* Spark Joins

How is join performed in spark?

=> Suppose we have two tables orderDF and productDF

1. Join Condition / Expression:

join_expr = order.product_id == product.product_id

joinType = "inner"

right table

... order_df.join(product_renamed_df, join_expr, "inner")

left table

- In this spark is going to take one row from your left data frame.
And evaluate the join expression for all the rows in right data frame to find a match.
Then it goes to the second row and repeats the same. And so on.

- The next step is to combine these matching records from the left and right data frames to create, new data frame

* orderDF

order_id	prod_id	unit_price	qty

productDF

prod_id	name	list_price	qty

When, we are performing join, simply displaying all the columns we get the output as this:

order_id | prodid | unitprice | qty | prod_id | name | list_price | qty

Now, suppose I don't want all these columns I am only interested in 4 columns

```
order_df.join(product_df, joinExpr, "inner") /
    .select("order_id", "name", "unit price", "qty")
    .show()
```

This code is not going to run. This is because

we have two "qty" the spark get = confused and throws this error.

But, when I am selecting all the columns Spark works fine, but Spark complaining about ambiguity when we explicitly refer a column?

⇒ The answer is simple. Every Dataframe column has a unique ID in the catalog. and spark engine always works using those internal ids.

These ids are not shown to us, are we expected to work with the column names.

However, the Spark engine will translate these columns names to ids during analysis phase.

When I refer to column name in any expression, then the Spark engine will translate the column names into column id internally.

And that's where it complains about ambiguity.

But when we are using select * it takes all the column ids & shows them.

So, column name ambiguity is common mistake in join operations

-We have two approaches to avoid these mistakes:

1) The first approach is to rename the ambiguous columns, even before joining these two data frames.

```
product_renamed_df = product_df.withColumnRenamed
                        ("qty", "reorder_qty")

order_df.join(product_renamed_df, join_exp, "inner")
        .select("orderid", "prodname", "qty")
        .show()
```

2) Second way is drop the ambiguous column after the join.

```
order_df.join(prod_df, join_exp, "inner")
        .drop(prod_df.qty)
```

.select("order_id", "name", "qty")

.show()

* Other Spark joins:

1] Outer join - Full outer :

The full outer join will bring all the records from both the sides, even if we do not have join

2) Left Join - Left Outer:

This will bring all the records from left side even if we do not have a matching pair on the right side:

3) Right join - Right Outer:

This will bring all records from right side

* Join Internals And Shuffle

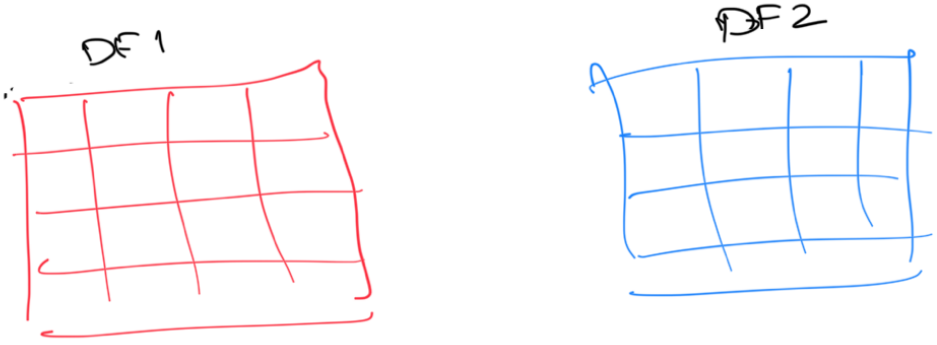
- Spark joins are one of the most common causes for slowing down your application.
- Spark implements two approaches to join your data frames -
  1] Shuffle Join
  2] Broadcast Join

1] Shuffle Join:

