2021-11 Spark SQL Analysis

Background

This research aims at understanding different strategies to effectively write out Spark's DataFrame to persistent storage. It also involves understanding Spark's partitioning and the shuffling cost associated with different Spark's physical operators.

Previously, the write method for output Parquet file is as follows:

```python
def write_parquet(df, path, coalesce_partitions, repartition_partitions):
    df.cache()
    partitions = df.count() // 1000000 + 1
    df.repartition(partitions).write.parquet(path, mode='overwrite')
```

In this method, we first **cache()** the output DataFrame before performing a **count()** action which will trigger all previous lazy transformations and count the number of records in the output data frame. Once the count is obtained, we want to make sure that the DataFrame will be written out in a number of output files such that **each** has less than 1000000 records. By default, the **number of output files = number of current partitions**, we first calculate the desired number of partitions by **mod 1000000**. This makes sure that each partition will have less than 1000000 records. Afterwards, we perform **repartition()**to **increase/decrease** the existing number of partitions to the desired number. Repartition uses a **hash partitioner** which attempts to evenly distribute data across all the desired number of partitions. This requires shuffling the existing data between multiple executors across the network.

Bottleneck

There can be some potential bottlenecks with the above method, such as:

- Caching:
  - Currently, this **cache()** is performed right before the**count()**, which may not provide much speed benefit for this count(), unless the Data Frame is really large and span across many partitions. This is because caching incurs **overhead** as well, and it should be invoked when the **cached computation** will be used **multiple** times. In this case, it is cached only for counting
  - Also, if we only cache for counting, then we don't need all the DataFrame columns. Caching the full Data Frame may be potentially troublesome when the partition sizes are too large and will incur overhead for caching. As such, we may just select 1 column of the DataFrame to cache in order to compute the count() which can result in significant memory saving.
- Current number of **partitions**:
  - During previous transformations such as **groupBy(), union() or join(),** data will be shuffled between multiple executors and even nodes and then repartitioned into**200 partitions** by**default.**The number of shuffling partitions is set by the**spark.sql.shuffle.partitions**configuration.
  - As such,**df**may have minimally 200 partitions by the time we want to write its output out. After inspection, majority of the current output data models have 1 Parquet file only as the number of records are less than 1000000
  - At the same time, there are some models with up to a few thousand output files, as such, **counting()** might have been quite expensive
- Repartition:
  - For repartitioning, a full shuffle will need to be performed in order to evenly distribute the data. Shuffling is one of the most expensive operations in Spark as it incurs network IO. As such, we would want to minimise shuffling whenever possible

Analysis

Existing output partitions

First, most models which are running daily have quite a low number of output files. Exceptions are **coin_movement** in the image below. For monthly models such as **coin_earn_source**, the number of output files are much larger. With the current output repartitioning method, each partition will have less than 1000000 records and will output 1 corresponding parquet file.

```
/user/finance/data_model/cod_receivables/grass_region=VN/grass_date=2021-12-09
1
/user/finance/data_model/coin_earn_source/grass_region=VN/grass_date=2021-12-01
139
/user/finance/data_model/coin_movement/grass_region=VN/grass_date=2021-12-09
33
/user/finance/data_model/coin_reclass_percentage/grass_region=VN/grass_date=2021-11-30
1
/user/finance/data_model/coin_spend_on_order_not_escrow/grass_region=VN/grass_date=2021-12-01
2
/user/finance/data_model/coin_spend_redeem_voucher/grass_region=VN/grass_date=2021-12-01
1
/user/finance/data_model/coin_spend_redeem_voucher_utilization/grass_region=VN/grass_date=2021-12-01
1
/user/finance/data_model/credit_cash_loan_disbursement_success/grass_region=TH/grass_date=2021-12-09
```

An existing model:dp_auto_refund

DataFrame before **cache()** and **count()**:

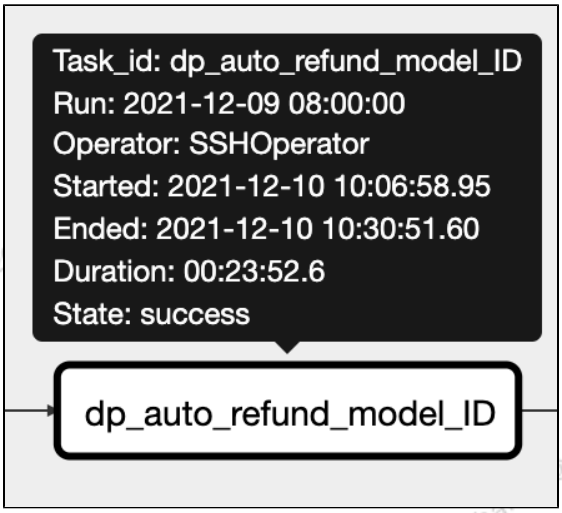
- Only 300+ records
- Partitions: 200
- Each partition: 0-10 records

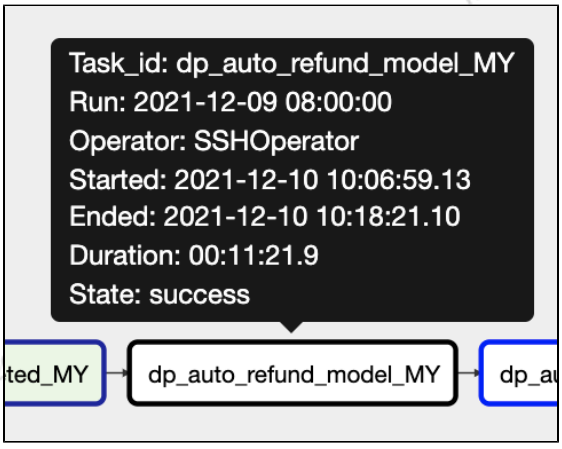
```
PARTITIONS:  200
Each partition size:  [1, 1, 5, 3, 1, 1, 1, 1, 0, 1, 0, 1, 4, 0, 1, 3, 2, 1, 1, 2, 2, 3, 1, 2, 0, 2, 2, 0, 0, 4, 1, 1, 2, 0, 1, 1, 2, 2, 1, 2, 2, 0, 1, 0, 3, 4, 3, 1, 0, 4, 1, 0, 1, 0, 1, 1, 5, 1, 2, 4, 2
, 1, 4, 3, 4, 1, 3, 0, 3, 0, 2, 0, 1, 2, 1, 5, 3, 0, 1, 2, 1, 1, 2, 4, 4, 1, 1, 2, 0, 2, 3, 3, 1, 1, 2, 3, 1, 0, 2, 2, 3, 2, 0, 1, 0, 1, 1, 2, 2, 0, 0, 3, 1, 3, 2, 1, 2, 0, 2, 3, 3, 0, 1, 6, 1, 3, 2, 1, 1
, 0, 1, 1, 0, 1, 0, 1, 2, 4, 4, 1, 2, 1, 1, 2, 3, 1, 1, 2, 1, 0, 1, 1, 3, 2, 2, 2, 0, 3, 4, 0, 1, 3, 4, 1, 1, 1, 1, 1, 3, 1, 1, 2, 3, 1, 0, 1, 1, 4, 1, 1, 4, 2, 3, 0, 2, 0, 5, 1, 2, 1, 2, 2, 2, 2, 1, 1, 3
, 2, 2, 2]
```

The number of partitions here is the result of shuffling operations. As such, with the current method, we are trying to reduce from 200 partitions to 1. Also, caching this DataFrame will not overflow memory because it is quite small.

However, despite the small output DataFrame size, the model takes quite long (>15mins), meaning that **intermediate transformations** may require costly shuffling and is the bottleneck.

Discrepancy between runtime across different regions:





Possible reason:

Huge input file only for 'ID' region.

Duration	Tasks: Succeeded/Total	Input
17 min	1797/1797	175.3 GiB

However, the cluster configuration is the same across all regions.

```
--num-executors 100 \
--conf spark.executor.memoryOverhead=8G \
--conf spark.executor.cores=2 \
```

With only 100 executors and no parameter forspark.sql.shuffle.partitions,it will be defaulted to 200. However, one of the input data is so large, processing it with insufficient number of partitions will result in partitions being too large which can lead to**disk spill** which is expensive.

In this model, disk spill has happened.

Tasks (200)

Show 20 entries Search:

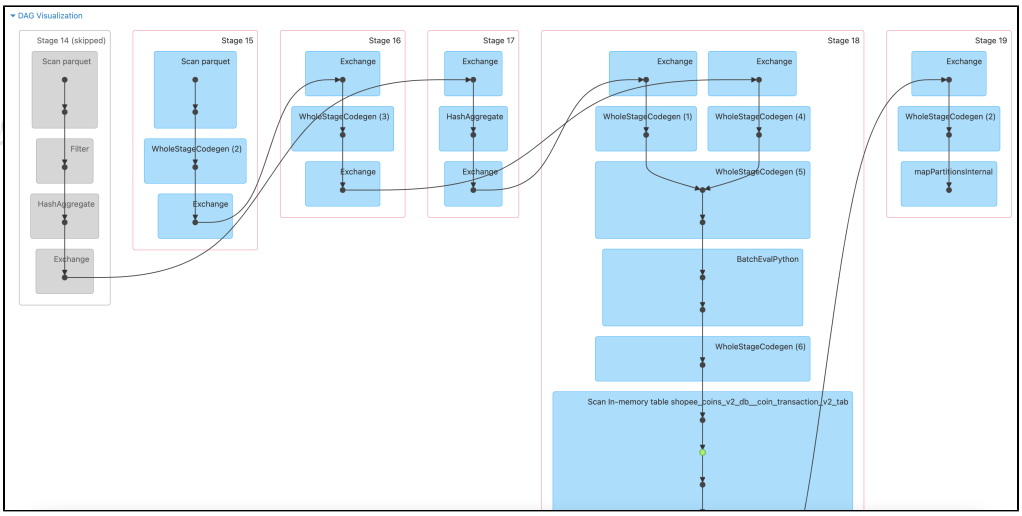
Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Write Size / Records	Shuffle Read Size / Records	Spill (Memory)	Spill (Disk)	Errors
0	9022	0	SUCCESS	PROCESS_LOCAL	98	ip-10-128-146-132.idata-server.shopee.io	stdout stderr	2021-12-13 12:04:19	1.7 min	0.4 s	358.7 MiB / 3374262	2.5 GiB / 3374262	6.6 GiB	2.6 GiB	
1	9023	0	SUCCESS	PROCESS_LOCAL	77	ip-10-128-138-230.idata-server.shopee.io	stdout stderr	2021-12-13 12:04:19	1.4 min	0.6 s	359 MiB / 3378482	2.5 GiB / 3378482	6.6 GiB	2.6 GiB	
2	9024	0	SUCCESS	PROCESS_LOCAL	51	ip-10-130-10-103.idata-server.shopee.io	stdout stderr	2021-12-13 12:04:19	1.5 min	0.6 s	358.8 MiB / 3375001	2.5 GiB / 3375001	6.6 GiB	2.6 GiB	
3	9025	0	SUCCESS	PROCESS_LOCAL	30	ip-10-128-137-230.idata-server.shopee.io	stdout stderr	2021-12-13 12:04:19	1.3 min	0.4 s	358.9 MiB / 3378336	2.5 GiB / 3378336	6.6 GiB	2.6 GiB	
4	9026	0	SUCCESS	PROCESS_LOCAL	4	ip-10-130-160-12.idata-server.shopee.io	stdout stderr	2021-12-13 12:04:19	1.2 min	0.4 s	358.5 MiB / 3374522	2.5 GiB / 3374522	6.6 GiB	2.6 GiB	
5	9027	0	SUCCESS	PROCESS_LOCAL	10	ip-10-128-137-68.idata-server.shopee.io	stdout stderr	2021-12-13 12:04:19	1.5 min	0.3 s	358.9 MiB / 3376958	2.5 GiB / 3376958	6.6 GiB	2.6 GiB	
6	9028	0	SUCCESS	PROCESS_LOCAL	5	ip-10-130-13-138.idata-server.shopee.io	stdout stderr	2021-12-13 12:04:19	1.4 min	0.4 s	358.6 MiB / 3372203	2.5 GiB / 3372203	6.6 GiB	2.6 GiB	
7	9029	0	SUCCESS	PROCESS_LOCAL	42	ip-10-128-147-68.idata-server.shopee.io	stdout stderr	2021-12-13 12:04:19	1.7 min	0.4 s	358.7 MiB / 3375533	2.5 GiB / 3375533	6.6 GiB	2.6 GiB	

Understanding bottlenecks

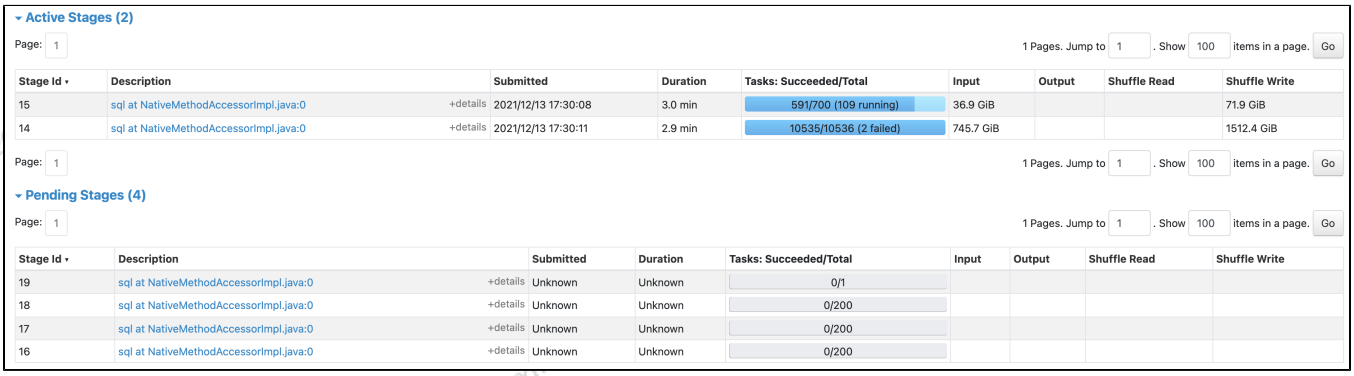
To understand the impact of the configurations on a model, we can look at**coin_earn_source** model in details. Originally, this is its current configuration:

```
sparkConfig=$(cat <<-END
    --num-executors 300 \
    --executor-memory 30G \
    --conf spark.executor.memoryOverhead=20G \
    --conf spark.executor.cores=2 \
    --conf spark.sql.adaptive.enabled=false \
```

The DAG of stages:



The number of shuffle partitions is not set, hence it is default to 200. Let's look at the main computational Job for this application:



Active Stages (2)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
15	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 17:30:08	3.0 min	591/700 (109 running)	36.9 GiB			71.9 GiB
14	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 17:30:11	2.9 min	10535/10536 (2 failed)	745.7 GiB			1512.4 GiB

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Pending Stages (4)

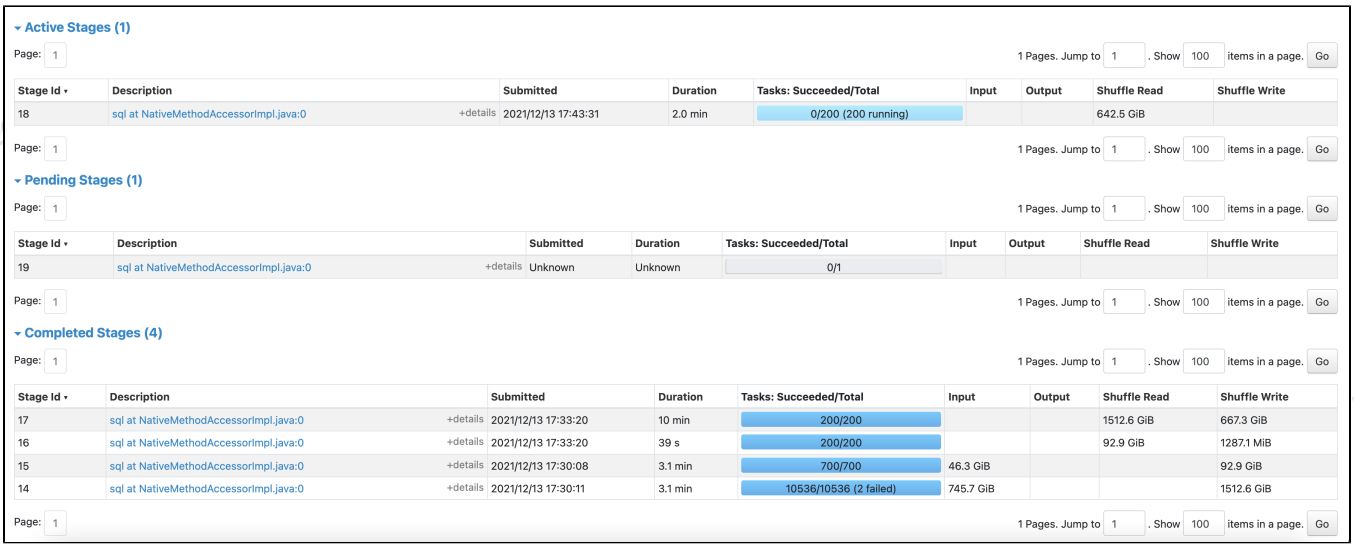
Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
19	sql at NativeMethodAccessorImpl.java:0 +details	Unknown	Unknown	0/1				
18	sql at NativeMethodAccessorImpl.java:0 +details	Unknown	Unknown	0/200				
17	sql at NativeMethodAccessorImpl.java:0 +details	Unknown	Unknown	0/200				
16	sql at NativeMethodAccessorImpl.java:0 +details	Unknown	Unknown	0/200				

- First, we can see that for**stage 14 and 15**, the task is to load the input parquet files in. The size of both files are very large (745.7GiB and around 40GiB) and thus the number of partitions to store them is 10535 and 700 respectively. Also, looking at stages 16, 17, 18 we can see that the number of tasks correspond to the number of shuffle partitions as shuffling is involved here and records with the**same join key** need to be distributed to the same partition out of 200 partitions.
- Also, since the next stages are shuffle stages, stages 14 & 15 need to prepare for shuffling by grouping the join key in blocks and write them to disk. This is equivalent to the Map side from Map Reduce framework. These blocks of data will be subsequently read by the Reduce side. Thus, the total size of blocks being written out is denoted by**Shuffle Write** which is even larger than input size (1512GiB and 71GiB respectively).

Note: Stage 15 has not completed reading data, so the total input is 46.3GiB and shuffle write is 92.9GiB.

