
databricks
Databricks
Performance
Optimization
Databricks Academy
January 2025
©2024 Databricks Inc. — All rights reserved

Agenda

1. Spark Architecture	Lecture	Demo	Lab
Spark UI Introduction	✓		
2. Designing the Foundation	**Lecture**	**Demo**	**Lab**
File Explosion		✓	
Data Skipping and Liquid Clustering	✓		✓
3. Code Optimization	**Lecture**	**Demo**	**Lab**
Skew	✓		
Shuffle	✓	✓	
Spill	✓		
Exploding Join			✓
Serialization	✓		
User-Defined Functions		✓	
4. Fine-Tuning: Choosing the Right Cluster	**Lecture**	**Demo**	**Lab**
Fine-Tuning: Choosing the Right Cluster	✓		
Pick the Best Instance Types	✓		

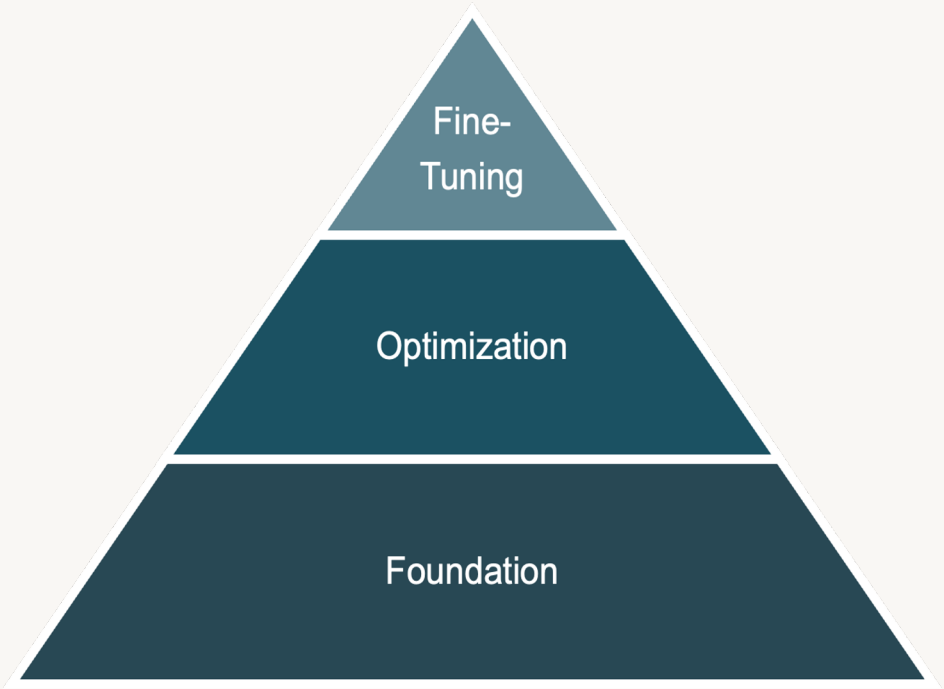
Introduction

We commence with Designing the Foundation, focusing on establishing fundamental principles in Spark programming. Following this, we delve into Code Optimization, uncovering strategies to elevate code efficiency and performance. Our exploration further extends to understanding the intricate layers of Spark Architecture and optimizing clusters for diverse workloads in Fine-Tuning - Choosing the Right Cluster.

Beyond theory, our sessions offer immersive hands-on experiences. Engage in real-time simulation through Follow Along - Spark Simulator, and dive deep into critical operational aspects such as Shuffles, Spill, Skew, alongside understanding the prowess of Serialization in Spark.

This course aims to equip you with comprehensive expertise in advanced data engineering, leveraging the powerful tools and techniques offered by Databricks.

Building Performance Analytics



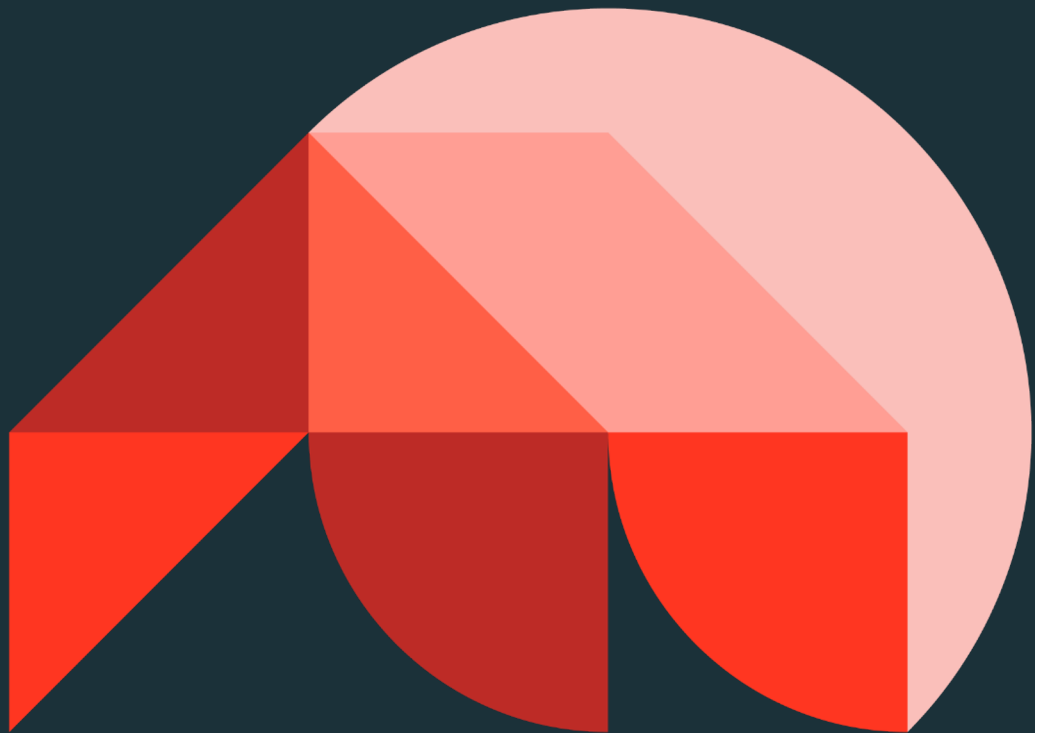
Cluster Sizing

Code
Optimization

File Layout

databricks

Spark Architecture



Databricks Performance Optimization

databricks

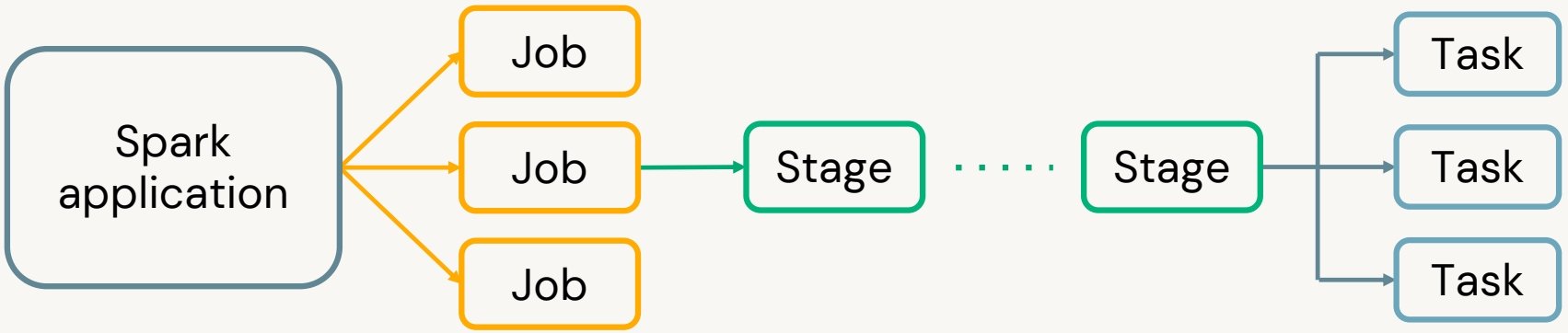
Spark Architecture

LECTURE

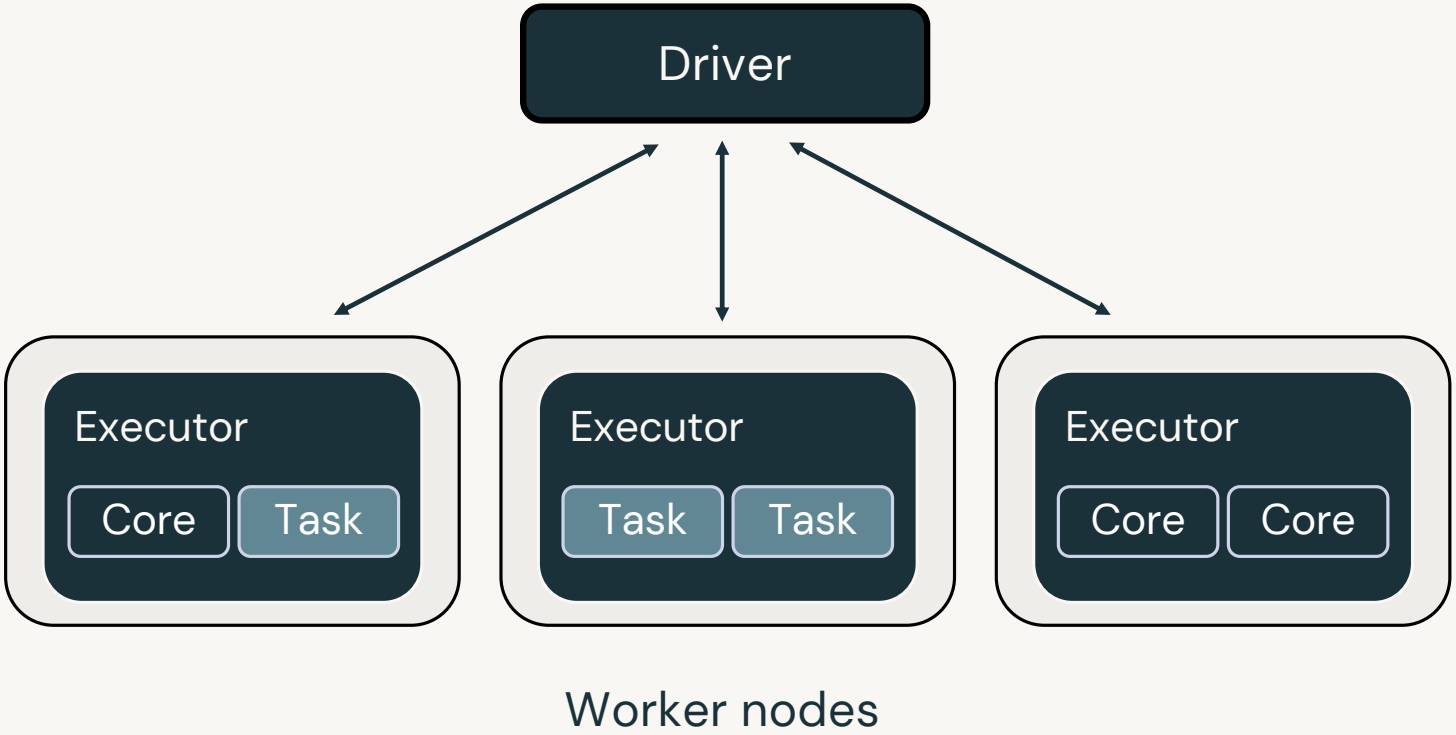
Spark UI Introduction

Executing a Spark Application

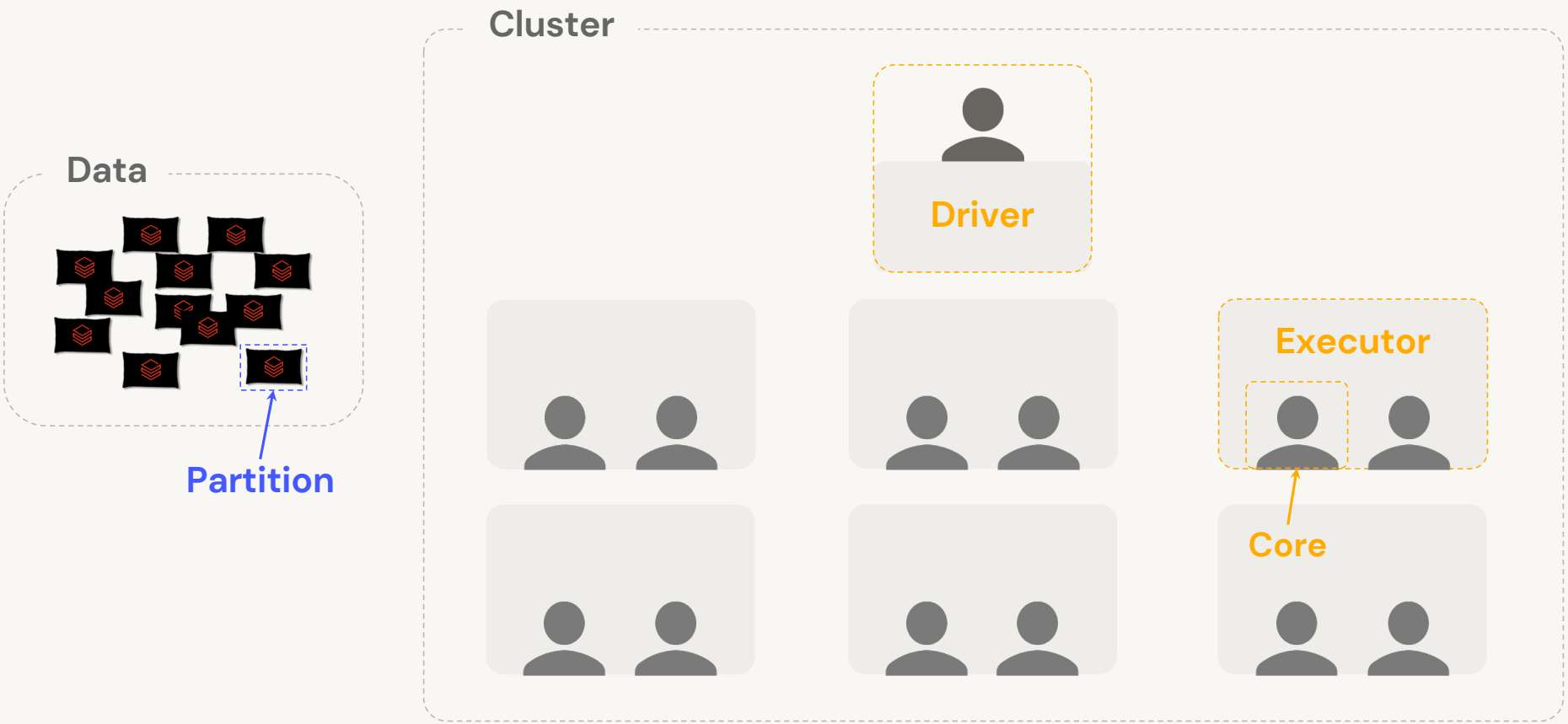
Data processing tasks run in parallel across a cluster of machines

