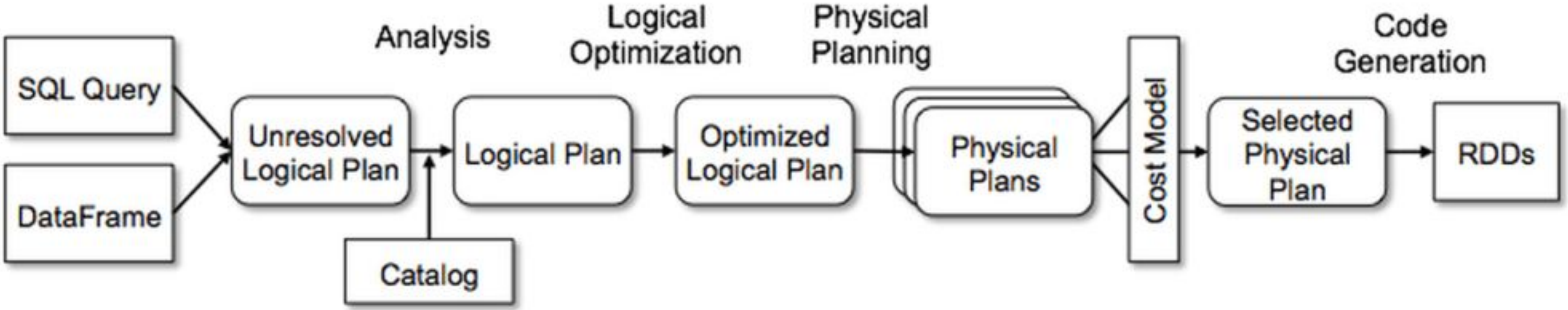
# Spark 3.0

## performance improvement

# Adaptive Query Execution

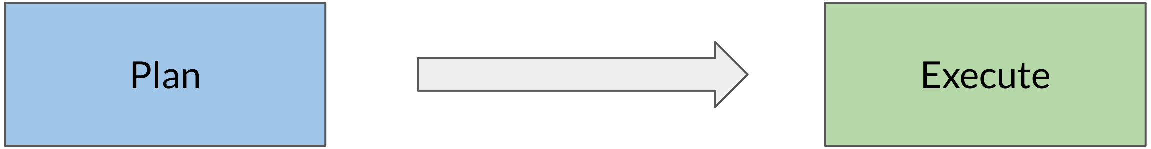
# Without AQE

- Plan computed before execution
- Source Table metrics are gathered before plan
- Intermediate Table metrics are inferred
- Single-pass optimisation



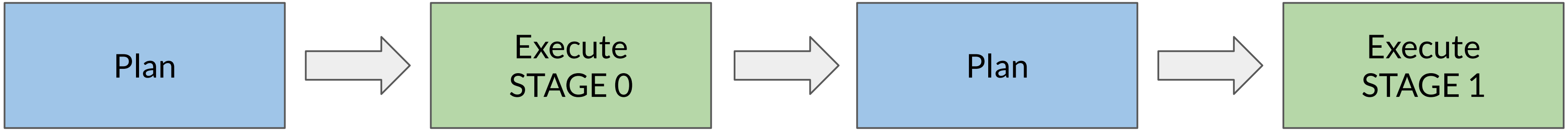
# Without AQE

- Plan computed before execution
- Source Table metrics are gathered before plan
- Intermediate Table metrics are inferred
- Single-pass optimisation



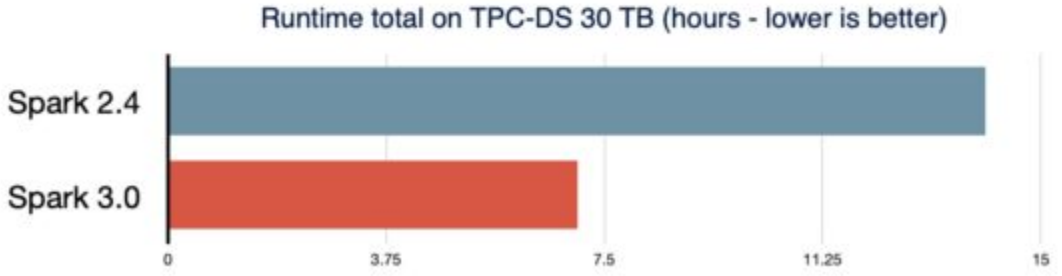
# With AQE

- Plan computed before execution
- Source Table metrics are gathered before plan
- Intermediate Table metrics are inferred
- During Execution new Table metrics are gathered
- During Execution new plan are computed based on the newest metrics
- New plan after each stage (shuffle)



# Adaptive Query Execution

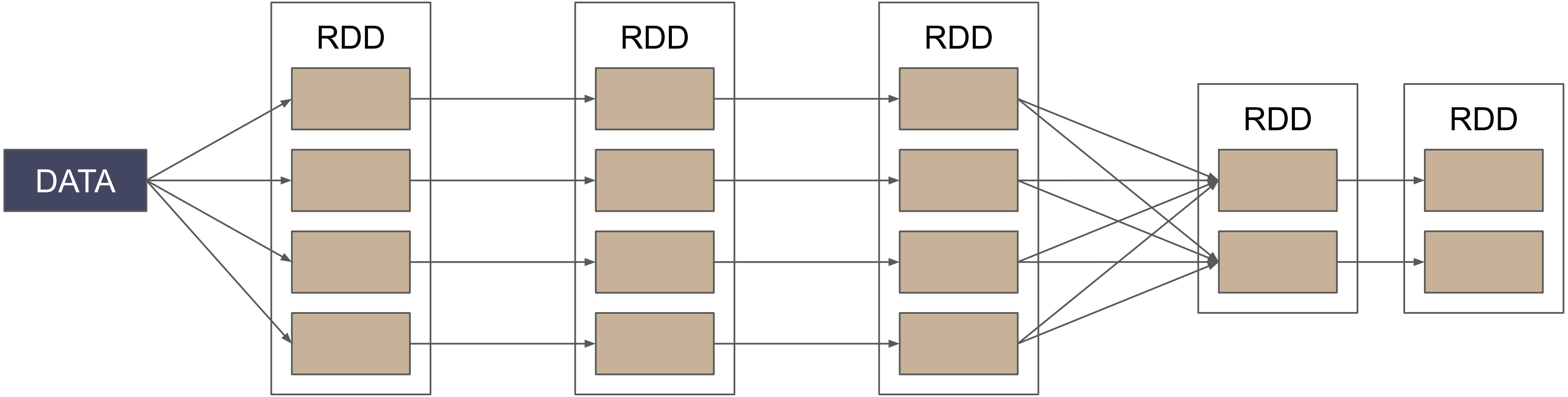
- Disable by default
- `spark.sql.adaptive.enabled = true`
- Optimisations
  - Dynamic Partition Pruning
  - Partition Coalesce
  - Skew Join optimisation
  - Smarter Broadcast Join
- 2x faster on TPC-DS than spark 2.4



# Task, Stage, Job and Partition

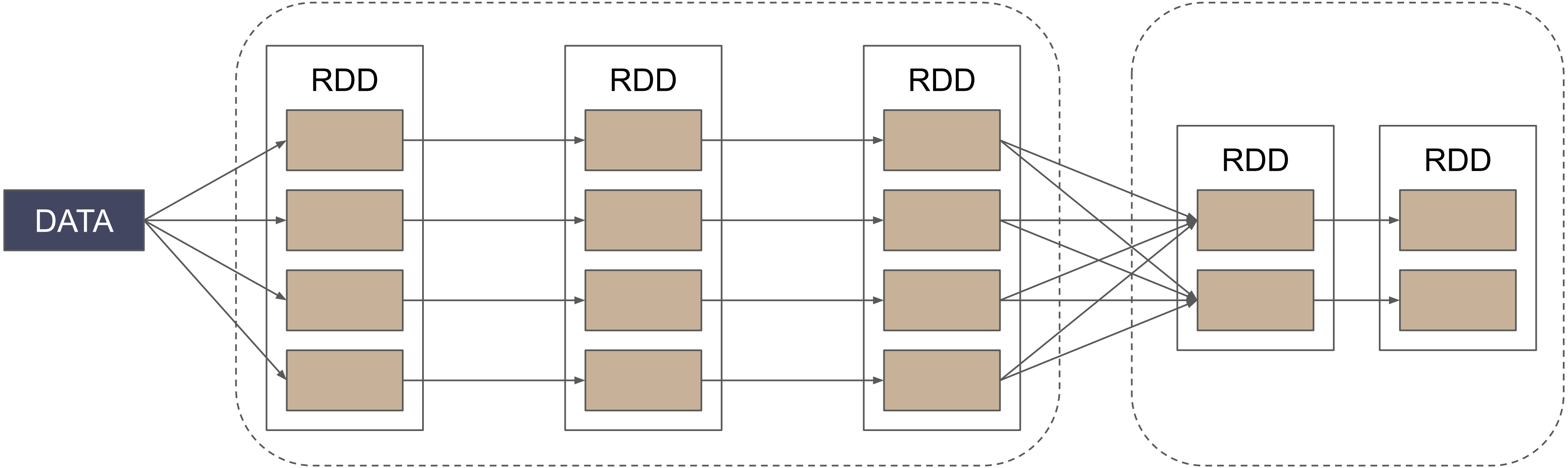
# Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1)))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



