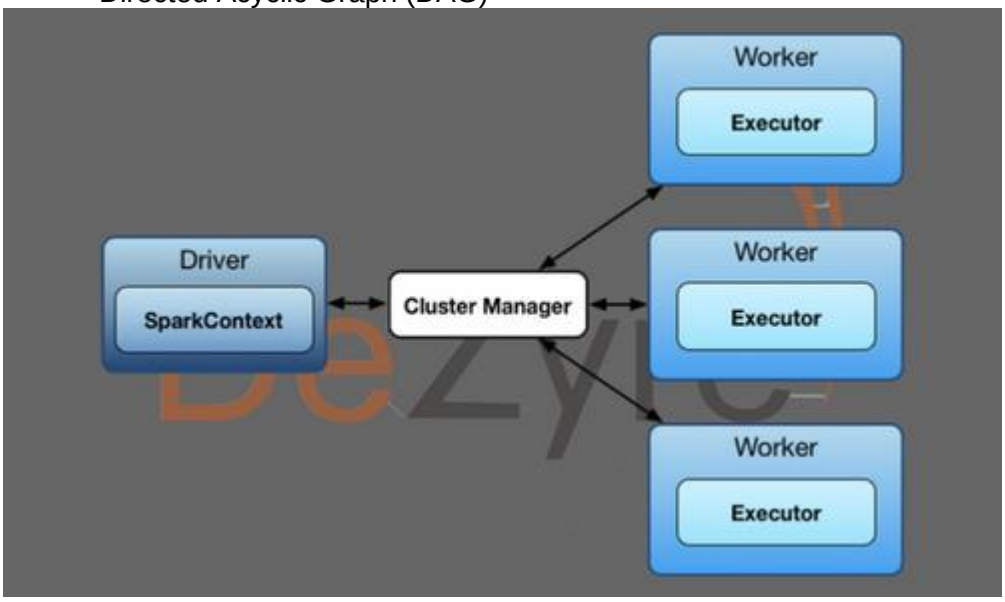
Data Engineering

Understanding Spark Architecture

Apache Spark has a well-defined and layered architecture where all the spark components and layers are loosely coupled and integrated with various extensions and libraries. Apache Spark Architecture is based on two main abstractions-

- Resilient Distributed Datasets (RDD)
- Directed Acyclic Graph (DAG)



Resilient Distributed Dataset (RDD)
Resilient distributed dataset (RDD) is a spark core data-structure which holds data. It can be considered as an abstraction layer which holds multiple pieces of partitioned data as a single dataset. It is resilient because it can be re-built from the lineage graph or DAG created during the transformation phase by spark engine. So it there is a load failure, spark engine can always refer the lineage graph to re-create RDD.
It is distributed as below the abstraction layer, the data actually sits in a distributed fashion across multiple clusters. The underlying data is partitioned and the partitions are spread across multiple machines. This gives each node fewer data to work with, thus resulting in overall faster operation.

Directed Acyclic Graph (DAG)
Direct - Transformation is an action which transitions data partition state from A to B.
Acyclic -Transformation cannot return to the older partition
DAG is a sequence of computations performed on data where each node is an RDD partition and edge is a transformation on top of data. The DAG abstraction helps eliminate the Hadoop MapReduce multistage execution model and provides performance enhancements over Hadoop.

Spark Architecture Overview
Apache Spark follows a master/slave architecture with two main daemons and a cluster manager –

1. Master Daemon – (Master/Driver Process)
2. Worker Daemon –(Slave Process)

A spark cluster has a single Master and any number of Slaves/Workers. The driver and the executors run their individual Java processes and users can run them on the same horizontal spark cluster or on separate machines i.e. in a vertical spark cluster or in mixed machine configuration.

Spark Driver – Master Node of a Spark Application
It is the central point and the entry point of the Spark Shell (Scala, Python, and R). The driver program runs the main () function of the application and is the place where the Spark Context is created. Spark Driver contains various components – DAGScheduler, TaskScheduler, BackendScheduler and BlockManager responsible for the translation of spark user code into actual spark jobs executed on the cluster.

- The driver program that runs on the master node of the spark cluster schedules the job execution and negotiates with the cluster manager.
- It translates the RDD's into the execution graph and splits the graph into multiple stages.
- Driver stores the metadata about all the Resilient Distributed Databases and their partitions.
- Cockpits of Jobs and Tasks Execution -Driver program converts a user application into smaller execution units known as tasks. Tasks are then executed by the executors i.e. the worker processes which run individual tasks.
- Driver exposes the information about the running spark application through a Web UI at port 4040.

Spark Executor
Executor is a distributed agent responsible for the execution of tasks. Every spark applications has its own executor process. Executors usually run for the entire lifetime of a Spark application and this phenomenon is known as "Static Allocation of Executors". However, users can also opt for dynamic allocations of executors wherein they can add or remove spark executors dynamically to match with the overall workload.

- Executor performs all the data processing.
- Reads from and Writes data to external sources.
- Executor stores the computation results data in-memory, cache or on hard disk drives.
- Interacts with the storage systems.

Cluster Manager
An external service responsible for acquiring resources on the spark cluster and allocating them to a spark job. There are 3 different types of cluster managers a Spark application can leverage for the allocation and deallocation of various physical resources such as memory for client spark jobs, CPU memory, etc. Hadoop YARN, Apache Mesos or the simple standalone spark cluster manager either of them can be launched on-premise or in the cloud for a spark application to run.
Choosing a cluster manager for any spark application depends on the goals of the application because all cluster managers provide different set of scheduling capabilities. To get started with apache spark, the standalone cluster manager is the easiest one to use when developing a new spark application.

What happens when Spark Job is submitted?
When a client submits a spark user application code, the driver implicitly converts the code containing transformations and actions into a logical directed acyclic graph (DAG). At this stage, the driver program also performs certain optimizations like pipelining transformations and then it converts the logical DAG into physical execution plan with set of stages. After creating the physical execution plan, it creates small physical execution units referred to as tasks under each stage. Then tasks are bundled to be sent to the Spark Cluster.

The driver program then talks to the cluster manager and negotiates for resources. The cluster manager then launches executors on the worker nodes on behalf of the driver. At this point the driver sends tasks to the cluster manager based on data placement. Before executors begin execution, they register themselves with the driver program so that the driver has holistic view of all the executors. Now executors start executing the various tasks assigned by the driver program. At any point of time when the spark application is running, the driver program will monitor the set of executors that run. Driver program in the spark architecture also schedules future tasks based on data placement by tracking the location of cached data. When driver programs main () method exits or when it call the stop () method of the Spark Context, it will terminate all the executors and release the resources from the cluster manager.

The structure of a Spark program at higher level is - RDD's are created from the input data and new RDD's are derived from the existing RDD's using different transformations, after which an action is performed on the data. In any spark program, the DAG operations are created by default and whenever the driver runs the Spark DAG will be converted into a physical execution plan.

Dataframes
Spark dataframes are distributed collection of data in rows and columns. It is the structured representation of data similar like a table. We can say dataframes are relational database tables with better optimization.

Persistence
All spark computations are presented as lineage graphs. Different parts of job computations are connected by dependencies and computations are lazy. To trigger a computation, you have to call an action on RDD or dataframe, for example: count. This makes Spark start computation from the beginning of the graph and compute the result step by step, partition by partition. The point here is that if you have the same lineage and you call another action collect(), spark will re-compute the whole graph even though nothing has changed. And this thing can be really bad for several types of computations, for example, if you have an iterative algorithm which loops through stable pre-computed data like machine-learning algorithm. That's why Spark provides you with a method to actually store intermediate results of a computation, and this is called persist.

RDD and dataframe has a persist function which actually allows you the option to safe keep/store intermediate transformations applied on a dataframe/RDD in either memory or disk IO and this is called persist. When you persist, you have to choose where to persist the data. There are three kinds of storage levels available in pi Spark. The preferable one is MEMORY_ONLY, your executor has some reserved memory for persisting and when you choose MEMORY_ONLY, Spark tries to save partitions in this part of memory if there is enough space. There is also a MEMORY_ONLY_2 option, which provides a bit of redundancy by storing two replicas on different cluster nodes. The second level is MEMORY_AND_DISK. Basically, Spark tries to store as much of data as possible in memory and if there is not enough space, it will spill records to disk. This, however, introduces performance considerations because disks IO is pretty slow.

So, what do you have to keep in mind when using persistence? Spark is an in-memory framework and you have to keep it so for the best performance. That's why store your partitions in memory if they fit. And even if it does not fit, you should probably think about increasing the memory of your executors.

