DATA+AI SUMMIT 2022

Tuning Apache Spark

Optimizing Apache Spark on Databricks

ORGANIZED BY databricks

DATA+AI SUMMIT 2022

Optimizing Apache Spark

Introductions & Agenda

ORGANIZED BY databricks

Introductions & Agenda
The Course

- This course will focus on some of the most significant performance problems associated with developing Spark applications
- We will learn what those problems are
- We will learn how to identify those problems in existing code
- And we will look at options for mitigating those problems

Introductions & Agenda
Lesson #0.1

Introduction to the Spark Architecture

While knowledge of Spark SQL or the DataFrame APIs is often enough to get started with Apache Spark, every developer needs to have a working knowledge of Apache Spark's main components and how those components interact to execute the simplest of queries. Through a series of real-world analogies, students are walked through various use cases, and from these exercises, develop a robust understanding of how Apache Spark executes some of the most common transformations and actions.

Note: This is optional content used to meet potential prerequisites

Introductions & Agenda

Lesson #0.2

Working with the Spark UI

The Spark UI is the primary tool for debugging both bugs and performance issues within a Spark application. By developing the skills necessary to use this tool and to interpret the data it captures, developers will be better equipped to troubleshoot and tune nearly any issue encountered with a Spark job.

Introductions & Agenda
Lesson #1

The 5 Most Common Performance Problems (The 5 Ss)

The "5 Ss" refers to the five most common performance problems that every developer needs to be aware of: Spill, Skew, Shuffle, Storage, and Serialization. By developing a solid understanding of these problems, every developer is better equipped to diagnose and fix various performance problems.

Introductions & Agenda
Lesson #2

Key Ingestion Concepts

The one optimization applicable to nearly every Spark job is the optimization and reduction of data ingestion. In this course, we explore key ingestion concepts including file formats, data formats, data storage strategies and how they can all work together to maximize a job's performance

Introductions & Agenda
Lesson #3

Optimizing with AQE & DPP

Spark 3.x introduces a series of new strategies around Adaptive Query Execution & Dynamic Partition Pruning that aim to change how data lakes are built and consumed. This course explores six of those strategies and how Apache Spark can expedite dataset development and optimize dataset consumption with more efficient queries.

Introductions & Agenda
Lesson #4

Designing Clusters for High Performance

Proper cluster configuration, VM selection, memory allocation, compute levels, and general topology can play as important of a role in optimizing a job for Apache Spark as can any other topic. In this course, we explore the multitude of factors that go into configuring a cluster.

Introductions & Agenda
The Spark UI Simulator

- https://www.databricks.training/spark-ui-simulator
- Preran notebooks
- A full capture of the notebook, cluster and history server's state
- Experiments are tailored to specific topic
- Experiments are 100% reproducible by students
- Always available for future reference

DATA+AI
SUMMIT 2022

Optimizing Apache Spark

The Five Most Common Performance Problems

ORGANIZED BY databricks

The 5 Most Common Performance Problems
Five Basic Problems (The 5 Ss)

The most egregious problems fall into one of five categories:

- **Spill**: The writing of temp files to disk due to a lack of memory
- **Skew**: An imbalance in the size of partitions
- **Shuffle**: The act of moving data between executors
- **Storage**: A set of problems directly related to how data is stored on disk
- **Serialization**: The distribution of code segments across the cluster

The 5 Most Common Performance Problems
Five Basic Problems (The 5 Ss) - Why it's hard

- Root sourcing problems is hard when one problem can causes another
- **Skew** can induce **Spill**
- **Storage** issues can induce excess **Shuffle**
- Incorrectly addressing **Shuffle** can exacerbate **Skew**
- Many of these problems can be present at the same time
- To better illustrate this problem...
  let's take a quick look at how we benchmark our experiments

Benchmarking

The 5 Most Common Performance Problems (The 5 Ss)
Benchmarking

There are generally three common approaches to benchmarking:

- The **count()** action
- The **foreach()** action with a do-nothing lambda
- A **noop** (or no operation) write

We can see how these three strategies differ with our Spark UI Simulator

The 5 Most Common Performance Problems (The 5 Ss)

Benchmarking – In Action, Part 1

See [Experiment #5980]

- Compare **Step B-1** and **Step B-2**
  - Note the total duration
  - Why did **Step B-1** take 2x longer than **Step B-2**?
- See **Step C**, the **count()** operation
  - Note the duration
  - Note that the Python and Scala samples are nearly identical
  - Note the number of jobs
  - Why is there one less job as compared to **Step B-2**?

The 5 Most Common Performance Problems (The 5 Ss)

Benchmarking - In Action, Part 2

See Experiment #5980

- See **Step D**, the **foreach()** action with a do-nothing lambda
  - Note the total duration (esp compared to the **count()** action)
  - Compare the Scala a Python versions
  - Why is the Python version significantly slower than the Scala version?
- See **Step E**, the **noop** write.
  - Note the total duration of both the Python and Scala

The 5 Most Common Performance Problems (The 5 Ss)
Benchmarking – Review

- About **Step B-1** and **Step B-2**
  - Loading the schema in **Step B-1** and not **Step B-2** provided a side effect
- About the **count()** action
  - Count is optimized – doesn't process all the data
  - Metadata & columnar reads affect execution
- About the **foreach()** action
  - Simulates processing of every record
  - The serialization side effect is quite significant in Python
- About the **noop** with a schema – it just works as expected!

DATA+AI
SUMMIT 2022

Optimizing Apache Spark

The Five Most Common Performance Problems: Skew

Add your Name

Add your title, company

ORGANIZED BY databricks

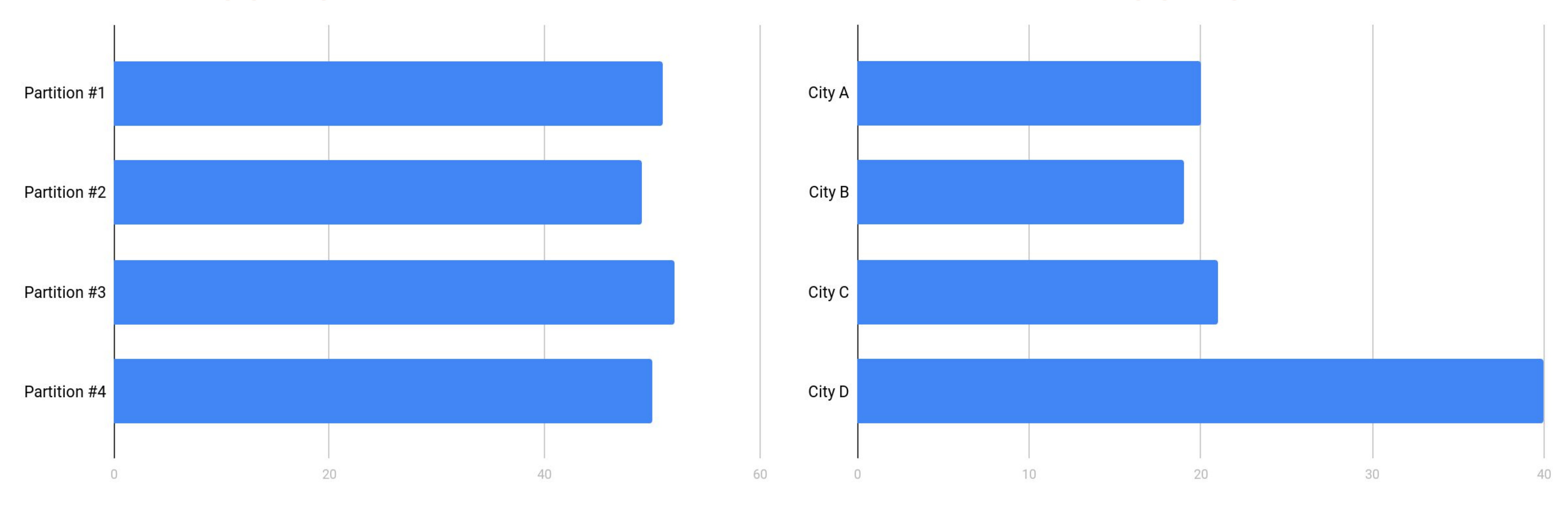
The 5 Most Common Performance Problems (The 5 Ss)
Skew

- Data is typically read in as 128 MB partitions and evenly distributed ...*more on maxPartitionBytes later*
- As the data is transformed (e.g. aggregated), it's possible to have significantly more records in one partition than another
- A small amount of skew is ignorable
- But large skews can result in spill or worse, hard to diagnose OOM Errors

The 5 Most Common Performance Problems (The 5 Ss)
Skew – Before & After

Before aggregation

After aggregation by city

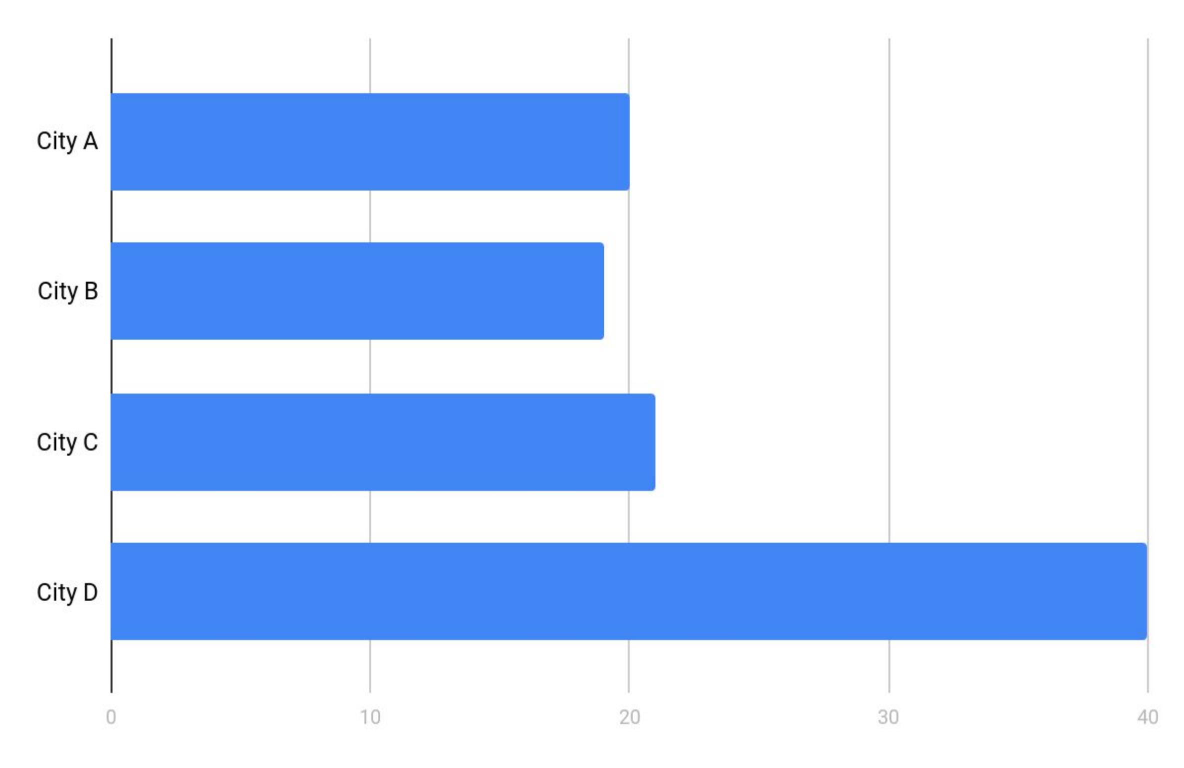


The 5 Most Common Performance Problems (The 5 Ss)
Skew – Ramifications

If **City D** is 2x larger than A, B or C...
- It takes 2x as long to process
- It requires 2x as much RAM

The ramifications of that is...
- The entire stage will take as long as the longest running task
- We may not have enough RAM for these skewed partitions

After aggregation by city



How can we mitigate skew?

The 5 Most Common Performance Problems (The 5 Ss)
Skew - Time vs RAM

We need to first ask which problem are we solving for?

- Solving for the RAM problem is only treating the symptoms and not the root cause.
- The RAM problem manifests itself as **Spill** and/or OOM Errors and should **not** be the first thing we solve for…*more on spill later*
- The first problem to solve for is the uneven distribution of records across all partitions which manifests itself as proportionally slower tasks

The 5 Most Common Performance Problems (The 5 Ss)
Skew - Mitigation

There are several strategies for fixing skew:

- Employ a Databricks-specific skew hint
  (see Skew Join optimization)
- Enable Adaptive Query Execution in Spark 3
  - *...more on AQE and Spark 3 later*
- Salt the skewed column with a random number creating better distribution across each partition at the cost of extra processing

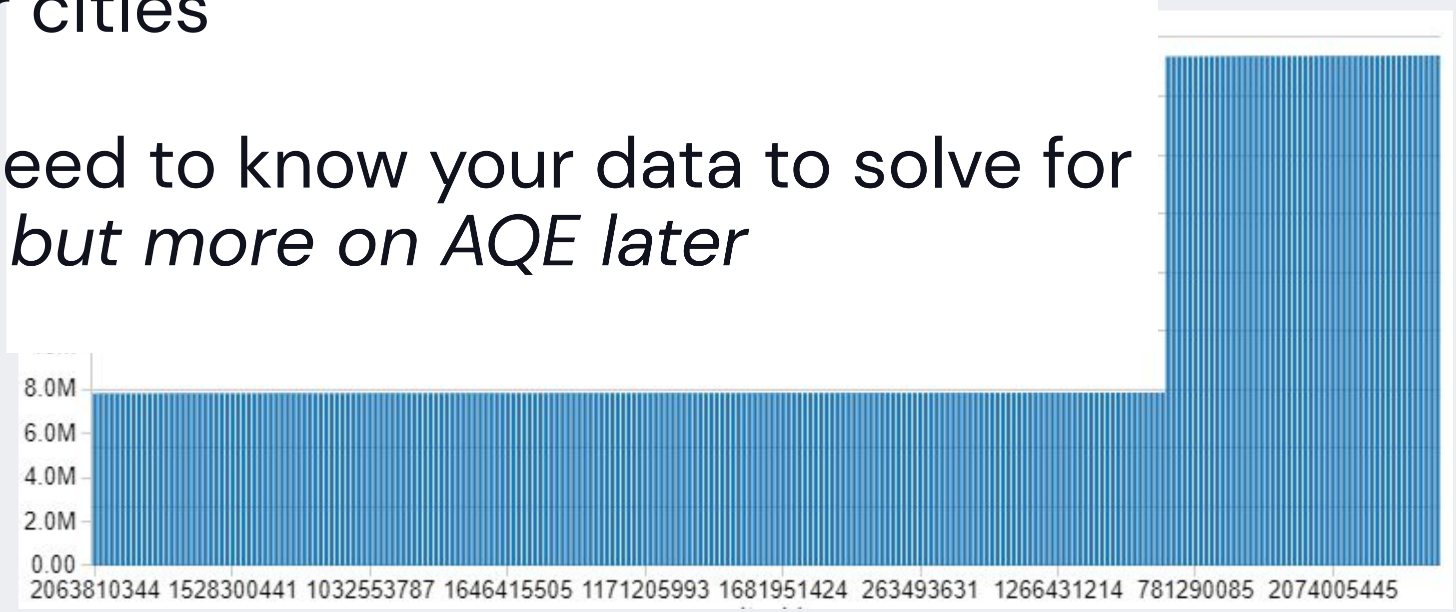
The 5 Most Common Performance Problems (The 5 Ss)
Skew – How Skewed?

See Experiment #1596, **Step B**

- Our perfectly engineered data has a skew in US cities that is ~3x larger than all other countries

- Counts come in at 23 million for skewed cities vs 8 million for other cities

- As we will see, you really need to know your data to solve for this…*maybe not with AQE, but more on AQE later*



DATA+AI SUMMIT 2022

The 5 Most Common Performance Problems (The 5 Ss)

Skew – Baseline vs Hint

See Experiment #1596, **Step C** and **Step D**

- Contrast the **last stage** of the **last job** for the two commands
  - Note the key code differences
  - Note the total execution time of the corresponding commands
  - Note the total number of tasks
  - In the **Spark UI**, **Stage Details**
    - Note the "health" of the stage as seen in the **Event Timeline**
    - Note the min/median/max **Shuffle Read Size** under **Summary Metrics**
    - Note the total amount of spill under **Aggregated Metrics by Executor**

The 5 Most Common Performance Problems (The 5 Ss)

Skew – Baseline vs Hint, Review

Step	Code	Duration	Tasks	Health	Shuffle	Spill
C	Standard	~30 min	832	Bad	0 / 0 / ~100 KB / ~400 MB / ~3 GB	~50 GB
D	Skew Hint	~35 min	832	Mostly OK	134 MB / 174 MB / 184 MB / 195 MB / 1.1 GB	~4 GB

- This scenario introduces the Databricks-specific skew hint (see Skew Join optimization)
- Note the call **.hint("skew", "city_id")**

The 5 Most Common Performance Problems (The 5 Ss)

Skew - Baseline vs Hint vs w/AQE

See Experiment #1596, **Step E** with **Step C** and **Step D**

- Contrast the **last stage** of the **last job** for the two commands
  - Note the key code differences
  - Note the total execution time of the corresponding commands
  - Note the total number of tasks
  - In the **Spark UI**, **Stage Details**
    - Note the "health" of the stage as seen in the **Event Timeline**
    - Note the min/median/max **Shuffle Read Size** under **Summary Metrics**
    - Note the total amount of spill under **Aggregated Metrics by Executor**

The 5 Most Common Performance Problems (The 5 Ss)
Skew – Baseline vs Hint, Review

Step	Code	Duration	Tasks	Health	Shuffle	Spill
C	Standard	~30 min	832	Bad	0 / 0 / ~100 KB / ~400 MB / ~3 GB	~50 GB
D	Skew Hint	~35 min	832	Mostly OK	~135 MB / ~175 MB / ~180 MB / ~200 MB / ~1 GB	~4 GB
E	w/AQE	~25 min	1489	Excellent	0 / ~115 MB / ~115 MB / ~125 MB / ~130 MB	0

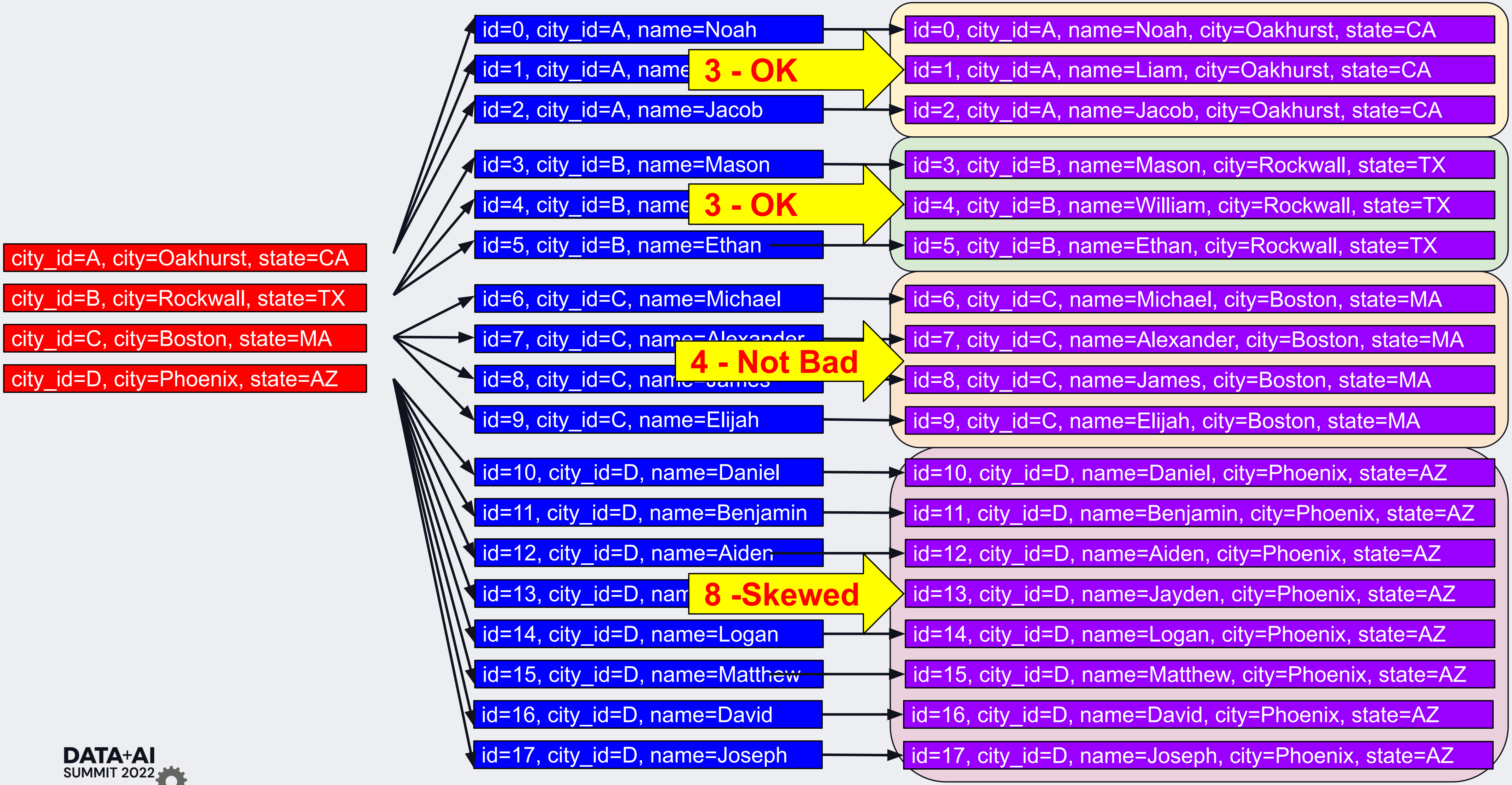
- **Step E** uses **Spark 3's** new feature **Adaptive Skewed Join**
  - See **spark.sql.adaptive.skewJoin.enabled**
  - See **spark.sql.adaptive.advisoryPartitionSizeInBytes**
- The first two **jobs** are read in parallel