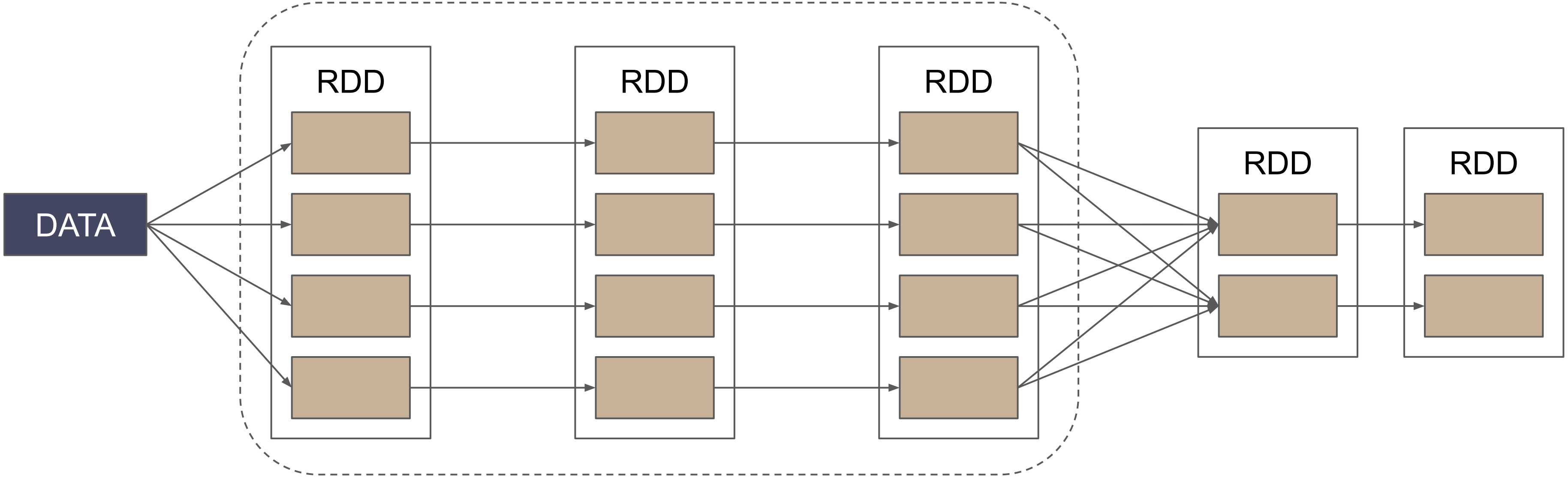
# Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1)))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



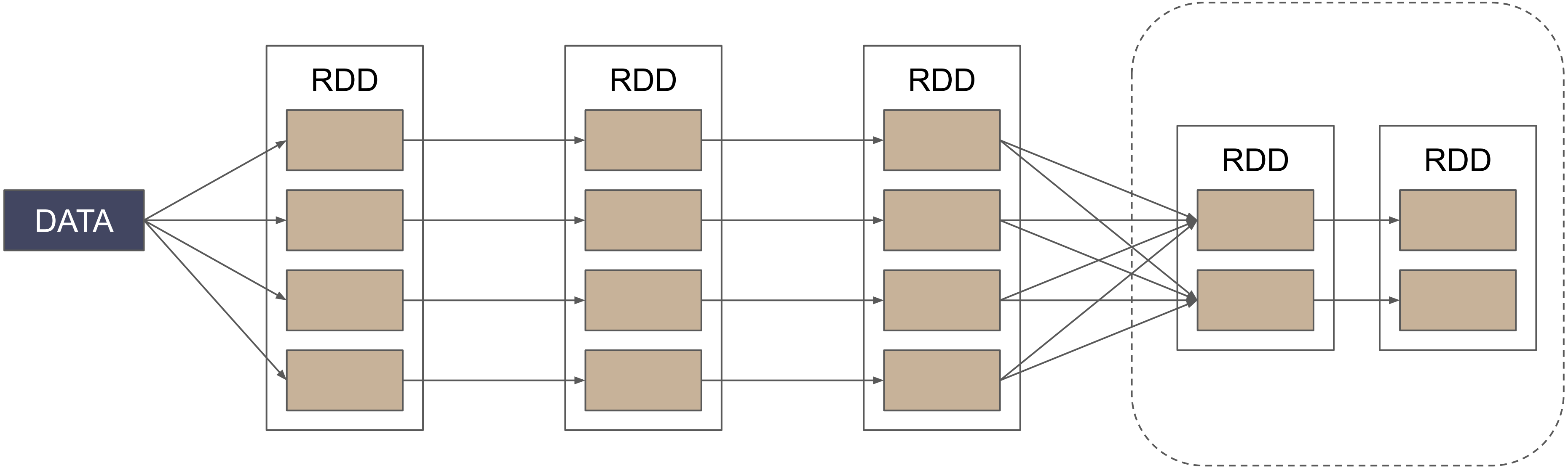
# Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1)))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



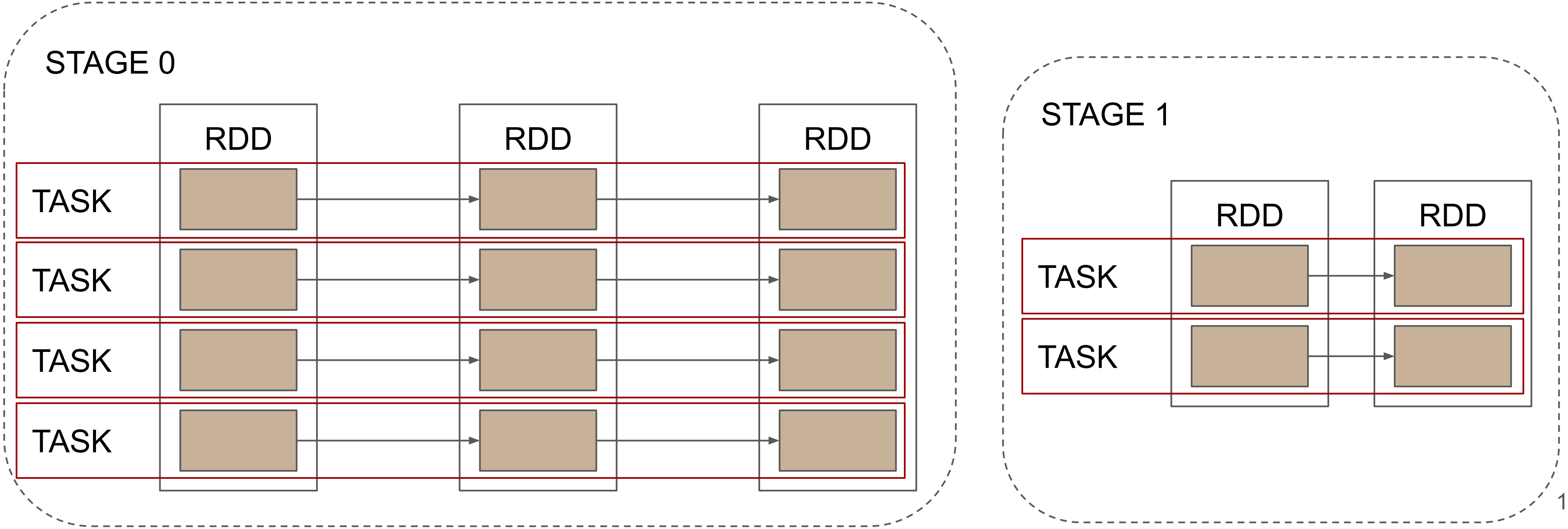
# Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1)))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



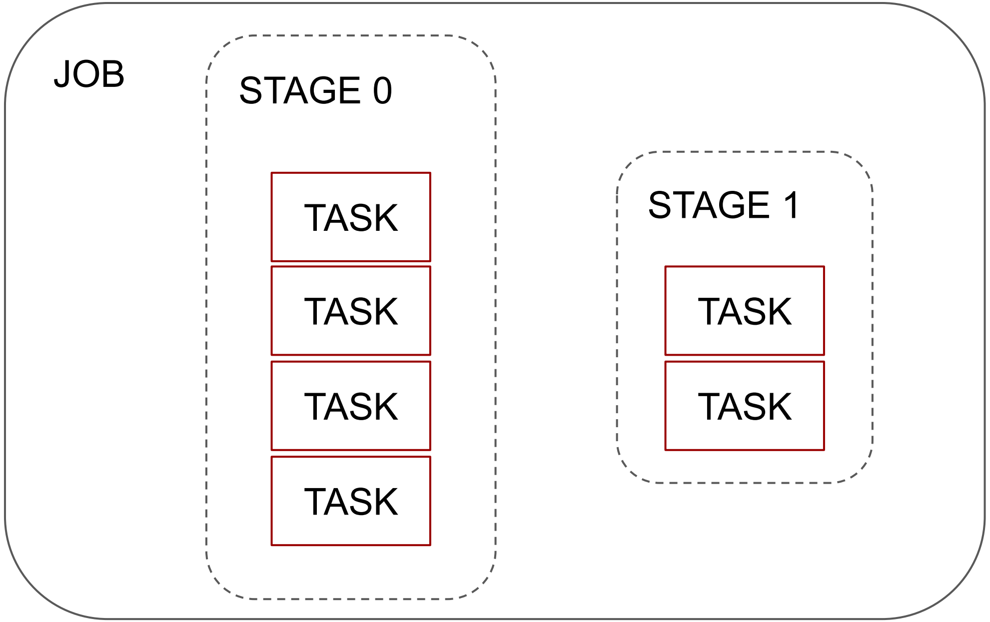
# Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1)))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



# Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1)))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



