Optimizing Apache Spark

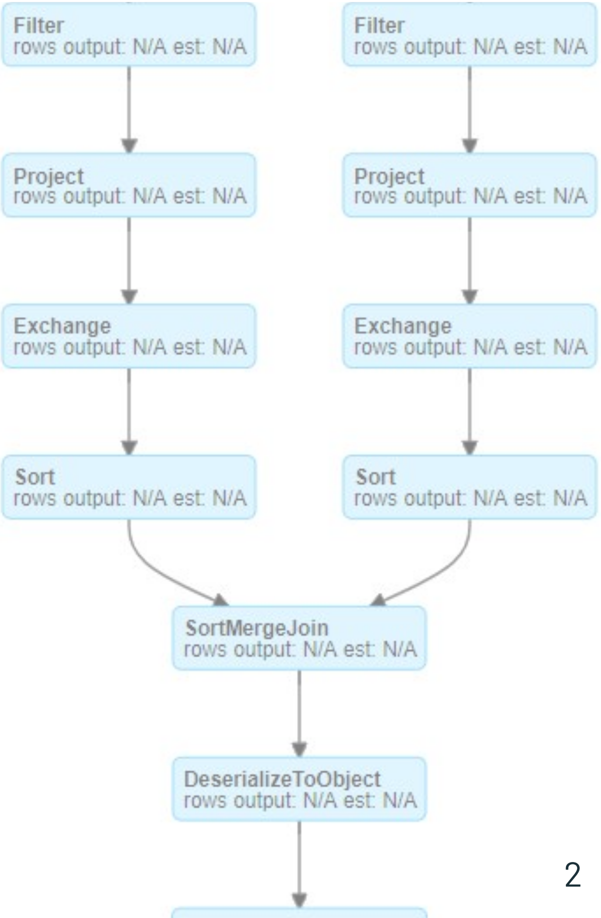
The Five Most Common Performance Problems

Shuffle

Shuffle

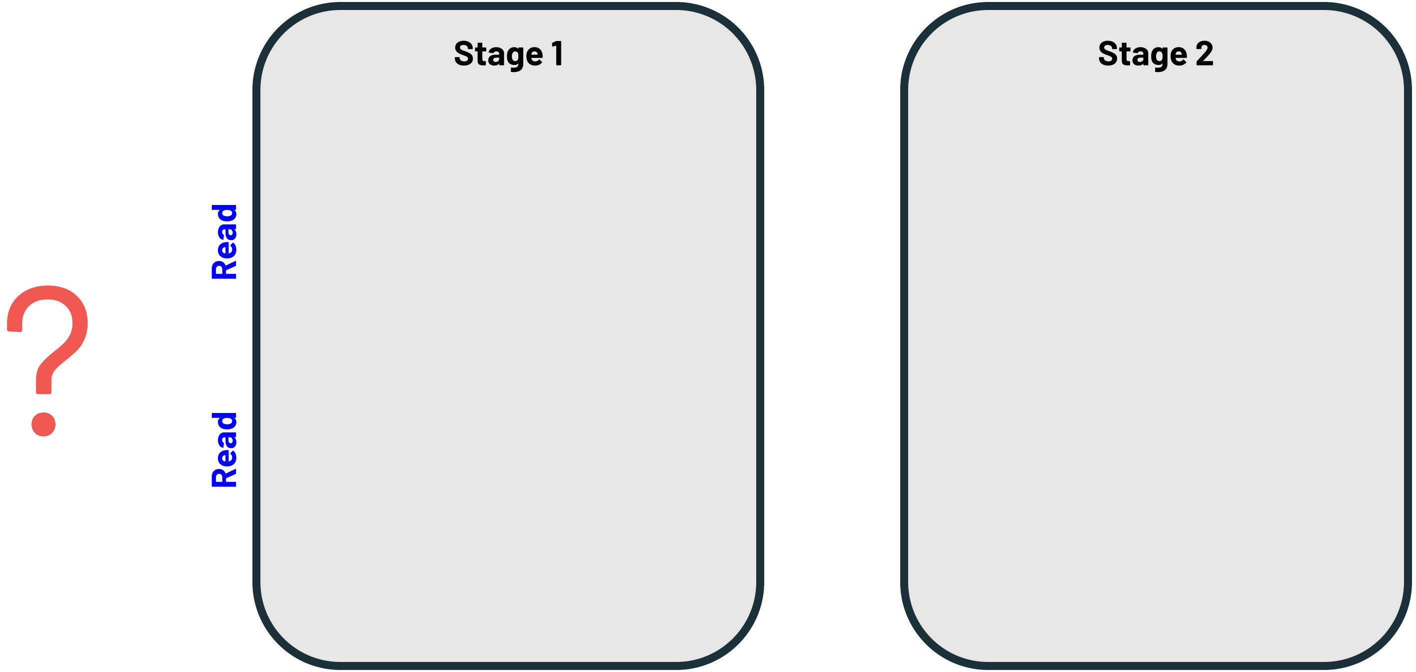
Shuffling is a side effect of wide transformation:
- **`join()`**
- **`distinct()`**
- **`groupBy()`**
- **`orderBy()`**

And technically some actions, e.g. **`count()`**

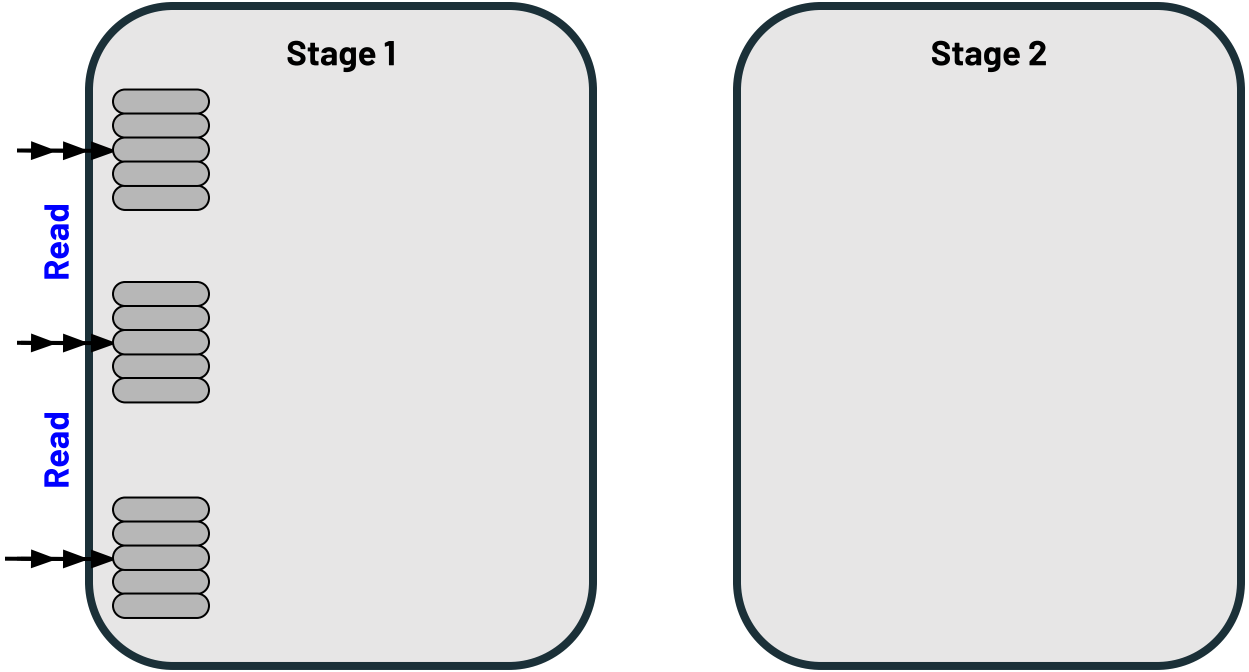


Let’s take a look at how a shuffle works...