The 5 Most Common Performance Problems (The 5 Ss)
Skew - Salted Join

- This approach is by far the most complicated to implement
- It can actually take longer to execute in some cases
- Remains a viable option when other solutions are not available
- The idea is to split large partitions into smaller ones using a “salt”
- Has the side effect of splitting small partitions into even smaller ones
- It’s more about guaranteeing execution of all tasks
  And **not** a uniform duration for each task

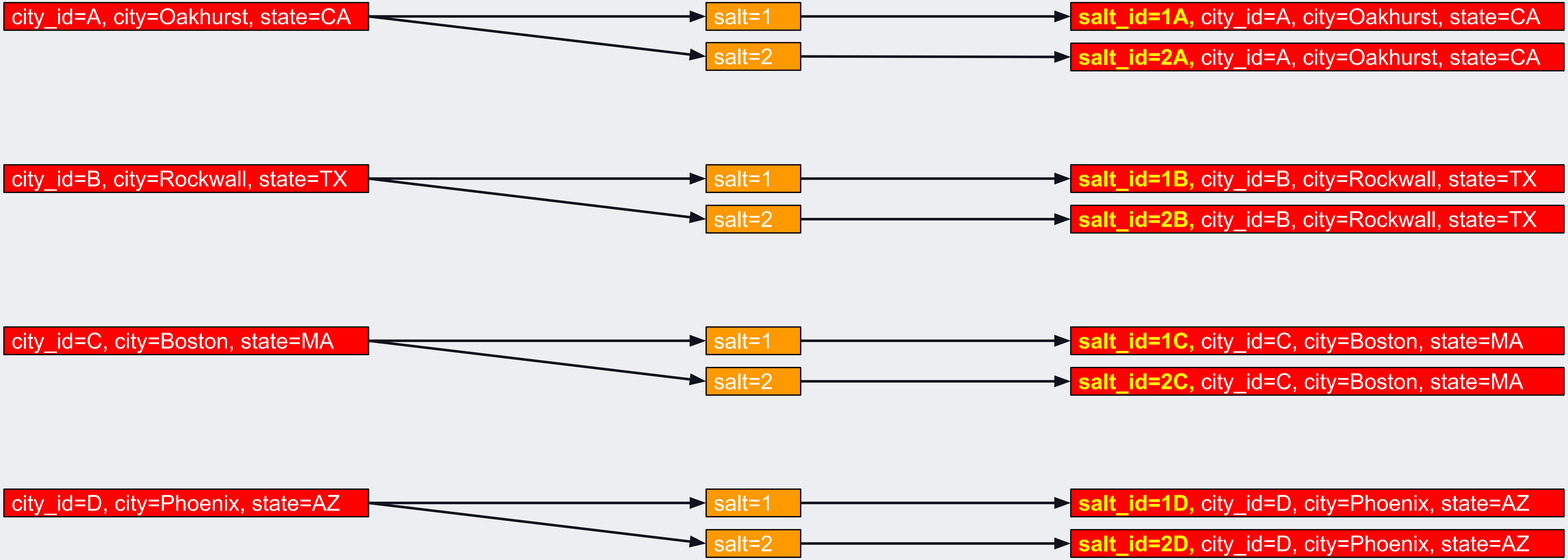
Let’s review how a “standard” join works…

…4 distinct partitions



Let’s review how a “salted” join works…

Step #1: Cross join the dimensions to the salt value

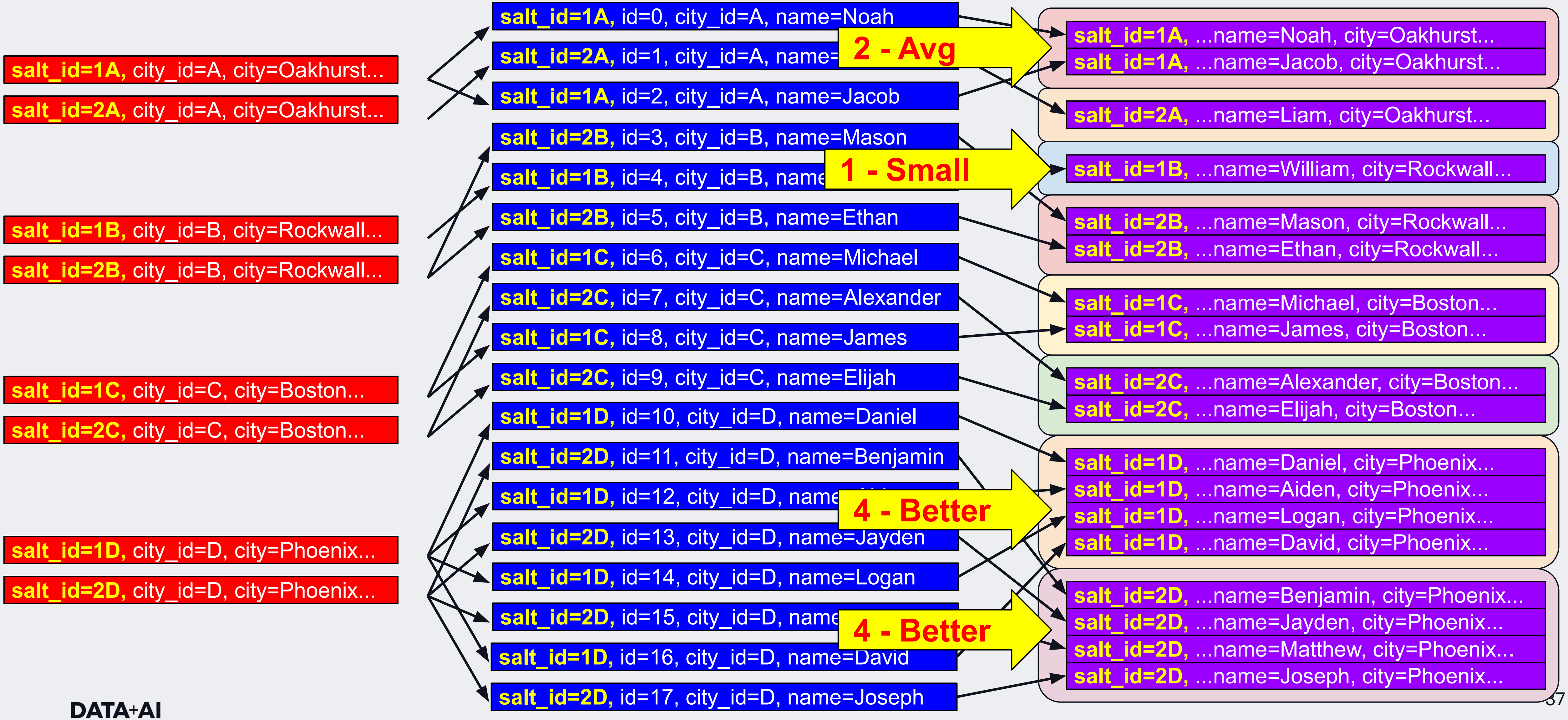


Step #2: Randomly salt the fact table

id=0, city_id=A, name=Noah	salt=rand(2)	1	**salt_id=1A,** id=0, city_id=A, name=Noah
id=1, city_id=A, name=Liam	salt=rand(2)	2	**salt_id=2A,** id=1, city_id=A, name=Liam
id=2, city_id=A, name=Jacob	salt=rand(2)	1	**salt_id=1A,** id=2, city_id=A, name=Jacob
id=3, city_id=B, name=Mason	salt=rand(2)	2	**salt_id=2B,** id=3, city_id=B, name=Mason
id=4, city_id=B, name=William	salt=rand(2)	1	**salt_id=1B,** id=4, city_id=B, name=William
id=5, city_id=B, name=Ethan	salt=rand(2)	2	**salt_id=2B,** id=5, city_id=B, name=Ethan
id=6, city_id=C, name=Michael	salt=rand(2)	1	**salt_id=1C,** id=6, city_id=C, name=Michael
id=7, city_id=C, name=Alexander	salt=rand(2)	2	**salt_id=2C,** id=7, city_id=C, name=Alexander
id=8, city_id=C, name=James	salt=rand(2)	1	**salt_id=1C,** id=8, city_id=C, name=James
id=9, city_id=C, name=Elijah	salt=rand(2)	2	**salt_id=2C,** id=9, city_id=C, name=Elijah
id=10, city_id=D, name=Daniel	salt=rand(2)	1	**salt_id=1D,** id=10, city_id=D, name=Daniel
id=11, city_id=D, name=Benjamin	salt=rand(2)	2	**salt_id=2D,** id=11, city_id=D, name=Benjamin
id=12, city_id=D, name=Aiden	salt=rand(2)	1	**salt_id=1D,** id=12, city_id=D, name=Aiden
id=13, city_id=D, name=Jayden	salt=rand(2)	2	**salt_id=2D,** id=13, city_id=D, name=Jayden
id=14, city_id=D, name=Logan	salt=rand(2)	1	**salt_id=1D,** id=14, city_id=D, name=Logan
id=15, city_id=D, name=Matthew	salt=rand(2)	2	**salt_id=2D,** id=15, city_id=D, name=Matthew
id=16, city_id=D, name=David	salt=rand(2)	1	**salt_id=1D,** id=16, city_id=D, name=David
id=17, city_id=D, name=Joseph	salt=rand(2)	2	**salt_id=2D,** id=17, city_id=D, name=Joseph

Step #3: Join the salted tables

… 8 distinct partitions



The 5 Most Common Performance Problems (The 5 Ss)
Skew - Skew Join, in Action

- Step F-1: Create a DataFrame based on the range of our "skew factor"
  - In the visual example, we used "2"
  - In this code example, we are using "7"
  - You can estimate this based on how many times larger the maximum partition is compared to the median partition size

- Step F-2: For the dimension table, cross join the salts with the city table (repartitioning can help mitigate spills and evenly redistributes the new dimension table across all partitions)

- Step F-3: For the fact table, randomly assign a salt to each record

- Step F-4: Join the two tables based on the **salted_city_id**

The 5 Most Common Performance Problems (The 5 Ss)

Skew – Baseline vs Hint vs w/AQE vs Salted

See Experiment #1596, **Step F-4** with **Step C** through **Step E**

- Contrast the **last stage** of the **last job** for the two commands
  - Note the key code differences
  - Note the total execution time of the corresponding commands
  - Note the total number of tasks
  - In the **Spark UI**, **Stage Details**
    - Note the “health” of the stage as seen in the **Event Timeline**
    - Note the min/median/max **Shuffle Read Size** under **Summary Metrics**
    - Note the total amount of spill under **Aggregated Metrics by Executor**

The 5 Most Common Performance Problems (The 5 Ss)
Skew – Baseline vs Hint, Review

Step	Code	Duration	Tasks	Health	Shuffle	Spill
C	Standard	~30 min	832	Bad	0 / 0 / ~100 KB / ~400 MB / ~3 GB	~50 GB
D	Skew Hint	~35 min	832	Mostly OK	~135 MB / ~175 MB / ~180 MB / ~200 MB / ~1 GB	~4 GB
E	w/AQE	~25 min	1489	Excellent	0 / ~115 MB / ~115 MB / ~125 MB / ~130 MB	0
F	Salted	>35 min	832	Better/Bad	~400 KB / ~75 MB / ~150 MB / ~300 MB / ~800 MB	0

- Salting a skewed dataset has a number of problems
- You don’t want to salt on the fly – it should be a persisted view of the data
- Consider instead, invest the energy to salt only the skewed keys
- In our example, that would mean salting US cities only

DATA+AI
SUMMIT 2022

Optimizing Apache Spark

The Five Most Common Performance Problems: Spill

Add your Name
Add your title, company

The 5 Most Common Performance Problems (The 5 Ss)
Spill

- Spill is the term used to refer to the act of moving an RDD from RAM to disk, and later back into RAM again
- This occurs when a given partition is simply too large to fit into RAM
- In this case, Spark is forced into [potentially] expensive disk reads and writes to free up local RAM
- All of this just to avoid the dreaded OOM Error

The 5 Most Common Performance Problems (The 5 Ss)
Spill - Examples

There are a number of ways to induce this problem:

- Set **spark.sql.files.maxPartitionBytes** to high (default is 128 MB)
  ...more on maxPartitionBytes later
- The **explode()** of even a small array
- The **join()** or **crossJoin()** of two tables
- Aggregating results by a skewed feature

The 5 Most Common Performance Problems (The 5 Ss)
Spill – Memory & Disk

In the Spark UI, spill is represented by two values:

- **Spill (Memory):** For the partition that was spilled, this is the size of that data as it existed in memory
- **Spill (Disk):** Likewise, for the partition that was spilled, this is the size of the data as it existed on disk

The two values are always presented together

The size on disk will always be smaller due to the natural compression gained in the act of serializing that data before writing it to disk

The 5 Most Common Performance Problems (The 5 Ss)
Spill – In the Spark UI

A couple of notes:

- Spill is only represented in the details page for a single stage…
  - **Summary Metrics**
  - **Aggregated Metrics by Executor**
  - The **Tasks** table

- Or in the corresponding query details

- This makes it hard to recognize because one has to hunt for it

- When no spill is present, the corresponding columns don't even appear in the Spark UI – that means if the column is there, there is spill somewhere

The 5 Most Common Performance Problems (The 5 Ss)
Spill - Spill Listener

See Experiment #6518, **Step A-2**

- The **SpillListener** is taken from Apache Spark's test framework
- The **SpillListener** is a type of **SparkListener** and tracks when a stage spills
- Useful to identify spill in a job when you are not looking for it
- We can see example usage in **Step B** through **Step E**

The 5 Most Common Performance Problems (The 5 Ss)
Spill – Examples

See Experiment #6518

- Note the four examples:
  - Step B: Spill induced by ingesting large partitions
  - Step C: Spill induced by unioning tables
  - Step D: Spill induced with explode operations
  - Step E: Spill induced by a skewed join
- For each example…
  - Find and note the total **Spill (Memory)** and **Spill (Disk)**
  - Find and note the min, median and max **Spill (Memory)** and **Spill (Disk)**
- Which of the four examples is uniquely different in how it manifests spill?

The 5 Most Common Performance Problems (The 5 Ss)
Spill – Examples, Review

Step	Min	25th	Median	75th	Max	Total
B – shuffle	~2 GB / ~550 MB	~2 GB / ~560 MB	~2 GB / ~565 MB	~2 GB / ~570 MB	~2 GB / ~580 MB	~33 GB
C – union	~2 GB / ~110 MB	~2 GB / ~120 MB	~2 GB / ~125 MB	~2 GB / ~130 MB	~2 GB / ~150 MB	~60 GB
D – explode	0 / ~1.5 GB	0 / ~1.5 GB	0 / ~1.5 GB	0 / ~1.5 GB	0 / ~1.5 GB	~750 GB
E – join*	0 / 0	0 / 0	0 / 0	0 / 0	6 GB / 3 GB	~50 GB

- In **Step B**, the config value **spark.sql.shuffle.partitions** is not managed
- **Steps C & D** simply grow too large as a result of their transformations
- In **Step E** the spill is a manifestation of the underlying skew

What can we do to mitigate spill?

The 5 Most Common Performance Problems (The 5 Ss)
Spill - Mitigation

- The quick answer: allocate a cluster with more memory per worker
  ...more on cluster configurations later

- In the case of skew, address that root cause first

- Decrease the size of each partition by increasing the number of partitions
  - By managing **spark.sql.shuffle.partitions**
  - By explicitly **repartitioning**
  - By managing **spark.sql.files.maxPartitionBytes**
    ...more on maxPartitionBytes later
  - Not an effective strategy against skew

The 5 Most Common Performance Problems (The 5 Ss)
Spill – Mitigation

- Ignore it – consider the example in **Step E**.
  - Out of ~800 tasks only ~50 tasks spilled
  - Is that 6% worth your time?

