Ministry
of Justice

Deep Dive
Using Spark & AWS Glue efficiently

Theodore Manassis

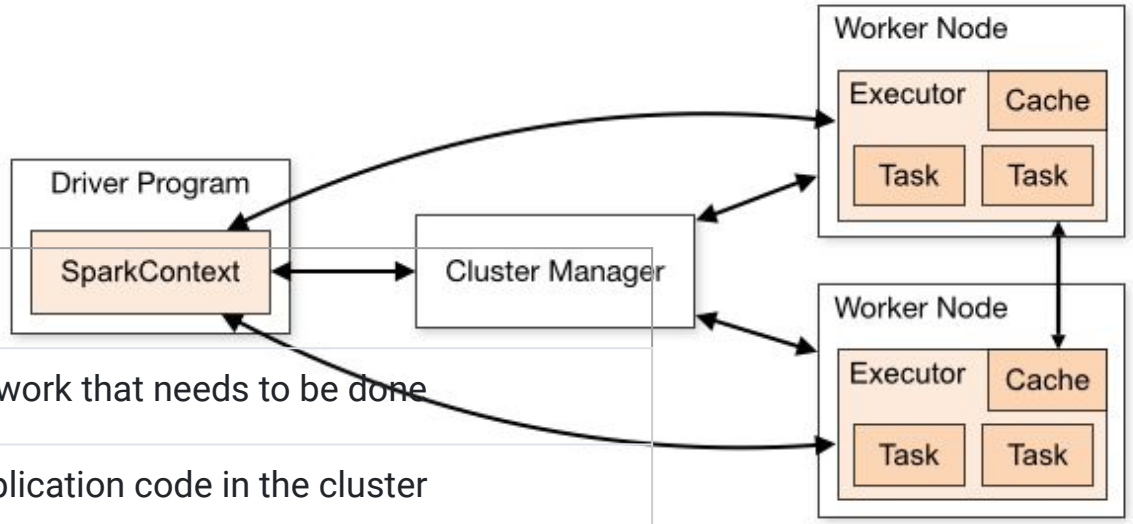
June 2023

Protecting and advancing the principles of justice

What is Apache Spark

- Apache Spark is an open-source, distributed computing system used for big data processing and analytics.
- Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.

