WIZELINE ACADEMY

Grow your career:
Free courses in Artificial Intelligence, Software Development, User Experience, and More

WIFI: Wizeline Academy
Password: academyGDL

@WizelineAcademy

/WizelineAcademy

academy.wizeline.com

Get notified about courses:
tinyurl.com/WL-academy

Joins and Distributed Variables

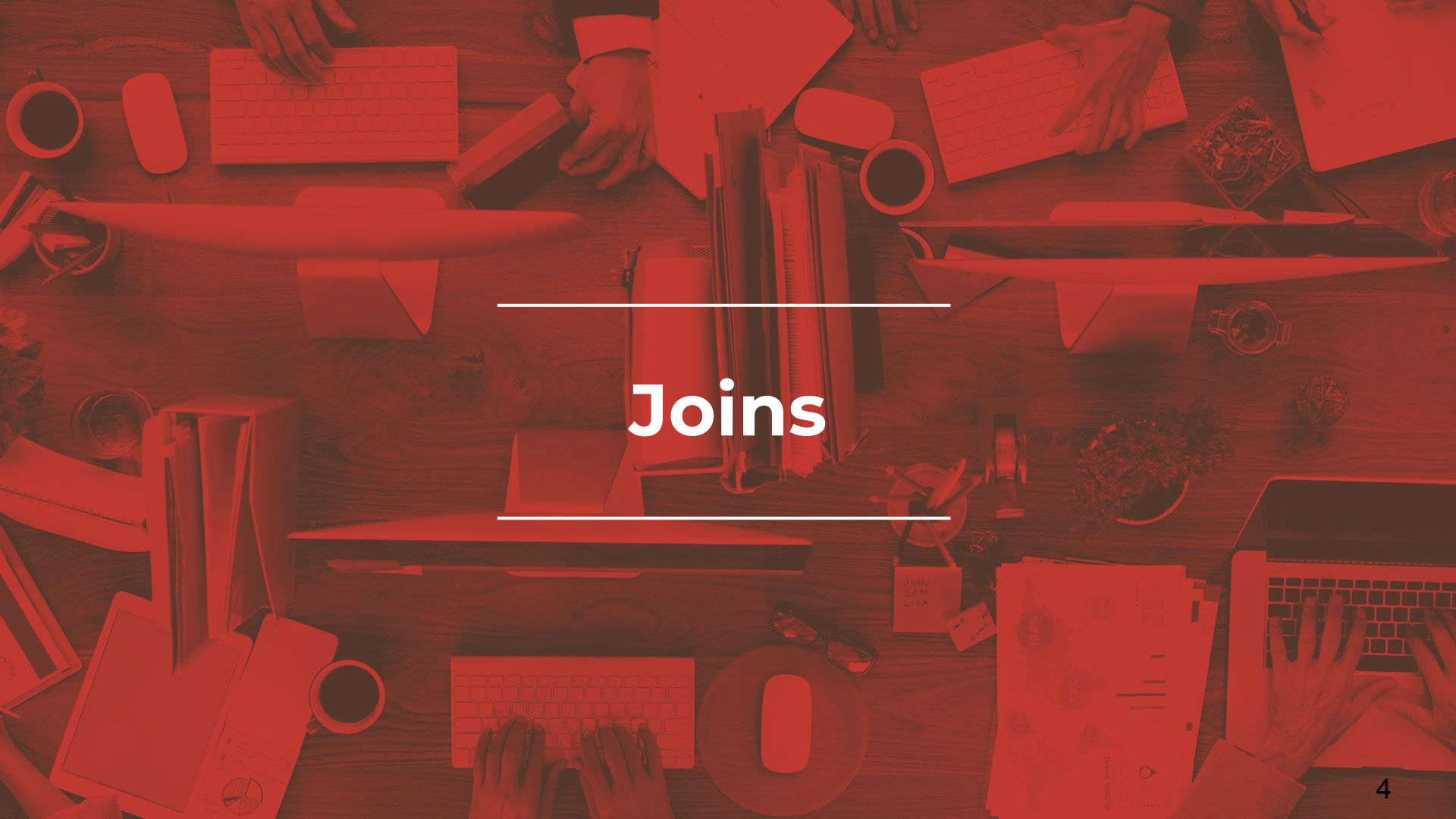
Structured Data

Week 4 - Session 1/2

WIZELINE

Agenda

- **Joins**
  - Shuffle Hash Join
  - Broadcast Join
  - Join Optimizations
- **Distributed Shared Variables**
  - Broadcast Variables
  - Accumulators



Joins

Joins

Join Tasks and Their Importance

- A join operation brings together two sets of data by comparing the value of one or more keys in both sets. This comparison allows evaluating whether or not it's possible to bring the sets together.

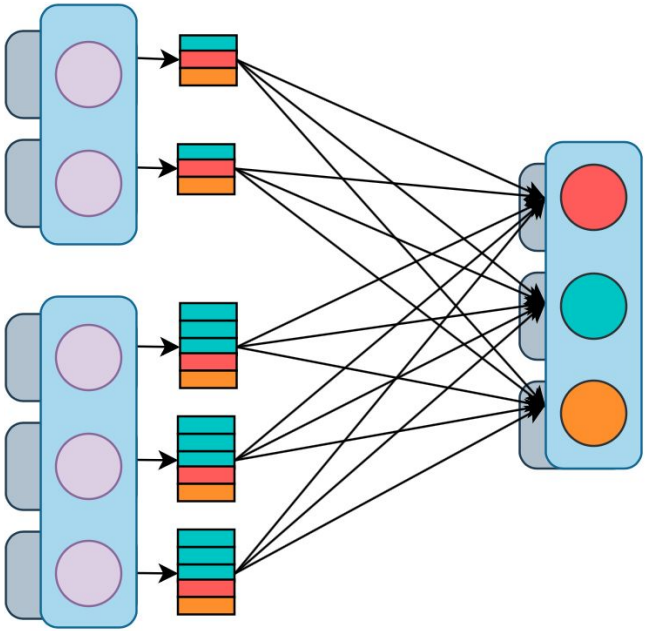
- Join is one of the most expensive operations you commonly use in Spark, so it is worth doing whatever is possible to shrink your data before performing a join.