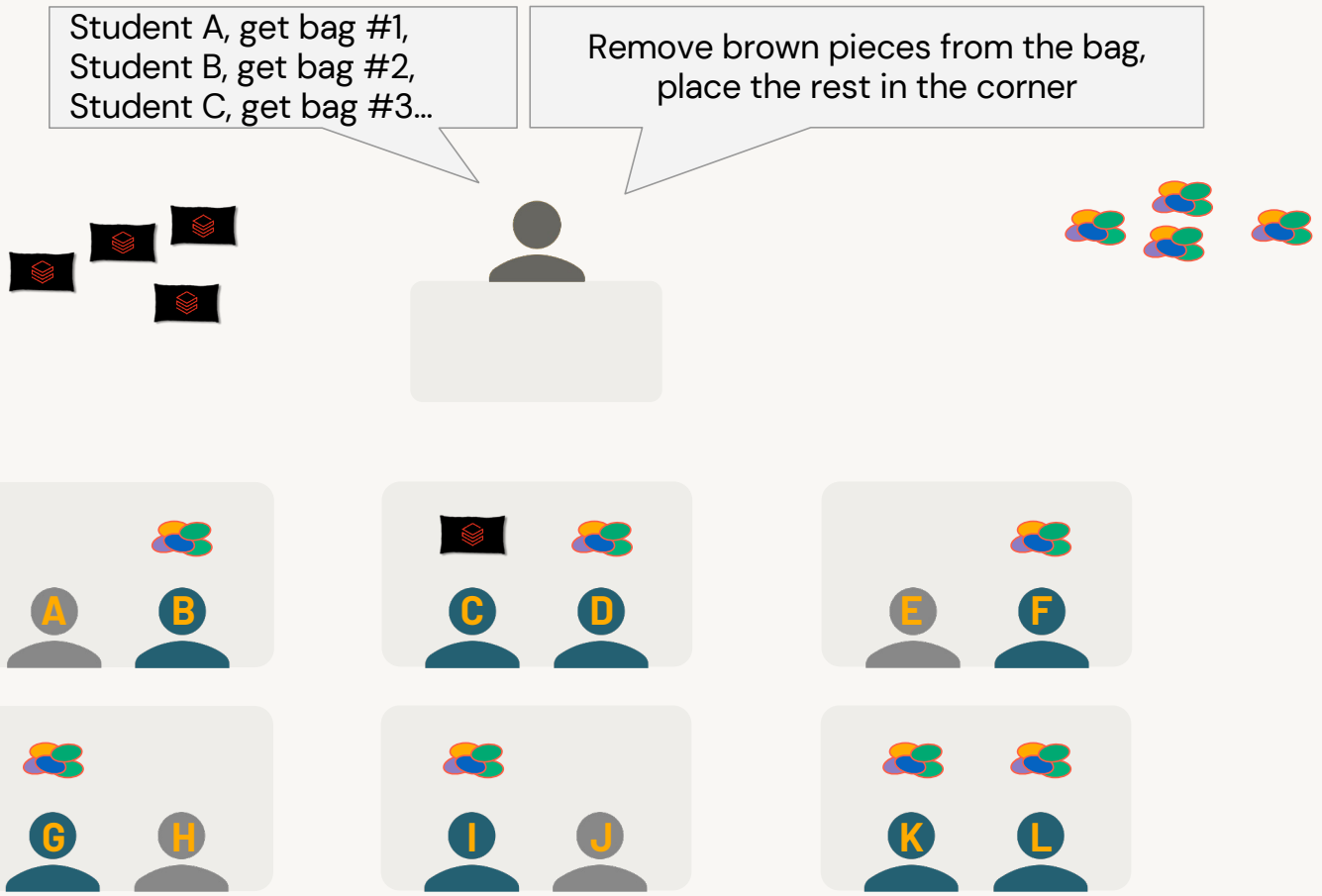


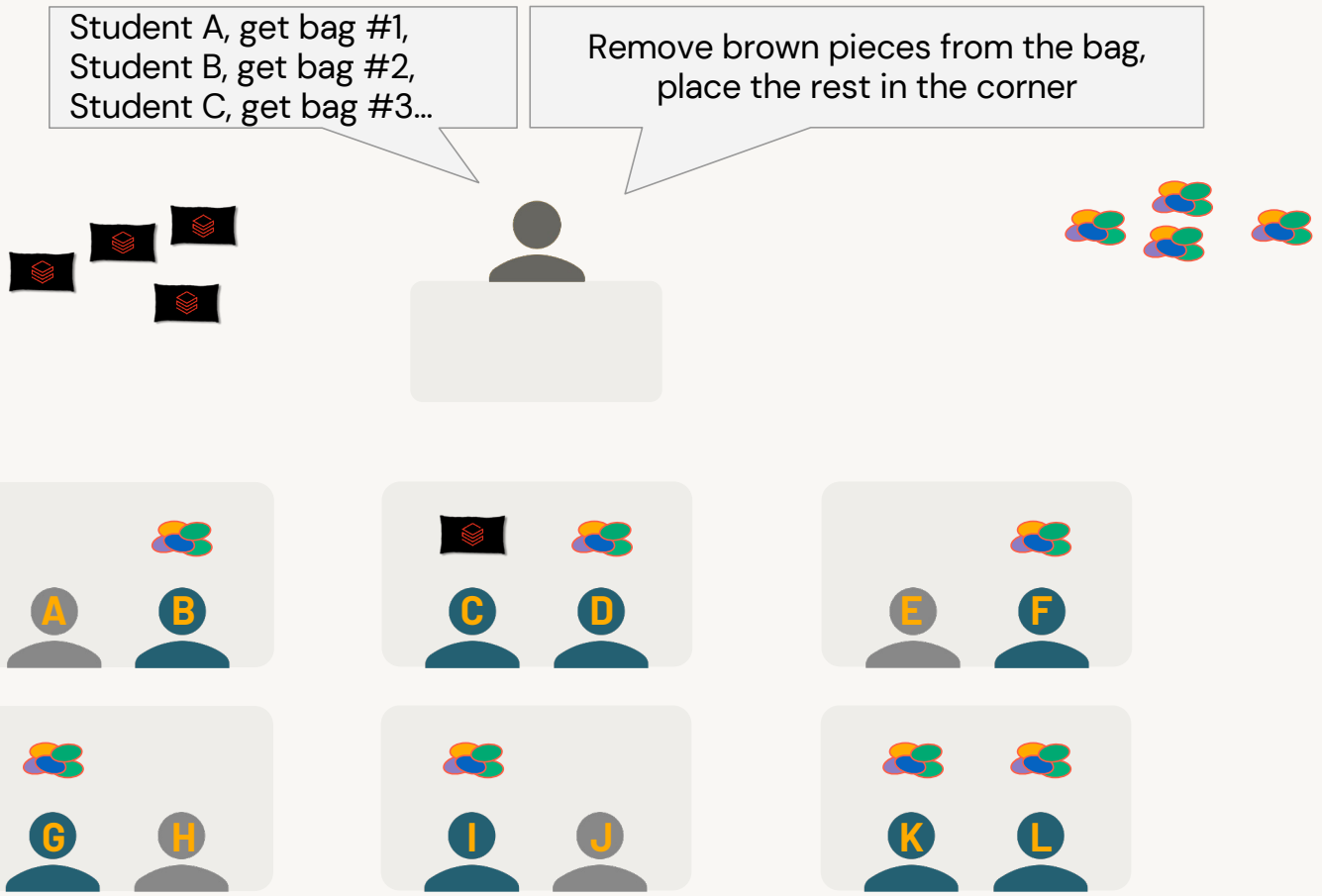
Spark Architecture

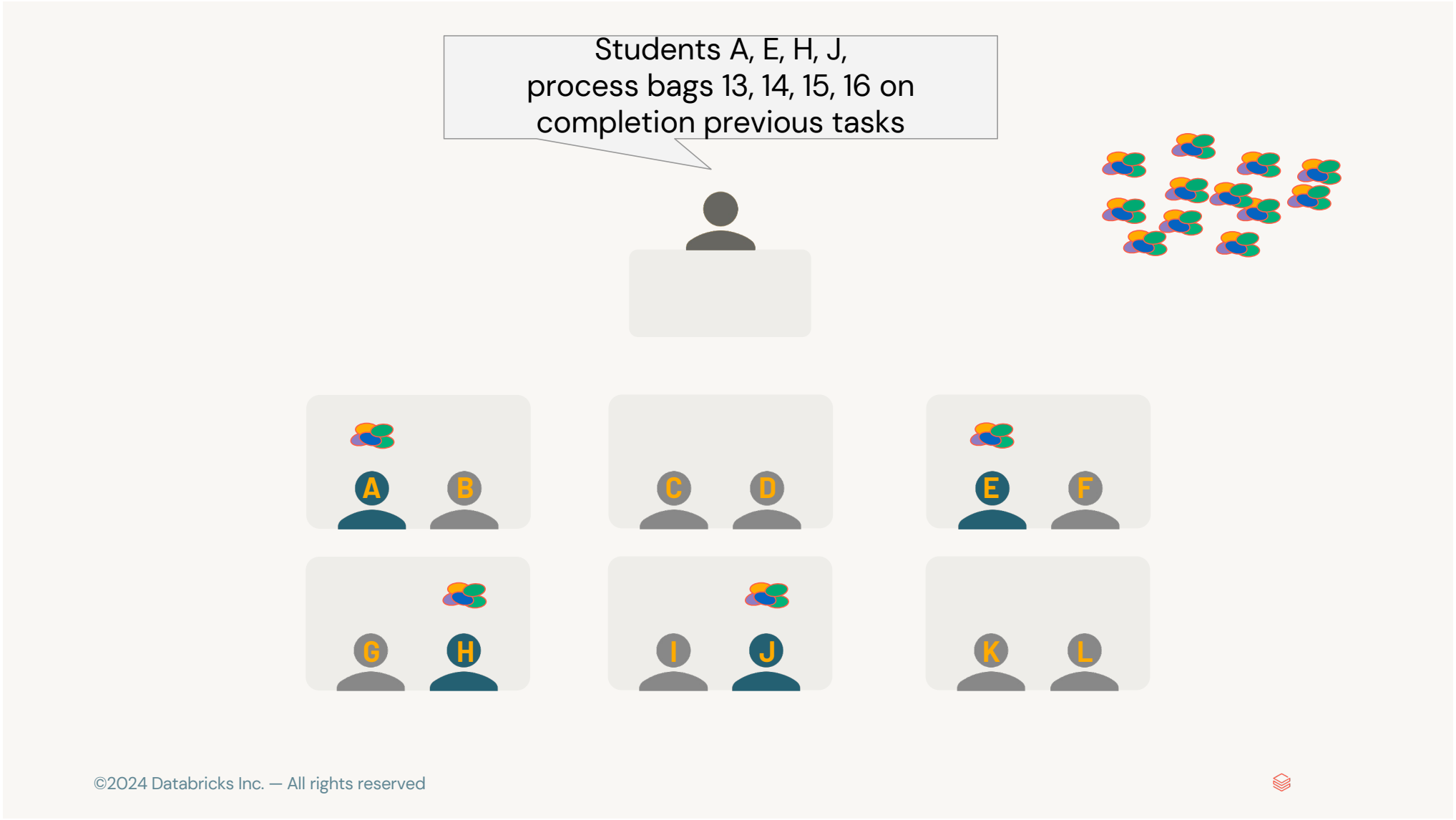


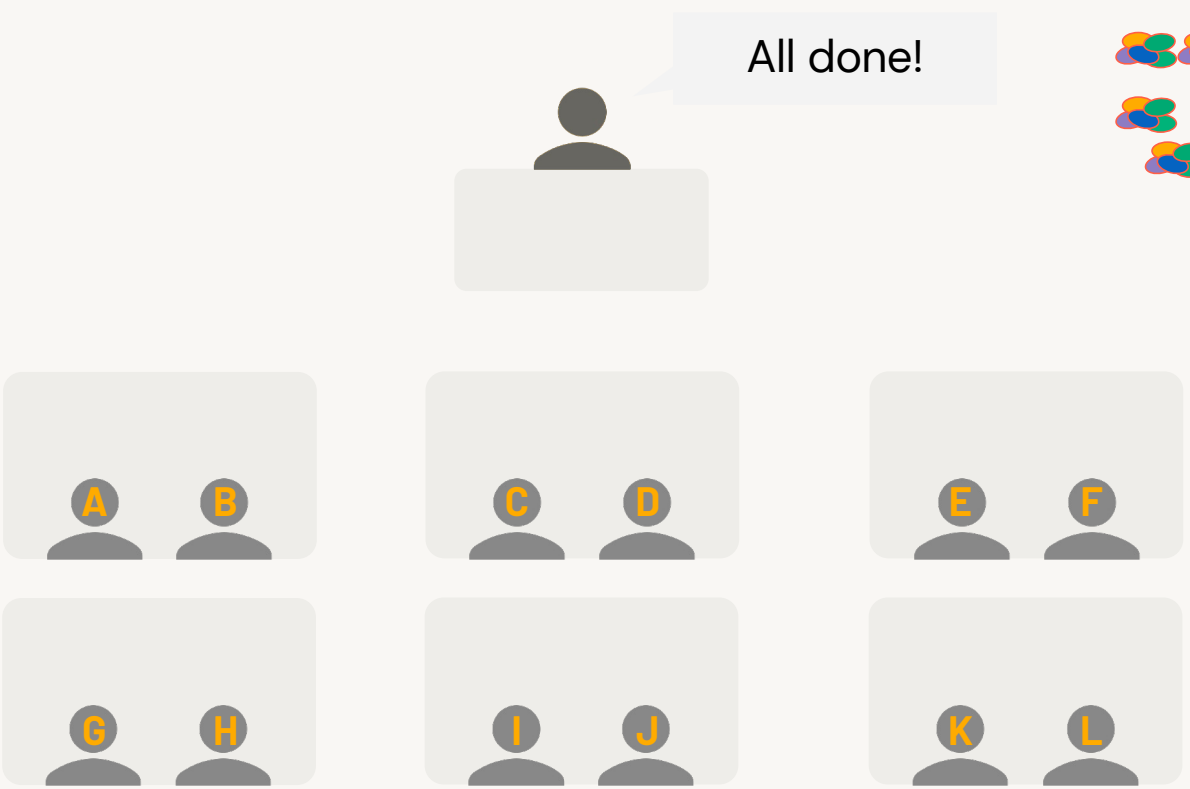
Scenario: Filter out brown pieces from these candy bags