Checkpoints

And finally, if you use some kind of iterative algorithm like machine-learning algorithm in case of all specialization, you should definitely think of persisting the results of intermediate computations like training dataset. Okay, as I have already mentioned, when you persist, intermediate results are stored in the memory of an executor or in disk of an executor. So actually, if an executor fails, Spark has to re-compute partitions of the particular executor from scratch, but your cluster can be really crowded by different jobs and users and the resource manager will not allocate new containers for this re-computation. This situation may dramatically increase the running time of your job. That's why Spark provides you with the mechanism of a reliable persistence called checkpointing. When you checkpoint, Spark writes partitions of an RDD or a dataframe to a reliable storage and this storage is HDFS. And another interesting thing with checkpointing is that it also turnkeys the whole lineage graph which produced this intermediate result. To make a checkpoint, you need to specify a directory in HDFS where to store the data and then call checkpoint method of an RDD. And the caveat here is that you can't checkpoint a dataframe. For a dataframe, you have to checkpoint the underlying RDD. So, want to make a checkpoint? You should definitely think about checkpointing in a noisy cluster. Cluster is called noisy if there are lots of jobs and users which compete for resources and there are not enough resources to run all the jobs simultaneously. You must think about checkpointing if your computations are really expensive and take long time to finish because it could be faster to write an RDD to HDFS and read it back in parallel than recompute from scratch. And there's a slight inconvenience; there is no way to checkpoint a dataframe so you have to checkpoint the underlying RDD. So, to sum up, in this video, you learned that Spark provides ways to store intermediate results by persisting or checkpointing. It helps to speed up several kinds of workloads like iterative algorithms in general and machine-learning algorithms in particular. You have learned that persisting is unreliable. If an executor fails, Spark has to recompute its partitions and

checkpointing is reliable because it stores data in a reliable HDFS. Now, you also know that checkpointing is slow because you have to write data to HDFS, and persisting may be slow if you use not only the memory, but the disks for persisting, as well.

reduceByKey() vs groupByKey()

The reduceByKey() performs better than groupByKey() because the first one combines the data in each of the respective machines before the data is shuffeled and then submit the data for the reduce operation due to which there are less unnecessary traffic over the network.
The groupByKey() on the other hand gives all the data for shuffling at the same time due to which there is more traffic over the network and hence slower. Infact, reduceByKey is the most efficient way of aggregating data using Spark.

Partitioning

Spark's partitioning is available on all RDDs of key/value pairs, and causes the system to group elements based on a function of each key. Although Spark does not give explicit control of which worker node each key goes to (partly because the system is designed to work even if specific nodes fail), it lets the program ensure that a set of keys will appear together on some node. For example, you might choose to hashpartition an RDD into 100 partitions so that keys that have the same hash value modulo 100 appear on the same node. Or you might range-partition the RDD into sorted ranges of keys so that elements with keys in the same range appear on the same node.

The processNewLogs() method can remain unchanged: the events RDD is local to processNewLogs(), and is used only once within this method, so there is no advantage in specifying a partitioner for events. Because we called partitionBy() when building userData, Spark will now know that it is hash-partitioned, and calls to join() on it will take advantage of this information. In particular, when we call user Data.join(events), Spark will shuffle only the events RDD, sending events with each particular UserID to the machine that contains the corresponding hash partition of userData (see Figure 4-5). The result is that a lot less data is communicated over the network, and the program runs significantly faster.

Note that partitionBy() is a transformation, so it always returns a new RDD—it does not change the original RDD in place. RDDs can never be modified once created. Therefore it is important to persist and save as userData the result of partitionBy(), not the original sequenceFile(). Also, the 100 passed to partitionBy() represents the number of partitions, which will control how many parallel tasks perform further operations on the RDD (e.g., joins); in general, make this at least as large as the number of cores in your cluster.

```
import org.apache.spark.HashPartitioner
sc= SparkContext()
basedata= sc.sequenceFile ("...")
userdata= basedata.partitionBy (new HashPartitioner(8)).persist()
```

Spark Coalesce against Repartition

Spark splits data into partitions and execute computations on the split data in parallel. You should understand how data is partitioned and when you need to manually adjust the partitioning to keep your Spark computation running properly.

Coalesce method reduces the number of partitions in a Dataframe. When the function is called, it uses the existing partitions to append more data to it by releasing certain no. of partitions. There are no **shuffling** involved during the process. **For ex:** if there are 100 partitions with 4 records in each partition, then on coalesce(50), the partition reduces to 50 where the initial 50 partitions are not disturbed but records from the remaining 50 are appended to these first 50 partitions.

Repartition on the other hand can be used to either increase or reduce the number of partitions in a RDD. During repartition the data is completely shuffled and then distributed among partitions. Repartitions create almost equal sized partitions.

Coalesce operation may run faster than Repartition but working with unequal sized partitions are generally slower to work with than equal sized partitions. You usually need to repartition the dataset if it's very large as spark is built to work faster with equal sized partitions.

```
scala> val x= (1 to 10).toList
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val num_rdd= sc.parallelize(x)
num_rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at parallelize at <console>:29

scala> num_rdd.partitions.size
res21: Int = 2

scala> num_rdd.coalesce(1).partitions.size
res22: Int = 1

scala> num_rdd.coalesce(4).partitions.size
res23: Int = 2

scala> num_rdd.repartition(3).partitions.size
res24: Int = 3
```

Spark Checkpoint

Checkpoint is actually a feature of Spark Core (that Spark SQL uses for distributed computations) that allows a driver to be restarted on failure with previously computed state of a distributed computation described as an RDD. That has been successfully used in Spark Streaming - the now-obsolete Spark module for stream processing based on RDD API.
Checkpoint **truncates** the lineage of a RDD to be checkpointed. That has been used successfully in Spark MLLIB in iterative machine learning algorithms.
Checkpoint can be eager or lazy per eager flag of checkpoint operator. **Eager checkpoint** is the default checkpoint and happens immediately when requested. **Lazy checkpoint** does not and will only happen when an action is executed.

Using Dataset checkpoint requires that you specify the **checkpoint directory**. The directory stores the checkpoint files for RDDs to be checkpoint. Use SparkContext.setCheckpointDir to set the path to a checkpoint directory.

For ex:

```
val checkpointDir= "/user/fgb_dip/sumx_data/checkpoints"
sc.setCheckpointDir(checkpointDir)

val nums= sc.parallelize((1 to 5).toList).toDF("id").withColumn("random",rand()).filter($"random"> 0.2)
nums.show()

println(nums.queryExecution.toRdd.toDebugString)

nums.rdd.checkpoint

nums.rdd.isCheckpointed
```

nums.rdd.count *(perform an action on the base RDD to get the data checkpointed)*

nums.rdd.isCheckpointed

```
scala> val checkpointDir= "/user/fgb_dip/sumx_data/checkpoints"
checkpointDir: String = /user/fgb_dip/sumx_data/checkpoints

scala> sc.setCheckpointDir(checkpointDir)
19/12/05 16:52:00 WARN spark.SparkContext: Checkpoint directory must be non-local if Spark is running on a cluster: /user/fgb_dip/sumx_data/checkpoints

scala> val nums= sc.parallelize((1 to 5).toList).toDF("id").withColumn("random",rand()).filter($"random"> 0.2)
nums: org.apache.spark.sql.DataFrame = [id: int, random: double]

scala> nums.show()
+---+------------------+
| id|            random|
+---+------------------+
|  1|0.7431970107656017|
|  2|0.2267652563991961|
|  3|0.7629104039649968|
|  4|0.8178629769673094|
|  5|0.6414742200666248|
+---+------------------+


scala> println(nums.queryExecution.toRdd.toDebugString)
(2) MapPartitionsRDD[91] at toRdd at <console>:31 []
 |  MapPartitionsRDD[90] at toRdd at <console>:31 []
 |  MapPartitionsRDD[85] at intRddToDataFrameHolder at <console>:28 []
 |  ParallelCollectionRDD[84] at parallelize at <console>:28 []

scala> nums.rdd.checkpoint

scala> nums.rdd.isCheckpointed
res68: Boolean = false
```

```
scala> nums.rdd.count()
res97: Long = 5

scala> nums.rdd.isCheckpointed
res98: Boolean = true
```

```
[fgb_dip@bda1node04 ~]$ hadoop fs -ls /user/fgb_dip/sumx_data/checkpoints/501485cb-96ac-4674-894e-a6b40d5afcec
Found 1 items
drwxr-xr-x   - fgb_dip hadoop          0 2019-12-05 17:31 /user/fgb_dip/sumx_data/checkpoints/501485cb-96ac-4674-894e-a6b40d5afcec/rdd-92
```

