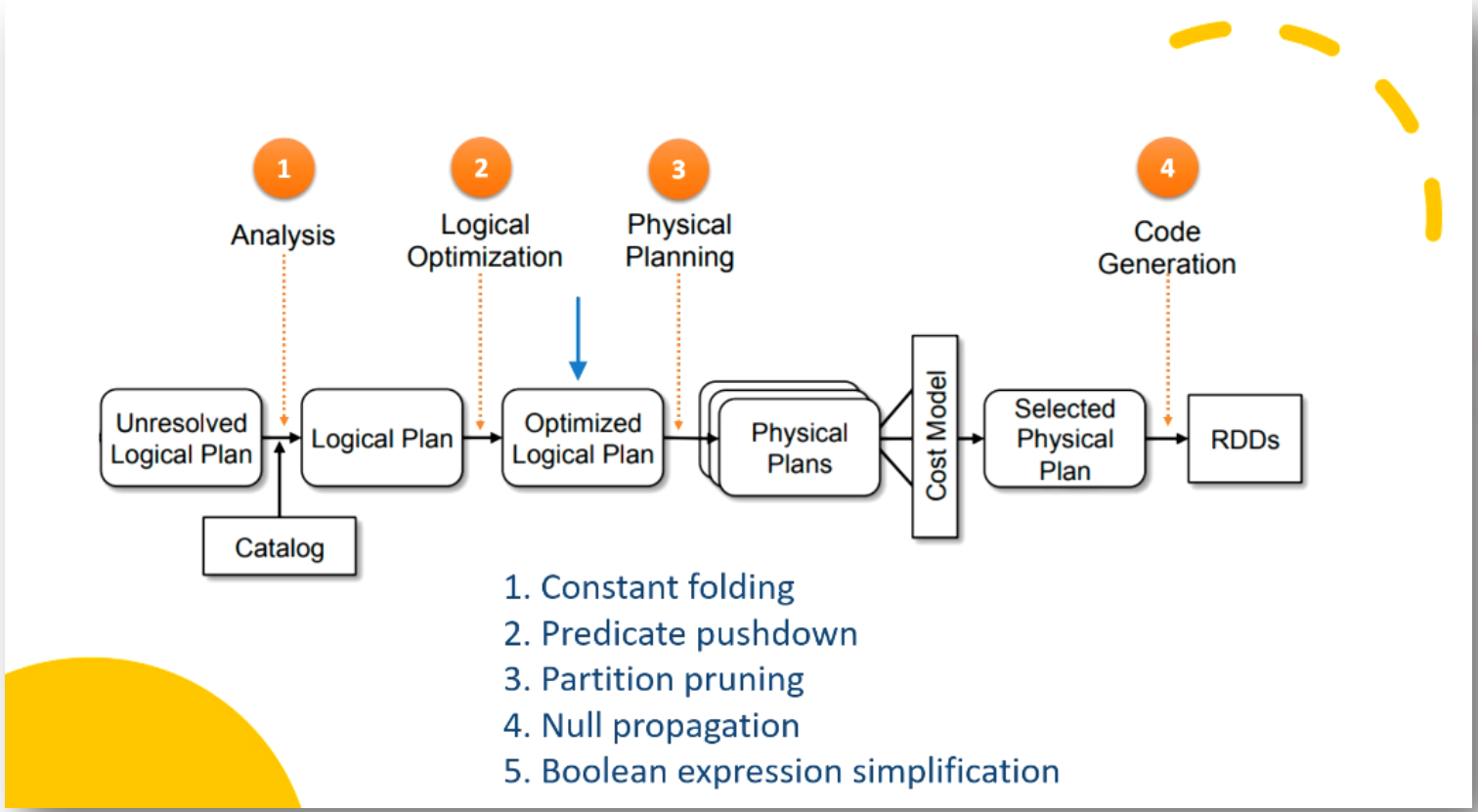
The next stage is logical optimization.

The logical optimization phase applies standard rule-based optimizations to the logical plan.

Here are some examples of logical optimization listed in the image below.

And the list of such logical optimizations is long. We do not care what do they include here.