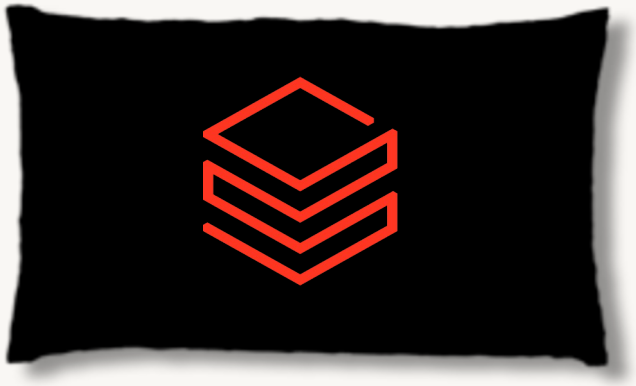








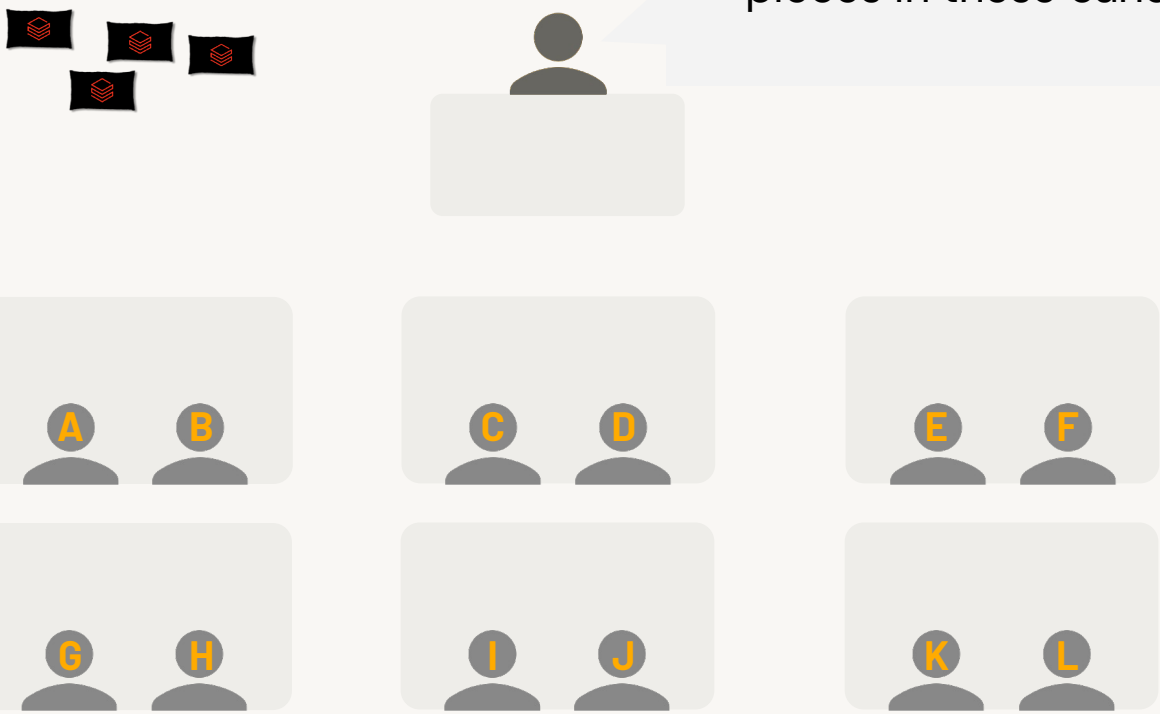




Scenario 2: Count Total Pieces in Candy Bags Introducing Stages

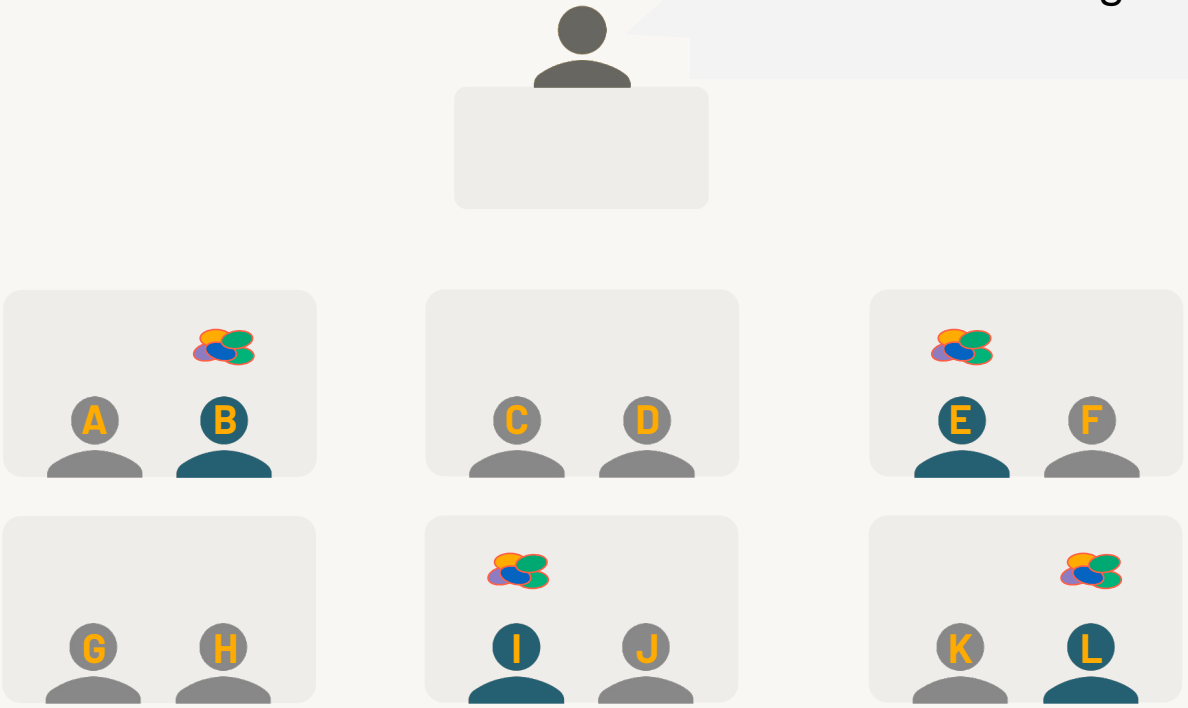
Stage 1: Local Count

We need to count the total pieces in these candy bags



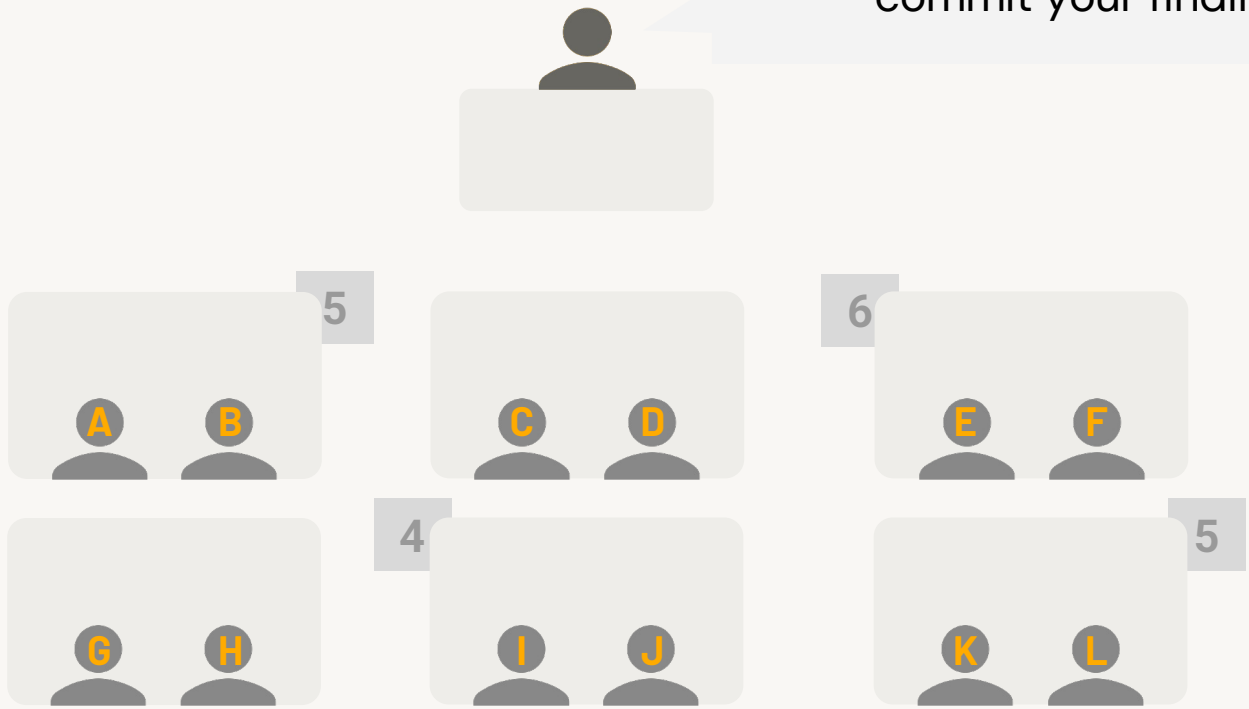
Stage 1: Local Count

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



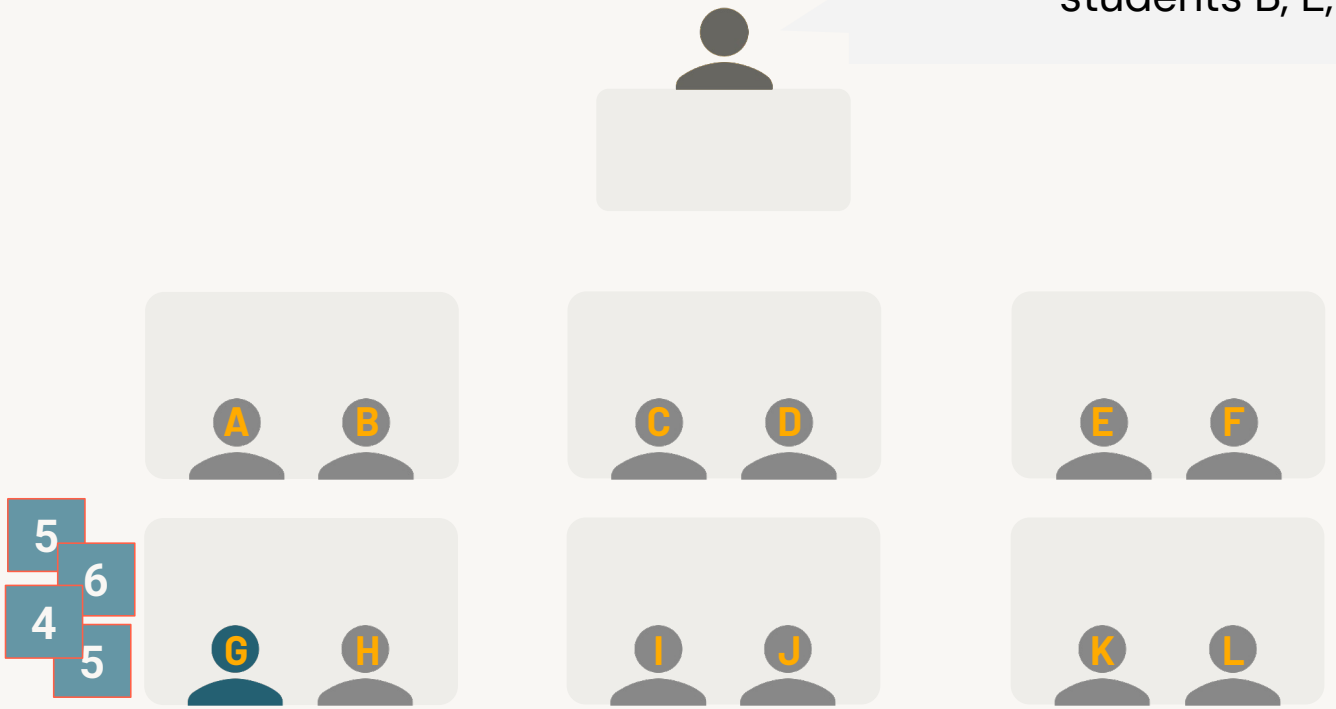
Stage 1: Local Count

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

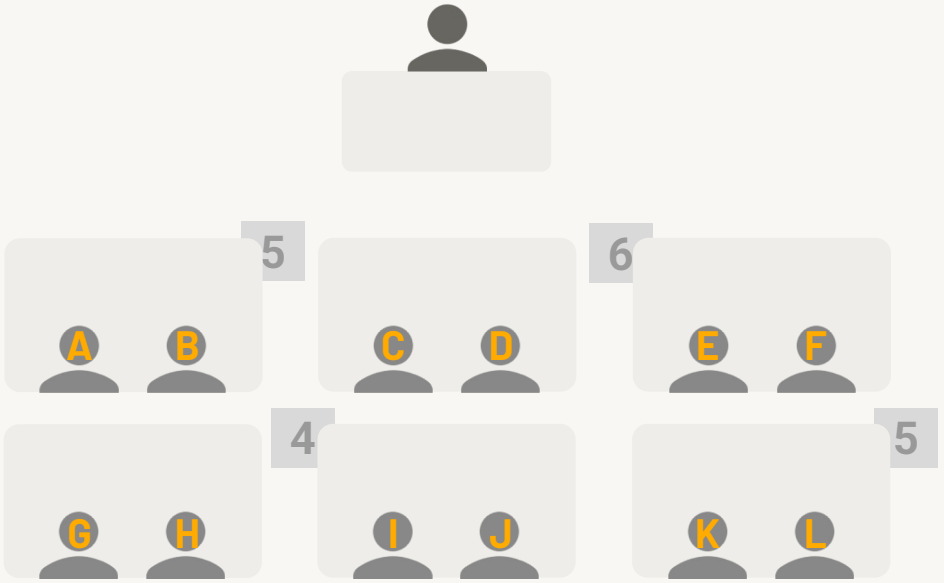


Stage 2: Global Count

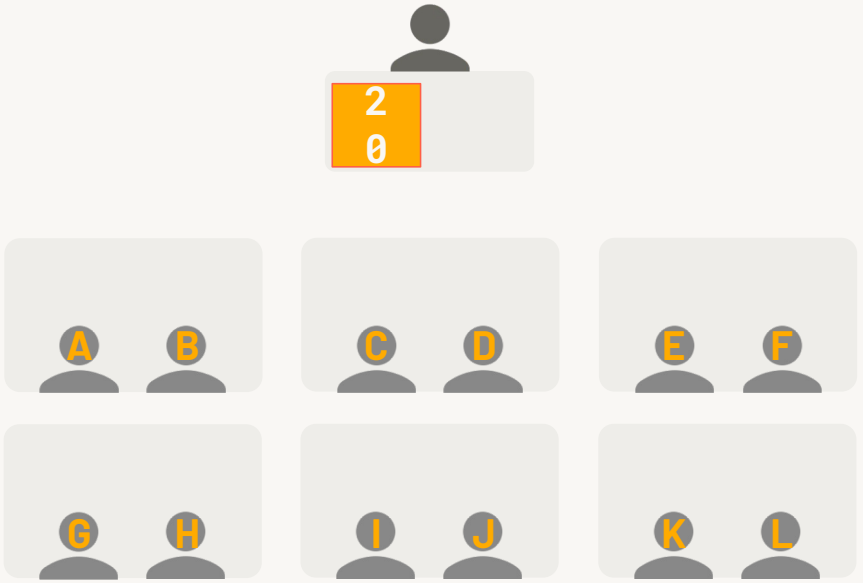
Student G, total counts from students B, E, I, L