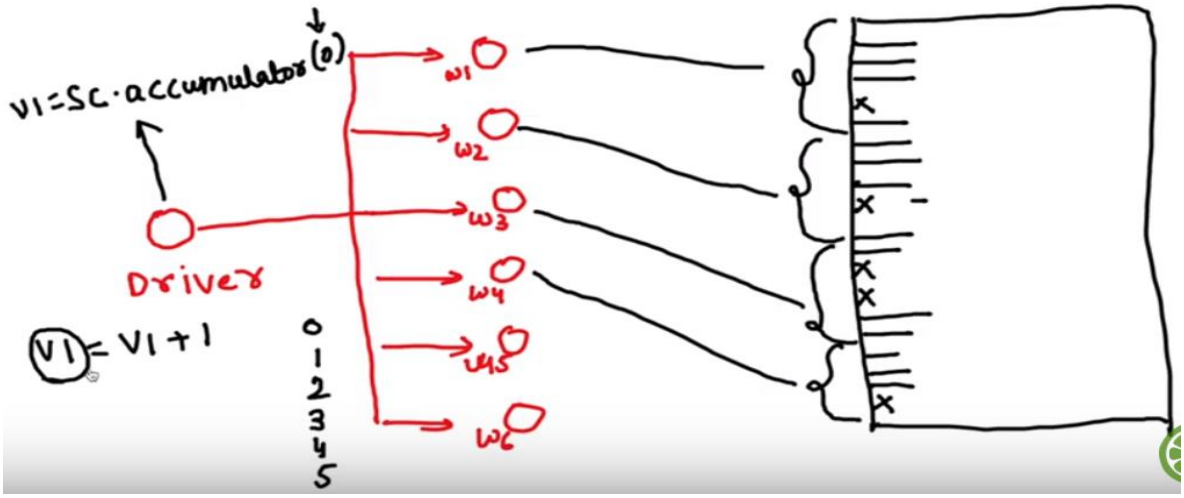
https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-checkpointing.html

Accumulator VS Broadcast Variables

Accumulator variables are such variables which can be used by your worker nodes for writing some data. If you want to maintain some state or you want to update some data point then you can use accumulators which are available across all worker nodes.
Broadcast variables are used by all the worker nodes for reading the data.
These are basically two types of shared variables used across the worker nodes. One for writing and the other one for reading.

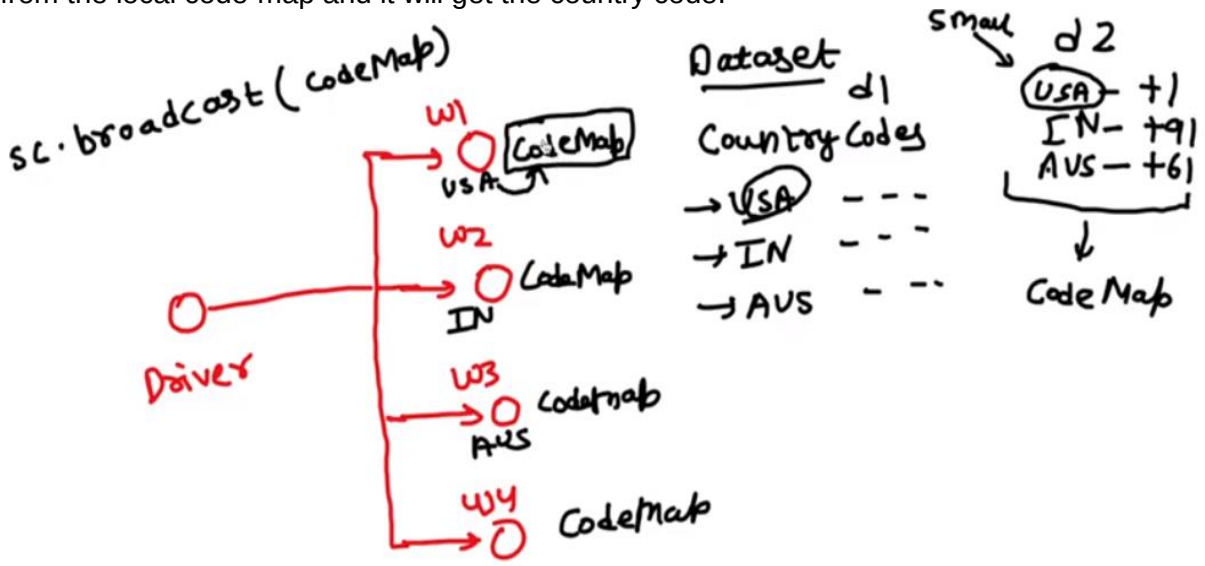
How accumulator variable works?

In the below diagram, all the worker nodes are processing different parts of a large text file and each part being processed consists of lines along with a space (x) in between. Whenever the worker node reads a space, they increase the count of the accumulator variable by one. Thus all the worker nodes update the variable count and by the end of the processing the accumulator variable has a total count of spaces. The workers can't read this variable but the driver program can. The updated variable will be available within the driver program.

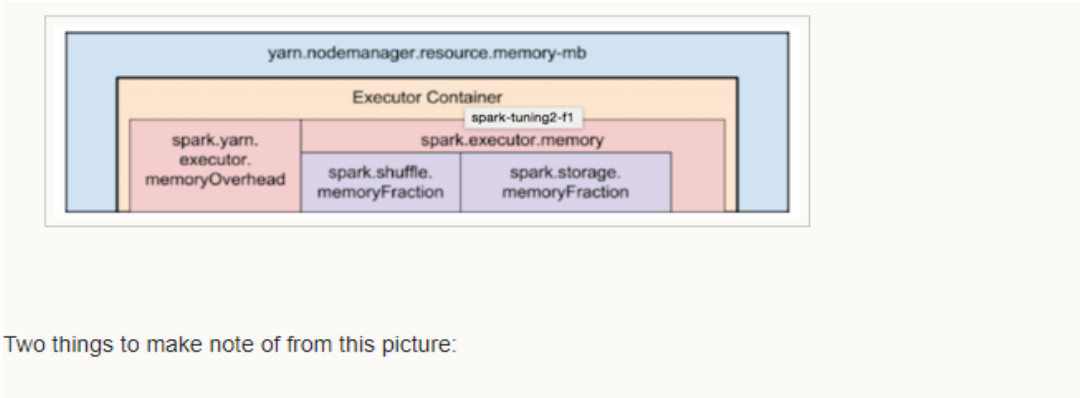


How broadcast variable works?

Let say you have some dataset and this dataset is being processed by the worker nodes. Let's assume dataset contains a field called country codes. Now you want to join this dataset with another dataset to fetch the mobile country code for that particular country code. But this is not the efficient way to do as every time a worker node joins the dataset with the lookup dataset there will be shuffling. Instead we can broadcast this map variable consisting of (country code → mobile country code) into a broadcast variable. Doing this will make this map variable available locally across all the worker nodes. This time when the worker nodes joins with the smaller dataset there is no shuffling required as the whole dataset is available locally for use. Spark will do a lookup from the local code-map and it will get the country code.



How to specify optimum no. of executor cores and executor memory for a Spark Job execution?



Two things to make note of from this picture:

```
Full memory requested to yarn per executor =
         spark-executor-memory + spark.yarn.executor.memoryOverhead.
spark.yarn.executor.memoryOverhead =
               Max(384MB, 7% of spark.executor-memory)
```

Let's assume we have Hadoop cluster with the below configuration:
10 nodes
16 cores per node
64 GB RAM per node

TOTAL CORES= 10 * 16 = 160

Approach 1 TINY executors:
If we allocate 1 core for 1 executor, total no of executors= 160,
Then total memory for each executor= 640/160= 4GB

But 4GB of memory for 1 executor is less, and should be avoided. NOT GOOD

Approach 2 FAT executors:
FAT executors generally means 1 executor per node. Thus, in this case,
10 nodes = 10 executors
1 executor = 160/10= 16 cores
1 executor memory = 640/10= 64GB

Thus, with FAT executors, we have each executor with 16 cores and 64GB of ram

Approach 3 Intermediate approach (balance between fat and tiny):
1. Let's assign 5 cores to each executors, ideal for HDFS throughput.
2. Leave 1 core from each node for yarn/hadoop deamons. So, total cores available per node= 16-1= 15
3. Total cores available= 15*10= 150
4. Total no of executors= 150 cores/5 cores = 30 executors
5. Leave 1 executor out of 30 executors for application manager. So, --num-executors= 29
6. Number of executors per node= 30/10= 3
7. Memory per executor= 64/3= 21GB
8. Counting off heap overhead= 7% of 21GB = 3GB. So, actual memory: --executor-memory= 21-3= 18GB

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

So, --num-executors= 29 --executor-memory= 18GB

Spark Important Concepts – Highlights

→ The executor will process a partition and not the individual files. The more the partition you have the more the parallel execution from spark side and the processing speed will thereby increase accordingly.

→ Ideally when executors are used by a spark job to execute the various transformation and actions they are in use and adds to the speed of the data processing in Spark. But, once the job gets over, these executors are removed by YARN depending on an idle time limit. But, in case you want this to be more effective, you can use the below configuration:
set spark.dynamicAllocation.enabled = true

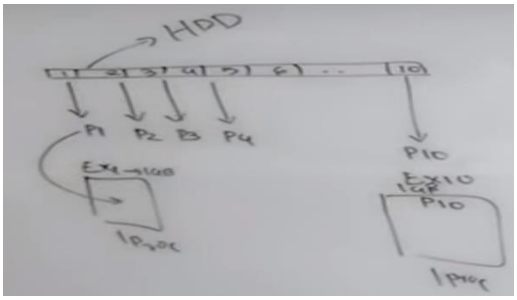
This will allow the YARN to allocate or scales the number of executors up and down depending on the number of executors required for the processing.

