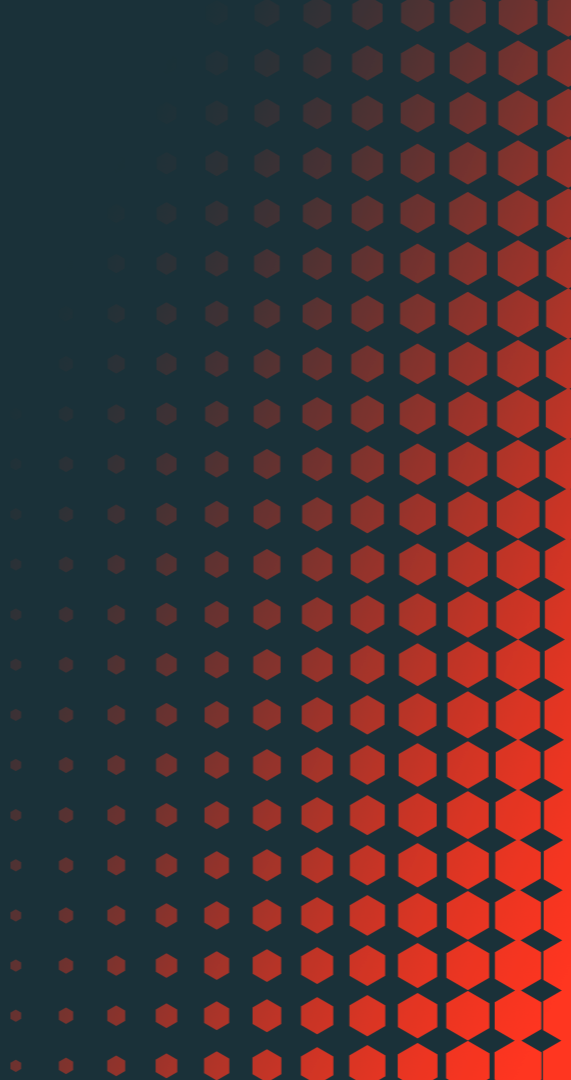
Optimizing Apache Spark:

The Five Most Common Performance Problems - Condensed



databricks

The 5 Most Common Performance Problems (The 5 Ss)

Five Basic Problems

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 (The 5 Ss)

Five Basic Problems - 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

The 5 Most Common Performance Problems (The 5 Ss)

The Spark UI

Before we can get started, we need to establish a baseline

- We need to define terminology
- We need to know how to navigate the Spark UI
- We need to know how to interpret the Spark UI
- Introducing the Spark UI Simulator
  https://www.databricks.training/spark-ui-simulator

The 5 Most Common Performance Problems (The 5 Ss)
Quick Note on Benchmarking

There are generally three common approaches to benchmarking:

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

Count operations are optimized and report significantly shorter times

For-Each operations induce serialization overhead which skews benchmarks

No-Op writes limit execute to ingestion, excludes writes & processes everything

Experiment #5980 illustrates how these strategies can very

Optimizing Apache Spark

The Five Most Common Performance Problems - Condensed

Storage

If you had only 60 seconds to pick up as many coins as you can, one coin at a time, which pile do you want to work from?

$0.13 vs $3.25

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Examples #1

Tiny Files

- See Experiment #8923
- **Step B** with 41 million records clocks in at **~ 3 minutes**
- **Step C** with 2.7 billion records clocks in at **~10 minutes**
- **Step D** with 34 million record clocks in at **~1.5 hours**

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Examples #2

Scanning

- See Experiment #8973, contrast **Step B**, **Step C** and **Step D**
- **Step B**: 1 directory, 345K files clocks in at **~1 minute**
- **Step C**: 12 directories, 6K files clocks in at **~5 seconds**
- **Step D:** 8K directories, 6k files clocks in at **~14 minutes**

The 5 Most Common Performance Problems (The 5 Ss)