All we want to learn is they apply some logical optimizations to our code and create an optimized logical plan.

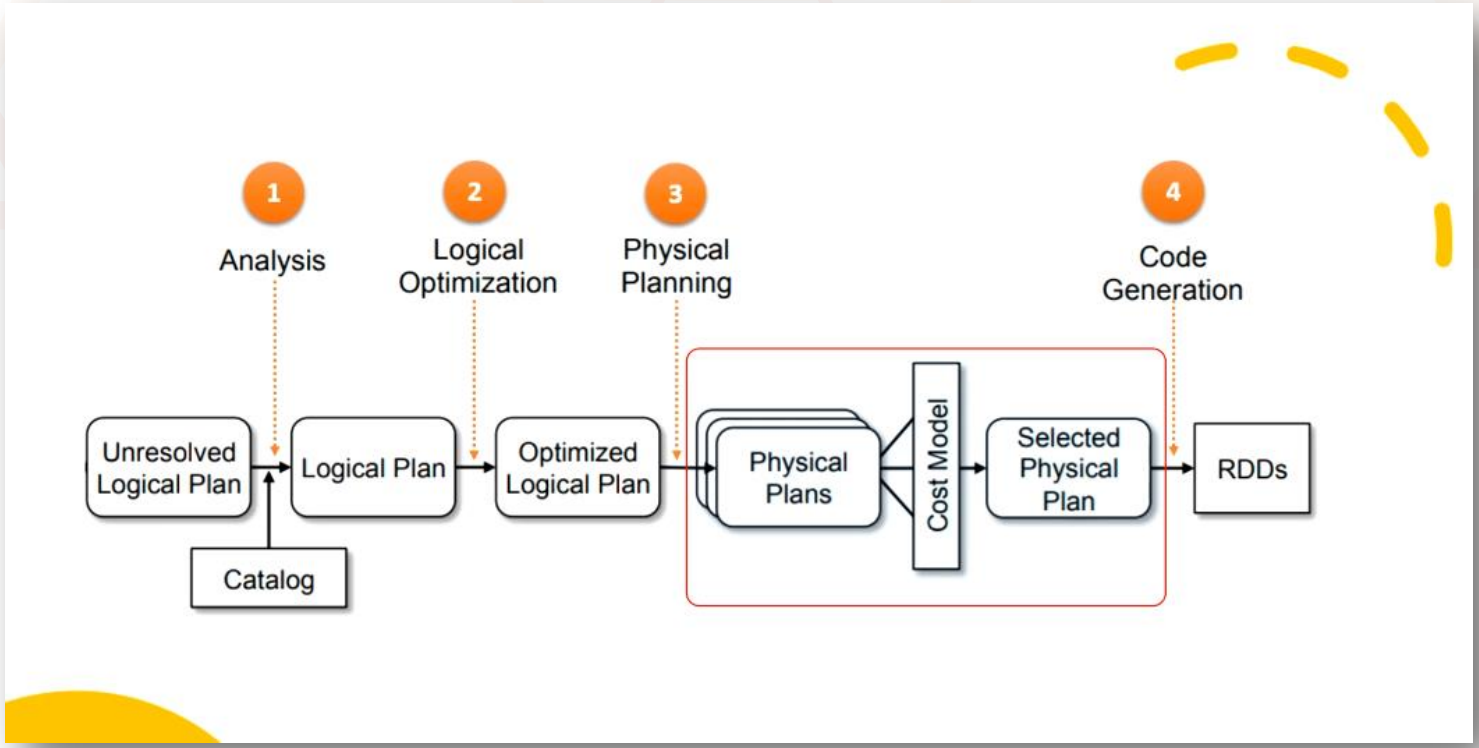


3. Physical Planning:

Then this optimized logical plan goes into the next phase of physical planning.

Spark SQL takes a logical plan and generates one or more physical plans in the physical planning phase. Physical planning applies cost-based optimization. So the engine will create multiple plans, calculate each plan's cost, and finally select the plan with the least cost. At this stage, they mostly use different join algorithms to create more than one physical plan.

