Spark Azure Databricks

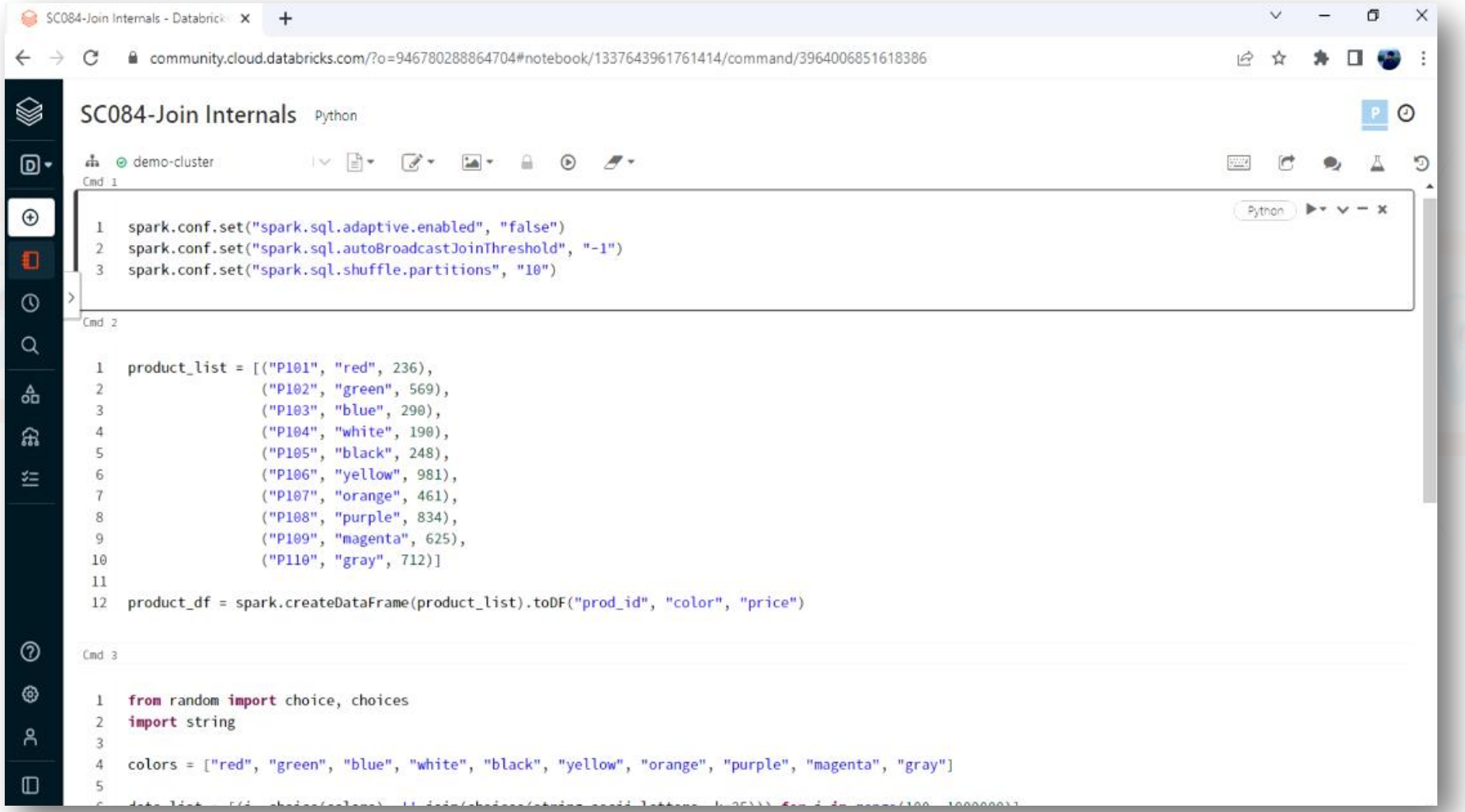
Databricks Spark Certification and Beyond

Module:
Spark Joins and Optimization

Lecture:
Internals of Spark Joins and Shuffle

Internals of Spark Joins and Shuffle

You already learned various join types in Spark. Let me show you an example of how to apply joins in Spark. Here is a notebook **(Reference: SC084-Join Internals)**, and I already have some code here to create two data frames and join them.



You will see some code to create the product_df data frame. You can see the product_df columns here. So I have three columns: product_id, color, and price.

SC084-Join Internals Python

demo-cluster

Cmd 2

```
product_list = [("P101", "red", 236),
                ("P102", "green", 569),
                ("P103", "blue", 290),
                ("P104", "white", 190),
                ("P105", "black", 248),
                ("P106", "yellow", 981),
                ("P107", "orange", 461),
                ("P108", "purple", 834),
                ("P109", "magenta", 625),
                ("P110", "gray", 712)]

product_df = spark.createDataFrame(product_list).toDF("prod_id", "color", "price")
```

Cmd 3

```
from random import choice, choices
import string

colors = ["red", "green", "blue", "white", "black", "yellow", "orange", "purple", "magenta", "gray"]

data_list = [(i, choice(colors), ''.join(choices(string.ascii_letters, k=25))) for i in range(100, 1000000)]

orders_df = spark.createDataFrame(data_list).toDF("orde_id", "color", "discription")
```

Cmd 4

Then I have some more code below to create an order data frame. You can also see the orders_df dataframe columns here. So we have order_id, color, and description.

SC084-Join Internals Python

demo-cluster

```
10                ("P110", "gray", 712)]
11
12  product_df = spark.createDataFrame(product_list).toDF("prod_id", "color", "price")
```

Cmd 3

```
1  from random import choice, choices
2  import string
3
4  colors = ["red", "green", "blue", "white", "black", "yellow", "orange", "purple", "magenta", "gray"]
5
6  data_list = [(i, choice(colors), ''.join(choices(string.ascii_letters, k=25))) for i in range(100, 1000000)]
7
8  orders_df = spark.createDataFrame(data_list).toDF("orde_id", "color", "discription")
```

Cmd 4

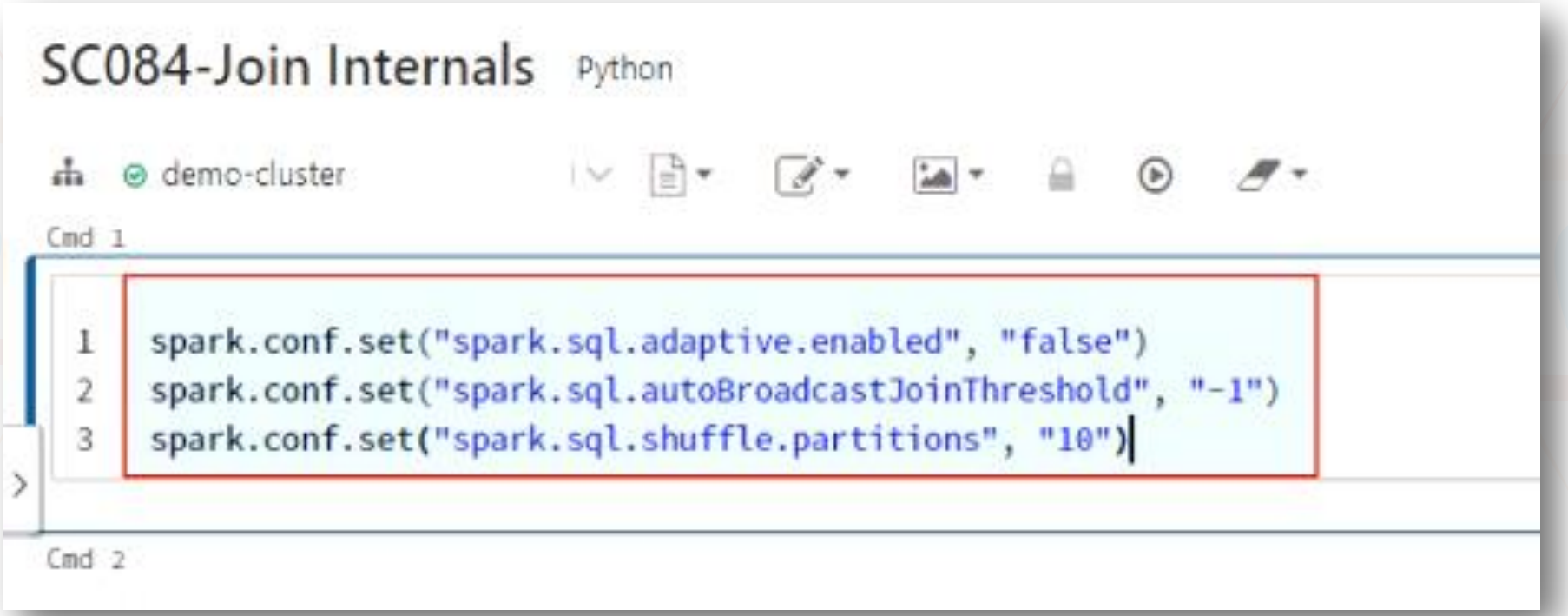
```
1  result_df = orders_df.join(product_df, "color", "inner")
2  result_df.write.format("parquet").mode("overwrite").saveAsTable("sales_tbl")
```


Now we want to join the product_df and the orders_df using the color column.
And I have code to join these two data frames. And after joining, I save the results as a table.
You already learned all this.

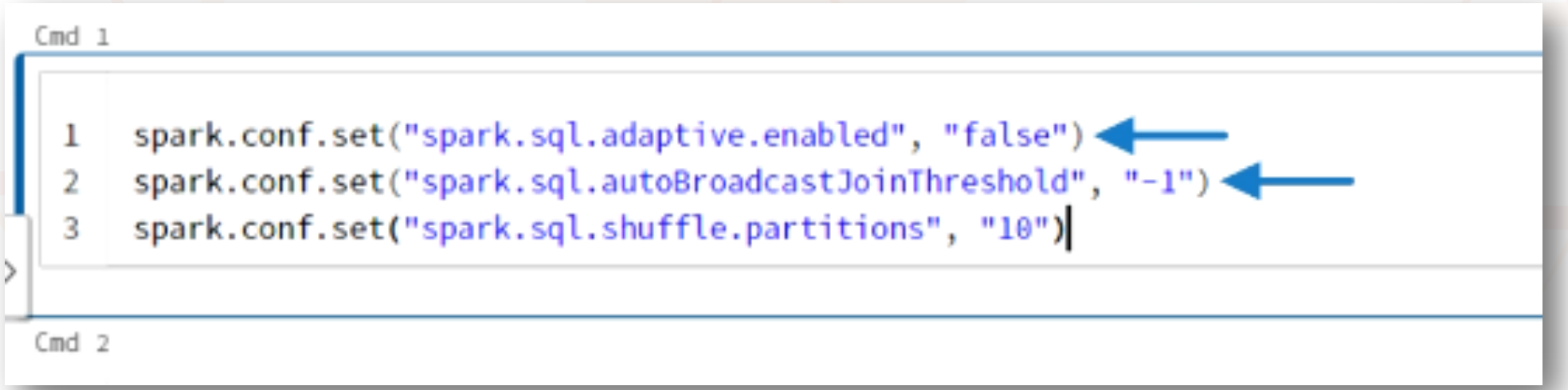
We know how to join two data frames, and this example applies a super simple join.
But what happens under the hood. How Spark applies a join?

```
Cmd 4

1  result_df = orders_df.join(product_df, "color", "inner")
2  result_df.write.format("parquet").mode("overwrite").saveAsTable("sales_tbl")

Shift+Enter to run
```

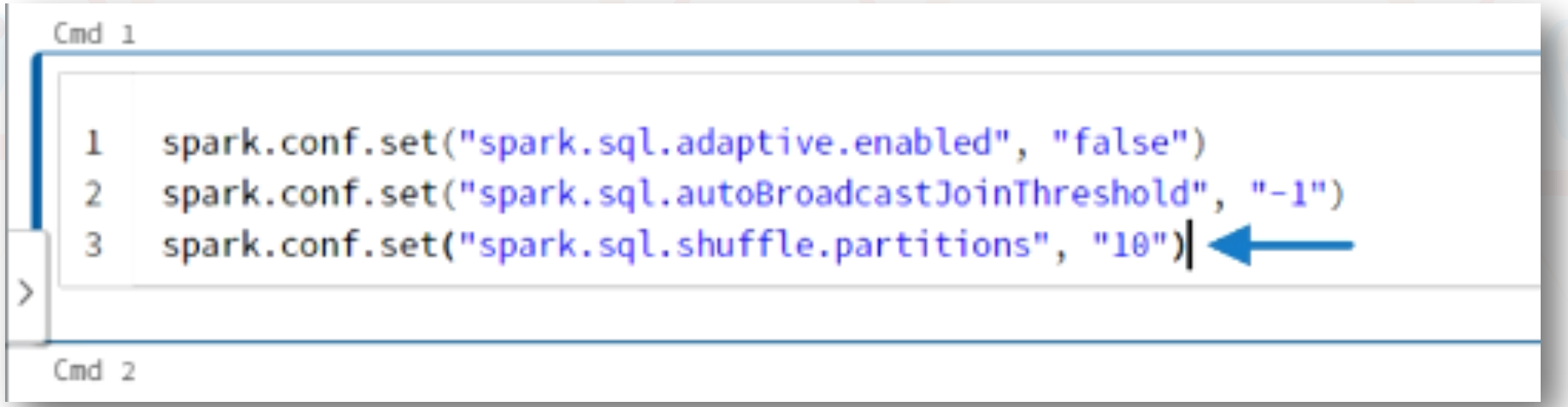

Let's try to understand that with the help of this super simple example.
I will go to the first cell and start running the code.
Here is the first cell highlighted below. I am setting some configurations in the first cell.



The first one will disable AQE, and the second one will disable broadcast join.
These two are automatic optimizations. I am disabling these two optimizations so we can understand how Join happens.
Optimizations might confuse us in understanding what happens inside.
So I am disabling it, and we will learn these optimizations in the coming lectures.



The last configuration is to set the shuffle partitions.
The default value is 200, but I am setting it to 10.
Why 10? Because I will have 10 unique join key values.
I already know that, so I am setting shuffle partitions to 10.
This setting will help us understand the internals of the Join operation.
Now, go ahead and run all the cells of your notebook.



So, I created two data frames, joined them, and finally displayed the result.
Now go to the spark UI and see what happened.

SC084-Join Internals Python

demo-cluster

```
colors = ["red", "green", "blue", "white", "black", "yellow", "orange", "purple", "magenta", "gray"]

data_list = [(i, choice(colors), ''.join(choices(string.ascii_letters, k=25))) for i in range(100, 1000000)]

orders_df = spark.createDataFrame(data_list).toDF("orde_id", "color", "discription")
```

Command took 33.90 seconds -- by prashant@scholarnest.com at 7/31/2022, 2:29:38 PM on demo-cluster

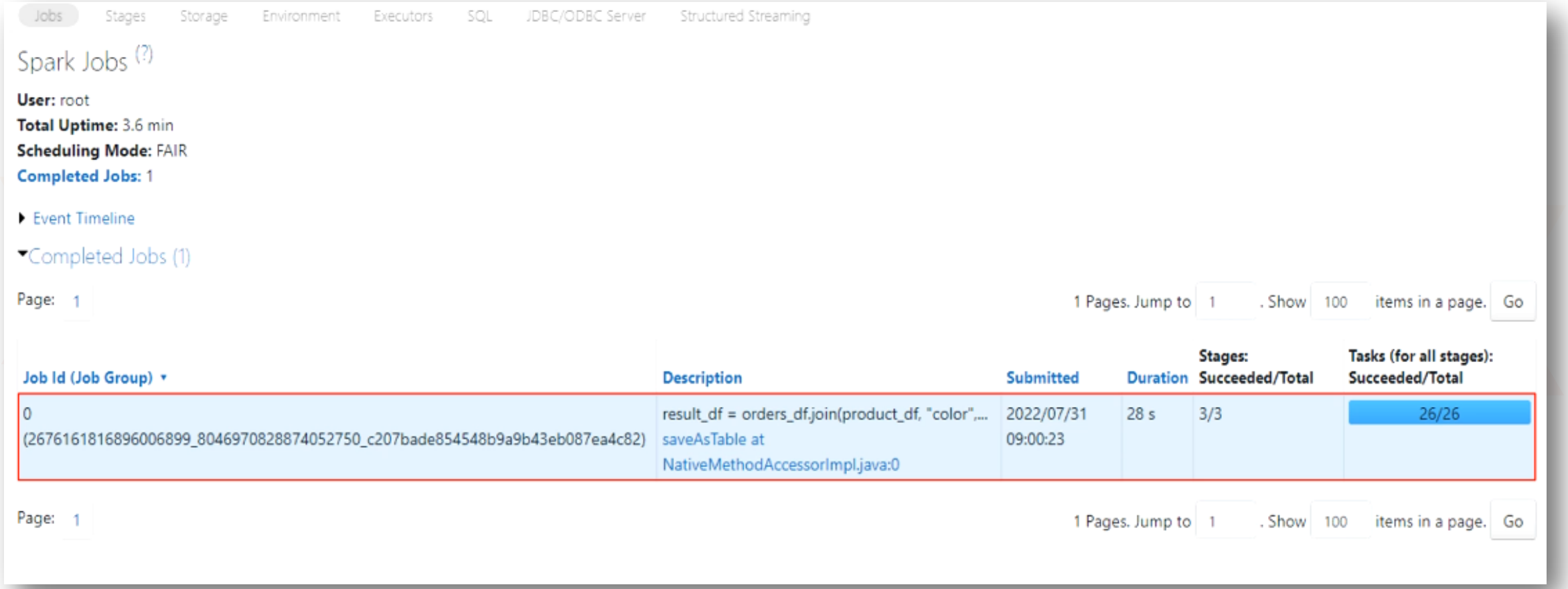
Cmd 4

```
result_df = orders_df.join(product_df, "color", "inner")
result_df.write.format("parquet").mode("overwrite").saveAsTable("sales_tbl")
```

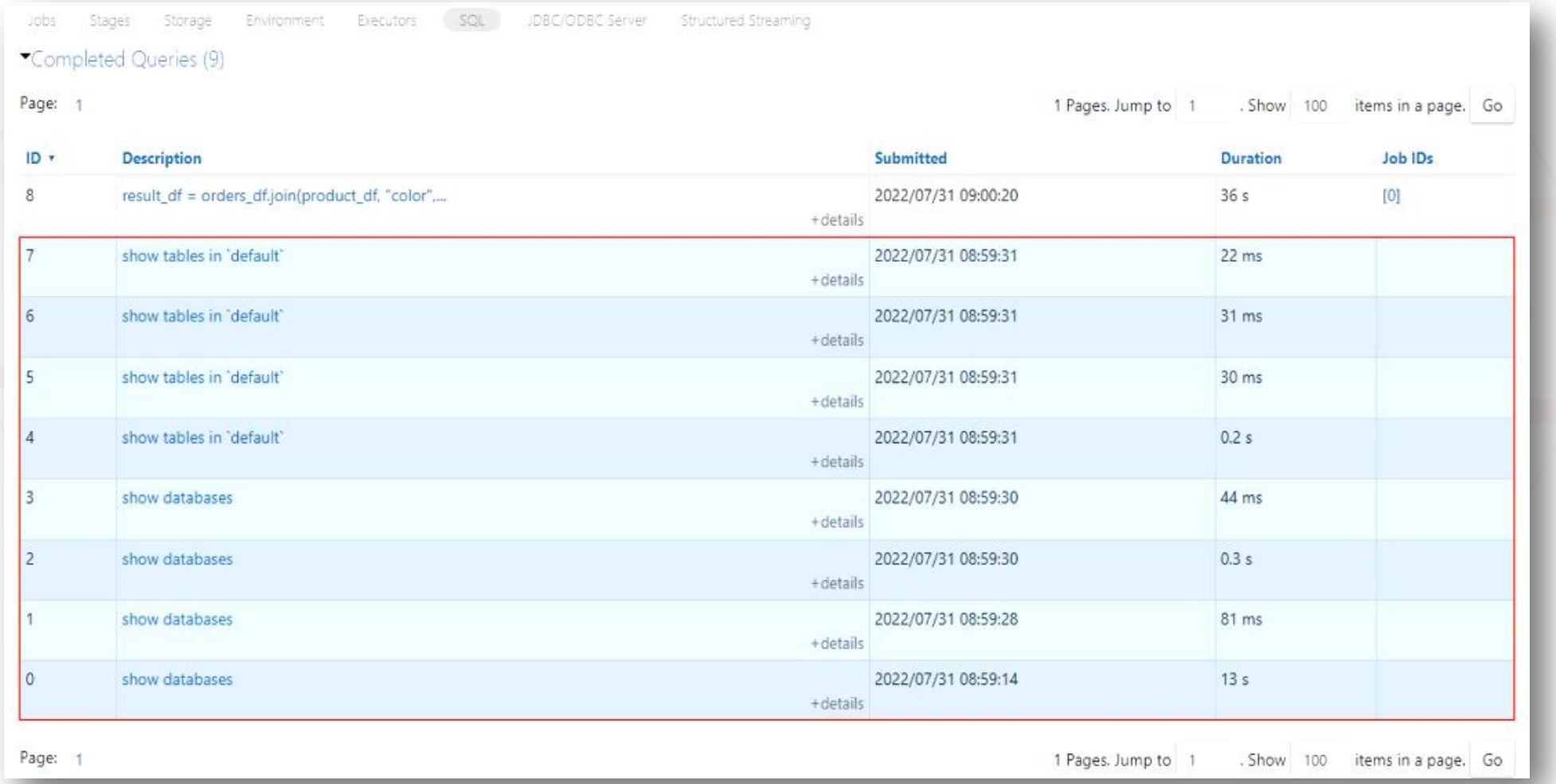
▸ (1) Spark Jobs

Command took 37.96 seconds -- by prashant@scholarnest.com at 7/31/2022, 2:30:18 PM on demo-cluster

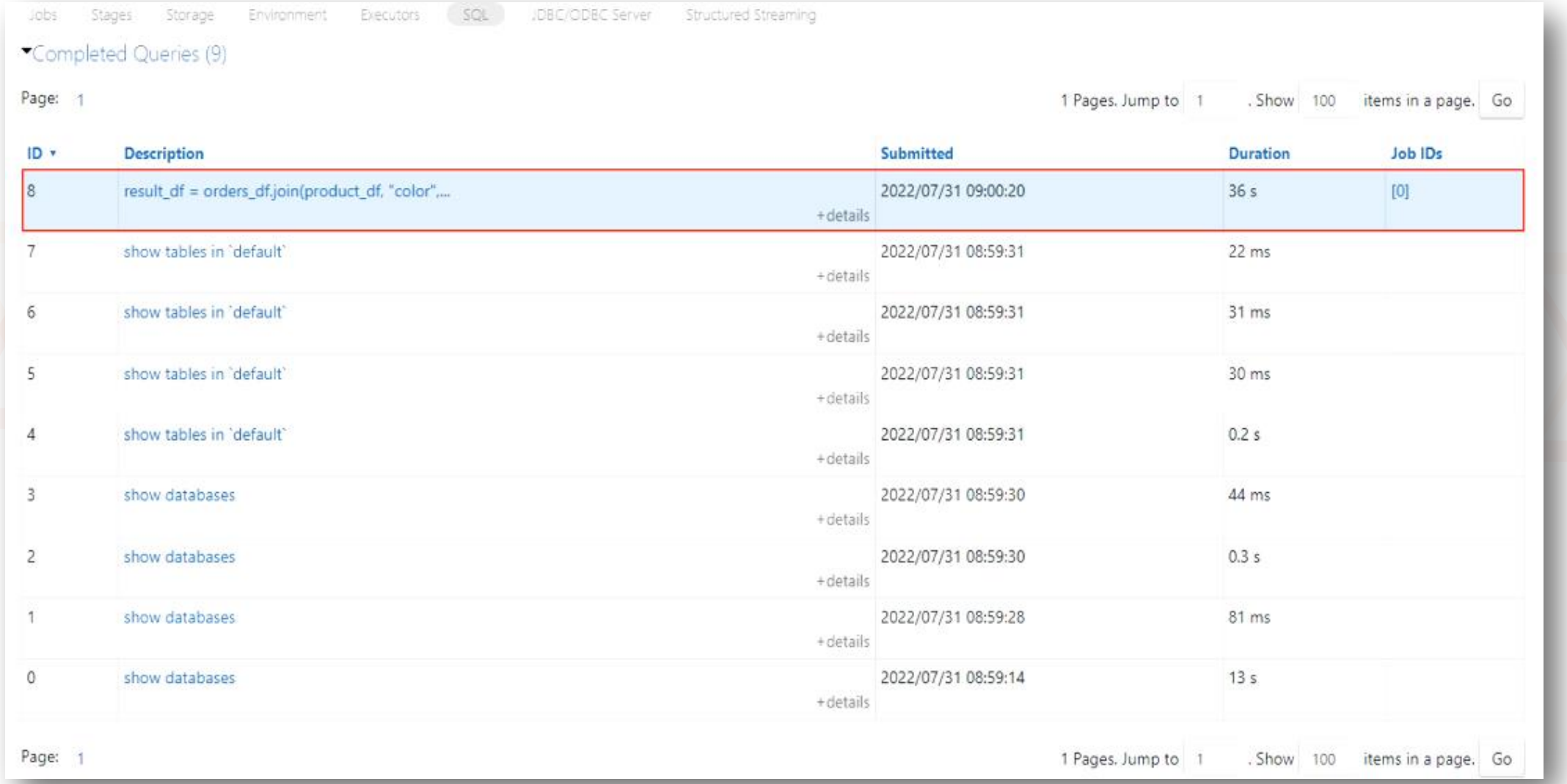
Go to the Jobs tab, and you will see one job.
Why only one job? We executed only one action, so we see only one job here.



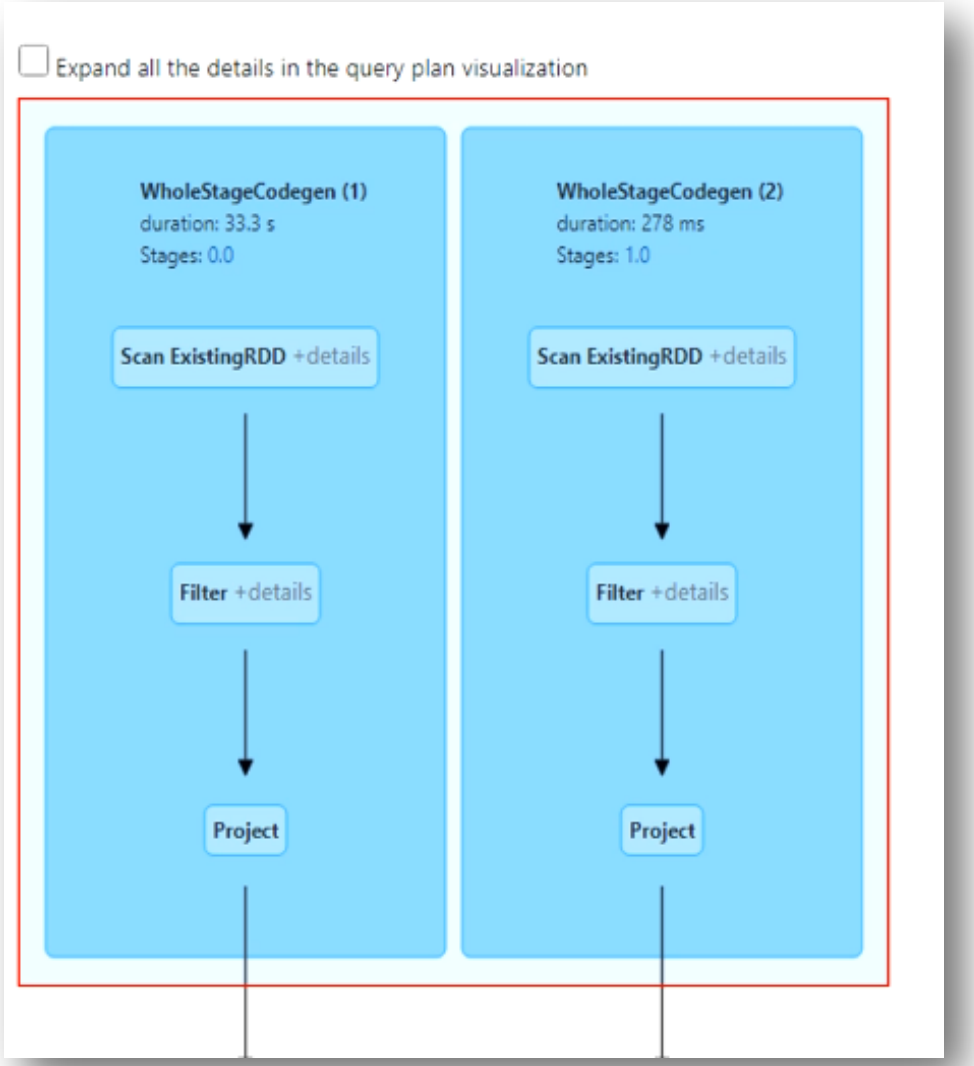
Now go to the SQL tab and you might see many things here.
But most of these are metadata queries triggered by Spark. Spark will trigger these metadata queries for parsing the data frame expressions. Look at the job id column. There is no job id in those metadata queries. We have only one query, which is tagged as job id zero.



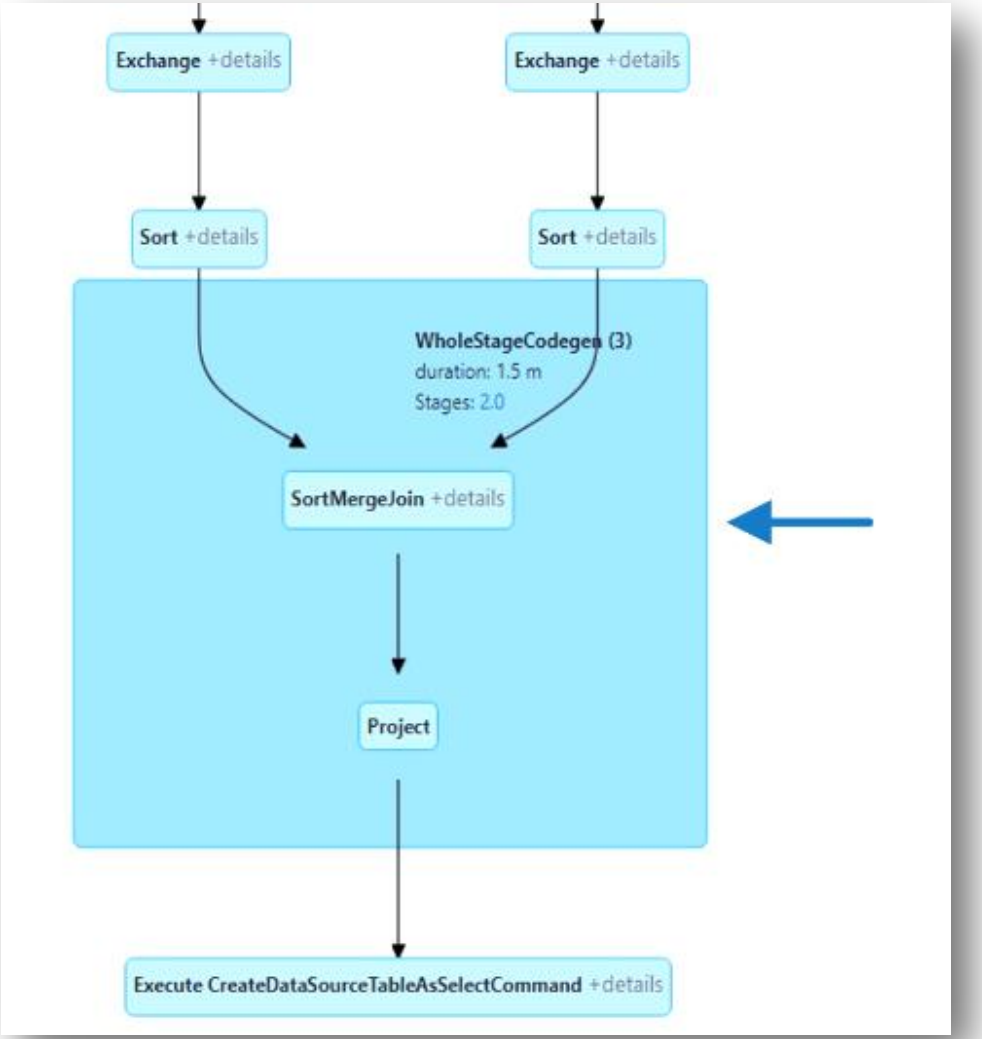
And that's the query for the write() method we executed in the end.
Click the query description so you can see the execution plan.



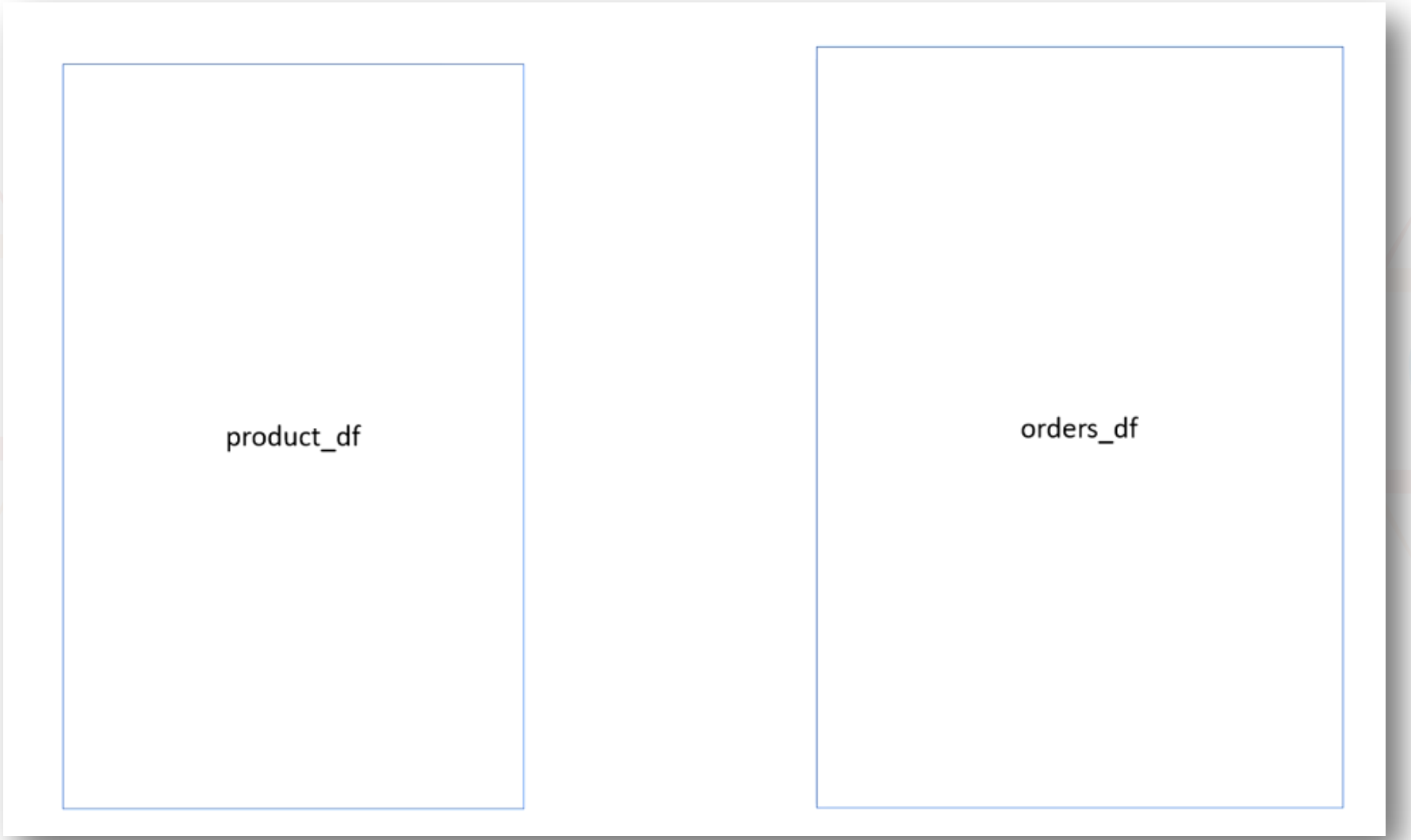
We see two blue boxes here. They are showing side by side. These two boxes are the two stages. You can see the stage id. One box shows stage id zero, and the other shows stage id one. Stage zero is to create the first data frame, and stage one creates the second data frame.



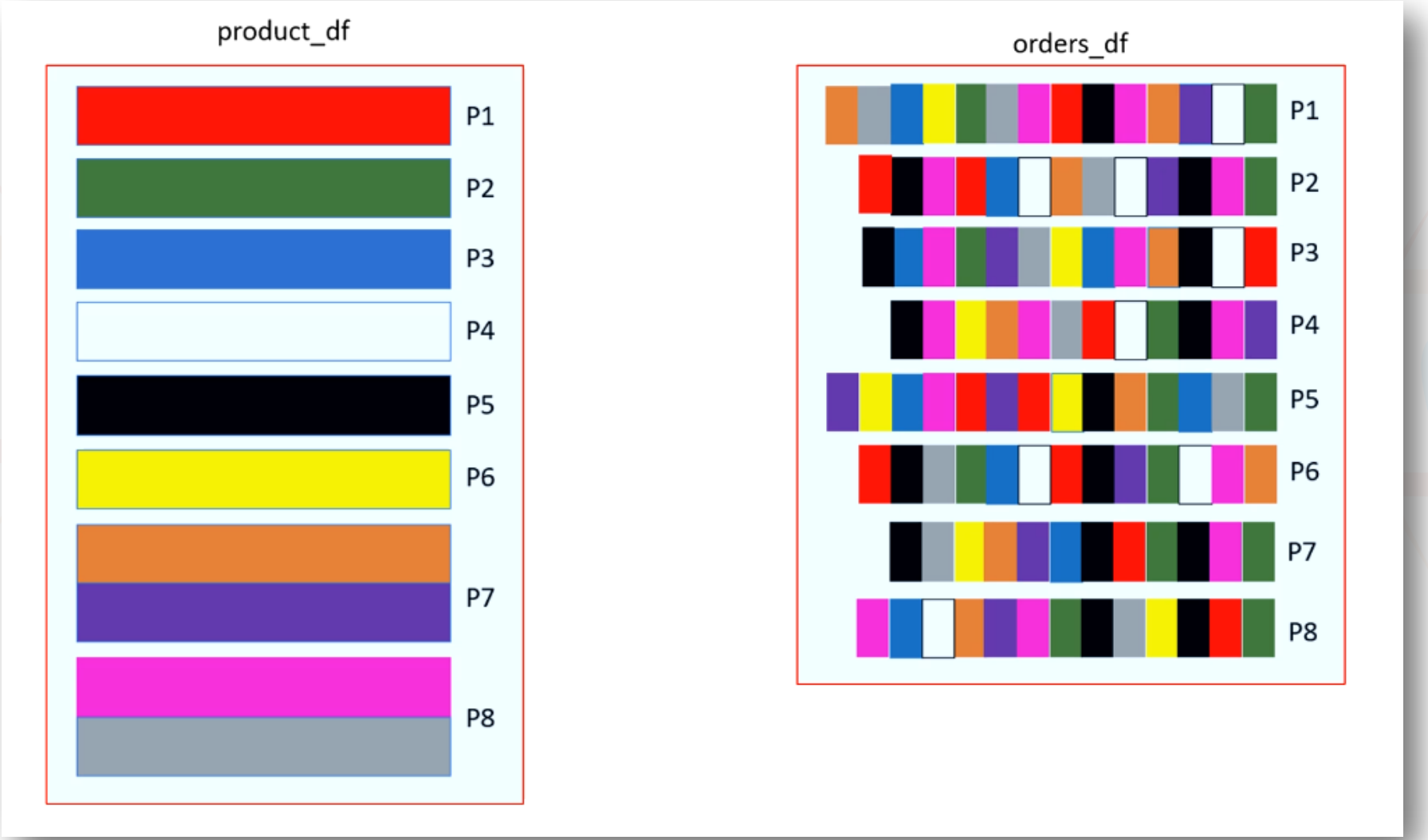
Scroll down, and you will see that the output of these two stages will flow into the third stage. And the third stage is to join the first two Dataframe and create the final result data frame. You will also notice the join strategy. It says sort-merge Join.



But how does joining happen?
Let's try to understand it. We created two data frames. Here are they shown below.



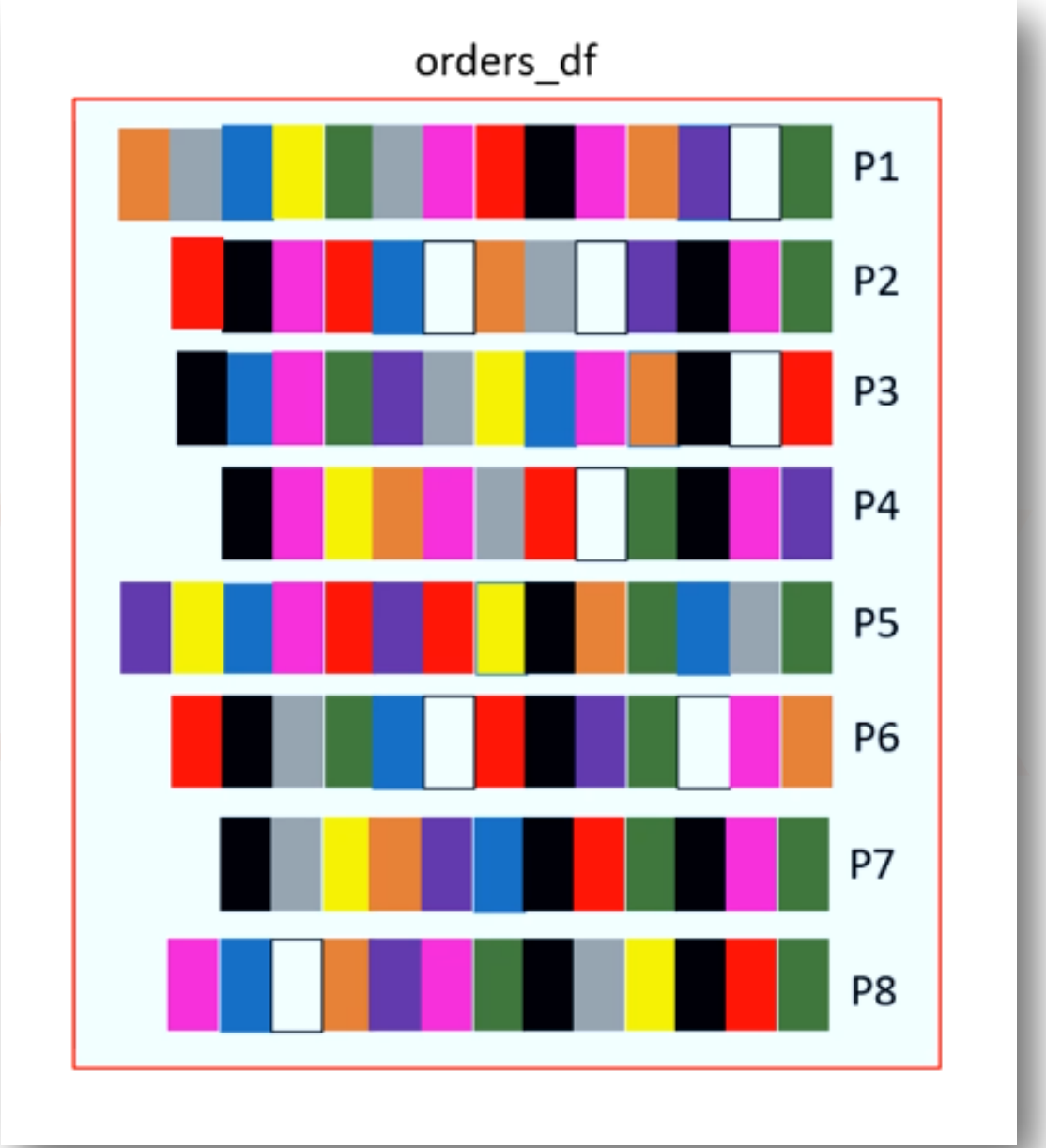
But we know Spark Dataframe is internally broken into partitions. So let's assume we have eight partitions for both the data frames.



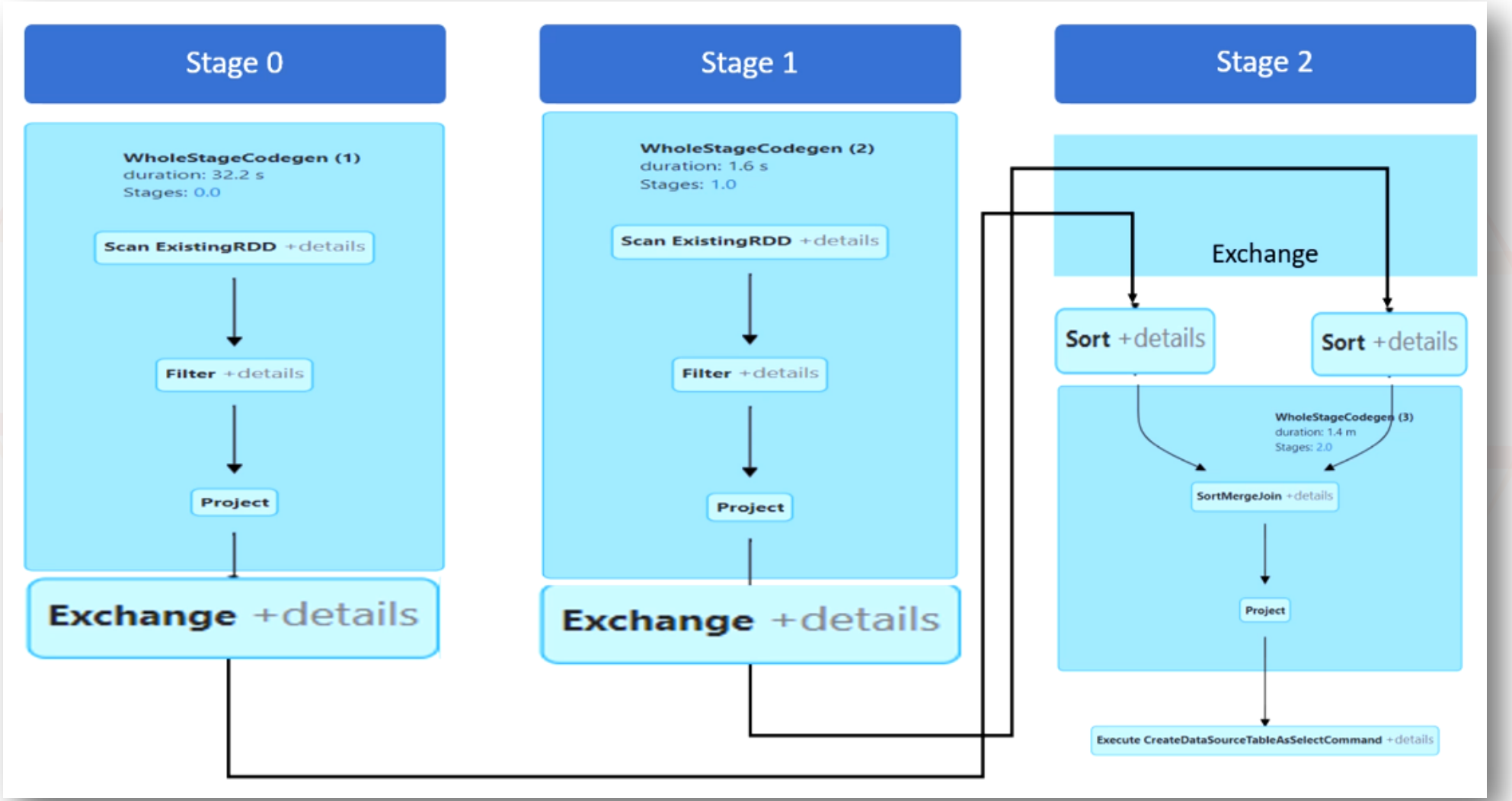
The product_df has got only ten records.
So the first six partitions are super tiny. They have only one record in each partition.
But the last two partitions have got two records in each partition.
So we have eight partitions here.
P1 to P6 are single record partitions, and P7 and P8 are two records per partition.

But why do we have eight partitions? Why not 10 or just one partition?
Because the driver decided to load the data frame into eight initial partitions.
I will explain later how the driver decides on these initial partitions.

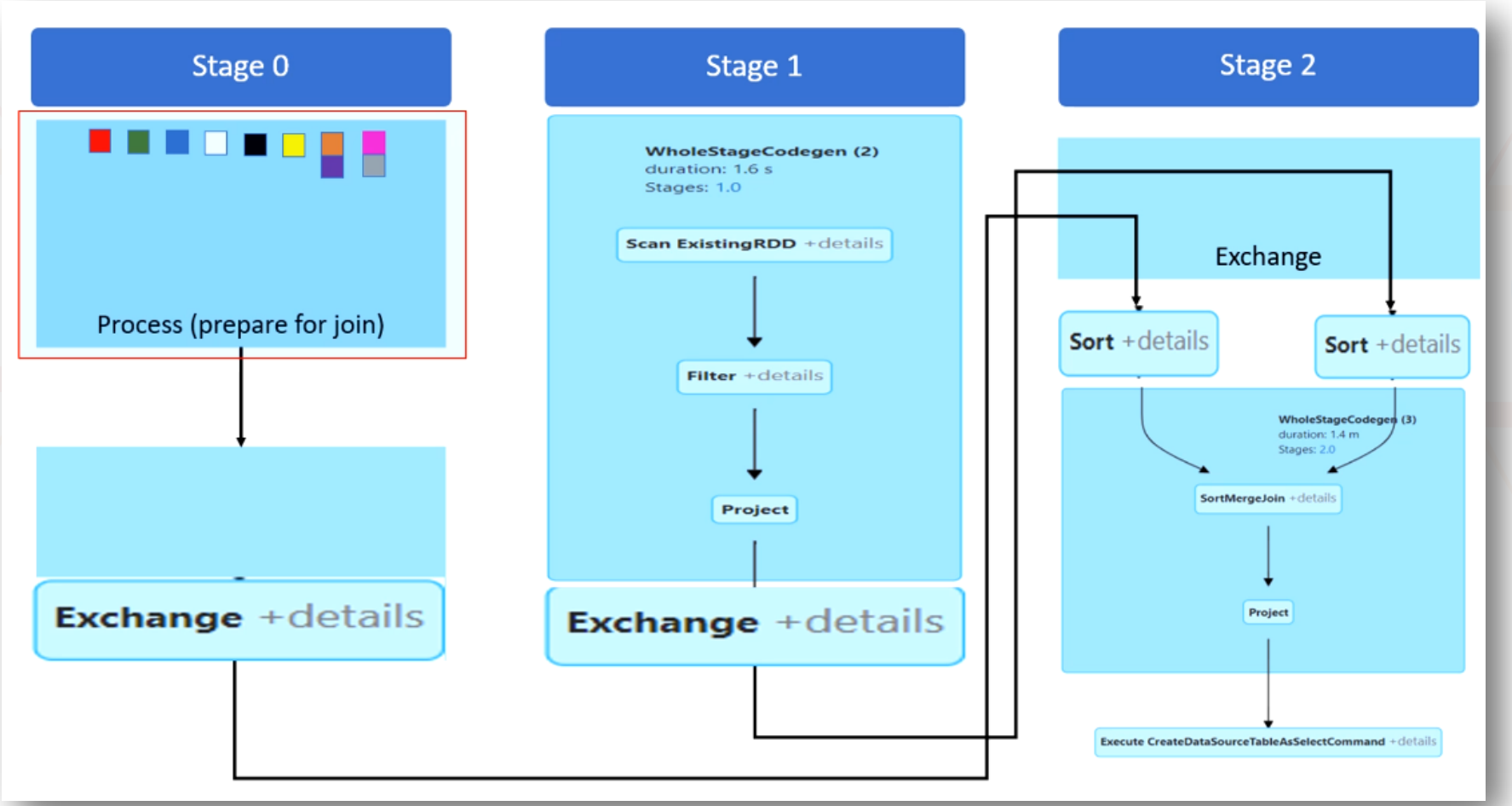
Similarly, the order_df is also loaded as eight partitions.
But these are not single record partitions.
These are big partitions, and each partition has records for multiple colours.
All partitions are not equal, but they are almost evenly sized.
The driver will take care of creating even size initial partitions.
So it is not possible to have initial partitions skewed. Spark automatically takes care of that.



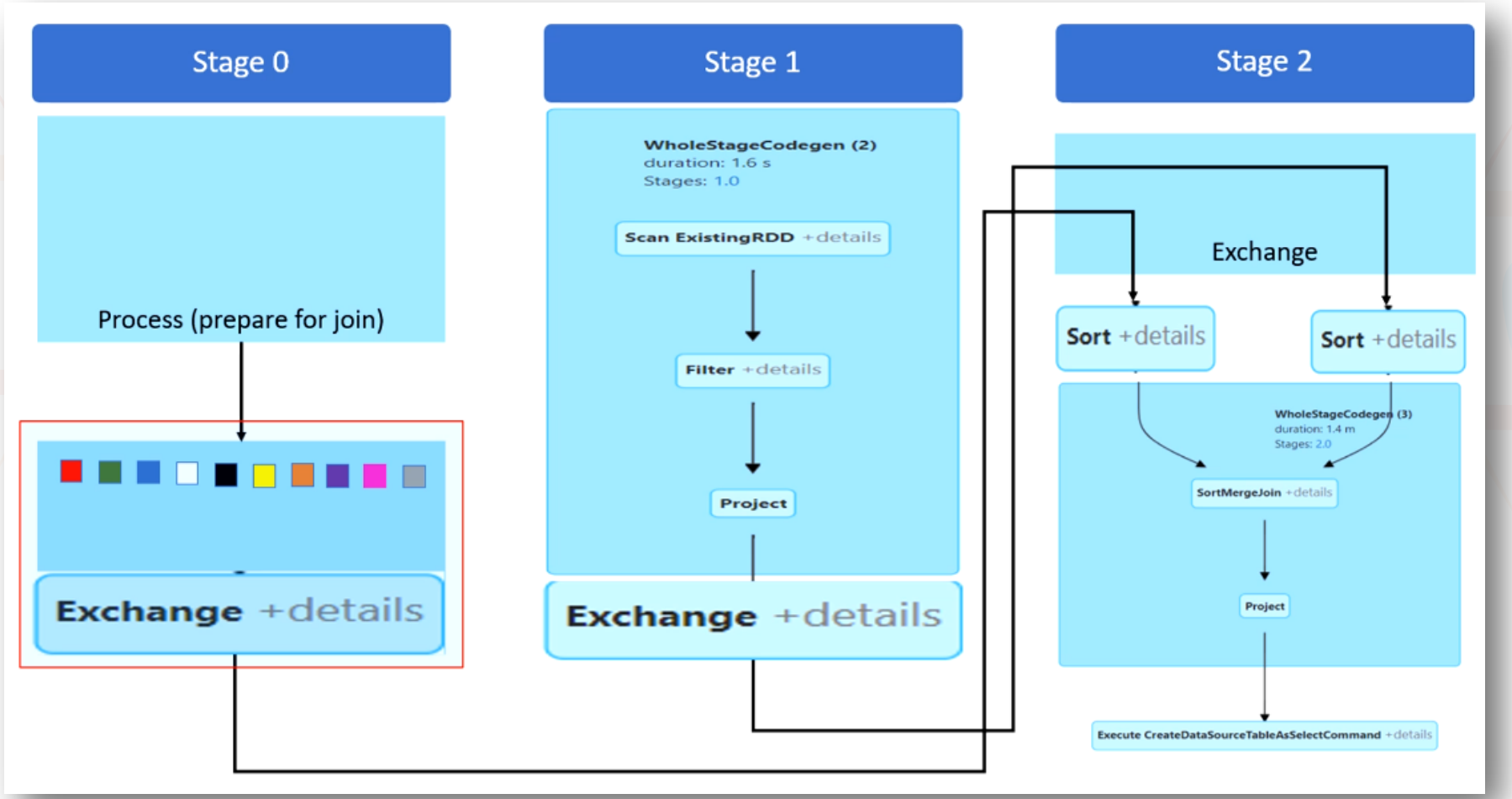
We are performing Join. And we already saw the following execution plan on Spark UI.
Our code ran in three stages: Stage 0, stage 1, and stage 2.



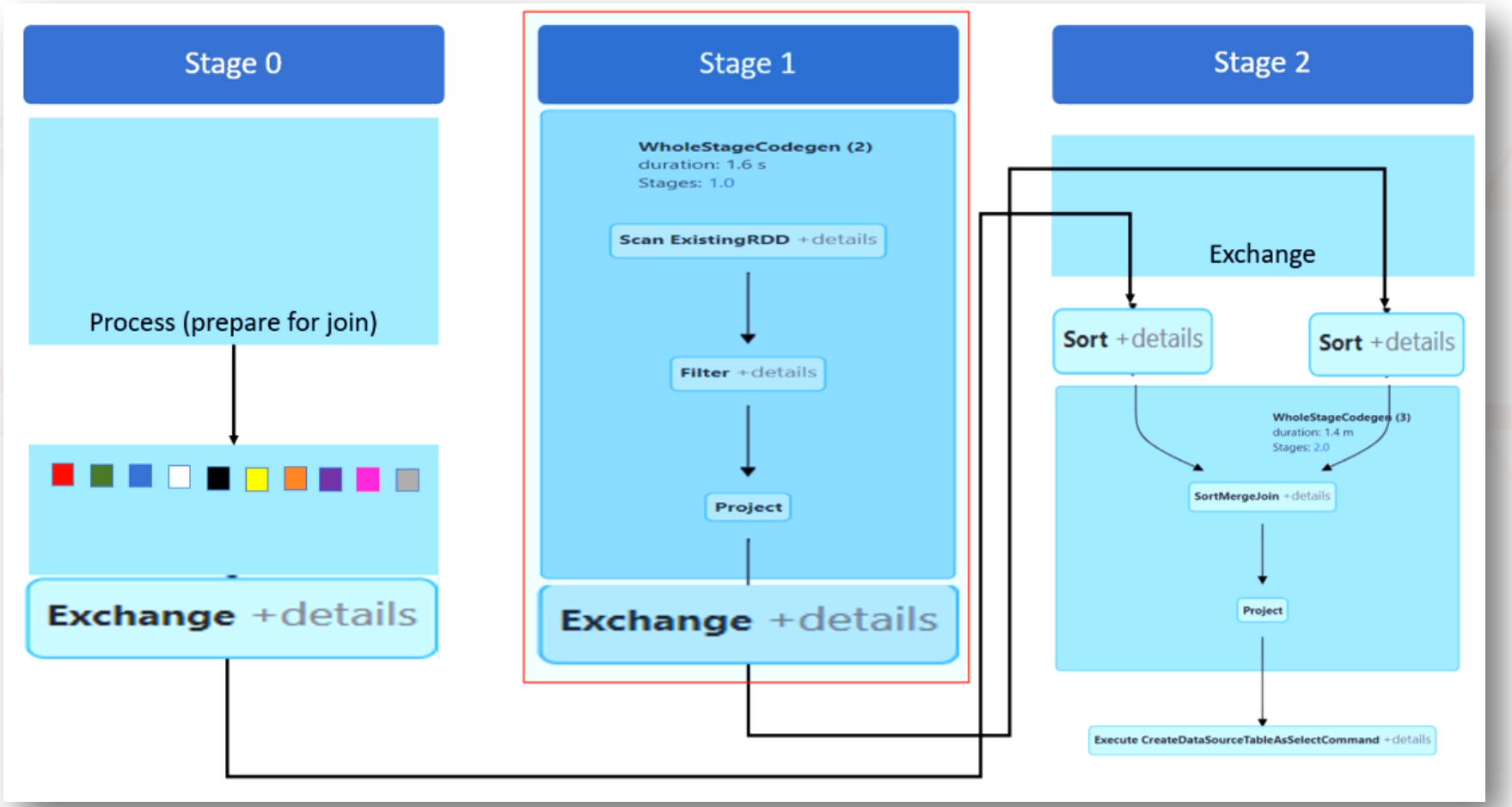
Stage zero reads the first data frame. So if we uncover stage zero, you will see something like this. So we have eight initial partitions in the first data frame. So stage zero will read eight partitions. Then it will prepare for the Join. And preparing for Join is all about separating join each key. In our case, the key is the color.



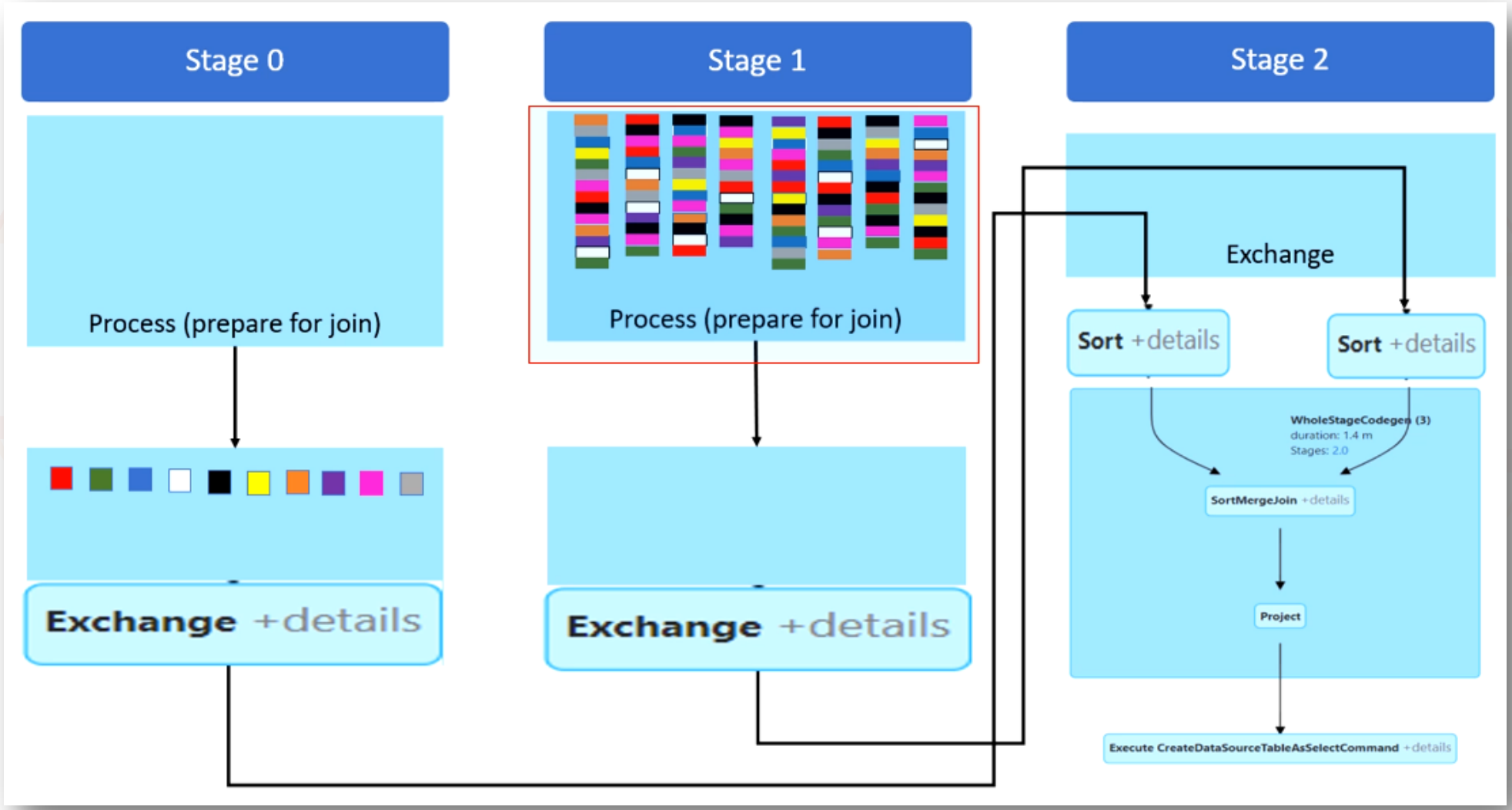
So, stage zero will separate each color and send it to exchange. The exchange is disk storage, and we call it an output exchange. The output of the stage goes to the output exchange. We are performing Join. So the output will separate each join key. We have ten unique colours, so we will have ten partitions in the output exchange.



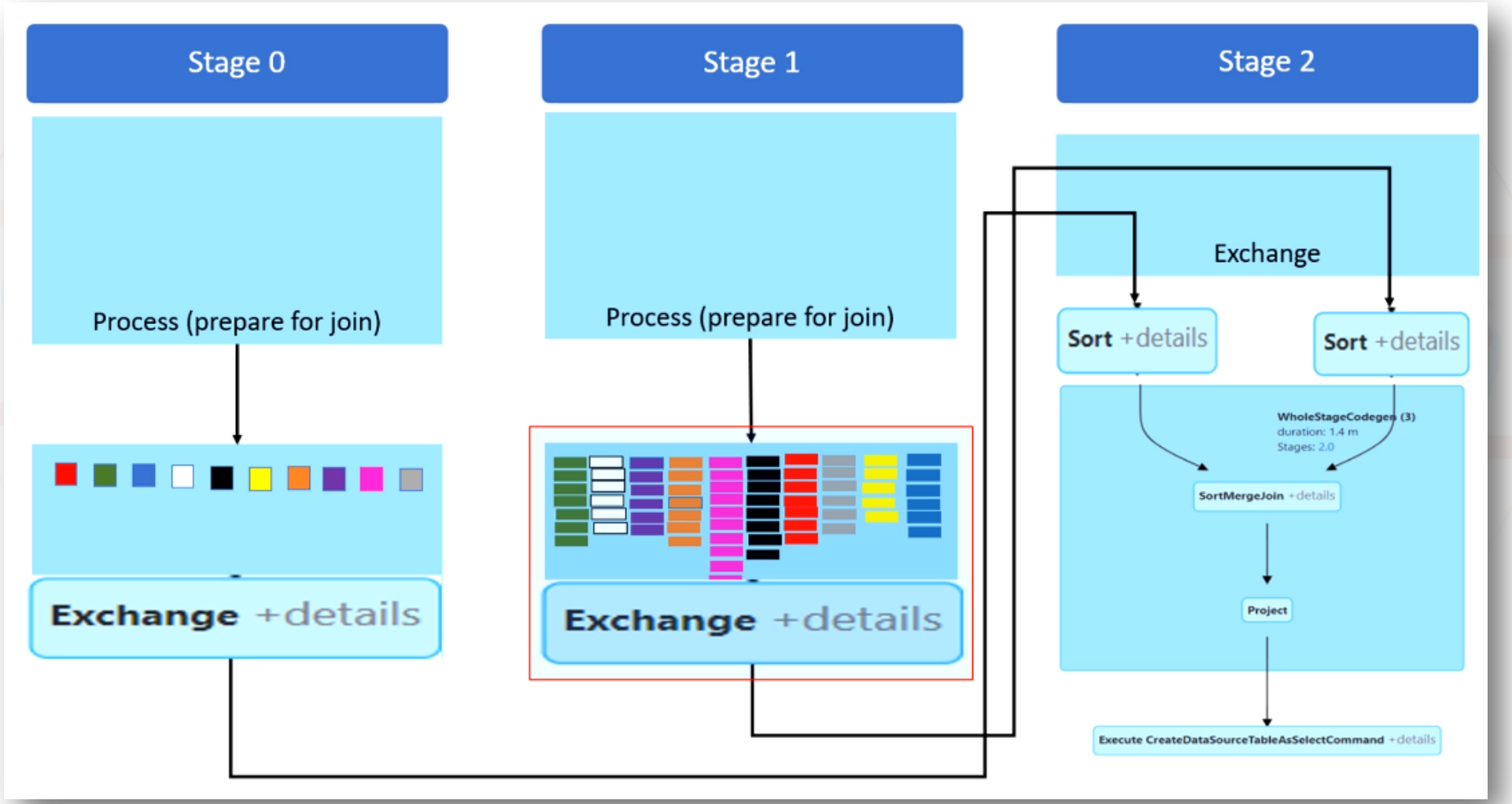
Once stage zero is done, Spark will start stage one. Stage one is not dependent on stage zero. Spark could run the bot in parallel. However, it will not run in parallel. It will always run in a sequence. Optimizing it to run in parallel is a topic for another day. For now, let's see the default behaviour. Spark will finish stage zero and then start stage one.



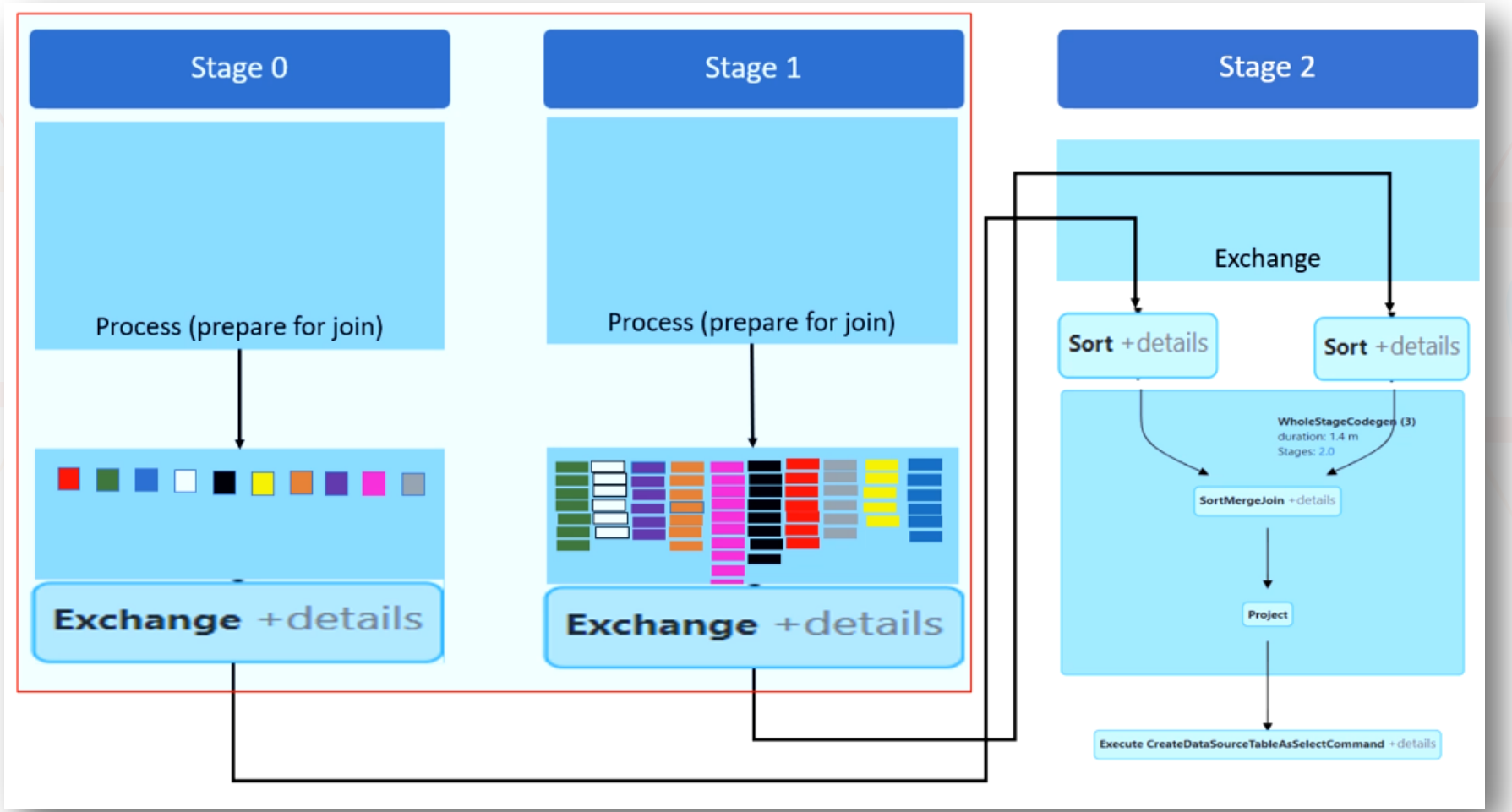
Here I have uncovered stage one, which will look as follows.
We have eight initial partitions in stage one. These eight initial partitions are created by the driver.



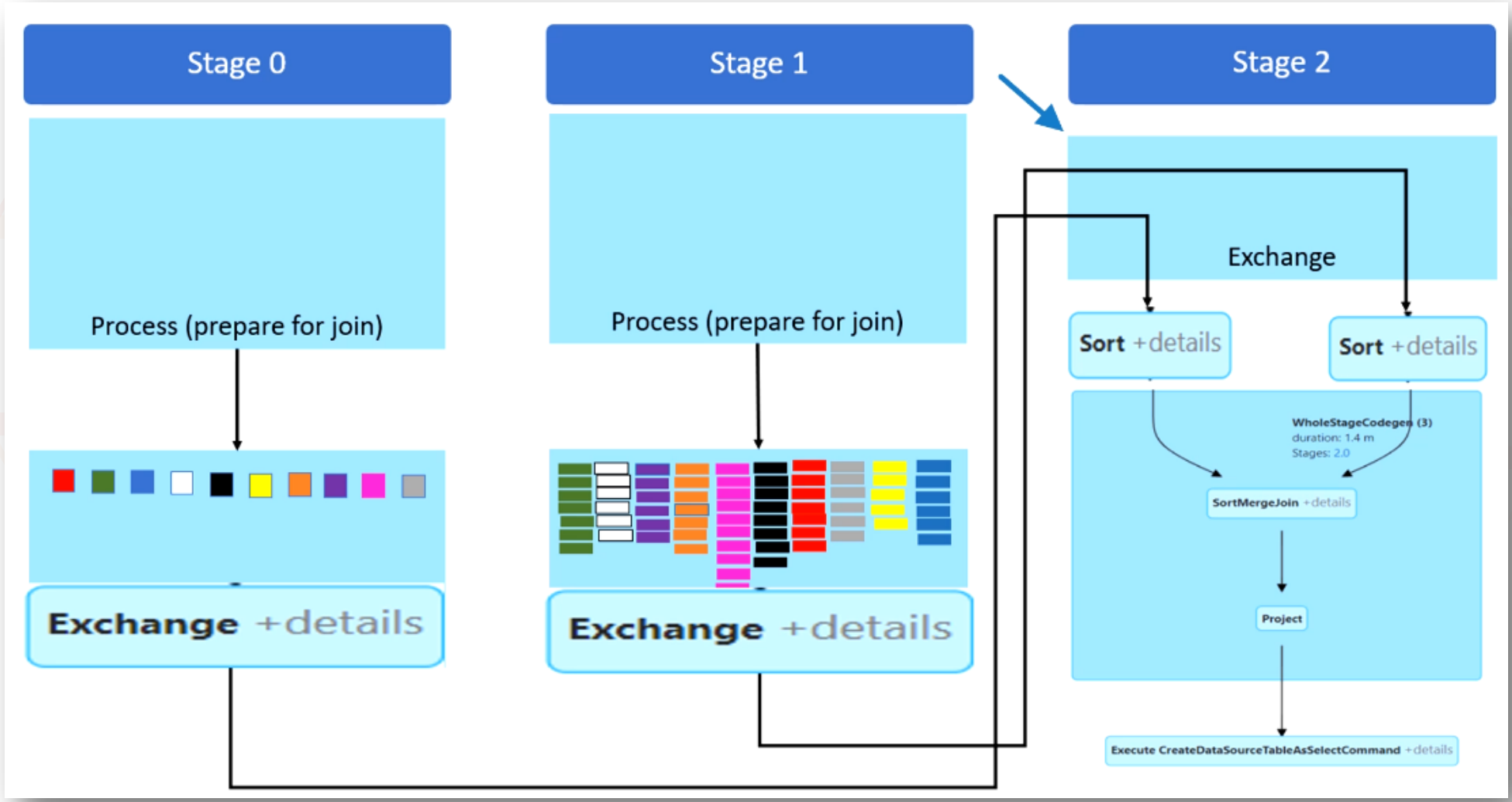
So the Spark will process these eight partitions and send the output of stage one to the write-exchange. We are performing Join. So the output will separate each join key. We have ten unique colours, so we will have ten partitions in the output exchange.



