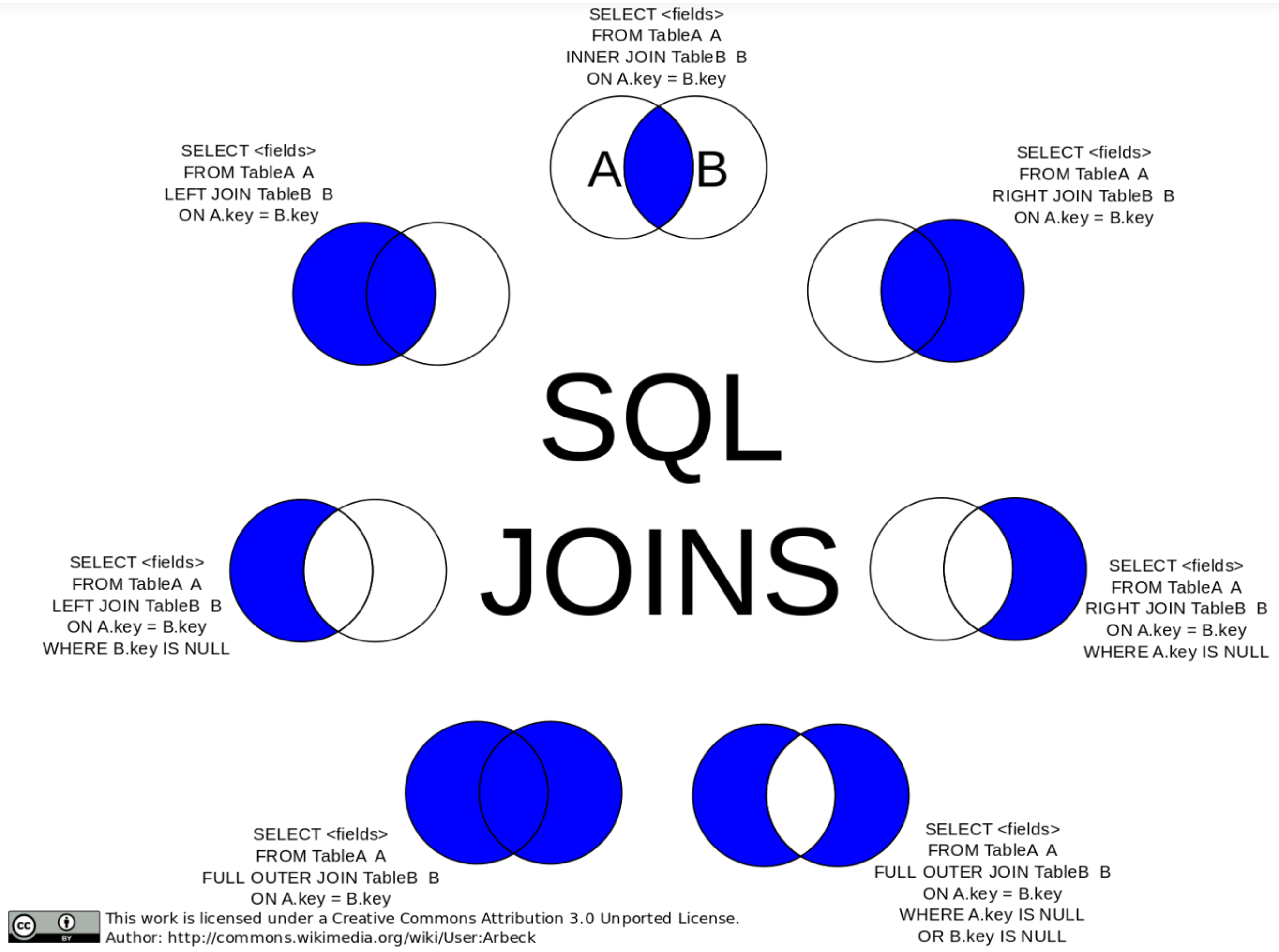
achilleus Follow
Feb 27, 2019 · 5 min read · Listen

