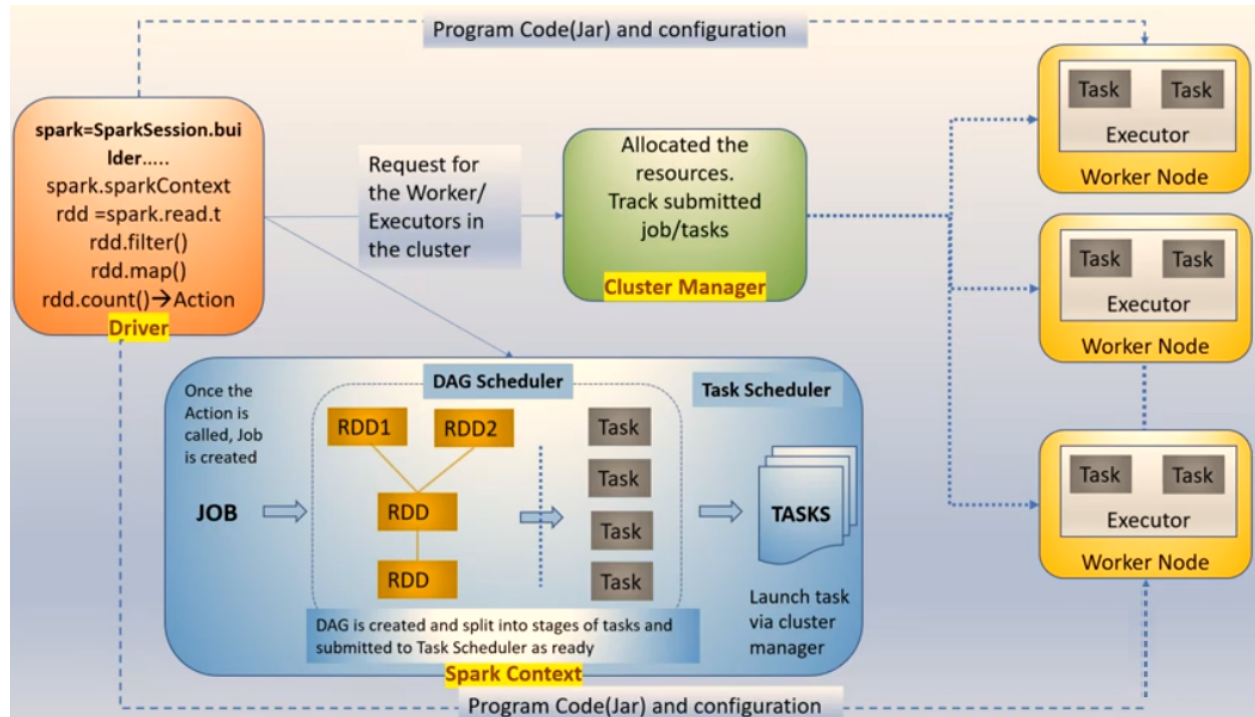


## Spark Performance Tuning



### Tuning Considerations:

- [Data Serialization](#)
- [Common Practices](#)
- [Memory Tuning](#)
- [Level of Parallelism](#)
- [Joins](#)
- [Shuffling](#)
- [Pushdown optimization, pruning](#)
- [Memory Usage](#)
- [Broadcasting Large Variables](#)
- [Data Locality](#)
- [Spark UI Management](#)

### Usage of UDFs in Spark SQL

Can we use UDFs in Spark or why UDF impacts the performance of spark?

When we use UDFs we end up losing all the optimization Spark does on our Dataframe/Dataset. UDF is a black box to Spark's optimizer. Let's consider an example of a general optimization when reading data from a storage is applied automatically with PredicatePushdown. What it does is **minimize the amount of data that is read from the source itself at the time of read**. Eg:

```

import org.apache.spark.sql.types._
val
transstrt=StructType(Array(StructField("txnid", IntegerType,true),StructField("dt",StringType,true),StructField("cid",StringType,true),StructField("amt",FloatType,true),StructField("category",StringType,true) ,StructField("product",StringType,true) ,StructField("city",StringType,true) ,StructField("state",StringType,true) ,StructField("transtype",StringType,true)))
val txns=spark.read.schema(transstrt).
csv("file:///home/hduser/hive/data/txns").
toDF("txnid","dt","cid","amt","category","product","city","state","transtype")
txns.show(10);
txns.createOrReplaceTempView("trans");
val
strt=StructType(Array(StructField("id",IntegerType,true),StructField("fname",StringType,true),StructField("lname",StringType,true),StructField("age",IntegerType,true),StructField("profession",StringType,true)))
val ds=spark.read.schema(strt).csv("file:///home/hduser/hive/data/custs")
ds.write.mode("overwrite").parquet("file:///home/hduser/parquetds")
val dfparquetread = spark.read.parquet ("file:///home/hduser/parquetds");
dfparquetread.where('fname === "Paige").queryExecution.executedPlan

```

If we create an udf and called, see the difference.

```

val isJoey = udf((name:String) => name == "Paige")
dfparquetread.where(isJoey('fname)).queryExecution.executedPlan

```

1. Optimizations that spark do will be end up of using UDFs. So it is always suggested to avoid UDFs as long as it is inevitable
2. It is the responsibility of the programmer to make sure that the UDFs are handled gracefully

**Conclusion:** Avoid using UDFs as much as you can

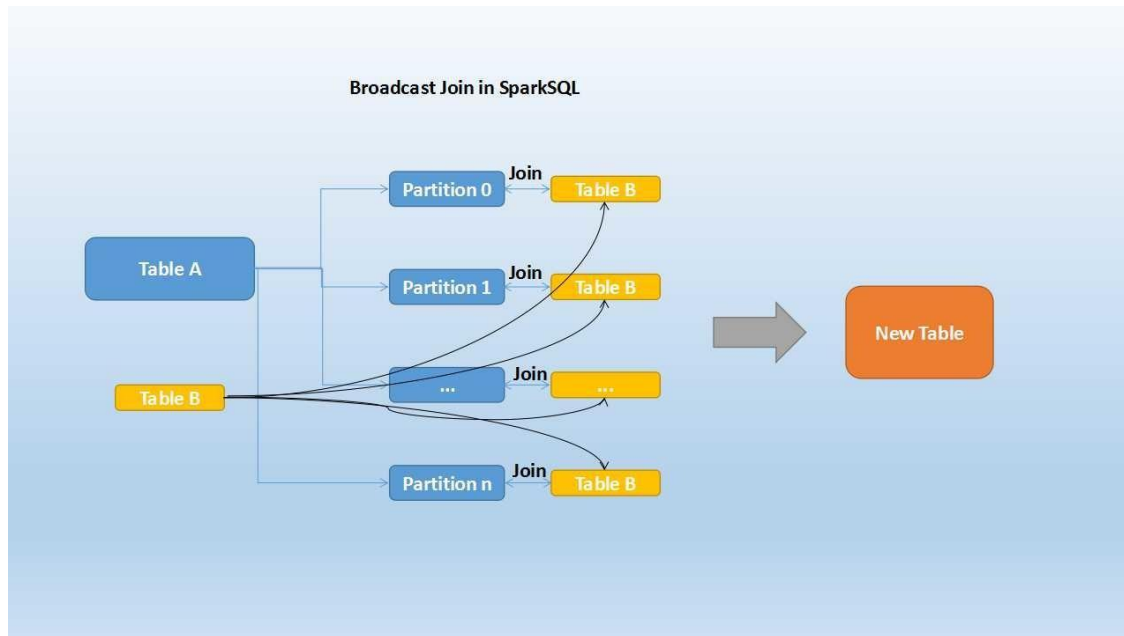
### Spark SQL Joins

**How to optimize joins in spark ? or how do you improve join performance in Spark ? or how to improve spark sql performance?**