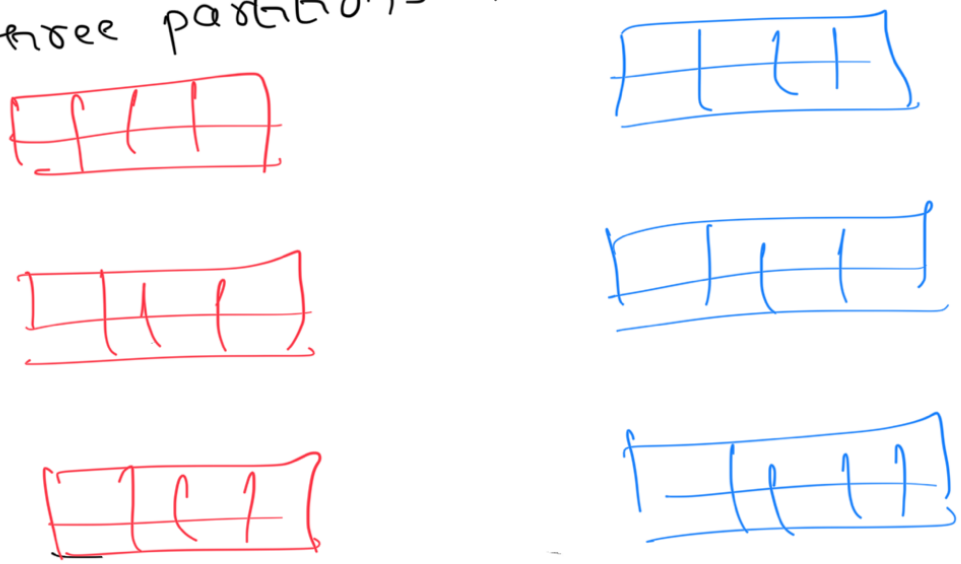
most commonly used join

- This is the most ... type.
- And the internal implementation goes back to the notion of Hadoop MapReduce implementation.

⇒ Let us assume, we have two dataframes:



Now, both these are partitioned, consider three partitions for each

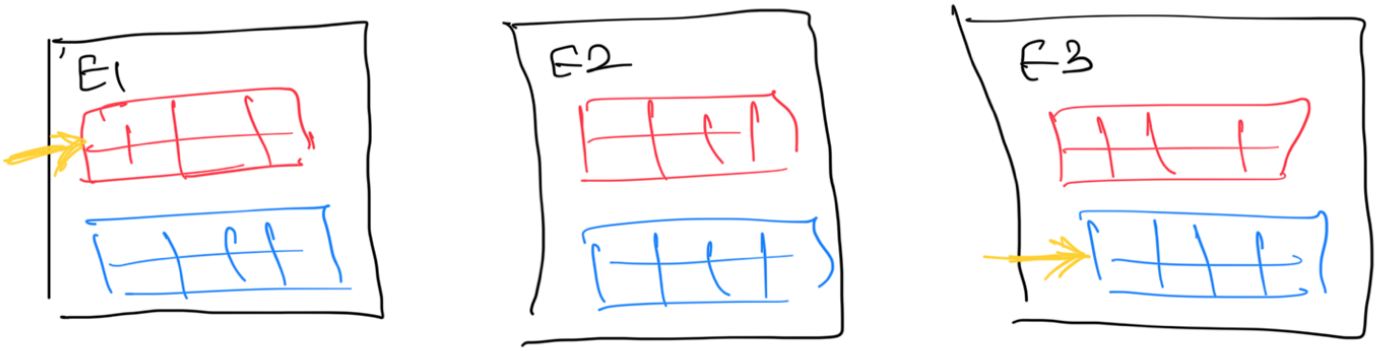


In reality, these dataframes could be huge and may have hundreds of partitions. But for simplicity and to understand the idea, let's assume you have 3 partitions.

Now, we want to join them using a key

right.

