databricks

Performance Optimization with Spark and Delta Lake

©2023 Databricks Inc. — All rights reserved

Agenda

Performance Optimization with Spark and Delta Lake

Lecture: Designing the Foundation

Lecture: Code Optimization

Lecture: Spark Architecture

Follow Along Demo - Spark Simulator

Lecture/Demo: Shuffles

Lecture/Demo: Spill

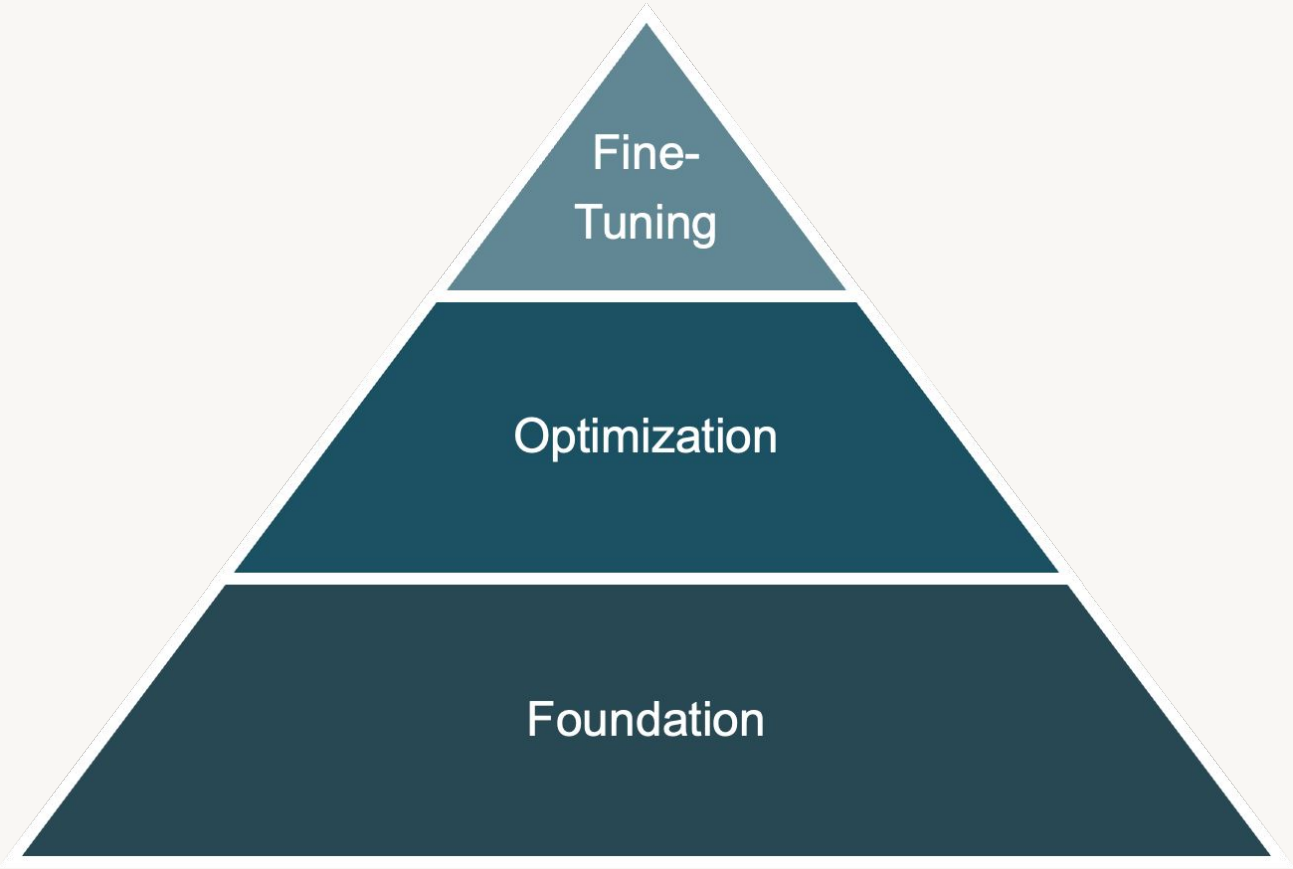
Lecture/Demo: Skew

Lecture/Demo: Serialization

Lecture: Fine-Tuning - Choosing the Right Cluster

©2023 Databricks Inc. — All rights reserved

Building Performant Analytics



Cluster Sizing

Code
Optimization

File Layout

©2023 Databricks Inc. — All rights reserved

Designing the Foundation

©2023 Databricks Inc. — All rights reserved

Fundamental Concepts

Why some schemas and queries perform faster than others

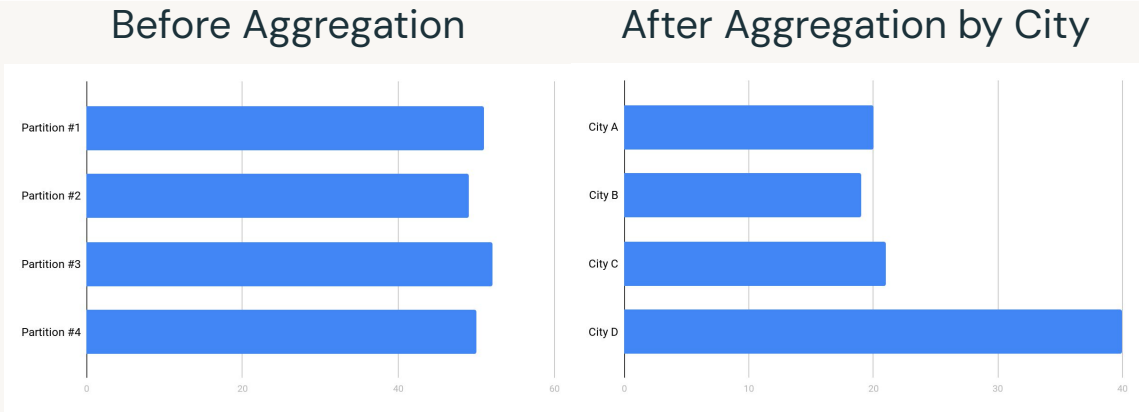
- Number of bytes read
- Query complexity/computation
- Number of files accessed
- Parallelism

©2023 Databricks Inc. — All rights reserved

Common Performance Bottlenecks

Encountered with any big data or MPP system

Bottleneck	Details
Small File Problem	• Listing and metadata operation for too many small files can be expensive • Can also result in throttling from cloud storage I/O limits
Data Skew	• Large amounts of data skew can result in more work handled by a single executor • Even if data read in is not skewed, certain transformations can lead to in-memory skew
Processing More Than Needed	• Traditional data lake platforms often require rewriting entire datasets or partitions
Resource Contention	• Processing large ingestion, ETL jobs at the same time as ad-hoc and BI queries results in slow query performance without cluster isolation

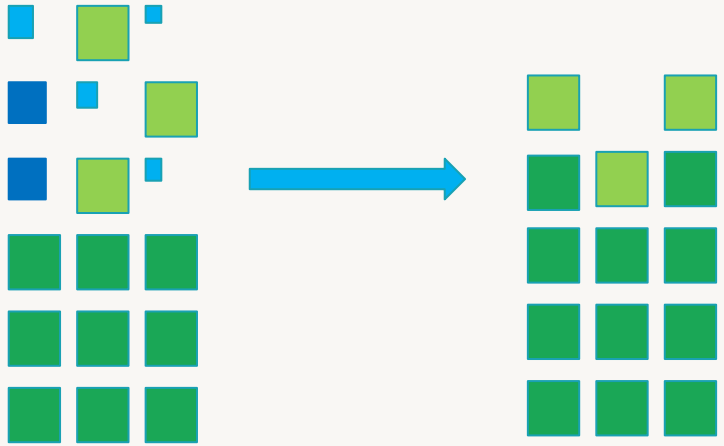


©2023 Databricks Inc. — All rights reserved

Avoiding the Small File Problem

Automatically handle this common performance challenge in Data Lakes

- Too many small files greatly increases overhead for reads
- Too few large files reduces parallelism on reads
- Over-partitioning is a common problem
- Databricks will automatically tune the size of Delta Lake tables
- Databricks can automatically compact small files on write with auto-optimize



©2023 Databricks Inc. — All rights reserved

Data Skipping

Reducing the amount of data read in reduces processing time

- Track file level stats such as min & max to avoid scanning irrelevant files
- File-skipping stats are automatically collected on Delta Lake tables
  - Note: file stats are only collected automatically on the first 32 columns. Make sure the columns frequently used in joins are in the first 32 cols or modify the number of stats collected
- Delta Lake and Z-Order brings this technique known as indexing from RDMS systems to the data lake.
- Unlike traditional sort-based indexing techniques, Z-Ordering uses multi-dimensional clustering for more effective data skipping.

```
SELECT * FROM table WHERE col < 5
```

```
SELECT file_name FROM index
WHERE col_min < 5
```

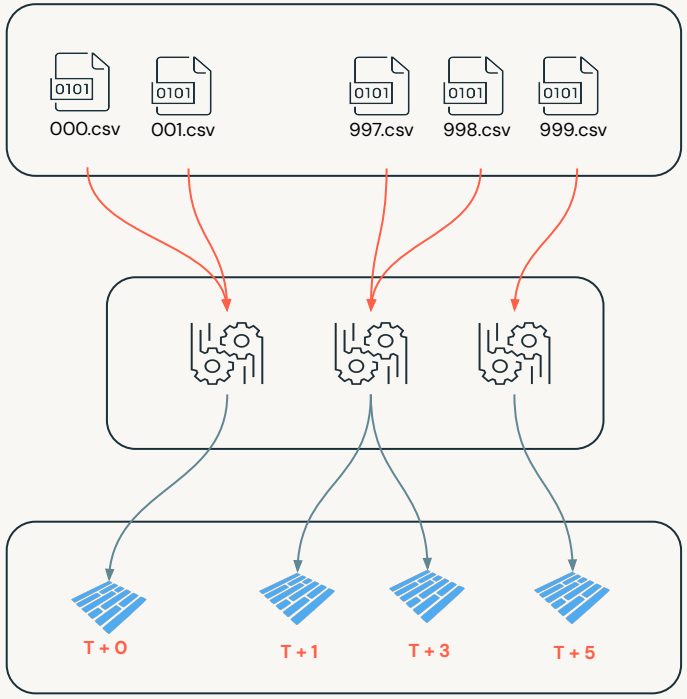
file_name	col_min	col_max
file1.csv	6	8
file2.csv	3	10
file3.csv	1	4

©2023 Databricks Inc. — All rights reserved

Ingestion Time Clustering

Out of the box data skipping with no partitioning or z-order required

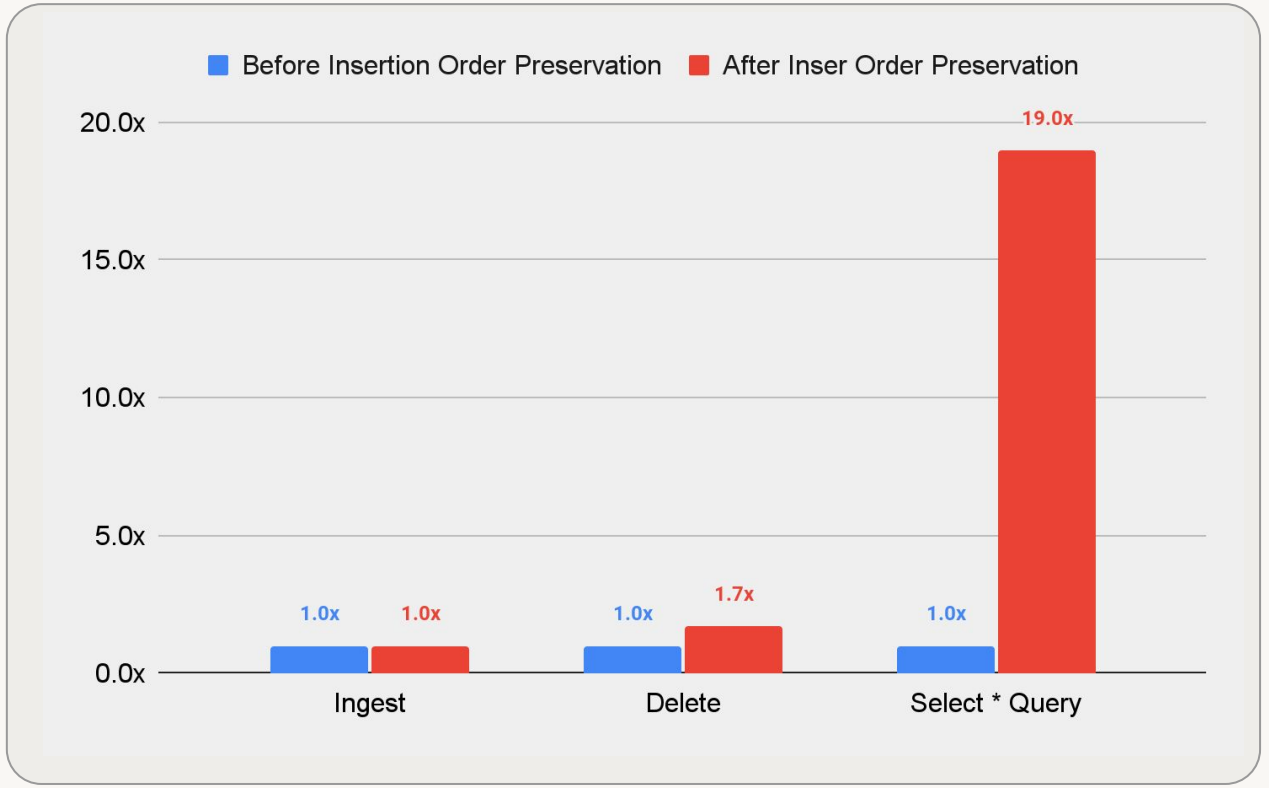
Preserves natural clustering across all Delta operations (DML, ingestion, maintenance)



	AWS	GCP
Status	GA	GA

19x better query performance out of the box

Higher is better



Setup: Store Sales table with data naturally ordered by date

©

Table Statistics

Keeping table statistics to date for best results with Cost Based Optimizer

- Collects statistics on all columns in table
- Helps Adaptive Query Execution
  - Choose proper join type
  - Select correct build side in a hash-join
  - Calibrating the join order in a multi-way join

```
ANALYZE TABLE mytable COMPUTE STATISTICS FOR ALL COLUMNS
```

©2023 Databricks Inc. — All rights reserved

Foundational Recommendations

- Leverage Databricks and Delta Lake to take advantage of auto-tuning:
  - Auto-tuning file size and auto-optimize to avoid small file problem
  - Automatic skew handling with AQE
  - Natural sort order preservation removes need for partitioning tables < 1 TB
- Leverage data skipping with Z-Order and create Z-Order indexes on high cardinality columns frequently used in filters (weekly maintenance job)
- Collect table stats, especially on columns used for joins (weekly maintenance job)
- Use partitioning for data skipping on low cardinality columns frequently used in filters (i.e. year, month, day) - only for tables > 1 TB
- Leverage SQL DML capabilities with Delta Lake to move to a CDC architecture and only process change data.
- Leverage isolated job clusters and SQL warehouses to avoid resource contention

©2023 Databricks Inc. — All rights reserved

Delta Optimizer

- Field managed tool available today to automate foundational optimizations (Z-Order and ANALYZE TABLE)
- Pulls and analyzes the query history + Delta transaction logs and builds a data profile to determine the most important columns that each tables should be Z-ordered by.
- Aims to drastically reduce the amount of manual discovery and tuning users must do to properly optimize their delta tables, especially when the primary query interface is through a DBSQL Warehouse (as an analyst using SQL or a BI tool that auto-generates SQL)