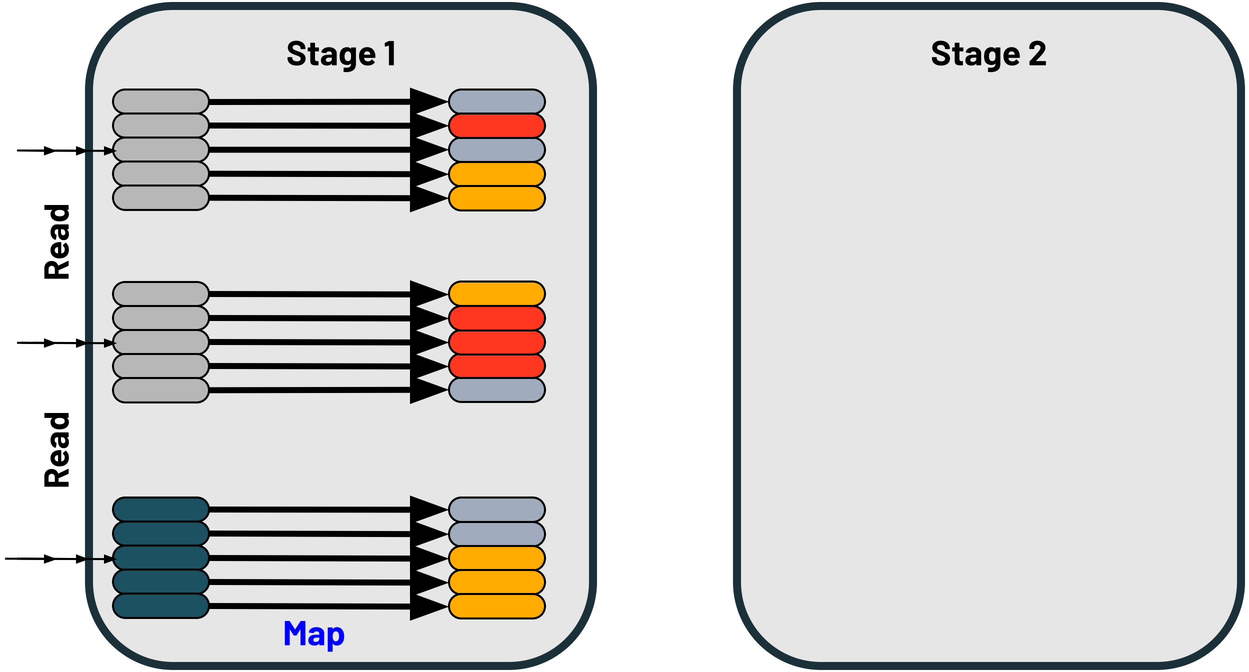
Step #1: From source or another stage, the process is the same



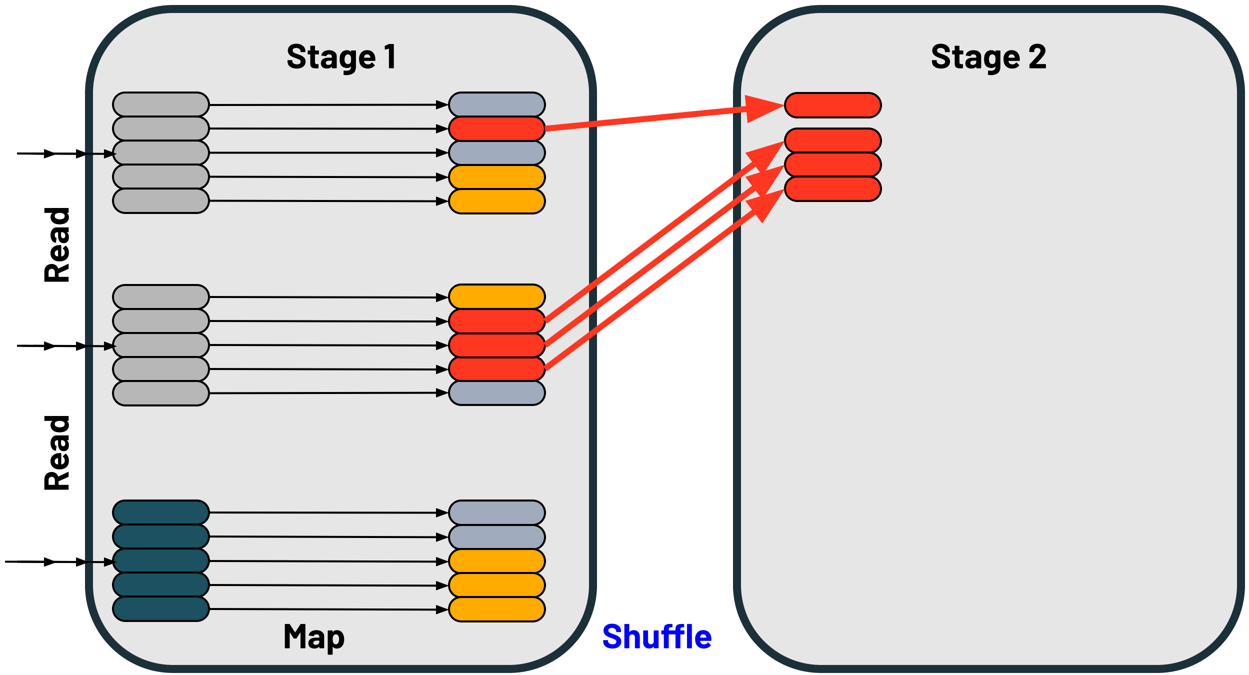
Step #2: Read the data into Spark-Partitions



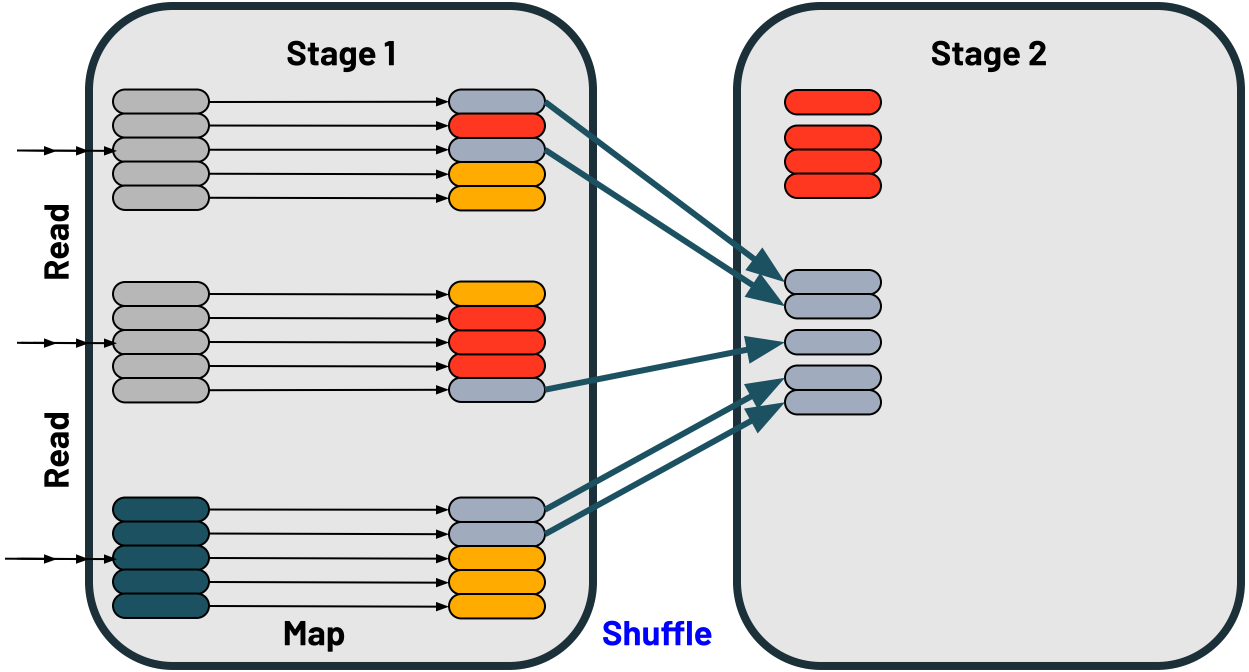
Step #3: Map reach record (e.g. by key) to a new partition



