EXPLAIN plan

Syntax

- `EXPLAIN [ EXTENDED | CODEGEN | COST | FORMATTED ] statement`
- `query.explain(mode="extended|formatted|cost|codegen")`

Example

```
trips = spark.read.table('pablo.yellow_tripdata')
zones = spark.read.table('pablo.taxi_zone_lookup')
from pyspark.sql.functions import count
query = trips \
.filter(trips.PULocationID == 142) \
.groupBy('VendorID','PULocationID') \
.agg(count('*').alias('cnt')) \
.join(zones, trips.PULocationID == zones.LocationID, 'left')
```

Formats

- **Formatted:** naked execution tree, only showing the names of the operators.
- **Cost:** Physical plan + optimized logical plan with the statistics for each operator.
- **Codegen:** Generated Java code that will be executed.

Graphical view

- Cluster / Spark UI / SQL tab, then click on your query.

Scan parquet operator

- Reading data from a parquet file.
- By default, there is a column pruning rule in the optimizer that will be applied and select only the required columns.
- *PartitionFilters*: are filters applied on columns by which the data source is partitioned in the file system. This helps to read as little data as possible.
- *PushedFilters*: Leverage the internal parquet file structure.
  - A parquet file has row groups and the footer of the file contains metadata about those (including statistical information). Spark then will decide if it will read the row group or not.

Filter operator

- Represents the filtering condition on the data.
- Filters are first processed by the Catalyst Optimizer.
- They are pruned (for example, to remove conditions that evaluate always to true) and combined.

Project operator

- Represents the columns that will be projected/selected.

Exchange

- This represents shuffles (physical data movement on the cluster).
- Quite expensive operation, as data is moved over the network.
- You can see how data is partitioning with `hashpartitioning` function.
  - A hash of the column is computed modulo 200 and copied around the cluster.
- Other hashing strategies:
  - **Round-robin:** data is partitioned in n different partitions, that the user specifies with the `repartition(n)` function.
  - **Single partition:** All data are moved to a single partition where a single executor does the job. This can happen when using window functions, if the `partitionBy` function is passed without arguments.
  - **Range partitioning"** Used when sorting the data, after calling `orderBy` or `sort` operations.