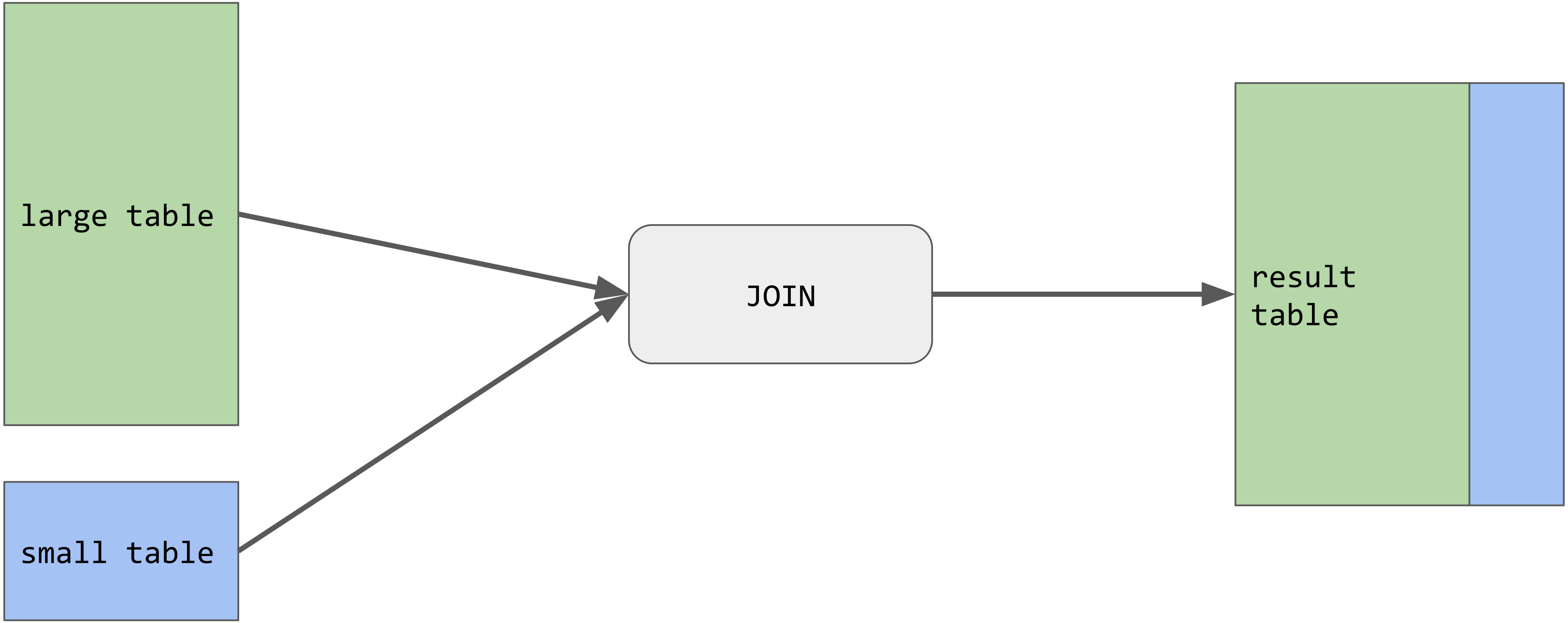
# Partition Coalesce

# Partition coalesce

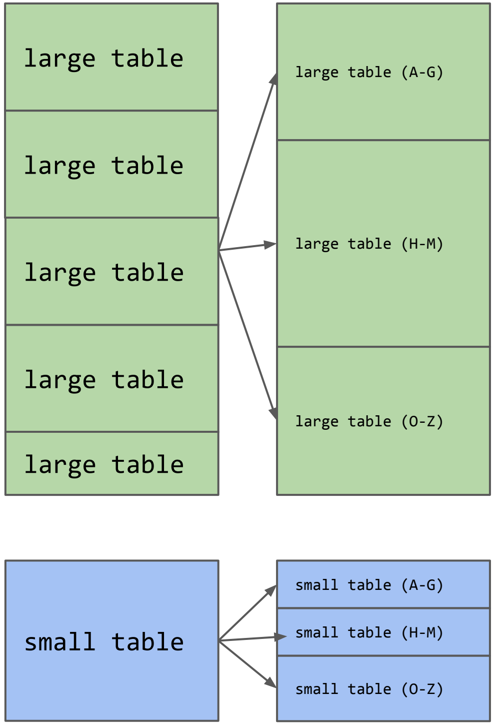
- `spark.sql.shuffle.partitions = 200`
- automatique partition coalesce
- coalesce partition according to table metrics between stage
- `spark.sql.adaptive.coalescePartitions.enabled = true`
- `spark.sql.adaptive.coalescePartitions.minPartitionNum = 200`
- `spark.sql.adaptive.coalescePartitions.initialPartitionNum = 200`
- `spark.sql.adaptive.advisoryPartitionSizeInBytes = 64Mb`

# Broadcast Join

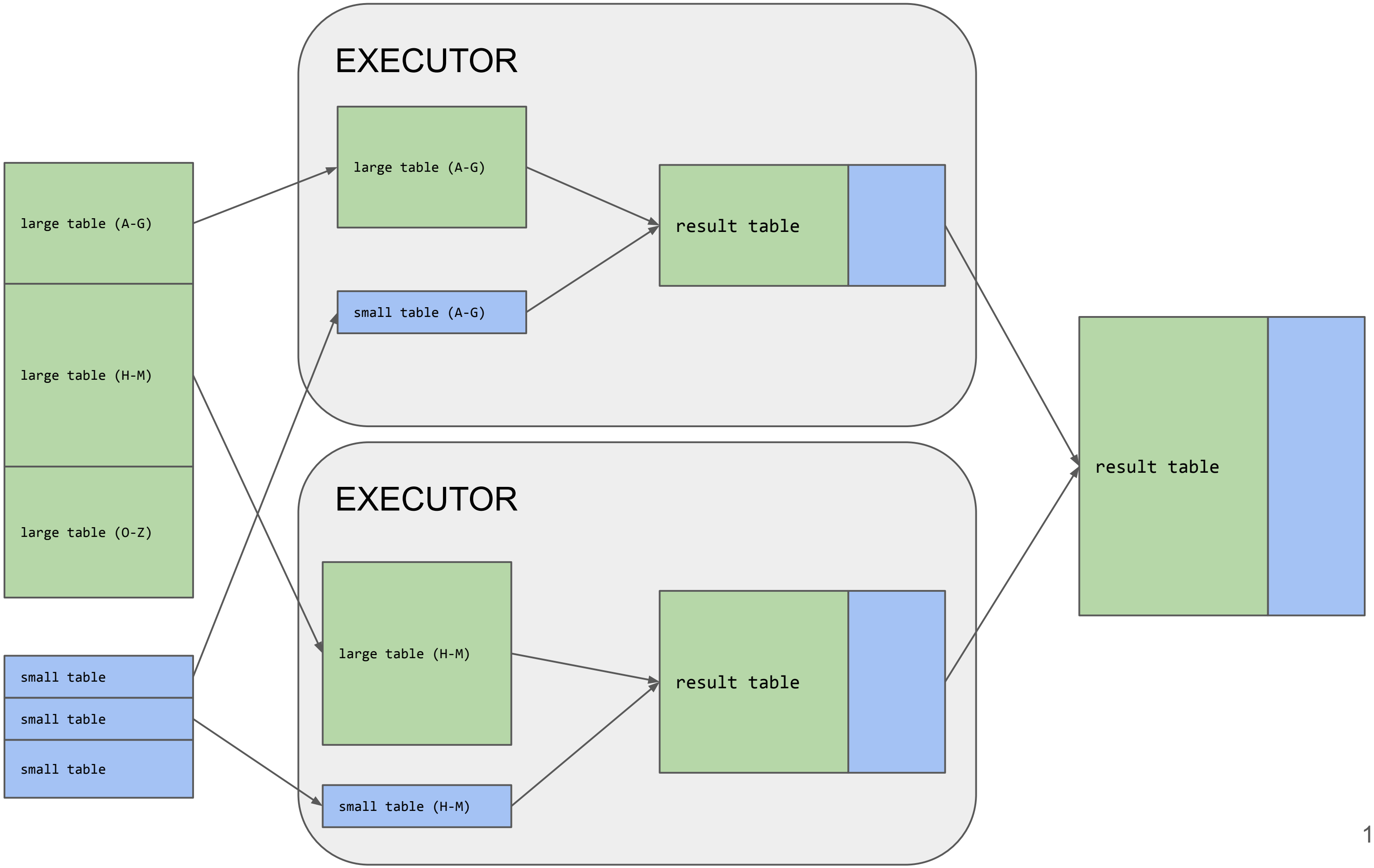
# Join



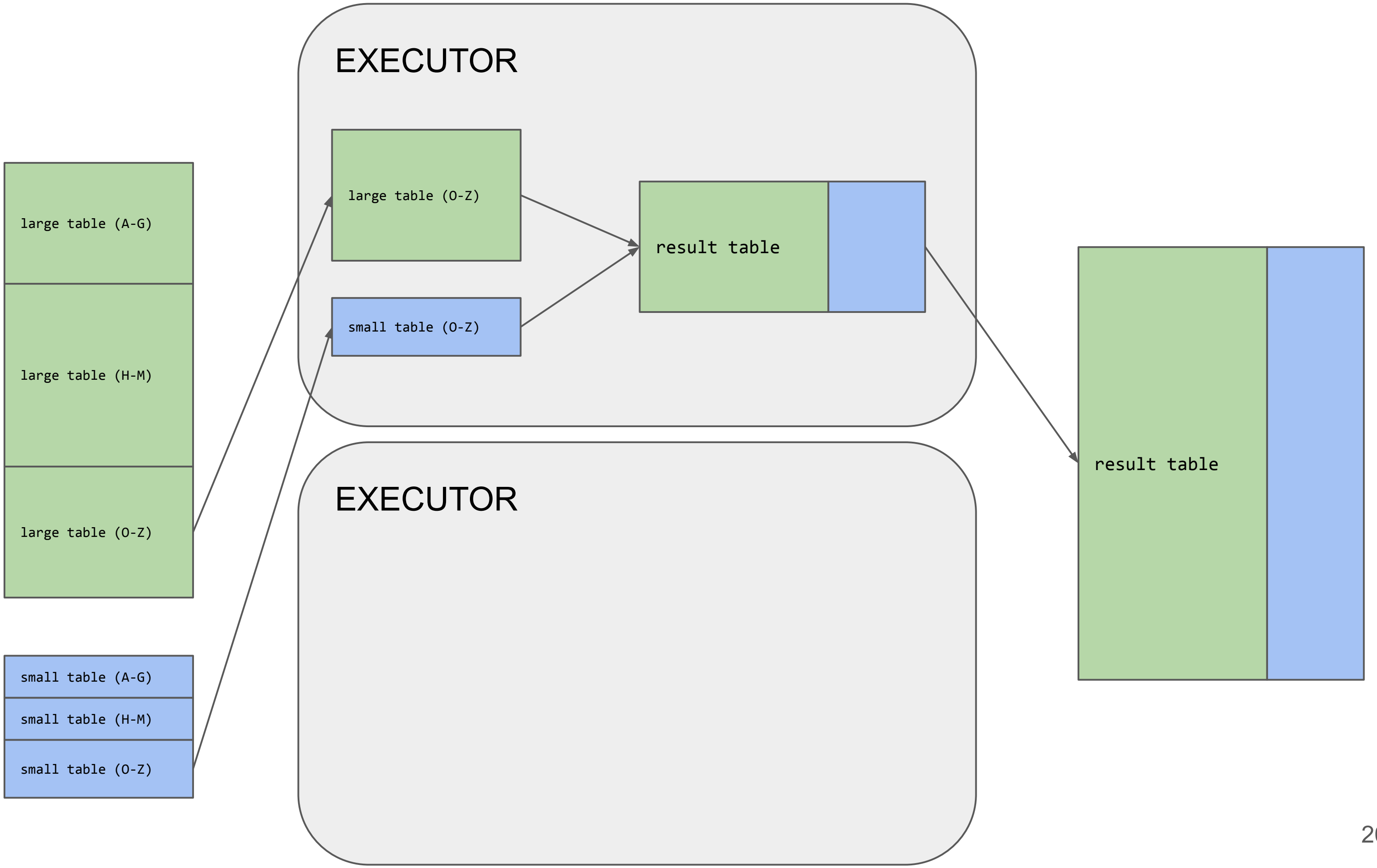
# Join


large table
large table
large table
large table
large table
large table (A-G)
large table (H-M)
large table (O-Z)
small table
small table (A-G)
small table (H-M)
small table (O-Z)