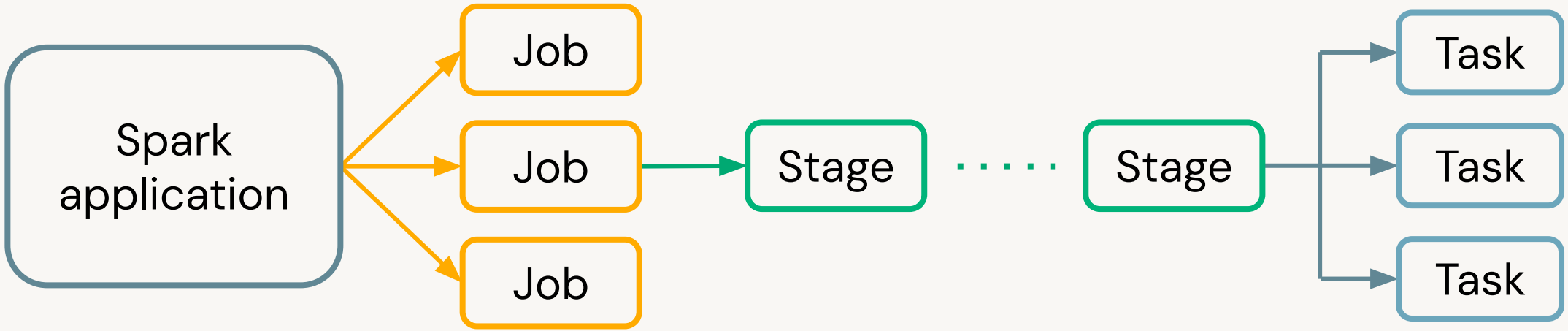
©2023 Databricks Inc. — All rights reserved

Assess and Debug Spark Applications

©2023 Databricks Inc. — All rights reserved

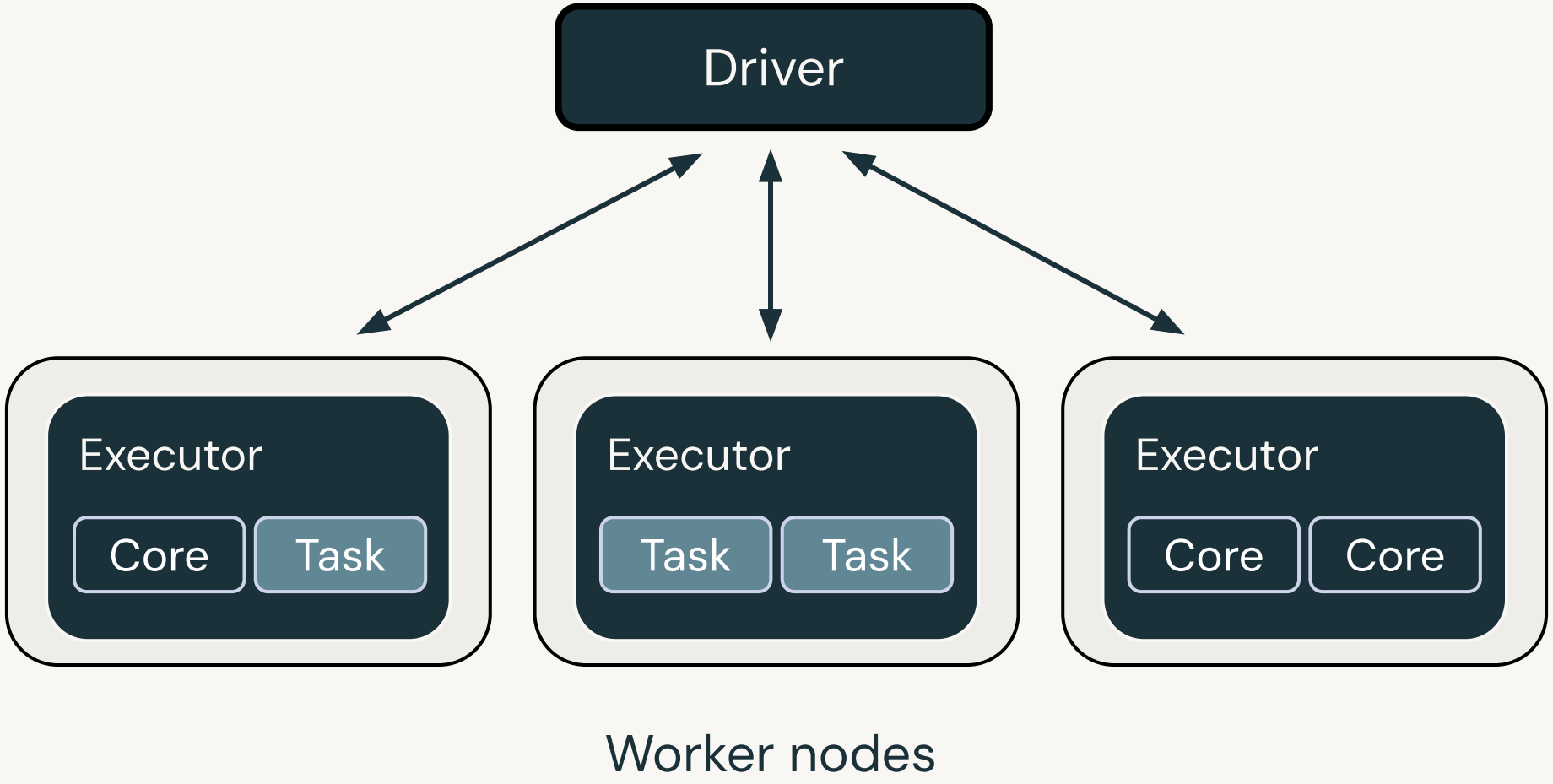
Executing a Spark Application

Data processing tasks run in parallel across a cluster of machines



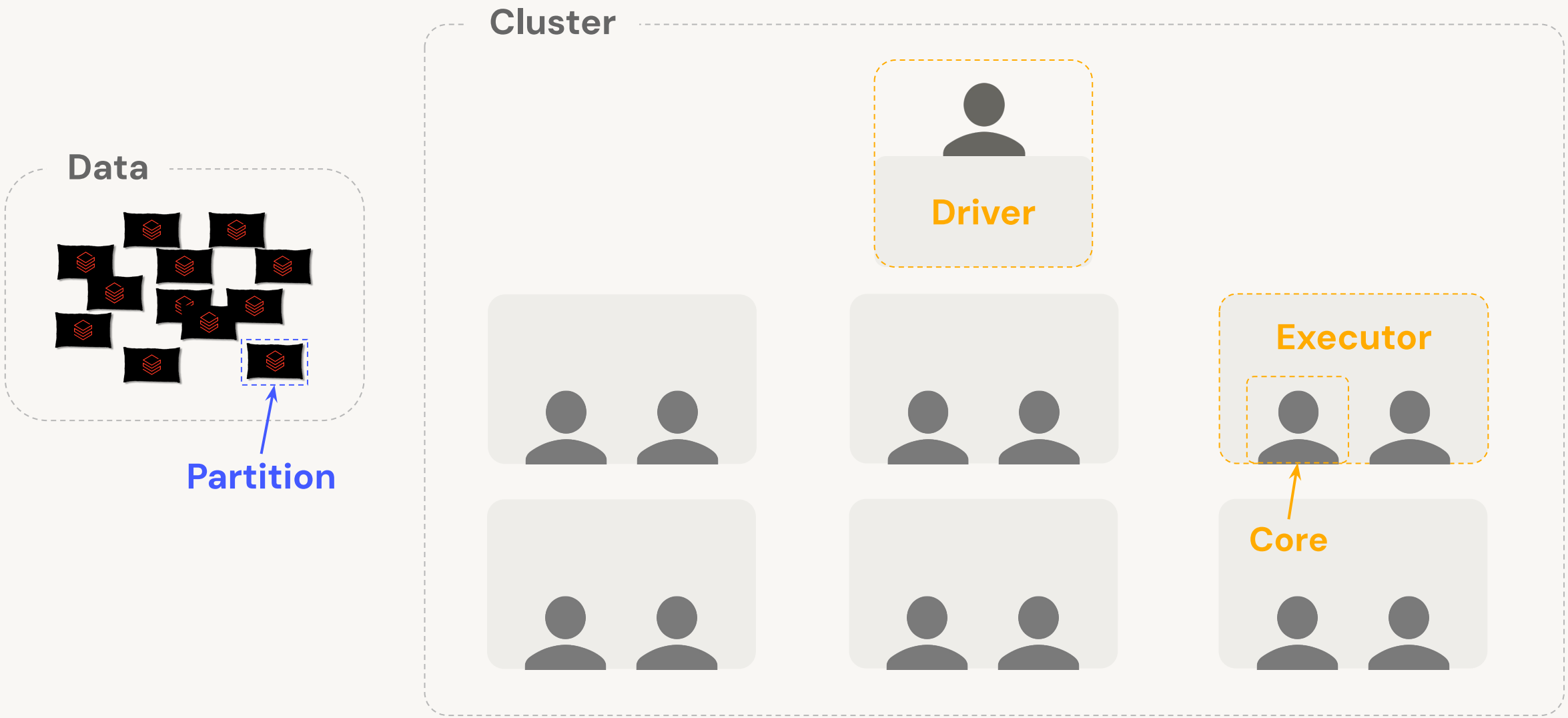
©2022 Databricks Inc. — All rights reserved

Spark Architecture

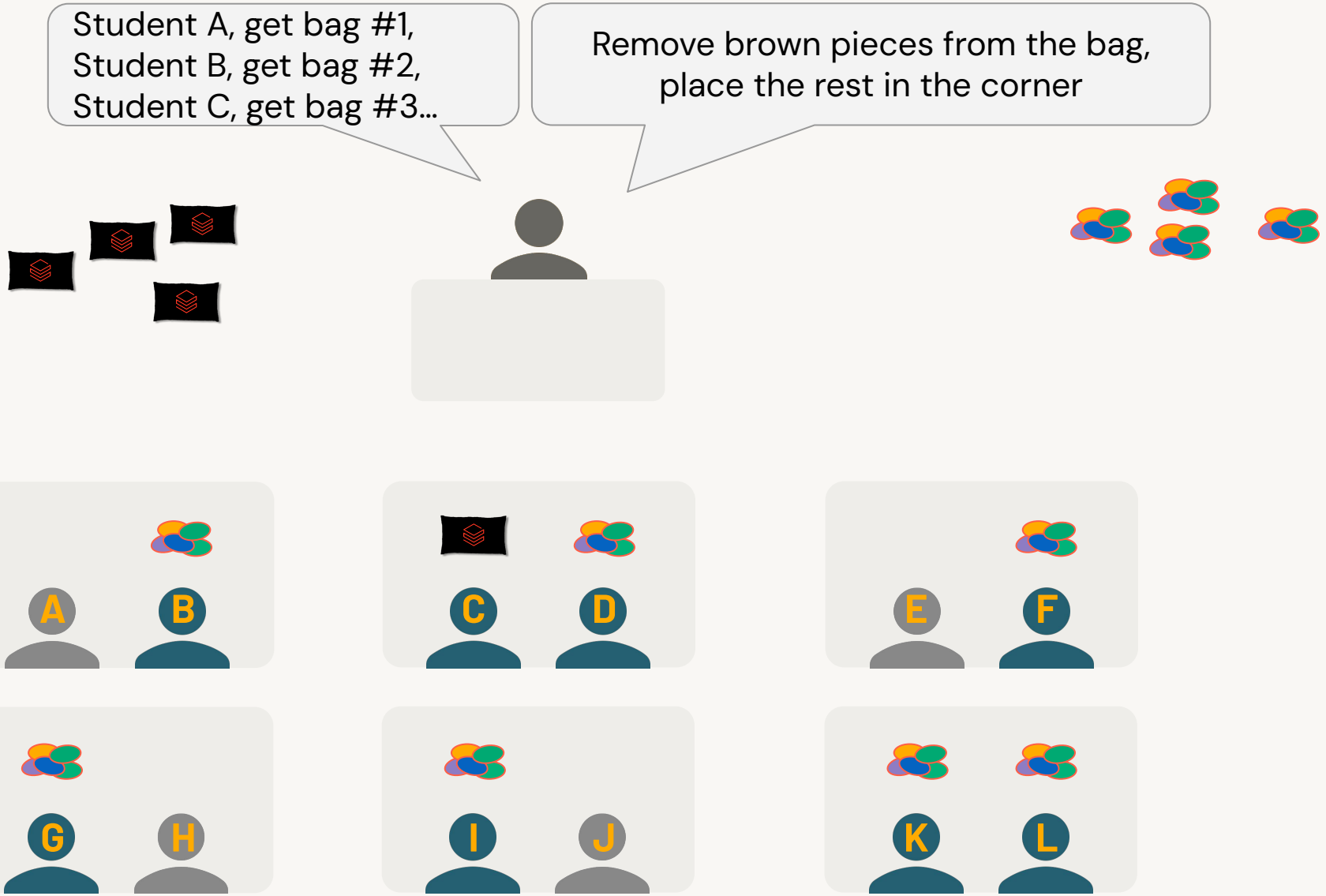


©2023 Databricks Inc. — All rights reserved

Scenario: Filter out brown pieces from these candy bags

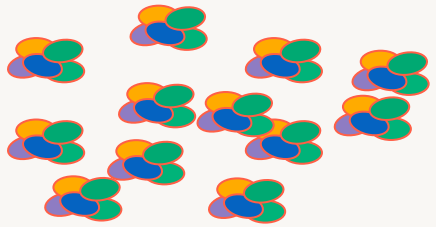


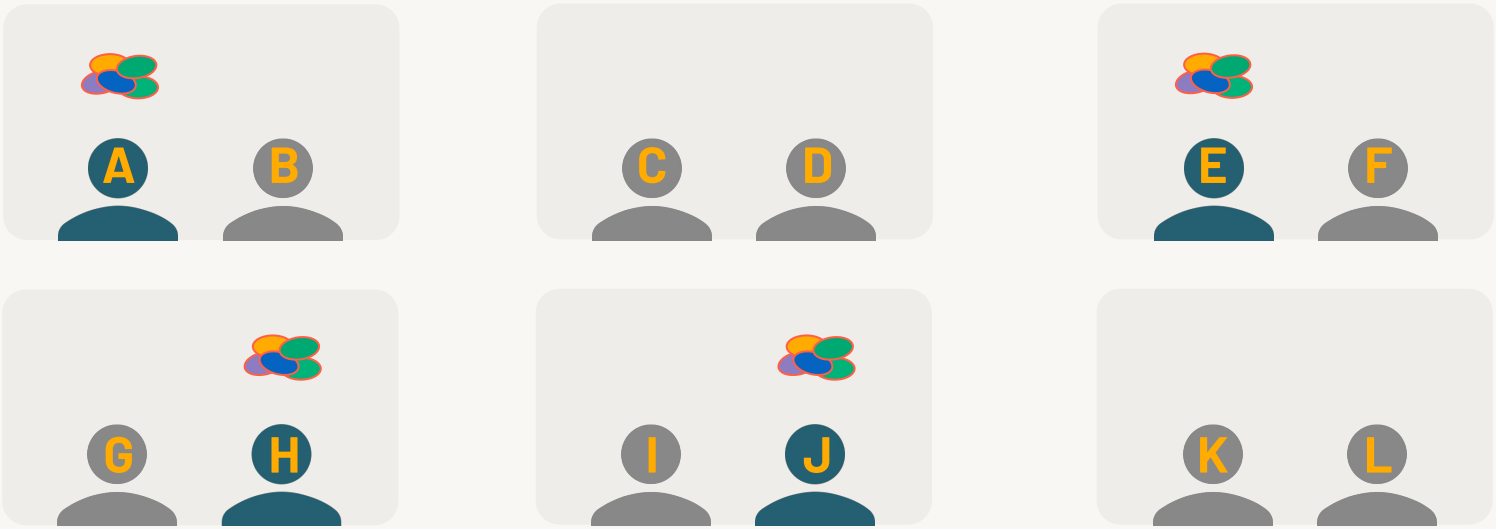
©2023 Databricks Inc. — All rights reserved