Join Optimization – Broadcast join, Sort Merge, Shuffle Hash join

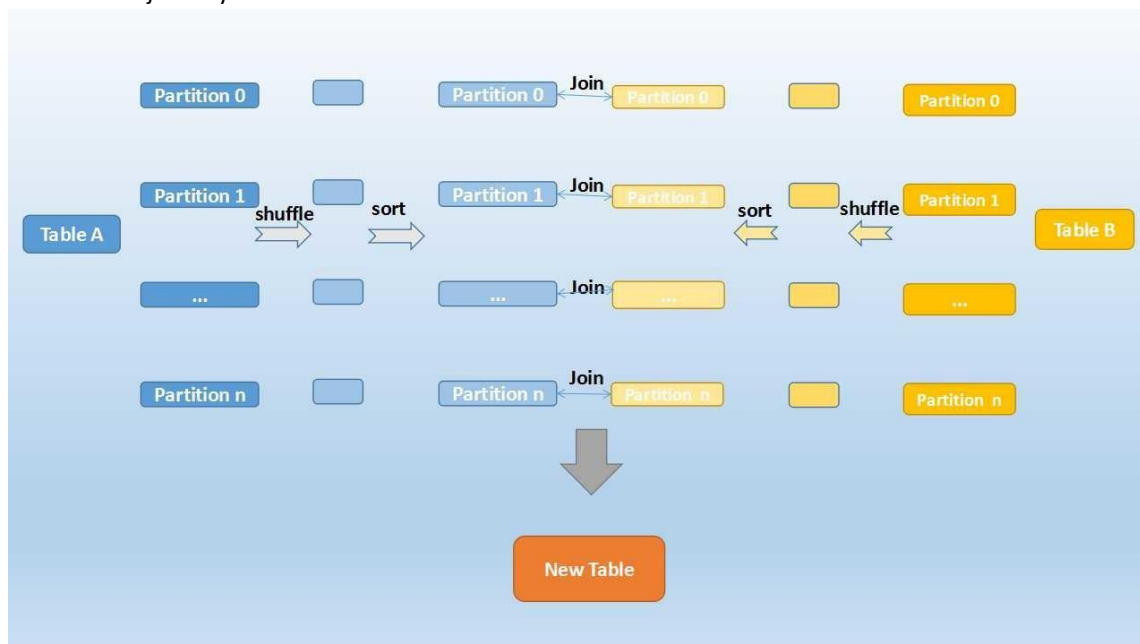
**Broadcast Join (Map side join) – usecase: Fact table Transactions with Dim table join product table**

*The Broadcast Join (BJ) is chosen when one of the Dataset participating in the join is known to be broadcastable. A Dataset is marked as broadcastable if its size is less than **spark.sql.autoBroadcastJoinThreshold** (default 10 mb)* . Easily Broadcast joins are the one which yield the **maximum performance** in spark. However, it is relevant only for **little datasets**. In broadcast join, the **smaller table will be broadcasted to all worker nodes**. Thus, when working with one large table and another smaller table always makes sure to broadcast the smaller table. **We can hint spark to broadcast a table.**

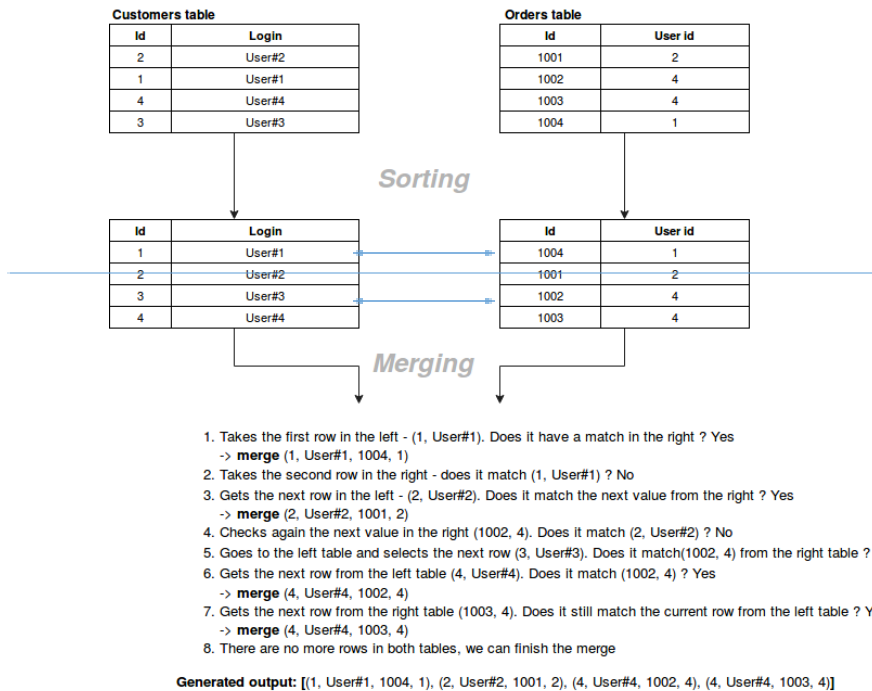


**Sort Merge Join (Sort merge reduce side sorted and bucket join) – usecase: Fact table Transactions with fact table join weblog\_events table**

If the **matching keys are sortable**, not eligible for shuffle/broadcast, it uses **MORE OF DISK**, When both tables are **very large**. When the two tables are very large, Spark SQL uses a new algorithm to join the table, that is, **Sort Merge Join**. The first step is the **ordering** operation made on 2 joined datasets. The second operation is the **merge** of **sorted** data into a single place by simply iterating over the elements and assembling the rows having the same value for the join key.

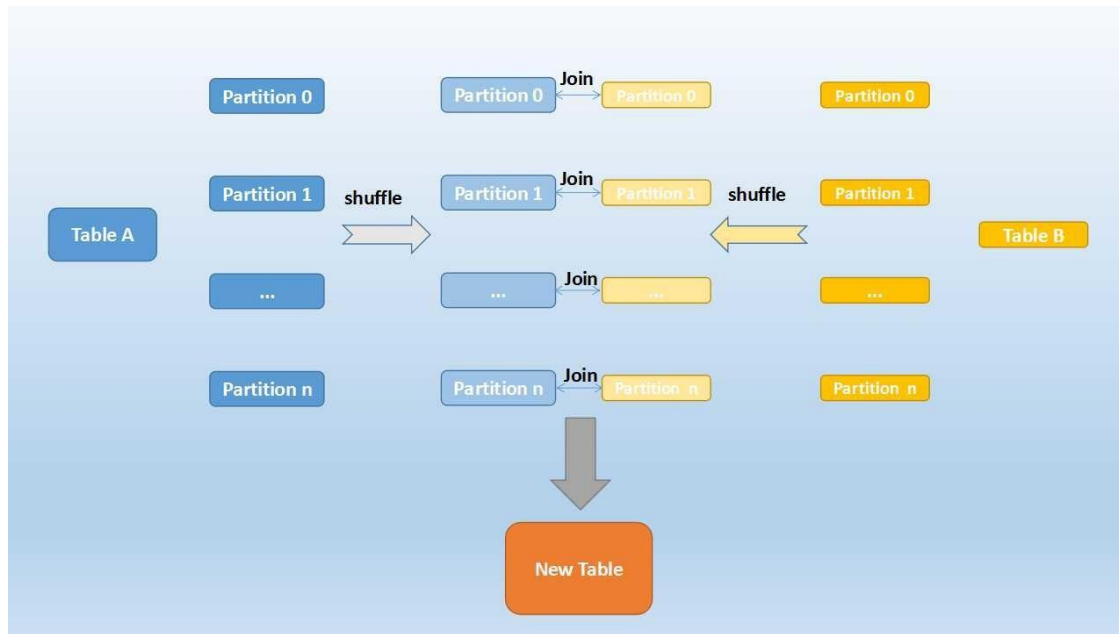


### Sort-merge join algorithm explained



### Shuffle Hash Join (Reduce side join) – Usecase: Fact table wholesale transactions with wholesale weblogevents

Shuffle Hash join works based on the concept of **map reduce generates hashed tables which fits into MEMORY**. Map through the data frames and use the values of the join column as output key. **Shuffles the data frames based on the output keys and join the data frames in the reduce phase as the rows from the different data frame with the same keys will ended up in the same machine. Medium to large dataset.** Spark chooses Shuffle Hash join when Sort merge join is turned off.



#### Conclusion:

1. **Sort-Merge join is the default join and performs well in most of the scenarios.** And for cases, if you are confident enough that Shuffle Hash join is better than Sort-Merge join, disable Sort-Merge join for those scenarios. **However, when the joined data size is smaller than the shuffle size, Shuffle Hash join will do better than Sort-Merge join.**
2. Tune the `spark.sql.autoBroadcastJoinThreshold` accordingly if deemed necessary. Try to use Broadcast joins wherever possible and filter out the irrelevant rows to the join key before the join to avoid unnecessary data shuffling.
3. Joins without unique join keys or **no join keys** can often be **very expensive** and should be avoided.
4. Filter columns and rows before joining as much as u can.