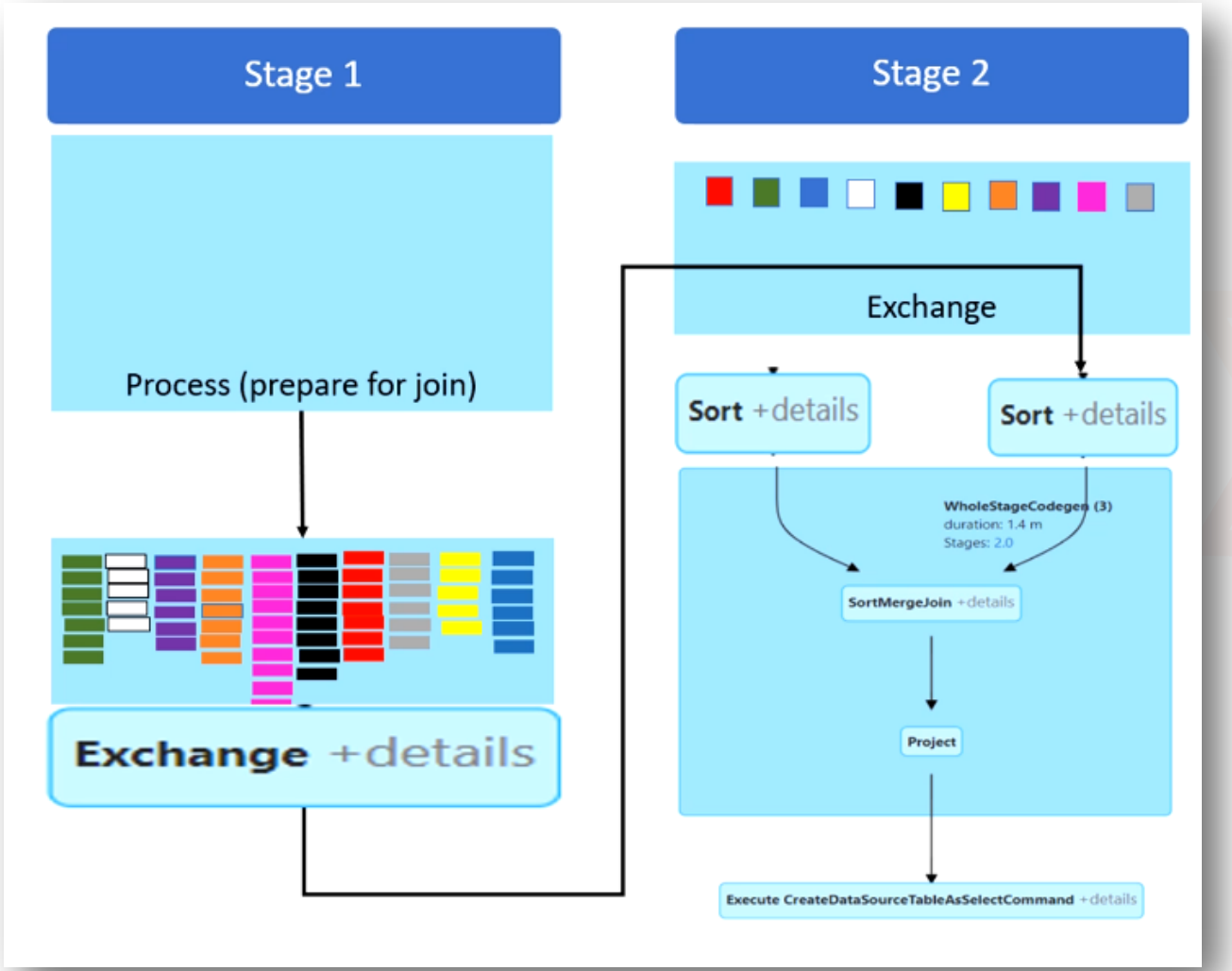
So we finished the first and second stages. This is how things look at the end of stage two. We have two exchanges. Stage zero exchange holds ten partitions of the first data frame. Stage one exchange holds ten partitions of the second data frame. Both these partitions are grouped together by the join key. Now we are ready to perform the Join in stage three.



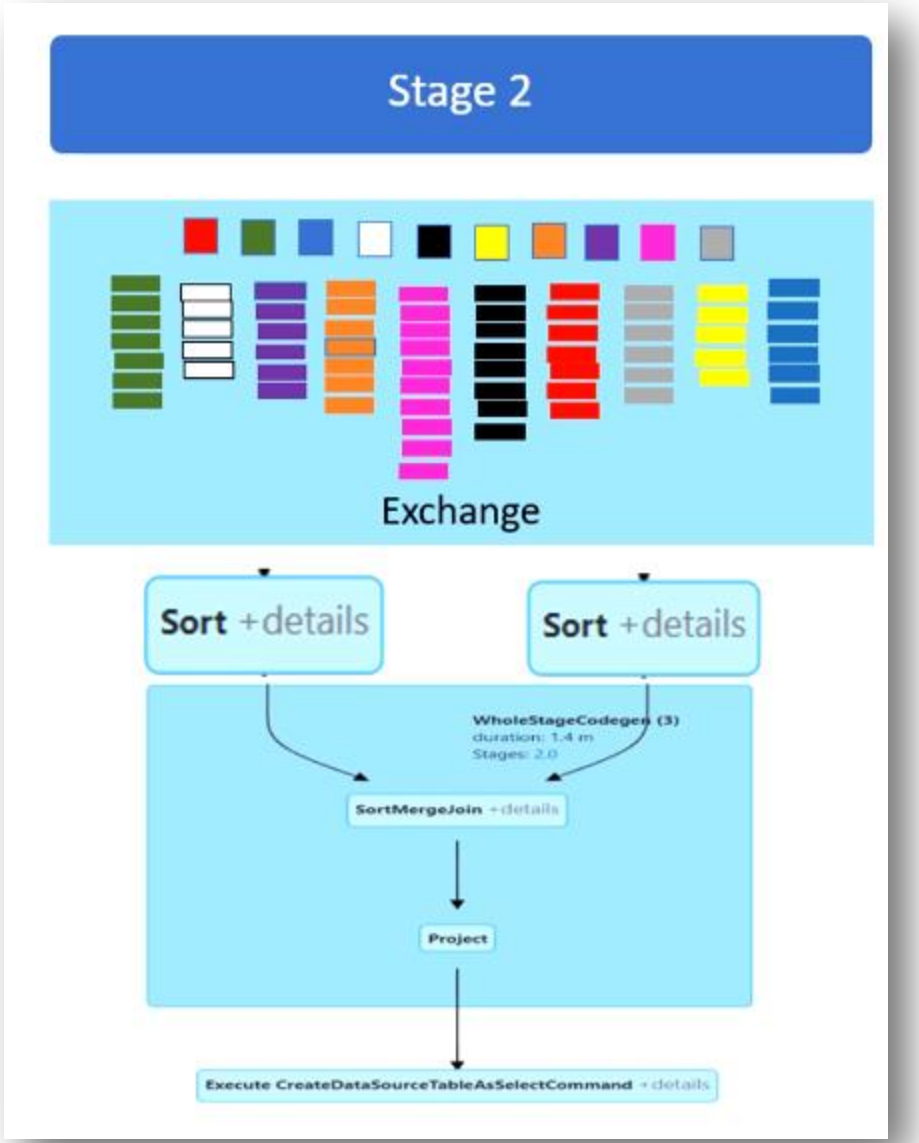
So stage three will read the stage zero exchange and bring data into the read exchange of stage three.



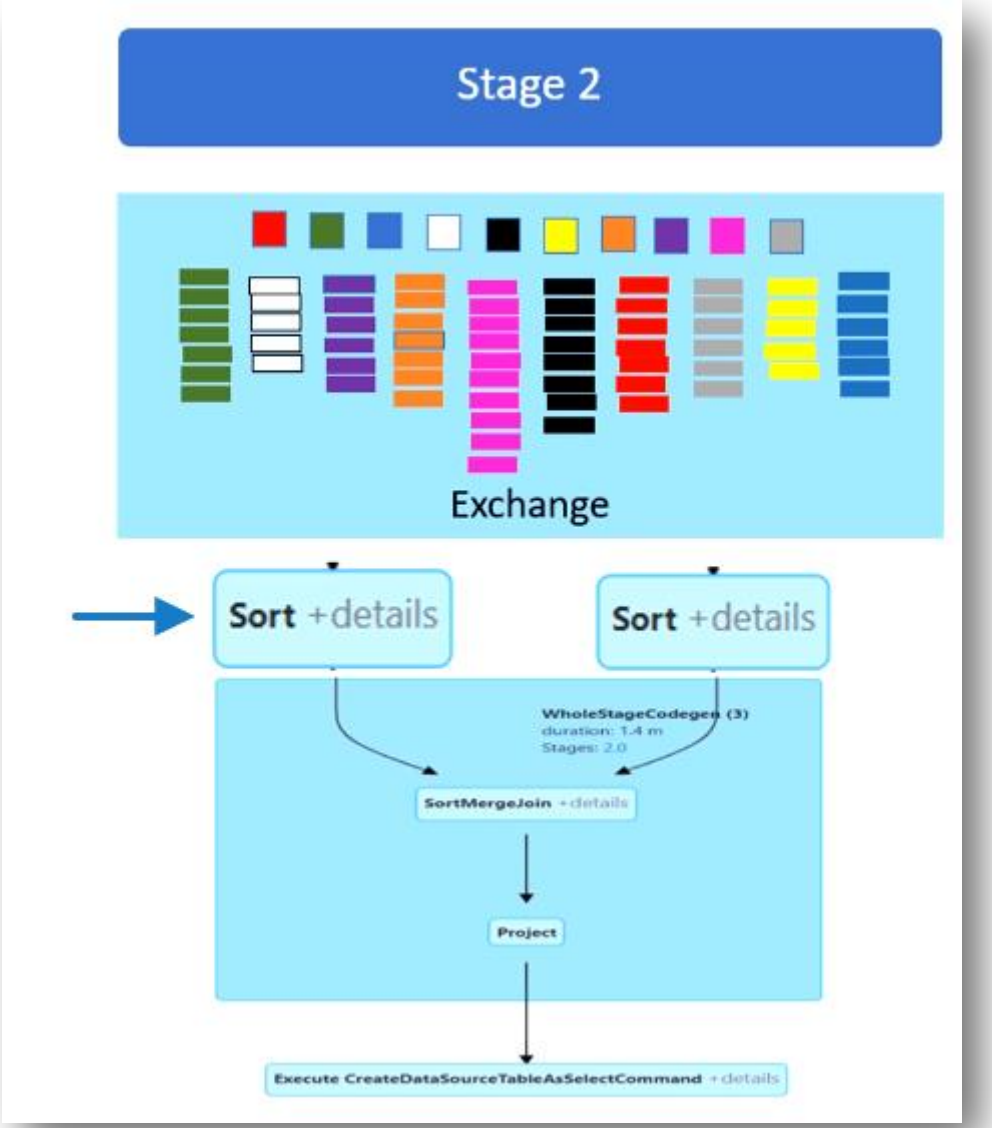
This is how it looks after the stage 0 transfer.
Similarly, Stage three will also read the data from the stage one exchange and bring it to the stage three exchange.



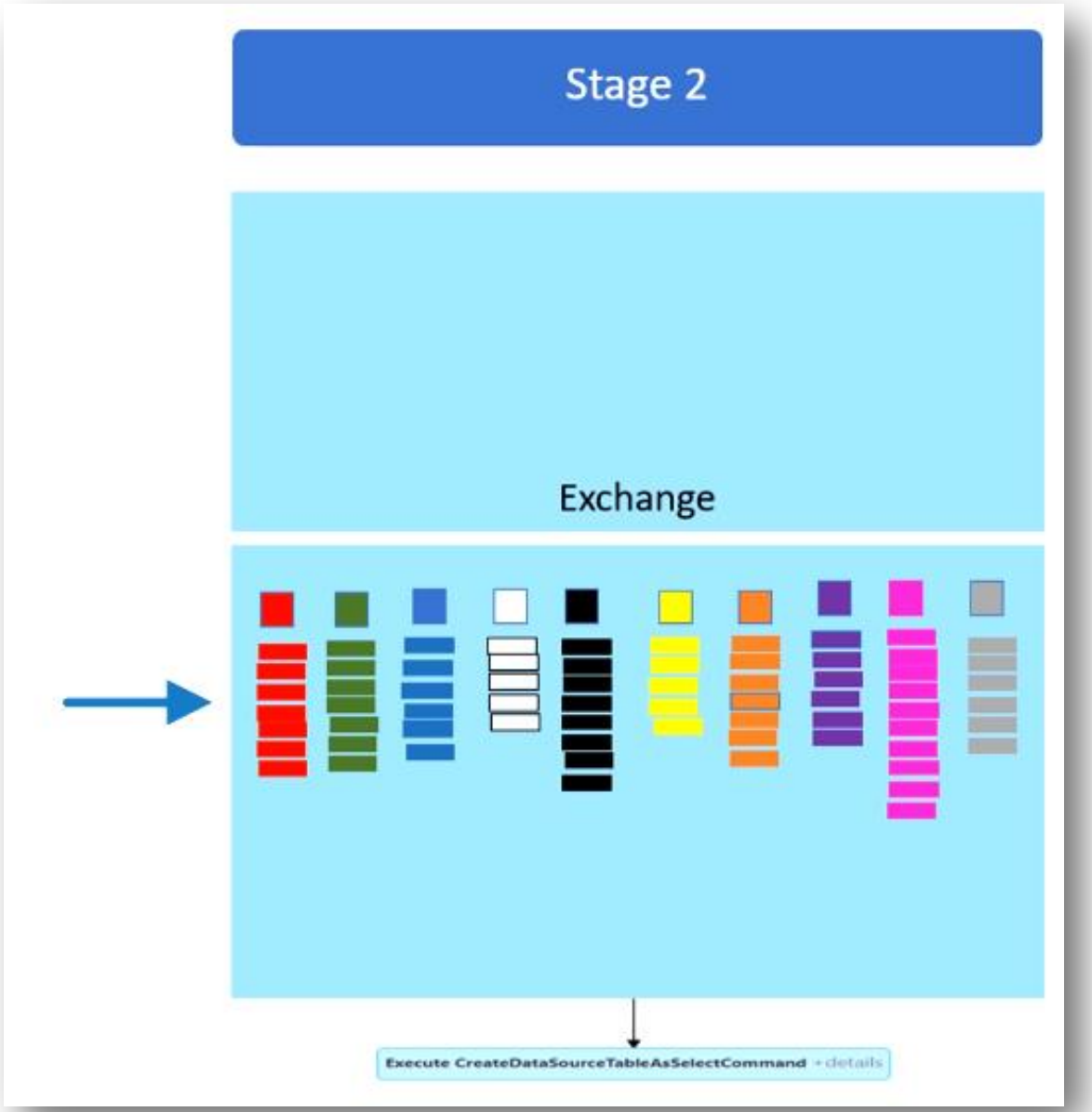
So now stage three has got all the data in one place. So we can perform the Join.



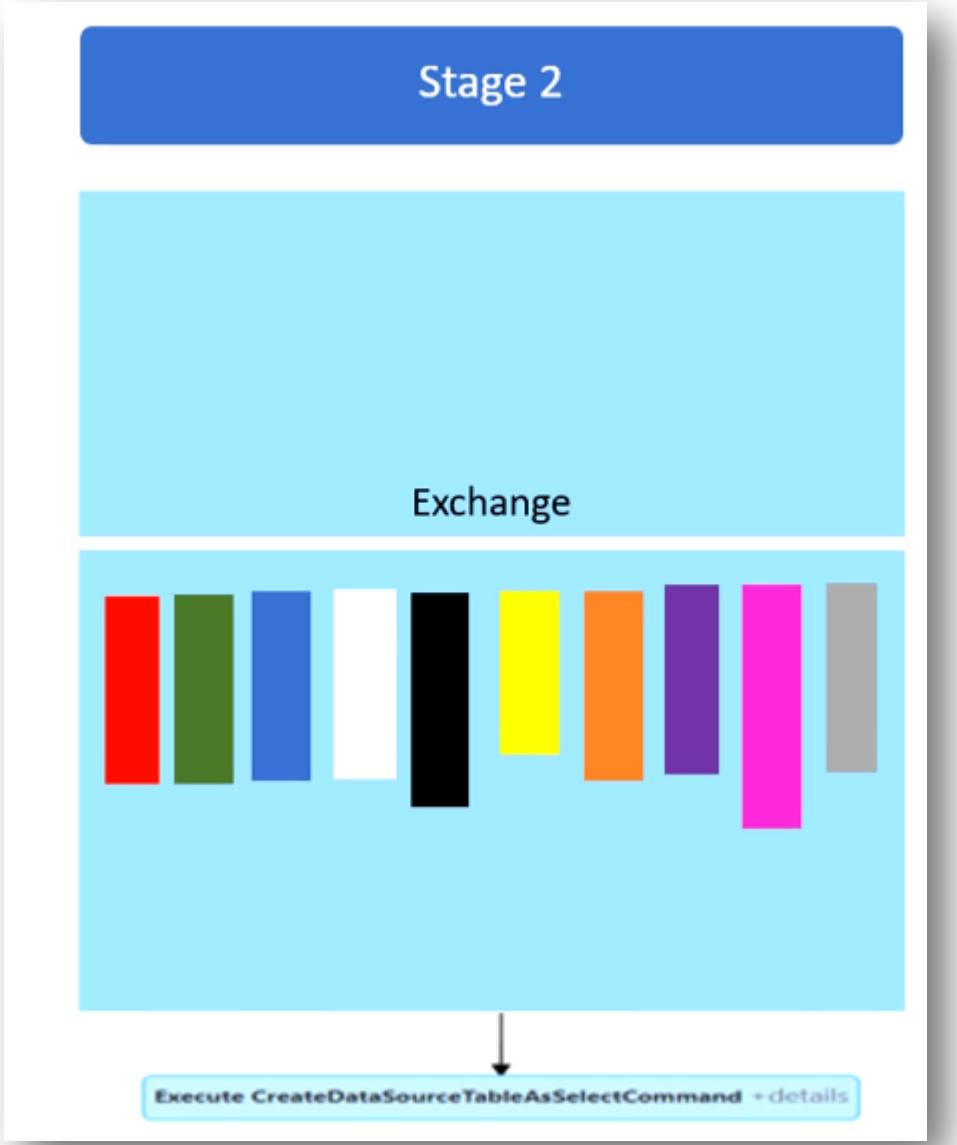
Spark will now choose a join strategy and apply the Join. In our example, Spark chooses to apply the sort/merge join strategy. That's why you can see the sort in the execution plan.



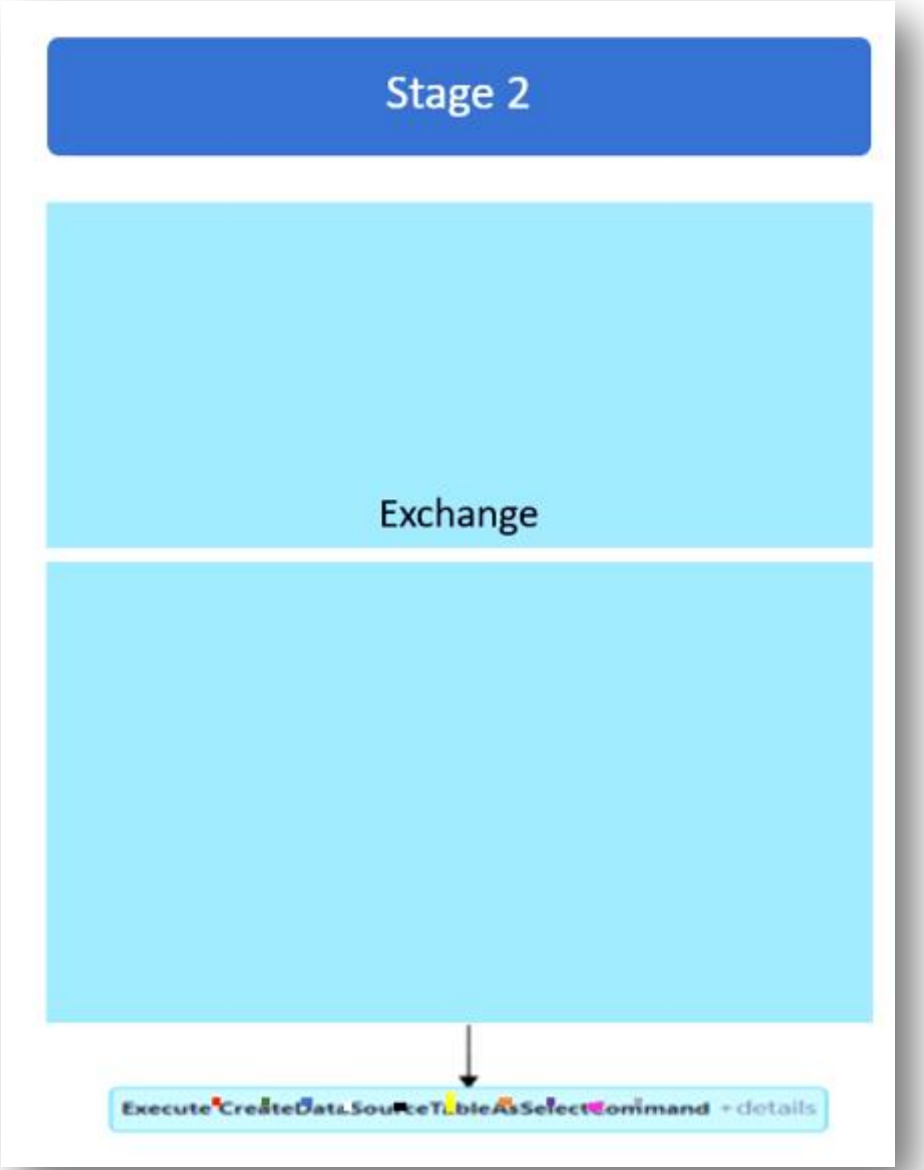
So, Spark will sort all the data frames partitions to arrange them in order.



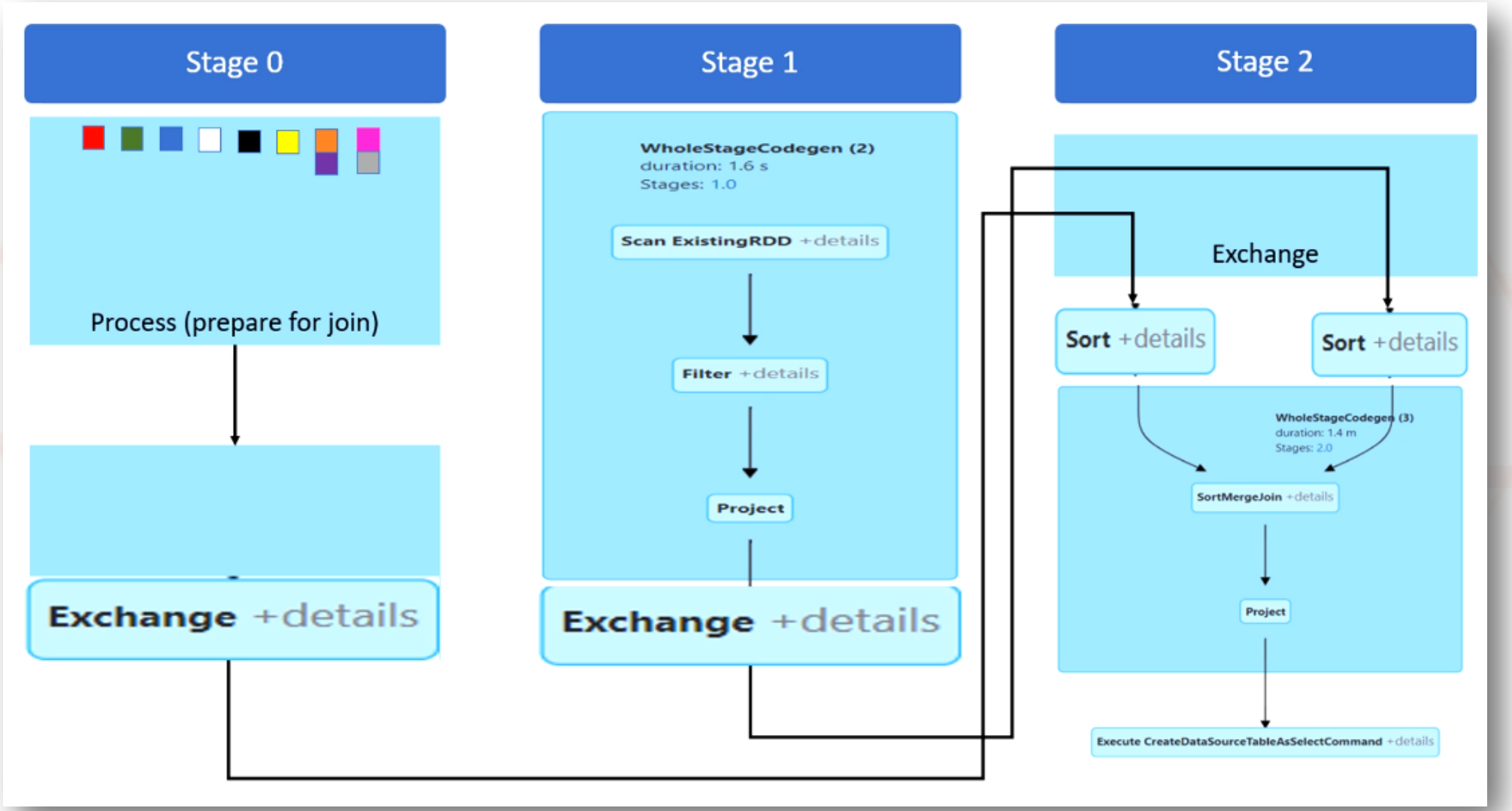
Once they are arranged, as we can see here, merging two partitions of the same color is super easy. So Spark will merge them.



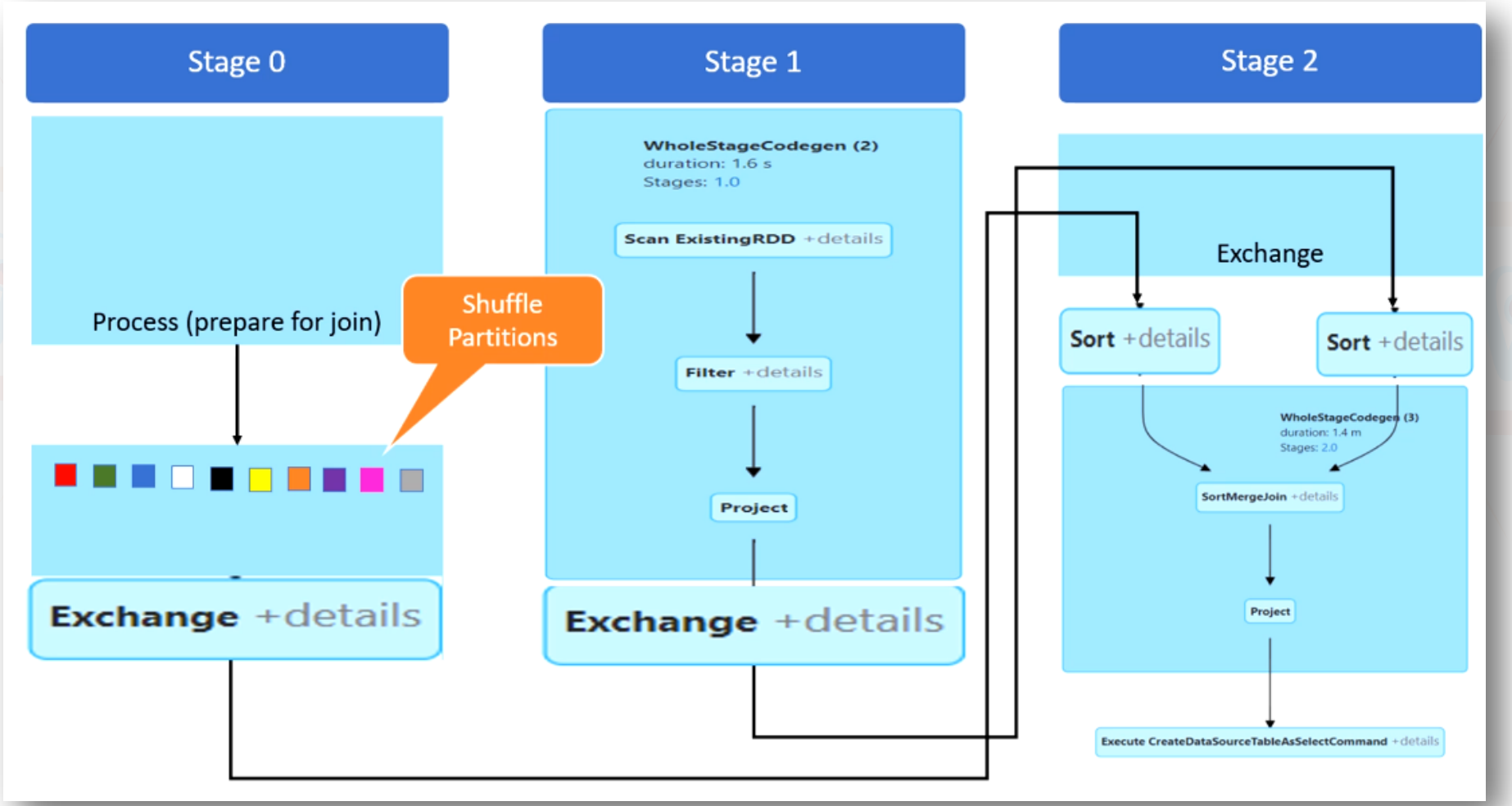
Now the last action is to write them to the disk. So these partitions will go to the disk below.



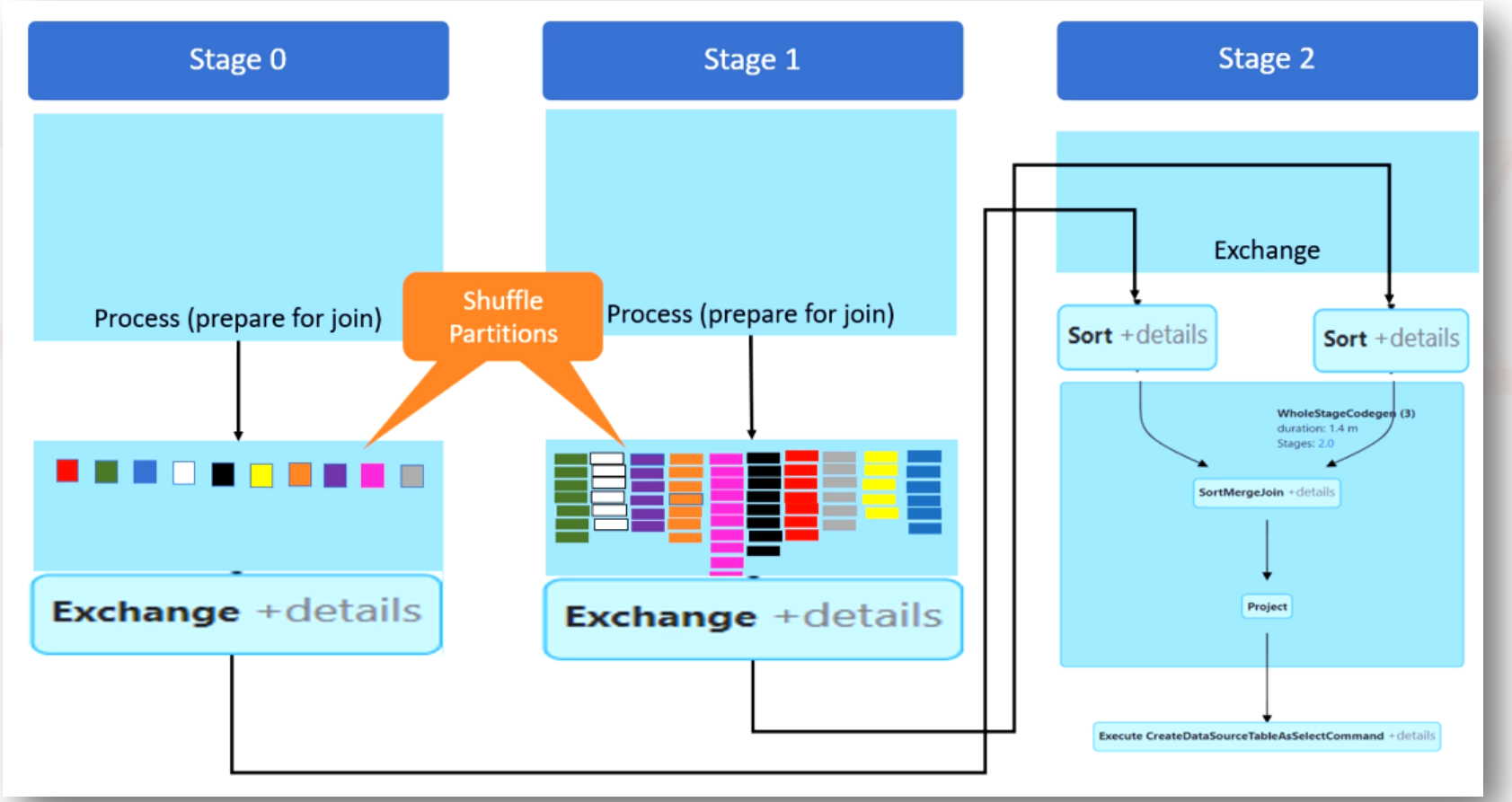
Let me recap once again how Join happens.
Here is the DAG at the start.



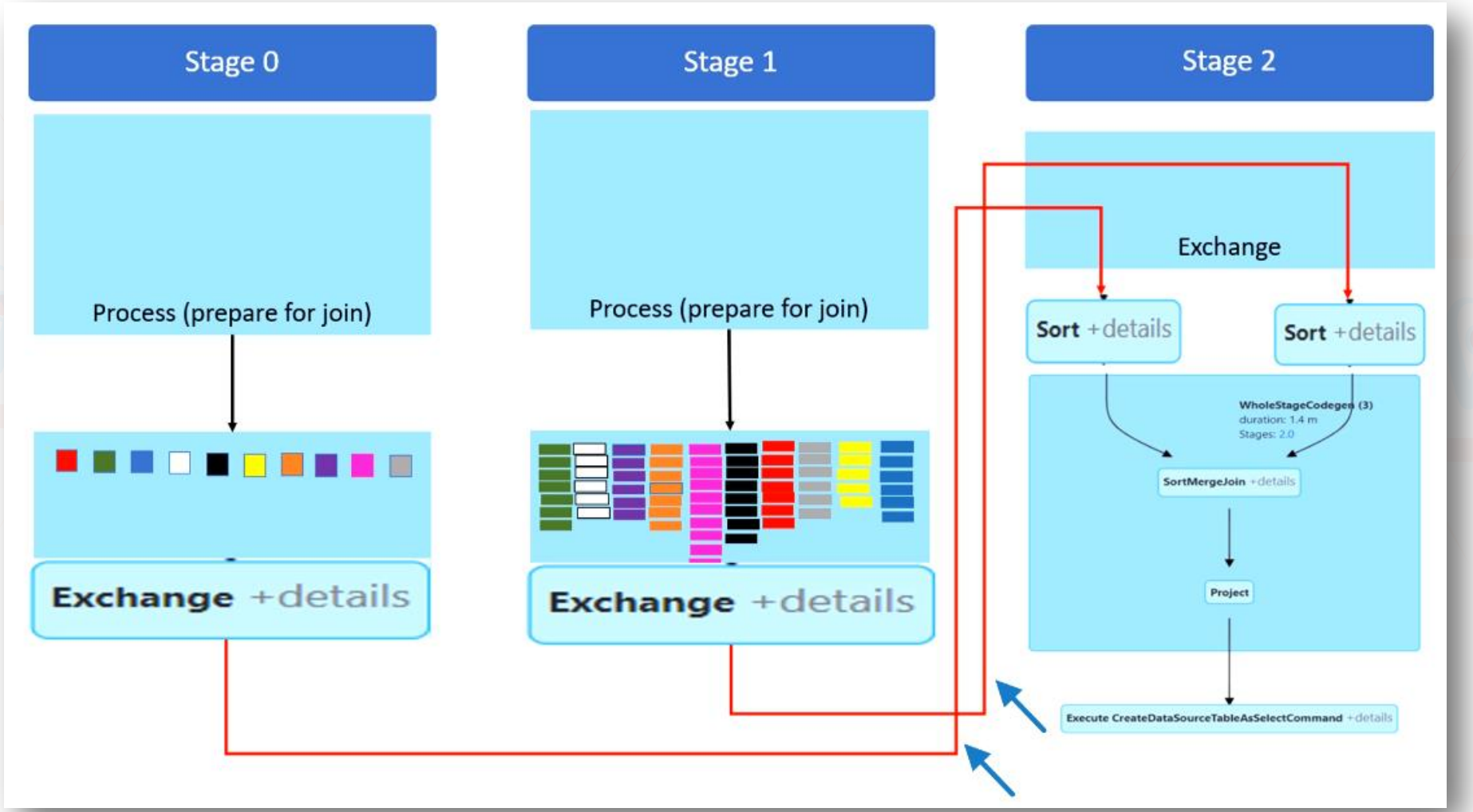
Spark will read all the partitions of the first data frame. It will then arrange the first data frame in the exchange. This arrangement is partitioned by the join key. And these are known as shuffle partitions. The shuffle partitions are saved on the disk in the exchange. The exchange is nothing but a disk location internally defined by the Spark.



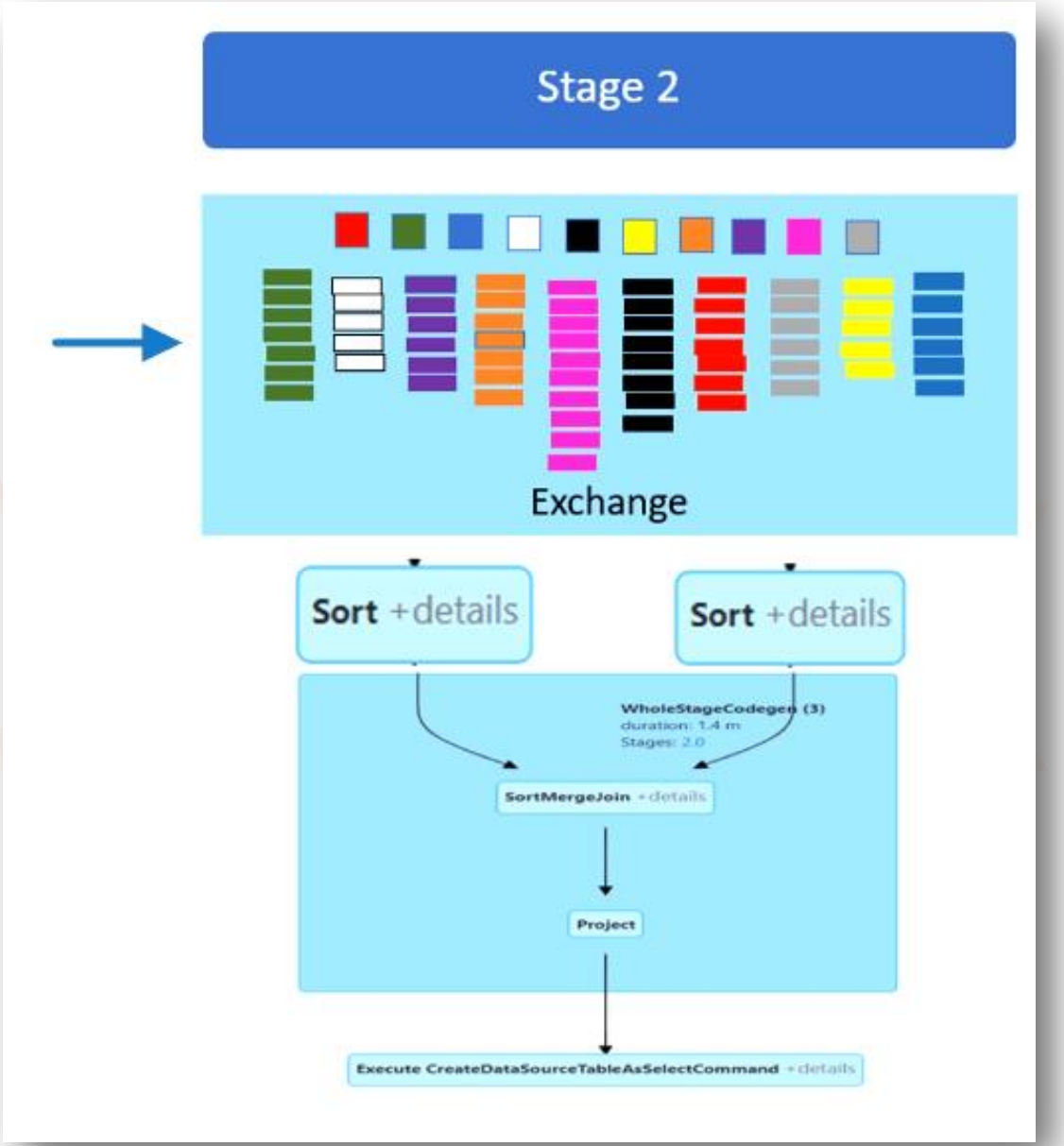
Now Spark will read all partitions of the second data frame. It will again arrange all the data in the exchange. This arrangement is again partitioned by the join key. And these are also known as shuffle partitions. So both the data frames will have shuffle partitions. The shuffle partitions are saved in the exchange location on the disk storage. These shuffle partitions are input for the next stage.



So the data from these exchanges will move to the next stage. This movement happens over the network using the red lines shown here. And this movement is known as the shuffle.
So, the shuffle is the movement of data from one stage to another stage over the network.

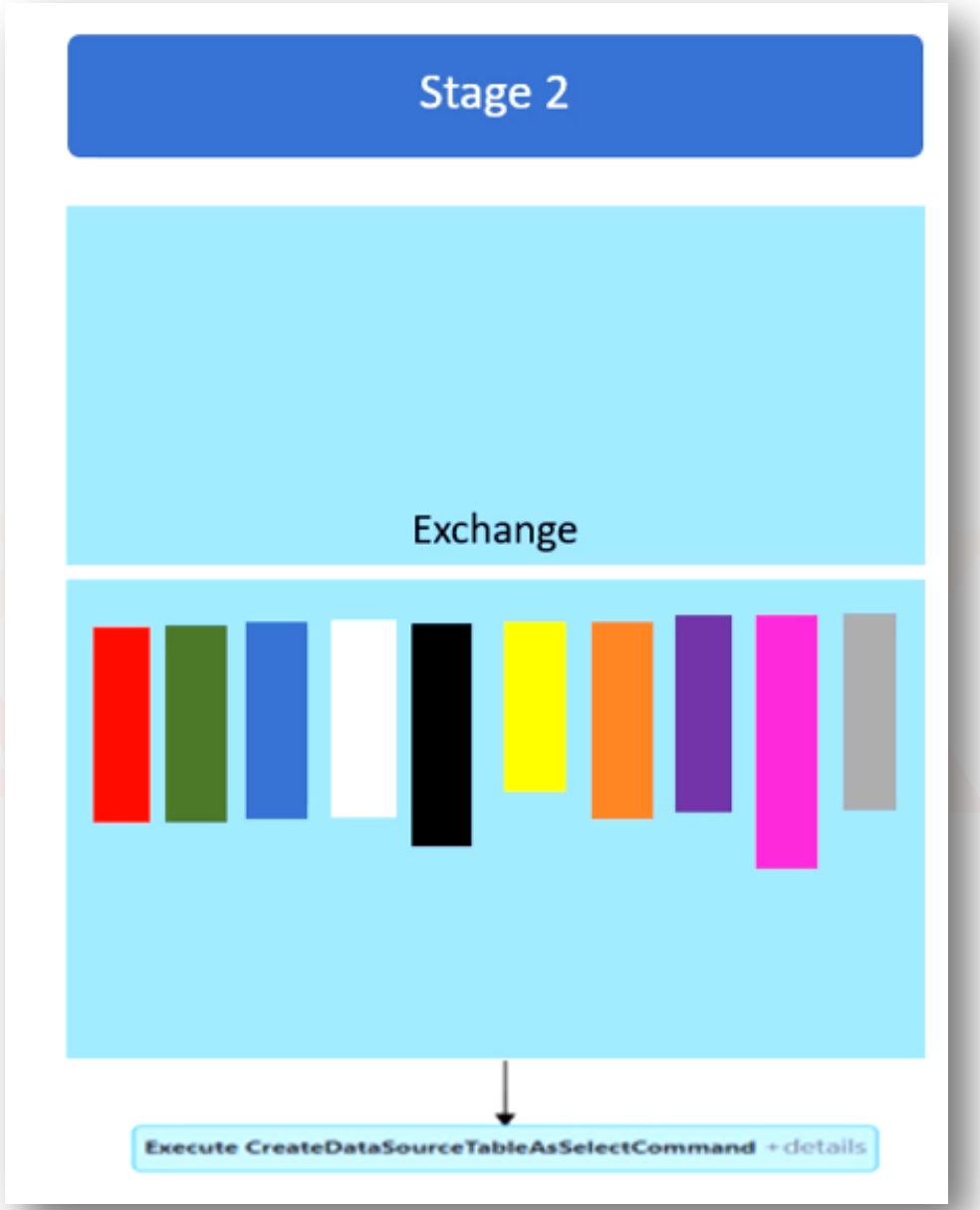


So now data is available in the input exchange of the last stage.
The input exchange is an in-memory location.
The output exchange of the previous stage was a disk location.
But the input exchange is an in-memory exchange. Remember that.
Spark should not throw an exception if you do not have enough memory.
It will spill some data to the disk and manage it.
Your performance will be slow, but it should not throw you an OOM exception.

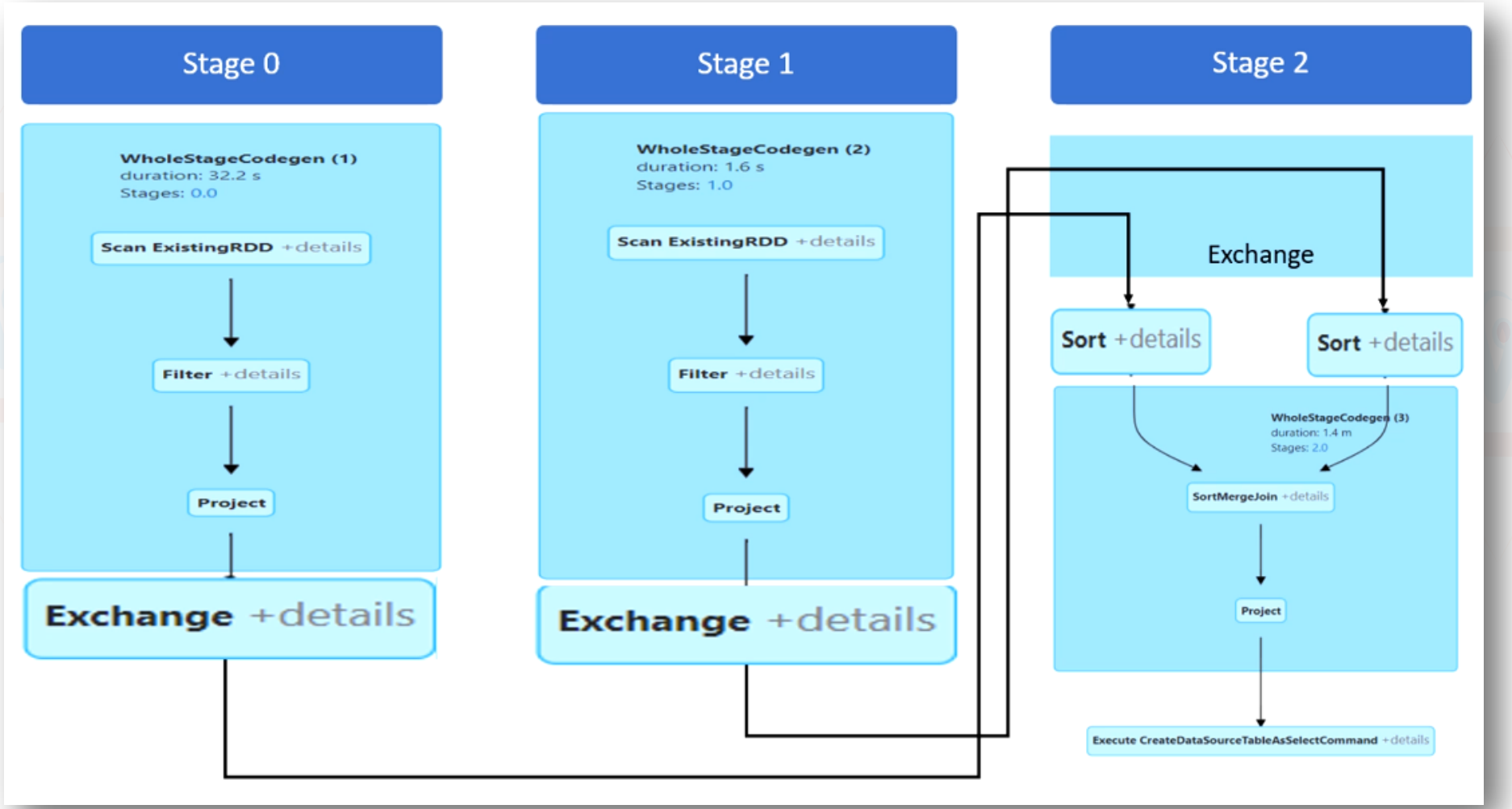


But you can still get an OOM exception here.
When? Let's try to understand.
The stage three will read partitions from stage one over the network.
At a time, Spark will read one partition from the previous stage.
But it will read it over the network and buffer it in the network buffer memory.
So you might see an OOM exception when the network memory buffer starts overflowing.
So, if you do not have enough network buffer memory, you will see an OOM exception.

The next step is to sort the data and then merge it.
But remember, sorting happens in memory.
So you should have enough memory to sort the data.
If you do not have enough memory to sort the data, you will again see an OOM exception.
Once sorted, merging is super simple.
Finally, the last step is to save it to the disk.
And how the Join operation takes place.



Here is the execution plan. You can see the same diagram on the Spark UI.
What do we call it? Execution plan or explain plan or Spark DAG. No matter what you call it, it shows you how things are happening.



I explained the internals of the Join happening on a single node.
However, things are a little more complicated because all this happens on the multi-node cluster.
I wanted to make sure you understand the following:

1. Shuffle partitions
2. Shuffle operation
3. Sort merge

The shuffle partitions are stored in the exchange.
The shuffle operation will send the shuffle partitions to the next stage.
The sort/merge is the join strategy applied at the final stage.

Here is my notebook. Look at the last command of the first cell. I am getting the shuffle partitions to ten. Why am I setting it to ten? Because I know I have ten unique colors. And hence, I will need ten shuffle partitions. So I am setting it to ten. The default value is 200.

SC084-Join Internals Python

demo-cluster

Cmd 1

```
1  spark.conf.set("spark.sql.adaptive.enabled", "false")
2  spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")
3  spark.conf.set("spark.sql.shuffle.partitions", "10")
```

Command took 0.09 seconds -- by prashant@scholarnest.com at 8/1/2022, 7:20:29 PM on demo-cluster

Cmd 2

```
1  product_list = [("P101", "red", 236),
2                  ("P102", "green", 569),
3                  ("P103", "blue", 290),
4                  ("P104", "white", 190),
5                  ("P105", "black", 248),
6                  ("P106", "yellow", 981),
7                  ("P107", "orange", 461),
8                  ("P108", "purple", 834),
9                  ("P109", "magenta", 625),
10                 ("P110", "gray", 712)]
11
12 product_df = spark.createDataFrame(product_list).toDF("prod_id", "color", "price")
```

Command took 2.28 seconds -- by prashant@scholarnest.com at 8/1/2022, 7:20:29 PM on demo-cluster

So if I do not set this to ten, the spark will create 200 shuffle partitions. One hundred ninety of those 200 shuffle partitions will be empty because my data comes with only ten unique colours. But Spark will still create 200 shuffle partitions because Spark doesn't know how many unique colours are in my data frame. The number of shuffle partitions is a performance tuning point in Spark. You need to make sure you set the correct value for this configuration.

SC084-Join Internals Python

demo-cluster

Cmd 1

```
1  spark.conf.set("spark.sql.adaptive.enabled", "false")
2  spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")
3  spark.conf.set("spark.sql.shuffle.partitions", "10")
```

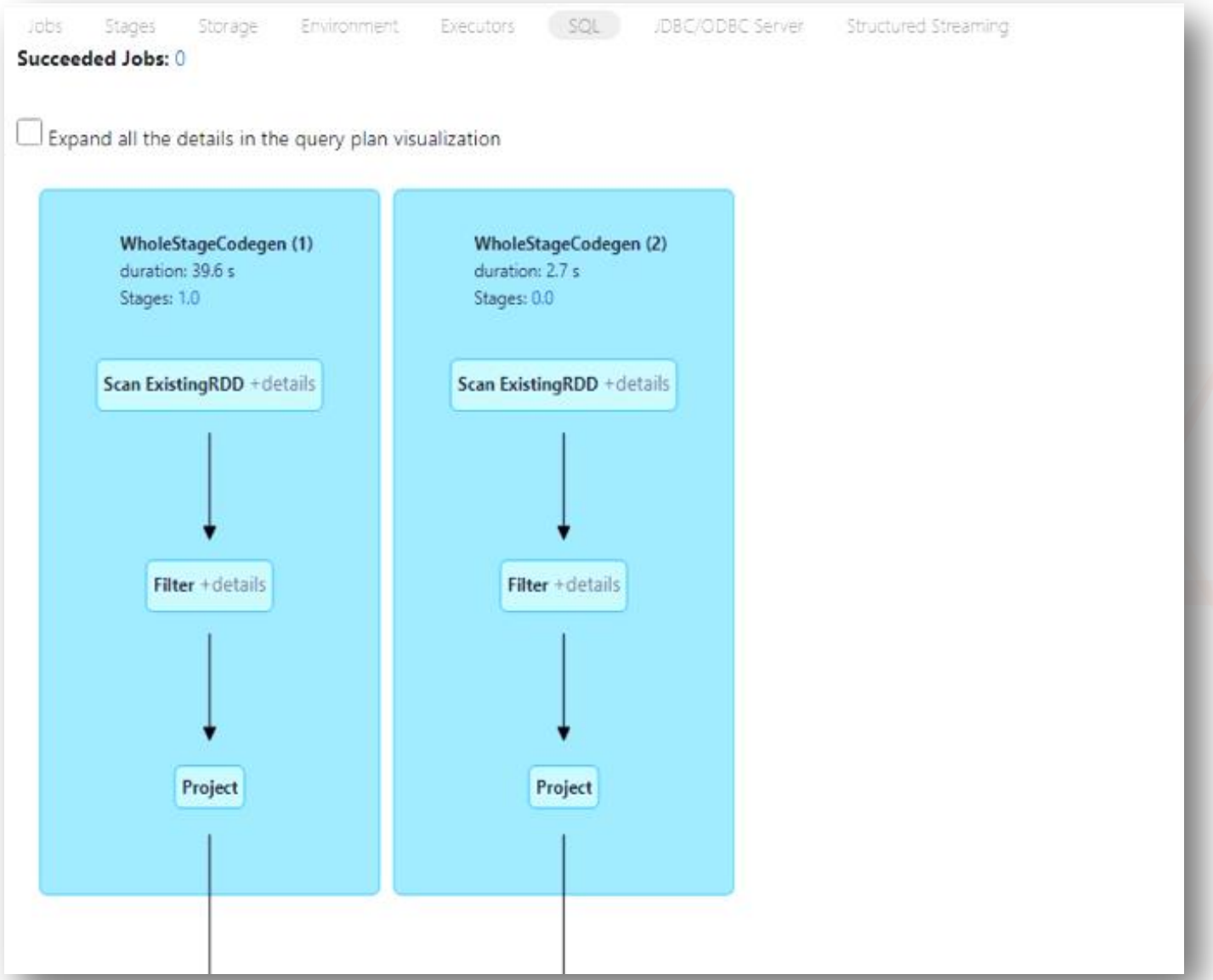
Command took 0.09 seconds -- by prashant@scholarnest.com at 8/1/2022, 7:20:29 PM on demo-cluster

Cmd 2

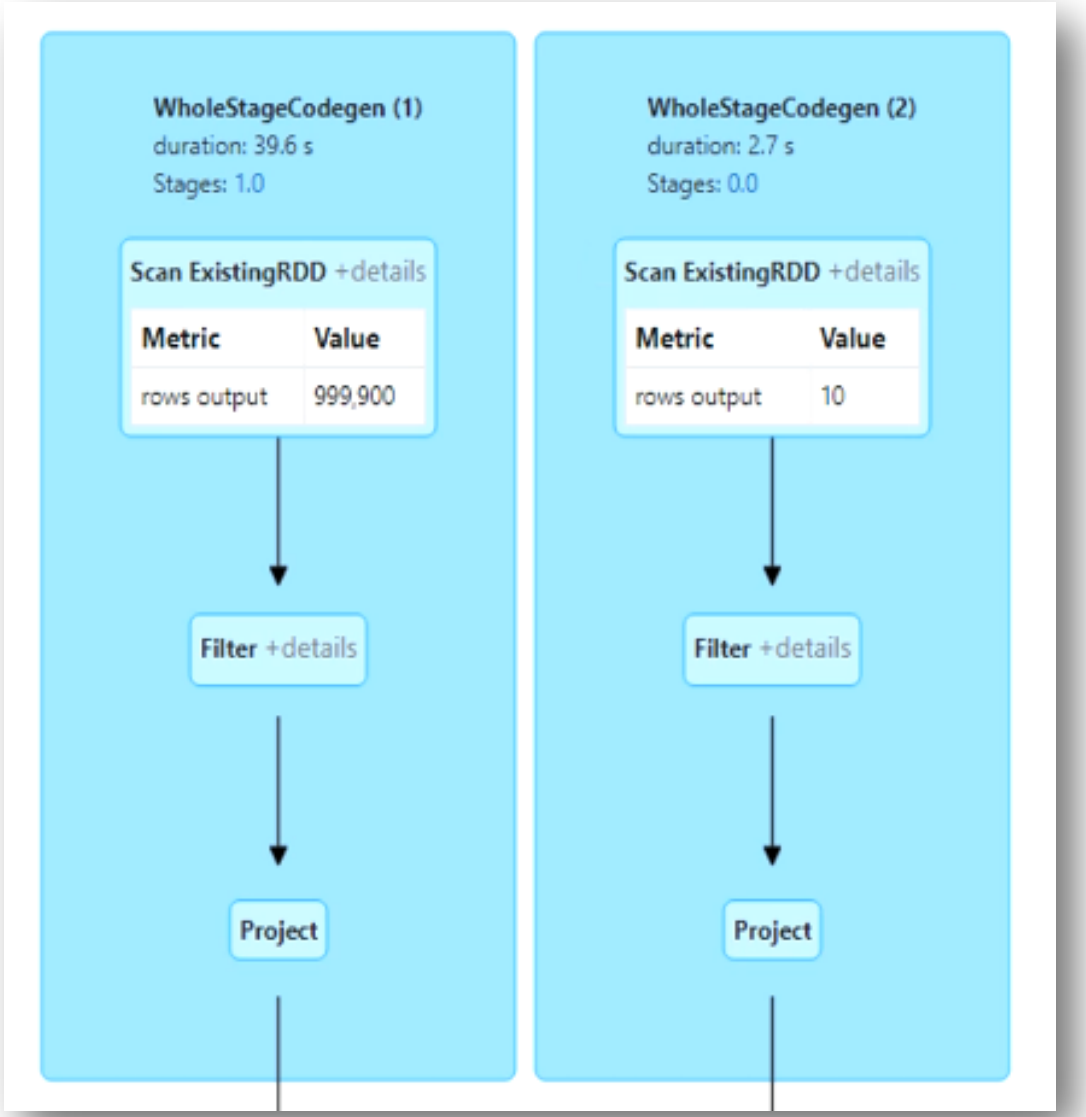
```
1  product_list = [("P101", "red", 236),
2                  ("P102", "green", 569),
3                  ("P103", "blue", 290),
4                  ("P104", "white", 190),
5                  ("P105", "black", 248),
6                  ("P106", "yellow", 981),
7                  ("P107", "orange", 461),
8                  ("P108", "purple", 834),
9                  ("P109", "magenta", 625),
10                 ("P110", "gray", 712)]
11
12 product_df = spark.createDataFrame(product_list).toDF("prod_id", "color", "price")
```

Command took 2.28 seconds -- by prashant@scholarnest.com at 8/1/2022, 7:20:29 PM on demo-cluster

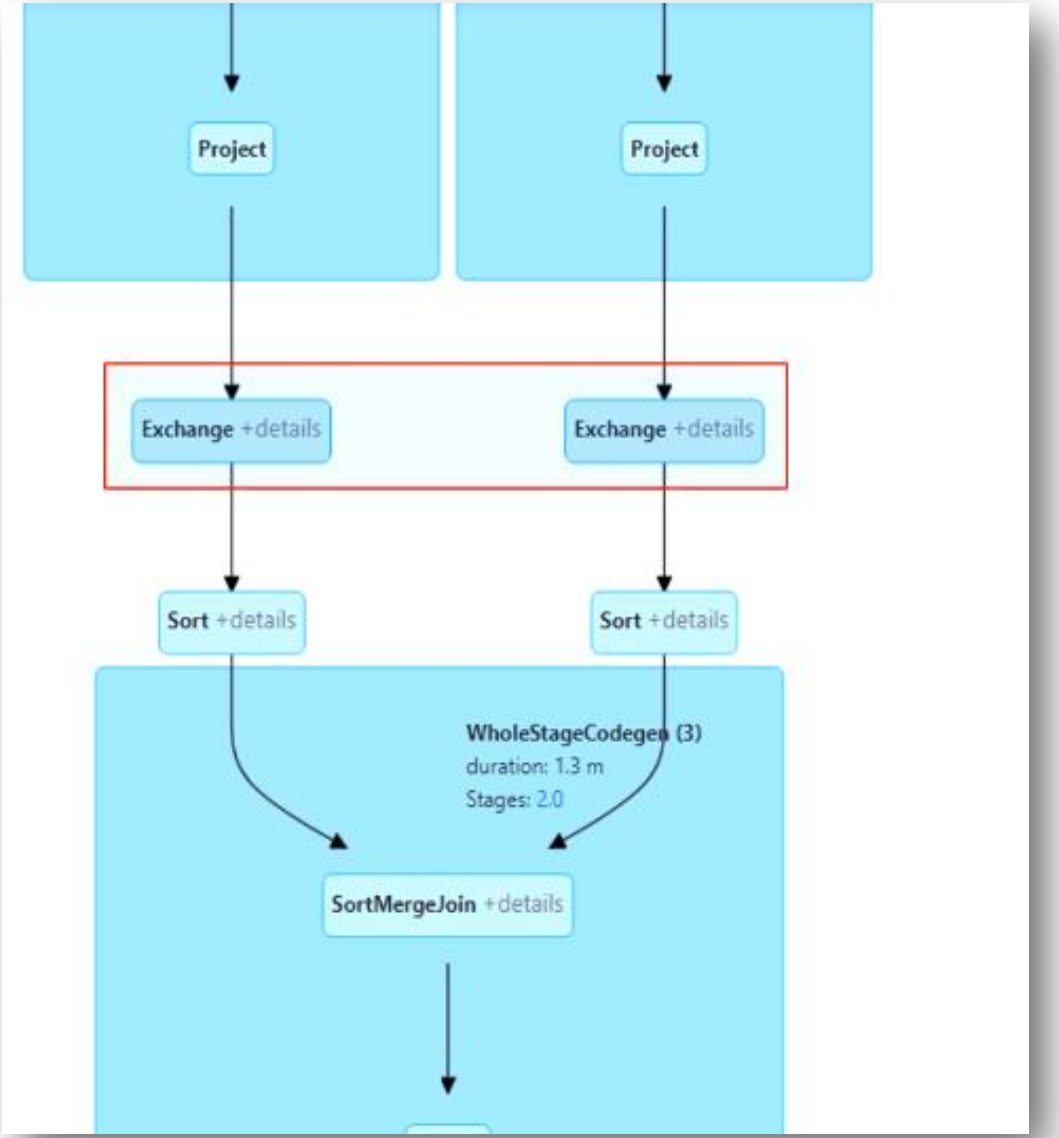
Next, run the notebook and come to the SQL section of your Spark UI, and open the execution plan of the last query.



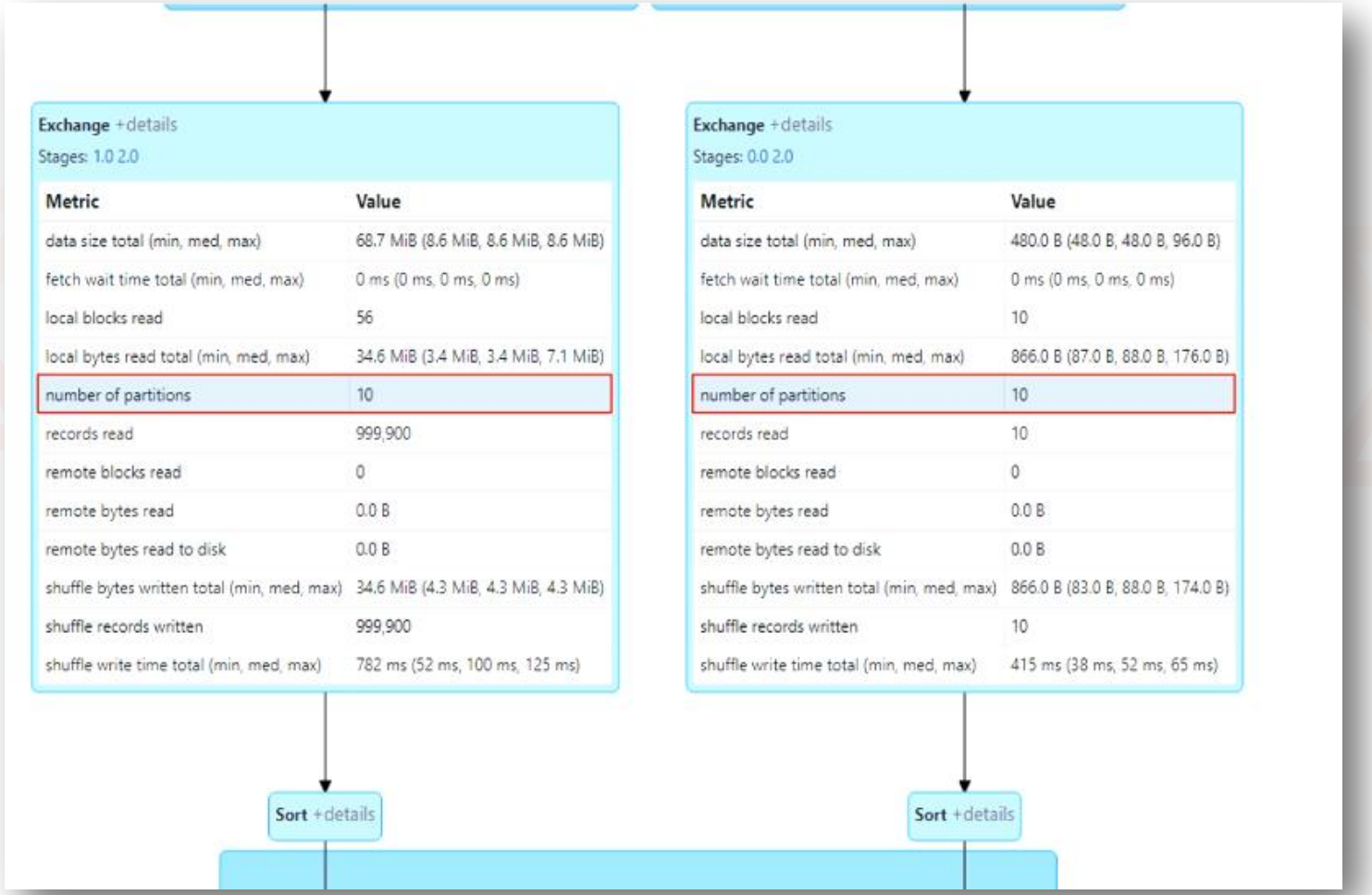
We have stage zero, and we have stage one. Click the +details link to see more details for both stages, and you will see that stage zero reads 10 records, and stage one reads almost a million records.



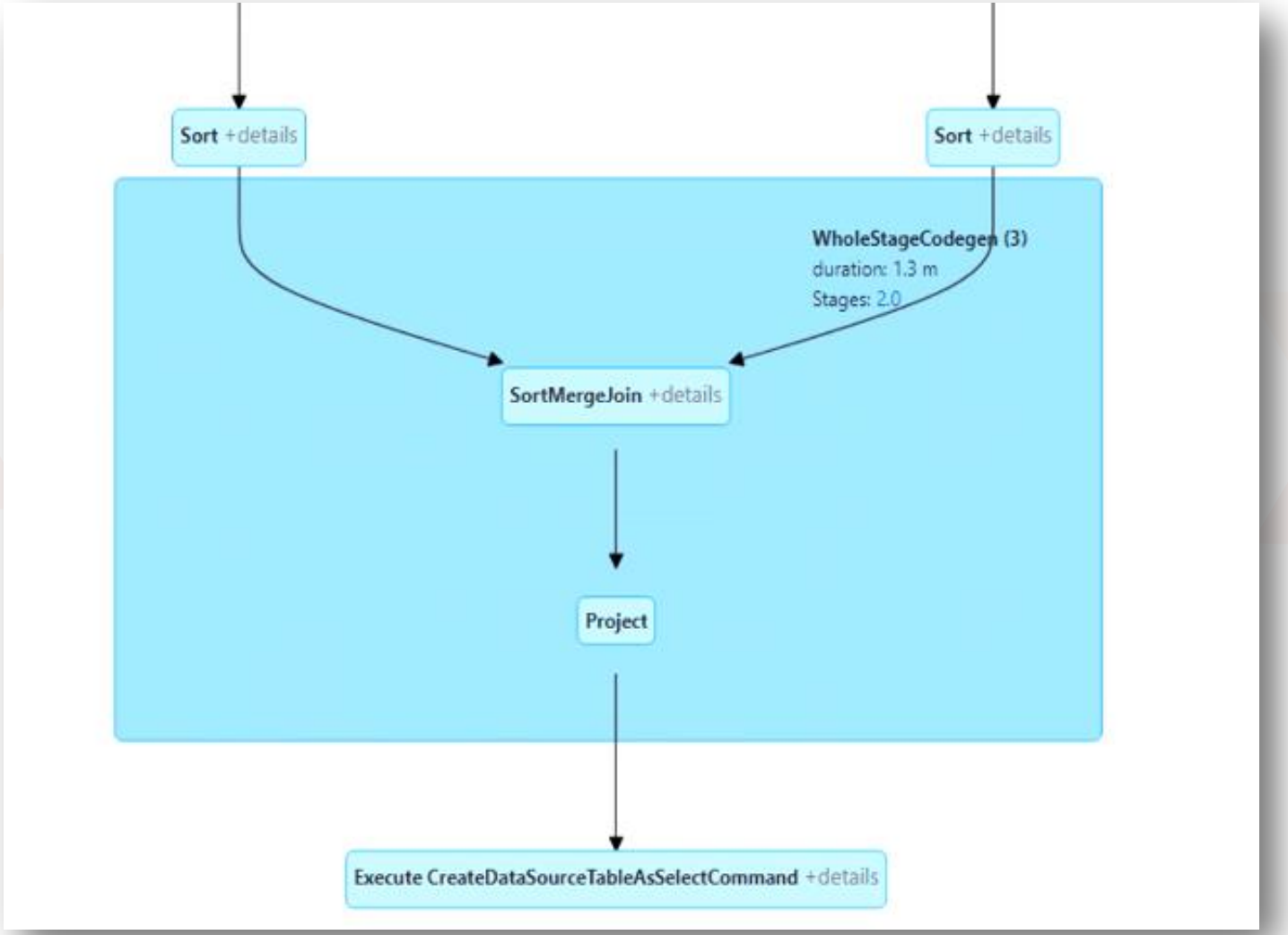
Scroll down, and you will see the exchanges of both stages. Expand the details for both exchanges.



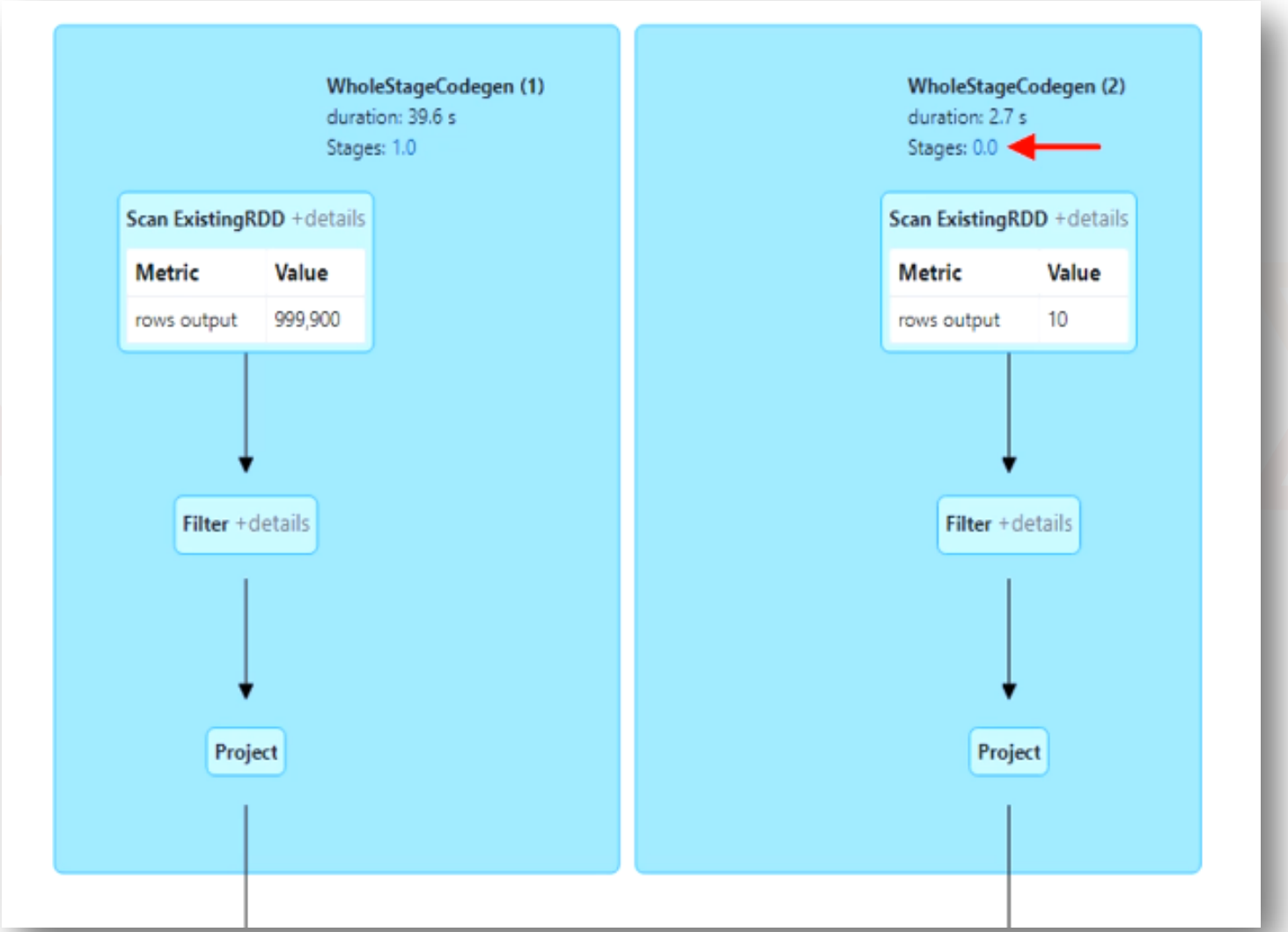
Look for the number of partitions. Both the stages show 10, and that's the number of shuffle partitions. These are not initial partitions. These are shuffle partitions. Why? Because we are looking inside the exchange. And the exchange keeps the shuffle partitions. So we have ten shuffle partitions for both stages.



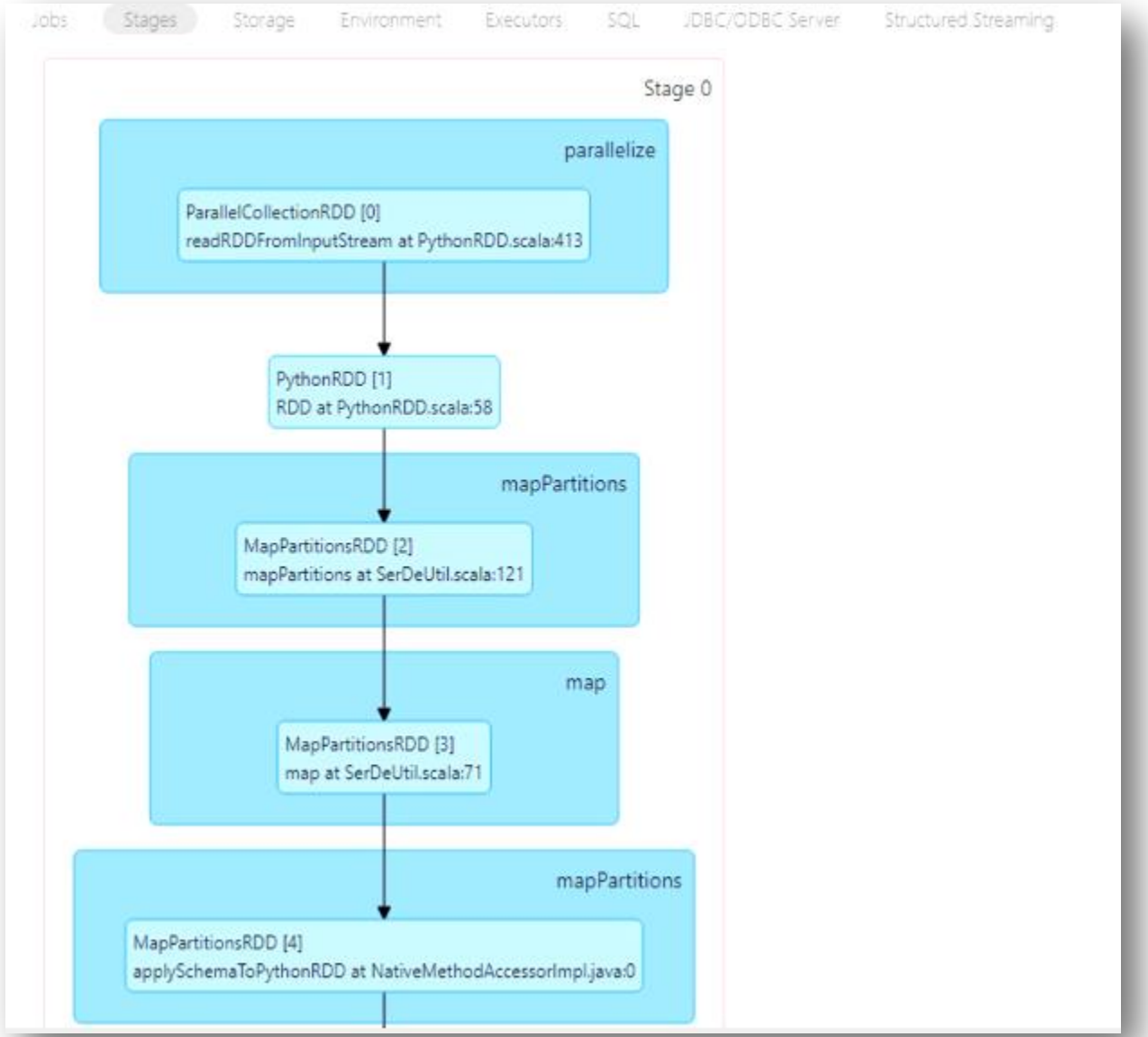
The rest is super simple.
We have to sort and then merge.



