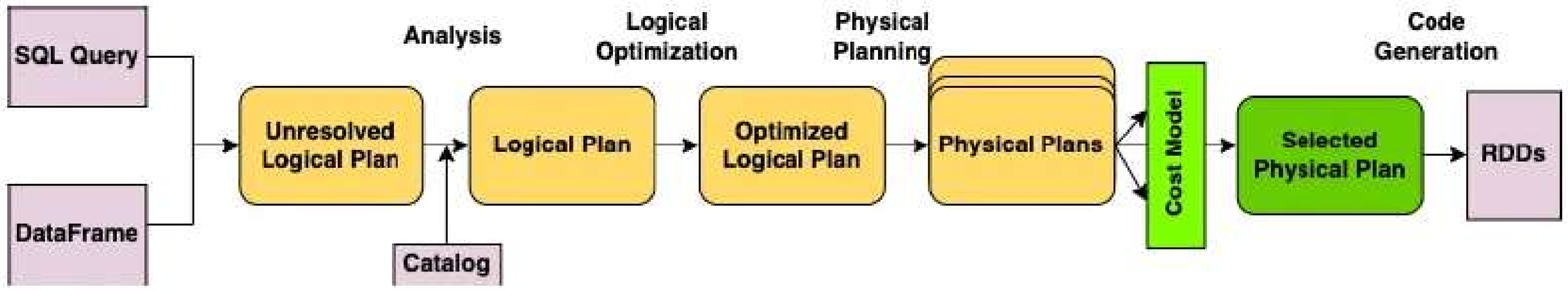
Spark Performance Tuning

Performance Tuning & Optimization Techniques

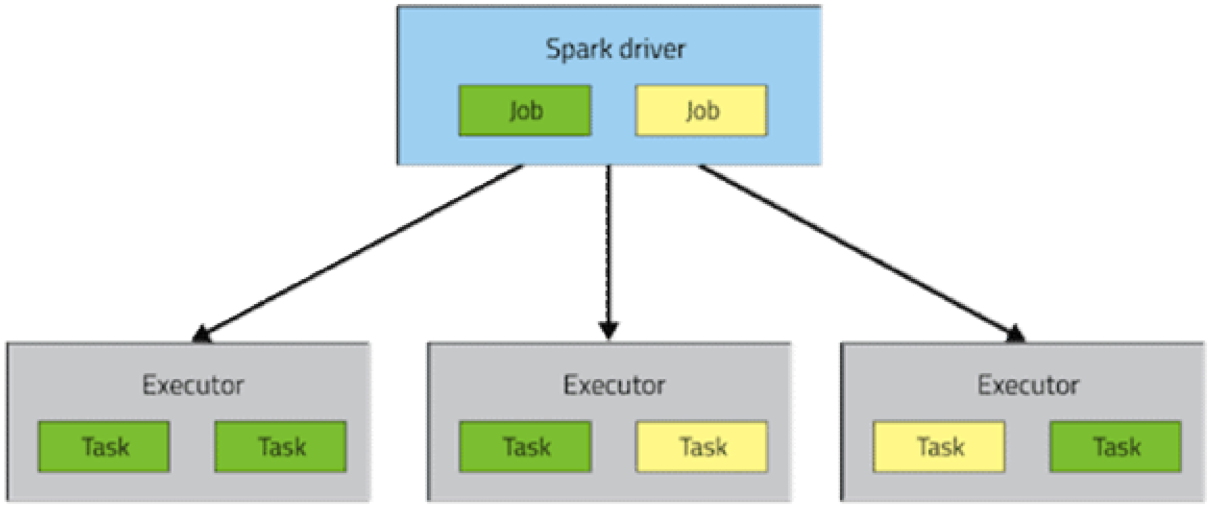
Spark Execution Process

Spark SQL Catalyst execution stages



Spark execution process

A Spark application consists of a single driver process and a set of executor processes scattered across nodes on the cluster.



Spark execution process

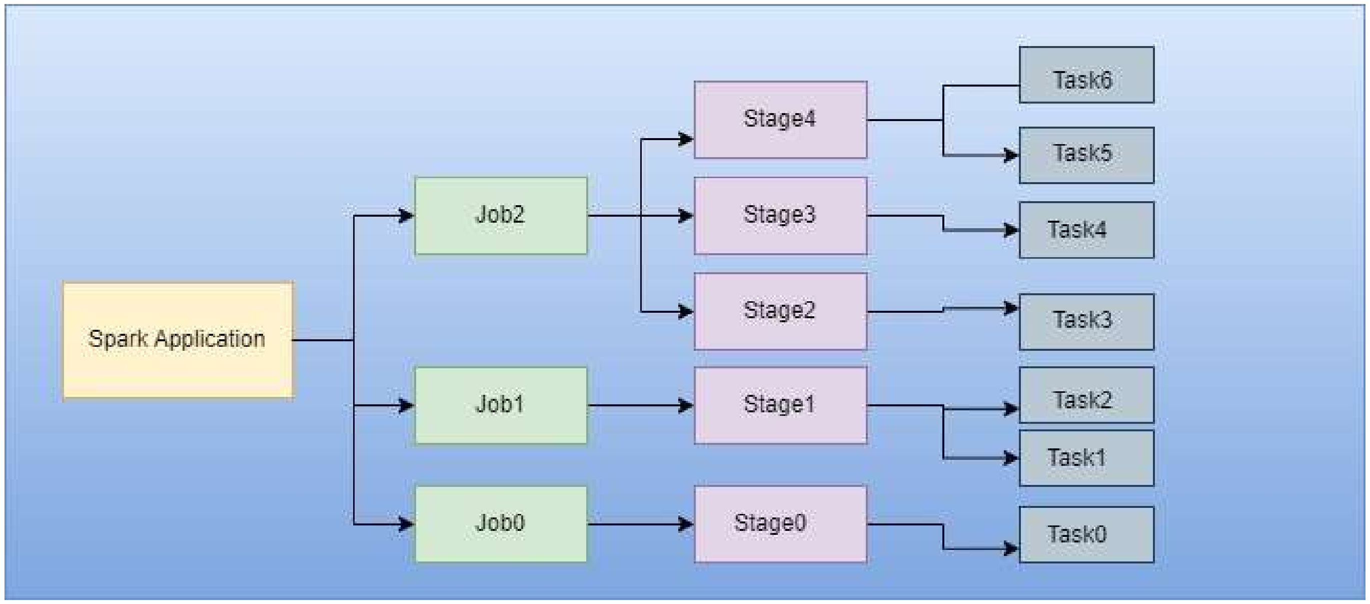
- The driver is the process that is in charge of the high-level control flow of work that needs to be done.
- An executor is the JVM process launched on a worker node. An executor runs tasks in threads and is responsible for keeping relevant partitions of data.
- The executor processes are responsible for :
  - Executing the tasks allocated by the driver.
  - Storing any data that the user chooses to cache.
- A single executor has a number of slots for running tasks, and will run many tasks concurrently throughout its lifetime.
- Deploying these processes on the cluster is up to the cluster manager in use, but the driver and executor themselves exist in every Spark application.

Spark execution plan

- At the top of the execution hierarchy are jobs.
- Invoking an action inside a Spark application triggers the launch of a Spark job to fulfill it.
- To decide what this job looks like, Spark examines the graph of RDDs on which that action depends and formulates an ***execution plan***.
- This plan starts with the farthest-back RDDs—that is, those that depend on no other RDDs or reference already-cached data–and culminates in the final RDD required to produce the action's results.