#### Example:

```
import org.apache.spark.sql.types._
val
transstr=StructType(Array(StructField("txnid",IntegerType,true),StructField("dt",StringType,true),StructField("cid",StringType,true),StructField("amt",FloatType,true),StructField("category",StringType,true),StructField("product",StringType,true),StructField("city",StringType,true),StructField("state",StringType,true),StructField("transtype",StringType,true)))
val txns=spark.read.schema(transstr).
csv("file:///home/hduser/hive/data/txns").
toDF("txnid","dt","cid","amt","category","product","city","state","transtype")
txns.show(10);
txns.createOrReplaceTempView("trans");

val
str=StructType(Array(StructField("id",IntegerType,true),StructField("fname",StringType,true),StructField("lname",StringType,true),StructField("age",IntegerType,true),StructField("profession",StringType,true)))
val ds=spark.read.schema(str).csv("file:///home/hduser/hive/data/custs")
ds.createOrReplaceTempView("cust");

val spark = org.apache.spark.sql.SparkSession.builder().appName("Sort merge join test").master("local[*]").getOrCreate()
```

#### **Broadcast join example with threshold of 10 mb by default:**

```
spark.sqlContext.getConf("spark.sql.join.preferSortMergeJoin")
spark.sqlContext.getConf("spark.sql.autoBroadcastJoinThreshold")
spark.sql("""select a.cid,b.id from trans a left join cust b on a.cid=b.id """).explain
```

```

val spark1 = org.apache.spark.sql.Session.builder().appName("Sort-merge join
test").master("local[*]").config("spark.sql.autoBroadcastJoinThreshold", "1").getOrCreate()

spark1.sqlContext.getConf("spark.sql.autoBroadcastJoinThreshold")
spark1.sqlContext.getConf("spark.sql.join.preferSortMergeJoin")

val dflookupsql1 = spark1.sql("""select a.cid from trans a left join cust b on a.cid=b.id """).explain

val spark2 = org.apache.spark.sql.Session.builder().appName("shuffle hash join
test").master("local[*]").config("spark.sql.join.preferSortMergeJoin", "false").config("spark.sql.autoBroadcastJoinThreshold", "-1").getOrCreate()

spark2.sqlContext.getConf("spark.sql.join.preferSortMergeJoin")

val dflookupsql1 = spark2.sql("""select a.cid from trans a left semi join cust b on a.cid=b.id """).explain

val dflookupsql = spark2.sql("""select a.id,b.amt,b.transtype from cust a inner join trans b on a.id=b.cid """)
dflookupsql.explain(true)
dflookupsql.queryExecution.optimizedPlan.statistics.sizeInBytes

```

## **Shuffle Partitions and Repartitions**

**spark-shell --num-executors 2 --executor-cores 1 --executor-memory 1g --master yarn --deploy-mode client**

Running with shuffle partition with default (200) and repartition to 4 later:

```

import org.apache.spark.sql.types._
val
transstrt=StructType(Array(StructField("txnid",IntegerType,true),StructField("dt",StringType,true),StructField("cid",StringType,true),StructField("amt",FloatType,true),StructField("category",StringType,true),StructField("product",StringType,true),StructField("city",StringType,true),StructField("state",StringType,true),StructField("transtype",StringType,true)))

val txns=spark.read.schema(transstrt).
csv("file:///home/hduser/hive/data/txns").
toDF("txnid","dt","cid","amt","category","product","city","state","transtype")
txns.show(10);
txns.createOrReplaceTempView("trans");

val
strt=StructType(Array(StructField("id",IntegerType,true),StructField("fname",StringType,true),StructField("lname",StringType,true),StructField("age",IntegerType,true),StructField("profession",StringType,true)))
val ds=spark.read.schema(strt).csv("file:///home/hduser/hive/data/custs")
ds.createOrReplaceTempView("cust");

```

**After shuffling you may need to increase (after shuffling you are going to join 2 tables where more partitions required for performance) or decrease partitions (when filter applied before shuffling filtered data).**

Usage of sql.shuffle.partitions or repartition / usage of sql.shuffle.partitions or coalesce - which one is good to increase or decrease respectively?

```
spark.sqlContext.getConf("spark.sql.shuffle.partitions")
```

```
val dsgrp=ds.repartition(10).filter($"age">10).
groupBy('age).agg(count($"age"))
```

```
dsgrp.show(10)
dsgrp.repartition(400).queryExecution.executedPlan
dsgrp.repartition(400).count()
```

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
41	count at <console>:36 +details	2021/05/13 17:24:46	70 ms	1/1			230.0 B	
40	count at <console>:36 +details	2021/05/13 17:24:45	0.1 s	4/4			2.5 KB	230.0 B
39	count at <console>:36 +details	2021/05/13 17:24:43	2 s	200/200			28.2 KB	2.5 KB
38	count at <console>:36 +details	2021/05/13 17:24:43	0.6 s	10/10			42.1 KB	28.2 KB
37	count at <console>:36 +details	2021/05/13 17:24:43	99 ms	1/1	382.2 KB			42.1 KB

#### Running with shuffle partition with default value:

```
spark.sqlContext.setConf("spark.sql.shuffle.partitions", "200")
dsgrp.queryExecution.executedPlan
dsgrp.count()
spark.sqlContext.getConf("spark.sql.shuffle.partitions")
```

#### Best one : Running with sql shuffle partition as 400 and repartition to 400 later (not needed):

```
spark.sqlContext.setConf("spark.sql.shuffle.partitions", "400")

val dsgrp=ds.repartition(10).filter($"age">10).
groupBy('age).agg(count($"age")).toDF("age", "cnt_age")
dsgrp.count()
dsgrp.queryExecution.executedPlan
```

If your data is less than a block worth of content then coalesce before writing or how to store the final result in a single file?  
**Notice the Shuffle Read and Write bytes in the UI of the job and look at the target location also.**

```
spark.sqlContext.setConf("spark.sql.shuffle.partitions", "200")
dsgrp.write.mode("overwrite").orc("file:///home/hduser/orcdata")
dsgrp.repartition(2).coalesce(1).write.mode("overwrite").orc("file:///home/hduser/orcdata")
```

#### Better way to use

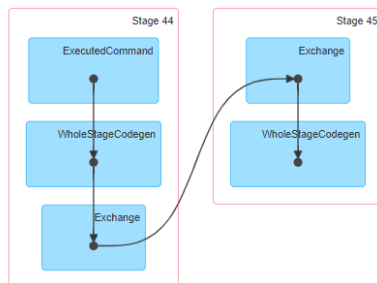
```
spark.sqlContext.setConf("spark.sql.shuffle.partitions", "1")
dsgrp.write.mode("overwrite").orc("file:///home/hduser/orcdata")
```

 2.0.1	Jobs	Stages	Storage	Environment	Executors	SQL	Spark shell applicatio
-------------------------------------------------------------------------------------------	------	--------	---------	-------------	-----------	-----	------------------------

#### Details for Job 18

Status: SUCCEEDED  
Completed Stages: 2

- Event Timeline
- DAG Visualization



#### Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
45	orc at <console>:33 +details	2021/05/20 15:22:40	63 ms	1/1			763.0 B	
44	orc at <console>:33 +details	2021/05/20 15:22:40	0.2 s	1/1	382.2 KB			763.0 B

If your data is more than a block worth of content then repartition before writing.  
**Notice the Shuffle Read and Write bytes in the UI of the job and look at the target location also.**  
ds.repartition(4).write.mode("overwrite").json("file:///home/hduser/jdata")

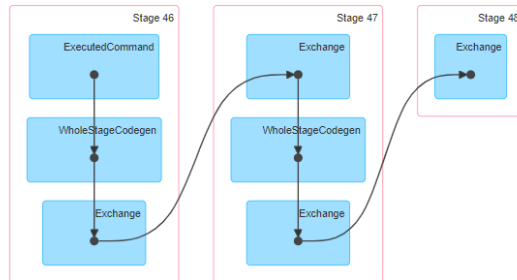
### Better way to use

```
spark.sqlContext.setConf("spark.sql.shuffle.partitions","4")
ds.write.mode("overwrite").json("file:///home/hduser/jdata")
```

### Details for Job 19

Status: SUCCEEDED  
Completed Stages: 3

► Event Timeline  
▼ DAG Visualization



### Completed Stages (3)

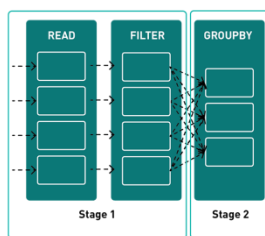
Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
48	json at <console>:33	+details	2021/05/20 15:25:30	0.6 s	4/4			979.0 B	
47	json at <console>:33	+details	2021/05/20 15:25:29	62 ms	1/1			763.0 B	979.0 B
46	json at <console>:33	+details	2021/05/20 15:25:29	0.3 s	1/1	382.2 KB			763.0 B

### Shuffle compress property:

**spark.shuffle.compress** - Whether to compress map output files. Generally a good idea. Compression will use spark.io.compression.codec.

Configures the number of partitions to use when shuffling data for joins or aggregations.