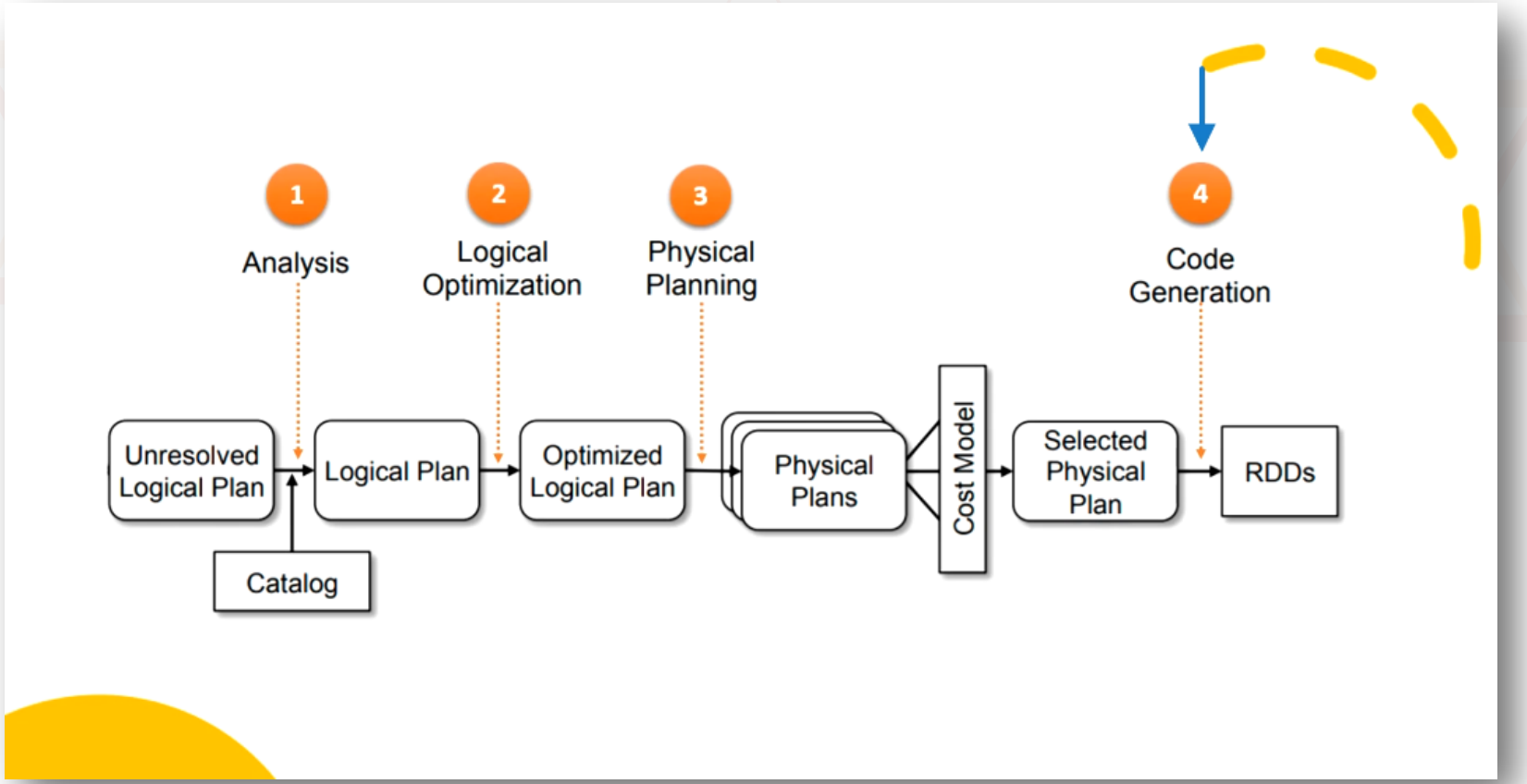
For example, they might create one plan using broadcast join and another using Sort merge, and one more using shuffle hash join. Then they apply a cost to each plan and choose the best one.