Joins in Apache Spark — Part 1

A SQL join is basically combining 2 or more different tables(sets) to get 1 set of the result based on some criteria. Joining of data is the most common usage of any ETL applications but also is the most tricky and compute heavy operation. Spark offers most of the commonly used joins in SQL.



https://commons.wikimedia.org/wiki/File:SQL_Joins.svg

Currently, Spark offers

1)Inner-Join,

2) Left-Join,

3)Right-Join,

6)Left-Semi-Join,

7)Left-Anti-Semi-Join

For the sake of the examples, we will be using these dataframes.

```
val payment = sc.parallelize(Seq(
  (1, 101,2500), (2,102,1110), (3,103,500), (4 ,104,400), (5 ,105,
150), (6 ,106, 450)
)).toDF("paymentId", "customerId","amount")

scala> payment.show
+---------+----------+------+
|paymentId|customerId|amount|
+---------+----------+------+
|        1|       101|  2500|
|        2|       102|  1110|
|        3|       103|   500|
|        4|       104|   400|
|        5|       105|   150|
|        6|       106|   450|
+---------+----------+------+

scala> val customer = sc.parallelize(Seq((101,"Jon") , (102,"Aron") ,
(103,"Sam"))).toDF("customerId", "name")
customer: org.apache.spark.sql.DataFrame = [customerId: int, name:
string]

scala> customer.show
+----------+----+
|customerId|name|
+----------+----+
|       101| Jon|
```

1) Inner-Join

This is the default join in Spark. Inner join basically removes all the things that are not common in both the tables. It returns back all the data that has a match on the join condition from both sides of the table. It is basically an intersection of sets on the join column if you visualize in terms of a Venn diagram.

```
scala> val innerJoinDf = customer.join(payment,"customerId")
innerJoinDf: org.apache.spark.sql.DataFrame = [customerId: int, name: 
string ... 2 more fields]

scala> innerJoinDf.show
+----------+----+---------+------+
|customerId|name|paymentId|amount|
+----------+----+---------+------+
|       101| Jon|        1|  2500|
|       103| Sam|        3|   500|
|       102|Aron|        2|  1110|
+----------+----+---------+------+
```

So in the example, only customerId 101,102,103 have entries in both the tables hence inner join returns only those.

There are some other extra things that spark gives us out of the box.
1) Spark automatically removes duplicated "customerId" column, so column names are unique(When we use the above-mentioned syntax, more on this later).
2) And you don't have to prefix the table name to address them in the join clause which gets really wasteful sometimes.
3) We did not specify that we wanted to do an "inner-join", by default spark performs an inner-join if no join type is given.

Some gotchas:

```
scala> val innerJoinDf = customer.join(payment,"customerId", "inner")
<console>:27: error: overloaded method value join with alternatives:
  (right: org.apache.spark.sql.Dataset[_],joinExprs:
org.apache.spark.sql.Column,joinType:
String)org.apache.spark.sql.DataFrame <and>
  (right: org.apache.spark.sql.Dataset[_],usingColumns:
Seq[String],joinType: String)org.apache.spark.sql.DataFrame
 cannot be applied to (org.apache.spark.sql.DataFrame, String, String)
       val innerJoinDf = customer.join(payment,"customerId", "inner")
```

This wouldn't work and error out as shown above. Either you should skip the join type or the column name should be wrapped into scala `Seq` if have a join

```
scala> val innerJoinDf1 = customer.join(payment,Seq("customerId"),
"inner")
innerJoinDf1: org.apache.spark.sql.DataFrame = [customerId: int, name:
string ... 2 more fields]

scala> innerJoinDf1.show
+----------+----+---------+------+
|customerId|name|paymentId|amount|
+----------+----+---------+------+
|       101| Jon|        1|  2500|
|       103| Sam|        3|   500|
|       102|Aron|        2|  1110|
+----------+----+---------+------+
```