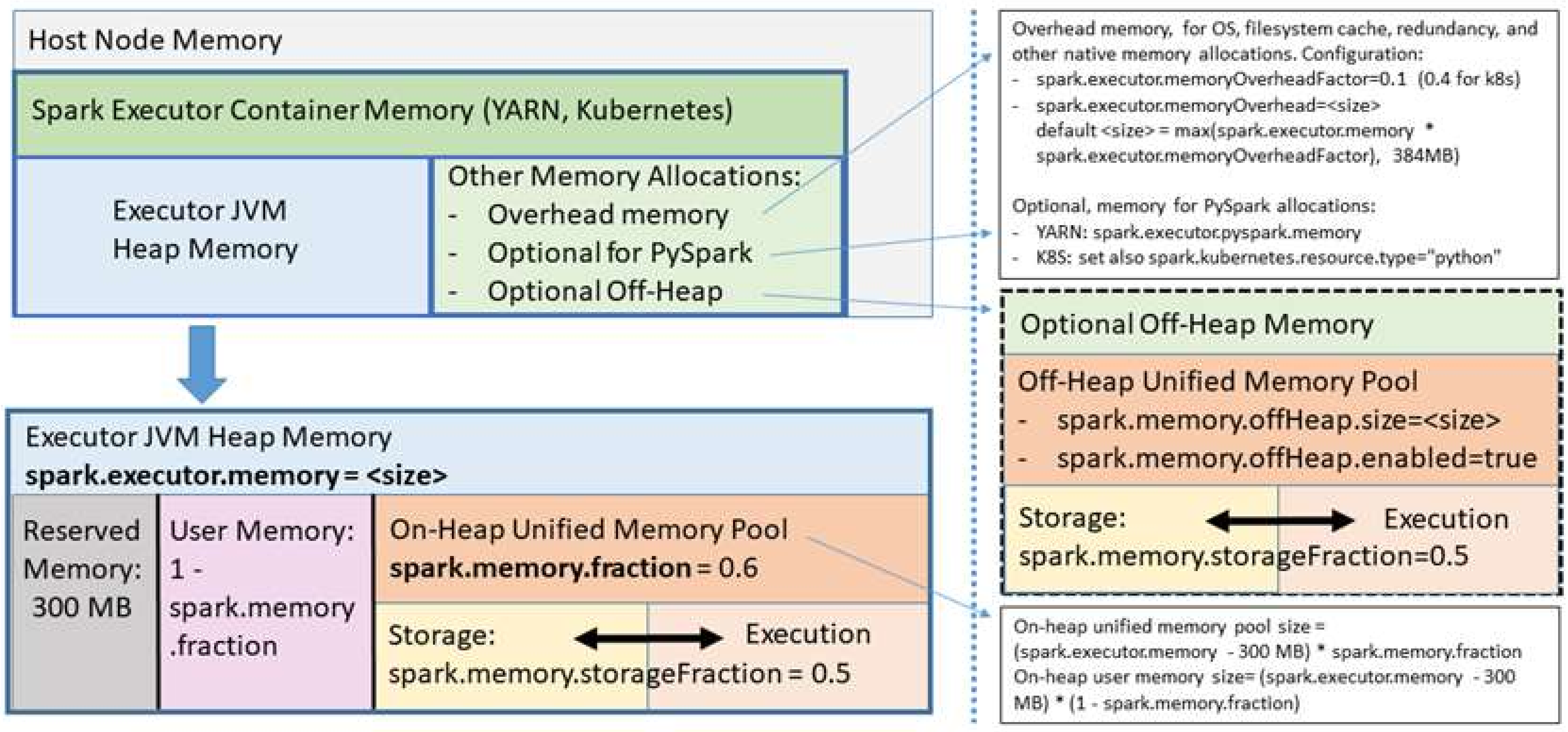
Spark stages

- The execution plan assembles the job's transformations into ***stages***.
- A stage corresponds to a set of tasks that all execute the same code, each on a different subset of the data and contains a sequence of transformations that can be completed without shuffling the full data.



Spark Memory Management

Spark memory management



Host Node Memory
Spark Executor Container Memory (YARN, Kubernetes)
Executor JVM Heap Memory
Other Memory Allocations:
- Overhead memory
- Optional for PySpark
- Optional Off-Heap
Executor JVM Heap Memory
spark.executor.memory = <size>
Reserved Memory: 300 MB
User Memory: 1 - spark.memory.fraction
On-Heap Unified Memory Pool
spark.memory.fraction = 0.6
Storage:
Execution
spark.memory.storageFraction = 0.5
Optional Off-Heap Memory
Off-Heap Unified Memory Pool
- spark.memory.offHeap.size=<size>
- spark.memory.offHeap.enabled=true
Storage:
Execution
spark.memory.storageFraction=0.5

Overhead memory, for OS, filesystem cache, redundancy, and other native memory allocations. Configuration:
- spark.executor.memoryOverheadFactor=0.1 (0.4 for k8s)
- spark.executor.memoryOverhead=<size>
  default <size> = max(spark.executor.memory * spark.executor.memoryOverheadFactor), 384MB)

Optional, memory for PySpark allocations:
- YARN: spark.executor.pyspark.memory
- K8S: set also spark.kubernetes.resource.type="python"

On-heap unified memory pool size = (spark.executor.memory - 300 MB) * spark.memory.fraction
On-heap user memory size= (spark.executor.memory - 300 MB) * (1 - spark.memory.fraction)

Driver memory configurations

- **`spark.driver.memory`**
  - Amount of memory to use for the driver process, i.e. where SparkContext is initialized

- **`spark.driver.memoryOverhead`**
  - Amount of non-heap memory in MiB to be allocated per driver process in cluster mode.
  - This accounts for things like VM overheads, interned strings, other native overheads, etc.
  - This is configured as a fraction of driver memory given by **`spark.driver.memoryOverheadFactor`** with minimum of 384 MiB
  - This option is currently supported on YARN, Mesos and Kubernetes.

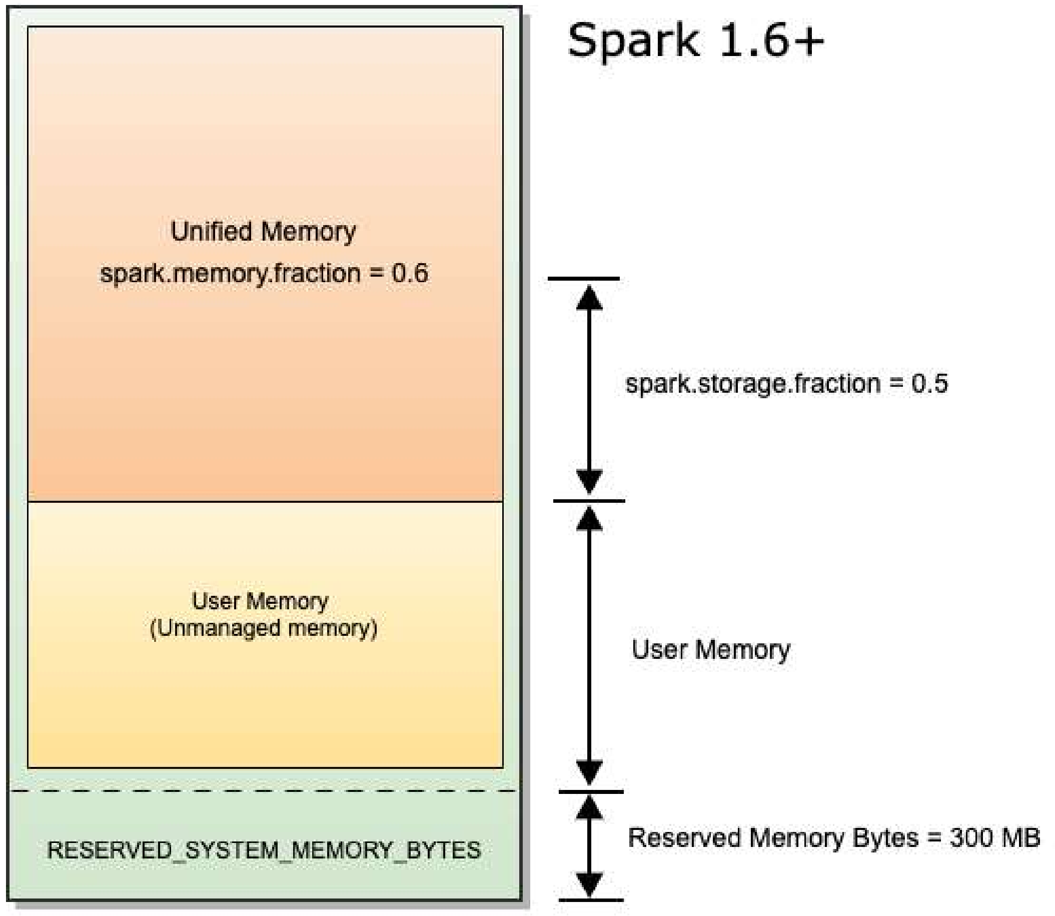
Executor memory configurations

- **`spark.executor.memory`**
  - Amount of memory to use per executor process
- **`spark.executor.pyspark.memory`**
  - The amount of memory to be allocated to PySpark in each executor, in MiB.
  - If set, PySpark memory for an executor will be limited to this amount. If not set, Spark will not limit Python's memory use and it is up to the application to avoid exceeding the overhead memory space shared with other non-JVM processes.
  - When PySpark is run in YARN or Kubernetes, this memory is added to executor resource requests.
- **`spark.executor.memoryOverhead`**
  - Amount of additional memory, in MiB, to be allocated per executor process.
  - This accounts for things like VM overheads, interned strings, other native overheads, etc.
  - This is configured as a fraction of executor memory given by **`spark.executor.memoryOverheadFactor`** with minimum of 384 MiB
  - This option is currently supported on YARN, Mesos and Kubernetes.
- **`spark.executor.memoryOverheadFactor`**

Executor memory configurations

- `spark.memory.fraction`
  - Fraction of (heap space - 300MB) used for execution and storage.
  - The lower this is, the more frequently spills and cached data eviction occur.
- `spark.memory.storageFraction`
  - Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by `spark.memory.fraction`.
  - The higher this is, the less working memory may be available to execution and tasks may spill to disk more often.
- `spark.memory.offHeap.enabled`
  - If true, Spark will attempt to use off-heap memory for certain operations. If off-heap memory use is enabled, then `spark.memory.offHeap.size` must be positive.
- `spark.memory.offHeap.size`
  - The absolute amount of memory which can be used for off-heap allocation, in bytes unless otherwise specified.
  - This setting has no impact on heap memory usage, so if your executors' total memory consumption must fit within some hard limit then be sure to shrink your JVM heap size accordingly.

