databricks

Databricks for Practitioners

Databricks Performance Tuning and Cost Optimizations

William Householder - Solutions Architect

Housekeeping

- This presentation will be recorded and we will share these materials after the session.
- There are no hands-on components so you only need something to take notes.
- Use the Q&A function to ask questions.
- Please fill out the survey at the end of the session so we can improve our future sessions.

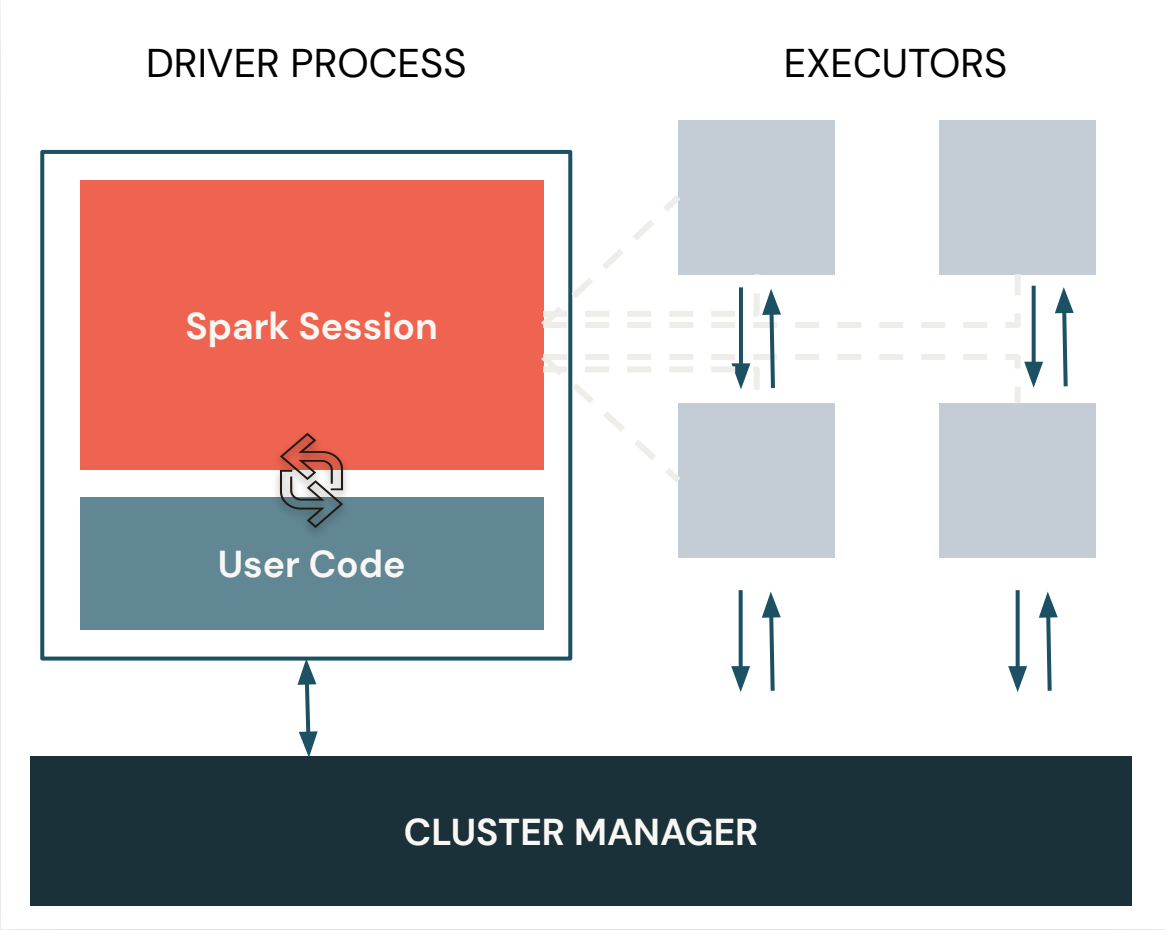
Agenda

- Intro to Spark Architecture
- 5 Key Areas of Performance
  - Skew
  - Spill
  - Storage
  - Shuffle
  - Serialization
- General Optimization Best Practices