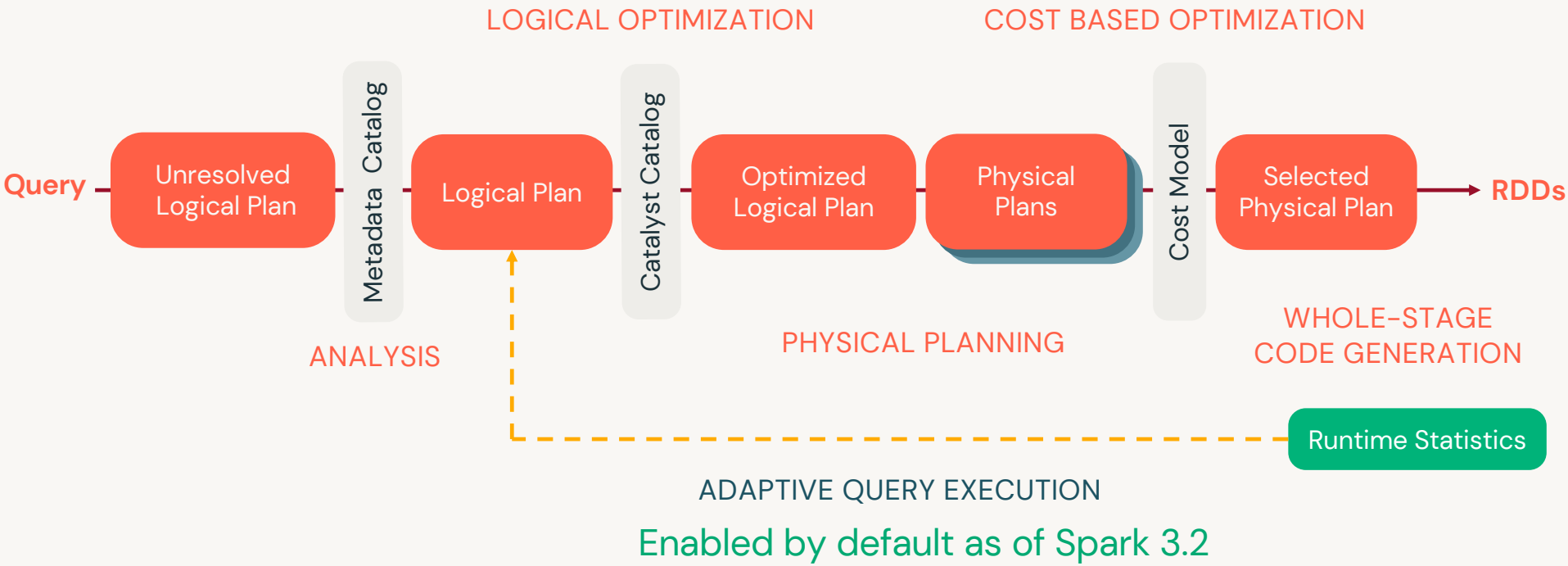
Stage 1: Local Count



Stage 2: Global Count



Query Optimization

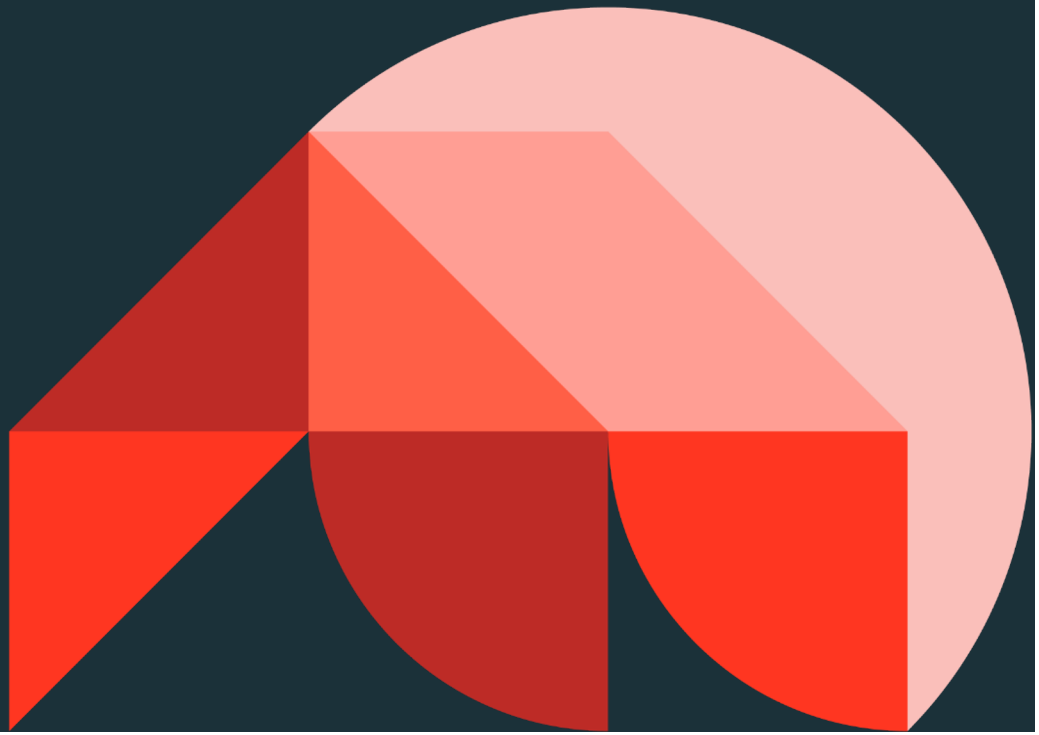


Code Optimization Recommendations

1. Using Dataframes or SQL instead of RDD APIs.
2. In production jobs, avoid unnecessary operations that trigger an action besides reading and writing files. These operations might include count(), display(), collect().
3. 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.

databricks

Designing the Foundation



Databricks Performance Optimization

databricks

Designing the Foundation

LECTURE

Introduction to Designing Foundation

Fundamental Concepts

Why some schemas and queries perform faster than others

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

Common Performance Bottlenecks

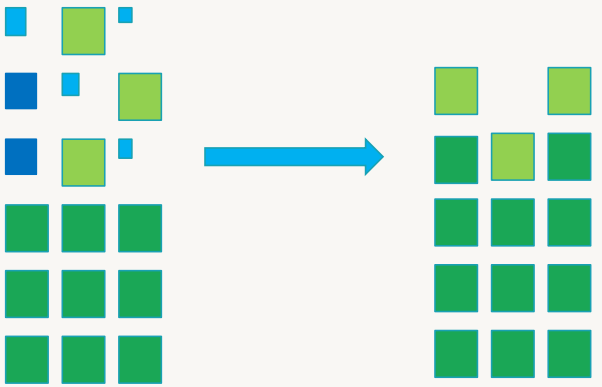
Encountered with any big data or MPP system

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

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 will automatically compact small files on write with auto-optimize

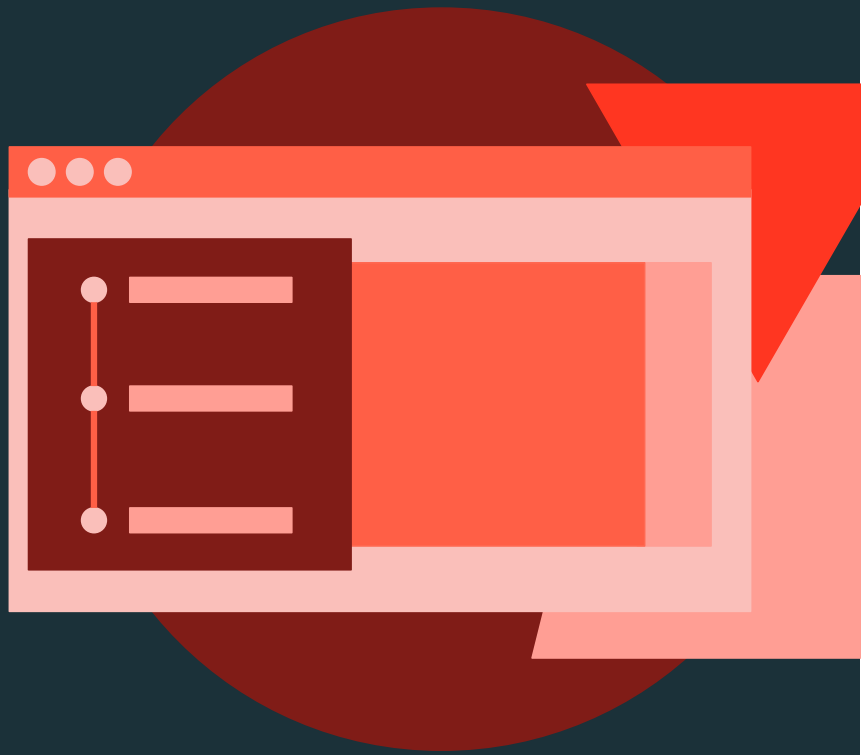


databricks

Designing the Foundation

DEMONSTRATION

File Explosion



databricks

Designing the Foundation

LECTURE

Data Skipping and Liquid Clustering

Data Skipping

Simple, well-known I/O pruning technique

- Track file-level stats like min & max
- Leverage them to avoid scanning irrelevant files

```
SELECT input_file_name() as
“file_name”,
       min(col) AS “col_min”,
       max(col) AS “col_max”
FROM table
GROUP BY input_file_name()
```

file_name	col_min	col_max
1. parquet	6	8
2. parquet	3	10
3. parquet	1	4

Z-Ordering

```
Optimize Table Z-Order by Column
```

Old Layout

file_name	col_min	col_max
1. parquet	6	8
2. parquet	3	10
3. parquet	1	4

New Layout

file_name	col_min	col_max
1. parquet	1	3
2. parquet	4	7
3. parquet	8	10

Z-Ordering

```
Select * from Table Where Col = 7
```

Old Layout

file_name	col_min	col_max
1. parquet	6	8
2. parquet	3	10
3. parquet	1	4

New Layout

file_name	col_min	col_max
1. parquet	1	3
2. parquet	4	7
3. parquet	8	10

Databricks Delta Lake and Stats

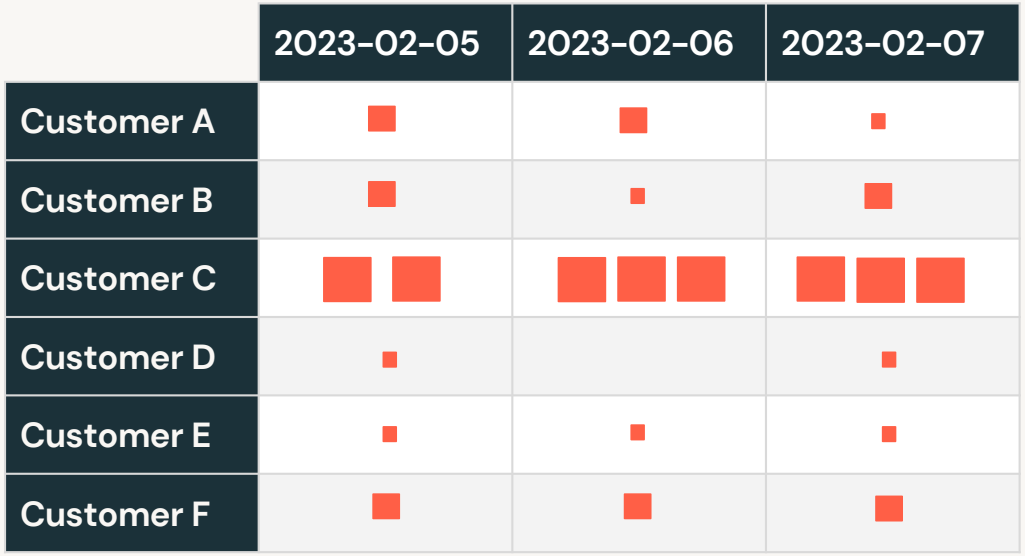
- Databricks Delta Lake collects stats about the first N columns
  - dataSkippingNumIndexedCols = 32
- These stats are used in queries:
  - Metadata only queries: select max(col) from table
    - Queries just the Delta Log, doesn't need to look at the files if col has stats
  - Allows us to skip files
    - Partition FIlters, **Data Filters**, Pushed Filters apply in that order
  - TimeStamp and String types aren't always very useful
    - Precision/Truncation prevent exact matches, have to fall back to files sometimes
- Avoid collecting stats on long strings
  - Put them outside first 32 columns or collect stats on fewer columns
    - alter table change column col after col32
    - set spark.databricks.delta.properties.defaults.dataSkippingNumIndexedCols = 3

What about Partitioning?

- Generally not recommended!
  - Partitioning is usually misused/overused
  - Tiny file problems or Skew
- Good use-cases for partitioning
  - Isolating data for separate schemas (single->multiplexing)
  - GDPR/CCP use cases where you commonly delete a partitions worth of data
  - Use cases requiring a physical boundary to isolate data SCD Type 2, partition on current or not for better performance.
- If you partition
  - Choose column with low cardinality
  - Try to keep each partition less than 1tb and greater than 1gb
  - Tables expected to grow TBs
  - Partition (usually) on a date, zorder on commonly used predicates in where clauses

Challenges with Disk Partitioning

Partition by customer ID and date + optimize



- **Many small files.**
  - High metadata operations overhead.
  - Slow read operations.

- **Data skew.**
  - Inconsistent file sizes across partitions.