- Join operations are supported in both **Spark Core** and **SQL**.

- The task of joining data requires large network transfers because of the way Spark approaches cluster communication during joins. It happens in two different ways: It either performs a **shuffle join**, which results in an all-to-all communication, or a **broadcast join**.

Shuffle Hash Join

- A **hash join** is the most basic type of join; this operation goes back to the **Map Reduce** fundamentals.



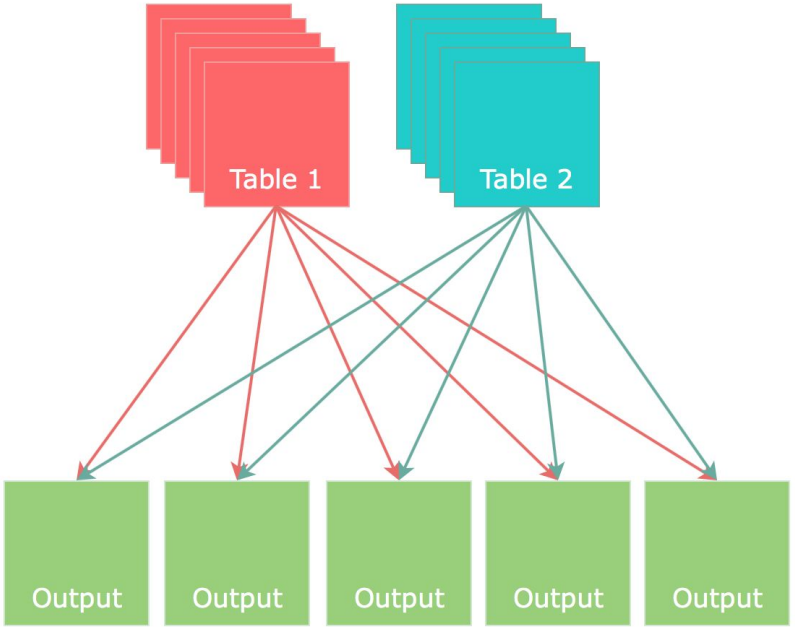
Good if **both** datasets are **large.**

Best when datasets:

- Are evenly distributed by the key used in the join.
- Have an adequate number of keys for parallelism relative to the dataset size.

Shuffle Hash Join

As a MapReduce Problem



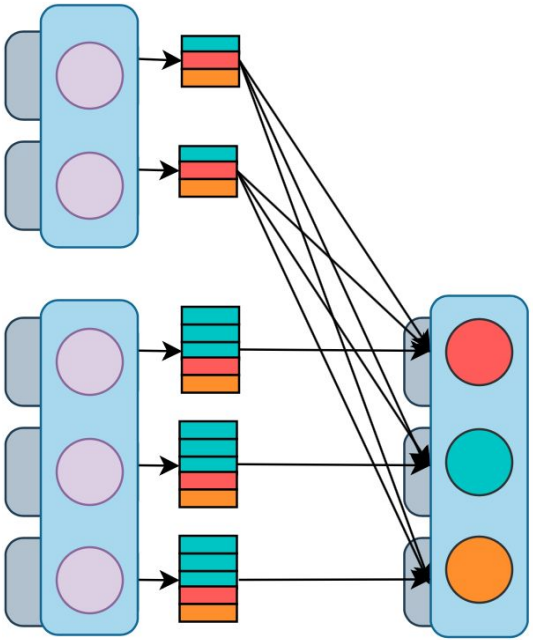
Map data sources and use the fields in the join as an **outputkey.**

Shuffle datasets by the **outputkey**.

Reduce by joining the datasets with any row that has the same **outputkey.**

Broadcast Join

- A **broadcast join** is a type of join that pushes the smallest dataset to every working node to reduce execution time.



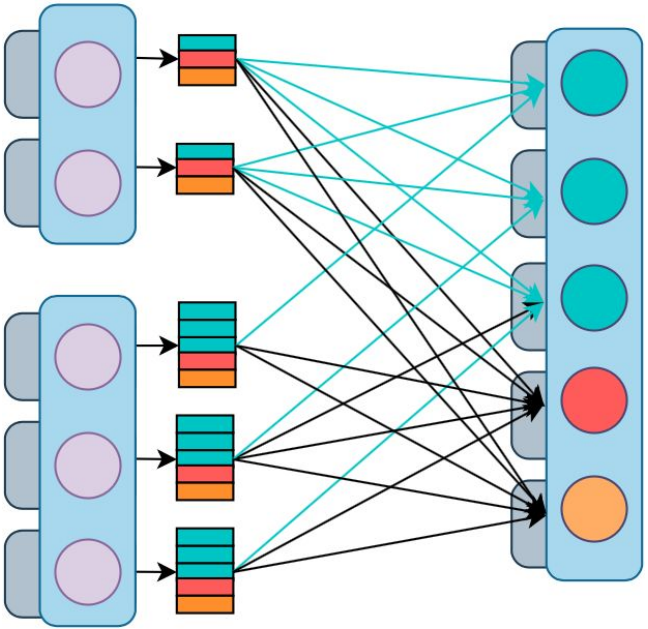
Good if the **smaller** datasets are **small enough to fit in memory.**

If one of your datasets fits in memory, it is always beneficial to do a **broadcast hash join**, because **it doesn't require a shuffle**.

