Day 3: S3 – Spark-Scala-Session

By Keshav Thakur ☺

Spark – Under the Hood | Spark SQL → Data frames && Data sets

Spark SQL → Oracle and relational db lover's → Here we have the SQL → Write SQL Queries

Implemented on Top of Spark ➔ everything starts with SparkSession

Three main API's

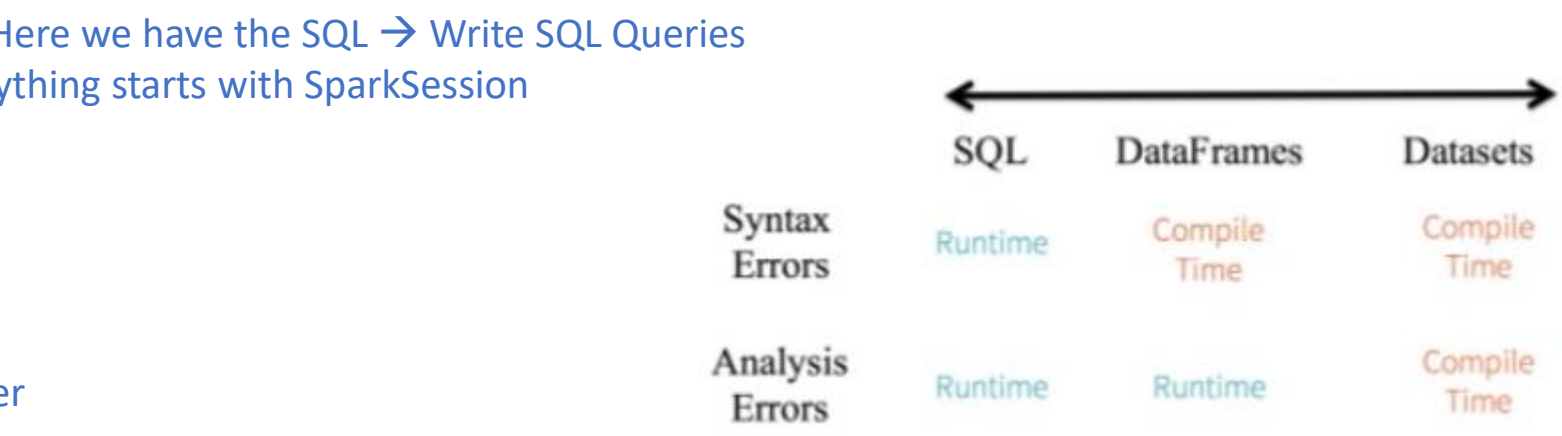
- SQL Literal Syntax
- Dataframe
- Datasets

But How? Who does this?

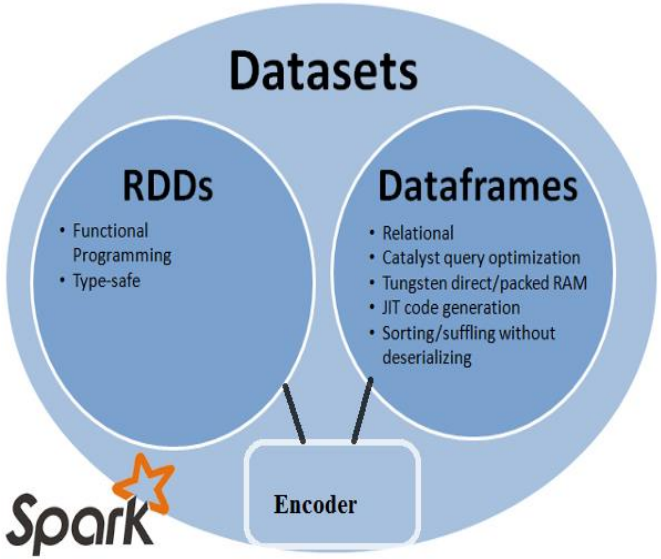
- Tungsten → off-heap serializer
- Catalyst → query optimizer