- However, it takes only one long task to delay an entire stage

DATA+AI SUMMIT 2022

Optimizing Apache Spark

The Five Most Common Performance Problems: Shuffle

ORGANIZED BY databricks

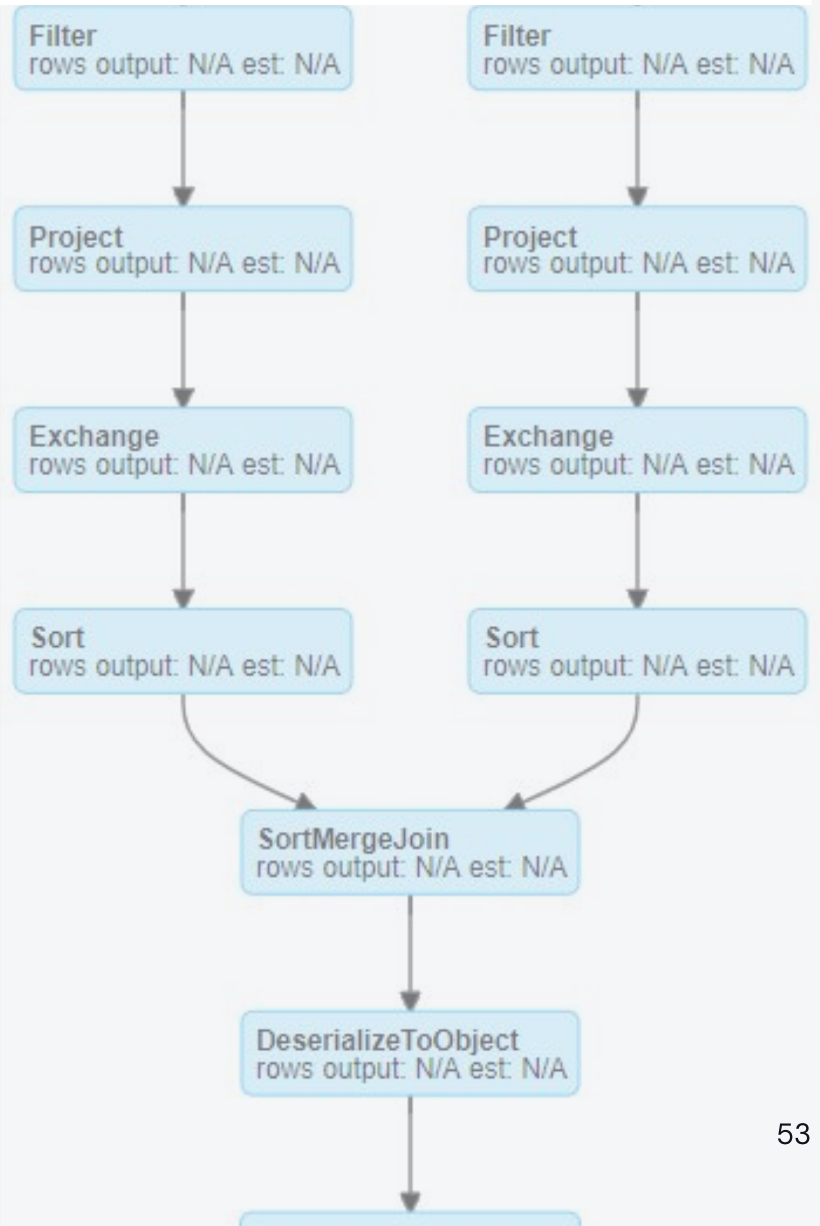
The 5 Most Common Performance Problems (The 5 Ss)

Shuffle

Shuffling is a side effect of wide transformation:

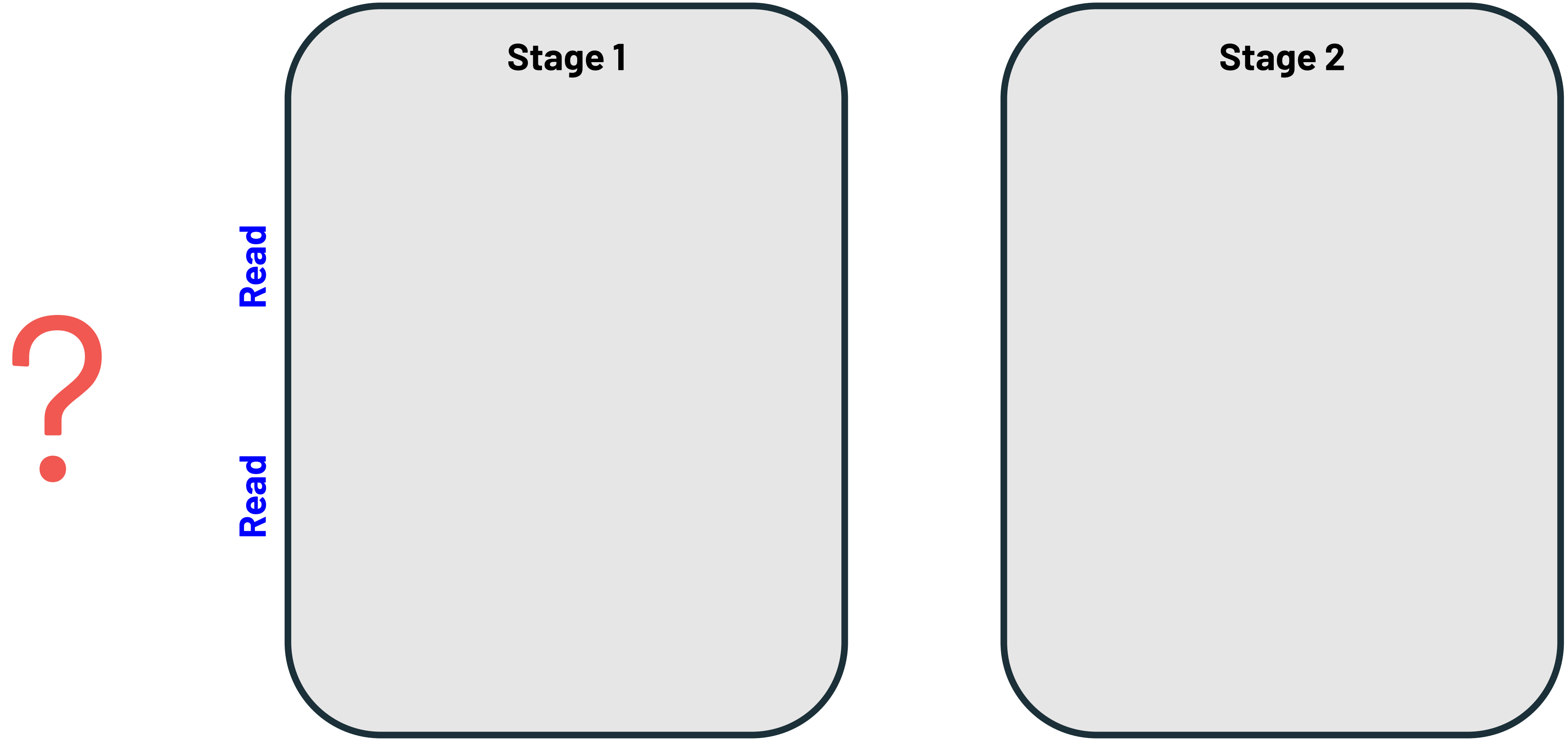
- `join()`
- `distinct()`
- `groupBy()`
- `orderBy()`

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



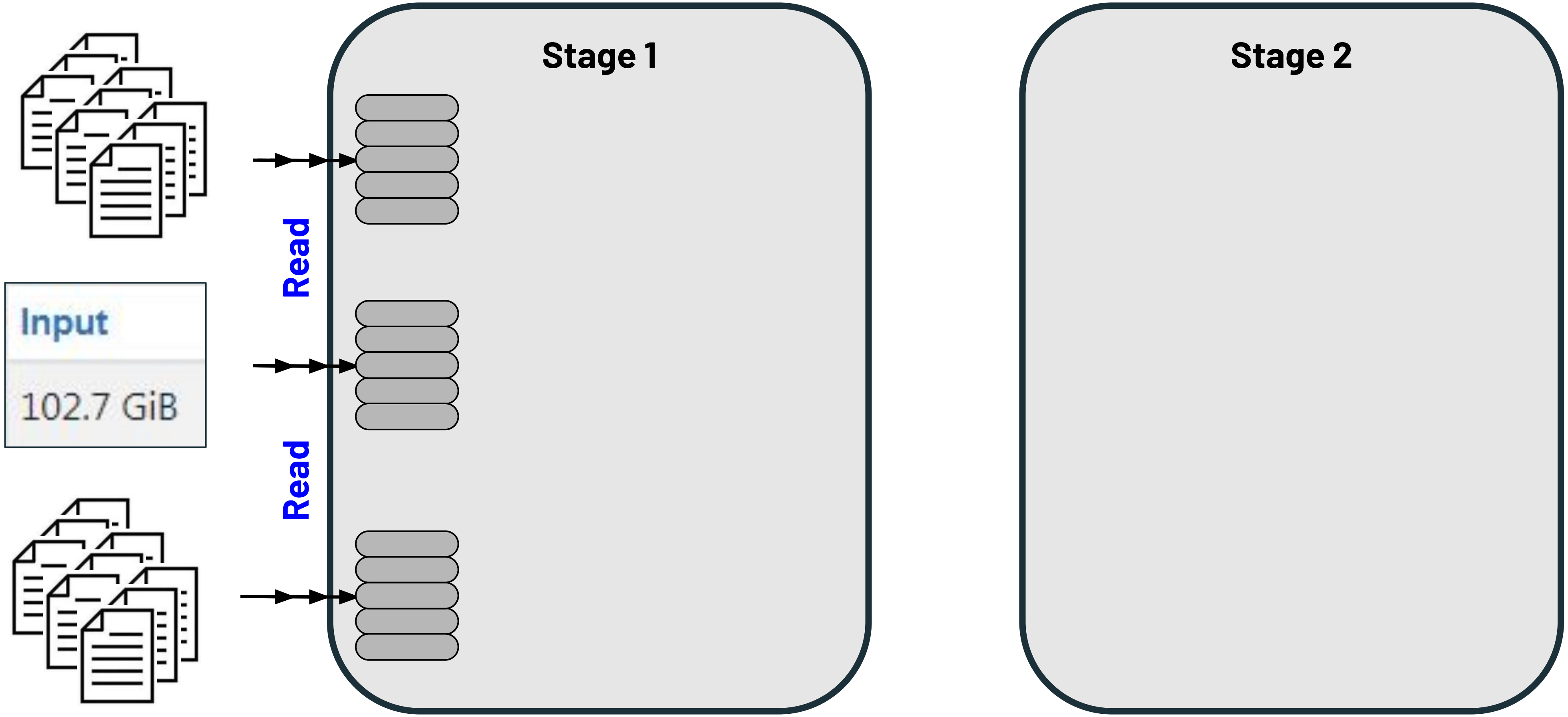
Let’s take a look at how a shuffle works…

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



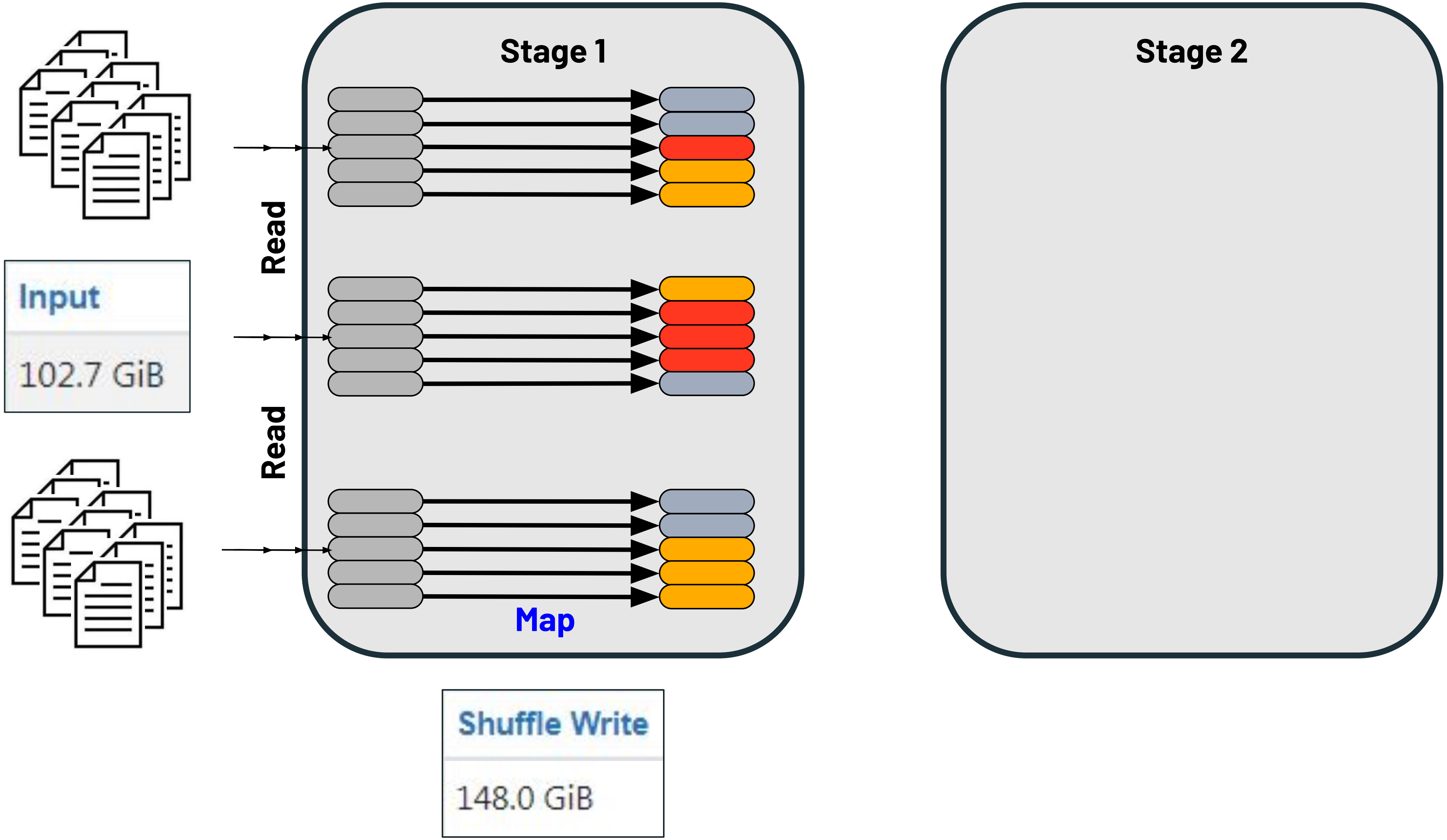

```
spark.read.parquet(eventsPath).groupBy("state")
      .avg("revenue")
      .write.mode("overwrite").csv("/tmp/")
```

Step #2: Read the data into Spark-Partitions

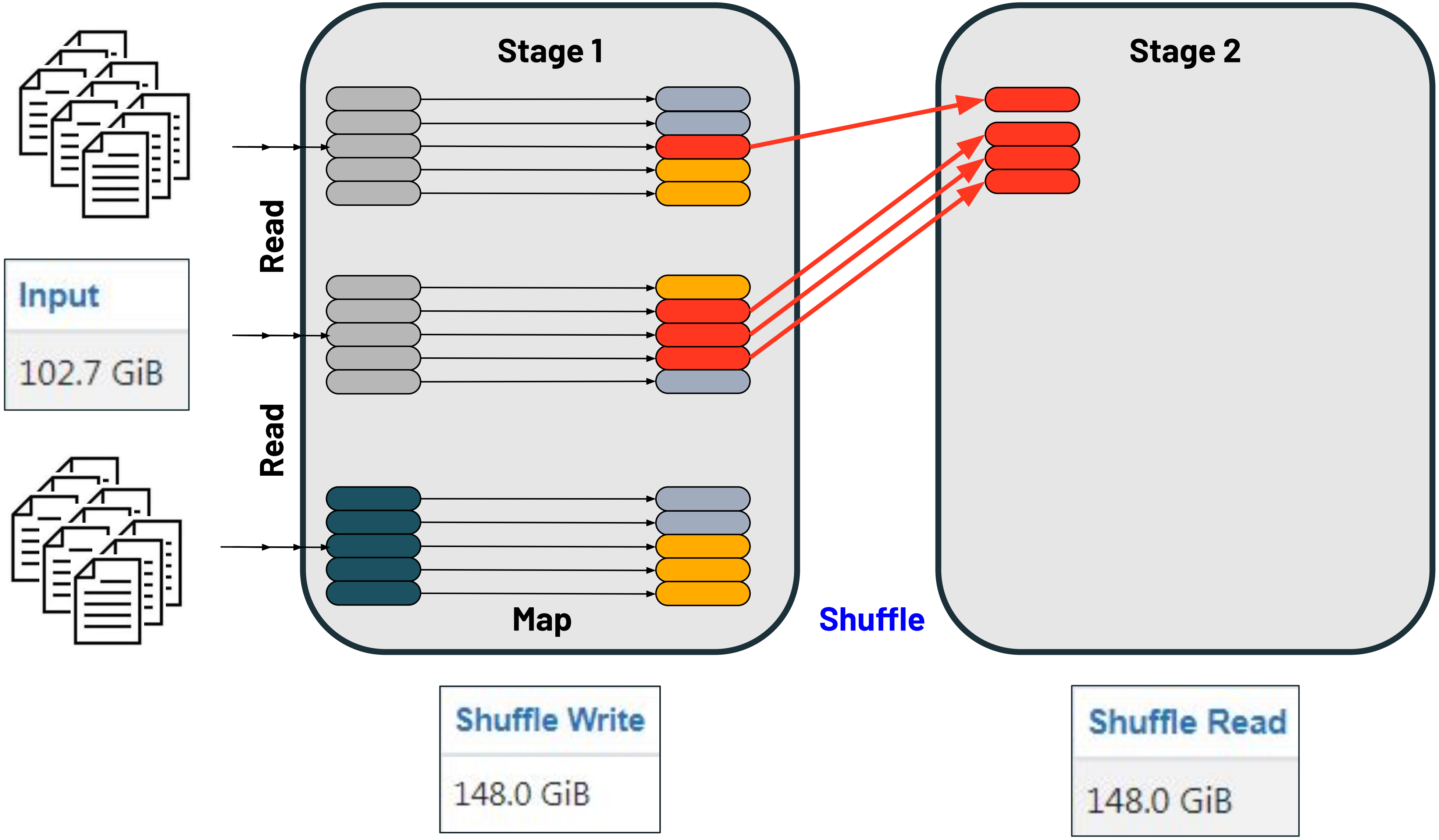



```
spark.read.parquet(eventsPath).groupBy("state")
      .avg("revenue")
      .write.mode("overwrite").csv("/tmp/")
```

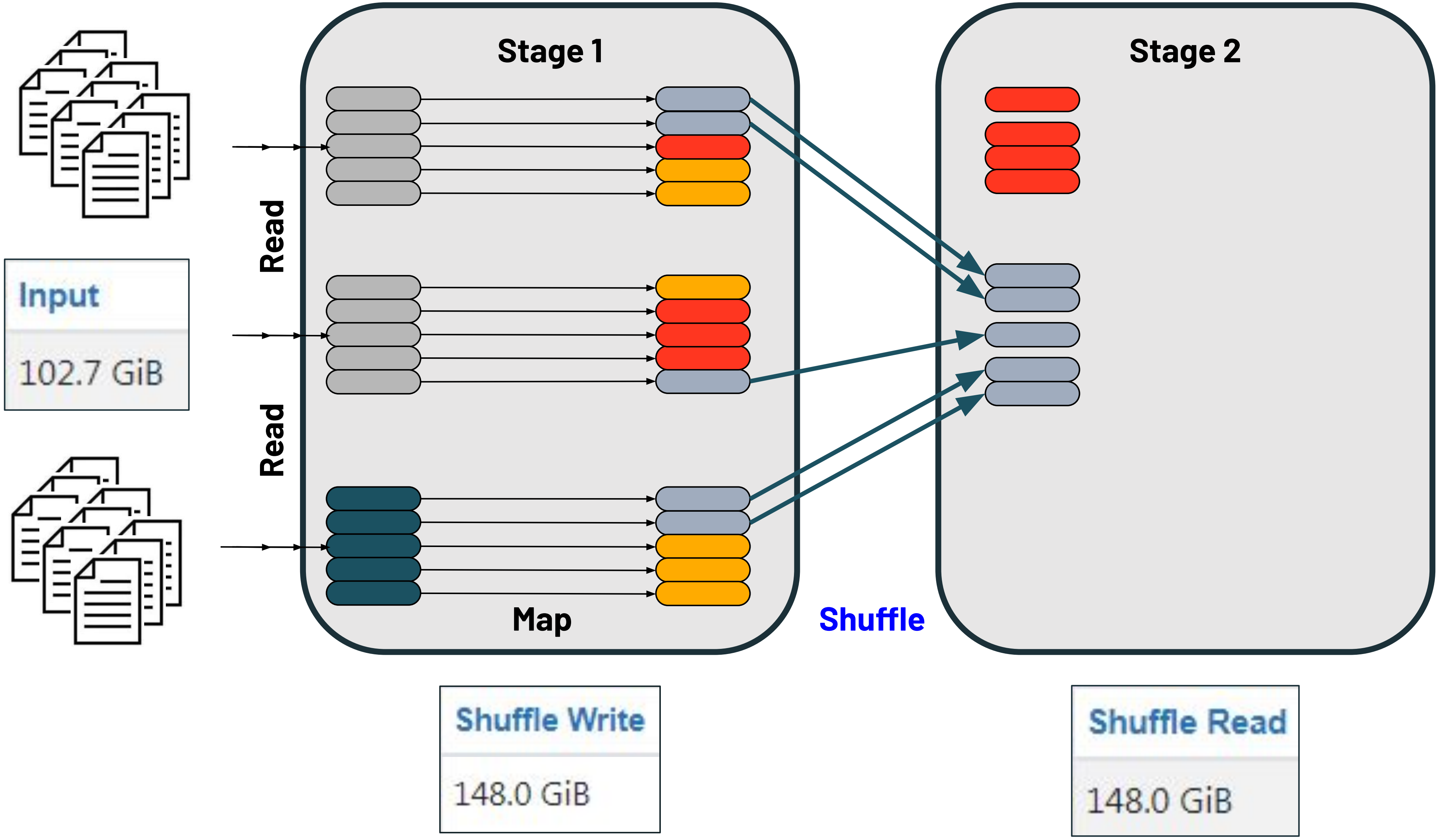
Step #3: Map reach record (e.g. by key) to a new Partition (and write files in Stage 1 prior to Shuffle to Stage 2)