After data has been written out, the Reduce side needs to read those blocks. This is showcased below:



Active Stages (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
18	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 17:43:31	2.0 min	0/200 (200 running)			642.5 GiB	

Pending Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
19	sql at NativeMethodAccessorImpl.java:0 +details	Unknown	Unknown	0/1				

Completed Stages (4)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
17	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 17:33:20	10 min	200/200			1512.6 GiB	667.3 GiB
16	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 17:33:20	39 s	200/200			92.9 GiB	1287.1 MiB
15	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 17:30:08	3.1 min	700/700	46.3 GiB			92.9 GiB
14	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 17:30:11	3.1 min	10536/10536 (2 failed)	745.7 GiB			1512.6 GiB

- Stages 16 and 17 thus have **Shuffle Read** values equivalent to the **Shuffle Write** values of previous stages. This is also why Spark put the shuffling at the **boundary** between stages as data needs to be transferred around resulting in **wide dependencies**.
- Looking at the duration of stage 17 it is **10 min**. This is really long so we can go into more details about this stage to observe the statistics. We can click on the link in **Description** column to go into details about this stage

Before looking at Stage 17, we note that there are **2 main** factors that slow down the execution time significantly: **Disk IO and Network IO.** To check for them, when we go into a Stage, we can look out for **2 metrics**:

Locality Level Summary: Any: 1474; Node local: 40; Rack local: 283

- Locality Level:
  - Order of speed: Process local << Node local << Rack local << Any

Spill (Memory): 1310.1 GiB

Spill (Disk): 515.3 GiB

- Disk Spill:
  - Spill (Memory): size in memory before spilling
  - Spill (Disk): size actually spilled on disk

Looking at stage 17:

Details for Stage 17 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 0 ms
Locality Level Summary: Process local: 200
Shuffle Read Size / Records: 1472.8 GiB / 18764305195
Shuffle Write Size / Records: 4.9 GiB / 160658143
Spill (Memory): 3.6 TiB
Spill (Disk): 822.2 GiB
Associated Job Ids: 14

- This snapshot is before the Stage completed, but already we can see some problem which is indicated by **Disk Spilling** which is almost 1 TiB
- Also, since data has been shuffled, the records having the **same join key** from both **joining relations** have already been read and thus they are now in the same process. So the locality level is **Process local** and there are 200 tasks processing 200 partitions **in parallel**

Within the Stage details, there are aggregated statistics for **Executor** and **Task** to check out for:

Aggregated Metrics by Executor

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Shuffle Read Size / Records	Shuffle Write Size / Records	Spill (Memory)	Spill (Disk)	Peak JVM Memory OnHeap / OffHeap	Peak Execution Memory OnHeap / OffHeap	Peak Storage Memory OnHeap / OffHeap	Peak Pool Memory Direct / Mapped
1	stdout stderr	ip-10-130-10-40.idata-server.shopee.io:24269	0.0 ms	0	0	0	0	false	7.6 GiB / 96360278		17.8 GiB	3.9 GiB	20.9 GiB / 130.3 MiB	17.8 GiB / 0.0 B	2.1 MiB / 0.0 B	4.9 MiB / 0.0 B
3	stdout stderr	ip-10-130-7-1.idata-server.shopee.io:28184	0.0 ms	0	0	0	0	false	7.6 GiB / 96350487		17.8 GiB	3.9 GiB	20.9 GiB / 130.3 MiB	17.8 GiB / 0.0 B	2.1 MiB / 0.0 B	5 MiB / 0.0 B
4	stdout stderr	ip-10-128-140-103.idata-server.shopee.io:37541	0.0 ms	0	0	0	0	false	7.6 GiB / 96359974		17.8 GiB	3.9 GiB	20.9 GiB / 131.7 MiB	17.8 GiB / 0.0 B	2.1 MiB / 0.0 B	4.9 MiB / 0.0 B
5	stdout stderr	ip-10-130-11-134.idata-server.shopee.io:21578	0.0 ms	0	0	0	0	false	7.6 GiB / 96348262		17.8 GiB	3.9 GiB	21.1 GiB / 129.3 MiB	17.8 GiB / 0.0 B	2.1 MiB / 0.0 B	4.9 MiB / 0.0 B

- For executor, we can see the Peak JVM and Peak Execution Memory, and how **Peak Execution Memory = Spill (Memory)** before Spilling. Thus, this can be an indicator on how much memory is required for each executor

sks (200)

Show 20 entries Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Scheduler Delay	Task Deserialization Time	Shuffle Read Blocked Time	Shuffle Remote Reads	Result Serialization Time	Getting Result Time	Shuffle Write Time	Shuffle Write Size / Records	Shuffle Read Size / Records	Spill (Memory)	Spill (Disk)
0	21442	0	RUNNING	PROCESS_LOCAL	144	ip-10-128-137-70.idata-server.shopee.io	stdout stderr	2021-12-13 17:33:20		0.3 s			5 s	7.5 GiB					7.6 GiB / 96317951	17.8 GiB	3.9 GiB
1	21443	0	RUNNING	PROCESS_LOCAL	100	ip-10-130-11-132.idata-server.shopee.io	stdout stderr	2021-12-13 17:33:20		0.6 s			0.0 ms	6.8 GiB					6.8 GiB / 87243738	17.8 GiB	3.9 GiB
2	21444	0	RUNNING	PROCESS_LOCAL	94	ip-10-128-146-193.idata-server.shopee.io	stdout stderr	2021-12-13 17:33:20		0.4 s			0.0 ms	6.8 GiB					6.8 GiB / 87185012	17.8 GiB	3.9 GiB
3	21445	0	RUNNING	PROCESS_LOCAL	247	ip-10-128-146-72.idata-server.shopee.io	stdout stderr	2021-12-13 17:33:20		0.4 s			0.0 ms	7.2 GiB					7.2 GiB / 92179700	17.8 GiB	3.9 GiB
4	21446	0	RUNNING	PROCESS_LOCAL	222	ip-10-130-8-228.idata-server.shopee.io	stdout stderr	2021-12-13 17:33:20		0.7 s			3.0 ms	7.5 GiB					7.6 GiB / 96351361	17.8 GiB	3.9 GiB
5	21447	0	RUNNING	PROCESS_LOCAL	160	ip-10-128-136-196.idata-server.shopee.io	stdout stderr	2021-12-13 17:33:20		0.3 s			3 s	7.5 GiB					7.6 GiB / 96356338	17.8 GiB	3.9 GiB
6	21448	0	RUNNING	PROCESS_LOCAL	205	ip-10-128-146-4.idata-server.shopee.io	stdout stderr	2021-12-13 17:33:20		0.4 s			0.0 ms	7.5 GiB					7.6 GiB / 96360165	17.8 GiB	3.9 GiB

- For each task (based on number of shuffle partitions), we can see the **Shuffle Read Size = Shuffle Remote Reads.** These remote reads mean that this task is collecting all Shuffle Write blocks from other tasks so that it has all the records of the same **key** in this partition. However, we can see that the size of each partition is too large (17.8 GiB). This is a strong indicator that the **number of partitions** is not sufficient, which results in the following issues:
  - Disk spill as seen above: With the data size so large, each partition gets too much data which is way beyond the recommended amount **(100-200Mb per partition)**

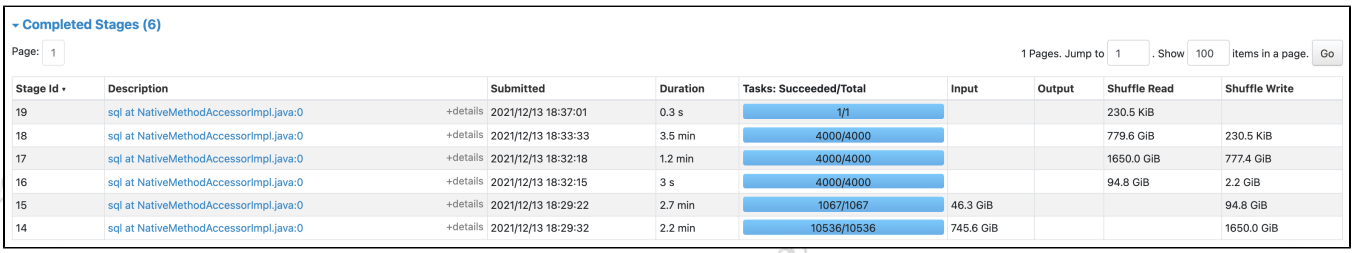
- Low degree of parallelism: Despite having **300 * 2 = 600** cores, each can execute a task on a partition, but we only have 200 partitions for shuffling, so 400 cores are **idle**.
- At the same time, when the number of partitions are too low, we have to allocate extra memory which can be expensive (30GiB executor memory + 20GiB memoryOverhead in this case)

Changing Shuffle Partition number

If we change the number of partitions to 4000 and executors to 500, the **number of partitions = 4x number of cores**

```
sparkConfig=$(cat <<-END
    --num-executors 500 \
    --executor-memory 30G \
    --conf spark.executor.memoryOverhead=20G \
    --conf spark.executor.cores=2 \
    --conf spark.sql.shuffle.partitions=4000 \
```

Runtime information for the same stages:

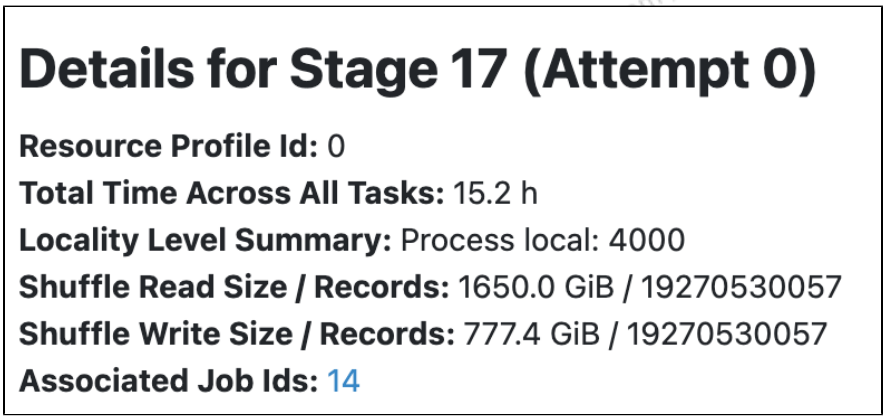


Completed Stages (6)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
19	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 18:37:01	0.3 s	1/1			230.5 KiB	
18	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 18:33:33	3.5 min	4000/4000			779.6 GiB	230.5 KiB
17	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 18:32:18	1.2 min	4000/4000			1650.0 GiB	777.4 GiB
16	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 18:32:15	3 s	4000/4000			94.8 GiB	2.2 GiB
15	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 18:29:22	2.7 min	1067/1067	46.3 GiB			94.8 GiB
14	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/13 18:29:32	2.2 min	10536/10536	745.6 GiB			1650.0 GiB

- Stage 17 now only takes up 1.2min as opposed to 10min previously. Even though the **number of executors** have been increased, we can then save on **memory and time.**



Details for Stage 17 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 15.2 h
Locality Level Summary: Process local: 4000
Shuffle Read Size / Records: 1650.0 GiB / 19270530057
Shuffle Write Size / Records: 777.4 GiB / 19270530057
Associated Job Ids: 14

- Stage 17 now has no **spill**

Aggregated Metrics by Executor

Show 20 entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Shuffle Read Size / Records	Shuffle Write Size / Records	Peak JVM Memory OnHeap / OffHeap	Peak Execution Memory OnHeap / OffHeap	Peak Storage Memory OnHeap / OffHeap	Peak Pool Memory Direct / Mapped
1	stdout stderr	ip-10-128-146-197.idata-server.shopee.io:37394	2.0 min	8	0	0	8	false	3.3 GiB / 38540541	1.6 GiB / 38540541	4.5 GiB / 129.9 MiB	3.2 GiB / 0.0 B	2.3 MiB / 0.0 B	4.8 MiB / 0.0 B
2	stdout stderr	ip-10-128-146-226.idata-server.shopee.io:37482	1.9 min	8	0	0	8	false	3.3 GiB / 38536587	1.6 GiB / 38536587	4.3 GiB / 129.3 MiB	3.3 GiB / 0.0 B	2.3 MiB / 0.0 B	4.9 MiB / 0.0 B
3	stdout stderr	ip-10-128-141-134.idata-server.shopee.io:41520	1.9 min	8	0	0	8	false	3.3 GiB / 38550767	1.6 GiB / 38550767	4.2 GiB / 130.7 MiB	3.3 GiB / 0.0 B	2.3 MiB / 0.0 B	4.9 MiB / 0.0 B
4	stdout stderr	ip-10-128-146-39.idata-server.shopee.io:23327	1.9 min	8	0	0	8	false	3.3 GiB / 38540297	1.6 GiB / 38540297	4.4 GiB / 125.9 MiB	3.3 GiB / 0.0 B	1.3 MiB / 0.0 B	4.9 MiB / 0.0 B
5	stdout stderr	ip-10-130-10-101.idata-server.shopee.io:40782	1.7 min	8	0	0	8	false	3.3 GiB / 38539870	1.6 GiB / 38539870	4.3 GiB / 128.2 MiB	3.4 GiB / 0.0 B	2.3 MiB / 0.0 B	4.9 MiB / 0.0 B
6	stdout stderr	ip-10-128-147-233.idata-server.shopee.io:35852	1.8 min	6	0	0	6	false	2.5 GiB / 28906579	1.2 GiB / 28906579	4.4 GiB / 128.9 MiB	3.3 GiB / 0.0 B	2.3 MiB / 0.0 B	4.8 MiB / 0.0 B
7	stdout stderr	ip-10-128-137-34.idata-server.shopee.io:37749	1.9 min	10	0	0	10	false	4.1 GiB / 48174198	1.9 GiB / 48174198	4.2 GiB / 128.9 MiB	3.2 GiB / 0.0 B	2.3 MiB / 0.0 B	4.8 MiB / 0.0 B
8	stdout stderr	ip-10-128-147-73.idata-server.shopee.io:32957	1.8 min	8	0	0	8	false	3.3 GiB / 38543430	1.6 GiB / 38543430	4.3 GiB / 127.1 MiB	3.4 GiB / 0.0 B	2.3 MiB / 0.0 B	4.9 MiB / 0.0 B
9	stdout stderr	ip-10-128-136-201.idata-server.shopee.io:24614	1.9 min	8	0	0	8	false	3.3 GiB / 38536773	1.6 GiB / 38536773	4.2 GiB / 129.3 MiB	3.3 GiB / 0.0 B	2.3 MiB / 0.0 B	4.8 MiB / 0.0 B

- Each executor only have **Peak Execution Memory** of **3+ GiB**

Tasks (4000)

Show 20 entries Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Scheduler Delay	Task Deserialization Time	Shuffle Read Blocked Time	Shuffle Remote Reads	Result Serialization Time	Getting Result Time	Peak Execution Memory	Shuffle Write Time	Shuffle Write Size / Records	Shuffle Read Size / Records
0	25609	0	SUCCESS	PROCESS_LOCAL	189	ip-10-128-144-201.idata-server.shopee.io	stdout stderr	2021-12-14 12:47:44	18 s	0.4 s	50.0 ms	30.0 ms	0.0 ms	421.1 MiB			2.3 GiB	0.3 s	198.8 MiB / 4813191	421.9 MiB / 4813191
1	25610	0	SUCCESS	PROCESS_LOCAL	163	ip-10-130-11-135.idata-server.shopee.io	stdout stderr	2021-12-14 12:47:44	15 s	0.5 s	49.0 ms	33.0 ms	0.0 ms	421.2 MiB			2.3 GiB	0.3 s	198.9 MiB / 4813938	422 MiB / 4813938
2	25611	0	SUCCESS	PROCESS_LOCAL	321	ip-10-130-160-105.idata-server.shopee.io	stdout stderr	2021-12-14 12:47:44	16 s	0.4 s	49.0 ms	33.0 ms	0.0 ms	421.7 MiB			2.3 GiB	0.2 s	199.1 MiB / 4819502	422.6 MiB / 4819502
3	25612	0	SUCCESS	PROCESS_LOCAL	248	ip-10-130-160-108.idata-server.shopee.io	stdout stderr	2021-12-14 12:47:44	14 s	0.3 s	52.0 ms	41.0 ms	0.0 ms	421.7 MiB			2.3 GiB	0.2 s	199.1 MiB / 4819429	422.6 MiB / 4819429
4	25613	0	SUCCESS	PROCESS_LOCAL	481	ip-10-130-11-195.idata-server.shopee.io	stdout stderr	2021-12-14 12:47:44	15 s	0.4 s	52.0 ms	30.0 ms	0.0 ms	421.3 MiB			2.3 GiB	0.3 s	198.9 MiB / 4815926	422.3 MiB / 4815926
5	25614	0	SUCCESS	PROCESS_LOCAL	62	ip-10-128-137-106.idata-server.shopee.io	stdout stderr	2021-12-14 12:47:44	15 s	0.3 s	54.0 ms	25.0 ms	0.0 ms	421.4 MiB			2.3 GiB	0.3 s	198.9 MiB / 4816607	422.2 MiB / 4816607
6	25615	0	SUCCESS	PROCESS_LOCAL	3	ip-10-128-141-134.idata-server.shopee.io	stdout stderr	2021-12-14 12:47:44	15 s	0.2 s	50.0 ms	0.6 s	0.0 ms	421.8 MiB			2.3 GiB	0.7 s	199.2 MiB / 4822403	422.7 MiB / 4822403

- For each task at each partition, the**PeakExecution Memory**is also only **2.3 GiB**
- The **ShuffleRemote Reads** is also much smaller of about **400 MiB**

Summary Metrics for 4000 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	8 s	12 s	14 s	15 s	29 s
GC Time	0.0 ms	0.2 s	0.2 s	0.2 s	0.8 s
Shuffle Read Size / Records	421.6 MiB / 4807960	422.2 MiB / 4816129	422.4 MiB / 4817624	422.5 MiB / 4819133	423.1 MiB / 4825351
Shuffle Write Size / Records	198.6 MiB / 4807960	198.9 MiB / 4816129	199 MiB / 4817624	199.1 MiB / 4819133	199.3 MiB / 4825351
Scheduler Delay	3.0 ms	5.0 ms	6.0 ms	36.0 ms	0.4 s
Task Deserialization Time	2.0 ms	3.0 ms	5.0 ms	19.0 ms	1 s
Shuffle Read Blocked Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	2 s
Shuffle Remote Reads	420.7 MiB	421.4 MiB	421.5 MiB	421.7 MiB	422.3 MiB
Result Serialization Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	14.0 ms
Getting Result Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Peak Execution Memory	2.3 GiB	2.3 GiB	2.3 GiB	2.3 GiB	2.3 GiB
Shuffle Write Time	0.1 s	0.2 s	0.3 s	0.4 s	14 s

Also, there is potential another**indicator**which we should look out for.