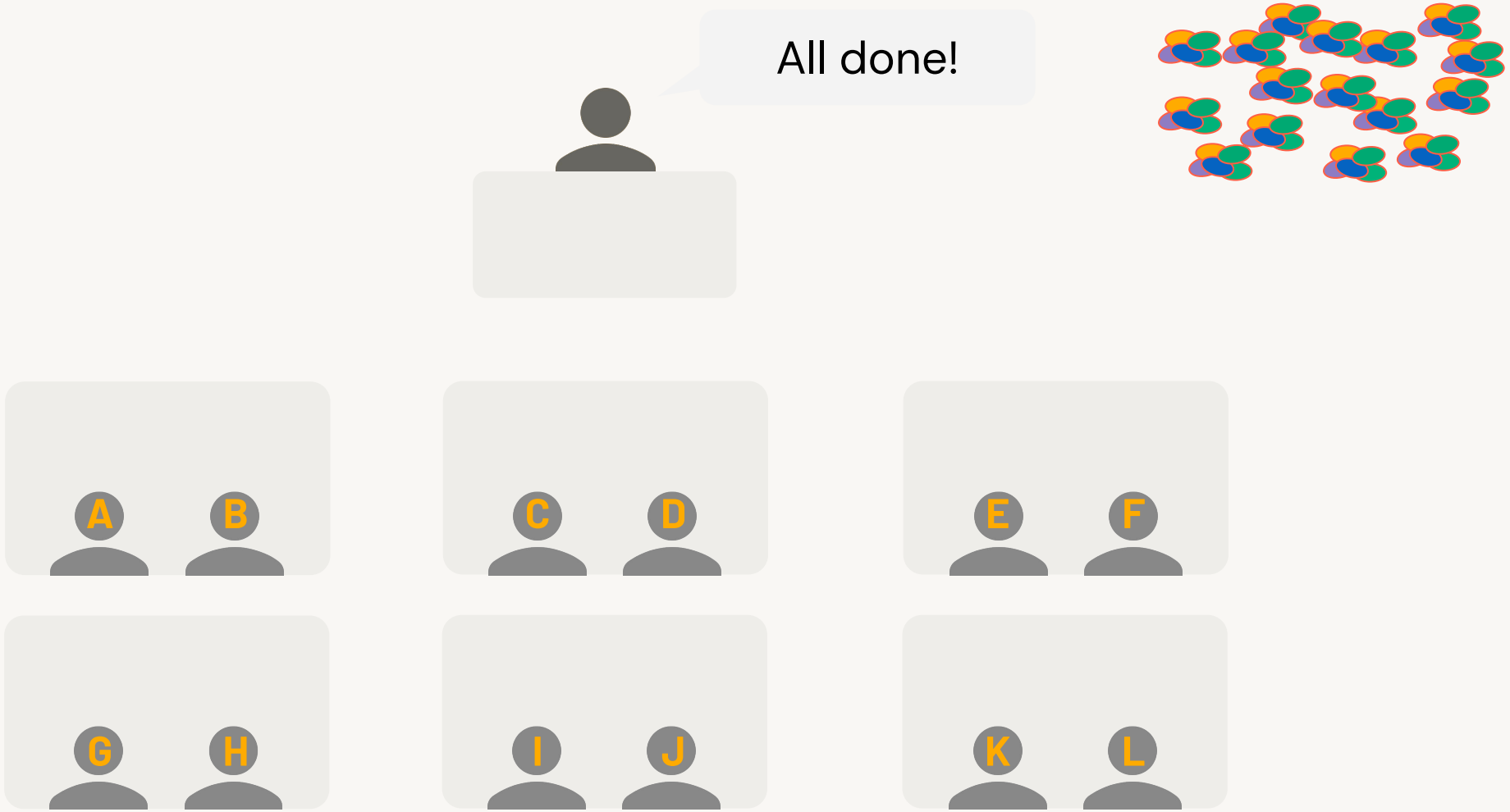
©2023 Databricks Inc. — All rights reserved

Students A, E, H, J,
count bags 13, 14, 15, 16

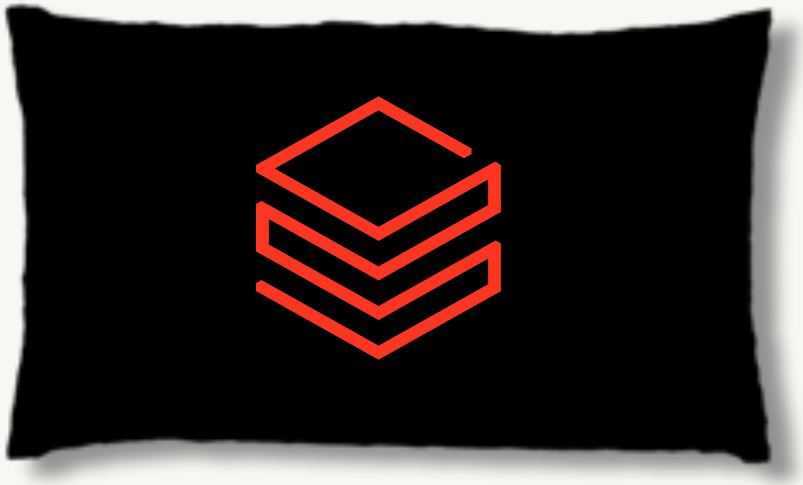




©2023 Databricks Inc. — All rights reserved



©2023 Databricks Inc. — All rights reserved

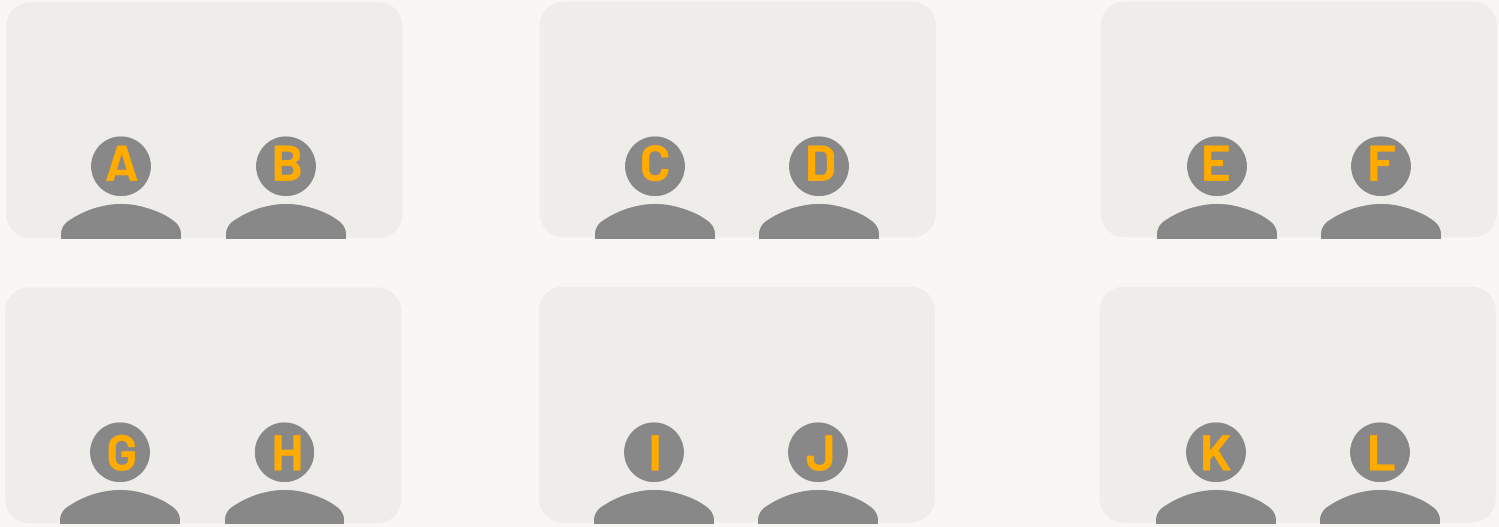


Scenario 2: Count total pieces in candy bags

©2023 Databricks Inc. — All rights reserved

Stage 1: Local Count

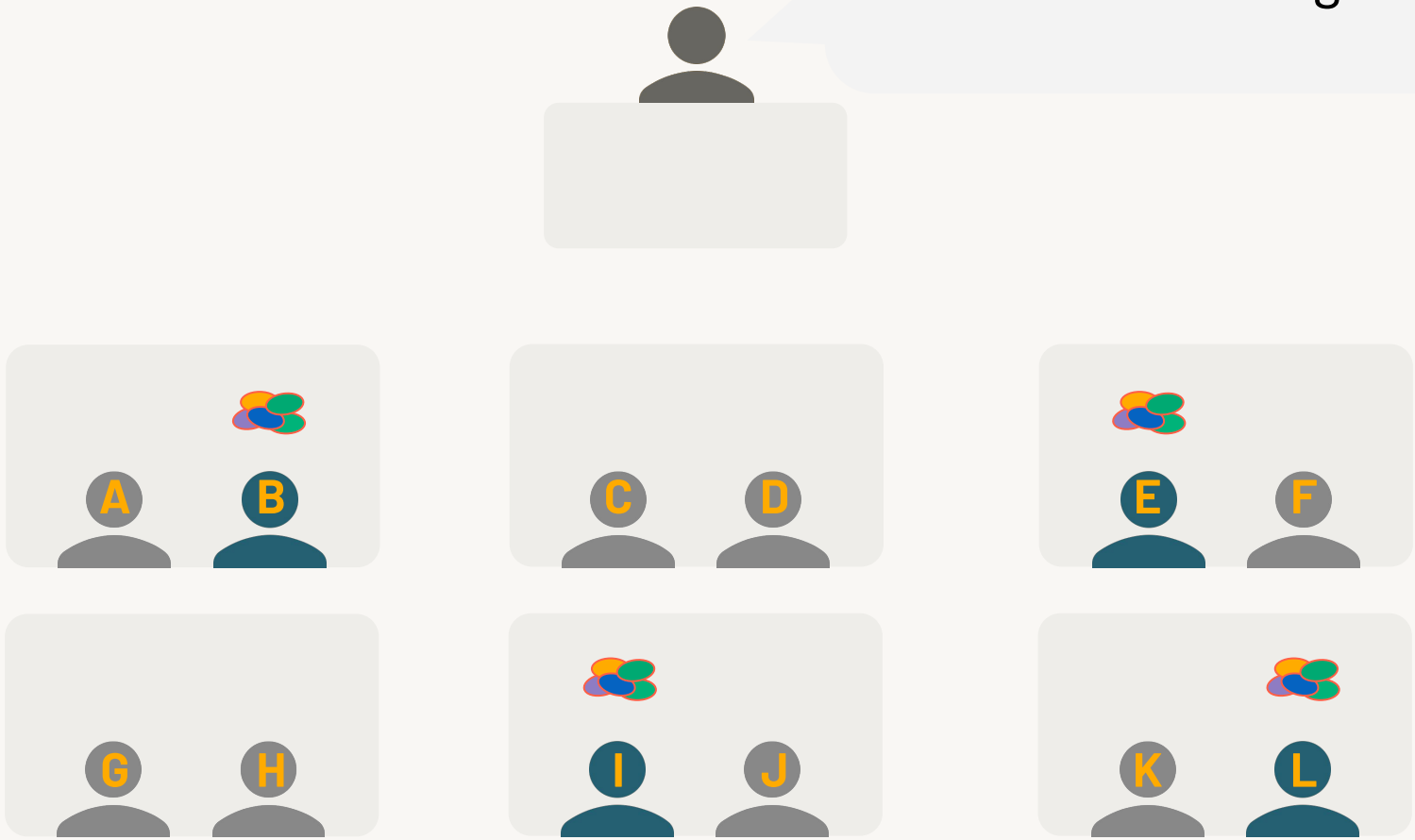
We need to count the total pieces in these candy bags



©2023 Databricks Inc. — All rights reserved

Stage 1: Local Count

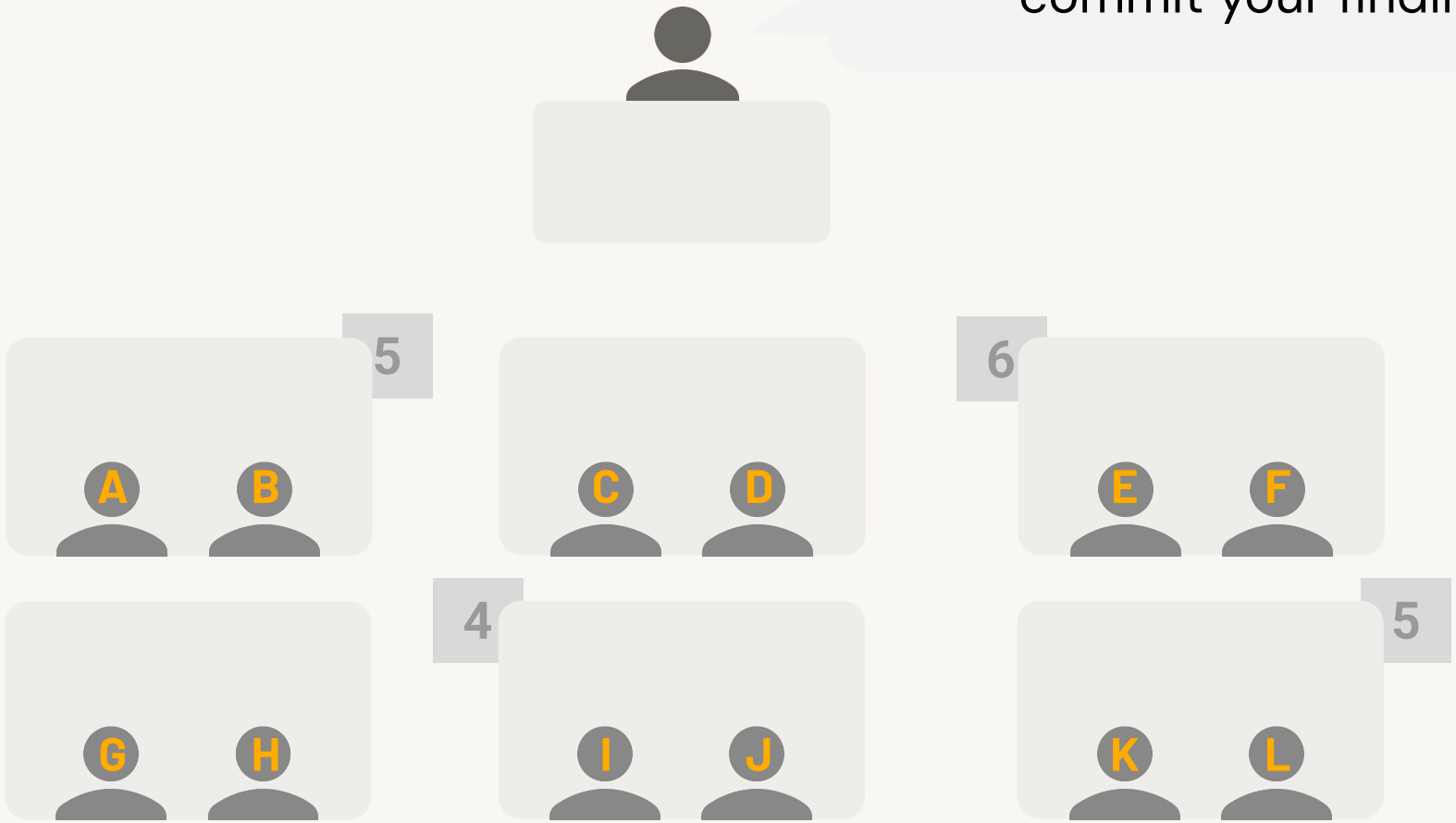
Students B, E, I, L, count these four bags