Step #4-A: Read the Shuffle files into the Stage 2

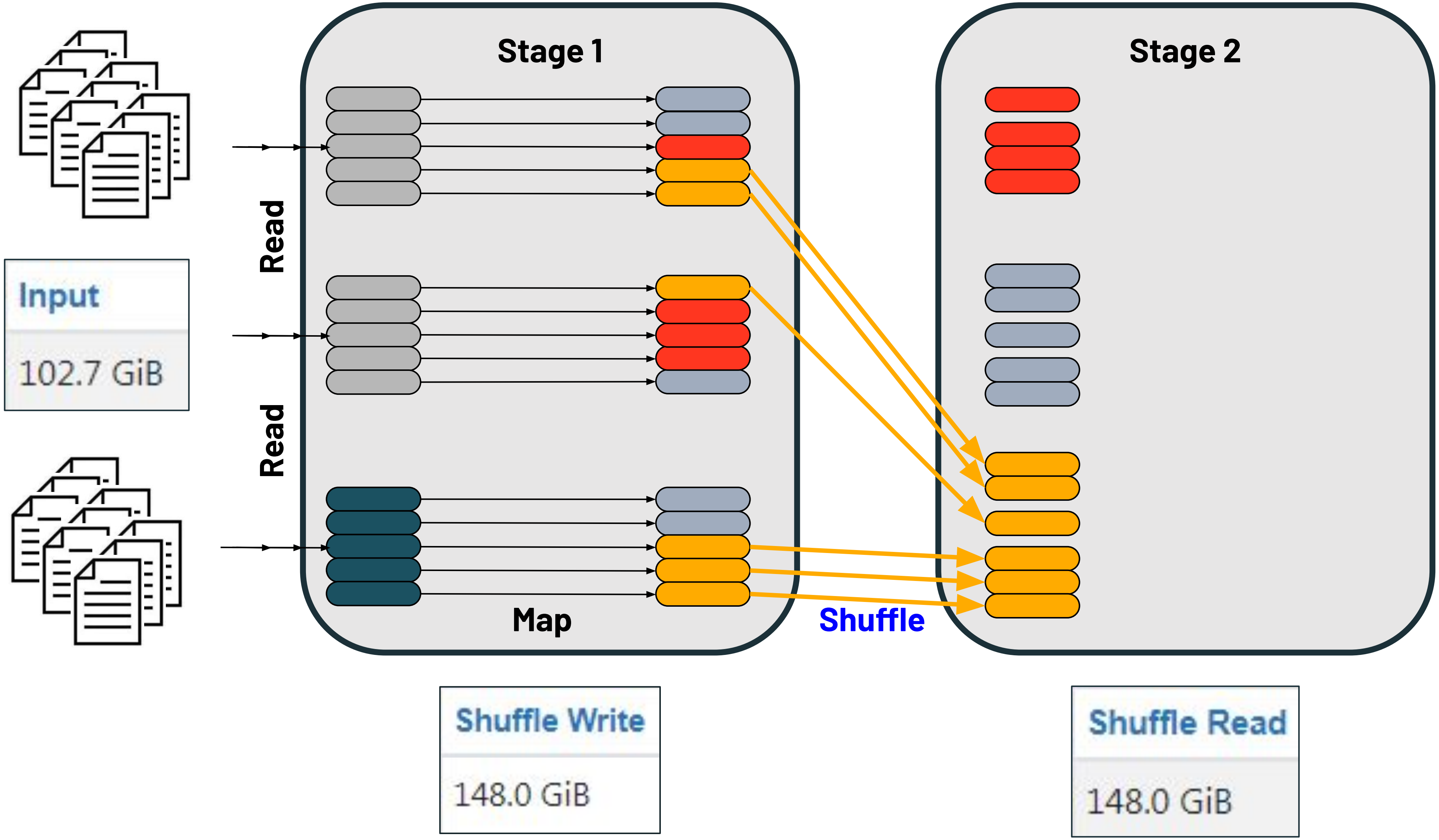


Step #4-B: Stage-1 would have written the Shuffle files

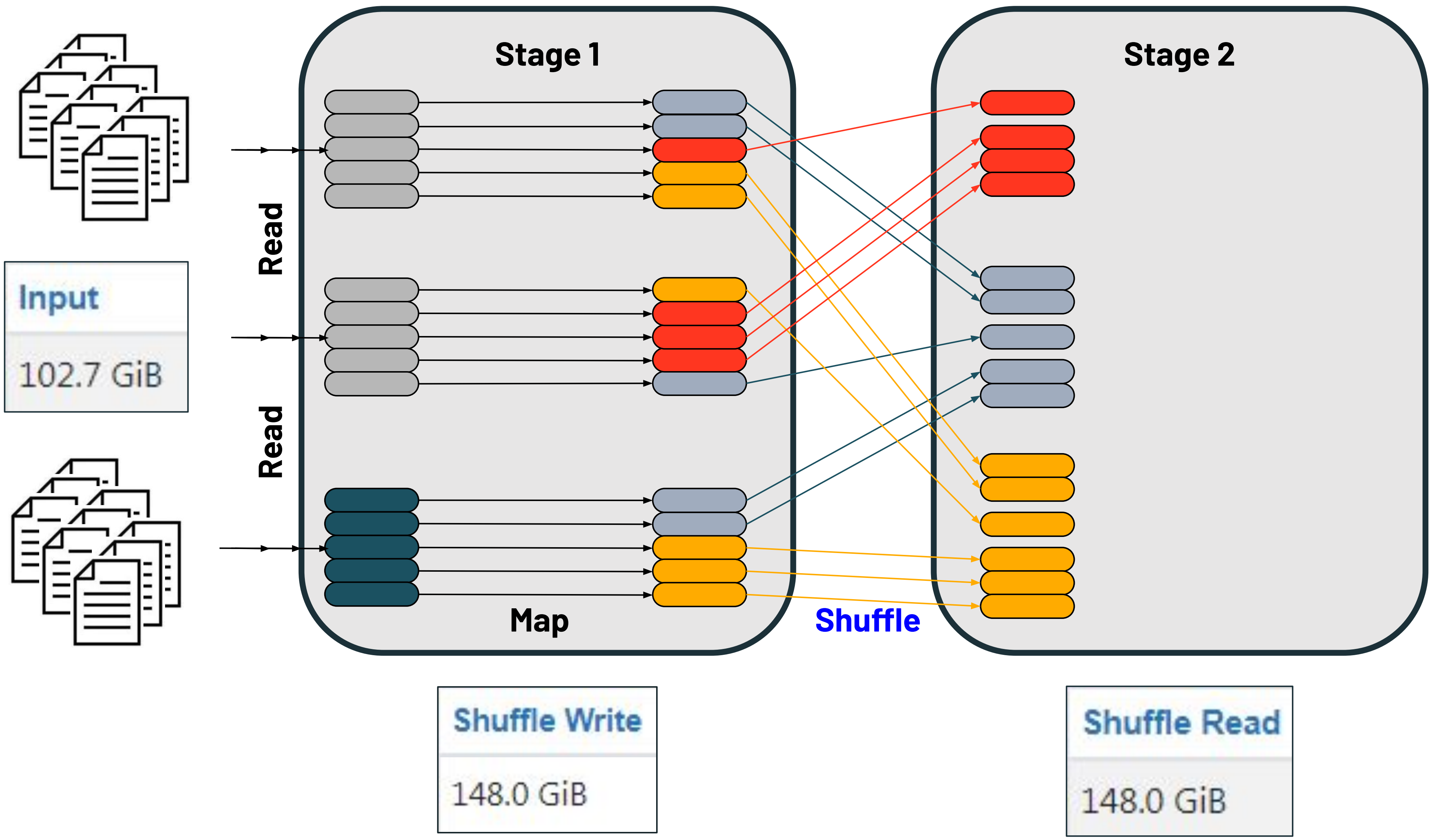


databricks

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

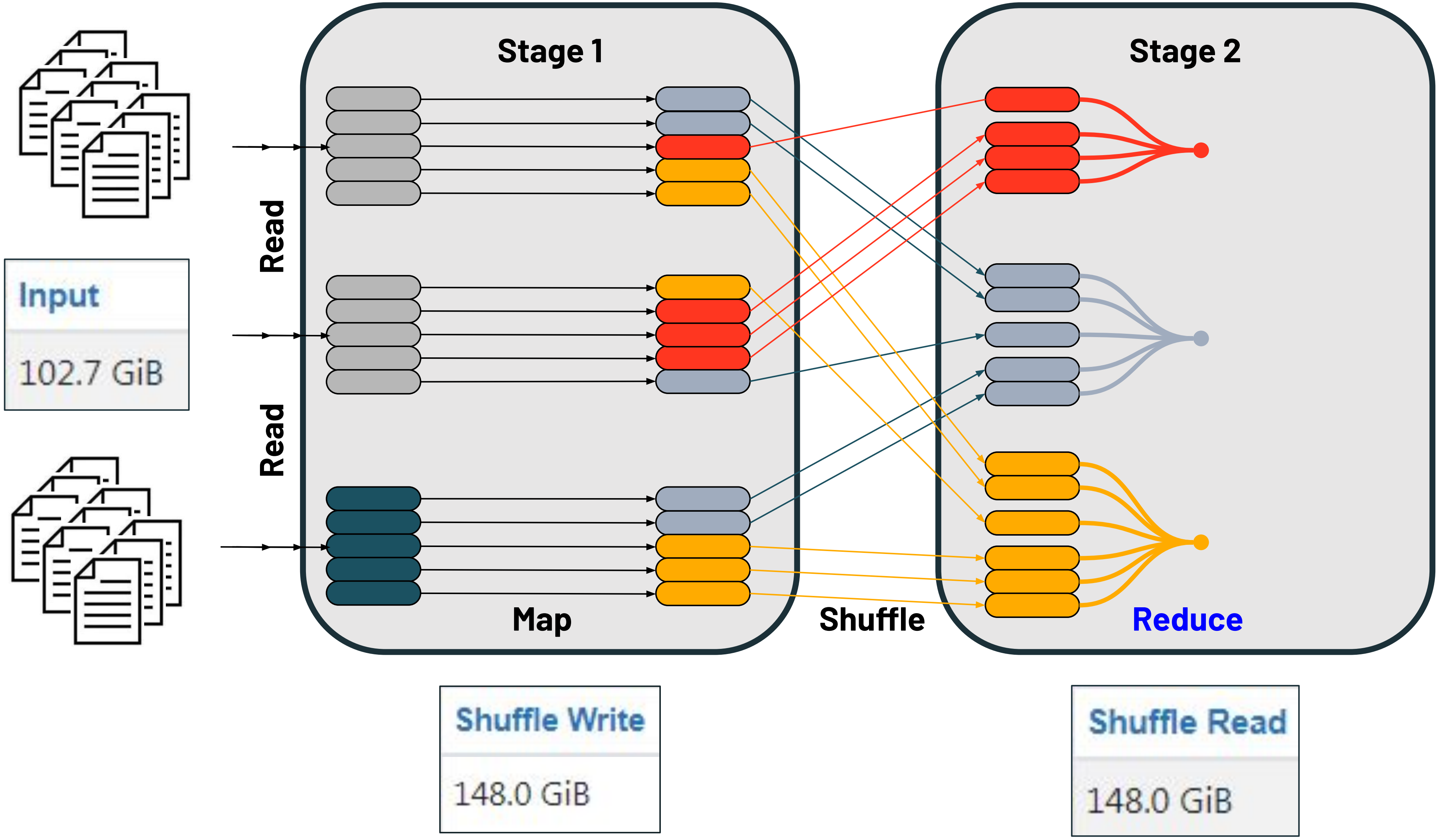


Step #4-D: Done simultaneously, this is a blocking operation

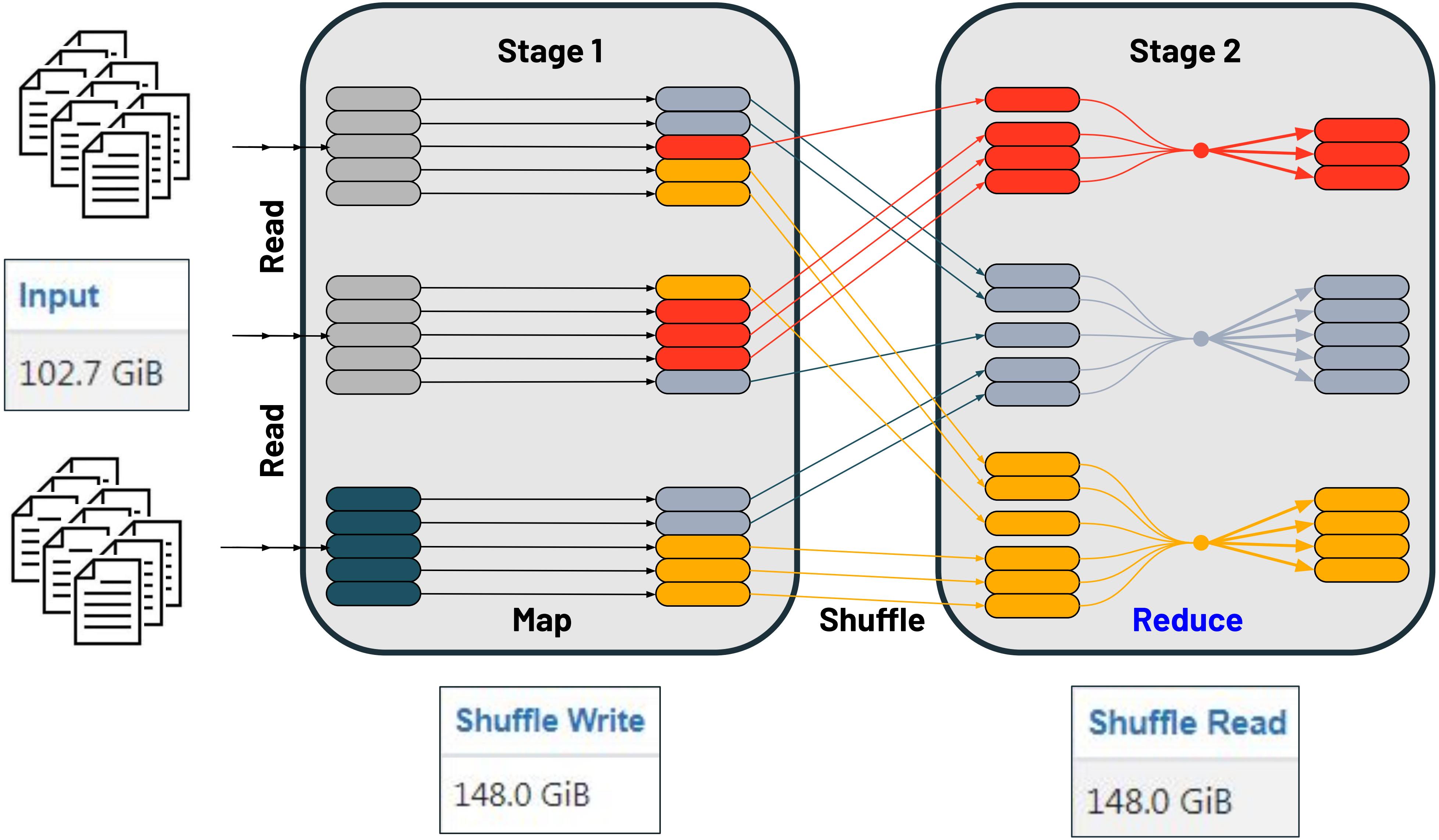


databricks

Step #5: The Partitions are 'reduced', how varies (Aggr, join,etc)

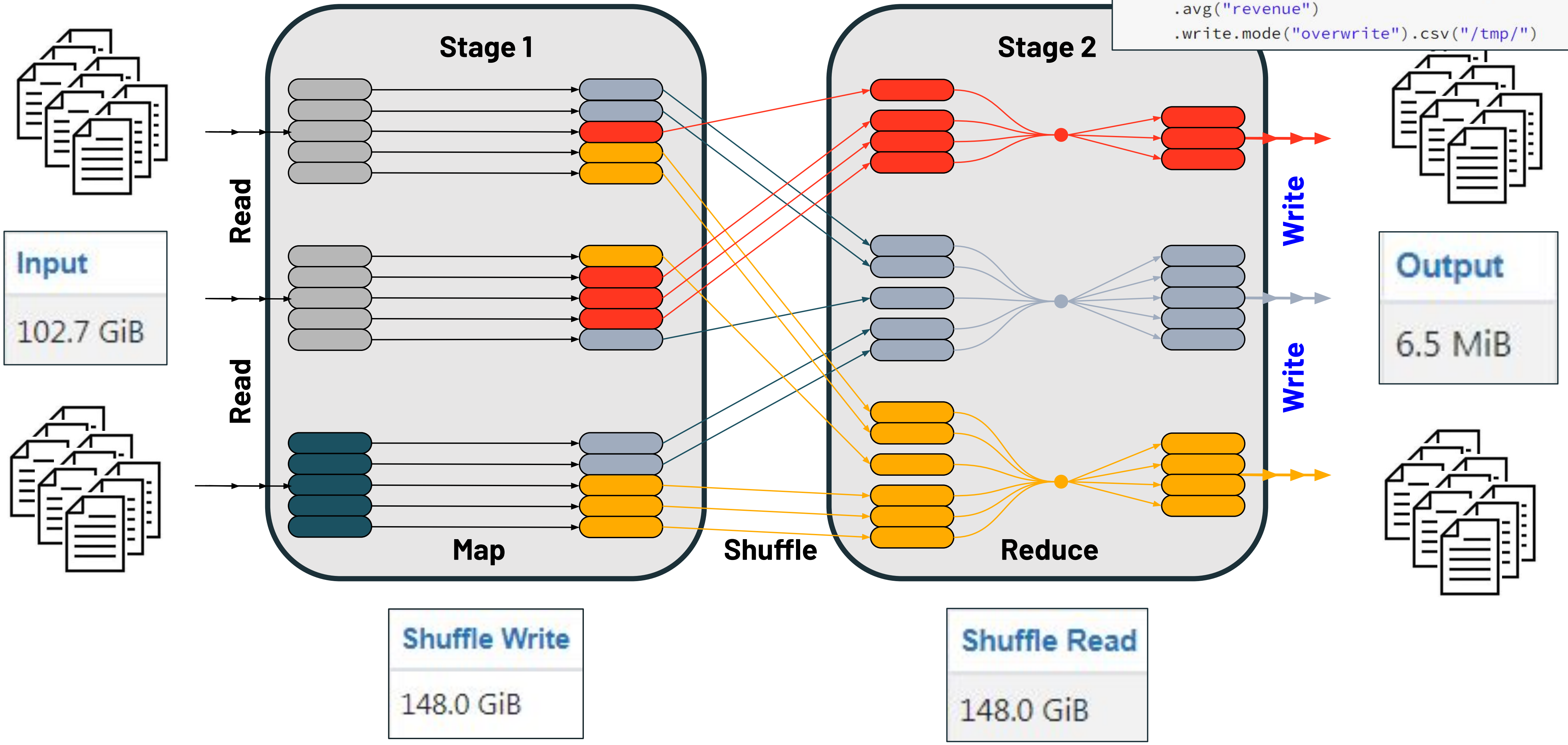


Step #6: Final result is a new set of Partitions



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

```
spark.read.parquet(eventsPath).groupBy("state")
      .avg("revenue")
      .write.mode("overwrite").csv("/tmp/")
```



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
  ... *more on optimizing cluster designs later*

- Reduce the amount of data being shuffled
  - Narrow your columns
  - Preemptively filter out unnecessary records
  - ... *more on optimizing data ingestion later*

- 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 …*more on this later*
  - See our presentation (The Apache Spark™ Cost-Based Optimizer) at https://youtu.be/WSIN6f-wHcQ

DATA+AI
SUMMIT 2022

Optimizing Apache Spark

The Five Most Common Performance Problems: Shuffle Mitigation - BroadcastHashJoins