Physical Plan



```
spark.sqlContext.setConf("spark.hadoop.mapred.output.compress", "true")
```

```
spark.sqlContext.setConf("spark.io.compression.codec", "org.apache.hadoop.io.compress.snappy")
```

```
spark.sqlContext.getConf("spark.hadoop.mapred.output.compress")
spark.sqlContext.getConf("spark.io.compression.codec")
```

### How do you control the number of partitions based on the size of data ?



## Partition Bytes

**spark.sql.files.maxPartitionBytes** - The maximum number of bytes to pack into a single partition when reading files. This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.

```
spark.sqlContext.getConf("spark.sql.files.maxPartitionBytes")
```

```
val dfparquet=spark.read.parquet("file:///home/hduser/parquetdata")
dfparquet.queryExecution.optimizedPlan.statistics.sizeInBytes
dfparquet.rdd.getNumPartitions
```

```
spark.sqlContext.setConf("spark.sql.files.maxPartitionBytes","40000")
val dfparquet=spark.read.parquet("file:///home/hduser/parquetdata")
dfparquet.rdd.getNumPartitions
```

```
ds.write.mode("overwrite").parquet("file:///home/hduser/parquetdata")
```

## Read only the column what ever needed (Projection pushdown)

Is it possible to write Spark SQL query on a direct file rather than create a temp view?

```
spark.sql("select age from parquet.`file:///home/hduser/parquetdata`").show
spark.sql("select age from parquet.`file:///home/hduser/parquetdata`").explain(true)
spark.sql("select age from orc.`file:///home/hduser/orcdata`").show
spark.sql("select age from json.`file:///home/hduser/jdata`").explain(true)
```

## Predicate Pushdown & Partition Pruning

Partition pruning in Spark is a performance optimization that limits the number of files and partitions that Spark reads when querying. After filtering the portioned data by using PartitionFilters, queries that match certain other filter criteria PushedFilters that dramatically improve performance by allowing Spark to only read a subset of the directories and files at partition level and inside partition data level.



```
ds.write.mode("overwrite").partitionBy("age").json("file:///home/hduser/jsonpart")
ds.write.mode("overwrite").json("file:///home/hduser/jsonpart1")
spark.read.json("file:///home/hduser/jsonpart").filter($"id">10).explain
spark.read.json("file:///home/hduser/jsonpart").filter($"age">60).explain(true)
spark.read.json("file:///home/hduser/jsonpart").filter($"age">10 && $"id">10.1).explain(true)
spark.read.json("file:///home/hduser/jsonpart").explain(true)
```

## Memory Tuning

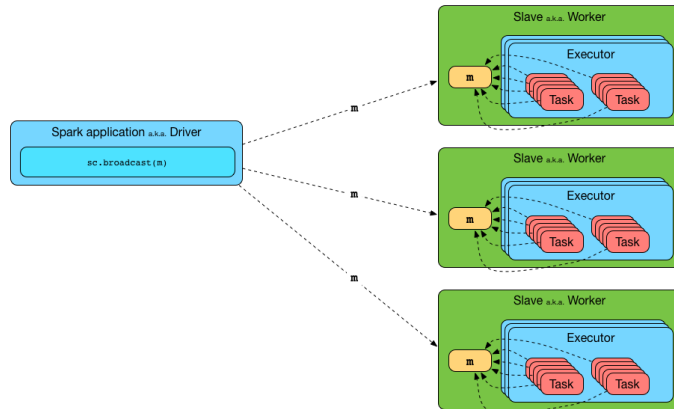
Consider the following three things in tuning memory usage:

- Amount of memory used by objects (the entire dataset should fit in-memory)
- The cost of accessing those objects
- Overhead of garbage collection.
  - **spark.broadcast.blockSize** – Size of block when broadcast.
  - **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.fraction** expresses the size of M as a fraction of the (JVM heap space - 300MB) (default 0.6). The rest of the space (40%) is reserved for user data structures, internal metadata in Spark, and safeguarding against **OOM** errors in the case of sparse and unusually large records.
- **spark.yarn.executor.memoryOverhead** property is added to the executor memory to determine the full memory request to YARN for each executor. It defaults to  $\max(\text{executorMemory} * 0.10 \text{ with minimum of } 384) \rightarrow 512+51+384$

### Broadcasting Large Variables

The size of each serialized task reduces by using broadcast functionality in SparkContext. If a task uses a large object from driver program inside of them, turn it into the broadcast variable.



```
spark.sparkContext.getConf.getOption("spark.broadcast.blockSize")
spark.sparkContext.getConf.getOption("spark.broadcast.compress")
```

### How to avoid data Skew?

Data skew means that data distribution is uneven or asymmetric. Symmetry means that one half of the distribution is a mirror image of the other half.

**Data skew happens when for one reason or another, a small percentage of partitions get most of the data being processed.** In normal usage, Spark will generally make sure that the data is evenly split across all tasks, so there isn't a big risk of skew. Eg: One top product may be purchased by more number of people, When you do a join with the product\_key, Spark distributes the data by join key, so that data from the two tables being joined will be in the same task. If you have a lot of rows with the same key, then you have some tasks with those keys taking much longer than the others.

**Solution: Repartitioning, Shuffle partitions and Broadcasting (spark.sql.autoBroadcastJoinThreshold) small tables or by adding salting to the join keys if you can't do it with broadcasting.**

**Salting for joins:** How can we make the data distributed evenly? The skew problems can be fixed using the salting technique. Salting is a technique where we will add random values to the join key of one of the tables. In the other table, we need to replicate the rows to match the random keys. The idea is if the join condition is satisfied by `LAPTOP == LAPTOP`, it should also get satisfied by `select table1.product_key from table1 where table1.product_key+random()=table2.product_key+random()` - `LAPTOP_100 = LAPTOP_100`, `LAPTOP_101 = LAPTOP_101`. The value of salt will help the dataset to be more evenly distributed.

### Serialized RDD Storage

When your objects are still too large to efficiently store despite this tuning, a much simpler way to reduce memory usage is to store them in *serialized* form, using the serialized StorageLevels in the RDD persistence API, such

as MEMORY\_ONLY\_SER. **Spark will then store each RDD partition as one large byte array. The only downside of storing data in serialized form is slower access times, due to having to deserialize each object on the fly.** Executor memory is less than the data size you are going to cache MEMORY\_ONLY\_SER otherwise MEMORY\_ONLY will be utilized, but partial data will be kept in disk and partial will be loaded into the memory.

### Login to Spark local mode

#### spark-shell

```
import org.apache.spark.sql.types._
val
transstrt=StructType(Array(StructField("txnid",IntegerType,true),StructField("dt",StringType,true),StructField("cid",StringType,true),StructField("amt",FloatType,true),StructField("category",StringType,true),StructField("product",StringType,true),StructField("city",StringType,true),StructField("state",StringType,true),StructField("transtype",StringType,true)))
val txns=spark.read.schema(transstrt).
csv("file:///home/hduser/hive/data/txns").
toDF("txnid","dt","cid","amt","category","product","city","state","transtype")
txns.show(10);
txns.createOrReplaceTempView("trans");
txns.unpersist()
txns.persist(org.apache.spark.storage.StorageLevel.MEMORY_ONLY)
txns.count
```

### Data Serialization (Kryo)

It is the process of converting the in-memory object to another format that can be used to store in a file or send over the network. It plays a distinctive role in the performance of any distributed application. Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all Serializable types and requires you to register the classes you'll use in the program in advance for best performance

```
spark.sqlContext.getConf("spark.serializer")
val spark = org.apache.spark.sql.SparkSession.builder().appName("kryo").master("local[*]").config("spark.serializer",
"org.apache.spark.serializer.KryoSerializer").getOrCreate()
spark.sqlContext.getConf("spark.serializer")

import org.apache.spark.sql.types._
val
transstrt=StructType(Array(StructField("txnid",IntegerType,true),StructField("dt",StringType,true),StructField("cid",StringType,true),StructField("amt",FloatType,true),StructField("category",StringType,true),StructField("product",StringType,true),StructField("city",StringType,true),StructField("state",StringType,true),StructField("transtype",StringType,true)))
val txns=spark.read.schema(transstrt).
csv("file:///home/hduser/hive/data/txns").
toDF("txnid","dt","cid","amt","category","product","city","state","transtype")
txns.show(10);
txns.createOrReplaceTempView("trans");
txns.unpersist()
txns.persist(org.apache.spark.storage.StorageLevel.MEMORY_ONLY_SER)
txns.count
```

### Look at the Task Deserialization time

### Don't forget to unpersist after completing all repeated actions on the persisted RDDs

`txns.unpersist()`

### Common Best practices

- ✓ Usage of Cache/persist
- ✓ Usage of serialized file formats and compression techniques

- ✓ Don't copy all elements of a large RDD to the driver If your RDD is so large that all of its elements won't fit in memory on the driver machine, don't do this: `val values = myVeryLargeRDD.collect()` , Collect will attempt to copy every single element in the RDD onto the single driver program, and then run out of memory and crash. Instead, you can make sure the number of elements you return is capped by calling `take` or `takeSample`, or perhaps filtering or sampling your RDD.
- ✓ **Gracefully Dealing with Bad Input Data** - Handle exception handling, `rdd.isEmpty` check, count the number of elements in the array, use dataset for typesafe, define the right datatype to avoid casting of types eg. `Filter ((age#372 > 10) && (cast(id#369L as double) > 10.0))` or `*Filter (isNotNull(id#387L) && (id#387L > 10))`
- ✓ Use Dataframes/Datasets as much as possible
- ✓ Use Dynamic Allocation where ever applicable

### Dynamic Allocation

#### How do you handle variable source data size in spark?

Identify the size/number of rows and increase or decrease the number of partitions using `coalesce/repartition`, shuffle partitions, broadcast or yarn dynamic allocation.