Introducing Liquid Clustering

What it is

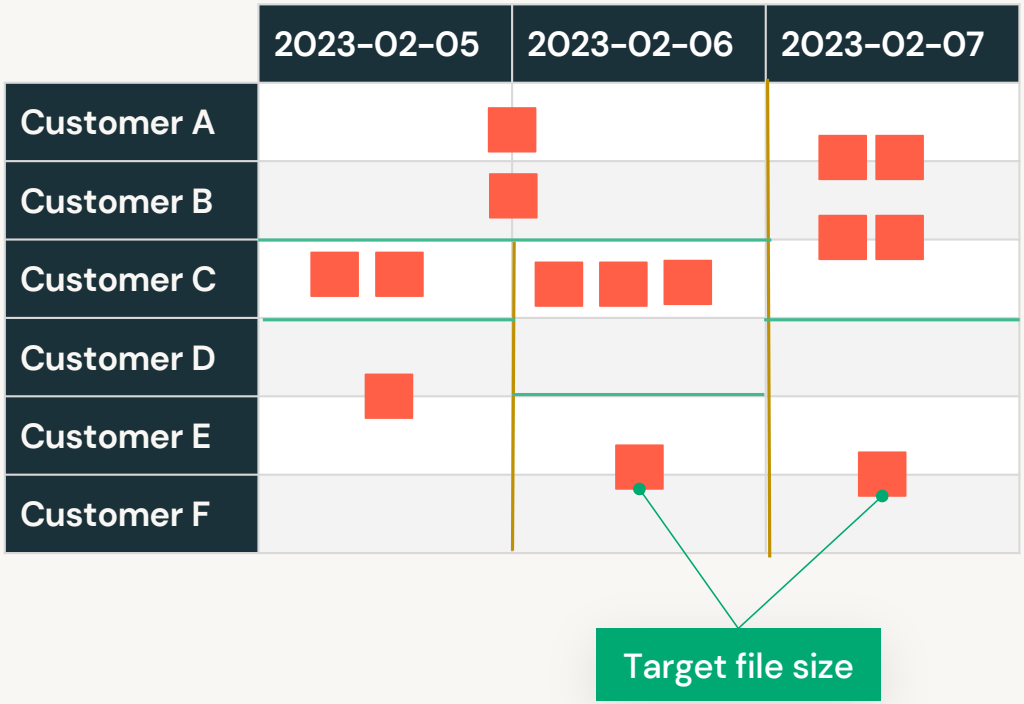
Innovative technique to clustering data layout to support efficient query access and reduce data management and tuning overhead. It's flexible and adaptive to data pattern changes, scaling, and data skew

Benefits

- Best performance out of the box
  - Clustering on write
- Most consistent data skipping
  - Immune to data skew
- Minimal write amplification on table maintenance
  - True incremental optimize
- Row Level Concurrency
  - Simplify logic of concurrent writers
- Reduced Cognitive Overhead
  - No worrying about cardinality

Liquid Clustering

Liquid cluster by customer ID and date



- Liquid is not subject to rigid boundaries
  - Liquid intelligently decides what ranges of data to combine
- Data skew is gone
  - Data sizes are consistent
- Liquid stores metadata
  - New data can be clustered into existing clusters on write