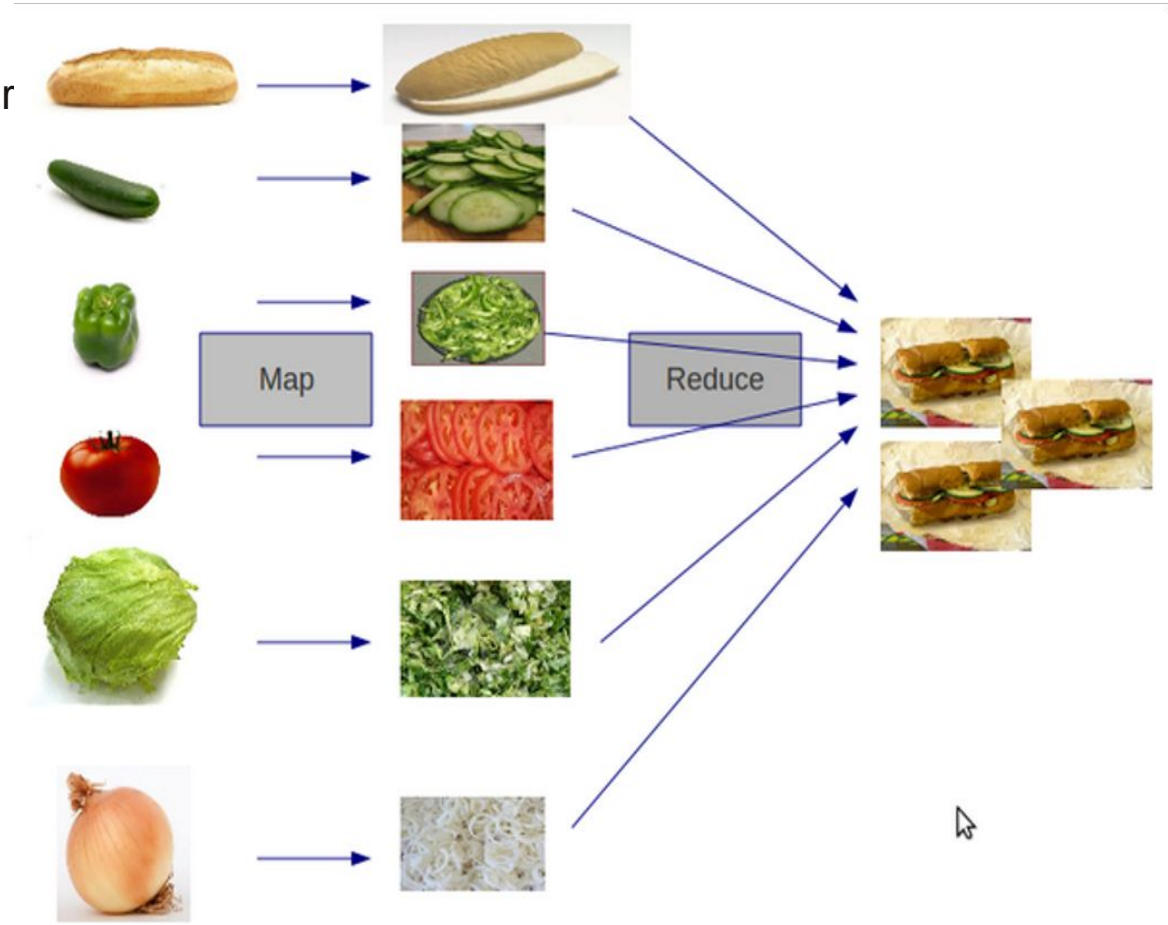
What is Apache Spark



Term	Meaning
Driver node	A node organising all the work that needs to be done
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Stage	Each job gets divided into smaller sets of tasks called *stages* that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.
Task	A unit of work that will be sent to one executor

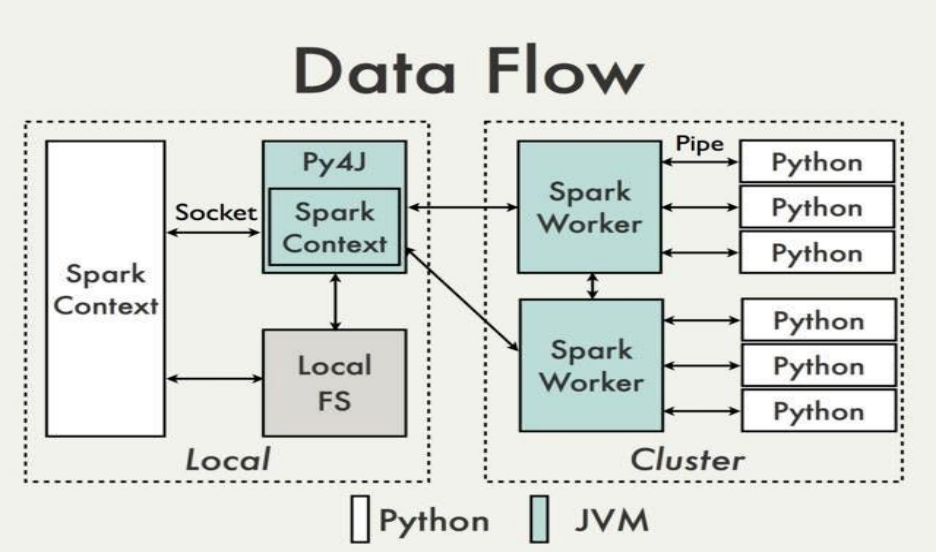
Apache Spark Pros

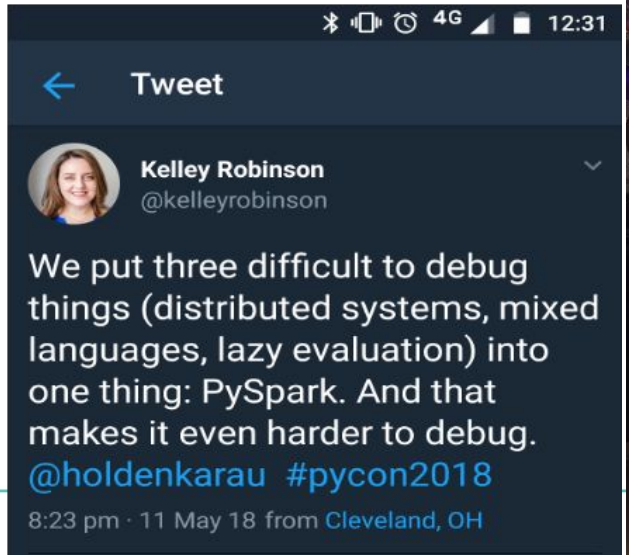
- Fast parallel processing for big datasets
- Using Map-Reduce paradigm
- In memory operations
- Up to (~1000x faster)



