Mastering Query Plans in Spark 3.0

Spark query plans in a nutshell.

Davi... Following
Jul 3 · 9 min rea...

In Spark SQL the query plan is the entry point for understanding the details about the query execution. It carries lots of useful information and provides insights about how the query will be executed. This is very important especially in heavy workloads or whenever the execution takes to long and becomes costly. Based on the information from the query plan we may find out what is not efficient and decide to rewrite part of the query to achieve better performance.

For someone not familiar with query plans, at first sight, the information may look a bit cryptic. It has a structure of a tree and each node represents an operator that provides some basic details about the

execution. The official Spark documentation which is otherwise nicely written and very informative becomes insufficient when it comes to execution plans. The motivation for this article is to provide some familiarity with the physical plans, we will take a tour of some of the most frequently used operators and explain what information they provide and how it can be interpreted.

The theory presented here is based mostly on the study of the Spark source code and on the practical experience with running and optimizing Spark queries on daily basis.

Basic example setup

For the sake of simplicity let's consider a query in which we apply a filter, carry out an aggregation, and join with another DataFrame:

```
# in PySpark API:

query = (
  questionsDF
  .filter(col('year') == 2019)
  .groupBy('user_id')
  .agg(
     count('*').alias('cnt')
  )
  .join(usersDF, 'user_id')
)
```

You can think about the data in this example in such a way that *usersDF* is a set of users that are asking questions that are represented by *questionsDF*. The questions are partitioned by the *year* column which is a year when the question was asked. In the query, we are interested in questions asked in 2019 and for each user, we want to know how many questions he/she asked. Also for each user, we want to have some additional information in the output, that is why we join with the *usersDF* after the aggregation.

There are two basic ways how to see the physical plan. The first one is by calling *explain* function on a DataFrame which shows a textual representation of the plan:

```
query.explain()
```

```
== Physical Plan ==
*(3) Project [user_id#216L, cnt#390L, display_name#231, about#232, location#233, downvotes#234L, upvotes#235L, reputa
tion#236L, views#237L]
+- *(3) BroadcastHashJoin [user_id#216L], [user_id#230L], Inner, BuildRight
   :- *(3) HashAggregate(keys=[user_id#216L], functions=[count(1)])
   :  +- Exchange hashpartitioning(user_id#216L, 200), true, [id=#332]
   :     +- *(1) HashAggregate(keys=[user_id#216L], functions=[partial_count(1)])
   :        +- *(1) Project [user_id#216L]
   :           +- *(1) Filter isnotnull(user_id#216L)
   :              +- *(1) ColumnarToRow
   :                 +- FileScan parquet [user_id#216L,year#218] Batched: true, DataFilters: [isnotnull(user_id#216
L)], Format: Parquet, Location: InMemoryFileIndex[file:/Users/david.vrba/data/questions], PartitionFilters: [isnotnul
l(year#218), (year#218 = 2019)], PushedFilters: [IsNotNull(user_id)], ReadSchema: struct<user_id:bigint>
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, true])), [id=#340]
      +- *(2) Project [user_id#230L, display_name#231, about#232, location#233, downvotes#234L, upvotes#235L, reputat
ion#236L, views#237L]
         +- *(2) Filter isnotnull(user_id#230L)
            +- *(2) ColumnarToRow
               +- FileScan parquet [user_id#230L,display_name#231,about#232,location#233,downvotes#234L,upvotes#235L,
reputation#236L,views#237L] Batched: true, DataFilters: [isnotnull(user_id#230L)], Format: Parquet, Location: InMemor
yFileIndex[file:/Users/david.vrba/data/users], PartitionFilters: [], PushedFilters: [IsNotNull(user_id)], ReadSchema:
```

```
struct<user_id:bigint,display_name:string,about:string,location:string,downvotes:bigint,upvotes:b...
```

There have been some improvements in Spark 3.0 in this regard and the *explain* function now takes a new argument *mode*. The value of this argument can be one of the following: *formatted*, *cost*, *codegen*. Using the *formatted* mode converts the query plan to a better organized output (here only part of the plan is displayed):

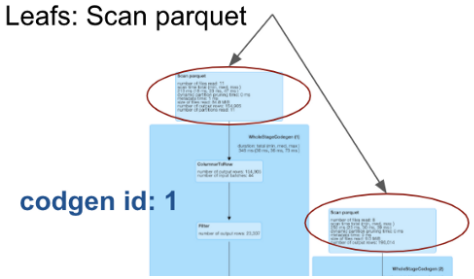
```
query.explain(mode='formatted')
```

```
== Physical Plan ==
* Project (14)
+- * BroadcastHashJoin Inner BuildRight (13)
   :- * HashAggregate (7)
   :  +- Exchange (6)
   :     +- * HashAggregate (5)
   :        +- * Project (4)
   :           +- * Filter (3)
   :              +- * ColumnarToRow (2)
   :                 +- Scan parquet  (1)
   +- BroadcastExchange (12)
      +- * Project (11)
         +- * Filter (10)
            +- * ColumnarToRow (9)
               +- Scan parquet  (8)


(1) Scan parquet
Output [2]: [user_id#216L, year#218]
Batched: true
Location: InMemoryFileIndex [file:/Users/david.vrba/data/questions]
PartitionFilters: [isnotnull(year#218), (year#218 = 2019)]
PushedFilters: [IsNotNull(user_id)]
ReadSchema: struct<user_id:bigint>

(2) ColumnarToRow [codegen id : 1]
Input [2]: [user_id#216L, year#218]

(3) Filter [codegen id : 1]
Input [2]: [user_id#216L, year#218]
Condition : isnotnull(user_id#216L)
```

```
(4) Project [codegen id : 1]
Output [1]: [user_id#216L]
Input [2]: [user_id#216L, year#218]
```