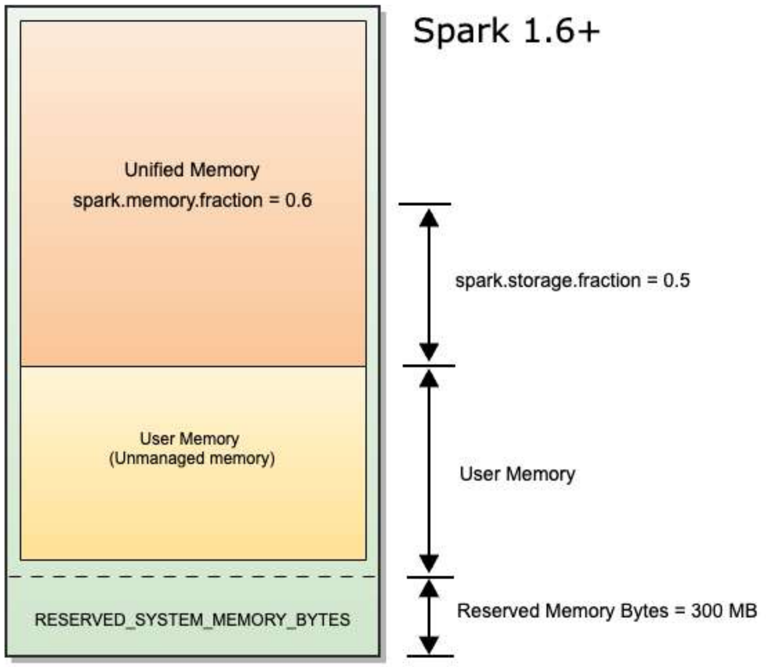
Spark memory management


Spark 1.6+
Unified Memory
spark.memory.fraction = 0.6
User Memory
(Unmanaged memory)
RESERVED_SYSTEM_MEMORY_BYTES
spark.storage.fraction = 0.5
User Memory
Reserved Memory Bytes = 300 MB

Spark memory management

Reserved memory

- This is the memory reserved by the system, and its size is hardcoded.
- As of Spark 1.6.0, its value is 300MB, which means that this 300MB of RAM does not participate in Spark memory region size calculations

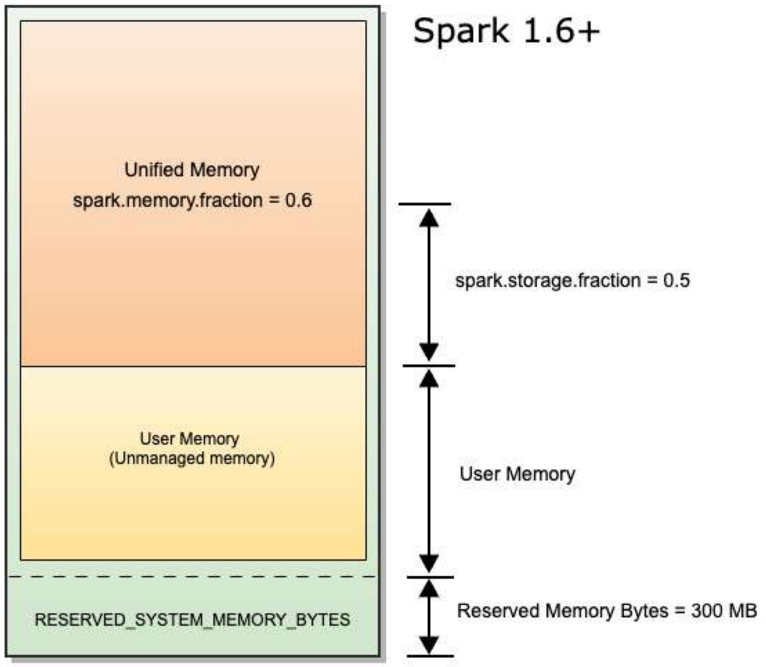


Spark 1.6+
Unified Memory
spark.memory.fraction = 0.6
spark.storage.fraction = 0.5
User Memory
(Unmanaged memory)
User Memory
RESERVED_SYSTEM_MEMORY_BYTES
Reserved Memory Bytes = 300 MB

Spark memory management

User memory

- This is the memory pool that remains after the allocation of Spark Memory
- It is completely up to you to use it in a way you like. You can store your own data structures there that would be used in RDD transformations.
- How to use this user memory is completely up to you what would be stored in this RAM and how, Spark makes no accounting on what you do there and whether you respect this boundary or not.
- Not respecting this boundary in your code might cause OOM error.


Spark 1.6+
Unified Memory
spark.memory.fraction = 0.6
spark.storage.fraction = 0.5
User Memory
(Unmanaged memory)
User Memory
RESERVED_SYSTEM_MEMORY_BYTES
Reserved Memory Bytes = 300 MB

Spark memory management

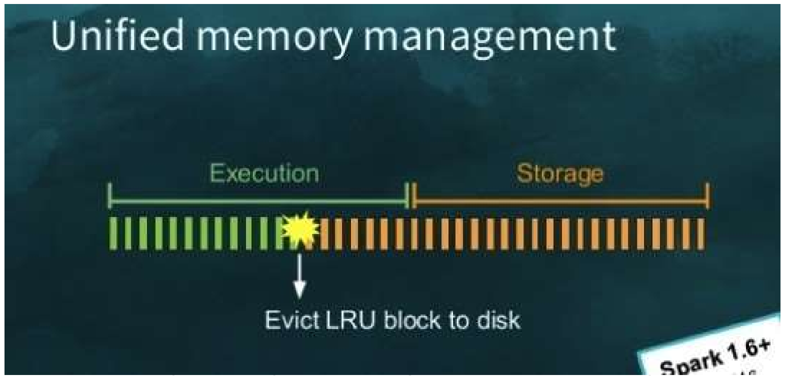
Spark memory (a.k.a unified memory)

- This is the memory pool managed by Apache Spark. Its size can be calculated as (“Java Heap” – “Reserved Memory”) * spark.memory.fraction.

- Spark tasks operate in two main memory regions:
  - Execution – used for shuffles, joins, sorts, and aggregations
  - Storage – used to cache partitions of data

- Execution memory tends to be more ‘short-lived’ than storage. It is evicted immediately after each operation, making space for the next ones.

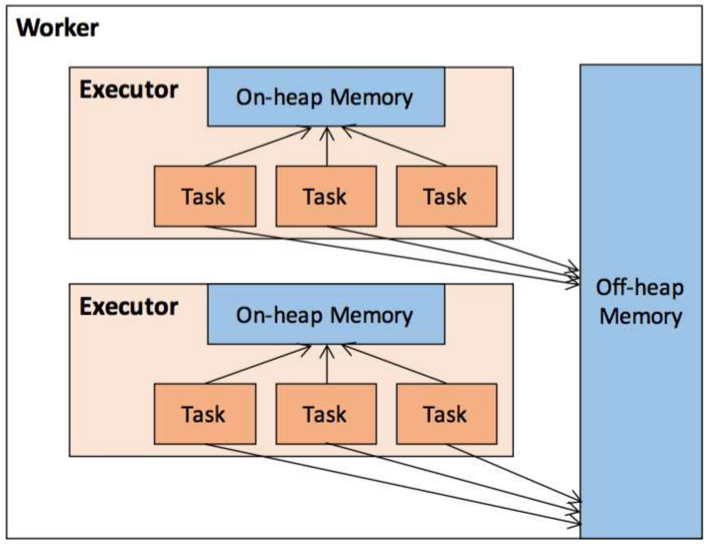
- Storage memory is used to persist RDD partitions and broadcast variables.



Spark memory management

Off-heap memory

- Off-heap refers to objects (serialized) that are managed by the OS and stored in native memory outside the heap (therefore, not subjected to GC).
- Accessing this data is slightly slower than accessing the on-heap storage but still faster than reading/writing from a disk.
- The downside is that the user has to manually deal with managing the allocated memory.