All these partitions are distributed across different executors in the cluster.
Let us assume we have 3 executors, and these partitions are equally assigned to these 3 executors



So, each executor owns two partitions - one from each dataframe

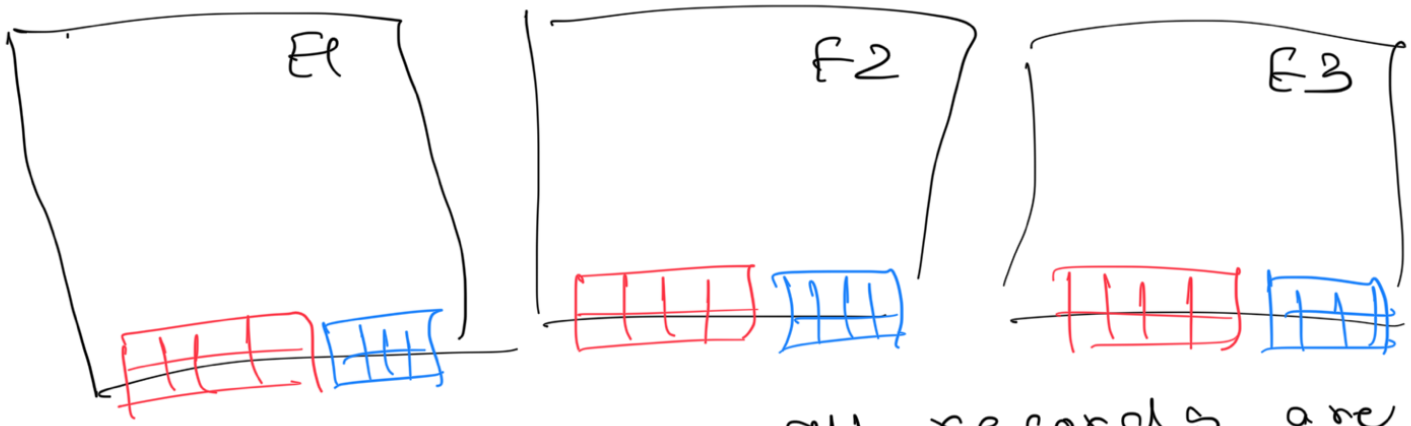
Can we perform a join?
- No we cannot. Why?

Because this key (first table in E1) has got a matching key here (second table partition in E3) (Marked in Yellow)
we cannot match them unless we have both the records on the same executor