Completed Stages (4)

Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
17	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/14 14:48:49	3.8 min	4000/4000			94.8 GiB	2.2 GiB
16	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/14 14:51:07	1.5 min	4000/4000			1650.0 GiB	777.4 GiB
15	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/14 14:48:03	3.0 min	10536/10536	745.6 GiB			1650.0 GiB
14	sql at NativeMethodAccessorImpl.java:0 +details	2021/12/14 14:48:05	43 s	1067/1067	46.3 GiB			94.8 GiB

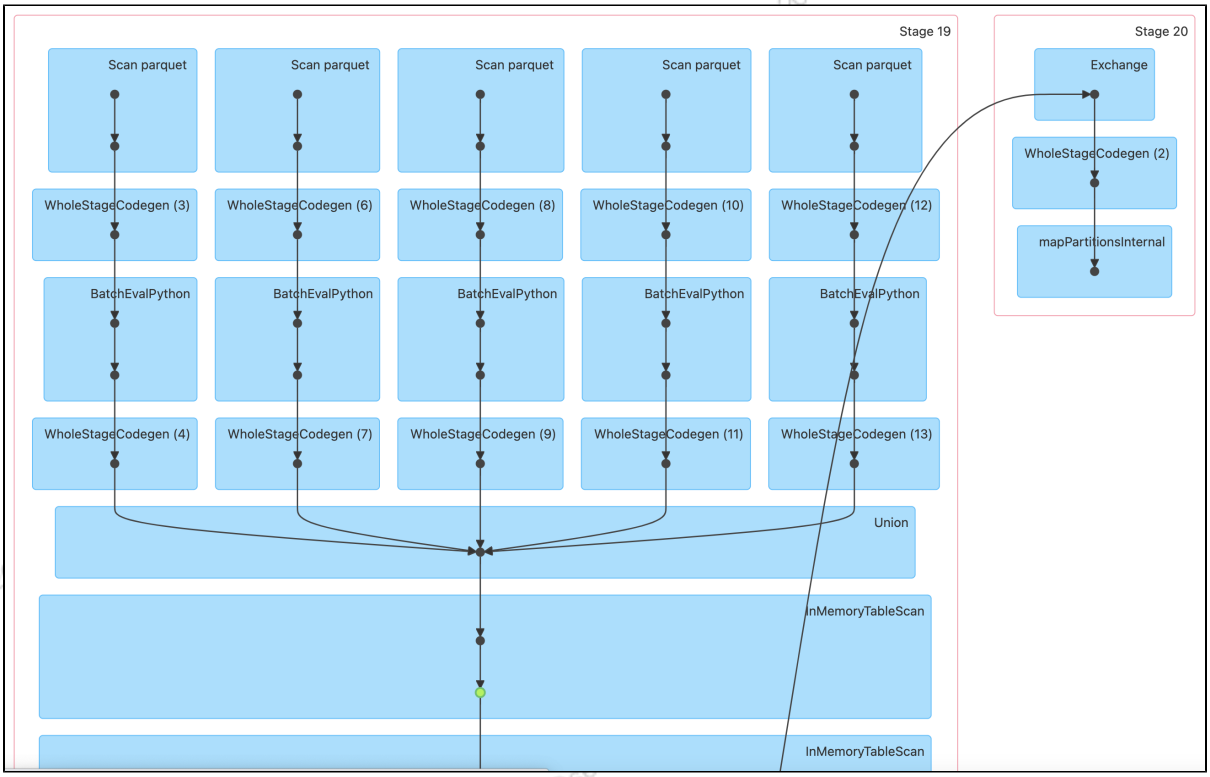
- Here for stages 16 and 17, even though stage 16 has lower memory usage, its duration is longer

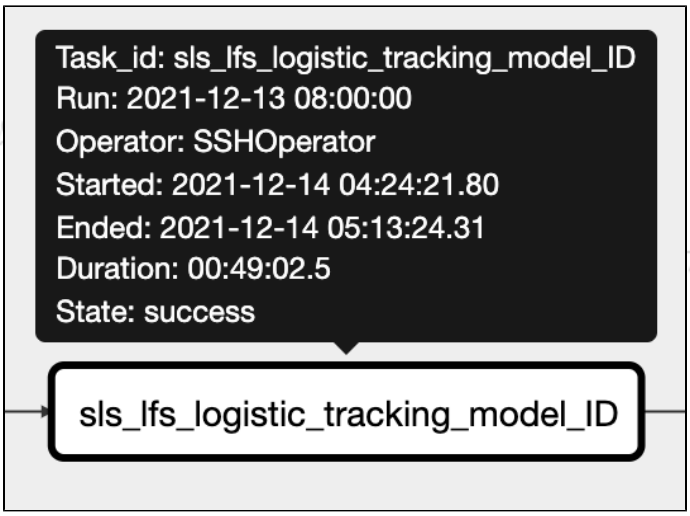
Summary Metrics for 4000 Completed Tasks

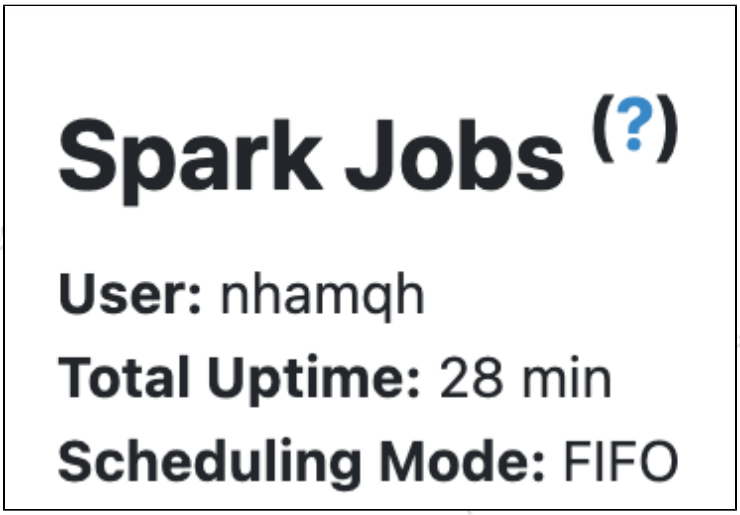
Metric	Min	25th percentile	Median	75th percentile	Max
Task Deserialization Time	2.0 ms	3.0 ms	5.0 ms	7.0 ms	0.4 s
Duration	0.3 s	0.5 s	0.7 s	26 s	1.3 min
GC Time	0.0 ms	10.0 ms	15.0 ms	20.0 ms	0.2 s
Result Serialization Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	44.0 ms
Getting Result Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Scheduler Delay	2.0 ms	4.0 ms	4.0 ms	5.0 ms	0.2 s
Peak Execution Memory	195 MiB	195 MiB	195 MiB	195 MiB	195 MiB
Shuffle Read Size / Records	23.9 MiB / 53065	24.2 MiB / 53729	24.3 MiB / 53888	24.3 MiB / 54046	24.6 MiB / 54743
Shuffle Read Blocked Time	0.0 ms	0.0 ms	0.1 s	25 s	1.3 min
Shuffle Remote Reads	23.9 MiB	24.1 MiB	24.2 MiB	24.3 MiB	24.6 MiB
Shuffle Write Size / Records	580 KiB / 53065	584.4 KiB / 53729	585.5 KiB / 53888	586.6 KiB / 54046	591.5 KiB / 54743
Shuffle Write Time	6.0 ms	12.0 ms	14.0 ms	15.0 ms	0.3 s

- Going into this stage details, we can see that for the**Duration row**, the **Max** duration is much more than**Median**, this can be an indication of**data skewness**. This is when a**join key**value is much more than others, hence when they are shuffled to a partition, that partition size is much larger than other partitions. This partition can then be a bottleneck as it slows down the entire stage.

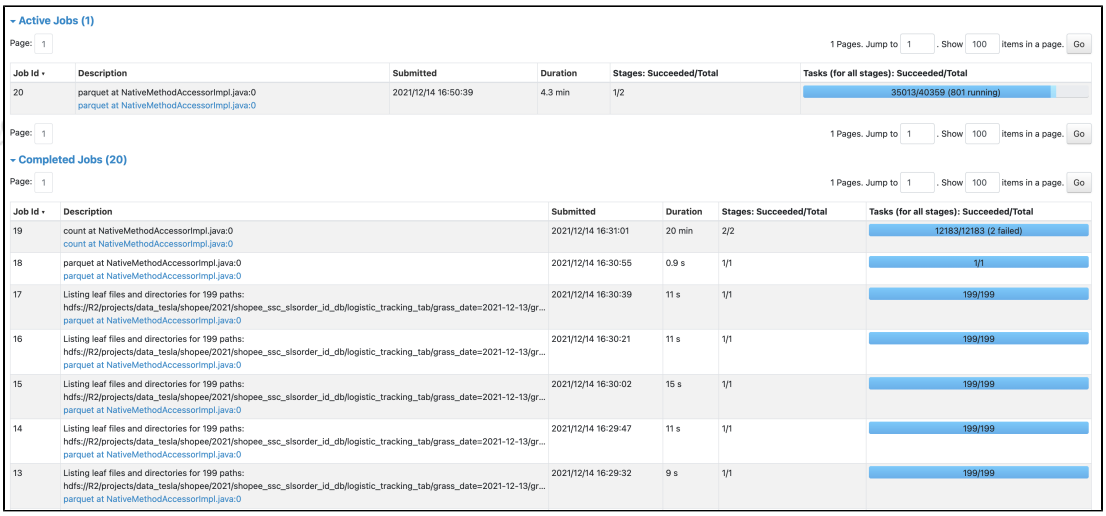
Effect of Caching on some models







ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
102	Union :- *(1) Project [cast(log_id#437L as decimal(20,0)) AS log_id#512, cast(channel_id#439L as decimal(20,0)) AS channel_id#513, channel_status#443, cast(local_update_time#440L as string) AS local_update_time#514, status#444L, tracking_detail#448, tracking_code#451, tracking_name#452] : +- *(1) Scan ExistingRDD[id#436L,log_id#437L,action_id#438L,channel_id#439L,local_update_time#440L,location#441,message#442,channel_status#443,status#444L,flag#445L,action_status#446L,store_info#447,tracking_detail#448,remark#449,ctime#450L,tracking_code#451,tracking_name#452,tracking_type#453,grass_sharding#454] :- *(2) Project [cast(log_id#475L as decimal(20,0)) AS log_id#526, cast(channel_id#477L as decimal(20,0)) AS channel_id#527, channel_status#481, cast(local_update_time#478L as string) AS local_update_time#528, status#482L, tracking_detail#486, tracking_code#489, tracking_name#490] : +- *(2) Scan ExistingRDD[id#474L,log_id#475L,action_id#476L,channel_id#477L,local_update_time#478L,location#479,message#480,channe...	Disk Memory Deserialized 1x Replicated	12182	100%	1540.8 GiB	0.0 B
113	InMemoryTableScan [log_id#512] +- InMemoryRelation [log_id#512, channel_id#513, channel_status#443, local_update_time#514, status#444L, tracking_detail#448, tracking_code#451, tracking_name#452], StorageLevel(disk, memory, deserialized, 1 replicas) +- Union :- *(1) Project [cast(log_id#437L as decimal(20,0)) AS log_id#512, cast(channel_id#439L as decimal(20,0)) AS channel_id#513, channel_status#443, cast(local_update_time#440L as string) AS local_update_time#514, status#444L, tracking_detail#448, tracking_code#451, tracking_name#452] : +- *(1) Scan ExistingRDD[id#436L,log_id#437L,action_id#438L,channel_id#439L,local_update_time#440L,location#441,message#442,channel_status#443,status#444L,flag#445L,action_status#446L,store_info#447,tracking_detail#448,remark#449,ctime#450L,tracking_code#451,tracking_name#452,tracking_type#453,grass_sharding#454] :- *(2) Project [cast(log_id#475L as decimal(20,0)) AS log_id#526, cast(channel_id#477L as decimal(20,0)) AS channe...	Disk Memory Deserialized 1x Replicated	12182	100%	248.6 GiB	0.0 B



Active Jobs (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
20	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2021/12/14 16:50:39	4.3 min	1/2	35013/40359 (801 running)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Completed Jobs (20)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
19	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/12/14 16:31:01	20 min	2/2	12183/12183 (2 failed)
18	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2021/12/14 16:30:55	0.9 s	1/1	1/1
17	Listing leaf files and directories for 199 paths: hdfs://R2/projects/data_tesla/shopee/2021/shopee_ssc_slsorder_id_db/logistic_tracking_tab/grass_date=2021-12-13/gr... parquet at NativeMethodAccessorImpl.java:0	2021/12/14 16:30:39	11 s	1/1	199/199
16	Listing leaf files and directories for 199 paths: hdfs://R2/projects/data_tesla/shopee/2021/shopee_ssc_slsorder_id_db/logistic_tracking_tab/grass_date=2021-12-13/gr... parquet at NativeMethodAccessorImpl.java:0	2021/12/14 16:30:21	11 s	1/1	199/199
15	Listing leaf files and directories for 199 paths: hdfs://R2/projects/data_tesla/shopee/2021/shopee_ssc_slsorder_id_db/logistic_tracking_tab/grass_date=2021-12-13/gr... parquet at NativeMethodAccessorImpl.java:0	2021/12/14 16:30:02	15 s	1/1	199/199
14	Listing leaf files and directories for 199 paths: hdfs://R2/projects/data_tesla/shopee/2021/shopee_ssc_slsorder_id_db/logistic_tracking_tab/grass_date=2021-12-13/gr... parquet at NativeMethodAccessorImpl.java:0	2021/12/14 16:29:47	11 s	1/1	199/199
13	Listing leaf files and directories for 199 paths: hdfs://R2/projects/data_tesla/shopee/2021/shopee_ssc_slsorder_id_db/logistic_tracking_tab/grass_date=2021-12-13/gr... parquet at NativeMethodAccessorImpl.java:0	2021/12/14 16:29:32	9 s	1/1	199/199

Ideas for Improvements

Caching Just Enough

Caching a single column such as:

```
small_df = df.select(df.columns[0])
cached_small_df = small_df.cache()
num_records = cached_small_df.count()
num_partitions = num_records // 1000000 + 1
```

The following displays the caching size difference between caching only a single column vs the entire DataFrame. The more columns there are, the more cost saving it would be. This is also especially good as a safeguard against**disk spill from caching** as well.

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
102	Union :- *(1) Project [cast(log_id#437L as decimal(20,0)) AS log_id#512, cast(channel_id#439L as decimal(20,0)) AS channel_id#513, channel_status#443, cast(local_update_time#440L as string) AS local_update_time#514, status#444L, tracking_detail#448, tracking_code#451, tracking_name#452] : +- *(1) Scan ExistingRDD[id#436L,log_id#437L,action_id#438L,channel_id#439L,local_update_time#440L,location#441,message#442,channel_status#443,status#444L,flag#445L,action_status#446L,store_info#447,tracking_detail#448,remark#449,ctime#450L,tracking_code#451,tracking_name#452,tracking_type#453,grass_sharding#454] :- *(2) Project [cast(log_id#475L as decimal(20,0)) AS log_id#526, cast(channel_id#477L as decimal(20,0)) AS channel_id#527, channel_status#481, cast(local_update_time#478L as string) AS local_update_time#528, status#482L, tracking_detail#486, tracking_code#489, tracking_name#490] : +- *(2) Scan ExistingRDD[id#474L,log_id#475L,action_id#476L,channel_id#477L,local_update_time#478L,location#479,message#480,channe...	Disk Memory Deserialized 1x Replicated	12182	100%	1540.8 GiB	0.0 B
113	InMemoryTableScan [log_id#512] +- InMemoryRelation [log_id#512, channel_id#513, channel_status#443, local_update_time#514, status#444L, tracking_detail#448, tracking_code#451, tracking_name#452], StorageLevel(disk, memory, deserialized, 1 replicas) +- Union :- *(1) Project [cast(log_id#437L as decimal(20,0)) AS log_id#512, cast(channel_id#439L as decimal(20,0)) AS channel_id#513, channel_status#443, cast(local_update_time#440L as string) AS local_update_time#514, status#444L, tracking_detail#448, tracking_code#451, tracking_name#452] : +- *(1) Scan ExistingRDD[id#436L,log_id#437L,action_id#438L,channel_id#439L,local_update_time#440L,location#441,message#442,channel_status#443,status#444L,flag#445L,action_status#446L,store_info#447,tracking_detail#448,remark#449,ctime#450L,tracking_code#451,tracking_name#452,tracking_type#453,grass_sharding#454] :- *(2) Project [cast(log_id#475L as decimal(20,0)) AS log_id#526, cast(channel_id#477L as decimal(20,0)) AS channe...	Disk Memory Deserialized 1x Replicated	12182	100%	248.6 GiB	0.0 B

- This is**sls_lfs_logistic_tracking**model cached with only 8 columns. Already the size has been significantly reduced from **1540.8 GiB to 248.6 GiB**. If caching is only for counting, then this results in a significant saving of memory because the cache type is by default **in memory**.

Setting the Right Number of Partitions

https://nealanalytics.com/blog/databricks-spark-jobs-optimization-techniques-shuffle-partition-technique-part-1/

The number of partitions for shuffling is also important for parallelism depending on the dataset size and number of cores. The following guidelines can be considered:

Each partition should be less than **200MB**for better performance. Ideal size is **(100-200MB)**

- For limited cluster size and small DataFrame:set number of partitions to 1x or 2x number of cores (each partition should be less than 200MB):

    - e.g. input size: 2 GB with 20 cores, set shuffle partitions to 20 or 40
- For limited cluster size but large DataFrame: set the number of shuffle partitions to Input Data Size / Partition Size (<= 200mb per partition)
    - e.g. input size: 20 GB with 40 cores, set shuffle partitions to 120 or 160 (3x to 4x of the cores & makes each partition less than 200 mb)
- For large cluster size with more cores than the needed number of partitions, set to 1x or 2x number of cores
    - e.g. input size: 80 GB (which requires less than 400 partitions) with 400 cores, set shuffle partitions to 400 or 800

Formula

spark.sql.shuffle.partitions= ((shuffle stage **input size**/**target size**)/**total cores**) * **total cores**

For instance, when input file is as follows:

Duration	Tasks: Succeeded/Total	Input
17 min	1797/1797	175.3 GiB

Assuming we have**500 executors**and**2 cores**each. The total number of cores = 500 * 2 = **1000 cores**

```
--num-executors 500 \
--conf spark.executor.memoryOverhead=8G \
--conf spark.executor.cores=2 \
```

The appropriate number of partitions could be = ((175.3 * 1024 / 200) / 500) * 500 approx**900 partitions**

In this case, we should set the number of shuffle partitions to 1000. This is because the number should not be lower than the number of cores in order to leave no core idle during shuffling.