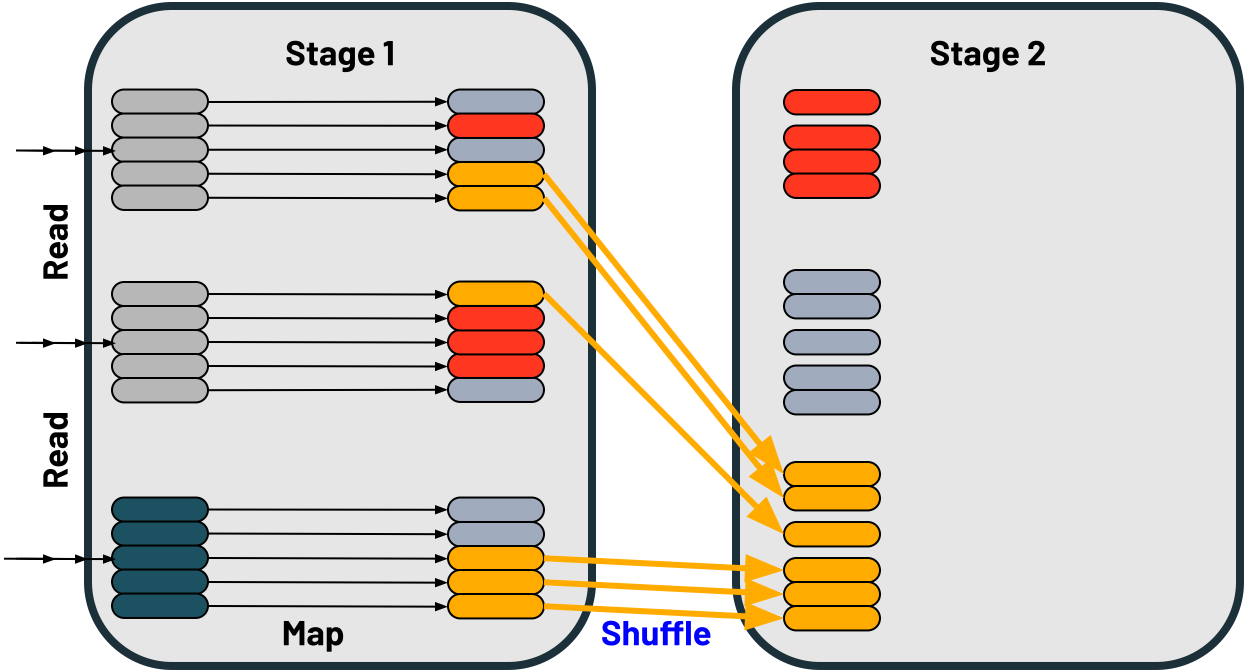
Step #4-A: Read the shuffle files into the next stage



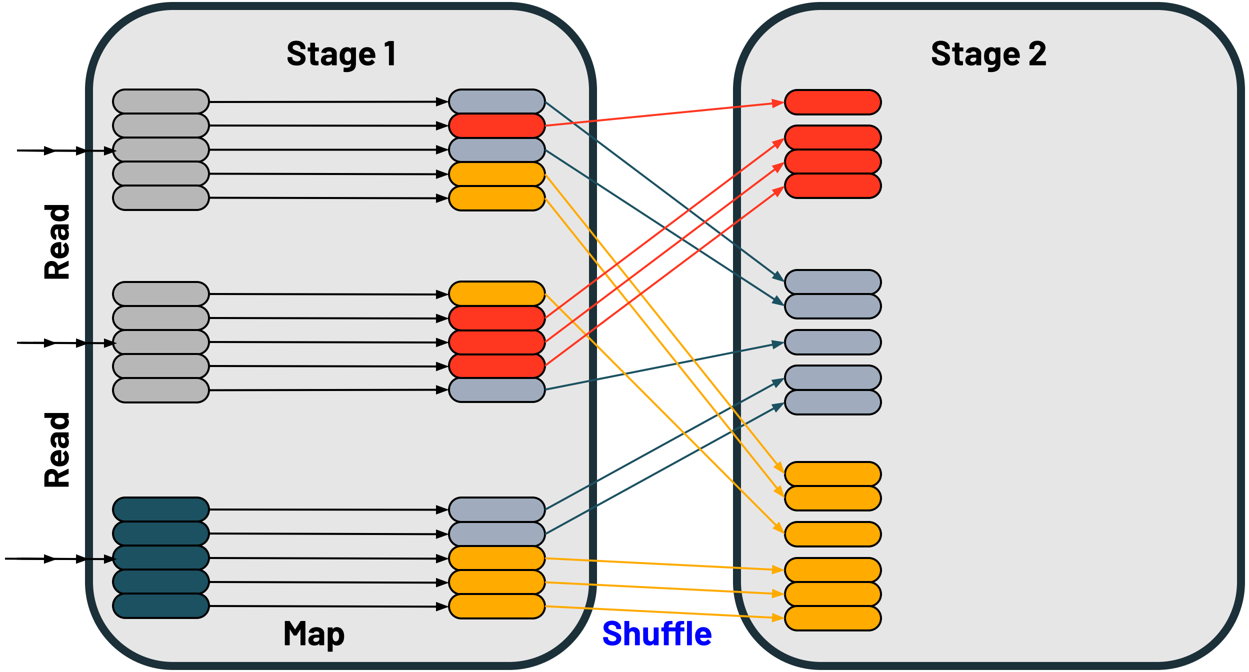
Step #4-B: Stage-1 would have written the shuffle files