Table Statistics

Keeping table statistics up 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
```


Predictive Optimization

What is Predictive Optimization?

- Predictive optimization refers to using predictive analytics techniques to automatically optimize and enhance the performance of systems, processes, or workflow.

- It involves leveraging data-driven insights to proactively identify and implement optimizations, improving efficiency, cost-effectiveness, and overall system performance.

Predictive Optimization

Key Features

Automatic Maintenance

- It automates the execution of background maintenance tasks on Delta tables.

Support Maintenance Operations

- It supports maintenance operations, including **OPTIMIZE** to improve query performance by optimizing file sizes and **VACUUM** to reduce storage costs by deleting unused data

Set and Forget Approach

- It intelligently and automatically runs maintenance jobs without requiring ongoing user supervision.

Serverless Computing

- It utilizes serverless compute, eliminating the need for users to manually manage compute cluster.

databricks

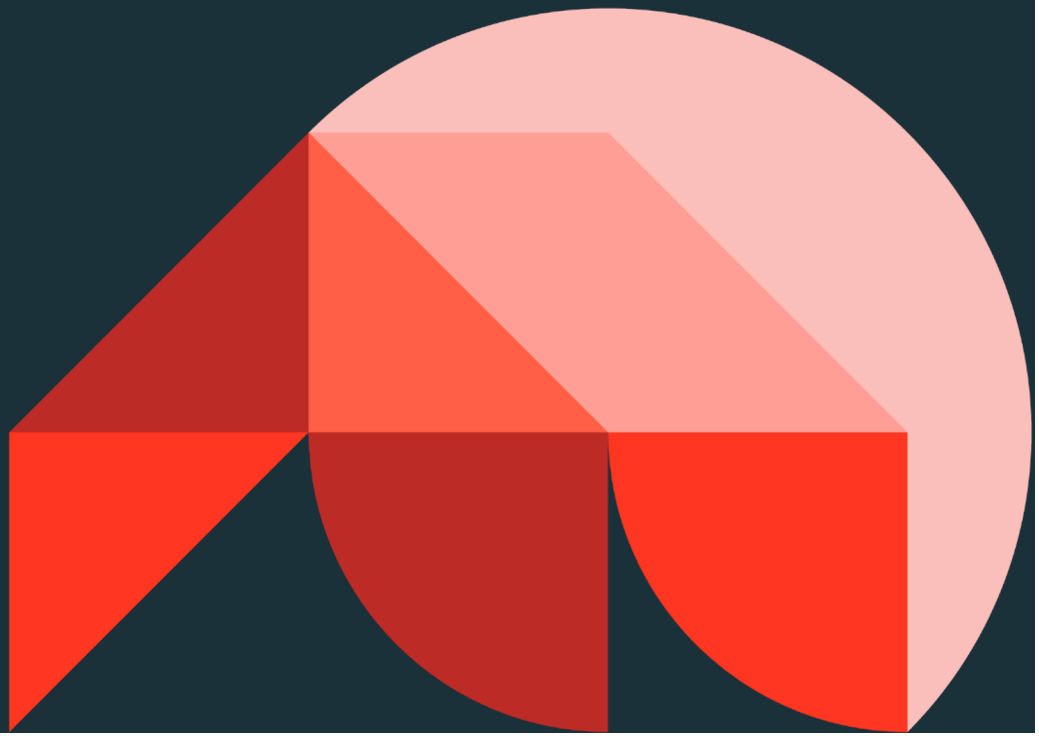
Designing the Foundation

LAB EXERCISE

Data Skipping and Liquid Clustering

databricks

Code Optimization



Databricks Performance Optimization

Code Optimization

4 commonly seen performance problems associate with Spark

- Skew
- Shuffles
- Spill
- Serialization
- Adaptive Query Execution in action

databricks

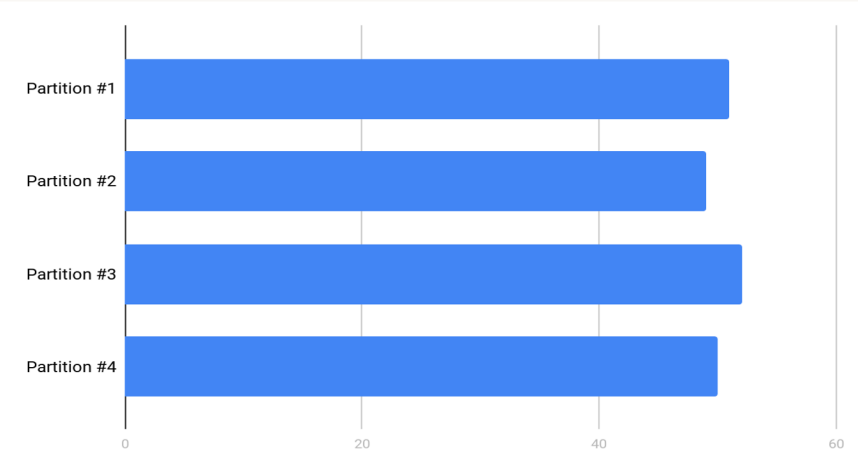
Code Optimization

LECTURE

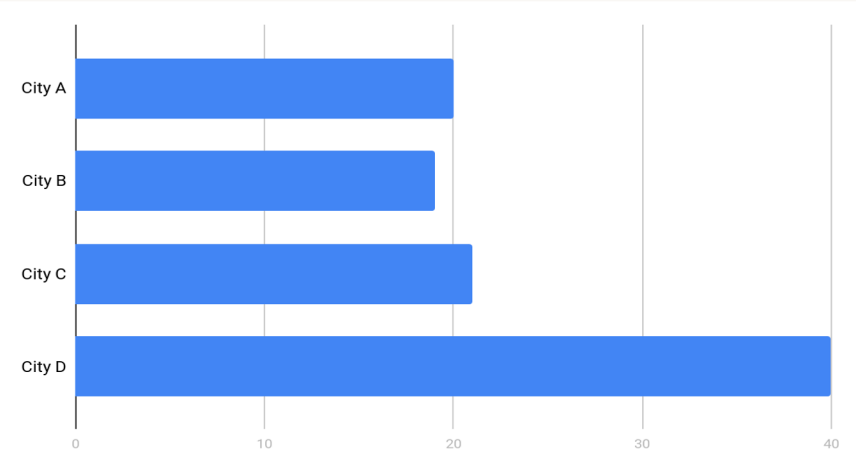
Skew

Skew – Before and After

Before aggregation



After aggregation by city



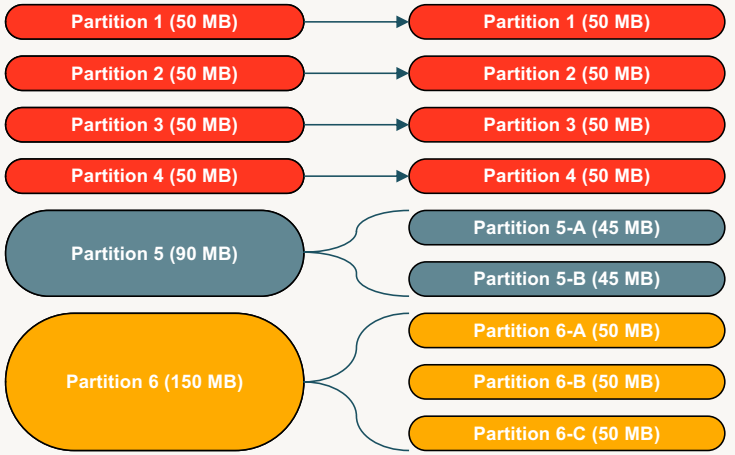
Handling Data Skew

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.



Skew – Mitigation

Three “common” solutions

1. Adaptive Query Execution (enabled by default in Spark 3.1)
2. Filter skewed values
3. Databricks’ [proprietary] Skew Hint
    - Easier to add a single hint than to salt your keys
    - Great option for version of Spark 2.x
4. Salt the join keys forcing even distribution during the shuffle
    - If none of the options are suitable, salting is the only alternative
    - It involves breaking a large skewed partition into smaller ones by adding random integers as suffixes.

databricks

Code Optimization

LECTURE

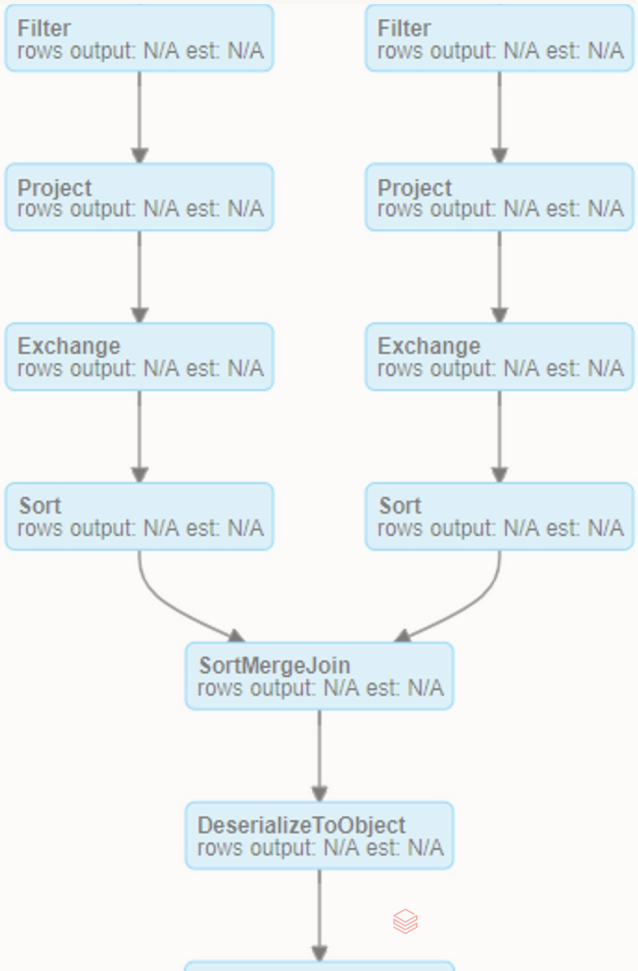
Shuffles

Shuffles

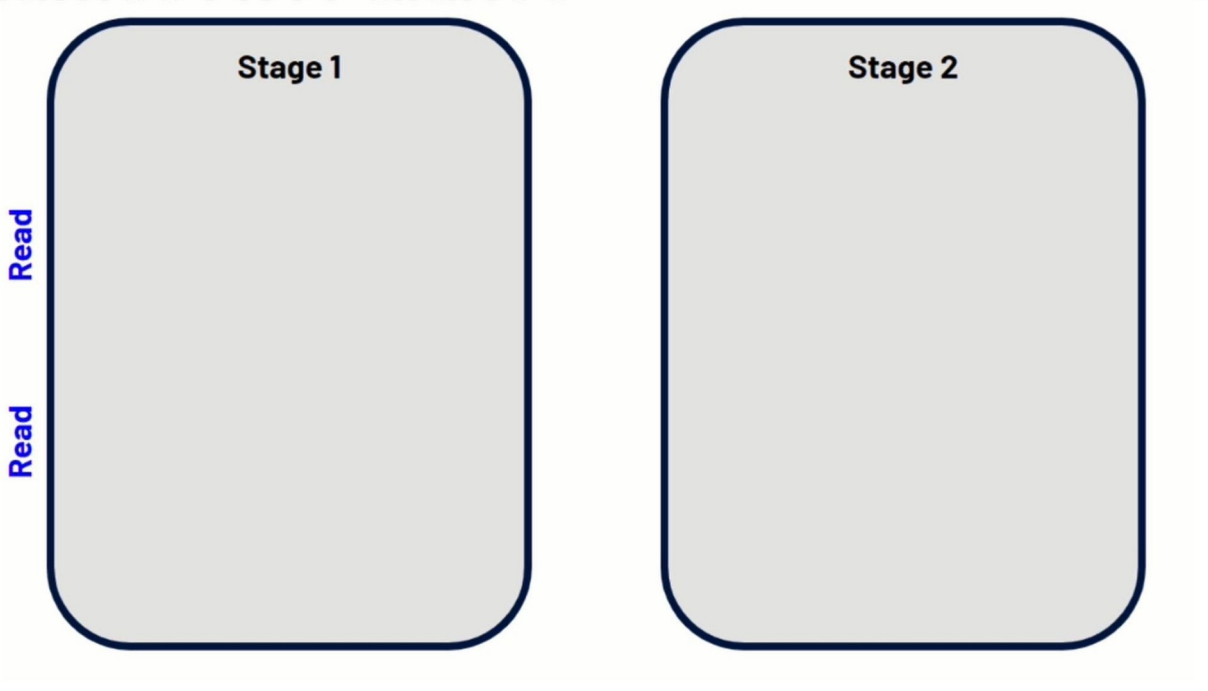
Shuffling is a side effect of **wide transformations**

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

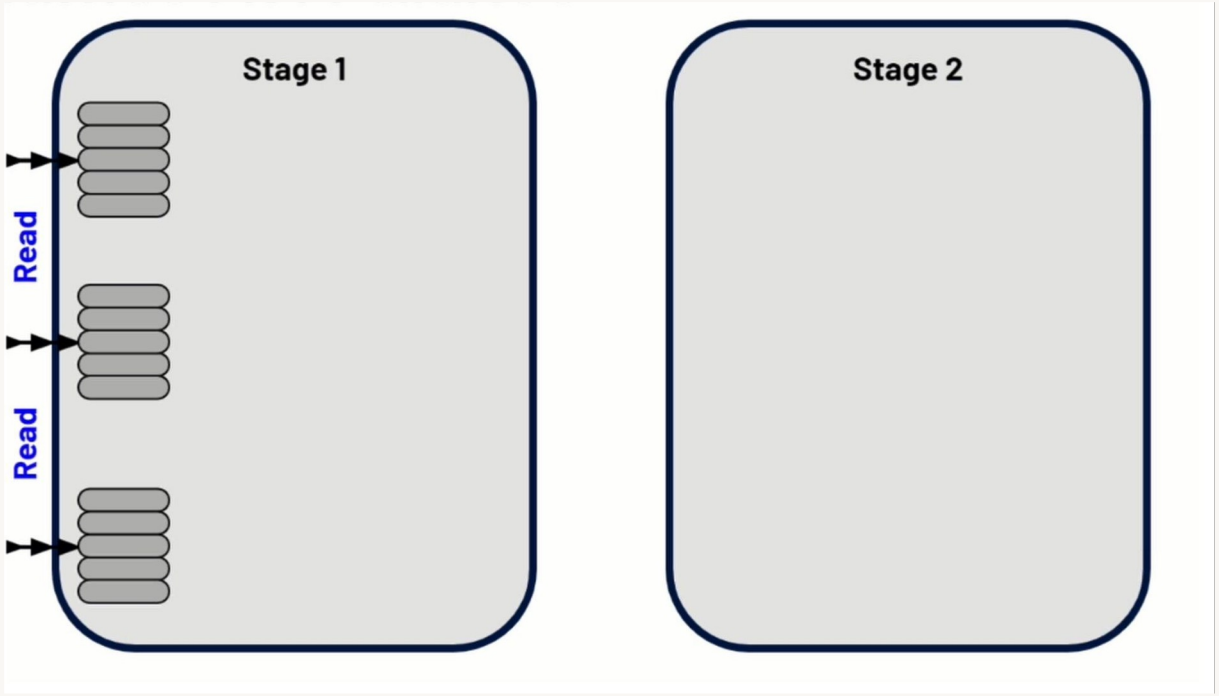
And technically some actions, e.g. **`count()`**



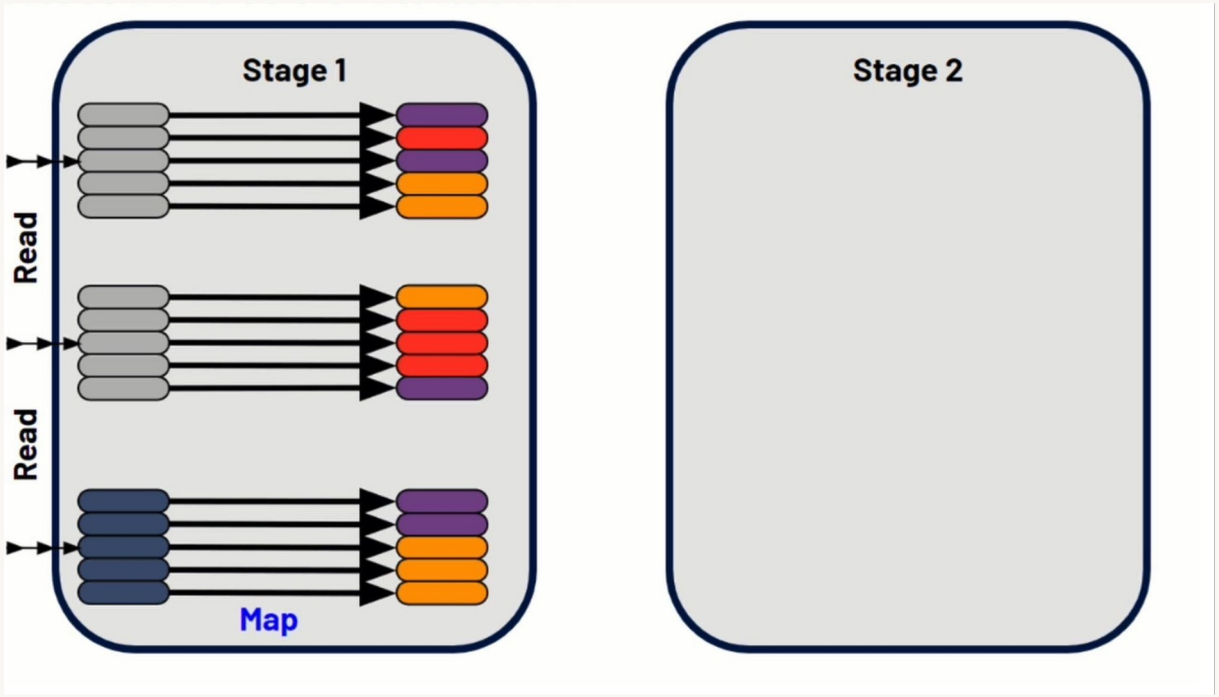
Shuffles at a Glance



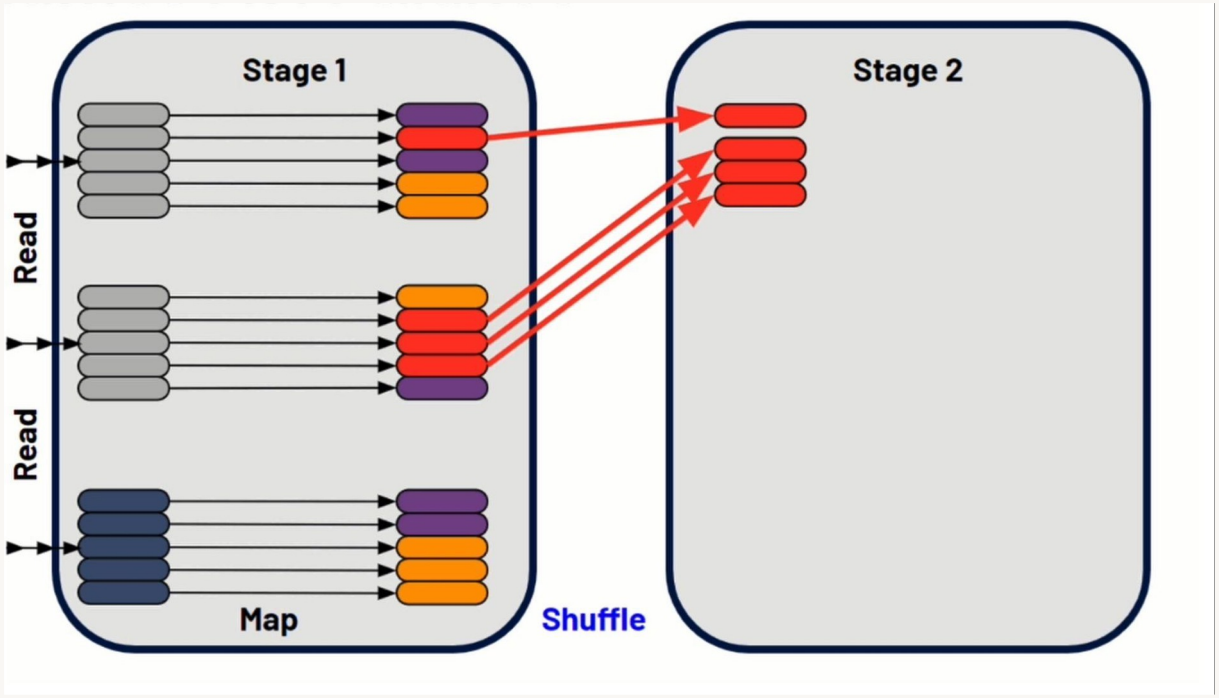
Shuffles at a Glance



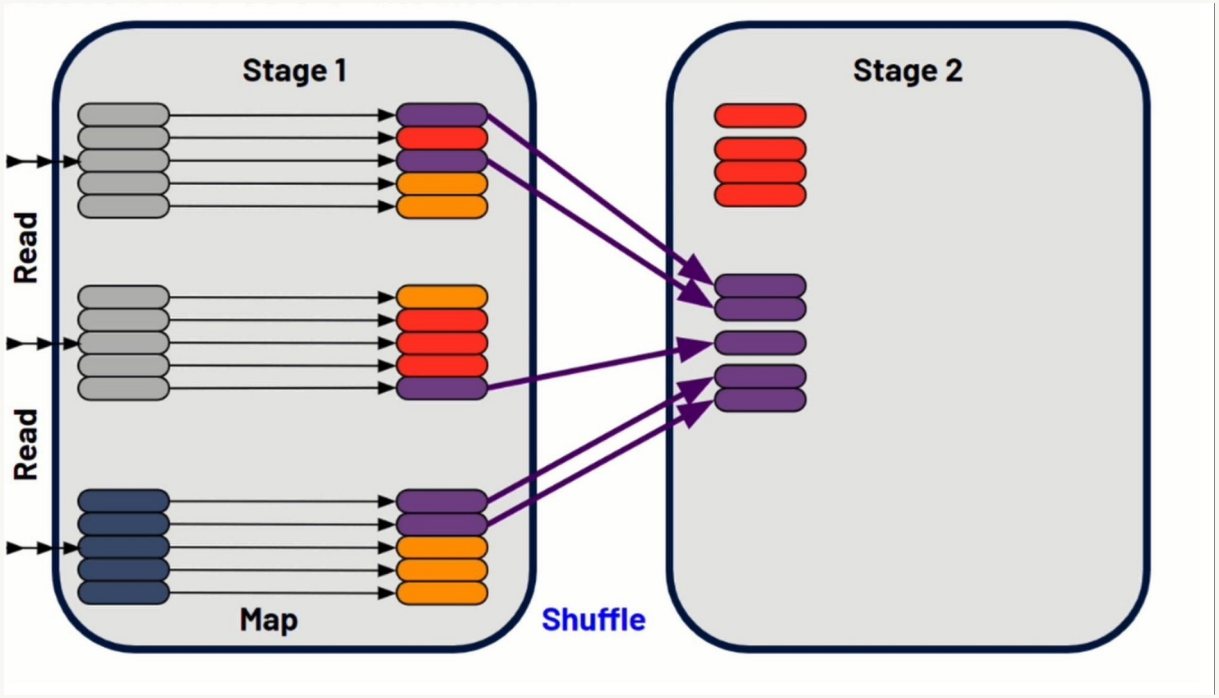
Shuffles at a Glance



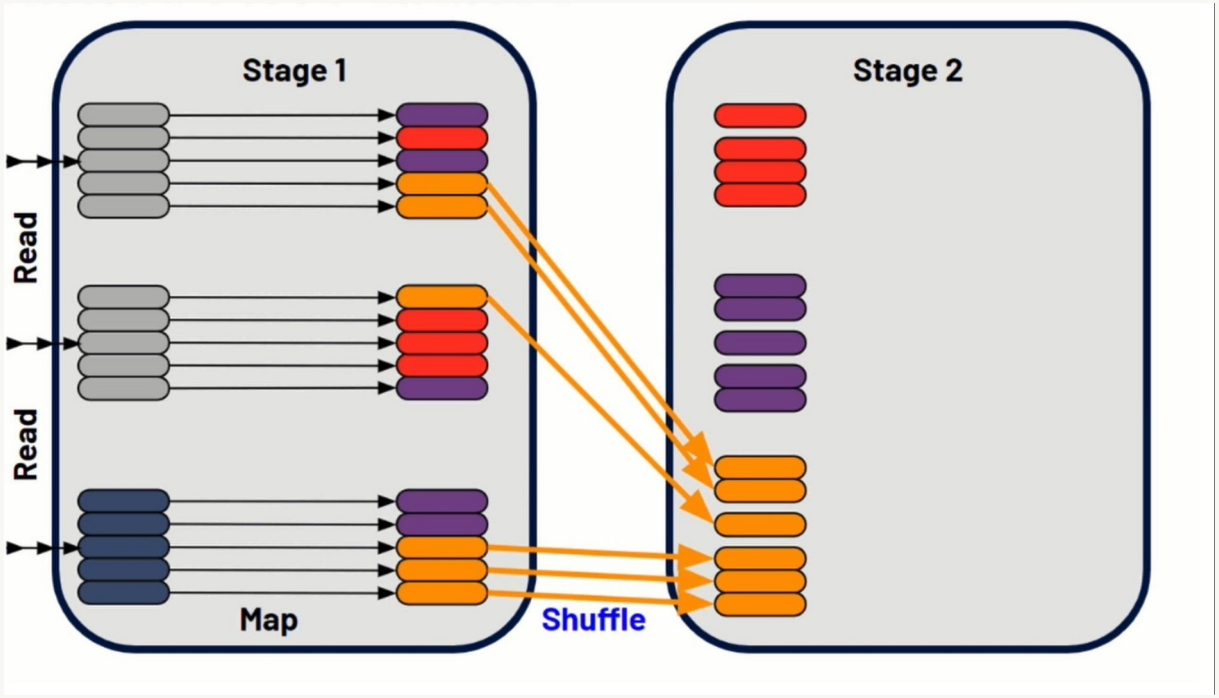
Shuffles at a Glance



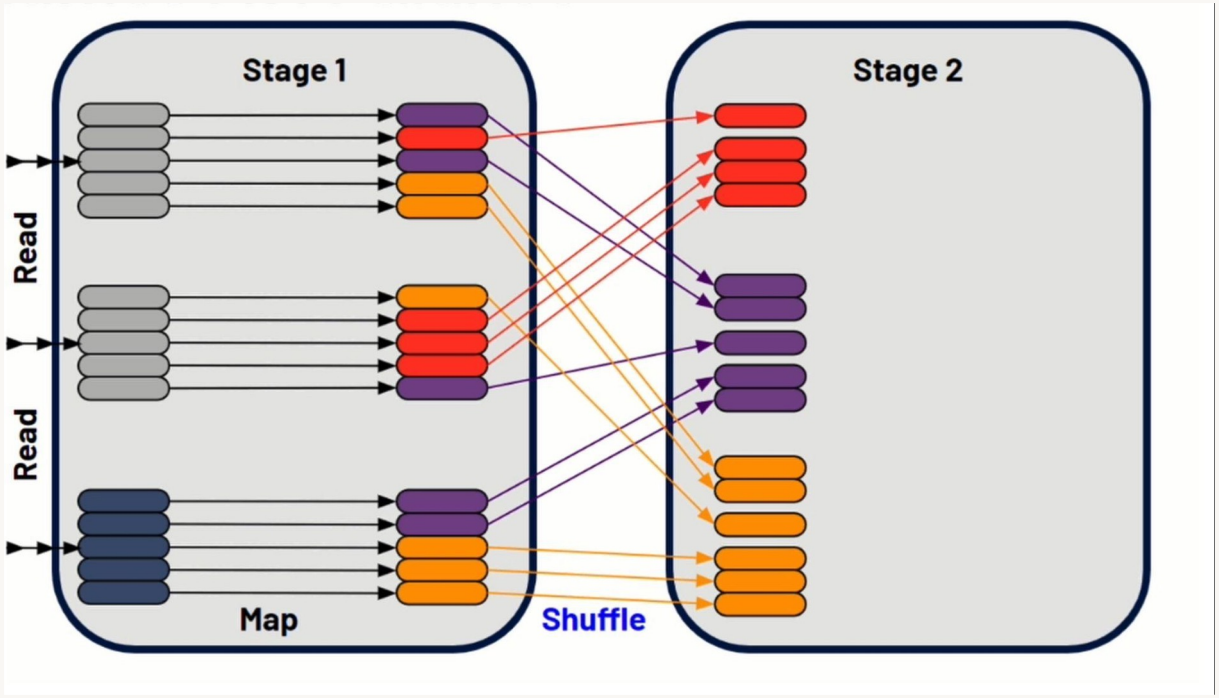
Shuffles at a Glance



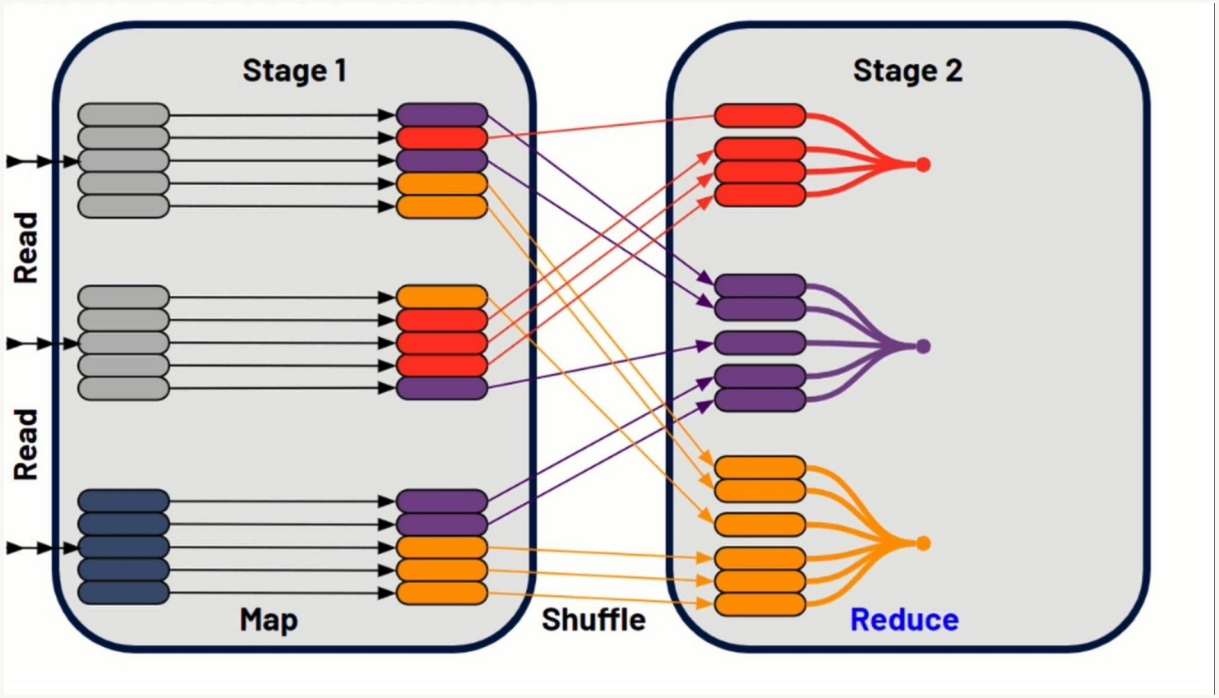
Shuffles at a Glance



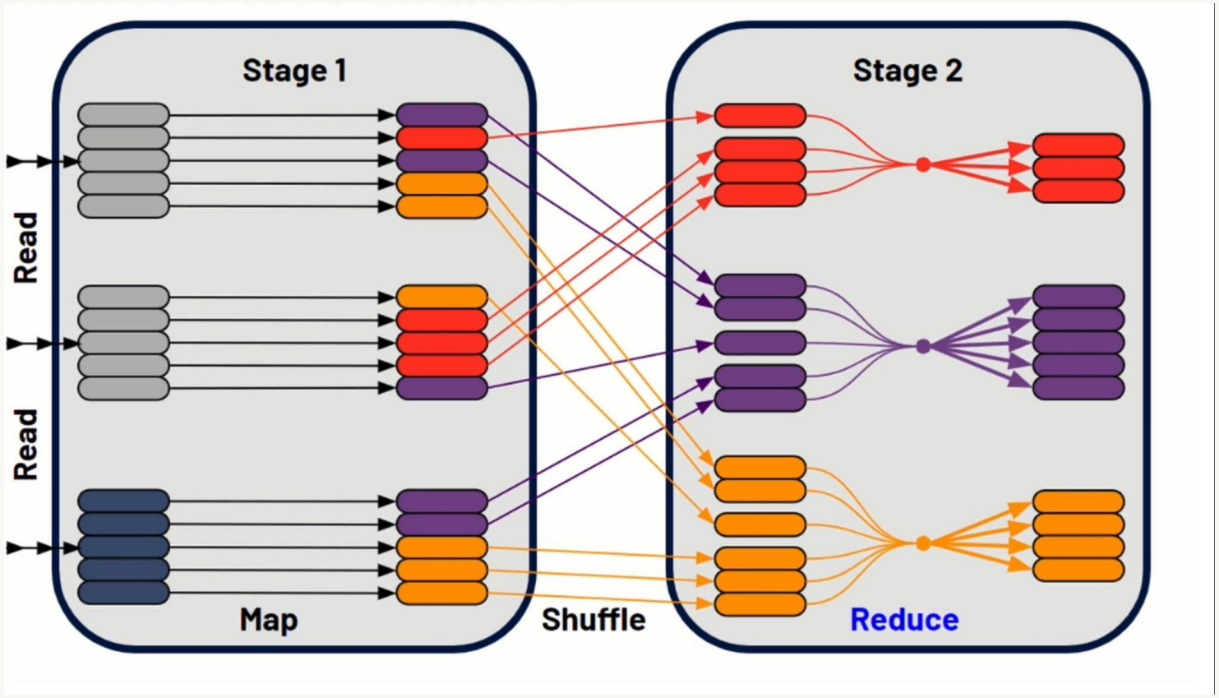
Shuffles at a Glance



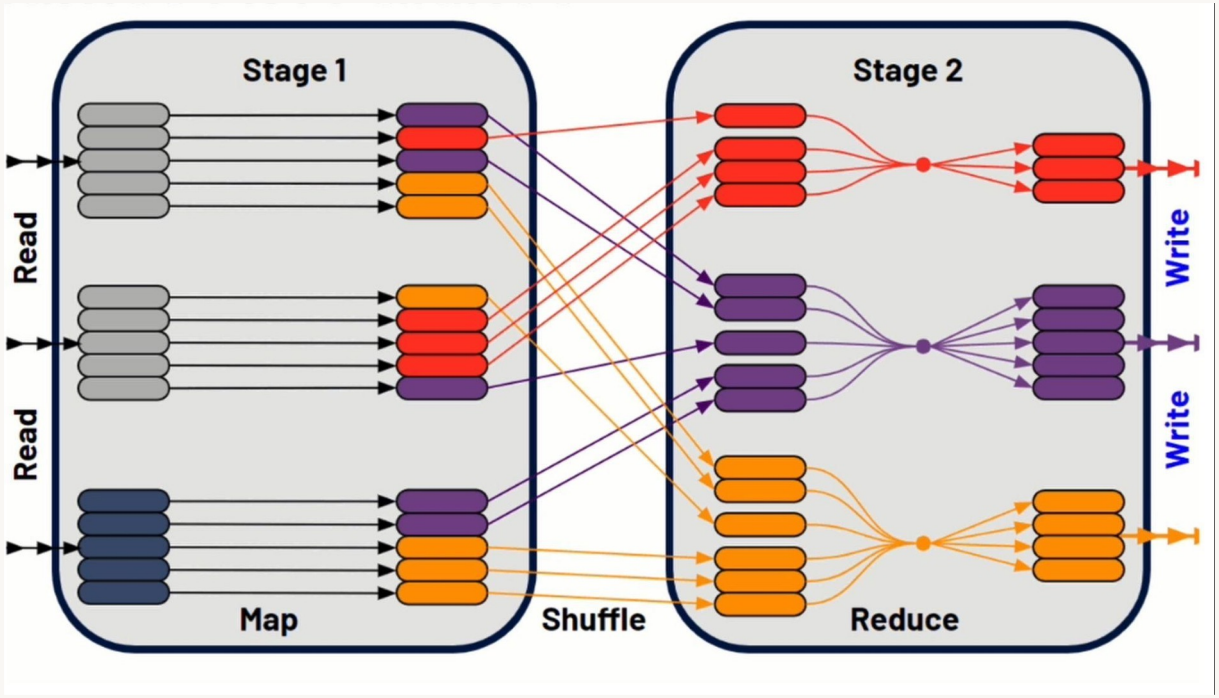
Shuffles at a Glance



Shuffles at a Glance



Shuffles at a Glance



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

Re-evaluate join strategy:

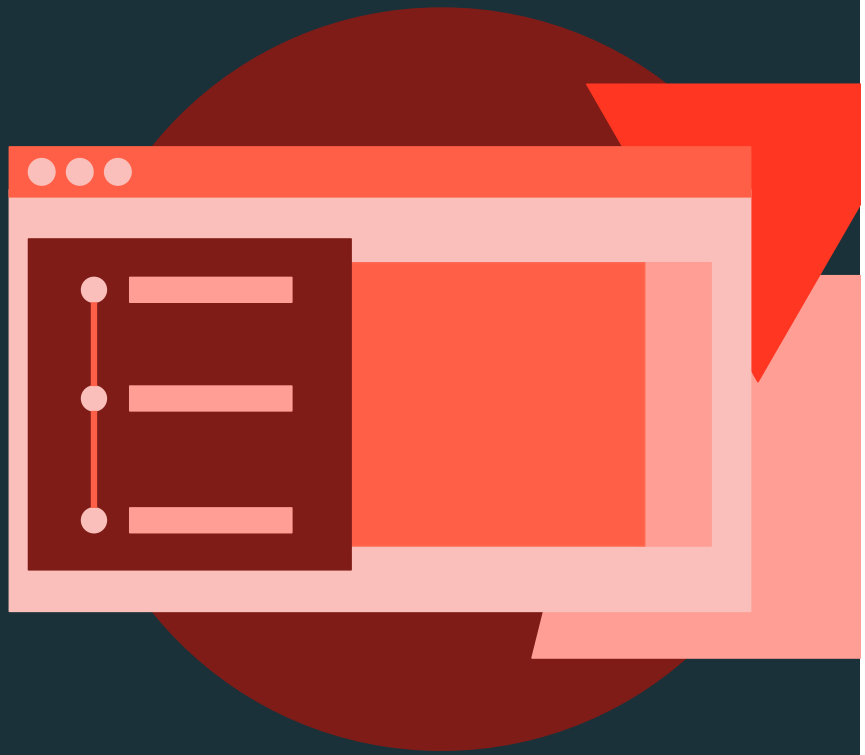
- Reordering the join
- Dynamically Switching Join Strategies
- Broadcast Hash Join
- Shuffle Hash Joins (default for Databricks Photon)
- Sort-Merge Join (default for OS Spark)

databricks

Code Optimization

DEMONSTRATION

Shuffle



databricks

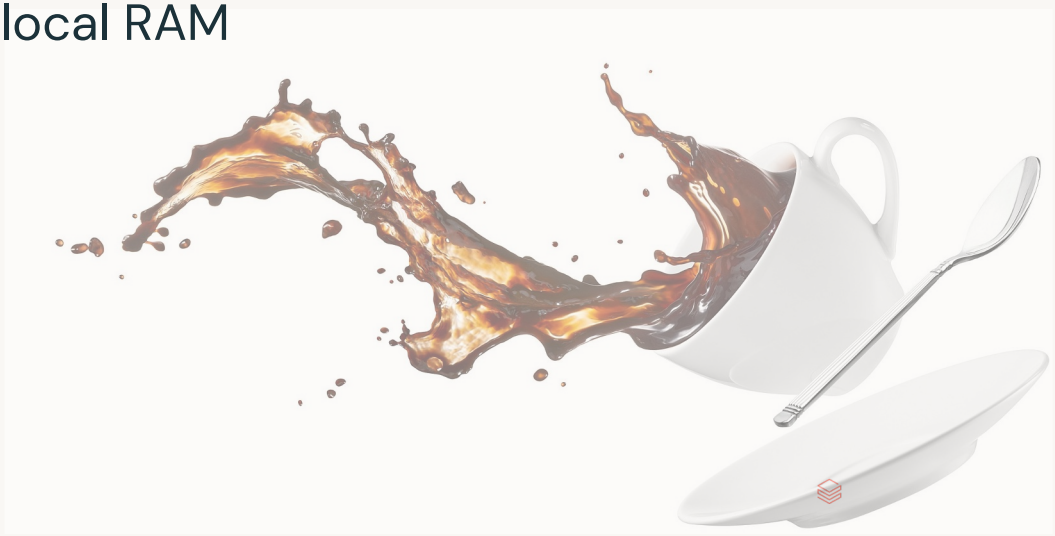
Code Optimization

LECTURE

Spill

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



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()**

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

Spill – Mitigations

- Allocate cluster with more RAM per Core
- Address data skew
- Manage size of Spark partitions
- Avoid expensive operations like explode()
- Reduce amount data preemptively whenever possible

databricks

Code Optimization

LAB EXERCISE

Exploding Join

databricks

Code Optimization

LECTURE

Serialization