What is Spark dynamic allocation, what are the configuration settings you do?

**Dynamic allocation** allows **Spark** to **dynamically** scale the cluster resources **allocated** to your application based on the workload. When **dynamic allocation** is enabled and a **Spark** application has a backlog of pending tasks, it can request executors.

```
spark-submit --verbose \
--class org.inceptez.stream.filestream \
--master yarn /home/hduser/install/Sparktwitterusecase/target/sparkInceptez-0.0.1-SNAPSHOT.jar \
--deploy-mode client \
--queue dataeng \
--driver-memory 512m \
--num-executors 1 \
--executor-cores 2 \
--executor-memory 512m \
--spark-shuffle-compress true \
--spark-speculation true \
--spark-dynamicAllocation-enabled true \
--spark-dynamicAllocation-initialExecutors 1 \
--spark-dynamicAllocation-minExecutors 1 \
--spark-dynamicAllocation-maxExecutors 8 \
--spark-dynamicAllocation-executorIdleTimeout 30s \
--spark-dynamicAllocation-schedulerBacklogTimeout 10s \
--conf spark.shuffle.service.enabled true \
--conf spark.sql.shuffle.partitions=4
```

*spark.dynamicAllocation.enabled* – when this is set to true we need not mention executors. The reason is below:

The static parameter numbers we give at spark-submit is for the entire job duration. However if dynamic allocation comes into picture, there would be different stages like the following:

#### **What is the number for executors to start with:**

Initial number of executors (*spark.dynamicAllocation.initialExecutors*) to start with

#### **Controlling the number of executors dynamically:**

Then based on load (tasks pending) how many executors to request. This would eventually be the number what we give at spark-submit in static way. So once the initial executor numbers are set, we go to min (*spark.dynamicAllocation.minExecutors*) and max (*spark.dynamicAllocation.maxExecutors*) numbers.

#### **When to ask new executors or give away current executors:**

When do we request new executors (*spark.dynamicAllocation.schedulerBacklogTimeout*) – This means that there have been pending tasks for this much duration. So the request for the number of executors requested in each round increases exponentially from the previous round. For instance, an application will add 1 executor in the first round, and then 2, 4, 8 and so on executors in the subsequent rounds. At a specific point, the above property max comes into picture. When do we give away an executor is set using *spark.dynamicAllocation.executorIdleTimeout*.

To conclude, if we need more control over the job execution time, monitor the job for unexpected data volume the static numbers would help. By moving to dynamic, the resources would be used at the background and the jobs involving unexpected volumes might affect other applications.

#### **How Do you tune the spark jobs using UI ? Spark Sql execution phases?**

<http://:18080>.

Let's walk through the three phases and the Spark UI information about the phases, with some example code.

#### **The Spark Execution Model**

The Spark execution model can be defined in three phases: creating the logical plan, translating that into a physical plan, and then executing the tasks on a cluster. You can view useful information about your Spark jobs in real time in a web browser with this URL: <http://:4040>. For Spark applications that have finished, you can use the Spark history server to see this information in a web browser at this URL

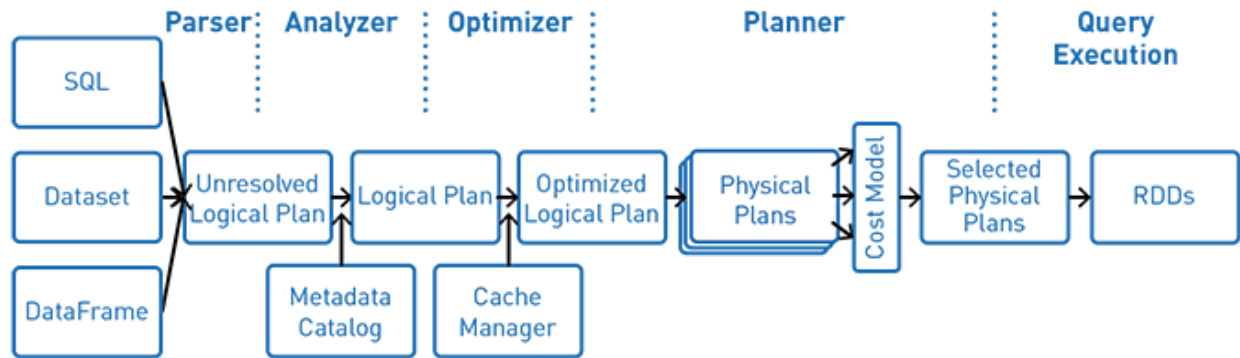
#### **The Logical Plan**

In the first phase, the logical plan is created. This is the plan that shows which steps will be executed when an action gets applied. Recall that when you apply a transformation on a Dataset, a new Dataset is created. When this happens, that new Dataset points back to the parent, resulting in a lineage or directed acyclic graph (DAG) for how Spark will execute these transformations.

#### **The Physical Plan**

Actions trigger the translation of the logical DAG into a physical execution plan. The Spark Catalyst query optimizer creates the physical execution plan for DataFrames, as shown in the diagram below:

JOB SUBMISSION -> AM -> EMPTY EXECUTORS-> **DRIVER - DF/SQL/DS (DSL) -> LP -> PP -> RDDs -> DAGs (DAG SCHEDULER) -> STAGES & TASKS -> TASK SCHEDULER -> DRIVER (DAG/TASKS) -> EXECUTORS -> SHUFFLE MAP STAGE (MAPPER STAGE) -> RESULT STAGE (REDUCER) -> RESULT PRODUCED IN FS/DB/NOSQLS/CONSOLE**



The physical plan identifies resources, such as memory partitions and compute tasks, that will execute the plan.

### Viewing the Logical and Physical Plan

You can see the logical and physical plan for a Dataset by calling the `explain(true)` method. In the code below, we see that the DAG for `df2` consists of a `FileScan`, a `Filter`, and selecting columns.

```

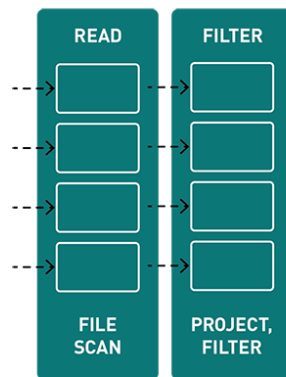
ds.explain(true)
result:
== Parsed Logical Plan ==
'Filter ('depdelay > 40)
+- Relation[_id#8, arrdelay#9,...] json

== Analyzed Logical Plan ==
_id: string, arrdelay: double...
Filter (depdelay#15 > cast(40 as double))
+- Relation[_id#8, arrdelay#9,...] json

== Optimized Logical Plan ==
Filter (isnotnull(depdelay#15) && (depdelay#15 > 40.0))
+- Relation[_id#8, arrdelay#9,...] json

== Physical Plan ==
*Project [_id#8, arrdelay#9,...]
+- *Filter (isnotnull(depdelay#15) && (depdelay#15 > 40.0))
   +- *FileScan json [_id#8, arrdelay#9,...] Batched: false, Format: JSON, Location: InMemoryFileIndex[maprfs:///...],

```

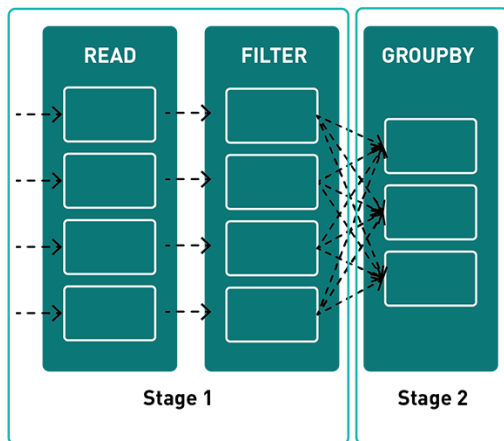


You can see more details about the plan produced by Catalyst on the web UI SQL tab (<http://:4040/SQL/>). Clicking on the query description link displays the DAG and details for the query.