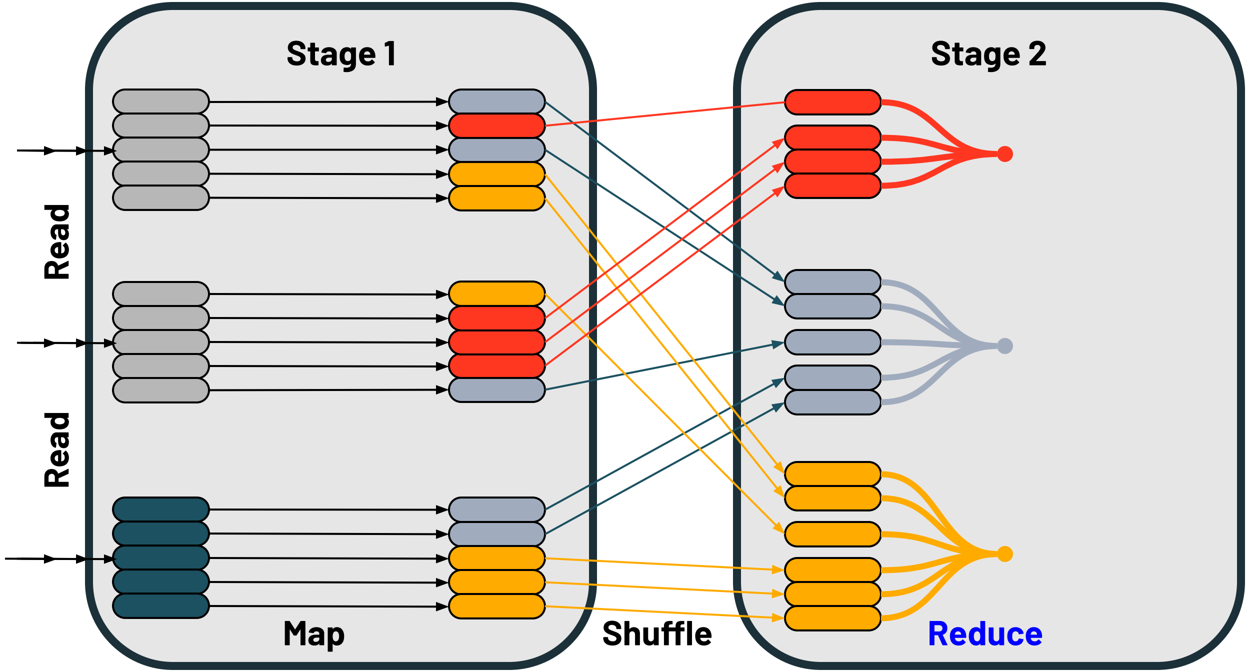
Step #4-C: Stage-2 would have read the shuffle files



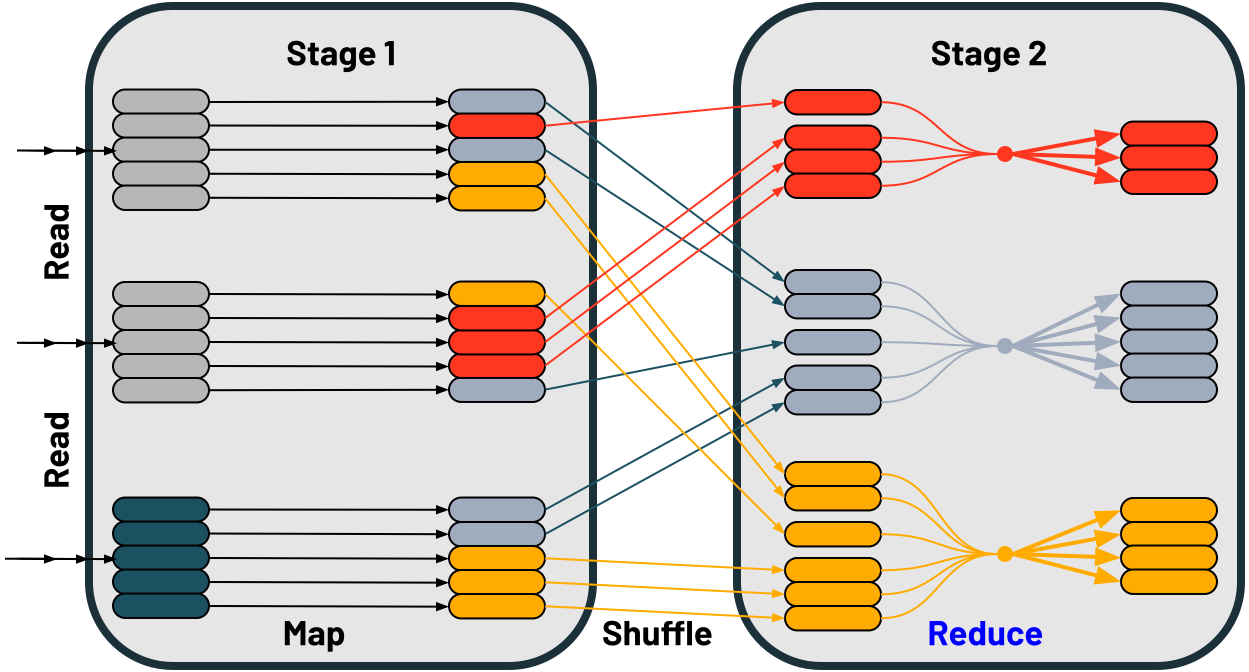
Step #4-D: Done simultaneously, this is a blocking operation



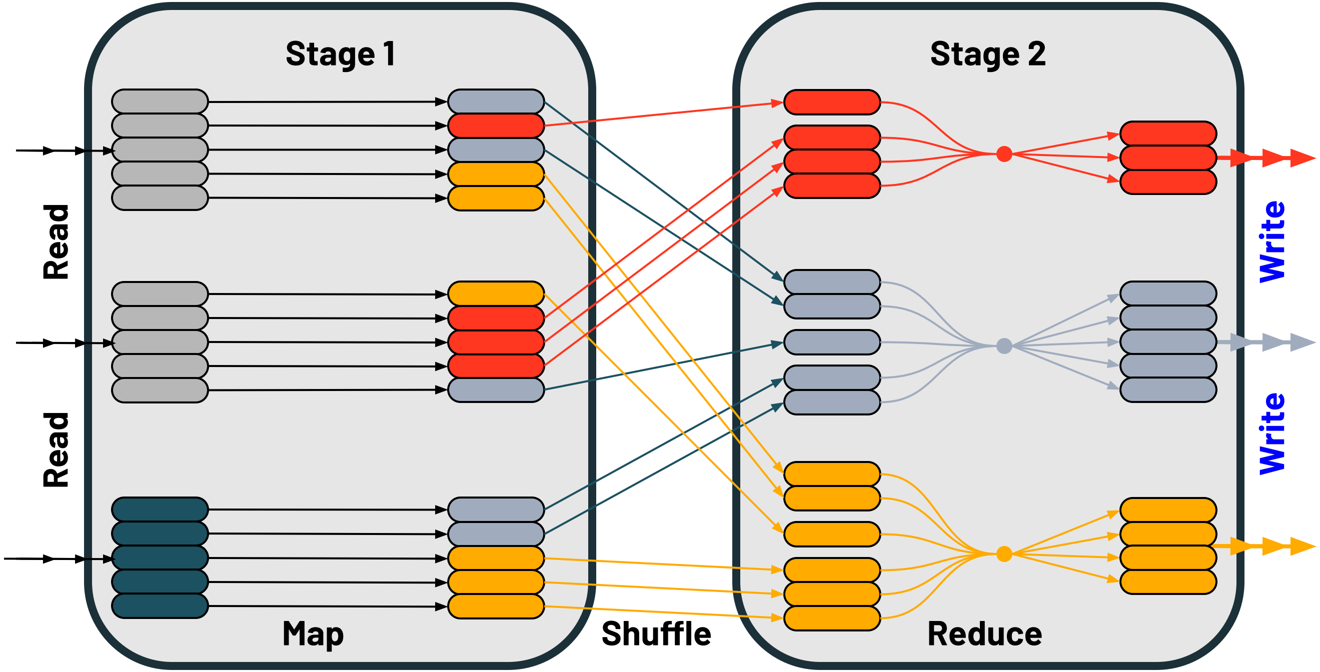
Step #5: The partitions are "reduced", how varies



Step #6: The final result is a new set of partitions



Step #7: New transformations can then be applied...