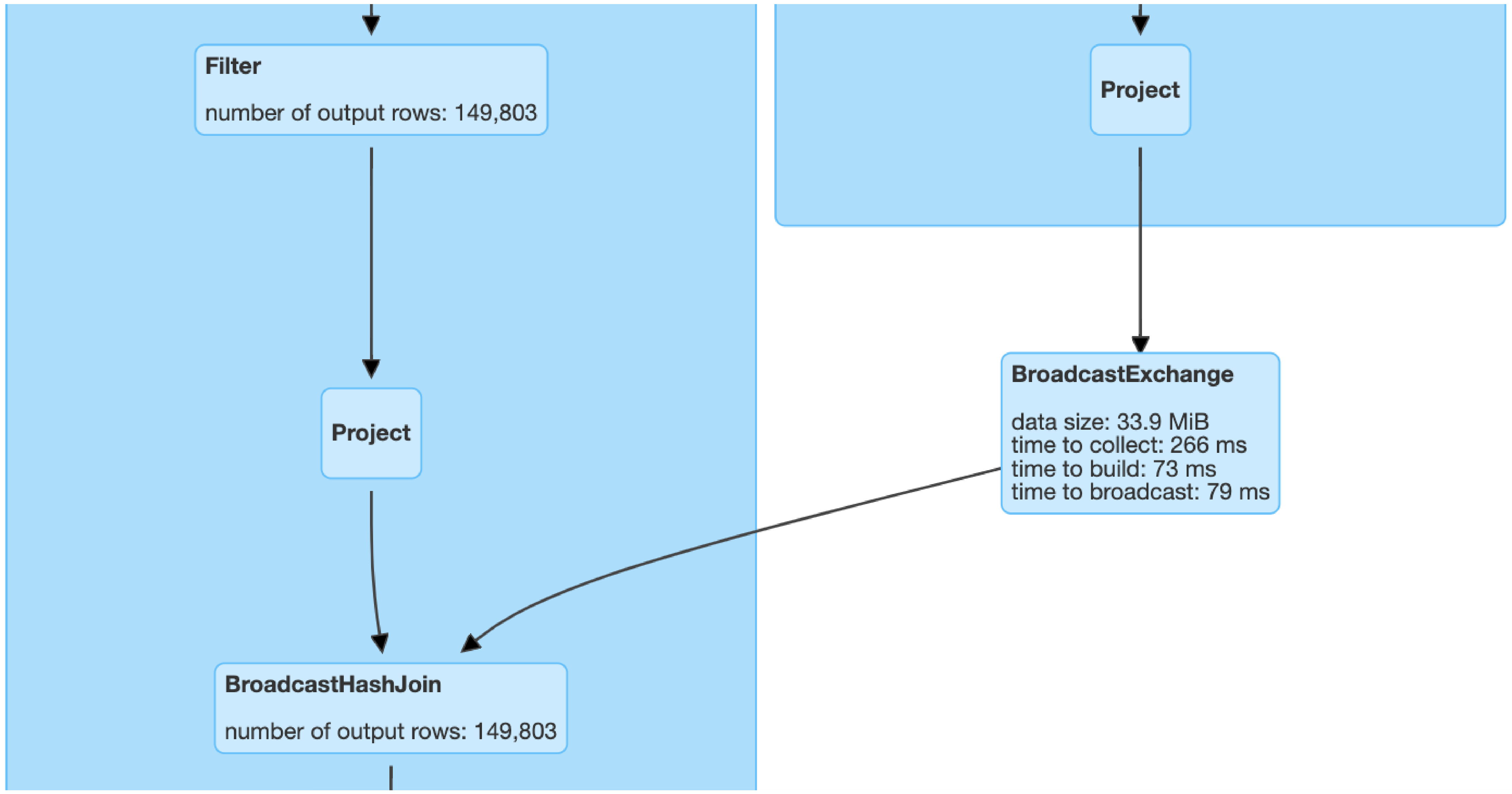
HashAggregate

- First hash aggregate does partial aggregation (each partition is sent to its executor).
- Final merge of the partial results is done in the second hash aggregate.