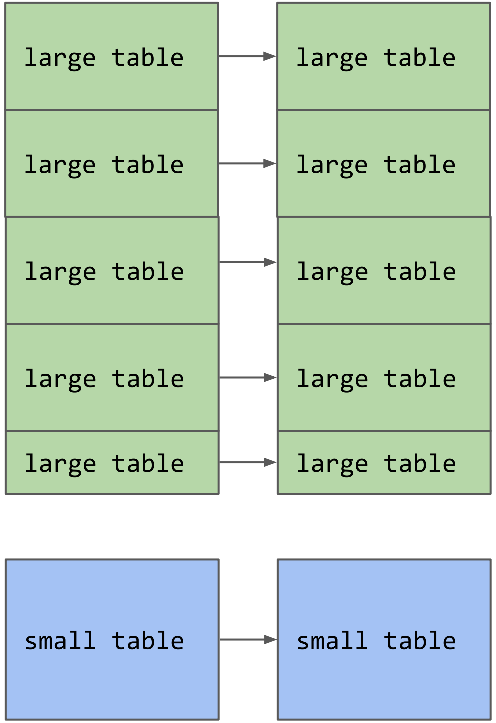
# Join


large table (A-G)
large table (H-M)
large table (O-Z)
small table
small table
small table
EXECUTOR
large table (A-G)
small table (A-G)
result table
EXECUTOR
large table (H-M)
small table (H-M)
result table
result table

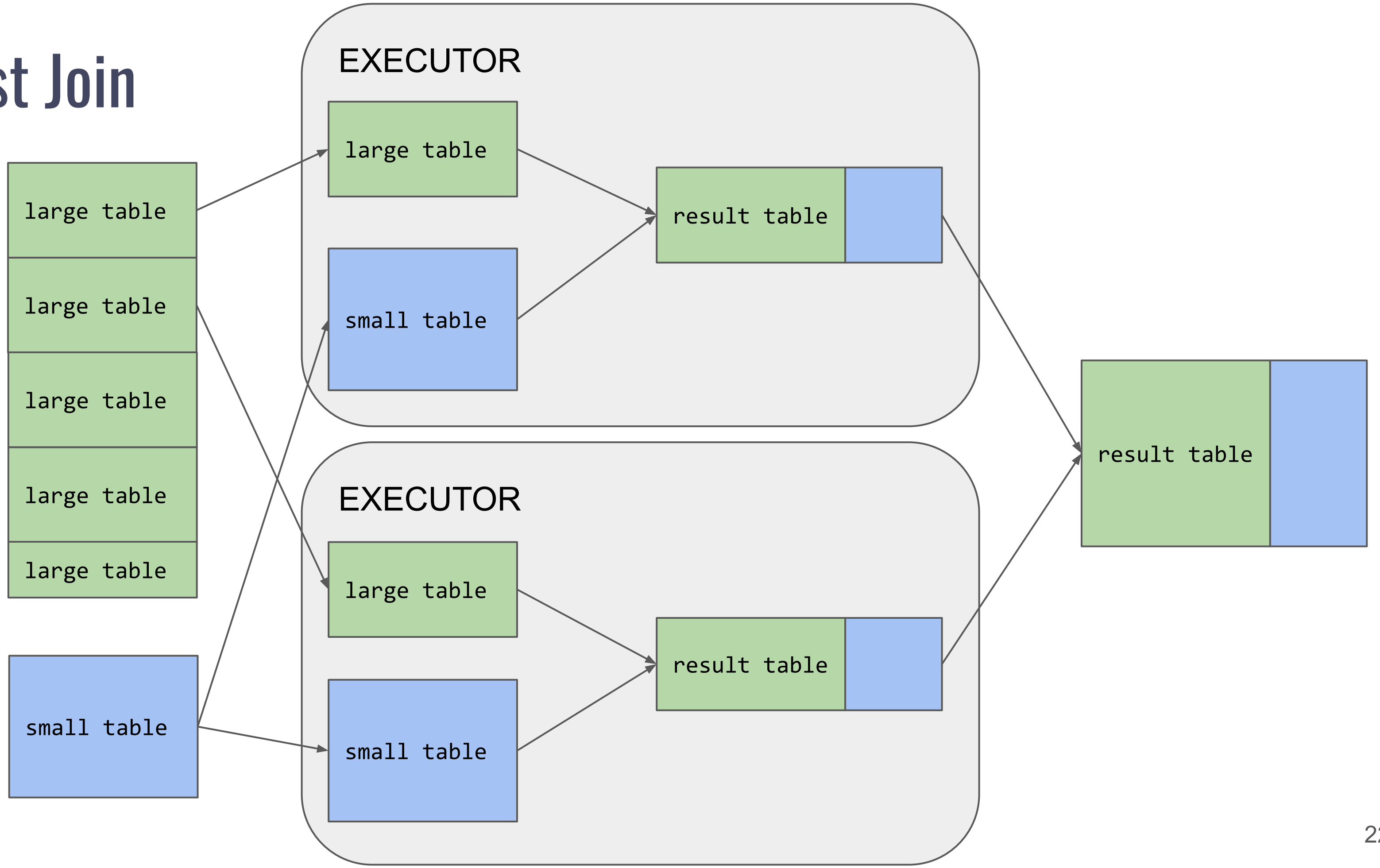
# Join


EXECUTOR
large table (A-G)
large table (H-M)
large table (O-Z)
small table (A-G)
small table (H-M)
small table (O-Z)
large table (O-Z)
small table (O-Z)
result table
EXECUTOR
result table