©2023 Databricks Inc. — All rights reserved

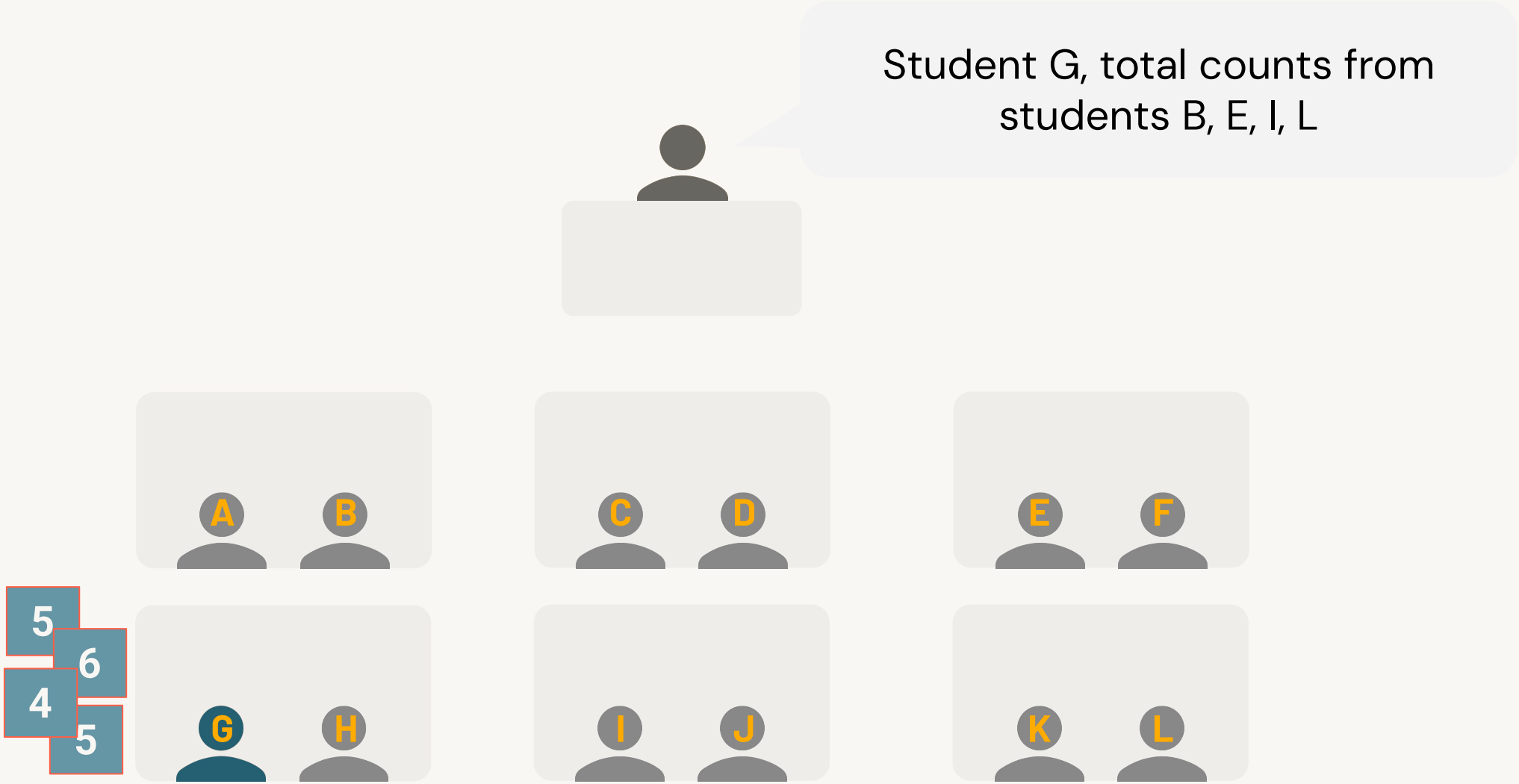
Stage 1: Local Count

Students B, E, I, L,
commit your findings



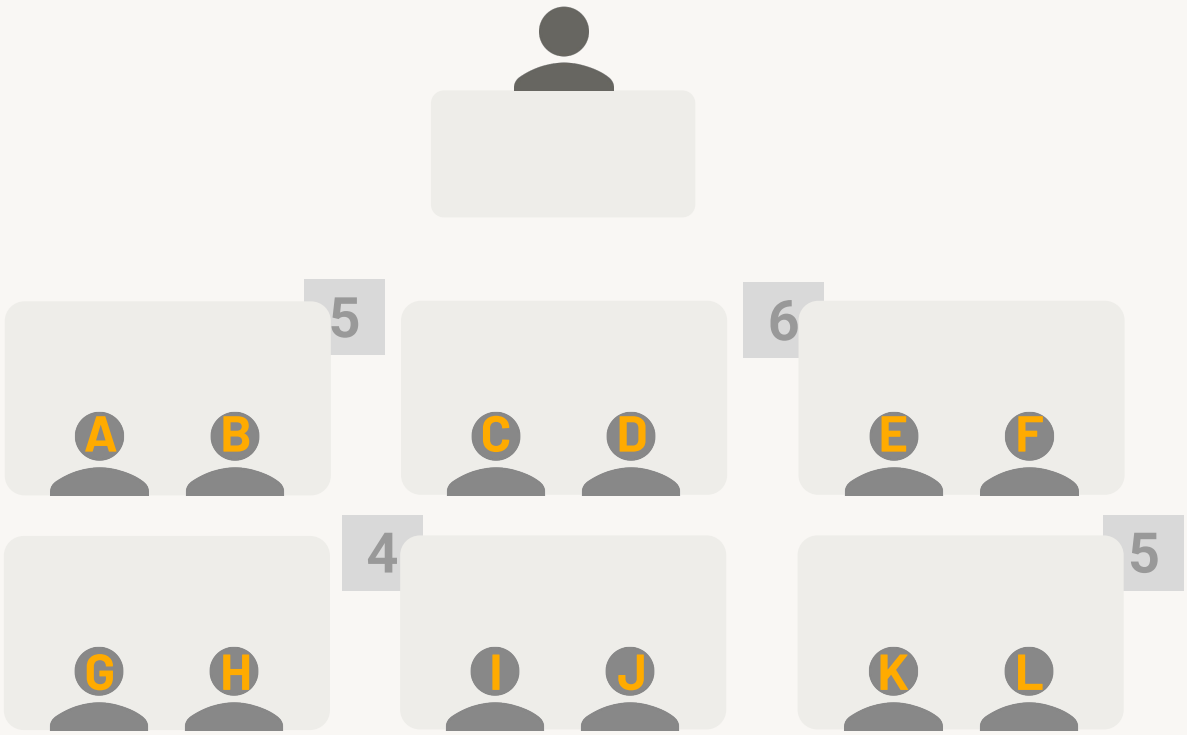
©2023 Databricks Inc. — All rights reserved

Stage 2: Global Count

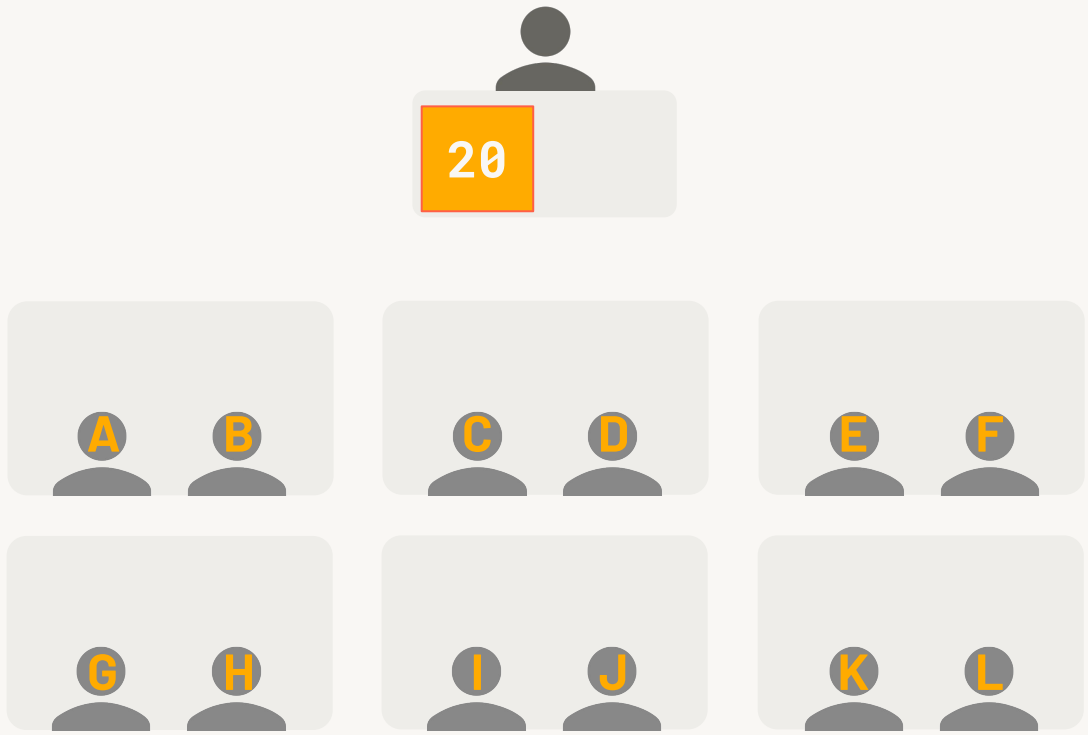


©2023 Databricks Inc. — All rights reserved

Stage 1: Local Count

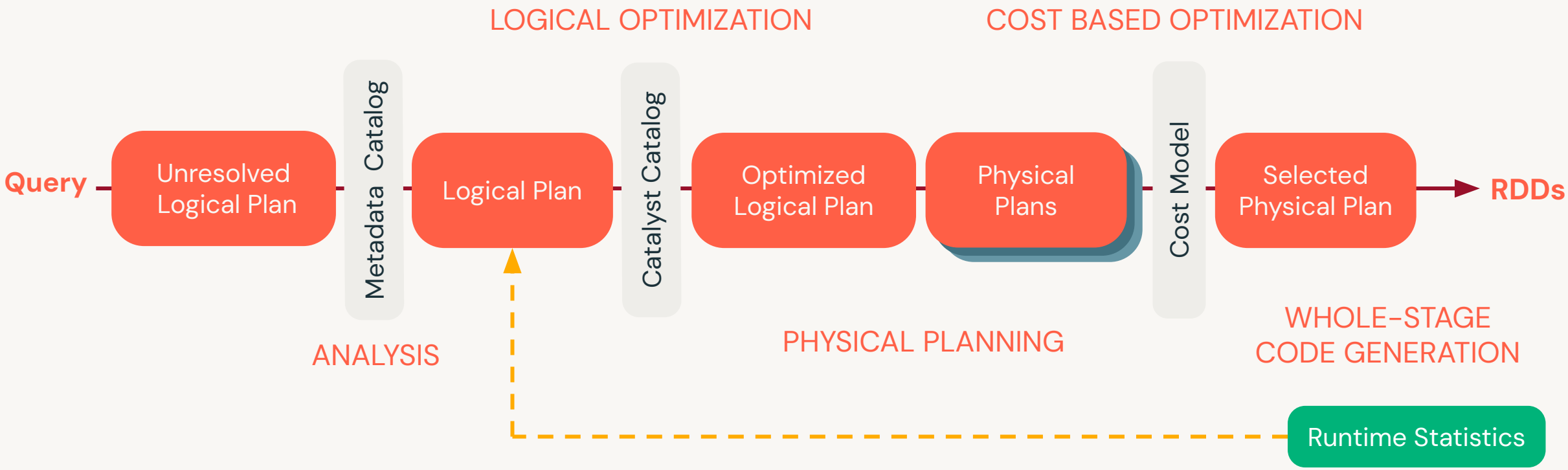


Stage 2: Global Count



©2023 Databricks Inc. — All rights reserved

Query Optimization



©2023 Databricks Inc. — All rights reserved

Code Optimization Recommendations

1. In production jobs, avoid operations that trigger an action besides reading and writing files. These include count(), display(), collect().
2. Avoid operations that will force all computation into the driver node such as using single threaded python/pandas. Use Pandas API on Spark instead to distribute pandas functions.
3. Avoid python UDFs which execute row-by-row. Instead use native pyspark functions or Pandas UDFs for vectorized UDFs.
4. Use Dataframes or Datasets instead of RDDs. RDDs cannot take advantage of the cost-based optimizer.

©2023 Databricks Inc. — All rights reserved

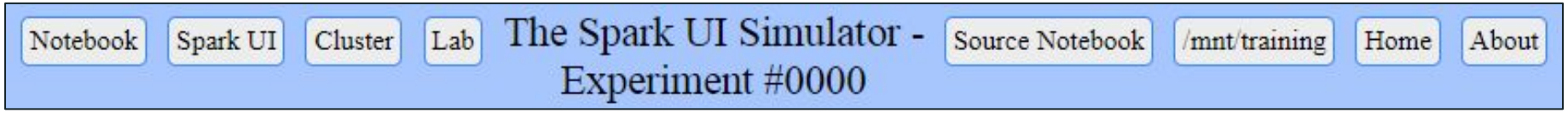
Spark UI Simulator

©2023 Databricks Inc. — All rights reserved

The Spark UI Simulator

https://www.databricks.training/spark-ui-simulator

- Notebook — All code
- Spark UI — Jobs / Stages / Storage / Environment / Executors / SQL tabs
- Cluster — Driver, Worker, Software version
- Lab — Online quiz
- Source Notebook — Export notebook to import into a workspace
- /mnt/training — How to edit paths to point to dataset files
- Home — Return to all experiments
- About — General information



©2023 Databricks Inc. — All rights reserved

Shuffles

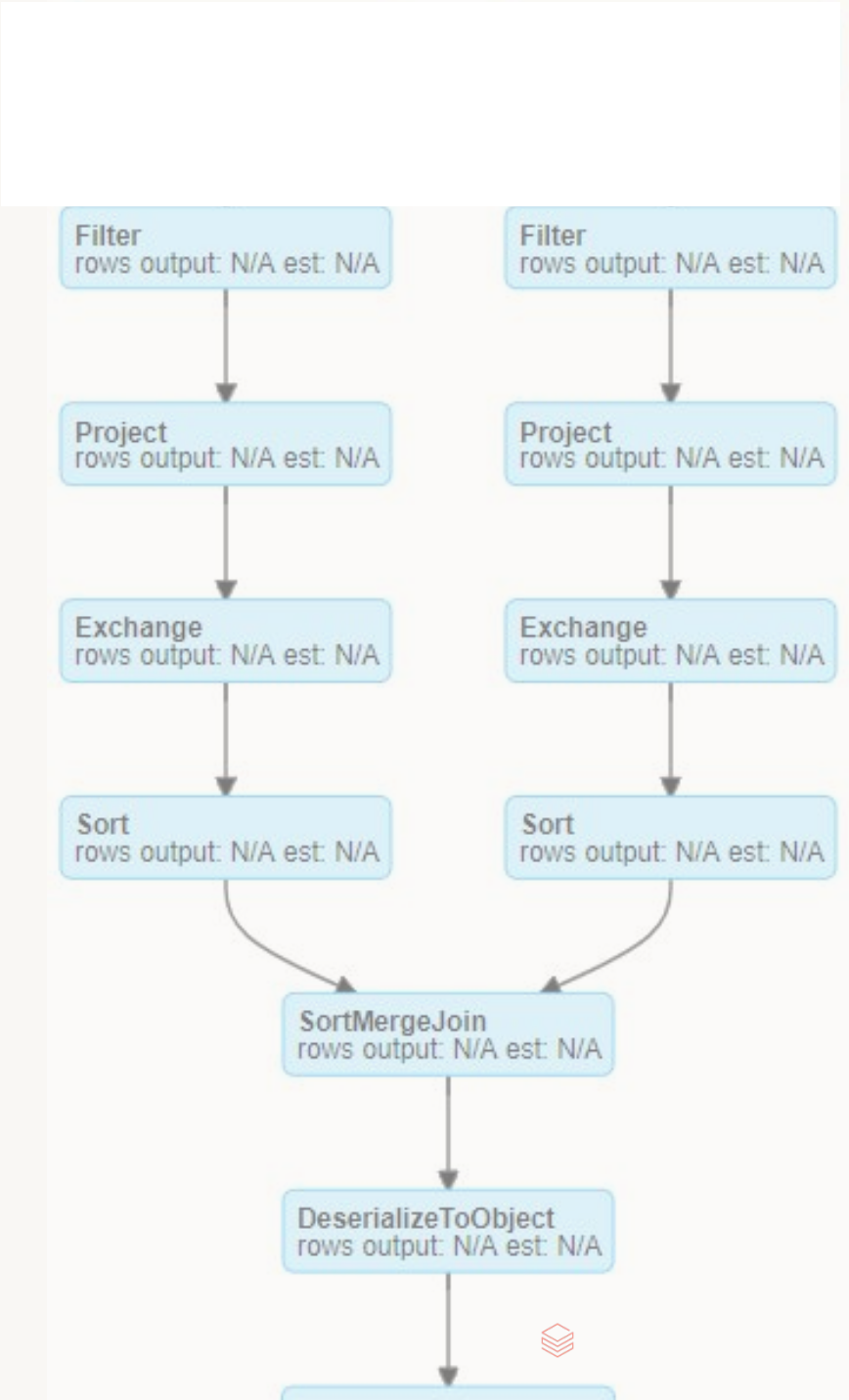
©2023 Databricks Inc. — All rights reserved