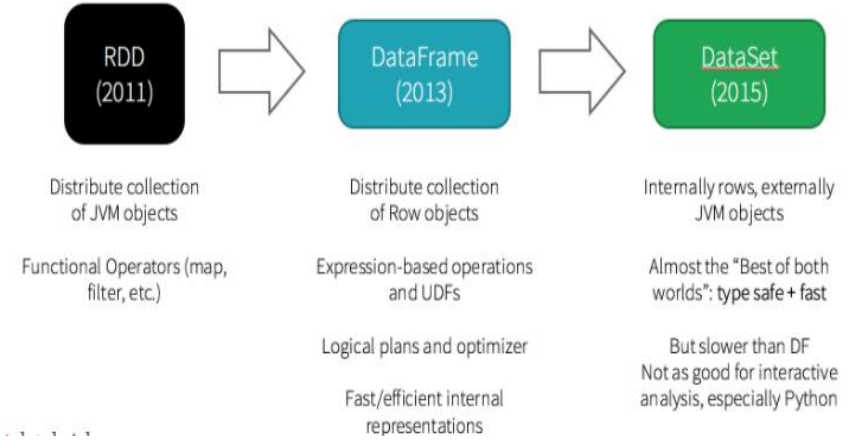
Datasets ➔something in between rdd and dataframe➔ unifies dataframe & rdd's ➔ typesafe API's

Dataframe ➔ RDD's with Schema ➔ Equivalent to table to DB ➔ Excel Spreadsheet ➔ JSON, CSV, Parquet, JDBC