ORGANIZED BY databricks

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

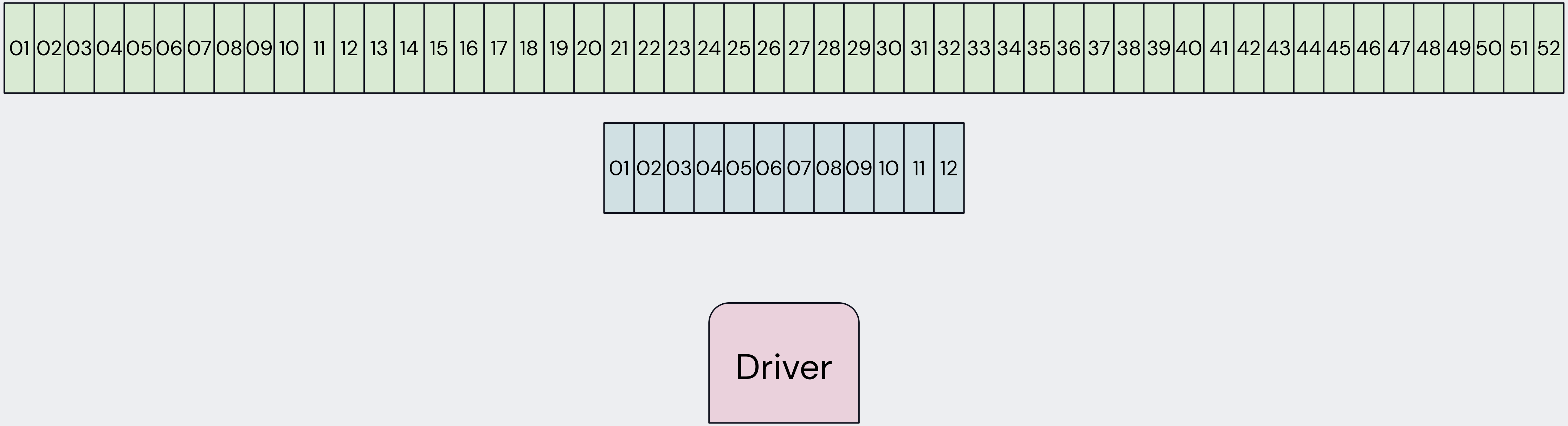
Let's review how the BroadcastHashJoin works…

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

Transactions

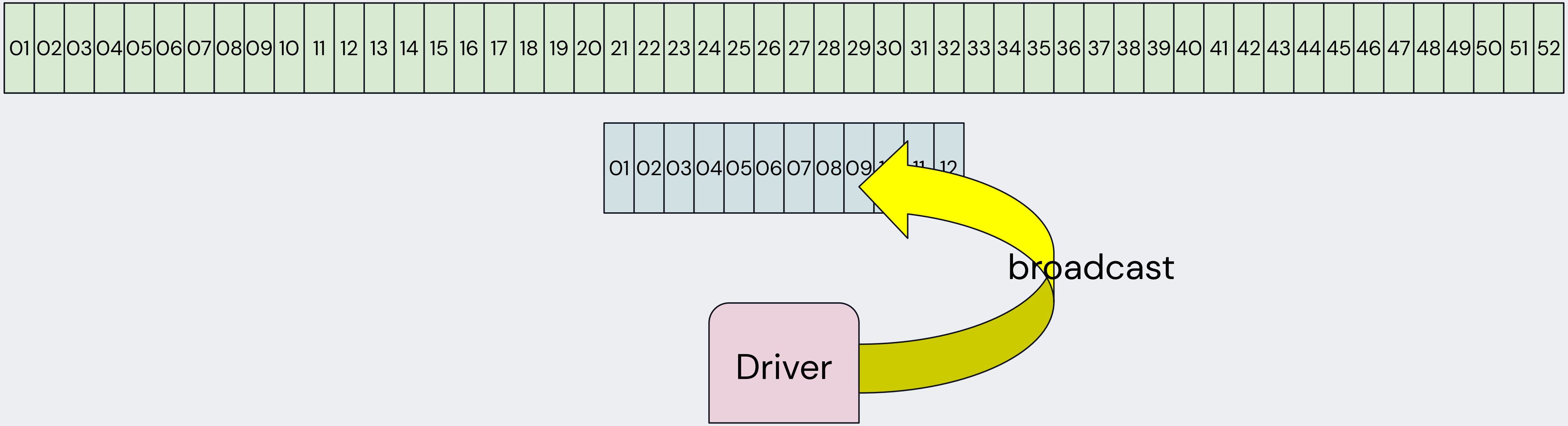
Cities

During planning the driver will partition our two datasets

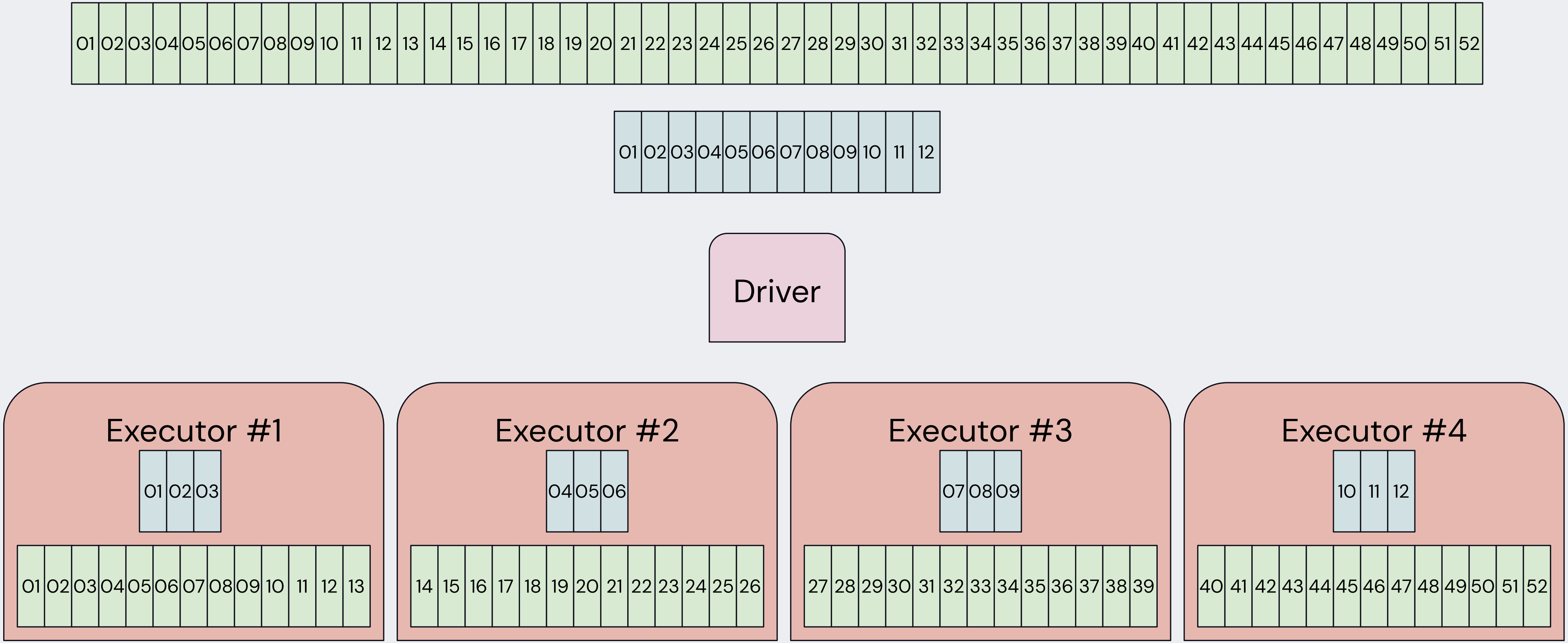


DATA+AI SUMMIT 2022

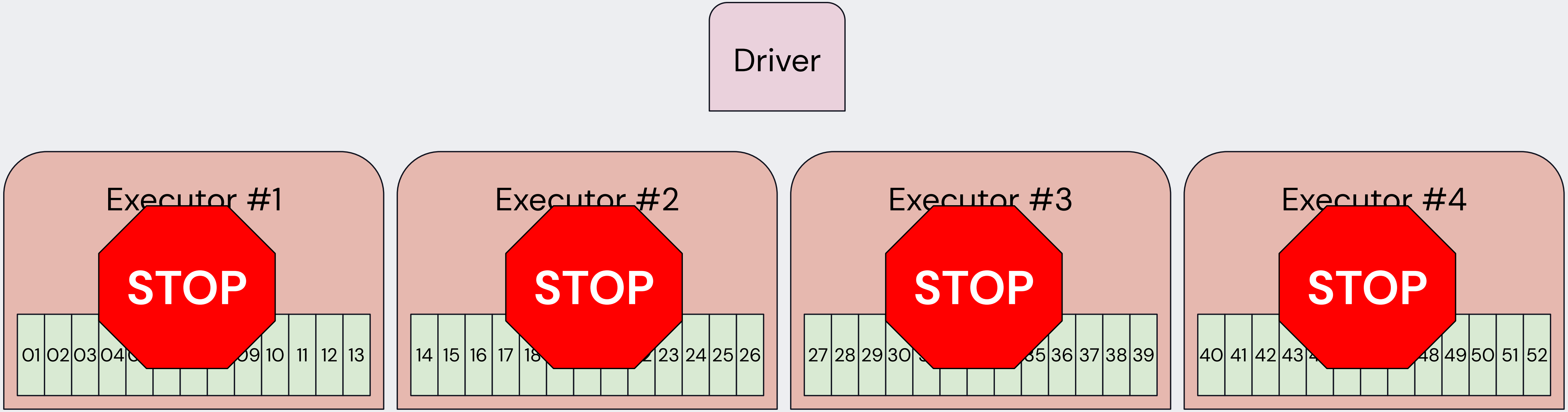
Because the cities table is < 10 MB, the Driver plans a BroadcastHashJoin



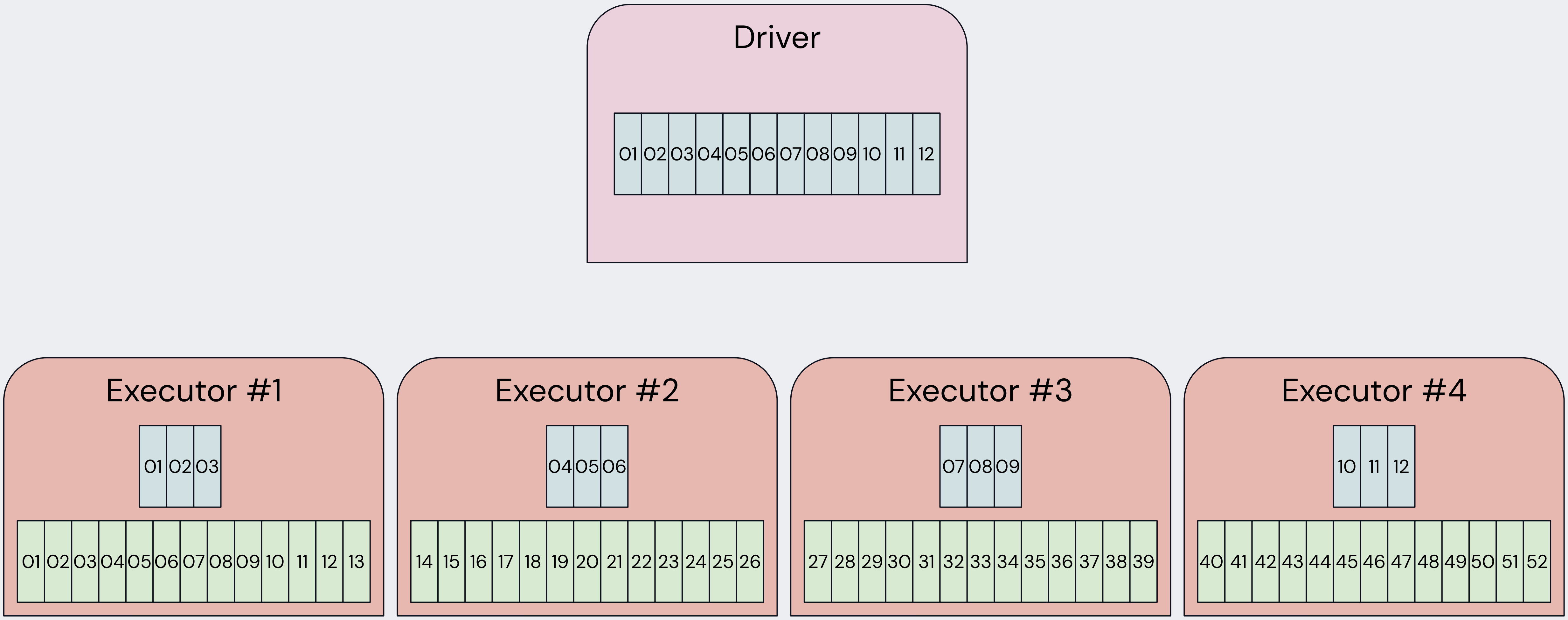
Each executor in turn reads in their assigned partitions



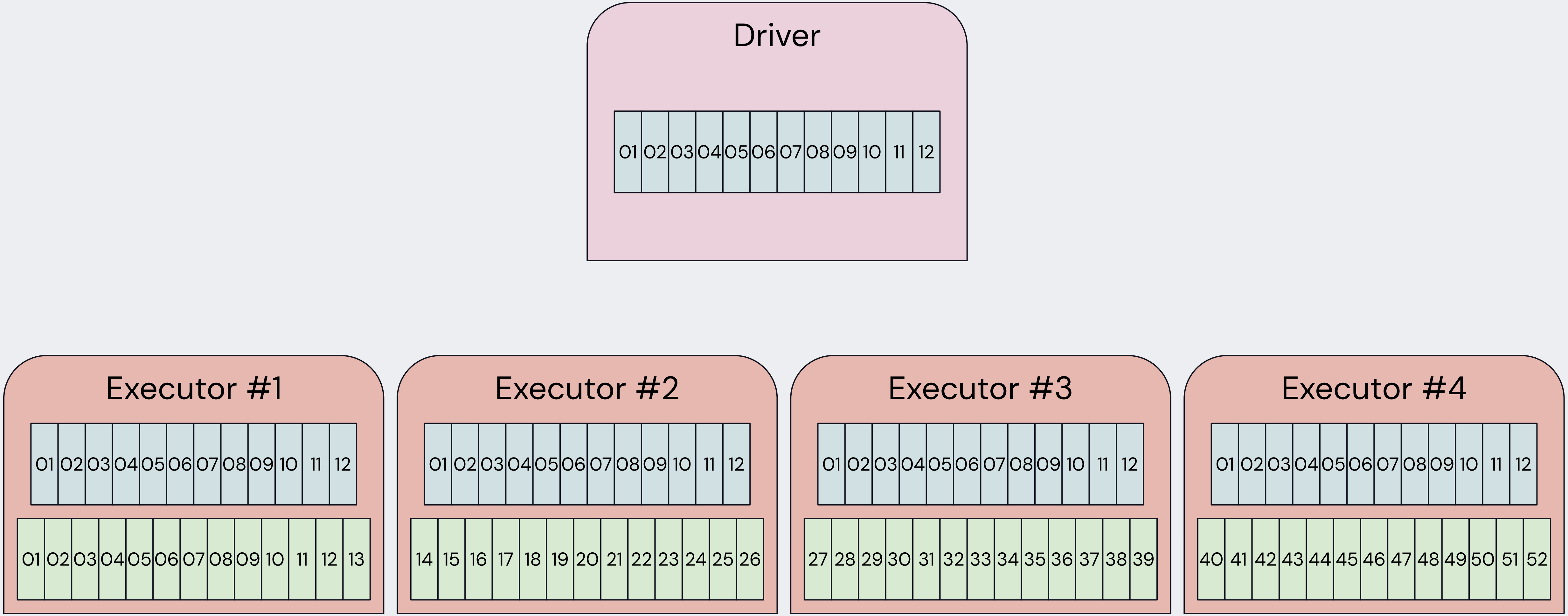
In a traditional join, we would proceed with the map and shuffle



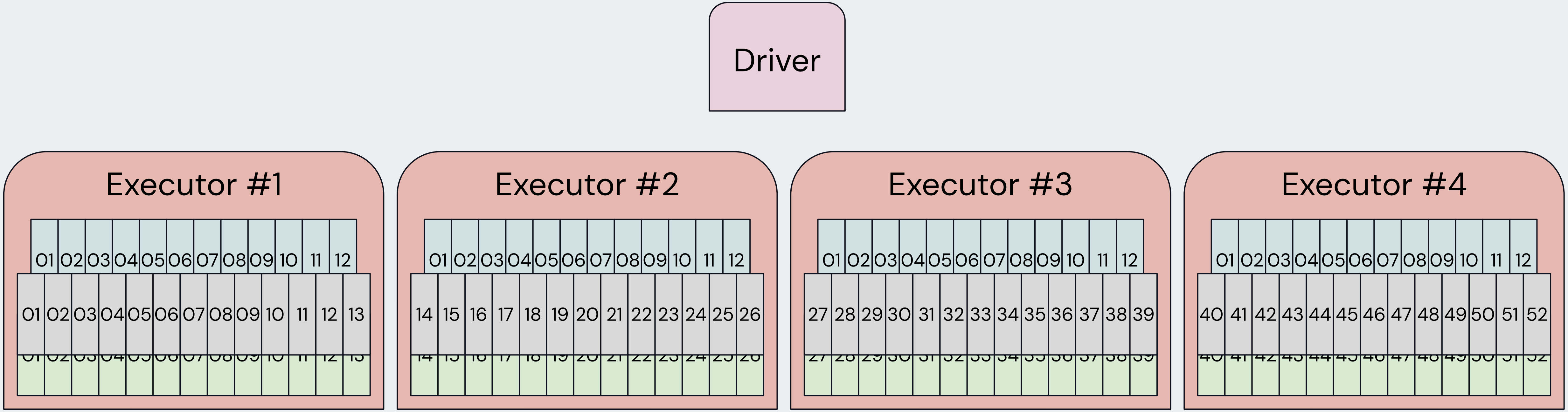
Instead, every partition of the the broadcasted table is sent to the driver