Intro to Spark

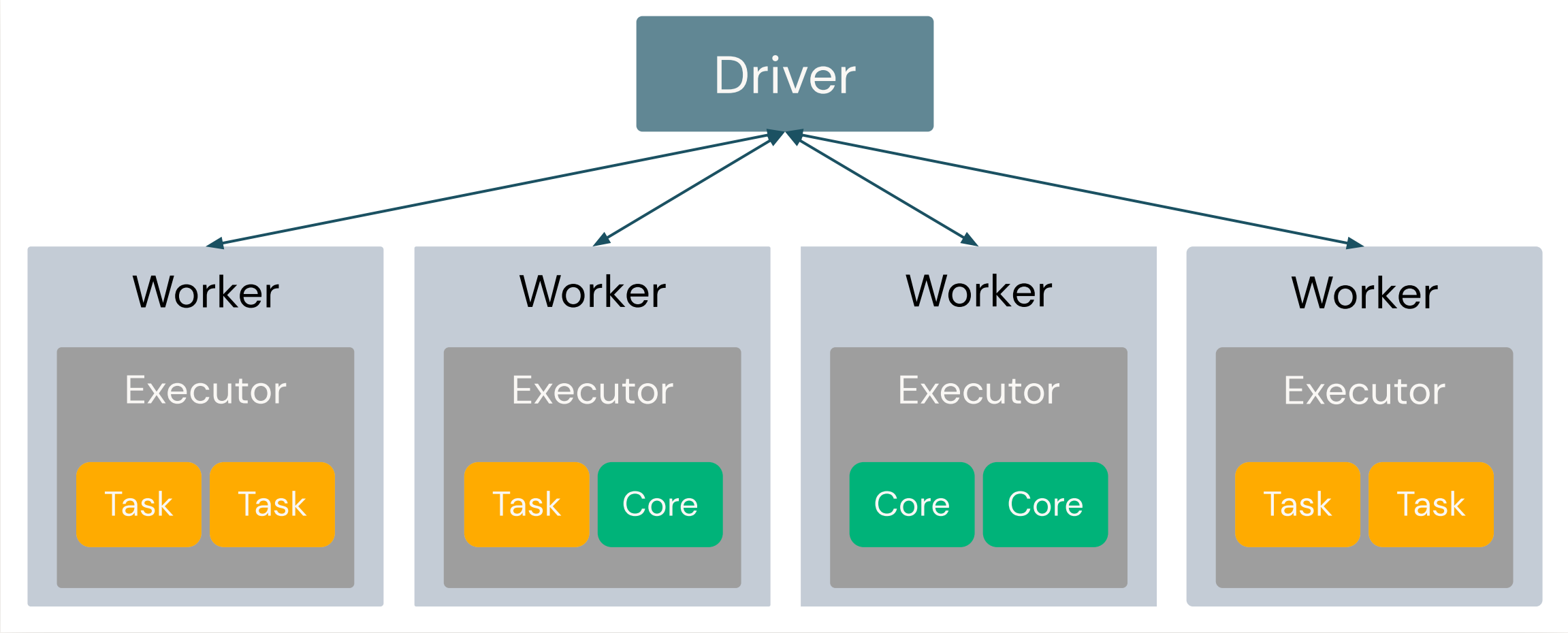
Basic Architecture

Apache Spark's Distributed Ecosystem

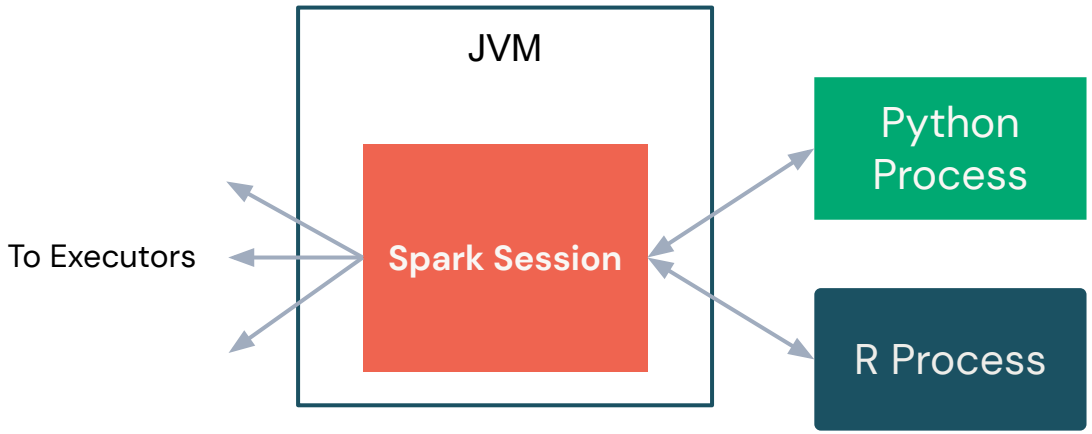


- The cluster manager controls physical machines and allocates resources to Spark Applications.
- The driver process is responsible for executing our driver program's commands across the executors in order to complete our task.
- Executors, will be running the Spark code.