Do you want to see the initial partitions? I mean, I showed you the number of shuffle partitions. You can see that in the exchange. But where do we see the initial partitions? Scroll up and go to stage zero. Click stage 0.0, and it will take you to the details of stage zero.



You will see a DAG for stage zero. The stage DAG doesn't tell much. So leave the DAG and scroll down.



Do you see these eight bars? These are the eight initial partitions of stage zero. The bar shows the time taken by each partition.

Scroll down, and you will see the list of eight tasks. You can see eight rows here.
Each row represents one task and hence one partition.

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 Write Size / Records	Errors
0	8	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:39	0.4 s		87 B / 1	
1	9	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.4 s		88 B / 1	
2	10	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.3 s		83 B / 1	
3	11	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.2 s		174 B / 2	
4	12	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.2 s		88 B / 1	
5	13	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.1 s		88 B / 1	
6	14	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.1 s		88 B / 1	
7	15	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.1 s		170 B / 2	

Showing 1 to 8 of 8 entries

Previous Next

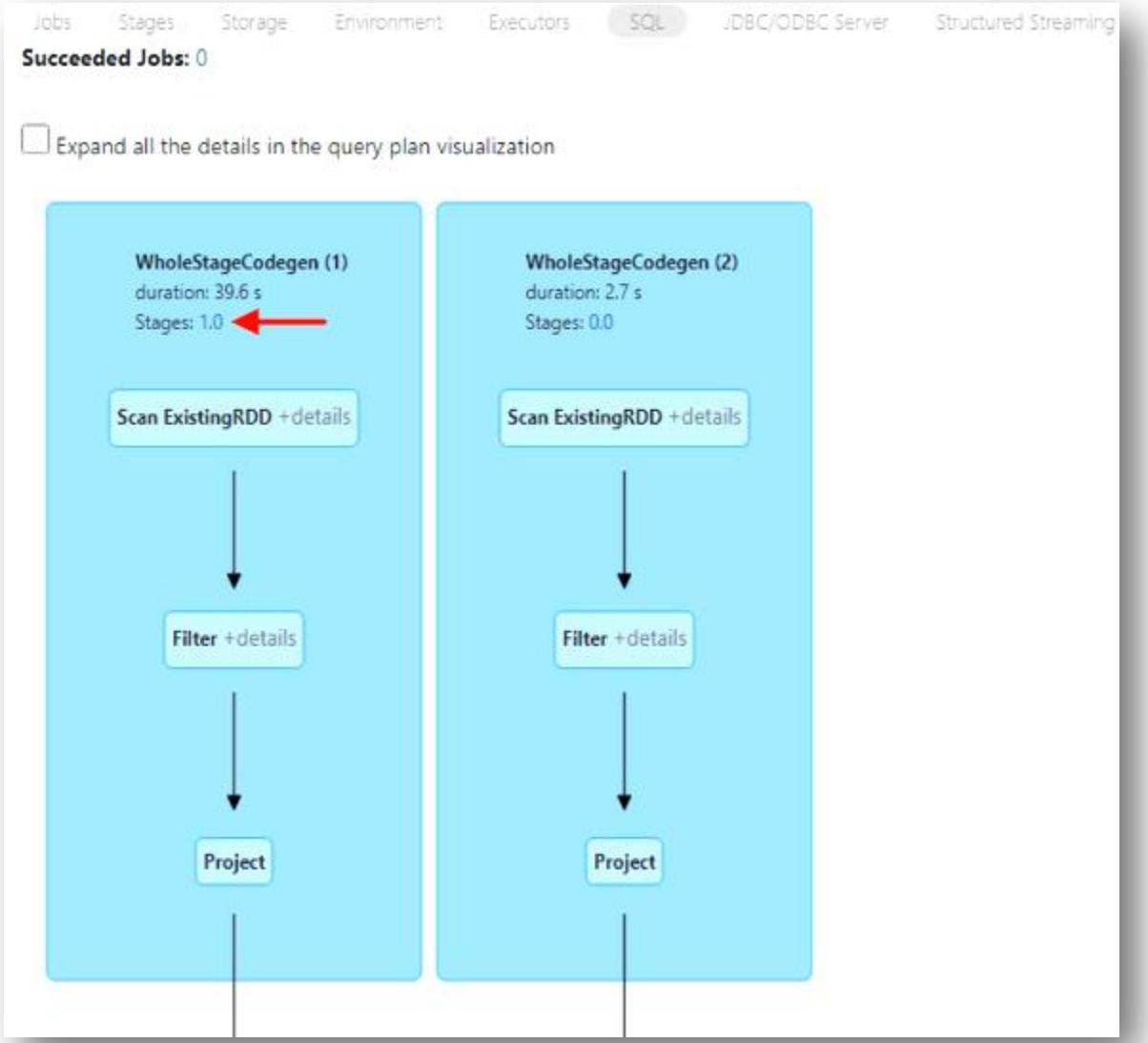
Look at the last column. It gives you the number of records in each partition.
We have two records in these two partitions and just one record in all other partitions.

Tasks (8)

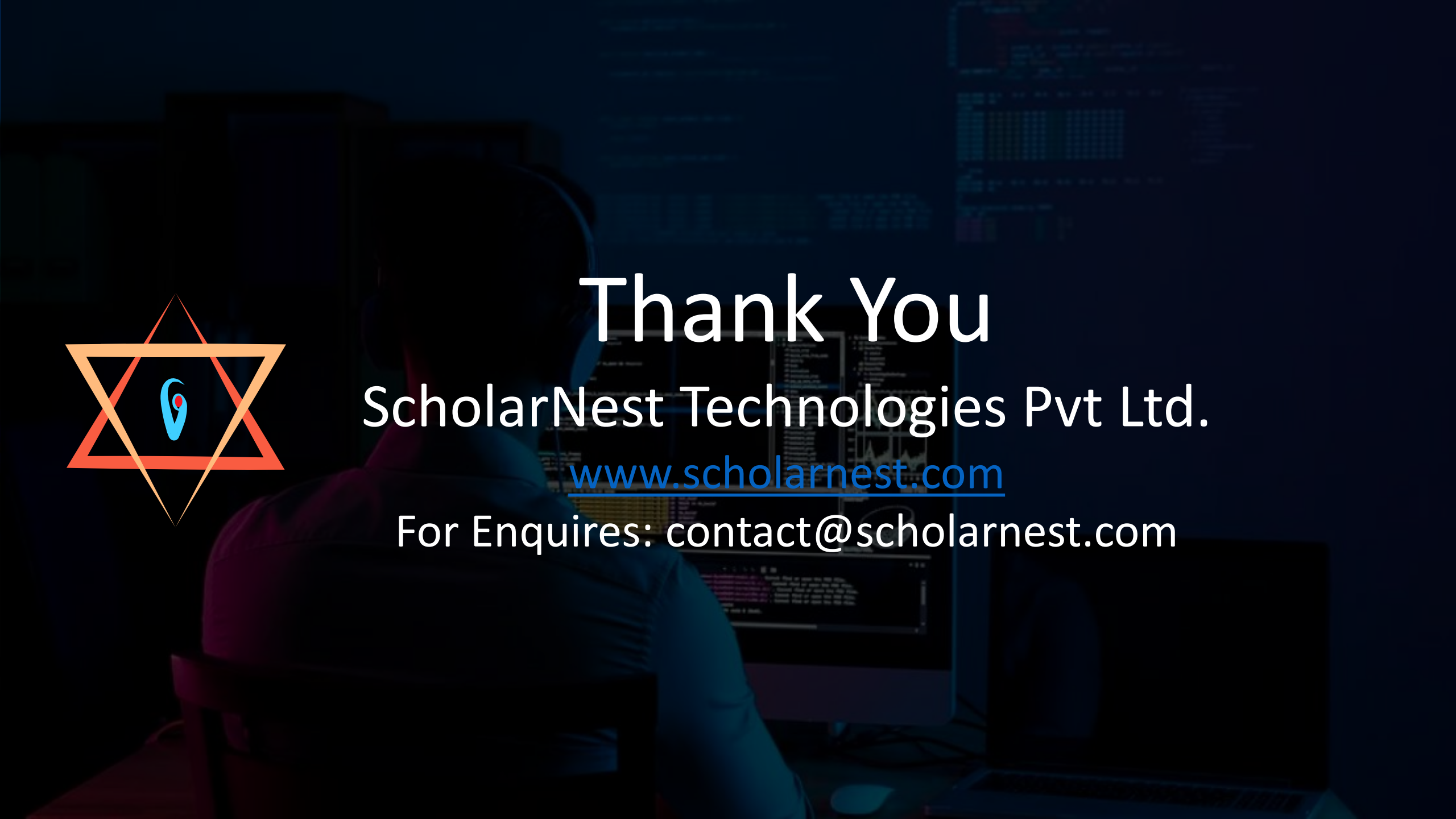
Show 20 entries Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Write Size / Records	Errors
0	8	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:39	0.4 s		87 B / 1	
1	9	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.4 s		88 B / 1	
2	10	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.3 s		83 B / 1	
3	11	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.2 s		174 B / 2	
4	12	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.2 s		88 B / 1	
5	13	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.1 s		88 B / 1	
6	14	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.1 s		88 B / 1	
7	15	0	SUCCESS	PROCESS_LOCAL	driver	ip-10-172-211-210.us-west-2.compute.internal		2022-08-04 19:19:40	0.1 s		170 B / 2	

Now let's go back to the SQL tab and open the execution plan again. Similarly, you can click on the stage 1. link and see the details of stage one. I leave it for you and an exercise.



So we learned how Spark performs the join operation.
The Joins in Spark are complex and time-consuming operations.
They are often the main cause of performance problems in Spark.
They are also the main reasons behind OOM exceptions.
This course section focuses on learning techniques to avoid the Join performance problems and OOM exceptions.
I covered join internals, so you know what happens under the hood.
And you can easily understand some performance tuning approaches explained in the rest of the section.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Joins and Optimization

Lecture:
Optimizing your Joins using Broadcast

Optimizing your Joins using Broadcast

In this lesson, we will take that learning further and understand some critical considerations for avoiding pitfalls of your shuffle join.
We will also learn about the broadcast joins in Spark and when and how to use broadcast joins.

Joining two Dataframes can bring the following two scenarios:

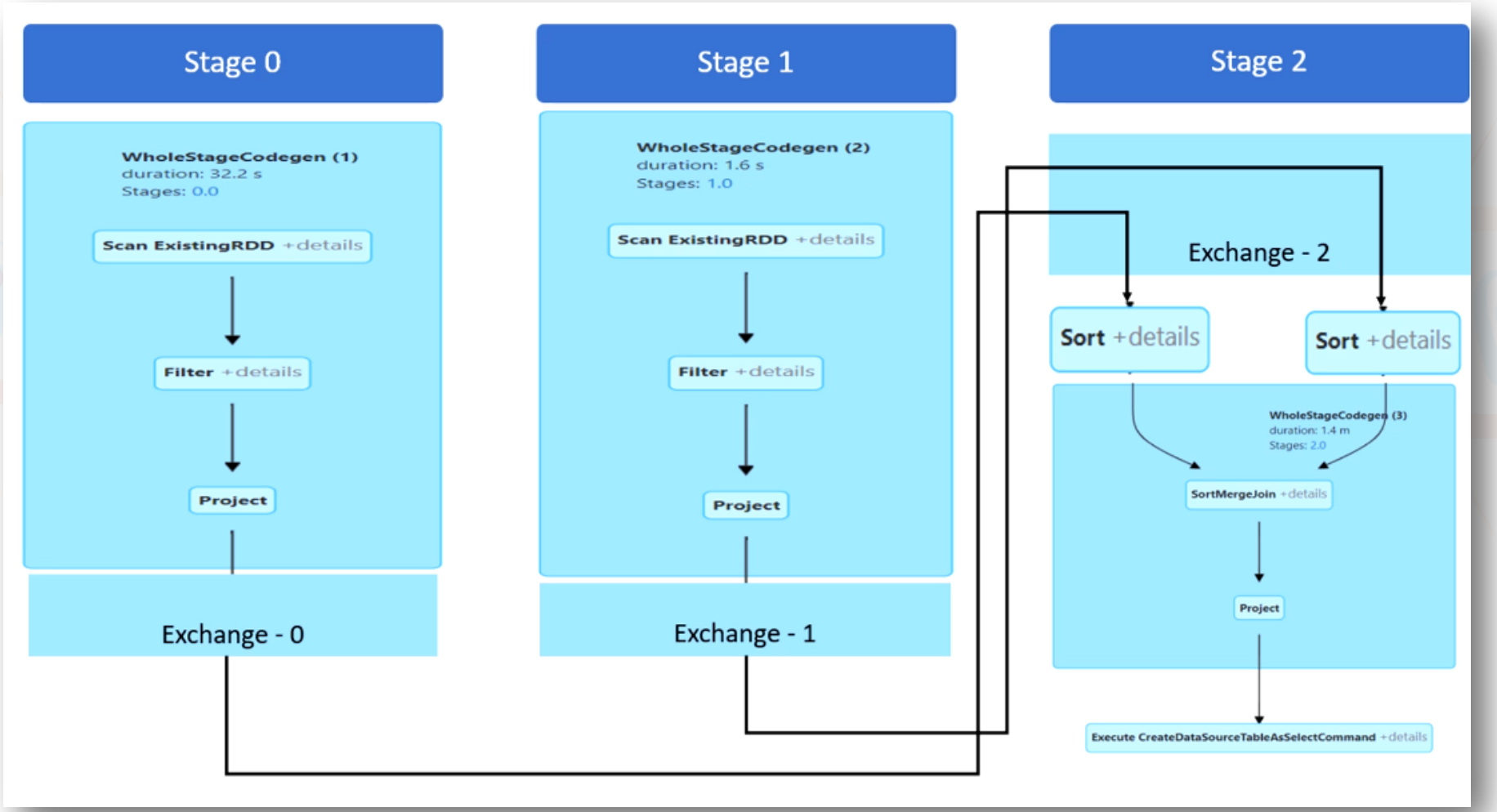
1. Large to Large: You are joining a large data frame with another large data frame. When we say large, we mean large enough not to fit into the memory of a single executor or the memory of the driver.
2. Large to Small: The second scenario is to join a large Dataframe with a tiny data frame. We consider it small when the Dataframe can fit into the driver memory.

When both of your Dataframes are large, you should watch out for the following things in your join operation:

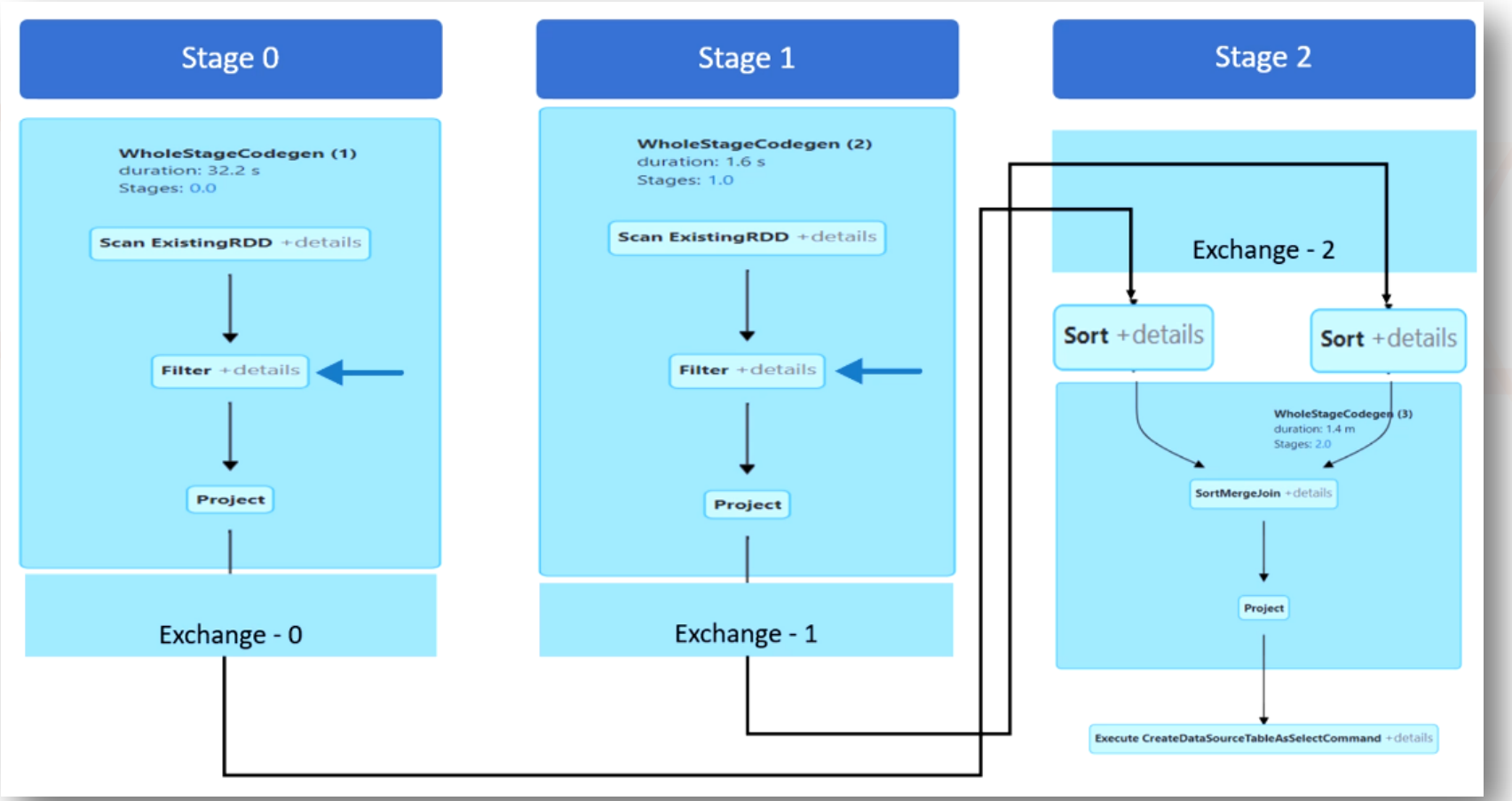
1. Apply your filters as early as possible.
2. Shuffle partitions
3. Plan Executor Cores
4. Key distribution and data skew

The first consideration is not specific to Apache Spark.
It is a general common sense which applies to any other data technology stack.

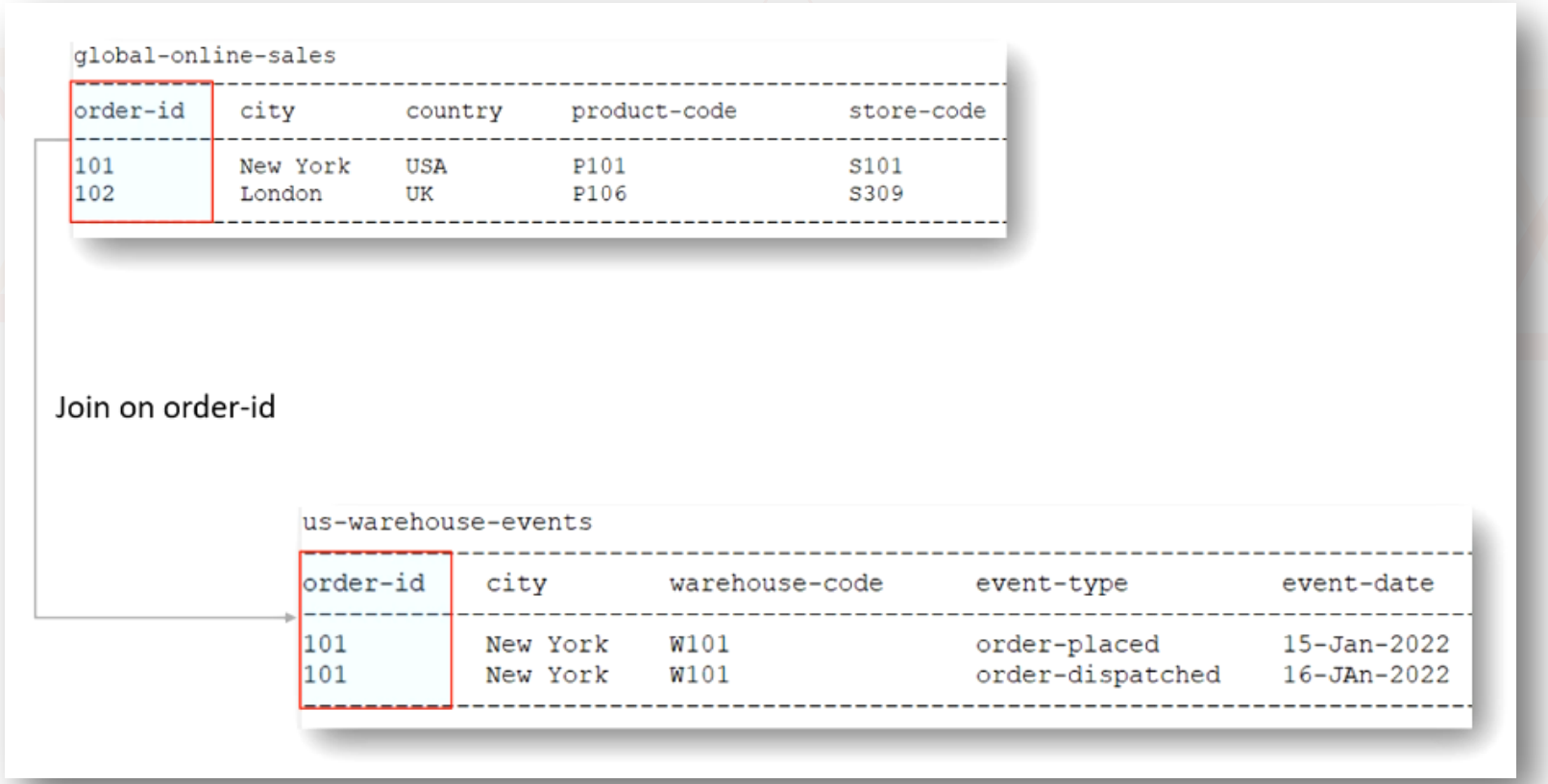
Let me highlight a couple of common mistakes. Let's start with the execution plan for the Join operation shown below. We already learned that shuffle join will send all your data from one exchange to the other. So, in this case, data from exchange-0 will move to exchange-2. Similarly, data from exchange-1 will also move to exchange-2.



So, filtering unnecessary data before you perform a join will obviously cut down the amount of data sent from exchange-0 and exchange-1.
Apply filters on your data frames before applying the Join so you will send fewer data from one exchange to another. Sometimes such filtering is obvious, but more often, these things are not apparent unless you know enough about your datasets.

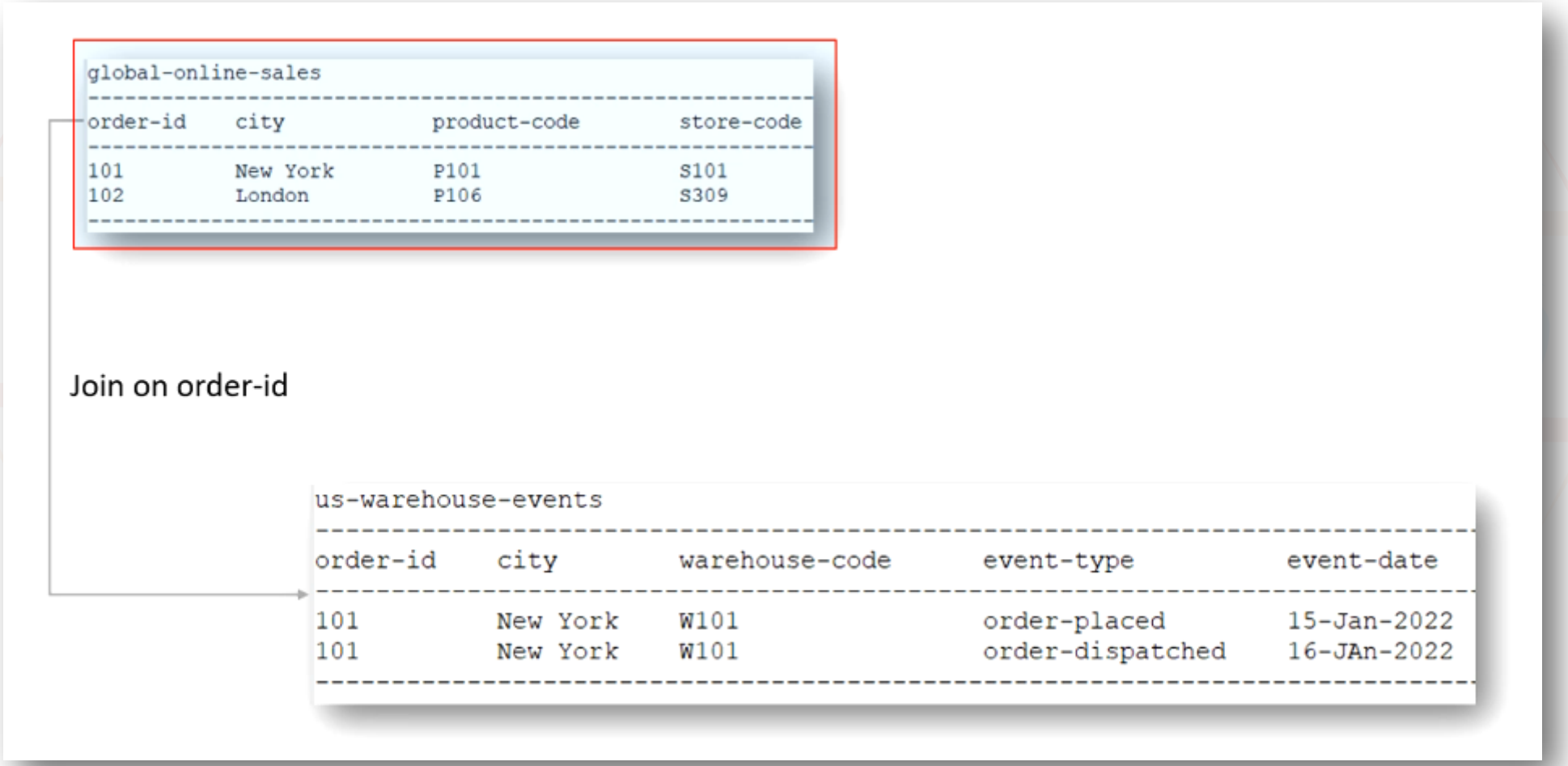


Let's consider the following datasets. The first one is the global online sales, and the second one is only for the US warehouse event dataset. You want to perform a join by order_id, and it looks obvious to join them without prior filtering. However, you know that orders from European countries such as the UK will not match orders and will be filtered out in the Join operation. It makes perfect sense to filter for the US orders from the global_sales even before we perform a join operation.

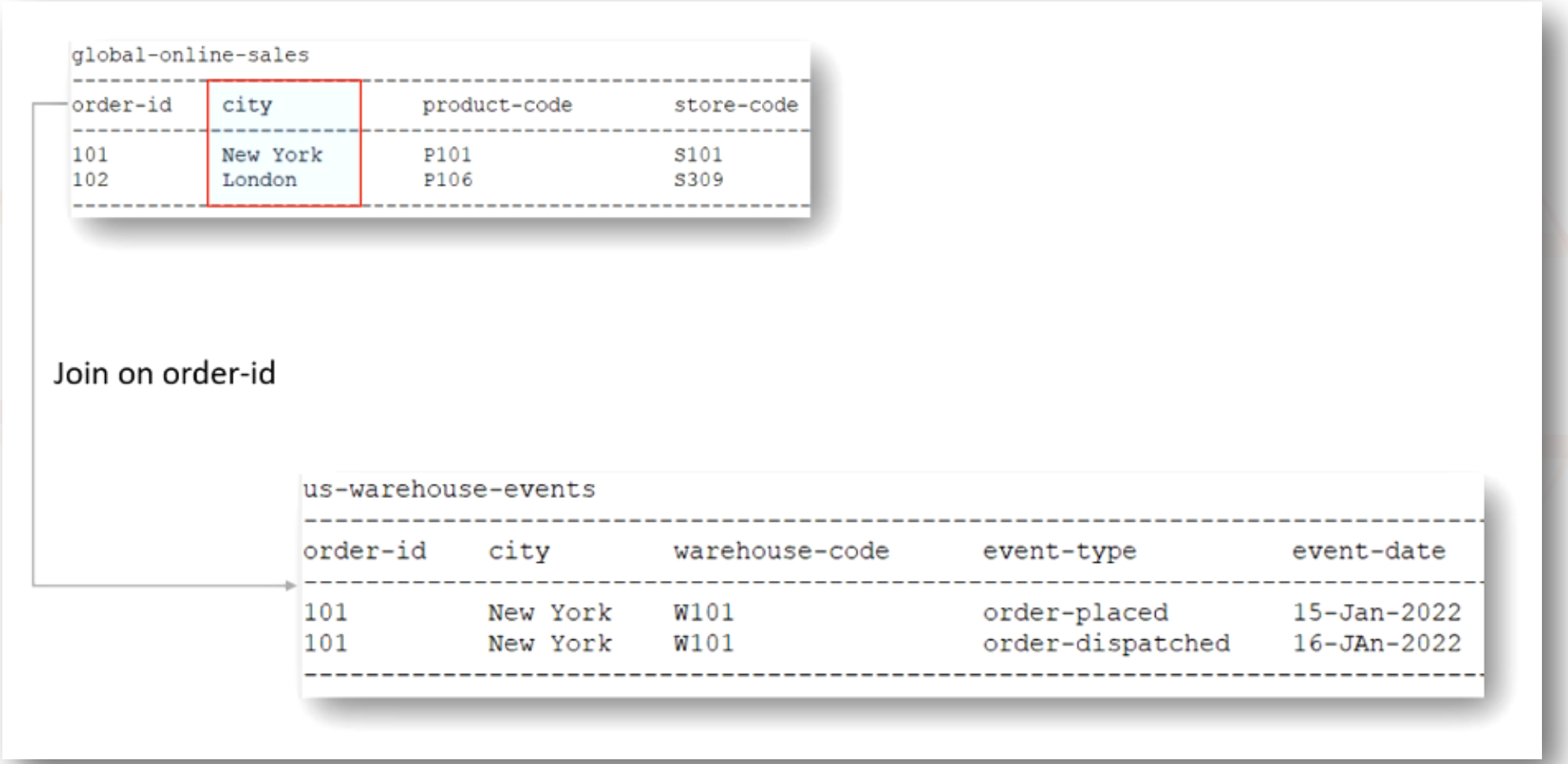


You already have a country field. Just apply filters for the US before you join, and you will reduce the amount of data in the first Dataframe.

Sometimes, you may not have a country field in your first Dataframe as highlighted below. You do not have a country in this data frame. So you cannot filter it using the country name. But you know you need only US data because you want to join it with us-warehouse-events.



You have a city.
So look for a city-to-country mapping table.



Let's assume I have another table for city and country mapping.
The city-to-country mapping table is small. It may be just a few MB of data.

global-online-sales

order-id	city	product-code	store-code
101	New York	P101	S101
102	London	P106	S309

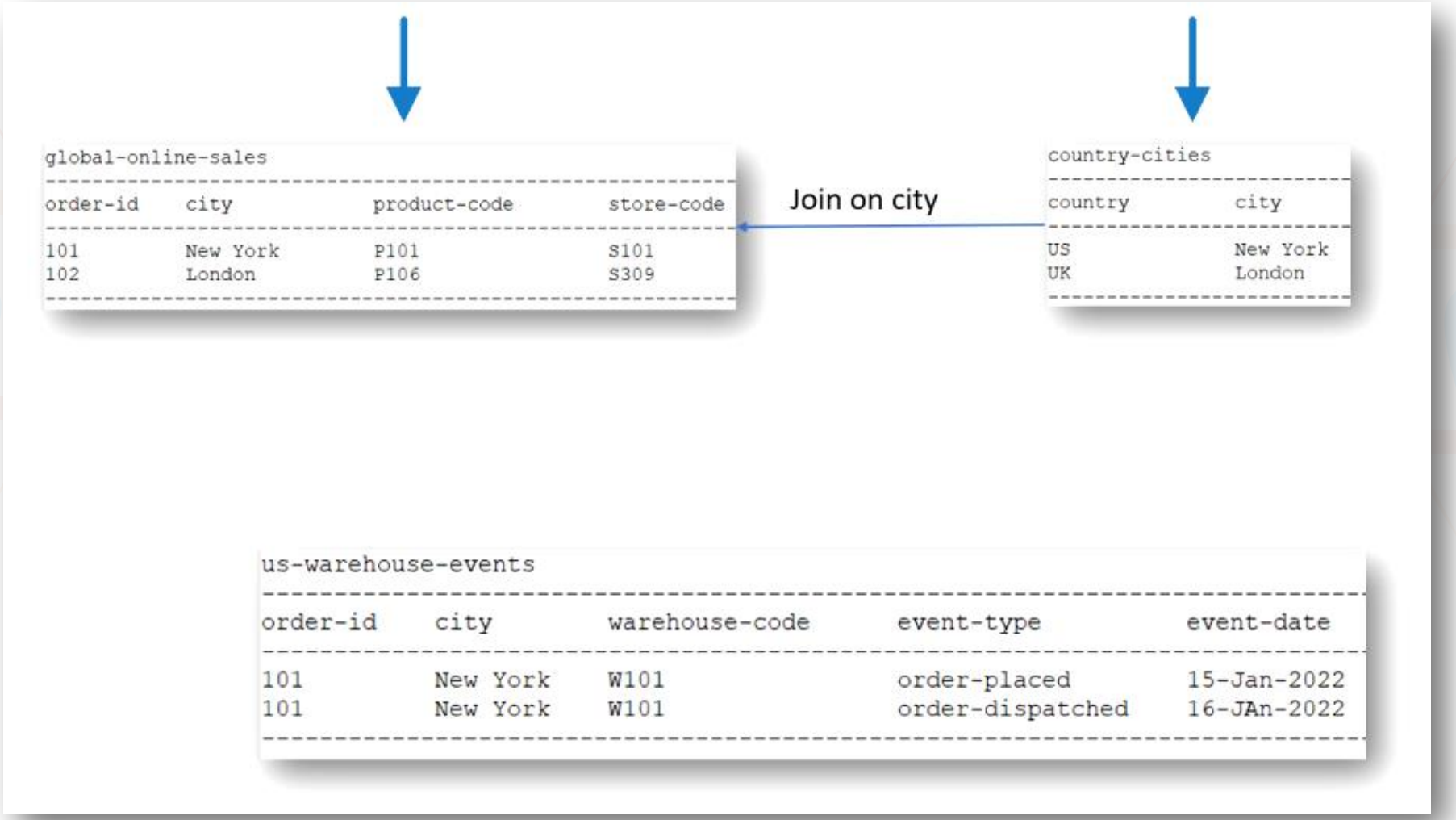
country-cities

country	city
US	New York
UK	London

us-warehouse-events

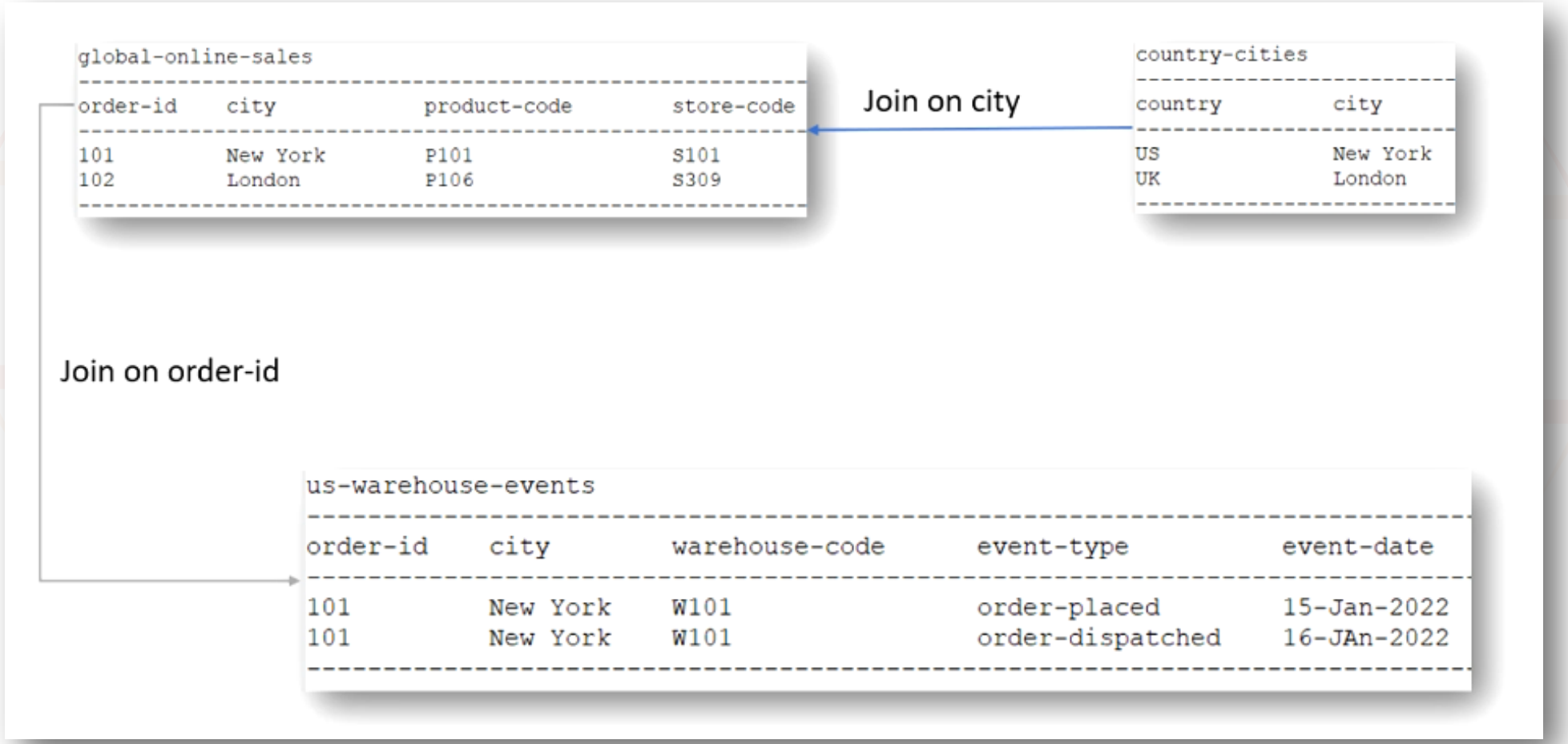
order-id	city	warehouse-code	event-type	event-date
101	New York	W101	order-placed	15-Jan-2022
101	New York	W101	order-dispatched	16-JAn-2022

So you can join these two tables first and filter out the data to keep only US country orders.
Joining one large order data frame with a small country Dataframe will be fast.
And this Join will eliminate a lot of records from the orders Dataframe.



Now you can join in with the warehouse events data frame.
This approach will reduce your data size and improve the performance of your join operation.

In this approach, we are performing two joins. But if you can filter a sizable amount of data in the first small table join, your large table join will perform faster. The combined time for both the Join might be significantly lower than the time to join two large data frames.



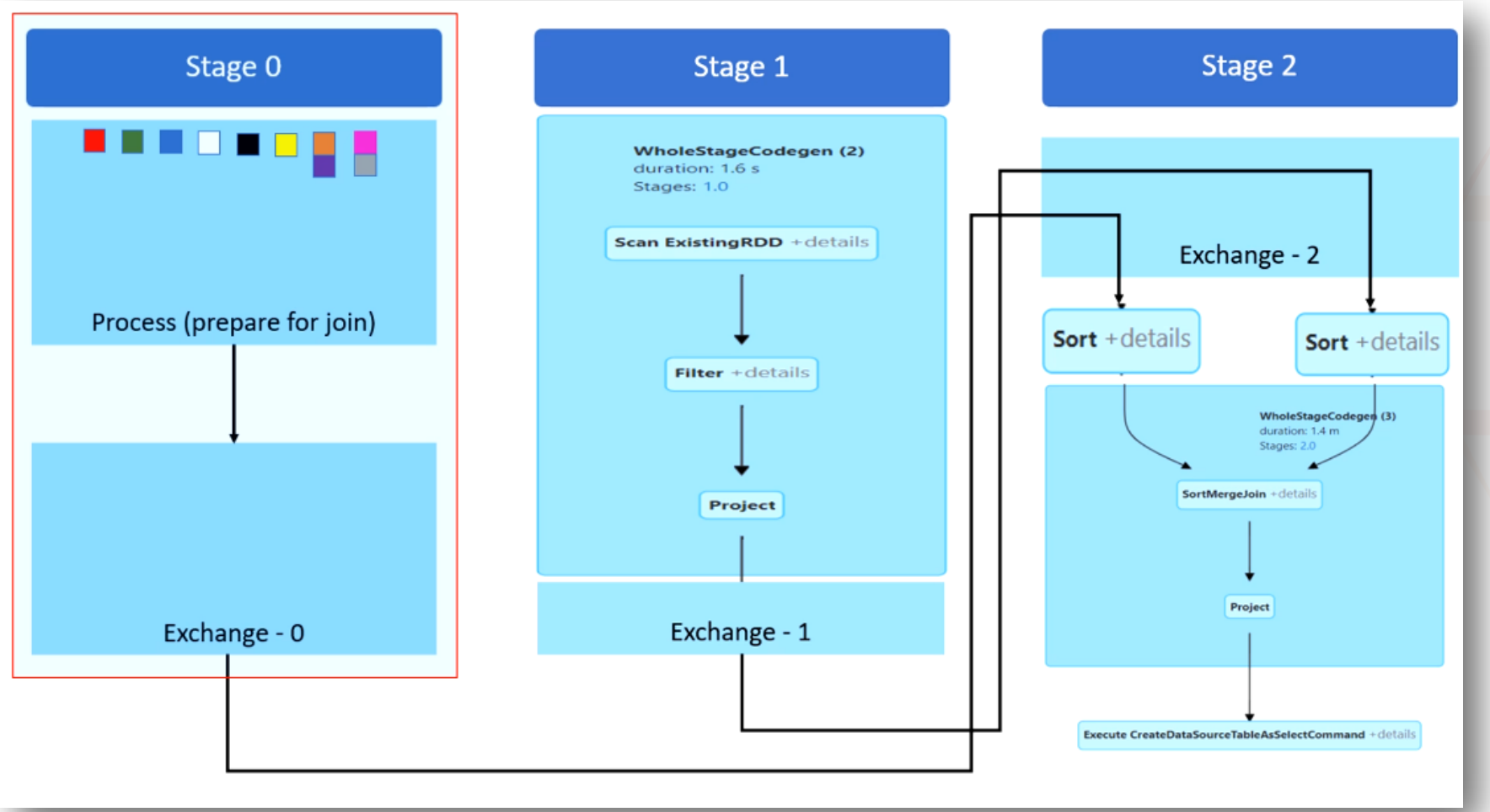
So, look for all the possible opportunities to reduce the Dataframe size.
I have seen examples where people join the Dataframes and then perform aggregations on the combined Dataframes.
The recommendation is to look for opportunities where you can aggregate even before joining the Dataframes.
The objective is almost always to cut down the size of your Dataframes as early as possible.
Smaller the Dataframe will result in a smaller shuffle and faster Join.

The second step in optimizing Join is to plan out the two things.

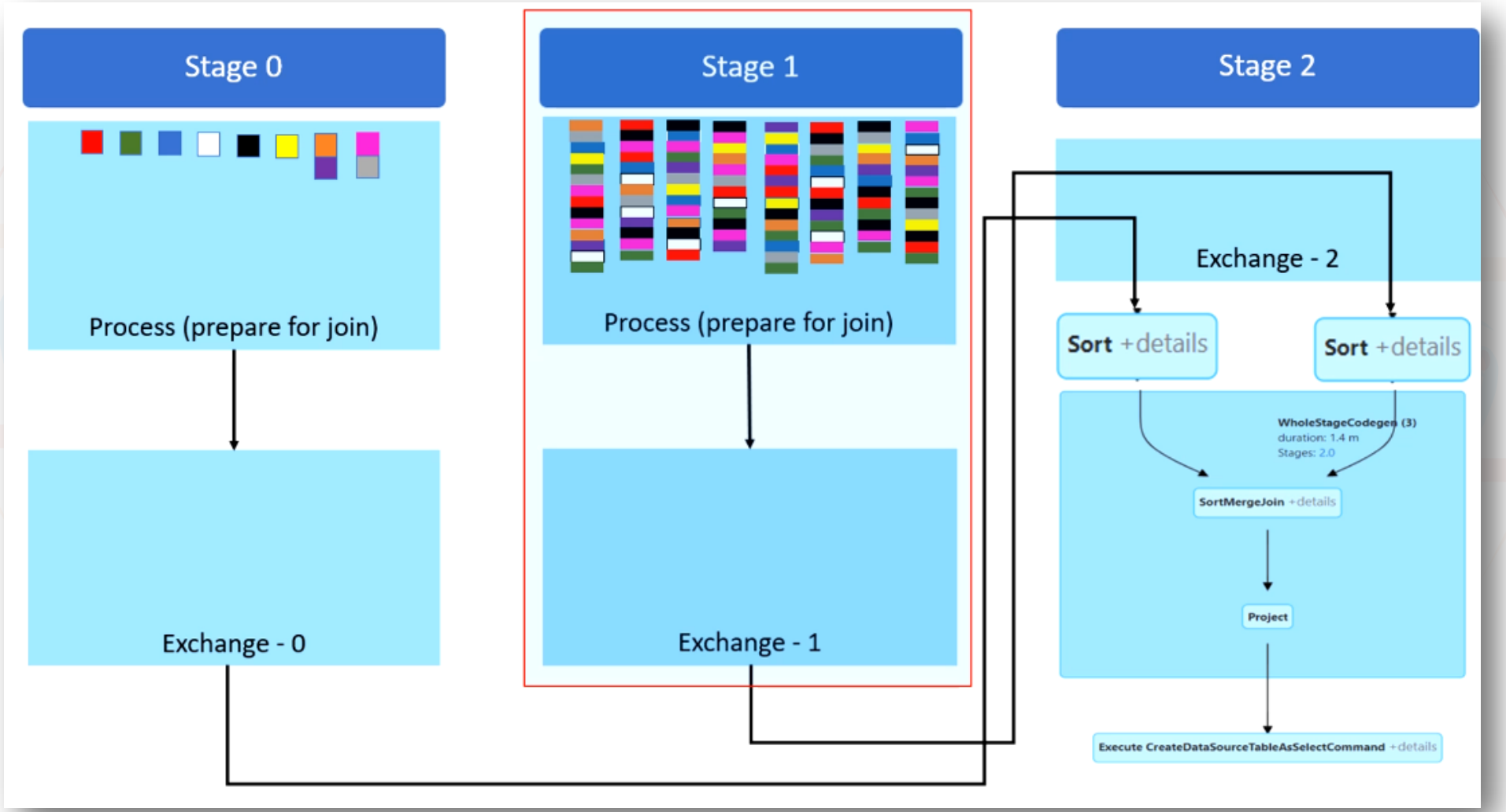
1. Shuffle partitions
2. Number of executors

Let's try to understand it.

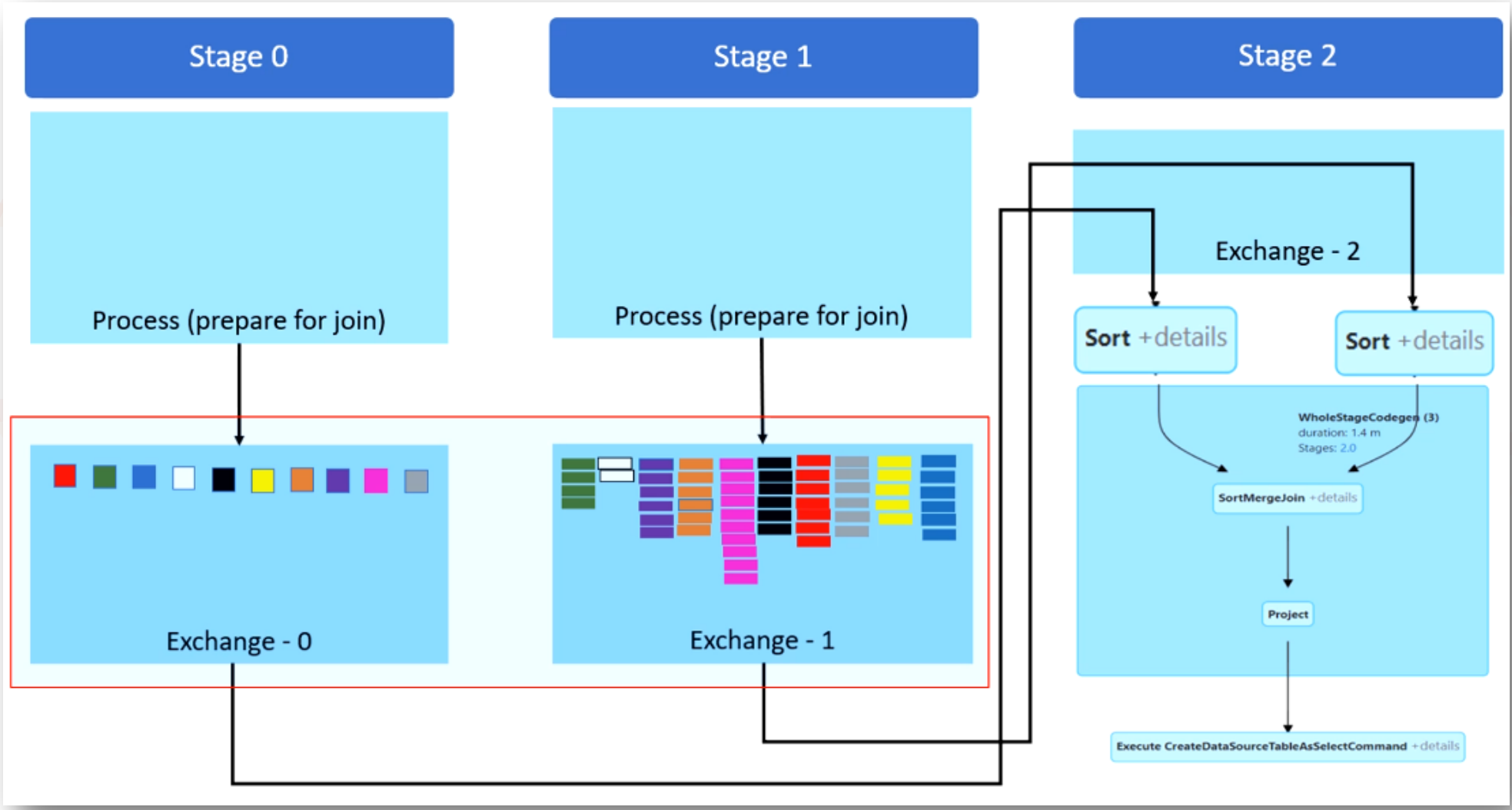
Recall your understanding of the Dataframe Join. We have 3 stages in the DAG shown below. The first stage reads the first Dataframe and creates initial Dataframe partitions. The driver determines the initial number of Dataframe partitions for stage one. We do not have many tuning opportunities for the initial partitions because the driver does a good job there.



The second stage will also read the second Dataframe and create some initial partitions. The driver also determines those initial partitions, and we do not have many tuning opportunities.



Then both these stages will partition the data using the join keys and create shuffle partitions. These shuffle partitions should go to the third stage for joining. The number of shuffle partitions is a critical factor for your join performance.

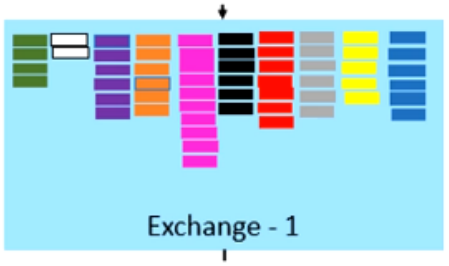


We need to make sure of two things:

1. Shuffle partition size should be moderate
2. Shuffle partitions are evenly sized

The first thing is to make sure that all the shuffle partitions are of almost equal size.
You might get one shuffle partition of 10 MB, another one of 200 MB, and the third one is 5 GB.
And that's a big problem. This problem is known as the data skew problem.
You must handle this data skew problem and ensure that all the partitions are of almost equal size.

The ideal case for spark is to have 128 MB to 1 GB shuffle partitions. That's the best. So if you have 10 GB of data, you should try getting 80 shuffle partitions of 128 MB each. It is not necessary to have exactly 128 MB partitions. But we always try to achieve close to 128 MB shuffle partitions.



Plan Shuffle partitions

Shuffle partition size sweet spot (128 MB to 1 GB)

Examples

Data Frame size 10 GB

→ Ideal shuffle partition size = 10 GB / 128 MB = 80

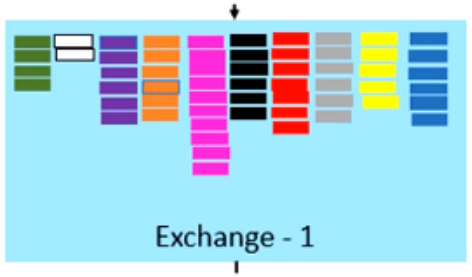
Data Frame size 1 TB

Ideal shuffle partition size = 1 TB / 1 GB = 1000

Configure Shuffle partitions

spark.sql.shuffle.partitions

However, for a one terabyte Dataframe, achieving 128 MB will result in approximately eight thousand shuffle partitions. So you can have 1 GB shuffle partitions instead of 128 MB, resulting in approximately one thousand shuffle partitions.



Plan Shuffle partitions

Shuffle partition size sweet spot (128 MB to 1 GB)

Examples

Data Frame size 10 GB

Ideal shuffle partition size = 10 GB / 128 MB = 80

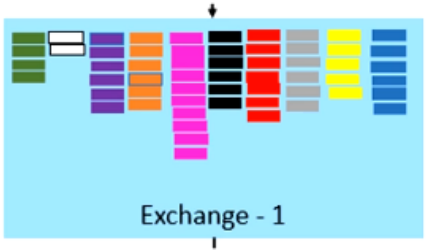
Data Frame size 1 TB

→ Ideal shuffle partition size = 1 TB / 1 GB = 1000

Configure Shuffle partitions

spark.sql.shuffle.partitions

Once you are done with your planning, you can set the spark.sql.shuffle.partitions.
For the first example, I will set the spark.sql.shuffle.partitions to eighty. And for the second example, I will set it to 1000. If you do not set anything, the default value is 200. The default value does not fit your requirement. So do not forget to plan and set the shuffle partitions for the best performance.



Plan Shuffle partitions

Shuffle partition size sweet spot (128 MB to 1 GB)

Examples

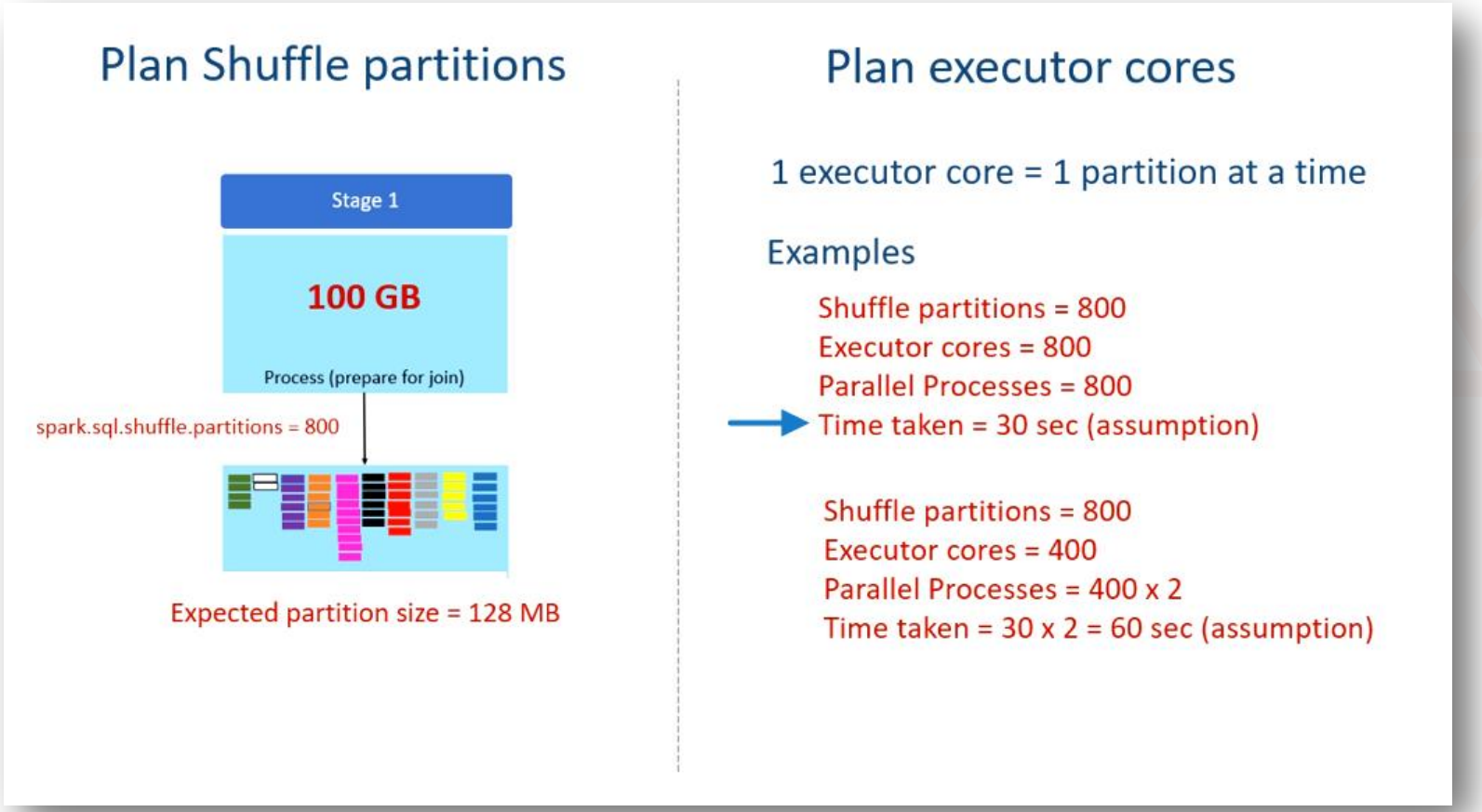
Data Frame size 10 GB
Ideal shuffle partition size = 10 GB / 128 MB = 80

Data Frame size 1 TB
Ideal shuffle partition size = 1 TB / 1 GB = 1000

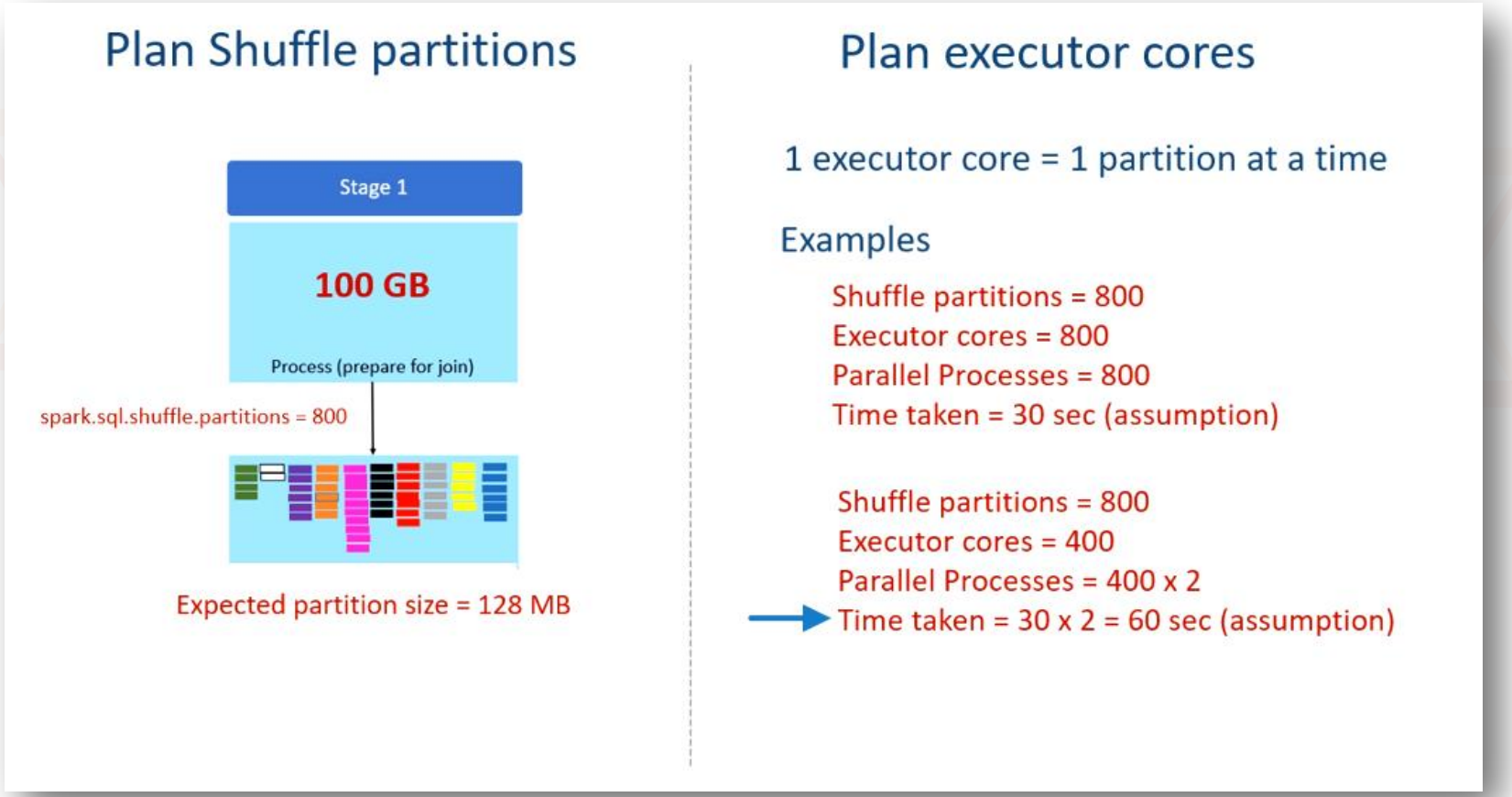
Configure Shuffle partitions

→ spark.sql.shuffle.partitions

The second consideration is to have enough executors to perform the Join operation. Let's assume you have a 100 GB Dataframe. You are setting 800 shuffle partitions and expecting to get 128 MB to shuffle partitions. But you know, one executor core can process only one partition at a time. So if you have 800 executor cores and 800 shuffle partitions, you will process all in parallel. Suppose it takes 30 seconds to process 800 partitions in parallel because you have 800 cores.



But what if you have only 400 executor cores? Spark will process 400 shuffle partitions in parallel, and then the remaining 400 will wait for the executor core to become free.
So your 800 partitions are processed in two batches of 400 each.
And it will take 60 seconds to process the same 800 partitions on 400 executor cores.



So you have two parameters:

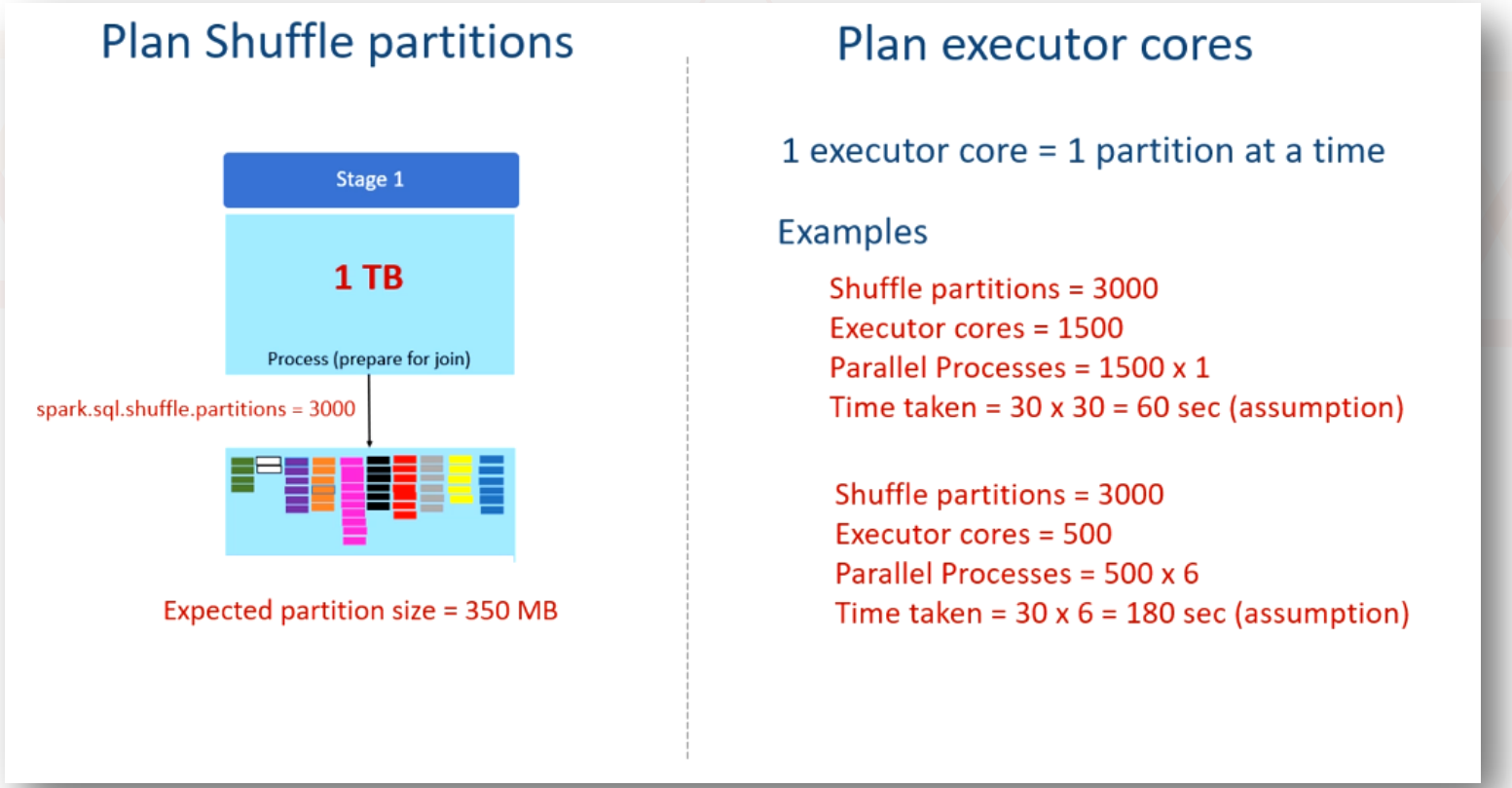
1. Number of shuffle partitions
2. Number of executor cores.

You need to find the balance between these two. And also, make sure your shuffle partition size is in the range of 128 MB to 1 GB.

So, you join performance depends on three things:

1. **Data volume** – The first critical concept is to reduce your join data volume. Apply filters or do whatever you can do to take what is necessary to join. Do not join unnecessary data.
2. **Shuffle partition size and number of shuffle partitions** – The second concept is calculating the number of shuffle partitions to keep the partition size within 128 MB to 1 GB. Smaller is better. The best performance is achieved with close to 128 MB shuffle partitions.
3. **Executor cores** – The last essential is to have enough cores to achieve maximum parallelism. If you want to process 100 shuffle partitions, the best number is to have 100 executor cores.

Then we have another problem to ensure that your shuffle partitions are not skewed. Setting *spark.SQL.shuffle.partitions* does not protect you against the data skew problem.
Let's look at the scenario. I have a 1 TB dataframe. I ask Spark to create 3000 shuffle partitions. So Spark will try to create 350 MB partitions. However, if you have a Join key with 2 GB, Spark cannot break it into two partitions. So your shuffle partition count is not a protection against the data skew. You can still have data skew, depending on the join key.



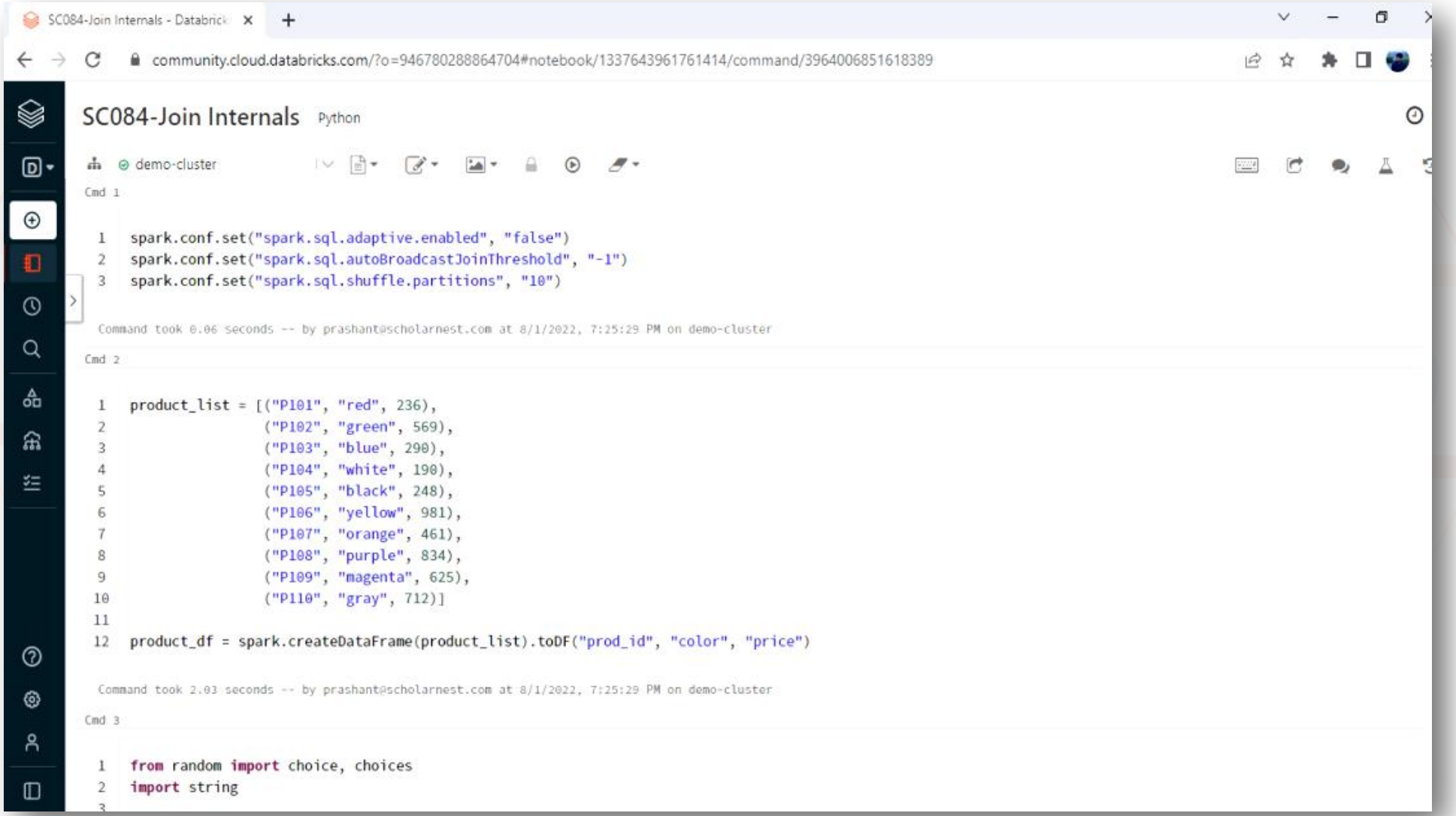
The data skew is one of the most critical things to handle. And how do we do it?
Well, we have some tricks to handle it. One most popular trick are known as data salting.
However, Spark 3.0 and above come with AQE optimization. The AQE helps you to handle the data skew problem. It will also determine the best value for *spark.sql.shuffle.partitions.*
So do not worry too much about the data skew and the best value for the shuffle partition number. AQE will take care of these problems, and we will learn AQE in the coming lectures.
But knowing the concept takes a long way to tuning your Join performance.

I talked about two scenarios for Dataframe joins in the beginning:

1. Large to Large – A large Dataframe to a large Dataframe join will always go for a shuffle join. We do not have any other option. And I gave you some tuning pointers for the shuffle join.
2. Large to Small – A large Dataframe join with a small Dataframe can take advantage of the broadcast join, which often works faster than the shuffle join.

Let's try to understand.

Let me go back to the data bricks workspace and open my notebook we used in the last lesson **(Reference: SC084-Join Internals)**



Scroll down, and you will see the product_df. This one is my first Dataframe, and it is super tiny. I have only 10 records in this Dataframe. So I know this is a small Dataframe.

SC084-Join Internals Python

demo-cluster

Command took 0.06 seconds -- by prashant@scholarnest.com at 8/1/2022, 7:25:29 PM on demo-cluster

Cmd 2

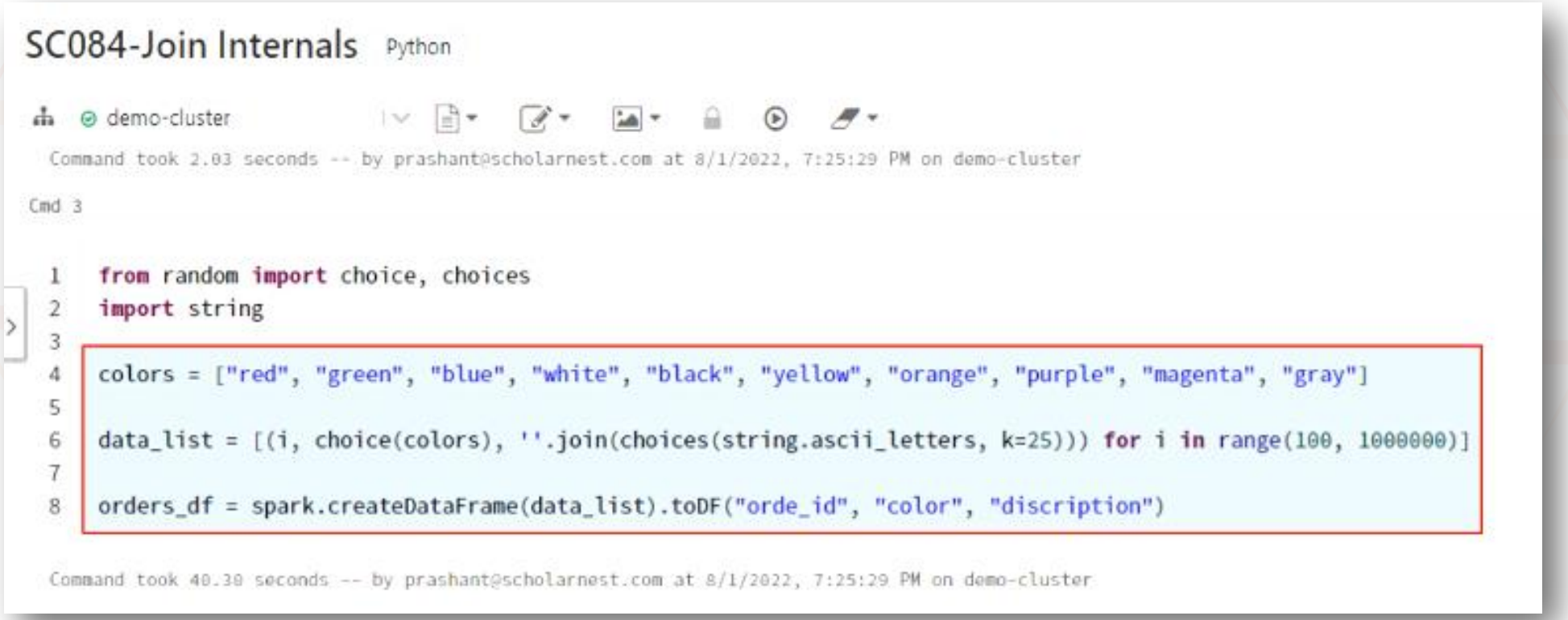
```
1  product_list = [("P101", "red", 236),
2                  ("P102", "green", 569),
3                  ("P103", "blue", 290),
4                  ("P104", "white", 190),
5                  ("P105", "black", 248),
6                  ("P106", "yellow", 981),
7                  ("P107", "orange", 461),
8                  ("P108", "purple", 834),
9                  ("P109", "magenta", 625),
10                 ("P110", "gray", 712)]
11
12 product_df = spark.createDataFrame(product_list).toDF("prod_id", "color", "price")
```

Command took 2.03 seconds -- by prashant@scholarnest.com at 8/1/2022, 7:25:29 PM on demo-cluster

Cmd 3

```
1  from random import choice, choices
2  import string
3
4  colors = ["red", "green", "blue", "white", "black", "yellow", "orange", "purple", "magenta", "gray"]
5
6  data_list = [(i, choice(colors), ''.join(choices(string.ascii_letters, k=25))) for i in range(100, 1000000)]
7
8  orders_df = spark.createDataFrame(data_list).toDF("orde_id", "color", "discription")
```


Scroll down, and you will see orders_df. This one is quite large. I have one million records in this Dataframe. So the orders_df is my large Dataframe. I am taking a super tiny example. But in a real sense, you may have one very large Dataframe. My order Dataframe could be 1 TB or even larger than that. And my product Dataframe could be 5 MB or 10 MB in size.