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

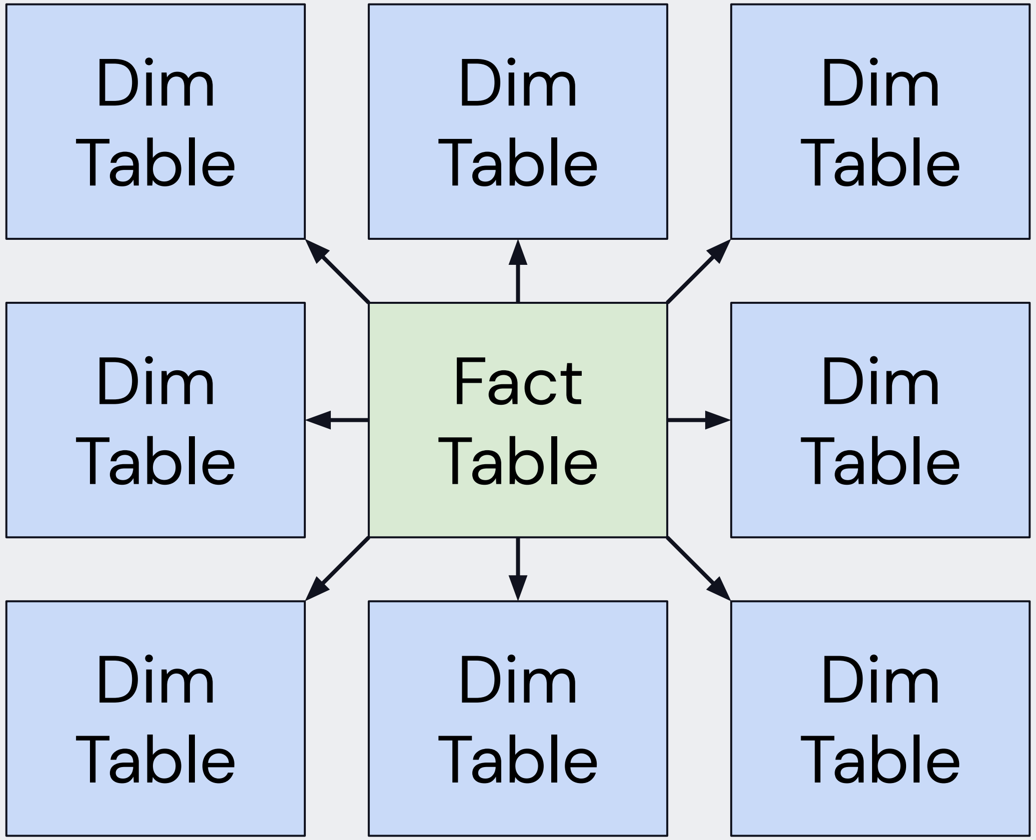


The 5 Most Common Performance Problems (The 5 Ss)
BroadcastHashJoins - Dangers

- 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



DATA+AI SUMMIT 2022

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

DATA+AI
SUMMIT 2022

Optimizing Apache Spark

The Five Most Common Performance Problems: Shuffle Mitigation - Bucketing

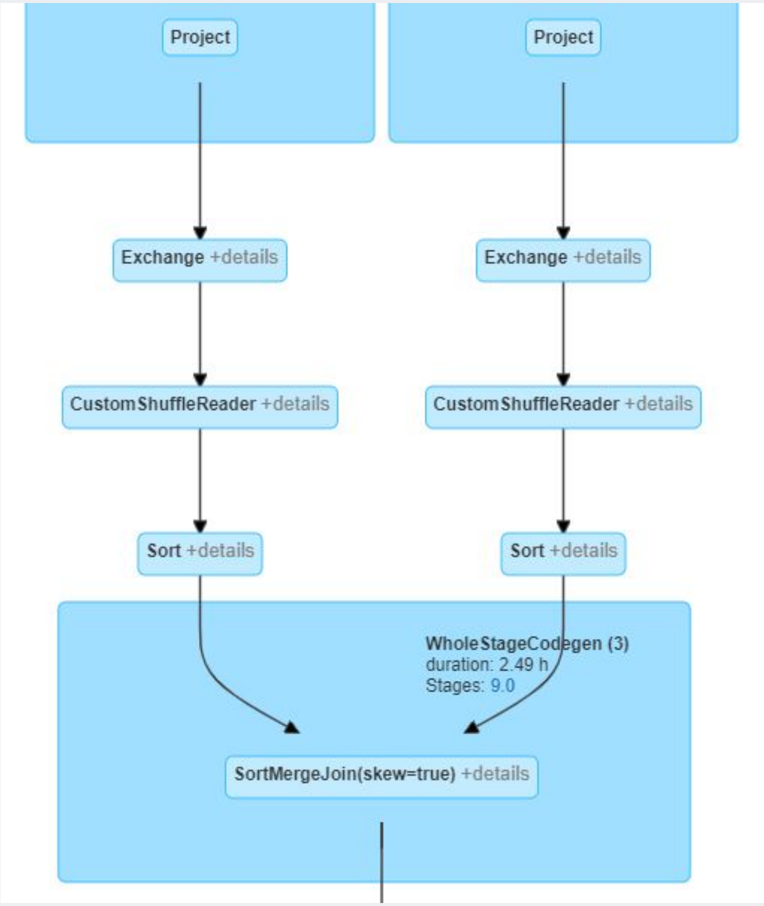
ORGANIZED BY databricks

The 5 Most Common Performance Problems (The 5 Ss)
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

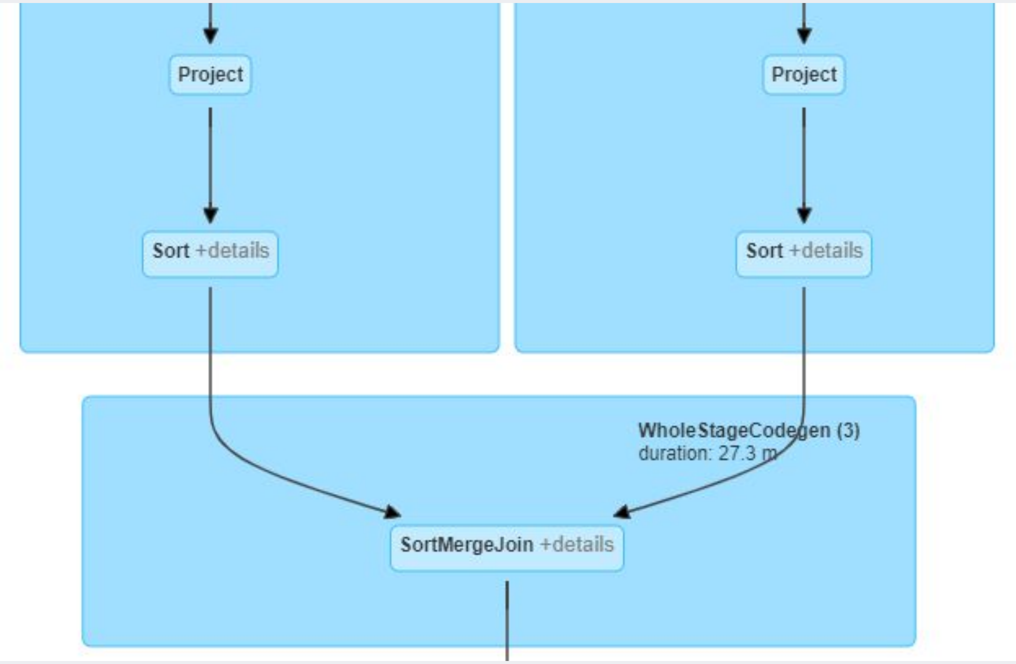
We will cover the benefits of bucketing real quick,
but we will differ how to bucket data for another segment…

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**

The 5 Most Common Performance Problems (The 5 Ss)
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** … *more on maxPartitionBytes later*
- 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

DATA+AI SUMMIT 2022

Optimizing Apache Spark

The Five Most Common Performance Problems: Serialization

ORGANIZED BY databricks

The 5 Most Common Performance Problems (The 5 Ss)
Serialization

- Serialization is the last of our "most common problems"
- Spark 1.x provided significant performance gains over other solutions
- Spark 2.x, namely Spark SQL & the DataFrames API brought even more performance gains by abstracting out the execution plans
- We no longer write "map" operations with custom code but instead express our transformations with the SQL and DataFrames APIs
- That means that with Spark 2.x, when we regress back to code, we are going to see more performance hits

The 5 Most Common Performance Problems (The 5 Ss)
Serialization – Why it's bad

- Spark SQL and DataFrame instructions are compact, and optimized for distributing those instructions from the Driver to each Executor
- When we use code, that code has to be serialized, sent to the Executors and then deserialized before it can be expected
- Python takes an even hard hit because the Python code has to be pickled **AND** Spark must instantiate an instance of the Python interpreter in each and every Executor
- Compared that to the Python version of the DataFrames API which uses Python to express the operations executed in the JVM

The 5 Most Common Performance Problems (The 5 Ss)
Serialization - Catalyst Optimizer

- Besides the cost of serialization, there is another problem…
- These features create an analysis barrier for the Catalyst Optimizer
- The Catalyst Optimizer cannot connect code before and after UDF
- The UDF is a black box which means it limits optimizations to the code before and after, excluding the UDF and how all the code works together

The 5 Most Common Performance Problems (The 5 Ss)
Serialization - How Bad?

Remember our benchmarks?

- See Experiment #5980, **Step D-S** and **Step D-P**
- These use a do-nothing Lambda and a strait read
- The Scala version takes < 15 minutes
- The Python version takes ~2.5 hours!
- All because we executed this python code: **lambda x: None**

The 5 Most Common Performance Problems (The 5 Ss)
Serialization – Scala's Overhead

Let's see how serialization effects Scala in [Experiment #4538 for Scala]

- See **Step D** which uses higher-order functions
  - Uses functions from `org.apache.spark.sql.functions`
  - Note the execution time
- See **Step E** which uses two UDFs
  - See `parseId(..)` and `parseType(..)`
  - Note the execution time
- See **Step F** which uses Typed Transformations
  - See the `map(..)` operation
  - Note the execution time

The 5 Most Common Performance Problems (The 5 Ss)

Serialization – Scala's Overhead, Review

Step	Type	Duration
C	Baseline	~3 min
D	Higher-order Functions	~25 min
E	UDFs	~35 min
F	Typed Transformations	25+ min

Winner

Close Second

The 5 Most Common Performance Problems (The 5 Ss)
Serialization – Python's Overhead

Let's see how serialization effects Python in [Experiment #4538 for Python]

- See **Step D** which uses higher-order functions
  - Uses functions from **pyspark.sql.functions**
  - Note the execution time

- See **Step E** which uses two UDFs
  - See **parseId(..)** and **parseType(..)**
  - Note the execution time

- See **Step F** which uses Pandas (or Vectorized) UDFs
  - See **@pandas_udf parseId(..)** and **@pandas_udf parseType(..)**
  - Note the execution time

The 5 Most Common Performance Problems (The 5 Ss)

Serialization - Python's Overhead, Review

Step	Type	Duration
C	Baseline	~3 min
D	Higher-order Functions	~25 min
E	UDFs	~105 min
F	Panda/Vectorized UDFs	~70 min

Winner

How do the two stack up against each other?

The 5 Most Common Performance Problems (The 5 Ss)
Serialization – Python vs Scala

Step	Type	Scala Duration	Python Duration
C	Baseline	~3 min	~3 min
D	Higher-order Functions	~25 min	~25 min
E	UDFs	~35 min	~105 min
F - Scala	Typed Transformations	~25 min	n/a
F - Python	Panda/Vectorized UDFs	n/a	> 70 min

Same

Really Bad

Bad

The 5 Most Common Performance Problems (The 5 Ss)
Serialization – Why?

Why do we still see such a proliferation of these [poorly performing] features?

- Integration with 3rd-party libraries
  - Common in the data sciences
  - In some cases there is no other choice
- Attempting to integrate with the company's existing frameworks
  - e.g. custom business objects
  - or proprietary libraries
- Migrations from legacy systems like Hadoop
  - Copy and pasting code instead of rewriting them as higher-order functions

What can we do to mitigate these serialization issues?

The 5 Most Common Performance Problems (The 5 Ss)

Serialization - Mitigation

- Short answer, don't use UDFs, Vectorized UDFs or Typed Transformations
- The need for these features is REALLY rare
- The SQL higher-order functions are very robust
- But if you have to...
  - Use Vectorized UDFs over "standard" Python UDFs
  - Use Typed Transformations over "standard" Scala UDFs

Should we rewrite our Spark code to use Scala instead of Python?

No!

Optimizing Apache Spark:

Optimizing with AQE & DPP

Optimizing with AQE & DPP
Custom Dataset Structures

- Reducing the amount of data pulled into Spark will always increase performance
- But ingesting properly structured data can also have a significant impact on performance
- The idea of bucketing data is one form of this idea (designed for optimizing joins)
- Denormalizing otherwise Normalized data is another…

Optimizing with AQE & DPP
Denormalized Datasets – Before & After

Transactions

Column Name	Data Type
trx_id	string
description	string
amount	decimal(38,2)
retailer_id	integer

Retailers

Column Name	Data Type
retailer_id	integer
name	string
city	string
state	string

Transactions & Retailers

Column Name	Data Type
trx_id	string
description	string
amount	decimal(38,2)
retailer_id	integer
name	string
city	string
state	string

Normalized

Denormalized

Optimizing with AQE & DPP
Denormalized Datasets

- Denormalizing aims to eliminate joins by combining fact and dimension tables into one new table
- This strategy mitigates or eliminates
  - Use of the **join()** transformations
  - The use of a **SortMergeJoin** and with that…
  - Skew in key partitions post-join
  - Spill during the exchange and sort
- Consumers would simply query from the new denormalized table

Optimizing with AQE & DPP

Denormalized Datasets – Problems

- Denormalized Datasets are not "normal"
  - A significant amount of "big data" starts in traditional systems
  - DBAs, engineers & analyst have decades of experience w/normalized datasets
  - Requires more disk space than normalized datasets
- There is extra work to denormalize an existing [normalized] dataset
- Without very clear requirements, the act of denormalizing is at best a guess as to what consumers of the data will want or need
- If the underlying data is updated (hourly, daily, weekly, etc) then the act of maintaining the denormalized datasets can become expensive
- It still remains a legitimate strategy for increasing query performance

Optimizing with AQE & DPP
Keep it Normal(ized)

- What if we didn't have to denormalize datasets or create highly customized products?
- What if we could reduce, if not eliminate, the overhead?
- What if Apache Spark could make our data lakes behave more like a relational database?
- Say hello to two new features in Spark 3.0:
  - **Adaptive Query Execution**
  - **Dynamic Partition Pruning**

Optimizing with AQE & DPP

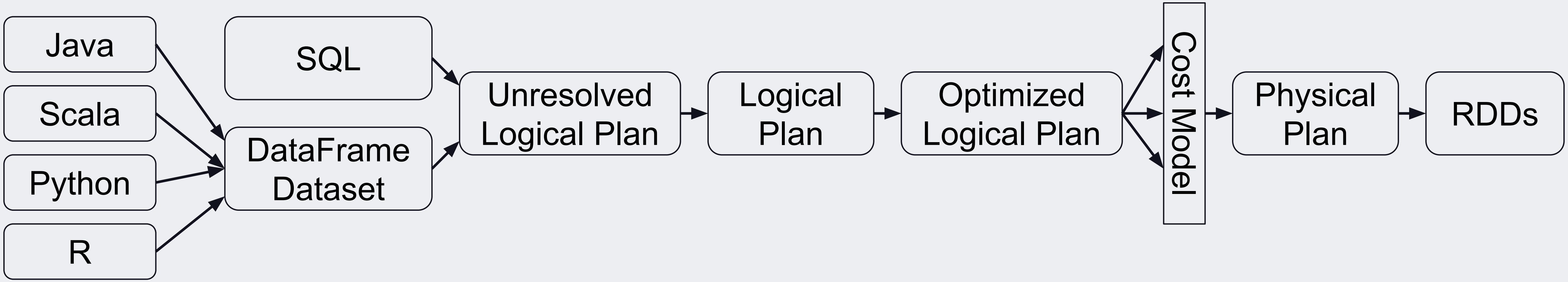
Adaptive Query Execution

- Also referred to as
  - Adaptive Query Optimisation
  - Adaptive Optimisation
- Adaptive Query Execution is an extensible framework
- It's akin to writing rules for the Catalyst Optimizer
- And in Spark 3.0, it must be enabled by setting **spark.sql.adaptive.enabled** to **true**
- Other AQE features may default to enabled, but are still gated by this master configuration flag

Optimizing with AQE & DPP

Adaptive Query Execution vs Catalyst Optimizer

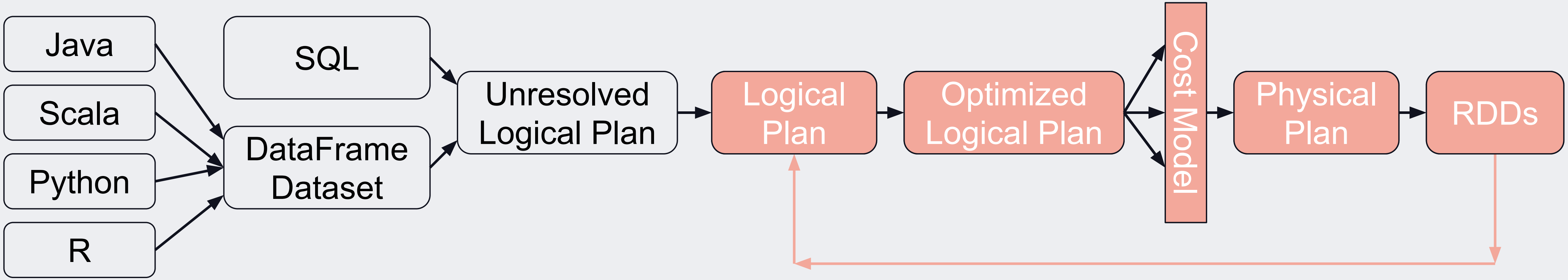
- The **Catalyst Optimizer** is a rules based engine
- It takes the **Logical Plan** and rewrites it as an optimized **Physical Plan**.
- The **Physical Plan** is developed **BEFORE** a query is executed
- For Example...



Optimizing with AQE & DPP

Adaptive Query Execution vs Catalyst Optimizer

- **Adaptive Query Execution**, on the other hand, modifies the **Physical Plan** based on runtime information, for example...



- Let's take a look a three key examples introduced with Spark 3.0
  - Tuning Shuffle Partitions
  - Join Optimization
  - Optimizing Skew Joins

Optimizing Apache Spark

Optimizing with AQE & DPP

Tuning Shuffle Partitions

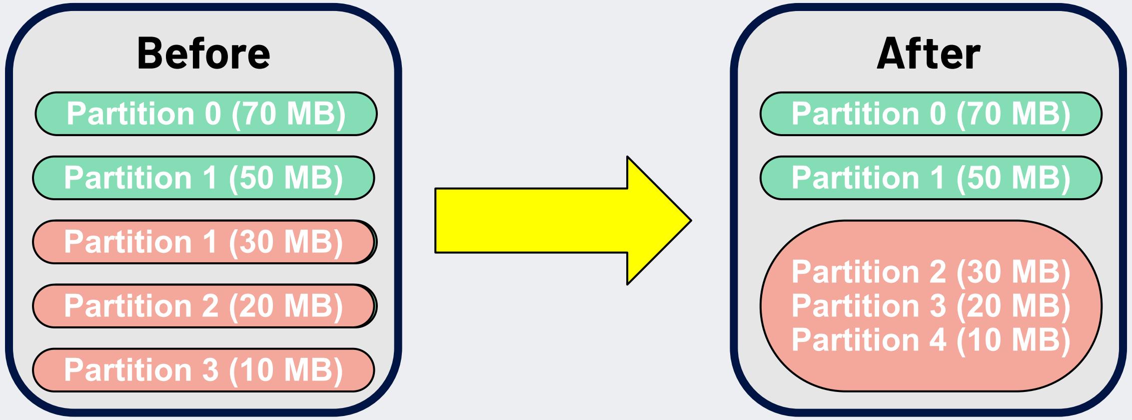
Optimizing with AQE & DPP
AQE - spark.sql.shuffle.partitions

- **spark.sql.shuffle.partitions** - everyone should know of this by now!
  - After every wide transformation, Spark needs to repartition the data
  - Indicates how many partitions Spark will create for the next stage
  - This setting MUST be managed by every user for every job
- The problems with this:
  - Too many partitions, and one has empty if not small spark-partitions putting undue pressure on the scheduler/driver
  - Too few partitions, and one has larger partitions resulting in spill during the exchange and sort if not OOM Errors
  - It can only be set once per job
  - As the number of stages increases, so do the odds of this value being inappropriate for all of the stages

Optimizing with AQE & DPP
AQE – Tuning Shuffle Partitions

- To enable the coalescing of shuffle partitions set **spark.sql.adaptive.coalescePartitions.enabled** to **true**

- The net effect is fewer partitions for subsequent stages, for example...



- Over simplifying, but we now only need to manage **spark.sql.shuffle.partitions** for the expected maximum

Optimizing with AQE & DPP

AQE – Tuning Shuffle Partitions – In Action

See Experiment #2653

- Contrast the **last** job for
  Step B (default) , **Step C** (832) and **Step D** (w/AQE)
  - ...the total execution time (entire command)
  - ...the number of tasks in the final stage of each job
  - ...the stage details for the final stage of each job
  - ...the query plans for the three jobs
- What major element is different in the **Query Plan** for **Step D** versus the other two?