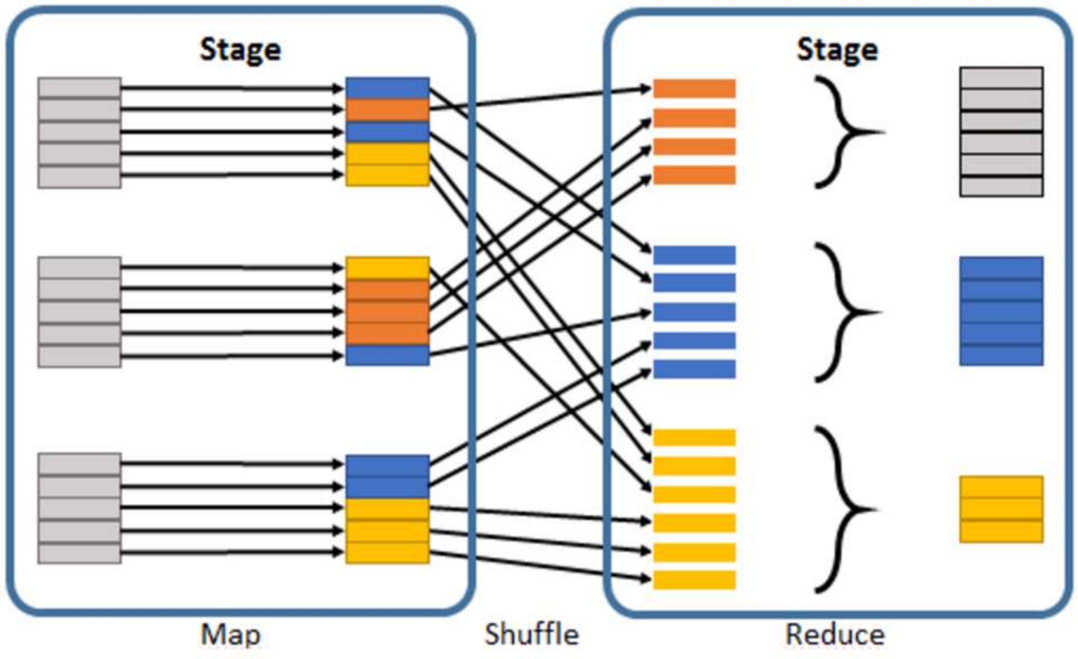


- A common problem with bigger memory configurations is that the application tends to freeze due to GC scans. The main benefit of activating off-heap memory is that we can mitigate this issue by using native system memory (which is not supervised by JVM).

Understanding Shuffle in Spark

When do shuffles happen?

In Spark, a shuffle occurs when the data needs to be redistributed across different executors or even different machines. This happens usually after wide transformations like reduceByKey, groupBy, join, etc., where the data needs to be grouped by certain keys. The shuffle process involves network IO, disk IO, and CPU overhead and can significantly affect the performance of your Spark job.



Stages & Shuffling

- At each stage boundary, data is written to disk by tasks in the *parent* stages and then fetched over the network by tasks in the *child* stage.
- Because they incur heavy disk and network I/O, stage boundaries can be expensive and should be avoided when possible.
- The number of data partitions in the parent stage may be different than the number of partitions in the child stage. Transformations that may trigger a stage boundary typically accept a *numPartitions* argument that determines how many partitions to split the data into in the child stage.
- Tuning the number of partitions at stage boundaries can often make or break an application’s performance.

Understanding Shuffle in Spark

- Default Shuffle Behavior
  - By default, Spark uses a hash-based shuffle with a fixed number of partitions (200 by default)
  - The default value of **`spark.sql.shuffle.partitions`** (which is 200) is often not optimal for all workloads. This setting determines the number of tasks that will be used for the shuffle operation and effectively, how many partitions the final shuffled data will consist of.

- Issues with Default Shuffle Partition Settings
  - Too many partitions can lead to a large number of small files, higher scheduling overhead, and overall a negative impact on performance.
  - Too few partitions can cause each task to process a large amount of data, leading to longer GC times, out of memory errors, and decreased parallelism.

- Configuring the Number of Shuffle Partitions
  - To tune Spark applications properly, it’s essential to adjust the number of shuffle partitions according to the volume of data being processed and the cluster’s capacity.

Shuffle Partition Tuning

- Setting Shuffle Partitions in SparkSession
- Enabling Adaptive Query Execution
- Tuning Shuffle Partitions Based on Data Size

When more shuffles are better ?

- There is an occasional exception to the rule of minimizing the number of shuffles.
- An extra shuffle can be advantageous to performance when it increases parallelism.
  - For example, if your data arrives in a few large unsplittable files, the partitioning dictated by the InputFormat might place large numbers of records in each partition, while not generating enough partitions to take advantage of all the available cores.
  - In this case, invoking repartition with a high number of partitions (which will trigger a shuffle) after loading the data will allow the operations that come after it to leverage more of the cluster's CPU.

Performance Tuning Considerations

Performance tuning

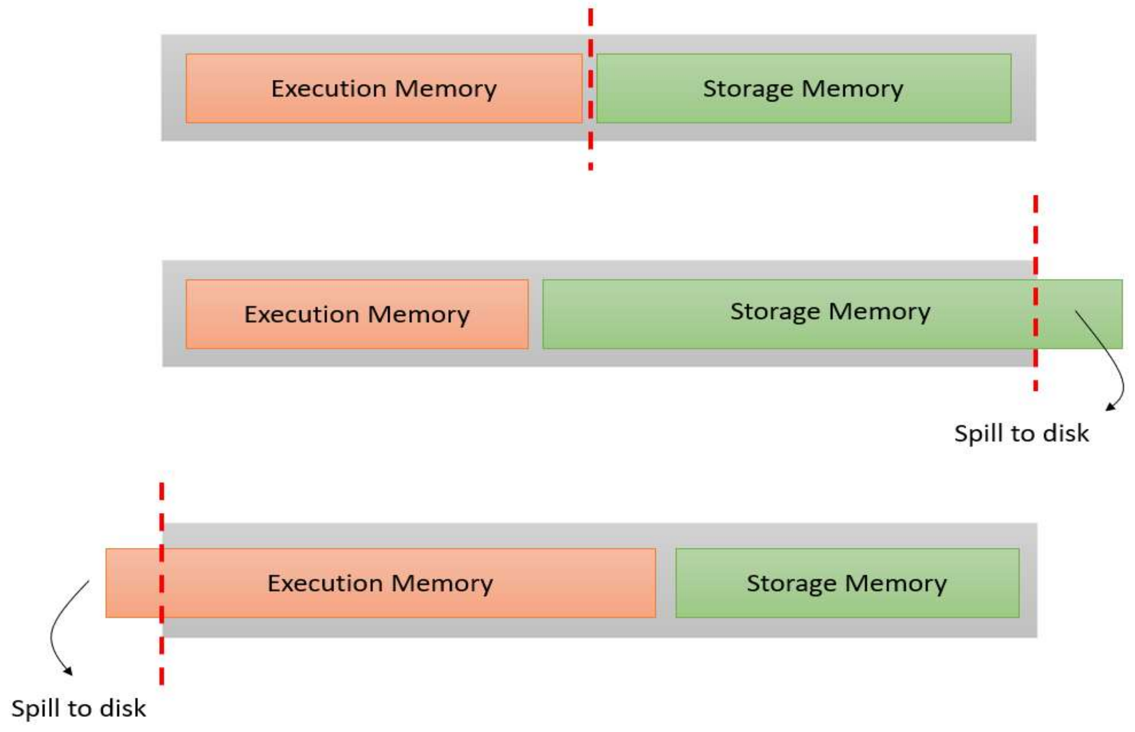
Some of the most important factors to consider for tuning the performance of spark jobs are:

1. **Spill**
2. **Skew**
3. **Shuffle**
4. **Storage**
5. **Serialization**

1. Spill

Spilling occurs when a partition is too large to fit in available RAM and writes temporary files to disk.