The 5 Most Common Performance Problems (The 5 Ss)
Shuffle - Not all the same

- The `distinct` operation aggregates many records based on one or more keys (the distinguisher) and reduces all duplicates to one record
- The `groupBy` / `count` combination aggregates many records based on a key and then returns one record which is the count of that key
- The `join` operation takes two datasets, aggregates each of those by a common key and produces one record for each matching combination (**total record count = max of a.count and b.count**)
- The `crossJoin` operation takes two datasets, aggregates each of those by a common key, and produces one record for every possible combination (**total record count = a.count x b.count**)

The 5 Most Common Performance Problems (The 5 Ss)

Shuffle - Similarities

- They read data from some source
- They aggregate records across all partitions together by some key
- The aggregated records are written to disk (shuffle files)
- Each executors read their aggregated records from the other executors
- This requires expensive disk and network IO

The 5 Most Common Performance Problems (The 5 Ss)

Shuffle - Being Pragmatic

There are some cases in which a shuffle can be avoided or mitigated

TIP: Don’t get hung up on trying to remove every shuffle

- Shuffles are often a necessary evil
- Focus on the [more] expensive operations instead
- Many shuffle operations are actually quite fast
- Targeting skew, spill, tiny files, etc often yield better payoffs

What can we do to mitigate the impact of shuffles?

The 5 Most Common Performance Problems (The 5 Ss)

Shuffle - Mitigation

The biggest pain with shuffle operations is the amount of data that is being shuffled across the cluster.

- Reduce network IO by using fewer and larger workers
- Reduce the amount of data being shuffled
  - Narrow your columns
  - Preemptively filter out unnecessary records
- Denormalize the datasets - especially when the shuffle is rooted in a join
  ...Spark 3 will most likely make this an anti-pattern for many cases

The 5 Most Common Performance Problems (The 5 Ss)

Shuffle - Mitigation Cont’

- Broadcast the smaller table
  - `spark.sql.autoBroadcastJoinThreshold`
  - `broadcast(tableName)`
  - Best suited for tables ~10 MB, but can be pushed higher
- For joins, pre-shuffle the data with a bucketed dataset
- Employ the Cost-Based Optimizer
  - Triggers other features like auto-broadcasting based on accurate metadata
  - Possibly negated by Spark 3 & AQE’s new features
  - See our presentation (The Apache Spark™ Cost-Based Optimizer) at https://youtu.be/WSIN6f-wHcQ

Optimizing Apache Spark

The Five Most Common Performance Problems

Shuffle Mitigation - BroadcastHashJoins

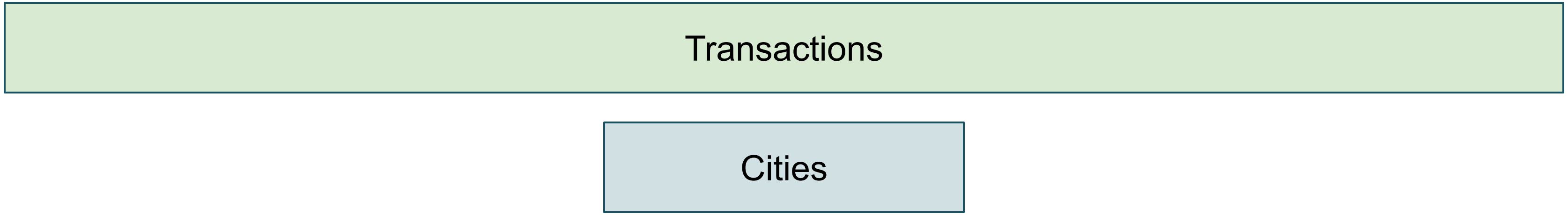
The 5 Most Common Performance Problems (The 5 Ss)

BroadcastHashJoins

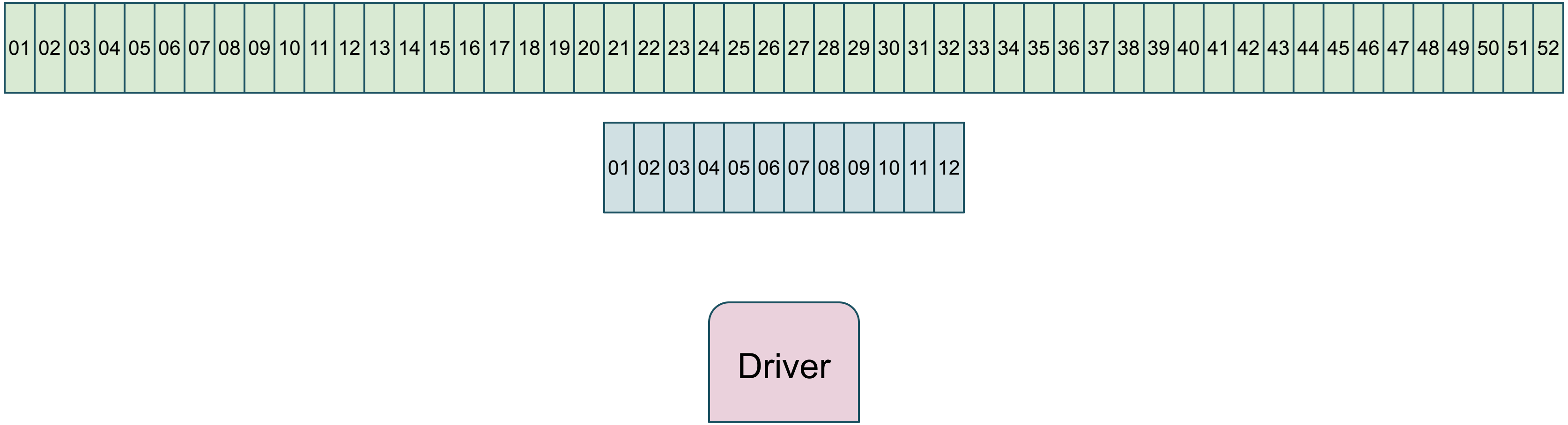
- **BroadcastHashJoins** are not a magic bullet
- The use cases are limited to small tables (under 10 MB by default)
- They can put undue pressure on the Driver resulting in OOMs
- In some cases, the alternative **SortMergeJoin** might be faster
- In general, Spark's automatic behavior might be your best bet

Let’s review how the BroadcastHashJoin works...