Sometimes, Spark SQL will be smart enough to configure the broadcast join itself.

Hybrid Join

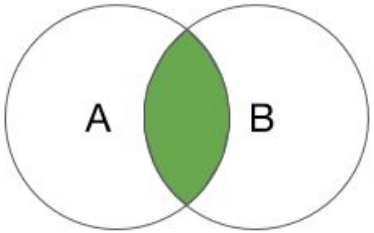
- A **hybrid join** is included in the execution plan of a join when Spark detects that the same key is used several times.



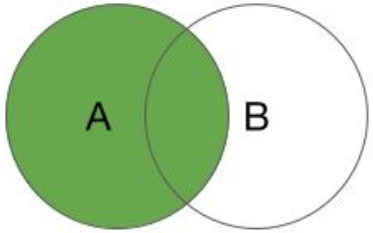
Broadcast popular keys, **shuffle** the rest.

- A hybrid join is part of the Adaptive execution plan introduced in Spark 2.0, which is one of the ways to tackle the data skew problem **(more on that later...)**

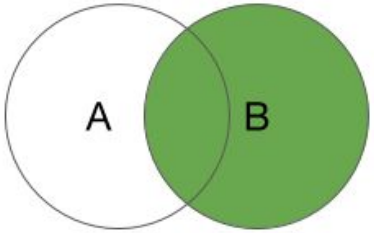
SQL Join Types



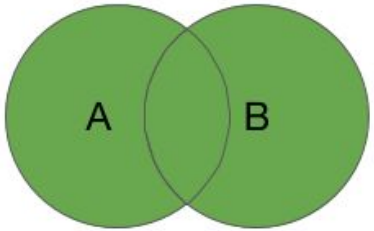
INNER JOIN



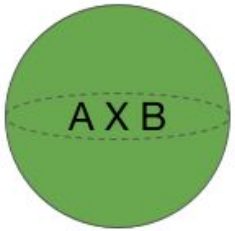
LEFT OUTER JOIN



RIGHT OUTER JOIN



FULL OUTER JOIN



CARTESIAN (CROSS) JOIN

SQL Join Types

Join Type	Description	joinType (Spark)
Inner Join	Keeps rows whose keys **exist in both** right and left datasets.	`inner`
Outer Join	Keeps rows that have **either right or left** datasets.	`outer`, `full`, `fullouter`
Left Outer Join	Keeps rows that have a key **in the left** dataset.	`left`, `leftouter`
Right Outer Join	Keeps rows that have a key **in the right** dataset.	`rightouter`, `right`
Left Semi-Join	Keeps the rows **in the left, keeping only the left** dataset **whose keys appear** in the right dataset.	`leftouter`, `left`
Left Anti-Join	Keeps the rows **in the left, keeping only the left** dataset **whose keys do not appear** in the right dataset.	`leftanti`
Natural Joins	Performs a join by implicitly matching the columns that have the same names in both datasets.	Special case for `inner`, `leftouter`, `rightouter`, `fullouter`
Cross Join	Matches **every row in the left** dataset with **every row in the right** dataset.	`cross`

Join Operations

You can join two datasets by using the join operators with an optional join expression.

Join operators

`join()`, `joinWith()`, `crossJoin()`

Join expression

```
df1.join(df2, df1.col("df1Key") === df2.col("df2Key"))
df1.join(df2).where(df1.col("df1Key") === df2.col("df2Key"))
df1.join(df2).filter(df1.col("df1Key") === df2.col("df2Key"))
```

You can also specify the **joinType** by passing the string representation shown in the previous table.

```
df1.join(df2, df1.col("df1Key") === df2.col("df2Key"), "inner")
```

Join Operations

People - Universities Examples

Try it out!

Import the notebook found on the URL below to see an example that uses join operations.

```
de-academy-1/notebooks/c7/c7-examples-exercises.json
```