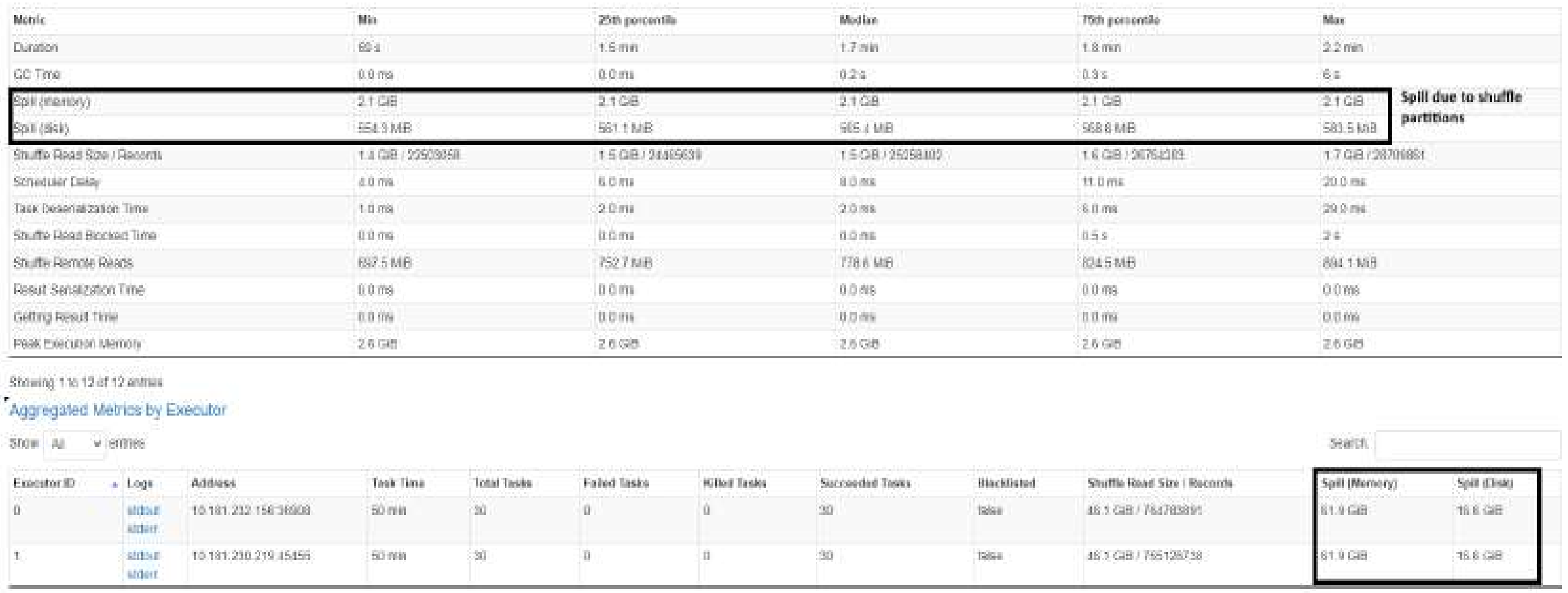
- To avoid Out of Memory (OOM) issues, an RDD is first relocated from RAM to Disk and then back to RAM. Disk reads and writes, on the other hand, can be quite expensive to compute and should thus be avoided as much as possible.



1. Spill

When performing Spark Jobs, Spill can be better understood by looking at the Spill (Memory) and Spill (Disk) values in the Spark UI.

- **Spill (Memory)**: size of the data (deserialized) as it exists in memory before it is spilled.
- **Spill (Disk)**: size of the data that gets spilled (serialized), written to disk.



Metric	Min	25th percentile	Median	75th percentile	Max
Duration	60 s	1.5 min	1.7 min	1.8 min	2.2 min
GC Time	0.0 ms	0.0 ms	0.2 s	0.3 s	6 s
Spill (memory)	2.1 GiB	2.1 GiB	2.1 GiB	2.1 GiB	2.1 GiB
Spill (disk)	554.3 MiB	561.1 MiB	565.4 MiB	568.8 MiB	583.5 MiB
Shuffle Read Size / Records	1.4 GiB / 22503058	1.5 GiB / 24465639	1.5 GiB / 25258402	1.6 GiB / 26764203	1.7 GiB / 28706881
Scheduler Delay	4.0 ms	6.0 ms	8.0 ms	11.0 ms	20.0 ms
Task Deserialization Time	1.0 ms	2.0 ms	2.0 ms	6.0 ms	29.0 ms
Shuffle Read Blocked Time	0.0 ms	0.0 ms	0.0 ms	0.5 s	2 s
Shuffle Remote Reads	697.5 MiB	752.7 MiB	778.6 MiB	824.5 MiB	894.1 MiB
Result Serialization Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Getting Result Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Peak Execution Memory	2.6 GiB	2.6 GiB	2.6 GiB	2.6 GiB	2.6 GiB

Spill due to shuffle partitions

Showing 1 to 12 of 12 entries

Aggregated Metrics by Executor

Show All entries

Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Blacklisted	Shuffle Read Size / Records	Spill (Memory)	Spill (Disk)
0	stdout stderr	10.181.232.156:36908	50 min	30	0	0	30	false	46.1 GiB / 764783891	61.9 GiB	16.6 GiB
1	stdout stderr	10.181.230.219:45456	50 min	30	0	0	30	false	46.1 GiB / 765128738	61.9 GiB	16.6 GiB

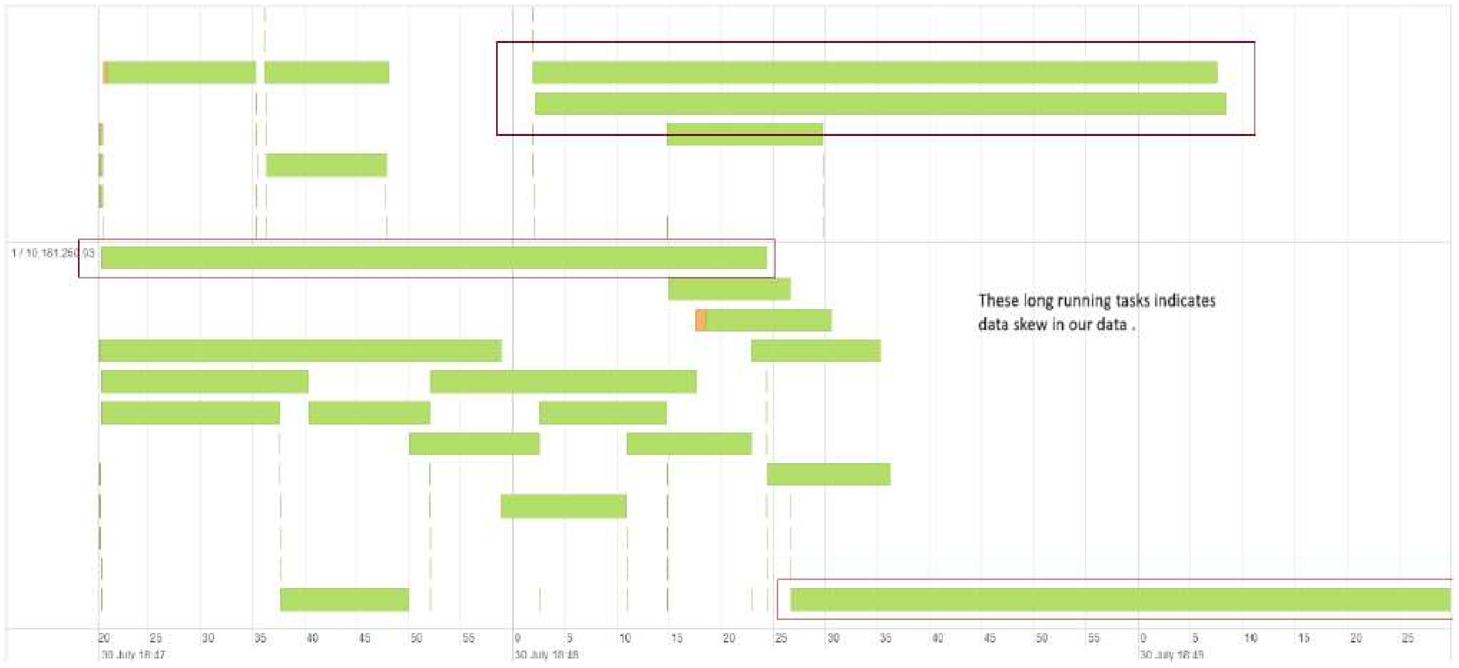
1. Spill

To reduce spill, two techniques are possible:

- Instantiating a cluster with greater memory per worker.
- Increasing the number of partitions (therefore making the existing partitions smaller) can reduce spill. Experiment with partition sizes to find the sweet spot between in-memory processing and disk usage.

2. Skew

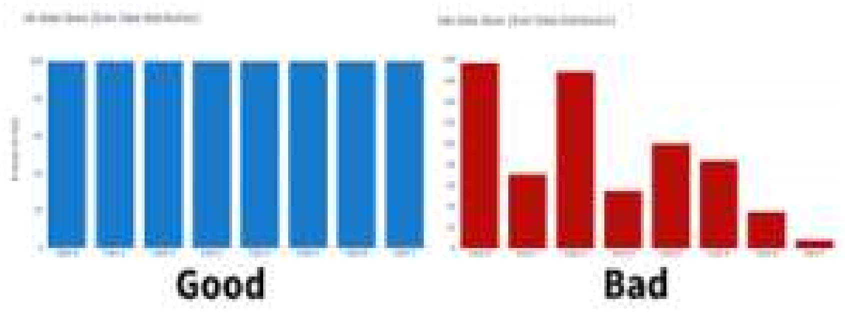
- In Spark, data is typically read in 128 MB evenly distributed partitions. When multiple transformations are applied to the data, some partitions may become significantly larger or smaller than their average.
- Skew is caused by an imbalance in partition sizes. Skew in small amounts can be entirely fine, but in excess, it can cause Spill and OOM issues.