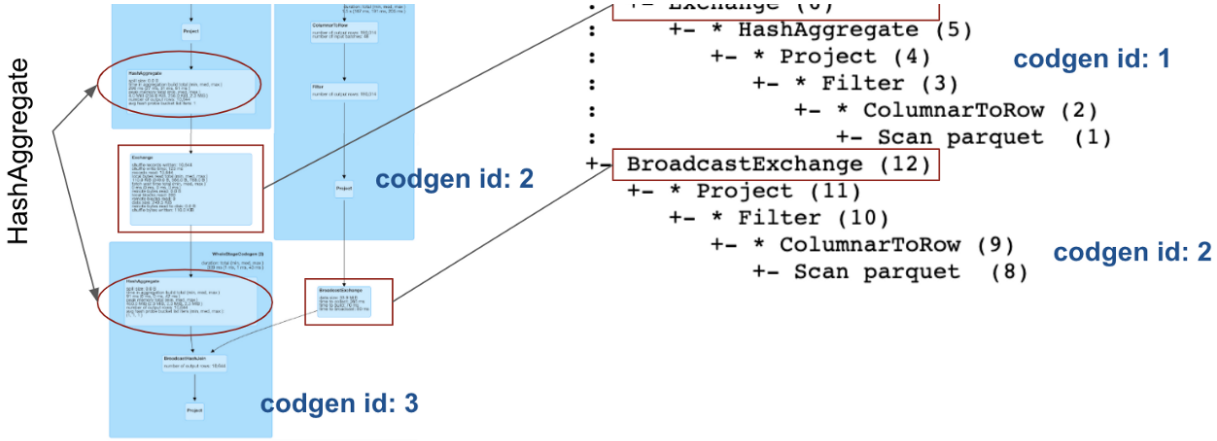
So in the formatted plan, you can see the “naked” tree which has only the names of the operators with a number in parenthesis. Below the tree, there is then a detailed description of each operator referenced by the number. The *cost* mode will show besides the physical plan also the optimized logical plan with the statistics for each operator so you can see what are the estimates for the data size at different steps of execution. Finally, the *codegen* mode shows the generated java code that will be executed.

The second option to see the plan is going to the SQL tab in Spark UI where are lists of all running and finished queries. By clicking on your query you will see the graphical representation of the physical plan. In the picture below I combined the graphical representation with the formatted textual tree to see how they correspond to each other:




```
== Physical Plan ==
* Project (14)
+- * BroadcastHashJoin Inner BuildRight (13)
   :- * HashAggregate (7)
   :  +- Exchange (6)
```

codgen id: 3



The difference here is that the graphical representation has the leaf nodes on the top and the root is at the bottom, while the textual tree is upside down.

CollapseCodegenStages

In the graphical representation of the physical plan, you can see that the operators are grouped into big blue rectangles. These big rectangles correspond to *codegen* stages. It is an optimization feature, which takes place in the phase of physical planning. There is a rule called *CollapseCodegenStages* which is responsible for that and the idea is to take operators that support code generation and collapse it together to speed-up the execution by eliminating virtual function calls. Not all operators support code generation, so some operators (for instance *Exchange*) are not part of the big rectangles. In our example,

there are three *codegen* stages that correspond to three big rectangles and in the formatted plan output, you can see the *id* of the *codegen* stage in the brackets at the operator. Also from the tree, you can tell if an operator supports the *codegen* or not because there is an asterisk with corresponding stage *codegen id* in the parenthesis if the *codegen* is supported.

Let's now briefly describe how to interpret each of the operators in our query plan.

Scan parquet

The *Scan parquet* operator represents reading the data from a parquet file format. From the detailed information, you can directly see what

columns will be selected from the source. Even though we do not select specific fields in our query, there is a *ColumnPruning* rule in the optimizer that will be applied and it makes sure that only those columns that are actually needed will be selected from the source. We can also see here two types of filters: *PartitionFilters* and *PushedFilters*.
The *PartitionFilters* are filters that are applied on columns by which the datasource is partitioned in the file system. These are very important because they allow for skipping the data that we don't need. It is always good to check whether the filters are propagated here correctly. The idea behind this is to read as little data as possible since the I/O is expensive. In Spark 2.4 there was also a field *partitionCount* which was the number of partitions that are actually scanned, but this field is no longer present in Spark 3.0.