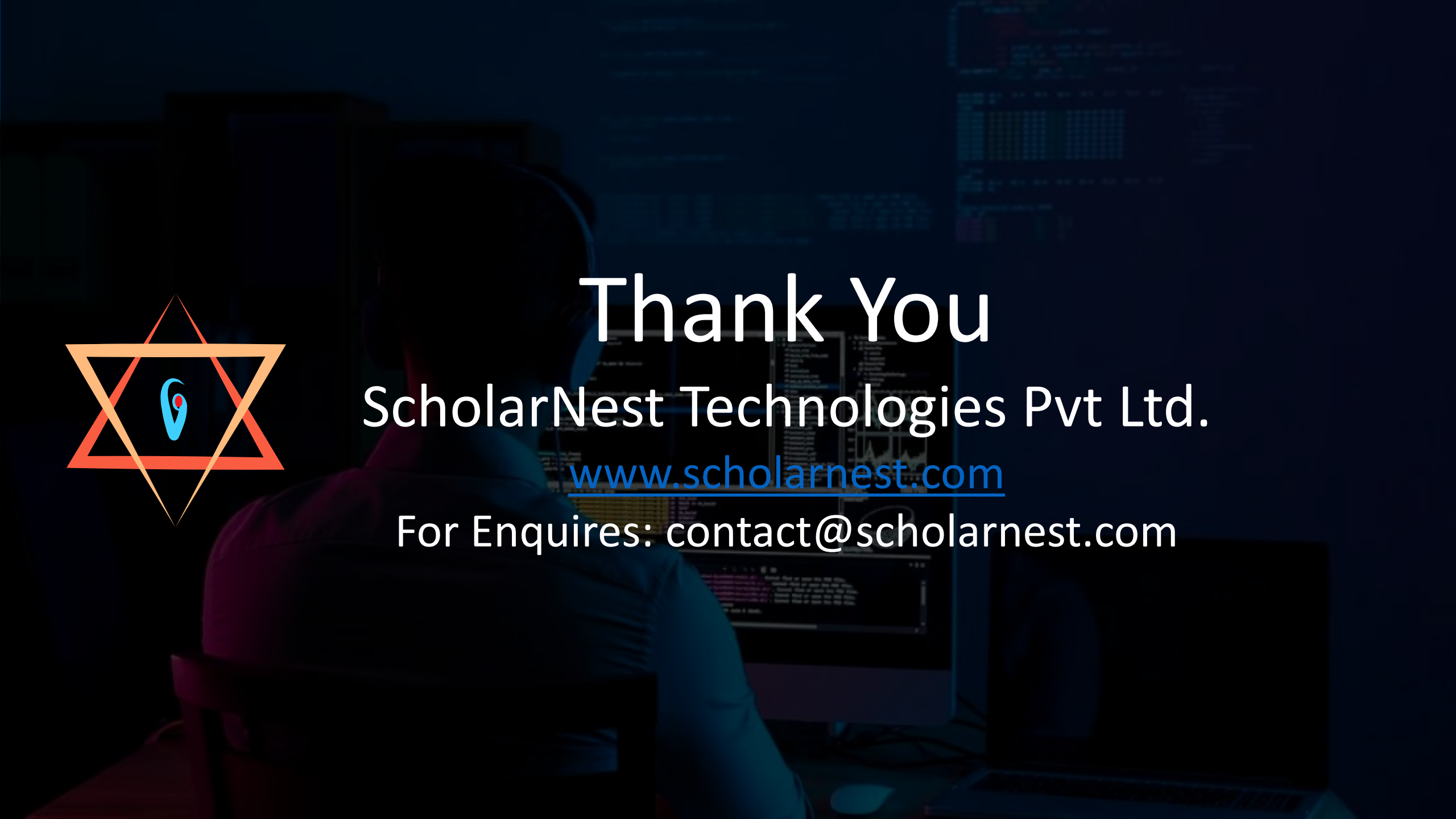


4. Code Generation:

The last stage is code generation.

So your best physical plan goes into code generation, and the engine will generate Java byte code for the RDD operations in the physical plan. And that's why Spark is also said to act as a compiler because it uses state-of-the-art compiler technology for code generation to speed up execution. And that's pretty much all about the Spark SQL Engine.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com