2. Skew

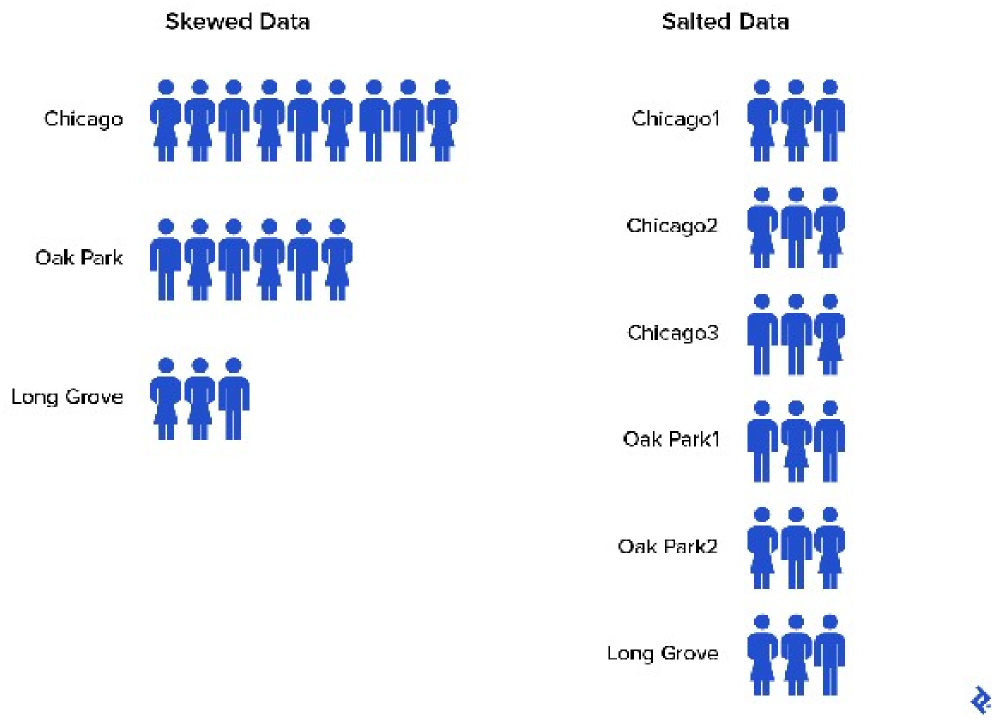
To reduce Skew, we can do the following:

- **Salting**: Distribute skewed data evenly across partitions using techniques like salting (adding a random value) before partitioning.
- **Filter before shuffle**: Reduce the amount of skewed data shuffled across the network by filtering it early on.
- **Adaptive Query Execution**: Enable and use AQE.



2. Skew – Salting technique

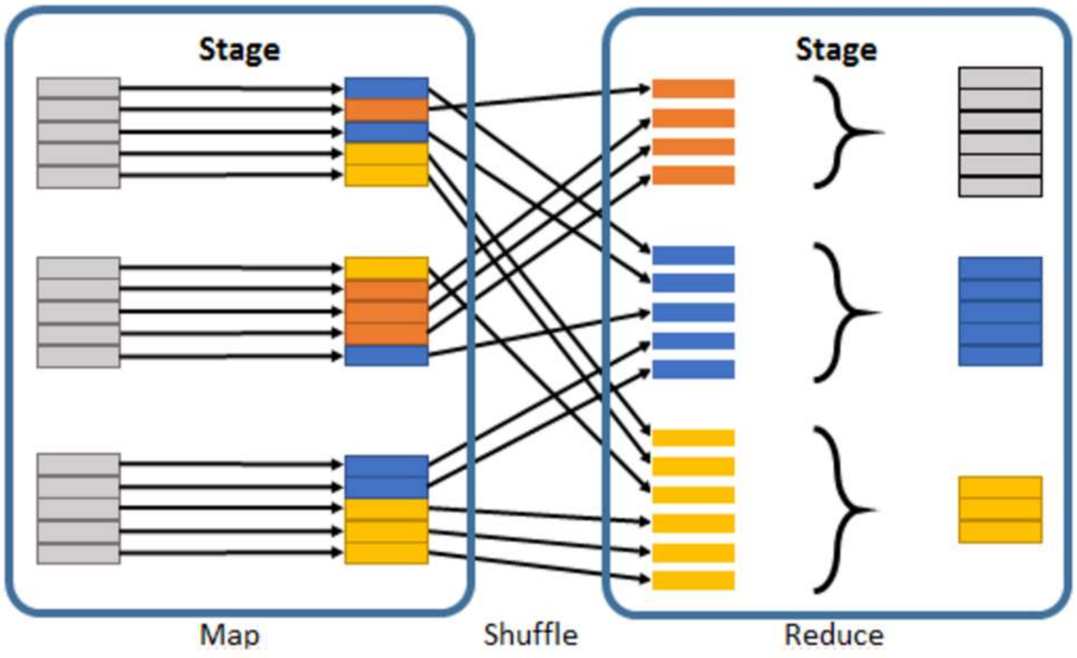
Salting is the process wherein we add some random value to the driving key of the data such that once reshuffling happens the data distribution is fairly even so that each task on its respective partition takes nearly the same amount of time thus establishing an ecosystem where resources are efficiently used.



3. Shuffle

Shuffle occurs when data is moved between executors during wide transformations.

- Skew can occur when shuffling difficulties are handled incorrectly.



3. Shuffle

- To reduce the amount of shuffling, we can follow the following techniques:
  - Creating fewer and larger workers (thus lowering network IO overheads).
  - Before shuffling, filter and project only required data to reduce its size.
  - Use Broadcast Hash Join while working with small tables.
- The ideal size of each shuffle partition is around 100-200 MB.
  - The smaller size of partitions will increase the parallel running jobs, which can improve performance, but too small of a partition will cause overhead and increase the GC time.

4. Storage

When data is saved on Disk in an inefficient manner, storage concerns develop. Storage issues have the potential to generate undue Shuffle.

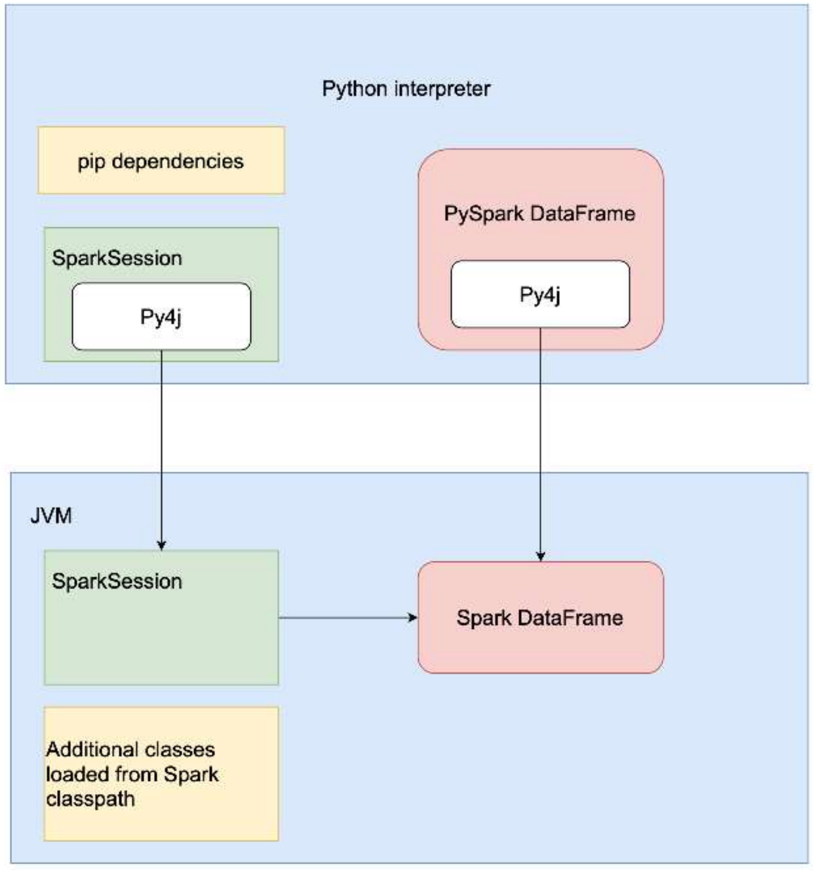
Tiny files, scanning, and schemas are three of the most common storage issues.

- **Tiny Files**: managing partition files with a size of less than 128 MB.
- **Scanning directories**: When scanning directories, we may have a huge list of files in a single directory or, in the case of heavily partitioned datasets, numerous tiers of folders. We can register it as a table to reduce the quantity of scanning.
- **Schema**: Schema issues might vary based on the file format used. To infer data types, for example, using JSON and CSV, the entire data set must be read. Instead, just one file must be read for Parquet.
- **Use columnar formats** like Parquet or ORC instead of row-oriented formats like CSV for faster reads and writes.

5. Serialization

- Serialization incorporates all of the issues involved with code distribution across clusters
    - Code is serialized, transmitted to executors, and then deserialized.