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

Mitigating Serialization Issues

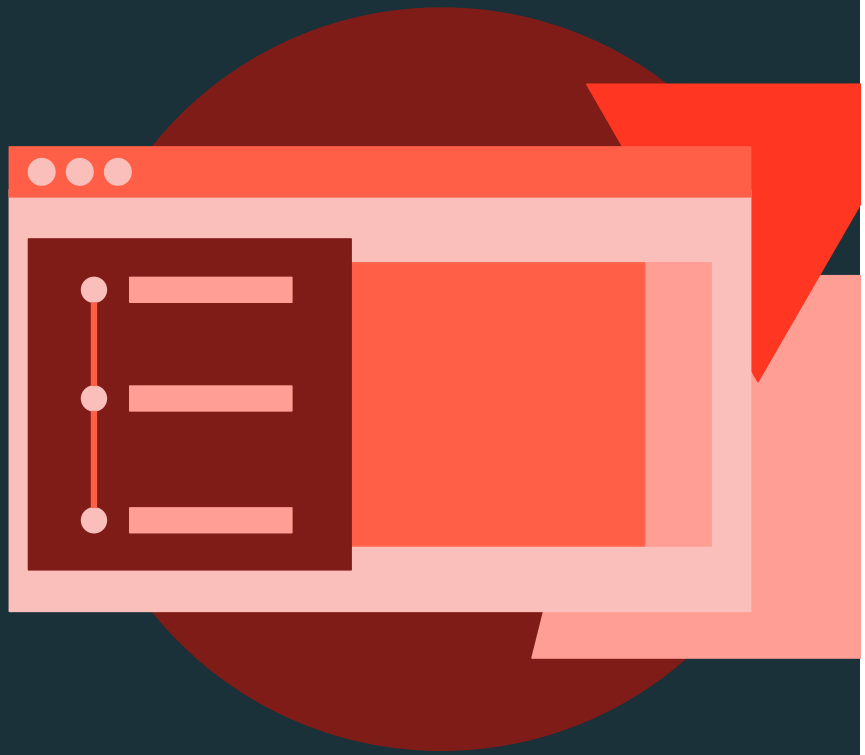
- Don't use UDFs
  - I challenge you to find a set of transformations that cannot be done with the built-in, continuously optimized, community supported, higher-order functions
- If you have to use UDFs in Python (common for Data Scientist) use the Vectorized UDFs as opposed to the stock Python UDFs or Apache Arrow Optimised 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

databricks

Code Optimization

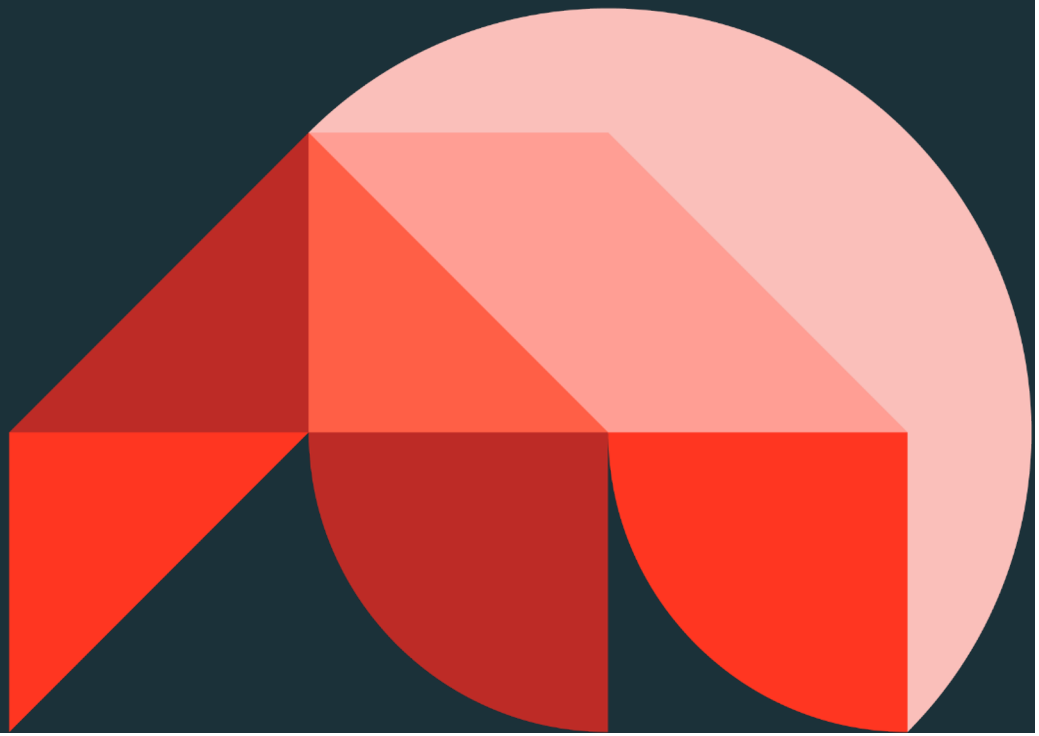
DEMONSTRATION

User-Defined Functions



databricks

Fine-Tuning: Choosing the Right Cluster



Databricks Performance Optimization

databricks

Fine-Tuning: Choosing the Right Cluster

LECTURE

Fine-Tuning: Choosing the Right Cluster

Cluster Types

ALL PURPOSE COMPUTE

- Designed to handle interactive workloads, including streaming workloads.
- Enable Auto-Scale to add capacity when needed and reduce time to answer
- Security considerations must be considered as auto-scaling can introduce additional risks.

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 and lower TCO

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	Auto Scaling 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

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

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

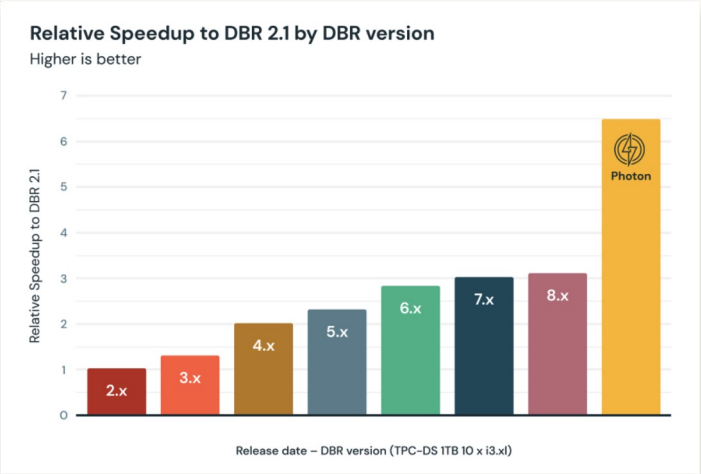
No code changes

- 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



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, 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 the latest LTS Databricks Runtime when possible
    c. Use Photon for best TCO when applicable
    d. Use latest gen VM, start with general purpose, then test memory/compute optimized

databricks

Fine-Tuning: Choosing the Right Cluster

LECTURE

Pick the Best Instance Types

Have an Open Mind When Picking Machines

- For AWS. i3's aren't always the best. Explore m7gd and r7gd

  Enable caching if needed.
  - Graviton instances work well, try those first
  - M7gd and r7gd have better processors, similar (albeit smaller) local disk and much more stable spot markets than i-series
- For Azure, try the eav4, dav4 and f-series over L-series
  - The ACU is very useful
- GCP defaults are pretty good
- Usually don't need network optimized instance types some occasions they help with Photon

How to Choose the Right Machine Is Pretty Simple

- Just a series of IFTTT questions and rules of thumb!
  - Side note - if you 2x the cluster and it runs in 1/2 the time, it costs the same

- Rules of thumb
  - First run: set spark.sql.shuffle.partitions = 2x # of cores
  - Keep total memory available to the machine less than 128gb
  - Number of cores should be a ratio of 1 core to 128mb -> 2gb of *reads* (Some caveats may apply)
  - Avoid setting any other configs at first (don't carry over configs from legacy platforms unless absolutely necessary)

How to Choose the Right Machine Is Pretty Simple

- Rules of thumb

Scan parquet +details
Stages: 655.0

file sorting by size time	2 ms
cache writes size (uncompressed) total (min, med, max)	262.4 MiB (576.8 KiB, 1031.5 KiB, 1085.7 KiB)
time spent in the cache locality manager in milliseconds total (min, med, max)	37 ms (0 ms, 0 ms, 37 ms)
number of files read	1,027
filesystem read data size total (min, med, max)	279.8 MiB (620.5 KiB, 1101.1 KiB, 1155.4 KiB)
cache async file status fetch waiting time total (min, med, max)	0 ms (0 ms, 0 ms, 0 ms)
scan time total (min, med, max)	14.6 m (1.9 s, 3.2 s, 7.1 s)
filesystem read data size (sampled) total (min, med, max)	279.8 MiB (620.5 KiB, 1101.1 KiB, 1155.4 KiB)
filesystem read time (sampled) total (min, med, max)	12.4 m (1.7 s, 2.8 s, 5.5 s)
metadata time	8 ms
size of files read	236.7 GiB

What You Care about with the Instance Type

- Core to Ram ratio
- Processor type
- Local vs remote storage
- Storage medium

Cloud	Family	Core:Ram	Processor	Storage
AWS	c5	1core:2gb	Intel Cascade 3.6 GHz	(d) Local NVME
Azure	f-series	1core:2gb	Intel Xeon 2.4 GHz	Local SSD
GCP	n2-highcpu	1core:1gb	Intel Cascade 3.4 GHz	Local SSD

Sizing a Driver