Data Skew in Join Operations

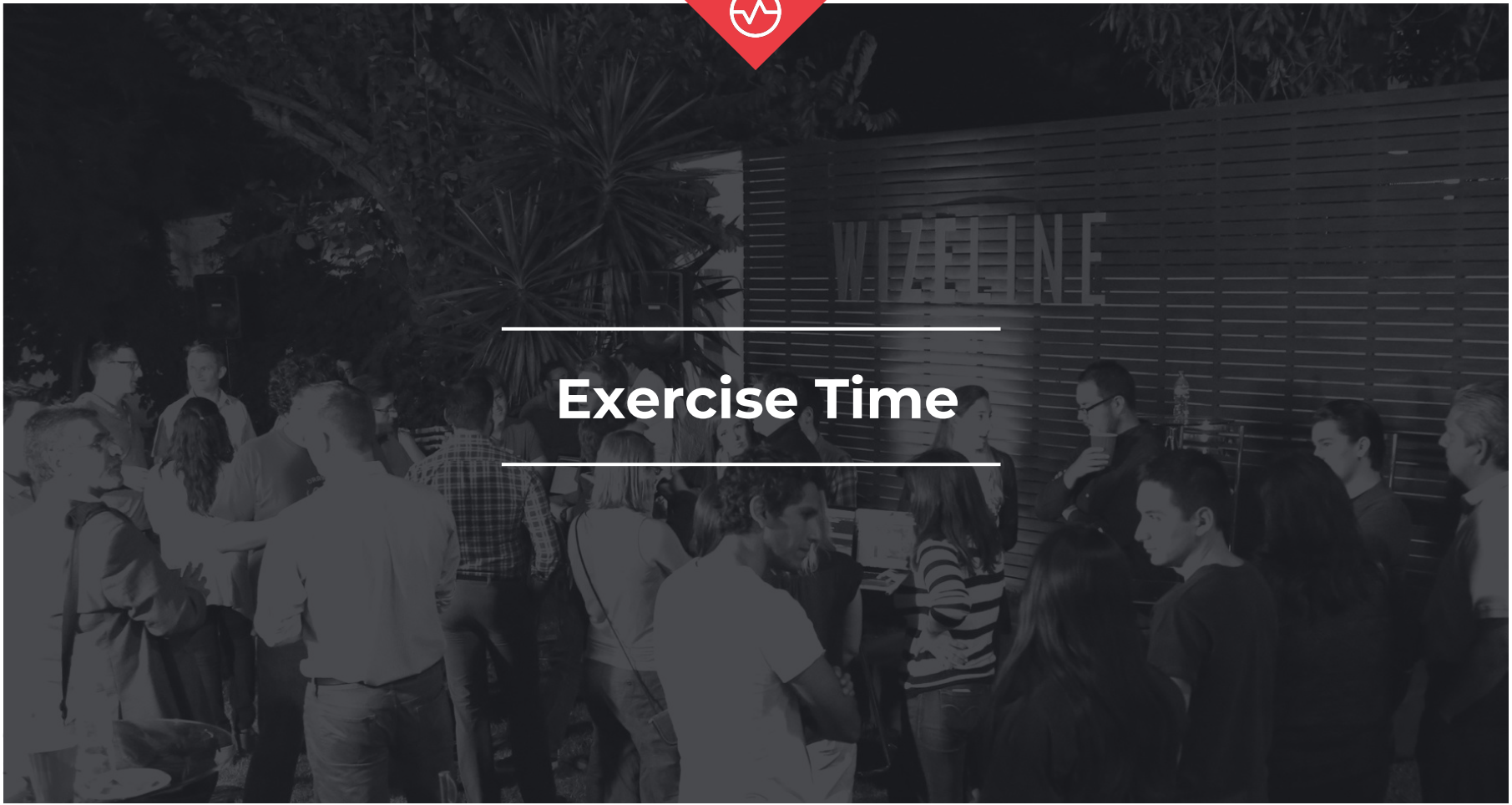
- Data skew happens when the amount of data contained in some partitions is much larger than the amount contained in others.
- Data skew is a common source of slowness in Shuffle Joins.
- Most of the tasks finish in a few seconds. However, those with the larger partitions take much more time, and as a result, job completion is time consuming.
- Jobs get bottlenecked by the amount of time that the worker node with the largest data partition takes to finish the task, rather than finishing tasks evenly across the cluster.

Join Performance Troubleshooting

Draw it all out!

- **Get to know your dataset!**
- Investigate how the **keys** of the join are **distributed.**
- Experiment with some **repartition** options beforehand and see if you can notice any increase in the execution speed.
- Explore the behaviour of your jobs in the **Spark UI.**

Comprehending how Spark performs joins can mean the difference between a job that completes quickly and one that never completes at all.



Exercise Time

Joins

Exercise

The Joins section of the **example notebook** has a task for you to complete. Look for the exercise at the end of the section.

```
de-academy-1/notebooks/c7/c7-examples-exercises.json
```