Spark Cluster



Spark Session



- The SparkSession is the single entry point to all DataFrame API functionality
- Automatically created in a Databricks notebook as the variable spark

5 Most Common Performance Problems

The 5 Most Common Performance Problems

The most impactful problems fall into one of five categories:

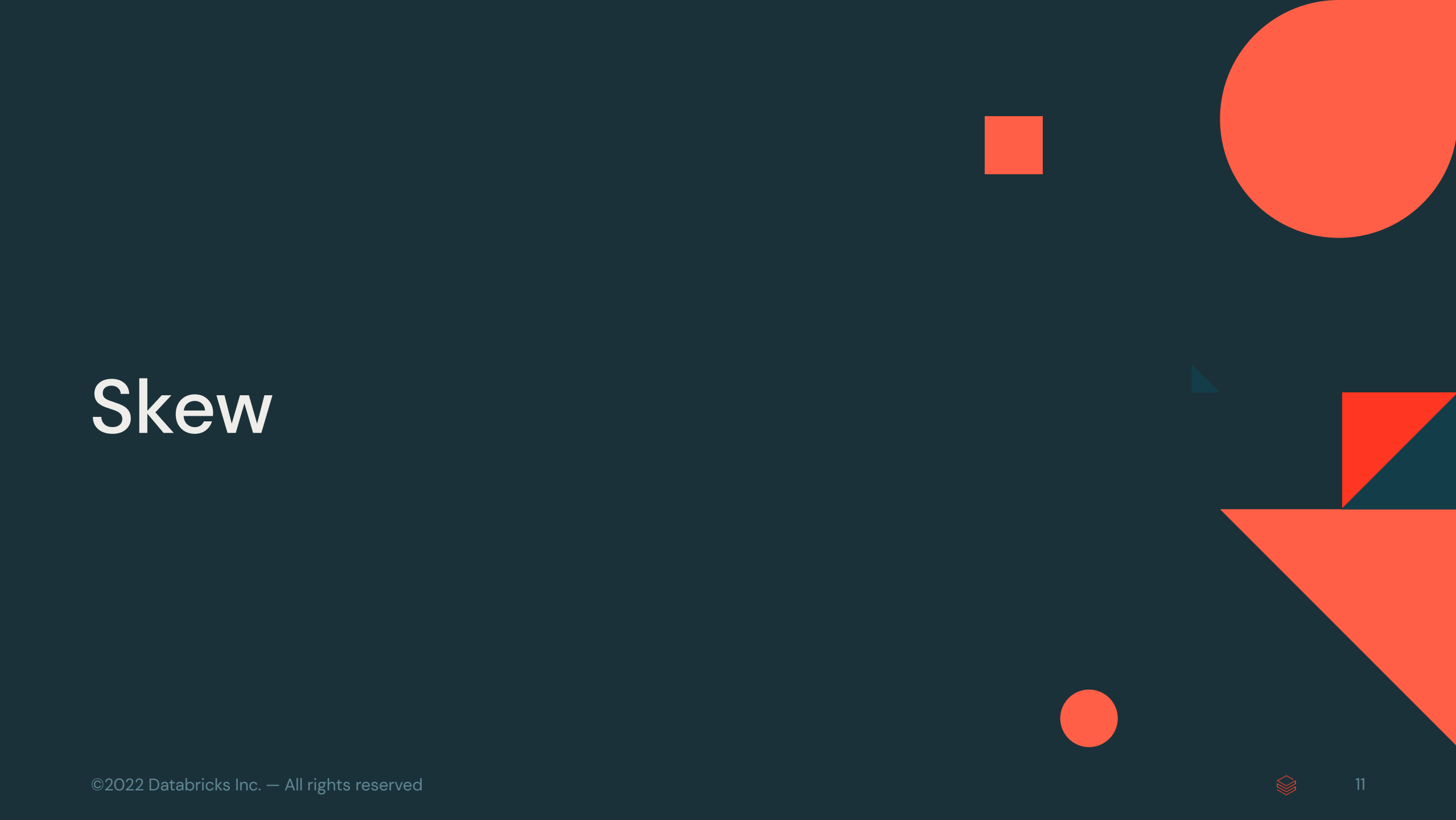
- **Skew**: An imbalance in the size of partitions
- **Spill**: The writing of temp files to disk due to a lack of memory
- **Shuffle**: The act of moving data between processes
- **Storage (Small Files)**: A set of problems directly related to how data is stored on disk
- **Serialization**: How byte streams are converted to data objects and visa versa for storage, transmission and processing.