Apache Spark Cons

- More Complex
  - Difficult to debug
  - Being Pythonic is sometimes counterproductive
- Useful in certain data proc situations
  - but not all (!)
  - either “using sword for steak”
  - or just not suited for task
    - SCD2 better with indexing





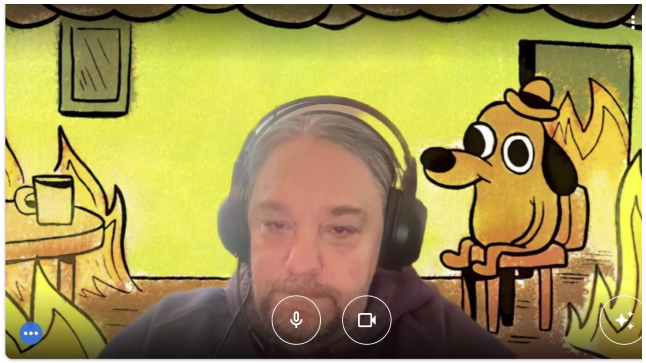
What is AWS Glue

- AWS Glue is a fully managed extract, transform, and load (ETL) service provided by Amazon Web Services (AWS).

- It offers a way of running batch Spark programs among other things

Problems with MPM Pipeline

- Certain jobs taking a lot of time to process
- Cost explosion
- Glue jobs failing for unclear reasons.
- Failures causing further data integrity issues



Reasons for these issues

(Possible?!) Reasons (IMHO) :

- Defaults on many configurations. That was the cause of many issues
- Inadequate logging (flying blind)
- Using Glue 2.0 (and only lever available at the time was upping # of workers)
- Pyspark in a Pythonic manner has traps

Glue 3.0 / 4.0

- From Glue 3.0 onwards:
- Further Tuning configs
- Autoscaling
- AQE (Adaptive Query Execution)
- Hudi , Iceberg , Deltalake data lakehouse
- (However not all our tables could be ported to glue 3.0 or 4.0 reliably)

AutoScaling

- Autoscaling
- Enable up to a specified number of workers
- `"--conf enable-auto-scaling=true"`
- not wasting compute (in more detail)

AQE (Adaptive Query Execution)

From Glue 3.0 onwards but really making a difference from 4.0 onwards

```
--conf spark.sql.adaptive.enabled=true
--conf spark.sql.adaptive.coalescePartitions.enabled=true
--conf spark.sql.adaptive.advisoryPartitionSizeInBytes=128m
--conf spark.sql.adaptive.fetchShuffleBlocksInBatch=true
--conf spark.sql.adaptive.skewJoin.enabled=true
--conf spark.sql.adaptive.localShuffleReader.enabled=true
```

Performance Tuning - Spark Documentation

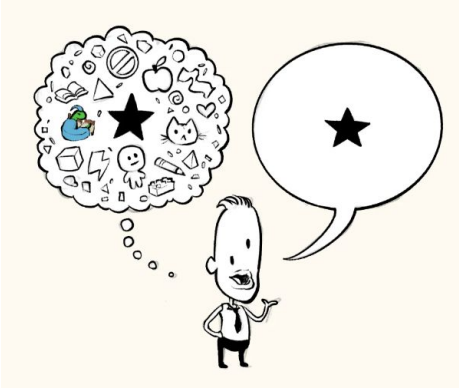
Adding custom logging

- There is a lot of information at every glue run that a data engineer needs to be aware of (especially when trying to debug). Even though existing spark logs are quite verbose on certain things there is a lot of important info that is implicit and is not shown. Good logging fixes that.