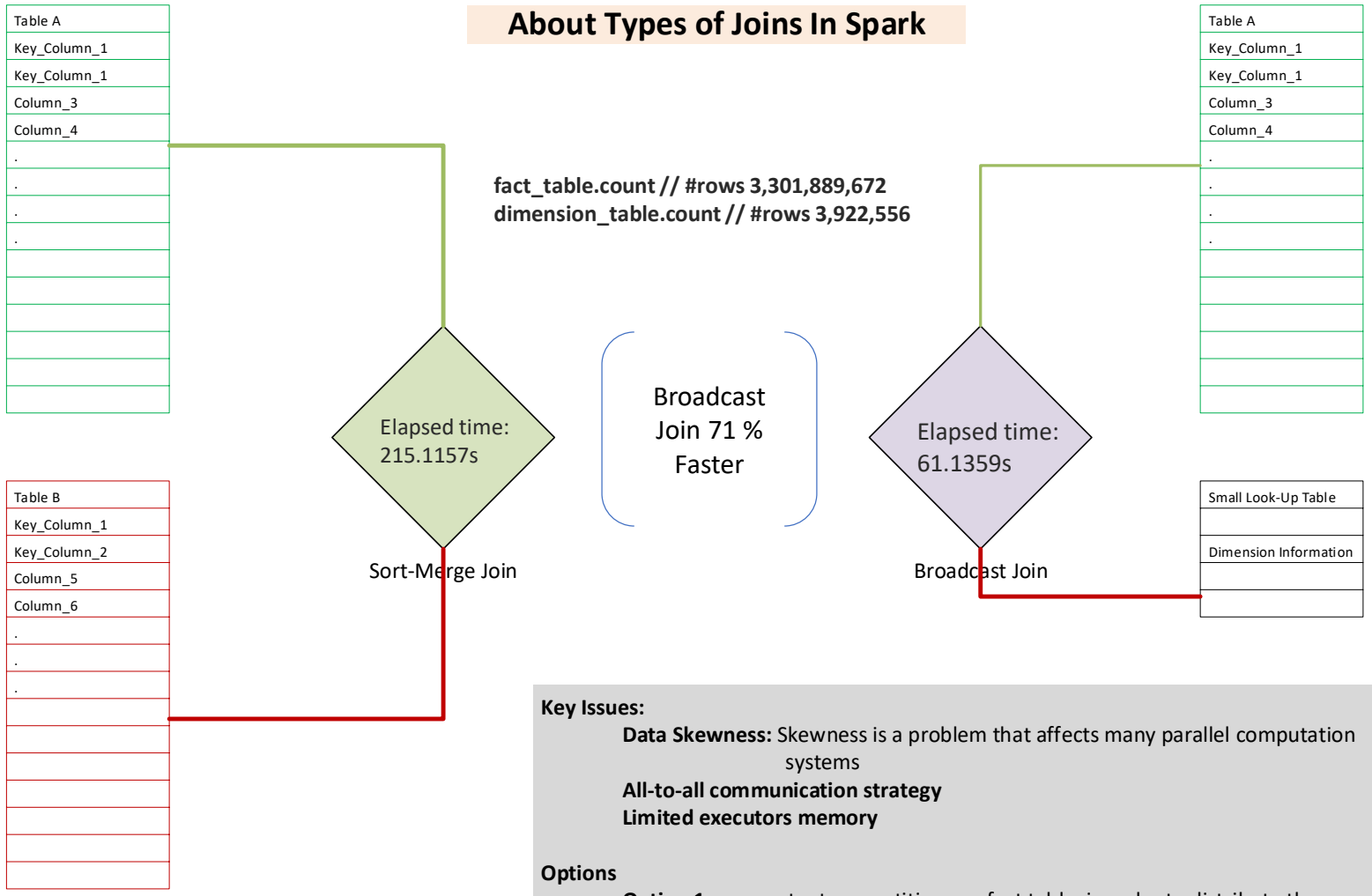


	RDD	Dataframe	Dataset
Optimization	No optimization engine, never use catalyst optimizer and tungsten execution engine.	Use catalyst optimizer tungsten execution engine.	More advance than DF.
Serialization	Use java serialization to store or distribute the data and its expensive and require sending both data and structure nodes.	Serialize data into off-heap (in-memory)in binary format and apply transformations and use tungsten execution engine to manage memory and dynamically generates bytecode.	Dataset API use encoder concept and store tabular representation using tungsten.
Garbage Collection	Overhead of garbage collection.	Avoids the garbage collection costs in constructing individual objects for each row in the dataset.	There is also no need for the garbage collector to destroy object because serialization takes place through Tungsten. That uses off heap data serialization.





Spark – About Join | Optimization & Pain Points



Key Issues:

Data Skewness: Skewness is a problem that affects many parallel computation systems

All-to-all communication strategy

Limited executors memory

Options

Option 1: we can try to repartition our fact table, in order to distribute the effort in the nodes

Option 2: we can artificially create a repartitioning key (key salting)

Concepts

- Joins can be difficult to tune since performance are bound to both the code and the Spark configuration (number of executors, memory, etc.)
- Some of the most common issues with joins are all-to-all communication between the nodes and data skewness
- We can avoid all-to-all communication using broadcasting of small tables or of medium-sized tables if we have enough memory in the cluster
- Broadcasting is not always beneficial to performance: we need to have an eye for the Spark config
- Broadcasting can make the code unstable if broadcast tables grow through time
- Skewness leads to an uneven workload on the cluster, resulting in a very small subset of tasks to take much longer than the average
- There are multiple ways to fight skewness, one is repartitioning.
- We can create our own repartitioning key, e.g. using the key salting technique
- Sort Merge Joins tend to minimize data movements in the cluster, especially compared to **Shuffle Hash Joins**

Spark – About Join | Sort-Merge & Shuffle-Hash Join

Sort -Merge Join

Sort-Merge join is composed of 2 steps.
The first step is to sort the datasets and
the second operation is to merge the sorted data in the partition by iterating over the elements and according to the join key join the rows having the same value.

From spark 2.3 Merge-Sort join is the default join algorithm in spark. However, this can be turned down by using the internal parameter '**spark.sql.join.preferSortMergeJoin**' which by default is true.

To accomplish ideal performance in Sort Merge Join:
• Make sure the partitions have been co-located. Otherwise, there will be shuffle operations to co-locate the data as it has a pre-requirement that all rows having the same value for the join key should be stored in the same partition.

• The DataFrames should be distributed uniformly on the joining columns.

• To leverage parallelism the DataFrames should have an adequate number of unique keys

Sort merge join is a very good candidate in most of times as it can spill the data to the disk and doesn't need to hold the data in memory like its counterpart Shuffle Hash join.