- In the case of Python, this process can be considerably more difficult because the code must be pickled and a Python interpreter instance must be allocated to each executor.



5. Serialization

Some techniques to handle serialization problems are:

- Use the Kryo serializer by default, as it's often more efficient than Java serialization.

```
spark.conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```

- Use columnar storage formats such as Parquet.
- Avoid using UDFs

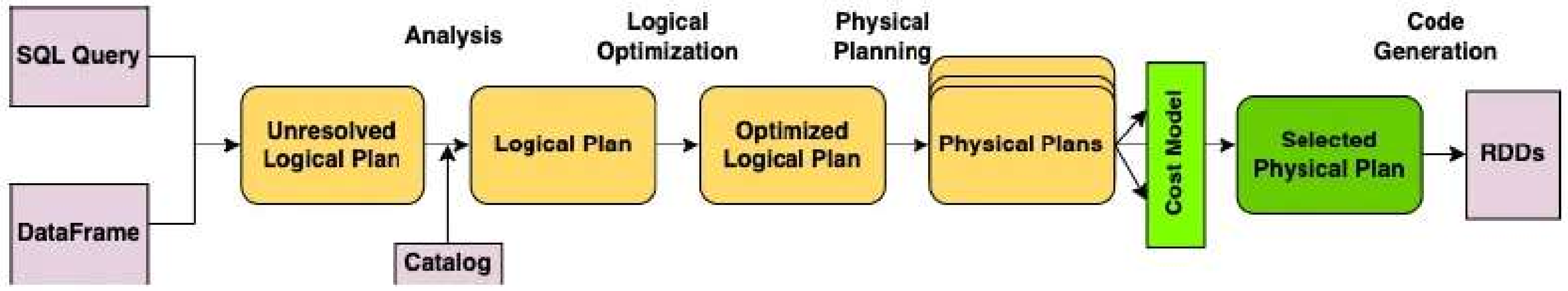
Adaptive Query Execution (AQE)

Adaptive Query Execution

- AQE Optimizer is introduced in Apache Spark 3.0 that makes use of the runtime statistics to choose the most efficient query execution plan
- Traditional query execution engines follow a fixed plan for processing the data, regardless of the data's distribution or statistics. This approach can result in suboptimal performance when dealing with large, complex datasets.
- To overcome this issue, Spark introduced the AQE optimizer, which **dynamically adapts the execution plan based on the characteristics of the data**.

Spark SQL Catalyst execution stages

- When we submit a query, DataFrame, or Dataset operations, Spark does the following in order.



AQE optimizer

- Spark SQL can turn on and off AQE by **spark.sql.adaptive.enabled** as an umbrella configuration.

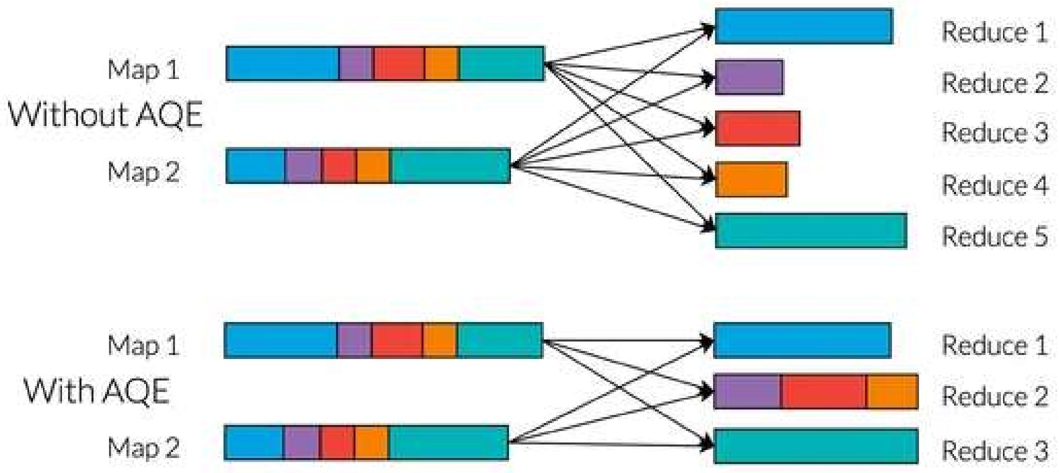
- As of Spark 3.0, there are three major features in AQE:

  - **Coalescing post-shuffle partitions**
  - **Converting sort-merge join to broadcast join**
  - **Skew join optimization.**

AQE - Coalescing post shuffle partitions

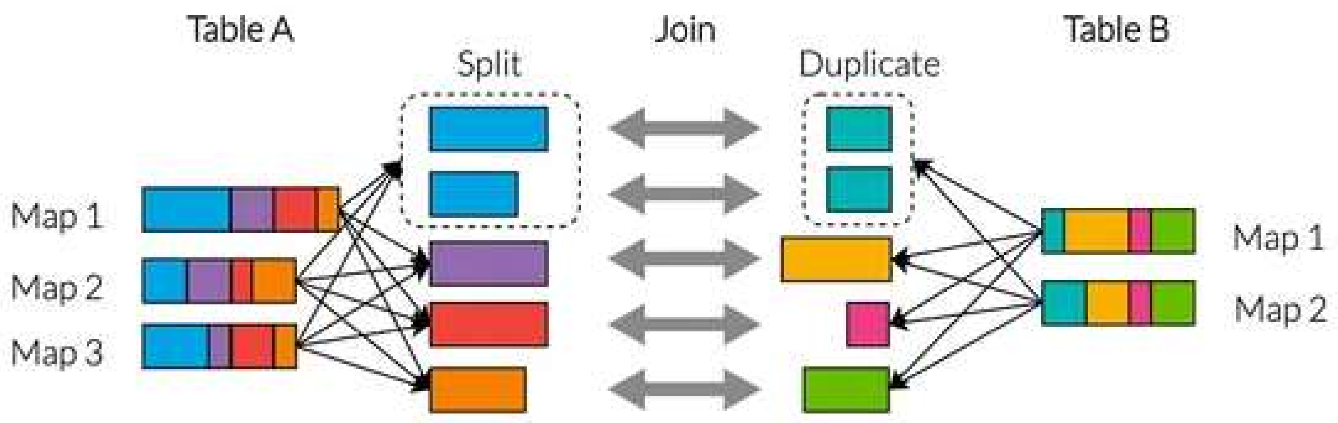
- This feature coalesces the post shuffle partitions based on the map output statistics when both **spark.sql.adaptive.enabled** and **spark.sql.adaptive.coalescePartitions.enabled** configurations are true.

- Simplifies the tuning of shuffle partition number when running queries. Spark can pick the proper shuffle partition number at runtime
    - Set a large enough initial number of shuffle partitions via **spark.sql.adaptive.coalescePartitions.initialPartitionNum** configuration.



AQE - Splitting skewed shuffle partitions

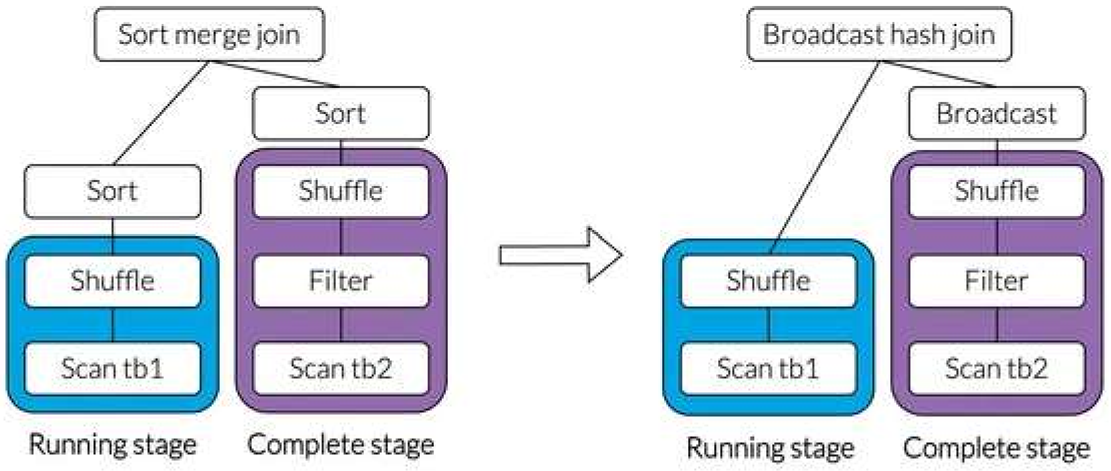
- When **spark.sql.adaptive.optimizeSkewsInRebalancePartitions.enabled** and **spark.sql.adaptive.enabled** are **true**, Spark will optimize the skewed shuffle partitions and split them to smaller ones according to the target size (**spark.sql.adaptive.advisoryPartitionSizeInBytes**), to avoid data skew.



AQE - Converting sort-merge to broadcast join

- AQE converts sort-merge join to broadcast hash join when the runtime statistics of any join side is smaller than the adaptive broadcast hash join threshold.

- This is not as efficient as planning a broadcast hash join in the first place, but it's better than keep doing the sort-merge join, as we can save the sorting of both the join sides, and read shuffle files locally to save network traffic (if **spark.sql.adaptive.localShuffleReader.enabled** is true)



Some more important considerations

1. Use DataFrames/Datasets over RDDs

- For Spark jobs, prefer using Dataset/DataFrame over RDD as Dataset and DataFrame includes several optimization modules to improve the performance of the Spark workloads.

- Using RDD directly leads to performance issues as Spark doesn’t know how to apply the optimization techniques and RDDs serialize and de-serialize the data when it distributes across a cluster.

2. Use serialized data formats

- Most of the Spark jobs run as a pipeline where one Spark job writes data into a file and another Spark job read the data, process it, and writes to another file for another Spark job to pick up. When you have such use case, prefer writing an intermediate file in serialized and optimized formats like Avro and Parquet. Transformations on these formats performs better than Text, CSV and JSON.

- Prefer Kryo serializer over default Java serializer for better performance esp. if you are using RDD API.

3. Persist intermediate data

- Use persistence to store the intermediate computation of a Spark DataFrame, so that, they can be reused in subsequent actions.
  - Spark’s persisted partitions are fault-tolerant meaning if any partition lost, it will automatically be recomputed using the original transformations that created it.

- When caching, use in-memory columnar format. By tuning the batchSize property you can also improve Spark performance. Spark provides several storage levels to store the cached data, use the once which suits your cluster.
  - spark.conf.set("spark.sql.inMemoryColumnarStorage.compressed", true)
  - spark.conf.set("spark.sql.inMemoryColumnarStorage.batchSize",10000)

4. Use correct join strategy

- Joining large datasets can be resource-intensive. Choosing the right join strategy can have a big performance impact on your join task.
- Use broadcast join when you are sure that the small DataFrame can fit in the memory of both driver and executors.
- When both DataFrames being joined are large, sort-merge join (which is default) works well in most cases.

5. Optimize parallelism

- One of the most important factors affecting the efficiency of parallel processing is the number of partitions. Choose the number of partitions so that they are not too small (and too many) or too big.

- Use repartition and coalesce to control the number of partitions.

Monitoring Spark Application Logs

Spark History Server

Spark History Server

- When you submit a Spark application, SparkContext is created which gives you Spark UI to monitor the execution of the application.

- When your application is done with the processing, Spark context will be terminated so does your Spark UI. If you wanted to see the monitoring for already finished application, we cannot do it. This is where 'Spark History Server' helps you

- Spark History Server keeps the history (event logs) of all completed applications and its runtime information which allows you to review metrics and monitor the application later in time.

Spark History Server

- Spark History server can keep the history of event logs for the following
    - All applications submitted via spark-submit
    - Submitted via REST API
    - Every spark-shell you run
    - Every pyspark shell you run
    - Submitted via Notebooks
- By default History server listens at 18080 port and you can access it from browser using http://localhost:18080/

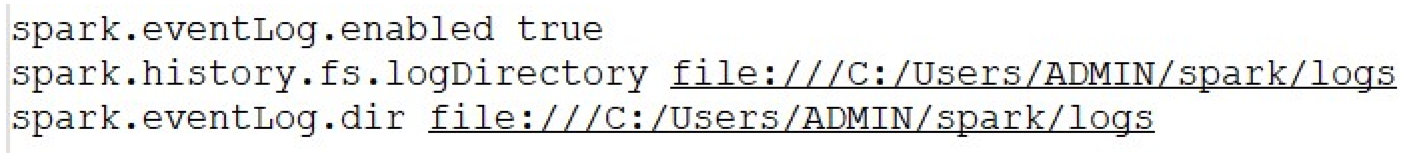
Setup History Server on Windows

- By default, the spark doesn't collect event log information. You can enable this by setting the below configs on **spark-defaults.conf**

- Open the file <Spark-Home>/conf/spark-defaults.conf.template and save it as spark-defaults.conf

- Add the following three lines at the end of spark-defaults.conf file. Replace <your-log-file-path> with a folder path where you want to store the logs. This folder should be created the has all the permissions to write and read.

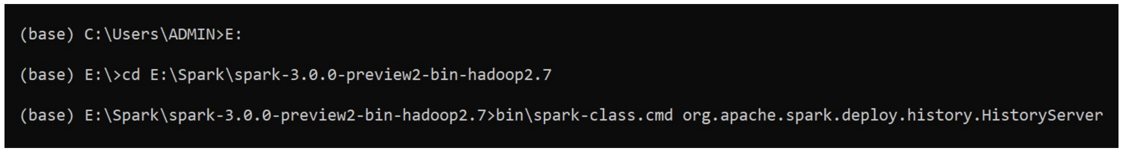
  - spark.eventLog.enabled true
  - spark.history.fs.logDirectory file:///<your-log-file-path>
  - spark.eventLog.dir file:///<your-log-file-path>