The 5 Most Common Performance Problems

Why they can be hard to identify

- Distributed processing adds layers of complexity to troubleshooting
- Root sourcing problems is hard when one problem can causes another
- Many of these problems can be present at the same time
  - **Skew** can induce **Spill**
  - **Storage** issues can induce excess **Shuffle**
  - Incorrectly addressing **Shuffle** can exacerbate **Skew**
  - **Serialization** issues are often hidden and mistaken as other problems

Skew



Skew – Identifying and avoiding

- Data is typically read in as 128 MB partitions and evenly distributed - this is partially controlled by **`maxPartitionBytes`**
- As the data is transformed (e.g. filtered), it's possible to have significantly more records in one partition than another
- A small amount of skew is ignorable and present in most datasets
- Large skews can result in spill or worse, hard to diagnose application failures

Skew - Impacts

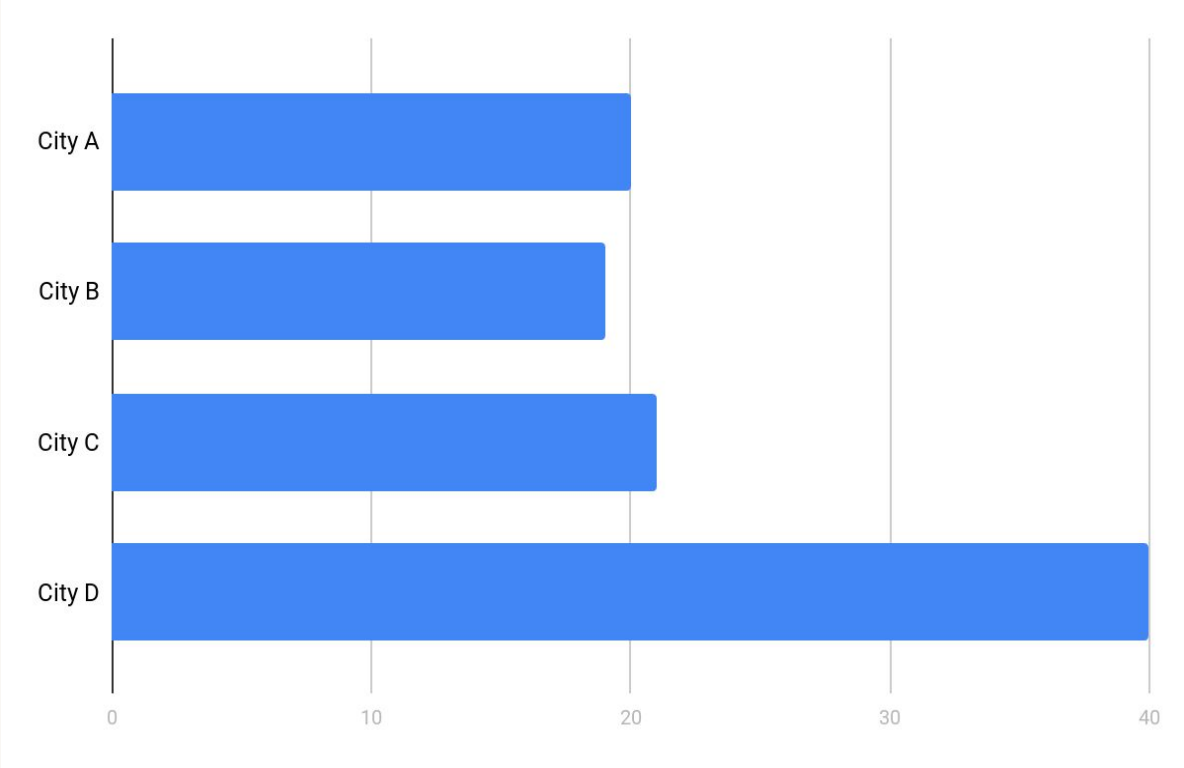
If **City D** is 2x larger than A, B or C...

