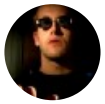

achilleus Follow

Feb 27, 2019 · 7 min read · Listen

Joins in Apache Spark — Part 2

In Part 1, we have covered some basic aspects of Spark join and some basic types of joins and how do they work in spark. In this section, we will be covering the Cartesian joins and Semi-Joins.

5) Cross Join:

As the saying goes, the cross product of big data and big data is an out-of-memory exception(From Holden's High-Performance Spark). Even before you start to read about it, try avoiding this with big tables in production. Unless it is the only way to do. Cross join basically computes a cartesian product of 2 tables. Say you have m rows in 1 table n rows in another, this would give (m*n) rows. So imagine a small table 10,000 customer table joined with a products table of 1000 records would give an exploding 10,000,000 records!

```
scala> val crossJoinDf = customer.crossJoin(payment)
crossJoinDf: org.apache.spark.sql.DataFrame = [customerId: int, name:
string ... 3 more fields]

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

```
|        102|Aron|         3|        103|   500|
|        102|Aron|         4|        104|   400|
|        102|Aron|         5|        105|   150|
|        102|Aron|         6|        106|   450|
|        103| Sam|         1|        101|  2500|
|        103| Sam|         2|        102|  1110|
|        103| Sam|         3|        103|   500|
|        103| Sam|         4|        104|   400|
|        103| Sam|         5|        105|   150|
|        103| Sam|         6|        106|   450|
+----------+----+----------+----------+------+
```

Spark is kind of restricting the users to accidentally trigger a cartesian join when no join condition was specified. Prior Spark, 2.1, `customer.join(payment)` would trigger a cross join. But now Spark throws an AnalysisException when the user forgets to give a condition on the joins.

```
scala> customer.join(payment)
res16: org.apache.spark.sql.DataFrame = [customerId: int, name: string
... 3 more fields]

scala> customer.join(payment).show
org.apache.spark.sql.AnalysisException: Detected implicit cartesian
product for INNER join between logical plans
Project [_1#17 AS customerId#20, _2#18 AS name#21]
....
Join condition is missing or trivial.
Either: use the CROSS JOIN syntax to allow cartesian products between
these
```




Also, to bypass this AnalysisException we have to either set the `spark.sql.crossJoin.enabled=true` in our Spark session builder object or set it for spark-shell : `spark-shell – conf spark.sql.crossJoin.enabled=true`. We can verify if this property is set by checking,

```
scala> spark.sparkContext.getConf.get("spark.sql.crossJoin.enabled")
res3: String = true
```

From 2.1, a dedicated function for Cross join has been added to support Cartesian joins.

```
val crossJoin = customer.crossJoin(payment)

scala> customer.crossJoin(payment).show
+----------+----+---------+----------+------+
|customerId|name|paymentId|customerId|amount|
+----------+----+---------+----------+------+
|       101| Jon|        1|       101|  2500|
|       101| Jon|        2|       102|  1110|
|       101| Jon|        3|       103|   500|
|       101| Jon|        4|       104|   400|
|       101| Jon|        5|       105|   150|
|       101| Jon|        6|       106|   450|
|       102|Aron|        1|       101|  2500|
|       102|Aron|        2|       102|  1110|
```

```
|        103| Sam|         3|        103|   500|
|        103| Sam|         4|        104|   400|
|        103| Sam|         5|        105|   150|
|        103| Sam|         6|        106|   450|
+----------+----+---------+----------+------+
```

6)Left-Semi-Join

This returns only the data from the left side that has a match on the right side based on the condition provided for the join statement. In contrast to Left join where all the rows from the Right side table are also present
in the output, there is right side table data in the output. This can also be achieved in subquery kind of queries in conjunction with IN/EXISTS in SQL but using semi_join restricts the amount of data that is read from the right side table.

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

```
+----------+----+
|       101| Jon|
|       102|Aron|
|       103| Sam|
+----------+----+

scala> payment.join(customer, payment("customerId") ===
customer("customerId"), "leftsemi").show
+---------+----------+------+
|paymentId|customerId|amount|
+---------+----------+------+
|        1|       101|  2500|
|        3|       103|   500|
|        2|       102|  1110|
+---------+----------+------+
```

If you look closely at the output, the joined output only consists of data from the Payment(Left) table which has a match for it in the Customer(Right) table. Rest of all the stuff is ignored. Also, note that the `name` column from the `Customer` table is not returned even for the matching `customerId` .This is really useful when you are trying to extract the only data in left that has a match on the right.

Semi-Join can feel similar to Inner Join but the difference between them is that `Left Semi Join` only returns the records from the left-hand table, whereas the `Inner Join` returns the `columns` from both tables.




As the name suggests, it does exactly the opposite of Left semi-join. The output would just `return the data` that `doesn't have a match` on the `right` side table. Only the columns on the left side table would be included in the result. Just the data filtered for the NOT IN condition.