In the end, I have code to join these two large and small Dataframes and write the output as a table.

Cmd 4

```
1  result_df = orders_df.join(product_df, "color", "inner")
2  result_df.write.format("parquet").mode("overwrite").saveAsTable("sales_tbl")
```

▸ (1) Spark Jobs

Command took 33.21 seconds -- by prashant@scholarnest.com at 8/1/2022, 7:25:29 PM on demo-cluster

Cmd 5

For this example, I am setting *spark.SQL.shuffle.partitions* to 10.
I have a single node cluster and a single executor core, so my 10 partitions will join in a sequence.
But that's fine for the example.

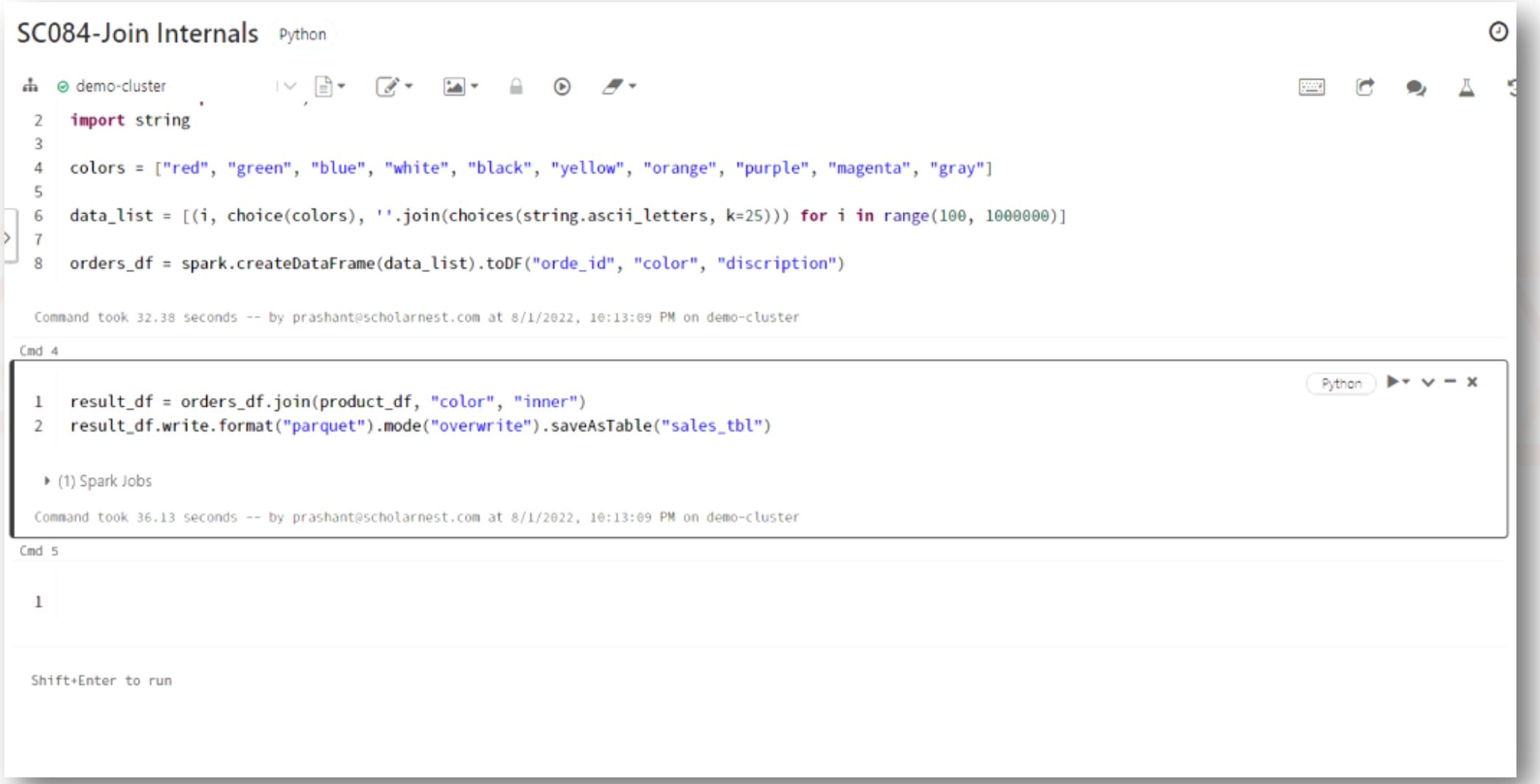
Cmd 1

```
1  spark.conf.set("spark.sql.adaptive.enabled", "false")
2  spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")
3  spark.conf.set("spark.sql.shuffle.partitions", "10")
```

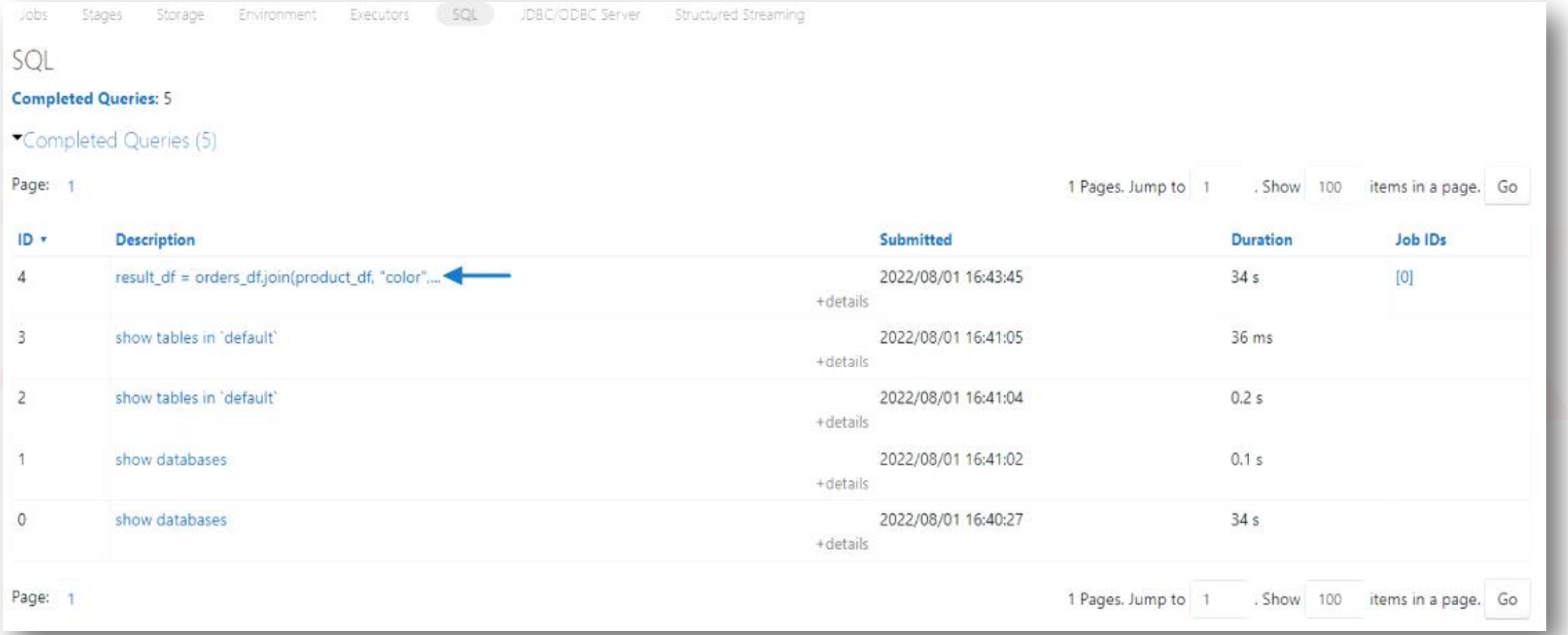
Command took 0.06 seconds -- by prashant@scholarnest.com at 8/1/2022, 7:25:29 PM on demo-cluster

I am disabling AQE and broadcast to see the default behaviour.
But we will come to those two settings in a minute.

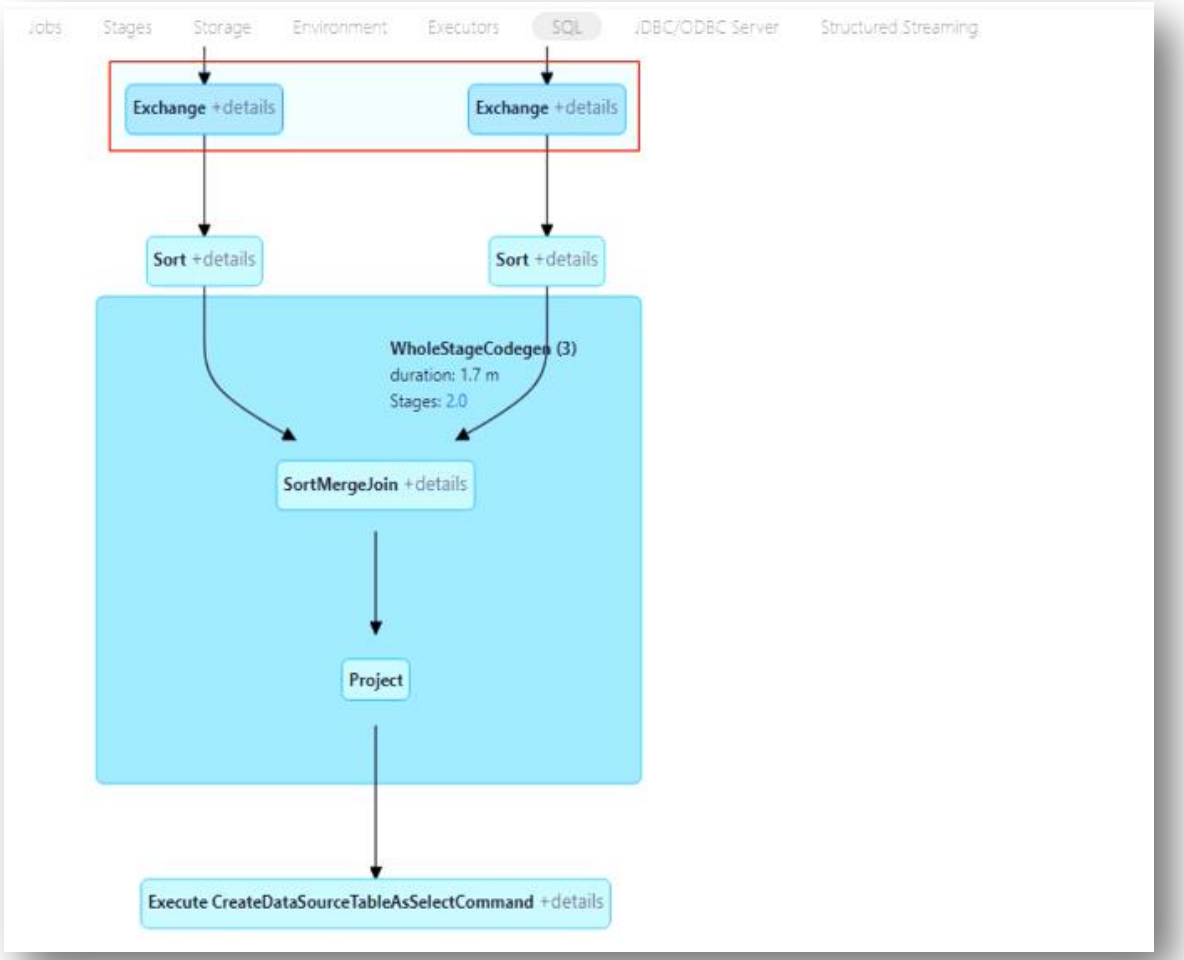
Now I ran my notebook.



Now go to the Spark UI and click the SQL tab and look for the execution plan for the last SQL.



Here is the execution plan. If you scroll down, and you will see the exchange and join type. It says sort-merge Join. The presence of the exchange and short merge join indicates that we are applying a shuffle join. Shuffle join will always show up as an exchange. It is not a Shuffle Join when you do not see the exchange. But if you see the exchange, it is applying a shuffle.

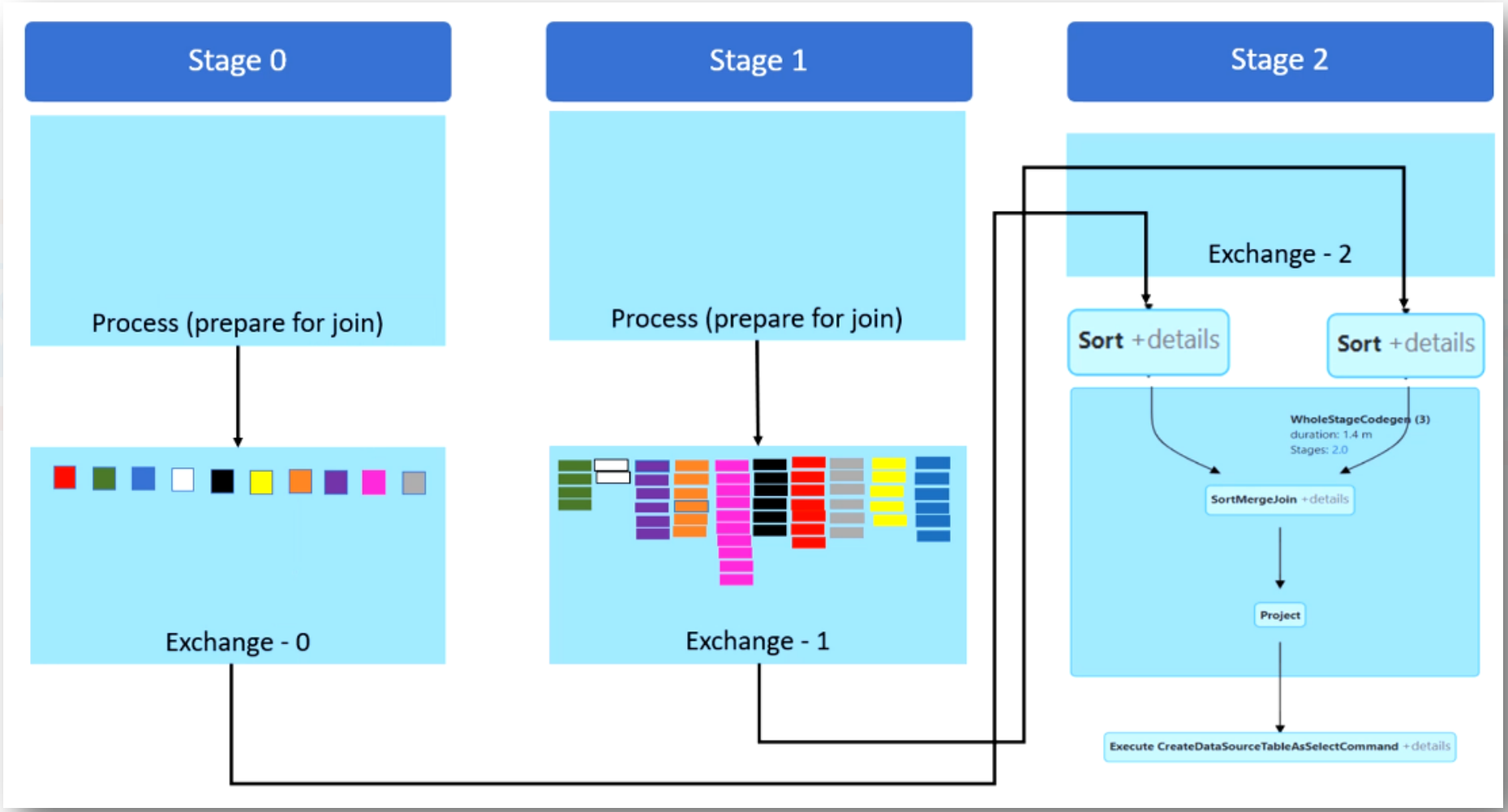


I am doing a small table to a large table join.
But Spark applied shuffle join.

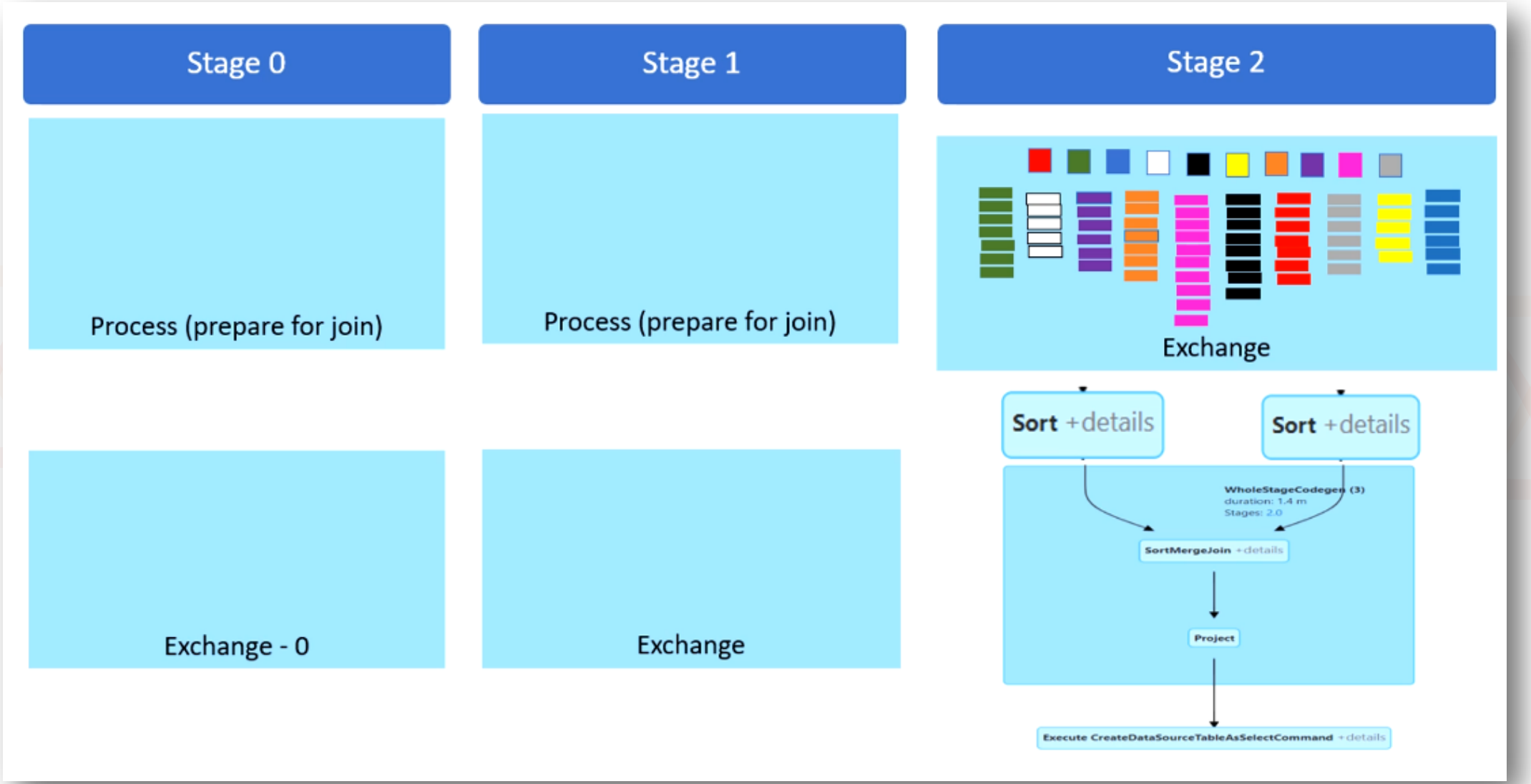
Shuffle join is a problem. Why? Because all the one million records from my orders_df will travel over the network.

But we have another much better alternative - broadcast join.
Let's try to understand it.

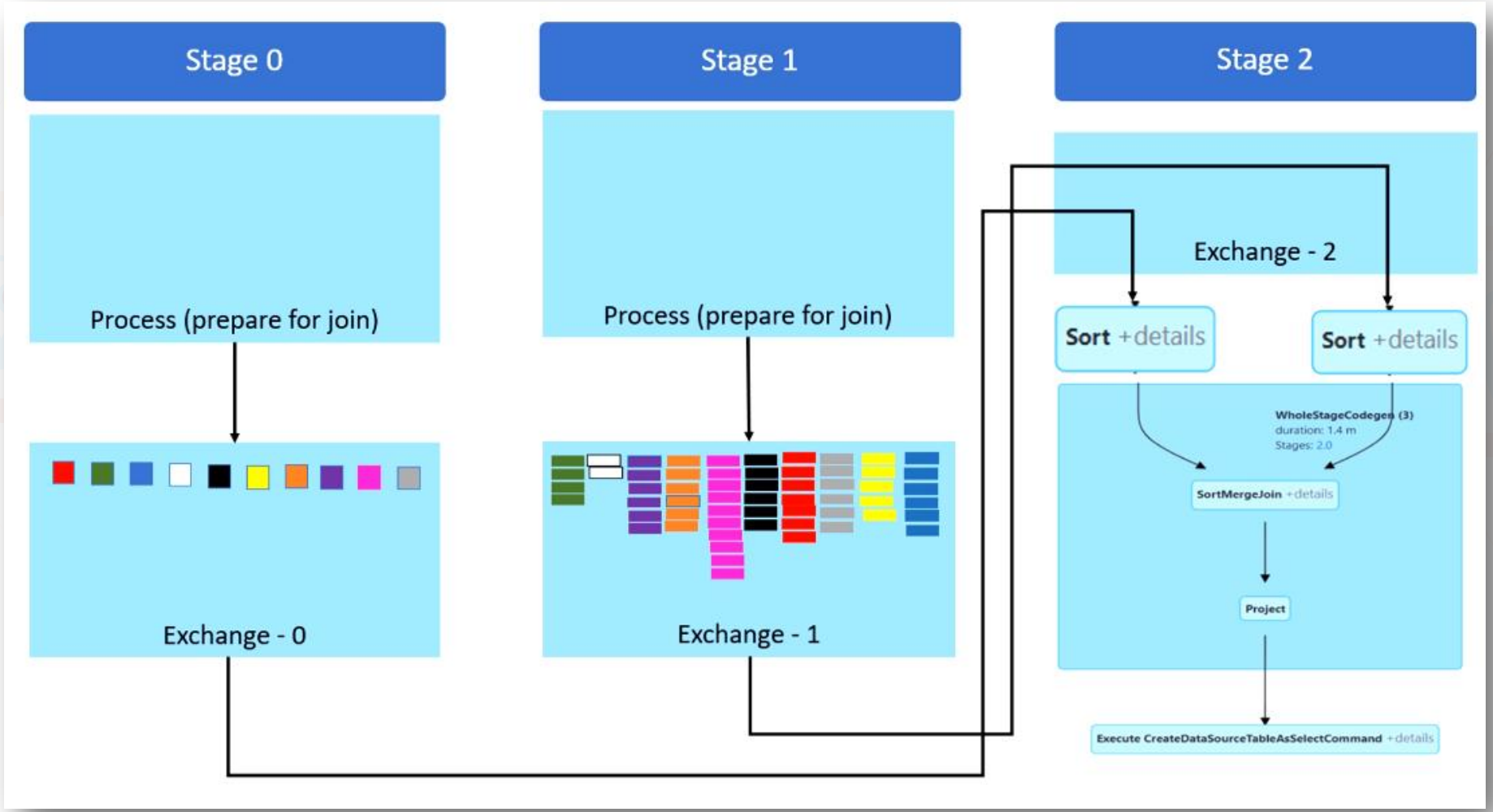
Here is how the execution plan looks for the shuffle join.
All the data from stage zero will go to stage two in this plan. And data from stage one will also go to stage two.



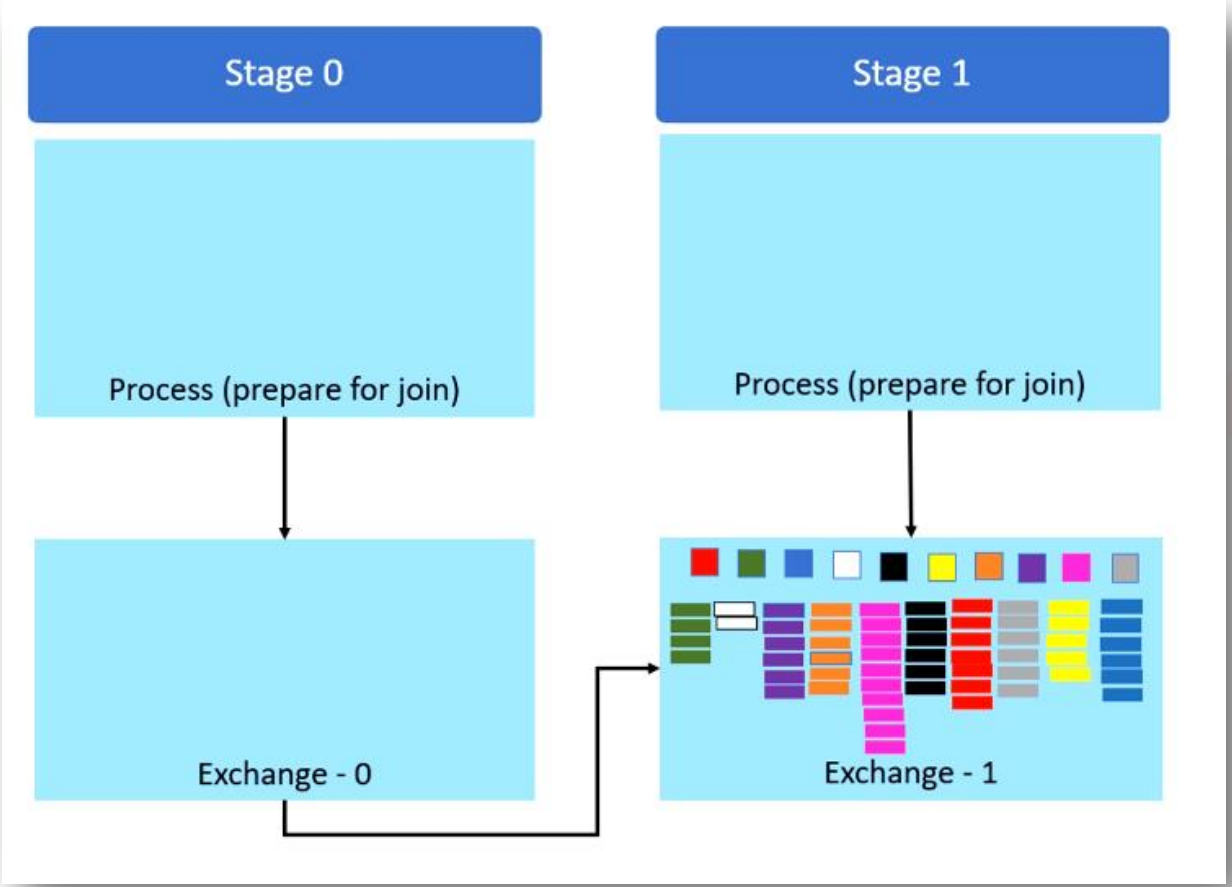
Here is how it will look after the shuffle join.



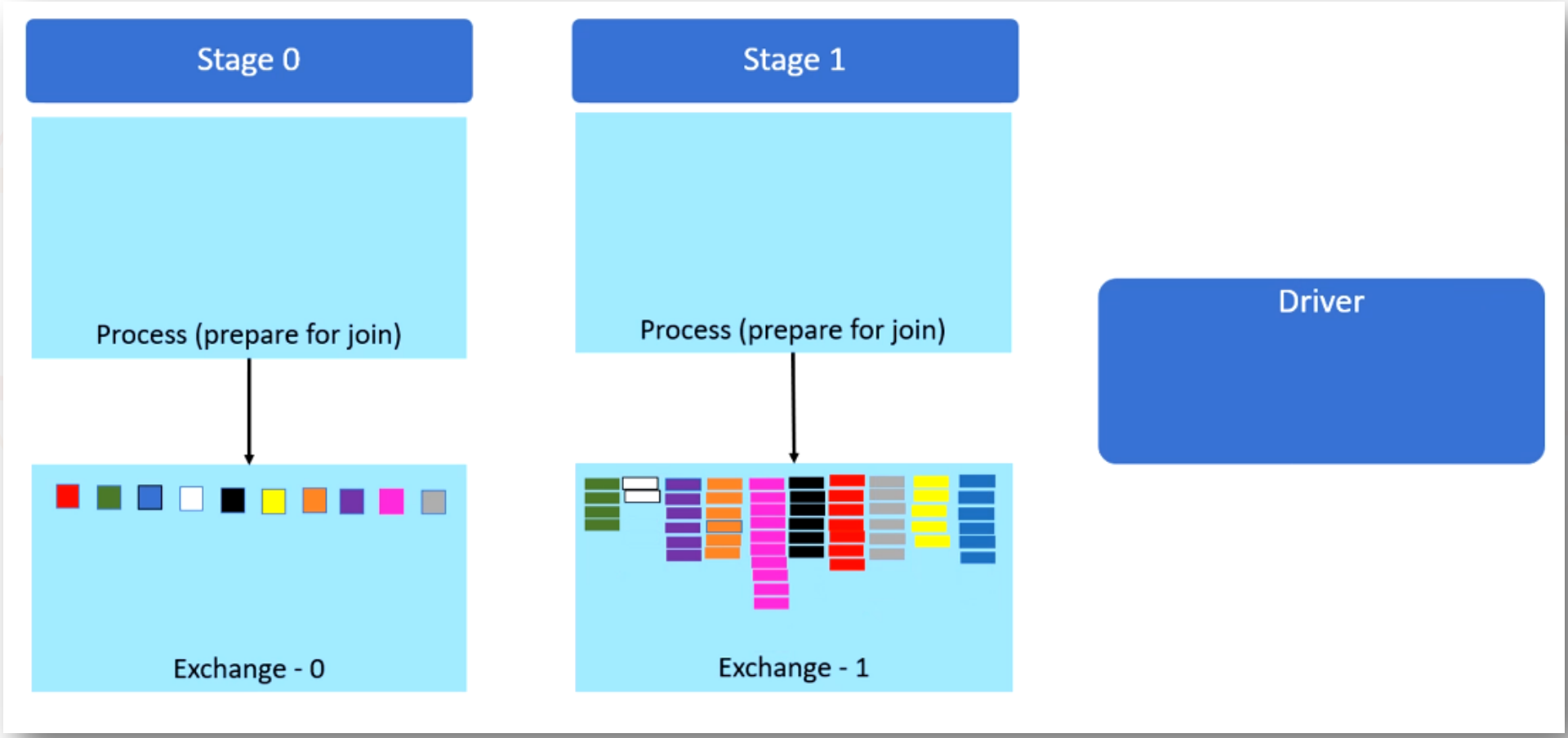
But we have another alternative. Here is our initial plan.
What if we send the data from stage zero to stage one? Stage one can perform the Join, and we do not need stage two.



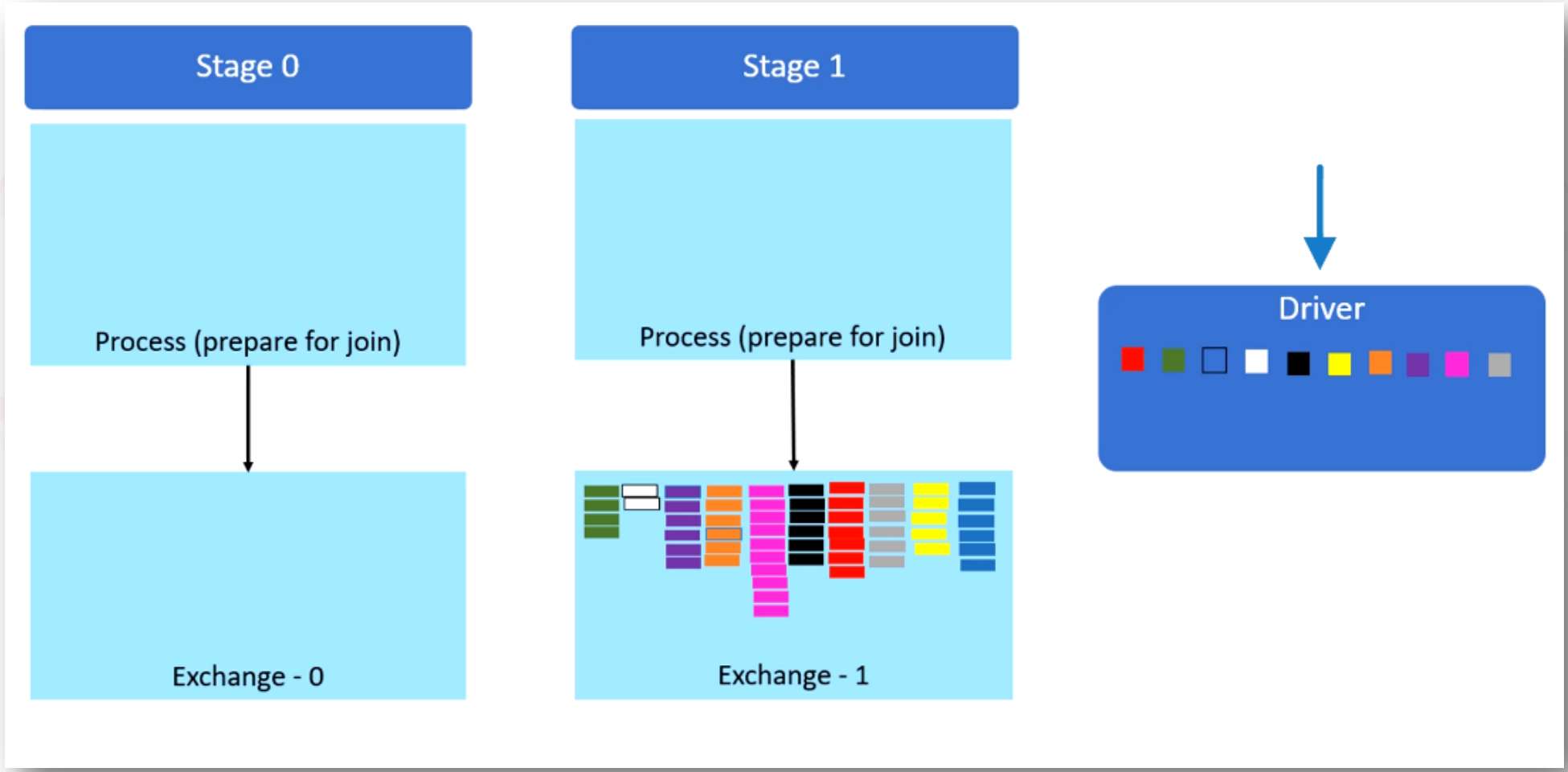
The plan looks like this shown below. Instead of sending both data sets to the third stage, we can send smaller data to the larger data. And tell the stage one executors to do the Join. And this is magically faster. Why? Because we are not sending large volumes of data across exchanges. We are sending the smaller Dataframe to the larger Dataframe. And this approach is called the broadcast join approach. This approach sends a small table to all the executors performing stage one tasks.



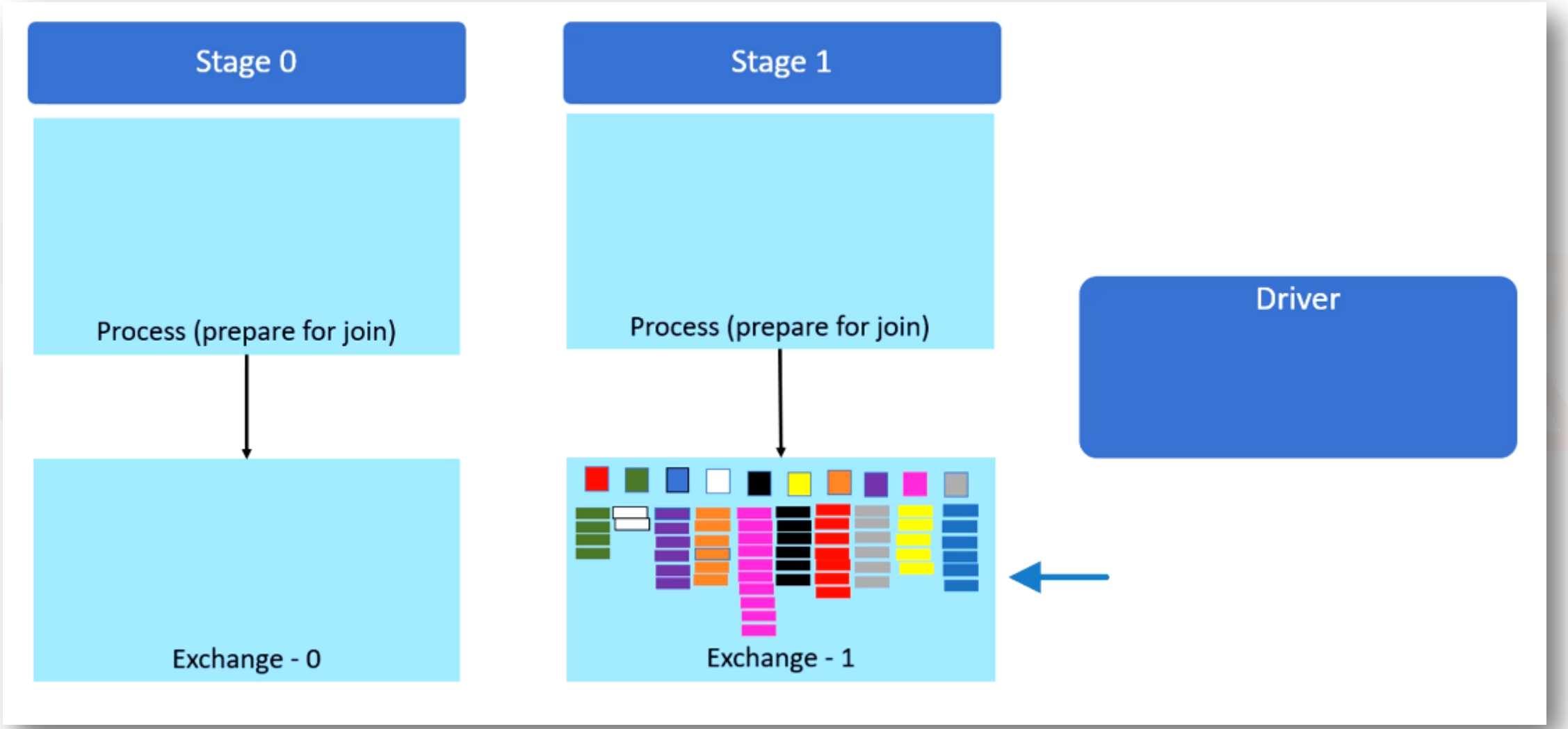
Let’s see how this happens from the beginning.
Spark will create a two-stage execution plan. The shuffle join will create three stage plan.
But the broadcast join will create a two-stage plan.



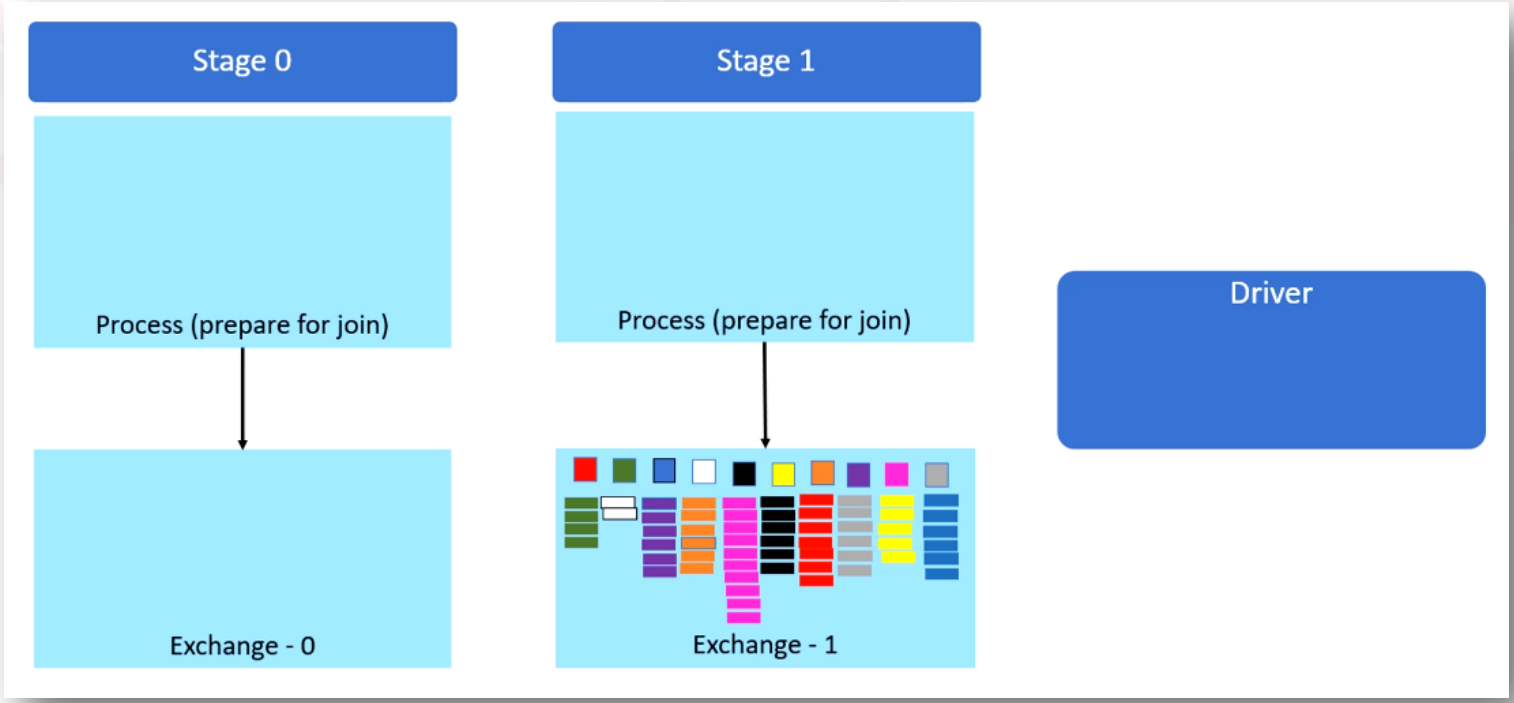
In this case, the stage zero data will go to the driver.
Why does it go to the driver? Because stage zero doesn't know where to send it. It doesn't know where stage one is being executed. The driver knows everything.



So the stage zero data will go to the driver, and then the driver will broadcast it back to all the stage one executors. Then the stage one executors will perform the Join.



This approach is known as broadcast join. Because the driver is broadcasting the smaller table to all the executors. The broadcast join is an easy method to eliminate the need for a shuffle. You can perform a join without a shuffle. However, this approach works when one of your Dataframe is small. When I say small, it doesn't mean a few MBs. It should be small enough to fit in the driver's memory. And it should be small enough to fit into the stage one executor memory. The stage one executors will also have data from stage one. And the stage two data will also come to them as an extra overhead. So the broadcast data should be small. Otherwise, the broadcast approach will throw an OOM error.



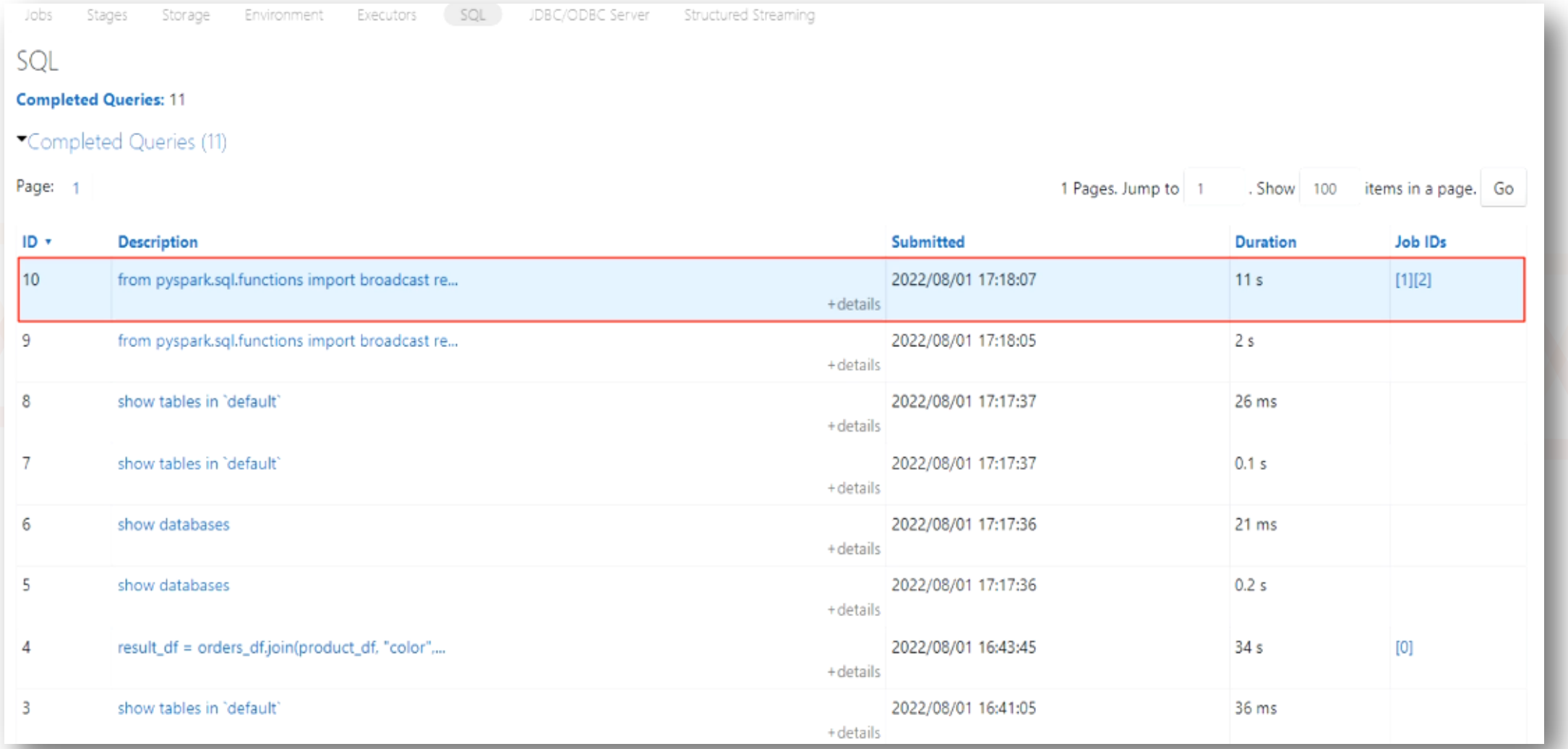
Now the next question is this:

How to implement broadcast join?

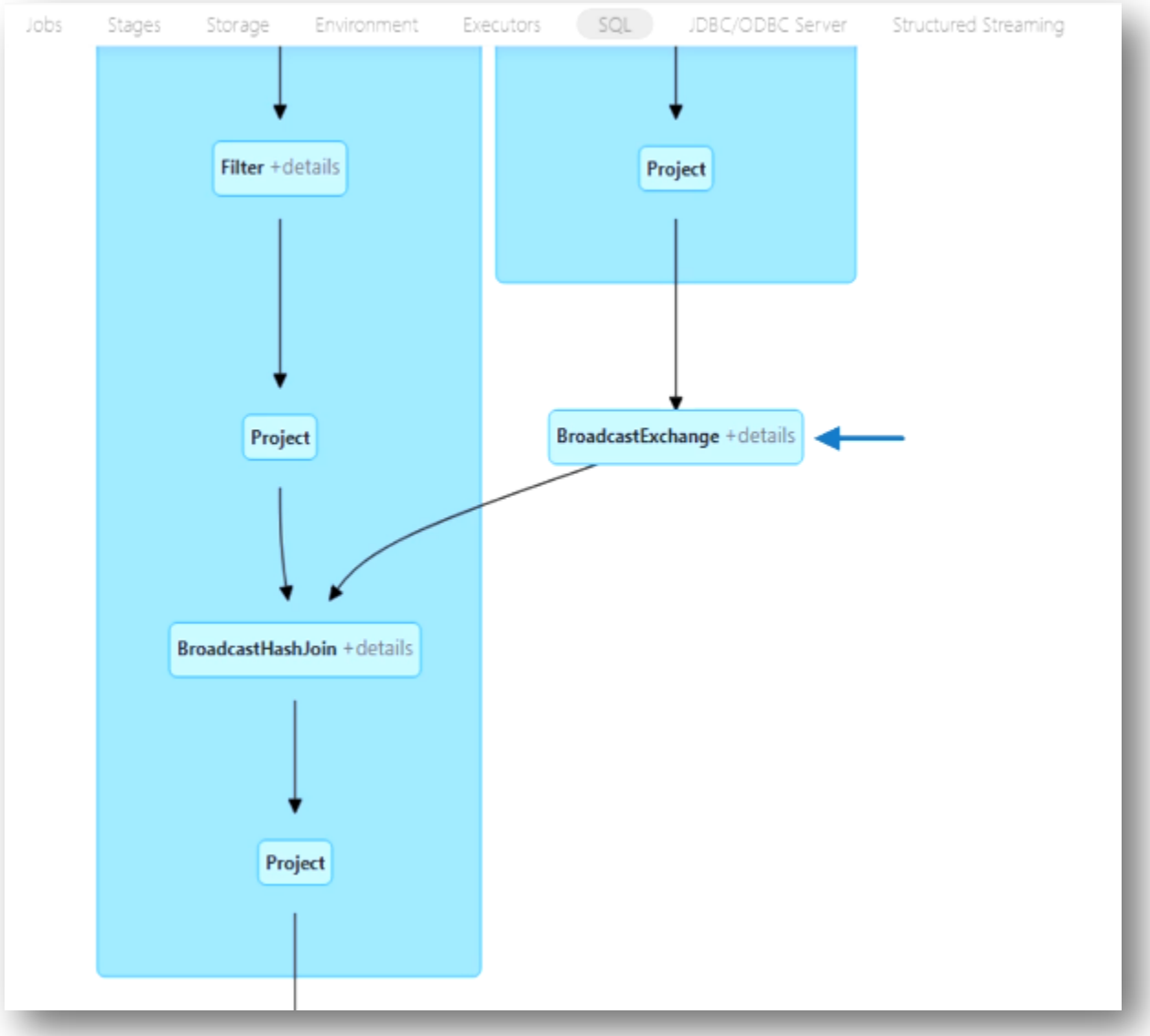
In most cases, Spark will automatically use the broadcast join when one of your Dataframe is smaller.
However, you know your Dataframes a lot better than Spark.
So you should apply a hint for using the broadcast join.
Let us see this in action.

Go to the join expression and apply the broadcast function as shown below, you should also import the broadcast function.
I am hinting to Spark to apply the broadcast join and avoid shuffle join.
Remember that the broadcast is a hint. It simply tells spark that broadcast join is applicable. A spark might still ignore your hint. But it is most likely to apply the hint.
Let me rerun the notebook and go to Spark UI to check the SQL tab.

Look for the execution plan for the last query.



Here is the DAG. Do you see this broadcast exchange? We have only two stages. And Spark is sending data from one stage to another stage. We do not see the shuffle join in the plan now.



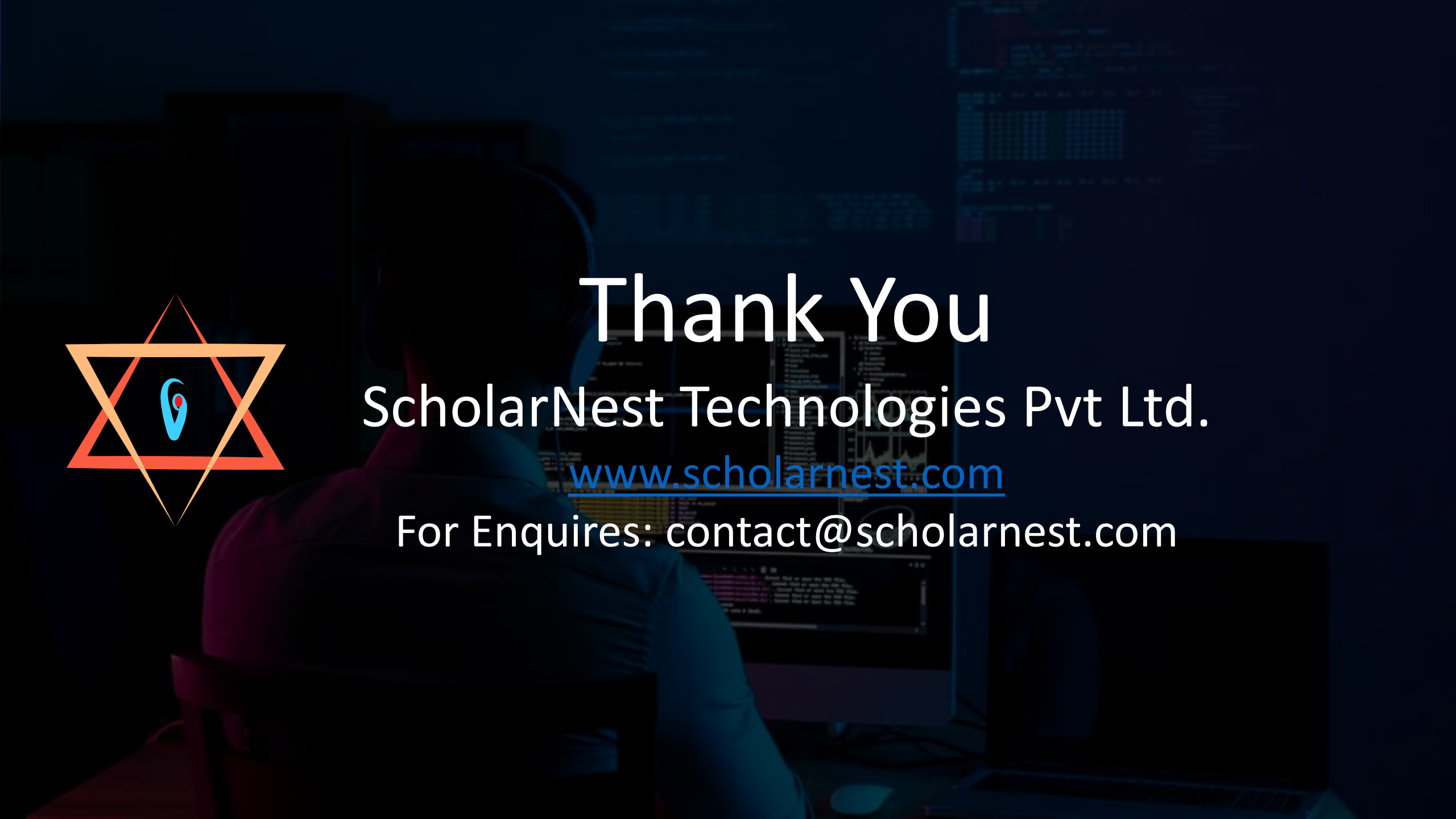
Now come back to the notebook and look at the first cell.
The spark comes with one broadcast configuration.
I am setting *spark.SQL.autoBroadcastJoinThreshold* to -1. The default value of this configuration is 10MB. And Spark will automatically apply the broadcast join if one Dataframe in a join is smaller than the 10MB threshold. You can disable broadcast join, setting this value to -1. I disabled it so I can show you both approaches. But you will never disable it.

So, Spark automatically applies broadcast join when your database is smaller than 10 MB.
If Spark is not doing it, you can apply the broadcast hint.

You can also increase the *spark.sql.autoBroadcastJoinThreshold.*
The default value is 10 MB, but you can increase it to 25MB or 200MB.
But make sure you have enough memory to hold the broadcast table at the driver.

I talked about shuffle join and three tuning considerations:

1. **Reduce your Dataframe size for best performance** – You can apply filters or do aggregations before the Join, so you have a smaller Dataframe. Make sure your shuffle partitions are in the sweet spot of 128 MB to 1GB and set the *spark.SQL.shuffle.partitions* accordingly. The best performance comes with approximately 128 MB shuffle partitions. So determining a good value of *spark.SQL.shuffle.partitions* is critical.
2. **Take enough executors to do things in parallel** – You also need to apply some approaches to avoid data skew. Data salting is one popular technique to break the skew. However, Spark 3.0 and above comes with AQE to take care of the data skew and determine the best value for *spark.SQL.shuffle.partitions.* So if you are using Spark 3.0 or above, you do not worry much about the data skew and setting the *spark.SQL.shuffle.partitions.*
3. **Broadcast join approach** – Spark will automatically apply broadcast join when one of your tables is smaller than 10 MB. However, you can change *spark.SQL.autoBroadcastJoinThreshold* to larger than 10 MB. It is a best practice to apply broadcast hints when you know you have a small table in the join. I showed you how to apply the broadcast hint.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Joins and Optimization

Lecture:
Bucket Joins

Implementing Bucket Joins

We learned many things in the earlier lectures.
However, I want to recall the following two topics.

1. DataFrameWriter API
2. Spark Shuffle Joins

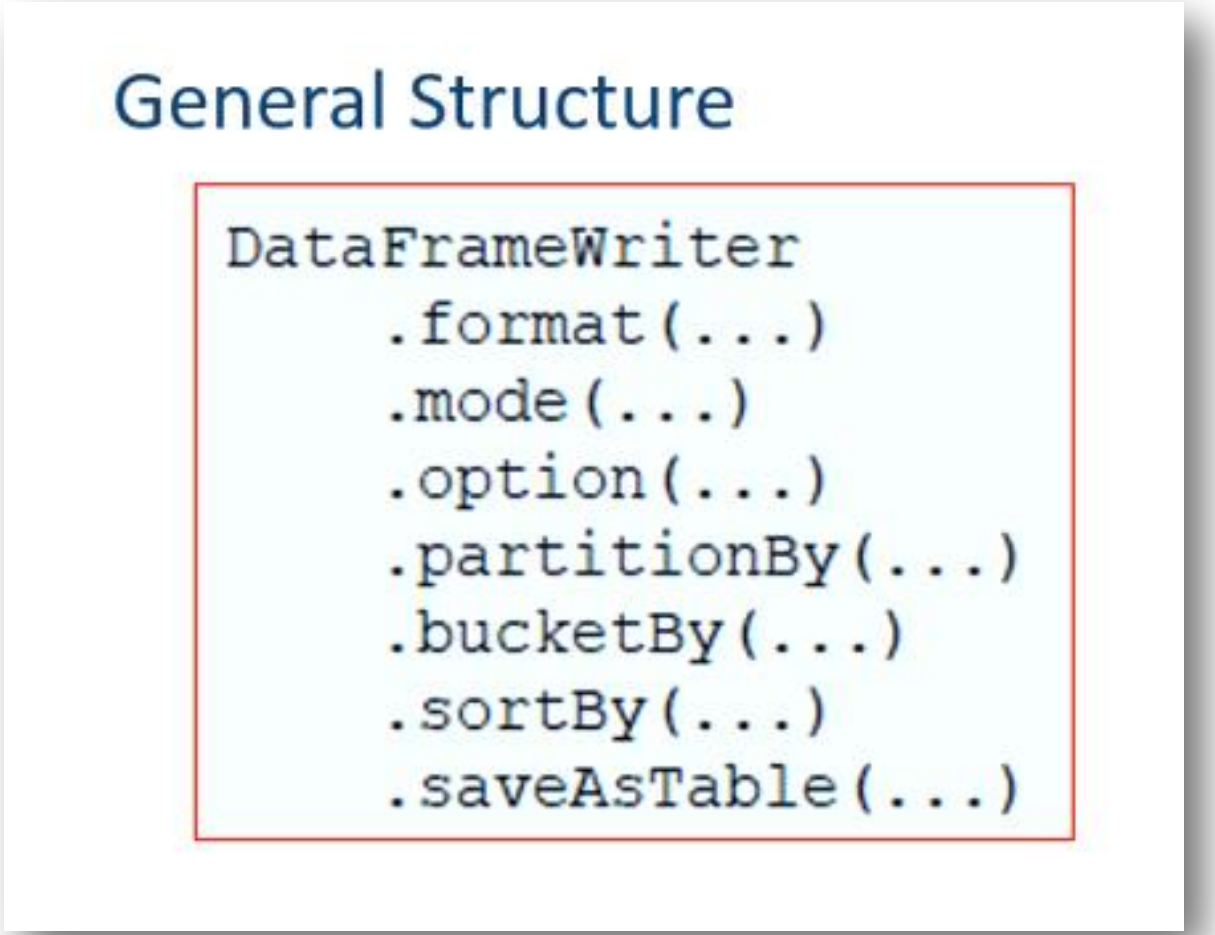
Let me quickly remind you of the DataFrameWriter API.
We use Spark DataFrameWriter API to write our Dataframe to the storage.

Here is an example of writing your Dataframe to the storage.
I am starting with the Dataframe dot write method, which gives me a DataFrameWriter.
Then I apply some DataFrameWriter methods such as format, mode, option, partitionBy, and finally save it as a table.

Indicative Example

```
df.write
  .format("parquet)
  .mode("overwrite)
  .option("compression", "snappy")
  .partitionBy("country", "month")
  .saveAsTable("my_tbl")
```


Here is the general structure of the DataFrameWriter API.
We already used format, mode, option, and partitionBy methods in the example.
The partitionBy() method is super helpful, and we also learned about it. We use partitionBy() to design a layout of our data. But why do we do that? We do it to gain performance benefits.



In this example, I am partitioning my table by country and month.
So when I query this table adding country or month in the where clause, Spark will apply partition pruning and run faster.

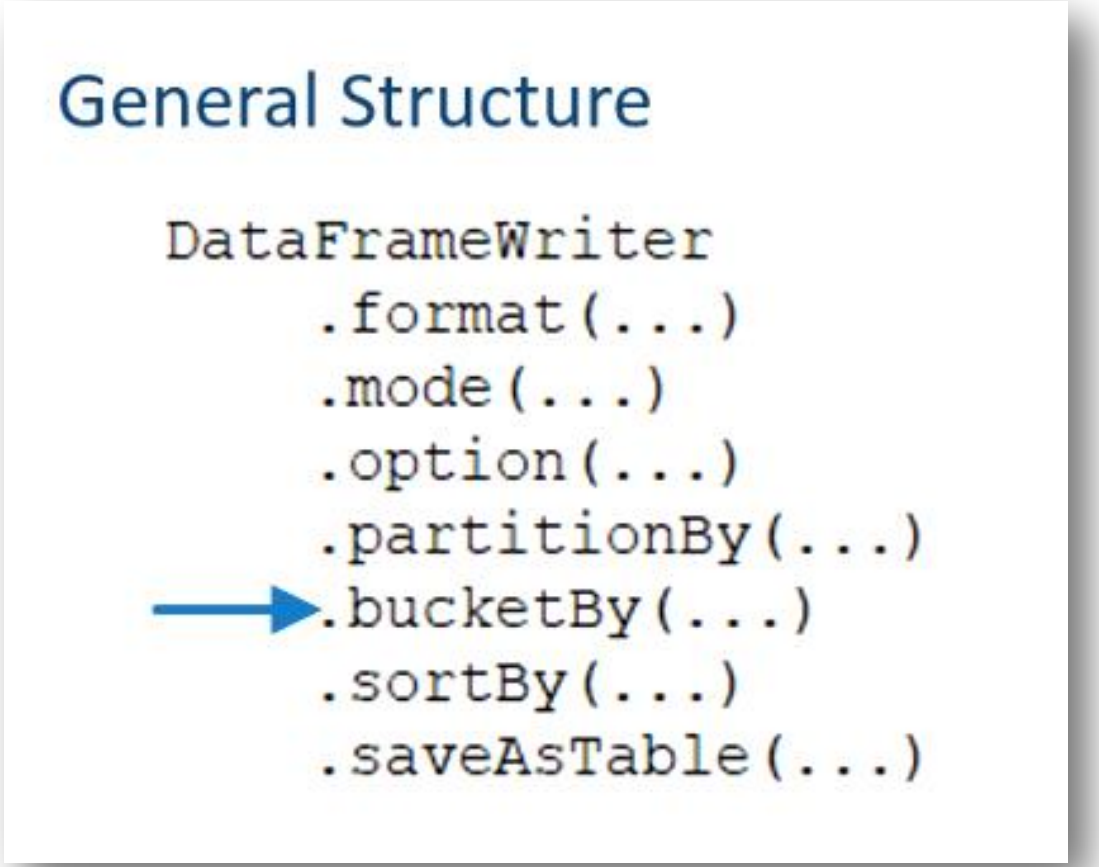
Indicative Example

```
df.write
  .format("parquet)
  .mode("overwrite)
  .option("compression", "snappy")
  .partitionBy("country", "month")
  .saveAsTable("my_tbl")
```

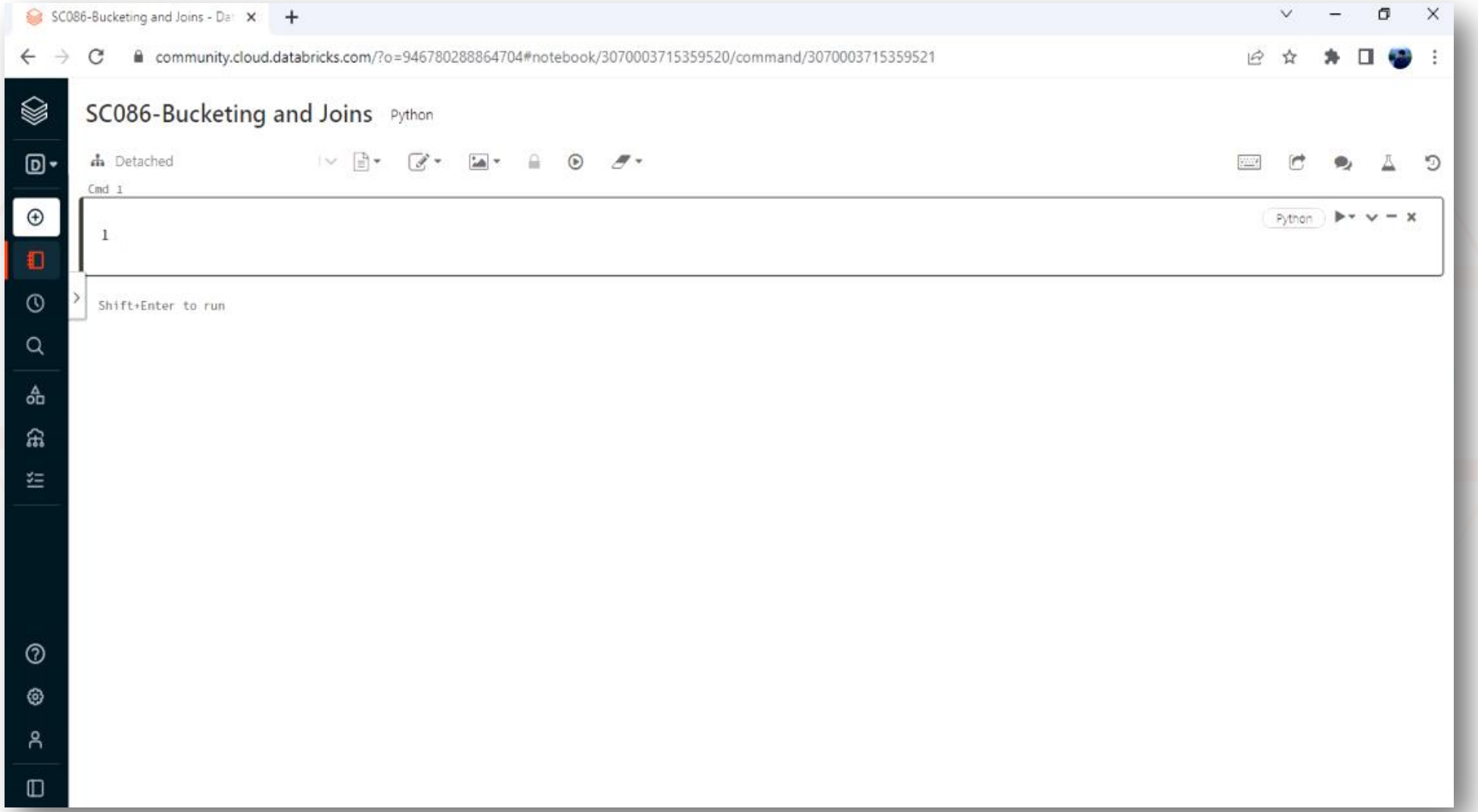
General Structure

```
DataFrameWriter
     .format(...)
     .mode(...)
     .option(...)
     .partitionBy(...)
     .bucketBy(...)
     .sortBy(...)
     .saveAsTable(...)
```


The DataFrameWriter also gives us bucketBy().
What is that?
It is similar to the partitionBy(), but it is different.
Let's try to understand it using an example.

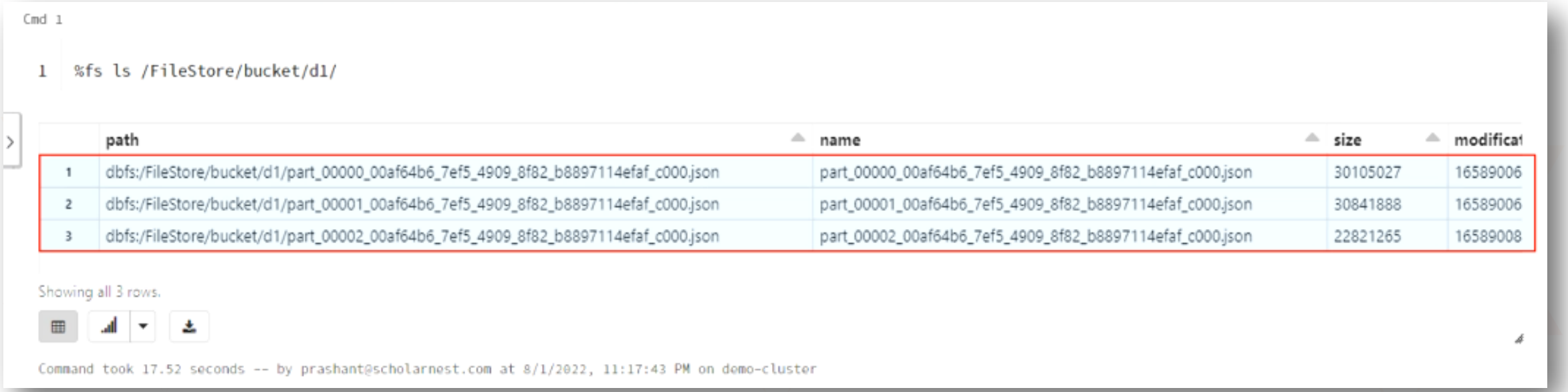


Let me go to the Databricks workspace and create a new notebook
(Reference: SC086-Bucketing and Joins)



I have some data that I uploaded to the DBFS, and you can see the 3 data files highlighted below.

(Reference: /data/bucket/d1)



Here I have the code to load those 3 files into a Dataframe.
I am loading the data and formatting the FL_DATE column to ensure it is a date column.

Cmd 2

```
1  from pyspark.sql.functions import to_date
2
3  df1 = spark.read \
4          .format("json") \
5          .load("/FileStore/bucket/d1/") \
6          .withColumn("FL_DATE", to_date("FL_DATE", "M/d/y"))
```

▸ (1) Spark Jobs

Command took 17.63 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:20:09 PM on demo-cluster

Then I wrote a select expression on the Dataframe where I am reading all columns where FL_DATE is 1st Jan 2000.

Cmd 3

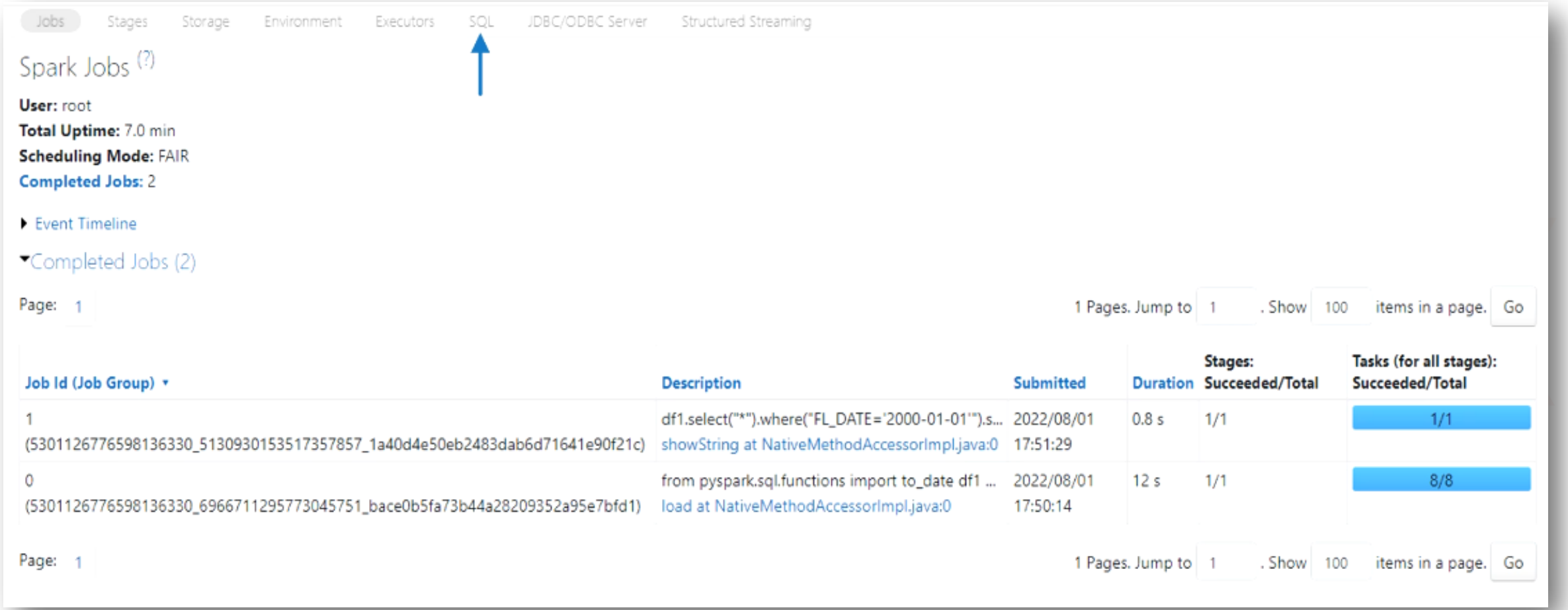
```
1  df1.select("*").where("FL_DATE='2000-01-01'").show()
```

▸ (1) Spark Jobs

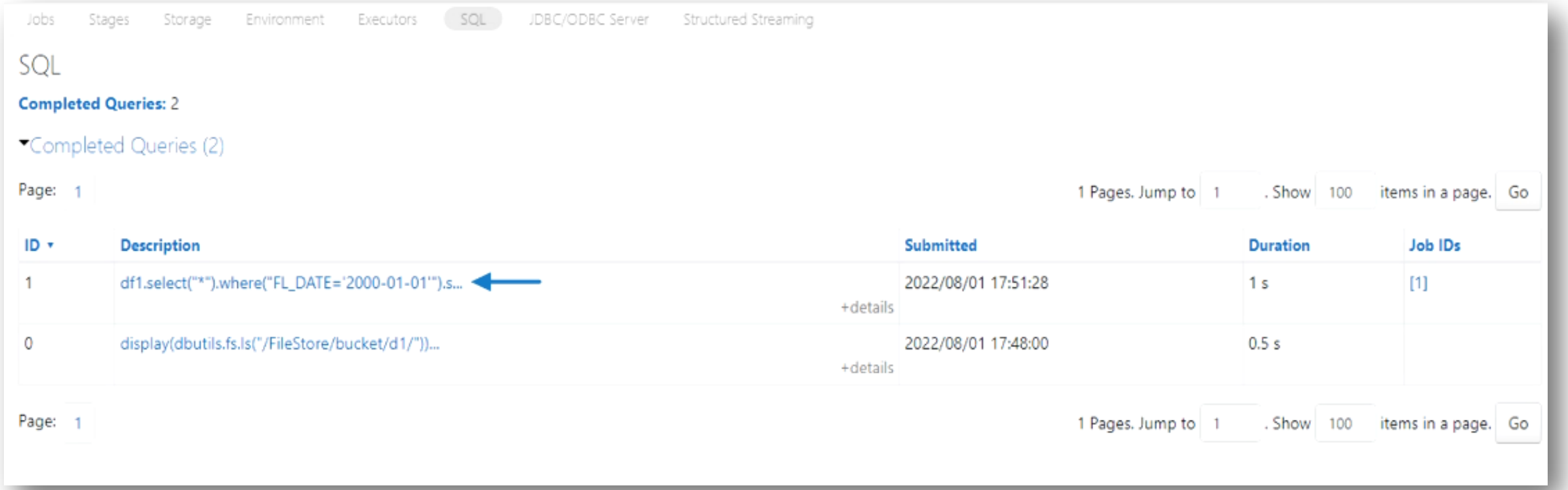
DEST	DEST_CITY_NAME	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	id
ATL	Atlanta, GA	2000-01-01	DL	1451	BOS	Boston, MA	0
ATL	Atlanta, GA	2000-01-01	DL	1479	BOS	Boston, MA	1
ATL	Atlanta, GA	2000-01-01	DL	1857	BOS	Boston, MA	2
ATL	Atlanta, GA	2000-01-01	DL	1997	BOS	Boston, MA	3
ATL	Atlanta, GA	2000-01-01	DL	2065	BOS	Boston, MA	4
ATL	Atlanta, GA	2000-01-01	US	2619	BOS	Boston, MA	5
ATL	Atlanta, GA	2000-01-01	US	2621	BOS	Boston, MA	6
ATL	Atlanta, GA	2000-01-01	DL	346	BTR	Baton Rouge, LA	7
ATL	Atlanta, GA	2000-01-01	DL	412	BTR	Baton Rouge, LA	8
ATL	Atlanta, GA	2000-01-01	DL	299	BUF	Buffalo, NY	9
ATL	Atlanta, GA	2000-01-01	DL	495	BUF	Buffalo, NY	10
ATL	Atlanta, GA	2000-01-01	DL	677	BUF	Buffalo, NY	11
ATL	Atlanta, GA	2000-01-01	DL	251	BWI	Baltimore, MD	12
ATL	Atlanta, GA	2000-01-01	DL	1003	BWI	Baltimore, MD	13
ATL	Atlanta, GA	2000-01-01	DL	1501	BWI	Baltimore, MD	14
ATL	Atlanta, GA	2000-01-01	DL	1907	BWI	Baltimore, MD	15
ATL	Atlanta, GA	2000-01-01	DL	2063	BWI	Baltimore, MD	16
ATL	Atlanta, GA	2000-01-01	DL	2111	BWI	Baltimore, MD	17

Command took 1.98 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:21:28 PM on demo-cluster

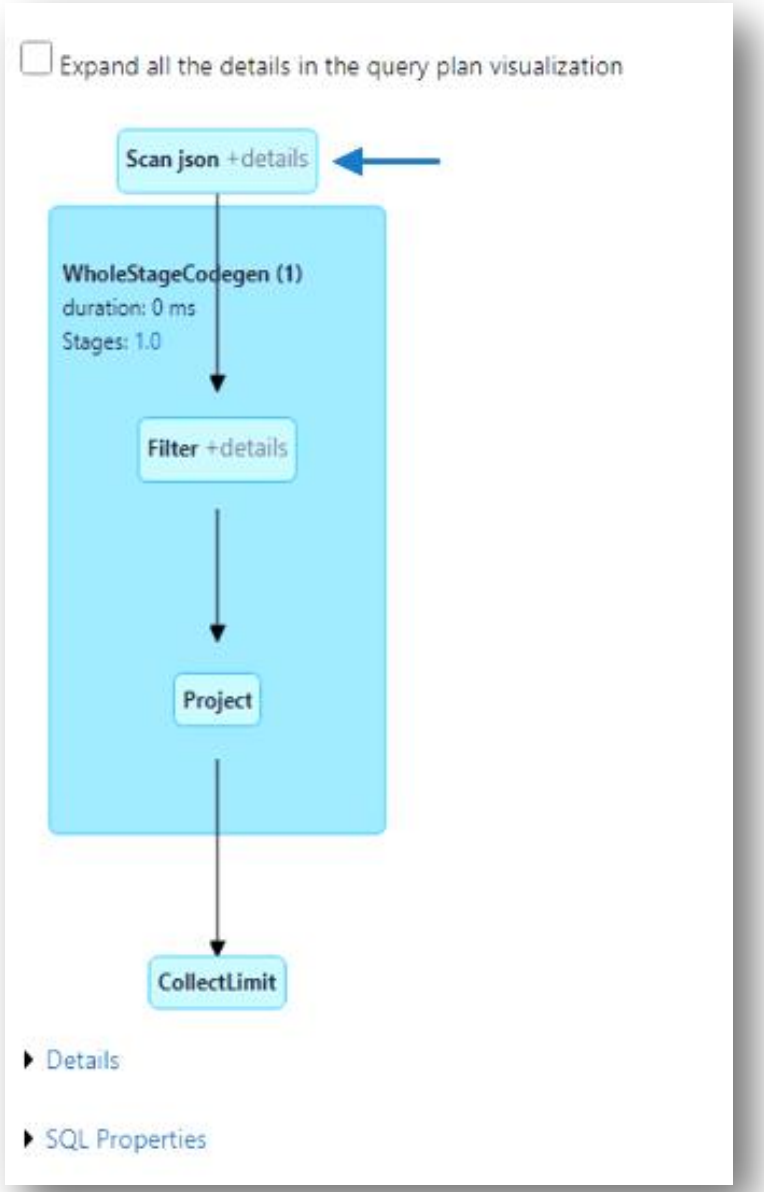
Go to the Spark UI, then the SQL tab.



