Ch. 8 - Joins

Join Expressions

A join brings together two sets of data, the left and the right, by comparing the value of one or more *keys* of the left and right and evaluating the result of a join expression that determines whether Spark should bring together the left set of data with the right set of data. The most common join expression, an ***equi-join,*** compares whether the specified keys in your left and right datasets are equal. If they are equal, Spark will combine the left and right datasets. The opposite is true for keys that do not match; Spark discards the rows that do not have matching keys. Spark also allows for much more sophisticated (typo) join policies in addition to equi-joins. We can even use complex types and perform something like checking whether a key exists within an array when you perform a join.

- Inner joins - keep rows with keys that exist in the left and right datasets
- Outer joins - keep rows with keys in either the left or right datasets
- Left outer joins - keep rows with keys in the left dataset
- Right outer joins - keep rows with keys in the right dataset
- Left semi joins - keep the rows in the left, and only the left, dataset where the key appears in the right dataset
- Left anti joins - keep the rows in the left, and only the left, dataset where they *do not appear* in the right dataset
- Natural joins - perform a join by implicitly matching the columns between the two datasets with the same names
- Cross (or Cartesian) joins - match every row in the left dataset with every row in the right dataset

Inner joins evaluate the keys in both of the DataFrames or tables to join only the rows that evaluate true. In the following example, we join the graduateProgram DataFrame with the person DataFrame to create a new DataFrame:

```
joinExpression = person["graduate_program"] == graduateProgram['id']
```

Keys that do not exist in both DataFrames will not show in the resulting DataFrame. For example, the following expression would result in zero values in the resulting DataFrame

```
wrongJoinExpression = person["name"] == graduateProgram["school"]
```

Inner joins are the default join, so we just need to specify our left DataFrame and join the right in the JOIN expression

```
person.join(graduateProgram,joinExpression).show()
```

Outer joins evaluate the keys in both of the DataFrames or tables and includes (and joins together) the rows that evaluate to true or false. If there is no equivalent row in either the left or right DataFrame, Spark will insert null

```
joinType="outer"
person.join(graduateProgram,joinExpression,joinType).show()
```

Semi joins do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. If the value does exist, those rows will be kept in the result, even if there are duplicate keys in the left DataFrame. Think of left semi joins as filters on a DataFrame, as opposed to the function of a conventional join

```
joinType="left_semi"
graduateProgram.join(person,joinExpression,joinType).show()
```

Left anti joins are the opposite of left semi joins - they do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. Rather than keeping the values that exist in the secondDataFrame, they keep only the values that *do not have a corresponding key in the second DataFrame*. Think of anti joins as a NOT IN SQL-style filter:

```
joinType="left_anti"
graduateProgram.join(person,joinExpression,joinType).show()
```

Natural joins make implicit guesses at the columns on which you would like to join. It finds matching columns and returns the results. Left, right, and outer natural joins are all supported
- implicit is always dangerous! A query can give us incorrect results because the two DataFrames/tables share a column name (id), but it means different things in the datasets. You should always use this join with caution.

Cross-joins = *cartesian products*

- Cross-joins are inner joins that do not specify a predicate
- Cross joins will join every single row in the left DataFrame to ever single row in the right DataFrame.
- This will square the number of rows contained in the resulting DataFrame. If you have 1,000 rows in each DataFrame, the cross-join of these will result in 1,000,000 (1,000 x 1,000) rows.
- Must explicitly state that you want a cross-join by using the cross join keyword
- Advanced users can set the session-level configuration *spark.sql.crossJoin.enable* to true in order to allow cross-joins without warnings or without Spark trying to perform another join for you

```
person.crossJoin(graduateProgram).show()
```

Joins on Complex Types- Any expression is a valid join expression, assuming that it returns a Boolean

```
from pyspark.sql.functions import expr

person.withColumnRenamed("id", "personId")\
  .join(sparkStatus, expr("array_contains(spark_status, id)")).show()
```

Handling Duplicate Column Names

- in a DataFrame, each column has a unique ID within Spark's SQL Engine, Catalyst
- This unique ID is purely internal and not something that you can directly reference
- Difficult to refer to a specific column when you have a DataFrame with duplicate column names
- This can occur in two distinct situations:
    - The join expression that you specify does not remove one key from one of the input DataFrames and the keys have the same column name
    - Two columns on which you are not performing the join have the same name

Approach 1: **Different join expression** - When you have two keys that have the same name, probably the easiest fix is to change the join expression from a Boolean expression to a string or sequence. This automatically removes one of the columns for you during the join.