Repartitioning or Coalesce

Expensive shuffling can be reduced by using**coalesce()** when we are confident that the number of partitions are not reduced too drastically due to the following reasons:

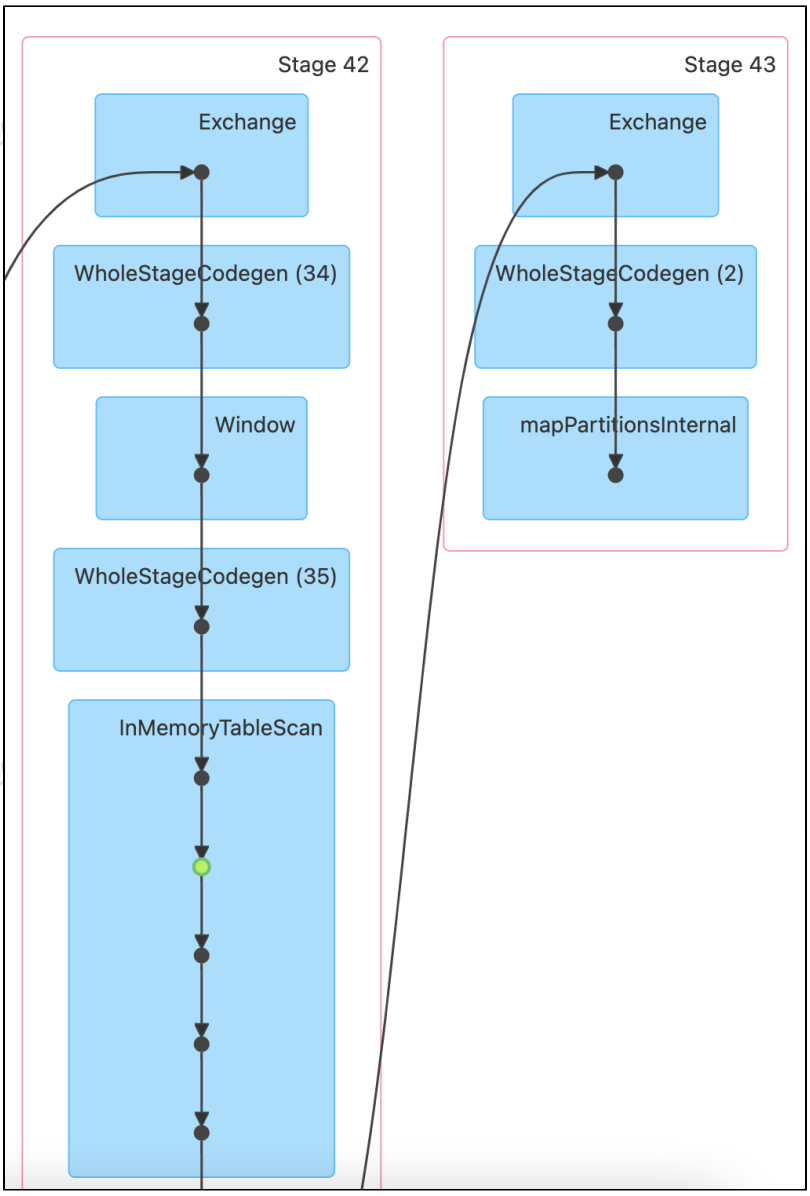
- Coalesce does not evenly distribute the data. So some partitions may get too large
- Drastic coalesce may reduce parallelism as discussed herehttps://blog.devgenius.io/a-neglected-fact-about-apache-spark-performance-comparison-of-coalesce-1-and-repartition-1-80bb4e30aae4

Thus,**coalesce()** can still be left as an option but will require more consideration from the user.

Besides, setting **maxRecordsPerFile** configuration for the DataFrameWriter is another potential method to avoid expensive repartitioning while also preventing an overly large partition. When some of the current partitions may have too many records that we have to**repartition()** to increase the number of existing partitions, we can avoid that by setting**maxRecordsPerFile**and just directly write out current partitions. If a partition is too large, it will be written out into multiple files.

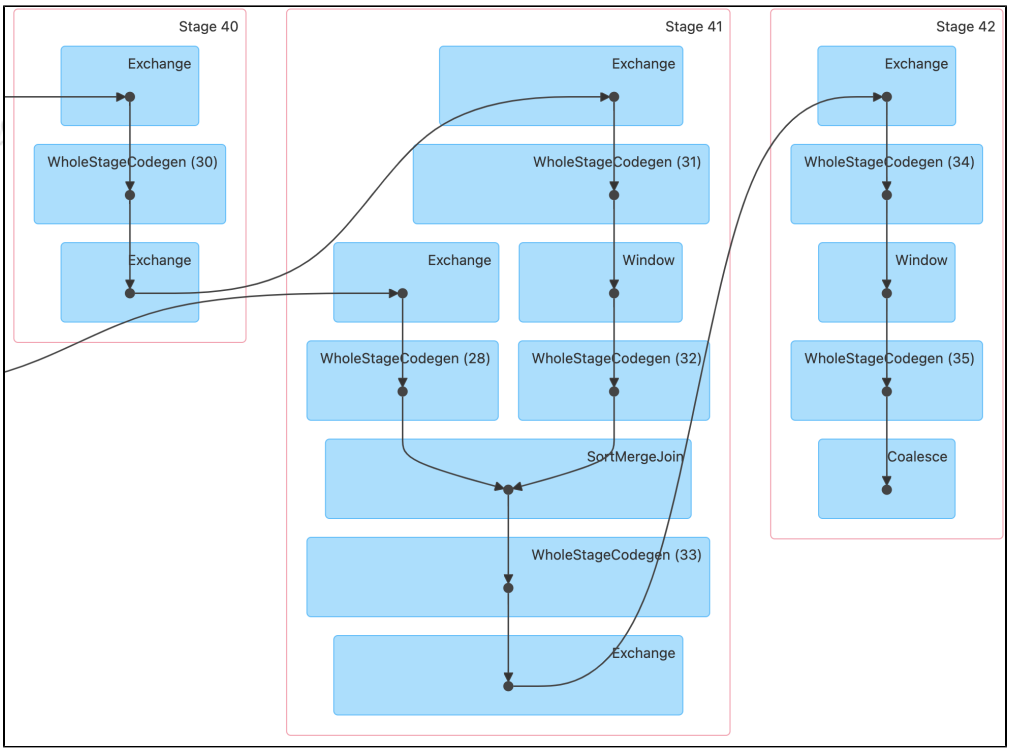
Let's compare 2 different methods of writing out the DataFrame:

1. Original method: **Caching** + **Counting** + **Repartition**



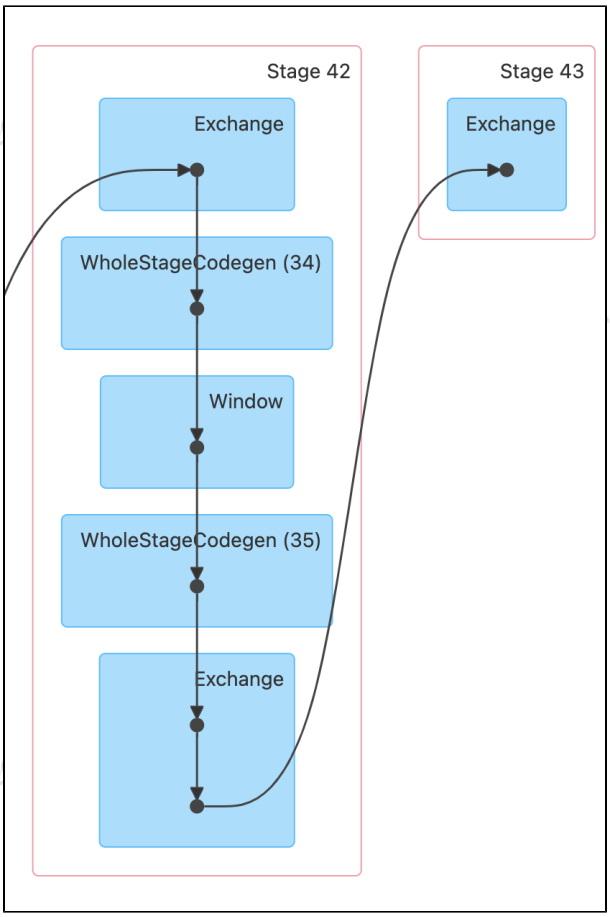
- **Caching** is denoted by the **InMemoryTableScan** operator in Stage 42. Afterwards, **counting** will be executed from the same stage as data has already been cached. After that, **repartition**triggers another shuffle to distribute the data to the desired number of partitions according to our calculation

2. Method 2: Using **Coalesce +without caching**instead of repartition



- Using this method, there won't be **Stage 43**as we no longer need another shuffle. Instead, we just coalesce at**Stage 42** after the previous shuffle with stage 41. However,**Coalesce** will try to collapse the parent RDD from stage 42 and thus, the previous shuffle may be hampered because the number of partitions will be reduced from stage 41 to 42. If the reduction in number of partitions is too drastic, doing**Coalesce** may slow down more than**Repartition**

3. Method 3: Using **Repartition +without Caching**



- There will be no table scan due to caching. Instead, counting is performed in stage 42 and then **repartitioning** is performed between stage 42 and stage 43. Even though there is an additional shuffling step. The number of partitions is stage 42 is preserved for exchange with stage 41 and thus the parallelism is not hampered. Also, data may be more evenly distributed at stage 43.

Spark 3 Configurations Tuning

Data Serialisation

https://spark.apache.org/docs/latest/tuning.html

By default, Kryo serialisation is not enabled by Spark, but is advised on the main documentation. It works better than Java default serialisation by serialising faster and consume less number of bytes so it is good for network intensive applications. Even though Kryo works for Java objects, but since we are working with PySpark API which is built on top of Java API, it should still have an impact on PySpark applications. To enable Kryo serialisation, we can set it up as Spark configuration when intialising the Spark context.

```
spark = SparkSession.builder \
    .config('spark.sql.parquet.compression.codec', 'snappy') \
    .config('spark.serializer', 'org.apache.spark.serializer.KryoSerializer') \
    .config('spark.kryo.registrationRequired', 'false') \
    .enableHiveSupport() \
    .getOrCreate()
```

Executor Memory

https://spark.apache.org/docs/latest/configuration.html

Spark memory structure/hierarchy is explained in more details at the Spark Memory Management section below. Based on that explanation, some parameters which we can tune are:

- `spark.memory.fraction` **(default=0.6)** which decides the fraction of memory allocated for **execution** and **storage** out of the familiar `spark.executor.memory` configuration. As **0.4** of `spark.executor.memory` is reserved for **user memory**, we can consider whether to lower this fraction if we do not need too much for this memory.
- `spark.executor.memoryOverhead` **(default=0.1 * spark.executor.memory):** accounts for things like VM overheads, interned strings, other native overheads, etc. Additional memory includes PySpark executor memory and memory used by other non-executor processes running in the same container. It should be set at **20%** in the case of heavy data to prevent OOM error

Adaptive Query Execution

Blog post: https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html

Notebook: https://docs.databricks.com/_static/notebooks/aqe-demo.html?_ga=2.92287386.665664063.1641001647-2021769539.1638122718

Configuration: https://spark.apache.org/docs/latest/sql-performance-tuning.html#adaptive-query-execution

AQE has an important aspect which makes it more effective: it can adapt the query plan according to runtime statistics instead of static analysis. At every boundaries between stages or after the Map operation, it receives statistics such as the partition sizes and thus could adapt the query plan more accurately based on real statistics rather than estimation.

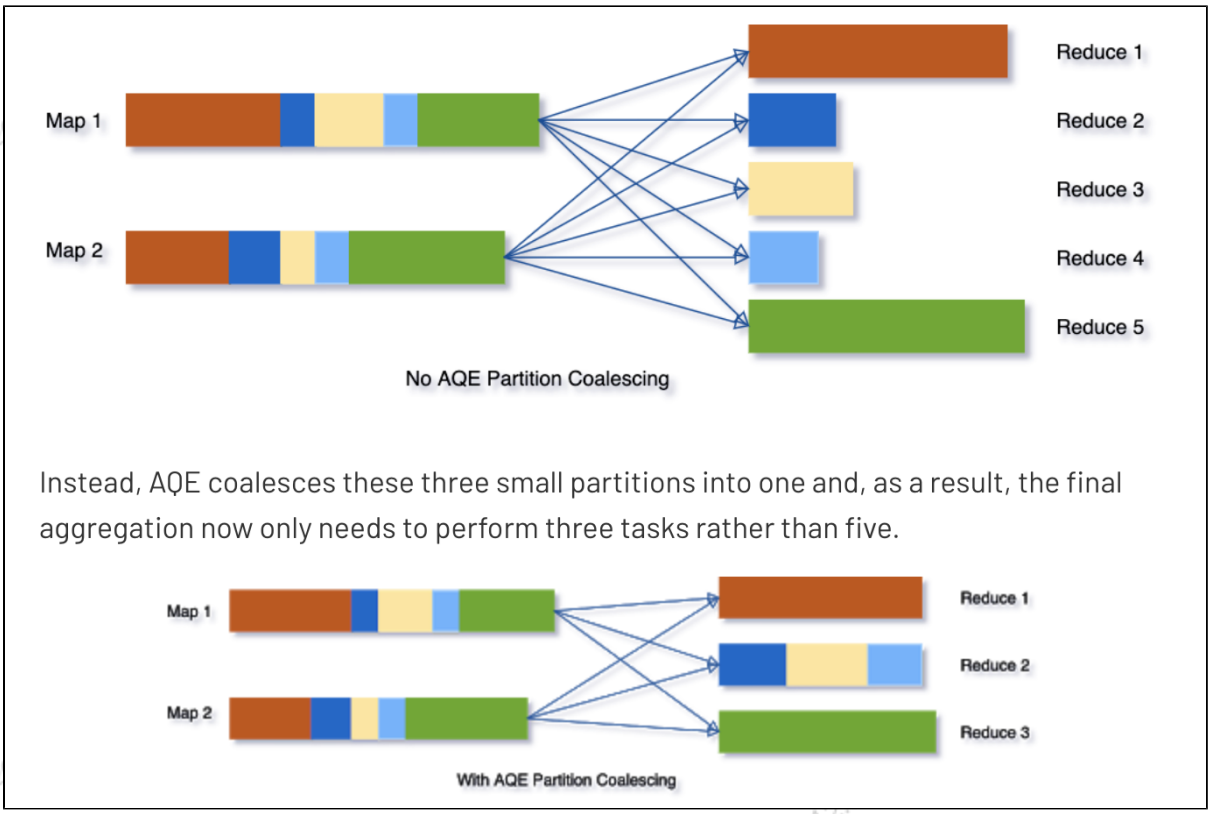
To enable AQE, set `spark.adaptive.enabled=true`. Another important configuration is `spark.sql.adaptive.advisoryPartitionSizeInBytes (default 64MB)` which determines the advisory partition size when coalescing shuffle partitions or split partitions during skewed join. This parameter should be set to **128MB or 256MB** which is also the default block size in Spark.

There are **4 main dynamic improvements** that AQE can help:

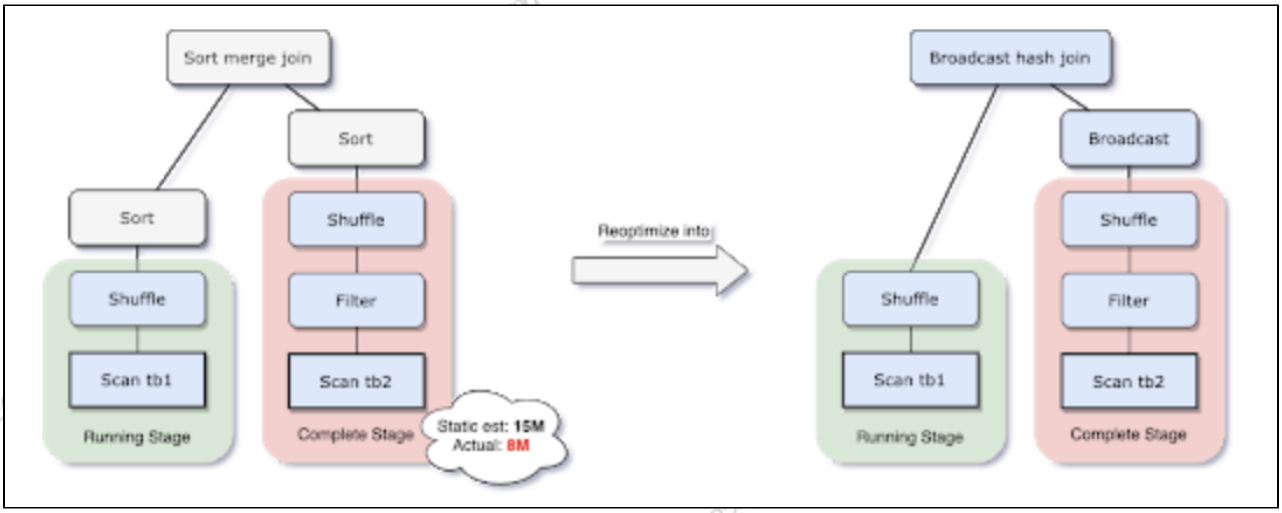
One key property of shuffle is the number of partitions. The best number of partitions is data dependent, yet data sizes may differ vastly from stage to stage, query to query, making this number hard to tune:

1. If there are too few partitions, then the data size of each partition may be very large, and the tasks to process these large partitions may need to spill data to disk (e.g., when sort or aggregate is involved) and, as a result, slow down the query.
2. If there are too many partitions, then the data size of each partition may be very small, and there will be a lot of small network data fetches to read the shuffle blocks, which can also slow down the query because of the inefficient I/O pattern. Having a large number of tasks also puts more burden on the Spark task scheduler.

- **Dynamically coalesce shuffle partitions**: helpful to reduce the number of small partitions by merging them together so we can worry less about setting the right number of partitions, specially setting **too many**. So, we can start with setting a relatively high number of shuffle partitions. Less partitions means **less tasks** to perform.
  - `spark.sql.adaptive.coalescePartitions.enabled=true` to enable it
  - `spark.sql.adaptive.coalescePartitions.minPartitionSize (default 1MB)` is the parameter for minimum partition size after coalescing multiple small partitions.