The *PushedFilters* are on the other hand filters on fields that can be pushed directly to parquet files and they can be useful if the parquet file is sorted by these filtered columns because in that case, we can leverage the internal parquet structure for data skipping as well. The parquet file is composed of row groups and the footer of the file contains metadata about each of these row groups. This metadata contains also statistical information such as *min*and *max* value for each row group and based on this information Spark can decide whether it will read the row group or not.

Filter

The *Filter* operator is quite intuitive to understand, it simply represents the filtering condition. What may not be so obvious is how the operator was created because very often it doesn't directly correspond to the filtering condition used in the query. The reason for that is that all the filters are first processed by the Catalyst optimizer which may modify and relocate them. There are several rules applied to the logical filters before they are converted to a physical operator. Let's list a couple of the rules here:

- PushDownPredicates — this rule will push filters closer to the source through several other operators, but not all of them. For example, it will not push them through expressions that are not deterministic. If we use functions such as *first*, *last*, *collect_set*, *collect_list*, *rand* (and some other)the *Filter* will not be pushed through them because these functions are not deterministic in Spark.
- CombineFilters — combines two neighboring operators into one (it collects the conditions from two following filters into one complex condition).
- InferFiltersFromConstraints — this rule actually creates a new *Filter*operator for example from a join condition (from a simple inner join it will create a filter condition *joining key is not null*).

- PruneFilters — removes redundant filters (for example if a filter always evaluates to *True*).

Project

This operator simply represents what columns will be projected (selected). Each time we call *select*, *withColumn,* or *drop* transformations on a DataFrame*,* Spark will add the *Project* operator to the logical plan which is then converted to its counterpart in the physical plan. Again there are some optimization rules applied to it before it is converted:

- ColumnPruning — this is a rule we already mentioned above, it prunes the columns that are not needed to reduce the data volume that will be scanned.
- CollapseProject — it combines neighboring *Project* operators into one.
- PushProjectionThroughUnion — this rule will push the *Project* through both sides of the *Union* operator.

Exchange

The *Exchange* operator represents shuffle, which is a physical data movement on the cluster. This operation is considered to be quite expensive because it moves the data over the network. The information in

the query plan contains also details about how the data will be repartitioned. In our example, it is *hashpartitioning(user_id, 200)* as you can see below:

This means that the data will be repartitioned according to the *user_id*column into 200 partitions and all rows with the same value of *user_id* will belong to the same partition and will be located on the same executor. To make sure that exactly 200 partitions are created, Spark will always compute the hash of the *user_id* and then will compute positive modulo 200. The consequence of this is that more different *user_id*s will be located in the same partition. And what can also happen is that some partitions can become empty. There are other types of partitioning worth to mention:

- RoundRobinPartitioning — with this partitioning the data will be distributed randomly into *n* approximately equally sized partitions, where *n* is specified by the user in the *repartition(n)* function

- SinglePartition — with this partitioning all the data are moved to a single partition to a single executor. This happens for example when calling a window function where the window becomes the whole DataFrame (when you don't provide an argument to the *partitionBy()*function in the *Window* definition).
- RangePartitioning — this partitioning is used when sorting the data, after calling *orderBy* or *sort* transformations.

HashAggregate

This operator represents data aggregation. It usually comes in pair of two operators which may or may not be divided by an *Exchange* as you can see here:

To understand better the logic behind the *Exchange* in these situations you can check my previous article about data distribution in Spark SQL where I describe it in detail. The reason for having two *HashAggregate* operators is that the first one does a partial aggregation, which aggregates separately each partition on each executor. In our example, you can see in the *Functions* field that it says *partial_count(1)*. The final merge of the partial results follows in the second *HashAggregate*. The operator also has the *Keys*field which shows the columns by which the data is grouped. The *Results*field shows the columns that are available after the aggregation.

BroadcastHashJoin & BroadcastExchange

The *BroadcastHashJoin (BHJ)* is an operator that represents a specific joining algorithm. Apart from this one, there are also other joining algorithms available in Spark such as *SortMergeJoin* or *ShuffleHashJoin* and to read more about them you can check my other article about joins in Spark 3.0. The *BHJ* always comes in a pair with *BroadcastExchange* which is an operator that represents the broadcasted shuffle — the data will be collected to the driver and then send over to each executor where it will be available for the join.

ColumnarToRow

This is a new operator introduced in Spark 3.0 and it is used as a transition between columnar and row execution.

Conclusion

The physical plans in Spark SQL are composed of operators that carry useful information about the execution. Having a proper understanding of each operator may help to get insights about the execution and by analyzing the plan we may discover what is not optimal and possibly try to fix it.

In this article, we described a set of operators that are frequently used in Spark physical plans. The set is by no means complete but we tried to focus on operators that are commonly used and very likely to be present in plans of basic analytical queries.

Spark Sql Apache Spark Data Engineering Data Science Query Optimization