- It takes 2x as long to process
- It requires 2x as much RAM

The effect is:

- The entire stage will take 2x long to run as we wait for the slowest task
- We may not have enough RAM for the skewed partition

After aggregation by city



Skew - Mitigation

There are several strategies for fixing skew:

- Employ a Databricks-specific skew hint
  (*see Skew Join optimization*)
  - `SELECT /*+ SKEW('orders') */ * FROM orders, customers WHERE c_custId = o_custId`

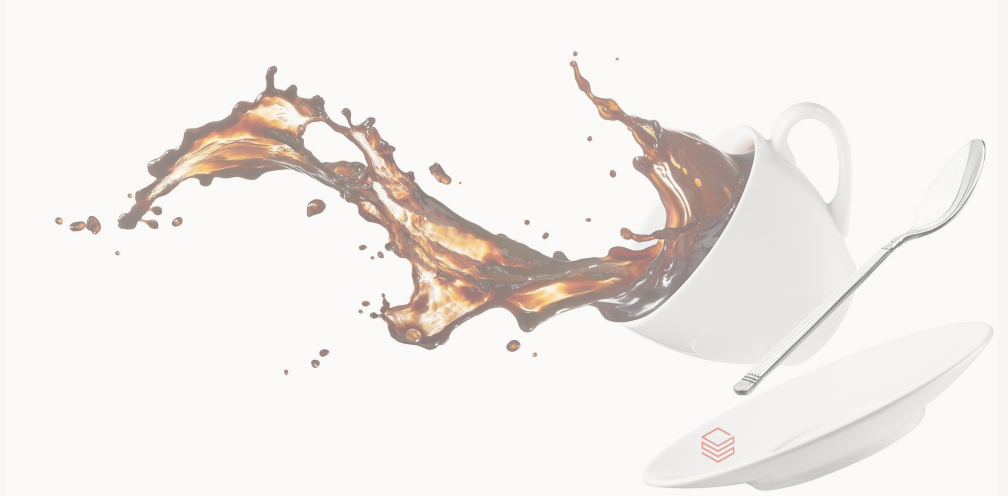
- Enable Adaptive Query Execution in Spark 3

- Salt the skewed column with a random number creating better distribution across each partition at the cost of extra processing (most complicated approach)

Spill

Spill

- Spill is the term used to refer to the act of moving an Partition 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 order to avoid OOM errors
- In this case, Spark is forced into [potentially] expensive disk reads and writes to free up local RAM



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

  - 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
    Note: Only columns associated with the Spill will be displayed in the UI

Spill - Mitigation

- If the root cause is Skew that should be addressed first
- The quick answer: allocate a cluster with more memory per worker
- 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**