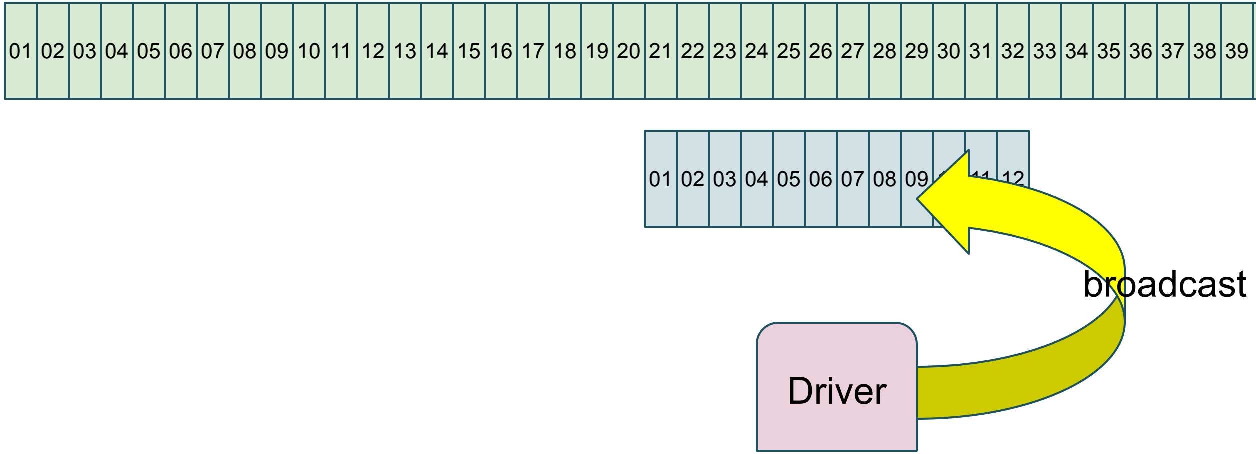
Presume we have two tables that we want to join based upon some common column