Optimizing with AQE & DPP
AQE - Tuning Shuffle Partitions, Review

Step	Total Duration	Number of Partitions	Stage Details Conclusions	Query Plan Optimization
Step B	~1.5 minutes	825 / 200	Bad distribution / Overhead @200 partitions are 4x Larger Potential Spill	-none-
Step C	~1 minute	825 / 832	Horrible distribution / Overhead	-none-
Step D	**~¾ of a minute**	825 / **17**	Good Distribution / Minor Overhead	**CustomShuffleReader**

Optimizing with AQE & DPP

Tuning Shuffle Partitions – Options

See Coalescing Post Shuffle Partitions for more information

Property Name	Default	Meaning
spark.sql.adaptive.coalescePartitions.enabled	true	When true and **spark.sql.adaptive.enabled** is true, Spark will coalesce contiguous shuffle partitions according to the target size (specified by **spark.sql.adaptive.advisoryPartitionSizeInBytes**), to avoid too many small tasks.
spark.sql.adaptive.coalescePartitions.minPartitionNum	Default Parallelism	The minimum number of shuffle partitions after coalescing. If not set, the default value is the default parallelism of the Spark cluster. This configuration only has an effect when **spark.sql.adaptive.enabled** and **spark.sql.adaptive.coalescePartitions.enabled** are both enabled.
spark.sql.adaptive.coalescePartitions.initialPartitionNum	200	The initial number of shuffle partitions before coalescing. By default it equals to **spark.sql.shuffle.partitions**. This configuration only has an effect when **spark.sql.adaptive.enabled** and **spark.sql.adaptive.coalescePartitions.enabled** are both enabled.
spark.sql.adaptive.advisoryPartitionSizeInBytes	64 MB	The advisory size in bytes of the shuffle partition during adaptive optimization (when **spark.sql.adaptive.enabled** is true). It takes effect when Spark coalesces small shuffle partitions or splits skewed shuffle partition.

Optimizing Apache Spark

Optimizing with AQE & DPP

Join Optimizations

Optimizing with AQE & DPP
AQE – Join Optimization

Joins can be optimized at runtime, for example:

- Table sizes are estimated at planning:
  - A large table is estimated to be 100 GB
  - A small table is estimated to be 11 MB and thus not a candidate for auto-broadcasting
- Both tables are read in as a distinct stages, but **in parallel**
- At runtime, the small table comes in under the 10 MB threshold
- At runtime, AQE adjusts the physical plan to broadcast the small table or potentially employ other strategies like subqueries

Optimizing with AQE & DPP

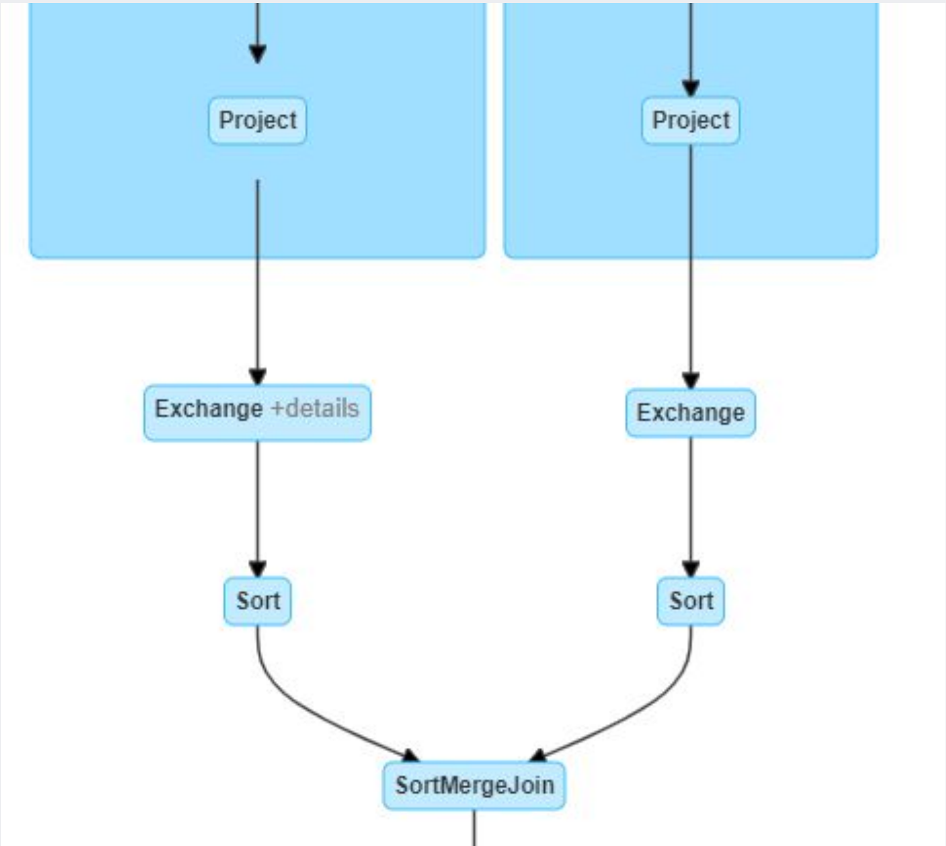
AQE – Join Optimization, In Action

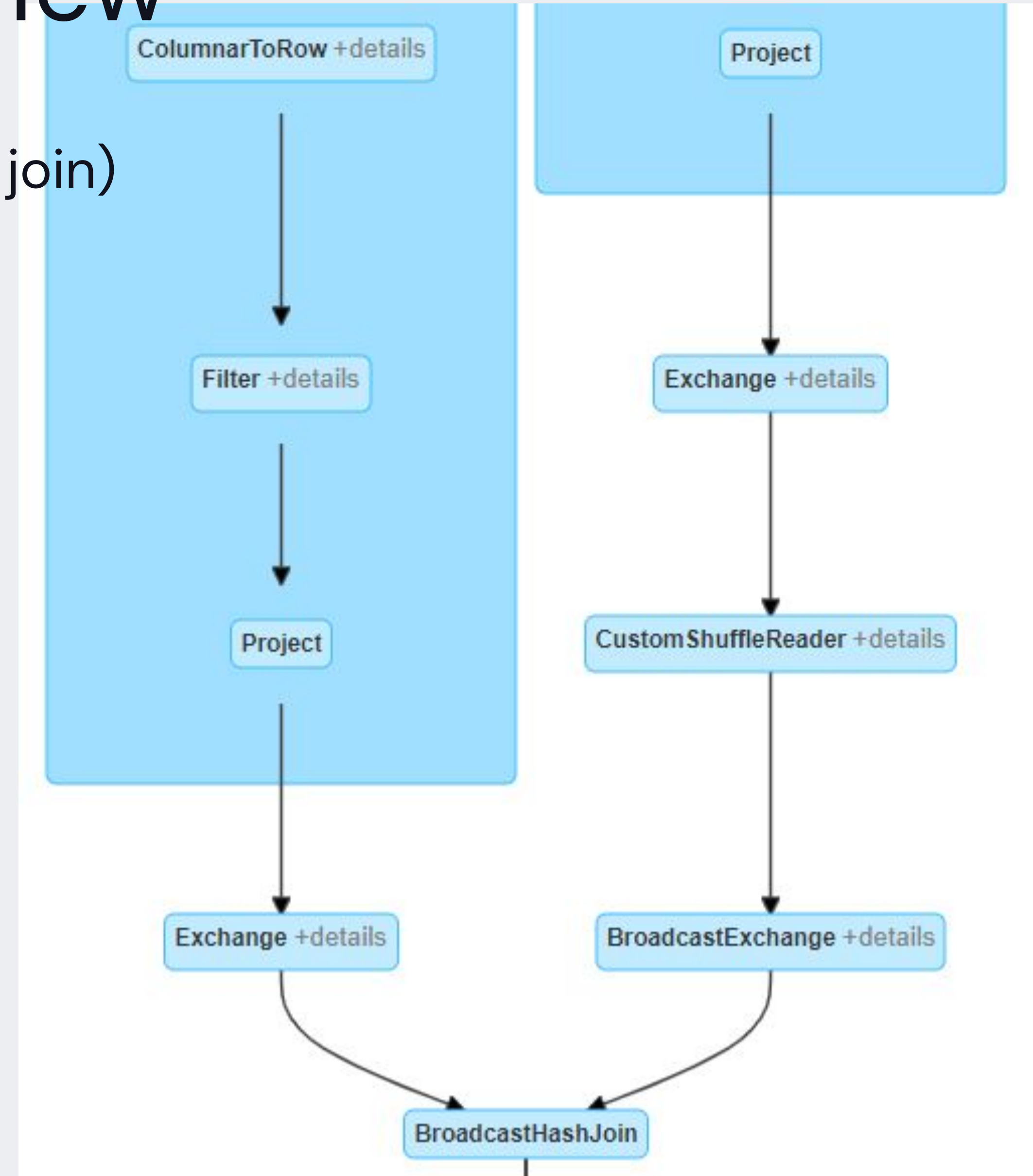
See Experiment #3799A

- Contrast the **Query Plans** for the **last** job for **Step C** (standard) and **Step D** (w/AQE)
- What major difference is there between the two **Query Plans**?

Optimizing with AQE & DPP
AQE - Join Optimization, Review

Kind of Cool: Run the query, and it will initially show a **SortMergeJoin**. Once the ingest stages (left & right of join) completes, the query and physical plan will update to use a **BroadcastHashJoin**





Optimizing with AQE & DPP

Join Optimization – Options

See [Converting sort-merge join to broadcast join] for more information

Property Name	Default	Meaning
spark.sql.adaptive.localShuffleReader.enabled	true	AQE converts sort-merge join to broadcast hash join when the runtime statistics of any join side is smaller than the broadcast hash join threshold
spark.sql.adaptive.nonEmptyPartitionRatioForBroadcastJoin	0.2	The relation with a non-empty partition ratio lower than this config will not be considered as the build side of a broadcast-hash join in adaptive execution regardless of its size.

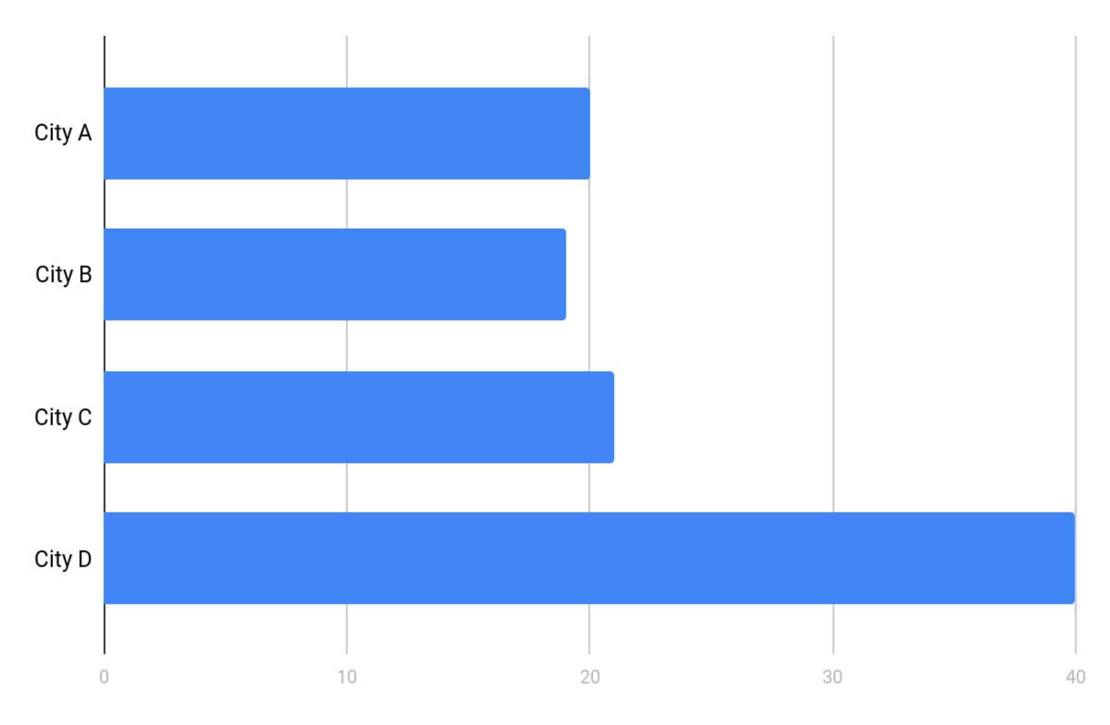
Optimizing Apache Spark

Optimizing with AQE & DPP

Skew Join Optimizations

Optimizing with AQE & DPP
AQE - Optimizing Skew Joins

- Skew is a hard problem to solve for
- Just to review, we can:
  - Salt the skewed keys
  - Employ skew hints
  - Tweak all kinds of settings:
    - Level of Compute
    - Available RAM for Execution
    - The Broadcast-Join Threshold
    - spark.sql.shuffle.partitions
  - Or we can just use Spark 3.x...

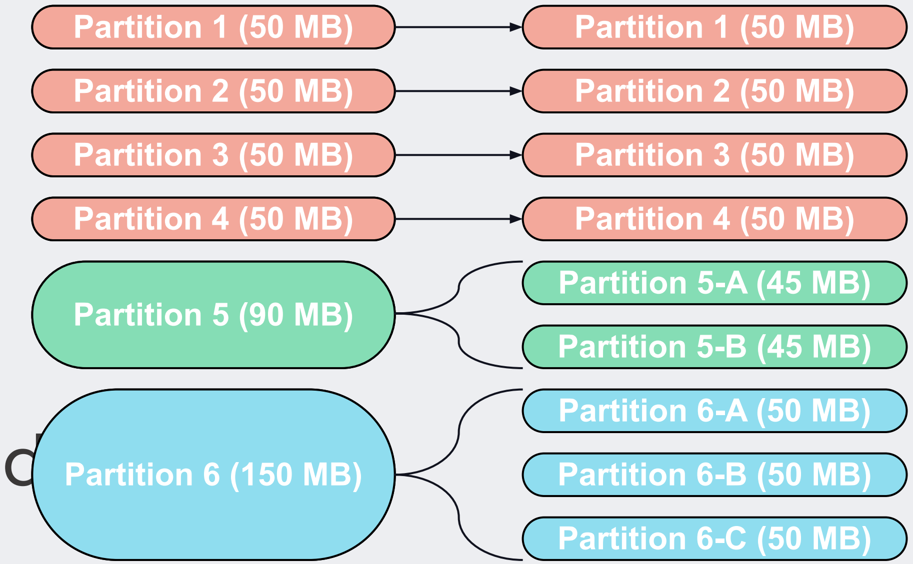


Optimizing with AQE & DPP

AQE – Optimizing Skew Joins

With Spark 3.x, solving for skew is easy and automatic:

- Enable by setting **spark.sql.adaptive.skewJoin.enabled** to **true**
- A partition is skewed if its data size or row count is N times larger than the median & also larger than a predefined threshold
- When skewed, AQE subdivides the one partition into many partitions and employs additional tasks to process
- For example…



Optimizing with AQE & DPP
AQE - Optimizing Skew Joins, In Action

See [Experiment #1596]

- Contrast the **last** job for **Step C** (standard) and **Step E** (w/AQE)
  - …the total execution time (entire command)
  - …the number of tasks for the final stage of each job
  - …the stage details for the final stage of each job
    - Event Timeline
    - Presence or absence of Spill
    - Shuffle Read Size / Records
  - ..the query plans for the two jobs
- What major difference is there in the **Query Plan** for **Step C** vs **Step E**?
- How many skewed partitions were reported in the **Query Plan**?

Optimizing with AQE & DPP
AQE - Optimizing Skew Joins, Review

Step	Cmd Duration	Tasks	Event Timeline	Spill	Shuffle Read Size / Records	Skew Count	Query Plan Optimization
C	~29 min	832	Bad	Yes	**0/0/<100K/<500M/2.6G**	n/a	-none-
E	~**25 min**	**1489**	**Good**	**No**	**0/115M/115M/125M/130M**	**1,327**	**CustomShuffleReader SortMergeJoin(skew)**

Optimizing with AQE & DPP

Optimizing Skew Joins – Options

See Optimizing Skew Join for more information

Property Name	Default	Meaning
spark.sql.adaptive.skewJoin.enabled	true	When true and **spark.sql.adaptive.enabled** is true, Spark dynamically handles skew in sort-merge join by splitting (and replicating if needed) skewed partitions.
spark.sql.adaptive.skewJoin.skewedPartitionFactor	10	A partition is considered as skewed if its size is larger than this factor multiplying the median partition size and also larger than **spark.sql.adaptive.skewedPartitionThresholdInBytes**
spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes	256 MB	A partition is considered as skewed if its size in bytes is larger than this threshold and also larger than **spark.sql.adaptive.skewJoin.skewedPartitionFactor** multiplying the median partition size. Ideally this config should be set larger than **spark.sql.adaptive.advisoryPartitionSizeInBytes**.

Optimizing Apache Spark

Optimizing with AQE & DPP

Dynamic Partition Pruning

New Strategies for Spark 3.x
Dynamic Partition Pruning

- Consider a 100GB table of **transactions** and a 30MB table of **cities**
- Without any filtering, this is a massive shuffle operation
- A 1-minute query on **transactions** can easily become an hour long join with the subsequent shuffle
- Filtering the **cities** table by country (e.g. USA only) means we are now joining a 100GB table to a 15MB table
- This is still a massive shuffle operation!

New Strategies for Spark 3.x
DPP – What Can We Do?

Both of these solutions can work but only to a very limited degree

We can broadcast the cities table:

- At 10+ MB, it's still too big for auto-broadcasting
- We would have to force a broadcast with a hint
- This wouldn't be an option if our "big" table was **1 TB** and our "small" table was **100 GB**

But it may not be reasonable to expect the consumers of your data to do this

We can create our own subquery

- Explicitly select all the US city ids and **collect()** them as the array **city_ids**
- Filter both the **transactions** and **cities** table with **$"city_id".isin(city_ids)**
- Join the ~70GB **transactions** table to the ~15MB **cities** table

New Strategies for Spark 3.x

What Does DPP Actually Do?

Dynamic Partition Pruning uses a combination of those strategies

- Spark will produce a query on the "small" table
- The result of which is used to produce a "dynamic filter" similar to our list of city_ids
- The "dynamic filter" is then broadcast to each executor
- At runtime, Spark's physical plan is adjusted so that our "large" table is reduced with the "dynamic filter"
- And if possible, that filter will employ a predicate pushdown so as to avoid an **InMemoryTableScan**

New Strategies for Spark 3.x
Dynamic Partition Pruning, In Action

See Experiment #3799B, **Step C** (standard)

- Note that DPP is enabled by default in Spark 3.0 (no contrast this time)
- Recalling how a "standard" **Scan** / **Filter** / **SortMergeJoin** works
- Identify the difference in this **Query Plan** compared to other queries we have seen

New Strategies for Spark 3.x

Dynamic Partition Pruning, Review

- The “left” of the join is the **transactions** table & the “right” is the **cities** table
- The “left” starts by scanning the **cities** table
- The results of the **cities** scan is fed into the **Subquery**
- The subsequent **Scan parquet** employes its predicate push down to…
  - …read 27 of 100 files
  - …read in only 7.5GB of the full 100GB
  - …read in only 2M of the 2B records
- The “right” is processed and ultimately fed into a **SortMergeJoin**
- Further proof is in the **Physical Plan** – see the **DataFilter** & **PushedFilter**

DATA+AI SUMMIT 2022

Optimizing Apache Spark

Designing Clusters for High Performance

Cluster Configurations Scenarios

ORGANIZED BY databricks

Cluster Configurations Scenarios
Getting Started...

Taking into consideration everything we know now...

- Who will be using the cluster?
- What will the cluster be used for?
- Where will the cluster and/or data reside?
- When are the results needed?
- How do I control/predict the costs?

Can we predict, for a given scenario, which cluster configuration and set of features will best meet the needs of each specific scenario?

Cluster Configurations Scenarios
“It Depends”

Really… it does depend…

- There is rarely a black or white, right or wrong, answer
- There are many different factors that could justify various decisions
- The conclusions presented here are generalizations only

Cluster Configurations Scenarios
Typical Analyst

Which of the following cluster configurations is best / least suited for a single SQL or Data Analyst?