- glueContext has a logger object that can be used quite easily. Other ways to log exist too.

```
logger = glueContext.get_logger()

logger.warn(f"{str(get_spark_config_values(spark))}")
```



Logging

```
def get_spark_config_values(spark: SparkSession):
    # Define a list of important Spark configuration keys
    spark_config_keys = [
        # Executor and driver configurations
        "spark.executor.cores", "spark.executor.memory",
        "spark.executor.instances", "spark.driver.cores",
        "spark.driver.memory", "spark.driver.maxResultSize",
        # Parallelism configurations
        "spark.default.parallelism", "spark.sql.shuffle.partitions",
        # Memory configurations
        "spark.memory.fraction",  "spark.memory.storageFraction",
    ]
    # Retrieve the values for each key and store them in a dictionary
    spark_config_values = {}
    for key in spark_config_keys:
        value = spark.conf.get(key, None)
        if value is not None:
            spark_config_values[key] = value
        else:
            spark_config_values[key] = "Not explicitly set"
    # Return the dictionary containing the Spark settings
    return spark_config_values
```

What can go wrong : Data Skew

Details for Stage 10

Total task time across all tasks: 3.0 min
Input: 80.5 MB
Shuffle write: 80.5 MB

▸ Show additional metrics

Min to 75th Percentile is A lot smaller than 75th Percentile to max indicating Data Skew

Summary Metrics for 36 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.2 s	0.2 s	0.2 s	0.7 s	1.2 min
GC Time	0 ms	0 ms	0 ms	0 ms	50 ms
Input	63.0 B	64.0 B	127.0 B	477.1 KB	25.7 MB
Shuffle Write	1006.0 B	1008.0 B	1051.0 B	478.3 KB	25.7 MB

Solutions: (implicit / explicit) partitioning & salting

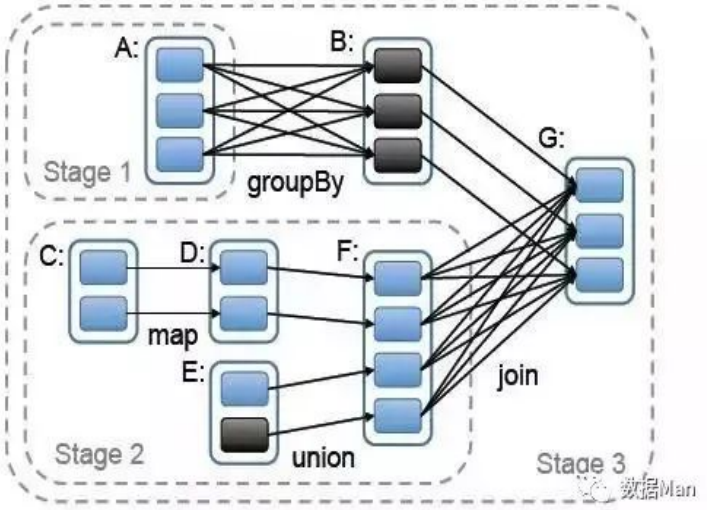
```
spark.sql.adaptive.skewJoin.enabled=true
```

dynamically handles skew by splitting (and replicating if needed) skewed partitions.

What can go wrong : excessive shuffling

Apache Spark processes queries by distributing data over multiple nodes and calculating the values separately on every node**.**

- However, occasionally, **the nodes need to exchange the data**.
- Need to avoid excessive shuffling. (use SparkUI, configs)