BroadcastHashJoin

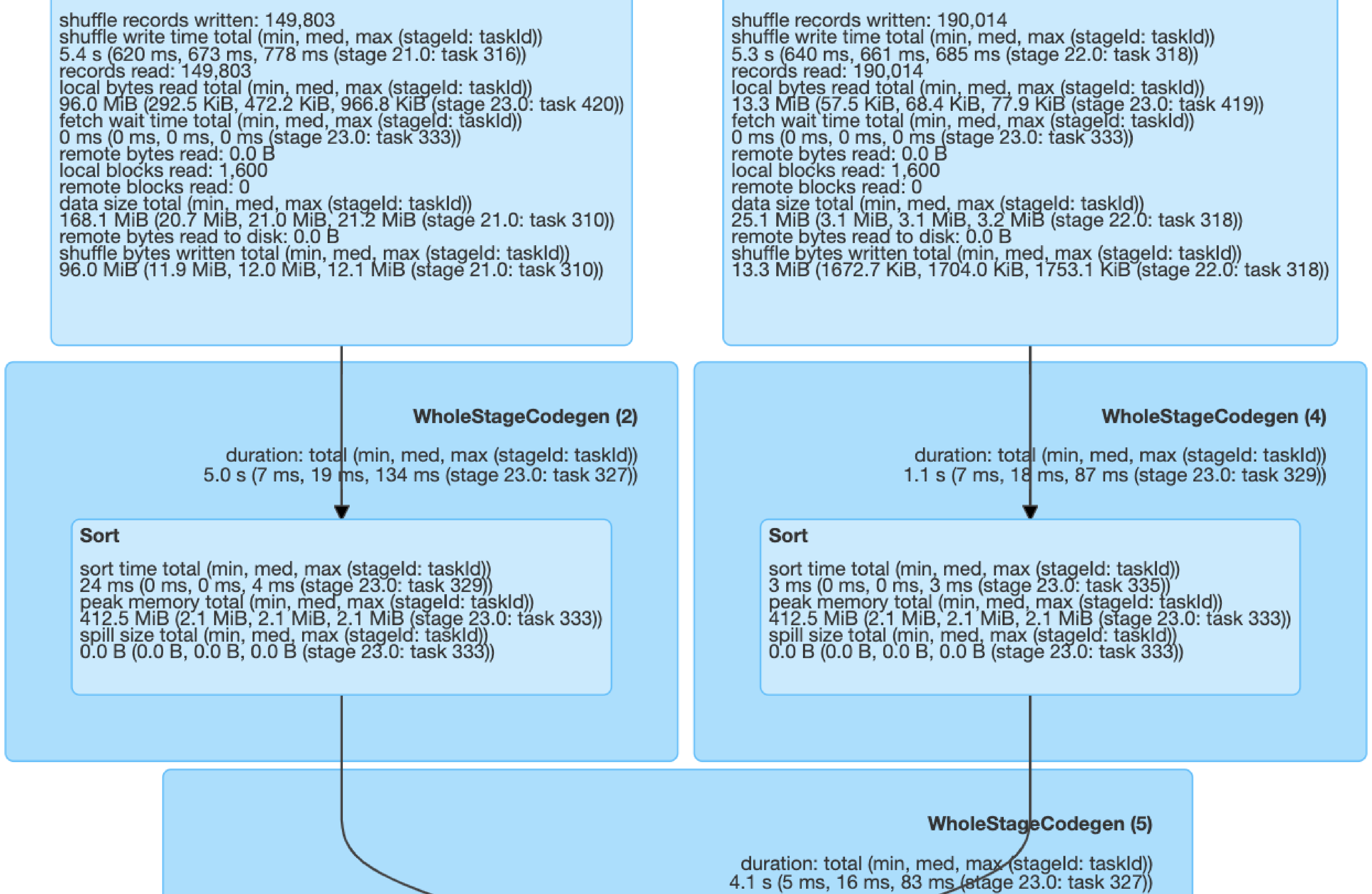
- Specifies the join algorithm. Other algorithms are `SortMergeJoin` and `ShuffleHashJoin`.
- In BHJ, data is collected to the driver and then sent over to each executor.
- Joins can be hinted:
  - `dfA.join(dfB.hint(algorithm), join_condition)` where algorithm can be *broadcast*, *shuffle_hash*, *shuffle_merge*.
- BHJ is the preferred algorithm if one side is much smaller than the other.

BroadcastHashJoin (cont.)



SortMergeJoin (SMJ)

- If neither of the dataframes can be broadcasted, this algorithm is used. It required that joining keys are sortable and there is an equality condition on the join.
- Both sides of the join have to be partitioned and ordered, which can make it expensive.



```
shuffle records written: 149,803
shuffle write time total (min, med, max (stageId: taskId))
5.4 s (620 ms, 673 ms, 778 ms (stage 21.0: task 316))
records read: 149,803
local bytes read total (min, med, max (stageId: taskId))
96.0 MiB (292.5 KiB, 472.2 KiB, 966.8 KiB (stage 23.0: task 420))
fetch wait time total (min, med, max (stageId: taskId))
0 ms (0 ms, 0 ms, 0 ms (stage 23.0: task 333))
remote bytes read: 0.0 B
local blocks read: 1,600
remote blocks read: 0
data size total (min, med, max (stageId: taskId))
168.1 MiB (20.7 MiB, 21.0 MiB, 21.2 MiB (stage 21.0: task 310))
remote bytes read to disk: 0.0 B
shuffle bytes written total (min, med, max (stageId: taskId))
96.0 MiB (11.9 MiB, 12.0 MiB, 12.1 MiB (stage 21.0: task 310))
```

```
shuffle records written: 190,014
shuffle write time total (min, med, max (stageId: taskId))
5.3 s (640 ms, 661 ms, 685 ms (stage 22.0: task 318))
records read: 190,014
local bytes read total (min, med, max (stageId: taskId))
13.3 MiB (57.5 KiB, 68.4 KiB, 77.9 KiB (stage 23.0: task 419))
fetch wait time total (min, med, max (stageId: taskId))
0 ms (0 ms, 0 ms, 0 ms (stage 23.0: task 333))
remote bytes read: 0.0 B
local blocks read: 1,600
remote blocks read: 0
data size total (min, med, max (stageId: taskId))
25.1 MiB (3.1 MiB, 3.1 MiB, 3.2 MiB (stage 22.0: task 318))
remote bytes read to disk: 0.0 B
shuffle bytes written total (min, med, max (stageId: taskId))
13.3 MiB (1672.7 KiB, 1704.0 KiB, 1753.1 KiB (stage 22.0: task 318))
```