Approach 2: **Dropping the column after the join -** Another approach is to drop the offending column after the join. When doing this, we need to refer to the column via the original source DataFrame. We can do this if the join uses the same key names or if the source DataFrames have columns that simply have the same name.

Approach 3: **Renaming a column before the join**

How Spark Performs Joins
two core resources at play:

- the node-to-node communication strategy
- per node computation strategy.

These internals are likely irrelevant to your business problem. However, comprehending how Spark performs joins can mean the difference between a job that completes quickly and one that never completes at all (this sounds pretty relevant IMO)

Communication Strategies
cluster communication in two different ways during joins

- It either incurs a shuffle join, which results in an all-to-all communication
- broadcast join

The core foundation of our simplified view of joins is that in Spark you will have either a big table or a small table

Shuffle join:

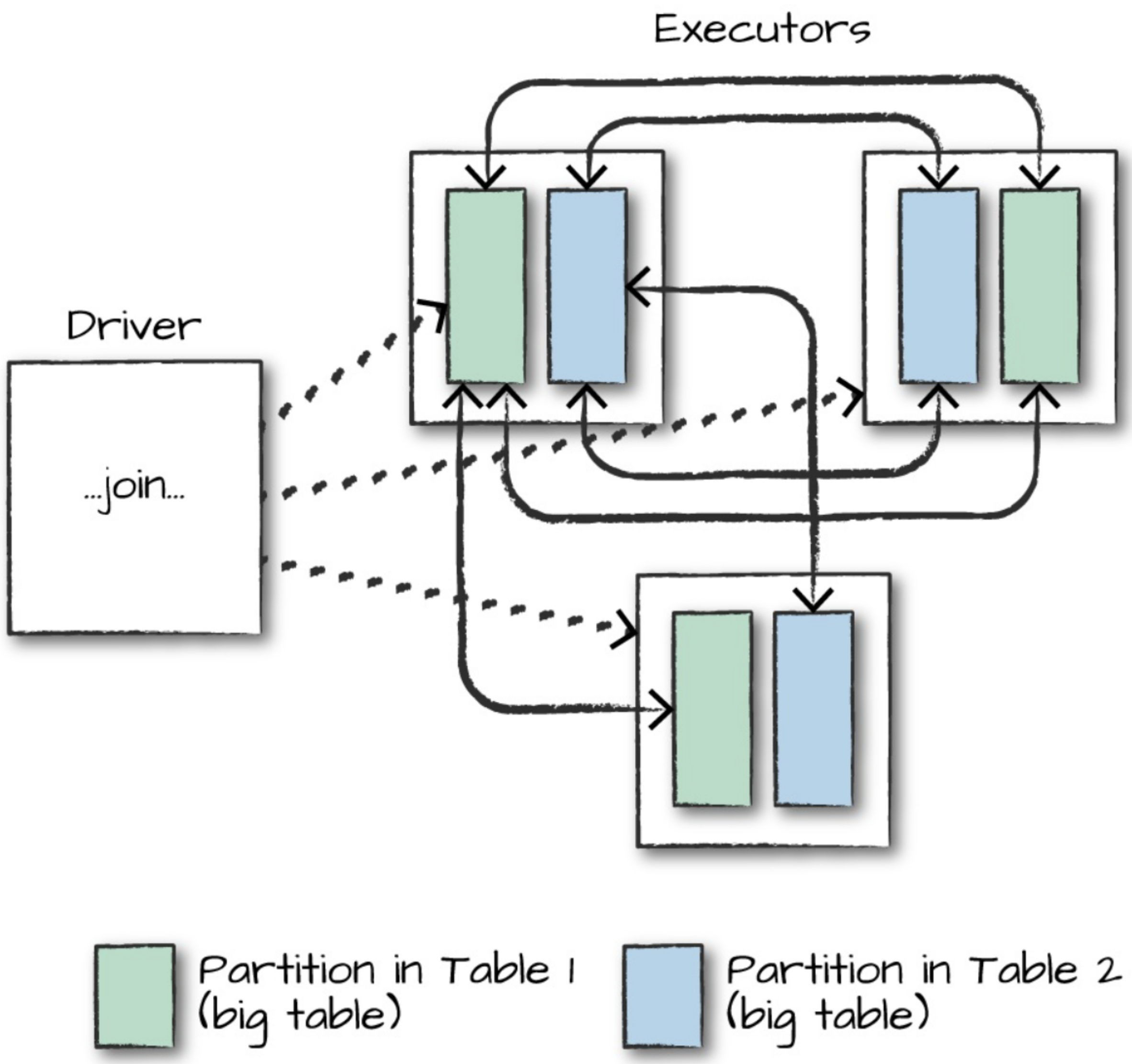
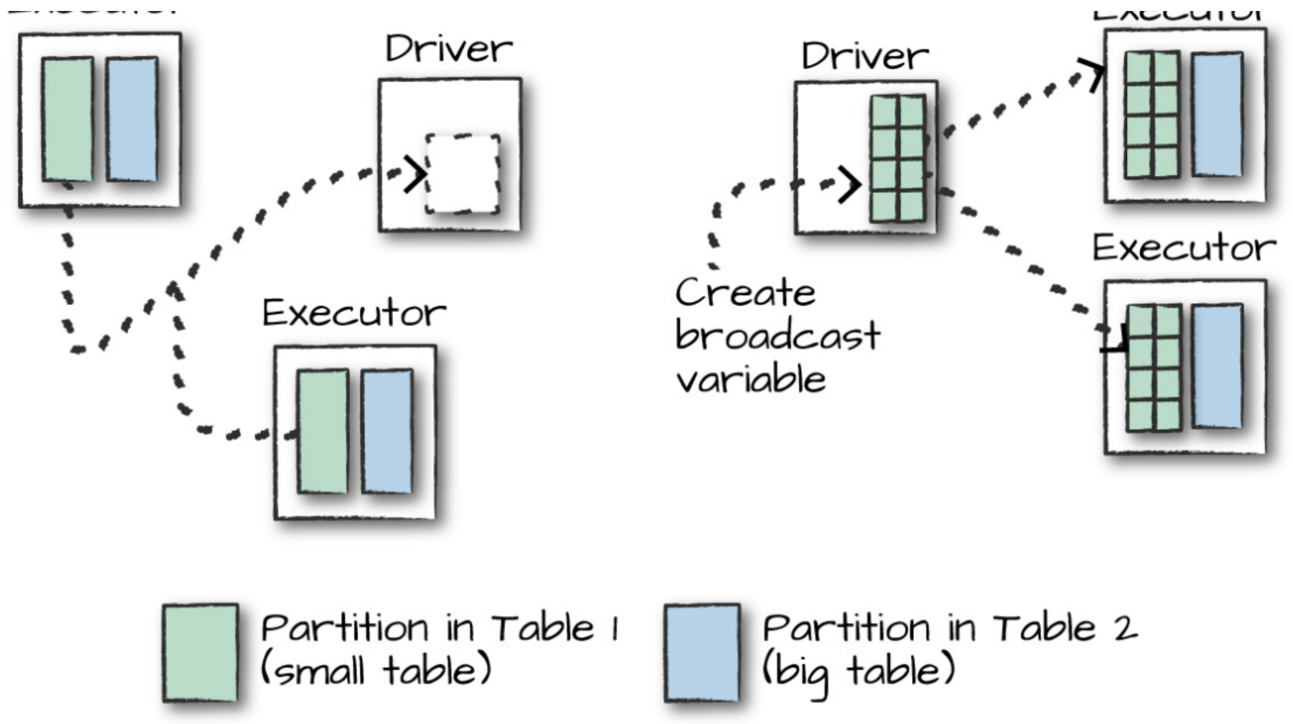

Executors
Driver
...join...
Partition in Table 1 (big table)
Partition in Table 2 (big table)

Figure 8-1. Joining two big tables

Shuffle join - every node talks to every other node and they share data according to which node has a certain key or set of keys (on which you are joining). These joins are expensive because the network can become congested with traffic, especially if your data is not partitioned well.

Big - to - Little Join:



Executor
Driver
Executor
Driver
Executor
Create broadcast variable
Executor
Partition in Table 1 (small table)
Partition in Table 2 (big table)

Figure 8-2. A broadcast join

Broadcast Join - When the table is small enough to fit into the memory of a single worker node we can optimize our join. Although we can use a big table–to–big table communication strategy, it can often be more efficient to use a broadcast join. What this means is that we will replicate our small DataFrame onto every worker node in the cluster (be it located on one machine or many). While it sounds computationally intensive, however, it prevents us from performing the all-to-all communication during the entire join process. Instead, we perform it only once at the beginning and then let each individual worker node perform the work without having to wait or communicate with any other worker node.

Little table–to–little table - When performing joins with small tables, it's usually best to let Spark decide how to join them. You can always force a broadcast join if you're noticing strange behavior