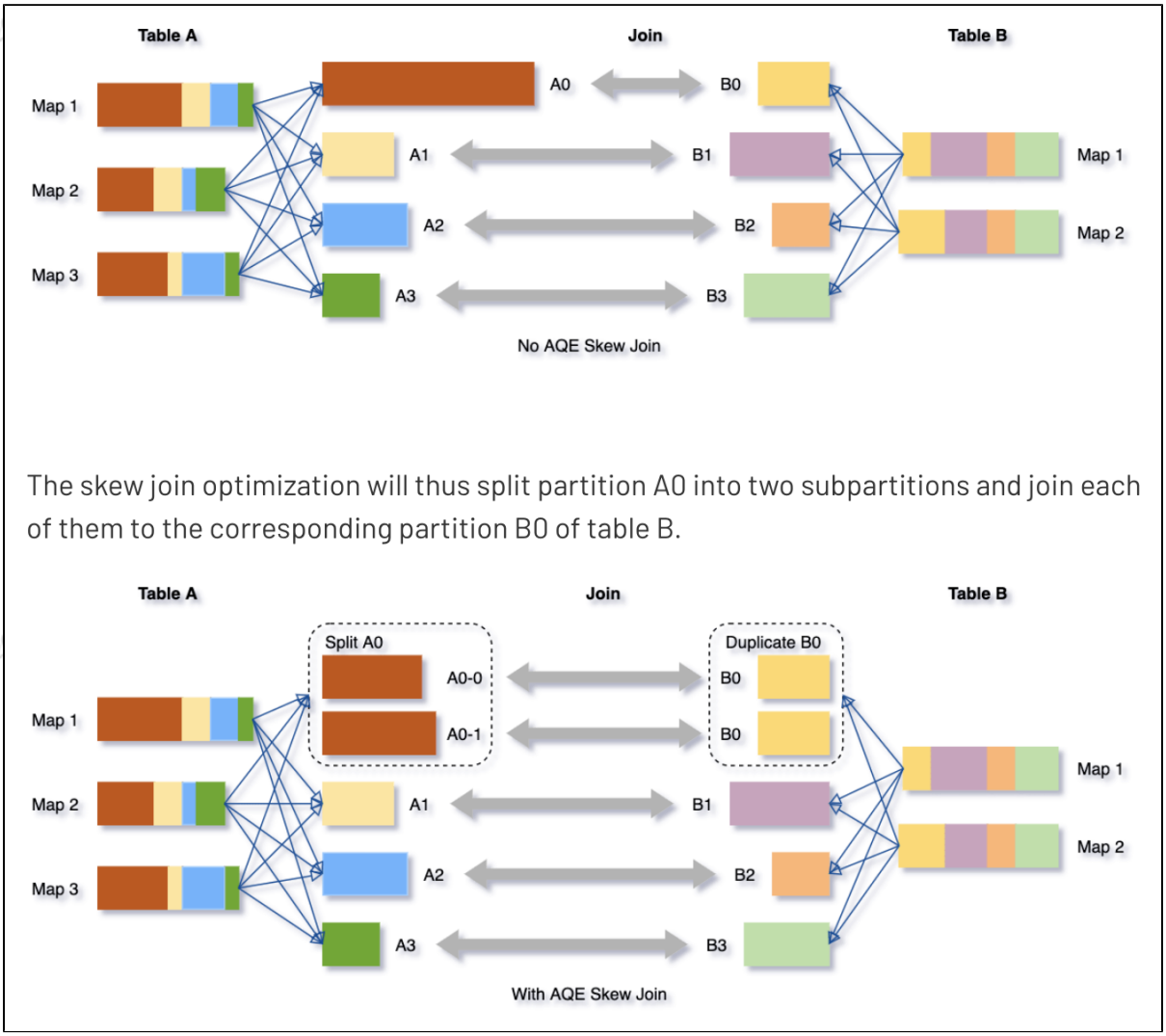
- **DynamicallySwitch from from Sort Merge Join to Broadcast Hash Join**: since Broadcast Hash Join is the most performant if one relation completely fits in memory. AQE improves the determination of this smaller relation than static estimation as due to its accurate runtime statistics.
  - `spark.sql.adaptive.autoBroadcastJoinThreshold` which is also`spark.sql.autoBroadcastJoinThreshold (default 10MB)`



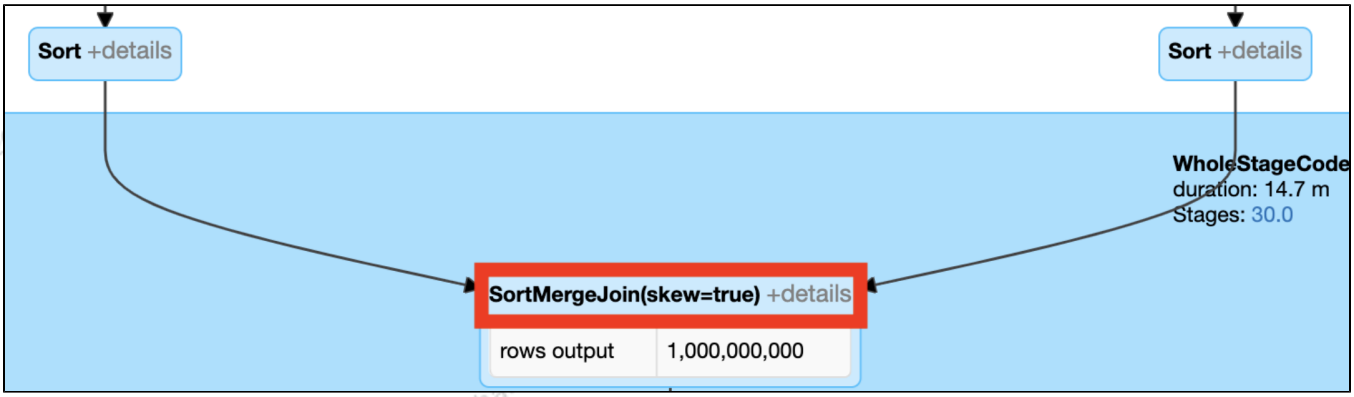
- **Dynamically convert Sort Merge Join to Shuffled Hash Join:**if the sorting step is too expensive or one of the relation is smaller and when partitioned for shuffling, their partitions can fit in memory then using Shuffle Hash Join can help to avoid the sorting step
  - `spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold`sets themaximum size in bytes per partition that can be allowed to build local hash map. If all partitions are below this threshold, then Shuffle Hash Join will be used instead

- **Dynamically Optimise Skew Join:**detect skewed partitions from shuffle file statistics. It then split the skewed partition into smaller sub-partitions to be joined with the partition from the other size
  - `spark.sql.adaptive.skewJoin.enabled=true`

- `spark.sql.adaptive.skewJoin.skewedPartitionFactor (default=5)`:A partition is considered as skewed if its size is larger than this factor multiplying the **median partition size** and also larger than`skewedPartitionThresholdInBytes`
- `spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes (default=256MB)`: should be set larger than`spark.sql.adaptive.advisoryPartitionSizeInBytes`



- In the above example, initially there are four tasks running but one partition is much larger and is the bottleneck. After skewed join optimisation, there are five tasks but each is even with each other which leads to an overall better performance



- Above is how a Sort Merge Join with skewness detected would look like in the Spark UI's SQL tab

Concurrent Fetch Request

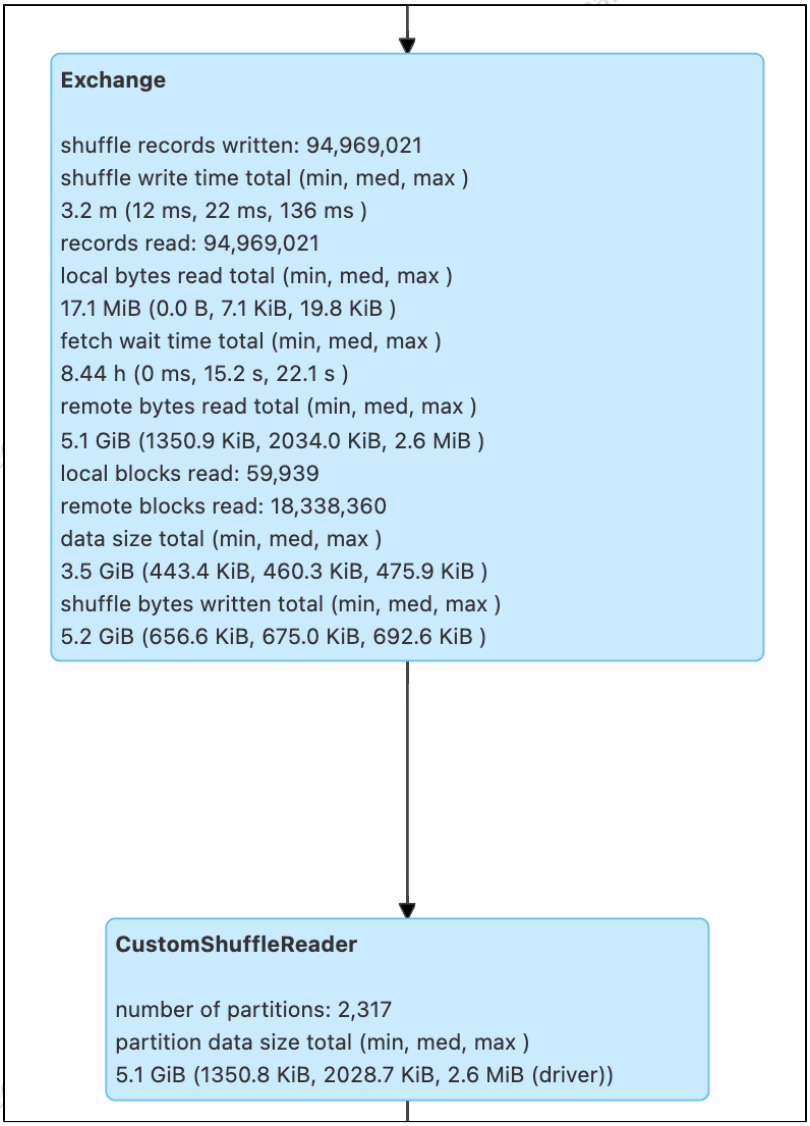
We can also limitthe number of remote requests to fetch blocks at any given point. When the number of hosts in the cluster increase, it might lead to very large number of inbound connections to one or more nodes, causing the workers to fail under load.

- `spark.reducer.maxReqsInFlight (default=Int.MAX_VALUE)`so we can set it to**1**if we want to maximise network bandwidth

Application to Coin Movement Model

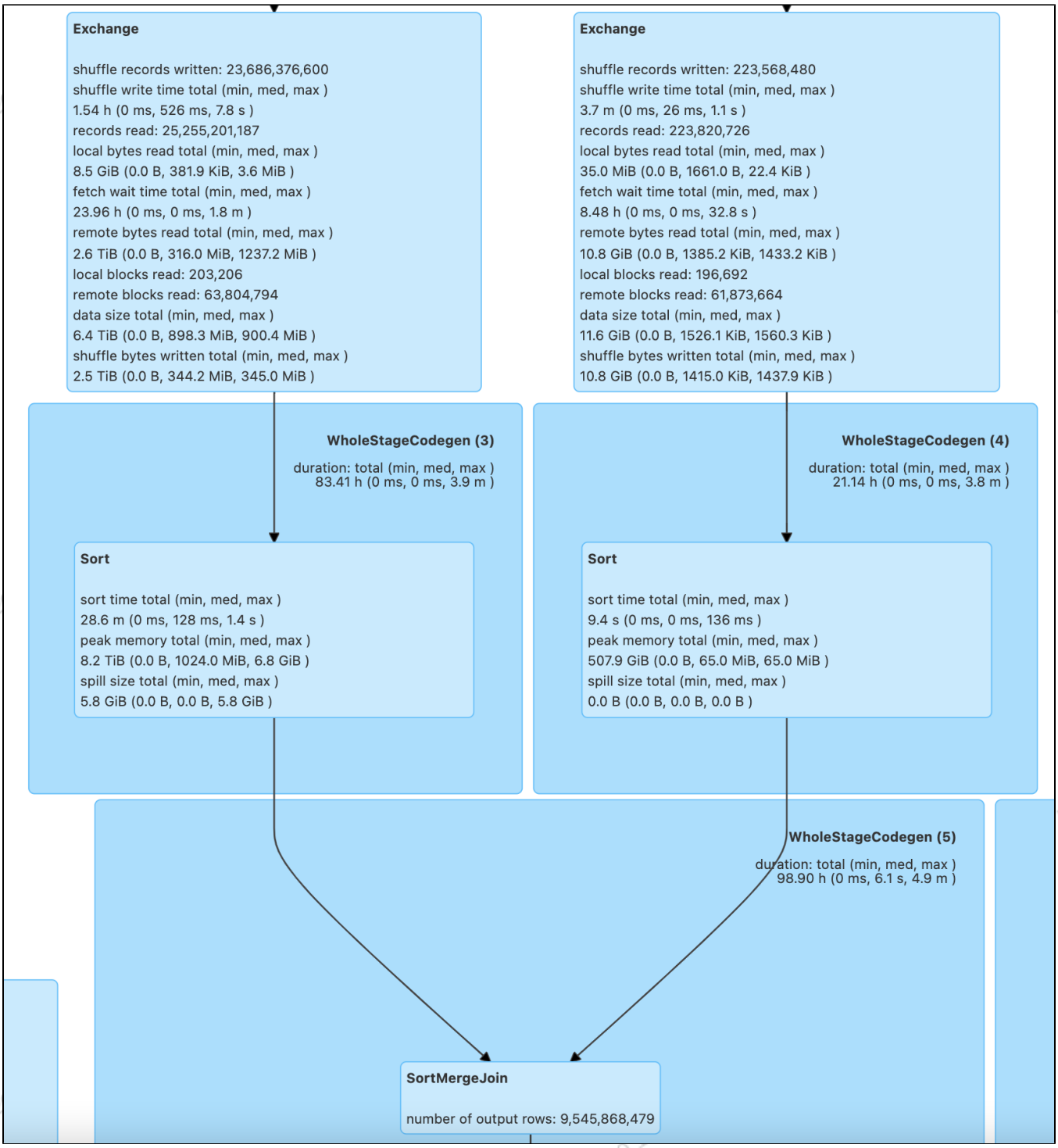
A heavy model under analysis is**coin movement**model. The following analyses resulted from studying this model.

Dynamic Shuffle Partitions Coalesce in action



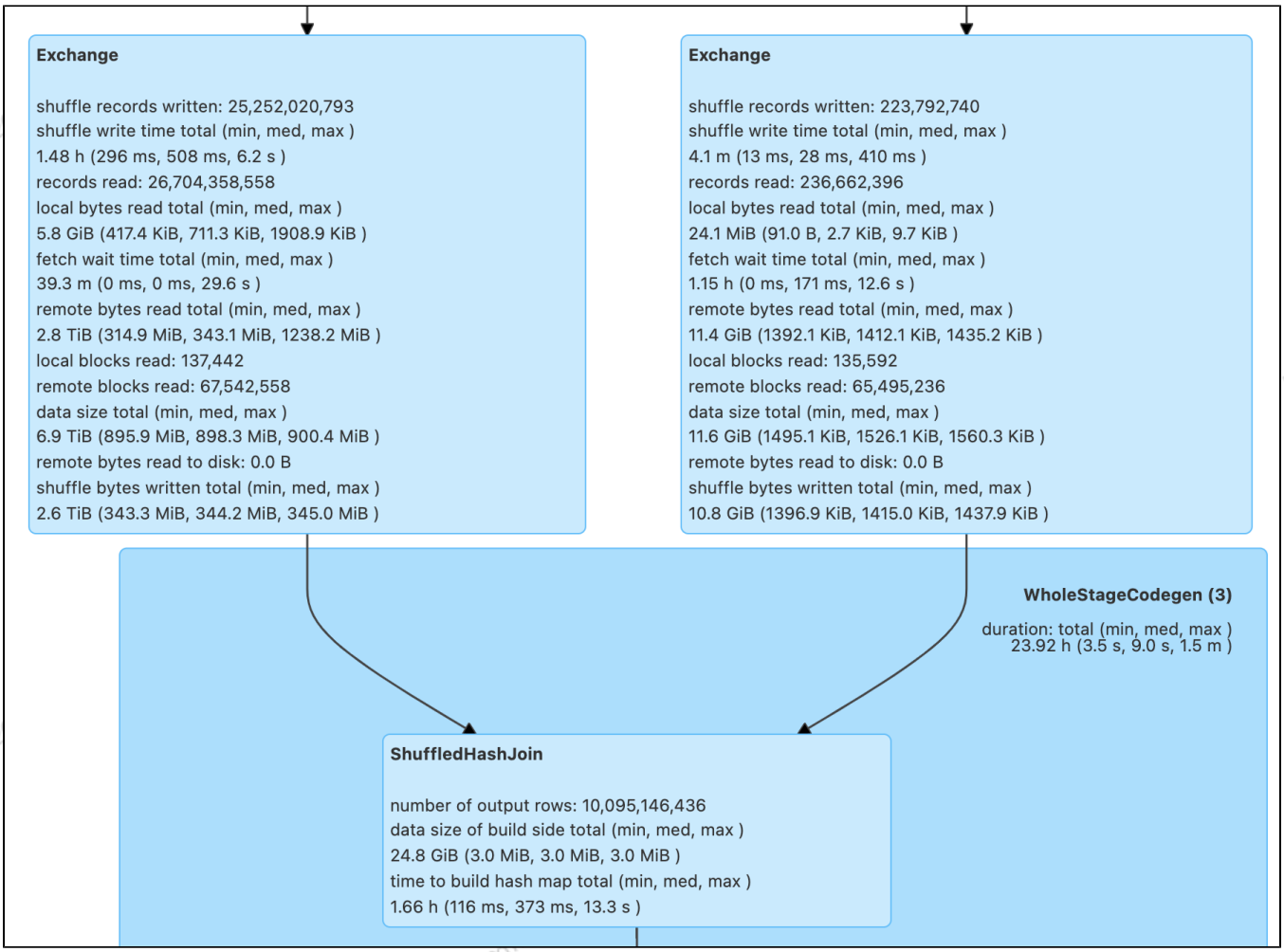
- Here, the data to be shuffled is **5.2 GiB**as shown by **5.2 GiB shuffle bytes being written to total**. The number of shuffle partitions is coalesced down from **8000** as set by `spark.sql.shuffle.partitions`to **2317.** This shows that AQE is making use of runtime statistics from the Map stage to optimise the plan

Convert Sort Merge Join to Shuffle Hash Join in action



- Even though sort merge join is generally good for joining two large relations as they can be spilled to disk if memory is insufficient hence preventing Out of Memory failure, the cost of the sorting step can be expensive. In this case, inspecting the Spark UI shows that sorting incurs disk spill for the **first relation** where **Spill Size Total** is **5.8 GiB**

- Also, it is observed that one of the joining relation is much smaller, so after shuffling, if we are confirmed that one of the relation's partitions are always below a certain threshold to fit completely in memory, we can choose **Shuffled Hash Join** which will build a hash table of the smaller relation's partition in memory to join with the external larger relation instead.
- By setting`spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold` to a value higher than



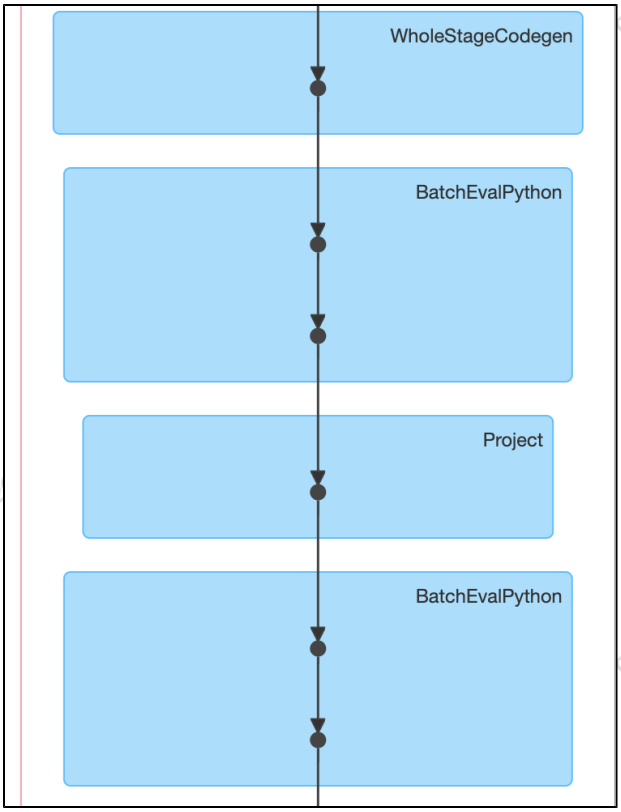
- As shown above, ShuffledHashJoin has been used instead, so the sorting steps are removed. Also, there seems to be no disk spill due to the sorting step any more.