Storage - Example #3

Inferring Schemas & Merging Schemas

- Inferring schemas for JSON & CSV requires a full read of 100% of the data even when filtered - **SLOWEST**
- For Parquet the schema is read from a single part file - **FAST**
- For Parquet w/Schema Merging requires a read of the schema from all part-files - **SLOW**
- For Delta the schema is read from the transaction log - **FASTER**
- For tables, the schema is read from the meta store - **FASTEST**

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Example #4

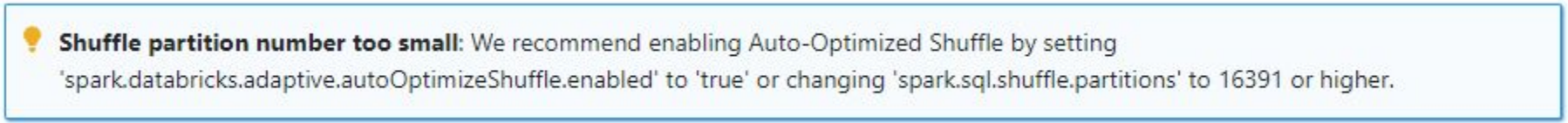
NVMe & SSD are up to 10x faster than Magnetic Drives

- Does not apply to traditional data reads - those come from cloud storage
- Applies to cluster-local disk operations like reading and writing shuffle files and Delta/Parquet IO Cache
- See Experiment #5136A for HDD: clocks in at **~1.5 hours** (SWT == 3 & 4.5 hr)
- See Experiment #5136B for SSD: clocks in at **~1 hour** (SWT == 2.5 & 1.5 hours)
- See Experiment #5136C for NVMe: clocks in at **~45 minutes** (SWT == 0 min)

SW* = Spill Write Time found in the Query Details

The 5 Most Common Performance Problems (The 5 Ss)
Mitigating Storage Issues

- Use Delta + Unity Catalog - Delta provides it's own optimizations and reading table schema from the catalog always outperforms reading from disk
- Use NVMe & SSD for faster disk IO - further helps to mitigate shuffle
- Employ AQE and **spark.sql.adaptive.coalescePartitions.enabled** to control your spark-partitions to minimize the production of tiny files
- Employ **spark.databricks.adaptive.autoOptimizeShuffle.enabled**



The 5 Most Common Performance Problems (The 5 Ss)
Mitigating Storage Issues, Cont'

- Use Delta's OPTIMIZE operation to compact tiny files
- Consider enabling Delta's Auto Compaction and Optimized Writes
- Consider employing Databricks' Auto Loader

Optimizing Apache Spark

The Five Most Common Performance Problems - Condensed

Serialization

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

- 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
- Python UDFs takes an even hard 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

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

- In addition to that…
- UDFs 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 optimizations are limited 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 Is It?

- See Experiment #4538 for Scala
  - **Step D** uses higher-order functions & clocks in at **~23 minutes** (same as Python)
  - **Step E** uses Scala UDFs & clocks in at **~36 minutes**
  - **Step F** uses Scala's Typed Transformations & clocks in at **~26 minutes**

- See Experiment #4538 for Python
  - **Step D** uses higher-order functions & clocks in at **~23 minutes** (same as Scala)
  - **Step E** uses Python UDFs & clocks in at **~1 hour & 45 minutes**
  - **Step F** uses Python and Vectorized UDFS & clocks in at **~1 hour & 20 minutes**

The 5 Most Common Performance Problems (The 5 Ss)