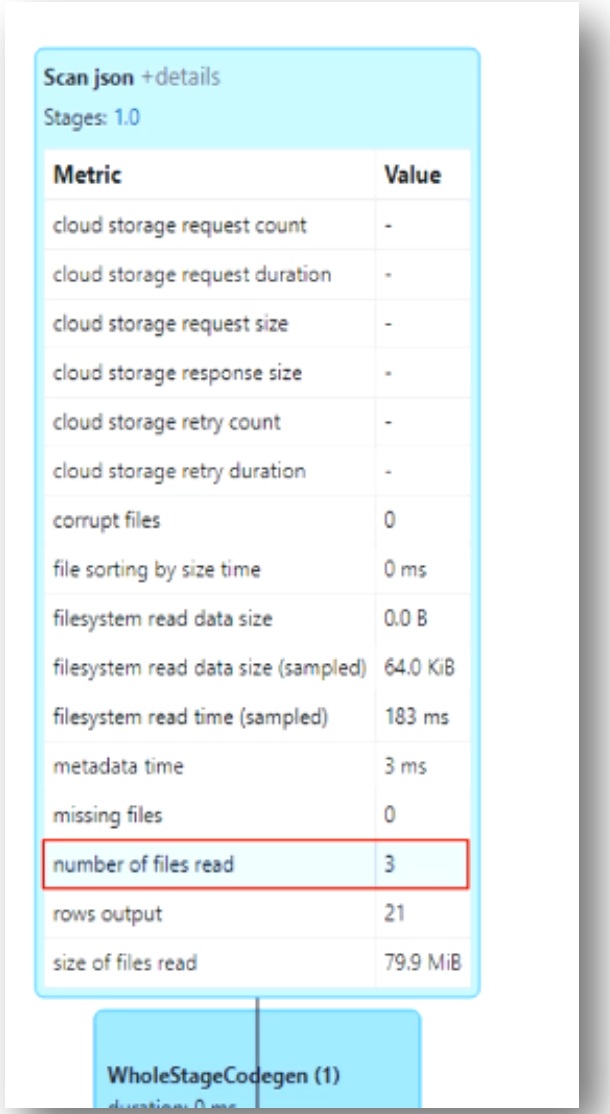
Click the latest query to see the execution plan.



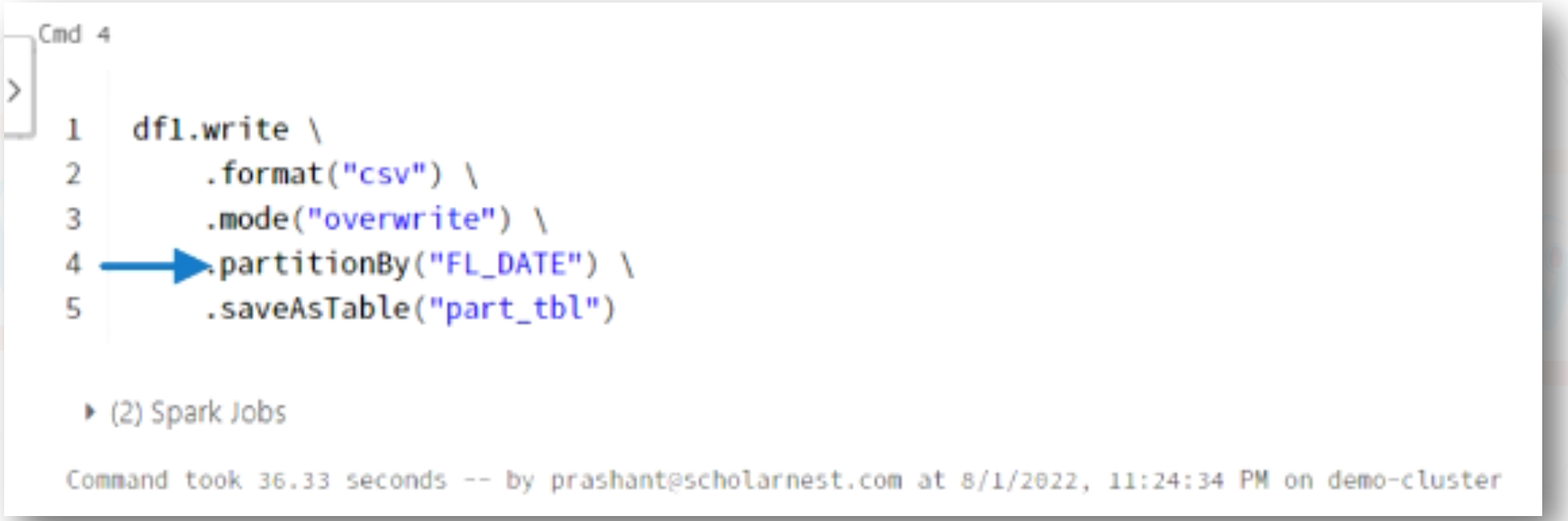
Here is the execution plan. The first step is to scan the JSON. Click the details.



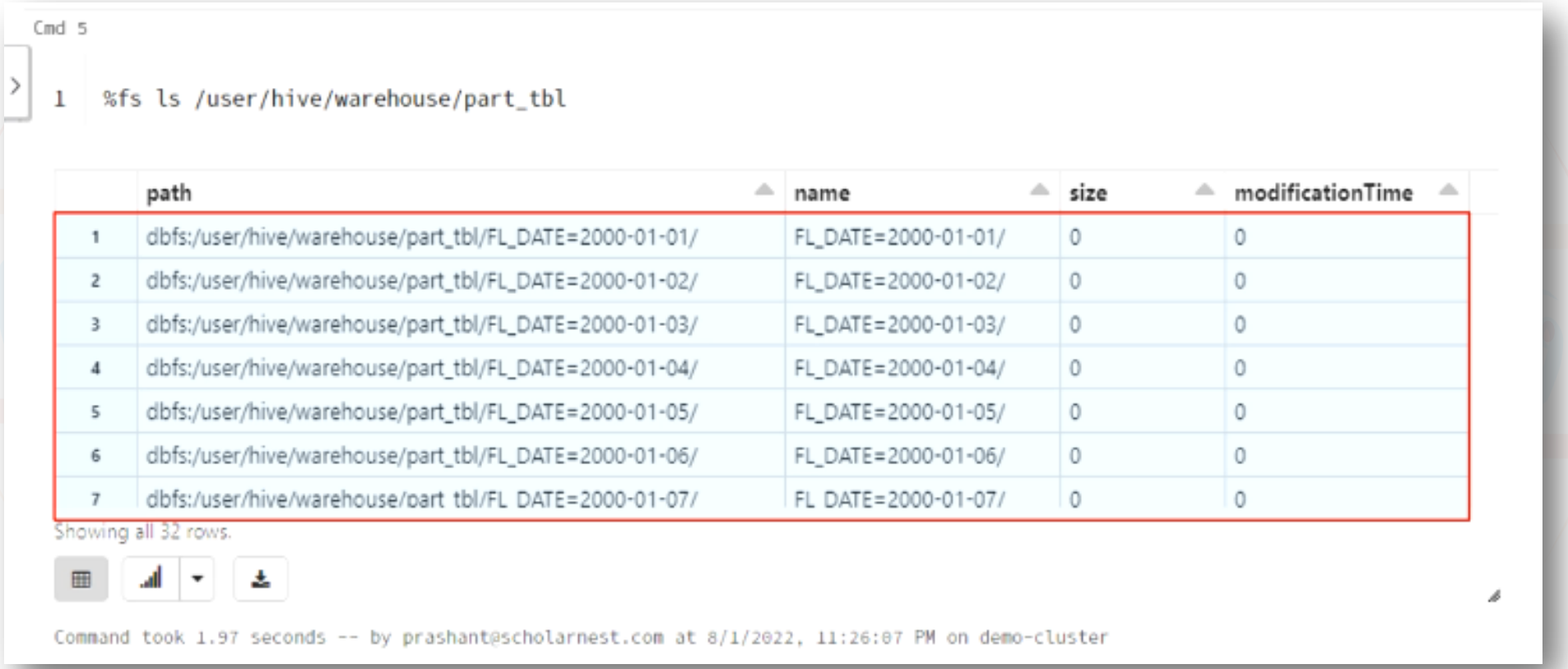
How many files do we read? Three files.
How many files do I have? Three files. Correct? So we are reading all three files.



Now go back to the notebook and create a partitioned table.
So I saved the same Dataframe to create a partitioned table on the FL_DATE column.
What will happen? Spark will create partition directories for each FL_DATE.



Here are the partitioned files.
You can see one directory for each date. And that's how partitionBy() works.


Cmd 5

```
1  %fs ls /user/hive/warehouse/part_tbl
```

	path	name	size	modificationTime
1	dbfs:/user/hive/warehouse/part_tbl/FL_DATE=2000-01-01/	FL_DATE=2000-01-01/	0	0
2	dbfs:/user/hive/warehouse/part_tbl/FL_DATE=2000-01-02/	FL_DATE=2000-01-02/	0	0
3	dbfs:/user/hive/warehouse/part_tbl/FL_DATE=2000-01-03/	FL_DATE=2000-01-03/	0	0
4	dbfs:/user/hive/warehouse/part_tbl/FL_DATE=2000-01-04/	FL_DATE=2000-01-04/	0	0
5	dbfs:/user/hive/warehouse/part_tbl/FL_DATE=2000-01-05/	FL_DATE=2000-01-05/	0	0
6	dbfs:/user/hive/warehouse/part_tbl/FL_DATE=2000-01-06/	FL_DATE=2000-01-06/	0	0
7	dbfs:/user/hive/warehouse/part_tbl/FL_DATE=2000-01-07/	FL_DATE=2000-01-07/	0	0

Showing all 32 rows.

Command took 1.97 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:26:07 PM on demo-cluster

I am running a query on the partitioned table. This is the same query that I executed on the Dataframe. The Dataframe was not partitioned, but now I am running this on the partitioned table. Go to the Spark UI and check the plan.

Cmd 6

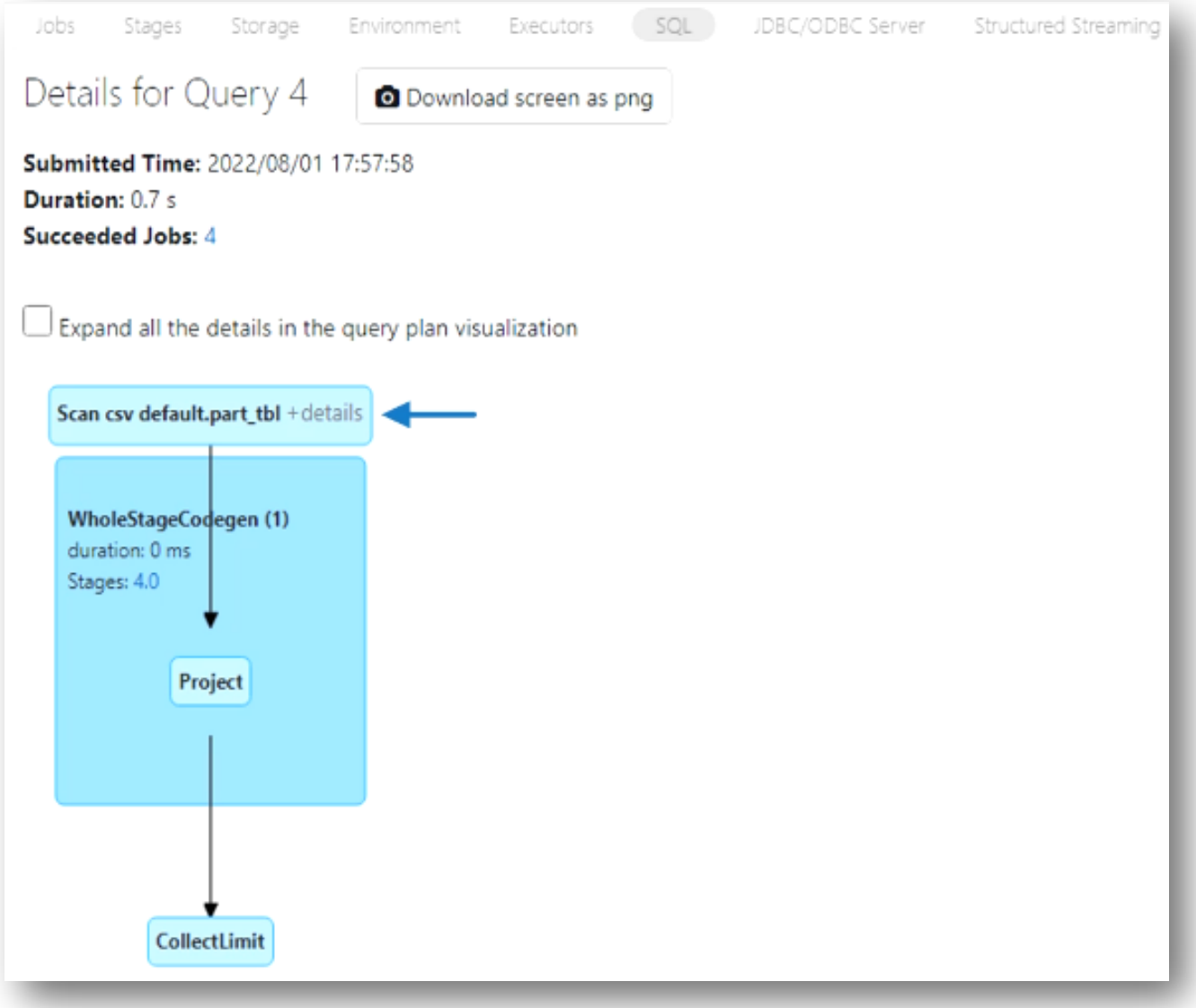
```
1  spark.sql("select * from part_tbl where FL_DATE = '2000-01-01'").show()
```

▸ (1) Spark Jobs

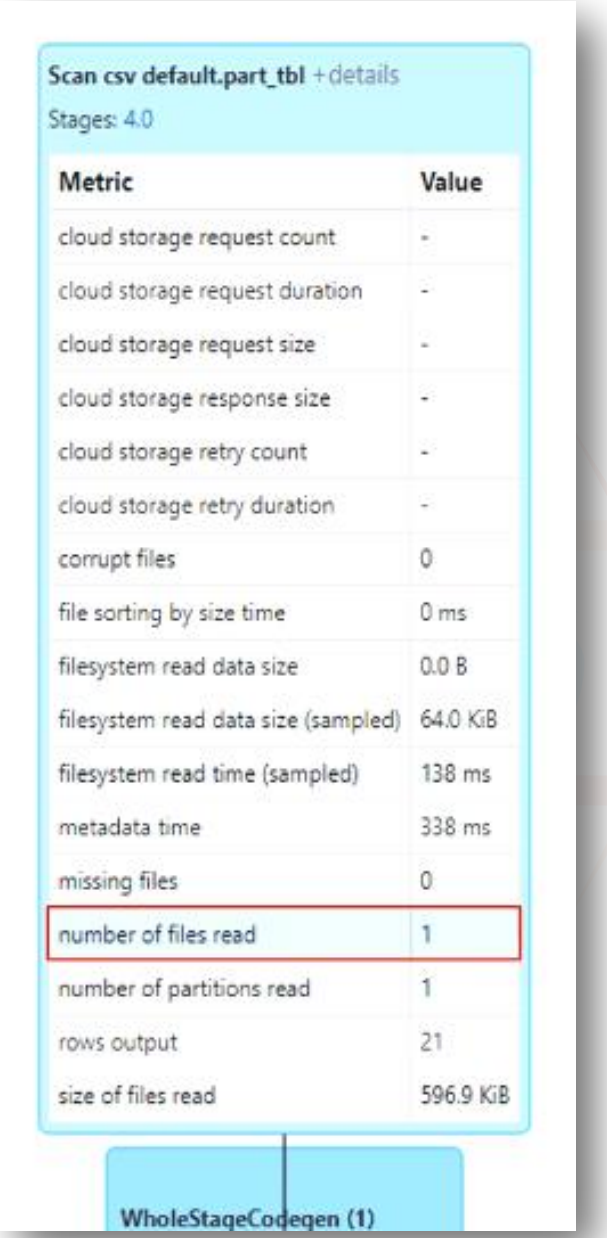
DEST	DEST_CITY_NAME	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	id	FL_DATE
ATL	Atlanta, GA	DL	1451	BOS	Boston, MA	0	2000-01-01
ATL	Atlanta, GA	DL	1479	BOS	Boston, MA	1	2000-01-01
ATL	Atlanta, GA	DL	1857	BOS	Boston, MA	2	2000-01-01
ATL	Atlanta, GA	DL	1997	BOS	Boston, MA	3	2000-01-01
ATL	Atlanta, GA	DL	2065	BOS	Boston, MA	4	2000-01-01
ATL	Atlanta, GA	US	2619	BOS	Boston, MA	5	2000-01-01
ATL	Atlanta, GA	US	2621	BOS	Boston, MA	6	2000-01-01
ATL	Atlanta, GA	DL	346	BTR	Baton Rouge, LA	7	2000-01-01
ATL	Atlanta, GA	DL	412	BTR	Baton Rouge, LA	8	2000-01-01
ATL	Atlanta, GA	DL	299	BUF	Buffalo, NY	9	2000-01-01
ATL	Atlanta, GA	DL	495	BUF	Buffalo, NY	10	2000-01-01
ATL	Atlanta, GA	DL	677	BUF	Buffalo, NY	11	2000-01-01
ATL	Atlanta, GA	DL	251	BWI	Baltimore, MD	12	2000-01-01
ATL	Atlanta, GA	DL	1003	BWI	Baltimore, MD	13	2000-01-01
ATL	Atlanta, GA	DL	1501	BWI	Baltimore, MD	14	2000-01-01
ATL	Atlanta, GA	DL	1907	BWI	Baltimore, MD	15	2000-01-01
ATL	Atlanta, GA	DL	2063	BWI	Baltimore, MD	16	2000-01-01
ATL	Atlanta. GA	DL	2111	BWI	Baltimore. MD	17	2000-01-01

Command took 1.67 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:27:57 PM on demo-cluster

Here is the execution plan for our latest query.
So, this query is scanning part_tbl. Click the details.

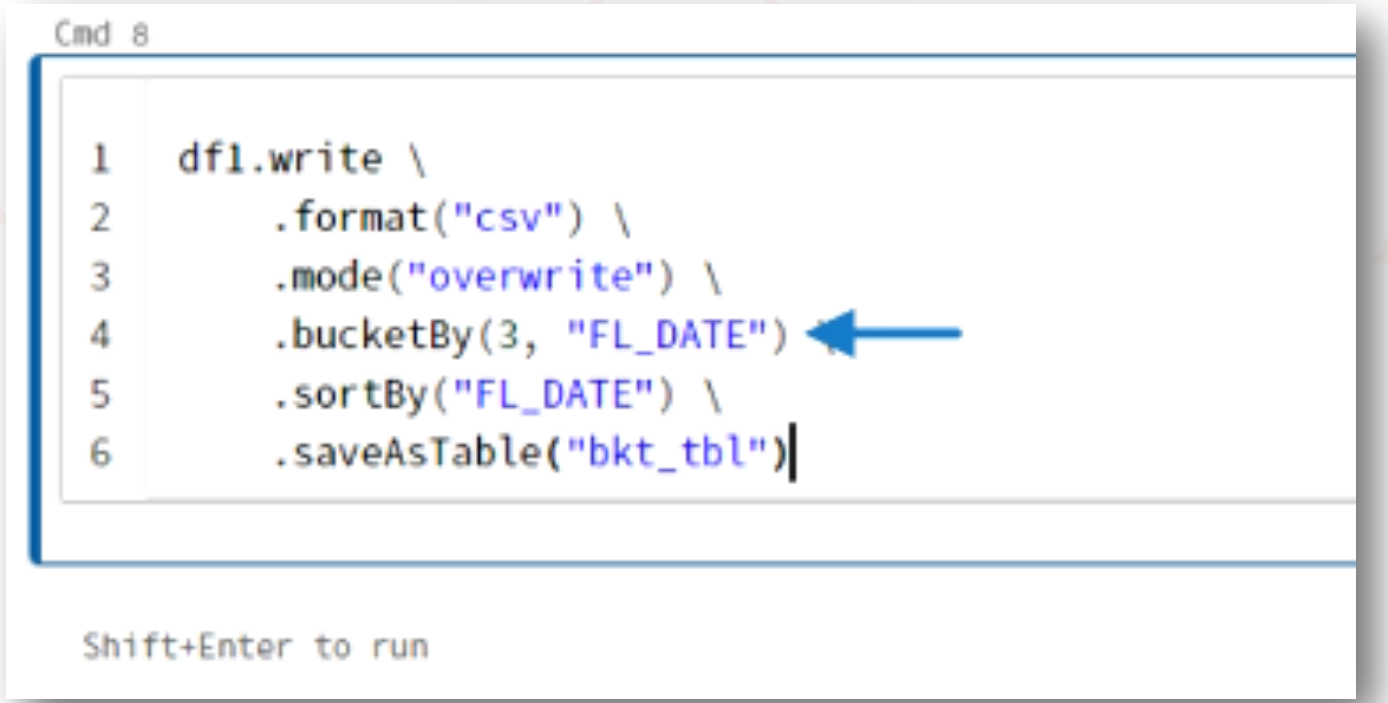


Check the number of files read.
There is only one file.
And that happens because Spark applied partition pruning.
The table was partitioned on FL_DATE. So when we query for FL_DATE equals 1st Jan 2000, Spark will look into a single directory.
You already learned all that, and we know partitionBy() is a great tool to optimize your where clause and record filtering.

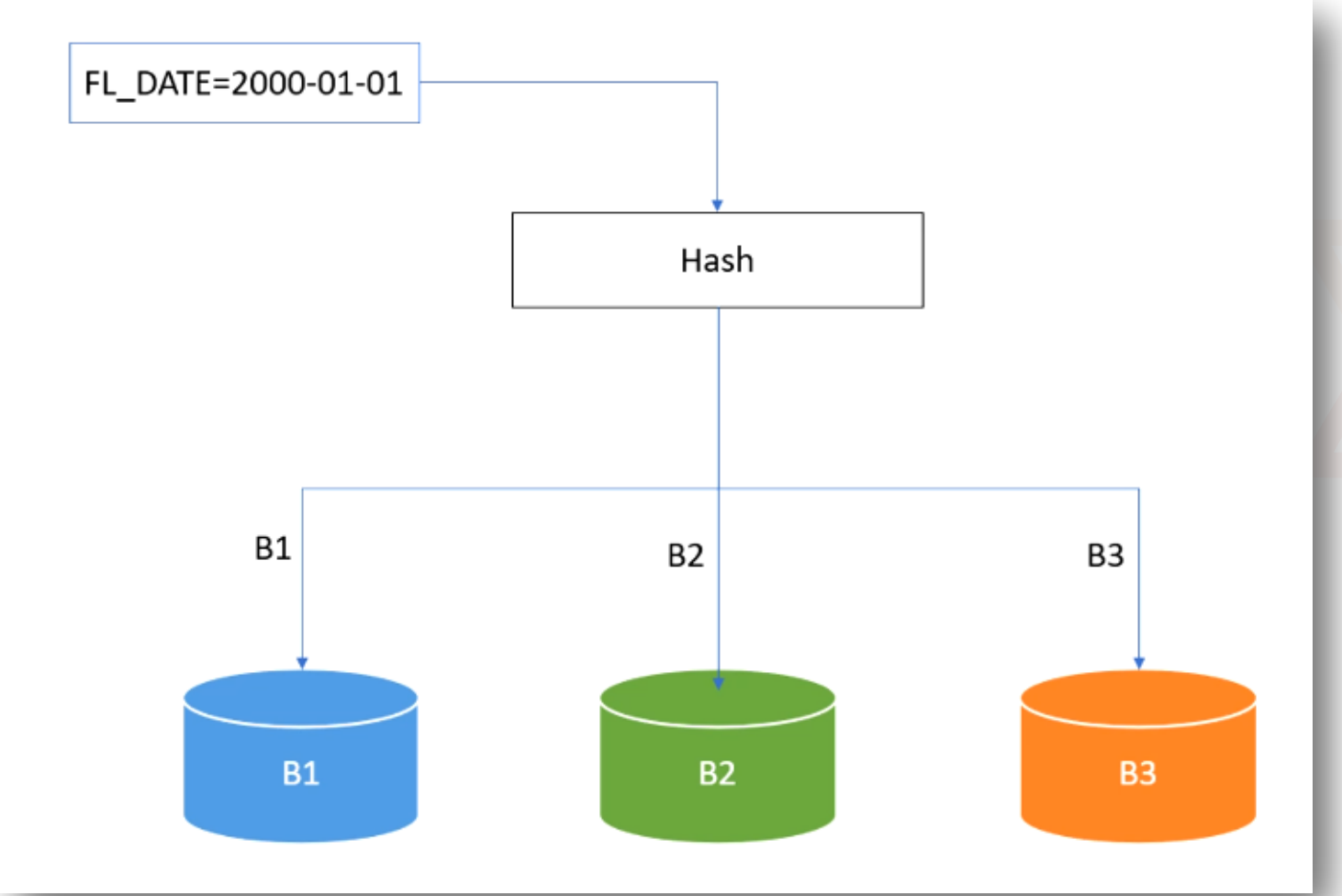


Now let me go back to the notebook once again, and type the command shown below.
What am I doing? I am checking the initial partition count in my Dataframe.
Why? I will tell you that in a minute.
My Dataframe is loaded into memory using eight RDD partitions. Keep that information for now, and we will come back to it.

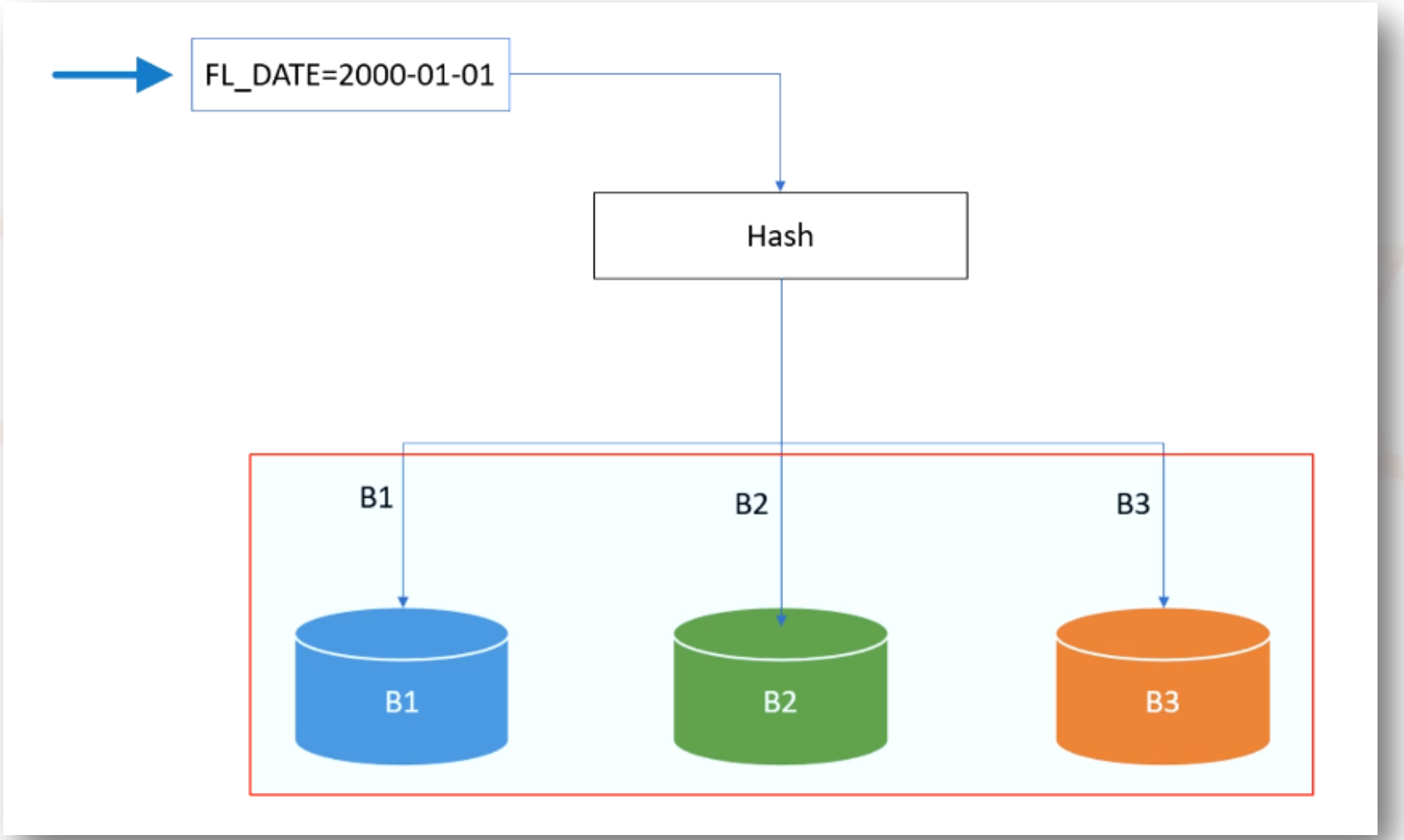
Now go ahead and create a bucketBy() table.
The bucketBy() is similar to partitionBy() with a small difference. The partitionBy() will read the FL_DATE column and create one partition directory for each unique value. So If you have 300 unique FL_DATE values, the partitionBy will create 300 partition directories.
But the bucketBy() takes the number as its first argument. I am giving 3 here. So the bucketBy() will create only three buckets. I might have 300 FL_DATE, but the bucketBy() will create only three buckets and fit all the data into those three buckets.



Let us see how this bucketing happens.
DataFrameWriter implements an intelligent hashing algorithm on the bucketing column.
We wanted to implement three buckets. So the hashing algorithm will generate only three numbers. It will take the record and hash it.



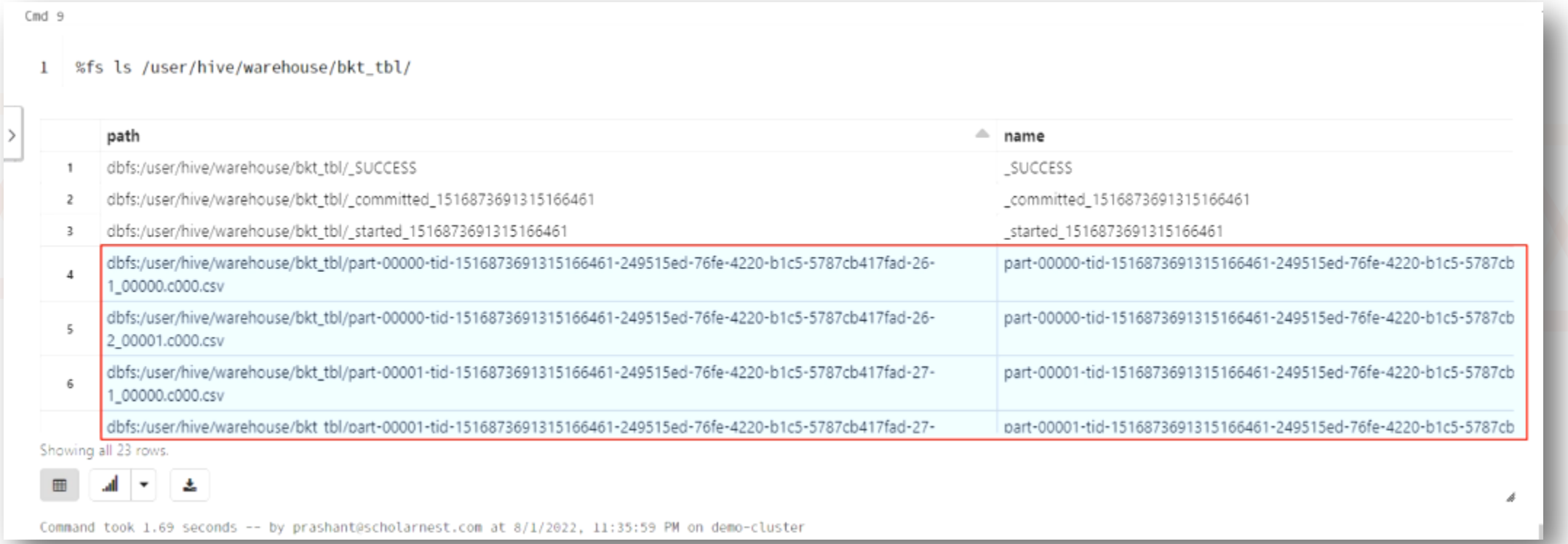
If the value is B1, the record goes to the B1 bucket. If the hash value is B2, the record goes in the B2 bucket. Similarly, the DataFrameWriter will hash the FL_DATE column for each record and place them in one of the three buckets.



Let's come back to the code.
So bucketBy() is similar to partitionBy(). But it will create a fixed number of buckets, whereas the partitionBy() will create one partition directory for each unique value. That's the first difference.
I am also using the sortBy() after the bucketBy(). The sortBy() is optional, but we almost always use it after the bucketBy(). The sortBy() will keep the data sorted inside the bucket.
And that helps to avoid sorting at the read time because data is always sorted in the bucket.

Now I ran the code and created a bucket table.
Let me show you another difference between partitionBy() and bucketBy().

Here is the directory structure for the bucketed table.
We do not see directories. Instead, you will see files. And that's the second difference.
The bucketBy() does not create partition directories. It will create bucket files.



Cmd 9

```
1  %fs ls /user/hive/warehouse/bkt_tbl/
```

	path	name
1	dbfs:/user/hive/warehouse/bkt_tbl/_SUCCESS	_SUCCESS
2	dbfs:/user/hive/warehouse/bkt_tbl/_committed_1516873691315166461	_committed_1516873691315166461
3	dbfs:/user/hive/warehouse/bkt_tbl/_started_1516873691315166461	_started_1516873691315166461
4	dbfs:/user/hive/warehouse/bkt_tbl/part-00000-tid-1516873691315166461-249515ed-76fe-4220-b1c5-5787cb417fad-26-1_00000.c000.csv	part-00000-tid-1516873691315166461-249515ed-76fe-4220-b1c5-5787cb
5	dbfs:/user/hive/warehouse/bkt_tbl/part-00000-tid-1516873691315166461-249515ed-76fe-4220-b1c5-5787cb417fad-26-2_00001.c000.csv	part-00000-tid-1516873691315166461-249515ed-76fe-4220-b1c5-5787cb
6	dbfs:/user/hive/warehouse/bkt_tbl/part-00001-tid-1516873691315166461-249515ed-76fe-4220-b1c5-5787cb417fad-27-1_00000.c000.csv	part-00001-tid-1516873691315166461-249515ed-76fe-4220-b1c5-5787cb
	dbfs:/user/hive/warehouse/bkt_tbl/part-00001-tid-1516873691315166461-249515ed-76fe-4220-b1c5-5787cb417fad-27-	part-00001-tid-1516873691315166461-249515ed-76fe-4220-b1c5-5787cb

Showing all 23 rows.

Command took 1.69 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:35:59 PM on demo-cluster

You should expect three files here. Why? Because we created three buckets. So I expect three bucket files. But how many are there?
If you scroll down, you will see many files. So we have more than three bucket files.
Why is that? I asked for three buckets, so it should create only three bucket files. But why do we have so many files? There comes the third difference. The bucketBy() will create buckets for each RDD partition. That's why I showed you the initial partition count (which was 8).

So let's assume your Dataframe is loaded as 8 RDD partitions. You are doing bucketBy and asking Spark to create three buckets. But Spark will create three buckets for each RDD partition. So if I have eight RDD partitions, I can expect 24 bucket files. It could be less than 24 because some buckets might not have data. But you can expect a maximum of 24 bucket files.

And that's a problem with the bucketing.
The bucketBy() will break your RDD partitions into smaller files, and you will end up creating a large number of small files.
And that's not good. Spark struggle to handle a large number of small files.
The best size for Spark is a 128 MB file size, and you can go up to 1 GB of files.
That's the sweet spot for Spark.

But the bucketBy() could break this sweet spot and create hundreds or thousands of small files, causing some performance problems.
How do we handle it?

You can apply the hash algorithm and repartition your Dataframe before writing it.
Let us try doing that.

What am I doing here in the modified code? I am hashing the FL_DATE, dividing it by three, and taking the mod(). This trick will make three RDD partitions for my Dataframe using the hash value of the FL_DATE. This is precisely the same as what bucketBy() will do. And this trick will result in only three bucket files solving the small file problem.
I ran this code, now let me check the directory again.

Cmd 8

```
1  df1.repartition(expr("pmod(hash(FL_DATE), 3)")) \
2      .write \
3      .format("csv") \
4      .mode("overwrite") \
5      .bucketBy(3, "FL_DATE") \
6      .sortBy("FL_DATE") \
7      .saveAsTable("bkt_tbl")
```

▸ (2) Spark Jobs

Command took 16.87 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:43:42 PM on demo-cluster

So now we have only three bucket files.
But what is the benefit of creating a bucket? We use partitionBy() to get partition pruning and run our SQL faster. Do we get similar benefits with the bucketBy()?
Let us check.

Cmd 9

```
1  %fs ls /user/hive/warehouse/bkt_tbl/
```

	path	name
1	dbfs:/user/hive/warehouse/bkt_tbl/_SUCCESS	_SUCCESS
2	dbfs:/user/hive/warehouse/bkt_tbl/_committed_2429790592294476289	_committed_2429790592294476289
3	dbfs:/user/hive/warehouse/bkt_tbl/_started_2429790592294476289	_started_2429790592294476289
4	dbfs:/user/hive/warehouse/bkt_tbl/part-00000-tid-2429790592294476289-0f0a242c-b283-4c18-b800-08d1ff86eed5-50-1_00001.c000.csv	part-00000-tid-2429790592294476289-0f0a242c-b283-4c18-b800-08d1ff8
5	dbfs:/user/hive/warehouse/bkt_tbl/part-00001-tid-2429790592294476289-0f0a242c-b283-4c18-b800-08d1ff86eed5-51-1_00002.c000.csv	part-00001-tid-2429790592294476289-0f0a242c-b283-4c18-b800-08d1ff8
6	dbfs:/user/hive/warehouse/bkt_tbl/part-00002-tid-2429790592294476289-0f0a242c-b283-4c18-b800-08d1ff86eed5-52-1_00000.c000.csv	part-00002-tid-2429790592294476289-0f0a242c-b283-4c18-b800-08d1ff8

Showing all 6 rows.

So this one is the same query. But I am running it on the bucket table.
Let's go to the Spark UI and see the plan now.

Cmd 10

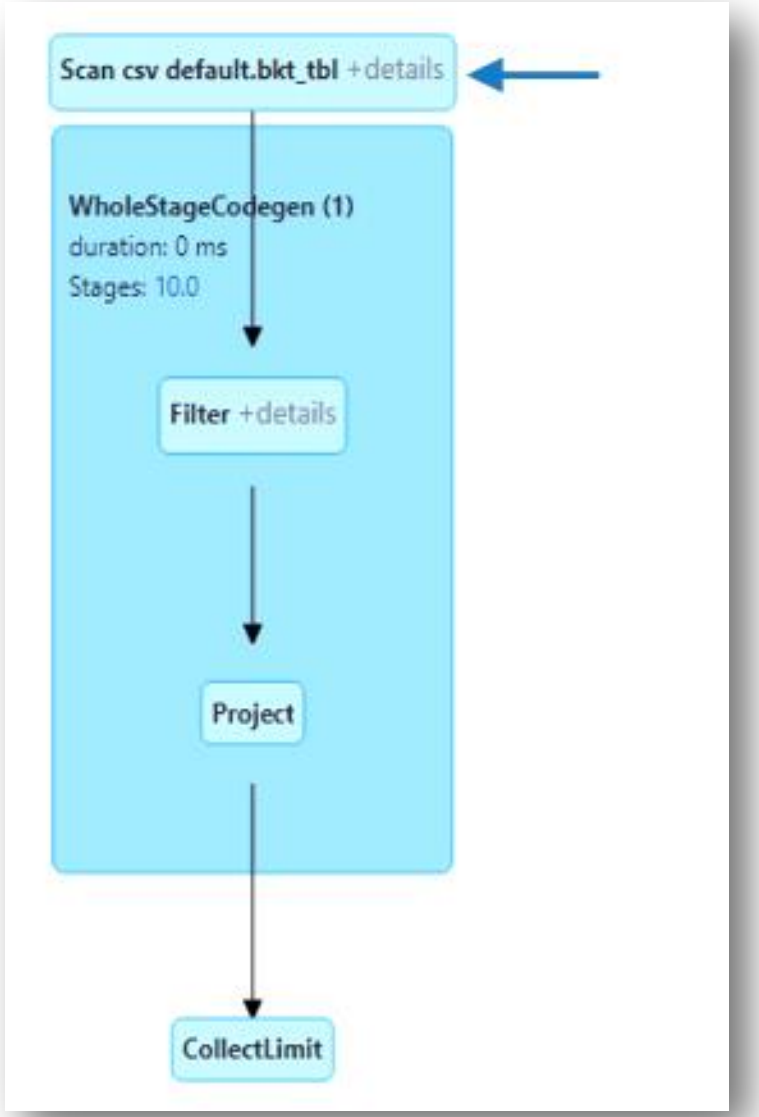
```
1  spark.sql("select * from bkt_tbl where FL_DATE='2000-01-01'").show()
```

▸ (1) Spark Jobs

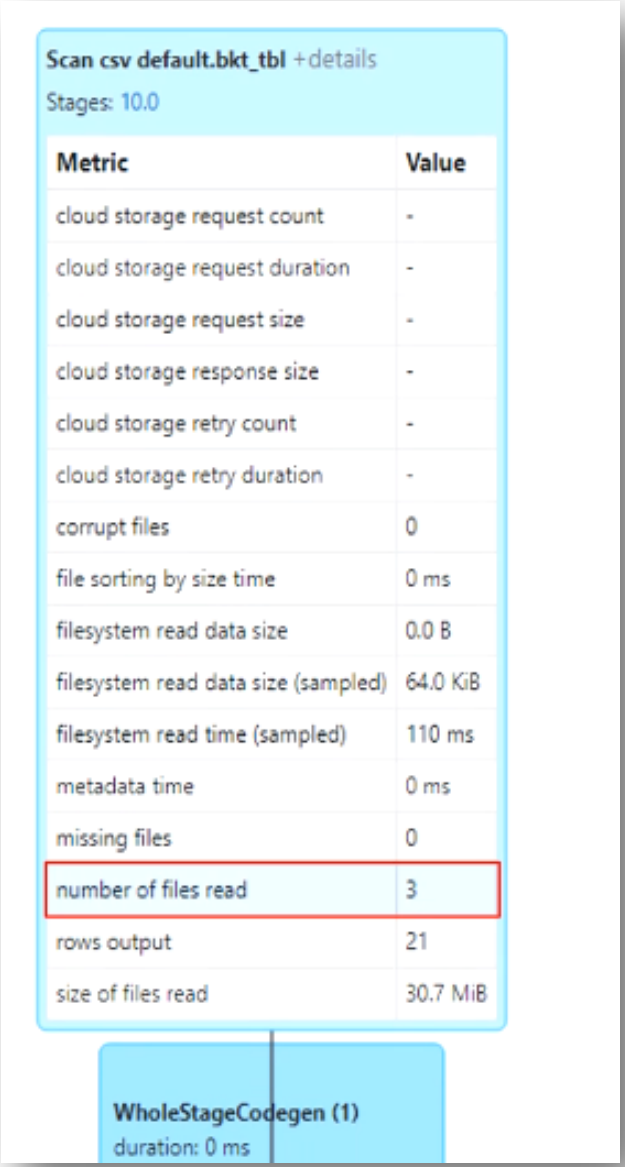
DEST	DEST_CITY_NAME	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	id
ATL	Atlanta, GA	2000-01-01	DL	1451	BOS	Boston, MA	0
ATL	Atlanta, GA	2000-01-01	DL	1479	BOS	Boston, MA	1
ATL	Atlanta, GA	2000-01-01	DL	1857	BOS	Boston, MA	2
ATL	Atlanta, GA	2000-01-01	DL	1997	BOS	Boston, MA	3
ATL	Atlanta, GA	2000-01-01	DL	2065	BOS	Boston, MA	4
ATL	Atlanta, GA	2000-01-01	US	2619	BOS	Boston, MA	5
ATL	Atlanta, GA	2000-01-01	US	2621	BOS	Boston, MA	6
ATL	Atlanta, GA	2000-01-01	DL	346	BTR	Baton Rouge, LA	7
ATL	Atlanta, GA	2000-01-01	DL	412	BTR	Baton Rouge, LA	8
ATL	Atlanta, GA	2000-01-01	DL	299	BUF	Buffalo, NY	9
ATL	Atlanta, GA	2000-01-01	DL	495	BUF	Buffalo, NY	10
ATL	Atlanta, GA	2000-01-01	DL	677	BUF	Buffalo, NY	11
ATL	Atlanta, GA	2000-01-01	DL	251	BWI	Baltimore, MD	12
ATL	Atlanta, GA	2000-01-01	DL	1003	BWI	Baltimore, MD	13
ATL	Atlanta, GA	2000-01-01	DL	1501	BWI	Baltimore, MD	14
ATL	Atlanta, GA	2000-01-01	DL	1907	BWI	Baltimore, MD	15
ATL	Atlanta, GA	2000-01-01	DL	2063	BWI	Baltimore, MD	16
ATL	Atlanta, GA	2000-01-01	DL	2111	BWI	Baltimore, MD	17

Command took 1.57 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:46:10 PM on demo-cluster

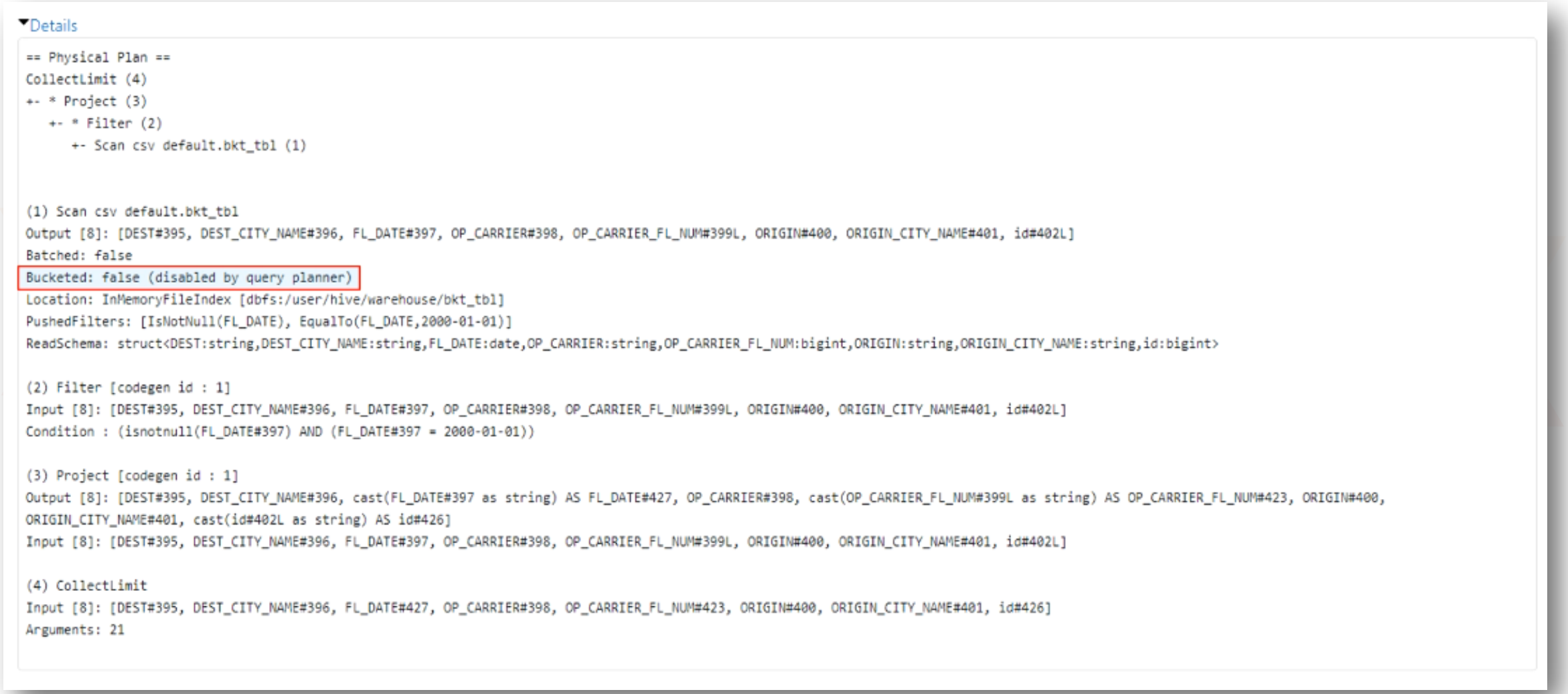
So, this plan is to scan the bucket table. Click the details.



How many files do we read?
All three.



Next, go down to the Details link at the bottom of the page.
And look for the Bucketed: false (disabled by query planner).



Details

```
== Physical Plan ==
CollectLimit (4)
+- * Project (3)
   +- * Filter (2)
      +- Scan csv default.bkt_tbl (1)


(1) Scan csv default.bkt_tbl
Output [8]: [DEST#395, DEST_CITY_NAME#396, FL_DATE#397, OP_CARRIER#398, OP_CARRIER_FL_NUM#399L, ORIGIN#400, ORIGIN_CITY_NAME#401, id#402L]
Batched: false
Bucketed: false (disabled by query planner)
Location: InMemoryFileIndex [dbfs:/user/hive/warehouse/bkt_tbl]
PushedFilters: [IsNotNull(FL_DATE), EqualTo(FL_DATE,2000-01-01)]
ReadSchema: struct<DEST:string,DEST_CITY_NAME:string,FL_DATE:date,OP_CARRIER:string,OP_CARRIER_FL_NUM:bigint,ORIGIN:string,ORIGIN_CITY_NAME:string,id:bigint>

(2) Filter [codegen id : 1]
Input [8]: [DEST#395, DEST_CITY_NAME#396, FL_DATE#397, OP_CARRIER#398, OP_CARRIER_FL_NUM#399L, ORIGIN#400, ORIGIN_CITY_NAME#401, id#402L]
Condition : (isnotnull(FL_DATE#397) AND (FL_DATE#397 = 2000-01-01))

(3) Project [codegen id : 1]
Output [8]: [DEST#395, DEST_CITY_NAME#396, cast(FL_DATE#397 as string) AS FL_DATE#427, OP_CARRIER#398, cast(OP_CARRIER_FL_NUM#399L as string) AS OP_CARRIER_FL_NUM#423, ORIGIN#400,
ORIGIN_CITY_NAME#401, cast(id#402L as string) AS id#426]
Input [8]: [DEST#395, DEST_CITY_NAME#396, FL_DATE#397, OP_CARRIER#398, OP_CARRIER_FL_NUM#399L, ORIGIN#400, ORIGIN_CITY_NAME#401, id#402L]

(4) CollectLimit
Input [8]: [DEST#395, DEST_CITY_NAME#396, FL_DATE#427, OP_CARRIER#398, OP_CARRIER_FL_NUM#423, ORIGIN#400, ORIGIN_CITY_NAME#401, id#426]
Arguments: 21
```


So this is what is happening?
Spark query planner decided to disable the bucket optimization.
So the bucketing is not likely to help your where clause and filtering.

The partitionBy() is designed to help you with partition pruning.
Bucketing is not a solution for filtering records and reading only a few buckets.
In an ideal case, it should do that.
But in Spark, this feature is still under refinement and is being disabled by the query planner.
It might start working correctly in later versions. But for now, bucketing is not for optimizing where clause and record filters.

So we learned some basics about bucketBy(). Let me summarize what we learned till now.

1. Spark bucketBy() allows you to define a fixed number of buckets for given columns and create a bucketed table.
2. You can also sort your buckets using the sortBy() method.
3. Bucketing is only applicable for Spark Managed tables. Why? Because the bucket information is stored in the metadata. So it applies to only Spark-managed tables.
4. The bucketBy() does not create directory similar to partitionBy(). Instead, It will create bucket files.
5. If you are creating 100 buckets, the bucketBy will create 100 bucket files for each input partition. This may lead to a small size problem. You can handle the small size problem if you repartition your input Dataframe before bucketing it.
6. Bucketing is not a good solution for optimizing where clauses and filtering records.

So what is the benefit of bucketing?
Why do we use bucketing if they do not help us in faster queries and bucket pruning?

The buckets are designed to avoid shuffling in some cases.
That's why they exist.

We already learned joining two large tables causes shuffle Join.
Small table to large table can use broadcast join and perform faster.
But two large table joins will always perform shuffle and take a lot of time.

Bucketing is a solution to avoid shuffling.

Let me show you.

I will start fresh by creating my first Dataframe.
I am reading data from the d1 directory and creating a Dataframe.
This is my first Dataframe.

Then I will save this Dataframe and create a bucket table.
This time, I am reading three buckets on the ID column. Why ID column, because I want to join this table using the ID column.

Cmd 12

```
1  df1.repartition(expr("pmod(hash(id), 3)")) \
2      .write \
3      .format("csv") \
4      .mode("overwrite") \
5      .bucketBy(3, "id") \
6      .sortBy("id") \
7      .saveAsTable("bkt_tbl_1")
```

▸ (2) Spark Jobs

Command took 11.99 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:56:56 PM on demo-cluster

So I have my first Dataframe, which is a plain non-bucketed Dataframe, and I also have a bucket table.

```
1  df1 = spark.read \
2          .format("json") \
3          .load("/FileStore/bucket/d1/")
```

▸ (1) Spark Jobs

Command took 3.94 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:53:57 PM on demo-cluster

Cmd 12

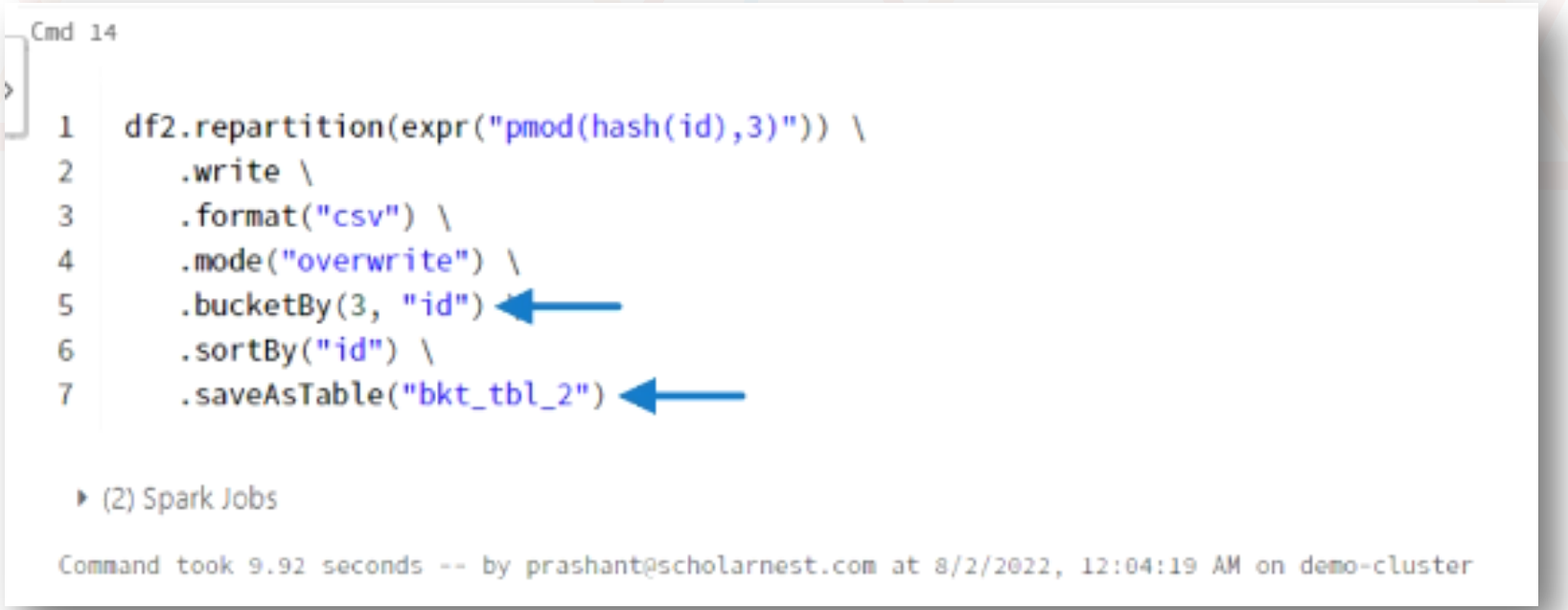
```
1  df1.repartition(expr("pmod(hash(id), 3)")) \
2      .write \
3      .format("csv") \
4      .mode("overwrite") \
5      .bucketBy(3, "id") \
6      .sortBy("id") \
7      .saveAsTable("bkt_tbl_1")
```

▸ (2) Spark Jobs

Command took 11.99 seconds -- by prashant@scholarnest.com at 8/1/2022, 11:56:56 PM on demo-cluster

Next, I am reading another dataset and creating one more Dataframe.
This time, I am reading data from the d2 directory and creating df2.
(Reference: /data/bucket/d2)

I am saving the second Dataframe as a bucket table as well, as shown below.
I saved the second dataframe, and the table name is bkt_tbl_2.
I created this table also using the same three buckets on the ID column. Why three buckets and an ID column. Because I plan to join this table with the previous table. For an optimal bucket join, both tables must have the same number of buckets using the same bucket columns. That's why I am trying to keep the same bucket structure for both tables.
So we are now ready with the two bucket tables. But before I join these two tables, let's join the two data frames.



Here is the code. The Dataframe is not bucketed. These are plain data frames.
So If I join them, we will see shuffle join. Let us go to the Spark UI and check that.

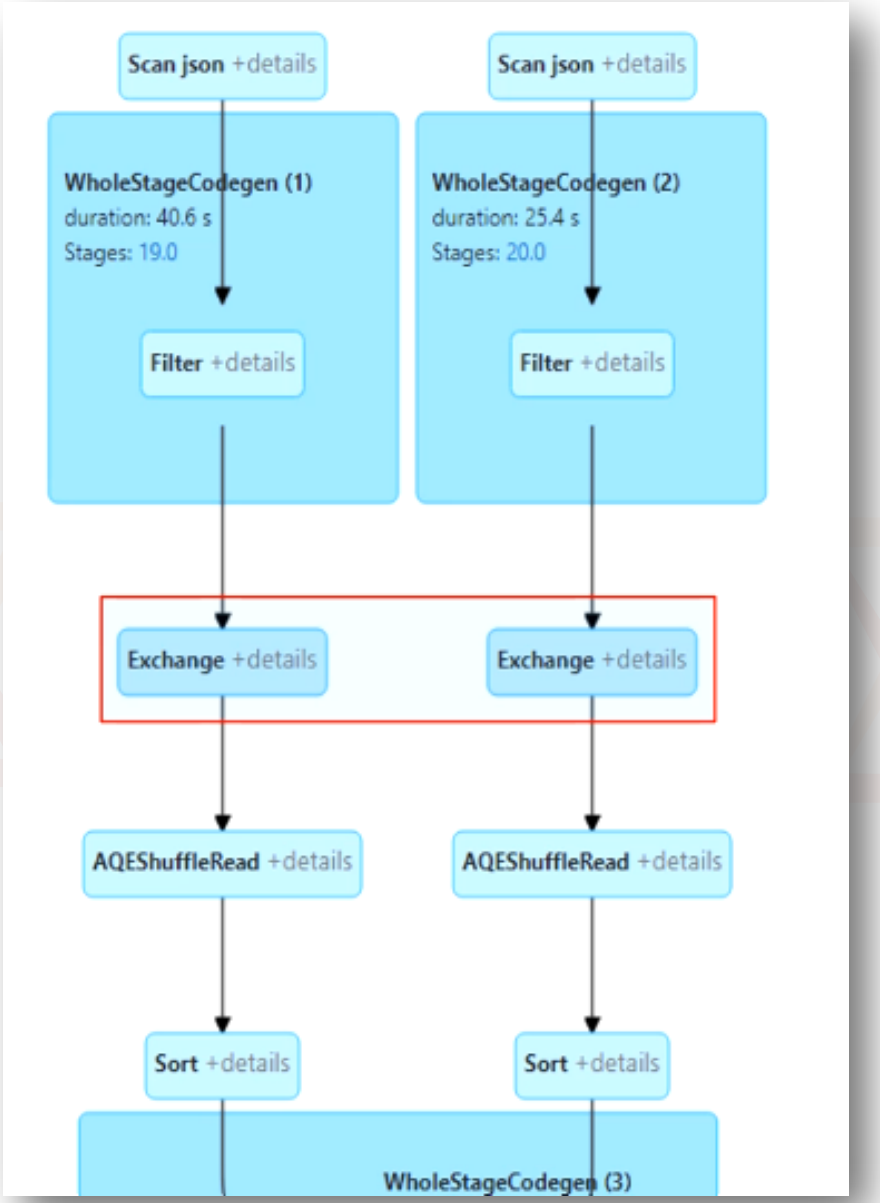
```
1  df1.join(df2, df1.id==df2.id, "inner").show()
```

▸ (3) Spark Jobs

```
+----+--------------+--------+----------+----------------+------+--------------------+---+--------+---------+-----------+-----------+--------+--------+---
----+---------+---+
|DEST|DEST_CITY_NAME| FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|    ORIGIN_CITY_NAME| id|ARR_TIME|CANCELLED|CRS_ARR_TIME|CRS_DEP_TIME|DEP_TIME|DISTANCE|TAX
I_IN|WHEELS_ON| id|
+----+--------------+--------+----------+----------------+------+--------------------+---+--------+---------+-----------+-----------+--------+--------+---
----+---------+---+
| ATL|   Atlanta, GA|1/1/2000|        DL|            1451|   BOS|          Boston, MA|  0|    1348|        0|       1400|       1115|    1113|     946|
5|     1343|  0|
| ATL|   Atlanta, GA|1/1/2000|        DL|             346|   BTR|     Baton Rouge, LA|  7|    2006|        0|       2008|       1740|    1744|     449|
9|     1957|  7|
| ATL|   Atlanta, GA|1/1/2000|        US|            2967|   BWI|       Baltimore, MD| 19|    1851|        0|       1853|       1700|    1700|     576|
6|     1845| 19|
| ATL|   Atlanta, GA|1/1/2000|        DL|            1289|   CAE|        Columbia, SC| 22|    1628|        0|       1633|       1515|    1514|     191|
9|     1619| 22|
| ATL|   Atlanta, GA|1/1/2000|        DL|            2030|   CAE|        Columbia, SC| 26|    null|        1|       1450|       1340|    null|     191|
null|     null| 26|
| ATL|   Atlanta, GA|1/1/2000|        DL|             717|   CHS|      Charleston, SC| 29|    1439|        0|       1454|       1340|    1340|     259|
7|     1432| 29|
| ATL|   Atlanta, GA|1/1/2000|        DL|            1808|   CHS|      Charleston, SC| 34|     838|        0|        842|        730|     728|     259|
12|      826| 34|
| ATL|   Atlanta. GA|1/1/2000|        US|            2187|   CLT|       Charlotte. NC| 50|    null|        1|       1451|       1330|    null|     227|
Command took 10.99 seconds -- by prashant@scholarnest.com at 8/2/2022, 12:07:27 AM on demo-cluster
```


Here is the DAG.
We have three stages in the join.
And we see two exchanges.
You will also see the sort-merge join.

This is a typical shuffle join plan.
We see an exchange, so it is a shuffle Join.



Next, I am trying to join the bucket tables.
I am using SQL. Why? Because these are managed tables, SQL is an easy way to work with managed tables. You could use the Dataframe expression also for doing the same thing. But I am using SQL because it is easy to use SQL.
Now let us check the execution plan for this.

Cmd 16

```
1  %sql
2  select * from bkt_tbl_1 join bkt_tbl_2 on bkt_tbl_1.id = bkt_tbl_2.id
```

▸ (1) Spark Jobs

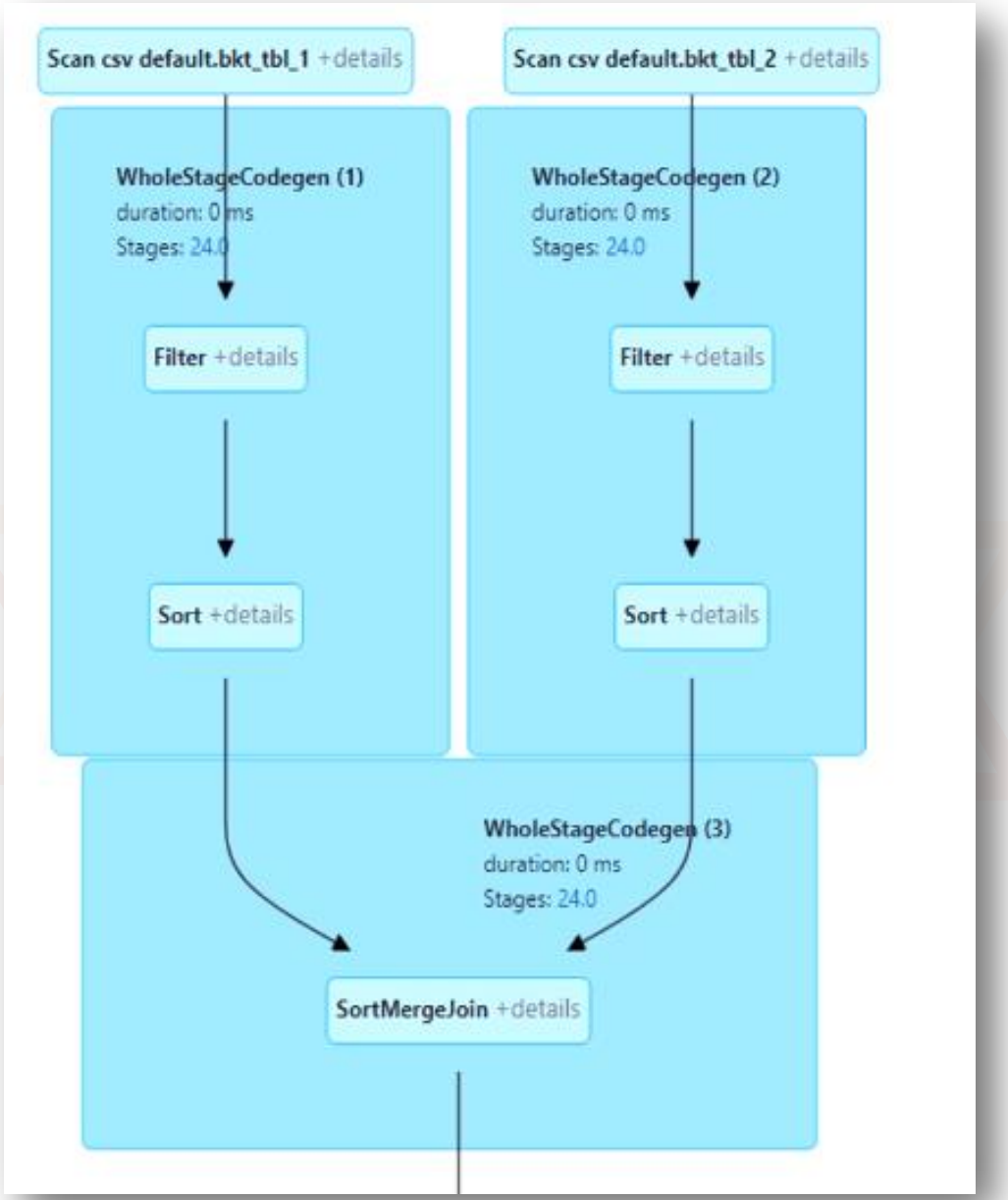
	DEST	DEST_CITY_NAME	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	id	ARR_TIME
1	ATL	Atlanta, GA	1/1/2000	DL	1997	BOS	Boston, MA	3	2005
2	ATL	Atlanta, GA	1/1/2000	US	2621	BOS	Boston, MA	6	1717
3	ATL	Atlanta, GA	1/1/2000	DL	346	BTR	Baton Rouge, LA	7	2006
4	ATL	Atlanta, GA	1/1/2000	DL	677	BUF	Buffalo, NY	11	947
5	ATL	Atlanta, GA	1/1/2000	DL	2063	BWI	Baltimore, MD	16	null
6	ATL	Atlanta, GA	1/1/2000	DL	1519	CAE	Columbia, SC	24	821
7	ATL	Atlanta, GA	1/1/2000	DL	423	CHS	Charleston, SC	28	711

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

Command took 4.83 seconds -- by prashant@scholarnest.com at 8/2/2022, 12:10:04 AM on demo-cluster

We still see a three-stage plan.
But we do not have an exchange.
What does it mean? Simple! It is not a shuffle Join.
Spark is not sending data from one exchange to another exchange.
And that's why it is not doing the shuffle and not showing the exchange here.

And that's the power of bucket tables.
So if you know you will join two large tables, planning the Join in advance makes sense.
Create bucket tables so you can join them faster.



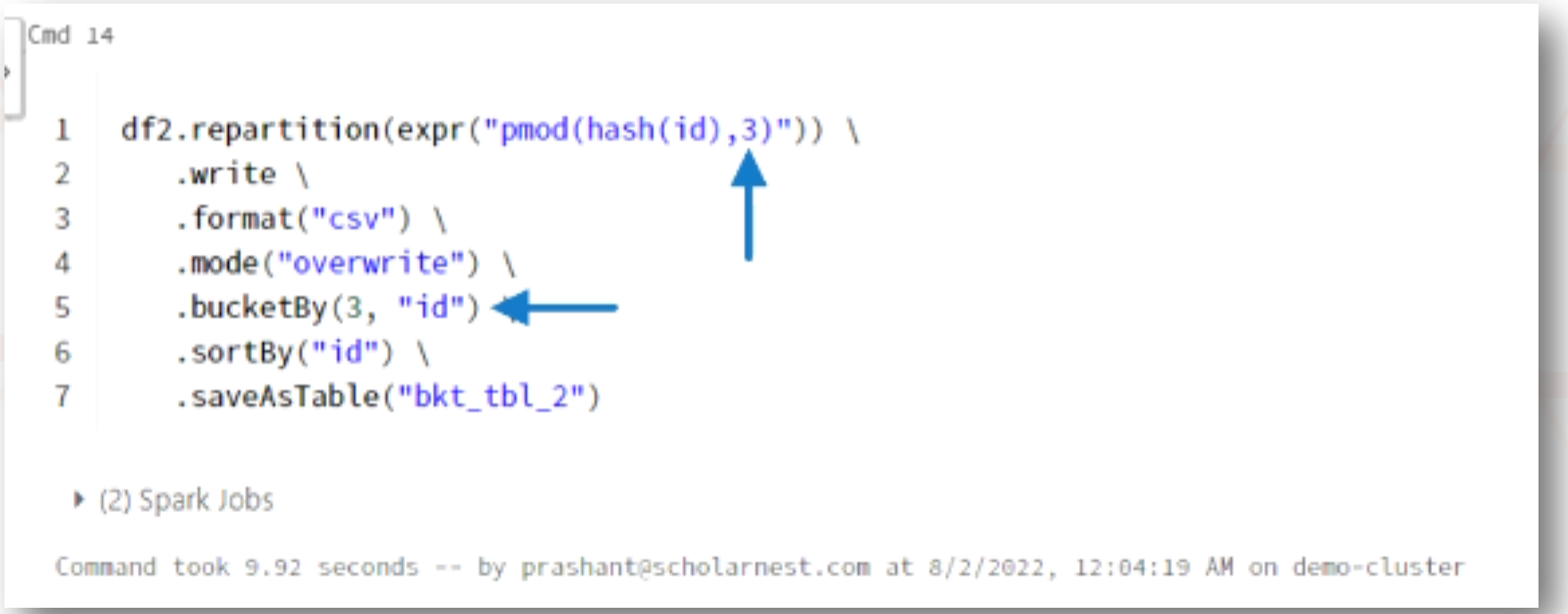
So we learned that bucket tables are a great way to optimize your joins.
If you know you have some tables you will be joining frequently for various queries, it makes sense to create bucket tables.

One last thing before we close this lecture.
I create three partitions for both bucket tables.
Why? Because I wanted to get the best results.

But join gives you the best results when you have the same number of buckets on both sides.
But it is not mandatory.

Let's see what happens when we have different bucket counts on both sides.

Go back to your notebooks.
Here is my df2 code that creates the bucket file.
I have three buckets here. But now, let me change it to two buckets at both the places.



We have changed the value to 2 at both the places.
So now I have two tables. The first table is three buckets, and the second table is two buckets.

Cmd 14

```
1  df2.repartition(expr("pmod(hash(id),2)")) \
2     .write \
3     .format("csv") \
4     .mode("overwrite") \
5     .bucketBy(2, "id") \
6     .sortBy("id") \
7     .saveAsTable("bkt_tbl_2")
```

▸ (2) Spark Jobs

Command took 9.55 seconds -- by prashant@scholarnest.com at 8/2/2022, 12:11:45 AM on demo-cluster

Now run the join SQL once again.
Then go to the Spark UI and check the execution plan for this query.