So how can we do it?
- The join is performed in two stages

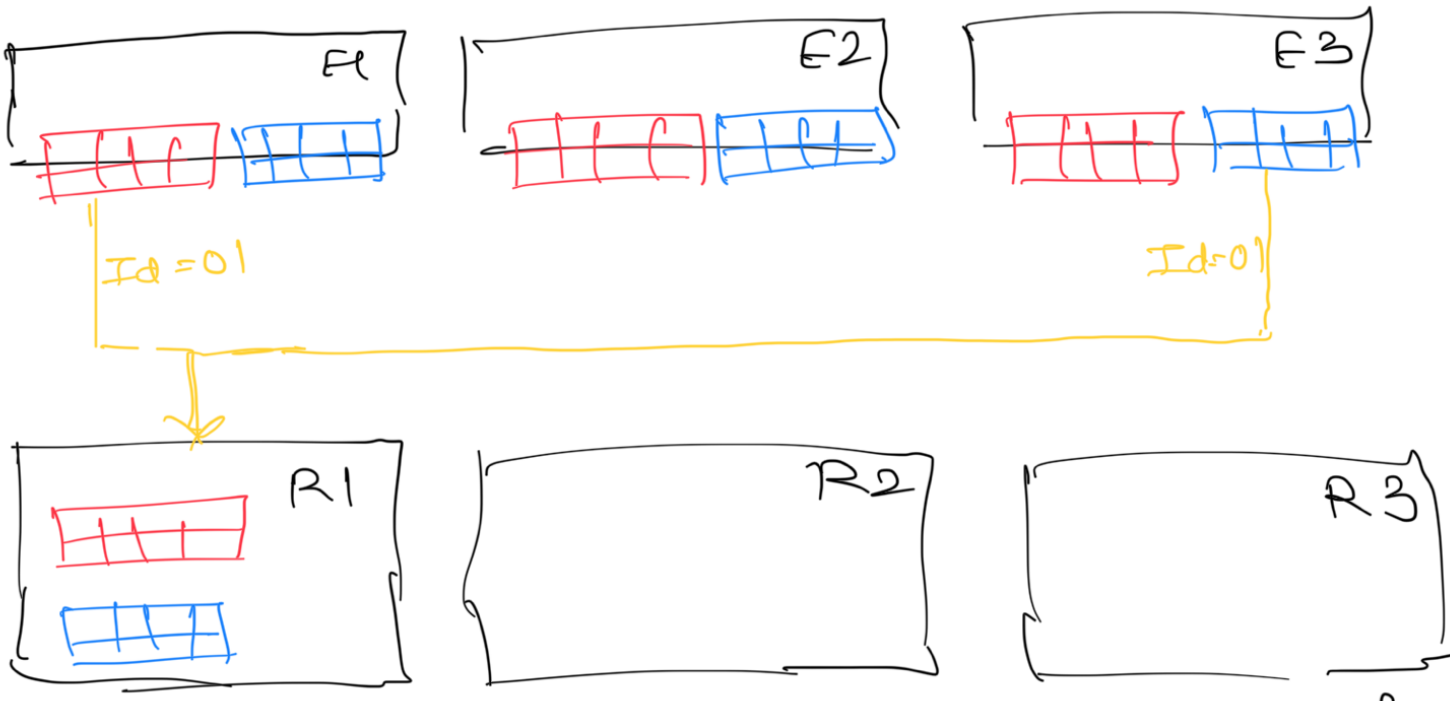
each executor will map

The first stage , [illegible]
these records , using the join key and
Send it to exchange .
And this is the map() phase of the
MapReduce implementation. So, let's call
these exchange as map exchange



So, in this first stage , all records are
identified by the key, and they are made
available in the map exchange to be
picked up by the Spark Framework
The map exchange is like a record buffer
at the executor
Now, the Spark framework will pick these
records and send them to reduce-exchange
A reduce-exchange is going to collect all
the records for same key. It means all
the records for the key-value 01 will go
to same exchange. These exchanges are

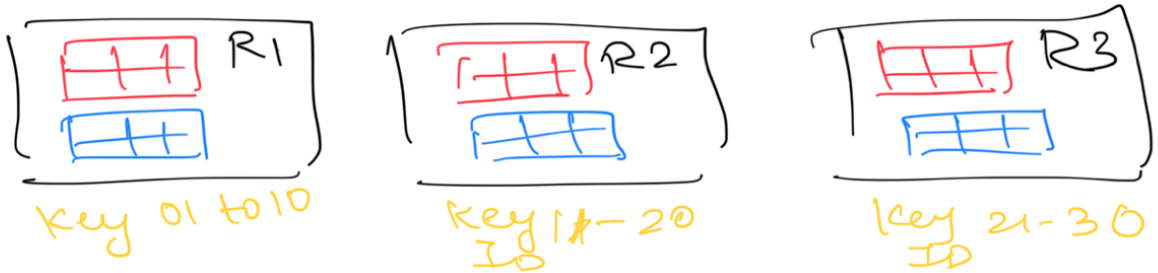
known as shuffle partitions.



And it's up to you to decide the number of Shuffle paratitions.

I selected here 3 shuffle paratitions, because I am running on three node cluster.

Let's assume I have 30 unique join keys So, in this case, each reduce-exchange should get 10 keys.

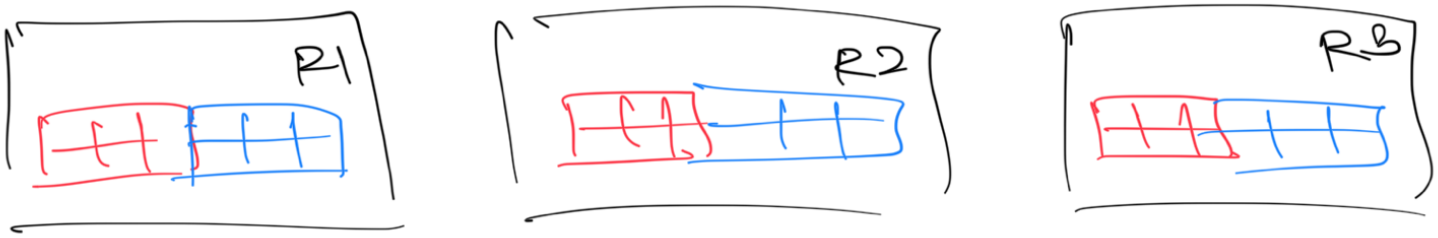


And all this transfer from map-exchange is what we call a

to reduce exchange [illegible]
shuffle operation.