Python UDFs Impact

```python
dp_payment_status_to_string_udf = F.udf(dp_payment_status_to_string, T.StringType())


def dp_fulfillment_status_to_string(value):
    """
    Convert dp fulfillment status encoding to string
    :param status:
    :return:
    """
    if value is None or value == '':
        return ''
    if value == const.DP.Status.READY_F1:
        return 'Ready_F1'
    elif value == const.DP.Status.NOT_READY_F2:
        return 'Not_Ready_F2'
    elif value == const.DP.Status.EXPIRED_F3:
        return 'Expired_F3'
    elif value == const.DP.Status.INITIATED_F4:
        return 'Initiated_F4'
```

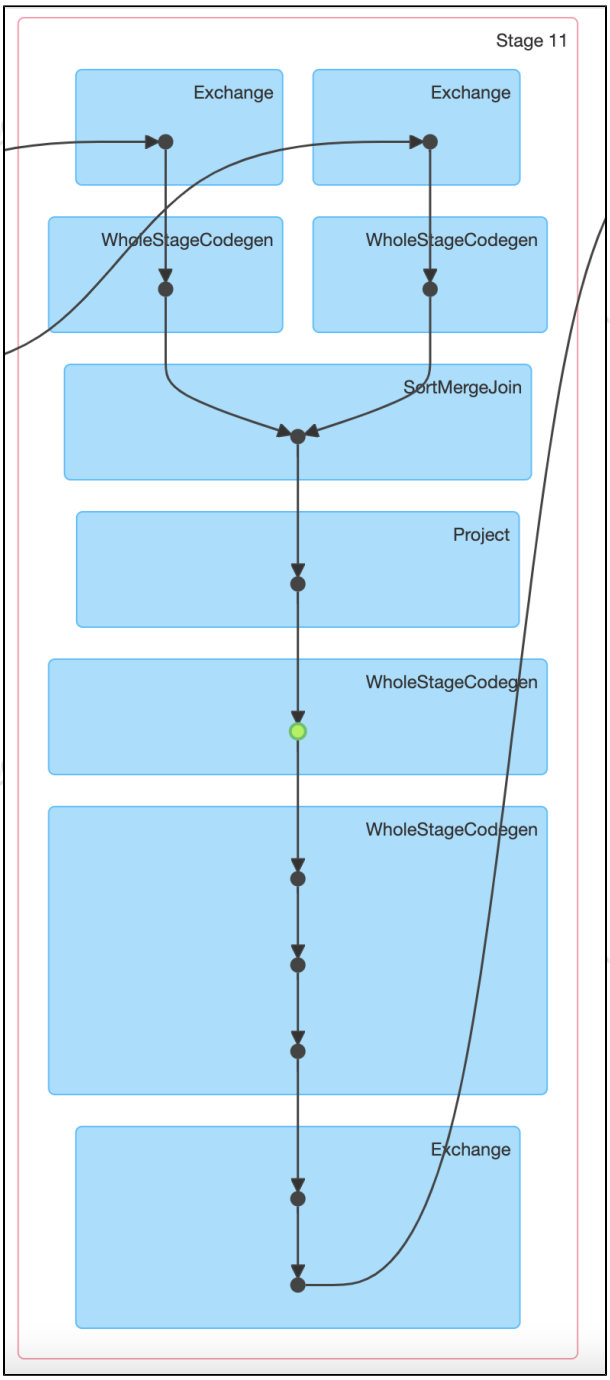
- Unlike Java/Scala UDFs, Python UDFs need to run separately in Python processes and cannot be a part of the Java Virtual Machine. As such, these functions do not inherit the optimisation underlying the SQL engine or Code generation. Also, data from the RDD needs to be moved into the Python processes to be able to run as indicated by the BatchEvalPython executor (moving data into Python process by batch) and this can be very slow because moving data is expensive (the reason why Spark try to move tasks to the data instead of moving data to the tasks).



- As such, when Python UDFs are used repeatedly, the above query plan shows many BatchEvalPython operator. Also, after these operators, Spark will lose shuffle statistics as well hence affect its query optimisation.

If possible, we should convert simple UDFs to using Spark SQL's built-in functions such as**when** and **otherwise**in this case

```
def dp_refund_status_to_string_spark(column):
    """
    Convert dp refund status encoding to string
    :param status:
    :return:
    """
    return F.when(column.isNull(), '') \
            .when(column == '', '') \
            .when(column == const.DP.Status.REFUND_REQUEST_CREATED_RR1, 'Refund_Request_Created_RR1') \
            .when(column == const.DP.Status.REFUND_REQUEST_REJECTED_RR2, 'Refund_Request_Rejected_RR2') \
            .when(column == const.DP.Status.REFUND_REQUEST_ACCEPTED_RR3, 'Refund_Request_Accepted_RR3') \
            .when(column == const.DP.Status.REFUND_NOT_YET_R0, 'Refund_Not_Yet_R0') \
            .when(column == const.DP.Status.REFUND_CREATED_R1, 'Refund_Created_R1') \
```



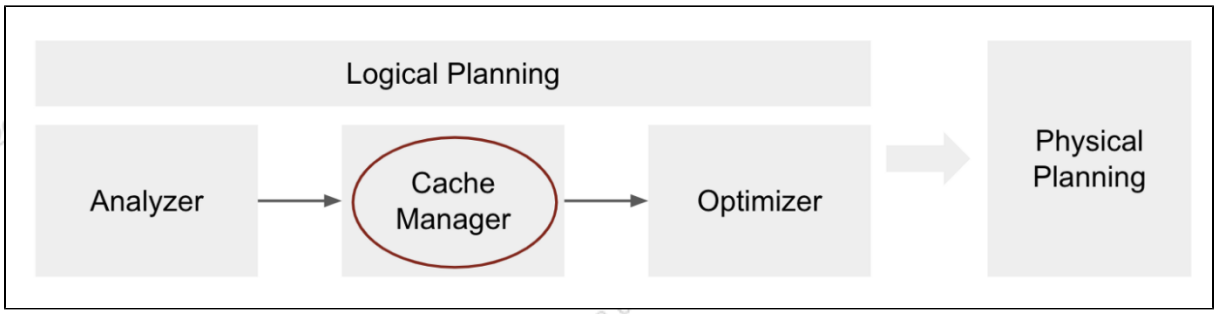
- When Python UDFs are removed, the entire stage is shown here with no **BatchEvalPython** operator

Other Relevant Research

Caching

https://towardsdatascience.com/best-practices-for-caching-in-spark-sql-b22fb0f02d34

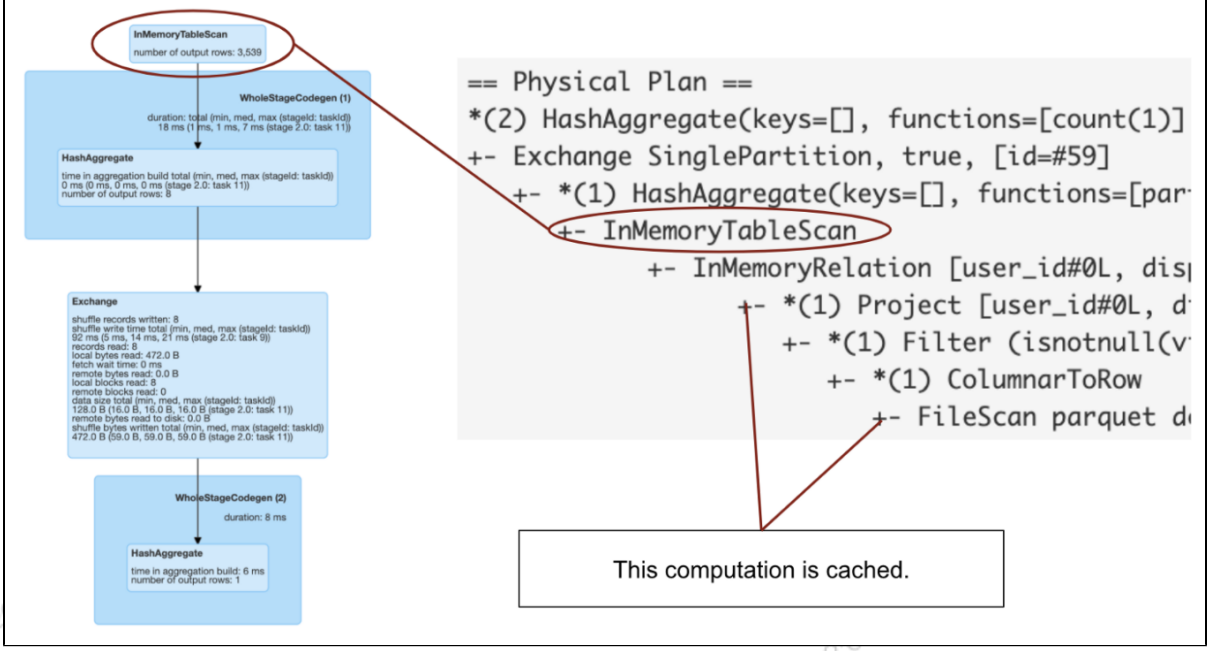
A **Cache Manager**keeps track of what computation has been cached in terms of the query plan. When the cache function is called, the Cache Manager pulls out the analysed logical query plan and store it in an indexed sequence called**cachedData**.



- The phase of Cache Manager is part of Logical Planning, after Analyzer and before Optimizer
- When we run a query with an action, the query plan is processed and transformed and at Cache Manager, Spark checks for each subtree of the analyzed plan if it has been stored in **cachedData** sequence.

```
df = spark.table("users").filter(col(col_name) > x).cache()

df.count() # now check the query plan in Spark UI
```



- If there is a match, it means the same plan/computation has already been cached and Spark can use that. Spark then adds that information to the query plan using **InMemoryRelation** operator
- During Physical Planning, the InMemoryRelation operator will create a corresponding physical operator called **InMemoryTableScan**

Good Practice

1. Call **cached_df = df.cache()** and subsequently **cached_df.action()** to ensure we are making use of the cached transformations
2. Only cache relevant subset of columns to avoid memory's spill to disk

Number of Partitions upon reading Input Data

https://dzone.com/articles/guide-to-partitions-calculation-for-processing-dat

Repartition Methods

Coalesce	Repartition
Returns an RDD that is **reduced** into `numPartitions` partitions	Returns a new RDD that has **exactly** `numPartitions` partitions
Only takes effect when**decrease** the level of parallelism	Can **increase** or **decrease** the level of parallelism. Internally, uses a shuffle to redistribute data
Results in a **narrow** dependency.	Results in a **wide** dependency due to shuffle
When doing a **drastic** coalesce such as to `numPartitions = 1`, may result in computation taking place on too few nodes, hampering parallelism of current upstream partitions. This is because in order to avoid shuffling, coalesce() shrinks its parent RDD to use the same number of partitions as well. As such, if coalesce() follows an expensive join or shuffle, it will limit the degree of parallelism.	Add a shuffle step but current upstream partitions will still be executed in parallel
e.g. From 1000 partitions to 100 partitions, there will be no shuffle. Each of the 100 new partitions will claim 10 of current partitions	The 1000 partitions will have their data distributed **e venly** across using a **hash partitioner**

DataFrameWriter's Option

Spark normally writes a single file out per task, thus the number of saved files is equal to number of RDD partitions. If a partition size is too large, the RDD needs to be repartitioned to control the number of output files. `Repartition` is expensive as data is shuffled across the network, hence an option can be set to limit the max number of records written per file.

From Spark 2.2, there is a flag to set when writing a DataFrame that controls the number of records per file, thus removing the need to perform a costly repartition.

```
df.write.option('maxRecordsPerFile', 1000000).mode('overwrite').parquet(outputDirectory)
```

Spark application breakdown

1. Jobs: Spark **driver** converts Spark application into one or more Spark jobs. Then it transforms each job into a DAG. This DAG is Spark's execution plan
2. Stages: Each job consists of one or more stages. Within each stage are narrow-dependency tasks, between two stages are wide-dependency tasks (requires **shuffle**)
3. Tasks: A unit of execution which are federated across each Spark **executor.**Each task maps to a single core and works on a single partition of data. For e.g., an executor with 16 cores can have 16 or more tasks working on 16 or more partitions in parallel.

SQL Engine

Catalyst Optimizer

Takes a computation query and converts into an execution plan. Goes through 4**transformational phases**. Regardless of language (Python, SQL or Scala), the computation undergoes same process and end up with a similar query plan and identical bytecode for execution

1. Analysis:
    a. Spark SQL engine generates an Abstract Syntax Tree (AST) for the SQL or DF query
    b. Any columns or table names resolved by consulting an internal Catalog
2. Logical Optimization:
    a. Construct a set of multiple logical plans. Each plan is laid out as an operator tree
    b. Use cost-based optimizer (CBO) to assign cost to each plan
3. Physical Planning:
    a. Generate an optimal physical plan for selected logical plan, using physical operators that match those available in Spark execution engine
4. Code generation:
    a. Generate efficient Java bytecode to run on each machine
    b. Spark SQL operates on data sets in memory→ can use state-of-the-art compiler technology for code generation to speed up execution. Thus, it acts as a compiler.

Second-generation Tungsten engine

Facilitate **whole-stage code generation**, which is:

- A physical query optimization phase that collapses the whole query into a single function, getting rid of virtual function calls and employ CPU registers for intermediate data. This helps to generate compact RDD code for final execution which improves CPU efficiency and performance.

Cost-Based Optimization (CBO)

Uses table statistics to determine most efficient query execution plan of a structured query (given logical plan)

Table Statistics

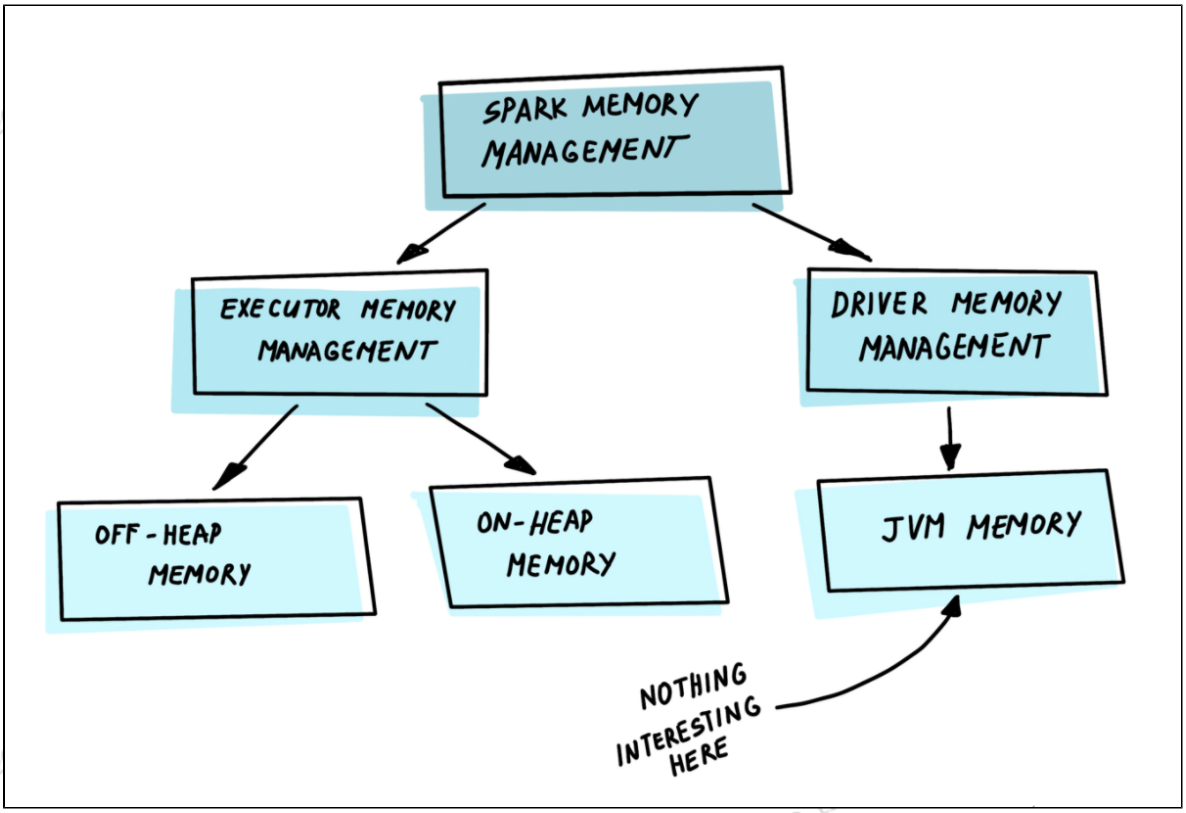
Can be computed for tables, partitions and columns as follows:

1. Total size (in bytes) of a table or table partitions
2. Row count
3. Column statistics: min/max/num_nulls/distinct/avg_col_len/max_col_len/histogram