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

Optimizing Apache Spark

The Five Most Common Performance Problems - Condensed

Skew

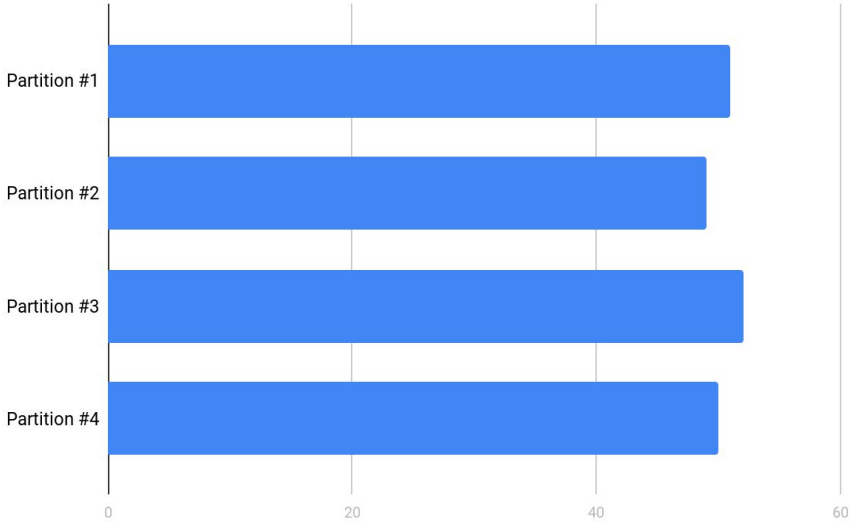
The 5 Most Common Performance Problems (The 5 Ss)
Skew

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

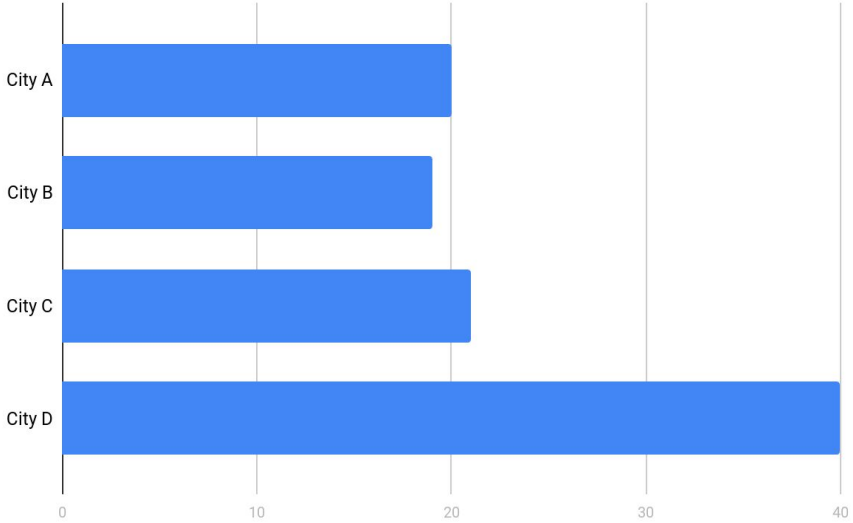
The 5 Most Common Performance Problems (The 5 Ss)

Skew - Before & After

Before aggregation



After aggregation by city



The 5 Most Common Performance Problems (The 5 Ss)

Skew - Ramifications

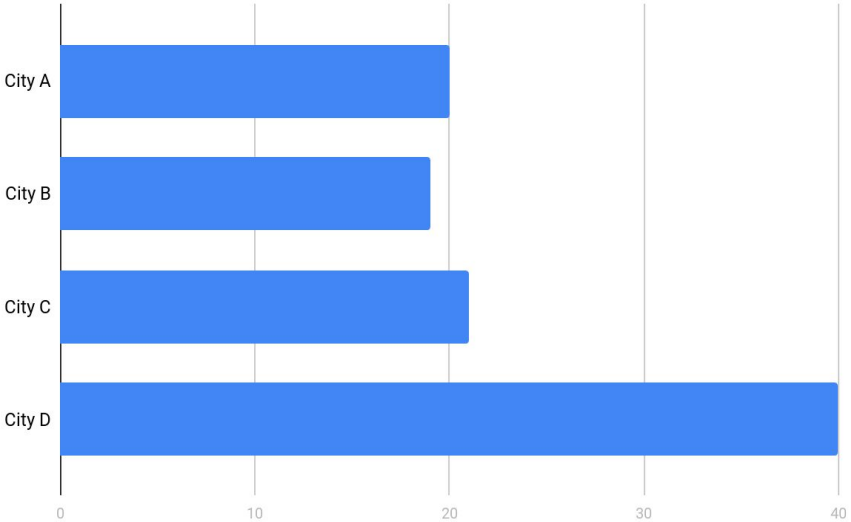
If **City D** is 2x larger than A, B or C...