Shuffles

Shuffling is a side effect of **wide transformations**

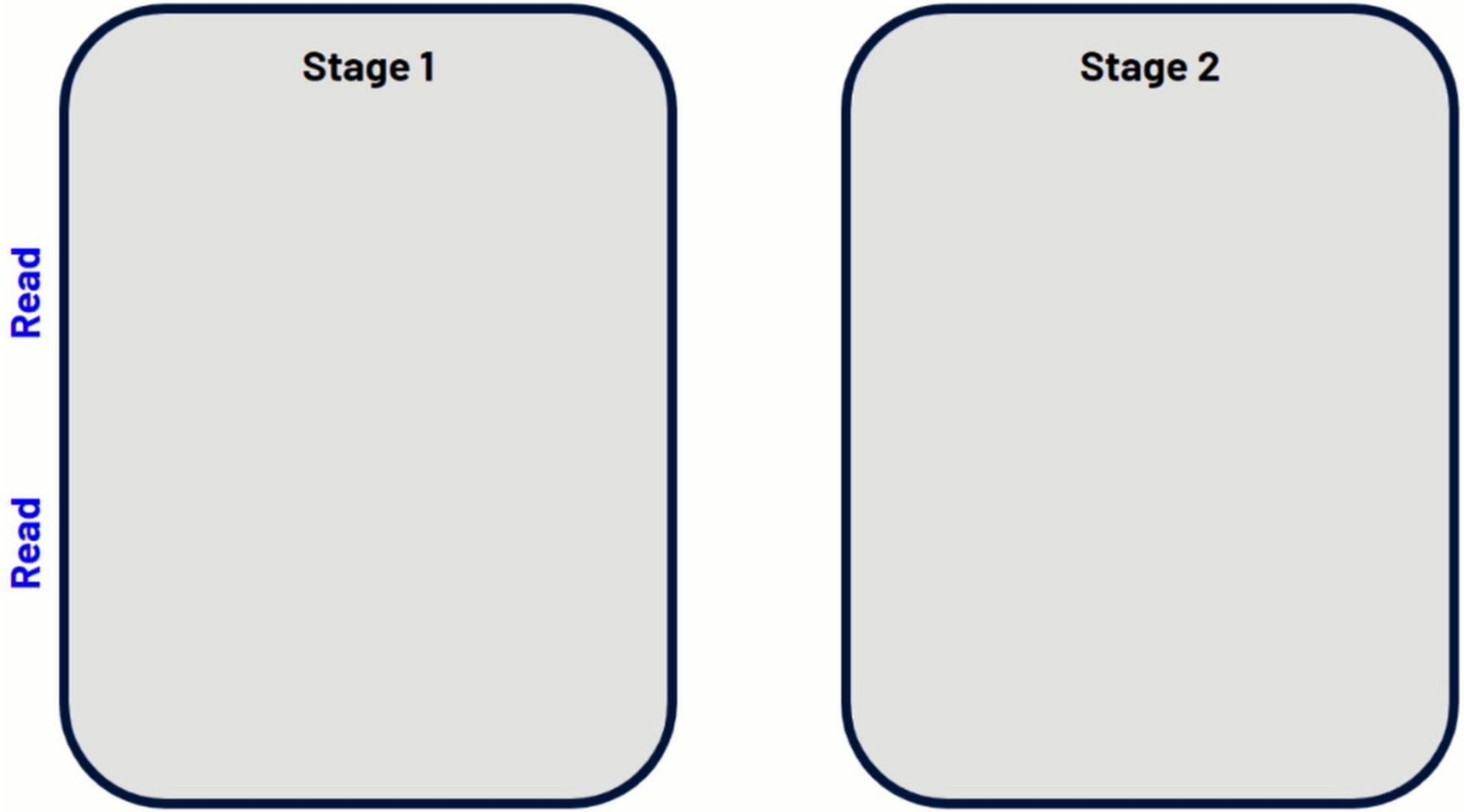
- `join()`
- `distinct()`
- `groupBy()`
- `orderBy()`

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



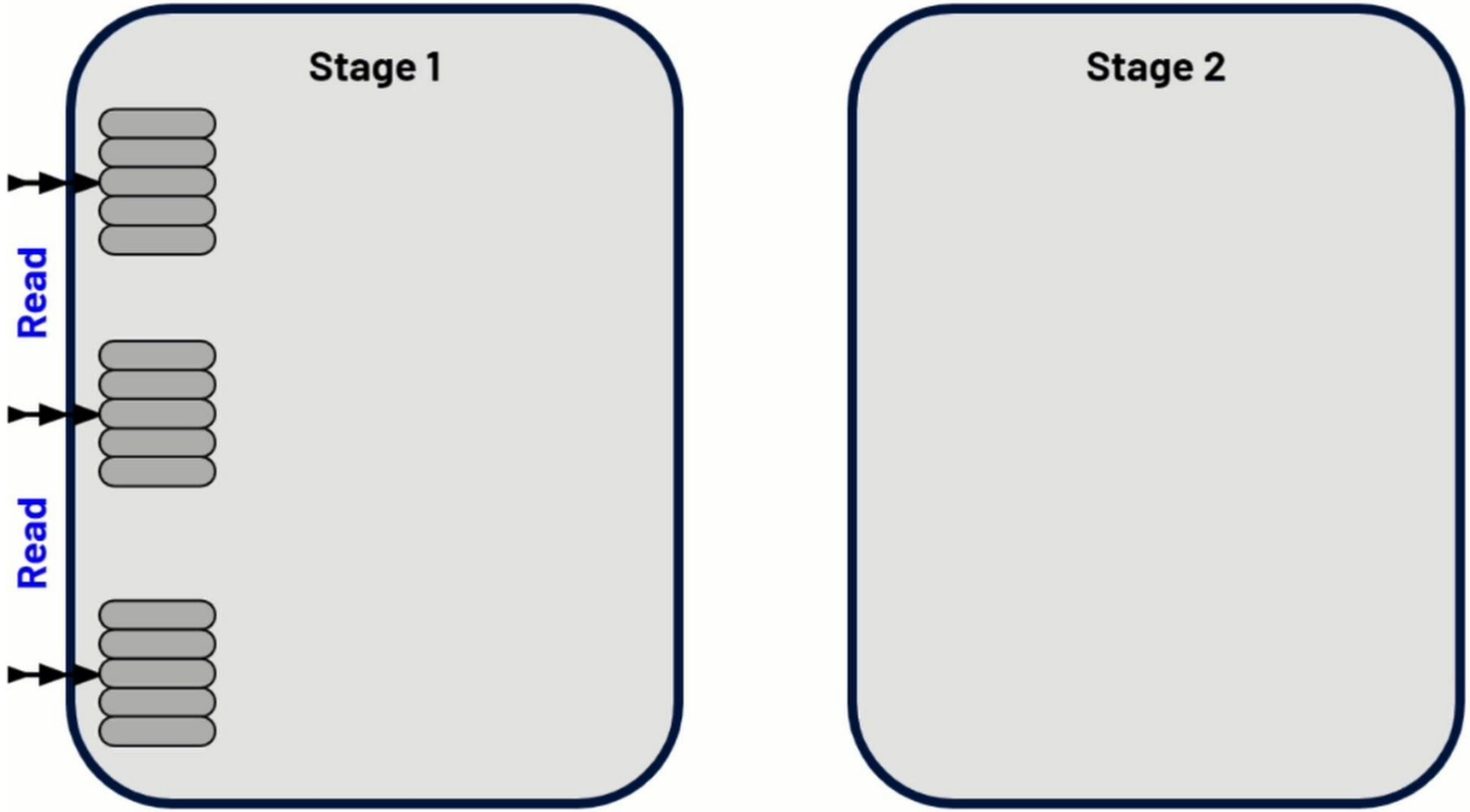
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



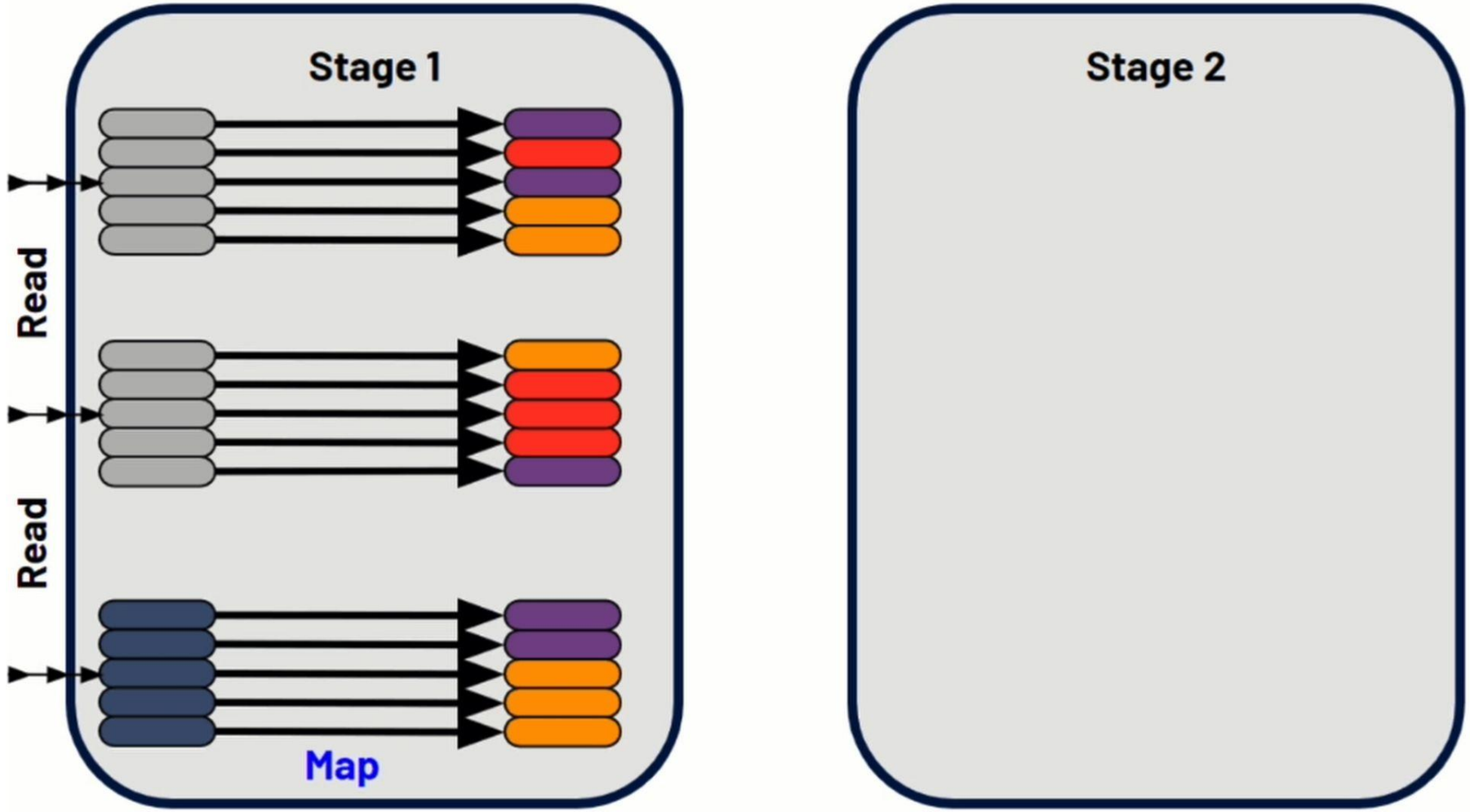
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



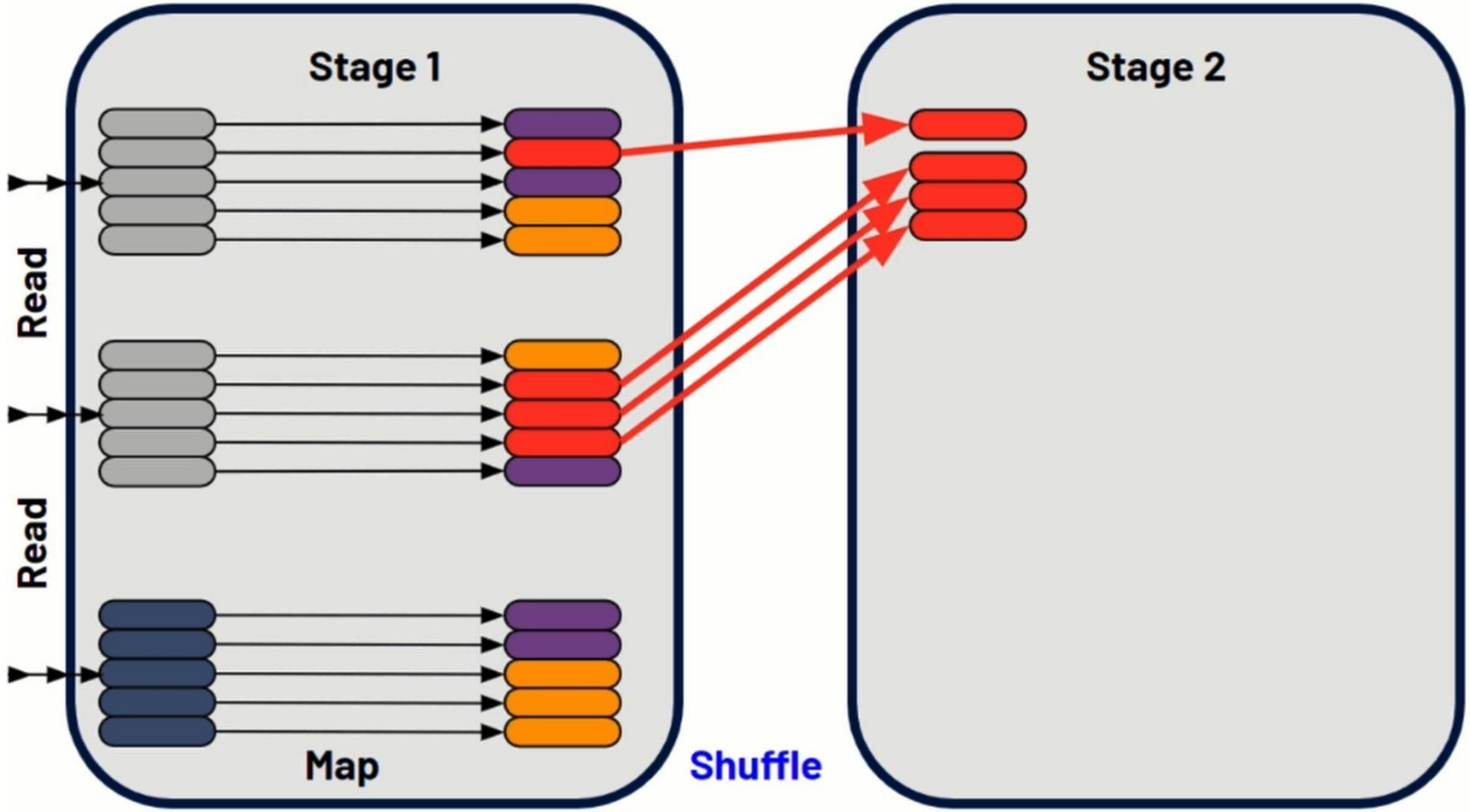
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