However, when the build size is smaller than the stream size Shuffle Hash join will outperform Sort Merge join.

During joins if there are rows which are irrelevant to the key, filter the rows before the join. Otherwise, there will be more data shuffle over the network.

if you are confident enough that Shuffle Hash join is better than Sort Merge join, disable Sort Merge join for those scenarios.

Shuffle-Hash Join

Spark chooses Shuffle Hash join when Sort merge join is turned off or if the key is not suitable and also based on the accompanying two functions.

Spark – Performance Tuning | Get our hands dirty

Best Practices and Optimization tips for Spark

How to leverage Tungsten?

- Use Dataset structures rather than DataFrames
- Avoid User-Defined Functions (UDFs) as much as possible
- Avoid User-Defined Aggregate Functions (UDAFs)

Execution plan analysis,

- Look under the hood

Data management (caching, broadcasting)

Know your data and manage it efficiently

- Highly imbalanced datasets
- Inappropriate use of caching
- Broadcasting
- Unnecessary Data Shuffles

Cloud-related optimizations (including S3).

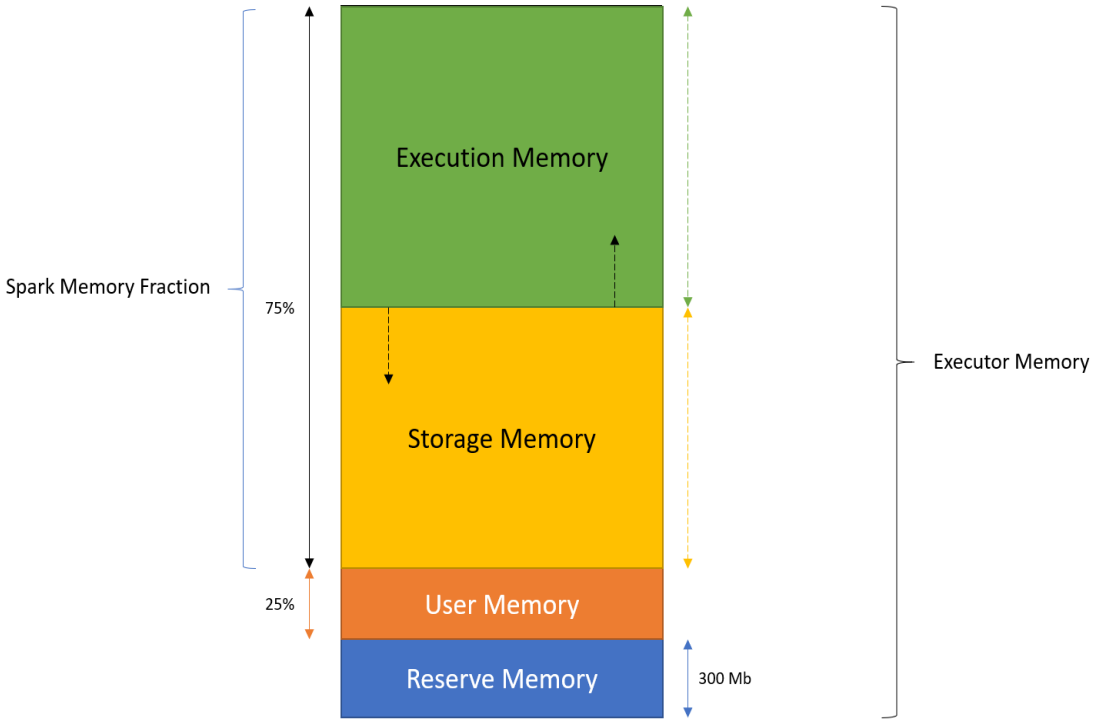
Cluster-related Optimizations

```
spark.executor.memory spark.executor.cores
spark.executor.instances
spark.yarn.executor.memoryOverhead
spark.default.parallelism
```

Few Important Tweaks:

- Grouping Sets – grouping_id()
- UDF in Scala than in Python
- Partitioning & Bucketing
- Apache Parquet Format – for read & write
- Cache Judiciously and use Checkpointing
- Use the right level of parallelism
- Use Broadcast Joins
- Compact columnar memory format
- Direct memory access
- Reduced garbage collection processing overhead
- Catalyst query optimization
- Whole-stage code generation
- Partition Pruning and Predicate Pushdown
- Partitioning Tips
  - Coalesce and Repartition
  - Bucketing
    - Queries on bucketed values
    - Aggregations on bucketed values (wide transformations)
    - Joins on bucketed values

Spark – OOM Error -- java.lang.OutOfMemoryError



Execution Memory — Spark Processing or generated data like RDD transformation. Used for shuffle, join, sort. Will spill to disk in case of allocated memory limit is breached. This is short lived.
Storage Memory — Cached objects and broadcast variables are stored in here. Upon storage limit breach it spills data to disk.

1.Reserve Memory — Hardcoded value of 300 Mb.
2.User Memory — 25% of allocated executor memory. Used for user defined objects like hash map. Stores information for RDD dependency. Throws OOM error, in case object exceeds the limit. Will not spill to disk.
3.Memory Fraction — 75% of allocated executor memory.

Executor Side Memory Errors

- *spark.executor.memory*
- *spark.yarn.executor.memoryOverhead*
- Concurrency
  - *spark.default.parallelism*
  - *spark.executor.cores*
- Big Partitions
  - *spark.sql.shuffle.partitions*

Driver Side Memory Errors

- *spark.driver.memory*
- *spark.driver.maxResultSize*
- Broadcast join
  - *spark.sql.autoBroadcastJoinThreshold*

Spark – Debug Queries Using Spark UI

```
spark2-shell --queue=P0 --num-executors 20

Spark context Web UI available at http://<hostname>:<port>
Spark context available as 'sc'
Spark session available as 'spark'
```

Every SparkContext launches a Web UI, by default on port 4040, that displays useful information about the application. This includes:

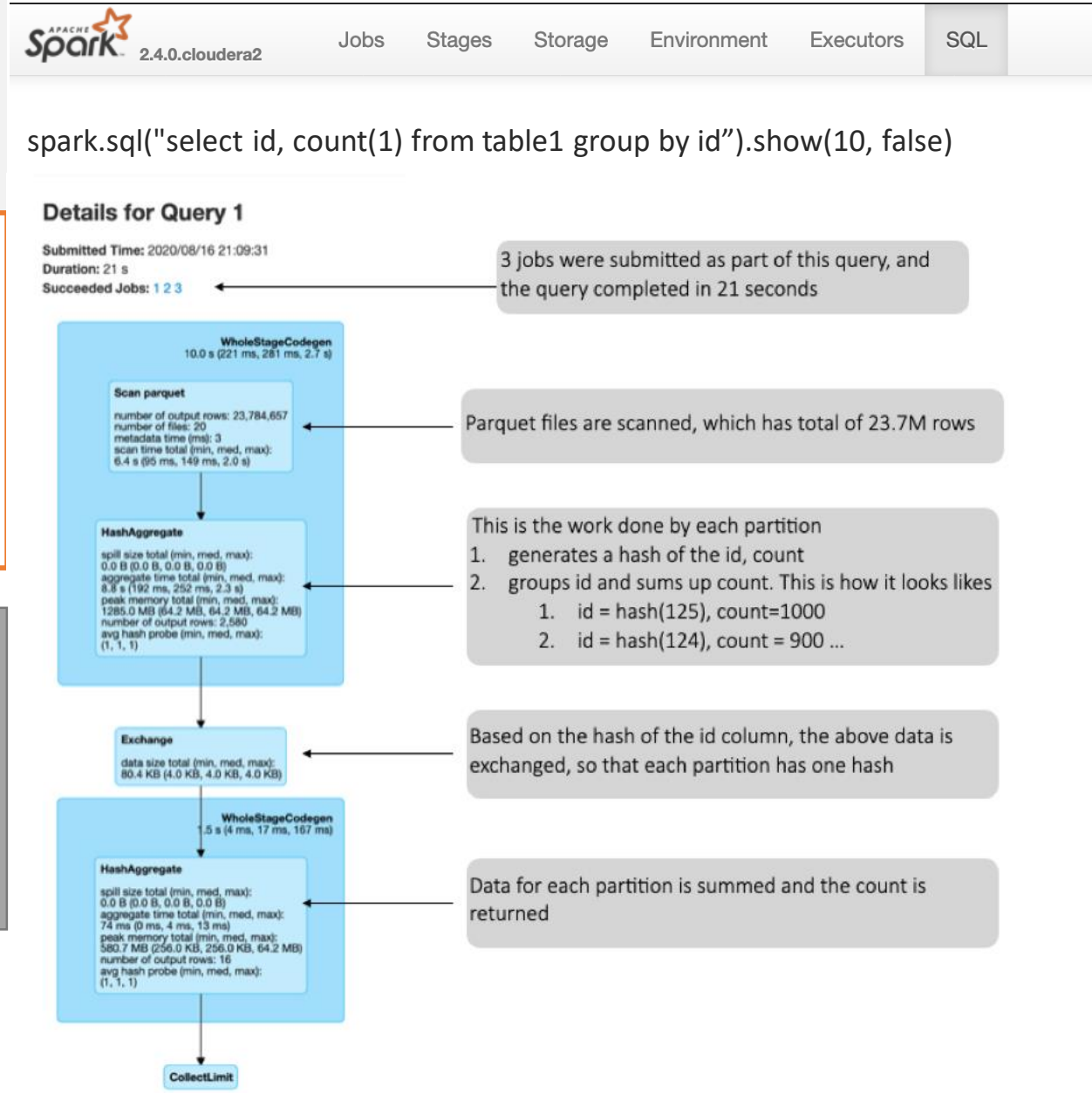
A list of scheduler stages and tasks

- A summary of RDD sizes and memory usage
- Environmental information.
- Information about the running executors
- You can access this interface by simply opening http://<driver-node>:4040 in a web browser. If multiple SparkContext are running on the same host, they will bind to successive ports beginning with 4040 (4041, 4042, etc).

Executor Tab: This tab gives us an idea of the number of executors your spark session is currently active with.
Environment Tab: The environment tab has the details of all the config parameters that the spark session is currently using.
Storage Tab: This shows information about Spark — Caching. Df.persist and df.cache

One of: `MEMORY_ONLY`, `MEMORY_AND_DISK`, `MEMORY_ONLY_SER`,`MEMORY_AND_DISK_SER`, `DISK_ONLY`, `MEMORY_ONLY_2`,`MEMORY_AND_DISK_2`, etc.

https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html
https://spark.apache.org/docs/3.0.0-preview/web-ui.html



Spark – Under the Hood | But shit happens → Interview

Typically, in an Interview: What is asked?

- Key concepts → RDD's vs Data frame vs Dataset → SparkContext vs SparkSession → groupByKey vs reduceByKey
- Situation Aware concepts: any particular use case given and solution is expected.
- What do you do? How has been the product or project lifecycle? What you are using and why? Spark Roadmap
- Spark Debugging → How do you debug locally? Log monitoring? Common Error Types?
- Spark UI Dashboard → What to see and how it helps?
- Elaborate some challenges you are facing on your projects? How often the job fails and what you do?
- What have you done? Your contribution? Major functions and best practices observed in your project
- && the PITA – Pain In The A** → Spark Job Optimization
  - Technique's
  - Workaround's
  - Best Practice's
    - JVM Optimization on all executors
    - Look into the DAG diagram in Spark UI → No of Tasks, No of stages, No of Jobs
    - Go through all the logs to observe latency – Partitioning & Shuffling – Data Read and Write
    - Source code optimization → Memory leaks, order of operations
    - Infrastructure scaling… adding more node… easiest way but difficult to convince.
    - Introduce checkpointing, caching.. Persistence
    - Garbage collection mechanism ……
    - Compression, Serialization, Data limitation
    - At the end--- History of Jobs --- Investigation → How often it fails? Network Bandwidth? Patterns