- 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



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

There are three "common" solutions:

1. 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

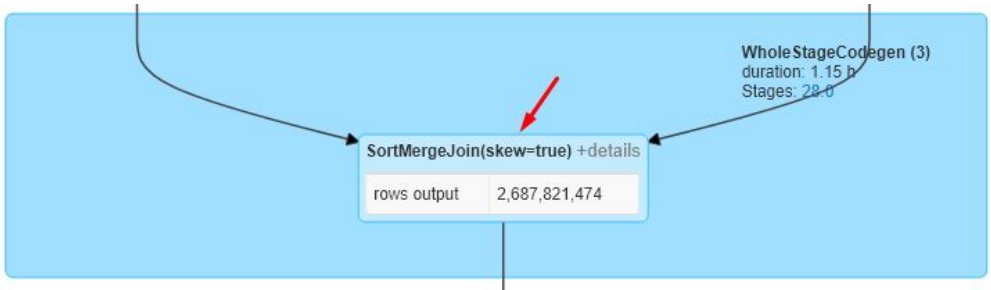
2. Databricks' [proprietary] Skew Hint
   - Easier to add a single hint than to salt your keys
   - Great option for version of Spark 2.x

3. Adaptive Skew Join (enabled by default in Spark 3.1)
   - Outperforms the other two options and is enabled by default!

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

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 Experiment #1596, **Step C**, **Step D**, **Step E**, and **Step F**
- Especially the SQL diagram for **Step E** showing **skew=true**



Optimizing Apache Spark

The Five Most Common Performance Problems - Condensed

Spill

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
- Possibly the most significant contributor to poorly performing Spark jobs

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

There are a number of ways to induce this problem:

- Mismanagement of **spark.sql.shuffle.partitions** (default is 200)
- The **explode()** of even a small array
- The **join()**, or worse, **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

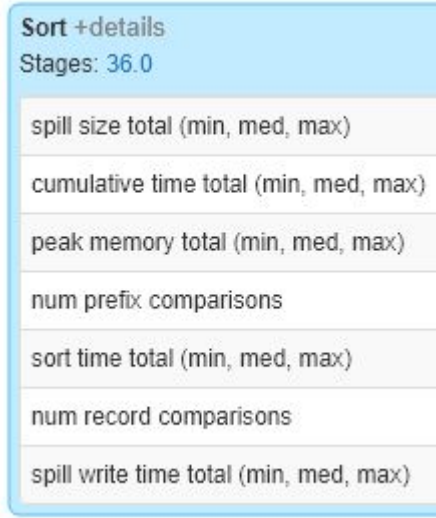
The two values are always presented together

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 shown on the Stage Details page under:
  - **Summary Metrics**
  - **Aggregated Metrics by Executor**
  - The **Tasks** table



- Or in the corresponding query details
- 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 Listener To The Rescue!

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**
- While written in Scala, Databricks ability to mix Scala and Python in a single notebook means anyone can employ this little tool

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

- 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

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

- The quick answer: allocate a cluster with more memory per worker
- 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 setting **spark.sql.files.maxPartitionBytes** lower than the default 128MB
  - Not an effective strategy against skew
- Ignore it - consider the example in **Step E**.
  - Out of ~800 tasks only ~50 tasks spilled
  - Is that 6% worth your time?