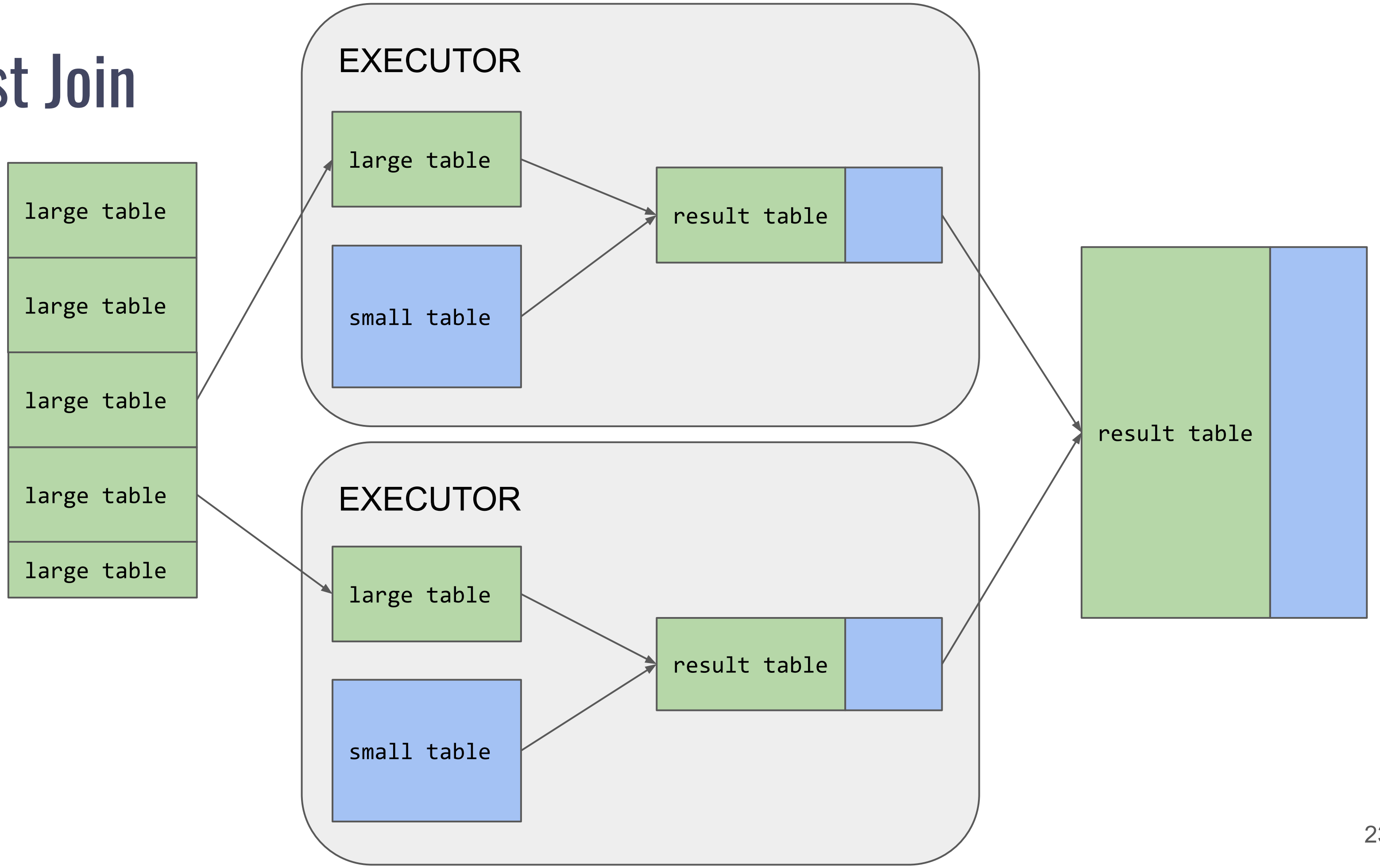
# Broadcast Join



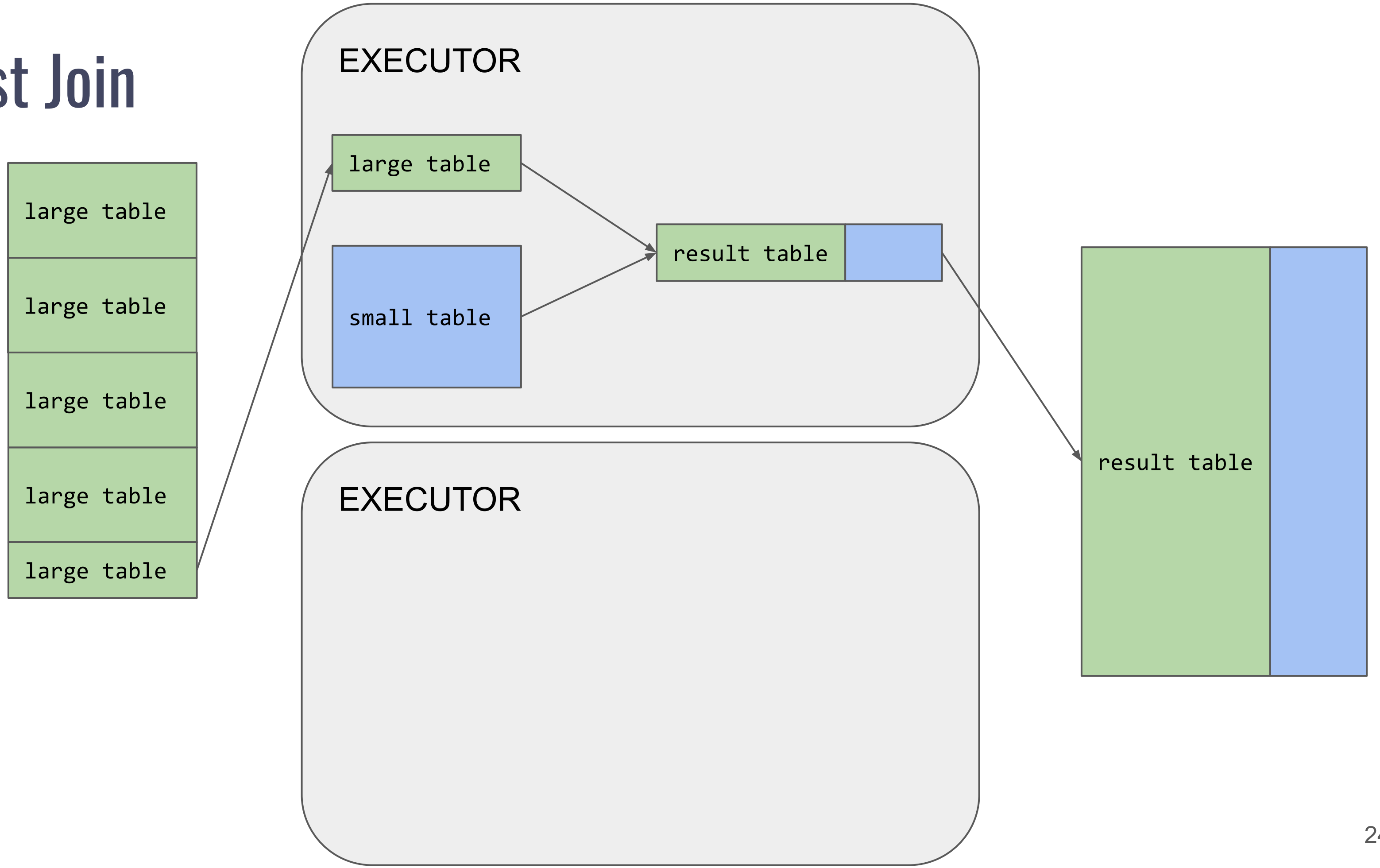
# Broadcast Join



# Broadcast Join



# Broadcast Join



# Broadcast join

- no shuffle
- the small table is sent (broadcast) to all executor
- the small table must fit in executor memory
- broadcast join is automatic
- `spark.sql.autoBroadcastJoinThreshold = 10Mb`

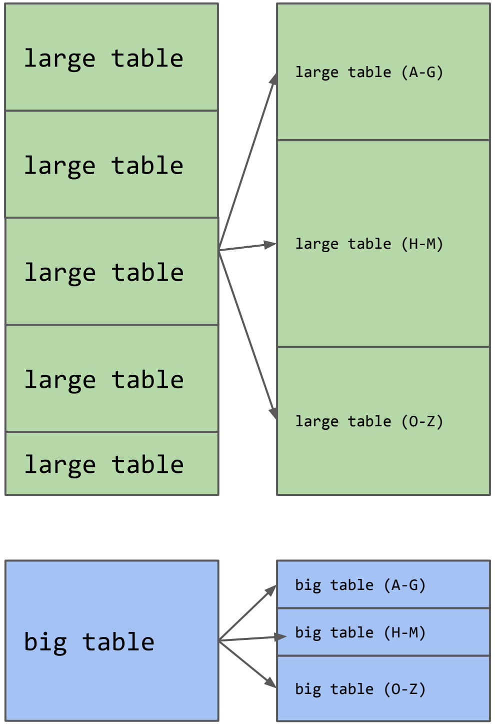
```
df1.join(broadcast(df2), Seq("column1"))
```

# Auto Broadcast Join

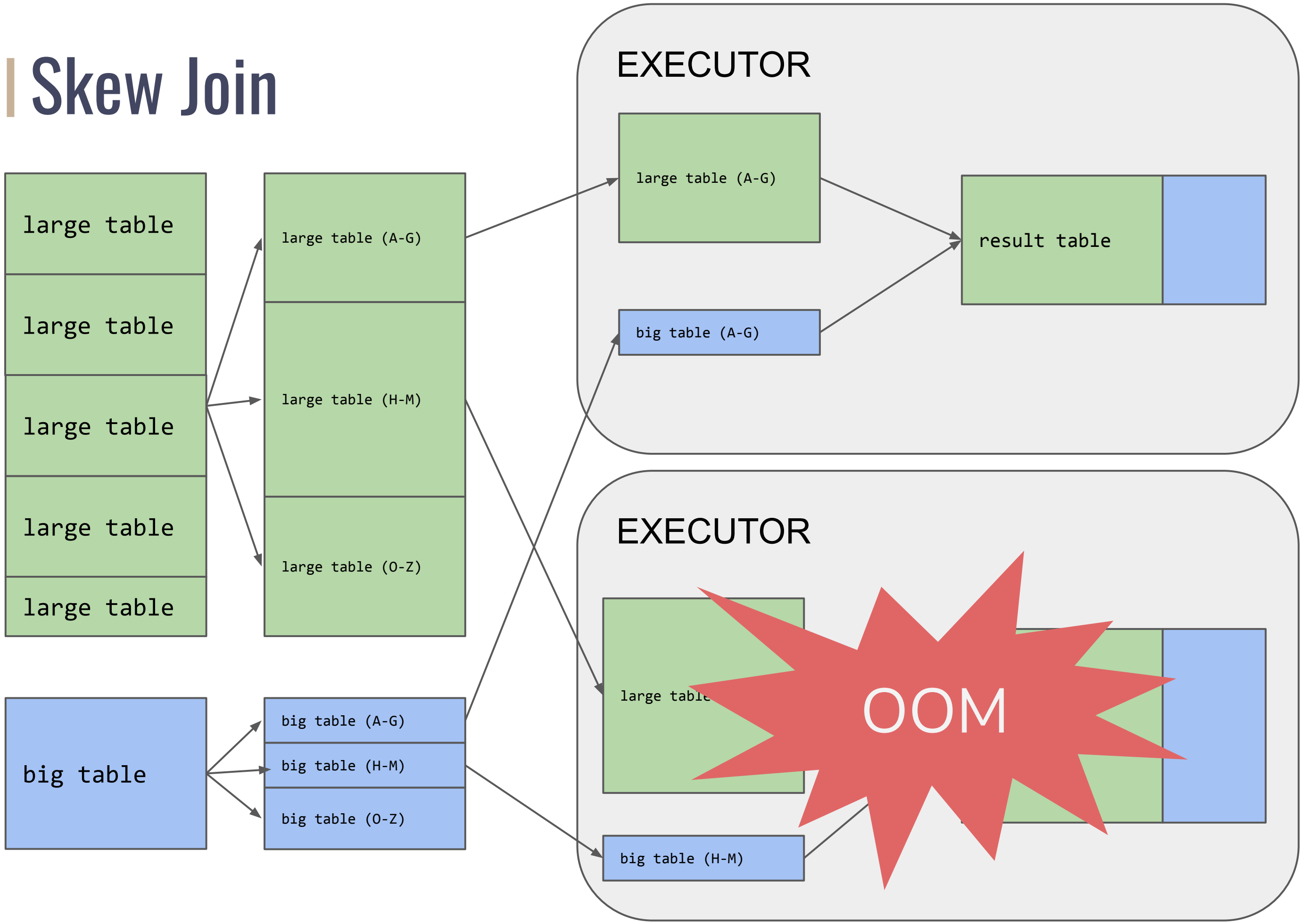
- with AQE enabled broadcast are smarter
- use inter stage table metrics to auto broadcast stable
- Without AQE auto broadcast only work with table

# Skew Join

# Skew Join

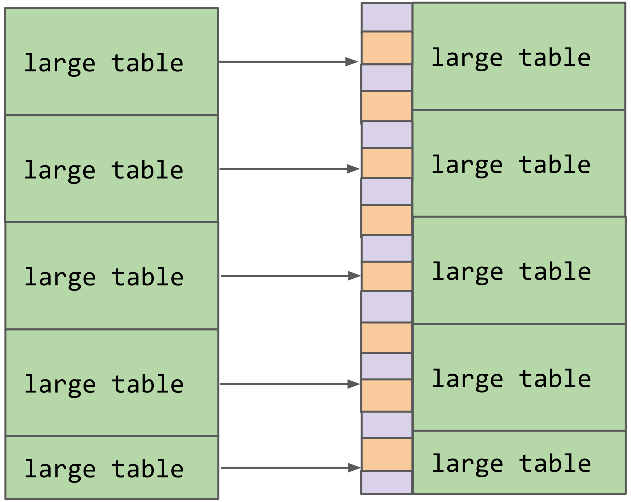

large table
large table
large table
large table
large table
large table (A-G)
large table (H-M)
large table (O-Z)
big table
big table (A-G)
big table (H-M)
big table (O-Z)