During planning the driver will partition our two datasets

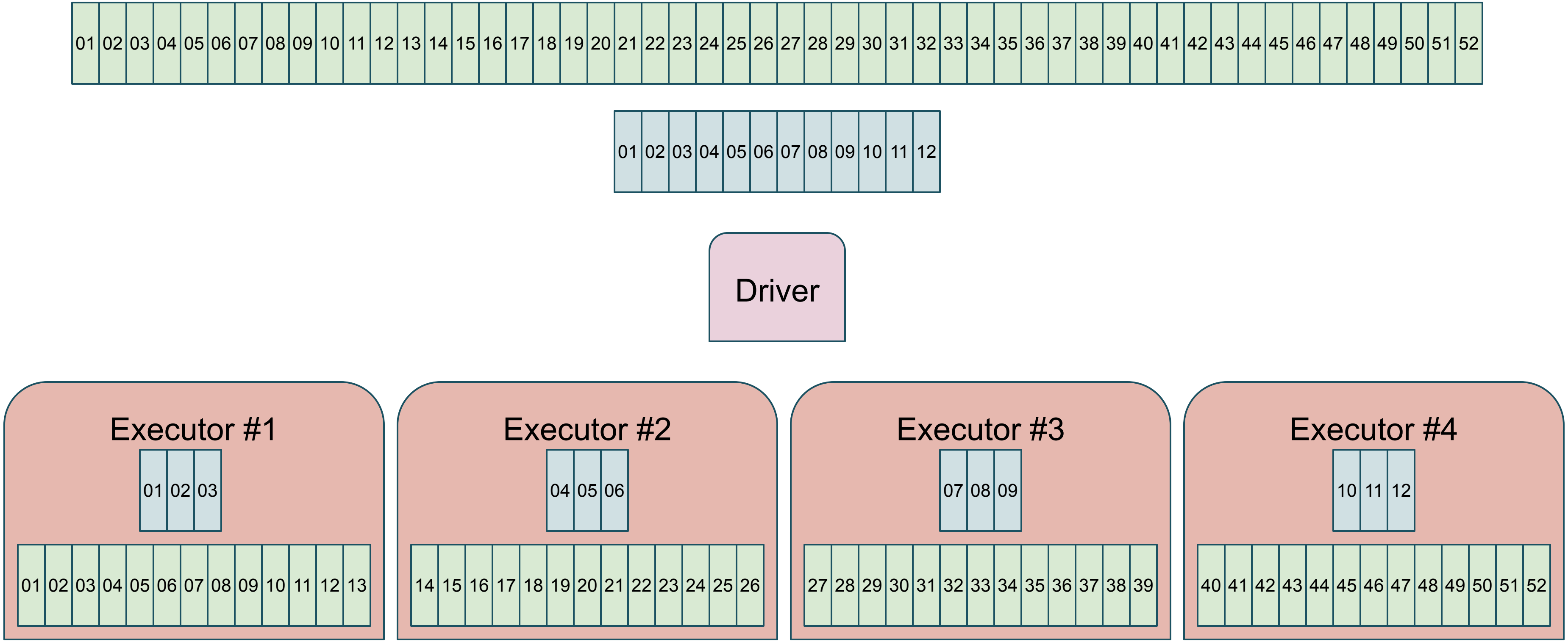


Because the cities table is < 10 MB, the Driver plans a BroadcastHashJoin



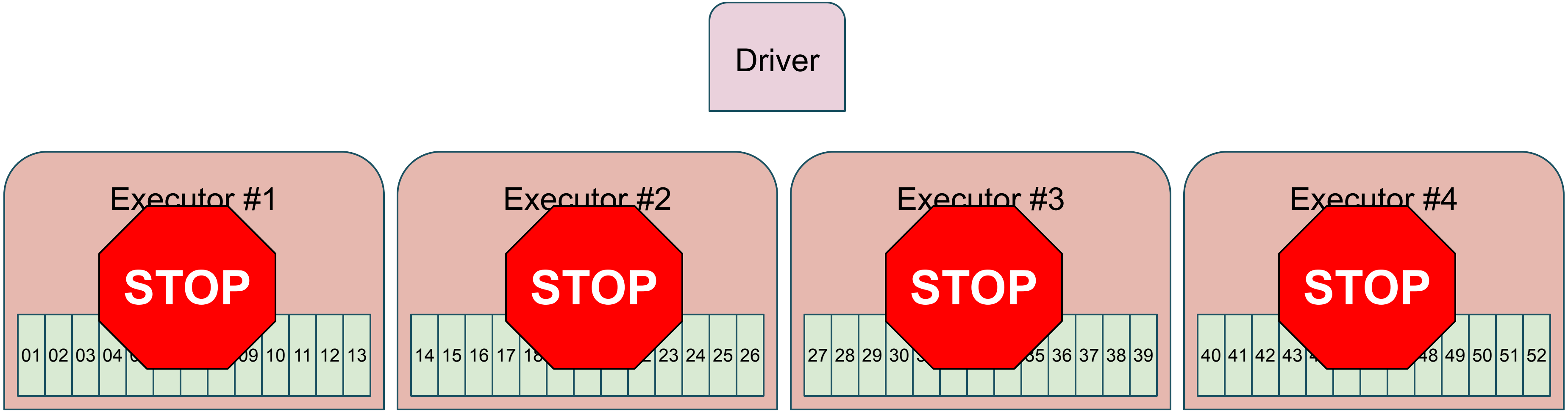
databricks

Each executor in turn reads in their assigned partitions



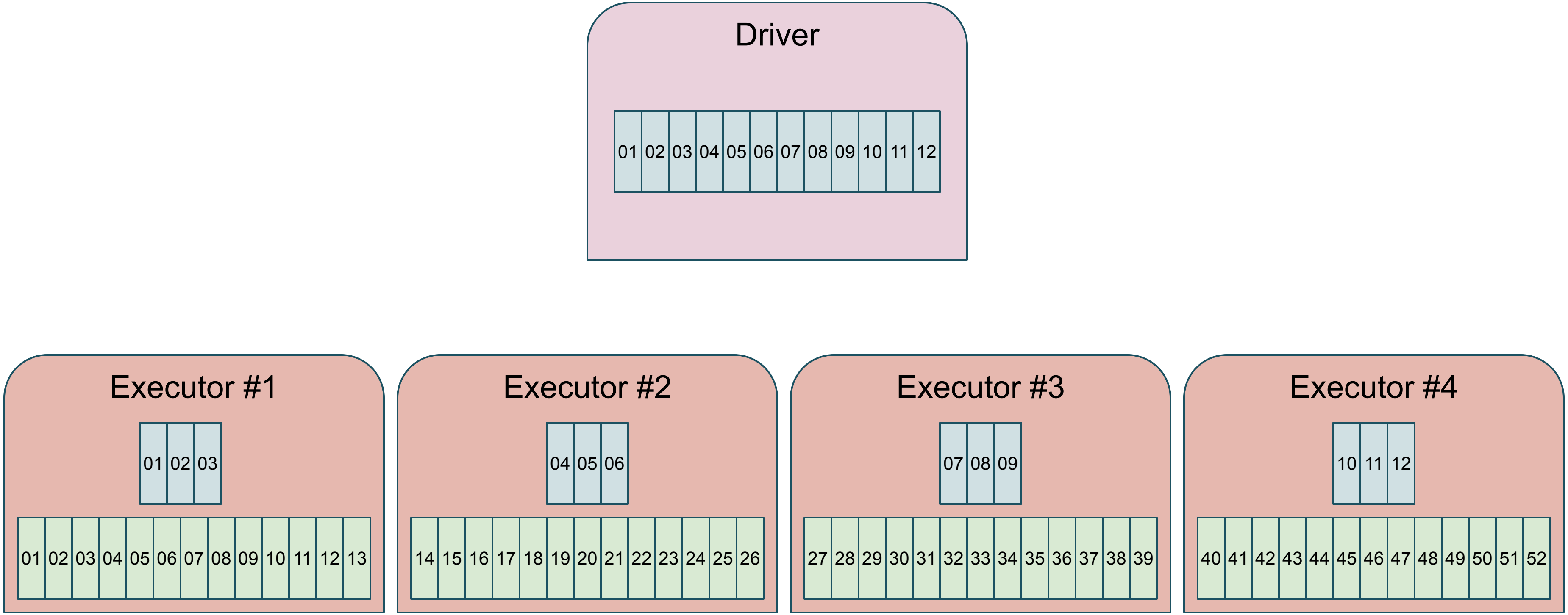
databricks

In a traditional join, we would proceed with the map and shuffle

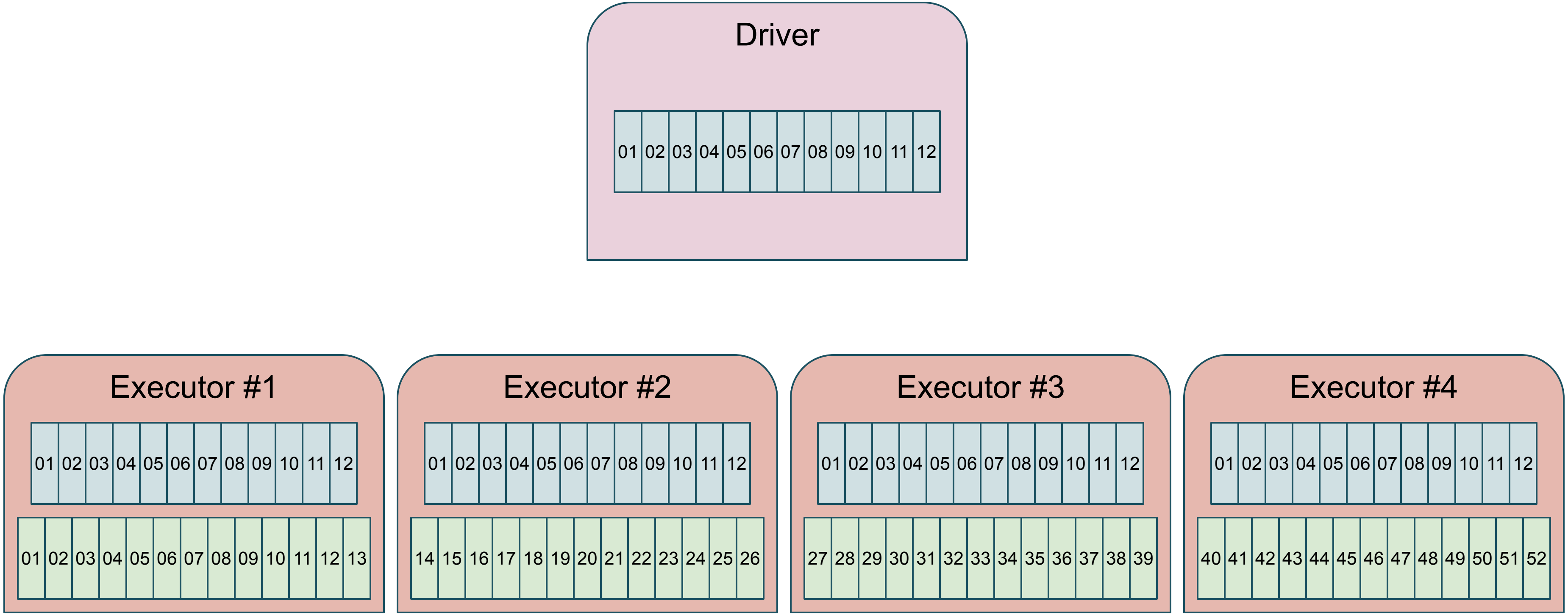


databricks

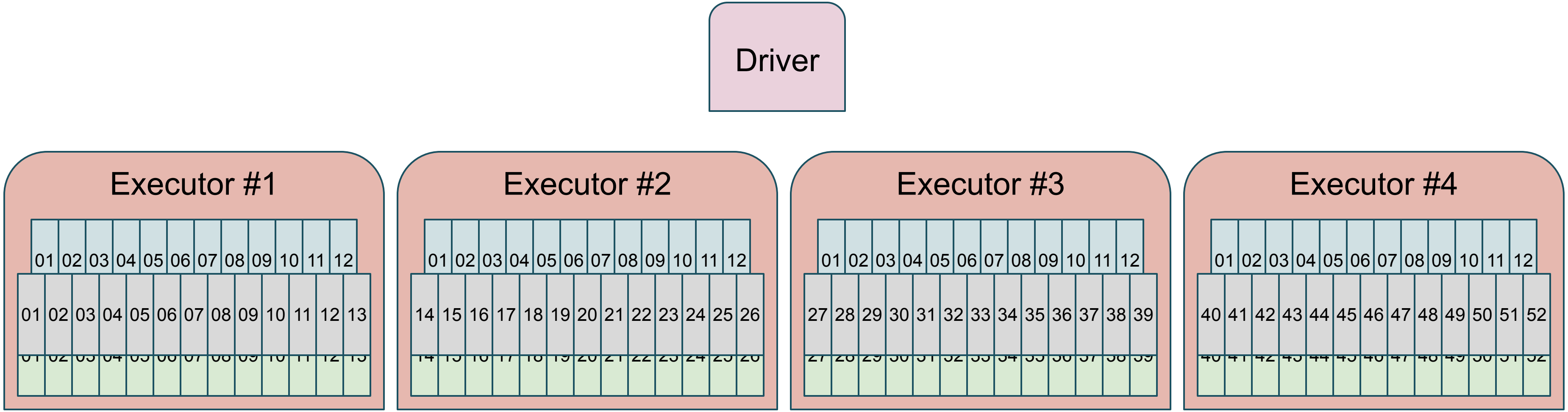
Instead, every partition of the the broadcasted table is sent to the driver