Cmd 16

```
1  %sql
2  select * from bkt_tbl_1 join bkt_tbl_2 on bkt_tbl_1.id = bkt_tbl_2.id
```

▸ (2) Spark Jobs

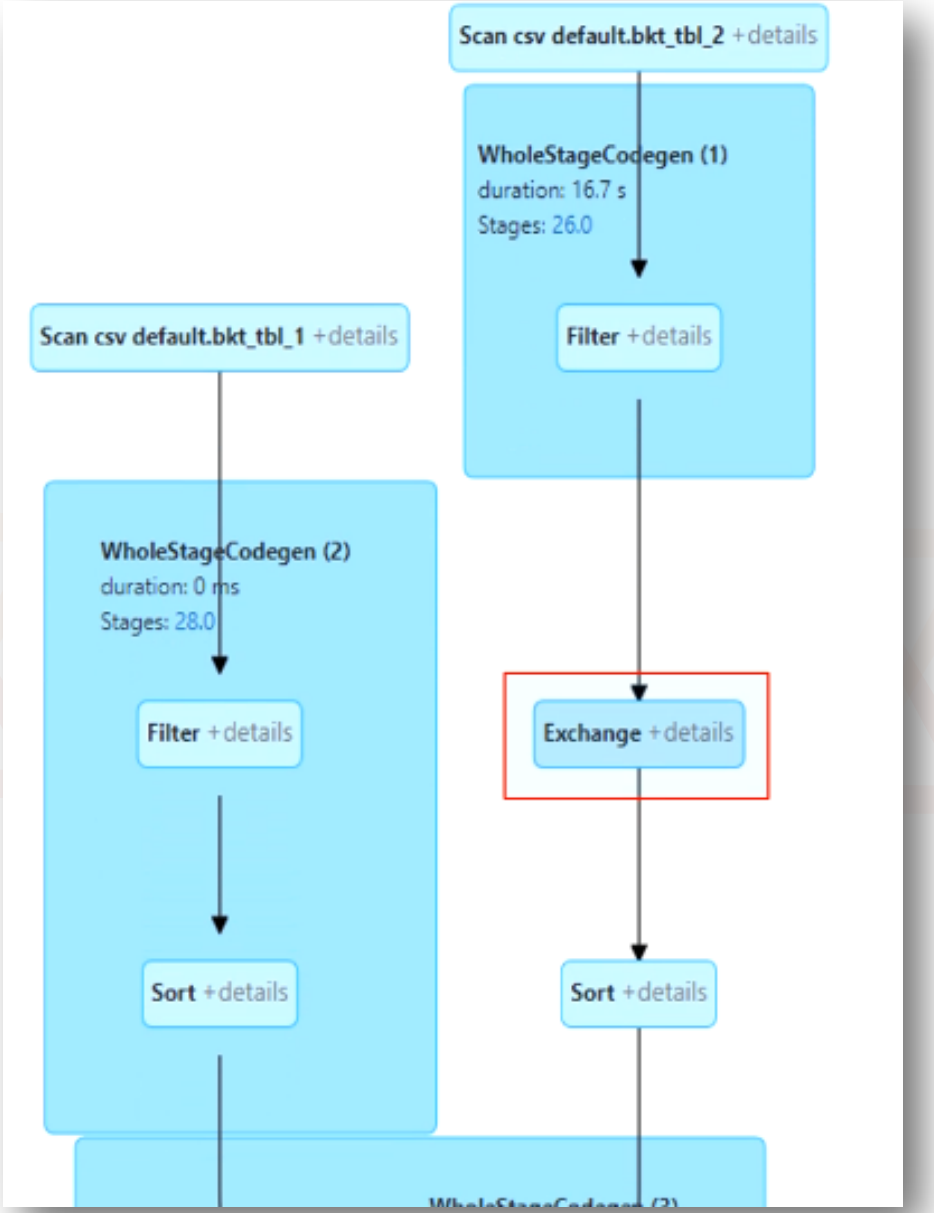
	DEST	DEST_CITY_NAME	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	id	ARR_TIME
1	ATL	Atlanta, GA	1/1/2000	DL	1997	BOS	Boston, MA	3	2005
2	ATL	Atlanta, GA	1/1/2000	US	2621	BOS	Boston, MA	6	1717
3	ATL	Atlanta, GA	1/1/2000	DL	346	BTR	Baton Rouge, LA	7	2006
4	ATL	Atlanta, GA	1/1/2000	DL	677	BUF	Buffalo, NY	11	947
5	ATL	Atlanta, GA	1/1/2000	DL	2063	BWI	Baltimore, MD	16	null
6	ATL	Atlanta, GA	1/1/2000	DL	1519	CAE	Columbia, SC	24	821
7	ATL	Atlanta, GA	1/1/2000	DL	423	CHS	Charleston, SC	28	711

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

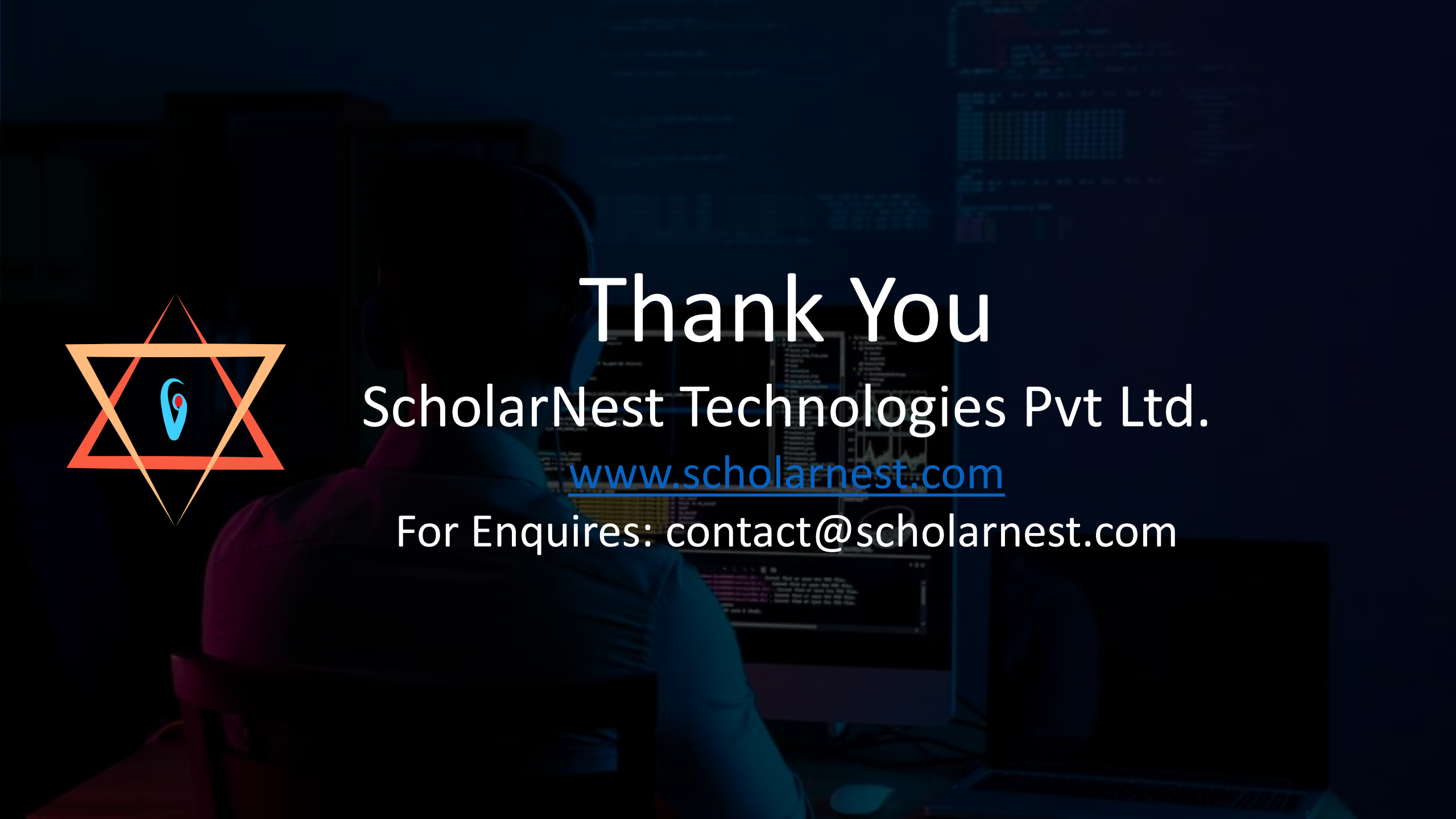
Command took 4.99 seconds -- by prashant@scholarnest.com at 8/2/2022, 12:12:25 AM on demo-cluster

Here is the execution plan.
So we have one exchange now.
This is like a partial shuffle.
A full shuffle will have two exchanges and distribute data from both tables.
But this one is a partial shuffle.
It will distribute data from one table.
But the other table doesn't go through an exchange.
This one is also good.
But not as good as full bucket join.

And that number of buckets is essential.
So when you bucket your tables, make sure you manage the number of buckets.



Bucketing is a great tool to optimize your joins. But it comes with data skew risk.
Let's assume you are bucketing the population using the country.
Your China and India buckets will be huge.
And the bucket for Estonia and Singapore will be pretty small.
And that's data skew. The data skew is a much bigger problem.
Spark AQE will help you break the data skew, but it cannot handle bucket skew.
So be careful with buckets and do not create bucket skew because even Spark AQE will not help you with the bucket skew.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Joins and Optimization

Lecture:
Adaptive Query Execution

Spark Adaptive Query Execution

In this video, I will talk about the Spark Adaptive Query Execution.
Spark Adaptive Query Execution or AQE is a new feature released in Apache Spark 3.0 which offers you three capabilities:

1. Dynamically coalescing shuffle partitions
2. Dynamically switching join strategies
3. Dynamically optimizing skew joins

But what are these features, and why do we need them? What problems do they solve?

So let's try to understand the problems first, and then I will talk about how AQE solves those problems.

Let's assume you are running a super simple group-by query in your Spark code as shown below.

```
SELECT tower_location,
        sum(call_duration) as duration_served
FROM  call_records
GROUP BY  tower_location;
```

Or you might have written a Dataframe expression similar to the following:

```
df.groupBy("tower_location")
   .agg(sum("call_duration").alias("duration_served"))
```


So, I have a call_records table. This table stores information about the cell phone calls made by different users. We record a lot of information in this table, but here is a simplified table structure shown in the screenshot below.
So we record call_id, then the duration of the call in minutes, and we also record which cell tower served the call.

call_id	call_duration	tower_location
10001	6	bangalore_tower_a
10002	12	bangalore_tower_b
10003	15	bangalore_tower_b
10004	28	bangalore_tower_a

And my Spark SQL is trying to get the sum of call duration by the tower_location.

call_id	call_duration	tower_location
10001	6	bangalore_tower_a
10002	12	bangalore_tower_b
10003	15	bangalore_tower_b
10004	28	bangalore_tower_a

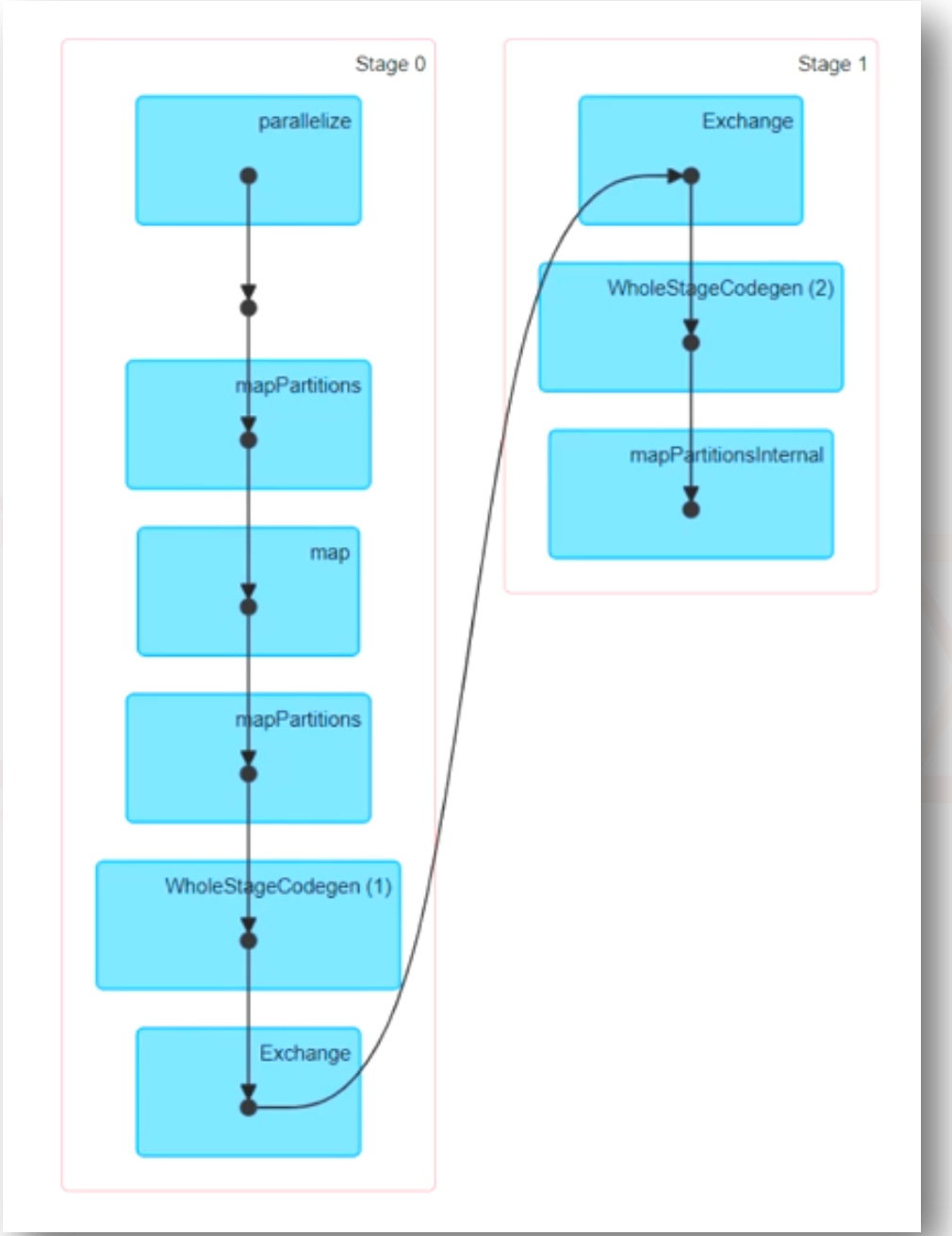
```
SELECT tower_location,
       sum(call_duration) as duration_served
FROM  call_records
GROUP BY  tower_location;
```

```
df.groupBy("tower_location")
   .agg(sum("call_duration").alias("duration_served"))
```

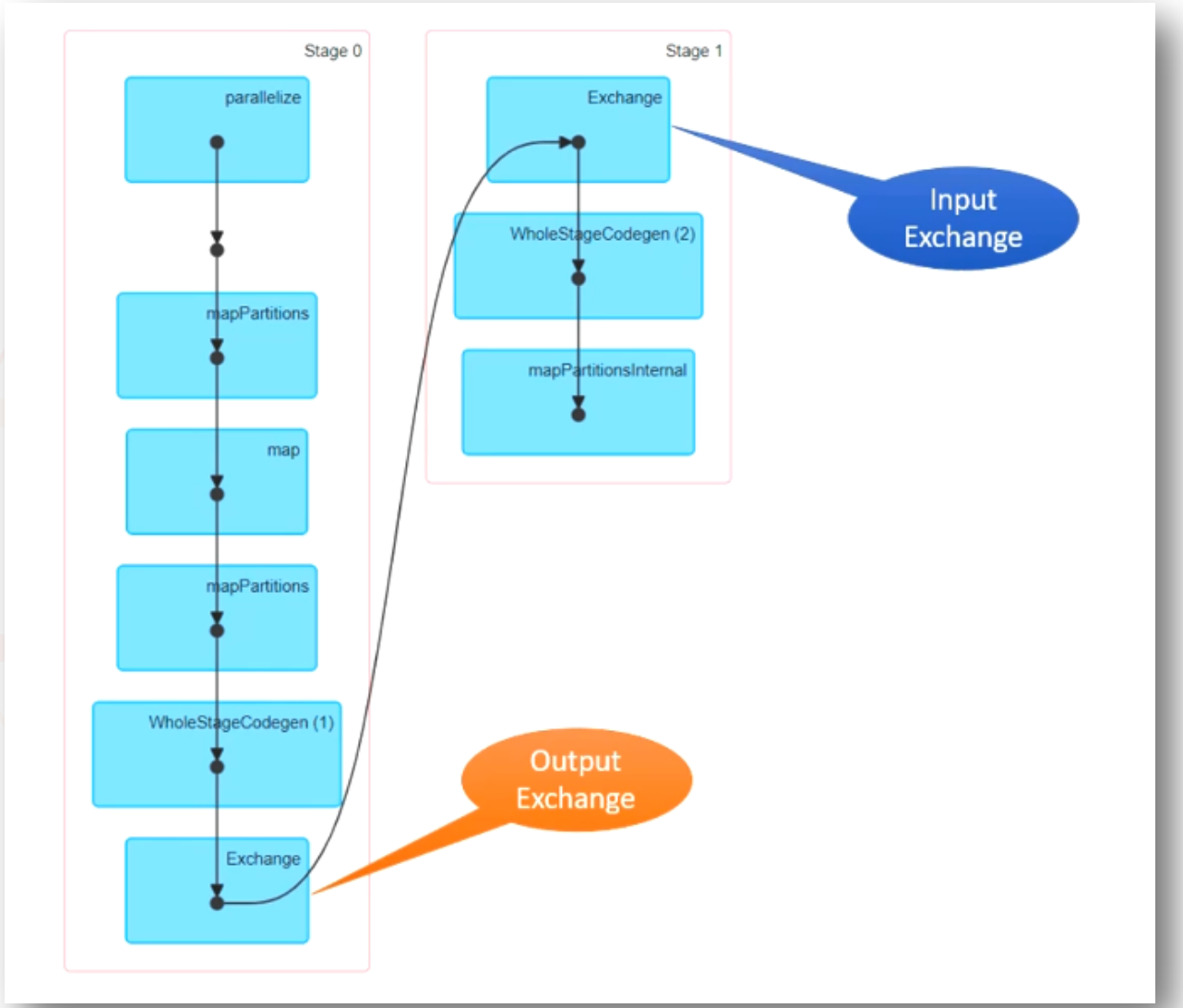

So for the four sample records shown earlier, you should get the result as shown below.

tower_location	duration_served
bangalore_tower_a	34
bangalore_tower_b	27

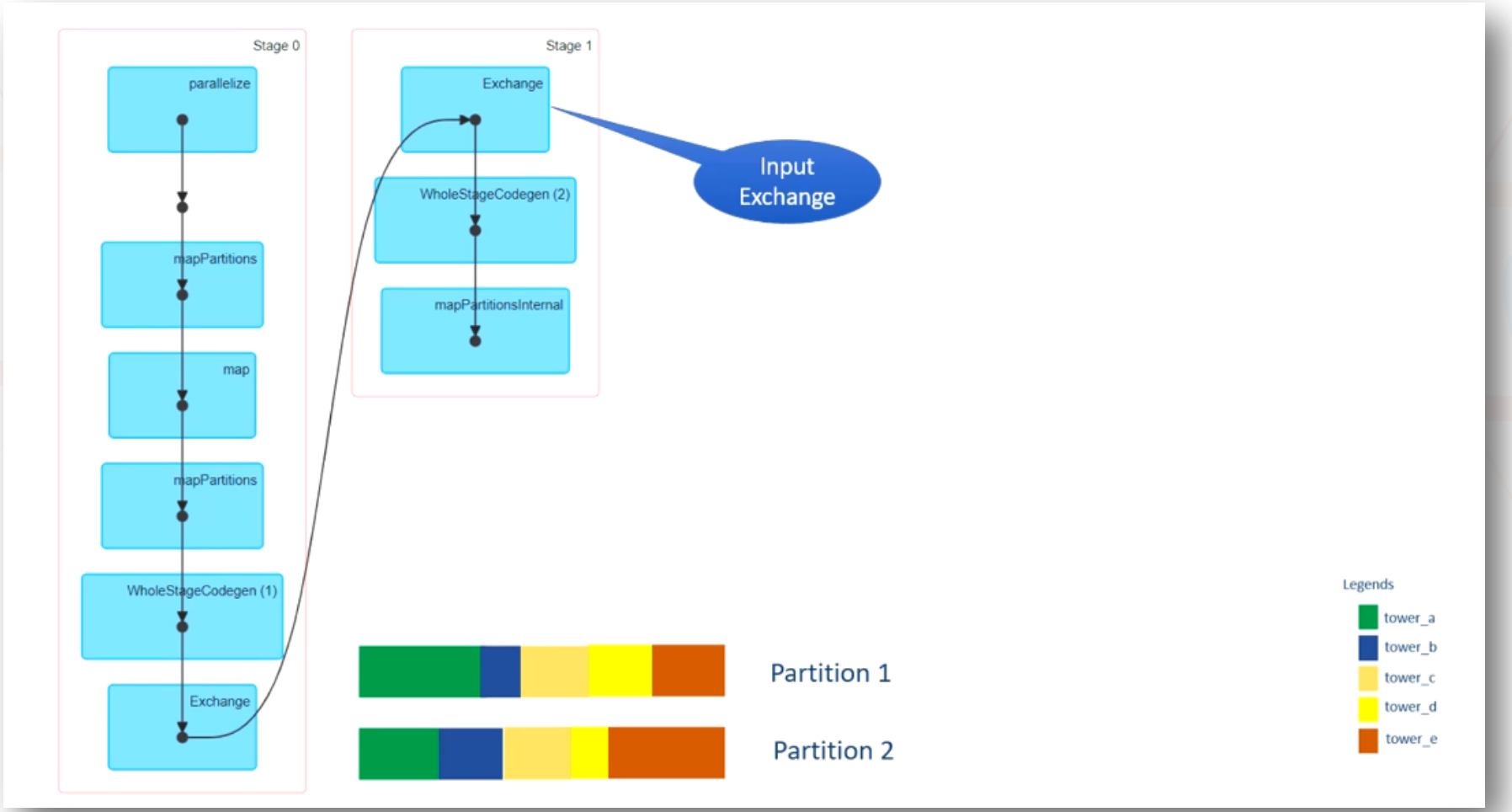
Spark will take the code, create an execution plan for the query, and execute it on the cluster.
The spark job that triggers this query should have a two-stage plan, which is similar to the one shown here at the right side. The actual plan for the large volume of data might not look the same. However, you will have two stages.



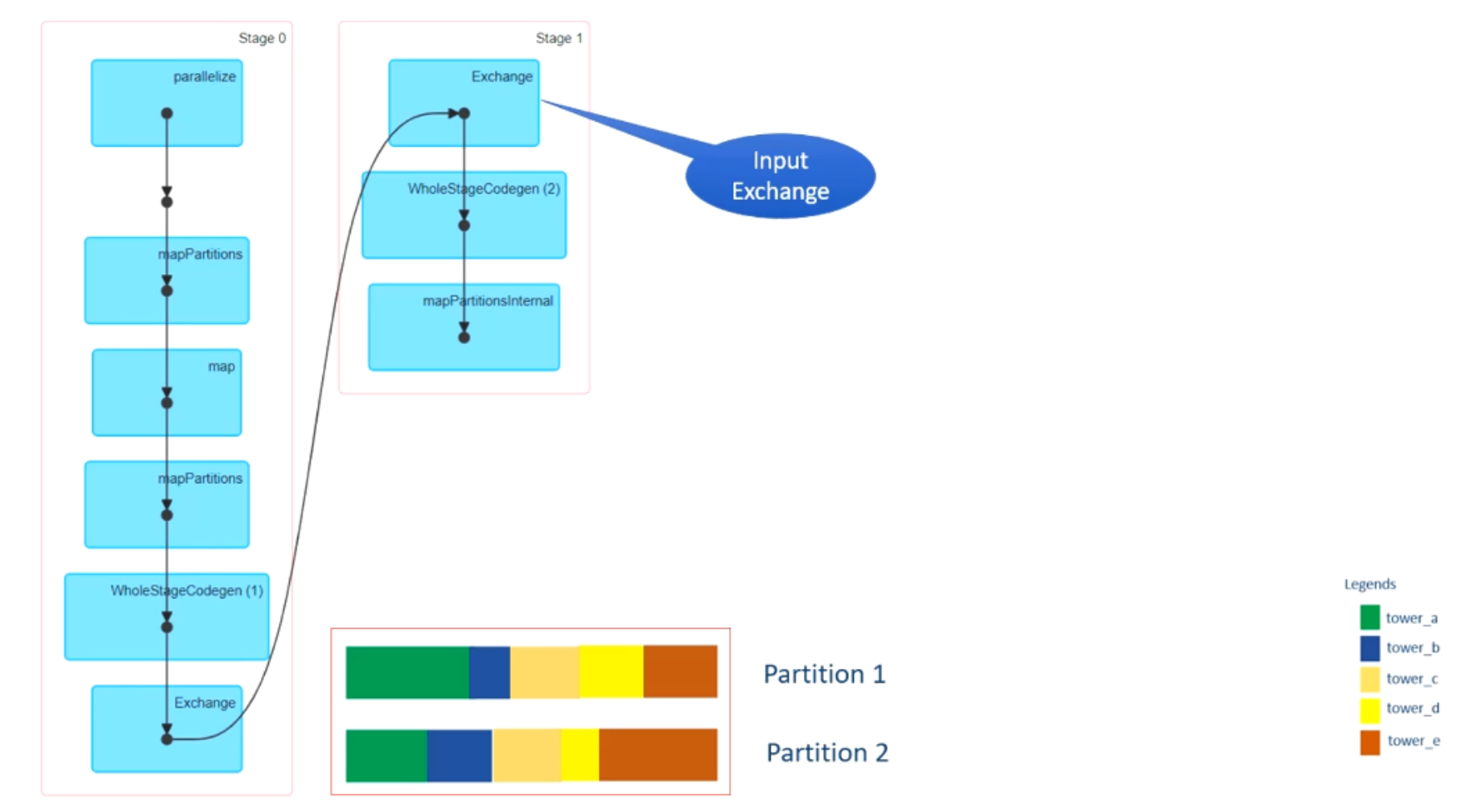
Stage zero reads the data from the source table and fills it to the output exchange.
The second stage will read the data from the output exchange and brings it to the input exchange. And this process is known as shuffle/sort.
Why do we have this shuffle/sort in our execution plan?
Because we are doing a groupBy operation and groupBy is a wide-dependency transformation.
So you are likely to have a shuffle/sort in your execution plan.
The stage 1 is dependent on stage zero, so stage one cannot start unless stage zero is complete.



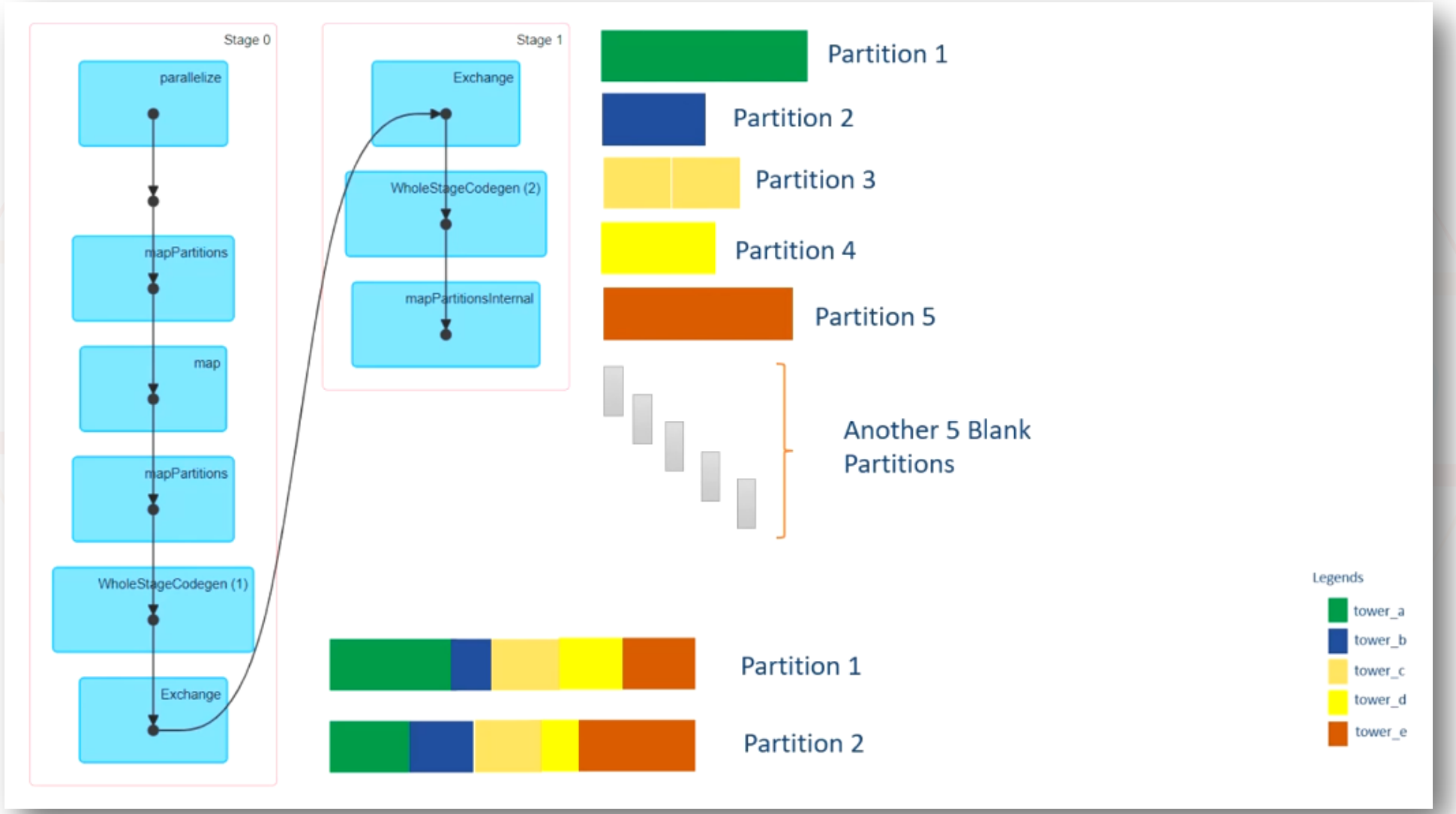
Let's look at the input exchange, which might look like this as shown below. I am assuming that my input data source has got only two partitions. So the stage zero exchange should have two partitions. I am also assuming that I have data for five towers. So each partition in this exchange might have some data for tower a, tower b, tower c, and so on.



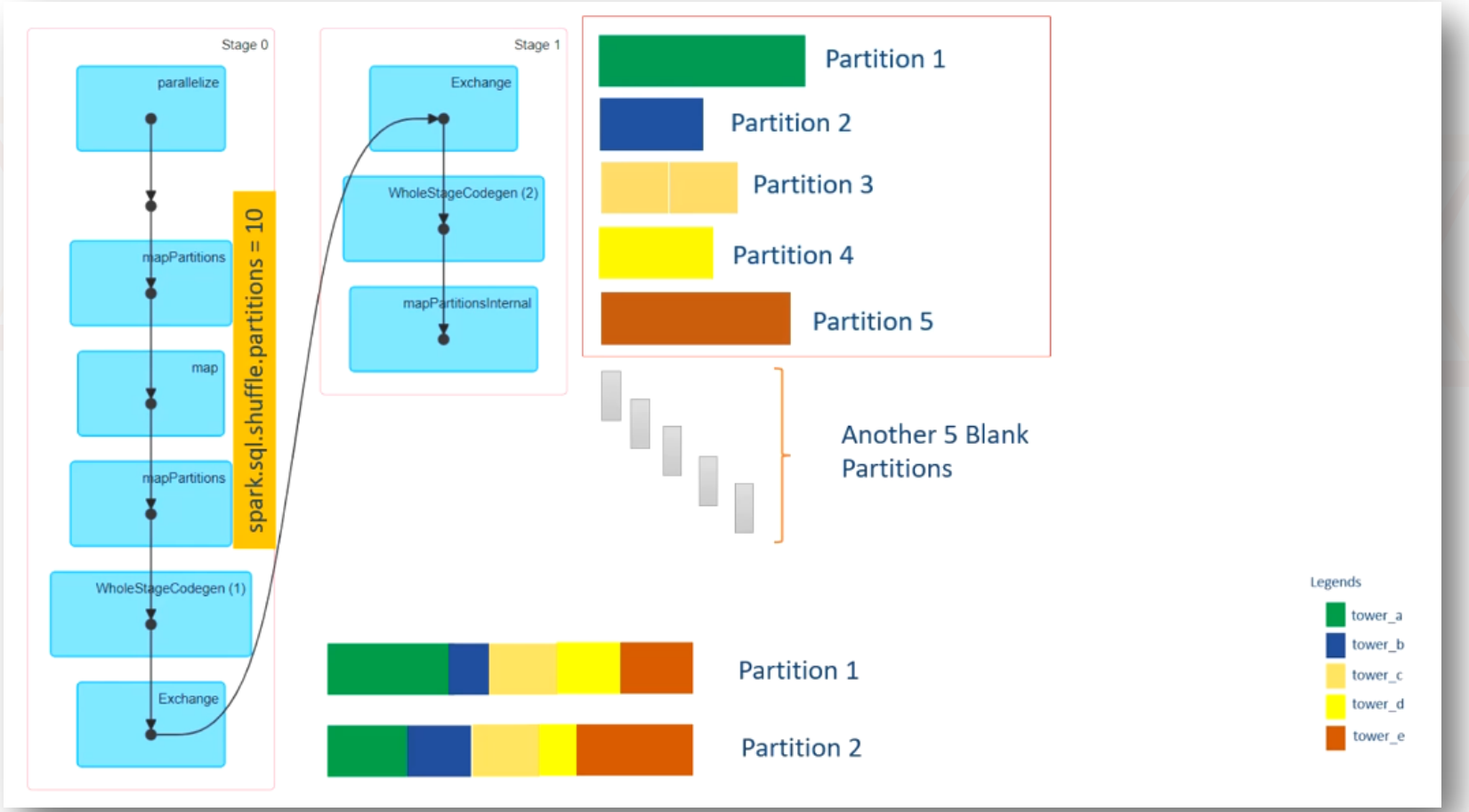
So, my stage zero will read data from the source table and make it available to the exchange for the second stage to read. I have only two partitions in my input source, so the exchange shows only two partitions here. Now the shuffle/sort operation will read these partitions, sort them by the tower_location and bring them to the input exchange of stage one.



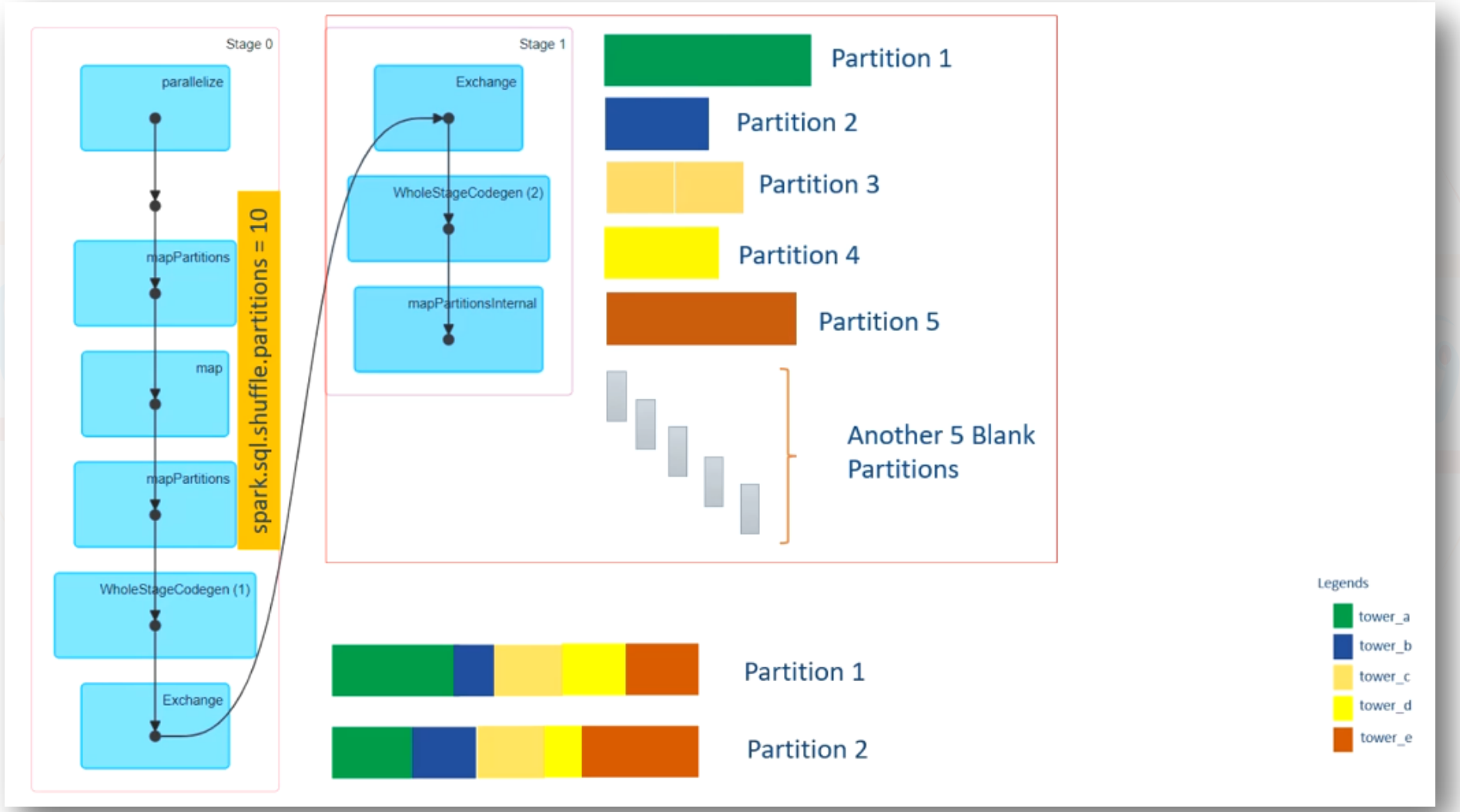
The final result after shuffle/sort should look like this shown below.
Now let us try to understand what is happening here.



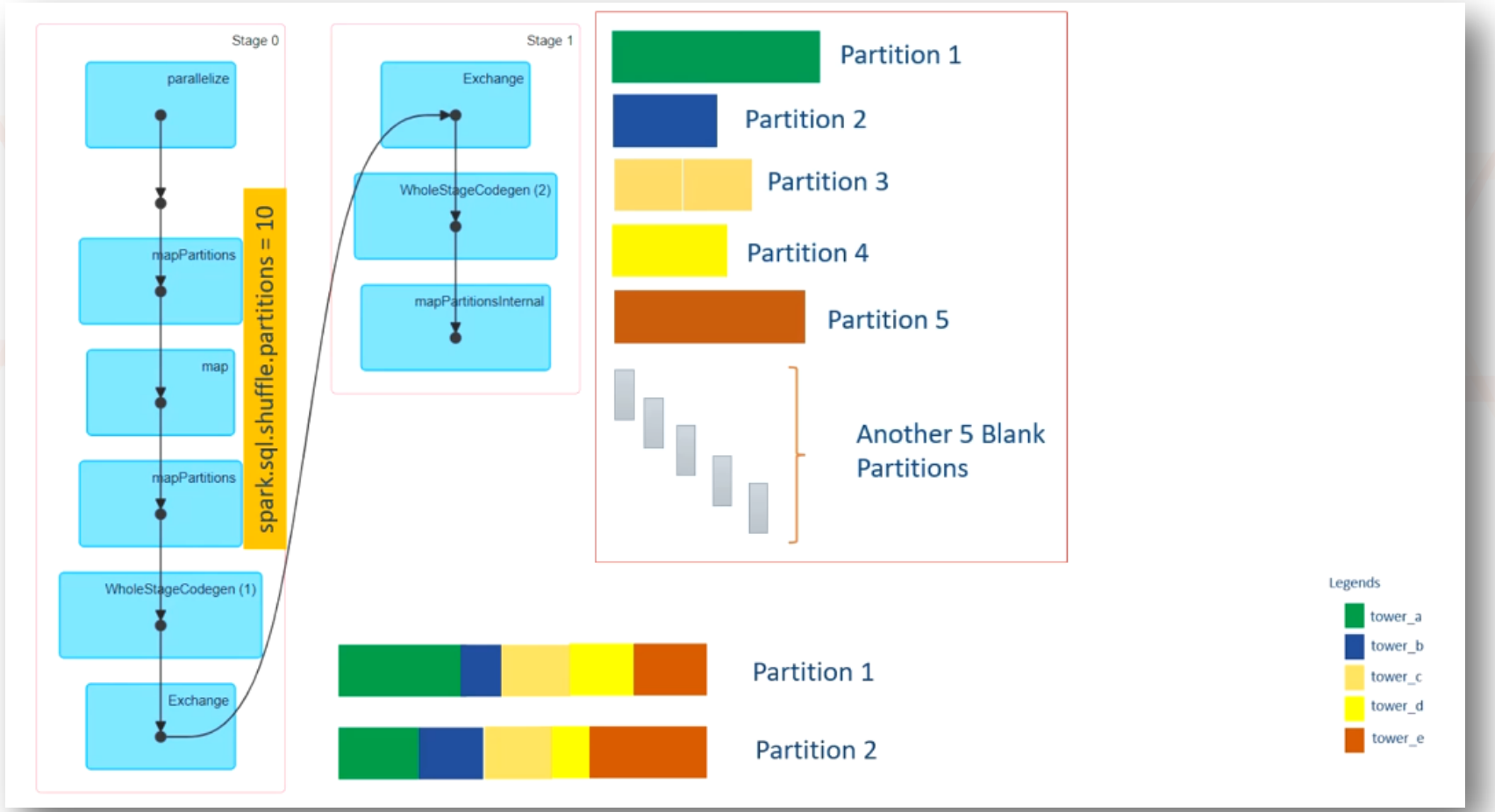
Let's assume I configured spark.sql.shuffle.partitions = 10. I know the default value for this configuration is 200, but I also know I do not have 200 unique towers. I am running a query to group by towers, so I reduced the number of shuffle partitions to 10. So the shuffle/sort will create ten partitions in the exchange. Even if I have only five unique values, the shuffle/sort will create ten partitions. Five of them will have data, and the remaining five will be blank partitions.



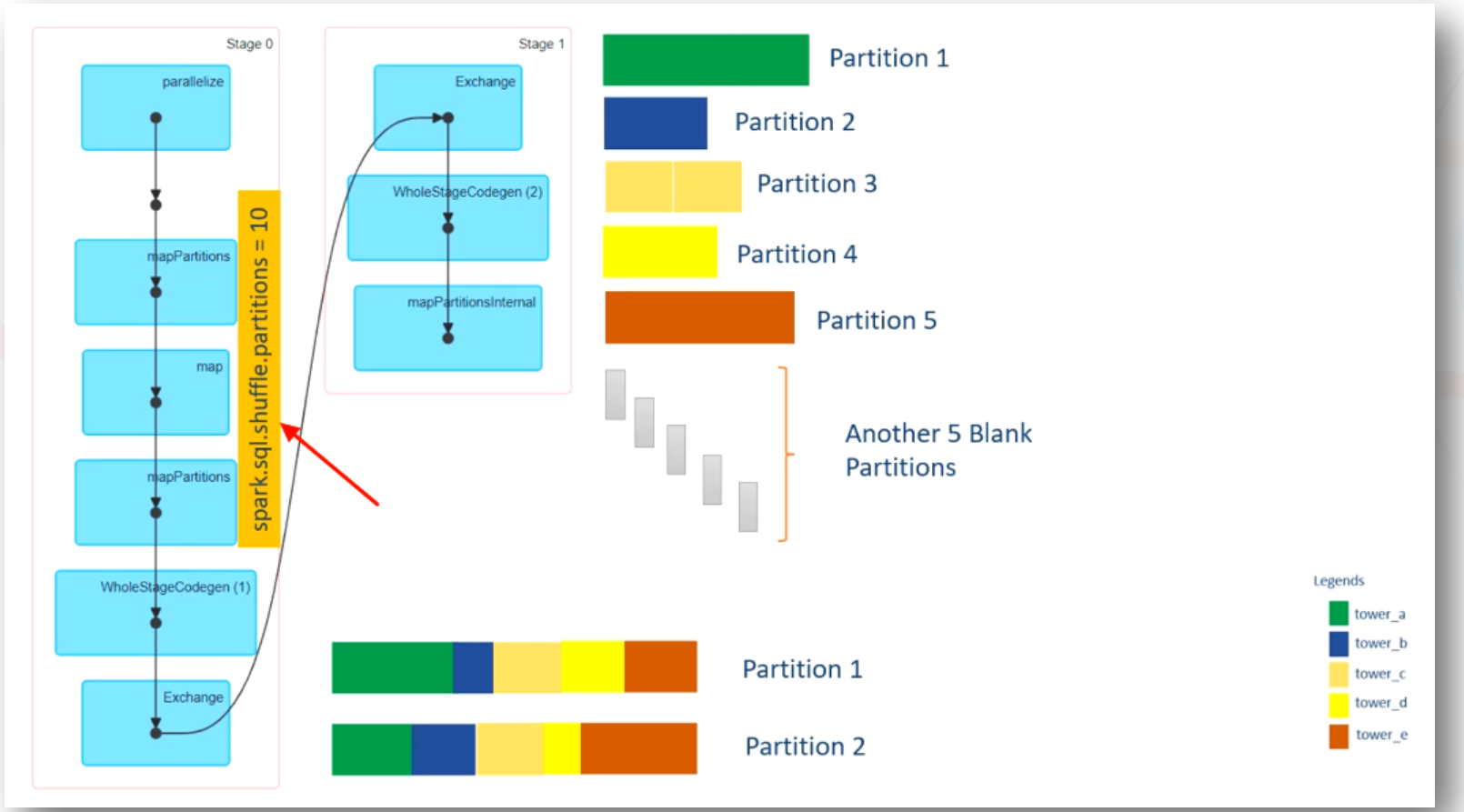
Now let me ask a question.
How many tasks do you need to execute this stage 1? Any guesses?



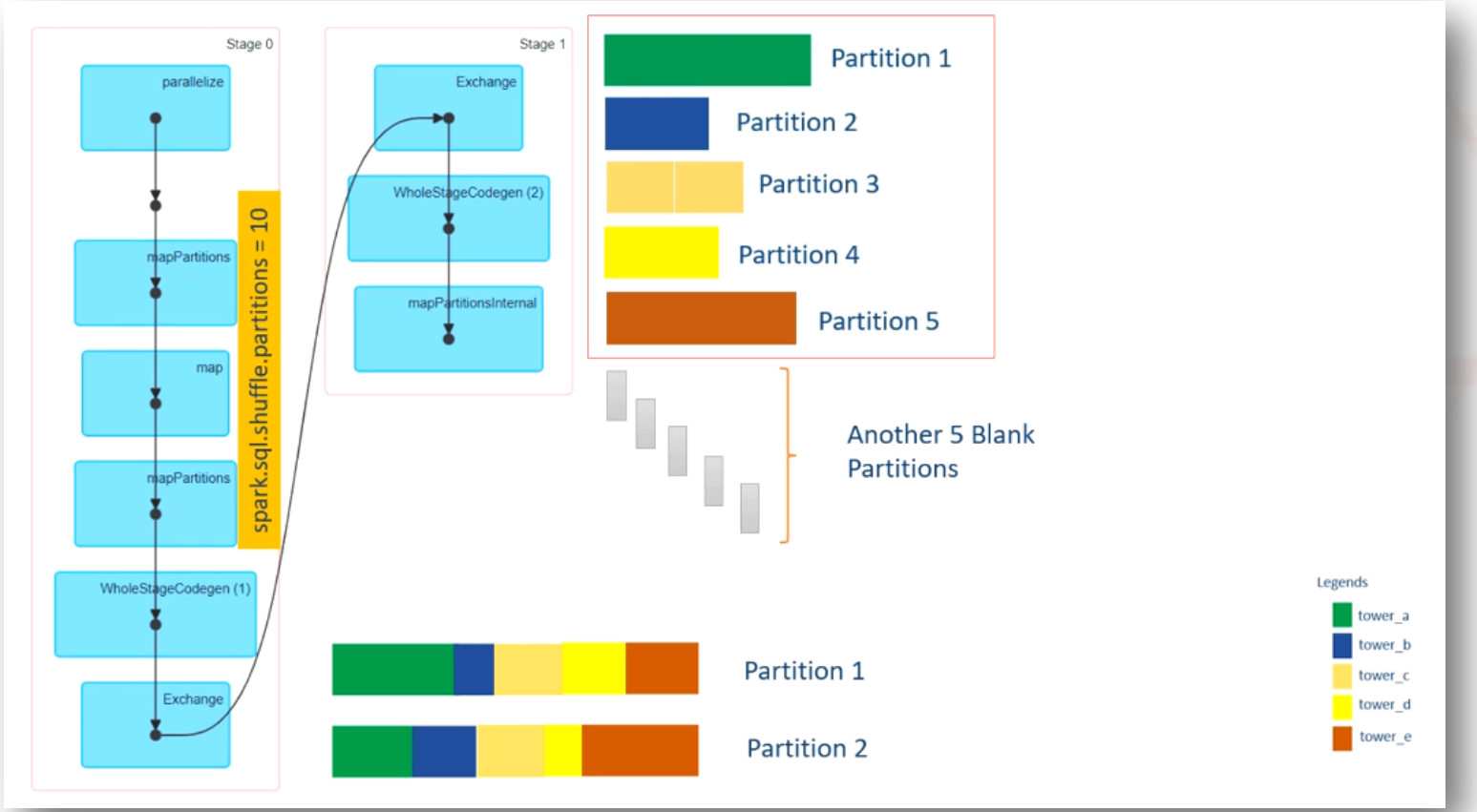
We need ten tasks. Why? Because we have ten partitions in the exchange. And that's a problem. Five partitions are empty, but Spark will still trigger ten tasks. The empty partition task will do nothing and finish in milliseconds. But Spark scheduler needs to spend time scheduling and monitoring these useless tasks. The overhead is small, but we do have some unnecessary overhead here.



I improved the situation by reducing the shuffle partitions to 10 from the default value of 200. But that's not an easy thing to do? Because I cannot keep changing the shuffle partitions for every query. Even if I want to do that, I do not know how many unique values my SQL will fetch? The number of unique keys is dynamic and depends on the dataset. How am I supposed to know how many shuffle partitions do I need? That's almost impossible for the developers to know in advance.

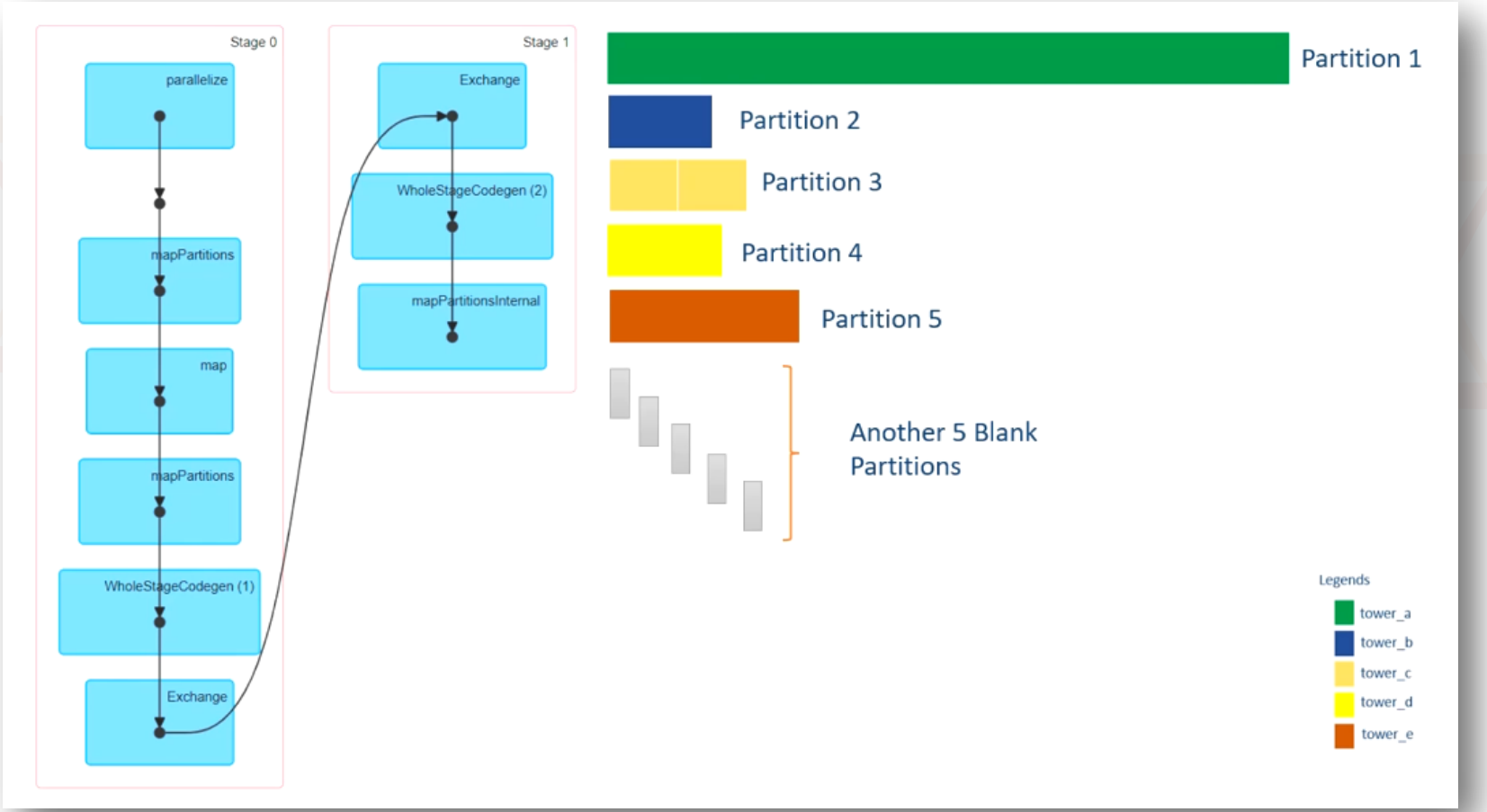


We have another problem here. Look at these partitions highlighted below. Some partitions are big, and others are small. So a task working on partition one will take a long time while the task doing the partition-2 will finish very quickly. And that's not good, because the stage is not complete until all the tasks of the stage are complete. Three tasks processing partitions 2,3 and 4 will finish quickly, but we still need to wait for partition-1 and partition -4. And that's a wastage of CPU resources.

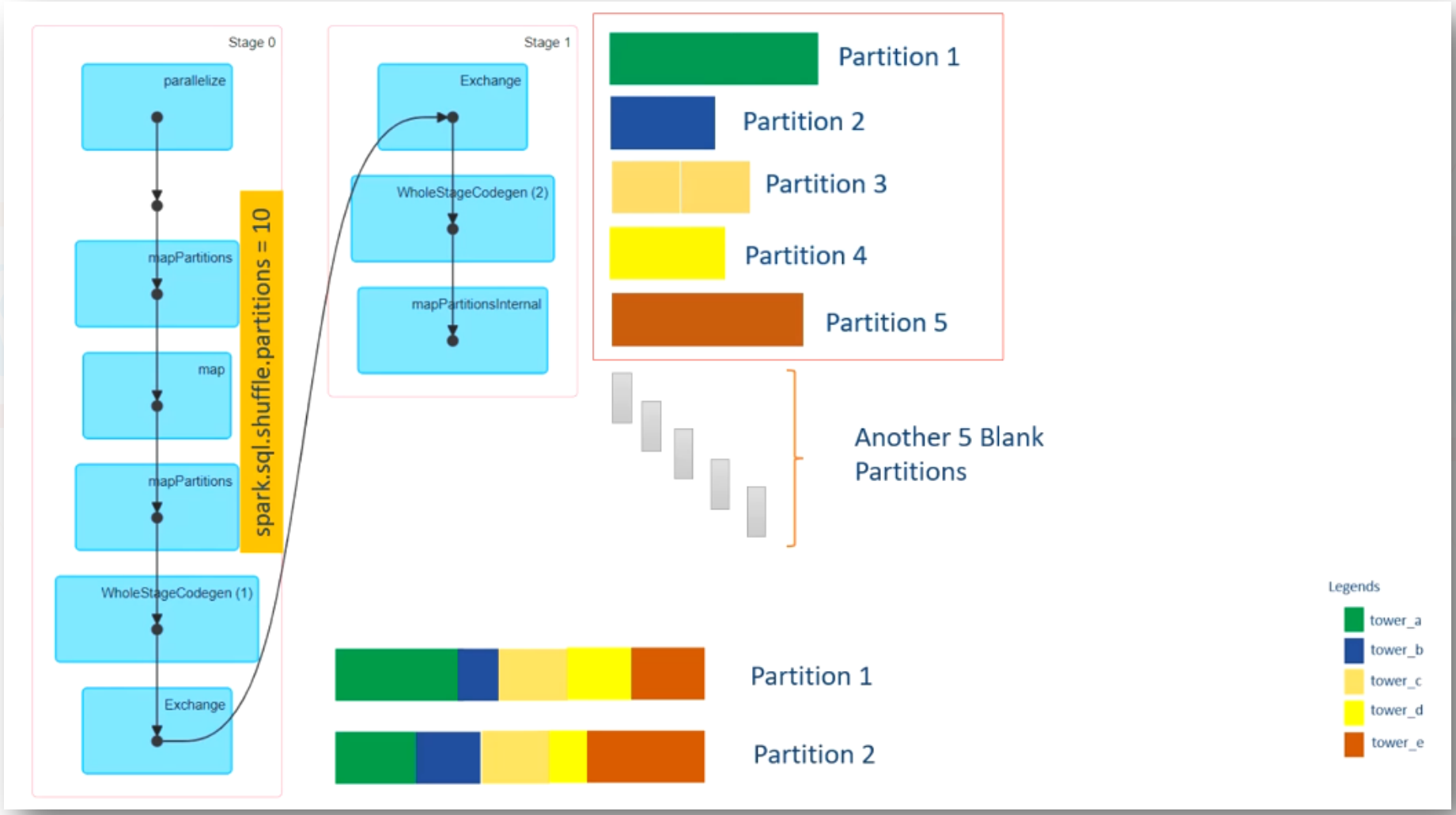


The situation becomes worse when one partition becomes excessively long. For example, look at this Partition 1 below.

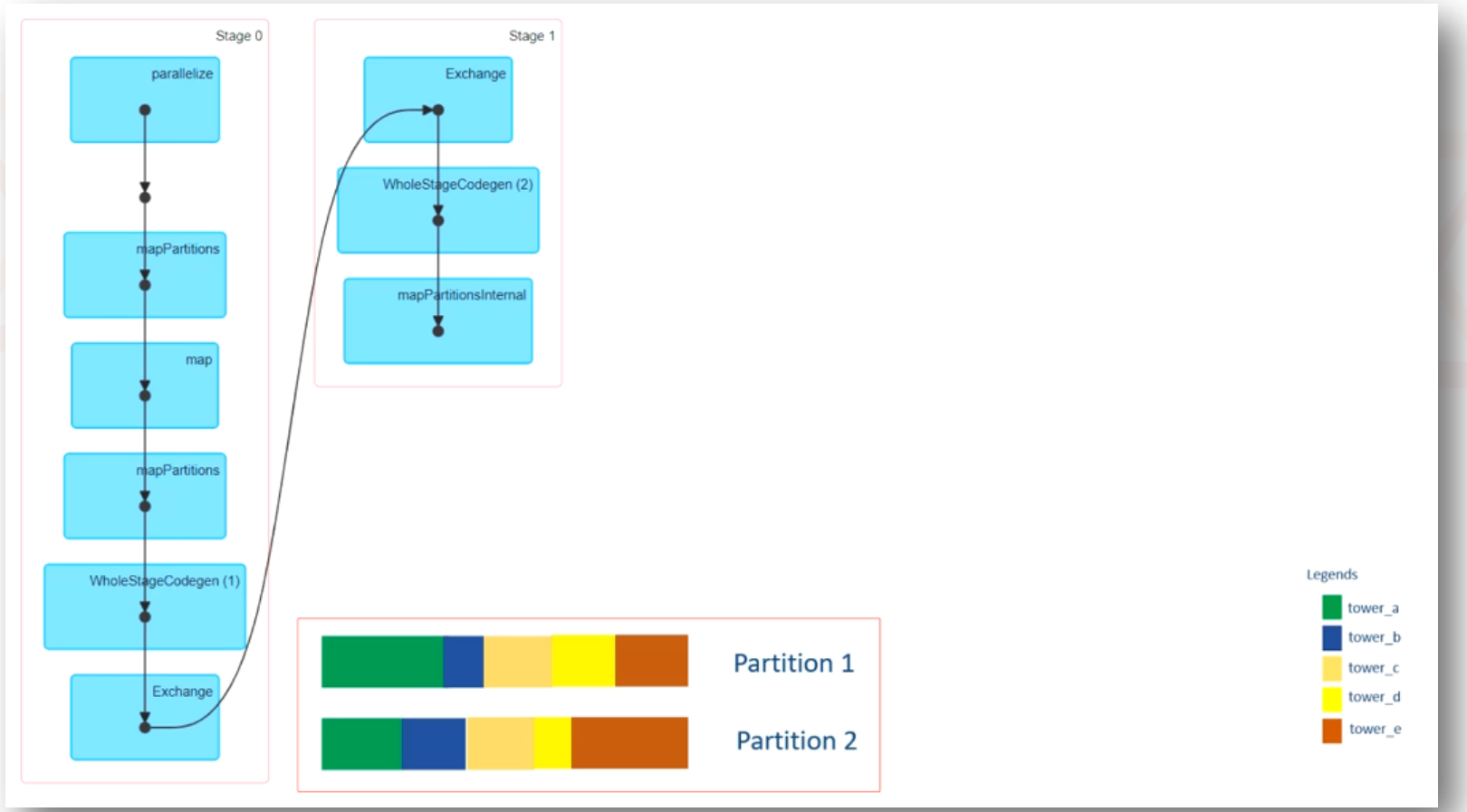
This diagram represents a data skew problem. In this case, one of your towers is processing many calls while others have smaller data sets. I will come back to the data skew in the following chapter.



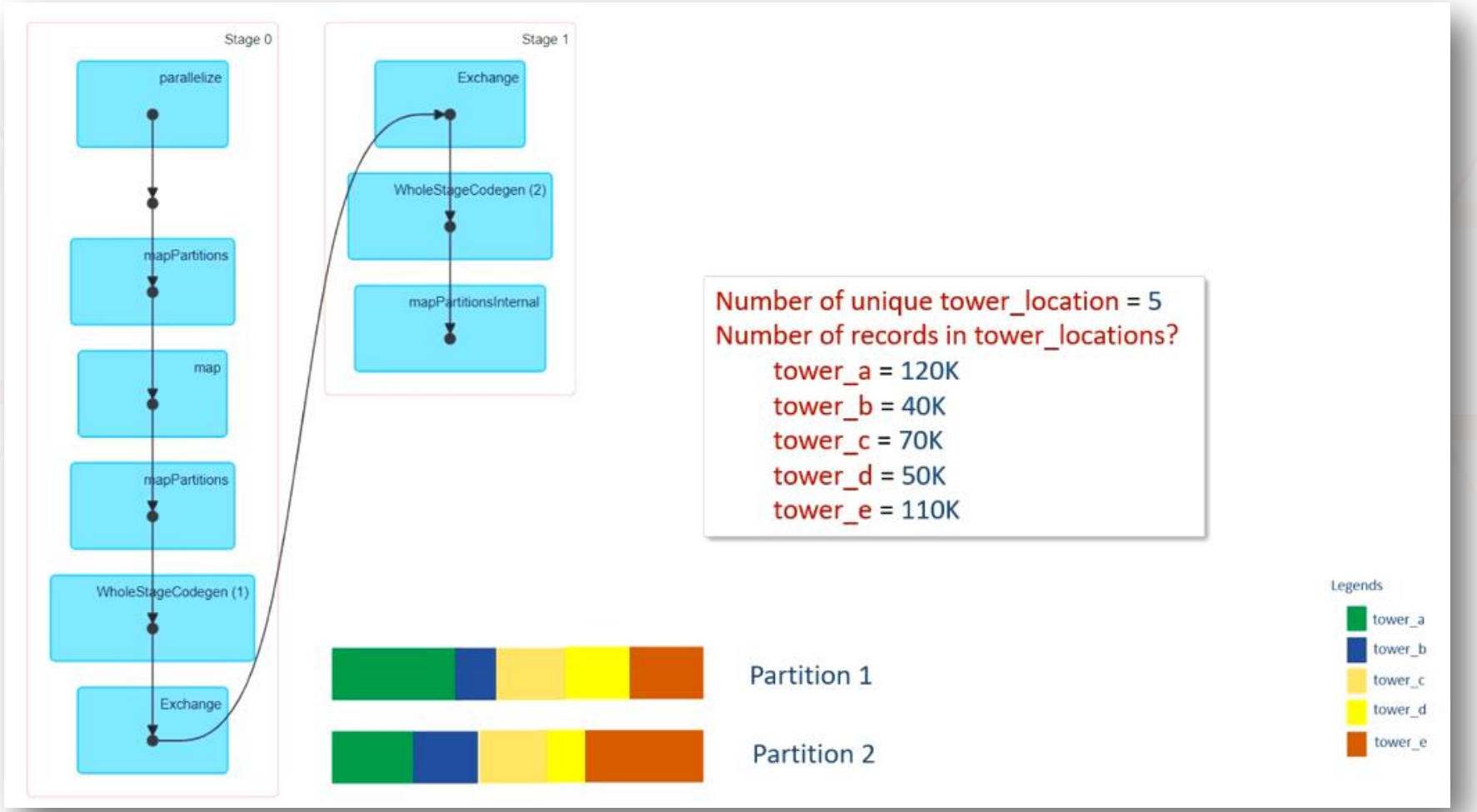
So come back to the shuffle partitions problem. My query resulted in five disproportionate shuffle partitions here. I made an intelligent guess and decided to reduce my shuffle partitions to ten. However, the situation is still not very good.



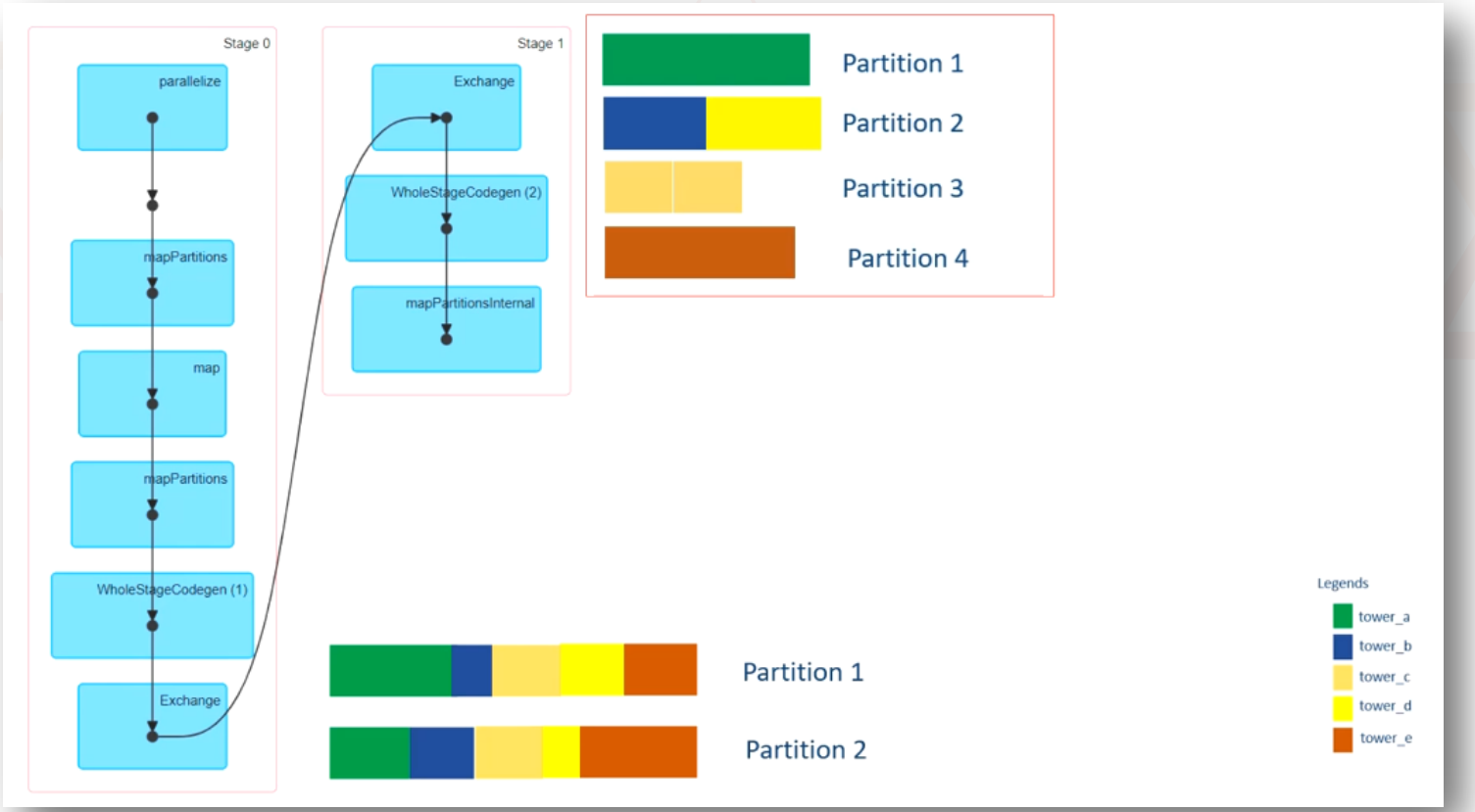
Now the question is this: How do I decide the number of shuffle partitions for my Spark job? Well, Spark 3.0 offers Adaptive Query Execution to solve this problem. You must enable it, and the AQE will take care of setting the number of your shuffle partitions. But how that magic happens? Super simple! Your input data is already loaded in stage zero exchange. Stage zero is already done, and data has come to the exchange. Now Spark will start the shuffle/sort.



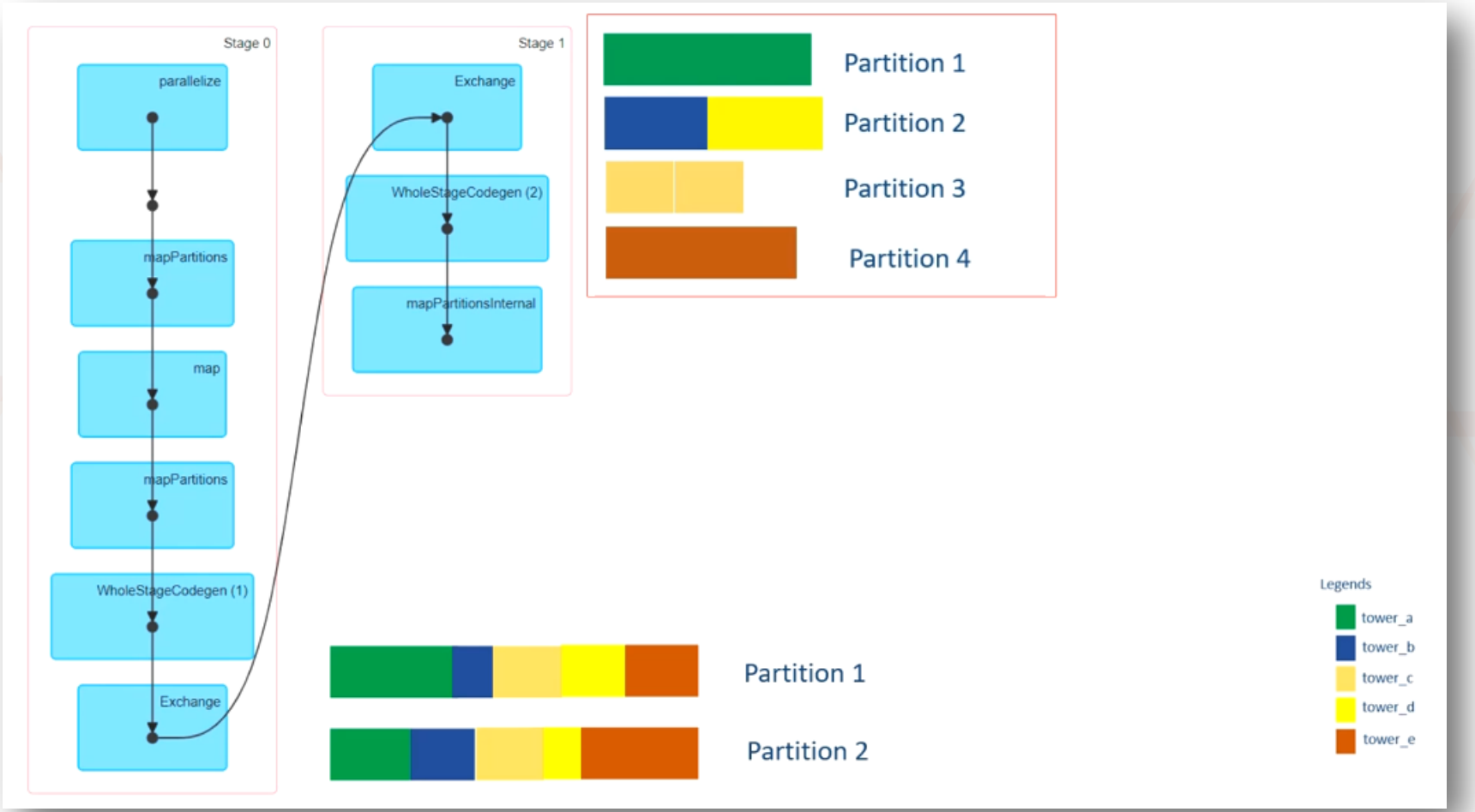
Once the shuffle/sort is started, it can compute the statistics on the given data and find out some details such as the following listed in the image below. And this is called dynamically computing the statistics on your data during the shuffle/sort. Such dynamic statistics are accurate and most up-to-date.



When Spark knows enough information about your data, it will dynamically adjust the number of shuffle partitions for the next stage. For example, in my case, Spark might dynamically set the shuffle partitions to four. And the result of that setting looks like this shown below. We have four shuffle partitions for stage one. Those five empty partitions are gone. Spark also merged two small partitions to create one larger partition. So instead of having five disproportionate partitions, we have four partitions. And these four are a little more proportionate.



Now let me ask you the same question: How many tasks do you need to run this stage? Any guess? Well, we need four tasks. Task 3 working on partition-3 will finish quickly, but the other three tasks will take almost equal time. So we saved one CPU slot, and we also eliminated the useless empty tasks.



I talked about three benefits of implementing adaptive-query-execution as follows:

1. Dynamically coalescing shuffle partitions
2. Dynamically switching join strategies
3. Dynamically optimizing skew joins

And I hope you now understand the first benefit of enabling adaptive-query-execution.
Let me quickly summarize it for you.

Spark shuffle/sort has a critical impact on Spark query performance. One fundamental tuning property of shuffle operation is the number of output partitions. You can set this number using *spark.sql.shuffle.partitions*. However, it is almost impossible to know the best number because it depends on your data size and other factors. So, we have two problems here:

1. If we configure a small number of partitions, then the data size of each partition may be very large. A large shuffle partition may cause two types of problems:
   a) Your task may need a lot of memory to process the partition, and it might slow down juggling data between memory and disk.
   b) In the worst case, you may also see an OOM exception.

2. What if you have too many partitions?
   a) The data size of each partition may be very small, and there will be a lot of small network data fetch to read the shuffle blocks.
   b) Your query performs slowly because of the inefficient network I/O.
   c) A large number of partitions will also result in many tasks and puts more burden on the Spark task scheduler.

To solve this problem, we can set a relatively large number of shuffle partitions at the beginning and enable Adaptive query execution.
The AQE feature of Apache Spark will compute shuffle file statistics at runtime and perform two things:

1. Determine the best shuffle partition number for you and set it for the next stage.
2. Combine or coalesce the small partitions into bigger partitions to achieve even data distribution amongst the task.