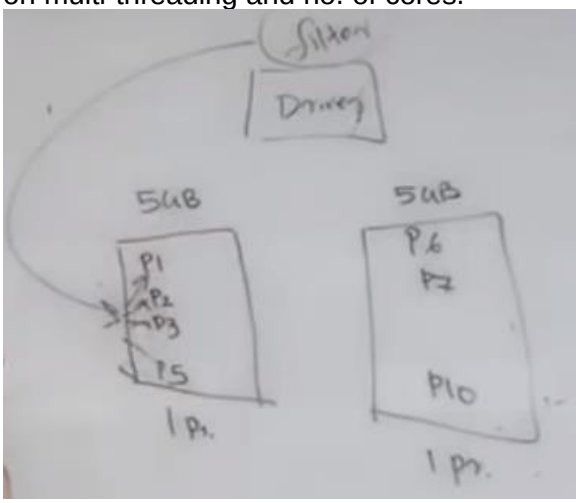
→ An executor is a container i.e. a JVM. One processor core is required to manage the container.

→ For a file of size x if the no. of partitions are more, the no. of splits are more. If I want 10 partitions and for each partition if you can assign one executor, then you get parallelism. If you get less than 10, say 5 then also you get parallelism, but it's a little bit slow than previous one.

Steps inside a Spark Program

1. First step is to create an RDD while the spark driver execute the first piece of the main program.
2. Let's say you have 6 nodes in Hadoop cluster and you have requested for 4 executors. Now how yarn will allocate the executors?
3. In a typical map reduce scenario, if the data is residing in Hadoop in 4 nodes, and each of these 4 nodes will have a JVM allocated to it when you call for processing the data. Then the data moves from HDFS inside the JVM for getting processed. The map function will specify 4 mappers to process 4 blocks of data. **1 mapper can't process 4 blocks. It can process only 1 block of data at a time. No. of mappers= No. of blocks (or input splits)**
   We generally don't change the number of mappers manually and let the map reduce code decide as it might affect the performance. But, for spark it is different. Here, you can specify the number of containers and their size also.
4. In spark, you define the number of executors. If you specify 1 executor, all these 4 blocks will come and sit inside one executor and gets processed. Now, if you want parallelism you can specify 4 executors, and then any 4 out of these 6 nodes are chosen at random and if the actual data is residing at some other node gets copied to the processing node HDFS.
5. Now, the data gets copied from the block to the executor memory i.e. ram memory. So, you have the data in RAM memory of 4 nodes/machines. This is called an RDD.
6. This is a partitioned data as we have blocks copied to 4 different executors. If your data is in HDFS and you are reading it's already partitioned but what if you are reading from a local file system. Then you can specify partitions to spark to use parallelism. **An executor will read the partitions and not the file as a whole**. It can be a case you have all the partitions into a single executor but since the data is partitioned the executor will process it fast.
7. For a file of size x if the no. of partitions are more, the no. of splits are more. If I want 10 partitions and for each partition if you can assign one executor, then you get parallelism. If you get less than 10, say 5 then also you get parallelism, but it's a little bit slow than previous one. **If you have enough resources, parallelism works better**.

8. In below diagram, each partition sits under each executor and data is processed parallel. Here, each executor has 1GB ram and 1 Processor Core.


9. Suppose you have only 2 executors and each of them have 5GB ram, then above distribution will be as below. Now, here 5 partitions will sit on two executors and if the no. of cores are more, say 4, then parallelism will be enhanced and each partition will be executed by the executors faster. The filter operation on each executor can function parallel on the 5 partitions (p1 to p5) or one by one depending on multi-threading and no. of cores.


10. Suppose you ask for executors of 10GB ram each. By default you get only 9BG because 10% is allocated for system calls. From 9GB you get around 60%. **So overall you get around 54% of the requested memory for each executors**. This is because your executor will run inside a container which is a JVM. It requires memory for garbage collection, JVM Management, etc. So, if you ask for 10GB container, you get around 5.5GB for RDD. Rest all system will take.

File Formats and which one to use

File Formats: https://blog.clairvoyantsoft.com/big-data-file-formats-3fb659903271

We generally have three different formats for writing files at a broad level:
1. Standard File Formats
2. Serialization Formats
3. Columnar Formats

A huge bottleneck for HDFS enabled applications like Map-Reduce and Spark is the time it takes to locate and read the file in a different location and time it takes to write it back to another location. The issues get complicated with the difficulties managing large datasets such as evolving schemas or storage constraints.

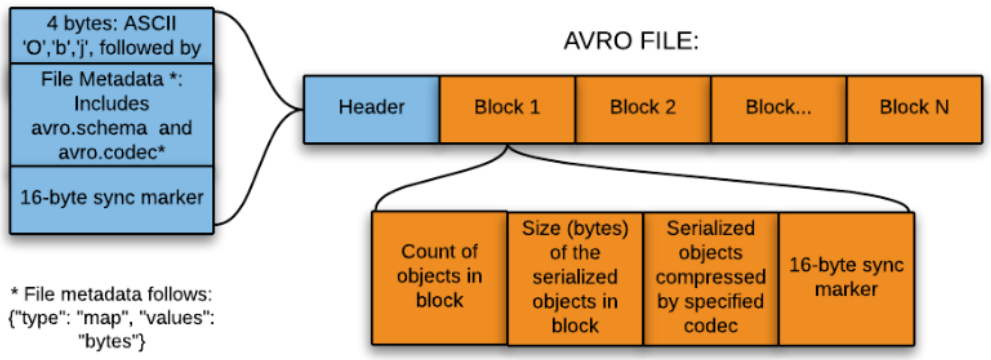
Choosing an appropriate file format can have some significant benefits:
1. Faster Read times
2. Faster Write times
3. Split files
4. Schema evolution support
5. Advanced compression support

AVRO File format

AVRO is a row based storage format for Hadoop which is highly used as a serialization platform. AVRO stores the data in JSON format making it easy to read and interpret by any program. The data itself is stored in binary format making it compact and efficient.

Avro is a language neutral data serialization system. It can be processed by multiple languages (currently C, C#, C++, Java, Python and Ruby). A key feature of Avro is the robust support for data schemas that change over time i.e. schema evolution. **Avro handles schema changes like missing fields, added fields and changed field.**

Avro provides rich data structures. For example: you create a record that contains an array, an enumerated type and a sub type.



This format is the ideal candidate for storing data in a data lake landing zone because:

1. Data from a landing zone is usually read as a whole for further processing by downstream systems (the row based format is more efficient in this case)
2. Downstream systems can easily retrieve table schemas from files (there is no need to store the schemas separately in an external meta store)
3. Any source schema change is easily handled

PARQUET File Format

Parquet, an open source file format in Hadoop, stores nested data structures in a **flat columnar format**. Compared to traditional approach where data is stored in row-oriented approach, parquet is more efficient in terms of storage and performance. Here values of each column are organized so that they are all adjacent, enabling better compression.
It is especially good for queries which read particular columns from a wide (with many columns) table since only needed columns are read and IO optimized.
What is a columnar storage format? → In a columnar-storage format values for each column of the same type in the records are stored together. For example, if there is a record which comprises of ID, EMP Name and Department then all the values for ID column will be stored together, values for Name will be stored together and so on.

ID	EMP Name	Department
1	Emp1	D1
2	Emp2	D2
3	Emp3	D3

For the table in a row wise storage format data will be stored as below:

1	Emp1	D1	2	Emp2	D2	3	Emp3	D3

Whereas the same data will be stored as follows in a Column oriented storage format:

1	2	3	Emp1	Emp2	Emp3	D1	D2	D3

If you need to query few columns from the table then columnar storage format is more efficient as it will read only required columns as they are adjacent thus minimizing IO. For example, let's say you want only the NAME column. In a row storage format, each record in the dataset has to be loaded, parsed into fields and then data for Name is extracted. With column-oriented format it can directly go to the Name column as all the values for that columns are stored together and get those values. No need to go through the whole record.

So, column-oriented format increases the query performance as less seek time is required to go to the required columns and less IO is required as it needs to read only the columns whose data is required. One of the unique features of Parquet is that it can store data with nested structures also in columnar fashion which means in Parquet file format even the nested fields can be read individually without the need to read all the fields in the nested structure.

https://www.youtube.com/watch?v=rVC9F1y38oU

A parquet file consists of three important properties:

1. **Block Size** – This means a block in HDFS and the meaning is unchanged for describing this file format. The file format is designed to work well on top of HDFS. The block size is the size of a row group being buffered in memory. This limits the memory usage when writing. Larger values will improve the I/O when reading but consume more memory when writing. Default size is 134217728 bytes (= 128 * 1024 * 1024)

2. **Row Group** – A logical horizontal partitioning of the data into rows. There is no physical structure that is guaranteed for a row group. A row group consists of a column chunk for each column in the dataset.