Joins Recap

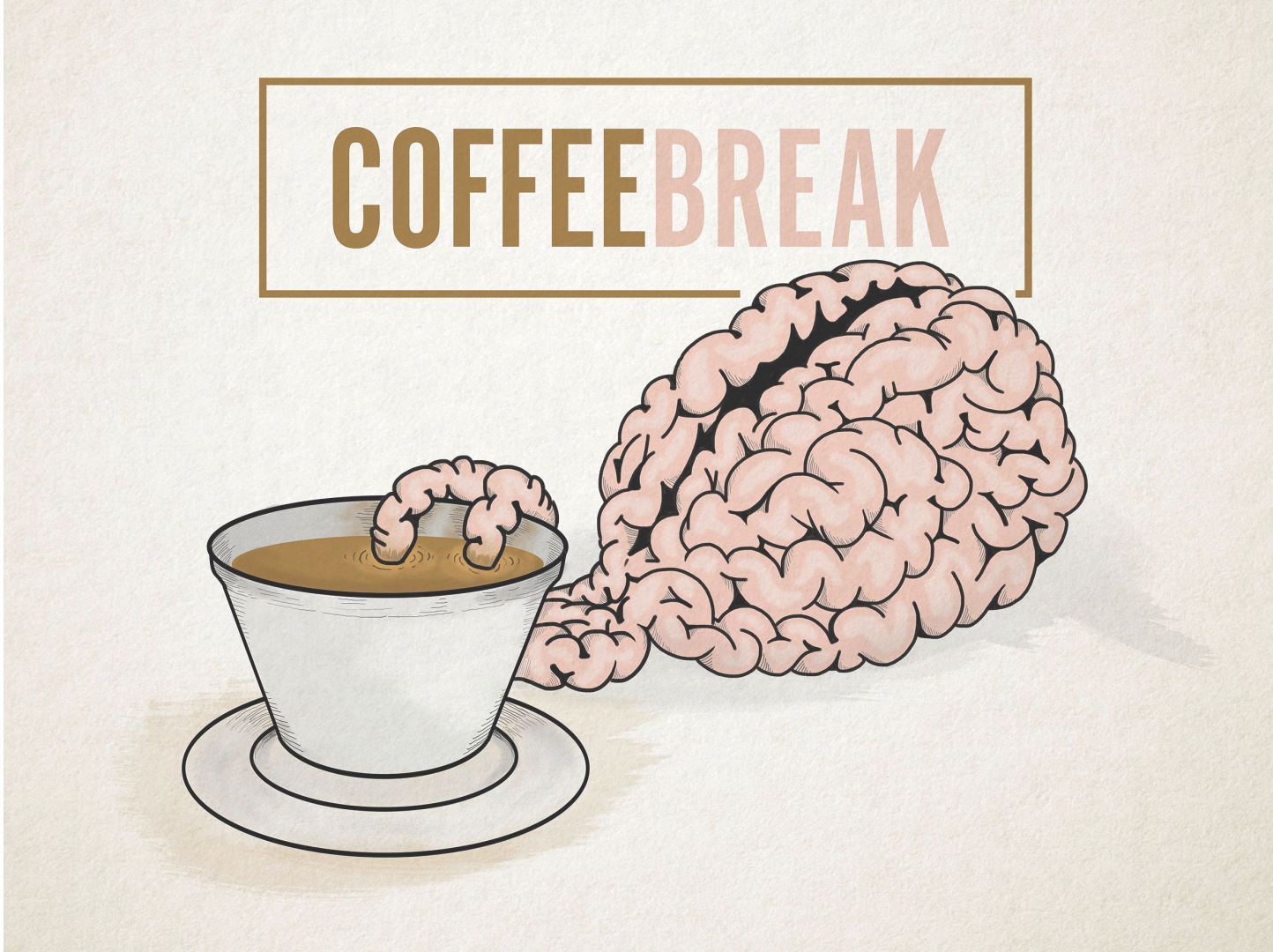
What We've Learned So Far

- What a **Join** operation is.
- How Spark handles Joins.
- How to perform Join operations using **joinTypes**.
- How the **data skew** problem appears.
- How to **troubleshoot** performance.

Q&A

Wait!

It's time for...



Broadcast Variables

Broadcast Variables

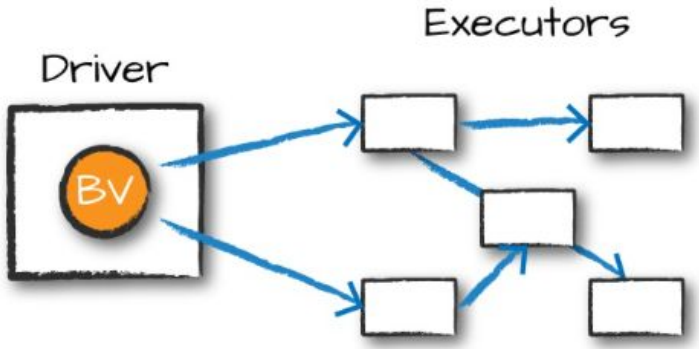
A **broadcast variable** is a way to share **immutable** values efficiently across the cluster.

Important features of Broadcast variables include:

- Variables are **lazily cached** and replicated on every machine in the cluster.
- Optimizations depend on the size of the variable being broadcasted.
- Broadcast variables can be used in the RDD, UDF or Dataset context.
- They enable efficient data broadcast.

Useful when:

- The same data is needed across multiple stages.
- It is important to cache data in a deserialized form.



How To Broadcast Variables

```
start_trips_count: org.apache.spark.sql.DataFrame = [start_station: int, trips_started_here: bigint]
```

```
val bc_start = sc.broadcast(start_trips_count)
bc_stc: org.apache.spark.broadcast.Broadcast[org.apache.spark.sql.DataFrame] = Broadcast(155)
```

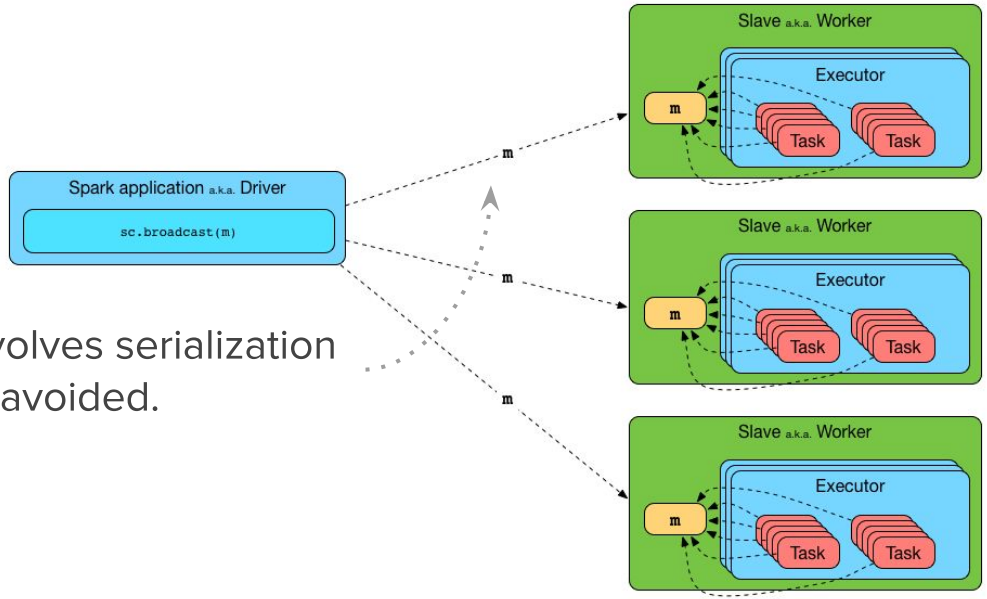
Use the **sparkContext** to broadcast variables.

```
bc_start.value.show()
+-------------+-----------------+
|start_station|trips_started_here|
+-------------+-----------------+
|           20|                2|
|           10|                3|
+-------------+-----------------+
```

Use the **`value`** method to retrieve the broadcasted variable.