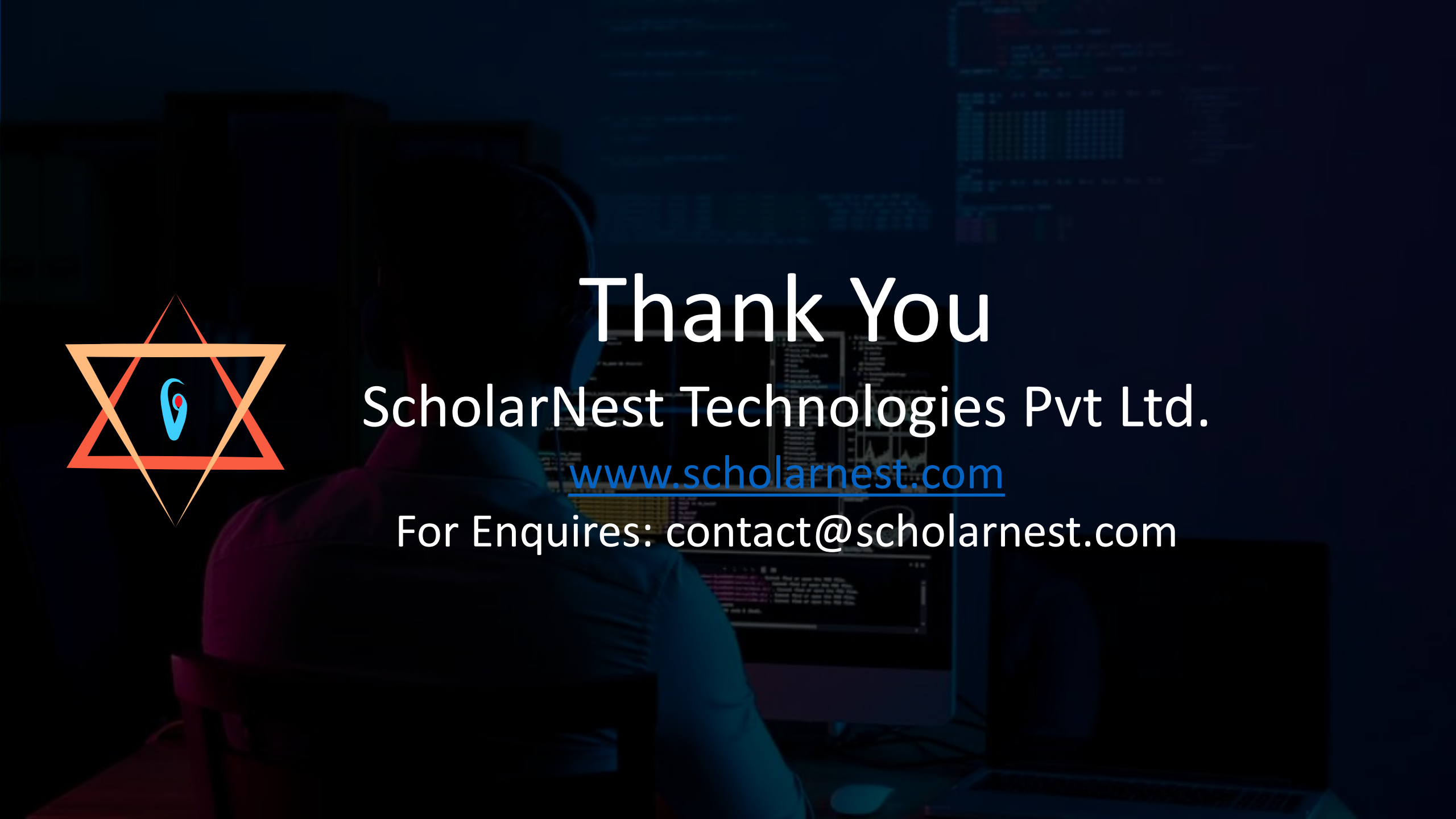
You can enable AQE using the following configuration:

1. **spark.sql.adaptive.enabled = true** : The default value of this configuration is false. You have four additional configurations to tune the AQE behavior.
2. **spark.sql.adaptive.coalescePartitions.initialPartitionNum** : The first configuration sets the initial number of shuffle partitions. Dynamically determining the best number and setting that value is applied later, AQE starts with this value. This configuration works as the max number of shuffle partitions. The Spark AQE cannot set a number larger than this. This configuration doesn't have a default value. So if you do not set this configuration, Spark will set it equal to spark.sql.shuffle.partitions.
3. **spark.sql.adaptive.coalescePartitions.minPartitionNum** : This configuration defines the minimum number of shuffle partitions after coalescing or combining multiple partitions. We do not have a default value for this configuration also. So if you do not set this configuration, Spark will set it equals to spark.default.parallelism.

So we learned about how to control the maximum and minimum shuffle partitions for AQE.

4. **spark.sql.adaptive.advisoryPartitionSizeInBytes** : The default value for this configuration is 64 MB, and it works as an advisory size of the shuffle partition during adaptive optimization. So this configuration takes effect when Spark coalesces small shuffle partitions or splits skewed shuffle partition. I will talk about splitting the skewed partitions in the following video. So, for now, you can think of this number as an advisory to AQE. The AQE will use this number for determining the size of the partitions and combine them accordingly.
5. **spark.sql.adaptive.coalescePartitions.enabled** : The default value of this configuration is true. If you set this value to false, Spark AQE will not combine or coalesce smaller partitions.

So you have five configurations.
The first one is a kind of master configuration to enable or disable the AQE feature.
If you disable it, the other four configurations do not take any effect.
But if you enabled AQE, you can tune it further using the other four configurations.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Joins and Optimization

Lecture:
AQE Dynamic Join Optimization

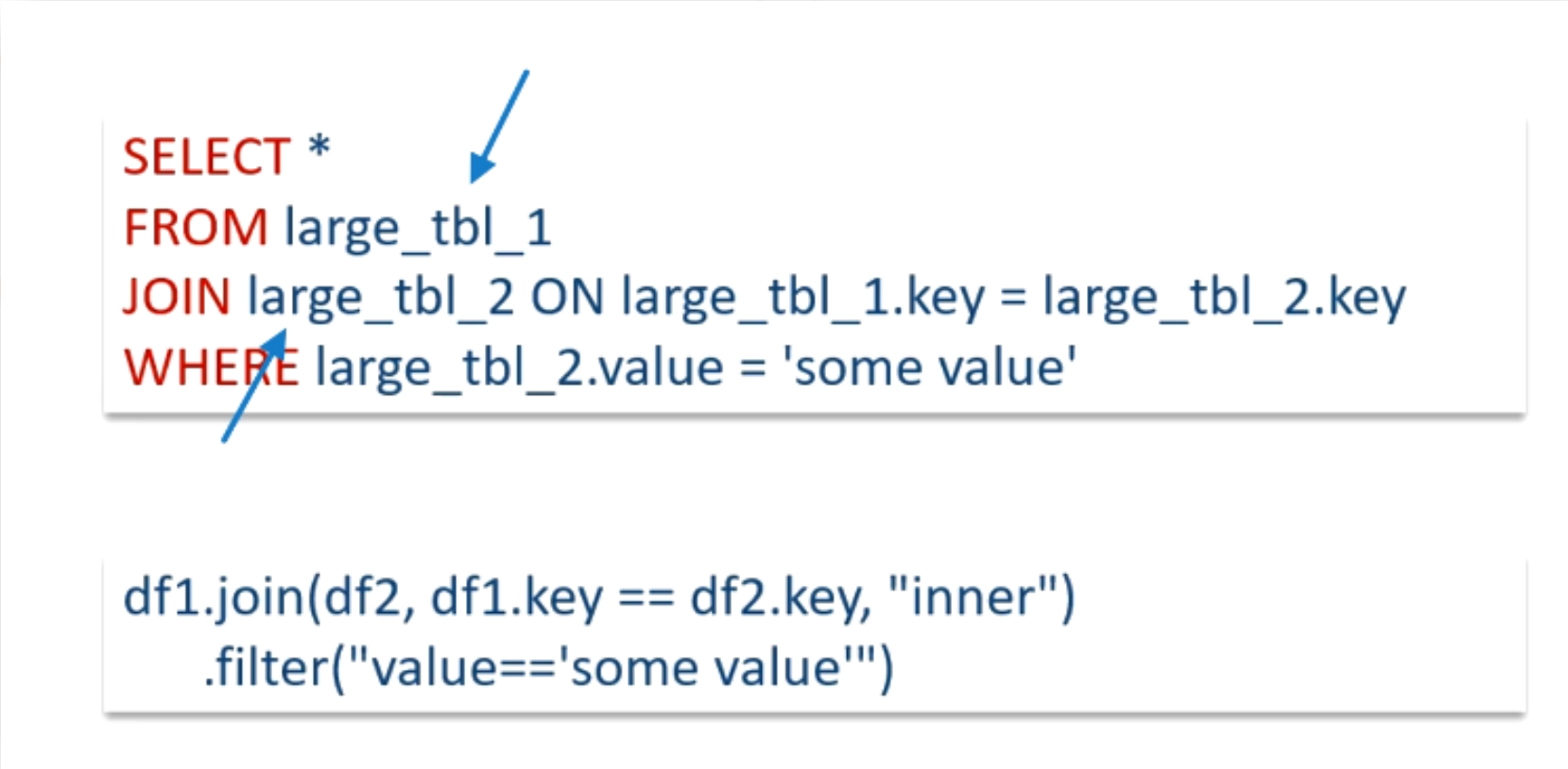
Spark AQE Dynamic Join Optimization

I talked about the following three features of Spark Adaptive Query Execution:

1. Dynamically coalescing shuffle partitions
2. Dynamically switching join strategies
3. Dynamically optimizing skew joins

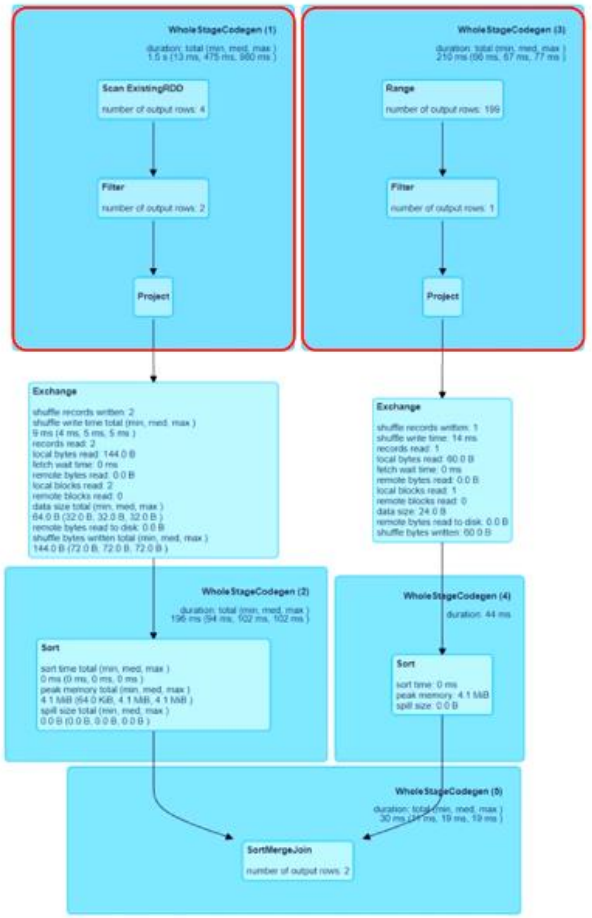
I also covered the Dynamically coalescing shuffle partitions.
In this lecture, I will talk about the Dynamically switching join strategies.
So let's start understanding the problem first. Why do we need dynamic join optimization?

Let's assume you have two large tables.
And you are joining these two tables using the following query.
I am showing a Spark SQL here, but you might be doing it using an equivalent Dataframe expression as shown in the bottom section of the screenshot.
Both are the Spark SQL and the Dataframe expression are the same.
Both of your tables are large tables, so you are expecting a sort-merge join to take place.



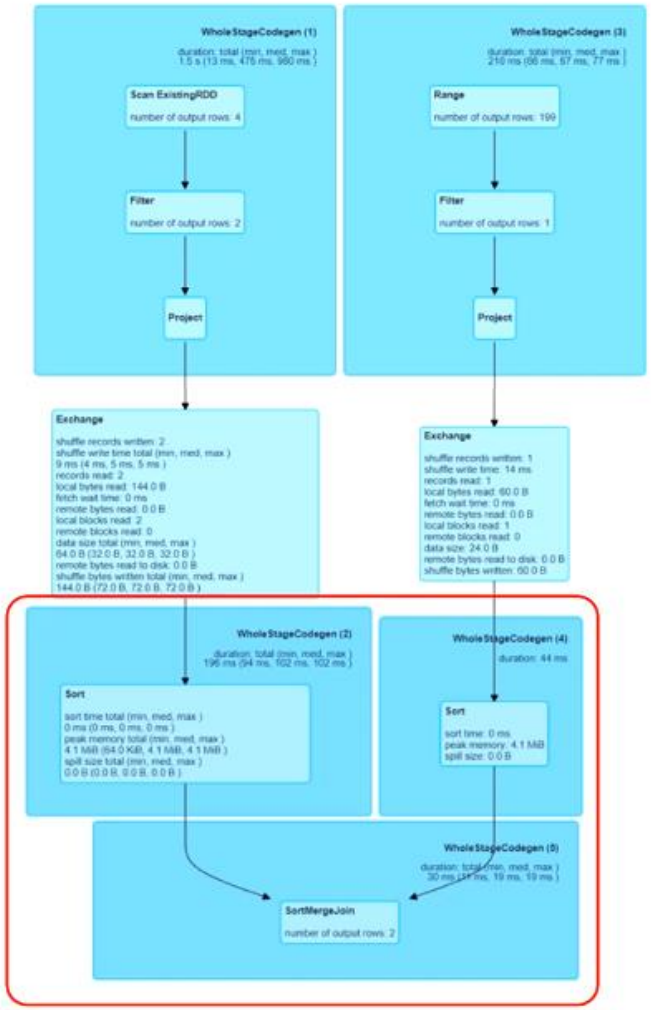
You ran your job and checked the execution plan. Here it is shown below. So what do you see? I am reading two tables, and I have two stages of reading those tables. So here is my first stage(one box on the top left side), and this one is the second stage (second box in the top right corner). Both these stages send data to exchange.

```
SELECT *
FROM large_tbl_1
JOIN large_tbl_2 ON large_tbl_1.key = large_tbl_2.key
WHERE large_tbl_2.value = 'some value'
```



And everything after the exchange is part of the third stage.
So my third stage collects the data from the exchange, performs the sorting operation on the data, and finally joins them. And that's what happens in a sort-merge join operation.

```
SELECT *
FROM large_tbl_1
JOIN large_tbl_2 ON large_tbl_1.key = large_tbl_2.key
WHERE large_tbl_2.value = 'some value'
```



Do you see any problems or opportunities to optimize this operation?
Let us try to understand. I am assuming both of my tables are large enough. But I am also applying a filter condition on the large_tbl_2 as shown below. The large table_2 is a 100GB table, but what is the effect of applying a filter? How many rows am I selecting post filter? You did some investigation and realized that you selected only 7 MB of data from the large table_2.

```
SELECT *
FROM large_tbl_1
JOIN large_tbl_2 ON large_tbl_1.key = large_tbl_2.key
WHERE large_tbl_2.value = 'some value'
```


Now think about the situation once again. Your large_table_1 is 100 GB, and you are selecting all rows from this table. Your large_tbl_2 is also 100 GB, but you are selecting only 7 MB from that table. If you know this information already, will you apply for a sort-merge join? No! I want to use broadcast hash join here because one of my tables is small enough.
I mean, the table is large, but I am filtering records and selecting only 7 MB of data.
It makes more sense to use broadcast join and avoid the shuffle/sort operation.
But why isn't it happening? Well, Spark will not apply Broadcast hash join if Broadcast Join Threshold is broken. So you decided to check the following configuration:

spark.sql.autoBroadcastJoinThreshold = 10MB

But the value is 10MB, and that's the default value. You haven't changed it.
It means you are selecting 7 MB from a large table, and that's well below the broadcast threshold.
But Spark is not applying the broadcast join.

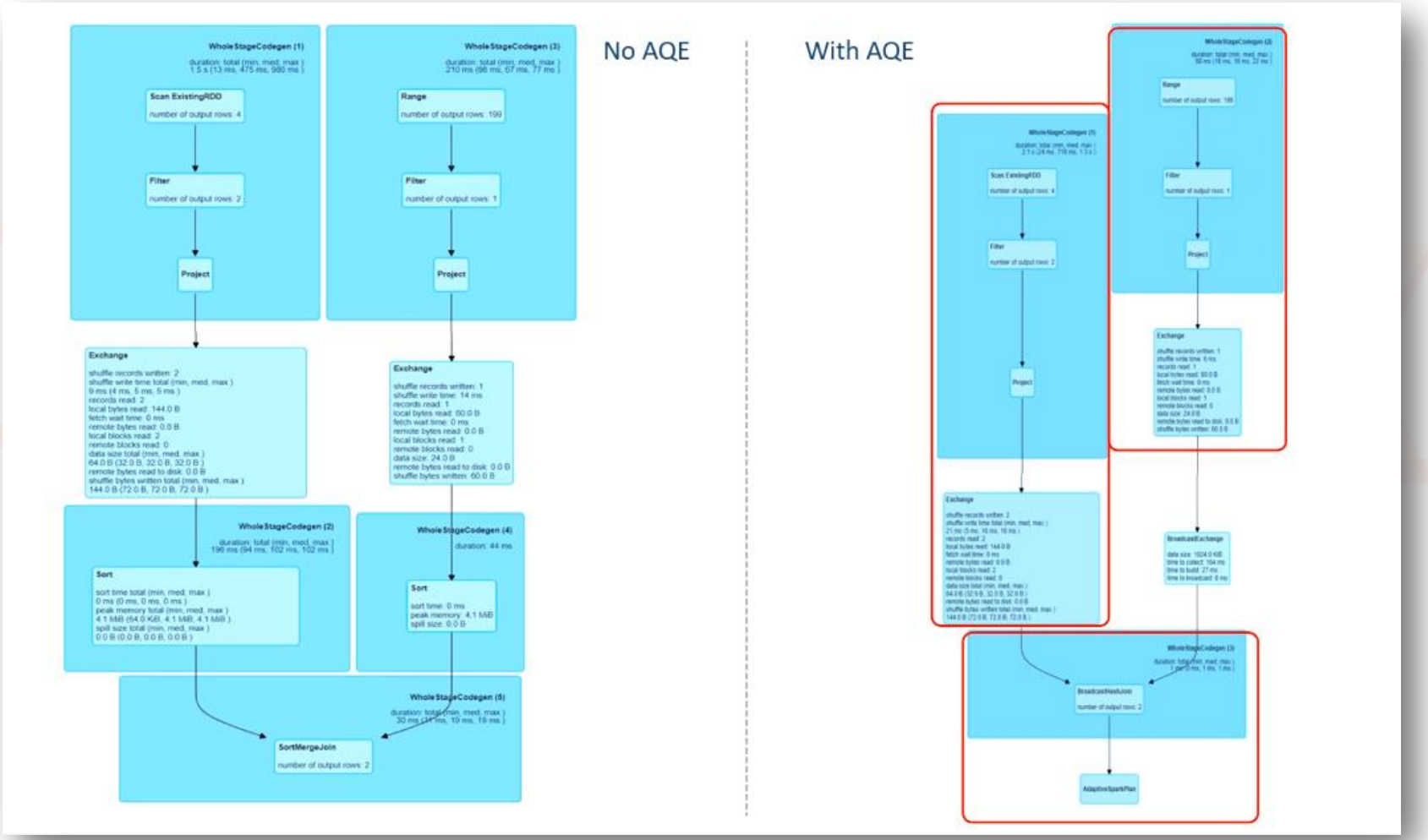
Spark is not applying broadcast join because the Spark execution plan is created before the spark starts the job execution. Spark doesn't know the size of the table, so it applied a sort-merge join.
Okay, so if I compute statistics on the table, will spark apply the broadcast join? Well, It may or may not.
Spark will not apply broadcast if you do not have a column histogram for the filter column.
It cannot apply broadcast join if your statistics are outdated.
So one solution is to analyze your Spark tables and keep your table and column statistics up to date.
Or another solution is to enable AQE.
Spark Adaptive Query Execution can help you in this situation.
We already learned that the AQE computes statistics on the shuffle data.

So AQE will compute the statistics on the shuffle data and use that information to do the following things:

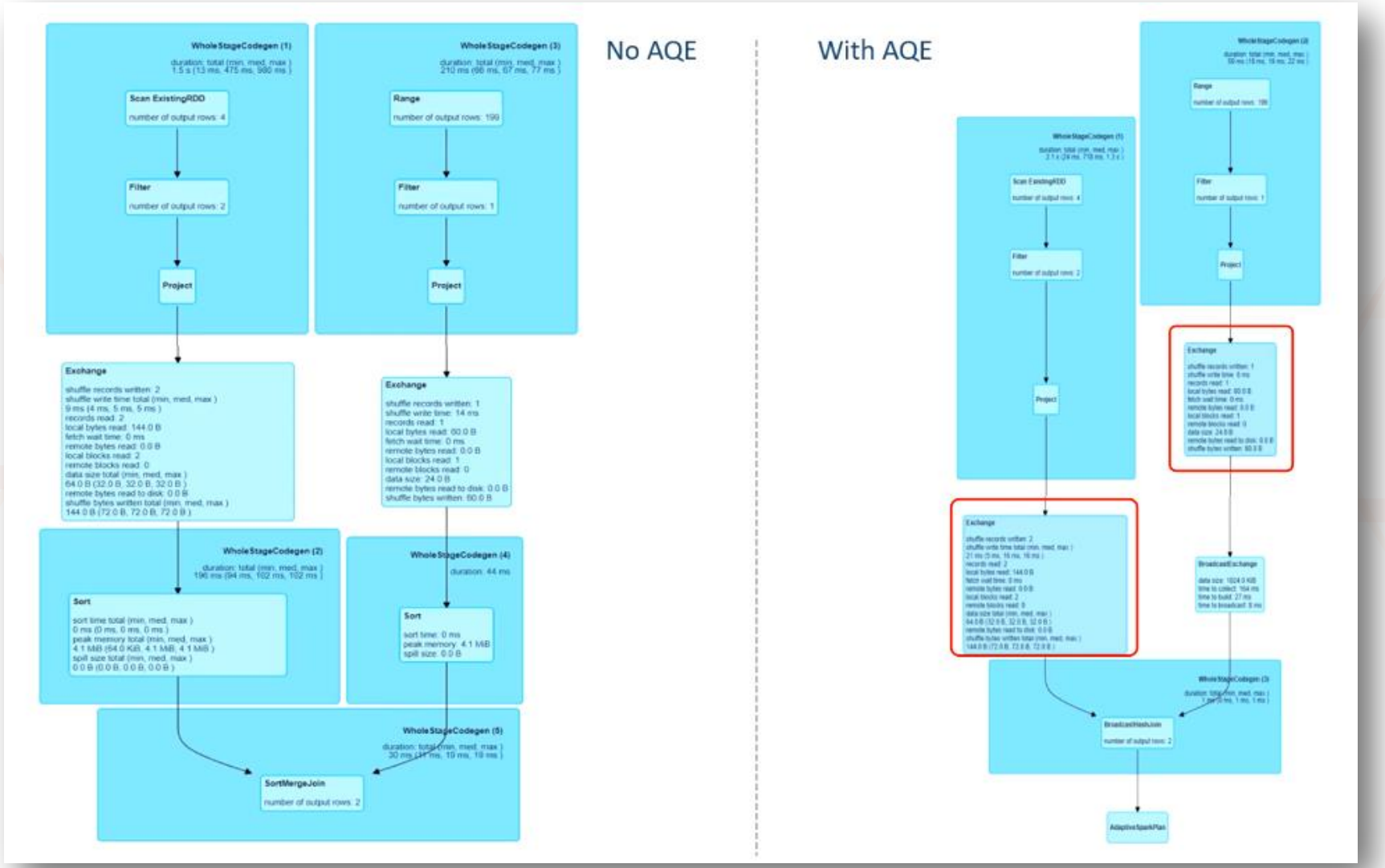
1. Dynamically coalesce the shuffle partitions
2. Dynamically change the Join strategy
3. Dynamically optimizing skew joins

We already learned the dynamic coalescing of shuffle partitions in the earlier lecture.
In this lecture, let's see how AQE changes the join strategy.

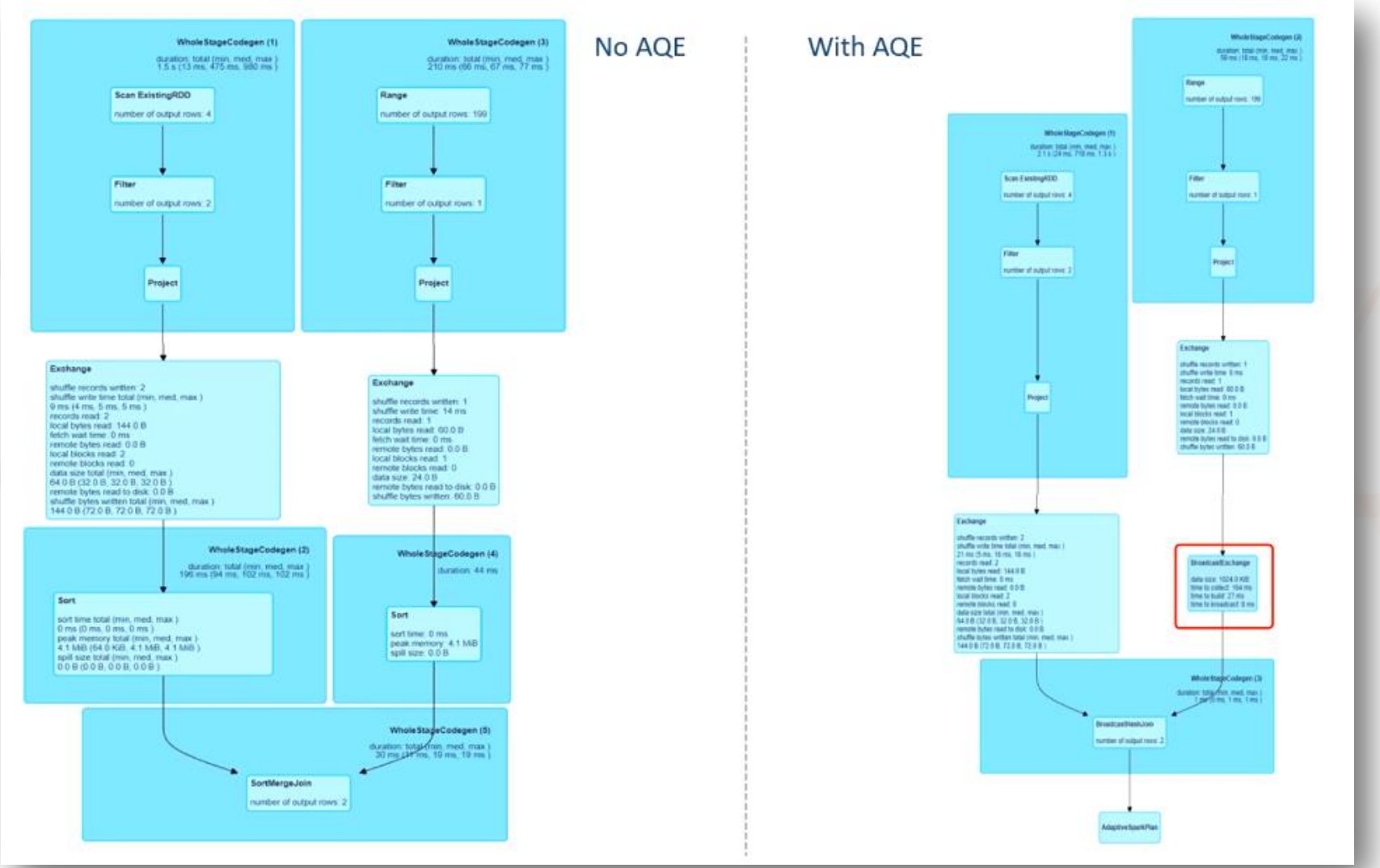
I enabled AQE and executed the same query to check the new execution plan which is shown below. We still have three stages when we are using AQE.
Stage one and Stage two are scanning the tables and sending data to exchange.



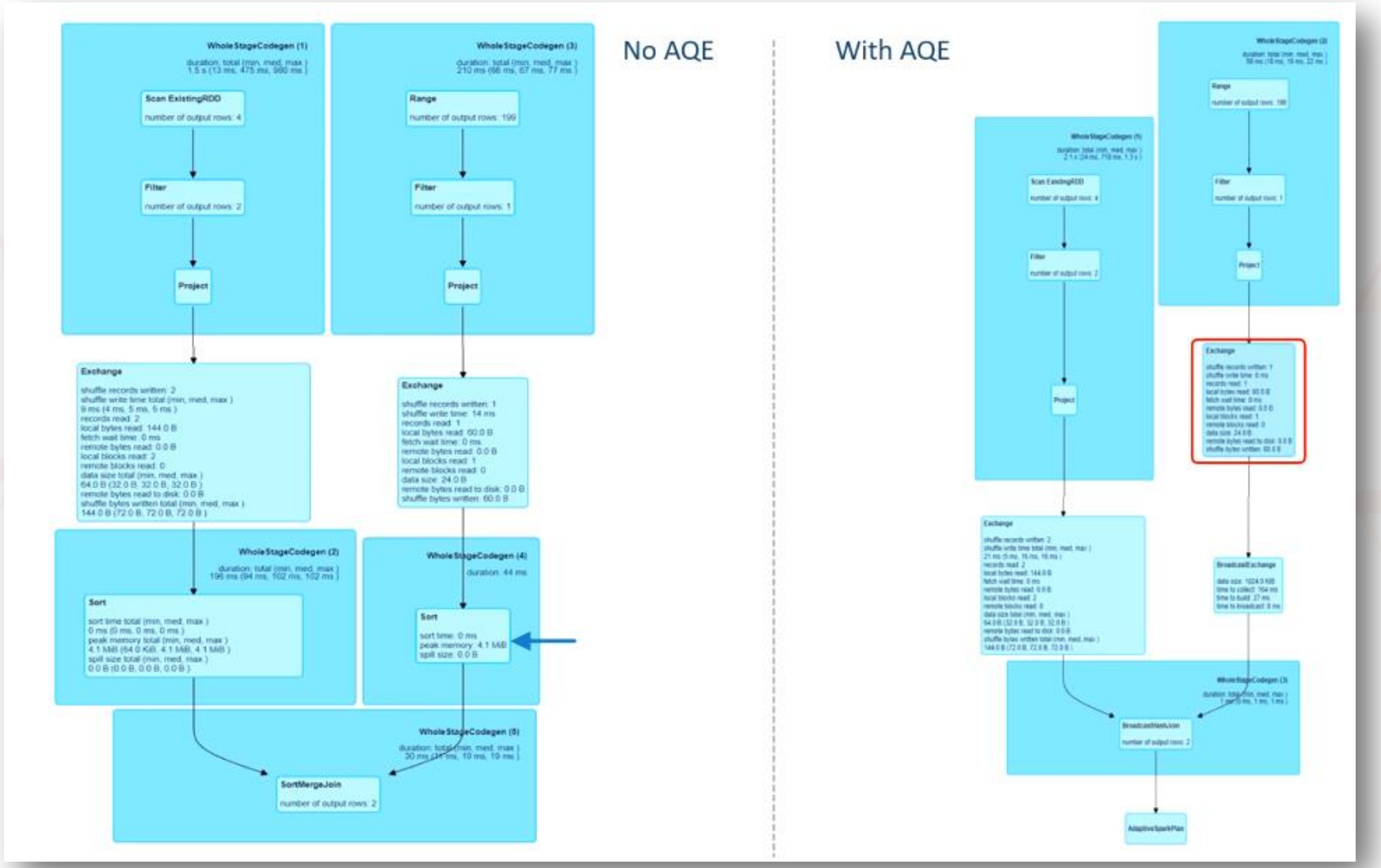
But we enabled AQE. So the Adaptive Query will compute statistics on the exchange data. The statistics tell that the data size of large_tbl_2 is small enough to apply broadcast join.



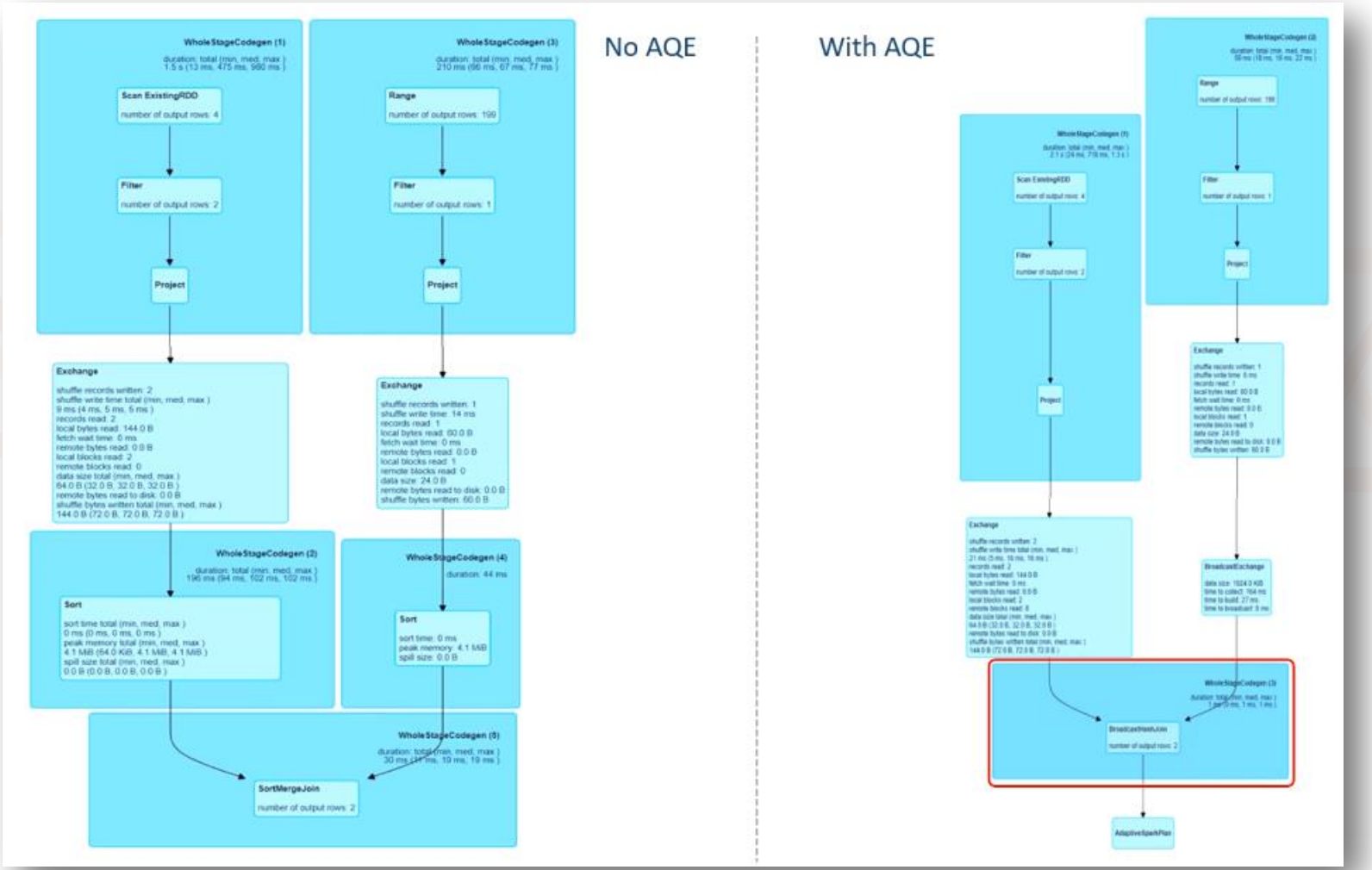
So the AQE will dynamically change the execution plan and apply broadcast hash join.
And you can see that in the new query plan highlighted below.



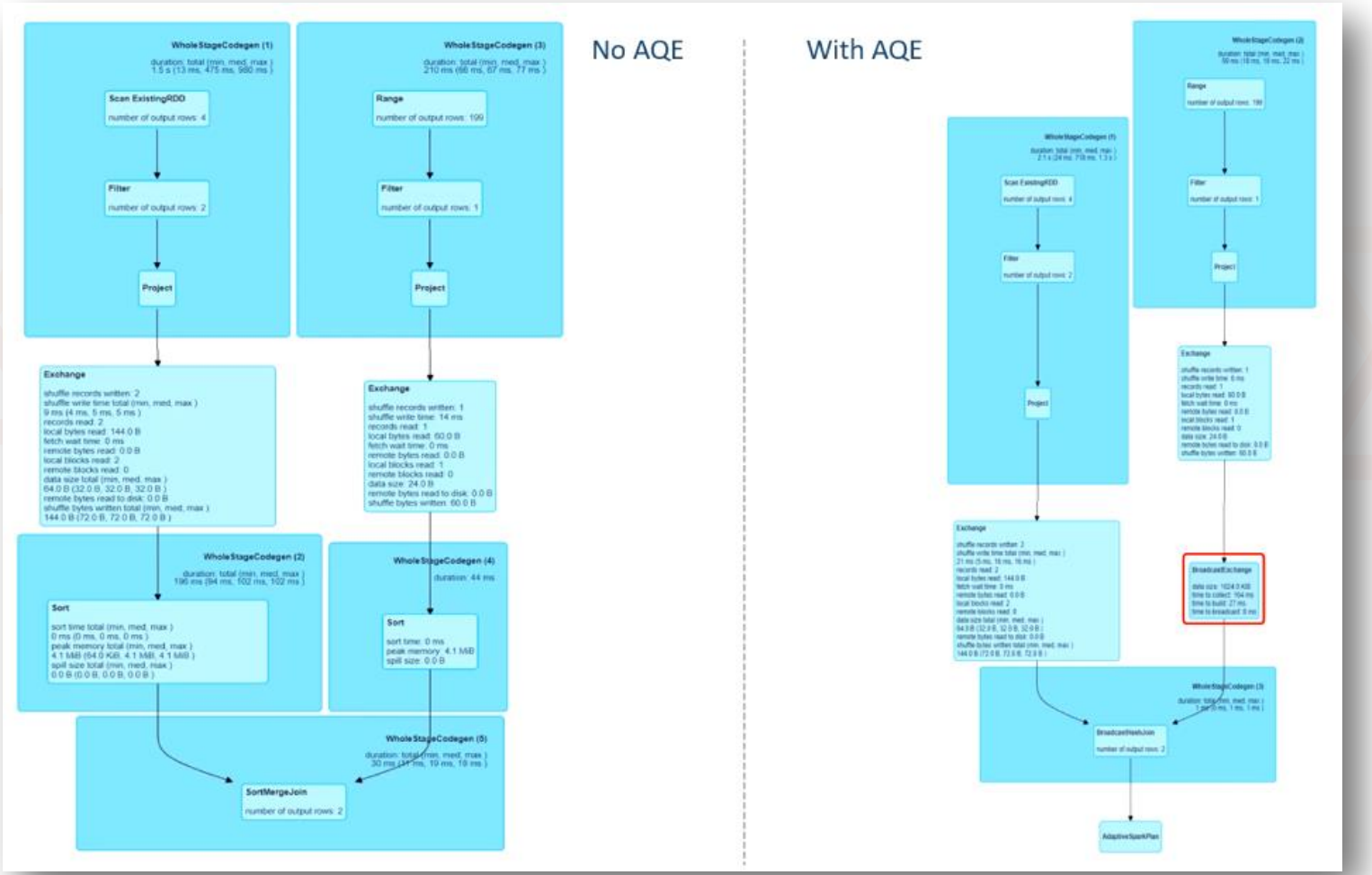
Unfortunately, we still have the shuffle, but we saved the sort operation. We couldn't save the shuffle operation, and you still see the exchange in the query plan. But the sort operation is gone.



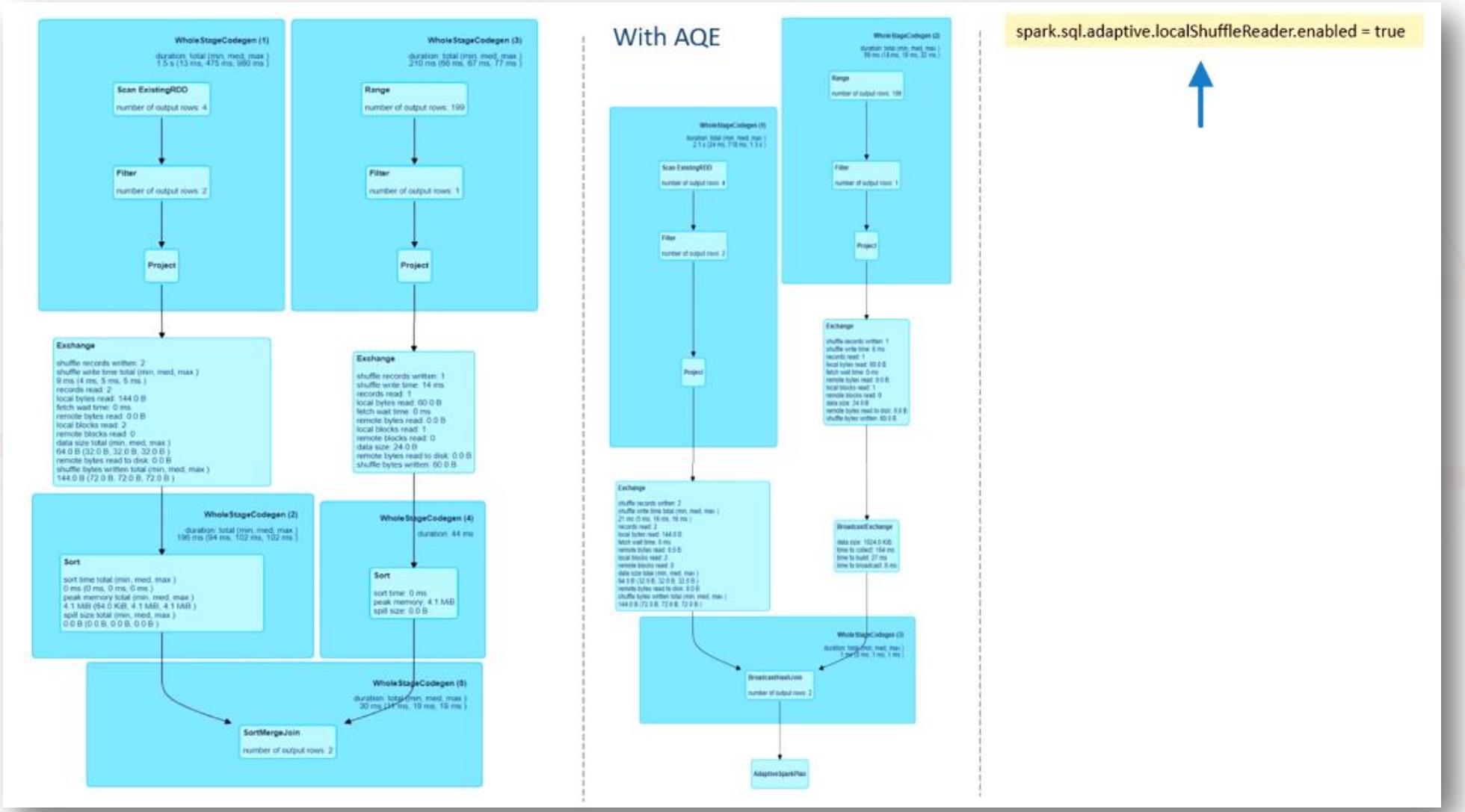
AQE cannot avoid shuffle. Why? Because the AQE computes the statistics during the shuffle. So I will be there. But the AQE will dynamically change the plan and apply broadcast hash join to save the expensive sort operation.



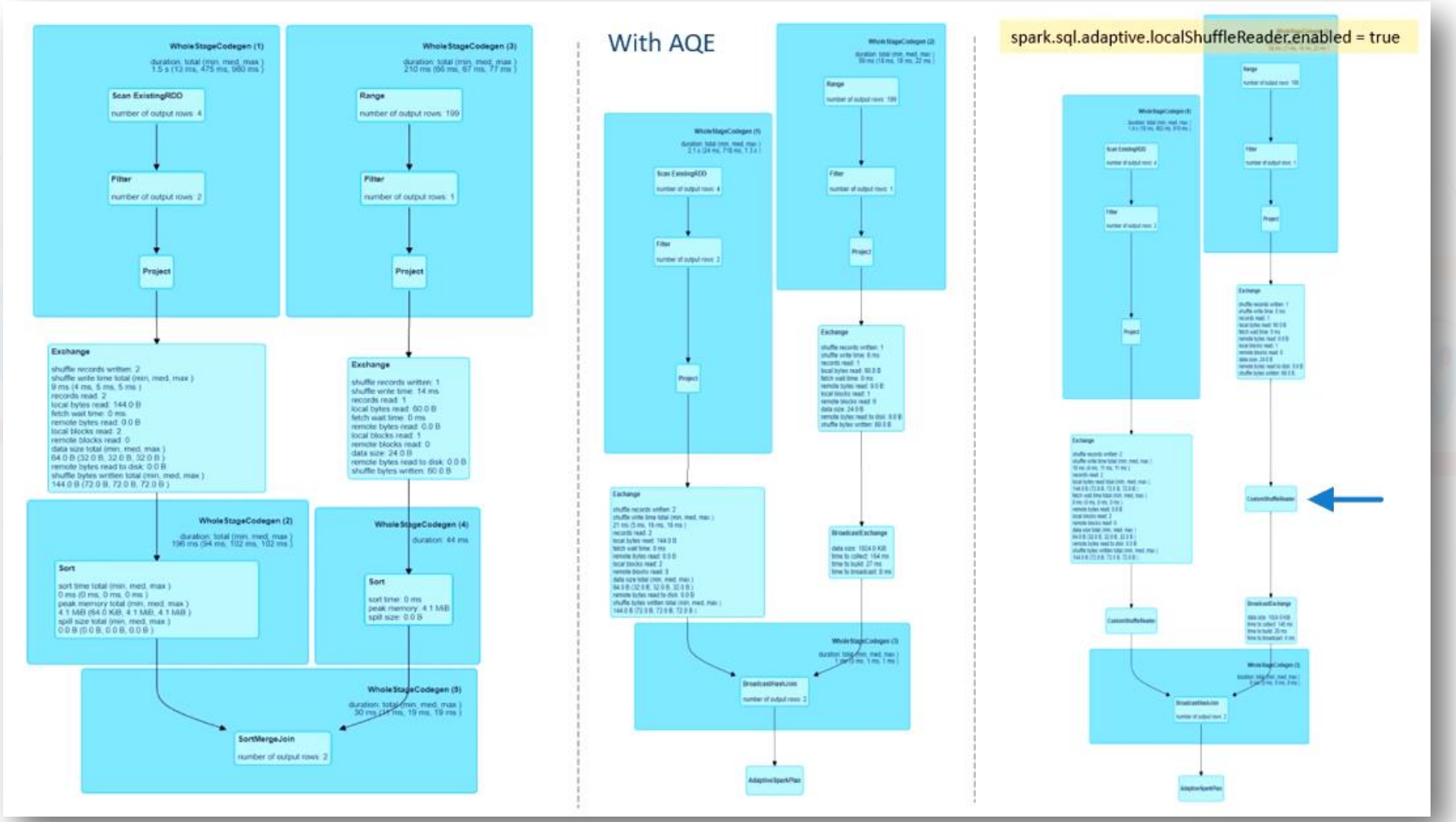
But we still have a small problem here. The shuffle operation is already complete.
We already distributed data from stage one and two to stage three exchange. But if we apply broadcast join now, are we going to broadcast the table once again? Yes. That's how the broadcast works.



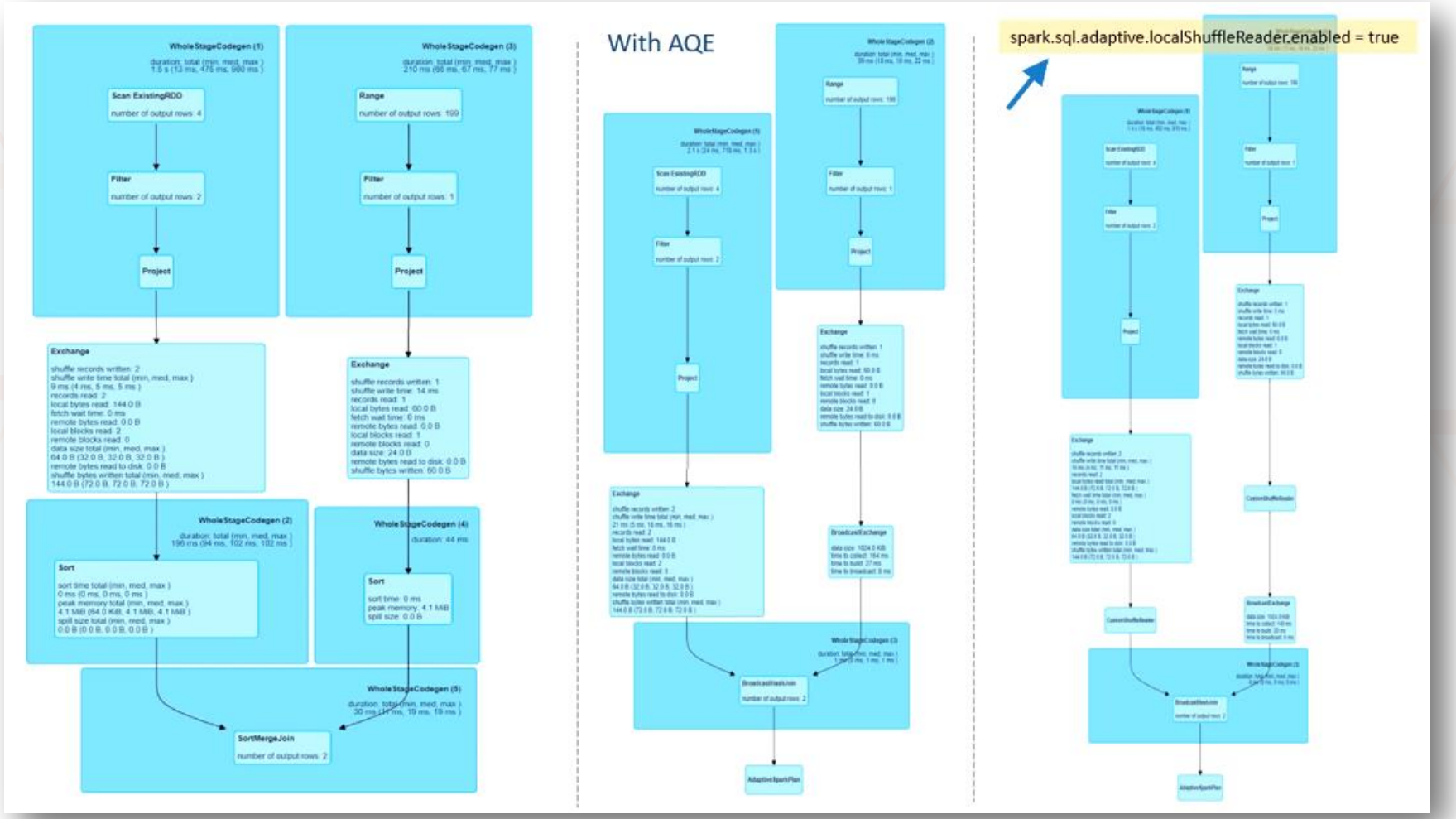
So AQE also gives you another configuration, as shown below.
The default value for this configuration is true.



So if you are not setting this value false, your optimized execution plan looks like the one shown below. What is the difference? Both the plans are almost the same. But you will see this Custom Shuffle Reader. They also call it the local shuffle reader.

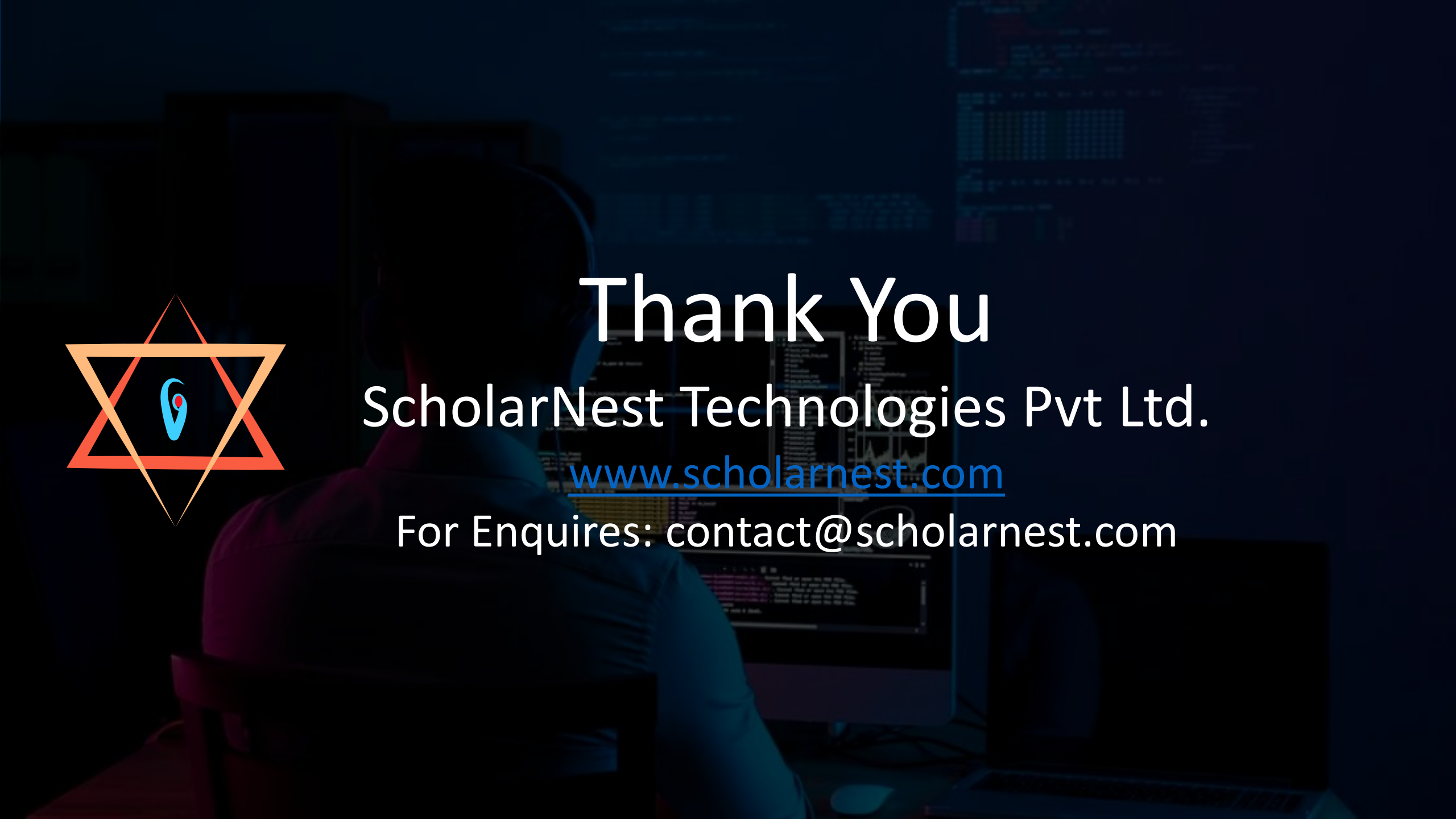


The custom local shuffle reader further optimizes the AQE broadcast by reducing network traffic. This Custom shuffle reader is specifically designed to further optimize the AQE broadcast join by reducing the network traffic. So if you are using AQE, do not disable localShuffleReader. It is anyway enabled by default.



Here is the summary of what we discussed in this chapter.

- Broadcast shuffle join is the most performant join strategy
- You can apply broadcast join if one side of the join can fit well in memory
- One table must me shorter than spark.sql.autoBroadcastJoinThreshold
- Estimating the table size is problematic in following scenarios
  - You applied a highly selective filter on the table
  - Your join table is generated at runtime after a series of complex operations
- Spark AQE can help
  - AQE computes the table size at shuffle time
  - Replans the join strategy at runtime converting sort-merge join to a broadcast hash join



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Joins and Optimization

Lecture:
Handling Data Skew

Handling Data Skew in Spark

I talked about the following three features of Spark Adaptive Query Execution:

1. Dynamically coalescing shuffle partitions
2. Dynamically switching join strategies
3. Dynamically optimizing skew joins

I also covered the first two.
In this video, I will talk about the Dynamically optimizing skew joins.
So let's start understanding the problem first. Why do we need dynamic join skew optimization?

Let's assume you have two larger tables.
And you are joining these two tables using the following query.

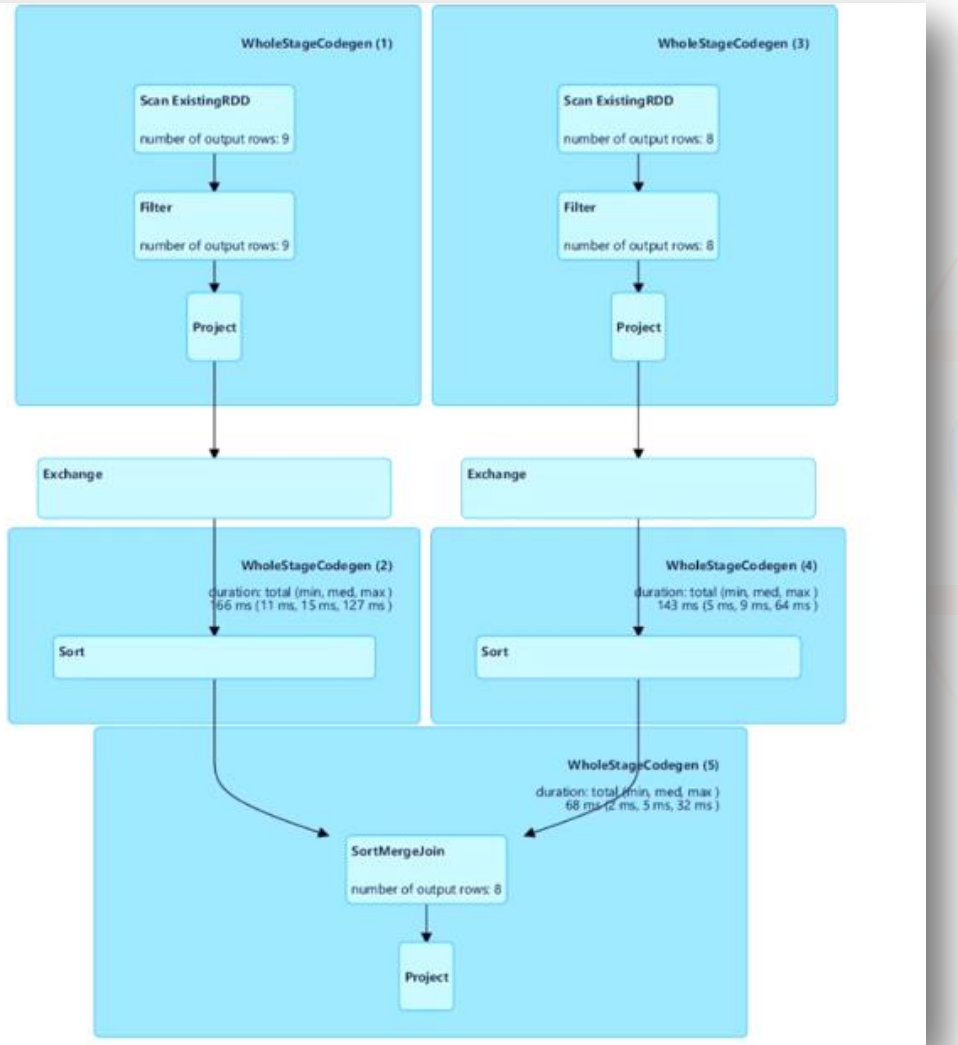
```
SELECT *
FROM large_tbl_1
        JOIN large_tbl_2
        ON large_tbl_1.key = large_tbl_2.key
```

I am showing a Spark SQL above, but you might be doing it using an equivalent Dataframe expression as shown below. Both the Spark SQL and the Dataframe queries are the same.

```
df1.join(df2, df1.key == df2.key, "inner")
    .filter("value=='some value'")
```

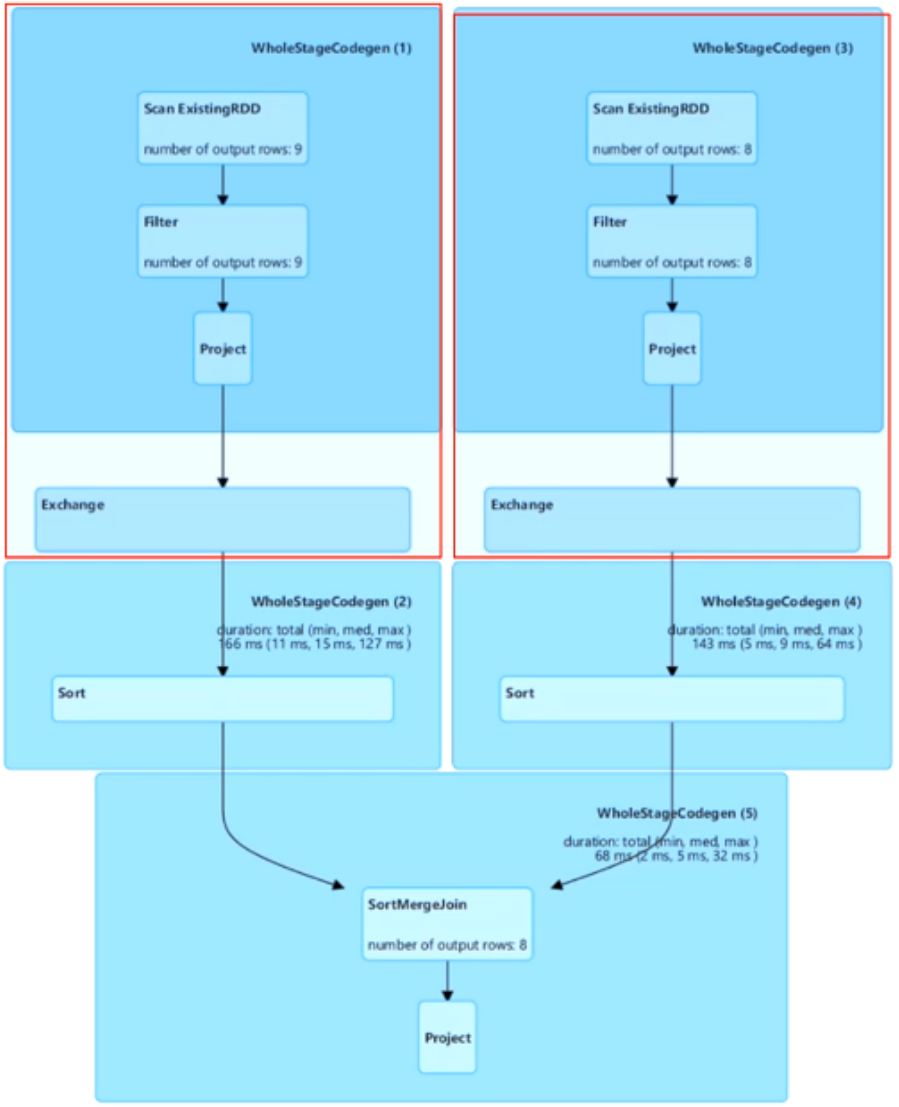

Both of your tables are large tables, so you are expecting a sort-merge join to take place. You ran the job and checked the execution plan you got as follows.

```
df1.join(df2, df1.key == df2.key, "inner")
    .filter("value=='some value'")
```



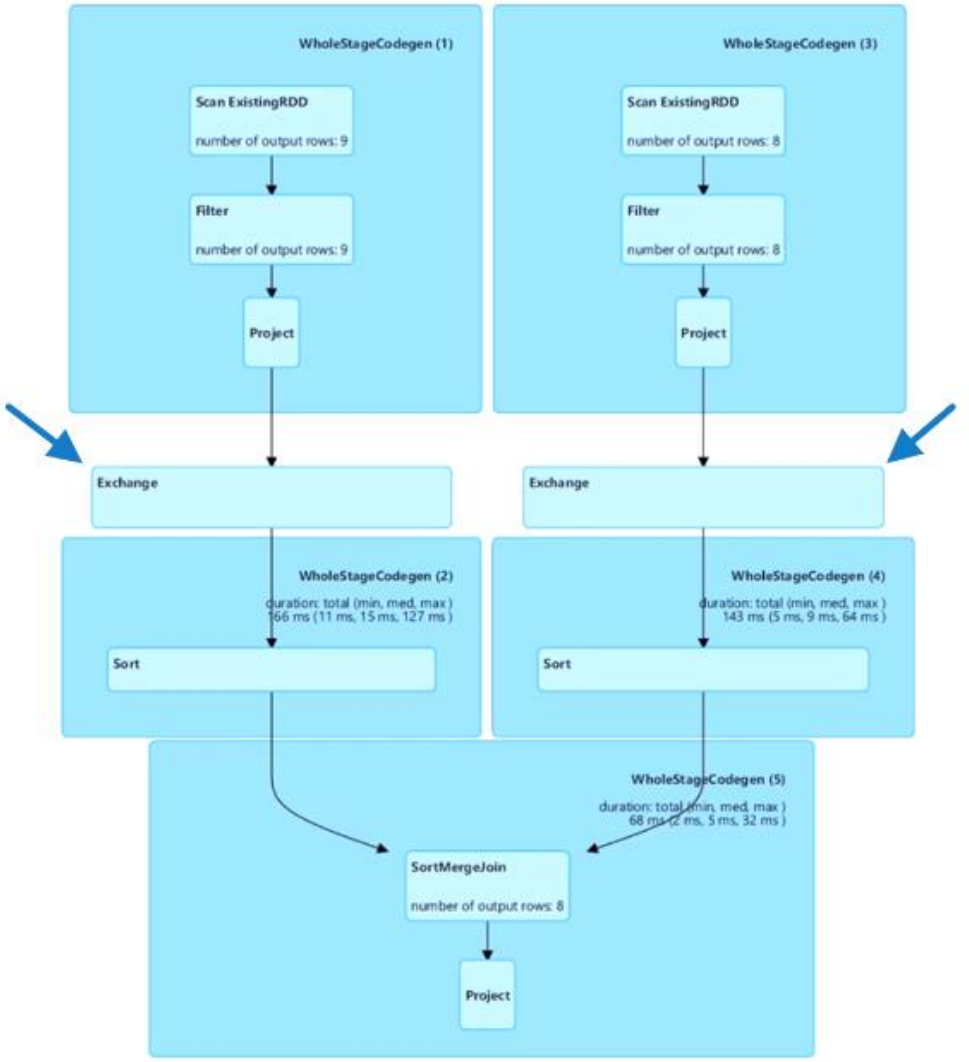
So, I am reading table one and table two. These two tables should join, so we have a shuffle operation for both tables.

```
df1.join(df2, df1.key == df2.key, "inner")
    .filter("value=='some value'")
```

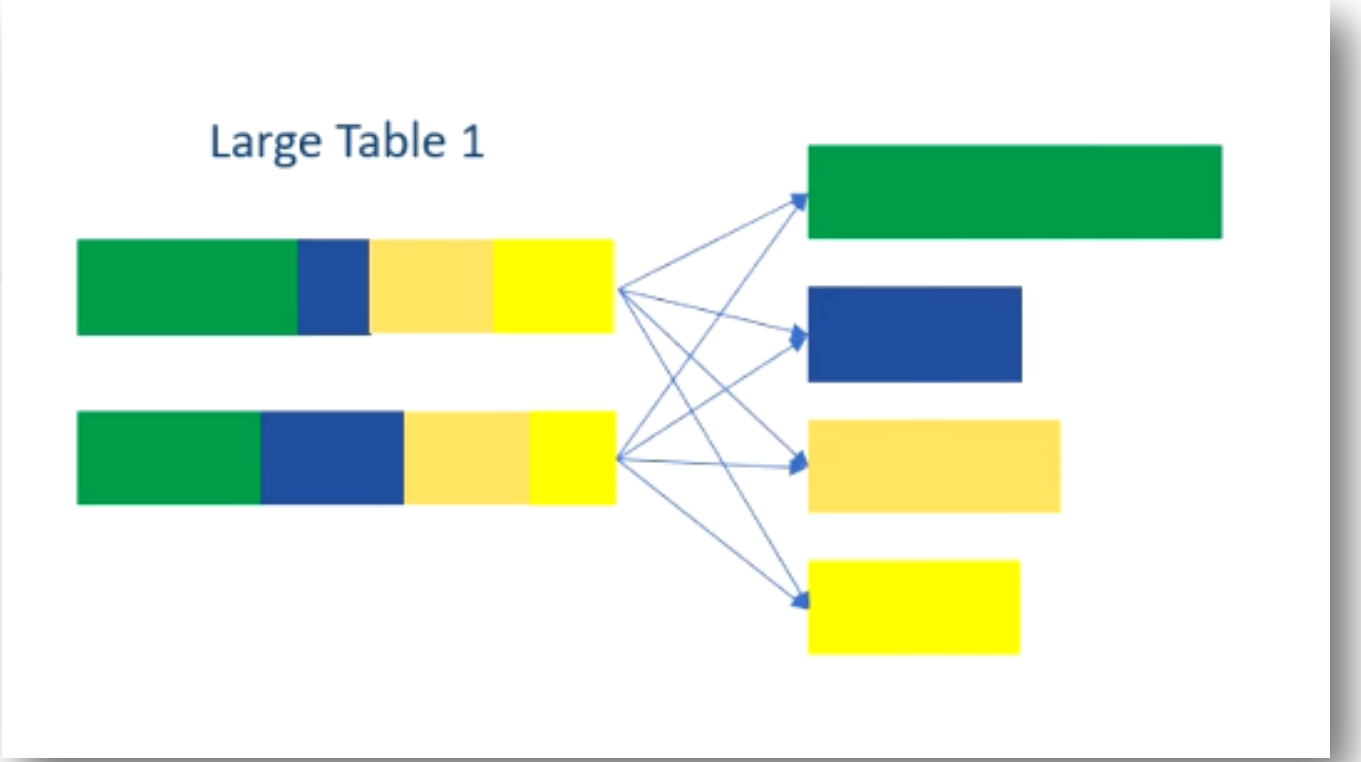


And that's why you see these two exchanges here. The first exchange (on the left side) partitions the data by the join key for the first table. And the second exchange (on the right side) partitions the data by the join key for the second table.

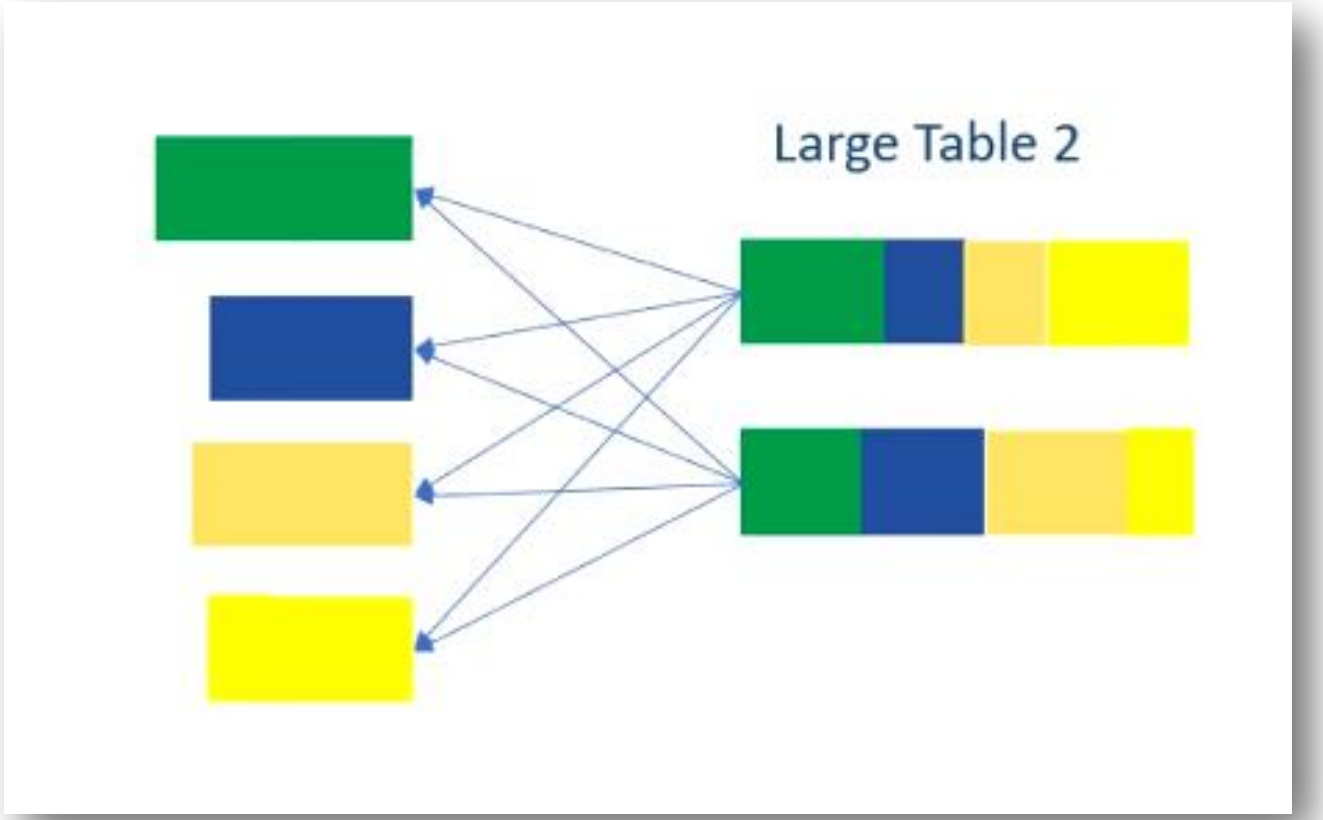
```
df1.join(df2, df1.key == df2.key, "inner")
    .filter("value=='some value'")
```



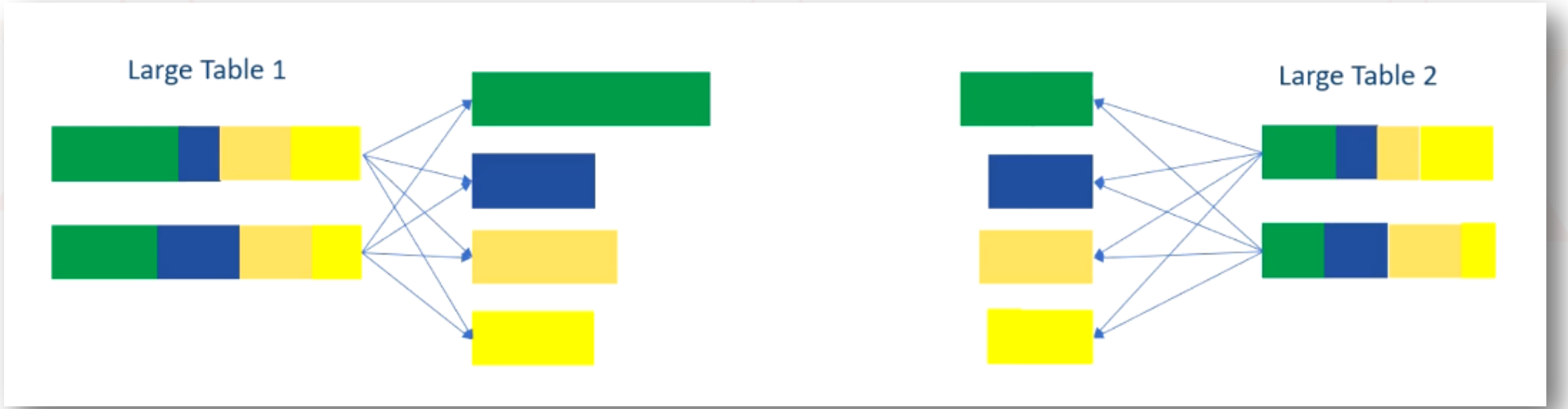
Now let's dig deeper into the two exchanges.
Let me assume that my first table had two partitions, as shown in left side of the image below. Each color represents data for one unique join key.
So we read these two initial partitions, and then we shuffled them. The result of the shuffle looks like the one shown in the right side of the image. The primary purpose of the shuffle is to repartition the data by the key. And that's what is happening here.



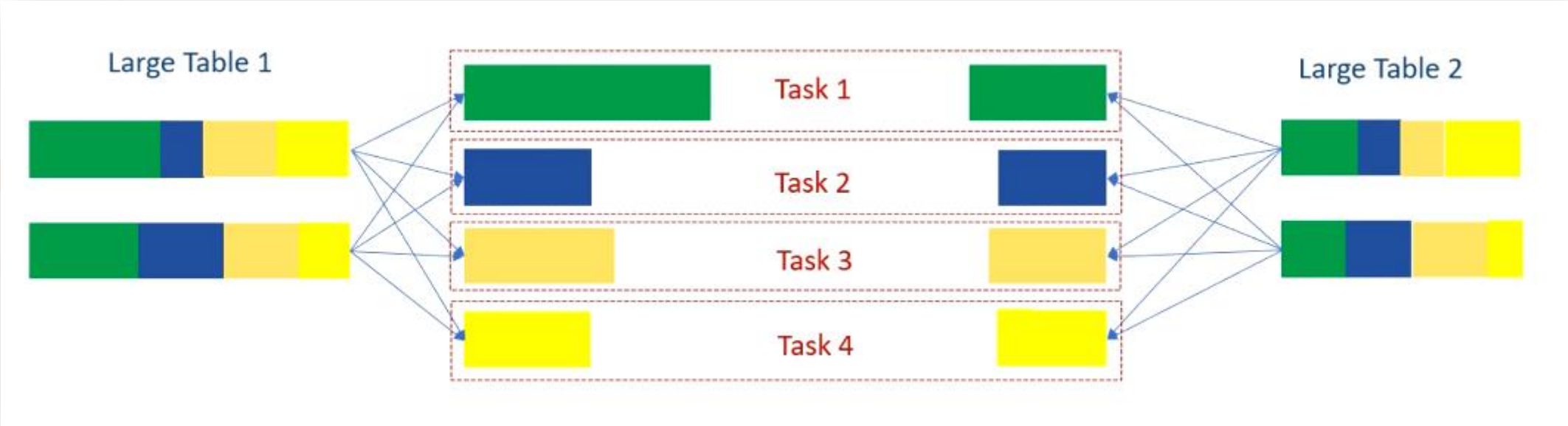
What about the second table?
Well, that also goes through the same process.