### Executing the Tasks on a Cluster

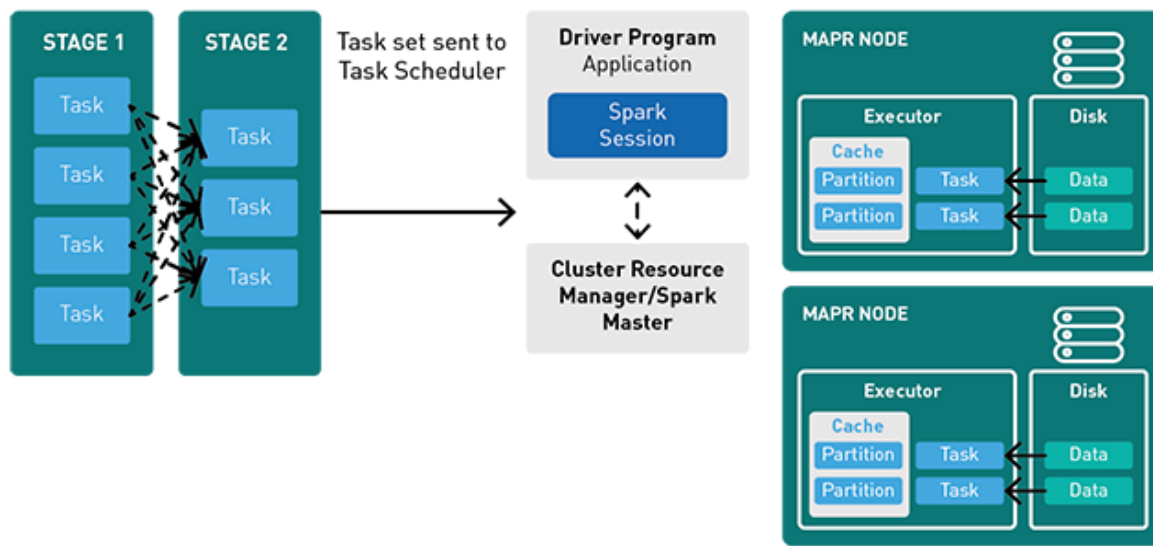
In the third phase, the tasks are scheduled and executed on the cluster. The scheduler splits the graph into stages, based on the transformations. The narrow transformations (transformations without data movement) will be grouped (pipe-lined) together into a single stage. The physical plan for this example has two stages, with everything before the exchange in the first stage.

#### Physical Plan



Each stage is comprised of tasks, based on partitions of the Dataset, which will perform the same computation in parallel.

The scheduler submits the stage task set to the task scheduler, which launches tasks via a cluster manager. These phases are executed in order, and the action is considered complete when the final phase in a job completes. This sequence can occur many times when new Datasets are created.



Here is a summary of the components of execution:

- **Task:** a unit of execution that runs on a single machine
- **Stage:** a group of tasks, based on partitions of the input data, which will perform the same computation in parallel
- **Job:** has one or more stages
- **Pipelining:** collapsing of Datasets into a single stage, when Dataset transformations can be computed without data movement
- **DAG:** Logical graph of Dataset operations

## Exploring the Task Execution on the Web UI

How many number of Jobs, Stages, Tasks created as a part of the below program?

```
txns.filter(function).map(function). repartition (6).join()
```

1 job, 3 stages, 8+6+200

txns(1gb).filter(function).map(function) - stage1 - number of task=8

repartition (6) - stage2 - number of task=6

join() - stage3 - number of task = sql.shuffle.partition (200)

### I want to debug my code in Prod, where my code is running properly in Dev? or for Performance tuning?

Convert the spark-submit job created into spark-shell (make the deploy-mode client, use rest of all parameters as is and use the raw code (github) not the jar), run each and every step to identify the root cause or tune.

```
spark-shell --num-executors 2 --executor-cores 1 --executor-memory 1g --master yarn --deploy-mode client
```



```
import org.apache.spark.sql.types._
val
transstrt=StructType(Array(StructField("txnid",IntegerType,true),StructField("dt",StringType,true),StructField("cid",
StringType,true),StructField("amt",FloatType,true),StructField("category",StringType,true)
,StructField("product",StringType,true) ,StructField("city",StringType,true) ,StructField("state",StringType,true)
,StructField("transtype",StringType,true)))
val txns=spark.read.schema(transstrt).
csv("file:///home/hduser/hive/data/bigtxns").
toDF("txnid","dt","cid","amt","category","product","city","state","transtype")
txns.cache
```

```
txns.count
```

```
txns.count
```

```
txns.coalesce(2).count
```

```
txns.repartition(4).count
```

```
spark.sqlContext.setConf("spark.sql.shuffle.partitions","1")
```

Single executor operation – only process\_local

```
txns.where("""category=='Games'""").show
txns.show(10);
```

```
import org.apache.spark.sql.functions._
txns.where("""category=='Games'""").groupBy("state").agg(max("amt")).show
```

Look at the skipped stage , check the locality level also

**Stage Skipped** means that data has been fetched from cache and re-execution of the given stage is not required. Basically the stage has been evaluated before, and the result is available without re-execution. It is consistent with your DAG which shows that the next stage requires shuffling (reduceByKey). Whenever there is shuffling involved **Spark automatically caches generated data**

```
txns.where("""category=='Games'""").groupBy("state").agg(max("amt")).show
spark.sqlContext.setConf("spark.sql.shuffle.partitions","20")
txns.createOrReplaceTempView("trans");
txns.where("""category=='Games'""").groupBy("state").agg(max("amt")).write.mode("overwrite").json("hdfs://loca
lhost:54310/user/hduser/jsondata")
txns.where("""category=='Games'""").groupBy("state").agg(max("amt")).show
```


Here is a screenshot of the web UI Jobs tab, after running the code above. The Jobs page gives you detailed execution information for active and recently completed Spark jobs. It gives you the performance of a job and also the progress of running jobs, stages, and tasks. In this example, Job Id 2 is the job that was triggered by the collect action on df3.

## Jobs Tab

← → ↺

127.0.01:4040/jobs/

🔍 ☆

 2.2.1-mapr-1803

Jobs

Stages

Storage

Environment

Executors

SQL

Spark shell application

### Spark Jobs <sup>(?)</sup>

User: mapr  
Total Uptime: 27 min  
Scheduling Mode: FIFO  
Completed Jobs: 3  
[▶ Event Timeline](#)

#### Completed Jobs(3)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	<a href="#">collect at &lt;console&gt;:43</a>	2018/07/31 20:37:39	4 s	2/2	202/202
1	<a href="#">take at &lt;console&gt;:41</a>	2018/07/31 20:37:35	0.3 s	1/1	1/1
0	<a href="#">json at &lt;console&gt;:36</a>	2018/07/31 20:37:31	2 s	1/1	2/2

Clicking the link in the Description column on the Jobs page takes you to the Job Details page. This page gives you details on the progress of the job, stages, and tasks. We see this job consists of 2 stages, with 2 tasks in the stage before the shuffle and 200 in the stage after the shuffle.

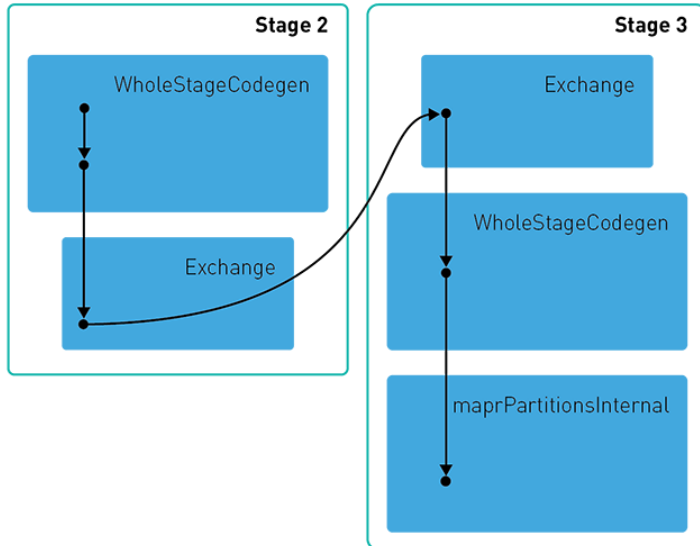
## DETAILS FOR JOB 2

Status: SUCCEEDED

Completed Stages: 2

► Event Timeline

▼ DAG Visualization



### Completed Stages (2)


Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	<a href="#">collect at &lt;console&gt;:43 +details</a>	2018/07/31/ 20:37:41	3 s	200/200			607.0 B	
2	<a href="#">collect at &lt;console&gt;:43 +details</a>	2018/07/31/ 20:37:39	1 s	2/2	8.2 MB			607.0 B

## Stage Tab

The number of tasks correspond to the partitions: after reading the file in the first stage, there are 2 partitions; after a shuffle, the default number of partitions is 200. You can see the number of partitions on a Dataset with the `rdd.partitions.size` method shown below.

Under the Stages tab, you can see the details for a stage by clicking on its link in the description column.

[←](#) [→](#) [↺](#)

 2.2.1-mapr-1803 [Jobs](#) [Stages](#) [Storage](#) [Environment](#) [Executors](#) [SQL](#) [Spark shell application](#)

## Stages for All Jobs

Completed Jobs: 4

### Completed Stages(4)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	<a href="#">collect at &lt;console&gt;:43 +details</a>	2018/07/31/ 20:37:41	3 s	200/200			607.0 B	
2	<a href="#">collect at &lt;console&gt;:43 +details</a>	2018/07/31/ 20:37:39	1 s	2/2	8.2 MB			607.0 B
1	<a href="#">take at &lt;console&gt;:41 +details</a>	2018/07/31/ 20:37:35	0.2 s	1/1	6.1 MB			
0	<a href="#">json at &lt;console&gt;:36 +details</a>	2018/07/31/ 20:37:31	2 s	2/2	8.2 MB			

Here we have summary metrics and aggregated metrics for tasks and aggregated metrics by executor. You can use these metrics to identify problems with an executor or task distribution. If your task process time is not balanced, the resources could be wasted.