Note: this is not an effective strategy against skew

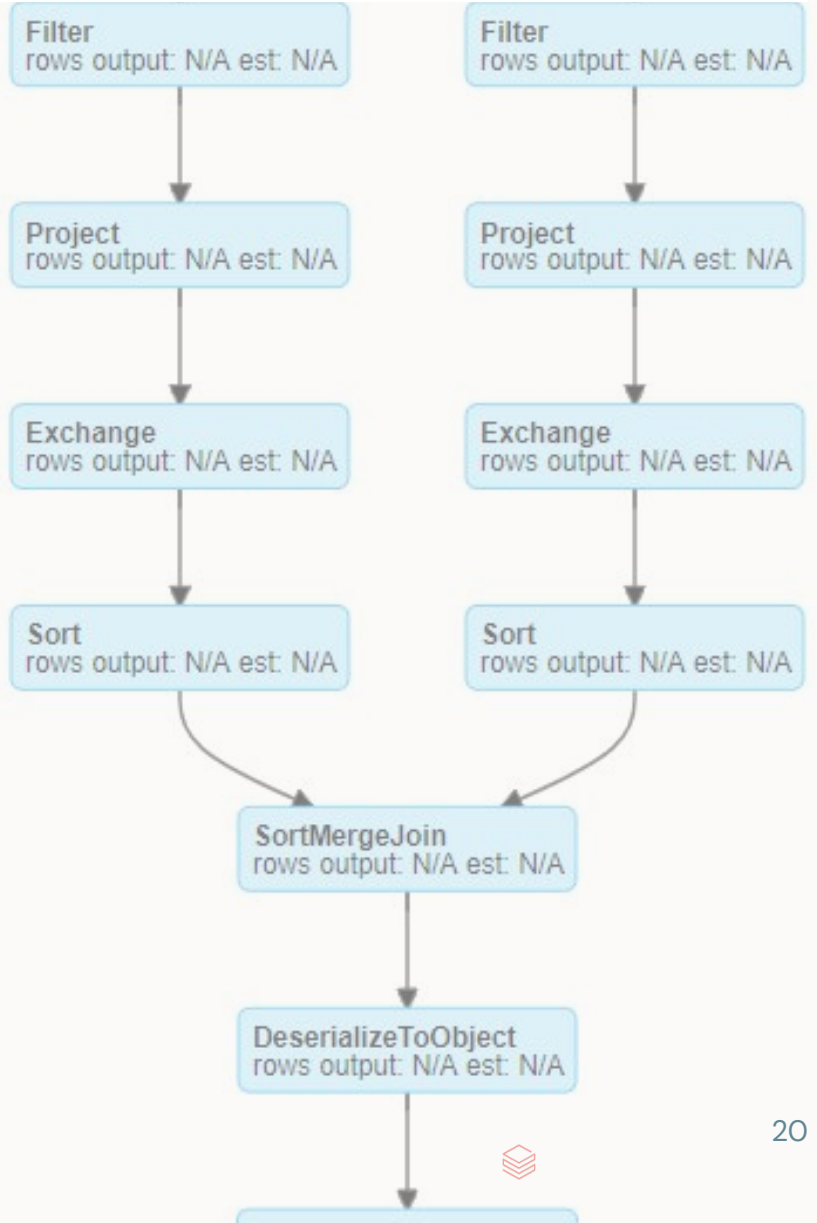
Shuffle

Shuffle

Shuffling is a side effect of wide transformations:

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

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



Shuffle - Why they're expensive

- They aggregate records across all partitions together by a key
- The aggregated records are written to disk (shuffle files)
- Each executor reads their aggregated records from the others
- The operations are limited by disk and network IO

Shuffle – Being Pragmatic

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

TIP: Don't try to remove every shuffle from ever application

- Shuffles are often a necessary evil
- Focus on the [more] expensive operations
- Many shuffle operations are quite fast and improve downstream performance by increasing parallelism
- Targeting skew, spill, tiny files, will net a greater return with less work

Shuffle - Mitigation (1)

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

Shuffle - Mitigation (2)

- Broadcast if the table is small enough
  - **spark.sql.autoBroadcastJoinThreshold**
  - **broadcast(tableName)**
  - Best suited for tables <2GB, 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

Storage

Storage - Small Files Problem

- Reading a file from cloud storage has an overhead
- Too many files per partition can lead to I/O bottleneck
- Delete/Update operations can compound the effects
- Size of partitions is important: compaction is done on per partition bases

Storage - Small Files Mitigation

- Compaction is the process of combining small files together to create larger files, minimizing the small files problem
- Bin-packing optimization is idempotent, meaning that if it is run twice on the same dataset, the second run has no effect.
- Bin-packing aims to produce evenly-balanced data files with respect to their size on disk, but not necessarily number of tuples per file. However, the two measures are most often correlated.

- To control the output file size, set the Spark configuration

```
spark.databricks.delta.optimize.maxFileSize.
```

- The default value is `1073741824`, which sets the size to 1 GB. Specifying the value 104857600 sets the file size to 100 MB.
- Example-

```
OPTIMIZE events
OPTIMIZE events WHERE date >= '2017-01-01'
OPTIMIZE events
WHERE date >= current_timestamp() - INTERVAL
1 day
ZORDER BY (eventType)
```

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Small Files Mitigation

Optimize Batch Job

- Schedule a batch job to optimize the tables frequently (e.g. daily)
- will create an in-place copy of all the data but with properly sized files, once the optimization is complete, the Delta metadata will be updated to point new data requests to the new files while not breaking existing connection
- Recommended for all tables even if you have turned on Auto Optimze