2)Left Join:

In a left join, all the rows from the left table are returned irrespective of whether there is a match in the right side table.
If a matching id is found in the right table is found, it is returned or else a null is appended. We use Left Join when nulls matter, make sense of data when there were no sales or something like that. Say we need all the days when there was no payment activity in the Rental Store.

```
object JoinType {
  def apply(typ: String): JoinType =
      case "leftanti" => LeftAnti
      case "cross" => Cross
```

Also, as you can see this is from the spark source code that the Left and left outer join are the same. It is just an alias in Spark code.

```
scala> val leftJoinDf = payment.join(customer,Seq("customerId"),
"left")
leftJoinDf: org.apache.spark.sql.DataFrame = [customerId: int,
paymentId: int ... 2 more fields]

scala> leftJoinDf.show
+----------+---------+------+----+
|customerId|paymentId|amount|name|
+----------+---------+------+----+
|       101|        1|  2500| Jon|
|       103|        3|   500| Sam|
|       102|        2|  1110|Aron|
|       105|        5|   150|null|
|       106|        6|   450|null|
|       104|        4|   400|null|
+----------+---------+------+----+

scala> payment.join(customer,Seq("customerId"), "left_outer").show
+----------+---------+------+----+
|customerId|paymentId|amount|name|
+----------+---------+------+----+
|       101|        1|  2500| Jon|
|       103|        3|   500| Sam|
|       102|        2|  1110|Aron|
|       105|        5|   150|null|
```

Here we are joining on `customerId` , and as you can see the resulting dataframe has all the entries for the rows in payment table. It is populated with customer data wherever a matching record is found in the right side Customer table else `nulls` are returned.

2)Right Join:

This is similar to Left join. In Right join, all the rows from the Right table are returned irrespective of whether there is a match in the left side table.

```
scala> val rightJoinDf = payment.join(customer,Seq("customerId"),
"right")
rightJoinDf: org.apache.spark.sql.DataFrame = [customerId: int,
paymentId: int ... 2 more fields]

scala> rightJoinDf.show
+----------+---------+------+----+
|customerId|paymentId|amount|name|
+----------+---------+------+----+
|       101|        1|  2500| Jon|
|       103|        3|   500| Sam|
|       102|        2|  1110|Aron|
+----------+---------+------+----+

scala> payment.join(customer,Seq("customerId"), "right_outer").show
+----------+---------+------+----+
|customerId|paymentId|amount|name|
+----------+---------+------+----+
```

Here the right side table is the customer, hence all the data from the customer table is returned back. Since there is no matching data on the left side payment table, no nulls are appended as a part of the output.

Also, the right join and Right Outer join yield the same output. Theoretically speaking all the things that could be achieved from the right join can be achieved by using left join but there can be few scenarios where right-join might come in handy.

4) Outer Join:

We use full outer joins to keep records from both the tables along with the associated null values in the respective left/right tables. It is kind of rare but generally used exception reports or situations when you would require data from both the tables. For example, you want to find a department which doesn't have an employee and also find an employee who doesn't have a department and also find a department which has an employee.

```
scala> val fullJoinDf = employees.join(departments,
Seq("departmentID"), "outer")
fullJoinDf: org.apache.spark.sql.DataFrame = [departmentId: int, name:
string ... 1 more field]
```

```
|          31|   Amy|             IT|
|          34|   Rob|        Medical|
|          34| Billy|        Medical|
|          27|Larson|           null|
|          35|  null|          Sales|
|          33| Bobby|            Law|
|          33| Richy|            Law|
+------------+------+---------------+
```

In this example, you can see that there is no employee in the sales dept and also Larson is not associated with any department.

In the *next part* of this, I will be covering the other types of joins available in Spark and the performance implications and some gotchas. Hope you are liking this series. Please leave a comment or suggestion and continue to the next article! Thanks for reading! Also, Check out my other articles, *Intro to Spark* and Spark 2.4.