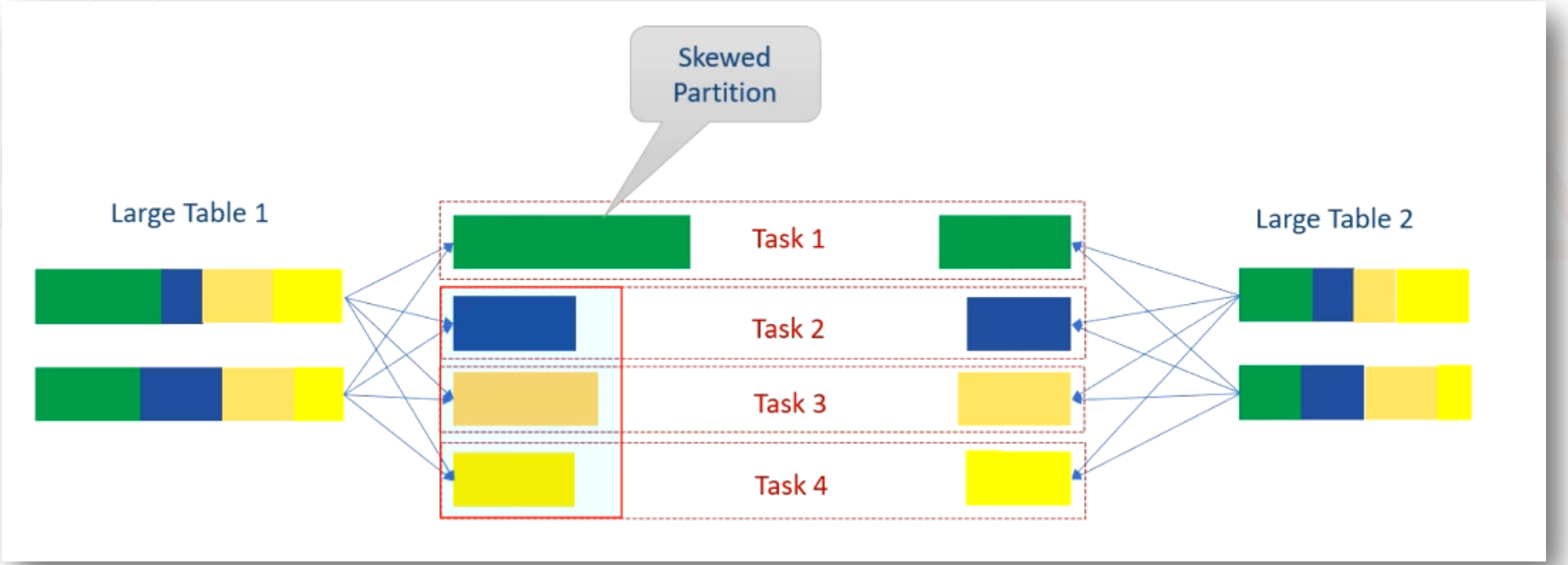
So partitions for the first table are coming from the left side, and the second table comes from the right side.
Spark appropriately partitioned the data by the key.
Now, all we need to do is sort each partition by the key and merge the records from both sides.



I have four partitions to join.
So I will need four tasks.
Each task will pickup one color, sort the data, and merge it to complete the join operation.



I see a problem here. Look at the green color partition on the left side, it is quite large. The green partition is skewed. Look at the blue, orange, and yellow. They are smaller, but the green one is almost double the size. And that's a problem. I need more memory to sort and merge the green partition. I planned 4 GB RAM for each task, and that should be sufficient for the other tasks. But the task working with the green color cannot manage the sort/merge operation with 4 GB RAM.



What can I do to tackle this data skew problem? Should I increase the Spark memory?
Well, I can increase it, but that's not a good solution.
Why? I have two reasons:

1. **Memory Wastage:** I can increase the memory, assuming that I will have a skewed partition. But I am not sure if all my joins will result in a skewed partition. What if they don't? You might have 10 or 15 join operations in your Spark application. All other join operations work perfectly fine with 4 GB task memory. However, you have one skewed join, and one task needs extra memory. Increasing memory for one specific join or task is not possible. So you will end up increasing memory for your entire application, and that's wastage.

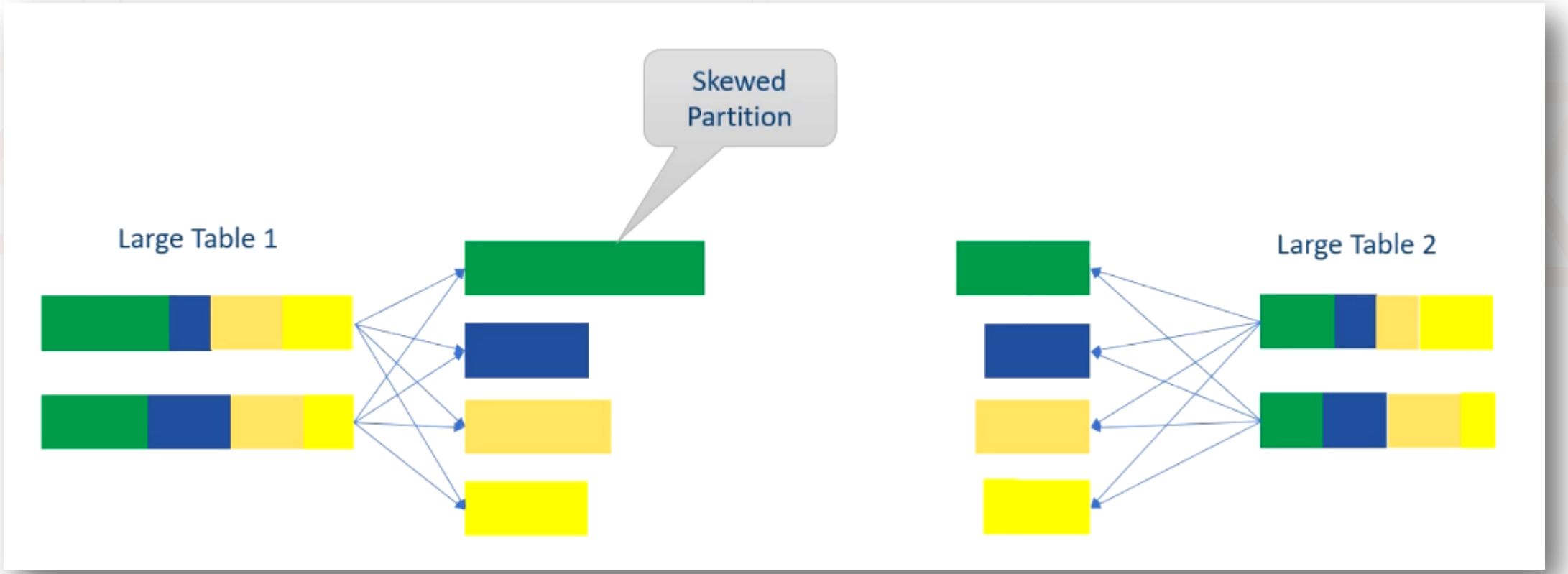
2. **Increasing memory for skewed join is not a permanent solution:** Because data keeps changing. You might have a skewed partition today, which required 6 GB to complete the sort/merge operation. But you do not know what happens a week later. You got new data, and the skew is now more significant. You now need 8 GB to pass through that skew. I cannot let my application fail every week or month, investigate the logs, and identify that we need more memory because data is now more skewed.

So what is the solution? Spark AQE offers you an excellent solution for this problem.
You can enable skew optimization using the following configurations:

1. **spark.sql.adaptive.enabled** – It enables the Spark AQE feature.
2. **spark.sql.adaptive.skewJoin.enabled** – It enables the skew-join optimization.

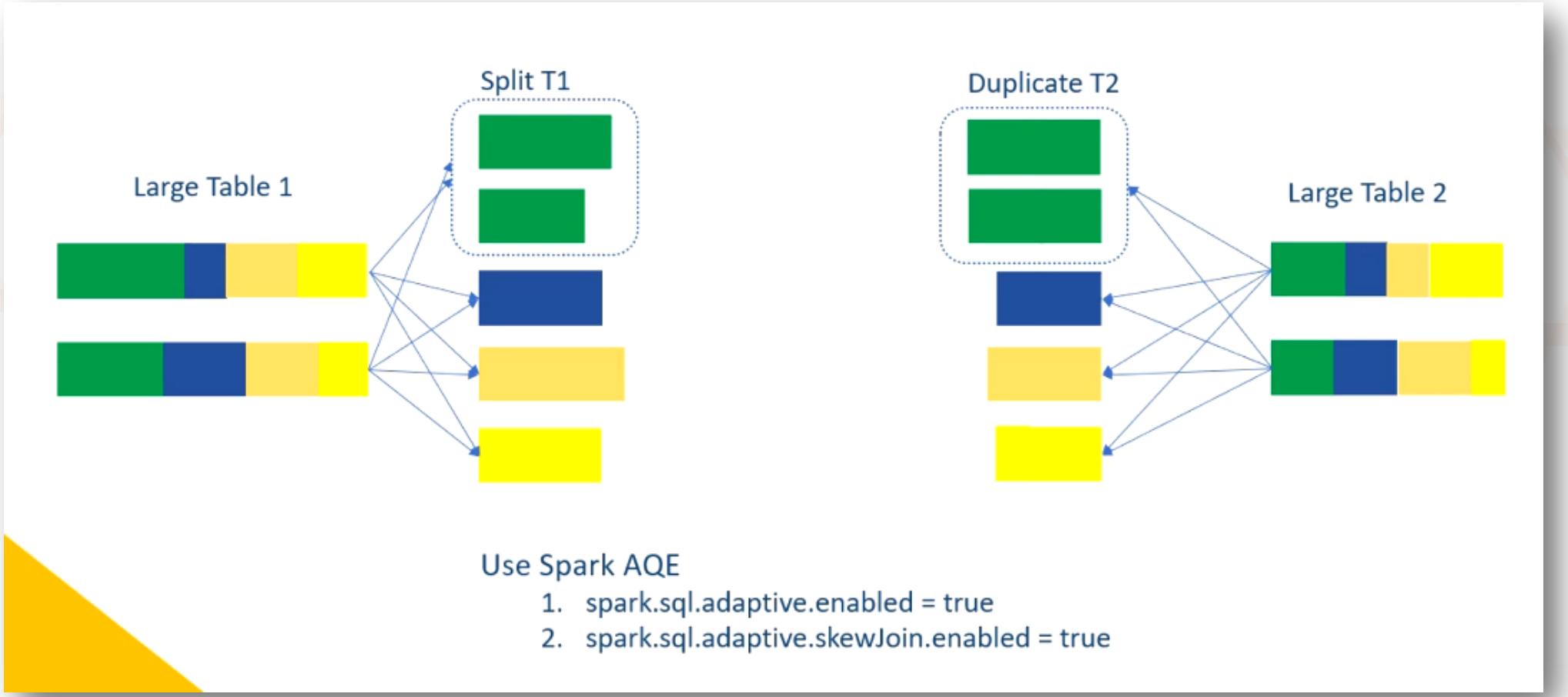
We have two more configurations, and I will talk about them in a minute.
But let's try to understand what happens when we enable AQE and skew-join optimization.

So here we are with our scenario.
I have four shuffle partitions, so I will need four tasks to perform this sort/merge join operation. However, the green partition is skewed. So the green task will struggle to finish and take longer to complete. In the worst case, it might fail due to a memory crunch.
But the other three tasks will complete quickly and normally because they are small enough.

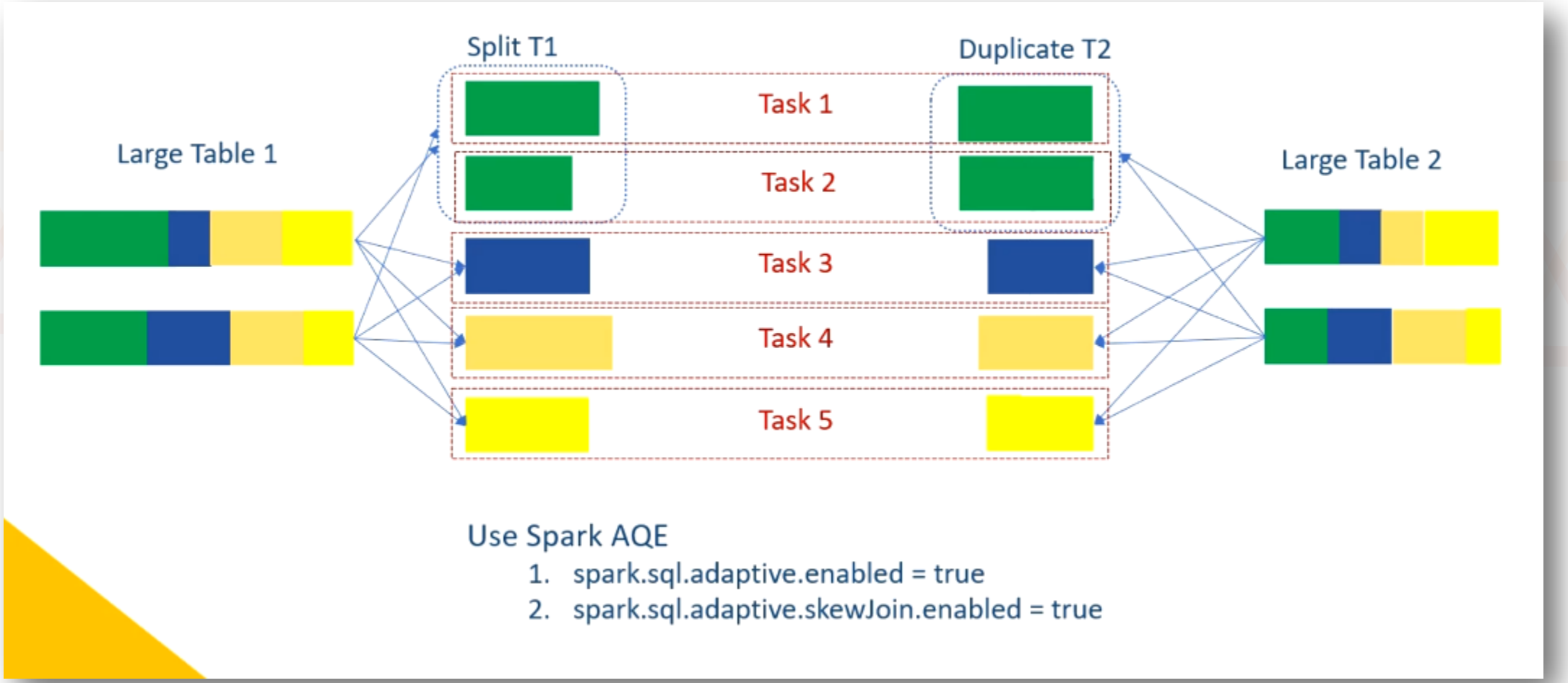


However, I enabled AQE and skew-join optimization. So Spark AQE will detect this skewed partition. And it will do two things:

1. Split the skewed partition on the left side into two or more partitions. In our example, splitting into two is sufficient so let's assume Spark splitting it into two partitions.
2. Spark will also duplicate the matching right-side partition.



And our problem is solved now that we have five partitions.
Earlier, we had four, but now we have five partitions. So we will need five tasks.
The data partitions for all five tasks are almost the same, so they will consume uniform resources and finish simultaneously.

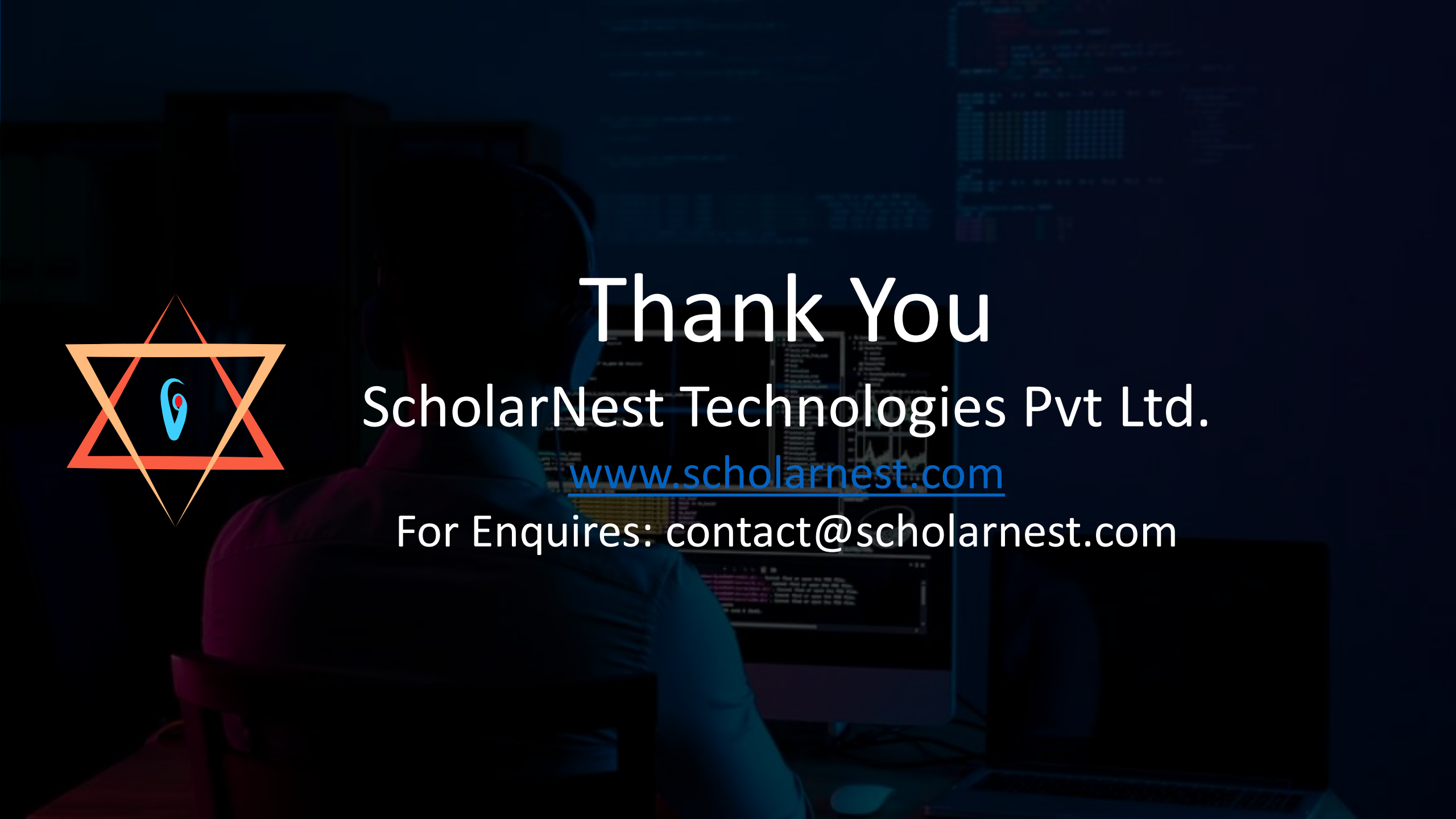


I talked about two configurations earlier. However, we also have two more configurations to customize the Skew Join optimization:

1. spark.sql.adaptive.skewJoin.skewedPartitionFactor
2. spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes

These two configurations are used to define the skew.
What does it mean? When do we consider a partition as a skewed partition and start splitting it? Well,
Spark AQE assumes that the partition is skewed and starts splitting when both thresholds are broken. Now let's talk about these thresholds.
The default value of *skewedPartitionFactor* is five.
So a partition is considered skewed if its size is larger than five times the median partition size.

The default value of *skewedPartitionThresholdInBytes* is 256MB.
So a partition is considered skewed if its size in bytes is larger than this threshold.
But remember! Spark AQE will initiate the split if only if both the thresholds are broken.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Spark Joins and Optimization

Lecture:
Dynamic Partition Pruning

Spark Dynamic Partition Pruning

In this chapter, I will talk about Dynamic Partition Pruning in Apache Spark.
Dynamic Partition Pruning is a new feature available in Spark 3.0 and above.
And this feature is enabled by default.
However, if you want to disable it, you can use the following configuration:

spark.sql.optimizer.dynamicPartitionPruning.enabled

The default value for this configuration is true in Spark 3.0.
So you don't have to do anything to enable Spark Dynamic Partition Pruning.
But in this chapter, we want to understand the following:

1. Why do we need Dynamic Partition Pruning?
2. What is Dynamic Partition Pruning?

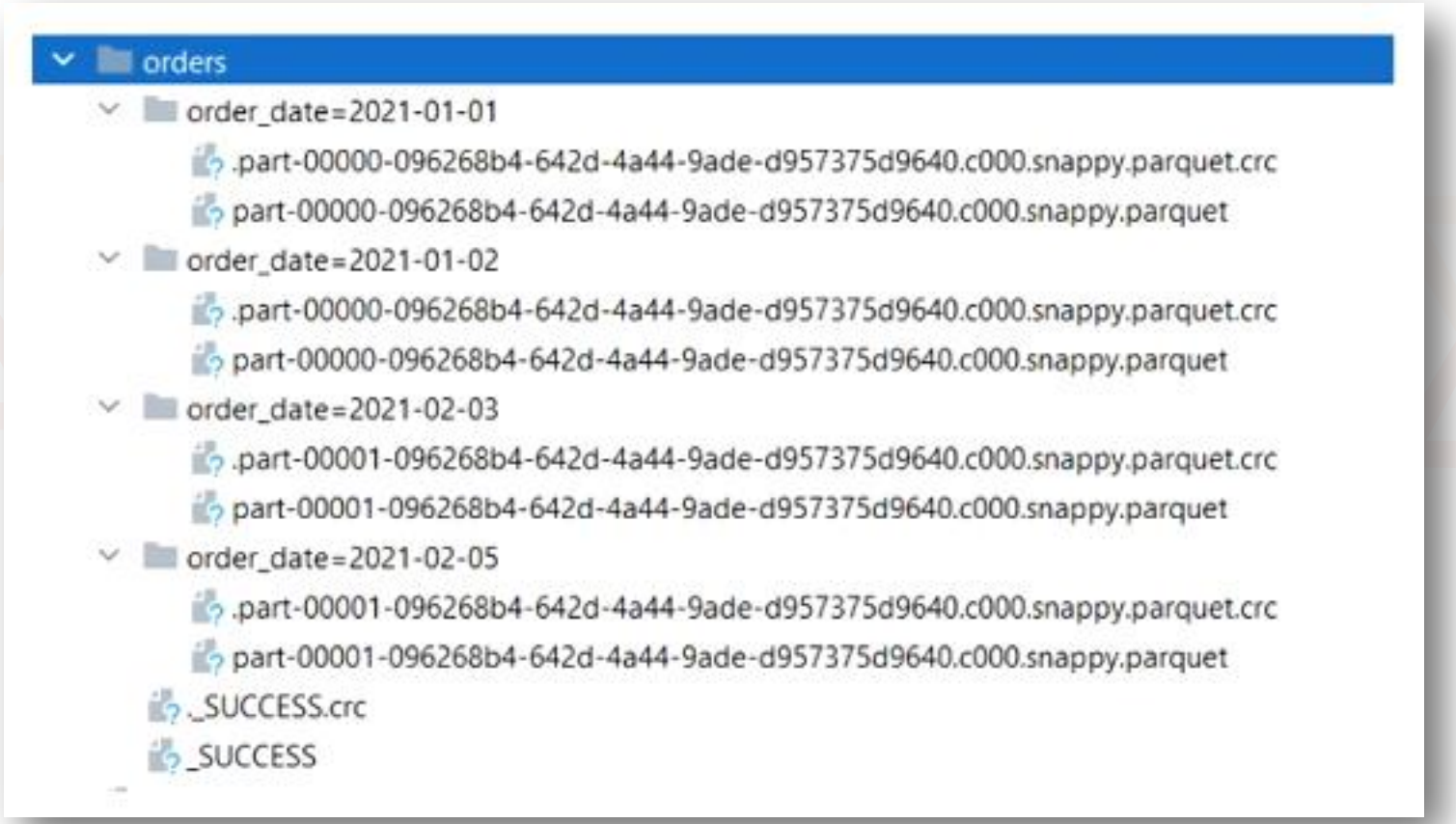
So let's start with the problem statement and try to understand why do we need it?

Let's assume you have two tables.
The first table is an orders table, and it stores all your orders. The high-level table structure looks like this. This table is huge because it stores thousands of orders every day and millions of orders a year. So you decided to partition it on order_date.

Orders

order_id	order_date	prod_id	unit_price	qty
101	2021-01-01	PDX1	350	7
102	2021-01-02	LDX1	580	5
102	2021-01-02	PDX1	350	3
104	2021-02-03	LDX1	580	8
105	2021-02-03	PDX1	350	6
106	2021-02-05	LDX1	580	9

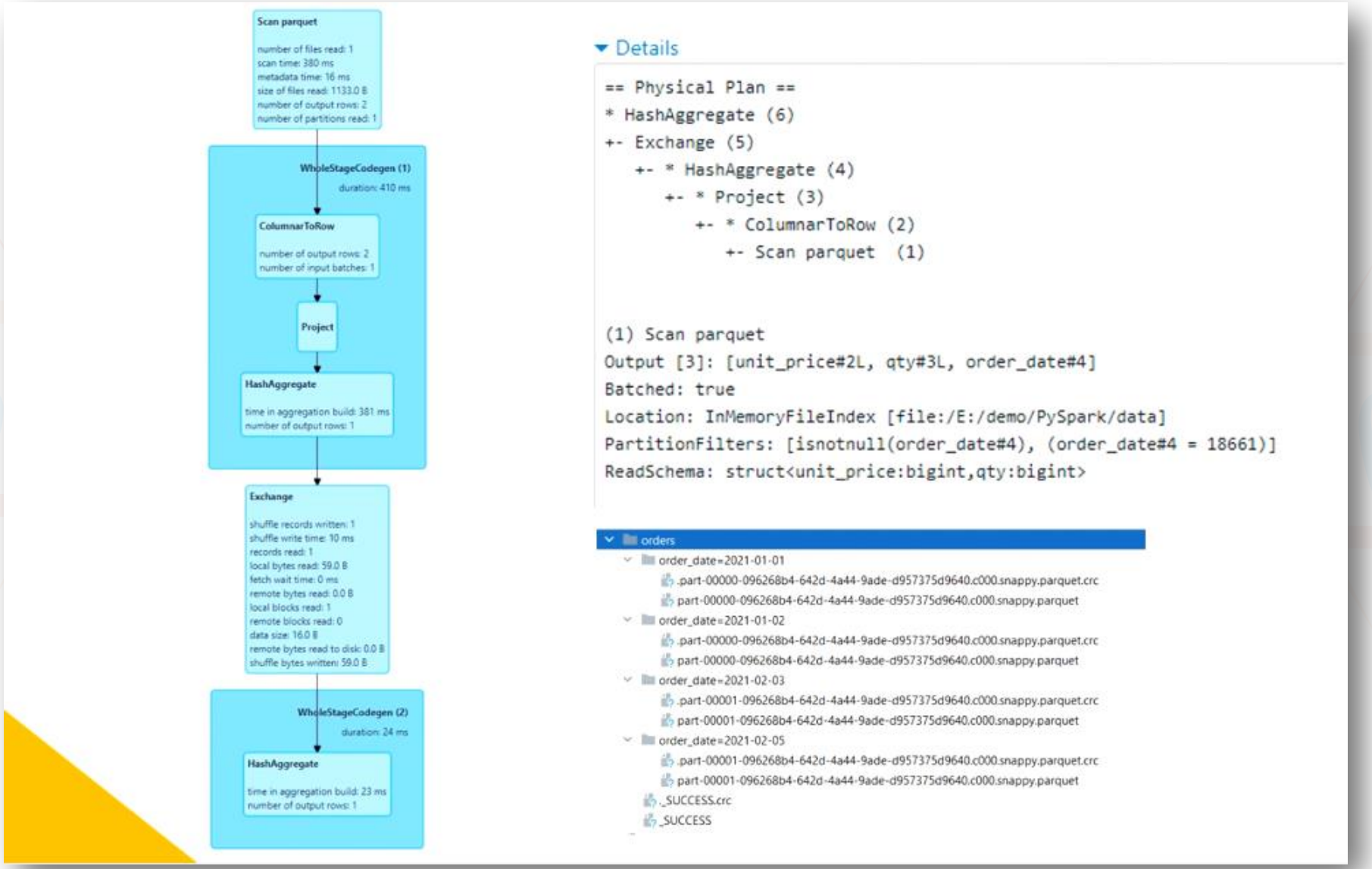
Here is a screenshot of how my orders data set is stored.
I have done it on a local machine. However, the arrangement is almost the same as on a distributed storage such as HDFS.



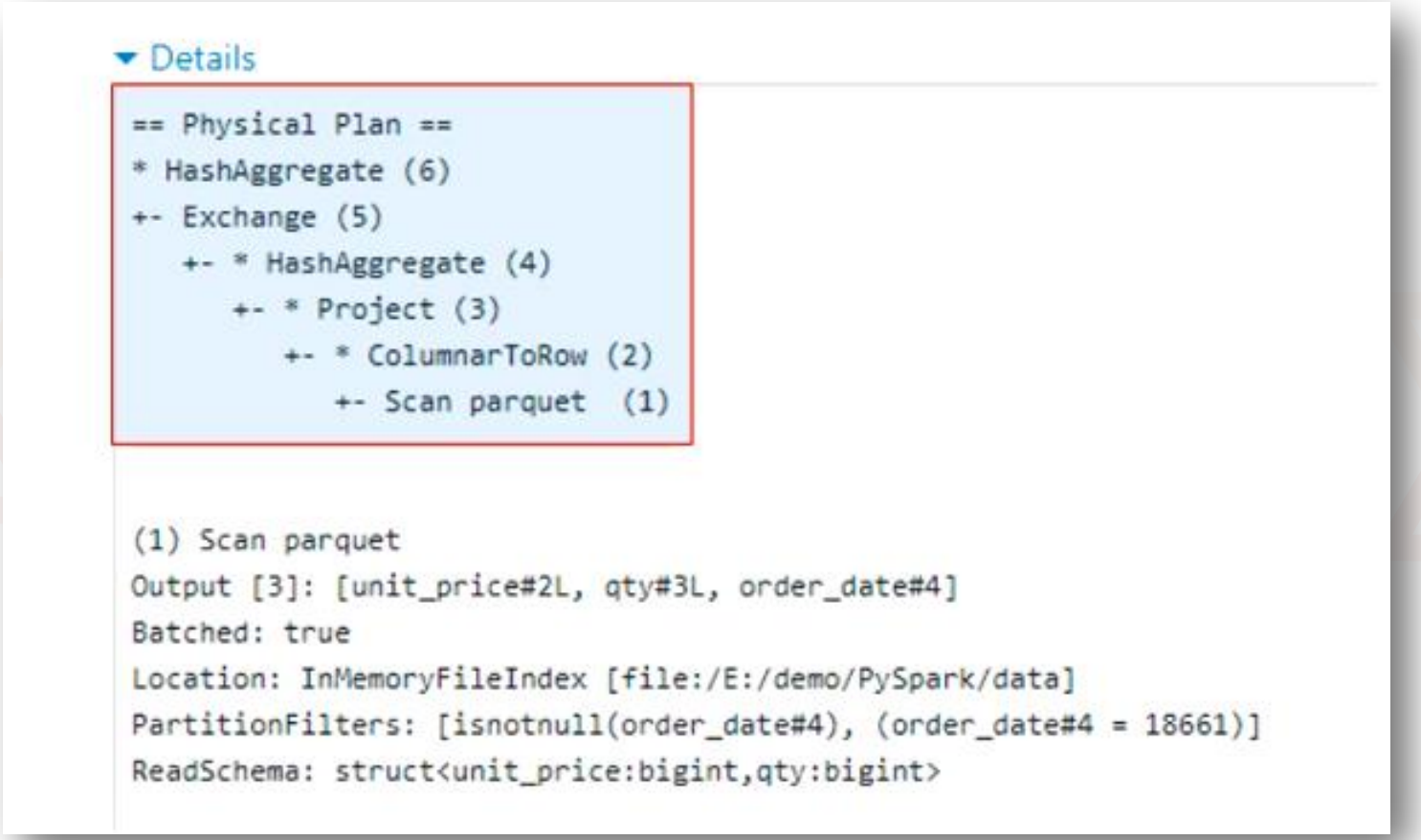
But you might ask the following question: Why do we partition it on the order date?
The answer is simple. You know you are querying this table on the order date. Here is an example code shown below.
In the code, I am reading the orders data set.
Then applying a filter on order_date and finally computing total sales for 3rd Feb 2021.

```
order_df = spark.read.parquet("orders")
summary_df = order_df
        .where("order_date=='2021-02-03'") \
        .selectExpr("sum(unit_price * qty) as total_sales")
```

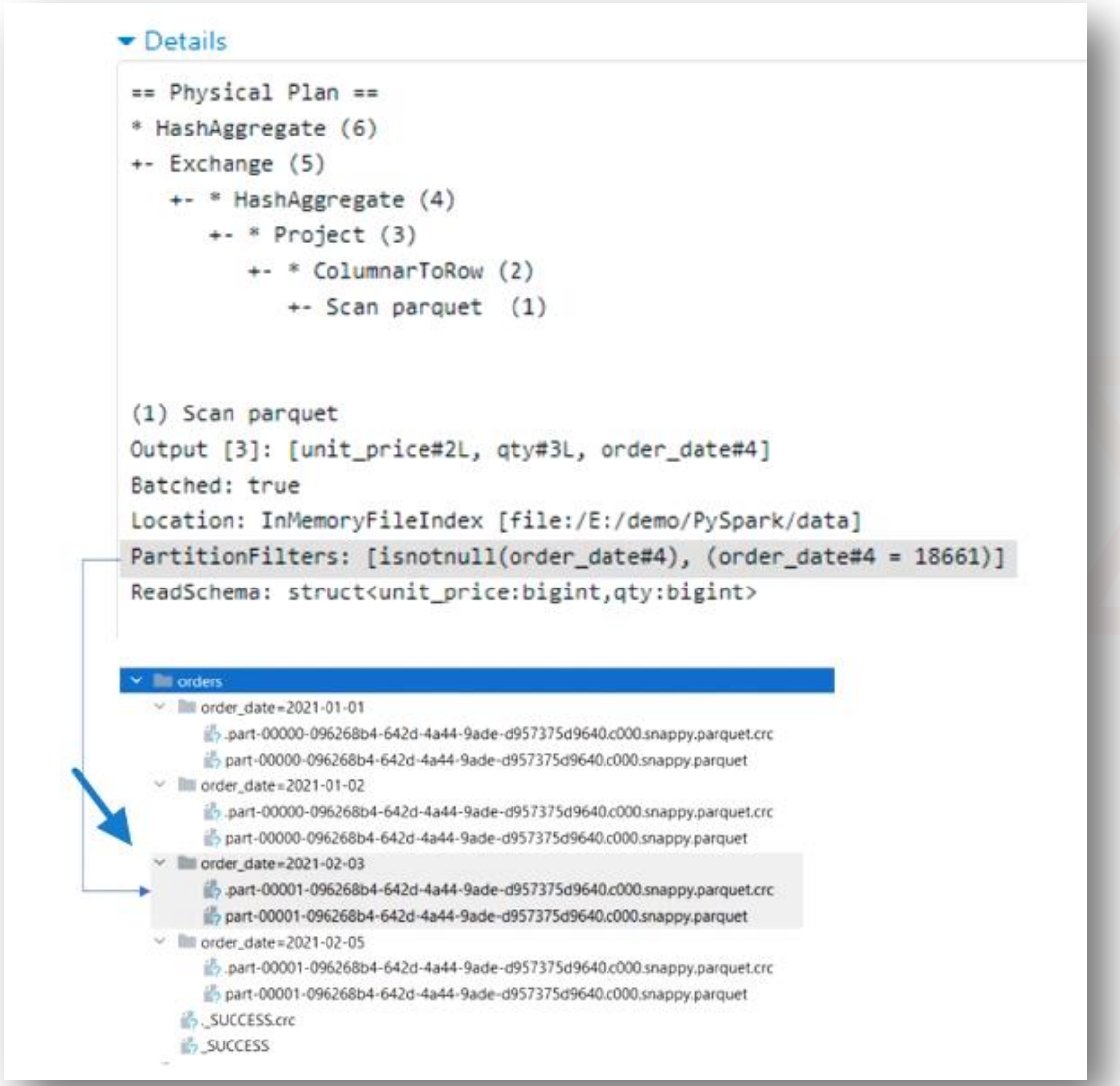

I ran this code and checked my query plan. Here is the screenshot shown below.



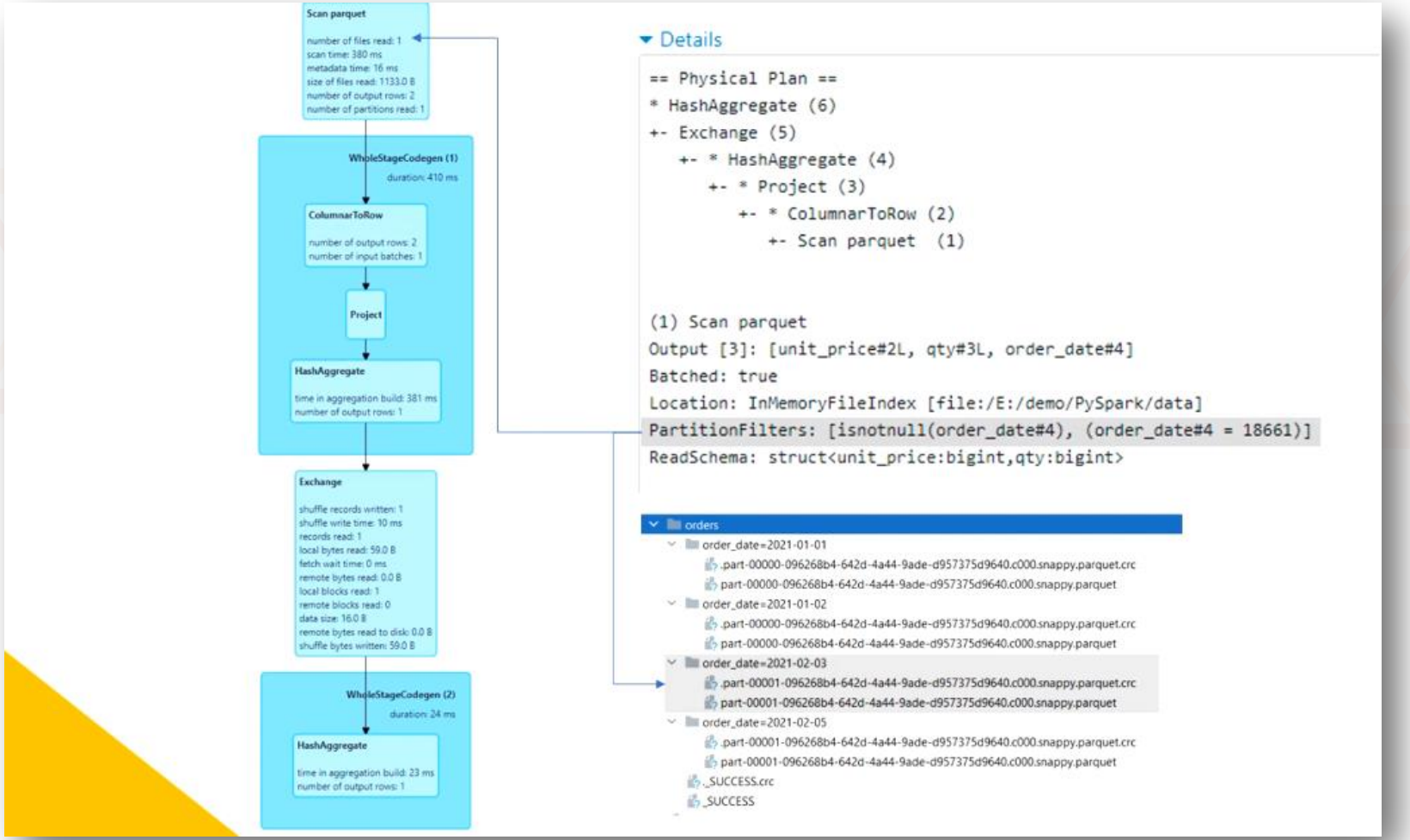
The highlighted image shows the physical plan for our query. It clearly shows that Spark performed six steps to complete this query. The first step is to scan or read the parquet file.



But Spark is doing a smart thing here. Let's look at the details of step one.
Do you see these Partition Filters? Spark applied Partition Filters on the order_date column, and it is reading only one partition.



The plan also shows a number of files read equals one.
So what is happening here? In a typical case, Spark should read all the partitions of my parquet data set and then apply the filter. But my data set was partitioned on the order_date column. So Spark decided to read only one partition that belongs to 3rd Feb 2021.



So I am trying to show you two features of Spark query optimization:

1. Predicate Pushdown
2. Partition Pruning

The predicate pushdown means Spark will push down the where clause filters down in the steps and apply them as early as possible.
I mean, Spark will not try a typical sequence to read the data first, then filter, and finally, calculate the sums. No! Spark will push the filter condition down to the scan step and apply the where clause when reading the data itself.
But predicate pushdown doesn't help much unless your data is partitioned on the filter columns. My data set was already partitioned on order_date. So Spark decided to read only one partition for 3rd Feb 2021. It can simply leave all other partitions, and that feature is known as Partition Pruning.

So, these two features will optimize your query and reduce the read volume of your data.
If you are reading less amount of data, your queries run faster.

Now come to the dynamic partition pruning. What is Dynamic Partition Pruning? Let's try to understand that. Let's assume you have another table, the dates table.

Orders - Fact

order_id	order_date	prod_id	unit_price	qty
101	2021-01-01	PDX1	350	7
102	2021-01-02	LDX1	580	5
102	2021-01-02	PDX1	350	3
104	2021-02-03	LDX1	580	8
105	2021-02-03	PDX1	350	6
106	2021-02-05	LDX1	580	9

Dates - Dimension

full_date	year	month	day
2021-01-01	2021	01	01
2021-01-02	2021	01	02
2021-02-03	2021	02	03
2021-02-04	2021	02	04
2021-02-05	2021	02	05

If you have any experience in a data warehouse, you can quickly identify this table. This table is known as the date dimension in the data warehousing world. My orders table is a fact table, and the dates table is a dimension table. This kind of table structure is common in data warehouses.

Orders - Fact

order_id	order_date	prod_id	unit_price	qty
101	2021-01-01	PDX1	350	7
102	2021-01-02	LDX1	580	5
102	2021-01-02	PDX1	350	3
104	2021-02-03	LDX1	580	8
105	2021-02-03	PDX1	350	6
106	2021-02-05	LDX1	580	9

Dates - Dimension

full_date	year	month	day
2021-01-01	2021	01	01
2021-01-02	2021	01	02
2021-02-03	2021	02	03
2021-02-04	2021	02	04
2021-02-05	2021	02	05

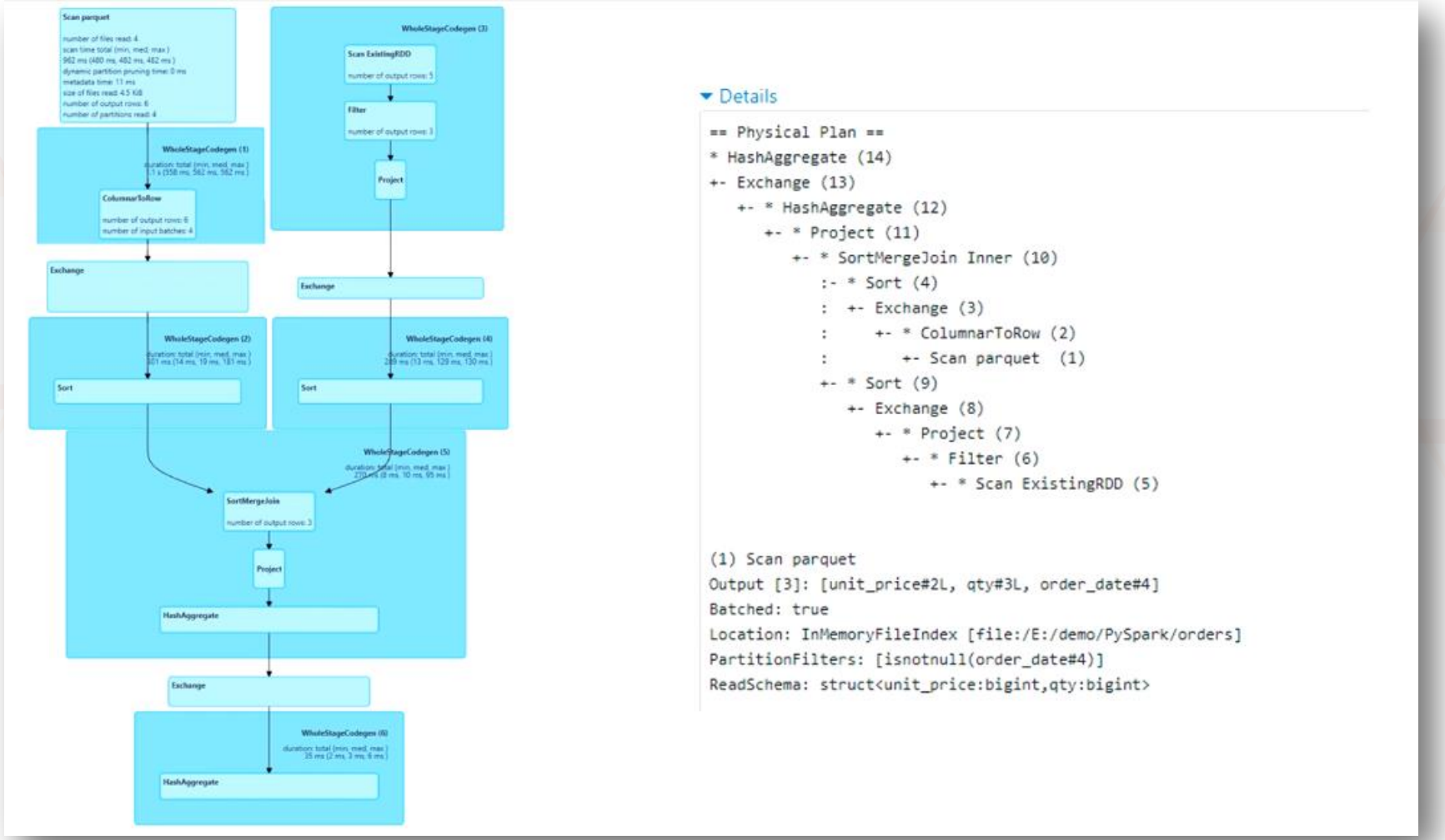
Now let's assume I am running a query like this shown below.
So what am I doing here?
I want to calculate the sum of sales for February 2021.

```
SELECT year, month, sum(unit_price * qty) as total_sales
FROM orders JOIN dates ON order_date == full_date
GROUP BY year, month
WHERE year=='2021' and month=='02'
```

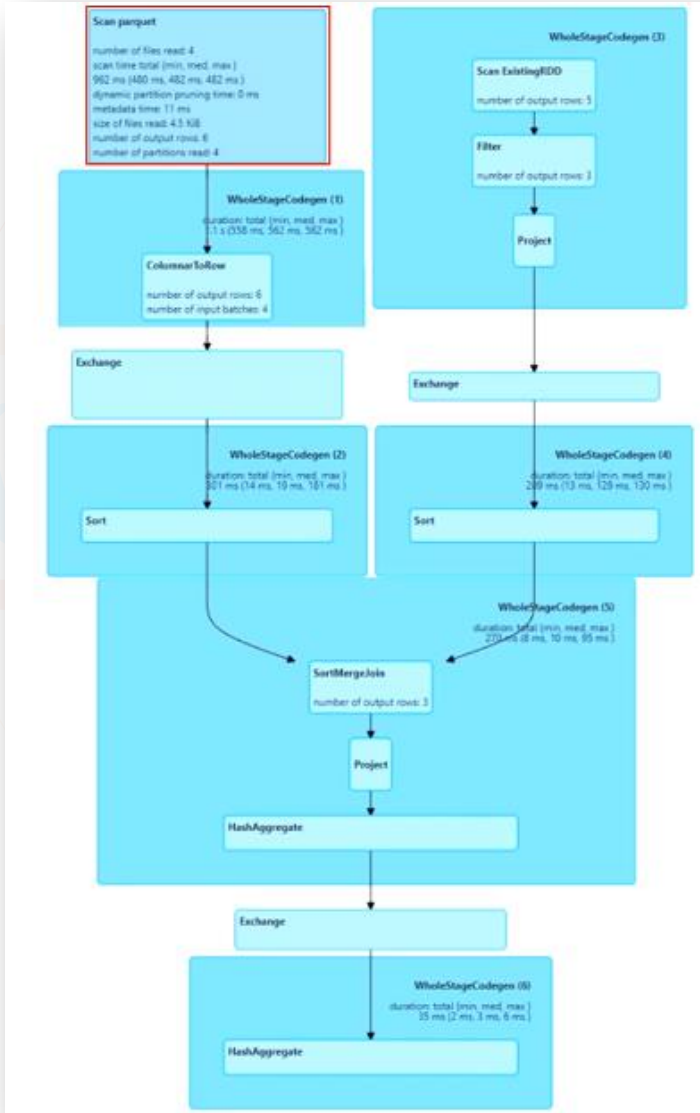

Here is an equivalent Dataframe expression.
The SQL and the Dataframe expressions are the same.
You can execute this code on an older version of Spark where dynamic partition pruning is not available.
Or, you can disable the dynamic partition pruning feature and try this SQL.

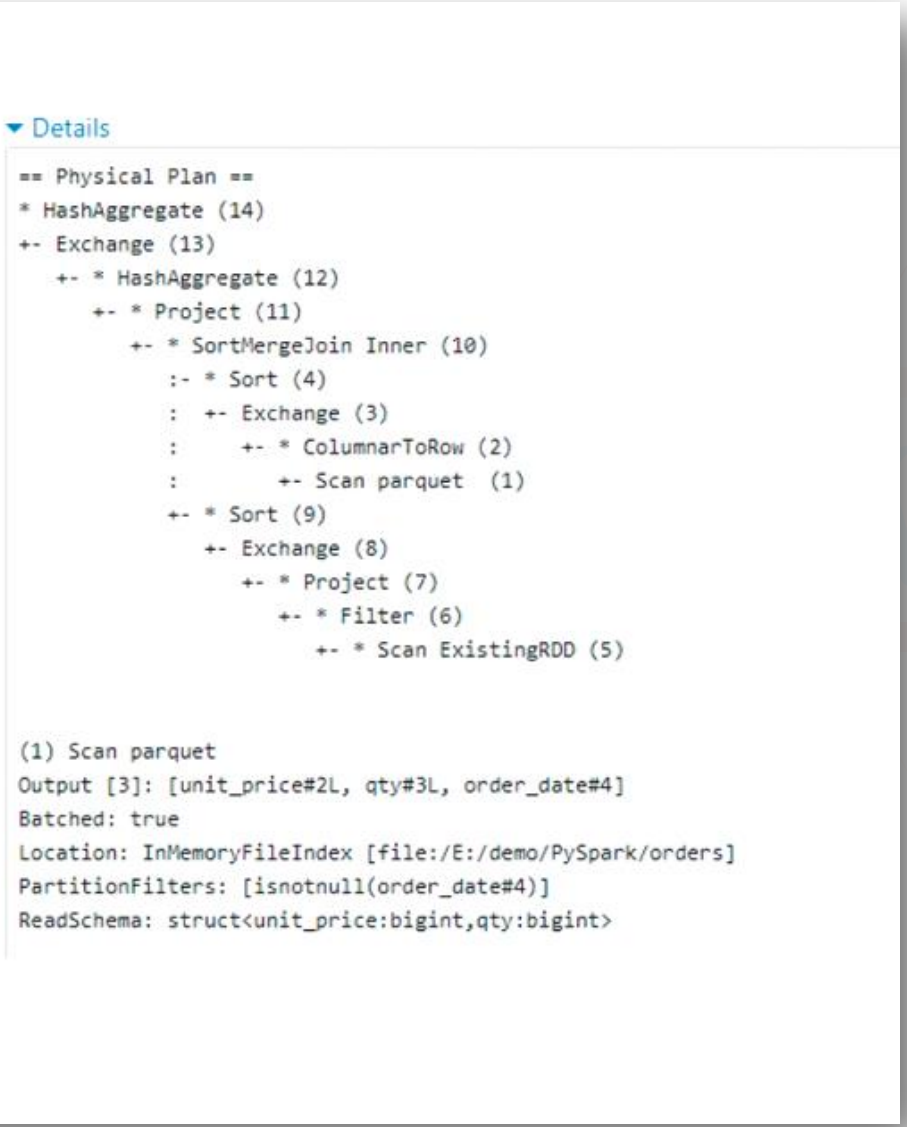
```
join_expr = order_df.order_date == date_df.full_date

order_df
    .join(date_df, join_expr, "inner") \
    .filter("year==2021 and month==2") \
    .groupBy("year", "month") \
    .agg(f.sum(f.expr("unit_price * qty"))
    .alias("total_sales"))
```


I tried, and I got the following execution plan. This is a typical sort/merge join execution plan. I am reading the parquet files and also reading the other data set. Both are going for a shuffle operation, and hence you can see these two exchanges. Finally, Spark will sort, then merge and finally aggregate it.

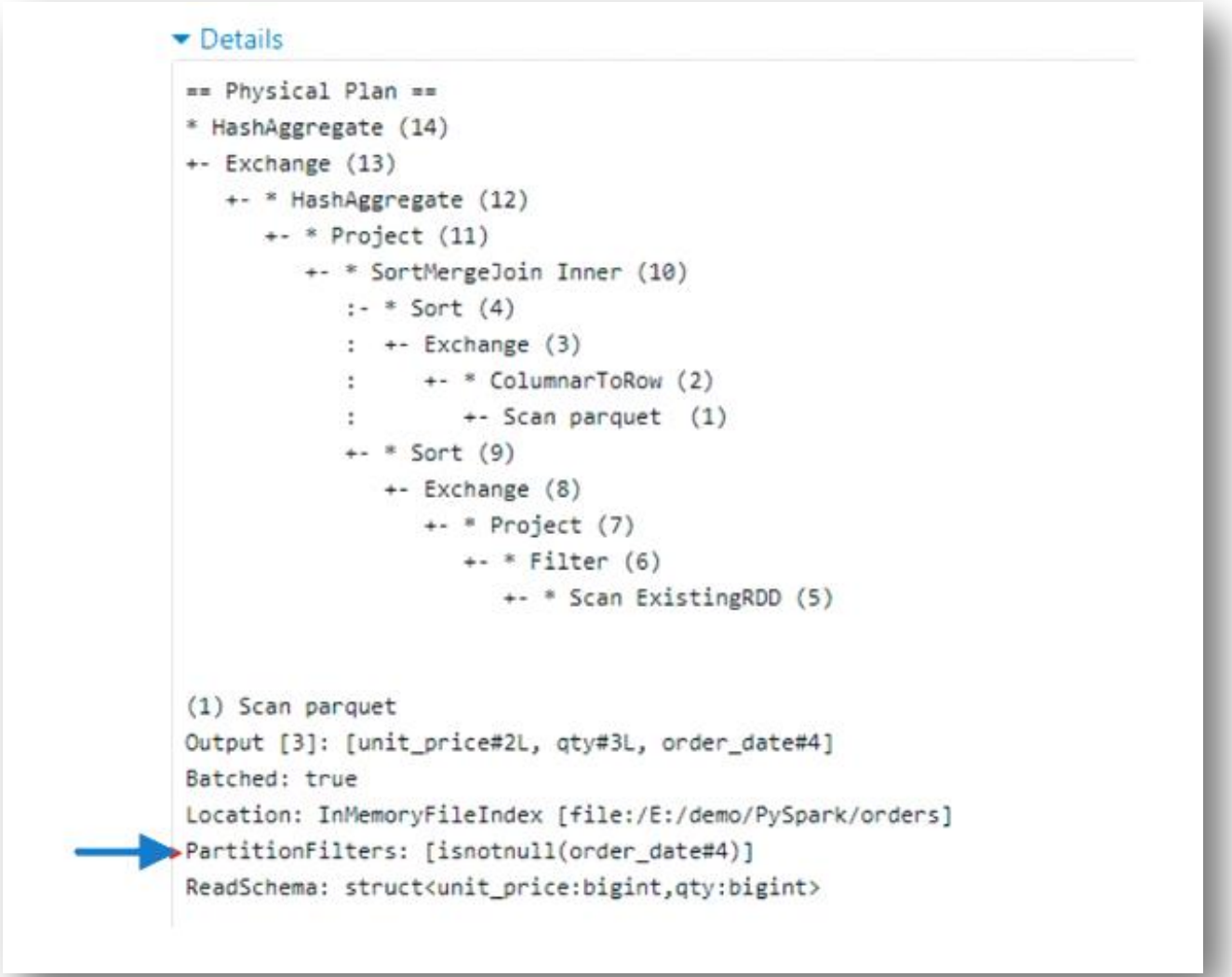


My Parquet files are partitioned on the order date column, but that partitioning does not benefit me. If you see the scan parquet step in the diagram, you can see the number of files read four.





The physical plan shows the partition filter step, but it is not pruning any partitions because we do not have any filter condition on the order date.



If you look at the SQL query, we are filtering for year and month columns on the dates table. It means, we want to read all the partitions for February 2021. But Spark is reading other month partitions also. The best case is to read only February 2021 partitions and leave all other partitions. However, Spark is not applying partition pruning here. Why? Because the filter conditions are on the dates table. They are not on the orders tables.

```
SELECT year, month, sum(unit_price * qty) as total_sales
FROM orders JOIN dates ON order_date == full_date
GROUP BY year, month
WHERE year=='2021' and month=='02'
```

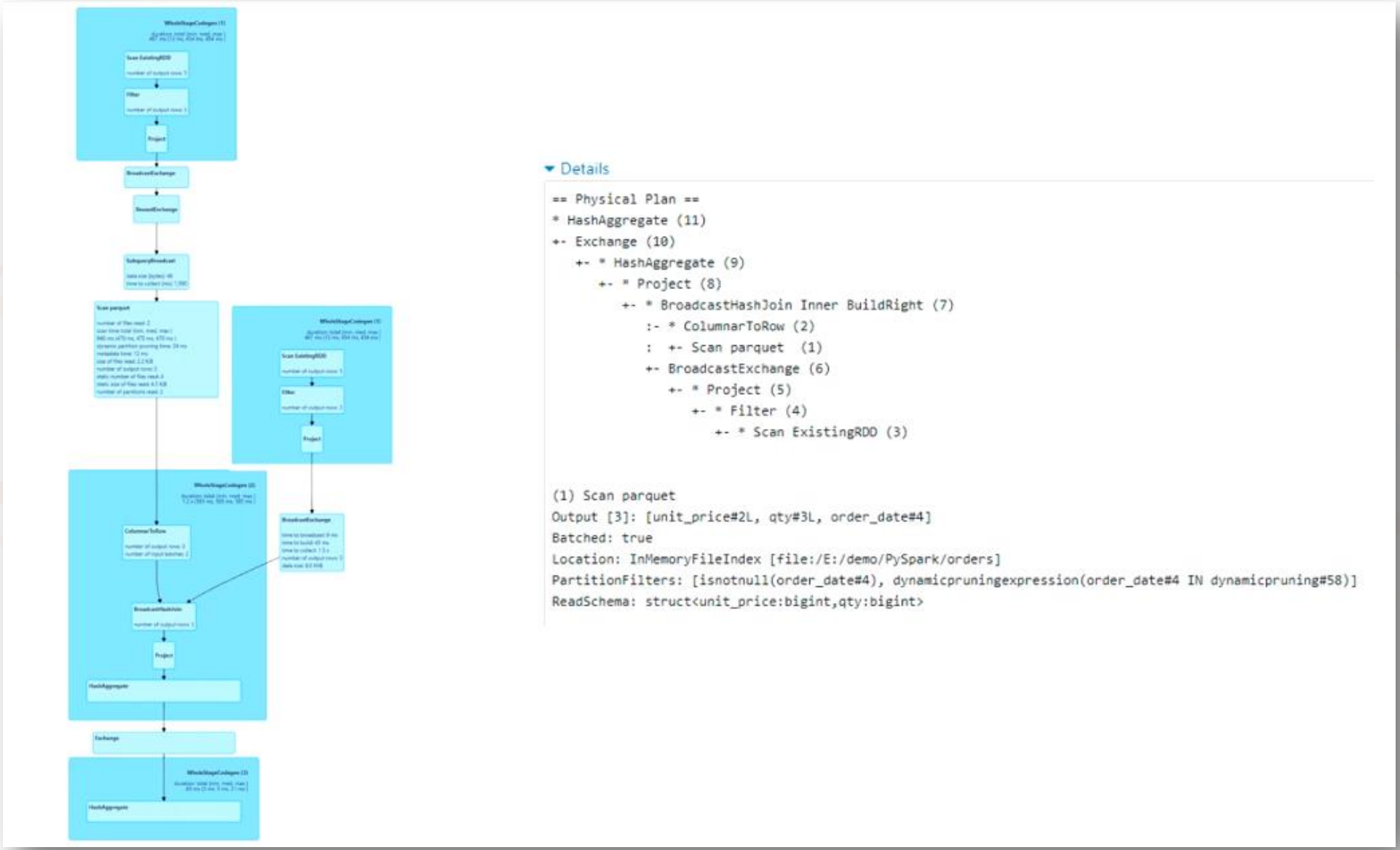

To improve this you must do two things:

1. Enable Dynamic Partition Pruning feature.
2. Apply broadcast on the dimension table

The Dynamic partition pruning is enabled by default in Spark 3 and above. So I do not need to do anything because I am using Spark 3. But I must apply a broadcast to my dimension table. So I modified my code, and it now looks like this shown below. This code is the same as earlier. But now I have applied broadcast() to the date_df.

```
join_expr = order_df.order_date == date_df.full_date

order_df
    .join(f.broadcast(date_df), join_expr, "inner") \
    .filter("year==2021 and month==2") \
    .groupBy("year", "month") \
    .agg(f.sum(f.expr("unit_price * qty"))
    .alias("total_sales"))
```

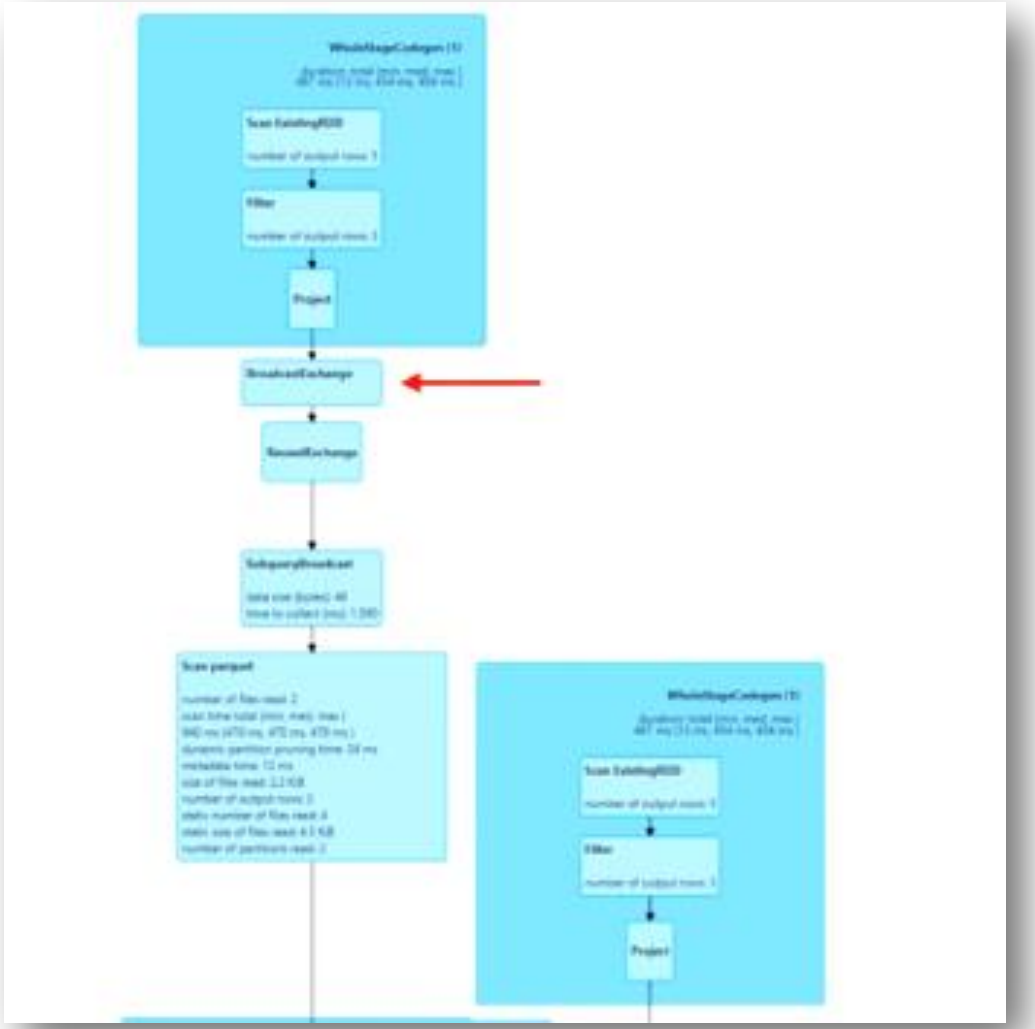

I ran the modified code once again, and here is the new execution plan.



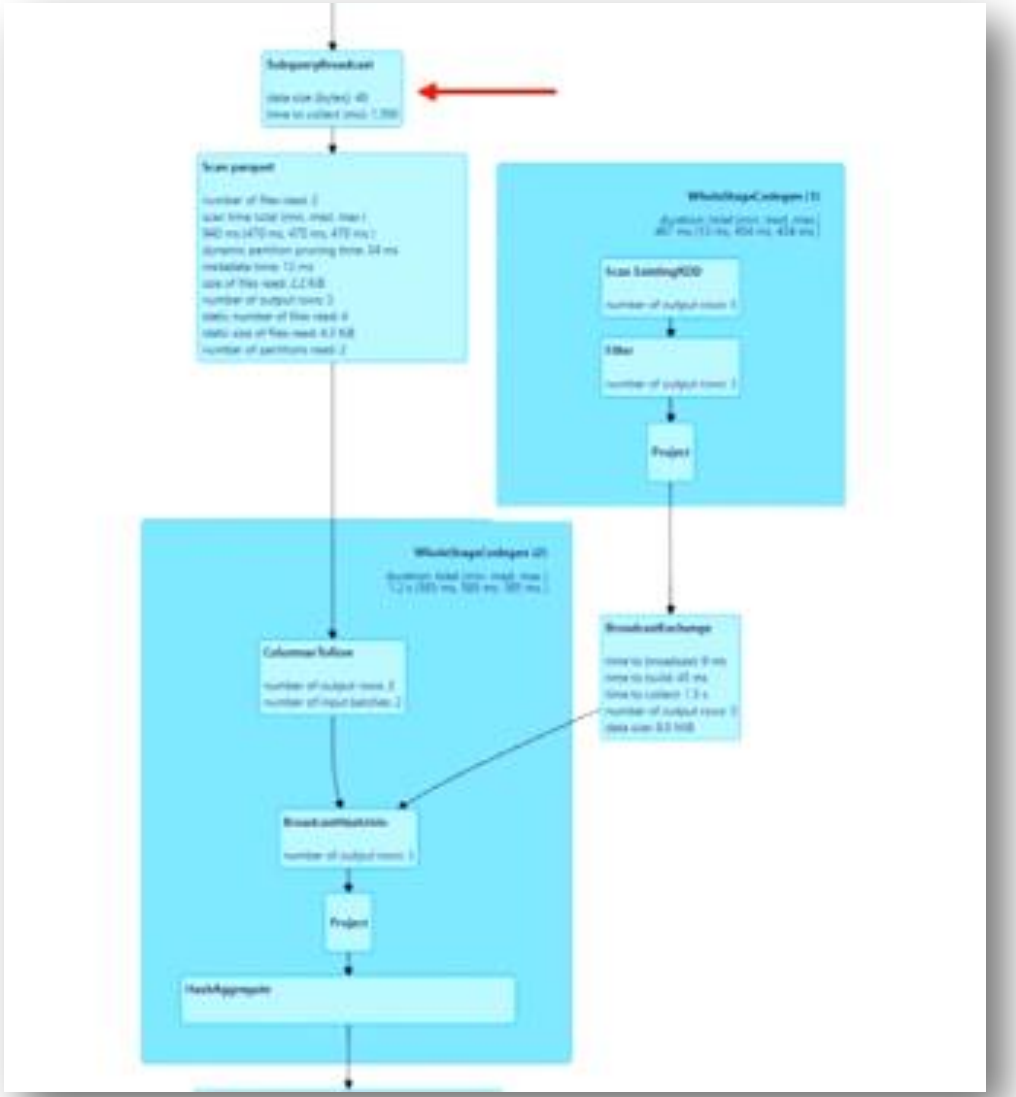
Details

```
== Physical Plan ==
* HashAggregate (11)
+- Exchange (10)
   +- * HashAggregate (9)
      +- * Project (8)
         +- * BroadcastHashJoin Inner BuildRight (7)
            :- * ColumnarToRow (2)
            :  +- Scan parquet  (1)
            +- BroadcastExchange (6)
               +- * Project (5)
                  +- * Filter (4)
                     +- * Scan ExistingRDD (3)


(1) Scan parquet
Output [3]: [unit_price#2L, qty#3L, order_date#4]
Batched: true
Location: InMemoryFileIndex [file:/E:/demo/PySpark/orders]
PartitionFilters: [isnotnull(order_date#4), dynamicpruningexpression(order_date#4 IN dynamicpruning#58)]
ReadSchema: struct<unit_price:bigint,qty:bigint>
```

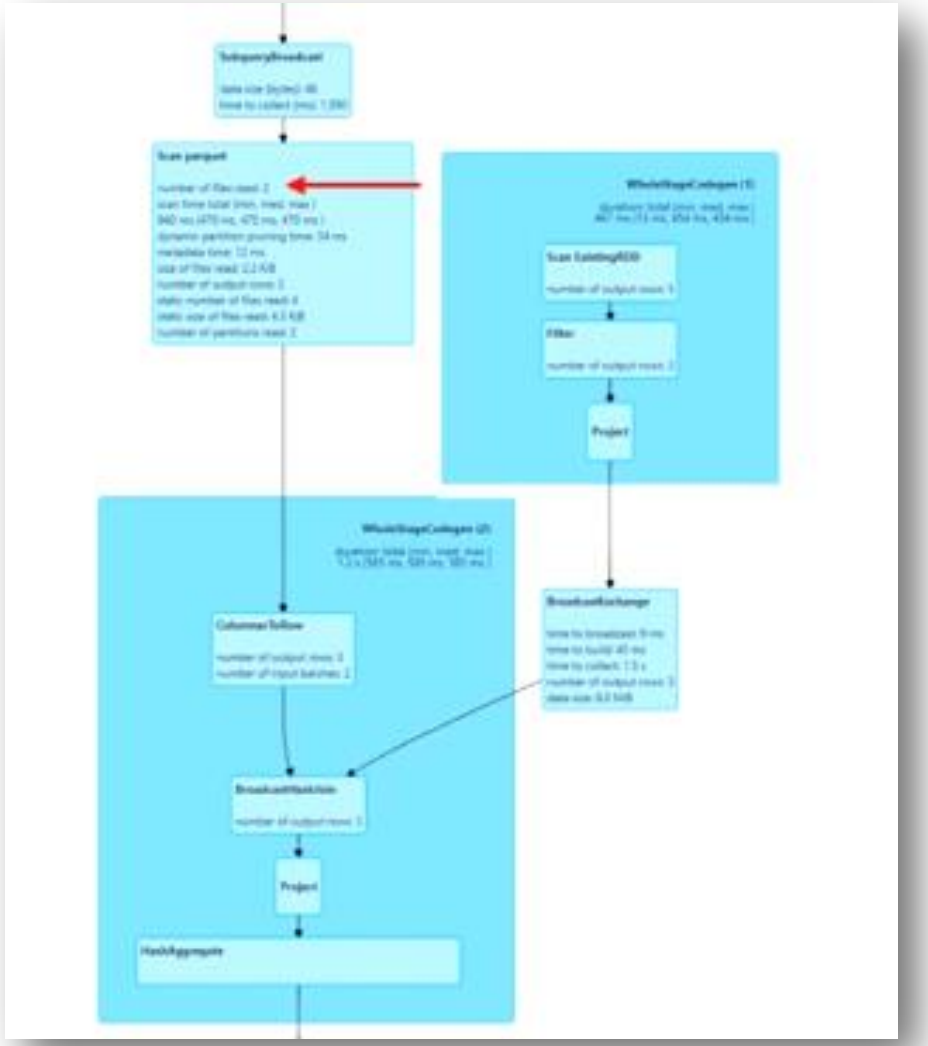

Let's look at the plan and see what is happening here.
My dates df is going for a broadcast exchange.



But then, Spark creates a subquery from the broadcasted dates_df and sending it as an input to the parquet scan.



And you can see the number of files read. We are now reading only two files. So what happened here? Spark applied partition pruning, and now it is reading only February 2021 partitions. If you look at the physical plan and check out the scan parquet details, you will see a dynamic pruning expression.

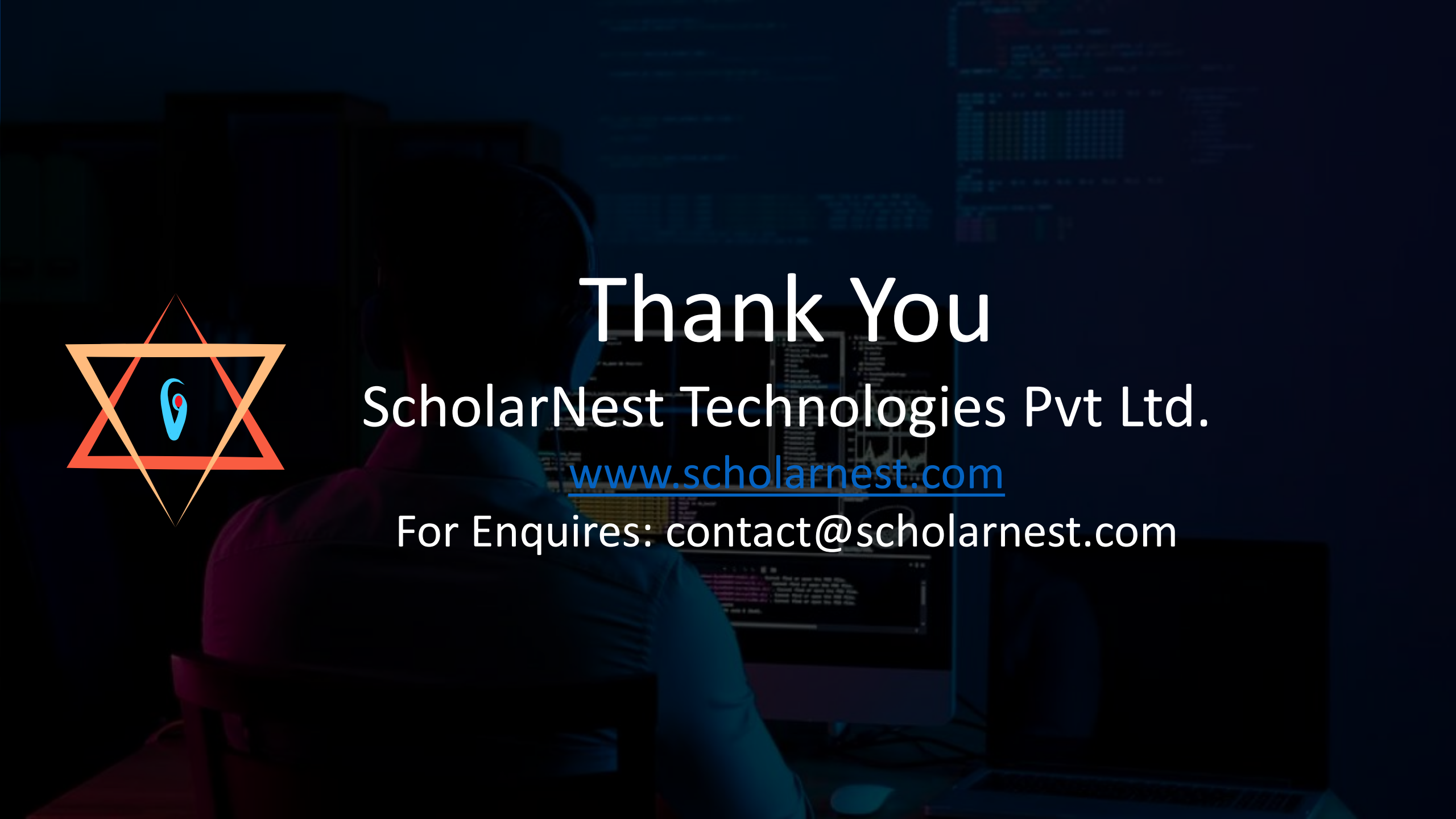


Spark Dynamic partition pruning can take a filter condition from your dimension table and inject it into your fact table as a subquery.
Once a subquery is injected into your fact table, Spark can apply partition pruning on your fact table.

It works like magic. But using this feature is not straightforward. You must understand the following things:

1. You must have a fact and dimension-like setup. When I say fact and dimensions, I mean one large table and another small table.
2. Your large table or the fact table must be partitioned so Spark can try partition pruning.
3. You must broadcast your smaller table or the dimension table. If it is smaller than 10 MB, then Spark should automatically broadcast it.

However, you should make sure that your dimension table is broadcasted.
If you meet all three conditions, Spark will most likely apply dynamic partition pruning to optimize your queries.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com