3. **Column Chunk** – A chunk of the data for a particular column.

4. **Page** – Column chunks are divided up into pages. The page size is for compression. When reading, each page can be decompressed independently. A block is composed of pages. The page is the smallest unit that must be read fully to access a single record. If this value is too small, the compression will deteriorate. Default size is 1048576 bytes (= 1 * 1024 * 1024).

Benefits of Parquet Format:

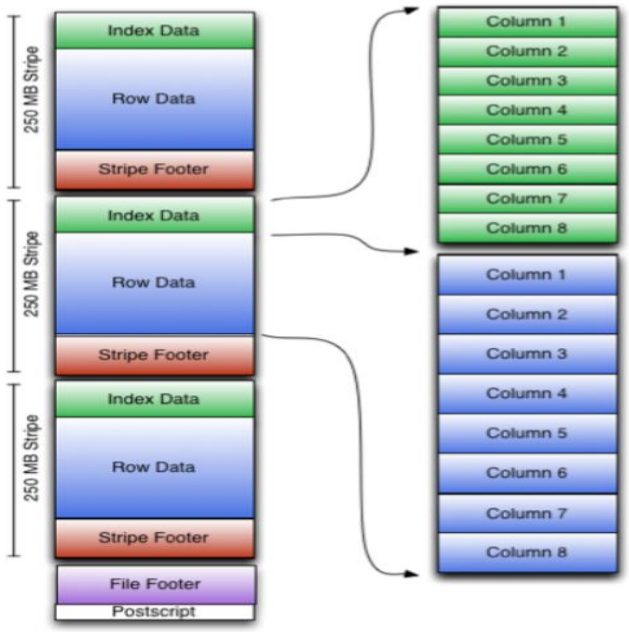
1. Efficient storage – Organizing by columns allows for better compression.
2. Efficient data IO and CPU Utilization – IO will be reduced as we efficiently scan only a subset of columns while reading the data. Better compression also reduces the bandwidth required to read the data.
3. Interoperability – As we store data of the same type for a particular column, we can use encoding better suited to the modern processors. It is understood by almost across the entire Hadoop ecosystem.

ORC File Format

The Optimized Row Columnar (ORC) file format provides a highly efficient way to store data. It was designed to overcome limitations of the other file formats. It ideally stores data compact and enables skipping over irrelevant parts without the need for large, complex or manually maintained indices. The ORC file format addresses all of these issues.

ORC File formats has many advantages:

1. A single file as the output of each task, which reduces Name Node's load
2. Hive type support including date-time, decimal and the complex types (struct, list, map and union)
3. Concurrent read of the same file using separate record readers
4. Ability to split files without scanning for markers
5. Bound the amount of memory needed for reading or writing
6. Metadata stored using Protocol Buffers, which allows the addition and removal of fields



ORC stores collections of rows in one file and within the collection the row data is stored in a columnar format.

An ORC file contains groups of row data called **stripes**, along with auxiliary information in a file **footer**. At the end of the file a **postscript** holds compression parameters and the size of the compressed footer.
The default stripe size is **250 MB**. Large stripe sizes enable large, efficient reads from HDFS.

The file footer contains a list of stripes in the file, the number of rows per stripe, and each column's data type. It also contains column-level aggregates count, min, max, and sum.

Stripe Footer contains a directory of stream location.
Row Data is used in table scans
Index Data includes min and max values for each column and the row position within each element. ORC index are used for the selection of stripes and row groups and not for answering queries.

AVRO v/s PARQUET

1. Avro is a row-based storage format whereas PARQUET is a columnar-based storage format.
2. PARQUET is much better for analytical querying i.e. reads and query are much more efficient than writing.
3. Write operations in AVRO are better than PARQUET

4. AVRO is much matured than PARQUET when it comes to schema evolution. PARQUET only supports schema append whereas AVRO supports much featured schema evolution i.e. adding or modifying columns.
5. PARQUET is ideal for querying subset of columns in a multi column table. AVRO is ideal in case of ETL operations where we need to query all the columns.

ORC v/s PARQUET

1. PARQUET is more capable of storing nested data.
2. ORC is more capable of Predicate Pushdown.
3. ORC supports ACID properties.
4. ORC is more compression efficient.

Storage Strategies

Before talking about the parquet side of the equation, one thing to consider is how the data will be used after you save it to parquet. If it's going to be read/processed often, you may want to consider what the access patterns are and decide to partition it accordingly. One common pattern is partitioning by date, because most of our queries have a time range. Partitioning your data appropriately will have a much bigger impact on performance on using that data after it is written.

Now, onto Parquet, the rule of thumb is for the parquet block size to be roughly the size of the underlying file system. That matters when you're using HDFS, but it doesn't matter much when you're using S3.

Again, the consideration for the Parquet block size, is how you're reading the data. Since a Parquet block has to be basically reconstructed in memory, the larger it is, the more memory is needed downstream. You also will need fewer workers, so if you're downstream workers have plenty of memory you can have larger parquet blocks as it will be slightly more efficient.

However, for better scalability, it's usually better having several smaller objects - especially according to some partitioning scheme - versus one large object, which may act as a performance bottleneck, depending on your use case.

To sum it up:

- a larger parquet block size means slightly smaller file size (since compression works better on large files) but larger memory footprint when serializing/deserializing
- the optimal file size depends on your setup
- if you store 30GB with 512MB parquet block size, since Parquet is a splittable file system and spark relies on HDFS getSplits() the first step in your spark job will have 60 tasks. They will use byte-range fetches to get different parts of the same S3 object in parallel. However, you'll get better performance if you break it down in several smaller (preferably partitioned) S3 objects, since they can be written in parallel (one large file has to be written sequentially) and also most likely have better reading performance when accessed by a large number of readers.

Choosing the right compression algorithm

The general human consumable/readable text formats like XML, CSV, TSV, JSON, TXT etc. are not only uncompressed but they are not stored as binary as well. So this consumes a lot of unnecessary space. When you ingest large amount of these files in Hadoop the benefit is nullified as it should be machine readable.

All compression algorithm does a **space/time** trade-off. Faster compression/decompression speed comes at the expense of space saving. For ex: an algorithm build for speed will not compress a file as much as an algorithm build to produce high compression ratio. When you are using compression algorithms to compress data, you must use computing resources and time that could be used for analytics or processing. If your resources are tied up with compression your productivity will drop until compression is complete. To avoid this loss of power and time, consider adding coprocessors to your system.

When compressing your data you can choose between **lossless** or **lossy** methods. Lossless compression preserves all data by replacing duplicated data with variables or references to the first instance of the data in a file.
Lossy compression eliminates data, creating a rough approximation of what the data originally was. Lossless compression is typically used for databases, text documents and other discrete data. Lossy compression is typically used for images, audio or video.
Lossless compression ensures that all data can be retrieved at decompression, it also takes up more storage space than lossy. To accommodate this difference, you can consider using both methods of compression, depending on your data type.

Select your codec carefully
Codec is a short form of compressor/ decompressor. It refers to software, hardware or a combination of both. You use codec to apply compression/ decompression algorithm on data. The type of codec you are able to use depends on the data and file type you are trying to compress. It also depends on whether you need your compressed file to be splittable. Splittable files can be processed in parallel by different processors.

GZIP – provides lossless compression that is not splittable. It is often used for HTTP compression. Gzip compression ratio is around 2.7x-3x. Compression speed is between 100MB/s and decompression speed is around 440MB/s.

SNAPPY – provides lossless compression that is splittable. It is integrated into Hadoop Common and often used for database compression. Snappy compression ratio is around 2x. Compression speed is around 580MB/s and decompression is around 2020 MB/s.

LZ4 – provides lossless compression that is splittable unless combined with the 4MC library. It is used for general-purpose analysis. LZ4 compression ratio is around 2.5x. Compression speed is around 800MB/s and decompression speed is around 4220MB/s.

ZSTD – provides lossless compression that is splittable. It is not data type-specific and is designed for real-time compression. Zstd compression ratio is around 2.8x. Compression speed is around 530MB/s and decompression is around 1360MB/s.

From all the above four algorithms, LZ4 is the fastest and has higher decompression speed than SNAPPY as well which enables faster read, but it is not supported by impala. Snappy compression is supported in pretty much all the stack.

→ GZIP uses more CPU resources than SNAPPY or LZO but provides a higher compression ratio
→ SNAPPY / LZO are more for speed and is a magnitude faster than GZIP for read/decompression.
→ SNAPPY is significantly faster than LZO for decompression
→ BZIP2 compresses more effectively than GZIP but it's slower.