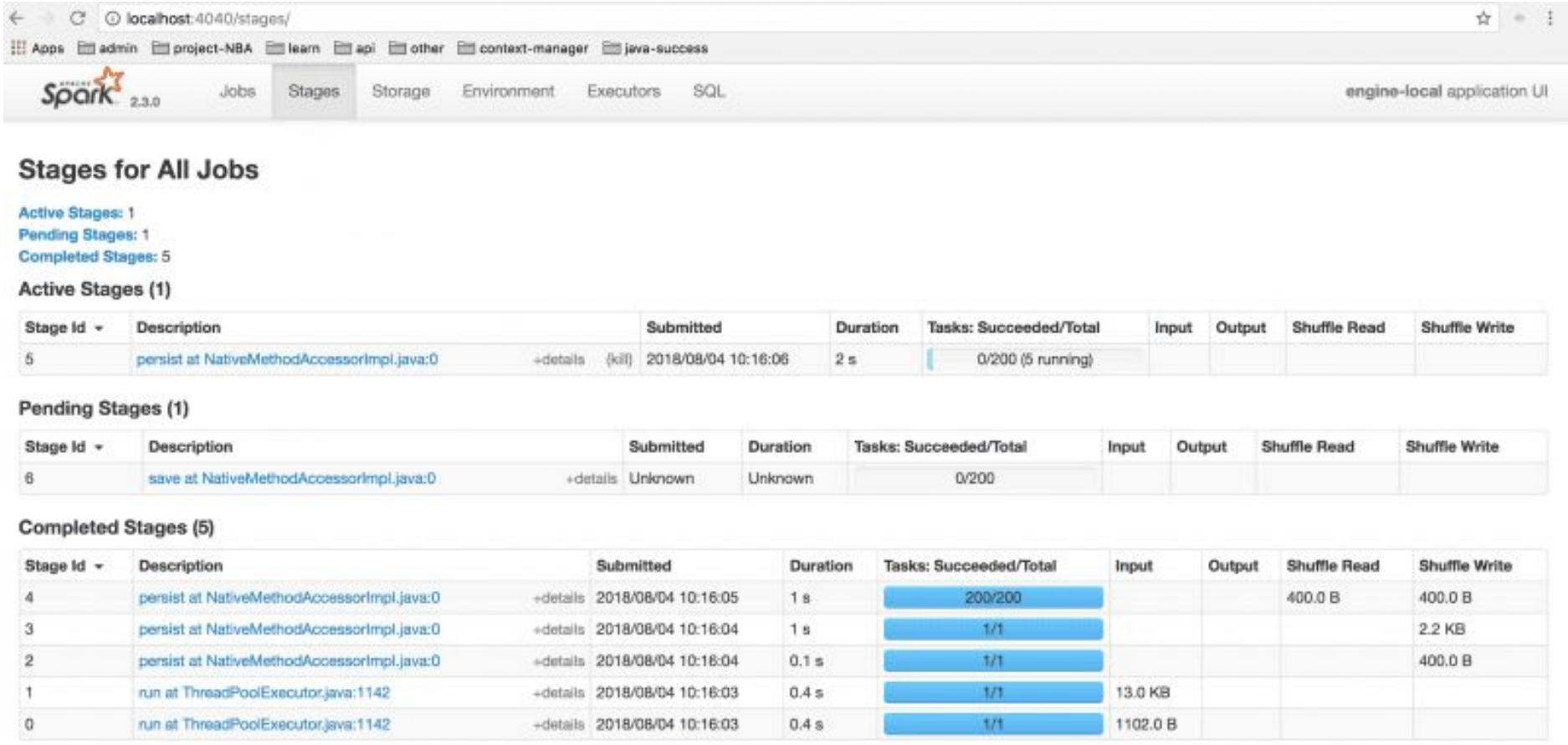



```
--conf spark.sql.adaptive.localShuffleReader.enabled=true
--conf spark.sql.adaptive.fetchShuffleBlocksInBatch=true
```

Spark UI


localhost:4040/jobs/job/?id=2
Apps
admin
project-NBA
learn
api
other
context-manager
java-success
Spark 2.3.0
Jobs
Stages
Storage
Environment
Executors
SQL
engine-local application UI
Details for Job 2
Status: RUNNING
Active Stages: 1
Pending Stages: 1
Completed Stages: 3
Event Timeline
DAG Visualization
Stage 5
Stage 2
Stage 6
Stage 3
Stage 4
Exchange
WholeStageCodegen
SortMergeJoin
BatchEvalPython
Window
mapPartitionsInternal
map