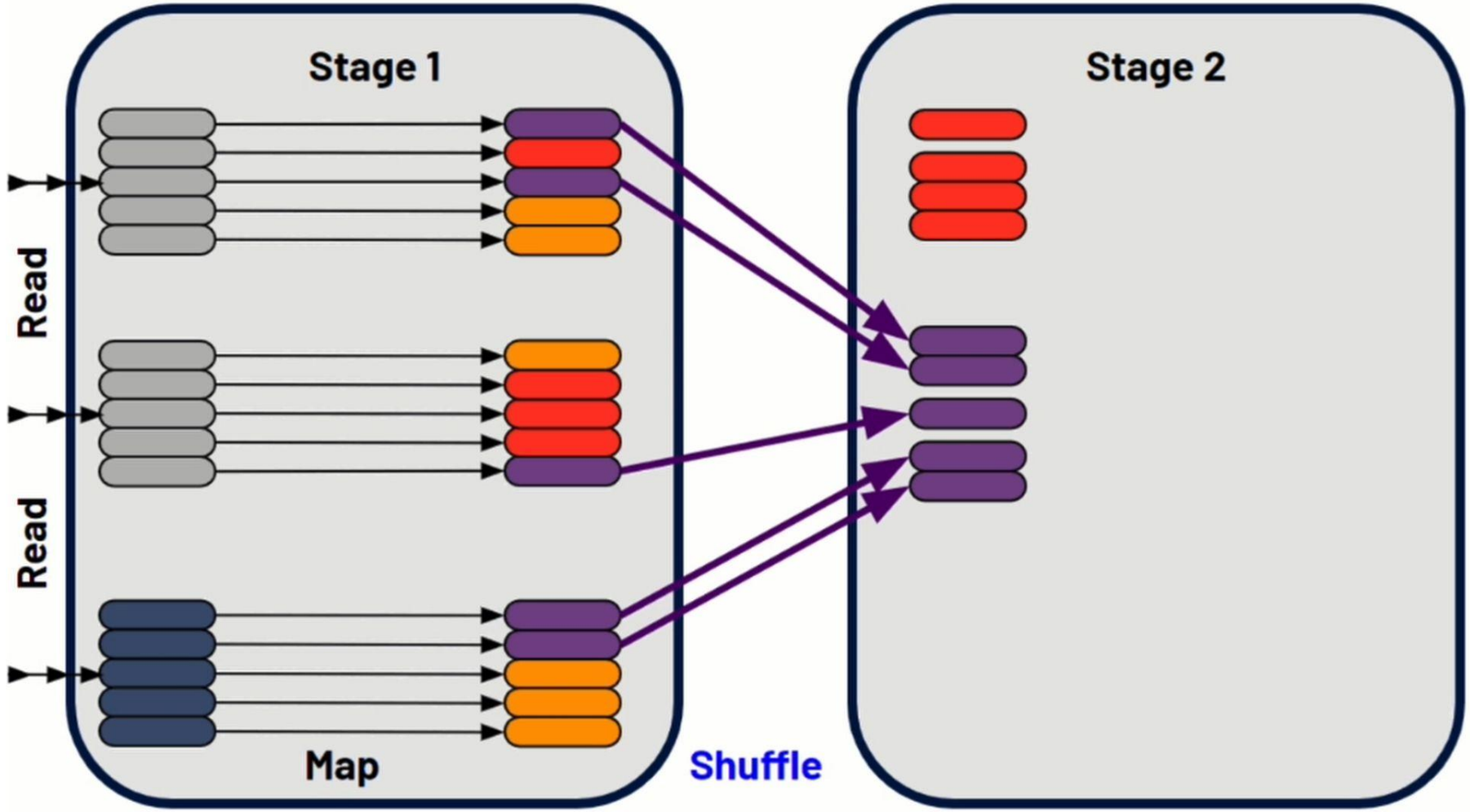
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



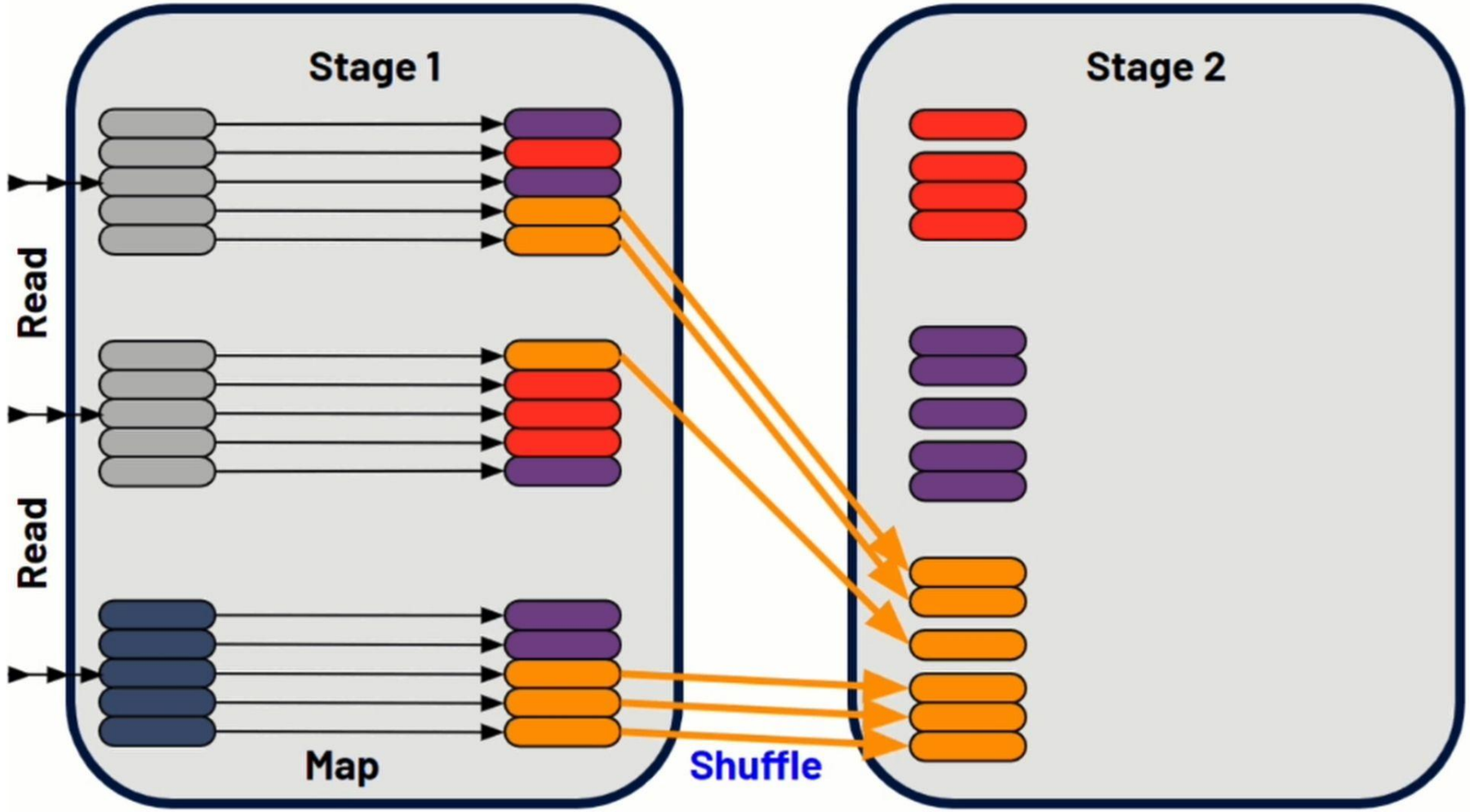
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



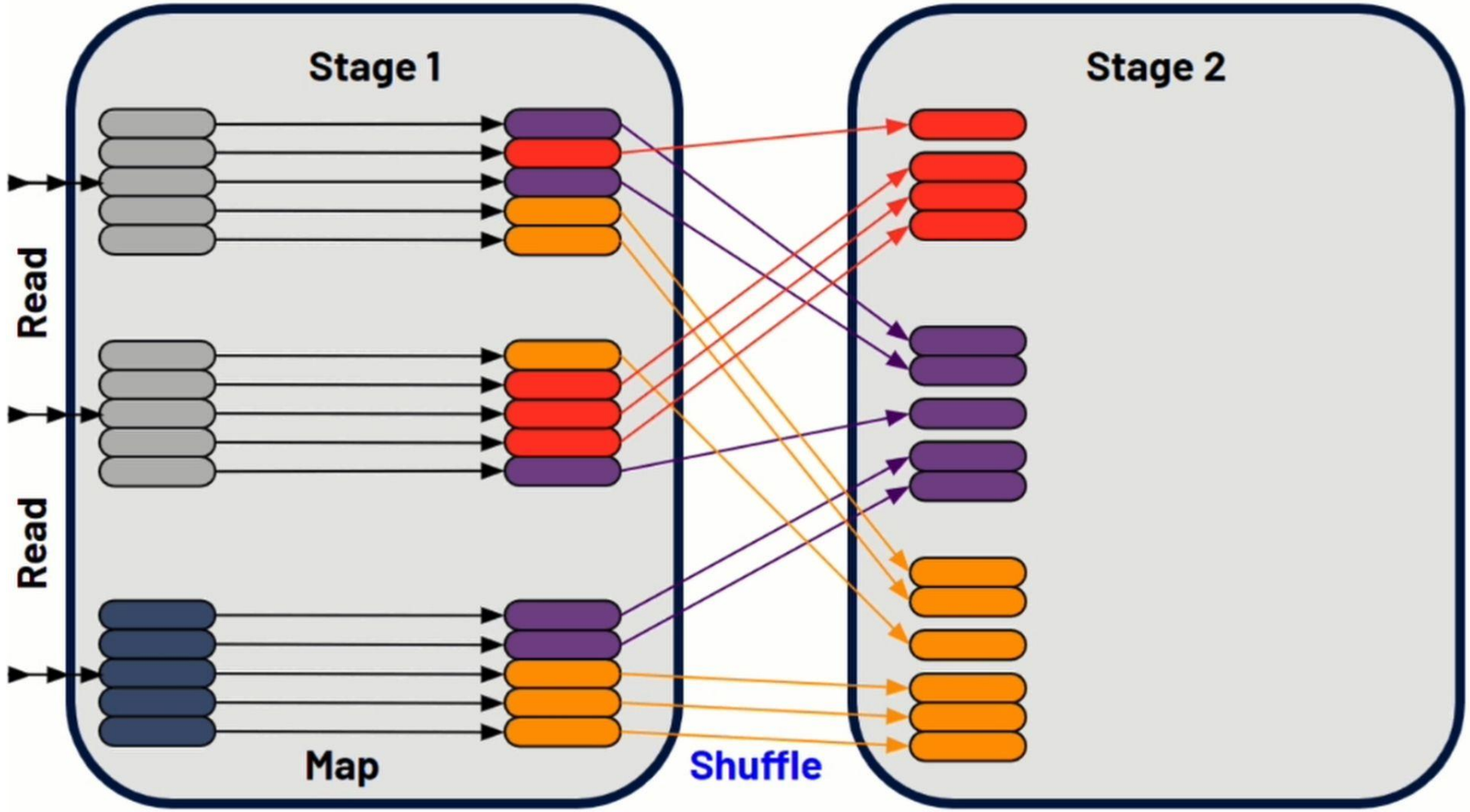
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



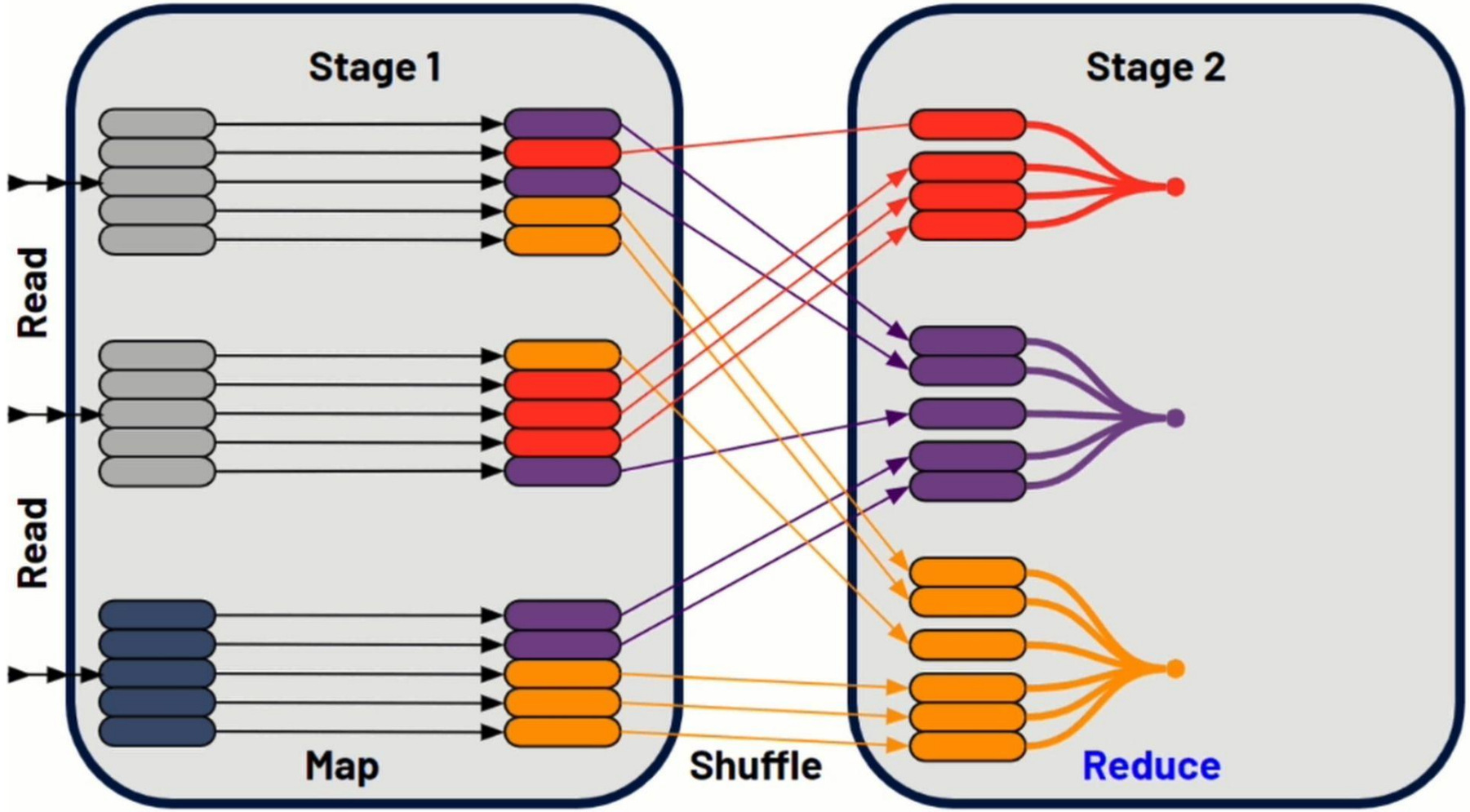
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



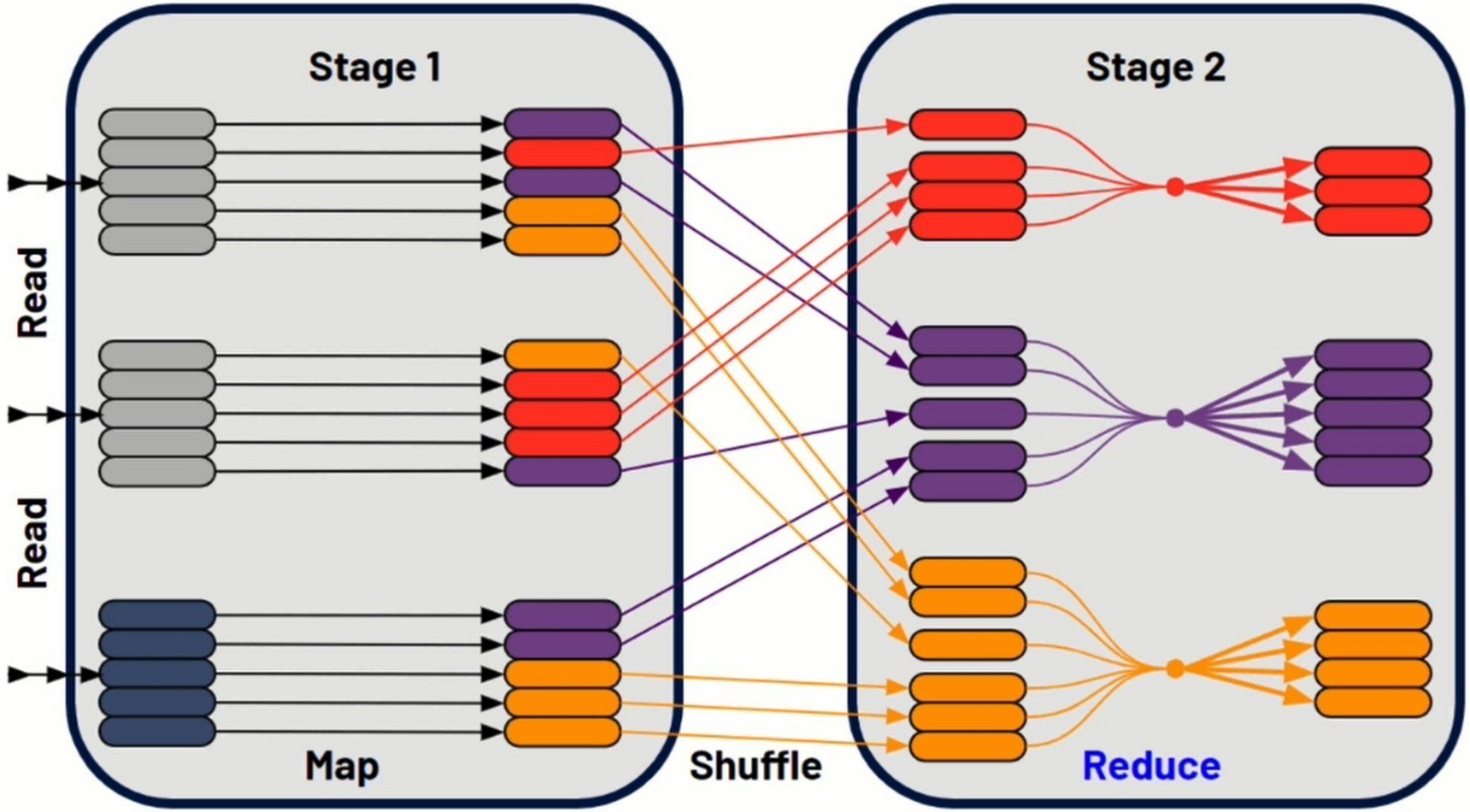
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