Next, a copy of the entire table is sent back to each executor



Lastly, each executor is able to join any two of its records because it has a complete copy of the broadcasted table



databricks

The 5 Most Common Performance Problems (The 5 Ss)

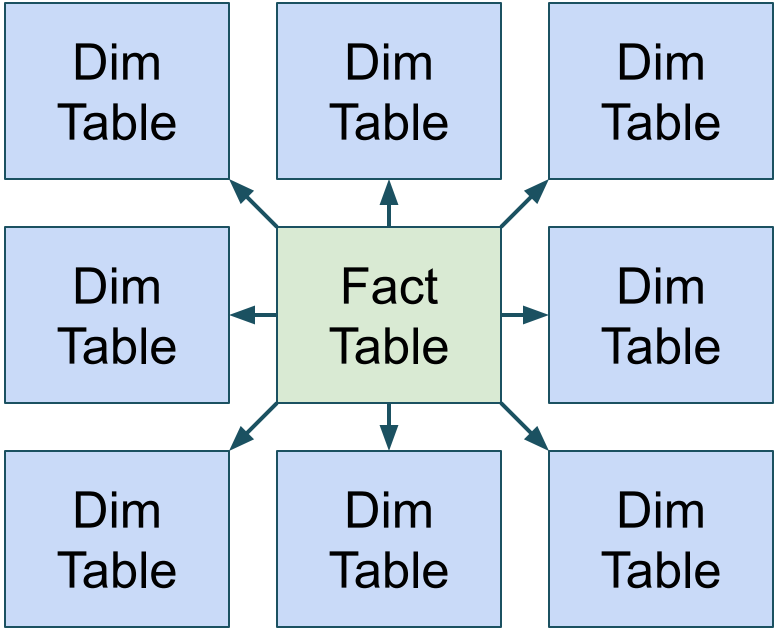
BroadcastHashJoins - Dangers

- Note the high level of IO between the Driver and Executors
- With small tables (e.g. around 10 MB), the cost is lower than the exchange
- When pushed to higher limits (say 100 MB), the balances start to shift
- Similarly, many empty partitions can adversely affect the BHJ
- The Driver & Executors both require enough RAM to receive the fully broadcasted table
- Performance depends on the relative scale of the left and right table

The 5 Most Common Performance Problems (The 5 Ss)

BroadcastHashJoins - w/Many Dim Tables

Even if you don't push the 10 MB limit, joining to many small tables can produce excessive load on the Driver & Executors resulting in GC delays and OOM Errors



The 5 Most Common Performance Problems (The 5 Ss)

BroadcastHashJoins vs SortMergeJoin

BroadcastHashJoin	SortMergeJoin
Avoids shuffling the bigger side	Shuffles both sides
Naturally handles data skew	Can suffer from data skew
Cheap for selective joins	Can produce unnecessary intermediate results
Broadcasted data needs to fit in memory	Data can be spilled and read from disk
Cannot be used for certain outer joins	Can be used for all joins
Overhead of E→D→E is high with few/large executors	Outperforms BHJ with few/large executors

The 5 Most Common Performance Problems (The 5 Ss)

BroadcastHashJoins - Going Deeper

- We encourage you to see the talk by **Jianneng Li**
  - Improving Broadcast Joins in Apache Spark
  - Presented at the **Spark-AI Summit 2020**

- He proposes the idea of an Executor-Side Broadcast
  - Based on Spark-17556
  - Instead of moving the data to the Driver, it is shuffled between Executors

- He also shares some interesting computations on how to predict when a SMJ might outperform the BHJ

Optimizing Apache Spark

The Five Most Common Performance Problems

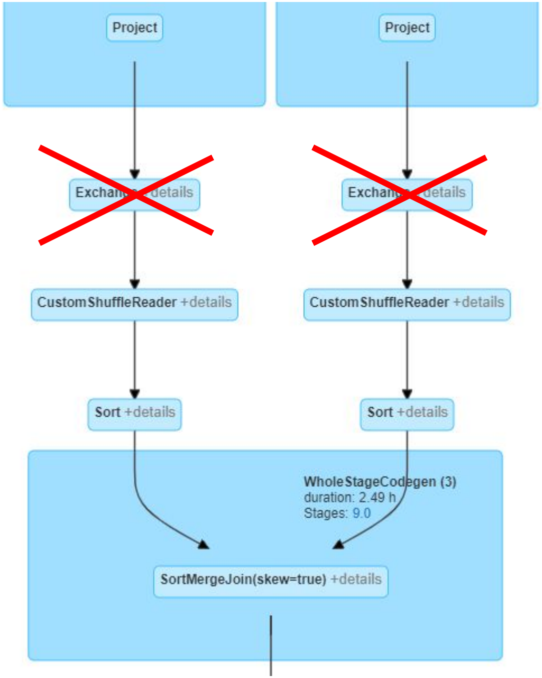
Shuffle Mitigation - Bucketing

Shuffle - Bucketing

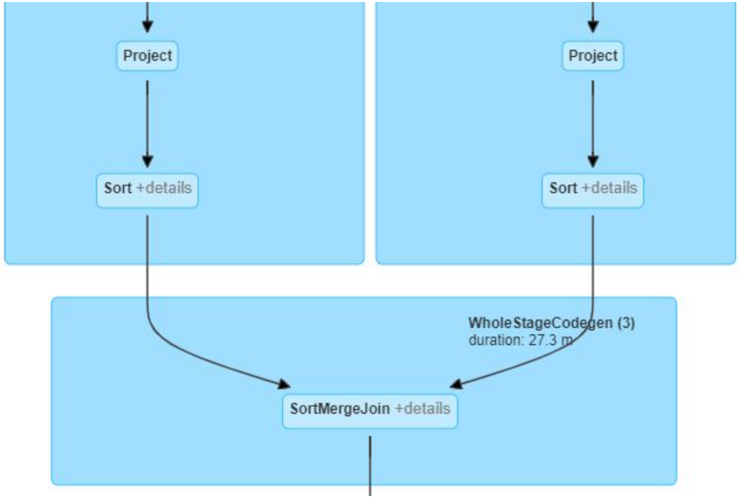
- The goal is to eliminate the exchange & sort by pre-shuffling the data
- The data is aggregated into N buckets and optionally sorted [locally]
- The result is then saved to a table and available for subsequent reads
- The bucketing operation pays for itself if the two tables are regularly joined and/or not reduced with some sort of filter

The 5 Most Common Performance Problems (The 5 Ss)

Shuffle - With & without bucketing



See **Experiment #6167**
and the query for **Step B**



See **Experiment #6167**
and the query for **Step D**

Shuffle - Bucketing Requirements

- To work properly, both tables must have the same number of buckets
- You must predetermine the number of buckets
  - The general rule is one bucket per core
- You must predetermined the, initial Spark-Partition size
  - Upon ingest, one bucket == one spark-partition
  - Overrides **spark.sql.files.maxPartitionBytes**
- The labor to produce & maintain is high... subsequently it must be justifiable
- Bucketing exposes skew - it should be mitigated during production

The 5 Most Common Performance Problems (The 5 Ss)

Shuffle - When to Bucket

When does bucketing make sense?

- With a 100 GB dataset, I can load all data into two 488 GB, 64 core workers
- With only two workers, the cost of shuffling is nearly nonexistent
- The sort needs to be slow
- And the cost of IO between executors needs to be high (e.g. many workers)
- At a 1 to 50 terabyte scales we are already using the largest VMs possible with dozens to scores to hundreds of workers

databricks