Spark UI


localhost:4040/stages/
Apps admin project-NBA learn api other context-manager java-success
Spark 2.3.0
Jobs Stages Storage Environment Executors SQL
engine-local application UI
Stages for All Jobs
Active Stages: 1
Pending Stages: 1
Completed Stages: 5
Active Stages (1)
Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write
5 | persist at NativeMethodAccessorImpl.java:0 +details (kill) | 2018/08/04 10:16:06 | 2 s | 0/200 (5 running)
Pending Stages (1)
Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write
6 | save at NativeMethodAccessorImpl.java:0 +details | Unknown | Unknown | 0/200
Completed Stages (5)
Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write
4 | persist at NativeMethodAccessorImpl.java:0 +details | 2018/08/04 10:16:05 | 1 s | 200/200 | | | 400.0 B | 400.0 B
3 | persist at NativeMethodAccessorImpl.java:0 +details | 2018/08/04 10:16:04 | 1 s | 1/1 | | | | 2.2 KB
2 | persist at NativeMethodAccessorImpl.java:0 +details | 2018/08/04 10:16:04 | 0.1 s | 1/1 | | | | 400.0 B
1 | run at ThreadPoolExecutor.java:1142 +details | 2018/08/04 10:16:03 | 0.4 s | 1/1 | 13.0 KB | | |
0 | run at ThreadPoolExecutor.java:1142 +details | 2018/08/04 10:16:03 | 0.4 s | 1/1 | 1102.0 B | | |

Input Parallelism

Set up parallelism configs

(**change default for large datasets!!**) :

Used heuristic eg. `parv = (2 * filenum / DPU)`

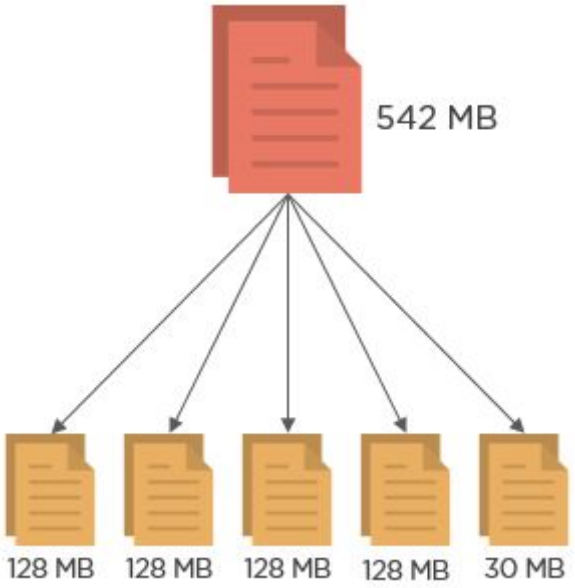
```
--conf spark.default.parallelism={parv}
--conf spark.sql.files.maxPartitionBytes
--conf spark.sql.adaptive.coalescePartitions.enabled=true
--conf spark.sql.adaptive.advisoryPartitionSizeInBytes=128m
```

to avoid too many small tasks.

Performance Tuning - Spark Documentation

Shuffle Parallelism

- Set up parallelism configs
- (**change default for large datasets !!**)
- heuristic: either same as `spark.default.parallelism`
- or amount_of_data / 128MB
- iterative process : SparkUI/logs




```
--conf spark.sql.shuffle.partitions={parv}
```

[Performance Tuning - Spark Documentation]

Output Parallelism suggestions

- `df.coalesce(n) # for shrinking partitions no shuffling involved !!`
- `df.repartition(n) # for expanding partitions`

- `df.write.option("MaxRecordsPerFile",n)`
- `df.write.PartitionBy(col) #low cardinality`
- `df.write.BucketBy(col) #high cardinality`

Recap & Lessons learned

- Logging (not only for debugging also for observability)
- Parallelism tuning configs (less idle workers)
- Autoscaling (not paying AWS for idle workers)
- AQE (Adaptive Query Execution) configs
- Shuffling configs

Outcome: Significant savings

- Using these suggestions in MPM pipeline

- From 20% improvement
- Up to 75% improvement (from $460 to $110 per run)

However we had Sprint looking for other solutions:

- We decided that a Athena/Iceberg solution would be cheaper (and as efficient or even better) for SCD2
- We have the option however to run same code in Spark/Iceberg if needed.

Hudi / Iceberg / Deltalake support in >= v.3

- Hudi , Iceberg , Deltalake data lakehouse formats supported . eg for Iceberg:
- `--conf datalake-format=iceberg`

Spark Surgeries

- Propose monthly meeting where people can talk about problems solutions or perform deep dives into a particular Spark area
  - I am doing 2 presentations to the linking team that could be beneficial for Data Engineers
    - Spark UDFs: Efficient (or not) ways of creating functions not available in Pyspark API
    - Develop Spark Scala UDFs: How to create functions in Scala that can be run very efficiently in Spark.

- #spark_and_glue slack Channel in ASD