# Skew Join


large table
large table
large table
large table
large table
large table (A-G)
large table (H-M)
large table (O-Z)
big table
big table (A-G)
big table (H-M)
big table (O-Z)
EXECUTOR
large table (A-G)
big table (A-G)
result table
EXECUTOR
large table
big table (H-M)
OOM

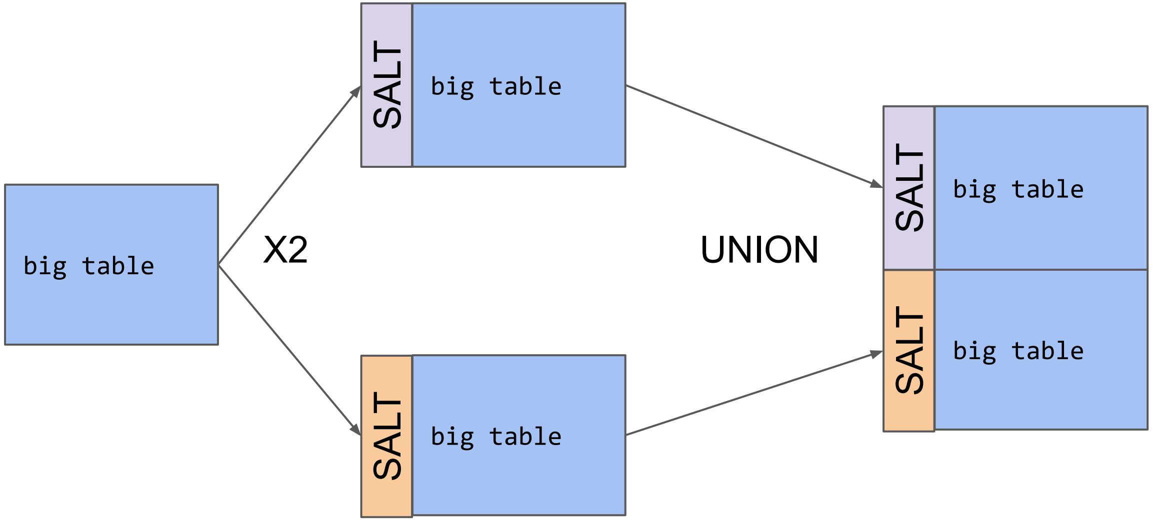
# Skew Join

we add a new column (salt), the value of this column is randomly chosen from the salt value (blue or orange)



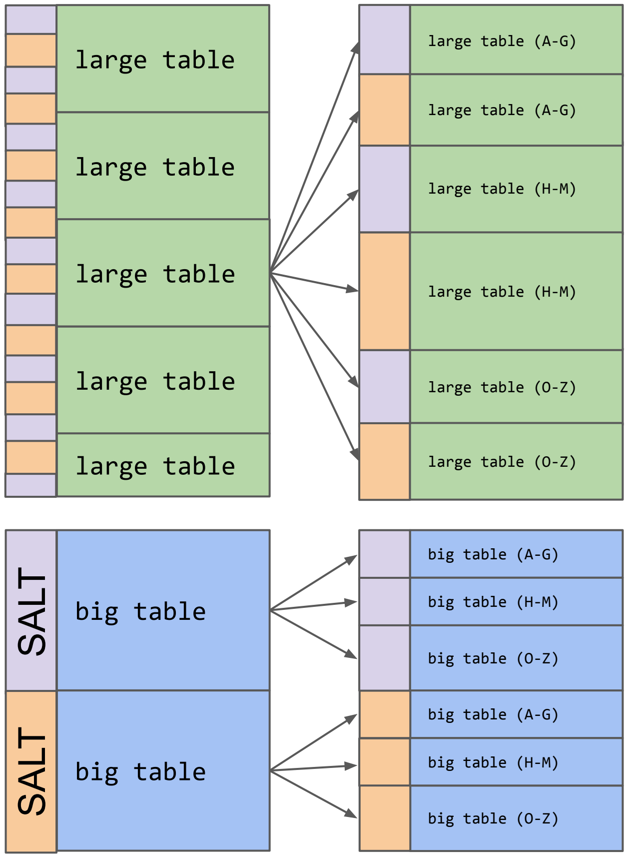
# Skew Join

Replicate the other table and add the value of the salt (bleu or orange)