Note: Broadcasted variables cannot be modified after their creation; if you do so, the change takes place only on one node.

Broadcast Variables Lifecycle



Tasks can access the broadcasted variable.

Broadcast involves serialization but shuffle is avoided.

When you are done with a broadcast variable, you should **destroy** it to release memory.

Before destroying a broadcast variable, you may **unpersist** it, which means asynchronously deleting cached copies of this broadcast on the executors.

Broadcast Variables Lifecycle

```
bc_start.id
```

```
res17: Long = 23
```

Every broadcasted variable has a unique identifier.

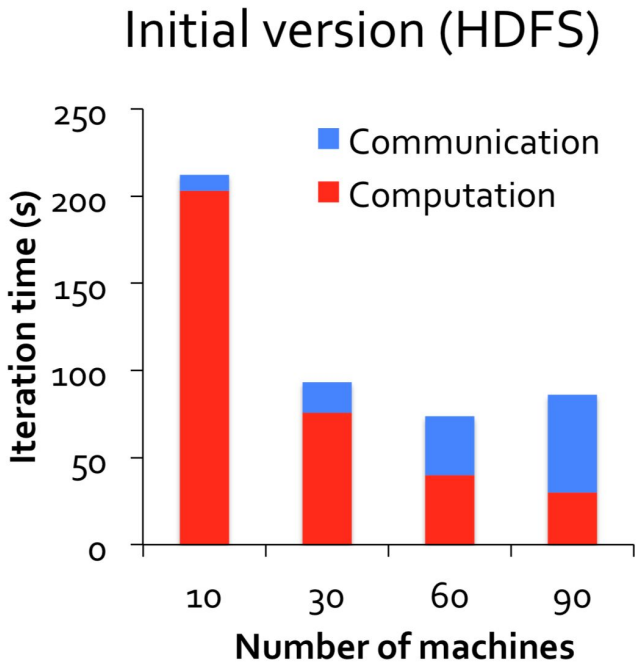
```
//Delete cached copies of this broadcast on the executors
bc_start.unpersist()
//Destroy all data and metadata related to this broadcast variable
bc_start.destroy()
```

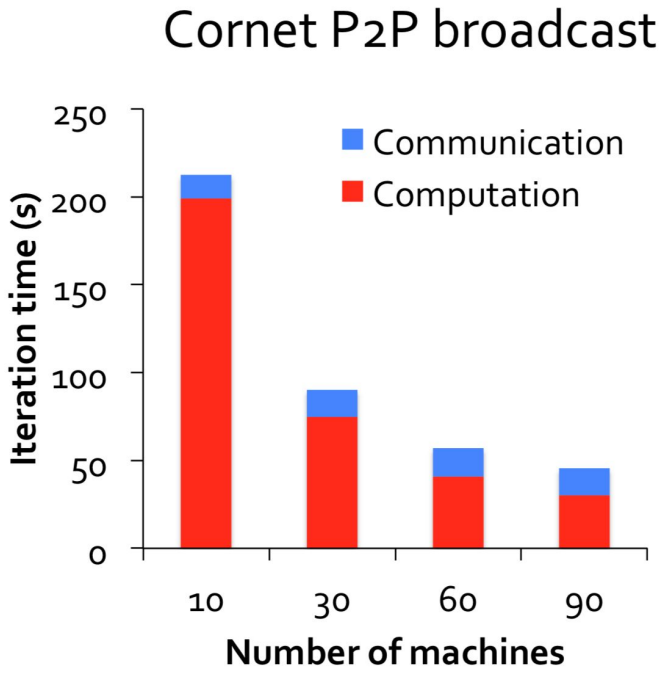
```
bs_start.id
```

```
<console>:27: error: not found: value bs_start
       bs_start.id
       ^
```

The variable is no longer available after calling **`unpersist`** or **`destroy`**.

Efficient Broadcast Methods





Broadcast Variables

GDL Neighborhoods Example

Go ahead!

Import the notebook found on the URL below to see an example that uses broadcast variables.

```
de-academy-1/notebooks/c7/c7-examples-exercises.json
```

Q&A

Accumulators

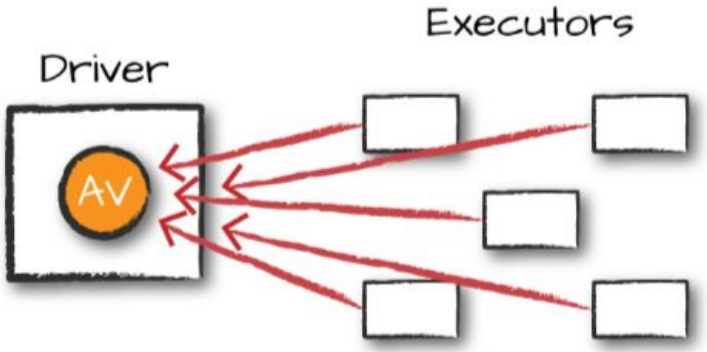
Accumulators

Accumulators are a way of **updating values** inside a variety of transformations and **propagating them** into the driver node in an efficient and fault-tolerant way.

Accumulators are variables that are **added only** through associative and commutative operations. Therefore, they can be efficiently supported in parallel.