Optimizing Apache Spark

The Five Most Common Performance Problems - Condensed

Shuffles

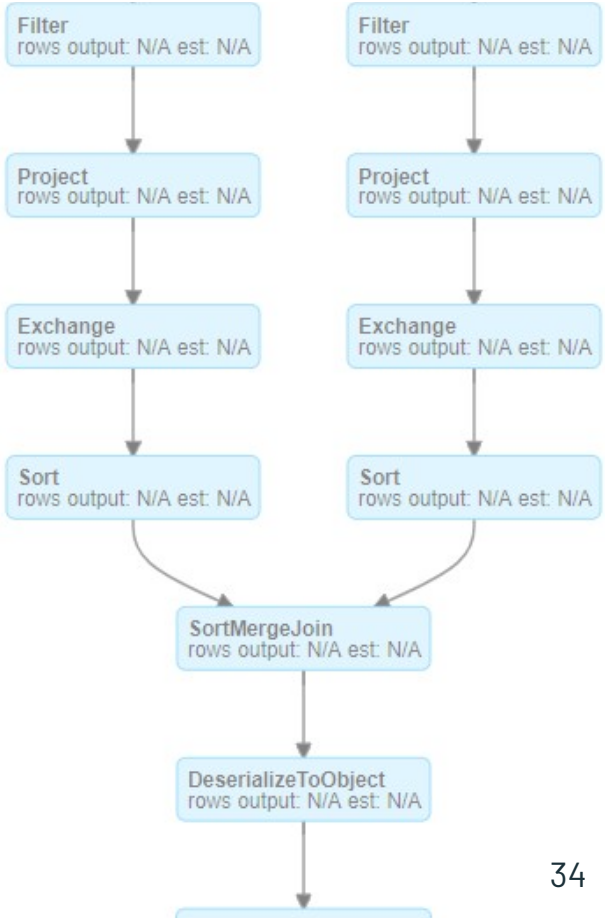
The 5 Most Common Performance Problems (The 5 Ss)

Shuffle

Shuffling is a side effect of wide transformation:

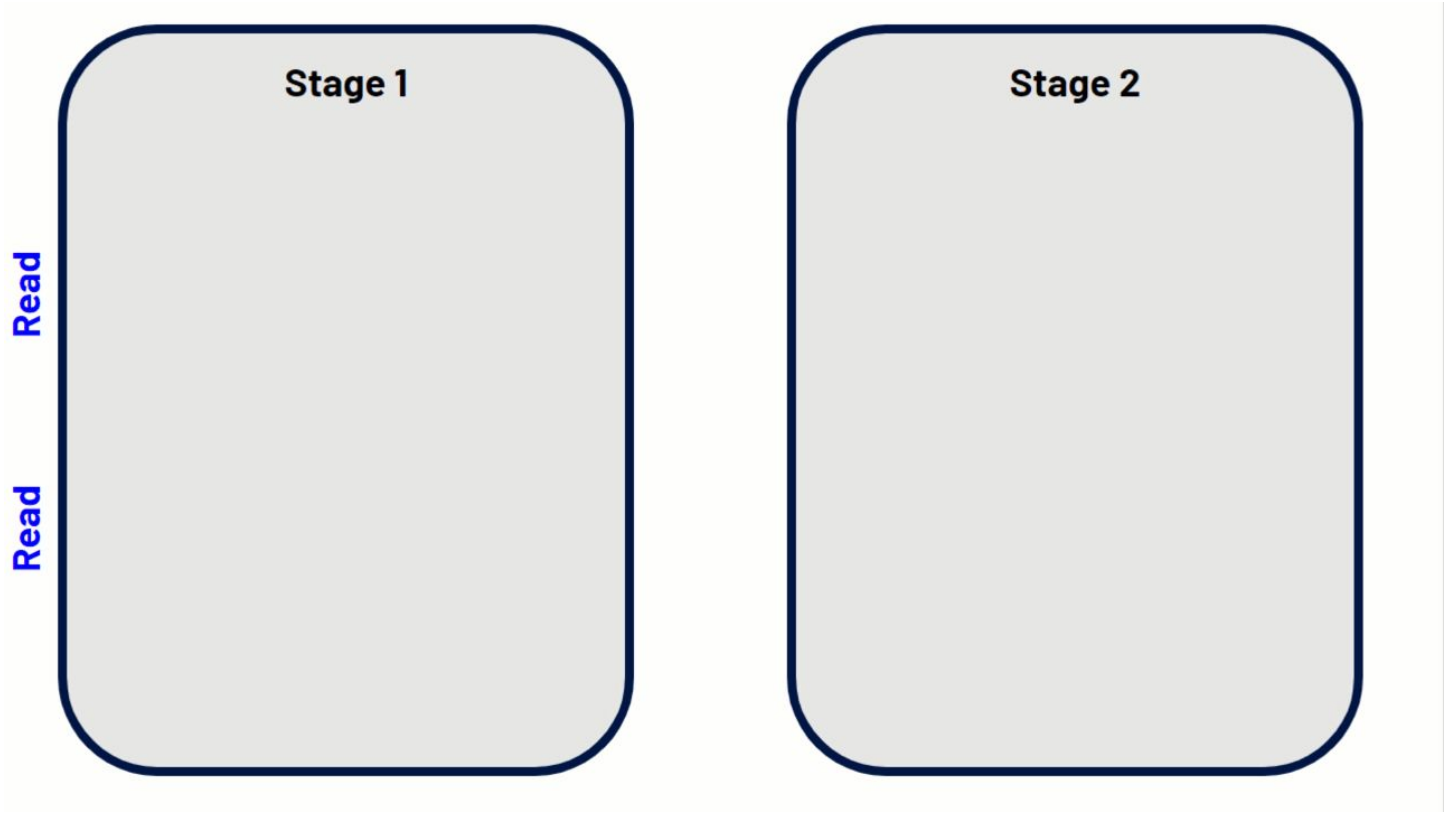
- **`join()`**
- **`distinct()`**
- **`groupBy()`**
- **`orderBy()`**