Auto Optimize

- Used with streaming workloads
- Automatically compact small files during individual writes to a Delta table and attempts to write out 128 MB files for each table partition
- Adds a spark stage to perform auto optimize and introduces a shuffle

```
ALTER TABLE
<table_name|delta.`table_path`>
SET TBLPROPERTIES
(delta.autoOptimize.optimizeWrite = true)
```

Auto Compaction

- Happens right after synchronous write is completed on streaming workload
- To control the output file size, set the Spark configuration `spark.databricks.delta.autoCompact.maxFileSize`.
- The default value is 134217728, which sets the size to 128 MB.
- Use when you don't have regular optimize scheduled on your table

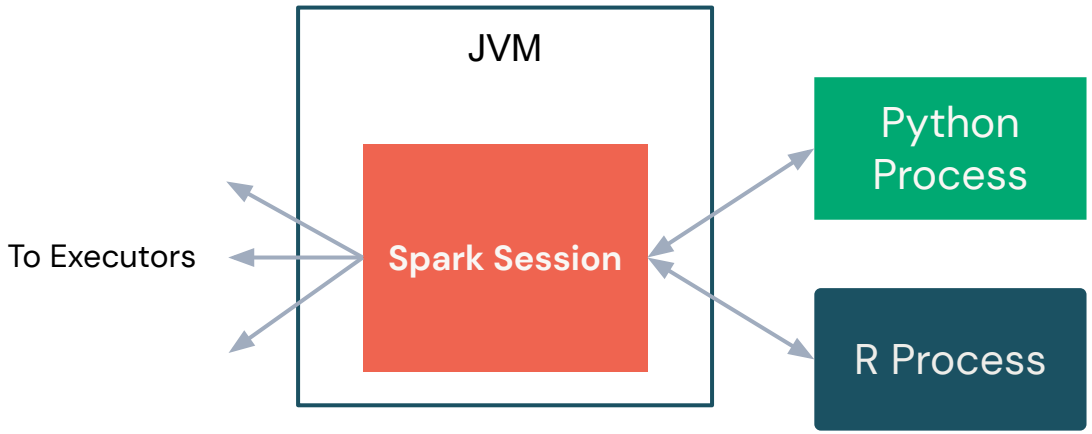
```
spark.sql("set
spark.databricks.delta.autoCompact
.enabled = true")
```


Serialization

Serialization – How serialization impacts the different APIs

- Spark SQL and DataFrame instructions are declarative and always converted to native Scala compiled to Java Bytecode (or Photon if enabled)
- When we use code, that code has to be serialized, sent to the Executors and then deserialized before it can be executed
- UDFs in languages other than Java or Scala have to be serialized natively **AND** Spark must instantiate an instance of the interpreter in each and every Executor (e.g. Pickle serialization and a Python agent)

Spark Session



- A custom Python UDF requires cross communication between the JVM and a Python agent.
- The same occurs for custom R code.
- ****Python Pandas** UDFs improve dramatically on this by using Arrow between the JVM and Python agent.

Serialization - UDFs & The Catalyst Optimizer

- Non-java native serialization creates an analysis barrier for the Catalyst Optimizer

- 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

Serialization - Mitigation

- Short answer, don't use UDFs, Vectorized UDFs or Typed Transformations if you can avoid them.
- The SQL higher-order functions are very robust and the Pyspark and Scala APIs provide native functions for the majority of tasks.
- If you must develop a UDF…
  - Use Vectorized Python UDFs
    - Vectorized Python UDFs can now be used in Databricks SQL (preview)
    - Due to advancements in the Databricks Photon engine these will soon be converted to native C++ code and will avoid all previously mentioned impacts.
  - If not using Python and/or Photon then use UDFs coded in Scala or Java

General Recommendations

Mindset when processing data

I/O – Use Delta Lake

- Read the least amount of data to get the correct answer
- Interact with your storage system as efficiently as possible

Network – Choose effective joins, use AQE

- Move data as minimally as possible
- Join the smallest tables first, then the next largest, then broadcasts

CPU – Use Photon and size your cluster correctly

- Do the most efficient set of operations to get the same answer
- Avoid inefficient or expensive operations (don't spill)
- Getting things to run in Photon completely almost always solves the problem