This shuffle operation can choke your cluster network and slow down your join performance.

- Now, let's assume shuffle operation is complete. Now each exchange is self sufficient to join records. why?
Because all records for same key-value from both dataframes are present at same exchange.
Now, it is all about combining these records and create a new df partition
And, this second stage of applying a sort-merge join algorithm and finishing the join.



* Optimizing Your Joins :

Joining two DF can bring the following

two scenarios:

1) Large DF to Large DF – Shuffle join

2) Large DF to Small DF - Broadcast join

Common mistakes done while performing join:

1] Don't code like a novice:

We already learned that shuffle join will send your data from map exchange to reduce – exchange.

So filtering out unneccessary data before you perform a join will obviously cut down the amount of data sent for shuffle operation.

Example:

Suppose you have two datasets. The first one is global online sales, and second one is only for USA warehouse dataset.

You want to perform left join by order_id and it looks obvious to join them without any prior filtering.

You know orders from European

however, ju
cities are not going to match us cities
orders, and they will be filtered out
in join operation.
It makes a perfect sense to filter
for us orders from global_sales
even before we perform a join operation.
The point is straight, look for an opportunity
to reduce DF size. Smaller the DF will
result in smaller shuffle and faster join.

2] Shuffle Partitions and Parallelism:
What is the maximum possible parallelism
for my join operations?
- The first most common limit is the number
of executors. If you can run the job
with 500 executors, then that's your
maximum limit.
- The second limit comes from the number
of shuffle partitions.
If you have 500 executors, but you
configured 400 shuffle partitions,
then your maximum is limited to 400? Why?
400 partitions to

Because you have only 200 [illegible]
process in parallel.

- The third limit comes from the number of unique join keys.
If you have only 200 unique keys, then you can have only 200 shuffle partitions.
Even if you define 200 shuffle partitions only 200 of those will have data and others will remain blank.

3] Key Distribution:

The next important thing is to look for the distribution of your data across the keys.
For example,
You are joining the sales and product Dataframe. consider you have 200 products. So, all your sales transactions are joined with 200 products resulting in 200 shuffle partitions.

[illegible] fast-moving

However, you may have some fast moving product and some slow moving products

So, a fast-moving product should have a lot transactions and all those will land in one partition because they all belong to same product key.

If you compare size of your fastest-moving product vs your slowest moving product, the difference could be significant.

The spark task which is sorting and joining the larger partition, may take a lot of time, whereas the smaller partition will get joined very quickly.

However, your join operation is not complete until all your partitions are joined

The point is straight, watch out for the time taken by individual task and amount of data processed by join task.

If some tasks are taking longer than other [illegible] have a way to fix your

task and you have a long journey or have some hack to break the larger partition into more than one partition.

2] Broadcast Join:

Suppose we have two dataframes. Sales dataframe and product dataframe

Sales dataframe is huge and it has millions of records. I am assuming that these records are partitioned and you have 100 partitions and these partitions are distributed across 100 executors.

Your product dataframe is just 200 records and it is worth 2 MBs. You have single partition and it is stored at one executor.

- Now, let's assume you are joining these two Dataframes, using a shuffle join So, you can expect 200 unique keys for your shuffle is going to

this join. ...
Create 200 shuffle partitions. And all the data from these 100 executors will be shuffled to these 200 shuffle partitions.

What does it mean? You are sending all those millions of records over the network for a shuffle operation.

- However, we have another much better alternative.

Instead of sending this data from 100 executors, we can dispatch this product table to all these executors.

This dataframe is small, just 2MB.

So even if we send it to all 100 executors, we send 2x100 = 200MB of data over the network, which is a lot smaller than the sales table.

- And these executors can smoothly perform a partitioned join because they have a list of all the products.

- This approach is known as broadcast

join. Why?

Because we are broadcasting the smaller table to all executors.

This broadcast is an easy method to eliminate the need for a shuffle.

However, this approach works only when your dataframe is small.

It could be 1GB or 2GB.