WholeStageCodegen (2)

```
duration: total (min, med, max (stageId: taskId))
5.0 s (7 ms, 19 ms, 134 ms (stage 23.0: task 327))
```

Sort

```
sort time total (min, med, max (stageId: taskId))
24 ms (0 ms, 0 ms, 4 ms (stage 23.0: task 329))
peak memory total (min, med, max (stageId: taskId))
412.5 MiB (2.1 MiB, 2.1 MiB, 2.1 MiB (stage 23.0: task 333))
spill size total (min, med, max (stageId: taskId))
0.0 B (0.0 B, 0.0 B, 0.0 B (stage 23.0: task 333))
```

WholeStageCodegen (4)

```
duration: total (min, med, max (stageId: taskId))
1.1 s (7 ms, 18 ms, 87 ms (stage 23.0: task 329))
```

Sort

```
sort time total (min, med, max (stageId: taskId))
3 ms (0 ms, 0 ms, 3 ms (stage 23.0: task 335))
peak memory total (min, med, max (stageId: taskId))
412.5 MiB (2.1 MiB, 2.1 MiB, 2.1 MiB (stage 23.0: task 333))
spill size total (min, med, max (stageId: taskId))
0.0 B (0.0 B, 0.0 B, 0.0 B (stage 23.0: task 333))
```

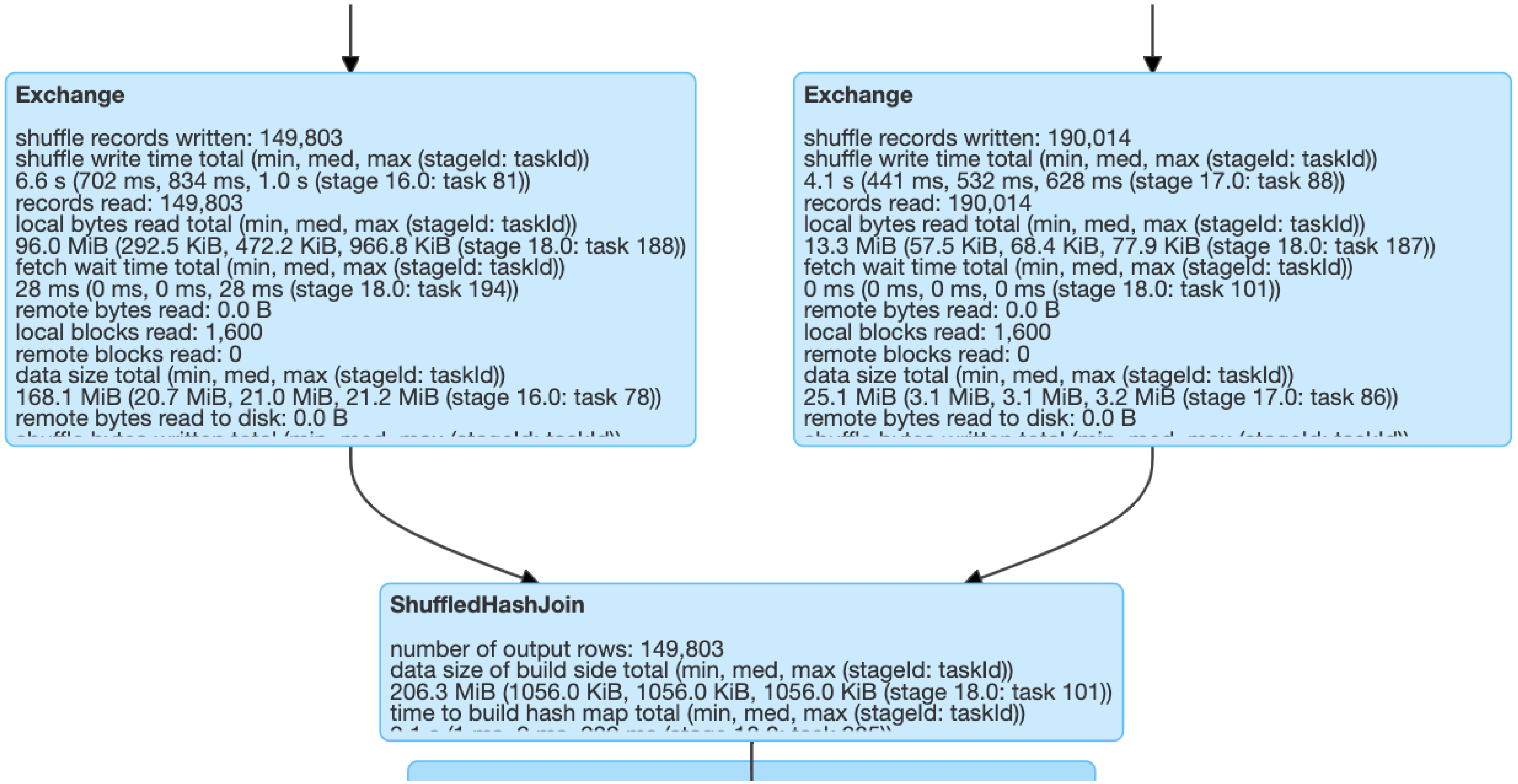
WholeStageCodegen (5)

```
duration: total (min, med, max (stageId: taskId))
4.1 s (5 ms, 16 ms, 83 ms (stage 23.0: task 327))
```