- Leave it the same size as your worker unless you care about being the absolute cheapest - dont make things more complicated than they need to be.
- Driver typically do very little work in a Spark application. Using a 4-8 core 16-32gb ram driver should be fine for most workloads
- Large commits to delta tables use more memory.
- This suggestion is voided when:
  - Running many streams/concurrent jobs on the same machine
  - Commiting a very large (100k+ files) amount of data to a delta table
  - Collecting large amount of data to the driver to use in Pandas/R

Spot Market Considerations

- The spot market is a great way to save money on infrastructure.
- Each instance type has a different level availability and price savings in each region.
- Example – i3s aren't great, r5d's look a lot better.

Instance Type	vCPU	Memory GiB	Savings over On-Demand*	Frequency of interruption
i3.xlarge	4	30.5	70%	>20%
i3.2xlarge	8	61	70%	>20%
i3.4xlarge	16	122	70%	>20%
r5d.large	2	16	85%	<5%
r5d.xlarge	4	32	85%	<5%
r5d.2xlarge	8	64	69%	<5%
r5d.4xlarge	16	128	80%	5-10%

Reference: https://aws.amazon.com/ec2/spot/instance-advisor/

IFTTT – Step 1

Want to use Photon?

No:
Next slide

Yes:

Cloud	Family
AWS	m6gd/r6gd/i4i m7gd/r7gd
Azure	Edsv4
GCP	n2-highmem n2-standard

IFTTT – Step 2

Is your job an ETL job that uses joins/windows/groupbys/aggregations

No:

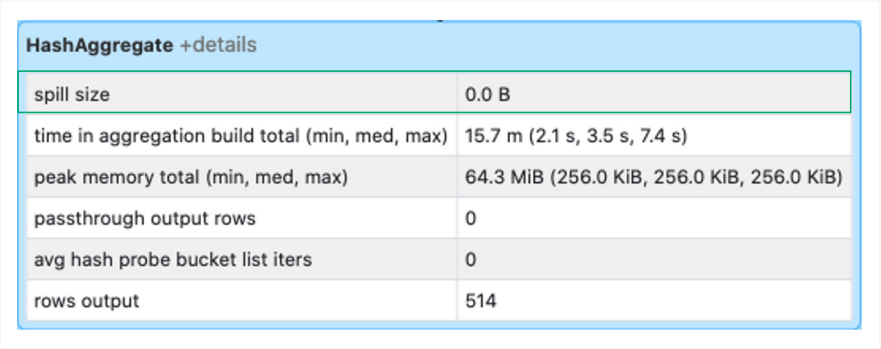
Cloud	Family
AWS	c7g/c6g
Azure	fsv2
GCP	e2-highcpu

Yes:

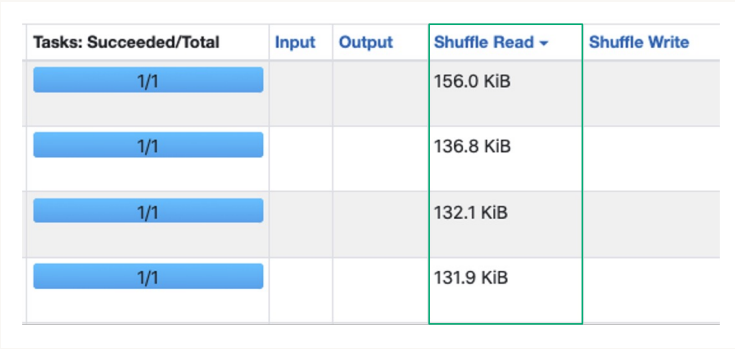
Cloud	Family
AWS	c7gd/c6gd
Azure	fsv2
GCP	n2-highcpu

IFTTT – Step 3

Run the job with the instance type, follow our rules of thumb and go to the SQL UI of the longest running query – do you see spill?



HashAggregate +details	
spill size	0.0 B
time in aggregation build total (min, med, max)	15.7 m (2.1 s, 3.5 s, 7.4 s)
peak memory total (min, med, max)	64.3 MiB (256.0 KiB, 256.0 KiB, 256.0 KiB)
passthrough output rows	0
avg hash probe bucket list iters	0
rows output	514



Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1/1			156.0 KiB	
1/1			136.8 KiB	
1/1			132.1 KiB	
1/1			131.9 KiB	

No:
Stop this is good enough

Yes:
Set spark.sql.shuffle.partitions to the largest shuffle read stage / 200mb
spark.sql.shuffle.partitions=auto

FYI: Spill is much less impactful when using Photon

IFTTT – Step 4

Run the job with the updated shuffle partitions – do you still see spill?

No:
Stop this is good enough

Yes:

Cloud	Family
AWS	m7gd
Azure	dav4/dasv4
GCP	n2-standard

IFTTT – Step 5

Run the job with the updated instance type – do you still see spill?

No:
Stop this is good enough

Yes:

Cloud	Family
AWS	r7gd/r6gd
Azure	Edsv4
GCP	n2-highmen

Reminder on Shuffle Partitions

spark.sql.shuffle.partitions = auto

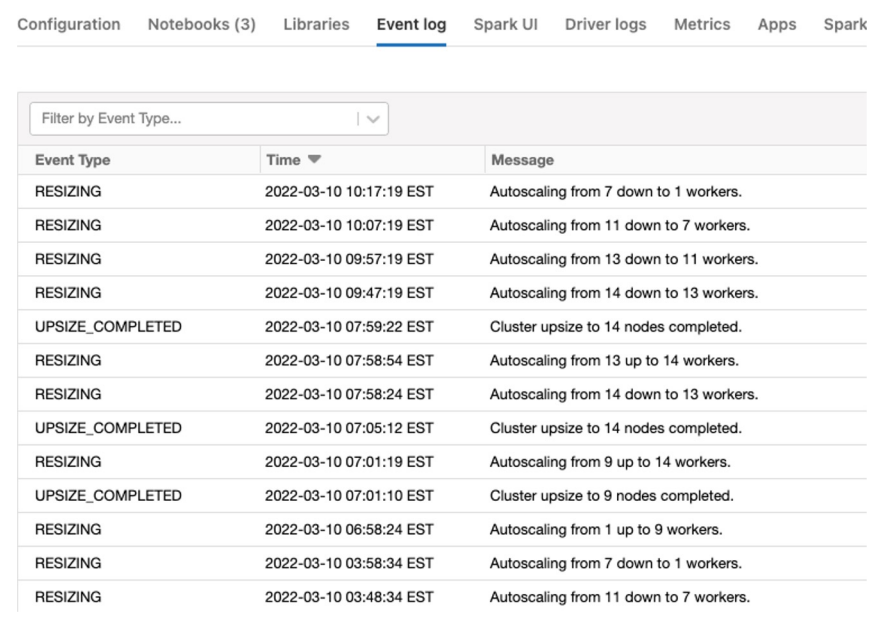
OR

Go to stage UI, find the largest shuffle read size, divide that by 200mb

Tasks: Succeeded/Total	Input	Output	Shuffle Read ▾	Shuffle Write
600/600		368.9 MiB	743.3 MiB	
600/600		368.9 MiB	743.3 MiB	
600/600		368.9 MiB	743.3 MiB	

Don’t Forget to Double Check the Event Log!

Spot failures happen. They slow things down. We know. Don’t forget to double check the event log. It’s probably the first thing you should do.



Event Type	Time	Message
RESIZING	2022-03-10 10:17:19 EST	Autoscaling from 7 down to 1 workers.
RESIZING	2022-03-10 10:07:19 EST	Autoscaling from 11 down to 7 workers.
RESIZING	2022-03-10 09:57:19 EST	Autoscaling from 13 down to 11 workers.
RESIZING	2022-03-10 09:47:19 EST	Autoscaling from 14 down to 13 workers.
UPSIZE_COMPLETED	2022-03-10 07:59:22 EST	Cluster upsize to 14 nodes completed.
RESIZING	2022-03-10 07:58:54 EST	Autoscaling from 13 up to 14 workers.
RESIZING	2022-03-10 07:58:24 EST	Autoscaling from 14 down to 13 workers.
UPSIZE_COMPLETED	2022-03-10 07:05:12 EST	Cluster upsize to 14 nodes completed.
RESIZING	2022-03-10 07:01:19 EST	Autoscaling from 9 up to 14 workers.
UPSIZE_COMPLETED	2022-03-10 07:01:10 EST	Cluster upsize to 9 nodes completed.
RESIZING	2022-03-10 06:58:24 EST	Autoscaling from 1 up to 9 workers.
RESIZING	2022-03-10 03:58:34 EST	Autoscaling from 7 down to 1 workers.
RESIZING	2022-03-10 03:48:34 EST	Autoscaling from 11 down to 7 workers.

Questions?