Important features of Accumulators include:

- They are **mutable** and can be **safely updated.**
- They are useful for debugging purposes.
- They **preserve** the lazy evaluation model.
- They can be tracked using the **Spark UI.**



How To Use Accumulators

Use the **sparkContext** to initialize accumulators.

Provide a **name** so that it can be tracked on the Spark UI.

```
trips.show()
val trips_count = sc.longAccumulator("Trips Count")
```

```
+-------+------------------+------------------+--------+-------------+-----------+-------+
|trip_id|         start_date|          end_date|bike_num|start_station|end_station|user_id|
+-------+------------------+------------------+--------+-------------+-----------+-------+
|      0|2015-08-31 23:26:00|2015-08-31 23:39:00|     288|           10|         20|      1|
|      1|2015-08-31 23:11:00|2015-08-31 23:28:00|     221|           10|         40|      2|
|      2|2015-08-31 23:13:00|2015-08-31 23:18:00|     122|           20|         40|      3|
|      3|2015-08-31 23:10:00|2015-08-31 23:17:00|     134|           10|         50|      4|
|      4|2015-08-31 23:09:00|2015-08-31 23:22:00|     288|           20|         10|      1|
+-------+------------------+------------------+--------+-------------+-----------+-------+
trips_count: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 3866, name: Some(Trips Count), value: 0)
```

```
trips.foreach(x  => trips_count.add(1))
trips_count.value
```

```
res171: Long = 5
```

Use the **`value`** method to retrieve the contents of the accumulator - **only on the master node.**

Some Details about Accumulators

- Updates performed **inside actions** guarantee that the update is performed only once, even if some tasks are restarted.

- Updates performed **inside transformations** can apply the update more than once if the stages are re-executed.

Custom Accumulators

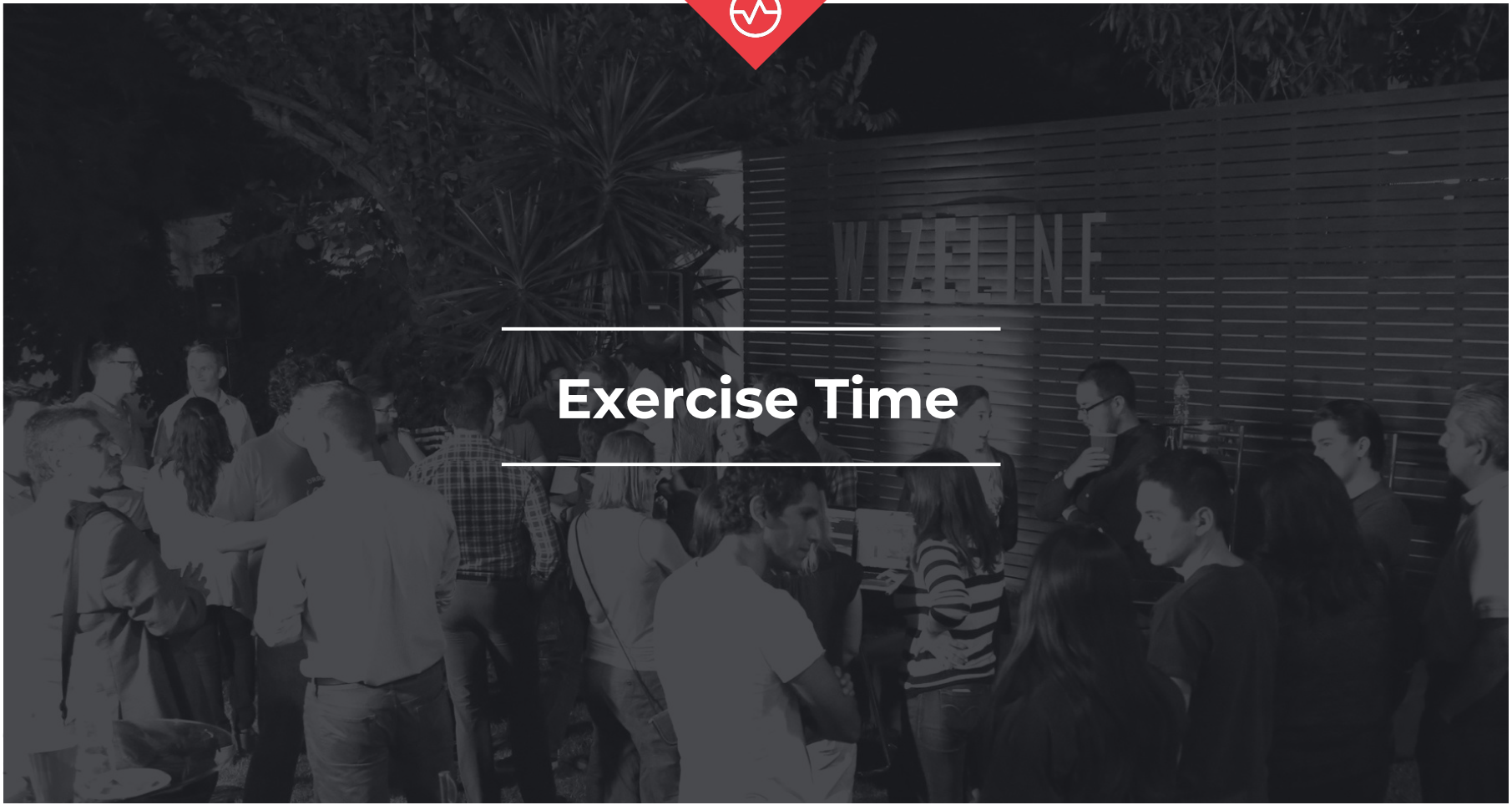
Spark supports accumulators of numeric type with the `longAccumulator` and `doubleAccumulator` constructs built in the sparkContext.