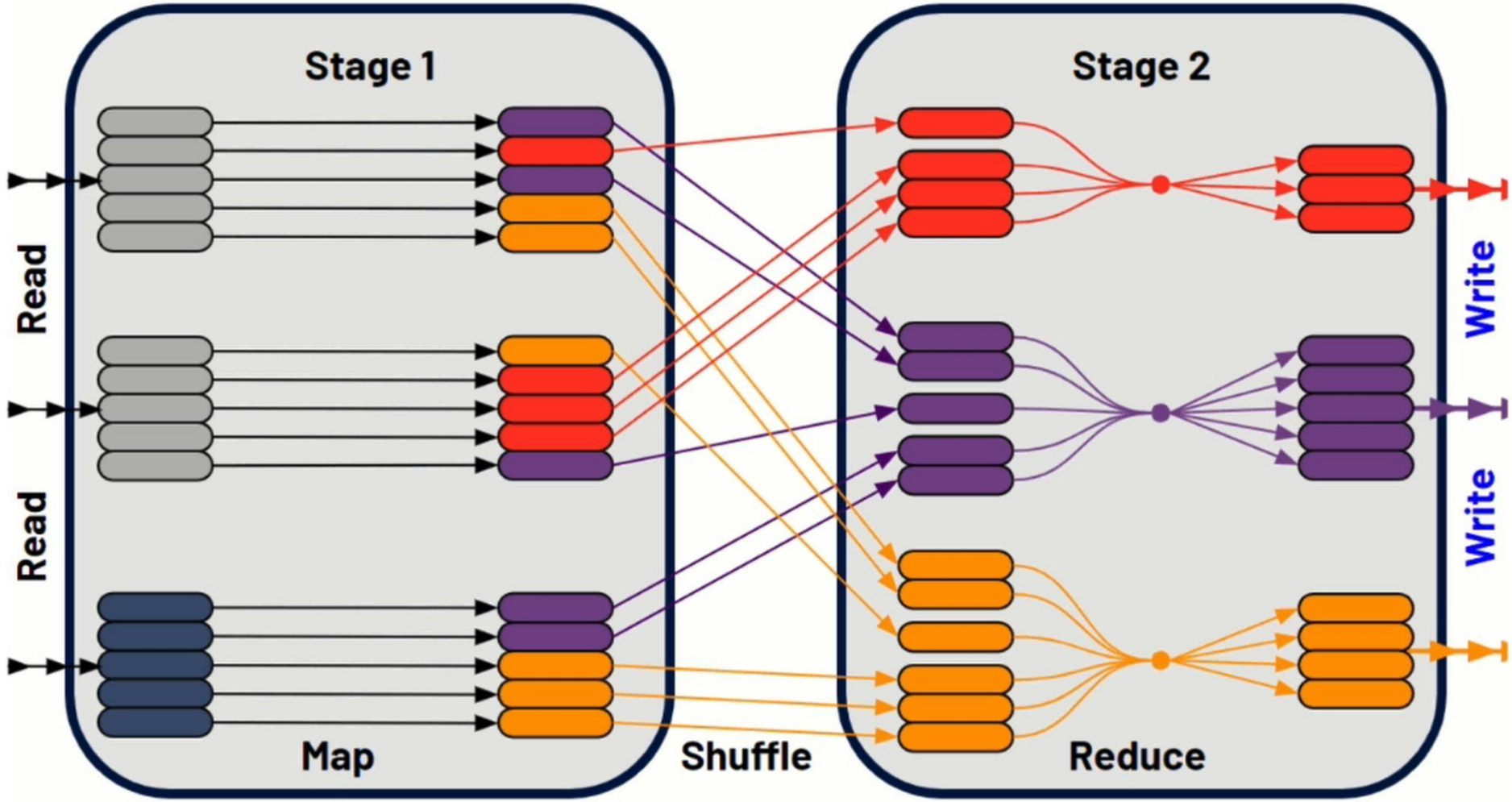
©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



©2023 Databricks Inc. — All rights reserved

Shuffles At A Glance



©2023 Databricks Inc. — All rights reserved

Shuffles – Mitigation

- Reduce network IO by using fewer, larger workers
- Speed up shuffle reads & writes by using NVMe & SSDs
- Reduce amount of shuffled data
  - Remove unnecessary columns
  - Filter out unnecessary records preemptively
- Denormalize datasets, esp when shuffle is rooted in a join

Reevaluate join strategy:

- Reordering the join
- Bucketing
- Broadcast Hash Join
- Shuffle Hash Joins (default for Databricks Photon)
- Sort-Merge Join (default for OS Spark)

©2023 Databricks Inc. — All rights reserved

Spark Join Strategies

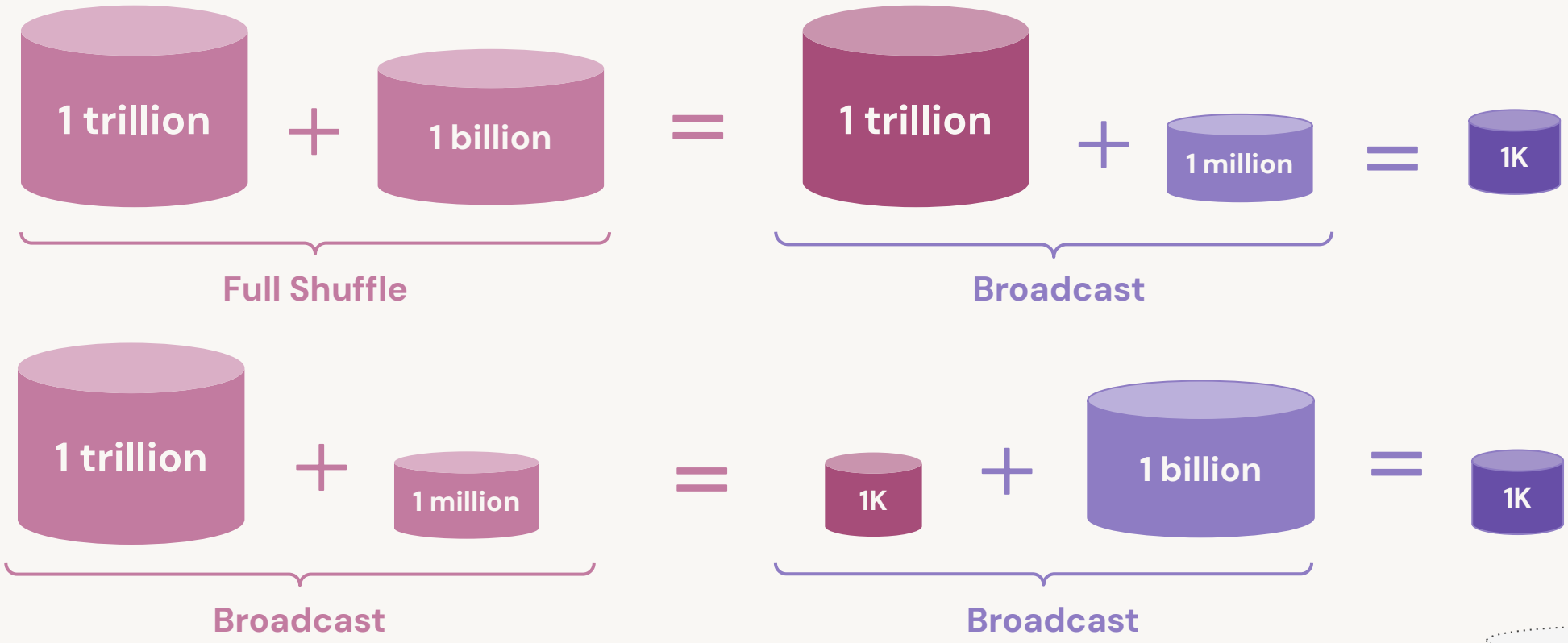
Spark Join Strategies are named after their associated distribution and join strategies

Distribution Strategy	Join Type	Join Strategy Name
Broadcast	Hash Join	**Broadcast-Hash Join**
Shuffle	Hash Join	**Shuffle-Hash Join**
Shuffle	Sort Merge Join	**Shuffle-Sort Merge Join**
Broadcast	Nested Loop	**Broadcast-Nested Loop Join**

©2023 Databricks Inc. — All rights reserved

Optimizing Joins: Reordering

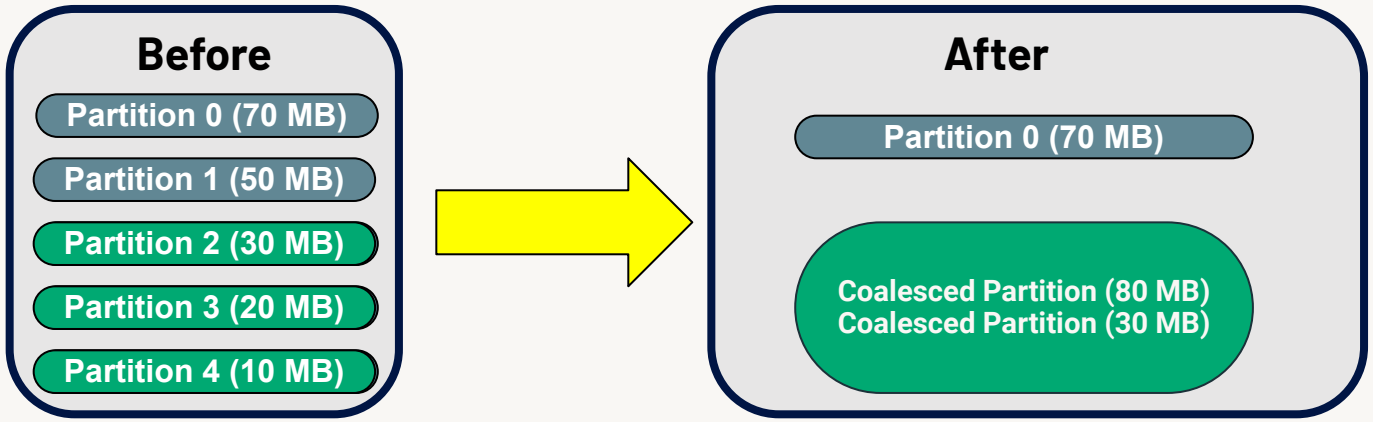
Reduce records per shuffle (mostly automatic w/ AQE, CBO)



©2023 Databricks Inc. — All rights reserved

AQE – Tuning Shuffle Partitions

Net effect is fewer partitions for subsequent stages



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

©2023 Databricks Inc. — All rights reserved

AQE – Tuning Shuffle Partitions

See Experiment #2653

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**

©2023 Databricks Inc. — All rights reserved

Spill

©2023 Databricks Inc. — All rights reserved

Spill

- Spill is the term used to refer to the act of moving data 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

©2023 Databricks Inc. — All rights reserved

Spill - Examples

- Set **spark.sql.files.maxPartitionBytes** too high (default is 128 MB)
- The **explode()** of even a small array
- The **join()** or **crossJoin()** of two tables which generates lots of new rows
- The **join()** or **crossJoin()** of two tables by a skewed key
- The **groupBy()** where the column has low cardinality
- The **countDistinct()** and **size(collect_set())**
- Setting **spark.sql.shuffle.partitions** too low or wrong use of **repartition()**

©2023 Databricks Inc. — All rights reserved

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

©2023 Databricks Inc. — All rights reserved

Spill Listener – 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

See Experiment #6518

- 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

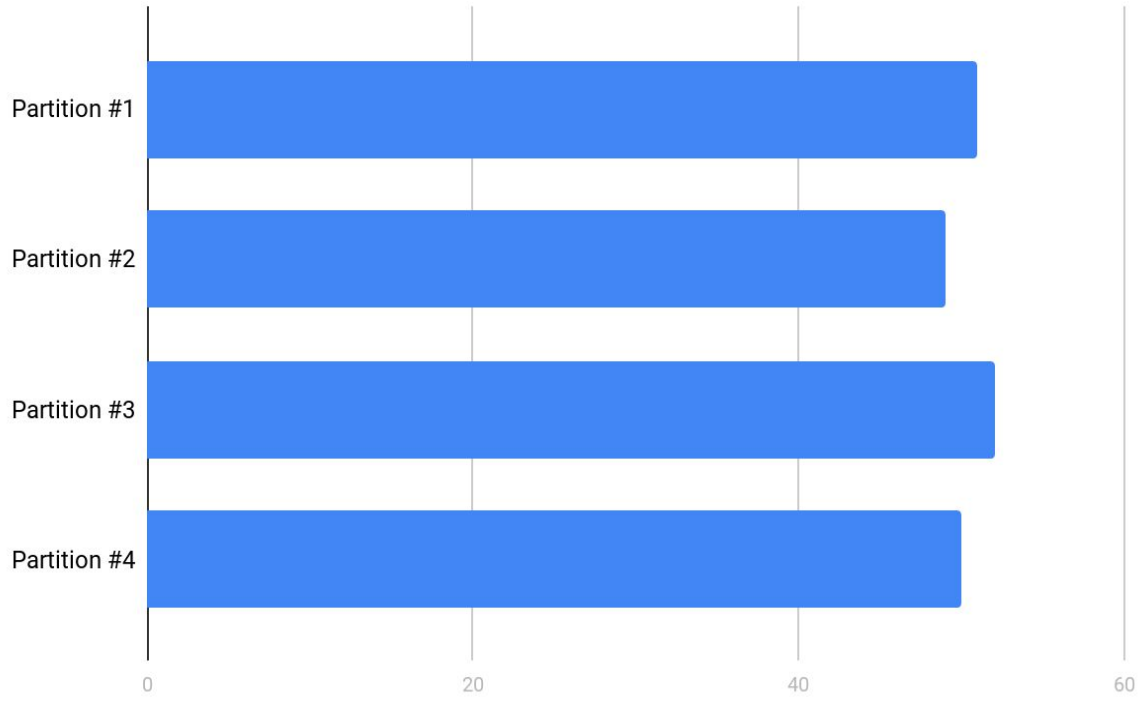
©2023 Databricks Inc. — All rights reserved