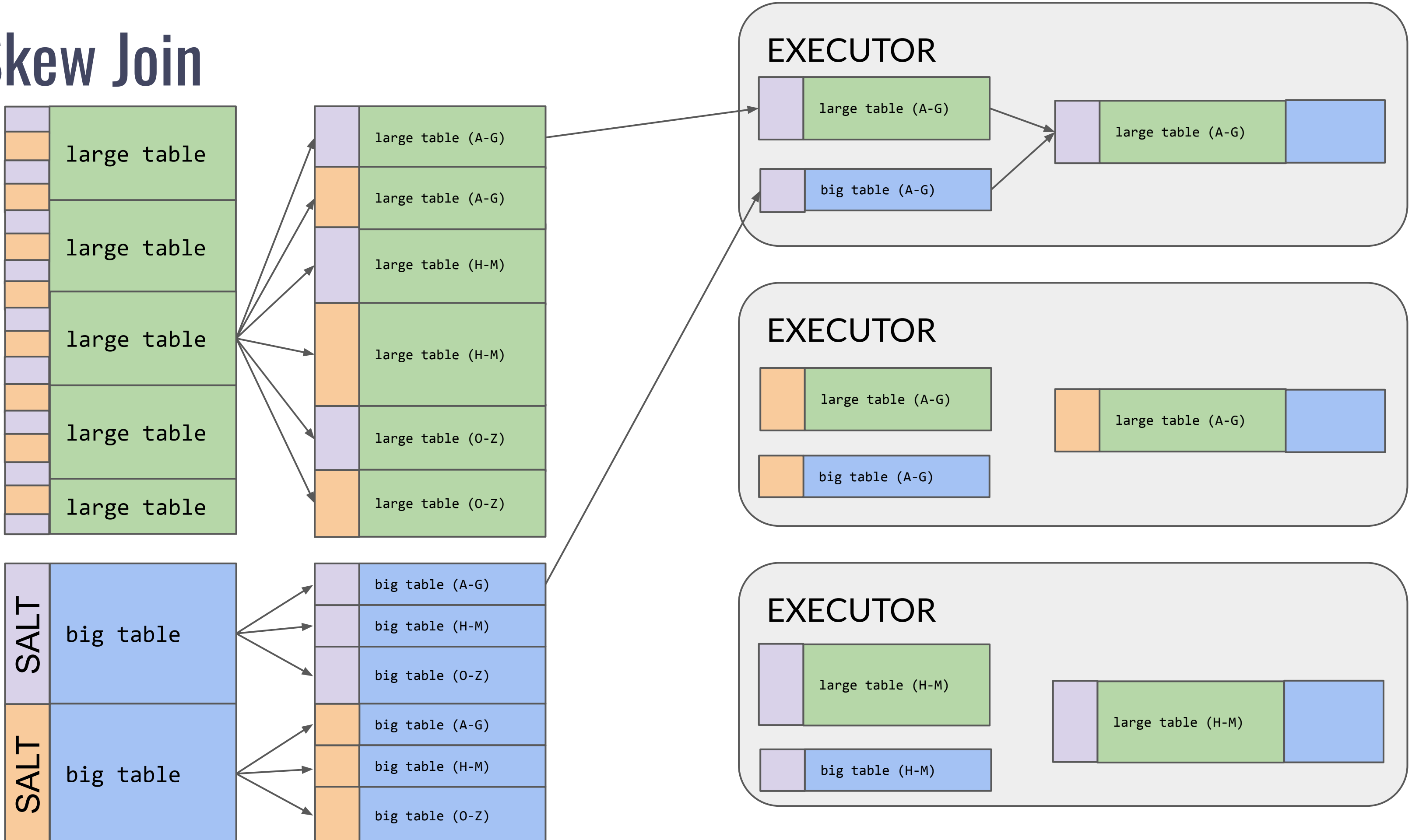
# Skew Join

To join we use our join key AND the salt




When shuffling before joining the partition H-M is smaller

# Skew Join

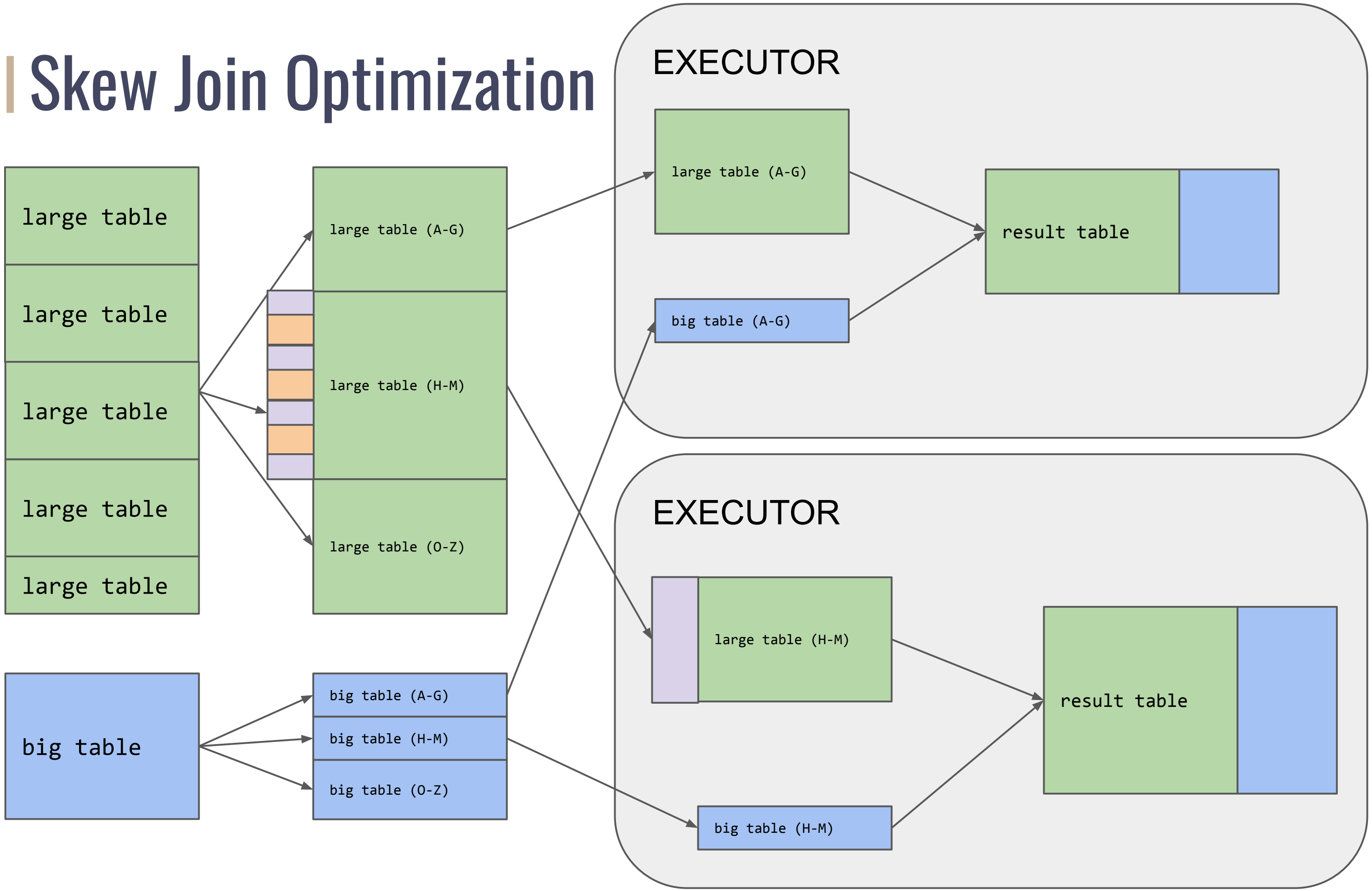

large table
large table
large table
large table
large table
large table (A-G)
large table (A-G)
large table (H-M)
large table (H-M)
large table (O-Z)
large table (O-Z)
SALT
big table
SALT
big table
big table (A-G)
big table (H-M)
big table (O-Z)
big table (A-G)
big table (H-M)
big table (O-Z)
EXECUTOR
large table (A-G)
big table (A-G)
large table (A-G)
EXECUTOR
large table (A-G)
big table (A-G)
large table (A-G)
EXECUTOR
large table (H-M)
big table (H-M)
large table (H-M)

# Manual Skew Join Drawback

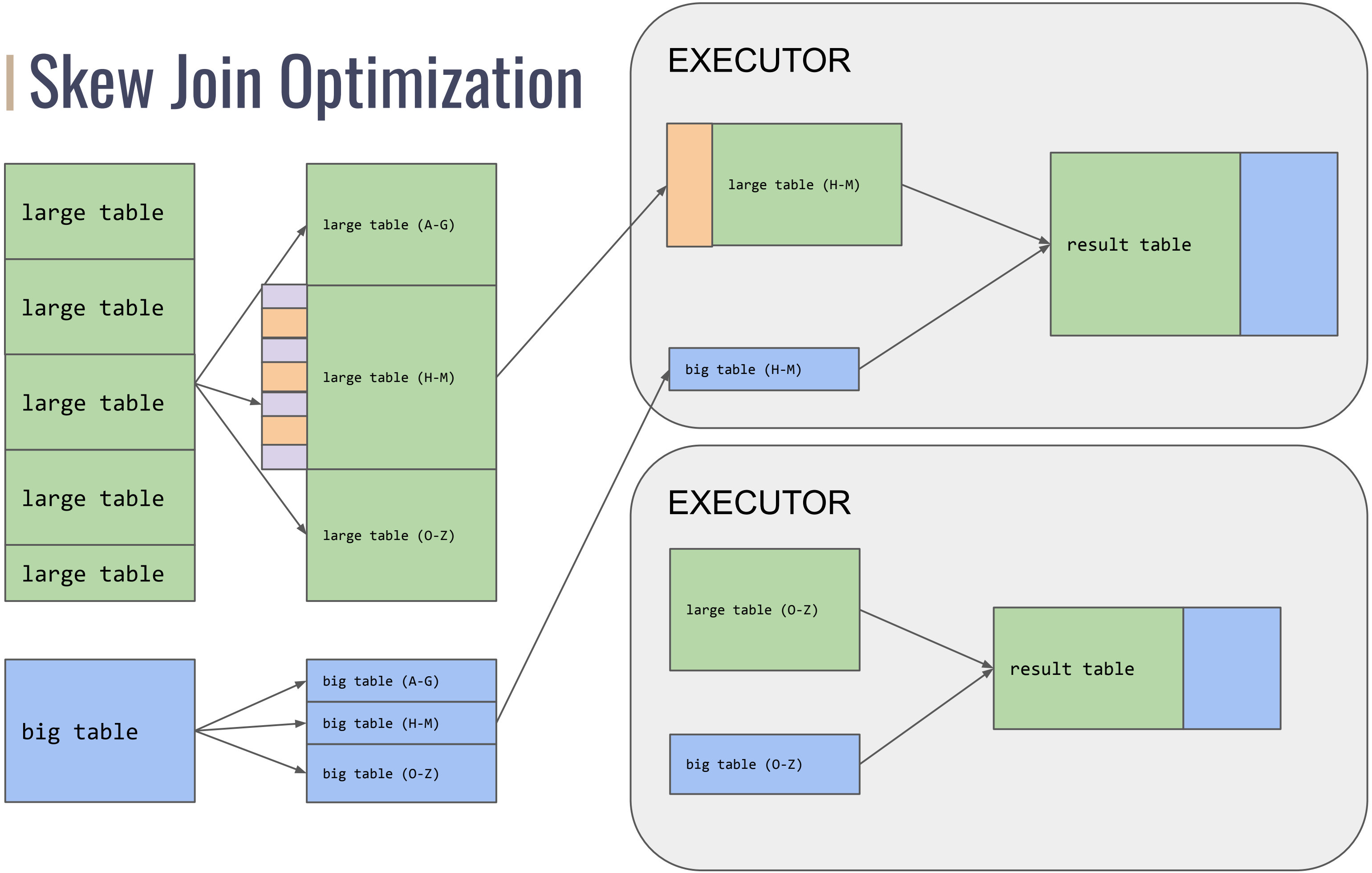
- duplicate a big table
- create smaller partition for already small partition
- You must add and remove your salt

# Skew Join Optimisation

# Skew Join Optimization



# Skew Join Optimization



# Adaptive Query Execution

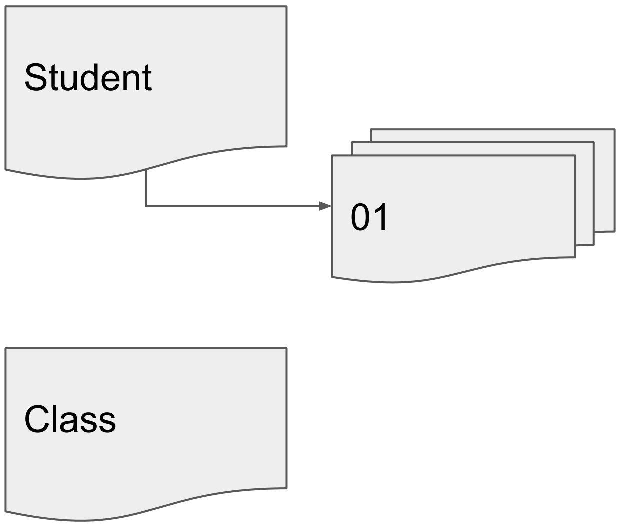
- split and duplicate only the skewed partition
- no more salt removal
- `spark.sql.adaptive.skewJoin.enabled = true`
- `spark.sql.adaptive.skewJoin.skewedPartitionFactor = 5`
- `spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes = 256Mb`

# Dynamic Partition Pruning

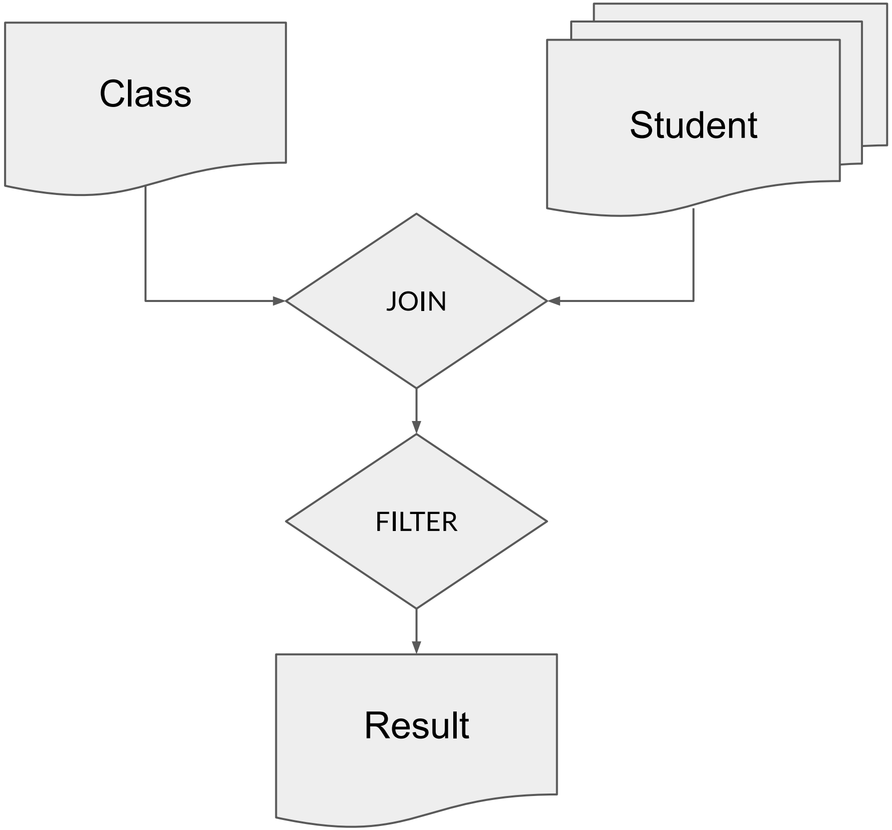
# Dynamic Partition Pruning

- only if
  - partitioned data
  - broadcast join
- filter the broadcasted table
- use the result to do a filter push down on the other side of the join
- independent from AQE