**The Storage tab** provides information about persisted Datasets. The Dataset is persisted if you call Persist or Cache on the Dataset, followed by an action to compute on that Dataset. This page tells you which fraction of the Dataset's underlying RDD is cached and the quantity of data cached in various storage media. Look at this page to see if important Datasets are fitting into memory. You can also click on the link to view more details about the persisted Dataset. If you no longer need a cached Dataset, you can call Unpersist to uncache it.

<div><div>← → ↺</div><div>127.0.0.1:4040/Storage/</div><div>🔍 ☆</div></div>					
<div><div>Spark2.2.1-mapr-1803</div><div>Jobs</div><div>Stages</div><div>Storage</div><div>Environment</div><div>Executors</div><div>SQL</div><div>Spark shell application</div></div>					
<h2>Storage</h2>					
<h3>RDDs</h3>					
RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
*Project [_id#8, arrdelay#9, carrier#10, crstime#11L, crsdephour#12L, crsdeptime#13L, crselapsedtime#14, depdelay#15, dest#16, dist#17, doFW#18L, origin#19] +- Filter (isontnull(depdelay#15) \$\$ (depdelay#15> 40.0)) +-FileScan json [_id#8, arrdelay#9, carrier#10, crstime#11L, crsdephour#12L, crsdeptime#13L, crselapsedtime#14, depdel...	Memory Deserialized 1x Replicated	2	100%	330.4 KB	0.0 B

Notice how the execution time is faster after caching.

```
df2.cache
df2.count
df3.collect
```

```
spark.sqlContext.getConf("spark.serializer")
val spark =
org.apache.spark.sql.Session.builder().appName("kryo").master("local[*]").config("spark.serializer",
"org.apache.spark.serializer.KryoSerializer").getOrCreate()
spark.sqlContext.getConf("spark.serializer")
txns.collect
java.lang.OutOfMemoryError: Java heap space
```

**Stage Skipped** means that data has been fetched from cache and re-execution of the given stage is not required. Basically the stage has been evaluated before, and the result is available without re-execution. It is consistent with your DAG which shows that the next stage requires shuffling (reduceByKey). Whenever there is shuffling involved Spark automatically caches generated data

## DETAILS FOR JOB 4

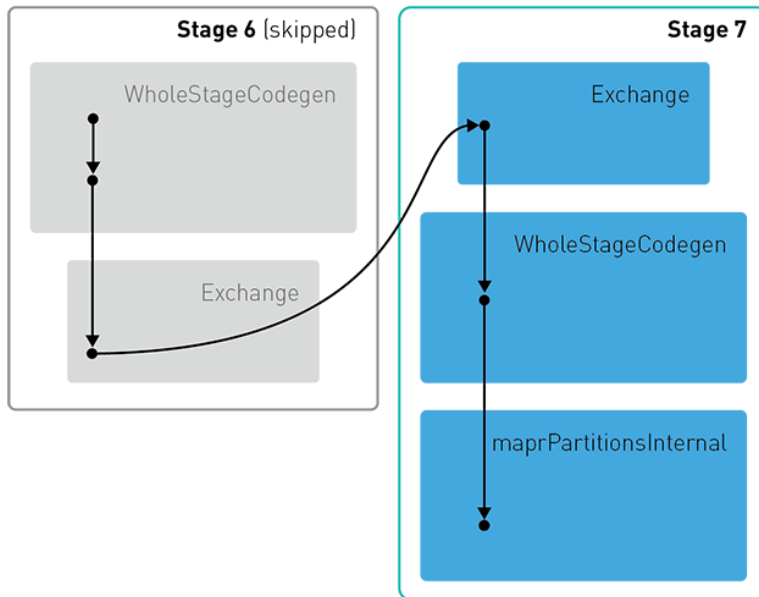
Status: SUCCEEDED

Completed Stages: 1

Skipped Stages: 1

► Event Timeline

▼ DAG Visualization



### Data Locality

Spark is a data parallel processing framework, which means it will execute tasks as close to where the data lives as possible (i.e. minimize data transfer).

### Checking Locality

The best means of checking whether a task ran locally is to inspect a given stage in the Spark UI. Notice from the screenshot below that the "Locality Level" column displays which locality a given task ran with.

#### Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
driver	192.168.153.147:35579	12 s	1000	0	1000	608.0 KB / 9999

#### Tasks (1000)

Page: 1 2 3 4 5 6 7 8 9 10 >

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

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	E
0	16	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	2 ms		621.0 B / 10	
1	17	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	3 ms		610.0 B / 10	
2	18	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	9 ms		638.0 B / 10	
3	19	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	15 ms		619.0 B / 10	
4	20	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	3 ms		561.0 B / 10	
5	21	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	6 ms		653.0 B / 10	
6	22	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	10 ms		637.0 B / 10	
7	23	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	3 ms		649.0 B / 10	
8	24	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2020/07/12 16:18:45	9 ms		601.0 B / 10	

You can adjust how long Spark will wait before it times out on each of the phases of data locality (data local --> process local --> node local --> rack local --> Any). For more information on these parameters, see the **spark.locality.\*** configs

**PROCESS\_LOCAL** // This task will be run within the same process as the source data

**NODE\_LOCAL** // This task will be run on the same machine as the source data but may be in a different process

**RACK\_LOCAL** // This task will be run in the same rack as the source data resides in different node

**NO\_PREF** (Shows up as ANY) // This task cannot be run on the same process as the source data or it doesn't matter

Property Name	Default	Meaning
spark.cores.max	(not set)	When running on a <a href="#">standalone deploy cluster</a> or a <a href="#">Mesos cluster in "coarse-grained" sharing mode</a> , the maximum amount of CPU cores to request for the application from across the cluster (not from each machine). If not set, the default will be spark.deploy.defaultCores on Spark's standalone cluster manager, or infinite (all available cores) on Mesos.
spark.locality.wait	3s	How long to wait to launch a data-local task before giving up and launching it on a less-local node. The same wait will be used to step through multiple locality levels (process-local, node-local, rack-local and then any). It is also possible to customize the waiting time for each level by setting spark.locality.wait.node, etc. You should increase this setting if your tasks are long and see poor locality, but the default usually works well.
spark.locality.wait.node	spark.locality.wait	Customize the locality wait for node locality. For example, you can set this to 0 to skip node locality and search immediately for rack locality (if your cluster has rack information).
spark.locality.wait.process	spark.locality.wait	Customize the locality wait for process locality. This affects tasks that attempt to access cached data in a particular executor process.
spark.locality.wait.rack	spark.locality.wait	Customize the locality wait for rack locality.

### Straggler Tasks (Long Running Tasks)

The straggler tasks can be identified in the Stages view and take a long time to complete. In this use case, the following are the straggler tasks that took longer time. how long for **garbage collection, serialization, deserialization and scheduling of stages/tasks**

#### RDD Implementation Straggler Task

5	<a href="#">map at FireServiceCallAnalysisSerialized.scala:60</a>	<a href="#">+details</a>	2017/06/14 11:51:56	3.4 min	12/12
4	<a href="#">collect at FireServiceCallAnalysisSerialized.scala:56</a>	<a href="#">+details</a>	2017/06/14 11:51:55	0.3 s	12/12
3	<a href="#">distinct at FireServiceCallAnalysisSerialized.scala:56</a>	<a href="#">+details</a>	2017/06/14 11:48:34	3.4 min	12/12
2	<a href="#">count at FireServiceCallAnalysisSerialized.scala:50</a>	<a href="#">+details</a>	2017/06/14 11:45:16	3.3 min	12/12

#### DataFrame Implementation Straggler Task

3	<a href="#">count at FireServiceCallAnalysisDF.scala:55</a>	<a href="#">+details</a>	2017/06/14 11:32:55	87 ms	1/1
2	<a href="#">count at FireServiceCallAnalysisDF.scala:55</a>	<a href="#">+details</a>	2017/06/14 11:31:45	1.2 min	12/12
1	<a href="#">show at FireServiceCallAnalysisDF.scala:52</a>	<a href="#">+details</a>	2017/06/14 11:31:45	79 ms	1/1
0	<a href="#">take at FireServiceCallAnalysisDF.scala:46</a>	<a href="#">+details</a>	2017/06/14 11:31:30	15 s	1/1

**The Environment tab** lists all the active properties of your Spark application environment. Use this page when you want to see which configuration flags are enabled. Only values specified through spark-defaults.conf, SparkSession, or the command line will be displayed here. For all other configuration properties, the default value is used.