Optimize JSON performance

The JavaScript Object Notion (JSON) format is mostly used to serialize and transfer structured data for web applications. This means that much of the big data collected is stored in a JSON format. Unfortunately, working with JSON files in big data tools, can be slow because it is neither schema-ed nor strongly typed. *You can optimize JSON performance by storing the data in AVRO or PARQUET format.*

Parquet is a column-based format that is compressible and splittable. Parquet stores data in binary files with metadata. This structure enables tools, like Spark, to determine column names, data type, and compression and encodings without parsing the file. The order that Parquet stores metadata in enables fast, one-pass writing.

Parquet is most useful if you only need to access specific fields. Unfortunately, this format cannot be created from streaming data. It is often used for read-heavy workloads and complex analysis.

Avro is a row-based format that is splittable and compressible. Avro files have schema stored in a JSON format and data stored in a binary format. This structure reduces file size and maximizes efficiency.

Avro can be created from streaming data and is most useful if all fields in a dataset need to be accessed. It is typically used for write-heavy workloads because new rows can be added simply and quickly.

Options while storing data in HDFS

1. Store the file uncompressed.
2. Split files into chunks and compress each chunks
3. Use a compression format that supports splitting
4. Use a container file format that supports both compression and splitting. For ex: parquet and avro file formats which supports splitting along with snappy compression which has faster read speed.

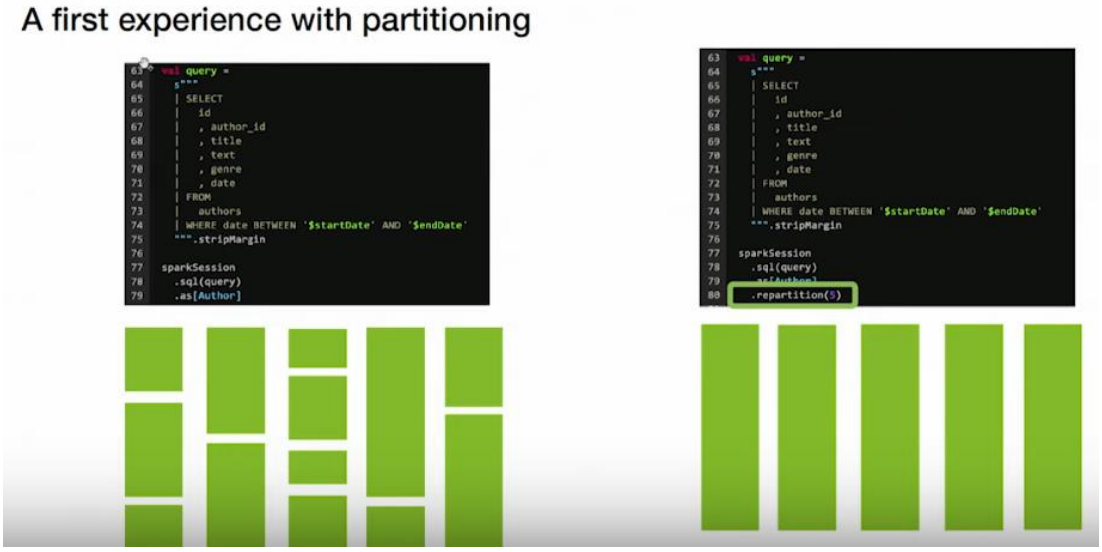
Note: It is always advisable to use compression formats which supports splitting, because there can be parallel map jobs running on each of the splits, but if they are not splittable then map job needs to collect the split parts in a HDFS cluster to a single node (if split parts are at different nodes) and then run the map job. In this case map job will be slower. BZIP2 map jobs are faster than GZIP due to this. But then, BZIP2 supports high compression so it takes more time to compress and decompress. But w.r.t. space utility, it's the best.

From Cloudera,

- Compression is not recommended if your data is already compressed (such as images in JPEG format). In fact, the resulting file can actually be larger than the original.
- GZIP compression uses more CPU resources than Snappy or LZO, but provides a higher compression ratio. GZip is often a good choice for cold data, which is accessed infrequently. Snappy or LZO are a better choice for hot data, which is accessed frequently.
- BZip2 can also produce more compression than GZip for some types of files, at the cost of some speed when compressing and decompressing. HBase does not support BZip2 compression.
- Snappy often performs better than LZO. It is worth running tests to see if you detect a significant difference.
- For MapReduce, if you need your compressed data to be splittable, BZip2, LZO, and Snappy formats are splittable, but GZip is not. Splittability is not relevant to HBase data.

Partitioning Strategies

Repartitioning is always a good practice for faster write and read as well.
Below diagram shows how the repartitioning enables evenly distributed partitions and this becomes really important for spark jobs to speed up and write data properly instead of getting stuck at some point which might happen if there are un-even partitions.



- ✓ .repartition() vs .coalesce()
- ✓ Custom partitioning is supported with the RDD API only (specifically through implicitly added PairRDDFunctions)
- ✓ Spark supports the HashPartitioner and RangePartitioner out of the box implementation
- ✓ One can create custom Partitioner to enable custom strategies in grouping data.

Repartitioning allows you to go both up and down in your partitions and it does a full data shuffle i.e. it will perform a lot of data movement around but guarantees some sort of evenness over a large dataset.
Coalesce allows you to go down only during the partitions and it tries to perform minimum amount of data shuffle/ movement and also tries to re-use the existing partitions. But only issue will be if data is very skewed you might end up with long running spark sessions and might create even more partitions.

HashPartitioner is more widely used. Below is a sample example to show how you can use HashPartitioner.

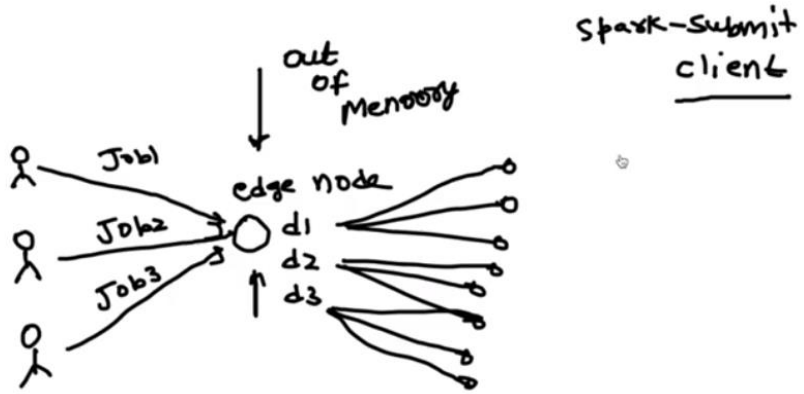
```
import org.apache.spark.sql.SparkSession
import org.apache.spark.HashPartitioner

object partitionerExample {

  def exploreData (implicit sparkSeeeion: SparkSession) = {
    val startDate="2019-02-01"
    val endDate="2019-02-08"

    val articles= Article.loadDataSetFromHive(startDate, endDate)
    val goalPartitions= 5

    articles.rdd
            .map{case(article) => (article.id, article)}
            .partitionBy (new HashPartitioner(goalPartitions))
            //...carry out more operations on article
  }
}
```

Spark Deployment YARN – Client VS Cluster Mode

YARN can operate both in client and cluster mode while submitting spark jobs in a clustered environment. There is an important difference between the two and this difference is important for scaling up data loading in production scenarios. In a typical scenario for a spark application, the driver program initiates the job processing by instantiating the configurations and context. The user will submit a spark job via an **edge node** of the cluster**.**

(**Edge nodes** are the interface between the Hadoop cluster and the outside network. For this reason, they're sometimes referred to as gateway nodes. Most commonly, edge nodes are used to run client applications and cluster administration tools.)

In a CLIENT mode, once the job is submitted the driver will be executed on the same edge node and then it will communicate with the cluster manager to distribute the jobs to the worker/executors. The executors can run their jobs at other nodes of the cluster. In this case driver program will utilize the resources of the edge node i.e. memory and CPU. Suppose there are more than one driver program running on the same client mode and on the same node, in that case all these driver programs will be utilizing the same memory of the same edge node. This will create lot of resource utilization on a single node and there will be **out of memory** issues if all the resources are exhausted. This is why it is always advisable to run spark jobs at cluster mode.