ShuffledHashJoin (SHJ)

- It is rarely used unless it is called by a hint.
    - By default, `spark.sql.join.preferSortMergeJoin` is set to `True`.
- It is recommended to use it if one side of the join is at least three times smaller than the other side.
- By hashing, one avoids the sorting step in SMJ, which improves performance

ShuffledHashJoin (SHJ) (cont.)



```
Exchange

shuffle records written: 149,803
shuffle write time total (min, med, max (stageId: taskId))
6.6 s (702 ms, 834 ms, 1.0 s (stage 16.0: task 81))
records read: 149,803
local bytes read total (min, med, max (stageId: taskId))
96.0 MiB (292.5 KiB, 472.2 KiB, 966.8 KiB (stage 18.0: task 188))
fetch wait time total (min, med, max (stageId: taskId))
28 ms (0 ms, 0 ms, 28 ms (stage 18.0: task 194))
remote bytes read: 0.0 B
local blocks read: 1,600
remote blocks read: 0
data size total (min, med, max (stageId: taskId))
168.1 MiB (20.7 MiB, 21.0 MiB, 21.2 MiB (stage 16.0: task 78))
remote bytes read to disk: 0.0 B
```

```
Exchange

shuffle records written: 190,014
shuffle write time total (min, med, max (stageId: taskId))
4.1 s (441 ms, 532 ms, 628 ms (stage 17.0: task 88))
records read: 190,014
local bytes read total (min, med, max (stageId: taskId))
13.3 MiB (57.5 KiB, 68.4 KiB, 77.9 KiB (stage 18.0: task 187))
fetch wait time total (min, med, max (stageId: taskId))
0 ms (0 ms, 0 ms, 0 ms (stage 18.0: task 101))
remote bytes read: 0.0 B
local blocks read: 1,600
remote blocks read: 0
data size total (min, med, max (stageId: taskId))
25.1 MiB (3.1 MiB, 3.1 MiB, 3.2 MiB (stage 17.0: task 86))
remote bytes read to disk: 0.0 B
```

```
ShuffledHashJoin

number of output rows: 149,803
data size of build side total (min, med, max (stageId: taskId))
206.3 MiB (1056.0 KiB, 1056.0 KiB, 1056.0 KiB (stage 18.0: task 101))
time to build hash map total (min, med, max (stageId: taskId))
```