Spark Memory Management

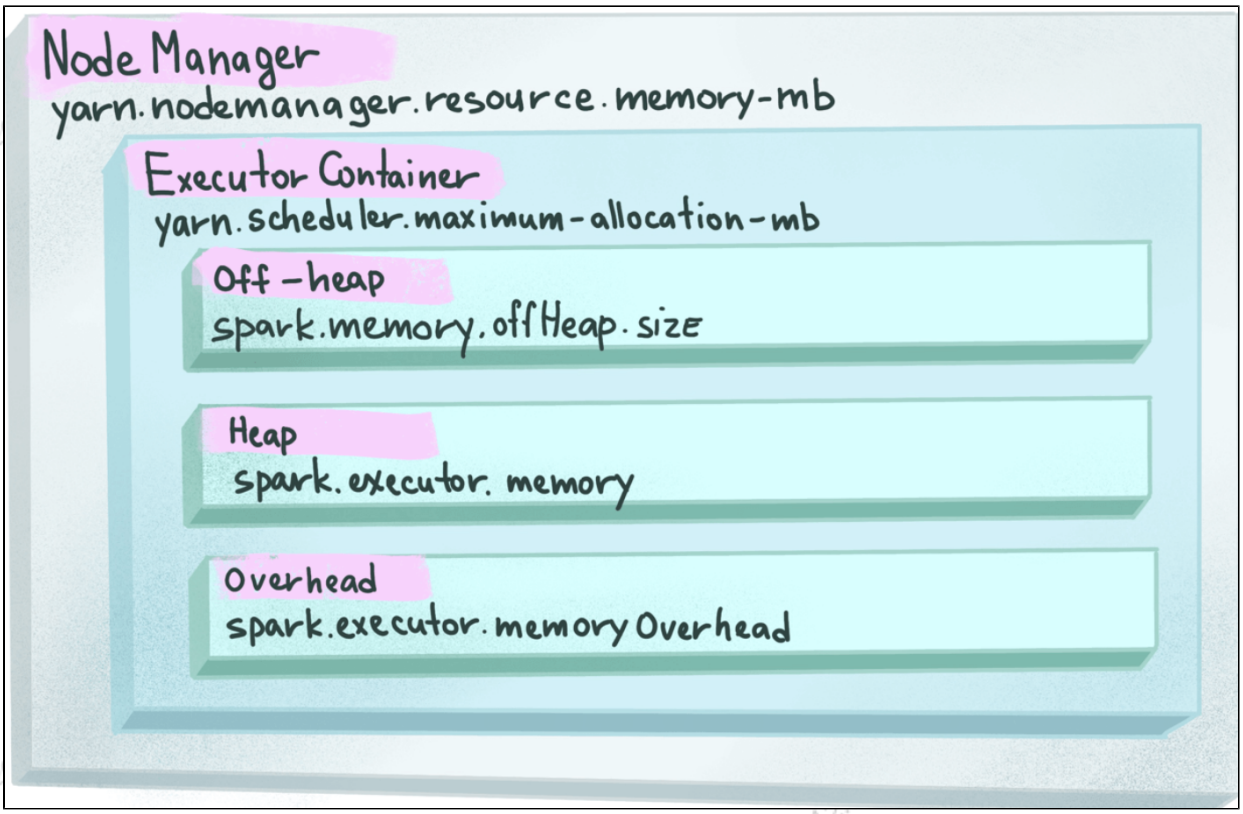
https://luminousmen.com/post/dive-into-spark-memory

High-level Overview



- The executor is the main one responsible for:
  - Performing specific **computational** tasks on worker nodes
  - Return **results** to **driver**
  - **Store RDDs**

Executor container



- When a Spark job is submitted in a cluster with Yarn, Yarn allocates **Executor Containers** to perform the job on different **nodes**
- Each node is assigned resources by Yarn according to **yarn.nodemanager.resource.memory-mb**configuration, thus this is their upper limit

- Resource Manager handles memory requests by Executor Containers and allocates resources to containers by up to**yarn.scheduler.maximum-allocation-mb**configuration.On each node, this resource allocation is done by the Node Manager whose**upper limit of resources = resources of one node=yarn.nodemanager.resource.memory-mb**configurationmentioned above

- One **ExecutorContainer**is just one JVM whose memory is divided into:
  - Heap memory (also **Executor Memory**):specified by**--executor-memory**
  - Off-heap memory: use for off-heap storage for certain operations
  - Overhead memory:a**seatbelt**used for various internal Spark overheads

Executor Memory

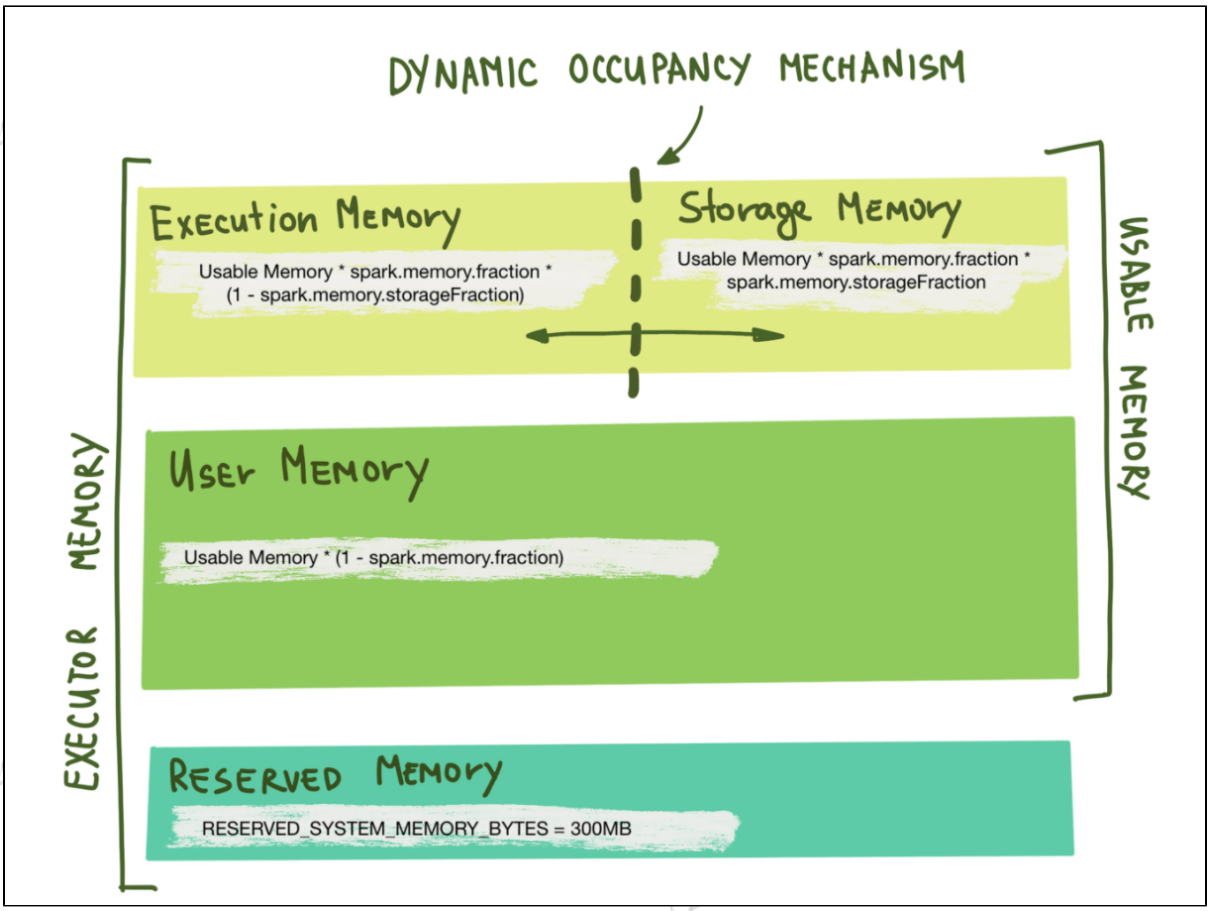
Interface for memory management via**MemoryManager** which implements policies for:

- Dividing available memory **across tasks**
- Allocating memory between **storage** and **execution**

Blocks

Data within Spark applications is physically grouped into blocks. They are transferable objects used as:

- Inputs to Spark tasks
- Returned as outputs
- Intermediate steps in shuffle process
- Store temporary files



Reserved Memory (300MB hardcoded)

- Just to store internal objects
- Guarantees to reserve sufficient memory for the system even for small JVM heaps

Storage Memory

Used for:

- Caching
- Broadcasting data

*Storage Memory = usableMemory * spark.memory.fraction * **spark.memory.storageFraction**= 1 * 0.6 * 0.5 = 0.3by default*

Execution Memory

Mainly used to store temporary data in shuffle, join, sort, aggregation, etc.

*Executor Memory = usableMemory * spark.memory.fraction * (1 - **spark.memory.storageFraction**) = 1 * 0.6 * (1 - 0.5) = 0.3 by default*

User Memory

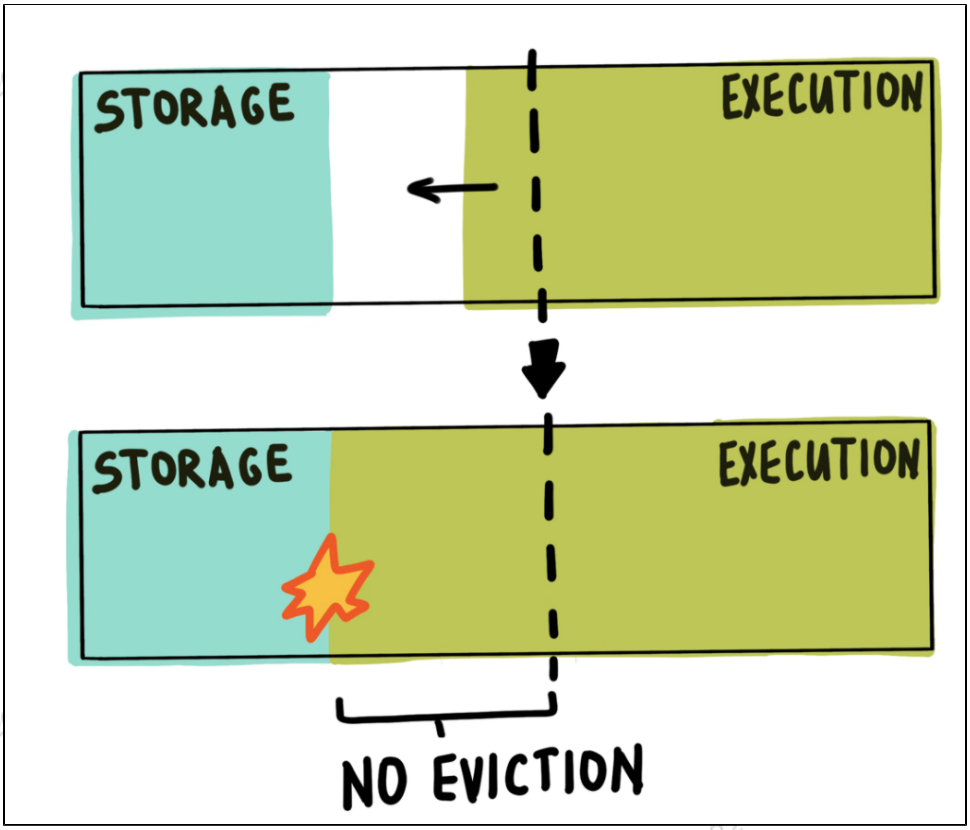
Used to:

- Store data needed for RDD conversion operations such as **lineage**.
- Store user **own data structures** that would be used **inside transformations**

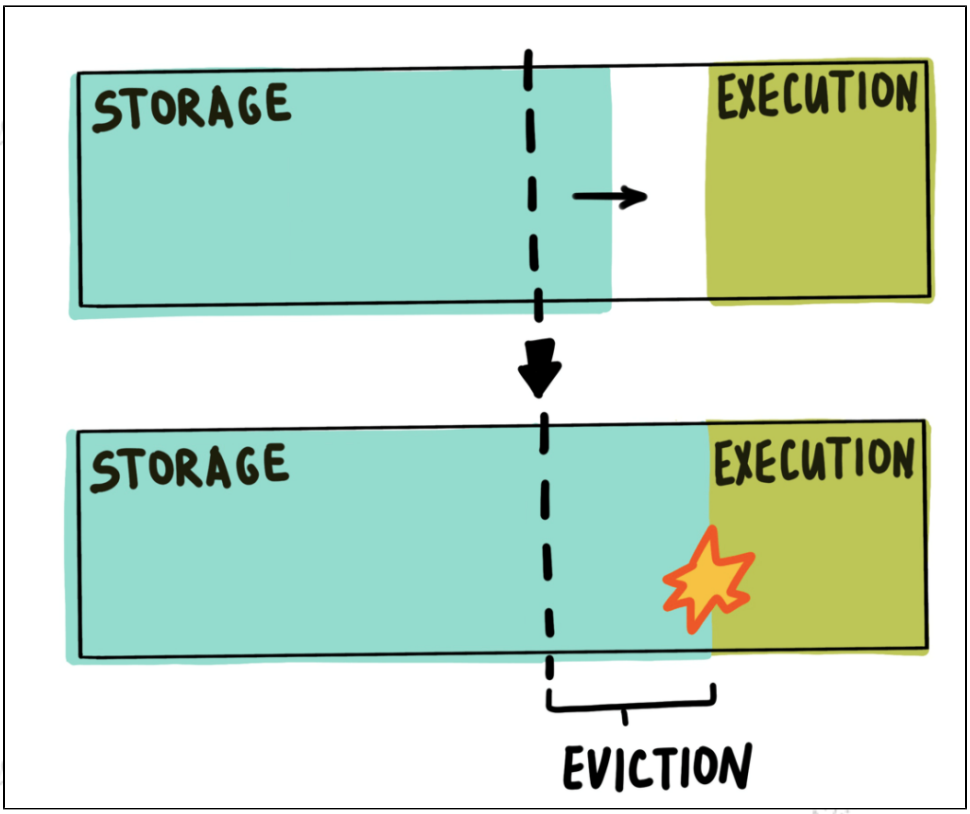
*User Memory = usableMemory * (1 - spark.memory.fraction) = 1 * (1 - 0.6) = 0.4 by default*

Dynamic Occupancy Mechanism

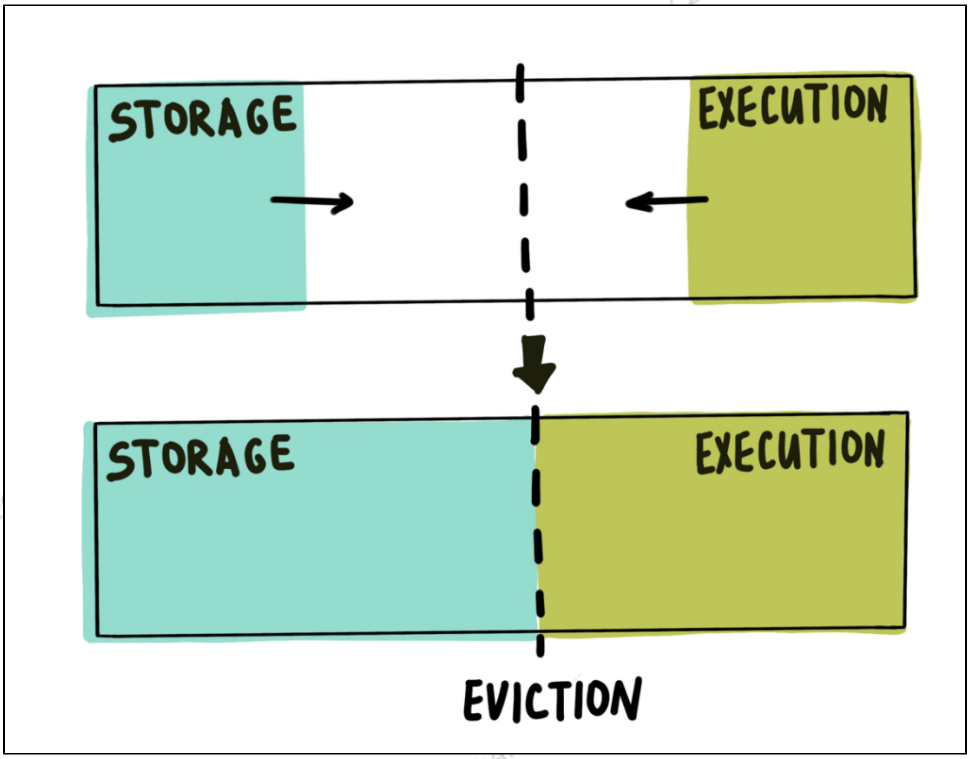
Execution and Storage have a **shared memory** and can**borrow**from each other in this mechanism



- As observed above, the Storage memory is **less prioritised**.It can only be used up to a **certain threshold (dashed line)** and only if not occupied by Execution memory.
- If it is occupied, Storage memory has to wait for the used memory to be released by executor processes



- When Execution memory is not used, Storage memory can borrow as much as available until Execution reclaims the space. When this happens, the cached blocks will be evicted from memory until sufficient borrowed memory is released to satisfy the Execution memory request



- If space from both sides is insufficient and still within appropriate boundaries, it is evicted according to their respective storage levels using LRU algorithm

Overhead Memory

Join Types

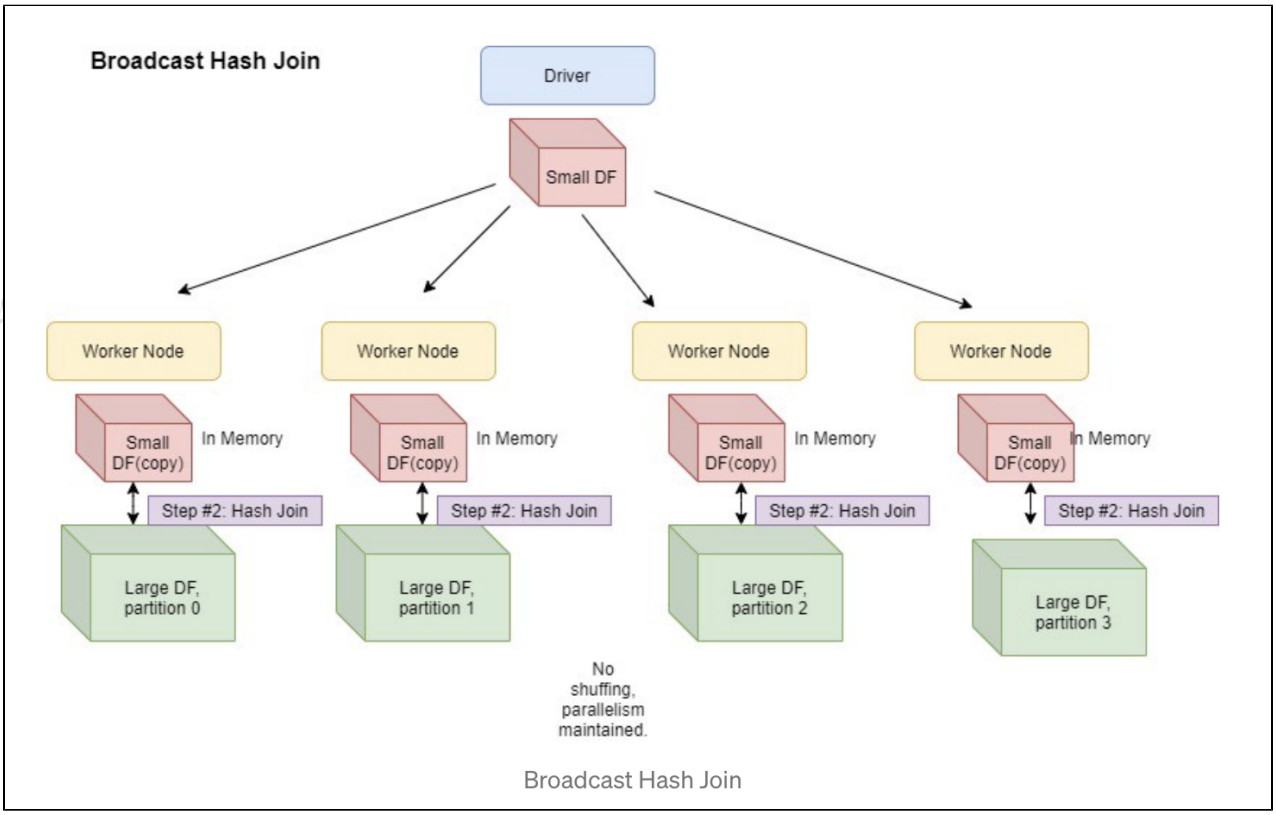
https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf

Broadcast Hash Join (Map-side Join)

A small DF that can fit in both memory of the Driver and each Worker Node will be broadcasted from Driver to all Worker Nodes. This saves shuffling cost. Default broadcast threshold is 10M.