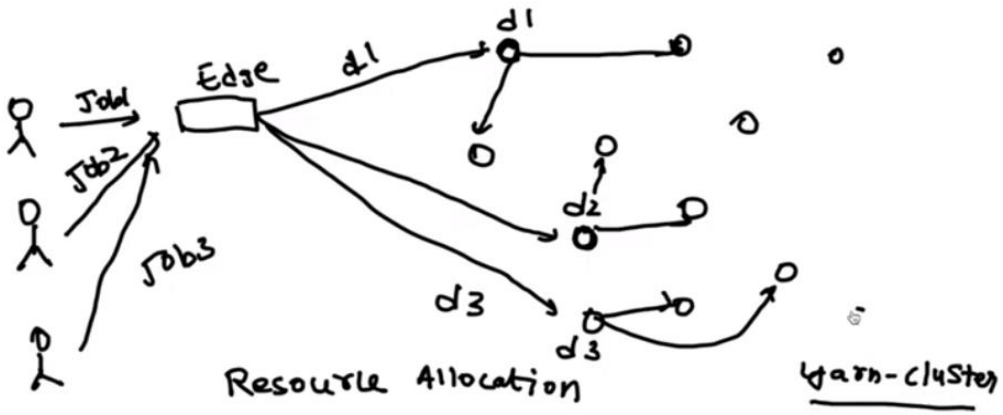


d1, d2, d3 are different driver programs

In a CLUSTER mode, when a user perform spark-submit on an edge node, the driver program doesn't run on the same edge node. Instead, spark will choose a different node which is free and has less resource utilization and run the driver program on that node. So, the driver node uses the resources from that particular node. Likewise, the driver program negotiates with cluster manager on worker nodes and elect executors to run on other nodes.

When there are more than one driver programs being submitted, spark will efficiently choose each of these driver programs to run on different nodes and then the worker nodes as well. In this way there are hardly such scenarios where you will be out of memory due to all the resource heavy spark submit jobs running on the same edge node.

Thus, in any production environment it is always advised to run spark-submit jobs with deployment mode as cluster mode so as not to affect other spark jobs and let spark utilize the cluster strength properly.



Spark Dynamic Memory Allocation

Dynamic allocation allows spark to dynamically scale the cluster resources allocated to your application based on the workload on the fly. When dynamic allocation is enabled and a spark application has a backlog of pending tasks, it can request executors. When application becomes idle, the executors are released and can be acquired by other applications. The dynamic memory management is more from the **spark driver side**.
In Cloudera platform (CDP) dynamic allocation is enabled by default.

In spark, **spark.dynamicAllocation.enabled** needs to be set to **true** because by default it's set to **false.**
This requires spark shuffle services**: spark.shuffle.service.enabled** to be set true as spark application is running on YARN.

spark.dynamicAllocation.enabled= true
spark.dynamicAllocation.executorIdleTimeout= 2m
spark.dynamicAllocation.minExecutors= 1
spark.dynamicAllocation.maxExecutors= 2000

These options can be configured for any spark application in three ways:

1. From Spark Submit with **--conf <prop_name>=<prop_value>**

```
spark-submit \
--cluster yarn \
--mode cluster \
--driver-memory 2G \
--num-executors 10 \
--executor-cores 5 \  (this is for parallelism)
--executor-memory 2G \
--conf.spark.dynamicAllocation.minExecutors= 5 \
--conf.spark.dynamicAllocation.maxExecutors= 30 \
--conf.spark.dynamicAllocation.initialExecutors= 10 \   [same as num-executors=10]
--class com.sql.spark.jdbc.SparkDFtoOracle2 \
Spark-hive-sql-Dataframe-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

2. Inside Spark Program with **SparkConf**

```
val conf:SparkConf = new SparkConf();
conf.set("spark.dynamicAllocation.minExecutors","5");
conf.set("spark.dynamicAllocation.maxExecutors", "30");
conf.set("spark.dynamicAllocation.initialExecutors", "10");
…..

val conf = new SparkConf().setAppName("Spark dynamic allocation demo")
                          .set("spark.dynamicAllocation.enabled", "true")
                          .set("spark.shuffle.service.enabled", "true")
                          .set("spark.dynamicAllocation.initialExecutors", "1")
                          .set("spark.dynamicAllocation.executorIdleTimeout", "120s")
                          .set("spark.dynamicAllocation.schedulerBacklogTimeout", "1s")
                          .set("spark.executor.cores", "1")
                          .set("spark.executor.memory", "512m")
```

3. **spark-defaults.conf** usually located in $SPARK_HOME/conf/
   Place the same configurations in spark-defaults.conf to apply for all spark applications if no configuration is passed from command-line as well as code.

Better Fetch Failure Handling

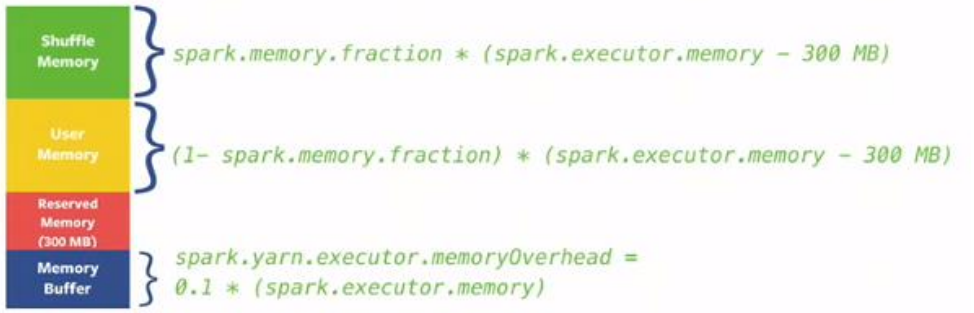
When the cluster size increases and multiple jobs are running at real time, there are chances of fetch failures. In case of a fetch failure in a spark job the driver does not run all the map tasks needed. Instead it keeps retrying the same one which is a bug in spark fetch failure scenario. It not only increases the latency of the job but even may lead to a job failure as **default maximum retries in fetch failure = 4**.
Configure max no of fetch failures: **spark.max.fetch.failures.per.stage = 10**
(SPARK-19753, 20163)

Scaling Spark Driver

Tune RPC server threads:
→ Frequent driver OOM (Out Of Memory) when running many tasks in parallel
→ Huge backlog of RPC requests built on Netty server of the driver
→ Increase RPC server thread to fix OOM: **spark.rpc.io.serverThreads=64** (by default it's 8)

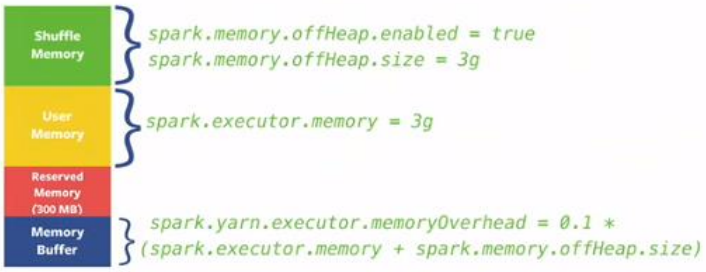
Executor Memory Layout

Executor memory layout contains: shuffle memory, user memory, reserved memory and memory buffer. The shuffle and user memory are the important ones and should be tuned properly. The map tasks takes place inside the shuffle memory and in theory if you configure it properly during running big map jobs it will prevent spilling data into disk and this will increase the speed of your job.
To understand, how much user memory to allocate depends completely on the application.



By default 40% of the executor memory is allocated to user memory. If you don't need so much of user memory, you can do a **trade-off between shuffle and user memory** and allocate more shuffle memory to avoid spill to disk.

Another way you can tune the performance by **enable off-heap memory** in which the heap is assigned as a part of the shuffle memory and not managed by JVM. This enables you to allocate the shuffle data structure of heap in native memory. This means they are not allocated and managed by JVM memory manager and they are not subjected to garbage collection. So, by enabling off heap memory, you can actually avoid Garbage Collection overhead which will help you speed up your job.

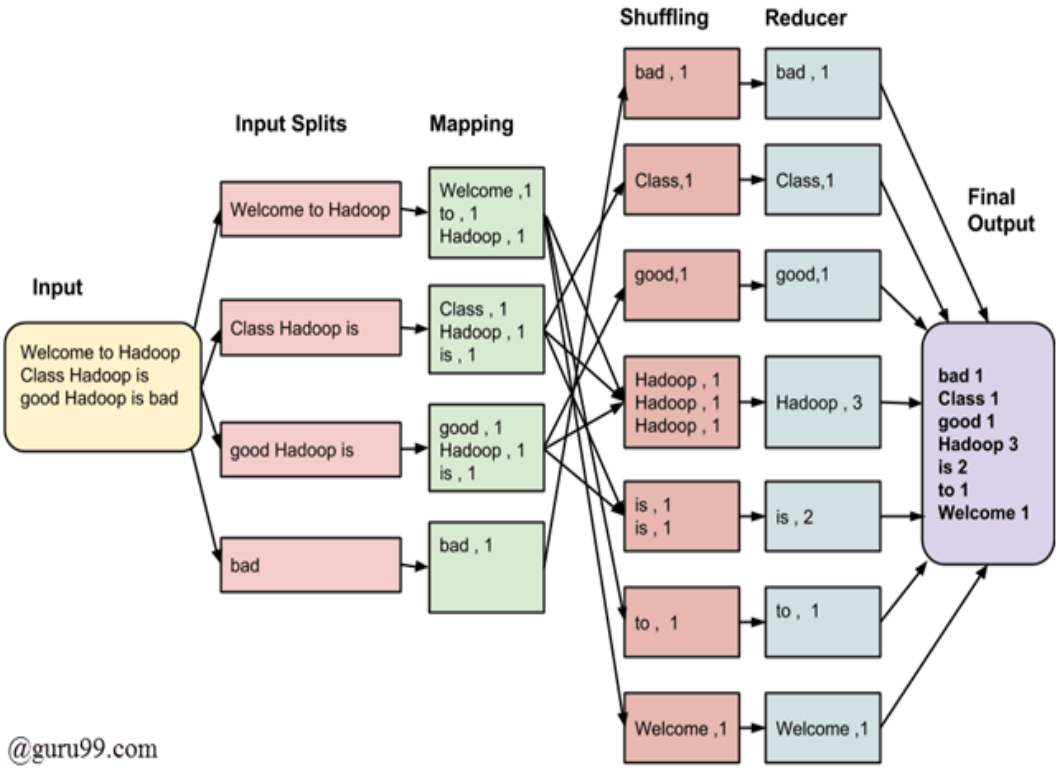


MAP REDUCE

The whole process goes through four phases of execution namely, splitting, mapping, shuffling, and reducing.

Consider you have following input data for your Map Reduce Program