You can also create your own Accumulator types by using the `AccumulatorV2` abstract class (in Scala), which has several methods that have to be overridden:

- **`reset:`** Resets the accumulator to zero.
- **`add:`** Adds a value to the accumulator.
- **`merge:`** Merges another same-type accumulator with the one at hand.

Let's take a look at an implementation that uses custom accumulators.

`de-academy-1/notebooks/c7/c7-examples-exercises.json`



Exercise Time

Recap

What We've Learned So Far

Broadcast variables

- What a **broadcast variable** is.
- Lifecycle of a broadcast variable.
- Broadcast performance on Spark.

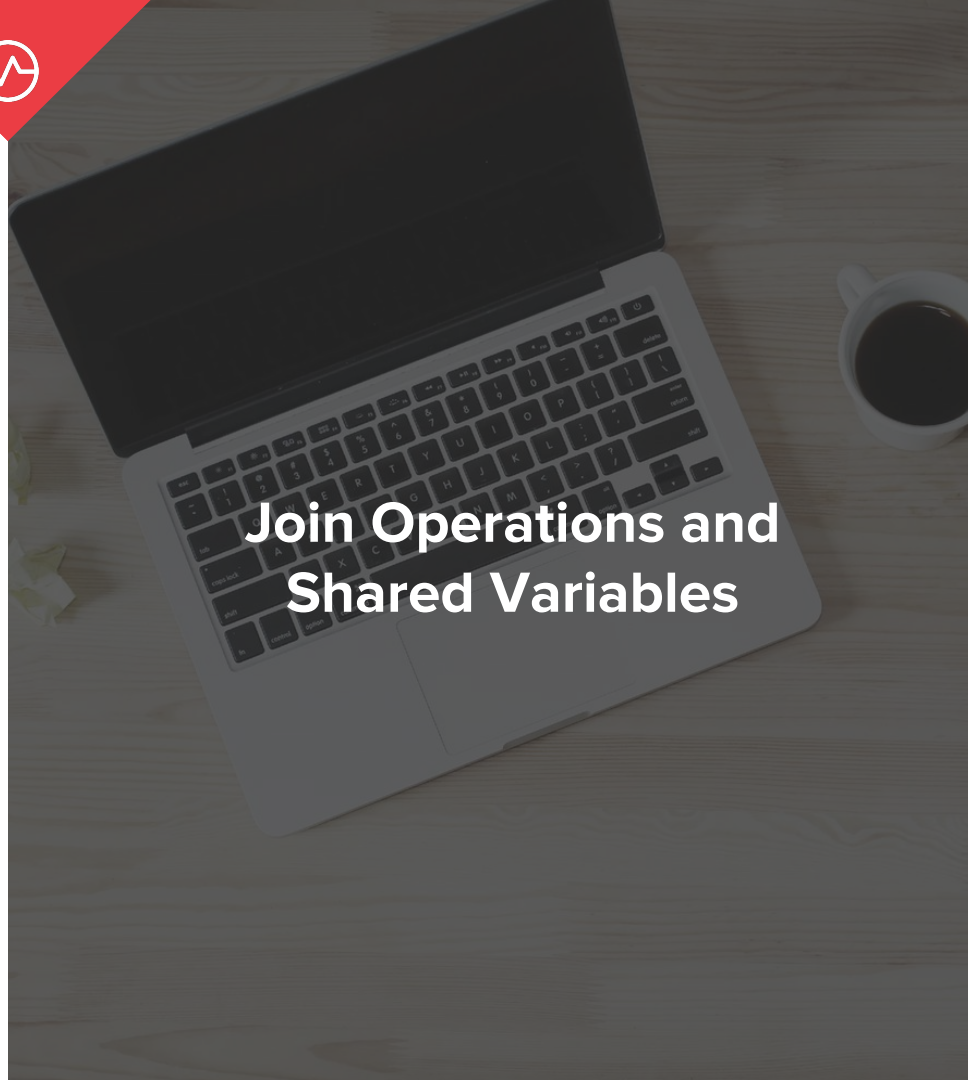
Accumulators

- What an **accumulator** is.
- How to use them.
- How to create a **custom type accumulator.**

Q&A

Assignment

To Work on Your Own at Home



Assignment

Join Operations and Shared Variables

Problems' Description

You must accomplish two different tasks. For the expected output, see the next slide.

1. Using the Alimazon Dataset, you must list the best clients information based on the number of orders placed and the largest amount of money spent.

2. Also, as a data caretaker, you need to verify the health of your database, so you must run a *health check report* on the Alimazon dataset and identify:
    - Any order that points to an invalid client.
    - The products that were sold without having enough stock.

Expected Output

Name	Type	Notes
client_id	String	UUIDv4
name	String	Less than 100 characters
gender	String	Either `male`, `female` or `null`
country	String	Valid [country code]
registration_date	TimeStamp	ISO 8601
total_transactions	Long	Count of transactions
total_amount	Double	Money spent
total_products	Long	Sum of quantity

Name	Type	Notes
product_id	String	UUIDv4
total_bought	Long	
total_sold	Long	
is_valid	Boolean	

Where can I get this presentation?

Slack Channel PDF

C7 - Joins and Distributed Shared Variables

Data Engineering Academy

Your feedback is very valuable to us!

goo.gl/forms/OrRVxGA7chtWoE5K2


THANK
YOU
WIZELINE®