← → ↺

127.0.0.1:4040/Environment/

🔍 ☆

spark

2.2.1-mapr-1803

Jobs

Stages

Storage

Environment

Executors

SQL

Spark shell application

Environment

Runtime Information

Name	Value
Java Home	/usr/lib/jvm/java-1.80...
Java Version	1.80_102 (Oracle Co...
Scala Version	version 2.11.8

Spark Properties

Name	Value
spark.app.id	local-1533071702543
spark.app.name	Spark shell
spark.driver.host	10.0.2.15
spark.driver.port	43211
spark.eventLog.dir	maprfs:///apps/spark

Under the Executors tab, you can see processing and storage for each executor:

- **Shuffle Read Write Columns:** shows size of data transferred between stages
- **Storage Memory Column:** shows the current used/available memory
- **Task Time Column:** shows task time/garbage collection time

Use this page to confirm that your application has the amount of resources you were expecting. You can look at the thread call stack by clicking on the thread dump link.

← → ↺

127.0.0.1:4040/Executors/

🔍 ☆

spark

2.2.1-mapr-1803

Jobs

Stages

Storage

Environment

Executors

SQL

Spark shell application

Executors

▶ Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Tasks Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(1)	0	0.0 B / 384.1 MB	0.0 B	2	0	0	204	204	9 s (0.2 s)	17.2 MB	0.0 B	607 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total (1)	0	0.0 B / 384.1 MB	0.0 B	2	0	0	204	204	9 s (0.2 s)	17.2 MB	0.0 B	607 B	0

Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Tasks Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	10.0.2.15:36610	Active	0	0.0 B / 384.1 MB	0.0 B	2	0	0	204	204	9 s (0.2 s)	17.2 MB	0.0 B	607 B	<a href="#">Thread Dump</a>

Showing 1 to 1 of 1 entries

### Spark History UI enablement:

```
val spark=org.apache.spark.sql.SparkSession.builder().appName("appName").master("local[*]").config("spark.history.fs.logDirectory","file:///tmp/spark-events").config("spark.eventLog.dir","file:///tmp/spark-events").config("spark.eventLog.enabled", "true").getOrCreate();
```

### Configuration Parameters

#### Resource Planning (Executors, Core, and Memory) (Ignore Tiny, Fat and go with Balanced of both)

A balanced number of executors, core, and memory will significantly improve the performance without any code changes in the Spark application while running on YARN.

### Spark Memory Allocation

Let's consider a 10 node cluster with following config and analyse different possibilities of executors-core-memory distribution:

#### **\*\*Cluster Config:\*\***

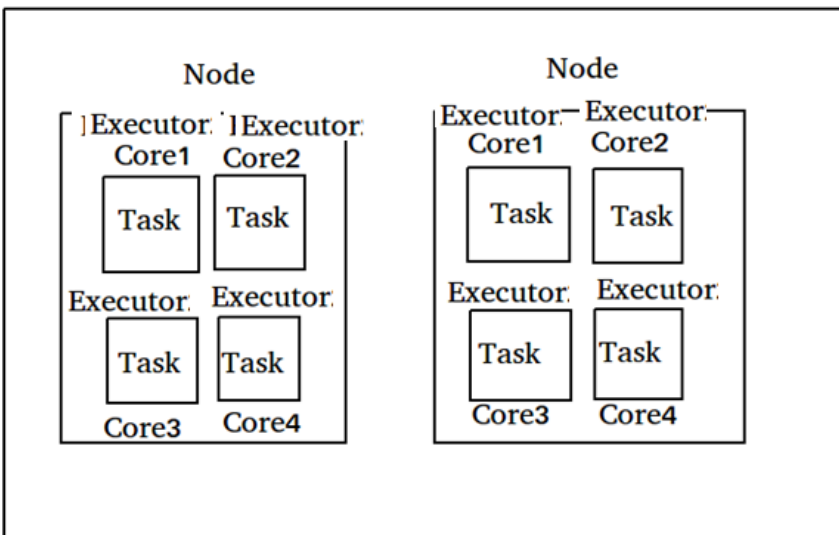
12 node ( 10 Worker/Data Nodes, 1 Master, 1 Client)

16 cores per Node

64GB RAM per Node

#### First Approach: Tiny executors [One Executor per core]:

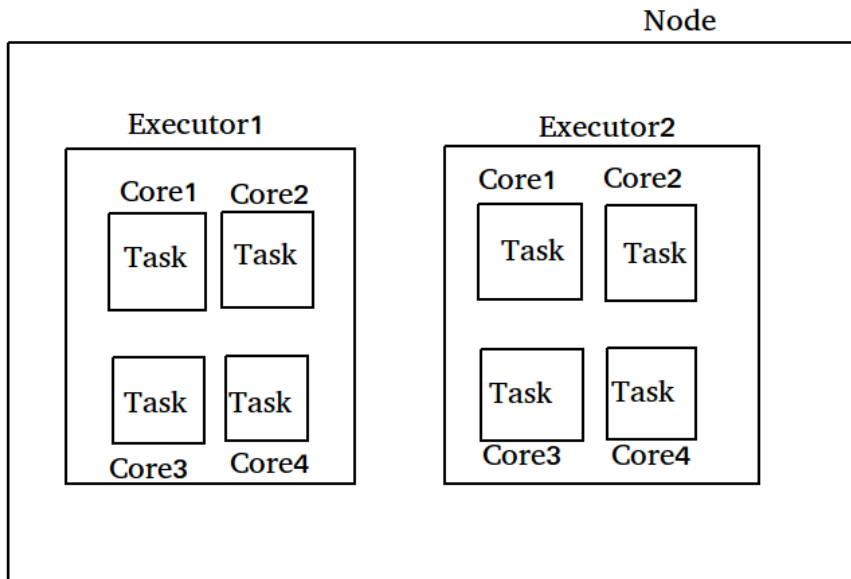
1. Multiple tasks in the same executor JVM can't be run, huge number of small JVMs start and kill overhead.
2. Shared/cached variables like broadcast variables and accumulators will be replicated in each executor of the nodes which is 16 times.
3. Not leaving enough memory overhead for Hadoop/Yarn daemon processes and not counting in ApplicationManager.





### Second Approach: Fat executors (One Executor per node)

1. With all 16 cores per executor,
2. ApplicationMaster and daemon processes are not counted
3. HDFS throughput will hurt and it'll result in excessive garbage results.



### Third Approach: Balance between Fat (vs) Tiny

With the cluster of 10 Nodes , 16 cores per Node, 64GB RAM per Node

Assign 5 cores per executors => `--executor-cores = 5` (for good HDFS throughput)

Leave 1 core per node for DN/Yarn daemons => Num cores available per node =  $16 - 1 = 15$

Total available of cores in cluster =  $15 \text{ cores} \times 10 \text{ nodes} = 150 \text{ cores}$

Number of available executors =  $(\text{total cores} / \text{num-cores-per-executor}) = 150 \text{ total cores} / 5 \text{ executor cores} = 30$  executors available

Leave 1 executor for ApplicationMaster =>  $30 - 1 = 29$  total executors

Number of executors per node => `--num-executors = 30` executors / 10 nodes = **3 executors per node**

Memory per executor =>  $64\text{GB per node} / 3 \text{ executors} = 21\text{GB per executor}$

Memory per executor after Counting off heap overhead = off heap allocation of ~7% of 21GB = 3GB, hence `--executor-memory = 21 - 3 = 18GB` Memory per executor

So, recommended config is: 29 executors, 18GB memory each and 5 cores each!!

Third approach has found right balance between Fat vs Tiny approaches.

Finally it achieved parallelism of a fat executor and best throughputs of a tiny executor!!

Leaving all the above calculations, iterate or benchmark for optimal/better performances considering ondemand/continuous other resources consumption in the cluster for other jobs at that time.

## Reference Urls:

[https://umbertogriffo.gitbooks.io/apache-spark-best-practices-and-tuning/content/which\\_storage\\_level\\_to\\_choose.html](https://umbertogriffo.gitbooks.io/apache-spark-best-practices-and-tuning/content/which_storage_level_to_choose.html)  
<https://html.developreference.com/article/11818128/Spark+Serialization%3A+How+Tungsten+and+Kryo+work+together%3F>  
<https://unraveldata.com/common-reasons-spark-applications-slow-fail-part-1/>  
<https://databricks.com/session/optimizing-apache-spark-sql-joins>  
<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-adaptive-query-execution.html>  
<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-debugging-query-execution.html>  
<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-catalyst.html>  
<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-catalyst-QueryPlan.html>  
<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-AstBuilder.html>  
<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-SubExprUtils.html>  
<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-CompressionCodecs.html>  
<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-performance-tuning-groupBy-aggregation.html>  
<https://medium.com/inspiredbrilliance/spark-optimization-techniques-a192e8f7d1e4>  
[https://medium.com/@\\_kostya\\_/how-to-optimize-apache-spark-45951466bde3](https://medium.com/@_kostya_/how-to-optimize-apache-spark-45951466bde3)  
<https://medium.com/teads-engineering/spark-performance-tuning-from-the-trenches-7cbde521cf60>  
<https://docs.cloudera.com/runtime/7.2.0/configuring-spark/topics/spark-dynamic-resource-allocation-properties.html>