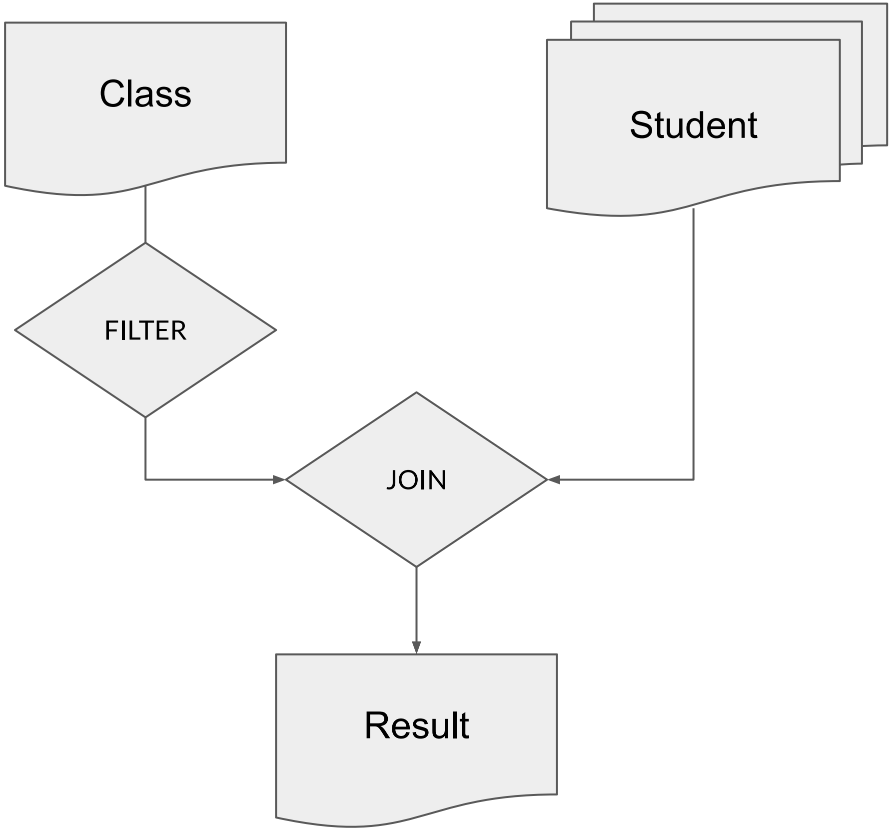
```
SELECT *
FROM Students
JOIN Class
WHERE Class.name = ‘6A’ AND year = 2014;
```



# Before Dynamic Partition Pruning



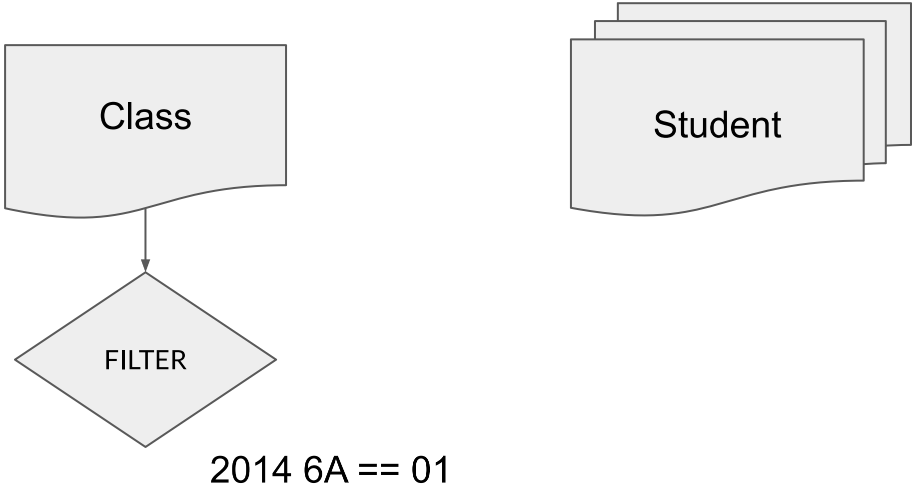
# Before Dynamic Partition Pruning



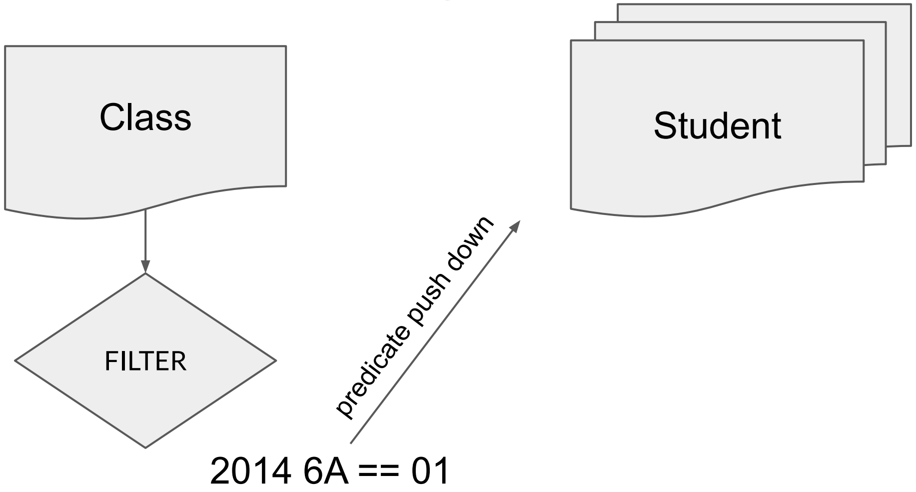
# Before Dynamic Partition Pruning

- Class table is small
- Student table is large
- Broadcast of Class table since it’s small
- Full scan of student table every time

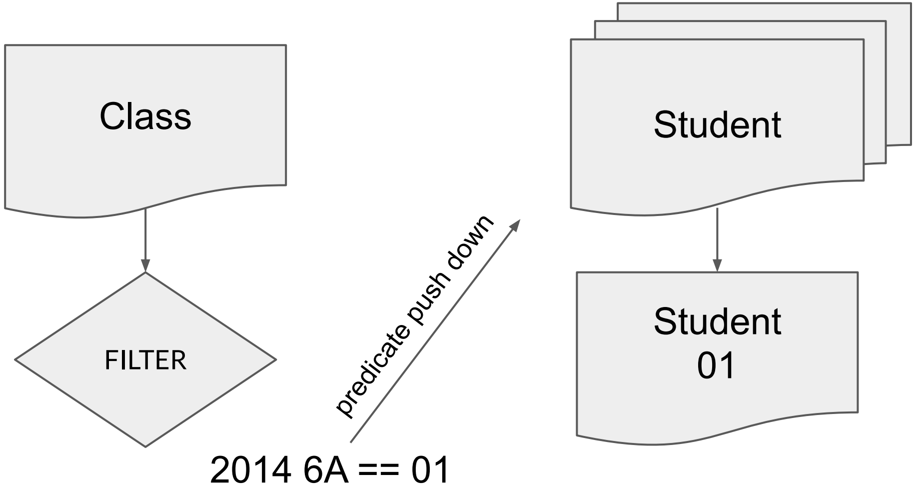
# After Dynamic Partition Pruning



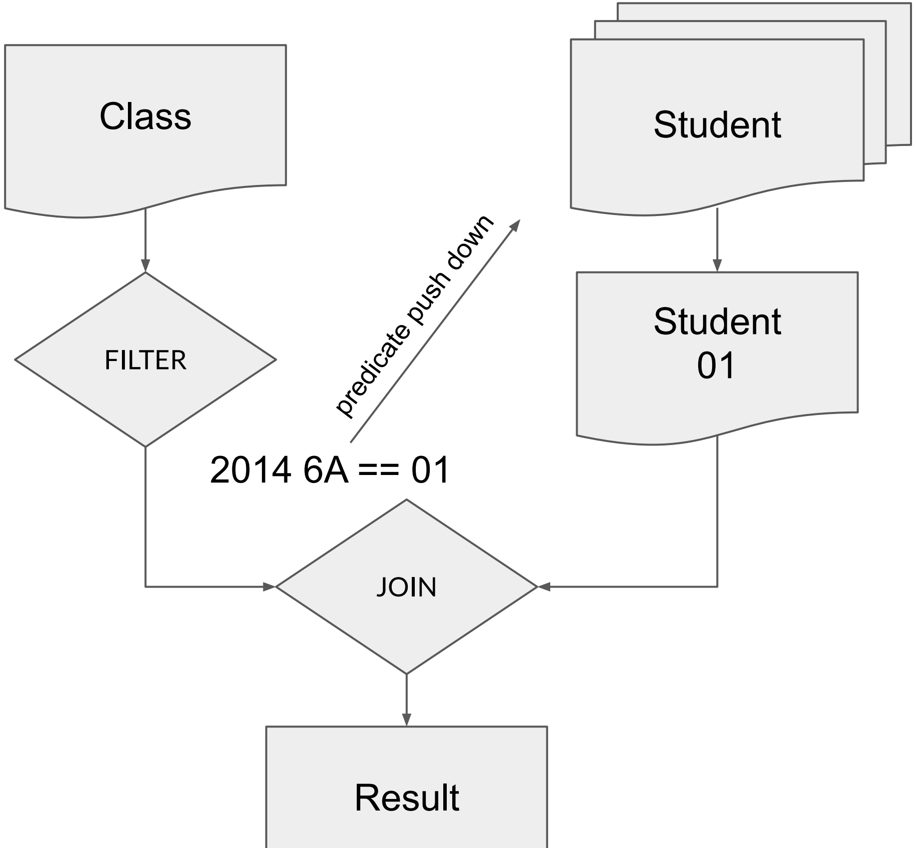
# After Dynamic Partition Pruning



# After Dynamic Partition Pruning



# After Dynamic Partition Pruning



# After Dynamic Partition Pruning

- Small table is filtered
- Result after filtering are used as predicate push down on the large table
- Large table is not fully scanned
- Fewer data during Join (Broadcast Join)

# The End

Question?