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



The 5 Most Common Performance Problems (The 5 Ss)

Shuffles At A Glance



The 5 Most Common Performance Problems (The 5 Ss)
Shuffle - Being Pragmatic

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

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

- Reduce network IO by using fewer and larger workers
- Use NVMe & SSDs so that the shuffle reads and writes are faster
- Reduce the amount of data being shuffled
  - Remove any unnecessary columns
  - Preemptively filter out unnecessary records
- Denormalize the datasets - especially when the shuffle is rooted in a join
- Reevaluate your join strategy - the default is not always the best

The 5 Most Common Performance Problems (The 5 Ss)
Optimizing Joins

We have a number of different options we can explore here:

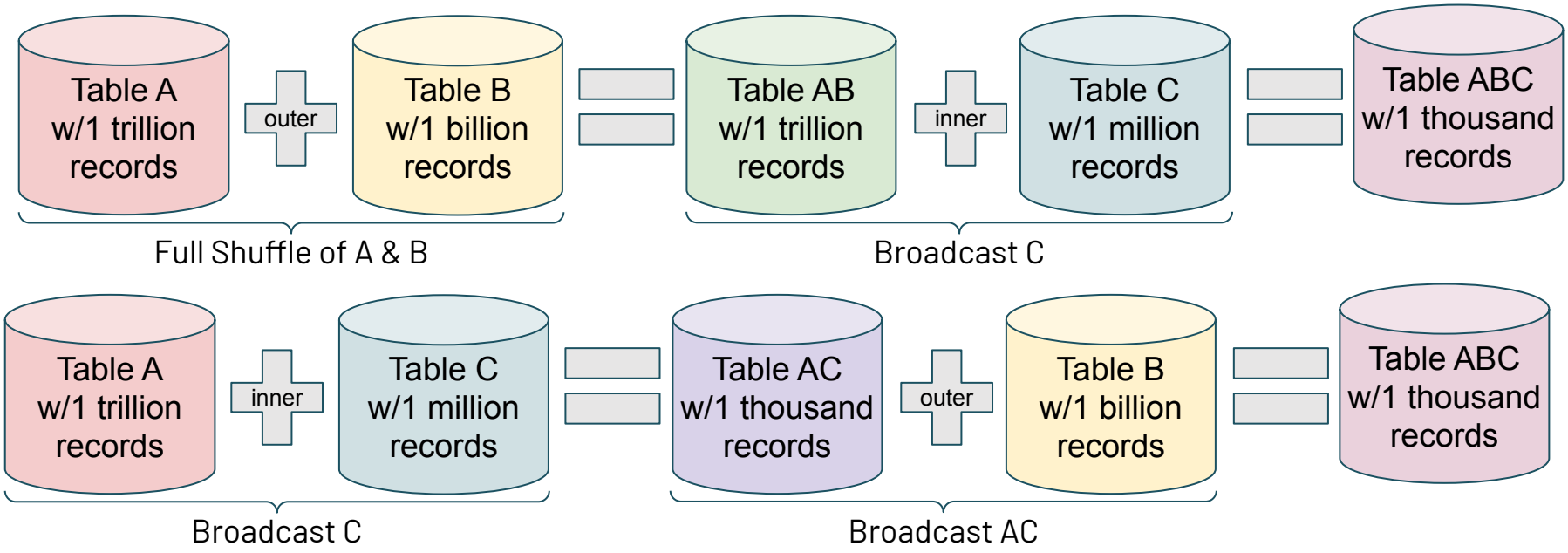
- Reordering the join
- Bucketing
- Broadcast Hash Join
- Shuffle Hash Joins (default for Databricks Photon)
- Sort-Merge Join (default for OS Spark)