```
scala> payment.join(customer, payment("customerId") ===
customer("customerId"), "leftanti").show
+---------+----------+------+
|paymentId|customerId|amount|
+---------+----------+------+
|        5|       105|   150|
|        6|       106|   450|
|        4|       104|   400|
+---------+----------+------+
```

8) Self Join

In self join, we join the dataframe with itself. We have to make sure we are aliasing the dataframe so that we can access the individual columns without name collisions.

Traditionally, self-joins are used to querying hierarchical data, comparing 2 attributes of the same table. In this example, we have the `employee` table

```
val employee1 = spark.createDataFrame(Seq(
  (1,"ceo",None),
  (2,"manager1",Some(1)),
  (3,"manager2",Some(1)),
  (101,"Amy",Some(2)),
  (102,"Sam",Some(2)),
  (103,"Aron",Some(3)),
  (104,"Bobby",Some(3)),
  (105,"Jon", Some(3))
)).toDF("employeeId","employeeName","managerId")
```

```
scala> val selfJoinedEmp =
employee1.as("e").join(employee1.as("m"),$"m.employeeId" ===
$"e.managerId")
selfJoinedEmp: org.apache.spark.sql.DataFrame = [employeeId: int,
employeeName: string ... 4 more fields]
```

```
scala> selfJoinedEmp.show
+----------+------------+---------+----------+------------+---------+
|employeeId|employeeName|managerId|employeeId|employeeName|managerId|
+----------+------------+---------+----------+------------+---------+
|         3|    manager2|        1|         1|         ceo|     null|
|         2|    manager1|        1|         1|         ceo|     null|
|       102|         Sam|        2|         2|    manager1|        1|
|       101|         Amy|        2|         2|    manager1|        1|
|       105|         Jon|        3|         3|    manager2|        1|
|       104|       Bobby|        3|         3|    manager2|        1|
|       103|        Aron|        3|         3|    manager2|        1|
+----------+------------+---------+----------+------------+---------+
```

This joined dataset has a lot of redundant data and doesn't give us a clear picture of what our requirement is.




```
Name"))
.show
+--------+-----------+
|employee|managerName|
+--------+-----------+
|manager2|        ceo|
|manager1|        ceo|
|     Sam|   manager1|
|     Amy|   manager1|
|     Jon|   manager2|
|   Bobby|   manager2|
|    Aron|   manager2|
+--------+-----------+
```

We can select the required columns and alias them to make the output more understandable.

Joins on columns with nulls

Let's say we wanted to join on columns which have nulls in it. By default, Spark would skip these columns.

Say I want to join, `df1` and `df2` on `id` column which has `nulls` in it. The result would not have the null values.

```
| 456|name3|                  | 223|Legal|
|null|name2|                  | 456|   IT|
+----+-----+                  +----+-----+

scala> df1.join(df2, Seq("id")).show
+---+-----+-----+
| id| name| dept|
+---+-----+-----+
|123|name1|sales|
|456|name3|   IT|
+---+-----+-----+
```

But let say we don't want to lose that data,

```
scala> df1.join(df2,df1("id") <=> df2("id")).drop(df1("id")).show
+-----+----+-----+
| name|  id| dept|
+-----+----+-----+
|name1| 123|sales|
|name3|    |   IT|
|name2|null|sales|
+-----+----+-----+
```

The `<=>` operator is an *Equality test operator that is safe to use when the columns have null values vs* `===` . `<=>` returns the same result as the = operator for non-null operands, but returns true if both are null, false if one of them is null.




When you join 2 columns, we generally ended having at-least 1 column duplicated if we join using the below signature. Here `id` is repeated twice. Either we have to drop that column or use another elegant way.

```
scala> df1.join(df2, df1.col("id") === df2.col("id")).show
+---+-----+---+-----+
| id| name| id| dept|
+---+-----+---+-----+
|123|name1|123|sales|
|   |name3|   |   IT|
+---+-----+---+-----+
```

The `join` S method in spark has a method that takes `usingColumns` as 1 of the parameter. When we use this method spark prevents duplicated columns when joining 2 dataframes.

```
scala> df1.join(df2 , Seq("id")).show
+---+-----+-----+
| id| name| dept|
+---+-----+-----+
|123|name1|sales|
|   |name3|   IT|
+---+-----+-----+
```




the column should be the same.

Joins in Datasets

We have the `joinWith` function to join 2 Datasets. This is similar to other join discussed above but the only difference is that `joinWith` preserves the type information of the resulting Dataset. It returns a `Dataset[(T, U)]` compared to `DataFrame` in all the above-mentioned joins which can be really precious if type-safety matters to you.

```
val paymentDs: Dataset[Payments] = payment.as[Payments]
val customerDs: Dataset[Customer] = customer.as[Customer]

val x: Dataset[(Customer, Payments)] =
customerDs.joinWith(paymentDs,paymentDs.col("customerId") ===
customerDs.col("paymentDs"))
```

If you notice above, we get back a `Dataset[(Customer, Payments)]` when compared the join operation in the previous examples where we get a `Dataframe` back. This has all the types of join that are available for Dataframes as discussed above. 1 gotcha is that it is relatively tricky to use the returned object as it is a Dataset of Set of (Customer and Payment) object

Thanks for reading! Please do share the article, if you liked it. Any comments or suggestions are welcome! Check out my other articles, *Intro to Spark* and Spark 2.4.