Works by creating HashTable based on the **join_key** the smaller relation in memory then loop through larger relation to match the hashed join key value. Only support:

- '=' join
- inner, left, right except outer join

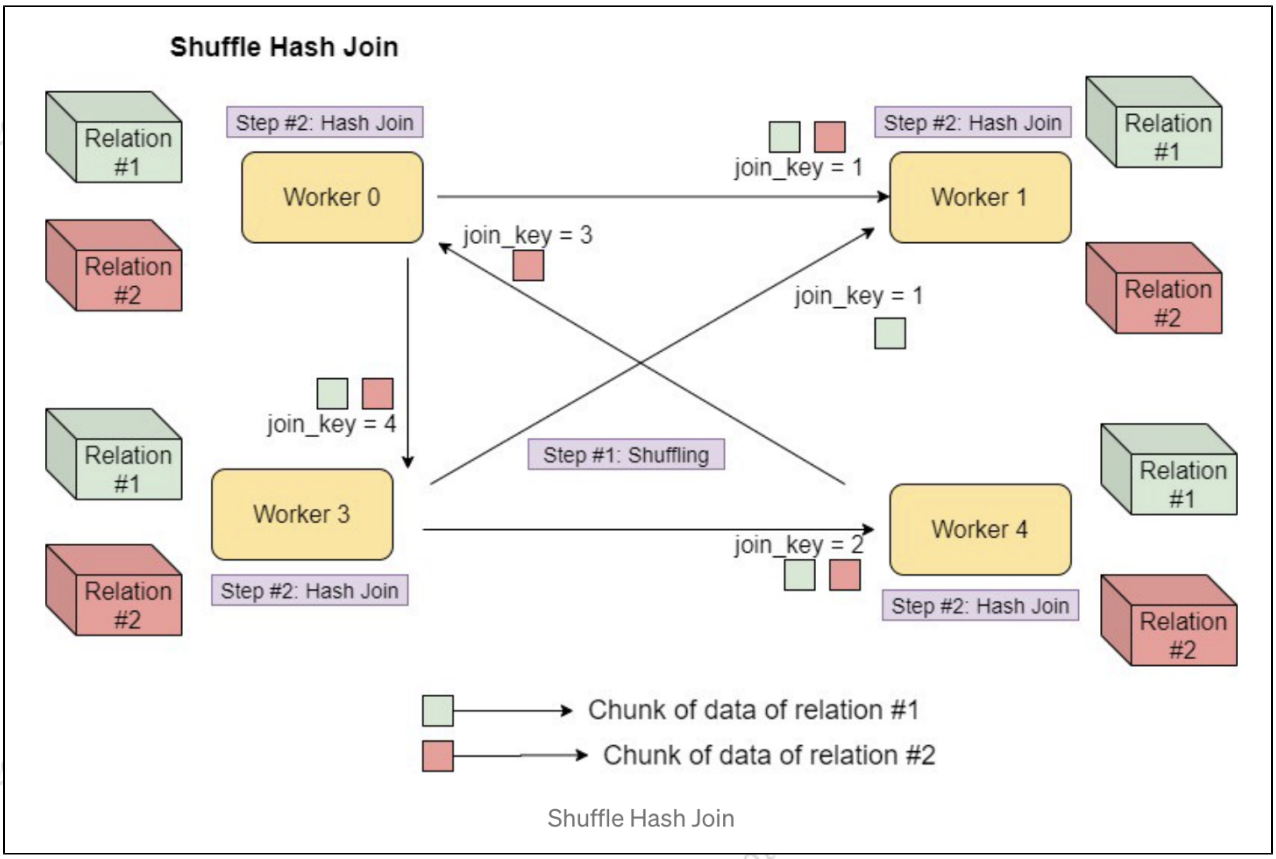


Broadcast Hash Join

Shuffle Hash Join

Move data with same join key to the same executor node followed by Hash Join. Data is thus fully shuffled across Worker Nodes and combined by Hash Join as data of the same key will be on the same executor. Only support:

- '=' join
- inner, left, right except outer join
- Join key don't need to be **sortable**
- Expensive as it involves both shuffling and hashing



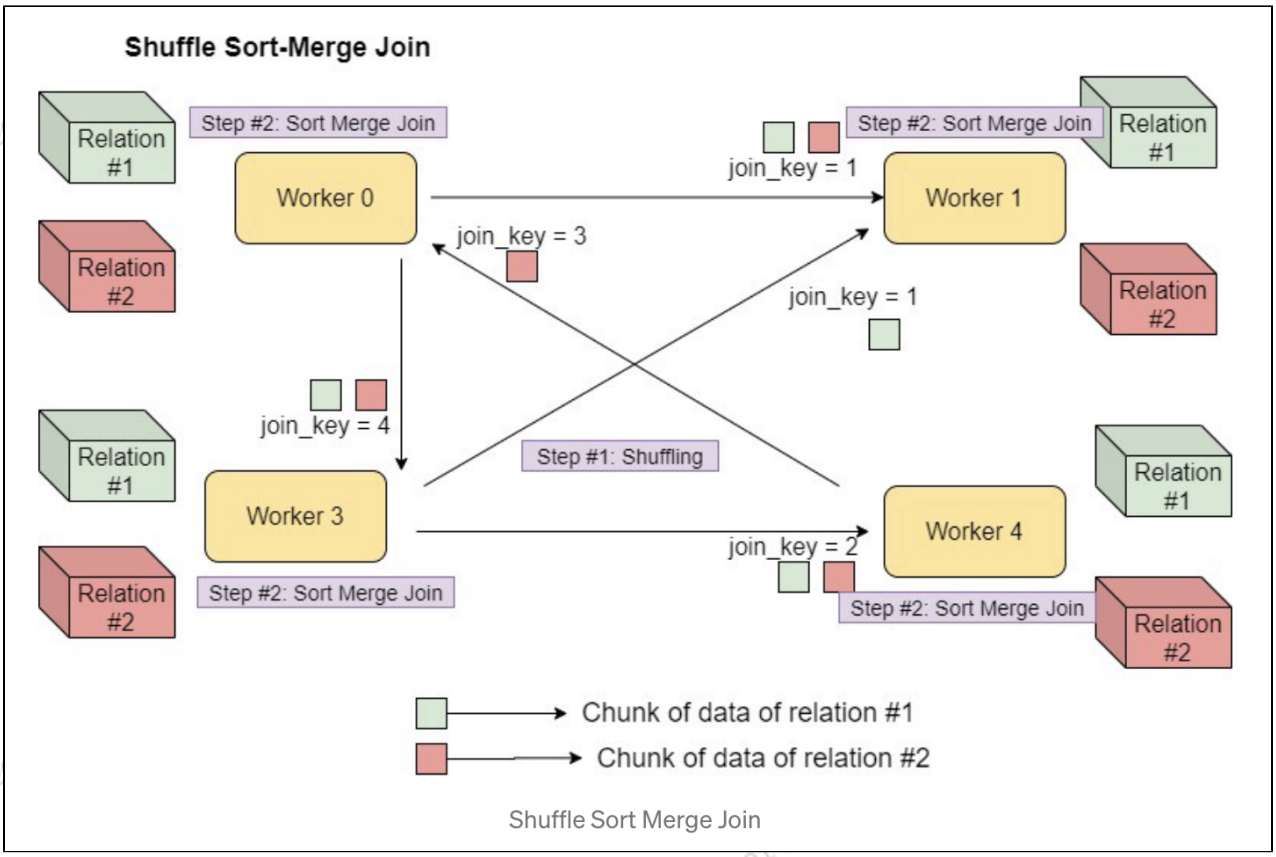
Shuffle Hash Join

Shuffle Sort Merge Join

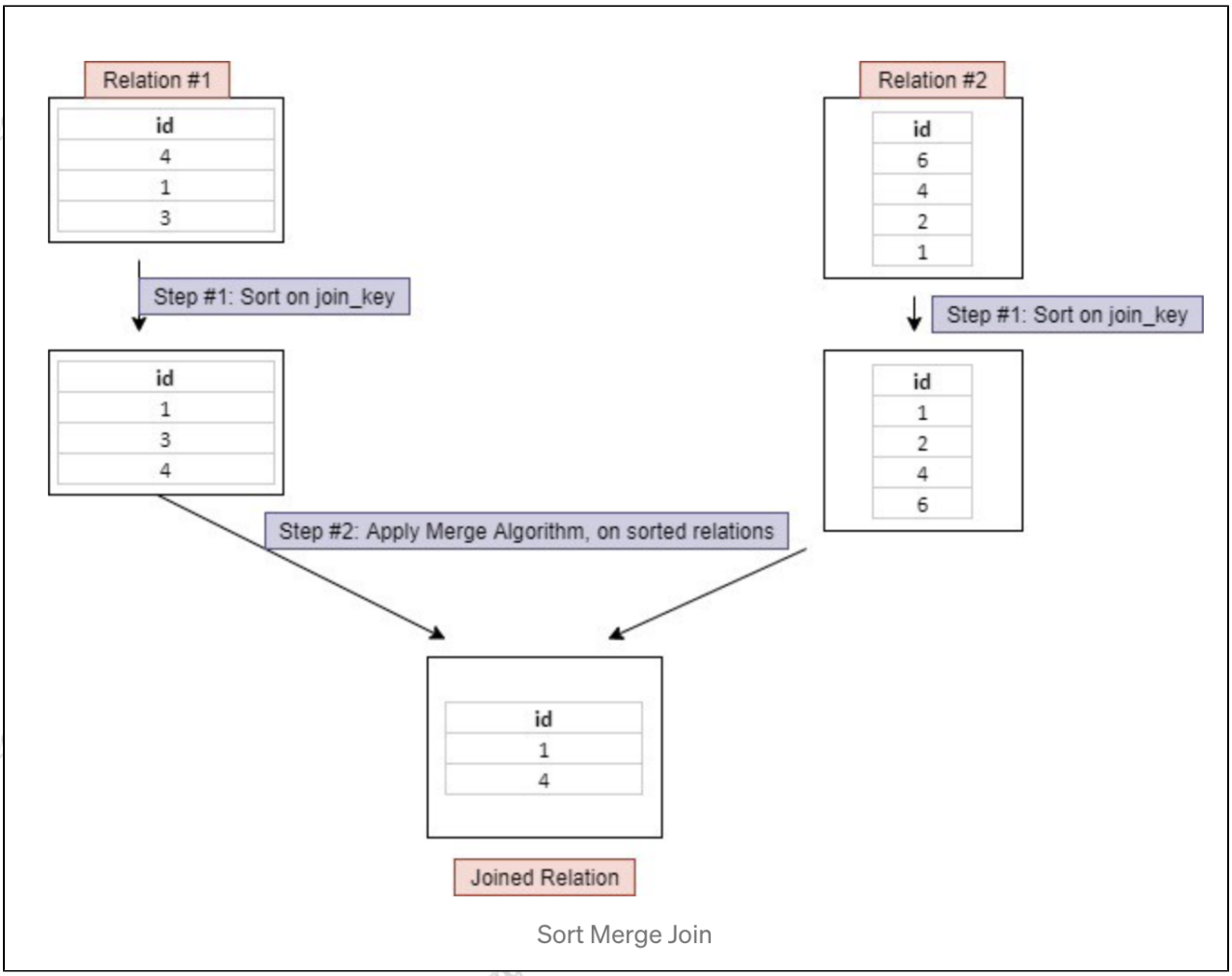
Also shuffle data to get same join_key on the same worker node, then perform sort-merge join at the partition level in worker nodes. Only support:

- '=' join
- Join key needs to be sortable
- Support all join types

- Default join strategy when broadcast hash join not possible



Shuffle Sort Merge Join



Sort Merge Join

Optimization Techniques

Bucketing

An optimization technique that uses**buckets**to determine data partition and avoid data shuffles when performing join. Bucketing results in fewer exchanges (thus fewer stages).

Bucketing is most beneficial when**pre-shuffled bucketed tables**are used more than once as bucketing takes time. Thus we hope that multiple join queries later will offset bucketing cost.

```
E.g.: Joining two tables using SortMergeJoin will generate Exchange physical operators to shuffle the table
datasets. If we bucket each table beforehand:
1. Original number of partitions: 8
2. Bucket by: 4
3. Number of bucketing files: num partitions * buckets = 8 * 4 = 32 files

With bucketing, exchange no longer needed. Number of partitions of a bucketed table = number of buckets = 4.
```

However, below are requirements for Spark Optimizer to give no-Exchange query plan:

1. Number of partitions on both sides of a join is exactly the same
2. Both join operators have to use**HashPartitioning**partitioning scheme

UI breakdown

https://spark.apache.org/docs/latest/web-ui.html

Jobs Tab

- A **summary page** for all jobs
  - Event timeline: Chronological order of executors (added or removed) and jobs
  - Jobs details grouped by status
    - Job ID, description (with link to Job page), submitted time, duration, stages summary and tasks progress bar
- A **details page** for each job:
  - Job status (running, succeeded, failed)
  - Number of stages per status (active, pending, completed, skipped, failed)
  - Associated SQL query (link to SQL tab)
  - Event timeline: executors and stages of the job
  - DAG
  - List of stages grouped by status (active, pending, completed, skipped, failed)
    - Stage ID, description, submitted timestamp, duration of stage, tasks progress bar
    - Input: bytes read from storage in this stage
    - Output: bytes written to storage in this stage
    - **Shuffle read**: total shuffle bytes and records read, include both data read locally and from remote executors
    - **Shuffle write**: bytes and records written to disk in order to be read by a shuffle in future stage

Stages Tab

- A**summary page** for current stage of all stages in all jobs in the Spark application
  - Details of stages per status(active, pending, completed, skipped, failed)
    - Description: view**task**detail
- A**stage detail**page:
  - Total time across all tasks
  - Locality level summary
    - If data and the code that operates on it are local on the same machine then the computation would be faster. If not, normally code will be serialised to be shipped to data as its size is smaller.
      - PROCESS_LOCAL: data in same JVM as running code, **best** locality possible
      - NODE_LOCAL: data on same node, can be in HDFS on same node or in another executor (core) of the same node. A little slower as data is transferred across processes
      - NO_PREF, RACK_LOCAL, ANY
  - Shuffle Read Size / Records
  - Associated Job IDs
  - DAG: nodes are RDD, edges are operation applied
  - **Summary statistics** of all tasks
    - Tasks deserialisation time
    - Duration of tasks
    - **GC time** is total JVM garbage collection time
    - Result serialisation time: time spent serialising a task result from executor to send back to driver
    - Getting result time: time driver spends fetching task results from workers
    - Scheduler delay: time task waits to be scheduled for execution
    - **Peek execution memory**: max memory used by internal data structures created during shuffles, aggregations and joins
    - **Shuffle Read Size / Records:**Total shuffle bytes read, includes both data read locally and data read from remote executors
    - Shuffle Read Blocked Time: Time tasks spent blocked waiting for shuffle data to be read from remote machines
    - Shuffle Remote Reads:total shuffle bytes read from remote executors
    - Shuffle Spill (memory): size of deserialised shuffle data in memory
    - Shuffle Spill (disk): size of serialised data on disk
  - **Aggregated metrics**by executor show same information for executor
  - **Accumulators:** shared variables
  - **Task Detail** Page:
    - Some others: Logs/ re-try attempts/ accumulators value at the end of this task

Storage Tab

Display persisted RDDs and DataFrames if any within the application. The summary page shows:

- Storage level
- Number of partitions
- Memory overhead

We can click on each RDD to view details of data persistence such as data distribution on the cluster

Environment Tab

Display values of different environment and configuration variables, including properties from:

- Runtime: versions of Java and Scala
- Spark: **application** properties
- Hadoop:
- System: More details about the JVM
- Classpath Entries: list classes loaded from different sources

Executors Tab

Summary information about the executors created for the application, including:

- Memory, disk and cores used by each executor
- Performance information (GC, shuffle, task)
- Thread dump: a snapshot of the state of all threads that are part of the process, useful for performance analysis

SQL Tab

Display durations, jobs, physical and logical plans for the queries.

SQL metrics, useful when diving into the execution details of each operator:

- Number of output rows
- Data size: Size of broadcast/shuffled/collected data (BroadcastExchange, ShuffleExchange, Subquery operators)
- Time to collect: Time spent collecting data (BroadcastExchange, Subquery)
- Scan time: Time spent scanning data (ColumnarBatchScan, FileSourceScan)
- Metadata time: time spent getting metadata like number of partitions, number of files (FileSourceScan)
- Spill size: Number of bytes spilled to disk from memory in operator
- Data size of build side: size of built hash map (ShuffledHashJoin)
- Time to build hash map: (ShuffledHashJoin)

References:

http://www.gatorsmile.io/anticipated-feature-in-spark-2-2-max-records-written-per-file/

https://gist.github.com/dusenberrymw/30cebf98263fae206ea0ffd2cb155813

https://github.com/apache/spark/blob/128c29035b4e7383cc3a9a6c7a9ab6136205ac6c/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L376

https://stackoverflow.com/questions/31610971/spark-repartition-vs-coalesce

https://luminousmen.com/post/spark-partitions