The 5 Most Common Performance Problems (The 5 Ss)

Optimizing Joins - Reordering

Mostly Automatic w/AQE & CBO

If we join three tables, it logically makes more sense to order the joins such that we reduce the number of records involved in each shuffle



The 5 Most Common Performance Problems (The 5 Ss)
Optimizing Joins - Bucketing

- "If you are bucketing datasets, you are doing it wrong" - DT
- Bucketing is hard to get right and is an expensive operation to being with...
  ...especially if you are bucketing a periodically changing dataset
- Eliminates the sort in the Sort-Merge Join by pre-sorting partitions
- The cost is paid in production of the dataset on the assumption that savings will be made by frequent joins of both tables
- Not worth considering for datasets less than 1-5 terabytes

The 5 Most Common Performance Problems (The 5 Ss)

Optimizing Joins - Broadcast Hash Join

- Minimize the shuffle operation by sending the broadcasted table to each executor resulting in an executor-local join w/o the expense of SMJ or HJ
- By default, applied to tables under 10 MB
  - AQE & CBO together can detect when a predicate pushes a dataset below the prescribed threshold
  - See **spark.sql.autoBroadcastJoinThreshold** and **spark.databricks.adaptive.autoBroadcastJoinThreshold**

The 5 Most Common Performance Problems (The 5 Ss)
Optimizing Joins - Broadcast Hash Join

- BHJ put extra pressure on the Driver, possibly resulting in OOM Errors
- All fragments of the table are collected in the driver from each executor
- After the table is reassembled, it is redistributed to each of the executors
- For small tables, this overhead is significantly lower than a SMJ or HJ
- It's not uncommon to broadcast tables as large as 1GB (vs the 10MB default)
  - There is no magic formula for this, you just have to experiment with it
  - Make sure that the Driver and Executors are capable of possessing the table

The 5 Most Common Performance Problems (The 5 Ss)
Sort-Merge Join (SMJ) vs Hash Join (HJ)

- When we cannot employ a BHJ, we default to a SMJ. Why?
- Because SMJs, while comparatively slow, scale REALLY well
- In most cases SMJs are also slower than HJ so why default to SMJ?
  - It takes less time to create the hash than it does to execute the sort
  - Both scenarios still shuffle the data
  - But, HJs requires the data to fit into memory == potential OOM Errors
  - Compared to SMJs which are, for the most part, OOM-Proof
- But wait! With Databricks Photon, the default is to use the HJ
- See **spark.sql.join.preferSortMergeJoin**


databricks