Clusters Top Tuning Tips

Performance Optimizations

- Use the **latest LTS** version of Databricks Runtime
  - The latest Databricks runtime is almost always faster than the one before it
    - DBR 11.3 New LTS with H3 functions, new AQE visualizations
    - DBR 11.2 Upgraded Delta Lake API to 2.1
    - DBR 11.1 Photon Generally Available, Delta Sharing GA
    - DBR 11.0 Updated to Spark 3.3
- Use Photon – fastest Spark execution engine written in C++ (launch blog)
  - Charged at a DBU premium providing up to 12x performance improvements and up to 80% TCO reductions
  - Free for DB SQL workloads and enabled by default
- Restart clusters periodically
  - It's faster and easier than monitoring for and repairing resource locks or leaks
- Use Cluster Policies to enforce best practices!
  - Cluster policies can limit the instance types, spark configuration, spot policies, etc.

Clusters Top Tuning Tips

Cost Optimization

- Don't use Interactive Clusters for Jobs
  - Use ephemeral Job Clusters for Jobs
  - This is the single biggest cost optimization impact: the DBU cost is 50% for a Job Cluster compared with an Interactive one
- Use Cluster Autoscaling
  - Use minimum of 1 to reduce cost at the expense of user experience
  - Use Resource Pools to improve user experience and lower billing if many VMs are being created/destroyed
  - Increase spark.databricks.aggressiveWindowDownS to increase/reduce the scaling latency
- Use Cluster Automatic Termination to prevent idle resources
  - Serverless Compute and Resource pools can allow for rapidly restarting
- Use Spot or Preemptible VMs to use spare VM instances for below market rate
  - Great for batch jobs, development and shared clusters - avoid for SLA based workloads

Delta Top Tuning Tips

Performance & Cost Optimization

- Use Delta Lake instead of plain Parquet or text formats
- Use Delta Cache
  - Transparent Cache of Parquet data on fast local SSDs - no need to manage the cache or JVM memory
  - Best to use Accelerated instances with Delta Cache enabled by default (such as Standard_E16ds_v4)
- Don't partition tables <1TB in size and plan carefully when partitioning
  - Partitions should be >=1GB
- Use Auto Optimize (both Optimized Writes and Auto Compaction)
- Z-Order your tables by up to 3-5 columns
  - Start with common filter columns, followed by join keys
- Make sure you use Data skipping
  - By default, only the 32 columns are used for data skipping (can be increased at an overhead for writes)
- Run periodic maintenance jobs:
  - Manual compaction: `OPTIMIZE table_name`
  - Vacuum: `VACUUM table_name`
  - Collect statistics: `ANALYZE TABLE table_name COMPUTE STATISTICS FOR ALL COLUMNS`

Spark Top Tuning Tips

Performance & Cost Optimization

- Start by finding out the best candidate code for optimizations
  - Usually these are notebooks or code called multiple times from various pipelines (different being a job parameter)
  - And pipelines or notebooks that takes a majority of the job processing time
- Use Spark 3.x (DBR >7.3) to benefit from Adaptive query execution
- Set the shuffle partitions to a multiple of number of cores
  - Default is 200 which is wasteful for clusters with > 200 cores:
    - `spark.conf.set("spark.sql.shuffle.partitions", 2 * sc.defaultParallelism)`
    - If Autoscaling, use the maximum number of nodes multiplied by the number of cores per node
  - Try the new Auto Optimized Shuffle: `spark.conf.set("spark.sql.shuffle.partitions", "auto")`
- Optimize joins
  - Increase `spark.sql.autoBroadcastJoinThreshold` to 100MB+
  - Set `spark.sql.join.preferSortMergeJoin` to false
  - Use join hints whenever possible
- Don't use Spark Caching unless absolutely necessary – use the automated Delta Caching
  - Spark Caching is only useful if repeatedly using an expensive temporary dataset (e.g. machine learning iterations)
- Remove unnecessary aggregations or sorts –
  - Ensure things like debug count() operations are removed

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

No!

Key Takeaways

- Always use Delta Lake and the latest DBR if possible and turn on Photon
- Focus on easy fixes for your biggest problems first
- Most optimizations start at the data
- Python is becoming a first class citizen and is your best low-level API
- SQL is becoming a first class citizen and is your best high-level API
- Don’t partition tables <1TB
- ZOrder and use Auto Optimize for all your tables
- Cost savings start with using the right resource type and avoiding idle time
- Spot and preemptible instances can provide 40-60% savings
- SQL Based workloads can use photon for free in premium workspaces

"That's all Folks!"