Cluster A	Cluster B	Cluster C	Cluster D
Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB
Total Cores: 160	Total Cores: 160	Total Cores: 160	Total Cores: 160
VM Type: X-5	VM Type: X-4	VM Type: X-3	VM Type: X-2
Total VMs: 1	Total VMs: 2	Total VMs: 4	Total VMs: 8
400 GB / Exec	200 GB / Exec	100 GB / Exec	50 GB / Exec
160 Cores / Exec	80 Cores / Exec	40 Cores / Exec	20 Cores / Exec
Silver Tables	Silver Tables	Silver Tables	Silver Tables

Category
- Memory Optimized
- Compute Optimized
- Storage Optimized
- GPU Optimized
- General Purpose

Features
- Delta Cache
- Auto Termination
- Auto Scale VMs
- Auto Scale Storage
- High Concurrency
- Cluster Pool

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Configurations Scenarios
Team of Analysts

Which of the following cluster configurations is best / least suited for a team of SQL and/or Data Analysts?

Cluster A
Total RAM: 400 GB
Total Cores: 160

VM Type: X-5
Total VMs: 1

400 GB / Exec
160 Cores / Exec

Silver Tables

Cluster B
Total RAM: 400 GB
Total Cores: 160

VM Type: X-4
Total VMs: 2

200 GB / Exec
80 Cores / Exec

Silver Tables

Cluster C
Total RAM: 400 GB
Total Cores: 160

VM Type: X-3
Total VMs: 4

100 GB / Exec
40 Cores / Exec

Silver Tables

Cluster D
Total RAM: 400 GB
Total Cores: 160

VM Type: X-2
Total VMs: 8

50 GB / Exec
20 Cores / Exec

Silver Tables

Category
- Memory Optimized
- Compute Optimized
- Storage Optimized
- GPU Optimized
- General Purpose

Features
- Delta Cache
- Auto Termination
- Auto Scale VMs
- Auto Scale Storage
- High Concurrency
- Cluster Pool

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Configurations Scenarios
Cluster Stability

Which of the following cluster configurations is most / least likely to survive a [random] executor failure during a long-running job?

Cluster A	Cluster B	Cluster C	Cluster D	Cluster E
Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB
Total Cores: 160	Total Cores: 160	Total Cores: 160	Total Cores: 160	Total Cores: 160
VM Type: X-5	VM Type: X-4	VM Type: X-3	VM Type: X-2	VM Level: X-1
Total VMs: 1	Total VMs: 2	Total VMs: 4	Total VMs: 8	Total VMs: 16
400 GB / Exec	200 GB / Exec	100 GB / Exec	50 GB / Exec	25 GB / Exec
160 Cores / Exec	80 Cores / Exec	40 Cores / Exec	20 Cores / Exec	10 Cores / Exec
Random Datasets	Random Datasets	Random Datasets	Random Datasets	Random Datasets

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Configurations Scenarios
Training ML Models, 1st Iteration

Which of the following cluster configurations is best / least suited for training the first iteration of an ML or DL Model?

Cluster A
Total RAM: 400 G
Total Cores: 160

VM Type: X-5
Total VMs: 1

400 GB / Exec
160 Cores / Exec

Silver & Gold Tables

Cluster B
Total RAM: 400 G
Total Cores: 160

VM Type: X-4
Total VMs: 2

200 GB / Exec
80 Cores / Exec

Silver & Gold Tables

Cluster C
Total RAM: 400 G
Total Cores: 160

VM Type: X-3
Total VMs: 4

100 GB / Exec
40 Cores / Exec

Silver & Gold Tables

Cluster D
Total RAM: 400 GB
Total Cores: 160

VM Type: X-2
Total VMs: 8

50 GB / Exec
20 Cores / Exec

Silver & Gold Tables

Category
- Memory Optimized
- Compute Optimized
- Storage Optimized
- GPU Optimized
- General Purpose

Features
- Delta Cache
- Auto Termination
- Auto Scale VMs
- Auto Scale Storage
- High Concurrency
- Cluster Pool

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

DATA+AI SUMMIT 2022

Cluster Configurations Scenarios
Training ML Models, 2nd+ Iteration

Which of the following cluster configurations is best / least suited for training the second iteration of an ML or DL Model?

Cluster A	Cluster B	Cluster C	Cluster D
Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB
Total Cores: 160	Total Cores: 160	Total Cores: 160	Total Cores: 160
VM Type: X-5	VM Type: X-4	VM Type: X-3	VM Type: X-2
Total VMs: 1	Total VMs: 2	Total VMs: 4	Total VMs: 8
400 GB / Exec	200 GB / Exec	100 GB / Exec	50 GB / Exec
160 Cores / Exec	80 Cores / Exec	40 Cores / Exec	20 Cores / Exec
Gold Tables	Gold Tables	Gold Tables	Gold Tables

Category
- Memory Optimized
- Compute Optimized
- Storage Optimized
- GPU Optimized
- General Purpose

Features
- Delta Cache
- Auto Termination
- Auto Scale VMs
- Auto Scale Storage
- High Concurrency
- Cluster Pool

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Configurations Scenarios
Garbage Collection

Which of the following cluster configurations is most / least likely to be adversely impacted by a long garbage collection sweep?

Cluster A
Total RAM: 400 GB
Total Cores: 160

VM Type: X-5
Total VMs: 1

400 GB / Exec
160 Cores / Exec

Random Datasets

Cluster B
Total RAM: 400 GB
Total Cores: 160

VM Type: X-4
Total VMs: 2

200 GB / Exec
80 Cores / Exec

Random Datasets

Cluster C
Total RAM: 400 GB
Total Cores: 160

VM Type: X-3
Total VMs: 4

100 GB / Exec
40 Cores / Exec

Random Datasets

Cluster D
Total RAM: 400 GB
Total Cores: 160

VM Type: X-2
Total VMs: 8

50 GB / Exec
20 Cores / Exec

Random Datasets

Cluster E
Total RAM: 400 GB
Total Cores: 160

VM Level: X-1
Total VMs: 16

25 GB / Exec
10 Cores / Exec

Random Datasets

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Configurations Scenarios
General OOM Error

Which of the following cluster configurations is most / least likely to encounter an OOM Error?

Cluster A	Cluster B	Cluster C	Cluster D	Cluster E
tal RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB
Total Cores: 160	Total Cores: 160	Total Cores: 160	Total Cores: 160	Total Cores: 160
VM Type: X-5	VM Type: X-4	VM Type: X-3	VM Type: X-2	VM Level: X-1
Total VMs: 1	Total VMs: 2	Total VMs: 4	Total VMs: 8	Total VMs: 16
400 GB / Exec	200 GB / Exec	100 GB / Exec	50 GB / Exec	25 GB / Exec
160 Cores / Exec	80 Cores / Exec	40 Cores / Exec	20 Cores / Exec	10 Cores / Exec
Random Datasets	Random Datasets	Random Datasets	Random Datasets	Random Datasets

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Configurations Scenarios
Caching Induced OOM Error

Which of the following usage cases is most / least likely to induce an OOM Error induced by caching?

An ETL Job that is consuming CSV data, updating data types, removing duplicates and then writing it out to parquet

Not Caching

A report that joins three tables and writes the result to a Delta table used by BI tools

Not Caching

A team of 5 analyst engaged in heavy, ad hoc analysis against a single shared cluster

Excessive Caching

A data scientist that is training the first iteration of a model against a 1,000 GB dataset

Heavy Caching

A single analyst attempting to validate sales-tax calculations for the previous year against a well formed 100 GB dataset

Light Caching

Cluster Configurations Scenarios
More Cores == More Money   Version #1

Assuming the data in 320 partitions is equally distributed, which cluster configuration will cost the most / least amount of money for this job?

Cluster A
Total RAM: **400 GB**
Total Cores: **160**
VM Type: **X-5**
Total VMs: **1**
400 GB / Exec
160 Cores / Exec

320 Partitions

Cluster B
Total RAM: **400 GB**
Total Cores: **160**
VM Type: **X-4**
Total VMs: **2**
200 GB / Exec
80 Cores / Exec

320 Partitions

Cluster C
Total RAM: **400 GB**
Total Cores: **160**
VM Type: **X-3**
Total VMs: **4**
100 GB / Exec
40 Cores / Exec

320 Partitions

Cluster D
Total RAM: **400 GB**
Total Cores: **160**
VM Type: **X-2**
Total VMs: **8**
50 GB / Exec
20 Cores / Exec

320 Partitions

Cluster E
Total RAM: **400 GB**
Total Cores: **160**
VM Level: **X-1**
Total VMs: **16**
25 GB / Exec
10 Cores / Exec

320 Partitions

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Configurations Scenarios
More Cores == More Money Version #2

Assuming each partition takes 3 minutes to process... Calculate the **compute-time**, **number of iterations** and **run-time** for each scenario:

Cluster A
Total RAM: 400 GB
Total Cores: 160

VM Type: X-5
Total VMs: 1

400 GB / Exec
160 Cores / Exec

2 iterations
6 minutes

320 Partitions

Cluster B
Total RAM: 200 GB
Total Cores: 80

VM Type: X-4
Total VMs: 1

200 GB / Exec
80 Cores / Exec

4 iterations
12 minutes

320 Partitions

Cluster C
Total RAM: 200 GB
Total Cores: 80

VM Type: X-3
Total VMs: 3

100 GB / Exec
40 Cores / Exec

2.6 iterations
9 minutes

320 Partitions

Cluster D
Total RAM: 300 GB
Total Cores: 120

VM Type: X-2
Total VMs: 2

50 GB / Exec
20 Cores / Exec

8 iterations
24 minutes

320 Partitions

Cluster E
Total RAM: 1000 GB
Total Cores: 400

VM Level: X-1
Total VMs: 40

25 GB / Exec
10 Cores / Exec

0.8 iterations
3 minutes

320 Partitions

Over Provisioned !!
On Spark 3, it might be a good thing !!!

Cluster Configurations Scenarios
Batch ETL: Raw -> Bronze

Which of the following cluster configurations is best / least suited for a simple ETL job that does not employ wide transformations (no joins)?

Cluster A
Total RAM: **400 GB**
Total Cores: **160**

VM Type: **X-5**
Total VMs: **1**

400 GB / Exec
160 Cores / Exec

320 Partitions

Cluster B
Total RAM: **400 GB**
Total Cores: **160**

VM Type: **X-4**
Total VMs: **2**

200 GB / Exec
80 Cores / Exec

320 Partitions

Cluster C
Total RAM: **400 GB**
Total Cores: **160**

VM Type: **X-3**
Total VMs: **4**

100 GB / Exec
40 Cores / Exec

320 Partitions

Cluster D
Total RAM: **400 GB**
Total Cores: **160**

VM Type: **X-2**
Total VMs: **8**

50 GB / Exec
20 Cores / Exec

320 Partitions

Category
- Memory Optimized
- Compute Optimized
- Storage Optimized
- GPU Optimized
- General Purpose

Features
- Delta Cache
- Auto Termination
- Auto Scale VMs
- Auto Scale Storage
- High Concurrency
- Cluster Pool

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Cluster Configurations Scenarios
Batch ETL: Silver -> Gold

Which of the following cluster configurations is best / least suited for an ETL job that unions and joins multiple tables into a single, new table?

Cluster A	Cluster B	Cluster C	Cluster D
Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB	Total RAM: 400 GB
Total Cores: 160	Total Cores: 160	Total Cores: 160	Total Cores: 160
VM Type: X-5	VM Type: X-4	VM Type: X-3	VM Type: X-2
Total VMs: 1	Total VMs: 2	Total VMs: 4	Total VMs: 8
400 GB / Exec	200 GB / Exec	100 GB / Exec	50 GB / Exec
160 Cores / Exec	80 Cores / Exec	40 Cores / Exec	20 Cores / Exec
320 Partitions	320 Partitions	320 Partitions	320 Partitions

Category
- Memory Optimized
- Compute Optimized
- Storage Optimized
- GPU Optimized
- General Purpose

Features
- Delta Cache
- Auto Termination
- Auto Scale VMs
- Auto Scale Storage
- High Concurrency
- Cluster Pool

For this scenario, it can be assumed that each cluster has the same level of compute (total cores) and storage (total RAM)

Optimizing Apache Spark

Designing Clusters for High Performance - Condensed

Designing Clusters for High Performance

Before designing the cluster, we need to answer 6 questions:

- Who will be using the cluster?
- What will the cluster be used for?
- Where will the cluster [and data] reside?
- When are the results needed?
- How do I control/predict the costs?
- Why do I care about all these details?

Designing Clusters for High Performance
Who will be using the cluster?

At one level, we can split on personas…

- Data Analyst
- SQL Analyst
- Data Scientist
- Data Engineers
- …and everyone else (intentionally oversimplified)

Designing Clusters for High Performance
Who - Groups

Besides personas, we also have to consider groups:

- Different data restrictions for Group A vs Group B
- Groups with heavy cluster demands (e.g. engineers)
- Groups with light cluster demands (e.g. SQL Analyst)
- Groups that will share a cluster

Designing Clusters for High Performance
What will the cluster be used for?

- Ad Hoc Data Analysis
- Reporting Generation
- Training ML & Deep Learning Models

- Structured Streaming Jobs
- Batch ETL
- Data Pipelines

How a cluster is used often follows the persona of the person using it

Designing Clusters for High Performance
Where will the cluster [and data] reside?

Where is often dictated to us, but consider these options…

- Personal PC or Laptop
- On-Prem
- Cloud (MSA, AWS, GCP, Other)
- Gov-Cloud
- Various Cloud Regions

Designing Clusters for High Performance
When are the results needed?

A Spark job's SLA generally refers to how long it takes to "deliver" data

- Real-Time
  - The data needs to be "processed" as soon as it arrives
- Near Real-Time
  - The data needs to be "processed" faster than it arrives
- ...and everything else kind of depends

Designing Clusters for High Performance When – “It Depends” #1

Example #1: Yesterday’s Sales

- Data arrives at midnight
- The report must be ready by 9 AM the following morning
- We have up to 9 hours to “deliver” the data
- Given the hours of execution, cluster stability might be a concern
- Multiple executors will help mitigate this, but we may want to limit ourselves to 4 hours of execution in case it has to be reran
- In this case a job-specific cluster sized and tuned to 4 hours of execution would be enough to support retrying the job
- There is no need/harm to tune to 1 hour of execution

Designing Clusters for High Performance When – “It Depends” #2

Example #2: Last Month’s Sales

- Data is collected over the course of the month (1st to ~31st)
- The report must be ready by the 7th of the following month
- We have up to 7 days to “deliver” the data
- Our untuned implementation takes 24 hours to complete
- A commodity or even a shared cluster would suffice for this job
- If performance is impacted by low memory (e.g. spill) or other jobs, there is still plenty of time. A job-specific cluster may be unwarranted.
- Prudence would dictate that one not tune this job
- The cost of tuning this job is not justifiable given its SLA and possible labor

Designing Clusters for High Performance
How do I control/predict the costs?

The price between a **Level-N** VM and a **Level-N+1** VM is 2x the cost, with 2x the resources

Level	Cores	Size	Cloud-A		Cloud-B	
1	4	Small	30.5 GB	$0.266 / hour	28 GB	$0.299 / hour
2	8	Medium	61.0 GB	$0.532 / hour	56 GB	$0.598 / hour
3	16	Large	122.0 GB	$1.064 / hour	112 GB	$1.196 / hour

Designing Clusters for High Performance Costs - Actual Consumption Cost

It's all about compute-time!

Assume you have a job with 256 partitions and that each partition takes 2 minute to process.

Level	Cores	VMs	Max Compute (cores * VMs)	Iterations (max/part)	Actual Durations (iterations * min)	~Price/Hour (level $ * VMs)	VM Costs (VMs * dur * price / 60)
1	4	1	4	64	128 minutes	$0.283	60¢
1	4	64	256	1	[illegible] minutes	$0.283	60¢
2	8	1	8	[illegible]	[illegible]utes	$0.565	60¢
3	16	1	[illegible]	[illegible]	[illegible]s	$1.130	60¢
3	16	8	1[illegible]	[illegible]	[illegible] minutes	$1.130	60¢

And it's always 512 minutes

Designing Clusters for High Performance Costs - Developer Costs

Another factor to consider is the cost of the developers:

- What are they doing when the job is running?
- How much time does it take to tune?

Let the money/costs decide !!!

Level	Cores	VMs	Max Compute (cores * VMs)	Iterations (max/part)	Actual Durations (iterations * min)	~Price/Hour (level $ * VMs)	Co	r dur / 60)
1	4	1	4	64	128 min	$0.283	60¢	$106.66
1	4	64	256	1	2 min	$0.283	60¢	$1.66
2	8	1	8	32	64 min	$0.565	60¢	$53.33
3	16	1	16	16	32 min	$1.130	60¢	$26.66
3	16	8	128	2	4 min	$1.130	60¢	$3.33

Optimizing Apache Spark

Designing Clusters for High Performance VM Selection

Designing Clusters for High Performance VM Selection: Effect on Shuffles

If cost is not a primary factor, what about the effect on performance?

Level	Cores	VMs	Max Compute (cores * VMs)	Iterations (max/part)	Actual Durations (iterations * min)	Notes
1	4	1	4	64	128 minutes	No network IO
1	**4**	**64**	**256**	**1**	**2 minutes**	**Heavy network IO between 64 VMs**
2	8	1	8	32	64 minutes	No network IO
3	16	1	16	16	32 minutes	No network IO
3	16	8	128	2	4 minutes	Reasonable(?) network IO
7	**256**	**1**	**256**	**1**	**2 minutes**	**Most optimal shuffle experience**

Designing Clusters for High Performance VM Selection: Categories

So which VM should we use? Start by breaking them down by category:

Categorization	Amazon	GBs	Cores	MS Azure	GBs	Cores
Memory Optimized	r4.xlarge	30.5	4	DS12_v2	28.0	4
Compute Optimized	c5.xlarge	8.0	4	F4s	8.0	4
Storage Optimized **Delta Cache**	i3.xlarge**	30.5	4	L4s**	32.0	4
GPU Optimized	p2.xlarge	61.0	1	NC6s_v3	112.0	1
General Purpose	m5.xlarge	16.0	4	DS3_v2	14.0	4

Only a sample of VMs are shown here. Each type is represented by N different levels of memory and cores. Availability varies by cloud.

Designing Clusters for High Performance

VM Categories

Memory Optimized

- ML workload with data caching
- If shuffle-spill remains a problem (no other mitigation strategy)
- When spark-caching is a requirement

Compute Optimized

- ETL with full file scans and no data reuse
- Structured Streaming Jobs

General Purpose

- Used in absence of specific requirements

Storage Optimized

- Optimized with Delta IO Caching !!
- ML & DL workloads with data caching
- Data Analysis / Analytics
- If shuffle-spill remains a problem (no other mitigation strategy)
  - Solid State Drives
  - Non-Volatile Memory Express (NVME)
- When spark-caching is a requirement

GPU Optimized

- ML & DL workloads with exceptionally large memory and compute requirements (presumes caching)

Designing Clusters for High Performance
Guessing Compute Level

Experimentation is easy...

- Make a guess
- If you are spilling, assume you need more RAM (unless you have skew)
- If you shuffles are slow, increase VM Level while decreasing the number of VMs

- How many iterations did it take? Increasing the VM Level or number of VMs for more cores
- Is your cluster underutilized? Reduce the VM level or number of VMs for fewer cores
- Expect this processes to take a fair amount of trial and error (aka time, aka money)

Designing Clusters for High Performance
Estimate Compute Level

1. Calculate the data's size on disk
2. **spark.sql.files.maxPartitionBytes?**
3. Compute the number of partitions or cheat and call **df.rdd.getNumPartitions()**
4. Decide which category of VM you want
5. Based on the SLA, quota, and budget, select the type and level of VM
6. Select the number of iterations
7. Compute the number of VMs
8. Adjust, experiment and retest – at least time (and money) is saved by starting with a semi-reasonable configuration

1. Assume we have 100 GB or **102,400 MB**
2. Assume **maxPartitionBytes** is **128 MB**
3. **102,400 MB / 128 MB** = **800 partitions**
4. **Compute Optimized**
5. **Level 5**, **144 GB**, **72 cores** each
6. **2 iterations**
7. **800 par / 72 cores / 2 iterations** = **6 VMs**

DATA+AI SUMMIT 2022

Thank you

Your Name

You Title

databricks