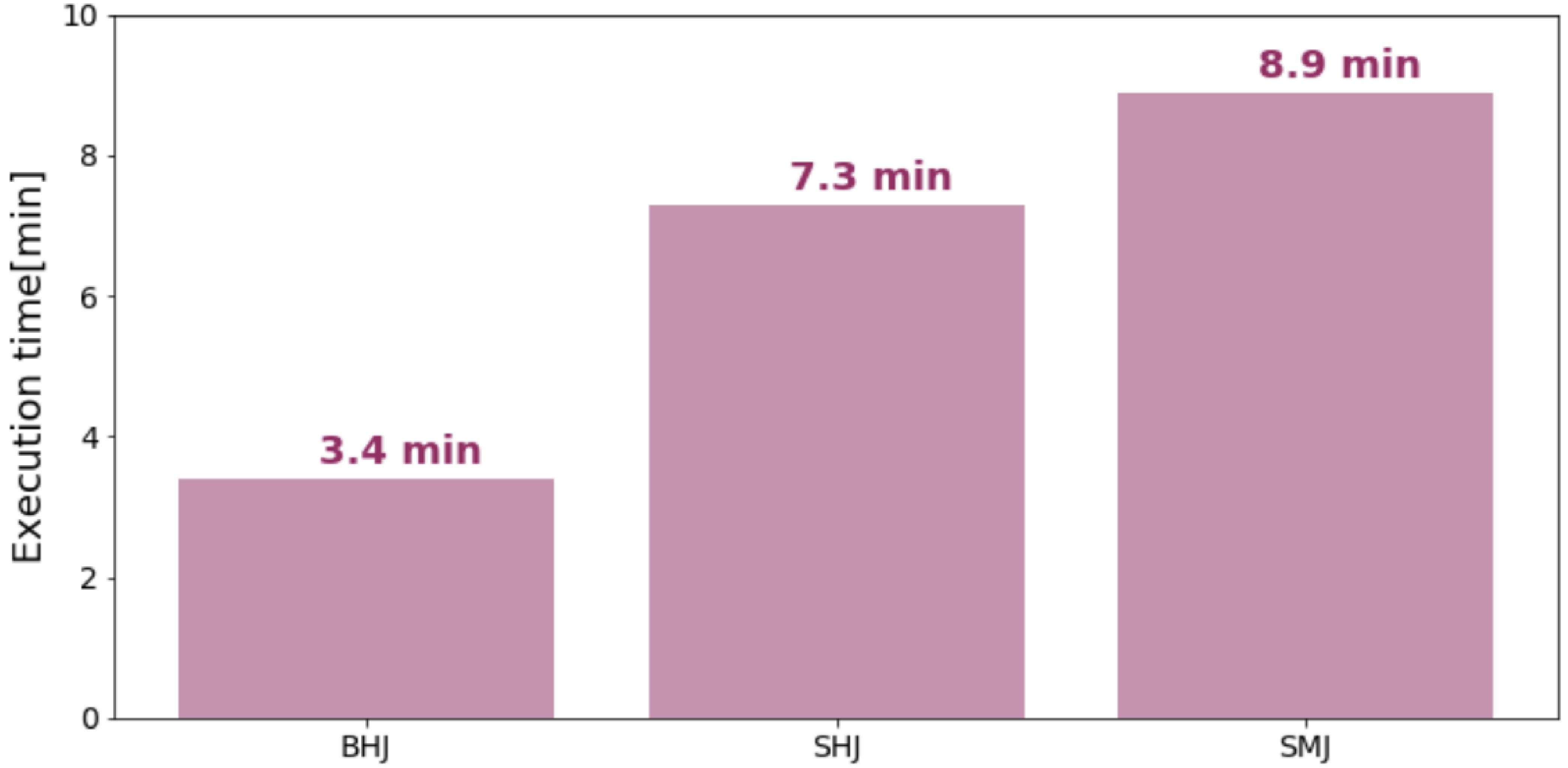
BroadcastNestedLoopJoin (BNLJ) / CartesianProductJoin (CPJ)

- These are cartesian product join, and is used if there is no equality condition on the join.
- They are really slow but can be avoided by providing an equality condition in the join.

Performance Comparison

- Dataframe A: 194 GB
- Dataframe B: 816 MB
- Cluster: driver m4.xlarge, 4 x c5.4xlarge nodes (64 cores).

Performance Comparison (cont.)


Execution time[min]
3.4 min
7.3 min
8.9 min
BHJ
SHJ
SMJ