```
spark.eventLog.enabled true
spark.history.fs.logDirectory file:///C:/Users/ADMIN/spark/logs
spark.eventLog.dir file:///C:/Users/ADMIN/spark/logs
```

Starting the History Server

- Open the terminal and navigate to <Spark-Home> folder and run the following command to start the history server.
  - bin\spark-class.cmd org.apache.spark.deploy.history.HistoryServer

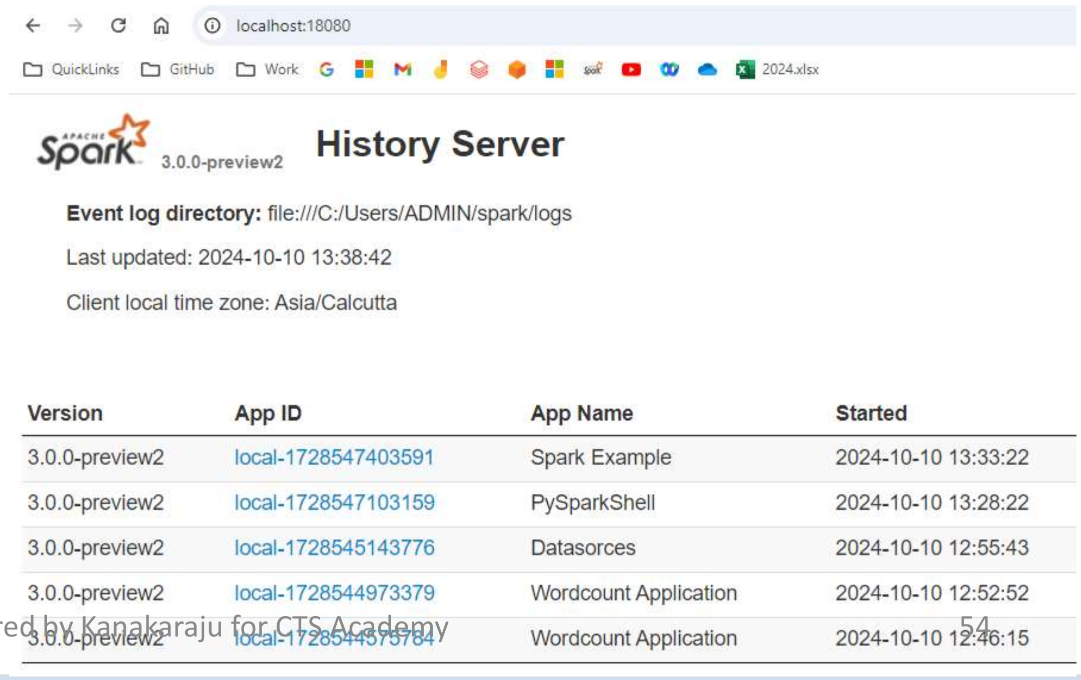


```
(base) C:\Users\ADMIN>E:

(base) E:\>cd E:\Spark\spark-3.0.0-preview2-bin-hadoop2.7

(base) E:\Spark\spark-3.0.0-preview2-bin-hadoop2.7>bin\spark-class.cmd org.apache.spark.deploy.history.HistoryServer
```

- Access history-server @ http://localhost:18080/



3.0.0-preview2 **History Server**

Event log directory: file:///C:/Users/ADMIN/spark/logs

Last updated: 2024-10-10 13:38:42

Client local time zone: Asia/Calcutta

Version	App ID	App Name	Started
3.0.0-preview2	local-1728547403591	Spark Example	2024-10-10 13:33:22
3.0.0-preview2	local-1728547103159	PySparkShell	2024-10-10 13:28:22
3.0.0-preview2	local-1728545143776	Datasorces	2024-10-10 12:55:43
3.0.0-preview2	local-1728544973379	Wordcount Application	2024-10-10 12:52:52
3.0.0-preview2	local-1728544575784	Wordcount Application	2024-10-10 12:46:15

THANK YOU