Skew

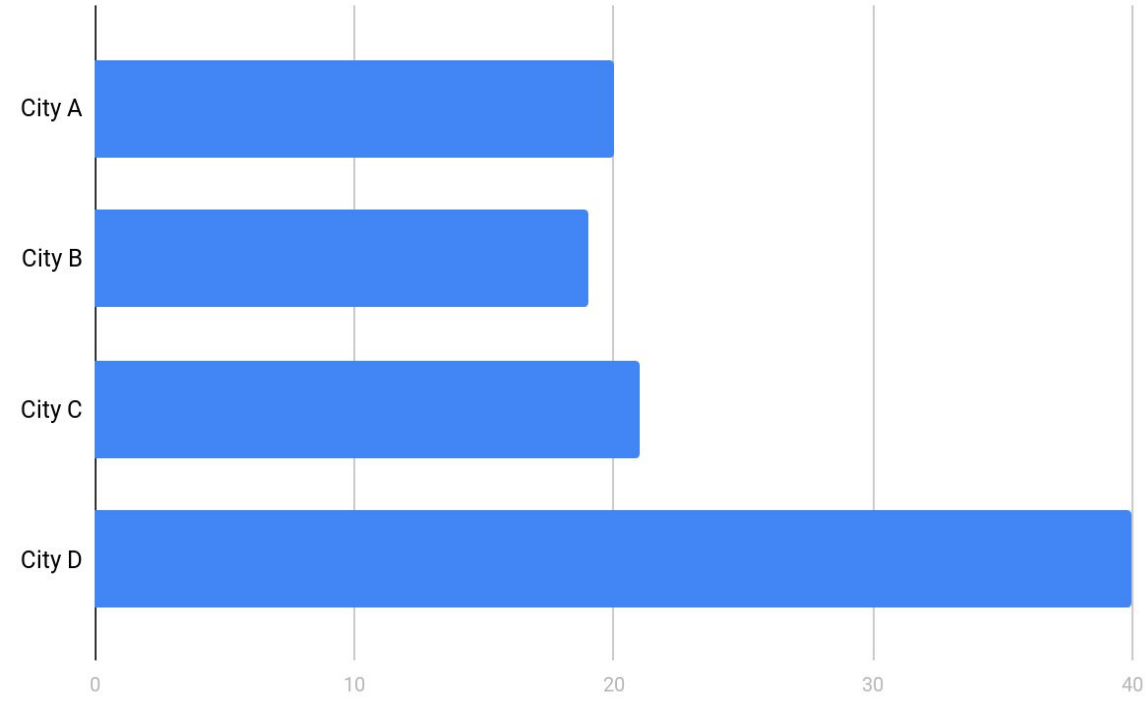
©2023 Databricks Inc. — All rights reserved

Skew – Before and After

Before aggregation



After aggregation by city

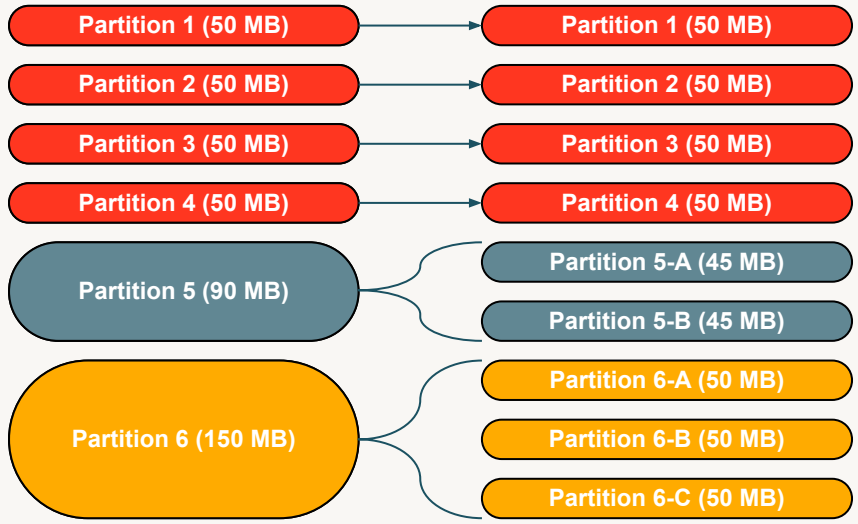


©2023 Databricks Inc. — All rights reserved

Handling Data Skew

Data skew is unavoidable, Databricks handles this automatically

- In MPP systems, data skew significantly impacts performance because some workers are processing much more data
- Most cloud DWs require a manual, offline redistribution to solve for data skew
- With Adaptive Query Execution Spark automatically breaks down larger partitions into smaller, similar sized partitions



©2023 Databricks Inc. — All rights reserved

Skew – Mitigation

Three “common” solutions

1. Filter skewed values
2. Databricks’ [proprietary] Skew Hint
   - Easier to add a single hint than to salt your keys
   - Great option for version of Spark 2.x
3. Adaptive Query Execution (enabled by default in Spark 3.1)
4. Salt the join keys forcing even distribution during the shuffle
   - This is the most complicated solution to implement
   - e.g. “normal” partitions might become too small if not properly adjusted
   - Can sometimes take longer to execute than other solutions

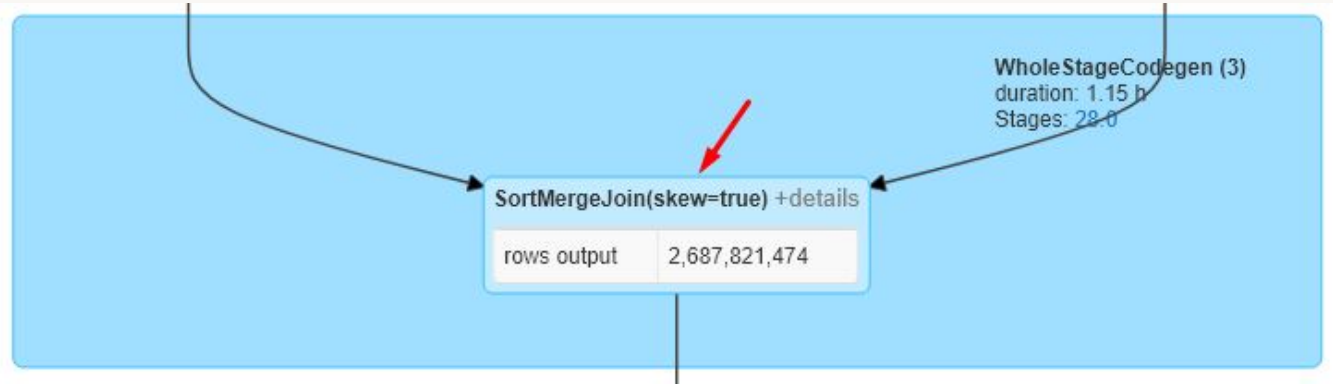
©2023 Databricks Inc. — All rights reserved

Skew Mitigation

See Experiment #1596

Step	Code	Duration	Tasks	Health	Shuffle	Spill
C	Baseline	~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	~37 min	832	OK	~400K / ~70 MB / ~150 MB / ~290 MB / ~790 MB	0

See SQL diagram for **Step E** showing **skew=true**



©2023 Databricks Inc. — All rights reserved

Unhandled Skew

See Experiment #2755

Some transformations can be processed by only one worker when data is not evenly distributed.

Transformations like applyInPandas and Window functions require a full shuffle of the data.

Occurs when a transformation is applied to large tables grouped by low cardinality columns.

There is a potential OOM risk if certain groups are too large to fit in the memory of the worker

Step	Tasks	Stage Execution Time
A - GroupBy + ApplyInPandas	1	~31 secs
B - Window function skew on partition by column	1	~1.7 mins

©2023 Databricks Inc. — All rights reserved

Other Cases of Unhandled Skew

AQE & Streaming

- AQE does not support structure streaming - *forEachBatch()* - so it is automatically disabled in these cases.
- Side effect is that we have no auto handling of data skew on joins
- **Mitigation**: Use skew hints or broadcast joins if possible

©2023 Databricks Inc. — All rights reserved

Serialization

©2023 Databricks Inc. — All rights reserved

Performance problems with serialization

- Spark SQL and DataFrame instructions are highly optimized
- All UDFs must be serialized and distributed to each executor
- The parameters and return value of each UDF must be converted for each row of data before distributing to executors
- Python UDFs takes an even harder hit
  - The Python code has to be pickled
  - Spark must instantiate a Python interpreter in each and every Executor
  - The conversion of each row from Python to DataFrame costs even more

Experiment #4538

©2023 Databricks Inc. — All rights reserved

Mitigating serialization issues

- Don’t use UDFs
  - I challenge you to find a set of transformations that cannot be done with the built-in, continuously optimized, community supported, higher-order functions
- If you have to use UDFs in Python (common for Data Scientist) use the Vectorized UDFs as opposed to the stock Python UDFs
- If you have to use UDFs in Scala use Typed Transformations as opposed to the stock Scala UDFs
- Resist the temptation to use UDFs to integrate Spark code with existing business logic - porting that logic to Spark almost always pays off

©2023 Databricks Inc. — All rights reserved

Serialization – Python vs Scala

Step	Type	Scala/Java 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

©2023 Databricks Inc. — All rights reserved

Fine-Tuning: Choosing the Right Cluster

©2023 Databricks Inc. — All rights reserved

Cluster Types

ALL PURPOSE COMPUTE

- Analysis and ad-hoc DE & DS development
- Shared clusters but best practice is to separate by team or workload
- Anytime an already-running cluster is utilized (including API or scheduled)
- More expensive

JOBS COMPUTE

- Run on ephemeral clusters that are created for the job, and terminate on completion
- Pre-scheduled or submitted via API
- Single-user
- Great for isolation and debugging
- Production and repeat workloads
- Lower cost

SQL WAREHOUSE

- Built for high concurrency ad-hoc SQL analytics and BI serving
- Photon included
- Recommended shared warehouse for ad-hoc SQL analytics, isolated warehouse for specific workloads
- Serverless available for instant startup

©2023 Databricks Inc. — All rights reserved

Autoscaling

- Dynamically resizes cluster based on workload
  - Can run faster than a statically-sized, under-provisioned cluster
  - Can reduce overall costs compared to a statically-sized cluster
- Setting range for the number of workers requires some experimenting

Use Case	Autoscaling Range
Ad-hoc usage or business analytics	Large variance
Production batch jobs	Not needed or buffer on upper limit
Streaming	Available in Delta Live Tables

©2023 Databricks Inc. — All rights reserved

Spot Instances

- Use spot instances to use spare VM instances for below market rate
  - Great for ad-hoc/shared clusters
  - Not recommended for jobs with mission critical SLAs
  - Never use for driver!
- Combine on-demand and spot instances (with custom spot price) to tailor clusters to different use cases

SLA	Spot or On-Demand
Non-mission critical jobs	Driver on-demand and workers spot
Workflows with tight SLAs	Use spot instance w/fallback to on-demand

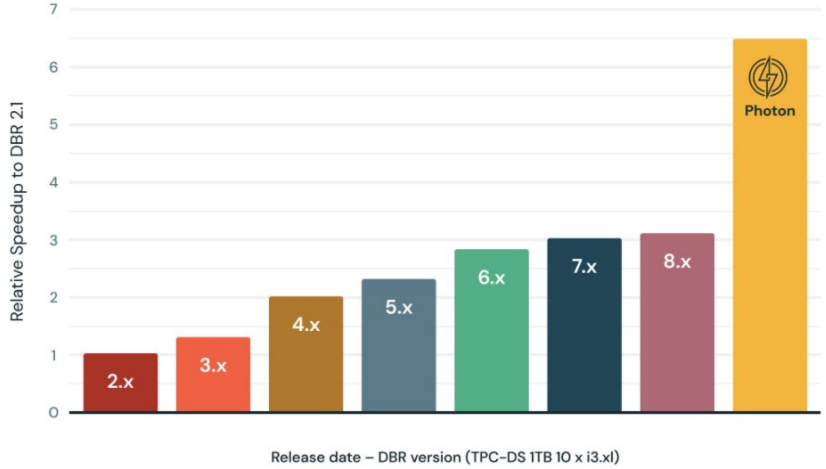
©2023 Databricks Inc. — All rights reserved

Photon

World record achieving query engine with zero tuning or setup

- Save on compute costs
  - ETL customers are saving up to 40% on their compute cost
- Fast query performance
  - Built for modern hardware with up to 12x better price/perf compared to other cloud data warehouses
- No code changes
  - Spark APIs that can do exploration, ETL, big data, small data, low latency, high concurrency, batch, and streaming
- Broad language support
  - Support for SQL, Python, Scala, R, and Java

Databricks Sets Official Data Warehousing Performance Record

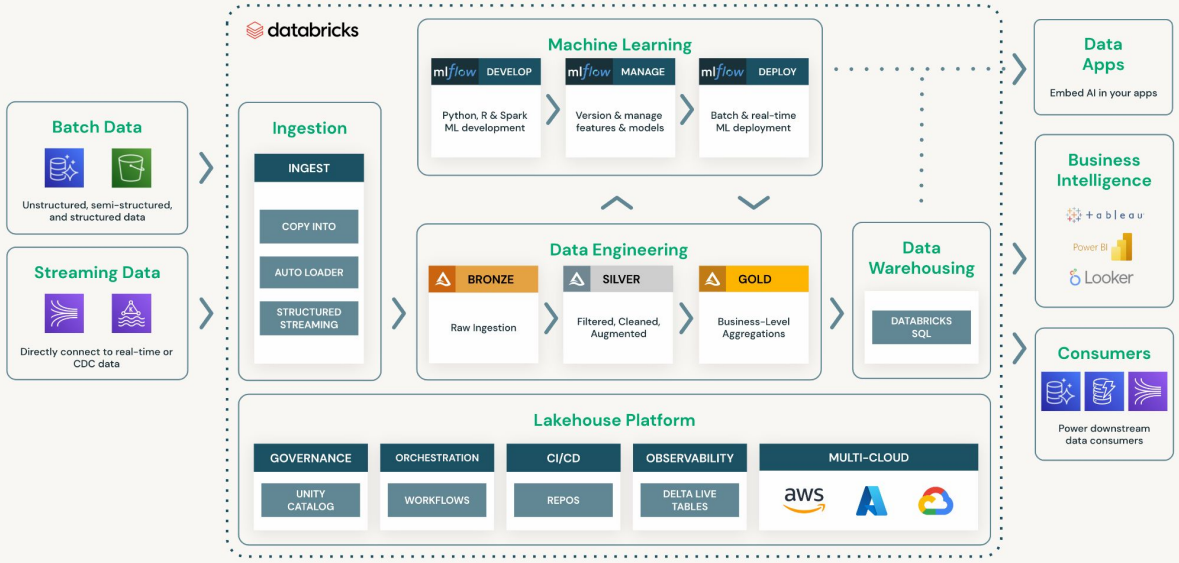


©2023 Databricks Inc. — All rights reserved

Architectural Considerations

Isolated Clusters & Warehouses to Avoid Resource Contention

Databricks AWS Reference Architecture



1. **Ephemeral Job compute**
   - Jobs - Isolated compute for ingestion + ETL jobs, can be sized/optimized for that workload, run on a schedule
   - Only charged for when the job is running
2. **Shared development clusters**
   - All-purpose - Auto-scale, auto-pause to only use when teams are actively developing, only resources needed
   - Recommended to develop and test with a subset of the full dataset
3. **Shared SQL warehouse for ad-hoc analysis**
   - SQL warehouse - Auto-scale, auto-pause to only use when teams are actively querying, only resources needed
   - Serverless available for instant startup, shutdown to reduce idle time
4. **Separate SQL warehouse for BI reporting**
   - Size appropriately for BI needs, avoid contention with other processes

Note: Most other big data and data warehousing platform architectures are monolithic and require tedious, manual workload management

©2023 Databricks Inc. — All rights reserved

Cluster Optimization Recommendations

1. **DS & DE development:** all-purpose compute, auto-scale and auto-stop enabled, develop & test on a subset of the data
2. **Ingestion & ETL jobs:** jobs compute, auto-scale enabled and size accordingly to job SLA
3. **Ad-hoc SQL analytics**: (serverless) SQL warehouse, auto-scale and auto-stop enabled
4. **BI Reporting:** isolated SQL warehouse, sized according to BI SLAs
5. **Best practices**:
    a. Enable spot instances on worker nodes
    b. Use Graviton instances when possible
    c. Use the latest LTS Databricks Runtime when possible
    d. Use Photon for best TCO when applicable
    e. Use latest gen VM, start with general purpose, then test memory/compute optimized

©2023 Databricks Inc. — All rights reserved