```
Welcome to Hadoop Class
Hadoop is good
Hadoop is bad
```



Input Splits: An input to a MapReduce job is divided into fixed-size pieces called **input splits** Input split is a chunk of the input that is consumed by a single map

Mapping This is the very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, a job of mapping phase is to count a number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

Shuffling This phase consumes the output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, the same words are clubed together along with their respective frequency.

Reducing In this phase, output values from the Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

In Details:

- One map task is created for each split which then executes map function for each record in the split.

- It is always beneficial to have multiple splits because the time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better to load balanced since we are processing the splits in parallel.

- However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
- For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 64 MB, by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
- Map output is intermediate output which is processed by reduce tasks to produce the final output.
- Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
- In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
- Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine, the output is merged and then passed to the user-defined reduce function.
- Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

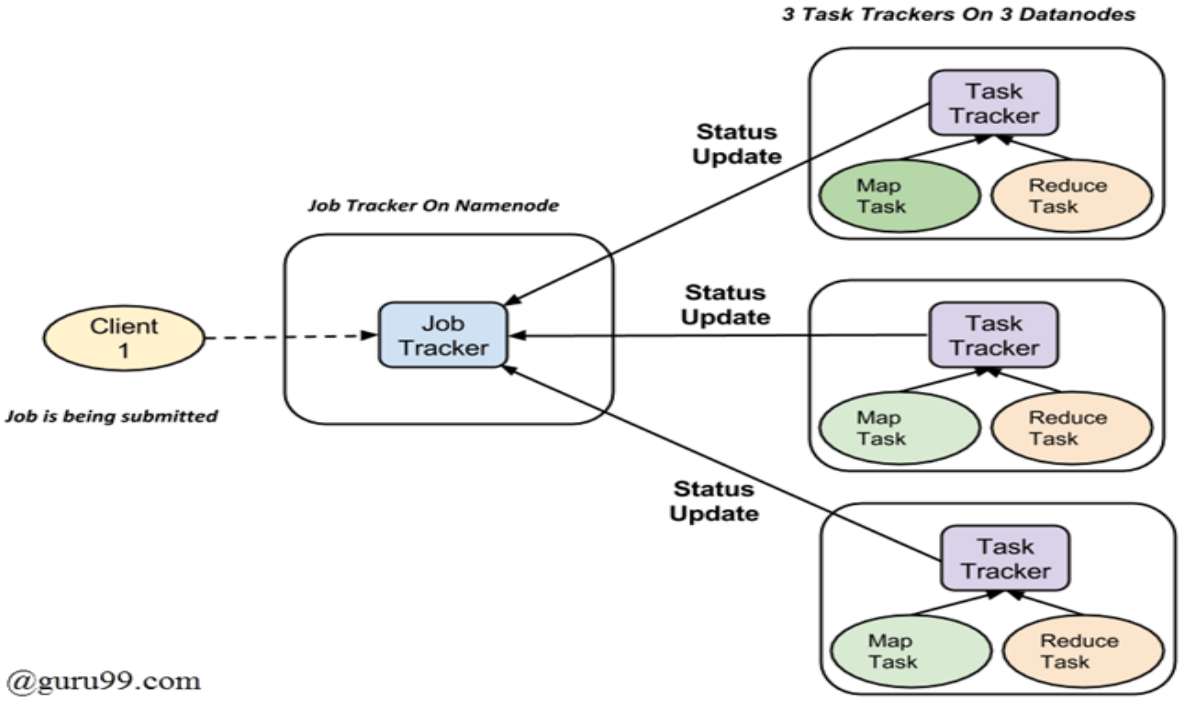
How Map Reduce Organizes work?

Hadoop divides the job into tasks. There are two types of tasks:

1. Map tasks (Splits & Mapping)
2. Reduce tasks (Shuffling, Reducing)

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called:

1. **Jobtracker:** Acts like a master (responsible for complete execution of submitted job)
2. **Multiple Task Trackers:** Acts like slaves, each of them performing the job



- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.
- In addition, task tracker periodically sends 'heartbeat' signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

Spark VS map-reduce:
1. Spark is near real-time while map reduce is batch processing
2. Spark is much faster almost 100 times than map-reduce.
3. Spark is easy to code/program while map reduce is difficult
4. Ensures low latency by caching the interim results across its distributed memory.
5. Spark has graph processing while map reduce don't.
6. Spark is memory based- in memory while map reduce is disk based. This affects the processing speed.

Key features of spark:
1. In-memory computing. Speed is much better for repetitive calculation.
2. Can be programmed using python, java, scala etc.
3. Calculates the DAC cycle to get the optimal path to provide result. Lazy execution.
4. Gels with hadoop ecosystem very well.
5. Real time spark streaming.
6. Has machine language algorithms already implemented (MLLIB)
7. Graphx supports graph theory.

KAFKA – For Basics

CREATING MY FIRST KAFKA TOPIC
kafka-topics --create \
--zookeeper bda1node01.fgb.ae:2181, bda1node02.fgb.ae:2181 \
--replication-factor 3 \
--partitions 1 \
--topic MyFirstKafkaTopic

LISTING ALL KAFKA TOPICS
kafka-topics.sh --list \
--zookeeper bda1node01.fgb.ae:2181, bda1node02.fgb.ae:2181 \
--topic MyFirstKafkaTopic (if you want to list one topic provide topic parameter)

CREATING A KAFKA PRODUCER
kafka-console-producer.sh \
--broker-list bda1node01.fgb.ae:6667, bda1node02.fgb.ae:6667\
--topic MyFirstKafkaTopic

CREATING A KAFKA CONSUMER
kafka-console-producer.sh \
--bootstrap-server bda1node01.fgb.ae:6667, bda1node02.fgb.ae:6667\
--topic MyFirstKafkaTopic \

--from-beginning

KAFKA TOPICS ANATOMY:

If the replication factor is more than one, for each topic there will be more than one copies maintained for a topic. This is for high availability.If any nodes goes down, kafka topics should continue with the other replicated nodes.

Partition is for scalability
Replication is for high availability.

Inside the topics folder under kafka/logs folder, there will be 2 files:
1. index file
2. log file
The log file will be updated whenever we publish messages.
If we mention partition index, then once the above files are written into topic_name-partition_no folder.
For ex: in the above case we have one partition, so the there will be 1 partition and the topic folder is created as MyFirstKafkaTopic-0
If there are three partitions, then there will be three such folders
MyFirstKafkaTopic-0
MyFirstKafkaTopic-1
MyFirstKafkaTopic-2

A topic can have multiple partitions. Each partition will have some messages which will be attended as they receive based on some hash algorithm. Then there will be the subscriber group.

Publisher group will publish the messages and they will append the new messages to the existing partitions. The publisher will keep track of where it appended last time and it will keep track and will append new messages from the next.

Subscriber will start from beginning. It will start reading the messages from left. There will be a group of subscribers who will consume the messages and the entire group will keep track of the offset. If there are new messages the subscriber can start reading them as and when they are ready.
There can be a subscriber group with multiple subscribers reading the messages due to scalanility purpose.

FLUME VS KAFKA
Flume is not as reliable as KAFKA
Major challange with flume is scalability

DISADVANTAGE OF KAFKA:
If the application is already running and if it is logging the messages, still kafka can't be used to get data from logs. While flume can do that. So we can use applications like flume in that case along with kafka. Flume has the capability to get messages from log files without making changes to code. But with kafka, we have to refactor the code so that messages can be published to topics. So existing applications which are already running, making changes to read logs using kafka is big. So in such cases, we use flume to read the logs and flume sinks to kafka. Thus messages are pushed to kafka from flume and we have subscribers using up the messages to do whatever required post that.
For new applications, we do everything using Kafka which is much reliable.

https://www.golinuxcloud.com/run-shell-scripts-in-parallel-collect-exit-status-process/
https://www.youtube.com/watch?v=18CDJm9OVjQ